Mybatis-plus笔记

0、准备工作

> 准备数据

```
CREATE TABLE `user` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT
'id',
  `user_name` varchar(20) NOT NULL COMMENT '用户名',
  `password` varchar(20) NOT NULL COMMENT '密码',
  `name` varchar(30) DEFAULT NULL COMMENT '姓名',
  `age` int(11) DEFAULT NULL COMMENT '年龄',
  `address` varchar(100) DEFAULT NULL COMMENT '地址',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT
CHARSET=utf8;

insert  into
`user`(`id`,`user_name`,`password`,`name`,`age`,`add
ress`) values (1,'ruiwen','123','瑞文',12,'山东'),
(2,'gailun','1332','盖伦',13,'平顶山'),
(3,'timu','123','提姆',22,'蘑菇石'),
(4,'daji','1222','妲己',221,'狐山');
```

> 创建SpringBoot工程 并添加依赖 **pom.xml**

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-
parent</artifactId>
    <version>2.5.0</version>
</parent>
```

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-
web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-
test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

创建启动类 **com.hq.HQApplication.java**

```
@SpringBootApplication
public class HQApplication {

    public static void main(String[] args) {
        SpringApplication.run(HQApplication.class);
    }
}
```

准备实体类 **com.hq.pojo.User.java**

```java
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {
    private Long id;
    private String userName;
    private String password;
    private String name;
    private Integer age;
    private String address;
}
```

1、SpringBoot整合MybatisPlus

1.1、添加依赖

```xml
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-
starter</artifactId>
    <version>3.4.3</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

1.2、配置数据库信息

> resource下的application.yml

```yaml
spring:
  datasource:
    url:
jdbc:mysql://localhost:3306/mybatis_plus_db?
characterEncoding=utf-8&serverTimezone=UTC
    username: root
    password: 302305
    driver-class-name: com.mysql.cj.jdbc.Driver
```

1.3、创建Mapper接口

com.hq.mapper.UserMapper.java

创建Mapper接口继承BaseMapper接口

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

BaseMapper接口中已经提供了很多常用方法。所以我们只需要直接从容器中获取Mapper就可以进行操作了，不需要自己去编写Sql语句。

1.4、配置Mapper扫描

在启动类上配置我们的Mapper在哪个包。

```java
@SpringBootApplication
@MapperScan("com.hq.mapper")
public class HQApplication {}
```

1.5、获取Mapper进行测试

test包下建立com.hq.MPTest

```
@SpringBootTest
public class MPTest {

    @Autowired
    private UserMapper userMapper;

    @Test
    public void testQueryList(){

 System.out.println(userMapper.selectList(null));
    }

}
```

2.常用设置

2.1 设置表映射规则

默认情况下MP操作的表名就是实体类的类名，但是如果表名和类名不一致就需要我们自己设置映射规则。

2.1.1 单独设置表映射规则

可以在实体类的类名上加上@TableName注解进行标识。

例如：如果表名是tb_user，而实体类名是User则可以使用以下写法。

```
@TableName("tb_user")
public class User {
    //....
}
```

2.1.2 全局设置表映射规则

一般一个项目表名的前缀都是统一风格的，这个时候如果一个个设置就太麻烦了。我们可以通过配置来**设置全局的表名前缀**。

例如：**如果一个项目中所有的表名相比于类名都是多了个前缀：`tb_` 。这可以使用如下方式配置**

```
mybatis-plus:
  global-config:
    db-config:
      #表名前缀
      table-prefix: tb_
```

2.2 设置主键生成策略

2.2.0 测试代码

```
@Test
public void testInsert(){
    User user = new User();
    user.setUserName("三更草堂222");
    user.setPassword("7777");
    int r = userMapper.insert(user);
    System.out.println(r);
}
```

2.2.1 单独设置主键生成策略

默认情况下使用MP插入数据时，如果**在我们没有设置主键生成策略的情况下默认的策略是基于雪花算法的自增id(非常长的数字)。**

如果我们需要使用别的策略可以在定义实体类时，**在代表主键的字段上加上`@TableId`注解，使用其`type`属性指定主键生成策略。**

例如：我们要设置主键自动增长则可以设置如下

```java
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {
    @TableId(type = IdType.AUTO)
    private Long id;
    //.....
}
```

全部主键策略定义在了枚举类`IdType`中， `IdType`有如下的取值

- **`AUTO`**

  数据库ID自增，**依赖于数据库。该类型请确保数据库设置了 ID自增否则无效**

- `NONE`

  未设置主键类型。若在代码中没有手动设置主键，则会根据**主键的全局策略(雪花算法)**自动生成（默认的主键全局策略是基于雪花算法的自增ID）

- `INPUT`

  需要手动设置主键，若不设置。插入操作生成SQL语句时，主键这一列的值会是`null`。

- `ASSIGN_ID`

  当没有手动设置主键，即实体类中的主键属性为空时，才会自动填充，使用雪花算法

- `ASSIGN_UUID`

  当实体类的主键属性为空时，才会自动填充，使用UUID

2.2.2 全局设置主键生成策略

```yaml
mybatis-plus:
  global-config:
    db-config:
      # id生成策略 auto为数据库自增
      id-type: auto
```

