mybatis-plus的使用 （进阶）

mybatis-plus的使用 （进阶）

前言：

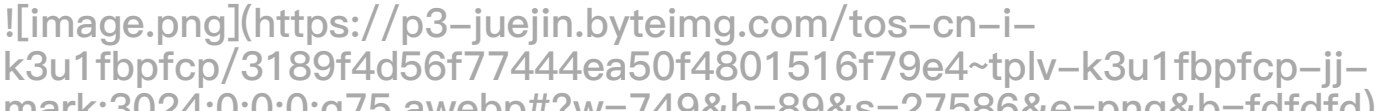
关于mybatis-plus的简介以及基本使用，在《mybatis-plus的使用 （入门）》一文中已做介绍，此处不再赘述。本文主要对mybatis-plus的AR模式、插件、逆向工程、自定义全局操作、公共字段自动填充等知识点进行讲解。

一、ActiveRecord:

Active Record(活动记录)，是一种领域模型模式，特点是一个模型类对应关系型数据库中的一个表，而模型类的一个实例对应表中的一行记录。
ActiveRecord 一直广受动态语言（ PHP 、 Ruby 等）的喜爱，而 Java 作为准静态语言，对于 ActiveRecord 往往只能感叹其优雅，所以 MP 也在 AR 道路上进行了一定的探索，仅仅需要让实体类继承 Model 类且实现主键指定方法，即可开启 AR 之旅。接下来看具体代码:

1、entity:

```
@Data
public class User extends Model<User> {
    private Integer id;
    private String name;
    private Integer age;
    private Integer gender;
    //重写这个方法，return当前类的主键
    @Override
    protected Serializable pkVal() {
        return id;
    }
}
```

```

注：实体类继承Model类，重写pkVal方法。

**2、mapper:**

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

注：虽然AR模式用不到该接口，但是一定要定义，否则使用AR时会报空指针异常。

**3、使用AR:**

*(1)、AR插入操作：*

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration({"classpath:spring/spring-dao.xml"})
public class TestAR {
    @Test
    public void testArInsert(){
        User user = new User();
        user.setName("林青霞");
        user.setAge(22);
        user.setGender(1);
        boolean result = user.insert();
        System.out.println(result);
    }
}
```


```

注：可以看到我们并不需要注入mapper接口，不过正如刚才所说，不使用但还是要定义，否则会报错。AR操作是通过对象本身调用相关方法，比如要insert一个user，那就用这个user调用insert方法即可。返回值为布尔类型，由上图可看到返回了true，是操作成功的。

(2)、AR更新操作：

```
    @Test
    public void testArUpdate(){
        User user = new User();
        user.setId(1);
        user.setName(”刘亦菲”);
        boolean result = user.updateById();
        System.out.println(result);
    }
```

注：user调用updateById方法，将id为1的用户进行更新。

(3)、AR查询操作：

```
    @Test
    public void testArSelect(){
        User user = new User();
        //1、根据id查询
        //user = user.selectById(1);
        //或者这样用
        //user.setId(1);
        //user = user.selectById();

        //2、查询所有
        //List<User> users = user.selectAll();

        //3、根据条件查询
        //List<User> users = user.selectList(new
EntityWrapper<User>().like(”name”,”刘”));

        //4、查询符合条件的总数
        int result = user.selectCount(new
```

```
EntityWrapper<User>().eq("gender",1));
        System.out.println(result);
    }

```

注：上面的代码涉及到了四个不同的查询操作，其实用法与MP的BaseMapper提供的方法的用法差不多，只不过这里是实体对象调用。

(4)、AR删除操作：

```
@Test
    public void testArDelete(){
        User user = new User();
        //删除数据库中不存在的数据也是返回true
        //1、根据id删除数据
        //boolean result = user.deleteById(1);
        //或者这样写
        //user.setId(1);
        //boolean result = user.deleteById();

        //2、根据条件删除
        boolean result = user.delete(new
EntityWrapper<User>().like("name","玲"));
        System.out.println(result);
    }

```

注：这里介绍了两个删除方法，代码中已有注释说明。需要注意的是，删除数据库中不存在的数据，结果也是true。

(5)、AR分页操作：

```
@Test
    public void testArPage(){
       User user = new User();
       Page<User> page =
               user.selectPage(new Page<>(1,4),
               new EntityWrapper<User>().like("name","刘"));
```

```
        List<User> users = page.getRecords();
        System.out.println(users);
    }
```

