

Activiti

工作流引擎

**讲师：传智.燕青**

采购系统学习

业务流程

采购系统完成了企业的日常采购管理，包括采购单管理、采购单审核、入库、结算等模块。

用户角色包括：员工、部门经理、总经理、财务。

基本业务流程如下：



## 业务的概念

什么是业务？

用户的需求就是业务。

什么是业务功能？

通过计算机技术实现用户需求，通过计算机研发软件功能，所开发的软件功能就是业务功能。

什么是业务系统？

实现用户需求，所开发的软件系统，叫业务系统。

什么是业务系统核心（灵魂）？

技术不是业务系统核心，业务流程是业务系统的核心。

什么是业务流程？

很多的功能组成一个流程，完成用户需求为目标，叫业务流程。

## 数据库设计

采购单表：pur\_bus\_order

采购单审核表：pur\_bus\_order\_audit

用户表：pur\_sys\_user

角色表：pur\_sys\_role

用户角色关系表：pur\_sys\_user\_role

## 功能设计

流程简化为：



### 员工创建采购单

#### 操作流程：

员工登录系统

进入创建采购单

填写采购单信息，点击保存

进入采购单待提交页面

选择需要提交的采购单点击提交

#### 操作：

* 保存采购单

提交后向pur\_bus\_order表写一条记录：

Status：写入1（未提交）

* 查询待提交采购单

查询pur\_bus\_order表的记录，只查询status值等于1(未提交)的记录

* 提交采购单

根据主键更新pur\_bus\_order表中的status字段，更新为2（部门经理审核中）

### 部门经理审核采购单

#### 操作流程：

部门经理登录系统

进入采购单审核页面查询待审核信息

点击审核，进入审核页面

输入审核信息

点击提交

#### 操作

* 采购单待审核信息

查询pur\_bus\_order表的记录，只查询status值等于2 (部门经理审核中)的记录

* 提交审核信息

向pur\_bus\_order\_audit表插入一条记录（审核信息）

根据主键更新pur\_bus\_order表的记录

当采购单审核不通过时：status值更新为3(部门经理审核不通过)

当采购单审核通过时，status值更新为4(总经理审核中)

### 总经理审核采购单

参考部门经理审核实现

#### 操作流程

总经理登录系统

进入采购单审核页面查询待审核信息

点击审核，进入审核页面

输入审核信息

点击提交

#### 操作

* 采购单待审核信息

查询pur\_bus\_order表的记录，只查询status值等于4 (总经理审核中)的记录

* 提交审核信息

向pur\_bus\_order\_audit表插入一条记录（审核信息）

根据主键更新pur\_bus\_order表的记录

当采购单审核不通过时：status值更新为5(总经理审核不通过)

当采购单审核通过时，status值更新为6(财务审核中)

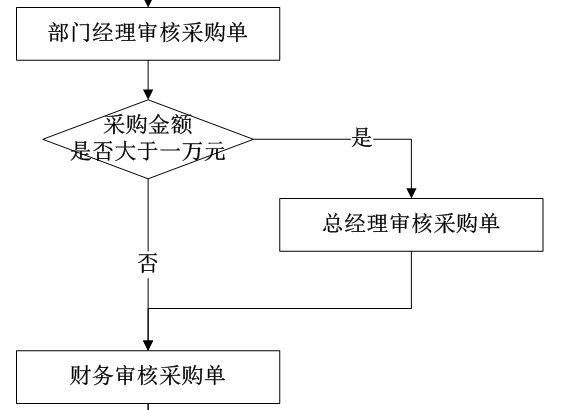
### 财务审核采购单

参考部门经理审核实现

## 需求变更

### 流程变更1

针对上边的流程变更如下：



需要修改部门经理审核采购单功能，判断采购金额：

如果大于1万元则更新pur\_bus\_order表中的status字段为4总经理审核中

如果小于等于1万元则更新pur\_bus\_order表中的status字段为6财务审核中

### 流程变更2

将上边的流程中总经理审核和财务审核位置颠倒，变为：



针对上边的流程变更需要将财务审核采购单和总经理审核采购单功能全部修改。

比如财务审核采购单修改：

审核通过更新pur\_bus\_order表中的status状态为4（总经理审核中）

## 结论

业务系统是通过在采购单表中添加状态字段对业务流程进行管理，在采购单创建提交、部门经理审核、总经理审核、财务审核等操作时在程序代码中对状态值进行硬编码，如果业务流程变更需要修改代码，不利于系统维护。

如果我们采用工作流可以解决上边的问题。

**建议：**

**对于简单的业务流程，流程节点少，流程变化不大，建议业务系统自己管理流程，因为采用工作流将增加工作量和开发成本。**

**对于复杂的业务流程，流程节点多，流程变化较大，建议使用工作流，虽然增加了工作量和开发成本，但是将有利于系统扩展维护，降低系统维护成本，因为系统上线后主要的工作量是维护。**

# 什么是工作流

## 工作流介绍

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

## 工作流系统

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

### 适用行业：

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

### 具体应用

1. 关键业务流程：订单、报价处理、采购处理、合同审核、客户电话处理、供应链管理等
2. 行政管理类:出差申请、加班申请、请假申请、用车申请、各种办公用品申请、购买申请、日报周报等凡是原来手工流转处理的行政表单。
3. 人事管理类：员工培训安排、绩效考评、职位变动处理、员工档案信息管理等。
4. 财务相关类：付款请求、应收款处理、日常报销处理、出差报销、预算和计划申请等。
5. 客户服务类：客户信息管理、客户投诉、请求处理、售后服务管理等。
6. 特殊服务类：ISO系列对应流程、质量管理对应流程、产品数据信息管理、贸易公司报关处理、物流公司货物跟踪处理等各种通过表单逐步手工流转完成的任务均可应用工作流软件自动规范地实施。

# 什么是Activiti

## Activiti介绍

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

Activiti是一个工作流引擎， activiti可以将业务系统中复杂的业务流程抽取出来，使用专门的建模语言（BPMN2.0）进行定义，业务系统按照预先定义的流程进行执行，实现了业务系统的业务流程由activiti进行管理，减少业务系统由于流程变更进行系统升级改造的工作量，从而提高系统的健壮性，同时也减少了系统开发维护成本。

### BPM

BPM（Business Process Management），即业务流程管理，是一种以规范化的构造端到端的卓越[业务](http://baike.baidu.com/view/64906.htm)流程为中心，以持续的提高组织业务[绩效](http://baike.baidu.com/view/122994.htm)为目的系统化方法，常见商业管理教育如[EMBA](http://baike.baidu.com/view/29918.htm)、MBA等均将BPM包含在内。

企业流程管理主要是对企业内部改革，改变企业职能管理机构重叠、中间层次多、流程不闭环等，做到机构不重叠、业务不重复，达到缩短流程周期、节约运作资本、提高企业效益的作用。

比如下边的两个采购流程哪个效率更高？





上边两个流程的区别在于第二个流程在执行时，如果采购金额在一万元以内不再由总经理审批将比第一个流程缩短流程周期，从而提交效率。

再比如下边的例子，哪个效率更高？





上边两个流程的区别在于第二个流程将交费和取药放在一起进行，这样导致的结果是此窗口的工作人员必须具备财务、药学专业知识，岗位强度加大，人员培训难度加大从而导致人员不易扩展，工作效率低下。

### Bpm软件

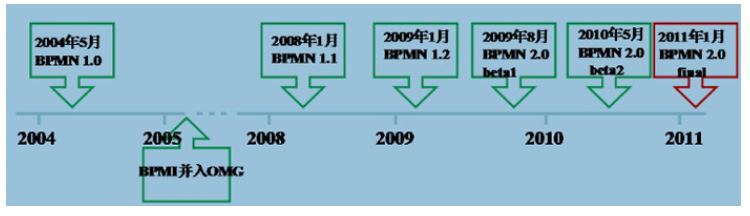
Bpm软件就是根据企业中业务环境的变化，推进人与人之间、人与系统之间以及系统与系统之间的整合及调整的经营方法与解决方案的IT工具。 通常以Internet方式实现信息传递、数据同步、业务监控和企业业务流程的持续升级优化，从而实现跨应用、跨部门、跨合作伙伴与客户的企业运作。通过BPM软件对企业内部及外部的业务流程的整个生命周期进行建模、自动化、管理监控和优化，使企业成本降低，利润得以大幅提升。

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

### BPMN

BPMN（Business Process Model And Notation）- 业务流程模型和符号 是由BPMI（Business Process Management Initiative）开发的一套标准的业务流程建模符号，使用BPMN提供的符号可以创建业务流程。 2004年5月发布了BPMN1.0规范.BPMI于2005年9月并入OMG（The Object Management Group对象管理组织)组织。OMG于2011年1月发布BPMN2.0的最终版本。

具体发展历史如下:

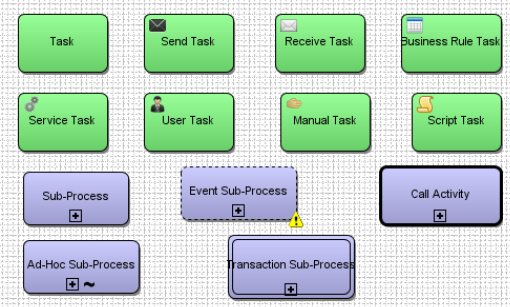


BPMN 是目前被各 BPM 厂商广泛接受的 BPM 标准。Activiti 就是使用 BPMN 2.0 进行流程建模、流程执行管理，它包括很多的建模符号，比如：

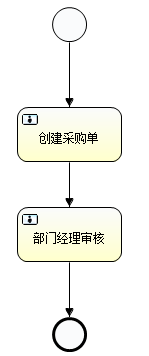
Event 用一个圆圈表示，它是流程中运行过程中发生的事情。



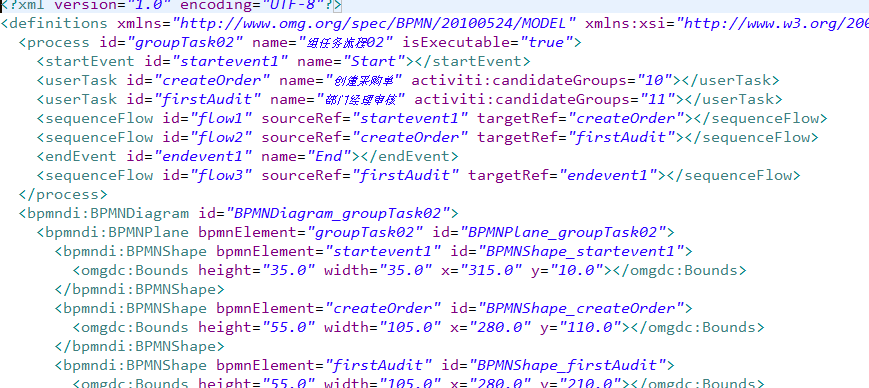
活动用圆角矩形表示，一个流程由一个活动或多个活动组成



一个bpmn图形的例子：



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



## Activit如何使用

1. **部署****activiti**

Activiti是一个工作流引擎（其实就是一堆jar包API），业务系统使用activiti来对系统的业务流程进行自动化管理，为了方便业务系统访问(操作)activiti的接口或功能，通常将activiti环境与业务系统的环境集成在一起。

1. **流程定义**

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

.bpmn文件就是业务流程定义文件，通过xml定义业务流程。

如果使用其它公司开发的工作作引擎一般都提供了可视化的建模工具(Process Designer)用于生成流程定义文件，建模工具操作直观，一般都支持图形化拖拽方式、多窗口的用户界面、丰富的过程图形元素、过程元素拷贝、粘贴、删除等功能。

1. **流程定义部署**

向activiti部署业务流程定义（.bpmn文件）。

使用activiti提供的api向activiti中部署.bpmn文件（一般情况还需要一块儿部署业务流程的图片.png）

1. **启动一个流程实例（ProcessInstance）**

启动一个流程实例表示开始一次业务流程的运行，比如员工请假流程部署完成，如果张三要请假就可以启动一个流程实例，如果李四要请假也启动一个流程实例，两个流程的执行互相不影响，就好比定义一个java类，实例化两个对象一样，部署的流程就好比java类，启动一个流程实例就好比new一个java对象。

1. **用户查询待办任务(Task)**

因为现在系统的业务流程已经交给activiti管理，通过activiti就可以查询当前流程执行到哪了，当前用户需要办理什么任务了，这些activiti帮我们管理了，而不像上边需要我们在sql语句中的where条件中指定当前查询的状态值是多少。

1. **用户办理任务**

用户查询待办任务后，就可以办理某个任务，如果这个任务办理完成还需要其它用户办理，比如采购单创建后由部门经理审核，这个过程也是由activiti帮我们完成了，不需要我们在代码中硬编码指定下一个任务办理人了。

