mybatis

**第二天**

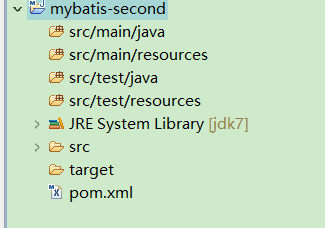
# 课程计划

1. ：mybatis高级知识
2. mybatis输入输出映射
3. mybatis动态sql
4. mybatis关联查询
5. mybatis逆向工程
6. mybatis+spring整合

# mybatis输入输出映射

## 准备环境

### 创建工程



### 配置pom.xml，加入依赖

|  |
| --- |
| <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.mybatis.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.mybatis.mapper"*/>  </mappers>  </configuration> |

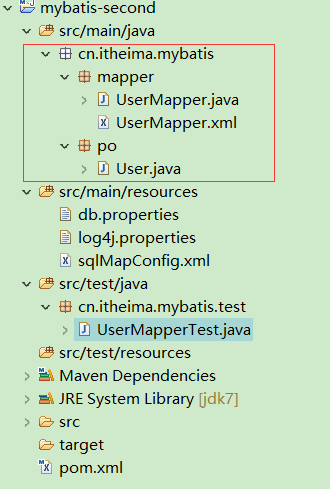
#### db.properties

|  |
| --- |
| db.driver=com.mysql.jdbc.Driver  db.url=jdbc:mysql://127.0.0.1:3306/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 |

### 复制用户pojo，mapper接口和映射文件

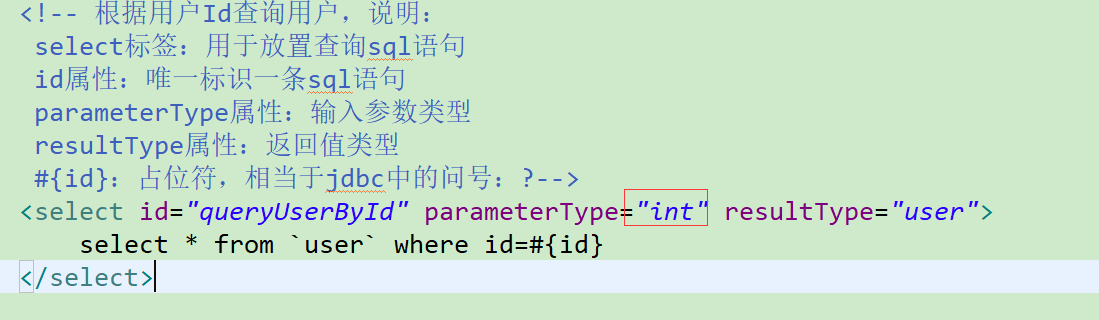


## ParameterType输入映射

说明：企业项目中常用的参数类型

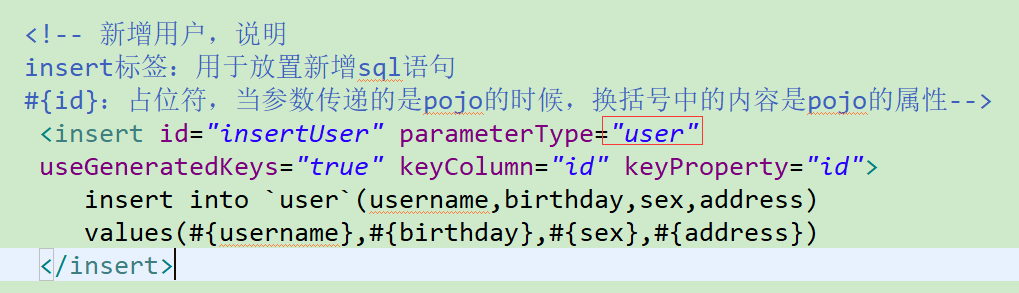
### java简单类型

参考第一天课程内容：



### pojo类型

参考第一天课程内容：



### pojo包装类型

说明：pojo包装类型，就是在pojo中包含了其他的pojo。一般用于综合查询条件参数的接收。



#### 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

##### 准备pojo包装类型

|  |
| --- |
| **public** **class** QueryVo {    // 包装用户  **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;  }      } |

##### 准备sql语句

|  |
| --- |
| select \* from `user` where username like '%小%' |

##### 配置映射文件

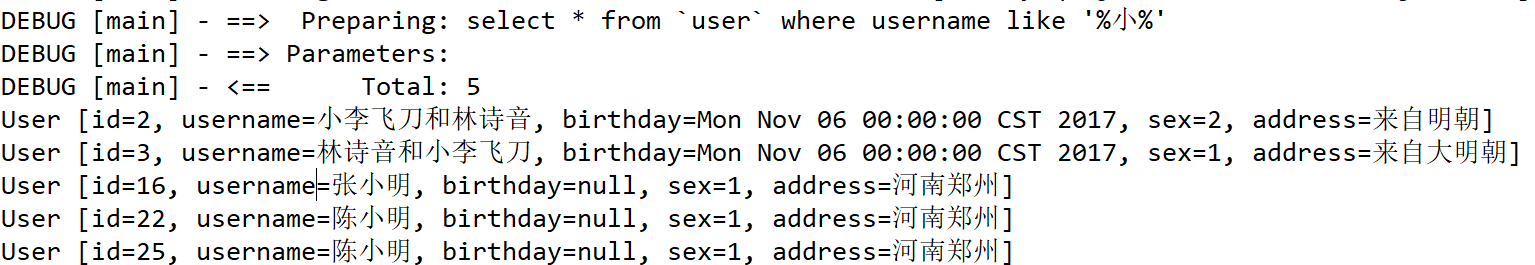
|  |
| --- |
| <!-- 使用pojo包装类型，实现根据用户名称模糊查询用于,说明：  %${user.username}%：字符串拼接符，当参数传递的是pojo的时候，花括号中的内容  是pojo的属性 -->  <select id=*"queryUserByCondition"* parameterType=*"queryVo"* resultType=*"user"*>  select \* from `user` where username like '%${user.username}%'  </select> |

##### 声明接口方法

|  |
| --- |
| **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);    // 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);  } |

##### 编写测试代码

|  |
| --- |
| // 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对象  SqlSession sqlSession = **this**.sqlSessionFactory.openSession();    // 2.使用sqlSession对象，获取mapper代理对象  UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);    // 3.使用mapper对象，调用方法执行  // 创建包装类型pojo  QueryVo queryVo = **new** QueryVo();    // 创建用户对象  User user = **new** User();  user.setUsername("小");    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：企业项目开发中常用的类型。