```
注：这个分页方法和BaseMapper提供的分页一样都是内存分页，并非物理分页，因为sql语句中没用limit，和BaseMapper的selectPage方法一样，配置了分页插件后就可以实现真正的物理分页。AR的分页方法与BaseMapper提供的分页方法不同的是，BaseMapper的selectPage方法返回值是查询到的记录的list集合，而AR的selectPage方法返回的是page对象，该page对象封装了查询到的信息，可以通过getRecords方法获取信息。


**二、插件的配置：**
------------


MP提供了很多好用的插件，而且配置简单，使用方便。接下来一起看看MP的插件如何使用。


### **1、分页插件：**


之前就有说到，BaseMapper的selectPage方法和AR提供的selectPage方法都不是物理分页，需要配置分页插件后才是物理分页，那么现在就来看看如何配置这个插件。



```
<!-- 3、配置mybatisplus的sqlSessionFactory -->
    <bean id="sqlSessionFactory" class=
"com.baomidou.mybatisplus.spring.MybatisSqlSessionFactoryBean">
   <property name="dataSource" ref="dataSource" />
        <property name="configLocation" value="classpath:mybatis-
config.xml"/>
        <property name="typeAliasesPackage"
value="com.ty.mybatisplus.entity"/>
        <!-- 注入全局配置 -->
        <property name="globalConfig" ref="globalConfiguration"/>
        <!-- 配置插件 -->
        <property name="plugins">
            <list>
                <!-- 分页插件 -->
                <bean
class="com.baomidou.mybatisplus.plugins.PaginationInterceptor"/>
```
```

```
            </list>
        </property>
    </bean>
```

注：在sqlSessionFactory这个bean中，通过配置插件，接下来的所有插件都配置在这个list中。

```
@Test
    public void testPage() {
        //配置了分页插件后，还是和以前一样的使用selectpage方法，
        //但是现在就是真正的物理分页了，sql语句中有limit了
        Page<Employee> page = new Page<>(1, 2);
        List<Employee> employeeList =
                emplopyeeDao.selectPage(page, null);
        System.out.println(employeeList);
        System.out.println("================= 相关的分页信息 ==================");
        System.out.println("总条数:" + page.getTotal());
        System.out.println("当前页码:" + page.getCurrent());
        System.out.println("总页数:" + page.getPages());
        System.out.println("每页显示条数:" + page.getSize());
        System.out.println("是否有上一页:" + page.hasPrevious());
        System.out.println("是否有下一页:" + page.hasNext());
        //还可以将查询到的结果set进page对象中
        page.setRecords(employeeList);
    }
```


由图可知，sql语句中已经有了limit，是物理分页了。

也可以通过page调用相关方法获取到相关的分页信息，而且还可以把查询到的结果set回page对象中，方便前端使用。

2、性能分析插件：

在plugin的list中添加如下bean即可开启性能分析插件：

```
 <!-- 输出每条SQL语句及其执行时间，生产环境不建议使用该插件 -->
 <bean
class="com.baomidou.mybatisplus.plugins.PerformanceInterceptor">
        <property name="format" value="true"/><!-- 格式化SQL语句 -->
        <property name="maxTime" value="1000"/><!-- sql执行时间超过
value值就会停止执行，
                                                单位是毫秒 -->
 </bean>
```

注：这个性能分析插件配置了两个属性，第一个是格式化sql语句，设置为true后，sql语句格式就像上面的截图中的一样；第二个属性是sql语句执行的最大时间，超过value值就会报错，这里表示超过1000毫秒就会停止执行sql语句。

3、执行分析插件：

```
<!-- 如果是对全表的删除或更新操作，就会终止该操作 -->
<bean class="com.baomidou.mybatisplus.plugins.SqlExplainInterceptor">
        <property name="stopProceed" value="true"/>
</bean>
```

注：这个插件配置了一个属性，stopProceed设置为true后，如果执行的是删除表中全部内容，那就会抛出异常，终止该操作。该插件主要是防止手抖误删数据。

```
@Test
public void testSqlExplain(){
        //条件为null，就是删除全表，执行分析插件会终止该操作
    emplopyeeDao.delete(null);
}
```

运行该junit测试，可以看到报如下错误，说明该插件生效了。


三、MP的逆向工程：

MyBatis 的代码生成器基于xml文件进行生成，可生成: 实体类、Mapper 接口、Mapper 映射文件。
MP 的代码生成器基于Java代码进行生成，可生成: 实体类(可以选择是否支持AR)、Mapper 接口、Mapper 映射文件、 Service 层、Controller 层。

1、添加依赖：

