

MyBatis-Plus快速入门

MyBatis-Plus快速入门

介绍

特性:

- 1、mybatis-plus 快速使用
- 2、基于mybatis-plus的入门helloworld---CRUD实验
- 3、不得不提的条件构造器---Wrapper
- 4.扩展

全局ID生成策略

逻辑删除

执行 SQL 分析打印

数据安全保护

乐观锁插件使用

- 5、代码生成器

介绍

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

就像 [魂斗罗](#) 中的 1P、2P, 基友搭配, 效率翻倍。



TO BE THE BEST PARTNER OF MYBATIS

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

官网：

<https://baomidou.com/>

1、mybatis-plus 快速使用

1.1、引入mybatis-plus相关maven依赖

```
1 <!-- https://mvnrepository.com/artifact/com.baomidou/mybatis-plus -->
2 <dependency>
3   <groupId>com.baomidou</groupId>
4   <artifactId>mybatis-plus</artifactId>
5   <version>3.3.1</version>
6 </dependency>
```

引入mybatis-plus在spring boot中的场景启动器

```
1 <!-- https://mvnrepository.com/artifact/com.baomidou/mybatis-plus-boot-starter -->
2 <dependency>
3   <groupId>com.baomidou</groupId>
4   <artifactId>mybatis-plus-boot-starter</artifactId>
5   <version>3.3.1</version>
6 </dependency>
```

ps:切记不可再在pom.xml文件中引入mybatis与mybatis-spring的maven依赖,这一点,mybatis-plus的官方文档中已经说明的很清楚了.

1.2、创建数据表

(1)SQL语句

```
1 -- 创建表
2 CREATE TABLE tbl_employee(
3   id INT(11) PRIMARY KEY AUTO_INCREMENT,
4   last_name VARCHAR(50),
5   email VARCHAR(50),
6   gender CHAR(1),
7   age INT
8 );
9 INSERT INTO tbl_employee(last_name,email,gender,age) VALUES('Tom','tom@atguigu.com',1,22);
10 INSERT INTO tbl_employee(last_name,email,gender,age) VALUES('Jerry','jerry@atguigu.com',0,25);
11 INSERT INTO tbl_employee(last_name,email,gender,age) VALUES('Black','black@atguigu.com',1,30);
12 INSERT INTO tbl_employee(last_name,email,gender,age) VALUES('White','white@atguigu.com',0,35);
```

(2) 数据表结构

<input type="checkbox"/>	id	last name	email	gender	age
<input type="checkbox"/>	1	Tom	tom@atguigu.com	1	22
<input type="checkbox"/>	2	Jerry	jerry@atguigu.com	0	25
<input type="checkbox"/>	3	Black	black@atguigu.com	1	30
<input type="checkbox"/>	4	White	white@atguigu.com	0	35

https://blog.csdn.net/qg_42681757

1.3、创建java bean

根据数据表新建相关实体类

```
1 package com.example.demo.pojo;
```

```
2
3 public class Employee {
4     private Integer id;
5     private String lastName;
6     private String email;
7     private Integer gender;
8     private Integer age;
9     public Employee() {
10         super();
11         // TODO Auto-generated constructor stub
12     }
13     public Employee(Integer id, String lastName, String email, Integer gender, Integer age) {
14         super();
15         this.id = id;
16         this.lastName = lastName;
17         this.email = email;
18         this.gender = gender;
19         this.age = age;
20     }
21     public Integer getId() {
22         return id;
23     }
24     public void setId(Integer id) {
25         this.id = id;
26     }
27     public String getLastName() {
28         return lastName;
29     }
30     public void setLastName(String lastName) {
31         this.lastName = lastName;
32     }
33     public String getEmail() {
34         return email;
35     }
36     public void setEmail(String email) {
37         this.email = email;
38     }
39     public Integer getGender() {
40         return gender;
41     }
42     public void setGender(Integer gender) {
43         this.gender = gender;
44     }
45     public Integer getAge() {
46         return age;
47     }
48     public void setAge(Integer age) {
49         this.age = age;
50     }
51     @Override
52     public String toString() {
53         return "Employee [id=" + id + ", lastName=" + lastName + ", email=" + email + ", gender=" + gender +
54             ", age="
```

```

54 + age + "】";
55 }
56
57
58 }

```

1.4、配置application.properties

数据源使用druid

```

1 spring.datasource.username=root
2 spring.datasource.password=20182022
3 spring.datasource.url=jdbc:mysql://127.0.0.1:3306/my?useUnicode=true&characterEncoding=UTF-8&useSSL=false&serverTimezone=GMT%2B8
4 spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
5
6 spring.datasource.type=com.alibaba.druid.pool.DruidDataSource

```

2、基于mybatis-plus的入门helloworld---CRUD实验

ps:在进行crud实验之前,简单对mybatis与mybatis-plus做一个简单的对比

2.1、mybatis与mybatis-plus实现方式对比

(1)提出问题: 假设我们已存在一张 tbl_employee 表, 且已有对应的实体类 Employee, 实现 tbl_employee 表的 CRUD 操作我们需要做什么呢?

(2)实现方式: 基于 Mybatis 需要编写 EmployeeMapper 接口, 并手动编写 CRUD 方法 提供 EmployeeMapper.xml 映射文件, 并手动编写每个方法对应的 SQL 语句. 基于 Mybatis-plus 只需要创建 EmployeeMapper 接口, 并继承 BaseMapper 接口.这就是使用 mybatis-plus 需要完成的所有操作, 甚至不需要创建 SQL 映射文件。

2.2、BaseMapper接口介绍

(1)如何理解核心接口BaseMapper?

在使用Mybatis-Plus是,核心操作类是BaseMapper接口, 其最终也是利用的Mybatis接口编程的实现机制, 其默认提供了一系列的增删改查的基础方法, 并且开发人员对于这些基础操作不需要写SQL进行处理操作 (Mybatis提供的机制就是需要开发人员在mapper.xml中提供sql语句), 那样我们可以猜测肯定是Mybatis-Plus完成了BaseMapper接口提供的方法的SQL语句的生成操作。

