MyBatis-Plus

1、基于 Mapper 接口 CRUD

通用 CRUD 封装 **BaseMapper 接口**，Mybatis-Plus 启动时**自动解析实体表关系映射转换为 Mybatis 内部对象**并**注入容器**，包含常见单表操作。

在 UserMapper.java 中：（**泛型类型填要操作的实体对象**）

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

}
```

测试效果：

```
public class MyBatisPlusTest{
    @Autowired
    private UserMapper userMapper;

    @Test
    public void test_insert(){
        User user =new User;
        user.setAge(25);
        user.setName("Tom");
        user.setEmail("xxx");
        //baseMapper提供的数据库插入方法
        int row = userMapper.insert(user);
    }

    @Test
    public void test_delete(){
        Map param = new HashMap();
        param.put("age",20);
        //baseMapper提供的数据库删除方法
        int row = userMapper.deleteByMap(param);
    }
```

```
    @Test
    public void test_update(){
        //update 当属性值为null的时候，不修改
        User user =new User;
        user.setId(1L);
        user.setAge(20);
        //update user set age = 20 where id = 1, 这里的name就不会修改
        int row = userMapper.updateById(user);

        //将所有人的年龄改为30
        User user1 =new User;
        user1.setAge(30);
        int row = userMapper.update(user1,null);null表示没有条件

    }

    @Test
    public void test_select(){
        List<long> ids = new ArrayList();
        ids.add(1L);ids.add(2L);
        //根据id批量查询
        List<User> users = userMapper.selectBatchIds(ids);
    }
```

2、基于 Service 接口 CRUD(具体方法看官方文档)

通用 Service CRUD 封装 **IService 接口**，进一步封装 CRUD 采用 **get 查询单行**、**remove 删除**、**list 查询集合**、**page 分页**的前缀命名方式区分 Mapper 层避免混淆。

1）**对比 Mapper 接口 CRUD 区别：**

1> service 添加了**批量方法**；2> service 层的方法**自动添加事务**。

2)使用 Iservice 接口方式(两步继承，IService 中有些方法没实现)

1> **UserService 接口继承 IService 接口**

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

2> **UserServiceImpl 类继承 ServiceImpl 实现类**

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

}
```

3、分页查询实现

1）**导入分页插件**（可在启动类中加，因为其本身就是一个配置类）

MybatisPlusInterceptor 是 MybatisPlus 插件集合，加入集合就能用

```
//mybatis-plus插件加入ioc容器中
@Bean
public MybatisPlusInterceptor mybatisPlusInterceptor() {
    //mybatis-plus的插件集合【加入到这个集合即可，分页插件，乐观锁插件】
    MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
    //分页插件
    interceptor.addInnerInterceptor(new PaginationInnerInterceptor(DbType.MYSQL));
    return interceptor;
}
```

2）**使用分页查询**

（IPage 接口→Page 实现类）

设置分页参数 Page<User> **page** = new Page<>(1, 5);;

最后**结果也会被封装在 page 中**，使用 **page.getXxx()**获取分页数据。

```
@Test
public void testPageQuery(){
    //设置分页参数 (页码，页容量)
    Page<User> page = new Page<>(1, 5);
    userMapper.selectPage(page, null);
    //获取分页数据
    List<User> list = page.getRecords();
    list.forEach(System.out::println);
    System.out.println("当前页: "+page.getCurrent());
    System.out.println("每页显示的条数: "+page.getSize());
    System.out.println("总记录数: "+page.getTotal());
    System.out.println("总页数: "+page.getPages());
    System.out.println("是否有上一页: "+page.hasPrevious());
    System.out.println("是否有下一页: "+page.hasNext());
}
```

3）**自定义的 mapper 方法使用分页**

1> **UserMapper 接口里自定义方法 selectPageVo**

```
//传入参数携带Ipage接口，返回结果为IPage
IPage<User> selectPageVo(IPage<User> page, Integer id);
```

