

学习目标

- ☐ 能够基于MyBatisPlus完成标准Dao开发
- ☐ 能够掌握MyBatisPlus的条件查询
- ☐ 能够掌握MyBatisPlus的字段映射与表名映射
- ☐ 能够掌握id生成策略控制
- ☐ 能够理解代码生成器的相关配置

一、MyBatisPlus简介

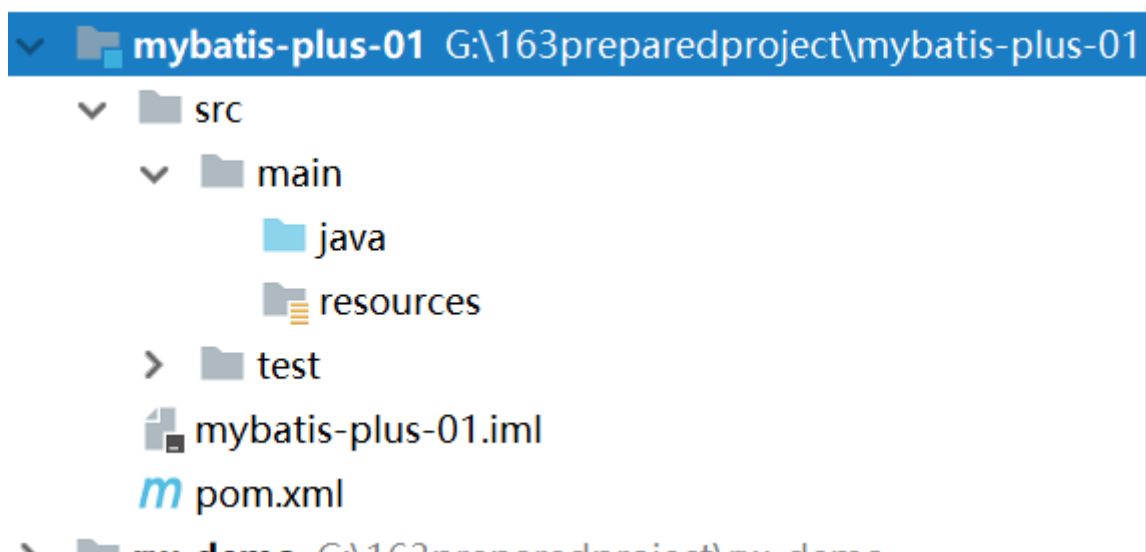
1. 入门案例

问题导入

MyBatisPlus环境搭建的步骤？

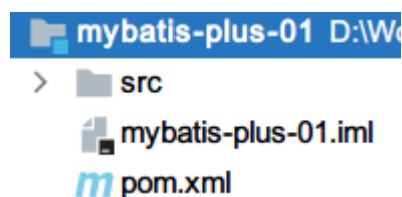
SpringBoot整合MyBatisPlus入门程序

①：创建新模块mybatis-plus-01



③：添加相关的起步依赖

多余的文件和依赖配置都可以删除了



pom.xml文件如下：

```
<parent>
  <groupId>org.springframework.boot</groupId>
```

```

        <artifactId>spring-boot-dependencies</artifactId>
        <version>2.5.6</version>
    </parent>

    <dependencies>
        <!--springboot-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>

        <!-- spring整合test -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
        </dependency>

        <!-- mybatis-plus的驱动包 -->
        <dependency>
            <groupId>com.baomidou</groupId>
            <artifactId>mybatis-plus-boot-starter</artifactId>
            <version>3.4.2</version>
        </dependency>

        <!-- 连接池 -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid-spring-boot-starter</artifactId>
            <version>1.1.23</version>
        </dependency>

        <!-- mysql 要选择正确版本的驱动-->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.47</version>
            <scope>runtime</scope>
        </dependency>

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

```

注意事项：如果使用Druid数据源，需要导入对应坐标

④：制作实体类与表结构

(类名与表名对应，属性名与字段名对应)

SQL脚本

```

CREATE DATABASE IF NOT EXISTS mybatisplus_db CHARACTER SET utf8;
USE mybatisplus_db;

CREATE TABLE `user` (
    id BIGINT(20) PRIMARY KEY AUTO_INCREMENT,

```

```

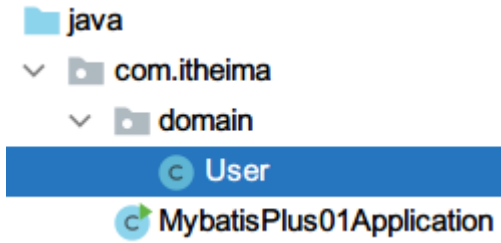
        `name` VARCHAR(32) NOT NULL,
        gender CHAR(1),
        `password` VARCHAR(32) NOT NULL,
        age INT(3) NOT NULL ,
        tel VARCHAR(32) NOT NULL
    );
INSERT INTO `user` VALUES(NULL, 'Rose', '女', '123456', 12, '12345678910');
INSERT INTO `user` VALUES(NULL, 'Jack', '男', '123456', 8, '12345678910');
INSERT INTO `user` VALUES(NULL, 'Jerry', '男', '123456', 15, '12345678910');
INSERT INTO `user` VALUES(NULL, 'NewBoy', '男', '123456', 19, '12345678910');
INSERT INTO `user` VALUES(NULL, 'Kate', '女', '123456', 28, '12345678910');
INSERT INTO `user` VALUES(NULL, '张晓', '女', '123456', 22, '12345678910');
INSERT INTO `user` VALUES(NULL, '张大炮', '男', '123456', 16, '12345678910');

SELECT * FROM `user`;

```



在domain目录下创建实体类



```

package com.itheima.domain;

import lombok.Data;

@Data
public class User {
    private Long id;
    private String name;
    private String gender;
    private String password;
    private Integer age;
    private String tel;
}

```

⑤: 设置Jdbc参数 (application.yml)

```
# 数据源的配置
spring:
  datasource:
    username: root
    password: root
    driver-class-name: com.mysql.jdbc.Driver
    type: com.alibaba.druid.pool.DruidDataSource
    url: jdbc:mysql:///mybatisplus_db

# 显示SQL语句
mybatis-plus:
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
```

