**流程引擎**

## 一、工作流（Workflow）

是对工作流程及其各操作步骤之间业务规则的抽象、概括描述。工作流建模，即将工作流程中的工作如何前后组织在一起的逻辑和规则，在计算机中以恰当的模型表达并对其实施计算。工作流要解决的主要问题是：为实现某个业务目标，利用计算机在多个参与者之间按某种预定规则自动传递文档、信息或者任务。

## 二、工作流管理系统（WfMS）

**1 处理工作流的电脑软件系统，**其主要功能是通过计算机技术的支持去定义、执行和管理工作流，协调工作流执行过程中工作之间以及群体成员之间的信息交互。工作流需要依靠工作流管理系统来实现。工作流属于计算机支持的协同工作（CSCW）的一部分。后者是普遍地研究一个群体如何在计算机的帮助下实现协同工作的。

**2 J2EE项目中流行的工作流引擎** **1、JBPM 2、Activiti 3、Shark 4、osworkflow**

**3 工作流引擎的边界：**

1、必然存在自己的数据表，用来制定、存储工作流模板和实例

2、必然与操作用户关联

3、必然与用户表单关联

4、可以自定义模板，然后反复实例化（具体的每次运用）

5、模板的创建，可以图形化，可以用XML,可以用HTML本质都是把流程模板持久化

## 三、工作流常用术语

**1 节点**：操作的具体每个步骤（上图的，【项目组长审批】就是一个节点）

**2 过度**：过渡是连接各种工作流活动的通路。 节点通过过渡线相互连接以形成工作流。

**3 模板：**预先设置好一个工作流可能涉及到的节点、过度、关联用户，可以被多个具体的流程进行使用

**4 实例**：具体开始执行一个流程，实例会绑定上用户表单

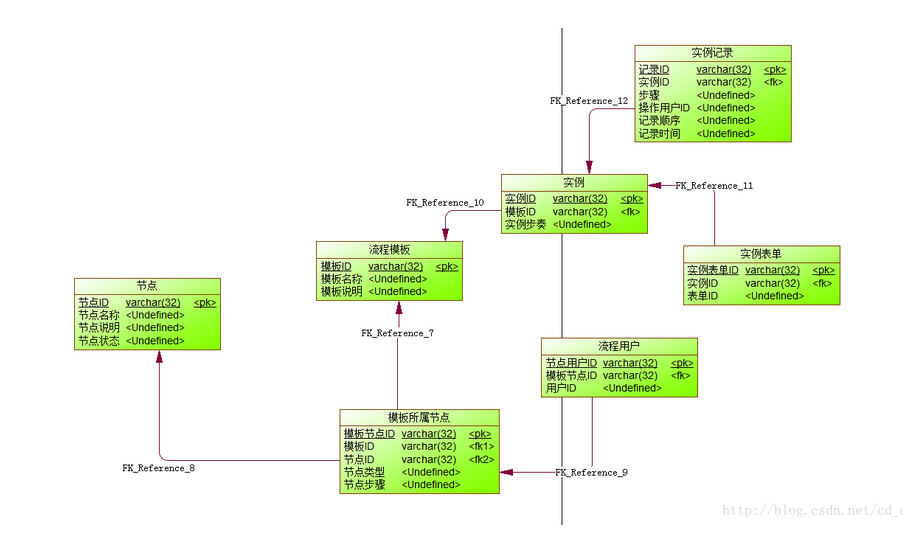
**5 历史轨迹：**执行任何一次步骤都被保存下来

**6 合并审批：** 如：总经理审批通过后需要，技术部副总裁和产品副总裁同时通过这个节点才算完成

## 四、优秀工作流引擎的特点

独立与应用系统存在（外挂式），一个工作流引擎可以外挂到所有需要用到工作流引擎的系统

## 工作流引擎PD分析



**1 节点表：** 也就是每个步骤的名称，独立存在不去主动关联任何表

**2 流程模板表 保存模板名称**

（1）模板关联到多个节点（关联的节点还要指定在模板中的位置，也就是第几步执行这个节点）

（2）模板关联的每个节点需要关联到多个操作人

**3 流程实例表 正在进行的流程**

（1）关联当前执行到的最新步骤

（2）关联模板可以查询出下一步执行节点

（3）实例必须关联用户自定义表单，流程才会与表单数据绑定，也就是审批的产品

**4 流程记录表：记录下实例化流程过程中的每一次操作**

## 六、流程引擎比较

**1 总览**

纵观jBPM：从jBPM3到jBPM5以及Activiti5：<http://www.infoq.com/cn/articles/rh-jbpm5-activiti5#>

工作流引擎选择（为何使用activiti而不是jbpm）：<http://blog.csdn.net/classfoo/article/details/20645779>

Java工作流引擎：jBPM、Activiti以及SWF：<http://blog.csdn.net/liangyixin19800304/article/details/12761573>

用OSWorkFlow和JBPM开发工作流异同：<http://blog.csdn.net/victor16345/article/details/5614676>

JBPM（Java Business Process Management）：JAVA业务流程管理，是一个可扩展、灵活、开源的流程引擎， 它可以运行在独立的服务器上或者嵌入任何Java应用中。

**2 几种工作流引擎对比**

（1）jBPM3是一个完整的工作流系统实现，面向开发人员，目的在于简化对组织核心流程进行支撑的软件创建，不支持标准。

（2）jBPM4引入PVM，使其拥有更强大的扩展性，同时增加BPMS特性，这些特性包括了对BPMN的支持、面向业务人员的Web建模器和简单统计分析功能的加入。

（3）jBPM5基于原先的Drools Flow，支持BPMN，通过与Drools的合并支持BAM，通过内容仓库增加对流程可视化的支持。由于放弃了jBPM4的PVM，引擎的可扩展性受到损害，并且不再支持jPDL。

（4）Activiti5基于jBPM4的开源工作流系统，与Alfresco的集成增加了其流程可视化与管理能力，同时通过创新的Activiti Cycle协作组件支持流程相关人员之间的协调，最后，它加强了集成能力。

（5）SWF与其说是工作流引擎，不如说是分布式计算调度框架，SWF中只包括Task和History两部分，甚至是每个Task之间如果要传递一些数据的话，都只能通过第三方存储（比如Message Queue或者Redis），不过这也给了编程更大的灵活性，问题是这种灵活性是不是非常需要。 一个SWF由Worker和Decider组成，Worker执行实际的任务，而Decider进行流程控制，两者严格上来讲没有区别，只是所执行的任务不同罢了。每个Worker和Decider会定期的去SWF的一个Task List取下一个任务。可以看出来这更像是一个“多线程”的结构，而SWF官方网站的Use Case是NASA的火星探索计划中需要处理图片的系统，这其实也是一个更多侧重于计算的系统，流程反而非常简单。另外，SWF（Simple Workflow）的一个Workflow不能太复杂，因为所有的流程控制都集中于Decider，如果太复杂的话Decider将无比庞大，给维护和扩展带来一定的困扰。

**3 Activiti的优势：**

**（1）与jBPM4相比，Activiti5最令人瞩目的特性就在于它的协作工具组件。**

Activiti Modeler—建模器，基于开源Signavio Web流程编辑器的一个定制版本，提供了对BPMN2.0图形化规范的支持，建模后的流程以文件格式进行存储。

Activiti probe—管理及监控组件，对流程引擎运行期实例提供管理及监控的Web控制台。包含部署的管理、流程定义的管理、数据库表的检视、日志查看、事务的平均执行时间、失败多次的工作等功能。

**（2）Activiti拥有更简洁健壮的接口**

Activiti中提供TaskQuery接口，可以设置各种查询过滤，排序方式，最终通过list方法执行查询，相比jbpm，它还提供了分页查询功能，双方高下立判。

**（3）Activiti拥有更友好的用户体验**

JBPM核心引擎完全没有关于表单的任何抽象，它的工作机制是通过全局常量，流程变量，任务变量，这些概念十分技术化。

相比之下Activiti则更贴近实际的应用场景，它将为开始节点，以及人工任务提供了表单设置，用户可以设置字段名称，字段类型。通过Activiti的平台可以根据这些设置去生成表单，

但如果不使用其平台只使用引擎的话，也支持通过它来表达与第三方表单的关系。这些表单设置的元数据信息也可以通过接口去获取。

**（4）Activiti支持启动引擎后随时热部署**

JBPM存在一个软肋，一个RuntimeService只能在启动的时候指定bpmn资源，一旦启动后便不再能够去更新或者增加bpmn了，这会导致我们系统集成的困难，因为我们自然希望整个系统只有一个工作流引擎实例运行。Activiti则提供了Deploy机制，将bpmn资源的热部署，热更新都做了很好的支持

**（5）Activiti拥有更友好易用的Eclipse编辑插件和在线插件**

**（6）Activiti依赖更少的jar包**

Activiti依赖的第三方jar包较少，主要就是mybatics，而JBPM则依赖了一大堆的jar，从drools到繁杂的hibernate，再到自身拆分的零零散散的jar包，让人不由觉得它是一个庞大的怪物。工作流有版本的概念，jBPM和Activiti上传一个新的版本后，版本号会增加1，旧版本还没执行完的流程实例还会继续执行。SWF的版本是个字符串，随意指定好了，这样也很好，字符串名称更明确。嵌入式部署即将流程引擎嵌入部署于Web应用中。

