**MybatisPlus第一天**

上海千锋 文威

课程目标：

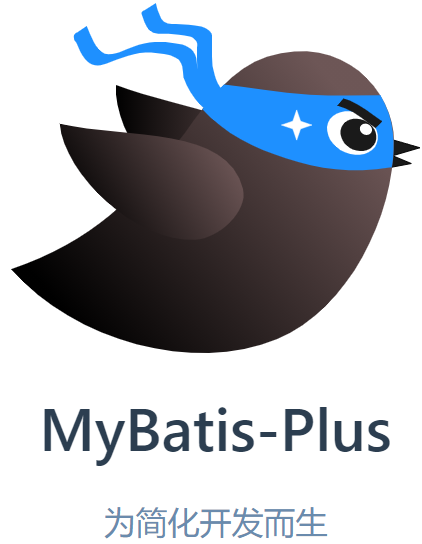
* 了解Mybatis-Plus
* 整合Mybatis-Plus
* 通用CRUD
* Mybatis-Plu的配置
* 条件构造器

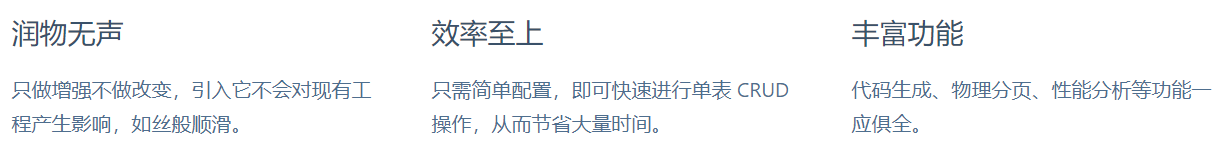
## 1.1 了解Mybatis-Plus

### 1.1.1 Mybatis-Plus介绍

MyBatis-Plus（简称 MP）是一个 MyBatis 的增强工具，在 MyBatis 的基础上只做增强不做改变，为简化开发、提高效率而生。

官网：https://mybatis.plus/ 或 <https://mp.baomidou.com/>







### 1.1.2 代码以及文档

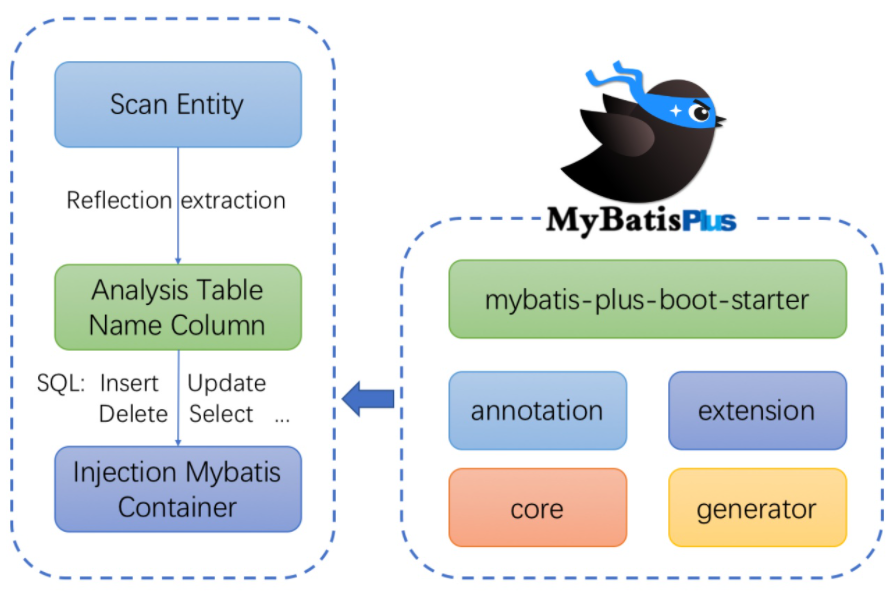
文档地址：<https://mybatis.plus/guide/>

源码地址：<https://github.com/baomidou/mybatis-plus>

### 1.1.3特性







### 1.1.4 mybatisplus作者

Mybatis-Plus是由baomidou（苞米豆）组织开发并且开源的，目前该组织大概有16人左右。

码云地址：<https://gitee.com/organizations/baomidou>



## 1.2 mybatisplus快速开始

对于Mybatis整合MP有常常有三种用法，分别是Mybatis+MP、Spring+Mybatis+MP、Spring Boot+Mybatis+MP。我们今天先用前两种。

### 1.2.1 创建数据表

|  |
| --- |
| -- 创建测试表  CREATE TABLE `tb\_user` (  `id` BIGINT(20) NOT NULL AUTO\_INCREMENT COMMENT '主键ID',  `user\_name` VARCHAR(20) NOT NULL COMMENT '用户名',  `password` VARCHAR(20) NOT NULL COMMENT '密码',  `name` VARCHAR(30) DEFAULT NULL COMMENT '姓名',  `age` INT(11) DEFAULT NULL COMMENT '年龄',  `email` VARCHAR(50) DEFAULT NULL COMMENT '邮箱',  PRIMARY KEY (`id`)  ) ENGINE=INNODB AUTO\_INCREMENT=1 DEFAULT CHARSET=utf8;  -- 插入测试数据  INSERT INTO `tb\_user` (`id`, `user\_name`, `password`, `name`, `age`, `email`) VALUES  ('1', 'zhangsan', '123456', '张三', '18', 'test1@1000phone.com');  INSERT INTO `tb\_user` (`id`, `user\_name`, `password`, `name`, `age`, `email`) VALUES  ('2', 'lisi', '123456', '李四', '20', 'test2@1000phone.com');  INSERT INTO `tb\_user` (`id`, `user\_name`, `password`, `name`, `age`, `email`) VALUES  ('3', 'wangwu', '123456', '王五', '28', 'test3@1000phone.com');  INSERT INTO `tb\_user` (`id`, `user\_name`, `password`, `name`, `age`, `email`) VALUES  ('4', 'zhaoliu', '123456', '赵六', '21', 'test4@1000phone.com');  INSERT INTO `tb\_user` (`id`, `user\_name`, `password`, `name`, `age`, `email`) VALUES  ('5', 'sunqi', '123456', '孙七', '24', 'test5@1000phone.com'); |

### 1.2.2 创建工程

创建一个父工程，导入依赖。

|  |
| --- |
| <**dependencies**> *<!-- mybatis-plus插件依赖 -->* <**dependency**>  <**groupId**>com.baomidou</**groupId**>  <**artifactId**>mybatis-plus</**artifactId**>  <**version**>3.1.1</**version**> </**dependency**> *<!-- MySql -->* <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.47</**version**>  </**dependency**>  *<!-- 连接池 -->* <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid</**artifactId**>  <**version**>1.0.11</**version**>  </**dependency**>  *<!--简化bean代码的工具包-->* <**dependency**>  <**groupId**>org.projectlombok</**groupId**>  <**artifactId**>lombok</**artifactId**>  <**optional**>true</**optional**>  <**version**>1.18.4</**version**>  </**dependency**>  <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.12</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.slf4j</**groupId**>  <**artifactId**>slf4j-log4j12</**artifactId**>  <**version**>1.6.4</**version**>  </**dependency**> </**dependencies**> |