1. **流程结束**

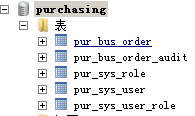
当任务办理完成没有下一个任务/结点了，这个流程实例就完成了。

# 环境准备

## 三个环境

第一个环境：没有加入工作流采购系统

作用：主要学习业务流程和springmvc+mybatis



第二个环境：activiti测试环境

作用：用于测试activiti的api，提供各种service接口。

需要创建一个数据库：

仅仅有activiti的数据表

第三个环境：activiti应用环境，加入工作流的采购系统

需要创建一个数据库：

包括activiti的数据表和业务表（采购系统的表）

## 开发环境

### Java环境

Jdk1.7



### 数据库

Mysql 5.1社区版

本教程采用5.1.33-community版本

### Web容器



### 开发工具

Mysql 客户端连接工具，Sqlyog或其它

文本编辑器EditPlus或其它

Java开发工具：限定为eclipse-indigo



**注意：activiti的流程定义工具插件目前只支持两个版本的eclipse：Indigo 和 Juno，本教程使用**indigo

## Activiti环境

Activiti5.14默认支持mybatis3.2.2（替换为mybatis3.2.3）

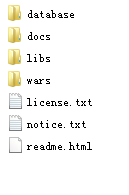
默认支持spring3.2.2（替换为3.1.4）

参考：\jar包\activiti核心jar包及依赖包

### 下载activiti5.14

Activiti下载地址：<http://activiti.org/download.html>

下载activiti5.14版本，解压：



1. **Database：**

activiti运行需要有数据库的支持，支持的数据库有：h2, mysql, oracle, postgres, mssql, db2等，该目录存放activiti的建表脚本。

1. **Docs**

Activiti的帮助文档。

1. **Libs**

Activiti所需要的jar 包

1. **Wars**

官方自带的示例工程。

### Activiti Designer流程设计器

本教程使用Activiti –Designer-eclipse-plugin（activiti流程设计器插件）完成流程的制作。

下面介绍了activiti designer设计器插件的安装方式，本教程使用的插件安装方式详细参考“activiti开发环境配置.docx”文档的“Eclipse插件安装”章节。

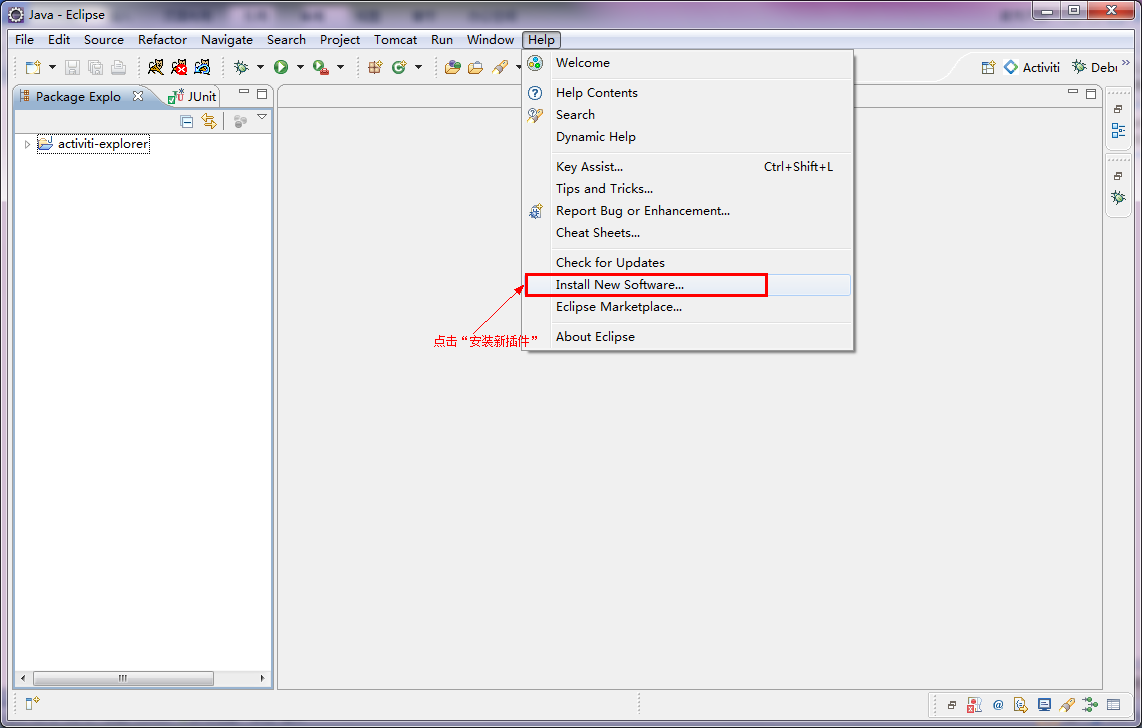
#### 插件安装方式1

参数文档开发工具目录下的“activiti开发环境配置.docx”中“eclipse插件安装”，其中包括了Activiti插件。

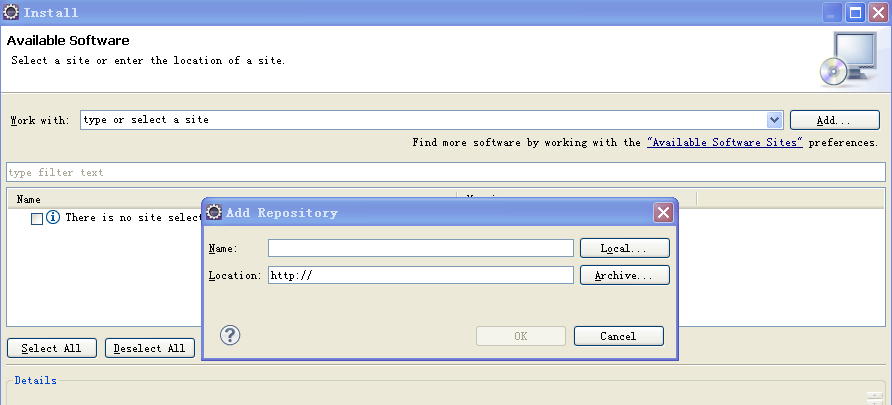
#### 插件安装方式2

**网络在线安装**：

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



1. 在如下Install界面板中，点击Add按钮：

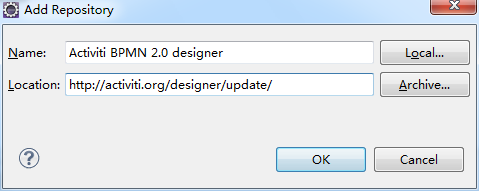


配置新装插件的地址和名称

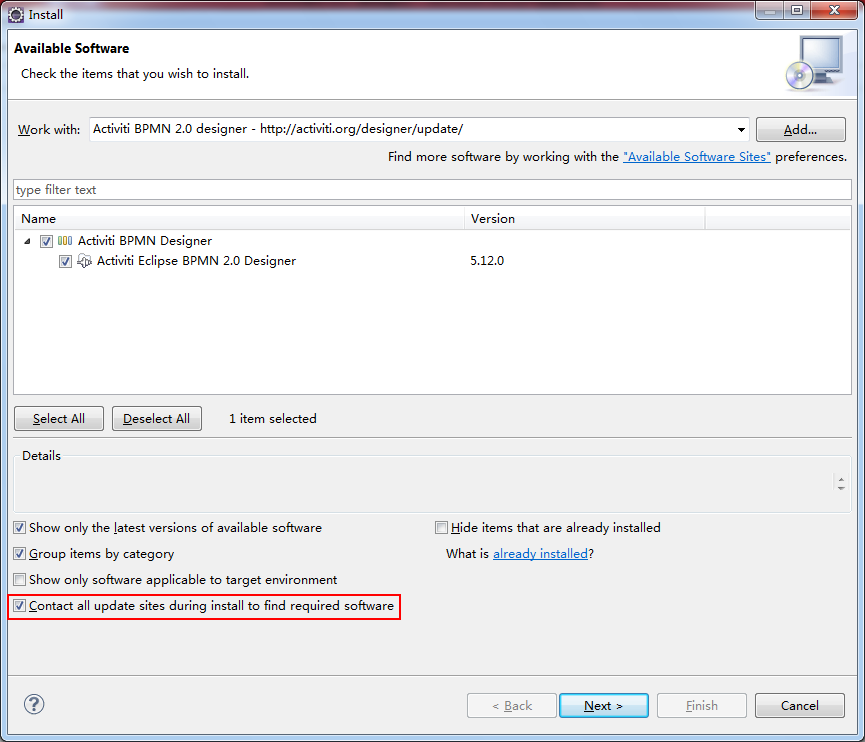
1. 然后填入下列字段

Name: Activiti BPMN 2.0 designer

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



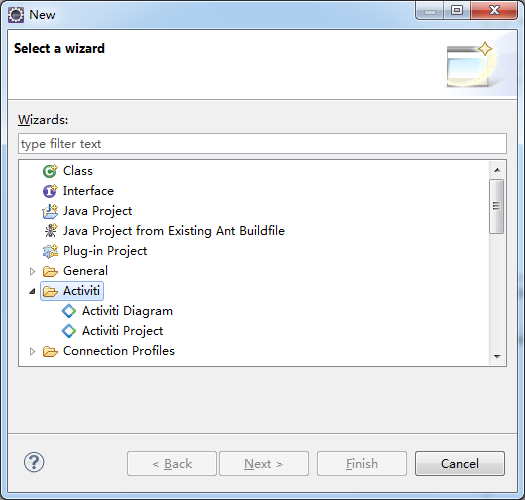
1. 回到Install界面，在面板正中列表中把所有展示出来的项目都勾上：



1. 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.

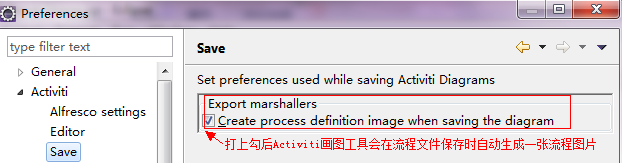
1. 安装完以后，点击新建工程new->Other…打开面板，如果看到下图内容：



说明安装成功了。

#### 补充说明

打开菜单Windows->Preferences->Activiti->Save下流程图片的生成方式:



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。

### Activiti支持的数据库

Activiti的运行需要数据库支撑，需要安装activiti数据库，支持如下版本：

| Activiti数据库类型 | 测试版本 | JDBC URL实例 | 备注 |
| --- | --- | --- | --- |
| h2 | 1.3.168 | jdbc:h2:tcp://localhost/activiti | 默认配置的数据库 |
| mysql | 5.1.21 | jdbc:mysql://localhost:3306/activiti?autoReconnect=true | 使用mysql-connector-java驱动测试 |
| oracle | 11.2.0.1.0 | jdbc:oracle:thin:@localhost:1521:xe |  |
| postgres | 8.1 | jdbc:postgresql://localhost:5432/activiti |  |
| db2 | DB2 10.1 using db2jcc4 | jdbc:db2://localhost:50000/activiti |  |
| mssql | 2008 using sqljdbc4 | jdbc:sqlserver://localhost:1433/activiti |  |

创建mysql数据库

本教程使用mysql数据库。

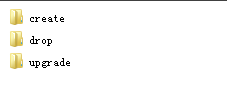
创建mysql数据库activiti（名字任意）：

CREATE DATABASE activiti DEFAULT CHARACTER SET utf8;

创建表方式1

可通过官方提供的数据库脚本创建表

进入activiti5.14下database目录：



Create目录存放了数据库创建脚本

Drop目录存放了数据库删除脚本

Upgrade目录存放了数据库升级脚本

手动执行Create目录的建库脚本即可完成数据库的创建。

**（本教程采用下边介绍的程序创建数据库方式）**

创建表方式2

通过运行java程序创建表。

创建java工程

使用eclipse创建java工程。

加入jar包

首先需要在java工程中加入ProcessEngine所需要的jar包，包括：

1. activiti-engine-5.14.jar
2. activiti依赖的jar包：mybatis、alf4j、log4j等
3. activiti依赖的spring包
4. 数据库驱动
5. 第三方数据连接池dbcp
6. 单元测试Junit-4.11.jar和hamcrest-core-1.3.jar

说明：



* activiti核心jar包及依赖包：

是activiti独立运行所需要的jar包。

比“spring+mybatis+activiti完整jar包”少了spring的aop、tx、jdbc等jar包。

* spring+mybatis+activiti完整jar包：

包括了spring所有的jar包及activiti的jar，实际使用activiti时要将activiti和业务系统整合运行，所以这里我们选用此包。

log4j.properties

# Global logging configuration

log4j.rootLogger=DEBUG, stdout

# Console output...

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n

activiti.cfg.xml

在classpath下创建activiti.cgf.xml文件：

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.1.xsd*

*http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.1.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-3.1.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-3.1.xsd "*>

</beans>

在activiti.cfg.xml中配置*数据源*和*processEngineConfiguration*

1. 数据源

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"* />

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/activiti"* />

<property name=*"username"* value=*"root"* />

<property name=*"password"* value=*"mysql"* />

<property name=*"maxActive"* value=*"3"* />

<property name=*"maxIdle"* value=*"1"* />

</bean>

1. processEngineConfiguration

processEngineConfiguration用来创建ProcessEngine，在创建ProcessEngine时会执行数据库的操作。

<bean id=*"processEngineConfiguration"*

class=*"org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration"*>

<!-- 数据源 -->

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

<!-- activiti数据库表处理策略 -->

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

</bean>

关于*processEngineConfiguration*中的**databaseSchemaUpdate参数，**通过此参数设计activiti数据表的处理策略，参数如下：

false（默认）：检查数据库表的版本和依赖库的版本， 如果版本不匹配就抛出异常。

true: 构建流程引擎时，执行检查，如果需要就执行更新。 如果表不存在，就创建。

create-drop: 构建流程引擎时创建数据库表， 关闭流程引擎时删除这些表。

drop-create：先删除表再创建表。

create: 构建流程引擎时创建数据库表， 关闭流程引擎时不删除这些表。

编写程序

创建ProcessEngineConfiguration，通过ProcessEngineConfiguration创建ProcessEngine，在创建ProcessEngine时会自动创建数据库。

