# 01【熟悉】工作流的概念

## 生活中常见的工作流

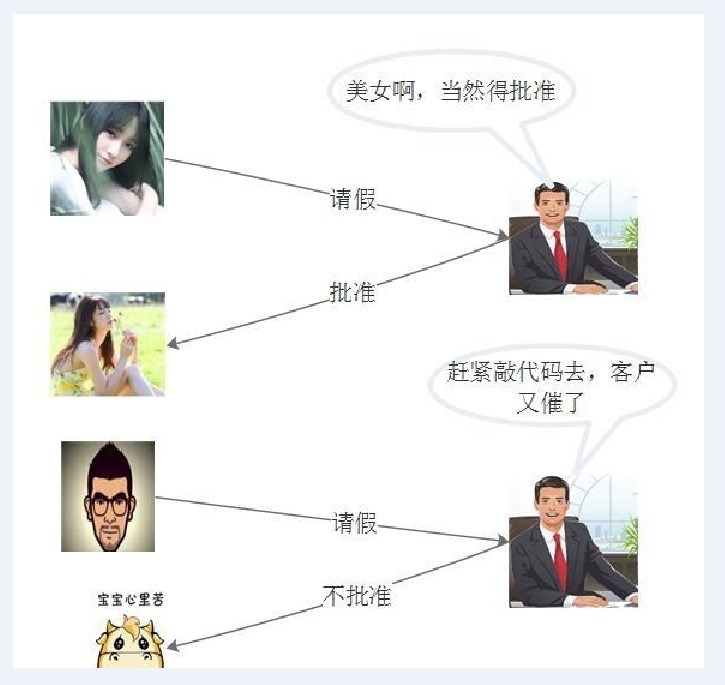
1，请假

2，报销

3，申请转武汉户口

4，购物

## 概述

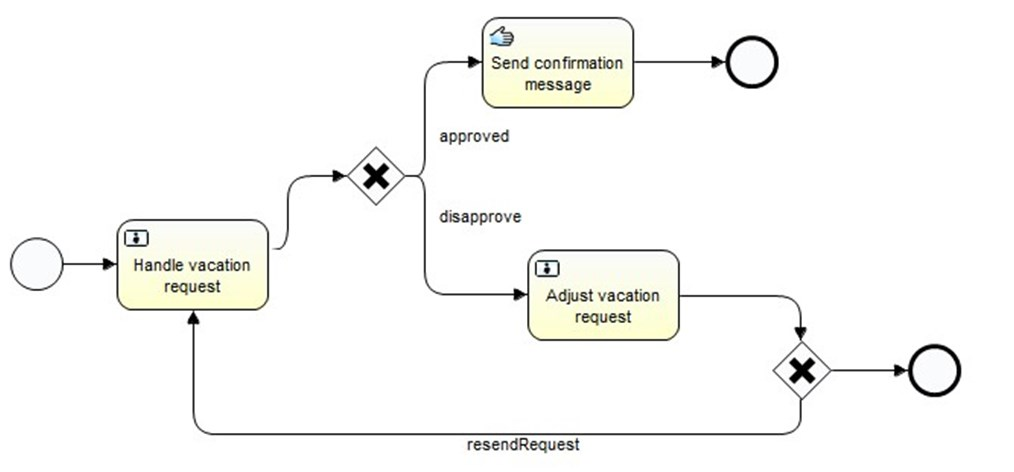


|  |
| --- |
| 1)假设：这两张图就是请假流程图  2)图的组成部分：  A.人物：美女 程序员 领导  B.事件（动作）：请假、批准、不批准  工作流(Workflow)，就是“业务过程的部分或整体在计算机应用环境下的自动化”，它主要解决的是“使在多个参与者之间按照某种预定义的规则传递文档、信息或任务的过程自动进行，从而实现某个预期的业务目标，或者促使此目标的实现”。  工作流管理系统(Workflow Management System, WfMS)是一个软件系统，它完成工作量的定义和管理，并按照在系统中预先定义好的工作流规则进行工作流实例的执行。工作流管理系统不是企业的业务系统，而是为企业的业务系统的运行提供了一个软件的支撑环境。  工作流管理联盟(WfMC，Workflow Management Coalition)给出的关于工作流管理系统的定义是：工作流管理系统是一个软件系统，它通过执行经过计算的流程定义去支持一批专门设定的业务流程。工作流管理系统被用来定义、管理、和执行工作流程。  工作流管理系统的目标：管理工作的流程以确保工作在正确的时间被期望的人员所执行——在自动化进行的业务过程中插入人工的执行和干预。 |

# 02【熟悉】Activiti介绍

## 1，概述

Activiti5是由Alfresco软件在2010年5月17日发布的业务流程管理（BPM）框架，它是覆盖了业务流程管理、工作流、服务协作等领域的一个开源的、灵活的、易扩展的可执行流程语言框架。Activiti基于Apache许可的开源BPM平台，创始人Tom Baeyens是JBoss jBPM的项目架构师，它特色是提供了eclipse插件，开发人员可以通过插件直接绘画出业务



## 2，工作流引擎

ProcessEngine对象，这是Activiti工作的核心。负责生成流程运行时的各种实例及数据、监控和管理流程的运行。

## 3，BPMN

业务流程建模与标注（Business Process Model and Notation，BPMN) ，描述流程的基本符号，包括这些图元如何组合成一个业务流程图（Business Process Diagram）

## 4，数据库（先看再说）

Activiti数据库支持：

Activiti的后台是有数据库的支持，所有的表都以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\_ID\_\*: 'ID'表示identity。 这些表包含身份信息，比如用户，组等等。

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\_\*: 通用数据， 用于不同场景下，如存放资源文件。

**1：资源库流程规则表**

1) act\_re\_deployment 部署信息表

~~2) act\_re\_model 流程设计模型部署表~~

3) act\_re\_procdef 流程定义数据表

**2：运行时数据库表**

1) act\_ru\_execution 运行时流程执行实例表

2) act\_ru\_identitylink 运行时流程人员表，主要存储任务节点与参与者的相关信息

3) act\_ru\_task 运行时任务节点表

4) act\_ru\_variable 运行时流程变量数据表

**3：历史数据库表**

1) act\_hi\_actinst 历史节点表

2) act\_hi\_attachment 历史附件表

3) act\_hi\_comment 历史意见表

4) act\_hi\_identitylink 历史流程人员表

5) act\_hi\_detail 历史详情表，提供历史变量的查询

6) act\_hi\_procinst 历史流程实例表

7) act\_hi\_taskinst 历史任务实例表

8) act\_hi\_varinst 历史变量表

**4：组织机构表**

1) act\_id\_group 用户组信息表

2) act\_id\_info 用户扩展信息表

3) act\_id\_membership 用户与用户组对应信息表

4) act\_id\_user 用户信息表

**这四张表很常见，基本的组织机构管理，关于用户认证方面建议还是自己开发一套，组件自带的功能太简单，使用中有很多需求难以满足**

**5：通用数据表**

1) act\_ge\_bytearray 二进制数据表

2) act\_ge\_property 属性数据表存储整个流程引擎级别的数据,初始化表结构时，会默认插入三条记录，

# 03【掌握】准备工作+插件安装

## 1，activiti5 软件环境

1)JDK1.6或者更高版本

2)支持的数据库有：h2, mysql, oracle, postgres, mssql, db2等。

3)支持activiti5运行的jar包 或者maven库

4)开发环境为Eclipse4.1或者以上版本,sts最新版本

## 2，相关资源下载

1)JDK可以到sun的官网下载

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

2)数据库，例如：mysql可以在官网上下载。http://www.mysql.com

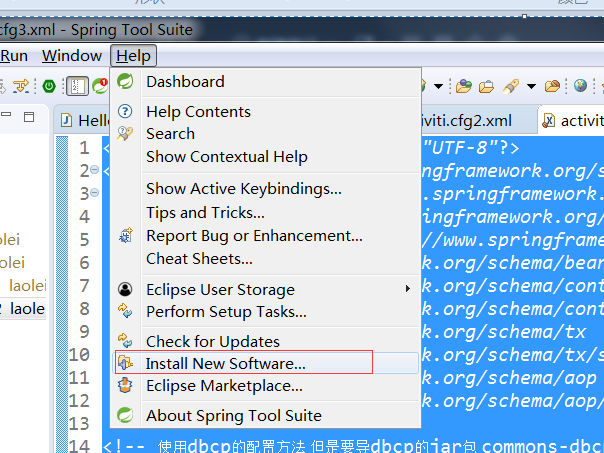
3)activiti也可以到Activiti官方网站下载得到。http://activiti.org/download.html

4) Eclipse4.2或者sts3.9.4也可以到相应的网站上获得下载。http://spring.io/tools/sts

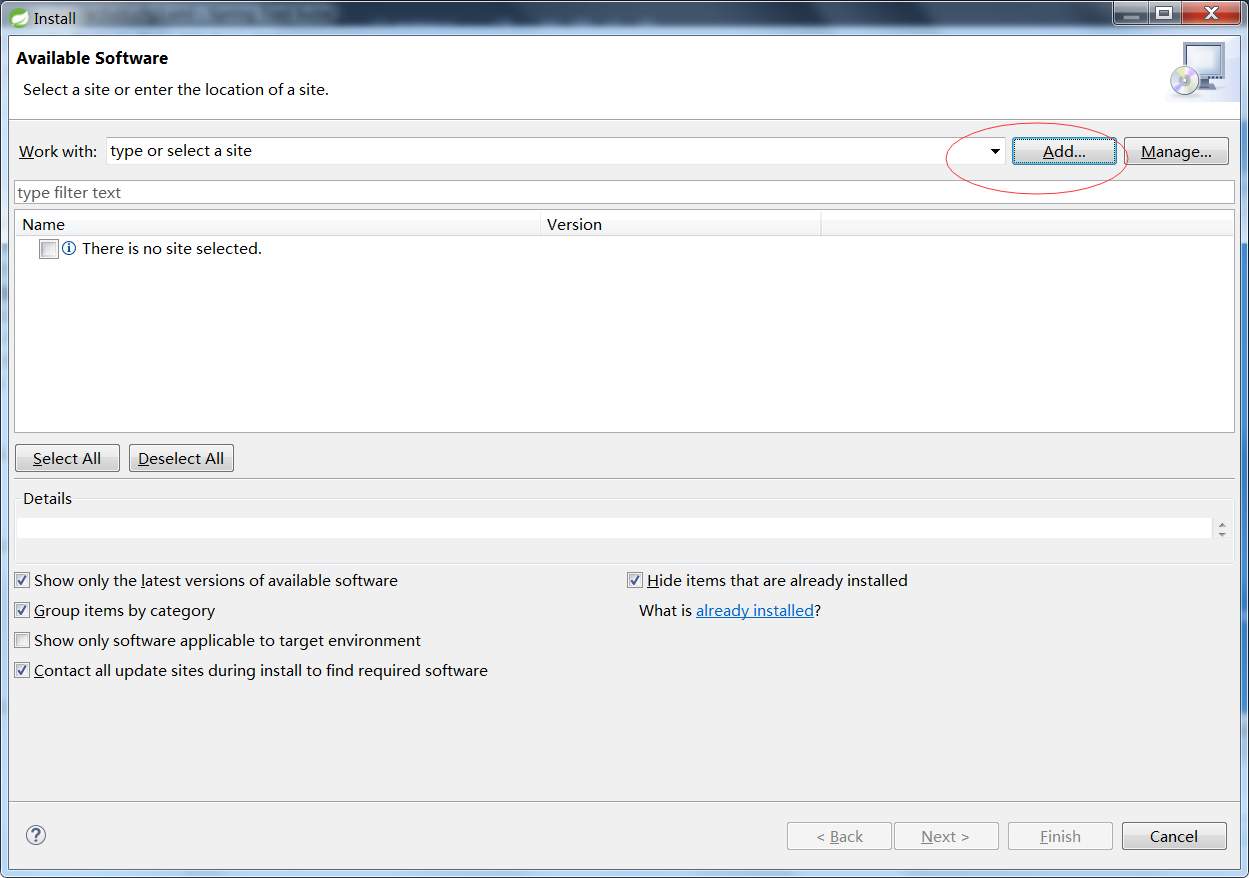
## 3，安装流程设计器(eclipse插件)【安装方式一】

在有网络的情况下，安装流程设计器步骤如下：

### 1)打开 Help -> Install New Software. 在如下面板中



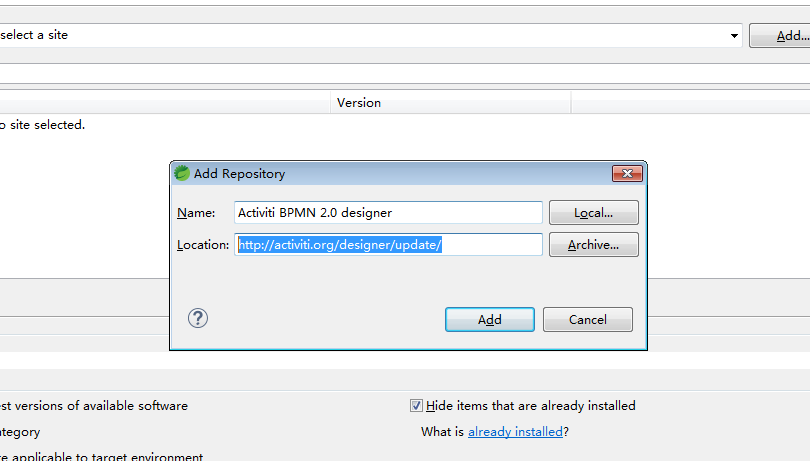
### 2)在如下Install界面板中，点击Add按钮：



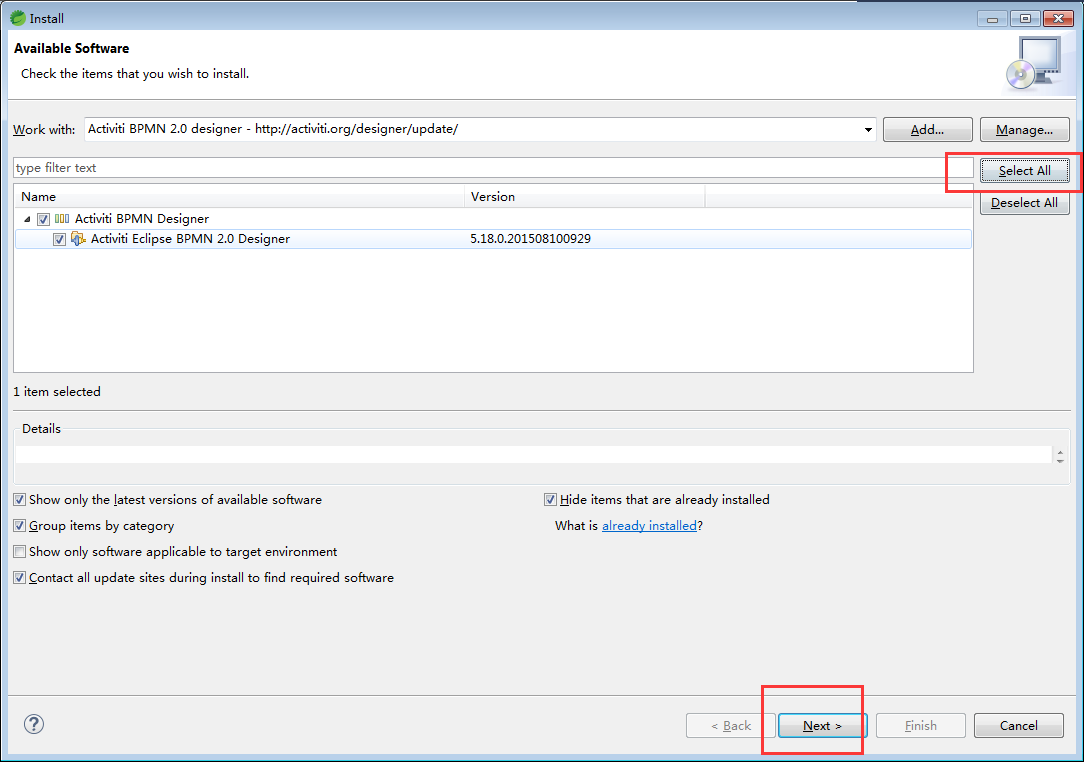
### 3)然后填入下列字段s

Name: Activiti BPMN 2.0 designer

Location: http://activiti.org/designer/update/



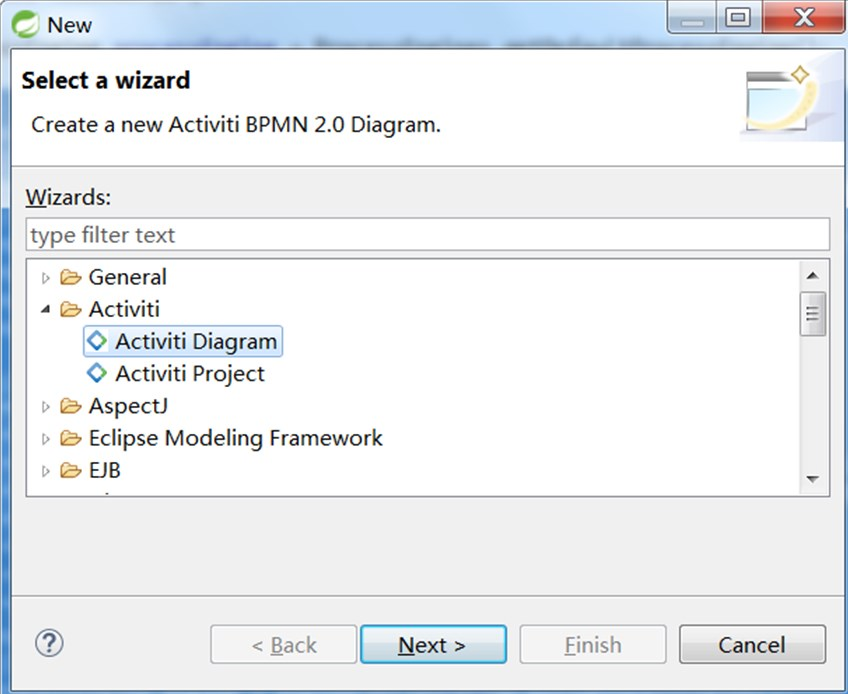
### 4)回到Install界面，在面板正中列表中把所有展示出来的项目都勾上：



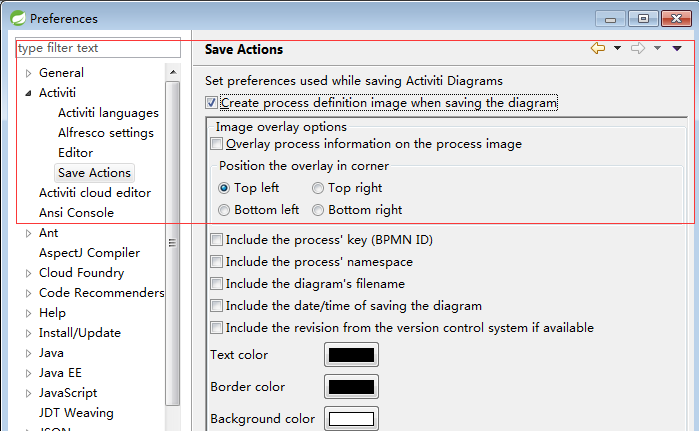
### 5)点击复选框

在Detail部分记得选中 "Contact all updates sites.." , 因为它会检查所有当前安装所需要的插件并可以被Eclipse下载.