#### 1.2.2.1 Mybatis与Mybatis-Plus整合

* 创建子Module

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  <**parent**>  <**artifactId**>mybatisplus-project</**artifactId**>  <**groupId**>com.mybatisplus</**groupId**>  <**version**>1.0-SNAPSHOT</**version**>  </**parent**>  <**modelVersion**>4.0.0</**modelVersion**>  <**groupId**>com.mybatisplus</**groupId**>  <**artifactId**>mybatisplus-demo1</**artifactId**> </**project**> |

log4j.properties：

|  |
| --- |
| **log4j.rootLogger**=**DEBUG,A1 log4j.appender.A1**=**org.apache.log4j.ConsoleAppender log4j.appender.A1.layout**=**org.apache.log4j.PatternLayout log4j.appender.A1.layout.ConversionPattern**=**[%t] [%c]-[%p] %m%n** |

#### 1.2.2.2 Mybatis实现查询User

第一步，编写mybatis-config.xml文件：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"** *?>* **<!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd"*>*** <**configuration**>  <**environments default="development"**>  <**environment id="development"**>  <**transactionManager type="JDBC"**/>  <**dataSource type="POOLED"**>  <**property name="driver" value="com.mysql.jdbc.Driver"**/>  <**property name="url" value="jdbc:mysql://192.168.10.137:3306/mp"**/>  <**property name="username" value="root"**/>  <**property name="password" value="Admin123!"**/>  </**dataSource**>  </**environment**>  </**environments**>  <**mappers**>  <**package name="com.qf.mapper"**></**package**>  </**mappers**> </**configuration**> |

第二步:编写实体对象

|  |
| --- |
| @Data @NoArgsConstructor @AllArgsConstructor @TableName(**"tb\_user"**) **public class** User {  @TableId(type = IdType.***AUTO***)  **private** Long **id**;  **private** String **userName**;  **private** String **password**;  **private** String **name**;  **private** Integer **age**;  **private** String **email**; } |

第三步：编写UserMapper接口

|  |
| --- |
| **public interface** UserMapper {  List<User> findAll(); } |

第四步: 编写UserMapper.xml文件

|  |
| --- |
| <**mapper namespace="com.qf.mapper.UserMapper"**>  <**select id="findAll" resultType="com.qf.pojo.User"**>  select \* from tb\_user  </**select**> </**mapper**> |

第五步:测试

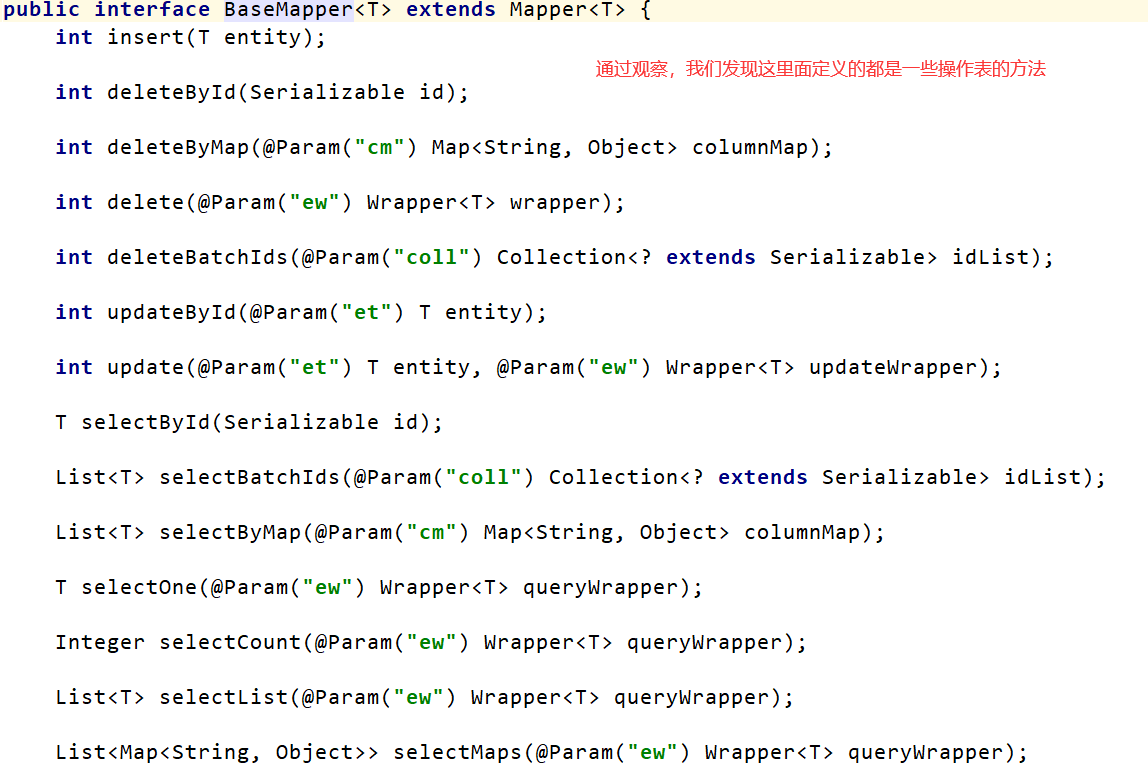
|  |
| --- |
| @Test **public void** testUserList() **throws** Exception{  String resource = **"mybatis-config.xml"**;  InputStream inputStream = Resources.*getResourceAsStream*(resource);  SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);  SqlSession sqlSession = sqlSessionFactory.openSession();  UserMapper userMapper = sqlSession.getMapper(UserMapper.**class**);  List<User> list = userMapper.findAll();  **for** (User user : list) {  System.***out***.println(user);  } } |

### 1.2.3 使用MybatisPlus实现查询User

第一步，将UserMapper继承BaseMapper，将拥有了BaseMapper中的所有方法

|  |
| --- |
| **public interface** UserMapper **extends** BaseMapper<User> { } |

为什么要继承BaseMapper?,我们可以打开BaseMapper的源码查看:

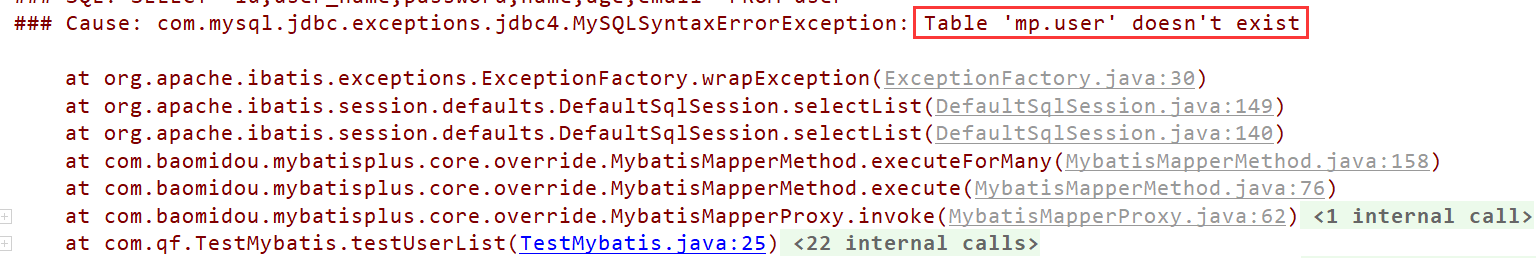


第二步，使用MP中的MybatisSqlSessionFactoryBuilder进程构建

|  |
| --- |
| @Test **public void** testUserList() **throws** Exception{  String resource = **"mybatis-config.xml"**;  InputStream inputStream = Resources.*getResourceAsStream*(resource);  *//这里使用的是MP中的MybatisSqlSessionFactoryBuilder* SqlSessionFactory sqlSessionFactory = **new MybatisSqlSessionFactoryBuilder**().build(inputStream);  SqlSession sqlSession = sqlSessionFactory.openSession();  UserMapper userMapper = sqlSession.getMapper(UserMapper.**class**);  List<User> userList = userMapper.selectList(**null**);  **for** (User user : userList) {  System.***out***.println(user);  } } |

