

# MybatisPlus-课堂笔记

刘银朋

重要性：拓展 < 了解 < 掌握/必会

web应用

## 一、MP简介

MybatisPlus，简称MP，是一个 Mybatis 的增强工具，**在 Mybatis 的基础上只做增强不做改变**。MP为简化开发、提高效率而生。它已经封装好了单表curd方法，我们直接调用这些方法就能实现单表CURD。

官网地址：<https://baomidou.com/>

### 愿景

我们的愿景是成为 MyBatis 最好的搭档，就像 魂斗罗 中的 1P、2P，基友搭配，效率翻倍。



实际开发中，MP和Mybatis通常是混用的：

- 单表CURD：直接使用MP提供好的方法即可
- 复杂SQL与多表联查：按Mybatis的方式自己编写Mapper接口和SQL语句

## 二、使用入门

使用步骤：

1. 准备工作：执行SQL脚本准备数据库；导入工程代码：添加依赖、配置数据库连接信息、创建引导类
2. 使用MP：创建实体类和Mapper接口。我们使用MybatisPlus插件一键生成
3. 功能测试：编写单元测试方法

### 1. 准备工作

#### 1.1 打开工程

将资料里的《day01-mp1-mapper》拷贝到工作区里(一个不含中文、空格、特殊字符的目录里)，然后使用idea打开它

## 1.2 添加依赖

pom.xml里已添加好了依赖，说明如下：

```
<!--SpringBoot父工程坐标-->
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.7.3</version>
</parent>

<dependencies>
  <!--MP的依赖坐标，这里用的是3.5.3.1版本（版本比较新，还未推广开，目前企业里应用的很少）-->
  <dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.5.3.1</version>
  </dependency>
  <!--MySQL8的驱动包-->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
  </dependency>

  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
  </dependency>
  <!--SpringBoot单元测试起步依赖-->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
  </dependency>
</dependencies>
```

## 1.3 修改配置

要操作数据库，则必须要配置数据库的连接信息。打开application.yaml，**修改成自己的数据库帐号和密码**

```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/mp
    username: root
    password: root
```

## 1.4 创建引导类

工程里已有，说明如下：

```
@SpringBootApplication
@MapperScan("com.itheima.mapper") //扫描com.itheima.mapper包下的Mapper接口
public class DemoMpApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoMpApplication.class, args);
    }
}
```

## 2. 使用MP

无论是使用Mybatis还是MybatisPlus操作数据库，都必须要准备实体类和Mapper接口。

MybatisPlus为了提高开发效率，提供了一个idea插件，可以连接数据库，直接给指定表生成实体类和Mapper接口

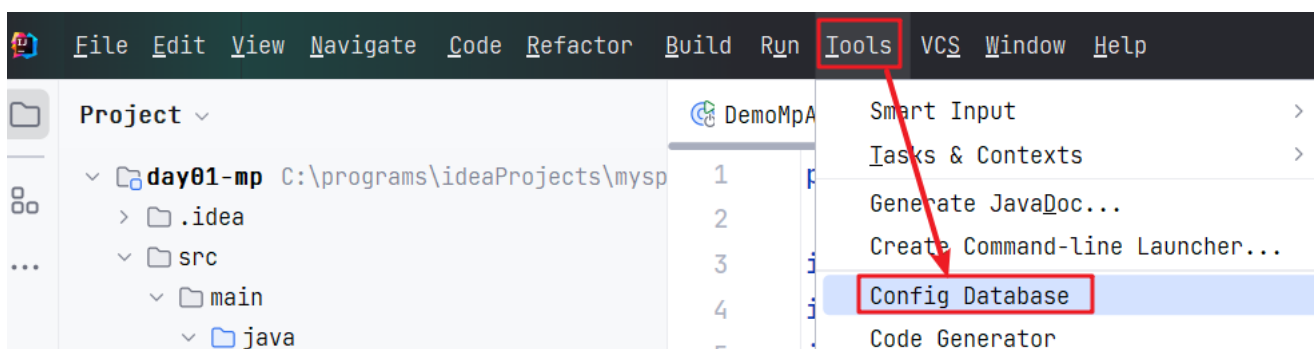
建议大家都在自己idea里安装上这个插件，插件名称也叫：MybatisPlus

### 2.1 安装MybatisPlus插件



### 2.2 生成实体类和Mapper接口

给插件配置数据库信息



数据库地址：jdbc:mysql://localhost:3306/mp?serverTimezone=UTC

dbUrl

jdbc:mysql://localhost:3306/mp?serverTimezone=UTC

jdbcDriver

mysql

user

root

password

....

