Mybatis-Plus最优化持久层开发

Mybatis-plus：最优化持久层开发

一：Mybatis-plus快速入门：

1.1：简介：

```
Mybatis-plus（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; 提高效率；

自动生成单表的CRUD功能;
提供了丰富的条件拼接方式;
全自动ORM类型持久层框架；（不仅提供数据库操作的方法，还会提供sql语句的实现）
```

1.2Mybatis-plus快速入门：

```
！！！！！！！！！！！！！！！！！！！！！！！！！！！！！
如果我们想要对User表的数据进行单表CRUD：
  我们使用Mybatis-plus之后：我们只需要：
   1.创建mapper接口
   2.继承 BaseMapper<User> （<>:要操作的表/实体类）：我们就会拥有CRUD方法+CURD的sql语句

   注意：
   继承的BaseMapper(它里面有单表的增删改查方法),就不用写mapper.xml文件了,
   之后就可以直接使用mapper对象调用相应的CRUD方法即可进行数据库的操作了！！！
```

```

```

Mapper接口

```
    public interface UserMapper extends BaseMapper<User>{
        /*
        原来Mybatis中：我们需要在mapper中自定义方法，然后在
mapper.xml中使用sql实现方法
        但是使用了Mybatis-plus之后，我们直接继承“BaseMapper<>”：它里
面有单表的增删改查各种方法以及实现
        我们继承它以后就拥有了这些方法，就不用写mapper.xml文件了;

         */
    }
```

SpringBoot测试类

```
    @SpringBootTest  //这个注解：就会自动完成ioc容器初始化，我们想要谁
直接拿即可！！！！！！！！！！！！！！！！
    public class SpringBootMybatisPlusTest {

        @Autowired
        private UserMapper userMapper; //拿到对象！！！

        public void test(){
            List<User> users = userMapper.selectList(null); //不传参数：直接
写null：代表查询全部数据;
                                直接调用baseMapper接口中相应的方法即可
        }
}
```

```

二：Mybatis-plus的核心功能
--------------------
```

Mybatis-plus是如何增强的

```
Mybatis-plus可以对三层架构的两层进行增强：
1.MapperC层：只要继承，就拥有了crud方法
2.Service层：继承
```

```
原理：！！！！！！！！！
mapper接口只要继承BaseMapper<实体类> 接口：
  接下来我们就能使用通过mapper对象和BaseMapper接口中提供的CRUD方法来对  实体类表 进行操作;
```

2.1基于Mapper接口的CRUD

```
```

（1）mapper接口

```
        如果我们想要对User表的数据进行单表CRUD：
          我们使用Mybatis-plus之后：我们只需要：
           1.创建mapper接口
           2.继承 BaseMapper<User> （要操作的表/实体类）：我们就会拥有CRUD方法+CURD的sql语句
         */
        public interface Use rMapper extends BaseMapper<User> {

            //继承之后：下面就相当于有了方法
        }
```

（2）测试类：

```
    @SpringBootTest
public class MybatisPlusTest {
```

```
@Autowired
private UserMapper userMapper;

//（1）insert：
@Test
public void test(){
   User user=new User();
   user.setName(”kun”);
   user.setAge(88);
   user.setEmail(”xxx”);
   //baseMapper提供的数据库insert方法
    int row = userMapper.insert(user);
}

//(2)delete:
@Test
public void test_delete(){
    //1.根据id删除：（）内装值
    int rows = userMapper.deleteById(1687124323556002889L);

    //2.根据id删除：age=20  name=jack; ()内装条件：  -->拼接条件
    Map param=new HashMap();
    param.put(”age”,20);     //-->删除age=20 + name=jack
    param.put(”name”,”jack”);
    int i = userMapper.deleteByMap(param);
    System.out.println(”i = ” + i);

    //wrapper 条件封装对象，无限封装条件;
    //userMapper.delete(wrapper)
}


//(3)Update()
@Test
public void test_update(){
    //1.将主键userId为1的age改为30:（)内装值
      //update user set age=30 where id=1:就等同于下方
    User user=new User();
    user.setId(1L);
    user.setAge(30);
    int i = userMapper.updateById(user);

    //2.将所有人age改为20
    User user1=new User();
    user1.setAge(20); //
    int update = userMapper.update(user1, null);// null:代表没条件，该所有
    System.out.println(”update = ” + update);
```

```
        /*
        TODO:update：当属性为null时：代表不修改
            updateById():实体类的id必须有值
            update() :实体类可以没有id值
         */
    }



    //(4)select a:根据主键查询  b：根据主键集合查询
    public void test_select(){
        //a:根据id查询
        User user = userMapper.selectById(1);
        //b：根据集合查询：eg：根据ids查询
        List<Long> ids=new ArrayList<>();
        ids.add(1L);
        ids.add(2L);

        List<User> users = userMapper.selectBatchIds(ids);
//selectBatchIds：批量ids
        System.out.println("users = " + users);
    }

}
```

