mybatis

**第二天**

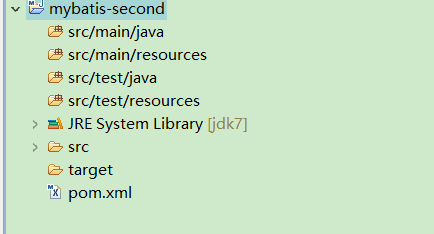
# 学习目标

* 能够使用Mapper映射文件中的parameterType属性
* 能够使用Mapper映射文件中的resultType属性
* 能够使用Mapper映射文件中的resultMap属性
* 能够使用Mybatis动态标签
* 能够在mybatis中完成两张表一对一的关联查询
* 能够在mybatis中完成两种表的一对多关联查询
* 能够使用逆向工程根据数据库表生成pojo和映射文件
* 能够完成mybatis和spring框架整合

# mybatis输入输出映射

## 准备环境

### 创建项目



### 配置pom.xml，导入依赖

* 数据库驱动包
* mybatis框架包
* log4j日志包
* junti单元测试框架包

|  |
| --- |
| <project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>  <modelVersion>4.0.0</modelVersion>    <groupId>cn.itheima</groupId>  <artifactId>mybatis-second</artifactId>  <version>0.0.1-SNAPSHOT</version>    <packaging>jar</packaging>  <name>mybatis-second</name>  <url>http://maven.apache.org</url>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <!-- mysql版本 -->  <mysql.version>5.1.30</mysql.version>  <!-- junit版本 -->  <junit.version>4.12</junit.version>  <!-- mybatis版本号 -->  <mybatis.version>3.4.5</mybatis.version>  <!-- log4j日志包版本 -->  <slf4j.version>1.7.7</slf4j.version>  <log4j.version>1.2.17</log4j.version>  </properties>  <dependencies>  <!-- mysql数据库依赖 -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>${mysql.version}</version>  </dependency>  <!-- mybatis核心包 -->  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>${mybatis.version}</version>  </dependency>  <!-- log4j日志包 -->  <dependency>  <groupId>log4j</groupId>  <artifactId>log4j</artifactId>  <version>${log4j.version}</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>${slf4j.version}</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  <version>${slf4j.version}</version>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>${junit.version}</version>  <scope>test</scope>  </dependency>  </dependencies>  </project> |

### 准备配置文件

#### 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>  <!--加载属性配置文件，说明：  1.resource属性：加载本地项目中的属性文件  2.url属性：加载网络上的属性文件  3.使用内部property标签定义属性  4.加载顺序：首先加载内部property标签定义的属性，再加载属性文件中定义的属性，  如果定义了相同的属性，属性文件中的属性覆盖property标签定义的属性 -->  <properties resource=*"db.properties"* >  <property name=*"db.username"* value=*"root"*/>  <property name=*"db.password"* value=*"admin"*/>  </properties>    <!-- typeAliases标签：配置自定义别名 -->  <typeAliases>  <!-- 以包扫描的方式，配置自定义别名，说明：  1.如果有多个pojo包需要扫描，配置多个package-->  <package name=*"cn.itheima.po"*/>  </typeAliases>  <!-- 数据源运行环境配置 -->  <!-- default属性：指定使用哪一个 environment-->  <environments default=*"development"*>  <!-- id属性：区分每一个environment -->  <environment id=*"development"*>  <!-- 使用jdbc的事务 -->  <transactionManager type=*"JDBC"* />  <!--POOLED:mybagtis提供的连接池 -->  <dataSource type=*"POOLED"*>  <!-- 使用db.properties文件中的属性 -->  <property name=*"driver"* value=*"${db.driver}"* />  <property name=*"url"* value=*"${db.url}"* />  <property name=*"username"* value=*"${db.username}"* />  <property name=*"password"* value=*"${db.password}"* />    </dataSource>  </environment>  </environments>    <!--mapper映射文件配置 -->  <mappers>  <!-- 以包扫描的方式，配置mapper映射文件 ，说明：  1.前提必须是mapper代理开发方法  2.mapper映射文件，必须与mapper代理接口在同一个目录  3.mapper映射文件的名称，必须与mapper代理接口的名称一致  4.如果有多个mapper包要扫描，配置多个package-->  <package name=*"cn.itheima.mapper"*/>  </mappers>  </configuration> |

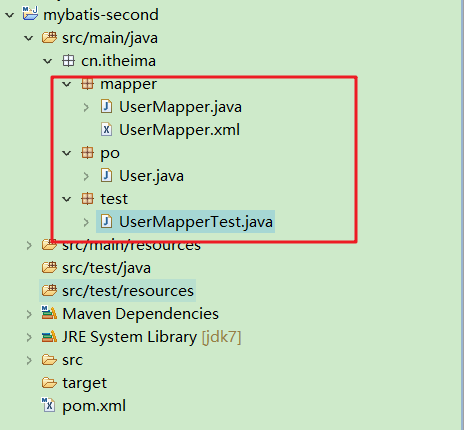
#### db.properties

|  |
| --- |
| db.driver=com.mysql.jdbc.Driver  db.url=jdbc:mysql://127.0.0.1:3306/82\_mybatis  db.username=root  db.password=admin |

#### log4j.properties

|  |
| --- |
| # Global logging configuration  log4j.rootLogger=DEBUG, stdout  # Console output...  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n |

#### 复制入门程序中的实体类和mapper接口以及映射文件

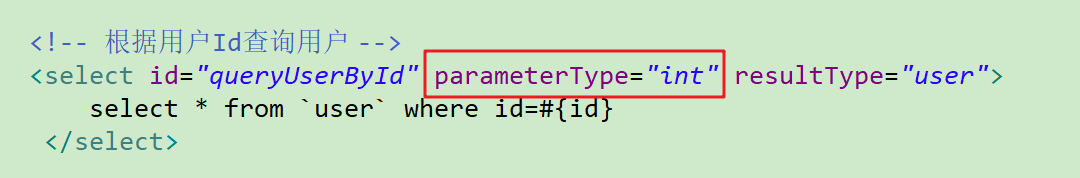


## ParameterType输入映射

说明：企业项目开发中常用的类型。

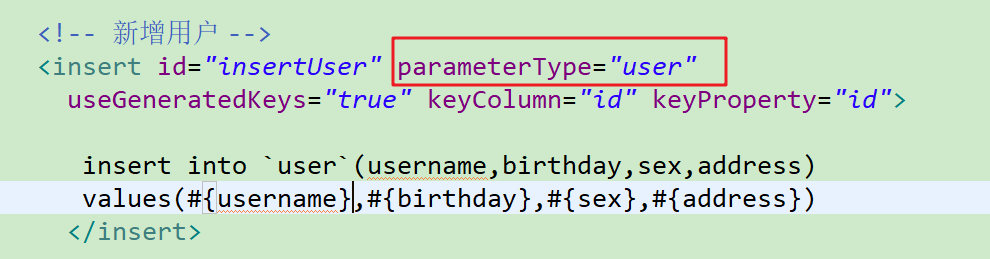
### java简单类型

参考入门程序内容：



### pojo类型

参考入门程序内容：



### pojo包装类型

定义：pojo包装类型，就是在pojo中包含了其它的pojo。通常用于接收综合查询条件。



说明：

1. 如果不使用pojo包装类型，至少需要三个pojo来接收条件（职位/行业/公司）
2. 使用pojo包装类型，只需要一个即可

#### 需求

使用pojo包装类型，实现根据用户名称模糊查询用户。

#### 需求实现

##### 编写pojo包装类型

|  |
| --- |
| **package** cn.itheima.po;  /\*\*  \* **@ClassName**: QueryVo  \* **@Description**: pojo包装类型  \* **@author** 传智 小杨老师  \* **@date** 2018-8-12 下午3:13:36  \*  \*/  **public** **class** QueryVo {    // 包装用户对象  **private** User user;  /\*\*  \* **@return** the user  \*/  **public** User getUser() {  **return** user;  }  /\*\*  \* **@param** user the user to set  \*/  **public** **void** setUser(User user) {  **this**.user = user;  }      } |

##### 声明mapper接口方法

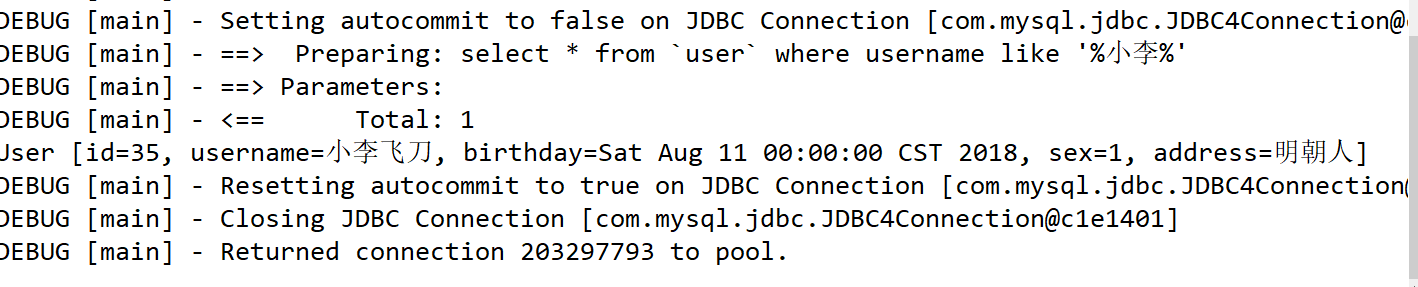
|  |
| --- |
| **package** cn.itheima.mapper;  **import** java.util.List;  **import** cn.itheima.po.QueryVo;  **import** cn.itheima.po.User;  /\*\*  \* **@ClassName**: UserMapper  \* **@Description**: 用户mapper接口  \* **@author** 传智 小杨老师  \* **@date** 2018-8-11 下午5:56:48  \*  \*/  **public** **interface** UserMapper {    /\*\*  \* 1.根据用户Id查询用户  \*/  User queryUserById(Integer id);    /\*\*  \* 2.新增用户  \*/  **void** insertUser(User user);    /\*\*  \* 3.使用pojo包装类型，实现根据用户名称模糊查询用户  \*/  List<User> queryUserByCondition(QueryVo queryVo);  } |

##### 配置mapper映射文件

|  |
| --- |
| <!-- 使用pojo包装类型，实现根据用户名称模糊查询用户 -->  <select id=*"queryUserByCondition"* parameterType=*"queryVo"* resultType=*"user"*>  select \* from `user` where username like '%${user.username}%'  </select> |

##### 测试

|  |
| --- |
| /\*\*  \* 测试使用pojo包装类型，实现根据用户名称模糊查询用户  \*/  @Test  **public** **void** queryUserByConditionTest(){  // 1.创建SqlSession  SqlSession sqlSession = **this**.sqlSessionFactory.openSession();    // 2.获取mapper接口代理对象  UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);      // 3.调用方法执行  // 创建包装对象  QueryVo queryVo = **new** QueryVo();    // 创建用户对象  User user = **new** User();  user.setUsername("小李");    queryVo.setUser(user);    List<User> list = mapper.queryUserByCondition(queryVo);  **for**(User u:list){  System.*out*.println(u);  }    // 4.释放资源  sqlSession.close();  } |



