[**SpringBoot Activiti6系列教程(一)-activiti-app部署**](https://segmentfault.com/a/1190000020286897)

[[https://avatar-static.segmentfault.com/868/271/868271510-54cb382abb7a1_small](https://segmentfault.com/t/java) java](https://segmentfault.com/t/java)[bpmn](https://segmentfault.com/t/bpmn)[activiti](https://segmentfault.com/t/activiti)

 阅读约 6 分钟

**背景**

在开源的工作流引擎中，[activiti](https://www.activiti.org/)和[jbpm](https://www.jbpm.org/)目前是唯二的两个选择，相比jbpm，activiti应该算是晚辈，但有意思的是，在github上，activiti有5.6k的start，而jbpm却只有978，可见开发者们更喜欢较为年轻的activiti，实际情况也是这样的，近几年在企业里面，activiti大有燎原之势，做流程的没有不知道activiti。看了网上activiti的教程，质量参差不齐，找不到一个完整的教程，想看个视频，动不动几十集的，因此打算自己整理一套完整的activiti教程，本套教程基于activiti6。

**实现**

**下载**

可以从[这里](https://www.activiti.org/get-started)下载Activiti6资源包的zip文件，在这个页面还包含[开发文档](https://www.activiti.org/userguide)和[JavaDoc](https://www.activiti.org/javadocs)，官方文档写的非常详细，而且篇幅不长，有条件的同学建议能够通读一遍。

**安装**

下载完的zip包名称为activiti-6.0.0.zip，解压开文件如下:

.

├── activiti\_readme.png

├── database

│   ├── create

│   ├── drop

│   └── upgrade

├── libs

│   ├── activiti-app-logic-6.0.0-sources.jar

│   ├── activiti-app-logic-6.0.0.jar

│   ├── activiti-app-rest-6.0.0-sources.jar

│   ├── activiti-app-rest-6.0.0.jar

│   ├── ......

├── license.txt

├── notice.txt

├── readme.html

└── wars

├── activiti-admin.war

├── activiti-app.war

└── activiti-rest.war

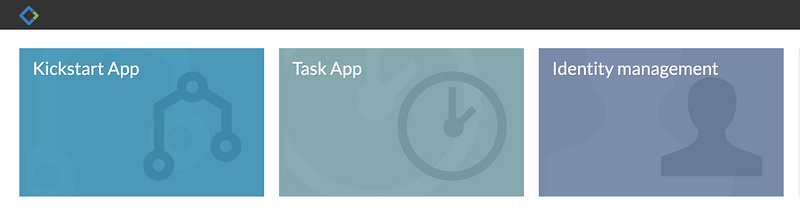
* database存放数据库对象相关脚本，包含不同的数据库脚本
* libs包含activiti开发过程中需要用到的jar包和源码，不建议通过jar包直接引用，建议通过maven进行管理
* wars包含三个app，activiti-app提供了一个流程的建模工具，支持流程模型的导出和发布到activi系统，也提供了一个相对简单到流程审批界面和流程发起界面，但完成度不高。
* activiti-admin提供管理员功能，可以查看当前流程实例运行情况。
* activiti-rest提供activiti 所有rest api接口

将wars文件夹下三个war文件复制到tomcat的webapps目录下，启动tomcat即可自动部署，首次部署需要初始化数据库，时间可能比较长，默认用的是自带的内存数据库h2，我们也会介绍如何修改为我们自己的数据库。

**activiti-app**

这个应用包含流程建模和用户管理相关功能，activiti-app提供了一个bpmn的建模工具，可以将模型导出或者部署到activiti系统中。

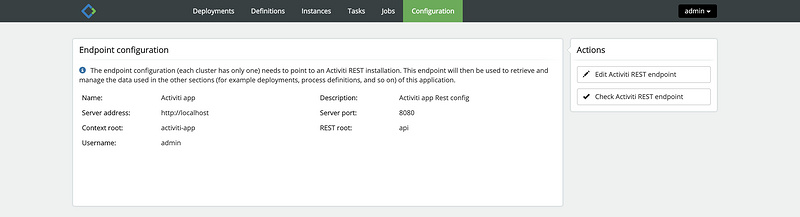
访问地址为<http://localhost:8001/activiti-app>，默认账号为 admin/test，默认有三个app



* Kickstart app包含四个功能，流程模型设计，表单设计，决策表设计，app管理
* Task App提供一个简单到待办审批界面和流程发起界面。
* Identity management用来管理用户和用户组

**activity-admin**

activity-admin应用提供一些基本的流程运行时管理功能，可以查看当前运行的所有实例和任务，也可以上传流程文件



**activity-rest**

activity-rest提供一些activiti rest api功能，不过需要通过 http basic认证进行调用。

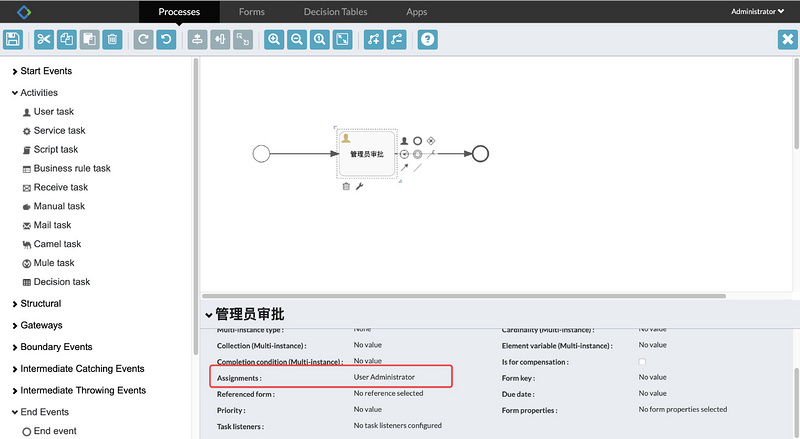
三个应用如果需要连通，还需要做相关的配置，下一章节会介绍如何将三个应用连通。

**流程模型创建**

activiti支持BPMN 2.0的标准，所以理论上任何BPMN 2.0编辑器创建的模型都支持，不管是本地版本的还是在线版，在介绍activiti自带的建模工具之前，介绍几个可供选择的建模工具

* bpmn官放提供的开源建模工具[bpmn-js](https://demo.bpmn.io/)，支持基本的模型设计，但缺少属性配置，不过可能是故意这样设计的，这个工具主要是给activiti开发者进行二次开发。
* [yaoqiang](http://www.yaoqiang.org/)用java开发的bpmn设计器，由于是java开发，所以支持跨平台。
* actiBPM，idea插件。

登录[activiti-app](http://localhost:8001/activiti-app)，进入Kickstar App->CreateProcess创建新的流程模型，需要输入模型名称和模型主键，这个主键必须唯一，代码中发起流程需要用到。这里创建了只有一个UserTask的审批节点



可以修改UserTask的assignments属性设置审批人，这里设置了管理员作为审批人。保存后退出编辑器界面，在查看页面可以对模型进行导出为bpmn文件，也可以通过创建一个App并且将选择流程模型，将app进行发布(Publish)后就可以在Task App中进行流程的发起，也可以通过程序进行发起。

**总结**

本章节主要介绍如何部署activit6的三个应用，相对来说还是比较简单的，对三个app的详细应用在接下来的章节也会继续介绍。

[**SpringBoot Activiti6系列教程(二)-基于mysql数据库初始化**](https://segmentfault.com/a/1190000020288808)

[activiti](https://segmentfault.com/t/activiti)[bpmn](https://segmentfault.com/t/bpmn)[[https://avatar-static.segmentfault.com/868/271/868271510-54cb382abb7a1_small](https://segmentfault.com/t/java) java](https://segmentfault.com/t/java)

