**前言**

Activiti 从入门到放弃 [滑稽]

**1.什么是工作流**

**1.1 工作流介绍**

专家解释：[工作流](https://baike.baidu.com/item/%E5%B7%A5%E4%BD%9C%E6%B5%81)

个人理解：在计算机的帮助下，实现一个业务过程的自动化控制

比如请假流程：

员工申请 -> 部门经理审批 -> 总经理审批 -> 人事存档

在传统方式下，请假条是通过人工传递来实现的

在无纸化办公下，请假条通过线上申请 -> 线上审批 -> 保存到数据库，这样就形成了一条请假记录

**1.2 工作流的实现**

在没有使用工作流引擎之前，我们之前为了实现流程控制，通常是通过状态字段的值来跟踪流程状态

比如请假流程：

员工申请 -> 部门经理审批 -> 总经理审批 -> 人事存档

表设计：

| **列名** |
| --- |
| 请假单ID |
| 部门ID |
| 员工ID |
| 请假天数 |
| 请假开始日期 |
| 请假结束日期 |
| 请假原因 |
| 状态 |

员工：状态=0 表示：未提交；状态=1 表示：已提交

部门经理：状态=2 表示：同意；状态=3 表示：不同意

总经理：状态=4 表示：同意；状态=5 表示：不同意

人事：查询状态=4的申请成功的请假单数据

通过状态字段，也能够实现流程的控制，但是有个弊端：当我们业务需求有变动，代码也要发生改变

比如：总经理审批环节不需要了，则程序不能使用了

那么能不能做到业务流程变化后，程序可以不用改变？而 Activiti 就能做到，当需求变更时，代码不需要发生改变，只需要更新业务流程图

**2.什么是 Activiti**

**2.1 Activiti 介绍**

专家介绍：[Activiti](https://baike.baidu.com/item/activiti/10300939?fr=aladdin)

官网：[官网](https://www.activiti.org/)

快速开始：[Quickstart](https://activiti.gitbook.io/activiti-7-developers-guide/getting-started)

官方示例：[Activiti Core Examples](https://github.com/Activiti/activiti-examples)

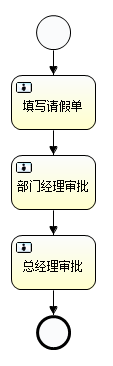
**2.2 Activiti 原理**

为什么 Activiti 能解决当业务需求变更时，不需要更新代码，只需要更新业务流程图，就可以实现流程变化？

大概原理：

1. 将流程图画好
2. 将流程图的第一个节点读取到表中
3. 查询表中的要处理的节点，进行处理后，删除，并读取下一个节点存入数据库表中
4. 循环第3个步骤，直到结束

比如请假流程：

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565750967521.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565750967521.png" \o "1565750967521)

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565751669869.png)](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565751669869.png)

为了实现上面的大概原理，需要满足：

1. 业务流程图必须遵循一套标准进行制作
2. 业务流程图需要能进行解析，如果业务流程图是一个 XML 文件，这样能存入我们需要的数据，比如审批人是谁
3. 读取业务流程图的过程就是解析 XML 文件的过程
4. 读取业务流程图中的一个节点，就相当于解析一个 XML 节点，然后将数据保存到数据库表中，形成一条记录，后面只需要读取数据库表记录即可，读取数据库一条记录，就等于读取一个节点
5. 业务流程的推进，就转换成读取表数据、处理表数据，处理完成删除这条数据并读取下一条数据

**2.3 相关知识**

为了实现上面的大概原理的条件，需要了解一些小知识

**2.3.1 BPMN**

上面说到业务流程图必须遵循一套标准进行制作，而这个标准就是 BPMN

专家介绍：[BPMN](https://baike.baidu.com/item/BPMN/9818373?fr=aladdin)

简介：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。

Activiti 就是使用 BPMN 2.0 进行流程建模、流程执行管理。

**BPMN 文件 本质是一个 XML 文件**

**2.3.2 Activiti 流程设计器**

流程设计器能帮助我们，使用图像界面进行建模，而不是自己写 XML

**2.3.2.1 Eclipse 安装**

[官方安装教程](https://docs.alfresco.com/5.2/tasks/wf-install-activiti-designer.html)

**2.3.2.2 IDEA 安装**

[actiBPM](https://plugins.jetbrains.com/plugin/7429-actibpm)

**开发时，推荐使用 Eclipse 的流程设计器，不推荐使用 IDEA 的流程设计器，actiBPM 对 2014 版本后的 IDEA 不怎么兼容，经常报错**

**2.3.3 Activiti 使用步骤**

1. 部署 Activiti：引入相关 jar 包
2. 流程定义：使用建模工具定义业务流程，生成 .bpmn 文件
3. 流程定义部署：向 Activiti 部署业务流程定义，即 .bpmn 文件
4. 启动一个流程实例：启动一个流程实例，表示开始一次业务流程的运行，比如请假流程定义部署完成，张三需要请假，就可以启动一个流程实例，李四需求请假，也可以启动一个流程实例，两个流程的执行互相不影响。可以理解为流程定义部署就是定义了一个 Java 类，启动流程实例就是 new Java 对象
5. 查询待办任务：因为现在系统业务流程已经交给 Activiti 管理，所以通过 Activiti 就可以查询当前流程执行到哪，用户需要处理什么任务了
6. 处理待办任务：用户查询待办任务后，就可以处理某个任务，如果这个任务处理完成，还有其他流程，Activiti 会帮我们完成下一个任务的创建和指派
7. 流程结束：当任务处理完成没有下一个任务了，该流程实例就完成了

**3.Activiti 环境搭建**

**3.1 项目搭建**

项目使用 PostgreSQL 数据库

1. 创建 activiti\_test 数据库
2. 创建 Maven 项目，并引入 Activiti 相关依赖

pom.xml 文件

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 | <properties>  <activiti.version>7.1.0.M2</activiti.version> </properties>  <dependencies>  <!-- activiti start -->  <dependency>  <groupId>org.activiti</groupId>  <artifactId>activiti-engine</artifactId>  <version>${activiti.version}</version>  </dependency>  <dependency>  <groupId>org.activiti</groupId>  <artifactId>activiti-spring</artifactId>  <version>${activiti.version}</version>  </dependency>  <dependency>  <groupId>org.activiti</groupId>  <artifactId>activiti-bpmn-model</artifactId>  <version>${activiti.version}</version>  </dependency>  <dependency>  <groupId>org.activiti</groupId>  <artifactId>activiti-bpmn-converter</artifactId>  <version>${activiti.version}</version>  </dependency>  <dependency>  <groupId>org.activiti</groupId>  <artifactId>activiti-json-converter</artifactId>  <version>${activiti.version}</version>  </dependency>  <dependency>  <groupId>org.activiti</groupId>  <artifactId>activiti-bpmn-layout</artifactId>  <version>${activiti.version}</version>  </dependency>  <!-- activiti end -->  <!-- log start -->  <dependency>  <groupId>log4j</groupId>  <artifactId>log4j</artifactId>  <version>1.2.12</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>1.6.6</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  <version>1.6.6</version>  </dependency>  <!-- log end -->  <dependency>  <groupId>org.postgresql</groupId>  <artifactId>postgresql</artifactId>  <version>42.2.6</version>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  <version>1.1.13</version>  </dependency> </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <configuration>  <source>8</source>  <target>8</target>  </configuration>  </plugin>  </plugins> </build> |

1. 配置日志

log4j.properties 文件

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 | log4j.rootCategory=debug, CONSOLE, LOGFILE  log4j.logger.org.apache.axis.enterprise=FATAL, CONSOLE  log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout log4j.appender.CONSOLE.layout.ConversionPattern=%d{ISO8601} %-6r [%15.15t] %-5p %30.30c %x - %m\n |

1. 配置 Activiti

activiti.cfg.xml

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 | <?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:tx="http://www.springframework.org/schema/tx"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/contex http://www.springframework.org/schema/context/spring-context.xsd  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd">   <!--数据库配置-->  <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">  <property name="driverClassName" value="org.postgresql.Driver"/>  <property name="url" value="jdbc:postgresql://localhost:5432/activiti\_test"/>  <property name="username" value="postgres"/>  <property name="password" value="postgres"/>  </bean>   <!--  activiti单独运行的ProcessEngine配置对象（ProcessEngineCOnfiguration），使用单独启动方式  默认情况下，bean的id为 processEngineConfiguration  -->  <bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">  <!--数据原配置-->  <property name="dataSource" ref="dataSource"/>  <!--代表是否生成表结构-->  <property name="databaseSchemaUpdate" value="true"/>  </bean>  </beans> |

1. 初始化 Activiti 需要的表：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | public class ActivitiTest {   public static void main(String[] args) {  // 1.创建 ProcessEngineConfiguration 对象  // ProcessEngineConfiguration processEngineConfiguration = ProcessEngineConfiguration.createProcessEngineConfigurationFromResource("activiti.cfg.xml");  // 2.创建 ProcessEngine 对象  // ProcessEngine processEngine = processEngineConfiguration.buildProcessEngine();  // 快速创建  ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();  System.err.println(processEngine);   RepositoryService repositoryService = processEngine.getRepositoryService();  RuntimeService runtimeService = processEngine.getRuntimeService();  TaskService taskService = processEngine.getTaskService();  HistoryService historyService = processEngine.getHistoryService();  ManagementService managementService = processEngine.getManagementService();  DynamicBpmnService dynamicBpmnService = processEngine.getDynamicBpmnService();  }  } |

**3.2 Activiti 数据库表**

创建 ProcessEngine 对象后，会根据配置决定是否创建表。如果需要创建表，会创建以 act\_ 开头的25张表

其中表的命名规则为：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

**3.2.1 act\_re\_\* 开头的表**

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）

| **表名** | **描述** |
| --- | --- |
| act\_re\_deployment | 流程定义部署表 |
| act\_re\_model | 流程定义模型表 |
| act\_re\_procdef | 流程定义表 |

**3.2.2 act\_ru\_\* 开头的表**

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

| **表名** | **描述** |
| --- | --- |
| act\_ru\_deadletter\_job | 无法执行的工作表 |
| act\_ru\_event\_subscr | 事件描述表 |
| act\_ru\_execution | 执行表，保存流程实例和执行数据 |
| act\_ru\_identitylink | 流程与身份的关系表 ，如用户与任务的关系 |
| act\_ru\_integration |  |
| act\_ru\_job | 一般的工作表 |
| act\_ru\_suspended\_job | 中断的工作表 |
| act\_ru\_task | 任务表 |
| act\_ru\_timer\_job | 定时器工作表 |
| act\_ru\_variable | 参数表，用于任务参数，流程参数 |

**3.2.3 act\_hi\_\* 开头的表**

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

| **表名** | **描述** |
| --- | --- |
| act\_hi\_actinst | 历史流程实例活动表 |
| act\_hi\_attachment | 流程附件表 |
| act\_hi\_comment | 流程评论表 |
| act\_hi\_detail | 变更历史表 |
| act\_hi\_identitylink | 历史参与者表 |
| act\_hi\_procinst | 流程实例的历史数据表 |
| act\_hi\_taskinst | 流程实例历史任务表 |
| act\_hi\_varinst | 历史参数表 |

