# 第一部分 Activiti基础

## Activiti表结构

5.18版本有23张表，支持多种关系数据库。

**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。**

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。 流程结束时会把runtime中的数据移入到history中，包括流程实例，任务，变量等。这样运行时的表可以一直很小保持在一个范围内，保证速度很快。

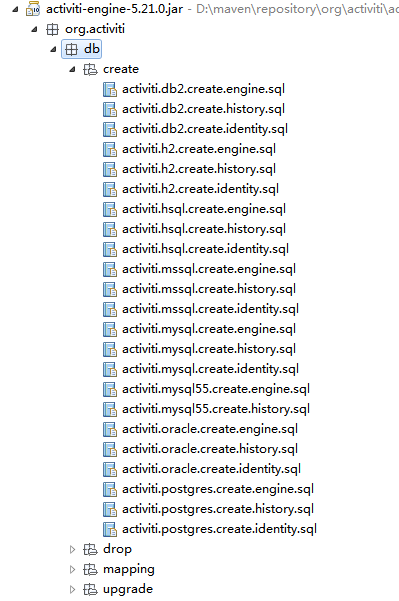
ACT\_ID\_\*: 'ID'表示identity。 这些表包含身份信息，比如用户，组等等。

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， 如存放资源文件。

|  |  |  |
| --- | --- | --- |
| **序号** | **表名** | **说明** |
| 1 | act\_ge\_bytearray | 二进制数据表，存储流程图xml,json和图片信息 |
| 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， |
| 3 | act\_hi\_actinst | 历史节点表 |
| 4 | act\_hi\_attachment | 历史附件表 |
| 5 | act\_hi\_comment | 历史意见表 |
| 6 | act\_hi\_identitylink | 历史流程参与人员表 |
| 7 | act\_hi\_detail | 历史详情表，提供历史变量的查询 |
| 8 | act\_hi\_procinst | 历史流程实例表 |
| 9 | act\_hi\_taskinst | 历史任务表 |
| 10 | act\_hi\_varinst | 历史变量表 |
| 11 | act\_id\_group | 用户组信息表 |
| 12 | act\_id\_info | 用户扩展信息表 |
| 13 | act\_id\_membership | 用户与用户组对应信息表 |
| 14 | act\_id\_user | 用户信息表 |
| 15. | act\_re\_deployment | 部署信息表 |
| 16. | act\_re\_model | 流程设计模型部署表 |
| 17 | act\_re\_procdef | 流程定义数据表 |
| 18 | act\_ru\_event\_subscr | throwEvent、catchEvent时间监听信息表 |
| 19 | act\_ru\_execution | 运行时流程执行实例表 |
| 20 | 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 |
| 21 | act\_ru\_job | 运行时定时任务数据表 |
| 22 | act\_ru\_task | 运行时任务节点表 |
| 23 | act\_ru\_variable | 运行时流程变量数据表 |

Jar包中的脚本，配置可以自动建表。



## 核心API

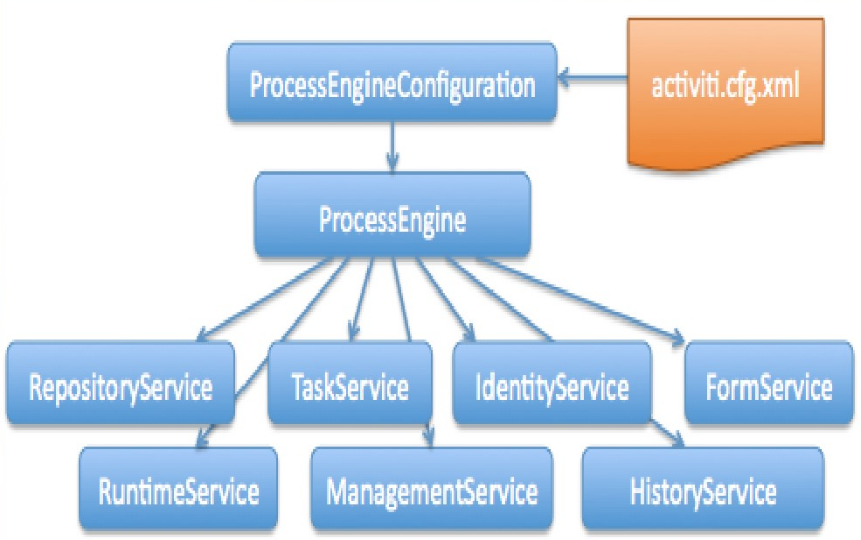
ProcessEngine 是Activiti中最核心的类，其他的类都是由他而来。

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

RepositoryService repositoryService =processEngine.getRepositoryService();

RuntimeService runtimeService = processEngine.getRuntimeService();

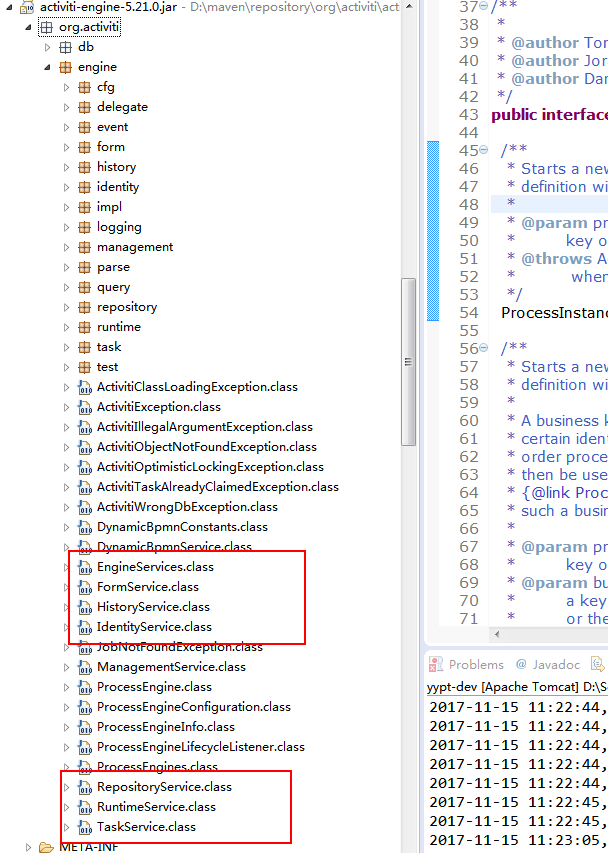
TaskService taskService =processEngine.getTaskService();



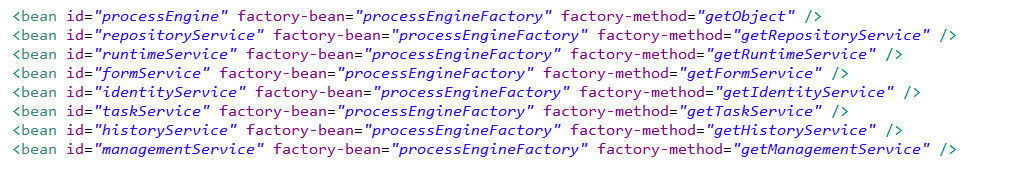
**各个Service的作用：**

|  |  |
| --- | --- |
| **RepositoryService** | 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  该service可以用来删除部署的流程定义。 |
| **RuntimeService** | 执行管理，包括启动，推进，删除流程实例等操作 |
| **TaskService** | 是activiti的任务服务类。可以从这个类中获取任务的相关信息，如当前正在执行的个人待办和用户组待办任务。 |
| **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，可以跟踪流程实例对应所有待办节点的运行情况。 |
| **IdentityService** | 认证服务，在工作流执行过程中进行用户查询、认证等操作 |
| **FormService** | Activiti表单引擎产生的用户任务表单服务 |

**包中的位置：**



**配置信息：**



## 网关分类



**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。如果没有任何一个出口符合条件，**则抛出异常**。

${stuType==1}

${stuType==2}



**无条件触发，不会解析条件，写了条件也会被忽略。**进入和外出的数目不一定相等。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。

流入下一个节点可以设置条件，可以是全部子任务都完成还是50%完成。



集中了前两个网关的特点，**可以定义条件，多条件执行，只要条件返回true就会执行**，包含网关只会等待被选中执行了的进入顺序流。



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

## 任务分类

**用户任务**：指定一个人或者一个变量，设置的人

直接指定办理人：${assignee}

任务监听器设置处理人：

TaskListenerImpl **implements** TaskListener

delegateTask.setAssignee("XXX");

**脚本任务：**

脚本任务是一个自动化活动。当一个流程执行到达脚本任务时，执行相应的脚本。

https://www.cnblogs.com/dengjiahai/p/6942433.html

**服务任务：**



public class ServiceTask implements JavaDelegate{

@Override

public void execute(DelegateExecution execution) throws Exception

{ 业务逻辑... } }

**消息任务**：当流程到达这个节点处于停滞状态，程序收到一个消息之后signal 流程实例。

runtimeService.signal(execution.getId());

