# EasyExcel

## 1. EasyExcel介绍

EasyExcel是一个基于Java的、快速、简洁、解决大文件内存溢出的Excel处理工具。他能让

你在不用考虑性能、内存的等因素的情况下，快速完成Excel的读、写等功能。EasyExcel

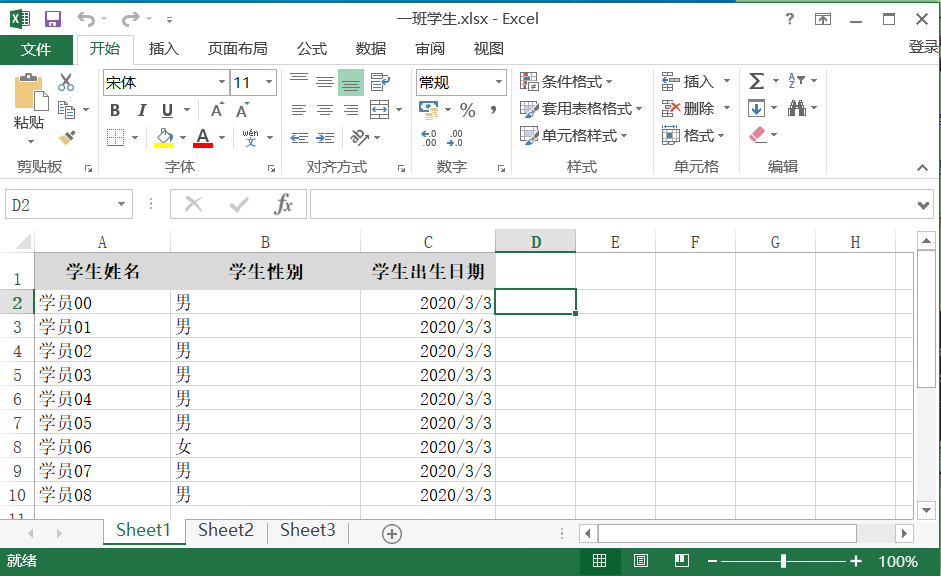
是Alibaba开源的操作excel的框架。

## 2. EasyExcel官网

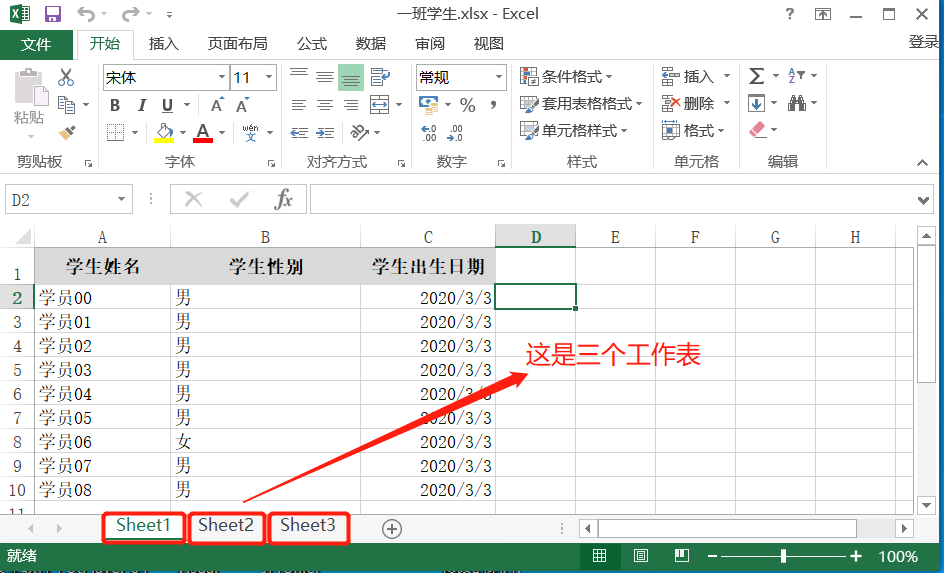
<https://easyexcel.opensource.alibaba.com/>

## 3.Excel文件结构

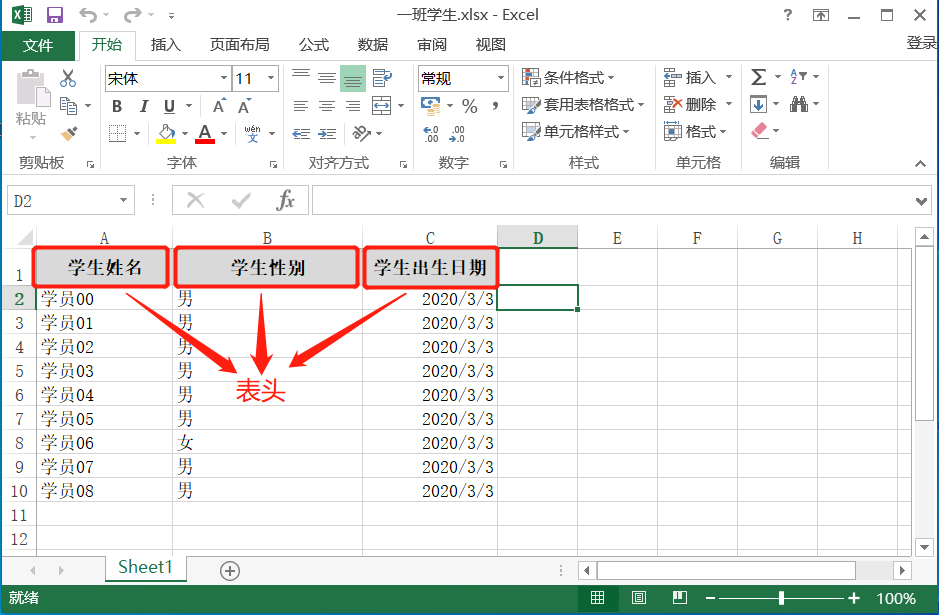
(1)一个Excel文件就是一个工作簿。图解：



(2)一个工作簿中可以有多个工作表(sheet)。图解：



(3)表头



## 4.使用EasyExcel读\*.xlsx格式的文件

(1)导入maven依赖

<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>easyexcel</artifactId>  
 <version>3.1.1</version>  
</dependency>

注意：这个依赖中包含有其他的jar包，比如：slf4j-api、lombok。

(2)创建一个实体类

@Data  
public class StudentForRead {  
 private String name;  
 private String gender;  
 private Date birthday;  
 private String id;  
}

(3)创建一个监听器类

public class StudentListener extends AnalysisEventListener<StudentForRead> {  
  
 */\*\*  
 \* 针对捕捉到异常的数据触发的函数，抛出异常时会停止读取  
 \** ***@param*** *exception  
 \** ***@param*** *context  
 \** ***@throws*** *Exception  
 \*/* @Override  
 public void onException(Exception exception, AnalysisContext context) throws Exception {  
  
 }  
  
 */\*\*  
 \* 每读一行内容，都会调用一次invoke方法，在invoke方法中可以操作读取到的内容  
 \** ***@param*** *studentForRead 每次读取到的数据封装的对象  
 \** ***@param*** *analysisContext  
 \*/* @Override  
 public void invoke(StudentForRead studentForRead, AnalysisContext analysisContext) {  
 System.*out*.println("读取的内容："+ studentForRead);  
 }  
  
 */\*\*  
 \* 读取第一行表头信息后调用的方法  
 \** ***@param*** *headMap  
 \** ***@param*** *context  
 \*/* @Override  
 public void invokeHeadMap(Map<Integer, String> headMap, AnalysisContext context) {  
 super.invokeHeadMap(headMap, context);  
 }  
  
 */\*\*  
 \* 读取额外信息：批注、超链接、合并单元格信息等  
 \** ***@param*** *extra  
 \** ***@param*** *context  
 \*/* @Override  
 public void extra(CellExtra extra, AnalysisContext context) {  
  
 }  
  
 */\*\*  
 \* 读取完整个工作簿后调用的方法  
 \** ***@param*** *analysisContext  
 \*/* @Override  
 public void doAfterAllAnalysed(AnalysisContext analysisContext) {  
 System.*out*.println("文件已读取完成！");  
 }  
  
 */\*\*  
 \* 返回值表示是否还有下一行数据  
 \** ***@param*** *context  
 \** ***@return*** *\*/* @Override  
 public boolean hasNext(AnalysisContext context) {  
 return true;  
 }  
  
}

(4)读取文件数据

@Test  
public void testRead(){  
 //获取一个工作簿对象，read还有更多重载方法  
 ExcelReaderBuilder workbook = EasyExcel.*read*("一班学生.xlsx", StudentForRead.class, new StudentListener());  
 //获取工作簿中的第一个工作表对象，sheet还有更多重载方法  
 ExcelReaderSheetBuilder sheet = workbook.sheet();  
 //读取工作表中的内容，每读一行，都会调用监听器的invoke方法  
 sheet.doRead();  
}

## 5.使用EasyExcel写\*.xlsx格式的文件

(1)导入maven依赖

<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>easyexcel</artifactId>  
 <version>3.1.1</version>  
</dependency>

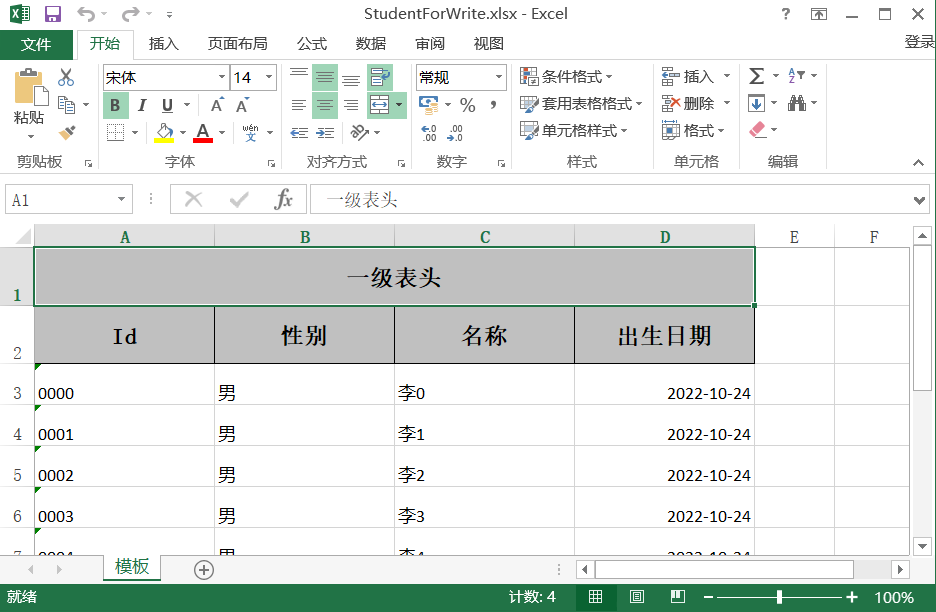
(2)创建一个实体类并使用相关注解

@Data  
@HeadRowHeight(35) //表头行高  
@ContentRowHeight(25) //内容行高  
@ColumnWidth(20) //列宽。也可以修饰属性，指定某列的宽度  
public class StudentForWrite {  
 */\*\*  
 \** ***@ExcelProperty注解用于设置标题和排序*** *\* value的值是一个字符串数组，其第一个值为一级表头，第二个值为二级表头，以此类推  
 \* order的值是int类型，为排序的值  
 \* index(未展示)的值是int类型，指定列的索引，跟order作用相似。  
 \* 不同之处是order没法设置空列，而指定index时，可以指定属性值  
 \* 在哪一列显示。比如：当index=0时，是在第一列展示；index=3时，  
 \* 是在第四列展示。  
 \*/* @ExcelProperty(value = {"一级表头","Id"},order = 1)  
 private String id;  
 @ExcelProperty(value = {"一级表头","名称"},order = 5)  
 private String name;  
 @ExcelProperty(value = {"一级表头","性别"},order = 2)  
 private String gender;  
 @ExcelProperty(value = {"一级表头","出生日期"},order = 4)  
 @DateTimeFormat("yyyy-MM-dd") //指定日期的格式  
 private Date birthday;  
  
