mybatis第一天 mybatis基础知识

教学思路：

整个教学思路以订单商品的项目作为驱动。

**mybatis第一天：**基础知识

mybatis是一个java持久层框架，java中操作关系型 数据库用的是jdbc，mybatis是对jdbc的一个封装。

1、从一个jdbc程序开始，通过jdbc 程序找到使用原生态的jdbc开发程序，存在哪些问题？？通过学习mybatis，mybatis是如何解决这些问题。

2、mybatis的架构（重点）

3、mybatis的入门程序（重点）

实现 用户的查询、添加、修改、删除。

4、mybatis开发dao的两种方法。（重点）

原始的dao开发方式（dao接口和dao实现都需要编写）

mapper代理方式（只需要写dao接口）

5、输入映射类型和输出映射类型。

6、动态sql

**第二天：**高级知识

高级映射查询（一对一、一对多、多对多）（重点）

查询缓存

延迟加载

mybatis和spring整合（重点）

mybatis逆向工程 。

# 开发环境

jdk1.7.0\_72

eclipse：eclipse-3.7-indigo

mysql：mysql5.1

## 创建数据库



先导入sql\_table.sql，再导入sql\_data.sql(记录系统的初始化数据)

通常需要提供初始化数据的数据库脚本。



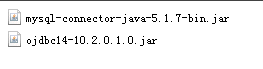
# jdbc编程中问题

企业开发中，根据项目大小、特点进行技术选型 ，jdbc操作数据库时效率是很高的，jdbc也是技术选型的参考。

## jdbc程序

参考教案

需要数据库的驱动包：



上边是mysql的驱动，下边是oracle的驱动。

## jdbc问题总结

1、数据库连接频繁的创建和关闭，缺点浪费数据库的资源，影响操作效率

设想：使用数据库连接池

2、sql语句是硬编码，如果需求变更需要修改sql，就需要修改java代码，需要重新编译，系统不易维护。

设想：将sql语句 统一配置在文件中，修改sql不需要修改java代码。

3、通过preparedStatement向占位符设置参数，存在硬编码（ 参数位置，参数）问题。系统不易维护。

设想：将sql中的占位符及对应的参数类型配置在配置文件中，能够自动输入 映射。

4、遍历查询结果集存在硬编码（列名）。

设想：自动进行sql查询结果向java对象的映射（输出映射）。

# mybatis架构(重点)

## mybatis介绍

MyBatis 本是[apache](http://baike.baidu.com/view/28283.htm)的一个开源项目[iBatis](http://baike.baidu.com/view/628102.htm), 2010年这个项目由apache software foundation 迁移到了google code，并且改名为MyBatis，实质上Mybatis对ibatis进行一些改进。 目前mybatis在github上托管。git（分布式版本控制，当前比较流程）

MyBatis是一个优秀的持久层框架，它对jdbc的操作数据库的过程进行封装，使开发者只需要关注 SQL 本身，而不需要花费精力去处理例如注册驱动、创建connection、创建statement、手动设置参数、结果集检索等jdbc繁杂的过程代码。

Mybatis通过xml或注解的方式将要执行的各种statement（statement、preparedStatemnt、CallableStatement）配置起来，并通过java对象和statement中的sql进行映射生成最终执行的sql语句，最后由mybatis框架执行sql并将结果映射成java对象并返回。

## mybatis架构

SqlMapConfig.xml（mybatis全局配置文件，名称不固定）

运行环境（数据源、事务）

mapper.xml(配置sql语句), mapper.xml.....