### 6)安装完以后，点击新建工程new->Other…打开面板，如果看到下图内容：



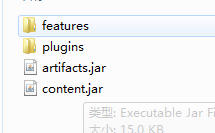
### 6，对流程设计器的使用说明

虽然流程引擎在单独部署bpmn文件时会自动生成图片，但在实际开发过程中，自动生成的图片会导致和BPMN中的坐标有出入，在实际项目中展示流程当前位置图会有问题。

所在完成以上配置后，会由我们自己来管理流程图片。在发布流程时把流程规则文件和流程图片一起上传就行了。

## 4，安装流程设计器(eclipse插件)【安装方式二】





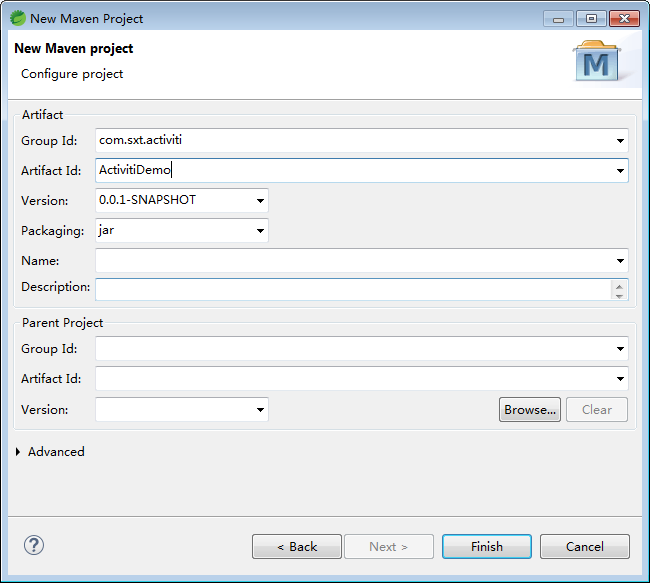
复制到sts里面的相关目录

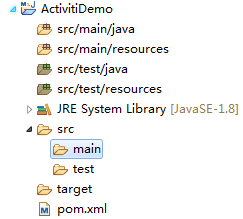
## 5，安装SQLyog - 64 bit



# 04【掌握】Activiti5开发环境

## 创建项目





## 配置pom.xml

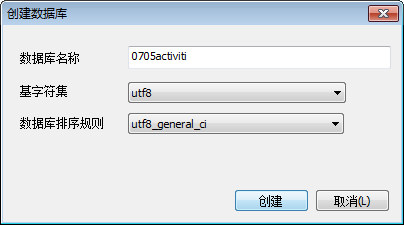
|  |
| --- |
| <project xmlns=*"http://maven.apache.org/POM/4.0.0"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>  <modelVersion>4.0.0</modelVersion>  <groupId>com.sxt.activiti</groupId>  <artifactId>ActivitiDemo</artifactId>  <version>0.0.1-SNAPSHOT</version>  <!-- 配置版本 -->  <properties>  <spring.version>4.3.17.RELEASE</spring.version>  <mysql.version>5.1.39</mysql.version>  <!-- 注意只能使用2.0以下的版本 -->  <activiti.version>5.22.0</activiti.version>  <mybatis.version>3.4.6</mybatis.version>  <!-- 注意只能使用2.0以下的版本 -->  <log4j.version>1.2.17</log4j.version>  </properties>  <dependencies>  <!-- activiti的依赖 -->  <dependency>  <groupId>org.activiti</groupId>  <artifactId>activiti-engine</artifactId>  <version>${activiti.version}</version>  </dependency>  <!-- ssm集成的时候使用 -->  <dependency>  <groupId>org.activiti</groupId>  <artifactId>activiti-spring</artifactId>  <version>${activiti.version}</version>  </dependency>  <!-- mysql驱动 -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>${mysql.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aspects</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aop</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-beans</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-expression</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jdbc</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-tx</artifactId>  <version>${spring.version}</version>  </dependency>  <!-- mybatis -->  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>${mybatis.version}</version>  </dependency>  <!-- log4j -->  <dependency>  <groupId>log4j</groupId>  <artifactId>log4j</artifactId>  <version>${log4j.version}</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>1.7.25</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-simple</artifactId>  <version>1.7.25</version>  </dependency>  </dependencies>  <build>  <plugins>  <!-- 配置编译的jdk版本 -->  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.7.0</version>  <configuration>  <!-- 指定source和target的版本 -->  <source>1.8</source>  <target>1.8</target>  </configuration>  </plugin>  </plugins>  </build>  </project> |

## 配置log4j.properties

|  |
| --- |
| log4j.rootLogger=INFO, stdout  # Console Appender  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern= %d{hh:mm:ss,SSS} [%t] %-5p %c %x - %m%n  # Custom tweaks  log4j.logger.com.codahale.metrics=WARN  log4j.logger.com.ryantenney=WARN  log4j.logger.com.zaxxer=WARN  log4j.logger.org.apache=WARN  log4j.logger.org.hibernate=WARN  log4j.logger.org.hibernate.engine.internal=WARN  log4j.logger.org.hibernate.validator=WARN  log4j.logger.org.springframework=WARN  log4j.logger.org.springframework.web=WARN  log4j.logger.org.springframework.security=WARN |

## 初始化数据

### 创建数据库



### 创建类去初始化表方式一

|  |
| --- |
| @Test  **public** **void** initTables() {  //创建数据源  DriverManagerDataSource dataSource=**new** DriverManagerDataSource();  dataSource.setDriverClassName("com.mysql.jdbc.Driver");  dataSource.setUrl("jdbc:mysql://127.0.0.1:3306/0705activiti");  dataSource.setUsername("root");  dataSource.setPassword("123456");    // 创建流程引擎的配置  ProcessEngineConfiguration configuration = ProcessEngineConfiguration  .*createStandaloneProcessEngineConfiguration*();  configuration.setJdbcDriver("com.mysql.jdbc.Driver");  configuration.setJdbcUrl("jdbc:mysql://127.0.0.1:3306/0705activiti");  configuration.setJdbcUsername("root");  configuration.setJdbcPassword("123456");  // configuration.setDataSource(dataSource);  /\*\*  \* ProcessEngineConfiguration.DB\_SCHEMA\_UPDATE\_FALSE 如果数据库里面没有activit的表，也不会创建  \* ProcessEngineConfiguration.DB\_SCHEMA\_UPDATE\_CREATE\_DROP 创建表，使用完之后删除  \* ProcessEngineConfiguration.DB\_SCHEMA\_UPDATE\_TRUE 如果数据库里面没有表，就创建  \*  \* dorp-create 代表如果数据库里面有表，那么先删除再创建  \*  \*/  //配置表的初始化的方式  configuration.setDatabaseSchemaUpdate(ProcessEngineConfiguration.***DB\_SCHEMA\_UPDATE\_TRUE***);    //得到流程引擎  ProcessEngine processEngine=configuration.buildProcessEngine();  System.***out***.println(processEngine);  } |

### 创建类去初始化表方式二

#### 创建activiti.cfg.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"* xmlns:tx=*"http://www.springframework.org/schema/tx"*  xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.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"*>      <bean id=*"processEngineConfiguration"* class=*"org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration"*>  <property name=*"jdbcDriver"* value=*"com.mysql.jdbc.Driver"*></property>  <property name=*"jdbcUrl"* value=*"jdbc:mysql://localhost:3306/0705activiti"*></property>  <property name=*"jdbcUsername"* value=*"root"*></property>  <property name=*"jdbcPassword"* value=*"123456"*></property>  <!--  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。  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。  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）。  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"*></property>  </bean>  </beans> |

#### 创建方法

|  |
| --- |
| @Test  **public** **void** intiTables2() {  ProcessEngineConfiguration configuration = ProcessEngineConfiguration  .*createProcessEngineConfigurationFromResource*("/activiti.cfg.xmls");  // 得到流程引擎  ProcessEngine processEngine = configuration.buildProcessEngine();  System.***out***.println(processEngine);  } |

### 创建类去初始化表方式三

|  |
| --- |
| @Test  **public** **void** intiTables3() {  //必须创建activiti.cfg.xml 并配置好数据库的信息  ProcessEngine processEngine=ProcessEngines.*getDefaultProcessEngine*();  System.***out***.println(processEngine);  } |

# 05【掌握】activiti.cfg.xml的说明

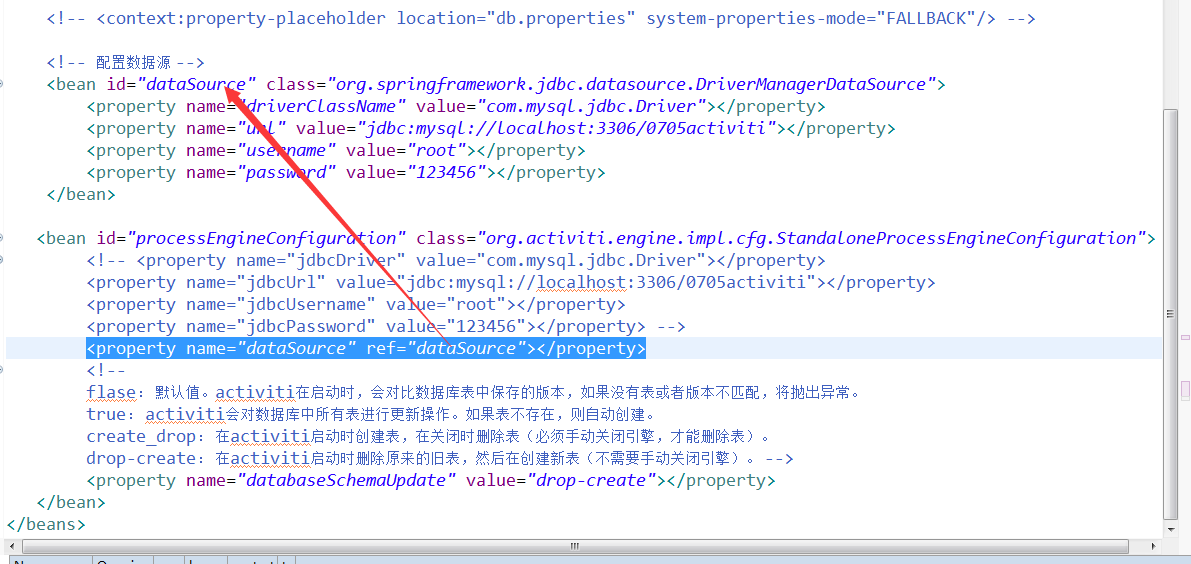
## 1，Activiti核心配置文件，配置流程引擎创建工具的基本参数和数据库连接池参数。

|  |
| --- |
| **1，Activiti核心配置文件，配置流程引擎创建工具的基本参数和数据库连接池参数。**  定义数据库配置参数：  1，jdbcUrl: 数据库的JDBC URL。  2，jdbcDriver: 对应不同数据库类型的驱动。  3，jdbcUsername: 连接数据库的用户名。  4，jdbcPassword: 连接数据库的密码。  基于JDBC参数配置的数据库连接 会使用默认的MyBatis连接池。 下面的参数可以用来配置连接池（来自MyBatis参数）：  1，jdbcMaxActiveConnections: 连接池中处于被使用状态的连接的最大值。默认为10。  2，jdbcMaxIdleConnections: 连接池中处于空闲状态的连接的最大值。  3，jdbcMaxCheckoutTime: 连接被取出使用的最长时间，超过时间会被强制回收。 默认为20000（20秒）。  4，jdbcMaxWaitTime: 这是一个底层配置，让连接池可以在长时间无法获得连接时， 打印一条日志，并重新尝试获取一个连接。（避免因为错误配置导致沉默的操作失败）。 默认为20000（20秒）。  5，dataSource 可以配置其它数据源  6，databaseSchemaUpdate  设置数据库表的创建策略 |

## 2，示例数据库配置：



## 3，使用其它数据源：



# 06【掌握】核心API

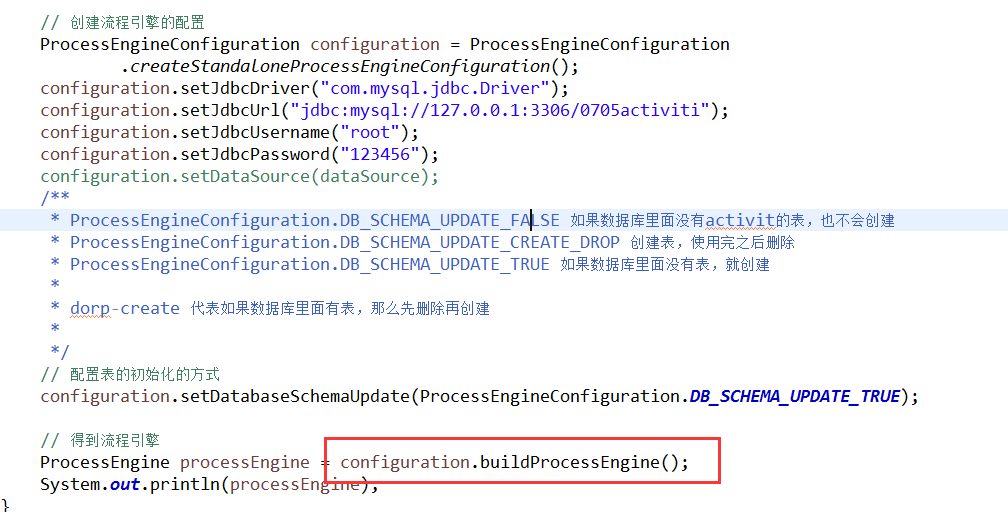
## 1，ProcessEngine

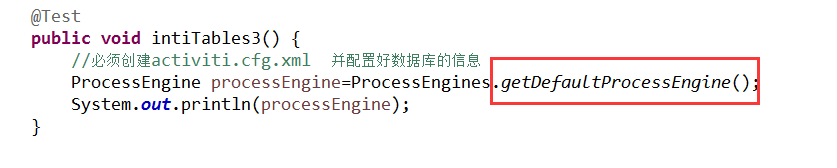
说明：

1)在Activiti中最核心的类，其他的类都是由他而来。

2)产生方式。

### 1，ProcessEngine获取方式





|  |
| --- |
| @Test  **public** **void** intiTables3() {  //必须创建activiti.cfg.xml 并配置好数据库的信息  ProcessEngine processEngine=ProcessEngines.*getDefaultProcessEngine*();  System.***out***.println(processEngine);  //流程图的部署 修改 删除的服务器 act\_ge\_bytearray, act\_re\_deployment, act\_re\_model, act\_re\_procdef  RepositoryService repositoryService = processEngine.getRepositoryService();  //流程的运行 act\_ru\_event\_subscr, act\_ru\_execution, act\_ru\_identitylink, act\_ru\_job, act\_ru\_task, act\_ru\_variable  RuntimeService runtimeService = processEngine.getRuntimeService();  TaskService taskService = processEngine.getTaskService();  //查询历史记录的服务器act\_hi\_actinst, act\_hi\_attachment, act\_hi\_comment, act\_hi\_detail, act\_hi\_identitylink, act\_hi\_procinst, act\_hi\_taskinst, act\_hi\_varinst  HistoryService historyService = processEngine.getHistoryService();  //页面表单的服务器[了解]  FormService formService = processEngine.getFormService();  //对工作流的用户管理的表的操作act\_id\_group, act\_id\_info, act\_id\_membership, act\_id\_user  IdentityService identityService = processEngine.getIdentityService();  //管理器  ManagementService managementService = processEngine.getManagementService();  } |

## 2，RepositoryService

是Activiti的仓库服务类。所谓的仓库指流程定义文档的两个文件，bpmn文件和流程图片。

1)产生方式

C:\Users\LJH\Documents\My Knowledge\temp\ad8f1fde-91a2-4ee5-a29b-b75abfa0ca56\128\index_files\9a9087ba-f498-4b08-a157-8f44ef40aadf.jpg