**4 总结**

shark：系统和功能都比较复杂

Osworkflow：比较灵活的轻量级的框架，但是在流程建模方面不太友好，需要手动编写xml文件去定义流程文件。

SWF：还有不能支持太复杂的流程

**5 J2EE常用工作流比较**

|  |  |  |  |
| --- | --- | --- | --- |
| **[引用链接](http://blog.csdn.net/shadowkiss/article/details/8895378" \t "https://blog.csdn.net/SAM_XIE_52/article/details/_blank)** | **Shark(EnhydraShark)** | **Osworkflow opensymphony** | **JBPM(JBoss JBPM)**  **Java Business Process Management** |
| **工作流描述语言** | XPDL：WFMC制定的描述业务流程控制流的XML格式规范，格式复杂，与具体语言无关，不灵活 | 1、XML：流程定义格式简单，使用灵活  2、基于有限状态机模型 | 1、JPdl：JBoss jBPM Processdefinition language，一个商务流程被看作是一个UML状态图。  2、基于UML的状态图和活动图来定义流程，已加入JBOSS大家庭，市场前景看好。 |
| **是否开源，开源协议** | 一个可扩展的工作流引擎框架。现在不再开源，用于商业用途 | 开源的嵌入式工作流引擎 ，它的使用遵循 Apache License | 一种基于J2EE的轻量级工作流管理系统，它的使用遵循 Apache License |
| **相关开源项目** | Jawe | Osworkflow for .net |  |
| **支持是否全面** | 流程定制工具JAWE | 1、带有一个简陋的流程定制工具，但十分简陋且常有错误  2、需要专业技术用户使用 | 1、Jbpm3的图形化流程定义嵌入到jbosseclipse IDE  2、流程定制方式更接近用户的理解 |
| **拓展性** | 体系和功能最为复杂，秉承“模块化”的思想，比较容易扩展 | 有良好的扩展性，绝对的灵活（同时也增加了开发者的工作量，需要自己写一些必要的函数） | 最适宜扩展（Jbpm的过程模式支持是比较固定的，但是其对任务的中action扩展是很的灵活）最适宜被商业化应用 |
| **持久化** | Shark的持久层采用DODS来实现 | 1、它提供的持久化API：EJB,Hibernate,JDBC等  2、Osworkflow 可以与Spring集成。 | 利用hibernate持久化 |
| **小结** | Shark是体系和功能最为复杂的代表。它是一款遵循WfMC的XPDL标准开源工作流引擎，并且同时遵循OMG组织的WorkflowManagement Facility规范。XPDL的两个最重要的概念是Process和Activity。XPDL中的Activity是基于UML1.x中的活动图的概念。活动图天生的适于工作流程建模，它相对于状态图的一个最大的优点是容易做并发线程的分叉控制，这些并发线程可以同时执行也可以顺序执行；它还有一个优点是有泳道的概念，可以控制工作流引擎中的任务的产生。      在所有开源工作流引擎中，Shark的体系最为完备和复杂。其一直秉承着“模块化”的思想，所以比较容易扩展。但是自从被Together公司收购后，Shark的商业化色彩已经越来越浓，改称为Together Workflow Server，并仅以Community Edition的形式提供了部分开源代码供参考。 | OSWorkflow是最轻量型的代表，也是一款非常灵活和低级别定位的工作流引擎的实现框架。低级别定位的意思是说，它不是定位在解决流程模型对象和运转场景，而是提供一套可维护调度的机制，供开发人员自主扩展。这个维护流程调度机制OSWorkflow选择的是基于行为（Action）的FSM理论，所以OSWorkflow更像是一个复杂而灵活的有限状态调度机。      Osworkflow有个重要概念是State，State是由step和status联合表达的，一个State就是一个step中的某个status；而state的转换由action来驱动，类似状态图中的event,因为一个event对应一个action      OSWorkflow在国内项目应用得较多，很多国内的简易审批流程项目都是基于其引擎二次开发而来。这主要是由于OSWorkflow是基于Action驱动的，而国内的客户也很容易接受这样的操作习惯。但OSWorkflow所依赖的FSM模型对于分支、聚合、子流程的支持度很低，这一点在实施过程中需要注意。 | Jbpm结合应用了状态图+活动图+PetriNet的知识，而且这里的活动图还是UML2.0版的。UML2.0的活动图中，节点不叫活动（Activity）而叫动作(action)，活动成了一个高层次的概念，它包含一个动作序列。一个活动图展现一系列的动作，这些动作组成了活动。Jbpm把action也改名了，称为state。Jbpm使用的状态图的概念有transition/event等。Jbpm来内部实现中还采用了PetriNet的概念，如token,signal等，jBpm对Token的应用很有特色，巧妙地利用Parent-Child Token的机制处理分支、父子流程等复杂应用场景。      jBpm是最适合扩展的代表，是在所有开源引擎中最适宜被商业化应用的一款。首先其流程建模模型是基于Activity Diagram（活动图）的，并在引擎构建上融入了FSM和PetriNet思想，所以其内核和根基比较牢固扎实。其次，自从被JBoss收购后，其3.x系列的结构更加趋于微内核，Plug-in思想也更加深入。其同时还提供了对BPEL扩展，存储支持JBossHibernate实现，集成了JBoss seam，规则引擎准备采用JBossrules，并准备集成JBoss Messaging。这样，不论从内核和外围应用，jBpm都具有了强劲的动力。 |

**6 用OSWorkFlow和JBPM开发工作流异同**

|  |  |  |
| --- | --- | --- |
| **[引用链接](http://blog.csdn.net/victor16345/article/details/5614676" \t "https://blog.csdn.net/SAM_XIE_52/article/details/_blank)** | **JPBM** | **OSWorkFlow** |
| **编写流程描述文件方式** | BPM是通过图形化的编辑工具（JBPM自带的Eclipse插件）来编写业务流程如开始状态，结束状态，以及状态之间的转换，之后会自动生成XML文件，但具体每一步相关的细节操作还是要手工配置（如该节点是属于什么类型节点，相关的函数，要不要较验）。 | OSWorkFlow则要通过手工写XML文件来定义流程文件，而且它涉及的标签元素比较多，其用户手册上也建议实施人员不要去修改流程，否则流程很容易破坏。 |
| **工作流信息保存方式** | JBPM是将流程信息直接保存到数据库，可以用任何方式对数据库的操作，这样就要引入保存工作流信息的相关表。 | OSWorkFlow是既可以保存在XML文件里，也可以保存在数据库中，保存在数据库中时需要配置propertySet.xml文件，比较复杂，而且它不是完全支持hibernate（如引入osuser来作权限分析时），此时要自定义操作数据库的方式。 |
| **与系统集成** | JBPM集成比较容易，加入Spring支持包spring-modules-jbpm31.jar,该包加入了Spring对JBPM的包装，所有的集成都是在此包基础之上。之后还要配置sessionFactoryForJbpm ,jbpmConfiguration,jbpmTemplate, jbpmDao。 | OSWorkFlow跟Spring集成须要如下所需组件：      1)SpringHibernateWorkflowStore，让工作流程实例(如果需要的话)分享当前事务。      2) SpringTypeResolver，允许 OSWorkflow 从 Spring ApplicationContext中获得业务逻辑组件(conditions, functions等等)。      3)SpringConfiguration， 这是一个 Workflow Configuration 接口的实现类， 它包含指向 store和 factory的引用，这样可以在 spring 中注射或者连接。      4) SpringWorkflowFactory，这是一个 XMLWorkflowFactory 封装包，它可以允许从容器中注入 configuration，从而不再从其它的 XML 配置文件中读取它们。      如果OSWorkFlow引入osuser.xml来设置权限，则不支持Hibernate3，因为osuser是比较独立的模块，目前还没有支持hibernate3，所以跟Spring集成时要修改配置文件applicationContext-hibernate3.xml |
| **重点难点** | JPDL语言的学习，主要是用来编写流程文件；理解3个接口:动作处理接口(提供影响流程执行的方法，在event和action元素中被回调)，判定处理接口（用在decision判定节点中，提供方法来判定节点的转向），委派处理接口（用在task的委派子元素assignment中，用来指定将任务分配给指定的人员或角色）。 | 工作流文件定义的元素，主要用来编写工作流；OSWorkFlow.xml及propertySet.xml文件的配置；InputMap接口、Workflow 接口及WorkflowDescriptor接口。 |
| **开发步骤** | 通过图形编辑工具编写业务流程――>生成xml流程文件――>修改流程文件（判断节点是任务节点、普通节点还是判定节点,之后作相应修改）――>导入保存流程信息的数据库表――>部署流程定义文件（将流程文件中的内容放到数据库中）――>创建流程实例――>调用JBPM提供的signal方法执行流程流转 | 手工编写工作流文件——>配置OSWorkflow.xml――>配置WorkStore——>配置propertySet.xml,osUser.xml文件（如果需要用户权限）――>调用AbatractWorkflow类加载OSWorkflow.xml(它会自动加载工作流文件及数据库配置文件)――>调用WorkFlow接口方法(initial()方法,transitionWorkflow（）方法，doAction（）方法) |
| **流转方法** | 先确定节点是什么节点，如果是普通的Node节点，则是流程执行到此节点不会中断，继续执行；如果是state节点，则流程执行到此节点会中断，直到系统外的参与者发会命令才能继续执行，即调用signal（）或end（）方法；如果节点是Task-node，则会根据task任务列表的任务有没全部执行完来决定流转。 | 一个工作流包含多个步骤。每一个步骤都有一个当前状态(例如, Queued, Underway, or Finished)。每一个步骤中都有一个或者多个动作可以被执行。每一个动作都可以设置执行条件(condition)，也可以设置执行函数(pre-function or post-function)。动作产生结果(result)，导致工作流的状态和当前步骤发生改变 |
| **流程定义文件主要元素** | 一个JBPM的流程定义XML文件中包含一个< process-definition>元素，而一个< process-definition>元素又包含零个或一个< description>元素，零个或多个的< swimlane>元素，一个< start-state>元素，零个或多个的< state>元素或< decision>元素或< fork>元素或< join>元素，以及零个或多个的< action>元素，零个或多个<task-node>和<node>元素，一个< end-state>元素等等。此外，< process definition>元素有一个标示符，以“name”属性来表示，这个属性必须存在，用来表示该流程的名称。 | 步骤（step）、条件（conditions）、循环（loops）、分支（spilts）、合并（joins）、角色（roles）、函数（function） |
| **其他** | JBPM的Scheduler可以实现在JBPM流程中定时触发某一动作。在流程中JPBM提供了timer节点供我们使用，通过这个节点我们可以实现节点动作的定时触发。 |  |