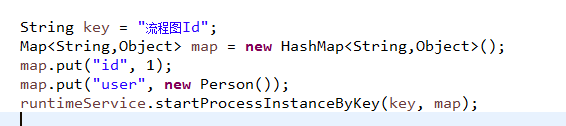
## 流程变量

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。分为两种：流程实例和流程任务的流程变量。

### 1 设置流程变量

1）设置流程变量有多种方式，可以在流程启动，和任务执行的时候设置

a.启动流程的时候设置



b. 通过taskService或者runtimeService设置



**提示：如果设置的流程变量值是一个对象类型，这个对象必须实现序列化Serializable操作,并且要为该对象类指定一个 版本ID**

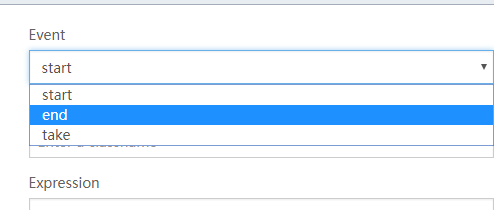
### 2 获取流程变量



## 监听器

1. **全局监听器**

主要使用的场景就是监控这个流程的启动和结束。流程开始的时候可以监控，流程结束的时候可以监控，这里说的是流程实例启动结束的监控



**包括流程的启动和停止，线条的take监听（）**

ExecutionListener定义如下：流程实例start、end、take的时候调用。take是监控连线的时候使用的。

public interface ExecutionListener extends Serializable {

  String EVENTNAME\_START = "start";

  String EVENTNAME\_END = "end";

  String EVENTNAME\_TAKE = "take";

 void notify(DelegateExecution execution) throws Exception;

}

package com.daling.ch1.listener;

import org.activiti.engine.delegate.DelegateExecution;

import org.activiti.engine.delegate.ExecutionListener;

public class MyExecutionListener implements ExecutionListener {

public void notify(DelegateExecution execution) throws Exception {

String eventName = execution.getEventName();

//start

if ("start".equals(eventName)) {

System.out.println("start=========");

}else if ("end".equals(eventName)) {

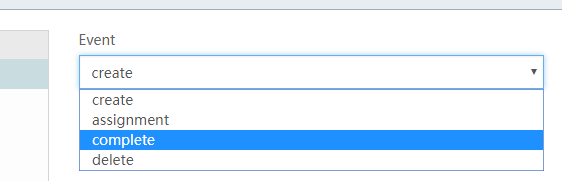
System.out.println("end=========");

}

}

}

**2 任务节点监听器**



节点监听器的定义接口[org](eclipse-javadoc:%E2%98%82=activitidemo/E:\\/WorkTool-E+M+T+J\\/Maven\\/maven_repository\\/org\\/activiti\\/activiti-engine\\/5.19.0.2\\/activiti-engine-5.19.0.2.jar<org" \t "http://blog.csdn.net/qq_30739519/article/details/_blank).[activiti](eclipse-javadoc:%E2%98%82=activitidemo/E:\\/WorkTool-E+M+T+J\\/Maven\\/maven_repository\\/org\\/activiti\\/activiti-engine\\/5.19.0.2\\/activiti-engine-5.19.0.2.jar<org.activiti" \t "http://blog.csdn.net/qq_30739519/article/details/_blank).[engine](eclipse-javadoc:%E2%98%82=activitidemo/E:\\/WorkTool-E+M+T+J\\/Maven\\/maven_repository\\/org\\/activiti\\/activiti-engine\\/5.19.0.2\\/activiti-engine-5.19.0.2.jar<org.activiti.engine" \t "http://blog.csdn.net/qq_30739519/article/details/_blank).[delegate](eclipse-javadoc:%E2%98%82=activitidemo/E:\\/WorkTool-E+M+T+J\\/Maven\\/maven_repository\\/org\\/activiti\\/activiti-engine\\/5.19.0.2\\/activiti-engine-5.19.0.2.jar<org.activiti.engine.delegate" \t "http://blog.csdn.net/qq_30739519/article/details/_blank).TaskListener

**public class MyExecutionListener implements TaskListener {**

**public void notify(DelegateTask delegateTask) {**

**String eventName = delegateTask.getEventName();**

**}**

**}**

1. **AOP式的监听器**
2. 一般思路ACT\_RE\_actdef\_ext表配置每个节点的处理人（变量，角色，用户组，团队），邮件通知人，回调服务等信息。

在配置文件中配置ParseHandler，启动服务器会自动为所有流程的 process(start,end); usertask(start,complete,end); sequence(take) 添加监听器,根据 配置的节点对应的信息做相应的业务逻辑处理。

<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">

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

<property name="transactionManager" ref="transactionManager" />

<property name="databaseSchemaUpdate" value="true" />

<property name="jpaHandleTransaction" value="true" />

<property name="jpaCloseEntityManager" value="true" />

<property name="jobExecutorActivate" value="false" />

<property name="idGenerator" ref="uuidGenerator"/>

<!-- <property name="deploymentResources" value="classpath\*:diagrams/\*.\*" /> -->

<property name="customDefaultBpmnParseHandlers">

<list>

**<bean class="com.parse.ActivitiProcessExtParseHandler" />**

**<bean class="com.parse.ActivitiReceiveTaskExtParseHandler" />**

**<bean class="com.parse.ActivitiSequenceFlowExtParseHandler" />**

**<bean class="com.parse.ActivitiServiceTaskExtParseHandler" />**

**<bean class="com.parse.ActivitiUserTaskExtParseHandler" />**

</list>

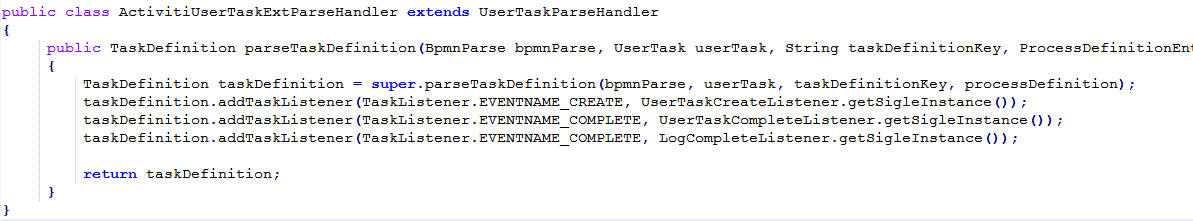
</property>

</bean>

（2）[扩展Activiti流程定义文件,实现自定义节点属性](http://www.baidu.com/link?url=v9QoFYBcaMSxecCu08PdCpCLAFyYxUMEtDeHGD2l38YR9pHQaSOeZzfHqWt7q8Yq8OSNO0SsvRNLJd4Mn8FIk6-X4eZSvFlVG69F82gm7nu" \t "https://www.baidu.com/_blank)

customDefaultBpmnParseHandlers 注册各种监听器





监听器包括启动流程日志，任务完成日志，流程开始，流程结束，消息任务开始，消息任务结束，服务任务开始，服务任务结束，用户任务开始，用户任务完成等监听器。



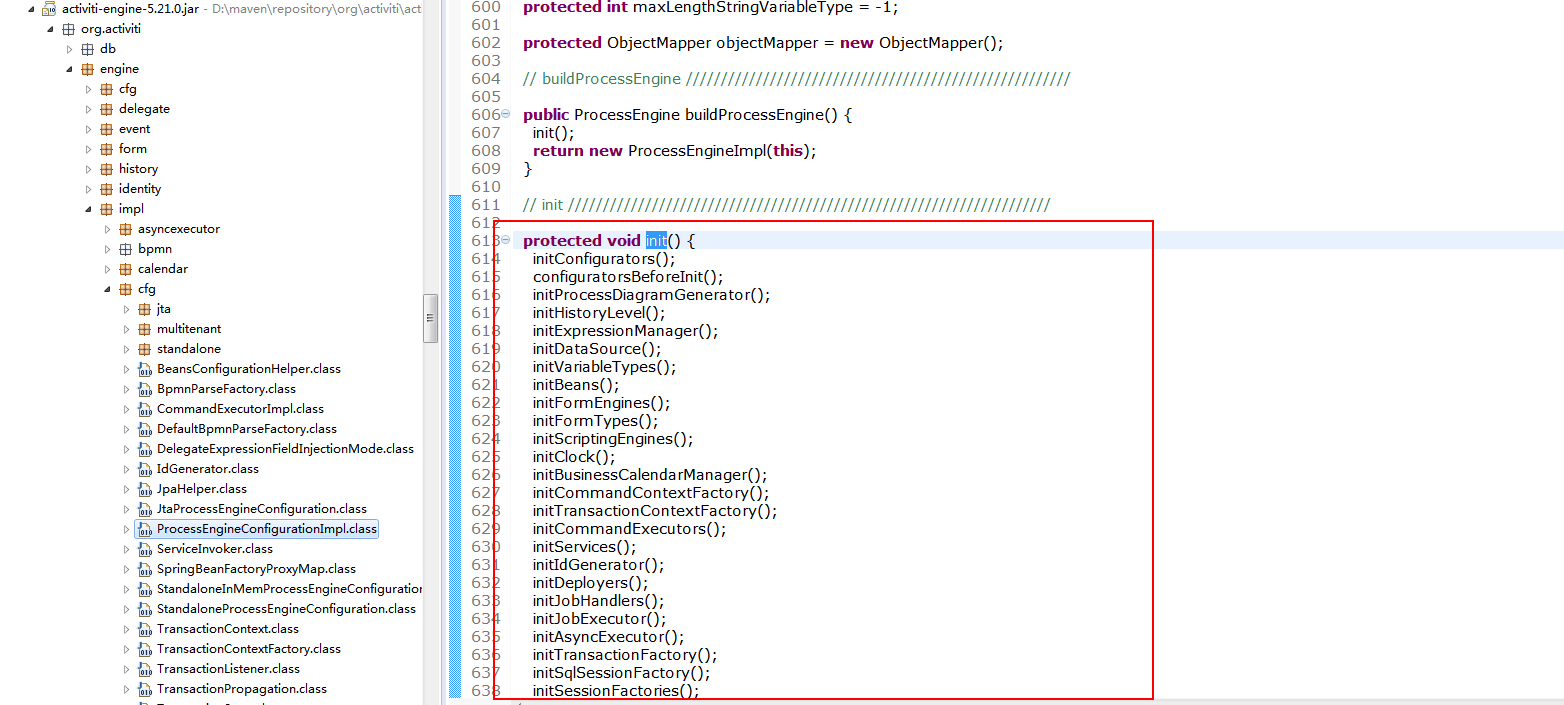
## 命令模式和责任链模式

Activiti任务的执行 用到了设计模式中的命令模式和责任链模式。

命令模式是将行为请求者和行为实现者解耦合的方式。对命令进行封装，将命令和执行命令分隔开。请求的一方发出命令，要求执行某些操作，接受一方收到命令，执行这些操作的真正实现。请求的一方不必知道接受方的接口，以及如何被操作。