//创建ProcessEngineConfiguration

ProcessEngineConfiguration configuration = ProcessEngineConfiguration

.*createProcessEngineConfigurationFromResource*("activiti.cfg.xml")

//通过ProcessEngineConfiguration创建ProcessEngine，此时会创建数据库

ProcessEngine processEngine = configuration.buildProcessEngine();

System.*out*.println(processEngine);

说明：

1、运行以上程序段即可完成activiti数据库创建，通过改变activiti.cfg.xml中

databaseSchemaUpdate参数的值执行不同的数据表处理策略。

2、上边的方法*createProcessEngineConfigurationFromResource在执行时在activiti.cfg.xml中找固定的名称**processEngineConfiguration*

也可以使用重载方法调用，这时可以不用限定*processEngineConfiguration*名称

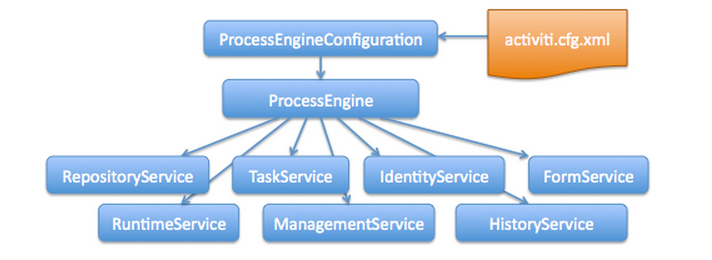


### 数据库表的命名规则

Activiti的表都以**ACT\_**开头。 第二部分是表示表的用途的两个字母标识。 用途也和服务的API对应。

* **ACT\_RE\_\***: 'RE'表示repository。 这个前缀的表包含了流程定义和流程静态资源 （图片，规则，等等）。
* **ACT\_RU\_\***: 'RU'表示runtime。 这些运行时的表，包含流程实例，任务，变量，异步任务，等运行中的数据。 Activiti只在流程实例执行过程中保存这些数据， 在流程结束时就会删除这些记录。 这样运行时表可以一直很小速度很快。
* **ACT\_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\_\***: GE表示general。通用数据， 用于不同场景下。

Activiti服务架构图



activiti.cfg.xml

activiti的引擎配置文件，包括：ProcessEngineConfiguration的定义、数据源定义、事务管理器等，此文件其实就是一个spring配置文件，下面是一个基本的配置只配置了ProcessEngineConfiguration和数据源：

<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:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.1.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.1.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-3.1.xsd"*>

<!--数据库连接池 -->

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"* />

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/activiti"* />

<property name=*"username"* value=*"root"* />

<property name=*"password"* value=*"mysql"* />

</bean>

<bean id=*"processEngineConfiguration"*

class=*"org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration"*>

<!-- 数据源 -->

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

<!-- 数据库策略 -->

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

</bean>

</beans>

ProcessEngineConfiguration：

流程引擎的配置类，通过ProcessEngineConfiguration可以创建工作流引擎ProceccEngine，常用的两种方法如下：

**StandaloneProcessEngineConfiguration**

**通过org.activiti.engine.impl.cfg.****StandaloneProcessEngineConfiguration**

Activiti可以单独运行，使用它创建的ProcessEngine，Activiti会自己处理事务。

配置文件方式：

通常在activiti.cfg.xml配置文件中定义一个id为 processEngineConfiguration 的bean，这里会使用spring的依赖注入来构建引擎。

方法如下：

<bean id=*"processEngineConfiguration"*

class=*"org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration"*>

<!-- 数据源 -->

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

<!-- 数据库策略 -->

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

</bean>

**SpringProcessEngineConfiguration**

通过**org.activiti.spring.****SpringProcessEngineConfiguration 与**Spring整合。

创建spring与activiti的整合配置文件：

activity-spring.cfg.xml（名称不固定）

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.1.xsd*

*http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.1.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-3.1.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-3.1.xsd "*>

<!-- 工作流引擎配置bean -->

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

<!-- 数据源 -->

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

<!-- 使用spring事务管理器 -->

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

<!-- 数据库策略 -->

<property name=*"databaseSchemaUpdate"* value=*"drop-create"* />

<!-- activiti的定时任务关闭 -->

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

</bean>

<!-- 流程引擎 -->

<bean id=*"processEngine"* class=*"org.activiti.spring.ProcessEngineFactoryBean"*>

<property name=*"processEngineConfiguration"* ref=*"processEngineConfiguration"* />

</bean>

<!-- 资源服务service -->

<bean id=*"repositoryService"* factory-bean=*"processEngine"*

factory-method=*"getRepositoryService"* />

<!-- 流程运行service -->

<bean id=*"runtimeService"* factory-bean=*"processEngine"*

factory-method=*"getRuntimeService"* />

<!-- 任务管理service -->

<bean id=*"taskService"* factory-bean=*"processEngine"*

factory-method=*"getTaskService"* />

<!-- 历史管理service -->

<bean id=*"historyService"* factory-bean=*"processEngine"*

factory-method=*"getHistoryService"* />

<!-- 用户管理service -->

<bean id=*"identityService"* factory-bean=*"processEngine"*

factory-method=*"getIdentityService"* />

<!-- 引擎管理service -->

<bean id=*"managementService"* factory-bean=*"processEngine"*

factory-method=*"getManagementService"* />

<!-- 数据源 -->

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"* />

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/activiti"* />

<property name=*"username"* value=*"root"* />

<property name=*"password"* value=*"mysql"* />

<property name=*"maxActive"* value=*"3"* />

<property name=*"maxIdle"* value=*"1"* />

</bean>

<!-- 事务管理器 -->

<bean id=*"transactionManager"*

class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

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

</bean>

<!-- 通知 -->

<tx:advice id=*"txAdvice"* transaction-manager=*"transactionManager"*>

<tx:attributes>

<!-- 传播行为 -->

<tx:method name=*"save\*"* propagation=*"REQUIRED"* />

<tx:method name=*"insert\*"* propagation=*"REQUIRED"* />

<tx:method name=*"delete\*"* propagation=*"REQUIRED"* />

<tx:method name=*"update\*"* propagation=*"REQUIRED"* />

<tx:method name=*"find\*"* propagation=*"SUPPORTS"* read-only=*"true"* />

<tx:method name=*"get\*"* propagation=*"SUPPORTS"* read-only=*"true"* />

</tx:attributes>

</tx:advice>

<!-- 切面，根据具体项目修改切点配置 -->

<aop:config proxy-target-class=*"true"*>

<aop:advisor advice-ref=*"txAdvice"*

pointcut=*"execution(\* cn.itcast.purchasing.service.impl.\*.\*(..))"* />

</aop:config>

</beans>

**创建processEngineConfiguration**

ProcessEngineConfiguration configuration = ProcessEngineConfiguration

.*createProcessEngineConfigurationFromResource*("activiti.cfg.xml")

上边的代码要求activiti.cfg.xml中必须有一个processEngineConfiguration的bean

也可以使用下边的方法，更改bean 的名字：

ProcessEngineConfiguration.createProcessEngineConfigurationFromResource(String resource, String beanName);

ProcessEngine：

工作流引擎，相当于一个门面接口，通过ProcessEngineConfiguration创建processEngine，通过ProcessEngine创建各各service接口。

一般创建方式：

//通过ProcessEngineConfiguration创建ProcessEngine

ProcessEngine processEngine =

processEngineConfiguration.buildProcessEngine();

简单创建方式

将activiti.cfg.xml文件名及路径固定，且activiti.cfg.xml文件中有 processEngineConfiguration的配置， 可以使用如下代码创建processEngine:

//使用classpath下的activiti.cfg.xml中的配置创建processEngine

ProcessEngine processEngine = ProcessEngines.*getDefaultProcessEngine*();

System.*out*.println(processEngine);

Service ：

Service创建方式

通过ProcessEngine创建Service，Service是工作流引擎提供用于进行工作流部署、执行、管理的服务接口。

方式如下：

**RuntimeService runtimeService = processEngine.getRuntimeService();**

**RepositoryService repositoryService = processEngine.getRepositoryService();**

**TaskService taskService = processEngine.getTaskService();**

**……**

Service总览

|  |  |
| --- | --- |
| RepositoryService | activiti的资源管理类 |
| RuntimeService | activiti的流程运行管理类 |
| TaskService | activiti的任务管理类 |
| HistoryService | activiti的历史管理类 |
| IdentityService | activiti的用户身份管理类 |
| FormService | activiti的表单管理类 |
| ManagerService | activiti的引擎管理类 |

注：红色标注为常用service。

****RepositoryService****

是activiti的资源管理类，提供了管理和控制流程发布包和流程定义的操作。使用工作流建模工具设计的业务流程图需要使用此service将流程定义文件的内容部署到计算机。

除了部署流程定义以外还可以：

查询引擎中的发布包和流程定义。

暂停或激活发布包，对应全部和特定流程定义。 暂停意味着它们不能再执行任何操作了，激活是对应的反向操作。

获得多种资源，像是包含在发布包里的文件， 或引擎自动生成的流程图。

获得流程定义的pojo版本， 可以用来通过java解析流程，而不必通过xml。

****RuntimeService****

它是activiti的流程运行管理类。可以从这个服务类中获取很多关于流程执行相关的信息

TaskService

是activiti的任务管理类。可以从这个类中获取任务的信息。

HistoryService

是activiti的历史管理类，可以查询历史信息，执行流程时，引擎会保存很多数据（根据配置），比如流程实例启动时间，任务的参与者， 完成任务的时间，每个流程实例的执行路径，等等。 这个服务主要通过查询功能来获得这些数据。

IdentityService

是activiti的用户管理类，它可以管理（创建，更新，删除，查询...）群组和用户。

FormService

是activiti的表单管理类，一个可选服务。即使不使用它，Activiti也可以完美运行， 不会损失任何功能。这个服务提供了启动表单和任务表单两个概念。 启动表单会在流程实例启动之前展示给用户， 任务表单会在用户完成任务时展示

ManagementService

是activiti的引擎管理类，提供了对 Activiti 流程引擎的管理和维护功能，这些功能不在工作流驱动的应用程序中使用，主要用于 Activiti 系统的日常维护。

Activiti入门体验

流程定义

### Activiti-Designer使用

Palette（画板）

在eclipse中安装activiti-designer插件即可在eclipse中使用，画板中包括以下结点：

Connection—连接

Event---事件

Task---任务

Gateway---网关

Container—容器

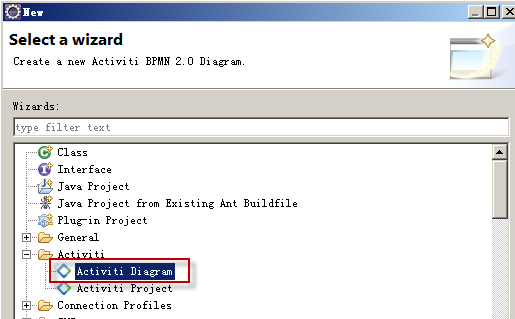
Boundary event—边界事件

Intermediate event- -中间事件

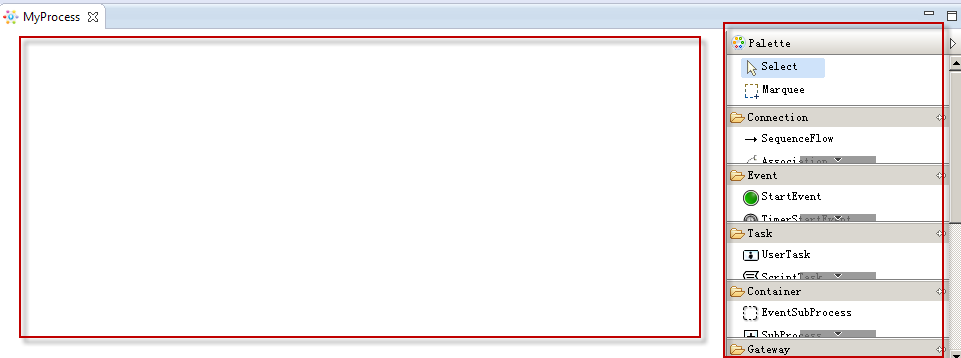
流程图设计完毕保存生成.bpmn文件。

新建流程

首先选中存放图形的目录，点击菜单：File—》new—》Other



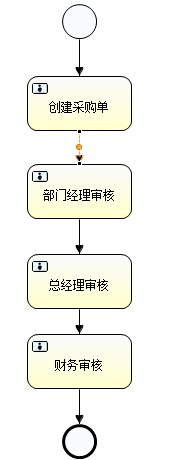
创建成功：



左侧区域是绘图区，右侧区域是palette画板区域

鼠标先点击画板的元素即可在左侧绘图。

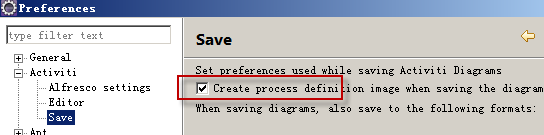
### 绘制流程



图形绘制好后会生成两个文件：



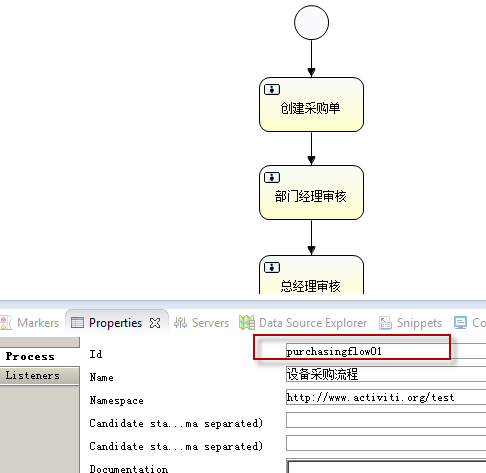
自动生成图形，需要设置eclipse:



