

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。

官网: <https://baomidou.com/>

简单使用

1. 测试数据

```
1 drop table if exists user;
2
3 create table user
4 (
5     id bigint(20) not null
6     auto_increment comment '主键id',
7     name varchar(30) null default null
8     comment '姓名',
9     age int(11) null default null
10    comment '年龄',
11    email varchar(50) null default null
12    comment '邮箱',
13    primary key (id)
14 );
15
16 delete from user;
```

```
13
14 insert into user (id, name, age, email)
    values
15 (1, 'Jone', 18, 'test1@baomidou.com'),
16 (2, 'Jack', 20, 'test2@baomidou.com'),
17 (3, 'Tom', 28, 'test3@baomidou.com'),
18 (4, 'Sandy', 21, 'test4@baomidou.com'),
19 (5, 'Billie', 24, 'test5@baomidou.com');
```

2. 导入pom依赖, mybatis-plus包会自动导入对应版本的mybatis

```
1 <!-- spring-webmvc -->
2 <dependency>
3
4     <groupId>org.springframework</groupId>
5     <artifactId>spring-
webmvc</artifactId>
6     <version>5.2.9.RELEASE</version>
7 </dependency>
8 <!-- spring-jdbc -->
9 <dependency>
10
11     <groupId>org.springframework</groupId>
12     <artifactId>spring-jdbc</artifactId>
13     <version>5.2.9.RELEASE</version>
14 </dependency>
15 <!-- spring织入 -->
16 <dependency>
17
18     <groupId>org.aspectj</groupId>
```

```
16     <artifactId>aspectjweaver</artifactId>
17     <version>1.9.4</version>
18 </dependency>
19 <!-- spring-test -->
20 <dependency>
21
22     <groupId>org.springframework</groupId>
23     <artifactId>spring-test</artifactId>
24     <version>5.2.9.RELEASE</version>
25 </dependency>
26 <!-- junit -->
27 <dependency>
28     <groupId>junit</groupId>
29     <artifactId>junit</artifactId>
30     <version>4.12</version>
31 </dependency>
32 <!-- mysql-jdbc -->
33 <dependency>
34     <groupId>mysql</groupId>
35     <artifactId>mysql-connector-
36     java</artifactId>
37     <version>5.1.47</version>
38 </dependency>
39 <!-- mybatis-plus -->
40 <dependency>
41     <groupId>com.baomidou</groupId>
42     <artifactId>mybatis-
43     plus</artifactId>
44     <version>3.4.3</version>
```

```
42 </dependency>
43 <!-- lombok -->
44 <dependency>
45     <groupId>org.projectlombok</groupId>
46     <artifactId>lombok</artifactId>
47     <version>1.18.22</version>
48 </dependency>
```

4. 创建数据库配置文件

database.properties

```
1 jdbc.driverClassName=com.mysql.jdbc.Driver
2 jdbc.url=jdbc:mysql://localhost:3306/mybatis_plus?
  useSSL=false&userUnicode=true&characterEncoding=utf-8&serverTimezone=GMT%2B8
3 jdbc.username=root
4 jdbc.password=lishuang001219
```

5. 创建Mybatis核心配置文件

mybatis-config.xml

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE configuration
3     PUBLIC "-//mybatis.org//DTD
4     Config 3.0//EN"
5     "http://mybatis.org/dtd/mybatis-
6     3-config.dtd">
7 <configuration>
8     <!--开启别名（包扫描）-->
9     <typeAliases>
10         <package name="entity"/>
11     </typeAliases>
12 </configuration>
```

6. 创建spring-mybatis-plus配置文件

spring-mybatis-plus.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans
3     xmlns="http://www.springframework.org/sc
4     hema/beans"
5     xmlns:xsi="http://www.w3.org/2001/XMLSch
6     ema-instance"
7     xmlns:context="http://www.springframewor
8     k.org/schema/context">
```

5

```
xsi:schemaLocation="http://www.springframework.org/schema/beans
```

6

```
https://www.springframework.org/schema/beans/spring-beans.xsd
```

7

```
http://www.springframework.org/schema/context
```

8

```
https://www.springframework.org/schema/context/spring-context.xsd">
```

9

```
<!--开启注解支持-->
```

```
<context:annotation-config/>
```

12

```
<!--关联数据库配置文件-->
```

```
<context:property-placeholder  
location="classpath:database.properties"  
/>
```

15

```
<!--配置数据源：数据源有非常多，可以使用第三方的，也可使使用Spring的-->
```

```
<bean id="dataSource"  
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
```

```
<property name="driverClassName"  
value="${jdbc.driverClassName}"/>
```

19

```
<property name="url"  
value="${jdbc.url}"/>
```

```
20         <property name="username"
value="${jdbc.username}"/>
21         <property name="password"
value="${jdbc.password}"/>
22     </bean>
23
24     <!--这里使用MP提供的sqlSessionFactory,
完成了Spring与MP的整合-->
25     <bean id="sqlSessionFactory"
26
class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean">
27         <property name="dataSource"
ref="dataSource"/>
28     </bean>
29
30 </beans>
```

