工作流-Activiti进阶

# 流程操作的环境准备和回顾

## Activiti和Spring的整合

通过坐标导入Jar

|  |
| --- |
| <activiti.version>5.19.0.2</activiti.version>  <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> |

applicationContext.xml:

|  |
| --- |
| <!-- 引入外部属性文件 -->  <context:property-placeholder location=*"classpath:db.properties"* />  <!-- 连接池 -->  <bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>  <property name=*"driverClass"* value=*"${jdbc.driver}"* />  <property name=*"jdbcUrl"* value=*"${jdbc.url}"* />  <property name=*"user"* value=*"${jdbc.username}"* />  <property name=*"password"* value=*"${jdbc.password}"* />  </bean>  <!-- 流程引擎配置对象:activit与spring整合的配置 -->  <bean id=*"processEngineConfiguration"* class=*"org.activiti.spring.SpringProcessEngineConfiguration"*>  <!-- 注入数据源 -->  <property name=*"dataSource"* ref=*"dataSource"*/>  <!-- 自动建表 -->  <property name=*"databaseSchemaUpdate"* value=*"true"*/>  <!-- 字体 -->  <property name=*"activityFontName"* value=*"宋体"*/>  <property name=*"labelFontName"* value=*"宋体"*/>  <!-- 注入事务管理器 -->  <property name=*"transactionManager"* ref=*"transactionManager"*/>  </bean>  <!-- 事务管理器 -->  <bean id=*"transactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>  <property name=*"dataSource"* ref=*"dataSource"* />  </bean>  <!-- 流程引擎对象：spring整合方式 -->  <bean id=*"processEngine"* class=*"org.activiti.spring.ProcessEngineFactoryBean"*>  <property name=*"processEngineConfiguration"* ref=*"processEngineConfiguration"* />    <!-- 流程控制的相关service -->  <!-- 仓库对象 -->  <bean id=*"repositoryService"* factory-bean=*"processEngine"* factory-method=*"getRepositoryService"*/>  <!-- 运行时Service -->  <bean id=*"runtimeService"* factory-bean=*"processEngine"* factory-method=*"getRuntimeService"*/>  <!-- 任务Service -->  <bean id=*"taskService"* factory-bean=*"processEngine"* factory-method=*"getTaskService"*/> |

## 入门操作回顾

1．流程定义文档图（略）

2．发布流程定义（代码）

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.**class**)  @ContextConfiguration(locations="classpath:applicationContext.xml")  //Spring的junit与事务整合，自动开启了事务注解驱动，默认事务是自动回滚  @TransactionConfiguration(transactionManager = "transactionManager", defaultRollback = **false**)  @Transactional  **public** **class** ActivitiTest {  //仓库管理对象（主要用来管理资源，发布流程定义相关）  @Autowired  **private** RepositoryService repositoryService;  //运行时service  @Autowired  **private** RuntimeService runtimeService;  //任务操作的service  @Autowired  **private** TaskService taskService;    //发布流程  @Test  **public** **void** deploy(){  repositoryService.createDeployment()  .addClasspathResource("diagrams/Leave.bpmn")  .name("流程的名字")  .deploy();  System.***out***.println("流程发布成功！");  }    //可以查询流程定义列表  **public** **void** queryProcessDefinition(){  List<ProcessDefinition> list = repositoryService.createProcessDefinitionQuery()  .list();  **for** (ProcessDefinition processDefinition : list) {  System.***out***.println("流程定义的id："+processDefinition.getId());  System.***out***.println("流程定义的key："+processDefinition.getKey());//画图id  System.***out***.println("流程定义的名字："+processDefinition.getName());  }    }    //启动流程  @Test  **public** **void** startProcee(){  runtimeService.startProcessInstanceById("1232");  }    //查询个人任务(代办)  @Test  **public** **void** queryPersonTask(){  List<Task> list = taskService.createTaskQuery()  .taskAssignee("查询的办理人")//自己方进去  .list();  **for** (Task task : list) {  System.***out***.println("任务编号："+task.getId());  System.***out***.println("任务名字："+task.getName());  System.***out***.println("任务归属的流程实例id："+task.getProcessInstanceId());  System.***out***.println("任务归属的流程定义的id："+task.getProcessDefinitionId());  }  }    //完成办理个人任务  @Test  **public** **void** completeTask(){  taskService.complete("任务编号");  System.***out***.println("任务办理成功！");  } |