 阅读约 12 分钟

**数据库初始化**

activiti默认采用内存数据库h2，作为本地测试是够了，但是作为测试环境，开发环境和生产环境，是远远不够的，我们需要使用更为强大和灵活的数据库，以下是zip包里提供的数据库创建脚本

activiti.db2.create.engine.sql

activiti.db2.create.history.sql

activiti.db2.create.identity.sql

activiti.h2.create.engine.sql

activiti.h2.create.history.sql

activiti.h2.create.identity.sql

activiti.hsql.create.engine.sql

activiti.hsql.create.history.sql

activiti.hsql.create.identity.sql

activiti.mssql.create.engine.sql

activiti.mssql.create.history.sql

activiti.mssql.create.identity.sql

activiti.mysql.create.engine.sql

activiti.mysql.create.history.sql

activiti.mysql.create.identity.sql

activiti.mysql55.create.engine.sql

activiti.mysql55.create.history.sql

activiti.oracle.create.engine.sql

activiti.oracle.create.history.sql

activiti.oracle.create.identity.sql

activiti.postgres.create.engine.sql

activiti.postgres.create.history.sql

activiti.postgres.create.identity.sql

从脚本上看，activiti支持以下几种数据库

* db2
* h2
* hsqldb
* sql server
* mysql
* oracle
* postgres

本次以mysql为例。

activiti提供两种方式进行数据库初始化，一种是通过脚本，一种是在libs/activiti-engine-6.0.0.jar中提供了DbSchemaCreate类用来创建数据库。一般开发人员无法拿到生产环境的数据库账号密码那么就需要提供脚本给dba执行，但如果有数据库的用户名密码，通过第二种更为方便。

不管通过哪种方式，都需要提前创建mysql数据库，以下就是创建数据库脚本参考(基于mysql 8.0.15)

**CREATE** **DATABASE** activiti **default** **charset** utf8 **COLLATE** utf8\_general\_ci;

**CREATE** **USER** 'activiti'@'%' **IDENTIFIED** **BY** 'definesys';

**GRANT** ALL **PRIVILEGES** **ON** activiti.\* **TO** 'activiti'@'%';

**ALTER** **USER** 'activiti'@'%' **IDENTIFIED** **WITH** mysql\_native\_password;

**alter** **user** 'activiti'@'%' **identified** **by** 'definesys';

**FLUSH** **PRIVILEGES**;

**sql脚本方式初始化**

如果是通过脚本方式，将以下四个脚本依次执行即可

activiti.mysql.create.engine.sql

activiti.mysql.create.history.sql

activiti.mysql.create.identity.sql

如果mysql的版本低于5.6.4的话，需要执行以下几个脚本，原因是由于5.6.4以下的mysql版本不支持timestamps类型。

activiti.mysql55.create.engine.sql

activiti.mysql55.create.history.sql

activiti.mysql.create.identity.sql

**DbSchemaCreate方式初始化**

* 创建maven项目，引入依赖包

<dependency>

<groupId>org.activiti</groupId>

<artifactId>activiti-engine</artifactId>

<version>6.0.0</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

* 在src/resources文件夹下创建activiti.cfg.xml文件，文件名不可以更改，参考下面配置修改

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

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

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.xsd">

<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">

<property name="databaseType" value="mysql"></property>

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

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

<property name="jdbcUsername" value="activiti"></property>

<property name="jdbcPassword" value="definesys"></property>

</bean>

</beans>

* 创建一个包含main方法的java类

**public** **class** **DBCreate** {

**public** **static** **void** **main**(String[] cmd) {

DbSchemaCreate.main(cmd);

}

}

* 运行程序DBCreate

如果执行成功，将会有28张表被创建，除了ACT\_EVT\_LOG表外，其他的都是业务表。

ACT\_EVT\_LOG

ACT\_GE\_BYTEARRAY

ACT\_GE\_PROPERTY

ACT\_HI\_ACTINST

ACT\_HI\_ATTACHMENT

ACT\_HI\_COMMENT

ACT\_HI\_DETAIL

ACT\_HI\_IDENTITYLINK

ACT\_HI\_PROCINST

ACT\_HI\_TASKINST

ACT\_HI\_VARINST

ACT\_ID\_GROUP

ACT\_ID\_INFO

ACT\_ID\_MEMBERSHIP

ACT\_ID\_USER

ACT\_PROCDEF\_INFO

ACT\_RE\_DEPLOYMENT

ACT\_RE\_MODEL

ACT\_RE\_PROCDEF

ACT\_RU\_DEADLETTER\_JOB

ACT\_RU\_EVENT\_SUBSCR

ACT\_RU\_EXECUTION

ACT\_RU\_IDENTITYLINK

ACT\_RU\_JOB

ACT\_RU\_SUSPENDED\_JOB

ACT\_RU\_TASK

ACT\_RU\_TIMER\_JOB

ACT\_RU\_VARIABLE

**数据库表说明**

所有表都是以ACT开头，并且两个字母表示模块。

* ACT\_RE\_\*: RE表示repository（仓库），存储流程静态资源，如流程模型文件等
* ACT\_RU\_\*: RU表示runtime（运行时），存储activiti运行时产生的数据，比如实例信息，用户任务信息，job信息等，另外，当流程结束后，运行时数据将会被删除，以保证数据量尽可能少，保证性能。
* ACT\_ID\_\*: ID表示identity（认证），存储认证信息，比如用户，组等。
* ACT\_HI\_\*: HI表示history（历史），存储流程历史数据，比如实例信息，变量数据等。
* ACT\_GE\_\*: GE表示general（通用），存储通用数据。

**应用信息修改**

现在，我们创建好了所有的表，可以替换掉[上一章节](https://segmentfault.com/a/1190000020286897)中部署的三个应用的数据库，下面介绍如何替换默认的数据库。

**activiti-app**

tomcat中，如果已经启动过一次应用，tomcat会在webapps文件夹下创建一个和war包名称相同的文件夹，并且将所有war包中的资源解压到该文件夹中，因为只需修改解压后文件夹中的内容即可。如果还未启动，可以用类似7-zip的解压软件进行修改。

修改文件activiti-app/WEB-INF/classes/META-INF/activiti-app/activiti-app.properties文件，默认配置的是h2数据库的连接，将之修改为目标数据库（这里以mysql为例）

datasource.driver=com.mysql.jdbc.Driver

datasource.url=jdbc:mysql:*//activitidb:3306/activiti?useUnicode=true&characterEncoding=utf8&useSSL=false*

datasource.username=activiti

datasource.password=definesys

hibernate.dialect=org.hibernate.dialect.MySQLDialect

应用自带的驱动为 mysql-connector-java-5.1.30.jar，如果驱动版本不对，只需将该文件删除，将指定的驱动拷贝至目录WEB-INF/lib目录下即可，其他数据库驱动操作一样。