### java简单类型

#### 需求

统计用户数量。

#### 需求实现

##### 准备sql语句

|  |
| --- |
| select count(\*) from `user` |

##### 配置映射文件

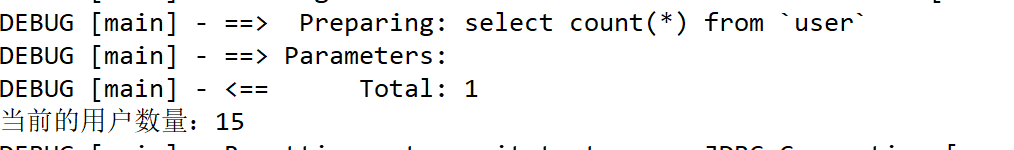
|  |
| --- |
| <!-- 统计用户数量 -->  <select id=*"countUsers"* resultType=*"int"*>  select count(\*) from `user`  </select> |

##### 声明接口方法

|  |
| --- |
| **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();  } |

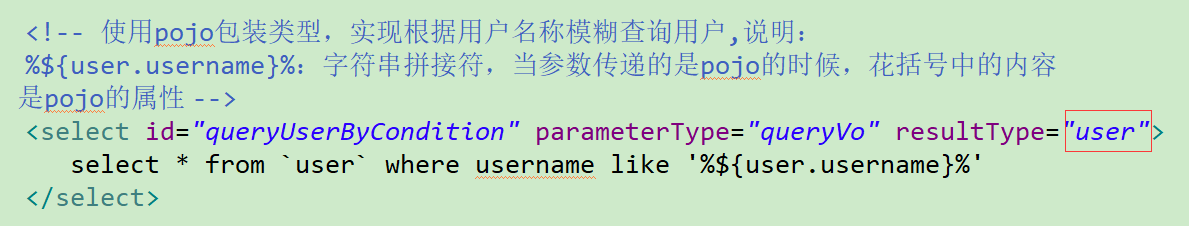
##### 编写测试代码

|  |
| --- |
| // 测试统计用户数量  @Test  **public** **void** countUsersTest(){  // 1.创建sqlSession对象  SqlSession sqlSession = **this**.sqlSessionFactory.openSession();    // 2.使用sqlSession对象，获取mapper代理对象  UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);    // 3.使用mapper对象，调用方法执行  Integer users = mapper.countUsers();  System.*out*.println("当前的用户数量："+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

#### 准备订单pojo

|  |
| --- |
| **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 + "]";  }      } |

#### 准备订单mapper接口

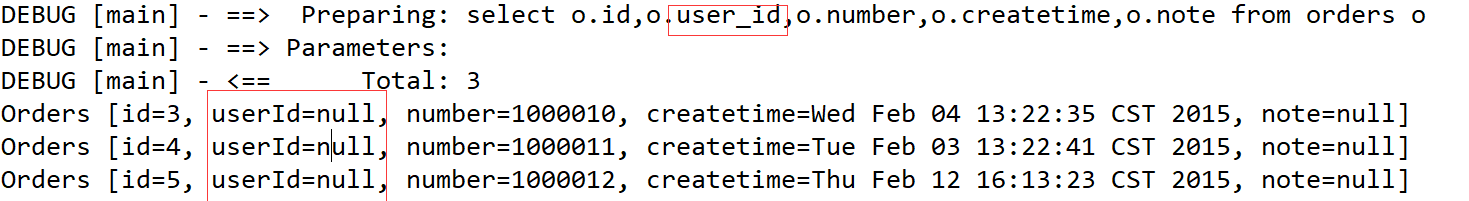
|  |
| --- |
| **public** **interface** OrdersMapper {    // 1.查询全部订单数据  List<Orders> queryAllOrders();  } |

#### 准备订单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属性：名称空间，相当于java中的package，用于放置sql语句冲突 -->  <!--mapper代理开发方法中，namespace属性值，必须是mapper接口的全路径 -->  <mapper namespace=*"cn.itheima.mybatis.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> |

#### 编写测试代码

|  |
| --- |
| **public** **class** OrdersMapperTest {    **private** SqlSessionFactory sqlSessionFactory = **null**;    @Before  **public** **void** init() **throws** IOException{  // 1.加载核心配置文件：sqlMapConfig.xml  // resource属性：指定核心配置文件位置  InputStream inputStream = Resources.*getResourceAsStream*("sqlMapConfig.xml");    // 2.读取配置文件内容  SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();  // SqlSessionFactory对象：mybatis框架的核心对象，一个项目中只需要一个（单例设计模式）  sqlSessionFactory = builder.build(inputStream);  }    // 测试查询全部订单数据  @Test  **public** **void** queryAllOrdersTest(){    // 1.创建sqlSession对象  SqlSession sqlSession = **this**.sqlSessionFactory.openSession();    // 2.使用sqlSession对象，获取mapper对象  OrdersMapper mapper = sqlSession.getMapper(OrdersMapper.**class**);    // 3.使用mapper对象，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();    }  } |



#### 使用resultMap实现

说明：resultMap是用于配置java对象属性，与sql语句的字段对应关系，本质上还是把执行结果映射到java对象上。

##### 配置映射文件

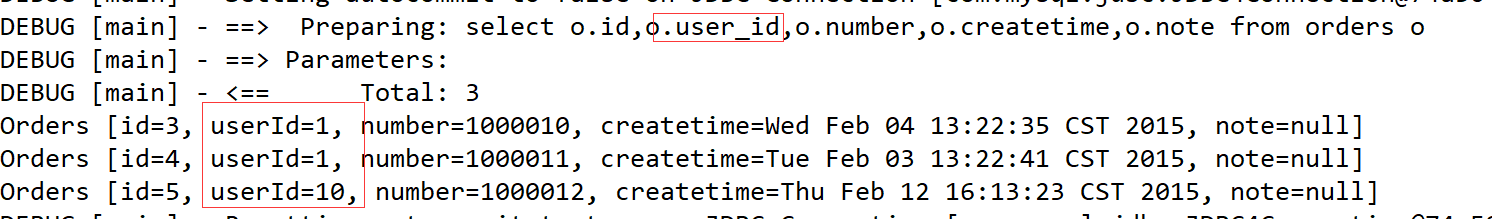
|  |
| --- |
| <!-- resultMap标签：配置java对象属性，与sql语句的字段对应关系，说明：  type属性：映射的类型  id属性：唯一标识，通过id引用该resultMap-->  <resultMap type=*"orders"* id=*"ordersResultMap"*>  <!-- 配置主键字段对应关系，说明：  column属性：主键字段（表）  property属性：主键属性（pojo）-->  <id column=*"id"* property=*"id"*/>    <!-- 配置普通字段对应关系 -->  <result column=*"user\_id"* property=*"userId"*/>  <result column=*"number"* property=*"number"*/>  <result column=*"createtime"* property=*"createtime"*/>  <result column=*"note"* property=*"note"*/>  </resultMap>    <!-- 查询全部订单数据,使用resultMap实现 -->  <select id=*"queryAllOrdersByResultMap"* resultMap=*"ordersResultMap"* >  select o.id,o.user\_id,o.number,o.createtime,o.note from orders o  </select> |

