# 课程计划

1. mybatis的介绍
2. mybatis的框架原理（重点）
3. 入门程序

订单商品案例（用户表）

1. Mybatis开发dao的方式（重点）
   1. 原始dao开发方式（开发dao接口和dao实现类，由ibatis遗留下来的风格）
   2. Mapper代理的开发方式（推荐，开发mapper接口（相当于dao接口））
2. 全局配置文件
3. 映射文件（重点）
   1. 输入映射
   2. 输出映射
   3. 动态sql

# 分析jdbc的问题

## 原生态的jdbc代码

package com.zhiyou100;  
import java.sql.\*;  
*/\*\*  
 \* Created by lijike 2016/12/1.  
 \*/*public class JdbcDemo {  
public static void main(String[] args) throws Exception {  
 Connection connection = null;  
PreparedStatement preparedStatement = null;  
ResultSet resultSet = null;  
 try {  
//1、加载数据库驱动  
Class.*forName*("com.mysql.jdbc.Driver");  
//2、通过驱动管理类获取数据库链接  
connection = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8", "root", "root");  
//3、定义sql语句 ?表示占位符  
String sql = "select \* from tb\_user where username = ?";  
//4、获取预处理statement  
preparedStatement = connection.prepareStatement(sql);  
//5、设置参数，第一个参数为sql语句中参数的序号（从1开始），第二个参数为设置的参数值  
preparedStatement.setString(1, "王五");  
//6、向数据库发出sql执行查询，查询出结果集  
resultSet = preparedStatement.executeQuery();  
//7、遍历查询结果集  
while (resultSet.next()) {  
 System.*out*.println(resultSet.getString("id") + "" + resultSet.getString("username"));  
}  
 } catch (Exception e) {  
 e.printStackTrace();  
} finally {  
//8、释放资源  
if (resultSet != null) {  
try {  
resultSet.close();  
} catch (SQLException e) {  
 e.printStackTrace();  
}  
 }  
if (preparedStatement != null) {  
try {  
 preparedStatement.close();  
} catch (SQLException e) {  
 e.printStackTrace();  
}  
 }  
if (connection != null) {  
try {  
 connection.close();  
} catch (SQLException e) {  
 e.printStackTrace();  
}  
 }  
 }  
 }  
}

## 问题总结

1. 在创建连接时，存在硬编码

配置文件（全局配置文件）

1. 在执行statement时存在硬编码

配置文件（映射文件）

1. 频繁的开启和关闭数据库连接，会造成数据库性能下降。

数据库连接池（全局配置文件）

1. 从resultSet中遍历结果集数据时，存在硬编码，将获取表的字段进行硬编码，不利于系统维护。

将查询的结果集自动映射成java对象

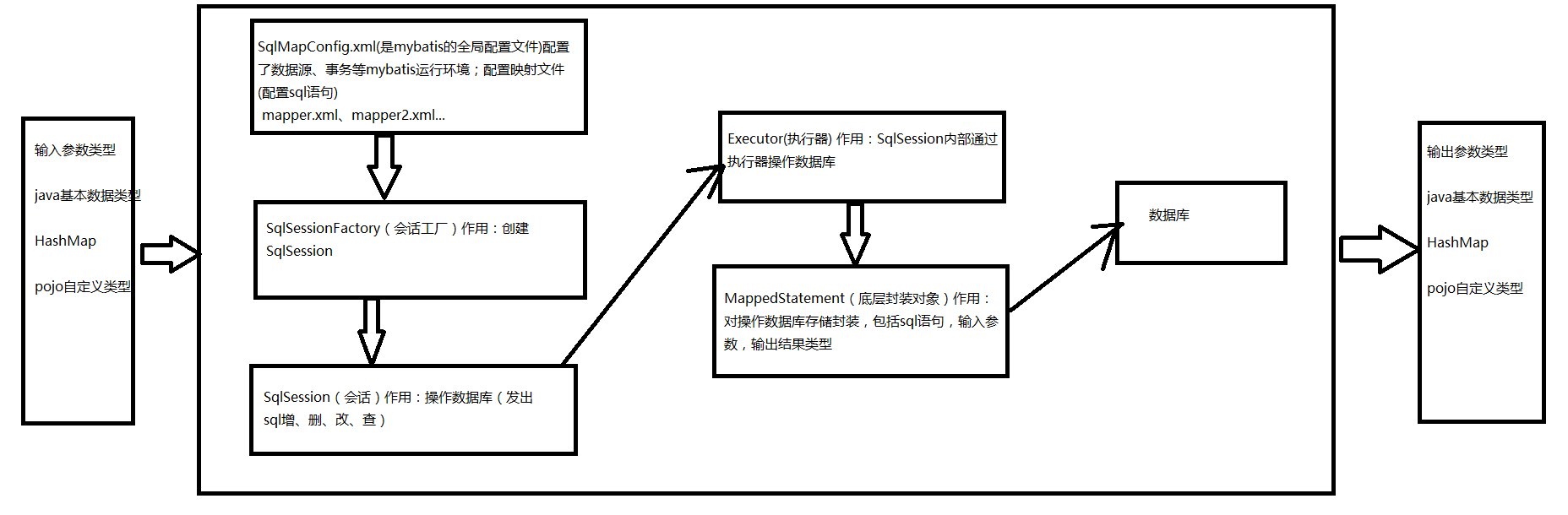
# mybatis的介绍

mybatis就是一个封装来jdbc的持久层框架，它和hibernate都属于ORM框架，但是具体的说，hibernate是一个完全的orm框架，而mybatis是一个不完全的orm框架。

Mybatis让程序员**只关注sql本身**，而不需要去关注如连接的创建、statement的创建等操作，通过mybatis提供的映射方式，自由灵活生成满足需求的sql语句，注意是半自动化生成，大部分需要程序员自己编写。