2.2就Service接口的CRUD

```
service接口继承：
```

2.3分页查询实现:

```
Mybatis-plus实现分页查询：！！！！
    使用步骤：
    1.导入分页插件
    2.使用分页查询
```

```

```
-04

@SpringBootApplication
@MapperScan("com.atguigu.mapper")
/*
启动类也是配置类：所以我们可以声明带@Bean的方法
1.配置一个分页的插件：Mybatis-plus提供的，我们直接加进来使用就行了;
  在配置类/启动类：中使用@Bean返回
 */

public class Main {
    public static void main(String[] args) {

        SpringApplication.run(Main.class, args);
    }
        @Bean
        //1.将Mybatis-plus插件加到ioc容器
        public MybatisPlusInterceptor plusInterceptor(){
            //a:是所有Mybatis-plus的插件集合:我们想要使用任何插件都可以加到这个集合中去;eg:我们可以将分页插件加到里面;
            MybatisPlusInterceptor mybatisPlusInterceptor = new MybatisPlusInterceptor();
            //b:加入分页插件：
            mybatisPlusInterceptor.addInnerInterceptor(new PaginationInnerInterceptor(DbType.MYSQL));
                                                                                    //指定数据库
            return mybatisPlusInterceptor;
        }
}

/*
测试类：
2.之后就可以使用分页插件了：
 */
@SpringBootTest
public class MybatisPlusTest {
    @Autowired
    private UserMapper userMapper; //注入UserMapper对象：进行数据库操作

    @Test
```
```

```
    public void testPage(){

        //a：创建Page对象: 参数解释：(参数1：页码，  参数2：页容量)
        Page<User> page=new Page<>(1,3);

        userMapper.selectPage(page,null); //需要放入Page对象

        //最后分页的结果也会被封装到page里面:-->所以直接通过page对象获取结果即可
        long pages = page.getPages();
        long current = page.getCurrent(); //获取当前页码、
        List<User> records = page.getRecords(); //获取当前页数据
        long total = page.getTotal();//获取总条数

    }

}
```

自定义mapper方法使用分页

```
上面：我们是使用Mybatis提供的方法：  userMapper.selectPage(page,null);
如果我们自定义的mapper方法想要使用分页查询：
   1.在mapper接口中定义方法：
       自定义方法想要分页：在第一个参数加一个IPage对象，返回值也是IPage
   2.在mapper文件中实现：使用sql语句实现：不要加limit和;
   3.测试
```

```
1.

public interface UserMapper extends BaseMapper<User> {
    //继承BaseMapper接口:里面有单表的方法

    /*
    演示：自定义mapper方法实现分页：
      eg:查询

     */
    //1.在mapper接口中定义抽象方法：但是：方法的第一个参数：要传一个
```

IPage，返回值要是IPage，里面写查询集合的泛型即可
  IPage<User> queryByAge(IPage<User>page, @Param(”age”) Integer age);
}

2.
  <!--2.使用sql语句实现UserMapper接口的方法：不用在最后加limit，！-->
  <select id=”queryByAge” resultType=”com.atguigu.pojo.User”> <!--resultType:泛型-->
    select * from user where age>#{age}
  </select>

 3.