 //忽略这个字段  
 @ExcelIgnore  
 private String ignore;  
}

(3)写入文件数据

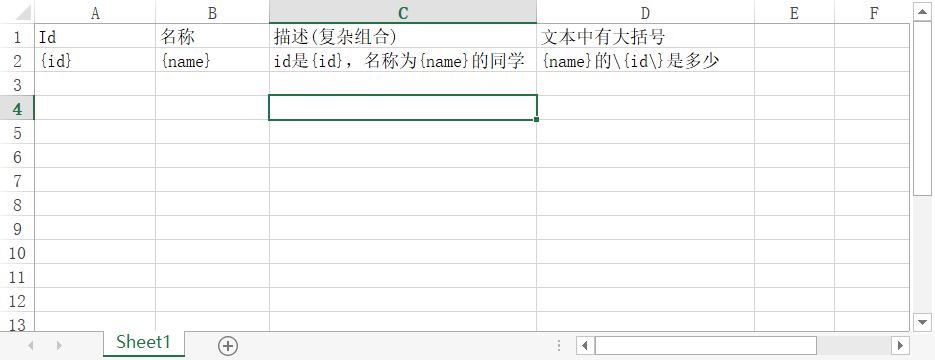
@Test  
public void testWrite(){  
 List<StudentForWrite> list = new ArrayList<>();  
 for(int i = 0;i < 10;i++){  
 StudentForWrite write = new StudentForWrite();  
 write.setId("000"+i);  
 write.setName("李"+i);  
 write.setGender("男");  
 write.setBirthday(new Date());  
 list.add(write);  
 }  
 Set<String> set = new HashSet<>();  
 set.add("id");  
 set.add("name");  
 String fileName = "StudentForWrite.xlsx";  
 //数据写入Excel文件，如果文件已存在，则先清空文件内容，再写入新数据  
 EasyExcel.*write*(fileName, StudentForWrite.class)  
 //表示execl文件只写入实体类中属性名和set集合中的值相同的列  
// .includeColumnFieldNames(set)  
 //(实体类不使用@ColumnWidth指定列宽时)自适应列宽，但不是特别精确  
// .registerWriteHandler(new LongestMatchColumnWidthStyleStrategy())  
 .sheet("模板").doWrite(list);  
}

(4)生成的文件内容展示



## 6.填充一组数据

(1)准备模板execl文件，预览图。



说明：{}用来包裹需要被替换的变量值，如果文本中本来就有{或者}，需要使用\来转义。

代码中用来填充数据的实体类对象的属性名或Map集合的key需要和{}中被包裹的变量

名一致。

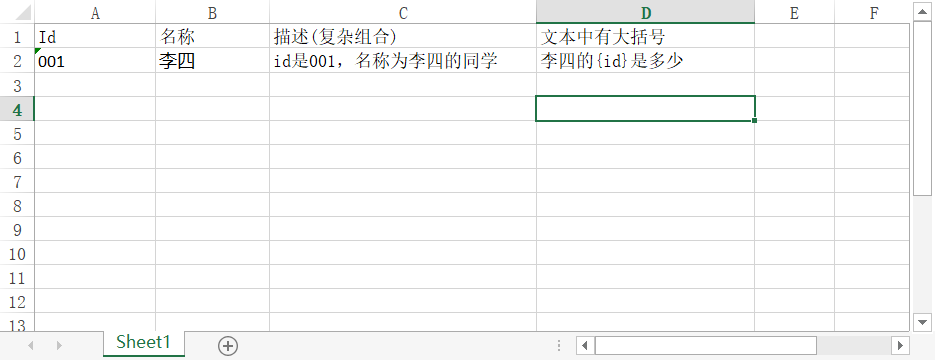
(2)填充数据的实体类(使用Map也可以)

@Data  
public class StudentForFill {  
 private String id;  
 private String name;  
}

(3)填充方法示例

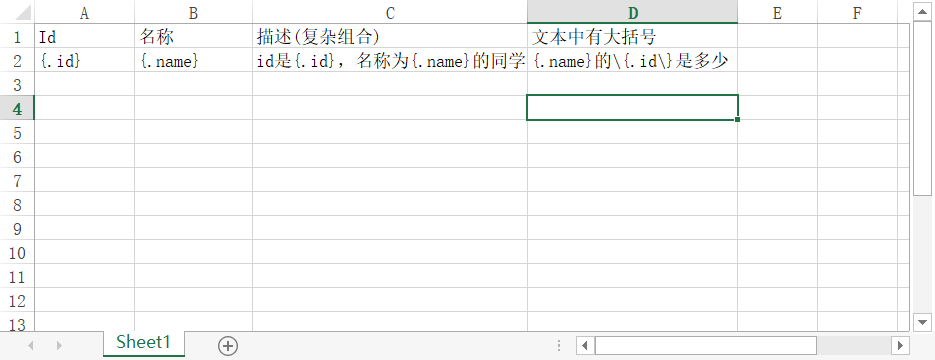
@Test  
public void testFillOneSet(){  
 //准备模板文件  
 String template = "填充一组数据\_模板.xlsx";  
 //准备数据  
 StudentForFill fill = new StudentForFill();  
 fill.setId("001");  
 fill.setName("李四");  
 //创建工作簿对象，withTemplate有更多重载方法  
 ExcelWriterBuilder excelWriterBuilder = EasyExcel.*write*("填充一组数据\_实现.xlsx", StudentForFill.class).withTemplate(template);  
 //创建工作表对象  
 ExcelWriterSheetBuilder sheet = excelWriterBuilder.sheet();  
 //填充数据  
 sheet.doFill(fill);  
}

(4)填充完成的预览图



## 7.填充多组数据

(1)准备模板execl文件，预览图。

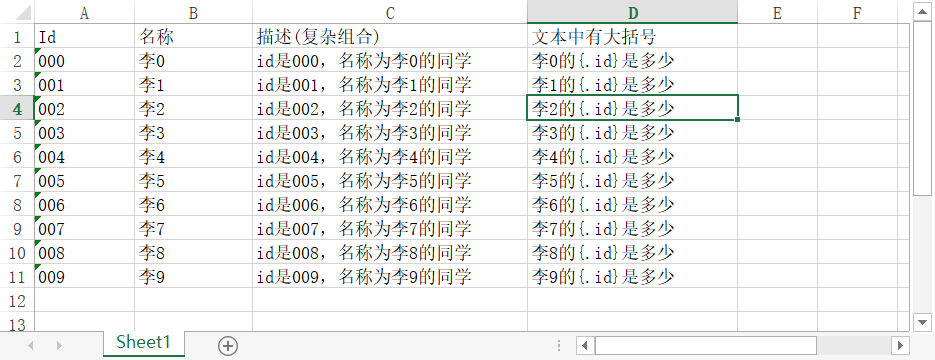


说明：与一组不同的是，在{}中加了.表示为多组数据填充。

(2)填充方法示例

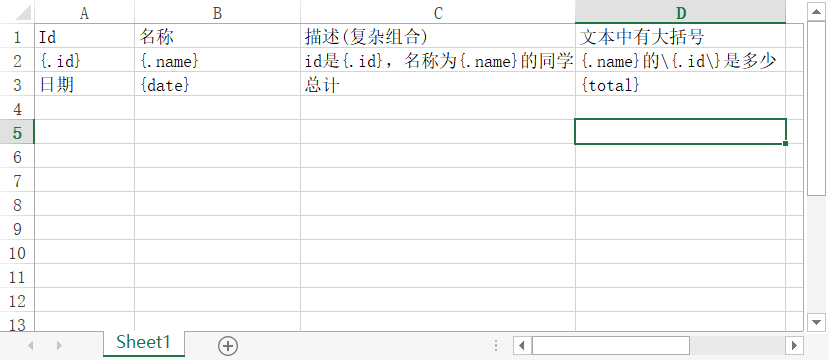
@Test  
public void testFillMultipleSet(){  
 //准备模板文件  
 String template = "填充多组数据\_模板.xlsx";  
 //准备数据  
 List<StudentForFill> list = new ArrayList<>();  
 for(int i = 0;i < 10;i++){  
 StudentForFill fill = new StudentForFill();  
 fill.setId("00"+i);  
 fill.setName("李"+i);  
 list.add(fill);  
 }  
 //创建工作簿对象，withTemplate有更多重载方法  
 ExcelWriterBuilder excelWriterBuilder = EasyExcel.*write*("填充多组数据\_实现.xlsx", StudentForFill.class).withTemplate(template);  
 //创建工作表对象  
 ExcelWriterSheetBuilder sheet = excelWriterBuilder.sheet();  
 //填充数据，doXxx系列方法会在读/写完成后自动关闭流  
 sheet.doFill(list);  
}

(3)填充完成的预览图



## 8.组合填充，既有一组，也有多组

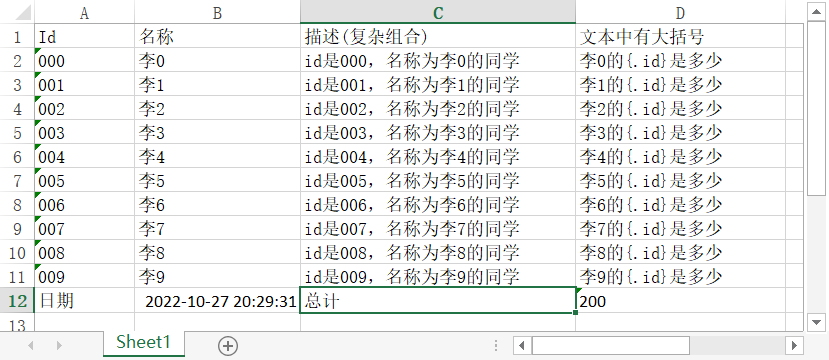
(1)准备模板execl文件，预览图。



(2)填充方法示例

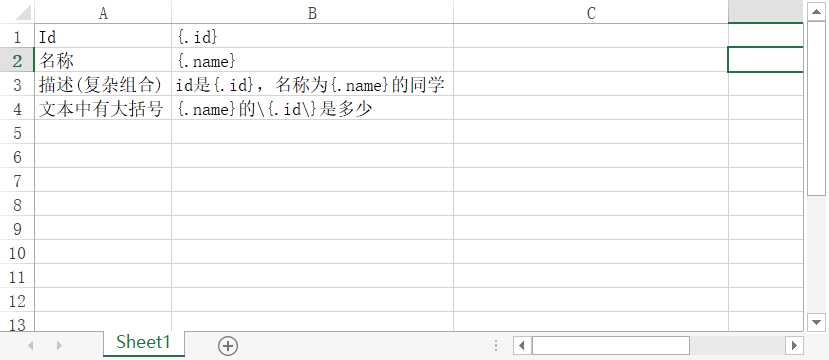
@Test  
public void testFillSet(){  
 //准备模板文件  
 String template = "组合填充数据\_模板.xlsx";  
 //准备数据  
 List<StudentForFill> list = new ArrayList<>();  
 for(int i = 0;i < 10;i++){  
 StudentForFill fill = new StudentForFill();  
 fill.setId("00"+i);  
 fill.setName("李"+i);  
 list.add(fill);  
 }  
 Map<String,Object> data = new HashMap<>();  
 data.put("date",new Date());  
 data.put("total","200");  
 //创建工作簿对象，withTemplate有更多重载方法  
 ExcelWriter workBook = EasyExcel.*write*("组合填充数据\_实现.xlsx", StudentForFill.class).  
 withTemplate(template).build();  
 //创建工作表对象  
 WriteSheet sheet = EasyExcel.*writerSheet*().build();  
 //设置换行，否则在模板文件中，如果多组数据在单组数据的上面，则多组数据会覆盖单组数据  
 FillConfig fillConfig = FillConfig.*builder*().forceNewRow(true).build();  
 //多组填充数据  
 workBook.fill(list,fillConfig,sheet);  
 //单组填充数据  
 workBook.fill(data,sheet);  
 //关闭流  
 workBook.finish();  
}