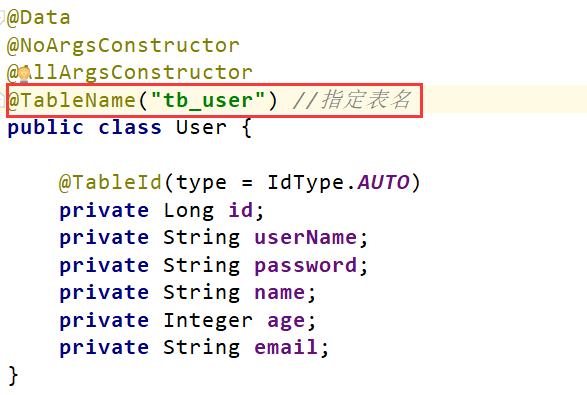
然后测试运行:

发现报错

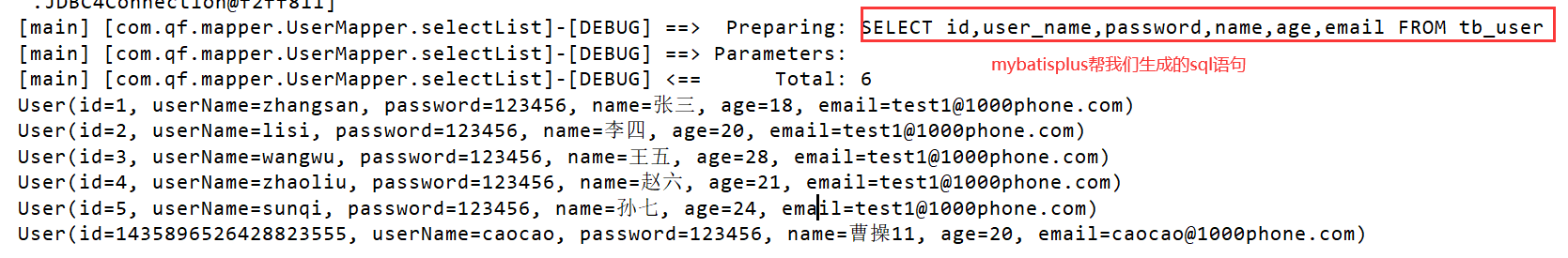


表不存在的异常。

解决：在User对象中添加@TableName，指定数据库表名。



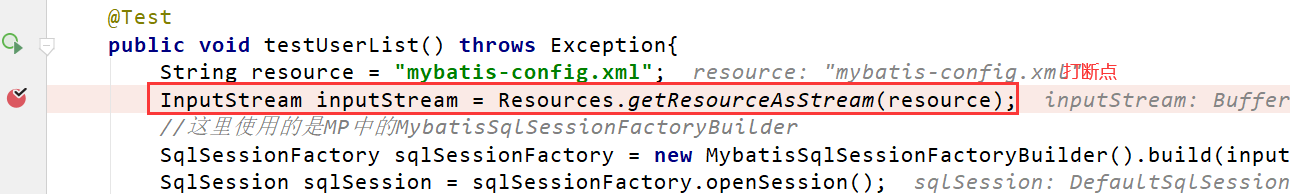
然后再测试:



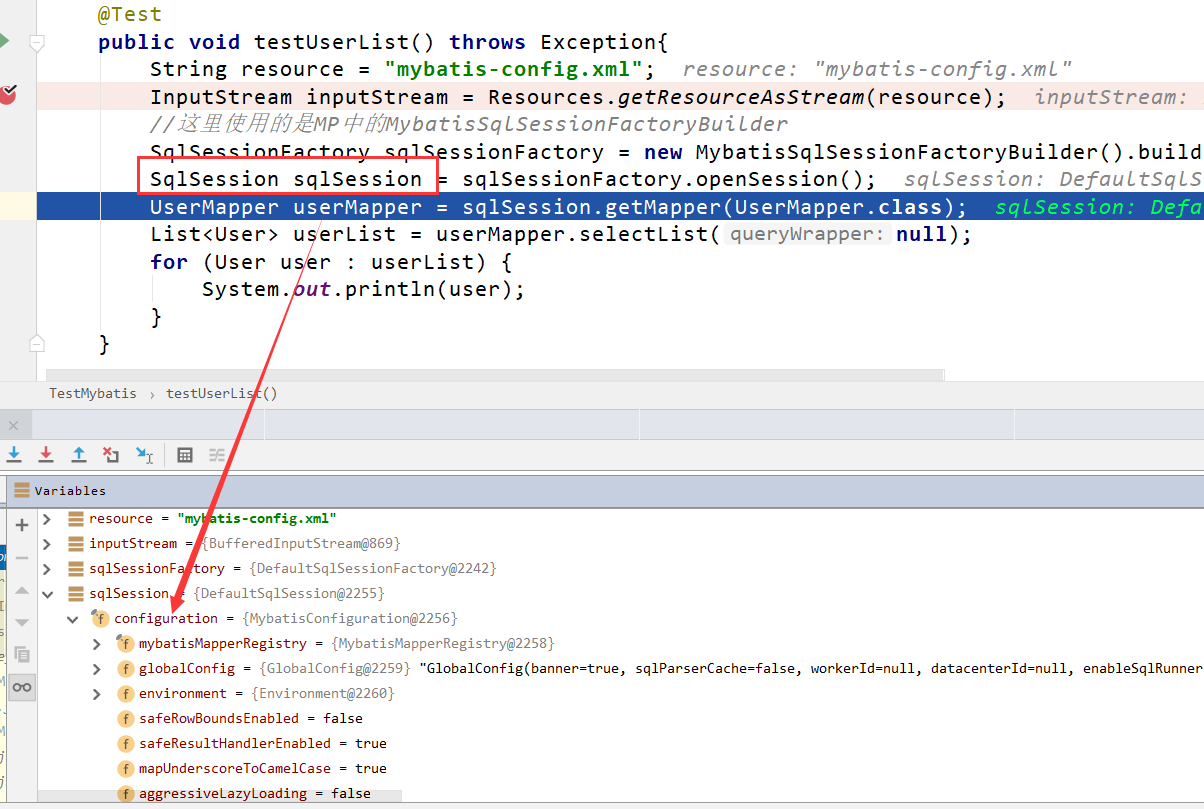
思考？

为什么我们继承了BaseMapper接口，就可以对数据表进行操作?

我们可以自己打断点去跟踪一下。



查看SqlSession的变量信息



找到mappedStatements:



简单说明：

由于使用了MybatisSqlSessionFactoryBuilder进行了构建，继承的BaseMapper中的方法就载入到了 SqlSession中，所以就可以直接使用相关的方法；

### 1.2.4 Spring + Mybatis + MybatisPlus

引入了Spring框架，数据源、构建等工作就交给了Spring管理。

#### 1.2.4.1 创建子模块，引入相关的依赖

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-webmvc</**artifactId**>  <**version**>5.2.0.RELEASE</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-jdbc</**artifactId**>  <**version**>5.2.0.RELEASE</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-test</**artifactId**>  <**version**>5.2.0.RELEASE</**version**>  </**dependency**> </**dependencies**> |

#### 1.2.4.2 创建实体和接口

实体:

|  |
| --- |
| @Data @NoArgsConstructor @AllArgsConstructor @TableName(**"tb\_user"**) *//指定表名* **public class** User {   @TableId(type = IdType.***AUTO***)  **private** Long **id**;  **private** String **userName**;  **private** String **password**;  **private** String **name**;  **private** Integer **age**;  **private** String **email**; } |

接口:

|  |
| --- |
| **public interface** UserMapper **extends** BaseMapper<User> { } |