2> **UserMapper.xml 对此接口实现**

```
<select id="selectPageVo" resultType="xxx.xxx.xxx.User">
    SELECT * FROM user WHERE id > #{id}
</select>
```

3> 测试效果

```
@Test
public void testQuick(){

    IPage page = new Page(1,2);

    userMapper.selectPageVo(page,2);

    long current = page.getCurrent();
    System.out.println("current = " + current);
    long pages = page.getPages();
    System.out.println("pages = " + pages);
    long total = page.getTotal();
    System.out.println("total = " + total);
    List records = page.getRecords();
    System.out.println("records = " + records);

}
```

4、条件构造器作用

条件构造器，提供了许多方法来**支持各种条件操作符**，并**可链式调用**。

Wrapper：条件构造**抽象类**，最顶端父类

>**AbstractWrapper**：用于查询条件封装，**生成 sql 的 where 条件**；

>**QueryWrapper**：**查询/删除**条件封装；

>**UpdateWrapper**：**修改**条件封装；

>**AbstractLambdaWrapper**：使用 Lambda 语法**(类名::get 方法名)**；

>**LambdaQueryWrapper**:用于 Lambda 语法使用的查询 Wrapper；

>**LambdaUpdateWrapper**：用于 Lambda 更新封装 Wrapper；

5、基于 QueryWrapper 组装条件

函数名	说明	说明/例子
eq	等于=	例: eq("name", "老王")--->name = '老王'
ne	不等于<>	例: ne("name", "老王")--->name <> '老王'
gt	大于>	例: gt("age", 18)--->age > 18
ge	大于等于>=	例: ge("age", 18)--->age >= 18
lt	小于<	例: lt("age", 18)--->age < 18
le	小于<=	例: le("age", 18)--->age <= 18
between	BETWEEN 值1 AND 值2	例: between("age", 18, 30)--->age between 18 and 30
notBetween	NOT BETWEEN 值1 AND 值2	例: notBetween("age", 18, 30)--->age not between 18 and 30
like	LIKE '%值%'	例: like("name", "王")--->name like '%王%'
notLike	NOT LIKE '%值%'	例: notLike("name", "王")--->name not like '%王%'
likeLeft	LIKE '%值'	例: likeLeft("name", "王")--->name like '%王'
likeRight	LIKE '值%'	例: likeRight("name", "王")--->name like '王%'
isNull	字段 IS NULL	例: isNull("name")--->name is null
isNotNull	字段 IS NOT NULL	例: isNotNull("name")--->name is not null
in	字段 IN (v0, v1, ...)	例: in("age",{1,2,3})--->age in (1,2,3)
notIn	字段 NOT IN (v0, v1, ...)	例: notIn("age", 1, 2, 3)--->age not in (1,2,3)
inSql	字段 IN (sql语句)	inSql("id", "select id from table where id < 3") --->id in (select id from table where id < 3)
notInSql	字段 NOT IN (sql语句)	notInSql("id", "select id from table where id < 3") --->age not in (select id from table where id < 3)
groupBy	分组：GROUP BY 字段, ...	例: groupBy("id", "name")--->group by id,name
orderByAsc	排序：ORDER BY 字段, ... ASC	例: orderByAsc("id", "name")--->order by id ASC,name ASC
orderByDesc	排序：ORDER BY 字段, ... DESC	例: orderByDesc("id", "name")--->order by id DESC,name DESC
orderBy	排序：ORDER BY 字段, ...	例: orderBy(true, true, "id", "name") --->order by id ASC,name ASC
having	HAVING (sql语句)	having("sum(age) > {0}", 11)--->having sum(age) > 11
or	拼接 OR	注意事项: 主动调用or表示紧接着下一个方法不是用and连接!(不调用or则默认为使用and连接) 例:eq("id",1).or().eq("name","老王")--->id = 1 or name = '老王
and	AND 嵌套	例: and(i -> i.eq("name", "李白").ne("status", "活着")) --->and (name = '李白' and status <> '活着')