7. 创建Spring核心配置文件

applicationContext.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans
  xmlns="http://www.springframework.org/sch
    ema/beans"
3
  xmlns:xsi="http://www.w3.org/2001/XMLSche
    ma-instance"
4
  xsi:schemaLocation="http://www.springfram
    ework.org/schema/beans
5
    https://www.springframework.org/schema/b
    eans/spring-beans.xsd">
6
7     <import resource="spring-mybatis-
    plus.xml" />
8
9 </beans>
```

8. 创建实体类

```
1 @Data
2 @NoArgsConstructor
3 @AllArgsConstructor
4 public class User {
5     private long id;
6     private String name;
7     private int age;
8     private String email;
9 }
```


9. 创建Mapper接口

```
1 // 可以使用Mpper注解标记Mapper
2 @Mapper
3 // 继承BaseMapper完成大部分CRUD操作
4 public interface UserMapper extends
    BaseMapper<User> {
5
6 }
```

Mpper接口过多的时候，配置自动扫描

```
1 <!--扫描mapper接口，使用的依然是Mybatis原生的
   扫描器-->
2 <bean
   class="org.mybatis.spring.mapper.MapperScannerConfigurer">
3     <property name="basePackage"
   value="mapper"/>
4 </bean>
```

10. 测试

```
1 @RunWith(SpringJUnit4ClassRunner.class)
2 @ContextConfiguration("classpath:applicationContext.xml") //注解寻找配置文件
3 public class MyTest {
4     @Autowired
5     private UserMapper userMapper;
6
7     @Test
```

```
8      public void test(){
9          List<User> users =
userMapper.selectList(null);
10         for (User user : users) {
11             System.out.println(user);
12         }
13     }
14 }
```

注解

表名映射

在实体类的名字与表名不对应的时候，我们需要手动给实体类配置表名映射

```
1 //注解的参数就是数据库表的名字
2 @TableName("tb_user")
3 public class User {
4     private long id;
5     private String name;
6     private int age;
7     private String email;
8 }
```

属性映射

在实体类字段与数据库表列名不一致的时候，可以手动配置，使其对应

```
1 @TableField("age")
2 private int ageOne;
```

主键生成策略

在插入数据时如果不设置主键,MP默认会使用雪花算法生成一个唯一id

雪花算法

snowflake是Twitter开源的分布式ID生成算法，结果是一个long型的ID。其核心思想是：使用41bit作为毫秒数，10bit作为机器的ID（5个bit是数据中心（北京、香港…），5个bit的机器ID），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），最后还有一个符号位，永远是0。

```
1 @TableId(type = IdType.AUTO)
2 private Long id;
```

```
1 public enum IdType {
2     AUTO, //数据库id自增，如果选择此方式，数据库主键也必须添加自增属性
3     INPUT, //手动输入，选择此方式，如果插入时没有id字段，会默认用null进行填充
4     ID_WORKER, //生成全局唯一id 雪花算法，默认
5     UUID, //全球唯一id uuid
6     NONE; //未设置主键
7 }
```

全局配置

对于批量的设置，使用注解会比较繁琐，我们可以使用xml进行批量全局配置

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans
  xmlns="http://www.springframework.org/sc
    hema/beans"
3
  xmlns:xsi="http://www.w3.org/2001/XMLSch
    ema-instance"
4
  xmlns:context="http://www.springframewor
    k.org/schema/context"
5
  xsi:schemaLocation="http://www.springfra
    mework.org/schema/beans
6
    https://www.springframework.org/schema/
    beans/spring-beans.xsd
7
    http://www.springframework.org/schema/c
    ontext
8
    https://www.springframework.org/schema/
    context/spring-context.xsd">
9
10 <!--开启包扫描，自动将注解注册为bean-->
```

```
11     <context:component-scan base-  
package="mapper" />  
12     <context:component-scan base-  
package="service" />  
13  
14     <!--关联数据库配置文件-->  
15     <context:property-placeholder  
location="classpath:database.properties"  
/>  
16  
17     <!--配置数据源：数据源有非常多，可以使用第  
三方的，也可使使用Spring的-->  
18     <bean id="dataSource"  
class="org.springframework.jdbc.datasour  
ce.DriverManagerDataSource">  
19         <property name="driverClassName"  
value="${jdbc.driverClassName}"/>  
20         <property name="url"  
value="${jdbc.url}"/>  
21         <property name="username"  
value="${jdbc.username}"/>  
22         <property name="password"  
value="${jdbc.password}"/>  
23     </bean>  
24  
25     <!--这里使用MP提供的sqlSessionFactory,  
完成了Spring与MP的整合-->  
26     <bean id="sqlSessionFactory"
```

```
27         class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean">
28             <property name="dataSource"
29                 ref="dataSource"/>
30             <!--将全局配置类注入到
31                 MybatisSqlSessionFactoryBean中-->
32             <property name="globalConfig"
33                 ref="globalConfig"/>
34             <!--将配置类注入到
35                 MybatisSqlSessionFactoryBean中-->
36             <property name="configuration"
37                 ref="configuration" />
38         </bean>
39
40         <!--动态扫描mapper接口，使用的依然是
41             Mybatis原生的扫描器-->
42         <bean
43             class="org.mybatis.spring.mapper.MapperScannerConfigurer">
44             <property name="basePackage"
45                 value="mapper"/>
46         </bean>
47
48         <bean id="dbConfig"
49             class="com.baomidou.mybatisplus.core.config.GlobalConfig$DbConfig">
50             <!--配置实体类表映射，统一前缀-->
51             <property name="tablePrefix"
52                 value="tb_" />
53         </bean>
54     </beans>
55 }
```

```
43         <!--配置主键生成策略-->
44         <property name="idType"
value="AUTO" />
45     </bean>
46
47     <bean id="globalConfig"
class="com.baomidou.mybatisplus.core.con
fig.GlobalConfig">
48         <property name="dbConfig"
ref="dbConfig" />
49     </bean>
50
51     <bean id="configuration"
class="com.baomidou.mybatisplus.core.Myb
atisConfiguration" >
52         <!--关闭驼峰自动映射-->
53         <property
name="mapUnderscoreToCamelCase"
value="false" />
54         <!--配置日志实现类-->
55         <property name="logImpl"
value="org.apache.ibatis.logging.stdout.
StdOutImpl" />
56     </bean>
57 </beans>
```