**activiti-admin**

activiti-admin是通过rest接口获取数据，因为不需要配置数据库信息，只需配置rest接口的地址即可。

修改文件activiti-admin/WEB-INF/classes/META-INF/activiti-admin/activiti-admin.properties文件，修改如下配置

rest.app.name=Activiti app

rest.app.description=Activiti app Rest config

rest.app.host=http:*//localhost*

rest.app.port=8080

rest.app.contextroot=activiti-app

rest.app.restroot=api

rest.app.user=admin

rest.app.password=test

rest.app.port端口默认是9999，需要修改为部署activiti-app的tomcat的端口号，其余的如果没有做修改保持默认即可

**activiti-rest**

修改文件activiti-rest/WEB-INF/classes/db.properties文件

db=mysql

jdbc.driver=com.mysql.jdbc.Driver

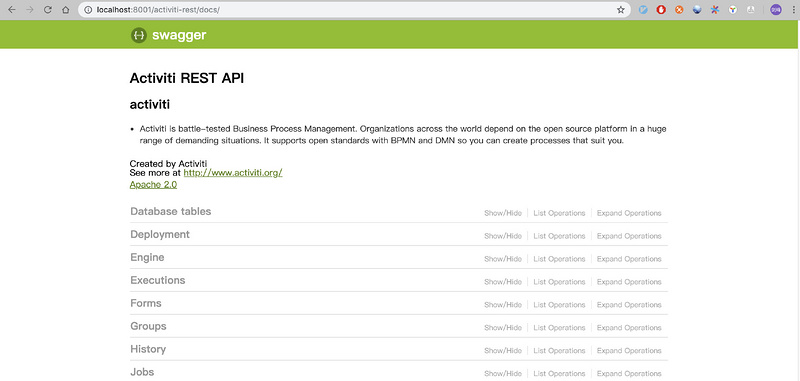
jdbc.url=jdbc:mysql://activitidb:3306/activiti

jdbc.username=activiti

jdbc.password=definesys

同样，可以将驱动拷贝至目录WEB-INF/lib目录下

成功启动后，访问<http://localhost:8080/activiti-rest/docs/>将打开activiti swagger api文档，包含所有的rest接口。



**总结**

本文主要介绍了如何使用自定义的数据库，以mysql为例，其他数据库操作方式跟mysql类似，就需要读者自行研究。

[**SpringBoot Activiti6系列教程(三)-开发一个简单的SpringBoot activiti6应用程序**](https://segmentfault.com/a/1190000020298085)

[[https://avatar-static.segmentfault.com/868/271/868271510-54cb382abb7a1_small](https://segmentfault.com/t/java) java](https://segmentfault.com/t/java)[bpmn](https://segmentfault.com/t/bpmn)[activiti](https://segmentfault.com/t/activiti)

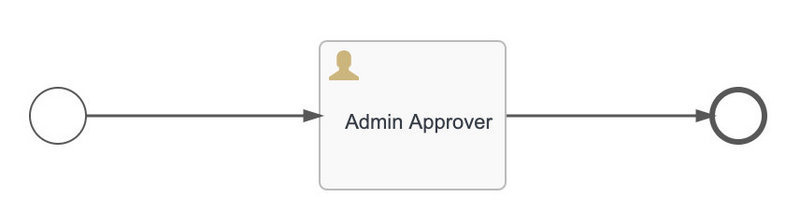
 阅读约 22 分钟

**BPMN建模**

在前面两节，我们介绍了如何部署activiti三个应用以及如何使用第三方数据库，如果你还没阅读前两章也不影响本文的阅读，如果有兴趣了解下，可以点击以下链接

* [SpringBoot Activiti6系列教程(一)-activiti-app部署](https://segmentfault.com/a/1190000020286897)
* [SpringBoot Activiti6系列教程(二)-基于mysql数据库初始化](https://segmentfault.com/a/1190000020288808)

从本章开始，就正式开始activiti程序开发，我们先从一个最简单的activiti流程开发，流程图如下：



流程很简单，发起后，由管理员admin审批，然后结束。

bpmn源文件代码如下

**<?**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:xsd="http://www.w3.org/2001/XMLSchema" 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/processdef">

<process id="SimpleProcess" name="SimpleProcess" isExecutable="true">

<startEvent id="startEvent1"></startEvent>

<userTask id="sid-2B2AA039-82D2-479C-8FE9-6F48E4478BD8" name="Admin Approver" activiti:assignee="admin">

<extensionElements>

<modeler:activiti-idm-assignee xmlns:modeler="http://activiti.com/modeler"><![CDATA[true]]></modeler:activiti-idm-assignee>

<modeler:assignee-info-email xmlns:modeler="http://activiti.com/modeler"><![CDATA[admin]]></modeler:assignee-info-email>

<modeler:assignee-info-lastname xmlns:modeler="http://activiti.com/modeler"><![CDATA[Administrator]]></modeler:assignee-info-lastname>

<modeler:initiator-can-complete xmlns:modeler="http://activiti.com/modeler"><![CDATA[false]]></modeler:initiator-can-complete>

</extensionElements>

</userTask>

<sequenceFlow id="sid-F506EEC7-0411-4A97-930D-87500FF1414F" sourceRef="startEvent1" targetRef="sid-2B2AA039-82D2-479C-8FE9-6F48E4478BD8"></sequenceFlow>

<endEvent id="sid-2C31753A-3AE2-4C91-B4F8-3B5EED2B4FC5"></endEvent>

<sequenceFlow id="sid-9FC989EE-BBC3-4EB2-A7AF-DA4F2649A68F" sourceRef="sid-2B2AA039-82D2-479C-8FE9-6F48E4478BD8" targetRef="sid-2C31753A-3AE2-4C91-B4F8-3B5EED2B4FC5"></sequenceFlow>

</process>

<bpmndi:BPMNDiagram id="BPMNDiagram\_SimpleProcess">

<bpmndi:BPMNPlane bpmnElement="SimpleProcess" id="BPMNPlane\_SimpleProcess">

<bpmndi:BPMNShape bpmnElement="startEvent1" id="BPMNShape\_startEvent1">

<omgdc:Bounds height="30.0" width="30.0" x="100.0" y="163.0"></omgdc:Bounds>

</bpmndi:BPMNShape>

<bpmndi:BPMNShape bpmnElement="sid-2B2AA039-82D2-479C-8FE9-6F48E4478BD8" id="BPMNShape\_sid-2B2AA039-82D2-479C-8FE9-6F48E4478BD8">

<omgdc:Bounds height="80.0" width="100.0" x="255.0" y="138.0"></omgdc:Bounds>

</bpmndi:BPMNShape>

<bpmndi:BPMNShape bpmnElement="sid-2C31753A-3AE2-4C91-B4F8-3B5EED2B4FC5" id="BPMNShape\_sid-2C31753A-3AE2-4C91-B4F8-3B5EED2B4FC5">

<omgdc:Bounds height="28.0" width="28.0" x="465.0" y="164.0"></omgdc:Bounds>

</bpmndi:BPMNShape>

<bpmndi:BPMNEdge bpmnElement="sid-F506EEC7-0411-4A97-930D-87500FF1414F" id="BPMNEdge\_sid-F506EEC7-0411-4A97-930D-87500FF1414F">

<omgdi:waypoint x="130.0" y="178.0"></omgdi:waypoint>

<omgdi:waypoint x="255.0" y="178.0"></omgdi:waypoint>

</bpmndi:BPMNEdge>

<bpmndi:BPMNEdge bpmnElement="sid-9FC989EE-BBC3-4EB2-A7AF-DA4F2649A68F" id="BPMNEdge\_sid-9FC989EE-BBC3-4EB2-A7AF-DA4F2649A68F">

<omgdi:waypoint x="355.0" y="178.0"></omgdi:waypoint>

<omgdi:waypoint x="465.0" y="178.0"></omgdi:waypoint>

</bpmndi:BPMNEdge>

</bpmndi:BPMNPlane>

</bpmndi:BPMNDiagram>

</definitions>

你可以使用任何一个支持BPMN2.0的工具进行流程的建模，如果你是用eclipse开发，activiti在eclipse下有一个非常好用的插件，并且activit也提供一个web版的建模工具，就在activiti-app这个应用下，关于如何部署该应用，你可以查看之前的文章[SpringBoot Activiti6系列教程(一)-activiti-app部署](https://segmentfault.com/a/1190000020286897)。以上模型就是通过activiti-app创建导出的。