```
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.37</version>
        </dependency>
        <!-- mp 依赖 -->
        <dependency>
            <groupId>com.baomidou</groupId>
            <artifactId>mybatis-plus</artifactId>
            <version>2.3</version>
        </dependency>
        <!-- mybatisplus逆向工程需要模板引擎，用freemaker也行 -->
        <dependency>
            <groupId>org.apache.velocity</groupId>
            <artifactId>velocity-engine-core</artifactId>
            <version>2.0</version>
        </dependency>
```

注：上面是必须的三个依赖，为了可以在控制台直观的看到生成情况，可以添加日志包(slf4j-api和slf4j-log4j2)，为了让生成的代码不会报错，还可以根据情况添加spring相关的依赖、lombok依赖等。

2、生成器示例代码：

```
/**
 * @author: ty
 * @date: 2023/8/20 11:17
 * mybatis-plus逆向工程示例代码
 */
public class test {
    @Test
    public void testGenerator(){
        //1、全局配置
        GlobalConfig config = new GlobalConfig();
        config.setActiveRecord(true)//开启AR模式
                .setAuthor("ty")//设置作者
                //生成路径(一般都是生成在此项目的src/main/java下面)
.setOutputDir("E:\\develop\\Java\\workspace\\ideaworkspace\\mpg\\src\\main\\java")
                .setFileOverride(true)//第二次生成会把第一次生成的覆盖掉
                .setIdType(IdType.AUTO)//主键策略
                .setServiceName("%sService")//生成的service接口名字首字母是否为I，这样设置就没有I
                .setBaseResultMap(true)//生成resultMap
                .setBaseColumnList(true);//在xml中生成基础列
        //2、数据源配置
        DataSourceConfig dataSourceConfig = new DataSourceConfig();
        dataSourceConfig.setDbType(DbType.MYSQL)//数据库类型
                        .setDriverName("com.mysql.jdbc.Driver")
                        .setUrl("jdbc:mysql:///数据库名")
                        .setUsername("数据库用户名")
                        .setPassword("数据库密码");
        //3、策略配置
        StrategyConfig strategyConfig = new StrategyConfig();
        strategyConfig.setCapitalMode(true)//开启全局大写命名
                      .setDbColumnUnderline(true)//表名字段名使用下划线
                      .setNaming(NamingStrategy.underline_to_camel)//下划线到驼峰的命名方式
                      .setTablePrefix("tb_")//表名前缀
                      .setEntityLombokModel(true)//使用lombok
                      .setInclude("表1","表2");//逆向工程使用的表
        //4、包名策略配置
        PackageConfig packageConfig = new PackageConfig();
        packageConfig.setParent("com.ty.mpg")//设置包名的parent
                     .setMapper("mapper")
                     .setService("service")
                     .setController("controller")
                     .setEntity("entity")
```

```
                .setXml("mapper");//设置xml文件的目录
        //5、整合配置
        AutoGenerator autoGenerator = new AutoGenerator();
        autoGenerator.setGlobalConfig(config)
                .setDataSource(dataSourceConfig)
                .setStrategy(strategyConfig)
                .setPackageInfo(packageConfig);
        //6、执行
        autoGenerator.execute();
    }
}
```

注：以上便是示例代码，只要运行该junit测试，就会生成entity、mapper接口、mapper的xml文件、service、serviceImpl、controller代码。每一个设置代码中均有详细注释，此处不再赘述。

四、自定义全局操作：

(一)、AutoSqlInjector ：

BaseMapper提供了17个常用方法，但是有些需求这些方法还是不能很好的实现，那么怎么办呢？大家肯定会想到是在xml文件中写sql语句解决。这样确实可以，因为MP是只做增强不做改变，我们完全可以按照mybatis的原来的方式来解决。不过MP也提供了另一种解决办法，那就是自定义全局操作。所谓自定义全局操作，也就是我们可以在mapper中自定义一些方法，然后通过某些操作，让自定义的这个方法也能像BaseMapper的内置方法，供全局调用。接下来就看看如何实现(以deleteAll方法为例)。

1、在mapper接口中定义方法：