test connect

ok

## 生成实体类和Mapper接口

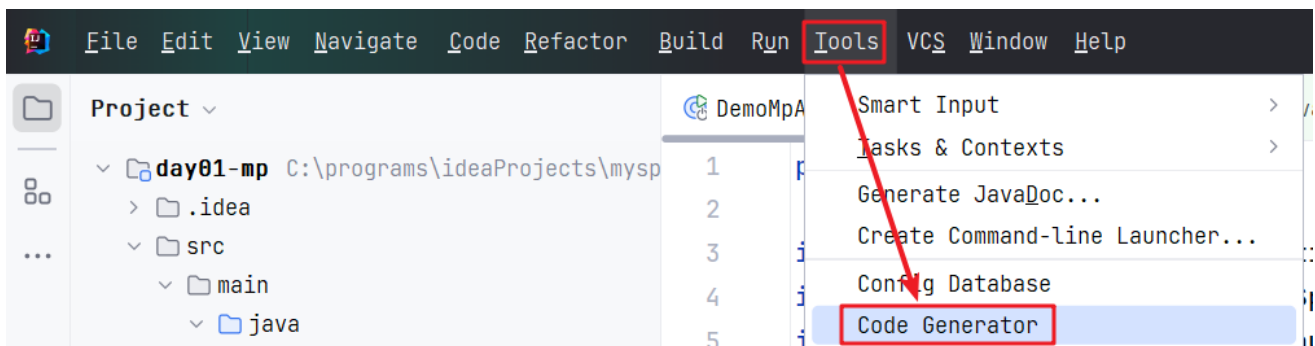


table name

create time

engine

coding

remark

address

2024-07-15 09:07:55

InnoDB

utf8mb3\_general\_ci

user

2024-07-15 09:07:55

InnoDB

utf8mb3\_general\_ci

用户表 ①选中表，要给哪张表生成代码，就选中哪张表

②将代码生成到哪个Module里。目前的工程没有Module，就空着

module

package

com.itheima②将代码生成到哪个包里，写包名

author

liuyp②代码作者

☒ over file

分配ID(主键类型为②实体类的主键生成策略 可在生成代码后再调整

☒ Entity

pojo ②实体类放到哪个包里

☒ Mapper

mapper ②Mapper接口放哪个包里

☐ Controller

controller

☐ Service

service

☐ ServiceImpl

service.impl

TablePrefix

☒ lombok

☐ restController

☐ swagger

☐ ResultMap

☐ is fill

☐ is enable cache

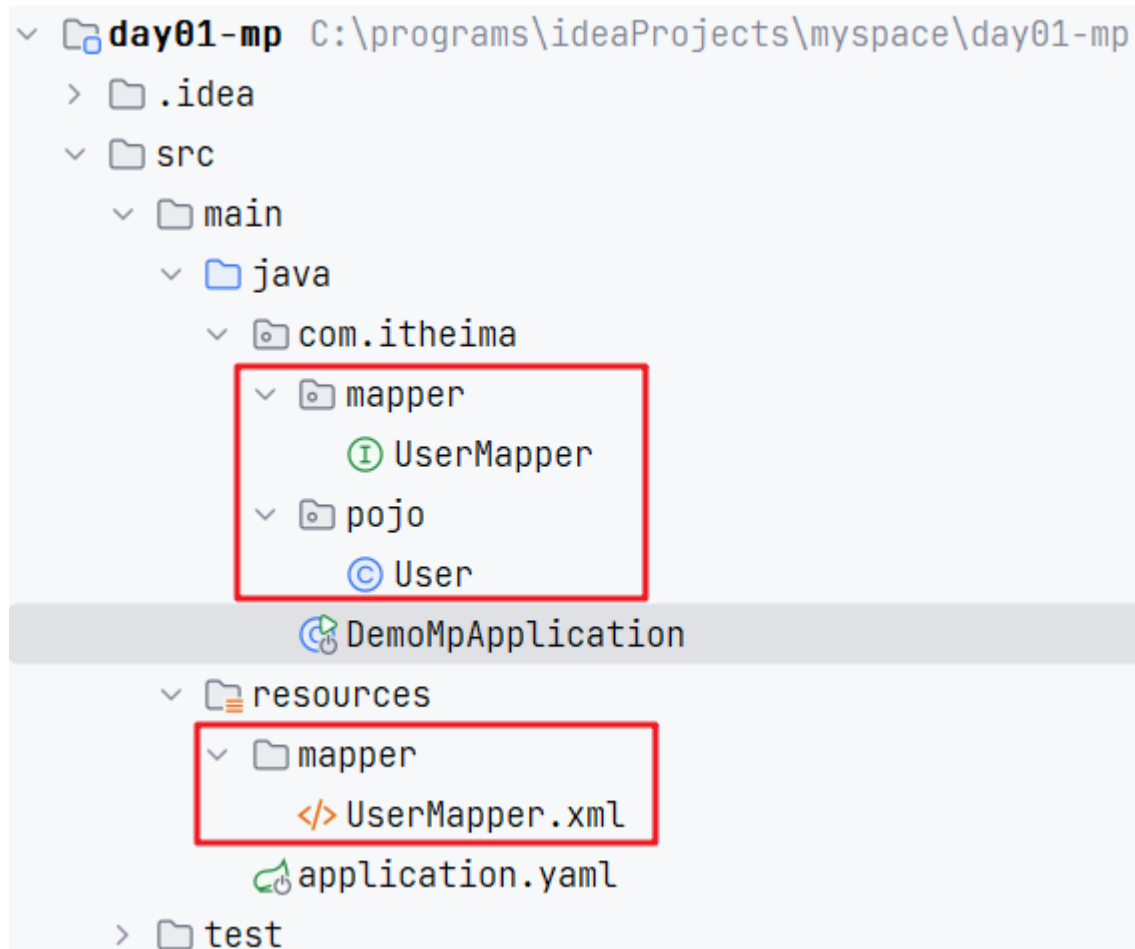
☐ is base column

save

check field

code generatro

查看生成的代码：



### 3. 功能测试

编写单元测试类，在测试类里编写测试方法：

```
@SpringBootTest
public class Demo01MpCurdTest {
    @Resource
    private UserMapper userMapper;

    @Test
    public void testMpStart(){
        User user = userMapper.selectById(1L);
        System.out.println("user = " + user);
    }
}
```