(3)填充完成的预览图



## 9.多组水平填充

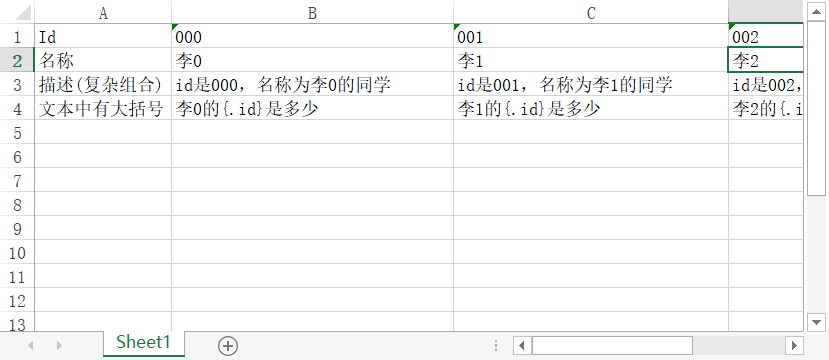
(1)准备模板execl文件，预览图。



(2)填充方法示例

@Test  
public void testFillMultipleSetToHorizontal(){  
 //准备模板文件  
 String template = "多组水平填充\_模板.xlsx";  
 //准备数据  
 List<StudentForFill> list = new ArrayList<>();  
 for(int i = 0;i < 10;i++){  
 StudentForFill fill = new StudentForFill();  
 fill.setId("00"+i);  
 fill.setName("李"+i);  
 list.add(fill);  
 }  
 //创建工作簿对象，withTemplate有更多重载方法  
 ExcelWriter workBook = EasyExcel.*write*("多组水平填充\_实现.xlsx", StudentForFill.class).  
 withTemplate(template).build();  
 //创建工作表对象  
 WriteSheet sheet = EasyExcel.*writerSheet*().build();  
 //设置水平填充  
 FillConfig fillConfig = FillConfig.*builder*().direction(WriteDirectionEnum.*HORIZONTAL*).build();  
 //多组填充数据  
 workBook.fill(list,fillConfig,sheet);  
 //关闭流  
 workBook.finish();  
}

(3)填充完成的预览图



## 10.补充说明

在EasyExcel官网上还有许多关于读/写的其他功能，包括读表头数据、读额外信息(批注、超

链接)、写数据时指定单元格样式、写图片等。详情可以去官网查看，这里不再一一测

试。

# MybatisPlus

## 1.MybatisPlus介绍

MybatisPlus是Mybatis的增强版，使用类似jpa的方式大大简化了单表的增删改查，原因是其内部封装了很多对于单表的增删改查方法。使用时需要引入依赖：

<dependency>  
 <groupId>com.baomidou</groupId>  
 <artifactId>mybatis-plus-boot-starter</artifactId>  
 <version>3.5.1</version>  
</dependency>

当引入mybatisPlus后就不需要再引入mybatis依赖。

## 2.自带的注解

(1)@TableName

作用：修饰实体类的类名，用于指定实体类对应的数据库表名。

例如：

@TableName("t\_user")  
public class User {

补充：t\_user是表名。

(2)@TableId

作用：修饰实体类中，作为主键的字段。

例如：

*/\*\*  
 \** ***@TableId*** *可以指定表的主键所对应的属性。  
 \* 1.其value表示主键字段的名称，用于处理当字段名和属性名不一样时的无法映射的问题。  
 \* 2.其type表示主键值的生成策略，值是一个枚举类型。含义如下：  
 \* IdType.AUTO — 主键自增，系统分配，不需要手动输入。  
 \* 使用这个类型的前提是数据库的主键已经设置为自动递增。  
 \* IdType.NONE — 未设置主键(默认)  
 \* IdType.INPUT — 需要自己输入 主键值。  
 \* IdType.ASSIGN\_ID — 系统分配 ID，用于数值型数据  
 \* 这个是基于雪花算法生成的id，与数据库是否主键自增无关。  
 \* IdType.ASSIGN\_UUID — 系统分配 UUID，用于字符串型数据  
 \*/*@TableId(value = "id",type = IdType.*AUTO*)  
private Long id;

(3)@TableField

作用：修饰实体类中，作为列的字段。

例如：

*/\*\*  
 \** ***@TableField*** *可以指定表的字段对应哪个属性。  
 \* 1.其value表示字段的名称，用于处理当字段名和属性名不一样时的无法映射的问题。  
 \* 2.其exist表示该属性在数据库的表中是否有对应的字段，默认是true(有对应的字段)  
 \* 3.其fill表示自动填充，多用于属性-创建时间、更新时间。值是一个枚举类型，含义如下：  
 \* FieldFill.DEFAULT 不自动填充  
 \* FieldFill.INSERT 插入时自动填充  
 \* FieldFill.UPDATE 更新时自动填充  
 \* FieldFill.INSERT\_UPDATE 插入或更新时自动填充  
 \*  
 \* 如果该属性是userName，而数据库的表名是user\_name，则可以直接映射成功而不需要做其他配置。  
 \* 因为mybatis\_plus有默认的映射规则，即它会自动将数据库中的下划线命名风格转化为实体类中的驼峰命名风格。  
 \*/*@TableField(value = "name",exist = true,fill = FieldFill.*DEFAULT*)  
private String name;

(4)@TableLogic

作用：修饰实体类中，作为逻辑删除的列的字段。

*/\*\*  
 \** ***@TableLogic*** *用来修饰逻辑删除的字段，这样调用mybatisPlus自带的删除方法时，会进行逻辑删除，  
 \* 即改变isDeleted属性的值为已删除，而不是直接物理删除。在调用mybatisPlus自带的查询方法时，  
 \* 也会加上这个逻辑删除的字段，即is\_deleted = '0'。  
 \* 1.value表示未删除的标识，默认是0  
 \* 2.delval表示删除的标识，默认是1  
 \*  
 \*/*@TableLogic(value = "0", delval = "1")  
private String isDeleted;

(5)@Version-乐观锁

*/\*\*  
 \** ***@Version*** *标识version属性是乐观锁需要的版本号字段  
 \* 数据库中必须有对应的字段  
 \*/*@Version  
private Integer version;

(6)

## 3.自带的单表方法

说明：在service和mapper中都有单表方法，为了加以区分，将mapper中的单表方法的前缀和SQL语句的关键字保持一致，比如：mapper中查询的方法以select开头、新增的方法以insert开头、删除的方法以delete开头、更新的方法以update开头。

(1)Service方法

boolean save(T entity)

boolean saveBatch(Collection<T> entityList)

boolean saveBatch(Collection<T> entityList, int batchSize)

boolean saveOrUpdate(T entity)

boolean saveOrUpdate(T entity, Wrapper<T> updateWrapper)

boolean saveOrUpdateBatch(Collection<T> entityList)

boolean saveOrUpdateBatch(Collection<T> entityList, int batchSize)

T getById(Serializable id)

T getOne(Wrapper<T> queryWrapper)

T getOne(Wrapper<T> queryWrapper, boolean throwEx)

Map<String, Object> getMap(Wrapper<T> queryWrapper)

<V> V getObj(Wrapper<T> queryWrapper, Function<? super Object, V> mapper)

boolean remove(Wrapper<T> queryWrapper)

boolean removeById(Serializable id)

boolean removeById(Serializable id, boolean useFill)

boolean removeById(T entity)

boolean removeByIds(Collection<?> list)

boolean removeByIds(Collection<?> list, boolean useFill)

boolean removeBatchByIds(Collection<?> list, int batchSize)

boolean removeBatchByIds(Collection<?> list, int batchSize, boolean useFill)

boolean removeBatchByIds(Collection<?> list)

boolean removeBatchByIds(Collection<?> list, boolean useFill)

boolean removeByMap(Map<String, Object> columnMap)

boolean update(Wrapper<T> updateWrapper)

boolean update(T entity, Wrapper<T> updateWrapper)

boolean updateById(T entity)

boolean updateBatchById(Collection<T> entityList)

boolean updateBatchById(Collection<T> entityList, int batchSize)

List<T> list()

List<T> list(Wrapper<T> queryWrapper)

List<T> listByIds(Collection<? extends Serializable> idList)

List<T> listByMap(Map<String, Object> columnMap)

List<Map<String, Object>> listMaps()

List<Map<String, Object>> listMaps(Wrapper<T> queryWrapper)

List<Object> listObjs()

List<Object> listObjs(Wrapper<T> queryWrapper)

<V> List<V> listObjs(Function<? super Object, V> mapper)

<V> List<V> listObjs(Wrapper<T> queryWrapper, Function<? super Object, V>

mapper)

long count()

long count(Wrapper<T> queryWrapper)

<E extends IPage<T>> E page(E page)

<E extends IPage<T>> E page(E page, Wrapper<T> queryWrapper)

(2)Mapper方法

int insert(T entity)

T selectById(Serializable id)

T selectOne(@Param("ew") Wrapper<T> queryWrapper)

List<T> selectList(@Param("ew") Wrapper<T> queryWrapper)

List<T> selectBatchIds(@Param("coll") Collection<? extends Serializable> idList)

List<T> selectByMap(@Param("cm") Map<String, Object> columnMap)

Long selectCount(@Param("ew") Wrapper<T> queryWrapper)

List<Object> selectObjs(@Param("ew") Wrapper<T> queryWrapper)

List<Map<String, Object>> selectMaps(@Param("ew") Wrapper<T> queryWrapper)

<P extends IPage<T>> P selectPage(P page, @Param("ew") Wrapper<T> queryWrapper)

<P extends IPage<Map<String, Object>>> P selectMapsPage(P page, @Param("ew") Wrapper<T> queryWrapper)

int delete(@Param("ew") Wrapper<T> queryWrapper)

int deleteById(Serializable id)

int deleteById(T entity)

int deleteByMap(@Param("cm") Map<String, Object> columnMap)

int deleteBatchIds(@Param("coll") Collection<?> idList)

int update(@Param("et") T entity, @Param("ew") Wrapper<T> updateWrapper)

int updateById(@Param("et") T entity)

(3)方法举例

@Test  
public void test01() {  
 */\*\*  
 \* 查询用户名包含a，年龄在20到30之间，并且邮箱不为null的用户信息  
 \* SELECT id,name,age,email,create\_time,update\_time,is\_deleted FROM t\_user  
 \* WHERE is\_deleted='0' AND (name LIKE ? AND age BETWEEN ? AND ?  
 \* AND email IS NOT NULL)  
 \*/* QueryWrapper<User> queryWrapper = new QueryWrapper<>();  
 queryWrapper.like("name", "a")  
 //between也可以用ge(大于等于)和le(小于等于)这俩方法替换  
 //gt(大于)和lt(小于)  
 .between("age", 20, 30)  
 .isNotNull("email");  
 List<User> list = userMapper.selectList(queryWrapper);  
 list.forEach(System.*out*::println);  
}  
  
@Test  
public void test02() {  
 */\*\*  
 \* 按年龄降序查询用户，如果年龄相同则按id升序排列  
 \* SELECT id,name,age,email,create\_time,update\_time,is\_deleted FROM t\_user  
 \* WHERE is\_deleted='0' ORDER BY age DESC,id ASC  
 \*/* QueryWrapper<User> queryWrapper = new QueryWrapper<>();  
 queryWrapper.orderByDesc("age").orderByAsc("id");  
 List<User> users = userMapper.selectList(queryWrapper);  
 users.forEach(System.*out*::println);  
}  
  