⑥：定义数据接口，继承BaseMapper

接口上要添加@Mapper注解，==注意：启动类记得添加@MapperScan注解扫描dao包。==

```
package com.itheima.dao;

import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.itheima.domain.User;

/*
1. Dao接口以后只需要继承BaseMapper接口，那么单表的增删查改全部都帮你实现了
2. 继承BaseMapper接口的时候一定要指定操作哪个实体类？
alt+7 查看到该类的所有成员
*/
public interface UserDao extends BaseMapper<User>{
}
```

⑦：测试类中注入dao接口，测试功能

1. 建议把测试类的包移入com.itheima.test包中，修改类名Demo1MybatisPlusTest
2. 测试类上添加@SpringBootTest注解
3. 注入UserMapper
4. selectList查询所有用户，参数为null

```
package com.itheima.test;

import com.itheima.dao.UserDao;
import com.itheima.domain.User;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.List;

@SpringBootTest
public class UserTest {

    @Autowired(required = false)
    private UserDao userDao;
```

```

//根据id查找
@Test
public void testFindById(){
    User user = userDao.selectById(4);
    System.out.println("用户: "+user);
}

@Test
public void testFindAll(){
    List<User> userList = userDao.selectList(null);
    System.out.println("用户: "+userList);
}

}

```

小结

- 使用mybatisplus的步骤
 - 导入依赖
 - 在配置文件上配置对应的参数 log-impl
 - 编写实体类
 - 编写Mapper接口，需要继承BaseMapper，并且BaseMapper必须要指定操作的实体类
实体类就是对应一张表的。

2. MyBatisPlus概述

问题导入

通过入门案例制作，MyBatisPlus的优点有哪些？

1. MyBatis介绍

- [MyBatis-Plus \(opens new window\)](#) (简称 MP) 是一个 [MyBatis \(opens new window\)](#) 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/>

2. MyBatisPlus特性

- 无侵入：只做增强不做改变，不会对现有工程产生影响
- 强大的 CRUD 操作：内置通用 Mapper，少量配置即可实现单表CRUD 操作(如果只做单表增删查改不需要你写任何的sql)
- 支持 Lambda：编写查询条件无需担心字段写错
- 支持主键自动生成
- 内置分页插件
-

二、标准数据层开发

1. MyBatisPlus的CRUD操作

功能	自定义接口	MP接口
新增	<small>之前我们自定义的dao/mapper接口方法</small> <code>boolean save(T t)</code>	<code>int insert(T t)</code>
删除	<code>boolean delete(int id)</code>	<code>int deleteById(Serializable id)</code>
修改	<code>boolean update(T t)</code>	<code>int updateById(T t)</code>
根据id查询	<code>T getById(int id)</code>	<code>T selectById(Serializable id)</code>
查询全部	<code>List<T> getAll()</code>	<code>List<T> selectList()</code>
分页查询	<code>PageInfo<T> getAll(int page, int size)</code>	<code>IPage<T> selectPage(IPage<T> page)</code>
按条件查询	<code>List<T> getAll(Condition condition)</code>	<code>IPage<T> selectPage(Wrapper<T> queryWrapper)</code>

```
package com.itheima.test;

import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.itheima.domain.User;
import com.itheima.mapper.UserMapper;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.List;

@SpringBootTest
public class CRUDTest {

    @Autowired
    private UserMapper userMapper;

    @Test
    public void testFindById() {
        User user = userMapper.selectById(1);
        System.out.println("用户: " + user);
    }

    @Test
    public void testFindAll() {
        List<User> userList = userMapper.selectList(null);
        System.out.println("用户: " + userList);
    }

    // 增加
    @Test
    public void insertTest() {
        User user = new User();
        user.setName("张在晶");
        user.setGender("男");
        user.setPassword("123456");
    }
}
```

```

        user.setAge(38);
        user.setTel("18000110011");
        userMapper.insert(user);
    }

    // 删除
    @Test
    public void deleteTest() {
        userMapper.deleteById(7);    //delete from 表 where id = 11;
    }

    // 修改, 修改该实体类的所有非空字段
    @Test
    public void updateById() {
        User user = new User();
        user.setGender("女");
        user.setId(8);
        userMapper.updateById(user);    // update user set xx=xx,xx=xx where id =
xx;
    }

    //根据条件查询, 查找出年龄大于18的
    @Test
    public void testFindByCondition() {
        //QueryWrapper 就是一个条件对象
        QueryWrapper<User> queryWrapper = new QueryWrapper<>();
        //查找出年龄大于18的
        queryWrapper.gt("age", 18);
        List<User> userList = userMapper.selectList(queryWrapper);
        System.out.println("用户: " + userList);
    }
}

```

2. MyBatisPlus分页功能

问题导入

思考一下Mybatis分页插件是如何用的?

1 分页功能接口

功能	自定义接口	MP接口
新增	<code>boolean save(T t)</code>	<code>int insert(T t)</code>
删除	<code>boolean delete(int id)</code>	<code>int deleteById(Serializable id)</code>
修改	<code>boolean update(T t)</code>	<code>int updateById(T t)</code>
根据id查询	<code>T getById(int id)</code>	<code>T selectById(Serializable id)</code>
查询全部	<code>List<T> getAll()</code>	<code>List<T> selectList()</code>
分页查询	<code>PageInfo<T> getAll(int page, int size)</code>	<code>IPage<T> selectPage(IPage<T> page)</code>
按条件查询	<code>List<T> getAll(Condition condition)</code>	<code>IPage<T> selectPage(Wrapper<T> queryWrapper)</code>

2 MyBatisPlus分页使用

①：设置分页拦截器作为Spring管理的bean

1. 在config包下创建一个配置类：MybatisPlusConfig
2. 在类上添加@Configuration
3. 编写方法
 1. 方法上使用@Bean注解：添加 MybatisPlusInterceptor 对象到容器中
 2. 创建MybatisPlusInterceptor拦截器对象
 3. 添加内部分页拦截器：创建PaginationInnerInterceptor

```
package com.itheima.config;

import com.baomidou.mybatisplus.extension.plugins.MybatisPlusInterceptor;
import com.baomidou.mybatisplus.extension.plugins.inner.PaginationInnerInterceptor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MybatisConfig {

    /**
     * 在你执行selectPage方法的时候，这个拦截器就会帮你拼接limit的语句，从而帮你实现分页
     */
    @Bean
    public MybatisPlusInterceptor createMybatisPlusInterceptor(){
        //1. 创建mybatis-plus拦截器
        MybatisPlusInterceptor mybatisPlusInterceptor = new
MybatisPlusInterceptor();
        //2. 给mybatis-plus拦截器添加分页拦截器
        mybatisPlusInterceptor.addInnerInterceptor(new
PaginationInnerInterceptor());
        //3, 返回拦截器
        return mybatisPlusInterceptor;
    }
}
```