#### 1.2.4.3 定义配置文件

数据库配置文件 jdbc.properties

|  |
| --- |
| **jdbc.driver**=**com.mysql.jdbc.Driver jdbc.url**=**jdbc:mysql://192.168.10.137:3306/mp?useUnicode=true&characterEncoding=utf8&autoReconnect=true&allowMultiQueries=true&useSSL=false jdbc.username**=**root jdbc.password**=**Admin123!** |

定义spring的配置文件

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd"**>   <**context:property-placeholder location="classpath:jdbc.properties"**/>  *<!-- 定义数据源 -->* <**bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" destroy-method="close"**>  <**property name="url" value="${jdbc.url}"**/>  <**property name="username" value="${jdbc.username}"**/>  <**property name="password" value="${jdbc.password}"**/>  <**property name="driverClassName" value="${jdbc.driver}"**/>  <**property name="maxActive" value="10"**/>  <**property name="minIdle" value="5"**/>  </**bean**>  *<!--这里使用MP提供的sqlSessionFactory，完成了Spring与MP的整合-->* <**bean id="sqlSessionFactory"  class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean"**>  <**property name="dataSource" ref="dataSource"**/>  </**bean**>  *<!--扫描mapper接口，使用的依然是Mybatis原生的扫描器-->* <**bean class="org.mybatis.spring.mapper.MapperScannerConfigurer"**>  <**property name="basePackage" value="com.qf.mapper"**/>  </**bean**> </**beans**> |

#### 1.2.4.4 测试

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.**class**) @ContextConfiguration(locations = **"classpath:applicationContext.xml"**) **public class** TestSpringMP {  @Autowired  **private** UserMapper **userMapper**;  @Test  **public void** testSelectList(){  List<User> users = **this**.**userMapper**.selectList(**null**);  **for** (User user : users) {  System.***out***.println(user);  } } } |

## 1.3 通用的crud操作

通过前面的学习，我们了解到通过继承BaseMapper就可以获取到各种各样的单表操作，接下来我们将详细讲解这些操作。

### 1.3.1 插入操作

* 方法定义:

int insert(T entity)

* 测试用例：

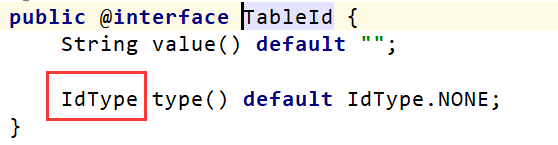
|  |
| --- |
| @Test **public void** testInsert(){  User user = **new** User();  user.setAge(20);  user.setEmail(**"test@1000phone.com"**);  user.setName(**"吕布"**);  user.setUserName(**"lvbu"**);  user.setPassword(**"123456"**);  **int** result = **this**.**userMapper**.insert(user); *//返回的result是受影响的行数，并不是自增后的id* System.***out***.println(**"result = "** + result);  System.***out***.println(user.getId()); *//自增后的id会回填到对象中* } |

我们查看数据表:

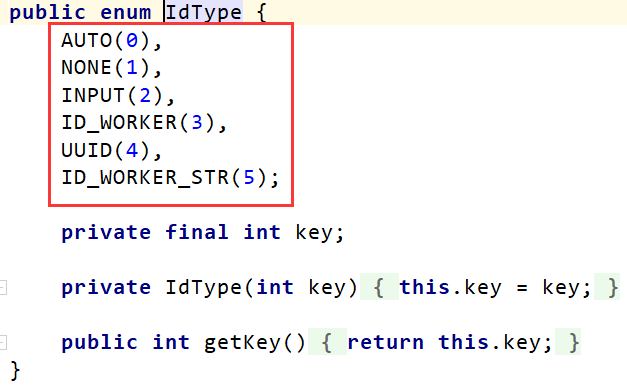


可以看到，数据已经写入到了数据库，但是，id的值不是我们期望的那样，我们期望的是数据库自增长，实际是MP生成了id的值 写入到了数据库。如何设置id的生成策略呢？MP支持的id策略：

我们可以在实体类的id属性上添加@TableId注解，然后点进去查看。



再点IdType查看：



解读：

**AUTO(0):数据库id自增**

NONE(1): 该类型为未设置主键类型。

INPUT(2): 用户输入ID, 该类型可以通过自己注册自动填充插件进行填充。

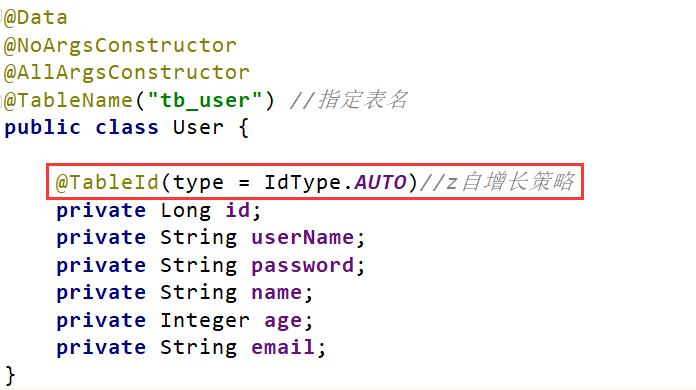
以下3种类型、只有当插入对象ID 为空，才自动填充。

ID\_WORKER(3): 全局唯一ID (idWorker)。

UUID(4): 全局唯一ID (UUID)。

ID\_WORKER\_STR(5):字符串全局唯一ID (idWorker 的字符串表示)。

修改User对象：



清空表重新执行:



此时主键实现自增长了。

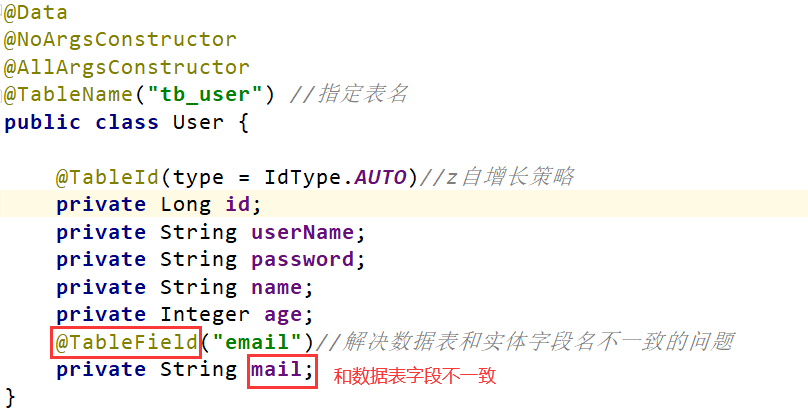
### 1.3.2 @TableField详解

在MP中通过@TableField注解可以指定字段的一些属性，常常解决的问题有2个：

1、对象中的属性名和字段名不一致的问题（非驼峰）

2、对象中的属性字段在表中不存在的问题 使用：

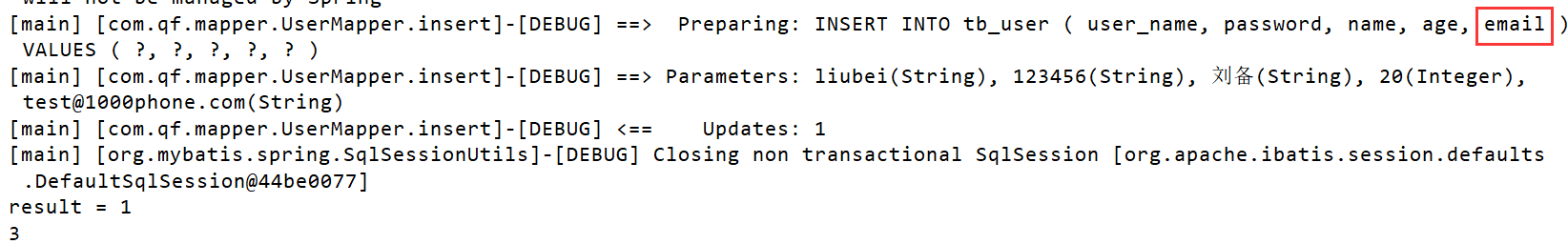
* 解决属性字段不一致的问题



现在我们自己手动映射之后，我们再去执行新增的操作:



查看控制台执行日志信息

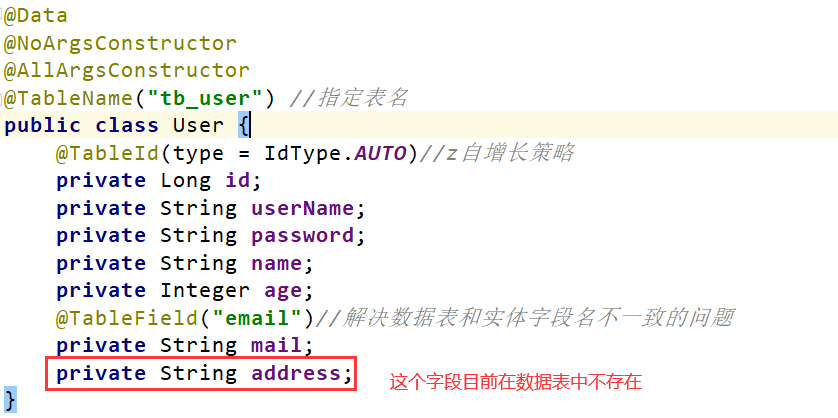


查看数据表信息



* 解决数据表字段不存在的问题

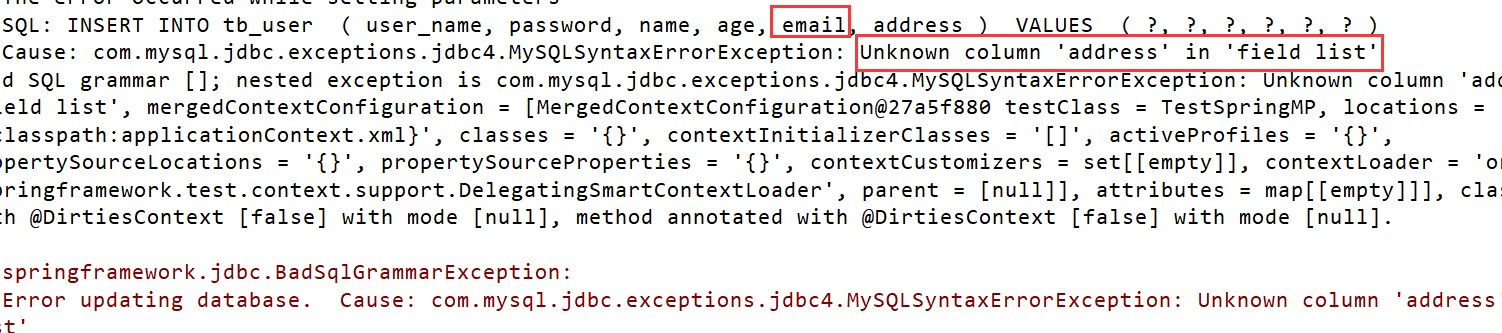
现在我们在实体类中添加一个address字段。



在测试用例中的新增方法也进行setAddress操作



测试执行新增的方法，并查看控制台信息：



这个时候，mybatisplus直接将address作为数据表字段进行插入了。但是数据表中并没有address字段，所以报错。那么我们需要忽略address字段。



再次测试，没有问题。

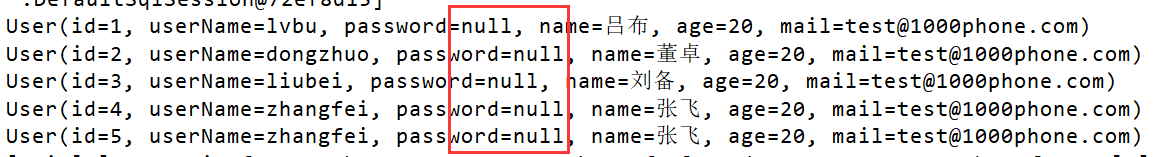
* 不想查询某个字段的数据

比如我不想在查询的时候，显示密码信息



执行查询操作，查看数据信息

|  |
| --- |
| @Test **public void** testSelectList(){  List<User> users = **this**.**userMapper**.selectList(**null**);  **for** (User user : users) {  System.***out***.println(user);  } } |



## 1.4 更新的操作

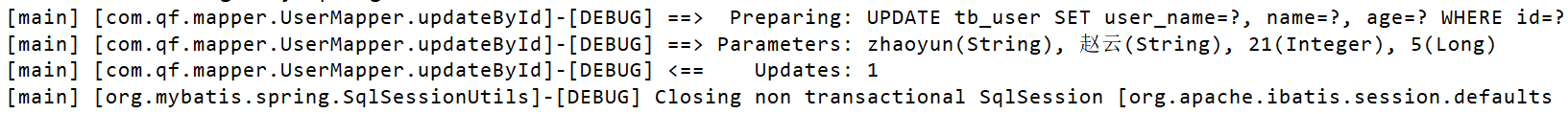
在MP中，更新操作有2种，一种是根据id更新，另一种是根据条件更新。

方法定义：

* int updateById(@Param(Constants.ENTITY) T entity); 根据ID修改

|  |
| --- |
| @Test **public void** testUpdateById() {  User user = **new** User();  user.setId(5L); *//主键* user.setAge(21); *//更新的字段* user.setName(**"赵云"**);  user.setUserName(**"zhaoyun"**);  *//根据id更新，更新不为null的字段* **this**.**userMapper**.updateById(user); } |

结果：





* int update(@Param(Constants.ENTITY) T entity, @Param(Constants.WRAPPER) Wrapper updateWrapper);
* @param entity 实体对象 (set 条件值,可以为 null)。
* @param updateWrapper 实体对象封装操作类(可以为 null,里面的 entity 用于生成 where 语句)。

测试用例1：

|  |
| --- |
| @Test **public void** testUpdateUser1(){  User user = **new** User();  user.setAge(22); *//更新的字段* user.setPassword(**"abcdefg"**);  *//更新的条件* QueryWrapper<User> wrapper = **new** QueryWrapper<>();  wrapper.eq(**"id"**, 5);  *//执行更新操作* **int** result = **this**.**userMapper**.update(user, wrapper);  System.***out***.println(**"result = "** + result); } |

或者还可以这么写

测试用例2:

|  |
| --- |
| @Test **public void** testUpdate2() {  *//更新的条件以及字段* UpdateWrapper<User> wrapper = **new** UpdateWrapper<>();  wrapper.eq(**"id"**, 5).set(**"age"**, 23);  *//执行更新操作* **int** result = **this**.**userMapper**.update(**null**, wrapper);  System.***out***.println(**"result = "** + result); } |

## 1.5删除操作

方法定义：

* int deleteById(Serializable id); 根据id删除

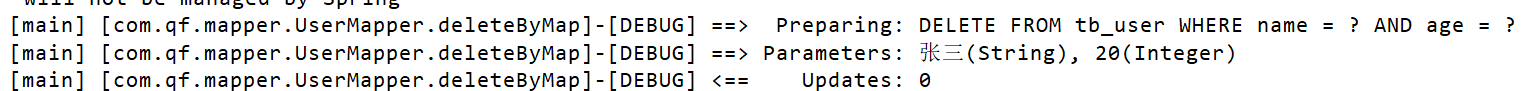
测试用例:

|  |
| --- |
| @Test **public void** testDeleteById() {  *//执行删除操作* **int** result = **this**.**userMapper**.deleteById(5L);  System.***out***.println(**"result = "** + result); } |