## ResultType输出映射

说明：在企业项目开发中，常用的类型。

### java简单类型

#### 需求

统计用户表记录数量。

#### 需求实现

##### 声明mapper接口方法

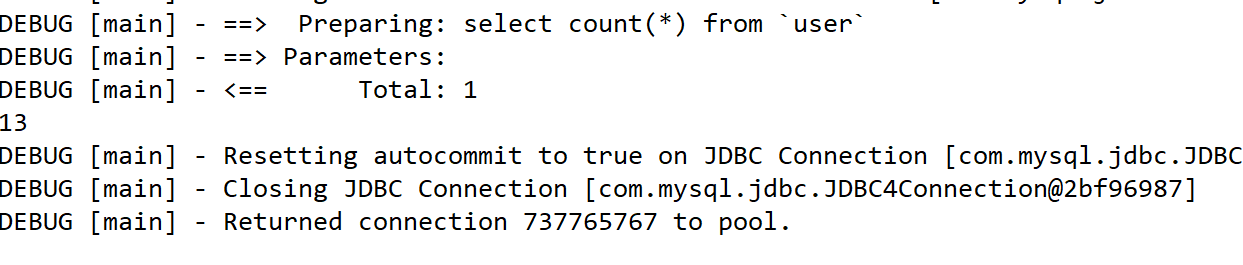
|  |
| --- |
| **package** cn.itheima.mapper;  **import** java.util.List;  **import** cn.itheima.po.QueryVo;  **import** cn.itheima.po.User;  /\*\*  \* **@ClassName**: UserMapper  \* **@Description**: 用户mapper接口  \* **@author** 传智 小杨老师  \* **@date** 2018-8-11 下午5:56:48  \*  \*/  **public** **interface** UserMapper {    /\*\*  \* 1.根据用户Id查询用户  \*/  User queryUserById(Integer id);    /\*\*  \* 2.新增用户  \*/  **void** insertUser(User user);    /\*\*  \* 3.使用pojo包装类型，实现根据用户名称模糊查询用户  \*/  List<User> queryUserByCondition(QueryVo queryVo);    /\*\*  \* 4.统计用户数量  \*/  Integer countUsers();  } |

##### 配置mapper映射文件

|  |
| --- |
| <!-- 统计用户数量 -->  <select id=*"countUsers"* resultType=*"int"*>  select count(\*) from `user`  </select> |

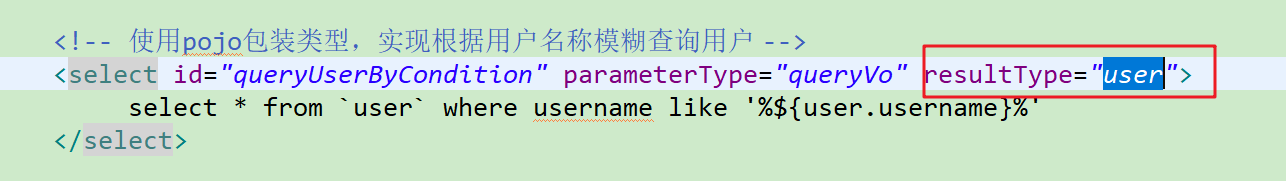
##### 测试

|  |
| --- |
| /\*\*  \* 测试统计用户数量  \*/  @Test  **public** **void** countUsersTest(){  // 1.创建SqlSession  SqlSession sqlSession = **this**.sqlSessionFactory.openSession();    // 2.获取mapper接口代理对象  UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);      // 3.调用方法执行  Integer users = mapper.countUsers();  System.*out*.println(users);    // 4.释放资源  sqlSession.close();  } |



### pojo类型

参考前边的课程内容：



## ResultMap输出映射

### 需求

查询全部订单数据。

### 需求实现

#### 编写订单实体类

|  |
| --- |
| **package** cn.itheima.po;  **import** java.util.Date;  /\*\*  \* **@ClassName**: orders  \* **@Description**: 订单实体类  \* **@author** 传智 小杨老师  \* **@date** 2018-8-12 下午3:42:19  \*  \*/  **public** **class** Orders {    **private** Integer id; // int(11) NOT NULL AUTO\_INCREMENT,    // user\_id  **private** Integer userId; // int(11) NOT NULL COMMENT '下单用户id',    **private** String number; // varchar(32) NOT NULL COMMENT '订单号',  **private** Date createtime; // datetime NOT NULL COMMENT '创建订单时间',  **private** String note; // varchar(100) DEFAULT NULL COMMENT '备注',  /\*\*  \* **@return** the id  \*/  **public** Integer getId() {  **return** id;  }  /\*\*  \* **@param** id the id to set  \*/  **public** **void** setId(Integer id) {  **this**.id = id;  }  /\*\*  \* **@return** the userId  \*/  **public** Integer getUserId() {  **return** userId;  }  /\*\*  \* **@param** userId the userId to set  \*/  **public** **void** setUserId(Integer userId) {  **this**.userId = userId;  }  /\*\*  \* **@return** the number  \*/  **public** String getNumber() {  **return** number;  }  /\*\*  \* **@param** number the number to set  \*/  **public** **void** setNumber(String number) {  **this**.number = number;  }  /\*\*  \* **@return** the createtime  \*/  **public** Date getCreatetime() {  **return** createtime;  }  /\*\*  \* **@param** createtime the createtime to set  \*/  **public** **void** setCreatetime(Date createtime) {  **this**.createtime = createtime;  }  /\*\*  \* **@return** the note  \*/  **public** String getNote() {  **return** note;  }  /\*\*  \* **@param** note the note to set  \*/  **public** **void** setNote(String note) {  **this**.note = note;  }  /\* (non-Javadoc)  \* @see java.lang.Object#toString()  \*/  @Override  **public** String toString() {  **return** "Orders [id=" + id + ", userId=" + userId + ", number=" + number  + ", createtime=" + createtime + ", note=" + note + "]";  }  } |

#### 编写订单mapper接口

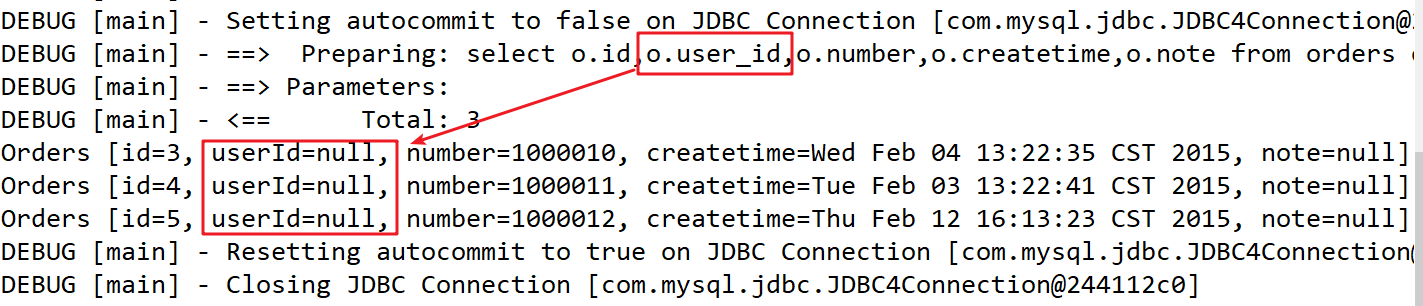
|  |
| --- |
| **package** cn.itheima.mapper;  **import** java.util.List;  **import** cn.itheima.po.Orders;  /\*\*  \* **@ClassName**: OrdersMapper  \* **@Description**: 订单mapper接口  \* **@author** 传智 小杨老师  \* **@date** 2018-8-12 下午3:44:21  \*  \*/  **public** **interface** OrdersMapper {    /\*\*  \* 查询全部订单数据  \*/  List<Orders> queryAllOrders();  } |

#### 编写订单mapper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">  <!--namespace：名称空间，相当于java中的package，用于防止sql语句名称冲突（sql语句隔离） -->  <!--mapper代理开发方法，namespace属性值必须是mapper接口全路径 -->  <mapper namespace=*"cn.itheima.mapper.OrdersMapper"*>  <!-- 查询全部订单数据 -->  <select id=*"queryAllOrders"* resultType=*"orders"*>  select o.id,o.user\_id,o.number,o.createtime,o.note from orders o  </select>  </mapper> |

#### 测试

|  |
| --- |
| **package** cn.itheima.test;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.util.List;  **import** org.apache.ibatis.io.Resources;  **import** org.apache.ibatis.session.SqlSession;  **import** org.apache.ibatis.session.SqlSessionFactory;  **import** org.apache.ibatis.session.SqlSessionFactoryBuilder;  **import** org.junit.Before;  **import** org.junit.Test;  **import** cn.itheima.mapper.OrdersMapper;  **import** cn.itheima.po.Orders;  /\*\*  \* **@ClassName**: OrdersMapperTest  \* **@Description**: 订单测试  \* **@author** 传智 小杨老师  \* **@date** 2018-8-12 下午3:46:40  \*  \*/  **public** **class** OrdersMapperTest {    // 定义SqlSessionFactory  **private** SqlSessionFactory sqlSessionFactory = **null**;    /\*\*  \* 初始化SqlSessionFactory  \* **@throws** IOException  \*/  @Before  **public** **void** init() **throws** IOException{  // 1.加载主配置文件（sqlMapConfig.xml）  InputStream inputStream = Resources.*getResourceAsStream*("sqlMapConfig.xml");    // 2.读取配置文件内容  SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();  /\*\*  \* sqlSessionFactory:  \* 1.它是mybatis框架的核心对象  \* 2.它是线程安全的  \* 3.一个项目中通常只需要一个（单例设计模式的应用）  \* 4.它给我们创建SqlSession对象  \*/  sqlSessionFactory = builder.build(inputStream);  }    /\*\*  \* 查询全部订单数据  \*/  @Test  **public** **void** queryAllOrdersTest(){    // 1.创建sqlSession  SqlSession sqlSession = **this**.sqlSessionFactory.openSession();    // 2.获取mapper接口代理对象  OrdersMapper mapper = sqlSession.getMapper(OrdersMapper.**class**);      // 3.调用方法执行  List<Orders> list = mapper.queryAllOrders();  **for**(Orders o:list){  System.*out*.println(o);  }    // 4.释放资源  sqlSession.close();    }    } |



