MyBatis-Plus笔记(入门)

作者:故事我忘了¢
个人微信公众号:程序猿的月光宝盒

官方文档

https://mybatis.plus/guide/

本篇基于`springboot`,`mybatis Plus`的版本为`3.4.2`

本篇对应的github地址

https://github.com/monkeyKinn/StudyMyBatisPlus

觉得有用给个Star哦~~~~

Star

Star

Star

建库建表

```
#创建用户表
CREATE TABLE user (
    id BIGINT(20) PRIMARY KEY NOT NULL COMMENT '主键',
    name VARCHAR(30) DEFAULT NULL COMMENT '姓名',
    age INT(11) DEFAULT NULL COMMENT '年龄',
    email VARCHAR(50) DEFAULT NULL COMMENT '邮箱',
    manager_id BIGINT(20) DEFAULT NULL COMMENT '直属上级id',
    create_time DATETIME DEFAULT NULL COMMENT '创建时间',
    CONSTRAINT manager_fk FOREIGN KEY (manager_id)
        REFERENCES user (id)
)  ENGINE=INNODB CHARSET=UTF8;
```

```
#初始化数据:
INSERT INTO user (id, name, age, email, manager_id
    , create_time)
VALUES (1087982257332887553, '大boss', 40, 'boss@baomidou.com', NULL
        , '2019-01-11 14:20:20'),
    (1088248166370832385, '王天风', 25, 'wtf@baomidou.com', 1087982257332887553
        , '2019-02-05 11:12:22'),
    (1088250446457389058, '李艺伟', 28, 'lyw@baomidou.com', 1088248166370832385
        , '2019-02-14 08:31:16'),
    (1094590409767661570, '张雨琪', 31, 'zjq@baomidou.com', 1088248166370832385
        , '2019-01-14 09:15:15'),
    (1094592041087729666, '刘红雨', 32, 'lhm@baomidou.com', 1088248166370832385
        , '2019-01-14 09:48:16');
```

引入依赖

```
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <scope>runtime</scope>
    </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>
      <exclusions>
        <exclusion>
          <groupId>org.junit.vintage</groupId>
          <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
      </exclusions>
    </dependency>

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

    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13</version>
      <scope>test</scope>
```

```
    </dependency>
```

配置文件

```
spring.application.name=MyBatisPlus
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/mp?
useSSL=false&serverTimezone=GMT%2B8
spring.datasource.username=root
spring.datasource.password=admin
```

创建实体类

```
@Data
public class User{
    /** 主键 */
    private Long id;
    /** 姓名 */
    private String name;
    /** 年龄 */
    private Integer age;
    /** 邮件 */
    private String email;
    /** 直属上级id */
    private Long managerId;
    /** 创建时间 */
    private LocalDateTime createTime;
}
```

创建mapper接口

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

启动类上加上注解

```
@SpringBootApplication
// 注意这里的包扫描路径 要写到mapper所在的包名上
@MapperScan("com.jsc.mybatisplus.mapper")
public class MyBatisPlusApplication {

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

}
```

创建测试类

```
@SpringBootTest
@RunWith(SpringRunner.class)
class MyBatisPlusApplicationTests {
    // 后面会用到,此为创建条件构造器
    private QueryWrapper<User> query = Wrappers.query();
    @Autowired
    private UserMapper userMapper;
     @Test
    void selectAllTest() {
        List<User> users = userMapper.selectList(null);
        Assertions.assertEquals(5,users.size());
        users.forEach(System.out::println);
    }

}
```

以上 ,入门练手

通用mapper新增方法

配置文件加上日志输出

```
spring.application.name=MyBatisPlus
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/mp?
useSSL=false&serverTimezone=GMT%2B8
spring.datasource.username=root
spring.datasource.password=admin


#日志输出配置
logging.level.root=warn
#只想看这个包里的sql日志 trace是最低级别
logging.level.com.jsc.mybatisplus.mapper=trace
```

