

1.环境搭建

开发环境：

activiti 5.22.0

jdk 1.8

mysql 5.6

tomcat 8.5

注意，activiti 需要在数据库里创建表，可以在官网下载源码安装包，里面有数据库的表。或者通过 java 代码自动创建表。所以 1.1 和 1.2 节中的方法二选一即可。








1.1 下载源码包

<https://www.activiti.org/download-links>




Previous Releases

activiti-6.0.0.zip	115 MB	Apache License	26 May 2017
activiti-5.22.0.zip	90 MB	Apache License	9 Dec 2016

本教程使用的是 5.22.0 版本，解压后如下：

	database	2017/12/10 19:58	文件夹	
	docs	2017/12/10 19:58	文件夹	
	libs	2017/12/10 19:58	文件夹	
	wars	2018/3/30 16:49	文件夹	
	license.txt	2016/11/3 15:12	文本文档	12 KB
	notice.txt	2016/11/3 15:12	文本文档	9 KB
	readme.html	2016/11/3 15:12	Firefox HTML D...	228 KB

在 database\create 文件夹下有各种版本数据库使用的创建数据库的 sql 语句。5.22.0 版本里有 25 张基础表。此处使用 mysql5.6 数据库，新建一个数据库，运行如下文件

	activiti.mysql.create.engine.sql	2016/11/3 15:12	SQL 文件	10 KB
	activiti.mysql.create.history.sql	2016/11/3 15:12	SQL 文件	6 KB
	activiti.mysql.create.identity.sql	2016/11/3 15:12	SQL 文件	2 KB

运行后得到 25 张表：

- 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_procinstant
- 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_event_subscr
- act_ru_execution
- act_ru_identitylink
- act_ru_job
- act_ru_task
- act_ru_variable

数据库表说明：

简介		
#	前缀	描述
1	ACT_RE_	RE表示Repository资源库，保存流程定义，模型等设计阶段的数据。
2	ACT_RU_	RU表示Runtime运行时，保存流程实例，任务，变量等运行阶段的数据。
3	ACT_HI_	HI表示History历史，保存历史实例，历史任务等流程历史数据。
4	ACT_ID_	ID表示Identity身份，保存用户，群组，关系等组织机构相关数据。（Activiti中的组织机构过于简单，仅用于演示。）
5	ACT_GE_	GE表示General通用，属于一些通用配置。
6	其他	ACT_EVT_LOG和ACT_PROCDEF_INFO没有按照规则来，两者分别属于HI和RE。

1.2 通过 java 代码创建表

引入依赖的 jar

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
```

```
</dependency>
<dependency>
  <groupId>org.activiti</groupId>
  <artifactId>activiti-bpmn-converter</artifactId>
  <version>5.22.0</version>
</dependency>
<dependency>
  <groupId>org.activiti</groupId>
  <artifactId>activiti-bpmn-model</artifactId>
  <version>5.22.0</version>
</dependency>
<dependency>
  <groupId>org.activiti</groupId>
  <artifactId>activiti-image-generator</artifactId>
  <version>5.22.0</version>
</dependency>
<dependency>
  <groupId>org.activiti</groupId>
  <artifactId>activiti-process-validation</artifactId>
  <version>5.22.0</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.6.1</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>1.6.1</version>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.44</version>
</dependency>
<dependency>
  <groupId>commons-dbcp</groupId>
  <artifactId>commons-dbcp</artifactId>
  <version>1.4</version>
</dependency>
<dependency>
  <groupId>org.activiti</groupId>
  <artifactId>activiti-root</artifactId>
```

```

    <version>5.22.0</version>
  </dependency>
</dependency>
  <groupId>org.activiti</groupId>
  <artifactId>activiti-spring</artifactId>
  <version>5.22.0</version>
</dependency>
</dependencies>

```

在 resources 目录下创建 activiti.cfg.xml，配置数据源和默认引擎

```

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" >
  <property name="driverClassName" value="com.mysql.jdbc.Driver" />
  <property name="url"
value="jdbc:mysql://localhost:3306/activiti2?characterEncoding=utf-8" />
  <property name="username" value="root" />
  <property name="password" value="" />
</bean>

<bean id="processEngineConfiguration"
class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
  <property name="dataSource" ref="dataSource" />
  <property name="databaseSchemaUpdate" value="true" />
</bean>

```

其中 databaseSchemaUpdate 属性允许在进程引擎启动和关闭时设置策略来处理数据库模式。

- 1) false（默认）：当创建流程引擎时，检查数据库模式对库的版本，如果版本不匹配则抛出异常。
- 2) true：在构建流程引擎时，执行检查，如果需要，执行模式的更新。如果模式不存在，则创建它。
- 3) create-drop：在创建流程引擎时创建模式，并在流程引擎关闭时删除模式。这里设置成 true，运行程序的时候自动生成表。

activiti 底层使用的是 mybatis，可以通过 log4j 查看执行的语句：

```

# For all other servers: Comment out the Log4J listener in web.xml to activate Log4J.
log4j.rootLogger=DEBUG, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - %m%n
#begin
#for normal test,delete when online
log4j.logger.com.ibatis=DEBUG
log4j.logger.com.ibatis.common.jdbc.SimpleDataSource=DEBUG
log4j.logger.com.ibatis.common.jdbc.ScriptRunner=DEBUG
log4j.logger.com.ibatis.sqlmap.engine.impl.SqlMapClientDelegate=DEBUG
log4j.logger.java.sql.Connection=DEBUG
log4j.logger.java.sql.Statement=DEBUG

```

```
log4j.logger.java.sql.PreparedStatement=DEBUG  
log4j.logger.java.sql.ResultSet=DEBUG
```

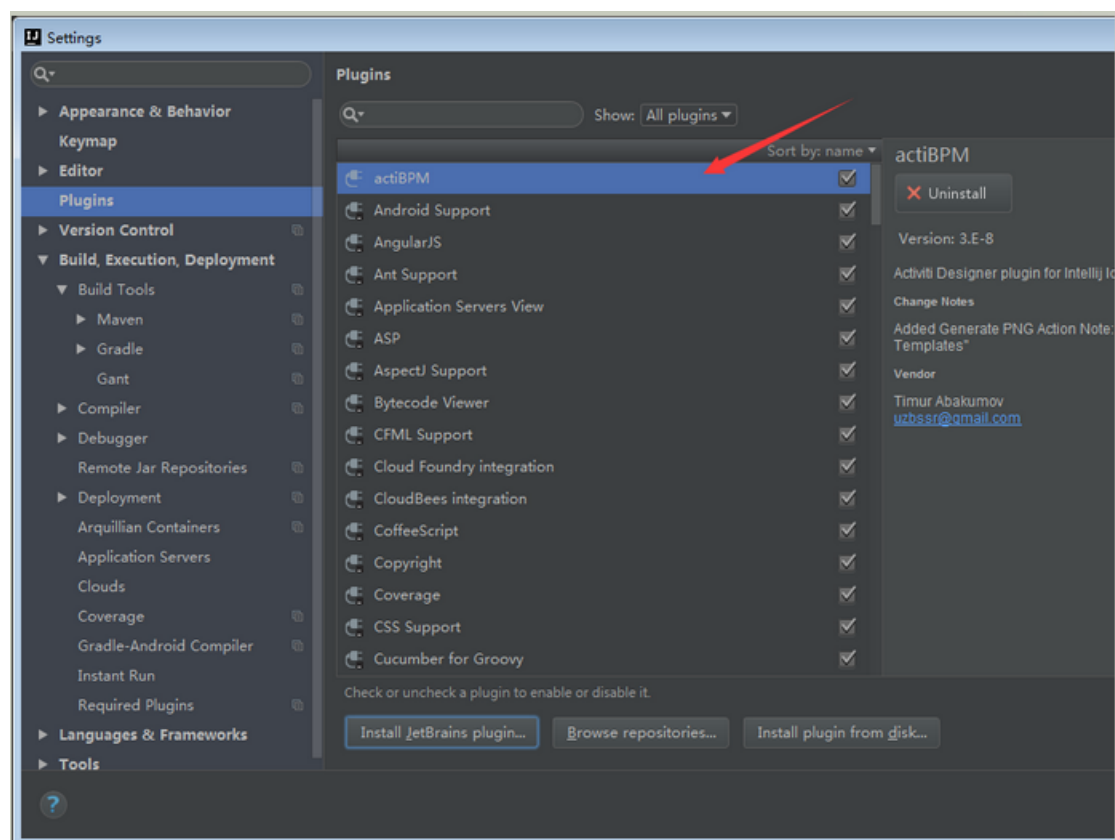
测试代码：

```
@Test  
public void createTable() {  
    ProcessEngineConfiguration configuration =  
  