2.3 设置字段映射关系

默认情况下MP会根据实体类的属性名去映射表的列名。

如果数据库的列表和实体类的属性名不一致了。我们可以使用**`@TableField`注解的`value`属性去设置映射关系。**

例如：如果表中一个列名叫 address而 实体类中的属性名为addressStr则可以使用如下方式进行配置。

```
@TableField("address")
private String addressStr;
```

2.4 设置字段和列名的驼峰映射

默认情况下MP会开启字段名列名的驼峰映射， 即从经典数据库列名A_COLUMN（下划线命名） 到经典 Java 属性名 aColumn（驼峰命名）的类似映射 。**如果需要关闭我们可以使用如下配置进行关闭（false）。**

```
mybatis-plus:
  configuration:
    #是否开启自动驼峰命名规则（camel case）映射
    map-underscore-to-camel-case: false
```

2.5 日志

如果需要打印MP操作对应的SQL语句等，可以配置日志输出。

配置方式如下：

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

3.基本使用

3.1 插入数据

> 我们可以使用insert方法来实现数据的插入。

示例:

```
@Test
public void testInsert(){
    User user = new User();
    user.setUserName("三更草堂333");
    user.setPassword("7777888");
    int r = userMapper.insert(user);
    System.out.println(r);
}
```

3.2 删除操作

> 我们可以使用**deleteXXX**方法来实现数据的删除。

示例:

```
按照ids批次删除
@Test
public void testDelete(){
    List<Integer> ids = new ArrayList<>();
    ids.add(5);
    ids.add(6);
    ids.add(7);
    int i = userMapper.deleteBatchIds(ids);
    System.out.println(i);
}

根据id删除
```

```
@Test
public void testDeleteById(){
    int i = userMapper.deleteById(8);
    System.out.println(i);
}

按照匹配的条件进行删除
@Test
public void testDeleteByMap(){
    Map<String, Object> map = new HashMap<>();
    map.put("name","提姆");
    map.put("age",22);
    int i = userMapper.deleteByMap(map);
    System.out.println(i);
}
```

3.3 更新操作

> 我们可以使用updateXXX方法来实现数据的删除。**执行UPDATE user SET age=? WHERE id=?**

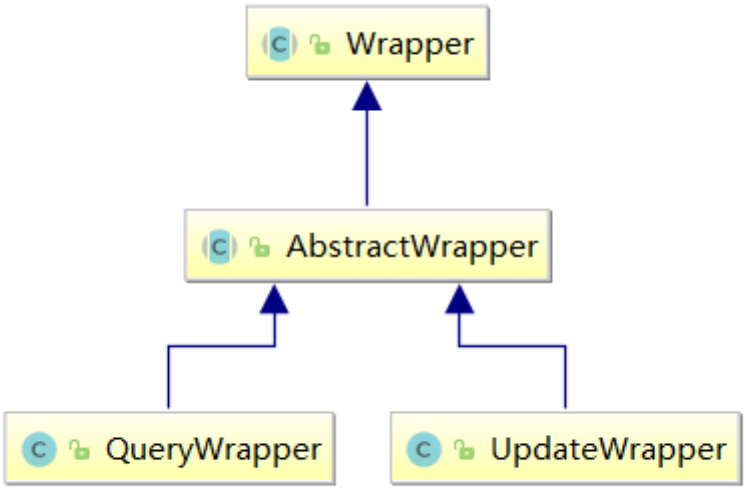
示例：

```
@Test
public void testUpdate(){
    //把id为2的用户的年龄改为14
    User user = new User();
    user.setId(2L);
    user.setAge(14);
    int i = userMapper.updateById(user);
    System.out.println(i);
}
```

4.条件构造器Wrapper

4.1 概述

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



在其子类AbstractWrapper中提供了很多用于构造Where条件的方法。

AbstractWrapper的子类QueryWrapper则额外提供了用于针对Select语法的select方法。可以用来设置查询哪些列。

AbstractWrapper的子类UpdateWrapper则额外提供了用于针对SET语法的set方法。可以用来设置对哪些列进行更新。

完整的AbstractWrapper方法可以参照：https://baomidou.com/guide/wrapper.html#abstractwrapper 介绍是用来干什么的。它的实现类有哪些QueryWrapper,UpdateWrapper,
【LambdaQueryWrapper】

4.2 常用AbstractWrapper方法

> eq: equals, 等于
>
> gt: greater than , 大于 >

> ge: greater than or equals，大于等于≥
>
> lt: less than，小于<
>
> le: less than or equals，小于等于≤
>
> between: 相当于SQL中的BETWEEN
>
> like: 模糊匹配。like("name","黄")，相当于SQL的name like '%黄%'
>
> likeRight: 模糊匹配右半边。likeRight("name","黄")，相当于SQL的name like '黄%'
>
> likeLeft: 模糊匹配左半边。likeLeft("name","黄")，相当于SQL的name like '%黄'
>
> notLike: notLike("name","黄")，相当于SQL的name not like '%黄%'
>
> isNull
>
> isNotNull
>
> and: SQL连接符AND
>
> or: SQL连接符OR
>
> in: in(“age",{1,2,3})相当于 age in(1,2,3)
>
> groupBy: groupBy("id","name")相当于 group by id,name
>
> orderByAsc :orderByAsc("id","name")相当于 order by id ASC,name ASC
>
> orderByDesc :orderByDesc ("id","name")相当于 order by id DESC,name DESC

