MyBatis Plus 3.x 版本

MyBatis Plus 概述

- MyBatis Plus 就是用来简化开发的，所有的CRUD功能全部都由它来帮我们实现
- 以前我们用 Mybatis 来写代码的时候，首要要写实体类，然后创建 mapper 接口，然后再创建 Mapper.xml 文件来写 xml 文件，然而当我们使用了 MyBatis Plus 后，这些操作都会由 Mybatis Plus 来帮我们实现。

MyBatis Plus 官网：https://mp.baomidou.com/

MyBatis Plus 特性

这些都是官网的描述：可以看出 MP 的功能很强大。

- **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

快速入门

1. 创建数据库 `mybatis_plus`
2. 创建 user 表

```
DROP TABLE IF EXISTS user;

CREATE TABLE user
(
    id BIGINT(20) NOT NULL COMMENT '主键ID',
    name VARCHAR(30) NULL DEFAULT NULL COMMENT '姓名',
    age INT(11) NULL DEFAULT NULL COMMENT '年龄',
    email VARCHAR(50) NULL DEFAULT NULL COMMENT '邮箱',
    PRIMARY KEY (id)
```

```
);
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');
```

3. 创建 SpringBoot 项目
4. 导入依赖

```
<!-- 数据库驱动 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>

<!-- lombok -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>

<!-- mybatis plus -->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.0.5.tmp</version>
</dependency>
```

在使用 mybatis-plus 时可以节省大量的代码，尽量不要同时导入 mybatis 和 mybatis-plus ，避免版本差异出现问题。

5. 修改配置文件 applicaton.properties，配置数据库连接信息

```
# mysql 5 驱动不同 com.mysql.jdbc.Driver
# mysql 8 驱动不同 com.mysql.cj.jdbc.Driver 需要增加时区的设置
serverTimezone=GMT%2B8
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.url=jdbc:mysql://localhost:3306/mybatis_plus?
useSSL=false&useUnicode=true&characterEncoding=utf-8&serverTimezone=GMT%2B8
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

6. 创建实体类【因为这里使用 lombok 所以简单的添加几个注解就即可了】

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

7. 创建 mapper 接口

```
package com.javaboy.mapper;

import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.javaboy.pojo.User;
import org.springframework.stereotype.Repository;

/**
 * @Author: 红颜祸水nvn <bai211425401@126.com>
 * @Description: CSDN <https://blog.csdn.net/qq_43647359>
 */
// 在对应的 Mapper 上面继承基本的类 BaseMapper
@Repository
public interface UserMapper extends BaseMapper<User> {
    // 所有的 CRUD 方法 BaseMapper 都已经帮我们编写完成了
}
```

8. 在主启动类上添加注解

```
@SpringBootApplication
@MapperScan("com.javaboy.mapper") // 去扫描 mapper 包下的所有接口
public class MybatisPlusApplication {
    public static void main(String[] args) {
        SpringApplication.run(MybatisPlusApplication.class, args);
    }
}
```

9. 开始测试

```
package com.javaboy;

import com.javaboy.mapper.UserMapper;
import com.javaboy.pojo.User;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import javax.annotation.Resource;
import java.util.List;

@SpringBootTest
class MybatisPlusApplicationTests {

    // 继承了 BaseMapper 所有的方法都由 BaseMapper 来帮我们实现
    // 我们也可以添加自定义的扩展方法
    @Autowired
    private UserMapper userMapper;

    @Test
    public void test() {
        // selectList 参数是一个 Wrapper，它是一个条件构造器，传入 null 就代表查询全部
        // selectList(null); 就表示查询该表中所有数据
        List<User> users = userMapper.selectList(null);
        users.forEach(System.out::println);
    }
```

```
}
```

10. 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)
```

配置日志

通过配置，就可以看到sql的日志打印，这样在编码阶段出现问题有助于帮着我们快速学着问题，不适用于生产阶段。

```
# 配置日志
mybatis-plus.configuration.log-impl=org.apache.ibatis.logging.stdout.StdOutImpl
```

```
JBC Connection [HikariProxyConnection@899736725 wrapping com.mysql.cj.jdbc
=>  Preparing: SELECT id,name,age,email FROM user
=> Parameters:
==    Columns: id, name, age, email
==        Row: 1, Jone, 18, test1@baomidou.com
==        Row: 2, Jack, 20, test2@baomidou.com
==        Row: 3, Tom, 28, test3@baomidou.com
==        Row: 4, Sandy, 21, test4@baomidou.com
==        Row: 5, Billie, 24, test5@baomidou.com
==      Total: 5
```

可以明确的看到打印的 sql 语句
这样遇到问题非常有助于我们分析错误!