扩展：删除表数据

为了方便操作，可以自己写个存储过程：



|  |
| --- |
|  |

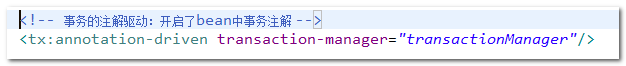
【小知识点1】：

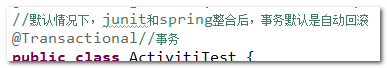
存储过程中执行字符串的语句，使用execute immediate 字符串

【小知识点2】：

junit和spring整合的事务回滚解决方案：

方案1：使用@Rollback注解







方案2：使用了集成事务的方案，设置默认是否回滚

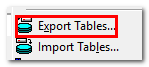


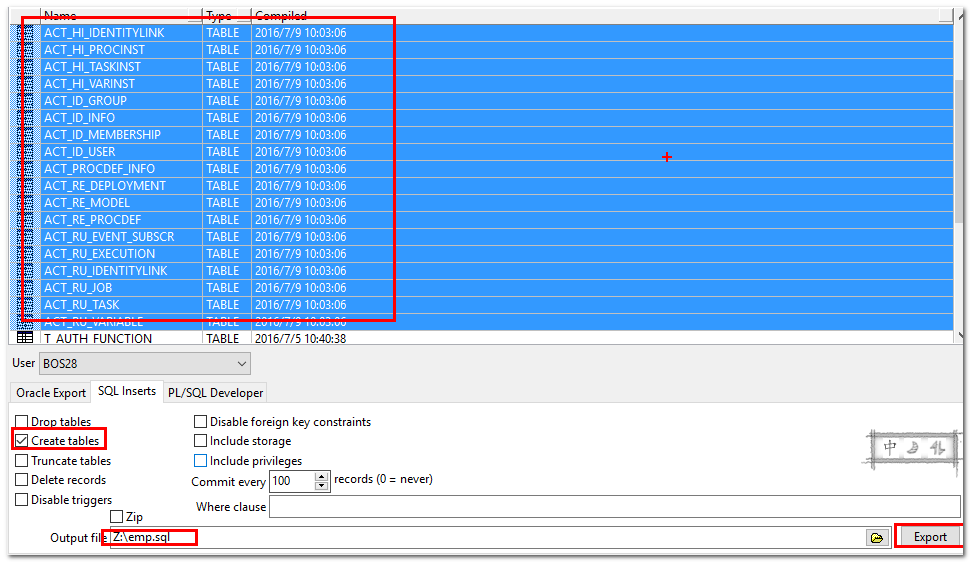
扩展：

表结构反转为pd 数据模型图

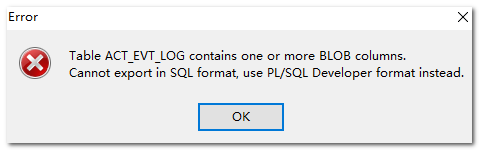
方案1：

先导出sql，然后使用sql反转为模型图





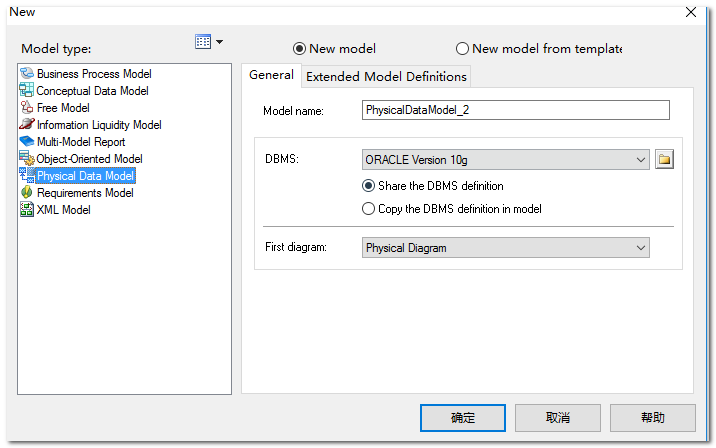
出现该错误，是正常的：

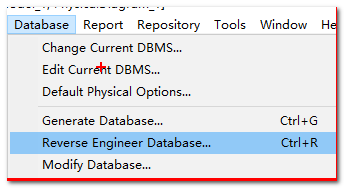


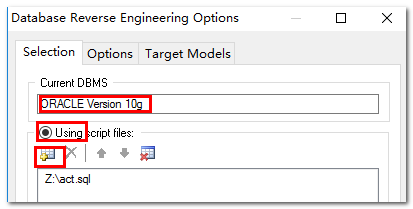
因为blog是二进制的，无法导出为sql的文本。

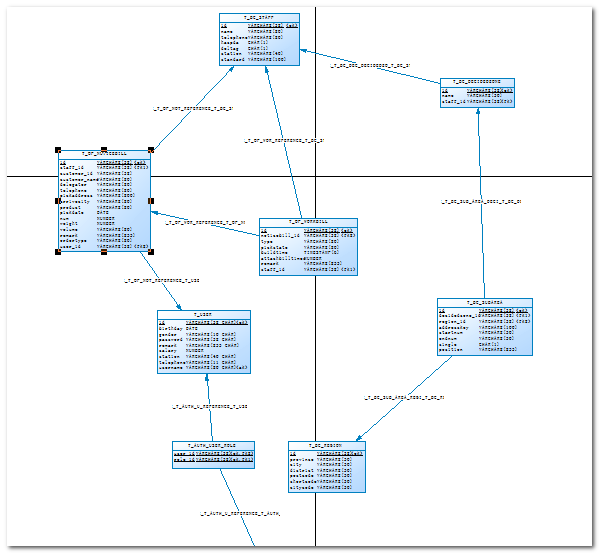


pd中新建数据模型：







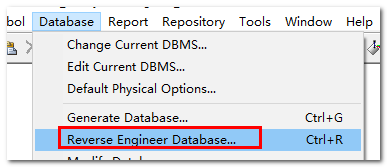


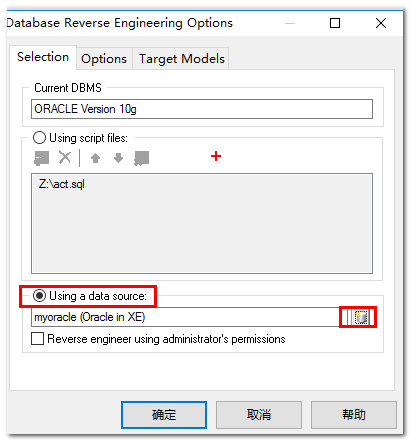
方法2：

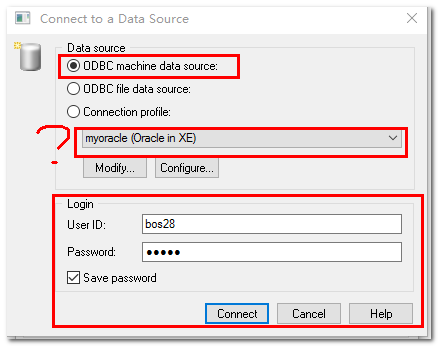
建模工具pd直接连接数据库，反转模型。

新建空的数据模型：



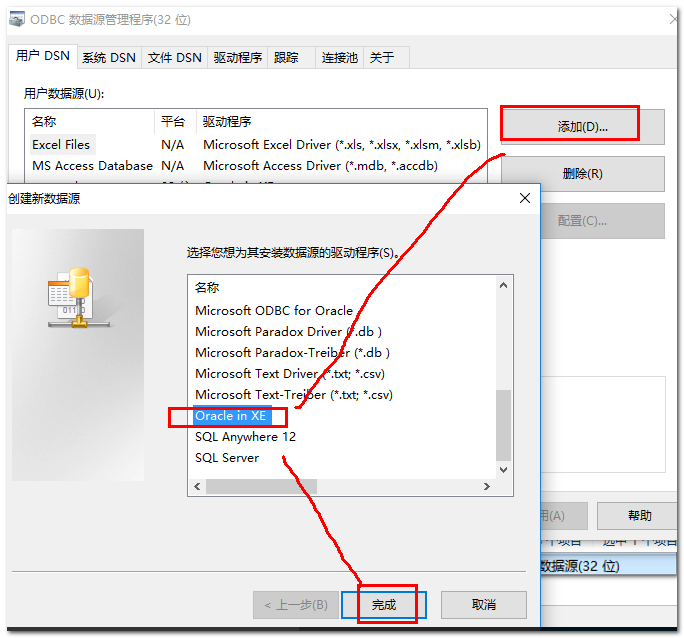


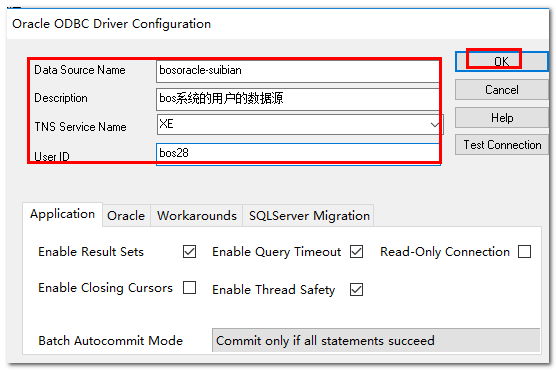


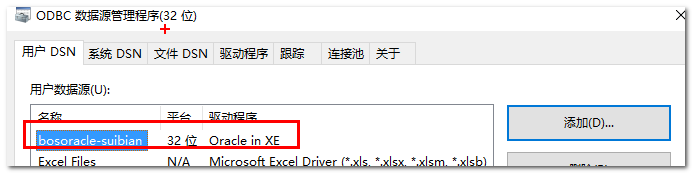


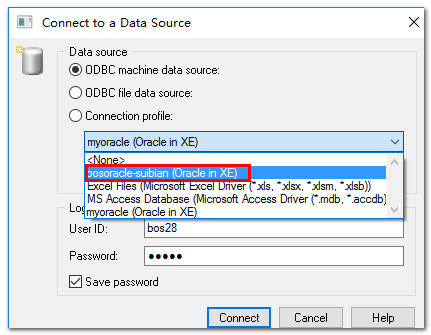
如何配置添加一个ODBC的数据源：

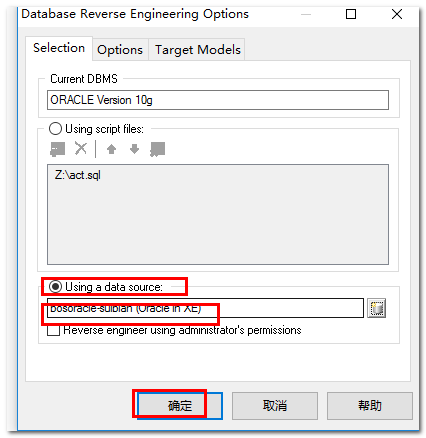




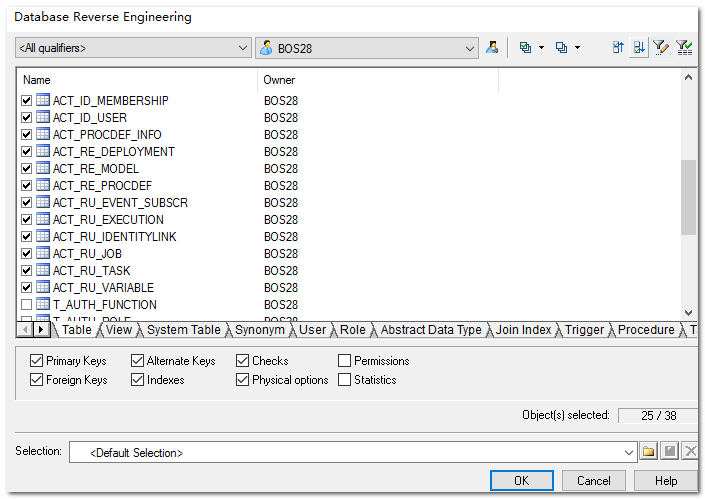


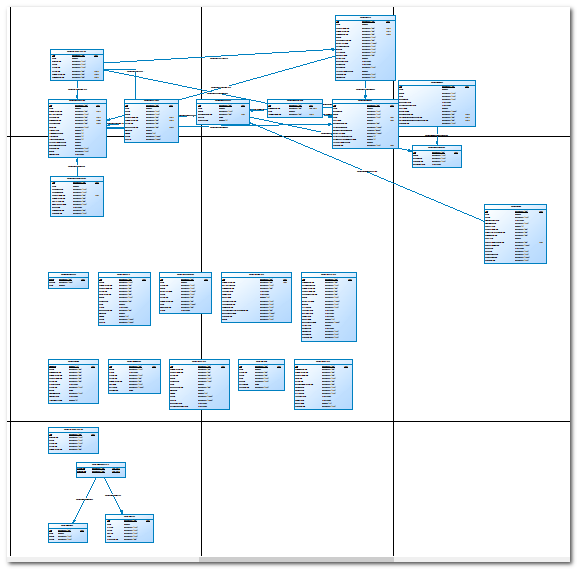






选择要反转的表：





# 流程变量管理

流程变量非常重要，临时存储一些变量，该变量一般是与具体业务相关的，也可以是任意想保存的，比如后面用流程跳转，也无需要流程变量；

流程变量就是工作流对业务进行交流的一个途径，窗口。

早期，非常的重要，尤其是JBPM。

但，Activiti逐步在弱化流程变量，它提供了更多的api，可以满足你的业务需要。

用法：将变量存在流程中，在某一个节点可以取出来。

## 流程变量的基本使用

两种基本的保存方式：

* 一种是在启动流程的时候保存了变量和值，比如，保存了一个业务表的主键id。
* 一种是在任务运行中保存了变量和值，比如保存一些临时业务信息以及判断连线走向的信息。

取值：可以在任意的时候取出（说明：必须是当前流程实例中）。

启动流程的同时保存变量

|  |
| --- |