**3.2.4 act\_ge\_\* 开头的表**

ge 表示 general，通用数据，用于不同场景下

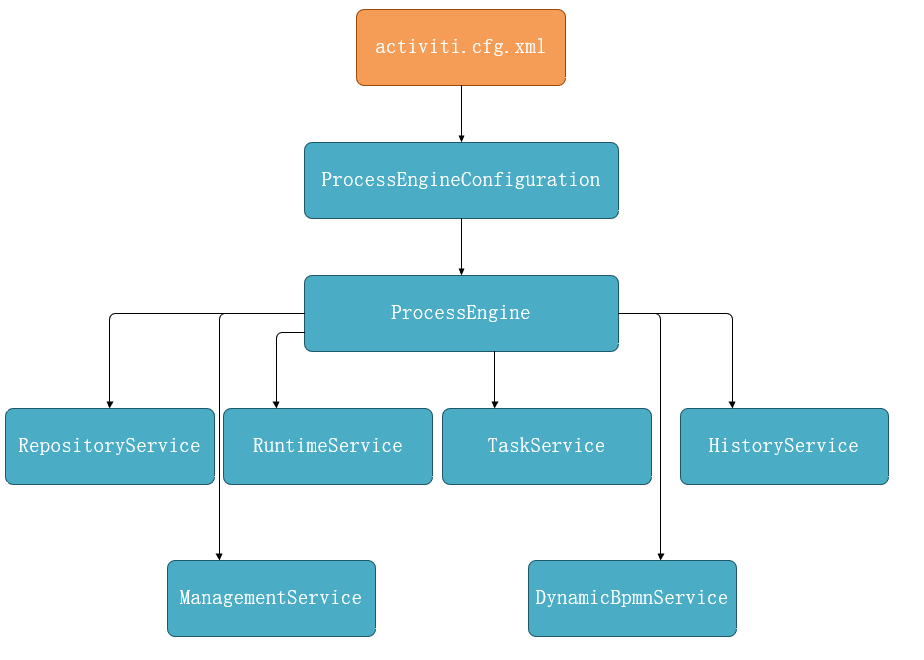
| **表名** | **描述** |
| --- | --- |
| act\_ge\_bytearray | bpmn、png 等二进制内容 |
| act\_ge\_property | 引擎版本信息 |

**3.2.5 其他表**

| **表名** | **描述** |
| --- | --- |
| act\_evt\_log | 事件日志 |
| act\_procdef\_info | 流程定义信息 |

**4.Activiti 服务架构**

**4.1 服务架构图**

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565762310444.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565762310444.png" \o "1565762310444)

**4.2 activiti.cfg.xml**

activiti 的引擎配置文件，包括：ProcessEngineConfiguration 的定义、数据源定义、事物管理器等，此文件其实就是一个 spring 配置文件

**4.3 ProcessEngineConfiguration**

流程引擎的配置类，通过ProcessEngineConfiguration 可以创建工作流引擎 ProcessEngine，常用两种实现：

1. StandaloneProcessEngineConfiguration：Activiti 可以单独运行，使用它创建的 ProcessEngine，Acitiviti 会自己处理事物
2. SpringProcessEngineConfiguration：通过这个类与 Spring 整合

创建 ProcessEngineConfiguration 实例的方法：

1. 当 activiti.cfg.xml 中 ProcessEngineConfiguration 的 bean 的 id 为 processEngineConfiguration 时，可以使用：

|  |  |
| --- | --- |
| 1 | ProcessEngineConfiguration.createProcessEngineConfigurationFromResource("activiti.cfg.xml"); |

1. 也可以通过指定 bean 的名称来创建

|  |  |
| --- | --- |
| 1 | ProcessEngineConfiguration.createProcessEngineConfigurationFromResource("activiti.cfg.xml", "beanName"); |

**4.4 ProcessEngine**

ProcessEngine 工作流引擎，相当于一个门面接口。通过 ProcessEngineConfiguration 创建 ProcessEngine，通过 ProcessEngine 获取各个 service 接口

创建 ProcessEngine 实例的方法：

1. 一般创建方式：

|  |  |
| --- | --- |
| 1 | processEngineConfiguration.buildProcessEngine(); |

1. 简单创建方式，要求 activiti 配置文件的文件位置固定以及文件名为 activiti.cfg.xml，并且文件中有 processEngineConfiguration 的配置：

|  |  |
| --- | --- |
| 1 | ProcessEngines.getDefaultProcessEngine(); |

**4.5 Service**

**4.5.1 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

|  |  |
| --- | --- |
| 1 2 3 4 5 6 | RepositoryService repositoryService = processEngine.getRepositoryService(); RuntimeService runtimeService = processEngine.getRuntimeService(); TaskService taskService = processEngine.getTaskService(); HistoryService historyService = processEngine.getHistoryService(); ManagementService managementService = processEngine.getManagementService(); DynamicBpmnService dynamicBpmnService = processEngine.getDynamicBpmnService(); |

**4.5.2 Service 总览**

| **名称** | **描述** | **作用** |
| --- | --- | --- |
| RepositoryService | 仓库服务 | 仓库服务是存储相关的服务，一般用来部署流程文件，获取流程文件，查询流程定义信息等操作，是引擎中一个重要的服务。 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 | 运行时服务 | 流程运行时的流程实例，流程定义，流程版本，流程节点等信息，使用运行时服务操作，是引擎中的一个重要服务。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 | 任务服务 | 流程运行时产生的任务，对任务进行接受、办理、完成等操作使用任务服务完成，是引擎中的一个重要服务。是 activiti 的任务管理类。可以从这个类中获取任务的信息 |
| HistoryService | 历史服务 | 流程运行时，和运行完成之后的一些历史信息，包括历史任务，历史节点等，是引擎中的一个重要服务。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 |
| ManagementService | 引擎管理服务 | 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 |
| DynamicBpmnService | 动态流程服务 | 用于动态修改流程中的一些参数信息等，是引擎中一个辅助服务 |

**5.Activiti 入门使用**

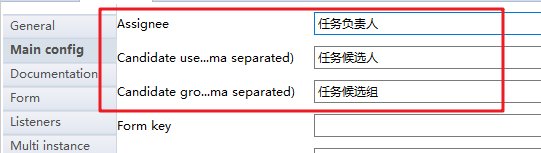
**5.1 流程定义**

使用 activiti-designer 进行流程定义，需要指定 流程定义Key，一般来说不同流程定义的 key 是不同的，在流程定义部署的时候 Activiti 也是根据这个 key 来修改版本号的

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565763690030.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565763690030.png" \o "1565763690030)

还可以指定：

* 任务负责人
* 任务候选人
* 任务候选人组

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565763678848.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565763678848.png" \o "1565763678848)

**[1565763678848](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565763678848.png" \o "1565763678848)**

[**1565763678848**](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565763678848.png)

**5.2 流程定义部署**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | // 1.创建 ProcessEngine 实例 ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine(); // 2.获取 RepositoryService 仓库服务 RepositoryService repositoryService = processEngine.getRepositoryService(); // 3.进行部署 Deployment deploy = repositoryService.createDeployment()  .name("请假流程")  .addClasspathResource("process/leave.bpmn")  .addClasspathResource("process/leave.png")  .deploy(); System.err.println("流程部署ID：" + deploy.getId()); System.err.println("流程部署名字：" + deploy.getName());  /\*  \* 涉及到的表：  \*act\_re\_procdef：流程定义  \*act\_re\_deployment：流程部署  \*act\_ge\_bytearray：bpmn、png 等二进制信息  \*/ |

影响的表：

* act\_re\_procdef：流程定义
* act\_re\_deployment：流程部署
* act\_ge\_bytearray：bpmn、png 等二进制信息

**5.3 启动流程实例**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | // 1.获取 ProcessEngine ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine(); // 2.获取 RuntimeService 运行时服务 RuntimeService runtimeService = processEngine.getRuntimeService(); // 3.启动流程实例 ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("leave"); System.err.println("流程定义ID：" + processInstance.getProcessDefinitionId()); System.err.println("流程实例ID：" + processInstance.getId()); System.err.println("当前活动ID：" + processInstance.getActivityId()); /\*  \* 涉及到的表：  \* act\_hi\_taskinst：流程实例的历史任务  \* act\_hi\_procinst：流程实例的历史数据  \* act\_hi\_actinst：历史流程实例  \* act\_hi\_identitylink：流程与参与者的信息  \* act\_ru\_execution：执行表，保存流程实例和执行数据  \* act\_ru\_task：任务  \* act\_ru\_identitylink：流程与参与者的信息  \*/ |

影响的表：

* act\_hi\_taskinst：流程实例的历史任务
* act\_hi\_procinst：流程实例的历史数据
* act\_hi\_actinst：历史流程实例
* act\_hi\_identitylink：流程与参与者的信息
* act\_ru\_execution：执行表，保存流程实例和执行数据
* act\_ru\_task：任务
* act\_ru\_identitylink：流程与参与者的信息

**5.4 查询任务**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | // 1.获取 ProcessEngine 实例 ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine(); // 2.获取 TaskService 任务服务 TaskService taskService = processEngine.getTaskService(); // 3.根据 流程定义的key 和 参与者 查询用户的任务 List<Task> taskList = taskService.createTaskQuery()  .processDefinitionKey("leave")  .taskAssignee("zhangsan")  .list(); System.err.println("当前任务总数：" + taskList.size()); for (Task task : taskList) {  System.err.println("流程实例ID：" + task.getProcessInstanceId());  System.err.println("当前任务ID：" + task.getId());  System.err.println("当前任务名称：" + task.getName()); } |

查询语句：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | SELECT DISTINCT  RES.\*  FROM  ACT\_RU\_TASK RES  INNER JOIN ACT\_RE\_PROCDEF D ON RES.PROC\_DEF\_ID\_ = D.ID\_  WHERE  RES.ASSIGNEE\_ = 'zhangsan'   AND D.KEY\_ = 'leave'  ORDER BY  RES.ID\_ ASC   LIMIT 2147483647 OFFSET 0; |

**5.5 处理任务**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | // 1.获取流程引擎实例 ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine(); // 2.获取任务服务 TaskService taskService = processEngine.getTaskService(); // 3.完成任务 taskService.complete("2505"); /\*  \* 涉及到的表：  \* insert：  \* act\_hi\_taskinst：流程实例的历史任务  \* act\_hi\_actinst：历史流程实例  \* act\_hi\_identitylink：流程与参与者的信息  \* act\_ru\_task：任务  \* act\_ru\_identitylink：流程与参与者的信息  \*  \* update：  \* act\_ru\_execution：执行表，保存流程实例和执行数据  \* act\_hi\_actinst：历史流程实例  \* act\_hi\_taskinst：流程实例的历史任务  \*  \* delete：  \* act\_ru\_task：任务  \*/ |

涉及到的表：

* insert
  + act\_hi\_taskinst：流程实例的历史任务
  + act\_hi\_actinst：历史流程实例
  + act\_hi\_identitylink：流程与参与者的信息
  + act\_ru\_task：任务
  + act\_ru\_identitylink：流程与参与者的信息
* update
  + act\_hi\_taskinst：流程实例的历史任务
  + act\_hi\_actinst：历史流程实例
  + act\_hi\_identitylink：流程与参与者的信息
  + act\_ru\_task：任务
  + act\_ru\_identitylink：流程与参与者的信息
* delete
  + act\_ru\_task：任务

当最后一个流程走完：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 | // 1.获取流程引擎实例 ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine(); // 2.获取任务服务 TaskService taskService = processEngine.getTaskService(); // 3.查询用户任务，假设能够确定只有一条 Task task = taskService.createTaskQuery()  .processDefinitionKey("leave")  .taskAssignee("wangwu")  .singleResult(); System.err.println("流程实例ID：" + task.getProcessInstanceId()); System.err.println("当前任务ID：" + task.getId()); System.err.println("当前任务名称：" + task.getName()); // 3.完成任务 taskService.complete(task.getId()); /\*  \* 当最后一个流程走完  \* 涉及到的表：  \* insert：  \* act\_hi\_actinst：历史流程实例  \*  \* update：  \* act\_hi\_actinst：历史流程实例  \* act\_hi\_taskinst：流程实例的历史任务  \* act\_ru\_execution：执行表，保存流程实例和执行数据，会修改两条  \* act\_hi\_procinst： 流程实例的历史数据  \*  \* delete：  \* act\_ru\_identitylink：流程与参与者的信息，会删除三条  \* act\_ru\_task：任务  \* act\_ru\_execution：执行表，保存流程实例和执行数据，会删除两条  \*/ |