②：在测试类中执行分页查询

1. 创建分页对象，前面是接口IPage，后面是实现类Page(第几页，每页大小)
2. 调用selectPage方法，传入page对象，无需接收返回值
3. 获取分页结果

```
/**
 * 分页查询
 */
@Test
public void selectPageTest(){
    //1. 创建page对象，设置当前页与页面大小
    Page<User> page = new Page<>(1,2);
```



```
        userDao.selectPage(page,null); //page对象就相当于以前我们的pageBean对象，封装
        页面的所有数据， queryWrapper分页条件
        System.out.println("当前页: "+page.getCurrent() );
        System.out.println("页面大小: "+page.getSize());
        System.out.println("总记录数: "+ page.getTotal());
        System.out.println("总页数: "+ page.getPages());
        System.out.println("当前页的数据: "+ page.getRecords());
    }
}
```

三、DQL编程控制

1. 条件查询方式

- MyBatisPlus将书写复杂的SQL查询条件进行了封装，使用编程的形式完成查询条件的组合

```
(m) selectById(Serializable): T
(m) selectBatchIds(Collection<? extends Serializable>): List<T>
(m) selectByMap(Map<String, Object>): List<T>
(m) selectOne(Wrapper<T>): T
(m) selectCount(Wrapper<T>): Integer
(m) selectList(Wrapper<T>): List<T>
(m) selectMaps(Wrapper<T>): List<Map<String, Object>>
(m) selectObjs(Wrapper<T>): List<Object>
(m) selectPage(IPage<T>, Wrapper<T>): IPage<T>
(m) selectMapsPage(IPage<T>, Wrapper<T>): IPage<Map<String, Object>>
```

1.1 条件查询

创建新的测试类：Demo2ConditionTest

```
package com.itheima.test;

import com.itheima.dao.UserDao;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
public class Demo2ConditionTest {

    @Autowired
    private UserDao userDao;
}
```

1.1.1 方式一：按条件查询

查询年龄大于18岁的用户

```
package com.itheima.test;

import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.itheima.dao.UserDao;
import com.itheima.domain.User;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.List;

@SpringBootTest
public class Demo2ConditionTest {

    @Autowired(required = false)
    private UserDao userDao;

    //    查询年龄大于18岁的用户
    @Test
    public void testAge(){
        QueryWrapper<User> queryWrapper = new QueryWrapper<>();
        queryWrapper.gt("age",18);
        List<User> userList = userDao.selectList(queryWrapper);
        System.out.println("集合列表: "+ userList);
    }

    //
}
```

1.1.2 方式二：lambda格式按条件查询（推荐）

查询年龄小于10的用户

```
//lambda格式按条件查询
@Test
public void testLambdaAge(){
    LambdaQueryWrapper<User> queryWrapper = new LambdaQueryWrapper<>();
    queryWrapper.gt(User::getAge,18);
    List<User> userList = userDao.selectList(queryWrapper);
    System.out.println("集合列表: "+ userList);
}
```

1.2 组合条件

1.2.1 并且关系 (and)

查询年龄小于30岁，而且大于10岁的用户

```
// 查询年龄小于30岁，而且大于10岁的用户
@Test
public void testAndCondition(){
    LambdaQueryWrapper<User> queryWrapper = new LambdaQueryWrapper<>();
    queryWrapper.lt(User::getAge,30);
    queryWrapper.gt(User::getAge,10);
    List<User> userList = userDao.selectList(queryWrapper);
    System.out.println("集合列表: "+ userList);
}
```

生成的SQL语句

```
SELECT id,name,gender,password,age,te1 FROM user WHERE (age < ? AND age > ?)
```

1.2.2 或者关系 (or)

查询年龄小于10岁或者大于30岁的用户

```
// 查询年龄小于10岁或者大于30岁的用户
@Test
public void testOrCondition(){
    LambdaQueryWrapper<User> queryWrapper = new LambdaQueryWrapper<>();
    queryWrapper.lt(User::getAge,10);
    queryWrapper.or();
    queryWrapper.gt(User::getAge,30);
    List<User> userList = userDao.selectList(queryWrapper);
    System.out.println("集合列表: "+ userList);
}
```

生成的SQL语句

```
SELECT id,name,gender,password,age,te1 FROM user WHERE (age < ? OR age > ?)
```

1.3 NULL值处理

问题导入

如下搜索场景，在多条件查询中，有条件的值为空应该怎么解决？



1.3.1 if语句控制条件追加

- 如果最小年龄不为空，则查询大于这个年龄的用户
- 如果最大年龄不为空，则查询小于这个年龄的用户

// 根据年龄搜索，分别最小年龄，最大年龄，名字，只要三个变量中任何一个不为空都要作为条件查询

```
@Test
public void testIfNull(){
    Integer minAge = 10;
    Integer maxAge = 30;
    String name= "J";
    //期望的sql语句: select * from user where age<30 and name like '%J%'
    LambdaQueryWrapper<User> queryWrapper = new LambdaQueryWrapper<>();
    if(minAge!=null){
        queryWrapper.gt(User::getAge,minAge);
    }

    if(maxAge!=null){
        queryWrapper.lt(User::getAge,maxAge);
    }

    if(name!=null){
        queryWrapper.like(User::getName,name);
    }
    List<User> userList = userDao.selectList(queryWrapper);
    System.out.println("用户列表: "+userList);
}
```

1.3.2 条件参数控制

// 根据年龄搜索，分别最小年龄，最大年龄，名字，只要三个变量中任何一个不为空都要作为条件查询

```
@Test
public void testIfNull2(){
    Integer minAge = 10;
    Integer maxAge = 30;
    String name= "J";
    //期望的sql语句: select * from user where age<30 and name like '%J%'
    LambdaQueryWrapper<User> queryWrapper = new LambdaQueryWrapper<>();

    queryWrapper.gt(minAge!=null,User::getAge,minAge);
```

```

        queryWrapper.lt(maxAge!=null,User::getAge,maxAge);

        queryWrapper.like(name!=null,User::getName,name);

        List<User> userList = userDao.selectList(queryWrapper);
        System.out.println("用户列表: "+userList);

    }