创建测试类

```
    @Test
    void insertTest() {
        User user = new User();
        user.setName("老金");
        user.setAge(18);
        // 没有email 默认不给插入 没有id 是因为默认雪花id
        user.setManagerId(1087982257332887553L);
        user.setCreateTime(LocalDateTime.now());

        int rows = userMapper.insert(user);
        System.out.println("影响行数: " + rows);
    }
```

常用注解

```
@TableName("表名") //在实体类上,用来当db中表和实体类名不一致时候,指定表
@TableId //在字段上,用来指定对应的主键,当数据库中的主键不是id命名时,实体类也不是id时用
@TableField("字段名") //在实体类上,用来指定对应的字段名
```

排除非表字段的三种方式

1.

```
transient //关键字 无法被序列化(存到磁盘)
```

2.

```
static //关键字  只有一份 属于类
```

3.

```
@TableField(exit=false) //默认是true,数据库中存在
```

查询

基本查询

```
    @Test
    void selectByIdTest() {
        // 根据id查询
        User user = userMapper.selectById(1382699085670719489L);
        // User(id=1382699085670719489, name=老金, age=18, email=null,
managerId=1087982257332887553, createTime=2021-04-15T22:15:26)
        System.out.println(user);
    }
    @Test
    void selectBatchIdsTest() {
        // 根据id批量查询
        List<Long> ids = Arrays.asList(1382699085670719489L,
1094592041087729666L, 1094590409767661570L);
        List<User> users = userMapper.selectBatchIds(ids);
        users.forEach(System.out::println);
    }

    @Test
    void selectByMapTest() {
        Map<String, Object> map = new HashMap<>();
        // map.put("name", "老金");
        // map的key是数据库中的字段
        map.put("age", 27);
        // WHERE name = ? AND age = ?
        // WHERE age = ?
        List<User> users = userMapper.selectByMap(map);
        System.out.println(users);
    }
```

以条件构造器为参数

```
    @Test
    void selectByWrapper0() {
        /*
         * 1、名字中包含雨并且年龄小于40
         * name like '%雨%' and age<40
         * 条件构造器
         */

        query.like("name", "雨").lt("age", 40);
        List<User> users = userMapper.selectList(query);
        users.forEach(System.out::println);
    }

    @Test
    void selectByWrapper1() {
        /*
         * 名字中包含雨年并且龄大于等于20且小于等于40并且email不为空
         * name like '%雨%' and age between 20 and 40 and email is not null
         */
        query.like("name", "雨").between("age", 20, 40).isNotNull("email");
        List<User> users = userMapper.selectList(query);
        users.forEach(System.out::println);
    }

    @Test
    void selectByWrapper2() {
        /*
         * 名字为王姓或者年龄大于等于25，按照年龄降序排列，年龄相同按照id升序排列
         *  name like '王%' or age>=25 order by age desc,id asc
         */
        query.like("name", "王").or().ge("age",
18).orderByDesc("age").orderByAsc("id");
        List<User> users = userMapper.selectList(query);
        users.forEach(System.out::println);
    }

    @Test
    void selectByWrapper3() {
        /*
         * 创建日期为2019年2月14日并且直属上级为名字为王姓 --函数开头用apple拼接sql
         * date_format(create_time,'%Y-%m-%d')='2019-02-14' and manager_id in
(select id from user where name like '王%')
         */
        query.apply("date_format(create_time,'%Y-%m-%d')={0}", "2019-02-14")
                .inSql("manager_id", "select id from user where name like
'王%'");

        List<User> users = userMapper.selectList(query);
        users.forEach(System.out::println);
    }

    @Test
    void selectByWrapper4() {
        /*
         * 名字为王姓并且（年龄小于40或邮箱不为空）
```

```
         * name like '王%' and (age<40 or email is not null)
         */
        query.likeRight("name","王").and(qw->qw.lt("age",40
).or().isNotNull("email"));

        List<User> users = userMapper.selectList(query);
        users.forEach(System.out::println);
    }

    @Test
    void selectByWrapper5() {
        /*
         * 名字为王姓或者（年龄小于40并且年龄大于20并且邮箱不为空）
         * name like '王%' or (age<40 and age>20 and email is not null)
         */
        query.likeRight("name","王").or(qw->
                qw.lt("age",40 )
                        .gt("age",20).isNotNull("email"));
        List<User> users = userMapper.selectList(query);
        users.forEach(System.out::println);
    }

    @Test
    void selectByWrapper6() {
        /*
         * （年龄小于40或邮箱不为空）并且名字为王姓,nested 正常嵌套 不带 AND 或者 OR
         * (age<40 or email is not null) and name like '王%'
         */
        query.nested(qw->
                qw.lt("age",40).or().isNotNull("email")).likeRight("name","王");
        List<User> users = userMapper.selectList(query);
        users.forEach(System.out::println);
    }

    @Test
    void selectByWrapper7() {
        /*
         * 年龄为30、31、34、35
         *   age in (30、31、34、35)
         */
        query.in("age",18,30,31,34,35);
        List<User> users = userMapper.selectList(query);
        users.forEach(System.out::println);
    }
```