    ProcessEngineConfiguration.createProcessEngineConfigurationFromResource("activiti.  
cfg.xml");  
    //ProcessEngines.getDefaultProcessEngine();  
    ProcessEngine processEngine = configuration.buildProcessEngine();  
    System.out.println(processEngine);  
}
```

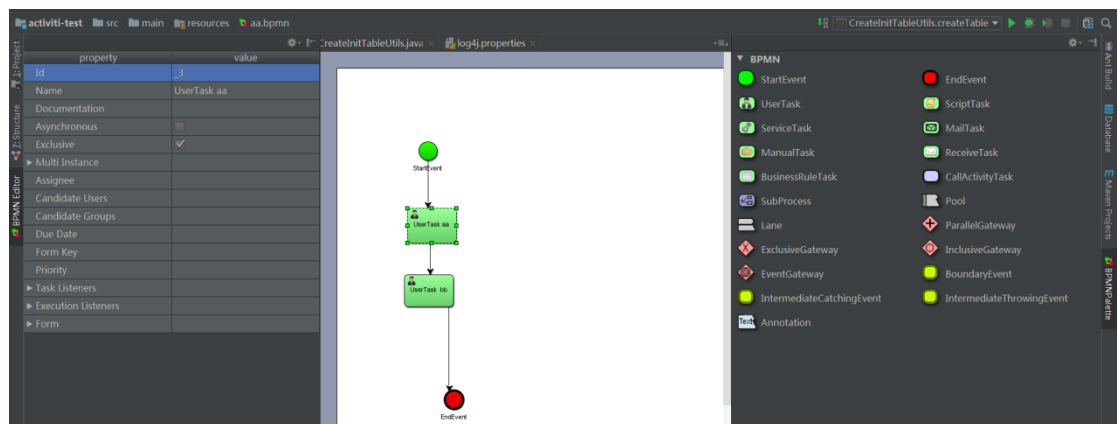
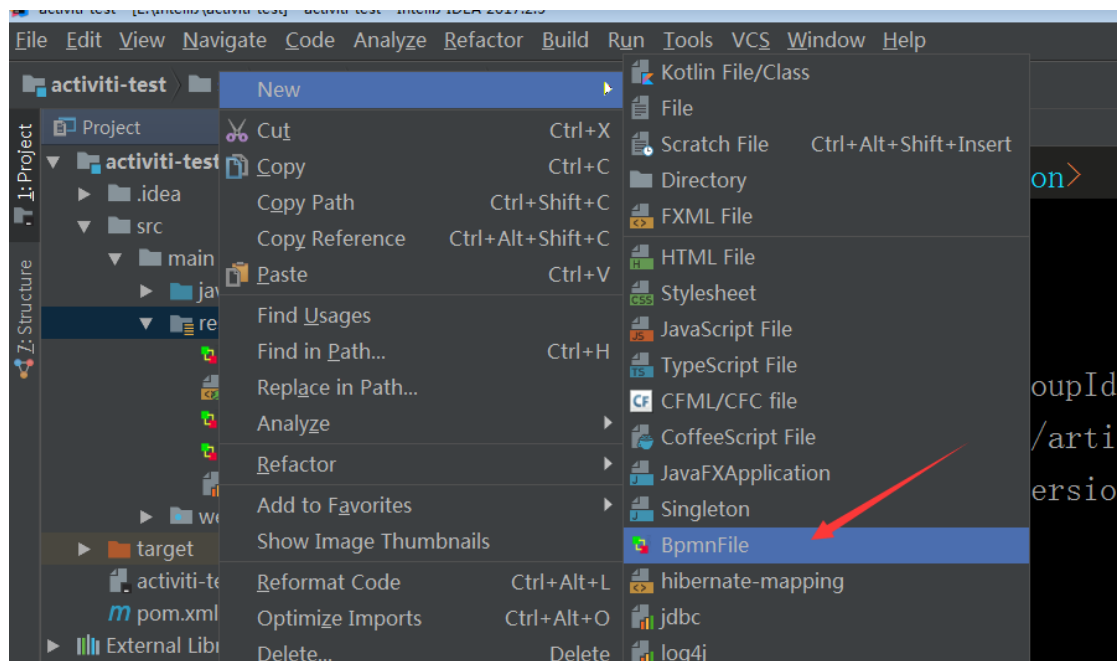
1.3 idea 安装 activiti 插件

File->Settings->Plugins

搜索 actiBPM，右侧选择 Install，安装成功后重启 Idea。



在工程中可以创建 bpmn 文件了。bpmn 规范会在后续课程中讲解。



idea 中绘制流程图的时候中文会乱码，在 idea 安装路径的 bin 目录下修改配置文件：

idea.exe.vmoptions	2017/9/29 14:55	VMOPTIONS
idea.ico	2017/9/29 14:55	图标
idea.properties	2017/11/28 11:27	PROPERTIES
idea.properties.bak	2017/11/28 11:27	BAK 文件
idea64.exe	2017/9/29 14:55	应用程序
idea64.exe.vmoptions	2017/9/29 14:55	VMOPTIONS

在文件末尾加一句，如图：-Dfile.encoding=UTF-8

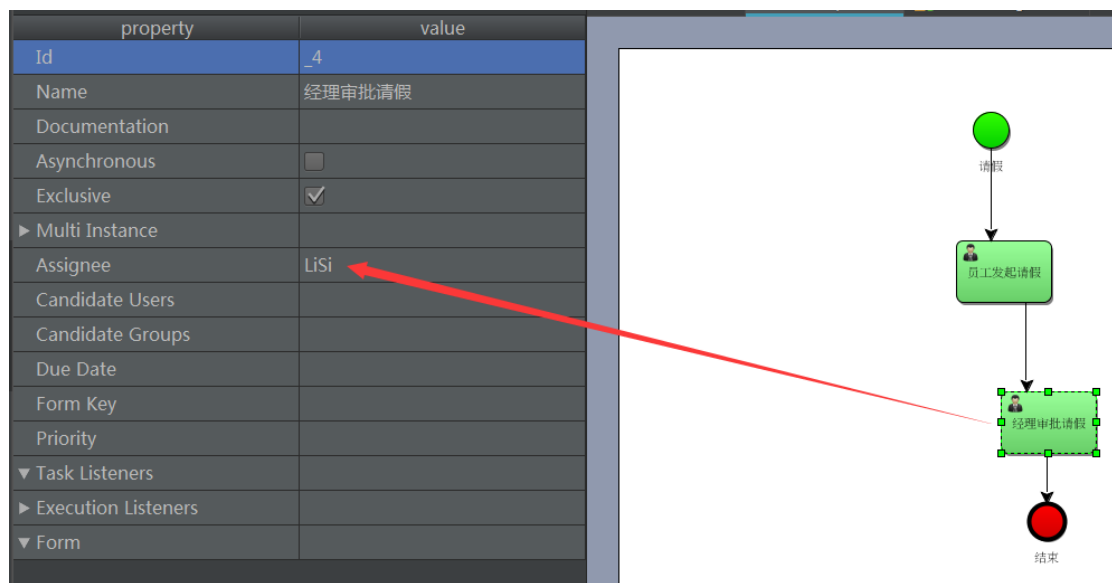
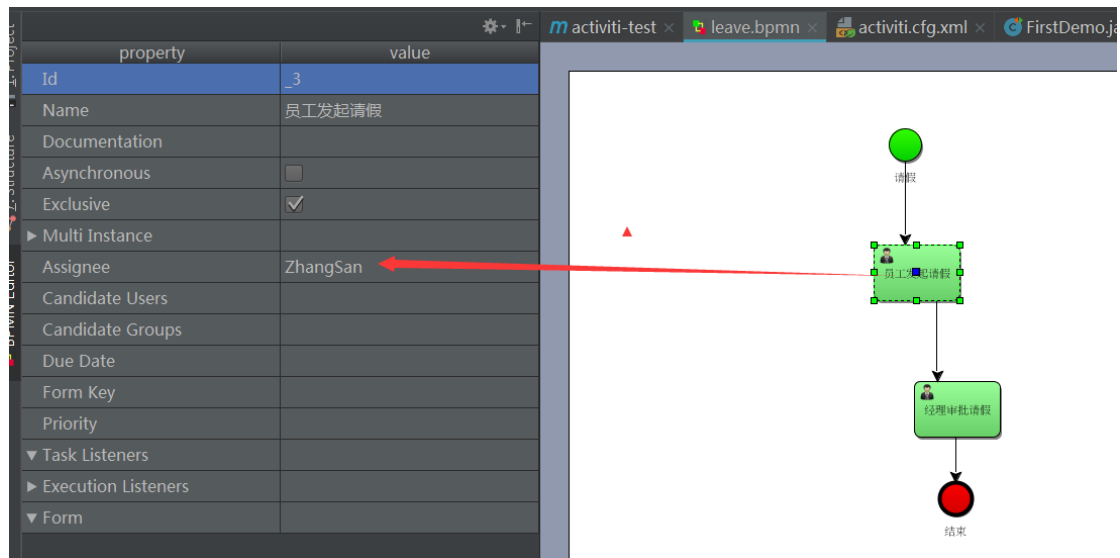
```

1 -Xms128m
2 -Xmx750m
3 -XX:ReservedCodeCacheSize=240m
4 -XX:+UseConcMarkSweepGC
5 -XX:SoftRefLRUPolicyMSPerMB=50
6 -ea
7 -Dsun.io.useCanonCaches=false
8 -Djava.net.preferIPv4Stack=true
9 -XX:+HeapDumpOnOutOfMemoryError
10 -XX:-OmitStackTraceInFastThrow
11 -Dfile.encoding=UTF-8

```

1.4 示例工程

绘制流程图 leave.bpmn: ZhangSan 发起请假, LiSi 审批请假



部署任务:

```
@Test
public void addDeployment() {
    Deployment deployment = processEngine.getRepositoryService() // 与流程定义
    和部署对象相关的 Service
        .createDeployment() // 创建一个部署对象
        .name("请假流程") // 设置对应流程的名称
        .addClasspathResource("leave.bpmn")
        .deploy(); // 完成部署
}
```

数据库里已经插入了对应的数据

act_id_user @activiti (loc... x

act_re_deployment @ac... x

act_ge_bytearray @acti... x

文件

编辑

查看

窗口

帮助

导入向导

导出向导

筛选向导

网格查看

表单查看

备注

十六进制

图像

ID_	NAME_	CATEGORY_	TENANT_ID_	DEPLOY_TIME_
▶ 2501	请假流程	(Null)		2018-04-02 15:15:19.259

act_id_user @activiti (loc... x

act_re_deployment @ac... x

act_ge_bytearray @acti... x

文件

编辑

查看

窗口

帮助

导入向导

导出向导

筛选向导

网格查看

表单查看

备注

十六进制

图像

ID_	REV_	NAME_	DEPLOYMENT_ID_	BYTES_	GENERATED_
▶ 2502	1	leave.bpmn	2501	(BLOB)	0
2503	1	leave.myProcess_1.png	2501	(BLOB)	1

act_id_user @activiti (loc... x

act_re_deployment @ac... x

act_ge_bytearray @acti... x

act_re_procdef @activiti... x

文件

编辑

查看

窗口

帮助

导入向导

导出向导

筛选向导

网格查看

表单查看

备注

十六进制

图像

A-Z 升序排序

A-Z 降序排序

移除排序

自定义排序

ID_	REV_	CATEGORY_	NAME_	KEY_	VERSION_	DEPLOYMENT_ID_	RESOURCE_NAME_	DGRM_RESOURCE_NAME_
myProcess_1:1:2504	1	http://www.activiti.org/tes	(Null)	myProcess_1	1	2501	leave.bpmn	leave.myProcess_1.png

启动任务：

```
@Test
public void startProcess() {
    RuntimeService runtimeService = processEngine.getRuntimeService();
    //key 是 act_re_procdef 中的 KEY_, bpmn 的 id
    runtimeService.startProcessInstanceByKey("myProcess_1");
}
```

查看 ZhangSan 的任务并提交给下一个操作者。查看李四的任务并处理。(act_ru_task 表)

```
@Test
public void queryZhangSanTask() {
    String assignee = "ZhangSan";
    List<Task> taskList = processEngine.getTaskService().createTaskQuery().taskAssignee(assignee).list();
    for (Task task : taskList) {
        System.out.println("代办任务 ID:" + task.getId());
        System.out.println("代办任务 name:" + task.getName());
        System.out.println("代办任务创建时间:" + task.getCreateTime());
        System.out.println("代办任务办理人:" + task.getAssignee());
        System.out.println("流程实例 ID:" + task.getProcessInstanceId());
        System.out.println("执行对象 ID:" + task.getExecutionId());
        //提交任务到下一个代理人
        engine.getTaskService().complete(task.getId());
    }
}
@Test
```



```

public void queryLiSiTask() {
    String assignee = "LiSi";
    List<Task> taskList = processEngine.getTaskService()//获取任务 service
        .createTaskQuery()//创建查询对象
        .taskAssignee(assignee)//指定查询人
        .list();
    for (Task task : taskList) {
        System.out.println("代办任务 ID:" + task.getId());
        System.out.println("代办任务 name:" + task.getName());
        System.out.println("代办任务创建时间:" + task.getCreateTime());
        System.out.println("代办任务办理人:" + task.getAssignee());
        System.out.println("流程实例 ID:" + task.getProcessInstanceId());
        System.out.println("执行对象 ID:" + task.getExecutionId());
        processEngine.getTaskService().complete(task.getId());
    }
}

```

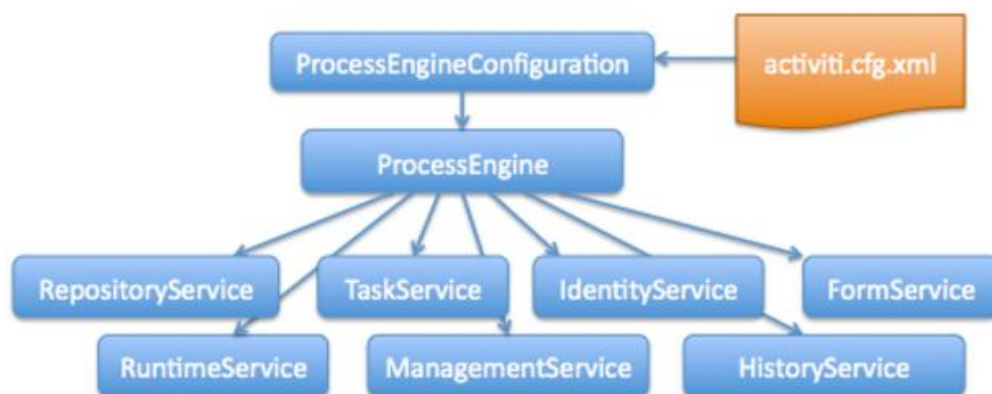
查看执行历史 act_hi_taskinst 表:

```

@Test
public void viewHistory() {
    HistoryService historyService = processEngine.getHistoryService();
    HistoricProcessInstance historicProcessInstance =
        historyService
            .createHistoricProcessInstanceQuery()
            .processInstanceId("15001").singleResult();
    System.out.println("开始时间: " + historicProcessInstance.getStartTime());
    System.out.println("结束时间: " + historicProcessInstance.getEndTime());
}

```

2.Activiti 核心配置



ProcessEngine 流程引擎是 Activiti 的核心。Activiti 默认读取 activiti.cfg.xml 文件，获得 ProcessEngineConfiguration 对象，通过 ProcessEngineConfiguration 创建

ProcessEngine。

2.1 ProcessEngineConfiguration 配置

2.1.1 不使用配置文件

```
@Test
public void createEngineWithoutXml() {
    ProcessEngineConfiguration processEngineConfiguration =
        ProcessEngineConfiguration.createStandaloneInMemProcessEngineConfiguration();
    processEngineConfiguration.setJdbcDriver("com.mysql.jdbc.Driver");

    processEngineConfiguration.setJdbcUrl("jdbc:mysql://localhost:3306/activiti?characterEncoding=utf-8");
    processEngineConfiguration.setJdbcUsername("root");
    processEngineConfiguration.setJdbcPassword("");
    processEngineConfiguration.setDatabaseSchemaUpdate("true");
    ProcessEngine processEngine = processEngineConfiguration.buildProcessEngine();
    System.out.println(processEngine);
}
```

2.1.2 读取配置文件

配置文件 activiti.cfg.xml，注意配置 XML 文件其实是一个 spring 的配置文件。但不是说 **Activiti** 只能用在 **Spring** 环境中！我们只是利用了 **Spring** 的解析和依赖注入功能 来构建引擎。

读取配置文件一共有五个方法：

- 1) ProcessEngineConfiguration.createProcessEngineConfigurationFromResourceDefault();
- 2) ProcessEngineConfiguration.createProcessEngineConfigurationFromResource(String resource);
- 3) ProcessEngineConfiguration.createProcessEngineConfigurationFromResource(String resource, String beanName);
- 4) ProcessEngineConfiguration.createProcessEngineConfigurationFromInputStream(InputStream inputStream);
- 5) ProcessEngineConfiguration.createProcessEngineConfigurationFromInputStream(InputStream inputStream, String beanName);

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
</bean>
```

```

    <property name="url"
value="jdbc:mysql://localhost:3306/activiti?characterEncoding=utf-8"/>
    <property name="username" value="root"/>
    <property name="password" value=""/>
</bean>
<!--带数据源-->
<bean id="processEngineConfiguration"
class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
    <property name="dataSource" ref="dataSource"/>
    <property name="databaseSchemaUpdate" value="true"/>
</bean>

<!--不带数据源-->
<!--<bean id="processEngineConfiguration"
class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">-->
    <!--<property name="jdbcUrl" value="com.mysql.jdbc.Driver" />-->
    <!--<property name="jdbcDriver"
value="jdbc:mysql://localhost:3306/activiti?characterEncoding=utf-8" />-->
    <!--<property name="jdbcUsername" value="root" />-->
    <!--<property name="jdbcPassword" value=""/>-->
    <!--<property name="databaseSchemaUpdate" value="true"/>-->
<!--</bean>-->

```

```

@Test
public void createEngineWithXml() {
    //默认读取的是 activiti.cfg.xml，并且 bean 的 id 必须是
processEngineConfiguration
//      ProcessEngineConfiguration processEngineConfiguration =
//
ProcessEngineConfiguration.createProcessEngineConfigurationFromResourceDefault();
    ProcessEngineConfiguration processEngineConfiguration =
ProcessEngineConfiguration.createProcessEngineConfigurationFromResource("activiti.
cfg.xml");
    ProcessEngine processEngine =
processEngineConfiguration.buildProcessEngine();
    System.out.println(processEngine);
}

```

如果自定义 bean 的 id，可以这样读取：

```

ProcessEngineConfiguration processEngineConfiguration =
ProcessEngineConfiguration.createProcessEngineConfigurationFromResource("activiti.
cfg.xml","configuration");

```

2.1.3 ProcessEngineConfiguration 子类

创建 `ProcessEngineConfiguration` 时，目前可以使用如下类，以后会更多：

- 1) `org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration`: 单独运行的流程引擎。Activiti 会自己处理事务。默认，数据库只在引擎启动时检测（如果没有 Activiti 的表或者表结构不正确就会抛出异常）。
- 2) `org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration`: 单元测试时的辅助类。Activiti 会自己控制事务。默认使用 H2 内存数据库。数据库表会在引擎启动时创建，关闭时删除。使用它时，不需要其他配置（除非使用 job 执行器或邮件功能）。
- 3) `org.activiti.spring.SpringProcessEngineConfiguration`: 在 Spring 环境下使用流程引擎。
- 4) `org.activiti.engine.impl.cfg.JtaProcessEngineConfiguration`: 单独运行流程引擎，并使用 JTA 事务。

2.1.4 数据库配置

Activiti 支持如下数据库：

数据库类型	url 示例
h2	<code>jdbc:h2:tcp://localhost/activiti</code>
mysql	<code>jdbc:mysql://localhost:3306/activiti?autoReconnect=true</code>
oracle	<code>jdbc:oracle:thin:@localhost:1521:xe</code>
postgres	<code>jdbc:postgresql://localhost:5432/activiti</code>
db2	<code>jdbc:db2://localhost:50000/activiti</code>
mssql	<code>jdbc:sqlserver://localhost:1433;databaseName=activiti</code> (<code>jdbc.driver=com.microsoft.sqlserver.jdbc.SQLServerDriver</code>) OR <code>jdbc:jtds:sqlserver://localhost:1433/activiti</code> (<code>jdbc.driver=net.sourceforge.jtds.jdbc.Driver</code>)

Activiti 可能使用两种方式配置数据库。第一种方式是定义数据库配置参数，另一种是直接配置 `DataSource`。

直接配置数据库参数，有如下参数：

- `jdbcUrl`: 数据库的 JDBC URL。
- `jdbcDriver`: 对应不同数据库类型的驱动。
- `jdbcUsername`: 连接数据库的用户名。
- `jdbcPassword`: 连接数据库的密码。

基于 JDBC 参数配置的数据库连接 会使用默认的 MyBatis 连接池。 下面的参数可以用来配置连接池（来自 MyBatis 参数）：

- **jdbcMaxActiveConnections**: 连接池中处于被使用状态的连接的最大值。默认为 10。
- **jdbcMaxIdleConnections**: 连接池中处于空闲状态的连接的最大值。
- **jdbcMaxCheckoutTime**: 连接被取出使用的最长时间，超过时间会被强制回收。默认为 20000（20 秒）。
- **jdbcMaxWaitTime**: 这是一个底层配置，让连接池可以在长时间无法获得连接时，打印一条日志，并重新尝试获取一个连接。（避免因错误配置导致沉默的操作失败）。默认为 20000（20 秒）。

无论你使用 JDBC 还是 DataSource 的方式，都可以设置下面的配置：

- **databaseType**: 一般不用设置，因为可以自动通过数据库连接的元数据获取。只有自动检测失败时才需要设置。可能的值有：{h2, mysql, oracle, postgres, mssql, db2}。这个配置会决定使用哪些创建/删除脚本和查询语句。
- **databaseSchemaUpdate**: 设置流程引擎启动和关闭时如何处理数据库表，可能值如下：
 - 1) false（默认）：检查数据库表的版本和依赖库的版本，如果版本不匹配就抛出异常。
 - 2) true: 构建流程引擎时，执行检查，如果需要就执行更新。如果表不存在，就创建。
 - 3) create-drop: 构建流程引擎时创建数据库表，关闭流程引擎时删除这些表。

使用数据源的配置参考各种 DataSource 的使用说明，ProcessEngineConfiguration 的 dataSource 属性引用定义的数据源即可。如 2.1.2 中使用了 DBCP 的数据源。

2.2 ProcessEngine 创建

```
@Test
public void createDefaultEngine() {
    //默认读取的是 activiti.cfg.xml, 并且 bean 的 id 必须是 processEngineConfiguration
    ProcessEngine processEngine= ProcessEngines.getDefaultProcessEngine();
    System.out.println(processEngine);
}
```

ProcessEngines.getDefaultProcessEngine() 方法读取的是 activiti.cfg.xml，且 bean 的 id 必须是 processEngineConfiguration。

使用 ProcessEngineConfiguration 创建：

```
ProcessEngineConfiguration processEngineConfiguration =
    ProcessEngineConfiguration.createProcessEngineConfigurationFromResource("activiti.
    cfg.xml");
```

```
ProcessEngine processEngine = processEngineConfiguration.buildProcessEngine();
System.out.println(processEngine);
```

2.3 数据库说明

Activiti 的后台是有数据库的支持，所有的表都以 ACT_开头。 第二部分是表示表的用途的两个字母标识。用途也和服务的 API 对应。

ACT_RE_*: 'RE' 表示 repository。 这个前缀的表包含了流程定义和流程静态资源（图片，规则，等等）。

ACT_RU_*: 'RU' 表示 runtime。 这些运行时的表，包含流程实例，任务，变量，异步任务，等运行中的数据。 Activiti 只在流程实例执行过程中保存这些数据， 在流程结束时就会删除这些记录。 这样运行时表可以一直很小速度很快。

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

ACT_HI_*: 'HI' 表示 history。 这些表包含历史数据，比如历史流程实例， 变量，任务等等。

ACT_GE_*: 通用数据， 用于不同场景下，如存放资源文件。

资源库流程规则表

- 1) act_re_deployment 部署信息表
- 2) act_re_model 流程设计模型部署表
- 3) act_re_procdef 流程定义数据表
- 4) act_re_event_subscr 事件监听

运行时数据库表

- 1) act_ru_execution 运行时流程执行实例表
- 2) act_ru_identitylink 运行时流程人员表，主要存储任务节点与参与者的相关信息
- 3) act_ru_task 运行时任务节点表
- 4) act_ru_variable 运行时流程变量数据表
- 5) act_ru_job 异步作业

历史数据库表

- 1) act_hi_actinst 历史节点表
- 2) act_hi_attachment 历史附件表
- 3) act_hi_comment 历史意见表，评论
- 4) act_hi_identitylink 历史流程人员表
- 5) act_hi_detail 历史详情表，提供历史变量的查询
- 6) act_hi_procinst 历史流程实例表
- 7) act_hi_taskinst 历史任务实例表
- 8) act_hi_varinst 历史变量表

组织机构表

- 1) act_id_group 用户组信息表
- 2) act_id_info 用户扩展信息表
- 3) act_id_membership 用户与用户组对应信息表