* int deleteByMap(@Param(Constants.COLUMN\_MAP) Map columnMap); 根据条件删除

测试用例:

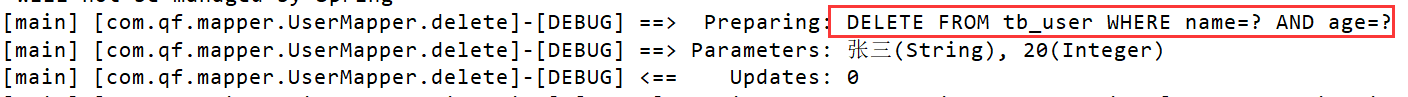
|  |
| --- |
| @Test **public void** testDeleteByMap() {  Map<String, Object> columnMap = **new** HashMap<>();  columnMap.put(**"age"**,20);  columnMap.put(**"name"**,**"张三"**);  *//将columnMap中的元素设置为删除的条件，多个之间为and关系* **int** result = **this**.**userMapper**.deleteByMap(columnMap);  System.***out***.println(**"result = "** + result); } |



* int delete(@Param(Constants.WRAPPER) Wrapper wrapper);

测试用例:

|  |
| --- |
| @Test **public void** testDeleteByMap1() {  User user = **new** User();  user.setAge(20);  user.setName(**"张三"**);  *//将实体对象进行包装，包装为操作条件* QueryWrapper<User> wrapper = **new** QueryWrapper<>(user);  **int** result = **this**.**userMapper**.delete(wrapper);  System.***out***.println(**"result = "** + result); } |

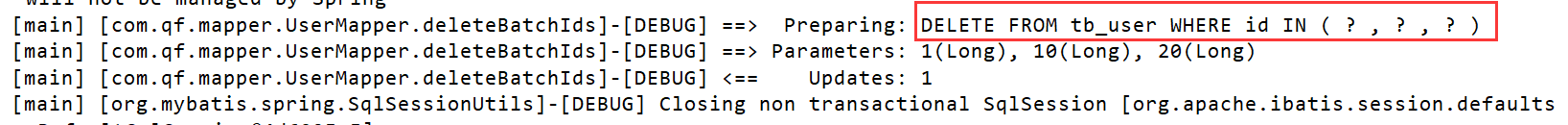


* int deleteBatchIds(@Param(Constants.COLLECTION) Collection idList);

批量删除

测试用例:

|  |
| --- |
| @Test **public void** testDeleteByIds() {  *//根据id集合批量删除* **int** result = **this**.**userMapper**.deleteBatchIds(Arrays.*asList*(1L,10L,20L));  System.***out***.println(**"result = "** + result); } |



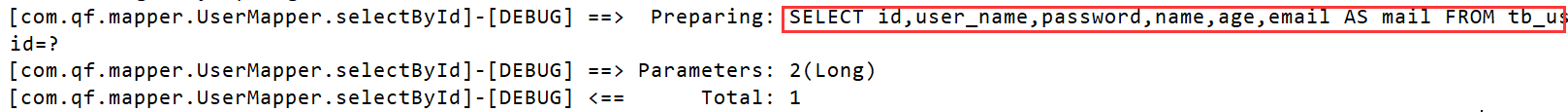
## 1.6 查询操作

MP提供了多种查询操作，包括根据id查询、批量查询、查询单条数据、查询列表、分页查询等操作。

* T selectById(Serializable id) 根据id查询

测试用例：

|  |
| --- |
| @Test **public void** testSelectById() {  *//根据id查询数据* User user = **this**.**userMapper**.selectById(2L);  System.***out***.println(**"result = "** + user); } |



* List selectBatchIds(@Param(Constants.COLLECTION) Collection idList);批量查询

测试用例:

|  |
| --- |
| @Test **public void** testSelectBatchIds() {  *//根据id集合批量查询* List<User> users = **this**.**userMapper**.selectBatchIds(Arrays.*asList*(2L, 3L, 10L));  **for** (User user : users) {  System.***out***.println(user);  } } |

* T selectOne(@Param(Constants.WRAPPER) Wrapper queryWrapper);

根据entity条件，查询一条记录

测试用例:

|  |
| --- |
| @Test **public void** testSelectOne() {  QueryWrapper<User> wrapper = **new** QueryWrapper<User>();  wrapper.eq(**"name"**, **"李四"**);  *//根据条件查询一条数据，如果结果超过一条会报错* User user = **this**.**userMapper**.selectOne(wrapper);  System.***out***.println(user);} } |

我们可以在数据表中早两条name一样的数据，测试是否报错?



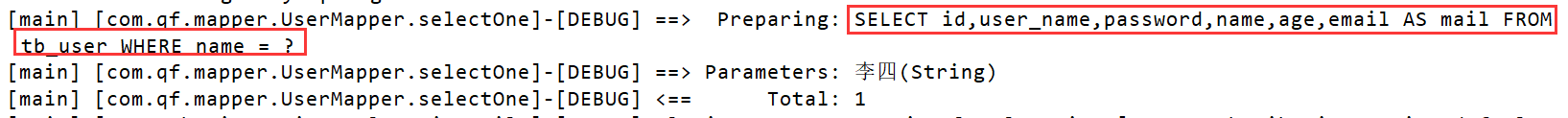
执行查询之后:



* Integer selectCount(@Param(Constants.WRAPPER) Wrapper queryWrapper)查询总记录数

测试用例：

|  |
| --- |
| @Test **public void** testSelectCount() {  QueryWrapper<User> wrapper = **new** QueryWrapper<User>();  wrapper.gt(**"age"**, 23); *//年龄大于23岁  //根据条件查询数据条数* Integer count = **this**.**userMapper**.selectCount(wrapper);  System.***out***.println(**"count = "** + count); } |

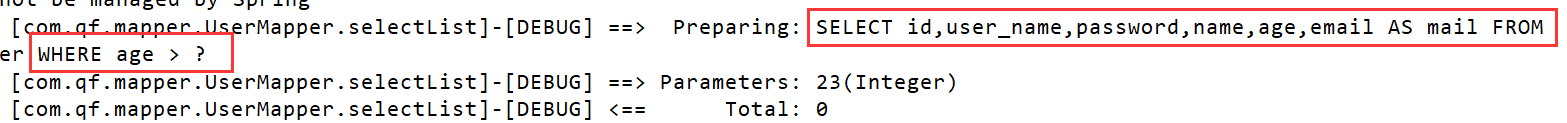


* List selectList(@Param(Constants.WRAPPER) Wrapper queryWrapper)

根据 entity 条件，查询全部记录

测试用例:

|  |
| --- |
| @Test **public void** testSelectList1() {  QueryWrapper<User> wrapper = **new** QueryWrapper<User>();  wrapper.gt(**"age"**, 23); *//年龄大于23岁  //根据条件查询数据* List<User> users = **this**.**userMapper**.selectList(wrapper);  **for** (User user : users) {  System.***out***.println(**"user = "** + user);  } } |



* IPage selectPage(IPage page, @Param(Constants.WRAPPER) Wrapper queryWrapper); 分页查询