Wrapper

我们在实际操作数据库的时候会涉及到很多的条件，所以MP为我们提供了一个功能强大的条件构造器Wrapper，使用它可以让我们非常方便的构造条件。

Wrapper接口提供了一个抽象子类**AbstractWrapper**

AbstractWrapper抽象类有两个子类**QueryWrapper**和**UpdateWrapper**，我们使用这两个类来编写条件

QueryWrapper

常用方法

eq: equals, 等于

gt: greater than , 大于 >

ge: greater than or equals, 大于等于≥

lt: less than, 小于<

le: less than or equals, 小于等于≤

between: 相当于SQL中的BETWEEN

like: 模糊匹配。like("name","黄"), 相当于SQL的
name like '%黄%'

likeRight: 模糊匹配右半边。likeRight("name","黄"),
相当于SQL的name like '黄%'

likeLeft: 模糊匹配左半边。likeLeft("name","黄"), 相
当于SQL的name like '%黄'

notLike: notLike("name","黄"), 相当于SQL的name
not like '%黄%'

isNull

isNotNull

and: SQL连接符AND

or: SQL连接符OR

in: in("age",{1,2,3})相当于 age in(1,2,3)

groupBy: groupBy("id","name")相当于 group by
id,name

orderByAsc :orderByAsc("id","name")相当于 order by
id ASC,name ASC

orderByDesc :orderByDesc ("id","name")相当于 order
by id DESC,name DESC

select id,name,age,email from user where age>18
and email='349636607@qq.com'

```
1 @Test
2 public void testWrapper01(){
3     QueryWrapper wrapper = new
4     QueryWrapper();
5     wrapper.gt("age",18);
6     wrapper.eq("email","349636607@qq.com");
7     List<User> users =
8     userMapper.selectList(wrapper);
9     System.out.println(users);
10 }
```

select id,name,age,email from user where id
in(1,2,3) and age between 12 and 29 and name
like '%李%'

```
1 @Test
2 public void testWrapper02(){
3     QueryWrapper<User> wrapper = new
4     QueryWrapper<>();
5     wrapper.in("id",1,2,3);
6     wrapper.between("age",12,29);
7     wrapper.like("name","李");
8     List<User> users =
9     userMapper.selectList(wrapper);
10    System.out.println(users);
11 }
```

select id,name,age,email from user where id
in(1,2,3) and age > 10 order by age desc

```
1 @Test
2 public void testWrapper03(){
3     QueryWrapper<User> queryWrapper = new
4     QueryWrapper<>();
5     queryWrapper.in("id",1,2,3);
6     queryWrapper.gt("age",10);
7     queryWrapper.orderByDesc("age");
8     List<User> users =
9     userMapper.selectList(queryWrapper);
10    System.out.println(users);
11 }
```

select方法

select用于指定查询哪些列

select id,name from user

```
1 @Test
2 public void testSelect01(){
3     QueryWrapper<User> queryWrapper = new
4     QueryWrapper<>();
5     queryWrapper.select("id","name");
6     List<User> users =
7     userMapper.selectList(queryWrapper);
8     System.out.println(users);
9 }
```

方法的第一个参数为实体类的class对象，第二个参数为Predicate类型，可以使用lambda的写法，过滤要查询的字段(主键除外)。

重写test方法，return true表示此字段要查询，false表示忽略不查询

select id,name from user

```

1  @Test
2  public void testSelect02(){
3      QueryWrapper<User> queryWrapper =
4      new QueryWrapper<>();
5      queryWrapper.select(User.class, new
6      Predicate<TableFieldInfo>() {
7          @Override
8          public boolean
9          test(TableFieldInfo tableFieldInfo) {
10             return
11             "name".equals(tableFieldInfo.getColumn()
12             );
13         }
14     });
15     List<User> users =
16     userMapper.selectList(queryWrapper);
17     System.out.println(users);
18 }