注意事项：使用resultType输出映射，要求sql语句中的字段的名称与java对象的属性名称要一致，才能完成映射。

### 使用ResultMap实现

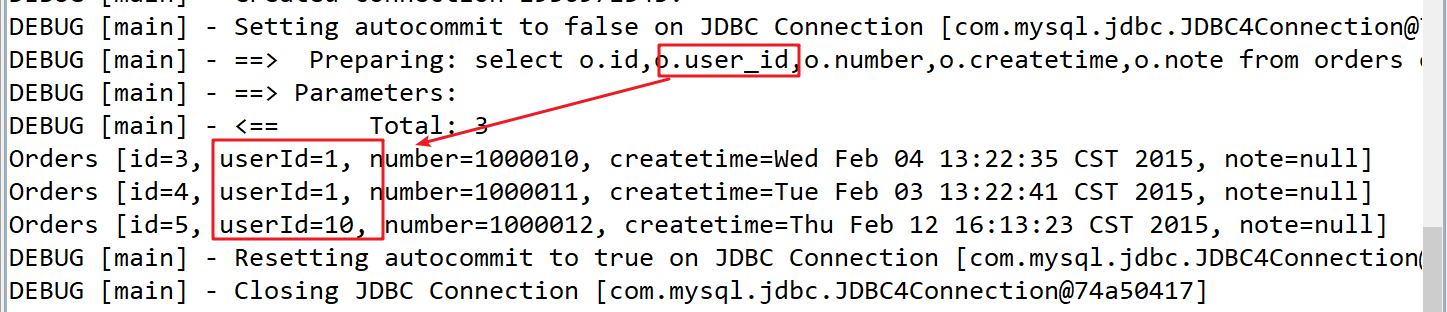
说明：ResultMap用于配置java对象的属性名称，与sql语句字段的名称对应关系。本质上还是把结果映射到java对象上。

#### 声明mapper接口方法

|  |
| --- |
| **package** cn.itheima.mapper;  **import** java.util.List;  **import** cn.itheima.po.Orders;  /\*\*  \* **@ClassName**: OrdersMapper  \* **@Description**: 订单mapper接口  \* **@author** 传智 小杨老师  \* **@date** 2018-8-12 下午3:44:21  \*  \*/  **public** **interface** OrdersMapper {    /\*\*  \* 查询全部订单数据  \*/  List<Orders> queryAllOrders();    /\*\*  \* 查询全部订单数据，使用ResultMap实现  \*/  List<Orders> queryAllOrdersByResultMap();  } |

#### 配置mapper映射文件

|  |
| --- |
| <!--resultMap:配置java对象的属性名称，与sql语句字段的名称对应关系，说明：  type：要映射的类型  id:唯一标识名称，通过id引用该resultMap  -->  <resultMap type=*"orders"* id=*"orderResultMap"*>  <!-- id：配置主键字段对应关系，说明：  column：字段名称  property：属性名称  -->  <id column=*"id"* property=*"id"*/>    <!-- result：配置普通字段对应关系 -->  <result column=*"user\_id"* property=*"userId"*/>  <result column=*"number"* property=*"number"*/>  <result column=*"createtime"* property=*"createtime"*/>  <result column=*"note"* property=*"note"*/>    </resultMap>    <!-- 查询全部订单数据，使用ResultMap实现 -->  <select id=*"queryAllOrdersByResultMap"* resultMap=*"orderResultMap"* >  select o.id,o.user\_id,o.number,o.createtime,o.note from orders o  </select> |



# mybatis动态sql

说明：学习标签if、where、set、sql、include、foreach的使用。

## 动态sql案例演示

### 需求

根据用户名称和性别查询用户。

### 需求实现

#### 声明mapper接口方法

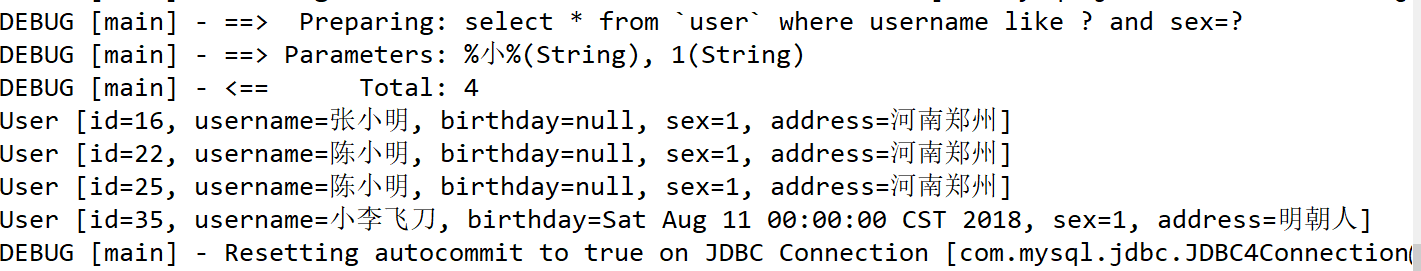
|  |
| --- |
| **package** cn.itheima.mapper;  **import** java.util.List;  **import** cn.itheima.po.QueryVo;  **import** cn.itheima.po.User;  /\*\*  \* **@ClassName**: UserMapper  \* **@Description**: 用户mapper接口  \* **@author** 传智 小杨老师  \* **@date** 2018-8-11 下午5:56:48  \*  \*/  **public** **interface** UserMapper {    /\*\*  \* 1.根据用户Id查询用户  \*/  User queryUserById(Integer id);    /\*\*  \* 2.新增用户  \*/  **void** insertUser(User user);    /\*\*  \* 3.使用pojo包装类型，实现根据用户名称模糊查询用户  \*/  List<User> queryUserByCondition(QueryVo queryVo);    /\*\*  \* 4.统计用户数量  \*/  Integer countUsers();    /\*\*  \* 5.根据用户名称和性别查询用户  \*/  List<User> queryUserByNameAndSex(User user);  } |

#### 配置mapper映射文件

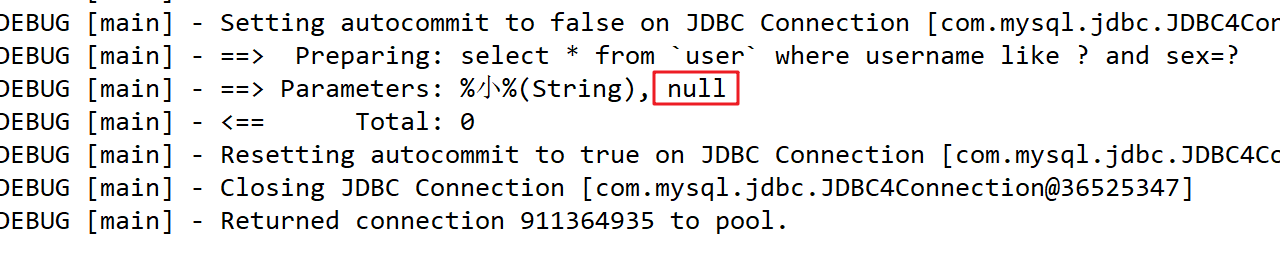
|  |
| --- |
| <!-- 根据用户名称和性别查询用户 -->  <select id=*"queryUserByNameAndSex"* parameterType=*"user"* resultType=*"User"*>  select \* from `user` where username like #{username} and sex=#{sex}  </select> |

#### 测试

|  |
| --- |
| /\*\*  \* 测试根据用户名称和性别查询用户  \*/  @Test  **public** **void** queryUserByNameAndSexTest(){  // 1.创建SqlSession  SqlSession sqlSession = **this**.sqlSessionFactory.openSession();    // 2.获取mapper接口代理对象  UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);      // 3.调用方法执行  // 创建用户对象  User user = **new** User();  user.setUsername("%小%");  user.setSex("1");    List<User> list = mapper.queryUserByNameAndSex(user);  **for**(User u:list){  System.*out*.println(u);  }    // 4.释放资源  sqlSession.close();  } |



问题：当参数不传递，mybatis框架会默认设置一个null值，导致查询不到数据。



## if标签

作用：判断参数请求，拼装sql语句片段。

|  |
| --- |
| <!-- 根据用户名称和性别查询用户 -->  <select id=*"queryUserByNameAndSex"* parameterType=*"user"* resultType=*"User"*>  select \* from `user`  where <!-- username like #{username} and sex=#{sex} -->  <!--if:判断用户名称不为空，且不为空字符串，才作为where条件 -->  <if test=*"username !=null and username !=''"*>  username like #{username}  </if>  <!--if:判断用户性别不为空，且不为空字符串，才作为where条件 -->  <if test=*"sex !=null and sex !=''"*>  and sex=#{sex}  </if>  </select> |

## where标签

作用：

1. where标签，相当于sql语句中的where关键字
2. 根据传入的参数情况，智能的去掉多余的and，or关键字

3.根据传入的参数情况，智能的去掉多余where关键字

|  |
| --- |
| <!-- 根据用户名称和性别查询用户 -->  <select id=*"queryUserByNameAndSex"* parameterType=*"user"* resultType=*"User"*>  select \* from `user`  <!--where username like #{username} and sex=#{sex} -->  <where>  <!--if:判断用户名称不为空，且不为空字符串，才作为where条件 -->  <if test=*"username !=null and username !=''"*>  username like #{username}  </if>  <!--if:判断用户性别不为空，且不为空字符串，才作为where条件 -->  <if test=*"sex !=null and sex !=''"*>  and sex=#{sex}  </if>  </where>    </select> |

## set标签

作用：

1. set关键字，相当于sql语句中的set关键字
2. 根据传入的参数情况，智能的去掉最后一个多余的逗号

### 需求

动态修改用户数据。

### 需求实现

#### 声明mapper接口方法

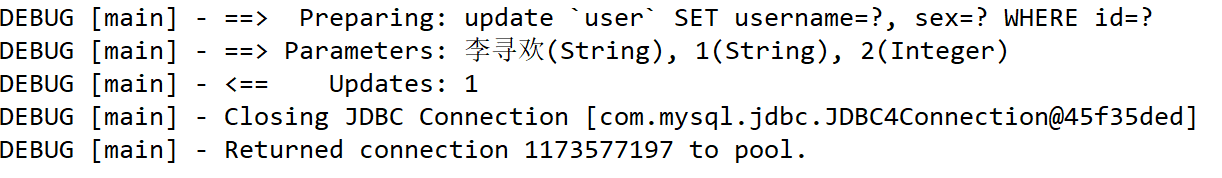
|  |
| --- |
| **package** cn.itheima.mapper;  **import** java.util.List;  **import** cn.itheima.po.QueryVo;  **import** cn.itheima.po.User;  /\*\*  \* **@ClassName**: UserMapper  \* **@Description**: 用户mapper接口  \* **@author** 传智 小杨老师  \* **@date** 2018-8-11 下午5:56:48  \*  \*/  **public** **interface** UserMapper {    /\*\*  \* 1.根据用户Id查询用户  \*/  User queryUserById(Integer id);    /\*\*  \* 2.新增用户  \*/  **void** insertUser(User user);    /\*\*  \* 3.使用pojo包装类型，实现根据用户名称模糊查询用户  \*/  List<User> queryUserByCondition(QueryVo queryVo);    /\*\*  \* 4.统计用户数量  \*/  Integer countUsers();    /\*\*  \* 5.根据用户名称和性别查询用户  \*/  List<User> queryUserByNameAndSex(User user);    /\*\*  \* 6.动态修改用户数据  \*/  **void** dynamicUpdateUser(User user);  } |