QueryWrapper<User> queryWrapper = new QueryWrapper<>();
userMapper.xXxx(queryWrapper)执行条件查询
组装查询条件：

```
@Test
public void test01(){
    //查询用户名包含a，年龄在20到30之间，并且邮箱不为null的用户信息
    //SELECT id,username AS name,age,email,is_deleted FROM t_user WHERE is_deleted=0 AND (username LIKE ? AND age BETWEEN ? AND ? AND email IS NOT NULL)
    QueryWrapper<User> queryWrapper = new QueryWrapper<>();
    queryWrapper.like("username", "a")
                .between("age", 20, 30)
                .isNotNull("email");
    List<User> list = userMapper.selectList(queryWrapper);
    list.forEach(System.out::println);
}
```

组装排序条件：

```
@Test
public void test02(){
    //按年龄降序查询用户，如果年龄相同则按id升序排列
    //SELECT id,username AS name,age,email,is_deleted FROM t_user WHERE is_deleted=0 ORDER BY age DESC,id ASC
    QueryWrapper<User> queryWrapper = new QueryWrapper<>();
    queryWrapper
            .orderByDesc("age")
            .orderByAsc("id");
    List<User> users = userMapper.selectList(queryWrapper);
    users.forEach(System.out::println);
}
```

组装删除条件：

```
@Test
public void test03(){
    //删除email为空的用户
    //DELETE FROM t_user WHERE (email IS NULL)
    QueryWrapper<User> queryWrapper = new QueryWrapper<>()
    queryWrapper.isNull("email");
    //条件构造器也可以构建删除语句的条件
    int result = userMapper.delete(queryWrapper);
    System.out.println("受影响的行数: " + result);
}
```

and 和 or 关键字使用(修改)：
【注意】条件**默认**使用 **and 拼接**，.or()**只起一次作用**；
使用“**queryWrapper+实体类**”形式**可以实现修改**，但是**无法将列值修改为 null 值**。即：userMapper.update(user,queryWrapper)

```
@Test
public void test04() {
    QueryWrapper<User> queryWrapper = new QueryWrapper<>();
    //将年龄大于20并且用户名中包含有a或邮箱为null的用户信息修改
//UPDATE t_user SET age=?, email=? WHERE username LIKE ? AND age > ? OR email IS NULL)
    queryWrapper
            .like("username", "a")
            .gt("age", 20)
            .or()
            .isNull("email");
    User user = new User();
    user.setAge(18);
    user.setEmail("user@atguigu.com");
    int result = userMapper.update(user, queryWrapper);
    System.out.println("受影响的行数: " + result);
}
```

指定列映射的查询：（避免 User 对象中没有被查询到的列值为 null）
queryWrapper.select(“指定列”);selectMaps()返回 Map 集合列表。

```
@Test
public void test05() {
    //查询用户信息的username和age字段
    //SELECT username,age FROM t_user
    QueryWrapper<User> queryWrapper = new QueryWrapper<>();
    queryWrapper.select("username", "age");
    //selectMaps()返回Map集合列表，通常配合select()使用，避免User对象中没有被查询到的列值为null
    List<Map<String, Object>> maps = userMapper.selectMaps(queryWrapper);
    maps.forEach(System.out::println);
}
```

condition 判断组织条件：（第一个参数为判断条件）

```
@Test
public void testQuick3(){
    String name = "root";
    int    age = 18;

    //方案2: 拼接condition判断
    //每个条件拼接方法都condition参数,这是一个比较运算,为true追加当前条件!
    //eq(condition,列名,值)
    queryWrapper.eq(!StringUtils.isEmpty(name),"name",name)
                .eq(age>1,"age",age);
}
```