4) `act_id_user` 用户信息表

这四张表很常见，基本的组织机构管理，关于用户认证方面建议还是自己开发一套，组件自带的功能太简单，使用中有很多需求难以满足

通用数据表

1) `act_ge_bytearray` 二进制数据表，所有二进制内容保存到这个表，比如 bpmn 文件。

2) `act_ge_property` 属性数据表存储整个流程引擎级别的数据，初始化表结构时，会默认插入三条记录，

其它数据表

1) `act_evt_log` 事件日志，默认不开启

2) `act_procdef_info` 流程定义的动态变更信息

2.4 Activiti 对象说明

a) 几个和流程相关的对象

Deployment: 部署对象，和部署表(`act_re_deployment`)对应

ProcessDefinition: 流程定义对象，和流程定义表(`act_re_procdef`)对应

ProcessInstance: 流程实例对象，和流程实例表(`act_ru_execution`)对应

Task: 任务对象，和任务表(`act_ru_task`)对应

b) 几个 Service 对象

RepositoryService: 操作部署、流程定义等静态资源信息

RuntimeService: 操作流程实例，启动流程实例、查询流程实例、删除流程实例等动态信息

TaskService: 操作任务，查询任务、办理任务等和任务相关的信息

HistoryService: 操作历史信息的，查询历史信息

IdentityService: 操作用户和组

c) 几个 Query 对象

DeploymentQuery: 对应查询部署表(`act_re_deployment`)

ProcessDefinitionQuery: 对应查询流程定义表(`act_re_procdef`)

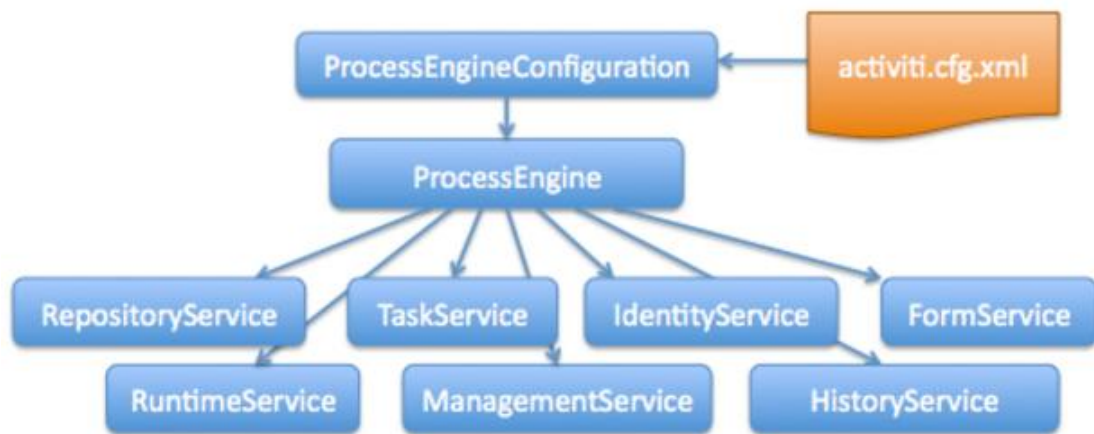
ProcessInstanceQuery: 对应查询流程实例表(`act_ru_execution`)

TaskQuery: 对应查询任务表(`act_ru_task`)

bpmn 通过 RepositoryService 部署，引擎会把 bpmn 解析成可执行的东西，生成一个 Deployment(部署)，对应的生成流程定义(ProcessDefinition)。

通过 RuntimeService 启动流程定义，得到一个流程实例。同时会生 Task(任务)，任务是绑定到用户的，记录当前操作者是谁，任务状态是什么，用 TaskService 操作任务。

3.Service 服务



所有的 Service 都通过流程引擎获得。

3.1 RepositoryService

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（bpmn 和图片），查询流程定义信息等操作，是引擎中的一个重要的服务。部署流程定义操作的数据库表有：部署表(act_re_deployment)、流程定义表(act_re_procdef)和二进制表(act_ge_bytearray)

```

//获取仓库服务
RepositoryService repositoryService= processEngine.getRepositoryService();
//部署流程
Deployment deployment =repositoryService.createDeployment()
    .name("请假流程")
    .addClasspathResource("leave.bpmn")
    .deploy(); // 完成部署
ProcessDefinitionQuery query = repositoryService.createProcessDefinitionQuery();
// 根据流程定义的 key 来过滤
query.processDefinitionKey("leave2");
// 添加排序条件
query.orderByProcessDefinitionVersion().desc();
//分页查询，从哪开始，查询几条
query.listPage(0, 2);
// 查询的是所有的流程定义
List<ProcessDefinition> list = query.list();
for (ProcessDefinition pd : list) {
    System.out.println(pd.getId() + "-" + pd.getName() + "-" + pd.getVersion());
}
//根据 id 删除流程
repositoryService.deleteDeployment("1005");
  
```

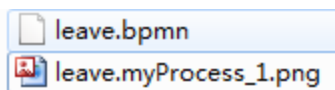

部署文件实际是一个 bpmn 文件和一个 png 图片

ID_	REV_	NAME_	DEPLOYMENT_ID_	BYTES_	GENERATED_
12502	1	leave.bpmn	12501	(BLOB)	0
12503	1	leave.myProcess_1.png	12501	(BLOB)	1

部署 bpmn 的时候，系统自动生成了一个 png 图片，可以下载查看：

```
RepositoryService repositoryService = processEngine.getRepositoryService();
String deploymentId = "72501";//部署 id
List<String> names = repositoryService.getDeploymentResourceNames(deploymentId);
for (String name : names) {
    System.out.println(name);
    // 获得两个流程定义文件对应的输入流
    InputStream in = repositoryService.getResourceAsStream(deploymentId, name);
    // 读取输入流写到指定的本地磁盘上
    FileCopyUtils.copy(in, new FileOutputStream("F://" + name));
    in.close();
}
```

去 F 盘下可以看到两个文件：



自动生成的 png 图片中文会乱码，解决这个问题需要在 processEngineConfiguration 的 bean 中配置字体，如图：

```
<property name="activityFontName" value="宋体"/>
<property name="labelFontName" value="宋体"/>

<!--带数据源-->
<bean id="processEngineConfiguration" class="org.activiti.e
    <property name="dataSource" ref="dataSource"/>
    <property name="databaseSchemaUpdate" value="true"/>
    <property name="activityFontName" value="宋体"/>
    <property name="labelFontName" value="宋体"/>
</bean>
```

如果不想生成 png 图片，可以配置如下属性：

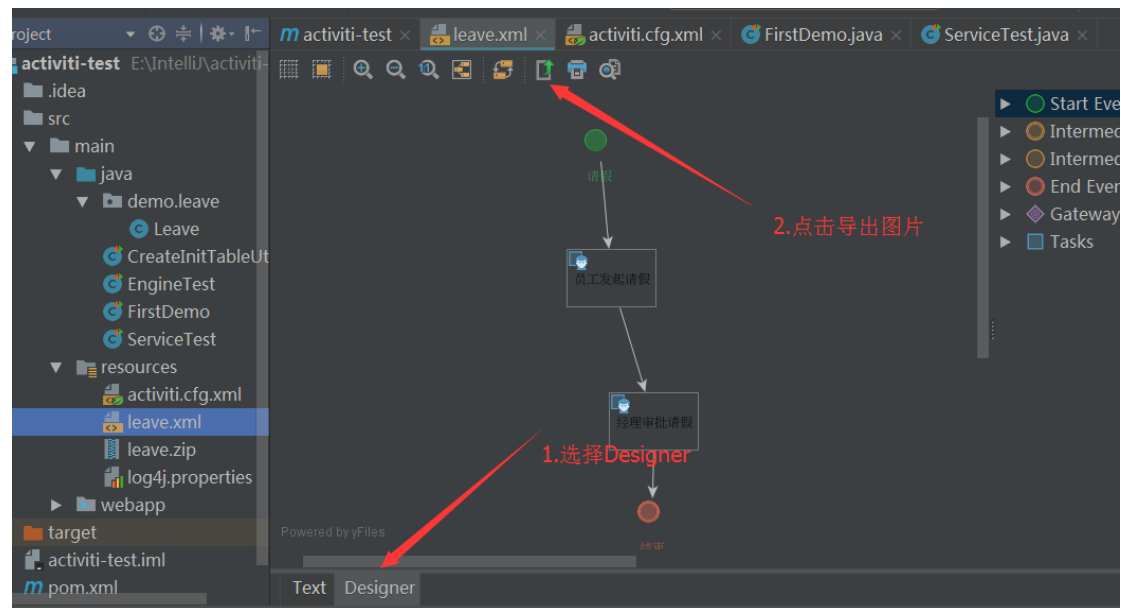
```
<property name="createDiagramOnDeploy" value="false" />
```

还可以将 png 和 bpmn 添加到 zip 压缩包中，部署 zip 压缩包。

```
// 从类路径下读取 leave.zip 压缩文件，并把它包装成一个输入流
ZipInputStream zipInputStream = new ZipInputStream(this.getClass()
    .getClassLoader().getResourceAsStream("leave.zip"));
Deployment deployment = repositoryService.createDeployment()
    .addZipInputStream(zipInputStream)
    .deploy(); // 完成部署
```

Activiti 会把压缩包内的 bpmn 和 png 添加到数据库，即数据库里存的是解压后的两个文件而不是 zip。

使用 idea 生成 bpmn 对应的 png，需要把 bpmn 的文件后缀名改成 xml



根据流程定义 id 获取文件流：

```
String processDefinitionId = "leave2:5:62504"; // 流程定义 id
// 直接获得 png 图片的名称
// 根据流程定义 id 查询流程定义对象
ProcessDefinitionQuery query =
    repositoryService.createProcessDefinitionQuery();
query.processDefinitionId(processDefinitionId);
ProcessDefinition processDefinition = query.singleResult();
// 根据流程定义对象获得 png 图片的名称,getResourceName()是获得 bpmn
String pngName = processDefinition.getDiagramResourceName();
// 直接获得 png 图片对应的输入流
InputStream pngStream =
    repositoryService.getProcessDiagram(processDefinitionId);
// 读取输入流写到指定的本地磁盘上
FileCopyUtils.copy(pngStream, new FileOutputStream("F:" + pngName));
pngStream.close();
```

3.2 RuntimeService

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的服务，启动流程实例操作的数据表有流程实例表(act_ru_execution)、任务表(act_ru_task)。

```
RuntimeService runtimeService = processEngine.getRuntimeService();
//key 是 act_re_procdef 中的 KEY_,即 bpmn 文件的 id
runtimeService.startProcessInstanceByKey("leave2");
```

```
//或者根据流程定义 id 启动, id 是 act_re_procdef 中的 ID_, 推荐使用
runtimeService.startProcessInstanceById("leave2:5:62504");
```

一个 KEY_对应多个 ID_, 即一个流程定义可以产生多个流程实例。

```
//查询流程, 操作的是流程实例表(act_ru_execution)
ProcessInstanceQuery query = runtimeService.createProcessInstanceQuery();
List<ProcessInstance> list = query.list();
```

删除一个流程实例

```
String processInstanceId = "1001"; // 流程实例 id
String deleteReason = "不请假了"; // 删除原因, 任君写
runtimeService.deleteProcessInstance(processInstanceId, deleteReason);
```