#### 配置mapper映射文件

|  |
| --- |
| <!-- 动态修改用户数据 -->  <update id=*"dynamicUpdateUser"* parameterType=*"user"*>  update `user`  <!--set username='小李飞刀',sex='2' where id=2 -->  <set>  <if test=*"username !=null and username !=''"*>  username=#{username},  </if>  <if test=*"sex != null and sex !=''"*>  sex=#{sex},  </if>  </set>    <where>  id=#{id}  </where>  </update> |

#### 测试

|  |
| --- |
| /\*\*  \* 测试动态修改用户数据  \*/  @Test  **public** **void** dynamicUpdatUserTest(){  // 1.创建SqlSession  SqlSession sqlSession = **this**.sqlSessionFactory.openSession(**true**);    // 2.获取mapper接口代理对象  UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);      // 3.调用方法执行  // 创建用户对象  User user = **new** User();  user.setUsername("李寻欢");  user.setSex("1");  user.setId(2);    mapper.dynamicUpdateUser(user);    // 4.释放资源  sqlSession.close();  } |



## sql标签（include标签）

作用：提取公共的sql片段。

|  |
| --- |
| <!--sql：提取公共的sql片段，说明：  id：唯一标识名称，通过id引用该sql片段  -->  <sql id=*"select\_orders\_list"*>  o.id,o.user\_id,o.number,o.createtime,o.note  </sql> |
| <!-- 查询全部订单数据 -->  <select id=*"queryAllOrders"* resultType=*"orders"*>  <!-- select o.id,o.user\_id,o.number,o.createtime,o.note from orders o -->    <!-- include：引用sql片段，说明：  refid：被引用的sql片段的id属性值  -->  select <include refid=*"select\_orders\_list"*></include> from orders o  </select> |

## foreach标签

作用：循环遍历参数集合（list，数组）。

### 需求

1. 批量新增用户
2. 批量删除用户

### 实现批量新增用户

#### 声明mapper接口方法

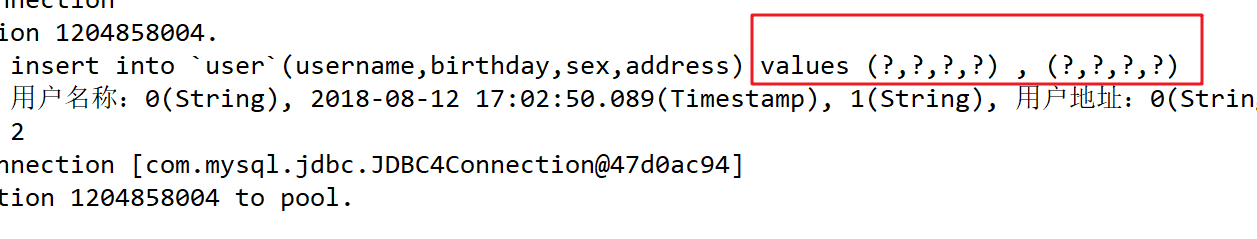
|  |
| --- |
| **package** cn.itheima.mapper;  **import** java.util.List;  **import** cn.itheima.po.QueryVo;  **import** cn.itheima.po.User;  /\*\*  \* **@ClassName**: UserMapper  \* **@Description**: 用户mapper接口  \* **@author** 传智 小杨老师  \* **@date** 2018-8-11 下午5:56:48  \*  \*/  **public** **interface** UserMapper {    /\*\*  \* 1.根据用户Id查询用户  \*/  User queryUserById(Integer id);    /\*\*  \* 2.新增用户  \*/  **void** insertUser(User user);    /\*\*  \* 3.使用pojo包装类型，实现根据用户名称模糊查询用户  \*/  List<User> queryUserByCondition(QueryVo queryVo);    /\*\*  \* 4.统计用户数量  \*/  Integer countUsers();    /\*\*  \* 5.根据用户名称和性别查询用户  \*/  List<User> queryUserByNameAndSex(User user);    /\*\*  \* 6.动态修改用户数据  \*/  **void** dynamicUpdateUser(User user);    /\*\*  \* 7.批量新增用户  \*/  **void** batchInsertUser(List<User> list);  } |

#### 配置mapper映射文件

|  |
| --- |
| <!-- 批量新增用户 -->  <insert id=*"batchInsertUser"* parameterType=*"list"*>  insert into `user`(username,birthday,sex,address)  values  <!-- foreach：循环处理参数集合，说明：  collection：参数集合，这里是list（固定写法）  item：当前遍历的对象  separator：指定分割符  -->  <foreach collection=*"list"* item=*"u"* separator=*","*>  (#{u.username},#{u.birthday},#{u.sex},#{u.address})  </foreach>  <!-- ('名称1','2018-08-12','1','地址1'),('名称2','2018-08-12','1','地址2') -->  </insert> |

#### 测试

|  |
| --- |
| /\*\*  \* 测试批量新增用户  \*/  @Test  **public** **void** batchInsertUserTest(){  // 1.创建SqlSession  SqlSession sqlSession = **this**.sqlSessionFactory.openSession(**true**);    // 2.获取mapper接口代理对象  UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);      // 3.调用方法执行  // 创建用户集合list  List<User> list = **new** ArrayList<User>();  **for**(**int** i=0;i<2;i++){  // 创建用户  User user = **new** User();  user.setUsername("用户名称："+i);  user.setBirthday(**new** Date());  user.setSex("1");  user.setAddress("用户地址："+i);    list.add(user);  }  mapper.batchInsertUser(list);    // 4.释放资源  sqlSession.close();  } |



### 实现批量删除用户

#### 声明mapper接口方法

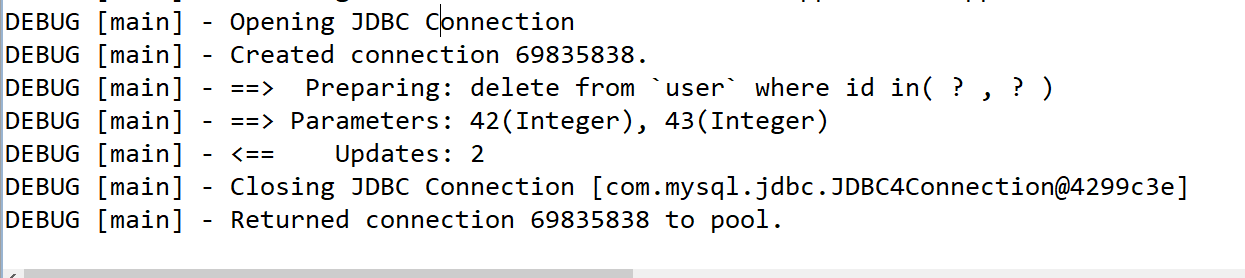
|  |
| --- |
| **package** cn.itheima.mapper;  **import** java.util.List;  **import** cn.itheima.po.QueryVo;  **import** cn.itheima.po.User;  /\*\*  \* **@ClassName**: UserMapper  \* **@Description**: 用户mapper接口  \* **@author** 传智 小杨老师  \* **@date** 2018-8-11 下午5:56:48  \*  \*/  **public** **interface** UserMapper {    /\*\*  \* 1.根据用户Id查询用户  \*/  User queryUserById(Integer id);    /\*\*  \* 2.新增用户  \*/  **void** insertUser(User user);    /\*\*  \* 3.使用pojo包装类型，实现根据用户名称模糊查询用户  \*/  List<User> queryUserByCondition(QueryVo queryVo);    /\*\*  \* 4.统计用户数量  \*/  Integer countUsers();    /\*\*  \* 5.根据用户名称和性别查询用户  \*/  List<User> queryUserByNameAndSex(User user);    /\*\*  \* 6.动态修改用户数据  \*/  **void** dynamicUpdateUser(User user);    /\*\*  \* 7.批量新增用户  \*/  **void** batchInsertUser(List<User> list);    /\*\*  \* 8.批量删除用户  \*/  **void** batchDeleteUser(Integer[] ids);  } |

#### 配置mapper映射文件

|  |
| --- |
| <!-- 批量删除用户 ,说明：  parameterType：参数传递的是list和数组的时候，都建议写成list  -->  <delete id=*"batchDeleteUser"* parameterType=*"list"*>  delete from `user`  where  <!-- foreach：循环遍历参数集合，说明：  collection：参数集合，这里是数组array（固定写法）  open：拼装的sql片段的开始部分  close：拼装的sql片段的结束部分  item：当前遍历的元素  separator：指定元素的分割符  -->  <foreach collection=*"array"* open=*"id in("* close=*")"* item=*"id"* separator=*","*>  #{id}  </foreach>  <!-- id in(40,41) -->  </delete> |

#### 测试

|  |
| --- |
| /\*\*  \* 测试批量删除用户  \*/  @Test  **public** **void** batchDeleteUserTest(){  // 1.创建SqlSession  SqlSession sqlSession = **this**.sqlSessionFactory.openSession(**true**);    // 2.获取mapper接口代理对象  UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);      // 3.调用方法执行  // 创建用户Id数组  Integer[] ids = {42,43};  mapper.batchDeleteUser(ids);    // 4.释放资源  sqlSession.close();  } |