涉及到的表：

* insert
  + act\_hi\_actinst：历史流程实例
* update
  + act\_hi\_actinst：历史流程实例
  + act\_hi\_taskinst：流程实例的历史任务
  + act\_ru\_execution：执行表，保存流程实例和执行数据，会修改两条
  + act\_hi\_procinst： 流程实例的历史数据
* delete
  + act\_ru\_identitylink：流程与参与者的信息，会删除三条
  + act\_ru\_task：任务
  + act\_ru\_execution：执行表，保存流程实例和执行数据，会删除两条

**6.流程定义**

**6.1 什么是流程定义**

流程定义是线下按照 bpmn2.0 标准去描述业务流程，通常使用 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

**6.2 什么是流程定义部署**

将线下定义的流程部署到 activiti 数据库中，这就是流程定义部署，通过调用 activiti 的 api 将流程定义的 bpmn 和 png 文件添加部署到 activiti 中，也可以将两个文件打包成 ZIP 进行部署

**6.2.1 单个文件部署方式**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | // 1.创建 ProcessEngine 实例 ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine(); // 2.获取 RepositoryService 仓库服务 RepositoryService repositoryService = processEngine.getRepositoryService(); // 3.进行部署 Deployment deploy = repositoryService.createDeployment()  .name("请假流程")  .addClasspathResource("process/leave.bpmn")  .addClasspathResource("process/leave.png")  .deploy(); System.err.println("流程部署ID：" + deploy.getId()); System.err.println("流程部署名字：" + deploy.getName()); /\*  \* 涉及到的表：  \* act\_re\_procdef：流程定义  \* act\_re\_deployment：流程部署  \* act\_ge\_bytearray：bpmn、png 等二进制信息  \*/ |

**6.2.2 ZIP 压缩包部署方式**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | // 1.获取流程引擎实例 ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine(); // 2.获取仓库服务 RepositoryService repositoryService = processEngine.getRepositoryService(); // 3.获取 ZIP 压缩包输入流 InputStream inputStream = ActivitiZipDeploy.class.getClassLoader().getResourceAsStream("process/leave.zip"); assert inputStream != null; ZipInputStream zipInputStream = new ZipInputStream(inputStream); // 3.进行部署 Deployment deploy = repositoryService.createDeployment()  .name("请假流程")  .addZipInputStream(zipInputStream)  .deploy(); System.err.println("流程部署ID：" + deploy.getId()); System.err.println("流程部署名称：" + deploy.getName()); |

**6.2.3 操作的数据库表**

* act\_re\_procdef：流程定义，记录流程定义信息
* act\_re\_deployment：流程部署，记录了流程部署信息
* act\_ge\_bytearray：bpmn、png 等二制信息，资源表

说明：

act\_re\_deployment 和 act\_re\_procdef 是一对多关系，一次部署在流程部署表中生成一条记录，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关系，方便读取流程部署及流程定义信息

**6.3 流程定义操作**

**6.3.1 流程定义查询**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | // 1.获取流程引擎实例 ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine(); // 2.获取仓库服务 RepositoryService repositoryService = processEngine.getRepositoryService(); // 3.获取流程定义查询实例，设置查询条件，并查询 List<ProcessDefinition> processDefinitionList = repositoryService.createProcessDefinitionQuery()  .processDefinitionKey("leave")  .orderByProcessDefinitionVersion()  .desc()  .list(); // 4.遍历 System.err.println("当前部署的流程定义总数：" + processDefinitionList.size()); for (ProcessDefinition processDefinition : processDefinitionList) {  System.err.println("流程定义ID："+processDefinition.getId());  System.err.println("流程定义名字："+processDefinition.getName());  System.err.println("流程定义Key："+processDefinition.getKey());  System.err.println("流程定义版本号："+processDefinition.getVersion());  System.err.println("流程定义部署ID："+processDefinition.getDeploymentId()); } |

**6.3.2 流程定义删除**

使用 RepositoryService 删除流程定义

**6.3.2.1 普通删除**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 | // 1.获取流程引擎实例 ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine(); // 2.获取仓库服务实例 RepositoryService repositoryService = processEngine.getRepositoryService(); // 3.删除流程实例 repositoryService.deleteDeployment("2501"); |

**6.3.2.2 级联删除**

如果该流程定义下存在已经运行的流程，使用普通删除会报错，可以使用级联删除方法，将流程以及相关记录全部删除，项目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

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 | // 1.获取流程引擎实例 ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine(); // 2.获取仓库服务 RepositoryService repositoryService = processEngine.getRepositoryService(); // 3.获取流程定义查询实例，设置查询条件，并查询 ProcessDefinition processDefinition = repositoryService.createProcessDefinitionQuery()  .processDefinitionKey("leave")  .singleResult(); System.err.println("流程定义ID：" + processDefinition.getId()); System.err.println("流程定义名字：" + processDefinition.getName()); System.err.println("流程定义Key：" + processDefinition.getKey()); System.err.println("流程定义版本号：" + processDefinition.getVersion()); System.err.println("流程定义部署ID：" + processDefinition.getDeploymentId()); // 4.删除流程实例 // 参数1：流程定义部署的ID // 参数2：true为级联删除，即如果这个流程定义有实例正在执行，那么现在删除正在执行的实例，然后再删除流程定义部署 repositoryService.deleteDeployment(processDefinition.getDeploymentId(), 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实例，那么可以普通删除  \* 3.如果该流程定义下存在已经运行的流程，使用普通删除会报错，可以使用级联删除方法，将流程以及相关记录全部删除  \* 项目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  \*/ |

**6.3.3 流程定义资源查询**

**6.3.3.1 方式1 通过查询流程定义对象获取流程定义资源**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 | // 1.获取流程引擎对象 ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine(); // 2.获取仓库服务对象 RepositoryService repositoryService = processEngine.getRepositoryService(); // 方式一 通过查询流程定义信息获取流程定义资源 // 1.查询流程定义信息 ProcessDefinition processDefinition = repositoryService.createProcessDefinitionQuery()  .processDefinitionKey("leave")  .singleResult(); // 2.获取流程定义资源 bpmn png 的名称 String bpmnFileName = processDefinition.getResourceName(); String pngFileName = processDefinition.getDiagramResourceName(); System.err.println("bpmn文件名：" + bpmnFileName); System.err.println("png文件名：" + pngFileName); // 3.获取输入流，并输出文件 try (  InputStream bpmnFileIs = repositoryService.getResourceAsStream(processDefinition.getDeploymentId(), bpmnFileName);  InputStream pngFileIs = repositoryService.getResourceAsStream(processDefinition.getDeploymentId(), pngFileName);  OutputStream bpmnOs = new FileOutputStream("D:\\Users\\Administrator\\Desktop\\test\\" + bpmnFileName);  OutputStream pngOs = new FileOutputStream("D:\\Users\\Administrator\\Desktop\\test\\" + pngFileName); ) {  // 输出 bpmn 文件  byte[] buffer = new byte[1024];  int len = -1;  while ((len = bpmnFileIs.read(buffer, 0, 1024)) != -1) {  bpmnOs.write(buffer, 0, len);  }  // 输出 png 文件  while ((len = pngFileIs.read(buffer, 0, 1024)) != -1) {  pngOs.write(buffer, 0, len);  } } catch (IOException e) {  e.printStackTrace(); } |

**6.3.3.1 方式2 通过查询流程部署信息获取流程定义资源**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 | // 方式二 通过查询流程部署信息获取流程定义资源 List<String> deploymentResourceNameList = repositoryService.getDeploymentResourceNames(processDefinition.getDeploymentId()); for (String fileName : deploymentResourceNameList) {  if (fileName.endsWith(".png")) {  pngFileName = fileName;  } else if (fileName.endsWith(".bpmn")) {  bpmnFileName = fileName;  } } System.err.println("bpmn文件名：" + bpmnFileName); System.err.println("png文件名：" + pngFileName); try (  final InputStream bpmnIs = repositoryService.getResourceAsStream(processDefinition.getDeploymentId(), bpmnFileName);  final InputStream pngIs = repositoryService.getResourceAsStream(processDefinition.getDeploymentId(), pngFileName);  final FileOutputStream bpmnOs = new FileOutputStream("D:\\Users\\Administrator\\Desktop\\test\\2-" + bpmnFileName);  final FileOutputStream pngOs = new FileOutputStream("D:\\Users\\Administrator\\Desktop\\test\\2-" + pngFileName); ) {  byte[] buffer = new byte[1024];  int let = -1;  while ((let = bpmnIs.read(buffer, 0, 1024)) != -1) {  bpmnOs.write(buffer, 0, let);  }  while ((let = pngIs.read(buffer, 0, 1024)) != -1) {  pngOs.write(buffer, 0, let);  } } catch (Exception e) {  e.printStackTrace(); } |

**6.3.4 流程历史信息查询**

即使流程定义以及删除（前提是没有级联删除），流程执行的历史信息依然保存再 act\_hi\_\* 相关的表中。所以我们可以查询流程执行的历史信息。可以通过 HistoryService 来查看相关的历史记录

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | // 1.获取流程引擎对象 ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine(); // 2.获取历史服务 HistoryService historyService = processEngine.getHistoryService(); // 3.查询历史信息 List<HistoricActivityInstance> historicActivityInstances = historyService.createHistoricActivityInstanceQuery()  .processInstanceId("27501")  .orderByHistoricActivityInstanceEndTime()  .desc()  .list(); historicActivityInstances.forEach(historicActivityInstance -> {  System.err.println("关联的流程实例ID："+historicActivityInstance.getProcessInstanceId());  System.err.println("活动ID："+historicActivityInstance.getActivityId());  System.err.println("活动名称："+historicActivityInstance.getActivityName());  System.err.println("活动类型："+historicActivityInstance.getActivityType());  System.err.println("参与者："+historicActivityInstance.getAssignee());  System.err.println("========================"); }); |

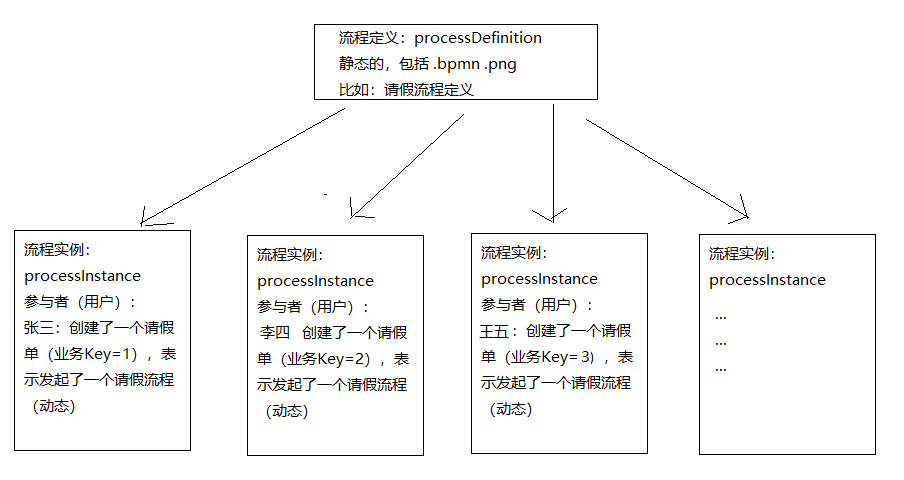
**7.流程实例**

**7.1 什么是流程实例**

参与者（可以是用户也可以是程序）按照流程定义内容发起一个流程，这个就是一个流程实例，是动态的

流程定义是静态的，可以理解为 Java 类，流程实例的动态的，可以理解为 Java 实例

流程定义和流程实例图解：

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565764380063.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565764380063.png" \o "1565764380063)