Mybatis会将输入参数、输出结果进行映射。

# Mybatis的框架原理



# 入门程序

## 需求

对用户表进行增删改查操作

1. 根据用户ID查询用户信息
2. 根据用户名称模糊查询用户列表
3. 添加用户
4. 删除用户（练习）
5. 修改用户（练习）

## 环境准备

* Jdk：1.7
* Ide：idea
* Mybatis：3.3.0
* 数据库：MySQL 5X

### 数据库脚本初始化

1. 创建数据库和表结构

**create database** mybatis;

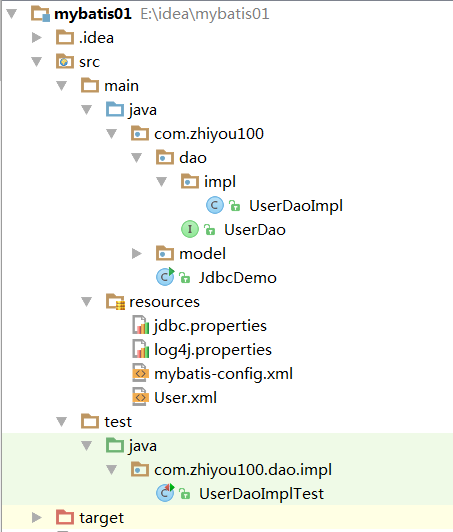
1. 创建表初始化数据

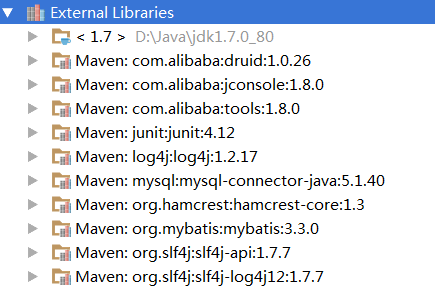
-- ----------------------------  
-- Table structure for tb\_goods  
-- ----------------------------  
**DROP TABLE IF EXISTS** `tb\_goods`;  
**CREATE TABLE** `tb\_goods` (  
`id` **int**(11) **NOT NULL AUTO\_INCREMENT**,  
`name` **varchar**(255) **DEFAULT NULL**,  
`price` **float**(10,2) **DEFAULT NULL**,  
`create\_time` **datetime DEFAULT NULL**,  
`detail` **varchar**(255) **DEFAULT NULL**,  
**PRIMARY KEY** (`id`)  
) **ENGINE**=InnoDB **AUTO\_INCREMENT**=5 **DEFAULT CHARSET**=utf8;  
  
-- ----------------------------  
-- Records of tb\_goods  
-- ----------------------------  
**INSERT INTO** `tb\_goods` **VALUES** ('1', '荣耀V8', '2799.00', '2016-11-22 14:25:11', '荣耀V8手机');  
**INSERT INTO** `tb\_goods` **VALUES** ('2', 'IPHONE9', '3999.00', '2016-11-22 14:25:47', '苹果手机');  
**INSERT INTO** `tb\_goods` **VALUES** ('3', 'JAVA编程思想', '80.90', '2016-11-22 14:26:17', 'Java技术书籍');  
**INSERT INTO** `tb\_goods` **VALUES** ('4', '明朝那些事', '30.00', '2016-11-22 14:26:48', '明朝那些事');  
  
-- ----------------------------  
-- Table structure for tb\_order  
-- ----------------------------  
**DROP TABLE IF EXISTS** `tb\_order`;  
**CREATE TABLE** `tb\_order` (  
`id` **int**(11) **NOT NULL AUTO\_INCREMENT**,  
`user\_id` **int**(11) **DEFAULT NULL**,  
`order\_number` **varchar**(255) **DEFAULT NULL**,  
`create\_time` **datetime DEFAULT NULL**,  
`detail` **varchar**(255) **DEFAULT NULL**,  
**PRIMARY KEY** (`id`),  
**KEY** `user\_id` (`user\_id`),  
**CONSTRAINT** `tb\_order\_ibfk\_1` **FOREIGN KEY** (`user\_id`) **REFERENCES** `tb\_user` (`id`)  
) **ENGINE**=InnoDB **AUTO\_INCREMENT**=3 **DEFAULT CHARSET**=utf8;  
  
-- ----------------------------  
-- Records of tb\_order  
-- ----------------------------  
**INSERT INTO** `tb\_order` **VALUES** ('1', '1', '2016112201', '2016-11-22 14:20:24', '1订单');  
**INSERT INTO** `tb\_order` **VALUES** ('2', '2', '2016112202', '2016-11-22 14:37:15', '2订单');  
  
-- ----------------------------  
-- Table structure for tb\_order\_detail  
-- ----------------------------  
**DROP TABLE IF EXISTS** `tb\_order\_detail`;  
**CREATE TABLE** `tb\_order\_detail` (  
`id` **int**(11) **NOT NULL AUTO\_INCREMENT**,  
`order\_id` **int**(11) **DEFAULT NULL**,  
`goods\_id` **int**(11) **DEFAULT NULL**,  
`goods\_num` **int**(11) **DEFAULT NULL**,  
**PRIMARY KEY** (`id`),  
**KEY** `order\_id` (`order\_id`),  
**KEY** `goods\_id` (`goods\_id`),  
**CONSTRAINT** `tb\_order\_detail\_ibfk\_1` **FOREIGN KEY** (`order\_id`) **REFERENCES** `tb\_order` (`id`),  
**CONSTRAINT** `tb\_order\_detail\_ibfk\_2` **FOREIGN KEY** (`goods\_id`) **REFERENCES** `tb\_goods` (`id`)  
) **ENGINE**=InnoDB **AUTO\_INCREMENT**=5 **DEFAULT CHARSET**=utf8;  
  
-- ----------------------------  
-- Records of tb\_order\_detail  
-- ----------------------------  
**INSERT INTO** `tb\_order\_detail` **VALUES** ('1', '1', '1', '3');  
**INSERT INTO** `tb\_order\_detail` **VALUES** ('2', '1', '2', '2');  
**INSERT INTO** `tb\_order\_detail` **VALUES** ('3', '1', '3', '1');  
**INSERT INTO** `tb\_order\_detail` **VALUES** ('4', '2', '1', '1');  
  
-- ----------------------------  
-- Table structure for tb\_user  
-- ----------------------------  
**DROP TABLE IF EXISTS** `tb\_user`;  
**CREATE TABLE** `tb\_user` (  
`id` **int**(11) **NOT NULL AUTO\_INCREMENT**,  
`username` **varchar**(255) **DEFAULT NULL**,  
`sex` **varchar**(255) **DEFAULT NULL**,  
`birthday` **datetime DEFAULT NULL**,  
`address` **varchar**(255) **DEFAULT NULL**,  
**PRIMARY KEY** (`id`)  
) **ENGINE**=InnoDB **AUTO\_INCREMENT**=25 **DEFAULT CHARSET**=utf8;  
  
-- ----------------------------  
-- Records of tb\_user  
-- ----------------------------  
**INSERT INTO** `tb\_user` **VALUES** ('1', '张三', '男', '2016-11-21 00:10:36', '河南安阳');  
**INSERT INTO** `tb\_user` **VALUES** ('2', '李四', '女', '2016-11-22 14:22:11', '河南安阳');

## 工程搭建

* Mybatis的核心包和依赖包
* MySQL的驱动包
* Junit（非必须）

<?xml version="1.0" encoding="UTF-8"?>  
<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>com.zhiyou100</groupId>  
<artifactId>mybatis01</artifactId>  
<version>1.0-SNAPSHOT</version>  
<dependencies>  
<!-- junit start-->  
<dependency>  
<groupId>junit</groupId>  
<artifactId>junit</artifactId>  
<version>4.12</version>  
<scope>test</scope>  
</dependency>  
<!-- junit end-->  
<!--database start-->  
<dependency>  
<groupId>mysql</groupId>  
<artifactId>mysql-connector-java</artifactId>  
<version>5.1.40</version>  
<scope>runtime</scope>  
</dependency>  
  