    /*
   3.测试自定义方法分页
    */
  public void test_MyPage(){
    //a:传一个分页插件的值：
    Page<User> page=new Page<>(1,3);
    //b.
    userMapper.queryByAge(page,1);
    //c:
    //最后分页的结果也会被封装到page里面:-->所以直接通过page对象获取结果即可
    long pages = page.getPages();
    long current = page.getCurrent(); //获取当前页码、
    List<User> records = page.getRecords(); //获取当前页数据
    long total = page.getTotal();//获取总条数
    System.out.println(”total:”+total);

  }

}

```

### 2.4:条件构造器使用：

#### 2.4.1条件构造器的作用：

```
warpper对象：动态进行 条件的拼接；
就相当于在sql语句后面加条件，只是使用java代码的格式;

---->:就不用在mapper.xml文件中使用sql语句实现了,直接使用java代码来代替sql语句；！！！！！！！！！

```

#### 2.4.2条件构造器的类结构：

```
一般使用：
   第一种： UpdateWrapper（修改）、QueryWrapper（删除、查询、修改）
   第二种： LambdaUpdateWrapper（修改）、 LambdaQueryWrapper（删除、查询、修改）
```

#### （1）第一种

##### 基于queryWrapper组装条件：增/删/改

```
使用方法：
   1.在mapper接口中定义方法，并继承base接口（提供了各种方法的实现）
   2.直接使用java代码实现：不需要在mapper.xml文件中实现了
   //a:创建QueryWrapper
    QueryWrapper<实体类>queryWrapper=new QueryWrapper<>();
    //b：条件拼接：调用queryWrapper方法
    //c：使用mappper对象：调用CRUD方法，传入对象即可
```

```
1.mapper接口：

public interface UserMapper extends BaseMapper<User> {
    //继承BaseMapper接口:里面有单表的方法

    /*
    演示：自定义mapper方法实现分页：
      eg:查询
     */
```
```

```
    //1.在mapper接口中定义抽象方法：但是：方法的第一个参数：要传一个IPage，返回值要是IPage，里面写查询集合的泛型即可
    IPage<User> queryByAge(IPage<User>page, @Param("age") Integer age);
}
```

2.

```
/*
演示条件构造器使用：进行条件的动态拼接：

（1）QueryWrapper测试：
 */
@SpringBootTest
public class MybatisPlusQueryWrapper {

    @Autowired
    private UserMapper userMapper; //拿到对象：操作数据库

    @Test
    public void test_01(){
        //查询用户名包含 a，age在20到30之间的，且邮箱不为空的用户;
         //分析：查询：返回的是集合List-->selectList

        //a：创建一个QueryWrapper集合：相当于容器来装条件
        QueryWrapper<User> queryWrapper=new QueryWrapper<>();
        //b：之后sql语句的条件：直接使用queryWrapper动态调用wrapper的方法来完成动态拼接;
        queryWrapper.like("name","a"); //条件1：name种包含a的
        queryWrapper.between("age",20,30); //条件2：age在20-30之间
        queryWrapper.isNotNull("email"); //第三个条件：email不为null
        List<User> users = userMapper.selectList(queryWrapper); ----->:因为继承了base接口，所以单表直接调用方法即可！
        //queryWrapper就是sql条件，将它·传进去！！！
        //之后这个就会变成sql语句；！！！！！！！！！！！
        //就相当于：select  * from user where name(like %a% and age>=20 and age<=30 and email is not null)

        /*
        !!!一个个调用麻烦，可以使用链式调用：！！！！
queryWrapper.like("name","a").between("age",20,30).isNotNull("email");
        */

    }
}
```

```

（2）queryWrapper实战使用：

```
@SpringBootTest
public class MybatisPlusQueryWrapper {

    @Autowired
    private UserMapper userMapper; //拿到对象：操作数据库按照年龄降序查询用户，如果年龄age相同则按id升序排列

     //(2)queryWrapper:按年龄降序查询用户，如果年龄相同则按id升序排列
    @Test
    public void test_02(){

        //a:创建QueryMapper集合：装条件
        QueryWrapper<User> queryWrapper=new QueryWrapper<>();
        //b:使用条件构造器进行条件的拼接：
        queryWrapper.orderByDesc(”age”).orderByAsc(”id”);
                            //a:条件1：age降序排列; b:条件2：id增序
        //c：放到queryWrapper中，进行条件拼接; ==order by age desc,id asc
        List<User> users = userMapper.selectList(queryWrapper);
    }

    //(3)删除email为null的用户
    @Test
    public void test_03(){
        //a:
        QueryWrapper<User> queryWrapper=new QueryWrapper<>();
        //b:
        queryWrapper.isNotNull(”email”);
        //c:
        int delete = userMapper.delete(queryWrapper);
    }