4.2.1、wrapper.gt("age",18)

> SQL语句如下:

```
SELECT
    id,user_name,PASSWORD,NAME,age,address
FROM
    USER
WHERE
    age > 18 AND address = '狐山'
```

> 用Wrapper写法如下:

```
@Test
public void testWrapper01(){
    QueryWrapper wrapper = new QueryWrapper();
    wrapper.gt("age",18);
    wrapper.eq("address","狐山");
    List<User> users =
userMapper.selectList(wrapper);
    System.out.println(users);
}
```

4.2.2、wrapper.in("id",1,2,3)

> SQL语句如下:

```
SELECT
    id,user_name,PASSWORD,NAME,age,address
FROM
    USER
WHERE
    id IN(1,2,3) AND
    age BETWEEN 12 AND 29 AND
    address LIKE '%山%'
```

> 用Wrapper写法如下:

```
@Test
public void testWrapper02(){
    QueryWrapper<User> wrapper = new QueryWrapper<>
();
    wrapper.in("id",1,2,3);
    wrapper.between("age",12,29);
    wrapper.like("address","山");
    List<User> users =
userMapper.selectList(wrapper);
    System.out.println(users);
}
```

4.2.3、wrapper.orderByDesc("age");

> SQL语句如下:

```
SELECT
    id,user_name,PASSWORD,NAME,age,address
FROM
    USER
WHERE
    id IN(1,2,3) AND
    age > 10
ORDER BY
    age DESC
```

> 用Wrapper写法如下:

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

4.3 常用QueryWrapper方法(可以设置要查询的列)

QueryWrapper的 select 可以设置要查询的列。

4.3.1 queryWrapper.select("id","user_name");

> select(String... sqlSelect) 方法的测试为要查询的列名

SQL语句如下:

```
SELECT
    id,user_name
FROM
    USER
```

> MP写法如下:

```
@Test
public void testSelect01(){
    QueryWrapper<User> queryWrapper = new
QueryWrapper<>();
    queryWrapper.select("id","user_name");
    List<User> users =
userMapper.selectList(queryWrapper);
    System.out.println(users);
}
```

4.3.2 queryWrapper.select(User.class, new Predicate())

> select(Class entityClass, Predicate predicate)

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

> SQL语句如下：

```
SELECT
    id,user_name,age
FROM
    USER
```

> MP写法如下：**QueryWrapper<User> queryWrapper = new QueryWrapper<>();**

```
@Test
public void testSelect02(){
    QueryWrapper<User> queryWrapper = new
QueryWrapper<>();
    queryWrapper.select(User.class, new
Predicate<TableFieldInfo>() {
        @Override
        public boolean test(TableFieldInfo
tableFieldInfo) {
            return
"user_name".equals(tableFieldInfo.getColumn()) ||

 "age".equals(tableFieldInfo.getColumn());
        }
    });
    List<User> users =
userMapper.selectList(queryWrapper);
    System.out.println(users);
}
```

4.3.3 queryWrapper.select(new Predicate() {}

> select(Predicate predicate)

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

> SQL语句如下:

```
SELECT
    id,user_name,PASSWORD,NAME,age
FROM
    USER
```

> MP写法如下：`QueryWrapper<User> queryWrapper = new QueryWrapper<>(new User());`

```
@Test
public void testSelect03(){
    QueryWrapper<User> queryWrapper = new
QueryWrapper<>(new User());
    queryWrapper.select(new
Predicate<TableFieldInfo>() {
        @Override
        public boolean test(TableFieldInfo
tableFieldInfo) {
            return
!"address".equals(tableFieldInfo.getColumn());
        }
    });
    List<User> users =
userMapper.selectList(queryWrapper);
    System.out.println(users);
}
```

4.4 常用UpdateWrapper方法

我们前面在使用update方法时需要创建一个实体类对象传入，用来指定要更新的列及对应的值。但是如果需要更新的列比较少时，创建这么一个对象显的有点麻烦和复杂。**我们可以使用UpdateWrapper的set方法来设置要更新的列及其值**。同时这种方式也可以使用Wrapper去指定更复杂的更新条件。

`int updateById(@Param(Constants.ENTITY) T entity)`更新单个实体

`int update(@Param(Constants.ENTITY)Tentity, @Param(Constants.WRAPPER)Wrapper<T>updateWrapper)`多条件更新

SQL语句如下：

```
UPDATE
    USER
SET
    age = 99
where
    id > 4
```

我们想把id大于1的用户的年龄修改为99，则可以使用如下写法：

```
@Test
public void testUpdateWrapper(){
    UpdateWrapper<User> updateWrapper = new
UpdateWrapper<>();
    updateWrapper.gt("id",4);
    updateWrapper.set("age",99);
    userMapper.update(null,updateWrapper);
}
```

4.5 Lambda条件构造器

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

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

要执行的查询对应的SQL如下

```
SELECT
    id,user_name,PASSWORD,NAME,age,address
FROM
    USER
WHERE
    age > 18 AND address = '狐山'
```