<dependency>  
<groupId>org.mybatis</groupId>  
<artifactId>mybatis</artifactId>  
<version>3.3.0</version>  
</dependency>  
  
<dependency>  
<groupId>com.alibaba</groupId>  
<artifactId>druid</artifactId>  
<version>1.0.26</version>  
</dependency>  
<!--database end -->  
  
<!-- loging begin-->  
<dependency>  
<groupId>org.slf4j</groupId>  
<artifactId>slf4j-log4j12</artifactId>  
<version>1.7.7</version>  
</dependency>  
<!-- loging end -->  
</dependencies>  
</project>





## 代码实现

### 创建User类

public class User {  
private Long id;  
 private String username;  
 private String sex;  
 private Date birthday;  
 private String address;

### 创建UserDao接口

public interface UserDao {  
*/\*\*  
 \* 根据用户id查询用户  
\** ***@param*** *id 用户id  
 \** ***@return*** *实体对象  
\*/*User getUserById(Long id);  
  
*/\*\*  
 \* 根据用户名查询用户  
\** ***@param*** *username 用户名  
\** ***@return*** *实体集合  
\*/*List<User>findUserByUsername(String username);  
  
*/\*\*  
 \* 新增用户  
\** ***@param*** *user 用户  
\** ***@return*** *结果  
\*/*int insertUser(User user);  
  
*/\*\*  
 \* 更新用户  
\** ***@param*** *user 用户  
\** ***@return*** *结果  
\*/*int updateUser(User user);  
  
*/\*\*  
 \* 删除用户  
\** ***@param*** *user 用户  
\** ***@return*** *结果  
\*/*int deleteUserById(Long id);  
}

### 创建UserDao接口实现类

public class UserDaoImpl implements UserDao {  
  
private SqlSessionFactory sqlSessionFactory;  
  
 public UserDaoImpl(SqlSessionFactory sqlSessionFactory){  
this.sqlSessionFactory = sqlSessionFactory;  
}  
  
@Override  
public User getUserById(Long id) {  
 return null;  
}  
  
@Override  
public List<User>findUserByUsername(String username) {  
return null;  
}  
  
@Override  
public int insertUser(User user) {  
return 0;  
}  
  
@Override  
public int updateUser(User user) {  
return 0;  
}  
  
@Override  
public int deleteUserById(Long id) {  
return 0;  
}  
}

### 创建log4.properties配置文件

在src/main/resources目录下，创建log4.properties文件

log4j.rootLogger=DEBUG, Console  
#Console  
log4j.appender.Console=org.apache.log4j.ConsoleAppender  
log4j.appender.Console.layout=org.apache.log4j.PatternLayout  
log4j.appender.Console.layout.ConversionPattern=%d [%t] %-5p [%c] - %m%n  
log4j.logger.org.apache.ibatis=DEBUG

### 创建数据库连接配置文件

在src/main/resources目录下，创建jdbc.properties文件

#数据库连接信息  
jdbc.driverClassName=com.mysql.jdbc.Driver  
jdbc.url=jdbc:mysql://localhost:3306/mybatis  
jdbc.username=root  
jdbc.password=123456

### 创建全局配置文件

在src/main/resources目录下，创建mybatis-config.xml文件

<?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>  
<!-- 加载属性文件 -->  
<properties resource="jdbc.properties"/>  
<environments default="development">  
<environment id="development">  
<!-- 使用jdbc管理事务mybatis -->  
<transactionManager type="JDBC"/>  
<dataSource type="POOLED">  
<property name="driver" value="${jdbc.driverClassName}"/>  
<property name="url" value="${jdbc.url}"/>  
<property name="username" value="${jdbc.username}"/>  
<property name="password" value="${jdbc.password}"/>  
</dataSource>  
</environment>  
</environments>  
<!-- 加载映射文件 -->  
<mappers>

</mappers>  
</configuration>

### 需求开发

#### 根据用户ID查询用户信息

##### 映射文件

在src/main/resource目录下，创建User.xml（这种命名规范是由ibatis遗留下来）

<?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">  
<!--  
1.User.xml(原始ibatis命名)，mapper代理开发的映射文件命名XxxMapper.xml  
2.namespace命名空间，作用就是对sql进行分类化管理，理解sql隔离  
3.注意:使用mapper代理开发，namespace有特殊的作用  
-->  
<mapper namespace="user">  
<!--在映射文件中配置很多sql语句-->  
<!--需求：通过id查询用户表的记录-->  
<!--  
 1.标识映射文件中的sql，将sql语句封装到mapped statement对象中，所以称为statement的id  
 2.parameterType：指定输入参数类型  
 3.#{}：表示一个占位符  
 4.#{id}：其中的id表示接收的输入参数，参数名称就是id，如果输入参数是简单类型，#{}中的参数名可以任意，可以为value或者其他  
 5.resultType：指定sql输出结果的所映射的java对象类型，select指定resultType表示将单条记录映射成的java对象  
-->  
<select id="findUserById" parameterType="java.lang.Long" resultType="com.zhiyou100.model.User">  
SELECT \* FROM tb\_user WHERE id = #{id}  
</select>  
</mapper>

##### 在全局配置文件中加载映射文件

<!-- 加载映射文件 -->  
<mappers>  
<mapper resource="User.xml"/>  
</mappers>

##### 接口实现

@Override  
public User getUserById(Long id) {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
User user = sqlSession.selectOne("user.getUserById",id);  
sqlSession.close();  
 return user;  
}

##### 测试代码

public class UserDaoImplTest {  
private SqlSessionFactory sqlSessionFactory;  
@Before  
public void setUp(){  
try {  
 InputStream inputStream = Resources.*getResourceAsStream*("mybatis-config.xml");  
sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
} catch (IOException e) {  
 e.printStackTrace();  
}  
 }  
@Test  
public void getUserById() throws Exception {  
 UserDao userDao = new UserDaoImpl(sqlSessionFactory);  
User user = userDao.getUserById(1L);  
System.*out*.println(user);  
}

}

#### 根据用户名称模糊查询用户列表

##### 映射文件

<!--  
 1.需求根据用户名称模糊查询  
 2.思路：模糊查询返回结果可能是多条纪录  
 3.resultType:指定就是单条记录所映射成java对象类型  
 4.${}:表示拼接sql串，将接收到参数的内容不加任何修饰拼接在sql中  
 5.使用${}容易引起sql注入  
 6.${value}:接受输入参数的内容，如果传入类型是简单类型，${}中只能用value  
-->  
<select id="findUserByUsername" parameterType="java.lang.String" resultType="com.zhiyou100.model.User">  
SELECT \* FROM tb\_user WHERE username LIKE '%${value}%'  
</select>