# mybatis关联查询

关联关系：一对一关联关系、一对多关联关系、多对多关联关系（通过中间表看成两个一对多的关系）。

## 分析用户订单数据模型

分析思路：

1. 先确定单表有什么字段
2. 找出表的关联字段
3. 根据业务需求确定表之间的关系

用户表：user

用户id：id

用户名称：username

生日：birthday

性别：sex

地址：address

订单表：orders

订单id：id

所属用户Id：user\_id

订单编号：number

下单时间：createtime

备注信息：note

一个用户可以有多个订单：

一对多

一个订单只能属于一个用户：一对一

## 一对一关联查询

### 需求

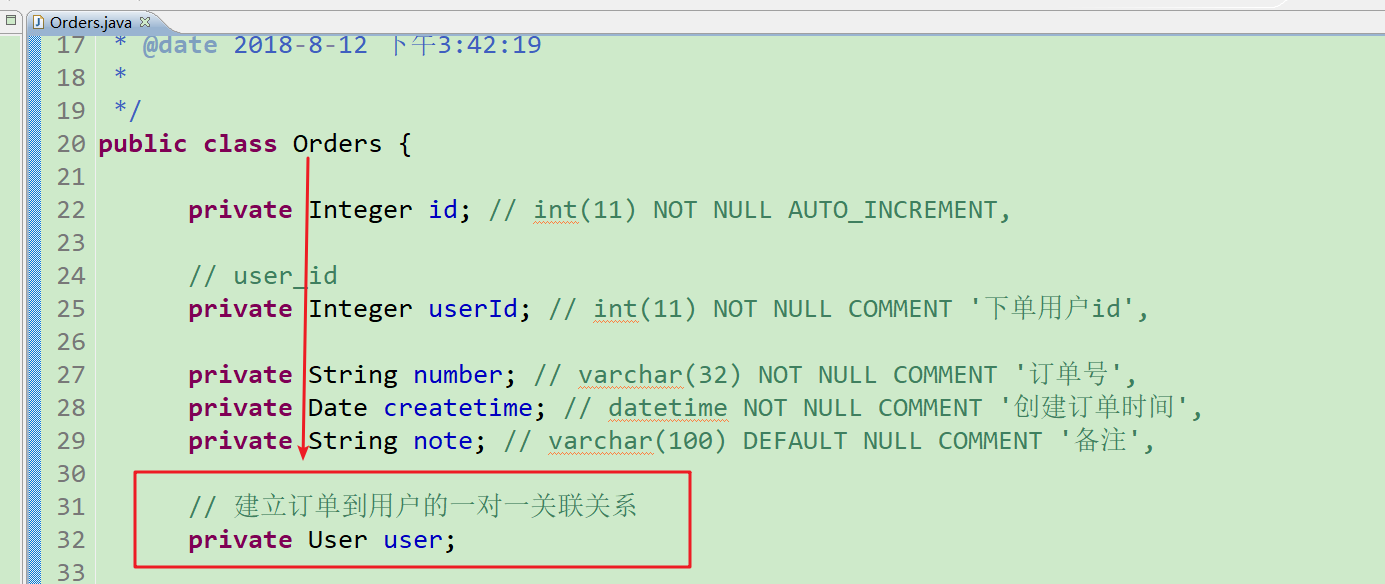
查询全部订单数据，并且关联查询出订单所属的用户数据。

### 需求实现

#### 编写sql语句

|  |
| --- |
| SELECT  o.id,  o.user\_id,  o.number,  o.createtime,  o.note,  u.username,  u.address  FROM  orders o  LEFT JOIN `user` u ON o.user\_id = u.id |

#### 修改订单pojo，建立订单到用户的一对一关联关系



#### 声明mapper接口方法

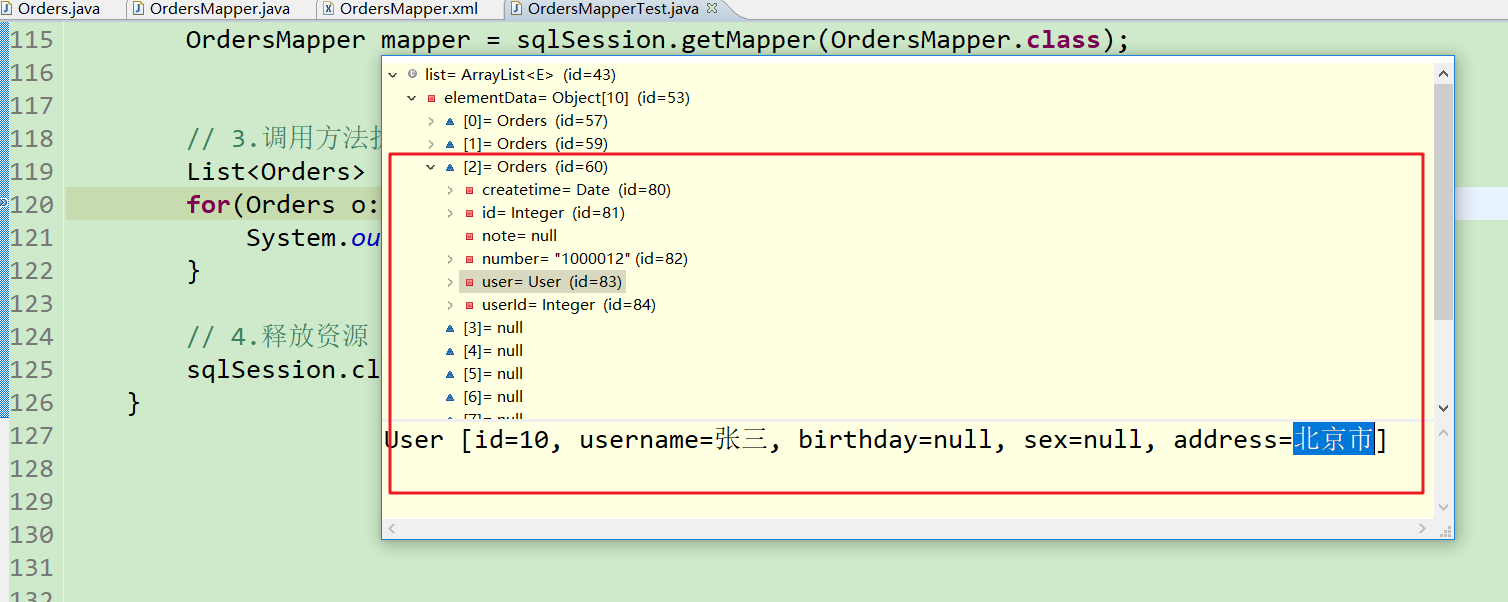
|  |
| --- |
| **package** cn.itheima.mapper;  **import** java.util.List;  **import** cn.itheima.po.Orders;  /\*\*  \* **@ClassName**: OrdersMapper  \* **@Description**: 订单mapper接口  \* **@author** 传智 小杨老师  \* **@date** 2018-8-12 下午3:44:21  \*  \*/  **public** **interface** OrdersMapper {    /\*\*  \* 查询全部订单数据  \*/  List<Orders> queryAllOrders();    /\*\*  \* 查询全部订单数据，使用ResultMap实现  \*/  List<Orders> queryAllOrdersByResultMap();    /\*\*  \* 查询全部订单数据，并且关联查询出订单所属的用户数据  \*/  List<Orders> queryAllOrdersAndUser();  } |

#### 配置mapper映射文件

|  |
| --- |
| <!-- 配置订单到用户的一对一关联关系，说明：  type:要映射的类型  id：唯一标识名称，通过id引用该resultMap  -->  <resultMap type=*"orders"* id=*"ordersUserResultMap"*>  <!-- 配置订单的主键字段对应关系 -->  <id column=*"id"* property=*"id"*/>  <!-- 配置普通字段对应关系 -->  <result column=*"user\_id"* property=*"userId"*/>  <result column=*"number"* property=*"number"*/>  <result column=*"createtime"* property=*"createtime"*/>  <result column=*"note"* property=*"note"*/>    <!-- association：配置一对一关联关系，说明：  property：要映射的属性名称  javaType：要映射的属性类型（必须要指定）  -->  <association property=*"user"* javaType=*"User"*>  <!-- 配置用户的主键字段对应关系 -->  <id column=*"user\_id"* property=*"id"*/>  <!-- 配置用户的普通字段对应关系 -->  <result column=*"username"* property=*"username"*/>  <result column=*"address"* property=*"address"*/>  </association>    </resultMap>    <!--查询全部订单数据，并且关联查询出订单所属的用户数据 -->  <select id=*"queryAllOrdersAndUser"* resultMap=*"ordersUserResultMap"*>  SELECT  o.id,  o.user\_id,  o.number,  o.createtime,  o.note,  u.username,  u.address  FROM  orders o  LEFT JOIN `user` u ON o.user\_id = u.id  </select> |

#### 测试

|  |
| --- |
| /\*\*  \* 测试查询全部订单数据，并且关联查询出订单所属的用户数据  \*/  @Test  **public** **void** queryAllOrdersAndUserTest(){  // 1.创建sqlSession  SqlSession sqlSession = **this**.sqlSessionFactory.openSession();    // 2.获取mapper接口代理对象  OrdersMapper mapper = sqlSession.getMapper(OrdersMapper.**class**);      // 3.调用方法执行  List<Orders> list = mapper.queryAllOrdersAndUser();  **for**(Orders o:list){  System.*out*.println(o);  }    // 4.释放资源  sqlSession.close();  } |



## 一对多关联查询

### 需求

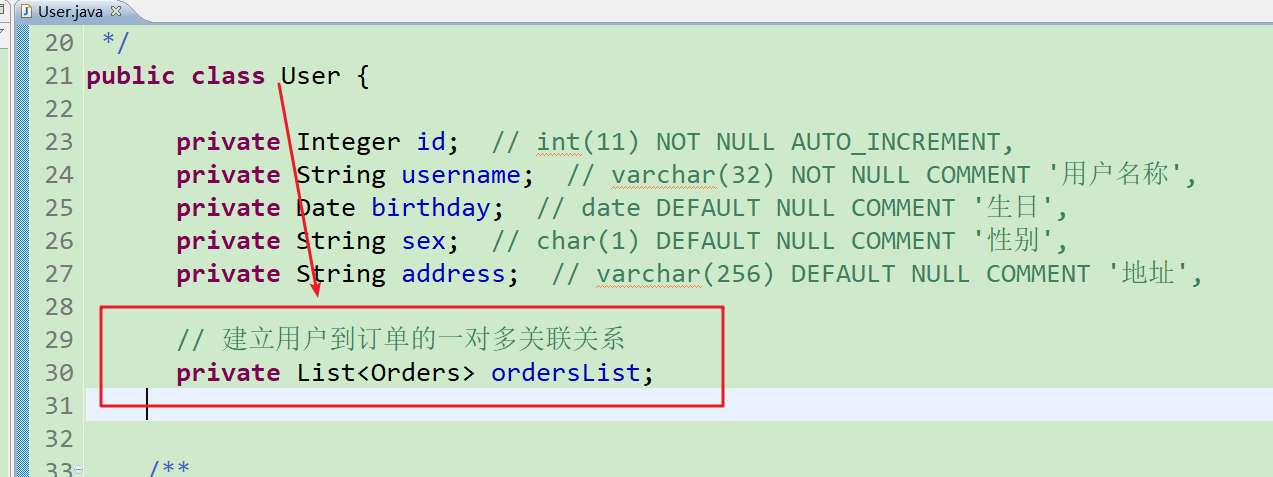
查询全部用户数据。并且关联查询出用户的全部订单数据。

### 需求实现

#### 编写sql语句

|  |
| --- |
| SELECT  u.id,  u.username,  u.birthday,  u.sex,  u.address,  o.id oid,  o.number,  o.createtime  FROM  `user` u  LEFT JOIN orders o ON u.id = o.user\_id |