CRUD 扩展

插入操作

```
@Test
public void testInsert() {
    User user = new User();
    user.setName("小白");
    user.setAge(15);
    user.setEmail("xb@126.com");

    /**
     * 在添加的时候我们需要注意，如果我们将主键id设置为 Integer 类型
     * 那么插入就会出错，因为随机生成的主键 ID 长度超过了 Integer 所限制的范围
     */
    // 帮我们自动生成主键 id
    int result = userMapper.insert(user);
    // 受影响的行数
    System.out.println(result);
```

```
        // 主键 id 会自动回填
        System.out.println(user.getId());
    }
```

```
Creating a new SqlSession
SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@4d266391] was not registered for synchr
JDBC Connection [HikariProxyConnection@1402599109 wrapping com.mysql.cj.jdbc.ConnectionImpl@4fe64d23] wi
==>  Preparing: INSERT INTO user ( id, name, age, email ) VALUES ( ?, ?, ?, ? )
==> Parameters: 1258372412710449154(Long), 小白(String), 15(Integer), xb@126.com(String)
<==    Updates: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@4d266391]
1
1258372412710449154
```

受影响的行数

主键

> 数据库插入的 id 的默认值为：全局的唯一 id

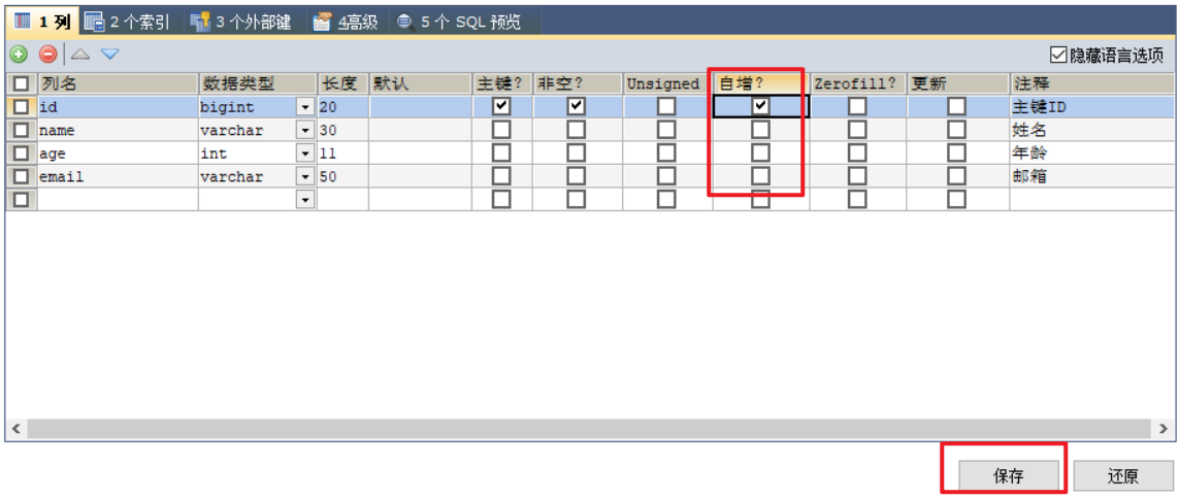
主键生成策略

> 默认的主键生成策略是 ID_WORKER

MP 中所有的主键策略：

```
public enum IdType {
    AUTO(0),     // 数据库id自增
    NONE(1),     // 未设置主键
    INPUT(2),    // 手动输入
    ID_WORKER(3),    // 默认的全局唯一 id
    UUID(4),     // 全局唯一 id UUID
    ID_WORKER_STR(5);    // ID_WORKER 字符串表示
}
```

1. 一般数据库的主键都是自增，现在我们将数据库的主键设置为自增列：



2. 然后在实体类上添加注解 `TableId(type = IdType.AUTO)`，修改主键策略为自增

3. 再次测试

id	name	age	email
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
1258372412710449154	小白	15	xb@126.com
1258375518919606273	小白	15	xb@126.com
1258375518919606274	小白	15	xb@126.com
1258375518919606275	洗礼	22	xx@126.com

修改为自增列后，发现正确，结尾是 3 4 5

更新操作

```
@Test
public void testUpdate() {
    User user = new User();
    // 通过添加自动拼接动态 SQL
    user.setId(1L);
    user.setName("张三");
    user.setAge(20);

    // updateById 接受对象类型
    int result = userMapper.updateById(user);
    System.out.println(result);
}
```

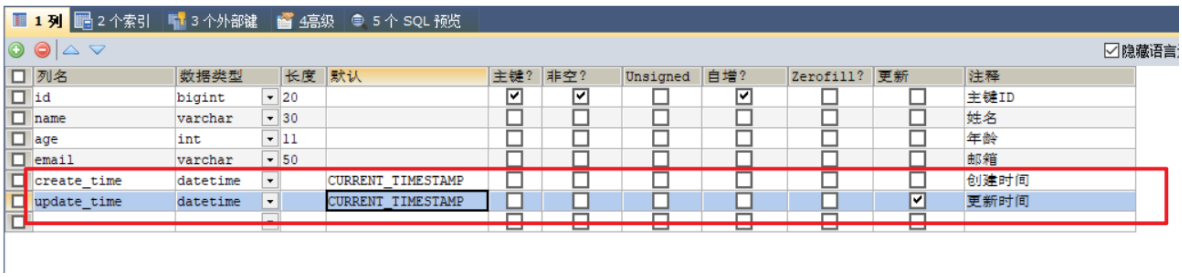
```
Creating a new SqlSession
SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@60d40ff4] was not registered fo
JDBC Connection [HikariProxyConnection@1340493091 wrapping com.mysql.cj.jdbc.ConnectionImpl@5843
==>  Preparing: UPDATE user SET name=?, age=? WHERE id=?
==> Parameters: 张三(String), 20(Integer), 1(Long)
<==    Updates: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@60d40
1
```