```

方法第一个参数为Predicate类型，可以使用lambda的写法，过滤要查询的字段(主键除外)。(此方法存在问题)

忽略email这一列进行查询

```

| select id,name,age from user

```

```
1  @Test
2  public void testSelect03(){
3      QueryWrapper<User> queryWrapper =
4      new QueryWrapper<>(new User());
5      queryWrapper.select(new
6      Predicate<TableFieldInfo>() {
7          @Override
8          public boolean
9          test(TableFieldInfo tableFieldInfo) {
10             return
11             !"email".equals(tableFieldInfo.getColumn
12             ());
13         }
14     });
15     List<User> users =
16     userMapper.selectList(queryWrapper);
17     System.out.println(users);
18 }
```

UpdateWrapper

我们前面在使用update方法时需要创建一个实体类对象传入，用来指定要更新的列及对应的值。但是如果需要更新的列比较少时，创建这么一个对象显的有点麻烦和复杂。

我们可以使用UpdateWrapper的set方法来设置要更新的列及其值。同时这种方式也可以使用Wrapper去指定更复杂的更新条件。

```
update user set age = 99 where id > 1
```

```
1 @Test
2 public void testUpdateWrapper(){
3     UpdateWrapper<User> updateWrapper =
4     new UpdateWrapper<>();
5     updateWrapper.gt("id",1);
6     updateWrapper.set("age",99);
7
8     userMapper.update(null,updateWrapper);
9 }
```

Lambda条件构造器

我们前面在使用条件构造器时列名都是用字符串的形式去指定。这种方式无法在编译期确定列名的合法性。

所以MP提供了一个Lambda条件构造器可以让我们直接以实体类的方法引用的形式来指定列名。这样就可以弥补上述缺陷。

```
select id,name,age,email from user where age >
18 and email = '349636607@qq.com'
```

如果使用之前的条件构造器写法如下

```
1 @Test
2 public void testLambdaWrapper(){
3     QueryWrapper<User> queryWrapper = new
    QueryWrapper();
4     queryWrapper.gt("age",18);
5
6     queryWrapper.eq("email","349636607@qq.co
    m");
7     List<User> users =
    userMapper.selectList(queryWrapper);
8 }
```

如果使用Lambda条件构造器写法如下

```
1 @Test
2 public void testLambdaWrapper2(){
3     LambdaQueryWrapper<User> queryWrapper
    = new LambdaQueryWrapper<>();
4     queryWrapper.gt(User::getAge,18);
5
6     queryWrapper.eq(User::getEmail,"34963660
    7@qq.com");
7     List<User> users =
    userMapper.selectList(queryWrapper);
8 }
```

自定义SQL

虽然MP为我们提供了很多常用的方法，并且也提供了条件构造器。但是如果真的遇到了复杂的SQL时，我们还是需要自己去定义方法，自己去写对应的SQL，这样SQL也更有利于后期维护。

因为MP是对mybatis做了增强，所以还是支持之前Mybatis的方式去自定义方法。

同时也支持在使用Mybatis的自定义方法时使用MP的条件构造器帮助我们进行条件构造。

1. 在Mapper中定义方法

```
1 public interface UserMapper extends  
   BaseMapper<User> {  
2  
3     User findMyUser(Long id);  
4 }  
5
```

2. 创建Mapper映射文件

UserMapper.xml


```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD
4     Mapper 3.0//EN"
5     "http://mybatis.org/dtd/mybatis-
6     3-mapper.dtd">
7 <!-- namespace命名空间：绑定一个对应的Mapper
8     接口 -->
9 <mapper namespace="mapper.UserMapper">
10     <select id="findMyUser"
11         resultType="entity.User">
12         select * from tb_user where id =
13         #{id}
14     </select>
15 </mapper>

```

3. 如果没有动态扫码mapper，就需要在在配置文件中手动注入mapper

```

1 <bean id="sqlSessionFactory"
2     class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean">
3     <property name="mapperLocations"
4     value="classpath:/mapper/**/*.xml" />
5 </bean>

```

4. 测试

```
1 @Test
2 public void test(){
3     User myUser =
4     userMapper.findMyUser(11);
5     System.out.println(myUser);
6 }
```

分页插件

1. 配置分页过滤器

spring-mybatis-plus.xml

```
1 <bean id="mybatisPlusInterceptor"
2     class="com.baomidou.mybatisplus.extension.plugins.MybatisPlusInterceptor">
3     <property name="interceptors">
4         <list>
5             <ref
6                 bean="paginationInnerInterceptor"/>
7         </list>
8     </property>
9 </bean>
10
11 <bean id="paginationInnerInterceptor"
12     class="com.baomidou.mybatisplus.extension.plugins.inner.PaginationInnerInterceptor">
13     <!-- 对于单一数据库类型来说,都建议配置该
14     值,避免每次分页都去抓取数据库类型 -->
```

```
11     <constructor-arg name="dbType"
value="H2"/>
12 </bean>
13
14 <bean id="sqlSessionFactory"
15
    class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean">
16     <property name="dataSource"
ref="dataSource"/>
17     <property name="globalConfig"
ref="globalConfig"/>
18     <property name="configuration"
ref="configuration" />
19     <property name="plugins">
20         <array>
21             <ref
bean="mybatisPlusInterceptor"/>
22         </array>
23     </property>
24 </bean>
```