**7 深入了解jBPM5与Activiti之间的差异对比**

|  |  |  |
| --- | --- | --- |
| **[引用链接](http://blog.csdn.net/hanxuemin12345/article/details/43303767" \t "https://blog.csdn.net/SAM_XIE_52/article/details/_blank)** | **Activiti** | **JBPM5** |
| **相似之处** | 1、都是BPMN2过程建模和执行环境。  2、都是BPM系统（符合BPM规范）。  3、都是开源项目-遵循ASL协议（ Apache的 软件许可）。  4、都源自JBoss（Activiti5是jBPM4的衍生，jBPM5则基于Drools Flow）。  5、都很成熟，从无到有，双方开始约始于2年半前。 都有对人工任务的生命周期管理。  6、Activiti5和jBPM5唯一的区别是jBPM5基于WebService - HumanTask标准来描述人工任务和管理生命周期。 如有兴趣了解这方面的标准及其优点，可参阅WS - HT规范介绍 。  7、 都使用了不同风格的 Oryx 流程编辑器对BPMN2建模。 jBPM5采用的是 Intalio 维护的开源项目分支。 Activiti5则使用了Signavio维护的分支。 | |
| **数据库持久层ORM** | MyBatis3 | Hibernate3 |
| **持久化标准** | 无 | JPA规范 |
| **事务管理** | MyBatis机制/Spring事务控制 | Bitronix，基于JTA事务管理 |
| **数据库连接方式** | Jdbc/DataSource | Jdbc/DataSource |
| **支持数据库** | Oracle、SQL Server、MySQL等多数数据库 | Oracle、SQL Server、MySQL等多数数据库 |
| **设计模式** | Command模式、观察者模式等 |  |
| **内部服务通讯** | Service间通过API调用 | 基于Apache Mina异步通讯 |
| **集成接口** | SOAP、Mule、RESTful | 消息通讯 |
| **支持的流程格式** | BPMN2、xPDL、jPDL等 | 目前仅只支持BPMN2 xml |
| **引擎核心** | PVM（流程虚拟机） | Drools |
| **技术前身** | jBPM3、jBPM4 | Drools Flow |
| **所属公司** | Alfresco | jBoss.org |
| **集成** | Activiti5使用Spring进行引擎配置以及各个Bean的管理，综合使用IoC和AOP技术，使用CXF作为Web Services实现的基础，使用MyBatis进行底层数据库ORM的管理，预先提供Bundle化包能较容易的与OSGi进行集成，通过与Mule ESB的集成和对外部服务（Web Service、RESTful等）的接口可以构建全面的SOA应用； | jBPM5使用jBoss.org社区的大多数组件，以Drools Flow为核心组件作为流程引擎的核心构成，以Hibernate作为数据持久化ORM实现，采用基于JPA/JTA的可插拔的持久化和事务控制规范，使用Guvnor作为流程管理仓库，能够与Seam、Spring、OSGi等集成。 |
| **优劣对比** | 从技术组成来看，Activiti最大的优势是采用了PVM（流程虚拟机），支持除了BPMN2.0规范之外的流程格式，与外部服务有良好的集成能力，延续了jBPM3、jBPM4良好的社区支持，服务接口清晰，链式API更为优雅；Activiti上手比较快，界面也比较简洁、直观        劣势是持久化层没有遵循JPA规范。 | jBPM最大的优势是采用了Apache Mina异步通信技术，采用JPA/JTA持久化方面的标准，以功能齐全的Guvnor作为流程仓库，有RedHat(jBoss.org被红帽收购)的专业化支持；        但其劣势也很明显，对自身技术依赖过紧且目前仅支持BPMN2。 |

**Activiti**

## 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。

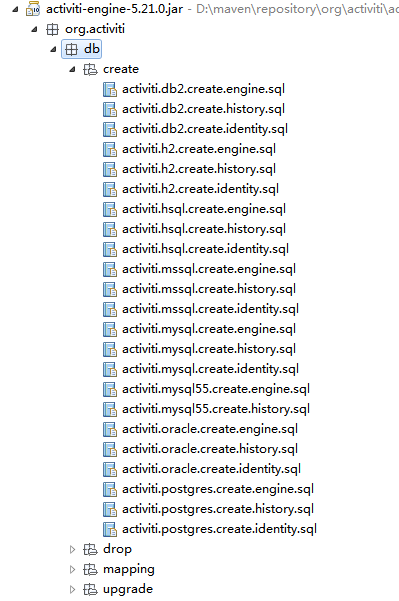
ACT\_ID\_\*: 'ID'表示identity。 这些表包含身份信息，比如用户，组等等。

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

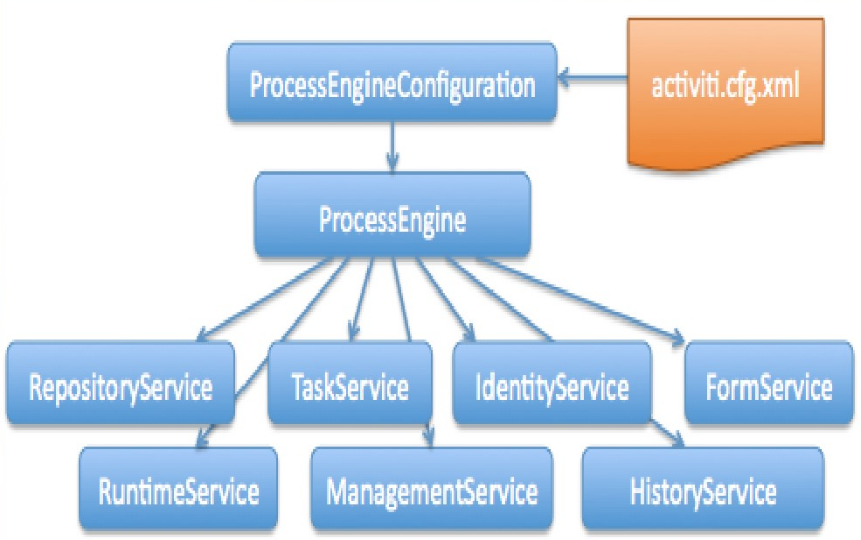
ProcessEngine 是Activiti中最核心的类，其他的类都是由他而来。

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

RepositoryService repositoryService =processEngine.getRepositoryService();

RuntimeService runtimeService = processEngine.getRuntimeService();

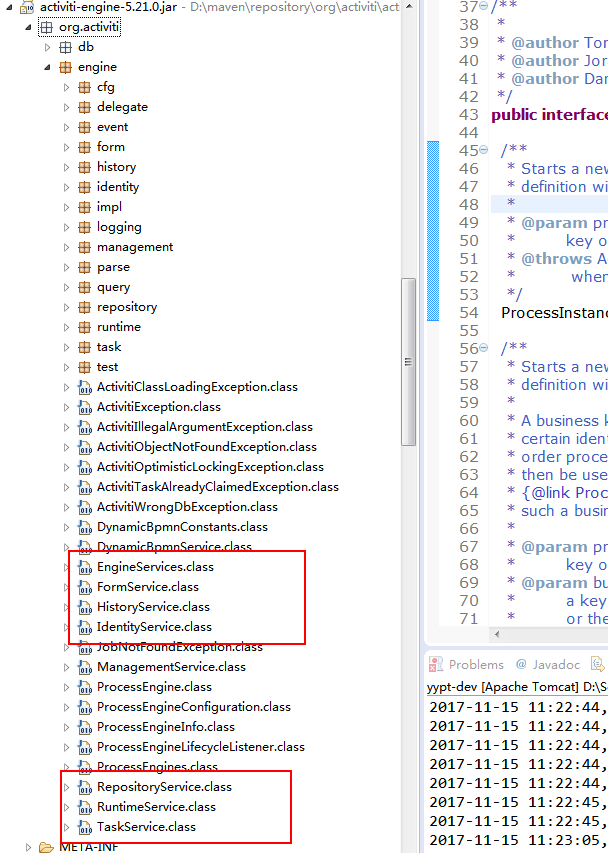
TaskService taskService =processEngine.getTaskService();



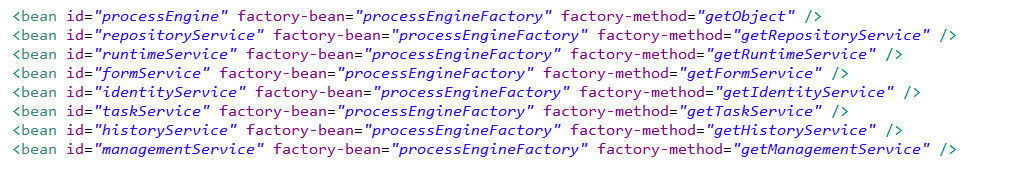
**各个Service的作用：**

|  |  |
| --- | --- |
| **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());

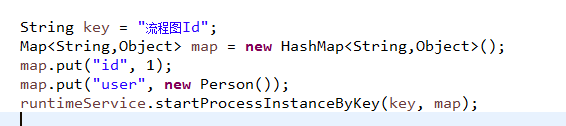
## 流程变量

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



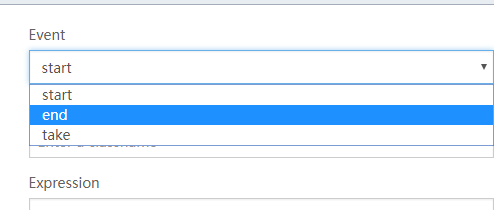
**setVariable：设置流程变量的时候，流程变量名称相同的时候，后一次的值替换前一次的值，而且可以看到TASK\_ID的字段不会存放任务ID的值**

**setVariableLocal：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。**

## 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)) {

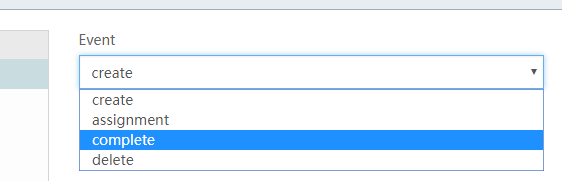
System.out.println("end=========");