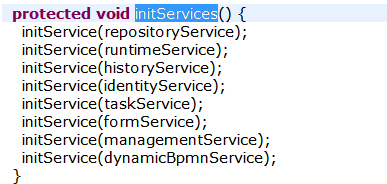
**以启动流程为例：**

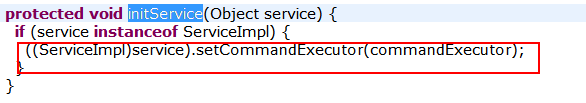
ProcessEngineConfigurationImpl 会初始化一系列配置和注入信息，包括 initCommandExecutors();



RuntimeServiceImpl 继承ServiceImpl ，

在初始化的时候已经注入commandExecutor命令执行器。





**public** **class** RuntimeServiceImpl **extends** ServiceImpl **implements** RuntimeService

**public** ProcessInstance startProcessInstanceByKey(String processDefinitionKey, String businessKey) {

**return** commandExecutor.execute(**new** StartProcessInstanceCmd<ProcessInstance>(processDefinitionKey, **null**, businessKey, **null**));

}

protected void initService(Object service)

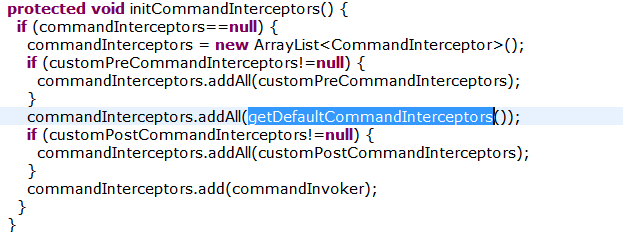
{

if (service instanceof ServiceImpl) ((ServiceImpl) service).setCommandExecutor(this.commandExecutor);

}

启动流程类StartProcessInstanceCmd实现Command接口，重写execute方法。注入CommandContext对象，commandContext可以得到流程实例，流程定义，历史数据，任务数据的API，做相应的流程逻辑处理。

以启动流程拦截器链为，一般是默认的拦截器，可以自定义before/after拦截器。   
logger拦截器-->spring事务拦截器-->CommandContext拦截器-->CommandInvoker拦截器



其中CommandContext拦截器的工作主要是设置Context：

[java] [view plain](http://blog.csdn.net/bluejoe2000/article/details/41800889" \o "view plain) [copy](http://blog.csdn.net/bluejoe2000/article/details/41800889" \o "copy)

Context.setCommandContext(context);

Context.setProcessEngineConfiguration(processEngineConfiguration);

return next.execute(config, command);

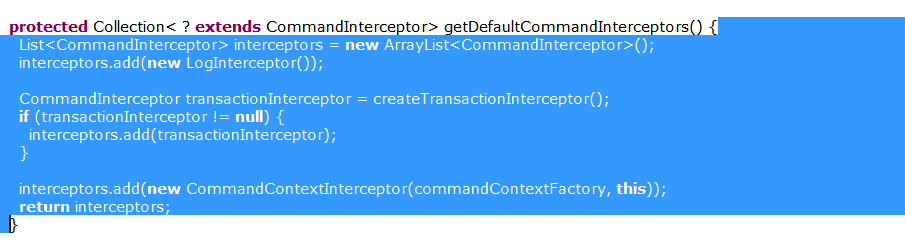
Activiti默认的拦截器：

    1. LogInterceptor日志拦截器，拦截器打印执行的日志。

    2.事务拦截器。

    3.CommandContextInterceptor 命令上下文拦截器设置Context.

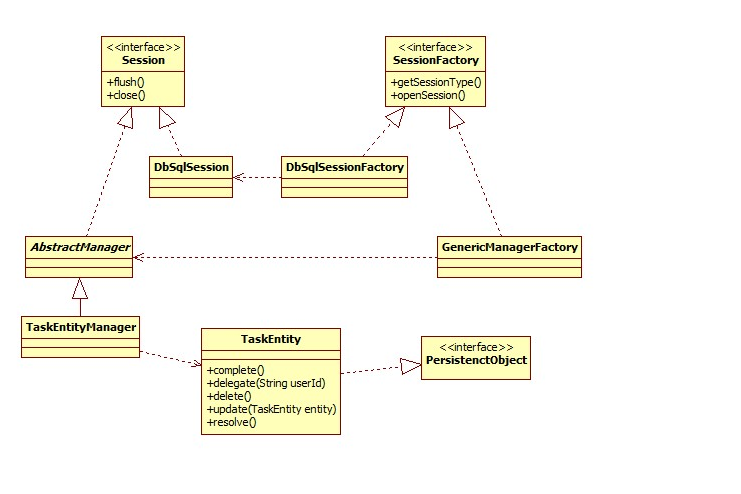
     4 CommandInvoker拦截器,执行命令



**操作数据库：**



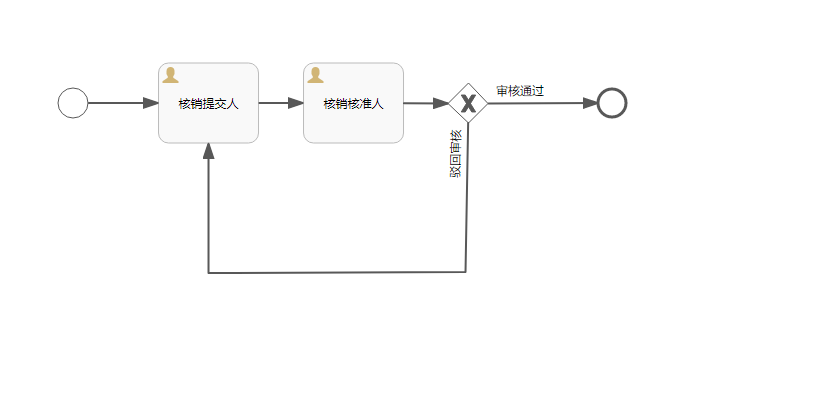
Activiti使用ibatis作为ORMapping工具。在此基础之上Activiti设计了自己的持久化框架



## 单元测试

自己可以绘制各种流程，使用某些任务节点和网关等构件，写单元测试跑流程，查看数据库中各个表数据的变化，加深对流程流转的理解。

**下面以简单的核销流程为例：**



**测试基类 BaseTest**

@RunWith(SpringJUnit4ClassRunner.**class**)

@WebAppConfiguration

@ContextConfiguration({ "classpath:spring-context.xml",

"classpath:spring-context-activiti.xml" })

**public** **class** BaseTest

{

@Before

**public** **void** before()

{}

@After

**public** **void** after()

{}

}

**流程测试基类 BaseActivitiTest**

**public** **class** BaseActivitiTest **extends** BaseTest

{

@Autowired

**protected** ProcessEngine processEngine;

@Autowired

**protected** RepositoryService repositoryService;

@Autowired

**protected** RuntimeService runtimeService;

@Autowired

**protected** TaskService taskService;

@Autowired

**protected** HistoryService historyService;

@Autowired

**protected** FormService formService;

}

**核销流程测试类 VerificationWorkflowServiceTest**

**public** **class** VerificationWorkflowServiceTest **extends** BaseActivitiTest

{

//业务key

String businessKey = "1003";

//流程定义key

String processDefineKey = "verification\_flow";

//处理人AR

String arAssignee = "wade";

//处理人CRO

String croAssignee = "james";

@Test

**public** **void** testVerificationProcess( )

{

startProcessInstance();

//completeTaskAr();

//backTaskCro();

//completeTaskAr();

//completeTaskCro();

}

//启动流程

**public** **void** startProcessInstance( )

{

Map<String, Object> variables = Maps.*newHashMap*();

variables.put("assignee", arAssignee);

runtimeService.startProcessInstanceByKey(processDefineKey, businessKey,variables);

}

//AR完成任务

**public** **void** completeTaskAr( )

{

Task task = taskService.createTaskQuery().active().taskAssignee(arAssignee).list().get(0);

Map<String, Object> variables = Maps.*newHashMap*();

variables.put("assignee", croAssignee);

taskService.complete(task.getId(), variables);

}

//CRO回退任务

**public** **void** backTaskCro( )

{

Task task = taskService.createTaskQuery().active().taskAssignee(croAssignee).list().get(0);

Map<String, Object> variables = Maps.*newHashMap*();

variables.put("assignee", arAssignee);

variables.put("appoveResult", "N");

taskService.complete(task.getId(), variables);

}

//CRO完成任务

**public** **void** completeTaskCro( )

{

Task task = taskService.createTaskQuery().active().processInstanceBusinessKey(businessKey).taskAssignee(croAssignee).list().get(0);

Map<String, Object> variables = Maps.*newHashMap*();

variables.put("appoveResult", "Y");

taskService.complete(task.getId(), variables);

