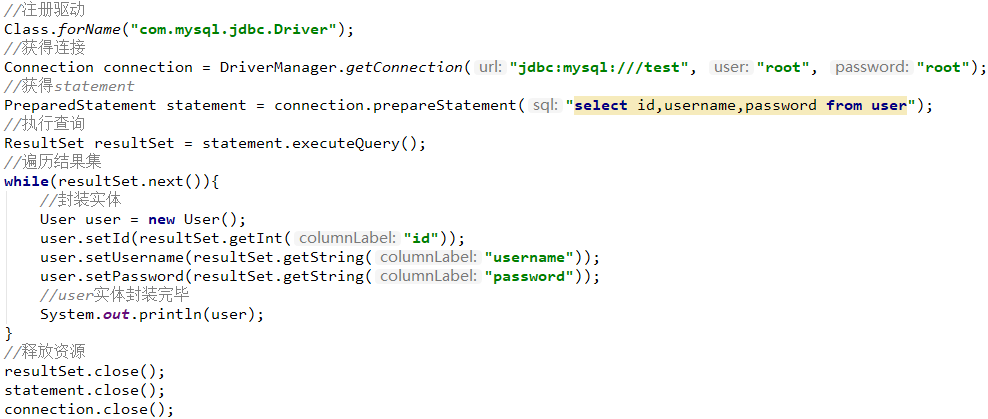
动力节点：https://www.bilibili.com/video/BV185411s7Ry?p=34

# 第一章 MyBatis入门操作

## 一、MyBatis的简介

### 原始jdbc操作

#### 1.1 查询数据



#### 1.2 插入数据



#### 1.3 原始jdbc操作的分析

原始jdbc开发存在的问题如下：

* 数据库连接创建、释放频繁造成系统资源浪费从而影响系统性能
* sql 语句在代码中硬编码，造成代码不易维护，实际应用 sql 变化的可能较大，sql 变动需要改变java代码。
* 查询操作时，需要手动将结果集中的数据手动封装到实体中。插入操作时，需要手动将实体的数据设置到sql语句的占位符位置

应对上述问题给出的解决方案：

* 使用数据库连接池初始化连接资源
* 将sql语句抽取到xml配置文件中
* 使用反射、内省等底层技术，自动将实体与表进行属性与字段的自动映射

### 2、Mybatis简介

* mybatis 是一个优秀的基于java的持久层框架，它内部封装了jdbc，使开发者只需要关注sql语句本身，而不需要花费精力去处理加载驱动、创建连接、创建statement等繁杂的过程。
* mybatis通过xml或注解的方式将要执行的各种 statement配置起来，并通过java对象和statement中sql的动态参数进行映射生成最终执行的sql语句。
* 最后mybatis框架执行sql并将结果映射为java对象并返回。采用ORM思想解决了实体和数据库映射的问题，对jdbc 进行了封装，屏蔽了jdbc api 底层访问细节，使我们不用与jdbc api 打交道，就可以完成对数据库的持久化操作。

## 二、MyBatis的快速入门

### MyBatis开发步骤

MyBatis官网地址：<http://www.mybatis.org/mybatis-3/>

MyBatis开发步骤：

1. 添加MyBatis的坐标
2. 创建user数据表
3. 编写User实体类
4. 编写映射文件UserMapper.xml
5. 编写核心文件SqlMapConfig.xml
6. 编写测试类

### 环境搭建

1. 添加MyBatis的坐标

<!--mybatis坐标-->  
<dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis</artifactId>  
 <version>3.4.5</version>  
</dependency>

<!--mysql驱动坐标-->  
<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.6</version>  
 <scope>runtime</scope>  
</dependency>

<!--单元测试坐标-->  
<dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.12</version>  
 <scope>test</scope>  
</dependency>

<!--日志坐标-->  
<dependency>  
 <groupId>log4j</groupId>  
 <artifactId>log4j</artifactId>  
 <version>1.2.12</version>  
</dependency>

1. 创建user数据表



1. 编写User实体类

**public class** User {  
  
 **private int id**;  
 **private** String **username**;  
 **private** String **password**;

//省略get个set方法

}

1. 编写映射文件：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="userMapper"**>  
  
 *<!--查询操作-->* <**select id="findAll" resultType="User"**>  
 select *\** from user  
 </**select**>  
  
</**mapper**>

1. 编写核心文件：sqlMapConfig.xml

*<?***xml version="1.0" encoding="UTF-8"** *?>***<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-config.dtd"*>***<**configuration**>  
  
 *<!--数据源环境-->* <**environments default="developement"**>  
 <**environment id="developement"**>  
 <**transactionManager type="JDBC"**></**transactionManager**>  
 <**dataSource type="POOLED"**>  
 <**property name="driver" value="com.mysql.jdbc.Driver"**/>  
 <**property name="url" value="jdbc:mysql://localhost:3306/test"**/>  
 <**property name="username" value="root"**/>  
 <**property name="password" value="root"**/>  
 </**dataSource**>  
 </**environment**>  
 </**environments**>  
  
 *<!--加载映射文件-->* <**mappers**>  
 <**mapper resource="com/itheima/mapper/UserMapper.xml"**></**mapper**>  
 </**mappers**>

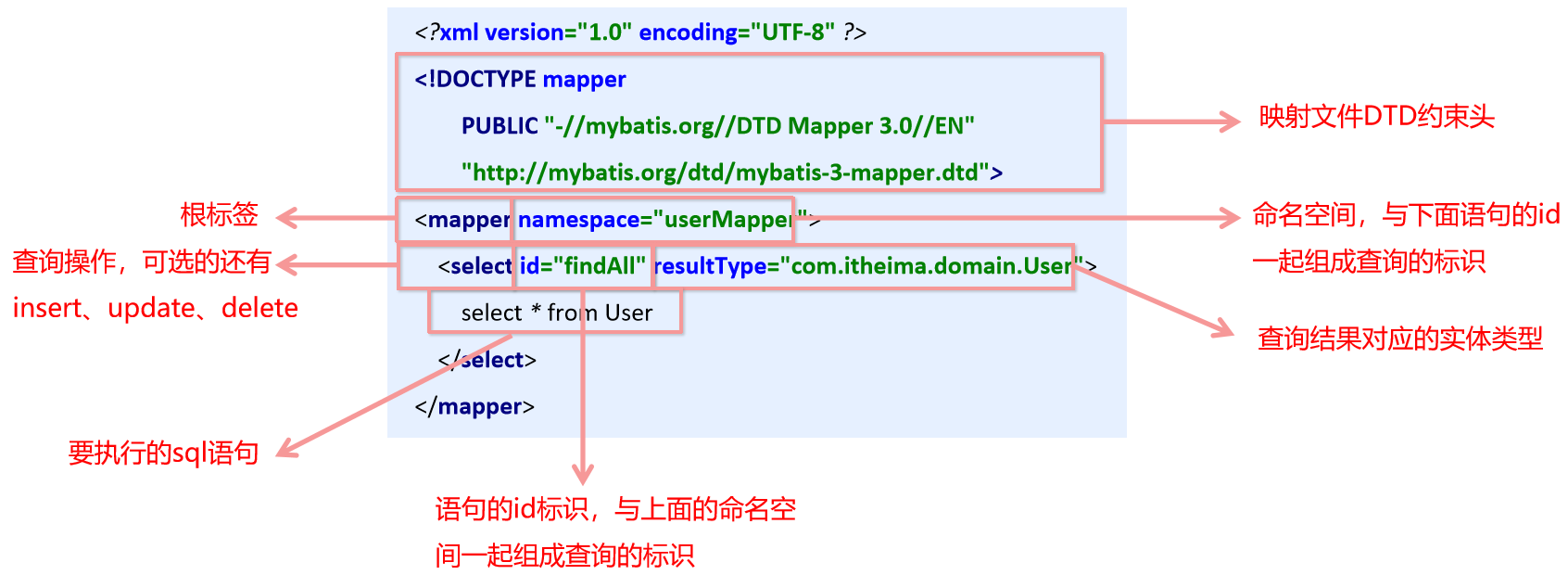
</**configuration**>

### 编写测试代码

**查询操作**

*//获得核心配置文件*InputStream resourceAsStream = Resources.*getResourceAsStream*(**"sqlMapConfig.xml"**);  
*//获得session工厂对象*SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(resourceAsStream);  
*//获得session回话对象*SqlSession sqlSession = sqlSessionFactory.openSession();  
*//执行操作 参数：namespace+id*List<User> userList = sqlSession.selectList(**"userMapper.findAll"**);  
*//打印数据*System.***out***.println(userList);  
*//释放资源*sqlSession.close();

## MyBatis的映射文件概述



## MyBatis的增删改查操作

### 插入数据操作

1）编写UserMapper映射文件

*<?***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="userMapper"**>

*<!--插入操作-->* <**insert id="save" parameterType="com.itheima.domain.User"**>  
 insert into user values(#{id},#{username},#{password})  
 </**insert**>

</**mapper**>

1. 编写测试代码

**public class** MyBatisTest {  
 @Test  
 *//插入操作* **public void** test2() **throws** IOException {  
  
 *//模拟user对象* User user = **new** User();  
 user.setUsername(**"xxx"**);  
 user.setPassword(**"abc"**);  
  
 *//获得核心配置文件* InputStream resourceAsStream = Resources.*getResourceAsStream*(**"sqlMapConfig.xml"**);  
 *//获得session工厂对象* SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(resourceAsStream);  
 *//获得session回话对象* SqlSession sqlSession = sqlSessionFactory.openSession(**true**);  
 *//执行操作 参数：namespace+id* sqlSession.insert(**"userMapper.save"**,user);  
  
 *//mybatis执行更新操作 提交事务* sqlSession.commit(); *//释放资源* sqlSession.close();  
 }  
}

1. 插入操作注意问题：

* 插入语句使用insert标签
* 在映射文件中使用parameterType属性指定要插入的数据类型
* Sql语句中使用#{实体属性名}方式引用实体中的属性值
* 插入操作使用的API是sqlSession.insert(“命名空间.id”,实体对象);
* 插入操作涉及数据库数据变化，所以要使用sqlSession对象显示的提交事务，即sqlSession.commit()

### 修改数据操作

1. 编写UserMapper映射文件

*<?***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="userMapper"**>

*<!--修改操作-->* <**update id="update" parameterType="com.itheima.domain.User"**>  
 update user set username=#{username},password=#{password} where id=#{id}  
 </**update**>  
  
</**mapper**>

1. 编写测试代码

**public class** MyBatisTest {  
 @Test  
 *//修改操作* **public void** test3() **throws** IOException {  
  
 *//模拟user对象* User user = **new** User();  
 user.setId(7);  
 user.setUsername(**"lucy"**);  
 user.setPassword(**"123"**);  
  
 *//获得核心配置文件* InputStream resourceAsStream = Resources.*getResourceAsStream*(**"sqlMapConfig.xml"**);  
 *//获得session工厂对象* SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(resourceAsStream);  
 *//获得session回话对象* SqlSession sqlSession = sqlSessionFactory.openSession();  
 *//执行操作 参数：namespace+id* sqlSession.update(**"userMapper.update"**,user);  
  
 *//mybatis执行更新操作 提交事务* sqlSession.commit();  
  
 *//释放资源* sqlSession.close();  
 }  
}

1. 修改操作注意问题：

* 修改语句使用update标签
* 修改操作使用的API是sqlSession.update(“命名空间.id”,实体对象);

### 删除数据操作

1. 编写UserMapper映射文件

*<?***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="userMapper"**>

*<!--删除操作-->* <**delete id="delete" parameterType="int"**> //也可以写java.lang.Integer  
 delete from user where id=#{abc}  
 </**delete**>  
  
</**mapper**>

1. 编写测试代码

**public class** MyBatisTest {  
 @Test  
 *//删除操作* **public void** test4() **throws** IOException {  
  
 *//获得核心配置文件* InputStream resourceAsStream = Resources.*getResourceAsStream*(**"sqlMapConfig.xml"**);  
 *//获得session工厂对象* SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(resourceAsStream);  
 *//获得session回话对象* SqlSession sqlSession = sqlSessionFactory.openSession();  
 *//执行操作 参数：namespace+id* sqlSession.delete(**"userMapper.delete"**,8);  
  
 *//mybatis执行更新操作 提交事务* sqlSession.commit();  
  
 *//释放资源* sqlSession.close();  
 }  
}

1. 删除操作注意问题

* 删除语句使用delete标签
* Sql语句中使用#{任意字符串}方式引用传递的单个参数
* 删除操作使用的API是sqlSession.delete(“命名空间.id”,Object)

### 查询指定数据操作

1. 编写UserMapper映射文件

*<?***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="userMapper"**>  
  
 *<!--根据id进行查询-->* <**select id="findById" resultType="User" parameterType="int"**>  
 select *\** from user where id=#{id}  
 </**select**>  
  
</**mapper**>

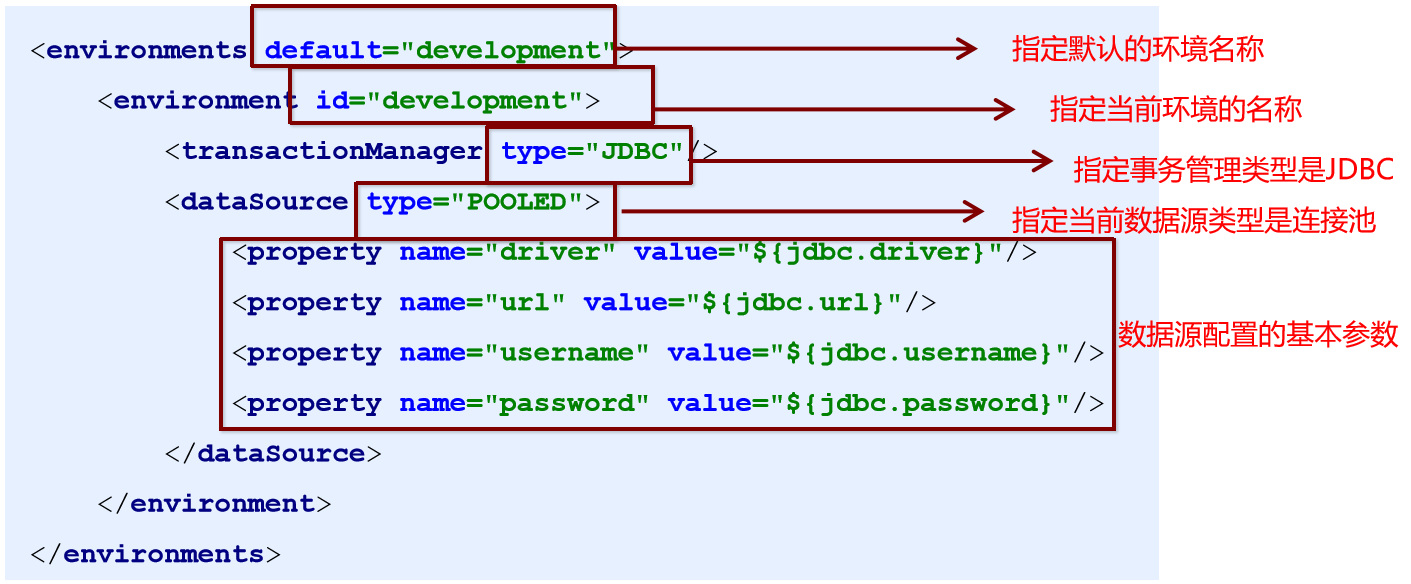
1. 编写测试代码

**public class** MyBatisTest {  
 @Test  
 *//查询一个对象* **public void** test5() **throws** IOException {  
 *//获得核心配置文件* InputStream resourceAsStream = Resources.*getResourceAsStream*(**"sqlMapConfig.xml"**);  
 *//获得session工厂对象* SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(resourceAsStream);  
 *//获得session回话对象* SqlSession sqlSession = sqlSessionFactory.openSession();  
 *//执行操作 参数：namespace+id* User user = sqlSession.selectOne(**"userMapper.findById"**, 1);  
 *//打印数据* System.***out***.println(user);  
 *//释放资源* sqlSession.close();  
 }  
}

## MyBatis的核心配置文件概述



### environments标签



数据库环境的配置，支持多环境配置

属性 default ，必须是某个environment的id属性值。 表示mybatis当前连接的数据库

其中，事务管理器（transactionManager）类型有两种：

* JDBC：这个配置就是直接使用了JDBC 的提交和回滚设置，它依赖于从数据源得到的连接来管理事务作用域。
* MANAGED：这个配置几乎没做什么。它从来不提交或回滚一个连接，而是让容器来管理事务的整个生命周期（比如 JEE 应用服务器的上下文）。 默认情况下它会关闭连接，然而一些容器并不希望这样，因此需要将 closeConnection 属性设置为 false 来阻止它默认的关闭行为。

其中，数据源（dataSource）类型有三种：

* UNPOOLED：这个数据源的实现只是每次被请求时打开和关闭连接。
* POOLED：这种数据源的实现利用“池”的概念将 JDBC 连接对象组织起来。
* JNDI：这个数据源的实现是为了能在如 EJB 或应用服务器这类容器中使用，容器可以集中或在外部配置数据源，然后放置一个 JNDI 上下文的引用。

### mappers标签

*<!--指定其他mapper文件的位置：  
 其他其他mapper文件目的是找到其他文件的sql语句  
-->*<**mappers**>  
 *<!--第一种方式， resources="mapper文件的路径"  
 优点：文件清晰。 加载的文件是明确的。  
 文件的位置比较灵活。  
 缺点：文件比较多， 代码量会比较大， 管理难度大  
 -->* <**mapper resource="com/bjpowernode/dao/StudentDao.xml"**/>  
  
