

day01-MybatisPlus

本篇学习文档对应B站视频：

<https://www.bilibili.com/video/BV1S142197x7?p=2>

大家在日常开发中应该能发现，单表的CRUD功能代码重复度很高，也没有什么难度。而这部分代码量往往比较大，开发起来比较费时。

因此，目前企业中都会使用一些组件来简化或省略单表的CRUD开发工作。目前在国内使用较多的一个组件就是MybatisPlus.

官方网站如下：

<https://www.baomidou.com/>

当然，MybatisPlus不仅仅可以简化单表操作，而且对Mybatis的功能有很多的增强。可以让我们的开发更加的简单，高效。

通过今天的学习，我们要达成下面的目标：

- 能利用MybatisPlus实现基本的CRUD
- 会使用条件构建造器构建查询和更新语句
- 会使用MybatisPlus中的常用注解
- 会使用MybatisPlus处理枚举、JSON类型字段
- 会使用MybatisPlus实现分页

1. 快速入门

为了方便测试，我们先创建一个新的项目，并准备一些基础数据。

1.1. 环境准备

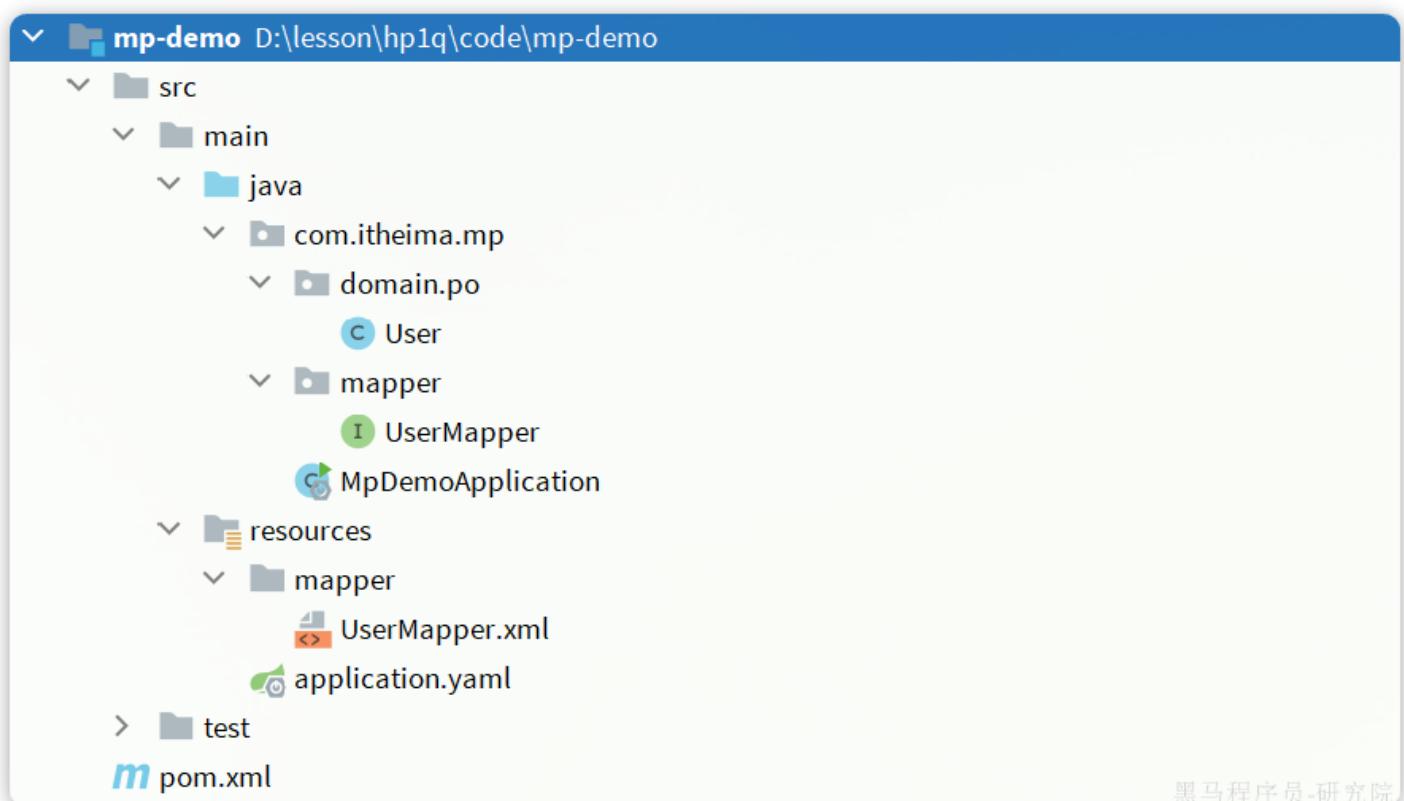
复制课前资料提供好的一个项目到你的工作空间（不要包含空格和特殊字符）：

新加卷 (D:) > 课程资料 > 服务框架 > day01-MybatisPlus > 资料 >

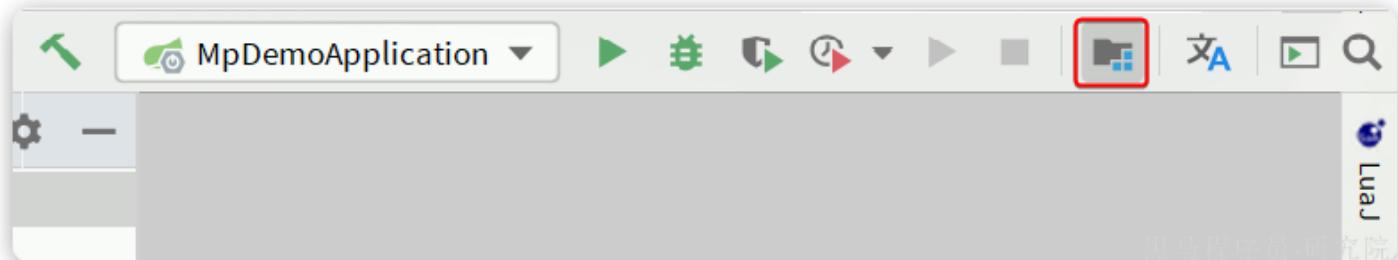
| 名称 | 类型 | 大小 |
|---------|---------|------|
| mp-demo | 文件夹 | |
| mp.sql | SQL 源文件 | 6 KB |

黑马程序员-研究院

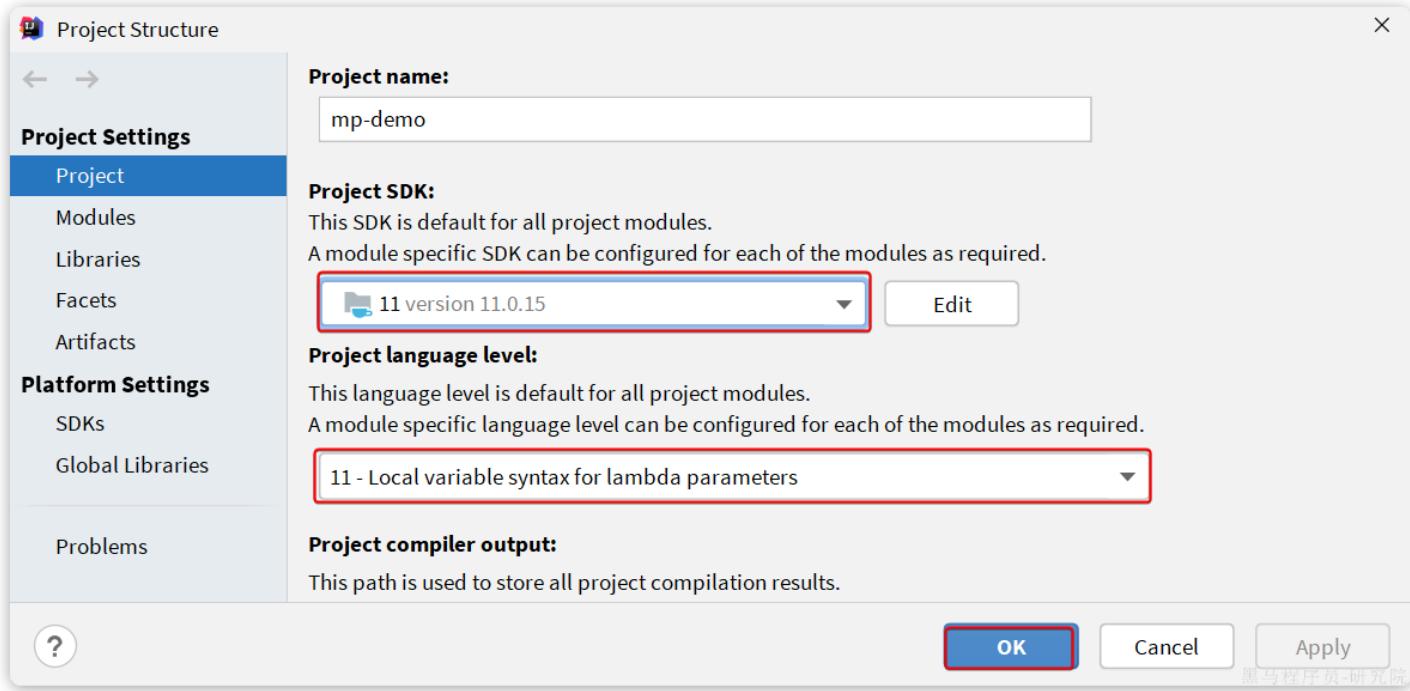
然后用你的IDEA工具打开，项目结构如下：



注意配置一下项目的JDK版本为JDK11。首先点击项目结构设置：



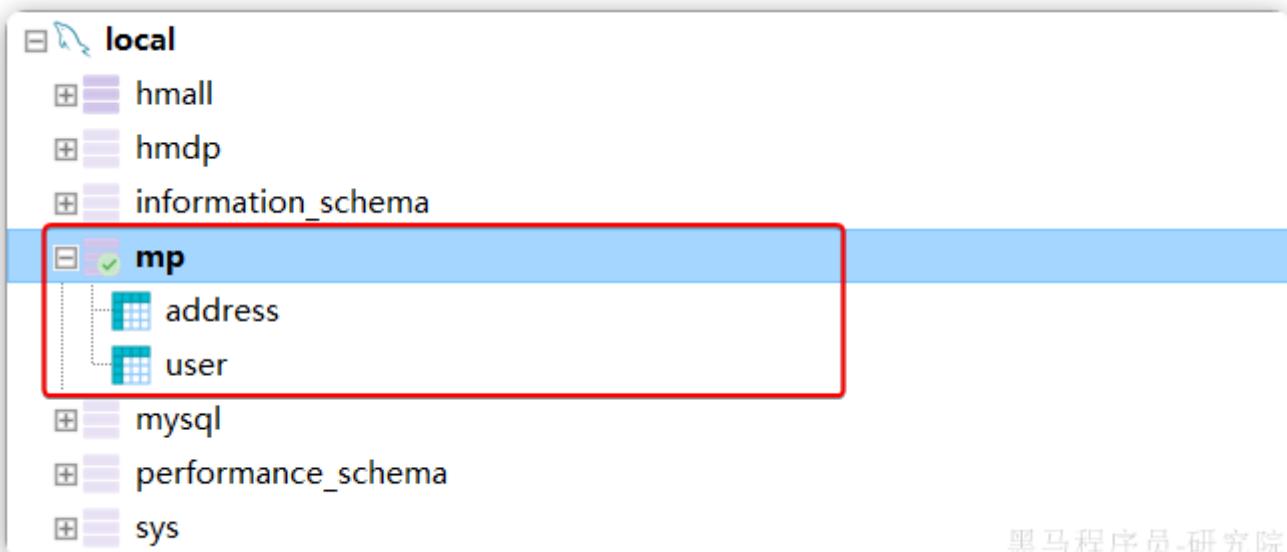
在弹窗中配置JDK：



接下来，要导入两张表，在课前资料中已经提供了SQL文件：

| 新加卷 (D:) > 课程资料 > 服务框架 > day01-MybatisPlus > 资料 > | | | |
|---|---------|------|--|
| 名称 | 类型 | 大小 | |
| po | 文件夹 | | |
| mp.sql | SQL 源文件 | 5 KB | |

对应的数据表结构如下：



最后，在 `application.yaml` 中修改jdbc参数为自己的数据库参数：

```
1 spring:
2   datasource:
3     url: jdbc:mysql://127.0.0.1:3306/mp?useUnicode=true&characterEncoding=UTF-
4       &autoReconnect=true&serverTimezone=Asia/Shanghai
5     driver-class-name: com.mysql.cj.jdbc.Driver
6     username: root
7     password: MySQL123
8   logging:
9     level:
10    com.itheima: debug
11   pattern:
12     dateformat: HH:mm:ss
```

1.2.快速开始

比如我们要实现User表的CRUD，只需要下面几步：

- 引入MybatisPlus依赖
- 定义Mapper

1.2.1引入依赖

MybatisPlus提供了starter，实现了自动Mybatis以及MybatisPlus的自动装配功能，坐标如下：

```
1 <dependency>
2   <groupId>com.baomidou</groupId>
3   <artifactId>mybatis-plus-boot-starter</artifactId>
4   <version>3.5.3.1</version>
5 </dependency>
```

由于这个starter包含对mybatis的自动装配，因此完全可以替换掉Mybatis的starter。

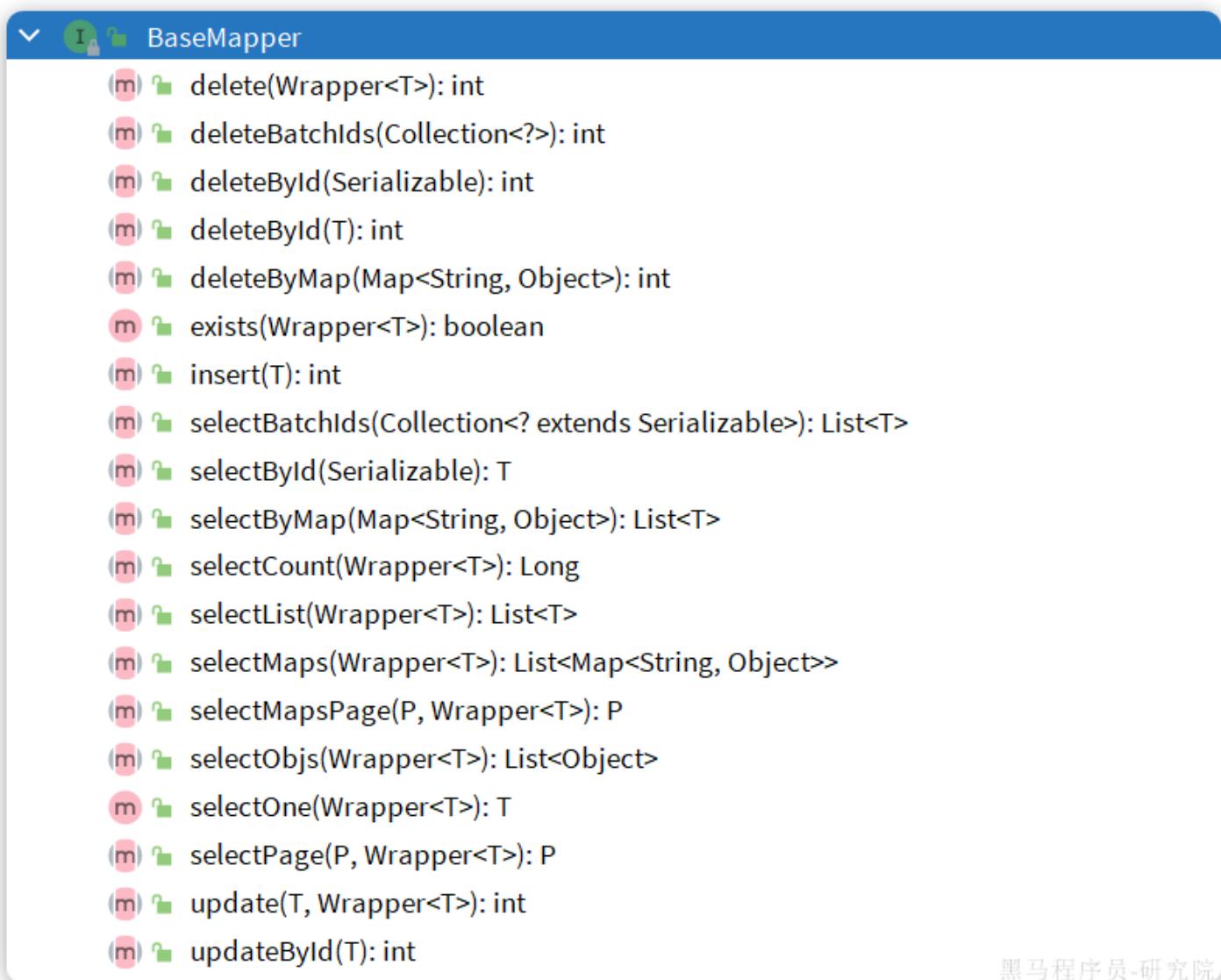
最终，项目的依赖如下：

```
1 <dependencies>
2   <dependency>
3     <groupId>com.baomidou</groupId>
4     <artifactId>mybatis-plus-boot-starter</artifactId>
5     <version>3.5.3.1</version>
6   </dependency>
7   <dependency>
8     <groupId>com.mysql</groupId>
9     <artifactId>mysql-connector-j</artifactId>
10    <scope>runtime</scope>
```

```
11    </dependency>
12    <dependency>
13        <groupId>org.projectlombok</groupId>
14        <artifactId>lombok</artifactId>
15        <optional>true</optional>
16    </dependency>
17    <dependency>
18        <groupId>org.springframework.boot</groupId>
19        <artifactId>spring-boot-starter-test</artifactId>
20        <scope>test</scope>
21    </dependency>
22 </dependencies>
```

1.2.2. 定义Mapper

为了简化单表CRUD，MybatisPlus提供了一个基础的 `BaseMapper` 接口，其中已经实现了单表的CRUD：



因此我们自定义的Mapper只要实现了这个 `BaseMapper`，就无需自己实现单表CRUD了。修改mp-demo中的 `com.itheima.mp.mapper` 包下的 `UserMapper` 接口，让其继承 `BaseMapper`：

```
1 package com.itheima.mp.mapper;  
2  
3 import com.baomidou.mybatisplus.core.mapper.BaseMapper;  
4 import com.itheima.mp.domain.po.User;  
5  
6 public interface UserMapper extends BaseMapper<User> {  
7 }
```

黑马程序员·研究院

代码如下：

```
1 package com.itheima.mp.mapper;  
2  
3 import com.baomidou.mybatisplus.core.mapper.BaseMapper;  
4 import com.itheima.mp.domain.po.User;  
5  
6 public interface UserMapper extends BaseMapper<User> {  
7 }
```

1.2.3. 测试

新建一个测试类，编写几个单元测试，测试基本的CRUD功能：

```
1 package com.itheima.mp.mapper;  
2  
3 import com.itheima.mp.domain.po.User;  
4 import org.junit.jupiter.api.Test;  
5 import org.springframework.beans.factory.annotation.Autowired;  
6 import org.springframework.boot.test.context.SpringBootTest;  
7  
8 import java.time.LocalDateTime;  
9 import java.util.List;  
10  
11 @SpringBootTest
```

```
12 class UserMapperTest {  
13  
14     @Autowired  
15     private UserMapper userMapper;  
16  
17     @Test  
18     void testInsert() {  
19         User user = new User();  
20         user.setId(5L);  
21         user.setUsername("Lucy");  
22         user.setPassword("123");  
23         user.setPhone("18688990011");  
24         user.setBalance(200);  
25         user.setInfo("{\"age\": 24, \"intro\": \"英文老师\", \"gender\":  
26             \"female\"}");  
27         user.setCreateTime(LocalDateTime.now());  
28         user.setUpdateTime(LocalDateTime.now());  
29         userMapper.insert(user);  
30     }  
31  
32     @Test  
33     void testSelectById() {  
34         User user = userMapper.selectById(5L);  
35         System.out.println("user = " + user);  
36     }  
37  
38     @Test  
39     void testSelectByIds() {  
40         List<User> users = userMapper.selectBatchIds(List.of(1L, 2L, 3L, 4L,  
41             5L));  
42         users.forEach(System.out::println);  
43     }  
44  
45     @Test  
46     void testUpdateById() {  
47         User user = new User();  
48         user.setId(5L);  
49         user.setBalance(20000);  
50         userMapper.updateById(user);  
51     }  
52  
53     @Test  
54     void testDelete() {  
55         userMapper.deleteById(5L);  
56     }  
57 }
```

可以看到，在运行过程中打印出的SQL日志，非常标准：

```
1 11:05:01 INFO 15524 --- [           main] com.zaxxer.hikari.HikariDataSource
   : HikariPool-1 - Starting...
2 11:05:02 INFO 15524 --- [           main] com.zaxxer.hikari.HikariDataSource
   : HikariPool-1 - Start completed.
3 11:05:02 DEBUG 15524 --- [          main] c.i.mp.mapper.UserMapper.selectById
   : ==> Preparing: SELECT
   id,username,password,phone,info,status,balance,create_time,update_time FROM
   user WHERE id=?
4 11:05:02 DEBUG 15524 --- [          main] c.i.mp.mapper.UserMapper.selectById
   : ==> Parameters: 5(Long)
5 11:05:02 DEBUG 15524 --- [          main] c.i.mp.mapper.UserMapper.selectById
   : <==      Total: 1
6 user = User(id=5, username=Lucy, password=123, phone=18688990011, info={"age":21}, status=1, balance=20000, createTime=Fri Jun 30 11:02:30 CST 2023, updateTime=Fri Jun 30 11:02:30 CST 2023)
```

只需要继承BaseMapper就能省去所有的单表CRUD，是不是非常简单！

1.3. 常见注解

在刚刚的入门案例中，我们仅仅引入了依赖，继承了BaseMapper就能使用MybatisPlus，非常简单。但是问题来了：

MybatisPlus如何知道我们要查询的是哪张表？表中有哪些字段呢？

大家回忆一下，UserMapper在继承BaseMapper的时候指定了一个泛型：

```
package com.itheima.mp.mapper;

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

public interface UserMapper extends BaseMapper<User> {
}
```

黑马程序员-研究院

泛型中的User就是与数据库对应的PO.