3.3 TaskService

对任务进行查询、接收、办理、完成等操作。查询任务操作的数据表是任务表(act_ru_task)。

```
TaskService taskService = processEngine.getTaskService();
//创建查询对象, 指定查询人, 进行查询
List<Task> taskList
=taskService.createTaskQuery().taskAssignee("ZhangSan").list();
```

```
//提交任务到下一个代理人
taskService.complete(task.getId());
```

listPage(offset, pageSize)用于分页查询

3.4 IdentityService

流程运行过程中的一些用户信息, 组信息等操作使用认证服务, 但是认证服务一般只作为辅助, 每一个系统都有一个比较完整的人员系统, 创建用户和用户组等操作。一般不使用自带的认证。

```
//添加用户组
IdentityService identityService = processEngine.getIdentityService();
Group groupEntity = identityService.newGroup("1001");
groupEntity.setName("超级管理员");
groupEntity.setType("administrator");
identityService.saveGroup(groupEntity); //保存用户组
//删除用户组
identityService.deleteGroup("1");

//添加用户
User user = identityService.newUser("10001");
user.setEmail("admin@sina.com");
user.setFirstName("zhang");
user.setLastName("san");
```

```
user.setPassword("admin");  
identityService.saveUser(user); //保存用户  
identityService.deleteUser("10001"); //删除用户
```

3.5 HistoryService

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

```
HistoryService historyService = processEngine.getHistoryService();  
//查询历史运行信息  
HistoricProcessInstance historicProcessInstance =  
    historyService  
        .createHistoricProcessInstanceQuery()  
        .processInstanceId("40001").singleResult();  
System.out.println("开始时间: " + historicProcessInstance.getStartTime());  
System.out.println("结束时间: " + historicProcessInstance.getEndTime());
```

3.6 FormService

可选服务，任务表单管理
后续讲解

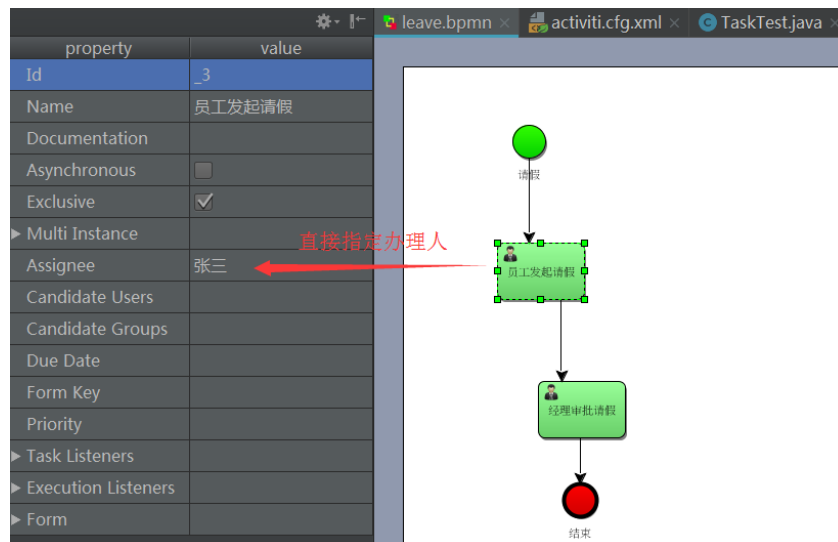
3.7 ManagementService

Management Service 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。

```
ManagementService managementService = processEngine.getManagementService();  
String taskTable = managementService.getTableName(Task.class);  
System.out.println(taskTable);
```

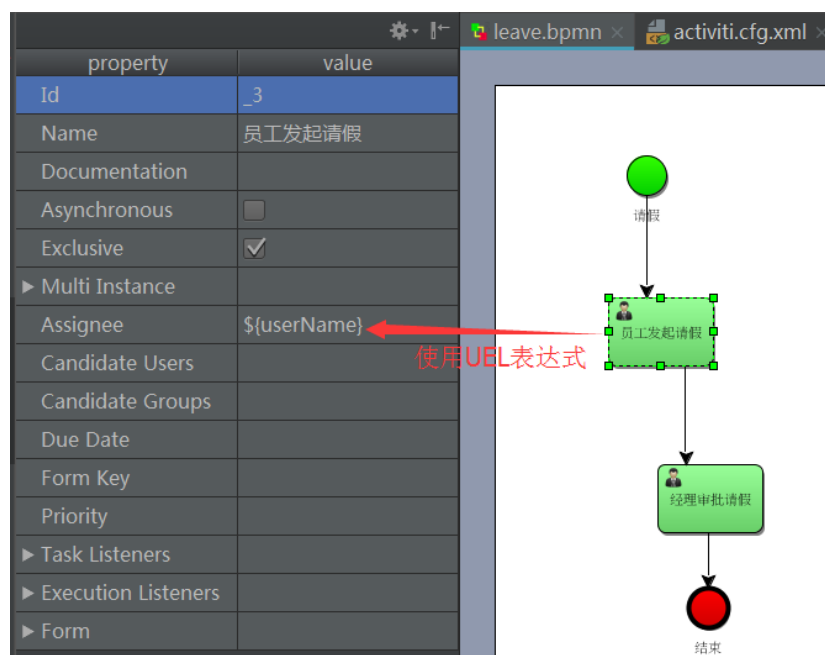
4.任务管理

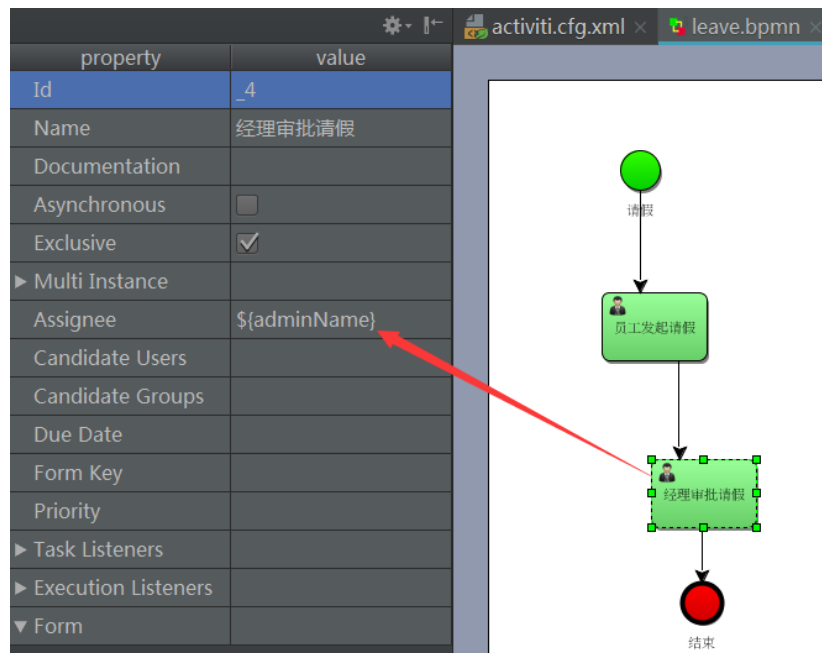
4.1 指定办理人



4.2 使用 UEL

4.2.1 UEL-VALUE



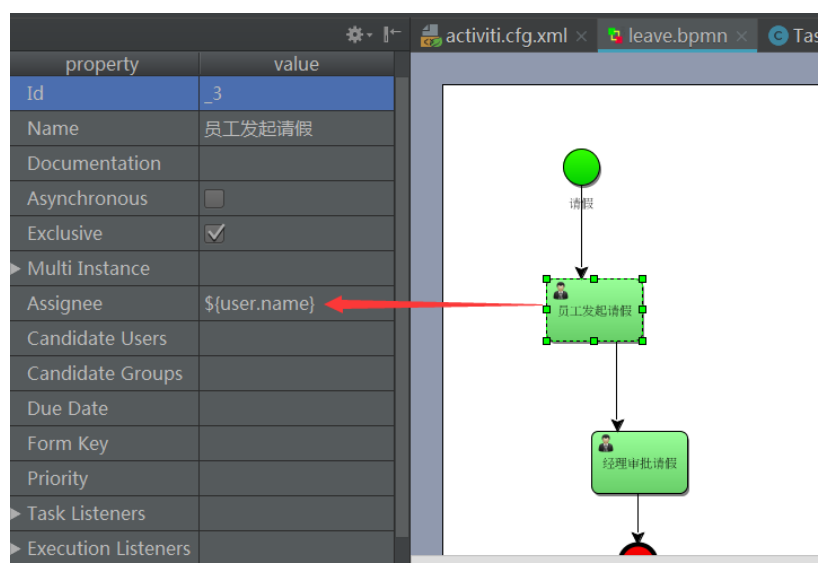


```
Map<String, Object> variable = new HashMap<>();
//指定当前节点的办理人是 WangWu
variable.put("userName", "WangWu");
runtimeService.startProcessInstanceId("leave:5:110003", variable);
```

执行任务

```
List<Task> taskList = taskService.createTaskQuery().taskAssignee("WangWu").list();
for (Task task : taskList) {
    //提交任务到下一个代理人
    Map<String, Object> variable = new HashMap<>();
    //指定处理人是 ZhaoLiu
    variable.put("adminName", "ZhaoLiu");
    taskService.complete(task.getId(), variable);
}
```

也可以绑定 pojo 实体类



注意实体类必须实现 java.io.Serializable 接口

```
public class User implements Serializable {
    private Integer id;
    private String name;
    //getter/setter 方法略
}
```

```
Map<String, Object> variable = new HashMap<>();
//绑定实体类属性, ${user.name} 就会调用 user 对象的 getName() 方法
User user = new User();
user.setName("小明");
variable.put("user", user);
runtimeService.startProcessInstanceId("leave:6:127503", variable);
```

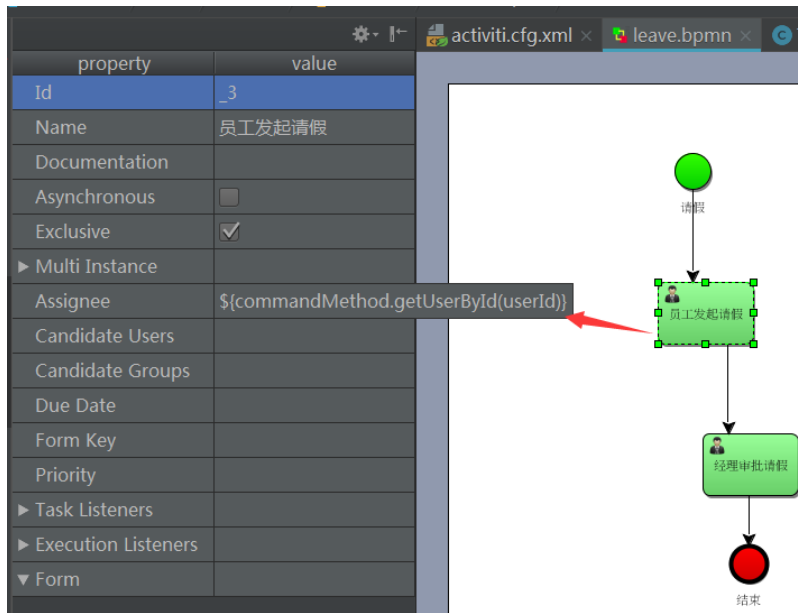
4.2.2 UEL-METHOD

执行步骤

- 1) 设置节点的执行人为\${bean.methodName(param)}，其中 bean 方法是我们注入到 spring 中的一个 bean 的 id，methodName 是 bean 对应的类中的方法名。
- 2) 将 method 方法注入到 activiti 的 processEngineConfiguration 的 bean 中（在我们的 activiti.cfg.xml 中）
- 3) 启动一个流程设置全局变量 param 作为启动参数。

```
package demo.leave;

public class CommandMethod {
    public String getUserById(int userId) {
        //模拟数据库中查询的数据
        return "activiti"+userId;
    }
}
```



```
<bean id="commandMethod" class="demo.leave.CommandMethod"/>
<!--带数据源-->
<bean id="processEngineConfiguration"
class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
    <property name="dataSource" ref="dataSource"/>
    <property name="databaseSchemaUpdate" value="true"/>
    <property name="beans">
        <map>
            <entry key="commandMethod" value-ref="commandMethod"></entry>
        </map>
    </property>
</bean>
```

```
Map<String, Object> variable = new HashMap<>();
//绑定方法中参数
variable.put("userId", 123);
runtimeService.startProcessInstanceId("leave:10:167503", variable);
```