```
public interface EmplopyeeDao extends BaseMapper<Employee> {
    int deleteAll();
}
public interface UserDao extends BaseMapper<User> {
    int deleteAll();
}
```

在这两个mapper接口中都定义了deleteAll方法。

2、编写自定义注入类：

```
public class MySqlInjector extends AutoSqlInjector {
    @Override
    public void inject(Configuration configuration, MapperBuilderAssistant builderAssistant,
                       Class<?> mapperClass, Class<?> modelClass, TableInfo table) {
        /* 添加一个自定义方法 */
        deleteAllUser(mapperClass, modelClass, table);
        System.out.println(table.getTableName());
    }

    public void deleteAllUser(Class<?> mapperClass, Class<?> modelClass, TableInfo table) {
        /* 执行 SQL ，动态 SQL 参考类 SqlMethod */
        String sql = "delete from " + table.getTableName();
        /* mapper 接口方法名一致 */
        String method = "deleteAll";
        SqlSource sqlSource = languageDriver.createSqlSource(configuration, sql, modelClass);
        this.addDeleteMappedStatement(mapperClass, method, sqlSource);
    }
}
```

注：该类继承AutoSqlInjector，重写inject方法。然后编写sql语句，指定mapper接口中的方法，最后调用addDeleteMappedStatement方法即可。

3、在spring配置文件中配置：

```
<!-- 定义自定义注入器 -->
<bean class="com.ty.mybatisplus.injector.MySqlInjector" id="mySqlInjector"/>
<!-- 5、mybatisplus的全局策略配置 -->
    <bean id="globalConfiguration"
```

```
class="com.baomidou.mybatisplus.entity.GlobalConfiguration">
        <property name="idType" value="0"/>
        <property name="tablePrefix" value="tb_"/>
        <!-- 注入自定义全局操作 -->
        <property name="sqlInjector" ref="mySqlInjector"/>
    </bean>
```

注：先把刚才自定义的类注册成bean，然后在全局策略配置的bean中引用自定义类的bean即可。

4、测试：

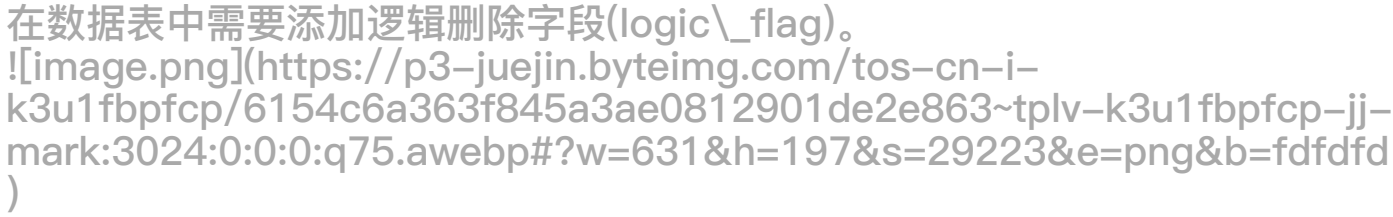
```
@Test
public void testMySqlInjector(){
    Integer result = userDao.deleteAll();
    System.out.println(result);
}

@Test
public void testMySqlInjector2(){
    Integer result = emplopyeeDao.deleteAll();
    System.out.println(result);
}
```

注：经测试，当userDao调用deleteAll方法时，会删除tb_user表的所有数据，employeeDao调用deleteAll方法时，会删除tb_employee表的所有数据。说明deleteAll方法是有效的。不过在运行这两个测试时，由于是全表删除操作，所有要先把执行分析插件关了。

(二)、逻辑删除：

其实数据并不会轻易的删除掉，毕竟数据收集不易，所以就有了逻辑删除。逻辑删除: 并不会真正的从数据库中将数据删除掉，而是将当前被删除的这条数据中的一个逻辑删除字段置为删除状态，比如该数据有一个字段logic_flag，当其值为1表示未删除，值为-1表示删除，那么逻辑删除就是将1变成-1。

1、数据表：

在数据表中需要添加逻辑删除字段(logic_flag)。

)

2、实体类：

```
@Data
public class User{
    private Integer id;
    private String name;
    private Integer age;
    private Integer gender;
    @TableLogic //标记逻辑删除属性
    private Integer logicFlag;
}
```

注：数据库中逻辑删除字段是logic_flag，所以实体类中的logicFlag需要用@TableLogic注解标记。

3、mapper:

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

4、配置逻辑删除：

需要在spring-dao.xml中做如下配置：
首先定义逻辑删除的bean：