SqlSessionFactory（会话工厂）

作用：创建SqlSession

SqlSession（向面用户的接口，数据库操作方法）

作用：操作数据库

Executor（数据库操作的执行器）

Executor是一个接口，有两个实现 （默认执行器和缓存执行器）

mysql

MappedStatement（mybatis的封装对象）

封装：sql语句(占位符)

输入 映射

java简单类型

pojo类型

hashmap

输出 映射

java简单类型

pojo类型

hashmap

# mybatis入门程序

## 需求

实现用户查询：

根据用户id（主键）查询用户信息（单条记录）

根据用户名称模糊查询用户信息（多条记录）

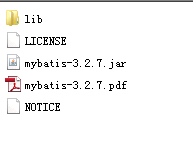
用户添加

用户删除

用户修改

## 导入jar包

从mybatis管网下载（地址：https://github.com/mybatis/mybatis-3/releases）



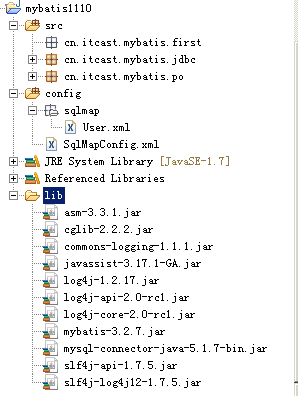
mybatis-3.2.7.pdf---操作手册

mybatis-3.2.7.jar--核心 jar包

依赖的jar包



## 工程结构



## log4j.properties（公用文件）

*## 日志级别  
# FATAL 0  
# ERROR 3  
# WARN 4  
# INFO 6  
# DEBUG 7  
  
# 配置根Logger***log4j.rootLogger**=**DEBUG, Console, DebugFile, ErrorFile***### 输出到控制台 ###  
# ppender 为日志输出目的地***log4j.appender.Console**=**org.apache.log4j.ConsoleAppender***# Threshold：输出？级别以上的日志***log4j.appender.Console.Threshold**=**DEBUG***# Layout：日志输出格式***log4j.appender.Console.layout**=**org.apache.log4j.PatternLayout  
log4j.appender.Console.layout.ConversionPattern**=**%-d{yyyy-MM-dd HH:mm:ss} [%-5p] - [%t:%r] %m %n***### 输出到日志文件 ###***log4j.appender.DebugFile**=**org.apache.log4j.DailyRollingFileAppender  
log4j.appender.DebugFile.Threshold**=**DEBUG  
log4j.appender.DebugFile.File**=**F:/Log/temp/DebugFile.log  
log4j.appender.DebugFile.Append** = **true  
log4j.appender.DebugFile.layout**=**org.apache.log4j.PatternLayout  
log4j.appender.DebugFile.layout.ConversionPattern**=**%-d{yyyy-MM-dd HH:mm:ss} [%-5p] - [%t:%r] %m %n***### 保存异常信息到单独文件 ###***log4j.appender.ErrorFile**=**org.apache.log4j.DailyRollingFileAppender  
log4j.appender.ErrorFile.Threshold**=**ERROR  
log4j.appender.ErrorFile.File**=**F:/Log/temp/ErrorFile.log  
log4j.appender.ErrorFile.Append** = **true  
log4j.appender.ErrorFile.layout**=**org.apache.log4j.PatternLayout  
log4j.appender.ErrorFile.layout.ConversionPattern**=**%-d{yyyy-MM-dd HH:mm:ss} [%-5p] - [%t:%r] %m %n**

## SqlMapConfig.xml(公用文件)

通过SqlMapConfig.xml加载mybatis运行环境。

*<?***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**>  
  
 <**properties resource="dbconfig.properties"**/>  
  
 *<!-- 和spring整合后 environments配置将废除-->* <**environments default="development"**>  
 <**environment id="development"**>  
 *<!-- 使用jdbc事务管理-->* <**transactionManager type="JDBC"**/>  
 *<!-- 数据库连接池-->* <**dataSource type="POOLED"**>  
 <**property name="driver" value="${jdbc.driver}"**/>  
 <**property name="url" value="${jdbc.url}"**/>  
 <**property name="username" value="${jdbc.username}"**/>  
 <**property name="password" value="${jdbc.password}"**/>  
 </**dataSource**>  
 </**environment**>  
 </**environments**>