**7.2 启动流程实例**

流程定义部署在 activiti 后，就可以在系统中通过 acitiviti 去管理该流程的执行，执行流程表示流程的一次执行。比如部署系统请假流程后，如果某用户需要申请请假单时就需要执行这个流程，如果另外一个用户也要申请请假则也需要执行该流程，每个执行互补影响，每个执行是单独的流程实例

执行流程首先要启动流程实例

|  |  |
| --- | --- |
| 1 2 | RuntimeService runtimeService = ProcessEngines.getDefaultProcessEngine().getRuntimeService(); runtimeService.startProcessInstanceByKey("流程定义的Key"); |

**7.3 BusinessKey 业务标识**

启动流程实例时，指定的 businessKey，就会在 act\_ru\_execution 表（流程实例执行表）中存储 businessKey

BusinessKey：业务标识，通常为业务表的主键，业务标识和流程实例一一对应。业务标识来源于业务系统。存储业务标识就是根据业务系统标识来关联查询业务系统的数据

比如：请假流程启动一个流程实例，就可以将请假单 ID 作为业务标识存储到 activiti 中，将来查询 activiti 的流程实例信息就可以获取请假单的 ID 从而关联查询业务系统数据库得到请假单信息

|  |  |
| --- | --- |
| 1 2 3 | RuntimeService runtimeService = ProcessEngines.getDefaultProcessEngine().getRuntimeService(); // 根据流程定义Key启动一个流程实例，并且设置业务标识 runtimeService.startProcessInstanceByKey("流程定义的Key", "业务标识"); |

BusinessKey 保存到表：

* act\_hi\_procinst：流程实例的历史数据
* act\_ru\_execution：执行表，保存流程实例和执行数据，重点是这张表

**7.4 数据库表操作**

启动流程实例，操作了如下数据库表：

1. 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并且执行表的主键 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

1. 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节点，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

1. act\_ru\_identitylink：任务参与者，记录当前参与任务的用户或组
2. act\_hi\_procinst：历史流程实例表

说明：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

1. act\_hi\_taskinst：历史任务表，记录所有任务

说明：开始一个任务，不仅在 act\_ru\_task 表插入记录，也会在历史任务表中插入一条记录，任务历史表的主键 ID 就是任务 ID，任务完成后，此表记录不会删除

1. act\_hi\_actinst：历史活动表，记录所有活动

活动包括任务，所以此表不仅记录了任务，还记录了流程执行过程的其他活动，比如开始事件、结束事件

**7.5 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，当前运行节点等信息

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | RuntimeService runtimeService = ProcessEngines.getDefaultProcessEngine().getRuntimeService(); ProcessInstance processInstance = runtimeService.createProcessInstanceQuery()  .processInstanceBusinessKey("98k")  .singleResult(); System.err.println("当前流程实例ID：" + processInstance.getId()); System.err.println("当前流程实例BusinessKey：" + processInstance.getBusinessKey()); System.err.println("当前流程实例所属的流程定义ID：" + processInstance.getProcessDefinitionId()); System.err.println("当前流程实例是否完成：" + processInstance.isEnded()); System.err.println("当前流程实例是否挂起：" + processInstance.isSuspended()); System.err.println("当前流程实例活动标识：" + processInstance.getActivityId()); |

**7.5.1 查询流程实例关联 BusinessKey**

需求：在 activiti 实际应用中，查询流程实例表时可能需要显示出业务系统的一些相关信息，比如：查询当前运行的请假流程列表需要请假单名称、请假单天数等信息出来，请假天数等信息在业务系统中存在，而并没有在 activiti 数据库中存在，所以是无法通过 activiti 的 api 查询到请假天数等信息

实现：在查询流程实例时，通过 businessKey（业务标识）关联查询业务系统的请假单表，查询出请假天数等信息

通过下面的代码就可以获取 activiti 中所对应流程实例保存的业务 key，而这个业务 key 一般会保存相关联的业务操作表的主键，再通过主键 ID 去查询业务信息，比如通过请假单的 ID，去查询更多的请求信息（请假人、请假实际、请假天数、请假事由等）

|  |  |
| --- | --- |
| 1 | String businessKey = processInstance.getBusinessKey(); |

在 activiti 的 act\_ru\_execution 表，字段 business\_key\_ 就是存放业务 key 的

**7.6 挂起、激活流程实例**

某些情况下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

**7.6.1 挂起、激活全部流程实例**

操作流程定义为挂起状态，该流程定义下面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流程实例将全部挂起暂停执行

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | // 1.获取仓库服务 RepositoryService repositoryService = ProcessEngines.getDefaultProcessEngine().getRepositoryService(); // 2.查流程定义 ProcessDefinition processDefinition = repositoryService.createProcessDefinitionQuery()  .processDefinitionKey("leave")  .singleResult(); // 3.根据状态进行暂停或启动流程定义并暂停或启动关联的流程实例 if (processDefinition.isSuspended()) {  // 参数1：流程定义的 Key  // 参数2：是否激活关联的流程实例  // 参数3：激活流程定义的日期，如果为 null 则立即执行，否则在指定的时间点执行  repositoryService.activateProcessDefinitionByKey("leave", true, null);  System.err.println("启动流程定义"); } else {  // 参数1：流程定义的 Key  // 参数2：是否暂停关联的流程实例  // 参数3：暂停流程定义的日期，如果为 null 则立即执行，否则在指定的时间点执行  repositoryService.suspendProcessDefinitionByKey("leave", true, null);  System.err.println("暂停流程定义"); } |

**7.6.2 挂起、激活单个流程实例**

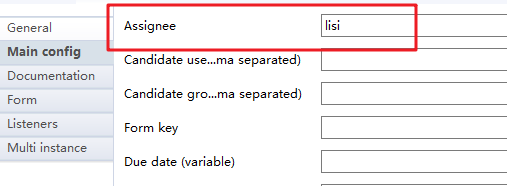
|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | // 1.获取运行时服务 RuntimeService runtimeService = ProcessEngines.getDefaultProcessEngine().getRuntimeService(); // 2.查询要暂停的流程实例 ProcessInstance processInstance = runtimeService.createProcessInstanceQuery()  .processInstanceBusinessKey("98k")  .singleResult(); // 3.根据状态进行启动或暂停 if (processInstance.isSuspended()) {  runtimeService.activateProcessInstanceById(processInstance.getId());  System.err.println("启用流程实例"); } else {  runtimeService.suspendProcessInstanceById(processInstance.getId());  System.err.println("暂停流程实例"); } |

**8.任务分配**

**8.1 分配任务负责人**

**8.1.1 固定分配**

在进行业务流程建模时指定固定的任务负责人，填写 Assignee 项为任务负责人

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565764900965.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565764900965.png" \o "1565764900965)

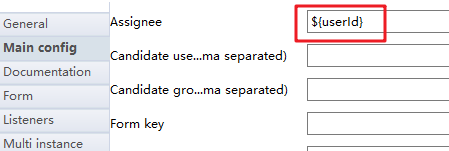
**注意事项：由于固定分配方式，任务只管一步一步执行任务，执行到每一个任务将按照 bpmn 的配置去分配任务负责人**

**8.1.2 表达式分配**

**8.1.2.1 UEL表达式**

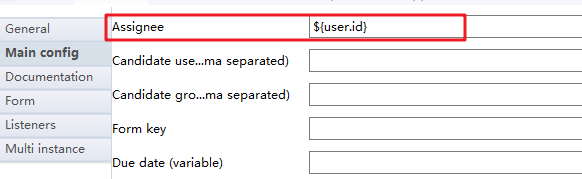
Activiti 使用 UEL 表达式，URL 是 Java EE6 规范的一部分，UEL（Unified Expression Language）即统一表达式语言，Activiti 支持两个 UEL 表达式：UEL-value 和 UEL-method

* UEL-value 定义：

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565764967879.png)](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565764967879.png)

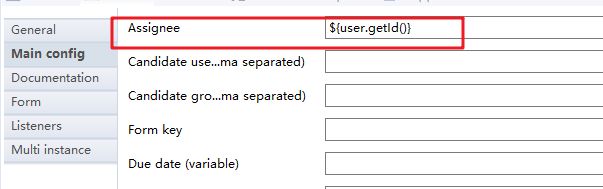
userId 这个变量是 activiti 的一个流程变量

或

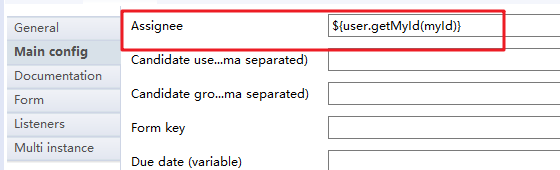
[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565765006703.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565765006703.png" \o "1565765006703)

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

* UEL-method 方式：

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565765070543.png)](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565765070543.png)

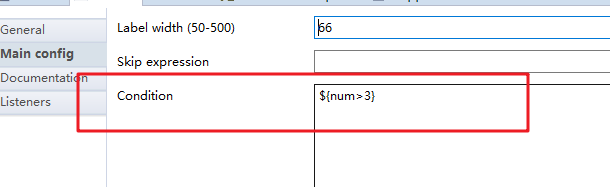
* UEL-method 与 UEL-vaule 结合：

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565765136569.png)](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565765136569.png)

调用 user 对象的 getMyId() 方法，并传入方法参数 myId，user 和 myId 都是 Activiti 的流程变量

* 其他：表达式支持解析基础类型、bean、list、array、map，也可以作为条件判断

如：${num > 3}

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565765251876.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565765251876.png" \o "1565765251876)

**8.1.2.2 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

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | // 1.获取运行时服务 RuntimeService runtimeService = ProcessEngines.getDefaultProcessEngine().getRuntimeService(); // 2.定义流程变量 String userId = "userId"; UserEntity user = new UserEntity("user.id"); String myId = "getMyId"; Map<String, Object> params = new HashMap<>(3); params.put("userId", userId); params.put("user", user); params.put("myId", myId); // 3.创建流程实例，并设置流程变量 // 参数1：流程定义 Key // 参数2：业务标识 // 参数3：流程变量 Map ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("leave2", "98k", params); System.err.println("流程实例ID：" + processInstance.getId()); |

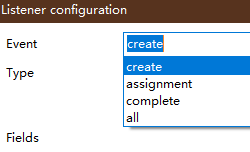
**8.1.2.3 注意事项**

1. 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，必须保证流程变量存在，否则 activiti 异常
2. 如果流程变量是 JavaBean 需要实现 Serializable 接口

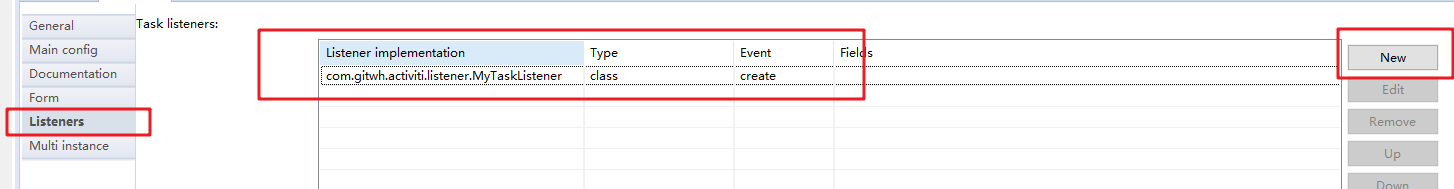
**8.1.3 监听器分配**

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

任务事件包括：

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565765520083.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565765520083.png" \o "1565765520083)

| **事件** | **描述** |
| --- | --- |
| Create | 任务创建后触发 |
| Assignment | 任务分配后触发 |
| Delete | 任务完成后触发 |
| All | 所有事件发生都触发 |