4.3 使用监听器

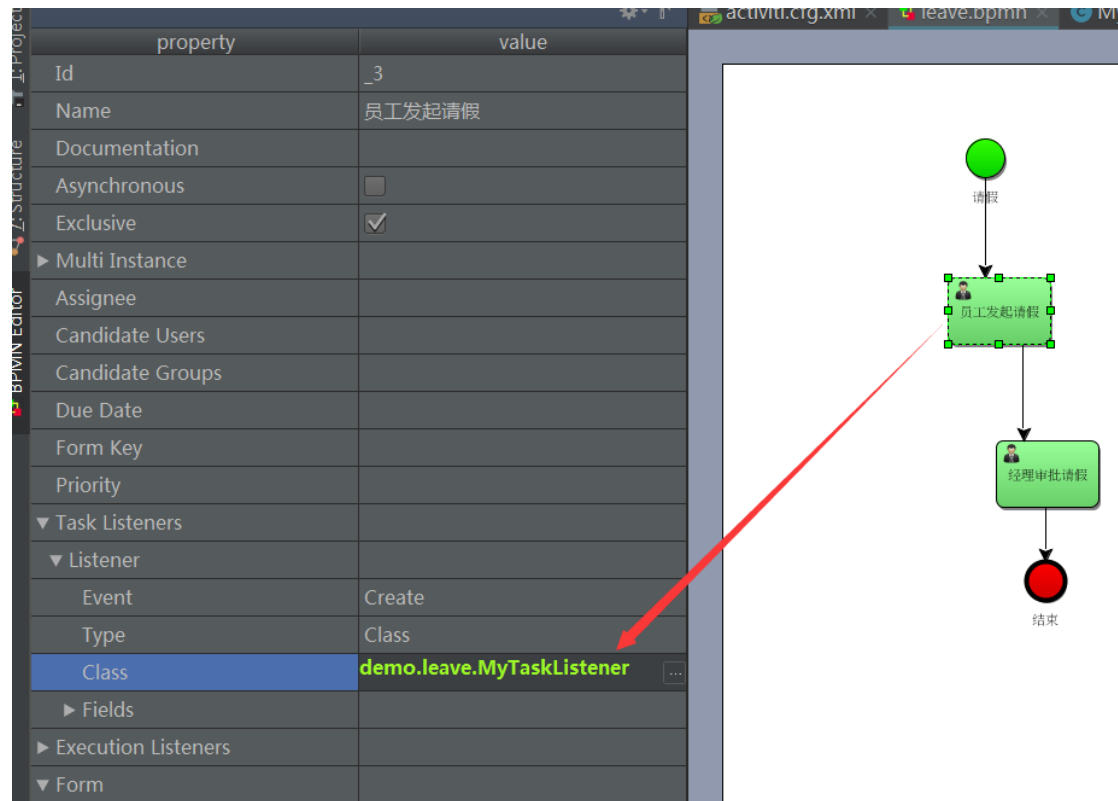
```
package demo.leave;

import org.activiti.engine.delegate.DelegateTask;
import org.activiti.engine.delegate.TaskListener;

public class MyTaskListener implements TaskListener {
```

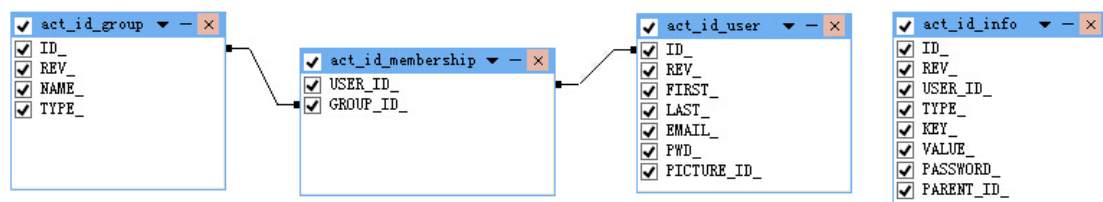


```
//设置待办事宜的处理人
@Override
public void notify(DelegateTask delegateTask) {
    delegateTask.setAssignee("zhangsan");
}
}
```



发布之后启动流程，会自动设定处理人。

5. 用户与用户组



activiti 的用户表和其它表没有外键关系，方便进行扩展。activiti 的用户表提供了简单的信息，和其它框架整合的时候，可以使用自己定义的表。

5.1 用户组

```
IdentityService identityService = processEngine.getIdentityService();
GroupQuery groupQuery = identityService.createGroupQuery();
List<Group> datas = groupQuery.list();
//其他类似查询方法
groupQuery.listPage(1, 2); //分页返回查询结果; 偏移量, 每页几条
groupQuery.count(); //结果总数
groupQuery.groupName("超级管理员").list(); //根据 name 查询
groupQuery.groupNameLike("%超级 A%").list(); //根据 name 模糊查询
groupQuery.groupType("typeA").list(); //根据 type 查询
groupQuery.groupId("1").singleResult(); //根据 id 查询, 查询结果唯一
groupQuery.orderByGroupId().asc().list(); //结果升序
groupQuery.orderByGroupName().desc().list(); //结果降序
```

5.2 用户

```
User user = identityService.newUser("10001");
user.setEmail("zhangsan@xx.com");
user.setFirstName("zhang");
user.setLastName("san");
user.setPassword("admin");
identityService.saveUser(user); //保存用户
identityService.deleteUser("10001"); //删除用户
//验证用户密码, 参数用户 id 和密码
boolean flag = identityService.checkPassword("10001", "123456");
//设置用户 info 信息
identityService.setUserInfo("用户 id", "INFO 表 KEY", "INFO 表 VALUE");
//查询相关信息.
identityService.getUserInfo("INFO 表 USER_ID", "INFO 表 KEY");
identityService.createMembership("用户 ID", "组 ID"); //给用户添加分组
identityService.deleteMembership("用户 ID", "组 ID"); //删除用户和组关系
//查询某用户的分组
List<Group> datas = identityService.createGroupQuery().groupMember("userId").list();
//查询某组下的用户
List<User> datas =
identityService.createUserQuery().memberOfGroup("groupId").list();
```

6. 表单

除了使用 **Assignee** 保存任务操作者，每个任务节点可以保存一些简单的信息。表单能保存的数据较少，可以自己创建表保存数据。

6.1 内置表单

property	value
Id	_4
Name	发布请假
Documentation	
Asynchronous	<input type="checkbox"/>
Exclusive	<input checked="" type="checkbox"/>
Multi Instance	
Assignee	\${userName}
Candidate Users	
Candidate Groups	
Due Date	
Form Key	
Priority	
Task Listeners	
Execution Listeners	
Form	
Form Property	
Id	startDate
Name	请假日期
Type	date
Expression	
Variable	
Default	
Date Pattern	yyyy-MM-dd
Readable	
Writable	
Required	True

```
<userTask activiti:assignee="${userName}" activiti:exclusive="true" id="_4" name="
发布请假">
  <extensionElements>
    <activiti:formProperty datePattern="yyyy-MM-dd" id="startDate" name="请假日期"
required="true" type="date"/>
    <activiti:formProperty id="reason" name="请假原因" required="true" type="string"/>
  </extensionElements>
</userTask>
<userTask activiti:assignee="${adminName}" activiti:exclusive="true" id="_5" name="
审批请假">
  <extensionElements>
```

```

    <activiti:formProperty id="advise" name="意见" type="enum">
      <activiti:value id="true" name="同意"/>
      <activiti:value id="false" name="拒绝"/>
    </activiti:formProperty>
  </extensionElements>
</userTask>

```

操作表单：

```

ProcessDefinition definition = processEngine.getRepositoryService()
    .createProcessDefinitionQuery().processDefinitionKey("leave_mybatis").late
stVersion().singleResult();
FormService formService = processEngine.getFormService();
TaskService taskService = processEngine.getTaskService();
RuntimeService runtimeService = processEngine.getRuntimeService();
Map<String, String> param1 = new HashMap<>();
param1.put("userName", "user1");
//模拟当前请假条的 id, 保证 task 能查出唯一的任务。当然也可以查出一个 task list 批量处
理
String businessKey = System.currentTimeMillis() + "";
//启动任务
ProcessInstance processInstance =
formService.submitStartFormData(definition.getId(), businessKey, param1);
//查询任务
Task task =
taskService.createTaskQuery().taskAssignee("user1").processDefinitionKey("leave_my
batis").processInstanceBusinessKey(businessKey).singleResult();
//查询任务中的表单信息
TaskFormData formData = formService.getTaskFormData(task.getId());
for (FormProperty property : formData.getFormProperties()) {
    System.out.println(property.getName() + "=" + property.getType());
}
Map<String, String> variable = new HashMap<>();
variable.put("reason", "有事");
variable.put("startDate", "2018-01-01");
variable.put("adminName", "admin1");
//提交表单(完成当前节点的任务)
formService.submitTaskFormData(task.getId(), variable);

//查询出下一节点的任务
Task task2 =
taskService.createTaskQuery().taskAssignee("admin1").processDefinitionKey("leave_m
ybatis").processInstanceBusinessKey(businessKey).singleResult();
//查询出任务中的参数
Map<String, Object> variables = runtimeService.getVariables(processInstance.getId())
for (String key : variables.keySet()) {

```

```

        System.out.println(key + "=" + variables.get(key));
    }
    //完成任务
    Map<String, String> variable2 = new HashMap<>();
    variable2.put("advise", "true");
    formService.submitTaskFormData(task2.getId(), variable2);

```

6.2 外置表单

外置表单可以自定义一个代码片段，如 html 片段，另存为 leave_apply.form:

```

<div class="control-group">
    <label class="control-label" for="startDate">开始时间: </label>
    <div class="controls">
        <input type="text" id="startDate" name="startDate" class="datepicker"
data-date-format="yyyy-mm-dd" required />
    </div>
</div>
<div class="control-group">
    <label class="control-label" for="reason">请假原因: </label>
    <div class="controls">
        <textarea id="reason" name="reason" required></textarea>
    </div>
</div>