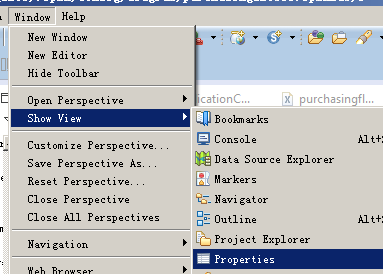
### 指定流程定义key

流程定义key即流程定义的标识，在eclipse中通过properties视图查看流程的key

建议：相同的业务流程，流程定义的key名字定义一样，比如采购流程定义为purchasingflow，后期如果再修改采购流程，流程定义key仍然为purchasingflow，如果需要创建新的业务流程，比如请假流程则使用新的key。



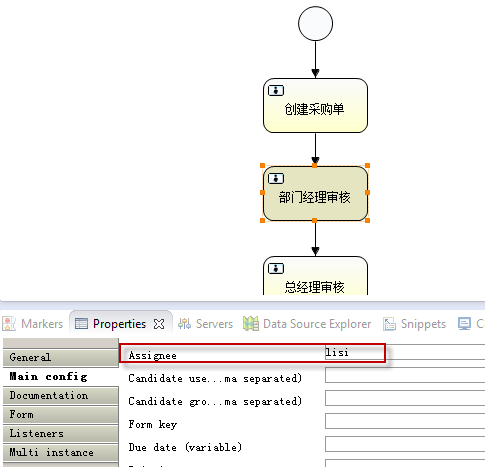
如果properties视图没有显示需要在eclipse中打开：



### 指定任务负责人

在properties视图指定每个任务结点的负责人，

比如下边是指定部门经理审核的负责人为lisi



## 部署流程定义

部署流程定义就是要将上边绘制的图形即流程定义（.bpmn）部署在工作流程引擎activiti中，方法如下：

使用ProcessEngine创建RepositoryService，代码如下：

// 获取repositoryService

RepositoryService repositoryService = processEngine

.getRepositoryService();

//部署对象

Deployment deployment = repositoryService.createDeployment()

.addClasspathResource("diagram/test/purchasingflow01.bpmn")// bpmn文件

.addClasspathResource("diagram/test/purchasingflow01.png")// 图片文件

.name("企业采购流程")

.deploy();

System.*out*.println("流程部署id:" + deployment.getId());

System.*out*.println("流程部署名称:" + deployment.getName());

执行此操作后activiti会将上边代码中指定的bpm文件和图片文件保存在activiti数据库。

## 启动一个流程实例

流程定义部署在activiti后就可以通过工作流管理业务流程了，也就是说上边部署的采购流程可以使用了。

针对该采购流程，启动一个流程表示发起一个新的采购单，这就相当于java类与java对象的关系，类定义好后需要new创建一个对象使用，当然可以new多个对象。对于采购流程，张三发起一个采购单需要启动一个流程实例，李四发起一个采购单也需要启动一个流程实例。

代码如下：

// 启动一个流程实例

@Test

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

// 获取RunTimeService

RuntimeService runtimeService = processEngine.getRuntimeService();

// 根据流程定义key启动流程

ProcessInstance processInstance = runtimeService

.startProcessInstanceByKey("purchasingflow01");

System.*out*.println("流程定义id：" + processInstance.getProcessDefinitionId());

System.*out*.println("流程实例id：" + processInstance.getId());

System.*out*.println("当前活动Id：" + processInstance.getActivityId());

}

## 任务查询

流程启动后，各各任务的负责人就可以查询自己当前需要处理的任务，查询出来的任务都是该用户的待办任务。

// 查询当前个人待执行的任务

@Test

**public** **void** findPersonalTaskList() {

// 任务负责人

String assignee = "zhangsan";

// 创建TaskService

TaskService taskService = processEngine.getTaskService();

List<Task> list = taskService.createTaskQuery()//

.processDefinitionKey("purchasingflow01")//

.taskAssignee(assignee)//只查询该任务负责人的任务

.list();

**for** (Task task : list) {

System.*out*.println("流程实例id：" + task.getProcessInstanceId());

System.*out*.println("任务id：" + task.getId());

System.*out*.println("任务负责人：" + task.getAssignee());

System.*out*.println("任务名称：" + task.getName());

}

}

## 任务处理

任务负责人查询待办任务，选择任务进行处理，完成任务。

// 完成任务

@Test

**public** **void** completTask() {

//任务id

String taskId = "8305";

// 创建TaskService

TaskService taskService = processEngine.getTaskService();

//完成任务

taskService.complete(taskId);

System.*out*.println("完成任务id="+taskId);

}

# 流程定义

## 流程定义

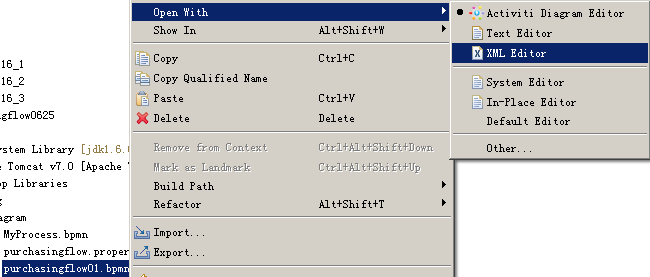
使用activiti-explorer（web控制台）或activiti-eclipse-designer插件对业务流程进行建模，这两种方式都遵循bpmn2.0标准。本教程使用activiti-eclipse-designer插件完成流程建模。

使用designer设计器绘制流程，会生成两个文件：.bpmn和.png

### .bpmn文件

使用activiti-desinger设计业务流程，会生成.bpmn文件，

选择XML editor查询方式：



.bpmn内容如下：

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<definitions xmlns=*"http://www.omg.org/spec/BPMN/20100524/MODEL"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:activiti=*"http://activiti.org/bpmn"*

xmlns:bpmndi=*"http://www.omg.org/spec/BPMN/20100524/DI"* xmlns:omgdc=*"http://www.omg.org/spec/DD/20100524/DC"*

xmlns:omgdi=*"http://www.omg.org/spec/DD/20100524/DI"* typeLanguage=*"http://www.w3.org/2001/XMLSchema"*

expressionLanguage=*"http://www.w3.org/1999/XPath"* targetNamespace=*"http://www.activiti.org/test"*>

<!-- 流程定义部分 -->

<process id=*"myProcess"* name=*"My process"* isExecutable=*"true"*>

</process>

<!-- 流程布局定义 -->

<bpmndi:BPMNDiagram id=*"BPMNDiagram\_myProcess"*>

<bpmndi:BPMNPlane bpmnElement=*"myProcess"* id=*"BPMNPlane\_myProcess"*>

</bpmndi:BPMNPlane>

</bpmndi:BPMNDiagram>

</definitions>

BPMN 2.0根节点是definitions节点。 这个元素中，可以定义多个流程定义（不过我们建议每个文件只包含一个流程定义， 可以简化开发过程中的维护难度）。 注意，definitions元素 最少也要包含xmlns 和 targetNamespace的声明。 targetNamespace可以是任意值，它用来对流程实例进行分类。

流程定义部分：定义了流程每个结点的描述及结点之间的流程流转。

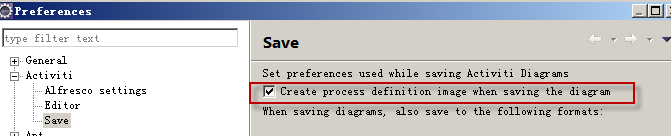
流程布局定义：定义流程每个结点在流程图上的位置坐标等信息。

### .png图片文件

流程图片生成的两种方式：

* 使用activiti-designer设计流程图时自动生成

需在eclipse中进行配置：



使用designer设计流程图的同时自动生成与bpmn文件同名的图片文件(.png)

* 由activiti自动生成图形

流程图设计完毕向activiti中部署流程定义文件bpmn，部署时由activiti自动生成流程图片。(流程部署在下面的章节讲解)

注意：此方法生成时如果图形中有中文生成的图片上显示乱码，且.bpmn中的坐标和图片显示错位。

## 流程定义部署

### 单个文件部署方式

分别将bpmn文件和png图片文件部署。

@Test

**public** **void** deployProcess() {

// 获取repositoryService

RepositoryService repositoryService = processEngine

.getRepositoryService();

// bpmn输入流

InputStream inputStream\_bpmn = **this**

.getClass()

.getClassLoader()

.getResourceAsStream(

"cn/itcast/activiti/first/diagram/purchasing/purchasingflow01.bpmn");

// 图片输入流

InputStream inputStream\_png = **this**

.getClass()

.getClassLoader()

.getResourceAsStream(

"cn/itcast/activiti/first/diagram/purchasing/purchasingflow01.png");

// 流程部署对象

Deployment deployment = repositoryService.createDeployment()

.addInputStream("purchasingflow01.bpmn", inputStream\_bpmn)

.addInputStream("purchasingflow01.png", inputStream\_png)

.deploy();

System.*out*.println("流程部署id：" + deployment.getId());

System.*out*.println("流程部署名称：" + deployment.getName());

}

执行此操作后activiti会将上边代码中指定的bpm文件和图片文件保存在activiti数据库。

### 压缩包部署方式

将purchasingflow01.bpmn和purchasingflow01.png压缩成zip包。

@Test

**public** **void** deployProcessByZip() {

// 定义zip输入流

InputStream inputStream = **this**

.getClass()

.getClassLoader()

.getResourceAsStream(

"cn/itcast/activiti/first/diagram/purchasing/purchasingflow01.zip");

ZipInputStream zipInputStream = **new** ZipInputStream(inputStream);

// 获取repositoryService

RepositoryService repositoryService = processEngine

.getRepositoryService();

// 流程部署

Deployment deployment = repositoryService.createDeployment()//

.addZipInputStream(zipInputStream)//

.deploy();

System.*out*.println("流程部署id：" + deployment.getId());

System.*out*.println("流程部署名称：" + deployment.getName());

}

执行此操作后activiti会将上边代码中指定的bpm文件和图片文件保存在activiti数据库。

### 操作数据表

流程定义部署后操作activiti数据表如下：

SELECT \* FROM act\_re\_deployment #流程定义部署表，记录流程部署信息

SELECT \* FROM act\_re\_procdef #流程定义表，记录流程定义信息

SELECT \* FROM act\_ge\_bytearray #资源表

说明：

act\_re\_deployment和act\_re\_procdef一对多关系，一次部署在流程部署表生成一条记录，但一次部署可以部署多个流程定义，每个流程定义在流程定义表生成一条记录。每一个流程定义在act\_ge\_bytearray会存在两个资源记录，bpmn和png。

建议：一次部署一个流程，这样部署表和流程定义表是一对一有关系，方便读取流程部署及流程定义信息。

## 流程定义查询

查询部署的流程定义。

// 流程定义查询

@Test

**public** **void** queryProceccDefinition() {

// 流程定义key

String processDefinitionKey = "purchasingflow01";

// 获取repositoryService

RepositoryService repositoryService = processEngine

.getRepositoryService();

// 查询流程定义

ProcessDefinitionQuery processDefinitionQuery = repositoryService

.createProcessDefinitionQuery();

//遍历查询结果

List<ProcessDefinition> list = processDefinitionQuery

.processDefinitionKey(processDefinitionKey)

.orderByProcessDefinitionVersion().desc().list();

**for** (ProcessDefinition processDefinition : list) {

System.*out*.println("------------------------");

System.*out*.println("流程部署id：" + processDefinition.getDeploymentId());

System.*out*.println("流程定义id：" + processDefinition.getId());

System.*out*.println("流程定义名称：" + processDefinition.getName());

System.*out*.println("流程定义key：" + processDefinition.getKey());

System.*out*.println("流程定义版本：" + processDefinition.getVersion());

}

}

## 流程定义删除

删除已经部署成功的流程定义。

**public** **void** deleteDeployment() {

// 流程部署id

String deploymentId = "8801";

// 通过流程引擎获取repositoryService

RepositoryService repositoryService = processEngine

.getRepositoryService();

//删除流程定义，如果该流程定义已有流程实例启动则删除时出错

repositoryService.deleteDeployment(deploymentId);

//设置true 级联删除流程定义，即使该流程有流程实例启动也可以删除，设置为false非级别删除方式，如果流程

//repositoryService.deleteDeployment(deploymentId, true);

}

说明：

1. 使用repositoryService删除流程定义
2. 如果该流程定义下没有正在运行的流程，则可以用普通删除。
3. 如果该流程定义下存在已经运行的流程，使用普通删除报错，可用级联删除方法将流程及相关记录全部删除。项目开发中使用级联删除的情况比较多，删除操作一般只开放给超级管理员使用。

## 流程定义资源查询

### 方式1

通过流程定义对象获取流程定义资源，获取bpmn和png。

@Test

**public** **void** getProcessResources() **throws** IOException {

// 流程定义id

String processDefinitionId = "";

// 获取repositoryService

RepositoryService repositoryService = processEngine

.getRepositoryService();

// 流程定义对象

ProcessDefinition processDefinition = repositoryService

.createProcessDefinitionQuery()

.processDefinitionId(processDefinitionId).singleResult();