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565765644463.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565765644463.png" \o "1565765644463)

定义任务监听器类，必须实现 org.activiti.engine.delegate.TaskListener 接口，并且重写 public void notify(DelegateTask delegateTask) 方法

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | public class MyTaskListener implements TaskListener {  private static final long serialVersionUID = -9093762200587658336L;   @Override  public void notify(DelegateTask delegateTask) {  delegateTask.setAssignee("boss");  } } |

**8.2 查询任务**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 | // 1.获取任务服务 TaskService taskService = ProcessEngines.getDefaultProcessEngine().getTaskService(); // 获取运行时服务，用来查询流程实例 RuntimeService runtimeService = ProcessEngines.getDefaultProcessEngine().getRuntimeService(); for (int i = 0; i < 4; i++) {  // 2.查询任务  Task task = taskService.createTaskQuery()  .processInstanceBusinessKey("98k")  .singleResult();  System.err.println("任务ID：" + task.getId());  System.err.println("任务名称：" + task.getName());  System.err.println("任务参与者：" + task.getAssignee());  System.err.println("任务创建时间：" + task.getCreateTime());  // 获取任务关联的流程实例的 BusinessKey  ProcessInstance processInstance = runtimeService.createProcessInstanceQuery()  .processInstanceId(task.getProcessInstanceId())  .singleResult();  System.err.println("任务关联的业务Key：" + processInstance.getBusinessKey());  // 3.处理任务  taskService.complete(task.getId()); } |

**8.2.1 根据任务获取 BusinessKey**

|  |  |
| --- | --- |
| 1 2 3 4 5 | // 获取任务关联的流程实例的 BusinessKey ProcessInstance processInstance = runtimeService.createProcessInstanceQuery()  .processInstanceId(task.getProcessInstanceId())  .singleResult(); System.err.println("任务关联的业务Key：" + processInstance.getBusinessKey()); |

**8.3 处理任务**

|  |  |
| --- | --- |
| 1 | taskService.complete(task.getId()); |

注意：在实际应用中，完成任务前需要校验任务的负责人是否具有该任务的办理权限，需要根据任务 ID 和当前操作的用户 ID 查询任务，如果任务存在则有权限，否则没有权限

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 | // 1.获取任务服务 TaskService taskService = ProcessEngines.getDefaultProcessEngine().getTaskService(); for (int i = 0; i < 4; i++) {  // 2.查询任务  Task task = taskService.createTaskQuery()  .processInstanceBusinessKey("98k")  .singleResult();  if (task == null) {  System.err.println("任务不存在");  continue;  }  System.err.println("任务ID：" + task.getId());  System.err.println("任务名称：" + task.getName());  System.err.println("任务参与者：" + task.getAssignee());  System.err.println("任务创建时间：" + task.getCreateTime());  // 3.处理任务，判断当前的任务负责人是否有权限处理该任务  // 要处理的任务 ID  String taskId = task.getId();  // 当前任务负责人，假设通过解析 JWT 获取  String assignee = task.getAssignee();  // 根据任务 ID 和负责人查询当前任务，如果查询到，则说明该用户有完成权限，否则没有  Task result = taskService.createTaskQuery()  .taskId(taskId)  .taskAssignee(assignee)  .singleResult();  if (result == null) {  throw new RuntimeException("没有权限处理该任务");  }  taskService.complete(task.getId()); } |

**9.流程变量**

**9.1 什么是流程变量**

流程变量在 activiti 中是一个非常重要的角色，流程运转时需要靠流程变量，业务系统和 activiti 结合时少不了流程变量，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

比如在请假流程流转时如果请假天数大于3天则由总经理审核，否则由人事审核，请假天数就可以设置为流程变量，在流程运转时使用

**注意：虽然流程变量中可以储存业务数据，可以通过 activiti 的 api 查询流程变量从而实现查询业务数据，但是不建议这样使用，因为业务数据查询由业务系统负责，activiti 设置流程变量是为了流程执行需要而创建**

**9.2 流程变量类型**

* String
* Integer
* Short
* Long
* Double
* Boolean
* Date
* Binary
* Serializable

注意：如果将 POJO 储存到流程变量中，必须实现序列化接口，为了防止由于新字段无法反序列化，需要生成 serialVersionUID

**9.3 流程变量的作用域**

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，可以成为 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

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面设置的变量值

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 变量名也可以和 Global 变量名相同，没有影响

**9.4 流程变量的使用方法**

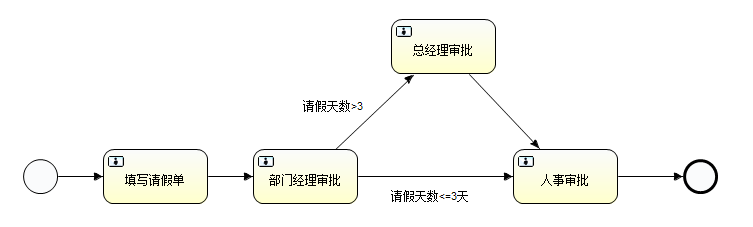
1. 设置流程变量
2. 通过 UEL 表达式使用流程变量
   1. 可以在 assignee 处设置 UEL 表达式，表达式的值为任务的负责人，如：${userId} userId 就是一个流程变量名称，Activiti 获取 UEL 表达式的值，就流程变量 userId 的值，将 userId 值作为任务的负责人进行分配
   2. 可以在连接线上设置 UEL 表达式，决定流程走向，比如：${price >= 10000} 和 ${price < 10000} ，price 就是一个流程变量名称，UEL 表达式结果类型为布尔类型，如果 UEL 表达式是 true，则决定流程执行走向

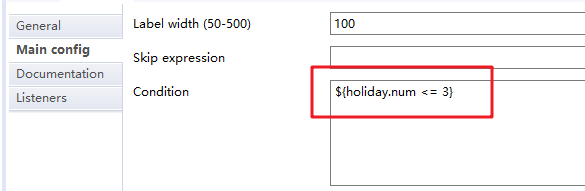
**9.5 使用 Global 变量控制流程**

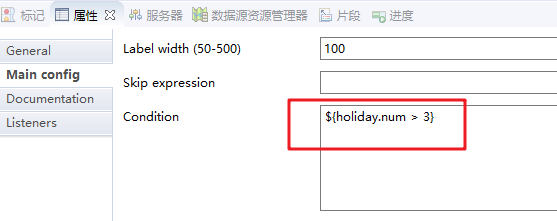
**9.5.1 需求**

员工创建请假申请单，由部门经理审核，部门经理审核通过后，请假3天及以下由人事直接审核，3天以上由总经理审核，总经理审核再由人事审核

**9.5.2 流程定义**

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565765850793.png)](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565765850793.png)

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565765854754.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565765854754.png" \o "1565765854754)

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565765858516.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565765858516.png" \o "1565765858516)

**[1565765858516](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565765858516.png" \o "1565765858516)**

[**1565765858516**](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565765858516.png)

**9.5.3 设置 Global 流程变量**

在部门经理审核前设置流程变量，变量值为请假单信息（包括请假天数），部门经理审核后可以根据流程变量决定流程走向

**9.5.3.1 启动流程变量时设置**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | // 1.获取运行时服务 RuntimeService runtimeService = ProcessEngines.getDefaultProcessEngine().getRuntimeService(); // 2.准备流程实例变量 Map<String, Object> variables = new HashMap<>(1); Holiday holiday = new Holiday(); holiday.setNum(4D); variables.put("holiday", holiday); // 3.启动流程实例并且设置业务标识和流程实例变量 runtimeService.startProcessInstanceByKey("holiday", "98k", variables); |

说明：startProcessInstanceByKey(String processDefinitionKey, String businessKey, Map<String, Object> variables) 流程变量作用域是一个流程实例，流程变量使用 Map 储存，同一个流程实例设置变量 Map 中 Key 相同，后者覆盖前者

**9.5.3.2 任务处理时设置**

在完成任务时设置流程变量，该流程变量只有在该任务完成后其他节点才可以使用该变量，它的作用域是整个流程实例，如果设置的流程变量的 key 在该流程实例中以已经存在相同的名字则后设置的变量替换前面设置的变量

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | // 1.获取任务服务 TaskService taskService = ProcessEngines.getDefaultProcessEngine().getTaskService(); // 2.根据业务标识查询当前任务 Task task = taskService.createTaskQuery()  .processInstanceBusinessKey("98k")  .singleResult(); if (task != null) {  // 3.准备流程实例变量  Map<String, Object> variables = new HashMap<>(1);  Holiday holiday = new Holiday();  holiday.setNum(3D);  variables.put("holiday", holiday);  // 参数1：任务ID  // 参数2：流程实例变量，作用域是整个流程实例，完成任务后其他节点可以使用这个流程变量  taskService.complete(task.getId(), variables);  System.err.println("任务ID：" + task.getId());  System.err.println("任务名称：" + task.getName()); } |

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

**9.5.3.3 通过当前流程实例的运行实例设置**

通过流程实例的运行实例 ID 设置全局变量，该流程实例必须未执行完成

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 | // 1.获取运行时服务 RuntimeService runtimeService = ProcessEngines.getDefaultProcessEngine().getRuntimeService(); // 2.查询流程实例的运行实例 Execution execution = runtimeService.createExecutionQuery()  .processInstanceBusinessKey("98k")  .singleResult(); if (execution != null) {  // 3.准备流程参数  Holiday holiday = new Holiday();  holiday.setNum(6D);  // 4.根据流程实例运行ID，设置流程实例的流程参数  // 4.1 设置一个参数  // 参数1：流程实例的运行实例ID，一般流程实例的运行ID会等于流程实例ID  // 参数2：参数key  // 参数3：参数value  runtimeService.setVariable(execution.getId(), "holiday", holiday);  // 4.2 设置多个参数  Map<String, Object> variables = new HashMap<>(1);  variables.put("holiday", holiday);  holiday.setNum(1D);  // 参数1：流程实例的运行实例ID  // 参数2：流程参数 Map  runtimeService.setVariables(execution.getId(), variables);  // 5.获取流程参数  // 5.1 获取所有流程参数  Map<String, Object> variables1 = runtimeService.getVariables(execution.getId());  System.err.println(variables1);  // 5.2 根据 key 获取  Holiday holiday1 = runtimeService.getVariable(execution.getId(), "holiday", Holiday.class);  System.err.println(holiday1); } |

注意：executionId 必须是当前未结束的流程实例的执行 ID，通常此 ID 是流程实例的 ID，也可以通过 RuntimeService.getVariable() 获取流程变量

**9.5.3.4 通过当前任务设置**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 | // 1.获取任务服务 TaskService taskService = ProcessEngines.getDefaultProcessEngine().getTaskService(); // 2.准备流程参数 Holiday holiday = new Holiday(); holiday.setNum(6D); // 3.根据任务 ID 设置流程参数 String taskId = "2505"; // 3.1 设置一个参数 // 参数1：任务ID // 参数2：流程变量 key // 参数3：流程变量 value taskService.setVariable(taskId, "holiday", holiday); // 3.2 设置多个参数 Map<String, Object> variables = new HashMap<>(1); variables.put("holiday", holiday); // 参数1：任务ID // 参数2：流程变量 Map taskService.setVariables(taskId, variables); // 4.获取任务的流程参数 // 4.1 获取所有 Map<String, Object> variables1 = taskService.getVariables(taskId); System.err.println(variables1); // 4.2 根据 key 获取 Holiday holiday1 = taskService.getVariable(taskId, "holiday", Holiday.class); System.err.println(holiday1); |