 *<!--  
 第二种方式，使用<package>  
 name:包名， mapper文件所在的包名。  
 特点： 把这个包中的所有mapper文件，一次加载。  
  
 使用要求：  
 1. mapper文件和dao接口在同一目录  
 2. mapper文件和dao接口名称完全一样。  
 -->* <**package name="com.bjpowernode.dao"** />  
</**mappers**>

该标签的作用是加载映射的，加载方式有如下几种：

* 使用相对于类路径的资源引用，例如：<mapper resource="org/mybatis/builder/AuthorMapper.xml"/>
* 使用完全限定资源定位符（URL），例如：<mapper url="file:///var/mappers/AuthorMapper.xml"/>
* 使用映射器接口实现类的完全限定类名，例如：<mapper class="org.mybatis.builder.AuthorMapper"/>
* 将包内的映射器接口实现全部注册为映射器，例如：<package name="org.mybatis.builder"/>

### Properties标签

实际开发中，习惯将数据源的配置信息单独抽取成一个properties文件，该标签可以加载额外配置的properties文件

**jdbc.properties文件内容：**

**jdbc.driver**=**com.mysql.jdbc.Driver  
jdbc.url**=**jdbc:mysql://localhost:3306/test  
jdbc.username**=**root  
jdbc.password**=**root**



### typeAliases标签（自定义别名）

自定义别名的步骤：

1）在mybatis主配置文件，使用 typeAliases标签声明别名

2）在mapper文件中， resultType="别名"

mybatis.xml配置文件：

*<!--声明别名-->*<**typeAliases**>

*<!--第一种语法格式  
 type：java类型的全限定名称（自定义类型）  
 alias：自定义别名  
 优点： 别名可以自定义  
 缺点： 每个类型必须单独定义  
 -->* <**typeAlias type="com.bjpowernode.domain.Student" alias="stu"** />  
 <**typeAlias type="com.bjpowernode.vo.QueryParam" alias="qp"** />  
  
</**typeAliases**>

Map.xml文件：

<**select id="selectById" parameterType="integer" resultType="com.bjpowernode.domain.Student"**> //不使用别名  
 select id,name,email,age from student where id=#{studentId}  
</**select**>

<**select id="selectById" parameterType="integer" resultType="stu"**> //使用别名  
 select id,name,email,age from student where id=#{studentId}  
</**select**>

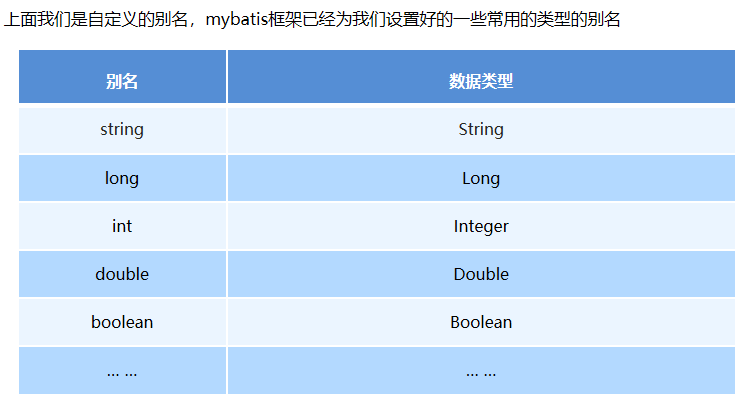
mybatis.xml配置文件：

*<!--声明别名-->*<**typeAliases**>

*<!--第二种方式  
 name：包名， mybatis会把这个包中所有类名作为别名（不用区分大小写）  
 优点：使用方便，一次给多个类定义别名  
 缺点: 别名不能自定义，必须是类名。若不同包具有相同类名时，执行sql会报错  
 -->* <**package name="com.bjpowernode.domain"** />  
 <**package name="com.bjpowernode.vo"** />  
  
</**typeAliases**>

Map.xml文件：

<**select id="selectById" parameterType="integer" resultType="Student"**> //或者student  
 select id,name,email,age from student where id=#{studentId}  
</**select**>



**注意点：**配置文件中，自定义的别名位置应该在settings后，具体顺序为：





### 总结

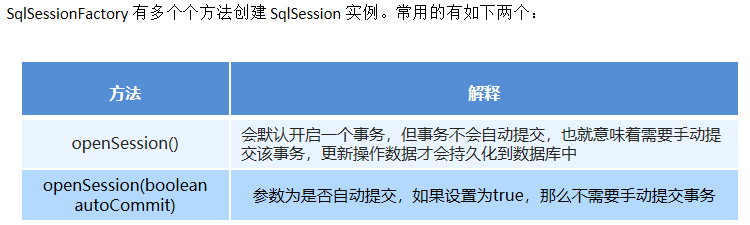


## 六、MyBatis的相应API

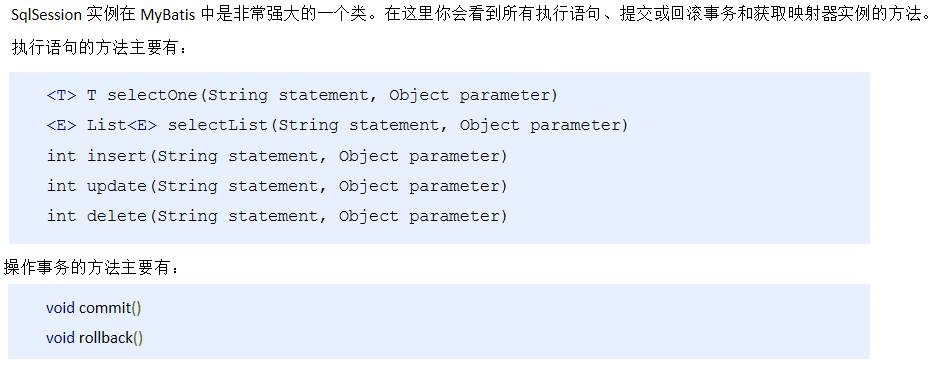
### SqlSession工厂构建器SqlSessionFactoryBuilder



### SqlSession工厂对象SqlSessionFactory



### SqlSession会话对象



# 第二章 MyBatis的Dao层实现方式

## 一、传统开发方式

### 编写UserDao接口

## 二、代理开发方式

### 代理开发方式介绍

采用 Mybatis 的代理开发方式实现 DAO 层的开发，这种方式是我们后面进入企业的主流。

Mapper 接口开发方法只需要程序员编写Mapper 接口（相当于Dao 接口），由Mybatis 框架根据接口定义创建接口的动态代理对象，代理对象的方法体同上边Dao接口实现类方法。

Mapper 接口开发需要遵循以下规范：

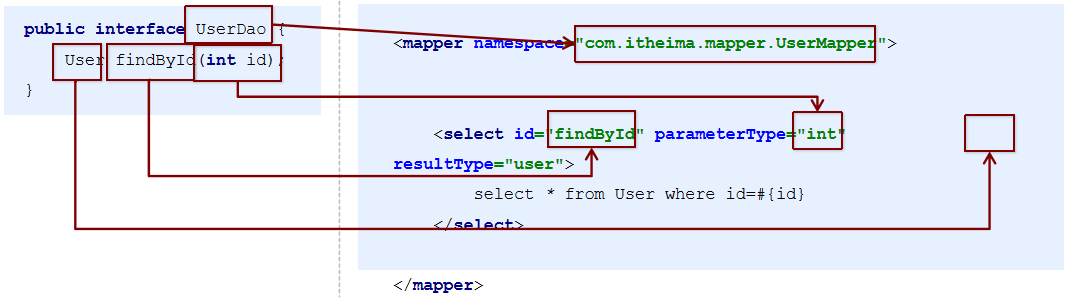
1、 Mapper.xml文件中的namespace与mapper接口的全限定名相同

2、 Mapper接口方法名和Mapper.xml中定义的每个statement的id相同

3、 Mapper接口方法的输入参数类型和mapper.xml中定义的每个sql的parameterType的类型相同

4、 Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的resultType的类型相同

### 编写UserMapper接口



### 测试代理方式

InputStream resourceAsStream = Resources.*getResourceAsStream*(**"sqlMapConfig.xml"**);  
SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(resourceAsStream);  
SqlSession sqlSession = sqlSessionFactory.openSession();

//获得MyBatis框架生成的UserMapper接口的实现类  
UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);  
List<User> all = mapper.findAll();  
System.***out***.println(all);  
  
User user = mapper.findById(1);  
System.***out***.println(user);

# 第三章 MyBatis的参数传递

## 一、理解参数

通过java程序把数据传入到mapper文件中的sql语句。 参数主要是指dao接口方法的形参

### parameterType

parameterType：表示参数的类型， 指定dao方法的形参数据类型。 这个形参的数据类型是给mybatis使用。 mybatis在给sql语句的参数赋值时使用。 PreparedStatement.setXXX( 位置， 值)

* 第一个用法： java类型的全限定类型名称 parameterType="java.lang.Integer"
* 第二个用法： mybatis定义的java类型的别名 parameterType="int"
* parameterType：mybatis通过反射机制可以获取 dao接口方法参数的类型，可以不写

DAO接口方法形参类型为Integer：

Student selectById(Integer id);

Map.xml文件parameterType的值可以为integer、int、java.lang.Integer：

<**select id="selectById" parameterType="integer" resultType="student"**>  
 select id,name,email,age from student where id=#{studentId}  
</**select**>

### 一个简单类型参数

Dao接口方法的参数只有一个简单类型（java 基本类型和 String），可使用占位符 #{ 任意字符 }，和方法的参数名无关。

DAO接口方法形参是一个简单类型的：

Student selectByEmail(String email);

Map.xml文件，获取这个参数值，使用#{任意字符}

<**select id="selectByEmail" resultType="student"**>  
 select id,name,email,age from student where email=#{studentEmail}   
</**select**> //studentEmail是自定义的变量名称，和方法参数名无关

### 多个简单类型参数

当Dao接口方法多个参数，需要通过名称使用参数。在方法形参前面加入@Param(“自定义参数名”)，mapper 文件使用#{自定义参数名}。

例如定义List<Student> selectStudent( @Param(“personName”) String name ) { … }

mapper文件：select \* from student where name = #{ personName}

*/\*DAO接口方法：  
 多个简单类型的参数  
 使用@Param命名参数， 注解是mybatis提供的  
 位置：在形参定义的前面  
 属性：value自定义的参数名称  
 \*/*List<Student> selectByNameOrAge(@Param(value = **"myname"**) String name,  
 @Param(**"myage"**) Integer age);

*<!--*mapper文件： *多个简单类型的参数.  
 当使用了@Param命名后，例如@Param("myname").  
 在mapper中，使用#{命名的参数}， 例如 #{myname}  
-->*<**select id="selectByNameOrAge" resultType="com.bjpowernode.domain.Student"**>  
 select id,name,email,age from student where name=#{myname} or age=#{myage}  
</**select**>

### 使用对象作为参数

使用java对象传递参数，java的属性值就是sql需要的参数值。每一个属性就是一个参数。

语法格式：#{ property,javaType=java中数据类型名,jdbcType=数据类型名称}，javaType、jdbcType的类型MyBatis可以检测出来，一般不需要设置。常用格式 #{property}

**public class** Student {  
 **private** Integer **id**;  
 **private** String **name**;  
 **private** String **email**;  
 **private** Integer **age**;  
 *//提供set、get方法*}

**public class** QueryParam {  
 **private** Object **p1**;  
 **private** Object **p2**;  
 *//提供get、set方法*}

*/\* DAO接口方法：  
 \* 一个java对象作为参数(对象有属性，每个属性有set，get方法)  
 \*/*List<Student> selectByObject(Student student);  
  
List<Student> selectByQueryParam(QueryParam param);

*<!--*mapper文件： *一个java对象作为方法的参数，使用对象的属性作为参数值使用  
 简单的语法格式： #{属性名} ， mybatis调用此属性的getXXX()方法获取属性值  
-->*<**select id="selectByObject" resultType="com.bjpowernode.domain.Student"**>  
 select id,name,email,age from student where name=#{name} or age=#{age}  
</**select**>  
  
*<!--复杂的语法格式： #{属性名,javaType=java类型的全限定名称,jdbcType=mybatis中定义列的数据类型}-->*<**select id="selectByObject" resultType="com.bjpowernode.domain.Student"**>  
 select id,name,email,age from student where  
 name=#{name,javaType=java.lang.String,jdbcType=VARCHAR}  
 or  
 age=#{age,javaType=java.lang.Integer,jdbcType=INTEGER}  
</**select**>  
  
*//任一java对象均可*  
<**select id="selectByQueryParam" resultType="com.bjpowernode.domain.Student"**>  
 select id,name,email,age from student where name=#{p1} or age=#{p2}  
</**select**>

### 多个参数，使用位置（了解）

参数位置从0开始，引用参数语法 #{arg 位置}，第一个参数是#{arg0}, 第二个是#{arg1}。注意：mybatis-3.3版本和之前的版本使用#{0},#{1}方式， 从mybatis3.4开始使用#{arg0}方式。

*/\* DAO接口方法：  
 使用位置，获取参数  
 \*/*List<Student> selectByPosition(String name,Integer age);

*<!--*mapper文件： *使用位置获取参数值， dao接口方法是多个简单类型的参数  
 语法： #{arg0}, #{arg1}....  
-->*<**select id="selectByPosition" resultType="com.bjpowernode.domain.Student"**>  
 select id,name,email,age from student where name=#{arg0} or age=#{arg1}  
</**select**>

### 多个参数，使用Map

Map 集合可以存储多个值，使用Map向mapper文件一次传入多个参数。Map集合使用String的key，

Object类型的值存储参数。mapper文件使用#{key}引用参数值。

*/\* DAO接口方法：  
 使用Map作为参数  
 \*/*List<Student> selectStudentByMap(Map<String,Object> map);

*<!--*mapper文件： *使用Map传递参数，  
 在mapper文件中，获取map的值，是通过key获取的，语法：#{key}  
-->*<**select id="selectStudentByMap" resultType="com.bjpowernode.domain.Student"**>  
 select id,name,email,age from student where name=#{myname} or age=#{myage}  
</**select**>

测试：

SqlSession sqlSession = MyBatisUtil.*getSqlSession*();  
StudentDao dao = sqlSession.getMapper(StudentDao.**class**);  
*//使用map传递参数*Map<String,Object> data = **new** HashMap<>();  
data.put(**"myname"**, **"李思思"**);  
data.put(**"myage"**, 20);  
List<Student> students = dao.selectStudentByMap(data);  
  
students.forEach( stu-> System.***out***.println(**"stu="**+stu));  
sqlSession.close();

## # 和 $ 的区别

### # 占位符

告诉mybatis使用实际的参数值代替。并使用PrepareStatement 对象执行 sql 语句, #{…}代替 sql 语句的“?”。这样做更

安全，更迅速，通常也是首选做法

#{}特点：

1）使用的PrepareStatement对象，执行sql语句，效率高。

2）使用的PrepareStatement对象，能避免sql语句， sql语句执行更安全。

3）#{}常常作为列值使用的，位于等号的右侧，#{}位置的值和数据类型有关的。(自动区分数据类型)

mapper文件：

<**select id="selectById" parameterType="integer" resultType="student"**>  
 select id,name,email,age from student where id=#{studentId}  
</**select**>

*//mybatis创建PrepareStatement对象，执行sql语句*String ***sql***=**" select id,name,email,age from student where id=?"**;  
PrepareStatement ***pst*** = conn.prepareStatement(***sql***);  
***pst***.setInt(1,1001); *//传递参数*

***//pst***.setSring(1,**"1002"**); 会根据参数类型自动识别数据类型ResultSet ***rs*** = ***pst***.executeQuery(); *//执行sql语句*

*解释：*

*where id=? 就是 where id=#{studentId}*

*ps.setInt(1,1005) 1005会替换掉 #{studentId}*

### $ 占位符

告诉mybatis使用$包含的“字符串”替换所在位置。使用Statement 把 sql 语句和${}的内容连接起来。主要用在替换表名，列名，不同列排序等操作。

${}的特点:

1）使用Statement对象，执行sql语句，效率低

2）${}占位符的值，使用的字符串连接方式， 有sql注入的风险。 有代码安全的问题

3) ${}数据是原样使用的，不会区分数据类型。

4）${}常用作表名或者列名，在能保证数据安全的情况下使用${}

mapper文件：

<**select id="selectById" parameterType="integer" resultType="student"**>  
 select id,name,email,age from student where id=${studentId}  
</**select**>

*//mybatis创建PrepareStatement对象，执行sql语句*String ***sql*** = **"select id,name,email,age from student where id=" + "1001"**;

Statement ***stmt*** = conn.createStatement(***sql***);ResultSet ***rs*** = ***stmt*** .executeQuery(); *//执行sql语句*

*DAO接口方法：*

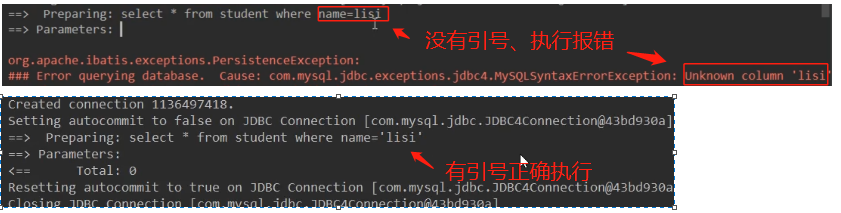
*//${}占位符的使用, 使用@Param 命名*List<Student> queryStudent(@Param(**"studentName"**) String name);  
  
List<Student> queryStudentOrderById();  
List<Student> queryStudentOrderByName();  
  