2)可以产生DeploymentBuilder，用来定义流程部署的相关参数

C:\Users\LJH\Documents\My Knowledge\temp\ad8f1fde-91a2-4ee5-a29b-b75abfa0ca56\128\index_files\e12ccc86-a9a8-4fcd-b7ad-1522ee51f9c5.jpg

3)删除流程定义

C:\Users\LJH\Documents\My Knowledge\temp\ad8f1fde-91a2-4ee5-a29b-b75abfa0ca56\128\index_files\c8465cd1-5a55-4a8c-a85e-460592c3e238.jpg

## 3，RuntimeService

是activiti的流程执行服务类。可以从这个服务类中获取很多关于流程执行相关的信息。

## 4，TaskService

是activiti的任务服务类。可以从这个类中获取任务的信息。

## 5，HistoryService

是activiti的查询历史信息的类。在一个流程执行完成后，这个对象为我们提供查询历史信息。

## 6，ProcessDefinition act\_re\_procdef

流程定义类。可以从这里获得资源文件等。当流程图被部署之后，查询出来的数据就是流程定义的数据

## 7，ProcessInstance act\_ru\_execution

代表流程定义的执行实例。如范冰冰请了一天的假，她就必须发出一个流程实例的申请。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。流程实例就表示一个流程从开始到结束的最大的流程分支，即一个流程中流程实例只有一个。

## 8，Execution act\_ru\_execution

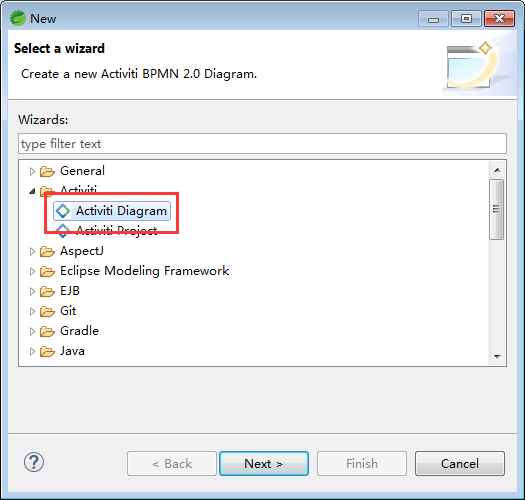
Activiti用这个对象去描述流程执行的每一个节点。在没有并发的情况下，Execution就是同ProcessInstance。流程按照流程定义的规则执行一次的过程，就可以表示执行对象Execution。

如图为ProcessInstance的源代码，

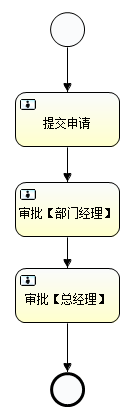
## 8，TaskInstance 任务实例act\_ru\_task

# 07【掌握】HelloWorld程序(模拟流程的执行)

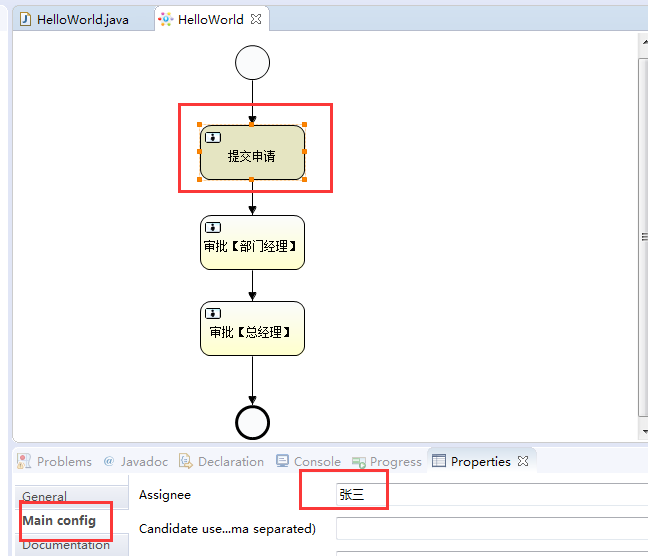
## 画流程图

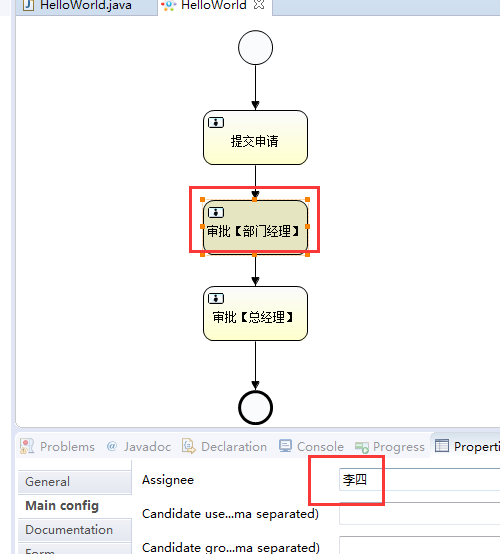


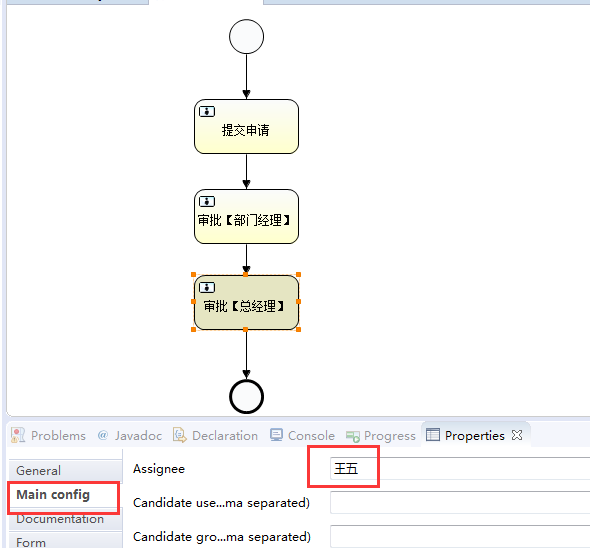
### 画流程图



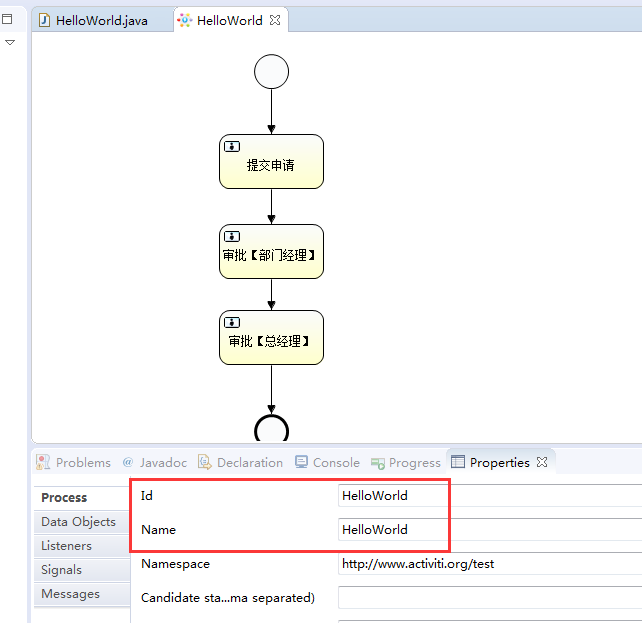
### 设置任务的办理人







### 设置流程的ID和NAME



## 部署流程定义



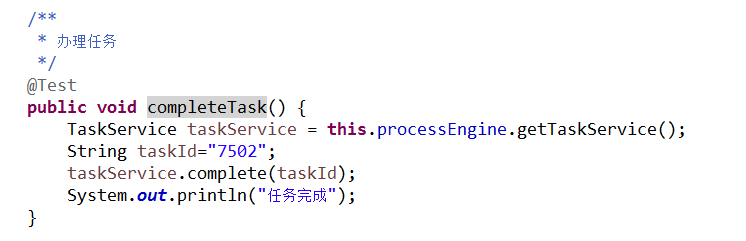
## 启动流程



## 查询任务



## 办理任务



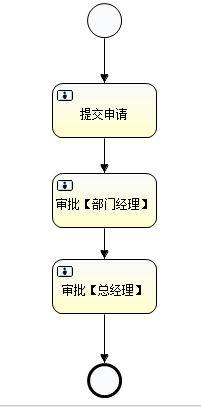
|  |
| --- |
| #RepositoryService  SELECT \* FROM `act\_ge\_bytearray`; #二进制文件表  SELECT \* FROM `act\_re\_deployment`;#流程部署表  SELECT \* FROM `act\_re\_procdef`;#流程定义  SELECT \* FROM `act\_ge\_property`;#工作流的ID算法和版本信息表  #RuntimeService TaskService  SELECT \* FROM `act\_ru\_execution`;#流程启动一次只要没有执行完，就会有一条数据  SELECT \* FROM `act\_ru\_task`;#可能有多条数据  SELECT \* FROM `act\_ru\_variable`;#记录流程运行时的流程变量  SELECT \* FROM `act\_ru\_identitylink`;#存放流程办理人的信息  #HistroyService  SELECT \* FROM `act\_hi\_procinst`;#历史流程实例  SELECT \* FROM `act\_hi\_taskinst`;#历史任务实例  SELECT \* FROM `act\_hi\_actinst`;#历史活动节点表  SELECT \* FROM `act\_hi\_varinst`;#历史流程变量表  SELECT \* FROM `act\_hi\_identitylink`;##历史办理人表  SELECT \* FROM `act\_hi\_comment`;#批注表  SELECT \* FROM `act\_hi\_attachment`;#附件表  #IdentityService  SELECT \* FROM `act\_id\_group` #角色  SELECT \* FROM `act\_id\_membership`#用户和角色之间的关系  SELECT \* FROM `act\_id\_info`#用户的详细信息  SELECT \* FROM `act\_id\_user`#用户表 |

# 08【掌握】管理流程定义

功能：对流程的增加 修改 删除 查询

act\_ge\_bytearray act\_re\_deployment act\_re\_procdef

## 流程图



## 部署流程【两种方式 classpath zip】

|  |
| --- |
| //得到流程引擎  **private** ProcessEngine processEngine=ProcessEngines.*getDefaultProcessEngine*();    /\*\*  \* 部署流程使用classpath  \*/  @Test  **public** **void** deployProcess01() {  //得到流程部署的service  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  Deployment deploy = repositoryService.createDeployment().name("请假流程001")  .addClasspathResource("HelloWorld.bpmn")  .addClasspathResource("HelloWorld.png")  .deploy();  System.***out***.println("部署成功:流程部署ID："+deploy.getId());  }    /\*\*  \* 部署流程使用zip  \* 流程图的文件必须是xxxx.zip结尾  \*/  @Test  **public** **void** deployProcess02() {  //如果不加/代表从当前包里面找文件  InputStream inputStream = **this**.getClass().getResourceAsStream("/HelloWorld.zip");  //如果加/代表从classpath的根目录里面找文件  //InputStream inputStream = this.getClass().getResourceAsStream("/HelloWorld.zip");  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  ZipInputStream zipInputStream =**new** ZipInputStream(inputStream);  Deployment deploy = repositoryService.createDeployment().name("请假流程001")  .addZipInputStream(zipInputStream)//添加流程图的流  .deploy();//确定部署  System.***out***.println("部署成功,部署ID:"+deploy.getId());  } |

## 查询部署信息act\_re\_deployment

|  |
| --- |
| /\*\*  \* 查询流程部署信息 act\_re\_deployment  \*/  @Test  **public** **void** queryProcessDeploy () {  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  //创建部署信息的查询  String deploymentId="1";  // Deployment deploy = repositoryService.createDeploymentQuery()  List<Deployment> list= repositoryService.createDeploymentQuery()  //条件  //.deploymentId(deploymentId) //根据部署ID去查询  //.deploymentName(name)//根据部署名称去查询  //.deploymentTenantId(tenantId)//根据tenantId去查询  // .deploymentNameLike(nameLike)//根据部署名称模糊查询  //.deploymentTenantIdLike(tenantIdLike)//根据tenantId模糊查询  //排序  // .orderByDeploymentId().asc() //根据部署ID升序  //.orderByDeploymenTime().desc() //根据部署时间降序  //.orderByDeploymentName()//根据部署名称升序  //结果集  .list(); //查询返回list集合  // .listPage(firstResult, maxResults) 分页查询返回list集合  //.singleResult(); //返回单个对象  // .count();    /\*System.out.println("部署ID:"+deploy.getId());  System.out.println("部署名称:"+deploy.getName());  System.out.println("部署时间:"+deploy.getDeploymentTime());\*/  // System.out.println(count);  **for** (Deployment deployment : list) {  System.***out***.println("部署ID:"+deployment.getId());  System.***out***.println("部署名称:"+deployment.getName());  System.***out***.println("部署时间:"+deployment.getDeploymentTime());  System.***out***.println("########################");  }  } |

## 查询流程定义信息act\_re\_procdef

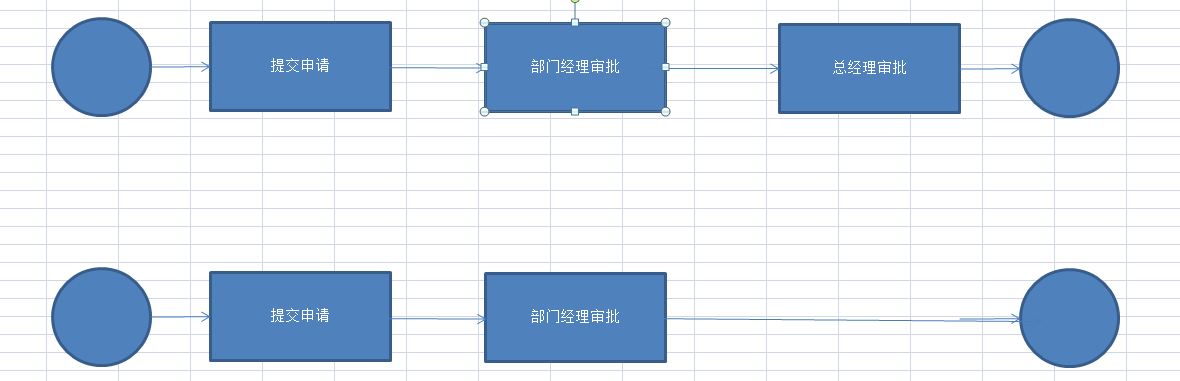
|  |
| --- |
| /\*\*  \* 查询流程定义  \*/  @Test  **public** **void** queryProcDef() {  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  List<ProcessDefinition> list = repositoryService.createProcessDefinitionQuery()  //条件  // .deploymentId(deploymentId) 根据部署ID查询  // .deploymentIds(deploymentIds) 根据部署ID的集合查询Set<String> deploymentIds  // .processDefinitionId(processDefinitionId)//根据流程定义IDHelloWorld:1:4  // .processDefinitionIds(processDefinitionIds)//根据流程定义的IDS查询  // .processDefinitionKey(processDefinitionKey)//根据流程定义的的key查询  // .processDefinitionKeyLike(processDefinitionKeyLike)//根据流程定义的的key模糊查询  // .processDefinitionName(processDefinitionName)//根据流程定义的名称查询  // .processDefinitionNameLike(processDefinitionNameLike)//根据流程定义的名称模糊查询  // .processDefinitionResourceName(resourceName)//根据流程图的BPMN文件名查询  // .processDefinitionResourceNameLike(resourceNameLike)//根据流程图的BPMN文件名模糊查询  // .processDefinitionVersion(processDefinitionVersion)//根据流程定义的版本查询  // .processDefinitionVersionGreaterThan(processDefinitionVersion)//version>num  // .processDefinitionVersionGreaterThanOrEquals(processDefinitionVersion)//version>=num  // .processDefinitionVersionLowerThan(processDefinitionVersion)//version<num  // .processDefinitionVersionLowerThanOrEquals(processDefinitionVersion)//version<=num  //排序  // .orderByDeploymentId()  // .orderByProcessDefinitionId()  // .orderByProcessDefinitionKey()  // .orderByProcessDefinitionName()  // .orderByProcessDefinitionVersion()  //结果集  .list();  // .listPage(firstResult, maxResults)\  // .count()  // .singleResult()  **if**(**null**!=list&&list.size()>0) {  **for** (ProcessDefinition pd : list) {  System.***out***.println("流程定义ID:"+pd.getId());  System.***out***.println("流程部署ID:"+pd.getDeploymentId());  System.***out***.println("流程定义KEY:"+pd.getKey());  System.***out***.println("流程定义的名称:"+pd.getName());  System.***out***.println("流程定义的bpmn文件名:"+pd.getResourceName());//bpmn的name  System.***out***.println("流程图片名:"+pd.getDiagramResourceName());//png的name  System.***out***.println("流程定义的版本号:"+pd.getVersion());  System.***out***.println("##################");  }  }    } |

## 删除流程定义信息

|  |
| --- |
| /\*\*  \* 删除流程定义  \*/  @Test  **public** **void** deleteProcessDef() {  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  String deploymentId="2501";  //根据流程部署id删除流程定义 如果当前id的流程正在执行，那么会报错  //repositoryService.deleteDeployment(deploymentId);  //根据流程部署id删除删除流程定义 如果当前id的流程正在执行,会把正在执行的流程数据删除 act\_ru\_\*和act\_hi\_\*表里面的数据  repositoryService.deleteDeployment(deploymentId, **true**);  // repositoryService.deleteDeploymentCascade(deploymentId);==repositoryService.deleteDeployment(deploymentId, true);  System.***out***.println("删除成功");  } |