### 4. 小结

使用MP实现入门，步骤：

1. 准备工作：添加依赖，配置数据库地址，创建引导类
2. 创建实体类和Mapper接口：使用MybatisPlus插件直接生成会生成实体类

会生成Mapper接口，并且接口会自动继承 `BaseMapper<实体类>`

3. 使用mapper接口实现CURD了

## 三、常用方法与注解【必会】

### 1. 常用方法

#### 1.1 方法说明

刚刚使用插件生成的UserMapper，是一个空接口。但是它继承了 `BaseMapper<实体类>`，所以从父接口里继承了一批单表CURD方法，如下：



#### 1.2 使用示例

在单元测试类里添加方法：

```
@Test
public void testInsert(){
    User user = new User();
    user.setUsername("张三");
    user.setPassword("123456");
    user.setPhone("12345678901");
    user.setStatus(1);
    user.setInfo("高富帅");
    user.setBalance(1000);

    int i = userMapper.insert(user);
    System.out.println("影响行数i = " + i);

    //MP的插入方法，会自动获取主键值
    System.out.println("插入后user = " + user);
}

@Test
public void testDelete(){
    int i = userMapper.deleteById(1L);
    System.out.println("影响行数i = " + i);
}

@Test
public void testUpdate(){
    User user = new User();
    user.setId(2L);
    user.setInfo("白富美");
    user.setBalance(1000000);
    user.setUpdateTime(LocalDateTime.now());

    int i = userMapper.updateById(user);
    System.out.println("影响行数i = " + i);
}

@Test
public void testSelect(){
    //根据id查询一个
    User user = userMapper.selectById(2L);
    System.out.println("user = " + user);
    //查询数量
    long count = userMapper.selectCount(null);
    System.out.println("count = " + count);
    //查询列表
    List<User> users = userMapper.selectList(null);
    users.forEach(System.out::println);
}
```

## 2. 常用注解

MP已经能够帮我们执行CURD操作了，但是MP怎么知道要操作哪张表呢？这是因为实体类上有一些注解已经配置好了

注解	作用	用法
@TableName	用于设置实体类对应的表名	加在实体类上
@TableId	用于标识主键字段	加在实体类的属性上，表示此属性对应主键字段
@TableField	用于标识非主键字段	加在实体类的属性上，用于设置属性与字段的对应关系

## 2.1 表名映射@TableName

@TableName：用于告诉MP 实体类对应的表名是什么

使用方法：加在实体类上， @TableName("表名")

```
@TableName("user")
public class User implements Serializable {
    ...略...
}
```

💡 如果实体类上不加这个注解，可以吗？

如果实体类上没有这个注解，MP将会以 下划线-驼峰映射 的方式，根据实体类名查找对应的表名，例如：

- 实体类名是TbOrder，则MP会操作tb\_order表
- 实体类名是TbOrderDetail，则MP会操作tb\_order\_detail表
- 实体类名是User，则MP会操作User表

但是如果实体类与表名不符合 下划线-驼峰映射 规则，就必须在实体类上添加注解，显示设置表名了：

- 实体类名是Order，但是类上加了 @TableName("tb\_order")，则MP会操作tb\_order表
- 实体类名是OrderDetail，但是类上加了 @TableName("tb\_order\_detail") 则MP会操作tb\_order\_detail表

## 2.2 主键字段@TableId

@TableId：加在实体类里的属性上，表示这个属性对应的是主键字段。

使用方法： @TableId(value="主键字段名", type=IdType.主键生成策略)。

- value属性：设置主键字段名。如果主键字段名和属性名相同，可以省略 value="主键字段名" 不写
- type属性：设置主键生成策略。当我们调用了Mapper的insert方法时，MP会根据主键生成策略，自动设置主键值

MP提供了常用的主键生成策略供我们使用：

- IdType.NONE：不设置，跟随全局
- IdType.AUTO：数据库主键自增
- IdType.INPUT：MP不生成主键值，在插入数据时由我们提供一个主键值
- IdType.ASSIGN\_ID：由MP使用雪花算法生成一个id值，可兼容数值型(long类型)和字符串型
- IdType.ASSIGN\_UUID：由MP使用UUID算法生成一个id值



使用示例：

```
@TableName("user")
public class User implements Serializable {

    /**
     * 用户id，设置主键生成策略为雪花算法ASSIGN_ID
     */
    @TableId(value = "id", type = IdType.ASSIGN_ID)
    private Long id;

    ...略...
}
```

再调用一次插入数据的测试方法，去数据库里查看user表的数据，发现主键值是一个Long类型的

	id	username	password	phone
1	1	Jack	123	13900112224
2	2	Rose	123	13900112223
3	3	Hope	123	13900112222
4	4	Thomas	123	17701265258
5	1812690162031910914	张三	123456	12345678901

## 2.3 非主键字段@TableField

@TableField：加在实体类的属性上，用于设置属性对应的表字段（用于非主键字段）

使用方法：@TableField(value="字段名", exist=false)

- value属性：设置字段名。如果字段名与属性名相同 或符合 下划线-驼峰映射 规则，可以省略不写
- exist属性：如果此属性没有对应的字段，则必须设置为false，否则MP报错

使用示例：

```
@TableName("user")
public class User implements Serializable {
    @TableId(value = "id", type = IdType.ASSIGN_ID)
    private Long id;

    /**
     * 属性名是name；字段名是username
     * 两者不同，且不符合 下划线-驼峰映射规则
     * 所以：需要添加注解，设置此属性对应的字段名是username
     */
    @TableField(value="username")
    private String name;

    /**

```

```

* 属性名是email；数据库表里没有对应的字段
* 所以：需要添加注解，设置此属性没有对应的字段，避免MP报错“找不到字段email”
*/
@TableField(exist=false)
private String email;

private String password;
private String phone;
private String info;
private Integer status;
private Integer balance;
private LocalDateTime createTime;
private LocalDateTime updateTime;
}

```

## 3. 条件构造器

### 3.1 API说明

#### BaseMapper的条件操作方法

MP提供的所有**条件式的操作**，无论是条件查询，还是条件删除、条件修改，都需要提供Wrapper对象作为参数。而这个Wrapper就是条件构造器，用于构造where条件