##### 声明接口方法

|  |
| --- |
| **public** **interface** OrdersMapper {    // 1.查询全部订单数据  List<Orders> queryAllOrders();    // 2.查询全部订单数据,使用resultMap实现  List<Orders> queryAllOrdersByResultMap();  } |

##### 编写测试代码

|  |
| --- |
| // 测试查询全部订单数据,使用resultMap实现  @Test  **public** **void** queryAllOrdersByResultMapTest(){    // 1.创建sqlSession对象  SqlSession sqlSession = **this**.sqlSessionFactory.openSession();    // 2.使用sqlSession对象，获取mapper对象  OrdersMapper mapper = sqlSession.getMapper(OrdersMapper.**class**);    // 3.使用mapper对象，调用方法执行  List<Orders> list = mapper.queryAllOrdersByResultMap();  **for**(Orders o:list){  System.*out*.println(o);  }    // 4.释放资源  sqlSession.close();    } |



# mybatis动态sql

说明：if、where、set、sql、foreach

## 需求

根据用户名称和性别查询用户。

## 需求实现

### 准备sql语句

|  |
| --- |
| select \* from `user` where username like '%小%' and sex='1' |

### 配置映射文件

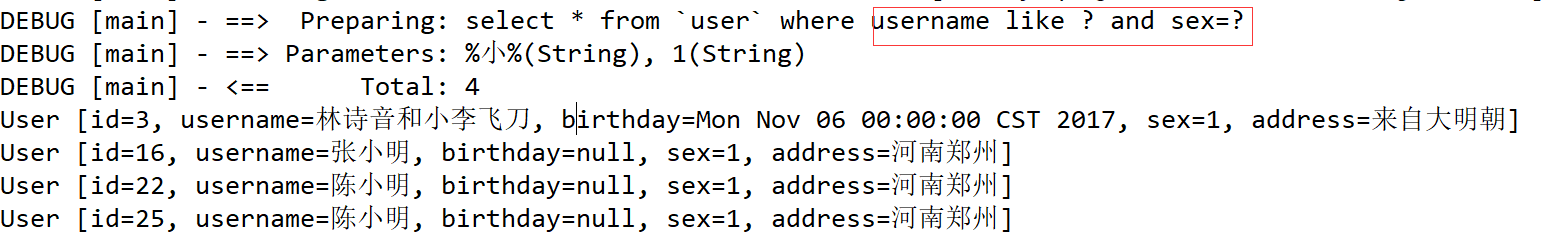
|  |
| --- |
| <!-- 根据用户名称和性别查询用户 -->  <select id=*"queryUserByNameAndSex"* parameterType=*"user"* resultType=*"user"*>  select \* from `user` where username like #{username} and sex=#{sex}  </select> |

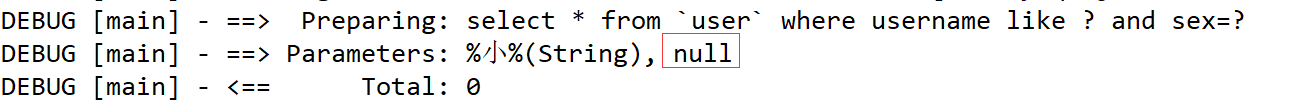
### 声明接口方法

|  |
| --- |
| **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);  } |

### 编写测试代码

|  |
| --- |
| // 测试根据用户名称和性别查询用户  @Test  **public** **void** queryUserByNameAndSexTest(){  // 1.创建sqlSession对象  SqlSession sqlSession = **this**.sqlSessionFactory.openSession();    // 2.使用sqlSession对象，获取mapper代理对象  UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);    // 3.使用mapper对象，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();  } |



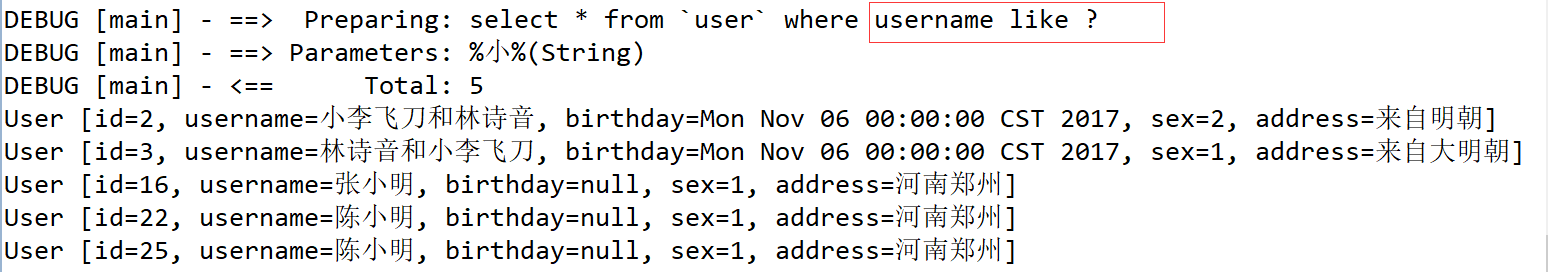


问题：当性别查询条件不传递，mybatis会默认设置一个null值，导致查询不到结果数据。

## if标签

作用：根据传入的参数情况，拼装sql语句。

|  |
| --- |
| <!-- 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标签判断用户名称不为空，且不为空字符串，就作为查询的条件 -->  <if test=*"username !=null and username !=''"*>  username like #{username}  </if>  <!-- if标签判断性别不为空，且不为空字符串，就作为查询的条件 -->  <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. 可以根据传入的参数情况，智能的去掉多余的and，or关键字
3. 可以根据传入的参数情况，智能的去掉多余的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标签判断用户名称不为空，且不为空字符串，就作为查询的条件 -->  <if test=*"username !=null and username !=''"*>  username like #{username}  </if>  <!-- if标签判断性别不为空，且不为空字符串，就作为查询的条件 -->  <if test=*"sex !=null and sex !=''"*>  and sex=#{sex}  </if>  </where>  </select> |