**SpringBoot 开发**

**SpringBoot 介绍**

[Spring Boot](https://spring.io/projects/spring-boot/)是一个应用开发框架，基于spring，相比spring开发，spring boot极大简化了配置，并且遵守[约定优于配置](https://facilethings.com/blog/en/convention-over-configuration)的原则即使0配置也能正常运行，这在spring中是难以想象的。spring boot应用程序可以独立运行，框架内嵌web容器，使得web应用程序可以像本地程序一样启动和调试，十分的方便，这种设计方式也使得spring boot应用程序非常适合容器化进行大规模部署，而且spring boot提供了非常丰富的组件，流行的java web框架基本都有spring boot版本，生态十分庞大，是目前java web开发最好的方案。

**创建spring boot的工程**

如果你已经对spring boot非常熟悉，可以跳过该章节。  
本次教程代码都是通过IntelliJ IDEA进行开发，基于Spring Boot 2.1.2版本,如果你还没用过IntelliJ IDEA，建议你尝试下。

* **1.** 创建一个maven工程
* **2.** 设置项目parent并且加入spring boot的依赖和activiti相关依赖，参考pom.xml如下

***pom.xml***

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

<project xmlns="http://maven.apache.org/POM/4.0.0"

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

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.definesys.tutorial.activiti</groupId>

<artifactId>activiti-tutorial</artifactId>

<packaging>pom</packaging>

<version>1.0-SNAPSHOT</version>

<modules>

<module>activiti-tutorial1</module>

</modules>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.2.RELEASE</version>

<relativePath/>

</parent>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.activiti</groupId>

<artifactId>activiti-spring-boot-starter-basic</artifactId>

<version>6.0.0</version>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<version>1.4.185</version>

</dependency>

</dependencies>

</project>

spring-boot-starter-web是spring boot用来创建rest服务的组件，h2是一个内存数据库，如果你没有配置数据库，那么加入h2后，无需配置第三方数据库即可运行activit应用。

* **3.** 在resource文件夹下创建文件夹processes（文件夹名字不能修改），并且将第一步创建的模型文件拷贝至该文件下。
* **4.** 创建两个接口，/acitiviti/start可以发起该流程，/activiti/task?uid=admin可以查询admin的流程待办，依次创建appliaction，dto，service，controller相关代码文件，如下

本文所有代码已上传至github，仓库地址为 <https://github.com/wls1036/springboot-activiti6-tutorial> 欢迎star

***ActivititiApplication.java***

**@SpringBootApplication**(exclude = SecurityAutoConfiguration.class)

**public** **class** **ActivititiApplication** {

**public** **static** **void** **main**(String[] args) {

SpringApplication.run(ActivititiApplication.class);

}

}

之所以要加exclude = SecurityAutoConfiguration.class，原因是不加的话启动会报Caused by: java.lang.ArrayStoreException: sun.reflect.annotation.TypeNotPresentExceptionProxy错误，可能是spring boot的版本原因引起的。

***TaskRepresentation.java***

**public** **class** **TaskRepresentation** {

**private** String id;

**private** String name;

**public** **TaskRepresentation**(String id, String name) {

**this**.id = id;

**this**.name = name;

}

**public** String **getId**() {

**return** id;

}

**public** **void** **setId**(String id) {

**this**.id = id;

}

**public** String **getName**() {

**return** name;

}

**public** **void** **setName**(String name) {

**this**.name = name;

}

}

***ActivitiService.java***

**@Service**

**public** **class** **ActivitiService** {

**@Autowired**

**private** RuntimeService runtimeService;

**@Autowired**

**private** TaskService taskService;

*/\*\**

*\* start activiti process*

*\**

*\* @return instance id*

*\*/*

**public** String **start**() {

ProcessInstance instance = runtimeService.startProcessInstanceByKey("SimpleProcess");

**return** instance.getId();

}

*/\*\**

*\* get user task list*

*\**

*\* @param uid*

*\* @return user task list*

*\*/*

**public** List<Task> **getTask**(String uid) {

List<Task> tasks = taskService.createTaskQuery().taskAssignee(uid).list();

**return** tasks;

}

}

RuntimeService包含了activiti运行时接口，可以通过点击进入查看源文件，acitivi api设计的相当友好，很多接口看名称就差不多知道功能，startProcessInstanceByKey是根据流程编号发起流程，参数为bpmn文件中 process id="SimpleProcess"中的id字段。

TaskService包含任务相关api，createTaskQuery可以创建一个任务查询，activiti的api使用方法基本都一样，比如查询类的，都是createXXXQuery创建一个查询类后执行查询条件，list()是返回一个list数组。这里是指定审批人，在bpmn文件中，我们指定了审批人为admin

<userTask id="sid-2B2AA039-82D2-479C-8FE9-6F48E4478BD8" name="Admin Approver" activiti:assignee="admin">

***ActivitiController***

**@RestController**

**@RequestMapping**(value = "activiti")

**public** **class** **ActivitiController** {

**@Autowired**

**private** ActivitiService service;

**@RequestMapping**(value = "start", method = RequestMethod.GET)

**public** String **start**() {

**return** service.start();

}

**@RequestMapping**(value = "task", method = RequestMethod.GET)

**public** List<TaskRepresentation> **getTask**(@RequestParam(value = "uid") String uid) {

List<Task> tasks = service.getTask(uid);

List<TaskRepresentation> dtos = **new** ArrayList<>();

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

dtos.add(**new** TaskRepresentation(task.getId(), task.getName()));

}

**return** dtos;

}

}

**应用测试**

* 流程发起测试

curl http://localhost:8080/activiti/start

65

* 获取待办测试

curl http://localhost:8080/activiti/task?uid=admin

[{"id":"65","name":"Admin Approver"}]

**更改数据源**

现在的数据都保存到内存数据库h2中，程序重新启动就会丢失，这显然不是我们想要的结果，我们需要将数据保存到外部数据库，关于如何初始化外部数据库，可以参考之前的文章[SpringBoot Activiti6系列教程(二)-基于mysql数据库初始化](https://segmentfault.com/a/1190000020288808)，这里不在赘述。假设你已经初始化完数据库。

* 新建一个数据源配置类

**@Configuration**

**public** **class** **ActivitiConfiguration** {

**@Bean**

**public** DataSource **database**() {

**return** DataSourceBuilder.create()

.url("jdbc:mysql://localhost:3306/activiti?useUnicode=true&characterEncoding=utf8")

.username("activiti")

.password("definesys")

.driverClassName("com.mysql.jdbc.Driver")

.build();

}

}

这里为了简单把连接信息写死在代码里，不是一种好的方式，你可以将连接信息放到application.properties里。

* 修改pom文件去掉h2的依赖增加mysql驱动

....

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.activiti</groupId>

<artifactId>activiti-spring-boot-starter-basic</artifactId>

<version>6.0.0</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

</dependencies>

* 重新运行程序

重新调用start接口后，流程发起后，会在数据库表ACT\_RU\_TASK中创建记录。

**总结**

本文介绍了一个最简单的activiti流程在spring boot中的应用。

本文所有代码已经上传至github，仓库地址为<https://github.com/wls1036/springboot-activiti6-tutorial>欢迎star

[**SpringBoot Activiti6系列教程(四)-流程部署**](https://segmentfault.com/a/1190000020439236)

[bpmn](https://segmentfault.com/t/bpmn)[springboot](https://segmentfault.com/t/springboot)[[https://avatar-static.segmentfault.com/868/271/868271510-54cb382abb7a1_small](https://segmentfault.com/t/java) java](https://segmentfault.com/t/java)[activiti](https://segmentfault.com/t/activiti)

 阅读约 10 分钟

**说明**

在[上一章节](https://segmentfault.com/a/1190000020298085)中，介绍了如何基于bpmn2.0的xml文件发起流程和获取待办，其中流程文件和代码打包在一起，但实际项目中很少会把流程文件和代码一起打包部署，这样的话，每次流程更新或者发布新流程都需要重新部署应用，因此我们制定了以下部署方案：