BaseMapper

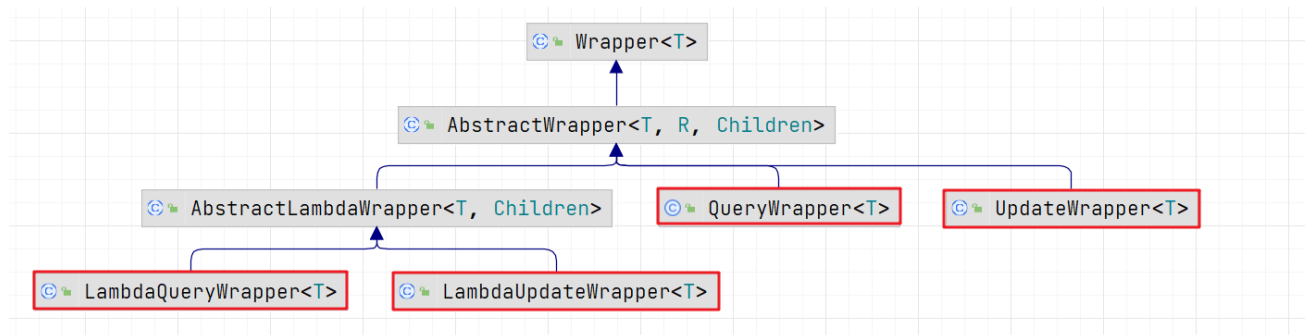
- `delete(Wrapper<T>): int` 根据条件删除
- `deleteBatchIds(Collection<?>): int`
- `deleteById(Serializable): int`
- `deleteById(T): int`
- `deleteByMap(Map<String, Object>): int`
- `exists(Wrapper<T>): boolean` 根据条件判断数据是否存在
- `insert(T): int`
- `selectBatchIds(Collection<? extends Serializable>): List<T>`
- `selectById(Serializable): T`
- `selectByMap(Map<String, Object>): List<T>`
- `selectCount(Wrapper<T>): Long` 查询符合条件的数据的数量
- `selectList(Wrapper<T>): List<T>` 查询符合条件的数据列表
- `selectMaps(Wrapper<T>): List<Map<String, Object>>`
- `selectMapsPage(P, Wrapper<T>): P`
- `selectObjs(Wrapper<T>): List<Object>`
- `selectOne(Wrapper<T>): T` 查询符合条件的数据（一条）
- `selectPage(P, Wrapper<T>): P` 分页查询符合条件的数据
- `update(T, Wrapper<T>): int` 修改符合条件的数据
- `updateById(T): int`

#### Wrapper的API介绍

Wrapper有4个常用的子类，我们根据写法进行划分：

- QueryWrapper和UpdateWrapper：分别用于构造查询条件、更新条件
- LambdaQueryWrapper和LambdaUpdateWrapper：分别用于构造查询条件、更新条件。

以上两类的作用是相同的，仅仅是写法不同。推荐使用Lambda形式的写法



## 3.2 条件查询

### QueryWrapper的API

#### 1. 创建QueryWrapper对象

方式一： `QueryWrapper wrapper = new QueryWrapper<实体类>()`

方式二： `QueryWrapper wrapper = Wrappers.<实体类>query();`

#### 2. 构造where条件，常用方法有：

查询方法	说明	备注
<code>eq(字段名,值)</code>	等于=	字段名=值 条件
<code>ne(字段名,值)</code>	不等与<>	
<code>gt(字段名,值)</code>	大于>	
<code>ge(字段名,值)</code>	大于等于>=	
<code>lt(字段名,值)</code>	小于<	
<code>le(字段名,值)</code>	小于等于<=	
<code>like(字段名,值)</code>	模糊查询 LIKE	MybatisPlus会自动加上%
<code>notLike(字段名,值)</code>	模糊查询 NOT LIKE	MybatisPlus会自动加上%
<code>in(字段名,值集合/数组)</code>	IN 查询	
<code>notIn(字段名,值集合/数组)</code>	NOT IN 查询	
<code>isNull(字段名)</code>	NULL 值查询	
<code>isNotNull(字段名)</code>	IS NOT NULL	

💡 如果需要动态拼接SQL条件，可使用以下重载的方法：

查询方法	说明	备注
eq(条件, 字段名, 值)	当条件为true时才生效, 等于=	字段名=值 条件
ne(条件, 字段名, 值)	当条件为true时才生效, 不等于<>	
gt(条件, 字段名, 值)	当条件为true时才生效, 大于>	
ge(条件, 字段名, 值)	当条件为true时才生效, 大于等于>=	
lt(条件, 字段名, 值)	当条件为true时才生效, 小于<	
le(条件, 字段名, 值)	当条件为true时才生效, 小于等于<=	
like(条件, 字段名, 值)	当条件为true时才生效, 模糊查询 LIKE	MP会自动加上%
notLike(条件, 字段名, 值)	当条件为true时才生效, 模糊查询 NOT LIKE	MP会自动加上%
in(条件, 字段名, 值集合/数组)	当条件为true时才生效, IN 查询	
notIn(条件, 字段名, 值集合/数组)	当条件为true时才生效, NOT IN 查询	
isNull(条件, 字段名)	当条件为true时才生效, NULL 值查询	
isNotNull(条件, 字段名)	当条件为true时才生效, IS NOT NULL	

## LambdaQueryWrapper的API

### 1. 创建LambdaQueryWrapper对象

方式一: `LambdaQueryWrapper wrapper = new LambdaQueryWrapper<实体类>()`

方式二: `LambdaQueryWrapper wrapper = Wrappers.<实体类>lambdaQuery();`

### 2. 构造where条件, 常用方法有:

查询方法	说明	备注
eq(javaBean属性, 值)	等于=	字段名=值 条件
ne(javaBean属性, 值)	不等于<>	
gt(javaBean属性, 值)	大于>	
ge(javaBean属性, 值)	大于等于>=	
lt(javaBean属性, 值)	小于<	
le(javaBean属性, 值)	小于等于<=	
like(javaBean属性, 值)	模糊查询 LIKE	MybatisPlus会自动加上%
notLike(javaBean属性, 值)	模糊查询 NOT LIKE	MybatisPlus会自动加上%
in(javaBean属性, 值集合/数组)	IN 查询	