select中字段不全部出现的查询

```
    @Test
    void selectByWrapper8() {
        /*
         * 10、名字中包含雨并且年龄小于40(需求1加强版)
                    第一种情况：   select id,name -- 就用select("")
                                   from user
                                   where name like '%雨%' and age<40
```

```
                    第二种情况：   select id,name,age,email
                                  from user
                                  where name like '%雨%' and age<40
         */

        query.like("name","雨").lt("age",40).select("name","age");
        List<User> users = userMapper.selectList(query);
        users.forEach(System.out::println);
    }

    @Test
    void selectByWrapper9() {
        /*
         * 10、名字中包含雨并且年龄小于40(需求1加强版)
                    第一种情况：   select id,name -- 就用select("")
                                  from user
                                  where name like '%雨%' and age<40
                    第二种情况：   select id,name,age,email -- 就用select("")
                                  from user
                                  where name like '%雨%' and age<40
         */

        query.like("name","雨").lt("age",40).select(User.class,info ->
                !info.getColumn().equals("create_time")
                        &&!info.getColumn().equals("manager_id"));
        List<User> users = userMapper.selectList(query);
        users.forEach(System.out::println);
    }
```

条件构造器中condition的作用(执行条件.可理解为条件查询)

```
@Test
    void selectByCondition() {
        // 根据参数条件查询
        // String name = "金";
        String name = "玉";
        String age = "18";
        // 如果不为空,就有后面的条件加入到sql中
        query.like(StringUtils.isNotBlank(name),"name",name)
                .eq(StringUtils.isNotBlank(age),"age",age);
        List<User> users = userMapper.selectList(query);
        users.forEach(System.out::println);
    }
```

创建条件构造器时,传入实体对象

```
    @Test
    void selectByWrapperEntity() {
        /*
         * 使用场景:
         *  1、通过实体传过来数据
         *  2.不想永理科这样的构造器,默认是等值的
```

```
         *           如果不想等值,在实体类中加上注解
         *           @TableField(condition=SqlCondition.like)
         *                       小于的话就是="%s&lt;#{%s}"
         *                                   列名<列值
         */
        User whereUser = new User();
        whereUser.setName("老金");
        whereUser.setAge(18);
        query = Wrappers.query(whereUser);
        // query.like("name", "雨").lt("age", 40);
        List<User> users = userMapper.selectList(query);
        users.forEach(System.out::println);
    }
```

条件构造器中allEq用法

```
@Test
    void selectByWrapperAllEq() {
        HashMap<String, Object> params = new HashMap<>();
        params.put("name", "王天风");
        params.put("age", null);

        // 后面的false,是忽略null值的列
        // query.allEq(params,false);
        // 前面一个函数是过滤用的  比如这里就是不包含age列
        query.allEq((k,v)->!"age".equals(k),params);
        List<User> users = userMapper.selectList(query);
        users.forEach(System.out::println);
    }
```