@Test  
public void test03() {  
 */\*\*  
 \* 删除email为空的用户  
 \* 因为在User类中配置了注解@TableLogic，所以是逻辑删除  
 \* UPDATE t\_user SET is\_deleted='1' WHERE is\_deleted='0' AND (email IS NULL)  
 \*/* QueryWrapper<User> queryWrapper = new QueryWrapper<>();  
 queryWrapper.isNull("email");  
 //条件构造器也可以构建删除语句的条件  
 int result = userMapper.delete(queryWrapper);  
 System.*out*.println("受影响的行数：" + result);  
}  
  
@Test  
public void test04() {  
 QueryWrapper<User> queryWrapper = new QueryWrapper<>();  
 */\*\*  
 \* 将（年龄大于20并且用户名中包含有a）或邮箱为null的用户信息修改  
 \* UPDATE t\_user SET age=?, email=? WHERE is\_deleted='0'  
 \* AND (name LIKE ? AND age > ? OR email IS NULL)  
 \*/* queryWrapper.like("name", "a").gt("age", 20)  
 .or().isNull("email");  
 User user = new User();  
 user.setAge(18);  
 user.setEmail("user@atguigu.com");  
 int result = userMapper.update(user, queryWrapper);  
 System.*out*.println("受影响的行数：" + result);  
}  
  
@Test  
public void test05() {  
 QueryWrapper<User> queryWrapper = new QueryWrapper<>();  
 */\*\*  
 \* 将用户名中包含有a并且（年龄大于20或邮箱为null）的用户信息修改  
 \* UPDATE t\_user SET age=?, email=? WHERE is\_deleted='0'  
 \* AND (name LIKE ? AND (age > ? OR email IS NULL))  
 \*/* //lambda表达式内的逻辑优先运算  
 queryWrapper.like("name", "a")  
 .and(i -> i.gt("age", 20)  
 .or().isNull("email"));  
 User user = new User();  
 user.setAge(18);  
 user.setEmail("user@atguigu.com");  
 int result = userMapper.update(user, queryWrapper);  
 System.*out*.println("受影响的行数：" + result);  
}  
  
@Test  
public void test06() {  
 */\*\*  
 \* 查询用户信息的name和age字段  
 \* SELECT name,age FROM t\_user WHERE is\_deleted='0'  
 \*/* QueryWrapper<User> queryWrapper = new QueryWrapper<>();  
 queryWrapper.select("name", "age");  
 //selectMaps()返回Map集合列表，通常配合select()使用，避免User对象中没有被查询到的列值为null  
 List<Map<String, Object>> maps = userMapper.selectMaps(queryWrapper);  
 maps.forEach(System.*out*::println);  
}  
  
@Test  
public void test07() {  
 */\*\*  
 \* 查询id小于等于3的用户信息  
 \* SELECT id,name,age,email,create\_time,update\_time,is\_deleted  
 \* FROM t\_user WHERE is\_deleted='0' AND  
 \* (id IN (select id from t\_user where id <= 3))  
 \*/* QueryWrapper<User> queryWrapper = new QueryWrapper<>();  
 queryWrapper.inSql("id", "select id from t\_user where id <= 3");  
 List<User> list = userMapper.selectList(queryWrapper);  
 list.forEach(System.*out*::println);  
}  
  
@Test  
public void test08() {  
 */\*\*  
 \* 将（年龄大于20或邮箱为null）并且用户名中包含有a的用户信息修改，  
 \* 组装set子句以及修改条件  
 \*/* UpdateWrapper<User> updateWrapper = new UpdateWrapper<>();  
 //lambda表达式内的逻辑优先运算  
 updateWrapper  
 .set("age", 18)  
 .set("email", "user@atguigu.com")  
 .like("name", "a")  
 .and(i -> i.gt("age", 20).or().isNull("email"));  
 */\*\*  
 \* 这里必须要创建User对象，否则无法应用自动填充。如果没有自动填充，可以设置为null  
 \* UPDATE t\_user SET name=?, age=?,email=? WHERE is\_deleted='0' AND (name LIKE ?  
 \* AND (age > ? OR email IS NULL))  
 \*/* User user = new User();  
 user.setName("张三");  
 //第一种修改方式，需要传一个User对象  
 int result1 = userMapper.update(user, updateWrapper);  
 */\*\*  
 \* UPDATE t\_user SET age=?,email=? WHERE is\_deleted='0' AND (name LIKE ?  
 \* AND (age > ? OR email IS NULL))  
 \*/* //第二种修改方式，不需要传User对象  
 int result2 = userMapper.update(null, updateWrapper);  
 System.*out*.println(result2);  
}  
  
@Test  
public void test09() {  
 //定义查询条件，有可能为null（用户未输入或未选择）  
 String name = null;  
 Integer ageBegin = 10;  
 Integer ageEnd = 24;  
 QueryWrapper<User> queryWrapper = new QueryWrapper<>();  
 //isNotBlank()判断某字符串是否不为空且长度不为0且不由空白符(whitespace)构成  
 if(StringUtils.*isNotBlank*(name)){  
 queryWrapper.like("name","a");  
 }  
 if(ageBegin != null){  
 queryWrapper.ge("age", ageBegin);  
 }  
 if(ageEnd != null){  
 queryWrapper.le("age", ageEnd);  
 }  
 */\*\*  
 \* SELECT id,name,age,email,create\_time,update\_time,is\_deleted FROM  
 \* t\_user WHERE is\_deleted='0' AND (age >= ? AND age <= ?)  
 \*/* List<User> users = userMapper.selectList(queryWrapper);  
 users.forEach(System.*out*::println);  
}  
  
@Test  
public void test10() {  
 //定义查询条件，有可能为null（用户未输入或未选择）  
 String name = null;  
 Integer ageBegin = 10;  
 Integer ageEnd = 24;  
 QueryWrapper<User> queryWrapper = new QueryWrapper<>();  
 //isNotBlank()判断某字符串是否不为空且长度不为0且不由空白符(whitespace)构成  
 queryWrapper.like(StringUtils.*isNotBlank*(name), "name", "a")  
 .ge(ageBegin != null, "age", ageBegin)  
 .le(ageEnd != null, "age", ageEnd);  
 */\*\*  
 \* SELECT id,name,age,email,create\_time,update\_time,is\_deleted  
 \* FROM t\_user WHERE is\_deleted='0' AND (age >= ? AND age <= ?)  
 \*/* List<User> users = userMapper.selectList(queryWrapper);  
 users.forEach(System.*out*::println);  
}  
  
@Test  
public void test11() {  
 //定义查询条件，有可能为null（用户未输入）  
 String name = "a";  
 Integer ageBegin = 10;  
 Integer ageEnd = 24;  
 LambdaQueryWrapper<User> queryWrapper = new LambdaQueryWrapper<>();  
 //避免使用字符串表示字段，防止运行时错误  
 queryWrapper  
 .like(StringUtils.*isNotBlank*(name), User::getName, name)  
 .ge(ageBegin != null, User::getAge, ageBegin)  
 .le(ageEnd != null, User::getAge, ageEnd);  
 */\*\*  
 \* SELECT id,name,age,email,create\_time,update\_time,is\_deleted  
 \* FROM t\_user WHERE is\_deleted='0' AND (name LIKE ?  
 \* AND age >= ? AND age <= ?)  
 \*/* List<User> users = userMapper.selectList(queryWrapper);  
 users.forEach(System.*out*::println);  
}  
  
@Test  
public void test12() {  
 //组装set子句  
 LambdaUpdateWrapper<User> updateWrapper = new LambdaUpdateWrapper<>();  
 updateWrapper  
 .set(User::getAge, 18)  
 .set(User::getEmail, "user@atguigu.com")  
 .like(User::getName, "a")  
 .and(i -> i.lt(User::getAge, 24).or().isNull(User::getEmail)); //lambda  
 //表达式内的逻辑优先运算  
 User user = new User();  
 */\*\*  
 \* UPDATE t\_user SET age=?,email=? WHERE is\_deleted='0' AND  
 \* (name LIKE ? AND (age < ? OR email IS NULL))  
 \*/* int result = userMapper.update(user, updateWrapper);  
 System.*out*.println("受影响的行数：" + result);  
}

## 4.wrapper条件构造器

说明：条件构造器用于复杂的单表的增删改查的处理。

(1) EntityWrapper

EntityWrapper是Mybatis-Plus2.x及以下版本使用的查询条件构造器。

(2) QueryWrapper

QueryWrapper是Mybatis-Plus3.x及以上版本使用的查询条件构造器。

(3) LambdaQueryWrapper

LambdaQueryWrapper和QueryWrapper类似，不同的是它的对象在使用方法时，方法的参数可以引用属性而不用写明确的字符串。例如：

LambdaQueryWrapper< UserEntity> wrapper = new LambdaQueryWrapper();

wrapper.eq(UserEntity::getUsername,”名称”);

## 5.条件构造器方法

(1)allEq-全部条件都相等

public List<UserEntity> getListByAllEq() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 Map<String, Object> paramsMap = new HashMap<>();  
 paramsMap.put("username", "1");  
 paramsMap.put("pickname", "张三");  
 queryWrapper.allEq(paramsMap);  
 return userService.list(queryWrapper);  
}

(2)eq-指定条件相等

public List<UserEntity> getListByEq() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().eq(UserEntity::getUsername, "123");  
 return userService.list(queryWrapper);  
}

(3)ne-指定条件不相等

public List<UserEntity> getListByNe() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().ne(UserEntity::getUsername, "123");  
 return userService.list(queryWrapper);  
}

(4)gt-大于指定条件

public List<UserEntity> getListByGt() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().gt(UserEntity::getAge, 12);  
 return userService.list(queryWrapper);  
}

(5)ge-大于等于指定条件

public List<UserEntity> getListByGt() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().ge(UserEntity::getAge, 12);  
 return userService.list(queryWrapper);  
}

(6)lt-小于指定条件

public List<UserEntity> getListByGt() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().lt(UserEntity::getAge, 12);  
 return userService.list(queryWrapper);  
}

(7)le-小于等于指定条件

public List<UserEntity> getListByGt() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().le(UserEntity::getAge, 12);  
 return userService.list(queryWrapper);  
}

(8)between-介于指定范围之间

public List<UserEntity> getListByBetween() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().between(UserEntity::getAge, 11, 12);  
 return userService.list(queryWrapper);  
}

(9)notBetween-不介于指定范围之间

public List<UserEntity> getListByBetween() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().notBetween(UserEntity::getAge, 11, 12);  
 return userService.list(queryWrapper);  
}

(10)like-模糊匹配

public List<UserEntity> getListByLike() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().like(UserEntity::getUsername, "11");  
 return userService.list(queryWrapper);  
}

(11)notLike-非模糊匹配

public List<UserEntity> getListByLike() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().notLike(UserEntity::getUsername, "11");  
 return userService.list(queryWrapper);  
}

(12)likeLeft-左模糊匹配

public List<UserEntity> getListByLike() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().likeLeft(UserEntity::getUsername, "11");  
 return userService.list(queryWrapper);  
}

(13)likeRight-右模糊匹配

public List<UserEntity> getListByLike() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().likeRight(UserEntity::getUsername, "11");  
 return userService.list(queryWrapper);  
}

(14)isNull-指定字段为null

public List<UserEntity> getListByIsNull() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().isNull(UserEntity::getSex);  
 return userService.list(queryWrapper);  
}

(15) isNotNull-指定字段不为null

public List<UserEntity> getListByIsNull() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().isNotNull(UserEntity::getSex);  
 return userService.list(queryWrapper);  
}

(16)in-满足指定条件之一

public List<UserEntity> getListByIn() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().in(UserEntity::getUsername, "11", "123");  
 return userService.list(queryWrapper);  
}

(17)notIn-不满足任一指定条件

public List<UserEntity> getListByIn() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().notIn(UserEntity::getUsername, "11", "123");  
 return userService.list(queryWrapper);  
}

(18)inSql-满足指定条件之一

public List<UserEntity> getListByInSql() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().inSql(UserEntity::getUsername, "11,123");  
 return userService.list(queryWrapper);  
}

