# MybatisPlus

## MybatisPlus简介

### 简介

**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**。



### 特性

**无侵入**：只做增强不做改变，引入它不会对现有工程产生影响，如丝般顺滑

**损耗小**：启动即会自动注入基本 CURD，性能基本无损耗，直接面向对象操作

**强大的 CRUD 操作**：内置通用 Mapper、通用 Service，仅仅通过少量配置即可实现单表大部分

CRUD 操作，更有强大的条件构造器，满足各类使用需求

**支持 Lambda 形式调用**：通过 Lambda 表达式，方便的编写各类查询条件，无需再担心字段写错

**支持主键自动生成**：支持多达 4 种主键策略（内含分布式唯一 ID 生成器 - Sequence），可自由

配置，完美解决主键问题

**支持 ActiveRecord 模式**：支持 ActiveRecord 形式调用，实体类只需继承 Model 类即可进行强

大的 CRUD 操作

**支持自定义全局通用操作**：支持全局通用方法注入（ Write once, use anywhere ）

**内置代码生成器**：采用代码或者 Maven 插件可快速生成 Mapper 、 Model 、 Service 、

Controller 层代码，支持模板引擎，更有超多自定义配置等您来使用

**内置分页插件**：基于 MyBatis 物理分页，开发者无需关心具体操作，配置好插件之后，写分页等

同于普通 List 查询

**分页插件支持多种数据库**：支持 MySQL、MariaDB、Oracle、DB2、H2、HSQL、SQLite、

Postgre、SQLServer 等多种数据库

**内置性能分析插件**：可输出 SQL 语句以及其执行时间，建议开发测试时启用该功能，能快速揪出

慢查询

**内置全局拦截插件**：提供全表 delete 、 update 操作智能分析阻断，也可自定义拦截规则，预防

误操作

### 支持数据库

任何能使用MyBatis进行 CRUD, 并且支持标准 SQL 的数据库，具体支持情况如下 ：

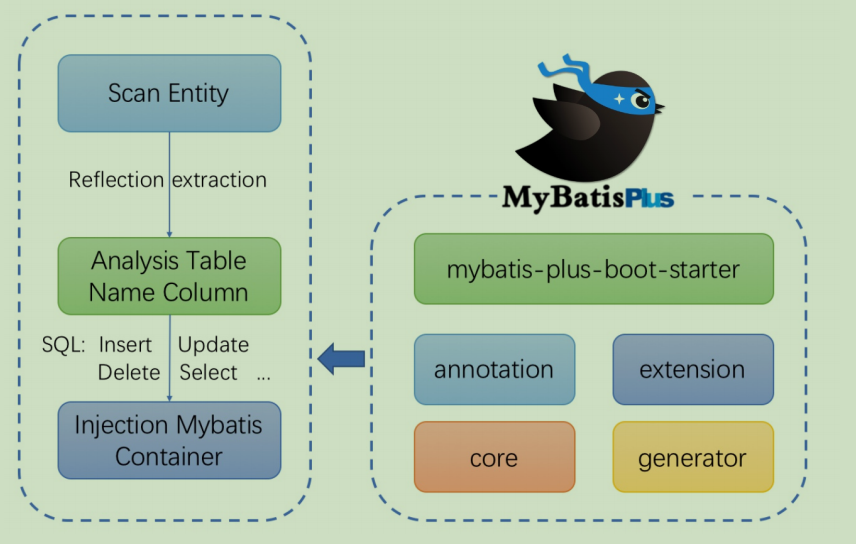
MySQL，Oracle，DB2，H2，HSQL，SQLite，PostgreSQL，SQLServer，Phoenix，Gauss ，

ClickHouse，Sybase，OceanBase，Firebird，Cubrid，Goldilocks，csiidb

达梦数据库，虚谷数据库，人大金仓数据库，南大通用(华库)数据库，南大通用数据库，神通数据

库，瀚高数据库。

### 框架结构



### 代码及文档地址

官方地址: <http://mp.baomidou.com>

代码发布地址:

Github: [https://github.com/baomidou/mybatis-plus](https://github.com/baomidou/mybatis-plus%20)

Gitee: [https://gitee.com/baomidou/mybatis-plus](https://gitee.com/baomidou/mybatis-plus%20)

文档发布地址: https://baomidou.com/pages/24112f

## 入门案例

### 开发环境

IDE：idea 2020.23

JDK：JDK8+

构建工具：maven 3.8.3

MySQL版本：MySQL 8+

Spring Boot：2.7.2

MyBatis-Plus：3.5.1

### 创建数据库和表

#### 创建表

CREATE DATABASE `mybatis\_plus` /\*!40100 DEFAULT CHARACTER SET utf8mb4 \*/; use `mybatis\_plus`;   
CREATE TABLE `user` ( `id` bigint(20) NOT NULL COMMENT '主键ID', `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 DEFAULT CHARSET=utf8;

#### 添加数据

INSERT INTO user (id, name, age, email) VALUES (1, 'Jone', 18, 'test1@baomidou.com'), (2, 'Jack', 20, 'test2@baomidou.com'), (3, 'Tom', 28, 'test3@baomidou.com'), (4, 'Sandy', 21, 'test4@baomidou.com'), (5, 'Billie', 24, 'test5@baomidou.com');

### 编写代码

新建SpringBoot工程...先不勾选依赖，后面手动导入熟悉一下依赖

#### 添加依赖

<dependency>  
 <groupId>com.baomidou</groupId>  
 <artifactId>mybatis-plus-boot-starter</artifactId>  
 <version>3.5.2</version>  
 </dependency>  
  
 <!--lombok简化开发-->  
 <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <optional>true</optional>  
 </dependency>  
   
 <!--数据库驱动-->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <scope>runtime</scope>  
 </dependency>

#### 配置application.yml配置文件

spring:  
 #配置数据源信息  
 datasource:  
 #配置数据源类型  
 type: com.zaxxer.hikari.HikariDataSource  
 driver-class-name: com.mysql.cj.jdbc.Driver  
 url: jdbc:mysql://localhost:3306/mybatis\_plus?serverTimezone=GMT%2B8&characterEncoding=utf-8&userSSL=false  
 username: root  
 password: password

**注意：**

1、驱动类driver-class-name

spring boot 2.0（内置jdbc5驱动），驱动类使用：

driver-class-name: com.mysql.jdbc.Driver

spring boot 2.1及以上（内置jdbc8驱动），驱动类使用：

driver-class-name: com.mysql.cj.jdbc.Driver

否则运行测试用例的时候会有 WARN 信息

2、连接地址url

MySQL5.7版本的url：

jdbc:mysql://localhost:3306/mybatis\_plus?characterEncoding=utf-8&useSSL=false

MySQL8.0版本的url：

jdbc:mysql://localhost:3306/mybatis\_plus?

serverTimezone=GMT%2B8&characterEncoding=utf-8&useSSL=false

否则运行测试用例报告如下错误：

java.sql.SQLException: The server time zone value 'ÖÐ¹ú±ê×¼Ê±¼ä' is unrecognized or

represents more

#### 创建实体类：User

@Data  
public class User {  
  
 private Long id;  
  
 private String name;  
  
 private Integer age;  
  
 private String email;  
}

#### 添加Mapper接口：UserMapper

public interface UserMapper extends BaseMapper<User> {  
  
  
}

#### 在Spring Boot启动类中添加@MapperScan注解，扫描mapper包

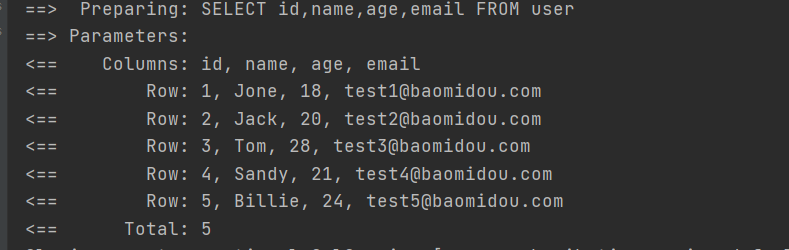
@SpringBootApplication  
//扫描mapper接口所在的包  
@MapperScan("com.lai.mybatisplus.mapper")  
public class MybatisplusApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.run(MybatisplusApplication.class, args);  
 }  
  
}

#### 测试查询

@SpringBootTest  
public class MybatisPlusTest {  
  
 @Autowired  
 private UserMapper userMapper;  
  
 @Test  
 public void testSelectList(){  
 //通过条件构造器查询一个List集合，如果没有条件则可以设置null为参数  
 List<User> list = userMapper.selectList(null);  
 list.forEach(System.out::println);  
 }  
  
}  
  
//输出以下结果  
User(id=1, name=Jone, age=18, email=test1@baomidou.com)  
User(id=2, name=Jack, age=20, email=test2@baomidou.com)  
User(id=3, name=Tom, age=28, email=test3@baomidou.com)  
User(id=4, name=Sandy, age=21, email=test4@baomidou.com)  
User(id=5, name=Billie, age=24, email=test5@baomidou.com)

#### 添加日志

mybatis-plus:  
 configuration:  
 log-impl: org.apache.ibatis.logging.stdout.StdOutImpl



## 基本CRUD

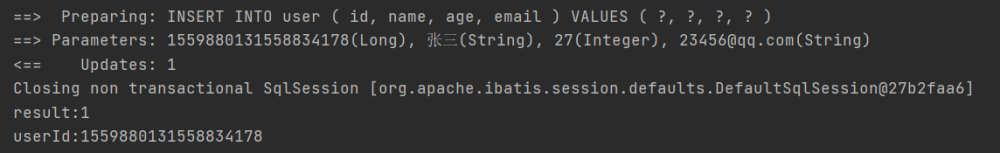
### BaseMapper

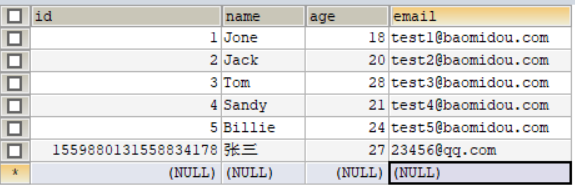
public interface BaseMapper<T> extends Mapper<T> {  
  