如果使用之前的条件构造器写法如下

```
@Test
public void testLambdaWrapper(){
    QueryWrapper<User> queryWrapper = new
QueryWrapper();
    queryWrapper.gt("age",18);
    queryWrapper.eq("address","狐山");
    List<User> users =
userMapper.selectList(queryWrapper);
}
```

如果使用Lambda条件构造器写法如下

```
@Test
public void testLambdaWrapper2(){
    LambdaQueryWrapper<User> queryWrapper = new
LambdaQueryWrapper<>();
    queryWrapper.gt(User::getAge,18);
    queryWrapper.eq(User::getAddress,"狐山");
    List<User> users =
userMapper.selectList(queryWrapper);
    System.out.println(users);
}
```

5.自定义SQL

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

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

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

接下去我们分别来讲讲。

5.0 准备工作

> 准备数据

```
CREATE TABLE `orders` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `price` int(11) DEFAULT NULL COMMENT '价格',
  `remark` varchar(100) DEFAULT NULL COMMENT '备注',
  `user_id` int(11) DEFAULT NULL COMMENT '用户id',
  `update_time` timestamp NULL DEFAULT NULL COMMENT
'更新时间',
  `create_time` timestamp NULL DEFAULT NULL COMMENT
'创建时间',
  `version` int(11) DEFAULT '1' COMMENT '版本',
  `del_flag` int(1) DEFAULT '0' COMMENT '逻辑删除标
识,0-未删除,1-已删除',
  `create_by` varchar(100) DEFAULT NULL COMMENT '创建
人',
  `update_by` varchar(100) DEFAULT NULL COMMENT '更新
人',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT
CHARSET=utf8;

/*Data for the table `orders` */

insert  into
`orders`(`id`,`price`,`remark`,`user_id`,`update_tim
e`,`create_time`,`version`,`del_flag`,`create_by`,`u
pdate_by`) values (1,2000,'无',2,'2021-08-24
21:02:43','2021-08-24 21:02:46',1,0,NULL,NULL),
(2,3000,'无',3,'2021-08-24 21:03:32','2021-08-24
21:03:35',1,0,NULL,NULL),(3,4000,'无',2,'2021-08-24
21:03:39','2021-08-24 21:03:41',1,0,NULL,NULL);
```

创建实体类

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Orders  {

    private Long id;

    /**
     * 价格
     */
    private Integer price;

    /**
     * 备注
     */
    private String remark;

    /**
     * 用户id
     */
    private Integer userId;

    /**
     * 更新时间
     */
    private LocalDateTime updateTime;

    /**
     * 创建时间
     */
    private LocalDateTime createTime;

    /**
     * 版本
```

```
     */
    private Integer version;

    /**
     * 逻辑删除标识,0-未删除,1-已删除
     */
    private Integer delFlag;


}
```

5.1 Mybatis方式

> 在Mapper接口中定义方法

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

    User findMyUser(Long id);
}
```

> **先配置xml文件的存放目录**

```
mybatis-plus:
  mapper-locations: classpath*:/mapper/**/*.xml
```

> 创建对应的xml映射文件

创建对应的标签，编写对应的SQL语句

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper
3.0//EN" "http://mybatis.org/dtd/mybatis-3-
mapper.dtd" >
<mapper namespace="com.hq.mapper.UserMapper">

    <select id="findMyUser"
resultType="com.hq.pojo.User">
        select * from user where id = #{id}
    </select>
</mapper>
```

测试方法

```
@Test
public void testUserXml(){
    User user = userMapper.findMyUser(1L);
    System.out.println(user);
}
```

5.2 结合条件构造器ew.customSqlSegment

我们在使用上述方式自定义方法时。如果也希望我们的自定义方法能像MP自带方法一样使用条件构造器来进行条件构造的话只需要使用如下方式即可。

> 方法定义中添加Warpper类型的参数

添加Warpper类型的参数，并且要注意给其指定参数名。

```
public interface UserMapper extends BaseMapper<User>
{
    User
findMyUserByWrapper(@Param(Constants.WRAPPER)
QueryWrapper wrapper);
}
```

> 在SQL语句中获取Warpper拼接的SQL片段进行拼接。

```
<select id="findMyUserByWrapper"
resultType="com.hq.pojo.User">
    select * from user ${ew.customSqlSegment}
</select>
```

注意：不能使用#{}应该用${}，字符串拼接

测试方法

```
@Test
public void testWrapper(){
    QueryWrapper<User> queryWrapper = new
QueryWrapper<>();
    queryWrapper.eq("id",1);
    User user =
userMapper.findMyUserByWrapper(queryWrapper);
    System.out.println(user);
}
```

6.分页查询

6.1 基本分页查询IPage page = new Page<>()

> 配置分页查询拦截器

```
@Configuration
public class PageConfig {

    /**
     * 3.4.0之前的版本
     * @return
     */
   /* @Bean
```

```
    public PaginationInterceptor
paginationInterceptor(){
        return  new PaginationInterceptor();
    }*/