}

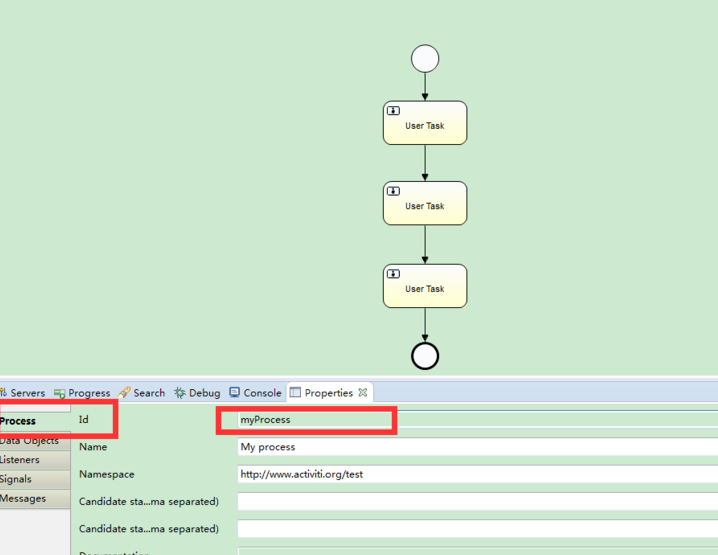
}

# 第二部分 OPS聚引客流程

## 流程绘制和部署

1. **传统的绘制和部署**

Eclipse安装 [Activiti Designer](http://www.baidu.com/link?url=Ke4bltPoQV7_CdENNivjkDU1mMoO10LabJoolp5kwTebkVyNIy8DjvX5uO9YVXajI1jWGHcM3D_zSLI8bO_fra" \t "https://www.baidu.com/_blank)插件，可以打开bpmn类型的文件，引入各种构件编辑流程图，并且通过代码部署流程。



Deployment deployment = processEngine.getRepositoryService()

.createDeployment()

.name("核销流程")

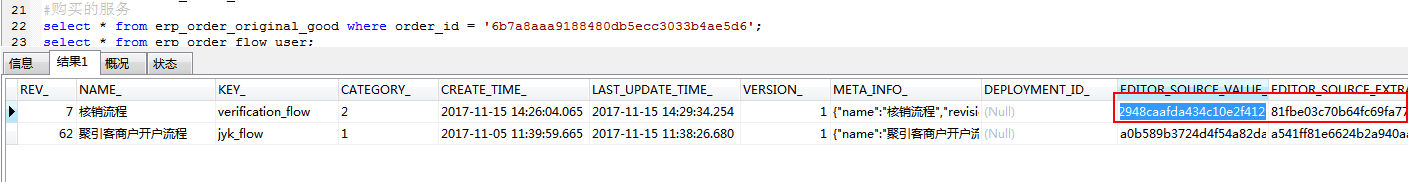
.addClasspathResource("diagrams/verification\_flow.bpmn") ;

1. **[Activiti Model Editor](http://www.baidu.com/link?url=ZX3unY_JKI-3xrP0gaK_ZXmBLEzmneYomOm_NMSs4yDWmxy17FHniYLX7pXZF5rjgxW1c6OcvqtcvtixyCWWOa" \t "https://www.baidu.com/_blank)**

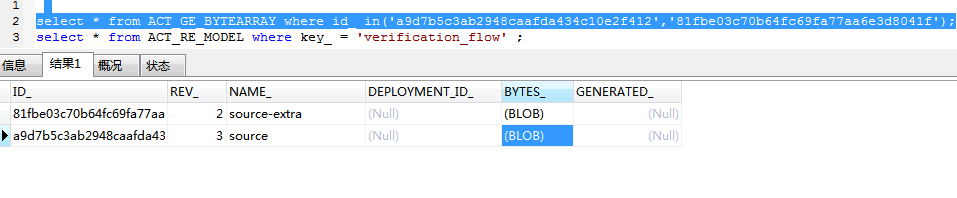
（1）新建模型，编辑流程，流程分类在数据字典编辑，新增一个分类 .

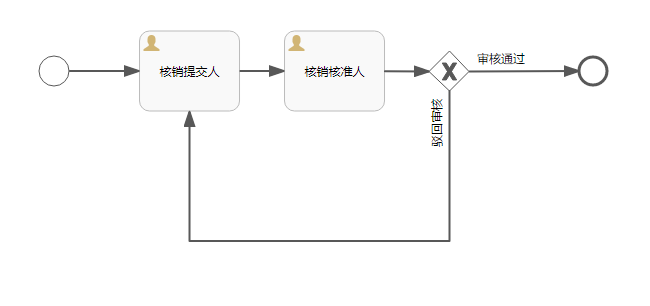


这个时候 ACT\_RE\_MODEL新增加一条记录，EDITOR\_SOURCE\_VALUE\_ID\_和EDITOR\_SOURCE\_EXTRA\_VALUE\_ID\_分别对应ACT\_GE\_BYTEARRAY的id. 存储模型的JSON信息和图片信息。



JSON文件和图片信息，可以另存为





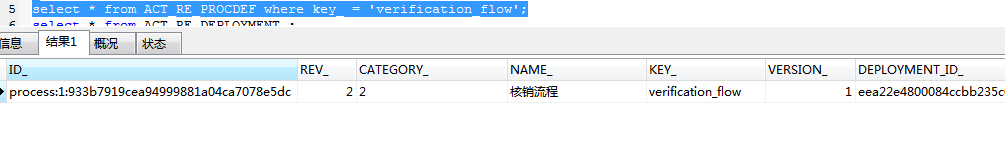
保存在数据库中的JSON 文件。

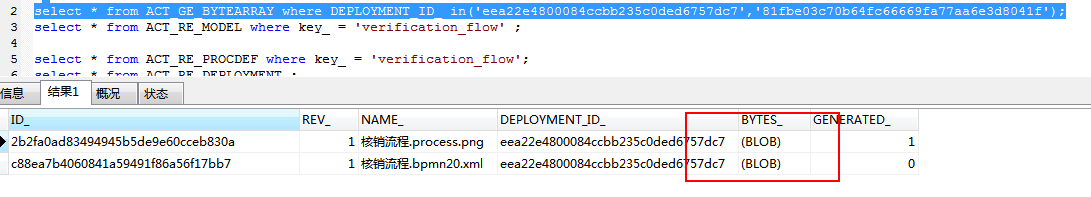


1. 部署流程

有了模型之后，可以部署流程定义，部署成功后ACT\_RE\_PROCDEF出现一条流程定义数据。

DEPLOYMENT\_ID\_和ACT\_GE\_BYTEARRAY的DEPLOYMENT\_ID\_关联可以查询出对应的xml文件和png图片。







1. 重新部署流程产生的问题

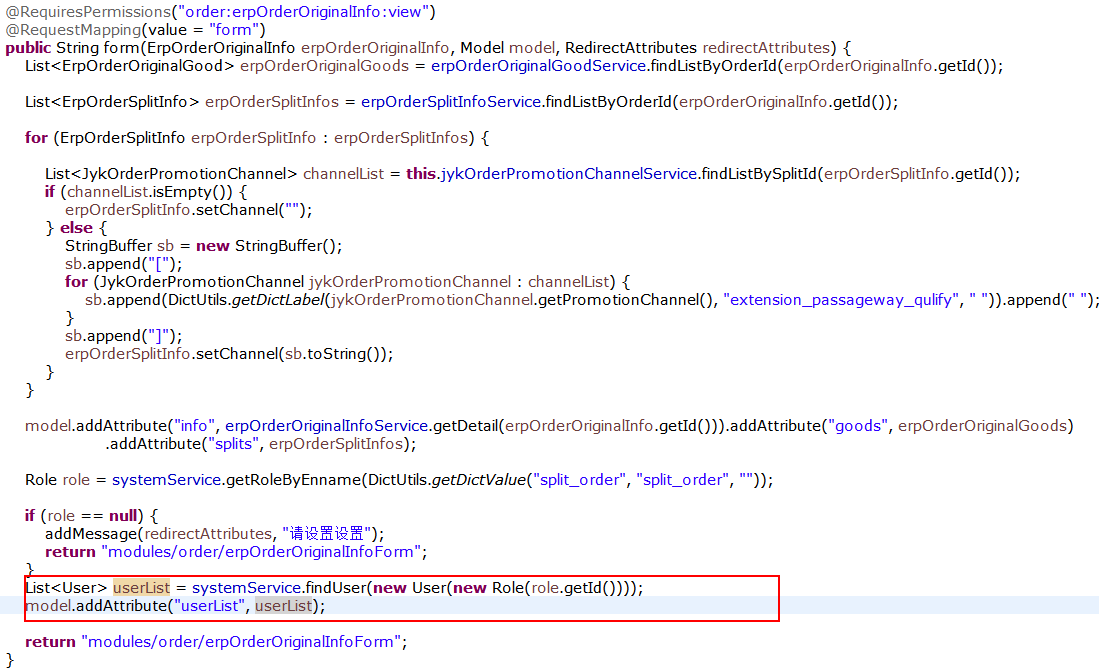
流程定义转化成一个新的模型，

流程定义有版本的概念，新启动的流程实例按照最新的版本去跑，中途的流程按照之前的版本走下去。所以注意在途的单子不受到影响。部署流程之后需要重启服务器，这是因为服务启动之后会把整个流程图加内存中。