* 提供流程部署接口，可以通过上传流程文件对流程进行部署。
* 如果流程文件没有发生变化，不做新的部署，防止因为重新部署导致版本号上升。

**资源部署**

activit部署资源文件需要通过RepositoryService创建一个deployment，通过该deployment进行资源的部署，不单单是bpmn流程文件，activiti可以部署任何文件。

上传资源到activiti

**@Service**

**public** **class** **DeploymentService** {

**@Autowired**

**private** RepositoryService repositoryService;

*/\*\**

*\* deploy resource*

*\**

*\* @param name resource name*

*\* @param fin resource inputstream*

*\* @return*

*\*/*

**public** String **deploy**(String name, InputStream fin) {

String deploymentId = repositoryService.createDeployment()

.addInputStream(name, fin)

.name(name)

.key(name)

.deploy()

.getId();

**return** deploymentId;

}

}

部署的时候指定部署的name和key，方便后续对部署进行进一步操作。

声明restController

**@RestController**

**public** **class** **DeploymentController** {

**@Autowired**

**private** DeploymentService service;

**@PostMapping**(value = "/deploy")

**public** String **deploy**(@RequestParam("file") MultipartFile file) {

**try** {

**return** service.deploy(file.getOriginalFilename(), file.getInputStream());

} **catch** (IOException e) {

e.printStackTrace();

**throw** **new** RuntimeException("upload failed");

}

}

}

这里以文件名作为部署的名称，可以根据实际情况指定名称。

测试(可以通过postman进行测试，这里使用curl工具进行上传测试)

curl -X POST -F 'file=@WechatIMG1.jpeg' http://localhost:8080/deploy

25001

返回的25001就是部署id。

当执行deploy操作时，activiti后台做了以下事

* 在表ACT\_RE\_DEPLOYMENT创建一条记录
* 将资源存储至表ACT\_GE\_BYTEARRAY，字段BYTES\_存储文件内容

**流程部署**

流程部署和资源部署一样，但有一点需要注意，部署流程时，资源的名称必须以bpmn20.xml或者bpmn结尾，否则activiti会当作普通资源上传，你可以在controller上加个判断避免用户上错错误文件。

@PostMapping(value = "/deployBpmn")

public String deployBpmn(@RequestParam("file") MultipartFile file) {

try {

String name = file.getOriginalFilename();

if (!name.endsWith(".bpmn20.xm") && !name.endsWith(".bpmn")) {

name = name + ".bpmn";

}

return service.deploy(name, file.getInputStream());

} catch (IOException e) {

e.printStackTrace();

throw new RuntimeException("upload failed");

}

}

当部署流程后，activit后台除了执行部署资源的动作外，还额外执行以下动作

* 在表ACT\_RE\_PROCDEF上创建一条记录，引用deployment\_id。

**流程部署优化**

以上我们实现了流程的部署，但每次部署都会导致流程版本递增，我们的方案是如果文件不发生变化则不执行部署操作，保持流程版本好不变，不至于让版本号升的太快，也节省系统资源。那么我们需要完成以下两个逻辑

* 根据key查询到最新的一次部署
* 获取最新的一次流程源文件
* 因为xml是文本文件，我们可以进行文本比对，如果内容一致就认为版本不变

代码如下：

*/\*\**

*\* deploy resource*

*\**

*\* @param name resource name*

*\* @param fin resource inputstream*

*\* @return*

*\*/*

**public** String **noChangeNoDeploy**(String name, InputStream fin) {

*//获取最新的一次部署*

Deployment latestDeployment = repositoryService.createDeploymentQuery()

.deploymentName(name)

.deploymentKey(name)

.latest()

.singleResult();

String sbpmn = ActivitiUtil.text(fin);

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

*//检测是否内容发生变化，只重新部署有修改的流程*

InputStream input = repositoryService.getResourceAsStream(latestDeployment.getId(), name);

String dbpmn = ActivitiUtil.text(input);

**if** (sbpmn.length() == dbpmn.length() && sbpmn.equals(dbpmn)) {

**return** latestDeployment.getId();

}

}

String deploymentId = repositoryService.createDeployment()

.addString(name, sbpmn)

.name(name)

.key(name)

.deploy()

.getId();

**return** deploymentId;

}

测试：

curl -X POST -F 'file=@ComplexDemo.bpmn20.xml' http://localhost:8080/deployBpmn

27501

curl -X POST -F 'file=@ComplexDemo.bpmn20.xml' http://localhost:8080/deployBpmn

27501

可以看到，如果两次内容一样deployid不会改变

**其他**

除了通过addInputStream增加资源文件外，RepositoryService还提供了其他api进行资源的添加。如下

DeploymentBuilder **addInputStream**(String resourceName, InputStream inputStream);

DeploymentBuilder **addClasspathResource**(String resource);

DeploymentBuilder **addString**(String resourceName, String text);

DeploymentBuilder **addBytes**(String resourceName, **byte**[] bytes);

DeploymentBuilder **addZipInputStream**(ZipInputStream zipInputStream);

DeploymentBuilder **addBpmnModel**(String resourceName, BpmnModel bpmnModel);

本文所有代码已经上传至github，仓库地址为<https://github.com/wls1036/springboot-activiti6-tutorial>欢迎star

[**SpringBoot Activiti6系列教程(五)-activiti api**](https://segmentfault.com/a/1190000020993940)

[bpmn](https://segmentfault.com/t/bpmn)[jav](https://segmentfault.com/t/jav)[springboot](https://segmentfault.com/t/springboot)[activiti](https://segmentfault.com/t/activiti)

 阅读约 6 分钟

**介绍**

[Activiti api](https://www.activiti.org/userguide/#chapterApi)设计的非常友好，使用的过程中也是学习到了api设计的一些技巧，有时间也会整理下，activit api主要是分两大块

* Service
* Query

Service负责执行动作，Query负责执行查询，也就是涉及到数据的增、删、改由Service负责，涉及到数据的查由Query负责，在spring boot中，Service可以通过注入获取，Query可以通过相应的Service获取，所有的Service都可以通过ProcessEngine获取。

**Service**

activiti有8个service管理着activiti所有的资源

| **service名称** | **说明** |
| --- | --- |
| TaskService | 对用户任务进行操作和查询 |
| RepositoryService | 对activiti资源进行操作，比如部署文件，附件 |
| RuntimeService | 运行时服务，可以对运行时流程进行修改，如增加变量，移除变量等 |
| IdentityService | 身份认证服务，对用户，用户组，用户角色进行操作 |
| HistoryService | 历史记录服务，对审批历史进行操作 |
| FormService | 表单服务，操作表单数据 |
| DynamicBpmnService | 通过该服务，可以动态修改流程 |
| ManagementService | 管理服务，查看当前activiti系统信息，不会在应用里用到，一般用于管理系统里 |

**Query**

[之前章节](https://segmentfault.com/a/1190000020288808)提到，activiti总共会创建28张表，对于一个流程系统来说，28张表并不算多，你完全可以阅读源码或者发起几个流程看出各表之间的关系，然后自定义一些查询，但完全没这个必要，因为api的Query查询基本涵盖了所有的查询需求，而且因为其友好的api设计大部分的函数通过名称就能知道其功能，也可以通过功能反推其api名称。

这里稍微介绍下Query的设计，大家也可以学习下

**public** **interface** **Query**<**T** **extends** **Query**<?, ?>, **U** **extends** **Object**> {

*/\*\* 升序操作 \*/*

T **asc**();

*/\*\* 降序操作 \*/*

T **desc**();

*/\*\* 计算总数 \*/*

**long** **count**();

*/\*\* 查询单个结果 \*/*

U **singleResult**();

*/\*\* 查询列表 \*/*

List<U> **list**();

*/\*\* 分页查询 \*/*

List<U> **listPage**(**int** firstResult, **int** maxResults);