(2)BaseMapper接口为我们定义了哪些方法?

```

BaseMapper<T>
  insert(T) : int
  deleteById(Serializable) : int
  deleteByMap(@Param(value="cm") Map<String, Object>) : int
  delete(@Param(value="ew") Wrapper<T>) : int
  deleteBatchIds(@Param(value="coll") Collection<? extends Serializable>) : int
  updateById(@Param(value="et") T) : int
  update(@Param(value="et") T, @Param(value="ew") Wrapper<T>) : int
  selectById(Serializable) : T
  selectBatchIds(@Param(value="coll") Collection<? extends Serializable>) : List<T>
  selectByMap(@Param(value="cm") Map<String, Object>) : List<T>
  selectOne(@Param(value="ew") Wrapper<T>) : T
  selectCount(@Param(value="ew") Wrapper<T>) : Integer
  selectList(@Param(value="ew") Wrapper<T>) : List<T>
  selectMaps(@Param(value="ew") Wrapper<T>) : List<Map<String, Object>>
  selectObjs(@Param(value="ew") Wrapper<T>) : List<Object>
  selectPage(E, @Param(value="ew") Wrapper<T>) <E extends IPage<T>> : E
  selectMapsPage(E, @Param(value="ew") Wrapper<T>) <E extends IPage<Map<String, Object>>> : E

```

BaseMapper接口源码:

```

1 /*
2  * Copyright (c) 2011-2020, baomidou (jobob@qq.com).
3  *
4  * <p>
5  * Licensed under the Apache License, Version 2.0 (the "License"); you may not
6  * use this file except in compliance with the License. You may obtain a copy of
7  * the License at
8  *
9  * <p>
10  * https://www.apache.org/licenses/LICENSE-2.0

```

```

9  * <p>
10 * Unless required by applicable law or agreed to in writing, software
11 * distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
12 * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
13 * License for the specific language governing permissions and limitations under
14 * the License.
15 */
16 package com.baomidou.mybatisplus.core.mapper;
17
18 import com.baomidou.mybatisplus.core.conditions Wrapper;
19 import com.baomidou.mybatisplus.core.metadata.IPage;
20 import com.baomidou.mybatisplus.core.toolkit.Constants;
21 import org.apache.ibatis.annotations.Param;
22
23 import java.io.Serializable;
24 import java.util.Collection;
25 import java.util.List;
26 import java.util.Map;
27
28 /**
29 * Mapper 继承该接口后，无需编写 mapper.xml 文件，即可获得CRUD功能
30 * <p>这个 Mapper 支持 id 泛型</p>
31 *
32 * @author hubin
33 * @since 2016-01-23
34 */
35 public interface BaseMapper<T> extends Mapper<T> {
36
37     /**
38     * 插入一条记录
39     *
40     * @param entity 实体对象
41     */
42     int insert(T entity);
43
44     /**
45     * 根据 ID 删除
46     *
47     * @param id 主键ID
48     */
49     int deleteById(Serializable id);
50
51     /**
52     * 根据 columnMap 条件，删除记录
53     *
54     * @param columnMap 表字段 map 对象
55     */
56     int deleteByMap(@Param(Constants.COLUMN_MAP) Map<String, Object> columnMap);
57
58     /**
59     * 根据 entity 条件，删除记录
60     *
61     * @param wrapper 实体对象封装操作类（可以为 null）
62     */

```

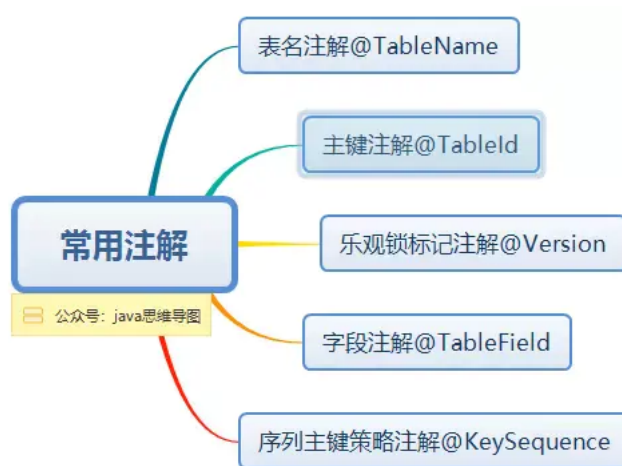
```
63  int delete(@Param(Constants.WRAPPER) Wrapper<T> wrapper);
64
65  /**
66   * 删除（根据ID 批量删除）
67   *
68   * @param idList 主键ID列表(不能为 null 以及 empty)
69   */
70  int deleteBatchIds(@Param(Constants.COLLECTION) Collection<? extends Serializable> idList);
71
72  /**
73   * 根据 ID 修改
74   *
75   * @param entity 实体对象
76   */
77  int updateById(@Param(Constants.ENTITY) T entity);
78
79  /**
80   * 根据 whereEntity 条件，更新记录
81   *
82   * @param entity 实体对象 (set 条件值,可以为 null)
83   * @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）
84   */
85  int update(@Param(Constants.ENTITY) T entity, @Param(Constants.WRAPPER) Wrapper<T> updateWrapper);
86
87  /**
88   * 根据 ID 查询
89   *
90   * @param id 主键ID
91   */
92  T selectById(Serializable id);
93
94  /**
95   * 查询（根据ID 批量查询）
96   *
97   * @param idList 主键ID列表(不能为 null 以及 empty)
98   */
99  List<T> selectBatchIds(@Param(Constants.COLLECTION) Collection<? extends Serializable> idList);
100
101  /**
102   * 查询（根据 columnMap 条件）
103   *
104   * @param columnMap 表字段 map 对象
105   */
106  List<T> selectByMap(@Param(Constants.COLUMN_MAP) Map<String, Object> columnMap);
107
108  /**
109   * 根据 entity 条件，查询一条记录
110   *
111   * @param queryWrapper 实体对象封装操作类（可以为 null）
112   */
113  T selectOne(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);
114
115  /**
116   * 根据 Wrapper 条件，查询总记录数
```

```

117  *
118  * @param queryWrapper 实体对象封装操作类（可以为 null）
119  */
120  Integer selectCount(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);
121
122  /**
123  * 根据 entity 条件，查询全部记录
124  *
125  * @param queryWrapper 实体对象封装操作类（可以为 null）
126  */
127  List<T> selectList(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);
128
129  /**
130  * 根据 wrapper 条件，查询全部记录
131  *
132  * @param queryWrapper 实体对象封装操作类（可以为 null）
133  */
134  List<Map<String, Object>> selectMaps(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);
135
136  /**
137  * 根据 wrapper 条件，查询全部记录
138  * <p>注意： 只返回第一个字段的值</p>
139  *
140  * @param queryWrapper 实体对象封装操作类（可以为 null）
141  */
142  List<Object> selectObjs(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);
143
144  /**
145  * 根据 entity 条件，查询全部记录（并翻页）
146  *
147  * @param page 分页查询条件（可以为 RowBounds.DEFAULT）
148  * @param queryWrapper 实体对象封装操作类（可以为 null）
149  */
150  <E extends IPage<T>> E selectPage(E page, @Param(Constants.WRAPPER) Wrapper<T> queryWrapper);
151
152  /**
153  * 根据 wrapper 条件，查询全部记录（并翻页）
154  *
155  * @param page 分页查询条件
156  * @param queryWrapper 实体对象封装操作类
157  */
158  <E extends IPage<Map<String, Object>>> E selectMapsPage(E page, @Param(Constants.WRAPPER)
    Wrapper<T> queryWrapper);
159  }