| //流程变量  //在启动流程的时候同时保存一个变量  @Test  **public** **void** setVarOnStart(){  //参数2：就是流程变量，结果是map，里面是key和value  Map<String, Object> variables = **new** HashMap<>();  //保存公司的名字  variables.put("company", "传智播客");  //。。。。放入nduo的变量--全局的变量  runtimeService.startProcessInstanceByKey("LeaveProcess", variables);  } |

任务执行过程中随时保存变量：

|  |
| --- |
| //在任务流转过程中设置变量  @Test  **public** **void** setVarOnTask(){  //某个任务，从数据库查询出来一个值，根据这个值，判断 是否走下一个节点  // taskService.setVariable(taskId, variableName, value);//key:value  // taskService.setVariables(taskId, variables);//map    //某个任务节点设置一个变量：(全局的变量)  taskService.setVariable("2505", "age", 18);  taskService.setVariable("2505", "name", "Rose");  taskService.setVariable("2505", "birthday", **new** Date());  taskService.setVariable("2505", "简历", **new** User(1001,"Jack","123456"));  } |

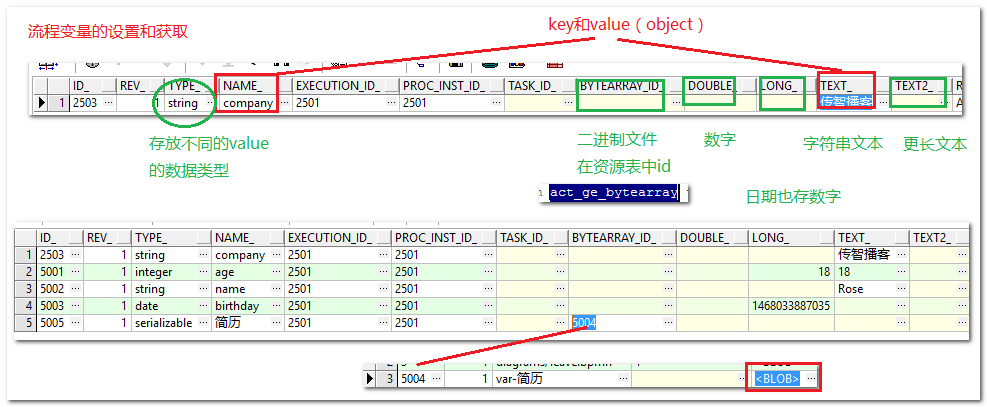
获取流程变量（使用运行时和任务api均可）

|  |
| --- |
| //获取流程变量  @Test  **public** **void** getVar(){  //在流程运行过程中随时可以取出，任意的变量，那条线中的变量  //参数1：执行id，参数2：变量名字  String company =(String) runtimeService.getVariable("2501", "company");  System.***out***.println(company);  //在任意的任务中获取：该任务可拥有的  //参数1：任务id，参数2：变量的名字  String company2 =(String)taskService.getVariable("2505", "company");  System.***out***.println(company2);    System.***out***.println("------");    System.***out***.println(runtimeService.getVariable("2501", "name"));  System.***out***.println(runtimeService.getVariable("2501", "age"));  System.***out***.println(runtimeService.getVariable("2501", "birthday"));  System.***out***.println(runtimeService.getVariable("2501", "简历"));//反序列化回来了  } |