注意：任务 ID 必须是当前代办任务 ID，在 act\_ru\_task 中存在，如果任务已经结束，会报错

也可以通过 TaskService.getVeriable() 来获取流程变量

**9.5.3.5 操作的数据库表**

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

* insert：
  + act\_ge\_bytearray add var-holiday
  + act\_ge\_bytearray add hist.var-holiday
  + act\_hi\_varinst
  + act\_ru\_variable
* update：
  + act\_ru\_execution

act\_ru\_variable：当前流程变量表，记录当前运行流程实例可以使用的流程变量，包括 global 和 local 变量

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | Id\_：主键 Type\_：变量类型 Name\_：变量名称 Execution\_id\_：所属流程实例执行 id，global和 local变量都存储 Proc\_inst\_id\_：所属流程实例 id，global和 local变量都存储 Task\_id\_：所属任务 id，local变量存储 Bytearray\_：serializable 类型变量存储对应act\_ge\_bytearray 表的 id Double\_：double 类型变量值 Long\_：long类型变量值 Text\_：text 类型变量值 |

act\_hi\_varinst：历史流程变量表，记录所有已创建的流程变量，包括 Global 和 Local 变量

**9.6 设置 Local 流程变量**

**9.6.1 任务办理时设置**

任务办理时设置 Local 流程变量，当前运行的流程实例只能在该任务结束前使用，任务结束该变量无法在当前实例中使用，可以通过查询历史任务查询

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 | // 1.获取任务服务 TaskService taskService = ProcessEngines.getDefaultProcessEngine().getTaskService(); // 2.查询任务 Task task = taskService.createTaskQuery()  .processInstanceBusinessKey("98k")  .singleResult(); // 3.完成任务并设置当前任务务的 Local 流程变量 Map<String, Object> variables = new HashMap<>(1); variables.put("holiday", new Holiday(3.5D)); // 参数1：任务ID // 参数2：当前任务的流程变量 // 参数3：如果为 true 设置为 当前任务 Local 变量，否则是整个流程实例的流程变量 taskService.complete(task.getId(), variables, true); // 4.查询当前任务的流程变量，由于任务已经完成，需要通过查询历史来获取 HistoryService historyService = ProcessEngines.getDefaultProcessEngine().getHistoryService(); List<HistoricTaskInstance> list = historyService.createHistoricTaskInstanceQuery()  .processInstanceBusinessKey("98k")  // 需要包含流程变量  .includeProcessVariables()  // 需要包含任务变量  .includeTaskLocalVariables()  .list(); for (HistoricTaskInstance historicTaskInstance : list) {  Map<String, Object> processVariables = historicTaskInstance.getProcessVariables();  Map<String, Object> taskLocalVariables = historicTaskInstance.getTaskLocalVariables();  System.err.println("任务名【" + historicTaskInstance.getName() + "】的 Global 变量：" + processVariables);  System.err.println("任务名【" + historicTaskInstance.getName() + "】的 Local 变量：" + taskLocalVariables); } // 5.查询下一个任务的流程变量 Task task2 = taskService.createTaskQuery()  .processInstanceBusinessKey("98k")  .singleResult(); Map<String, Object> variables2 = taskService.getVariables(task2.getId()); Map<String, Object> variablesLocal2 = taskService.getVariablesLocal(task2.getId()); System.err.println("variables2：" + variables2); System.err.println("variablesLocal2：" + variablesLocal2); |

注意：通过 HistoryService 查询历史任务时，如果需要查询流程实例变量，需要调用 includeProcessVariables() 方法，如果要查询任务变量，需要调用 includeTaskLocalVariables() 方法

**9.6.2 通过当前任务设置**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 | // 1.获取任务服务 TaskService taskService = ProcessEngines.getDefaultProcessEngine().getTaskService(); // 2.查询当前任务 Task task = taskService.createTaskQuery()  .processInstanceBusinessKey("98k")  .singleResult(); // 3.设置任务 Local 流程变量 Holiday holiday = new Holiday(4D); taskService.setVariableLocal(task.getId(), "holiday", holiday); // 4.查询任务的流程变量 Map<String, Object> variables = taskService.getVariables(task.getId()); System.err.println("variables：" + variables); Map<String, Object> variablesLocal = taskService.getVariablesLocal(task.getId()); System.err.println("variablesLocal：" + variablesLocal); // 5.完成任务 taskService.complete(task.getId()); // 6.获取下一个任务 Task task2 = taskService.createTaskQuery()  .processInstanceBusinessKey("98k")  .singleResult(); // 7.获取下一个任务的流程变量 Map<String, Object> variables2 = taskService.getVariables(task2.getId()); System.err.println("variables2：" + variables2); Map<String, Object> variablesLocal2 = taskService.getVariablesLocal(task2.getId()); System.err.println("variablesLocal2：" + variablesLocal2); |

注意：任务 ID 必须是当前代表任务 ID，在 act\_ru\_task 中存在

**10.组任务**

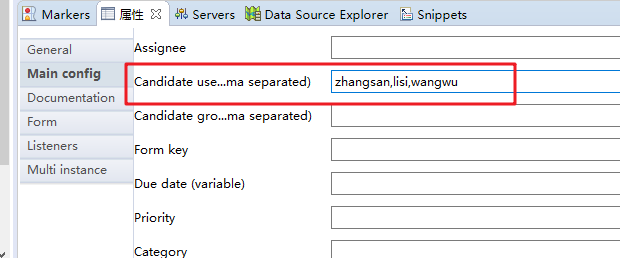
**10.1 CandidateUsers 候选人**

**10.1.1 需求**

在流程定义中在任务节点的 assignee 固定设置任务负责人，在流程定义时将参与者固定设置在 .bpmn 文件中，如果临时任务负责人变更则需要修改流程定义，系统可扩展性差。针对这总情况，可以给任务设置多个候选人，可以从候选人中选择参与者来完成任务

**10.1.2 设置任务候选人**

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

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565766382469.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565766382469.png" \o "1565766382469)

查看 bpmn 文件：

|  |  |
| --- | --- |
| 1 | <userTask id="usertask2" name="部门经理审批" activiti:candidateUsers="zhangsan,lisi,wangwu"></userTask> |

我们可以看到部门经理的候选人已经设置为 zhangsan,lishi,wangwu 这样的一组候选人，可以使用  
activiti:candiateUsers=”用户1,用户 2,用户 3”的这种方式来实现设置一组候选人

**10.1.3 组任务办理流程**

1. 查询组任务
   * 指定候选人，查询该候选人的当前待办任务，候选人不能办理任务
2. 拾取任务
   * 该组任务的所有候选人都能十七，将候选人的组任务变成个人任务，原来候选人就变成了该任务的负责人
   * 如果拾取后不想办理任务？需要将已经拾取的个人任务归还到组里面，将个人任务变成组任务
3. 查询个人任务
   * 根据 assignee 查询用户负责人的个人任务
4. 办理个人任务

**10.1.3.1 用户查询组任务**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 | // 2.根据候选ID 或者 候选人的组ID 查询任务 String candidateId = "lisi"; String candidateGroupId = "部门经理"; Task task = taskService.createTaskQuery()  .processInstanceBusinessKey("98k")  .taskCandidateUser(candidateId, Collections.singletonList(candidateGroupId))  .singleResult(); |

**10.1.3.2 用户拾取组任务**

|  |  |
| --- | --- |
| 1 2 | // 3.拾取任务 taskService.claim(task.getId(), candidateId); |

说明：即使该用户部署候选人也能拾取，建议拾取时校验是否有资格，任务组拾取后，该任务已经有负责人，通过候选查询不到该任务

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | select  distinct RES.\*  FROM  ACT\_RU\_TASK RES  INNER JOIN  ACT\_RU\_IDENTITYLINK I   ON I.TASK\_ID\_ = RES.ID\_  INNER JOIN  ACT\_RU\_EXECUTION E   ON RES.PROC\_INST\_ID\_ = E.ID\_  WHERE  E.BUSINESS\_KEY\_ = '98k'   and RES.ASSIGNEE\_ is null   and I.TYPE\_ = 'candidate'   and (  I.USER\_ID\_ = 'lisi'   or I.GROUP\_ID\_ IN (  '部门经理'   )   )  order by  RES.ID\_ asc LIMIT 2147483647 OFFSET 0; |

**10.1.3.3 用户查询个人待办任务和处理**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | TaskService taskService = ProcessEngines.getDefaultProcessEngine().getTaskService(); // 当前任务参与者ID String userId = "zhangsan"; Task task = taskService.createTaskQuery()  .taskAssignee(userId)  .processInstanceBusinessKey("98k")  .singleResult(); taskService.complete(task.getId()); System.err.println("任务ID：" + task.getId()); System.err.println("任务名字：" + task.getName()); System.err.println("任务参与者：" + task.getAssignee()); |

**10.1.3.4 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

|  |  |
| --- | --- |
| 1 2 | // 4.归还组任务 taskService.setAssignee("5002", 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）

**10.1.3.5 任务交接**

任务交接，任务负责人将任务交给其它候选人办理该任务

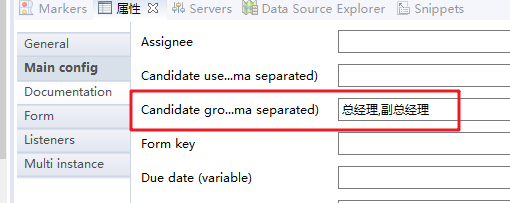
|  |  |
| --- | --- |
| 1 2 | // 5.任务交接 taskService.setAssignee("5002", "wangwu"); |

**10.1.4 操作的数据库表**

* act\_ru\_task：任务执行表，记录当前执行的任务，由于该任务是组任务，所以 assignee 为空，当拾取任务后该字段就是拾取用户的 ID
* act\_ru\_identitylink：任务参与者，记录当前参与任务用户或组，当前任务如果设置了候选人，就会向当前表插入候选人记录，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
* act\_hi\_actinst：历史活动表，update ASSIGNEE\_
* act\_hi\_taskinst：历史任务表，update ASSIGNEE\_

**10.2 CandidateGroups 候选人组**

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

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565766770413.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565766770413.png" \o "1565766770413)

查看 bpmn 文件：

|  |  |
| --- | --- |
| 1 | <userTask id="usertask3" name="总经理审批" activiti:candidateGroups="总经理,副总经理"></userTask> |

使用方法参考候选人

**11.网关**

**11.1 排他网关**

任务执行后的分支，经过排他网关分支只有一条有效

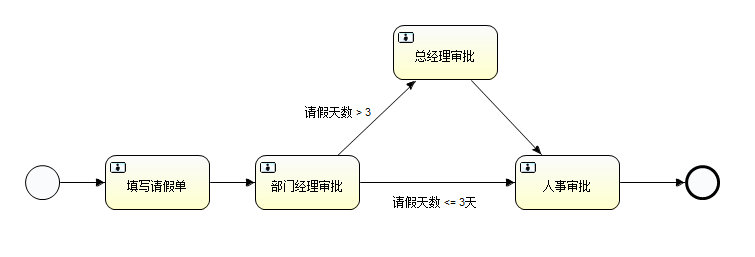
**11.1.1 什么是排他网关**

排他网关也叫异或（XOR）网络，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，即使有两个分支条件都为 true，排他网关只会选择一条 ID 较小的分支去执行**