List<Student> queryStudentOrderByColName(@Param(**"myname"**) String name,  
 @Param(**"colName"**) String colName,  
 @Param(**"tableName"**) String tableName);

mapper文件：<**select id="queryStudent" resultType="com.bjpowernode.domain.Student"**>  
 select *\** from student where name=${studentName}  
</**select**>  
  
*<!--id列排序-->*<**select id="queryStudentOrderById" resultType="com.bjpowernode.domain.Student"**>  
 select *\** from student order by id  
</**select**>  
  
*<!--name列排序-->*<**select id="queryStudentOrderByName" resultType="com.bjpowernode.domain.Student"**>  
 select *\** from student order by name  
</**select**>  
  
<**select id="queryStudentOrderByColName" resultType="com.bjpowernode.domain.Student"**>  
 select *\** from ${tableName} where name=#{myname} order by ${colName} desc  
</**select**>

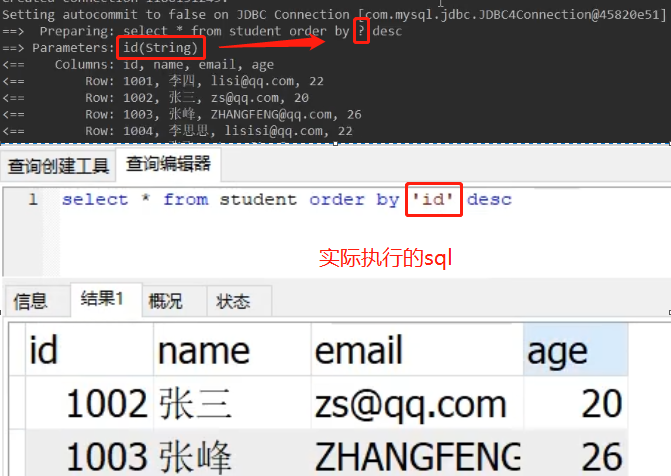
**注意1：**使用${studentName}，执行sql时，参数需根据具体的类型加以判断，若为String，需加引号，否则执行sql会报错

**public void** testQueryStudent(){  
 SqlSession sqlSession = MyBatisUtil.*getSqlSession*();  
 StudentDao dao = sqlSession.getMapper(StudentDao.**class**);  
  
 List<Student> students = dao.queryStudent(**"'lisi'"**);  
  
 students.forEach( stu-> System.***out***.println(**"stu="**+stu));  
 sqlSession.close();  
}

以下未加引号导致的报错：



**注意2：**使用#{colName}进行排序，执行sql时，参数类型会自动转换为String，导致未排序，此时只能采用${colName}进行排序



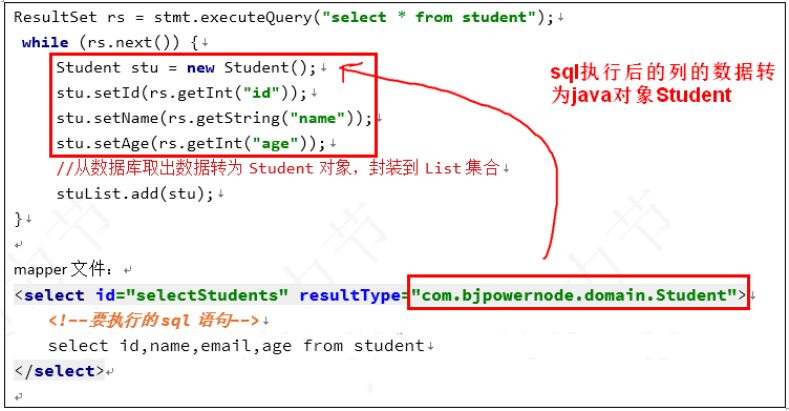
## 封装MyBatis输出结果

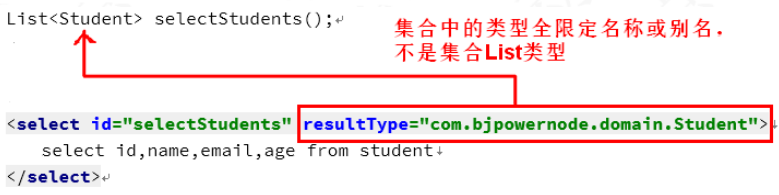
封装输出结果：MyBatis执行sql语句，得到ResultSet, 转为java对象

### resultType

resultType: 执行 sql 得到 ResultSet 转换的类型，表示mybatis执行sql后得到java对象类型，使用类型的完全限定名或别名。

注意：如果返回的是集合，那应该设置为集合包含的类型，而不是集合本身。resultType 和 resultMap，不能同时使用。





1. Java自定义对象

DAO接口方法：

Student selectById(Integer id);

Map.xml文件:

<**select id="selectById" resultType="com.bjpowernode.domain.Student"**> //或使用别名student  
 select id,name,email,age from student where email=#{studentId}   
</**select**>

*resultType：现在使用java类型的全限定名称。表示的意思 mybatis执行sql，把ResultSet中的数据转为Student类型的对象。 mybatis会做以下操作：  
1. 调用com.bjpowernode.domain.Student的无参数构造方法，创建对象。*Student ***student*** = **new** Student(); *//使用反射创建对象  
2. 同名的列赋值给同名的属性。*student.setId( rs.getInt(**"id"**));  
student.setName(rs.getString(**"name"**));  
*3. 得到java对象，如果dao接口返回值是List集合， mybatis把student对象放入到List集合。  
  
  
所以执行 Student mystudent = dao.selectById(1001); 得到 数据库中 id=1001这行数据，  
这行数据的列值， 付给了mystudent对象的属性。 你能得到mystudent对象。 就相当于是 id=1001这行数据。*

1. 简单数据类型

DAO接口方法：查询数量

**long** countStudent();

Map.xml文件:

*<!--*

*执行sql语句，得到是一个值（一行一列）*

*-->*

<**select id="countStudent" resultType="java.lang.Long"**> //或使用别名long  
 select count(\*) from student   
</**select**>

1. Map集合

**DAO接口方法**：

*//查询结果返回是一个Map*Map<Object,Object> selectMap(@Param(**"stuid"**) Integer id);

**Map.xml文件:**

*<!--  
 执行sql得到一个Map结构数据， mybatis执行sql，把ResultSet转为map  
 sql执行结果， 列名做map的key，列值作为value  
 sql执行得到是一行记录，转为map结构是正确的。  
 dao接口返回是一个map，****sql语句最多能获取一行记录，多余一行是错误*** *-->*<**select id="selectMap" resultType="java.util.HashMap"**>  
 select id,name,email from student where id != #{stuid}  
</**select**>

**//测试：**

**public void** testSelectMap(){  
 *//1.获取SqlSession* SqlSession session = MyBatisUtil.*getSqlSession*();  
 *//2.获取dao的代理* StudentDao dao = session.getMapper(StudentDao.**class**);  
  
 Map<Object,Object> map = dao.selectMap(1006);  
 System.***out***.println(**"map==="**+map);  
 *//3.关闭SqlSession对象* session.close();  
  
 System.***out***.println(**"name="**+map.get(**"name"**));  
 System.***out***.println(**"id="**+map.get(**"id"**));  
}

### 2、resultMap

resultMap可以自定义 sql 的结果和 java 对象属性的映射关系。更灵活的把列值赋值给指定属性。常用在列名和 java 对象属性名不一样的情况。

使用方式：

1.先定义 resultMap,指定列名和属性的对应关系。

2.在<select>中把 resultType 替换为 resultMap。

**定义java类：**属性cid、cname与数据库列名id、name不一致

**public class** CustomObject {  
  
 **private** Integer **cid**;  
 **private** String **cname**;  
 **private** String **email**;  
 **private** Integer **age**;  
 *//get、set方法*}

**DAO接口方法**：

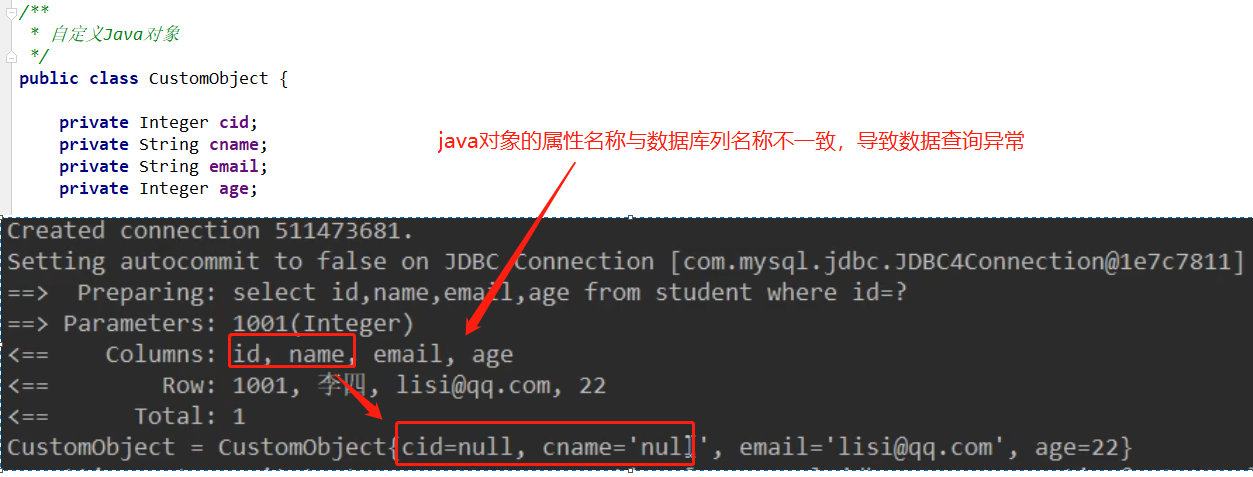
CustomObject selectById2(@Param(**"stuid"**) Integer id);

**Map.xml文件:**

*<!--使用resultMap定义列和属性的关系-->  
<!--定义resultMap  
 id：给resultMap的映射关系起个名称，唯一值  
 type：java类型的全限定名称  
-->*<**resultMap id="customMap" type="com.bjpowernode.vo.CustomObject"**>  
  
 *<!--定义列名和属性名的对应-->  
 <!--主键类型使用id标签-->* <**id column="id" property="cid"** />  
 *<!--非主键类型使用result标签-->* <**result column="name" property="cname"** />  
 *<!--列名和属性名相同的可以不用定义-->* <**result column="email" property="email"** />  
 <**result column="age" property="age"** />  
</**resultMap**>  
  
*<!--使用resultMap属性，指定映射关系的id  
 resultMap和resultType 不能同时使用， 二选一。  
-->*<**select id="selectById2" resultMap="customMap"**>  
 select id,name,email,age from student where id=#{stuid}  
</**select**>

### 3、解决属性名和列名不同的问题

由于Java命名规范与SQL命名规范可能存在差异性，导致实体类属性名和列名可能不一致，从而导致sql语句执行结果存在差异的问题。



解决方式一：使用resultMap：自定义列名和属性名称对应关系（参照resultMap的使用）

解决方式二：使用resultType: 使用列别名，让别名和java对象属性名称一样

**DAO接口方法**：

CustomObject selectById3(@Param(**"stuid"**) Integer id);

**Map.xml文件:**  
*<!-- 使用列别名，解决列名和属性名不同的问题 -->*<**select id="selectById2" resultType="com.bjpowernode.vo.CustomObject"**>  
 select id as cid,name cname,email,age from student where id=#{stuid} //可以省略as  
</**select**>

# 第四章 MyBatis的动态SQL

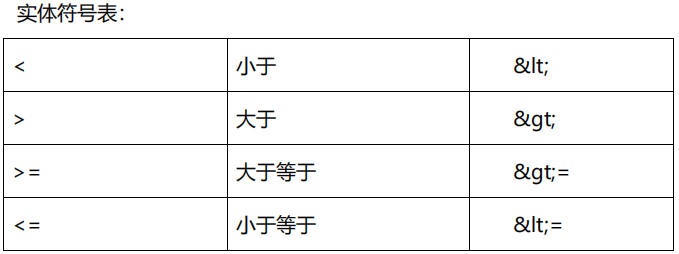
## 一、动态sql语句概述

动态SQL，通过MyBatis提供的各种标签对条件作出判断以实现动态拼接SQL语句。这里的条件判断使用的表达式为OGNL表达式。常用的动态 SQL标签有<if>、<where>、<choose/>、<foreach>等。MyBatis 的动态 SQL 语句，与 JSTL 中的语句非常相似。

动态SQL，主要用于解决查询条件不确定的情况：在程序运行期间，根据用户提交的查询条件进行查询。提交的查询条件不同，执行的 SQL 语句不同。若将每种可能的情况均逐一列出，对所有条件进行排列组合，将会出现大量的SQL 语句。此时，可使用动态 SQL 来解决这样的问题。

动态sql：同一个dao的方法，根据不同的条件可以表示不同的sql语句，主要是where部分有变化。使用mybatis提供的标签，实现动态sql的能力，主要讲if，where，foreach，sql。使用动态sql的时候，dao方法的形参使用java对象。

在mapper的动态SQL中若出现大于号（>）、小于号（<）、大于等于号（>=），小于等于号（<=）等符号，最好将其转换为实体符号。否则，XML可能会出现解析出错问题。特别是对于小于号（<），在XML中是绝不能出现的。否则解析mapper文件会出错。



## 二、动态sql之if标签

根据实体类的不同取值，使用不同的SQ语句来进行查询。比如在id如果不为空时可以根据id查询，如果username不同空时还要加入用户名作为条件。这种情况在多条件组合查询中经常会碰到。

对于该标签的执行，当test的值为true时，会将其包含的SQL片断拼接到其所在的SQL语句中。

语法：<if test=”条件”> sql语句的部分 </if>

**DAO接口方法**：

List<Student> selectIf(Student student);

**Map.xml文件:**

*<!--if  
 test: 使用对象的属性值作为条件  
-->*<**select id="selectIf" resultType="com.bjpowernode.domain.Student"**>  
  
 **select \* from** student  
 where  
 <**if test="name !=null and name!=''"**>  
 name = #{name}  
 </**if**>  
 <**if test="age >0"**>  
 or age **&lt;** #{age} //存在的弊端是若name不满足，age前有or会导致sql执行异常  
 </**if**>  
</**select**>

## 三、动态sql之where标签

<if/>标签的中存在一个比较麻烦的地方：需要在 where 后手工添加 1=1的子句。因为，若 where 后的所有<if/>条件均为 false，而 where 后若又没有 1=1 子句，则 SQL 中就会只剩下一个空的 where，SQL 出错。所以，在where 后，需要添加永为真子句 1=1，以防止这种情况的发生。但当数据量很大时，会严重影响查询效率。

使用<where/>标签，在有查询条件时，可以自动添加上 where 子句；没有查询条件时，不会添加 where 子句。需要注意的是，第一个<if/>标签中的SQL 片断，可以不包含 and。不过，写上 and 也不错，系统会将多出的 and去掉。但其它<if/>中 SQL 片断的 and，必须要求写上。否则 SQL 语句将拼接出错。

使用时 where ,里面是一个或多个if 标签， 当有一个if标签 判断条件为true， where标签会转为 WHERE 关键字附加到sql语句的后面。如果if没有一个条件为true ，忽略where和里面的if。where标签会删除和它最近的or或者and。

语法：<where> 其他动态 sql </where>

**DAO接口方法**：

List<Student> selectWhere(Student student);

**Map.xml文件:**

*<!--if  
 test: 使用对象的属性值作为条件  
-->*<**select id="selectIf" resultType="com.bjpowernode.domain.Student"**>  
  
 **select \* from** student  
 **<where>**  
 <**if test="name !=null and name!=''"**>  
 or name = #{name} //where标签会删除和它最近的or或者and，所以每个if内写上or或and也不会报错  
 </**if**>  
 <**if test="age >0"**>  
 or age **&lt;** #{age}   
 </**if**>

**</where>**  
</**select**>

## 动态sql之foreach标签

<foreach/>标签用于实现对于数组与集合的遍历。对其使用，需要注意：collection 表示要遍历的集合类型, list ，array等。open、close、separator 为对遍历内容的 SQL 拼接。

**语法：**

< **foreach collection="集合类型" open="开始的字符" close="结束的字符"  
 item="集合中的成员" separator="集合成员之间的分隔符"**>  
 #{item 的值}  
</**foreach**>  
  
标签属性：  
collection： 表示循环的对象是数组还是list集合。

如果dao接口方法的形参是数组， collection = "array"

如果dao接口方法的形参是List， collection = "list"  
  
open：循环开始时的字符。 sql.append("(");  
close：循环结束时字符。 sql.append(")");  
item：集合成员， 自定义的变量。 Integer item = idlist.get(i); //item是集合成员  
separator：集合成员之间的分隔符。 sql.append(","); //集合成员之间的分隔符  
#{item 的值}：获取集合成员的值。

### 遍历List<简单类型>

**DAO接口方法**：

List<Student> selectForeachOne(List<Integer> idlist);

**Map.xml文件:**

*<!--  
 foreach第一种方式， 循环简单类型的List  
-->*<**select id="selectForeachOne" resultType="com.bjpowernode.domain.Student"**>  
 select *\** from student  
 <**if test="list !=null and list.size>0"**> //若不对list进行判断，sql执行可能会报错  
 where id in  
 <**foreach collection="list" open="(" close=")" separator="," item="myid"**> //**myid为自定义变量**  
 #{myid}  
 </**foreach**>  
 </**if**>  
</**select**>

测试：

**public void** testSelectForeachOne(){  
 *//1.获取SqlSession* SqlSession session = MyBatisUtil.*getSqlSession*();  
 *//2.获取dao的代理* StudentDao dao = session.getMapper(StudentDao.**class**);  
  