    //(4):将age>20且用户名中包含a或email为null的用户信息修改：
      //分析：修改：--update：所以usermapper调用update方法
            ：相应的条件：age>20... 使用queryWrapper封装，然后作为条件放入update方法形参中;!!!!!
    @Test
    public void test_04(){
        //a:
        QueryWrapper<User> queryWrapper=new QueryWrapper<>();
```
```

```
        //b：添加条件：
        queryWrapper.gt("age",20).like("name","a")                  ---------
->B:相当于：where .....
                .or().isNotNull("email");
          //1.条件：age>20 name包含a，--->:条件直接调用时：默认自动使
用and进行拼接;！！！！！
             //queryWrapper.gt("age",20).like("name","a") == where
age>20 and ..
          //2.如果想要在条件之后追加or关键字：可以使用：“or().条件”；

        //c：创建User对象，进行update修改
        User user=new User();
        user.setAge(88);                  ------>A:相当于update user  set
name="" age=
        user.setEmail("hehe");
        //d:如果要修改：第一位传入要修改的对象
         userMapper.update(user, queryWrapper); //放入对象 +
queryWrapper
    }

    //(5)：查询用户信息的name和age字段；--->指定列查询！！！
      //select name,age from t_user where id>1;

    public void test_05() {

        //a:
        QueryWrapper<User> queryWrapper = new QueryWrapper<>();
        //b:条件拼接：
        queryWrapper.gt("age",1);
        //!!!查询时：默认是查询全部列！！！！！，但可以使用select指定
        queryWrapper.select("name","age");

        userMapper.selectList(queryWrapper);
    }

    //(6)前端传入两个参数,name，age:---->:动态条件组织判断！！！！
    // 只要name不为null，就作为条件 来查询符合的
    //只要age>18作为条件，就作为条件 来查询符合的
    @Test
    public void test_06(){

        String name="xx";
        Integer age=19;


        //a:
        QueryWrapper<User>queryWrapper=new QueryWrapper<>();
        //b:进行条件拼接：
```

```
        //b.1：如果name！=null,就使用queryWrapper进行条件拼接
        if(StringUtils.isNotBlank(name)){  //调用isNotBlank进行判断
            //动态条件判断：如果name不为null，就进行条件的拼接
            queryWrapper.eq("name",name); //使用queryWrapper进行拼接
        }

        //b.2:如果age>18 进行动态拼接：
        if (age>18 && age!=null){
            queryWrapper.eq("age",age);

        }

        userMapper.selectList(queryWrapper);


        /*
    注意：!!!!!!!!!!!!!!!
        Wrapper每个方法都有：一个boolean类型的condition，允许我们第一个参数放一个比较表达式，
        ：如果condition为true-->整个条件为true，就执行下面代码
        ：如果condition为false-->整个条件为false，就失效，不执行下面代码
        -->condition里面有动态判断，就不用我们在外部写判断了
        eg：
        */

        queryWrapper.eq(StringUtils.isNotBlank(name),"name",name);
        queryWrapper.eq(age>18 && age!=null,"age",age);
    }


```

updateWrapper：改

```
进行修改时：使用updateWrapper：！！！！

```

```
        //a:
        UpdateWrapper<User> queryWrapper=new UpdateWrapper<>();
        //b:
        queryWrapper.gt("age",20).like("name","a").isNotNull("email")
```

```
                .set(”email”,null).set(”age”,100);
        //c：
      userMapper.update(null,queryWrapper);
```

```
##### 在修改时updateWrapper和queryWrapper区别：
```

```
（参数1：条件； 参数2：修改的数据）
queryWrapper修改：参数1只能放条件，且列名不能为null
updateWrapper修改：参数1可以放条件， 且列名可以设置为null;
                且可以直接调用set方法进行修改
```

```
#### （2）第二种：
```

```
和第一种用法一样，只是对第一种的增强
 eg:
  Lambda：避免列写错的机制：传入的列名可以是”方法引用的表达式“：类名::getName -->相当于找方法对应的列名
```

```
##### LambdaQueryWrapper:增/删/改
```

```
eg:
查询用户名包含 a like，年龄在20-30之间，并且邮箱不为null的用户信息;
  LambdaQueryWrapper<User>lambdaQueryWrapper=new LambdaQueryWrapper<>();
        //即只要将原来的列名，变为Lambda方法引用即可，其余不变！！！！！！！！！！
lambdaQueryWrapper.like(User::getName,”a”).between(User::getAge,20,30).isNotNull(User::getEmail);

      userMapper.selectList(queryWrapper);
```

```
```