}

}

}

**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>

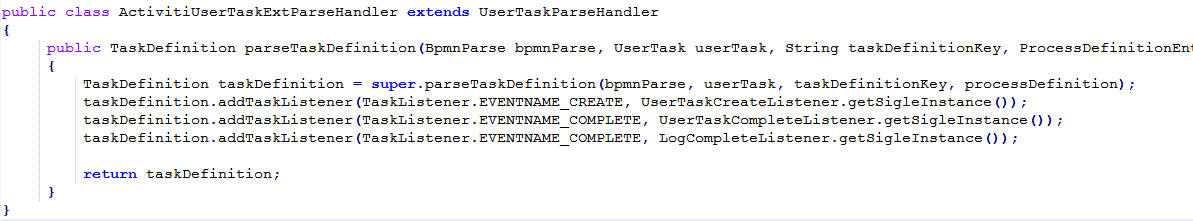
</property>

</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。



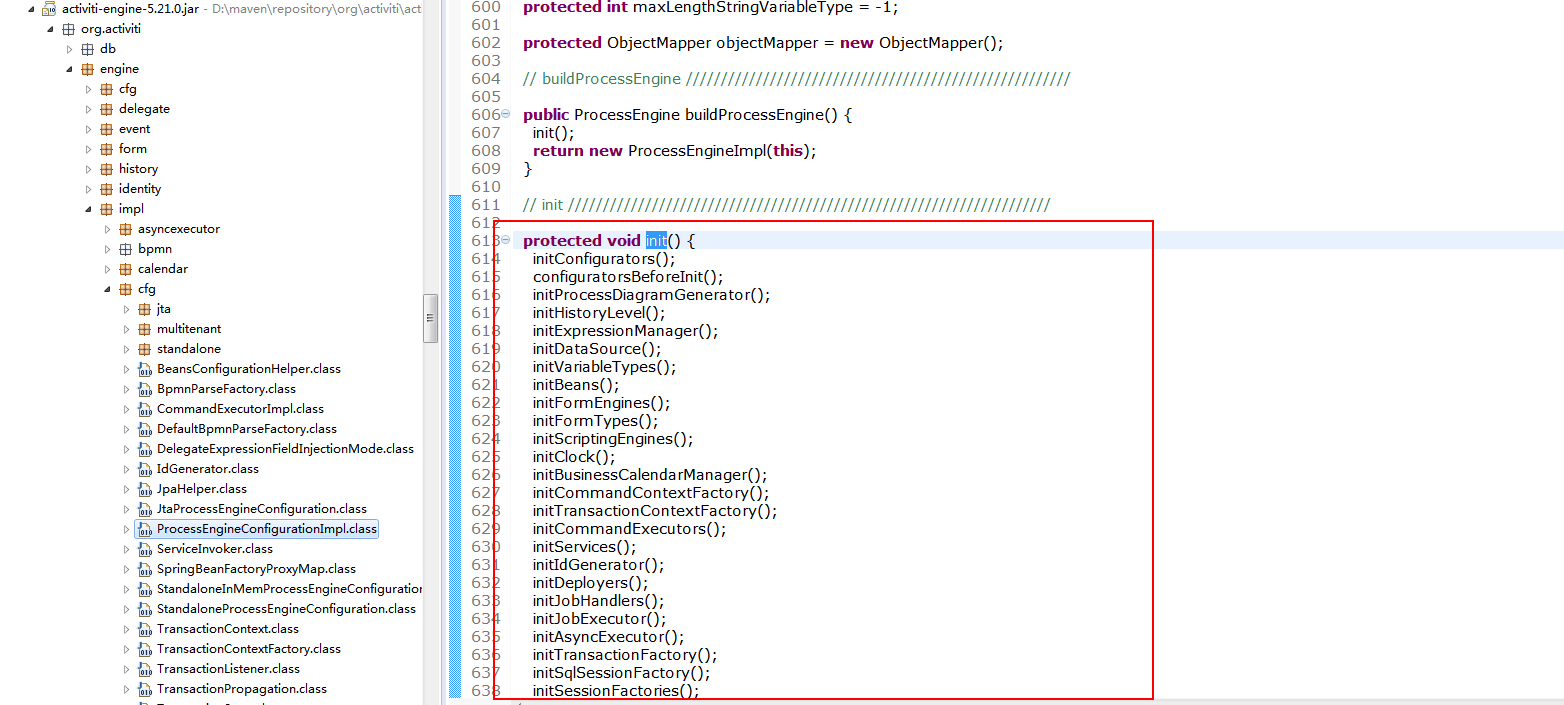
## 命令模式和责任链模式

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。

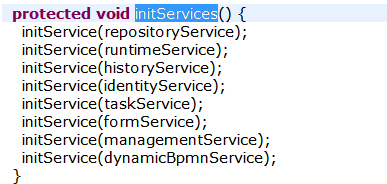
**以启动流程为例：**

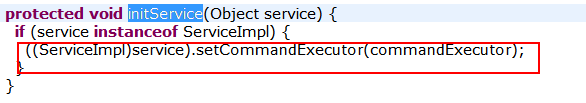
ProcessEngineConfigurationImpl 会初始化一系列配置和注入信息，包括 initCommandExecutors();



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)

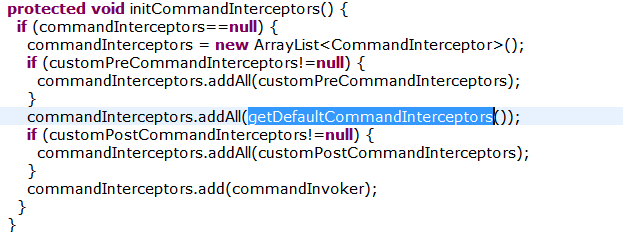
{

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);

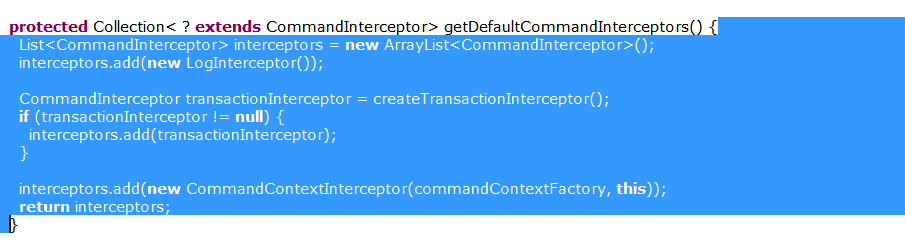
Activiti默认的拦截器：

    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.

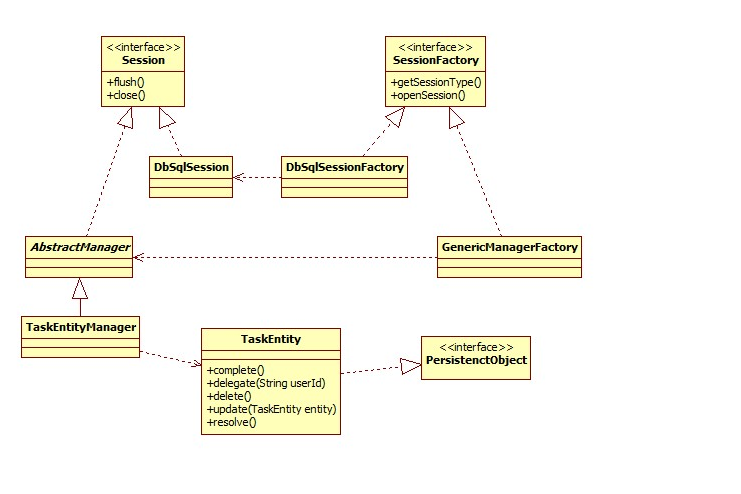
     4 CommandInvoker拦截器,执行命令



**操作数据库：**



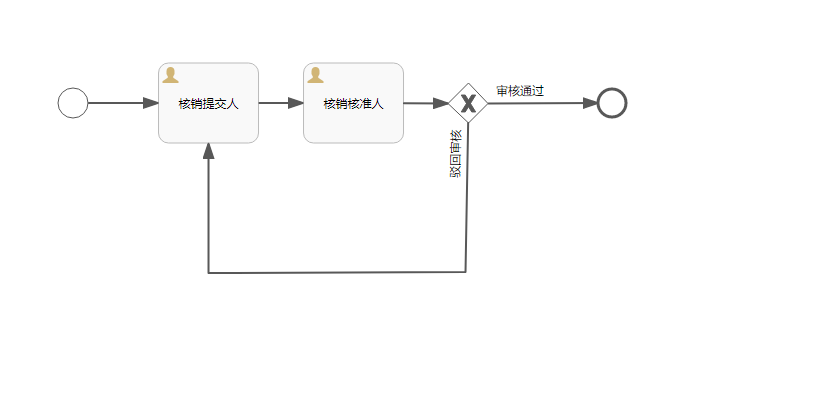
Activiti使用ibatis作为ORMapping工具。在此基础之上Activiti设计了自己的持久化框架



## 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);

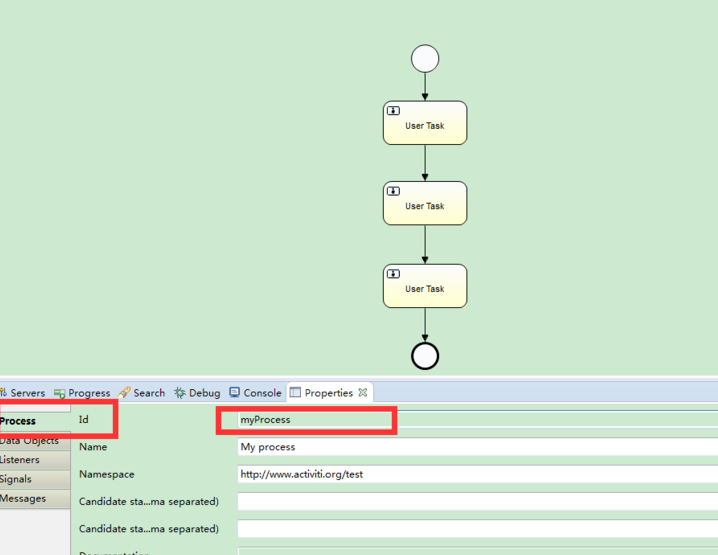
}

}

## 九、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")

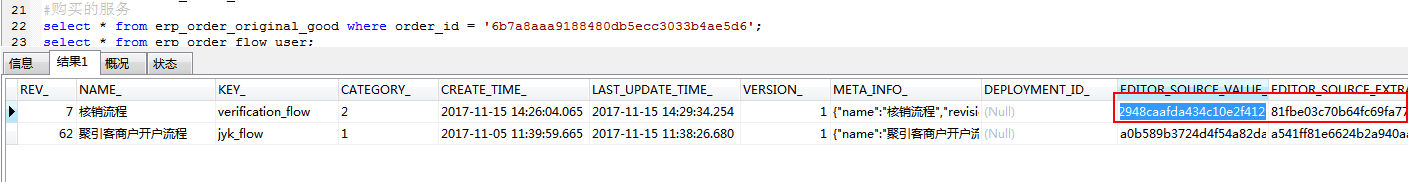
.addClasspathResource("diagrams/verification\_flow.bpmn") ;

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

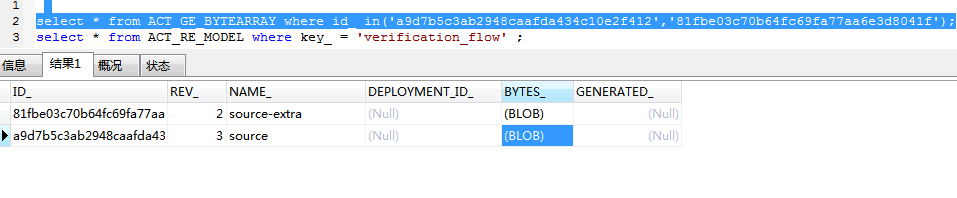
（1）新建模型，编辑流程，流程分类在数据字典编辑，新增一个分类 .