MybatisPlus就是根据PO实体的信息来推断出表的信息，从而生成SQL的。默认情况下：

- MybatisPlus会把PO实体的类名驼峰转下划线作为表名
- MybatisPlus会把PO实体的所有变量名驼峰转下划线作为表的字段名，并根据变量类型推断字段类型
- MybatisPlus会把名为id的字段作为主键

但很多情况下，默认的实现与实际场景不符，因此MybatisPlus提供了一些注解便于我们声明表信息。

1.3.1. @TableName

说明：

- 📌 • 描述：表名注解，标识实体类对应的表
• 使用位置：实体类

示例：

```
1 @TableName("user") 假设数据库表名为tb_user，则此处需要通过“@TableName("tb_
2 public class User { user")”指定表名
3     private Long id;
4     private String name;
5 }
```

TableName注解除了指定表名以外，还可以指定很多其它属性：

| 属性 | 类型 | 必须指定 | 默认值 | 描述 |
|------------------|---------|------|-------|---|
| value | String | 否 | "" | 表名 |
| schema | String | 否 | "" | schema |
| keepGlobalPrefix | boolean | 否 | false | 是否保持使用全局的 tablePrefix 的值（当全局 tablePrefix 生效时） |
| resultMap | String | 否 | "" | xml 中 resultMap 的 id（用于满足特定类型的实体类对象绑定） |
| autoResultMap | boolean | 否 | false | 是否自动构建 resultMap 并使用（如果设置 resultMap 则不会进行 resultMap 的自动构建与注入） |

| | | | | |
|---------------------|----------|---|----|-----------------------|
| excludeProper ty | String[] | 否 | {} | 需要排除的属性名 @since 3.3.1 |
|---------------------|----------|---|----|-----------------------|

1.3.2. @TableId

说明：

- 描述：主键注解，标识实体类中的主键字段
- 使用位置：实体类的主键字段

示例：

```

1 @TableName("user")
2 public class User {
3     @TableId
4     private Long id;      假如数据库表的主键叫id，则此处应该使用“@  

5     private String name;
6 }
```

`TableId` 注解支持两个属性：

| 属性 | 类型 | 必须指定 | 默认值 | 描述 |
|-------|--------|------|-------------|--------|
| value | String | 否 | "" | 表名 |
| type | Enum | 否 | IdType.NONE | 指定主键类型 |

`IdType` 支持的类型有：一般要注释上与数据库相同的类型，如自增长，否则默认使用ASSIGN_ID，雪花算法，使用一个很长的ID

| 值 | 描述 |
|-------|---|
| AUTO | 数据库 ID 自增 |
| NONE | 无状态，该类型为未设置主键类型（注解里等于跟随全局，全局里约等于 INPUT） |
| INPUT | insert 前自行 set 主键值 |

| | |
|---------------|---|
| ASSIGN_ID | 分配 ID(主键类型为 Number(Long 和 Integer)或 String)(since 3.3.0), 使用接口 IdentifierGenerator 的方法 nextId(默认实现类为 DefaultIdentifierGenerator 雪花算法) |
| ASSIGN_UUID | 分配 UUID, 主键类型为 String(since 3.3.0), 使用接口 IdentifierGenerator 的方法 nextUUID(默认 default 方法) |
| ID_WORKER | 分布式全局唯一 ID 长整型类型(please use ASSIGN_ID) |
| UUID | 32 位 UUID 字符串(please use ASSIGN_UUID) |
| ID_WORKER_STR | 分布式全局唯一 ID 字符串类型(please use ASSIGN_ID) |

这里比较常见的有三种：

- AUTO：利用数据库的id自增长
- INPUT：手动生成id
- ASSIGN_ID：雪花算法生成 Long 类型的全局唯一id，这是默认的ID策略

1.3.3. @TableField

说明：

描述：普通字段注解

示例：

```

1 @TableName("user")
2 public class User {
3     @TableId
4     private Long id;
5     private String name;
6     private Integer age;
7     @TableField("isMarried")
8     private Boolean isMarried;
9     @TableField("concat")
10    private String concat;
11 }
```

假如还有一个
private Integer order
属性

由于该属性与数据库关键字 order by 的 order 冲突
，所以也要使用 `@TableField` 注解来处理
@TableField("`order`")
使用转义字符，见下面的最后一条

假设 isMarried 数据库中的对应字段为 is_married
受到这条规则的影响，MP 处理的时候会自动去掉 "is"
所以要使用 `@TableField` 注解，注为：
@TableField("is_married")

假如还有一个
private String address
属性

该属性在数据库表中不存在，则需要使用
@TableField(exist = false)
来表明该属性不是数据库中的属性

一般情况下我们并不需要给字段添加 @TableField 注解，一些特殊情况除外：

- 成员变量名与数据库字段名不一致
- 成员变量是以 isXXX 命名，按照 JavaBean 的规范，MybatisPlus 识别字段时会把 is 去除，这就导致与数据库不符。

- 成员变量名与数据库一致，但是与数据库的关键字冲突。使用 `@TableField` 注解给字段名添加转义字符：` `

支持的其它属性如下：

| 属性 | 类型 | 必填 | 默认值 | 描述 |
|----------------------|----------|----|---------------------------|---|
| value | String | 否 | "" | 数据库字段名 |
| exist | boolean | 否 | true | 是否为数据库表字段 |
| condition | String | 否 | "" | 字段 where 实体查询比较条件，有值设置则按设置的值为准，没有则为默认全局的 %s=# {%s}， 参考 (opens new window) |
| update | String | 否 | "" | 字段 update set 部分注入，例如：当在 version字段上注解update="%s+1" 表示更新时会 set version=version+1 (该属性优先级高于 el 属性) |
| insertStrategy | Enum | 否 | FieldStrategy.DEFA ULT | 举例：NOT_NULL insert into table_a(<if test="columnProperty != null">column</if>) values (<if test="columnProperty != null">#{columnProperty}</if>) |
| updateStrateg y | Enum | 否 | FieldStrategy.DEFA ULT | 举例：IGNORED update table_a set column=#{columnProperty} |
| whereStrategy | Enum | 否 | FieldStrategy.DEFA ULT | 举例：NOT_EMPTY where <if test="columnProperty != null and columnProperty!="">column=#{columnProperty}</if> |
| fill | Enum | 否 | FieldFill.DEFAULT | 字段自动填充策略 |
| select | boolean | 否 | true | 是否进行 select 查询 |
| keepGlobalFor mat | boolean | 否 | false | 是否保持使用全局的 format 进行处理 |
| jdbcType | JdbcType | 否 | JdbcType.UNDEFIN ED | JDBC 类型 (该默认值不代表会按照该值生效) |
| typeHandler | TypeHan | 否 | | 类型处理器 (该默认值不代表会按照该值生效) |

| | | | | |
|--------------|--------|---|----|-------------|
| | der | | | |
| numericScale | String | 否 | "" | 指定小数点后保留的位数 |

1.4. 常见配置 <https://www.baomidou.com/reference/new-code-generator-configuration/>

MybatisPlus也支持基于yaml文件的自定义配置，详见官方文档：



<https://www.baomidou.com/pages/56bac0/#%E5%9F%BA%E6%9C%AC%E9%80%89>
使用配置 | MyBatis-Plus
MyBatis-Plus 官方文档

大多数的配置都有默认值，因此我们都无需配置。但还有一些是没有默认值的，例如：

- 实体类的别名扫描包
- 全局id类型

```

1 mybatis-plus: type-aliases-package是别名扫描包，指定实体类的路径
2   type-aliases-package: com.itheima.mp.domain.po
3   global-config:
4     db-config:
5       id-type: auto # 全局id类型为自增长 默认为auto，若为“assign_id”则是雪花算法
        update-strategy: not_null # 更新策略：只更新非空字段
    
```

需要注意的是，MyBatisPlus也支持手写SQL的，而mapper文件的读取地址可以自己配置：

MP更适合单表查询，对于多表查询还需要自己写，所以可以配置mapper的路径，这里给的是默认值

```

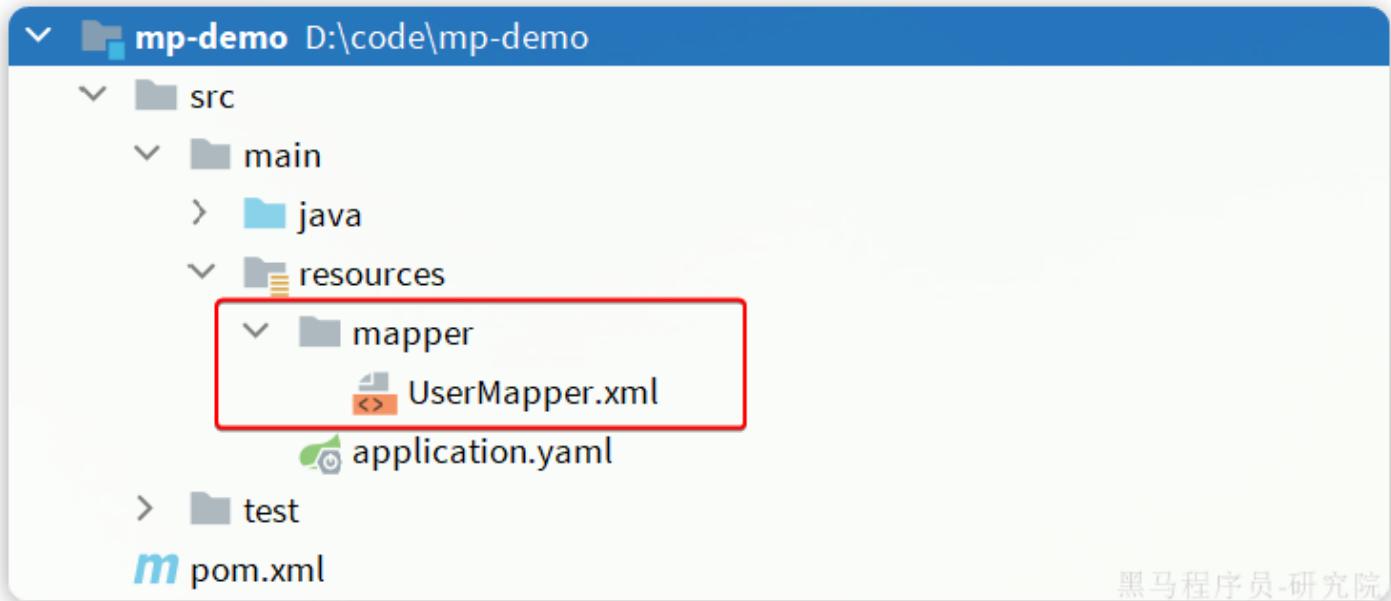
1 mybatis-plus:
2   mapper-locations: "classpath*:mapper/**/*.xml" # Mapper.xml文件地址，当前这个是
    默认值。
    
```

可以看到默认值是 `classpath*:mapper/**/*.xml`，也就是说我们只要把mapper.xml文件放置这个目录下就一定会被加载。

例如，我们新建一个 `UserMapper.xml` 文件：

```

mybatis-plus:
  configuration:
    map-underscore-to-camel-case: true # 是否开启下划线和驼峰的映射
    cache-enabled: false # 是否开启二级缓存
    
```



然后在其中定义一个方法：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
3 <mapper namespace="com.itheima.mp.mapper.UserMapper">
4
5   <select id="queryById" resultType="User">
6     SELECT * FROM user WHERE id = #{id}
7   </select>
8 </mapper>
```

然后在测试类 `UserMapperTest` 中测试该方法：

```
1 @Test
2 void testQuery() {
3   User user = userMapper.queryById(1L);
4   System.out.println("user = " + user);
5 }
```

2.核心功能

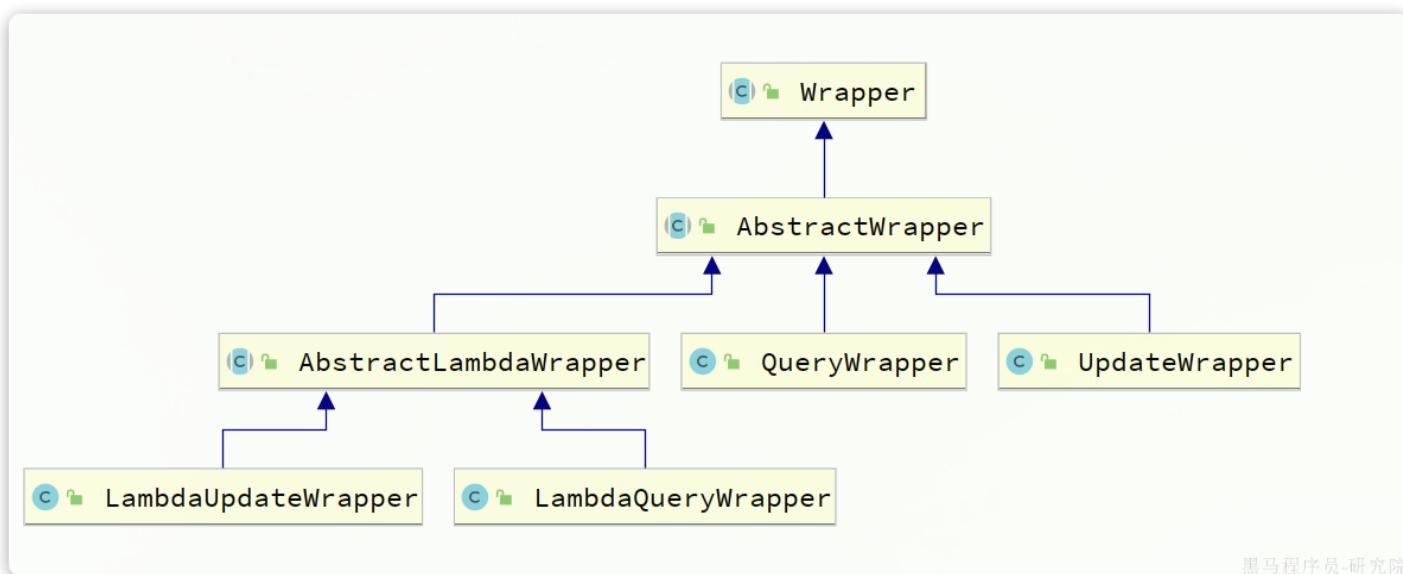
刚才的案例中都是以id为条件的简单CRUD，一些复杂条件的SQL语句就要用到一些更高级的功能了。

2.1. 条件构造器

除了新增以外，修改、删除、查询的SQL语句都需要指定where条件。因此BaseMapper中提供的相关方法除了以 id 作为 where 条件以外，还支持更加复杂的 where 条件。



参数中的 `Wrapper` 就是条件构造的抽象类，其下有很多默认实现，继承关系如图：



`Wrapper` 的子类 `AbstractWrapper` 提供了where中包含的所有条件构造方法：

AbstractWrapper

- m getEntity(): T ↑Wrapper
- m setEntity(T): Children
- m getEntityClass(): Class<T>
- m setEntityClass(Class<T>): Children
- m allEq(boolean, Map<R, V>, boolean): Children ↑Compare
- m allEq(boolean, BiPredicate<R, V>, Map<R, V>, boolean): Children ↑Compare
- m eq(boolean, R, Object): Children ↑Compare
- m ne(boolean, R, Object): Children ↑Compare
- m gt(boolean, R, Object): Children ↑Compare
- m ge(boolean, R, Object): Children ↑Compare
- m lt(boolean, R, Object): Children ↑Compare
- m le(boolean, R, Object): Children ↑Compare
- m like(boolean, R, Object): Children ↑Compare
- m notLike(boolean, R, Object): Children ↑Compare
- m likeLeft(boolean, R, Object): Children ↑Compare
- m likeRight(boolean, R, Object): Children ↑Compare
- m notLikeLeft(boolean, R, Object): Children ↑Compare
- m notLikeRight(boolean, R, Object): Children ↑Compare
- m between(boolean, R, Object, Object): Children ↑Compare
- m notBetween(boolean, R, Object, Object): Children ↑Compare
- m and(boolean, Consumer<Children>): Children ↑Nested
- m or(boolean, Consumer<Children>): Children ↑Nested
- m nested(boolean, Consumer<Children>): Children ↑Nested
- m not(boolean, Consumer<Children>): Children ↑Nested

黑马程序员-研究院

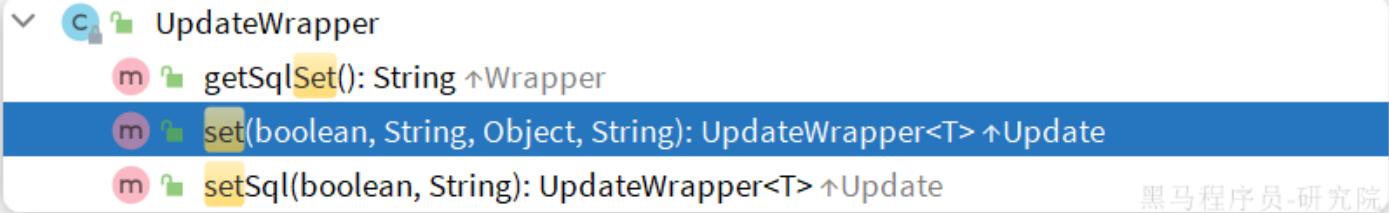
而QueryWrapper在AbstractWrapper的基础上拓展了一个select方法，允许指定查询字段：

QueryWrapper

- m select(String...): QueryWrapper<T> ↑Query
- m select(List<String>): QueryWrapper<T>
- m select(Class<T>, Predicate<TableFieldInfo>): QueryWrapper<T> ↑Query

黑马程序员-研究院

而UpdateWrapper在AbstractWrapper的基础上拓展了一个set方法，允许指定SQL中的SET部分：



接下来，我们就来看看如何利用 `Wrapper` 实现复杂查询。

2.1.1.QueryWrapper

无论是修改、删除、查询，都可以使用QueryWrapper来构建查询条件。接下来看一些例子：

查询：查询出名字中带 o 的，存款大于等于1000元的人。代码如下：

```
1 @Test
2 void testQueryWrapper() {
3     // 1.构建查询条件 where name like "%o%" AND balance >= 1000
4     QueryWrapper<User> wrapper = new QueryWrapper<User>()
5         .select("id", "username", "info", "balance")
6         .like("username", "o")
7         .ge("balance", 1000);
8     // 2.查询数据
9     List<User> users = userMapper.selectList(wrapper);
10    users.forEach(System.out::println);
11 }
```

更新：更新用户名为jack的用户的余额为2000，代码如下：

```
1 @Test
2 void testUpdateByQueryWrapper() {
3     // 1.构建查询条件 where name = "Jack"
4     QueryWrapper<User> wrapper = new QueryWrapper<User>().eq("username",
"Jack");
5     // 2.更新数据, user中非null字段都会作为set语句
6     User user = new User();
7     user.setBalance(2000);
8     userMapper.update(user, wrapper);
9 }
```

update user
set balance = 2000
where username = "jack"

写法2：