 /\*\*  
 \* 插入一条记录  
 \*  
 \* @param entity 实体对象  
 \*/  
 int insert(T entity);  
  
 /\*\*  
 \* 根据 ID 删除  
 \*  
 \* @param id 主键ID  
 \*/  
 int deleteById(Serializable id);  
  
 /\*\*  
 \* 根据实体(ID)删除  
 \*  
 \* @param entity 实体对象  
 \* @since 3.4.4  
 \*/  
 int deleteById(T entity);  
  
 /\*\*  
 \* 根据 columnMap 条件，删除记录  
 \*  
 \* @param columnMap 表字段 map 对象  
 \*/  
 int deleteByMap(@Param(Constants.COLUMN\_MAP) Map<String, Object> columnMap);  
  
 /\*\*  
 \* 根据 entity 条件，删除记录  
 \*  
 \* @param queryWrapper 实体对象封装操作类（可以为 null,里面的 entity 用于生成 where 语句）  
 \*/  
 int delete(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);  
  
 /\*\*  
 \* 删除（根据ID或实体 批量删除）  
 \*  
 \* @param idList 主键ID列表或实体列表(不能为 null 以及 empty)  
 \*/  
 int deleteBatchIds(@Param(Constants.COLL) Collection<?> idList);  
  
 /\*\*  
 \* 根据 ID 修改  
 \*  
 \* @param entity 实体对象  
 \*/  
 int updateById(@Param(Constants.ENTITY) T entity);  
  
 /\*\*  
 \* 根据 whereEntity 条件，更新记录  
 \*  
 \* @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）  
 \*/  
 int update(@Param(Constants.ENTITY) T entity, @Param(Constants.WRAPPER) Wrapper<T> updateWrapper);  
  
 /\*\*  
 \* 根据 ID 查询  
 \*  
 \* @param id 主键ID  
 \*/  
 T selectById(Serializable id);  
  
 /\*\*  
 \* 查询（根据ID 批量查询）  
 \*  
 \* @param idList 主键ID列表(不能为 null 以及 empty)  
 \*/  
 List<T> selectBatchIds(@Param(Constants.COLL) Collection<? extends Serializable> idList);  
  
 /\*\*  
 \* 查询（根据 columnMap 条件）  
 \*  
 \* @param columnMap 表字段 map 对象  
 \*/  
 List<T> selectByMap(@Param(Constants.COLUMN\_MAP) Map<String, Object> columnMap);  
  
 /\*\*  
 \* 根据 entity 条件，查询一条记录  
 \* <p>查询一条记录，例如 qw.last("limit 1") 限制取一条记录, 注意：多条数据会报异常</p>  
 \*  
 \* @param queryWrapper 实体对象封装操作类（可以为 null）  
 \*/  
 default T selectOne(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper) {  
 List<T> ts = this.selectList(queryWrapper);  
 if (CollectionUtils.isNotEmpty(ts)) {  
 if (ts.size() != 1) {  
 throw ExceptionUtils.mpe("One record is expected, but the query result is multiple records");  
 }  
 return ts.get(0);  
 }  
 return null;  
 }  
  
 /\*\*  
 \* 根据 Wrapper 条件，判断是否存在记录  
 \*  
 \* @param queryWrapper 实体对象封装操作类  
 \* @return 是否存在记录  
 \*/  
 default boolean exists(Wrapper<T> queryWrapper) {  
 Long count = this.selectCount(queryWrapper);  
 return null != count && count > 0;  
 }  
  
 /\*\*  
 \* 根据 Wrapper 条件，查询总记录数  
 \*  
 \* @param queryWrapper 实体对象封装操作类（可以为 null）  
 \*/  
 Long selectCount(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);  
  
 /\*\*  
 \* 根据 entity 条件，查询全部记录  
 \*  
 \* @param queryWrapper 实体对象封装操作类（可以为 null）  
 \*/  
 List<T> selectList(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);  
  
 /\*\*  
 \* 根据 Wrapper 条件，查询全部记录  
 \*  
 \* @param queryWrapper 实体对象封装操作类（可以为 null）  
 \*/  
 List<Map<String, Object>> selectMaps(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);  
  
 /\*\*  
 \* 根据 Wrapper 条件，查询全部记录  
 \* <p>注意： 只返回第一个字段的值</p>  
 \*  
 \* @param queryWrapper 实体对象封装操作类（可以为 null）  
 \*/  
 List<Object> selectObjs(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);  
  
 /\*\*  
 \* 根据 entity 条件，查询全部记录（并翻页）  
 \*  
 \* @param page 分页查询条件（可以为 RowBounds.DEFAULT）  
 \* @param queryWrapper 实体对象封装操作类（可以为 null）  
 \*/  
 <P extends IPage<T>> P selectPage(P page, @Param(Constants.WRAPPER) Wrapper<T> queryWrapper);  
  
 /\*\*  
 \* 根据 Wrapper 条件，查询全部记录（并翻页）  
 \*  
 \* @param page 分页查询条件  
 \* @param queryWrapper 实体对象封装操作类  
 \*/  
 <P extends IPage<Map<String, Object>>> P selectMapsPage(P page, @Param(Constants.WRAPPER) Wrapper<T> queryWrapper);  
}

### 插入

@Test  
 public void testInsert(){  
 // INSERT INTO user ( id, name, age, email ) VALUES ( ?, ?, ?, ? )  
 User user = new User(null, "张三", 27, "23456@qq.com");  
 int result = userMapper.insert(user);  
 System.out.println("result:"+result);  
 System.out.println("userId:"+user.getId());  
 }

Mybatis采用雪花算法给id赋值





### 删除

#### 通过id删除

@Test  
 public void testDelete(){  
 int result = userMapper.deleteById(1559883927642972161L);  
 System.out.println("result:"+result);  
 }

#### 通过id批量删除

@Test  
 public void testDelete(){  
 List<Long> list = Arrays.asList(1L, 2L, 3L);  
 int result = userMapper.deleteBatchIds(list);  
 System.out.println("result:"+result);  
  
 }

#### 通过Map条件删除记录

@Test  
 public void testDelete(){  
 Map<String,Object> map = new HashMap<>();  
 map.put("name","张三");  
 map.put("age",28);  
 int result = userMapper.deleteByMap(map);  
 System.out.println("result:"+result);  
 }  
}

### 修改

@Test  
 public void testUpdate(){  
 User user = new User(4L,"李四",26,"123@qq.com");  
 int result = userMapper.updateById(user);  
 System.out.println("result:"+result);  
 }

### 查询

#### 通过id查询

@Test  
 public void testSelectById(){  
 User user = userMapper.selectById(1L);  
 System.out.println(user);  
 }

#### 根据多个id查询多个用户信息

@Test  
 public void testSelectById(){  
 List<Long> list = Arrays.asList(1L, 2L, 3L);  
 List<User> users = userMapper.selectBatchIds(list);  
 users.forEach(System.out::println);  
 }

#### 根据Map条件查询

@Test  
 public void testSelectById(){  
 Map<String, Object> map = new HashMap<>();  
 //添加两个条件  
 map.put("name","jack");  
 map.put("age",20);  
 List<User> users = userMapper.selectByMap(map);  
 users.forEach(System.out::println);  
 }  
//SELECT id,name,age,email FROM user WHERE name = ? AND age = ?

#### 查询所有信息

@Test  
 public void testSelectById(){  
 List<User> list = userMapper.selectList(null);   
 list.forEach(System.out::println);  
 }

#### 测试自定义查询

public interface UserMapper extends BaseMapper<User> {  
  
 /\*\*  
 \* 根据id查询用户信息为map集合  
 \* @param id  
 \* @return  
 \*/  
 Map<String,Object> selectMapById(Long id);  
}

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
  
<mapper namespace="com.lai.mybatisplus.mapper.UserMapper">  
  
 <!--Map<String,Object> selectMapById(Long id);-->  
 <select id="selectMapById" resultType="Map">  
 select id, name, age, email from user where id = #{id}  
 </select>  
  
</mapper>

@Test  
 public void testSelectById(){  
 Map<String, Object> map = userMapper.selectMapById(1L);  
 System.out.println(map);  
 }

### 通用Service接口

说明:

通用 Service CRUD 封装IService接口，进一步封装 CRUD 采用 get 查询单行 remove 删

除 list 查询集合 page 分页 前缀命名方式区分 Mapper 层避免混淆， 泛型 T 为任意实体对象

建议如果存在自定义通用 Service 方法的可能，请创建自己的 IBaseService 继承 Mybatis-Plus 提供的基类

官网地址：https://baomidou.com/pages/49cc81/#service-crud-%E6%8E%A5%E5%8F%

A3

MyBatis-Plus中有一个接口 IService和其实现类 ServiceImpl，封装了常见的业务层逻辑

详情查看源码IService和ServiceImpl

#### 创建Service接口和实现类

public interface UserService extends IService<User> {  
  
}

@Service  
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements UserService {  
  
}

#### 查询总记录数

@SpringBootTest  
public class MybatisPlusServiceTest {  
  
 @Autowired  
 private UserService userService;  
  
 @Test  
 public void testGetCount(){  
 long count = userService.count();  
 System.out.println("总记录数："+count);  
 }  
}

#### 批量添加

@Test  
 public void testInsertMore(){  
 List<User> list = new ArrayList<>();  
 for (int i = 0; i < 10; i++) {  
 User user = new User();  
 user.setName("lai"+i);  
 user.setAge(20+i);  
 list.add(user);  
 }  
 boolean b = userService.saveBatch(list);  
 System.out.println(b);  
 }