其他以条件构造器为参数的方法

```
    @Test
    void selectByWrapperMaps() {
        /*
        * 使用场景:
        *  1.当表特别的多色时候只要查少数 几列,返回一个map,而不是属性大部分都为空的实体
        *  2.返回的是统计结果
        *       按照直属上级分组，查询每组的 平均年龄、最小龄、最大年龄。
        *           并且只取年龄总和小于500的组。
        *               select
        *                   avg(age) avg_age,
        *                   min(age) min_age,
        *                   max(age) max_age
        *               from user
        *               group by manager_id --上级id分组
        *               having sum(age) <500 --只有总和小于500
        * */
        // 第一种情况
        // query.like("name","雨").lt("age",40).select("id","name");

        // 第二种情况
        query.select("avg(age) avg_age","min(age) min_age","max(age) max_age")
                .groupBy("manager_id")
                .having("sum(age) < {0}",500);
        List<Map<String, Object>> users = userMapper.selectMaps(query);
```

```
        users.forEach(System.out::println);
    }

    @Test
    void selectByWrapperObjs() {
        query.select("id","name").like("name","雨").lt("age",40);
        // 返回第一列的值 只返回一列的时候用
        List<Object> users = userMapper.selectObjs(query);
        users.forEach(System.out::println);
    }

    @Test
    void selectByWrapperCounts() {
        // 不能设置查询的列名了
        query.like("name","雨").lt("age",40);
        // 查总记录数
        Integer count = userMapper.selectCount(query);
        System.out.println(count);
    }

    @Test
    void selectByWrapperOne() {
        // 不能设置查询的列名了
        query.like("name","老金").lt("age",40);
        // 只返回一条数据
        User user = userMapper.selectOne(query);
        System.out.println(user);
    }
```

lambda条件构造器

```
    @Test
    void selectLambda() {
        // 创建方式
        // QueryWrapper<User> userQueryWrapper = new QueryWrapper<>();
        // LambdaQueryWrapper<User> userLambdaQueryWrapper1 = new
LambdaQueryWrapper<>();
        LambdaQueryWrapper<User> lambdaQueryWrapper = Wrappers.lambdaQuery();

        // 好处: 防止误写字段名

        // where name like '%雨%' and age<40
        // 前面是类名 后面是get方法,表示属性
        lambdaQueryWrapper.like(User::getName, "雨").lt(User::getAge, 40);

        List<User> users = userMapper.selectList(lambdaQueryWrapper);
        users.forEach(System.out::println);
    }

    @Test
    void selectLambda1() {
        /*
         * 名字为王姓并且（年龄小于40或邮箱不为空）
         * name like '王%' and (age<40 or email is not null)
         */
        LambdaQueryWrapper<User> lambdaQueryWrapper = Wrappers.lambdaQuery();
        // 好处: 防止误写字段名
```

```
        // 前面是类名 后面是get方法,表示属性
        lambdaQueryWrapper.likeRight(User::getName,"王").and(lqw->lqw.lt(User::getAge,40).or().isNotNull(User::getEmail));

        List<User> users = userMapper.selectList(lambdaQueryWrapper);
        users.forEach(System.out::println);
    }

    @Test
    void selectLambda2() {
        /*
         * 名字为王姓并且（年龄小于40或邮箱不为空）
         * name like '王%' and (age<40 or email is not null)
         */
        // 3.0.7新增创建方式
        List<User> users = new LambdaQueryChainWrapper<>(userMapper)
                .like(User::getName, "雨").ge(User::getAge, 20).list();

        users.forEach(System.out::println);
    }
```

使用条件构造器的自定义SQL

方式1,sql写在接口中

在mapper接口中自定义

```
public interface UserMapper extends BaseMapper<User> {
    // 参数的注解是固定的 ew 就是WRAPPER的值
    @Select("select * from user ${ew.customSqlSegment}")
    List<User> selectAll(@Param(Constants.WRAPPER) Wrapper<User> wrapper);
}
```