自动填充

一般在工作中，我们的数据库表中都会添加两个字段，create_time，update_time 分别代表创建时间，修改时间，而且阿里巴巴开发手册中也说到：所有的数据库表：gmt_create、gmt_modified 几乎所有的表都要配置上，而且需要自动化！

> 方式一：数据库级别修改

1. 在表中新增字段 create_time，update_time



2. 实体类同步字段

```
private Date createTime;
private Date updateTime;
```

3. 再次测试更新，并查看结果

id	name	age	email	create_time	update_time
1	张三666	20	test1@baomidou.com	2020-05-07 20:49:36	2020-05-07 20:52:13
2	Jack	20	test2@baomidou.com	2020-05-07 20:49:36	2020-05-07 20:49:36
3	Tom	28	test3@baomidou.com	2020-05-07 20:49:36	2020-05-07 20:49:36
4	Sandy	21	test4@baomidou.com	2020-05-07 20:49:36	2020-05-07 20:49:36
5	Billie	24	test5@baomidou.com	2020-05-07 20:49:36	2020-05-07 20:49:36
1258372412710449154	小白	15	xb@126.com	2020-05-07 20:49:36	2020-05-07 20:49:36
1258375518919606273	小白	15	xb@126.com	2020-05-07 20:49:36	2020-05-07 20:49:36
1258375518919606274	小白	15	xb@126.com	2020-05-07 20:49:36	2020-05-07 20:49:36
1258375518919606275	洗礼	22	xx@126.com	2020-05-07 20:49:36	2020-05-07 20:49:36

很明显的看到更新时间改变了，但我们在修改时并没有修改这个字段

> 方法二：代码级别修改

1. 删除数据库的默认值，更新操作

列名	数据类型	长度	默认	主键?	非空?	Unsigned	自增?	Zerofill?	更新	注释
id	bigint	20		☑	☑	☐	☑	☐	☐	主键ID
name	varchar	30		☐	☐	☐	☐	☐	☐	姓名
age	int	11		☐	☐	☐	☐	☐	☐	年龄
email	varchar	50		☐	☐	☐	☐	☐	☐	邮箱
create_time	datetime			☐	☐	☐	☐	☐	☐	创建时间
update_time	datetime			☐	☐	☐	☐	☐	☐	更新时间
				☐	☐	☐	☐	☐	☐	

2. 实体类字段属性上需要添加注解：`@TableFieId`

```
@TableField(fill = FieldFill.INSERT)
private Date createTime;

@TableField(fill = FieldFill.INSERT_UPDATE)
private Date updateTime;
```

3. 自定义类实现 `MyMetaObjectHandler`

```
package com.javaboy.handler;

import com.baomidou.mybatisplus.core.handlers.MetaObjectHandler;
import lombok.extern.slf4j.Slf4j;
import org.apache.ibatis.reflection.MetaObject;
import org.springframework.stereotype.Component;

import java.util.Date;

/**
 * @Author: 红颜祸水nvn <bai211425401@126.com>
 * @Description: CSDN <https://blog.csdn.net/qq_43647359>
 */
@Slf4j
@Component  // 将处理器添加到 IOC 容器中去
public class MyMetaObjectHandler implements MetaObjectHandler {
    @Override
    public void insertFill(MetaObject metaObject) {
        // 插入时候的填充策略
        log.info("start insert fill...");
        // createTime 和 updateTime 是实体类中的属性
        this.setFieldValByName("createTime", new Date(), metaObject);
```

```
        this.setFieldValByName("updateTime", new Date(), metaObject);
    }

    @Override
    public void updateFill(MetaObject metaObject) {
        // 更新时后的填充策略
        log.info("start update fill.....");
        this.setFieldValByName("updateTime", new Date(), metaObject);
    }
}
```

4. 测试插入、观察时间时候自动填充
5. 测试更新、观察时间是否有变化

乐观锁插件

什么是乐观锁?