## 常用注解

### @TableName

在实体类类型上添加@TableName("t\_user")，标识实体类对应的表，即可成功执行SQL语句

//设置实体类所对应的表名  
@TableName("t\_user")  
@Data  
public class User {  
 ...  
 }

在开发的过程中，我们经常遇到以上的问题，即实体类所对应的表都有固定的前缀，例如t\_或tbl\_

此时，可以使用MyBatis-Plus提供的全局配置，为实体类所对应的表名设置默认的前缀，那么就

不需要在每个实体类上通过@TableName标识实体类对应的表

#MybatisPlus全局配置  
 global-config:  
 db-config:  
 table-prefix: t\_

### @TableId

经过以上的测试，MyBatis-Plus在实现CRUD时，会默认将id作为主键列，并在插入数据时，默认

基于雪花算法的策略生成id

若实体类和表中表示主键的不是id，而是其他字段，例如uid，MyBatis-Plus会自动识别uid为主

键列吗？

我们实体类中的属性id改为uid，将表中的字段id也改为uid，测试添加功能

**值 描述**

IdType.ASSIGN\_ID（默 认）

基于雪花算法的策略生成数据id，与数据库id是否设置自增无关

IdType.AUTO

使用数据库的自增策略，注意，该类型请确保数据库设置了id自增， 否则无效

程序抛出异常，Field 'uid' doesn't have a default value，说明MyBatis-Plus没有将uid作为主键

赋值

在实体类中uid属性上通过@TableId将其标识为主键，即可成功执行SQL语句

public class User {  
  
 //将属性所对应的字段标识为主键  
 @TableId  
 private Long uid;  
 ...  
 }

#### @TableId的value属性

若实体类中主键对应的属性为id，而数据库表中表示主键的字段为uid，此时若只在属性id上添加注解

@TableId，则抛出异常Unknown column 'id' in 'field list'，即MyBatis-Plus仍然会将id作为表的

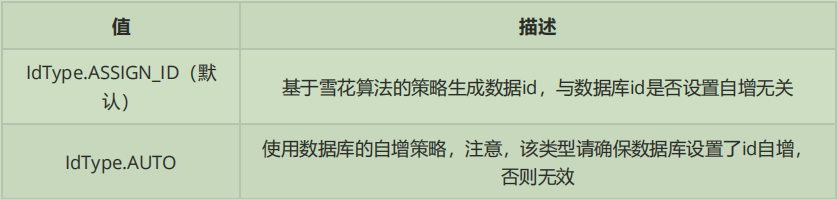
主键操作，而表中表示主键的是字段uid

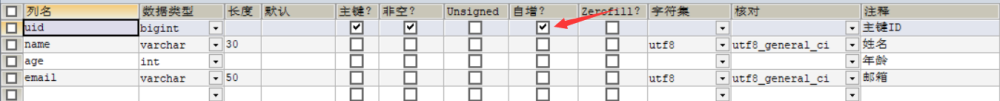
此时需要通过@TableId注解的value属性，指定表中的主键字段，@TableId("uid")或@TableId(value="uid")

#### @TableId的type属性

type属性用来定义主键策略

##### 常见主键生成策略





public class User {  
  
 //将属性所对应的字段标识为主键,并设置主键生成策略（主键自增）  
 @TableId(value = "uid",type = IdType.AUTO)  
 private Long id;  
 ...  
 }

##### 配置全局主键生成策略

mybatis-plus:  
 configuration:  
 log-impl: org.apache.ibatis.logging.stdout.StdOutImpl  
 #MybatisPlus全局配置  
 global-config:  
 db-config:  
 table-prefix: t\_  
 #设置统一的主键生成策略  
 id-type: auto

##### 雪花算法

**背景**

需要选择合适的方案去应对数据规模的增长，以应对逐渐增长的访问压力和数据量。

数据库的扩展方式主要包括：业务分库、主从复制，数据库分表。

**数据库分表**

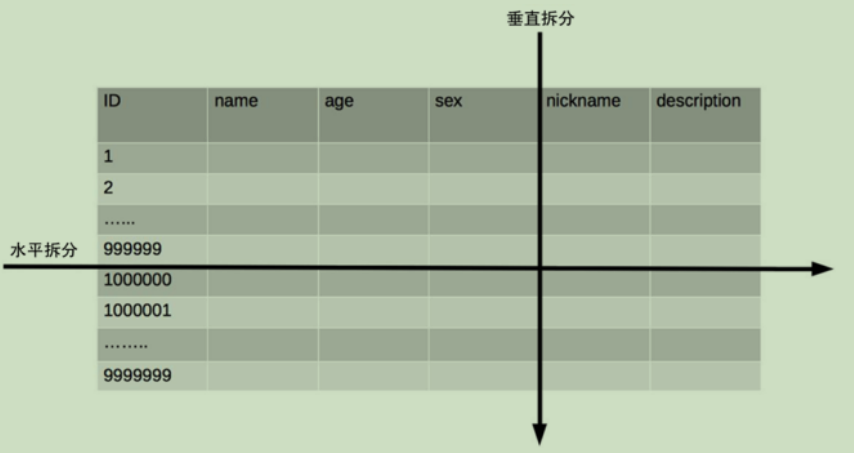
将不同业务数据分散存储到不同的数据库服务器，能够支撑百万甚至千万用户规模的业务，但如果业务

继续发展，同一业务的单表数据也会达到单台数据库服务器的处理瓶颈。例如，淘宝的几亿用户数据，

如果全部存放在一台数据库服务器的一张表中，肯定是无法满足性能要求的，此时就需要对单表数据进

行拆分。

单表数据拆分有两种方式：垂直分表和水平分表。示意图如下：



**垂直分表**

垂直分表适合将表中某些不常用且占了大量空间的列拆分出去。

例如，前面示意图中的 nickname 和 description 字段，假设我们是一个婚恋网站，用户在筛选其他用

户的时候，主要是用 age 和 sex 两个字段进行查询，而 nickname 和 description 两个字段主要用于展

示，一般不会在业务查询中用到。description 本身又比较长，因此我们可以将这两个字段独立到另外

一张表中，这样在查询 age 和 sex 时，就能带来一定的性能提升。

**水平分表**

水平分表适合表行数特别大的表，有的公司要求单表行数超过 5000 万就必须进行分表，这个数字可以

作为参考，但并不是绝对标准，关键还是要看表的访问性能。对于一些比较复杂的表，可能超过 1000

万就要分表了；而对于一些简单的表，即使存储数据超过 1 亿行，也可以不分表。

但不管怎样，当看到表的数据量达到千万级别时，作为架构师就要警觉起来，因为这很可能是架构的性

能瓶颈或者隐患。

水平分表相比垂直分表，会引入更多的复杂性，例如要求全局唯一的数据id该如何处理

**主键自增**

①以最常见的用户 ID 为例，可以按照 1000000 的范围大小进行分段，1 ~ 999999 放到表 1中，

1000000 ~ 1999999 放到表2中，以此类推。

②复杂点：分段大小的选取。分段太小会导致切分后子表数量过多，增加维护复杂度；分段太大可能会

导致单表依然存在性能问题，一般建议分段大小在 100 万至 2000 万之间，具体需要根据业务选取合适

的分段大小。

③优点：可以随着数据的增加平滑地扩充新的表。例如，现在的用户是 100 万，如果增加到 1000 万，

只需要增加新的表就可以了，原有的数据不需要动。

④缺点：分布不均匀。假如按照 1000 万来进行分表，有可能某个分段实际存储的数据量只有 1 条，而

另外一个分段实际存储的数据量有 1000 万条。

**取模**

①同样以用户 ID 为例，假如我们一开始就规划了 10 个数据库表，可以简单地用 user\_id % 10 的值来

表示数据所属的数据库表编号，ID 为 985 的用户放到编号为 5 的子表中，ID 为 10086 的用户放到编号

为 6 的子表中。

②复杂点：初始表数量的确定。表数量太多维护比较麻烦，表数量太少又可能导致单表性能存在问题。

③优点：表分布比较均匀。

④缺点：扩充新的表很麻烦，所有数据都要重分布。

**雪花算法**

雪花算法是由Twitter公布的分布式主键生成算法，它能够保证不同表的主键的不重复性，以及相同表的

主键的有序性。

①核心思想：

长度共64bit（一个long型）。

首先是一个符号位，1bit标识，由于long基本类型在Java中是带符号的，最高位是符号位，正数是0，负

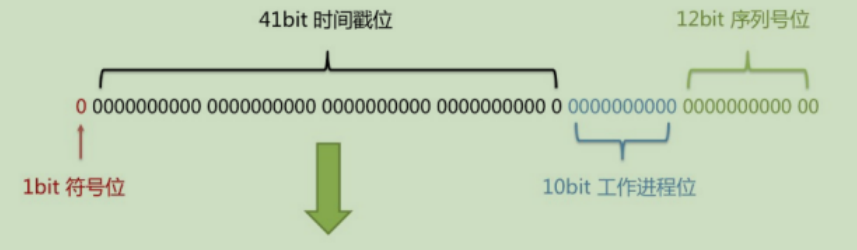
数是1，所以id一般是正数，最高位是0。

41bit时间截(毫秒级)，存储的是时间截的差值（当前时间截 - 开始时间截)，结果约等于69.73年。

10bit作为机器的ID（

5个bit是数据中心，5个bit的机器ID，可以部署在1024个节点）。

12bit作为毫秒内的流水号（意味着每个节点在每毫秒可以产生 4096 个 ID）。



②优点：整体上按照时间自增排序，并且整个分布式系统内不会产生ID碰撞，并且效率较高。

### @TableField

经过以上的测试，我们可以发现，MyBatis-Plus在执行SQL语句时，要保证实体类中的属性名和

表中的字段名一致

如果实体类中的属性名和字段名不一致的情况，会出现什么问题呢？

**a>情况1**

若实体类中的属性使用的是驼峰命名风格，而表中的字段使用的是下划线命名风格

例如实体类属性userName，表中字段user\_name

此时MyBatis-Plus会自动将下划线命名风格转化为驼峰命名风格

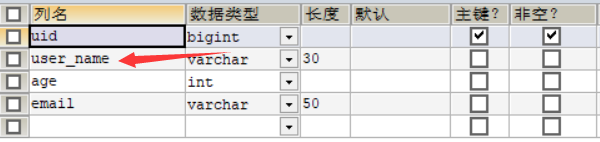
相当于在MyBatis中配置

**b>情况2**

若实体类中的属性和表中的字段不满足情况1

例如实体类属性name，表中字段username

此时需要在实体类属性上使用@TableField("username")设置属性所对应的字段名



public class User {  
  
 //将属性所对应的字段标识为主键,并设置主键生成策略（主键自增）  
 @TableId("uid")  
 private Long id;  
 //设置属性所对应的字段名  
 @TableField("user\_name")  
 private String name;  
 ...  
 }