## 修改流程定义信息

修改流程图之后重新部署，只要key不变，它的版本号就会+1



## 查询流程图

|  |
| --- |
| /\*\*  \* 查询流程图 根据流程定义ID  \*/  @Test  **public** **void** viewProcessImg() {  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  String processDefinitionId="HelloWorld:1:4";  InputStream inputStream = repositoryService.getProcessDiagram(processDefinitionId);    File file=**new** File("d:/HelloWorld.png");  **try** {  BufferedOutputStream outputStream=**new** BufferedOutputStream(**new** FileOutputStream(file));  **int** len=0;  **byte**[] b=**new** **byte**[1024];  **while**((len=inputStream.read(b))!=-1) {  outputStream.write(b, 0, len);  outputStream.flush();  }  outputStream.close();  inputStream.close();  System.***out***.println("查询成功");  } **catch** (Exception e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  /\*\*  \* 查询流程图 根据流流程部署ID  \*/  @Test  **public** **void** viewProcessImg2() {  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  //根据流程部署ID查询流程定义对象  String deploymentId="1";  ProcessDefinition processDefinition = repositoryService.createProcessDefinitionQuery().deploymentId(deploymentId).singleResult();  //从流程定义对象里面查询出流程定义ID  String processDefinitionId=processDefinition.getId();  InputStream inputStream = repositoryService.getProcessDiagram(processDefinitionId);    File file=**new** File("d:/"+processDefinition.getDiagramResourceName());  **try** {  BufferedOutputStream outputStream=**new** BufferedOutputStream(**new** FileOutputStream(file));  **int** len=0;  **byte**[] b=**new** **byte**[1024];  **while**((len=inputStream.read(b))!=-1) {  outputStream.write(b, 0, len);  outputStream.flush();  }  outputStream.close();  inputStream.close();  System.***out***.println("查询成功");  } **catch** (Exception e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  } |

## 附加功能：查询最新版本的流程定义

|  |
| --- |
| /\*\*  \* 查询最新的流程定义  \*/  @Test  **public** **void** queryNewProcessDef() {  Map<String, ProcessDefinition> map = **new** HashMap<>();  // 查询所有的流程定义根据版本号升序  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  List<ProcessDefinition> list = repositoryService.createProcessDefinitionQuery()  .orderByProcessDefinitionVersion().asc().list();  **if**(**null**!=list&&list.size()>0) {  **for** (ProcessDefinition pd : list) {  map.put(pd.getKey(), pd);  }  }  //循环map集合  Collection<ProcessDefinition> values = map.values();  **for** (ProcessDefinition pd : values) {  System.***out***.println("流程定义ID:" + pd.getId());  System.***out***.println("流程部署ID:" + pd.getDeploymentId());  System.***out***.println("流程定义KEY:" + pd.getKey());  System.***out***.println("流程定义的名称:" + pd.getName());  System.***out***.println("流程定义的bpmn文件名:" + pd.getResourceName());// bpmn的name  System.***out***.println("流程图片名:" + pd.getDiagramResourceName());// png的name  System.***out***.println("流程定义的版本号:" + pd.getVersion());  System.***out***.println("##################");  }  } |

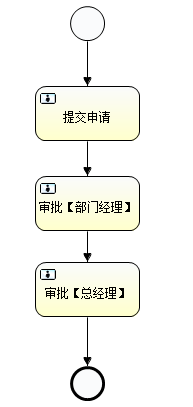
## 附加功能：删除流程定义（删除key相同的所有不同版本的流程定义）

已知key

|  |
| --- |
| /\*\*  \* 已知key 附加功能：删除流程定义（删除key相同的所有不同版本的流程定义）  \*/  **public** **void** deleteAllSameVersion() {  String processDefinitionKey = "HelloWorld";  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  // 根据流程定义的key查询流程集合  List<ProcessDefinition> list = repositoryService.createProcessDefinitionQuery()  .processDefinitionKey(processDefinitionKey).list();  **if** (**null** != list && list.size() > 0) {  **for** (ProcessDefinition pd : list) {  repositoryService.deleteDeployment(pd.getDeploymentId(), **true**);  }  }  } |

# 09【掌握】流程实例、任务的执行

## 1，流程图

****

## 2，部署流程定义

|  |
| --- |
| /\*\*  \* 部署流程使用classpath  \*/  @Test  **public** **void** deployProcess() {  // 得到流程部署的service  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  Deployment deploy = repositoryService.createDeployment().name("请假流程001").addClasspathResource("HelloWorld.bpmn")  .addClasspathResource("HelloWorld.png").deploy();  System.***out***.println("部署成功:流程部署ID：" + deploy.getId());  } |

## 3，启动流程实例

|  |
| --- |
| /\*\*  \* 启动流程  \*/  @Test  **public** **void** startProcess() {  RuntimeService runtimeService = **this**.processEngine.getRuntimeService();  // runtimeService.startProcessInstanceById(processDefinitionId)//根据流程定义ID启动流程  /\*\*  \* 参数1：流程定义ID  \* 参数2：Map<String,Object> 流程变量  \*/  // runtimeService.startProcessInstanceById(processDefinitionId, variables);  /\*\*  \* 参数1：流程定义ID  \* 参数2：String 业务ID 把业务ID和流程执行实例进行绑定  \*/  // runtimeService.startProcessInstanceById(processDefinitionId, businessKey);  /\*\*  \* 参数1：流程定义ID  \* 参数2：String 业务ID 把业务ID和流程执行实例进行绑定  \* 参数3：Map<String,Object> 流程变量  \*/  // runtimeService.startProcessInstanceById(processDefinitionId, businessKey, variables)  //  // runtimeService.startProcessInstanceByKey(processDefinitionKey)//根据流程定义的key启动  /\*\*  \* 参数1：流程定义的Key  \* 参数2：Map<String,Object> 流程变量  \*/  // runtimeService.startProcessInstanceByKey(processDefinitionKey, variables)  /\*\*  \* 参数1：流程定义Key  \* 参数2：String 业务ID 把业务ID和流程执行实例进行绑定  \*/  // runtimeService.startProcessInstanceByKey(processDefinitionId, businessKey);  /\*\*  \* 参数1：流程定义Key  \* 参数2：String 业务ID 把业务ID和流程执行实例进行绑定  \* 参数3：Map<String,Object> 流程变量  \*/  // runtimeService.startProcessInstanceByKey(processDefinitionId, businessKey, variables)    //实例开发中使用的  //runtimeService.startProcessInstanceByKey(processDefinitionId, businessKey, variables)  //runtimeService.startProcessInstanceByKey(processDefinitionId, businessKey);    String processDefinitionKey="HelloWorld";  ProcessInstance processInstance = runtimeService.startProcessInstanceByKey(processDefinitionKey);  System.***out***.println("流程启动成功:"+processInstance.getId()+" "+processInstance.getProcessDefinitionId()+" "+processInstance.getProcessInstanceId());    } |

## 4，查询我的个人任务

|  |
| --- |
| /\*\*  \* 查询我的个人任务act\_ru\_task  \*/  @Test  **public** **void** queryMyTask() {  TaskService taskService = **this**.processEngine.getTaskService();  String assignee="张三";  List<Task> list = taskService.createTaskQuery()  //条件  .taskAssignee(assignee)//根据任务办理人查询任务  // .deploymentId(deploymentId)//根据部署ID查询 where id=id  // .deploymentIdIn(deploymentIds)//根据部署ID集合查询 where id in (1,2,3,4)  // .executionId(executionId)//根据执行实例ID  // .processDefinitionId(processDefinitionId)//根据流程定义ID  // .processDefinitionKey(processDefinitionKey)//根据流程定义的key  // .processDefinitionKeyIn(processDefinitionKeys)  // .processDefinitionKeyLike(processDefinitionKeyLike)  // .processDefinitionName(processDefinitionName)  // .processDefinitionNameLike(processDefinitionNameLike)  // .processInstanceBusinessKey(processInstanceBusinessKey)  //排序  .orderByTaskCreateTime().desc()  //结果集  .list();  // .listPage(firstResult, maxResults)  // .count();  // .singleResult()    **if**(**null**!=list&&list.size()>0) {  **for** (Task task : list) {  System.***out***.println("任务ID:"+task.getId());  System.***out***.println("任务办理人:"+task.getAssignee());  System.***out***.println("执行实例ID:"+task.getExecutionId());  System.***out***.println("任务名称:"+task.getName());  System.***out***.println("流程定义ID:"+task.getProcessDefinitionId());  System.***out***.println("流程实例ID:"+task.getProcessInstanceId());  System.***out***.println("任务创建时间:"+task.getCreateTime());  System.***out***.println("####################");  }  }  } |

## 5，办理任务

|  |
| --- |
| /\*\*  \* 办理任务  \*/  @Test  **public** **void** completeTask() {  TaskService taskService = **this**.processEngine.getTaskService();  String taskId="2504";  //根据任务ID去完成任务  taskService.complete(taskId);  //根据任务ID去完成任务并指定流程变量  // taskService.complete(taskId, variables);  System.***out***.println("任务完成");  } |

## 6，查询流程状态（判断流程正在执行，还是结束）

|  |
| --- |
| /\*\*  \* 判断流程是否结束 作用：更新业务表里面的状态  \*/  @Test  **public** **void** isComplete() {  // 已知流程实例ID  RuntimeService runtimeService = **this**.processEngine.getRuntimeService();  String processInstanceId = "2501";  ProcessInstance processInstance = runtimeService.createProcessInstanceQuery()  .processInstanceId(processInstanceId).singleResult();  **if** (**null** != processInstance) {  System.***out***.println("流程未结束");  } **else** {  System.***out***.println("流程已结束");  }  //已知任务ID 5002 [了解]  //根据任务ID查询任务实例对象  /\*TaskService taskService = this.processEngine.getTaskService();  String taskId="5002";  Task task = taskService.createTaskQuery().taskId(taskId).singleResult();  //从任务实例里面取出流程实例ID  String processInstanceId2 = task.getProcessInstanceId();  //使用流程实例ID去流程实例表里面查询有没有数据  ProcessInstanceQuery processInstance2 = runtimeService.createProcessInstanceQuery().processInstanceId(processInstanceId2);  if (null != processInstance2) {  System.out.println("流程未结束");  } else {  System.out.println("流程已结束");  }\*/        } |

## 7，查询正在执行的流程实例

|  |
| --- |
| /\*\*  \* 7，查询当前的流程实例 act\_ru\_execution  \*  \*/  @Test  **public** **void** queryProcessInstance() {  RuntimeService runtimeService = **this**.processEngine.getRuntimeService();  List<ProcessInstance> list = runtimeService.createProcessInstanceQuery().list();  **if**(**null**!=list&&list.size()>0) {  **for** (ProcessInstance pi : list) {  System.***out***.println("执行实例ID:"+pi.getId());  System.***out***.println("流程定义ID:"+pi.getProcessDefinitionId());  System.***out***.println("流程实例ID:"+pi.getProcessInstanceId());  System.***out***.println("########################");  }  }  } |

## 8，附加功能，查询历史任务（后面讲）

|  |
| --- |
| /\*\*  \* 7，附加功能，查询历史任务（后面讲）act\_hi\_taskinst  \*/  @Test  **public** **void** queryHistoryTask() {  HistoryService historyService = **this**.processEngine.getHistoryService();  List<HistoricTaskInstance> list = historyService.createHistoricTaskInstanceQuery().list();  **if**(**null**!=list&&list.size()>0) {  **for** (HistoricTaskInstance hi : list) {  System.***out***.println("任务ID:" + hi.getId());  System.***out***.println("任务办理人:" + hi.getAssignee());  System.***out***.println("执行实例ID:" + hi.getExecutionId());  System.***out***.println("任务名称:" + hi.getName());  System.***out***.println("流程定义ID:" + hi.getProcessDefinitionId());  System.***out***.println("流程实例ID:" + hi.getProcessInstanceId());  System.***out***.println("任务创建时间:" + hi.getCreateTime());  System.***out***.println("任务结束时间:" + hi.getEndTime());  System.***out***.println("任务持续时间:" + hi.getDurationInMillis());  System.***out***.println("####################");  }  }    } |

## 9，附加功能，查询历史流程实例（后面讲）

|  |
| --- |
| /\*\*  \* 8，附加功能，查询历史流程实例（后面讲）  \*/  @Test  **public** **void** queryHistoryProcessInstance()  {  HistoryService historyService = **this**.processEngine.getHistoryService();  List<HistoricProcessInstance> list = historyService.createHistoricProcessInstanceQuery().list();  **if**(**null**!=list&&list.size()>0) {  **for** (HistoricProcessInstance hi : list) {  System.***out***.println("执行实例ID:" + hi.getId());  System.***out***.println("流程定义ID:" + hi.getProcessDefinitionId());  System.***out***.println("流程启动时间:" + hi.getStartTime());  System.***out***.println("########################");  }  }  } |

## 10，总结

**Execution 执行对象**

 按流程定义的规则执行一次的过程.

 对应的表：

  act\_ru\_execution， 正在执行的信息

  act\_hi\_procinst，已经执行完的历史流程实例信息

  act\_hi\_actinst，存放历史所有完成的活动

**ProcessInstance 流程实例**

 特指流程从开始到结束的那个最大的执行分支，一个执行的流程中，流程实例只有1个。

注意

（1）如果是单例流程，执行对象ID就是流程实例ID

（2）如果一个流程有分支和聚合，那么执行对象ID和流程实例ID就不相同

（3）一个流程中，流程实例只有1个，执行对象可以存在多个。

**Task 任务**

 执行到某任务环节时生成的任务信息。

 对应的表：

  act\_ru\_task，正在执行的任务信息

  act\_hi\_taskinst，已经执行完的历史任务信息

相关ID总结

部署ID ---act\_re\_deployment id

流程定义ID act\_re\_procdef id

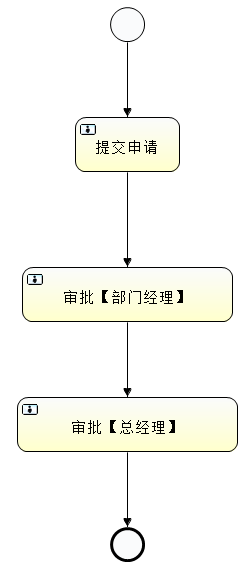
流程实例ID act\_ru\_execution id

执行实例ID act\_ru\_task execution\_id

任务ID act\_run\_task id

# 10【掌握】流程变量

## 1：流程图



流程变量在整个工作流中扮演很重要的作用。例如：请假流程中有请假天数、请假原因等一些参数都为流程变量的范围。流程变量的作用域范围是只对应一个流程实例。也就是说各个流程实例的流程变量是不相互影响的。流程实例结束完成以后流程变量还保存在数据库中（存放到流程变量的历史表中）。

设置流程变量的值

存在形式 key ---value

## 2：部署流程定义

|  |
| --- |
| /\*\*  \* 部署流程使用classpath  \*/  @Test  **public** **void** deployProcess() {  // 得到流程部署的service  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  Deployment deploy = repositoryService.createDeployment().name("请假流程001").addClasspathResource("HelloWorld.bpmn")  .addClasspathResource("HelloWorld.png").deploy();  System.***out***.println("部署成功:流程部署ID：" + deploy.getId());  } |

## 3：启动流程实例并设置流程变量

|  |
| --- |
| /\*\*  \* 启动流程  \*/  @Test  **public** **void** startProcess() {  RuntimeService runtimeService = **this**.processEngine.getRuntimeService();  String processDefinitionKey = "HelloWorld";  // ProcessInstance processInstance = runtimeService.startProcessInstanceByKey(processDefinitionKey);  //创建流程变量对象  Map<String,Object> variables=**new** HashMap<>();  variables.put("请假天数", 5);//int  variables.put("请假原因", "约会");  variables.put("请假时间", **new** Date());  ProcessInstance processInstance = runtimeService.startProcessInstanceByKey(processDefinitionKey, variables);  System.***out***.println("流程启动成功:" + processInstance.getId() + " " + processInstance.getProcessDefinitionId() + " "  + processInstance.getProcessInstanceId());  } |

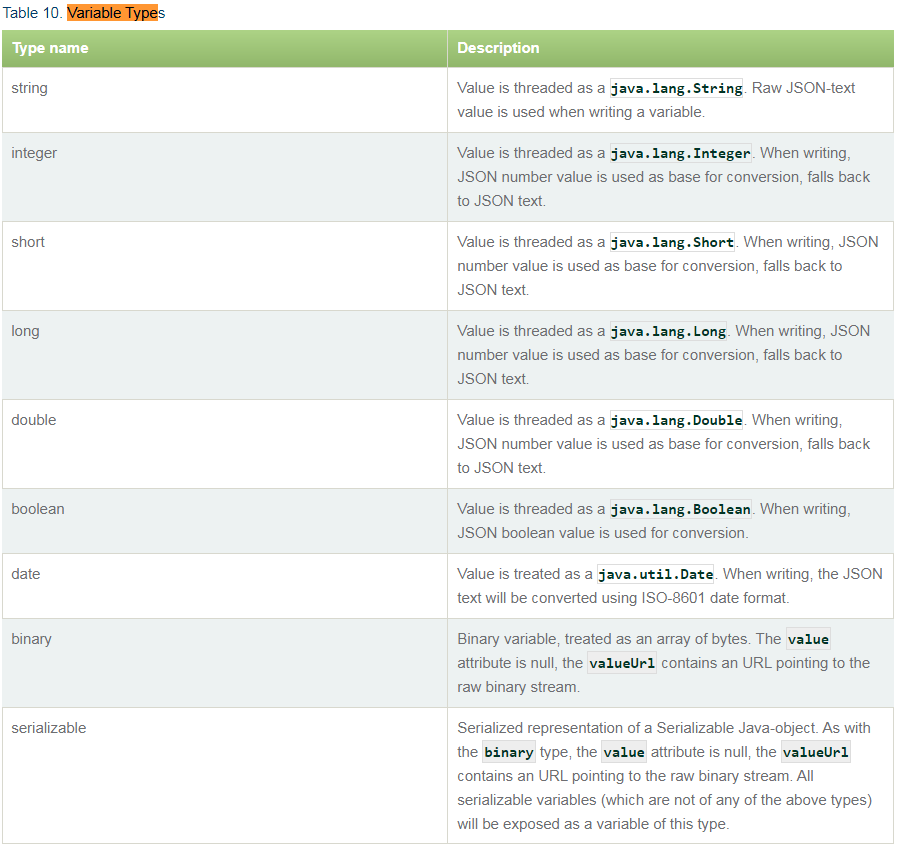
## 4：设置流程变量

|  |
| --- |
| /\*\*  \* 设置流程变量1  \*/  @Test  **public** **void** setVariables() { 存储对象的时候需要实现序列化，不然会报错，在新增属性的时候  RuntimeService runtimeService = **this**.processEngine.getRuntimeService();  String executionId="2501";  //runtimeService.setVariable(executionId, "请假人", "小明");  Map<String,Object> variables=**new** HashMap<>();  variables.put("请假天数", 6);//int  variables.put("请假原因", "约会妹子");  variables.put("请假时间", **new** Date());  variables.put("用户对象", **new** User(1,"小明"));  runtimeService.setVariables(executionId, variables);  System.***out***.println("流程变量设置成功");      }  /\*\*  \* 设置流程变量2  \*/  @Test  **public** **void** setVariables2() {  TaskService taskService = **this**.processEngine.getTaskService();    String taskId="2507";  //runtimeService.setVariable(executionId, "请假人", "小明");  Map<String,Object> variables=**new** HashMap<>();  variables.put("任务ID设置的", 9);//int  // taskService.setVariable(taskId, variableName, value);  taskService.setVariables(taskId, variables);  System.***out***.println("流程变量设置成功");  } |