//获取bpmn

String resource\_bpmn = processDefinition.getResourceName();

//获取png

String resource\_png = processDefinition.getDiagramResourceName();

// 资源信息

System.*out*.println("bpmn：" + resource\_bpmn);

System.*out*.println("png：" + resource\_png);

File file\_png = **new** File("d:/purchasingflow01.png");

File file\_bpmn = **new** File("d:/purchasingflow01.bpmn");

// 输出bpmn

InputStream resourceAsStream = **null**;

resourceAsStream = repositoryService.getResourceAsStream(

processDefinition.getDeploymentId(), resource\_bpmn);

FileOutputStream fileOutputStream = **new** FileOutputStream(file\_bpmn);

**byte**[] b = **new** **byte**[1024];

**int** len = -1;

**while** ((len = resourceAsStream.read(b, 0, 1024)) != -1) {

fileOutputStream.write(b, 0, len);

}

// 输出图片

resourceAsStream = repositoryService.getResourceAsStream(

processDefinition.getDeploymentId(), resource\_png);

fileOutputStream = **new** FileOutputStream(file\_png);

// byte[] b = new byte[1024];

// int len = -1;

**while** ((len = resourceAsStream.read(b, 0, 1024)) != -1) {

fileOutputStream.write(b, 0, len);

}

}

### 方式2

通过查询流程部署信息获取流程定义资源。

// 获取流程定义图片资源

@Test

**public** **void** getProcessResources() **throws** IOException {

//流程部署id

String deploymentId = "9001";

// 通过流程引擎获取repositoryService

RepositoryService repositoryService = processEngine

.getRepositoryService();

//读取资源名称

List<String> resources = repositoryService.getDeploymentResourceNames(deploymentId);

String resource\_image = **null**;

//获取图片

**for**(String resource\_name :resources){

**if**(resource\_name.indexOf(".png")>=0){

resource\_image = resource\_name;

}

}

//图片输入流

InputStream inputStream = repositoryService.getResourceAsStream(deploymentId, resource\_image);

File exportFile = **new** File("d:/purchasingflow.png");

FileOutputStream fileOutputStream = **new** FileOutputStream(exportFile);

**byte**[] buffer = **new** **byte**[1024];

**int** len = -1;

//输出图片

**while**((len = inputStream.read(buffer))!=-1){

fileOutputStream.write(buffer, 0, len);

}

inputStream.close();

fileOutputStream.close();

}

说明：

1. deploymentId为流程部署ID
2. resource\_name为act\_ge\_bytearray表中NAME\_列的值
3. 使用repositoryService的getDeploymentResourceNames方法可以获取指定部署下得所有文件的名称
4. 使用repositoryService的getResourceAsStream方法传入部署ID和资源图片名称可以获取部署下指定名称文件的输入流
5. 最后的将输入流中的图片资源进行输出。

# 应用

## 采购系统环境搭建

Activiti要管理业务系统的业务流程少不了业务系统，本教程采用一个企业采购业务流程作为activiti实际应用的案例场景。

**请参考原始没有添加工作流程的工程“purchasing”，它采用springmvc+mybatis开发。**

下面要创建一个新的web工程，将原始工程的内容拷入，下边要在此工程中集成activiti，最终实现activiti来管理业务流程。

下面步骤描述了工程的配置步骤：

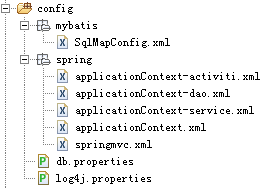
### Jar包

使用下边的jar替换原始工程的jar包



### 配置文件

红色标注是需要添加的文件。



SqlMapConfig.xml：mybatis的配置文件(原工程)

Springmvc.xml：springmvc框架的配置文件(原工程)

applicationContext.xml：spring容器配置文件，这里配置了数据源和事务管理(原工程)

applicationContext-dao.xml：配置了mybatis的自动扫描器(原工程)

applicationContext-service.xml： 配置了服务接口(原工程)

applicationContext-activiti.xml：是spring与actiivti整合的配置文件（参考上面章节的spring-activiti.cfg.xml）

applicationContext-activiti.xml

activiti与spring整合配置文件。

<!-- 工作流引擎配置bean -->

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

<!-- 数据源 -->

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

<!-- 使用spring事务管理器 Spring对业务系统的service和activiti的service使用一个事务管理器 -->

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

<!-- 数据库策略 -->

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

<!-- activiti的定时任务关闭 -->

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

</bean>

<!-- 流程引擎 -->

<bean id=*"processEngine"* class=*"org.activiti.spring.ProcessEngineFactoryBean"*>

<property name=*"processEngineConfiguration"* ref=*"processEngineConfiguration"* />

</bean>

<!-- 资源服务service -->

<bean id=*"repositoryService"* factory-bean=*"processEngine"*

factory-method=*"getRepositoryService"* />

<!-- 流程运行service -->

<bean id=*"runtimeService"* factory-bean=*"processEngine"*

factory-method=*"getRuntimeService"* />

<!-- 任务管理service -->

<bean id=*"taskService"* factory-bean=*"processEngine"*

factory-method=*"getTaskService"* />

<!-- 历史管理service -->

<bean id=*"historyService"* factory-bean=*"processEngine"*

factory-method=*"getHistoryService"* />

<!-- 用户管理service -->

<bean id=*"identityService"* factory-bean=*"processEngine"*

factory-method=*"getIdentityService"* />

<!-- 引擎管理service -->

<bean id=*"managementService"* factory-bean=*"processEngine"*

factory-method=*"getManagementService"* />

db.properties

数据库连接参数配置

jdbc.driver=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/mysql数据库名

jdbc.username=用户名

jdbc.password=密码

jdbc.maxActive=5

jdbc.maxIdle=2

### web.xml中加载spring容器

<!-- 加载spring容器 -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/classes/spring/applicationContext.xml,/WEB-INF/classes/spring/applicationContext-\*.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

### web.xml中配置Springmvc的前端控制器

<!-- springmvc的前端控制器 -->

<servlet>

<servlet-name>springmvc</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<!-- contextConfigLocation不是必须的， 如果不配置contextConfigLocation， springmvc的配置文件默认在：WEB-INF/servlet的name+"-servlet.xml" -->

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:spring/springmvc.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>springmvc</servlet-name>

<url-pattern>\*.action</url-pattern>

</servlet-mapping>

### web.xml中Post提交乱码过虑器

<!-- 解决post乱码 -->

<filter>

<filter-name>CharacterEncodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>utf-8</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>CharacterEncodingFilter</filter-name>

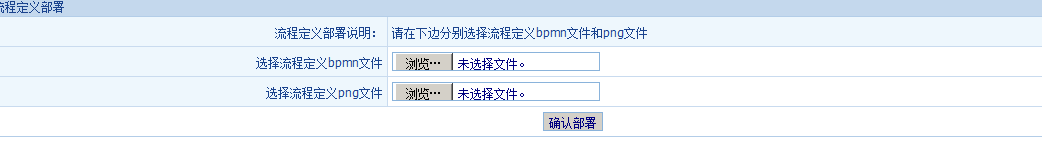
<url-pattern>/\*</url-pattern>

</filter-mapping>

功能开发

在采购系统中增加流程定义功能，功能如下：

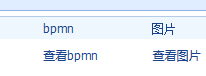
流程定义部署



流程定义查询



流程定义资源文件查询



流程部署删除



# 流程实例

## 启动流程实例

流程定义部署在activiti后，就可以在系统中通过activiti去管理该流程的执行，执行流程表示流程的一次执行。比如部署系统采购流程后，如果某用户要申请采购这时就需要执行这个流程，如果另外一个用户也要申请采购则也需要执行该流程，每个执行互不影响，每个执行是单独的流程实例。执行流程首先需要启动流程实例。

@Test

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

// 流程定义key

String processDefinitionKey = "";

// 获取RunTimeService

RuntimeService runtimeService = processEngine.getRuntimeService();

// 根据流程定义key启动流程

ProcessInstance processInstance = runtimeService

.startProcessInstanceByKey(processDefinitionKey);

System.*out*

.println("流程定义id：" + processInstance.getProcessDefinitionId());

System.*out*.println("流程实例id：" + processInstance.getId());

System.*out*.println("当前活动Id：" + processInstance.getActivityId());

}

### 操作数据库表

启动流程实例，操作如下数据库表：

SELECT \* FROM act\_ru\_execution #流程实例执行表，记录当前流程实例的执行情况

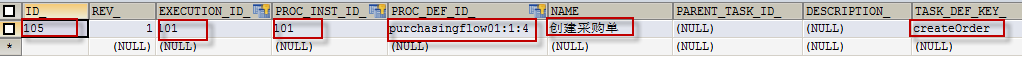


说明：

流程实例执行，如果当前只有一个分支时，一个流程实例只有一条记录且执行表的主键id和流程实例id相同，如果当前有多个分支正在运行则该执行表中有多条记录，存在执行表的主键和流程实例id不相同的记录。**不论当前有几个分支总会有一条记录的执行表的主键和流程实例id相同**

一个流程实例运行完成，此表中与流程实例相关的记录删除。

SELECT \* FROM act\_ru\_task #任务执行表，记录当前执行的任务

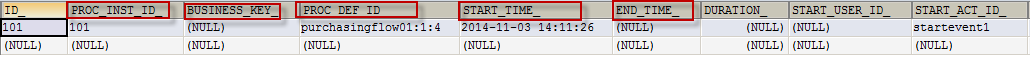


说明：启动流程实例，流程当前执行到第一个任务结点，此表会插入一条记录表示当前任务的执行情况，如果任务完成则记录删除。

SELECT \* FROM act\_ru\_identitylink #任务参与者，记录当前参与任务的用户或组

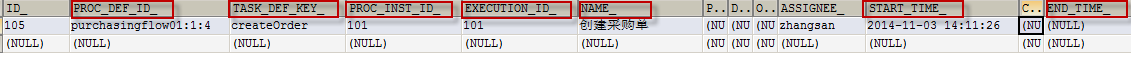


SELECT \* FROM act\_hi\_procinst #流程实例历史表



流程实例启动，会在此表插入一条记录，流程实例运行完成记录也不会删除。

SELECT \* FROM act\_hi\_taskinst #任务历史表，记录所有任务



开始一个任务，不仅在act\_ru\_task表插入记录，也会在历史任务表插入一条记录，任务历史表的主键就是任务id，任务完成此表记录不删除。

SELECT \* FROM act\_hi\_actinst #活动历史表，记录所有活动



活动包括任务，所以此表中不仅记录了任务，还记录了流程执行过程的其它活动，比如开始事件、结束事件。

## 查询流程实例

流程在运行过程中可以查询流程实例的状态，当前运行结点等信息。

@Test

**public** **void** queryProcessInstance() {

// 流程定义key

String processDefinitionKey = "purchasingflow01";

// 获取RunTimeService

RuntimeService runtimeService = processEngine.getRuntimeService();

List<ProcessInstance> list = runtimeService

.createProcessInstanceQuery()

.processDefinitionKey(processDefinitionKey)//

.list();

**for** (ProcessInstance processInstance : list) {

System.*out*.println("----------------------------");

System.*out*.println("流程实例id："

+ processInstance.getProcessInstanceId());

System.*out*.println("所属流程定义id："

+ processInstance.getProcessDefinitionId());

System.*out*.println("是否执行完成：" + processInstance.isEnded());

System.*out*.println("是否暂停：" + processInstance.isSuspended());

System.*out*.println("当前活动标识：" + processInstance.getActivityId());

}

}

## 挂起、激活流程实例(了解)

某些情况可能由于流程变更需要将当前运行的流程暂停而不是直接删除，流程暂停后将不会继续执行。

### 全部流程实例挂起

操作流程定义为挂起状态，该流程定义下边所有的流程实例全部暂停：

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

// 挂起激活流程定义

@Test

**public** **void** suspendOrActivateProcessDefinition() {

// 流程定义id

String processDefinitionId = "";

RepositoryService repositoryService = processEngine

.getRepositoryService();

// 获得流程定义

ProcessDefinition processDefinition = repositoryService

.createProcessDefinitionQuery()

.processDefinitionId(processDefinitionId).singleResult();

//是否暂停

**boolean** suspend = processDefinition.isSuspended();

**if**(suspend){

//如果暂停则激活，这里将流程定义下的所有流程实例全部激活

repositoryService.activateProcessDefinitionById(processDefinitionId, **true**, **null**);

System.*out*.println("流程定义："+processDefinitionId+"激活");

}**else**{

//如果激活则挂起，这里将流程定义下的所有流程实例全部挂起

repositoryService.suspendProcessDefinitionById(processDefinitionId, **true**, **null**);

System.*out*.println("流程定义："+processDefinitionId+"挂起");

}

}

### 单个流程实例挂起

操作流程实例对象，针对单个流程执行挂起操作，某个流程实例挂起则此流程不再继续执行，完成该流程实例的当前任务将报异常。

@Test

**public** **void** suspendOrActiveProcessInstance() {

// 流程实例id

String processInstanceId = "";

// 获取RunTimeService

RuntimeService runtimeService = processEngine.getRuntimeService();

//根据流程实例id查询流程实例

ProcessInstance processInstance = runtimeService.createProcessInstanceQuery()

.processInstanceId(processInstanceId).singleResult();

**boolean** suspend = processInstance.isSuspended();

**if**(suspend){

//如果暂停则激活

runtimeService.activateProcessInstanceById(processInstanceId);