```

2. 查询投影-设置【查询字段、分组、分页】

创建新的测试类：

```

/**
 * 查询投影
 */
@SpringBootTest
public class Demo3ProjectionTest {

    @Autowired
    private UserMapper userMapper;
}

```

1 查询结果包含模型类中部分属性

查询所有用户，只显示id, name, age三个属性，不是全部列。

使用 `select(列名...)` 方法，查询的结果如果封装成实体类，则只有这三个属性有值，其它属性为 NULL

```

@Test
void testSameColumn() {
    LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<>();
    //查询所有用户，只显示id, name, age三个属性，不是全部列
    wrapper.select(User::getId, User::getName, User::getAge);
    List<User> userList = userMapper.selectList(wrapper);
    System.out.println(userList);
}

```

SQL语句

```
SELECT id,name,age FROM user
```

2 查询结果包含模型类中未定义的属性

如果查询结果包含模型类中未定义的属性，则将每个元素封装成Map对象。

需求：按性别进行分组，统计每组的人数。只显示统计的人数和性别这两个字段

使用QueryWrapper包装对象的select方法

```
/**
 * 分组统计男女人数
 */
@Test
public void testGroup(){
    QueryWrapper<User> queryWrapper = new QueryWrapper<>();
    queryWrapper.select("gender,count(*)"); //select方法里面编写的内容和select
    语句后面的内容是一样 select gender count(*) from user group by gender;
    queryWrapper.groupBy("gender");

    //如果查询的结果字段和实体类不对应，我们可以使用
    List<Map<String, Object>> maps = userDao.selectMaps(queryWrapper);
    System.out.println("集合列表: "+ maps);
}
```

3. 查询条件

问题导入

多条件查询有哪些组合？

- 范围匹配 (>、=、between)
- 模糊匹配 (like)
- 空判定 (null)
- 包含性匹配 (in)
- 分组 (group)
- 排序 (order)
-

3.1 查询条件

- 购物设定价格区间、户籍设定年龄区间 (le ge匹配 或 between匹配)

```
LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<User>();
//范围查询 lt le gt ge eq between
wrapper.between(User::getAge, 10, 30);
List<User> userList = userMapper.selectList(wrapper);
System.out.println(userList);
```

- 查信息，搜索新闻（非全文检索版：like匹配）

```
LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<User>();
//模糊匹配like, %在右边
wrapper.likeRight(User::getName, "J");
List<User> userList = userMapper.selectList(wrapper);
System.out.println(userList);
```

- 统计报表（分组查询聚合函数）

```

QueryWrapper<User> qw = new QueryWrapper<User>();
qw.select("gender", "count(*) as nums");
qw.groupBy("gender");
List<Map<String, Object>> maps = userMapper.selectMaps(qw);
System.out.println(maps);

```

3.2 排序和limit

题目：显示年龄最大的5个用户

- 说明：
 - ①：提示：对年龄进行降序排序
 - ②：仅获取前5条数据（提示：使用分页功能控制数据显示数量）
- last()方法的说明：

无视优化规则直接拼接到 sql 的最后(有sql注入的风险,请谨慎使用)，注意只能调用一次,多次调用以最后一次为准

```

/**
 * 需求： 根据年龄排序降序，取前三位
 *
 */
@Test
public void testorder(){
    LambdaQueryWrapper<User> queryWrapper = new LambdaQueryWrapper<>();
    queryWrapper.orderByDesc(User::getAge).last("limit 3");
    List<User> userList = userDao.selectList(queryWrapper);
    System.out.println("用户列表: "+userList);
}

```

生成的SQL

```

SELECT id,name,gender,password,age,te1 FROM user ORDER BY age DESC limit 5

```

4. 字段映射与表名映射问题导入

思考表的字段和实体类的属性不对应，查询会怎么样？

4.1 问题一：表字段与编码属性设计不同步

- 在模型类属性上方，使用@TableField属性注解，通过==value==属性，设置当前属性对应的数据库表中的字段关系。

```

-- 修改表的列名
ALTER TABLE `user` CHANGE `password` `pwd` VARCHAR(20);

```

再次查询出现异常，报不知道的列password

```
com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: Unknown column
'password' in 'field list'
```

解决方法:

```
CREATE TABLE `user` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `name` varchar(32),
  `pwd` varchar(32),
  `age` int(3),
  `tel` varchar(32),
  PRIMARY KEY (`id`)
)
```

```
public class User {
    private Long id;
    private String name;
    @TableField(value="pwd")
    private String password;
    private Integer age;
    private String tel;
}
```

我改

生成的SQL语句, 自动给pwd这一列定义了别名为password

```
SELECT id,name,gender,pwd AS password,age,tel FROM user
```

4.2 问题二: 编码中添加了数据库中未定义的属性

在User实体类中添加新的属性 `Integer online`

查询报错:

```
Cause: com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: Unknown column
'online' in 'field list'
```

解决方法:

- 在模型类属性上方, 使用`@TableField`注解, 通过`==exist==`属性, 设置属性在数据库表字段中是否存在, 默认为true。此属性无法与value同时使用。

```
CREATE TABLE `user` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `name` varchar(32),
  `pwd` varchar(32),
  `age` int(3),
  `tel` varchar(32),
  PRIMARY KEY (`id`)
)
```

```
public class User {
    private Long id;
    private String name;
    @TableField(value="pwd")
    private String password;
    private Integer age;
    private String tel;
    @TableField(exist = false)
    private Integer online;
}
```

该属性在数据库字段中不存在

再次查询结果, 没有报错, 但online的属性值为空

```
User(id=4, name=NewBoy, gender=男, password=123456, age=19, tel=12345678910,
online=null)
```

4.3 问题三: 某些字段和属性不参与查询

需求: password这个字段不查询

- 在模型类属性上方，使用@**TableField**注解，通过**==select==**属性：设置该属性是否参与查询。此属性与select()映射配置不冲突。

```
SELECT id, name, pwd, age, tel, speciality FROM user
```

```
CREATE TABLE `user` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `name` varchar(32),
  `pwd` varchar(32),
  `age` int(3),
  `tel` varchar(32),
  PRIMARY KEY (`id`)
)
```

```
public class User {
    private Long id;
    private String name;
    @TableField(value="pwd", select = false)
    private String password;
    private Integer age;
    private String tel;
    @TableField(exist = false)
    private Integer online;
}
```