## 订单拆分流程

选择处理人：

ErpOrderOriginalInfoController，数据字典中配置好角色名称，找到对应的用户列表，传到前台，迭代显示。

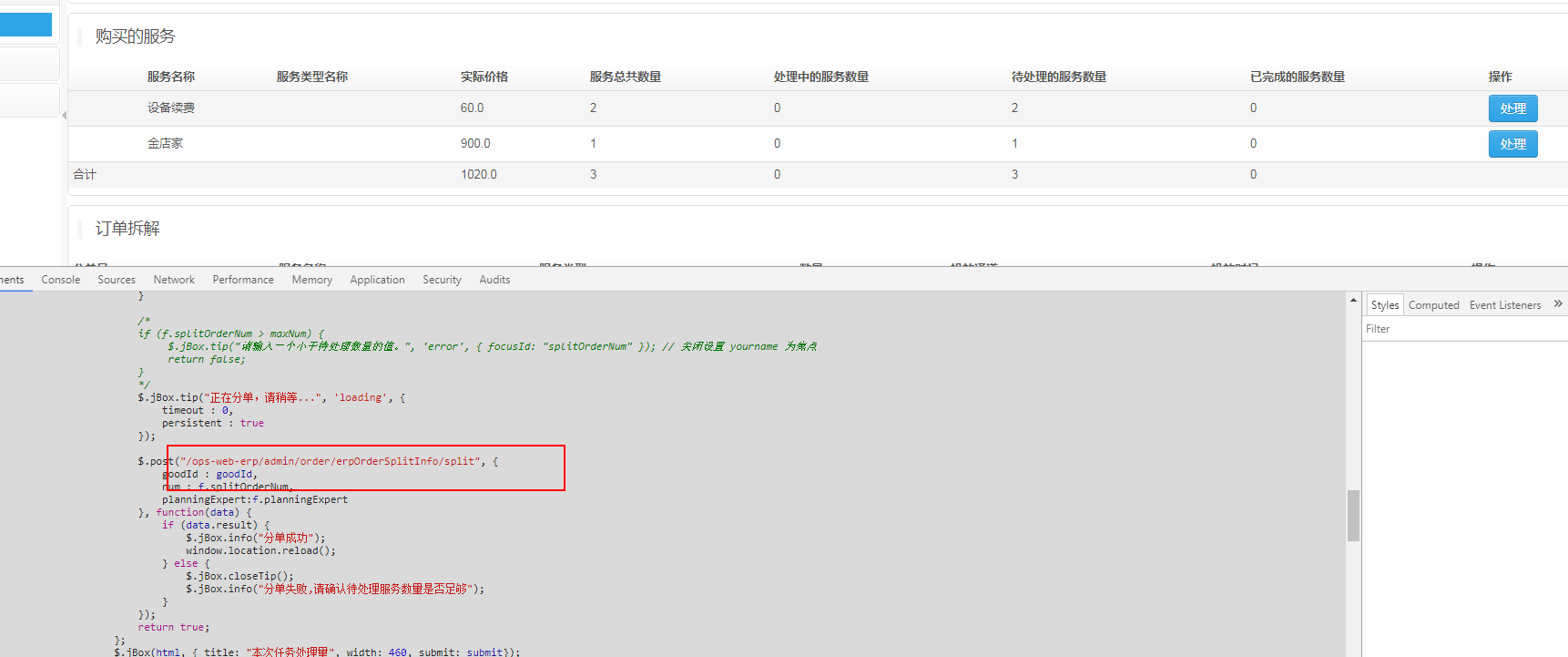


erpOrderOriginalInfoForm.jsp

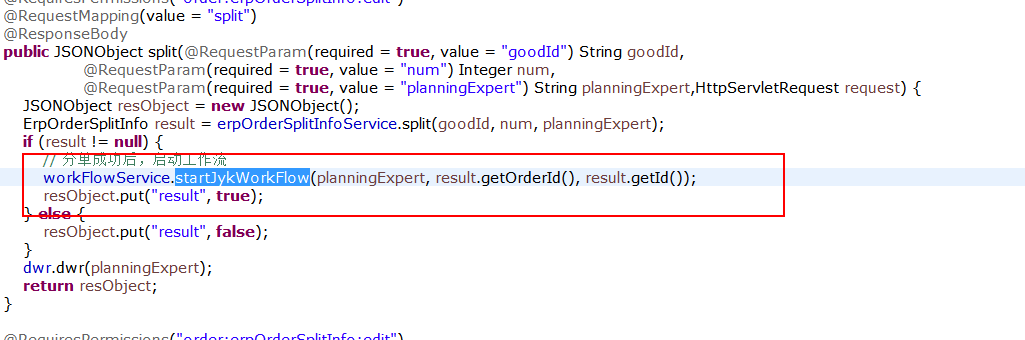
+"<c:forEach items='${userList }' var='user'>"

+" <option value='${user.id}'>${user.name }</option>"

+"</c:forEach>"



ErpOrderSplitInfoController：拆单控制器





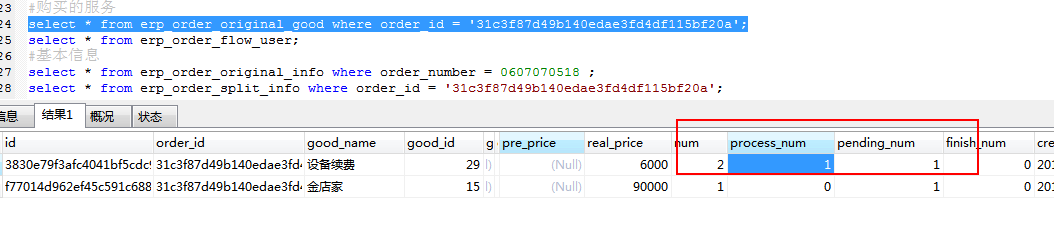
拆单的业务逻辑：

erp\_order\_original\_info根据order\_num (0607070518)

找到order\_id ：31c3f87d49b140edae3fd4df115bf20a。

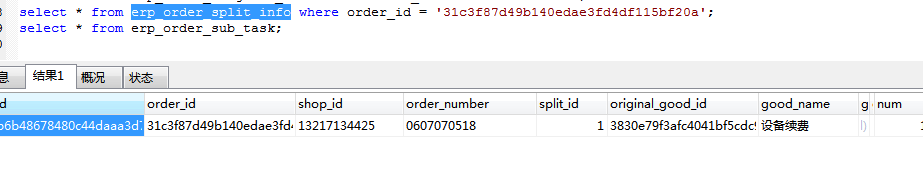


erp\_order\_original\_good订单商品表的处理数量和待处理数量改变

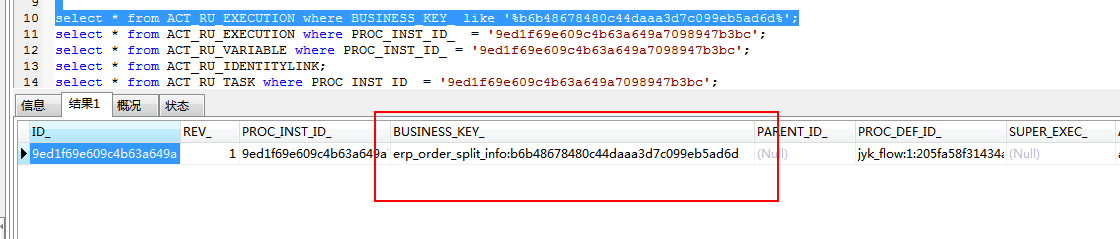


erp\_order\_split\_info表新增一条拆单信息，订单ID对应订单表的order\_id

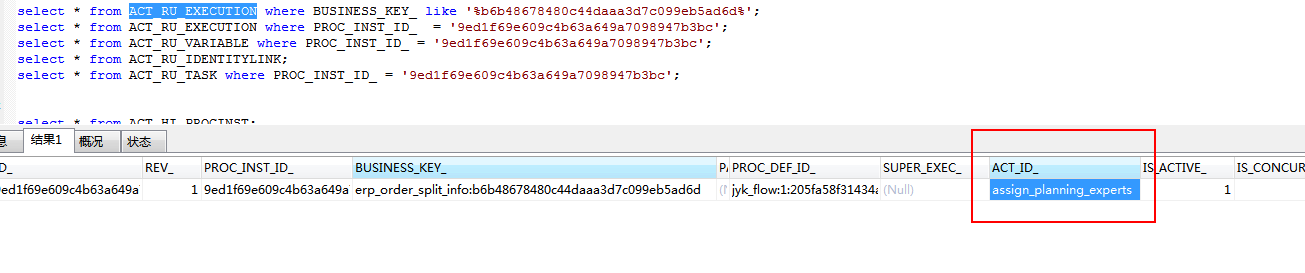
该表的id作为流程实例的businessKey .