测试类

```
    @Test
    void selectAllCustomize() {
        // 自定义的方法

        LambdaQueryWrapper<User> lambdaQueryWrapper = Wrappers.lambdaQuery();
        lambdaQueryWrapper.likeRight(User::getName,"王").and(lqw->lqw.lt(User::getAge,40).or().isNotNull(User::getEmail));
        List<User> users = userMapper.selectAll(lambdaQueryWrapper);
        users.forEach(System.out::println);
    }
```

方式2,sql写在mapper中

在配置文件中配置接口对应的mapper文件

```
#设置扫描路径
mybatis-plus.mapper-locations=mapper/*.xml
```

在resources下建mapper包

省略图

创建xml文件 UserMapper.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.jsc.mybatisplus.mapper.UserMapper">
  <select id="selectAll" resultType="com.jsc.mybatisplus.entity.User">
<!--      参数的注解是固定的 ew 就是WRAPPER的值-->
    select * from user ${ew.customSqlSegment}
  </select>
</mapper>
```

接口中的sql注解就不用了

分页查询

首先明确一点,mybatis分页rowbounds确实实现了分页,但是是逻辑分页/内存分页,他先把数据全查出来,load到memory中,然后给你你想要的,换句话说,是海选...你懂我意思吧...可想而知,数据爆炸的时代,你得给多大的内存,内存不要钱吗,海选可贵,选择又多,多了换一批时间就 慢 ,一个道理~

Mp就提供了物理分页的插件,解决上述问题

既然是插件,肯定要配置

1.创建配置包 config

省略

2.创建配置类

```
package com.jsc.mybatisplus.config;

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

/**
 * mybatis的分页配置
 *
 * @author 金聖聰
 * @version v1.0
 * @email jinshengcong@163.com
 * @date Created in 2021/04/16 15:48
 */
@Configuration
public class MyBatisPlusConfig {
    // 新版废弃
    // @Bean
    // public PaginationInterceptor paginationInterceptor() {
    //     return new PaginationInterceptor();
```

```
    // }
    //-------------------------------------------------
    /*
      未测试
      新的分页插件,一缓和二缓遵循mybatis的规则,
      需要设置 MybatisConfiguration#useDeprecatedExecutor = false
      避免缓存出现问题(该属性会在旧插件移除后一同移除)
     */
    /* @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor() {
        MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
        interceptor.addInnerInterceptor(new
PaginationInnerInterceptor(DbType.MYSQL));
        return interceptor;
    }

    @Bean
    public ConfigurationCustomizer configurationCustomizer() {
        return configuration -> configuration.setUseDeprecatedExecutor(false);
    }*/
    //**************************************************
    /**
     * 注册插件
     *  依赖以下版本+
     *      <dependency>
     *         <groupId>com.baomidou</groupId>
     *         <artifactId>mybatis-plus-boot-starter</artifactId>
     *         <version>3.4.1</version>
     *     </dependency>
     * @return com.baomidou.mybatisplus.extension.plugins.MybatisPlusInterceptor
拦截器
     * @author 金聖聰
     * @email jinshengcong@163.com
     * Modification History:
     * Date         Author        Description        version
     *-------------------------------------------------------*
     * 2021/04/16 15:56     金聖聰      修改原因             1.0
     */
    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor() {
        // 0.创建一个拦截器
        MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();

        // 1. 添加分页插件
        PaginationInnerInterceptor pageInterceptor = new
PaginationInnerInterceptor();
        // 2. 设置请求的页面大于最大页后操作，true调回到首页，false继续请求。默认false
        pageInterceptor.setOverflow(false);
        // 3. 单页分页条数限制，默认无限制
        pageInterceptor.setMaxLimit(500L);
        // 4. 设置数据库类型
        pageInterceptor.setDbType(DbType.MYSQL);
        // 5.添加内部拦截器
        interceptor.addInnerInterceptor(pageInterceptor);

        return interceptor;
    }
}
```

3.测试类测试

```
    @Test
    void selectPage() {
        //分页查询
        /*
         * 1、名字中包含雨并且年龄小于40
         * name like '%雨%' and age<40
         * 条件构造器
         */
        LambdaQueryWrapper<User> lambdaQueryWrapper = Wrappers.lambdaQuery();
        lambdaQueryWrapper.ge(User::getAge,18);