### @TableLogic

**逻辑删除**

物理删除：真实删除，将对应数据从数据库中删除，之后查询不到此条被删除的数据

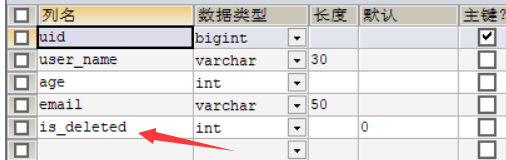
逻辑删除：假删除，将对应数据中代表是否被删除字段的状态修改为“被删除状态”，之后在数据库

中仍旧能看到此条数据记录

使用场景：可以进行数据恢复

**实现逻辑删除**

**step1：**数据库中创建逻辑删除状态列，设置默认值为0



**step2：**实体类中添加逻辑删除属性

public class User {  
  
 ...  
   
 @TableLogic  
 private Integer isDeleted;  
}

**step3：**测试

测试删除功能，真正执行的是修改

UPDATE t\_user SET is\_deleted=1 WHERE id=? AND is\_deleted=0

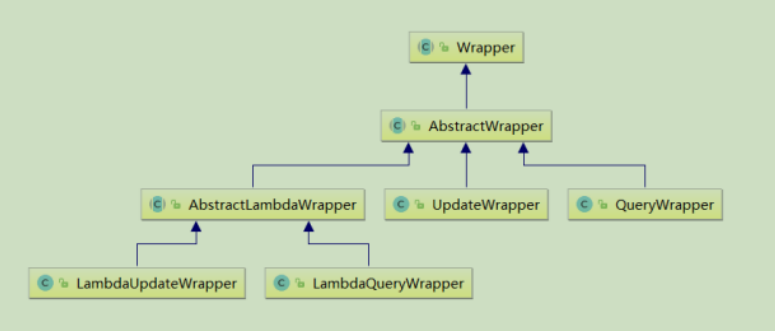
测试查询功能，被逻辑删除的数据默认不会被查询

SELECT id,username AS name,age,email,is\_deleted FROM t\_user WHERE is\_deleted=0

数据库里仍然有数据

## 条件构造器和常用接口

### wapper介绍



Wrapper ： 条件构造抽象类，最顶端父类

AbstractWrapper ： 用于查询条件封装，生成 sql 的 where 条件

QueryWrapper ： 查询条件封装

UpdateWrapper ： Update 条件封装

AbstractLambdaWrapper ： 使用Lambda 语法

LambdaQueryWrapper ：用于Lambda语法使用的查询Wrapper

LambdaUpdateWrapper ： Lambda 更新封装Wrapper

#### 组装查询条件

@Test  
 public void test01(){  
 //查询用户名 包含a，年龄20到30之间，邮箱不为null的用户信息  
 QueryWrapper<User> queryWrapper = new QueryWrapper<>();  
 queryWrapper.like("user\_name","a")  
 .between("age",20,30)  
 .isNotNull("email");  
 List<User> list = userMapper.selectList(queryWrapper);  
 list.forEach(System.out::println);  
 }

#### 组装排序条件

@Test  
 public void test02(){  
 //查询用户信息，按年龄降序排序，若年龄相同，则按id升序排序  
 QueryWrapper<User> queryWrapper = new QueryWrapper<>();  
 queryWrapper.orderByDesc("age")  
 .orderByAsc("uid");  
 List<User> list = userMapper.selectList(queryWrapper);  
 list.forEach(System.out::println);  
 }

#### 组装删除条件

@Test  
 public void test03(){  
 //删除邮箱为null的用户信息  
 QueryWrapper<User> queryWrapper = new QueryWrapper<>();  
 queryWrapper.isNull("email");  
 int result = userMapper.delete(queryWrapper);  
 System.out.println("result:"+result);  
 }

#### 实现修改

@Test  
 public void test04(){  
 //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  
 QueryWrapper<User> queryWrapper = new QueryWrapper<>();  
 queryWrapper.gt("age",20)  
 .like("user\_name","a")  
 .or()  
 .isNull("email");  
 User user = new User(null,"小明",null,"12445@qq.com",null);  
 int result = userMapper.update(user, queryWrapper);  
 System.out.println("result:"+result);  
 }

#### 条件的优先级

@Test  
 public void test05(){  
 //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 user\_name=?, email=? WHERE (age > ? AND user\_name LIKE ? OR email IS NULL)  
 //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 user\_name=?, email=? WHERE (user\_name LIKE ? AND (age > ? OR email IS NULL))  
 //lambda表达式里的条件优先执行  
 QueryWrapper<User> queryWrapper = new QueryWrapper<>();  
 queryWrapper.like("user\_name","a")  
 .and(i->i.gt("age",20).or().isNull("email"));  
 User user = new User(null,"韩立",null,"12090312@qq.com",null);  
 int result = userMapper.update(user, queryWrapper);  
 System.out.println("result:"+result);  
  
 }

#### 组装select子句

查询用户信息的username和age字段

@Test  
 public void test06(){  
 //查询用户信息的username和age字段  
 QueryWrapper<User> queryWrapper = new QueryWrapper<>();  
 queryWrapper.select("user\_name","age");  
 List<Map<String, Object>> list = userMapper.selectMaps(queryWrapper);  
 list.forEach(System.out::println);  
 }

#### 实现子查询

@Test  
 public void test07(){  
 //查询id小于等于3的用户信息  
 QueryWrapper<User> queryWrapper = new QueryWrapper<>();  
 queryWrapper.inSql("uid","select uid from t\_user where uid <= 3");  
 List<User> list = userMapper.selectList(queryWrapper);  
 list.forEach(System.out::println);  
 }

### UpdateWrapper

@Test  
 public void test08(){  
 UpdateWrapper<User> updateWrapper = new UpdateWrapper<>();  
 updateWrapper.like("user\_name","a")  
 .and(i->i.gt("age",20).or().isNull("email"));  
 updateWrapper.set("user\_name","韩老魔").set("email","923849@qq.com");  
 int result = userMapper.update(null, updateWrapper);  
 System.out.println("result:"+result);  
 }

#### 模拟开发中组装条件的情况

在真正开发的过程中，组装条件是常见的功能，而这些条件数据来源于用户输入，是可选的，因

此我们在组装这些条件时，必须先判断用户是否选择了这些条件，若选择则需要组装该条件，若

没有选择则一定不能组装，以免影响SQL执行的结果

@Test  
 public void test09() {  
 String username = "";  
 Integer ageBegin = 20;  
 Integer ageEnd = 30;  
 QueryWrapper<User> queryWrapper = new QueryWrapper<>();  
 if (StringUtils.isNotBlank(username)) {  
 //判断某字符串是否不为空且长度不为0且不由空白符  
 queryWrapper.like("user\_name", username);  
 }  
 if (ageBegin != null) {  
 queryWrapper.ge("age", ageBegin);  
 }  
 if (ageEnd != null) {  
 queryWrapper.le("age", ageEnd);  
 }  
 List<User> list = userMapper.selectList(queryWrapper);  
 list.forEach(System.out::println);  
 }

#### condition组装条件

上面的实现方案没有问题，但是代码比较复杂，我们可以使用带condition参数的重载方法构建查

询条件，简化代码的编写

@Test  
 public void test10(){  
 String username = "";  
 Integer ageBegin = 20;  
 Integer ageEnd = 30;  
 QueryWrapper<User> queryWrapper = new QueryWrapper<>();  
 queryWrapper.like(StringUtils.isNotBlank(username),"user\_name",username)  
 .ge(ageBegin != null,"age", ageBegin)  
 .le(ageEnd!=null,"age",ageEnd);  
 List<User> list = userMapper.selectList(queryWrapper);  
 list.forEach(System.out::println);  
 }

#### LambdaQueryWrapper

@Test  
 public void test11(){  
 String username = "";  
 Integer ageBegin = 20;  
 Integer ageEnd = 30;  
 LambdaQueryWrapper<User> queryWrapper = new LambdaQueryWrapper<>();  
 queryWrapper.like(StringUtils.isNotBlank(username),User::getName,username)  
 .ge(ageBegin != null,User::getAge,ageBegin)  
 .le(ageEnd!=null,User::getAge,ageEnd);  
 List<User> list = userMapper.selectList(queryWrapper);  
 list.forEach(System.out::println);  
 }