}

定义了5个基本操作，2个T和U的泛型，T为Query本身，在非数据返回的操作返回T自身可以让api以builder进行调用，U为操作的数据类型。

以下是一个实现类TaskQueryInfo的定义

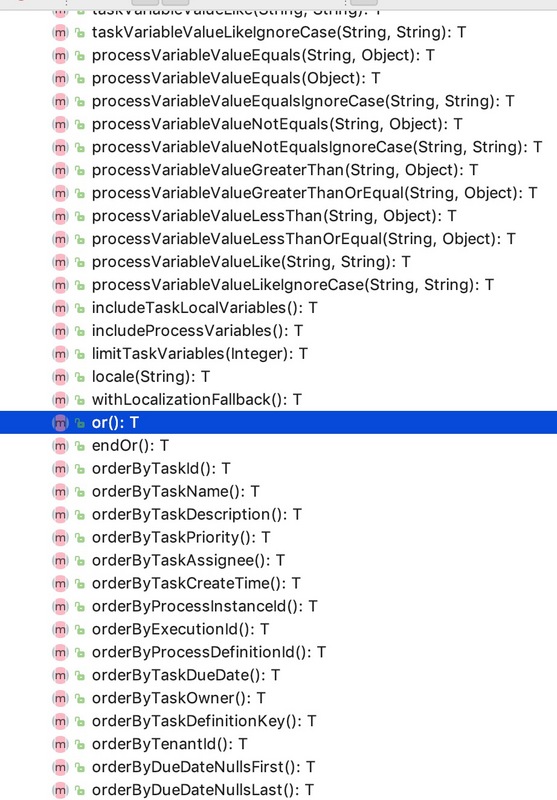
**public** **interface** **TaskInfoQuery**<**T** **extends** **TaskInfoQuery**<?, ?>, **V** **extends** **TaskInfo**> **extends** **Query**<**T**, **V**>{

}

然后将表中重要业务字段作为查询字段，并提供多种方式查询，以taskName为例

* T taskName(String name); 相等查询
* T taskNameIn(List<String> nameList); in查询
* T taskNameInIgnoreCase(List<String> nameList); 不分大小写in查询
* T taskNameLike(String nameLike); 模糊查询
* T taskNameLikeIgnoreCase(String nameLike); 不分大小写模糊查询

提供多个字段排序操作



**ProcessEngine**

ProcessEngine包含所有的service

**public** **interface** **ProcessEngine** {

String **getName**();

**void** **close**();

RepositoryService **getRepositoryService**();

RuntimeService **getRuntimeService**();

FormService **getFormService**();

TaskService **getTaskService**();

HistoryService **getHistoryService**();

IdentityService **getIdentityService**();

ManagementService **getManagementService**();

DynamicBpmnService **getDynamicBpmnService**();

ProcessEngineConfiguration **getProcessEngineConfiguration**();

FormRepositoryService **getFormEngineRepositoryService**();

org.activiti.form.api.FormService **getFormEngineFormService**();

}

[**SpringBoot Activiti6系列教程(六)-Execution说明**](https://segmentfault.com/a/1190000021003638)

[bpmn](https://segmentfault.com/t/bpmn)[activiti](https://segmentfault.com/t/activiti)[springboot](https://segmentfault.com/t/springboot)

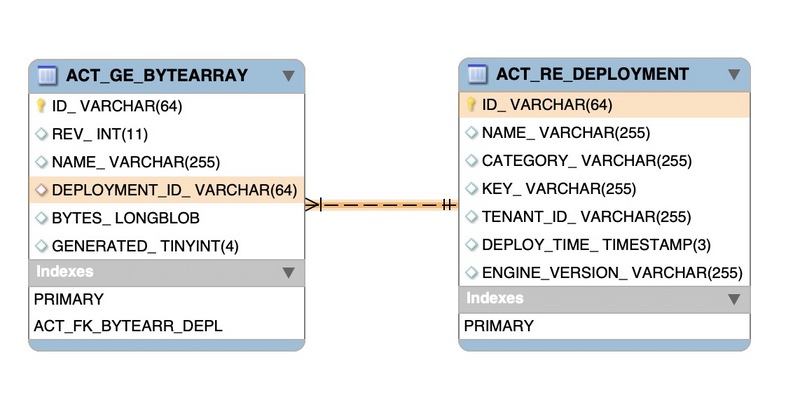
 阅读约 26 分钟

在activiti中有几个概念经常用，但文档也没有讲的很清楚，如果不理解，有可能会误用。本文就详细探讨以下几个概念

* deployment（部署）
* instance（实例）
* execution（执行）
* task（任务）

**deployment（部署）**

我们在[前面章节](https://segmentfault.com/a/1190000020439236)讲过，bpmn流程模型文件需要通过RepositoryService进行发布，相同名称的流程模型重新部署后会升一个版本，但并不影响之前的流程，还未结束的流程还是会按照发起时候的版本走。因此deployment就表示一次资源的部署，资源类型不一定是bpmn文件，可以是任意文件。上传的资源元信息保存在ACT\_RE\_DEPLOYMENT中，资源内容以二进制形式保存在表ACT\_GE\_BYTEARRAY中，两张表通过DEPLOYMENT\_ID字段关联



**instance（流程实例）**

一个流程每发起一次就是一个实例，实例数也是流程统计的重要数据，一个流程如果有10000个实例就表示这个流程已经被发起了10000次，流程实例在activiti中是没有实体表存储的，在流程发起时系统分配一个实例id(PROC\_INST\_ID\_)，在整个流程的流转中，该实例号都保持不变，因此可以根据PROC\_INST\_ID\_查询整个流程运行时的数据。当调用api发起流程时，应用程序应需要把PROC\_INST\_ID\_保存到自身的业务表中，后续对流程的管理和追踪都需要PROC\_INST\_ID\_，如要查询指定流程目前审批人信息，执行以下api：

TaskService taskService = ProcessEngines.getDefaultProcessEngine().getTaskService();

List<Task> tasks = taskService.createTaskQuery()

.processInstanceId(instanceId)

.list();

如果要追踪流程运行时信息，可以以PROC\_INST\_ID\_作为过滤条件，你可以根据PROC\_INST\_ID\_查询以下表的信息

mysql> SELECT table\_name,column\_name FROM information\_schema.COLUMNS where COLUMN\_NAME='PROC\_INST\_ID\_';

+-----------------------+---------------+

| TABLE\_NAME | COLUMN\_NAME |

+-----------------------+---------------+

| ACT\_EVT\_LOG | PROC\_INST\_ID\_ |

| ACT\_FO\_SUBMITTED\_FORM | PROC\_INST\_ID\_ |

| ACT\_HI\_ACTINST | PROC\_INST\_ID\_ |

| ACT\_HI\_ATTACHMENT | PROC\_INST\_ID\_ |

| ACT\_HI\_COMMENT | PROC\_INST\_ID\_ |

| ACT\_HI\_DETAIL | PROC\_INST\_ID\_ |

| ACT\_HI\_IDENTITYLINK | PROC\_INST\_ID\_ |

| ACT\_HI\_PROCINST | PROC\_INST\_ID\_ |

| ACT\_HI\_TASKINST | PROC\_INST\_ID\_ |

| ACT\_HI\_VARINST | PROC\_INST\_ID\_ |

| ACT\_RU\_EVENT\_SUBSCR | PROC\_INST\_ID\_ |

| ACT\_RU\_EXECUTION | PROC\_INST\_ID\_ |

| ACT\_RU\_IDENTITYLINK | PROC\_INST\_ID\_ |

| ACT\_RU\_TASK | PROC\_INST\_ID\_ |

| ACT\_RU\_VARIABLE | PROC\_INST\_ID\_ |

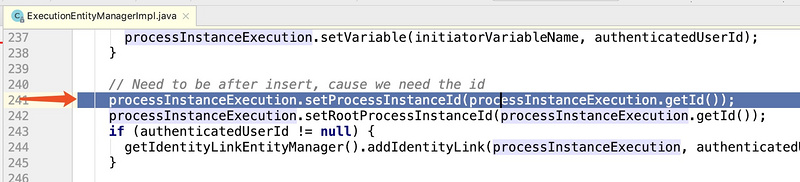
+-----------------------+---------------+