补充：其与in方法的区别在于：in同等于#{}，不存在SQL注入隐患，但是值必须是已经指定的；而inSql同等于${}，存在SQL注入隐患，但是其更加灵活，值可以是动态的，比如：

queryWrapper.lambda().inSql(UserEntity::getUsername, "select user\_name from user");

(19)notInSql-不满足任一指定条件

public List<UserEntity> getListByInSql() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().notInSql(UserEntity::getUsername, "11,123");  
 return userService.list(queryWrapper);  
}

补充：其与notIn方法的区别跟in和inSql之间类似。

(20)groupBy-分组

public List<UserEntity> getListByGroupBy() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().groupBy(UserEntity::getSex);  
 return userService.list(queryWrapper);  
}

(21)orderByAsc-升序排列

public List<UserEntity> getListByOrderByAsc() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().orderByAsc(UserEntity::getUsername);  
 return userService.list(queryWrapper);  
}

(22)orderByDesc-降序排序

public List<UserEntity> getListByOrderByAsc() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().orderByDesc(UserEntity::getUsername);  
 return userService.list(queryWrapper);  
}

(23)orderBy-排序

public List<UserEntity> getListByOrderBy() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().orderBy(true, false, UserEntity::getUsername);  
 return userService.list(queryWrapper);  
}

补充：第一个参数必须为true；第二个参数为true则升序排序，为false则降序排序。

(24)having-跟sql中的having类似

public List<Map<String, Object>> getListByHaving() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.select("password,count(\*) as sexCount")  
 .groupBy("password")  
 .having("count(\*)>1");  
 return userService.listMaps(queryWrapper);  
}

补充：having()需要配合select()、groupBy()一起配合使用；having里面只能使用聚合函数。

(25)func-主要方便在出现if...else下调用不同方法

public List<UserEntity> getListByFunc() {  
 LambdaQueryWrapper<UserEntity> queryWrapper = Wrappers.<UserEntity>lambdaQuery();  
 queryWrapper.func(x -> {  
 if (true) {  
 x.eq(UserEntity::getUsername, 1);  
 } else {  
 x.eq(UserEntity::getUsername, 1);  
 }  
 });  
 return userService.list(queryWrapper);  
}

(26)and-和sql中的and类似

public List<UserEntity> getListByAnd() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().eq(UserEntity::getPassword, "123456")  
 .and(t -> t.eq(UserEntity::getSex, "02"));  
 return userService.list(queryWrapper);  
}

(27)or-和sql中的or类似

public List<UserEntity> getListByOr() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.lambda().eq(UserEntity::getPassword, "123")  
 .or(t -> t.eq(UserEntity::getSex, "02"));  
 return userService.list(queryWrapper);  
}

(28) exists-和sql中的exists类似

public List<UserEntity> getListByExists() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.exists("select \* from users where password='123'");  
 return userService.list(queryWrapper);  
}

(29)notExists-和sql中的notExists类似

public List<UserEntity> getListByExists() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.notExists("select \* from users where password='123'");  
 return userService.list(queryWrapper);  
}

(30)select-查询字段

public List<UserEntity> getListBySelect() {  
 QueryWrapper<UserEntity> queryWrapper = new QueryWrapper();  
 queryWrapper.select("username,password");  
 return userService.list(queryWrapper);  
}

## 6.设置分页与乐观锁

(1)添加配置类

@Configuration  
//扫描mapper接口所在的包，该注解也可以放到springBoot启动类上  
@MapperScan("com.itcast.mapper")  
public class MybatisPlusConfig {  
 @Bean  
 public MybatisPlusInterceptor mybatisPlusInterceptor() {  
 MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();  
 //配置MybatisPlus的分页功能  
 interceptor.addInnerInterceptor(new  
 PaginationInnerInterceptor(DbType.*MYSQL*));  
 //配置MybatisPlus的乐观锁  
 interceptor.addInnerInterceptor(new  
 OptimisticLockerInnerInterceptor());  
 return interceptor;  
 }  
}

(2)测试分页

*/\*\*  
 \* 测试MybatisPlusConfig类中配置的分页功能  
 \*/*@Test  
public void testPage(){  
 //设置分页参数  
 Page<User> page = new Page<>(1, 5);  
 */\*\*  
 \* 可以看到执行了两条SQL：  
 \* SELECT COUNT(\*) AS total FROM t\_user WHERE is\_deleted = '0'  
 \*  
 \* SELECT id,name,age,email,create\_time,update\_time,is\_deleted  
 \* FROM t\_user WHERE is\_deleted='0' LIMIT ?  
 \*/* userMapper.selectPage(page, null);  
 //获取分页数据  
 List<User> list = page.getRecords();  
 list.forEach(System.*out*::println);  
 System.*out*.println("当前页："+page.getCurrent());// 1  
 System.*out*.println("每页显示的条数："+page.getSize());// 5  
 System.*out*.println("总记录数："+page.getTotal());// 2  
 System.*out*.println("总页数："+page.getPages());// 1  
 System.*out*.println("是否有上一页："+page.hasPrevious());// false  
 System.*out*.println("是否有下一页："+page.hasNext());// false  
}  
  
*/\*\*  
 \* 测试mapper中自定义的分页方法  
 \*/*@Test  
public void testSelectPageVo(){  
 //设置分页参数  
 Page<User> page = new Page<>(1, 5);  
 */\*\*  
 \* SELECT COUNT(\*) AS total FROM t\_user WHERE age > ?  
 \* SELECT id,name,age,email FROM t\_user WHERE age > ? LIMIT ?  
 \*/* userMapper.selectPageVo(page, 20);  
 //获取分页数据  
 List<User> list = page.getRecords();  
 list.forEach(System.*out*::println);  
 System.*out*.println("当前页："+page.getCurrent());// 1  
 System.*out*.println("每页显示的条数："+page.getSize());// 5  
 System.*out*.println("总记录数："+page.getTotal());// 3  
 System.*out*.println("总页数："+page.getPages());// 1  
 System.*out*.println("是否有上一页："+page.hasPrevious());// false  
 System.*out*.println("是否有下一页："+page.hasNext());//false  
}

(3)测试乐观锁

需要Product对象中有乐观锁字段，比如：

*/\*\*  
 \** ***@Version*** *标识version属性是乐观锁需要的版本号字段  
 \* 数据库中必须有对应的字段  
 \*/*@Version  
private Integer version;

@Test  
public void testConcurrentVersionUpdate() {  
 */\*\*  
 \* 小李取数据  
 \* SELECT id,name,price,version FROM t\_product WHERE id=?  
 \*/* Product p1 = productMapper.selectById(1L);  
 */\*\*  
 \* SELECT id,name,price,version FROM t\_product WHERE id=?  
 \* 小王取数据  
 \*/* Product p2 = productMapper.selectById(1L);  
 //小李修改 + 50  
 p1.setPrice(p1.getPrice() + 50);  
 */\*\*  
 \* UPDATE t\_product SET name=?, price=?, version=? WHERE id=? AND version=?  
 \* 这句SQL，WHERE条件中version=0，可以修改成功。  
 \*/* int result1 = productMapper.updateById(p1);  
 System.*out*.println("小李修改的结果：" + result1);  
 //小王修改 - 30  
 p2.setPrice(p2.getPrice() - 30);  
 */\*\*  
 \* UPDATE t\_product SET name=?, price=?, version=? WHERE id=? AND version=?  
 \* 这句SQL，WHERE条件中version=0，但是上面小李的update语句将version更新成了1，所以这条  
 \* 更新语句没有效果，所以result2的值为0，即受影响的记录行数为0。  
 \*/* int result2 = productMapper.updateById(p2);  
 System.*out*.println("小王修改的结果：" + result2);  
 if(result2 == 0){  
 */\*\*  
 \* 失败重试，重新获取version并更新  
 \* SELECT id,name,price,version FROM t\_product WHERE id=?  
 \*/* p2 = productMapper.selectById(1L);  
 p2.setPrice(p2.getPrice() - 30);  
 */\*\*  
 \* UPDATE t\_product SET name=?, price=?, version=? WHERE id=? AND version=?  
 \* 这句SQL，因为上面是重新查询的数据库，所以WHERE条件中version=1，可以修改成功。  
 \*/* result2 = productMapper.updateById(p2);  
 }  
 System.*out*.println("小王修改重试的结果：" + result2);  
 */\*\*  
 \* 老板看价格  
 \* SELECT id,name,price,version FROM t\_product WHERE id=?  
 \*/* Product p3 = productMapper.selectById(1L);  
 System.*out*.println("老板看价格：" + p3.getPrice());// 120  
}

## 7.设置通用枚举

举例：

(1)配置扫描枚举类

3.5.2版本开始无需配置。

mybatis-plus:

*# 配置扫描通用枚举* type-enums-package: com.itcast.enums

(2)在User类中设置一个枚举字段

private SexEnum sex;

(3)创建一个枚举并使用注解

@Getter  
public enum SexEnum {  
 *MALE*(1, "男"),  
 *FEMALE*(2, "女");  
 @EnumValue  
 private Integer sex;  
 private String sexName;  
 SexEnum(Integer sex, String sexName) {  
 this.sex = sex;  
 this.sexName = sexName;  
 }  
}

这样，在使用mybatisPlus自带的方法时，就可以自动识别并给列赋值。

## 8.代码生成器

说明：通过创建的表逆向生成一套javaBean、Controller、Service、ServiceImpl、Mapper、xml

文件。

(1)导入依赖

<!-- mybatisPlus逆向的代码生成器，示例中需要freemarker引擎模板 -->  
<dependency>  
 <groupId>com.baomidou</groupId>  
 <artifactId>mybatis-plus-generator</artifactId>  
 <version>3.5.1</version>  
</dependency>  
<!-- freemarker引擎模板 -->  
<dependency>  
 <groupId>org.freemarker</groupId>  
 <artifactId>freemarker</artifactId>  
 <version>2.3.31</version>  
</dependency>

(2)设置代码

public class FastAutoGeneratorTest {  
 public static void main(String[] args) {  
 FastAutoGenerator.*create*("jdbc:mysql://127.0.0.1:3306/mybatis\_plus?characterEncoding=utf-8&userSSL=false",  
 "root", "123456")  
 .globalConfig(builder -> {  
 builder.author("lijun") // 设置作者  
 //.enableSwagger() // 开启 swagger 模式  
 .fileOverride() // 覆盖已生成文件  
 .outputDir("F:\\workspace\\ideaSpace\\workspaceLearn\\mybatis\_plus\\generator"); // 指定输出目录  
 }).packageConfig(builder -> {  
 builder.parent("cn.itcast") // 设置父包名  
 .moduleName("mybatisplus") // 设置父包模块名  
 // 设置mapperXml生成路径  
 .pathInfo(Collections.*singletonMap*(OutputFile.*mapperXml*, "F:\\workspace\\ideaSpace\\workspaceLearn\\mybatis\_plus\\generator"));  
 }).strategyConfig(builder -> {  
 builder.addInclude("t\_user") // 设置需要生成的表名  
 .addTablePrefix("t\_", "c\_"); // 设置过滤表前缀  
 }).templateEngine(new FreemarkerTemplateEngine()) // 使用Freemarker引擎模板，默认的是Velocity引擎模板  
 .execute();  
 }  
}

## 9.多数据源

(1)引入依赖

<!-- 多数据源 -->  
<dependency>  
 <groupId>com.baomidou</groupId>  
 <artifactId>dynamic-datasource-spring-boot-starter</artifactId>  
 <version>3.5.0</version>  
</dependency>