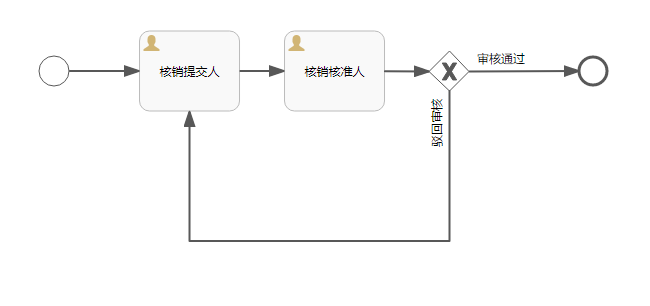


这个时候 ACT\_RE\_MODEL新增加一条记录，EDITOR\_SOURCE\_VALUE\_ID\_和EDITOR\_SOURCE\_EXTRA\_VALUE\_ID\_分别对应ACT\_GE\_BYTEARRAY的id. 存储模型的JSON信息和图片信息。



JSON文件和图片信息，可以另存为





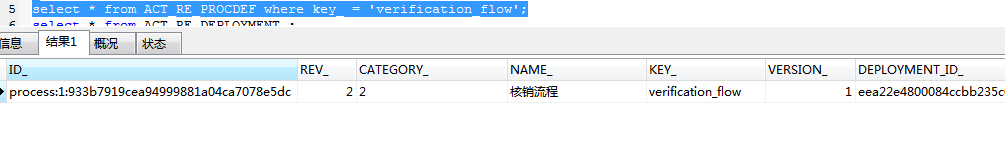
保存在数据库中的JSON 文件。

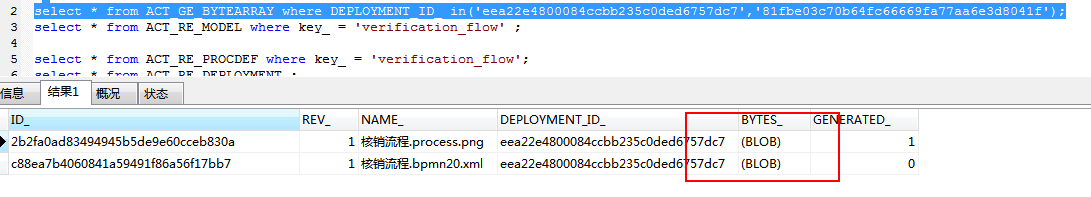


1. 部署流程

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。

1

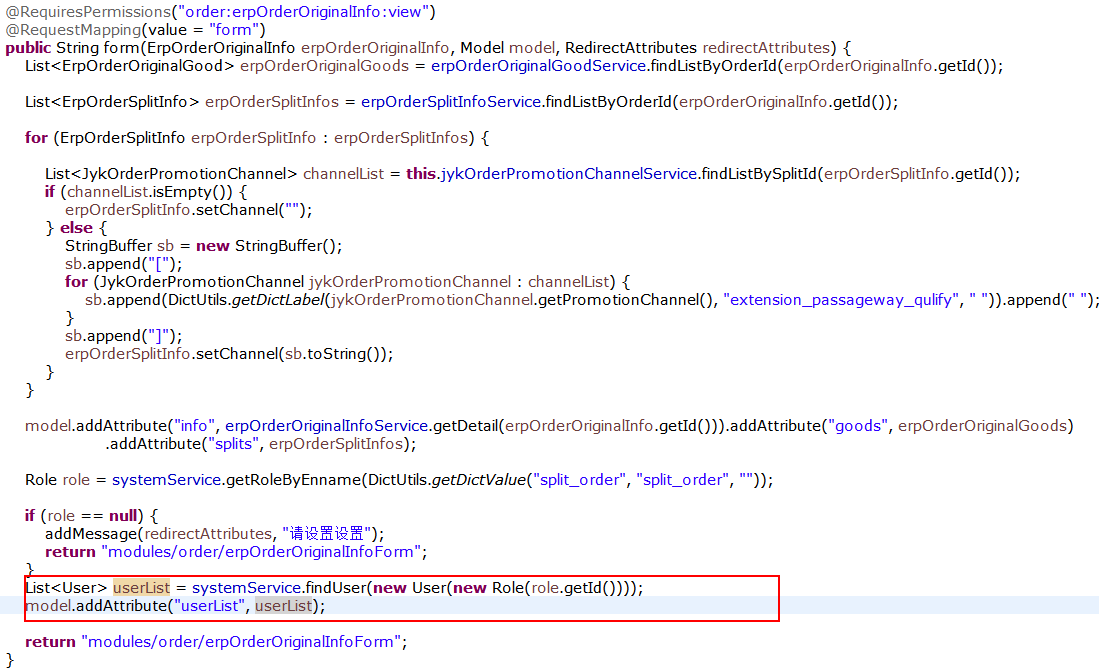
2

3

## 十、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。

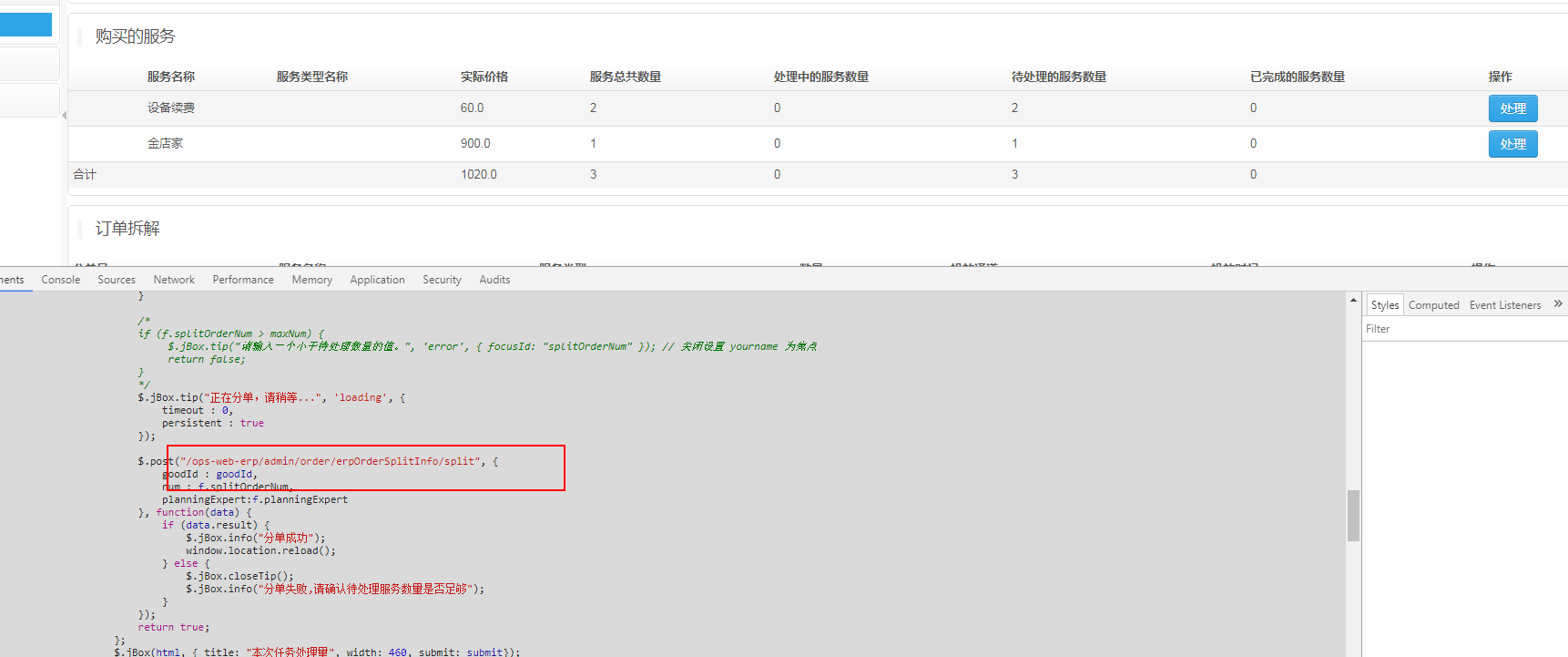


erpOrderOriginalInfoForm.jsp

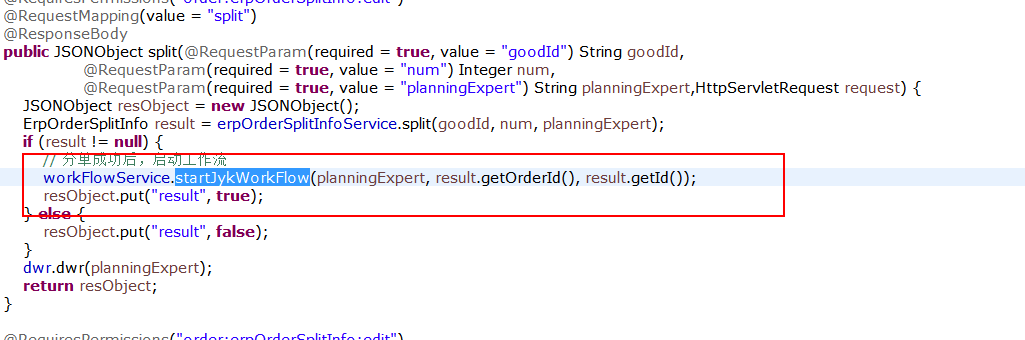
+"<c:forEach items='${userList }' var='user'>"