## 5：获取流程变量

|  |
| --- |
| /\*\*  \* 获取流程变量  \*/  @Test  **public** **void** getVariables() {  RuntimeService runtimeService = **this**.processEngine.getRuntimeService();  String executionId="2501";  Integer days=(Integer) runtimeService.getVariable(executionId, "请假天数");  Date date=(Date) runtimeService.getVariable(executionId, "请假时间");  User user=(User) runtimeService.getVariable(executionId, "用户对象");  System.***out***.println(days);  System.***out***.println(date.~~toLocaleString~~());  System.***out***.println(user.getId()+" "+user.getName());    } |

## 6：流程变量的支持的类型



## 7：查询历史的流程变量

|  |
| --- |
| /\*\*  \* 7：查询历史的流程变量  \*/  @Test  **public** **void** getHistoryVariables() {  HistoryService historyService = **this**.processEngine.getHistoryService();    /\*HistoricVariableInstance singleResult = historyService.createHistoricVariableInstanceQuery().id("2503").singleResult();;  System.out.println(singleResult.getId());  System.out.println(singleResult.getValue());  System.out.println(singleResult.getVariableName());  System.out.println(singleResult.getVariableTypeName());\*/  String processInstanceId="2501";  List<HistoricVariableInstance> list = historyService.createHistoricVariableInstanceQuery().processInstanceId(processInstanceId).list();    **for** (HistoricVariableInstance hvs : list) {  System.***out***.println("ID"+hvs.getId());  System.***out***.println("变量值"+hvs.getValue());  System.***out***.println("变量名"+hvs.getVariableName());  System.***out***.println("变量类型"+hvs.getVariableTypeName());  System.***out***.println("#####################");  }  } |

## 9：总结

**1：流程变量**

在流程执行或者任务执行的过程中，用于设置和获取变量，使用流程变量在流程传递的过程中传递业务参数。

对应的表：

act\_ru\_variable：正在执行的流程变量表

act\_hi\_varinst：流程变量历史表

**2：扩展知识：setVariable和setVariableLocal的区别**

setVariable：设置流程变量的时候，流程变量名称相同的时候，后一次的值替换前一次的值，而且可以看到TASK\_ID的字段不会存放任务ID的值

setVariableLocal：

1：设置流程变量的时候，针对当前活动的节点设置流程变量，如果一个流程中存在2个活动节点，对每个活动节点都设置流程变量，即使流程变量的名称相同，后一次的版本的值也不会替换前一次版本的值，它会使用不同的任务ID作为标识，存放2个流程变量值，而且可以看到TASK\_ID的字段会存放任务ID的值 例如act\_hi\_varinst 表的数据：不同的任务节点，即使流程变量名称相同，存放的值也是不同的。

如图：



2：还有，使用setVariableLocal说明流程变量绑定了当前的任务，当流程继续执行时，下个任务获取不到这个流程变量（因为正在执行的流程变量中没有这个数据），所有查询正在执行的任务时不能查询到我们需要的数据，此时需要查询历史的流程变量。

# 11【掌握】流程执行历史记录

## 1，查询历史流程实例

|  |
| --- |
| /\* 1，查询历史流程实例 \*/  @Test  **public** **void** historyProcessInstince() {  List<HistoricProcessInstance> list = historyService.createHistoricProcessInstanceQuery()  //条件  // .processDefinitionId(processDefinitionId)  // .processDefinitionKey(processDefinitionKey)  // .processDefinitionKeyIn(processDefinitionKeys)  // .processDefinitionName(processDefinitionName)  // .processDefinitionVersion(processDefinitionVersion)  // .processInstanceBusinessKey(processInstanceBusinessKey)  // .processInstanceId(processInstanceId)  // .processInstanceIds(processInstanceIds)  //排序  // .orderByProcessDefinitionId()  // .orderByProcessInstanceBusinessKey()  // .orderByProcessInstanceDuration()  // .orderByProcessInstanceStartTime()  // .orderByProcessInstanceId()  //结果集  .list();  // .listPage(firstResult, maxResults)  // .count()  // .singleResult();    **if**(**null**!=list&&list.size()>0) {  **for** (HistoricProcessInstance hpi : list) {  System.***out***.println("历史流程实例ID:" + hpi.getId());  System.***out***.println("流程定义ID:" + hpi.getProcessDefinitionId());  System.***out***.println("历史流程实例的业务ID:" + hpi.getBusinessKey());  System.***out***.println("流程部署ID:" + hpi.getDeploymentId());  System.***out***.println("流程定义KEY:" + hpi.getProcessDefinitionKey());  System.***out***.println("开始活动ID:" + hpi.getStartActivityId());  System.***out***.println("结束活动ID:" + hpi.~~getEndActivityId~~());  System.***out***.println("########################");  }  }    } |

## 2，查询历史活动

|  |
| --- |
| /\* 2，查询历史活动 \*/  @Test  **public** **void** queryHistoryAct() {  List<HistoricActivityInstance> list = **this**.historyService.createHistoricActivityInstanceQuery()  //条件  // .activityId(activityId)  // .activityInstanceId(activityInstanceId)  // .activityName(activityName)  //排序  // .orderByActivityId()  // .orderByActivityName()  //结果集  .list();  **if**(**null**!=list&&list.size()>0)  {  **for** (HistoricActivityInstance hai : list) {  System.***out***.println("ID:"+hai.getId());  System.***out***.println("流程定义ID:"+hai.getProcessDefinitionId());  System.***out***.println("流程实例ID:"+hai.getProcessInstanceId());  System.***out***.println("执行实例ID:"+hai.getExecutionId());  System.***out***.println("活动ID:"+hai.getActivityId());  System.***out***.println("任务ID:"+hai.getTaskId());  System.***out***.println("活动名称:"+hai.getActivityName());  System.***out***.println("活动类型:"+hai.getActivityType());  System.***out***.println("任务办理人:"+hai.getAssignee());  System.***out***.println("开始时间:"+hai.getStartTime());  System.***out***.println("结束时间:"+hai.getEndTime());  System.***out***.println("持续时间:"+hai.getDurationInMillis());  System.***out***.println("#######################################");  }  }  } |

## 3，查询历史任务

|  |
| --- |
| /\* 3，查询历史任务 act\_hi\_taskinst \*/  @Test  **public** **void** queryHistoryTask() {  List<HistoricTaskInstance> list = **this**.historyService.createHistoricTaskInstanceQuery()  //条件  // .deploymentId(deploymentId)  // .deploymentIdIn(deploymentIds)  // .executionId(executionId)  // .processDefinitionId(processDefinitionId)  // .processDefinitionKey(processDefinitionKey)  // .processDefinitionKeyIn(processDefinitionKeys)  // .processDefinitionKeyLike(processDefinitionKeyLike) processDefinitionKeyLike="%Hello%"  // .processDefinitionName(processDefinitionName)  // .processDefinitionNameLike(processDefinitionNameLike)  //排序  // .orderByTaskDefinitionKey()  //结果集  .list();  // .listPage(firstResult, maxResults)  // .count()  // .singleResult()  **if**(**null**!=list&&list.size()>0)  {  **for** (HistoricTaskInstance task : list) {  System.***out***.println("任务ID:" + task.getId());  System.***out***.println("任务办理人:" + task.getAssignee());  System.***out***.println("执行实例ID:" + task.getExecutionId());  System.***out***.println("任务名称:" + task.getName());  System.***out***.println("流程定义ID:" + task.getProcessDefinitionId());  System.***out***.println("流程实例ID:" + task.getProcessInstanceId());  System.***out***.println("任务创建时间:" + task.getCreateTime());  System.***out***.println("任务结束时间:" + task.getEndTime());  System.***out***.println("#######################################");  }  }  } |

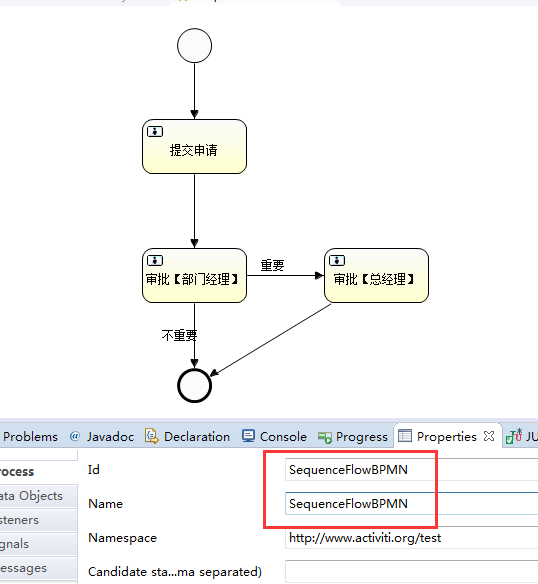
## 4，查询历史流程变量

|  |
| --- |
| /\*\*  \* 7：查询历史的流程变量  \*/  @Test  **public** **void** getHistoryVariables() {  HistoryService historyService = **this**.processEngine.getHistoryService();    /\*HistoricVariableInstance singleResult = historyService.createHistoricVariableInstanceQuery().id("2503").singleResult();;  System.out.println(singleResult.getId());  System.out.println(singleResult.getValue());  System.out.println(singleResult.getVariableName());  System.out.println(singleResult.getVariableTypeName());\*/  String processInstanceId="2501";  List<HistoricVariableInstance> list = historyService.createHistoricVariableInstanceQuery().processInstanceId(processInstanceId).list();    **for** (HistoricVariableInstance hvs : list) {  System.***out***.println("ID"+hvs.getId());  System.***out***.println("变量值"+hvs.getValue());  System.***out***.println("变量名"+hvs.getVariableName());  System.***out***.println("变量类型"+hvs.getVariableTypeName());  System.***out***.println("#####################");  }  } |

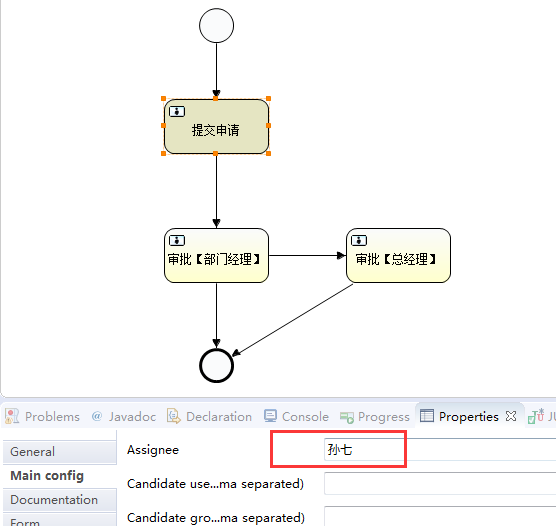
# 12【掌握】连线

使用流程变量去控制流程的走向

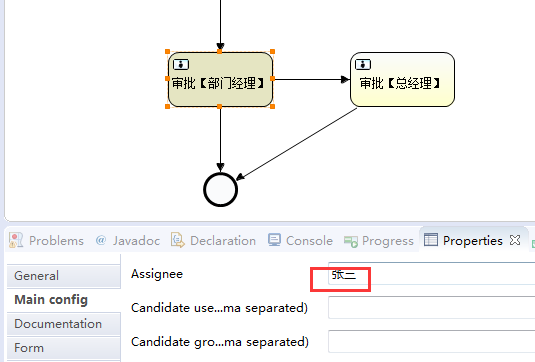
## 1，流程图



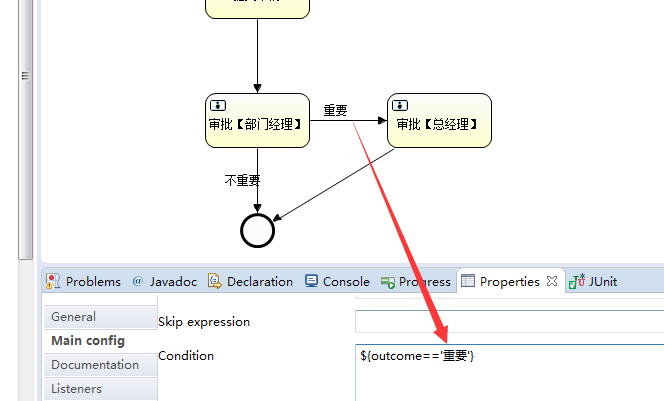
### 提交申请节点



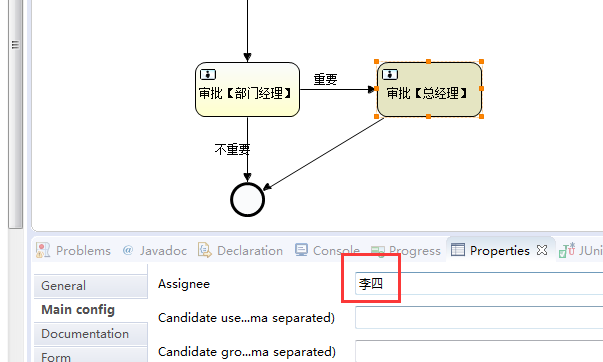
### 部门经理审批节点



### 部门经理审批节点连线



### 总经理审批节点



## 2，部署流程定义+启动流程实例

|  |
| --- |
| /\*\*  \* 部署流程使用zip  \*/  @Test  **public** **void** deployProcess() {  // 得到流程部署的service  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  InputStream inputStream=**this**.getClass().getResourceAsStream("SequenceFlowBPMN.zip");  ZipInputStream zipInputStream=**new** ZipInputStream(inputStream);  Deployment deploy = repositoryService.createDeployment().name("报销流程")  .addZipInputStream(zipInputStream).deploy();  System.***out***.println("部署成功:流程部署ID：" + deploy.getId());  }  /\*\*  \* 启动流程  \*/  @Test  **public** **void** startProcess() {  RuntimeService runtimeService = **this**.processEngine.getRuntimeService();  String processDefinitionKey = "SequenceFlowBPMN";  ProcessInstance processInstance = runtimeService.startProcessInstanceByKey(processDefinitionKey);  System.***out***.println("流程启动成功:" + processInstance.getId() + " " + processInstance.getProcessDefinitionId() + " "  + processInstance.getProcessInstanceId());  } |

## 3，查询我的个人任务

|  |
| --- |
| /\*\*  \* 查询我的个人任务act\_ru\_task  \*/  @Test  **public** **void** queryMyTask() {  TaskService taskService = **this**.processEngine.getTaskService();  String assignee = "李四";  List<Task> list = taskService.createTaskQuery()  // 条件  .taskAssignee(assignee)// 根据任务办理人查询任务  // 排序  .orderByTaskCreateTime().desc()  // 结果集  .list();  **if** (**null** != list && list.size() > 0) {  **for** (Task task : list) {  System.***out***.println("任务ID:" + task.getId());  System.***out***.println("任务办理人:" + task.getAssignee());  System.***out***.println("执行实例ID:" + task.getExecutionId());  System.***out***.println("任务名称:" + task.getName());  System.***out***.println("流程定义ID:" + task.getProcessDefinitionId());  System.***out***.println("流程实例ID:" + task.getProcessInstanceId());  System.***out***.println("任务创建时间:" + task.getCreateTime());  System.***out***.println("####################");  }  }  } |

## 4，完成任务

|  |
| --- |
| /\*\*  \* 办理任务  \*/  @Test  **public** **void** completeTask() {  TaskService taskService = **this**.processEngine.getTaskService();  String taskId = "15003";  // 根据任务ID去完成任务  taskService.complete(taskId);  // 根据任务ID去完成任务并指定流程变量  // taskService.complete(taskId, variables);  System.***out***.println("任务完成");  }  /\*\*  \* 办理任务并使用流程变量指定流程走向  \*/  @Test  **public** **void** completeTask2() {  TaskService taskService = **this**.processEngine.getTaskService();  String taskId = "12502";  Map<String, Object> variables=**new** HashMap<>();  variables.put("outcome", "重要");  // 根据任务ID去完成任务并指定流程变量  taskService.complete(taskId, variables);  System.***out***.println("任务完成");  } |