System.*out*.println("流程实例："+processInstanceId+"激活");

}**else**{

//如果激活则挂起

runtimeService.suspendProcessInstanceById(processInstanceId);

System.*out*.println("流程实例："+processInstanceId+"挂起");

}

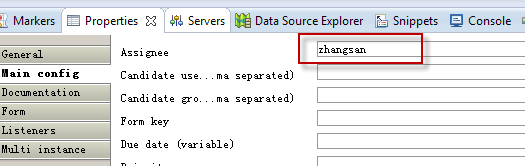
}

# 个人任务

## 分配任务负责人

### 固定分配

在进行业务流程建模时指定固定的任务负责人，



在properties视图中，填写Assignee项为任务负责人。

#### 注意事项

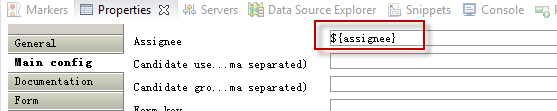
由于固定分配方式，任务只管一步一步执行任务，执行到每一个任务将按照bpmn的配置去分配任务负责人。

### 表达式分配

#### UEL表达式

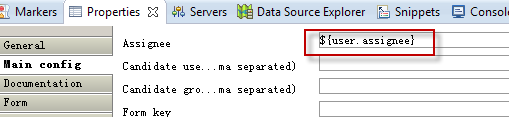
Activiti使用UEL表达式，UEL是java EE6规范的一部分，UEL（Unified Expression Language）即统一表达式语言，activiti支持两个UEL表达式：UEL-value和UEL-method。

* UEL-value定义如下：



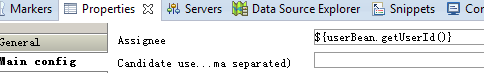
assignee这个变量是activiti的一个流程变量。

或：



user也是activiti的一个流程变量，user.assignee表示通过调用user的getter方法获取值。

* UEL-method方式如下：



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

* UEL-method与UEL-value结合

再比如：

${ldapService.findManagerForEmployee(emp)}

ldapService是spring容器的一个bean，findManagerForEmployee是该bean的一个方法，emp是activiti流程变量，emp作为参数传到ldapService.findManagerForEmployee方法中。

* 其它

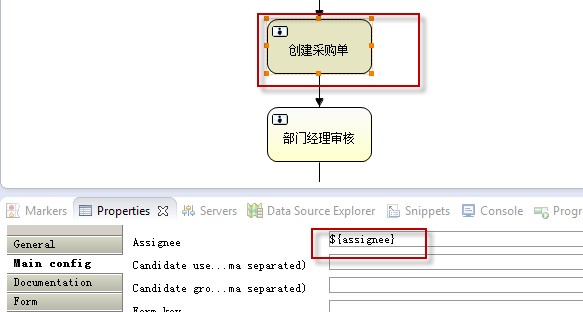
表达式支持解析基础类型、bean、list、array和map，也可作为条件判断。

如下：

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

#### 使用流程变量分配任务

* 定义任务分配流程变量



* 设置流程变量

在启动流程实例时设置流程变量，如下：

//启动流程实例时设计流程变量

//定义流程变量

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

//设置流程变量assignee

variables.put("assignee", "张三");

ProcessInstance processInstance = runtimeService

.startProcessInstanceByKey(processDefinitionKey, variables);

#### 注意事项

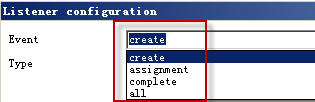
由于使用了表达式分配，必须保证在任务执行过程表达式执行成功，比如：

某个任务使用了表达式${order.price > 100 && order.price < 250}，当执行该任务时必须保证order在流程变量中存在，否则activiti异常。

### 监听器分配

任务监听器是发生对应的任务相关事件时执行自定义java逻辑 或表达式。

任务相当事件包括：



Create：任务创建后触发

Assignment：任务分配后触发

Complet：任务完成后触发

All：所有事件发生都触发

java逻辑 或表达式：

表达式参考上边的介绍的UEL表达式，这里主要介绍监听类使用。

定义任务监听类，且类必须实现org.activiti.engine.delegate.TaskListener接口

**public** **class** MyTaskListener **implements** TaskListener {

@Override

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

//这里指定任务负责人

delegateTask.setAssignee("张三");

}

}

#### 注意事项

使用监听器分配方式，按照监听事件去执行监听类的notify方法，方法如果不能正常执行也会影响任务的执行。

## 查询任务

查询任务负责人的待办任务：

// 查询当前个人待执行的任务

@Test

**public** **void** findPersonalTaskList() {

// 流程定义key

String processDefinitionKey = "purchasingflow01";

// 任务负责人

String assignee = "张三丰";

// 创建TaskService

TaskService taskService = processEngine.getTaskService();

List<Task> list = taskService.createTaskQuery()//

.processDefinitionKey(processDefinitionKey)//

.includeProcessVariables().taskAssignee(assignee).list();

**for** (Task task : list) {

System.*out*.println("----------------------------");

System.*out*.println("流程实例id：" + task.getProcessInstanceId());

System.*out*.println("任务id：" + task.getId());

System.*out*.println("任务负责人：" + task.getAssignee());

System.*out*.println("任务名称：" + task.getName());

}

}

## 办理任务

指定任务id，调用TaskService完成任务：

// 完成任务

@Test

**public** **void** completTask() {

//任务id

String taskId = "10305";

// 创建TaskService

TaskService taskService = processEngine.getTaskService();

taskService.complete(taskId);

System.*out*.println("完成任务");

}

# 应用

在采购系统中通过activiti管理采购流程，流程如下：

开始—》创建采购单—》部门经理审核采购单—》总经理审核采购单—》财务审核采购单—》结束

启动流程实例：

* 启动流程实例的时机：

创建采购单提交并启动流程实例。

* 业务系统与activiti数据共享：

启动实例记录business\_key，同时业务系统表中记录流程实例id。

* 配置文件需要配置的内容

流程定义key

查询待办任务：

从当前session中获取当前用户id，查询该用户的待办任务

* 采用单独的业务对象封装任务信息

通过service将action与activiti分离，使用activiti只与service交互，减少它们的耦合性。

完成任务：

* 提交采购单

完成之前进行校验，当前用户有无权限完成此任务，根据用户id和任务标识查询任务，能查询到则可以完成。

完成指定任务id的任务，调用taskService.complete(taskId)完成任务。

* 部门经理审核采购单

向采购系统审核表插入记录

调用taskService.complete(taskId)完成任务

* 总经理审核采购单

向采购系统审核表插入记录

调用taskService.complete(taskId)完成任务

* 财务审核采购单

向采购系统审核表插入记录

调用taskService.complete(taskId)完成任务

# 采购流程监控

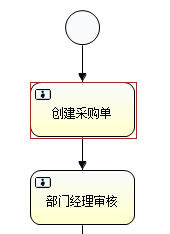
## 查询正在运行的流程

需求：查询系统中正在运行的采购流程，内容包括：流程实例id、采购单名称、采购单金额、开始时间、当前任务

分析：参考流程实例章节的内容，查询当前运行的流程，遍历查询出来的流程实例，根据流程实例id关联查询采购单信息。

动态流程图

需求：查询该流程实例的流程图，并且在图上标出当前运行到哪个结点，效果图如下：



当前运行的结点用红色边框标记。

分析：

查询每个流程实例的流程图，流程图属于流程定义的内容，根据流程实例id找到流程定义id（processDefinitionId）

ProcessInstance processInstance = runtimeService

.createProcessInstanceQuery()

.processInstanceId(processInstanceId).singleResult();

String processDefinitionId = processInstance

.getProcessDefinitionId();

据流程实例id找到当前运行活动标识activityId

// 当前任务id

String activityId = processInstance.getActivityId();

根据流程定义id得到流程定义实体对象ProcessDefinitionEntity，此对象中包括了bpmn定义的信息。

ProcessDefinitionEntity processDefinitionEntity = (ProcessDefinitionEntity) repositoryService

.getProcessDefinition(processDefinitionId);

从流程定义对象中根据上边找到的当前活动标识activityId找到当前活动定义对象ActivityImpl

ActivityImpl activityImpl = processDefinitionEntity

.findActivity(activityId);

从ActivityImpl中取出坐标信息.

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

// 使用map存放结点坐标

map.put("x", activityImpl.getX());

map.put("y", activityImpl.getY());

map.put("width", activityImpl.getWidth());

map.put("height", activityImpl.getHeight());

将坐标信息传到jsp页面，使用div显示红色边框效果

图形显示还使用查询流程定义资源的链接，页面代码如下：

<!-- 流程图 -->

<img style="position: *absolute*; top: *0px*; left: *0px*;"

src=*"*${baseurl}*flow/queryResource.action?processDefinitionId=*${processDefinitionId}*&resourceType=png"*>

<!-- 流程图中当前活动框 -->

<div

style=*"**position: absolute;border:1px solid red;width:* ${map.width }*px;height:*${map.height }*px;top:*${map.y }*px;left:* ${map.x }*px;"*></div>

## 查询已结束的流程

查询系统已经结束的采购流程，内容包括：流程实例id、采购单名称、采购单金额、开始时间、结束时间

使用HistoryService查询历史流程实例，设置查询条件只查询已完成的流程实例，再根据流程实例id关联采购单信息。

下边代码是查询已完成流程实例：

// 已结束流程查询

@Test

**public** **void** queryHistoryProcessInstance() **throws** Exception {

// 流程定义key

String processDefinitionKey = "purchasingflow01";

// 创建historyService

HistoryService historyService = processEngine.getHistoryService();

List<HistoricProcessInstance> list = historyService

.createHistoricProcessInstanceQuery()

.processDefinitionKey(processDefinitionKey)

.finished()//只查询已结束的流程

.list();

**for** (HistoricProcessInstance historicProcessInstance : list) {

System.*out*.println("流程定义id："

+ historicProcessInstance.getProcessDefinitionId());

System.*out*.println("流程实例id：" + historicProcessInstance.getId());

System.*out*

.println("开始时间：" + historicProcessInstance.getStartTime());

System.*out*.println("结束时间：" + historicProcessInstance.getEndTime());

System.*out*.println("持续时长："

+ historicProcessInstance.getDurationInMillis());

}

}

深入分析

* 问题：

通常，查询已结束的流程，用户的目的是要作历史数据操作，比如已完成采购单导出，历史采购统计分析等，这时就有一个问题，查询数据量可能会很大，比如一次把所有已完成的采购单导出使用上边的方法其性能是很差的，因为每条记录都要关联查询进行数据组合。

* 分析：

activiti中记录了流程的执行状态，但没有记录业务数据，虽然activiti可以通过流程变量记录业务系统，但不推荐将流程变量用于业务数据的查询和统计，因为不仅activiti使用流程变量查询速度不快，主要原因还是要分工明确，activiti只负责业务流程管理，不负责业务功能。

业务系统中有业务数据，但是没有记录采购单的状态，无法根据状态来查询已完成的采购单，而状态我们又不能在代码中硬编码，可以让activiti帮忙操作采购单表中的状态。

* 实现：

采购单流程完成后更新采购单表中的状态值为已完成，这样就可以根据采购单中的状态值是否完成来查询统计历史采购单记录了。

向流程结束结点添加事件监听器，监听器实现ExecutionListener接口，监听中根据businessKey更新采购单表的状态为complete。

## 查询流程历史任务

需求：查询当前运行的流程或历史结束的流程下所有任务完成情况，查询内容包括：

任务id、任务名称、任务负责人、开始时间、结束时间

思路：根据流程实例id查询该流程实例下所有历史任务，

代码如下：

// 历史任务查询

@Test

**public** **void** queryHistoryTask() **throws** Exception {

// 流程定义key

String processDefinitionKey = "purchasingflow01";

// 创建historyService

HistoryService historyService = processEngine.getHistoryService();

List<HistoricTaskInstance> list = historyService.createHistoricTaskInstanceQuery()

.processDefinitionKey(processDefinitionKey)

.processInstanceId("")//指定流程实例id

//.taskAssignee("")//只查询某个的历史任务

.list();

//遍历历史任务

**for**(HistoricTaskInstance historicTaskInstance:list){

System.*out*.println("任务id："+historicTaskInstance.getId());

System.*out*.println("任务标识："+historicTaskInstance.getTaskDefinitionKey());

System.*out*.println("任务名称："+historicTaskInstance.getName());

System.*out*.println("任务负责人："+historicTaskInstance.getAssignee());

System.*out*.println("任务开始时间："+historicTaskInstance.getStartTime());

System.*out*.println("任务结束时间："+historicTaskInstance.getEndTime());

System.*out*.println("持续时长："+historicTaskInstance.getDurationInMillis());

}

}

## 查询个人历史任务

查询用户办理的所有任务。

参考上边代码实现，从当前session中取出用户id设置到taskAssignee条件中。

# 流程变量

## 流程变量是什么

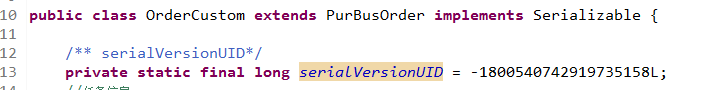
流程变量在activiti中是一个非常重要的角色，流程运转有时需要靠流程变量，业务系统和activiti结合时少不了流程变量，activiti流程变量就是activiti在管理工作流时根据管理需要而设置的变量