    /**
     * 3.4.0之后版本
     * @return
     */
   @Bean
   public MybatisPlusInterceptor
mybatisPlusInterceptor(){
       MybatisPlusInterceptor mybatisPlusInterceptor
= new MybatisPlusInterceptor();

mybatisPlusInterceptor.addInnerInterceptor(new
PaginationInnerInterceptor());
       return mybatisPlusInterceptor;
   }
}
```

进行分页查询 **userMapper.selectPage(page, null);**

```
@Test
public void testPage(){
    IPage<User> page = new Page<>();
    //设置每页条数
    page.setSize(2);
    //设置查询第几页
    page.setCurrent(1);
    userMapper.selectPage(page, null);
    System.out.println(page.getRecords());//获取当前页的数据
    System.out.println(page.getTotal());//获取总记录数6
    System.out.println(page.getCurrent());//当前页码1
}
```

6.2 多表分页查询

如果需要在多表查询时进行分页查询的话，就可以在mapper接口中自定义方法，然后让方法接收Page对象。

> 需求

我们需要去查询Orders表，并且要求查询的时候**除了要获取到Orders表中的字段，还要获取到每个订单的下单用户的用户名。**

> sql语句

```
SELECT
    o.*,u.`user_name`
FROM
    USER u,orders o
WHERE
    o.`user_id` = u.`id`
```

> 实体类修改

增加一个userName属性

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Orders  {
    //省略无关代码
    private String userName;
}
```

> 实现

①定义接口，定义方法

方法第一个测试定义成Page类型

```
public interface OrdersMapper extends
BaseMapper<Orders> {

    IPage<Orders> findAllOrders(Page<Orders> page);
}
```

在xml中不需要关心分页操作，MP会帮我们完成。

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper
3.0//EN" "http://mybatis.org/dtd/mybatis-3-
mapper.dtd" >
<mapper namespace="com.hq.mapper.OrdersMapper">
    <select id="findAllOrders"
resultType="com.hq.pojo.Orders">
        SELECT
            o.*,u.`user_name`
        FROM
            USER u,orders o
        WHERE
            o.`user_id` = u.`id`
    </select>
</mapper>
```

然后调用方法测试即可

```java
@Autowired
private OrdersMapper ordersMapper;
@Test
public void testOrdersPage(){
    Page<Orders> page = new Page<>();
    //设置每页大小
    page.setSize(1);
    //设置当前页码
    page.setCurrent(2);
    ordersMapper.findAllOrders(page);
    System.out.println(page.getRecords());
    System.out.println(page.getTotal());
}
```

7.Service 层接口

MP也为我们提供了Service层的实现。我们只需要编写一个接口，**继承IService，并创建一个接口实现类继承ServiceImpl**，即可使用。

相比于Mapper接口，Service层主要是支持了更多批量操作的方法。

7.1 基本使用

7.1.1 改造前

定义接口

```
public interface UserService {
    List<User> list();
}
```

定义实现类

```
@Service
public class UserServiceImpl implements UserService
{
    @Autowired
    private UserMapper userMapper;

    @Override
    public List<User> list() {
        return userMapper.selectList(null);
    }
}
```

7.1.2 改造后UserService

> 接口

```
public interface UserService extends IService<User>
{

}
```

> 实现类

```
@Service
public class UserServiceImpl extends
ServiceImpl<UserMapper,User> implements UserService
{

}
```

> 测试

```
@Autowired
private UserService userService;