涉及到的两张表：

|  |
| --- |
| select t.\*, t.rowid from act\_ru\_variable t;*--正在运行流程变量的表*  select t.\*, t.rowid from act\_hi\_varinst t;*--历史的流程变量表* |

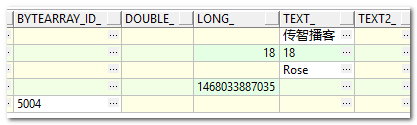
数据表的变化



扩展：

## 如何在关系型数据库中设计具有一定扩展性的表结构呢？

思路：冗余字段



上述的容易，提前已经知道有哪些类型，我程序中直接放入对应的字段就可以了。

问题：不知道将来会添加什么字段？

如我设计一个人员表，开始的时候，只有3个字段：

id username password 当时已经可以满足需要。

随着业务的扩展，需要加字段，age，。。。

解决：可以修改表，添加字段。

但有个弊端，对于持久层框架来说，添加字段可能意味着要更改持久层数据结构（po等）。

影响有点大，而且不便于维护扩展。

我们在设计数据表：

|  |  |  |
| --- | --- | --- |
| id | Number | 编号 |
| username | Varchar2(50) | 用户名 |
| password | Varchar2(50) | 密码 |
| dyVarchar1 | Varchar2(255) | 动态字段1 |
| dVarchar2 | Varchar2(10) | 动态字段1 |
| dVarchar3 | Varchar2(50) | 动态字段1 |
| dVarchar4 | Varchar2(100) | 动态字段1 |
| dVarchar5 | Varchar2(200) | 动态字段1 |
|  |  |  |

缺点：字段不容易识别！必须有较好的文档记录。

否则只能通过程序猜。

现在NoSQL已经不存在这个问题了。

只有行，没有列。

MongoDB：一张表，存任意长度的数据

1:username

2:username password,age

3:age

4:birthday, memo,/./…..

# 连线—SequenceFlow

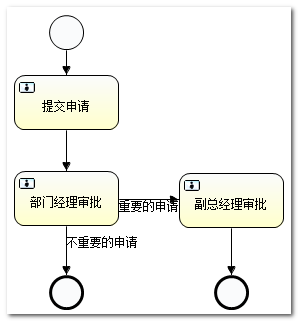
连线的作用：告诉工作流引擎，流程该怎么走。

## 编写画流程图

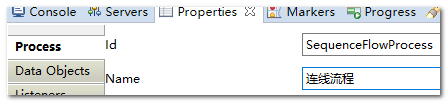
对于工作流系统，开始节点必须且只能有一个，结束节点必须有但可以有多个。

换句话：流程的发起，只能是从一个点。结束，可以有多个。

大部分流程都是一个开始，一个结束。一般是谁申请，谁最后确认。

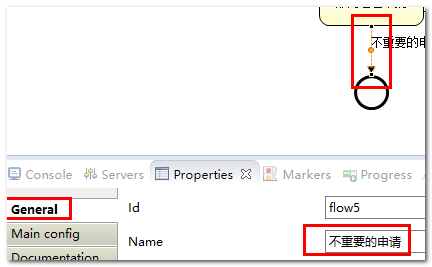


设置流程的信息：



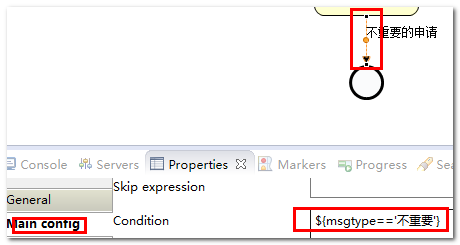
设置三个节点的审批人分别是张三、李四、王五。

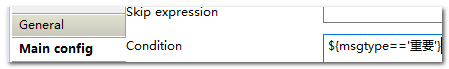
更改连线的名称：



设置连线的条件：

在Condition中设置一个布尔值，里面的表达式可以用${}或者#{}，如：${msgtype=='不重要'}





Bpmn的变化

## 测试流程

部署流程：（略）

启动流程：（略）

完成个人任务：

|  |
| --- |
| //个人任务办理  @Test  **public** **void** completePersonTask2(){  Map<String, Object> variables=**new** HashMap<String, Object>();  variables.put("msgtype", "重要");  taskService.complete("5002", variables);  } |