> 乐观锁：它总是很乐观，认为干什么都不会出现问题，所以无论对数据做什么操作都不会上锁。
>
> 悲观锁：他总是很悲观，认为干什么都会出现问题，所以无论对数据做什么操作都会上锁。

乐观锁实现方式：

- 取出记录时，获取当前版本号 version
- 更新时，带上版本号 version
- 执行更新操作时， set version = newVersion where version = oldVersion
- 如果 version 不对，就会更新失败

```
乐观锁：1.先查询，获得版本号 version = 1
-- 线程A
update user set name = "zs",version = version + 1
where id = 2 and version = 1

-- 线程B 抢先完成，这个时候 version = 2，会导致 A 修改失败
update user set name = "ls",version = version + 1
where id =2 and version = 2
```

> MP 的乐观锁实现步骤

1. 给数据库中增加 version 字段

id	name	age	email	version	create_time	update_ti
1	张三666	20	test1@baomidou.com	1	2020-05-07 20:49:36	2020-05-07
2	Jack	20	test2@baomidou.com	1	2020-05-07 20:49:36	2020-05-07
3	Tom	28	test3@baomidou.com	1	2020-05-07 20:49:36	2020-05-07
4	Sandy	21	test4@baomidou.com	1	2020-05-07 20:49:36	2020-05-07
5	Billie	24	test5@baomidou.com	1	2020-05-07 20:49:36	2020-05-07
1258372412710449154	小白	15	xb@126.com	1	2020-05-07 20:49:36	2020-05-07
1258375518919606273	小白	15	xb@126.com	1	2020-05-07 20:49:36	2020-05-07
1258375518919606274	小白	15	xb@126.com	1	2020-05-07 20:49:36	2020-05-07
1258375518919606275	洗礼	22	xx@126.com	1	2020-05-07 20:49:36	2020-05-07
1258375518919606276	里斯222	20	xx@126.com	1	2020-05-07 21:02:44	2020-05-07

2. 实体类添加对应的属性【此注解用来标识乐观锁对应的属性】

```
@Version // 乐观锁 Version 注解
private Integer version;
```

3. 注册乐观锁组件

```
import com.baomidou.mybatisplus.extension.plugins.OptimisticLockerInterceptor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.transaction.annotation.EnableTransactionManagement;

/**
 * @Author: 红颜祸水nvn <bai211425401@126.com>
 * @Description: CSDN <https://blog.csdn.net/qq_43647359>
 */
@EnableTransactionManagement
@Configuration // 配置类注解
public class MyBatisPlusConfig {

    // 注册乐观锁插件
    @Bean
    public OptimisticLockerInterceptor optimisticLockerInterceptor() {
        return new OptimisticLockerInterceptor();
    }
}
```

4. 测试

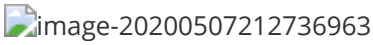
```
// 测试乐观锁成功
@Test
public void testOptimisticLocker() {
    // 1. 查询用户信息
    User user = userMapper.selectById(1L);
    // 2. 修改用户信息
    user.setName("libai");
    user.setEmail("libai@qq.com");
    // 3. 执行更新操作
    userMapper.updateById(user);
}
```

```
:1:21:43.307  INFO 15376 --- [           main] com.javaboy.handler.MyMetaObjectHandler  : start update fill.....
:ion [HikariProxyConnection@1888420238 wrapping com.mysql.cj.jdbc.ConnectionImpl@50f097b5] will not be managed by Spring
.ng  UPDATE user SET name=?, age=?, email=?, version=?, create_time=?, update_time=? WHERE id=? AND version=?
:rs: libai(String), 20(Integer), libai@qq.com(String), 2(Integer), 2020-05-07 20:49:36.0(Timestamp), 2020-05-07 21:21:43
:es: 1
 transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@4779aae6]
```

```
//测试乐观锁失败
@Test
public void testOptimisticLocker2() {
    // 线程 1
    User user = userMapper.selectById(1L);
    user.setName("libai666");
    user.setEmail("libai666@126.com");

    // 模拟另外一个线程执行了插队操作
    User user2 = userMapper.selectById(1L);
    user2.setName("libai777");
    user2.setEmail("libai666@126.com");
    userMapper.updateById(user2);
```

```
    // 如果没有乐观锁就会覆盖插队线程的值
    userMapper.updateById(user);
}
```



查询操作

```
// 测试单量查询
@Test
public void testSelectById() {
    User user = userMapper.selectById(1L);
    System.out.println(user);
}

// 测试批量查询
@Test
public void testSelectBatchIds() {
    List<User> users = userMapper.selectBatchIds(Arrays.asList(1L, 2L, 3L));
    users.forEach(System.out::println);
}

// 按条件查询使用 Map 操作
@Test
public void testSelectByMap() {
    Map<String, Object> map = new HashMap<>();
    // 自定义查询条件
    map.put("name", "libai777");
    map.put("age", 20);

    List<User> users = userMapper.selectByMap(map);
    users.forEach(System.out::println);
}
```

分页插件

如何使用分页插件：

1. 配置分页组件

```
// 分页插件
@Bean
public PaginationInterceptor paginationInterceptor() {
    PaginationInterceptor paginationInterceptor = new PaginationInterceptor();
    return paginationInterceptor;
}
```