```
<!-- 逻辑删除 -->
<bean class="com.baomidou.mybatisplus.mapper.LogicSqlInjector"
id="logicSqlInjector"/>
再在全局配置的bean中注入逻辑删除以及逻辑删除值：

<!-- 5、mybatisplus的全局策略配置 -->
    <bean id="globalConfiguration"
class="com.baomidou.mybatisplus.entity.GlobalConfiguration">
        <!-- 此处省略其他全局配置 -->
        <!-- 注入自定义全局操作，做逻辑删除时需要先注释掉 -->
        <!--<property name="sqlInjector" ref="mySqlInjector"/>-->
        <!-- 注入逻辑删除，先要把自定义的注释掉 -->
        <property name="sqlInjector" ref="logicSqlInjector"/>
        <!-- 注入逻辑删除值 -->
        <property name="logicDeleteValue" value="-1"/><!-- -1是删除状
态 -->
        <property name="logicNotDeleteValue" value="1"/><!-- 1是未删除
状态 -->
    </bean>
```

注：因为逻辑删除实际上也是一个sqlInjector，所以先要把刚才做自定义全局操作时注入的自定义全局操作注释掉，上面代码中已有详细注释说明。

5、测试：

```
@Test
    public void testLogicDelete(){
        Integer result = userDao.deleteById(1);
        System.out.println(result);
        //User user = userDao.selectById(1);
        //System.out.println(user);
    }
```

注：运行该测试，执行删除操作的时候，真正执行的sql语句是UPDATE tb_user SET logic_flag=-1 WHERE id=?，就是把逻辑删除字段的值设置为-1；当逻辑删除字段的值是-1时再执行查询操作，sql是SELECT ... FROM tb_user WHERE id=? AND logic_flag=1，所以查询结果是null。

五、公共字段自动填充：

我们知道，当我们进行插入或者更新操作时，没有设置值的属性，那么在数据表中要么是为null，要么是保留原来的值。有的时候我们我们没有赋值但是却不想让其为空，比如name属性，我们插入时会默认赋上“林志玲”，更新时会默认赋值上“朱茵”，那么就可以用公共字段自动填充。

1、使用@TableField注解标记填充字段

```
@TableField(fill = FieldFill.INSERT_UPDATE)//插入和更新时填充
 private String name;
```

2、编写公共字段填充处理器类：

```
public class MyMetaObjectHandler extends MetaObjectHandler {
    @Override
    public void insertFill(MetaObject metaObject) {
        Object fieldValue = getFieldValByName(”name”,metaObject); //获取需要填充的字段
        if(fieldValue == null){   //如果该字段没有设置值
            setFieldValByName(”name”,”林志玲”,metaObject); //那就将其设置为”林志玲”
        }
    }

    @Override
    public void updateFill(MetaObject metaObject) {
        Object fieldValue = getFieldValByName(”name”,metaObject);//获取需要填充的字段
        if(fieldValue == null){ //如果该字段没有设置值
            setFieldValByName(”name”,”朱茵”,metaObject);  //那就将其设置为”朱茵”
        }
    }
}
```

注：该类继承了MetaObjectHandler类，重写了insertFill和updateFill方法，在这两个方法获取需要填充的字段以及默认填充的值。

3、在spring-dao.xml中配置：

```
<!-- 公共字段填充处理器 -->
<bean class="com.ty.mybatisplus.handler.MyMetaObjectHandler"
id="myMetaObjectHandler"/>
<!-- 5、mybatisplus的全局策略配置 -->
    <bean id="globalConfiguration"
class="com.baomidou.mybatisplus.entity.GlobalConfiguration">
        <!-- 此处省略其他配置 -->
        <!-- 注入公共字段填充处理器 -->
        <property name="metaObjectHandler"
ref="myMetaObjectHandler"/>
    </bean>
```

注：和配置逻辑删除一样，都是先将自定义的类注册成bean，再在全局策略配置中引用这个bean即可。

4、测试：

```
@Test
public void testHandlerInsert() {
        User user = new User();
        user.setGender(1);
        user.setAge(22);
        user.setLogicFlag(1);
        userDao.insert(user);
}
```

注：可以看到，虽然我们并没有给name赋值，但是已经自动把“林志玲”传进去

了。更新时也一样有效，此处就不将测试代码贴出来了。

原文链接: https://juejin.cn/post/7386461612700712970