(2)yaml配置文件示例

spring:  
 datasource:  
 dynamic:  
 *# 是否严格匹配数据源，默认false。false表示未匹配到指定数据源时使用默认数据源， true表示未匹配到指定数据源时抛异常* strict: false  
 *# 设置默认的数据源或者数据源组，默认值即为master* primary: master  
 datasource:  
 *# 主节点* master:  
 url: jdbc:mysql://localhost:3306/mybatis\_plus?characterEncoding=utf8&useSSL=false  
 driver-class-name: com.mysql.cj.jdbc.Driver  
 username: root  
 password: 123456  
 *# 从节点* slave\_1:  
 url: jdbc:mysql://localhost:3306/mybatis\_plus\_1?characterEncoding=utf8&useSSL=false  
 driver-class-name: com.mysql.cj.jdbc.Driver  
 username: root  
 password: 123456

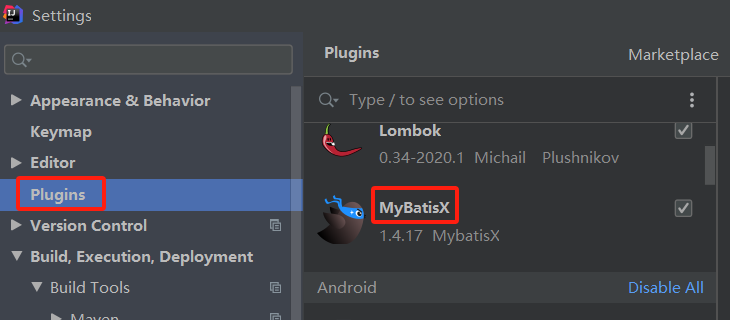
(3)使用注解指定使用哪个数据源

//指定所操作的数据源，也可以修饰方法。若方法和类上都有此注解，则以方法上的为准  
@DS("master")  
@Service  
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements UserService {  
  
}

//指定所操作的数据源，也可以修饰方法。若方法和类上都有此注解，则以方法上的为准  
@DS("slave\_1")  
@Service  
public class ProductServiceImpl extends ServiceImpl<ProductMapper, Product> implements ProductService {  
  
}

## 10.MybatisX插件

(1)在idea的插件商店安装此插件



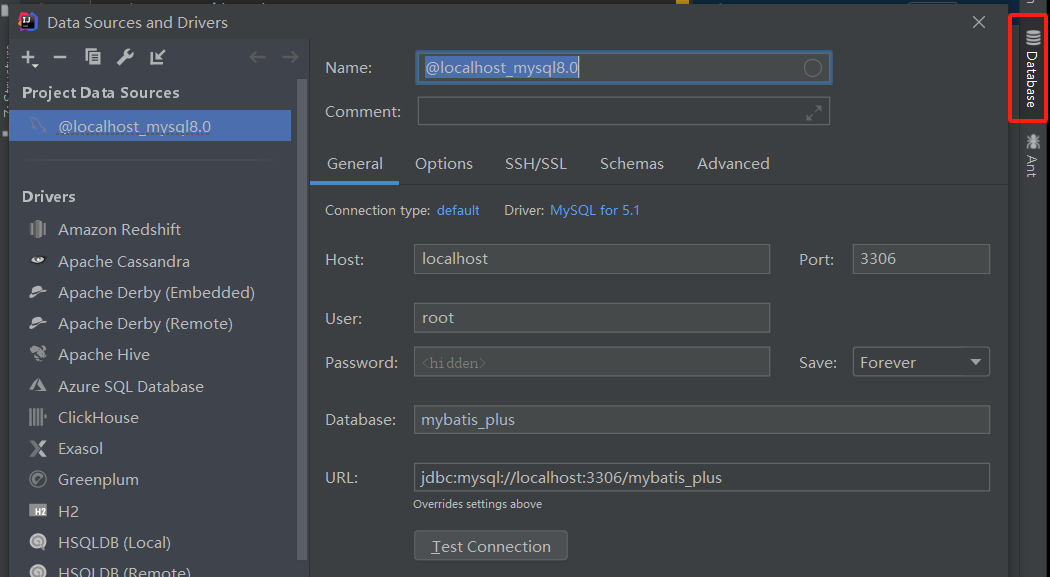
(2)使用此插件的代码生成功能

作用：根据数据库的表反向生成一套javaBean、Controller、Service、ServiceImpl、Mapper、

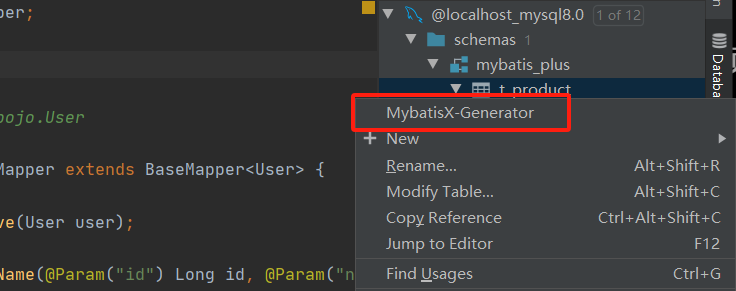
xml文件。

步骤：

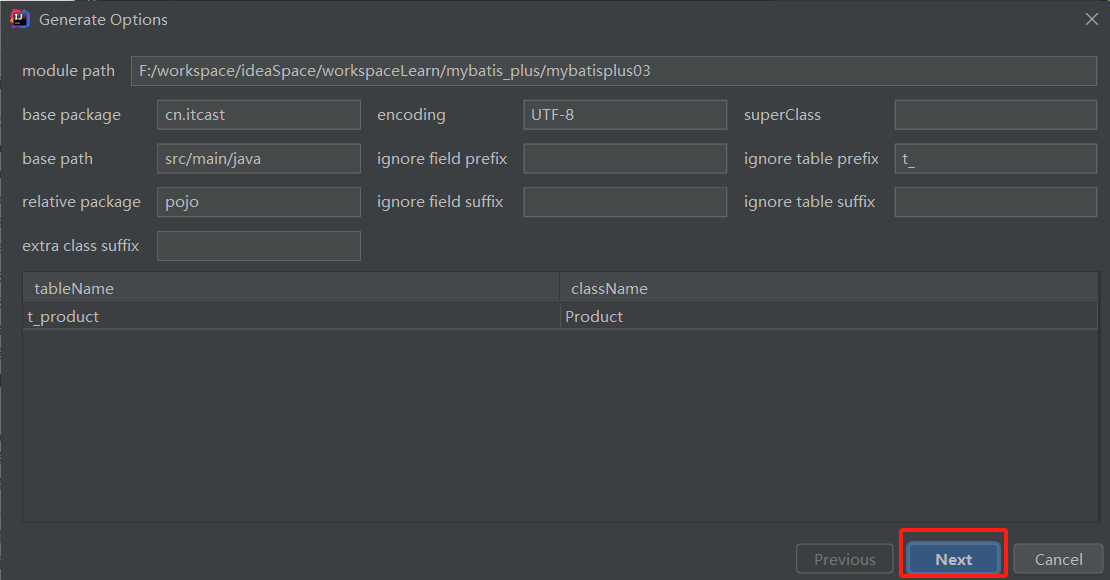
<1>使用右边的Database连接数据库

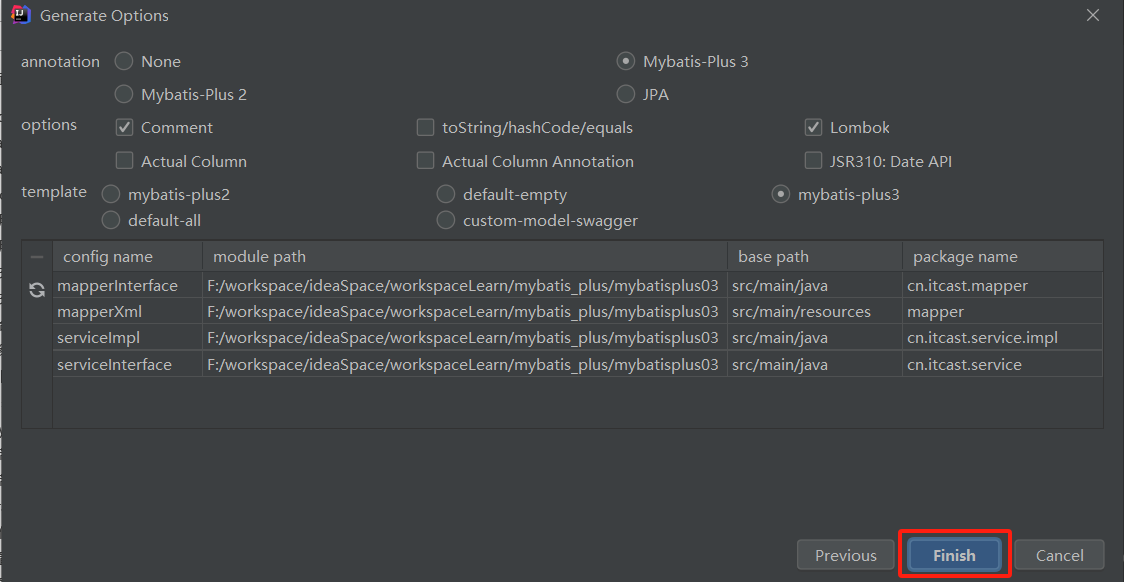


<2>在表名处右键并单机MybatisX-Generator



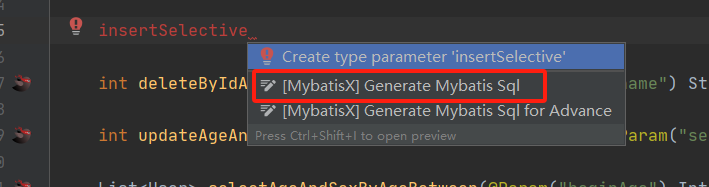
<3>设置参数并生成



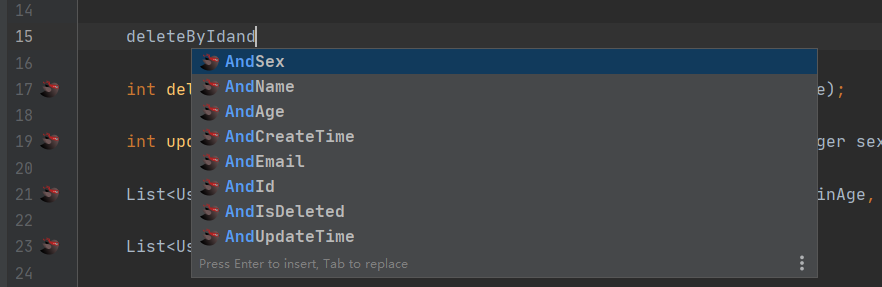


(3)使用此插件快速生成单表方法功能

<1>在mapper类中，自带模板方法，比如insertSelective，此时ALT+ENTER，并选择第二个选项，则会自动生成mapper方法以及xml方法。



<2>在mapper类中，也有智能提示方法。比如想根据某个字段删除记录，只需要写deleteBy，就会提示出所有的字段，如果是根据多个字段删除，只需要再加上and即可再次提示。



此时依然是ALT+ENTER，并选择第二个选项，则会自动生成mapper方法以及xml方法。

<3>在mapper类中，智能提示方法比如update、select方法都和delete类似，也可以使用order by这样的提示。

# Activiti5

## 1. Activiti5介绍

Activiti5是一个工作流框架，实现诸如请假等需要一层一层审核的场景，底层使用Mybatis实现增删改查。其最后一个版本为5.23.0，最后的git更新日期为2019年8月。