2. 直接使用 Page 对象即可

```
// 测试分页查询
@Test
public void testSelectPage() {
    /**
     * current:当前页
```

```
     * size：页大小
     * 使用了分页插件后，所有的分页操作也变得很简单
     */
    Page<User> page = new Page<>(1,3);
    userMapper.selectPage(page,null);

    page.getRecords().forEach(System.out::println);
    System.out.println("***分页查询总数量 total：" + page.getTotal());
}
```

```
==>  Preparing: SELECT id,name,age,email,version,create_time,update_time FROM user LIMIT 0,3
==> Parameters:
<==    Columns: id, name, age, email, version, create_time, update_time
<==        Row: 1, libai777, 20, libai666@126.com, 3, 2020-05-07 20:49:36, 2020-05-07 21:26:42
<==        Row: 2, Jack, 20, test2@baomidou.com, 1, 2020-05-07 20:49:36, 2020-05-07 20:49:36
<==        Row: 3, Tom, 28, test3@baomidou.com, 1, 2020-05-07 20:49:36, 2020-05-07 20:49:36
<==      Total: 3
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@7ad1caa2]
User(id=1, name=libai777, age=20, email=libai666@126.com, version=3, createTime=Thu May 07 20:49:36 CST 2020, update
User(id=2, name=Jack, age=20, email=test2@baomidou.com, version=1, createTime=Thu May 07 20:49:36 CST 2020, updateT
User(id=3, name=Tom, age=28, email=test3@baomidou.com, version=1, createTime=Thu May 07 20:49:36 CST 2020, updateTi
***分页查询总数量 total：10
```

删除操作

```
// 测试单量删除
@Test
public void testDeleteById() {
    int result = userMapper.deleteById(1258375518919606276L);
    System.out.println(result);
}

// 测试批量删除
@Test
public void testDeleteBatchIds() {
    int result = userMapper.deleteBatchIds(Arrays.asList(1258375518919606275L,
1258375518919606274L));
    System.out.println(result);
}

// 测试通过 map 删除
@Test
public void testDeleteByMap() {
    Map<String, Object> map = new HashMap<>();
    map.put("name", "小白");
    int result = userMapper.deleteByMap(map);
    System.out.println(result);
}
```

逻辑删除

什么是逻辑删除，见名知其一，也就是说根本没有从数据库中删除，而是给了一个变量暂时让它失效而已，很多商品网站都是这么设计得。不可能真的将数据删除，而是给出一个逻辑删除得效果。

> 物理删除：从数据库中直接移除
>
> 逻辑删除：数据库中依旧存在，而是通过一个变量来让它失效，deleted = 0 > deleted = 1

1. 数据库表增加一个 deleted 字段

列名	数据类型	长度	默认	主键?	非空?	Unsigned	自增?	Zerofill?	更新	注释
id	bigint	20		☑	☑	☐	☑	☐	☐	主键ID
name	varchar	30		☐	☐	☐	☐	☐	☐	姓名
age	int	11		☐	☐	☐	☐	☐	☐	年龄
email	varchar	50		☐	☐	☐	☐	☐	☐	邮箱
version	int	1	1	☐	☐	☐	☐	☐	☐	版本号
deleted	int	1	0	☐	☐	☐	☐	☐	☐	逻辑删除
create_time	datetime			☐	☐	☐	☐	☐	☐	创建时间
update_time	datetime			☐	☐	☐	☐	☐	☐	更新时间
				☐	☐	☐	☐	☐	☐	

2. 实体类中添加对应的属性【此注解用来标记逻辑删除的字段】

```
@TableLogic // 逻辑删除
private Integer deleted;
```

3. 添加对应得逻辑删除组件

```
// 逻辑删除
@Bean
public ISqlInjector sqlInjector() {
    return new LogicSqlInjector();
}
```

```
# 配置逻辑删除
# 逻辑已经删除得值为 1 （默认为 1）
mybatis-plus.global-config.db-config.logic-delete-value=1
# 逻辑未删除得值为 0 （默认为 0）
mybatis-plus.global-config.db-config.logic-not-delete-value=0
```

4. 测试删除

```
private UserMapper userMapper;

// 测试单量删除
@Test
public void testDeleteById() {
    int result = userMapper.deleteById( serializable: 1L);
    System.out.println(result);
}

// 测试批量删除
```

执行的是删除方法，但是却是更新操作，
这就是逻辑删除得应用。

```
MybatisPlusApplicationTests › testDeleteById()
MybatisPlusApplicationTests.testDeleteById
Tests passed: 1 of 1 test – 250 ms
Test Results 250 ms
MybatisPlusApplicationTests 250 ms
testDeleteById() 250 ms
Creating a new SqlSession
SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@2149594a] was not registered for synchronizatio
JDBC Connection [HikariProxyConnection@899736725 wrapping com.mysql.cj.jdbc.ConnectionImpl@5399f6c5] will not be
==>  Preparing: UPDATE user SET deleted=1 WHERE id=? AND deleted=0
==> Parameters: 1(Long)
<==    Updates: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@2149594a]
1
```