## set标签

作用：

1. set标签相当于sql语句中的set关键字
2. 根据传入的参数情况，智能的去掉多余逗号：”,”

### 需求

动态修改用户数据。

### 需求实现

#### 准备sql语句

|  |
| --- |
| update `user` set username='张君宝',sex='2' where id=24 |

#### 配置映射文件

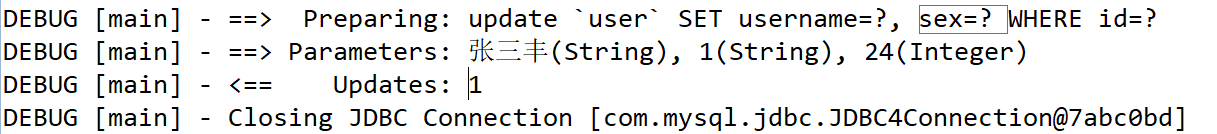
|  |
| --- |
| <!-- 动态修改用户数据 -->  <update id=*"dynamicUserById"* parameterType=*"user"*>  update `user`  <!-- set -->  <set>  <if test=*"username !=null and username !=''"*>  username=#{username},  </if>  <if test=*"sex != null and sex !=''"*>  sex=#{sex},  </if>  </set>  <where>  id=#{id}  </where>  <!-- username='张君宝',sex='2' where id=24 -->  </update> |

#### 声明接口方法

|  |
| --- |
| **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** dynamicUserById(User user);  } |

#### 编写测试代码

|  |
| --- |
| // 测试动态修改用户数据  @Test  **public** **void** dynamicUserByIdTest(){    // 1.创建sqlSession对象  SqlSession sqlSession = **this**.sqlSessionFactory.openSession(**true**);    // 2.使用sqlSession对象，获取mapper代理对象  UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);    // 3.使用mapper对象，调用方法执行  // 创建用户对象  User user = **new** User();  user.setUsername("张三丰");  user.setSex("1");  user.setId(24);    mapper.dynamicUserById(user);    // 4.释放资源  sqlSession.close();    } |



## sql标签（sql片段）

作用：提取公共的sql语句片段

|  |
| --- |
| <!--sql标签：提取公共的sql语句片段，说明：  id属性：唯一标识，通过id引用该sql -->  <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标签：引用公共的sql语句片段，说明：  refid属性：引用的sql片段id-->  select <include refid=*"select\_orders\_list"*></include> from orders o  </select> |

## foreach标签

作用：循环遍历处理参数集合（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

#### 准备sql语句

|  |
| --- |
| insert into `user`(username,birthday,sex,address) values('用户1','2017-11-07','1','用户地址1'),('用户2','2017-11-07','1','用户地址2') |

#### 配置映射文件

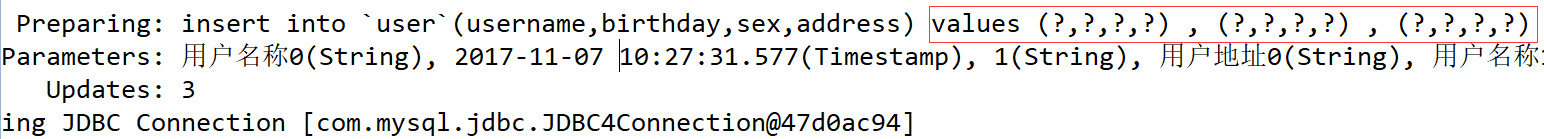
|  |
| --- |
| <!--批量新增用户 -->  <insert id=*"batchInsertUser"* parameterType=*"list"*>  insert into `user`(username,birthday,sex,address)  values  <!--foreach标签：循环处理参数集合  collection属性：参数集合，这里是list  item属性：当前遍历的对象  separator属性：指定分割符-->  <foreach collection=*"list"* item=*"u1"* separator=*","*>  (#{u1.username},#{u1.birthday},#{u1.sex},#{u1.address})  </foreach>  <!-- ('用户1','2017-11-07','1','用户地址1'),('用户2','2017-11-07','1','用户地址2') -->  </insert> |

#### 声明接口方法

|  |
| --- |
| **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** dynamicUserById(User user);    // 7.批量新增用户  **void** batchInsertUser(List<User> list);  } |

#### 编写测试代码

|  |
| --- |
| // 测试批量新增用户  @Test  **public** **void** batchInsertUserTest(){  // 1.创建sqlSession对象  SqlSession sqlSession = **this**.sqlSessionFactory.openSession(**true**);    // 2.使用sqlSession对象，获取mapper代理对象  UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);    // 3.使用mapper对象，调用方法执行  // 创建用户集合List  List<User> list = **new** ArrayList<User>();  **for**(**int** i=0;i<3;i++){  // 创建用户对象  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

#### 准备sql语句

|  |
| --- |
| delete from `user` where id in(50,51) |

#### 配置映射文件

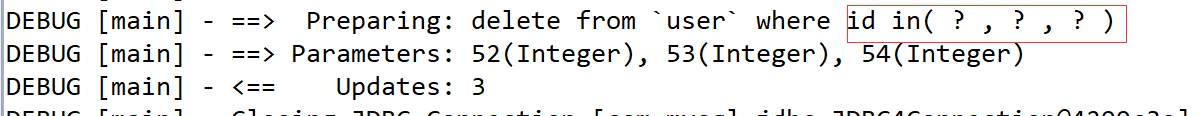
|  |
| --- |
| <!-- 批量删除用户 ,说明：  parameterType属性：当参数传递的是list，和数组的时候，都需要使用list-->  <delete id=*"batchDeleteUser"* parameterType=*"list"*>  delete from `user`  where  <!-- foreach标签：循环处理参数集合  collection属性：参数集合，这里是array  open属性：拼装的sql语句片段的开始部分  close属性：拼装的sql语句片段的结束部分  item属性：当前遍历的元素  separator属性：指定分隔符-->  <foreach collection=*"array"* open=*" id in( "* close=*" )"* item=*"id"* separator=*","*>  #{id}  </foreach>  <!-- id in( 50,51 ) -->  </delete> |

#### 声明接口方法

|  |
| --- |
| **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** dynamicUserById(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);  } |

#### 编写测试代码

|  |
| --- |
| // 测试批量删除用户  @Test  **public** **void** batchDeleteUserTest(){  // 1.创建sqlSession对象  SqlSession sqlSession = **this**.sqlSessionFactory.openSession(**true**);    // 2.使用sqlSession对象，获取mapper代理对象  UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);    // 3.使用mapper对象，调用方法执行  Integer[] ids = {52,53,54};    mapper.batchDeleteUser(ids);    // 4.释放资源  sqlSession.close();  } |