```

(3) mybatis-plus中常用的注解



```
1 @TableName: 对数据表名注解
2
3 @TableId: 表主键标识
4
5 @TableId(value = "id", type = IdType.AUTO): 自增
6
7 @TableId(value = "id", type = IdType.ID_WORKER_STR): 分布式全局唯一ID字符串类型
8
9 @TableId(value = "id", type = IdType.INPUT): 自行输入
10
11 @TableId(value = "id", type = IdType.ID_WORKER): 分布式全局唯一ID 长整型类型
12
13 @TableId(value = "id", type = IdType.UUID): 32位UUID字符串
14
15 @TableId(value = "id", type = IdType.NONE): 无状态
16
17 @TableField: 表字段标识
18
19 @TableField(exist = false): 表示该属性不为数据库表字段，但又是必须使用的。
20
21 @TableField(exist = true): 表示该属性为数据库表字段。
22
23 @TableField(condition = SqlCondition.LIKE): 表示该属性可以模糊搜索。
24
25 @TableField(fill = FieldFill.INSERT): 注解填充字段，生成器策略部分也可以配置！
26
27 @FieldStrategy:
28
29 @FieldFill
30
31 @Version: 乐观锁注解、标记
32
33 @EnumValue: 通枚举类注解
34
35 @TableLogic: 表字段逻辑处理注解（逻辑删除）
36
37 @SqlParser: 租户注解
38
39 @KeySequence: 序列主键策略
```

常用的就三个:@TableName @TableId @TableField

查看更多注解以及详解,请移步至官网:

<https://mybatis.plus/guide/annotation.html>

由于我们的数据表名于实体类的类名不一致,并且实体类于数据表还存在字段名不对应的情况,因此我们需要引入mybatis-plus的注解.

```
1 import com.baomidou.mybatisplus.annotation.IdType;
2 import com.baomidou.mybatisplus.annotation.TableField;
3 import com.baomidou.mybatisplus.annotation.TableId;
4 import com.baomidou.mybatisplus.annotation.TableName;
5 /*
6  * MybatisPlus会默认使用实体类的类名到数据中找对应的表.
7  *
8  */
9 @Component
10 @TableName(value = "tbl_employee")
11 public class Employee {
12     /*
13     * @TableId:
14     * value: 指定表中的主键列的列名, 如果实体属性名与列名一致, 可以省略不指定.
15     * type: 指定主键策略.
16     */
17     @TableId(value="id" , type =IdType.AUTO)
18     private Integer id;
19     @TableField(value = "last_name")
20     private String lastName;
21     private String email;
22     private Integer gender;
23     private Integer age;
24     public Employee() {
25         super();
26         // TODO Auto-generated constructor stub
27     }
28     public Employee(Integer id, String lastName, String email, Integer gender, Integer age) {
29         super();
30         this.id = id;
31         this.lastName = lastName;
32         this.email = email;
33         this.gender = gender;
34         this.age = age;
35     }
36     public Integer getId() {
37         return id;
38     }
39     public void setId(Integer id) {
40         this.id = id;
41     }
42     public String getLastName() {
43         return lastName;
44     }
45     public void setLastName(String lastName) {
46         this.lastName = lastName;
47     }
48     public String getEmail() {
49         return email;
50     }
```

```

51 public void setEmail(String email) {
52     this.email = email;
53 }
54 public Integer getGender() {
55     return gender;
56 }
57 public void setGender(Integer gender) {
58     this.gender = gender;
59 }
60 public Integer getAge() {
61     return age;
62 }
63 public void setAge(Integer age) {
64     this.age = age;
65 }
66 @Override
67 public String toString() {
68     return "Employee [id=" + id + ", lastName=" + lastName + ", email=" + email + ", gender=" + gender +
69     + age + "]";
70 }
71
72
73 }

```

3.增删查改操作

编写EmployeeMapper接口继承BaseMapper接口

```

1 package com.example.demo.mapper;
2
3 import org.apache.ibatis.annotations.Mapper;
4
5
6 import com.baomidou.mybatisplus.core.mapper.BaseMapper;
7 import com.example.demo.pojo.Employee;
8 /**
9  *
10  * @author zhou'en'xian
11  *基于Mybatis-plus实现：让XxxMapper接口继承 BaseMapper接口即可。
12  *BaseMapper<T> ：泛型指定的就是当前Mapper接口所操作的实体类类型
13  */
14 @Mapper
15 public interface EmpolyeeMapper extends BaseMapper<Employee> {
16
17 }
18

```

准备测试环境:

```

1 package com.example.demo;
2
3 import org.junit.jupiter.api.Test;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.boot.test.context.SpringBootTest;
6
7 import com.example.demo.mapper.EmpolyeeMapper;
8 import com.example.demo.pojo.Employee;
9

```

```

10 @SpringBootTest
11 class MybatisplusApplicationTests {
12     @Autowired
13     private Employee employee;
14
15     @Autowired
16     private EmpolyeeMapper empolyeeMapper;
17
18
19
20 }

```

(1)插入

1. // 插入一条记录

```

1 int insert(T entity);
2
3 @Test
4 void insert() {
5     employee.setAge(20);
6     employee.setEmail("123@qq.com");
7     employee.setGender(1);
8     employee.setLastName("张三");
9     empolyeeMapper.insert(employee);
10    //int id=employee.getId();此方法可以获取插入当前记录在数据库中的id
11    //在mybatis中如果立马获取插入数据的主键id,是不是需要配置呢?感受到mybatis-plus的强大了吗?
12
13
14 }

```

(2)修改

```

1 // 根据 ID 修改
2 int updateById(@Param(Constants.ENTITY) T entity);
3 //T entity 实体对象 (set 条件值,可为 null)
4 @Test
5 void update() {
6     employee.setId(1);
7     employee.setAge(18);
8     employee.setEmail("3123@hpu.edu");
9     employee.setGender(0);
10    employee.setLastName("lili");
11    empolyeeMapper.updateById(employee);
12 }

```

控制台打印出的sql语句

```
UPDATE tbl_employee SET last_name=?, email=?, gender=?, age=? WHERE id=?
lili(String), 318011@hpu.edu(String), 0(Integer), 18(Integer), 1(Integer)
```

如果我们不设置实体类的email与gender属性,结果是怎样的呢?

```

1 @Test
2 void update() {
3     employee.setId(2);
4     employee.setAge(21);
5     //employee.setEmail("318011@hpu.edu");
6     //employee.setGender(1);
7     employee.setLastName("lihua");
8     empolyeeMapper.updateById(employee);
9 }

```

控制台sql语句:

```
: ==> Preparing: UPDATE tbl_employee SET last_name=?, age=? WHERE id=?  
: ==> Parameters: lihua(String), 21(Integer), 2(Integer)  
: <==      Total: 1
```

显然,mybatis-plus为我们做了非空判断,空值的话,默认不更新对应的字段.想一想,这是不是类似于mybatis中的动态sql呢?
这种处理效果又会带来什么好处呢?