6、基于 UpdateWrapper 组装条件（实体对象写 null）

updateWrapper.set()设置新值，
使用 **updateWrapper** 可以随意设置列的值，**可以设置为 null**。

```
@Test
public void testQuick2(){
    UpdateWrapper<User> updateWrapper = new UpdateWrapper<>();
    //将id = 3 的email设置为null, age = 18
    updateWrapper.eq("id",3)
                 .set("email",null)  // set 指定列和结果
                 .set("age",18);
    //如果使用updateWrapper 实体对象写null即可!
    int result = userMapper.update(null, updateWrapper);
    System.out.println("result = " + result);
}
```

7、基于 LambdaQueryWrapper 组装条件（类名::get 方法名）

相比于 QueryWrapper，LambdaQueryWrapper 使用了**实体类的属性引用**（例如 User::getName、User::getAge），而**不是字符串来表示字段名**。
QueryWrapper 示例代码：

```
QueryWrapper<User> queryWrapper = new QueryWrapper<>();
queryWrapper.eq("name", "John")
            .ge("age", 18)
            .orderByDesc("create_time")
            .last("limit 10");
List<User> userList = userMapper.selectList(queryWrapper);
```

LambdaQueryWrapper 示例代码：

```
LambdaQueryWrapper<User> lambdaQueryWrapper = new LambdaQueryWrapper<>();

lambdaQueryWrapper.eq(User::getName, "John")
                  .ge(User::getAge, 18)
                  .orderByDesc(User::getCreateTime)
                  .last("limit 10");
List<User> userList = userMapper.selectList(lambdaQueryWrapper);
```

8、基于 LambdaUpdateWrapper 组装条件（**实体对象写 null**）

```
@Test
public void testQuick2(){

    UpdateWrapper<User> updateWrapper = new UpdateWrapper<>();
    //将id = 3 的email设置为null, age = 18
    updateWrapper.eq("id",3)
                 .set("email",null)  // set 指定列和结果
                 .set("age",18);

    //使用lambdaUpdateWrapper
    LambdaUpdateWrapper<User> updateWrapper1 = new LambdaUpdateWrapper<>();
    updateWrapper1.eq(User::getId,3)
                  .set(User::getEmail,null)
                  .set(User::getAge,18);

    //如果使用updateWrapper 实体对象写null即可!
    int result = userMapper.update(null, updateWrapper);
    System.out.println("result = " + result);
}
```

9、核心注解使用 - @TableName 注解
描述：表名注解，标识**实体类对应的表**；
使用位置：**实体类**。
特殊情况：如果**表名和实体类名相同**（忽略大小写）可以**省略该注解**。

```
@TableName("sys_user") //对应数据库表名
public class User {
    private Long id;
    private String name;
    private Integer age;
    private String email;
}
```

也可以**全局设置前缀**：（table-prefix:）

```
mybatis-plus: # mybatis-plus的配置
  global-config:
    db-config:
      table-prefix: sys_ # 表名前缀字符串
```

10、核心注解使用 - @TableId 注解
描述：**主键注解**；
使用位置：**实体类主键字段**。

```
@TableName("sys_user")
public class User {
    @TableId(value="主键列名",type=主键策略)
    private Long id;
    private String name;
    private Integer age;
    private String email;
}
```

属性	类型	必须指定	默认值	描述
value	String	否	""	主键字段名
type	Enum	否	IdType.NONE	指定主键类型

IdType 属性可选值：
1> **AUTO** 主键自增长的前提，mysql **数据库中主键**必须设置自增长 **AUTO_INCREAMENT**；
2> **默认为雪花算法**，随机产生一个不重复的数字给主键值；同时需要满足**数据库主键**为 **bigint/varchar(64)**；**实体类**为 **Long 类型**。

值	描述
AUTO	数据库 ID 自增 (mysql配置主键自增长)
ASSIGN_ID（默认）	分配 ID(主键类型为 Number(Long)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)