要点：使用流程变量控制连线的走向。

缺点：不容易清晰的描绘出分支的走向，尤其是并行或排他业务，得靠大量变量来判断。

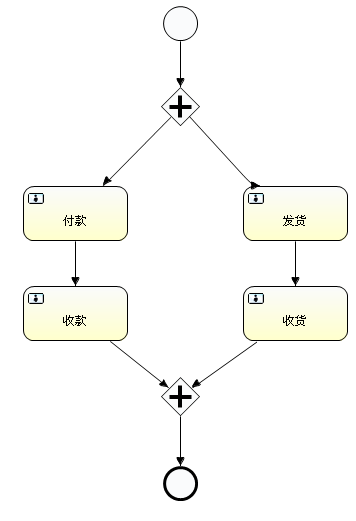
# 网关—Gateway

主要是用来控制分支流程的。并行（必须同时走多根），排他的（只能走一根），包含的走法（可以走多根，也可以走一根）



## 并行网关—Parallel Gateway

### 编写画流程图



这种方式在政府业务中叫，并联审批中的并行。

并行网关，必须是成对出现的。因为它要控制分支和汇聚。

### 测试流程

部署流程定义

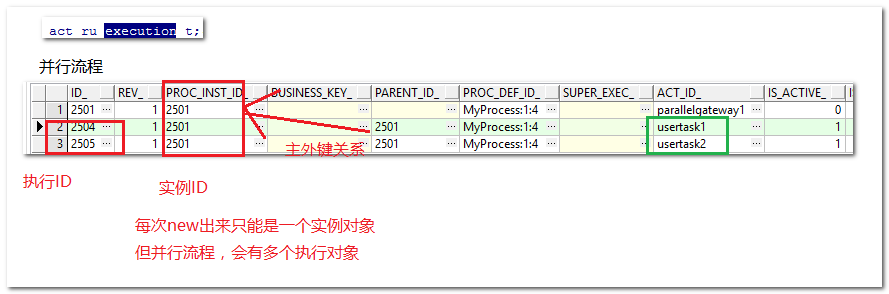
发布流程定义

查询个人任务

完成个人任务

### 小结

1. 一个流程中流程实例只有1个（启动一次，只能生成一个实例），执行对象有多个（有分支，同时在跑）



活动：任意的一个节点都是活动。包含用户任务。

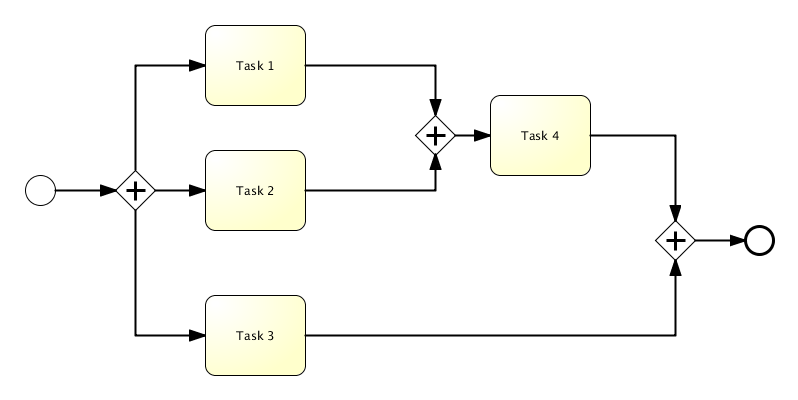
用户任务：方块（用户任务，带小人，其他的任务，工作流不认为是用户任务），

1. 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（执行对象）。

**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。（只有全部汇聚后，流程才能真正通过并行网关）

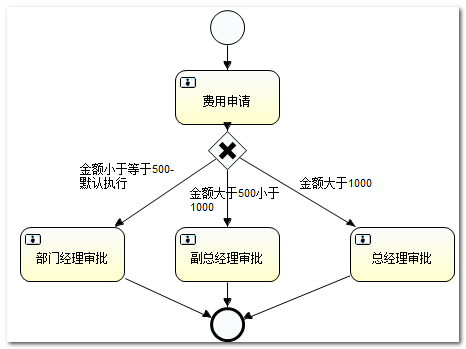
1. 并行网关的进入和外出都是使用相同节点标识（jbpm是用了两种节点图标）
2. 如果同一个并行网关有多个进入和多个外出顺序流， 它就同时具有分支和汇聚功能。 这时，网关会先汇聚所有进入的顺序流，然后再切分成多个并行分支。
3. 并行网关不会解析条件。 即使顺序流中定义了条件，也会被忽略。
4. 并行网关不需要是“平衡的”（比如， 对应并行网关的进入和外出节点数目不一定相等）。如图中标示是合法的：



## 排他网关—Exclusive Gateway

无论如何，只会走一条线。

### 编写画流程图



设置流程定义的key：

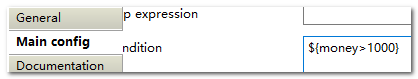


设置办理人分别为张三、李四、王五。

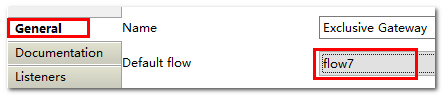
设置连线的名称，（略）

设置连线的条件：





选择排他网关，设置默认的连线：



排他网关和直接连线的最大区别：排他网关有个默认值，连线必须是每根先都得给条件。

排他网关必须给流程变量，当判断结果没有符合条件的话，就会走默认的连线。不给流程变量，会报错，因为无法判断。

连线：如果给流程变量，按照流程变量的判断走，没有默认指定的连线。可以不给流程变量，也会走个默认（先画的那根线，不容易控制）

两者根据实际情况，都可以选择。

### 测试流程

部署流程：（略）

启动流程：（略）

完成个人任务：

|  |
| --- |
| //个人任务办理  @Test  **public** **void** completePersonTask2(){  Map<String, Object> variables=**new** HashMap<String, Object>();  variables.put("money", 800);  taskService.complete("2504", variables);  } |

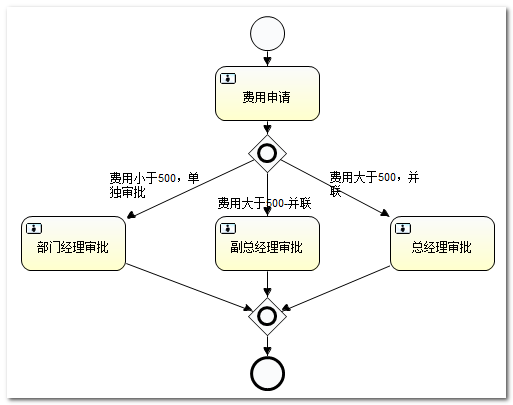
## 包含网关—Inclusive Gateway

相当于并行+排他。

inclusiveGateway与 exclusiveGateway的区别

* exclusiveGateway 只会寻找唯一一条能走完的flow，也就是说当有一个flow可以走通的情况下，它不会再次去寻找第二条可以走通的flow ，如是没有符合条件的，就走defalut的flow。
* inclusiveGateway 会寻找所有符合条件的 flow，也就是说他会走完所有的符合条件的flow，如果没有符合的，那么就去走defalut的flow

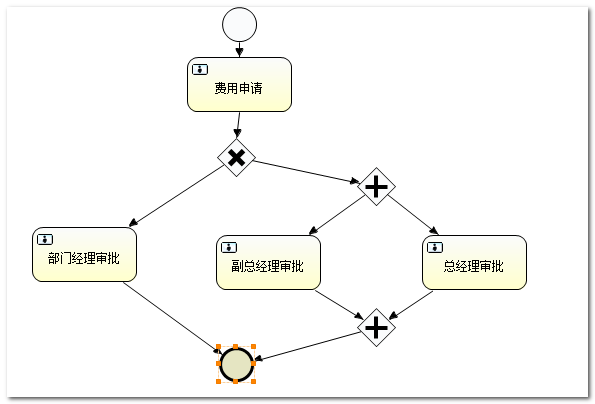
### 编写画流程图





小结：

提示：包含网关可以使用并行+排他来代替。



# 任务—Task

## 常用任务

### 任务分析





常用的task，有如下几个：

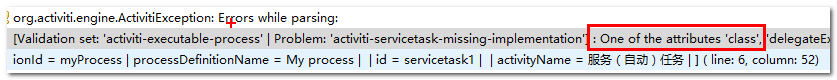
* User Task：用户任务，需由用户（参与者、必须有办理人）来完成任务。



* Java Service Task：服务任务，也称之为自动节点任务，当运行到该类型的任务节点的时候，会自动执行一个Java类（需实现一个接口）的一个方法。（发短信、处理数据等）