查询的SQL语句中不包含pwd字段

```
SELECT id,name,gender,age,tel FROM user WHERE id=?
```

实体类的password属性中没有值

```
User(id=4, name=NewBoy, gender=男, password=null, age=19, tel=12345678910,
online=null)
```

4.4 问题四：表名与实体类名不同

修改表名：

```
-- 修改表的名字
ALTER TABLE `user` RENAME TO tbl_user;
```

运行出现异常：

```
Cause: com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: Table
'mybatisplus_db.user' doesn't exist
```

解决方法：

- 在**模型类**上方，使用@**TableName**注解，通过**==value==**属性，设置当前类对应的数据库表名称。

```
CREATE TABLE `tbl_user` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `name` varchar(32),
  `pwd` varchar(32),
  `age` int(3),
  `tel` varchar(32),
  PRIMARY KEY (`id`)
)
```

```
@TableName("tbl_user")
public class User {
    private Long id;
    private String name;
    @TableField(value="pwd",select = false)
    private String password;
    private Integer age;
    private String tel;
    @TableField(exist = false)
    private Integer online;
}
```

模型类对应的表名称

```
@Data
@TableName("tbl_user")
public class User {
}
}
```

查询生成的SQL语句

```
SELECT id,name,gender,age,tel FROM tbl_user WHERE id=?
```

四、DML编程控制

1. id生成策略控制 (Insert)

问题导入

主键生成的策略有哪几种方式？

不同的表应用不同的id生成策略

- 日志：自增 (1,2,3,4,)
- 购物订单：特殊规则 (FQ23948AK3843)
- 外卖单：关联地区日期等信息 (10 04 20200314 34 91)
- 关系表：可省略id
-

1.1 id生成策略控制 (@TableId注解)

- 名称：@TableId
- 类型：属性注解
- 位置：模型类中用于表示主键的属性定义上方
- 作用：设置当前类中主键属性的生成策略
- 相关属性

type：设置主键属性的生成策略，值参照IdType枚举值

```
public class User {
    @TableId(type = IdType.AUTO)
    private Long id;
}
```

AUTO(0): 使用数据库id自增策略控制id生成
 NONE(1): 不设置id生成策略
 INPUT(2): 用户手工输入id
 ASSIGN_ID(3): 雪花算法生成id (可兼容数值型与字符串型)
 ASSIGN_UUID(4): 以UUID生成算法作为id生成策略

针对每个公司，随着服务化演进，单个服务越来越多，数据库分的越来越细，有的时候一个业务需要分成好几个库，这时候自增主键或者序列之类的主键id生成方式已经不再满足需求，分布式系统中需要的是一个全局唯一的id生成规则。

具体的算法和代码，参见资料文件夹。

添加无参和有参的构造方法

```
@Data
@TableName("tbl_user")
public class User {

    public User() {
    }

    public User(Long id, String name, String gender, String password, Integer
age, String tel) {
        this.id = id;
        this.name = name;
        this.gender = gender;
        this.password = password;
        this.age = age;
        this.tel = tel;
    }

    //使用雪花算法
    @TableId(type = IdType.ASSIGN_ID)
    private Long id;
    private String name;
    private String gender;
    @TableField("pwd")
    private String password;
    private Integer age;
    private String tel;
}
```

测试

```
@Test
void insertUser() {
    //创建用户
    User user = new User(null, "孙悟空", "男", "12345", 8, "15022334455");
    UserMapper.insert(user);
}
```

执行的SQL语句

```
=> Preparing: INSERT INTO tbl_user ( id, name, gender, pwd, age, tel ) VALUES ( ?, ?, ?, ?, ?, ? )
=> Parameters: 1451568288478834690(Long), 孙悟空(String), 男(String), 12345(String), 8(Integer), 15022334455(String)
<== Updates: 1
```

1.2 全局策略配置

也可以在application.yml中进行全局的配置

1. id-type 让所有表主键生成策略相同
2. table-prefix 在每个实体类的前面添加相同的前缀

```
mybatis-plus:
  global-config:
    db-config:
      id-type: assign_id
      table-prefix: tbl_
```

id生成策略全局配置

```
@TableName("tbl_user")
public class User {
    @TableId(type = IdType.AUTO)
    private Long id;
}
```

```
@TableName("tbl_score")
public class Score {
    @TableId(type = IdType.AUTO)
    private Long id;
}
```

```
@TableName("tbl_subject")
public class Subject {
    @TableId(type = IdType.AUTO)
    private Long id;
}
```

```
@TableName("tbl_order")
public class Order {
    @TableId(type = IdType.AUTO)
    private Long id;
}
```

```
@TableName("tbl_equipment")
public class Equipment {
    @TableId(type = IdType.AUTO)
    private Long id;
}
```

```
@TableName("tbl_log")
public class Log {
    @TableId(type = IdType.AUTO)
    private Long id;
}
```

表名前缀全局配置

```
@TableName("tbl_user")
public class User {
    @TableId(type = IdType.AUTO)
    private Long id;
}
```

```
@TableName("tbl_score")
public class Score {
    @TableId(type = IdType.AUTO)
    private Long id;
}
```

```
@TableName("tbl_subject")
public class Subject {
    @TableId(type = IdType.AUTO)
    private Long id;
}
```

```
@TableName("tbl_order")
public class Order {
    @TableId(type = IdType.AUTO)
    private Long id;
}
```

```
@TableName("tbl_equipment")
public class Equipment {
    @TableId(type = IdType.AUTO)
    private Long id;
}
```

```
@TableName("tbl_log")
public class Log {
    @TableId(type = IdType.AUTO)
    private Long id;
}
```

实体类：去了@TableName注解和@TableId

```
@Data
public class User {

    public User() {
    }

    public User(Long id, String name, String gender, String password, Integer age, String tel) {
        this.id = id;
        this.name = name;
        this.gender = gender;
        this.password = password;
    }
}
```

```

        this.age = age;
        this.tel = tel;
    }

    private Long id;
    private String name;
    private String gender;
    @TableField("pwd")
    private String password;
    private Integer age;
    private String tel;
}

```

再次测试结果

Preparing: INSERT INTO tbl_user (id, name, gender, pwd, age, tel) VALUES (?, ?, ?, ?, ?, ?)
 Parameters: 1451569644786798594(Long), 牛魔王(String), 男(String), 345671(String), 35(Integer), 15022336666(String)
 Updates: 1