2. 测试

```
1 @Test
2 public void testPage(){
3     /**
4         * new Page<>(current,size);
5         * current: 页码
6         * size: 每一页容量
7         */
```

```

8         IPage<User> page = new Page<>(1,2);
9
10        userMapper.selectPage(page, null);
11        System.out.println("总页数" +
page.getPages());
12        System.out.println("当前页的数据：" +
page.getRecords());
13        System.out.println("总记录数：" +
page.getTotal());
14        System.out.println("当前页码：" +
page.getCurrent());
15        System.out.println("页面容量" +
page.getSize());
16    }

```

除了使用MP提供的分页插件外，PageHelper也是可以正常用于MP的

Service CRUD接口

相比于Mapper接口，Service层主要是支持了更多批量操作的方法。

1. Service接口继承IService

```

1    public interface UserService extends
        IService<User> {
2
3    }

```

2. Service实现类继承ServiceImpl

```
1 @Service
2 public class UserServiceImpl extends
  ServiceImp1<UserMapper,User> implements
  UserService {
3
4 }
```

3. 测试

```
1 @Autowired
2 private UserService userService;
3
4
5 @Test
6 public void testSeervice(){
7     List<User> list = userService.list();
8     System.out.println(list);
9 }
```

代码生成器

MP提供了一个代码生成器，可以让我们一键生成实体类，Mapper接口，Service，Controller等全套代码

1. 添加依赖

```
1 <!--mybatisplus代码生成器-->
2 <dependency>
3     <groupId>com.baomidou</groupId>
```

```

4      <artifactId>mybatis-plus-
generator</artifactId>
5      <version>3.4.1</version>
6  </dependency>
7  <!--模板引擎-->
8  <dependency>
9      <groupId>org.freemarker</groupId>
10     <artifactId>freemarker</artifactId>
11 </dependency>
12 <!--slf4j日志-->
13 <!--springboot项目自带，spring项目需要手动导
    入-->
14 <dependency>
15     <groupId>org.slf4j</groupId>
16     <artifactId>slf4j-api</artifactId>
17     <version>1.7.7</version>
18 </dependency>
19 <dependency>
20     <groupId>org.slf4j</groupId>
21     <artifactId>slf4j-
    log4j12</artifactId>
22     <version>1.7.7</version>
23 </dependency>

```

2. 创建生成类，并执行

旧版

```

1 public class CodeGenerator {
2     public static void main(String[]
    args) {

```

```
3          // 1. 构建一个代码生成器对象
4          AutoGenerator ag = new
AutoGenerator();
5
6          // 2. 配置策略
7          // 2.1 全局配置
8          GlobalConfig gc = new
GlobalConfig();
9          String projectPath =
System.getProperty("user.dir"); //获取当前
项目路径
10
    gc.setOutputDir(projectPath+"/src/main/
java"); //将代码生成到项目路径下
11          gc.setAuthor("John.Cena"); // 设
置作者
12          gc.setOpen(false); // 生成完是否打
开文件夹
13          gc.setFileOverride(true); // 是否
覆盖文件
14          gc.setServiceName("%sService");
// service文件名字格式
15          gc.setIdType(IdType.AUTO); // 主
键生成策略
16
    gc.setDateType(DateType.ONLY_DATE); //
日期格式
17          gc.setSwagger2(true); // 是否配置
swagger
18          ag.setGlobalConfig(gc);
```

```
19
20      // 2.2 数据源配置
21      DataSourceConfig dsc = new
DataSourceConfig();
22      dsc.setDbType(DbType.MYSQL); //
数据库类型
23
    dsc.setDriverName("com.mysql.jdbc.Driver
r"); // 驱动名字
24
    dsc.setUrl("jdbc:mysql://localhost:3306
/mybatis_plus?
useUnicode=true&characterEncoding=utf-
8&serverTimezone=GMT%2B8"); // url
25      dsc.setUsername("root");
26
    dsc.setPassword("lishuang001219");
27      ag.setDataSource(dsc);
28
    // 2.3 包配置
29
    PackageConfig pc = new
PackageConfig();
30
    pc.setParent("com.generator");
31
    ag.setPackageInfo(pc);
32
    // 2.4 策略配置
33
    StrategyConfig sc = new
StrategyConfig();
34
    sc.setInclude("tb_user"); // 设置
要映射的表名
```



```

37         sc.setNaming(NamingStrategy.underline_t
o_camel); // 设置类命名规则-下划线转驼峰
38         sc.setColumnNaming(NamingStrategy.under
line_to_camel); //设置字段命名规则
39         sc.setEntityLombokModel(true);
//是否使用Lombok
40         sc.setRestControllerStyle(true);
// controller采用restful风格
41         //同时,也可以配置自动填充,逻辑删除,乐
观锁等等...
42         ag.setStrategy(sc);
43
44         //使用Freemarker作为模板引擎
45         ag.setTemplateEngine(new
FreemarkerTemplateEngine());
46
47         // 3. 执行代码生成器
48         ag.execute();
49     }
50 }