为什么要用排他网关？

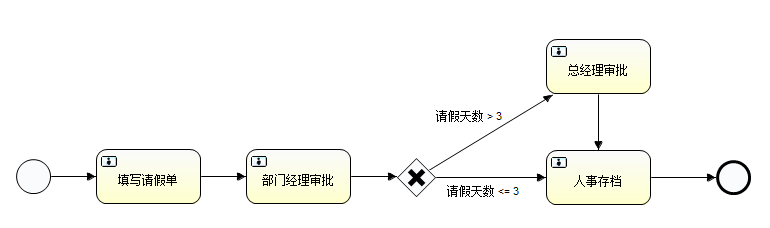
不用排他网关也可以实现分支，比如：

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565767036744.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565767036744.png" \o "1565767036744)

上图中，在连线的 condition 条件上设置分支条件

缺点：如果条件不满足，不使用排他网关，流程就结束了，是异常结束

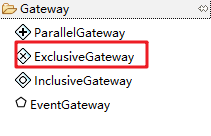
如果使用了排他网关决定分支的走向，如：

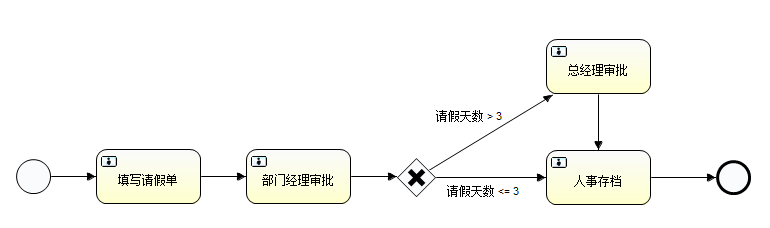
[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565767063911.png)](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565767063911.png)

如果从网关出去的线所有条件都不满足，则系统抛出异常

|  |  |
| --- | --- |
| 1 2 3 4 5 6 | 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.ExclusiveGatewayActivityBehav ior.leave(ExclusiveGatewayActivityBehavior.java:85) |

**11.1.2 流程定义及测试**

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565767103515.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565767103515.png" \o "1565767103515)

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565767063911.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565767063911.png" \o "1565767063911)

在部门经理审核后，走排他网关出来的两个分支，一个是判断请假天数大于3，另一条是小于等于3

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 | ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine(); // 1.部署 Deployment deploy = processEngine.getRepositoryService()  .createDeployment()  .name("排他网关")  .addClasspathResource("process/ExclusiveGateway.bpmn")  .deploy(); System.err.println(deploy.getId()); System.err.println(deploy.getName()); // 2.启动流程实例，并设置流程变量 RuntimeService runtimeService = processEngine.getRuntimeService(); ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("ExclusiveGateway", "ExclusiveGateway-98k"); runtimeService.setVariable(processInstance.getId(), "num", 3); System.err.println(processInstance.getId()); // 3.查询任务并处理任务 TaskService taskService = processEngine.getTaskService(); for (int i = 0; i < 4; i++) {  Task task = taskService.createTaskQuery()  .processInstanceBusinessKey("ExclusiveGateway-98k")  .singleResult();  if (task != null) {  taskService.complete(task.getId());  } } |

设置分支条件是，如果所有分支都不是true，则报错

|  |  |
| --- | --- |
| 1 | org.activiti.engine.ActivitiException: No outgoing sequence flow of the exclusive gateway 'exclusivegateway1' could be selected for continuing the process |

**11.2 并行网关**

任务执行后，可以多条分支，多条分支总会汇聚，汇聚完成，并行网关结束

**11.2.1 什么是并行网关**

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

* 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

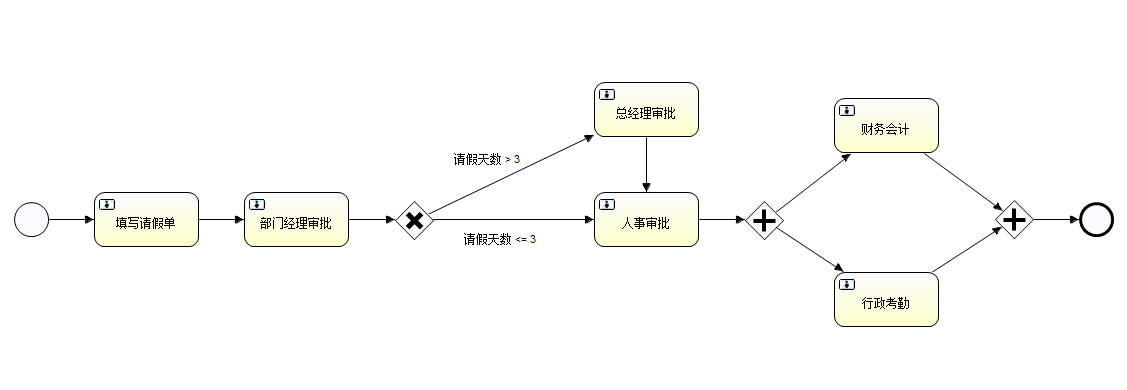
* join 汇聚：

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后，流程就行通过汇聚网关

**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，即使顺序流中定了条件，也会被忽略**

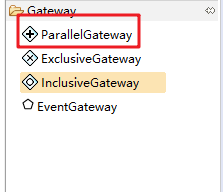
比如：

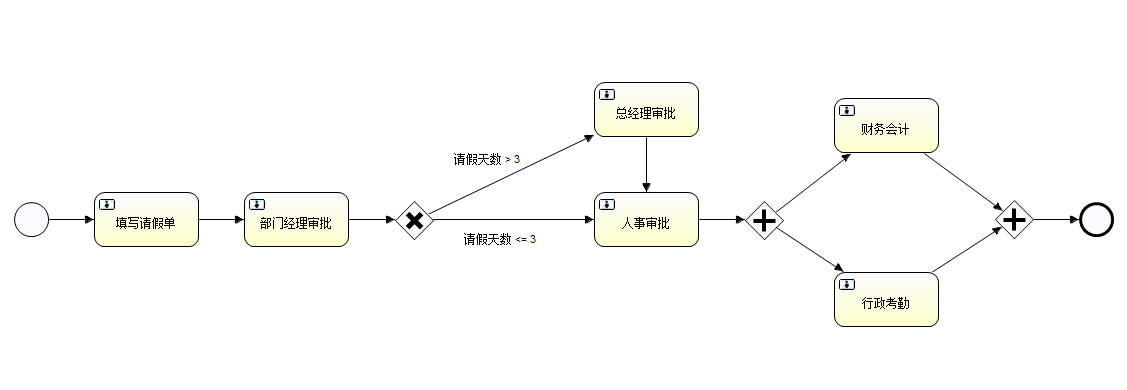
[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565767237121.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565767237121.png" \o "1565767237121)

说明：

财务会计和行政考勤是两个 execution 分支，在 act\_ru\_execution 表有两条记录分别是财务会计和行政考勤，act\_ru\_execution 还有一条记录表示该流程实例  
待财务会计和行政考勤任务全部完成，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

**11.2.2 流程定义及测试**

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565767222277.png)](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565767222277.png)

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565767237121.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565767237121.png" \o "1565767237121)

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 | ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine(); // 1.部署 Deployment deploy = processEngine.getRepositoryService()  .createDeployment()  .name("并行网关")  .addClasspathResource("process/ParallelGateway.bpmn")  .deploy(); System.err.println(deploy.getId()); System.err.println(deploy.getName()); // 2.启动流程实例，并设置流程变量 RuntimeService runtimeService = processEngine.getRuntimeService(); ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("ParallelGateway", "ParallelGateway-98k"); runtimeService.setVariable(processInstance.getId(), "num", 3); System.err.println(processInstance.getId()); // 3.查询任务并处理任务 TaskService taskService = processEngine.getTaskService(); for (int i = 0; i < 4; i++) {  List<Task> taskList = taskService.createTaskQuery()  .processInstanceBusinessKey("ParallelGateway-98k")  .list();  if (taskList != null && taskList.size() > 0) {  for (Task task : taskList) {  if (task != null) {  taskService.complete(task.getId());  }  }  }  } |

**11.3 包含网关**

是排他网关和并行网关的结合体

**11.3.1 什么是包含网关**

包含网关可以看作是排他网关和并行网关的结合体。和排他网关一样，可以在外出顺序流上定义条件，包含网关会解析他们，但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

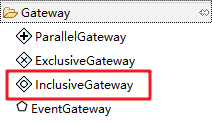
所有外出顺序流的条件都会被解析，结果为 true 的顺序流会以并行方式继续执行，会为每个顺序流创建一个分支

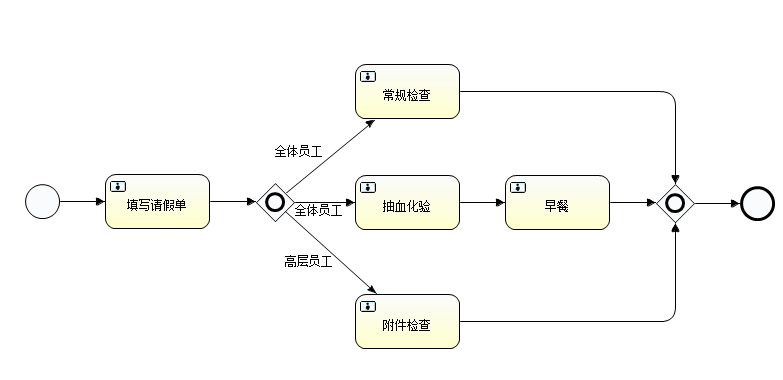
* 汇聚

所有并行分支到达包含网关，会进入等待状态，知道每个包含流程 token 的进入顺序流的分支都到达。这是于并行网关最大的不同。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

**11.3.2 流程定义及测试**

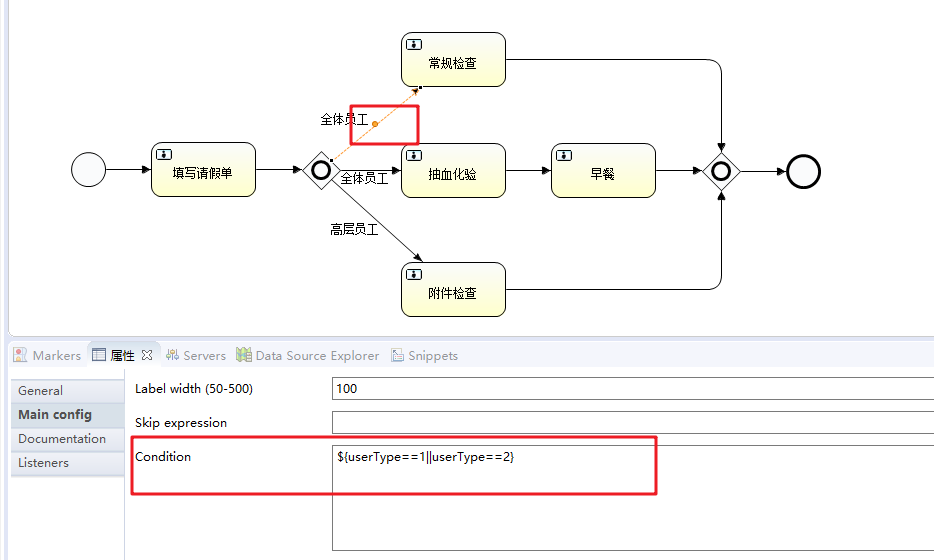
需求：企业体检流程，公司全体员工进行常规检查、抽血化验，公司管理层还可以增加体检项目