(3)查询

```
1 // 根据 ID 查询  
2 T selectById(Serializable id);  
3  
4  
5 // 查询 (根据ID 批量查询)  
6 List<T> selectBatchIds(@Param(Constants.COLLECTION) Collection<? extends Serializable> idList);  
7  
8 // 查询 (根据 columnMap 条件)  
9 List<T> selectByMap(@Param(Constants.COLUMN_MAP) Map<String, Object> columnMap);  
10
```

selectById方法

```
1 @Test  
2 void select() {  
3     Employee employee=empolyeeMapper.selectById(4);  
4     System.out.println(employee);  
5 }
```

```
Preparing: SELECT id,last_name,email,gender,age FROM tbl_employee WHERE id=?  
Parameters: 4(Integer)  
Total: 1
```

selectBatchIds方法

```
1 @Test  
2 void select() {  
3     List<Integer>list =new ArrayList<Integer>();  
4     list.add(1);  
5     list.add(2);  
6     list.add(3);  
7     List<Employee>li=empolyeeMapper.selectBatchIds(list);  
8     for(Employee employee:li) {  
9         System.out.println(employee);  
10    }  
11 }
```

```
: ==> Preparing: SELECT id,last_name,email,gender,age FROM tbl_employee WHERE id IN ( ?, ?, ? )  
: ==> Parameters: 1(Integer), 2(Integer), 3(Integer)  
: <==      Total: 2
```

ps:发现该方法底层使用的竟然是sql的in关键字

selectByMap方法

```
1 @Test  
2 void select() {  
3     Map<String,Object>map=new HashMap<String, Object>();  
4     map.put("age", 22);  
5     map.put("id", 16);  
6     List<Employee>li=empolyeeMapper.selectByMap(map);  
7     for(Employee employee:li) {  
8         System.out.println(employee);  
9     }  
10 }
```

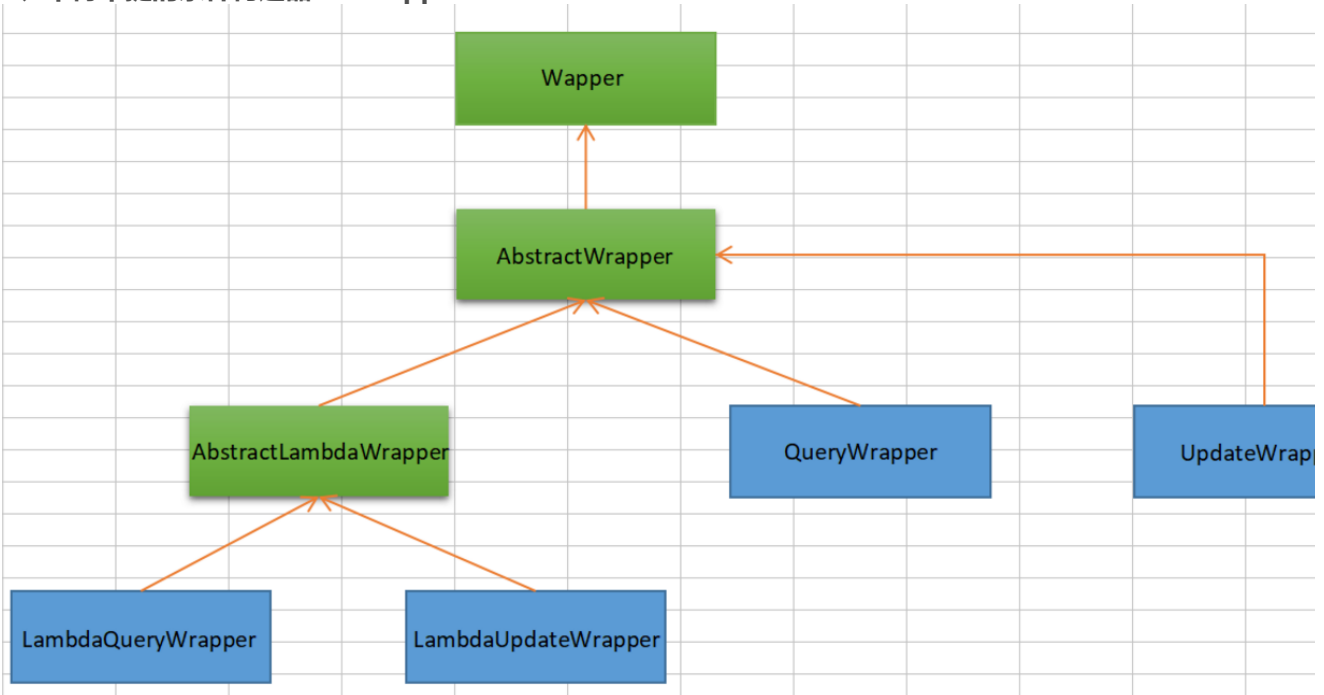
```
10 }
```

```
==> Preparing: SELECT id,last_name,email,gender,age FROM tbl_employee WHERE id = ? AND age = ?  
==> Parameters: 16(Integer), 22(Integer)  
<==      Total: 1
```

(4)删除

```
1  
2 // 删除（根据ID 批量删除）  
3 int deleteBatchIds(@Param(Constants.COLLECTION) Collection<? extends Serializable> idList);  
4 // 根据 ID 删除  
5 int deleteById(Serializable id);  
6 // 根据 columnMap 条件, 删除记录  
7 int deleteByMap(@Param(Constants.COLUMN_MAP) Map<String, Object> columnMap);
```