查询方法	说明	备注
notIn(JavaBean属性,值集合/数组)	NOT IN 查询	
isNull(JavaBean属性)	NULL 值查询	
isNotNull(JavaBean属性)	IS NOT NULL	

💡 如果需要动态拼接SQL条件，可使用以下方法重载：

查询方法	说明	备注
eq(条件, JavaBean的属性,值)	条件为true时才生效，等于=	字段名=值
ne(条件,JavaBean的属性,值)	条件为true时才生效，不等与<>	
gt(条件,JavaBean的属性,值)	条件为true时才生效，大于>	
ge(条件,JavaBean的属性,值)	条件为true时才生效，大于等于>=	
lt(条件,JavaBean的属性,值)	条件为true时才生效，小于<	
le(条件,JavaBean的属性,值)	条件为true时才生效，小于等于<=	
like(条件,JavaBean的属性,值)	条件为true时才生效，模糊查询 LIKE	会自动加上%
notLike(条件,JavaBean的属性,值)	条件为true时才生效，模糊查询 NOT LIKE	会自动加上%
in(条件,JavaBean的属性,值集合/数组)	条件为true时才生效，IN 查询	
notIn(条件,JavaBean的属性,值集合/数组)	条件为true时才生效，NOT IN 查询	
isNull(条件,JavaBean的属性)	条件为true时才生效，NULL 值查询	
isNotNull(条件,JavaBean的属性)	条件为true时才生效，IS NOT NULL	

## 使用示例

```
@SpringBootTest
public class Demo02MpWrapperTest {
    @Resource
    private UserMapper userMapper;

    /**
     * QueryWrapper使用示例：
     * 查询用户名包含a，余额在1000(包含)~10000(包含)之间的用户信息
     */

    @Test
```

```

public void testQueryWrapper() {
    QueryWrapper<User> wrapper = Wrappers.<User>query()
        .like("username", "a")
        .between("balance", 1000, 10000);
    List<User> users = userMapper.selectList(wrapper);
    users.forEach(System.out::println);
}

/**
 * LambdaQueryWrapper使用示例:
 * 查询用户名包含a, 余额在1000(包含)~10000(包含)之间的用户信息
 */
@Test
public void testLambdaQueryWrapper(){
    LambdaQueryWrapper<User> wrapper = Wrappers.<User>lambdaQuery()
        .like(User::getUsername, "a")
        .between(User::getBalance, 1000, 10000);
    List<User> users = userMapper.selectList(wrapper);
    users.forEach(System.out::println);
}
}

```

### 3.3 条件修改【拓展】

```

/**
 * UpdateWrapper使用示例:
 * 将Jack的密码改为123456
 */
@Test
public void testUpdateWrapper(){
    //方式1:
    // 准备where条件
    UpdateWrapper<User> wrapper1 = new UpdateWrapper<User>()
        .eq("username", "Jack");
    // 准备要set修改的字段值
    User user = new User();
    user.setPassword("123456");
    // 执行update更新
    userMapper.update(user, wrapper1);

    //方式2:
    // 准备where条件和要set修改的字段值
    UpdateWrapper<User> wrapper2 = Wrappers.<User>update()
        .eq("username", "Jack")
        .set("password", "123456");
    // 执行update更新
    int result = userMapper.update(null, wrapper2);
    System.out.println("result = " + result);
}

/**

```

```

* LambdaUpdateWrapper使用示例:
* 将Jack的密码改为666666
*/
@Test
public void testLambdaUpdateWrapper(){
    //方式1:
    LambdaUpdateWrapper<User> wrapper1 = new LambdaUpdateWrapper<User>()
        .eq(User::getUsername, "Jack");

    User user = new User();
    user.setPassword("666666");

    userMapper.update(user, wrapper1);

    //方式2:
    LambdaUpdateWrapper<User> wrapper2 = Wrappers.<User>lambdaUpdate()
        .eq(User::getUsername, "Jack")
        .set(User::getPassword, "666666");
    int result = userMapper.update(null, wrapper2);
    System.out.println("result = " + result);
}

```

## 4. 分页查询

### 4.1 介绍

参考: <https://baomidou.com/plugins/pagination/>

MybatisPlus本身就内置分页插件, 不需要再额外导入任何插件(像PageHelper), 也不需要我们再手动实现了。

只需要2步:

1. 配置分页插件: `PaginationInnerInterceptor`
2. 调用分页查询API: `BaseMapper`提供了方法 `IPage selectPage(IPage page, Wrapper wrapper)`  
 Wrapper: 查询条件, 略  
 IPage: 用于封装分页查询条件和分页查询结果

### 4.2 使用示例

#### 4.2.1 配置分页插件

在配置类或者引导类里配置MP插件

```

@Bean
public MybatisPlusInterceptor mybatisPlusInterceptor() {
    //创建MP插件容器
    MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();

    //添加分页插件，SQL方言为MySQL。可以添加多个插件到插件容器里，但分页插件要最后添加
    interceptor.addInnerInterceptor(new PaginationInnerInterceptor(DbType.MYSQL));

    return interceptor;
}

```

#### 4.2.2 实现分页查询

```

@SpringBootTest
public class Demo03MpPageTest {
    @Resource
    private UserMapper userMapper;

    @Test
    public void testPage(){
        //准备分页条件对象：new Page(页码, 每页几条)
        IPage<User> page = new Page<>(1, 2);
        //准备查询条件对象：where status = 1
        LambdaQueryWrapper<User> wrapper = Wrappers.<User>lambdaQuery().eq(User::getStatus, 1);

        //执行分页查询，得到结果：仍然是IPage对象
        page = userMapper.selectPage(page, wrapper);

        //获取分页查询结果
        System.out.println("总数量：" + page.getTotal());
        System.out.println("总页数：" + page.getPages());
        List<User> users = page.getRecords();
        users.forEach(System.out::println);
    }
}