**LambdaUpdateWrapper**

@Test  
 public void test12(){  
 LambdaUpdateWrapper<User> updateWrapper = new LambdaUpdateWrapper<>();  
 updateWrapper.like(User::getName,"a")  
 .and(i->i.gt(User::getAge,20).or().isNull(User::getEmail));  
 updateWrapper.set(User::getName,"韩老魔").set(User::getEmail,"923849@qq.com");  
 int result = userMapper.update(null, updateWrapper);  
 System.out.println("result:"+result);  
 }

## 插件

### 分页插件

MyBatis Plus自带分页插件，只要简单的配置即可实现分页功能

#### 添加配置类

@Configuration  
//扫描mapper接口所在的包,将启动类的注解移到此处  
@MapperScan("com.lai.mybatisplus.mapper")  
public class MybatisPlusConfig {  
  
 @Bean  
 public MybatisPlusInterceptor mybatisPlusInterceptor(){  
 MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();  
 interceptor.addInnerInterceptor(new PaginationInnerInterceptor(DbType.MYSQL));  
 return interceptor;  
 }  
}

#### 获取分页数据

@Test  
 public void testPage() {  
 Page page = new Page(2, 3);  
 userMapper.selectPage(page, null);  
 System.out.println(page.getRecords());  
 System.out.println("当前页：" + page.getCurrent());  
 System.out.println("每页显示的条数：" + page.getSize());  
 System.out.println("总记录数：" + page.getTotal());  
 System.out.println("总页数：" + page.getPages());  
 System.out.println("是否有上一页：" + page.hasPrevious());  
 System.out.println("是否有下一页：" + page.hasNext());  
 }

#### 自定义分页

##### UserMapper接口中定义方法

/\*\*  
 \* 根据年龄查询用户列表，分页显示  
 \* @param page page 分页对象,xml中可以从里面进行取值,传递参数 Page 即自动分页,必须放在第一位  
 \* @param age  
 \* @return  
 \*/  
 Page<User> selectPageVo(@Param("page") Page<User> page, @Param("age") Integer age);

#添加日志  
mybatis-plus:  
 configuration:  
 log-impl: org.apache.ibatis.logging.stdout.StdOutImpl  
 #MybatisPlus全局配置  
 global-config:  
 db-config:  
 table-prefix: t\_  
 #设置统一的主键生成策略  
 id-type: auto  
 #设置类型别名   
 type-aliases-package: com.lai.mybatisplus.pojo

##### 测试

@Test  
 public void testPageVo(){  
 Page page = new Page(1, 3);  
 userMapper.selectPageVo(page,20);  
 System.out.println(page.getRecords());  
 System.out.println("当前页：" + page.getCurrent());  
 System.out.println("每页显示的条数：" + page.getSize());  
 System.out.println("总记录数：" + page.getTotal());  
 System.out.println("总页数：" + page.getPages());  
 System.out.println("是否有上一页：" + page.hasPrevious());  
 System.out.println("是否有下一页：" + page.hasNext());  
 }

### 乐观锁

**场景**

一件商品，成本价是80元，售价是100元。老板先是通知小李，说你去把商品价格增加50元。小

李正在玩游戏，耽搁了一个小时。正好一个小时后，老板觉得商品价格增加到150元，价格太

高，可能会影响销量。又通知小王，你把商品价格降低30元。

此时，小李和小王同时操作商品后台系统。小李操作的时候，系统先取出商品价格100元；小王

也在操作，取出的商品价格也是100元。小李将价格加了50元，并将100+50=150元存入了数据

库；小王将商品减了30元，并将100-30=70元存入了数据库。是的，如果没有锁，小李的操作就

完全被小王的覆盖了。

现在商品价格是70元，比成本价低10元。几分钟后，这个商品很快出售了1千多件商品，老板亏1

万多。

**乐观锁与悲观锁**

上面的故事，如果是乐观锁，小王保存价格前，会检查下价格是否被人修改过了。如果被修改过

了，则重新取出的被修改后的价格，150元，这样他会将120元存入数据库。

如果是悲观锁，小李取出数据后，小王只能等小李操作完之后，才能对价格进行操作，也会保证

最终的价格是120元。

#### 模拟修改冲突

##### 数据库中增加商品表

CREATE TABLE t\_product ( id BIGINT(20) NOT NULL COMMENT '主键ID', NAME VARCHAR(30) NULL DEFAULT NULL COMMENT '商品名称', price INT(11) DEFAULT 0 COMMENT '价格', VERSION INT(11) DEFAULT 0 COMMENT '乐观锁版本号', PRIMARY KEY (id) );

##### 插入一条数据

INSERT INTO t\_product (id, NAME, price) VALUES (1, '外星人笔记本', 100);

##### 添加实体类

@Data  
public class Product {  
 private Long id;  
 private String name;  
 private Integer price;  
 private Integer version;  
}

##### 添加Mapper

public interface ProductMapper extends BaseMapper<Product> {  
  
  
}

##### 模拟冲突

@Test  
 public void testProduct01(){  
 //小李查询商品价格  
 Product productLi = productMapper.selectById(1);  
 System.out.println("小李查询的商品价格："+productLi.getPrice());  
 //小韩查询商品价格  
 Product productHan = productMapper.selectById(1);  
 System.out.println("小韩查询的商品价格："+productHan.getPrice());  
 //小李将商品价格+50  
 productLi.setPrice(productLi.getPrice()+50);  
 productMapper.updateById(productLi);  
 //小李将商品价格-30  
 productHan.setPrice(productHan.getPrice()-30);  
 productMapper.updateById(productHan);  
 //老板查询商品价格  
 Product productBoss = productMapper.selectById(1);  
 System.out.println("老板查询的商品价格："+productBoss.getPrice());  
 }

#### 乐观锁实现流程

数据库中添加version字段

取出记录时，获取当前version

SELECT id,`name`,price,`version` FROM product WHERE id=1

更新时，version + 1，如果where语句中的version版本不对，则更新失败

UPDATE product SET price=price+50, `version`=`version` + 1 WHERE id=1 AND `version`=1

#### Mybatis-Plus实现乐观锁

##### 添加版本号字段

@Data  
public class Product {  
 private Long id;  
 private String name;  
 private Integer price;  
 @Version//标识乐观锁版本号字段  
 private Integer version;  
}

##### 添加乐观锁插件配置

@Configuration  
//扫描mapper接口所在的包,将启动类的注解移到此处  
@MapperScan("com.lai.mybatisplus.mapper")  
public class MybatisPlusConfig {  
  
 @Bean  
 public MybatisPlusInterceptor mybatisPlusInterceptor(){  
 MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();  
 //添加分页插件  
 interceptor.addInnerInterceptor(new PaginationInnerInterceptor(DbType.MYSQL));  
 //添加乐观锁插件  
 interceptor.addInnerInterceptor(new OptimisticLockerInnerInterceptor());  
 return interceptor;  
 }  
}

##### 测试修改冲突

小李查询商品信息：

SELECT id,name,price,version FROM t\_product WHERE id=?

小王查询商品信息：

SELECT id,name,price,version FROM t\_product WHERE id=?

小李修改商品价格，自动将version+1

UPDATE t\_product SET name=?, price=?, version=? WHERE id=? AND version=?

Parameters: 外星人笔记本(String), 150(Integer), 1(Integer), 1(Long), 0(Integer)

小王修改商品价格，此时version已更新，条件不成立，修改失败

UPDATE t\_product SET name=?, price=?, version=? WHERE id=? AND version=?

Parameters: 外星人笔记本(String), 70(Integer), 1(Integer), 1(Long), 0(Integer)

最终，小王修改失败，查询价格：150

SELECT id,name,price,version FROM t\_product WHERE id=?

##### 优化流程

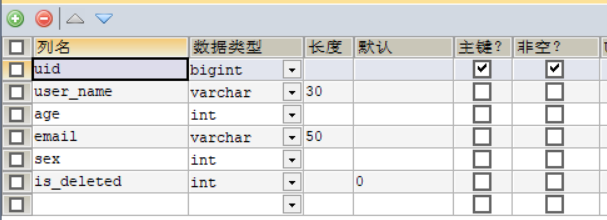
@Test  
 public void testProduct01(){  
 //小李查询商品价格  
 Product productLi = productMapper.selectById(1);  
 System.out.println("小李查询的商品价格："+productLi.getPrice());  
 //小韩查询商品价格  
 Product productHan = productMapper.selectById(1);  
 System.out.println("小韩查询的商品价格："+productHan.getPrice());  
 //小李将商品价格+50  
 productLi.setPrice(productLi.getPrice()+50);  
 productMapper.updateById(productLi);  
 //小韩将商品价格-30  
 productHan.setPrice(productHan.getPrice()-30);  
 int result = productMapper.updateById(productHan);  
 if (result==0){  
 Product productNew = productMapper.selectById(1);  
 productNew.setPrice(productNew.getPrice()-30);  
 productMapper.updateById(productNew);  
 }  
 //老板查询商品价格  
 Product productBoss = productMapper.selectById(1);  
 System.out.println("老板查询的商品价格："+productBoss.getPrice());  
 }

## 通用枚举

表中的有些字段值是固定的，例如性别（男或女），此时我们可以使用MyBatis-Plus的通用枚举

来实现

### 数据库表添加字段sex



### 创建通用枚举类型

@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;  
 }  
}

public class User {  
  
 //将属性所对应的字段标识为主键,并设置主键生成策略（主键自增）  
 @TableId("uid")  
 private Long id;  
 //设置属性所对应的字段名  
 @TableField("user\_name")  
 private String name;  
  
 private Integer age;  
  
 private SexEnum sex;  
  
 private String email;  
  
// @TableLogic  
 private Integer isDeleted;  
}

### 配置扫描通用枚举

mybatis-plus:  
 configuration:  
 log-impl: org.apache.ibatis.logging.stdout.StdOutImpl  
 #MybatisPlus全局配置  
 global-config:  
 db-config:  
 table-prefix: t\_  
 #设置统一的主键生成策略  
 id-type: auto  
 #设置类型别名  
 type-aliases-package: com.lai.mybatisplus.pojo  
 # 配置扫描通用枚举  
 type-enums-package: com.lai.mybatisplus.enums