LambdaupdateWrapper：改

```
eg：
        LambdaUpdateWrapper<User>lambdaUpdateWrapper=new
LambdaUpdateWrapper<>();

        //即只要将原来的列名，变为Lambda方法引用即可，其余不变
！！！！！！！！！！
lambdaUpdateWrapper.gt(User::getAge,20).like(User::getName,”a”).isNot
Null(User::getEmail)
                .set(User::getEmail,null).set(User::getAge,100);

        userMapper.update(null,lambdaUpdateWrapper);
```

(3)总结：

```
Wrapper：里面：完成了动态条件封装
最终推荐使用Lambda;（不用写列名，只需类名：：方法即可）

如果复杂的sql：qg:嵌套、子查询 不会使用Wrapper怎么办：
 我们仍然使用原来的：定义mapper
      如果为单表的话：直接使用mapper对象调用CRUD方法即可;
      如果为多表的话：在mapper.xml文件中实现;
```

2.5Mybatis-plus核心注解的使用：

```
注解的作用：指定实体类和表之间的映射;
```

1.@TableName

```
作用：指定数据库表名;(当【BaseMapper<实体类>】 实体类名 和 数据库表名不同时);
使用：@TableName(”数据库表名”)

a：位置：加到实体类上
b：可以不加，不加的话默认使用实体类的名字作为 表名！ 忽略大小写;
   eg:BaseMapper<User>:默认将实体类名：User，作为表名 来进行操作;

c:使用@TableName注解场景：
   当数据库的表名和实体类名称不同时（忽略大小写），需要使用@TableName来指定 表名：
    eg：数据库名：t_user; 实体类名：User
        两个名称不同，此时我们需要在实体类上方使用：”@TableName(”t_user”)“ 来指定表名;
d: 如果表名命名规范：eg：都以 ”t_”开头，且除了”t_“以外，实体类名=表名
   我们可以将前缀统一提取到配置文件中(application.yml)，这样就不需要每个实体类都加@TableName注解来指定表名了
   它会自动查找相应的表;
    mybatis-plus:
       global-config:
            db-config:
              table-prefix: t_ #表名前缀;
                            #加了这个之后，我们就不需要在每个实体类加@TableName注解了，它就会自己根据前缀”t_”进行查找;
                            #前提：数据库的表都是以 “t_”开头的


```

2.@TableId:

```
作用：描述主键的一些特性;
   eg: value:指定表主键名; type:设置主键type策略
使用：在实体类的主键属性上：+@TableId(value=”主键列名”,type=”主键策略”);

b：位置：加到实体类的主键属性上
c：使用@TableId的场景：
      场景1：
         表 主键的列名 和 实体类的属性名 不一致时,使用这个注解来指定表的列名;!!!
               eg: 1:如果 实体类属性名：id; 数据库主键列名：x_id; --->两个名称不一样，我们可以在实体类主键的属性上方加
```

@TableId(value=”x_id”)
场景2：
设置表主键增长：
type主键策略：

```

type增长策略：

```

@SpringBootTest

public class MybatisPlusTableIdTest {

    @Autowired
    private UserMapper userMapper;

    @Test
    public void test_01(){
        //以插入数据举例：
        User user=new User();
        user.setName(”坤坤”);
        user.setAge(20);
        user.setEmail(”xxx@qq.com”);
        //插入的时候：主键id不要赋值; 主键自增长 或者 type策略 进行赋值
！！！！！！！！！！！！
        /*

        1.默认主键的type（策略）是：雪花算法;--->:它会生成一个不重复的long类型的数字; eg：123456789
            雪花算法：
              a：要求数据库主键是 bigint / varchar(64) 类型;   -->:eg:varchar(64)/bigint money;!!!
              b：要求实体类相应属性：使用Long/String接值（int接不到） -->:eg:Long/String money;!!!
              c：随机生成一个数字，给予主键值（不重复）
        2.但我们想要把主键type（策略）变成auto自增长,
          方式1：直接在实体类的主键上方加：@TableId(type= IdType.AUTO);前提是mysql数据库必须设置了主键自增长(auto_increment)
            auto：
              a：要求mysql数据库的 表主键 必须设置了 自增长
              b：插入数据就是自增长了
            方式2：全局设置主键自增长策略：如果每个表都设置为自增长