```

新版

```

1 public class CodeGenerator {
2     public static void main(String[]
args) {

```

```
3      FastAutoGenerator.create("jdbc:mysql://
localhost:3306/mybatis_plus?
useUnicode=true&characterEncoding=utf-
8&serverTimezone=GMT%2B8", "root",
"lishuang001219")
4          .globalConfig(builder ->
5      {
6          builder.author("John.Cena") // 设置作者
7      /
8      .enableSwagger() // 开启 swagger 模式，这里
        开启的是对swagger2的支持
9      .fileOverride() // 覆盖已生成文件
10     .outputDir(".\\src\\main\\java"); // 指定
        输出到项目路径下
11     })
12     .packageConfig(builder -
13     > {
14         builder.parent("com") // 设置父包名
15     .moduleName("generator"); // 设置父包模块名
16     //
17     .pathInfo(Collections.singletonMap(Output
        File.mapperXml, "D://")); // 设置
        mapperXml生成路径
18     })
```

```
15         .strategyConfig(builder
-> {
16         builder.addInclude("tb_user") // 设置需要
        生成的表名
17         .addTablePrefix("tb_", "c_"); // 设置过滤
        表前缀
18         })
19         //使用freemark引擎要引入
        freemark的依赖
20         .templateEngine(new
        FreemarkerTemplateEngine()) // 使用
        Freemarker引擎模板，默认的是Velocity引擎模板
21         .execute();
22     }
23 }
```

自动填充

创建时间、更改时间！ 这些操作一般都是自动化完成，我们不希望手动更新

阿里巴巴开发手册：几乎所有的表都要配置

`gmt_create`、`gmt_modified`这两个字段，而且需要自动化

方式一：数据库级别（工作中不允许修改数据库级别）

1. 在表中增加字段：create_time,update_time，并设置默认值为当前时间戳

<input type="checkbox"/>	列名	数据类型	长度	默认	主键?	非空?	Unsigned	自增?	Zerofill?	更新	注释	虚拟	表达
<input type="checkbox"/>	id	bigint	20		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	主键ID	(n...)	
<input type="checkbox"/>	name	varchar	30		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	姓名	(n...)	
<input type="checkbox"/>	age	int	11		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	年龄	(n...)	
<input type="checkbox"/>	email	varchar	50		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	邮箱	(n...)	
<input type="checkbox"/>	create_time	datetime		CURRENT_TIMESTAMP	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	创建时间	(n...)	
<input type="checkbox"/>	update_time	datetime		CURRENT_TIMESTAMP	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	更新时间	(n...)	
<input type="checkbox"/>					<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>			

2. 更新实体类

```
1 @Data
2 @AllArgsConstructor
3 @NoArgsConstructor
4 @Component
5 public class User {
6
7     private Long id;
8     private String name;
9     private Integer age;
10    private String email;
11    private Date createTime;//驼峰命名
12    private Date updateTime;
13 }
```

3. 插入数据时，不用设置新增的两个字段，数据库会自动创建、更新

方式二：代码级别

1. 在表中增加字段：create_time,update_time，不用设置默认值

<input type="checkbox"/>	列名	数据类型	长度	默认	主键?	非空?	Unsigned	自增?	Zerofill?	更新	注释	虚拟	表达
<input type="checkbox"/>	id	bigint	20		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	主键ID	(n...	
<input type="checkbox"/>	name	varchar	30		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	姓名	(n...	
<input type="checkbox"/>	age	int	11		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	年龄	(n...	
<input type="checkbox"/>	email	varchar	50		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	邮箱	(n...	
<input type="checkbox"/>	create_time	datetime			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	创建时间	(n...	
<input type="checkbox"/>	update_time	datetime			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	更新时间	(n...	

2. 更新实体类，使用注解完成默认值操作

```
1 //字段  字段添加填充内容
2 @TableField(fill =
  FieldFill.INSERT)//value =
  ("create_time"),
3 private Date createTime;
4
5 @TableField(fill =
  FieldFill.INSERT_UPDATE)
6 private Date updateTime;
```

3. 创建处理器

```
1 @Slf4j//日志
2 @Component//注册组件
3 public class MyMetaObjectHandler extends
  MetaObjectHandler { //extends
4     @Override//插入时的填充策略
5     public void insertFill(MetaObject
  metaObject) {
6         log.info("==start insert
  .....==");
```

```

7      //setFieldValByName(java.lang.String
      fieldName, java.lang.Object fieldVal,
      org.apache.ibatis.reflection.MetaObject
      metaObject)
8
      this.setFieldValByName("createTime", new
      Date(), metaObject);
9
      this.setFieldValByName("updateTime", new
      Date(), metaObject);
10     }
11     @Override//更新时的填充策略
12     public void updateFill(MetaObject
      metaObject) {
13         log.info("==start update
      .....==");
14
      this.setFieldValByName("updateTime", new
      Date(), metaObject);
15     }
16 }