再查看数据库，发现 deleted 字段已经发生改变：

id	name	age	email	version	deleted	create_time	update_time
1	libai777	20	libai666@126.com	3	1	2020-05-07 20:49:36	2020-05-07 21:26:42
2	Jack	20	test2@baomidou.com	1	0	2020-05-07 20:49:36	2020-05-07 20:49:36
3	Tom	28	test3@baomidou.com	1	0	2020-05-07 20:49:36	2020-05-07 20:49:36

> 那么查询的时候会查询出 libai777 这个用户吗？

```
==>  Preparing: SELECT id,name,age,email,version,deleted,create_time,update_time FROM user WHERE deleted=0
==> Parameters:
<==    Columns: id, name, age, email, version, deleted, create_time, update_time
<==        Row: 2, Jack, 20, test2@baomidou.com, 1, 0, 2020-05-07 20:49:36, 2020-05-07 20:49:36
<==        Row: 3, Tom, 28, test3@baomidou.com, 1, 0, 2020-05-07 20:49:36, 2020-05-07 20:49:36
<==        Row: 4, Sandy, 21, test4@baomidou.com, 1, 0, 2020-05-07 20:49:36, 2020-05-07 20:49:36
<==        Row: 5, Billie, 24, test5@baomidou.com, 1, 0, 2020-05-07 20:49:36, 2020-05-07 20:49:36
<==      Total: 4
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@1f1e58ca]
User(id=2, name=Jack, age=20, email=test2@baomidou.com, version=1, deleted=0, createTime=Thu May 07 20:49:36 CST
User(id=3, name=Tom, age=28, email=test3@baomidou.com, version=1, deleted=0, createTime=Thu May 07 20:49:36 CST
User(id=4, name=Sandy, age=21, email=test4@baomidou.com, version=1, deleted=0, createTime=Thu May 07 20:49:36 CS
```

自动添加了过滤条件

性能分析插件

我们平时在开发中，会遇到一些慢SQL

我们可以通过性能分析并拿到这条慢SQL

性能分析作用：用于拦截每条SQL语句及其执行时间

1. 添加对应的插件

```
/**
 * SQL执行效率插件
 */
@Bean
@Profile({"dev", "test"})// 设置 dev test 环境开启
public PerformanceInterceptor performanceInterceptor() {
    PerformanceInterceptor performanceInterceptor = new
PerformanceInterceptor();
    // 设置 sql 执行的最大时间，如果超过了就不执行
    performanceInterceptor.setMaxTime(1);
    // 是否格式化代码
    performanceInterceptor.setFormat(true);
    return new PerformanceInterceptor();
}
```

2. 修改 application.properties 调整环境为 dev 开发环境或者 test 测试环境

```
# 默认开发环境
spring.profiles.active=dev
```

3. 测试

```
// SQL 性能分析
@Test
public void testSQLExplain() {
    List<User> users = userMapper.selectList(null);
    users.forEach(System.out::println);
}
```

```
<==      Total: 4
 Time:  16 ms - ID: com.javaboy.mapper.UserMapper.selectList
Execute SQL:
    SELECT
        id,
        name,
        age,
        email,
        version,
        deleted,
        create_time,
        update_time
    FROM
        user
    WHERE
        deleted=0
```

超出了指定时间就会报错哦

使用性能分析插件，可以优化查询的慢SQL，对于不懂SQL优化的就很尴尬了。

条件构造器

可以写一些比较复杂的 SQL ， 见名之意，就是多条件查询

1. 测试一：根据日志输出分析 SQL

```
// 查询 name 不为空的用户，并且邮箱不为空的用户，年龄大于等于 18
@Test
public void test1() {
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    wrapper.isNotNull("name")
            .isNotNull("email")
            .ge("age",12);
    userMapper.selectList(wrapper).forEach(System.out::println);
}
```