# mybatis关联查询

一对一关联关系、一对多关联关系、多对多关联关系（可以看成两个一对多关联关系）

## 分析用户订单数据模型

用户表：user

用户Id：id

用户名称：username

用户生日：brithday

用户性别：sex

用户地址：address

订单表：orders

订单id：id

所属用户id：user\_id

订单编号：number

订单创建时间：createtime

备注：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

查询订单数据，并且关联查询出所属的用户数据。

### 需求实现

#### 准备sql语句

|  |
| --- |
| SELECT  o.id,  o.user\_id userId,  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，增加用户属性对象

|  |
| --- |
| **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',    // 定义用户对象属性  **private** User user; |

#### 配置映射文件

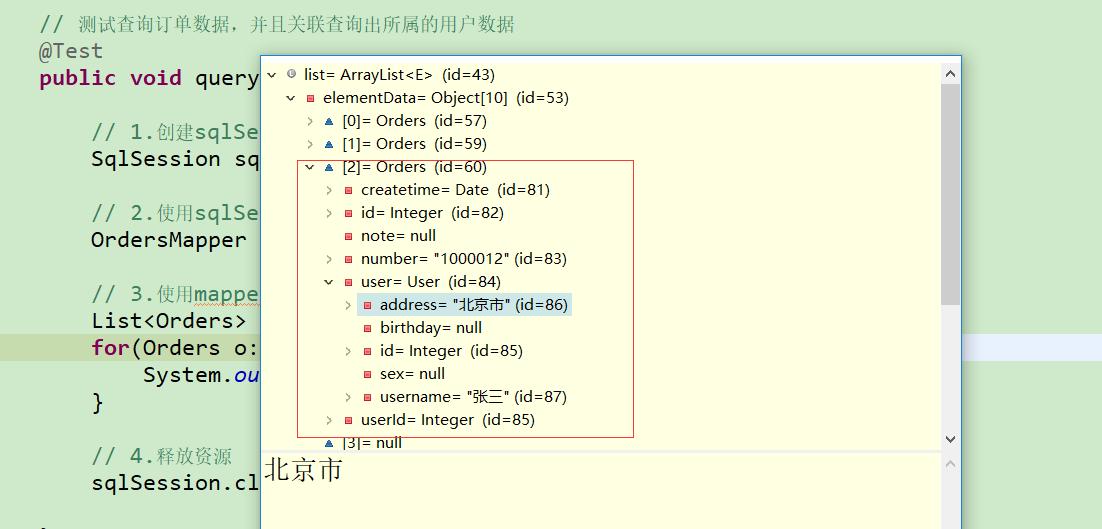
|  |
| --- |
| <!--resultMap:配置订单到用户的一对一关联关系，说明：  type属性：映射的类型  id属性：唯一标识，通过id引用该resultMap -->  <resultMap type=*"orders"* id=*"ordersUsersResultMap"*>  <!-- 配置订单主键对应关系 -->  <id column=*"id"* property=*"id"*/>    <!-- 配置订单普通字段对应关系 -->  <result column=*"user\_id"* property=*"userId"*/>  <result column=*"number"* property=*"number"*/>  <result column=*"createtime"* property=*"createtime"*/>  <result column=*"note"* property=*"note"*/>    <!--association标签：用于配置一对一关联关系，说明：  property属性：映射的属性名称  javaType属性：映射的属性类型（必须要指定）-->  <association property=*"user"* javaType=*"User"*>  <!-- 配置用户的主键对应关系 -->  <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>    <!-- 查询订单数据，并且关联查询出所属的用户数据 -->  <select id=*"queryOrdersAndUsers"* resultMap=*"ordersUsersResultMap"*>  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> |

#### 声明接口方法

|  |
| --- |
| **public** **interface** OrdersMapper {    // 1.查询全部订单数据  List<Orders> queryAllOrders();    // 2.查询全部订单数据,使用resultMap实现  List<Orders> queryAllOrdersByResultMap();    // 3.查询订单数据，并且关联查询出所属的用户数据  List<Orders> queryOrdersAndUsers();  } |

#### 编写测试代码

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



## 一对多关联查询

### 需求

查询用户数据，并且关联查询出所有的订单数据。

### 需求实现

#### 准备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pojo，增加订单集合list属性

|  |
| --- |
| **public** **class** User{    **private** Integer id; // int(11) NOT NULL AUTO\_INCREMENT,  **private** String username; // varchar(32) NOT NULL COMMENT '用户名称',  **private** Date birthday; // date DEFAULT NULL COMMENT '生日',  **private** String sex; // char(1) DEFAULT NULL COMMENT '性别',  **private** String address; // varchar(256) DEFAULT NULL COMMENT '地址',    // 订单集合属性list  **private** List<Orders> ordersList; |

#### 配置映射文件

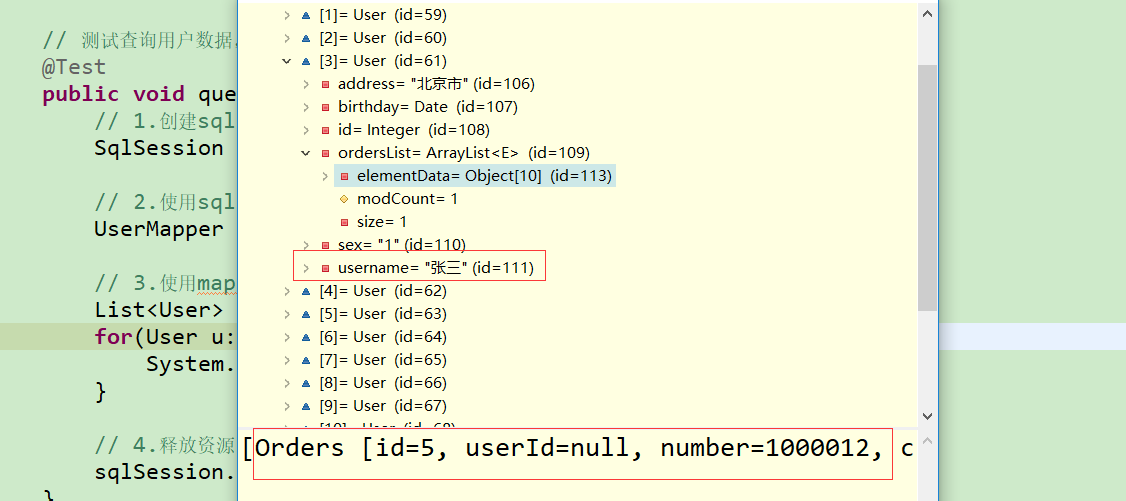
|  |
| --- |
| <!--resultMap:配置用户到订单的一对多关联关系，说明：  type属性：映射的属性类型  id属性：唯一标识，通过id引用该resultMap-->  <resultMap type=*"user"* id=*"usersOrdersResultMap"*>  <!-- 配置用户的主键对应关系 -->  <id column=*"id"* property=*"id"*/>    <!-- 配置用户普通字段对应关系 -->  <result column=*"username"* property=*"username"*/>  <result column=*"birthday"* property=*"birthday"*/>  <result column=*"sex"* property=*"sex"*/>  <result column=*"address"* property=*"address"*/>    <!--collection标签：用于配置一对多关联关系，说明：  property属性：映射的属性名称  javaType属性：映射的属性类型（可以指定，可以不指定，建议项目开放中指定）  ofType属性：list集合中存放的类型（必须要指定）-->  <collection property=*"ordersList"* javaType=*"list"* ofType=*"orders"*>  <!--配置订单主键对应关系 -->  <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>    <!-- 查询用户数据，并且关联查询出所有的订单数据 -->  <select id=*"queryUsersAndOrders"* resultMap=*"usersOrdersResultMap"*>  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> |