## 5，总结

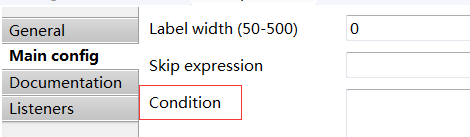
1、一个活动中可以指定一个或多个SequenceFlow（Start中有一个，End中没有）。

  \* 开始活动中有一个SequenceFlow 。

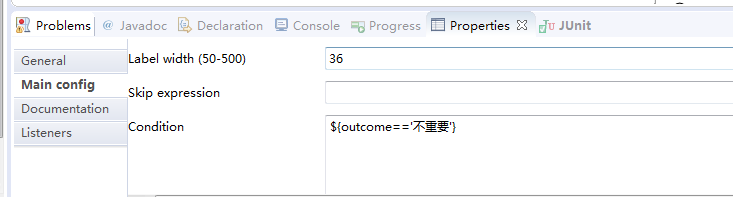
  \* 结束活动中没有SequenceFlow 。

  \* 其他活动中有1条或多条SequenceFlow

2、如果只有一个，则可以不使用流程变量设置codition的名称；

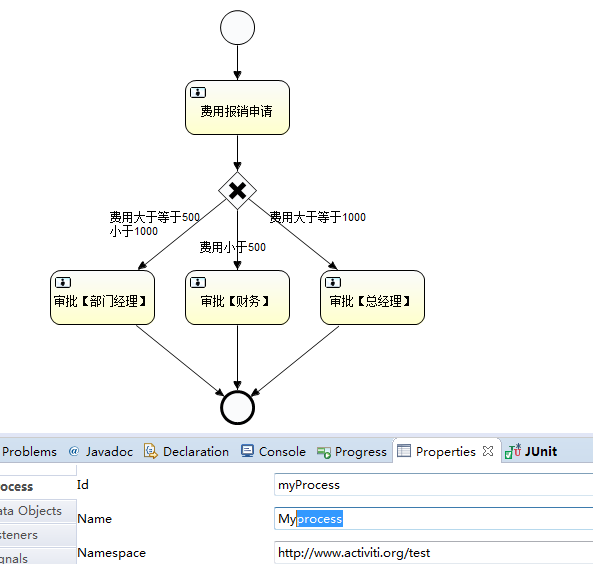


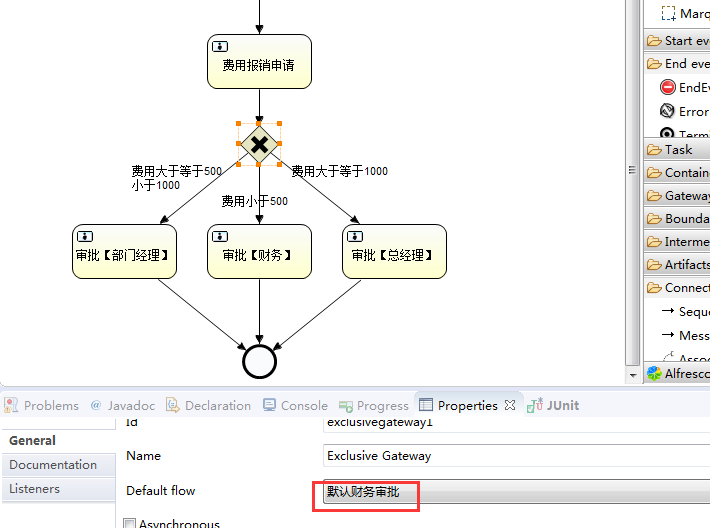
如果有多个，则需要使用流程变量设置codition的名称。outcomes表示流程变量的名称，‘不重要’表示流程变量的值，${}中间的内容要使用boolean类型的表达式，用来判断应该执行的连线。

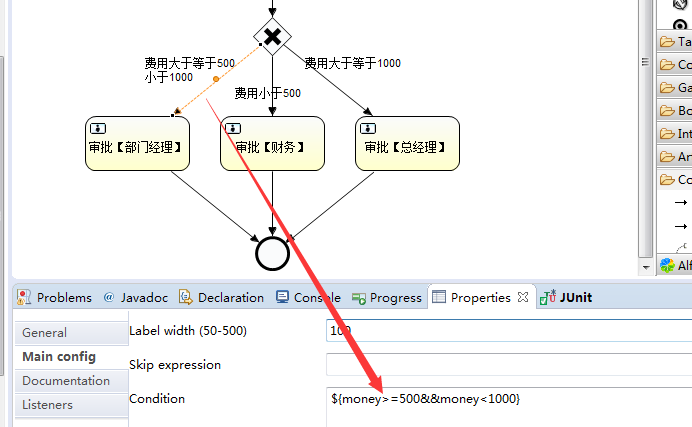


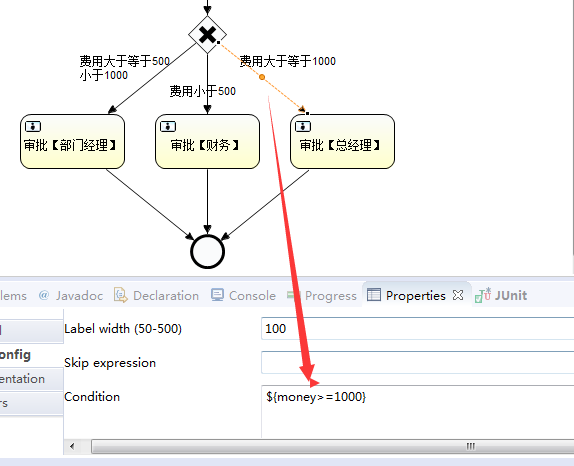
# 13【掌握】排他网关（ExclusiveGateWay）

## 1，流程图









## 2，部署流程定义+启动流程实例

|  |
| --- |
| /\*\*  \* 部署流程使用zip  \*/  @Test  **public** **void** deployProcess() {  // 得到流程部署的service  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  InputStream inputStream=**this**.getClass().getResourceAsStream("ExclusiveGateWay.zip");  ZipInputStream zipInputStream=**new** ZipInputStream(inputStream);  Deployment deploy = repositoryService.createDeployment().name("报销流程")  .addZipInputStream(zipInputStream).deploy();  System.***out***.println("部署成功:流程部署ID：" + deploy.getId());  }  /\*\*  \* 启动流程  \*/  @Test  **public** **void** startProcess() {  RuntimeService runtimeService = **this**.processEngine.getRuntimeService();  String processDefinitionKey = "myProcess";  ProcessInstance processInstance = runtimeService.startProcessInstanceByKey(processDefinitionKey);  System.***out***.println("流程启动成功:" + processInstance.getId() + " " + processInstance.getProcessDefinitionId() + " "  + processInstance.getProcessInstanceId());  } |

## 3，查询我的个人任务

|  |
| --- |
| /\*\*  \* 查询我的个人任务act\_ru\_task  \*/  @Test  **public** **void** queryMyTask() {  TaskService taskService = **this**.processEngine.getTaskService();  // String assignee = "张三";  // String assignee = "部门经理";  String assignee = "总经理";  List<Task> list = taskService.createTaskQuery()  // 条件  .taskAssignee(assignee)// 根据任务办理人查询任务  // 排序  .orderByTaskCreateTime().desc()  // 结果集  .list();  **if** (**null** != list && list.size() > 0) {  **for** (Task task : list) {  System.***out***.println("任务ID:" + task.getId());  System.***out***.println("任务办理人:" + task.getAssignee());  System.***out***.println("执行实例ID:" + task.getExecutionId());  System.***out***.println("任务名称:" + task.getName());  System.***out***.println("流程定义ID:" + task.getProcessDefinitionId());  System.***out***.println("流程实例ID:" + task.getProcessInstanceId());  System.***out***.println("任务创建时间:" + task.getCreateTime());  System.***out***.println("####################");  }  }  } |

## 4，完成我的个人任务

|  |
| --- |
| /\*\*  \* 办理任务  \*/  @Test  **public** **void** completeTask() {  TaskService taskService = **this**.processEngine.getTaskService();  String taskId = "22504";  // 根据任务ID去完成任务  taskService.complete(taskId);  // 根据任务ID去完成任务并指定流程变量  // taskService.complete(taskId, variables);  System.***out***.println("任务完成");  }  /\*\*  \* 办理任务并使用流程变量指定流程走向  \*/  @Test  **public** **void** completeTask2() {  TaskService taskService = **this**.processEngine.getTaskService();  String taskId = "20004";  Map<String, Object> variables=**new** HashMap<>();  variables.put("money", 1200);  // 根据任务ID去完成任务并指定流程变量  taskService.complete(taskId, variables);  System.***out***.println("任务完成");  } |

## 5，说明

1)一个排他网关对应一个以上的顺序流

2)由排他网关流出的顺序流都有个conditionExpression元素，在内部维护返回boolean类型的决策结果。

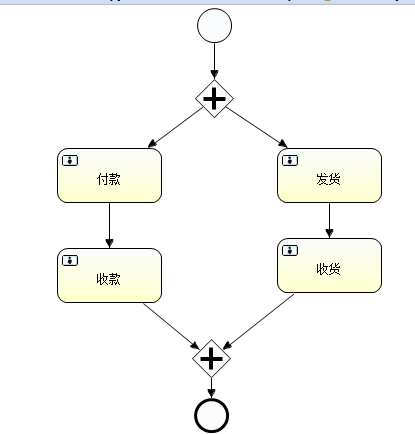
3)决策网关只会返回一条结果。当流程执行到排他网关时，流程引擎会自动检索网关出口，从上到下检索如果发现第一条决策结果为true或者没有设置条件的(默认为成立)，则流出。

4)如果没有任何一个出口符合条件，则抛出异常

5)使用流程变量，设置连线的条件，并按照连线的条件执行工作流，如果没有条件符合的条件，则以默认的连线离开。例如：

# 14【掌握】并行网关(parallelGateWay)

## 1，流程图



## 2，部署流程定义+启动流程实例

|  |
| --- |
| **private** ProcessEngine processEngine = ProcessEngines.*getDefaultProcessEngine*();  /\*\*  \* 部署流程使用zip  \*/  @Test  **public** **void** deployProcess() {  // 得到流程部署的service  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  InputStream inputStream=**this**.getClass().getResourceAsStream("ParallelGateWay.zip");  ZipInputStream zipInputStream=**new** ZipInputStream(inputStream);  Deployment deploy = repositoryService.createDeployment().name("tao宝流程")  .addZipInputStream(zipInputStream).deploy();  System.***out***.println("部署成功:流程部署ID：" + deploy.getId());  }  /\*\*  \* 启动流程  \*/  @Test  **public** **void** startProcess() {  RuntimeService runtimeService = **this**.processEngine.getRuntimeService();  String processDefinitionKey = "myProcess";  ProcessInstance processInstance = runtimeService.startProcessInstanceByKey(processDefinitionKey);  System.***out***.println("流程启动成功:" + processInstance.getId() + " " + processInstance.getProcessDefinitionId() + " "  + processInstance.getProcessInstanceId());  } |

## 3，查询我的个人任务

|  |
| --- |
| /\*\*  \* 查询我的个人任务act\_ru\_task  \*/  @Test  **public** **void** queryMyTask() {  TaskService taskService = **this**.processEngine.getTaskService();  // String assignee = "张三";  String assignee = "商家";  // String assignee = "买家";  List<Task> list = taskService.createTaskQuery()  // 条件  .taskAssignee(assignee)// 根据任务办理人查询任务  // 排序  .orderByTaskCreateTime().desc()  // 结果集  .list();  **if** (**null** != list && list.size() > 0) {  **for** (Task task : list) {  System.***out***.println("任务ID:" + task.getId());  System.***out***.println("任务办理人:" + task.getAssignee());  System.***out***.println("执行实例ID:" + task.getExecutionId());  System.***out***.println("任务名称:" + task.getName());  System.***out***.println("流程定义ID:" + task.getProcessDefinitionId());  System.***out***.println("流程实例ID:" + task.getProcessInstanceId());  System.***out***.println("任务创建时间:" + task.getCreateTime());  System.***out***.println("####################");  }  }  } |

## 4，完成我的个人任务

|  |
| --- |
| /\*\*  \* 办理任务  \*/  @Test  **public** **void** completeTask() {  TaskService taskService = **this**.processEngine.getTaskService();  String taskId = "10002";  // 根据任务ID去完成任务  taskService.complete(taskId);  // 根据任务ID去完成任务并指定流程变量  System.***out***.println("任务完成");  } |

## 5，说明

1）一个流程中流程实例只有1个，执行对象有多个

2）并行网关的功能是基于进入和外出的顺序流的

分支(fork):

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。

汇聚(join):

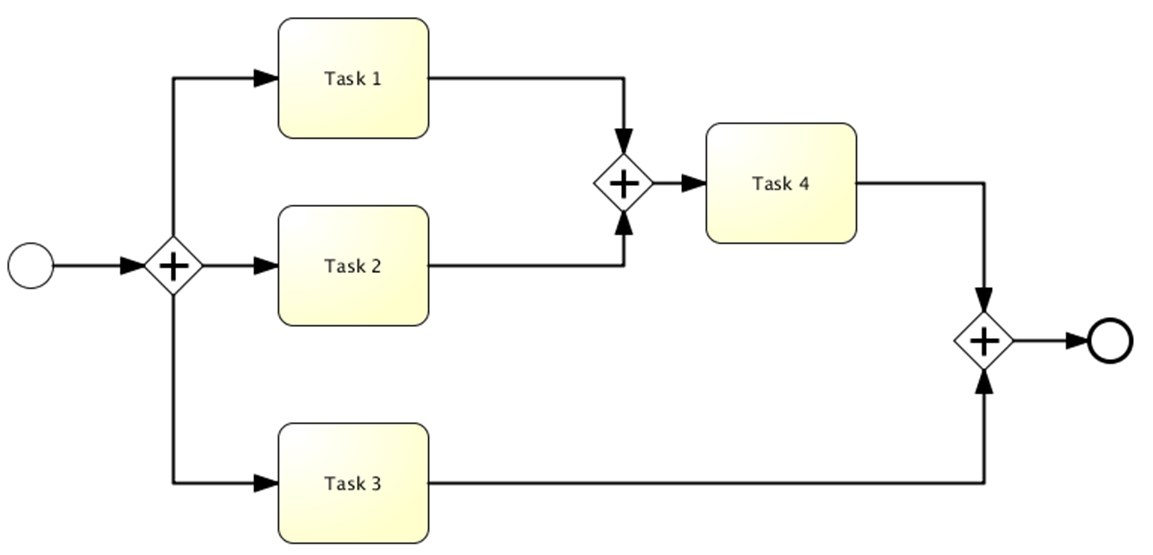
所有到达并行网关，在此等待的进入分支， 直到所有进入顺序流的分支都到达以后， 流程就会通过汇聚网关。

3）并行网关的进入和外出都是使用相同节点标识

4）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。

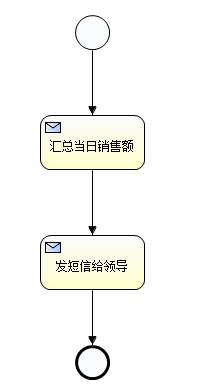
5）并行网关不会解析条件。 即使顺序流中定义了条件，也会被忽略。

6）并行网关不需要是“平衡的”（比如， 对应并行网关的进入和外出节点数目不一定相等）。如图中标示是合法的：



# 15【了解】接收活动（receiveTask，即等待活动）

## 1，流程图



## 2，部署流程定义

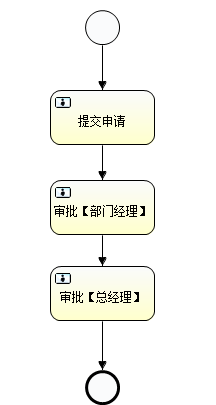
|  |
| --- |
| /\*\*  \* 部署流程使用zip  \*/  @Test  **public** **void** deployProcess() {  // 得到流程部署的service  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  InputStream inputStream=**this**.getClass().getResourceAsStream("Desktop.zip");  ZipInputStream zipInputStream=**new** ZipInputStream(inputStream);  Deployment deploy = repositoryService.createDeployment().name("汇总流程")  .addZipInputStream(zipInputStream).deploy();  System.***out***.println("部署成功:流程部署ID：" + deploy.getId());  } |