3、不得不提的条件构造器---Wrapper



3.1.wrapper及其子类介绍

(1)Wrapper：条件构造抽象类，最顶端父类，抽象类中提供3个方法以及其他方法。

```
public class Wrapper<T> {  
    Wrapper()  
    getEntity() : T  
    getSqlSelect() : String  
    getSqlSet() : String  
    getSqlComment() : String  
    getSqlFirst() : String  
    getExpression() : MergeSegments  
    getCustomSqlSegment() : String  
    isEmptyOfWhere() : boolean  
    nonEmptyOfWhere() : boolean  
    isEmptyOfNormal() : boolean  
    nonEmptyOfNormal() : boolean  
    nonEmptyOfEntity() : boolean  
    fieldStrategyMatch(T, TableFieldInfo) : boolean  
    isEmptyOfEntity() : boolean  
    getTargetSql() : String  
    clear() : void  
}
```

https://blog.csdn.net/qq_42681787

(2)AbstractWrapper：用于查询条件封装，生成 sql 的 where 条件,QueryWrapper(LambdaQueryWrapper) 和 UpdateWrapper(LambdaUpdateWrapper) 的父类用于生成 sql 的 where 条件, entity 属性也用于生成 sql 的 where 条件

```
• setEntityClass(Class<T>) : Children
• allEq(boolean, Map<R, V>, boolean) <V> : Children
• allEq(boolean, BiPredicate<R, V>, Map<R, V>, boolean) : Children
• eq(boolean, R, Object) : Children
• ne(boolean, R, Object) : Children
• gt(boolean, R, Object) : Children
• ge(boolean, R, Object) : Children
• lt(boolean, R, Object) : Children
• le(boolean, R, Object) : Children
• like(boolean, R, Object) : Children
• notLike(boolean, R, Object) : Children
• likeLeft(boolean, R, Object) : Children
• likeRight(boolean, R, Object) : Children
• between(boolean, R, Object, Object) : Children
• notBetween(boolean, R, Object, Object) : Children
• and(boolean, Consumer<Children>) : Children
• or(boolean, Consumer<Children>) : Children
• nested(boolean, Consumer<Children>) : Children
• or(boolean) : Children
• apply(boolean, String, Object...) : Children
• last(boolean, String) : Children
• comment(boolean, String) : Children
• first(boolean, String) : Children
```

AbstractWrapper比较重要,里面的方法需要重点学习.

该抽象类提供的重要方法如下:

函数名	说明	说明/例子
eq	等于=	例: eq("name", "老王")-->name = '老王'
ne	不等于<>	例: ne("name", "老王")-->name <> '老王'
gt	大于>	例: gt("age", 18)-->age > 18
ge	大于等于>=	例: ge("age", 18)-->age >= 18
lt	小于<	例: lt("age", 18)-->age < 18
le	小于=	例: le("age", 18)-->age <= 18
between	BETWEEN 值1 AND 值2	例: between("age", 18, 30)-->age between 18 and 30
notBetween	NOT BETWEEN 值1 AND 值2	例: notBetween("age", 18, 30)-->age not between 18 and 30
like	LIKE '%值%'	例: like("name", "王")-->name like '王%'
notLike	NOT LIKE '%值%'	例: notLike("name", "王")-->name not like '王%'
likeLeft	LIKE '%值'	例: likeLeft("name", "王")-->name like '王'
likeRight	LIKE '值%'	例: likeRight("name", "王")-->name like '王%'
isNull	字段 IS NULL	例: isNull("name")-->name is null
isNotNull	字段 IS NOT NULL	例: isNotNull("name")-->name is not null
in	字段 IN (v0, v1, ...)	例: in("age", {1, 2, 3})-->age in (1, 2, 3)
notIn	字段 NOT IN (v0, v1, ...)	例: notIn("age", 1, 2, 3)-->age not in (1, 2, 3)
inSql	字段 IN (sql语句)	inSql("id", "select id from table where id < 3") -->id in (select id from table where id < 3)
notInSql	字段 NOT IN (sql语句)	notInSql("id", "select id from table where id < 3") -->age not in (select id from table where id < 3)
groupBy	分组: GROUP BY 字段, ...	例: groupBy("id", "name")-->group by id,name
orderByAsc	排序: ORDER BY 字段, ... ASC	例: orderByAsc("id", "name")-->order by id ASC,name ASC
orderByDesc	排序: ORDER BY 字段, ... DESC	例: orderByDesc("id", "name")-->order by id DESC,name DESC
orderBy	排序: ORDER BY 字段, ...	例: orderBy(true, true, "id", "name") -->order by id ASC,name ASC
having	HAVING (sql语句)	having("sum(age) > {0}", 11)-->having sum(age) > 11
or	拼接 OR	注意事项: 主动调用or表示紧接着下一个方法不是用and连接! (不调用or则默认为使用and连接) 例: eq("id", 1).or().eq("name", "老王")-->id = 1 or name = '老王'
and	AND 嵌套	例: and(i -> i.eq("name", "李白").ne("status", "活着")) -->and (name = '李白' and status <> '活着')
apply	拼接 sql	注意事项: 该方法可用于数据库函数 动态入参的params对应前面sqlHaving内部的 {index} 部分. 这样是不会有sql注入风险的, 反之会有! 例: apply("date_format(dateColumn, '%Y-%m-%d') = {0}", "2008-08-08")-->date_format(dateColumn, '%Y-%m-%d') = '2008-08-08'
last	无视优化规则直接拼接到 sql 的最后	注意事项: 只能调用一次, 多次调用以最后一次为准 有sql注入的风险, 请谨慎使用 例: last("limit 1")
exists	拼接 EXISTS (sql语句)	例: exists("select id from table where age = 1") -->exists (select id from table where age = 1)
notExists	拼接 NOT EXISTS (sql语句)	例: notExists("select id from table where age = 1") -->not exists (select id from table where age = 1)
nested	正常嵌套 不带 AND 或者 OR	正常嵌套 不带 AND 或者 OR 例: nested(i -> i.eq("name", "李白").ne("status", "活着")) -->(name = '李白' and status <> '活着') blog.csdn.net/m0_37083429

(3)AbstractLambdaWrapper： Lambda 语法使用 Wrapper统一处理解析 lambda 获取 column。

(4)LambdaQueryWrapper： 看名称也能明白就是用于Lambda语法使用的查询Wrapper

(5)LambdaUpdateWrapper： Lambda 更新封装Wrapper

(6)QueryWrapper： Entity 对象封装操作类，不是用lambda语法,自身的内部属性 entity 也用于生成 where 条件
该类的重要方法:

select方法

```

1 select(String... sqlSelect)
2 select(Predicate<TableFieldInfo> predicate)
3 select(Class<T> entityClass, Predicate<TableFieldInfo> predicate)
4 /*
5 例: select("id", "name", "age")
6 例: select(i -> i.getProperty().startsWith("test"))
7 */