        // 泛型是实体类 ， 当前页数 默认是1，从1开始，不是0。每页最多多少条

        // 第一种用selectPage方法,返回的是 Page<User> userPage
        /*Page<User> page = new Page<>(1, 2);
        Page<User> userPage = userMapper.selectPage(page, lambdaQueryWrapper);

        System.out.println("总页数: "+userPage.getPages());
        System.out.println("总记录数: "+userPage.getTotal());
        userPage.getRecords().forEach(System.out::println);*/

        // 第二种用 selectMapsPage方法,返回的是 IPage<Map<String, Object>>
        // 如果第一个参数还是用的上面的page,此时page胡报错
        // 解决方案:
        //      把page 转成对应的类型IPage<Map<String, Object>>
        //          因为新版后(3.4.1+), 更改了源码 给他设定了具体类型
        IPage<Map<String, Object>> page1 = new Page<>(4, 2);
        IPage<Map<String, Object>> mapIPage = userMapper.selectMapsPage(page1,
lambdaQueryWrapper);

        System.out.println("总页数: "+mapIPage.getPages());
        System.out.println("总记录数: "+mapIPage.getTotal());
        List<Map<String, Object>> records = mapIPage.getRecords();
        records.forEach(System.out::println);
    }
```

多表联查分页

实际开发中有的查询是多表联查的,这样就不能用上面的两个方法了,但是不甘心使用传统的写法,,那怎么办?

用xml自定义查询方法

在接口中创建自定义方法

```
// 自定义分页
IPage<User> selectUserPage(Page<User> page,@Param(Constants.WRAPPER)
Wrapper<User> wrapper);
```

xml中配置

```
<select id="selectUserPage" resultType="com.jsc.mybatisplus.entity.User">
    <!--      参数的注解是固定的 ew 就是WRAPPER的值-->
    <!--     没有什么改变,但是可以自己多表联查-->
    select *
    from user ${ew.customSqlSegment}
  </select>
```

测试

```
    @Test
    void selectPageCustomize() {
        //分页查询
        /*
         * 1、名字中包含雨并且年龄小于40
         * name like '%雨%' and age<40
         * 条件构造器
         */
        LambdaQueryWrapper<User> lambdaQueryWrapper = Wrappers.lambdaQuery();
        lambdaQueryWrapper.ge(User::getAge,18);

        // 泛型是实体类 , 当前页数 默认是1，从1开始，不是0。每页最多多少条

        // 第一种用selectPage方法,返回的是 Page<User> userPage
        Page<User> page = new Page<>(1, 2);
        IPage<User> userIPage = userMapper.selectUserPage(page,
lambdaQueryWrapper);

        System.out.println("总页数: "+userIPage.getPages());
        System.out.println("总记录数: "+userIPage.getTotal());
        userIPage.getRecords().forEach(System.out::println);
    }
```

更新

根据id更新

```
    @Test
    void UpdateById() {
        User user = new User();
        user.setId(1382713714941648898L);
        user.setName("小陈");
        int i = userMapper.updateById(user);
        System.out.println("影响记录数: "+i);
    }
```

以条件构造器作为参数的更新方法

```
    private UpdateWrapper<User> update = Wrappers.update();

    @Test
    void UpdateByWrapper() {
        User user = new User();
        // user.setId(1382713714941648898L);
        user.setName("小金");

        update.eq("name","老金");
        int i = userMapper.update(user,update);
        System.out.println("影响记录数: "+i);
    }