 List<Integer> idlist = **new** ArrayList<>();  
 idlist.add(1001);  
 idlist.add(1002);  
 idlist.add(1003);  
  
 List<Student> students = dao.selectForeachOne(idlist);  
 students.forEach( stu-> System.***out***.println(**"stu=="**+stu));  
 *//3.关闭SqlSession对象* session.close();  
}

### 2、遍历List<对象类型>

**AO接口方法**：

List<Student> selectForeachTwo(List<Student> studentList);

**Map.xml文件:**

*<!--  
 foreach第二种方式，循环的List<Student>  
-->*<**select id="selectForeachTwo" resultType="com.bjpowernode.domain.Student"**>  
 select *\** from student  
 <**if test="list != null and list.size > 0"**> //若不对list进行判断，sql执行可能会报错  
 where id in  
 <**foreach collection="list" open="(" close=")" separator="," item="stu"**> //**stu相当于Student**  
 #{stu.id} **//根据Student的属性id遍历**  
 </**foreach**>  
 </**if**>  
</**select**>

测试：

**public void** testSelectForeachTwo(){  
 *//1.获取SqlSession* SqlSession session = MyBatisUtil.*getSqlSession*();  
 *//2.获取dao的代理* StudentDao dao = session.getMapper(StudentDao.**class**);  
  
 List<Student> list = **new** ArrayList<>();

Student s1 = new Student();

s1.setId(1001);

Student s2 = new Student();

s2.setId(1002);

list.add(s1);

list.add(s2);  
  
 List<Student> students = dao.selectForeachTwo(list);  
 students.forEach( stu-> System.***out***.println(**"stu=="**+stu));  
 *//3.关闭SqlSession对象* session.close();  
}

## SQL代码片段抽取

<sql/>标签用于定义SQL片断，以便其它 SQL 标签复用。而其它标签使用该 SQL 片断，需要使用<include/>子标签。该<sql/>标签可以定义 SQL 语句中的任何部分，所以<include/>子标签可以放在动态 SQL 的任何位置。

1) 在mapper文件中定义 sql代码片段 <sql id="唯一字符串"> 部分sql语句 </sql>

2）在其他的位置，使用include标签引用某个代码片段

**DAO接口方法**：

List<Student> selectWhere(Student student);

**Map.xml文件:**

*<!--定义代码片段-->*<**sql id="selectStudent"**>  
 select *\** from student  
</**sql**>  
  
<**sql id="studentFieldList"**>  
 id,name,email  
</**sql**>  
  
*<!--使用代码片段1-->*<**select id="selectIf" resultType="com.bjpowernode.domain.Student"**>  
  
 <**include refid="selectStudent"** />  
 where 1=1  
 <**if test="name !=null and name!=''"**>  
 or name = #{name}  
 </**if**>  
 <**if test="age >0"**>  
 or age **&lt;** #{age}  
 </**if**>  
</**select**>  
  
*<!--使用代码片段2-->*<**select id="selectWhere" resultType="com.bjpowernode.domain.Student"**>

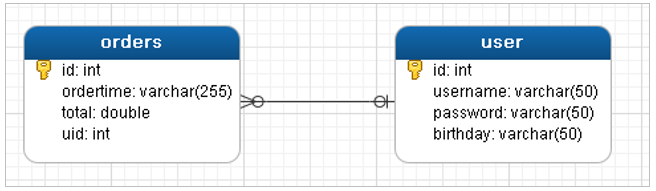
select <**include refid="studentFieldList"**/> from student  
 <**where**>  
 <**if test="name !=null and name!=''"**>  
 or name = #{name}  
 </**if**>  
 <**if test="age >0"**>  
 or age **&lt;** #{age}  
 </**if**>  
 </**where**>  
</**select**>

# 第五章 MyBatis的多表操作

## 一、一对一查询

### 一对一查询的模型

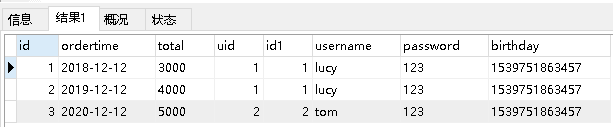
用户表和订单表的关系为，一个用户有多个订单，一个订单只从属于一个用户一对一查询的需求：查询一个订单，与此同时查询出该订单所属的用户



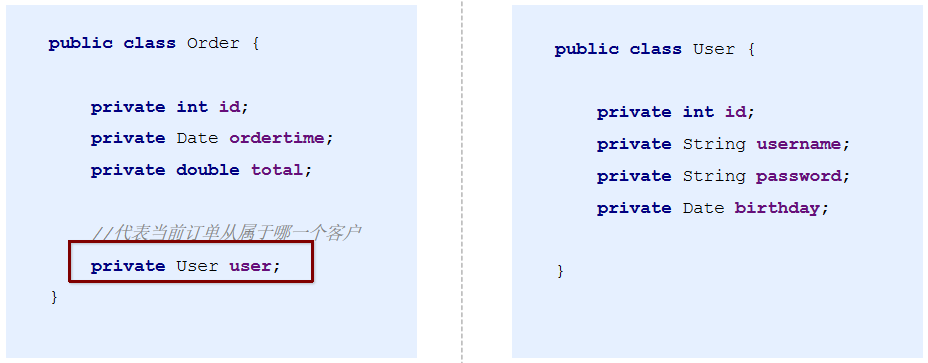
### 一对一查询的语句

对应的sql语句：select \* from orders o,user u where o.uid=u.id;

查询的结果如下：



### 创建Order和User实体



### 创建OrderMapper接口

**public interface** OrderMapper {  
  
 *//查询全部的方法* **public** List<Order> findAll();  
}

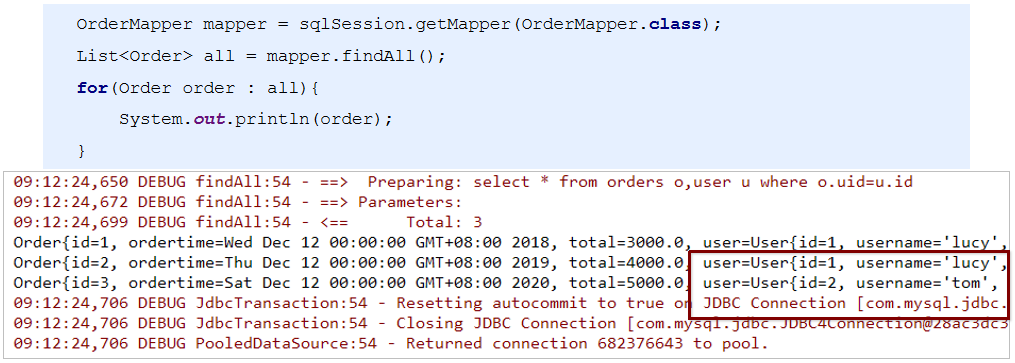
### 配置OrderMapper.xml

<**mapper namespace="com.itheima.mapper.OrderMapper"**>  
  
 <**resultMap id="orderMap" type="order"**>  
 *<!--手动指定字段与实体属性的映射关系  
 column: 数据表的字段名称  
 property：实体的属性名称  
 -->* <**id column="oid" property="id"**></**id**>  
 <**result column="ordertime" property="ordertime"**></**result**>  
 <**result column="total" property="total"**></**result**>

*<!--  
 方式一：直接采用对象.属性的方式  
 -->*  
 *<!--<result column="uid" property="user.id"></result>  
 <result column="username" property="user.username"></result>  
 <result column="password" property="user.password"></result>  
 <result column="birthday" property="user.birthday"></result>-->  
 <!--*

*方式二：先映射实体类  
 property: 当前实体(order)中的属性名称(private User user)  
 javaType: 当前实体(order)中的属性的类型(User)  
 -->* <**association property="user" javaType="user"**>  
 <**id column="uid" property="id"**></**id**>  
 <**result column="username" property="username"**></**result**>  
 <**result column="password" property="password"**></**result**>  
 <**result column="birthday" property="birthday"**></**result**>  
 </**association**>  
  
 </**resultMap**>  
  
 <**select id="findAll" resultMap="orderMap"**>  
 SELECT *\**,o.id oid FROM orders o,USER u WHERE o.uid=u.id  
 </**select**>  
  
</**mapper**>

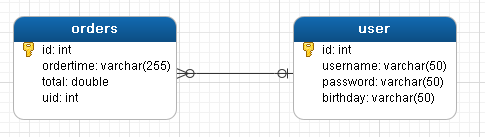
### 测试结果



## 一对多查询

### 一对多查询的模型

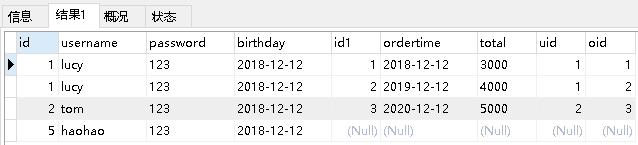
用户表和订单表的关系为，一个用户有多个订单，一个订单只从属于一个用户一对多查询的需求：查询一个用户，与此同时查询出该用户具有的订单。



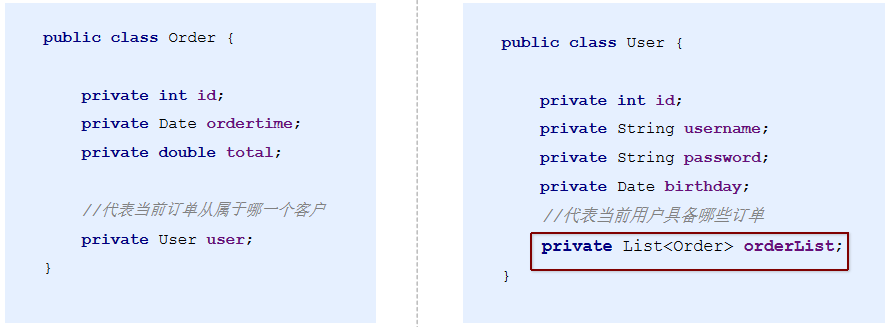
### 一对多查询的语句

对应的sql语句：select \*,o.id oid from user u left join orders o on u.id=o.uid;

查询的结果如下：



### 修改User实体



### 创建UserMapper接口

**public interface** UserMapper {  
  
 *//查询全部的方法* **public** List<User> findAll();  
}

### 配置UserMapper.xml

<**mapper namespace="com.itheima.mapper.UserMapper"**>  
  
 <**resultMap id="userMap" type="user"**>  
 <**id column="uid" property="id"**></**id**>  
 <**result column="username" property="username"**></**result**>  
 <**result column="password" property="password"**></**result**>  
 <**result column="birthday" property="birthday"**></**result**>  
 *<!--配置集合信息  
 property:集合名称  
 ofType：当前集合中的数据类型  
 -->* <**collection property="orderList" ofType="order"**>  
 *<!--封装order的数据-->* <**id column="oid" property="id"**></**id**>  
 <**result column="ordertime" property="ordertime"**></**result**>  
 <**result column="total" property="total"**></**result**>  
 </**collection**>  
 </**resultMap**>  
  
 <**select id="findAll" resultMap="userMap"**>  
 SELECT *\**,o.id oid FROM USER u,orders o WHERE u.id=o.uid  
 </**select**>  
</**mapper**>

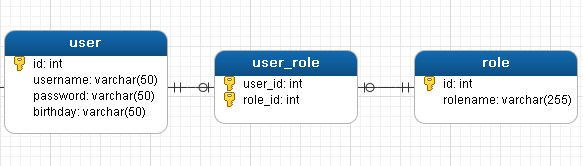
### 测试执行

**public void** test2() **throws** IOException {  
 InputStream resourceAsStream = Resources.*getResourceAsStream*(**"sqlMapConfig.xml"**);  
 SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(resourceAsStream);  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
  
 UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);  
 List<User> userList = mapper.findAll();  
 **for** (User user : userList) {  
 System.***out***.println(user);  
 }  
  
 sqlSession.close();  
}

## 多对多查询

### 多对多查询的模型

用户表和角色表的关系为，一个用户有多个角色，一个角色被多个用户使用多对多查询的需求：查询用户同时查询出该用户的所有角色。

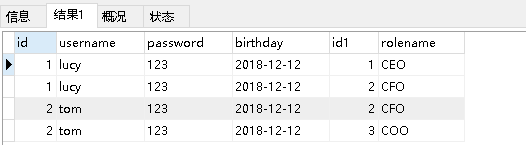


### 多对多查询的语句

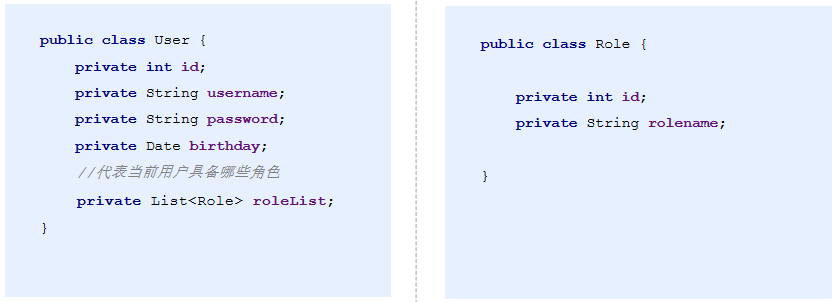
对应的sql语句：select u.\*,r.\*,r.id rid from user u left join user\_role ur on u.id=ur.user\_id

inner join role r on ur.role\_id=r.id;

查询的结果如下：



### 创建Role实体，修改User实体



### 添加UserMapper接口方法

**public interface** UserMapper {  
  
 *//查询用户及对应的角色* **public** List<User> findAllUserAndRole();  
}

### 配置UserMapper.xml

<**mapper namespace="com.itheima.mapper.UserMapper"**><**resultMap id="userRoleMap" type="user"**>  
 *<!--user的信息-->* <**id column="userId" property="id"**></**id**>  
 <**result column="username" property="username"**></**result**>  
 <**result column="password" property="password"**></**result**>  
 <**result column="birthday" property="birthday"**></**result**>  
 *<!--user内部的roleList信息-->* <**collection property="roleList" ofType="role"**>  
 <**id column="roleId" property="id"**></**id**>  
 <**result column="roleName" property="roleName"**></**result**>  
 </**collection**>  
 </**resultMap**>  
  
 <**select id="findUserAndRoleAll" resultMap="userRoleMap"**>  
 SELECT *\** FROM USER u,sys\_user\_role ur,sys\_role r WHERE u.id=ur.userId AND ur.roleId=r.id  
 </**select**>  
</**mapper**>

### 测试执行

**public void** test3() **throws** IOException {  
 InputStream resourceAsStream = Resources.*getResourceAsStream*(**"sqlMapConfig.xml"**);  
 SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(resourceAsStream);  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
  
 UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);  
 List<User> userAndRoleAll = mapper.findUserAndRoleAll();  
 **for** (User user : userAndRoleAll) {  
 System.***out***.println(user);  
 }  
 sqlSession.close();  
}

# 第六章 MyBatis注解开发



**public interface** UserMapper {  
  
 @Insert(**"insert into user values(#{id},#{username},#{password},#{birthday})"**)  
 **public void** save(User user);  
  
 @Update(**"update user set username=#{username},password=#{password} where id=#{id}"**)  
 **public void** update(User user);  
  
 @Delete(**"delete from user where id=#{id}"**)  
 **public void** delete(**int** id);  
  
 @Select(**"select \* from user where id=#{id}"**)  
 **public** User findById(**int** id);  
  
 @Select(**"select \* from user"**)  
 **public** List<User> findAll();  
  
 @Select(**"select \* from user"**)  
 @Results({  
 @Result(id=**true** ,column = **"id"**,property = **"id"**),  
 @Result(column = **"username"**,property = **"username"**),  
 @Result(column = **"password"**,property = **"password"**),  
 @Result(  
 property = **"orderList"**, *//要封装的属性名称* column = **"id"**, *//根据那个字段去查询user表的数据* javaType = List.**class**, *//要封装的实体类型  
 //select属性 代表查询那个接口的方法获得数据* many = @Many(select = **"com.itheima.mapper.OrderMapper.findByUid"**)  
 )  
 })  
 **public** List<User> findUserAndOrderAll();  
  
 @Select(**"SELECT \* FROM USER"**)  
 @Results({  
 @Result(id = **true**,column = **"id"**,property = **"id"**),  
 @Result(column = **"username"**,property = **"username"**),  
 @Result(column = **"password"**,property = **"password"**),  
 @Result(  
 property = **"roleList"**,  
 column = **"id"**,  
 javaType = List.**class**,  
 many = @Many(select = **"com.itheima.mapper.RoleMapper.findByUid"**)  
 )  
 })  
 **public** List<User> findUserAndRoleAll();  
}

# 第七章 MyBatis-Plus入门操作

## 一、MyBatis-Plus的简介

### 简介

MyBatis-Plus (opens new window)（简称 MP）是一个 MyBatis (opens new window) 的增强工具，在 MyBatis 的基础上只做增强不做改变，为简化开发、提高效率而生

### 特性

1）无侵入：只做增强不做改变，引入它不会对现有工程产生影响，如丝般顺滑

2）损耗小：启动即会自动注入基本 CURD，性能基本无损耗，直接面向对象操作

3）强大的 CRUD 操作：内置通用 Mapper、通用 Service，仅仅通过少量配置即可实现单表大部分 CRUD 操作，更有强大的条件 构造器，满足各类使用需求

4）支持 Lambda 形式调用：通过 Lambda 表达式，方便的编写各类查询条件，无需再担心字段写错

5）支持主键自动生成：支持多达 4 种主键策略（内含分布式唯一 ID 生成器 - Sequence），可自由配置，完美解决主键问题

6）支持 ActiveRecord 模式：支持 ActiveRecord 形式调用，实体类只需继承 Model 类即可进行强大的 CRUD 操作

7）支持自定义全局通用操作：支持全局通用方法注入（ Write once, use anywhere ）

8）内置代码生成器：采用代码或者 Maven 插件可快速生成 Mapper 、 Model 、 Service 、 Controller 层代码，支持模板引 擎，更有超多自定义配置等您来使用