```
UpdateWrapper<User> userUpdateWrapper1 = new UpdateWrapper<User>()
    .setSql("balance = 20000")
    .eq("username", "jack");
```

2.1.2.UpdateWrapper

基于BaseMapper中的update方法更新时只能直接赋值，对于一些复杂的需求就难以实现。

例如：更新id为 1, 2, 4 的用户的余额，扣200，对应的SQL应该是：

```
1 UPDATE user SET balance = balance - 200 WHERE id in (1, 2, 4)
```

SET的赋值结果是基于字段现有值的，这个时候就要利用UpdateWrapper中的setSql功能了：

```
1 @Test
2 void testUpdateWrapper() {
3     List<Long> ids = List.of(1L, 2L, 4L);
4     // 1.生成SQL
5     UpdateWrapper<User> wrapper = new UpdateWrapper<User>()
6         .setSql("balance = balance - 200") // SET balance = balance - 200
7         .in("id", ids); // WHERE id in (1, 2, 4)
8     // 2.更新，注意第一个参数可以给null，也就是不填更新字段和数据，
9     // 而是基于UpdateWrapper中的setSQL来更新
10    userMapper.update(null, wrapper);
11 }
```

2.1.3.LambdaQueryWrapper

无论是QueryWrapper还是UpdateWrapper在构造条件的时候都需要写死字段名称，会出现字符串 魔法值。这在编程规范中显然是不推荐的。

那怎么样才能不写字段名，又能知道字段名呢？

其中一种办法是基于变量的 getter 方法结合反射技术。因此我们只要将条件对应的字段的 getter 方法传递给MybatisPlus，它就能计算出对应的变量名了。而传递方法可以使用JDK8中的方法引用 和 Lambda 表达式。

因此MybatisPlus又提供了一套基于Lambda的Wrapper，包含两个：

- LambdaQueryWrapper
- LambdaUpdateWrapper

分别对应QueryWrapper和UpdateWrapper

其使用方式如下：

使用这种方式的好处在于，假如在日后更新了数据库字段名，就不需要修改这里的内容，只需要使用`@TableField`改一下实体类的映射就可以了

```
1 @Test
2 void testLambdaQueryWrapper() {
3     // 1.构建条件 WHERE username LIKE "%o%" AND balance >= 1000
4     QueryWrapper<User> wrapper = new QueryWrapper<>();
5     wrapper.lambda()
6         .select(User::getId, User::getUsername, User::getInfo,
7             User::getBalance)
8         .like(User::getUsername, "o")
9         .ge(User::getBalance, 1000);
10    // 2.查询
11    List<User> users = userMapper.selectList(wrapper);
12    users.forEach(System.out::println);
13 }
```

2.2.自定义SQL

在演示UpdateWrapper的案例中，我们在代码中编写了更新的SQL语句：

```
@Test
void testUpdateWrapper() {
    List<Long> ids = List.of(1L, 2L, 4L);
    // 1.生成SQL
    UpdateWrapper<User> wrapper = new UpdateWrapper<User>()
        .setSql("balance = balance - 200") // SET balance = balance - 200
        .in(column: "id", ids); // WHERE id in (1, 2, 4)

    // 2.更新，注意第一个参数可以给null，也就是不填更新字段和数据，
    // 而是基于UpdateWrapper中的setSQL来更新
    userMapper.update(entity: null, wrapper);
}
```

黑马程序员-研究院

这种写法在某些企业也是不允许的，因为SQL语句最好都维护在持久层，而不是业务层。就当前案例来说，由于条件是in语句，只能将SQL写在Mapper.xml文件，利用foreach来生成动态SQL。这实在是太麻烦了。假如查询条件更复杂，动态SQL的编写也会更加复杂。

所以，MybatisPlus提供了自定义SQL功能，可以让我们利用Wrapper生成查询条件，再结合Mapper.xml编写SQL

2.2.1.基本用法

以当前案例来说，我们可以这样写：

```
使用LambdaQueryWrapper, UpdateWrapper, LambdaUpdateWrapper都可以
LambdaUpdateWrapper<User> wrapper = new LambdaUpdateWrapper<User>()
1 @Test          .in(User::getId, ids);
2 void testCustomWrapper() {
3     // 1.准备自定义查询条件
4     List<Long> ids = List.of(1L, 2L, 4L);
5     QueryWrapper<User> wrapper = new QueryWrapper<User>().in("id", ids);
6
7     // 2.调用mapper的自定义方法，直接传递Wrapper
8     userMapper.deductBalanceByIds(200, wrapper);
9 }
```

然后在UserMapper中自定义SQL：

```
1 package com.itheima.mp.mapper;
2
3 import com.baomidou.mybatisplus.core.mapper.BaseMapper;
4 import com.itheima.mp.domain.po.User;
5 import org.apache.ibatis.annotations.Param;
6 import org.apache.ibatis.annotations.Update;
7 import org.apache.ibatis.annotations.Param;
8 使用`@Update("... ${ew.customSqlSegment}")`也是可以的
9 public interface UserMapper extends BaseMapper<User> {
10     @Select("UPDATE user SET balance = balance - #{money}
11             ${ew.customSqlSegment}")
12     void deductBalanceByIds(@Param("money") int money, @Param("ew")
QueryWrapper<User> wrapper);                                wrapper对象必须使用`@Param("ew")`注解，其他参数可以使用自己的名字
13 }
```

这样就省去了编写复杂查询条件的烦恼了。

2.2.2.多表关联

理论上来讲MyBatisPlus是不支持多表查询的，不过我们可以利用Wrapper中自定义条件结合自定义SQL来实现多表查询的效果。

例如，我们要查询出所有收货地址在北京的并且用户id在1、2、4之中的用户
要是自己基于mybatis实现SQL，大概是这样的：

```
1 <select id="queryUserByIdAndAddr" resultType="com.itheima.mp.domain.po.User">
2     SELECT *
3     FROM user u
4     INNER JOIN address a ON u.id = a.user_id
5     WHERE u.id
6         <foreach collection="ids" separator="," item="id" open="IN (" close=")">
7             #{id}
8         </foreach>
9         AND a.city = #{city}
10    </select>
```

可以看出其中最复杂的就是WHERE条件的编写，如果业务复杂一些，这里的SQL会更变态。

但是基于自定义SQL结合Wrapper的玩法，我们就可以利用Wrapper来构建查询条件，然后手写SELECT及FROM部分，实现多表查询。

查询条件这样来构建：

```
1 @Test
2 void testCustomJoinWrapper() {
3     // 1.准备自定义查询条件
4     QueryWrapper<User> wrapper = new QueryWrapper<User>()
5         .in("u.id", List.of(1L, 2L, 4L))
6         .eq("a.city", "北京");
7
8     // 2.调用mapper的自定义方法
9     List<User> users = userMapper.queryUserByWrapper(wrapper);
10
11    users.forEach(System.out::println);
12 }
```

然后在UserMapper中自定义方法：

```
1 @Select("SELECT u.* FROM user u INNER JOIN address a ON u.id = a.user_id
2         ${ew.customSqlSegment}")
3 List<User> queryUserByWrapper(@Param("ew")QueryWrapper<User> wrapper);
```

当然，也可以在 `UserMapper.xml` 中写SQL：

```
1 <select id="queryUserByIdAndAddr" resultType="com.itheima.mp.domain.po.User">
2     SELECT * FROM user u INNER JOIN address a ON u.id = a.user_id
3     ${ew.customSqlSegment}
4 </select>
```

2.3.Service接口

MybatisPlus不仅提供了BaseMapper，还提供了通用的Service接口及默认实现，封装了一些常用的service模板方法。

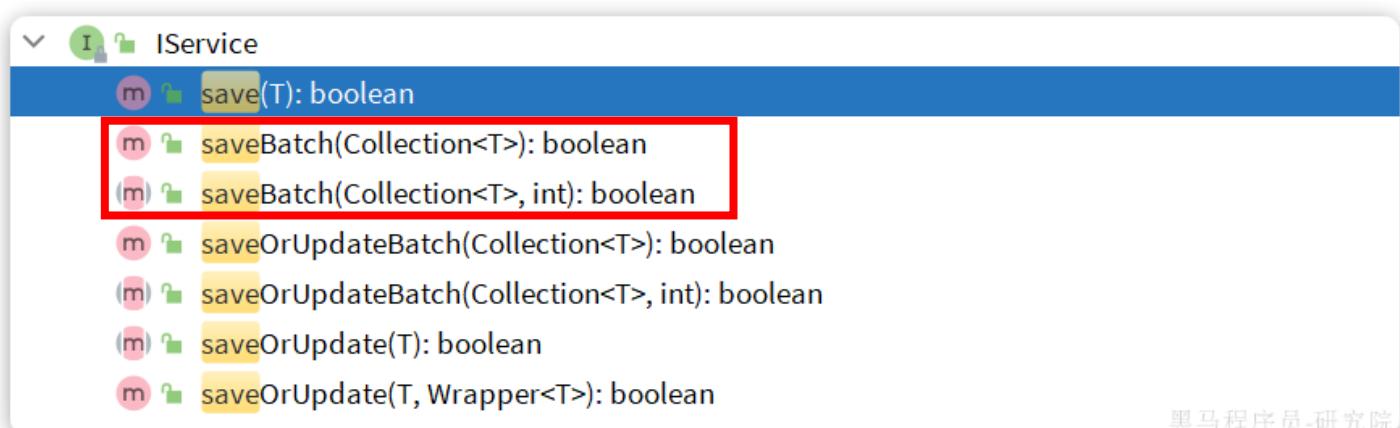
通用接口为 `IService`，默认实现为 `ServiceImpl`，其中封装的方法可以分为以下几类：

- `save`：新增
- `remove`：删除
- `update`：更新
- `get`：查询单个结果
- `list`：查询集合结果
- `count`：计数
- `page`：分页查询

2.3.1.CRUD

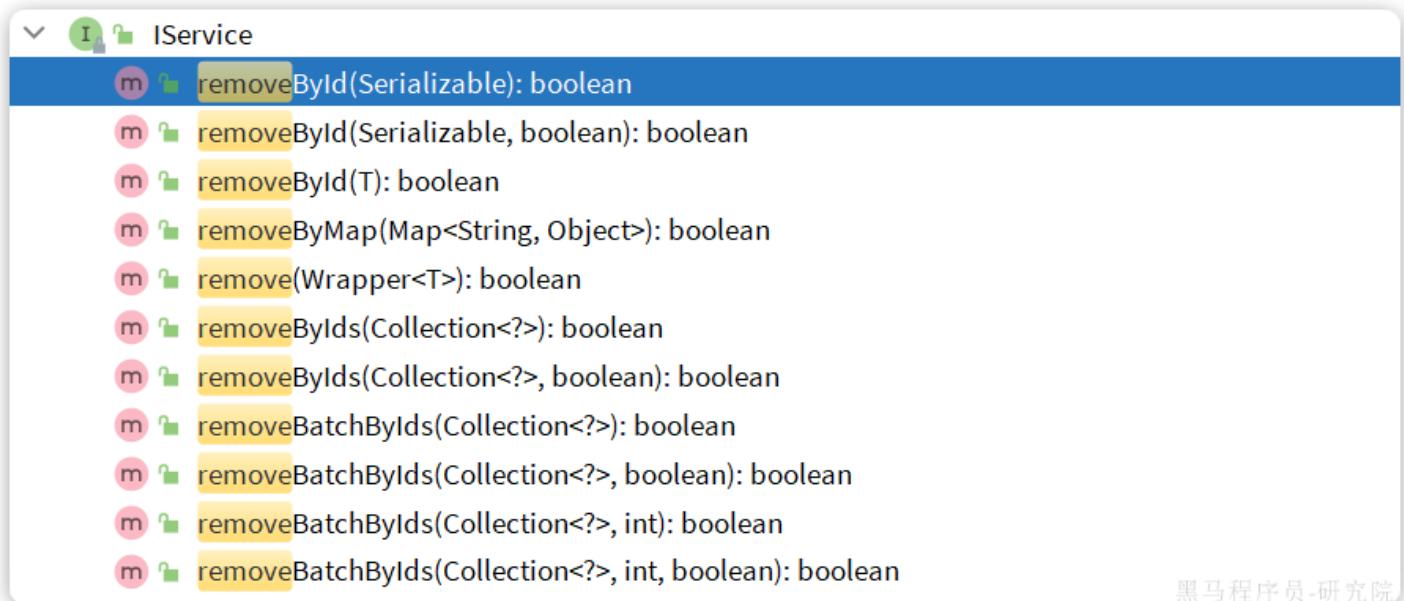
我们先来看看基本的CRUD接口。

新增：



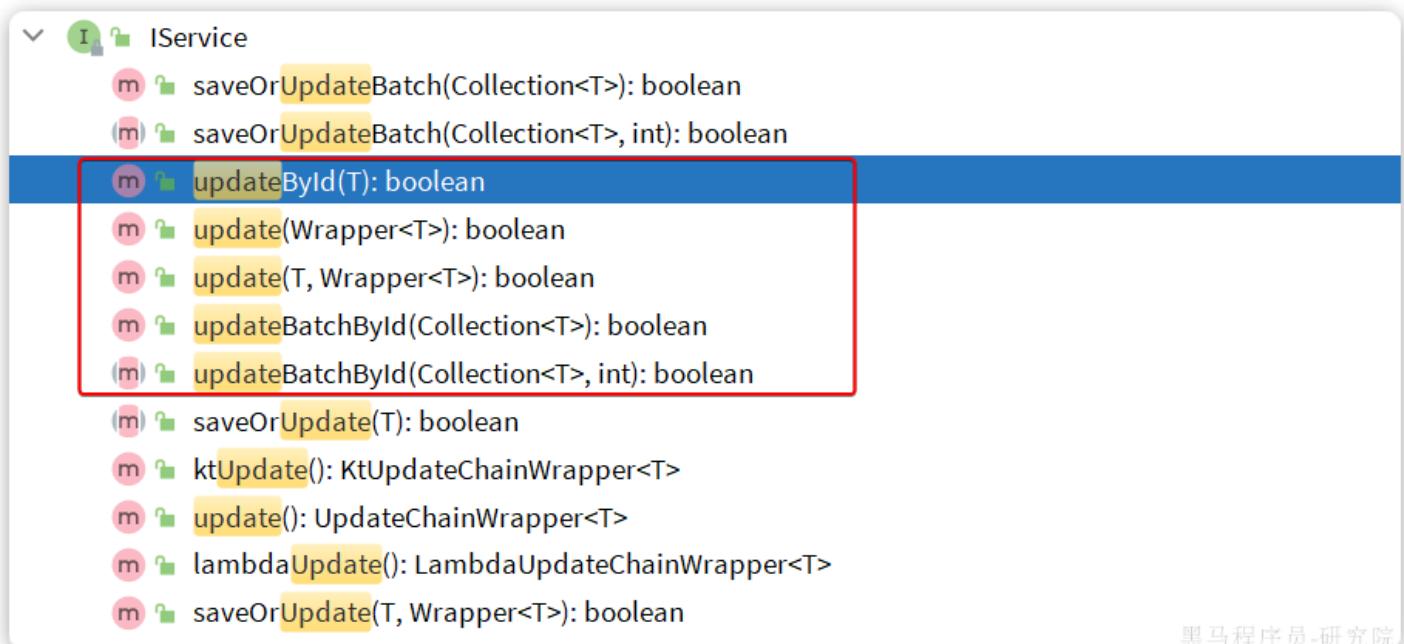
- `save` 是新增单个元素
- `saveBatch` 是批量新增
- `saveOrUpdate` 是根据id判断，如果数据存在就更新，不存在则新增
- `saveOrUpdateBatch` 是批量的新增或修改

删除：



- `removeById` : 根据id删除
- `removeByIds` : 根据id批量删除
- `removeByMap` : 根据Map中的键值对为条件删除
- `remove(Wrapper<T>)` : 根据Wrapper条件删除
- `~~removeBatchByIds~~` : 暂不支持

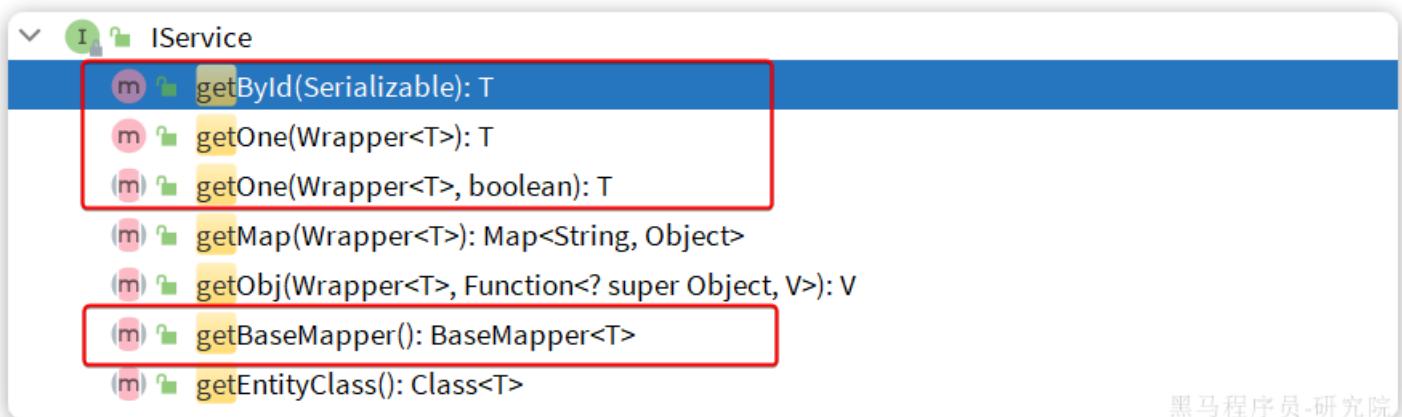
修改：



- `updateById` : 根据id修改

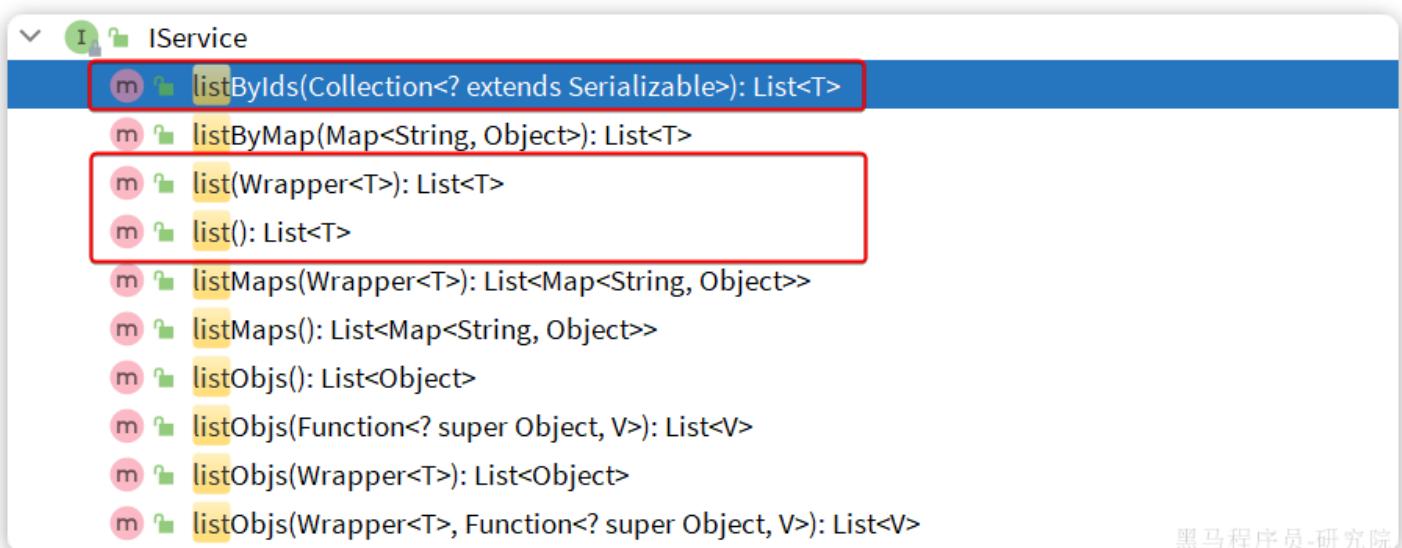
- `update(Wrapper<T>)`：根据 `UpdateWrapper` 修改，`Wrapper` 中包含 `set` 和 `where`
- `update(T, Wrapper<T>)`：按照 `T` 内的数据修改与 `Wrapper` 匹配到的数据
- `updateBatchById`：根据id批量修改

Get:



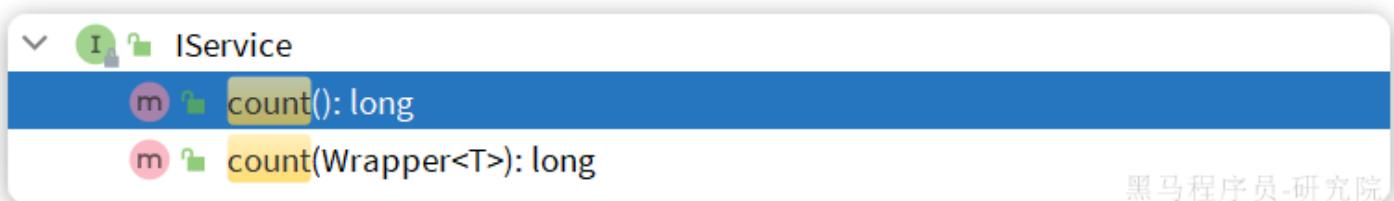
- `getById`：根据id查询1条数据
- `getOne(Wrapper<T>)`：根据 `Wrapper` 查询1条数据
- `getBaseMapper`：获取 `Service` 内的 `BaseMapper` 实现，某些时候需要直接调用 `Mapper` 内的自定义 `SQL` 时可以用这个方法获取到 `Mapper`

List:



- `listByIds`：根据id批量查询
- `list(Wrapper<T>)`：根据 `Wrapper` 条件查询多条数据
- `list()`：查询所有

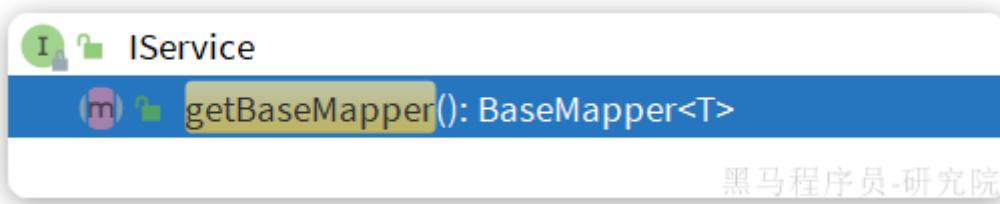
Count:



- `count()` : 统计所有数量
- `count(Wrapper<T>)` : 统计符合 `Wrapper` 条件的数据数量

getBaseMapper:

当我们在service中要调用Mapper中自定义SQL时，就必须获取service对应的Mapper，就可以通过这个方法：



2.3.2. 基本用法

由于 `Service` 中经常需要定义与业务有关的自定义方法，因此我们不能直接使用 `IService`，而是自定义 `Service` 接口，然后继承 `IService` 以拓展方法。同时，让自定义的 `Service` 实现类继承 `ServiceImpl`，这样就不用自己实现 `IService` 中的接口了。

首先，定义 `IUserService`，继承 `IService`：