#### 声明接口方法

|  |
| --- |
| **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** dynamicUserById(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.查询用户数据，并且关联查询出所有的订单数据  List<User> queryUsersAndOrders();  } |

#### 编写测试代码

|  |
| --- |
| // 测试查询用户数据，并且关联查询出所有的订单数据  @Test  **public** **void** queryUsersAndOrdersTest(){  // 1.创建sqlSession对象  SqlSession sqlSession = **this**.sqlSessionFactory.openSession();    // 2.使用sqlSession对象，获取mapper代理对象  UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);    // 3.使用mapper对象，调用方法执行  List<User> list = mapper.queryUsersAndOrders();  **for**(User u:list){  System.*out*.println(u);  }    // 4.释放资源  sqlSession.close();  } |

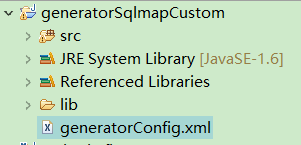


# mybatis逆向工程

## 逆向工程是什么

根据数据库表反向生成java代码，映射文件。

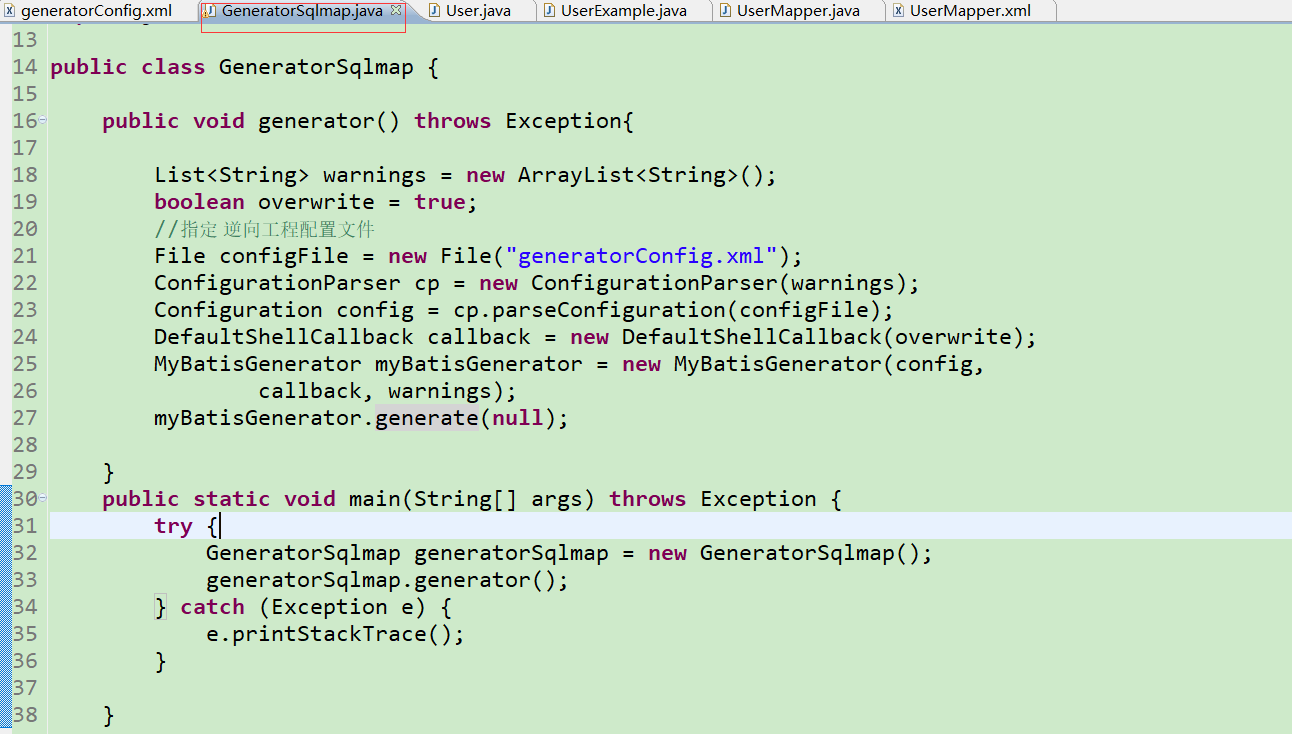
## 导入逆向工程

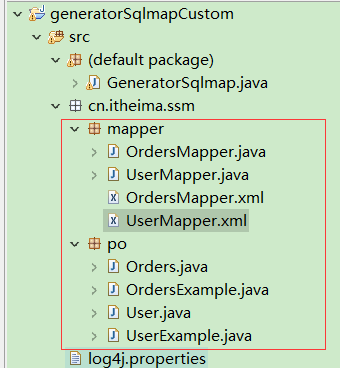


## 配置逆向工程

|  |
| --- |
| <?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"*>  <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/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> |

## 运行逆向工程





说明：

1. 逆向工程生成的是mapper代理开发方法
2. 逆向工程生成的都是单表操作
3. 使用逆向工程的好处，提升开发效率
4. 逆向工程生成的文件，直接使用，企业项目中不推荐大家修改
5. 执行逆向工程，需要将上一次执行的文件删除，再重新生成