##### 接口实现

@Override  
public List<User>findUserByUsername(String username) {  
// 创建SqlSession  
SqlSession sqlSession = sqlSessionFactory.openSession();  
List<User> list = sqlSession.selectList("user.findUserByUsername",username);  
// 关闭资源  
sqlSession.close();  
 return list;  
}

##### 测试代码

@Test  
public void findUserByUsername() throws Exception {  
 UserDao userDao = new UserDaoImpl(sqlSessionFactory);  
List<User> list = userDao.findUserByUsername("张三");  
System.*out*.println(list);  
}

#### 添加用户

##### 映射文件

<!--  
 1.需求：添加用户  
 2.思路：创建对象，输入参数是java对象类型User  
 3.#{}：指定pojo(model)的属性名，接收到pojo对象的属性值，mybatis通过OGNL获取对象的属性值  
 4.parameterType：指定输入参数类型是pojo（包括用户信息）  
-->  
<insert id="insertUser" parameterType="com.zhiyou100.model.User">  
<!--  
将插入数据的主键返回，返回到user对象中  
 SELECT LAST\_INSERT\_ID()：得到刚insert进去记录的主键值，只适用于自增主键  
 keyProperty：将查询到主键值设置到parameterType指定的对象的哪个属性  
 order：SELECT LAST\_INSERT\_ID()执行顺序，相对于insert语句来说它的执行顺序  
 resultType：指定SELECT LAST\_INSERT\_ID()的结果类型  
-->  
<selectKey keyProperty="id" order="AFTER" resultType="java.lang.Long">  
SELECT LAST\_INSERT\_ID()  
</selectKey>  
INSERT INTO tb\_user (USERNAME,SEX,BIRTHDAY,ADDRESS) VALUE  
 (  
 #{username},  
 #{sex},  
 #{birthday},  
 #{address}  
 )  
</insert>

##### 接口实现

@Override  
public int insertUser(User user) {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
 int result = sqlSession.insert("user.insertUser",user);

sqlSession.commit();  
sqlSession.close();  
 return result;  
}

##### 测试代码

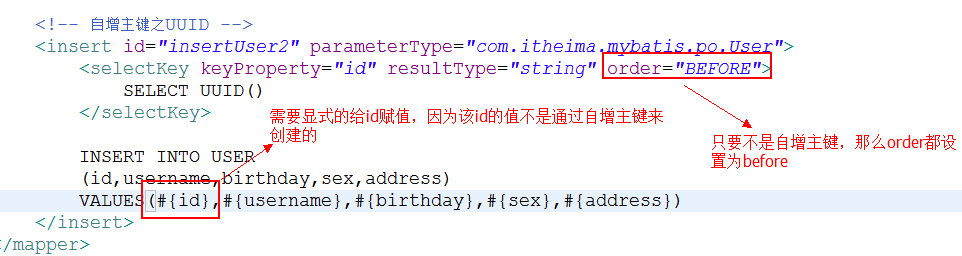
@Test  
public void insertUser() throws Exception {  
 UserDao userDao = new UserDaoImpl(sqlSessionFactory);  
User user = new User();  
user.setUsername("王五");  
user.setSex("男");  
user.setBirthday(new Date());  
user.setAddress("郑州市");  
userDao.insertUser(user);  
}

##### 主键返回之自增主键

<!--  
将插入数据的主键返回，返回到user对象中  
 SELECT LAST\_INSERT\_ID()：得到刚insert进去记录的主键值，只适用于自增主键  
 keyProperty：将查询到主键值设置到parameterType指定的对象的哪个属性  
 order：SELECT LAST\_INSERT\_ID()执行顺序，相对于insert语句来说它的执行顺序  
 resultType：指定SELECT LAST\_INSERT\_ID()的结果类型  
-->  
<selectKey keyProperty="id" order="AFTER" resultType="java.lang.Long">  
SELECT LAST\_INSERT\_ID()  
</selectKey>

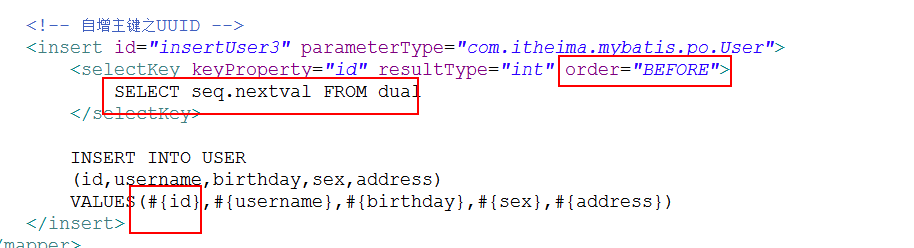
##### 主键返回值UUID

UUID函数是mysql的函数



##### 主键返回值序列

序列也就是sequence，它是Oracle的主键生成策略



#### 更新用户（练习）

#### 删除用户（练习）

### 小结

* #{}和${}

#{}表示占位符?，#{}接收简单类型的参数时，里面的名称可以任意

${}表示拼接符，${}接收简单类型的参数时，里面的名称必须是value

${}里面的值会原样输出，不加解析（如果该参数值是字符串，有不会添加引号）

${}存在sql注入的风险，但是有些场景下必须使用，比如排序后面会动态传入排序的列名

* parameterType和resultType

parameterType指定输入参数的java类型，parameterType只有一个，也就是说入参只有一个。

resultType指定输出结果的java类型（是单条记录的java类型）

* selectOne和selectList

selectOne查询单个对象

selectList查询集合对象

# mybatis开发dao的方式

## 需求

1. 根据用户ID查询用户信息
2. 根据用户名称模糊查询用户列表
3. 添加用户

## 原始dao的开发方式

即开发dao接口和dao实现类

### Dao接口

public interface UserDao {User getUserById(Long id);List<User>findUserByUsername(String username);List<User>findUser(User user);int insertUser(User user);int updateUser(User user);int deleteUserById(Long id);  
}

### Dao实现类

SqlSessionFactory，它的生命周期，应该是应用范围，全局范围只有一个工厂，使用单例模式来实现这个功能。

SqlSession，它内部含有一块数据区域，存在线程不安全的问题，所以应该将sqlsession声明到方法内部

public class UserDaoImpl implements UserDao {  
  
private SqlSessionFactory sqlSessionFactory;  
  
 public UserDaoImpl(SqlSessionFactory sqlSessionFactory){  
this.sqlSessionFactory = sqlSessionFactory;  
}  
@Override  
public User getUserById(Long id) {  
// 创建SqlSession  
SqlSession sqlSession = sqlSessionFactory.openSession();  
User user = sqlSession.selectOne("user.getUserById",id);  
// 关闭资源  
sqlSession.close();  
 return user;  
}  
  