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

+"</c:forEach>"



ErpOrderSplitInfoController：拆单控制器





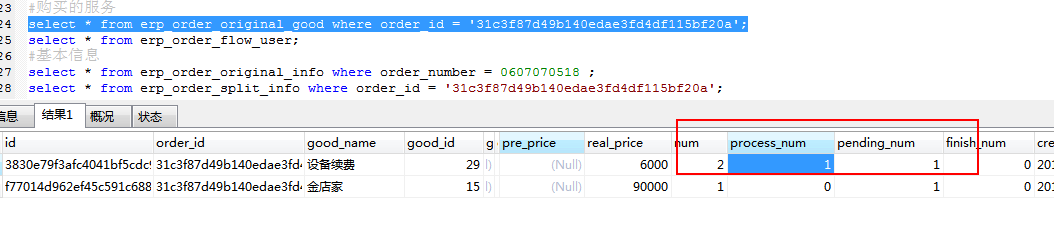
拆单的业务逻辑：

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

找到order\_id ：31c3f87d49b140edae3fd4df115bf20a。

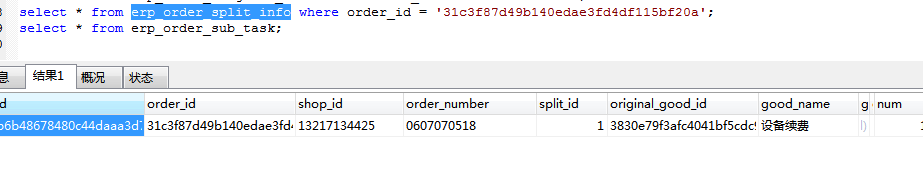


erp\_order\_original\_good订单商品表的处理数量和待处理数量改变

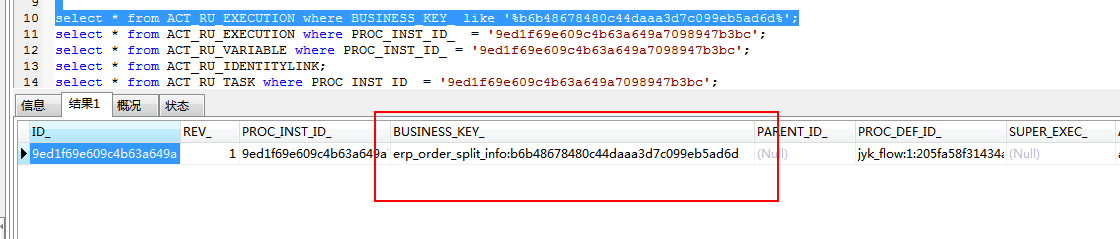


erp\_order\_split\_info表新增一条拆单信息，订单ID对应订单表的order\_id

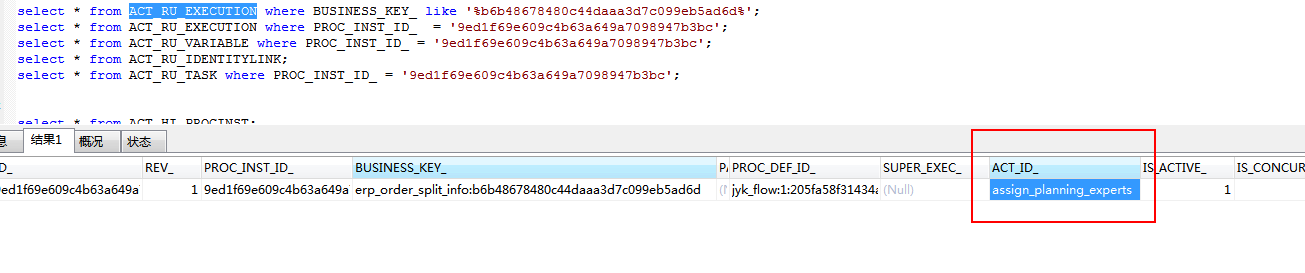
该表的id作为流程实例的businessKey .

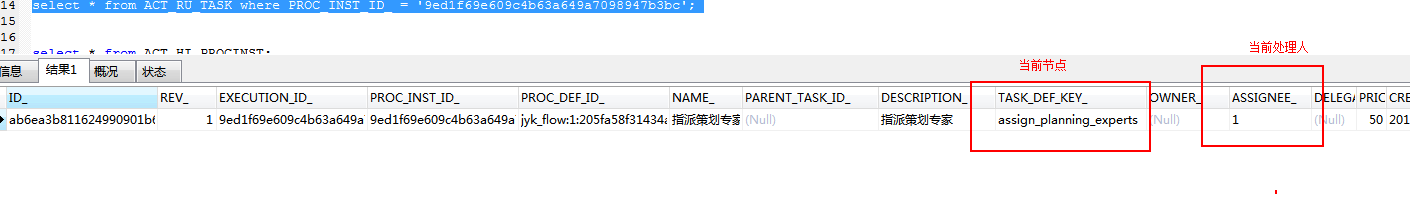


ACT\_RU\_EXECUTION表启动一个流程，businessKey用固定前缀+split\_info.id .其实完全可以直接用split\_info.id。



查看任务和当前节点





。

## 十一、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;

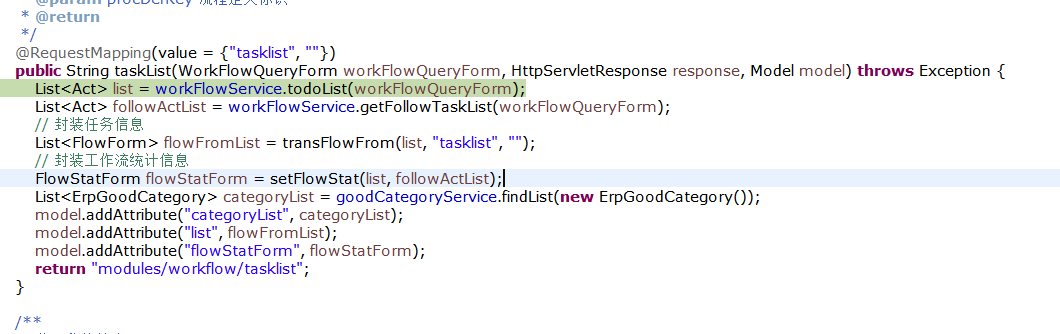
select \* from sys\_user\_role;

select \* from sys\_role\_menu;

select \* from sys\_menu;

## 十二、流程流转

查询我的任务：WorkFlowController



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

.desc();