```

4. 插入数据时，不用设置新增的两个字段，数据库会自动创建、更新

逻辑删除

我们在删除的时候通常不会使用delete真正地对数据进行删除。而是使用一个标志字段，控制数据的展示

注：如果3.3.0版本之前还需要在对应的逻辑字段上加上

@TableLogic 注解

1. 在数据库表中加入一个标志字段，例如flag，用1表示删除，用0表示未删除
2. 在配置文件中配置逻辑删除

```
1 <bean id="dbConfig"
  class="com.baomidou.mybatisplus.core.config.GlobalConfig$DbConfig">
2   <property name="tablePrefix"
  value="tb_" />
3   <property name="idType" value="AUTO"
  />
4   <!--全局逻辑删除字段名-->
5   <property name="logicDeleteField"
  value="flag" />
6   <!--用1表示逻辑已删除值-->
7   <property name="logicDeleteValue"
  value="1" />
8   <!--用0表示逻辑未删除值-->
9   <property name="logicNotDeleteValue"
  value="0" />
10 </bean>
11
12 <bean id="globalConfig"
  class="com.baomidou.mybatisplus.core.config.GlobalConfig">
13   <property name="dbConfig"
  ref="dbConfig" />
```

3. 测试

```
1 //查看控制台两条sql执行语句发现，逻辑删除已经实现
2 //UPDATE tb_user SET flag=1 WHERE id=?
  AND flag=0
3 //SELECT id,name,age,email,flag FROM
  tb_user WHERE id IN ( ? , ? , ? ) AND
  flag=0
4
5 @Test
6 public void testDeleteById(){
7     //删除单个值
8     userMapper.deleteById(1);
9 }
10
11 @Test//通过id查询多个用户
12 public void testSelectBatchIds(){
13     List<User> users =
14     userMapper.selectBatchIds(Arrays.asList(
15     1L, 2L, 3L));
16     users.forEach(System.out::println);
17     //System.out.println(users);
18 }
```

乐观锁&悲观锁

乐观锁：顾名思义十分乐观，他总是认为不会出现问题，无论干什么都不上锁，如果出现了问题，再次更新值测试

悲观锁：顾名思义十分悲观，他总是认为出现问题，无论干什么都会上锁，再去操作！

乐观锁机制

1. 取出记录时，获取当前version
2. 更新时，带上这个version
3. 执行更新时，set version = newVersion where version = oldVersion，如果version不对，就更新失败

```
1  --乐观锁：先查询，获得版本号
2  -- A
3  update user set name = "wsk",version =
   version+1
4  where id = 1 and version = 1
5  -- B   （B线程抢先完成，此时version=2，会导致A
   线程修改失败！）
6  update user set name = "wsk",version =
   version+1
7  where id = 1 and version = 1
```

MP中乐观锁的使用

1. 表中添加version字段
2. 实体类中添加属性

```
1 //乐观锁version注解
2 @Version
3 private Integer version;
```

3. 注册组件，一般会把Mybatis的配置专门抽取出来，单独写一个配置类

```
1 //扫描mapper文件夹
2 @MapperScan("com.wsk.mapper")//交给
  mybatis做的，可以让这个配置类做扫描
3 @EnableTransactionManagement//自动管理事务
4 @Configuration//配置类
5 public class MyBatisPlusConfig {
6     //注册乐观锁插件
7     @Bean
8     public OptimisticLockerInterceptor
  optimisticLockerInterceptor(){
9         return new
  OptimisticLockerInterceptor();
10     }
11 }
```

4. 测试

成功

```
1 @Test
2 public void test(){
3     QueryWrapper<User> wrapper = new
    QueryWrapper<>();
4     wrapper.eq("id", 3);
5     User user =
    userMapper.selectOne(wrapper);
6
7     user.setAge(25);
8     int i = userMapper.updateById(user);
9 }
```

失败

```
1 @Test
2 public void test(){
3     User user1 =
    userMapper.selectById(11);
4     user1.setAge(1);
5     user1.setName("线程一");
6
7     //模拟线程插队
8     User user2 =
    userMapper.selectById(11);
9     user2.setAge(100);
10    user2.setName("线程二");
11
12    userMapper.updateById(user2);
13    userMapper.updateById(user1);
14 }
```

CRUD

增

```
1 @Test
2 public void testInsert(){
3     User user = new User();
4     user.setName("John.Cena");
5     user.setAge(21);
6     user.setEmail("349636607@qq.com");
7     userMapper.insert(user);
8 }
```

删

```
1 @Test
2 public void testDeleteById(){
3     //删除单个值
4     userMapper.deleteById(1);
5 }
6
7 @Test
8 public void testDeleteBatchIds(){
9     ArrayList<Integer> integers = new
    ArrayList<Integer>();
10    //删除多个值
11    integers.add(2);
12    integers.add(3);
13    integers.add(4);
```

```

14     userMapper.deleteBatchIds(integers);
15 }
16
17 @Test
18 public void testD(){
19     HashMap<String, Object> map = new
    HashMap<String, Object>();
20     //按条件删除，多个条件用and连接
21     map.put("age", "18");
22     map.put("name", "John.Cena");
23     userMapper.deleteByMap(map);
24 }