```

(7)UpdateWrapper： Update 条件封装，用于Entity对象更新操作。

该类主要有以下三个重要的方法:

set方法

```

1 set(String column, Object val)
2 set(boolean condition, String column, Object val)
3 /*
4 SQL SET 字段
5 例: set("name", "老李头")
6 例: set("name", "")---->数据库字段值变为空字符串
7 例: set("name", null)---->数据库字段值变为null
8 说明:boolean condition为控制该字段是否拼接到最终的sql语句中
9 */

```

setSql方法

```

1 setSql(String sql)
2 /*
3 设置 SET 部分 SQL
4 例: setSql("name = '老李头'")
5 */

```

3.2.带条件的crud实验

(1)带条件的查询

```

1
2 // 根据 entity 条件, 查询一条记录
3 T selectOne(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);
4
5 // 根据 entity 条件, 查询全部记录
6 List<T> selectList(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);
7
8 // 根据 Wrapper 条件, 查询全部记录
9 List<Map<String, Object>> selectMaps(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);
10 // 根据 Wrapper 条件, 查询全部记录。注意: 只返回第一个字段的值
11 List<Object> selectObjs(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);
12
13 // 根据 entity 条件, 查询全部记录 (并翻页)
14 IPage<T> selectPage(IPage<T> page, @Param(Constants.WRAPPER) Wrapper<T> queryWrapper);
15 // 根据 Wrapper 条件, 查询全部记录 (并翻页)
16 IPage<Map<String, Object>> selectMapsPage(IPage<T> page, @Param(Constants.WRAPPER) Wrapper<T> queryWrapper);
17 // 根据 Wrapper 条件, 查询总记录数
18 Integer selectCount(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);

```

(2)带条件的更新

```

1 @Test
2 void update() {
3     UpdateWrapper<Employee> updateWrapper=new UpdateWrapper<Employee>();
4     updateWrapper.eq("last_name", "lili").eq("age", 18).set("id", 100).set(false, "email", "000@qq.com");
5     empolyeeMapper.update(employee, updateWrapper);
6
7 }
8 }

```

```

: ==> Preparing: UPDATE tbl_employee SET id=? WHERE (last_name = ? AND age = ?)
: ==> Parameters: 100(Integer), lili(String), 18(Integer)
: <== Updates: 1

```

其中set("id", 100).set(false, "email", "000@qq.com");中email属性设置为false,从执行的sql可以看出,设置为false不会拼接到最终的执行sql中

(3)带条件的删除

```

1 // 根据 entity 条件, 删除记录

```



```

2 int delete(@Param(Constants.WRAPPER) Wrapper<T> wrapper);
3
4 // 根据 columnMap 条件，删除记录
5 int deleteByMap(@Param(Constants.COLUMN_MAP) Map<String, Object> columnMap);

```

4.扩展

全局ID生成策略

在全局配置文件中：就不需要再每个Pojo主键上配置了

```

1
2 mybatis-plus:
3   global-config:
4     db-config:
5       id-type: auto

```

逻辑删除

物理删除：在删除的时候直接将数据从数据库干掉DELTE

逻辑删除：从逻辑层面控制删除，通常会在表里添加一个逻辑删除的字段比如 enabled 、 is_delete ，数据默认是有效的（值为1），当用户删除时将数据修改UPDATE 0，在查询的时候就只查where enabled=1。

1. 需要添加逻辑删除的字段
2. 局部单表逻辑删除，需要在对应的pojo类加入对应的逻辑删除标识字段

```

1 @TableLogic // 代表逻辑删除
2 private Integer flag;
3
4 public Integer getFlag() {
5   return flag;
6 }

```

全局逻辑删除配置，如果进行了全局逻辑删除配置并且指定了，就可以不用在每个pojo类中配置了@TableLogic

```

1 mybatis-plus:
2   global-config:
3     db-config:
4       logic-delete-field: flag # 全局逻辑删除的实体字段名(since 3.3.0,配置后可以忽略不配置步骤2)
5       logic-delete-value: 1 # 逻辑已删除值(默认为 1)
6       logic-not-delete-value: 0 # 逻辑未删除值(默认为 0)

```

名	类型	长度	小数点	不是 null	键	注释
id	int	11	0	<input checked="" type="checkbox"/>		
last_name	varchar	50	0	<input type="checkbox"/>		
email	varchar	50	0	<input type="checkbox"/>		
gender	char	1	0	<input type="checkbox"/>		
age	int	11	0	<input type="checkbox"/>		
flag	int	1	0	<input type="checkbox"/>		

默认:

☐ 自动递增

☐ 无符号

☐ 填充零

当执行删除，将会把逻辑删除字段进行修改

```

@Test
void logicDelete() {
    employeeService.removeById(1);
}

```

```
UPDATE tbl_employee SET flag=0 WHERE id=? AND flag=1
```

当执行查询，会自动查询有效数据 where flag=1

```

@Test
void logicList() {
    employeeService.list();
}

```

```
SELECT id,last_name,email,gender,age,flag FROM tbl_employee WHERE flag=1
```

执行 SQL 分析打印

```

1 <dependency>
2   <groupId>p6spy</groupId>
3   <artifactId>p6spy</artifactId>
4   <version>最新版本</version>
5 </dependency>

```

```

url: jdbc:p6spy:mysql://localhost:3306/mybatisplus?characterEncoding=utf8&useSSL=false&
driver-class-name: com.p6spy.engine.spy.P6SpyDriver #com.mysql.cj.jdbc.Driver

```

添加p6spy : spy.properties

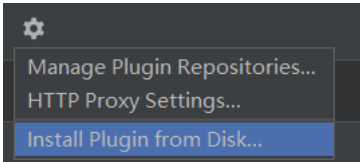
```


1 #3.2.1以上使用
2 modulelist=com.baomidou.mybatisplus.extension.p6spy.MybatisPlusLogFactory,com.p6spy.engine.outage.P6OutageFactory
3 #3.2.1以下使用或者不配置
4 #modulelist=com.p6spy.engine.logging.P6LogFactory,com.p6spy.engine.outage.P6OutageFactory
5 # 自定义日志打印
6 logMessageFormat=com.baomidou.mybatisplus.extension.p6spy.P6SpyLogger
7 #日志输出到控制台
8 appender=com.baomidou.mybatisplus.extension.p6spy.StdoutLogger
9 # 使用日志系统记录 sql
10 #appender=com.p6spy.engine.spy.appender.Slf4JLogger
11 # 设置 p6spy driver 代理
12 deregisterdrivers=true
13 # 取消JDBC URL前缀
14 useprefix=true
15 # 配置记录 Log 例外,可去掉的结果集有error,info,batch,debug,statement,commit,rollback,result,resultset.
16 excludecategories=info,debug,result,commit,resultset
17 # 日期格式
18 dateformat=yyyy-MM-dd HH:mm:ss
19 # 实际驱动可多个
20 #driverlist=org.h2.Driver
21 # 是否开启慢SQL记录
22 outagedetection=true
23 # 慢SQL记录标准 2 秒