    @Test
    void UpdateByWrapper1() {
        User whereUser = new User();
        // user.setId(1382713714941648898L);
        whereUser.setName("小金");
        // 可以直接把实体传进去
        update = new UpdateWrapper<>(whereUser);
        update.eq("age",18);

        User user = new User();
        user.setAge(21);
        int i = userMapper.update(user,update);
        System.out.println("影响记录数: "+i);
    }
```

条件构造器中set方法的使用

```
    @Test
    void UpdateByWrapper2() {
        // 不创建实体传入,直接在条件中set
        update.eq("name","小金").set("name","老金");

        int i = userMapper.update(null,update);
        System.out.println("影响记录数: "+i);
    }
```

用lambda语法更新

```
    @Test
    void UpdateByLambda() {
        LambdaUpdateWrapper<User> updateLambda = Wrappers.lambdaUpdate();
        updateLambda.eq(User::getName,"老金").set(User::getName,"小金");
        int i = userMapper.update(null,updateLambda);
        System.out.println("影响记录数: "+i);
    }
```

```
    @Test
    void UpdateByLambdaChain() {
        // 链式调用
        boolean update = new LambdaUpdateChainWrapper<User>
(userMapper).eq(User::getName, "小金").set(User::getName, "老金").update();
        System.out.println(update?"成功":"失败");
    }
```

删除

根据id删除

```
    @Test
    void deleteById() {
        int i = userMapper.deleteById(1382699085670719489L);
        System.out.println("影响行数: "+ i);
    }
```

其他普通方法删除

```
    @Test
    void deleteByMap() {
        Map<String, Object> columnMap = new HashMap<>();
        columnMap.put("name","小金");
        columnMap.put("age","18");
        // WHERE name = ? AND age = ?
        int i = userMapper.deleteByMap(columnMap);

        System.out.println("影响行数: "+ i);
    }

    @Test
    void deleteByBatchIds() {
        List<Long> longs = Arrays.asList(1383035889074692097L,
1383035840634646530L);
        int i = userMapper.deleteBatchIds(longs);
        // WHERE id IN ( ? , ? )
        // 根据id批量删除
        System.out.println("影响行数: " + i);
    }
```

以条件构造器为参数的删除方法

```
   @Test
    void deleteByWrapper() {
        LambdaQueryWrapper<User> lambdaQuery = Wrappers.lambdaQuery();
        lambdaQuery.eq(User::getId, "1383037094597267457");
        int delete = userMapper.delete(lambdaQuery);
        System.out.println("影响行数: " + delete);
    }
```

AR模式(ActiveRecord)

AR模式简介

活动记录,是一个领域模型模式,

特点:

一个模型类对应数据库中的一个表

模型类的一个实例对应表中的一个记录

简单来说就是通过实体类对象对表进行增删改查操作,方便开发人员的开发

MP中AR模式的实现

1.首先实体类要继承一个抽象类Model<泛型是这个类本身>

用lombok的`@Data`后,继承别的类后会有个警告,用注解`@EqualsAndHashCode(callSuper=false)`可以消除警告,再添加一个系列化id

```@EqualsAndHashCode(callSuper=false)```就是不调用父类,

但是,别的时候 大部分需要父类的一些属性作为等值比较的

```
@Data
@EqualsAndHashCode(callSuper = false)
public class User extends Model<User> {
    private static final long serialVersionUID = 75899303127780819 95L;
    /** 主键 */
    private Long id;
    /** 姓名 */
    private String name;
    /** 年龄 */
    private Integer age;
    /** 邮件 */
    private String email;
    /** 直属上级id */
    private Long managerId;
    /** 创建时间 */
    private LocalDateTime createTime;
}
```

2.必须存在原始mapper的接口,并继承baseMapper接口

```
    @Test
    void ARInsertTest() {
        User user = new User();
        user.setName("xx");
        user.setAge(0);
        user.setManagerId(1088248166370832385L);
        user.setCreateTime(LocalDateTime.now());

        // 直接insert,自己插自己可还行
```
```