比如在采购流程流转时如果采购金额大于一万元则由总经理审核，否则由财务直接审核，采购金额就可以设置为流程变量，在流程流转时使用。

## 流程变量类型



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



注意：由于OrderCustom也继承了PurBusOrder类，PurBusOrder也必须实现序列化接口，否则 PurBusOrder中的属性无法反序列化。

## 流程变量作用域

流程变量的作用域默认是一个流程实例(processInstance)，也可以是一个任务(task)或一个执行实例(execution)，这三个作用域流程实例的范围最大，可以**称为global变量**，任务和执行实例仅仅是针对一个任务和一个执行实例范围，范围没有流程实例大，**称为local变量**。

如下图：

一个流程实例

Global变量：userId（变量名）：zhangsan（变量值）

变量名不允许重复

Local变量：userId：lisi

Local变量：userId：zhangsan

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

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

Local变量名也可以global变量名相同，没有影响。

## 设置global流程变量

### 启动流程时设置

在启动流程时设置流程变量，变量的作用域是整个流程实例。

通过map<key,value>设置流程变量，map中可以设置多个变量，这个key就是流程变量的名字。

// 启动流程时设置流程变量

@Test

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

// 流程定义key

String processDefinitionKey = "";

// 定义流程变量

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

variables.put("assignee", "张三");

RuntimeService runtimeService = processEngine.getRuntimeService();

ProcessInstance processInstance = runtimeService

.startProcessInstanceByKey(processDefinitionKey, variables);

System.*out*.println("流程实例id:" + processInstance.getProcessInstanceId());

}

说明：

startProcessInstanceByKey(processDefinitionKey, variables)流程变量作用域是一个流程实例，流程变量使用Map存储，同一个流程实例设置变量map中key相同，后者覆盖前者。

### 任务办理时设置

// 办理任务时设置流程变量

@Test

**public** **void** completTask() {

//任务id

String taskId = "";

TaskService taskService = processEngine.getTaskService();

// 定义流程变量

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

variables.put("assignee", "张三");

taskService.complete(taskId, variables);

}

说明：

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

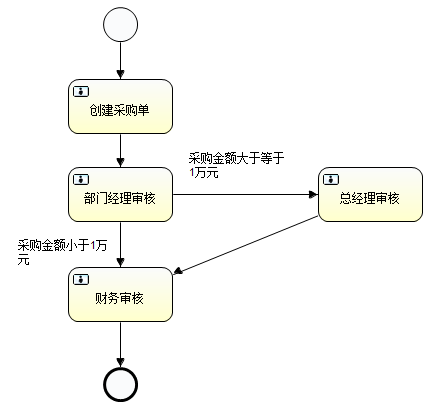
任务办理时也是通过map<key,value>设置流程变量，一次可以设置多个变量。

在完成任务时设置流程变量，流量的作用域是整个流程实例，如果设置的流程变量的key在流程实例中已存在相同的名字则后设置的变量替换前边设置的变量。

### Global变量测试

启动流程时设置流程变量，只要流程没有完成，在后边的任何结点都可以获取到该流程变量，因为该变量的作用域是整体流程实例。

需求：员工创建采购单，由部门经理审核，部门经理审核通过后一万元以下由财务直接审核，一万元以上先由总经理审核，总经理审核通过再由财务审核。



实现：启动采购流程，设置采购单信息为流程变量，每个结点就可以使用该流程变量信息。

在每个结点也可以修改global流程变量的信息，相同名称的变量后边修改的会覆盖前边设置的值。

## 设置local流程变量

### 任务办理时设置

// 办理任务时设置local流程变量

@Test

**public** **void** completTask() {

//任务id

String taskId = "";

TaskService taskService = processEngine.getTaskService();

// 定义流程变量

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

variables.put("assignee", "张三");

// 设置local变量，作用域为该任务

taskService.setVariablesLocal(tasked, variables);

taskService.complete(taskId);

}

说明：

设置作用域为任务的local变量，每个任务可以设置同名的变量，互不影响。

Local变量测试1

如果上边例子中设置global变量改为设置local变量是否可行？为什么？

### Local变量测试2

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

## 操作数据库表

设置流程变量会在当前执行流程变量表插入记录，同时也会在历史流程变量表也插入记录。

SELECT \* FROM act\_ru\_variable #当前执行流程变量表

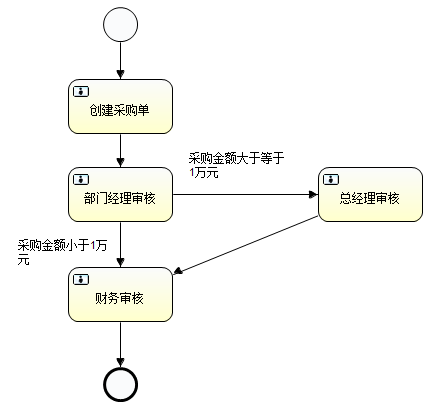
SELECT \* FROM act\_hi\_varinst #历史流程变量表

# 连线

连线是用于连接每个结点，使上一个结点执行完成 开始执行下一个结点。

## 案例1：

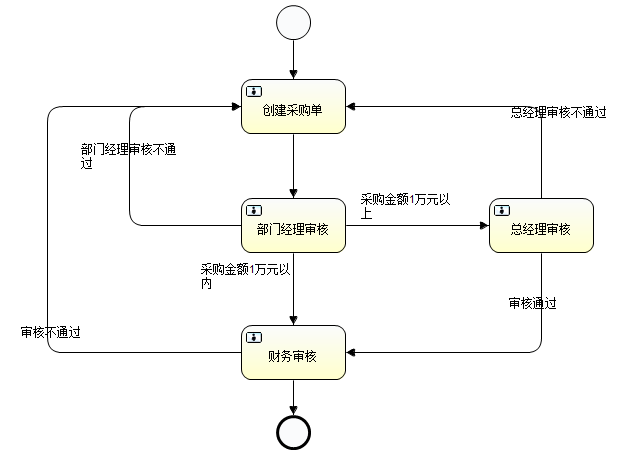
员工创建采购单，由部门经理审核，部门经理审核通过后一万元以下由财务直接审核，一万元以上先由总经理审核，总经理审核通过再由财务审核。



注意：如果连线条件全部不满足则流程结束。

## 案例2：

在上边的基础上添加审核不通过的连线。

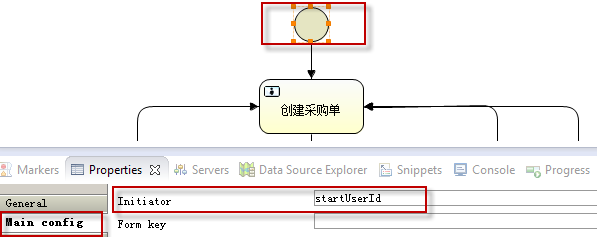


关于流程发起人

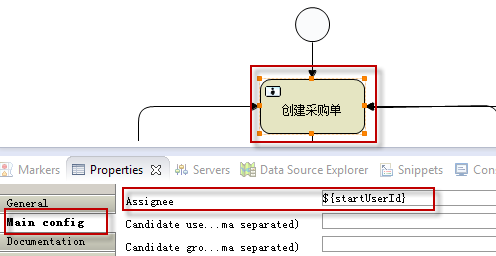
需求：创建采购单的人通常是不确定的，流程在启动时设置流程的发起人，在部门经理审核不通过、总经理审核不通过及财务审核不通过后由该发起人重新处理。

设置方法：

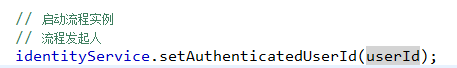
在开始结点，设置initiator（发起人）。



设置创建采购单的任务负责人为：



启动流程时设置任务发起人：



# 应用

在采购系统中实现流程审核不通过分支，功能如下：

* 部门经理审核不通过由员工重新修改采购单
* 总经理审核不通过由员工重新修改采购单
* 财务审核不通过由员工重新审核

图形参考连线—》案例2。

# 组任务

Candidate-users候选人

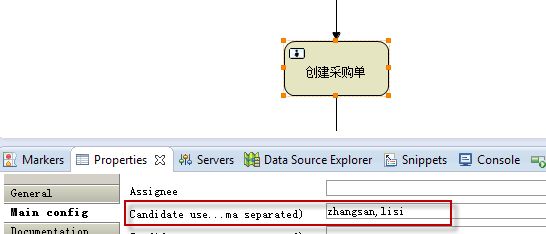
需求

业务流程在执行过程中，当执行到某个任务结点时大部分情况是不确定任务办理人的，比如总经理审核结点，用户为了提交工作效率可能会要求总经理和副总都可以审核，如果在流程定义时在assignee中已经指定任务办理人，这是无法满足需求的。

所以，在流程定义时不设置任务办理人，而是设置任务候选人。

### 设置任务候选人

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



### 办理组任务

用户查询组任务

根据候选人查询组任务

@Test

**public** **void** findGroupTaskList() {

// 流程定义key

String processDefinitionKey = "purchasingflow02";

// 任务候选人

String candidateUser = "zhangsan";

// 创建TaskService

TaskService taskService = processEngine.getTaskService();

//查询组任务

List<Task> list = taskService.createTaskQuery()//

.processDefinitionKey(processDefinitionKey)//

.taskCandidateUser(candidateUser)//根据候选人查询

.list();

**for** (Task task : list) {

System.*out*.println("----------------------------");

System.*out*.println("流程实例id：" + task.getProcessInstanceId());

System.*out*.println("任务id：" + task.getId());

System.*out*.println("任务负责人：" + task.getAssignee());

System.*out*.println("任务名称：" + task.getName());

}

}

用户拾取组任务

候选人员拾取组任务后该任务变为自己的个人任务。

@Test

**public** **void** claimTask(){

TaskService taskService = processEngine.getTaskService();

//要拾取的任务id

String taskId = "6302";

//任务候选人id

String userId = "lisi";

//拾取任务

//即使该用户不是候选人也能拾取(建议拾取时校验是否有资格)

//校验该用户有没有拾取任务的资格

Task task = taskService.createTaskQuery()//

.taskId(taskId)

.taskCandidateUser(userId)//根据候选人查询

.singleResult();

**if**(task!=**null**){

taskService.claim(taskId, userId);

System.*out*.println("任务拾取成功");

}

}

说明：即使该用户不是候选人也能拾取，建议拾取时校验是否有资格

组任务拾取后，该任务已有负责人，通过候选人将查询不到该任务

#### 用户查询个人待办任务

查询方式同个人任务查询

@Test

**public** **void** findPersonalTaskList() {

// 流程定义key

String processDefinitionKey = "groupTask01";

// 任务负责人

String assignee = "zhangsan";

// 创建TaskService

TaskService taskService = processEngine.getTaskService();

List<Task> list = taskService.createTaskQuery()//

.processDefinitionKey(processDefinitionKey)//

.taskAssignee(assignee).list();

**for** (Task task : list) {

System.*out*.println("----------------------------");

System.*out*.println("流程实例id：" + task.getProcessInstanceId());

System.*out*.println("任务id：" + task.getId());

System.*out*.println("任务负责人：" + task.getAssignee());

System.*out*.println("任务名称：" + task.getName());

}

}

用户办理个人任务

同个人任务办理

/\*\*完成任务\*/

@Test

**public** **void** completeTask(){

//任务ID

String taskId = "12304";

processEngine.getTaskService()//

.complete(taskId);

System.*out*.println("完成任务："+taskId);

}

说明：建议完成任务前校验该用户是否是该任务的负责人。

归还组任务

如果个人不想办理该组任务，可以归还组任务，归还后该用户不再是该任务的负责人

//归还任务

@Test

**public** **void** setAssignee(){

TaskService taskService = processEngine.getTaskService();

//要拾取的任务id

String taskId = "704";

//任务负责人

String userId = "zhangsan";

//归还任务，建议归还任务前校验该用户是否是该任务的负责人

taskService.setAssignee(taskId, **null**);

}

说明：建议归还任务前校验该用户是否是该任务的负责人

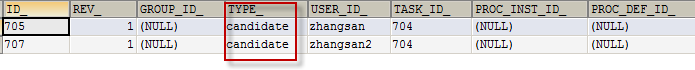
也可以通过setAssignee方法将任务委托给其它用户负责，注意被委托的用户可以不是候选人（建议不要这样使用）

#### 数据库表操作

SELECT \* FROM act\_ru\_task #任务执行表，记录当前执行的任务，由于该任务当前是组任务，所有assignee为空，当拾取任务后该字段就是拾取用户的id



SELECT \* FROM act\_ru\_identitylink #任务参与者，记录当前参考任务用户或组，当前任务如果设置了候选人，会向该表插入候选人记录，有几个候选就插入几个



于act\_ru\_identitylink对应的还有一张历史表act\_hi\_identitylink，向act\_ru\_identitylink插入记录的同时也会向历史表插入记录。任务完成

## Candidate-groups候选组

需求

候选人也是需要指定人，虽然是可以指定多个，但是业务系统用户可能会存在增加、删除的情况，如果增加人还得修改流程定义比较麻烦。

所以，不指定候选人而是指定候选组，组里的人可以调整的，调整组里边的人不需要修改程序定义。

Activiti中用户与组信息存储

表结构

Activiti使用三张表记录了用户与组的信息，

act\_id\_group #用户组表，相当于业务系统的角色表

act\_id\_user #用户表，相当于业务系统的用户表

act\_id\_membership #用户与组的对应关系，相当于业务系统的用户与角色的对应关系

设置用户和组信息

向activiti的这三张表设置信息，方法如下：

//设置用户和组信息

@Test

**public** **void** saveUserGroup(){

IdentityService identityService = processEngine.getIdentityService();