9）内置分页插件：基于 MyBatis 物理分页，开发者无需关心具体操作，配置好插件之后，写分页等同于普通 List 查询

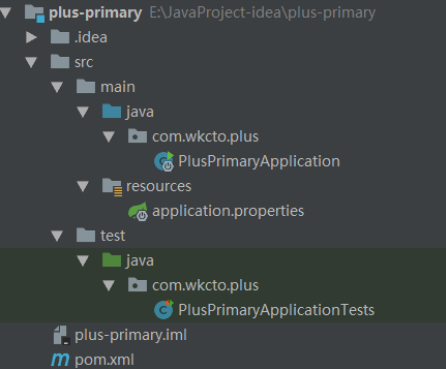
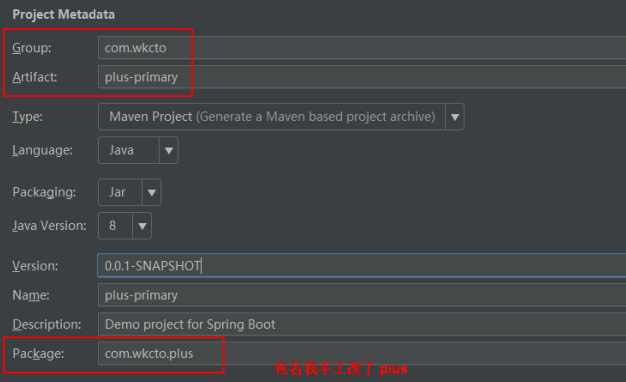
10）分页插件支持多种数据库：支持 MySQL、MariaDB、Oracle、DB2、H2、HSQL、SQLite、Postgre、SQLServer等多种数据库

11）内置性能分析插件：可输出 SQL 语句以及其执行时间，建议开发测试时启用该功能，能快速揪出慢查询

12）内置全局拦截插件：提供全表 delete 、 update 操作智能分析阻断，也可自定义拦截规则，预防误操作

## 二、MyBatis-Plus的快速入门

### 新建SpringBoot工程



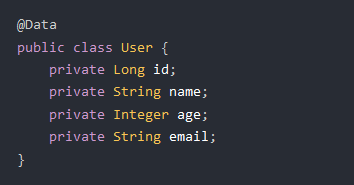
### pom.xml中引入依赖

<**parent**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-parent</**artifactId**>  
 <**version**>2.1.0.RELEASE</**version**>  
 <**relativePath**/> *<!-- lookup parent from repository -->*</**parent**>  
  
<**properties**>  
 <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  
 <**project.reporting.outputEncoding**>UTF-8</**project.reporting.outputEncoding**>  
 <**java.version**>1.8</**java.version**>  
</**properties**>  
  
<**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter</**artifactId**>  
 </**dependency**>  
  
 *<!-- mybatis-plus -->* <**dependency**>  
 <**groupId**>com.baomidou</**groupId**>  
 <**artifactId**>mybatis-plus-boot-starter</**artifactId**>  
 <**version**>3.3.1</**version**>  
 </**dependency**>  
  
 *<!-- 模板引擎 -->* <**dependency**>  
 <**groupId**>org.apache.velocity</**groupId**>  
 <**artifactId**>velocity-engine-core</**artifactId**>  
 <**version**>2.0</**version**>  
 </**dependency**>  
  
 <**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 <**scope**>runtime</**scope**>  
 <**version**>5.1.6</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-test</**artifactId**>  
 <**scope**>test</**scope**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.projectlombok</**groupId**>  
 <**artifactId**>lombok</**artifactId**>  
 <**version**>1.18.10</**version**>  
 </**dependency**>  
</**dependencies**>  
  
<**build**>  
 <**plugins**>  
 <**plugin**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-maven-plugin</**artifactId**>  
 </**plugin**>  
 </**plugins**>  
</**build**>

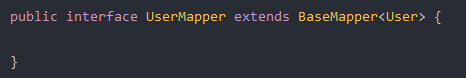
### application.yml中添加配置

**spring:  
 datasource:  
 driver-class-name:** com.mysql.jdbc.Driver  
 **url:** jdbc:mysql://127.0.0.1:3306/mybatis?useUnicode=true&characterEncoding=utf-8  
 **username:** root  
 **password:** 123456

### 实体类User.java



### 编写Mapper接口UserMapper.java



### 启动类上添加@MapperScan注解

@SpringBootApplication  
@MapperScan(value = **"com.wkcto.plus.mapper"**)**public class** PlusApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(PlusApplication.**class**, args);  
 }  
}

### 测试

@SuppressWarnings(**"all"**)  
@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public class** PlusApplicationTests {  
  
 *//使用自动注入， 注入Mapper对象(Dao)* @Autowired  
 **private** UserMapper **userMapper**;  
 *//测试添加操作， insert* @Test  
 **public void** testUserInsert(){  
 **for**(**int** i=0;i<10;i++){

*//创建User对象*  
 User user = **new** User();  
 user.setName(**"zhangsan"**+i);  
 user.setAge(20 + i);  
 user.setEmail(**"zhangsan@sina.com"**);

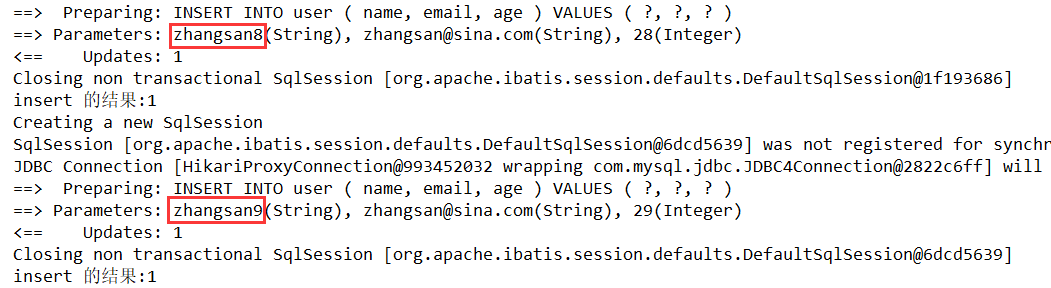
*//调用UserMapper的方法， 也就是父接口BaseMapper中的提供的方法* **int** rows = **userMapper**.insert(user);  
 System.***out***.println(**"insert的结果:"**+rows);  
 }  
 }

}

### 配置mybatis日志

**# application.yml文件中**

**mybatis-plus:  
 configuration:  
 log-impl:** org.apache.ibatis.logging.stdout.StdOutImpl



# 第八章 MyBatis-Plus的增删改查操作

## 一、Service CRUD 接口

### 简介

说明:

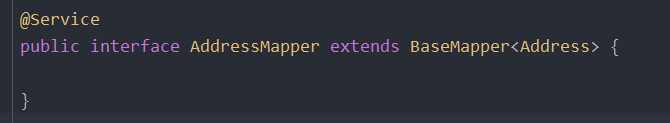
通用Service CRUD封装IService(opens new window)接口，进一步封装CRUD采用get查询单行remove删除list查询集合page分页前缀命名方式区分Mapper层避免混淆，

泛型 T 为任意实体对象

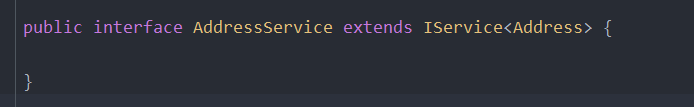
建议如果存在自定义通用Service方法的可能，请创建自己的IBaseService继承Mybatis-Plus提供的基类

对象Wrapper为条件构造器

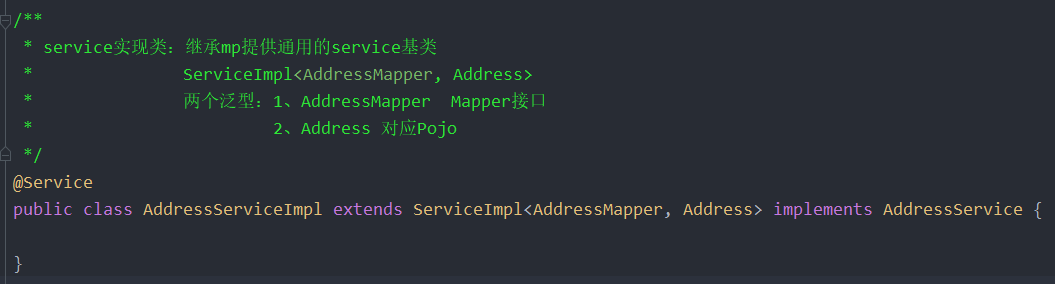
1. 定义mapper接口



1. 定义service接口



1. 定义service接口的实现类



### Save新增

@Autowired  
**private** AddressService **addressService**;  
  
*/\*\*  
 \* 插入一条记录（选择字段，策略插入）  
 \* boolean save(T entity);  
 \* 插入（批量）  
 \* boolean saveBatch(Collection<T> entityList);  
 \* 插入（批量）,batchSize为插入批次数量  
 \* boolean saveBatch(Collection<T> entityList, int batchSize);  
 \*/*@Test  
**public void** testSave(){  
  
 Address address = **new** Address();  
 address.setCity(**"成都"**);  
 address.setStreet(**"1111111111"**);  
 **addressService**.save(address);  
}  
  
@Test  
**public void** testSaveBatch(){  
  
 Address address = **new** Address();  
 address.setCity(**"成都"**);  
 address.setStreet(**"1111111111"**);  
  
 Address address1 = **new** Address();  
 address1.setCity(**"上海"**);  
 address1.setStreet(**"22222222"**);  
  
 ArrayList<Address> list = **new** ArrayList<>();  
 list.add(address);  
 list.add(address1);  
  
 **addressService**.saveBatch(list,3);  
}

### SaveOrUpdate新增或修改

*/\*\*  
 \* TableId注解存在更新记录，否则插入一条记录  
 \* boolean saveOrUpdate(T entity);  
 \* 根据updateWrapper尝试更新，否继续执行saveOrUpdate(T)方法  
 \* boolean saveOrUpdate(T entity, Wrapper<T> updateWrapper);  
 \* 批量修改插入  
 \* boolean saveOrUpdateBatch(Collection<T> entityList);  
 \* 批量修改插入,batchSize为插入批次数量  
 \* boolean saveOrUpdateBatch(Collection<T> entityList, int batchSize);  
 \*/*@Test  
**public void** testSaveOrUpdate(){  
  
 Address address = **new** Address();  
 address.setCity(**"成都"**);  
 address.setStreet(**"1111111111"**);  
 **addressService**.saveOrUpdate(address);  
}  
  
@Test  
**public void** testSaveOrUpdateBatch(){  
  
 Address address = **new** Address();  
 address.setCity(**"北京"**);  
 address.setStreet(**"77777777"**);  
  
 Address address1 = **new** Address();  
 address1.setId(6);  
*// Address address1 = addressService.getById(6);* address1.setCity(**"北京"**);  
 address1.setStreet(**"77777777"**);  
  
 ArrayList<Address> list = **new** ArrayList<>();  
 list.add(address);  
 list.add(address1);  
 **addressService**.saveOrUpdateBatch(list);  
}

### Update修改



### Remove删除



### Get查询

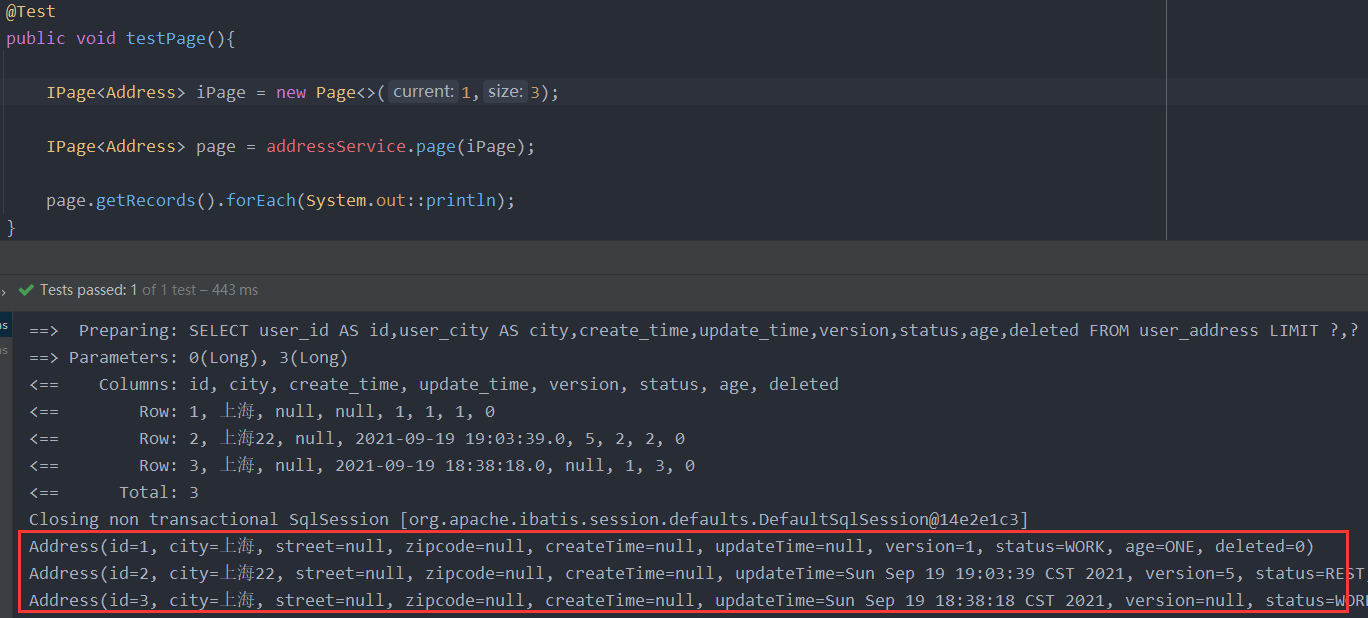


### List分页查询

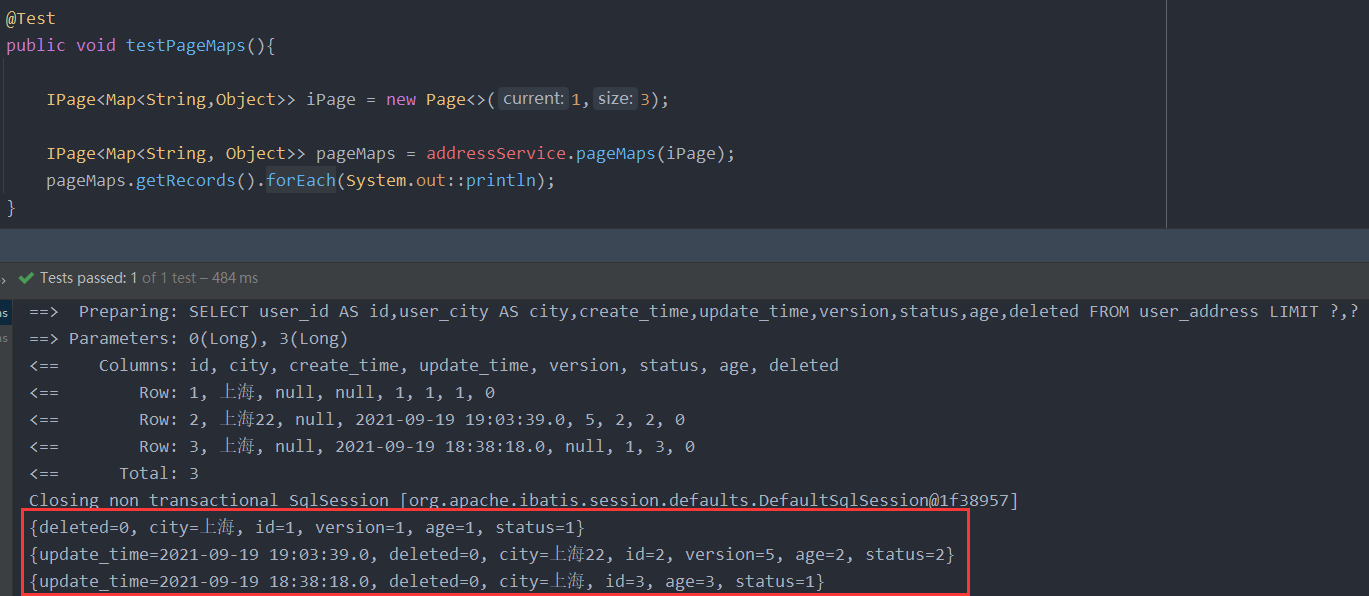


### Page分页查询









### Count查询数量



### Chain链式操作



## 二、Mapper CRUD接口

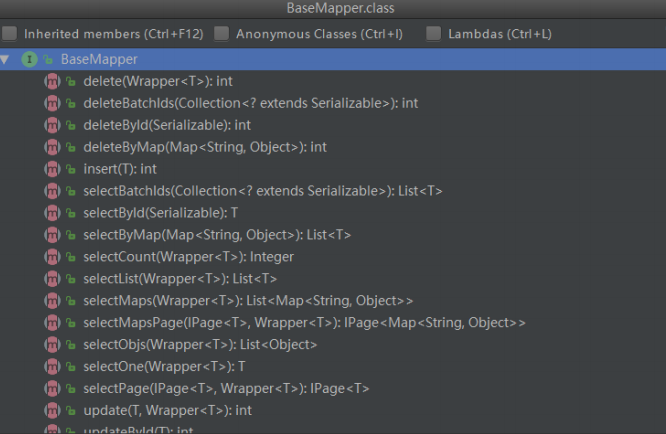
说明:

通用 CRUD 封装BaseMapper(opens new window)接口，为 Mybatis-Plus 启动时自动解析实体表关系映射转换为 Mybatis 内部对象注入容器。BaseMapper中共有17个方法

泛型 T 为任意实体对象

参数 Serializable 为任意类型主键 Mybatis-Plus 不推荐使用复合主键约定每一张表都有自己的唯一 id 主键

对象 Wrapper 为 条件构造器



### insert操作

@SuppressWarnings(**"all"**)  
@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public class** PlusApplicationTests {  
  
 *//使用自动注入，注入Mapper对象(Dao)* @Autowired  
 **private** UserMapper **userDao**;  
 *//测试添加操作， insert* @Test  
 **public void** testUserInsert(){  
 *//创建User对象* **for**(**int** i=0;i<10;i++){  
 User user = **new** User();  
 user.setName(**"zhangsan"**+i);  
 user.setAge(20 + i);  
 user.setEmail(**"zhangsan@sina.com"**);  
 *//调用UserMapper的方法，也就是父接口BaseMapper中的提供的方法* **int** rows = **userDao**.insert(user); *//返回数据插入成功的行数*  
 System.***out***.println(**"insert 的结果:"**+rows);  
 }  
 }  
  
 *//添加数据后，获取主键值* @Test  
 **public void** testInsertGetId(){  
 User user = **new** User();  
 user.setName(**"李四"**);  
 user.setAge(20);  
 user.setEmail(**"lisi@163.com"**);  
  
 **int** rows = **userDao**.insert(user);  
 System.***out***.println(**"insert user rows:"**+rows);  
  
 *//获取主键id ，刚添加数据库中的数据的id* **int** id = user.getId(); *//主键字段对应的get方法* System.***out***.println(**"主键id:"**+id);  
 }  
}

### Update操作

@SuppressWarnings(**"all"**)  
@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public class** PlusApplicationTests {  
  
 *//使用自动注入，注入Mapper对象(Dao)* @Autowired  
 **private** UserMapper **userDao**;  
 */\*\*  
 \* 更新操作update  
 \*/* @Test  
 **public void** testUpdateUser(){  
 User user = **new** User();  
 user.setName(**"修改的数据"**);  
 user.setAge(22);  
 user.setEmail(**"edit@163.com"**);  
 user.setId(2);  
 *//执行更新，根据主键值更新  
 /\*UPDATE user SET name=?, email=?, age=? WHERE id=?  
 \*更新了所有非null属性值， 条件where id = 主键值  
 \*/* **int** rows = **userDao**.updateById(user);  
 System.***out***.println(**"update rows:"**+rows);  
 }  
 */\*\*  
 \* 控制更新的属性  
 \*/* @Test  
 **public void** testUpdateUser2(){  
 User user = **new** User();  
 user.setId(2);  
 user.setName(**"zhangsan"**);  
 *//更新数据  
 //UPDATE user SET name=? WHERE id=?* **int** i = **userDao**.updateById(user);  
 System.***out***.println(**"i:"**+i);  
 }  
 */\*\*  
 \* 更新数据：实体类的属性是基本类型 - int age  
 \*/* @Test  
 **public void** testUpdateUser3(){  
 User user = **new** User();  
 user.setId(3);  
 user.setEmail(**"lisi@sina.com"**); *//判断字段是否要修改，加入到set语句， 是根据属性值是否为null*

*//实体对象 user: [name = null , email = "lisi@sina.com" , age = 0 ]* ***//name没有赋值，是null，没有被修改。age有默认值0，被更新了*** *//UPDATE user SET email=?, age=? WHERE id=?* **int** rows = **userDao**.updateById(user);  
 System.***out***.println(**"rows："**+rows);  
 }  
}

### Delete操作

@SuppressWarnings(**"all"**)  
@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public class** PlusApplicationTests {  
  
 *//使用自动注入，注入Mapper对象(Dao)* @Autowired  
 **private** UserMapper **userDao**; */\*\*  
 \* 1、按主键删除一条数据  
 \* 方法是deleteById()  
 \* 参数：主键值  
 \* 返回值：是删除的成功记录数  
 \*/* @Test  
 **public void** testDeleteById(){  
  
 *//DELETE FROM user WHERE id=?* **int** rows = **userDao**.deleteById(3);  
 System.***out***.println(**"deleteById:"**+rows);  
 }  
 */\*\*  
 \* 2、按条件删除数据， 条件是封装到Map对象中  
 \* 方法：deleteByMap(map对象);  
 \* 返回值：删除成功的记录数  
 \*/* @Test  
 **public void** testDeleteByMap(){  
 *//创建Map对象，保存条件值* Map<String,Object> map = **new** HashMap<>();  
 *//put("表的列名",条件值) ， 可以封装多个条件* map.put(**"name"**,**"zs"**);  
 map.put(**"age"**,20);  
 *//调用删除方法  
 //DELETE FROM user WHERE name = ? AND age = ?* **int** rows = **userDao**.deleteByMap(map);  
 System.***out***.println(**"deleteByMap rows:"**+rows);  
 }  
 */\*\*  
 \* 3、批处理方式：使用多个主键值，删除数据  
 \* 方法名称：deleteBatchIds()  
 \* 参数： Collection*<? extends Serializable> *var1  
 \* 返回值：删除的记录数  
 \*/* @Test  
 **public void** deleteByBatchId(){

*/\* List<Integer> ids = new ArrayList<>();  
 ids.add(1);  
 ids.add(2);  
 ids.add(3);  
 ids.add(4);  
 ids.add(5);\*/  
  
 //使用lambda创建List集合* List<Integer> ids = Stream.*of*(1, 2, 3, 4, 5).collect(Collectors.*toList*());  
 *//删除操作  
 //DELETE FROM user WHERE id IN ( ? , ? , ? , ? , ? )* **int** i = **userDao**.deleteBatchIds(ids);  
 System.***out***.println(**"deleteBatchIds:"**+i);  
 }  
}

### Select操作

@SuppressWarnings(**"all"**)  
@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public class** PlusApplicationTests {  
  
 *//使用自动注入，注入Mapper对象(Dao)* @Autowired  
 **private** UserMapper **userDao**;  
 */\*\*  
 \* 1、实现查询 selectById,根据主键值查询  
 \* 参数：主键值：  
 \* 返回值： 实体对象(唯一的一个对象)  
 \*/* @Test  
 **public void** testSelectById(){  
 */\*\*  
 \* 生成的sql: SELECT id,name,email,age FROM user WHERE id=?  
 \* 如果根据主键没有查找到数据，得到的返回值是 null  
 \*/* User user = **userDao**.selectById(6);  
 System.***out***.println(**"selectById:"**+user);  
  
 *//在使用对象之前，需要判断对象是否为null* **if**(user != **null**){  
 *//业务方法的调用* }  
 }  
 */\*\*  
 \* 2、实现批处理查询，根据多个主键值查询，获取到List  
 \* 方法：selectBatchIds  
 \* 参数：id的集合  
 \* 返回值：List<T>  
 \*/* @Test  
 **public void** testSelectBatchId(){  
 List<Integer> ids = **new** ArrayList<>();  
 ids.add(6);  
 ids.add(9);  
 ids.add(10);  
  
 *//查询数据  
 //SELECT id,name,email,age FROM user WHERE id IN ( ? , ? , ? )* List<User> users = **userDao**.selectBatchIds(ids);  
 System.***out***.println(**"size:"**+users.size());  
 **for** (User u:users){  
 System.***out***.println(**"查询的用户："**+u);  
 }  
 }  
  
 */\*\*  
 \* 使用lambda查询数据  
 \*/* @Test  
 **public void** testSelectBatchId2(){  
 List<Integer> ids = Stream.*of*(6, 9, 10, 15).collect(Collectors.*toList*());  
 *//SELECT id,name,email,age FROM user WHERE id IN ( ? , ? , ? , ? )* List<User> users = **userDao**.selectBatchIds(ids);  
 *//遍历集合* users.forEach( u -> {  
 System.***out***.println(**"查询的user对象："**+u);  
 });  
 }  
  
 */\*\*  
 \* 3、使用Map做多条件查询  
 \* 方法：selectByMap()  
 \* 参数：Map<String,Object>  
 \* 返回值：List<T>  
 \*  
 \*/* @Test  
 **public void** testSelectMap(){

*//创建Map,封装查询条件* Map<String,Object> map = **new** HashMap<>();  
 *//key是数据库列名， value：字段值 ，多个key，是and 联接* map.put(**"name"**,**"zhangsan"**);  
 map.put(**"age"**,20);  
 *//根据Map查询  
 //SELECT id,name,email,age FROM user WHERE name = ? AND age = ?* List<User> users = **userDao**.selectByMap(map);  
 users.forEach(System.***out***::println);  
 }  
}

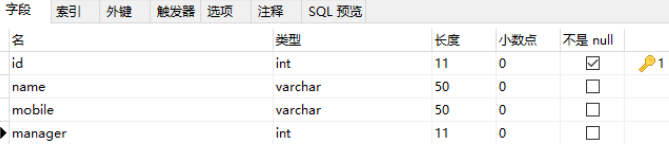
## 三、ActiveRecord（AR）

ActiveRecord介绍：

1. 每一个数据库表对应创建一个类，类的每一个对象实例对应于数据库中标的一行记录；通常标的每个字段在类中都有相应的Field
2. ActiveRecord负责把自己持久化，在ActiveRecord中封装了对数据库的访问，通过对象自己实现CRUD，实现优雅的数据库操作
3. ActiveRecord也封装了部分业务逻辑，可以作为业务对象使用

### insert操作

1. dept表设计



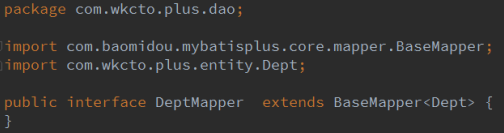
1. Entity实体类

*/\*\*  
 \* 使用AR,要求实体类需要继承MP中的Model  
 \* Model中提供了对数据库的CRUD的操作  
 \*/*@Data  
**public class** Dept **extends** Model<Dept> {

*//定义属性， 属性名和表的列名一样  
 //UUID* @TableId(value = **"id"**,type = IdType.***UUID***)  
 **private** String **id**;  
 **private** String **name**;  
 **private** String **mobile**;  
 **private** Integer **manager**;  
}

1. Mapper

不使用mapper，也需要定义这个类，MP通过mapper获取到表的结构；不定义时，MP报错无法获取表的结构信息



1. 测试

@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public class** DeptARTest {  
  
 @Test  
 **public void** testARInsert(){  
 *//定义dept的实体* Dept dept = **new** Dept();  
 dept.setName(**"市场部"**);  
 dept.setMobile(**"010-66666666"**);  
 dept.setManager(5);  
 *//调用实体对象自己的方法，完成对象自身到数据库的添加操作* **boolean** flag = dept.insert();  
 System.***out***.println(**"ar insert result:"**+flag);  
 }  
}

### update操作

@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public class** DeptARTest {  
  
 @Test  
 **public void** testARUpdate(){  
 *//定义实体Dept* Dept dept = **new** Dept();  
 dept.setId(2);dept.setMobile(**"010-22222222"**);  
 dept.setName(**"改为市场部"**);  
 dept.setManager(2);  
 *//根据主键id更新记录  
 // UPDATE dept SET name=?, mobile=?, manager=? WHERE id=? // id = 1* **boolean** result = dept.updateById(); *//使用dept实体主键的值，作为where id = 1* System.***out***.println(**"ar updateById result:"**+result);  
 }  
  
 @Test  
 **public void** testARUpdate2(){  
 *//定义实体Dept* Dept dept = **new** Dept();  
dept.setId("1");dept.setMobile(**"010-3333333"**);  
*// dept.setManager(2);  
// dept.setName("zhangsan");  
 //name , manager是没有修改的  
  
 //根据主键id更新记录  
 // UPDATE dept SET name=?, mobile=?, manager=? WHERE id=? // id = 1* ***// null的属性值不做更新处理，在update中没有null的字段*** *//UPDATE dept SET mobile=? WHERE id=?* **boolean** result = dept.updateById();*//使用dept实体主键的值，作为where id = 1* System.***out***.println(**"ar updateById result:"**+result);  
 }  
}

### delete操作

@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public class** DeptARTest {  
  
 */\*\*  
 \* deleteById()删除操作即使没有从数据库中删除数据，也返回是true。只要sql语法是正确的，返回就是true，和删除的记录无关  
 \*/* @Test  
 **public void** testARDeleteById(){  
  
 Dept dept = **new** Dept();  
 *//DELETE FROM dept WHERE id=?* **boolean** result = dept.deleteById(1);  
 System.***out***.println(**"ar deleteById result:"**+result);  
 }  
  
 @Test  
 **public void** testARDeleteById2(){  
 Dept dept = **new** Dept();  
 **dept.setId("2");** *//DELETE FROM dept WHERE id=?* **boolean** result = dept.deleteById();  
 System.***out***.println(**"ar deleteById result:"**+result);  
 }  
}

### select操作

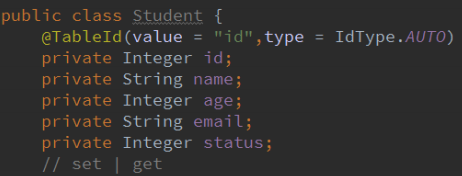
@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public class** DeptARTest {  
  
 */\*\*  
 \* selectByID  
 \* 1.按实体的主键能查找出数据，返回对象  
 \* 2.按实体的主键不能查出数据，是null ，不报错。  
 \*/* @Test  
 **public void** testARSelectById(){  
  
 Dept dept = **new** Dept();  
 *//设置主键的值* dept.setId("1"); *//调用查询方法  
 //SELECT id,name,mobile,manager FROM dept WHERE id=?* Dept dept1 = dept.selectById();  
 System.***out***.println(**"ar selectById result:"**+dept1);  
 }  
  
 */\*\*  
 \* selectById(主键)  
 \* 1.主键有记录，返回实体对象  
 \* 2.主键没有记录，返回是null，不提供主键id值，报错  
 \*/* @Test  
 **public void** testARSelectById2(){  
  
 Dept dept = **new** Dept();  
 Dept dept1 = dept.selectById(3);  
 System.***out***.println(**"dept1:"**+dept1);  
 }  
}

## 四、自定义sql

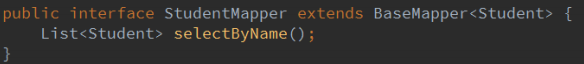
### 定义表



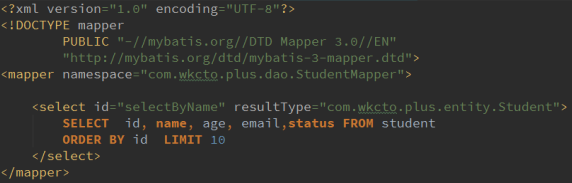
### 创建实体类



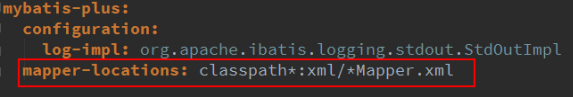
### 创建mapper



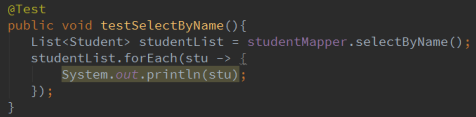
### 新建sql映射xml文件

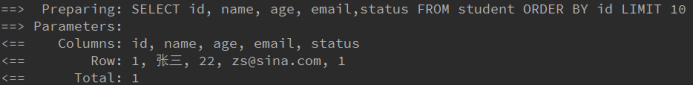


### 配置xml文件位置



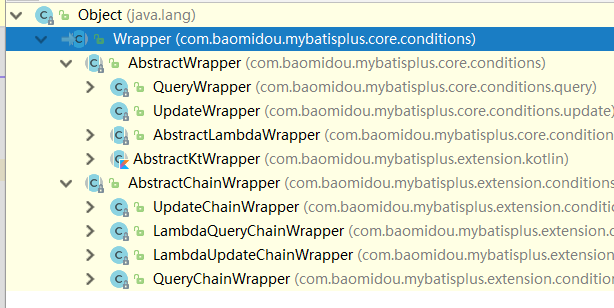
### 测试





# 第九章 MyBatis-Plus的查询与分页

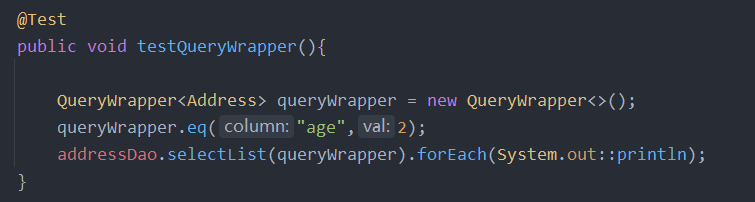
## 一、查询构造器：Wrapper



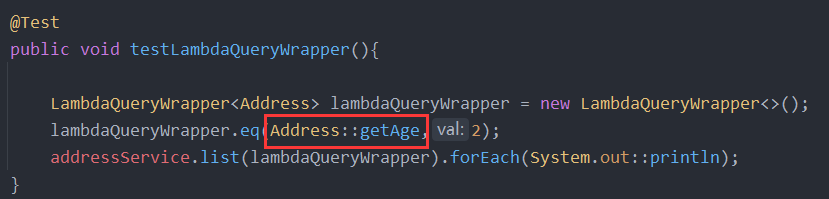
QueryWrapper（LamdbaQueryWrapper）和UpdateWrapper（LamdbaUpdateWrapper）的父类用于生成sql的where条件，entity属性也用于生成sql的where条件，MP3.X开始支持lamdba表达式。