@Test
public void testService(){
    List<User> list = userService.list();
    System.out.println(list);
}
```

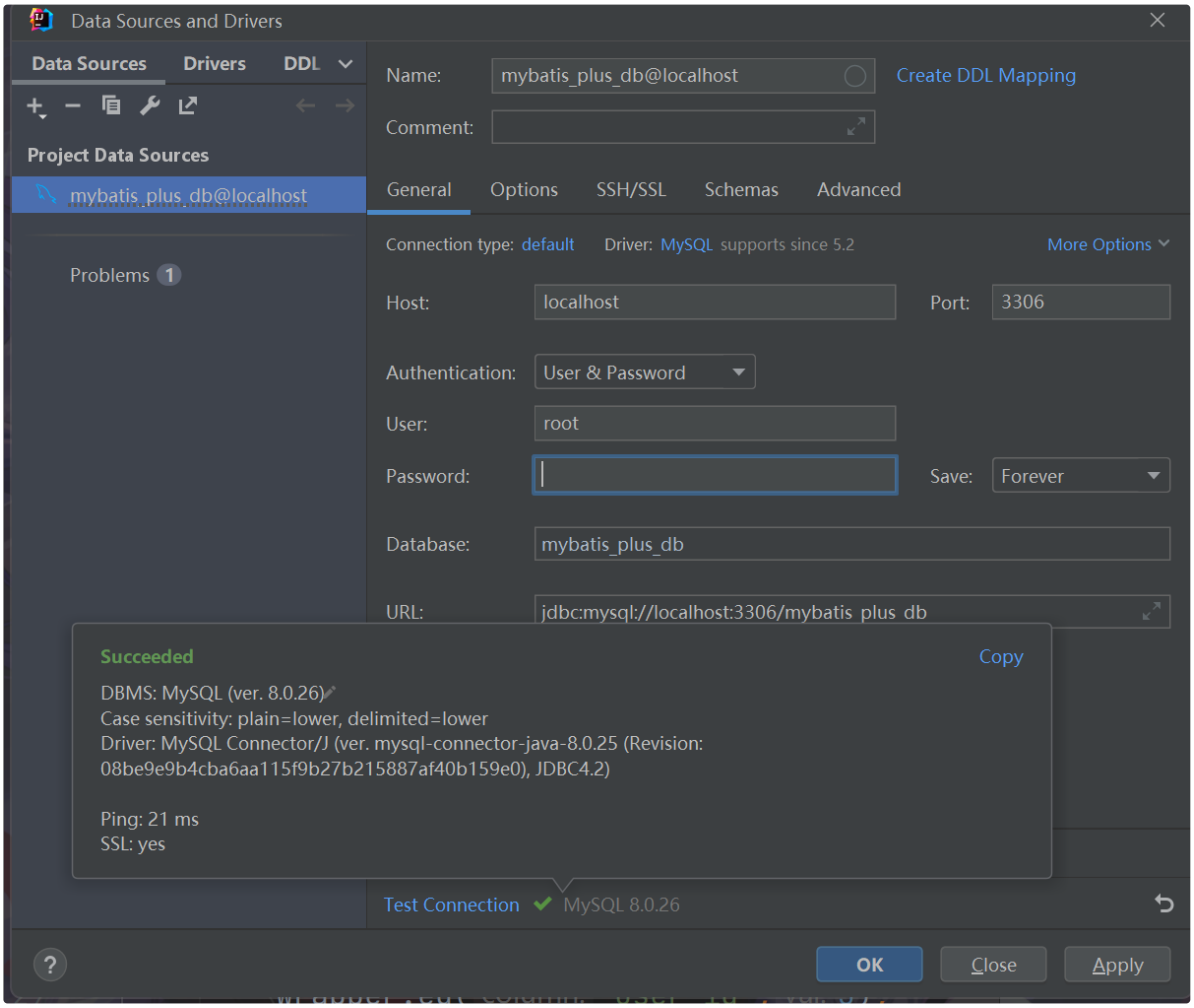
7.2自定义方法

```java
public interface UserService extends IService<User>
{

    User test();
}
```

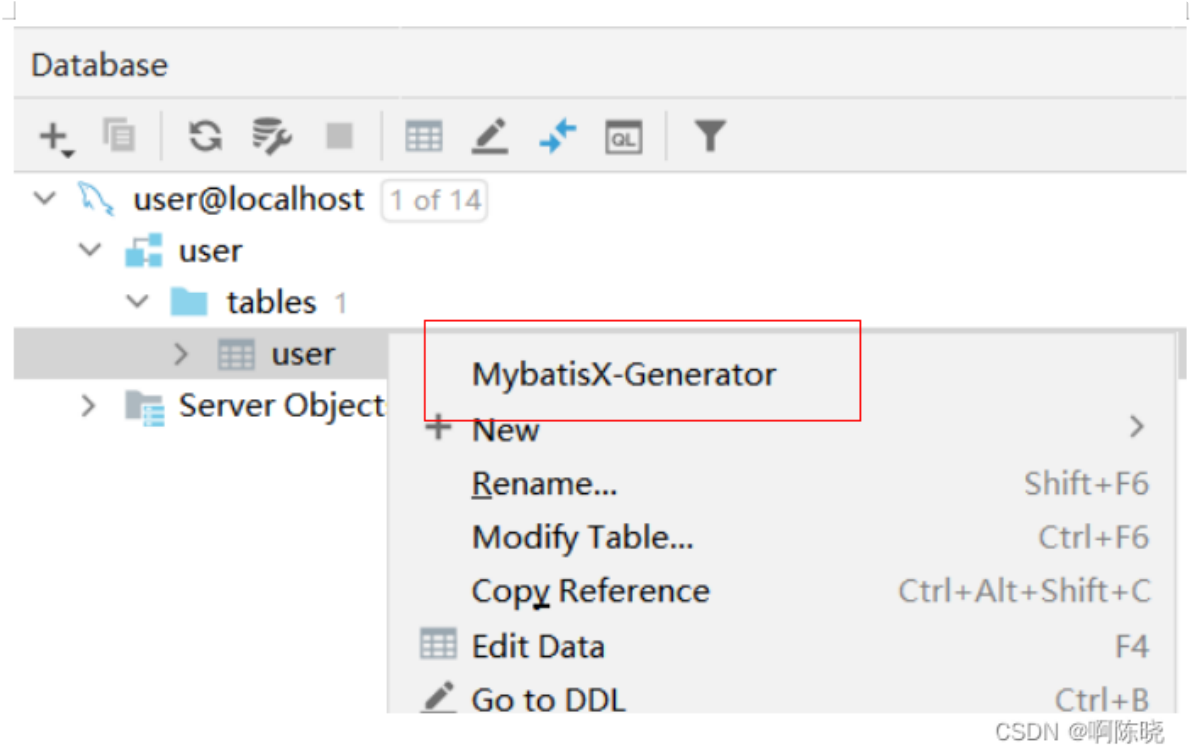
```java
@Service
public class UserServiceImpl extends
ServiceImpl<UserMapper,User> implements UserService
{