```

property	value
Id	_4
Name	UserTask
Documentation	
Asynchronous	<input type="checkbox"/>
Exclusive	<input checked="" type="checkbox"/>
Multi Instance	
Assignee	
Candidate Users	
Candidate Groups	
Due Date	
Form Key	leave_apply.form
Priority	
Task Listeners	
Execution Listeners	
Form	

```

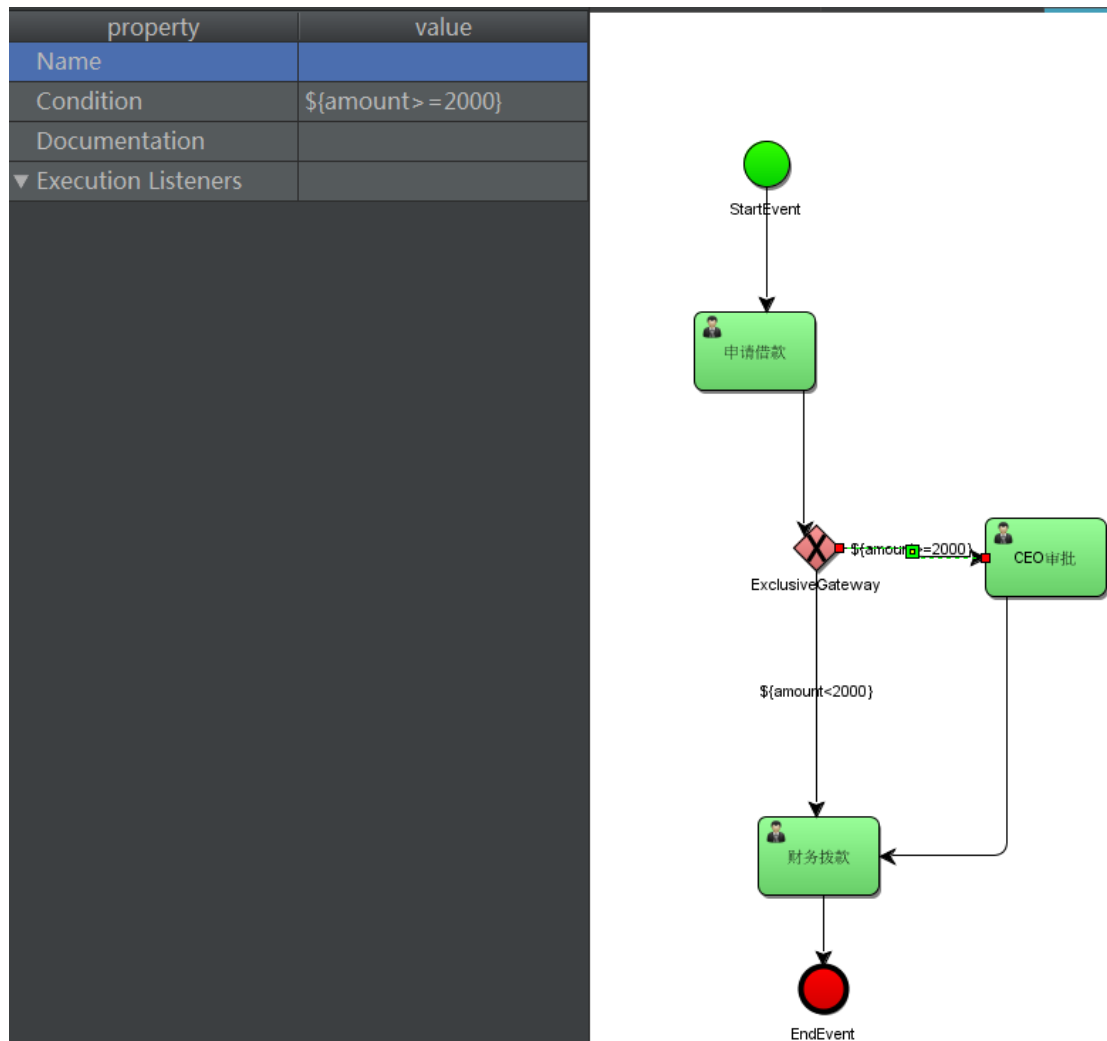
Deployment deployment = processEngine.getRepositoryService() // 与流程定义和部署对象相关的 Service
    .createDeployment() // 创建一个部署对象

```

```
.name("1801 请假流程-外置表单") // 设置对应流程的名称
.addClasspathResource("leave7.bpmn").addClasspathResource("leave_apply.form")
.deploy(); // 完成部署
```

```
Map<String, String> param1 = new HashMap<>();
param1.put("userName", "user1");
//模拟当前请假条的 id, 保证 task 能查出唯一的任务。当然也可以查出一个 task list 批量处理
String businessKey = System.currentTimeMillis() + "";
//启动任务
ProcessInstance processInstance =
formService.submitStartFormData(definition.getId(), businessKey, param1);
//查询任务
Task task =
taskService.createTaskQuery().taskAssignee("user1").processDefinitionKey("myProcess_1").processInstanceBusinessKey(businessKey).singleResult();
//查询出 html 片段, 可以显示在页面上
Object renderedStartForm = formService.getRenderedTaskForm(task.getId());
System.out.println(renderedStartForm);
```

7. 网关



网关类似于程序中的 if 判断，当满足某一条件时，任务进入下一个指定的节点。
网关有：排他网关、并行网关、包含网关、基于事件网关。

7.1 排他网关

排他网关只会选择一条顺序流执行。就是说，虽然多个顺序流的条件结果为 true，那么 XML 中的第一个顺序流（也只有这一条）会被选中，并用来继续运行流程。如果没有选中任何顺序流，会抛出一个异常。排他网关类似于 if-elseif

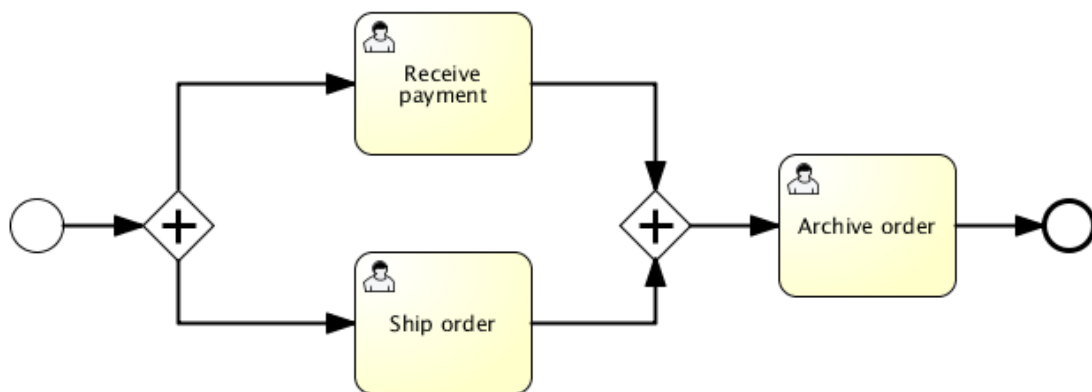
```

ProcessDefinition definition = processEngine.getRepositoryService()
    .createProcessDefinitionQuery().processDefinitionKey("loan_gateway").latestVersion().singleResult();
Map<String, Object> param1 = new HashMap<>();
param1.put("amount", "5500");
//模拟当前借款单 id, 保证 task 能查出唯一的任务。当然也可以查出一个 task list 批量处
  
```

理

```
String businessKey = System.currentTimeMillis() + "";
//启动任务
ProcessInstance processInstance =
    runtimeService.startProcessInstanceByKey("loan_gateway", businessKey,
param1);
//查询任务
Task task = taskService.createTaskQuery()
    .processDefinitionKey("loan_gateway")
    .processInstanceBusinessKey(businessKey).singleResult();
taskService.complete(task.getId());
```

7.2 并行网关



网关也可以表示流程中的并行情况。最简单的并行网关是 **并行网关**，它允许将流程分成多条分支，也可以把多条分支汇聚到一起。

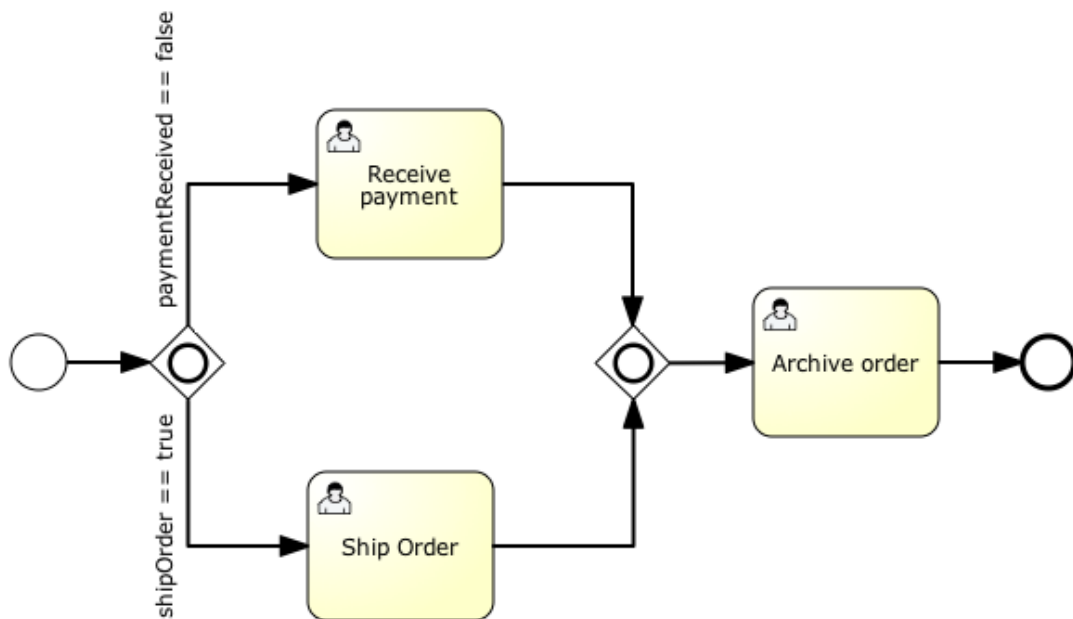
并行网关的功能是基于进入和外出的顺序流的：

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

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

与其他网关的主要区别是，并行网关不会解析条件。即使顺序流中定义了条件，也会被忽略。

7.3 包含网关



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

包含网关的功能是基于进入和外出顺序流的：

- **分支：** 所有外出顺序流的条件都会被解析，结果为 true 的顺序流会以并行方式继续执行，会为每个顺序流创建一个分支。
- **汇聚：** 所有并行分支到达包含网关，会进入等待章台，直到每个包含流程 token 的进入顺序流的分支都到达。这是与并行网关的最大不同。换句话说，包含网关只会等待被选中执行了的进入顺序流。在汇聚之后，流程会穿过包含网关继续执行。

注意，如果同一个包含节点拥有多个进入和外出顺序流，它就会同时**含有分支和汇聚功能**。这时，网关会先汇聚所有拥有流程 token 的进入顺序流，再根据条件判断结果为 true 的外出顺序流，为它们生成多条并行分支。