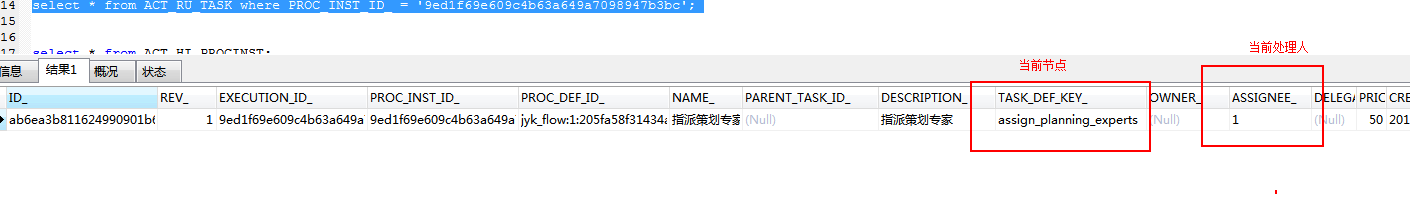


ACT\_RU\_EXECUTION表启动一个流程，businessKey用固定前缀+split\_info.id .其实完全可以直接用split\_info.id。



查看任务和当前节点





。

## 常见定位问题的SQL

#流程定义部署数据

select \* from ACT\_GE\_BYTEARRAY

where DEPLOYMENT\_ID\_ in('eea22e480757dc7','81fbe03c70e3d8041f');

select \* from ACT\_RE\_MODEL where key\_ = 'verification\_flow' ;

select \* from ACT\_RE\_PROCDEF where key\_ = 'verification\_flow';

select \* from ACT\_RE\_DEPLOYMENT ;

select \* from ACT\_GE\_BYTEARRAY

where id\_ in('e49ad30d90ef41ca1fee43496','1384e0f776434683929818743');

#运行的流程数据

select \* from ACT\_RU\_EXECUTION e43f18cdec004ae5a73e9%';

select \* from ACT\_RU\_EXECUTION

where BUSINESS\_KEY\_ like '%049b0b655b6

where PROC\_INST\_ID\_ = 'dcd28562008941da9f4d662f97d0fc0d';

select \* from ACT\_RU\_VARIABLE

where PROC\_INST\_ID\_ = 'dcd28562008941da9f4d662f97d0fc0d';

select \* from ACT\_RU\_IDENTITYLINK;

select \* from ACT\_RU\_TASK where PROC\_INST\_ID\_ = 'dcd28562008941da9f4d662f97d0fc0d';

#流程历史数据

select \* from ACT\_HI\_PROCINST;

select \* from ACT\_HI\_TASKINST;

select \* from ACT\_HI\_VARINST;

#订单信息

select \* from erp\_order\_file ;

select \* from erp\_order\_original\_good

where order\_id = '6b7a8aaa9188480db5ecc3033b4ae5d6';

select \* from erp\_order\_flow\_user;

select \* from erp\_order\_original\_info where order\_number = 2830864824 ;

select \* from erp\_order\_split\_info where order\_id = '6b7a8aaa9188480db5ecc3033b4ae5d6';

select \* from erp\_order\_sub\_task;

########用户角色相关#############

select \* from sys\_role;

select \* from sys\_user;

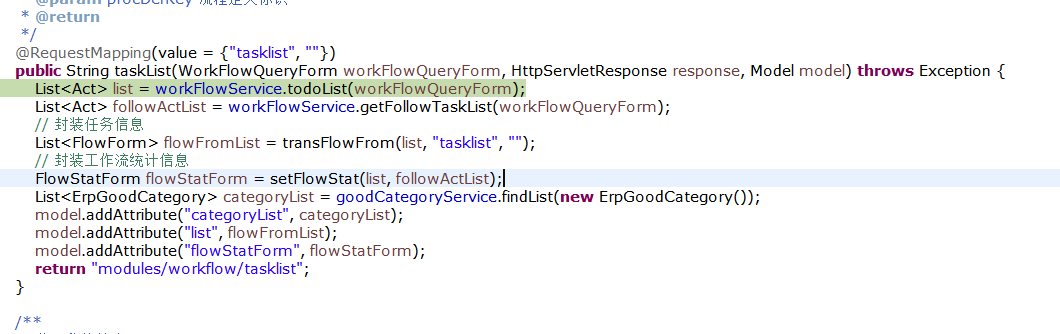
select \* from sys\_user\_role;

select \* from sys\_role\_menu;

select \* from sys\_menu;

## 流程流转

查询我的任务：WorkFlowController



TaskQuery todoTaskQuery = taskService.createTaskQuery().taskAssignee(userId).active().includeProcessVariables().orderByTaskCreateTime()

.desc();