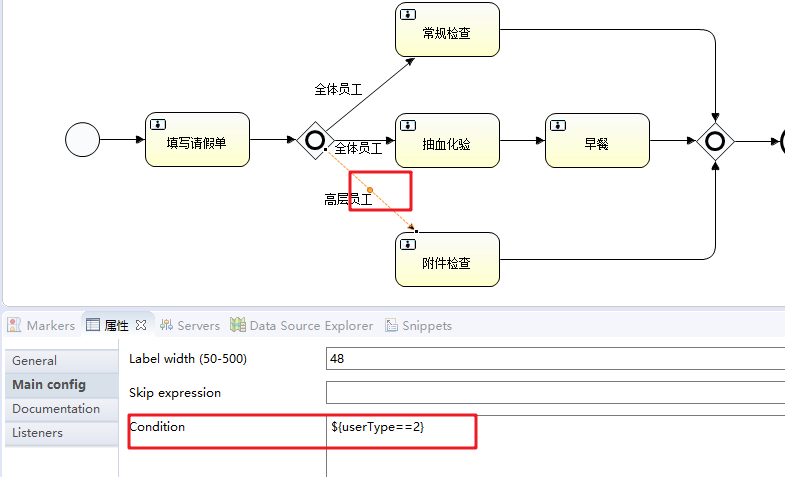
[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565767473355.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565767473355.png" \o "1565767473355)

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565767485781.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565767485781.png" \o "1565767485781)

通过流程变量 ${userType==1||userType==2} 来表示，1表示普通员工 2表示领导

注意：通过包含网关的每个分支的连接线上设置 condition 条件

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565767552904.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565767552904.png" \o "1565767552904)

[[](file:///C:\Users\Allen\Desktop\Activiti7%20学习笔记%20_%20Hello%20World_files\1565767570017.png)](file:///C:\\Users\\Allen\\Desktop\\Activiti7%20学习笔记%20_%20Hello%20World_files\\1565767570017.png" \o "1565767570017)

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 | ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine(); // 1.部署 Deployment deploy = processEngine.getRepositoryService()  .createDeployment()  .name("包含网关")  .addClasspathResource("process/InclusiveGateway.bpmn")  .deploy(); System.err.println(deploy.getId()); System.err.println(deploy.getName()); // 2.启动流程实例，并设置流程变量 RuntimeService runtimeService = processEngine.getRuntimeService(); ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("InclusiveGateway", "InclusiveGateway-98k"); runtimeService.setVariable(processInstance.getId(), "userType", 2); System.err.println(processInstance.getId()); // 3.查询任务并处理任务 TaskService taskService = processEngine.getTaskService(); for (int i = 0; i < 4; i++) {  List<Task> taskList = taskService.createTaskQuery()  .processInstanceBusinessKey("InclusiveGateway-98k")  .list();  if (taskList != null && taskList.size() > 0) {  for (Task task : taskList) {  if (task != null) {  taskService.complete(task.getId());  }  }  }  } |

如果包含网关设置的条件中，流程变量不存在，报错：

|  |  |
| --- | --- |
| 1 2 | org.activiti.engine.ActivitiException: Unknown property used in expression: ${userType=='1' || userType=='2'} |

需要在流程实例启动时设置流程变量 userType

**12.Activiti7 与 Springboot 整合**

环境：

* Springboot 2.1.7.RELEASE
* Activiti 7.1.0.M2
* Mabatis plus 3.1.2
* druid 1.1.13
* lombok 1.18.4
* swagger 2.7.0

pom.xml 文件：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 | <properties>  <java.version>1.8</java.version>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  <junit.version>4.12</junit.version>  <commons.lang.version>2.6</commons.lang.version>  <commons.fileupload.version>1.3.1</commons.fileupload.version>  <commons.io.version>2.5</commons.io.version>  <commons.codec.version>1.10</commons.codec.version>  <druid.version>1.1.13</druid.version>  <fastjson.version>1.2.45</fastjson.version>  <lombok.version>1.18.4</lombok.version>  <swagger.version>2.7.0</swagger.version>  <mybatisplus.version>3.1.2</mybatisplus.version>  <activiti.version>7.1.0.M2</activiti.version> </properties>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-freemarker</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-aop</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context-support</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-configuration-processor</artifactId>  <optional>true</optional>  </dependency>  <dependency>  <groupId>commons-lang</groupId>  <artifactId>commons-lang</artifactId>  <version>${commons.lang.version}</version>  </dependency>  <dependency>  <groupId>commons-fileupload</groupId>  <artifactId>commons-fileupload</artifactId>  <version>${commons.fileupload.version}</version>  </dependency>  <dependency>  <groupId>commons-io</groupId>  <artifactId>commons-io</artifactId>  <version>${commons.io.version}</version>  </dependency>  <dependency>  <groupId>commons-codec</groupId>  <artifactId>commons-codec</artifactId>  <version>${commons.codec.version}</version>  </dependency>  <dependency>  <groupId>com.baomidou</groupId>  <artifactId>mybatis-plus-boot-starter</artifactId>  <version>${mybatisplus.version}</version>  <exclusions>  <exclusion>  <groupId>com.baomidou</groupId>  <artifactId>mybatis-plus-generator</artifactId>  </exclusion>  </exclusions>  </dependency>  <dependency>  <groupId>org.postgresql</groupId>  <artifactId>postgresql</artifactId>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid-spring-boot-starter</artifactId>  <version>${druid.version}</version>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>fastjson</artifactId>  <version>${fastjson.version}</version>  </dependency>  <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId>  <version>${lombok.version}</version>  </dependency>  <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger2</artifactId>  <version>${swagger.version}</version>  </dependency>  <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger-ui</artifactId>  <version>${swagger.version}</version>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>${junit.version}</version>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <!-- activiti start -->  <dependency>  <groupId>org.activiti</groupId>  <artifactId>activiti-engine</artifactId>  <version>${activiti.version}</version>  <exclusions>  <exclusion>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  </exclusion>  </exclusions>  </dependency>  <dependency>  <groupId>org.activiti</groupId>  <artifactId>activiti-spring</artifactId>  <version>${activiti.version}</version>  </dependency>  <dependency>  <groupId>org.activiti</groupId>  <artifactId>activiti-bpmn-model</artifactId>  <version>${activiti.version}</version>  </dependency>  <dependency>  <groupId>org.activiti</groupId>  <artifactId>activiti-bpmn-converter</artifactId>  <version>${activiti.version}</version>  </dependency>  <dependency>  <groupId>org.activiti</groupId>  <artifactId>activiti-json-converter</artifactId>  <version>${activiti.version}</version>  </dependency>  <dependency>  <groupId>org.activiti</groupId>  <artifactId>activiti-bpmn-layout</artifactId>  <version>${activiti.version}</version>  </dependency>  <!-- activiti end -->  <!-- Activiti生成流程图 -->  <dependency>  <groupId>org.activiti</groupId>  <artifactId>activiti-image-generator</artifactId>  <version>${activiti.version}</version>  </dependency> </dependencies>  <build>  <finalName>${project.artifactId}</finalName>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  <!-- 跳过单元测试 -->  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-plugin</artifactId>  <configuration>  <skipTests>true</skipTests>  </configuration>  </plugin>  </plugins> </build> |

application.yml 文件：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 | # Tomcat server:  tomcat:  uri-encoding: UTF-8  max-threads: 1000  min-spare-threads: 30  port: 8080  servlet:  context-path: /  spring:  profiles:  active: dev  jackson:  date-format: yyyy-MM-dd HH:mm:ss  time-zone: GMT+8  servlet:  multipart:  max-file-size: 100MB  max-request-size: 100MB  enabled: true  freemarker:  suffix: .html  request-context-attribute: request  # mybatis mybatis-plus:  mapper-locations: classpath\*:/mapper/\*\*/\*.xml  # 实体扫描，多个package用逗号或者分号分隔  typeAliasesPackage: com.gitwh.activitispringboot.modules.\*.entity  # 枚举扫描  type-enums-package: com.gitwh.activitispringboot.modules.\*.enums  global-config:  # 数据库相关配置  db-config:  # 主键类型 AUTO：数据库自增，INPUT：自行输入，ID\_WORKER：64位分布式全局唯一ID 长整型类型，UUID：32位UUID字符串，NONE：无状态，ID\_WORKER\_STR：64位分布式全局唯一ID 字符串类型  id-type: UUID  # 字段策略 IGNORED：忽略判断，NOT\_NULL：非NULL判断，NOT\_EMPTY：非空判断(只对字符串类型字段,其他类型字段依然为非NULL判断)，DEFAULT：追随全局配置  insert-strategy: NOT\_NULL  update-strategy: NOT\_NULL  select-strategy: NOT\_NULL  # 逻辑已删除值  logic-delete-value: -1  # 逻辑未删除值  logic-not-delete-value: 0  banner: false  # MyBatis原生配置  configuration:  map-underscore-to-camel-case: true  cache-enabled: false  call-setters-on-nulls: true  jdbc-type-for-null: 'null'  # 配置默认枚举类型转换器  default-enum-type-handler: org.apache.ibatis.type.EnumOrdinalTypeHandler |

application-dev.yml 文件

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 | spring:  # 数据库配置  datasource:  type: com.alibaba.druid.pool.DruidDataSource  druid:  driverClassName: org.postgresql.Driver  url: jdbc:postgresql://localhost:5432/activiti\_springboot  username: postgres  password: postgres  initial-size: 10  max-active: 100  min-idle: 10  max-wait: 60000  pool-prepared-statements: true  max-pool-prepared-statement-per-connection-size: 20  time-between-eviction-runs-millis: 60000  min-evictable-idle-time-millis: 300000  # Oracle需要打开注释  #validation-query: SELECT 1 FROM DUAL  test-while-idle: true  test-on-borrow: false  test-on-return: false  stat-view-servlet:  enabled: true  url-pattern: /druid/\*  #login-username: admin  #login-password: admin  filter:  stat:  log-slow-sql: true  slow-sql-millis: 1000  merge-sql: false  wall:  config:  multi-statement-allow: true  # 打印 MyBatis 日志 mybatis-plus:  configuration:  log-impl: org.apache.ibatis.logging.stdout.StdOutImpl |

Activiti 配置类

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 | @Configuration public class ActivitiConfig {   @Bean  public ProcessEngineConfiguration processEngineConfiguration(DataSource dataSource, PlatformTransactionManager transactionManager) {  SpringProcessEngineConfiguration processEngineConfiguration = new SpringProcessEngineConfiguration();  processEngineConfiguration.setDataSource(dataSource);  // 表不存在创建表  processEngineConfiguration.setDatabaseSchemaUpdate("true");  // 指定数据库  processEngineConfiguration.setTransactionManager(transactionManager);  // 历史变量  processEngineConfiguration.setHistory("full");  return processEngineConfiguration;  }   @Bean  public ProcessEngineFactoryBean processEngine(ProcessEngineConfiguration processEngineConfiguration) {  ProcessEngineFactoryBean processEngineFactoryBean = new ProcessEngineFactoryBean();  processEngineFactoryBean.setProcessEngineConfiguration((ProcessEngineConfigurationImpl) processEngineConfiguration);  return processEngineFactoryBean;  }   @Bean  public RepositoryService repositoryService(ProcessEngine processEngine) {  return processEngine.getRepositoryService();  }   @Bean  public RuntimeService runtimeService(ProcessEngine processEngine) {  return processEngine.getRuntimeService();  }   @Bean  public TaskService taskService(ProcessEngine processEngine) {  return processEngine.getTaskService();  }   @Bean  public HistoryService historyService(ProcessEngine processEngine) {  return processEngine.getHistoryService();  }   @Bean  public ManagementService managementService(ProcessEngine processEngine) {  return processEngine.getManagementService();  }   @Bean  public DynamicBpmnService dynamicBpmnService(ProcessEngine processEngine) {  return processEngine.getDynamicBpmnService();  }   @Bean  public ProcessDiagramGenerator processDiagramGenerator() {  return new DefaultProcessDiagramGenerator();  }  } |