    @Autowired
    private OrdersMapper ordersMapper;

    @Override
    public User test() {
        UserMapper userMapper = getBaseMapper();
        List<Orders> orders =
ordersMapper.selectList(null);
        User user = userMapper.selectById(3);
        //查询用户对于的订单
        QueryWrapper<Orders> wrapper = new
QueryWrapper<>();
        wrapper.eq("user_id",3);
        List<Orders> ordersList =
ordersMapper.selectList(wrapper);
        return user;
    }
}
```

8、MybatisX插件使用

构建数据库连接

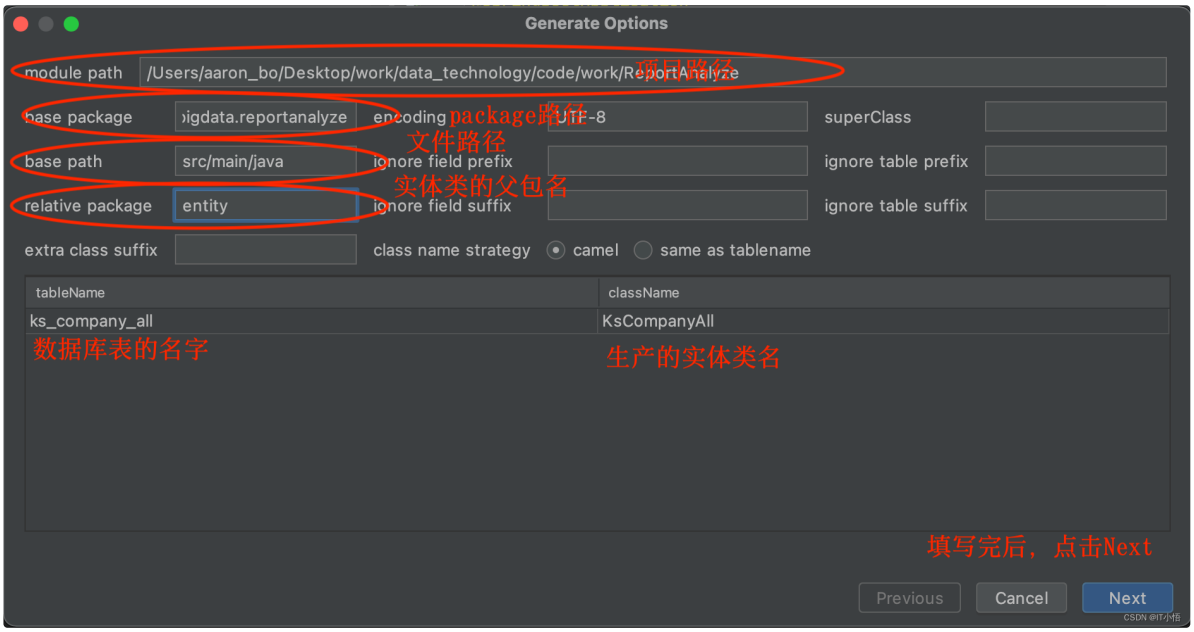


右键点击数据表，使用MybatisX-Generator

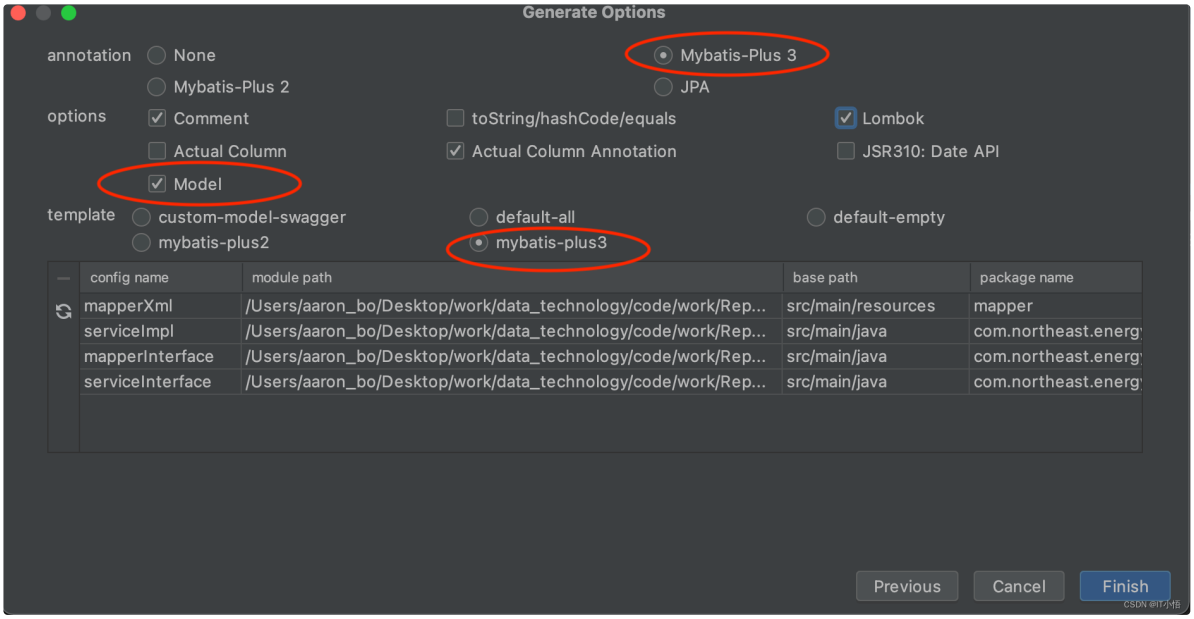


点击后我们会看到这样一个页面，我们可以在这个页面中设置需要消除的前后缀、文件存放目录等 . . .

package路径是会有一个新的包，作为生成文件的基包



点击Next，在下面是一些配置，我们勾选Mybatis-Plus的最新版本 Mybatix-Plus 3 和 简化开发的Lombok



9、字段自动填充

在实际项目中的表会和我们的orders表一样，有**更新时间**，**创建时间**，**创建人**，**更新人**等字段。

我们可以**使用 `@TableField` 的 `fill` 属性来设置字段的自动填充。**让我们能更方便的更新相关字段。

> 在对应字段上增加注解 `@TableField(fill = FieldFill.INSERT_UPDATE)`

使用TableField注解的fill属性来标注哪些字段需要在自动填充，加了注解MP才会在对应的SQL中为我们预留字段。而属性值代表我们在什么进行什么操作时需要预留字段。