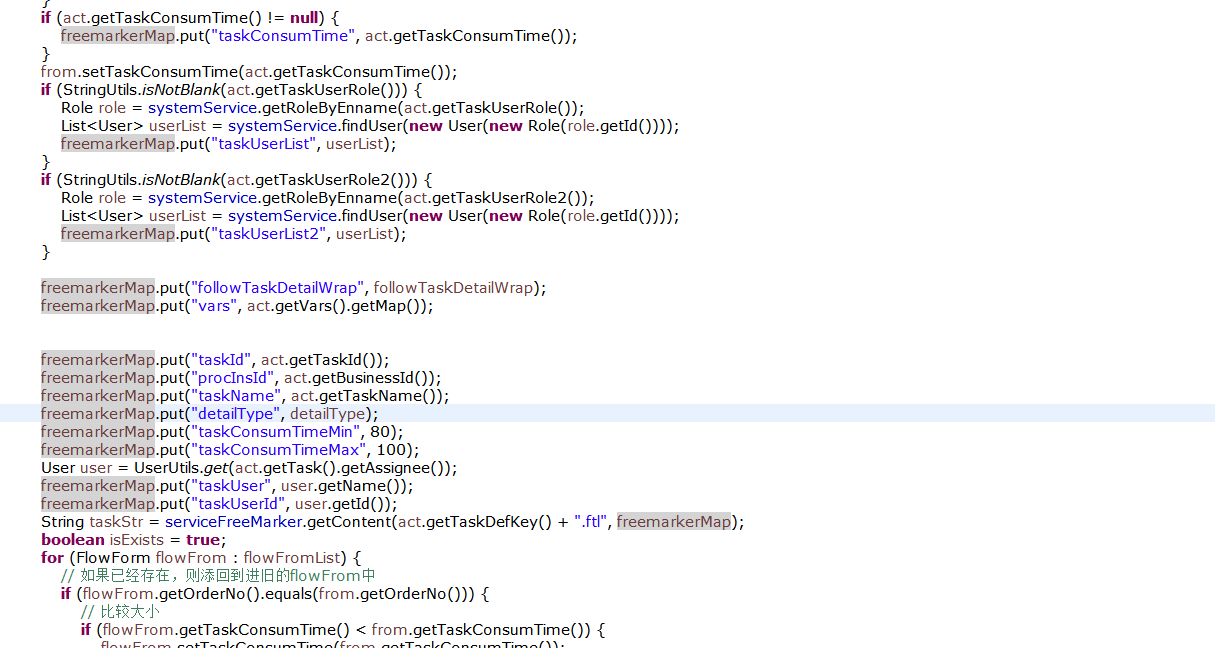
查询出所有代办，根据查询参数过滤结果集。

获取节点表单的数据：



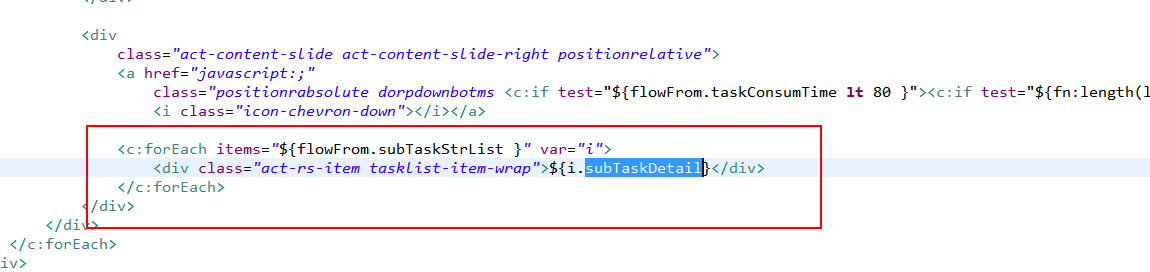


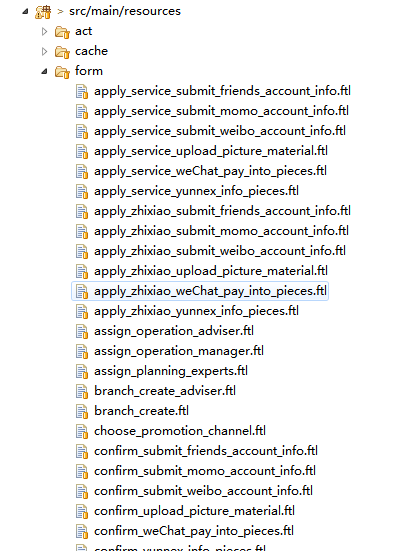
根据对应的模板文件设置参数，放入taskDetail字段中。



显示模板代码到前台tasklist.jsp,循环显示 子任务

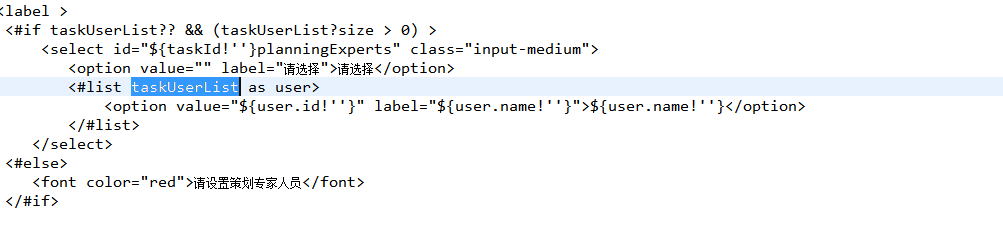
每个节点有对应的模板文件





显示审批的人员，节点封装好taskUserRole的表单属性，根据角色名称获取人员





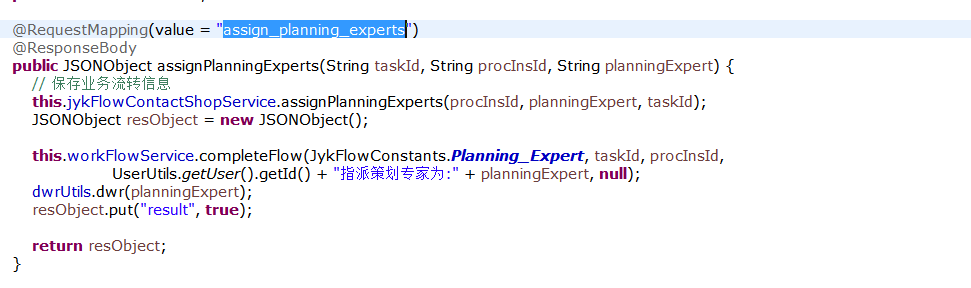
审批逻辑

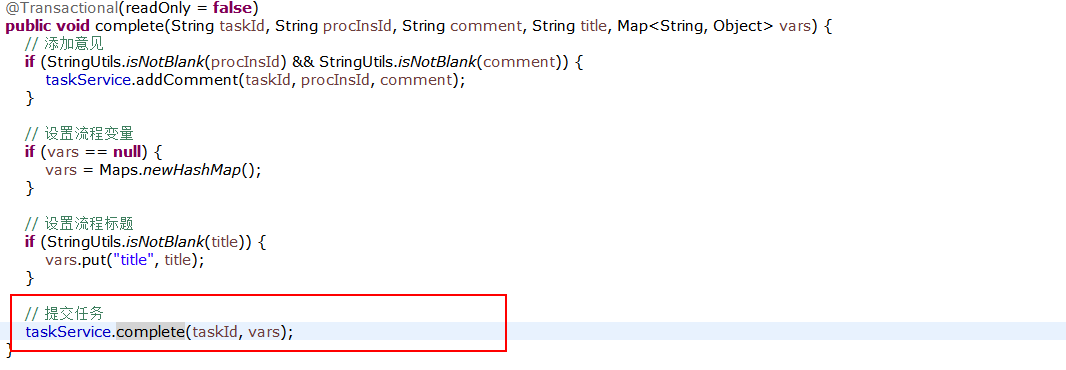
JykFlowContactShopController

JykFlowProductController

JykFlowServiceOrderController

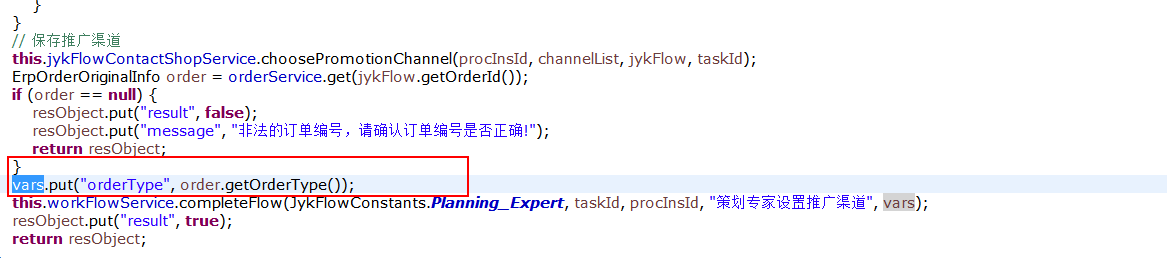
这三个controller封装了所有节点的审批逻辑，一个节点对应一个方法。

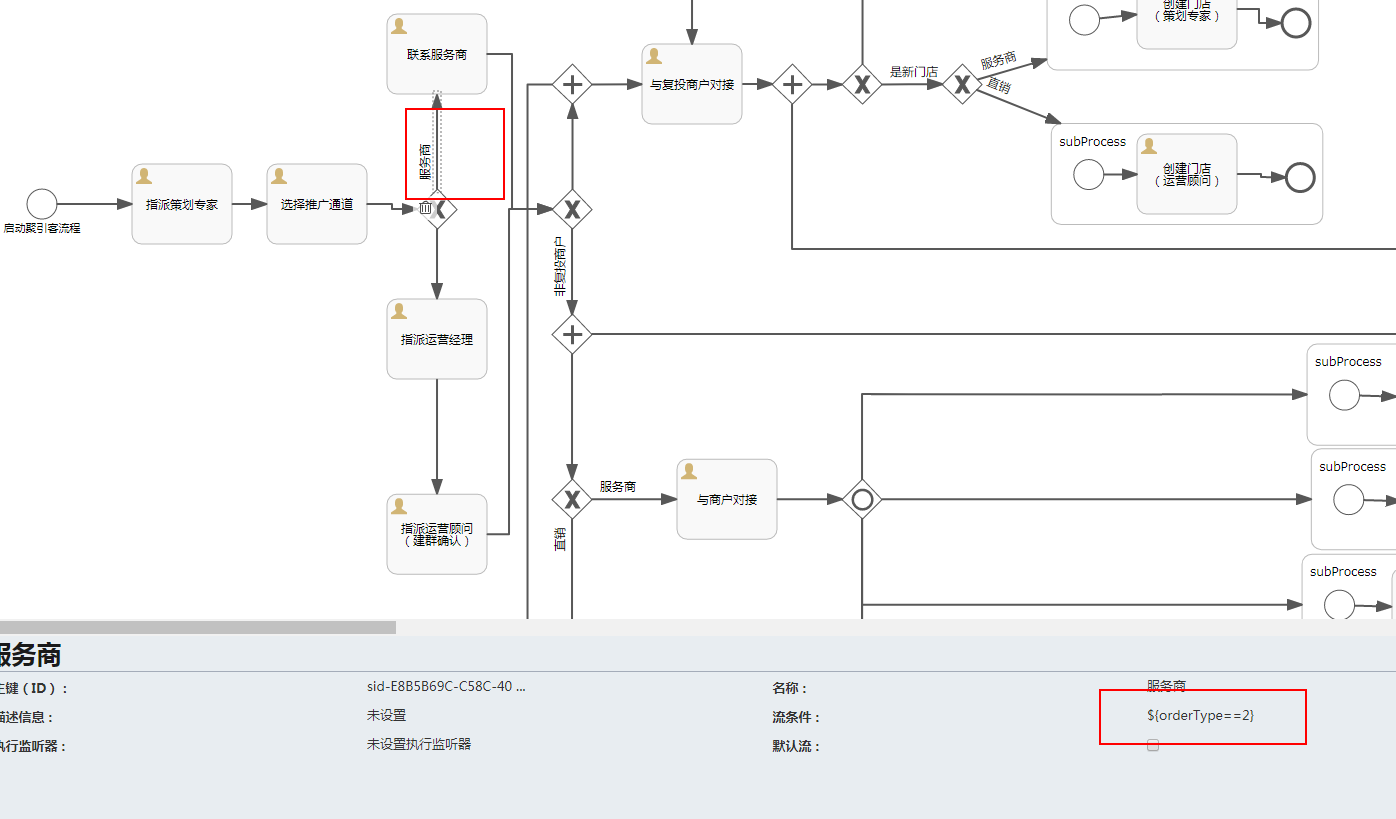




路由参数的设置：

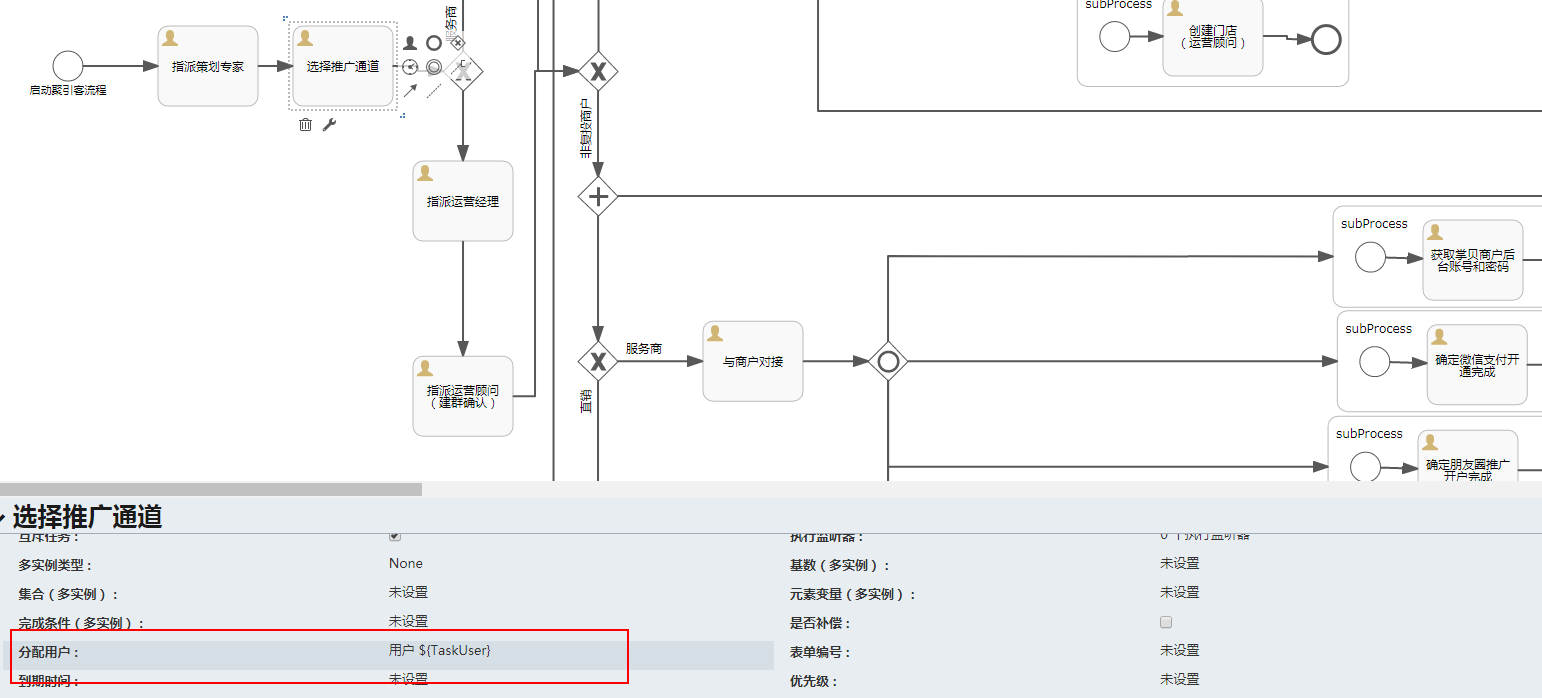
代码和流程图对应。

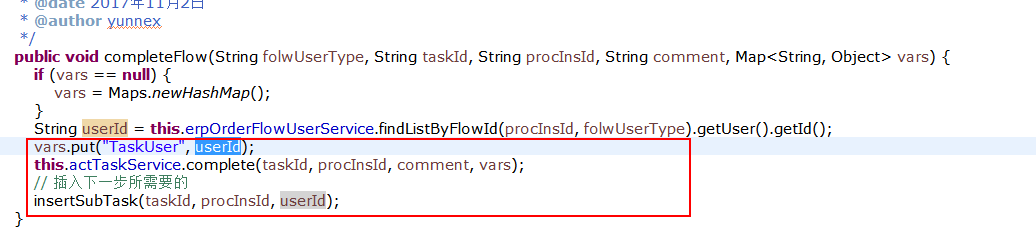




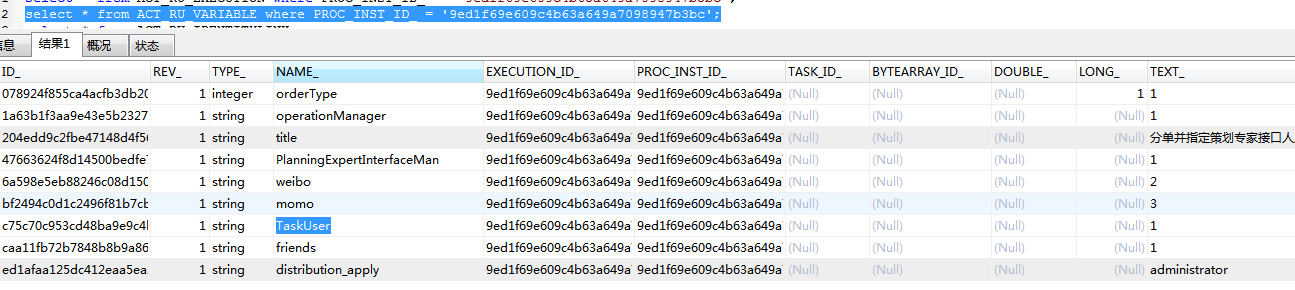