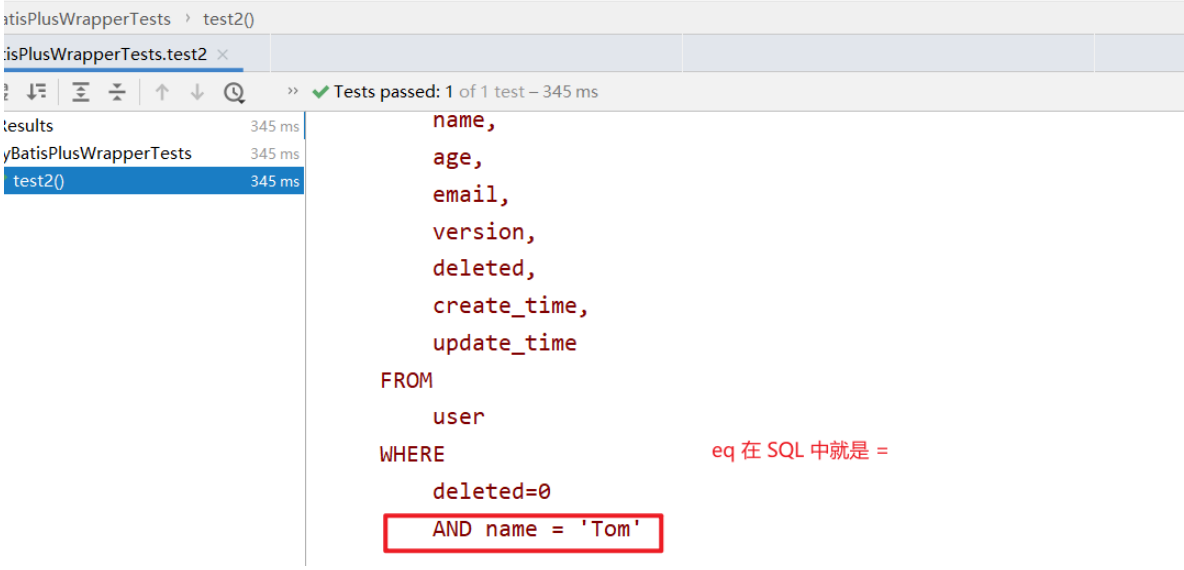
```
@Test
public void test1() {
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    wrapper.isNotNull( column: "name")
            .isNotNull( column: "email")
            .ge( column: "age", val: 12);
    userMapper.selectList(wrapper).forEach(System.out::println);
}
```

```
lyBatisPlusWrapperTests › test1()
/BatisPlusWrapperTests.test1
Tests passed: 1 of 1 test – 377 ms
st Results 377 ms
MyBatisPlusWrapperTests 377 ms
test1() 377 ms
        version,
        deleted,
        create_time,
        update_time
    FROM
        user
    WHERE
        deleted=0
        AND name IS NOT NULL
        AND email IS NOT NULL
        AND age >= 12
```

2. 测试，根据日志输出分析SQL

```
// 查询 name 为 Tom 的用户
@Test
public void test2() {
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    wrapper.eq("name","Tom");
    User user = userMapper.selectOne(wrapper);
    System.out.println(user);
}
```

```
// 查询 name 为 Tom 的用户
@Test
public void test2() {
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    wrapper.eq( column: "name", val: "Tom");
    User user = userMapper.selectOne(wrapper);
    System.out.println(user);
}
```



3. 测试3，查询指定区间的

```
// 查询年龄在 20 - 30 岁之间的用户
@Test
public void test3() {
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    wrapper.between("age",20,30);
    Integer count = userMapper.selectCount(wrapper); // 查询结果数量
    System.out.println(count);
}
```

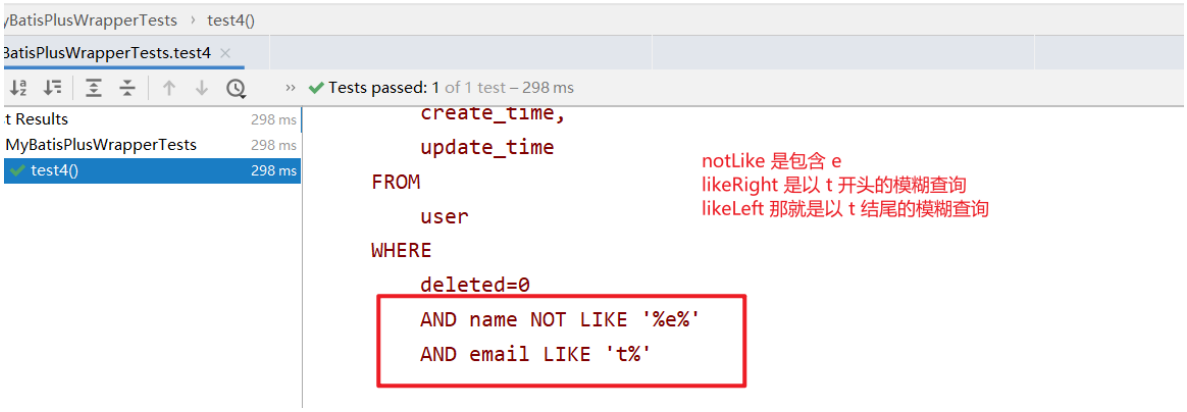
4. 测试4，模糊查询

```java
// 模糊查询
@Test
public void test4() {
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    wrapper.notLike("name","e")
            .likeRight("email","t");

    List<Map<String, Object>> maps = userMapper.selectMaps(wrapper);
    maps.forEach(System.out::println);
}
```

```java
@Test
public void test4() {
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    wrapper.notLike( column: "name", val: "e")
            .likeRight( column: "email", val: "t");

    List<Map<String, Object>> maps = userMapper.selectMaps(wrapper);
    maps.forEach(System.out::println);
}
```