@Override  
public List<User>findUserByUsername(String username) {  
// 创建SqlSession  
SqlSession sqlSession = sqlSessionFactory.openSession();  
List<User> list = sqlSession.selectList("user.findUserByUsername",username);  
// 关闭资源  
sqlSession.close();  
 return list;  
}  
  
@Override  
public List<User>findUser(User user) {  
return null;  
}

@Override  
public int insertUser(User user) {  
// 创建SqlSession  
SqlSession sqlSession = sqlSessionFactory.openSession();  
 int result = sqlSession.insert("user.insertUser",user);  
// 提交事务  
sqlSession.commit();  
// 关闭资源  
sqlSession.close();  
 return result;  
}  
  
@Override  
public int updateUser(User user) {  
return 0;  
}  
  
@Override  
public int deleteUserById(Long id) {  
return 0;  
}  
}

### 问题思考

1. 有大量的重复的模板代码
2. 存在硬编码

## Mapper代理的开发方式

即开发mapper接口（相当于dao接口）

Mapper代理使用的是jdk的代理策略。

### Mapper代理的开发规范

1. mapper接口的全限定名要和mapper映射文件的namespace值一致。
2. mapper接口的方法名称要和mapper映射文件的statement的id一致。
3. mapper接口的方法参数类型要和mapper映射文件的statement的parameterType的值一致，而且它的参数是一个。
4. mapper接口的方法返回值类型要和mapper映射文件的statement的resultType的值一致。

### mapper接口

public interface UserMapper {  
*/\*\*  
 \* 根据用户id查询用户  
\** ***@param*** *id 用户id  
 \** ***@return*** *实体对象  
\*/*User getUserById(Long id);  
  
*/\*\*  
 \* 根据用户名查询用户  
\** ***@param*** *username 用户名  
\** ***@return*** *实体集合  
\*/*List<User>findUserByUsername(String username);  
  
*/\*\*  
 \* 新增用户  
\** ***@param*** *user 用户  
\** ***@return*** *结果  
\*/*int insertUser(User user);  
  
*/\*\*  
 \* 更新用户  
\** ***@param*** *user 用户  
\** ***@return*** *结果  
\*/*int updateUser(User user);  
  
*/\*\*  
 \* 删除用户  
\** ***@param*** *user 用户  
\** ***@return*** *结果  
\*/*int deleteUserById(Long id);  
}

### mapper映射文件

在src/main/resources下创建mapper目录然后创建UserMapper.xml（这是mybatis的命名规范，当然，也不是必须是这个名称）

sqlSession内部的数据区域本身就是一级缓存，是通过map来存储的。

<?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">  
<!--  
1、 mapper接口的全限定名要和mapper映射文件的namespace值一致。  
2、 mapper接口的方法名称要和mapper映射文件的statement的id一致。  
3、 mapper接口的方法参数类型要和mapper映射文件的statement的parameterType的值一致，而且它的参数是一个。  
4、 mapper接口的方法返回值类型要和mapper映射文件的statement的resultType的值一致。  
-->  
<mapper namespace="com.zhiyou100.mapper.UserMapper">  
<!--在映射文件中配置很多sql语句-->  
<!--需求：通过id查询用户表的记录-->  
<!--  
 1.标识映射文件中的sql，将sql语句封装到mapped statement对象中，所以称为statement的id  
 2.parameterType：指定输入参数类型  
 3.#{}：表示一个占位符  
 4.#{id}：其中的id表示接收的输入参数，参数名称就是id，如果输入参数是简单类型，#{}中的参数名可以任意，可以为value或者其他  
 5.resultType：指定sql输出结果的所映射的java对象类型，select指定resultType表示将单条记录映射成的java对象  
-->  
<select id="getUserById" parameterType="java.lang.Long" resultType="com.zhiyou100.model.User">  
SELECT \* FROM tb\_user WHERE id = #{id}  
</select>  
  
<!--  
 1.需求根据用户名称模糊查询  
 2.思路：模糊查询返回结果可能是多条纪录  
 3.resultType:指定就是单条记录所映射成java对象类型  
 4.${}:表示拼接sql串，将接收到参数的内容不加任何修饰拼接在sql中  
 5.使用${}容易引起sql注入  
 6.${value}:接受输入参数的内容，如果传入类型是简单类型，${}中只能用value  
 -->  
<select id="findUserByUsername" parameterType="java.lang.String" resultType="com.zhiyou100.model.User">  
SELECT \* FROM tb\_user WHERE username LIKE '%${value}%'  
</select>  
  
<!--  
 1.需求：添加用户  
 2.思路：创建对象，输入参数是java对象类型User  
 3.#{}：指定pojo(model)的属性名，接收到pojo对象的属性值，mybatis通过OGNL获取对象的属性值  
 4.parameterType：指定输入参数类型是pojo（包括用户信息）  
-->  
<insert id="insertUser" parameterType="com.zhiyou100.model.User">  
<!--  
将插入数据的主键返回，返回到user对象中  
 SELECT LAST\_INSERT\_ID()：得到刚insert进去记录的主键值，只适用于自增主键  
 keyProperty：将查询到主键值设置到parameterType指定的对象的哪个属性  
 order：SELECT LAST\_INSERT\_ID()执行顺序，相对于insert语句来说它的执行顺序  
 resultType：指定SELECT LAST\_INSERT\_ID()的结果类型  
-->  
<selectKey keyProperty="id" order="AFTER" resultType="java.lang.Long">  
SELECT LAST\_INSERT\_ID()  
</selectKey>  
INSERT INTO tb\_user (USERNAME,SEX,BIRTHDAY,ADDRESS) VALUE  
 (  
 #{username},  
 #{sex},  
 #{birthday},  
 #{address}  
 )  
</insert>  
</mapper>

### 加载映射文件

<!-- 加载映射文件 -->  
<mappers>  
<mapper resource="mapper/UserMapper.xml"/>  
</mappers>

### 测试代码

public class UserMapperImplTest {  
private SqlSessionFactory sqlSessionFactory;

@Before  
public void setUp(){  
try {  
 InputStream inputStream = Resources.*getResourceAsStream*("mybatis-config.xml");  
sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
} catch (IOException e) {  
 e.printStackTrace();  
}  
  
 }  
@Test  
public void getUserById() throws Exception {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
User user = userMapper.getUserById(1L);  
System.*out*.println(user);

sqlSession.close();  
}  
  
@Test  
public void findUserByUsername() throws Exception {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
List<User> list = userMapper.findUserByUsername("张三");  
System.*out*.println(list);

sqlSession.close();  
  