全局配置修改主键策略：（id-type）

```
mybatis-plus:
  configuration:
    # 配置MyBatis日志
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
  global-config:
    db-config:
      # 配置MyBatis-Plus操作表的默认前缀
      table-prefix: t_
      # 配置MyBatis-Plus的主键策略
      id-type: auto
```

11、核心注解使用 - @TableField 注解
描述：**字段注解（非主键）**
Value 值匹配数据库字段名；
exist 表示**是否为数据库表字段**(默认 true)

```
@TableName("sys_user")
public class User {
    @TableId
    private Long id;
    @TableField(value="nickname", exist=false)
    private String name;
    private Integer age;
    private String email;
}
```

12、逻辑删除实现
逻辑删除，可以方便地实现对数据库记录的逻辑删除而不是物理删除。**逻辑删除**是指通过**更改记录的状态**或**添加标记字段**来模拟删除操作，从而**保留了删除前的数据**，便于后续的数据分析和恢复。
- 物理删除：真实删除，将对应数据从数据库中删除，之后查询不到此条被删除的数据；
- 逻辑删除：假删除，将对应数据中代表**是否被删除字段的状态修改为“被删除状态”**，之后在数据库中仍旧能看到此条数据记录。

逻辑删除实现：
1）数据库和实体类添加逻辑删除字段
1> **表添加逻辑删除字段**：可以是一个布尔类型、整数类型或枚举类型

```
ALTER TABLE USER ADD deleted INT DEFAULT 0 ;  # int 类型 1 逻辑删除 0 未逻辑删除
```

2> **实体类添加逻辑删除属性**（使用**@TableLogic 注解**）

```
@Data
public class User {

   // @TableId
    private Integer id;
    private String name;
    private Integer age;
    private String email;

    @TableLogic
    //逻辑删除字段 int mybatis-plus下,默认 逻辑删除值为1 未逻辑删除 1
    private Integer deleted;
}
```

2）指定逻辑删除字段和属性值
1> 单一指定（使用**@TableLogic 注解**）

```
@Data
public class User {

   // @TableId
    private Integer id;
    private String name;
    private Integer age;
    private String email;
     @TableLogic
    //逻辑删除字段 int mybatis-plus下,默认 逻辑删除值为1 未逻辑删除 1
    private Integer deleted;
}
```

2> 全局指定（**logic-delete-field**）

```
mybatis-plus:
  global-config:
    db-config:
      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)
      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)
```

3） **逻辑删除**以后，没有真正的删除语句，**删除改为修改语句**

```
//逻辑删除
@Test
public void testQuick5(){
    //逻辑删除
    userMapper.deleteById(5);
}
```

执行效果:

```
\==> Preparing: UPDATE user SET deleted=1 WHERE id=? AND deleted=0
\==> Parameters: 5(Integer)
<==    Updates: 1
```

4）测试查询数据（**默认查询非逻辑删除数据**）

```
@Test
public void testQuick6(){
    //正常查询.默认查询非逻辑删除数据
    userMapper.selectList(null);
}

//SELECT id,name,age,email,deleted FROM user WHERE deleted=0
```

13、乐观锁实现

1）悲观锁和乐观锁场景和介绍

1> 悲观锁的基本思想是，**"先保护，再修改"**，在整个数据**访问过程**中，将**共享资源锁定**，以确保其他线程或进程**不能同时访问和修改该资源**。在悲观锁的应用中，线程在**访问共享资源之前会获取到锁**，并在整个操作过程中**保持锁的状态，阻塞其他线程的访问**。只有当前线程**完成操作后**，才会**释放锁**，让**其他线程继续操作**资源。这种锁机制可以确保**资源独占性**和**数据的一致性**，但是在**高并发环境**下的**效率相对较低**。