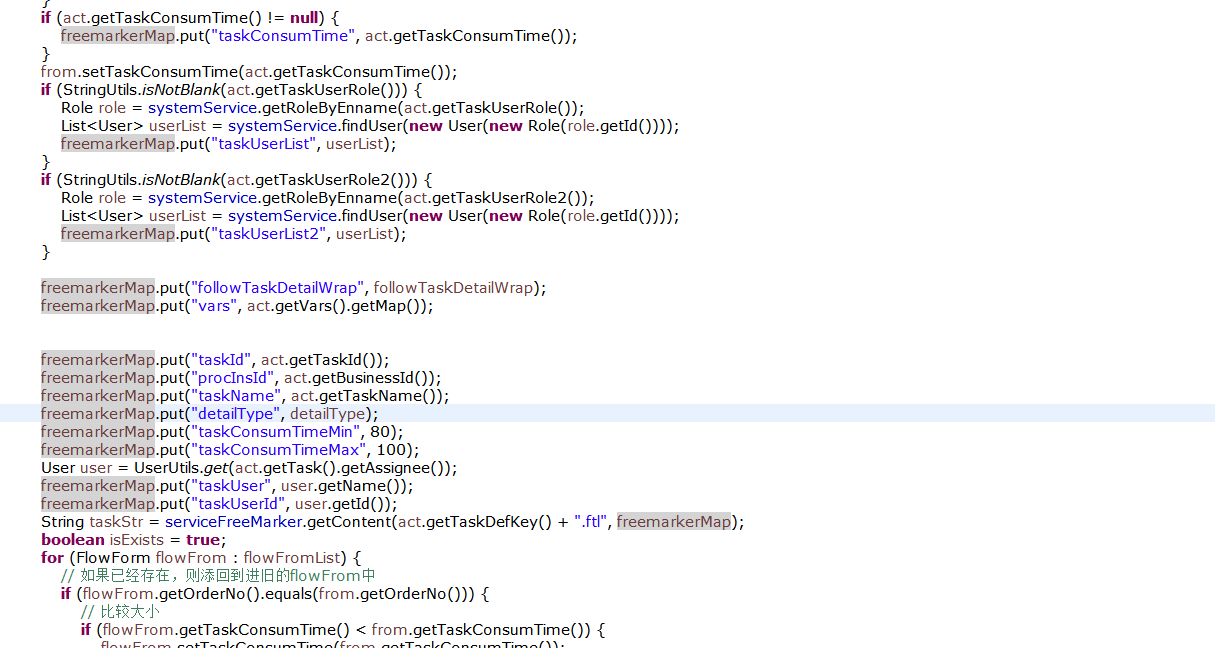
查询出所有代办，根据查询参数过滤结果集。

获取节点表单的数据：



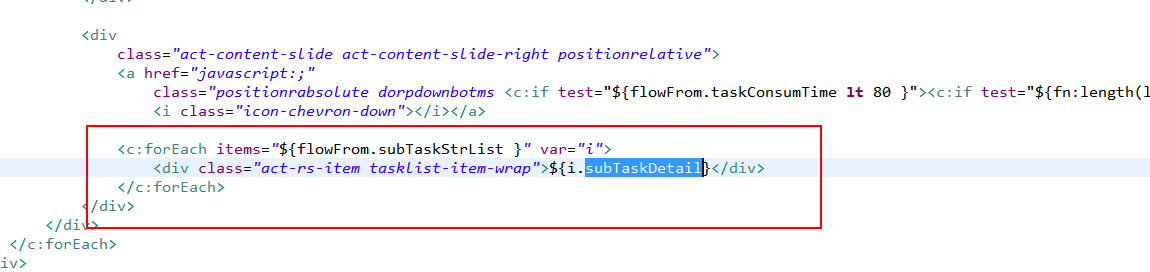
6

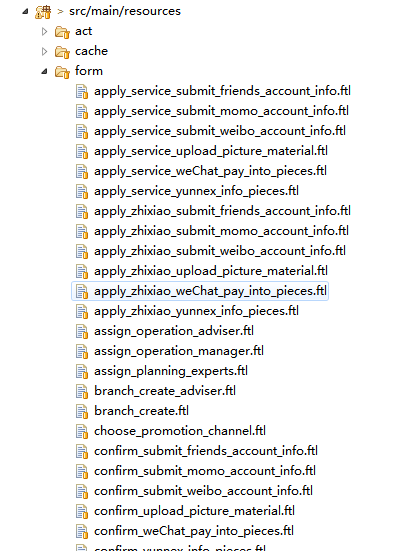
根据对应的模板文件设置参数，放入taskDetail字段中。



显示模板代码到前台tasklist.jsp,循环显示 子任务

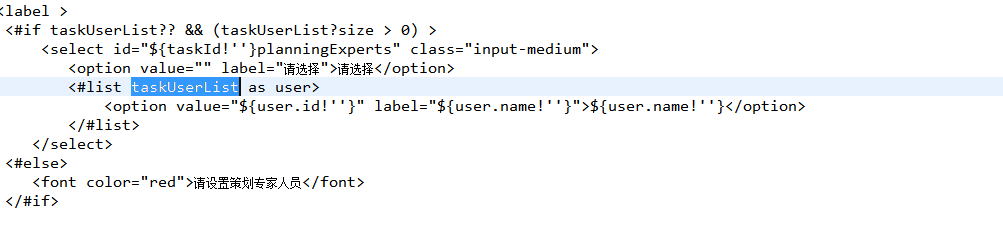
每个节点有对应的模板文件





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





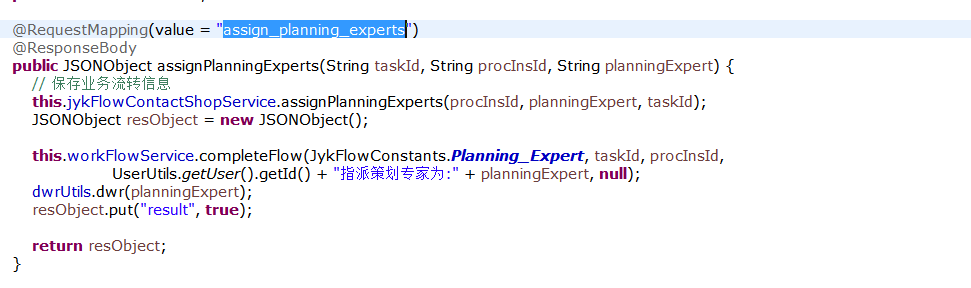
审批逻辑

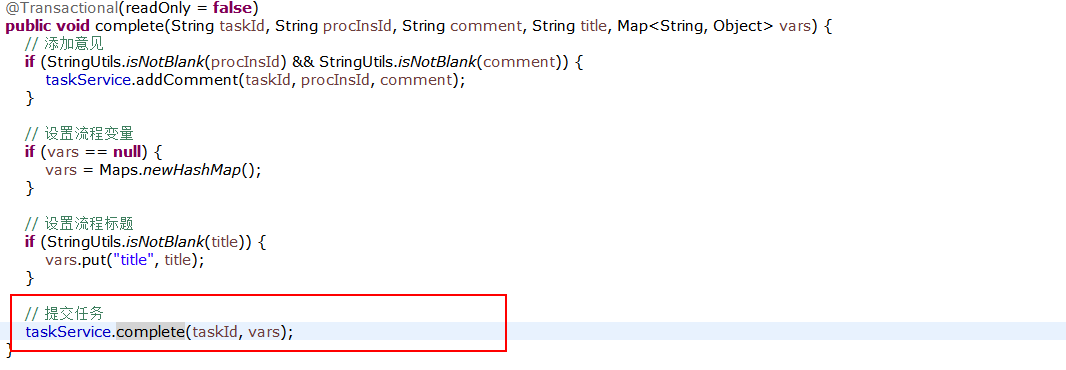
JykFlowContactShopController

JykFlowProductController

JykFlowServiceOrderController

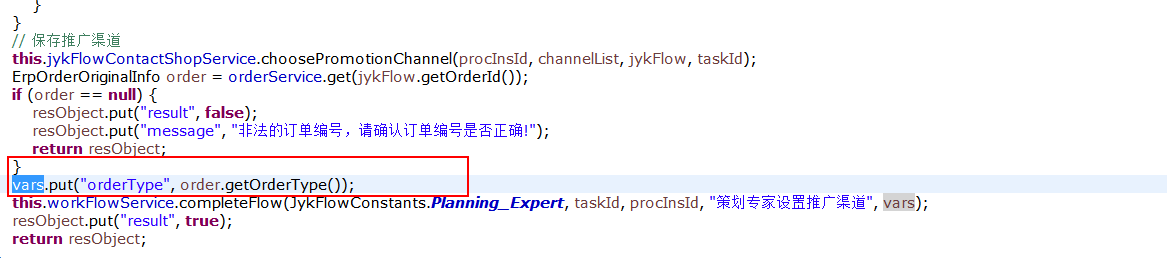
这三个controller封装了所有节点的审批逻辑，一个节点对应一个方法。

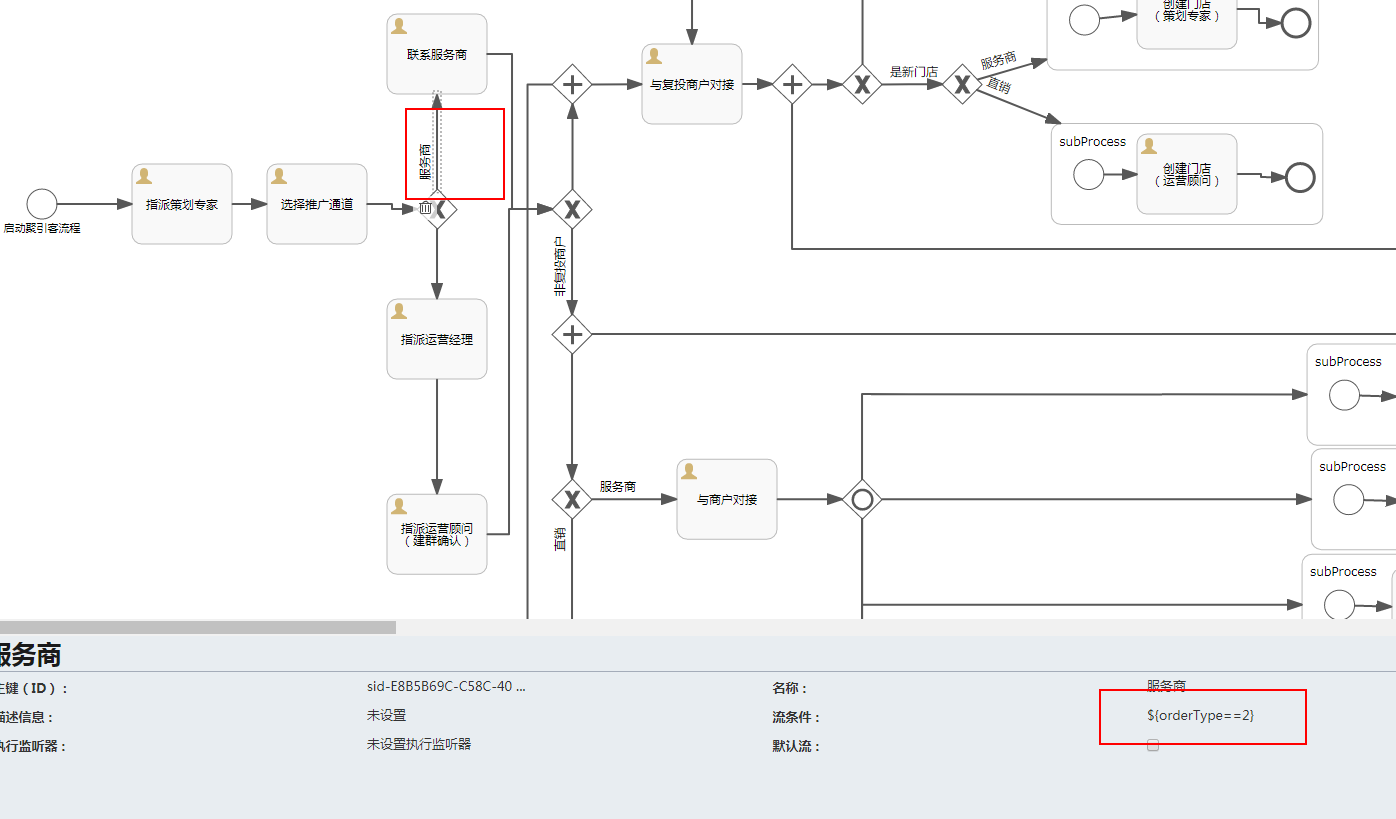