第一步：配置分页插件

|  |
| --- |
| <**bean id="mybatisSqlSessionFactoryBean" class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean"**>  *<!--  注入数据源  -->* <**property name="dataSource" ref="dataSource"**></**property**>   **<property name="plugins">  <array>  *<!-- 分页插件配置 -->* <bean id="paginationInterceptor" class="com.baomidou.mybatisplus.extension.plugins.PaginationInterceptor">  <property name="dialectType" value="mysql" />  </bean>  </array>  </property>** </**bean**> |

第二步:测试用例

|  |
| --- |
| @Test **public void** testSelectPage() {  QueryWrapper<User> wrapper = **new** QueryWrapper<User>();  wrapper.gt(**"age"**, 20); *//年龄大于20岁  //参数1 当前页码 参数2 每页显示的条数* Page<User> page = **new** Page<>(1,1);  *//根据条件查询数据* IPage<User> iPage = **this**.**userMapper**.selectPage(page, wrapper);  System.***out***.println(**"数据总条数："** + iPage.getTotal());  System.***out***.println(**"总页数："** + iPage.getPages());  List<User> users = iPage.getRecords();  **for** (User user : users) {  System.***out***.println(**"user = "** + user);  } } |

## 2.1 MybatisPlus配置

### 2.1.1 基本配置

#### 2.1.1.1 configLocation

MyBatis 配置文件位置，如果您有单独的 MyBatis 配置，请将其路径配置到 configLocation 中。MyBatis Configuration 的具体内容请参考MyBatis官方文档。

|  |
| --- |
| <**bean id="sqlSessionFactoryBean" class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean"**>  <**property name="configLocation" value="classpath:mybatis-config.xml"**></**property**> </**bean**> |

#### 2.1.1.2 mapperLocations

MyBatis Mapper 所对应的XML文件位置，如果您在Mapper中有自定义方法（XML 中有自定义实现），需要进行该配置，告诉Mapper所对应的XML文件位置。

|  |
| --- |
| <**bean id="sqlSessionFactoryBean" class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean"**>  <**property name="mapperLocations" value="classpath:mybatis/userMapper.xml"**></**property**> </**bean**> |

#### 2.1.1.3 typeAliasesPackage

MyBaits 别名包扫描路径，通过该属性可以给包中的类注册别名，注册后在 Mapper 对应的 XML 文件中可以直接使 用类名，而不用使用全限定的类名（即 XML 中调用的时候不用包含包名）。

|  |
| --- |
| <**bean id="sqlSessionFactoryBean" class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean"**>  <**property name="typeAliasesPackage" value="com.qf.pojo"**></**property**> </**bean**> |

### 2.1.2 DB策略配置

#### 2.1.2.1 idType

类型：com.baomidou.mybatisplus.annotation.IdType

默认值：ID\_WORKER

全局默认主键类型，设置后，即可省略实体对象中的@TableId(type = IdType.AUTO)配置。

示例：

SpringBoot：

|  |
| --- |
| mybatis-plus.global-config.db-config.id-type=auto |

Spring:

|  |
| --- |
| <**bean id="sqlSessionFactory"  class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean"**>  <**property name="dataSource" ref="dataSource"**/>  **<property name="globalConfig">  <bean class="com.baomidou.mybatisplus.core.config.GlobalConfig">  <property name="dbConfig">  <bean class="com.baomidou.mybatisplus.core.config.GlobalConfig$DbConfig">  <property name="idType" value="AUTO"></property>  </bean>  </property>  </bean>  </property>** </**bean**> |

测试:

|  |
| --- |
| @Test **public void** testInsert(){  User user = **new** User();  user.setAge(20);  user.setMail(**"test@1000phone.com"**);  user.setName(**"吕蒙"**);  user.setUserName(**"lvmeng"**);  user.setPassword(**"123456"**);  **int** result = **this**.**userMapper**.insert(user); *//返回的result是受影响的行数，并不是自增后的id* System.***out***.println(**"result = "** + result);  System.***out***.println(user.getId()); *//自增后的id会回填到对象中* } |



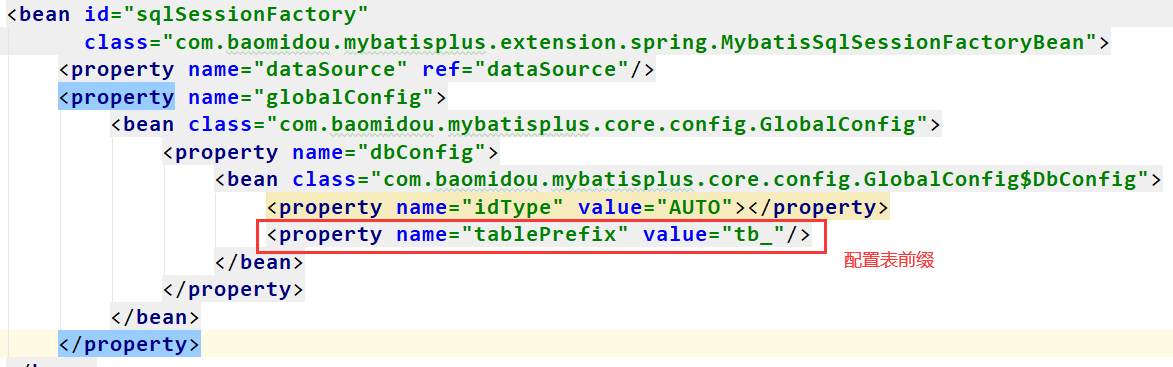
#### 2.1.2.2 tablePrefix

表名前缀，全局配置后可省略@TableName()配置。

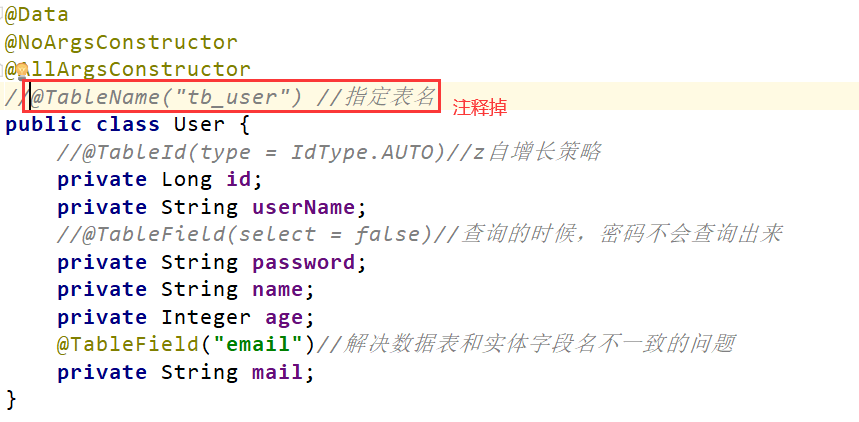
SpringBoot：

|  |
| --- |
| mybatis-plus.global-config.db-config.table-prefix=tb\_ |

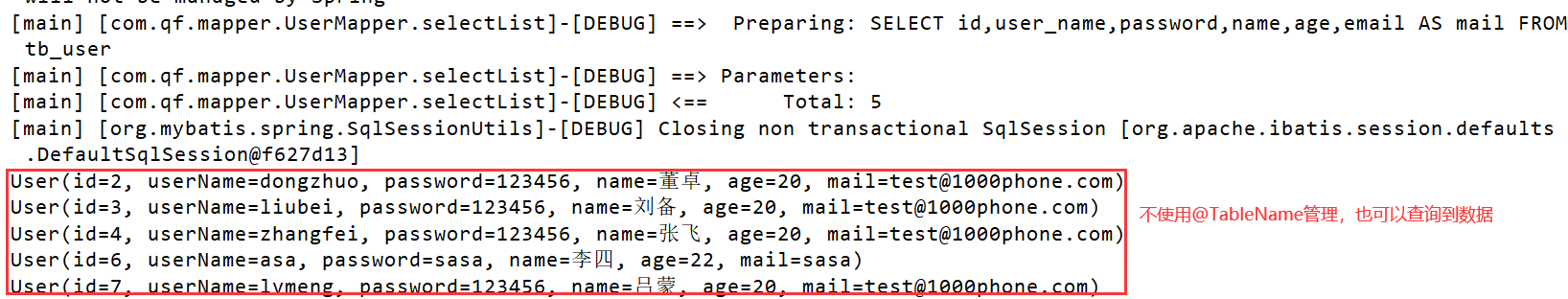
Spring:



测试:

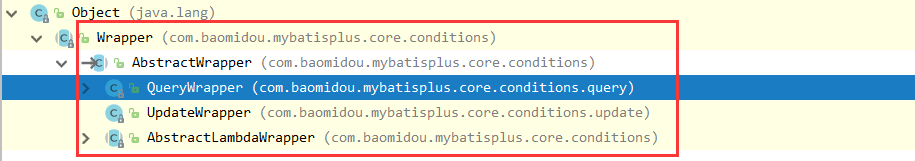


执行查询测试：



## 2.2 条件构造器

在MP中，Wrapper接口的实现类关系如下：



可以看到，AbstractWrapper和AbstractChainWrapper是重点实现，接下来我们重点学习AbstractWrapper以及其子类。

### 2.2.1 allEq

说明:

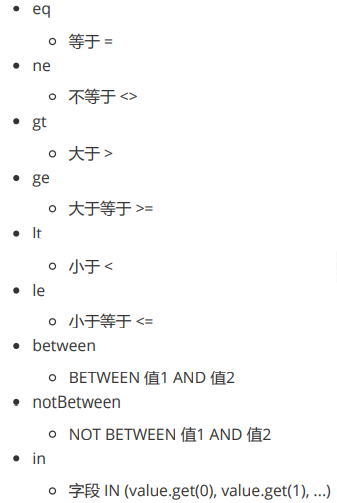
allEq(Map params)

allEq(Map params, boolean null2IsNull)

测试用例:

|  |
| --- |
| @Test **public void** testWrapper() {  QueryWrapper<User> wrapper = **new** QueryWrapper<>();  *//设置条件* Map<String,Object> params = **new** HashMap<>();  params.put(**"name"**, **"曹操"**);  params.put(**"age"**, **"20"**);  params.put(**"password"**, **null**);  wrapper.allEq(params);*//SELECT \* FROM tb\_user WHERE password IS NULL AND name = ? AND age = ?  // wrapper.allEq(params,false); //SELECT \* FROM tb\_user WHERE name = ? AND age= ?* List<User> users = **this**.**userMapper**.selectList(wrapper);  **for** (User user : users) {  System.***out***.println(user);  } } |

基本比较操作:

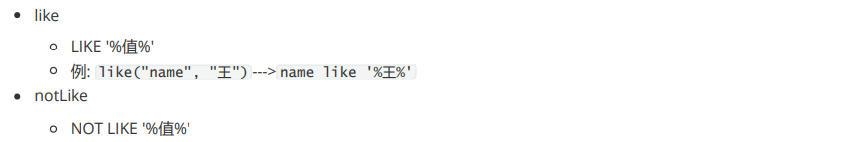


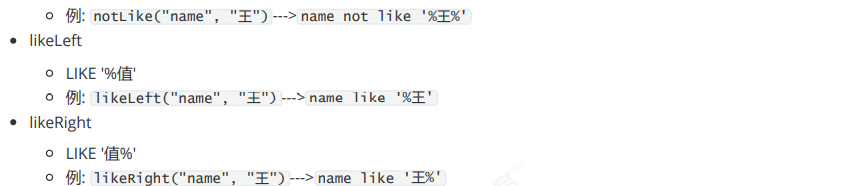
测试用例：

|  |
| --- |
| @Test **public void** testEq() {  QueryWrapper<User> wrapper = **new** QueryWrapper<>();  *//SELECT id,user\_name,password,name,age,email FROM tb\_user WHERE password = ?AND age >= ? AND name IN (?,?,?)* wrapper.eq(**"password"**, **"123456"**)  .ge(**"age"**, 20)  .in(**"name"**, **"李四"**, **"王五"**, **"赵六"**);  List<User> users = **this**.**userMapper**.selectList(wrapper);  **for** (User user : users) {  System.***out***.println(user);  } } |

### 2.2.2 模糊查询

说明：





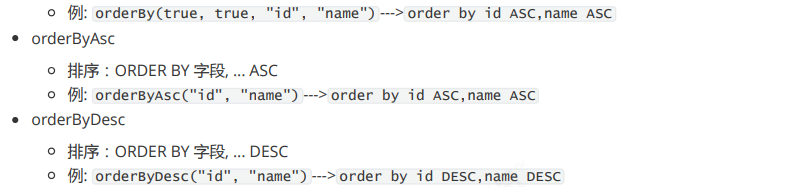
测试用例：

|  |
| --- |
| @Test **public void** testWrapper1() {  QueryWrapper<User> wrapper = **new** QueryWrapper<>();  *//SELECT id,user\_name,password,name,age,email FROM tb\_user WHERE name LIKE ?  //Parameters: %曹%(String)* wrapper.like(**"name"**, **"曹"**);  List<User> users = **this**.**userMapper**.selectList(wrapper);  **for** (User user : users) {  System.***out***.println(user);  } } |

### 2.2.3 排序

说明:



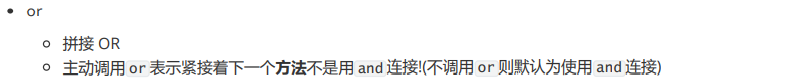


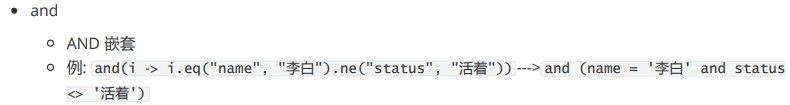
测试用例:

|  |
| --- |
| @Test **public void** testOrder() {  QueryWrapper<User> wrapper = **new** QueryWrapper<>();  *//SELECT id,user\_name,password,name,age,email FROM tb\_user ORDER BY age DESC* wrapper.orderByDesc(**"age"**);  List<User> users = **this**.**userMapper**.selectList(wrapper);  **for** (User user : users) {  System.***out***.println(user);  } } |

### 2.2.4 逻辑查询

说明：





测试用例：

|  |
| --- |
| @Test **public void** testWrapper3() {  QueryWrapper<User> wrapper = **new** QueryWrapper<>();  *//SELECT id,user\_name,password,name,age,email FROM tb\_user WHERE name = ? OR age = ?* wrapper.eq(**"name"**,**"李四"**).or().eq(**"age"**, 24);  List<User> users = **this**.**userMapper**.selectList(wrapper);  **for** (User user : users) {  System.***out***.println(user);  } } |

### 2.2.5 select

在MP查询中，默认查询所有的字段，如果有需要也可以通过select方法进行指定字段。

测试用例：

|  |
| --- |
| @Test **public void** testWrapper4() {  QueryWrapper<User> wrapper = **new** QueryWrapper<>();  *//SELECT id,name,age FROM tb\_user WHERE name = ? OR age = ?* wrapper.eq(**"name"**, **"李四"**)  .or()  .eq(**"age"**, 24)  .select(**"id"**, **"name"**, **"age"**);  List<User> users = **this**.**userMapper**.selectList(wrapper);  **for** (User user : users) {  System.***out***.println(user);  } } |