（auto）,在配置文件(application.yml)中配置

*/

userMapper.insert(user); //之间调用BaseMapper中的insert方法;

}
}

User视图类：
public class User(){
@TableId(type = IdType.AUTO)
private Long id;
}

```

#### 3.@TableFiled

```
作用：描述非主键字段
使用：在实体类的非主键字段上：+@TableFiled(value="",exist="")

a：使用场景1：value="" （不是必须指定的，String类型）
:当实体类的属性名 和 列名 不同的时候 使用@TableFiled指定数据库表名;
使用场景2：esist=ture/false； （默认为true）
：eg：@TableFiled(value="name",exist=false):
private String name;
-->:false代表：认为name没有数据库对应的列，因此在查询等时候不会对他进行操作;

```

三：Mybatis-plus高级扩展
---

### 3.1逻辑删除实现：

```
(1)逻辑删除概念：
逻辑删除：是 "假删除数据"：是将想要删除数据的状态修改为 "被删除状态"，但是数据并没有消失，之后在查询时，查询没有 删除状态 的数据，也是一种删除;
```
```

物理删除：delete from：删除之后就没有了相应的数据;

(2)逻辑删除的实现：
方式1：单个表逻辑删除的实现：

a：在数据库表中 插入一个逻辑删除字段(int类型，有1/0状态字段！)：
alert table User add deleted int default 1/0; -->: 1/0:逻辑删除的状态字段：1代表：逻辑删除; 0代表：未逻辑删除;!!!!

b：在实体类的对应属性上加：@TableLogic
eg：
@TableLogic //代表这个属性对应的列是 ： 逻辑删除状态字段
//当我们删除数据时候，自动变成修改此列的属性值（即将deleted变为1）； 默认：0是未删除，1是删除
//当我们查询数据时候：默认只查询 deleted=0 的列的数据
private Integer deleted;

方式2：多个表逻辑删除的实现（全局指定）：

a：同上

b：在配置文件(application.yml)中：配置

```
mybatis-plus:
  global-config:
    db-config:
      logic-delete-field: deleted # 全局逻辑删除的实体属性名
```

注意：这种就不需要在类的属性名上 + @TableLogic了
但是仍然需要在实体类中创建一个和列名deleted 对应的属性名;

```
```

代码演示：

```
```

以单个表实现逻辑删除为例：

```
@TableLogic
```

```
 private Integer deleted;


@SpringBootTest
public class MybatisPlusTableLogicTest {

    @Autowired
    private UserMapper userMapper;

    public void test_01(){

        //a:之前是物理删除：delete from user where id=5;

        //b:现在：逻辑删除：相当于：update user set deleted=1 where id=5
        userMapper.deleteById(5);

        //eg:
        userMapper.selectList(null); -->之后在查询时：就只会查询deleted=0的了：！！！！！ （自动在条件位置加上where deleted=0）

    }


```

3.2乐观锁的实现

```
1.乐观锁 和悲观锁 都是在 并发编程 中用于处理并发访问和资源竞争的两种不同机制;

2.(1)悲观锁:
    同步锁/互斥锁：线程在访问共享资源之前会获取到锁，在数据访问时只允许一个线程访问资源，只有当前线程访问完成只后，才会释放锁，让其它线程继续操作资源;

 (2)乐观锁：不需要提前加锁，而是在数据更新阶段进行检测

   乐观锁和悲观锁是两种解决并发数据问题的思路，不是技术！！

3.具体的技术和方案:
      1.乐观锁实现方案和技术
        版本号/时间戳
        CAS
        无锁数据结构
```

2.悲观锁的实现方案和技术
锁机制
数据库锁
信号量

4. 现在主要学习如何使用乐观锁(版本号)解决并发问题！！！！！!!
(1)版本号：解决数据并发更新出现错误数据的问题;--->保证数据一致性;

(2)乐观锁技术的实现流程：
a：每条数据添加一个版本号字段version
b：取出记录时，获取当前version
c：更新时，检查获取版本号是不是数据库当前最新版本号
d：如果是(代表没人修改过数据)，执行更新：set数据更新，version=version+1
e：如果不是（证明已经修改过了），更新失败