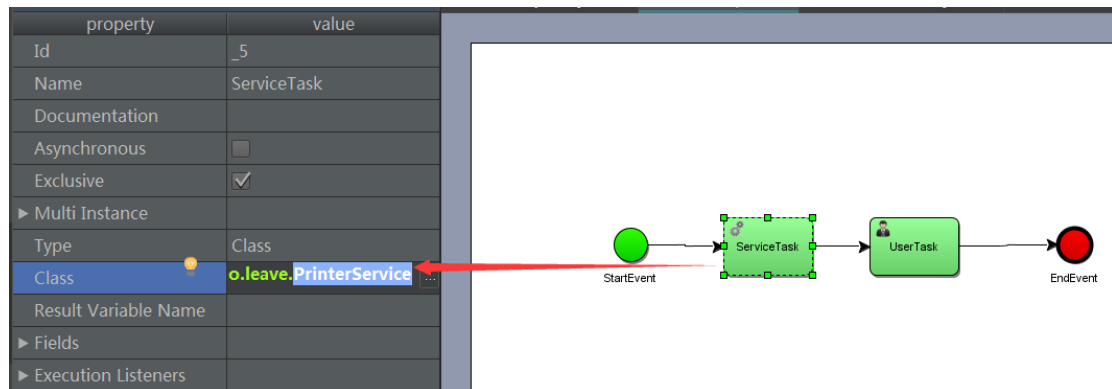
8.ServiceTask

定义 Service 类。

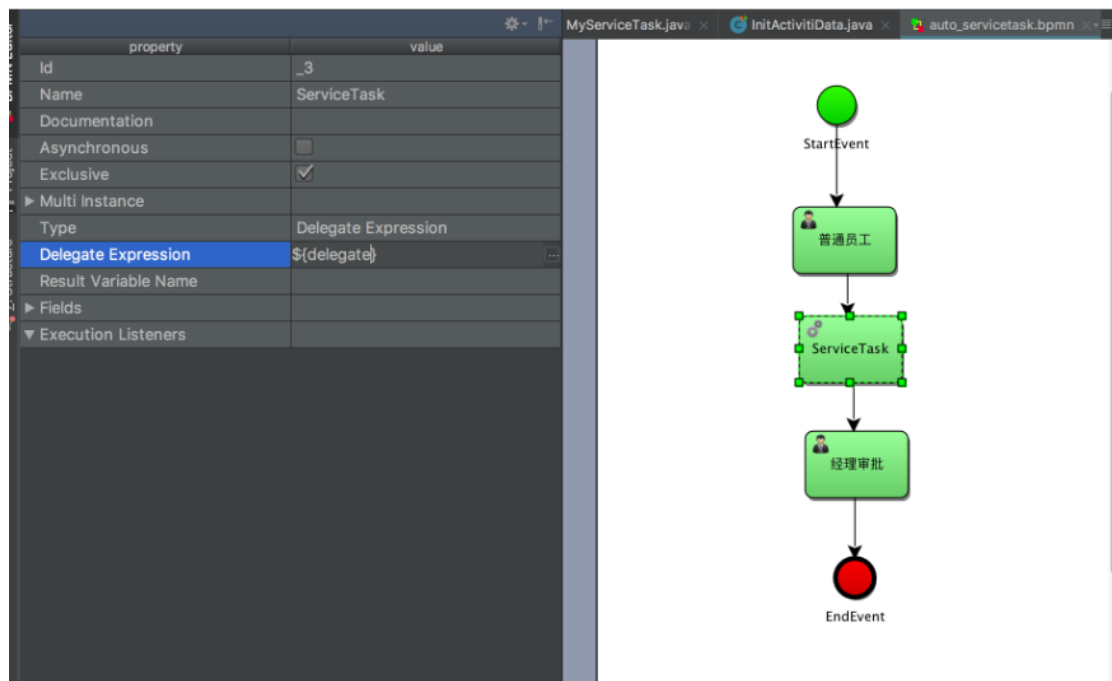
```
import org.activiti.engine.delegate.DelegateExecution;  
import org.activiti.engine.delegate.JavaDelegate;
```

```
import java.util.Map;

public class PrinterService implements JavaDelegate {
    @Override
    public void execute(DelegateExecution delegateExecution) throws Exception {
        Map<String, Object> variable = delegateExecution.getVariables();
        System.out.println(variable.get("name"));
    }
}
```



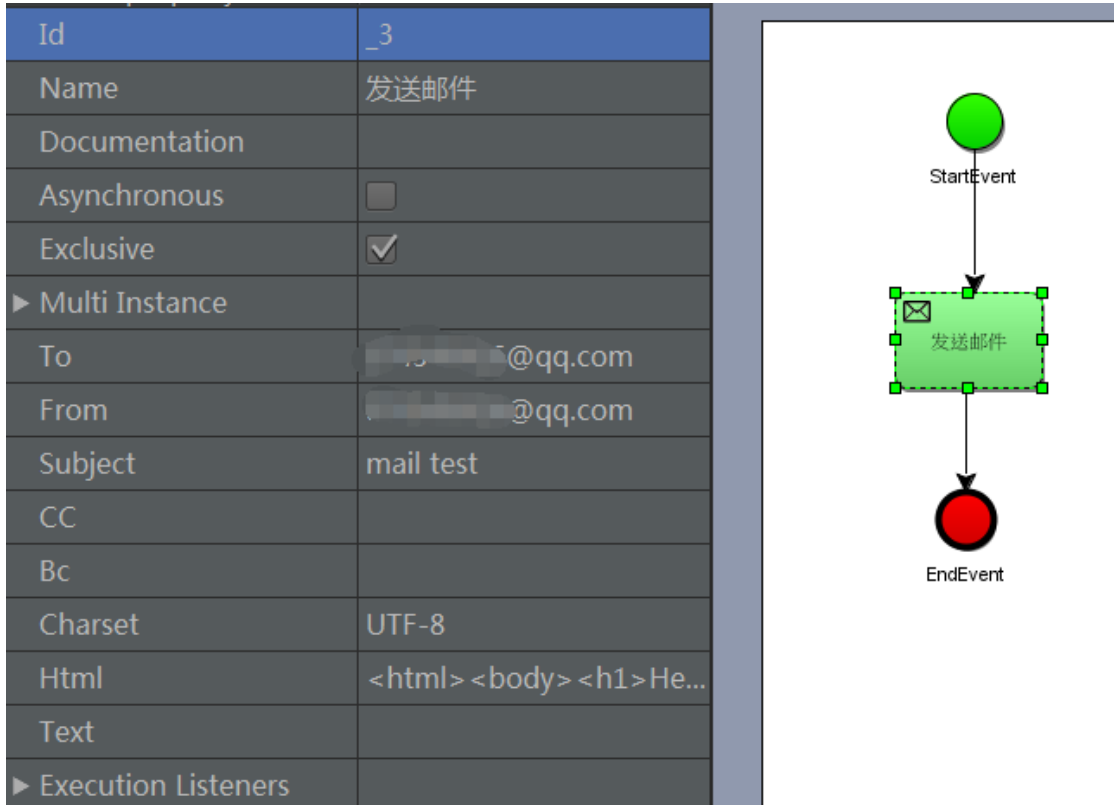
启动任务时，PrinterService 会自动执行。



```
Map<String, Object> param1 = new HashMap<>();
param1.put("delegate", new PrinterService());
param1.put("name", "abcd");
param1.put("userId", "123");
ProcessInstance processInstance =
    runtimeService.startProcessInstanceByKey("ServiceTask", param1);
```

9.MailTask 邮件任务

使用 QQ 邮箱测试邮件服务，默认 QQ 邮箱是不开启的，



```
<bean id="processEngineConfiguration"
class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
  <property name="dataSource" ref="dataSource"/>
  <property name="databaseSchemaUpdate" value="true"/>
```

```

<property name="mailServerHost" value="smtp.qq.com"/>
<property name="mailServerPort" value="465"/>
<property name="mailServerDefaultFrom" value="xxxx@qq.com"/>
<property name="mailServerUsername" value="xxxx@qq.com"/>
<property name="mailServerPassword" value="xxxxx"/>
<property name="mailServerUseSSL" value="true"/>
</bean>

```

其中的 mailServerPassword 是 QQ 邮箱验证成功后生成的授权码

10. BPMN 规范

业务流程模型注解（Business Process Modeling Notation - BPMN）是业务流程模型的一种标准图形注解。这个标准 是由对象管理组（Object Management Group - OMG）维护的。包含以下部分：

- 流对象：事件、活动、网关
- 连接对象：序列流、消息流、关联
- 泳道：池、道
- 人工制品：数据对象、组、注释



10.1 事件

启动事件：

```
<startEvent id="start" name="myStart" />
```

结束事件：

```
<endEvent id="end" name="myEnd" />
```

10.2 网关

包含网关：

```
<inclusiveGateway id="inclusiveGatewaySplit" default="flow3"/>
```

排他网关：

```
<exclusiveGateway id="decision" name="decideBasedOnAmountAndBankType" default="myFlow"/>
```

并行网关：

```
<parallelGateway id="myParallelGateway" name="My Parallel Gateway" />
```

10.3 任务

人工任务：

```
<userTask id="myTask" name="My task" />
```

服务任务：

```
<serviceTask id="MyServiceTask" name="My service task"
  implementation="Other" operationRef="myOperation" />
```

11. 与 Spring 和 Mybatis 整合

```
<bean id="activitiDataSource" class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
  <property name="url"
value="jdbc:mysql://localhost:3306/activiti?characterEncoding=utf-8"/>
  <property name="username" value="root"/>
  <property name="password" value="" />
</bean>
```

```

<!--配置事务管理器-->
<bean id="activitiTransactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="activitiDataSource"/>
</bean>

<!--流程引擎配置-->
<bean id="processEngineConfiguration"
class="org.activiti.spring.SpringProcessEngineConfiguration">
    <property name="dataSource" ref="activitiDataSource"/>
    <property name="transactionManager" ref="activitiTransactionManager"/>
    <property name="databaseSchemaUpdate" value="true"/>
    <property name="jobExecutorActivate" value="false"/>
</bean>


<bean id="processEngine" class="org.activiti.spring.ProcessEngineFactoryBean">
    <property name="processEngineConfiguration" ref="processEngineConfiguration"/>
</bean>

<!--获取各种服务-->
<bean id="repositoryService" factory-bean="processEngine"
factory-method="getRepositoryService"/>
<bean id="runtimeService" factory-bean="processEngine" factory-method="getRuntimeService"/>
<bean id="taskService" factory-bean="processEngine" factory-method="getTaskService"/>
<bean id="historyService" factory-bean="processEngine" factory-method="getHistoryService"/>
<bean id="managementService" factory-bean="processEngine"
factory-method="getManagementService"/>

```

整合思路

比如我们想把请假条保存到自己创建的表中,又想使用 activiti 操作请假流程,请假条表 LEAVE_APPLY

名	类型	长度	小数点	允许空值 (
> ID	int	11	0	<input type="checkbox"/>	 1
USER_ID	int	11	0	<input type="checkbox"/>	
REASON	varchar	255	0	<input type="checkbox"/>	
START_DATE	date	0	0	<input type="checkbox"/>	
END_DATE	date	0	0	<input type="checkbox"/>	
ADVISE	varchar	255	0	<input checked="" type="checkbox"/>	
STATUS	int	11	0	<input type="checkbox"/>	

添加一个请假条之后, 获得唯一标识符 (如 id)。

在启动 activiti 任务的时候, 可以传入一个 BUSINESS_ID, 可以把请假条的 id 当成 BUSINESS_ID 传递给 activiti。

```
runtimeService.startProcessInstanceByKey("ProcessDefinitionKey", "BUSINESS_KEY",  
variable);
```

在 act_ru_task 表中保存了 EXECUTION_ID_ 和 ASSIGNEE_ (处理人)

act_ru_execution 表中保存了 BUSINESS_KEY_

如果处理人指定的是任务组，那么在 act_ru_identitylink 中保存了 USER_ID_(处理人)和 TASK_ID_(任务 id)与 act_ru_task 表关联，其中处理人与我们自己设计的用户表关联。

如查询某个人要处理的请假条：

```
select  
L. ID, L. USER_ID, L. START_DATE, L. END_DATE, L. STATUS, L. REASON, L. ADVISE, L. TASK_ID, U. REAL_  
NAME  
from activiti.ACT_RU_TASK RES  
inner join activiti.ACT_RU_IDENTITYLINK I on I.TASK_ID_ = RES. ID_  
INNER JOIN activiti.act_ru_execution RU ON RES.EXECUTION_ID_=RU. ID_  
INNER JOIN test.leave_apply L ON RU.BUSINESS_KEY_=L. ID  
inner join ADMIN_USER U on L. USER_ID=U. ID  
WHERE RES. ASSIGNEE_ =#{userId}  
or(I. TYPE_ = 'candidate' and I. USER_ID_ = #{userId} )
```

其中#{userId}是任务的代办者