### 测试

@Test  
 public void test(){  
 User user = new User();  
 user.setName("admin");  
 user.setAge(30);  
 user.setSex(SexEnum.MALE);  
 int result = userMapper.insert(user);  
 System.out.println("result:"+result);  
 }

## 代码生成器

### 引入依赖

<dependency>  
 <groupId>com.baomidou</groupId>  
 <artifactId>mybatis-plus-generator</artifactId>  
 <version>3.5.1</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.freemarker</groupId>  
 <artifactId>freemarker</artifactId>  
 <version>2.3.31</version>  
 </dependency>

### 快速生成

public class FastAutoGeneratorTest {  
  
 public static void main(String[] args) {  
 FastAutoGenerator.create("jdbc:mysql://127.0.0.1:3306/mybatis\_plus?serverTimezone=GMT%2B8&characterEncoding=utf-8&userSSL=false", "root", "password")  
 .globalConfig(builder -> {  
 builder.author("lai") // 设置作者  
// .enableSwagger() // 开启 swagger 模式  
 .fileOverride() // 覆盖已生成文件  
 .outputDir("D://mybatis\_plus"); // 指定输出目录  
 }).packageConfig(builder -> {  
 builder.parent("com.lai")// 设置父包名  
 .moduleName("mybatisplus")// 设置父包模块名  
 .pathInfo(Collections.singletonMap(OutputFile.mapperXml, "D://mybatis\_plus"));// 设置mapperXml生成路径  
 }).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();  
 }  
}



## 多数据源

适用于多种场景：纯粹多库、 读写分离、 一主多从、 混合模式等

目前我们就来模拟一个纯粹多库的一个场景，其他场景类似

场景说明：

我们创建两个库，分别为：mybatis\_plus（以前的库不动）与mybatis\_plus\_1（新建），将

mybatis\_plus库的product表移动到mybatis\_plus\_1库，这样每个库一张表，通过一个测试用例

分别获取用户数据与商品数据，如果获取到说明多库模拟成功

### 创建数据库和表

CREATE DATABASE `mybatis\_plus\_1` /\*!40100 DEFAULT CHARACTER SET utf8mb4 \*/; use `mybatis\_plus\_1`; CREATE TABLE product ( id BIGINT(20) NOT NULL COMMENT '主键ID', name VARCHAR(30) NULL DEFAULT NULL COMMENT '商品名称', price INT(11) DEFAULT 0 COMMENT '价格', version INT(11) DEFAULT 0 COMMENT '乐观锁版本号', PRIMARY KEY (id) );

### 添加一条数据

INSERT INTO product (id, NAME, price) VALUES (1, '外星人笔记本', 100);

### 删除mybatis\_plus库product表

use mybatis\_plus;

DROP TABLE IF EXISTS product;

### 引入依赖

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

### 配置多数据源

spring:  
 # 配置数据源信息  
 datasource:  
 dynamic:  
 # 设置默认的数据源或者数据源组,默认值即为master  
 primary: master  
 # 严格匹配数据源,默认false.true未匹配到指定数据源时抛异常,false使用默认数据源  
 strict: false  
 datasource:  
 master:  
 url: jdbc:mysql://localhost:3306/mybatis\_plus?serverTimezone=GMT%2B8&characterEncoding=utf-8&useSSL=false  
 driver-class-name: com.mysql.cj.jdbc.Driver  
 username: root  
 password: password  
 slave\_1:  
 url: jdbc:mysql://localhost:3306/mybatis\_plus\_1?serverTimezone=GMT%2B8&characterEncoding=utf-8&useSSL=false  
 driver-class-name: com.mysql.cj.jdbc.Driver  
 username: root  
 password: password

### 实体类

@Data  
@TableName("t\_user")  
public class User {  
  
 @TableId  
 private Integer uid;  
  
 private String userName;  
  
 private Integer age;  
  
 private String email;  
  
 private Integer sex;  
  
 private Integer isDeleted;  
}

@Data  
public class Product {  
  
 private Integer id;  
  
 private String name;  
  
 private Integer price;  
  
 private Integer version;  
}

### Mapper接口

@Repository//防止@Autowired注入时报错  
public interface UserMapper extends BaseMapper<User> {  
  
}

@Repository//防止@Autowired注入时报错  
public interface ProductMapper extends BaseMapper<Product> {  
  
}

扫描Mapper所在包

@SpringBootApplication  
@MapperScan("com.lai.mybatisplus.mapper")  
public class MybatisPlusDatasourceApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.run(MybatisPlusDatasourceApplication.class, args);  
 }  
  
}

### Service和Impl

public interface UserService extends IService<User> {  
}

public interface ProductService extends IService<Product> {  
  
}

@Service  
@DS("master")//指定操作的数据源  
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements UserService {  
}

@Service  
@DS("slave\_1")//指定操作的数据源  
public class ProductServiceImpl extends ServiceImpl<ProductMapper, Product> implements ProductService {  
}

### 测试

@Autowired  
 private UserService userService;  
  
 @Autowired  
 private ProductService productService;  
  
 @Test  
 public void test(){  
 System.out.println(userService.getById(1));  
 System.out.println(productService.getById(1));  
 }

结果：

1、都能顺利获取对象，则测试成功

2、如果我们实现读写分离，将写操作方法加上主库数据源，读操作方法加上从库数据源，自动切

换，是不是就能实现读写分离？

## MyBatisX插件

MyBatis-Plus为我们提供了强大的mapper和service模板，能够大大的提高开发效率

但是在真正开发过程中，MyBatis-Plus并不能为我们解决所有问题，例如一些复杂的SQL，多表

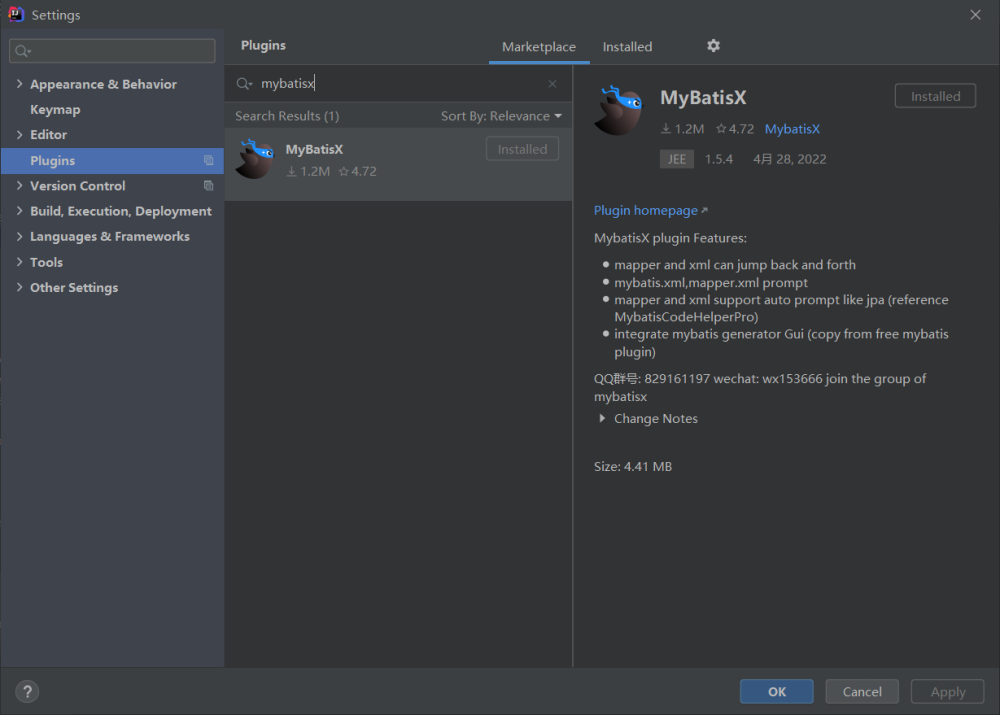
联查，我们就需要自己去编写代码和SQL语句，我们该如何快速的解决这个问题呢，这个时候可

以使用MyBatisX插件

MyBatisX一款基于 IDEA 的快速开发插件，为效率而生。

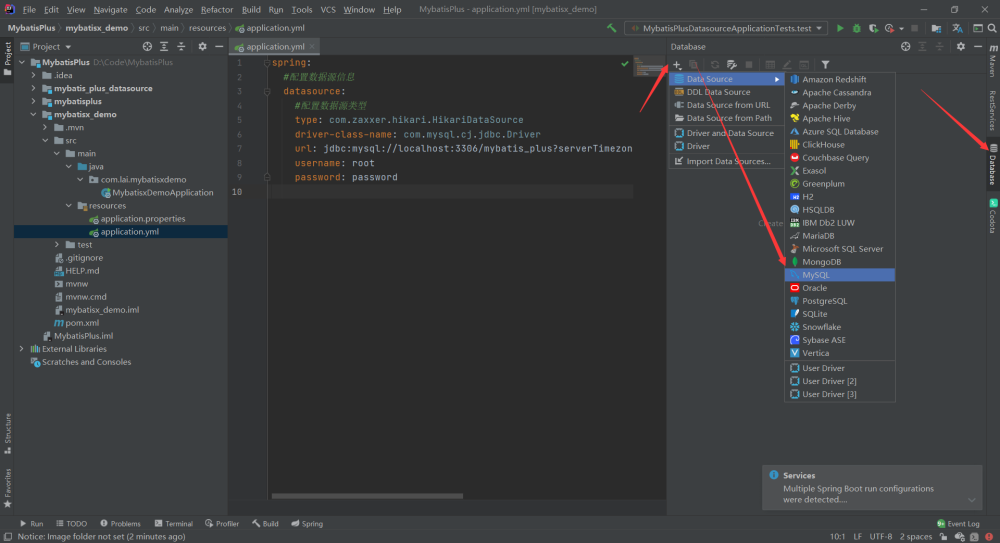
MyBatisX插件用法：https://baomidou.com/pages/ba5b24/