```
        boolean insert = user.insert();
        System.out.println(insert?"成功":"失败");
    }

    @Test
    void ARSelectByIdTest1() {
        // 不需要参数,直接实体上设置
        // 直接查,自己查自己可还行
        User user = new User();
        user.setId(1383044106274050049L);

        User user1 = user.selectById();
        System.out.println(user1);
    }
// 查出来的都是新的对象,并没有把值设置到原来的对象上

    @Test
    void ARUpdateTest() {
        User user = new User();
        user.setId(1383044106274050049L);
        user.setName("xxoo");
        user.setAge(16);
        user.setManagerId(1383035808669888513L);
        user.setCreateTime(LocalDateTime.now());

        // 自己操作
        boolean b = user.updateById();
        System.out.println(b?"success":"fail");

    }

    @Test
    void ARDeleteTest() {
        User user = new User();
        user.setId(1383044106274050049L);

        // 自己操作
        boolean b = user.deleteById();
        System.out.println(b?"success":"fail");

    }
```

主键策略

MP支持的主键策略介绍

0.基于雪花算法(默认的)

1.局部主键策略实现

通过在实体类上id的注解`@TableId(type=IdType.XXX)`设置

值	值
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)

2.全局主键策略实现

在配置文件中设置

```
mybatis-plus.global-config.db-config.id-type=uuid
```

如果局部策略和全局策略都设置了 ,就近原则

MP配置

基本配置

基本的在官网,这里不做过多记录

https://mybatis.plus/config/#%E5%9F%BA%E6%9C%AC%E9%85%8D%E7%BD%AE

进阶配置

https://mybatis.plus/config/#configuration

`mapUnderscoreToCamelCase`这个配置不能和`configLocation`一起出现,会报错

`字段验证策略`可以看一下~

不过一般都是默认的NOT_NULL

not_empty (字段为"")也是忽略

ignored 就是空的也插入 但是有风险 在更新的时候

但是在实体类中,

个别的,可以在字段上配置`@TableField(stragegy=FieldStrategy.NOT_EMPTY)`,就是字段为""的时候也忽略

同样的就近原则

tablePrefix 表名前缀 全局设置

`mybatis-plus.global-config.db-config.table-prefix=`

通用Service

1. 新建service包
2. 新建接口`UserService`
3. 继承`IService<T>`泛型写所对应的实体类
4. 创建service的实现包`impl`
5. 创建`UserServiceImp`
6. 继承`ServiceImpl<UserMapper, User>`第一个蚕食数索要操作的mapper接口,第二个参数是对应的实体类
7. 实现刚才的接口`UserService`,并给上`@Service`的注解

   UserService.java

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

   ServiceImpl.java

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

基本方法

```
    @Autowired
    private UserService userService;

    @Test
    public void testService() {
        User one = userService.getOne(Wrappers.
<User>lambdaQuery().gt(User::getAge,25),false);
        System.out.println(one);
    }
```

批量操作方法

```
    @Test
    public void batchTest() {
        // 批量
        User user = new User();
        user.setName("犀利");
        user.setAge(12);

        User user1 = new User();
        user1.setName("犀利1");
        user1.setAge(121);

        List<User> users = Arrays.asList(user1, user);
        userService.saveBatch(users);
    }
```

链式调用方法

```
    @Test
    public void chainTest() {
        // 查大于25的
        List<User> list = userService.lambdaQuery().gt(User::getAge, 25).list();
        list.forEach(System.out::println);
    }

    @Test
    public void chainTest1() {
        // 更新
        boolean update = userService.lambdaUpdate().eq(User::getName, "小
陈").set(User::getName, "小玉").update();
        System.out.println(update?"success":"fail");
    }

    @Test
    public void chainTest2() {
        // 直接删除
        boolean update = userService.lambdaUpdate().eq(User::getName,
"xx").remove();
        System.out.println(update?"success":"fail");
    }
```