```
1 package com.itheima.mp.service;
2
3 import com.baomidou.mybatisplus.extension.service.IService;
4 import com.itheima.mp.domain.po.User;
5
6 public interface IUserService extends IService<User> {
7     // 拓展自定义方法
8 }
```

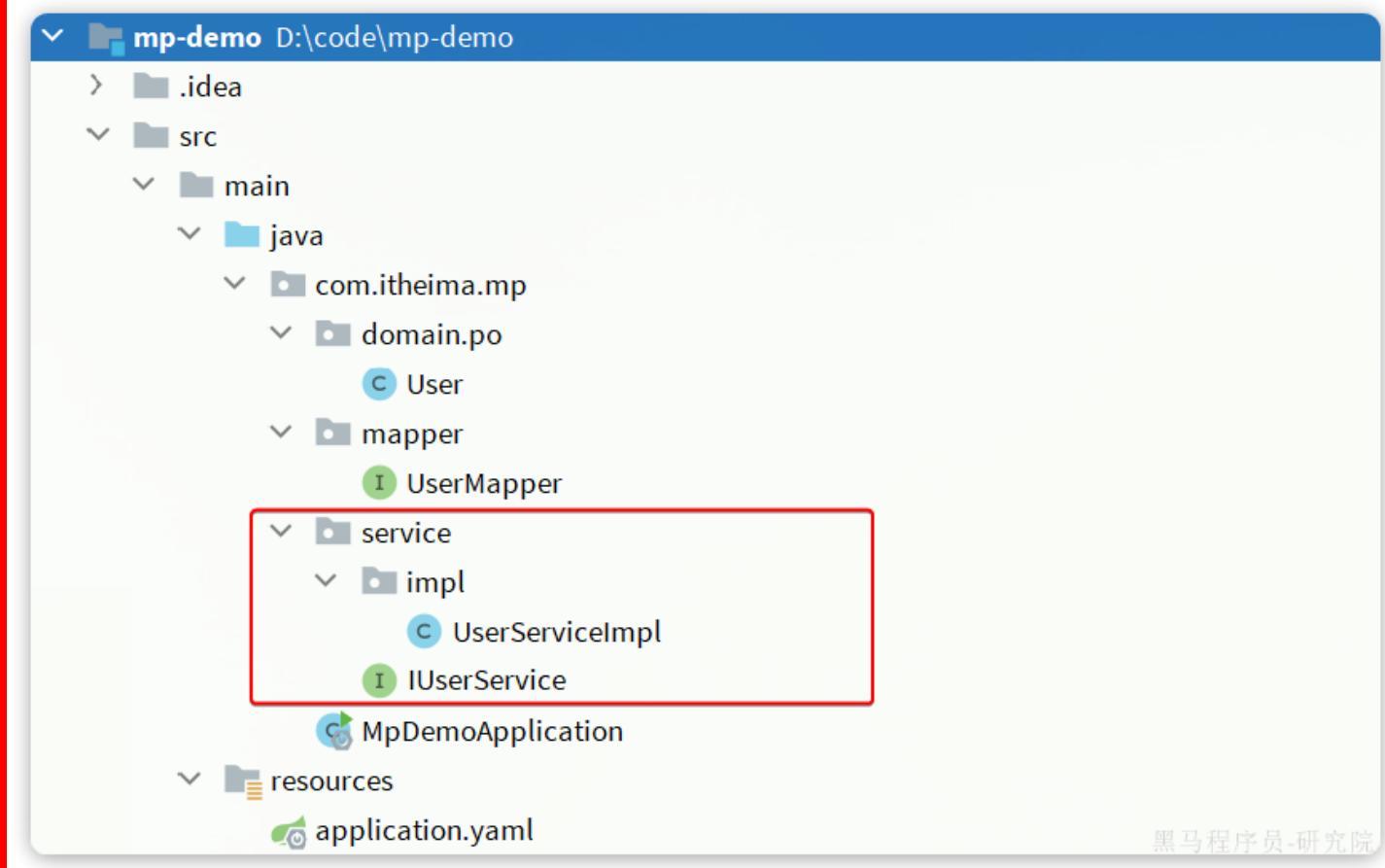
然后，编写 `UserServiceImpl` 类，继承 `ServiceImpl`，实现 `IUserService`：

```

1 package com.itheima.mp.service.impl;
2
3 import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
4 import com.itheima.mp.domain.po.User;
5 import com.itheima.mp.domain.po.service.IUserService;
6 import com.itheima.mp.mapper.UserMapper;
7 import org.springframework.stereotype.Service;
8
9 @Service
10 public class UserServiceImpl extends ServiceImpl<UserMapper, User>
11
12     implements IUserService {

```

项目结构如下：



接下来，我们快速实现下面4个接口：

| 编号 | 接口 | 请求方式 | 请求路径 | 请求参数 | 返回值 |
|----|------|--------|-------------|--------|-----|
| 1 | 新增用户 | POST | /users | 用户表单实体 | 无 |
| 2 | 删除用户 | DELETE | /users/{id} | 用户id | 无 |

| | | | | | |
|---|----------|-----|-------------|--------|--------|
| 3 | 根据id查询用户 | GET | /users/{id} | 用户id | 用户VO |
| 4 | 根据id批量查询 | GET | /users | 用户id集合 | 用户VO集合 |

首先，我们在项目中引入几个依赖：

```

1 <!--swagger-->
2 <dependency>
3   <groupId>com.github.xiaoymin</groupId>
4   <artifactId>knife4j-openapi2-spring-boot-starter</artifactId>
5   <version>4.1.0</version>
6 </dependency>
7 <!--web-->
8 <dependency>
9   <groupId>org.springframework.boot</groupId>
10  <artifactId>spring-boot-starter-web</artifactId>
11 </dependency>
```

然后需要配置swagger信息：

```

1 knife4j:
2   enable: true
3   openapi:
4     title: 用户管理接口文档
5     description: "用户管理接口文档"
6     email: zhanghuyi@itcast.cn
7     concat: 虎哥
8     url: https://www.itcast.cn
9     version: v1.0.0
10    group:
11      default:
12        group-name: default
13        api-rule: package
14        api-rule-resources:
15          - com.itheima.mp.controller
```

然后，接口需要两个实体：

- UserFormDTO：代表新增时的用户表单

- UserVO：代表查询的返回结果
首先是UserFormDTO：

要弄清楚PO、DTO、VO的区别
PO是对应数据库的实体类型，DTO是新增时的用户表单，VO是返回对应的结果

```
1 package com.itheima.mp.domain.dto;
2
3 import com.baomidou.mybatisplus.annotation.TableField;
4 import com.baomidou.mybatisplus.extension.handlers.JacksonTypeHandler;
5 import io.swagger.annotations.ApiModel;
6 import io.swagger.annotations.ApiModelProperty;
7 import lombok.Data;
8
9 @Data
10 @ApiModel(description = "用户表单实体")
11 public class UserFormDTO {
12
13     @ApiModelProperty("id")
14     private Long id;
15
16     @ApiModelProperty("用户名")
17     private String username;
18
19     @ApiModelProperty("密码")
20     private String password;
21
22     @ApiModelProperty("注册手机号")
23     private String phone;
24
25     @ApiModelProperty("详细信息，JSON风格")
26     private String info;
27
28     @ApiModelProperty("账户余额")
29     private Integer balance;
30 }
```

然后是UserVO：

```
1 package com.itheima.mp.domain.vo;
2
3 import io.swagger.annotations.ApiModel;
4 import io.swagger.annotations.ApiModelProperty;
5 import lombok.Data;
6
7 @Data
8 @ApiModel(description = "用户vo实体")
```

```
9 public class UserVO {  
10  
11     @ApiModelProperty("用户id")  
12     private Long id;  
13  
14     @ApiModelProperty("用户名")  
15     private String username;  
16  
17     @ApiModelProperty("详细信息")  
18     private String info;  
19  
20     @ApiModelProperty("使用状态 (1正常 2冻结) ")  
21     private Integer status;  
22  
23     @ApiModelProperty("账户余额")  
24     private Integer balance;  
25 }
```

最后，按照Restful风格编写Controller接口方法：

```
1 package com.itheima.mp.controller;  
2  
3 import cn.hutool.core.bean.BeanUtil;  
4 import com.itheima.mp.domain.dto.UserFormDTO;  
5 import com.itheima.mp.domain.po.User;  
6 import com.itheima.mp.domain.vo.UserVO;  
7 import com.itheima.mp.service.IUserService;  
8 import io.swagger.annotations.Api;  
9 import io.swagger.annotations.ApiOperation;  
10 import lombok.RequiredArgsConstructor;  
11 import org.springframework.web.bind.annotation.*;  
12  
13 import java.util.List;  
14  
15 @Api(tags = "用户管理接口")  
16 @RequiredArgsConstructor  
17 @RestController  
18 @RequestMapping("users")  
19 public class UserController {  
20  
21     private final IUserService userService;  
22  
23     @PostMapping  
24     @ApiOperation("新增用户")  
25     public void saveUser(@RequestBody UserFormDTO userFormDTO){
```

```
26     // 1.转换DTO为PO
27     User user = BeanUtil.copyProperties(userFormDTO, User.class);
28     // 2.新增
29     userService.save(user);
30 }
31
32 @DeleteMapping("/{id}")
33 @ApiOperation("删除用户")
34 public void removeUserById(@PathVariable("id") Long userId){
35     userService.removeById(userId);
36 }
37
38 @GetMapping("/{id}")
39 @ApiOperation("根据id查询用户")
40 public UserVO queryUserById(@PathVariable("id") Long userId{
41     // 1.查询用户
42     User user = userService.getById(userId);
43     // 2.处理vo
44     return BeanUtil.copyProperties(user, UserVO.class);
45 }
46
47 @GetMapping
48 @ApiOperation("根据id集合查询用户")
49 public List<UserVO> queryUserByIds(@RequestParam("ids") List<Long> ids){
50     // 1.查询用户
51     List<User> users = userService.listByIds(ids);
52     // 2.处理vo
53     return BeanUtil.copyToList(users, UserVO.class);
54 }
55 }
```

可以看到上述接口都直接在controller即可实现，无需编写任何service代码，非常方便。

不过，一些带有业务逻辑的接口则需要在service中自定义实现了。例如下面的需求：

- 根据id扣减用户余额

这看起来是个简单修改功能，只要修改用户余额即可。但这个业务包含一些业务逻辑处理：

- 判断用户状态是否正常
- 判断用户余额是否充足

这些业务逻辑都要在service层来做，另外更新余额需要自定义SQL，要在mapper中来实现。因此，我们除了要编写controller以外，具体的业务还要在service和mapper中编写。

首先在UserController中定义一个方法：

```
1 @PutMapping("/{id}/deduction/{money}")
2 @ApiOperation("扣减用户余额")
3 public void deductBalance(@PathVariable("id") Long id,
4                             @PathVariable("money") Integer money){
5     userService.deductBalance(id, money);
6 }
```

然后是UserService接口：

```
1 package com.itheima.mp.service;
2
3 import com.baomidou.mybatisplus.extension.service.IService;
4 import com.itheima.mp.domain.po.User;
5
6 public interface IUserService extends IService<User> {
7     void deductBalance(Long id, Integer money);
8 }
```

最后是UserServiceImpl实现类：

```
1 package com.itheima.mp.service.impl;
2
3 import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
4 import com.itheima.mp.domain.po.User;
5 import com.itheima.mp.mapper.UserMapper;
6 import com.itheima.mp.service.IUserService;
7 import org.springframework.stereotype.Service;
8
9 @Service
10 public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements
11     IUserService {
12     @Override
13     public void deductBalance(Long id, Integer money) {
14         // 1.查询用户
15         User user = getById(id);
16         // 2.判断用户状态
17         if (user.getStatus() == 1) {
18             user.setMoney(user.getMoney() - money);
19             updateById(user);
20         }
21     }
22 }
```

```
16     if (user == null || user.getStatus() == 2) {  
17         throw new RuntimeException("用户状态异常");  
18     }  
19     // 3. 判断用户余额  
20     if (user.getBalance() < money) {  
21         throw new RuntimeException("用户余额不足");  
22     }  
23     // 4. 扣减余额  
24     baseMapper.deductMoneyById(id, money);  
25 }  
26 }
```

最后是mapper:

```
1 @Update("UPDATE user SET balance = balance - #{money} WHERE id = #{id}")  
2 void deductMoneyById(@Param("id") Long id, @Param("money") Integer money);
```

2.3.3.Lambda

IService中还提供了Lambda功能来简化我们的复杂查询及更新功能。我们通过两个案例来学习一下。

案例一：实现一个根据复杂条件查询用户的接口，查询条件如下：

- name: 用户名关键字，可以为空
- status: 用户状态，可以为空
- minBalance: 最小余额，可以为空
- maxBalance: 最大余额，可以为空

可以理解成一个用户的后台管理界面，管理员可以自己选择条件来筛选用户，因此上述条件不一定存在，需要做判断。

我们首先需要定义一个查询条件实体，UserQuery实体：

```
1 package com.itheima.mp.domain.query;  
2  
3 import io.swagger.annotations.ApiModel;  
4 import io.swagger.annotations.ApiModelProperty;  
5 import lombok.Data;
```

```
6  
7 @Data  
8 @ApiModel(description = "用户查询条件实体")  
9 public class UserQuery {  
10     @ApiModelProperty("用户名关键字")  
11     private String name;  
12     @ApiModelProperty("用户状态: 1-正常, 2-冻结")  
13     private Integer status;  
14     @ApiModelProperty("余额最小值")  
15     private Integer minBalance;  
16     @ApiModelProperty("余额最大值")  
17     private Integer maxBalance;  
18 }
```

接下来我们在UserController中定义一个controller方法：

```
1 @GetMapping("/list")  
2 @ApiOperation("根据id集合查询用户")  
3 public List<UserVO> queryUsers(UserQuery query){  
4     // 1.组织条件  
5     String username = query.getName();  
6     Integer status = query.getStatus();  
7     Integer minBalance = query.getMinBalance();  
8     Integer maxBalance = query.getMaxBalance();  
9     LambdaQueryWrapper<User> wrapper = new QueryWrapper<User>().lambda()  
10         .like(username != null, User::getUsername, username)  
11         .eq(status != null, User::getStatus, status)  
12         .ge(minBalance != null, User::getBalance, minBalance)  
13         .le(maxBalance != null, User::getBalance, maxBalance);  
14     // 2.查询用户  
15     List<User> users = userService.list(wrapper);  
16     // 3.处理vo  
17     return BeanUtil.copyToList(users, UserVO.class);  
18 }
```

在组织查询条件的时候，我们加入了 `username != null` 这样的参数，意思就是当条件成立时才会添加这个查询条件，类似Mybatis的mapper.xml文件中的 `<if>` 标签。这样就实现了动态查询条件效果了。

不过，上述条件构建的代码太麻烦了。

因此Service中对 `LambdaQueryWrapper` 和 `LambdaUpdateWrapper` 的用法进一步做了简化。

我们无需自己通过 `new` 的方式来创建 `Wrapper`，而是直接调用 `lambdaQuery` 和 `lambdaUpdate` 方法：

基于Lambda查询：

```
1 @GetMapping("/list")
2 @ApiOperation("根据id集合查询用户")
3 public List<UserVO> queryUsers(UserQuery query){
4     // 1.组织条件
5     String username = query.getName();
6     Integer status = query.getStatus();
7     Integer minBalance = query.getMinBalance();
8     Integer maxBalance = query.getMaxBalance();
9     // 2.查询用户
10    List<User> users = userService.lambdaQuery()
11        .like(username != null, User::getUsername, username)
12        .eq(status != null, User::getStatus, status)
13        .ge(minBalance != null, User::getBalance, minBalance)
14        .le(maxBalance != null, User::getBalance, maxBalance)
15        .list();
16    // 3.处理vo
17    return BeanUtil.copyToList(users, UserVO.class);
18 }
```

可以发现`lambdaQuery`方法中除了可以构建条件，还需要在链式编程的最后添加一个 `list()`，这是在告诉MP我们的调用结果需要是一个list集合。这里不仅可以用 `list()`，可选的方法有：

- `.one()`：最多1个结果
- `.list()`：返回集合结果
- `.count()`：返回计数结果

MybatisPlus会根据链式编程的最后一个方法来判断最终的返回结果。

与`lambdaQuery`方法类似，IService中的`lambdaUpdate`方法可以非常方便的实现复杂更新业务。

例如下面的需求：

需求：改造根据id修改用户余额的接口，要求如下

- 如果扣减后余额为0，则将用户status修改为冻结状态 (2)

也就是说我们在扣减用户余额时，需要对用户剩余余额做出判断，如果发现剩余余额为0，则应该将status修改为2，这就是说update语句的set部分是动态的。

实现如下：

```
1 @Override
2 @Transactional
3 public void deductBalance(Long id, Integer money) {
4     // 1.查询用户
5     User user = getById(id);
6     // 2.校验用户状态
7     if (user == null || user.getStatus() == 2) {
8         throw new RuntimeException("用户状态异常！");
9     }
10    // 3.校验余额是否充足
11    if (user.getBalance() < money) {
12        throw new RuntimeException("用户余额不足！");
13    }
14    // 4.扣减余额 update tb_user set balance = balance - ?
15    int remainBalance = user.getBalance() - money;
16    lambdaUpdate()
17        .set(User::getBalance, remainBalance) // 更新余额
18        .set(remainBalance == 0, User::getStatus, 2) // 动态判断，是否更新
19        .eq(User::getId, id)
20        .eq(User::getBalance, user.getBalance()) // 乐观锁
21        .update();
22 }
```

2.3.4.批量新增

IService中的批量新增功能使用起来非常方便，但有一点注意事项，我们先来测试一下。

首先我们测试逐条插入数据：