#### 修改用户实体类，建立用户到订单的一对多关联关系



#### 声明mapper接口方法

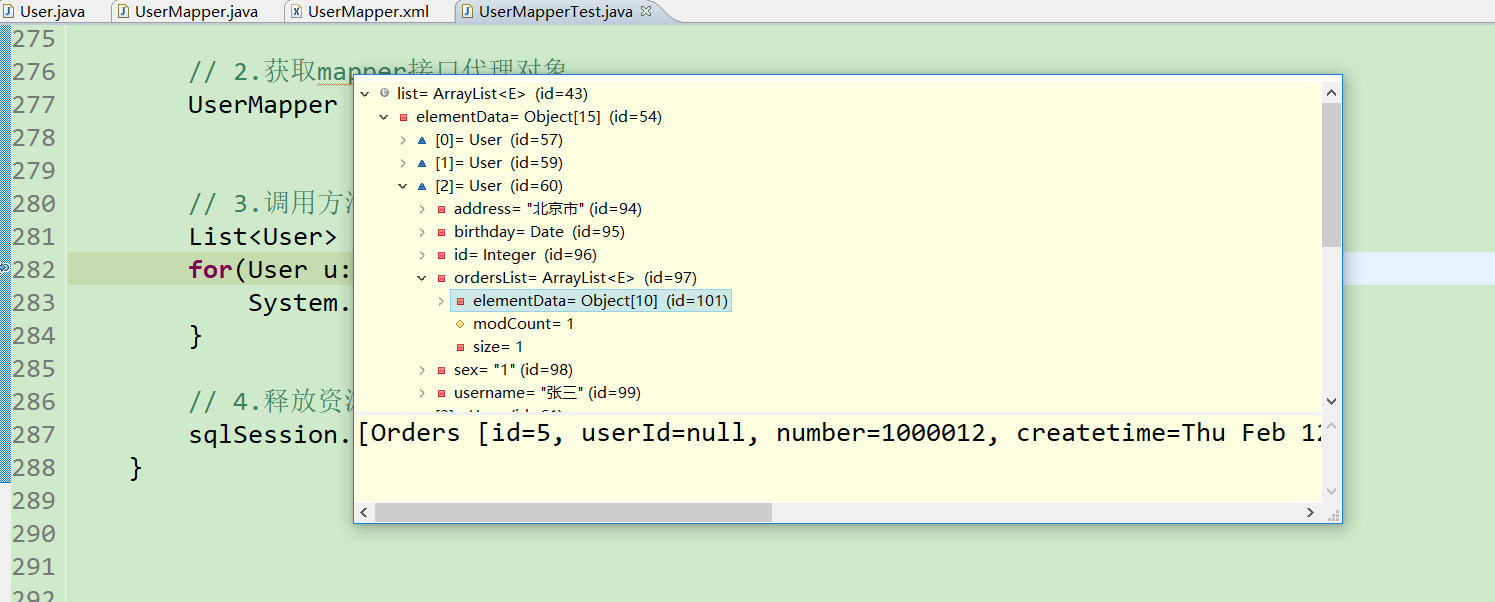
|  |
| --- |
| **package** cn.itheima.mapper;  **import** java.util.List;  **import** cn.itheima.po.QueryVo;  **import** cn.itheima.po.User;  /\*\*  \* **@ClassName**: UserMapper  \* **@Description**: 用户mapper接口  \* **@author** 传智 小杨老师  \* **@date** 2018-8-11 下午5:56:48  \*  \*/  **public** **interface** UserMapper {    /\*\*  \* 1.根据用户Id查询用户  \*/  User queryUserById(Integer id);    /\*\*  \* 2.新增用户  \*/  **void** insertUser(User user);    /\*\*  \* 3.使用pojo包装类型，实现根据用户名称模糊查询用户  \*/  List<User> queryUserByCondition(QueryVo queryVo);    /\*\*  \* 4.统计用户数量  \*/  Integer countUsers();    /\*\*  \* 5.根据用户名称和性别查询用户  \*/  List<User> queryUserByNameAndSex(User user);    /\*\*  \* 6.动态修改用户数据  \*/  **void** dynamicUpdateUser(User user);    /\*\*  \* 7.批量新增用户  \*/  **void** batchInsertUser(List<User> list);    /\*\*  \* 8.批量删除用户  \*/  **void** batchDeleteUser(Integer[] ids);    /\*\*  \* 9.查询全部用户数据，并且关联查询出用户的所有订单数据  \*/  List<User> queryAllUsersAndOrders();  } |

#### 配置mapper映射文件

|  |
| --- |
| <!-- 配置用户到订单的一对多关联关系，说明：  type：要映射的类型  id：唯一标识的名称，通过id引用该resultMap  -->  <resultMap type=*"user"* id=*"userOrdersResultMap"*>  <!-- 配置用户的主键字段对应关系 -->  <id column=*"id"* property=*"id"*/>  <!-- 配置用户的普通字段的对应关系 -->  <result column=*"username"* property=*"username"*/>  <result column=*"birthday"* property=*"birthday"*/>  <result column=*"sex"* property=*"sex"*/>  <result column=*"address"* property=*"address"*/>    <!-- collection：配置一对多关联关系，说明：  property：要映射的属性名称  javaType：要映射的类型（可以指定，可以不指定，建议都指定）  ofType：集合中存放的类型（必须要指定）  -->  <collection property=*"ordersList"* javaType=*"List"* ofType=*"Orders"*>  <!-- 配置订单的主键字段对应关系 -->  <id column=*"oid"* property=*"id"*/>  <!-- 配置订单的普通字段对应关系 -->  <result column=*"number"* property=*"number"*/>  <result column=*"createtime"* property=*"createtime"*/>  </collection>    </resultMap>      <!-- 查询全部用户数据，并且关联查询出用户的所有订单数据 -->  <select id=*"queryAllUsersAndOrders"* resultMap=*"userOrdersResultMap"*>  SELECT  u.id,  u.username,  u.birthday,  u.sex,  u.address,  o.id oid,  o.number,  o.createtime  FROM  `user` u  LEFT JOIN orders o ON u.id = o.user\_id  </select> |

#### 测试

|  |
| --- |
| /\*\*  \* 测试查询全部用户数据，并且关联查询出用户的所有订单数据  \*/  @Test  **public** **void** queryAllUsersAndOrdersTest(){  // 1.创建SqlSession  SqlSession sqlSession = **this**.sqlSessionFactory.openSession();    // 2.获取mapper接口代理对象  UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);      // 3.调用方法执行  List<User> list = mapper.queryAllUsersAndOrders();  **for**(User u:list){  System.*out*.println(u);  }    // 4.释放资源  sqlSession.close();  } |

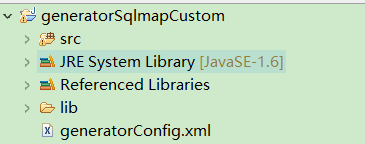


# mybatis逆向工程【了解】

## 什么是逆向工程

根据数据库表，反向生成java代码和配置文件（mapper接口映射文件）。

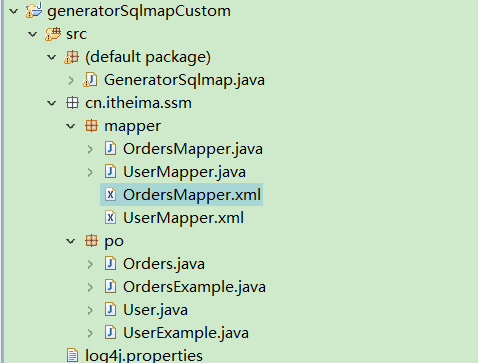
## 导入逆向工程



## 配置逆向工程

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE generatorConfiguration  PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"  "http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd">  <generatorConfiguration>  <context id=*"testTables"* targetRuntime=*"MyBatis3"*>    <!-- 配置生成toString方法的插件 -->  <plugin type=*"org.mybatis.generator.plugins.ToStringPlugin"*></plugin>    <commentGenerator>  <!-- 是否去除自动生成的注释 true：是 ： false:否 -->  <property name=*"suppressAllComments"* value=*"true"* />  </commentGenerator>    <!--1.数据库连接的信息：驱动类、连接地址、用户名、密码 -->  <jdbcConnection driverClass=*"com.mysql.jdbc.Driver"*  connectionURL=*"jdbc:mysql://localhost:3306/82\_mybatis"* userId=*"root"*  password=*"admin"*>  </jdbcConnection>  <!-- 默认false，把JDBC DECIMAL 和 NUMERIC 类型解析为 Integer，为 true时把JDBC DECIMAL 和  NUMERIC 类型解析为java.math.BigDecimal -->  <javaTypeResolver>  <property name=*"forceBigDecimals"* value=*"false"* />  </javaTypeResolver>  <!-- 2.targetPackage:生成PO类的位置 -->  <javaModelGenerator targetPackage=*"cn.itheima.ssm.po"*  targetProject=*".\src"*>  <!-- enableSubPackages:是否让schema作为包的后缀 -->  <property name=*"enableSubPackages"* value=*"false"* />  <!-- 从数据库返回的值被清理前后的空格 -->  <property name=*"trimStrings"* value=*"true"* />  </javaModelGenerator>    <!-- 3.targetPackage:mapper映射文件生成的位置 -->  <sqlMapGenerator targetPackage=*"cn.itheima.ssm.mapper"*  targetProject=*".\src"*>  <!-- enableSubPackages:是否让schema作为包的后缀 -->  <property name=*"enableSubPackages"* value=*"false"* />  </sqlMapGenerator>    <!-- 4.targetPackage：mapper接口生成的位置 -->  <javaClientGenerator type=*"XMLMAPPER"*  targetPackage=*"cn.itheima.ssm.mapper"*  targetProject=*".\src"*>  <!-- enableSubPackages:是否让schema作为包的后缀 -->  <property name=*"enableSubPackages"* value=*"false"* />  </javaClientGenerator>    <!-- 5.指定数据库表 -->  <table schema=*""* tableName=*"user"*></table>  <table schema=*""* tableName=*"orders"*></table>    </context>  </generatorConfiguration> |

## 执行逆向工程



说明：

1. 逆向工程生成的是mapper代理开发的方法
2. 逆向工程生成的都是单表的CRUD操作
3. 逆向工程生成的文件建议直接使用，不推荐进行修改。如果不能满足业务需求，建议创建新的mapper接口和映射文件
4. 使用逆向的优点是提升开发效率
5. 执行逆向工程，需要先将上一次生成的文件删除。再重新执行生成。

# mybatis+spring整合

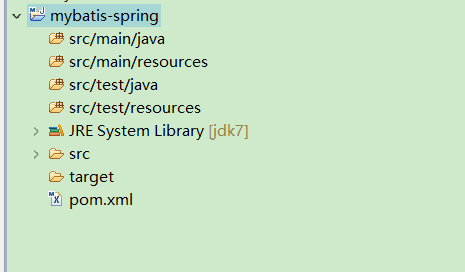
## 整合思路

思路：把mybatis框架的对象交给spring管理

1. 把SqlSessionFactory对象，交给spring管理
2. 把SqlSession对象交给spring管理
3. 在原始dao开发方法中，把dao实现类对象，交给spring管理
4. 在mapper代理开发方法中，把mapper代理对象交给spring管理
5. 把数据源对象交给spring管理