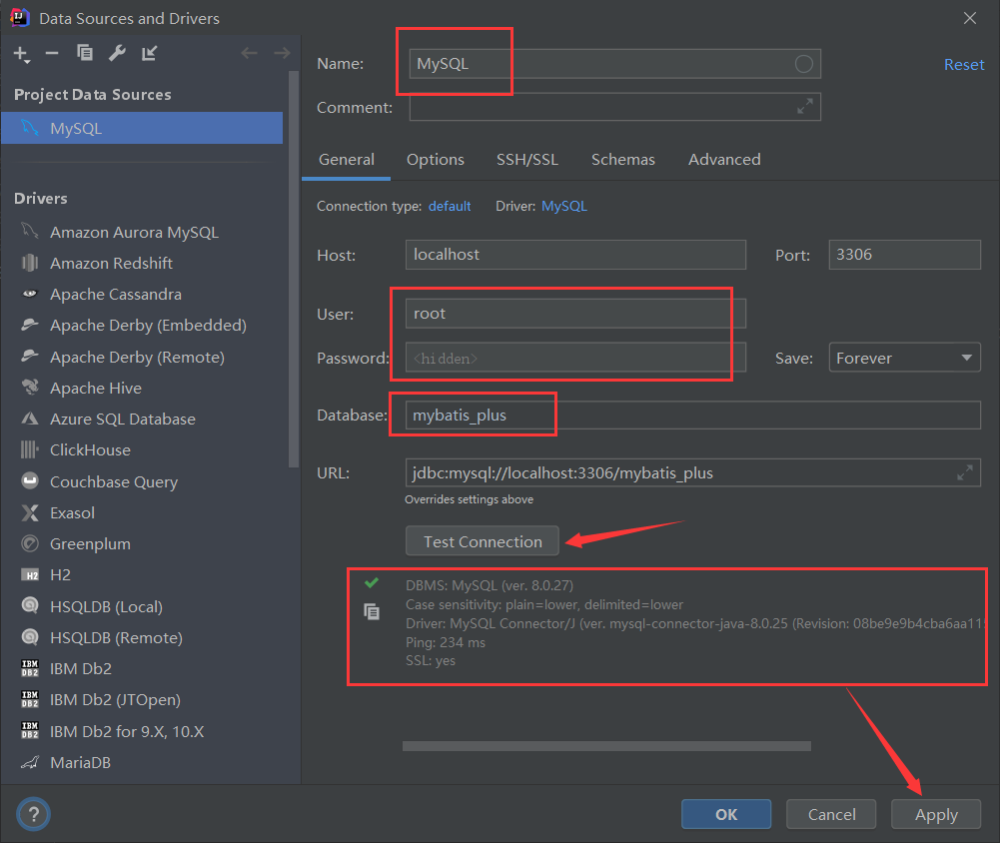
### 安装插件



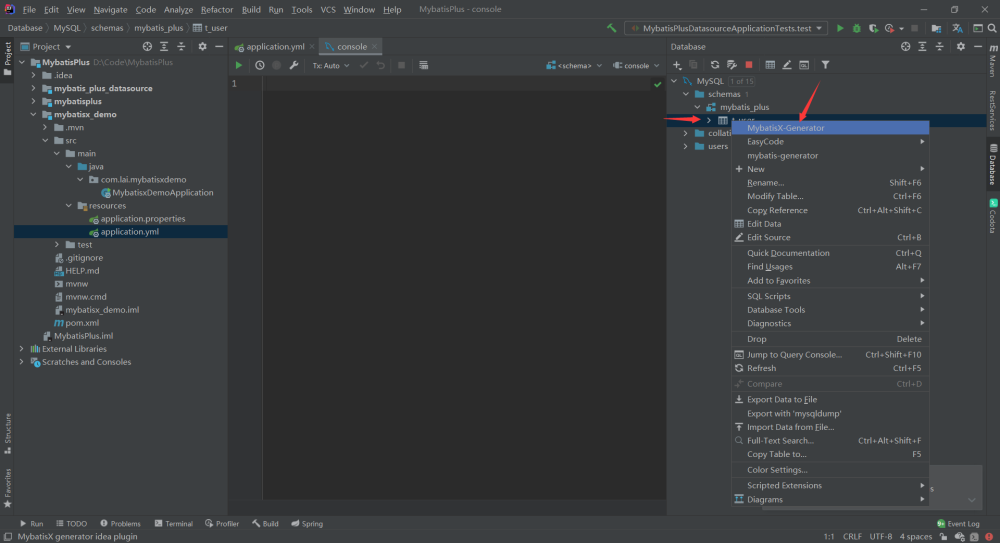
### 代码快速生成

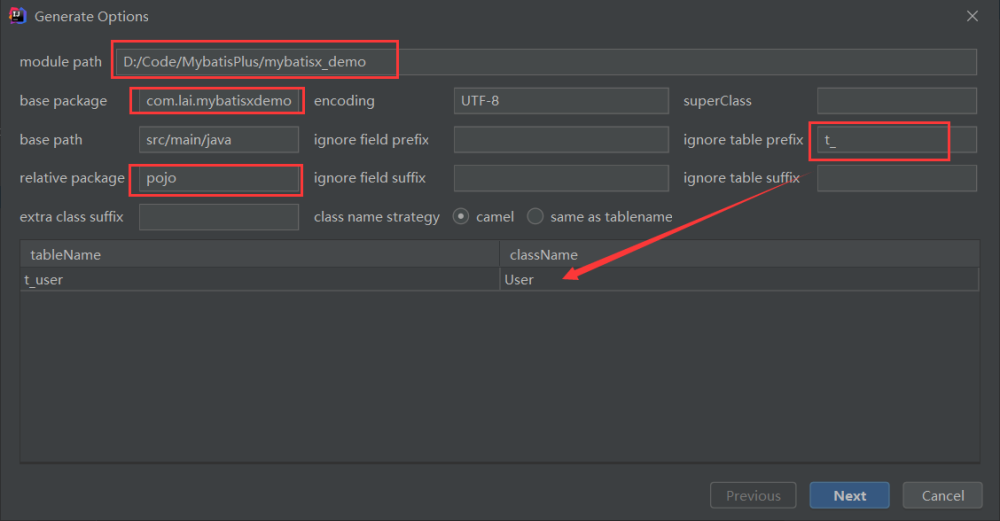
#### 测试连接

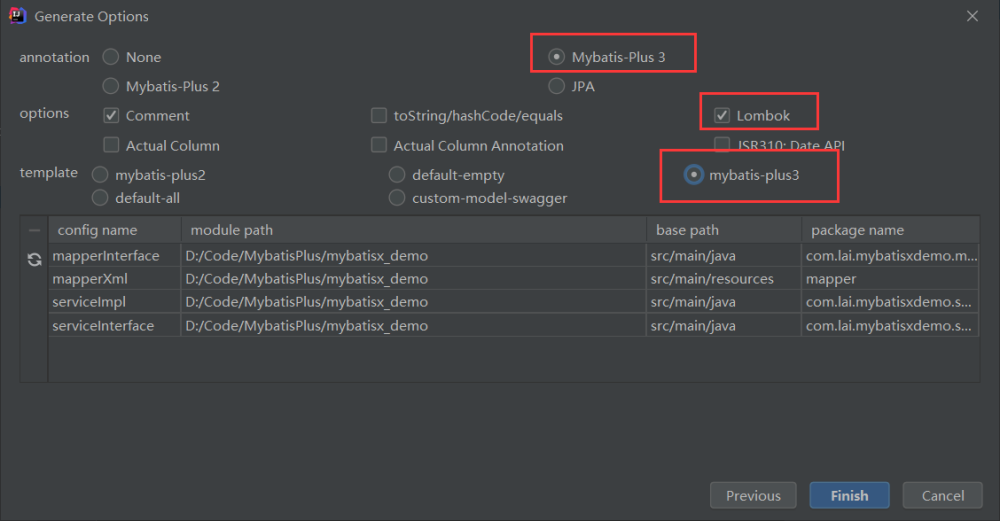




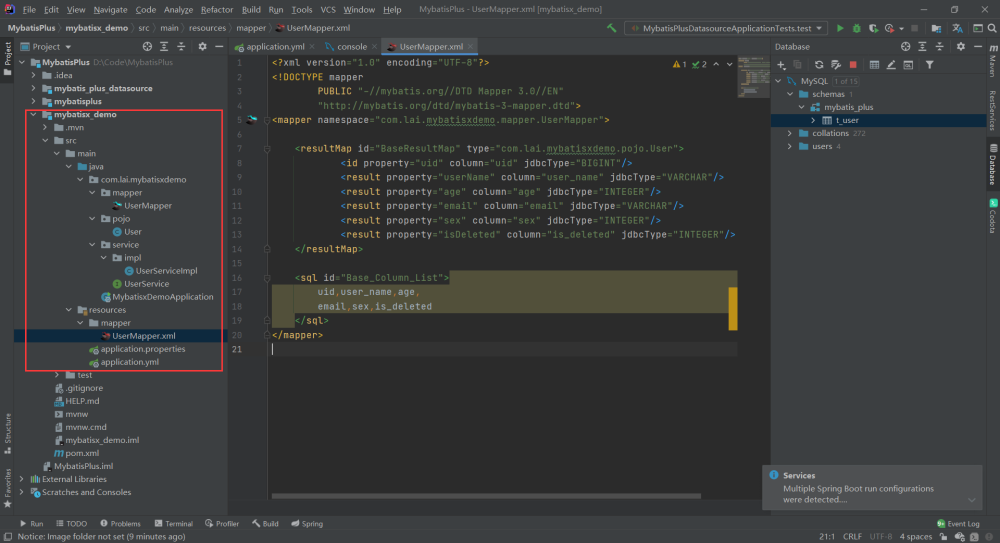
右击表选择



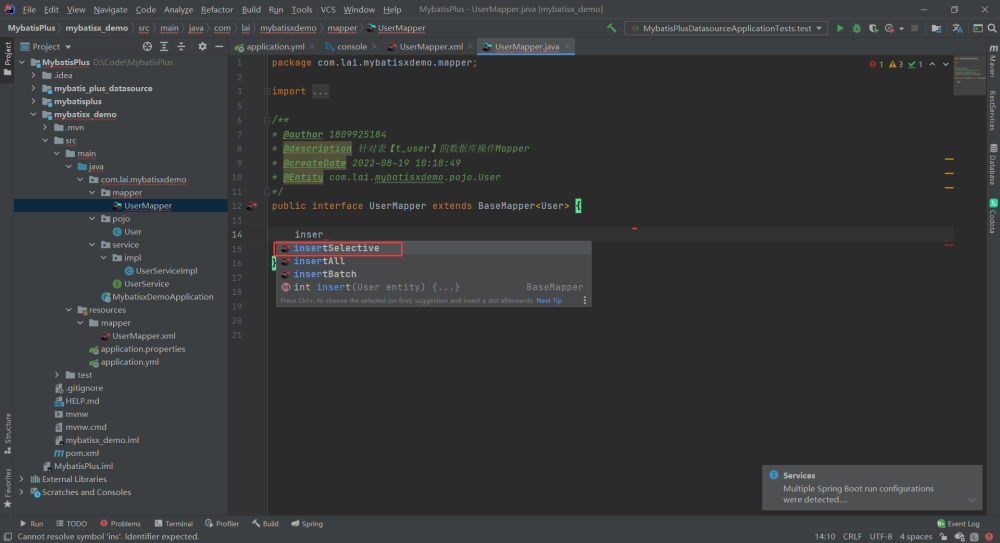




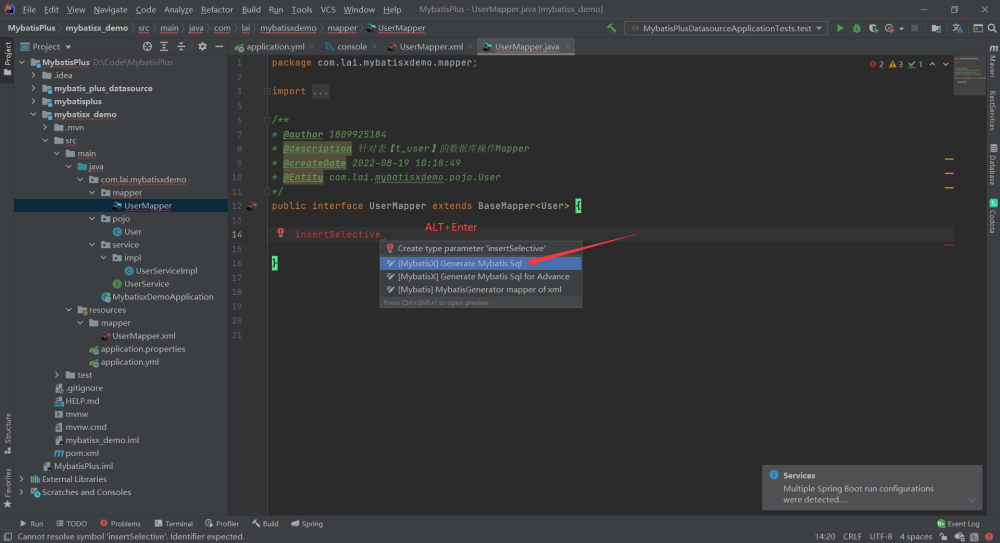
代码已经生成

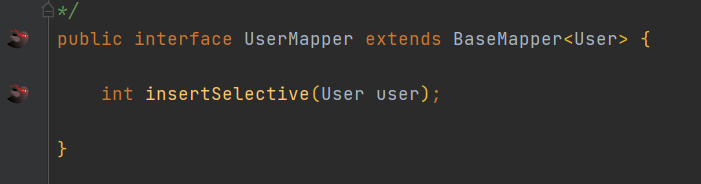


### 快速生成CRUD

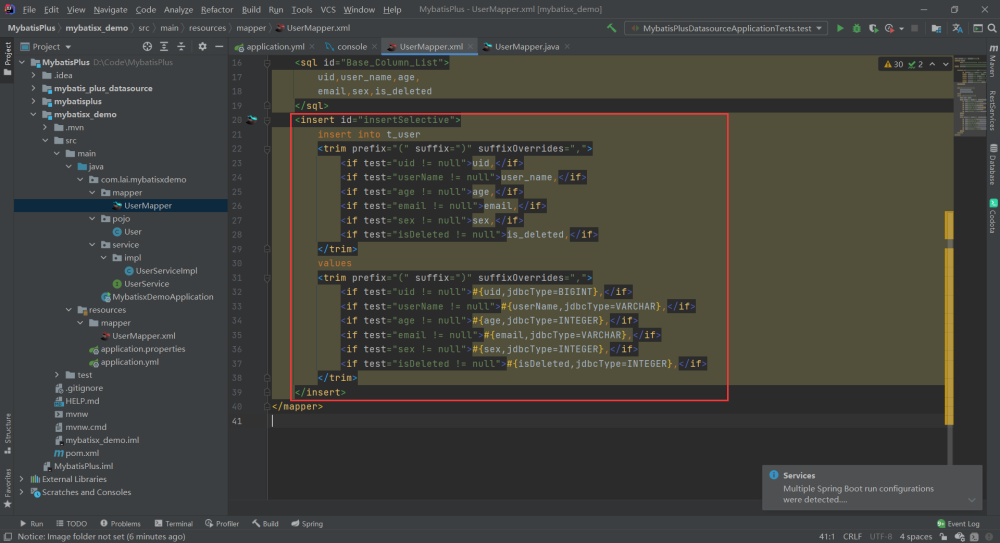


alt+enter





自动帮我们生成sql语句



同样方法生成多个语句

public interface UserMapper extends BaseMapper<User> {  
  
 int insertSelective(User user);  
  
 int delByUidAndUserName(@Param("uid") Long uid, @Param("userName") String userName);  
  
 int updateAgeAndEmailByUid(@Param("age") Integer age, @Param("email") String email, @Param("uid") Long uid);  
  
 List<User> selectAgeAndSexByAgeBetween(@Param("beginAge") Integer beginAge, @Param("endAge") Integer endAge);  
  
 List<User> selectAllByAgeOrderByAgeDesc(@Param("age") Integer age);  
  
}

<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE mapper  
 PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper namespace="com.lai.mybatisxdemo.mapper.UserMapper">  
  
 <resultMap id="BaseResultMap" type="com.lai.mybatisxdemo.pojo.User">  
 <id property="uid" column="uid" jdbcType="BIGINT"/>  
 <result property="userName" column="user\_name" jdbcType="VARCHAR"/>  
 <result property="age" column="age" jdbcType="INTEGER"/>  
 <result property="email" column="email" jdbcType="VARCHAR"/>  
 <result property="sex" column="sex" jdbcType="INTEGER"/>  
 <result property="isDeleted" column="is\_deleted" jdbcType="INTEGER"/>  
 </resultMap>  
  
 <sql id="Base\_Column\_List">  