15 rows **in** set (0.01 sec)

**execution（执行）**

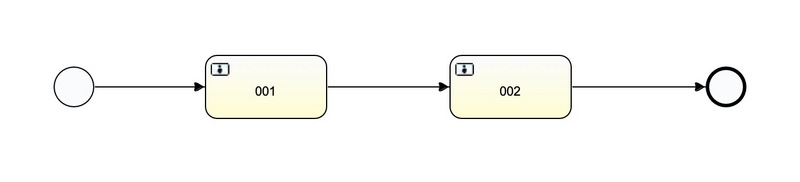
execution是比较难理解的一个概念，官方的解释是Represent a 'path of execution' in a process instance，代表流程实例的执行路径，有点抽象，什么意思呢，当流程没有分之，start到end就是一条直线，那么就只有一条执行路径，就是一个execution，如果流程有分之，比如平行审批，那么到平行网关就会分出两条路径，这时候就会产生两个的execution，新的execution通过PARENT\_ID\_与父exection进行关联，之所以要这么设计，我觉得是为了隔离不同执行路径间数据的隔离，比如在程序分之A上定义了一个变量，程序是不知道流程是否在分之上，在分之B上又重新定义了相同的变量，如果没有execution的概念就会导致变量被覆盖。

有一个比较特别的地方需要注意下，流程实例(ProcessInstance)也是一个execution，实例id（PROC\_INST\_ID\_）就是流程实例在execution表上的id，从源码上也可以看出：



下面我们通过两个具体例子来理解execution

* 简单无分之流程



以上就是一个简单的流程，流程发起后到001节点，001审批后到002，002审批后流程结束，通过代码进行发起，保存实例id，后续通过实例id进行观察

***发起代码***

**public** String **start**() {

ProcessInstance instance = runtimeService.startProcessInstanceByKey("execution\_simple");

**return** instance.getId();

}

发起后表信息

mysql> **SELECT** id\_,PROC\_INST\_ID\_,PARENT\_ID\_,PROC\_DEF\_ID\_,ROOT\_PROC\_INST\_ID\_,ACT\_ID\_ **FROM** activiti.ACT\_RU\_EXECUTION **where** proc\_inst\_id\_=77508;

+*-------+---------------+------------+-------------------+--------------------+---------+*

| id\_ | PROC\_INST\_ID\_ | PARENT\_ID\_ | PROC\_DEF\_ID\_ | ROOT\_PROC\_INST\_ID\_ | ACT\_ID\_ |

+*-------+---------------+------------+-------------------+--------------------+---------+*

| 77508 | 77508 | NULL | myProcess:2:77507 | 77508 | NULL |

| 77509 | 77508 | 77508 | myProcess:2:77507 | 77508 | task1 |

+*-------+---------------+------------+-------------------+--------------------+---------+*

2 rows in **set** (0.01 sec)

mysql> **select** ID\_,EXECUTION\_ID\_,PROC\_INST\_ID\_,PROC\_DEF\_ID\_,NAME\_,TASK\_DEF\_KEY\_ **from** ACT\_RU\_TASK **WHERE** PROC\_INST\_ID\_=77508;

+*-------+---------------+---------------+-------------------+-------+---------------+*

| ID\_ | EXECUTION\_ID\_ | PROC\_INST\_ID\_ | PROC\_DEF\_ID\_ | ASSIGNEE\_ | TASK\_DEF\_KEY\_ |

+*-------+---------------+---------------+-------------------+-----------+---------------+*

| 77512 | 77509 | 77508 | myProcess:2:77507 | 001 | task1 |

+*-------+---------------+---------------+-------------------+-----------+---------------+*

1 row in **set** (0.00 sec)

我们可以得到以下几个信息

* PROC\_INST\_ID\_为77508，和第一条exection的id一样
* 目前有产生两个execution，77512的task对应的execution为77509
* 77509的父execution是77508也就是流程实例的execution，ACT\_ID\_为task1，也就是第一个节点，说明目前该execution目前停留在第一个节点

001审批后表信息

mysql> **SELECT** id\_,PROC\_INST\_ID\_,PARENT\_ID\_,PROC\_DEF\_ID\_,ROOT\_PROC\_INST\_ID\_,ACT\_ID\_ **FROM** activiti.ACT\_RU\_EXECUTION **where** proc\_inst\_id\_=77508;

+*-------+---------------+------------+-------------------+--------------------+---------+*

| id\_ | PROC\_INST\_ID\_ | PARENT\_ID\_ | PROC\_DEF\_ID\_ | ROOT\_PROC\_INST\_ID\_ | ACT\_ID\_ |

+*-------+---------------+------------+-------------------+--------------------+---------+*

| 77508 | 77508 | NULL | myProcess:2:77507 | 77508 | NULL |

| 77509 | 77508 | 77508 | myProcess:2:77507 | 77508 | task2 |

+*-------+---------------+------------+-------------------+--------------------+---------+*

2 rows in **set** (0.00 sec)

mysql> **select** ID\_,EXECUTION\_ID\_,PROC\_INST\_ID\_,PROC\_DEF\_ID\_,ASSIGNEE\_,TASK\_DEF\_KEY\_ **from** ACT\_RU\_TASK **WHERE** PROC\_INST\_ID\_=77508;

+*-------+---------------+---------------+-------------------+-------+---------------+*

| ID\_ | EXECUTION\_ID\_ | PROC\_INST\_ID\_ | PROC\_DEF\_ID\_ | ASSIGNEE\_ | TASK\_DEF\_KEY\_ |

+*-------+---------------+---------------+-------------------+-----------+---------------+*

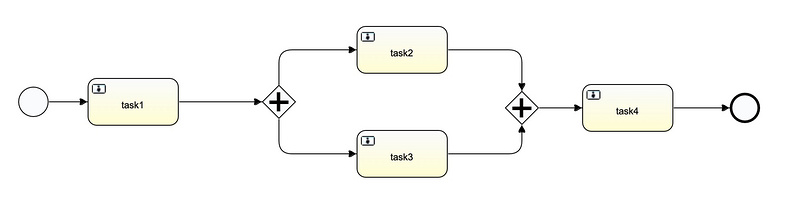
| 77515 | 77509 | 77508 | myProcess:2:77507 | 002 | task2 |

+*-------+---------------+---------------+-------------------+-------------+---------------+*

1 row in **set** (0.00 sec)

execution信息不变，因为当前流程没有分之。

* 平行审批流程



***流程发起***

mysql> **SELECT** id\_,PROC\_INST\_ID\_,PARENT\_ID\_,PROC\_DEF\_ID\_,ROOT\_PROC\_INST\_ID\_,ACT\_ID\_ **FROM** activiti.ACT\_RU\_EXECUTION **where** proc\_inst\_id\_=80011;

+*-------+---------------+------------+-------------------------+--------------------+---------+*

| id\_ | PROC\_INST\_ID\_ | PARENT\_ID\_ | PROC\_DEF\_ID\_ | ROOT\_PROC\_INST\_ID\_ | ACT\_ID\_ |

+*-------+---------------+------------+-------------------------+--------------------+---------+*

| 80011 | 80011 | NULL | parallelProcess:1:80009 | 80011 | NULL |

| 80012 | 80011 | 80011 | parallelProcess:1:80009 | 80011 | task1 |

+*-------+---------------+------------+-------------------------+--------------------+---------+*