# mybatis+spring整合

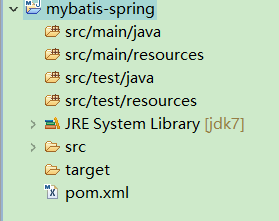
## 整合思路

思路：把mybatis框架对象，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. 在原始的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

### 创建工程



### 配置pom.xml文件，加入依赖

mybatis框架包

spring框架包

mybatis-spring整合包

|  |
| --- |
| <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-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

配置mapper映射文件加载

|  |
| --- |
| <?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

配置数据源对象

配置mybatis框架对象

|  |
| --- |
| <?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"*>  <!--设置数据源对象 -->  <property name=*"dataSource"* ref=*"dataSource"*></property>    <!-- 加载mybatis框架核心配置文件 -->  <property name=*"configLocation"* value=*"classpath:mybatis/sqlMapConfig.xml"*></property>  </bean>  </beans> |

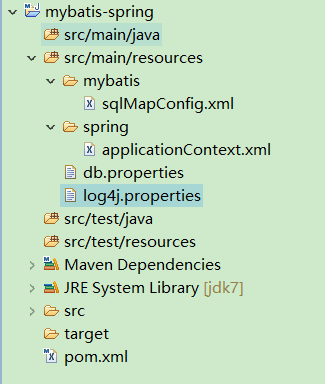
#### db.properties

|  |
| --- |
| db.driverClassName=com.mysql.jdbc.Driver  db.url=jdbc:mysql://127.0.0.1:3306/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

### 需求

1. 根据用户Id查询用户
2. 新增用户

### 需求实现

#### 准备用户pojo

|  |
| --- |
| **package** cn.itheima.ssm.po;  **import** java.util.Date;  **public** **class** User {  **private** Integer id;  **private** String username;  **private** Date birthday;  **private** String sex;  **private** String address;  **public** Integer getId() {  **return** id;  }  **public** **void** setId(Integer id) {  **this**.id = id;  }  **public** String getUsername() {  **return** username;  }  **public** **void** setUsername(String username) {  **this**.username = username == **null** ? **null** : username.trim();  }  **public** Date getBirthday() {  **return** birthday;  }  **public** **void** setBirthday(Date birthday) {  **this**.birthday = birthday;  }  **public** String getSex() {  **return** sex;  }  **public** **void** setSex(String sex) {  **this**.sex = sex == **null** ? **null** : sex.trim();  }  **public** String getAddress() {  **return** address;  }  **public** **void** setAddress(String address) {  **this**.address = address == **null** ? **null** : address.trim();  }  /\* (non-Javadoc)  \* @see java.lang.Object#toString()  \*/  @Override  **public** String toString() {  **return** "User [id=" + id + ", username=" + username + ", birthday="  + birthday + ", sex=" + sex + ", address=" + address + "]";  }    } |

#### 准备User.xml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属性：名称空间，相当于java中的package，用于放置sql语句冲突 -->  <mapper namespace=*"test"*>  <!--说明：mybatis框架针对每一种sql语句：新增/修改/删除/查询，提供了对应的标签：  insert/update/delete/select来放置 -->  <!-- 根据用户Id查询用户，说明：  select标签：用于放置查询sql语句  id属性：唯一标识一条sql语句  parameterType属性：输入参数类型  resultType属性：返回值类型  #{id}：占位符，相当于jdbc中的问号：?-->  <select id=*"queryUserById"* parameterType=*"int"* resultType=*"user"* >  select \* from `user` where id=#{id}  </select>  <!-- 新增用户，说明  insert标签：用于放置新增sql语句  #{id}：占位符，当参数传递的是pojo的时候，换括号中的内容是pojo的属性-->  <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> |

#### 准备用户dao接口

|  |
| --- |
| **public** **interface** UserDao {    // 1.根据用户id查询用户  User queryUserById(Integer id);    // 2.新增用户  **void** insertUser(User user);  } |

#### 实现用户dao接口

说明：整合包提供了SqlSessionDaoSupport类，dao实现类需要继承SqlSessionDaoSupport，获取sqlSession对象。

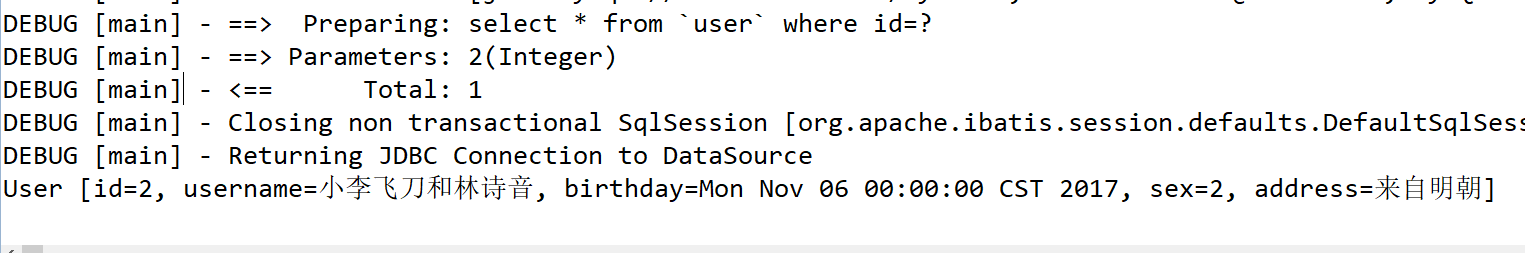
|  |
| --- |
| **public** **class** UserDaoImpl **extends** SqlSessionDaoSupport **implements** UserDao {  /\* (non-Javadoc)  \* @see cn.itheima.ssm.dao.UserDao#queryUserById(java.lang.Integer)  \*/  **public** User queryUserById(Integer id) {  // **TODO** Auto-generated method stub  // 1.获取sqlSession对象  SqlSession sqlSession = **this**.getSqlSession();    // 2.使用sqlSession对象，调用方法执行  Object user = sqlSession.selectOne("test.queryUserById", id);    // 说明：  // 1.与spring整合以后，sqlSession对象交给spring管理，不需要再手动释放（不能）    **return** (User) user;  }  /\* (non-Javadoc)  \* @see cn.itheima.ssm.dao.UserDao#insertUser(cn.itheima.ssm.po.User)  \*/  **public** **void** insertUser(User user) {  // **TODO** Auto-generated method stub  // 1.获取sqlSession对象  SqlSession sqlSession = **this**.getSqlSession();    // 2.使用sqlSession对象，调用方法执行  sqlSession.insert("test.insertUser", user);    // 说明：  // 1.与spring整合以后，sqlSession对象交给spring管理，不需要再手动释放（不能）  // 2.与spring整合以后，sqlSession对象交给spring管理，不需要再手动提交事务（不能）  // 如果配置了spring的事务，使用的spring事务，如果没有配置，默认使用jdbc事务  }  } |