如果不配置类：

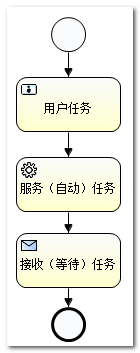


* Jave Receive Task：接收任务，也称之为等待任务，需手动来完成任务（调一个方法）。

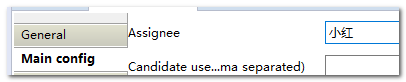


**提示：任务中除了用户任务，有参与者（人），其他都没有参与者。**

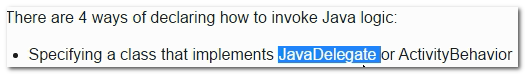
### 编写流程图



为用户任务指定办理人：



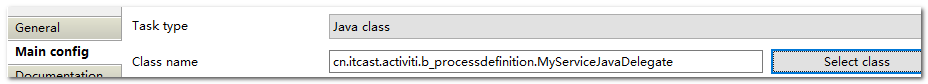
为服务任务来编写服务类：



需要实现JavaDelegate接口：

|  |
| --- |
| //服务任务的代理类：用来自动执行一段代码  **public** **class** MyServiceJavaDelegate **implements** JavaDelegate{  @Override  //命令模式  **public** **void** execute(DelegateExecution execution) **throws** Exception {  //do something  //当节点走到这里的的时候，会自动调用这个方法，执行里面的逻辑，无需人工参与  System.***out***.println("....服务节点（自动节点）正在执行一段xxxx的逻辑啊。。。");  }  } |

为服务任务指定服务的类：



### 测试流程

部署流程：（略）

启动流程：（略）

完成个人任务：

|  |
| --- |
| //个人任务办理  @Test  **public** **void** completePersonTask(){  taskService.complete("2504");  } |

告知完成（等待）任务：

|  |
| --- |
| //------任务  //调用执行等待任务（一般是嵌入到程序的一段代码中）:告知任务  @Test  **public** **void** signalTask(){  //申明告知：告诉工作流，我程序已经处理完了，你继续走流程  runtimeService.signal("2501");  } |

提示：等待任务没有具体的参与人。

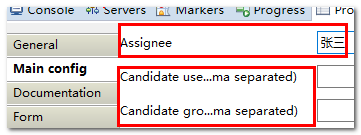
流程的某个任务，是在程序里面运行的一部分。

比如，我们需要写一个业务保存用户业务，但保存该用户之前，需要走一段流程，在流程的某个节点，我才有资格保存该用户，而且保存完了之后，还有后面流程。在保存用户的逻辑的地方调用一个方法就行。

换句话说，该任务，就等着程序去调用他（某些业务操作后），之后，才能往下走。

## 用户任务—UserTask

根据任务办理人的特点，分为两大类：个人任务和群组任务



用户的任务一般都是谁启动的，谁结束。

如申请贷款等业务。

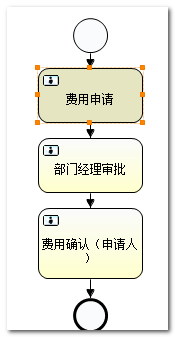
用户任务必须由人（参与者）来完成！！！

### 个人任务的分配

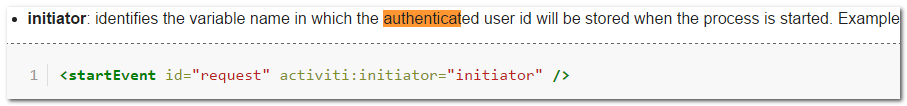
个人任务可以通过三种方式指定办理人

* 为assignee属性指定固定办理人（不常用）
* 为assignee属性指定“变量”（动态）办理人，即通过变量，在程序运行过程中，通过设置变量来动态指定办理人。
* 通过task listeners来动态指定办理人。操作：我们需要实现一个接口，写个类，类里面有个方法，方法中设置该节点的办理人。在画节点的时候，无需指定办理人。办理人会在进入到该节点之前，被lisener监听掉，调用上面的方法，设置进去办理人。

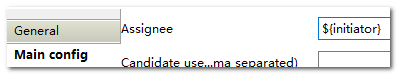
1．编写流程图



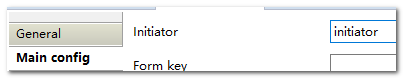
为任务节点设置办理人为发起人：



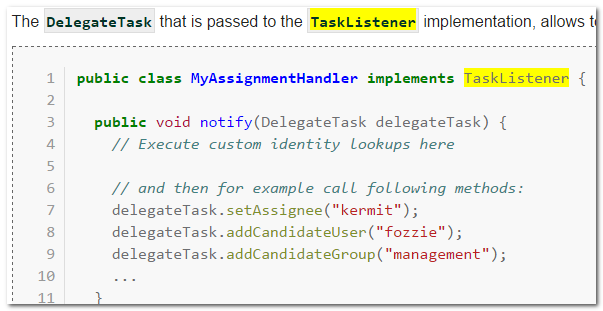
在任务节点指定办理人，使用变量的方式，变了名字叫：initiator（发起人）



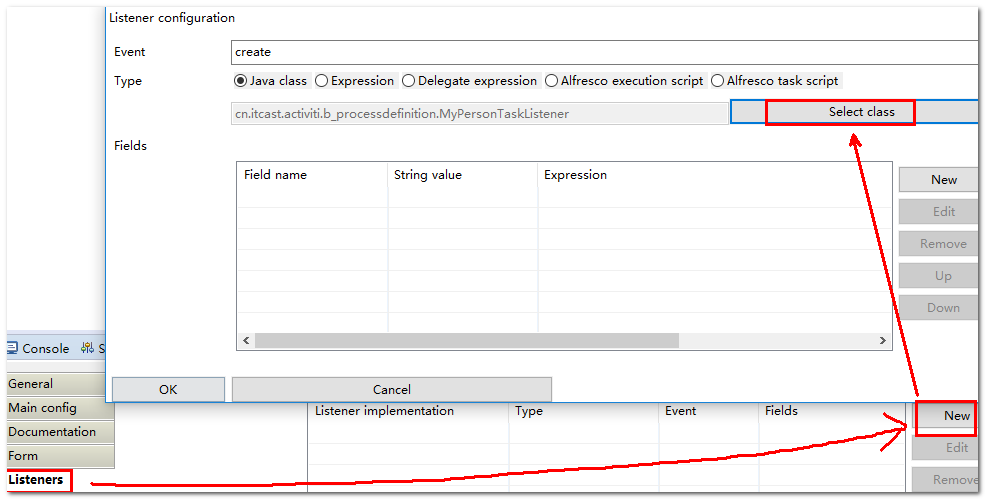
在开始节点上设置发起人的变量名字：



为中间节点，使用task listeners来指定class：



|  |
| --- |
| //用户任务的监听（当走到该节点的时候会自动调用该监听）  **public** **class** MyUserTaskListener **implements** TaskListener{  @Override  //将原来的任务包装  **public** **void** notify(DelegateTask delegateTask) {  //设置办理人:一般实际开发的后，需要从组织机构 树中获取领导来审批  delegateTask.setAssignee("李四");  }  } |



2．测试流程

部署流程：（略）

启动流程：（略）

添加spring注入：

|  |
| --- |
| <!-- 用户Service -->  <bean id=*"identityService"* factory-bean=*"processEngine"* factory-method=*"getIdentityService"*/> |