2 rows in **set** (0.00 sec)

mysql> mysql> **select** ID\_,EXECUTION\_ID\_,PROC\_INST\_ID\_,PROC\_DEF\_ID\_,ASSIGNEE\_,TASK\_DEF\_KEY\_ **from** ACT\_RU\_TASK **WHERE** PROC\_INST\_ID\_=80011;

+*-------+---------------+---------------+-------------------------+-----------+---------------+*

| ID\_ | EXECUTION\_ID\_ | PROC\_INST\_ID\_ | PROC\_DEF\_ID\_ | ASSIGNEE\_ | TASK\_DEF\_KEY\_ |

+*-------+---------------+---------------+-------------------------+-----------+---------------+*

| 80015 | 80012 | 80011 | parallelProcess:1:80009 | 001 | task1 |

+*-------+---------------+---------------+-------------------------+-----------+---------------+*

1 row in **set** (0.00 sec)

***001审批后***

mysql> **SELECT** id\_,PROC\_INST\_ID\_,PARENT\_ID\_,PROC\_DEF\_ID\_,ROOT\_PROC\_INST\_ID\_,ACT\_ID\_ **FROM** activiti.ACT\_RU\_EXECUTION **where** proc\_inst\_idINST\_ID\_=80011;

+*-------+---------------+------------+-------------------------+--------------------+---------+*

| id\_ | PROC\_INST\_ID\_ | PARENT\_ID\_ | PROC\_DEF\_ID\_ | ROOT\_PROC\_INST\_ID\_ | ACT\_ID\_ |

+*-------+---------------+------------+-------------------------+--------------------+---------+*

| 80011 | 80011 | NULL | parallelProcess:1:80009 | 80011 | NULL |

| 80012 | 80011 | 80011 | parallelProcess:1:80009 | 80011 | task3 |

| 80018 | 80011 | 80011 | parallelProcess:1:80009 | 80011 | task2 |

+*-------+---------------+------------+-------------------------+--------------------+---------+*

3 rows in **set** (0.00 sec)

mysql> **select** ID\_,EXECUTION\_ID\_,PROC\_INST\_ID\_,PROC\_DEF\_ID\_,ASSIGNEE\_,TASK\_DEF\_KEY\_ **from** ACT\_RU\_TASK **WHERE** PROC\_INST\_ID\_=80011;

+*-------+---------------+---------------+-------------------------+-----------+---------------+*

| ID\_ | EXECUTION\_ID\_ | PROC\_INST\_ID\_ | PROC\_DEF\_ID\_ | ASSIGNEE\_ | TASK\_DEF\_KEY\_ |

+*-------+---------------+---------------+-------------------------+-----------+---------------+*

| 80020 | 80012 | 80011 | parallelProcess:1:80009 | 003 | task3 |

| 80023 | 80018 | 80011 | parallelProcess:1:80009 | 002 | task2 |

+*-------+---------------+---------------+-------------------------+-----------+---------------+*

2 rows in **set** (0.00 sec)

可以看出80018是新创建的execution，task3复用了原来的exection 80012，两个execution的父id都是80011，也就是流程实例execution。

***003审批后***

mysql> **SELECT** id\_,PROC\_INST\_ID\_,PARENT\_ID\_,PROC\_DEF\_ID\_,ROOT\_PROC\_INST\_ID\_,ACT\_ID\_ **FROM** activiti.ACT\_RU\_EXECUTION **where** proc\_inst\_id\_=80011;

+*-------+---------------+------------+-------------------------+--------------------+------------------+*

| id\_ | PROC\_INST\_ID\_ | PARENT\_ID\_ | PROC\_DEF\_ID\_ | ROOT\_PROC\_INST\_ID\_ | ACT\_ID\_ |

+*-------+---------------+------------+-------------------------+--------------------+------------------+*

| 80011 | 80011 | NULL | parallelProcess:1:80009 | 80011 | NULL |

| 80012 | 80011 | 80011 | parallelProcess:1:80009 | 80011 | parallelgateway2 |

| 80018 | 80011 | 80011 | parallelProcess:1:80009 | 80011 | task2 |

+*-------+---------------+------------+-------------------------+--------------------+------------------+*

3 rows in **set** (0.00 sec)

mysql> **select** ID\_,EXECUTION\_ID\_,PROC\_INST\_ID\_,PROC\_DEF\_ID\_,ASSIGNEE\_,TASK\_DEF\_KEY\_ **from** ACT\_RU\_TASK **WHERE** PROC\_INST\_ID\_=80011;

+*-------+---------------+---------------+-------------------------+-----------+---------------+*

| ID\_ | EXECUTION\_ID\_ | PROC\_INST\_ID\_ | PROC\_DEF\_ID\_ | ASSIGNEE\_ | TASK\_DEF\_KEY\_ |

+*-------+---------------+---------------+-------------------------+-----------+---------------+*

| 80023 | 80018 | 80011 | parallelProcess:1:80009 | 002 | task2 |

+*-------+---------------+---------------+-------------------------+-----------+---------------+*

1 row in **set** (0.00 sec)

80012停留在parallelgateway2上等待80018结束，没有新的execution产生

***002审批后***

mysql> **SELECT** id\_,PROC\_INST\_ID\_,PARENT\_ID\_,PROC\_DEF\_ID\_,ROOT\_PROC\_INST\_ID\_,ACT\_ID\_ **FROM** activiti.ACT\_RU\_EXECUTION **where** proc\_inst\_id\_=80011;

+*-------+---------------+------------+-------------------------+--------------------+---------+*

| id\_ | PROC\_INST\_ID\_ | PARENT\_ID\_ | PROC\_DEF\_ID\_ | ROOT\_PROC\_INST\_ID\_ | ACT\_ID\_ |

+*-------+---------------+------------+-------------------------+--------------------+---------+*

| 80011 | 80011 | NULL | parallelProcess:1:80009 | 80011 | NULL |

| 80018 | 80011 | 80011 | parallelProcess:1:80009 | 80011 | task4 |

+*-------+---------------+------------+-------------------------+--------------------+---------+*

2 rows in **set** (0.00 sec)

mysql> **select** ID\_,EXECUTION\_ID\_,PROC\_INST\_ID\_,PROC\_DEF\_ID\_,ASSIGNEE\_,TASK\_DEF\_KEY\_ **from** ACT\_RU\_TASK **WHERE** PROC\_INST\_ID\_=80011;

+*-------+---------------+---------------+-------------------------+-----------+---------------+*

| ID\_ | EXECUTION\_ID\_ | PROC\_INST\_ID\_ | PROC\_DEF\_ID\_ | ASSIGNEE\_ | TASK\_DEF\_KEY\_ |

+*-------+---------------+---------------+-------------------------+-----------+---------------+*

| 80028 | 80018 | 80011 | parallelProcess:1:80009 | 004 | task4 |

+*-------+---------------+---------------+-------------------------+-----------+---------------+*

1 row in **set** (0.00 sec)

复用了80018这个execution

004审批后流程结束。

通过以上例子可以看出

* 如果没有分之，就只有两个execution（另外一个是实例execution）
* 遇到分之节点会产生新的execution，activiti会复用之前的execution，分之合并后保留其中一个execution

实际上不只是分之，如果一个节点多个人审批，通过multiInstanceLoopCharacteristics指定，那么每次循环都会创建一个新的execution，这个就留给读者自行去验证。

**task（任务）**

这里说的task指的是usertask，usertask就是一个人的待办，可以通过activiti提供的api可以查询指定人当前待办信息，当activiti遇到usertask节点就会进入等待状态，等待用户审批完成后继续执行。用户的待办任务存储在ACT\_RU\_TASK表上。