5.使用mybatis-plus数据使用乐观锁
步骤：
1.数据库插入一个version字段：alert table user add version int default 1 （int类型 乐观锁字段）
2 实体类创建相应的属性： private Integer version; + 在属性上方使用@Version注解
3.添加拦截器（添加版本号更新插件）
之后在更新的时候就会检查版本，检查数据错误的问题;

```

乐观锁测试：

```
1.
alter table user add version int default 1; #int 类型 乐观锁字段

2.
public class User {
@Version
private Integer version; //版本号字段
}

//注意：之后在数据修改时都会检查乐观锁;

3.在配置类/Main类中加入拦截器

public class Main {

```
    public static void main(String[] args) {

        SpringApplication.run(Main.class, args);
    }
        @Bean

        public MybatisPlusInterceptor plusInterceptor(){

            //拦截器2：乐观锁【版本号插件】 mybatis-plus在每次更新时候
，每次帮我们对比版本号字段和增加版本号+1
                mybatisPlusInterceptor.addInnerInterceptor(new
OptimisticLockerInnerInterceptor());

            return mybatisPlusInterceptor;
        }


4.
/*
测试乐观锁实现
 */
@SpringBootTest
public class MybatisPlusVersionTest {

    private UserMapper userMapper;

    @Test
    public void testById(){

        //步骤1：先查询，再更新 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=new User(); //user 版本号为：1
        User user1=new User(); //user1 版本号为：1

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

        userMapper.updateById(user); //20-->:此时数据库的version ->2
        //乐观锁生效：失败
        userMapper.updateById(user1);
        //因为user1的版本号为1，但是数据库的版本号为2，证明不是最新数据
-->1!=2,乐观锁生效，更新失败
          //所以最终age为：20

    }
}
```

```

### 3.3防止全表更新和删除

```
在开发中一般不会出现全表的更新/删除：
  Mybatis-plus提供一种防御插件：一旦检测到是全表的update/delete：就会报异常；

实现方法：添加防止全表更新和删除的拦截器：

```

代码测试：

```
1.加入拦截器/插件：
public class Main {
    public static void main(String[] args) {
        SpringApplication.run(Main.class, args);
    }
        @Bean
        public MybatisPlusInterceptor plusInterceptor(){

          //拦截器3：防止全表删除和更新的拦截器
            mybatisPlusInterceptor.addInnerInterceptor(new BlockAttackInnerInterceptor());

            return mybatisPlusInterceptor;
        }

 2.测试类中：
     public void test(){

     @Au..
     private UserMapper userMapper; //拿到mapper对象，进行数据库操作！！！；

     //a：全表删除！
     mapper.delete(null); ----->：null,代表没有写条件，代表：全表删除;
     //b：全表更新：
```
```

mapper.update(null); ------>: null,代表没有条件，代表全表更新！

----------->:之后就会抛出异常：不允许全表 delete/update！！！！！！！

}

```

四：Mybatis-Plus代码生成器(MybatisX插件)!!!!!!!!!!!!!
----------

```

学习目标：

1.如何使用MybatisX插件： 逆向工程 生成Mybatis-Plus代码；
//逆向工程：根据表自动生成实体类、mapper、service;!!!!!!!!!!!!!!!!!!!!!

2. 使用MybatisX插件 动态生成自定义的sql语句

```

### 4.1Mybatis插件逆向工程

```
//逆向工程：根据表自动生成实体类、mapper、service;!!!!!!!!!!!!!!!!!!!!!
```
```

步骤：
    连接数据库;
    .....

```

### 4.2Mybatis-plus快速生成单表CRUD代码（自定义sql）！！！！！！！！！

```
如果我们需要自己定义一些sql语句：
  eg:批量查询、根据id查询.....

//步骤：
      1.直接在mapper接口中书写相应的 ”方法名“ （//按照mybatisX的方法命名规则）
        eg：批量插入：insertBatch
            根据条件查询：selectByNameAndAgeAndAge

      2.然后鼠标放在方法名上，点击MybatisX,
        --->MybatisX 就会在mapper接口自动写好 且 在mapper.xml文件中自动生成sql语句！！！！！！！！

//之后有关单表的CRUD我们就不需要自己写了！！！！！！！！！！

//我们可以看mybatisX的官方文档：介绍了如何写

```
 原文链接: https://juejin.cn/post/7357141293111967784
```