## 整合步骤

### 创建整合项目



### 配置pom.xml文件，加入jar包

* spring框架包
* mybatis框架包
* mybatis-spring整合包
* 数据库驱动包
* 连接池包（dbcp）

|  |
| --- |
| <project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>  <modelVersion>4.0.0</modelVersion>    <groupId>cn.itheima</groupId>  <artifactId>mybatis-spring</artifactId>  <version>0.0.1-SNAPSHOT</version>    <packaging>jar</packaging>  <name>mybatis-spring</name>  <url>http://maven.apache.org</url>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <!-- mysql版本 -->  <mysql.version>5.1.30</mysql.version>  <!-- junit版本 -->  <junit.version>4.12</junit.version>  <!-- mybatis版本号 -->  <mybatis.version>3.4.5</mybatis.version>  <!-- log4j日志包版本 -->  <slf4j.version>1.7.7</slf4j.version>  <log4j.version>1.2.17</log4j.version>  <!-- dbcp数据源连接池jar包 -->  <dbcp.version>1.2.2</dbcp.version>  <!-- spring版本 -->  <spring.version>4.3.8.RELEASE</spring.version>  <!-- mybatis-spring整合包版本 -->  <mybatis.spring.version>1.3.1</mybatis.spring.version>  </properties>  <dependencies>  <!-- mysql数据库依赖 -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>${mysql.version}</version>  </dependency>  <!-- mybatis核心包 -->  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>${mybatis.version}</version>  </dependency>  <!-- log4j日志包 -->  <dependency>  <groupId>log4j</groupId>  <artifactId>log4j</artifactId>  <version>${log4j.version}</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>${slf4j.version}</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  <version>${slf4j.version}</version>  </dependency>  <!-- spring框架包 -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-tx</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jdbc</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aop</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context-support</artifactId>  <version>${spring.version}</version>  </dependency>  <!-- mybatis-spring整合包 -->  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis-spring</artifactId>  <version>${mybatis.spring.version}</version>  </dependency>  <!-- 导入dbcp数据源连接池jar包 -->  <dependency>  <groupId>commons-dbcp</groupId>  <artifactId>commons-dbcp</artifactId>  <version>${dbcp.version}</version>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>${junit.version}</version>  <scope>test</scope>  </dependency>  </dependencies>  </project> |

### 准备配置文件

#### sqlMapConfig.xml

* 配置别名
* 加载mapper映射文件（只限定原始的dao开发方法）

|  |
| --- |
| <?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>    <!-- 配置别名 -->  <typeAliases>  <!-- 包扫描方式配置别名 -->  <package name=*"cn.itheima.ssm.po"*/>  </typeAliases>    <!-- 配置映射文件 -->  <mappers>  <mapper resource=*"sqlmap/User.xml"*/>  </mappers>  </configuration> |

#### applicationContext.xml

* 加载db.properties属性文件
* 配置数据源对象（DataSource）
* 配置SqlSessionFactory

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xmlns:tx=*"http://www.springframework.org/schema/tx"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context-4.0.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop-4.0.xsd*  *http://www.springframework.org/schema/tx*  *http://www.springframework.org/schema/tx/spring-tx-4.0.xsd*  *http://www.springframework.org/schema/util*  *http://www.springframework.org/schema/util/spring-util-4.0.xsd"*>    <!-- 配置加载db.properties属性文件 -->  <context:property-placeholder location=*"classpath:db.properties"*/>    <!-- 配置数据库连接池 -->  <bean id=*"dataSource"*  class=*"org.apache.commons.dbcp.BasicDataSource"*  destroy-method=*"close"*>    <property name=*"driverClassName"* value=*"${db.driverClassName}"*/>  <property name=*"url"* value=*"${db.url}"* />  <property name=*"username"* value=*"${db.username}"* />  <property name=*"password"* value=*"${db.password}"* />    <!-- 最大连接数量 -->  <property name=*"maxActive"* value=*"${db.maxActive}"*/>  <!-- 最小空闲连接数量 -->  <property name=*"minIdle"* value=*"${db.minIdle}"*/>  <!-- 最大空闲连接数量 -->  <property name=*"maxIdle"* value=*"${db.maxIdle}"*/>  <!-- 初始化连接数数量 -->  <property name=*"initialSize"* value=*"${db.initialSize}"*/>  <!-- 超时等待时间,以毫秒为单位 -->  <property name=*"maxWait"* value=*"${db.maxWait}"*/>  </bean>  <!-- 配置SqlSessionFactoryBean -->  <bean id=*"sqlSessionFactory"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>  <!-- 注入数据源对象 -->  <property name=*"dataSource"* ref=*"dataSource"*></property>    <!--加载mybatis主配置文件 -->  <property name=*"configLocation"* value=*"classpath:mybatis/sqlMapConfig.xml"*></property>  </bean>  </beans> |

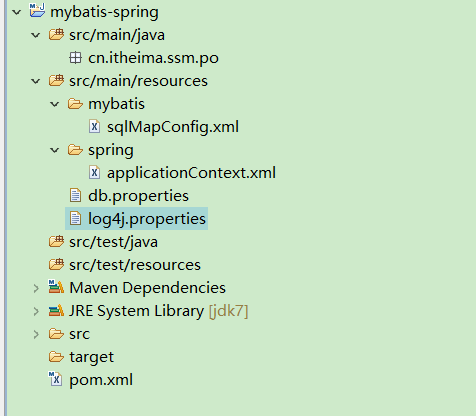
#### db.properties

|  |
| --- |
| db.driverClassName=com.mysql.jdbc.Driver  db.url=jdbc:mysql://127.0.0.1:3306/82\_mybatis  db.username=root  db.password=admin  db.maxActive=10  db.minIdle=2  db.maxIdle=5  db.initialSize=5  db.maxWait=6000 |

#### log4j.properties

|  |
| --- |
| # Global logging configuration  log4j.rootLogger=DEBUG, stdout  # Console output...  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n |

### 整合好的项目



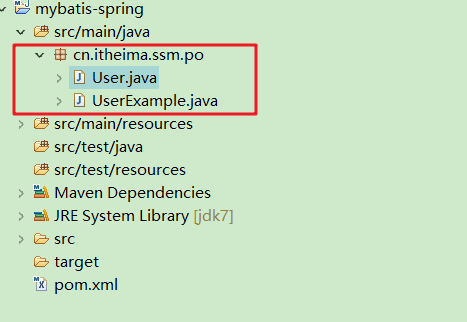
## 原始的dao开发方法

### 需求

1. 根据用户Id查询用户
2. 新增用户

### 需求实现

#### 编写用户实体类



#### 编写User.xml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">  <!--namespace：名称空间，相当于java中的package，用于防止sql语句名称冲突（sql语句隔离） -->  <mapper namespace=*"test"*>  <!-- 根据用户id查询用户 -->  <select id=*"queryUserById"* parameterType=*"int"* resultType=*"user"*>  select \* from `user` where id=#{id}  </select>    <!-- 新增用户 -->  <insert id=*"insertUser"* parameterType=*"user"*  useGeneratedKeys=*"true"* keyColumn=*"id"* keyProperty=*"id"*>    insert into `user`(username,birthday,sex,address)  values(#{username},#{birthday},#{sex},#{address})  </insert>  </mapper> |

#### 编写用户dao接口

|  |
| --- |
| **package** cn.itheima.ssm.dao;  **import** cn.itheima.ssm.po.User;  /\*\*  \* **@ClassName**: UserDao  \* **@Description**: 用户dao接口  \* **@author** 传智 小杨老师  \* **@date** 2018-8-12 下午7:20:21  \*  \*/  **public** **interface** UserDao {    /\*\*  \* 根据用户Id查询用户  \*/  User queryUserById(Integer id);    /\*\*  \* 新增用户  \*/  **void** insertUser(User user);  } |

#### 编写用户dao实现类

说明：整合包提供了一个SqlSessionDaoSupport类，我们的dao实现类需要继承SqlSessionDaoSupport，获取SqlSession对象。

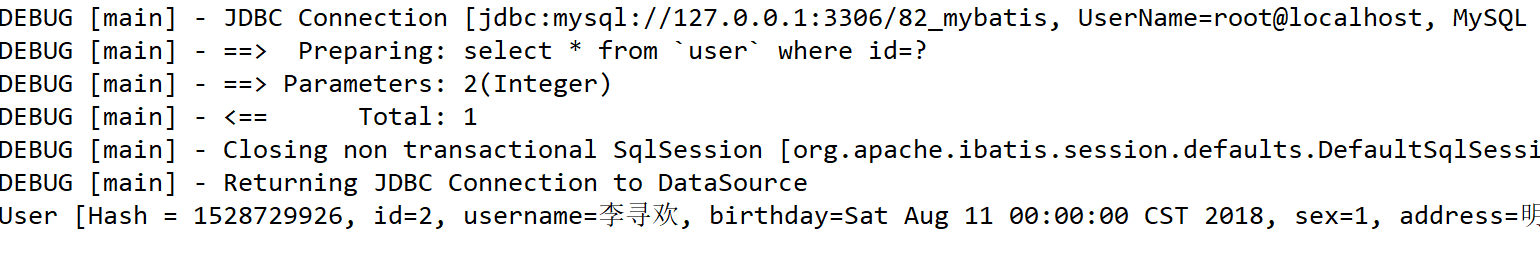
|  |
| --- |
| **package** cn.itheima.ssm.dao.impl;  **import** org.apache.ibatis.session.SqlSession;  **import** org.mybatis.spring.support.SqlSessionDaoSupport;  **import** cn.itheima.ssm.dao.UserDao;  **import** cn.itheima.ssm.po.User;  /\*\*  \* **@ClassName**: UserDaoImpl  \* **@Description**: 用户dao实现类  \* **@author** 传智 小杨老师  \* **@date** 2018-8-12 下午7:22:58  \*  \*/  **public** **class** UserDaoImpl **extends** SqlSessionDaoSupport **implements** UserDao {  /\*\*  \* 根据用户Id查询用户  \*/  **public** User queryUserById(Integer id) {  // 1.获取SqlSession对象  SqlSession sqlSession = **this**.getSqlSession();    // 2.执行数据库操作  Object user = sqlSession.selectOne("test.queryUserById", id);    /\*\*  \* 与spring整合以后，SqlSession对象交给spring管理，不需要在手动释放（不能）  \*/    **return** (User) user;  }  /\*\*  \* 新增用户  \*/  **public** **void** insertUser(User user) {  // 1.获取SqlSession对象  SqlSession sqlSession = **this**.getSqlSession();    // 2.执行数据库操作  sqlSession.insert("test.insertUser", user);    /\*\*  \* 1.与spring整合以后，SqlSession对象交给spring管理，不需要在手动释放（不能）  \* 2.与spring整合以后，SqlSession对象交给spring管理，不需要在关心事务（  \* 如果配置了spring的事务，使用spring的事务；如果没有配置，默认使用jdbc事务  \* ）  \*/  }  } |

#### 在applicationContext.xml配置dao实现类对象

|  |
| --- |
| <!-- 配置用户dao对象 -->  <bean id=*"userDao"* class=*"cn.itheima.ssm.dao.impl.UserDaoImpl"*>  <!-- 注入sqlSessionFactory -->  <property name=*"sqlSessionFactory"* ref=*"sqlSessionFactory"*></property>  </bean> |