```
1 @Test
2 void testSaveOneByOne() {
3     long b = System.currentTimeMillis();
4     for (int i = 1; i <= 100000; i++) {
5         userService.save(buildUser(i));
6     }
7     long e = System.currentTimeMillis();
```

```

8     System.out.println("耗时: " + (e - b));
9 }
10
11 private User buildUser(int i) {
12     User user = new User();
13     user.setUsername("user_" + i);
14     user.setPassword("123");
15     user.setPhone("'" + (18688190000L + i));
16     user.setBalance(2000);
17     user.setInfo("{\"age\": 24, \"intro\": \"英文老师\", \"gender\": "
18         +"\"female\"}");
19     user.setCreateTime(LocalDateTime.now());
20     user.setUpdateTime(user.getCreateTime());
21     return user;

```

执行结果如下：



可以看到速度非常慢。

然后再试试 MybatisPlus 的批处理：

```

1 @Test
2 void testSaveBatch() {
3     // 准备10万条数据
4     List<User> list = new ArrayList<>(1000);
5     long b = System.currentTimeMillis();
6     for (int i = 1; i <= 100000; i++) {
7         list.add(buildUser(i));
8         // 每1000条批量插入一次
9         if (i % 1000 == 0) {
10             userService.saveBatch(list);
11             list.clear();
12         }
13     }
14     long e = System.currentTimeMillis();
15     System.out.println("耗时: " + (e - b));

```

```
16 }
```

执行最终耗时如下：



可以看到使用了批处理以后，比逐条新增效率提高了10倍左右，性能还是不错的。

不过，我们简单查看一下 MybatisPlus 源码：

```
1 @Transactional(rollbackFor = Exception.class)
2 @Override
3 public boolean saveBatch(Collection<T> entityList, int batchSize) {
4     String sqlStatement = getSqlStatement(SqlMethod.INSERT_ONE);
5     return executeBatch(entityList, batchSize, (sqlSession, entity) ->
6         sqlSession.insert(sqlStatement, entity));
7 }
8 // ...SqlHelper
9 public static <E> boolean executeBatch(Class<?> entityClass, Log log,
10     Collection<E> list, int batchSize, BiConsumer<SqlSession, E> consumer) {
11     Assert.isFalse(batchSize < 1, "batchSize must not be less than one");
12     return !CollectionUtils.isEmpty(list) && executeBatch(entityClass, log,
13         sqlSession -> {
14             int size = list.size();
15             int idxLimit = Math.min(batchSize, size);
16             int i = 1;
17             for (E element : list) {
18                 consumer.accept(sqlSession, element);
19                 if (i == idxLimit) {
20                     sqlSession.flushStatements();
21                     idxLimit = Math.min(idxLimit + batchSize, size);
22                 }
23                 i++;
24             }
25         });
26     });
27 }
```

可以发现其实 MybatisPlus 的批处理是基于 `PreparedStatement` 的预编译模式，然后批量提交，最终在数据库执行时还是会有多条insert语句，逐条插入数据。SQL类似这样：

```
1 Preparing: INSERT INTO user ( username, password, phone, info, balance,
    create_time, update_time ) VALUES ( ?, ?, ?, ?, ?, ?, ?, ? )
2 Parameters: user_1, 123, 18688190001, "", 2000, 2023-07-01, 2023-07-01
3 Parameters: user_2, 123, 18688190002, "", 2000, 2023-07-01, 2023-07-01
4 Parameters: user_3, 123, 18688190003, "", 2000, 2023-07-01, 2023-07-01
```

而如果想要得到最佳性能，最好是将多条SQL合并为一条，像这样：

```
1 INSERT INTO user ( username, password, phone, info, balance, create_time,
    update_time )
2 VALUES
3 (user_1, 123, 18688190001, "", 2000, 2023-07-01, 2023-07-01),
4 (user_2, 123, 18688190002, "", 2000, 2023-07-01, 2023-07-01),
5 (user_3, 123, 18688190003, "", 2000, 2023-07-01, 2023-07-01),
6 (user_4, 123, 18688190004, "", 2000, 2023-07-01, 2023-07-01);
```

该怎么做呢？

MySQL的客户端连接参数中有这样的一个参数：`rewriteBatchedStatements`。顾名思义，就是重写批处理的 `statement` 语句。参考文档：

https://dev.mysql.com/doc/connector-j/8.0/en/connector-j-connp-props-performance-extensions.html#cj-conn-prop_rewriteBatchedStatements
dev.mysql.com

这个参数的默认值是false，我们需要修改连接参数，将其配置为true

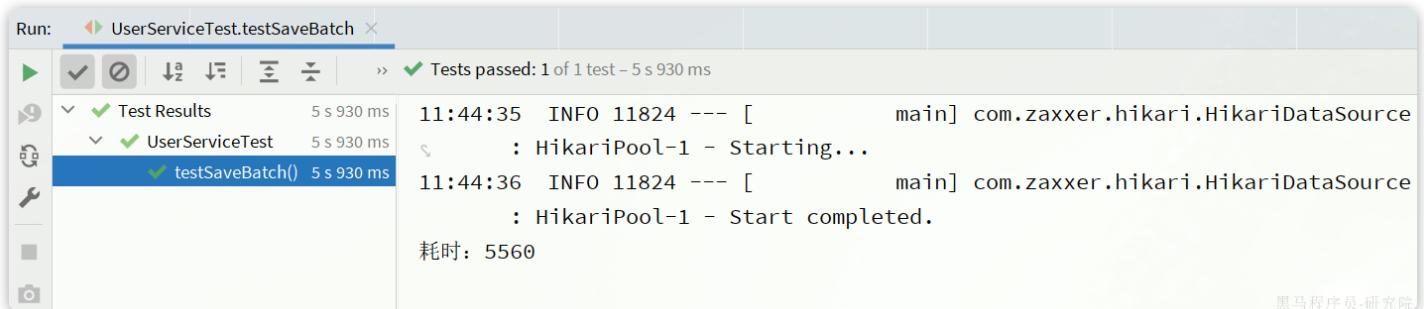
修改项目中的application.yml文件，在jdbc的url后面添加参数

`&rewriteBatchedStatements=true`：

```
1 spring:
```

```
2 datasource:  
3     url: jdbc:mysql://127.0.0.1:3306/mp?useUnicode=true&characterEncoding=UTF-  
8&autoReconnect=true&serverTimezone=Asia/Shanghai&rewriteBatchedStatements=true  
4     driver-class-name: com.mysql.cj.jdbc.Driver  
5     username: root  
6     password: MySQL123
```

再次测试插入10万条数据，可以发现速度有非常明显的提升：



在 `ClientPreparedStatement` 的 `executeBatchInternal` 中，有判断 `rewriteBatchedStatements` 值是否为true并重写SQL的功能：

最终，SQL被重写了：

```
com.mysql.cj.jdbc.ClientPreparedStatement: INSERT INTO user (username,  
password,  
phone,  
info,  
  
balance,  
create_time,  
update_time) VALUES ('user_1',  
'123',  
'18688190001',  
'{"age": 24, "intro": "英文老师", "gender": "female"}',  
  
2000,  
'2023-07-01 14:47:31.063882',  
'2023-07-01 14:47:31.063882'), ('user_2',  
'123',  
'18688190002',  
'{"age": 24, "intro": "英文老师", "gender": "female"}',  
  
2000,  
'2023-07-01 14:47:31.063882',  
'2023-07-01 14:47:31.063882'), ('user_3',  
'123'
```

3. 扩展功能

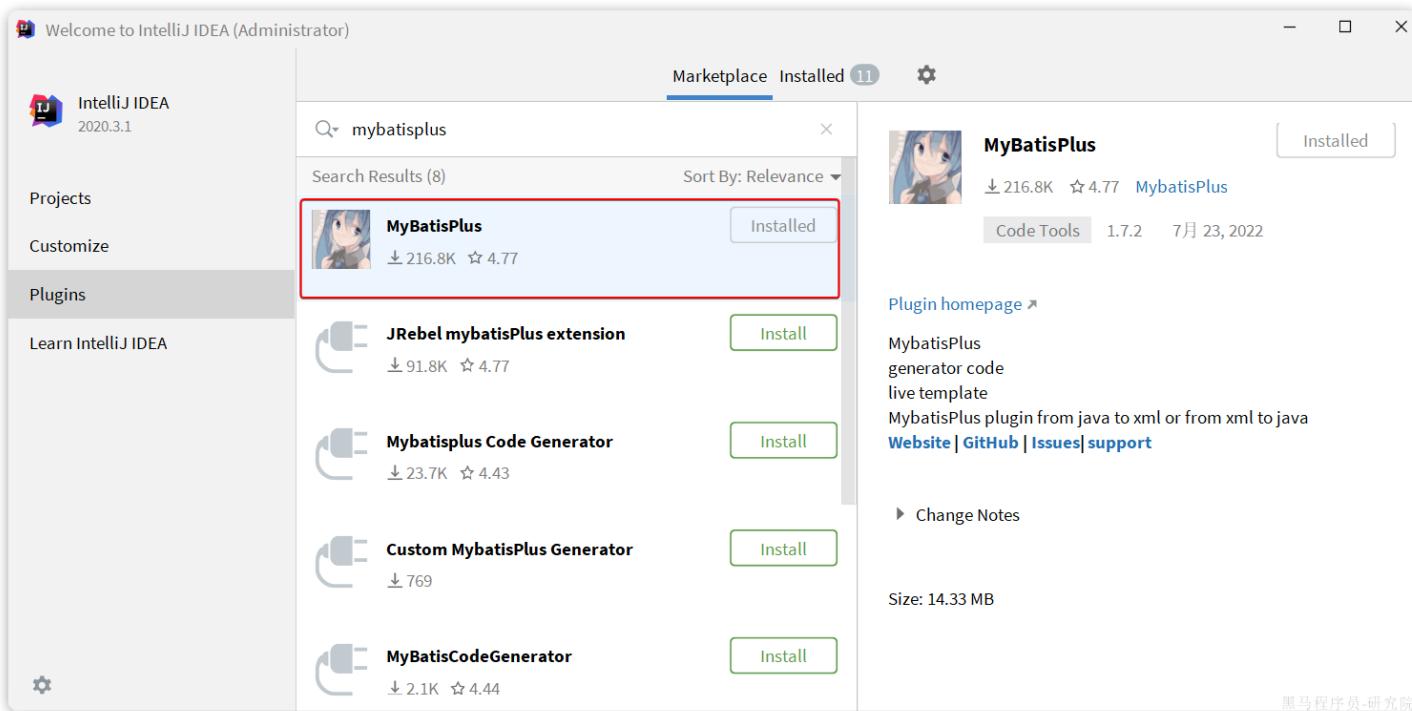
3.1. 代码生成

在使用MybatisPlus以后，基础的 `Mapper`、`Service`、`PO` 代码相对固定，重复编写也比较麻烦。因此MybatisPlus官方提供了代码生成器根据数据库表结构生成 `PO`、`Mapper`、`Service` 等相关代码。只不过代码生成器同样要编码使用，也很麻烦。

这里推荐大家使用一款 `MybatisPlus` 的插件，它可以基于图形化界面完成 `MybatisPlus` 的代码生成，非常简单。

3.1.1. 安装插件

在 `Idea` 的 `plugins` 市场中搜索并安装 `MyBatisPlus` 插件：

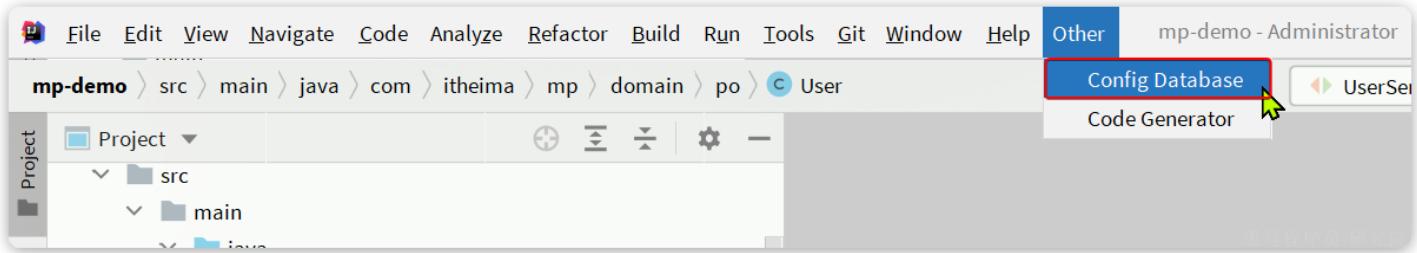


然后重启你的Idea即可使用。

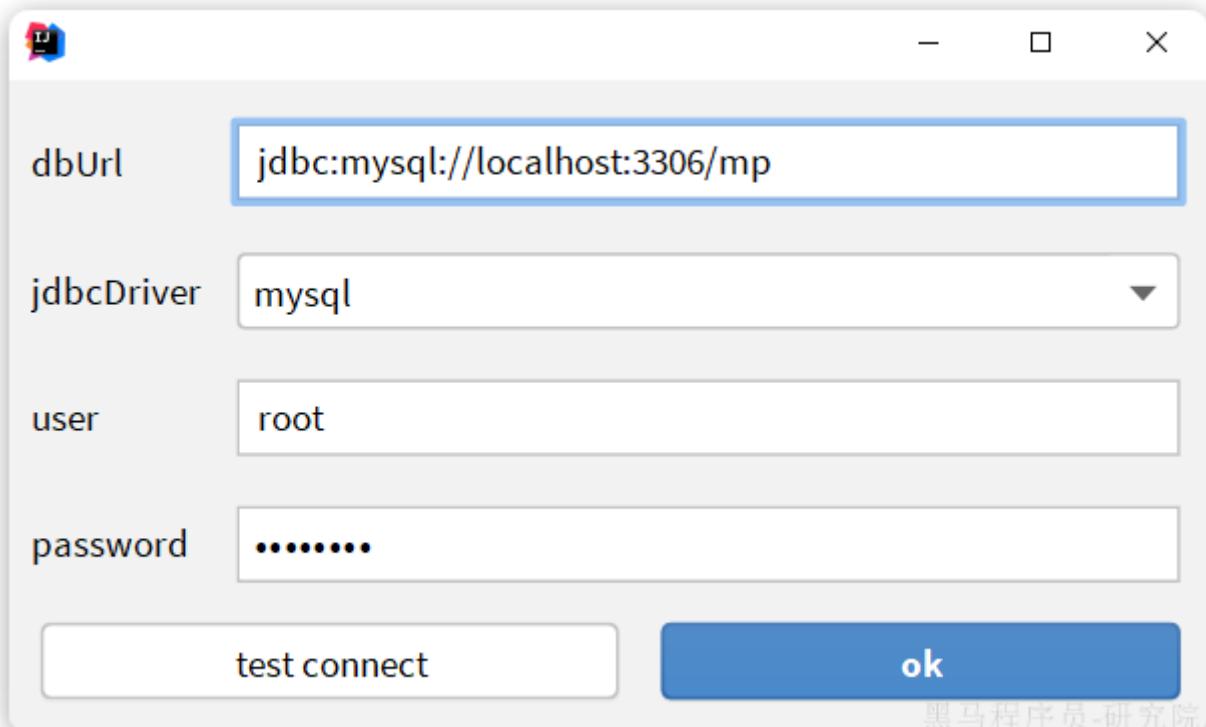
3.1.2. 使用

刚好数据库中还有一张address表尚未生成对应的实体和mapper等基础代码。我们利用插件生成一下。

首先需要配置数据库地址，在Idea顶部菜单中，找到 `other`，选择 `Config Database`：

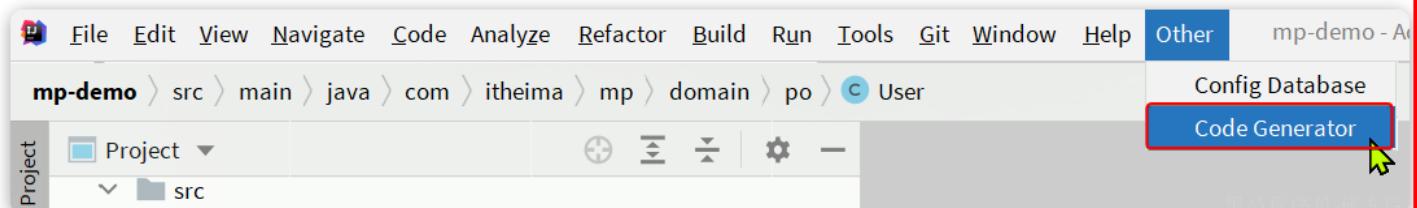


在弹出的窗口中填写数据库连接的基本信息：



点击OK保存。

然后再次点击Idea顶部菜单中的other，然后选择 Code Generator：



在弹出的表单中填写信息：

| table name | crea... | engine | coding | remark |
|------------|---------|--------|-----------------|-------------------|
| address | 202... | InnoDB | utf8_general_ci | |
| user | 202... | InnoDB | utf8_general_ci | 用户表 选中需要生成代码的表 |

module 由于我们只有Project没有module, 所以这里不填 package com.itheima.mp 填写代码生成的父级包路径

author 虎哥 over file AUTO(ID自增) Id策略, 与数据库表id策略保持一致

Entity domain.po 实体类的包 Mapper mapper Mapper所在包 Controller controller controller所在包, 非Web项目不要勾选

Service service service接口所在包 ServicImpl service.impl service实现类所在包 TablePrefix

lombok restController swagger ResultMap is fill is enable cache is base column

是否基于 lombok注解 是否生成swagger 注解

save check field code generatio 提交

最终，代码自动生成到指定的位置了：

3.2.静态工具

有的时候Service之间也会相互调用，为了避免出现循环依赖问题，MybatisPlus提供一个静态工具类：Db，其中的一些静态方法与IService中方法签名基本一致，也可以帮助我们实现CRUD功能：

只是需要在一些方法使用的时候，传入所需的参数类的class即可

Db

- m Db()
- m save(T): boolean
- m saveBatch(Collection<T>): boolean
- m saveBatch(Collection<T>, int): boolean
- m saveOrUpdateBatch(Collection<T>): boolean
- m saveOrUpdateBatch(Collection<T>, int): boolean
- m removeById(Serializable, Class<T>): boolean
- m removeById(T): boolean
- m remove(AbstractWrapper<T, ?, ?>): boolean
- m updateById(T): boolean
- m update(AbstractWrapper<T, ?, ?>): boolean
- m update(T, AbstractWrapper<T, ?, ?>): boolean
- m updateBatchById(Collection<T>): boolean
- m updateBatchById(Collection<T>, int): boolean
- m removeByIds(Collection<? extends Serializable>, Class<T>): boolean
- m removeByMap(Map<String, Object>, Class<T>): boolean
- m saveOrUpdate(T): boolean
- m getById(Serializable, Class<T>): T
- m getOne(AbstractWrapper<T, ?, ?>): T
- m getOne(AbstractWrapper<T, ?, ?, boolean>): T
- m listByMap(Map<String, Object>, Class<T>): List<T>
- m listByIds(Collection<? extends Serializable>, Class<T>): List<T>
- m getMap(AbstractWrapper<T, ?, ?>): Map<String, Object>
- m count(Class<T>): long
- m count(AbstractWrapper<T, ?, ?>): long
- m list(AbstractWrapper<T, ?, ?>): List<T>
- m list(Class<T>): List<T>

黑马程序员-研究院

示例：

```
1 @Test
2 void testDbGet() {
3     User user = Db.getById(1L, User.class);
4     System.out.println(user);
5 }
6
7 @Test
8 void testDbList() {
9     // 利用Db实现复杂条件查询
10    List<User> list = Db.lambdaQuery(User.class)
11        .like(User::getUsername, "o")
12        .ge(User::getBalance, 1000)
13        .list();
14    list.forEach(System.out::println);
15 }
```