}  
  
@Test  
public void insertUser() throws Exception {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
User user = new User();  
user.setUsername("王五");  
user.setSex("男");  
user.setBirthday(new Date());  
user.setAddress("郑州市");  
userMapper.insertUser(user);

sqlSession.commit();

sqlSession.close();  
}  
  
@Test  
public void updateUser() throws Exception {  
  
 }  
  
@Test  
public void deleteUserById() throws Exception {  
  
 }  
}

# 全局配置文件

## 概览

**mybatis-config.xml的配置内容和顺序如下（顺序不能乱）：**

properties（属性）用于配置属性信息

settings（全局参数设置）用于配置MyBatis的运行时方式

typeAliases（类型别名）配置类型别名，可以在xml中用别名取代全限定名

typeHandlers（类型处理器）配置类型处理器

objectFactory（对象工厂）

plugins（插件）配置拦截器，用于拦截sql语句的执行

environments（环境信息集合）

environment（单个环境信息）配置数据源信息、连接池、事务属性等

transactionManager（事物）

dataSource（数据源）

mappers（映射器）配置SQL映射文件

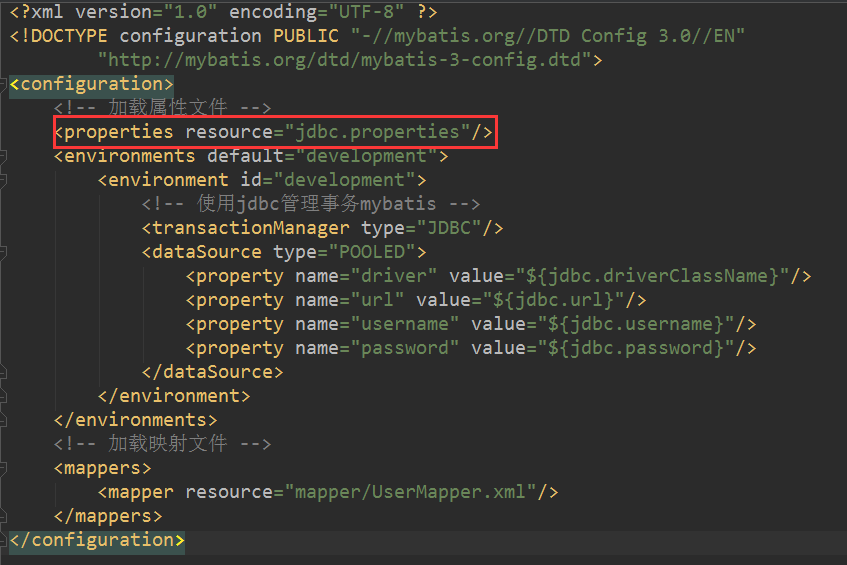
## 常用配置

### properties

jdbc.properties

#数据库连接信息  
jdbc.driverClassName=com.mysql.jdbc.Driver  
jdbc.url=jdbc:mysql://localhost:3306/mybatis  
jdbc.username=root  
jdbc.password=123456

mybatis-config.xml



加载的顺序

注意：parameterType的值会和properties的属性值发生冲突。

### settings

mybatis全局配置参数，全局参数将会影响mybatis的运行行为

配置格式：

<settings>  
<setting name="name1" value="value1" />  
<setting name="name2" value="value2" />  
<setting name="name3" value="value3" />  
......  
<setting name="nameN" value="valueN" />  
</settings>

setting可设置的项相关介绍参见下表：

|  |  |  |  |
| --- | --- | --- | --- |
| **设置参数** | **描述** | **有效值** | **默认值** |
| cacheEnabled | 这个配置使全局的映射器启用或禁用缓存。 | true | false | true |
| lazyLoadingEnabled | 全局启用或禁用延迟加载。当禁用时, 所有关联对象都会即时加载。 | true | false | false |
| aggressiveLazyLoading | 当启用时, 有延迟加载属性的对象在被调用时将会完全加载任意属性。否则,每种属性将会按需要加载。 | true | false | true |
| multipleResultSetsEnabled | 允许或不允许多种结果集从一个单独                 的语句中返回(需要适合的驱动) | true | false | true |
| useColumnLabel | 使用列标签代替列名。                不同的驱动在这                 方便表现不同。                 参考驱动文档或充分测                 试两种方法来决定所使用的驱动。 | true | false | true |
| useGeneratedKeys | 允许 JDBC 支持生成的键。                 需要适合的                 驱动。                 如果设置为 true 则这个设置强制                 生成的键被使用,                  尽管一些驱动拒绝兼                 容但仍然有效(比如 Derby) | true | false | False |
| autoMappingBehavior | 指定 MyBatis 如何自动映射列到字段/                 属性。PARTIAL 只会自动映射简单,                  没有嵌套的结果。FULL 会自动映射任                 意复杂的结果(嵌套的或其他情况)                   。 | NONE, PARTIAL, FULL | PARTIAL |
| defaultExecutorType | 配置默认的执行器。SIMPLE 执行器没                 有什么特别之处。REUSE 执行器重用                 预处理语句。BATCH 执行器重用语句                 和批量更新 | SIMPLE                REUSE                BATCH | SIMPLE |
| defaultStatementTimeout | 设置超时时间,                 它决定驱动等待一个数                据库响应的时间。 | Any positive integer | Not Set (null) |
| safeRowBoundsEnabled | Allows using RowBounds on nested                statements. | true | false | False |
| mapUnderscoreToCamelCase | Enables automatic mapping from                classic database column names                A\_COLUMN to camel case classic Java                property names aColumn. | true | false | False |
| localCacheScope | MyBatis uses local cache to prevent circular references and speed up repeated nested queries.                 By default (SESSION) all queries executed during a session are cached. If localCacheScope=STATEMENT local session will be used just for                 statement execution, no data will be shared between two different calls to the same SqlSession. | SESSION | STATEMENT | SESSION |
| jdbcTypeForNull | Specifies the JDBC type for null values when no specific JDBC type was provided for the parameter.                 Some drivers require specifying the column JDBC type but others work with generic values like NULL, VARCHAR or OTHER. | JdbcType enumeration. Most common are: NULL, VARCHAR and OTHER | OTHER |
| lazyLoadTriggerMethods | Specifies which Object's methods trigger a lazy load | A method name list separated by commas | equals,clone,hashCode,toString |
| defaultScriptingLanguage | Specifies the language used by default for dynamic SQL generation. | A type alias or fully qualified class name. | org.apache.ibatis.scripting.xmltags.XMLDynamicLanguageDriver |
| callSettersOnNulls | 当结果集中含有Null值时是否执行映射对象的setter或者Map对象的put方法。此设置对于原始类型如int,boolean等无效。 | true | false | false |
| logPrefix | Specifies the prefix string that MyBatis will add to the logger names. | Any String | Not set |
| logImpl | Specifies which logging implementation MyBatis should use. If this setting is not present logging implementation will be autodiscovered. | SLF4J | LOG4J | LOG4J2 | JDK\_LOGGING | COMMONS\_LOGGING | STDOUT\_LOGGING | NO\_LOGGING | Not set |
| proxyFactory | Specifies the proxy tool that MyBatis will use for creating lazy loading capable objects. | CGLIB | JAVASSIST | CGLIB |