```

改

```

1 @Test
2 public void testUpdate(){
3     User user = new User();
4     user.setId(3);
5     user.setName("John.Cena");
6     user.setAge(22);
7     user.setEmail("1111@qq.com");
8     //修改对应Id的内容
9     userMapper.updateById(user);
10 }

```

查

1. 通过id查询单个用户

```
1 @Test//通过id查询单个用户
2 public void testSelectById(){
3     User user =
4     userMapper.selectById(1L);
5     System.out.println(user);
6 }
```

2. 通过id查询多个用户

```
1 @Test//通过id查询多个用户
2 public void testSelectBatchIds(){
3     List<User> users =
4     userMapper.selectBatchIds(Arrays.asList(1
5     L, 2L, 3L));
6     users.forEach(System.out::println);
7     //System.out.println(users);
8 }
```

3. 条件查询，通过map封装

```

1  @Test//通过条件查询之一    map
2  public void testMap(){
3      HashMap<String, Object> map = new
HashMap<>();
4      //自定义要查询的
5      map.put("name", "www");
6      map.put("age", 18);
7      List<User> users =
userMapper.selectByMap(map);
8      users.forEach(System.out::println);
9  }

```

逻辑删除

- 物理删除：从数据库中直接删除
- 逻辑删除：在数据库中没有被删除，而是通过一个变量来使他失效！ deleted=0 ==> deleted=1

实现：

1. 数据库添加delete字段
2. 实体类添加属性

```

1  @TableLogic//逻辑删除注解
2  private Integer deleted;

```

3. 注册组件

```

1 //逻辑删除组件
2 @Bean
3 public ISqlInjector sqlInjector(){
4     return new LogicSqlInjector();
5 }

```

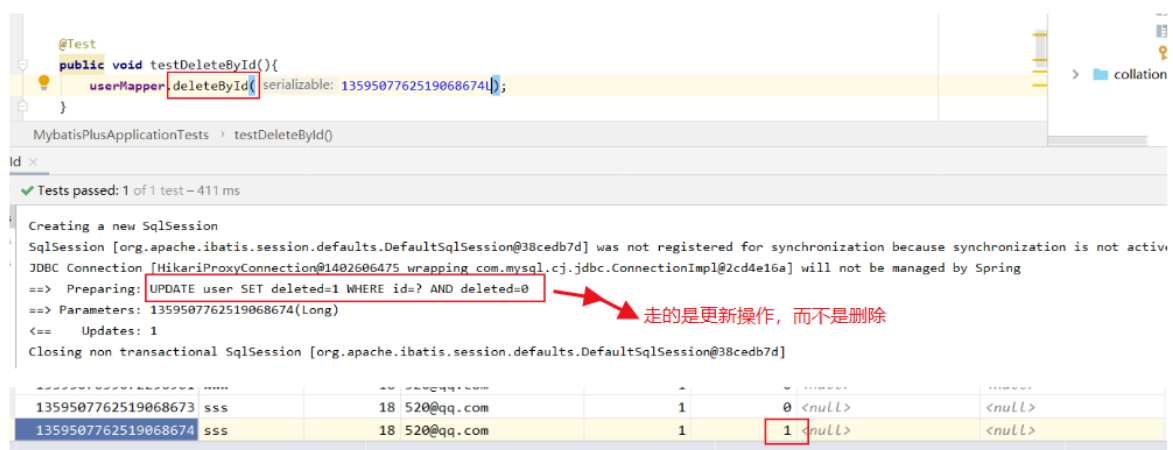
4. 添加配置

```

1 #配置逻辑删除 没删除的为0 删除的为1
2 mybatis-plus.global-config.db-
  config.logic-delete-value=1
3 mybatis-plus.global-config.db-
  config.logic-not-delete-value=0

```

5. 测试删除操作，会发现，数据库中的记录还在，只是 deleted 字段的值变为了 1，从日志中也可以看到，实际 sql 是写的更新操作



MybatisPlusApplicationTests > testDeleteById()

Tests passed: 1 of 1 test - 411 ms

Creating a new SqlSession
 SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@38cedb7d] was not registered for synchronization because synchronization is not active
 JDBC Connection [HikariProxyConnection@1402606475 wrapping com.mysql.cj.jdbc.ConnectionImpl@2cd4e16a] will not be managed by Spring
 ==> Preparing: UPDATE user SET deleted=1 WHERE id=? AND deleted=0
 ==> Parameters: 1359507762519068674(Long)
 <== Updates: 1
 Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@38cedb7d]

走的是更新操作，而不是删除

id	username	password	deleted	email	phone
1359507762519068673	sss	18	520@qq.com	1	0 <null>
1359507762519068674	sss	18	520@qq.com	1	1 <null>