## 3，启动流程实例

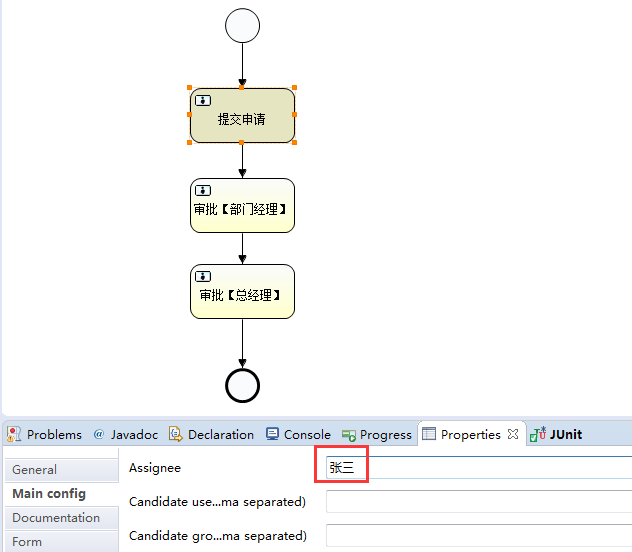
|  |
| --- |
| /\*\*  \* 启动流程  \*/  @Test  **public** **void** startProcess() {  RuntimeService runtimeService = **this**.processEngine.getRuntimeService();  String processDefinitionKey = "myProcess";  ProcessInstance processInstance = runtimeService.startProcessInstanceByKey(processDefinitionKey);  System.***out***.println("流程启动成功:" + processInstance.getId() + " " + processInstance.getProcessDefinitionId() + " "  + processInstance.getProcessInstanceId());  }  @Test  **public** **void** executionTask() {  RuntimeService runtimeService = **this**.processEngine.getRuntimeService();  String processInstanceId = "2501";  /\*\* 查询执行对象ID \*/  Execution execution1 = runtimeService//  .createExecutionQuery()// 创建执行对象查询  .processInstanceId(processInstanceId)// 使用流程实例ID查询  .activityId("receivetask1")// 当前活动的id，对应receiveTask.bpmn文件中的活动节点id的属性值  .singleResult();  System.***out***.println("执行实例ID:" + execution1.getId());  System.***out***.println("流程实例ID:" + execution1.getProcessInstanceId());  System.***out***.println("活动ID:" + execution1.getActivityId());  /\*\* 使用流程变量设置当日销售额，用来传递业务参数 \*/  **int** value = 10000;// 应该是去查询数据库，进行汇总 ---耗时操作  runtimeService.setVariable(execution1.getId(), "当前的销售额", value);  /\*\* 向后执行一步，如果流程处于等待状态，使得流程继续执行 \*/  runtimeService.signal(execution1.getId());  }  /\*\*  \* 发短信  \*/  @Test  **public** **void** sendMessage() {  RuntimeService runtimeService = **this**.processEngine.getRuntimeService();  String executionId = "2501";  /\*\* 从流程变量中获取汇总当日销售额的值 \*/  Integer value = (Integer) runtimeService//  .getVariable(executionId, "当前的销售额");  System.***out***.println(value);  System.***out***.println("发送短信");  Boolean flag = **false**;  **int** num = 0;  **do** {  flag = send();  num++;  **if** (num == 10) {  System.***out***.println("尝试10次发送。全部失败，已终止发送");  **break**;  }  } **while** (!flag);  /\*\* 向后执行一步，如果流程处于等待状态，使得流程继续执行 \*/  runtimeService.signal(executionId);  System.***out***.println("流程执行完成");  }  **private** Boolean send() {  System.***out***.println("发送成功");  **return** **true**;  }  @Test  **public** **void** doTask() {  RuntimeService runtimeService = **this**.processEngine.getRuntimeService();  String processDefinitionKey = "myProcess";  ProcessInstance processInstance = runtimeService.startProcessInstanceByKey(processDefinitionKey);  System.***out***.println("流程启动成功:" + processInstance.getId() + " " + processInstance.getProcessDefinitionId() + " "  + processInstance.getProcessInstanceId());  /\*\* 使用流程变量设置当日销售额，用来传递业务参数 \*/  **int** value = 0;// 应该是去查询数据库，进行汇总 ---耗时操作  **int** tryNum=0;  **while** (**true**) {  tryNum++;  **try** {  value = **this**.hzxx();  **break**;  } **catch** (Exception e) {  e.printStackTrace();  **if**(tryNum==10) {  System.***out***.println("尝试10次汇总。全部失败，已终止汇总");  **break**;  }  }  }  runtimeService.setVariable(processInstance.getId(), "当前的销售额", value);  /\*\* 向后执行一步，如果流程处于等待状态，使得流程继续执行 \*/  runtimeService.signal(processInstance.getId());    /\*\* 从流程变量中获取汇总当日销售额的值 \*/  Integer saleMoney = (Integer) runtimeService//  .getVariable(processInstance.getId(), "当前的销售额");  System.***out***.println(saleMoney);  System.***out***.println("发送短信");  Boolean flag = **false**;  **int** num = 0;  **do** {  flag = send();  num++;  **if** (num == 10) {  System.***out***.println("尝试10次发送。全部失败，已终止发送");  **break**;  }  } **while** (!flag);  /\*\* 向后执行一步，如果流程处于等待状态，使得流程继续执行 \*/  runtimeService.signal(processInstance.getId());  System.***out***.println("流程执行完成");  }  **public** Integer hzxx() {  // 查询数据库  System.***out***.println("数据汇总中....");  **try** {  Thread.*sleep*(2000);  } **catch** (Exception e) {  e.printStackTrace();  }  System.***out***.println("数据汇总完成");  **return** 10000;  } |

# 16【掌握】个人任务

## 1，流程图

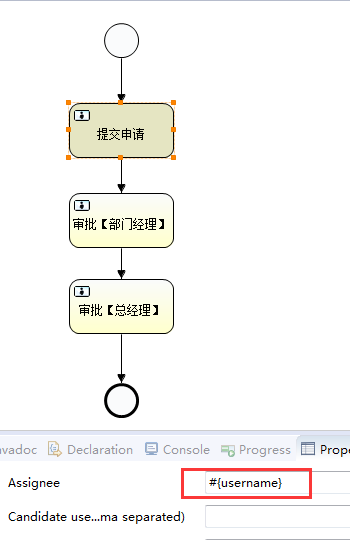


## 2，分配个人任务方式一（直接指定办理人）



缺点 办理人固定了，但是实际开发中，办理人是不固定的

## 3，分配个人任务方式二（使用流程变量）



### 部署

|  |
| --- |
| @Test  **public** **void** deployProcess() {  //得到流程部署的service  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  Deployment deploy = repositoryService.createDeployment().name("请假流程001")  .addClasspathResource("HelloWorld.bpmn")  .addClasspathResource("HelloWorld.png")  .deploy();  System.***out***.println("部署成功:流程部署ID："+deploy.getId());  } |

### 启动并指定下一个任务的办理人【从session里面取】

|  |
| --- |
| /\*\*  \* 启动流程并指定下一个任务的办理人  \*/  @Test  **public** **void** startProcess() {  RuntimeService runtimeService = **this**.processEngine.getRuntimeService();  String processDefinitionKey="HelloWorld";  Map<String,Object> variables=**new** HashMap<>();  variables.put("username", "张三");  runtimeService.startProcessInstanceByKey(processDefinitionKey,variables);  System.***out***.println("流程启动成功");  } |

### 查询个人任务

|  |
| --- |
| /\*\*  \* 查询任务  \*/  @Test  **public** **void** findTask() {  TaskService taskService = **this**.processEngine.getTaskService();  String assignee="张三";  List<Task> list = taskService.createTaskQuery().taskAssignee(assignee).list();  **if**(**null**!=list&&list.size()>0) {  **for** (Task task : list) {  System.***out***.println("任务ID:"+task.getId());  System.***out***.println("流程实例ID:"+task.getProcessInstanceId());  System.***out***.println("执行实例ID:"+task.getExecutionId());  System.***out***.println("流程定义ID:"+task.getProcessDefinitionId());  System.***out***.println("任务名称:"+task.getName());  System.***out***.println("任务办理人:"+task.getAssignee());  System.***out***.println("################################");  }  }  } |

### 完成任务并指定下一个任务的办理人

|  |
| --- |
| /\*\*  \* 办理任务  \*/  @Test  **public** **void** completeTask() {  TaskService taskService = **this**.processEngine.getTaskService();  String taskId="12505";  Map<String,Object> variables=**new** HashMap<>();  variables.put("username", "李四");  taskService.complete(taskId,variables);  System.***out***.println("任务完成");  } |

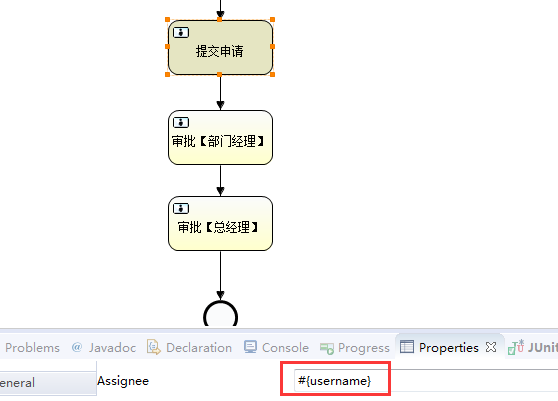
## 4，分配个人任务方式三（使用类）

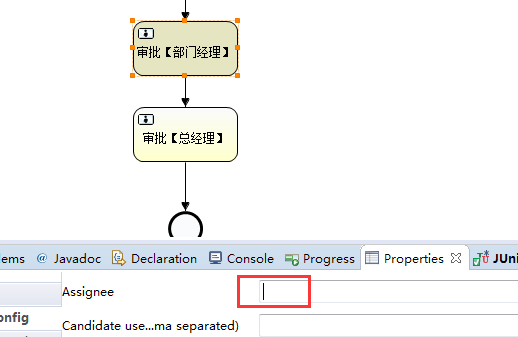
### 创建一个监听类

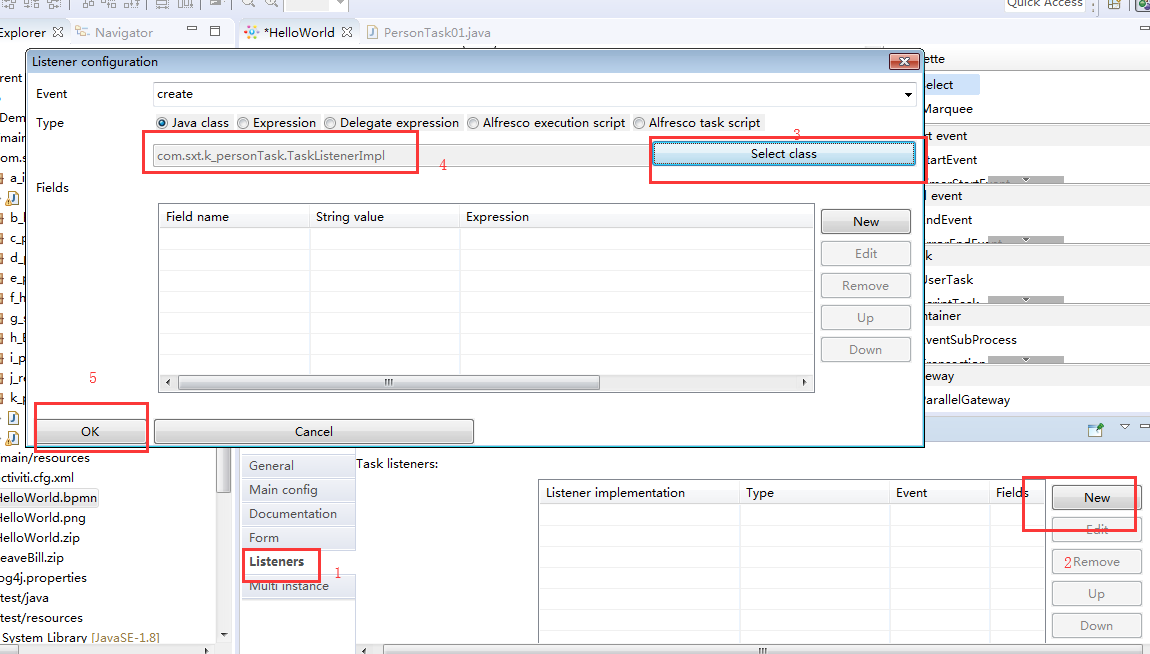
作用：当任务到达相应的节点时，会触发监听，让监听器去指定下个任务的办理人

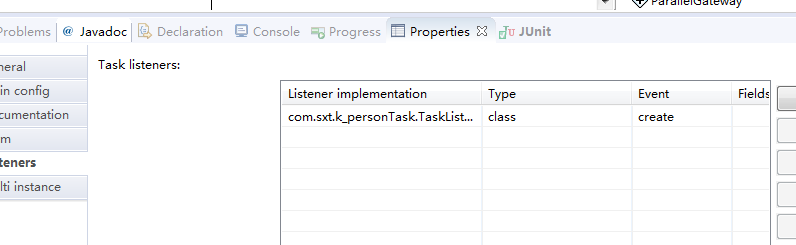


### 修改流程图









## 5，总结

个人任务及三种分配方式：

    1，在taskProcess.bpmn中直接写 assignee=“张三丰"

    2，在taskProcess.bpmn中写 assignee=“#{userID}”，变量的值要是String的。

         使用流程变量指定办理人

    3，使用TaskListener接口，要使类实现该接口，在类中定义，

         delegateTask.setAssignee(assignee);// 指定个人任务的办理人

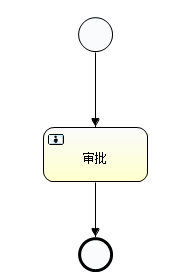
     使用任务ID和办理人重新指定办理人：

     processEngine.getTaskService()//

                           .setAssignee(taskId, userId);

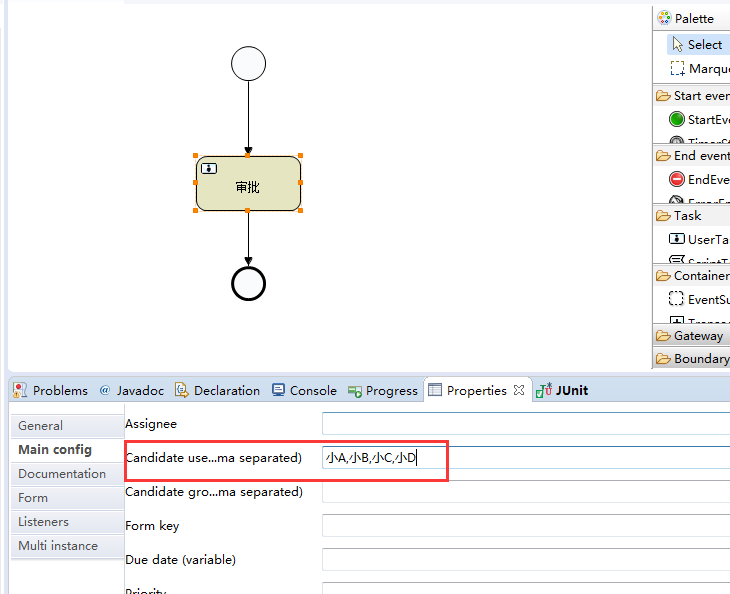
# 18【熟悉】用户组任务

## 1，流程图



## 2，分配组任务方式一（直接指定办理人）

### 修改流程图



### 部署流程

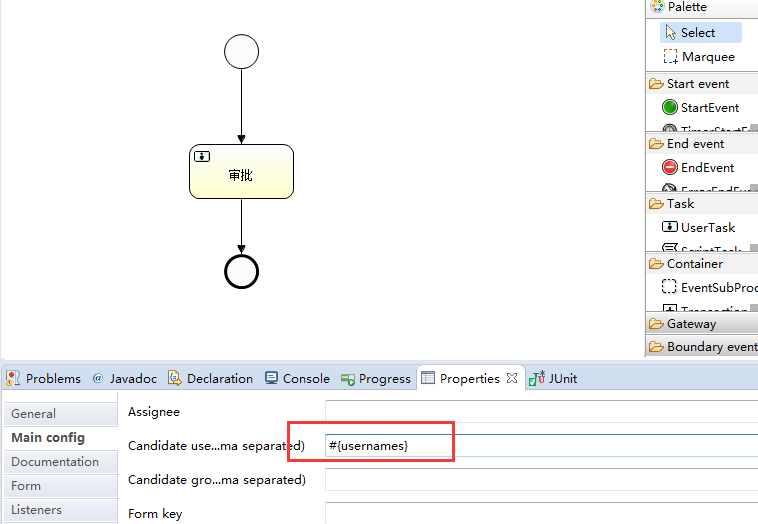
|  |
| --- |
| @Test  **public** **void** deployProcess() {  //得到流程部署的service  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  Deployment deploy = repositoryService.createDeployment().name("请假流程001")  .addClasspathResource("HelloWorld.bpmn")  .addClasspathResource("HelloWorld.png")  .deploy();  System.***out***.println("部署成功:流程部署ID："+deploy.getId());  } |

### 相关代码

|  |
| --- |
| /\*\*  \* 组任务使用流程变量去设置  \*  \* **@author** LJH  \*  \*/  **public** **class** GroupTask01 {  // 得到流程引擎  **private** ProcessEngine processEngine = ProcessEngines.*getDefaultProcessEngine*();  @Test  **public** **void** deployProcess() {  // 得到流程部署的service  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  Deployment deploy = repositoryService.createDeployment().name("请假流程001").addClasspathResource("HelloWorld.bpmn")  .addClasspathResource("HelloWorld.png").deploy();  System.***out***.println("部署成功:流程部署ID：" + deploy.getId());  }  /\*\*  \* 启动流程  \*/  @Test  **public** **void** startProcess() {  RuntimeService runtimeService = **this**.processEngine.getRuntimeService();  String processDefinitionKey = "HelloWorld";  runtimeService.startProcessInstanceByKey(processDefinitionKey);  System.***out***.println("流程启动成功");  }  /\*\*  \* 查询组任务  \*/  @Test  **public** **void** findGroupTask() {  TaskService taskService = **this**.processEngine.getTaskService();  String candidateUser = "小B";  List<Task> list = taskService.createTaskQuery().taskCandidateUser(candidateUser).list();  **if** (**null** != list && list.size() > 0) {  **for** (Task task : list) {  System.***out***.println("任务ID:" + task.getId());  System.***out***.println("流程实例ID:" + task.getProcessInstanceId());  System.***out***.println("执行实例ID:" + task.getExecutionId());  System.***out***.println("流程定义ID:" + task.getProcessDefinitionId());  System.***out***.println("任务名称:" + task.getName());  System.***out***.println("任务办理人:" + task.getAssignee());  System.***out***.println("################################");  }  }  }  /\*\*  \* 查询任务  \*/  @Test  **public** **void** findTask() {  TaskService taskService = **this**.processEngine.getTaskService();  String assignee = "小A";  List<Task> list = taskService.createTaskQuery().taskAssignee(assignee).list();  **if** (**null** != list && list.size() > 0) {  **for** (Task task : list) {  System.***out***.println("任务ID:" + task.getId());  System.***out***.println("流程实例ID:" + task.getProcessInstanceId());  System.***out***.println("执行实例ID:" + task.getExecutionId());  System.***out***.println("流程定义ID:" + task.getProcessDefinitionId());  System.***out***.println("任务名称:" + task.getName());  System.***out***.println("任务办理人:" + task.getAssignee());  System.***out***.println("################################");  }  }  }  /\*\*  \* 任务拾取  \*/  @Test  **public** **void** claim() {  String taskId="2504";  TaskService taskService = **this**.processEngine.getTaskService();  taskService.claim(taskId, "小A");  System.***out***.println("任务拾取成功");  }  /\*\*  \* 任务回退  \*/  @Test  **public** **void** claimBack() {  String taskId="2504";  TaskService taskService = **this**.processEngine.getTaskService();  taskService.setAssignee(taskId, **null**);  System.***out***.println("任务回退成功");  }    //5 查询组任务成员列表  @Test  **public** **void** findGroupUser(){  String taskId = "2504";  List<IdentityLink> list = processEngine.getTaskService()//  .getIdentityLinksForTask(taskId);  //List<IdentityLink> list = processEngine.getRuntimeService()//  // .getIdentityLinksForProcessInstance(instanceId);  **for**(IdentityLink identityLink:list ){  System.***out***.println("userId="+identityLink.getUserId());  System.***out***.println("taskId="+identityLink.getTaskId());  System.***out***.println("piId="+identityLink.getProcessInstanceId());  System.***out***.println("TYPE="+identityLink.getType());  System.***out***.println("######################");  }  }        /\*\*  \* 办理任务  \*/  @Test  **public** **void** completeTask() {  TaskService taskService = **this**.processEngine.getTaskService();  String taskId = "5002";  taskService.complete(taskId);  System.***out***.println("任务完成");  }  } |