2> 乐观锁的基本思想是，**"先修改，后校验"**，认为**并发冲突的概率较低**，因此不需要提前加锁，而是**在数据更新阶段**进行**冲突检测和处理**。在乐观锁的应用中，线程在**读取共享资源时不会加锁**，而是**记录特定的版本信息**。当线程**准备更新资源时**，会先**检查该资源的版本信息是否**与之前读取的版本信息**一致**，如果**一致则执行**更新操作，**否则**说明有其他线程修改了该资源，需要**进行相应的冲突处理**。乐观锁通过避免加锁操作，**提高了**系统的**并发性能**和**吞吐量**，但是在并发冲突较为频繁的情况下，乐观锁会**导致较多的冲突处理和重试**操作。

2）具体技术和方案：

1> 乐观锁实现方案和技术：

- **版本号/时间戳**：为数据**添加一个版本号或时间戳字段**，每次更新数据时，**比较当前版本号或时间戳**与**期望值是否一致**，若一致则更新成功，否则表示数据已被修改，需要进行冲突处理；
- **CAS**（Compare-and-Swap）：**使用原子操作**比较当前值与旧值是否一致，若一致则进行更新操作，否则重新尝试；
- **无锁数据结构**：采用无锁数据结构，如无锁队列、无锁哈希表等，通过**使用原子操作**实现并发安全。

2> 悲观锁实现方案和技术：

- **锁机制**：使用传统的锁机制，如**互斥锁(Mutex Lock)**或**读写锁(Read-Write Lock)** 来保证对共享资源的独占访问；
- **数据库锁**：在数据库层面使用**行级锁**或**表级锁**来控制并发访问；
- **信号量**（Semaphore）：使用信号量来限制对资源的并发访问。

3）介绍版本号乐观锁技术的实现流程：

1> 每条数据**添加**一个版本号字段 version；

2> 取出记录时，**获取当前** version；

3> **更新时，检查**获取版本号**是不是**数据库**当前最新版本号**；

4> 如果**是**[证明没人修改数据]，**set 数据更新，version=version+1；**

5> 如果 **version 不对**[证明有人已经修改了]，我们现在的其他记录就是**失效数据**，就**更新失败**。

4）使用 mybatis-plus 数据使用乐观锁

1> 添加版本号更新插件到 MybatisPlusInterceptor 插件集合中

```
@Bean
public MybatisPlusInterceptor mybatisPlusInterceptor() {
    MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
    interceptor.addInnerInterceptor(new OptimisticLockerInnerInterceptor());
    return interceptor;
}
```

2）**数据库**需要**添加 version 字段**

```
ALTER TABLE USER ADD VERSION INT DEFAULT 1 ;  # int 类型 乐观锁字段
```

3）**实体类**中需要**添加乐观锁字段**并**使用@Version 注解**

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

4）正常更新使用即可

```
//演示乐观锁生效场景
@Test
public void testQuick7(){
    //步骤1: 先查询,在更新 获取version数据
    //同时查询两条,但是version唯一,最后更新的失败
    User user  = userMapper.selectById(5);
    User user1  = userMapper.selectById(5);

    user.setAge(20);
    user1.setAge(30);

    userMapper.updateById(user);
    //乐观锁生效,失败!
    userMapper.updateById(user1);
}
```

14、防全表更新和删除实现

针对 update 和 delete 语句，作用：**阻止恶意的全表更新删除**

1）**添加防止全表更新和删除拦截器**到 MybatisPlusInterceptor 插件集合中

```
@Bean
public MybatisPlusInterceptor mybatisPlusInterceptor() {
  MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
  interceptor.addInnerInterceptor(new BlockAttackInnerInterceptor());
  return interceptor;
}
}
```

测试全部更新或者删除

```
@Test
public void testQuick8(){
    User user = new User();
    user.setName("custom_name");
    user.setEmail("xxx@mail.com");
    //Caused by: com.baomidou.mybatisplus.core.exceptions.MybatisPlusException:
Prohibition of table update operation
    //全局更新,报错
    userService.saveOrUpdate(user,null);
}
```