## 二、LamdbaQueryWrapper

1、使用QueryWrapper，需要硬编码数据库的列名age



2、使用LamdbaQueryWrapper



## 三、查询

### allEq

以Map为参数条件

@SuppressWarnings(**"all"**)  
@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public class** StudentTest {  
  
 *//定义StudentMapper* @Autowired  
 **private** StudentMapper **studentDao**;  
  
 @Test  
 **public void** testAllEq() {  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 *//组装条件* Map<String, Object> param = **new** HashMap<>();  
 *//map<key,value> key列名 , value：查询的值* param.put(**"name"**, **"张三"**);  
 param.put(**"age"**, 22);  
 param.put(**"status"**, 1);  
  
 qw.allEq(param);  
 *//调用MP自己的查询方法  
 //SELECT id,name,age,email,status FROM student WHERE name = ? AND age = ?  
 //WHERE name = ? AND age = ? AND status = ?* List<Student> students = **studentDao**.selectList(qw);  
 students.forEach(stu -> System.***out***.println(stu));  
 }  
 */\*\*  
 \* 1) Map对象中有 key的value是null  
 \* 使用的是 qw.allEq(param,true);  
 \* 结果：WHERE name = ? AND age IS NULL  
 \* <p>  
 \* 2) Map对象中有 key的value是null  
 \* qw.allEq(param,false);  
 \* 结果：WHERE name = ?  
 \* <p>  
 \* 结论：  
 \* allEq(map,boolean)  
 \* true:处理null值，where 条件加入 字段 is null  
 \* false:忽略null ，不作为where 条件  
 \*/* @Test  
 **public void** testAllEq2() {  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 *//组装条件* Map<String, Object> param = **new** HashMap<>();  
 *//map<key,value> key列名 , value：查询的值* param.put(**"name"**, **"张三"**);  
 *//age 是 null* param.put(**"age"**, **null**);  
  
 *//allEq第二个参数为true* qw.allEq(param, **false**);  
  
 *//调用MP自己的查询方法* List<Student> students = **studentDao**.selectList(qw);  
 students.forEach(stu -> System.***out***.println(stu));  
 }  
}

### eq等于

@SuppressWarnings(**"all"**)  
@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public class** StudentTest {  
  
 *//定义StudentMapper* @Autowired  
 **private** StudentMapper **studentDao**;  
  
 */\*\*  
 \* eq使用  
 \* eq("列名",值)  
 \*/* @Test  
 **public void** testEq() {  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 *//组成条件* qw.eq(**"name"**, **"李四"**);  
 *//WHERE name = ?* List<Student> students = **studentDao**.selectList(qw);  
 students.forEach(stu -> System.***out***.println(**"查询eq："** + stu));  
 }  
}

### ne不等于

@SuppressWarnings(**"all"**)  
@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public class** StudentTest {  
  
 *//定义StudentMapper* @Autowired  
 **private** StudentMapper **studentDao**;  
  
 */\*\*  
 \* ne使用  
 \* ne表示不等于 <>  
 \* <p>  
 \* ne("列名",值)  
 \*/* @Test  
 **public void** testNe() {  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 *//组成条件* qw.ne(**"name"**, **"张三"**);  
 *// WHERE name <> ?* List<Student> students = **studentDao**.selectList(qw);  
 students.forEach(stu -> System.***out***.println(**"查询ne:"** + stu));  
 }  
}

### gt大于

@SuppressWarnings(**"all"**)  
@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public class** StudentTest {  
  
 *//定义StudentMapper* @Autowired  
 **private** StudentMapper **studentDao**;  
  
 */\*\*  
 \* gt 大于( > )  
 \*/* @Test  
 **public void** testGt() {  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 qw.gt(**"age"**, 30); *//age > 30  
 // WHERE age > ?* List<Student> students = **studentDao**.selectList(qw);  
 students.forEach(stu -> System.***out***.println(**"stu:"** + stu));  
 }  
}

### ge大于等于

@SuppressWarnings(**"all"**)  
@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public class** StudentTest {  
  
 *//定义StudentMapper* @Autowired  
 **private** StudentMapper **studentDao**;  
  
 */\*\*  
 \* ge 大于等于 （ >=）  
 \*/* @Test  
 **public void** testGe() {  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 qw.ge(**"age"**, 31);*// >=31  
 //WHERE age >= ?* List<Student> students = **studentDao**.selectList(qw);  
 students.forEach(stu -> System.***out***.println(**"student:"** + stu));  
 }  
}

### lt小于

*/\*\*  
 \* lt 小于 ( < )  
 \*/*@Test  
**public void** testLt() {  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 qw.lt(**"age"**, 32);  
 *// WHERE age < ?* List<Student> students = **studentDao**.selectList(qw);  
 students.forEach(stu -> System.***out***.println(**"student:"** + stu));  
}

### le小于等于

*/\*\*  
 \* le 小于 ( <= )  
 \*/*@Test  
**public void** testLe() {  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 qw.le(**"age"**, 32);  
 *// WHERE age <= ?* List<Student> students = **studentDao**.selectList(qw);  
 students.forEach(stu -> System.***out***.println(**"student:"** + stu));  
}

### between在两个值之间

*/\*\*  
 \* between ( ? and ? )  
 \*/*@Test  
**public void** testBetween() {  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 *//between("列名",开始值，结束值)* qw.between(**"age"**, 22, 28);  
 *// where age >= 22 and age < 28* List<Student> students = **studentDao**.selectList(qw);  
 students.forEach(stu -> System.***out***.println(stu));  
}

### notBetween

*/\*\*  
 \* notBetween(不在范围区间内)  
 \*/*@Test  
**public void** testNotBetween() {  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 qw.notBetween(**"age"**, 18, 28);  
 *//WHERE age NOT BETWEEN ? AND ?  
 // where age < 18 or age > 28* List<Student> students = **studentDao**.selectList(qw);  
 students.forEach(stu -> System.***out***.println(stu));  
}

### like、notLike

*/\*\*  
 \* like 匹配某个值  
 \*/*@Test  
**public void** testLike() {  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 qw.like(**"name"**, **"张"**);  
 *// WHERE name LIKE %张%* List<Student> students = **studentDao**.selectList(qw);  
 students.forEach(stu -> System.***out***.println(stu));  
}  
  
*/\*\*  
 \* notLike 不匹配某个值  
 \*/*@Test  
**public void** testNotLike() {  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 qw.notLike(**"name"**, **"张"**);  
 *// WHERE name NOT LIKE ? %张%* List<Student> students = **studentDao**.selectList(qw);  
 students.forEach(stu -> System.***out***.println(stu));  
}

### likeLeft、likeRight

likeLeft：匹配like “%值”

likeRight：匹配like “值%”

*/\*\*  
 \* likeLeft "%值"  
 \*/*@Test  
**public void** testLikeLeft() {  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 qw.likeLeft(**"name"**, **"张"**);  
 *//WHERE name LIKE %张* List<Student> students = **studentDao**.selectList(qw);  
 students.forEach(student -> System.***out***.println(student));  
}  
  
*/\*\*  
 \* likeRight "%值"  
 \*/*@Test  
**public void** testLikeRight() {  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 qw.likeRight(**"name"**, **"李"**);  
 *//WHERE name LIKE 李%* List<Student> students = **studentDao**.selectList(qw);  
 students.forEach(student -> System.***out***.println(student));  
}

### isNull、isNotNull

isNull判断字段值为null

isNotNull判断字段值不为null

*/\*\*  
 \* isNull , 判断字段是 null  
 \*/*@Test  
**public void** testIsNull(){  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 *//判断email is null  
 //WHERE email IS NULL* qw.isNull(**"email"**);  
 print(qw);  
}  
  
*/\*\*  
 \* isNotNull , 判断字段是 is not null  
 \*/*@Test  
**public void** testIsNotNull(){  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 *// WHERE email IS NOT NULL* qw.isNotNull(**"email"**);  
 print(qw);  
}

### in、notIn

*/\*\*  
 \* in 值列表  
 \*/*@Test  
**public void** testIn(){  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 *//in(列名，多个值的列表)  
 //WHERE name IN (?,?,?)* qw.in(**"name"**,**"张三"**,**"李四"**,**"周丽"**);  
 print(qw);  
  
}  
  
*/\*\*  
 \* notIn 不在值列表  
 \*/*@Test  
**public void** testNoIn(){  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 *//in(列名，多个值的列表)  
 //WHERE name NOT IN (?,?,?)* qw.notIn(**"name"**,**"张三"**,**"李四"**,**"周丽"**);  
 print(qw);  
  
}

*/\*\*  
 \* in 值列表  
 \*/*@Test  
**public void** testIn2(){  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 List<Object> list = **new** ArrayList<>();  
 list.add(1);  
 list.add(2);  
 *//WHERE status IN (?,?)* qw.in(**"status"**,list);  
 print(qw);  
}

### inSql、notInSql

类似in和notIn

*/\*\*  
 \* inSql() : 使用子查询  
 \*/*@Test  
**public void** testInSQL(){  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 *//WHERE age IN (select age from student where id=1)* qw.inSql(**"age"**,**"select age from student where id=1"**);  
 print(qw);  
}  
  
  
*/\*\*  
 \* notInSql() : 使用子查询  
 \*/*@Test  
**public void** testNotInSQL(){  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 *//WHERE age NOT IN (select age from student where id=1)* qw.notInSql(**"age"**,**"select age from student where id=1"**);  
 print(qw);  
}  
  
**private void** print(QueryWrapper qw){  
 List<Student> students = **studentDao**.selectList(qw);  
 students.forEach(student -> System.***out***.println(student));  
}

### groupBy

基于多个字段分组

*/\*\*  
 \* groupBy：分组  
 \*/*@Test  
**public void** testGroupby(){  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 qw.select(**"name,count(\*) personNumbers"**);*//select name,count(\*) personNumbers* qw.groupBy(**"name"**);  
 *// SELECT name,count(\*) personNumbers FROM student GROUP BY name* print(qw);  
}

### orderByAsc、orderByDesc、orderBy

orderByAsc：按字段升序

orderByDesc：按字段降序

orderBy：每个字段指定排序方向

*/\*\*  
 \* orderbyAsc ： 按字段升序  
 \*/*@Test  
**public void** testOrderByAsc(){  
 QueryWrapper<Student> qw= **new** QueryWrapper<>();  
 *//FROM student ORDER BY name ASC , age ASC* qw.orderByAsc(**"name"**,**"age"**);  
 print(qw);  
}  
  
*/\*\*  
 \* orderbyDesc ： 按字段降序  
 \*/*@Test  
**public void** testOrderByDesc(){  
 QueryWrapper<Student> qw= **new** QueryWrapper<>();  
 *// ORDER BY name DESC , id DESC* qw.orderByDesc(**"name"**,**"id"**);  
 print(qw);  
}  
  
*/\*\*  
 \* order :指定字段和排序方向  
 \*  
 \* boolean condition : 条件内容是否加入到 sql语句的后面。  
 \* true：条件加入到sql语句  
 \* FROM student ORDER BY name ASC  
 \*  
 \* false:条件不加入到sql语句  
 \* FROM student  
 \*/*@Test  
**public void** testOrder(){  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 qw.orderBy(**true**,**true**,**"name"**)  
 .orderBy(**false**,**true**,**"age"**)  
 .orderBy(**true**,**false**,**"email"**);  
 *// name asc, email desc  
 //FROM student ORDER BY name ASC, email DESC* print(qw);  
}

### or、and

*/\*\*  
 \* and ,or方法，默认是and连接  
 \*/*@Test  
**public void** testOr(){  
 QueryWrapper<Student> qw= **new** QueryWrapper<>();  
 *//WHERE name = ? OR age = ?* qw.eq(**"name"**,**"张三"**)  
 .or()  
 .eq(**"age"**,22);  
 print(qw);  
}  
@Test  
**public void** testAnd(){  
 QueryWrapper<Student> qw= **new** QueryWrapper<>();  
 *//WHERE name = ? OR age = ?* qw.eq(**"name"**,**"张三"**)  
*// .and(i -> i.eq("name", "李白"))* .eq(**"age"**,22);  
 print(qw);  
}

### Last

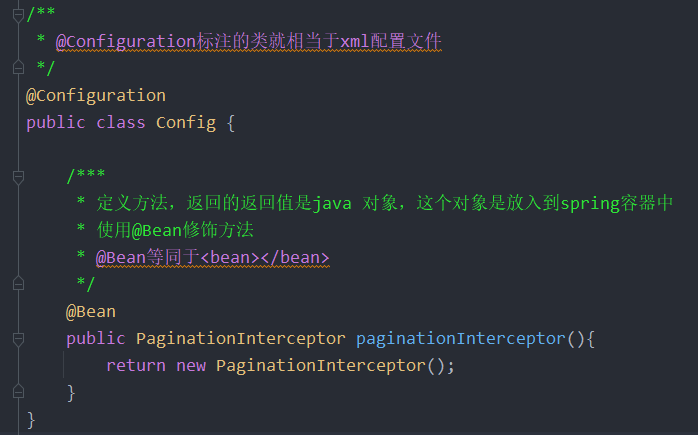
*/\*\*  
 \* last : 拼接sql语句到MP的sql语句的最后  
 \*/*@Test  
**public void** testLast(){  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 *//SELECT id,name,age,email,status FROM student WHERE name = ? OR age = ? limit 1* qw.eq(**"name"**,**"张三"**)  
 .or()  
 .eq(**"age"**,22)  
 .last(**"limit 1"**);  
 print(qw);  
}

### exists、notExists

*/\*\*  
 \* exists : 判断条件  
 \*  
 \* notExists  
 \*/*@Test  
**public void** testExists(){  
 QueryWrapper<Student> qw= **new** QueryWrapper<>();  
 *//SELECT id,name,age,email,status FROM student  
 // WHERE EXISTS (select id from student where age > 20)* qw.exists(**"select id from student where age > 90"**);  
  
 *//SELECT id,name,age,email,status FROM student  
 // WHERE NOT EXISTS (select id from student where age > 90)* qw.notExists(**"select id from student where age > 90"**);  
 print(qw);  
}

## 四、分页

配置分页插件，实现物理分页，默认是内存分页

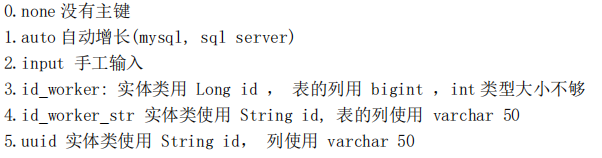
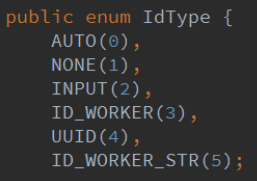


分页查询

*/\*\*  
 \* 分页：  
 \* 1.统计记录数，使用count(1)  
 \* SELECT COUNT(1) FROM student WHERE age > ?  
 \** ***2.实现分页，在sql语句的末尾加入 limit 0,3*** *\* SELECT id,name,age,email,status FROM student WHERE age > ? LIMIT 0,3  
 \*/*@Test  
**public void** testPage(){  
 QueryWrapper<Student> qw = **new** QueryWrapper<>();  
 qw.gt(**"age"**,22);  
  
 IPage<Student> page = **new** Page<>();  
 *//设置分页的数据* page.setCurrent(1);*//第一页* page.setSize(3);*// 每页的记录数* IPage<Student> result = **studentDao**.selectPage(page,qw);  
  
 *//获取分页后的记录* List<Student> students = result.getRecords();  
 System.***out***.println(**"students.size()="**+students.size());  
 *//分页的信息* **long** pages = result.getPages();  
 System.***out***.println(**"页数："**+pages);  
 System.***out***.println(**"总记录数："**+result.getTotal());  
 System.***out***.println(**"当前页码："**+result.getCurrent());  
 System.***out***.println(**"每页的记录数："**+result.getSize());  
}

# 第十章 MyBatis-Plus的表和列操作

## 主键类型



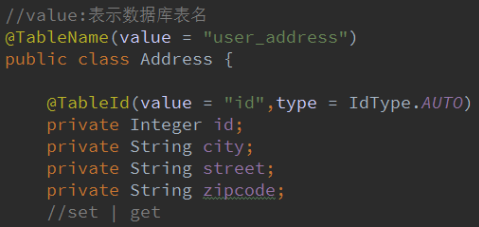
*/\*\*  
 \* 使用AR,要求实体类需要继承MP中的Model  
 \* Model中提供了对数据库的CRUD的操作  
 \*/*@Data  
**public class** Dept **extends** Model<Dept> {  
 *//定义属性， 属性名和表的列名一样  
  
 //配合数据库设置的自增长  
// @TableId(value = "id",type = IdType.AUTO)  
 //分布式id，使用叫做雪花算法，数据库id应设置为bigint，属性应设置为Long  
// @TableId(value = "id",type = IdType.ID\_WORKER)  
 //分布式id，使用叫做雪花算法，数据库id应设置为VARCHAR，属性应设置为String  
// @TableId(value = "id",type = IdType.ID\_WORKER\_STR)  
 //UUID* @TableId(value = **"id"**,type = IdType.***UUID***)  
 **private** String **id**;  
 **private** String **name**;  
 **private** String **mobile**;  
 **private** Integer **manager**;  
}

## 指定表名@TableName

定义实体类，默认的表名和实体类同名；如果不一致，在实体类定义上面使用@TableName说明表名称

例如：@TableName(value=”数据库表名”)