```
16
17 @Test
18 void testDbUpdate() {
19     Db.lambdaUpdate(User.class)
20         .set(User::getBalance, 2000)
21         .eq(User::getUsername, "Rose");
22 }
```

在之前的方法中，userService中还需要使用到addressService / addressMapper，导致可能出现循环依赖问题，所以使用Db静态工具类来解决这个问题

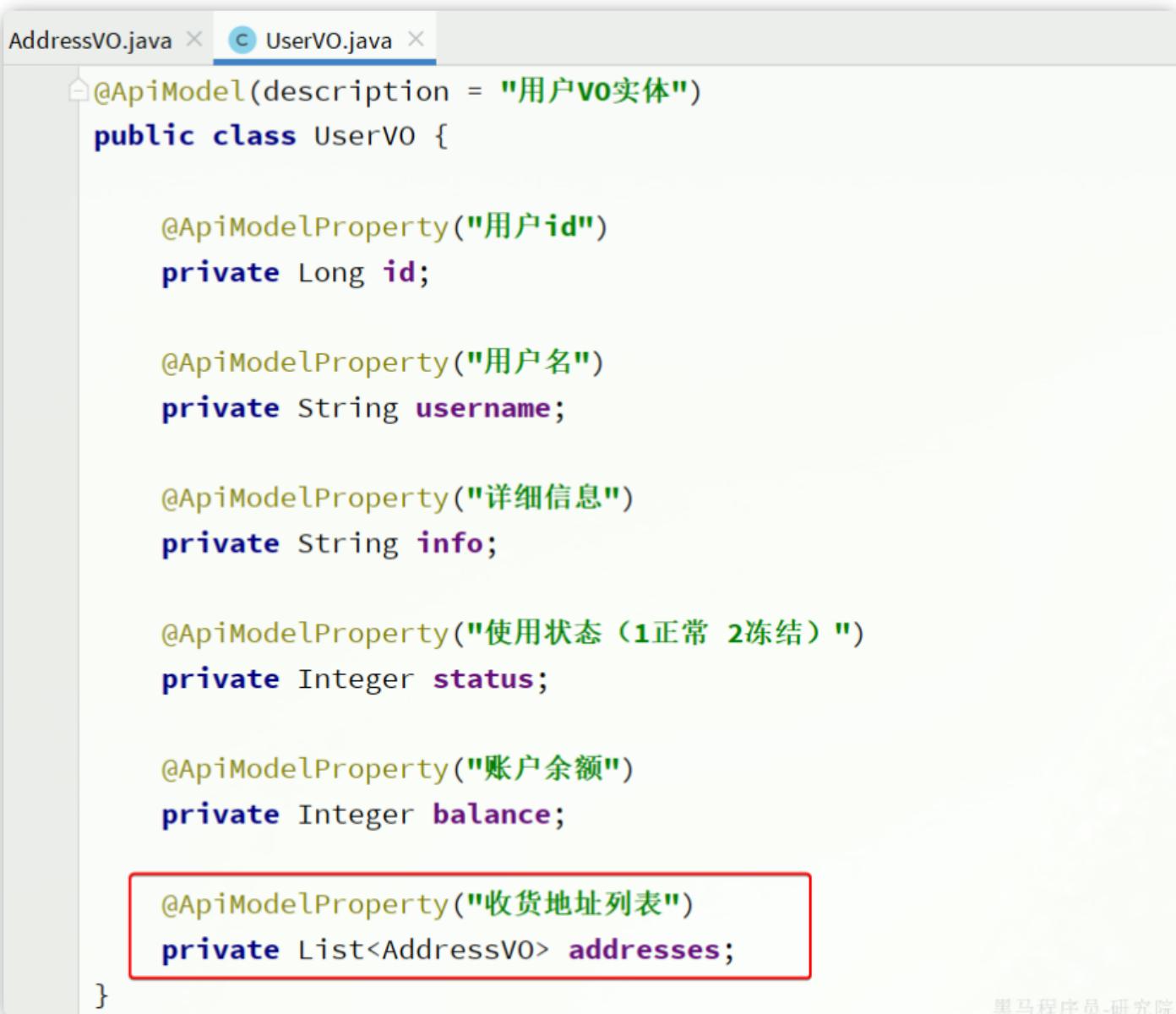
需求：改造根据id用户查询的接口，查询用户的同时返回用户收货地址列表

首先，我们要添加一个收货地址的VO对象：

```
1 package com.itheima.mp.domain.vo;
2
3 import io.swagger.annotations.ApiModel;
4 import io.swagger.annotations.ApiModelProperty;
5 import lombok.Data;
6
7 @Data
8 @ApiModel(description = "收货地址VO")
9 public class AddressVO{
10
11     @ApiModelProperty("id")
12     private Long id;
13
14     @ApiModelProperty("用户ID")
15     private Long userId;
16
17     @ApiModelProperty("省")
18     private String province;
19
20     @ApiModelProperty("市")
21     private String city;
22
23     @ApiModelProperty("县/区")
24     private String town;
25
26     @ApiModelProperty("手机")
27     private String mobile;
28
29     @ApiModelProperty("详细地址")
30     private String street;
31 }
```

```
32     @ApiModelProperty("联系人")
33     private String contact;
34
35     @ApiModelProperty("是否是默认 1默认 0否")
36     private Boolean isDefault;
37
38     @ApiModelProperty("备注")
39     private String notes;
40 }
```

然后，改造原来的UserVO，添加一个地址属性：



```
AddressVO.java ×  UserVO.java ×
@ApiModelProperty(description = "用户vo实体")
public class UserVO {

    @ApiModelProperty("用户id")
    private Long id;

    @ApiModelProperty("用户名")
    private String username;

    @ApiModelProperty("详细信息")
    private String info;

    @ApiModelProperty("使用状态（1正常 2冻结）")
    private Integer status;

    @ApiModelProperty("账户余额")
    private Integer balance;

    @ApiModelProperty("收货地址列表")
    private List<AddressVO> addresses;
}
```

接下来，修改UserController中根据id查询用户的业务接口：

```
1 @GetMapping("/{id}")
2 @ApiOperation("根据id查询用户")
3 public UserVO queryUserById(@PathVariable("id") Long userId){
4     // 基于自定义service方法查询
5     return userService.queryUserAndAddressById(userId);
6 }
```

由于查询业务复杂，所以要在service层来实现。首先在IUserService中定义方法：

```
1 package com.itheima.mp.service;
2
3 import com.baomidou.mybatisplus.extension.service.IService;
4 import com.itheima.mp.domain.po.User;
5 import com.itheima.mp.domain.vo.UserVO;
6
7 public interface IUserService extends IService<User> {
8     void deduct(Long id, Integer money);
9
10    UserVO queryUserAndAddressById(Long userId);
11 }
```

然后，在UserServiceImpl中实现该方法：

```
1 @Override
2 public UserVO queryUserAndAddressById(Long userId) {
3     // 1.查询用户
4     User user = getById(userId);
5     if (user == null) {
6         return null;
7     }
8     // 2.查询收货地址
9     List<Address> addresses = Db.lambdaQuery(Address.class)
10         .eq(Address::getUserID, userId)
11         .list();
12     // 3.处理vo
13     UserVO userVO = BeanUtil.copyProperties(user, UserVO.class);
14     userVO.setAddresses(BeanUtil.copyToList(addresses, AddressVO.class));
15     return userVO;
16 }
```

在查询地址时，我们采用了Db的静态方法，因此避免了注入AddressService，减少了循环依赖的风险。

再来实现一个功能：

- 根据id批量查询用户，并查询出用户对应的所有地址

思路为：

- 根据传入的ids，先批量查询出这些用户users
- 将查询出的用户id，提取到一个userIds列表中：users.stream().map(User::getId).collect(Collectors.toList);
- 使用Db来查询这些ID的地址信息列表addresses：Db.IambdaQuery(Address.class).in(Address::getId, userIds).list()，并使用BeanUtil转成AddressVO的形式：BeanUtil.copyToList(addresses, AddressVO.class)
- 由于我们还要将这些地址信息，分配到指定的用户UserVO中，所以我们要将同一个用户的地址信息存储到同一个集合addressMap中：addresses.stream().collect(Collectors.groupingBy(AddressVO::getUserId)); 其中，key为UserId，value为该用户的地址集合
- 建立一个UserVO列表，将users通过BeanUtil.copyToList转换成usersVO
- 使用foreach遍历usersVO，对于每一个对象设置地址：uservo.setAddress(addressMap.get(uservo.getId()))

3.3.逻辑删除

对于一些比较重要的数据，我们往往采用逻辑删除的方案，即：

- 在表中添加一个字段标记数据是否被删除
- 当删除数据时把标记置为true
- 查询时过滤掉标记为true的数据

一旦采用了逻辑删除，所有的查询和删除逻辑都要跟着变化，非常麻烦。

为了解决这个问题，MybatisPlus就添加了对逻辑删除的支持。

！ 注意，只有MybatisPlus生成的SQL语句才支持自动的逻辑删除，自定义SQL需要自己手动处理逻辑删除。

例如，我们给 address 表添加一个逻辑删除字段：

```
1 alter table address add deleted bit default b'0' null comment '逻辑删除';
```

然后给 Address 实体添加 deleted 字段：

Address.java

```
是否是默认 1默认 0否  
69   private Boolean isDefault;  
70  
71   备注  
72   private String notes;  
73  
74   逻辑删除  
75   private Boolean deleted;  
76 }  
77  
81 |
```

黑马程序员-研究院

接下来，我们要在 `application.yml` 中配置逻辑删除字段：

```
1 mybatis-plus:  
2   global-config:  
3     db-config:  
4       logic-delete-field: deleted # 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)
```

测试：

首先，我们执行一个删除操作：

```
1 @Test  
2 void testDeleteByLogic() {  
3     // 删除方法与以前没有区别  
4     addressService.removeById(59L);  
5 }
```

方法与普通删除一模一样，但是底层的SQL逻辑变了：

```
17:18:19 INFO 24844 --- [           main] com.zaxxer.hikari.HikariDataSource      : HikariPool-1?
  - Starting...
17:18:20 INFO 24844 --- [           main] com.zaxxer.hikari.HikariDataSource      : HikariPool-1
  - Start completed.
17:18:20 DEBUG 24844 --- [           main] c.i.mp.mapper.AddressMapper.deleteById   : ==>
  Preparing: UPDATE address SET deleted=1 WHERE id=? AND deleted=0
17:18:20 DEBUG 24844 --- [           main] c.i.mp.mapper.AddressMapper.deleteById   : ==>
  Parameters: 59(Long)
17:18:20 DEBUG 24844 --- [           main] c.i.mp.mapper.AddressMapper.deleteById   : <==
  Updates: 1
```

黑马程序员·研究院

查询一下试试：

```
1 @Test
2 void testQuery() {
3     List<Address> list = addressService.list();
4     list.forEach(System.out::println);
5 }
```

会发现id为59的确实没有查询出来，而且SQL中也对逻辑删除字段做了判断：

```
17:19:39 INFO 23528 --- [           main] com.zaxxer.hikari.HikariDataSource      : HikariPool-1
  - Start completed.
17:19:39 DEBUG 23528 --- [           main] c.i.mp.mapper.AddressMapper.selectList    : ==>
  Preparing: SELECT id,user_id,province,city,town,mobile,street,contact,is_default,notes,deleted
  FROM address WHERE deleted=0
17:19:39 DEBUG 23528 --- [           main] c.i.mp.mapper.AddressMapper.selectList    : ==>
  Parameters:
17:19:39 DEBUG 23528 --- [           main] c.i.mp.mapper.AddressMapper.selectList    : <==
  Total: 8
Address(id=60, userId=1, province=北京, city=北京, town=朝阳区, mobile=13700221122, street=修正大厦,
  contact=Jack, isDefault=false, notes=null, deleted=false)
Address(id=61, userId=1, province=上海, city=上海, town=浦东新区, mobile=13301212233, street=航头镇航
  头路, contact=Jack, isDefault=true, notes=null, deleted=false)
Address(id=63, userId=2, province=广东, city=佛山, town=永春, mobile=13301212233, street=永春武馆,
  contact=Rose, isDefault=false, notes=null, deleted=false)
```

黑马程序员·研究院

综上，开启了逻辑删除功能以后，我们就可以像普通删除一样做CRUD，基本不用考虑代码逻辑问题。还是非常方便的。

！ 注意：

逻辑删除本身也有自己的问题，比如：

- 会导致数据库表垃圾数据越来越多，从而影响查询效率
- SQL中全都需要对逻辑删除字段做判断，影响查询效率

因此，我不太推荐采用逻辑删除功能，如果数据不能删除，可以采用把数据迁移到其它表的办法。

3.3.通用枚举

User类中有一个用户状态字段：

```
34 // 详细信息
35
36
37
38 private String info;
39
40
41
42
43 // 使用状态 (1正常 2冻结)
44 private Integer status;
45
46
47
48 // 账户余额
49 private Integer balance;
50
51
52
53
54
```

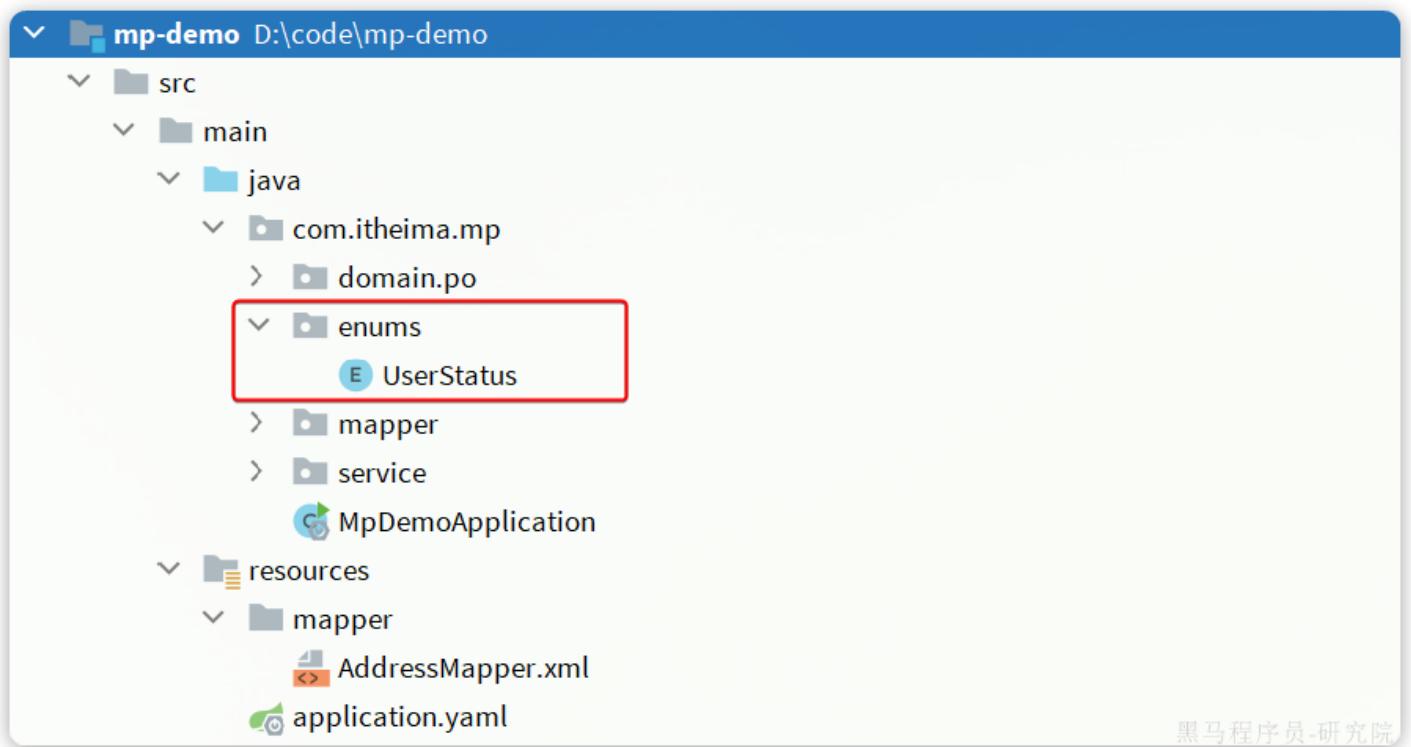
黑马程序员-研究院

像这种字段我们一般会定义一个枚举，做业务判断的时候就可以直接基于枚举做比较。但是我们数据库采用的是 int 类型，对应的PO也是 Integer。因此业务操作时必须手动把 枚举 与 Integer 转换，非常麻烦。

因此，MybatisPlus提供了一个处理枚举的类型转换器，可以帮我们**把枚举类型与数据库类型自动转换**。

3.3.1. 定义枚举

我们定义一个用户状态的枚举：



黑马程序员-研究院

代码如下：

```
1 package com.itheima.mp.enums;
2
3 import com.baomidou.mybatisplus.annotation.EnumValue;
4 import lombok.Getter;
5
6 @Getter
7 public enum UserStatus {
8     NORMAL(1, "正常"),
9     FREEZE(2, "冻结")
10    ;
11    private final int value;
12    private final String desc;
13
14    UserStatus(int value, String desc) {
15        this.value = value;
16        this.desc = desc;
17    }
18 }
```

然后把 User 类中的 status 字段改为 UserStatus 类型：

User.java

```
39     | 详细信息  
40     |  
41     | private String info;  
42     |  
43     | 使用状态 (1正常 2冻结)  
44     | private UserStatus status;  
45     |  
46     | 账户余额  
47     |  
48     | private Integer balance;
```

黑马程序员-研究院

要让 MybatisPlus 处理枚举与数据库类型自动转换，我们必须告诉 MybatisPlus，枚举中的哪个字段的值作为数据库值。

MybatisPlus 提供了 @EnumValue 注解来标记枚举属性：

```
@Getter  
public enum UserStatus {  
    NORMAL( value: 1, desc: "正常") ,  
    FREEZE( value: 2, desc: "冻结")  
;  
    @EnumValue  
    private final int value;  
    private final String desc;  
  
    UserStatus(int value, String desc) {  
        this.value = value;  
        this.desc = desc;  
    }  
}
```

黑马程序员-研究院

3.3.2.配置枚举处理器

在application.yaml文件中添加配置：

```
1 mybatis-plus:  
2     configuration:  
3         default-enum-type-handler:  
4             com.baomidou.mybatisplus.core.handlers.MybatisEnumTypeHandler
```

3.3.3.测试

```
1 @Test  
2 void testService() {  
3     List<User> list = userService.list();  
4     list.forEach(System.out::println);  
5 }
```

最终，查询出的 User 类的 status 字段会是枚举类型：

```
> └─ this = {UserServiceTest@7750}
└─ list = {ArrayList@7749} size = 4
  └─ 0 = {User@7762} "User(id=1, username=Jack, password=123, phone=13900112224"
    > └─ f id = {Long@7872} 1
    > └─ f username = "Jack"
    > └─ f password = "123"
    > └─ f phone = "13900112224"
    > └─ f info = "{\"age\": 20, \"intro\": \"佛系青年\", \"gender\": \"male\"}"
    > └─ f status = {UserStatus@7877} "NORMAL"
    > └─ f balance = {Integer@7878} 1600
```

黑马程序员-研究院

同时，为了使页面查询结果也是枚举格式，我们需要修改UserVO中的status属性：

```
15   @ApiModelProperty("用户id")
16   private Long id;
17
18   @ApiModelProperty("用户名")
19   private String username;
20
21   @ApiModelProperty("详细信息")
22   private UserInfo info;
23
24   @ApiModelProperty("使用状态（1正常 2冻结）")
25   private UserStatus status;
```

黑马程序员-研究院

并且，在UserStatus枚举中通过 `@JsonValue` 注解标记JSON序列化时展示的字段：

```
4   import com.fasterxml.jackson.annotation.JsonValue;
5   import lombok.Getter;
6
7   @Getter
8   public enum UserStatus {
9     NORMAL( value: 1, desc: "正常"),
10    FROZEN( value: 2, desc: "冻结"),
11    ;
12    @EnumValue
13    private final int value;
14    @JsonValue
15    private final String desc;
```

黑马程序员-研究院

最后，在页面查询，结果如下：

The screenshot shows a Swagger UI interface. On the left, there's a sidebar with various API endpoints. The main area has tabs for '文档' (Documentation) and '调试' (Debug). A 'GET /users/{id}' request is selected. The '请求参数' tab is active, showing a parameter 'id' with value '1'. The '响应内容' tab shows the JSON response:

```
1 {  
2   "id": 1,  
3   "username": "Jack",  
4   "info": {"age": 20, "intro": "\u4e1a\u4e1a\u4eba\u4eba", "gender": "male"},  
5   "status": 1,\r\n6   "balance": 1688,  
7   "addresses": [  
8     {"id": 68,  
9       "userId": 1,  
10      "province": "\u5316\u5316",  
11      "city": "\u5316\u5316",  
12      "town": "\u5316\u5316"}  
13 ]  
}
```

Annotations on the right side explain some fields:

- 用户id
- 用户名
- 详细信息
- 使用状态 (1正常 2冻结), 可用值: NORMAL, FROZEN
- 账户余额
- 用户的收获地址
- id
- 用户ID
- 省
- 市
- 县/区

3.4.JSON类型处理器

数据库的user表中有一个 `info` 字段，是JSON类型：

| # | 名称 | 数据类型 | 注释 | 长度/集合 |
|---|--------------------------|-----------------------|----------------|-------|
| 1 | <code>id</code> | <code>BIGINT</code> | 用户id | 19 |
| 2 | <code>username</code> | <code>VARCHAR</code> | 用户名 | 50 |
| 3 | <code>password</code> | <code>VARCHAR</code> | 密码 | 128 |
| 4 | <code>phone</code> | <code>VARCHAR</code> | 注册手机号 | 20 |
| 5 | <code>info</code> | <code>JSON</code> | 详细信息 | |
| 6 | <code>status</code> | <code>INT</code> | 使用状态 (1正常 2冻结) | 10 |
| 7 | <code>balance</code> | <code>INT</code> | 账户余额 | 10 |
| 8 | <code>create_time</code> | <code>DATETIME</code> | 创建时间 | |
| 9 | <code>update_time</code> | <code>DATETIME</code> | 更新时间 | |

格式像这样：

```
1 {"age": 20, "intro": "\u4e1a\u4e1a\u4eba\u4eba", "gender": "male"}
```

而目前 `User` 实体类中却是 `String` 类型：

User.java

```
34     private String phone;
```

```
35
```

详细信息

```
39
```

```
40     private String info;
```

```
41
```

使用状态（1正常 2冻结）

```
44     private UserStatus status;
```

```
45
```

黑马程序员-研究院

这样一来，我们要读取info中的属性时就非常不方便。如果要方便获取，info的类型最好是一个 Map 或者实体类。

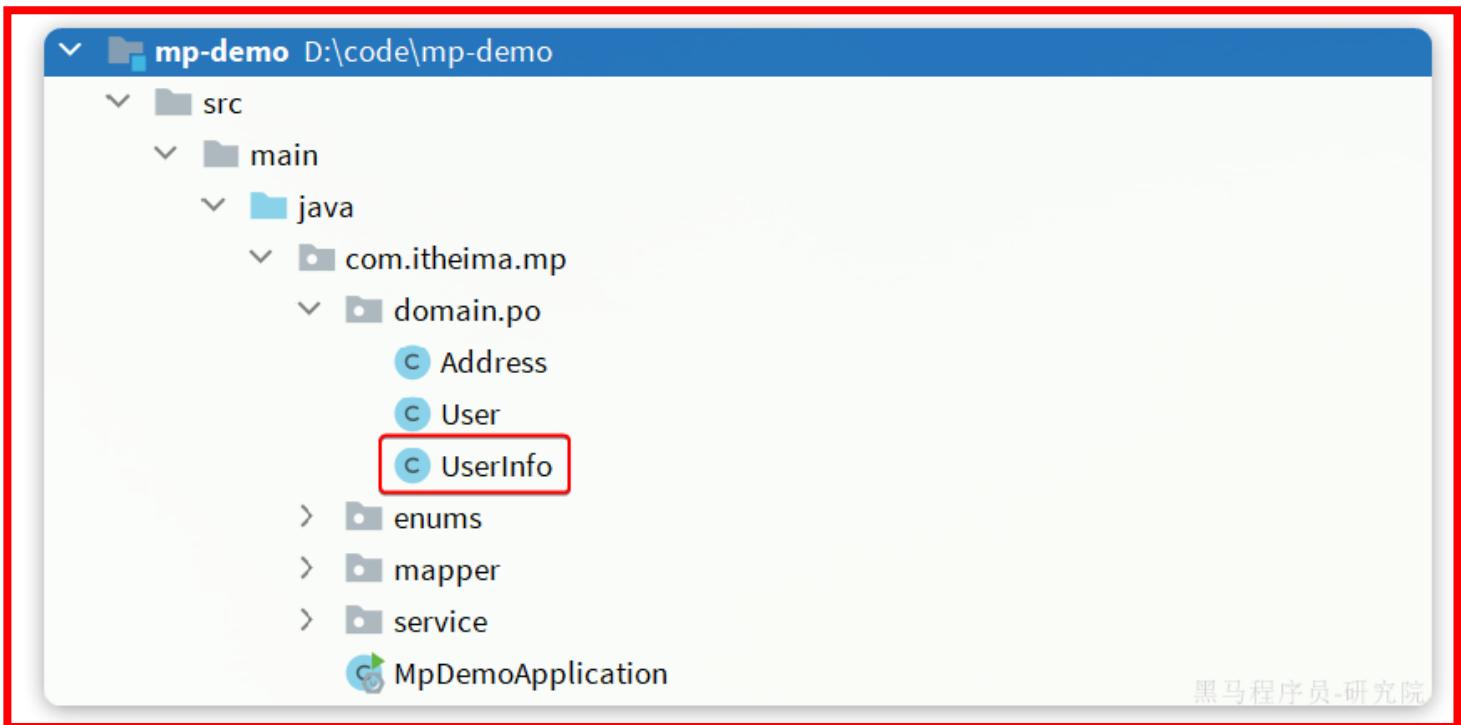
而一旦我们把 info 改为 对象 类型，就需要在写入数据库时手动转为 String，再读取数据库时，手动转换为 对象，这会非常麻烦。

因此MybatisPlus提供了很多特殊类型字段的类型处理器，解决特殊字段类型与数据库类型转换的问题。例如处理JSON就可以使用 JacksonTypeHandler 处理器。

接下来，我们就来看看这个处理器该如何使用。

3.4.1. 定义实体

首先，我们定义一个单独实体类来与info字段的属性匹配：



黑马程序员-研究院

代码如下：

```
1 package com.itheima.mp.domain.po;
2
3 import lombok.Data;
4
5 @Data
6 public class UserInfo {
7     private Integer age;
8     private String intro;
9     private String gender;
10 }
```

3.4.2. 使用类型处理器

接下来，将User类的info字段修改为UserInfo类型，并声明类型处理器：

User.java

```
32  
    | 注册手机号  
36    |     private String phone;  
37  
    |     详细信息  
41    | @TableField(typeHandler = JacksonTypeHandler.class)  
42    |     private UserInfo info;  
43
```

黑马程序员-研究院

测试可以发现，所有数据都正确封装到UserInfo当中了：

```
> └─ this ={UserServiceTest@8224}  
└─ list =[ArrayList@8223] size =4  
    └─ 0 ={User@8229} "User(id=1, username=Jack, password=123, phone=13900112224, info=UserInfo(age=20, intro=佛系青年, gender=male), status=NORMAL, balance=1600)"  
        > └─ id ={Long@8367} 1  
        > └─ username ="Jack"  
        > └─ password ="123"  
        > └─ phone ="13900112224"  
        > └─ info ={UserInfo@8371} "UserInfo(age=20, intro=佛系青年, gender=male)"  
        > └─ status ={UserStatus@8372} "NORMAL"  
        > └─ balance ={Integer@8373} 1600
```

黑马程序员-研究院

同时，为了让页面返回的结果也以对象格式返回，我们要修改UserVO中的info字段：

UserVO.java

```
15      @ApiModelProperty("用户id")  
16      private Long id;  
17  
18      @ApiModelProperty("用户名")  
19      private String username;  
20  
21      @ApiModelProperty("详细信息")  
22      private UserInfo info;
```

黑马程序员-研究院

此时，在页面查询结果如下：

```

1 {
2   "id": 1,
3   "username": "Jack",
4   "info": {
5     "age": 28,
6     "intro": "佛系青年",
7     "gender": "male"
8   },
9   "status": "正常",
10  "balance": 1600,
11  "addresses": [
12    {

```

显示说明 响应码: 200 耗时: 531

用户id
用户名
详细信息

使用状态 (1正常 2冻结), 可用值:NORMAL,FROZEN
账户余额
用户的收获地址

黑马程序员-研究院

3.5.配置加密 (选学)

目前我们配置文件中的很多参数都是明文，如果开发人员发生流动，很容易导致敏感信息的泄露。所以MybatisPlus支持配置文件的加密和解密功能。

我们以数据库的用户名和密码为例。

3.5.1.生成秘钥

首先，我们利用AES工具生成一个随机秘钥，然后对用户名、密码加密：

```

1 package com.itheima.mp;
2
3 import com.baomidou.mybatisplus.core.toolkit.AES;
4 import org.junit.jupiter.api.Test;
5
6 class MpDemoApplicationTests {
7     @Test
8     void contextLoads() {
9         // 生成 16 位随机 AES 密钥
10        String randomKey = AES.generateRandomKey();
11        System.out.println("randomKey = " + randomKey);
12
13        // 利用密钥对用户名加密
14        String username = AES.encrypt("root", randomKey);
15        System.out.println("username = " + username);
16
17        // 利用密钥对用户名加密

```

```
18     String password = AES.encrypt("MySQL123", randomKey);
19     System.out.println("password = " + password);
20
21 }
22 }
```

打印结果如下：

```
1 randomKey = 6234633a66fb399f
2 username = px2bAbnUfiY8K/IgsKvscg==
3 password = FGvCSEa0uga3ulDAsxw68Q==
```

3.5.2.修改配置

修改application.yaml文件，把jdbc的用户名、密码修改为刚刚加密生成的密文：

```
1 spring:
2   datasource:
3     url: jdbc:mysql://127.0.0.1:3306/mp?useUnicode=true&characterEncoding=UTF-
4       8&autoReconnect=true&serverTimezone=Asia/Shanghai&rewriteBatchedStatements=true
5     driver-class-name: com.mysql.cj.jdbc.Driver
6     username: mpw:QlWVnk10al3258x5rVhaeQ== # 密文要以 mpw:开头
7     password: mpw:EUFmeH3cNAzdRGd0QcabWg== # 密文要以 mpw:开头
```

3.5.3.测试

在启动项目的时候，需要把刚才生成的秘钥添加到启动参数中，像这样：

```
--mpw.key=6234633a66fb399f
```

单元测试的时候不能添加启动参数，所以要在测试类的注解上配置：

```
UserServiceTest.java
13
14 @SpringBootTest(args = "--mpw.key=6234633a66fb399f")
15 class UserServiceTest {
16
17     @Autowired
18     UserService userService;
19
20     @Test
21     void testService() {
22         List<User> list = userService.list();
23         list.forEach(System.out::println);
24     }
}
```

黑马程序员-研究院

然后随意运行一个单元测试，可以发现数据库查询正常。

4. 插件功能

MybatisPlus提供了很多的插件功能，进一步拓展其功能。目前已有的插件有：

- `PaginationInnerInterceptor`：自动分页
- `TenantLineInnerInterceptor`：多租户
- `DynamicTableNameInnerInterceptor`：动态表名
- `OptimisticLockerInnerInterceptor`：乐观锁
- `IllegalSQLInnerInterceptor`：sql 性能规范
- `BlockAttackInnerInterceptor`：防止全表更新与删除

！ 注意：

使用多个分页插件的时候需要注意插件定义顺序，建议使用顺序如下：

- 多租户,动态表名
- 分页,乐观锁
- sql 性能规范,防止全表更新与删除

这里我们以分页插件为里来学习插件的用法。

4.1. 分页插件

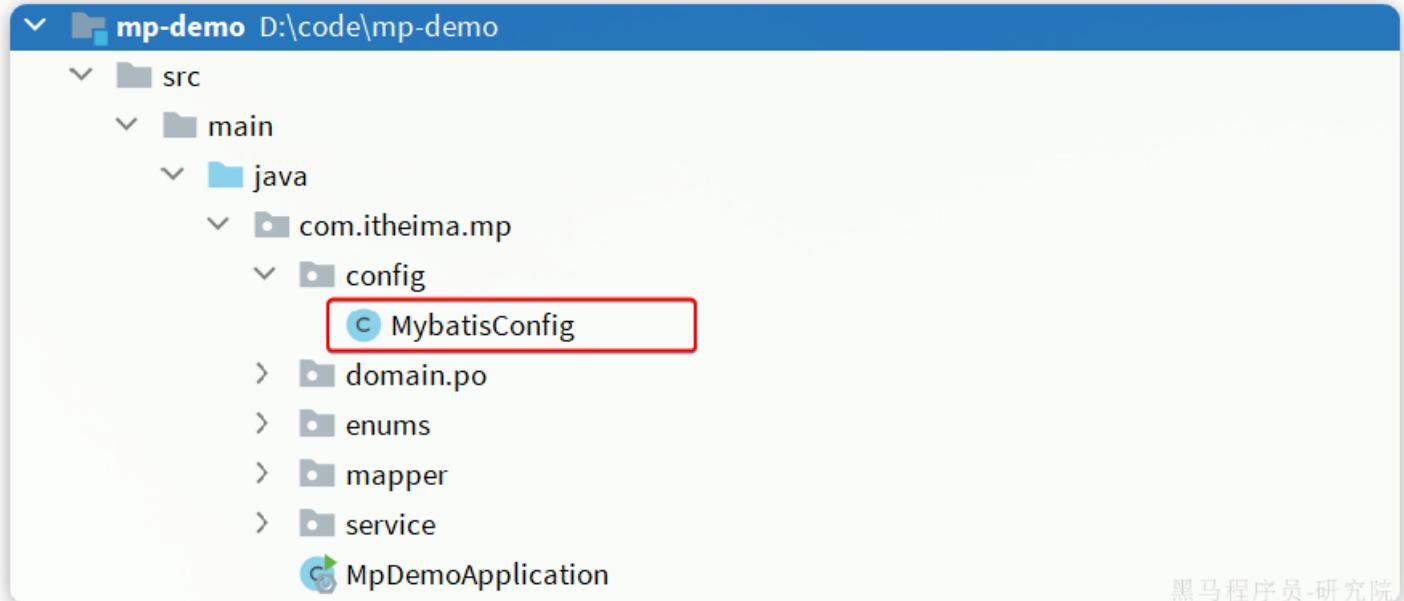
在未引入分页插件的情况下，`MybatisPlus` 是不支持分页功能的，`IService` 和 `BaseMapper`

中的分页方法都无法正常起效。

所以，我们必须配置分页插件。

4.1.1. 配置分页插件

在项目中新建一个配置类：



黑马程序员-研究院

其代码如下：

```
1 package com.itheima.mp.config;
2
3 import com.baomidou.mybatisplus.annotation.DbType;
4 import com.baomidou.mybatisplus.extension.plugins.MybatisPlusInterceptor;
5 import
    com.baomidou.mybatisplus.extension.plugins.inner.PaginationInnerInterceptor;
6 import org.springframework.context.annotation.Bean;
7 import org.springframework.context.annotation.Configuration;
8
9 @Configuration
10 public class MybatisConfig {      实际上是一个拦截器，拦截SQL
11                                         语句然后执行分页
12     @Bean
13     public MybatisPlusInterceptor mybatisPlusInterceptor() {
14         // 初始化核心插件
15         MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
16         // 添加分页插件
17         interceptor.addInnerInterceptor(new
            PaginationInnerInterceptor(DbType.MYSQL));
```

```
18         return interceptor;
19     }
20 }
```

4.1.2.分页API

编写一个分页查询的测试：

```
1 @Test
2 void testPageQuery() {
3     // 1.分页查询, new Page()的两个参数分别是：页码、每页大小
4     Page<User> p = userService.page(new Page<>(2, 2));
5     // 2.总条数
6     System.out.println("total = " + p.getTotal());
7     // 3.总页数
8     System.out.println("pages = " + p.getPages());
9     // 4.数据
10    List<User> records = p.getRecords();
11    records.forEach(System.out::println);
12 }
```

运行的SQL如下：

```
✓ Tests passed: 1 of 1 test - 1s 562 ms
22:09:29 DEBUG 6684 --- [           main] c.i.m.m.UserMapper.selectPage_mpCount : ==>
Preparing: SELECT COUNT(*) AS total FROM user   查询总条数
22:09:29 DEBUG 6684 --- [           main] c.i.m.m.UserMapper.selectPage_mpCount : ==>
Parameters:
22:09:29 DEBUG 6684 --- [           main] c.i.m.m.UserMapper.selectPage_mpCount : <==
Total: 1
22:09:29 DEBUG 6684 --- [           main] c.i.mp.mapper.UserMapper.selectPage : ==>
Preparing: SELECT id,username,password,phone,info,status,balance,create_time,update_time FROM user LIMIT ?,? 分页
22:09:29 DEBUG 6684 --- [           main] c.i.mp.mapper.UserMapper.selectPage : ==>
Parameters: 2(Long), 2(Long)
22:09:29 DEBUG 6684 --- [           main] c.i.mp.mapper.UserMapper.selectPage : <==
Total: 2
total = 4 结果
pages = 2
User(id=3, username=Hope, password=123, phone=13900112222, info=UserInfo(age=25, intro=上进青年, gender=male), status=NORMAL, balance=100000, createTime=2023-06-19T22:37:44, updateTime=2023-06-19T22:37:44)
User(id=4, username=Thomas, password=123, phone=17701265258, info=UserInfo(age=29, intro=伏地魔, gender=male), status=NORMAL, balance=800, createTime=2023-06-19T23:44:45, updateTime=2023-06-19T23:44:45)
```

这里用到了分页参数，Page，即可以支持分页参数，也可以支持排序参数。常见的API如下：

```
1 int pageNo = 1, pageSize = 5;
2 // 分页参数
3 Page<User> page = Page.of(pageNo, pageSize);
4 // 排序参数，通过OrderItem来指定
5 page.addOrder(new OrderItem("balance", false));
6
7 userService.page(page);
```

4.2.通用分页实体

现在要实现一个用户分页查询的接口，接口规范如下：

| 参数 | 说明 |
|------|--|
| 请求方式 | GET |
| 请求路径 | /users/page |
| 请求参数 | <pre>1 { 2 "pageNo": 1, 3 "pageSize": 5, 4 "sortBy": "balance", 5 "isAsc": false, 6 "name": "o", 7 "status": 1 8 }</pre> |
| 返回值 | <pre>1 { 2 "total": 100006, 3 "pages": 50003, 4 "list": [5 { 6 "id": 1685100878975279298, 7 "username": "user_9****", 8 "info": { 9 "age": 24, 10 "intro": "英文老师", 11 "gender": "female" 12 }, 13 } 14] 15 }</pre> |

```
13         "status": "正常",
14         "balance": 2000
15     }
16 ]
17 }
```

特殊说明

- 如果排序字段为空，默认按照更新时间排序
- 排序字段不为空，则按照排序字段排序

这里需要定义3个实体：

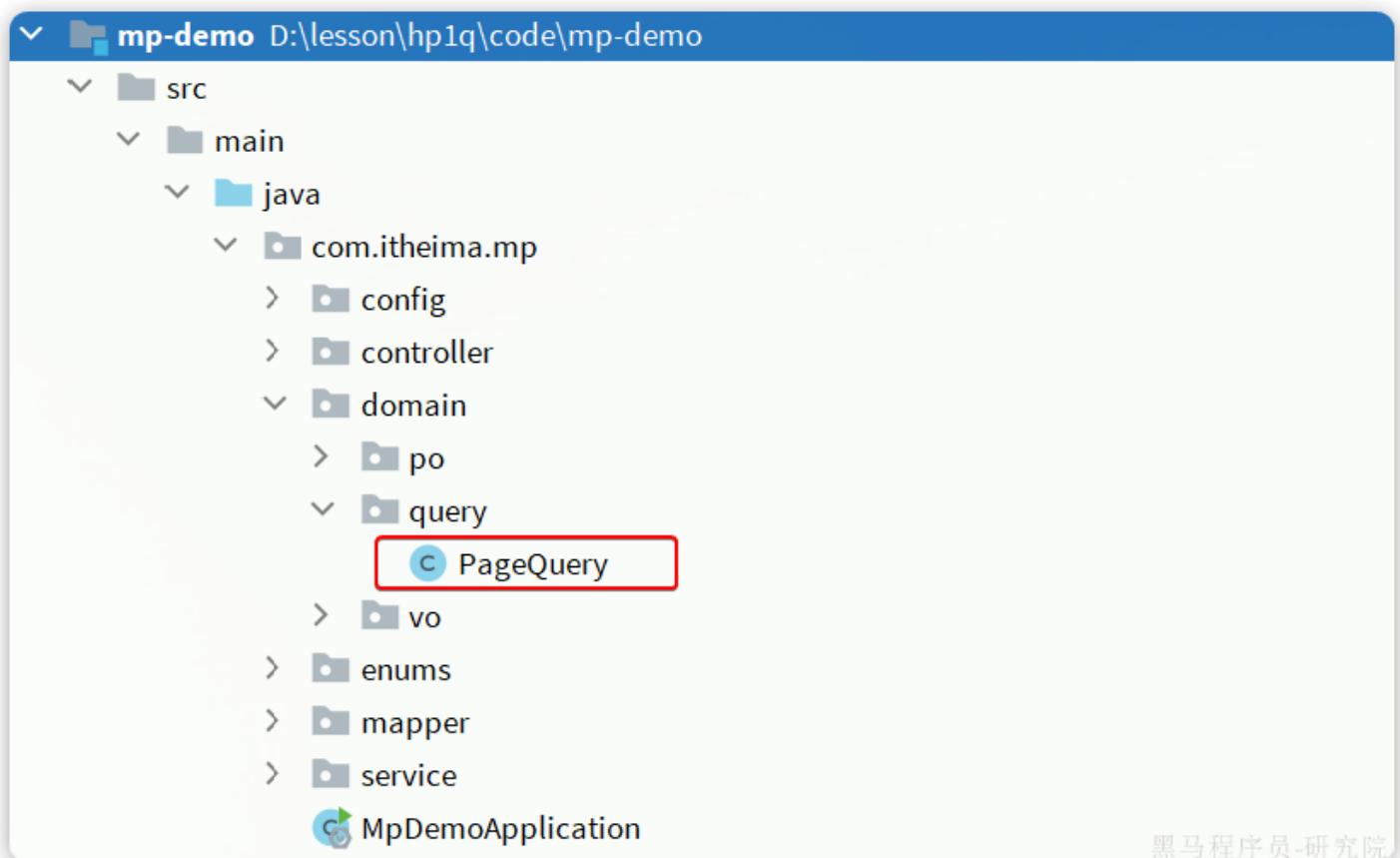
- `UserQuery`：分页查询条件的实体，包含分页、排序参数、过滤条件
- `PageDTO`：分页结果实体，包含总条数、总页数、当前页数据
- `UserVO`：用户页面视图实体

4.2.1. 实体

由于UserQuery之前已经定义过了，并且其中已经包含了过滤条件，具体代码如下：