## 2.自带的数据表

Activiti5自带25张表，表名和含义如下：

日志表：

act\_evt\_log：用于记录事件日志信息。

通用数据表：

act\_ge\_bytearray：用于存储字节数据。

act\_ge\_property：存储全局属性。

历史数据库表：

act\_hi\_actinst：存储历史活动实例信息。

act\_hi\_attachment：存储历史附件信息。

act\_hi\_comment：存储历史评论信息。

act\_hi\_detail：存储历史详细信息。

act\_hi\_identitylink：存储历史身份关联信息。

act\_hi\_procinst：存储历史流程实例信息。

act\_hi\_taskinst：存储历史任务实例信息。

act\_hi\_varinst：存储历史变量信息。

组织机构表(这四张表太简单，一般都是自己再写一套)：

act\_id\_group：存储用户组信息。

act\_id\_info：存储用户信息。

act\_id\_membership：存储用户和用户组的关系。

act\_id\_user：存储用户信息。

流程定义信息表(5.20版本前没有这张表)：

act\_procdef\_info：存储流程定义信息。

资源库流程规则表：

act\_re\_deployment：存储部署信息。

act\_re\_model：存储模型信息。

act\_re\_procdef：存储流程定义信息。

运行时数据库表：

act\_ru\_event\_subscr：存储运行时事件订阅信息。

act\_ru\_execution：存储运行时流程执行信息。

act\_ru\_identitylink：存储运行时身份关联信息。

act\_ru\_job：存储运行时作业信息。

act\_ru\_task：存储运行时任务信息。

act\_ru\_variable：存储运行时变量信息。

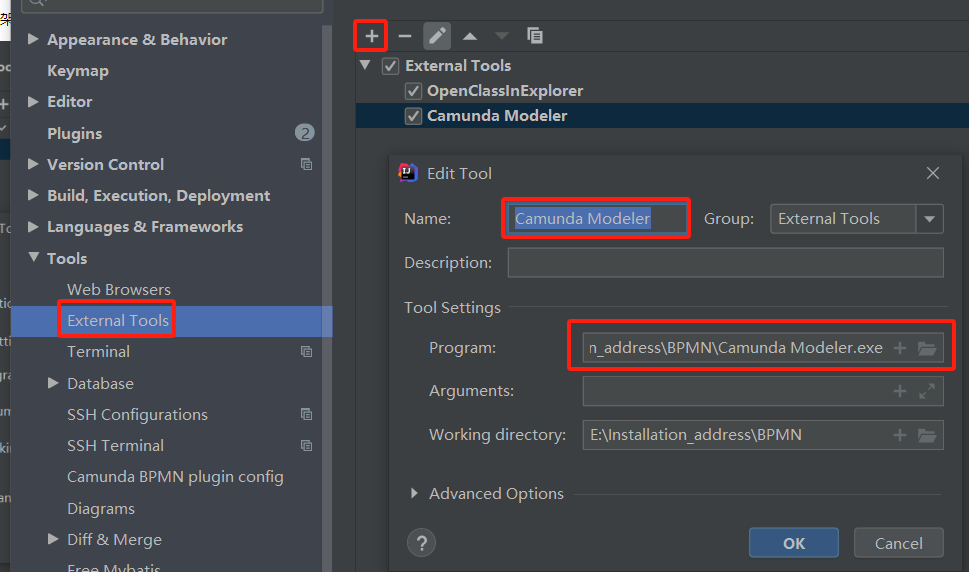
## 3.项目示例

### (1)准备软件环境

<1>下载camunda-modeler-4.11.1-win-x64.zip，其是一个专门绘制BPMN流程图的软件。

<2>解压缩camunda-modeler-4.11.1-win-x64.zip，并在idea的setting中外部工具配置上此软

件的exe文件路径。



这样就可以在项目名右键找到External Tools -> camunda-modeler打开该软件。

注意：该软件是专门为camunda流程引擎设计的，因此将流程图画好后，还需要将文件修

改为activiti专用的。步骤：

1.在文件中加入xmlns:activiti="http://activiti.org/bpmn"

2.将文件中的camunda改为activiti

3.删除xmlns:activiti="http://activiti.org/schema/1.0/bpmn"

### (2)创建Maven项目并引入依赖。项目使用的是SSM框架。

<properties>  
 <activiti.version>5.22.0</activiti.version>  
 <spring.version>5.0.2.RELEASE</spring.version>  
 <mybatis.version>3.5.7</mybatis.version>  
 </properties>  
 <dependencies>  
 <!-- activiti 引擎 -->  
 <dependency>  
 <groupId>org.activiti</groupId>  
 <artifactId>activiti-engine</artifactId>  
 <version>${activiti.version}</version>  
 </dependency>  
 <!-- activiti与spring集成 -->  
 <dependency>  
 <groupId>org.activiti</groupId>  
 <artifactId>activiti-spring</artifactId>  
 <version>${activiti.version}</version>  
 </dependency>  
 <!-- bpmn 模型处理 -->  
 <dependency>  
 <groupId>org.activiti</groupId>  
 <artifactId>activiti-bpmn-model</artifactId>  
 <version>${activiti.version}</version>  
 </dependency>  
 <!-- bpmn 转换 -->  
 <dependency>  
 <groupId>org.activiti</groupId>  
 <artifactId>activiti-bpmn-converter</artifactId>  
 <version>${activiti.version}</version>  
 </dependency>  
 <!-- bpmn json数据转换 -->  
 <dependency>  
 <groupId>org.activiti</groupId>  
 <artifactId>activiti-json-converter</artifactId>  
 <version>${activiti.version}</version>  
 </dependency>  
 <!-- bpmn 布局 -->  
 <dependency>  
 <groupId>org.activiti</groupId>  
 <artifactId>activiti-bpmn-layout</artifactId>  
 <version>${activiti.version}</version>  
 </dependency>  
 <!-- mysql驱动 -->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>8.0.12</version>  
 </dependency>  
 <!-- mybatis -->  
 <dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis</artifactId>  
 <version>${mybatis.version}</version>  
 </dependency>  
 <!-- 连接池 -->  
 <dependency>  
 <groupId>commons-dbcp</groupId>  
 <artifactId>commons-dbcp</artifactId>  
 <version>1.4</version>  
 </dependency>  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.12</version>  
 </dependency>  
 <dependency>  
 <groupId>log4j</groupId>  
 <artifactId>log4j</artifactId>  
 <version>1.2.17</version>  
 </dependency>  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-api</artifactId>  
 <version>1.7.25</version>  
 </dependency>  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-log4j12</artifactId>  
 <version>1.7.25</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-aspects</artifactId>  
 <version>${spring.version}</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-aop</artifactId>  
 <version>${spring.version}</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-beans</artifactId>  
 <version>${spring.version}</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>${spring.version}</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-core</artifactId>  
 <version>${spring.version}</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-expression</artifactId>  
 <version>${spring.version}</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-jdbc</artifactId>  
 <version>${spring.version}</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-tx</artifactId>  
 <version>${spring.version}</version>  
 </dependency>  
 </dependencies>

### (3)使用建好的数据库并创建类初始化25张表

public static ProcessEngine initActTable(){  
 //创建数据源  
 DriverManagerDataSource dataSource = new DriverManagerDataSource();  
 dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");  
 dataSource.setUrl("jdbc:mysql://localhost:3306/activiti?useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai&useSSL=false");  
 dataSource.setUsername("root");  
 dataSource.setPassword("123456");  
  
 // 创建流程引擎的配置  
 ProcessEngineConfiguration configuration = ProcessEngineConfiguration  
 .*createStandaloneProcessEngineConfiguration*();  
// configuration.setJdbcDriver("com.mysql.cj.jdbc.Driver");  
// configuration.setJdbcUrl("jdbc:mysql://localhost:3306/activiti?useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai&useSSL=false");  
// configuration.setJdbcUsername("root");  
// configuration.setJdbcPassword("123456");  
 configuration.setDataSource(dataSource);  
  
 /\*  
 配置表的初始化方式  
 DB\_SCHEMA\_UPDATE\_FALSE：不会创建表  
 DB\_SCHEMA\_UPDATE\_CREATE\_DROP：创建表，使用完之后删除(需要手动关闭引擎才能删除)  
 DB\_SCHEMA\_UPDATE\_TRUE：如果数据库中没有表，就创建  
 "drop-create"：启动时删除原来的表并创建新表  
 \*/  
 configuration.setDatabaseSchemaUpdate(ProcessEngineConfiguration.*DB\_SCHEMA\_UPDATE\_TRUE*);  
 //设置流程引擎  
 ProcessEngine engine = configuration.buildProcessEngine();  
 System.*out*.println(engine);  
 return engine;  
 }

### (4)使用初始化表方法的返回值获取核心API

核心API即是操作自带的25张表的Service。

public static void main(String[] args) {  
 ProcessEngine engine = *initActTable*();  
 /\*  
 流程图的 部署、修改、删除  
 相关表：act\_ge\_bytearray、  
 act\_re\_deployment、  
 act\_re\_model、  
 act\_re\_procdef(流程定义)-ProcessDefinition  
 \*/  
 RepositoryService repositoryService = engine.getRepositoryService();  
 /\*  
 流程的运行  
 相关表：act\_ru\_event\_subscr、  
 act\_ru\_execution(执行实例)-ProcessInstance、Execution  
 没有分支时，ProcessInstance和Execution含义相同；  
 有分支时，Execution表示所有的分支，ProcessInstance表示完整的流程  
 act\_ru\_identitylink、  
 act\_ru\_job、  
 act\_ru\_task(任务实例)-TaskInstance、  
 act\_ru\_variable  
 \*/  
 RuntimeService runtimeService = engine.getRuntimeService();  
 TaskService taskService = engine.getTaskService();  
 //获取act\_ru\_task表中的所有数据  
 List<Task> list = taskService.createTaskQuery().list();  
 /\*  
 查询历史记录  
 相关表：act\_hi\_actinst、  
 act\_hi\_attachment、  
 act\_hi\_comment、  
 act\_hi\_detail、  
 act\_hi\_identitylink、  
 act\_hi\_procinst、  
 act\_hi\_taskinst、  
 act\_hi\_varinst  
 \*/  
 HistoryService historyService = engine.getHistoryService();  
 /\*  
 页面表单(了解)  
 \*/  
 FormService formService = engine.getFormService();  
 /\*  
 工作流用户操作  
 相关表：act\_id\_group、  
 act\_id\_info、  
 act\_id\_membership、  
 act\_id\_user  
 \*/  
 IdentityService identityService = engine.getIdentityService();  
 /\*  
 管理器  
 \*/  
 ManagementService managementService = engine.getManagementService();  
}

### (5)使用核心API操作展示

<1>部署流程定义

*/\*\*  
 \* 部署流程定义  
 \*/*@Test  
public void deployProcess() {  
 // 得到流程部署的Service  
 RepositoryService repositoryService = this.processEngine.getRepositoryService();  
 Deployment deploy = repositoryService.createDeployment().name("请假流程")  
 .addClasspathResource("HelloWorld.bpmn")  
 .deploy();  
 System.*out*.println("部署成功：流程部署ID："+deploy.getId());  
}

<2>启动流程

*/\*\*  
 \* 启动流程  
 \*/*@Test  
public void startProcess() {  
 RuntimeService runtimeService = this.processEngine.getRuntimeService();  
 String processDefinitionKey = "HelloWorld";  
 runtimeService.startProcessInstanceByKey(processDefinitionKey);  
 System.*out*.println("流程启动成功！");  
}

<3>查询任务

*/\*\*  
 \* 查询任务  
 \*/*@Test  
public void findTask() {  
 TaskService taskService = this.processEngine.getTaskService();  
 String assignee = "张三";  
 List<Task> list = taskService.createTaskQuery().taskAssignee(assignee).list();  
 if (!CollectionUtils.*isEmpty*(list)) {  
 for (Task task : list) {  
 System.*out*.println("任务ID："+task.getId());  
 System.*out*.println("流程实例ID："+task.getProcessDefinitionId());  
 System.*out*.println("执行实例ID："+task.getExecutionId());  
 System.*out*.println("流程定义ID："+task.getProcessDefinitionId());  
 System.*out*.println("任务名称："+task.getName());  
 System.*out*.println("任务办理人："+task.getAssignee());  
 System.*out*.println("#####################################");  
 }  
 }  
}