一个设置信息元素的示例，完全的配置如下所示：

<settings>  
<setting name="cacheEnabled" value="true" />  
<setting name="lazyLoadingEnabled" value="true" />  
<setting name="multipleResultSetsEnabled" value="true" />  
<setting name="useColumnLabel" value="true" />  
<setting name="useGeneratedKeys" value="false" />  
<setting name="autoMappingBehavior" value="PARTIAL" />  
<setting name="defaultExecutorType" value="SIMPLE" />  
<setting name="defaultStatementTimeout" value="25" />  
<setting name="safeRowBoundsEnabled" value="false" />  
<setting name="mapUnderscoreToCamelCase" value="false" />  
<setting name="localCacheScope" value="SESSION" />  
<setting name="jdbcTypeForNull" value="OTHER" />  
<setting name="lazyLoadTriggerMethods" value="equals,clone,hashCode,toString" />  
</settings>

### typeAliases

对model类进行别名的定义

#### mybatis支持的别名

|  |  |
| --- | --- |
| 别名 | 映射的类型 |
| \_byte | byte |
| \_long | long |
| \_short | short |
| \_int | int |
| \_integer | int |
| \_double | double |
| \_float | float |
| \_boolean | boolean |
| string | String |
| byte | Byte |
| long | Long |
| short | Short |
| int | Integer |
| integer | Integer |
| double | Double |
| float | Float |
| boolean | Boolean |
| date | Date |
| decimal | BigDecimal |
| bigdecimal | BigDecimal |

#### 自定义别名

<!-- 自定义别名 -->  
<typeAliases>  
<!--单个别名定义-->  
<!-- <typeAlias type="com.zhiyou100.model.User" alias="User"/> -->  
<!-- 批量定义别名 (建议使用)-->  
<!-- package:指定包名称来为该包下的model类声明别名，默认的别名就是类型，首字符大小写均可-->  
<package name="com.zhiyou100.model"/>  
</typeAliases>

<!-- UserMapper.xml中使用别名 -->

<select id="findUserByUsername" parameterType="java.lang.String" resultType="User">  
SELECT \* FROM tb\_user WHERE username LIKE '%${value}%'  
</select>

### mappers

#### <mapper resource=’’/>

使用相对于类路径的资源

如：<mapper resource="sqlmap/User.xml" />

#### <mapper url=’’/>

使用完全限定路径

如：<mapper url="file:///D:\workspace\_spingmvc\mybatis\_01\config\sqlmap\User.xml" />

#### <mapper class=’’/>

使用mapper接口的全限定名

如：<mapper class="com.zhiyou100.mapper.UserMapper"/>

**注意：此种方法要求mapper接口和mapper映射文件要名称相同，且放到同一个目录下**；

#### <package name=’’/>（推荐）

注册指定包下的所有映射文件

如：<package name="com.zhiyou100.mapper"/>

**注意：此种方法要求mapper接口和mapper映射文件要名称相同，且放到同一个目录下**；

# 映射文件

## 输入映射

### 简单数据类型作为查询条件

参考入门程序之根据用户ID查询用户信息的映射文件

### model对象作为查询条件

User对象作为查询条件 参考入门程序之添加用户的映射文件

### 自定义对象类型作为查询条件

自定义一个查询对象作为查询条件

#### 需求

综合查询时，可能会根据用户信息、商品信息、订单信息等作为条件进行查询，用户信息中的查询条件由：用户的名称和性别进行查询

#### 创建包装类

public class UserQuery {  
  
private User user;  
  
 public UserQuery(User user) {  
this.user = user;  
}  
  
public User getUser() {  
return user;  
}  
  
public void setUser(User user) {  
this.user = user;  
}  
}

#### 映射文件

<!--综合查询 查询用户列表 -->  
<select id="findUserList" parameterType="com.zhiyou100.model.UserQuery" resultType="com.zhiyou100.model.User">  
SELECT \* FROM tb\_user WHERE username LIKE '%${user.username}%'  
 AND sex = #{user.sex}  
</select>

#### Mapper接口

*/\*\*  
 \* 综合查询  
\** ***@param*** *query  
\** ***@return****\*/*List<User>findUserList(UserQuery query);

#### 测试代码

@Test  
public void findUserList() throws Exception {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
User user = new User();  
user.setUsername("王五");  
user.setSex("男");  
List<User> list = userMapper.findUserList(new UserQuery(user));  
System.*out*.println(list);  
}

### Map作为查询条件

同传递POJO对象一样，**map的key相当于pojo的属性**。

#### 映射文件

<!-- 传递hashmap综合查询用户信息 -->  
<select id="findUserByHashmap" parameterType="hashmap" resultType="com.zhiyou100.model.User">  
select \* from tb\_user WHERE id = #{id} and username like '%${username}%'  
</select>

#### Mapper接口

*/\*\*  
 \* 综合查询  
\** ***@param*** *params  
 \** ***@return****\*/*List<User>findUserByHashmap(Map<String,Object>params);

#### 测试代码

@Test  
public void findUserByHashmap() throws Exception {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
// 构造查询条件  
Map<String,Object> map = new HashMap<String,Object>();  
map.put("id", 1);  
map.put("username", "张三");  
List<User> list = userMapper.findUserByHashmap(map);  
System.*out*.println(list);

sqlSession.close();  
}

## 输出映射

### resultType

#### 使用要求

使用resultType进行结果映射时，需要查询出的列名和映射的对象的属性名一致，才能映射成功。

如果查询的列名和对象的属性名全部不一致，那么映射的对象为空。

如果查询的列名和对象的属性名有一个一致，那么映射的对象不为空，但是只有映射正确那一个属性才有值。

如果查询的sql的列名有别名，那么这个别名就是和属性映射的列名。

#### 返回简单数据类型

注意，对简单类型的结果映射也是有要求的，查询的列必须是一列，才能映射为简单类型。

##### 需求

综合查询 查询满足条件的用户数量

##### 映射文件

<!-- 综合查询用户的总数 -->  
<select id="findUserCount" parameterType="com.zhiyou100.model.UserQuery" resultType="java.lang.Integer">  
select *count*(\*) from tb\_user WHERE username LIKE '%${user.username}%'  
 AND sex = #{user.sex}  
</select>

##### Mapper接口

*/\*\*  
 \* 综合查询 查询满足条件的用户数量  
\** ***@param*** *query  
\** ***@return****\*/*int findUserCount(UserQuery query);