```

## 5. 小结

1. MP的BaseMapper提供的常用方法有哪些：

查询方法：以select开头

新增方法：以insert开头

修改方法：以update开头

删除方法：以delete开头

2. 实体类名对应表名：用@TableName("表名")

3. 主键字段的注解：@TableId(value="字段名", type=主键生成策略)

AUTO：主键自增

INPUT：我们的代码手动赋值



ASSIGN\_ID: 雪花算法

ASSIGN\_UUID: UUID算法

#### 4. 条件查询

QueryWrapper

```
//1. 创建对象 2.添加条件
QueryWrapper<实体类> wrapper = Wrappers.<实体类>query()
    .ge("字段名", 值)
    .like("字段名", 值)
    ....;
```

LambdaQueryWrapper

```
//1. 创建对象 2.添加条件
LambdaQueryWrapper<实体类> wrapper = Wrappers.<实体类>lambdaQuery()
    .ge(实体类::getXxx, 值)
    .like(实体类::getXxx, 值)
    ....;
```

#### 5. 分页查询

第1步：配置分页插件。在任意一个配置类里使用@Bean添加以下代码

```
@Bean
public MybatisPlusInterceptor mybatisPlusInterceptor() {
    //创建MybatisPlus拦截器 容器
    MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();

    //向容器里可以添加多个拦截器（插件）
    //1. 添加一个乐观锁插件
    // interceptor.addInnerInterceptor(new OptimisticLockerInnerInterceptor());
    //2. 添加一个分页插件
    interceptor.addInnerInterceptor(new PaginationInnerInterceptor(DbType.MYSQL)); // 如果配置多个插件, 切记
    分页最后添加

    // 如果有多数据源可以不配具体类型, 否则都建议配上具体的 DbType
    return interceptor;
}
```

第2步：调用Mapper的方法 selectPage( IPage page, Wrapper wrapper )

```
//1. 执行分页查询
Page page = xxxMapper.selectPage( new Page(页码,每页几条), wrapper );
//2. 获取分页结果
获取总记录数量: page.getTotal();
获取总页数:    page.getPages();
获取数据列表:  page.getRecords();
```

## 四、IService接口

使用idea打开《day01-mp2-service》，在这个工程里演示Service的使用

### 1. IService接口介绍

我们的Mapper接口因为继承了BaseMapper，所以不需要写任何方法就继承了一批单表CURD方法可用

同样的，MP也为Service层提供了基类供我们继承。灵活使用这些方法，将会大大提升我们的开发效率

- 我们的Service接口要继承 `IService<实体类>`
- 我们的Service实现类要继承 `ServiceImpl<Mapper接口, 实体类>`

创建Service层的接口（也可以使用MybatisPlus插件直接生成）

```
public interface IUserService extends IService<User> {
}
```

创建Service层的实现类（也可以使用MybatisPlus插件直接生成）

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

### 2. IService基本用法【了解】

API说明：

- 查询数量：方法名全部以 `count` 开头
- 查询列表：方法名全部以 `list` 开头
- 查询一条：方法名全部以 `get` 开头
- 分页查询：方法名全部以 `page` 开头
- 删除数据：方法名全部以 `remove` 开头
- 保存数据：方法名全部以 `save` 开头
- 修改数据：方法名全部以 `update` 开头

使用示例：

```
@SpringBootTest
public class Demo01MpServiceCurdTest {
```

```

@Resource
private IUserService userService;

@Test
public void testSave(){
    //新增一条数据
    User user = new User();
    user.setUsername("Pony");
    user.setPassword("123456");
    user.setInfo("谁都别拦着我学习");
    boolean success = userService.save(user);
    System.out.println("success = " + success);
}

@Test
public void testQuery(){
    //查询数量
    long count = userService.count();
    System.out.println("count = " + count);

    //查询列表
    List<User> users = userService.list();
    users.forEach(System.out::println);

    //查询单个
    User user = userService.getById(1L);
    System.out.println("user = " + user);

    //分页查询
    Page<User> page = userService.page(new Page<>(1, 2));
    System.out.println("page = " + page);
}

@Test
public void testUpdate(){
    User user = userService.getById(1L);
    user.setUsername("Jack");
    boolean success = userService.updateById(user);
    System.out.println("success = " + success);
}

@Test
public void testDelete(){
    boolean success = userService.removeById(1L);
    System.out.println("success = " + success);
}
}

```

### 3. IService的lambda链式操作【必会】

MP的Service基类提供了强大的链式操作查询与链式修改方法，在实际开发中非常有用。

## 3.1 语法说明

### 链式查询

```
lambdaQuery()  
.条件方法(JavaBean属性, 值) //有eq、gt、ge、lt、le、like、between、in、notIn等一系列条件方法  
.条件方法(条件, JavaBean属性, 值) //每个条件方法都有重载，用于动态拼接SQL语句  
.终结方法() //list()获取列表；count()查询数量；one()获取一个；page()分页查询
```

### 链式修改

```
lambdaUpdate()  
.条件方法(JavaBean属性, 值) //有eq、gt、ge、lt、le、like、between、in、notIn等一系列条件方法  
.条件方法(条件, JavaBean属性, 值) //每个条件方法都有重载，用于动态拼接SQL语句  
.set(JavaBean属性, 值) //要修改的字段值  
.update(); //终结方法，必须有！否则update语句是不会执行的
```