## 处理人

每个节点都设置了处理人流程变量。在审批之前会把这个变量设置进去





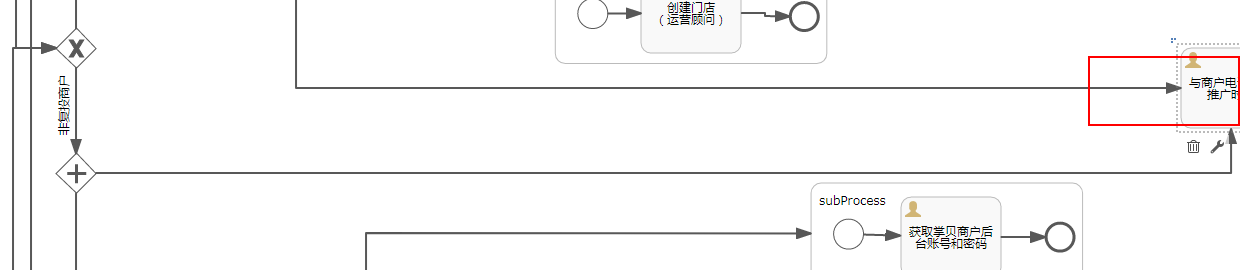
在流程变量中都可以找得到：



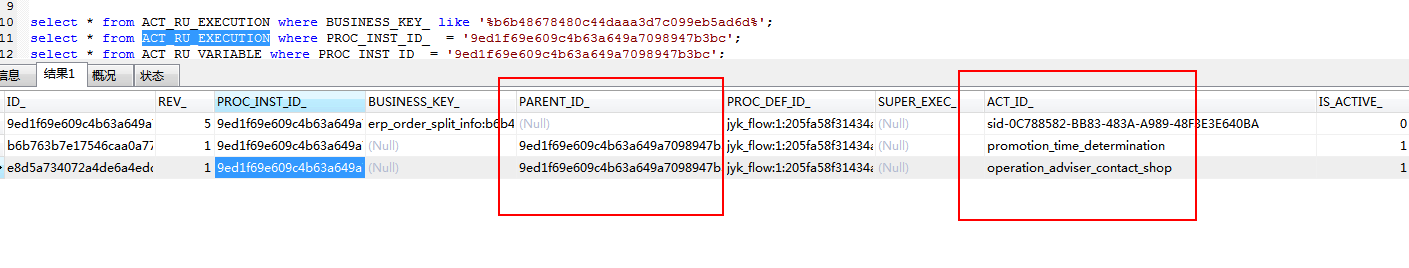
## 子流程和并行网关

当遇到子流程或者并行网关的时候，ACT\_RU\_EXECUTION会产生多条数据，根据PARENT\_ID\_判定父流程，但是PROC\_INST\_ID\_是一样的。主流程的EXECUTION\_id和PROC\_INST\_ID\_是同一个。

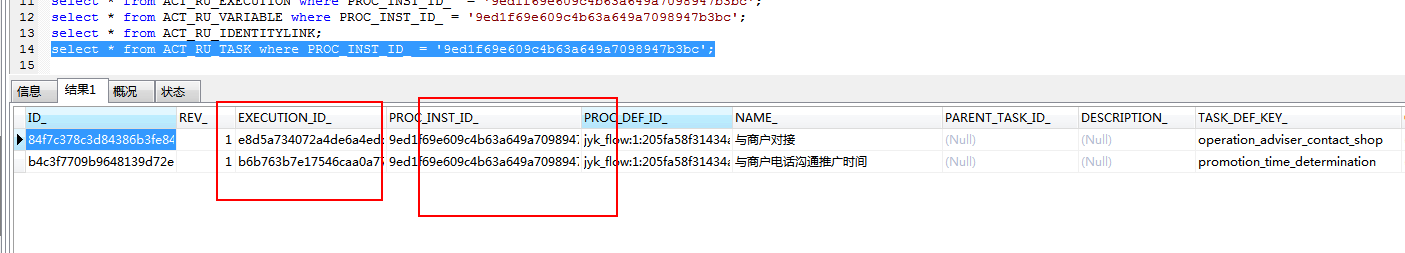
**举例并行网关：**



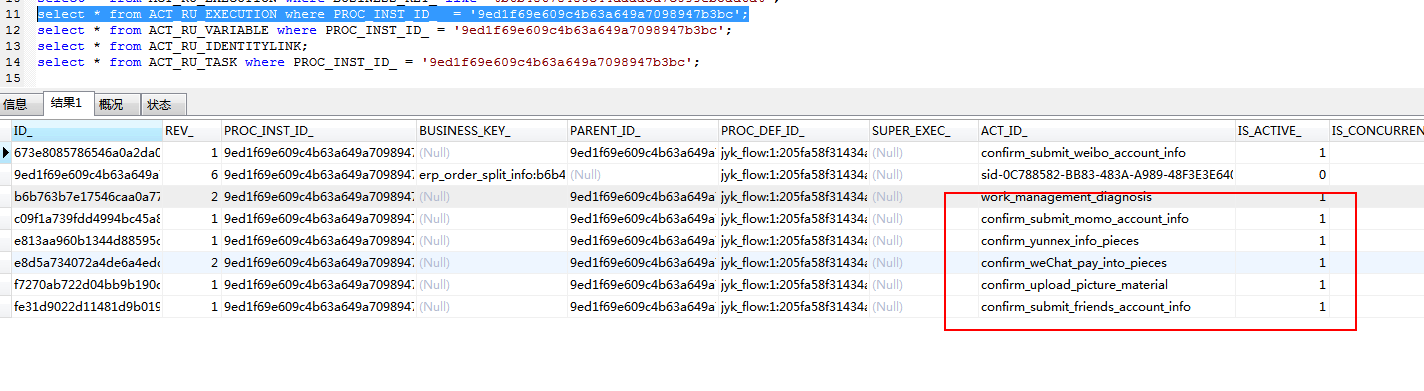
流程实例ID一样



任务表可以清楚看到当前节点和处理人



**举例子流程：**



可以看到，很多子流程的流程实例，PARENT\_ID\_都是 9ed1f69e609c4b63a649a7098947b3bc 。

注意，如果是并行网关任务发出的子流程，那么ACT\_RU\_EXECUTION表中会出现三层嵌套，其结构为：

PROC\_INST\_ID\_ 主流程

|

|

(并行任务1，PARENT\_ID\_是主流程) (并行任务2，PARENT\_ID\_是主流程)

|

|

(子流程1，PARENT\_ID\_是并行任务2)(子流程2，PARENT\_ID\_是并行任务2)

(子流程3，PARENT\_ID\_是并行任务2)

## 定位问题方法

1 根据订单ID找出相关的拆分的 流程实例是否异常

2 流程实例找到当前的节点，以及任务处理人是否异常

3 查看日志文件，分析每一步审批产生的异常

4 查看流程变量，查看是否缺少或者错误