Java代码：

|  |
| --- |
| //注入用户service  **private** IdentityService identityService;  //用户任务--个人任务  //启动流程  @Test  **public** **void** startProceeInstanceForPerson(){  //使用任务用户id来进行给变量赋值  //调用该方法之前，需要在开始节点告诉工作流，你的认证用户所使用的变量是谁？  identityService.setAuthenticatedUserId("张三");//当前登录人    runtimeService.startProcessInstanceByKey("myProcess");  //在启动的时候加一个变量initiator发起人--map  // runtimeService.startProcessInstanceByKey(processDefinitionKey, variables)  } |

完成个人任务：（略）

发起人在一个流程中一般参与的较多。

### 群组任务（候选人candidate）的分配

业务场景：比如行政大厅提交一个申请，找谁审批？不知道。但你知道找哪个部门，哪个人不知道。

当任务提交给某个部门（包含一堆候选人），到底谁来完成这个任务呢？

谁认领（马上变成个人任务），谁完成。

群组任务可以通过两大类（各三种方式）指定：

第一类：候选人列表：

* 为Candidate Users属性指定固定的群组人员。



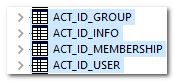
* 为Candidate Users属性指定 “变量”（动态）群组人员，即通过变量，在程序运行过程中，通过设置变量来动态指定群组。



* 通过task listeners在程序中动态的来指定群组人员。



第二类：候选人群组编号（人员和组、关系，都是工作流表中给你设计好了。）



* 为Candidate groups属性指定固定的群组id，该群组中包含群组人员。



* 为Candidate groups属性指定“变量”（动态）的群组id。



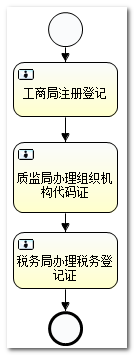
* 通过task listeners在程序中动态的来指定群组id。



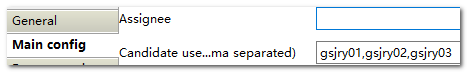
提示：第二类方式以前都不用。原因是：实际业务开发的时候，组织机构、人员信息，都是由业务系统维护的。

1. 编写流程图

业务：注册一个公司流程。



给“工商局注册登记”节点设置固定的群组人员：



给“质监局办理组织机构代码证”节点使用任务监听动态添加群组人员：

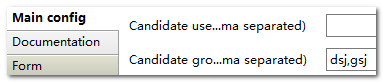
编写监听类：

|  |
| --- |
| //任务监听：用来指定办理人等  **public** **class** MyCandidateTaskListener **implements** TaskListener {  @Override  //将原来的任务包装  **public** **void** notify(DelegateTask delegateTask) {  //设置候选人列表  delegateTask.addCandidateUser("zjjry01");  delegateTask.addCandidateUser("zjjry02");  delegateTask.addCandidateUser("zjjry03");  // delegateTask.addCandidateGroups(“一堆候选人”);  //设置候选人群组的编号  // delegateTask.addCandidateGroup("zjj");  // delegateTask.addCandidateGroups("一堆单位");  } } |

配置监听：

|  |
| --- |
|  |

为Candidate groups属性指定固定的群组id

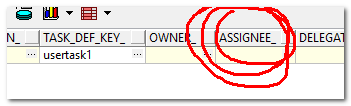


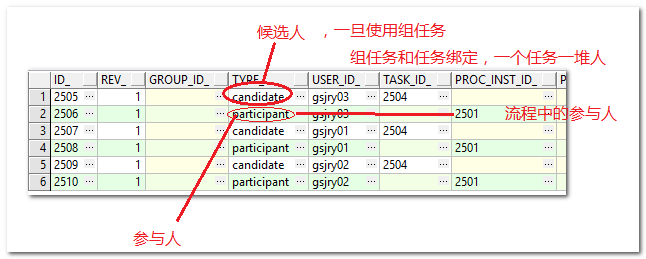
2．测试流程

部署流程：（略）

启动流程：（略）

查询个人任务：无





查询候选人群组任务：

|  |
| --- |
| @Test  //查询候选人任务--查询可以领取的任务（有没有组任务）  **public** **void** queryCandidateTask(){  List<Task> list = taskService.createTaskQuery()  // .taskAssignee(assignee)//个人 任务  //规则：查询任务表的办理人assignee为null,  //并且，当前的走的这个任务节点，人员表中identitylink,人员类型是candidate  //表关联查询，连接条件：任务id  //查询条件： act\_ru\_task表的办理人为null，并且act\_ru\_identitylink表中candidate是指定候选人  .taskCandidateUser("gsjry02")  .list();  **for** (Task task : list) {  System.***out***.println("任务编号："+task.getId());  System.***out***.println("任务名字："+task.getName());  System.***out***.println("任务归属的流程实例id："+task.getProcessInstanceId());  System.***out***.println("任务归属的流程定义的id："+task.getProcessDefinitionId());  }  } |

候选人认领任务：

|  |
| --- |
| //领取(认领)任务  @Test  **public** **void** lingquTask(){  //任务id是列表就有的，用户id：是登录用户  taskService.claim("7502", "gsjry01");  //当领取完成，该任务，变成了个人任务。  } |

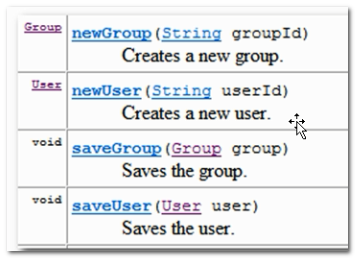
“候选人”查询个人任务

|  |
| --- |
| 略-和个人任务一样 |

“候选人”办理个人任务：

|  |
| --- |
| 略--和个人任务一样 |

创建候选人群组





|  |
| --- |
| //创建用户和群组,以及关联用户和群组  @Test  **public** **void** createUserAndGroup(){  //建立用户  User dsjry01User = identityService.newUser("dsjry01");  dsjry01User.setFirstName("小王");  dsjry01User.setPassword("123456");  identityService.saveUser(dsjry01User);  User dsjry02User = identityService.newUser("dsjry02");  dsjry02User.setFirstName("小李");  dsjry02User.setPassword("123456");  identityService.saveUser(dsjry02User);  User gsjry01User = identityService.newUser("gsjry01");  gsjry01User.setFirstName("老代");  gsjry01User.setPassword("123456");  identityService.saveUser(gsjry01User);  User gsjry02User = identityService.newUser("gsjry02");  gsjry02User.setFirstName("老张");  gsjry02User.setPassword("123456");  identityService.saveUser(gsjry02User);    //建立用户群组  Group dsjGroup = identityService.newGroup("dsj");  dsjGroup.setName("地税局");  identityService.saveGroup(dsjGroup);  Group gsjGroup = identityService.newGroup("gsj");  gsjGroup.setName("国税局");  identityService.saveGroup(gsjGroup);    //用户关联群组  identityService.createMembership("dsjry01", "dsj");  identityService.createMembership("dsjry02", "dsj");  identityService.createMembership("gsjry01", "gsj");  identityService.createMembership("gsjry02", "gsj");  } |