## 3.2 使用示例

在单元测试类中添加方法：

```
@SpringBootTest  
public class Demo02MpServiceLambdaTest {  
    @Resource  
    private IUserService userService;  
  
    /**  
     * lambdaQuery方法：查询status为1的，余额大于1000的用户列表  
     */  
    @Test  
    public void testQueryList() {  
        List<User> users = userService.lambdaQuery()  
            .eq(User::getStatus, 1)  
            .ge(User::getBalance, 1000)  
            .list();  
  
        users.forEach(System.out::println);  
    }  
  
    /**  
     * lambdaQuery方法：查询status为1的，余额大于1000的用户数量  
     */  
    @Test  
    public void testQueryCount(){  
        Long count = userService.lambdaQuery()  
            .eq(User::getStatus, 1)  
            .ge(User::getBalance, 1000)  
            .count();  
        System.out.println("count = " + count);  
    }  
}
```

```

/**
 * lambdaQuery方法：查询username为Rose的用户（查询一个）
 */
@Test
public void testQueryOne(){
    User user = userService.lambdaQuery()
        .eq(User::getUsername, "Rose")
        .one();
    System.out.println("user = " + user);
}

/**
 * lambdaQuery方法：分页查询status为1的，余额大于1000的用户列表
 */
@Test
public void testQueryPage(){
    Page<User> page = userService.lambdaQuery()
        .eq(User::getStatus, 1)
        .ge(User::getBalance, 1000)
        .page(new Page<>(1, 2));

    //打印总数量
    System.out.println("page.getTotal() = " + page.getTotal());
    //打印总页数
    System.out.println("page.getPages() = " + page.getPages());
    //打印数据列表
    page.getRecords().forEach(System.out::println);
}

/**
 * lambdaUpdate方法：更新username为Rose的用户的密码
 */
@Test
public void testUpdate(){
    boolean success = userService.lambdaUpdate()
        .eq(User::getUsername, "Rose")
        .set(User::getPassword, "123123")
        .update();
    System.out.println("success = " + success);
}
}

```

## 4. IService的批量新增【拓展】

### 4.1 需求说明

现在要往user表里插入10W条数据，该怎么做才会有更高的效率？

- 方式一：for循环10W次，每次调用插入数据的方法
- 方式二：IService的批量插入方法 `saveBatch(集合)`
- 方式三【推荐】：IService的批量插入方法 `saveBatch()` + jdbcUrl参数 `rewriteBatchedStatements=true`

注意：修改application.yml把日志设置为error或者直接注释掉，否则程序执行中输出的大量日志会占用很长的时间

```
logging:
  level:
    com.itheima.mapper: error
```

## 4.2 代码演示

### 4.2.1 for循环插入

```
@Test
public void saveBatch1(){
    long start = System.currentTimeMillis();
    for (int i = 0; i < 100000; i++) {
        //准备User对象
        User user = new User();
        user.setUsername("pony" + i);
        user.setPassword("123456");
        user.setInfo("Pony邀请你充值");

        //保存到数据库
        userService.save(user);
    }
    long end = System.currentTimeMillis();
    //耗时：210781ms
    System.out.println("耗时：" + (end - start) + "ms");
}
```


### 4.2.2 IService的批量插入

```
@Test
public void saveBatch2(){
    long start = System.currentTimeMillis();
    List<User> users = new ArrayList<>();
    for (int i = 0; i < 100000; i++) {
        //准备User对象
        User user = new User();
        user.setUsername("pony" + i);
        user.setPassword("123456");
        user.setInfo("Pony邀请你充值");
        //添加到集合里
        users.add(user);
    }
    userService.saveBatch(users);

    long end = System.currentTimeMillis();
    //17823ms
    System.out.println("耗时：" + (end - start) + "ms");
}
```

### 4.2.3 IService的批量插入+ jdbcUrl参数

在方式二的基础上，再添加jdbcUrl参数 `rewriteBatchedStatements=true`。这个参数的作用是：允许改造SQL  
insert into user (...) values (...), (...), (...)



```
application.yml x  IUserService.java  Demo04MpServiceTest.java  UserServiceImpl.java
1  spring:
2    datasource:
3      driver-class-name: com.mysql.cj.jdbc.Driver
4      url: jdbc:mysql://localhost:3306/mp?rewriteBatchedStatements=true #允许批量重写
5      username: root
6      password: root
7  logging:
```

```
@Test
public void saveBatch2(){
    long start = System.currentTimeMillis();
    List<User> users = new ArrayList<>();
    for (int i = 0; i < 100000; i++) {
        //准备User对象
        User user = new User();
        user.setUsername("pony" + i);
        user.setPassword("123456");
        user.setInfo("Pony邀请你充值");
        //添加到集合里
        users.add(user);
    }
    userService.saveBatch(users);

    long end = System.currentTimeMillis();
    //17823ms 修改jdbcUtils参数rewriteBatchedStatements=true后 耗时: 5567ms
    System.out.println("耗时: " + (end - start) + "ms");
}
```

## 5. 小结

MP的Service层封装，要求：

- 我们的Service接口，要继承 `IService<实体类>`
- 我们的Service实现类，要继承 `ServiceImpl<Mapper接口,实体类>`

之后：我们就可以直接调用Service的单表CURD方法了

基本方法有：

- 查询方法有：这些方法可以全部改用lambdaQuery()代替掉  
查询列表的方法名以list开头  
查询一条的方法名以get开头

查询数量的方法名以count开头

分页查询的方法名以page开头

- 新增方法：以save开头
- 修改方法：以update开头。可以使用lambdaUpdate()代替掉
- 删除方法：以remove开头

lambdaQuery()方法的使用

```
service对象.lambdaQuery()  
    .条件方法(JavaBean属性,值) //常用的条件方法有： eq, gt, lt, ge, le, like, between, in, ...  
    .条件方法(JavaBean属性,值)  
    .终结方法(); //终结方法有： list()查列表； page()分页查； one()查一条； count()查数量
```

lambdaUpdate()方法的使用

```
//最终MP会帮我们拼接成update 表名 set 字段1=值1,字段2=值2 where 条件1 and 条件2 ...  
boolean success = service对象.lambdaUpdate()  
    .set(JavaBean属性,值)  
    .set(JavaBean属性,值)  
    .条件方法(JavaBean属性,值) //常用的条件方法有： eq, gt, lt, ge, le, like, between, in, ...  
    .条件方法(JavaBean属性,值)  
    ...  
    .update();
```