路由参数的设置：

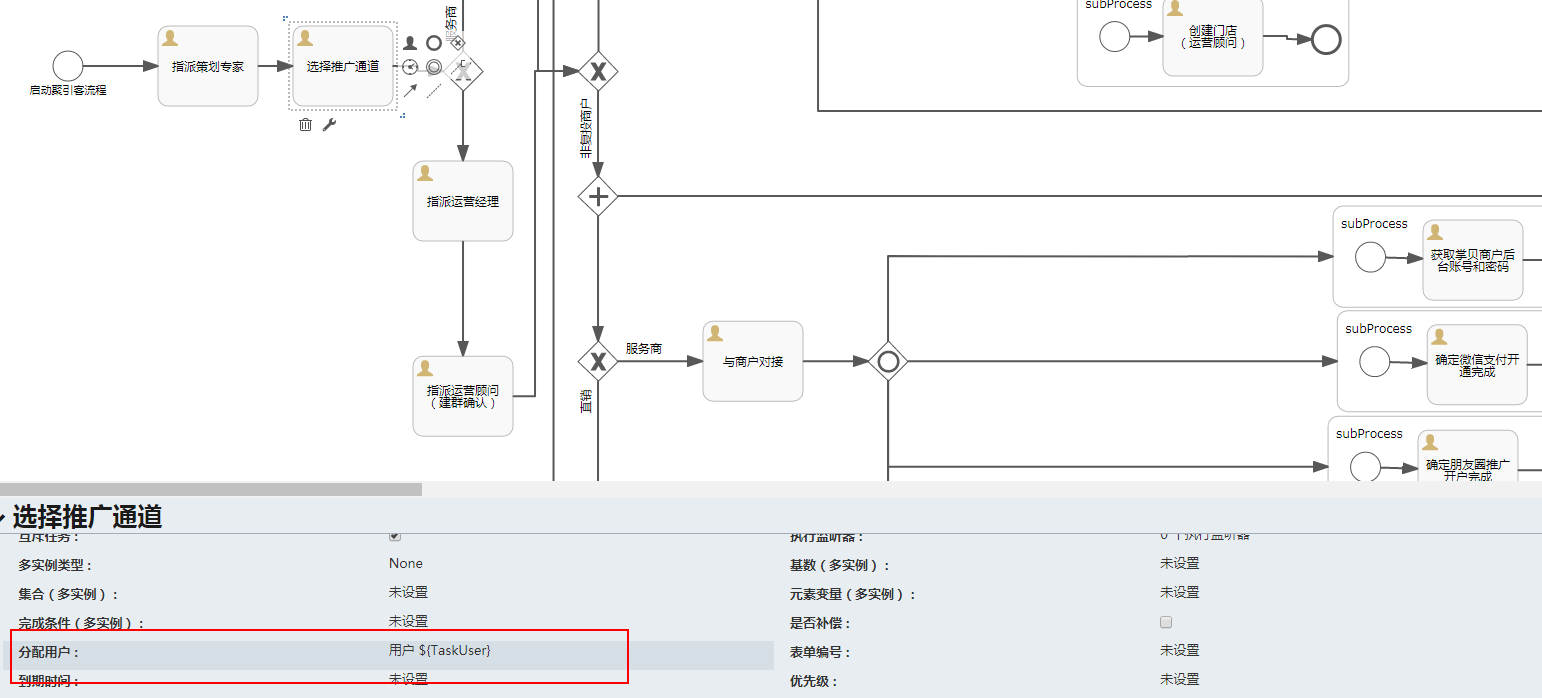
代码和流程图对应。

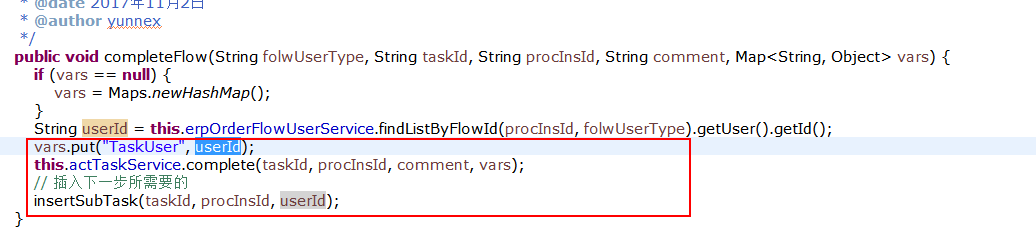




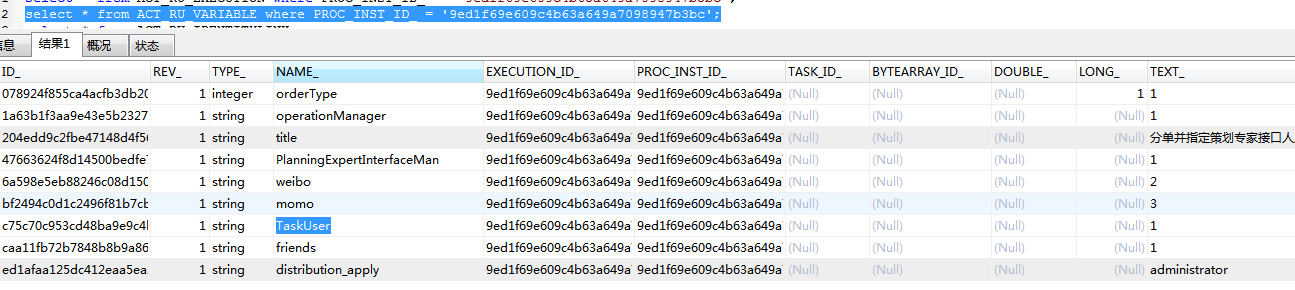
## 十三、处理人

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





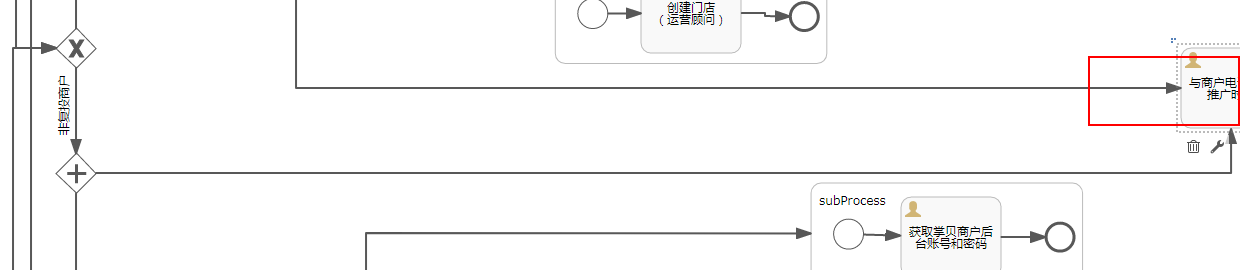
在流程变量中都可以找得到：



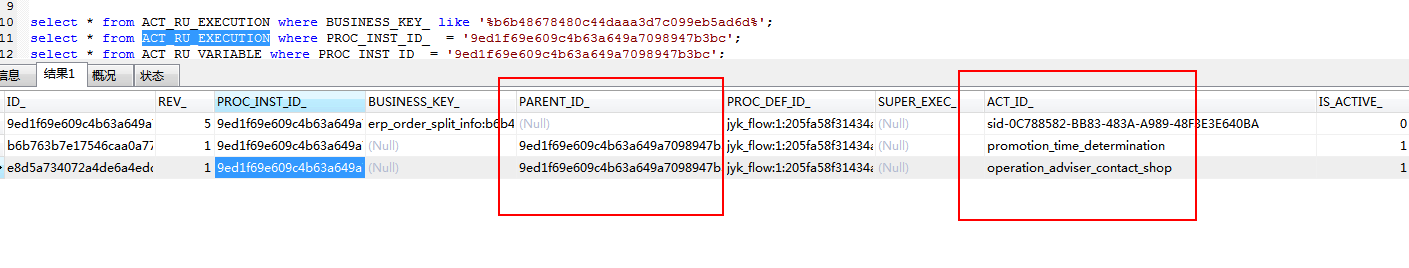
## 十四、子流程和并行网关

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。

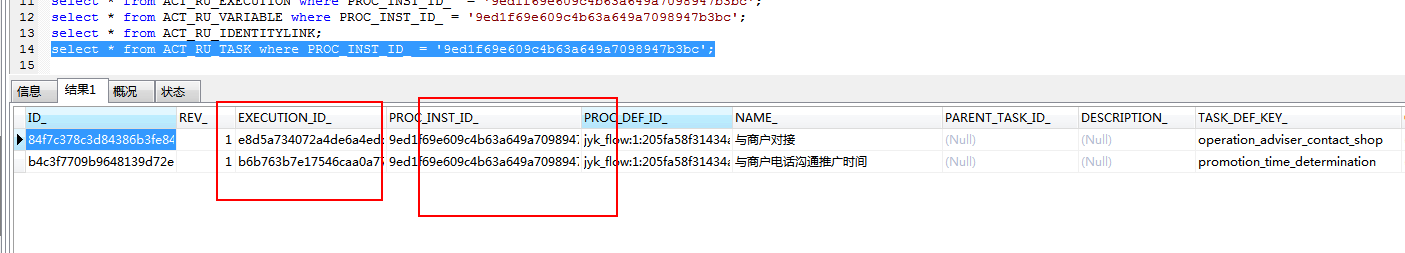
**举例并行网关：**



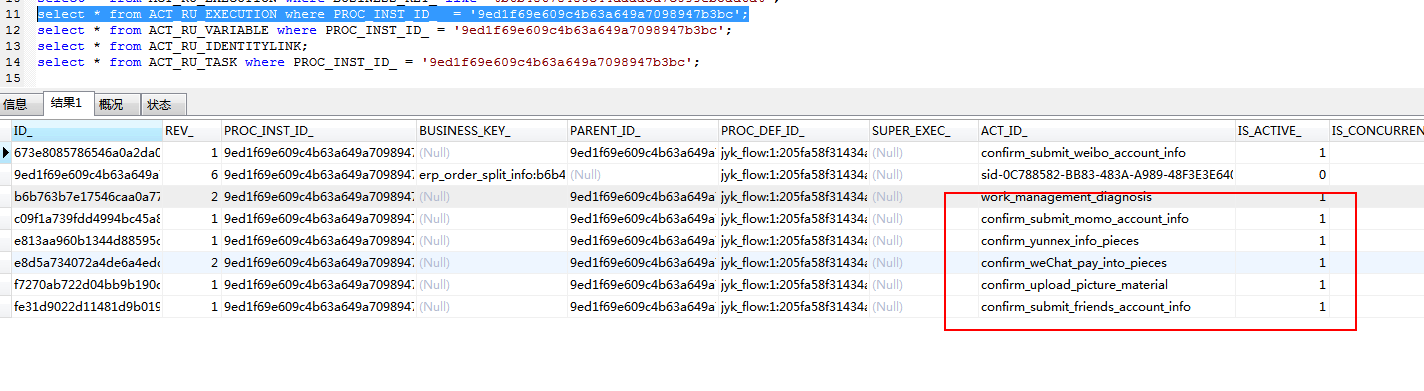
流程实例ID一样



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)

## 十五、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