 uid,user\_name,age,  
 email,sex,is\_deleted  
 </sql>  
 <insert id="insertSelective">  
 insert into t\_user  
 <trim prefix="(" suffix=")" suffixOverrides=",">  
 <if test="uid != null">uid,</if>  
 <if test="userName != null">user\_name,</if>  
 <if test="age != null">age,</if>  
 <if test="email != null">email,</if>  
 <if test="sex != null">sex,</if>  
 <if test="isDeleted != null">is\_deleted,</if>  
 </trim>  
 values  
 <trim prefix="(" suffix=")" suffixOverrides=",">  
 <if test="uid != null">#{uid,jdbcType=BIGINT},</if>  
 <if test="userName != null">#{userName,jdbcType=VARCHAR},</if>  
 <if test="age != null">#{age,jdbcType=INTEGER},</if>  
 <if test="email != null">#{email,jdbcType=VARCHAR},</if>  
 <if test="sex != null">#{sex,jdbcType=INTEGER},</if>  
 <if test="isDeleted != null">#{isDeleted,jdbcType=INTEGER},</if>  
 </trim>  
 </insert>  
 <delete id="delByUidAndUserName">  
 delete  
 from t\_user  
 where uid = #{uid,jdbcType=NUMERIC}  
 AND user\_name = #{userName,jdbcType=VARCHAR}  
 </delete>  
 <update id="updateAgeAndEmailByUid">  
 update t\_user  
 set age = #{age,jdbcType=NUMERIC},  
 email = #{email,jdbcType=VARCHAR}  
 where uid = #{uid,jdbcType=NUMERIC}  
 </update>  
 <select id="selectAgeAndSexByAgeBetween" resultMap="BaseResultMap">  
 select age, sex  
 from t\_user  
 where age between #{beginAge,jdbcType=INTEGER} and #{endAge,jdbcType=INTEGER}  
 </select>  
 <select id="selectAllByAgeOrderByAgeDesc" resultMap="BaseResultMap">  
 select  
 <include refid="Base\_Column\_List"/>  
 from t\_user  
 where  
 age = #{age,jdbcType=NUMERIC}  
 order by age desc  
 </select>  
  
  
</mapper>