## 五、扩展功能【了解】

使用idea打开《day01-mp3-ext》，在这个工程里演示扩展功能

### 1. 逻辑删除

#### 2.1 说明

##### 什么是逻辑删除

如果需要删除一条数据，开发中往往有两种方案可以实现：

- 物理删除：真正的从数据库中把数据删除掉
- 逻辑删除：有一个字段用于标识数据是否删除的状态。删除时仅仅是把字段设置为“已删除”状态，数据还在数据库里，并非真正的删除

在实际开发中，逻辑删除使用的更多一些。所以MP也提供了逻辑删除的支持，帮助我们更方便的实现逻辑删除

##### MP的逻辑删除用法

使用步骤：

1. 修改数据库表，增加一个字段 `deleted`。字段名称可以随意，仅仅是用于存储数据的状态的
2. 修改实体类，增加对应的属性，并在属性上添加注解 `@TableLogic`



3. 修改配置文件 `application.yaml`，声明删除与未删除的字面值

MP逻辑删除的本质是：

- 当执行删除时，MP实际上执行的是update操作，把状态字段修改为“已删除状态”
- 当执行查询时，MP会帮我们加上一个条件 `状态字段 = 未删除`，从而只查询未删除的数据

## 2.2 示例

### 2.2.1 增加状态字段

执行SQL语句：

```
use mp;
alter table `user` add deleted int default 0;
```

### 2.2.2 增加属性并加@TableLogic注解

```
@TableName("user")
public class User implements Serializable {

    ...略...

    /**
     * 是否删除（1正常 2删除）。逻辑删除字段，添加@TableLogic注解
     */
    @TableLogic
    private Integer deleted;
}
```

### 2.2.3 修改配置文件

```
mybatis-plus:
  global-config:
    db-config:
      #logic-delete-field: deleted #全局的默认逻辑删除字段名，即 状态字段名。
      logic-delete-value: 1 #已删除状态的值
      logic-not-delete-value: 0 #未删除状态的值
```

### 2.2.4 测试

```

@Test
public void testLogicDelete(){
    //删除id为5的数据
    userService.removeById(5L);

    //查询所有数据，查询结果中没有id为5的数据。但是数据库里id为5的数据还在，只是deleted为1（已删除状态）
    List<User> users = userService.list();
    for (User user : users) {
        System.out.println(user);
    }
}

```

## 2. 枚举处理器

### 2.1 介绍

数据库表里的数据，通常会有一些状态字段，比如：user表里的status字段，值为1时表示正常，值为2时表示冻结。

这样的值，是不允许开发者随意赋值的，为了规范取值，通常会使用枚举：实体类里使用枚举类型的属性，对应这个字段。

### 2.2 使用示例

#### 1. 准备枚举类

```

@Getter
public enum UserStatus {
    NORMAL(1, "正常"),
    LOCKED(2, "冻结");

    /**
     * 使用枚举项的value值，与数据库表字段值对应
     */
    @EnumValue
    private final int value;
    private final String desc;

    UserStatus(int value, String desc) {
        this.value = value;
        this.desc = desc;
    }
}

```

#### 2. 修改实体类

```

@Data
@TableName("user")
public class User implements Serializable {

```

```

@TableId(value = "id", type = IdType.ASSIGN_ID)
private Long id;
private String username;
private String password;
private String phone;
private String info;

/**
 * 使用状态（1正常 2冻结），使用枚举类型UserStatus
 */
private UserStatus status;

private Integer balance;
private LocalDateTime createTime;
private LocalDateTime updateTime;
@TableLogic
private Integer deleted;
}

```

### 3. 配置枚举处理器

修改application.yaml，增加配置参数

```

mybatis-plus:
  configuration:
    default-enum-type-handler: com.baomidou.mybatisplus.core.handlers.MybatisEnumTypeHandler

```

### 4. 功能测试

```

@SpringBootTest
public class Demo01MpEnumTest {
    @Resource
    private UserMapper userMapper;

    /**
     * 验证枚举处理器是否生效：从数据库里查询时，能正常得到枚举值
     */
    @Test
    public void testEnum1() {
        // 查询用户列表
        List<User> users = userMapper.selectList(null);
        // 得到用户列表，每个用户的status都获取到了值
        users.forEach(System.out::println);
    }

    /**
     * 验证枚举处理器是否生效；新增用户时，枚举值能正确写入数据库
     */
    @Test
    public void testEnum2(){
        User user = new User();

```

```
user.setUsername("张三");
user.setPassword("123456");
user.setStatus(UserStatus.LOCKED);
user.setInfo("高富帅");
userMapper.insert(user);
}
}
```

### 3. 小结

逻辑删除：并不是真正删除数据，而是执行update，把数据状态修改为“已删除”状态

1. 表里增加一个字段，用于存储数据的状态是否删除
2. 实体类里增加一个属性，属性上加@TableLogic
3. 修改配置文件application.yaml，设置 已删除的状态值 和 未删除的状态值

枚举处理器：如果实体类里的一个属性，是枚举，就需要在枚举类里成员变量上加@EnumValue

- 当MP操作数据库时，会使用枚举项的指定属性值 对应数据库表

```
@Getter
public enum UserStatus {
    NORMAL(1,"正常"),
    LOCKED(2, "冻结");

    @EnumValue
    private final int value;
    private final String desc;

    UserStatus(int value, String desc) {
        this.value = value;
        this.desc = desc;
    }
}
```

乐观锁