```

```
24 outagedetectioninterval=2
```

sql 日志美化插件:



 plugin.intelij.assistant.mybatislog-2.0.1.jar 2021-

```
-- 2021-02-20 21:10:52.256 DEBUG 8716 --- [
SELECT
    id,
    last_name AS lastName,
    email,
    gender,
    age
FROM
    tbl_employee
WHERE
    gender = 0
-- 2021-02-20 21:10:52.256 DEBUG 8716 --- [
SELECT
    id,
    last_name AS lastName,
    email,
```

数据安全保护

防止删库跑路

1.得到16位随机密钥

```
1 @Test
2 void test(){// 生成 16 位随机 AES 密钥
3     String randomKey = AES.generateRandomKey();
4     System.out.println(randomKey);
5 }
6 da12166c7db8a58f
```

2.根据密钥加密 数据库连接信息

```
1 @Test
2 void test(){
3
4     String url = AES.encrypt("jdbc:mysql://localhost:3306/mybatisplus?characterEncoding=utf8&useSSL=false
&serverTimezone=UTC&" , "da12166c7db8a58f");
5     String uname = AES.encrypt("root" , "da12166c7db8a58f");
6     String pwd = AES.encrypt("123456" , "da12166c7db8a58f");
7
8     System.out.println(url);
9     System.out.println(uname);
10    System.out.println(pwd);
```

```
11 }
```

```
qIh0E63gBfvpFbz2tXDyWdN2kFpD+apc9JaRYosGY5sKL3zyNwa1K30fGo27p8AM8BL011HGFwpfdELaf79NIxm8kfOMhUd0FLNy7g85BTCrEzbYEHqp3OCJ49ihj1Q6UbkRfixFdVg==yp192Xv01C0jq67MeCv1Ig==
```

3.修改配置文件 注意要mpw:开头

```
1 username: mpw:0Cj49ihj1Q6UbkRfixFdVg==
2 password: mpw:yp192Xv01C0jq67MeCv1Ig==
3 url: mpw:nIh0E63gBfvpFbz2tXDyWdN2kFpD+apc9JaRYosGY5sKL3zyNwa1K30fGo27p8AM8BL011HGFwpfdELaf79NIxm8kfOMhUd0FLNy7g85BTCrEzbYEHqp3THf7K0z80Ka
```

4.在部署的时候需要解密

```
1 java -jar xxxx.jar --mpw.key=你的16位随机密钥, 越少人知道越好
```

乐观锁插件使用

第一：什么是乐观锁

- 悲观锁：悲观锁，正如其名，具有强烈的独占和排他特性。它指的是对数据被外界(包括本系统当前的其他事务，以及来自外部系统的事务处理)修改持保守态度。因此，在整个数据处理过程中，将数据处于锁定状态。假设功能并发量非常大，就需要使用`synchronized`来处理高并发下产生线程不安全问题，会使其他线程进行挂起等待从而影响系统吞吐量

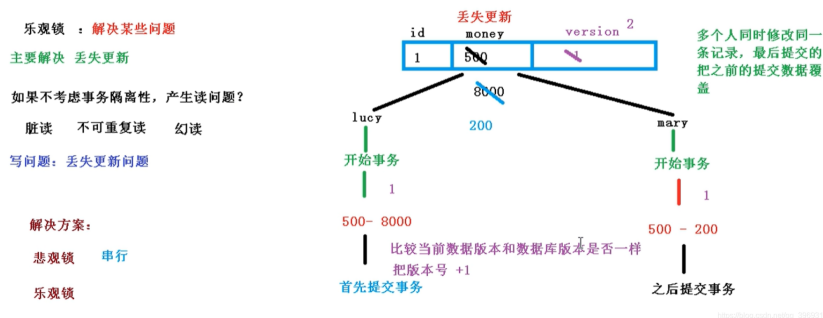
- 乐观锁：乐观锁是相对悲观锁而言的，乐观锁假设数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则返回给用户错误的信息，让用户决定如何去做。乐观锁适用于读操作多的场景，这样可以提高程序的吞吐量。假设功能产生并发几率极少，采用乐观锁版本机制对比，如果有冲突 返回给用户错误的信息

第二：为什么需要锁(并发控制)

在多用户环境中，在同一时间可能会有多个用户更新相同的记录，这会产生冲突。这就是著名的并发性问题

- 丢失更新：一个事务的更新覆盖了其它事务的更新结果，就是所谓的更新丢失。例如：用户1把值从500改为8000，用户B把值从500改为200，则多人同时提交同一条记录，后提交的把之前的提交数据覆盖。
- 脏读：当一个事务读取其它完成一半事务的记录时，就会发生脏读。例如：用户A,B看到的值都是500，用户B把值改为200，用户A读到的值仍为500。

针对一种问题的解决方案，为解决问题而生的。解决什么问题呢？主要是解决**丢失更新**问题如下图理解



为了解决这些并发带来的问题。我们需要引入并发控制机制。

第三：乐观锁使用MyBatisPlus的解决方式

由于锁这个字眼我们需要在数据库加个字段“version”来控制版本
在类中加个属性

```
1 @Version //这就是控制版本的
2 @TableField(fill = FieldFill.INSERT) //这个方便在添加的时候设置版本初始为1
3 private Integer version; //版本的字段
```

下面这个也是MyBatisPlus的一个插件 只需要实现MetaObjectHandler就可以了

```
1 @Component
2 public class MyMetaObjectHandler implements MetaObjectHandler {
```

```

3  @Override
4  public void insertFill(MetaObject metaObject) {
5  //这里的“version”就是指定的字段，设置初始值为1，之后每修改一次+1
6  this.setFieldValByName("version",1,metaObject);
7  }
8  @Override
9  public void updateFill(MetaObject metaObject) {
10
11  }
12 }

```

在MyBatis中存在一个乐观锁插件: OptimisticLockerInnerInterceptor

```

1  @Configuration
2  @MapperScan("com.lzz.mapper")
3  public class MyConfig {
4  //乐观锁插件
5  @Bean
6  public OptimisticLockerInterceptor optimisticLockerInterceptor(){
7  return new OptimisticLockerInterceptor();
8  }
9  }