#### 在applicationContext.xml中，配置用户dao实现类对象

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

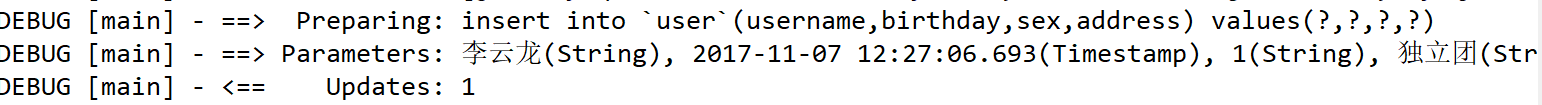
#### 编写测试代码

|  |
| --- |
| **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  ClassPathXmlApplicationContext context =  **new** ClassPathXmlApplicationContext("classpath:spring/applicationContext.xml");    // 2.使用上下文对象context，获取用户dao对象  UserDao userDao = (UserDao) context.getBean("userDao");    // 3.使用userDao对象，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  ClassPathXmlApplicationContext context =  **new** ClassPathXmlApplicationContext("classpath:spring/applicationContext.xml");    // 2.使用上下文对象context，获取用户dao对象  UserDao userDao = (UserDao) context.getBean("userDao");    // 3.使用userDao对象，调用方法执行  // 创建用户对象  User user = **new** User();  user.setUsername("李云龙");  user.setBirthday(**new** Date());  user.setSex("1");  user.setAddress("独立团");    userDao.insertUser(user);      } |



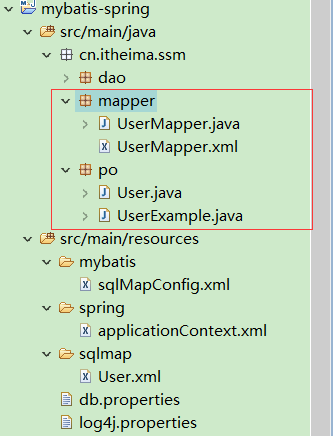
## mapper代理开发方法

### 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pojo，mapper接口和映射文件



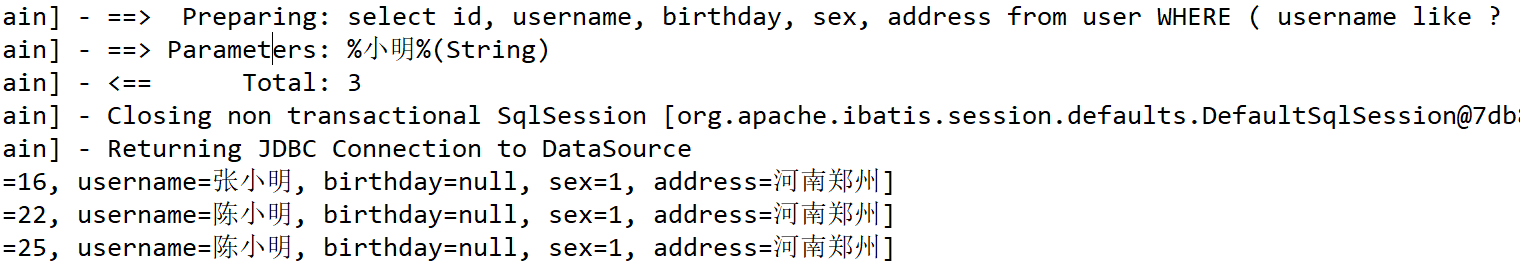
#### 准备测试

说明：整合包提供了MapperFactryBean类，配置mapper代理对象。

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

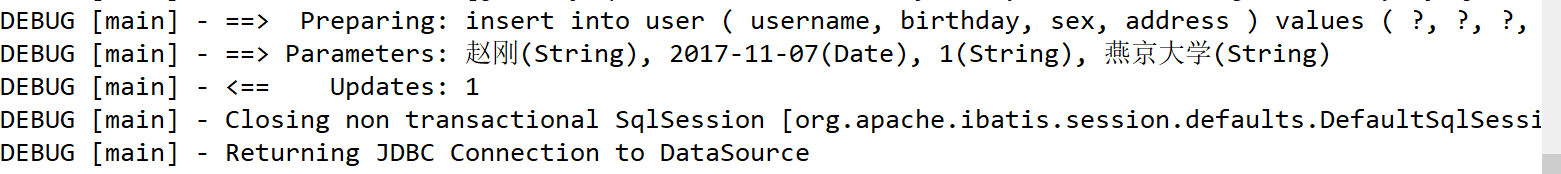
#### 编写测试代码

|  |
| --- |
| **public** **class** UserMapperTest {    // 测试根据用户名称模糊查询用户  @Test  **public** **void** queryUserByNameTest(){    // 1.加载spring配置文件  ClassPathXmlApplicationContext context =  **new** ClassPathXmlApplicationContext("classpath:spring/applicationContext.xml");    // 2.使用context对象，获取用户mapper对象  UserMapper userMapper = (UserMapper) context.getBean("userMapper");    // 3.使用userMapper调用方法执行  // 3.1.创建辅助类对象  UserExample example = **new** UserExample();    // 3.2.使用example对象，创建Criteria对象  Criteria cri = example.createCriteria();    // 3.3.使用Criteria对象，通过andXXX方法，设置where条件  cri.andUsernameLike("%小明%");  List<User> list = userMapper.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配置文件  ClassPathXmlApplicationContext context =  **new** ClassPathXmlApplicationContext("classpath:spring/applicationContext.xml");    // 2.使用context对象，获取用户mapper对象  UserMapper userMapper = (UserMapper) context.getBean("userMapper");    // 3.使用userMapper调用方法执行  // 创建用户对象  User user = **new** User();  user.setUsername("赵刚");  user.setBirthday(**new** Date());  user.setSex("1");  user.setAddress("燕京大学");    userMapper.insertSelective(user);    } |



#### mapper代理对象配置方式二（掌握）

|  |
| --- |
| <!-- 配置mapper代理对象方式二(mapper扫描器) -->  <bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>  <!-- 设置要扫描的包,说明：  1.如果有多个包，以半角逗号进行分割  2.企业项目中，推荐使用包扫描方式-->  <property name=*"basePackage"* value=*"cn.itheima.ssm.mapper"*/>  </bean> |