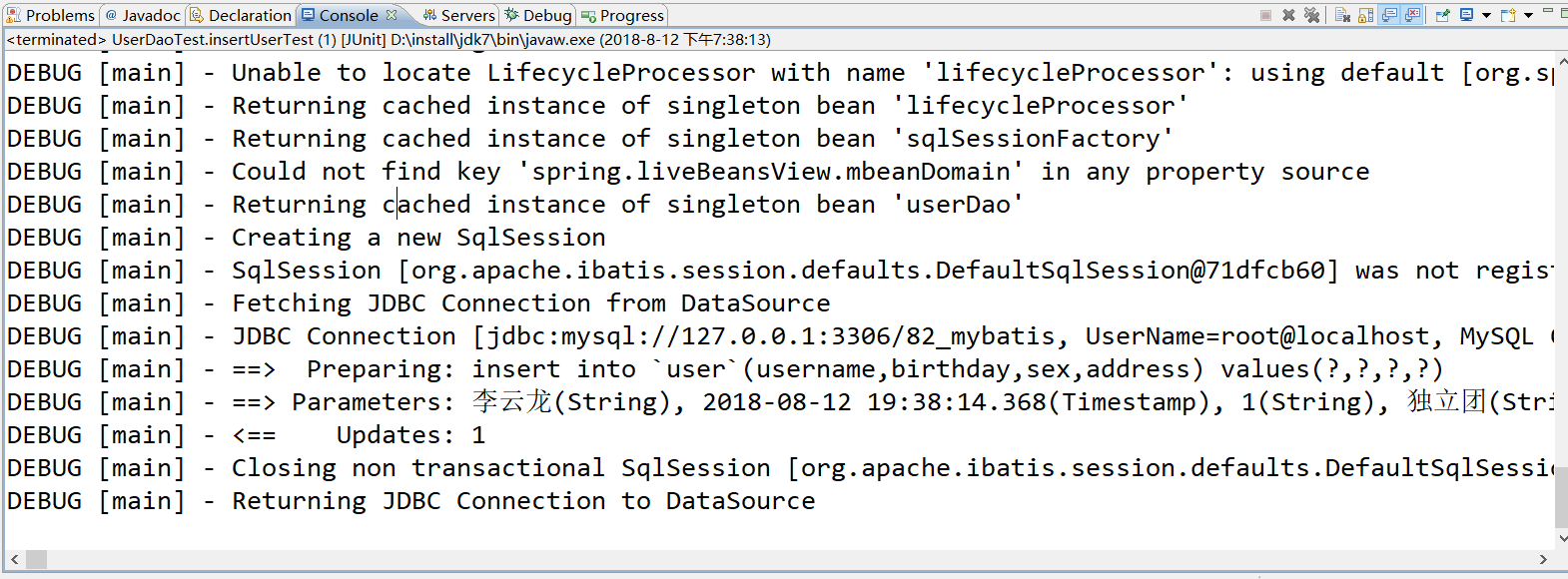
#### 测试

|  |
| --- |
| **package** cn.itheima.ssm.test;  **import** org.junit.Test;  **import** org.springframework.context.ApplicationContext;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **import** cn.itheima.ssm.dao.UserDao;  **import** cn.itheima.ssm.po.User;  /\*\*  \* **@ClassName**: UserDaoTest  \* **@Description**: 原始的dao开发方法测试  \* **@author** 传智 小杨老师  \* **@date** 2018-8-12 下午7:32:25  \*  \*/  **public** **class** UserDaoTest {    /\*\*  \* 测试根据用户Id查询用户  \*/  @Test  **public** **void** queryUserByIdTest(){    // 1.加载spring配置文件，创建spring容器  ApplicationContext context =  **new** ClassPathXmlApplicationContext("classpath:spring/applicationContext.xml");    // 2.取出UserDao对象  UserDao userDao =(UserDao) context.getBean("userDao");    // 3.调用方法执行  User user = userDao.queryUserById(2);  System.*out*.println(user);    }  } |



测试新增用户：

|  |
| --- |
| /\*\*  \* 测试新增用户  \*/  @Test  **public** **void** insertUserTest(){  // 1.加载spring配置文件，创建spring容器  ApplicationContext context =  **new** ClassPathXmlApplicationContext("classpath:spring/applicationContext.xml");    // 2.取出UserDao对象  UserDao userDao =(UserDao) context.getBean("userDao");    // 3.调用方法执行  // 创建用户对象  User user = **new** User();  user.setUsername("李云龙");  user.setSex("1");  user.setBirthday(**new** Date());  user.setAddress("独立团");    userDao.insertUser(user);  } |



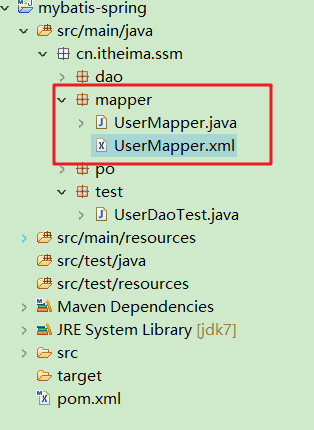
## mapper代理开发方法

### 需求

1. 根据用户名称模糊查询用户
2. 新增用户

### 需求实现

#### 使用逆向工程生成的mapper接口和映射文件



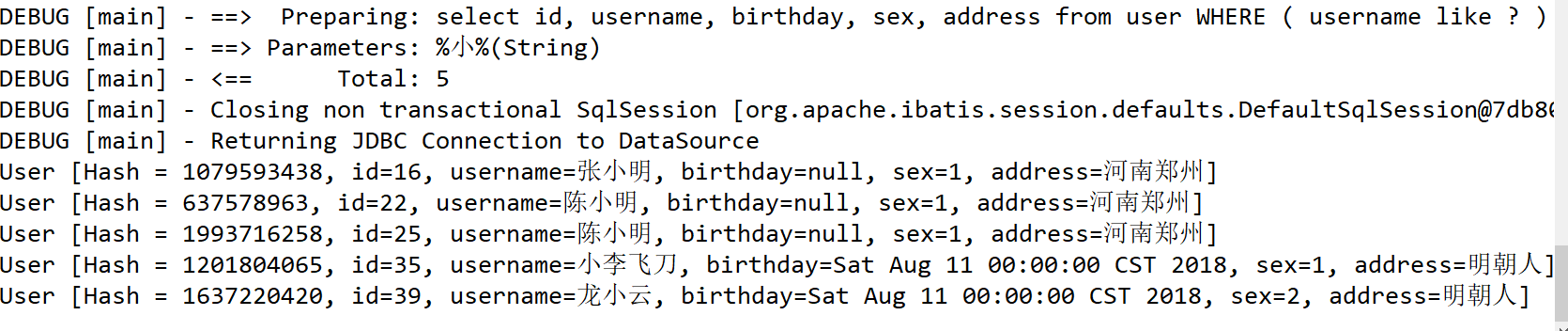
#### 准备测试

说明：整合包提供了一个类（MapperFactoryBean），配置mapper代理对象。

|  |
| --- |
| <!-- 配置用户mapper代理对象 -->  <bean id=*"userMapper"* class=*"org.mybatis.spring.mapper.MapperFactoryBean"*>  <!-- 注入sqlSessionFactory -->  <property name=*"sqlSessionFactory"* ref=*"sqlSessionFactory"*></property>  <!-- 注入被代理接口 -->  <property name=*"mapperInterface"* value=*"cn.itheima.ssm.mapper.UserMapper"*></property>  </bean> |

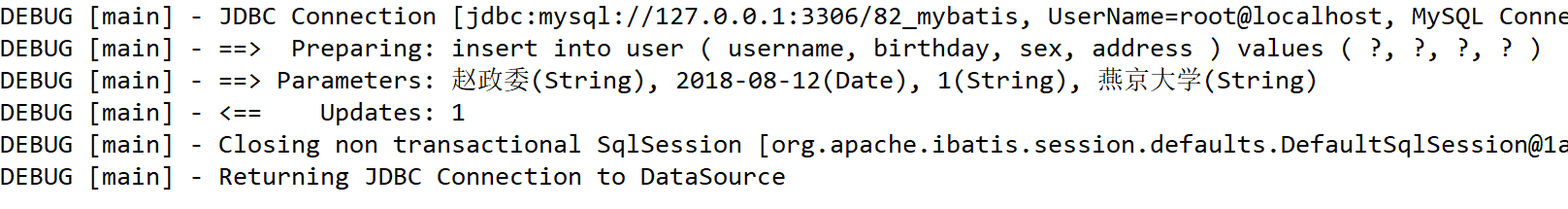
#### 测试

|  |
| --- |
| **package** cn.itheima.ssm.test;  **import** java.util.List;  **import** org.junit.Test;  **import** org.springframework.context.ApplicationContext;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **import** cn.itheima.ssm.mapper.UserMapper;  **import** cn.itheima.ssm.po.User;  **import** cn.itheima.ssm.po.UserExample;  **import** cn.itheima.ssm.po.UserExample.Criteria;  /\*\*  \* **@ClassName**: UserMapperTest  \* **@Description**: mapper代理开发方法测试  \* **@author** 传智 小杨老师  \* **@date** 2018-8-12 下午7:47:06  \*  \*/  **public** **class** UserMapperTest {    /\*\*  \* 根据用户名称模糊查询用户  \*/  @Test  **public** **void** queryUserByNameTest(){    // 1.加载spring的配置文件，创建spring容器  ApplicationContext context =  **new** ClassPathXmlApplicationContext("classpath:spring/applicationContext.xml");      // 2.获取mapper代理对象  UserMapper mapper =(UserMapper) context.getBean("userMapper");  // 3.调用方法执行(条件：名称)  // 3.1.创建辅助类对象  UserExample example = **new** UserExample();    // 3.2.通过辅助类对象，创建Criteria对象  Criteria cri = example.createCriteria();    // 3.3.通过Criteria对象，调用andXXX方法设置条件  cri.andUsernameLike("%小%");    List<User> list = mapper.selectByExample(example);    **for**(User u:list){  System.*out*.println(u);  }    }  } |



测试新增用户：

|  |
| --- |
| /\*\*  \* 测试新增用户  \*/  @Test  **public** **void** insertUserTest(){  // 1.加载spring的配置文件，创建spring容器  ApplicationContext context =  **new** ClassPathXmlApplicationContext("classpath:spring/applicationContext.xml");      // 2.获取mapper代理对象  UserMapper mapper =(UserMapper) context.getBean("userMapper");  // 3.调用方法执行  // 创用户对象  User user = **new** User();  user.setUsername("赵政委");  user.setSex("1");  user.setBirthday(**new** Date());  user.setAddress("燕京大学");    mapper.insertSelective(user);  } |



#### mapper代理对象配置方式二【掌握】

|  |
| --- |
| <!-- mapper代理对象配置方式二（mapper扫描器）[掌握] -->  <bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>  <!-- 配置要扫描的包,说明：  1.如果有多个包，在同一个父包下，配置父包即可  2.如果不在同一个父包下，以半角逗号进行分割：,  3.在企业项目中推荐使用的方式  -->  <property name=*"basePackage"* value=*"cn.itheima.ssm.mapper"*></property>    </bean> |