**if**(identityService.createGroupQuery().groupId("10").singleResult()==**null**){

GroupEntity groupEntity = **new** GroupEntity();

groupEntity.setId("10");

groupEntity.setName("员工");

identityService.saveGroup(groupEntity);

}

**if**(identityService.createGroupQuery().groupId("11").singleResult()==**null**){

GroupEntity groupEntity = **new** GroupEntity();

groupEntity.setId("11");

groupEntity.setName("部门经理");

identityService.saveGroup(groupEntity);

}

**if**(identityService.createGroupQuery().groupId("12").singleResult()==**null**){

GroupEntity groupEntity = **new** GroupEntity();

groupEntity.setId("12");

groupEntity.setName("总经理");

identityService.saveGroup(groupEntity);

}

**if**(identityService.createGroupQuery().groupId("13").singleResult()==**null**){

GroupEntity groupEntity = **new** GroupEntity();

groupEntity.setId("12");

groupEntity.setName("财务");

identityService.saveGroup(groupEntity);

}

**if**(identityService.createUserQuery().userId("zhangsan").singleResult()==**null**){

UserEntity userEntity = **new** UserEntity();

userEntity.setId("zhangsan");

userEntity.setFirstName("张三");

userEntity.setLastName("");

identityService.saveUser(userEntity);

}

**if**(identityService.createUserQuery().userId("lisi").singleResult()==**null**){

UserEntity userEntity = **new** UserEntity();

userEntity.setId("lisi");

userEntity.setFirstName("李四");

userEntity.setLastName("");

identityService.saveUser(userEntity);

}

**if**(identityService.createUserQuery().userId("wangwu").singleResult()==**null**){

UserEntity userEntity = **new** UserEntity();

userEntity.setId("wangwu");

userEntity.setFirstName("王五");

userEntity.setLastName("");

identityService.saveUser(userEntity);

}

**if**(identityService.createUserQuery().userId("zhaoliu").singleResult()==**null**){

UserEntity userEntity = **new** UserEntity();

userEntity.setId("zhaoliu");

userEntity.setFirstName("赵六");

userEntity.setLastName("");

identityService.saveUser(userEntity);

}

//添加用户和组关系

identityService.deleteMembership("zhangsan", "10");

identityService.createMembership("zhangsan", "10");

identityService.deleteMembership("lisi", "11");

identityService.createMembership("lisi", "11");

identityService.deleteMembership("wangwu", "12");

identityService.createMembership("wangwu", "12");

identityService.deleteMembership("zhaoliu", "13");

identityService.createMembership("zhaoliu", "13");

}

业务系统与activiti信息同步(了解)

系统正式运行时需要实时将业务系统的用户、角色、用户与角色关系表的记录同步到activiti的三张表中，通常采用从业务系统表到activiti表同步。

用户角色同步

业务系统添加角色-🡪activiti添加角色

业务系统修改角色🡪activiti修改角色

业务系统删除角色🡪activiti删除角色，删除之前将用户角色关系表先删除（根据角色删除）

用户同步：

业务系统添加用户----》activiti添加用户，添加用户与角色关系表

业务系统修改用户---》activiti修改用户，先原来用户与角色关系表，再添加用户与角色关系

业务系统删除用户—》activiti删除用户，删除之前将用户角色关系表删除（根据用户删除）

修改用户同步方法：

//user是业务系统更新的用户信息

//从activiti中查询用户信息

List<org.activiti.engine.identity.User> activitiUsers = identityService.createUserQuery().userId(userId).list();

//更新信息

org.activiti.engine.identity.User activitiUser = activitiUsers.get(0);

activitiUser.setFirstName(user.getName());

activitiUser.setLastName("");

activitiUser.setPassword(user.getPassword());

activitiUser.setEmail(user.getEmail());

identityService.saveUser(activitiUser);

// 删除用户的membership

List<Group> activitiGroups = identityService.createGroupQuery().groupMember(userId).list();

**for** (Group group : activitiGroups) {

identityService.deleteMembership(userId, group.getId());

}

// 添加membership

//roleIds是用户的最新角色信息，来自业务系统

**for** (Long roleId : roleIds) {

identityService.createMembership(userId, roleId);

}

添加用户同步方法：

//user是业务系统添加的用户

org.activiti.engine.identity.User newUser = identityService.newUser(userId);

newUser.setFirstName(user.getName());

newUser.setLastName("");

newUser.setPassword(user.getPassword());

newUser.setEmail(user.getEmail());

identityService.saveUser(newUser);

// 添加membership

//roleIds是业务系统添加用户的角色

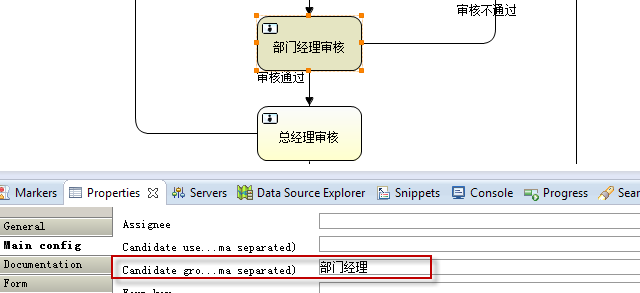
**for** (Long roleId : roleIds) {

identityService.createMembership(userId, roleId);

}

### 设置任务候选组

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



办理组任务

用户查询组任务

根据候选人查询组任务

@Test

**public** **void** findGroupTaskList() {

// 流程定义key

String processDefinitionKey = "purchasingflow02";

// 任务候选人

String candidateUser = "zhangsan";

// 创建TaskService

TaskService taskService = processEngine.getTaskService();

//查询组任务

List<Task> list = taskService.createTaskQuery()//

.processDefinitionKey(processDefinitionKey)//

.taskCandidateUser(candidateUser)//根据候选人查询

.list();

**for** (Task task : list) {

System.*out*.println("----------------------------");

System.*out*.println("流程实例id：" + task.getProcessInstanceId());

System.*out*.println("任务id：" + task.getId());

System.*out*.println("任务负责人：" + task.getAssignee());

System.*out*.println("任务名称：" + task.getName());

}

}

用户拾取组任务

候选人员拾取组任务

@Test

**public** **void** claimTask(){

TaskService taskService = processEngine.getTaskService();

//要拾取的任务id

String taskId = "704";

//任务候选人id

String userId = "zhangsan";

//拾取任务

//即使该用户不是候选人也能拾取(建议拾取时校验是否有资格)

taskService.claim(taskId, userId);

}

说明：即使该用户不是候选人也能拾取，建议拾取时校验是否有资格

组任务拾取后，该任务已有负责人，通过候选人将查询不到该任务书

用户查询个人待办任务

查询方式同个人任务查询

@Test

**public** **void** findPersonalTaskList() {

// 流程定义key

String processDefinitionKey = "groupTask01";

// 任务负责人

String assignee = "zhangsan";

// 创建TaskService

TaskService taskService = processEngine.getTaskService();

List<Task> list = taskService.createTaskQuery()//

.processDefinitionKey(processDefinitionKey)//

.taskAssignee(assignee).list();

**for** (Task task : list) {

System.*out*.println("----------------------------");

System.*out*.println("流程实例id：" + task.getProcessInstanceId());

System.*out*.println("任务id：" + task.getId());

System.*out*.println("任务负责人：" + task.getAssignee());

System.*out*.println("任务名称：" + task.getName());

}

}

用户办理个人任务

同个人任务办理

/\*\*完成任务\*/

@Test

**public** **void** completeTask(){

//任务ID

String taskId = "12304";

processEngine.getTaskService()//

.complete(taskId);

System.*out*.println("完成任务："+taskId);

}

说明：建议完成任务前校验该用户是否是该任务的负责人。

归还组任务

如果个人不想办理该组任务，可以归还组任务，归还后该用户不再是该任务的负责人

//归还任务

@Test

**public** **void** setAssignee(){

TaskService taskService = processEngine.getTaskService();

//要拾取的任务id

String taskId = "704";

//任务负责人

String userId = "zhangsan";

//归还任务，建议归还任务前校验该用户是否是该任务的负责人

taskService.setAssignee(taskId, **null**);

}

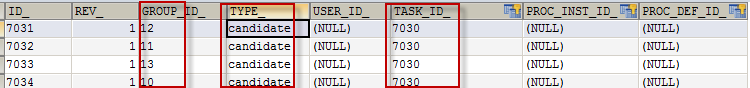
说明：建议归还任务前校验该用户是否是该任务的负责人

#### 数据库表操作

SELECT \* FROM act\_ru\_task #任务执行表，记录当前执行的任务，由于该任务当前是组任务，所有assignee为空，当拾取任务后该字段就是拾取用户的id



SELECT \* FROM act\_ru\_identitylink #任务参与者，记录当前参考任务用户或组，当前任务如果设置了候选组，会向该表插入候选组记录，有几个候选组就插入几个



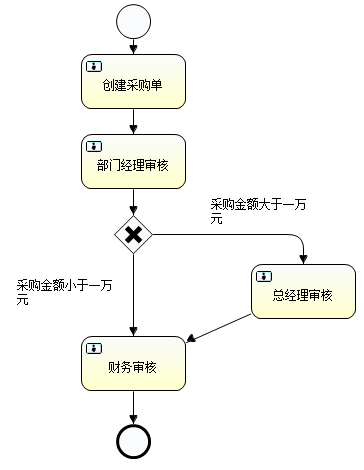
于act\_ru\_identitylink对应的还有一张历史表act\_hi\_identitylink，向act\_ru\_identitylink插入记录的同时也会向历史表插入记录。任务完成

# 网关

## 排他网关

排他网关（也叫异或（XOR）网关，或叫基于数据的排他网关），用来在流程中实现决策。 当流程执行到这个网关，所有分支都会判断条件是否为true，如果为true则执行该分支，注意，排他网关只会选择一个为true的分支执行。

将上边的例子改为排他网关实现，如下图：



说明：

使用排他网关的好处是，如果从网关出去的线所有条件都不满足则系统抛出异常。

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

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

## 并行网关

并行网关允许将流程分成多条分支，也可以把多条分支汇聚到一起，并行网关的功能是基于进入和外出的顺序流的：

* fork分支：

并行后的所有外出顺序流，为每个顺序流都创建一个并发分支。

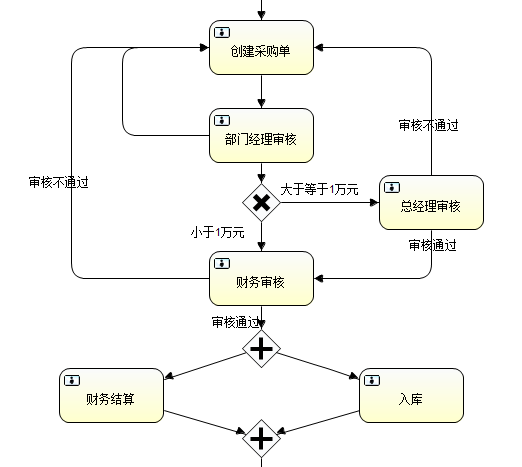
* join汇聚：

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

注意，如果同一个并行网关有多个进入和多个外出顺序流， 它就同时具有分支和汇聚功能。 这时，网关会先汇聚所有进入的顺序流，然后再切分成多个并行分支。

**与其他网关的主要区别是，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。**

例子：



说明：

财务结算和入库是两个execution分支，在act\_ru\_execution表有两条记录分别是财务结算和入库，act\_ru\_execution还有一条记录表示该流程实例。

待财务结算和入库任务全部完成，在汇聚点汇聚，通过parallelGateway并行网关。

并行网关在业务应用中常用于会签任务，会签任务即多个参与者共同办理的任务。

包含网关

包含网关可以看做是排他网关和并行网关的结合体。 和排他网关一样，你可以在外出顺序流上定义条件，包含网关会解析它们。 但是主要的区别是包含网关可以选择多于一条顺序流，这和并行网关一样。

包含网关的功能是基于进入和外出顺序流的：

* 分支：

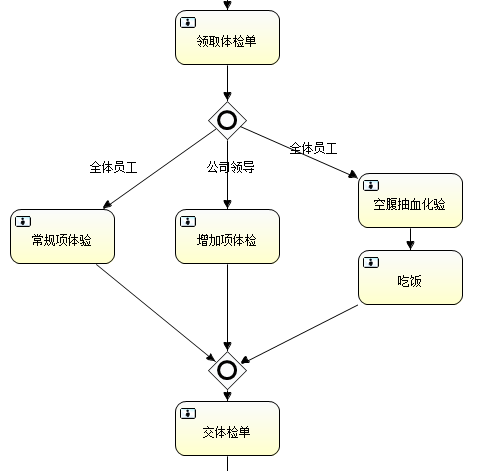
所有外出顺序流的条件都会被解析，结果为true的顺序流会以并行方式继续执行， 会为每个顺序流创建一个分支。

* 汇聚：

所有并行分支到达包含网关，会进入等待章台， 直到每个包含流程token的进入顺序流的分支都到达。 这是与并行网关的最大不同。换句话说，包含网关只会等待被选中执行了的进入顺序流。 在汇聚之后，流程会穿过包含网关继续执行。

例子：

企业体检流程，公司全体员工进行常规项检查、抽血化验，公司管理层除常规检查和抽血化验还要进行增加项检查。



# 应用

* 将采购流程改为组任务实现
* 在采购流程中实现排他网关
* 在采购流程中实现并行网关