##### 测试代码

@Test  
public void findUserCount() throws Exception {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
User user = new User();  
user.setUsername("王五");  
user.setSex("男");  
 int count = userMapper.findUserCount(new UserQuery(user));  
System.*out*.println(count);

sqlSession.close();  
}

##### 练习：通过用户id查询用户的名称

#### 返回对象和对象集合

参考入门程序之根据用户ID查询用户信息和根据用户名称模糊查询用户列表

### resultMap

#### 使用要求

使用resultMap进行结果映射时，不需要查询的列名和映射的属性名必须一致。但是需要声明一个resultMap，来对列名和属性名进行映射。

#### 需求

对以下sql查询的结果集进行对象映射

**SELECT**id id\_,  
username username\_,  
sex sex\_  
**FROM**tb\_user  
**WHERE**id = 1

#### 映射文件

<!--  
定义resultMap  
将SELECT id id\_,username username\_,sex sex\_ FROM tb\_user WHERE id = 1 和 User类中的属性作一个映射关系  
 type:resultMap最终映射的java对象类型，可以使用别名  
 id：对resultMap的唯一标识  
-->  
<resultMap id="UserResultMap" type="com.zhiyou100.model.User">  
<!--  
 id表示查询结果集中的唯一标识  
 column:查询出来的列名  
 property:type指定的pojo类型中的属性名  
 最终resultMap对column和property作一个映射关系  
-->  
<id column="id\_" property="id" />  
<!--  
 result:对普通名映射定义  
 column:查询出来的列名  
 property:type指定的pojo类型中的属性名  
 最终resultMap对column和property作一个映射关系  
-->  
<result column="username\_" property="username" />  
<result column="sex\_" property="sex" />  
<result column="birthday" property="birthday" />  
<result column="address" property="address" />  
</resultMap>

<select id="findUserResultMapById" parameterType="java.lang.Long" resultMap="UserResultMap">  
SELECT \* FROM tb\_user WHERE id = #{id}  
</select>

#### Mapper接口

*/\*\*  
 \* 根据用户id查询用户  
\** ***@param*** *id  
\** ***@return****\*/*User findUserResultMapB(Long id);

#### 测试代码

@Test  
public void findUserResultMapById() throws Exception {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
User user = userMapper.findUserResultMapById(1L);  
System.*out*.println(user);

sqlSession.close();  
}

### 动态sql

在mybatis中，它提供了一些动态sql标签，可以让程序员更快的进行mybatis的开发，这些动态sql可以通过sql的可重用性。。

常用的动态sql标签：if标签、where标签、sql片段、foreach标签

#### If标签/where标签

##### 需求

综合查询时，查询条件由用户来输入，用户名称可以为空，需要满足这种情况下的sql编写。

##### 映射文件

<select id="findUserList" parameterType="com.zhiyou100.model.UserQuery" resultMap="UserResultMap">  
SELECT id id\_,username username\_,sex sex\_ FROM tb\_user  
<!-- where标签可以去掉查询条件中的第一个and关键字 -->  
<where>  
<if test="user.username !=null and user.username !='' ">  
AND username LIKE '%${user.username}%'  
</if>  
<if test="user.sex!=null and user.sex!='' ">  
AND sex = #{user.sex}  
</if>  
</where>  
</select>

##### 测试代码

@Test  
public void findUserList() throws Exception {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
User user = new User();  
user.setUsername("张三");  
user.setSex("男");  
List<User> list = userMapper.findUserList(new UserQuery(user));  
System.*out*.println(list);

sqlSession.close();  
}

#### Sql片段

Sql片段可以让代码有更高的可重用性

Sql片段需要先定义后使用

<select id="findUserList" parameterType="com.zhiyou100.model.UserQuery" resultMap="UserResultMap">  
SELECT id id\_,username username\_,sex sex\_ FROM tb\_user  
<where>  
<!-- 引入SQL片段 -->  
<include refid="query\_user\_where" />  
<!-- 还可以引用其他sql片段 -->  
</where>  
</select>  
<!--  
定义sql片段 id：唯一标识  
 经验：是基于单表来定义sql片段，这样这个sql片段重用性才高 在sql片段中不要包括where关键字  
-->  
<sql id="query\_user\_where">  
<if test="user.username !=null and user.username !='' ">  
AND username LIKE '%${user.username}%'  
</if>  
<if test="user.sex!=null and user.sex!='' ">  
AND sex = #{user.sex}  
</if>  
</sql>

#### foreach标签

可以循环传入参数值

##### 需求

综合查询时，会根据用户ID集合进行查询

SELECT \* FROM TB\_USER WHERE id IN (1,2,10)

SELECT \* FROM TB\_USER WHERE (id =1 or id=2 or id=10);

##### 修改包装类UserQuery

public class UserQuery {  
  
private User user;  
// 用户id的集合  
private List<Long>ids;  
  
 public UserQuery(User user) {  
this.user = user;  
}  
  
public User getUser() {  
return user;  
}  
  
public void setUser(User user) {  
this.user = user;  
}  
  
public List<Long>getIds() {  
return ids;  
}  
  
public void setIds(List<Long> ids) {  
this.ids = ids;  
}  
}

##### 映射文件

<select id="findUserList" parameterType="com.zhiyou100.model.UserQuery" resultMap="UserResultMap">  
SELECT id id\_,username username\_,sex sex\_ FROM tb\_user  
<where>  
<!-- 引入SQL片段 -->  
<include refid="query\_user\_where" />  
<!-- 还可以引用其他sql片段 -->  
</where>  
</select>  
<!--  
定义sql片段 id：唯一标识  
 经验：是基于单表来定义sql片段，这样这个sql片段重用性才高 在sql片段中不要包括where关键字  
-->  
<sql id="query\_user\_where">  
<if test="user.username !=null and user.username !='' ">  
AND username LIKE '%${user.username}%'  
</if>  
<if test="user.sex!=null and user.sex!='' ">  
AND sex = #{user.sex}  
</if>  
<if test="user.ids !=null ">  
<!--  
使用foreach遍历传入ids  
 collection:指定输入对象中集合属性  
 item:每次遍历生成对象  
 open：开始遍历时拼接的串  
 close：结束遍历时拼接的串  
 separator分隔符:遍历的两个对象中需要拼接的串  
-->  
<!--  
使用实现下边的sql拼接  
 AND (ID =1 or ID=10 or ID=16)  
<foreach collection="ids" item="user\_id" open="and(" close=")" separator="or">  
 id=#{user\_id}  
</foreach>  
 -->  
<!-- 实现 and id IN(1,10,16)拼接 -->  
<foreach collection="ids" item="user\_id" open=" and id IN(" close=")" separator=",">  
#{user\_id}  
</foreach>  
</if>  
</sql>

##### 测试代码

@Test  
public void findUserList() throws Exception {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
User user = new User();  
user.setSex("男");  
UserQuery query = new UserQuery(user);  
List<Long> ids = new ArrayList<Long>();  
ids.add(1L);  
ids.add(2L);  
query.setIds(ids);  
List<User> list = userMapper.findUserList(query);  
System.*out*.println(list);

sqlSession.close();  
}