2. 多记录操作（批量Delete/Select）

问题导入

MyBatisPlus是否支持批量操作？



2.1 按照主键删除多条记录

使用方法：deleteBatchIds()

```

//删除指定多条数据
List<Long> list = new ArrayList<>();
list.add(1402551342481838081L);
list.add(1402553134049501186L);
list.add(1402553619611430913L);

userMapper.deleteBatchIds(list);

```

生成的SQL语句

```
DELETE FROM tbl_user WHERE id IN ( ?, ?, ? )
```

2.2 根据主键查询多条记录

使用方法: `selectBatchIds()`

```
//查询指定多条数据
List<Long> list = new ArrayList<>();
list.add(1L);
list.add(3L);
list.add(4L);
userMapper.selectBatchIds(list);
```

生成的SQL语句

```
SELECT id,name,gender,pwd AS password,age,te1 FROM tbl_user WHERE id IN ( ?, ?, ?, ? )
```

3. 逻辑删除 (Delete/Update)

问题导入

在实际环境中, 如果想删除一条数据, 是否会真的从数据库中删除该条数据?

- 删除操作业务问题: 业务数据从数据库中丢弃
- 逻辑删除: 为数据设置是否可用状态字段, 删除时设置状态字段为不可用状态, 数据保留在数据库中

合同编号	成交日期	金额	员工编号
1	2025/4/1	100,000.00	1
2	2025/5/12	300,000.00	1
3	2025/9/11	500,000.00	1
4	2025/11/14	80,000.00	2
5	2025/12/25	20,000.00	3

员工编号	姓名	工号	deleted
1	张业绩	9501	1
2	李小白	9502	0
3	王笑笑	9503	0

编号	姓名	业绩
1	张业绩	900,000.00
2	李小白	80,000.00
3	王笑笑	20,000.00
合计		1,000,000.00


3.1 逻辑删除案例

修改表结构

```
-- 添加一列deleted, 注意设置默认值为0
ALTER TABLE tbl_user ADD COLUMN deleted INT(1) DEFAULT 0;

-- 查看结构
DESC tbl_user;
```

①：数据库表中添加逻辑删除标记字段

字段	索引	外键	触发器	选项	注释	SQL 预览					
名					类型	长度	小数点	不是 null	虚拟	键	
id					bigint	20	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	 1	
name					varchar	32	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
pwd					varchar	32	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
age					int	3	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
tel					varchar	32	0	<input type="checkbox"/>	<input type="checkbox"/>		
* deleted					int	1		<input type="checkbox"/>	<input type="checkbox"/>		

②：实体类中添加对应字段，并设定当前字段为逻辑删除标记字段

@TableLogic包含以下属性

- value：未删除时的值
- delval：删除了的值

```
@Data
public class User {

    private Long id;

    //逻辑删除字段，标记当前记录是否被删除
    @TableLogic
    private Integer deleted;

}
```

③：配置逻辑删除字段值

```
mybatis-plus:
  global-config:
    db-config:
      table-prefix: tbl_
      # 逻辑删除字段名
      logic-delete-field: deleted
      # 逻辑删除字段值：未删除为0
      logic-not-delete-value: 0
      # 逻辑删除字段值：删除为1
      logic-delete-value: 1
```

逻辑删除本质：逻辑删除的本质其实是修改操作。如果加了逻辑删除字段，查询数据时也会自动带上逻辑删除字段。

```
@Test
void testLogicDeleted() {
    int row = UserMapper.deleteById(5);
    System.out.println(row + "条记录被逻辑删除");
}
```

执行SQL语句: UPDATE tbl_user SET **deleted=1** WHERE id=? AND **deleted=0**

执行数据结果:

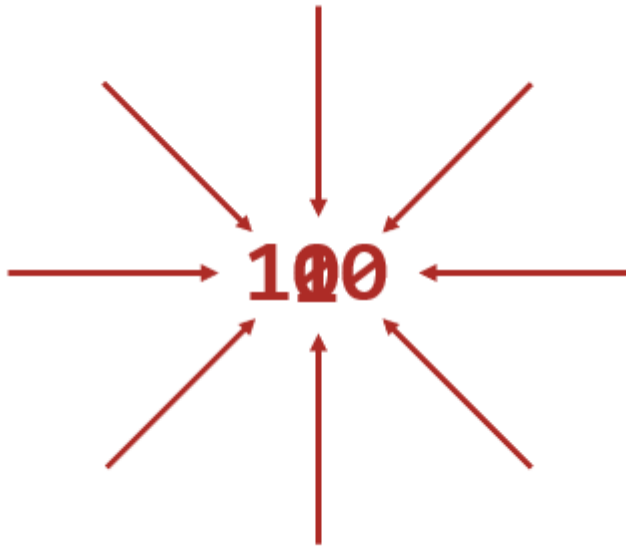
id	name	pwd	age	tel	deleted
1	Tom888	tom888	3	18866668888	1
2	Jerry	jerry	4	16688886666	0
3	Jock	123456	41	18812345678	0
4	传智播客	itcast	15	4006184000	0
5	黑马程序员	itheima	12	4006184000	0
666	黑马程序员	itheima	12	4006184000	0
667	黑马程序员	itheima	12	4006184000	0

4. 乐观锁 (Update)

4.1 问题导入

乐观锁主张的思想是什么?

- 业务并发现象带来的问题: 秒杀



4.2 什么是悲观锁和乐观锁

悲观锁(Pessimistic Lock)

当要对数据库中的一条数据进行修改的时候, 为了避免同时被其他人修改, 最好的办法就是直接对该数据进行加锁以防止并发。这种借助数据库锁机制, 在修改数据之前先锁定, 再修改的方式被称之为悲观锁。

之所以叫做悲观锁, 是因为这是一种对数据的修改持有悲观态度的并发控制方式。总是假设最坏的情况, 每次读取数据的时候都默认其他线程会更改数据, 因 线程想要访问数据时, 都需要阻塞挂起。

乐观锁(Optimistic Locking)

乐观锁是相对悲观锁而言的, 乐观锁假设数据一般情况不会造成冲突, 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, 乐观锁适用于读多写少的场景, 这样可以提高程序的吞吐量。

乐观锁采取了更加宽松的加锁机制。也是为了避免数据库幻读、业务处理时间过长等原因引起数据处理错误的一种机制, 但乐观锁不会刻意使用数据库本身的锁机制, 而是依据数据本身来保证数据的正确性。