后续流程操作步骤：

查询候选任务

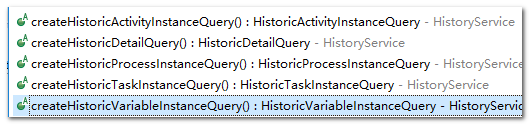
候选人认领任务

候选人查询个人任务

办理个人任务

# 其他api的使用

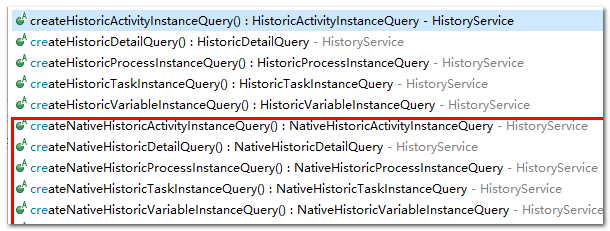
## 历史记录的查询—HistoryService



配置历史的service

|  |
| --- |
| <!-- 历史Service -->  <bean id=*"historyService"* factory-bean=*"processEngine"* factory-method=*"getHistoryService"*/> |

## 本地SQL查询—NativeSQL



官方参考：





配置管理Services：

|  |
| --- |
| <!-- 管理Service -->  <bean id=*"managementService"* factory-bean=*"processEngine"* factory-method=*"getManagementService"*/> |

代码：

示例1：

|  |
| --- |
| @Test  **public** **void** queryByNativeSQL(){  //当默认的api不能满足你的时候，你可以自定义sql语句进行查询  List<HistoricTaskInstance> list = historyService.createNativeHistoricTaskInstanceQuery()  // .sql("select \* from act\_hi\_taskinst")//必须传入sql  .sql("select \* from "+managementService.getTableName(HistoricTaskInstance.**class**))//不知道对象对应的表名，怎么办  .list();  **for** (HistoricTaskInstance historicTaskInstance : list) {  System.***out***.println("历史的任务名称："+historicTaskInstance.getName());  }  } |

示例2

|  |
| --- |
| //NativeSQL查询  @Test  **public** **void** queryByNativeSQL(){  List<Deployment> list = repositoryService.createNativeDeploymentQuery()  .sql("select \* from "+managementService.getTableName(Deployment.**class**))  .list();  **for** (Deployment deployment : list) {  System.***out***.println("发布的编号："+deployment.getId());  System.***out***.println("发布的名称："+deployment.getName());  System.***out***.println("发布的时间："+deployment.getDeploymentTime());  }  } |

## 表单操作—FormService

业务关联工作流系统：

1. 使用流程变量：存放业务表的表名#id，比如t\_leave#1001
2. businiessKey(就是变量)，和流程变量做法一样

执行表（实例）中：，存放t\_leave#1001

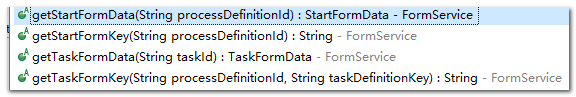
3．formKey

主要用于实现动态表单。(每一个节点的表单页面都不一样)

配置：

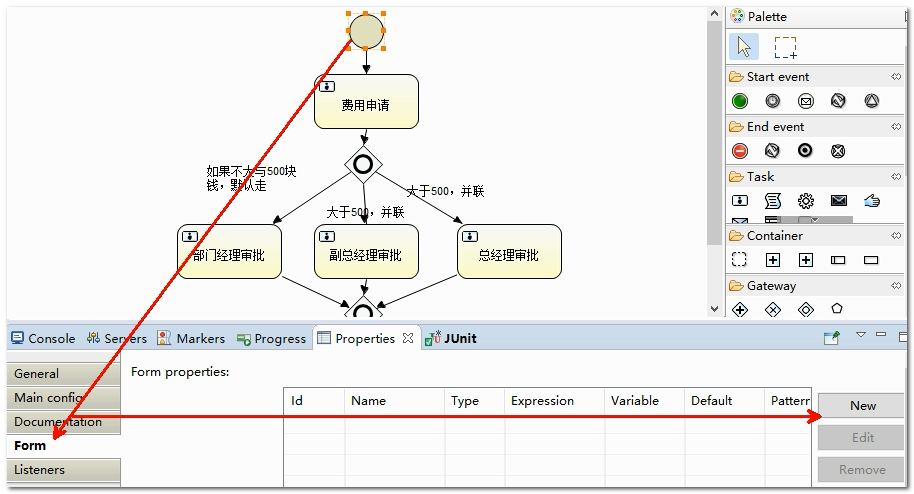
|  |
| --- |
| <!-- 表单Service -->  <bean id=*"formService"* factory-bean=*"processEngine"* factory-method=*"getFormService"*/> |

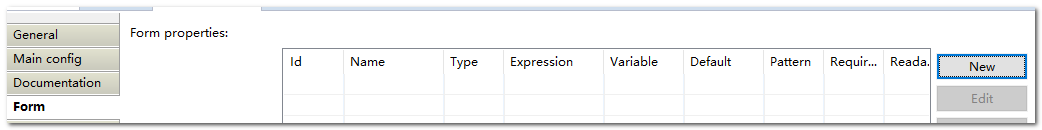
FormService对应的常用api

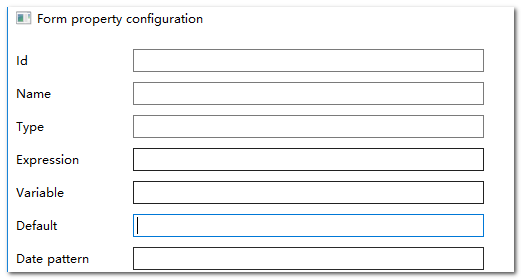


方法1：

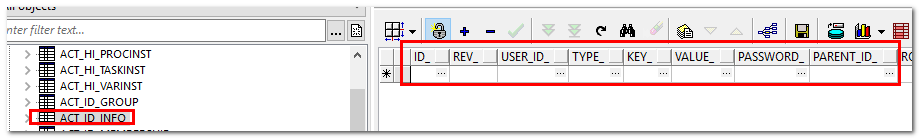
表单由工作流创建和管理（开始节点或任务节点）：







对应的数据表



程序中可以获取表单

|  |
| --- |
| formService.getStartFormData(processDefinitionId);//申请表单的数据  formService.getTaskFormData(taskId);//任务表单数据 |

方法2：

表单由应用程序创建和管理，工作流通过表单的url（form key）来引用



在应用程序中，可以获取该key，从而知道表单的url

|  |
| --- |
| formService.getStartFormKey(processDefinitionId);  formService.getTaskFormKey(processDefinitionId, taskDefinitionKey);//将得到url，放到响应中 |

# 请假流程

重点：

1. 流程变量
2. 连线
3. 网关
4. 任务
5. 历史记录的查询