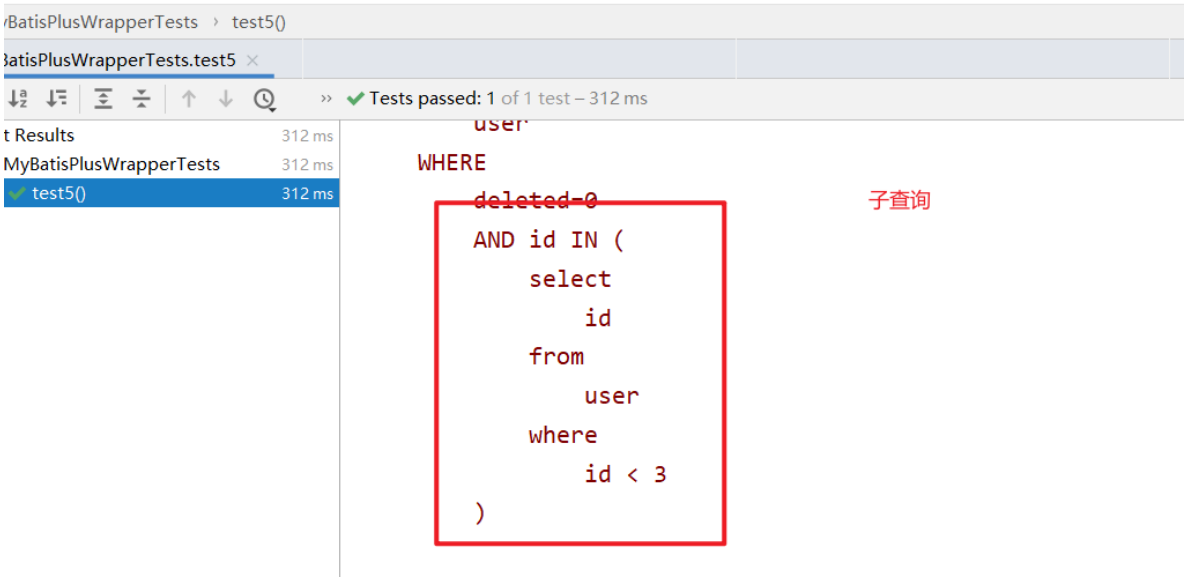
5. 测试5，子查询

```java
// 子查询
@Test
public void test5() {
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    wrapper.inSql("id","select id from user where id < 3");

    userMapper.selectObjs(wrapper).forEach(System.out::println);
}
```

```java
// 子查询
@Test
public void test5() {
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    wrapper.inSql( column: "id", inValue: "select id from user where id < 3");

    userMapper.selectObjs(wrapper).forEach(System.out::println);
}
```



6. 测试6，排序查询

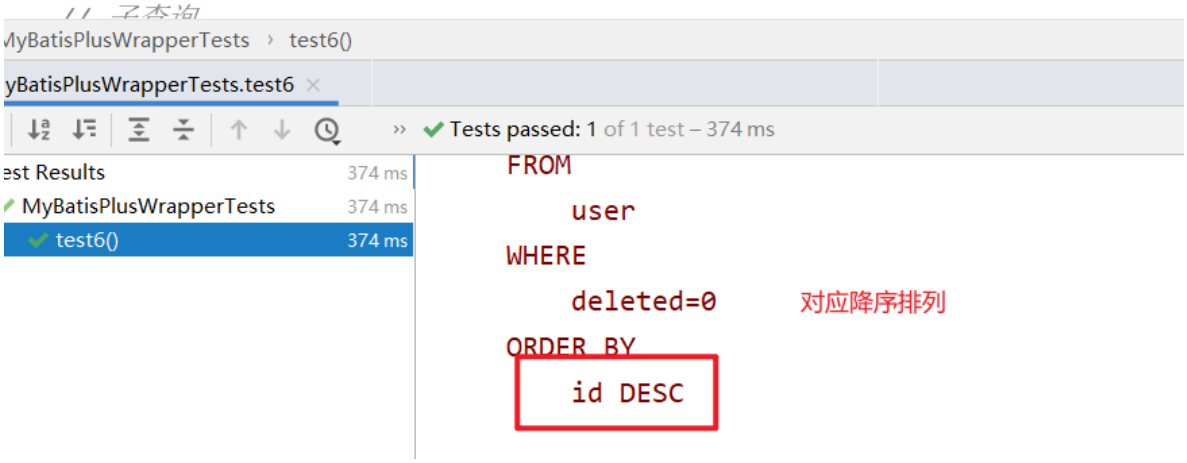
```java
// 排序查询
@Test
public void test6() {
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    // 通过Id进行排序
    wrapper.orderByDesc("id");

    userMapper.selectList(wrapper).forEach(System.out::println);
}
```

```java
// 排序查询
@Test
public void test6() {
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    // 通过Id进行排序
    wrapper.orderByDesc( ...columns: "id");

    userMapper.selectList(wrapper).forEach(System.out::println);
}

// 子查询
```



代码自动生成器

此功能用来生成 pojo、dao、service、controller

AutoGenerator 是 MyBatis-Plus 的代码生成器，通过 AutoGenerator 可以快速生成 entity、Mapper、Mapper.xml、Service、Controller 各个模块的代码，极大的提升了开发效率。

官网地址：[https://mp.baomidou.com/config/generator-config.html#%E5%9F%BA%E6%9C%AC%E9%85%8D%E7%BD%AE](https://mp.baomidou.com/config/generator-config.html#基本配置)