```
1 package com.itheima.mp.domain.query;
2
3 import io.swagger.annotations.ApiModel;
4 import io.swagger.annotations.ApiModelProperty;
5 import lombok.Data;
6
7 @Data
8 @ApiModel(description = "用户查询条件实体")
9 public class UserQuery {
10     @ApiModelProperty("用户名关键字")
11     private String name;
12     @ApiModelProperty("用户状态: 1-正常, 2-冻结")
13     private Integer status;
14     @ApiModelProperty("余额最小值")
15     private Integer minBalance;
16     @ApiModelProperty("余额最大值")
17     private Integer maxBalance;
18 }
```

其中缺少的仅仅是分页条件，而分页条件不仅仅用户分页查询需要，以后其它业务也都有分页查询的需求。因此建议将分页查询条件单独定义为一个 `PageQuery` 实体：



黑马程序员-研究院

`PageQuery` 是前端提交的查询参数，一般包含四个属性：

- `pageNo`：页码
- `pageSize`：每页数据条数
- `sortBy`：排序字段
- `isAsc`：是否升序

PageQuery是前端的请求参数，由于很多查询可能都包含有这些页数等信息，所以提成一个公共组件PageQuery，让其他Query继承这个PageQuery以获得里面的属性

```
1 @Data
2 @ApiModelProperty(description = "分页查询实体")
3 public class PageQuery {
4     @ApiModelProperty("页码")
5     private Long pageNo;
6     @ApiModelProperty("页码")
7     private Long pageSize;
8     @ApiModelProperty("排序字段")
9     private String sortBy;
10    @ApiModelProperty("是否升序")
11    private Boolean isAsc;
12 }
```

然后，让我们的UserQuery继承这个实体：

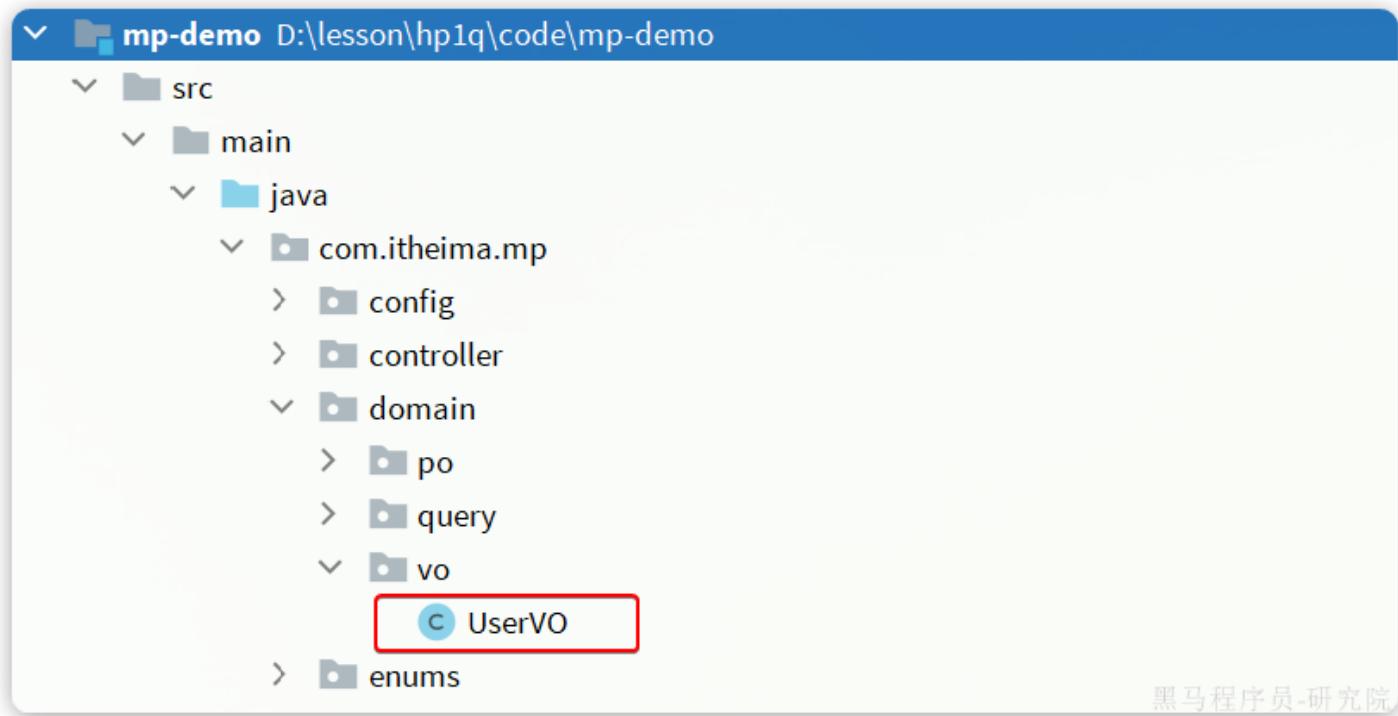
```
1 package com.itheima.mp.domain.query;
2
3 import io.swagger.annotations.ApiModel;
4 import io.swagger.annotations.ApiModelProperty;
5 import lombok.Data;
6 import lombok.EqualsAndHashCode;
7
8 @EqualsAndHashCode(callSuper = true)
9 @Data
10 @ApiModel(description = "用户查询条件实体")
11 public class UserQuery extends PageQuery {
12     @ApiModelProperty("用户名关键字")
13     private String name;
14     @ApiModelProperty("用户状态：1-正常，2-冻结")
15     private Integer status;
16     @ApiModelProperty("余额最小值")
17     private Integer minBalance;
18     @ApiModelProperty("余额最大值")
19     private Integer maxBalance;
20 }
```

返回值的用户实体沿用之前定一个 UserVO 实体：

返回结果是之前的UserVO实体，但是还是需要额外带上一些如：

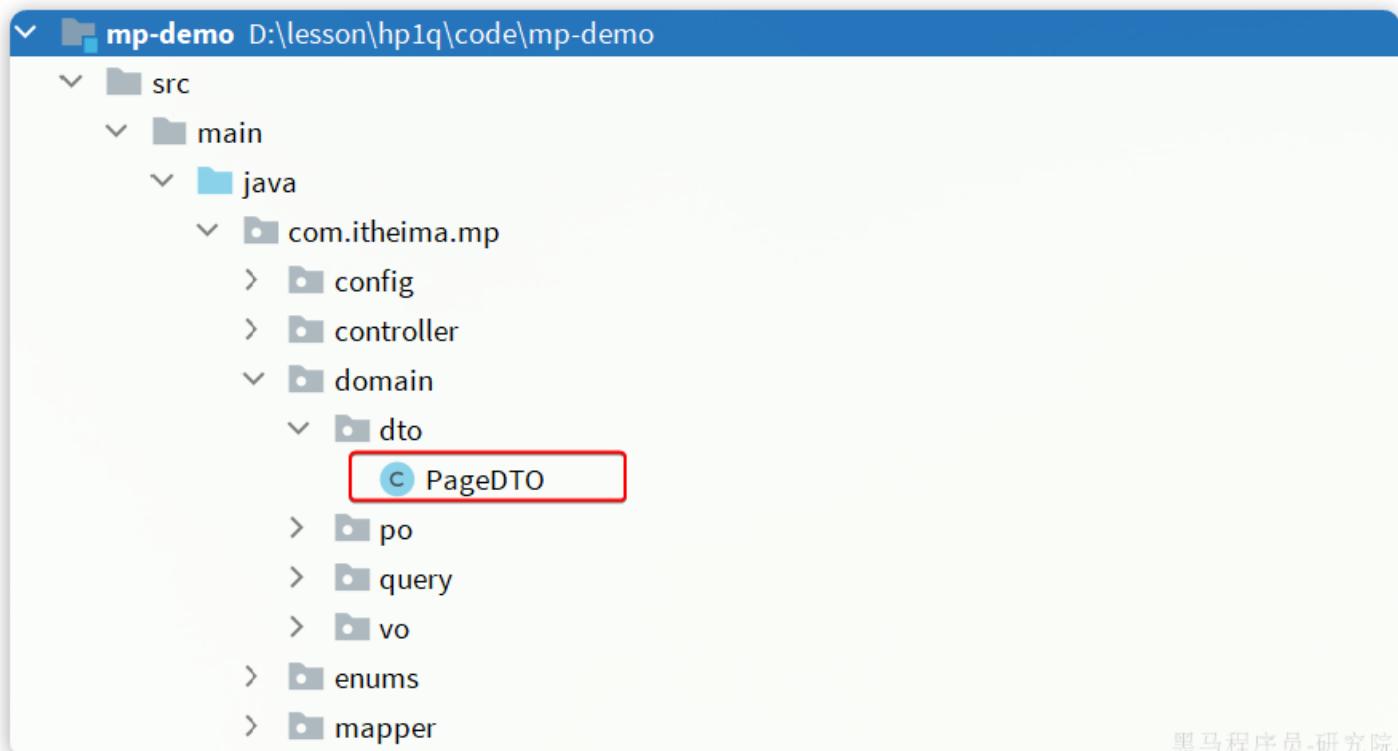
- 分页条数总大小
- 当前页码
- 当前页的数据

等一些常规Page拥有的信息，因为返回的结果应该是一个Page<UserVO>类，我们还要定义这个Page<?>类，以便后面使用



黑马程序员-研究院

最后，则是分页实体PageDTO：



黑马程序员-研究院

代码如下：

```
1 package com.itheima.mp.domain.dto;  
2  
3 import io.swagger.annotations.ApiModel;  
4 import io.swagger.annotations.ApiModelProperty;  
5 import lombok.Data;
```

```
6
7 import java.util.List;
8
9 @Data
10 @ApiModel(description = "分页结果")
11 public class PageDTO<T> {
12     @ApiModelProperty("总条数")
13     private Long total;
14     @ApiModelProperty("总页数")
15     private Long pages;
16     @ApiModelProperty("集合")
17     private List<T> list;
18 }
```

4.2.2.开发接口

我们在 UserController 中定义分页查询用户的接口：】

```
1 package com.itheima.mp.controller;
2
3 import com.itheima.mp.domain.dto.PageDTO;
4 import com.itheima.mp.domain.query.PageQuery;
5 import com.itheima.mp.domain.vo.UserVO;
6 import com.itheima.mp.service.UserService;
7 import lombok.RequiredArgsConstructor;
8 import org.springframework.web.bind.annotation.GetMapping;
9 import org.springframework.web.bind.annotation.RequestMapping;
10 import org.springframework.web.bind.annotation.RestController;
11
12 @RestController
13 @RequestMapping("users")
14 @RequiredArgsConstructor
15 public class UserController {
16
17     private final UserService userService;
18
19     @GetMapping("/page")
20     public PageDTO<UserVO> queryUsersPage(UserQuery query){
21         return userService.queryUsersPage(query);
22     }
23
24     // ...
25 }
```

然后在 `IUserService` 中创建 `queryUsersPage` 方法：

```
1 PageDTO<UserVO> queryUsersPage(PageQuery query);
```

接下来，在 `UserServiceImpl` 中实现该方法：

```
1 @Override
2 public PageDTO<UserVO> queryUsersPage(PageQuery query) {
3     // 1.构建条件
4     // 1.1.分页条件
5     Page<User> page = Page.of(query.getPageNo(), query.getPageSize());
6     // 1.2.排序条件 有判空的操作
7     if (query.getSortBy() != null) {
8         page.addOrder(new OrderItem(query.getSortBy(), query.getIsAsc()));
9     } else{
10        // 默认按照更新时间排序
11        page.addOrder(new OrderItem("update_time", false));
12    }
13    // 2.查询
14    page(page);
15    Page<User> userPage = LambdaQuery().like(...).eq(...).gt(...).lt(...).page(page)
16    // 3.数据非空校验
17    List<User> records = page.getRecords();
18    if (records == null || records.size() <= 0) {
19        // 无数据，返回空结果
20        return new PageDTO<>(page.getTotal(), page.getPages(),
21        Collections.emptyList());
22    }
23    // 4.有数据，转换
24    List<UserVO> list = BeanUtil.copyToList(records, UserVO.class);
25    // 5.封装返回
26    return new PageDTO<UserVO>(page.getTotal(), page.getPages(), list);
27 }
```

启动项目，在页面查看：

total : 100006
pages : 50003
list : 2

0

id : 1685100878975280115
username : Amy2
info
age : 18
intro : Java老师
gender : female
status : 正常
balance : 2000000

1

id : 1685100878975280114
username : Amy
info
age : 18
intro : Java老师
gender : female
status : 正常
balance : 2000000

黑马程序员-研究院

4.2.3. 改造PageQuery实体

在刚才的代码中，从 `PageQuery` 到 `MybatisPlus` 的 `Page` 之间转换的过程还是比较麻烦的。

我们完全可以在 `PageQuery` 这个实体中定义一个工具方法，简化开发。

像这样：

```
1 package com.itheima.mp.domain.query;  
2  
3 import com.baomidou.mybatisplus.core.metadata.OrderItem;  
4 import com.baomidou.mybatisplus.extension.plugins.pagination.Page;  
5 import lombok.Data;
```

```
6
7 @Data
8 public class PageQuery {
9     private Integer pageNo;
10    private Integer pageSize;
11    private String sortBy;
12    private Boolean isAsc;
13
14    public <T> Page<T> toMpPage(OrderItem ... orders){
15        // 1.分页条件
16        Page<T> p = Page.of(pageNo, pageSize);
17        // 2.排序条件
18        // 2.1.先看前端有没有传排序字段
19        if (sortBy != null) {
20            p.addOrder(new OrderItem(sortBy, isAsc));
21            return p;
22        }
23        // 2.2.再看有没有手动指定排序字段
24        if(orders != null){
25            p.addOrder(orders);
26        }
27        return p;
28    }
29
30    public <T> Page<T> toMpPage(String defaultSortBy, boolean isAsc){
31        return this.toMpPage(new OrderItem(defaultSortBy, isAsc));
32    }
33
34    public <T> Page<T> toMpPageDefaultSortByCreateTimeDesc() {
35        return toMpPage("create_time", false);
36    }
37
38    public <T> Page<T> toMpPageDefaultSortByUpdateTimeDesc() {
39        return toMpPage("update_time", false);
40    }
41 }
```

这样我们在开发时就可以省去对从 `PageQuery` 到 `Page` 的转换:

```
1 // 1.构建条件
2 Page<User> page = query.toMpPageDefaultSortByCreateTimeDesc();
```

4.2.4. 改造PageDTO实体

在查询出分页结果后，数据的非空校验，数据的vo转换都是模板代码，编写起来很麻烦。

我们完全可以将其封装到PageDTO的工具方法中，简化整个过程：

```
1 package com.itheima.mp.domain.dto;
2
3 import cn.hutool.core.bean.BeanUtil;
4 import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
5 import lombok.AllArgsConstructor;
6 import lombok.Data;
7 import lombok.NoArgsConstructor;
8
9 import java.util.Collections;
10 import java.util.List;
11 import java.util.function.Function;
12 import java.util.stream.Collectors;
13
14 @Data
15 @NoArgsConstructor
16 @AllArgsConstructor
17 public class PageDTO<V> {
18     private Long total;
19     private Long pages;
20     private List<V> list;
21
22     /**
23      * 返回空分页结果
24      * @param p MybatisPlus的分页结果
25      * @param <V> 目标VO类型
26      * @param <P> 原始PO类型
27      * @return VO的分页对象
28     */
29     public static <V, P> PageDTO<V> empty(Page<P> p){
30         return new PageDTO<>(p.getTotal(), p.getPages(),
31             Collections.emptyList());
32     }
33
34     /**
35      * 将MybatisPlus分页结果转为 VO分页结果
36      * @param p MybatisPlus的分页结果
37      * @param voClass 目标VO类型的字节码
38      * @param <V> 目标VO类型
39      * @param <P> 原始PO类型
40      * @return VO的分页对象
41     }
```

```

40     */
41     public static <V, P> PageDTO<V> of(Page<P> p, Class<V> voClass) {
42         // 1.非空校验
43         List<P> records = p.getRecords();
44         if (records == null || records.size() <= 0) {
45             // 无数据，返回空结果
46             return empty(p);
47         }
48         // 2.数据转换
49         List<V> vos = BeanUtil.copyToList(records, voClass);
50         // 3.封装返回
51         return new PageDTO<>(p.getTotal(), p.getPages(), vos);
52     }
53
54     /**
55      * 将MybatisPlus分页结果转为 VO分页结果，允许用户自定义PO到VO的转换方式
56      * @param p MybatisPlus的分页结果
57      * @param convertor PO到VO的转换函数
58      * @param <V> 目标VO类型
59      * @param <P> 原始PO类型
60      * @return VO的分页对象
61      */
62     public static <V, P> PageDTO<V> of(Page<P> p, Function<P, V> convertor) {
63         // 1.非空校验
64         List<P> records = p.getRecords();
65         if (records == null || records.size() <= 0) {
66             // 无数据，返回空结果
67             return empty(p);
68         }
69         // 2.数据转换
70         List<V> vos =
71             records.stream().map(convertor).collect(Collectors.toList());
72         // 3.封装返回
73         return new PageDTO<>(p.getTotal(), p.getPages(), vos);
74     }

```

最终，业务层的代码可以简化为：

```

1 @Override
2 public PageDTO<UserVO> queryUserByPage(PageQuery query) {
3     // 1.构建条件
4     Page<User> page = query.toMpPageDefaultSortByCreateTimeDesc();
5     // 2.查询

```

```
6     page(page);
7     // 3.封装返回
8     return PageDTO.of(page, UserVO.class);
9 }
```

如果是希望自定义PO到VO的转换过程，可以这样做：

```
1 @Override
2 public PageDTO<UserVO> queryUserByPage(PageQuery query) {
3     // 1.构建条件
4     Page<User> page = query.toMpPageDefaultSortByCreateTimeDesc();
5     // 2.查询
6     page(page);
7     // 3.封装返回
8     return PageDTO.of(page, user -> {
9         // 拷贝属性到vo
10        UserVO vo = BeanUtil.copyProperties(user, UserVO.class);
11        // 用户名脱敏
12        String username = vo.getUsername();
13        vo.setUsername(username.substring(0, username.length() - 2) + "****");
14        return vo;
15    });
16 }
```

最终查询的结果如下：

localhost:8080/users/page?pageNo=1&pageSize=2&sortBy=id&isAsc=true

JSON

```
total : 100004
pages : 50002
list : 2
  0
    id : 1
    username : Ja**
    info
      age : 20
      intro : 佛系青年
      gender : male
      status : NORMAL
      balance : 1600
  1
    id : 2
    username : Ro**
    info
      age : 19
      intro : 青涩少女
      gender : female
      status : NORMAL
      balance : 300
```

黑马程序员-研究院

5.作业

尝试改造项目一中的 Service 层和 Mapper 层实现，用 MybatisPlus 代替单表的CRUD