<4>办理任务

*/\*\*  
 \* 办理任务  
 \*/*@Test  
public void completeTask() {  
 String taskId = "5004";  
 TaskService taskService = this.processEngine.getTaskService();  
 taskService.complete(taskId);  
 System.*out*.println("任务完成");  
}

### (6)详细方法展示

<1>部署的两种方式

*/\*\*  
 \* 部署流程(使用classpath的方式)  
 \*/* @Test  
 public void deployProcess01() {  
 // 得到流程部署的Service  
 RepositoryService repositoryService = this.processEngine.getRepositoryService();  
 Deployment deploy = repositoryService.createDeployment().name("请假流程001")  
 .addClasspathResource("HelloWorld.bpmn")  
 .deploy();  
 System.*out*.println("部署成功：流程部署ID："+deploy.getId());  
 }  
  
 */\*\*  
 \* 部署流程(使用zip文件的方式)  
 \* 流程图文件必须是zip格式  
 \*/* @Test  
 public void deployProcess02() {  
 // 得到流程部署的Service  
 RepositoryService repositoryService = this.processEngine.getRepositoryService();  
 // 获取zip文件流。  
 // HelloWorld.zip表示从当前包下获取文件  
// InputStream inputStream = this.getClass().getResourceAsStream("HelloWorld.zip");  
 // /HelloWorld.zip表示从classpath下获取文件  
 InputStream inputStream = this.getClass().getResourceAsStream("/HelloWorld.zip");  
 // 包装流  
 ZipInputStream zipInputStream = new ZipInputStream(inputStream);  
 // 部署  
 Deployment deploy = repositoryService.createDeployment().name("请假流程001")  
 .addZipInputStream(zipInputStream)  
 .deploy();  
 System.*out*.println("部署成功：流程部署ID："+deploy.getId());  
 }

<2>查询流程部署信息，即act\_re\_deployment表

*/\*\*  
 \* 查询流程部署信息，即act\_re\_deployment表  
 \*/*@Test  
public void queryProcessDef() {  
 // 得到流程部署的Service  
 RepositoryService repositoryService = this.processEngine.getRepositoryService();  
 String deploymentId = "1";  
 // 创建部署信息的查询  
// Deployment deployment = repositoryService.createDeploymentQuery()  
// Long count = repositoryService.createDeploymentQuery()  
 List<Deployment> list = repositoryService.createDeploymentQuery()  
 // 条件  
// .deploymentId(deploymentId) //根据部署ID查询  
// .deploymentName(name) //根据部署名称查询  
// .deploymentTenantId(tenantId) //根据tenantId查询  
// .deploymentNameLike(nameLike) //根据部署名称模糊查询  
// .deploymentTenantIdLike(tenantIdLike) //根据tenantId模糊查询  
 // 排序  
// .orderByDeploymentId().desc() //根据部署ID降序排列// .orderByDeploymenTime().asc() //根据部署时间升序  
// .orderByDeploymentName().asc() //根据部署名称升序  
 // 结果集  
 .list(); //返回list集合，值是List<Deployment>类型  
// .listPage(firstResult, maxResults) //分页返回list集合  
// .singleResult(); //返回单个对象，值是Deployment类型  
// .count(); // 返回数量，值是Long类型  
  
// System.out.println("部署ID:"+deployment.getId());  
// System.out.println("部署名称："+deployment.getName());  
// System.out.println("部署时间："+deployment.getDeploymentTime());  
  
// System.out.println("数量："+count);  
  
 for (Deployment deployment : list) {  
 System.*out*.println("部署ID:"+deployment.getId());  
 System.*out*.println("部署名称："+deployment.getName());  
 System.*out*.println("部署时间："+deployment.getDeploymentTime());  
 System.*out*.println("#####################################");  
 }  
}

<3>查询流程定义信息，即act\_re\_procdef表

*/\*\*  
 \* 查询流程定义信息，即act\_re\_procdef表  
 \*/*@Test  
public void queryProcDef() {  
 RepositoryService repositoryService = this.processEngine.getRepositoryService();  
 List<ProcessDefinition> list = repositoryService.createProcessDefinitionQuery()  
 // 条件  
// .deploymentId(deploymentId) // 根据部署ID查询  
// .deploymentIds(deploymentIds) // 根据部署ID的集合查询，形参类型为Set<String>  
// .processDefinitionId(processDefinitionIds) // 根据流程定义的ids查询  
// .processDefinitionKey(processDefinitionKey) // 根据流程定义的key查询  
// .processDefinitionKeyLike(processDefinitionKey) // 根据流程定义的key模糊查询  
// .processDefinitionName(processDefinitionName) // 根据流程定义的名称查询  
// .processDefinitionNameLike(processDefinitionName) // 根据流程定义的名称模糊查询  
// .processDefinitionResourceName(resourceName) // 根据流程图的BPMN文件名查询  
// .processDefinitionResourceNameLike(resourceName) // 根据流程图的BPMN文件名模糊查询  
// .processDefinitionVersion(processDefinitionVersion) // 根据流程定义的版本查询  
// .processDefinitionVersionGreaterThan(processDefinitionVersion) // 根据版本号大于某个数字查询  
// .processDefinitionVersionGreaterThanOrEquals(processDefinitionVersion) // 根据版本号大于等于某个数字查询  
// .processDefinitionVersionLowerThan(processDefinitionVersion) // 根据版本号小于某个数字查询  
// .processDefinitionVersionLowerThanOrEquals(processDefinitionVersion) // 根据版本号小于等于某个数字查询  
 // 排序  
// .orderByDeploymentId()  
// .orderByProcessDefinitionId()  
// .orderByProcessDefinitionKey()  
// .orderByProcessDefinitionName()  
// .orderByProcessDefinitionVersion()  
 // 结果集  
 .list();  
// .listPage(firstResult, maxResults)  
// .count()  
// .singleResult()  
 if (!CollectionUtils.*isEmpty*(list)) {  
 for (ProcessDefinition pd : list) {  
 System.*out*.println("流程定义ID："+pd.getId());  
 System.*out*.println("流程部署ID："+pd.getDeploymentId());  
 System.*out*.println("流程定义KEY："+pd.getKey());  
 System.*out*.println("流程定义的名称："+pd.getName());  
 System.*out*.println("流程定义的bpmn文件名："+pd.getResourceName());  
 System.*out*.println("流程定义图片名："+pd.getDiagramResourceName());  
 System.*out*.println("流程定义的版本号："+pd.getVersion());  
 System.*out*.println("################################");  
 }  
 }  
}

<4>删除流程定义

*/\*\*  
 \* 启动流程  
 \*/*@Test  
public void startProcess() {  
 RuntimeService runtimeService = this.processEngine.getRuntimeService();  
 String processDefinitionKey = "HelloWorld";  
 runtimeService.startProcessInstanceByKey(processDefinitionKey);  
 System.*out*.println("流程启动成功！");  
}  
*/\*\*  
 \* 删除流程定义  
 \*/*@Test  
public void deleteProcessDef() {  
 RepositoryService repositoryService = this.processEngine.getRepositoryService();  
 String deploymentId = "2501";  
 // 根据流程部署id删除流程定义，若此时该流程定义被发起的流程引用，则删除会因外键约束而失败  
 // 该方法等价于 deleteDeployment(deploymentId, false)  
// repositoryService.deleteDeployment(deploymentId);  
 // 根据流程部署id删除流程定义，若此时该流程定义被发起的流程引用，则一并删除act\_ru\_\*和act\_hi\_\*中引用该定义的流程数据  
// repositoryService.deleteDeployment(deploymentId, true);  
 // 该方法等价于 deleteDeployment(deploymentId, true)  
 repositoryService.deleteDeploymentCascade(deploymentId);  
 System.*out*.println("删除成功");  
}

<5>修改流程定义信息

activiti没有直接提供方法来修改流程定义信息，因为可能该定义信息正在被发起的流程所使

用。那么如何修改呢？

答：需要修改流程图之后重新部署，只要key不变，它的版本号就会+1，并成为一个新的流

程定义，原来的流程定义不变。

<6>查询流程图，根据流程定义ID

*/\*\*  
 \* 查询流程图，根据流程定义ID  
 \*/*@Test  
public void viewProcessImg() {  
 RepositoryService service = this.processEngine.getRepositoryService();  
 String processDefinitionId = "HelloWorld:1:4";  
 InputStream stream = service.getProcessDiagram(processDefinitionId);  
  
 File file = new File("E:/HelloWorld.png");  
 BufferedOutputStream outputStream = null;  
 try {  
 outputStream = new BufferedOutputStream(new FileOutputStream(file));  
 int length = 0;  
 byte[] b = new byte[1024];  
 while ((length = stream.read(b)) != -1) {  
 outputStream.write(b, 0, length);  
 outputStream.flush();  
 }  
 System.*out*.println("查询成功！");  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 if (stream != null) {  
 stream.close();  
 }  
 if (outputStream != null) {  
 outputStream.close();  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

<7>查询流程图，根据流程部署ID

*/\*\*  
 \* 查询流程图，根据流程部署ID  
 \*/*@Test  
public void viewProcessImg2() {  
 RepositoryService service = this.processEngine.getRepositoryService();  
 // 流程部署ID  
 String deploymentId = "1";  
 // 根据流程部署ID获取流程定义对象  
 ProcessDefinition processDefinition = service.createProcessDefinitionQuery().deploymentId(deploymentId).singleResult();  
 // 获取流程定义ID  
 String processDefinitionId = processDefinition.getId();  
 InputStream stream = service.getProcessDiagram(processDefinitionId);  
  
 File file = new File("E:/"+processDefinition.getDiagramResourceName());  
 BufferedOutputStream outputStream = null;  
 try {  
 outputStream = new BufferedOutputStream(new FileOutputStream(file));  
 int length = 0;  
 byte[] b = new byte[1024];  
 while ((length = stream.read(b)) != -1) {  
 outputStream.write(b, 0, length);  
 outputStream.flush();  
 }  
 System.*out*.println("查询成功！");  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 if (stream != null) {  
 stream.close();  
 }  
 if (outputStream != null) {  
 outputStream.close();  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

<8>查询最新版本的流程定义

*/\*\*  
 \* 查询最新版本的流程定义  
 \*/*@Test  
public void queryNewProcessDef() {  
 RepositoryService service = this.processEngine.getRepositoryService();  
 // 获取最新版本的流程定义  
 ProcessDefinitionQuery processDefinitionQuery = service.createProcessDefinitionQuery().latestVersion();  
 List<ProcessDefinition> list = processDefinitionQuery.list();  
  
 for (ProcessDefinition pd : list) {  
 System.*out*.println("流程定义ID："+pd.getId());  
 System.*out*.println("流程部署ID："+pd.getDeploymentId());  
 System.*out*.println("流程定义KEY："+pd.getKey());  
 System.*out*.println("流程定义的名称："+pd.getName());  
 System.*out*.println("流程定义的bpmn文件名："+pd.getResourceName());  
 System.*out*.println("流程定义图片名："+pd.getDiagramResourceName());  
 System.*out*.println("流程定义的版本号："+pd.getVersion());  
 System.*out*.println("################################");  
 }  
}

<9>删除流程定义(删除key相同的所有不同版本的流程定义)

*/\*\*  
 \* 删除流程定义  
 \* (删除key相同的所有不同版本的流程定义)  
 \*/*@Test  
public void deleteAllSameVersion() {  
 RepositoryService service = this.processEngine.getRepositoryService();  
 String processDefinitionKey = "HelloWorld";  
 // 根据流程定义的key查询流程集合  
 List<ProcessDefinition> list = service.createProcessDefinitionQuery().processDefinitionKey(processDefinitionKey).list();  
  
 if (!CollectionUtils.*isEmpty*(list)) {  
 for (ProcessDefinition pd : list) {  
 service.deleteDeployment(pd.getDeploymentId(), true);  
 }  
 }  
}

# Quartz

## 1.只在某一天执行的CRON表达式

--只在2024年12月13号执行的cron表达式

00 53 14 13 12 ? 2024-2024