乐观锁的实现

1. CAS 实现：Java 中 `java.util.concurrent.atomic` 包下面的原子变量使用了乐观锁的一种 CAS 实现方式。
2. 版本号控制：一般是在数据表中加上一个数据版本号 `version` 字段，表示数据被修改的次数。当数据被修改时，`version` 值会 +1。当线程 A 要更新数据时，在读取数据的同时也会读取 `version` 值，在提交更新时，若刚才读取到的 `version` 值与当前数据库中的 `version` 值相等时才更新，否则重试更新操作，直到更新成功。

4.3 乐观锁案例

①：数据库表中添加锁标记字段

```
ALTER TABLE tbl_user ADD COLUMN `version` INT DEFAULT 0;
```

字段	索引	外键	触发器	选项	注释	SQL 预览				
名					类型	长度	小数点	不是 null	虚拟	键
id					bigint	20	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	🔑 1
name					varchar	32	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
pwd					varchar	32	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
age					int	3	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
tel					varchar	32	0	<input type="checkbox"/>	<input type="checkbox"/>	
deleted					int	1	0	<input type="checkbox"/>	<input type="checkbox"/>	
> version					int	11	0	<input type="checkbox"/>	<input type="checkbox"/>	

②：实体类中添加对应字段，并设定当前字段为版本控制字段

@Version注解

```
package com.itheima.domain;

@Data
public class User {

    private Long id;

    @Version
    private Integer version;
}
```

③：配置乐观锁拦截器实现锁机制对应的动态SQL语句拼装

```
package com.itheima.config;

@Configuration
public class MybatisPlusConfig {

    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor() {
        //1. 定义Mp拦截器
        MybatisPlusInterceptor mpInterceptor = new MybatisPlusInterceptor();

        //2. 添加乐观锁拦截器
        mpInterceptor.addInnerInterceptor(new OptimisticLockerInnerInterceptor());
    }
}
```

```

        return mpInterceptor;
    }
}

```

④：使用乐观锁机制在修改前必须先获取到对应数据的version方可正常进行

```

/**
 * 乐观锁的测试
 */
@Test
public void testLock(){
    //1. 查询当前要修改的记录
    User user = userDao.selectById(6);
    //2. 修改数据
    user.setAge(32);
    userDao.updateById(user);
}

```

执行修改前先执行查询语句：

```
SELECT id,name,age,tel,deleted,version FROM tbl_user WHERE id=?
```

执行修改时使用version字段作为乐观锁检查依据

```
UPDATE tbl_user SET name=?, age=?, tel=?, version=? WHERE id=? AND version=?
```

模拟多条记录同时更新

```

/**
 * 乐观锁的测试2,并发修改
 */
@Test
public void testLock2(){
    //1. 查询当前要修改的记录
    User user1 = userDao.selectById(6); //version=1
    User user2 = userDao.selectById(6); //version=1
    //2. 模拟线程1修改数据
    user1.setName("张小 AA");
    userDao.updateById(user1); //version=1 where version=1 and id=6 , 修改完
    //的时候version=2
    user2.setName("张晓 BB"); // where version = 1 and id=6
    userDao.updateById(user2);
}

```

第二条记录更新失败

```

Preparing: UPDATE tbl_user SET name=?, gender=?, pwd=?, age=?, tel=?, version=? WHERE id=? AND version=? AND deleted=0
Parameters: Jock aaa(String), 男(String), 123456(String), 8(Integer), 12345678910(String), 3(Integer), 2(Long), 2(Integer)
Updates: 1
ing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@2dfff7085]
ting a new SqlSession
ession [org.apache.ibatis.session.defaults.DefaultSqlSession@6897a4a] was not registered for synchronization because synchroni
Connection [com.mysql.jdbc.JDBC4Connection@739831a4] will not be managed by Spring
Preparing: UPDATE tbl_user SET name=?, gender=?, pwd=?, age=?, tel=?, version=? WHERE id=? AND version=? AND deleted=0
Parameters: Jock bbb(String), 男(String), 123456(String), 8(Integer), 12345678910(String), 3(Integer), 2(Long), 2(Integer)
Updates: 0

```

五、代码生成器

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 等各个模块的代码，极大的提升了开发效率。

以后只需要创建好表，然后enti实体类，dao、service、controller都可以自动生成。

5.1 创建maven工程

创建maven工程：mybatisplus_code

The screenshot shows the 'New Module' dialog box with the following fields filled out:

- Parent: <None>
- Name: mybatisplus_code
- Location: D:\itcast\jee148\project148\mybatisplus_code
- Artifact Coordinates section:
 - GroupId: cn.itcast (with a note: 'The name of the artifact group, usually a company domain')
 - ArtifactId: mybatisplus_code (with a note: 'The name of the artifact within the group, usually a module name')
 - Version: 1.0-SNAPSHOT

At the bottom right, there are buttons for 'Previous', 'Finish', 'Cancel', and 'Help'.

5.2 导入依赖

```
<!-- 父工程给生成后的代码用的-->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.6.RELEASE</version>
</parent>

<dependencies>

    <!-- 代码生成器 -->
    <dependency>
        <groupId>com.baomidou</groupId>
        <artifactId>mybatis-plus-generator</artifactId>
        <version>3.4.0</version>
    </dependency>

    <!-- 模块引擎 -->
    <dependency>
        <groupId>org.freemarker</groupId>
        <artifactId>freemarker</artifactId>
```

```

        <version>2.3.30</version>
    </dependency>

    <!--下面是生成后的代码需要用到的依赖-->

    <!--web启动器-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!--mybatis启动器-->
    <dependency>
        <groupId>com.baomidou</groupId>
        <artifactId>mybatis-plus-boot-starter</artifactId>
        <version>3.4.0</version>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
    </dependency>

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

</dependencies>

```

5.3 代码生成类

创建test包，复制此类到包下

```

package cn.itcast.test;

import com.baomidou.mybatisplus.core.exceptions.MybatisPlusException;
import com.baomidou.mybatisplus.core.toolkit.StringPool;
import com.baomidou.mybatisplus.core.toolkit.StringUtils;
import com.baomidou.mybatisplus.generator.AutoGenerator;
import com.baomidou.mybatisplus.generator.InjectionConfig;
import com.baomidou.mybatisplus.generator.config.*;
import com.baomidou.mybatisplus.generator.config.po.TableInfo;
import com.baomidou.mybatisplus.generator.config.rules.NamingStrategy;
import com.baomidou.mybatisplus.generator.engine.FreemarkerTemplateEngine;

import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

// 演示例子，执行 main 方法控制台输入模块表名回车自动生成对应项目目录中
public class CodeGenerator {