## 3，分配组任务方式二（使用流程变量）

### 修改流程图

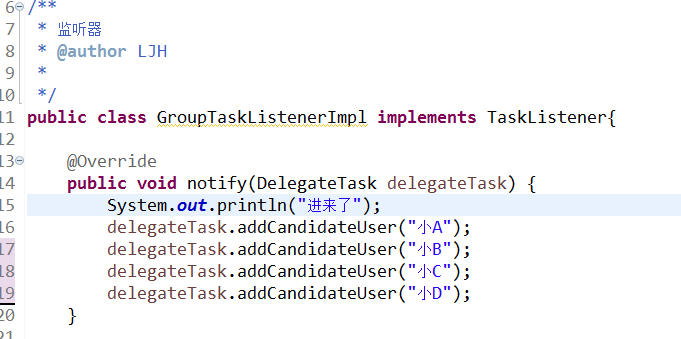


### 相关代码

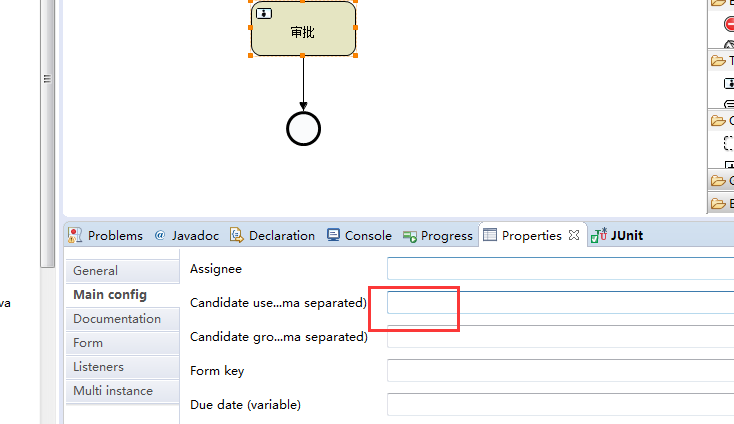
|  |
| --- |
| /\*\*  \* 组任务使用流程变量去设置  \*  \* **@author** LJH  \*  \*/  **public** **class** GroupTask01 {  // 得到流程引擎  **private** ProcessEngine processEngine = ProcessEngines.*getDefaultProcessEngine*();  @Test  **public** **void** deployProcess() {  // 得到流程部署的service  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  Deployment deploy = repositoryService.createDeployment().name("请假流程001").addClasspathResource("HelloWorld.bpmn")  .addClasspathResource("HelloWorld.png").deploy();  System.***out***.println("部署成功:流程部署ID：" + deploy.getId());  }  /\*\*  \* 启动流程  \*/  @Test  **public** **void** startProcess() {  RuntimeService runtimeService = **this**.processEngine.getRuntimeService();  String processDefinitionKey = "HelloWorld";  Map<String,Object> variables=**new** HashMap<>();  variables.put("usernames", "小A,小B,小C,小D");  runtimeService.startProcessInstanceByKey(processDefinitionKey,variables);  System.***out***.println("流程启动成功");  }  /\*\*  \* 查询组任务  \*/  @Test  **public** **void** findGroupTask() {  TaskService taskService = **this**.processEngine.getTaskService();  String candidateUser = "小B";  List<Task> list = taskService.createTaskQuery().taskCandidateUser(candidateUser).list();  **if** (**null** != list && list.size() > 0) {  **for** (Task task : list) {  System.***out***.println("任务ID:" + task.getId());  System.***out***.println("流程实例ID:" + task.getProcessInstanceId());  System.***out***.println("执行实例ID:" + task.getExecutionId());  System.***out***.println("流程定义ID:" + task.getProcessDefinitionId());  System.***out***.println("任务名称:" + task.getName());  System.***out***.println("任务办理人:" + task.getAssignee());  System.***out***.println("################################");  }  }  }  /\*\*  \* 查询任务  \*/  @Test  **public** **void** findTask() {  TaskService taskService = **this**.processEngine.getTaskService();  String assignee = "小A";  List<Task> list = taskService.createTaskQuery().taskAssignee(assignee).list();  **if** (**null** != list && list.size() > 0) {  **for** (Task task : list) {  System.***out***.println("任务ID:" + task.getId());  System.***out***.println("流程实例ID:" + task.getProcessInstanceId());  System.***out***.println("执行实例ID:" + task.getExecutionId());  System.***out***.println("流程定义ID:" + task.getProcessDefinitionId());  System.***out***.println("任务名称:" + task.getName());  System.***out***.println("任务办理人:" + task.getAssignee());  System.***out***.println("################################");  }  }  }  /\*\*  \* 任务拾取  \*/  @Test  **public** **void** claim() {  String taskId="2505";  TaskService taskService = **this**.processEngine.getTaskService();  taskService.claim(taskId, "小A");  System.***out***.println("任务拾取成功");  }  /\*\*  \* 任务回退  \*/  @Test  **public** **void** claimBack() {  String taskId="2504";  TaskService taskService = **this**.processEngine.getTaskService();  taskService.setAssignee(taskId, **null**);  System.***out***.println("任务回退成功");  }    //5 查询组任务成员列表  @Test  **public** **void** findGroupUser(){  String taskId = "2504";  List<IdentityLink> list = processEngine.getTaskService()//  .getIdentityLinksForTask(taskId);  //List<IdentityLink> list = processEngine.getRuntimeService()//  // .getIdentityLinksForProcessInstance(instanceId);  **for**(IdentityLink identityLink:list ){  System.***out***.println("userId="+identityLink.getUserId());  System.***out***.println("taskId="+identityLink.getTaskId());  System.***out***.println("piId="+identityLink.getProcessInstanceId());  System.***out***.println("TYPE="+identityLink.getType());  System.***out***.println("######################");  }  }        /\*\*  \* 办理任务  \*/  @Test  **public** **void** completeTask() {  TaskService taskService = **this**.processEngine.getTaskService();  String taskId = "5002";  taskService.complete(taskId);  System.***out***.println("任务完成");  }  } |

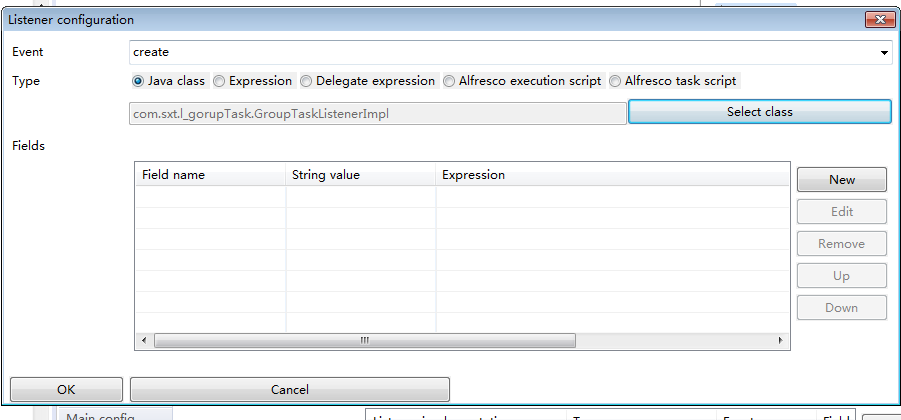
## 4，分配组任务方式三（使用类）

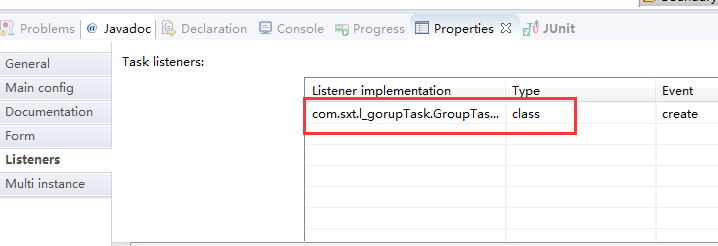
### 创建监听器



### 修改流程图



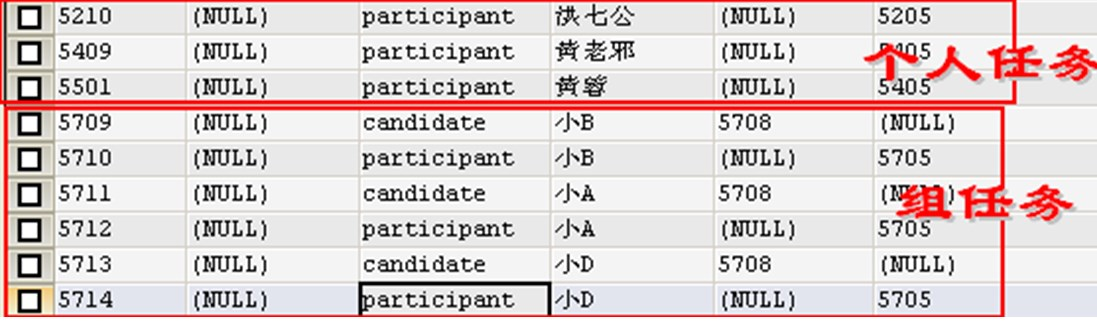




### 相关代码

|  |
| --- |
| /\*\*  \* 组任务使用监听器去设置  \*  \* **@author** LJH  \*  \*/  **public** **class** GroupTask01 {  // 得到流程引擎  **private** ProcessEngine processEngine = ProcessEngines.*getDefaultProcessEngine*();  @Test  **public** **void** deployProcess() {  // 得到流程部署的service  RepositoryService repositoryService = **this**.processEngine.getRepositoryService();  Deployment deploy = repositoryService.createDeployment().name("请假流程001").addClasspathResource("HelloWorld.bpmn")  .addClasspathResource("HelloWorld.png").deploy();  System.***out***.println("部署成功:流程部署ID：" + deploy.getId());  }  /\*\*  \* 启动流程  \*/  @Test  **public** **void** startProcess() {  RuntimeService runtimeService = **this**.processEngine.getRuntimeService();  String processDefinitionKey = "HelloWorld";  runtimeService.startProcessInstanceByKey(processDefinitionKey);  System.***out***.println("流程启动成功");  }  /\*\*  \* 查询组任务  \*/  @Test  **public** **void** findGroupTask() {  TaskService taskService = **this**.processEngine.getTaskService();  String candidateUser = "小B";  List<Task> list = taskService.createTaskQuery().taskCandidateUser(candidateUser).list();  **if** (**null** != list && list.size() > 0) {  **for** (Task task : list) {  System.***out***.println("任务ID:" + task.getId());  System.***out***.println("流程实例ID:" + task.getProcessInstanceId());  System.***out***.println("执行实例ID:" + task.getExecutionId());  System.***out***.println("流程定义ID:" + task.getProcessDefinitionId());  System.***out***.println("任务名称:" + task.getName());  System.***out***.println("任务办理人:" + task.getAssignee());  System.***out***.println("################################");  }  }  }  /\*\*  \* 查询任务  \*/  @Test  **public** **void** findTask() {  TaskService taskService = **this**.processEngine.getTaskService();  String assignee = "小A";  List<Task> list = taskService.createTaskQuery().taskAssignee(assignee).list();  **if** (**null** != list && list.size() > 0) {  **for** (Task task : list) {  System.***out***.println("任务ID:" + task.getId());  System.***out***.println("流程实例ID:" + task.getProcessInstanceId());  System.***out***.println("执行实例ID:" + task.getExecutionId());  System.***out***.println("流程定义ID:" + task.getProcessDefinitionId());  System.***out***.println("任务名称:" + task.getName());  System.***out***.println("任务办理人:" + task.getAssignee());  System.***out***.println("################################");  }  }  }  /\*\*  \* 任务拾取  \*/  @Test  **public** **void** claim() {  String taskId="2505";  TaskService taskService = **this**.processEngine.getTaskService();  taskService.claim(taskId, "小A");  System.***out***.println("任务拾取成功");  }  /\*\*  \* 任务回退  \*/  @Test  **public** **void** claimBack() {  String taskId="2504";  TaskService taskService = **this**.processEngine.getTaskService();  taskService.setAssignee(taskId, **null**);  System.***out***.println("任务回退成功");  }    //5 查询组任务成员列表  @Test  **public** **void** findGroupUser(){  String taskId = "2504";  List<IdentityLink> list = processEngine.getTaskService()//  .getIdentityLinksForTask(taskId);  //List<IdentityLink> list = processEngine.getRuntimeService()//  // .getIdentityLinksForProcessInstance(instanceId);  **for**(IdentityLink identityLink:list ){  System.***out***.println("userId="+identityLink.getUserId());  System.***out***.println("taskId="+identityLink.getTaskId());  System.***out***.println("piId="+identityLink.getProcessInstanceId());  System.***out***.println("TYPE="+identityLink.getType());  System.***out***.println("######################");  }  }        /\*\*  \* 办理任务  \*/  @Test  **public** **void** completeTask() {  TaskService taskService = **this**.processEngine.getTaskService();  String taskId = "5002";  taskService.complete(taskId);  System.***out***.println("任务完成");  }  } |

## 5，总结



组任务及三种分配方式，

    1，在taskProcess.bpmn中直接写 candidate-users=“小A,小B,小C,小D"

    2，在taskProcess.bpmn中写 candidate-users =“#{userIDs}”，变量的值要是String的。

         使用流程变量指定办理人

              Map<String, Object> variables = new HashMap<String, Object>();

              variables.put("userIDs", "大大,小小,中中");

    3，使用TaskListener接口，使用类实现该接口，在类中定义，

            //添加组任务的用户

delegateTask.addCandidateUser(userId1);

delegateTask.addCandidateUser(userId2);

组任务分配给个人任务（认领任务），

     processEngine.getTaskService().claim(taskId, userId);

个人任务分配给组任务，

     processEngine.getTaskService(). setAssignee(taskId, null);

向组任务添加人员，

     processEngine.getTaskService().addCandidateUser(taskId, userId);

向组任务删除人员，

     processEngine.getTaskService().deleteCandidateUser(taskId, userId);

个人任务和组任务存放办理人对应的表，

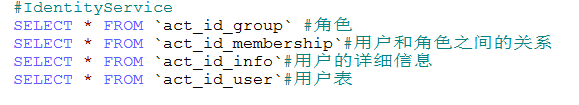
act\_ru\_identitylink表存放任务的办理人，包括个人任务和组任务，表示正在执行的任务

act\_hi\_identitylink表存放任务的办理人，包括个人任务和组任务，表示历史任务

区别在于，如果是个人任务TYPE的类型表示participant（参与者）

         如果是组任务TYPE的类型表示candidate（候选者）和participant（参与者）

# 19【了解】工作流定义的角色组



|  |
| --- |
| /\*\*  \* 用户组的处理  \* **@author** LJH  \*  \*/  **public** **class** TestUserGroup {  **private** ProcessEngine processEngine=ProcessEngines.*getDefaultProcessEngine*();      /\*\*  \* 创建用户和用户组  \*/  @Test  **public** **void** createUserAndGroup() {  IdentityService identityService = **this**.processEngine.getIdentityService();    //保存到act\_id\_group  GroupEntity group = **new** GroupEntity("1");  group.setName("部门经理");  identityService.saveGroup(group);  GroupEntity group2 = **new** GroupEntity("2");  group2.setName("总经理");  identityService.saveGroup(group2);  //保存到act\_id\_user  UserEntity user1 = **new** UserEntity("1");  user1.setFirstName("小明");  user1.setEmail("11111@aa.com");  identityService.saveUser(user1);  UserEntity user2 = **new** UserEntity("2");  user2.setFirstName("小张");  identityService.saveUser(user2);  UserEntity user3 = **new** UserEntity("3");  user3.setFirstName("小王");  identityService.saveUser(user3);  //建立用户和用户组的关系act\_id\_membership  identityService.createMembership("1", "1");  identityService.createMembership("2", "1");  identityService.createMembership("3", "2");  System.***out***.println("保存完成");  }  } |

# 20【掌握】Activiti总结