```
/**
    * 更新时间
    */
@TableField(fill = FieldFill.INSERT_UPDATE)
private LocalDateTime updateTime;

/**
    * 创建时间
    */
@TableField(fill = FieldFill.INSERT)
private LocalDateTime createTime;
```

自定义填充处理器 **MetaObjectHandler**

```
@Component
public class MyMetaObjectHandler implements
MetaObjectHandler {
    @Override
    public void insertFill(MetaObject metaObject) {
        this.setFieldValByName("createTime",
LocalDateTime.now(), metaObject);
        this.setFieldValByName("updateTime",
LocalDateTime.now(), metaObject);
    }

    @Override
    public void updateFill(MetaObject metaObject) {
        this.setFieldValByName("updateTime",
LocalDateTime.now(), metaObject);
    }
}
```

测试插入数据

```
@Test
public void testFillField(){
    Orders orders = new Orders();
    orders.setPrice(100);
    ordersMapper.insert(orders);
}
```

10、逻辑删除

MP也支持逻辑删除的处理。**我们只需要配置好逻辑删除的实体字段名，代表删除的字段值和代表未删除的字段值后即可。**

注意：**如果3.3.0版本之前还需要在对应的字段上加上`@TableLogic`注解**

```
mybatis-plus:
  global-config:
    db-config:
      logic-delete-field: delFlag  # 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)
      logic-delete-value: 1 # 逻辑已删除值(默认为 1)
      logic-not-delete-value: 0 # 逻辑未删除值(默认为 0)
```

11、乐观锁

- 并发操作时，我们需要保证对数据的操作不发生冲突。乐观锁就是其中一种方式。乐观锁就是先加上不存在并发冲突问题，在进行实际数据操作的时候再检查是否冲突。
- 我们**在使用乐观锁时一般在表中增加一个version列。**用来记录我们对每天记录操作的版本。每次对某条记录进行过操作是，对应的版本也需要+1。
- 然后我们在每次要进行更新操作时，先查询对应数据的version值。**在执行更新时，set version = 老版本+1 where version = 老版本**

- **如果在查询老版本号到更新操作的中间时刻有其他人更新了这条数据，这样这次更新语句（更新肯定会加行级锁）就会更新失败。**
- 这里在更新时对version的操作如果有我们自己做就会显的有点麻烦。所以MP提供了乐观锁插件。
- 使用后我们就可以非常方便的实现对version的操作。

> 配置对应插件

```
@Configuration
public class MybatisPlusConfig {
    /**
     * 旧版
     */
    @Bean
    public OptimisticLockerInterceptor
optimisticLockerInterceptor() {
        return new OptimisticLockerInterceptor();
    }

    /**
     * 新版
     */
    @Bean
    public MybatisPlusInterceptor
mybatisPlusInterceptor() {
        MybatisPlusInterceptor
mybatisPlusInterceptor = new
MybatisPlusInterceptor();

 mybatisPlusInterceptor.addInnerInterceptor(new
OptimisticLockerInnerInterceptor());
        return mybatisPlusInterceptor;
    }
}
```

> 在实体类的字段上加上@Version注解

```
@Version
private Integer version;
```

> **注意：在更新前我们一定要先查询到version设置到实体类上再进行更新才能生效**

```
@Test
public void testVersion(){
    //查询id为3的数据
    QueryWrapper<Orders> queryWrapper = new
QueryWrapper<>();
    queryWrapper.eq("id",3);
    Orders orders =
ordersMapper.selectOne(queryWrapper);

    //对id为3的数据进行更新  把price修改为88
    orders.setPrice(88);
    ordersMapper.updateById(orders);
}
```

> 这种情况下我们可以看到执行的sql已经发生了变化。

```
==>  Preparing: UPDATE orders SET price=?,
update_time=?, version=? WHERE id=? AND version=?
AND del_flag=0
==> Parameters: 8888(Integer), null, 2(Integer),
2(Long), 1(Integer)
```