```

接下来在做增加数据的时候，调用insert添加方法就可以了。

修改的时候呢，我们需要先查人后再做修改，因为我们为了防止问题的发生，需要先去查询版本号比对才进行后续操作！！

5、代码生成器

```

1  package com.tulingxueyuan;
2
3  import com.baomidou.mybatisplus.core.exceptions.MybatisPlusException;
4  import com.baomidou.mybatisplus.core.toolkit.StringPool;
5  import com.baomidou.mybatisplus.core.toolkit.StringUtils;
6  import com.baomidou.mybatisplus.generator.AutoGenerator;
7  import com.baomidou.mybatisplus.generator.InjectionConfig;
8  import com.baomidou.mybatisplus.generator.config.*;
9  import com.baomidou.mybatisplus.generator.config.po.LikeTable;
10 import com.baomidou.mybatisplus.generator.config.po.TableInfo;
11 import com.baomidou.mybatisplus.generator.config.rules.DateType;
12 import com.baomidou.mybatisplus.generator.config.rules.NamingStrategy;
13
14 import java.util.ArrayList;
15 import java.util.List;
16 import java.util.Scanner;
17
18 /**
19  * @Author 徐庶 QQ:1092002729
20  * @Slogan 致敬大师，致敬未来的你
21  *
22  * pms_product
23  */
24 public class GeneratorApp {
25
26  /**

```

```
27 * <p>
28 * 读取控制台内容
29 * </p>
30 */
31 public static String scanner(String tip) {
32     Scanner scanner = new Scanner(System.in);
33     StringBuilder help = new StringBuilder();
34     help.append("请输入" + tip + ": ");
35     System.out.println(help.toString());
36     // 判断用户是否输入
37     if (scanner.hasNext()) {
38         // 拿到输入内容
39         String ipt = scanner.next();
40         if (StringUtils.isNotBlank(ipt)) {
41             return ipt;
42         }
43     }
44     throw new MybatisPlusException("请输入正确的" + tip + "!!");
45 }
46
47 public static void main(String[] args) {
48
49     String moduleName = scanner("模块名");
50     String tableName = scanner("表名（多个用，号分隔，或者按前缀（pms*））");
51     String prefixName = scanner("需要替换的表前缀");
52
53
54     // 代码生成器
55     AutoGenerator mpg = new AutoGenerator();
56
57     // 全局配置
58     GlobalConfig gc = new GlobalConfig();
59     // 获得当前项目的路径
60     String projectPath = System.getProperty("user.dir")+"/05_generator";
61     // 设置生成路径
62     gc.setOutputDir(projectPath + "/src/main/java");
63     // 作者
64     gc.setAuthor("xushu");
65     // 代码生成是不是要打开所在文件夹
66     gc.setOpen(false);
67     // 生成Swagger2注解
68     gc.setSwagger2(true);
69     // 会在mapper.xml 生成一个基础的<ResultMap> 映射所有的字段
70     gc.setBaseResultMap(true);
71     // 同文件生成覆盖
72     gc.setFileOverride(true);
73     //gc.setDateType(DateType.ONLY_DATE)
74     // 实体名：直接用表名 %s=表名
75     gc.setEntityName("%s");
76     // mapper接口名
77     gc.setMapperName("%sMapper");
78     // mapper.xml 文件名
79     gc.setXmlName("%sMapper");
```

```
80 // 业务逻辑类接口名
81 gc.setServiceName("%sService");
82 // 业务逻辑类实现类名
83 gc.setServiceName("%sImplService");
84 // 将全局配置设置到AutoGenerator
85 mpg.setGlobalConfig(gc);
86
87
88
89 // 数据源配置
90 DataSourceConfig dsc = new DataSourceConfig();
91 dsc.setUrl("jdbc:mysql://localhost:3306/tuling_mall?characterEncoding=utf8&useSSL=false&serverTimezone=UTC&");
92 dsc.setDriverName("com.mysql.cj.jdbc.Driver");
93 dsc.setUsername("root");
94 dsc.setPassword("123456");
95 mpg.setDataSource(dsc);
96
97 // 包配置
98 PackageConfig pc = new PackageConfig();
99 // 模块名
100 pc.setModuleName(moduleName);
101 // 包名
102 pc.setParent("com.tulingxueyuan");
103 // 完整的报名: com.tulingxueyuan.pms
104 mpg.setPackageInfo(pc);
105
106
107
108 // 自定义配置
109 InjectionConfig cfg = new InjectionConfig() {
110     @Override
111     public void initMap() {
112         // to do nothing
113     }
114 };
115
116 // 如果模板引擎是 velocity
117 String templatePath = "/templates/mapper.xml.vm";
118 // 自定义输出配置
119 List<FileOutConfig> focList = new ArrayList<>();
120 // 自定义配置会被优先输出
121 focList.add(new FileOutConfig(templatePath) {
122     @Override
123     public String outputFile(TableInfo tableInfo) {
124         // 自定义输出文件名 , 如果你 Entity 设置了前后缀、此处注意 xml 的名称会跟着发生变化!!
125         return projectPath + "/src/main/resources/mapper/" + pc.getModuleName()
126             + "/" + tableInfo.getEntityName() + "Mapper" + StringPool.DOT_XML;
127     }
128 });
129
130 cfg.setFileOutConfigList(focList);
131 mpg.setCfg(cfg);
```

```
132
133 // 配置模板
134 TemplateConfig templateConfig = new TemplateConfig();
135
136 // 把已有的xml生成置空
137 templateConfig.setXml(null);
138 mpg.setTemplate(templateConfig);
139
140 // 策略配置
141 StrategyConfig strategy = new StrategyConfig();
142 // 表名的生成策略: 下划线转驼峰 pms_product -- PmsProduct
143 strategy.setNaming(NamingStrategy.underline_to_camel);
144 // 列名的生成策略: 下划线转驼峰 last_name -- lastName
145 strategy.setColumnNaming(NamingStrategy.underline_to_camel);
146 //strategy.setSuperEntityClass("你自己的父类实体,没有就不用设置!");
147 //strategy.setEntityLombokModel(true);
148 // 在controller类上是否生成@RestController
149 strategy.setRestControllerStyle(true);
150 // 公共父类
151 //strategy.setSuperControllerClass("你自己的父类控制器,没有就不用设置!");
152
153 if(tableName.indexOf('*')>0){
154 // 按前缀生成表
155 strategy.setLikeTable(new LikeTable(tableName.replace('*', '_')));
156 }
157 else{
158 // 要生成的表名 多个用逗号分隔
159 strategy.setInclude(tableName);
160 }
161 // 设置表替换前缀
162 strategy.setTablePrefix(prefixName);
163 // 驼峰转连字符 比如 pms_product --> controller @RequestMapping("/pms/pmsProduct")
164 //strategy.setControllerMappingHyphenStyle(true);
165 mpg.setStrategy(strategy);
166
167 // 进行生成
168 mpg.execute();
169
170 }
171 }
```