1. entity类



1. Mapper



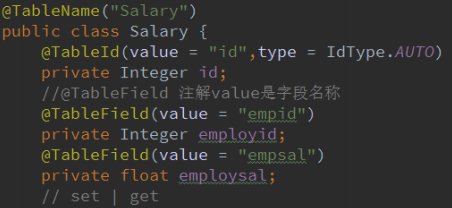
1. 测试





## 三、指定列名@TableField

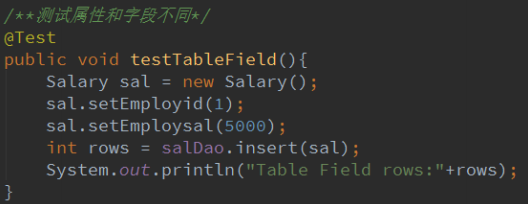
1. entity类



1. mapper类



1. 测试类





## 四、@TableField其他用法

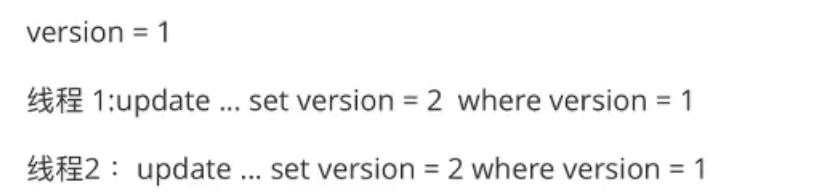
@Data  
**public class** Address {  
@TableId(value=**"user\_id"**,type = IdType.***AUTO***)  
 **private** Integer **id**;  
@TableField(value = **"user\_city"**)  
 **private** String **city**;  
  
 *//select表示是否查询该字段* @TableField(value = **"user\_street"**,select = **false**)  
 **private** String **street**;  
  
 *//exist表示是否为数据库字段，默认为true，改成false后不会去数据库查询，  
 //如果实体类中的成员变量在数据库中没有对应的字段，则可以使用exist* @TableField(exist = **false**)  
 **private** String **zipcode**;  
  
 *//fill表示是否自动填充，将对象存入数据库的时候，由mybatis-plus自动给对应字段赋值* @TableField(fill = FieldFill.***INSERT***)  
 **private** Date **createTime**;  
  
 @TableField(fill = FieldFill.***INSERT\_UPDATE***)  
 **private** Date **updateTime**;  
}

1）创建自动填充的处理器

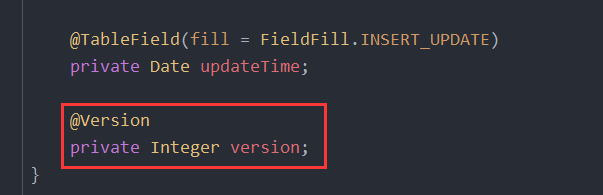
*/\*\*  
 \* 创建自动填充的处理器  
 \*/*@Component  
**public class** MyMetaObjectHandler **implements** MetaObjectHandler {  
  
 @Override  
 **public void** insertFill(MetaObject metaObject) {  
  
 **this**.setFieldValByName(**"creatTime"**,**new** Date(),metaObject);  
 **this**.setFieldValByName(**"updateTime"**,**new** Date(),metaObject);  
 }  
  
 @Override  
 **public void** updateFill(MetaObject metaObject) {  
  
 **this**.setFieldValByName(**"updateTime"**,**new** Date(),metaObject);  
 }  
}

## 五、@Version标记乐观锁

通过version字段来保证数据的安全性，当修改数据的时候，会以version作为条件，当条件成立的时候才会修改成功，修改成功后，version自动加1



1. 数据库表添加version字段，默认值为1
2. 实体类添加version成员变量，并且添加@Version

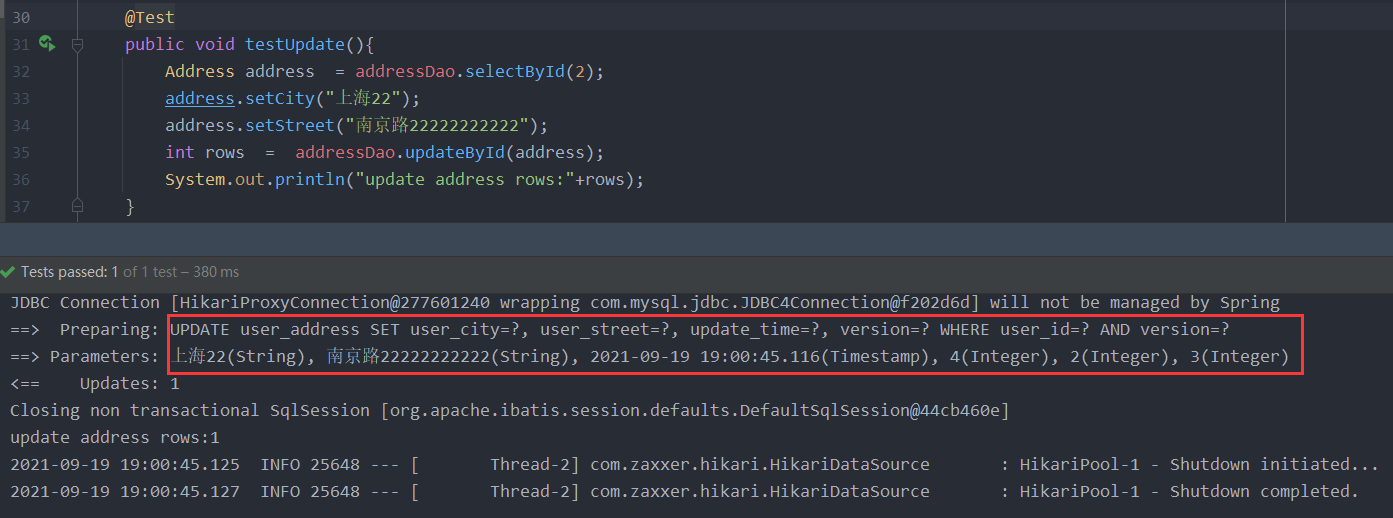


1. 注册配置类

*/\*\*  
 \** ***@Configuration标注的类就相当于xml配置文件*** *\*/*@Configuration  
**public class** Config {

*/\*\*  
 \** ***乐观锁插件*** *\*/*  
 @Bean  
 **public** OptimisticLockerInterceptor optimisticLockerInterceptor(){  
 **return new** OptimisticLockerInterceptor();  
 }  
}

1. 测试1



1. 测试2



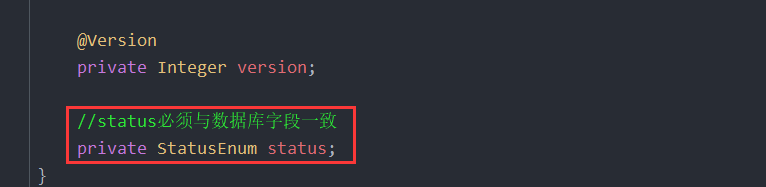
## 六、@EnumValue枚举类注解

将数据库字段映射成实体类的枚举类型成员变量

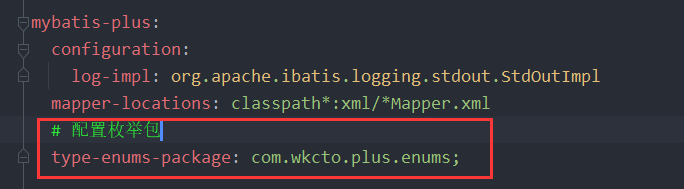
1. 定义枚举类

**public enum** StatusEnum {  
  
 ***WORK***(1,**"上班"**),  
 ***REST***(0,**"休息"**);  
  
 StatusEnum(Integer code, String msg) {  
 **this**.**code** = code;  
 **this**.**msg** = msg;  
 }  
 @EnumValue *//code与数据库status的值映射* **private** Integer **code**;  
 **private** String **msg**;  
}

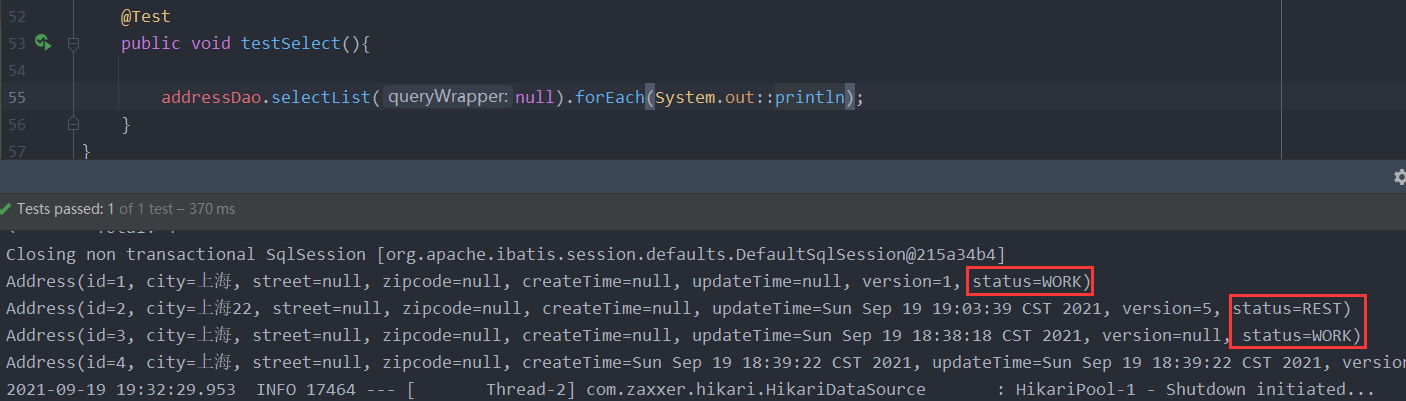
1. 实体类Address增加枚举类属性



1. application.yml配置枚举包



1. 数据库增加status字段
2. 测试



## 七、通过实现接口完成枚举映射

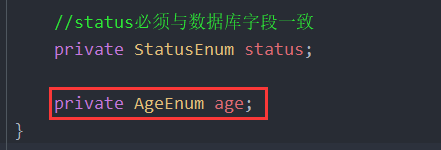
将数据库字段映射成实体类的枚举类型成员变量

1. 定义枚举类

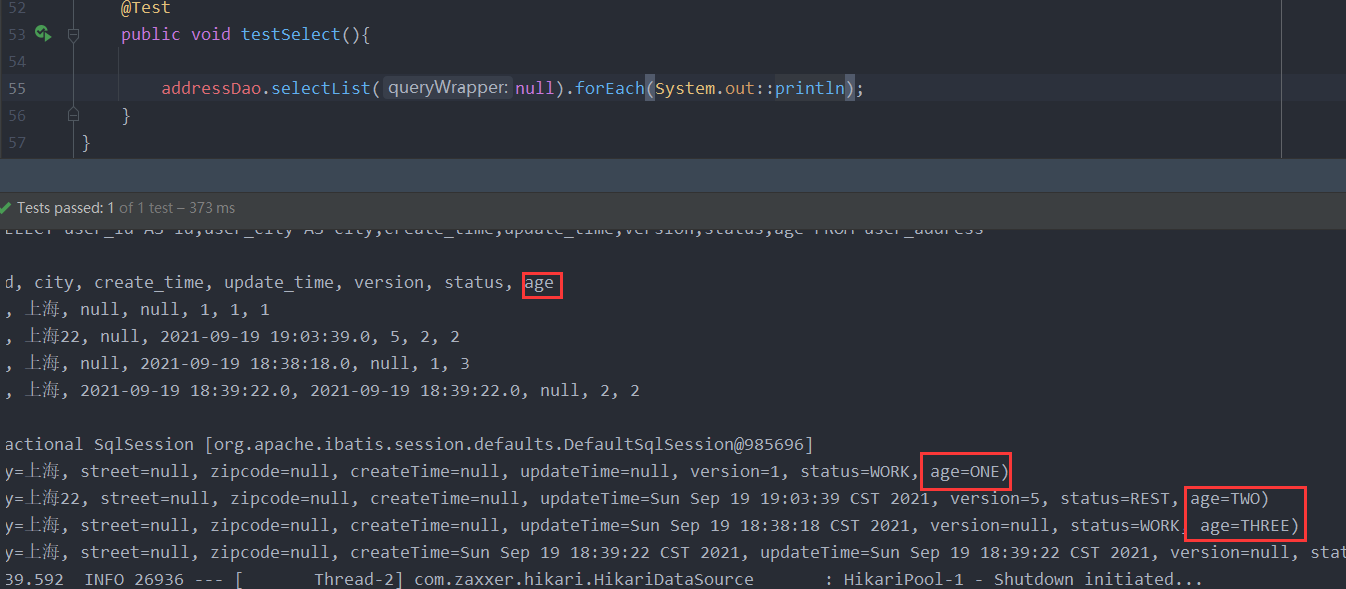
**public enum** AgeEnum **implements** IEnum<Integer> {  
  
 ***ONE***(1,**"一岁"**),  
 ***TWO***(2,**"二岁"**),  
 ***THREE***(3,**"三岁"**);

AgeEnum(Integer code, String msg) {  
 **this**.**code** = code;  
 **this**.**msg** = msg;  
 }  
 **private** Integer **code**;  
 **private** String **msg**;  
  
 @Override  
 **public** Integer getValue() {  
 **return this**.**code**; *//与数据库字段Status映射* }  
}

1. 实体类Address增加枚举类属性

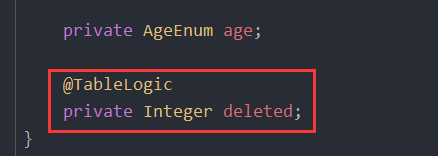


1. 数据库增加age字段
2. 测试

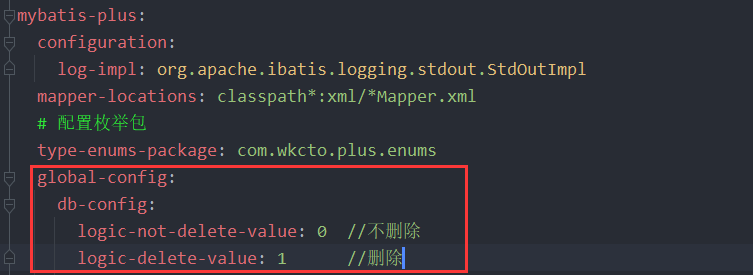


## 八、@TableLogic逻辑删除

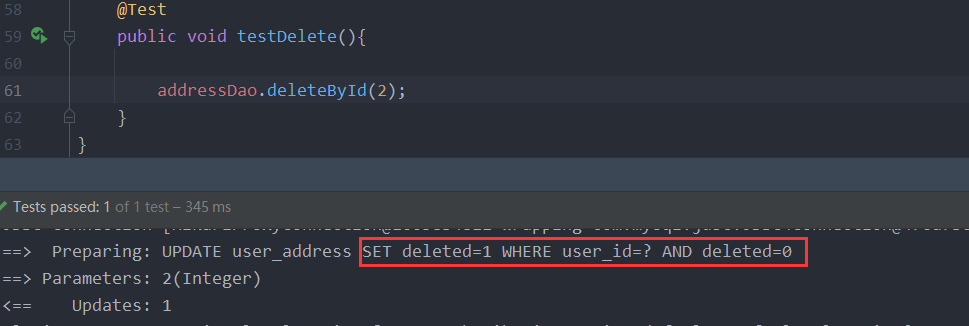
1. 数据库增加字段deleted
2. 实体类添加属性及其注解



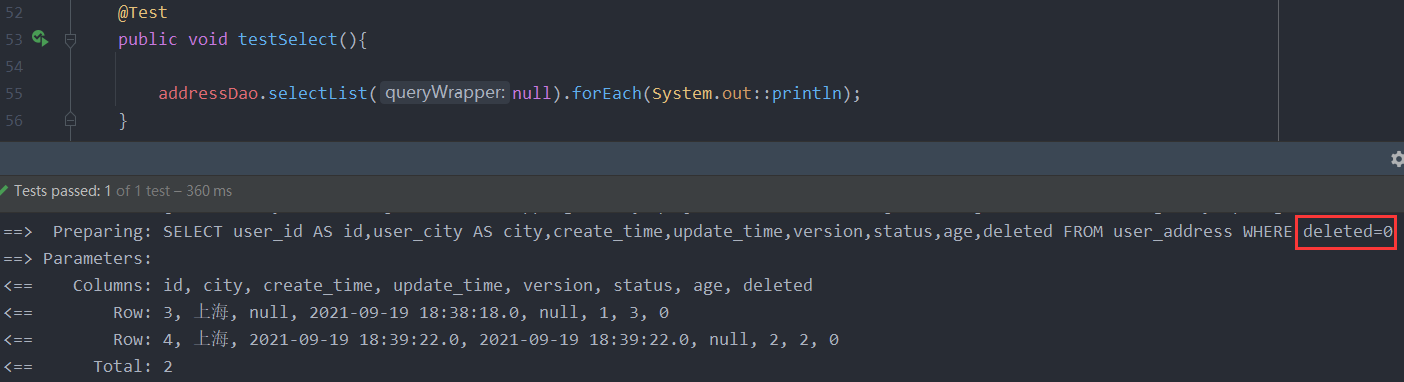
1. application.yml添加配置



1. 测试



1. 注意：此时查询只能查询出未被删除的数据

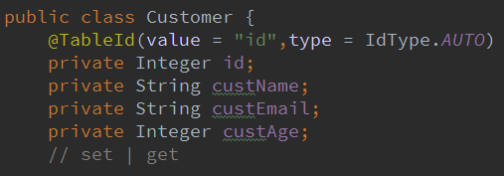


## 九、驼峰命名

1. 定义表



1. entity类



1. Mapper类



1. 测试