```

```

public static String scanner(String tip) {
    Scanner scanner = new Scanner(System.in);
    StringBuilder help = new StringBuilder();
    help.append("请输入" + tip + ": ");
    System.out.println(help.toString());
    if (scanner.hasNext()) {
        String ipt = scanner.next();
        if (StringUtils.isNotBlank(ipt)) {
            return ipt;
        }
    }
    throw new MybatisPlusException("请输入正确的" + tip + "!!");
}

public static void main(String[] args) {
    // 代码生成器
    AutoGenerator mpg = new AutoGenerator();

    // 全局配置
    GlobalConfig gc = new GlobalConfig();
    String projectPath = System.getProperty("user.dir");
    String moduleName = scanner("请代码存储的模块名");
    gc.setOutputDir(projectPath + "/" + moduleName + "/src/main/java");
    //代码的作者
    gc.setAuthor("itheima");
    gc.setOpen(false);
    mpg.setGlobalConfig(gc);

    // 数据源配置
    DataSourceConfig dsc = new DataSourceConfig();
    dsc.setUrl("jdbc:mysql://localhost:3306/springdb?
useUnicode=true&useSSL=false&characterEncoding=utf8");
    // dsc.setSchemaName("public");
    dsc.setDriverName("com.mysql.jdbc.Driver");
    dsc.setUsername("root");
    dsc.setPassword("root");
    mpg.setDataSource(dsc);

    // 包配置
    PackageConfig pc = new PackageConfig();
    pc.setModuleName(scanner("功能模块名"));
    //设置父级包名 com.itheima.user com.itheima.teacher
    pc.setParent("com.itheima");
    mpg.setPackageInfo(pc);

    // 自定义配置
    InjectionConfig cfg = new InjectionConfig() {
        @Override
        public void initMap() {
            // to do nothing
        }
    };

    // 如果模板引擎是 freemarker
    String templatePath = "/templates/mapper.xml.ftl";

    // 自定义输出配置

```

```

List<FileOutConfig> focList = new ArrayList<>();
// 自定义配置会被优先输出
focList.add(new FileOutConfig(templatePath) {
    @Override
    public String outputFile(TableInfo tableInfo) {
        // 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！！
        return projectPath +
            "/" + moduleName + "/src/main/resources/mapper/" + pc.getModuleName()
            + "/" + tableInfo.getEntityName() + "Mapper" +
StringPool.DOT_XML;
    }
});
cfg.setFileOutConfigList(focList);
mpg.setCfg(cfg);

// 配置模板
TemplateConfig templateConfig = new TemplateConfig();

templateConfig.setXml(null);
mpg.setTemplate(templateConfig);

// 策略配置
StrategyConfig strategy = new StrategyConfig();
strategy.setNaming(NamingStrategy.underline_to_camel);
strategy.setColumnNaming(NamingStrategy.underline_to_camel);
// strategy.setSuperEntityClass("你自己的父类实体,没有就不用设置!");
strategy.setEntityLombokModel(true);
strategy.setRestControllerStyle(true);
// 公共父类
// strategy.setSuperControllerClass("你自己的父类控制器,没有就不用设置!");
// 写于父类中的公共字段
// strategy.setSuperEntityColumns("id");
strategy.setInclude(scanner("表名，多个英文逗号分割").split(","));
strategy.setControllerMappingHyphenStyle(true);
String preName = scanner("请输入表前缀名");
strategy.setTablePrefix(preName); // 设置表前缀
mpg.setStrategy(strategy);
mpg.setTemplateEngine(new FreemarkerTemplateEngine());

// 执行
mpg.execute();
}
}

```

5.4 执行

右键运行main函数，在控制台输入功能模块名，表名，再按回车即可

```
E:\software\Java\jdk1.8.0_151\bin\java.exe ...
```

```
D:\itcast\ee148\project148
```

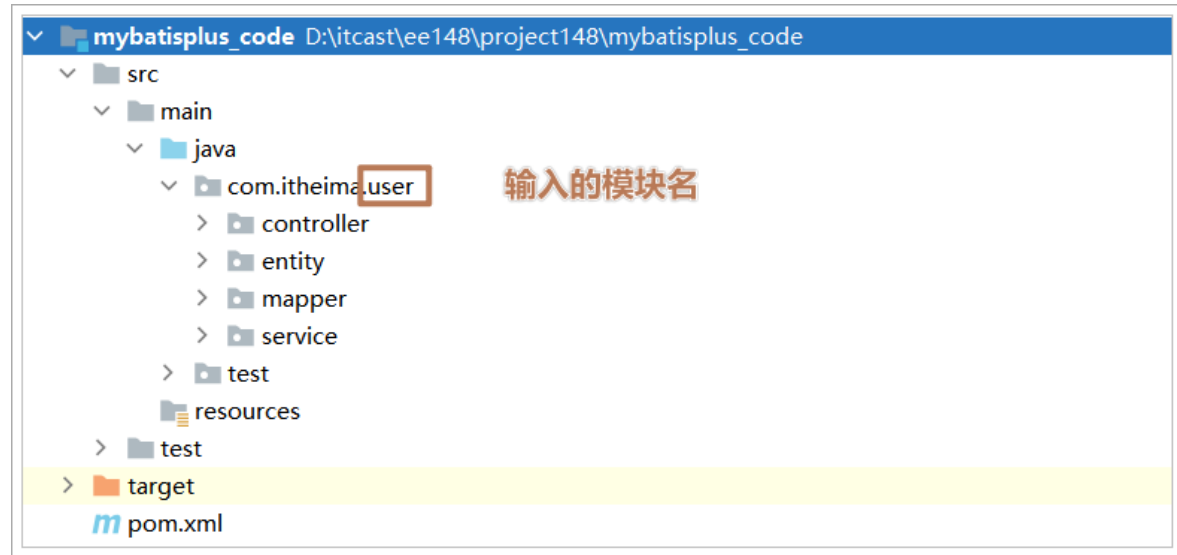
```
请输入功能模块名: |
```

```
user
```

```
请输入表名，多个英文逗号分割:
```

```
tb_user
```

执行完成后刷新项目，发现代码已经生成完毕，每个表都生成对应每一层的代码：



###