## 根据id查询用户

### pojo（User.java）

**private int id**;  
**private** String **username**;  
**private** String **sex**;  
**private** Date **birthday**;  
**private** String **address**;

### User.xml（重点）

建议命名规则：表名+mapper.xml

早期ibatis命名规则：表名.xml

*<?***xml version="1.0" encoding="UTF-8"** *?>***<!DOCTYPE mapper  
 PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd"*>***<**mapper namespace="test"**>  
 *<!-- 在mapper.xml文件中配置很多的sql语句，执行每个sql语句时，封装为MappedStatement对象  
 mapper.xml以statement为单位管理sql语句  
 -->  
  
 <!-- 根据id查询用户信息 -->  
 <!--  
 id：唯一标识 一个statement  
 #{}：表示 一个占位符，如果#{}中传入简单类型的参数，#{}中的名称随意  
 parameterType：输入 参数的类型，通过#{}接收parameterType输入 的参数  
 resultType：输出结果 类型，不管返回是多条还是单条，指定单条记录映射的pojo类型  
 -->* <**select id="findUserById" parameterType="int" resultType="cn.qdgxy.mybatis.po.User"**>  
 SELECT *\** FROM USER WHERE id= #{id}  
 </**select**>  
</**mapper**>

### 编码

创建SqlSessionFactory:

**private** SqlSessionFactory **sqlSessionFactory**;  
  
@Before  
**public void** init() **throws** IOException {  
 *// 配置文件* String resource = **"SqlMapConfig.xml"**;  
  
 *// 加载配置文件到输入流* InputStream inputStream = Resources.*getResourceAsStream*(resource);  
  
 *// 创建会话工厂* **sqlSessionFactory** = **new** SqlSessionFactoryBuilder().build(inputStream);  
}  
  
*/\*\*  
 \* 通过ID获得一个对象  
 \*/*@Test  
**public void** testFindById() {  
 *// 通过sqlSession操作数据库  
 // 第一个参数：statement的位置，等于namespace+statement的id  
 // 第二个参数：传入的参数* User user = **null**;  
 *// 通过sqlSessionFactory创建sqlSession* **try** (SqlSession sqlSession = **sqlSessionFactory**.openSession()) {  
 user = sqlSession.selectOne(**"test.findUserById"**, 1);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(user);  
}

## 根据用户名称模糊查询用户信息

根据用户名称模糊查询用户信息可能返回多条记录。

### User.xml

*<!-- 根据用户名称查询用户信息，可能返回多条  
${}：表示sql的拼接，通过${}接收参数，将参数的内容不加任何修饰拼接在sql中。-->*<**select id="findUserByName" parameterType="java.lang.String" resultType="cn.qdgxy.mybatis.po.User"**>  
 select *\** from user where username like #{id}  
 </**select**>

### 编码

*/\*\*  
 \* 通过用户名获得多个对象  
 \*/*@Test  
**public void** testFindByUsername() {  
 *// 通过sqlSession操作数据库  
 // 第一个参数：statement的位置，等于namespace+statement的id  
 // 第二个参数：传入的参数* List<User> users = **null**;  
  
 *// 通过sqlSessionFactory创建sqlSession* **try** (SqlSession sqlSession = **sqlSessionFactory**.openSession()) {  
 users = sqlSession.selectList(**"test.findUserByName"**, **"小明"**);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
  
 **if** (users != **null**) {  
 **for** (User user : users) {  
 System.***out***.println(user);  
 }  
 }  
}

### 使用${}接收参数

<*<!-- 根据用户名称查询用户信息，可能返回多条  
${}：表示sql的拼接，通过${}接收参数，将参数的内容不加任何修饰拼接在sql中。-->*<**select id="findUserByName" parameterType="java.lang.String" resultType="cn.qdgxy.mybatis.po.User"**>  
 select *\** from user where username like '%${value}%'  
 </**select**>

## mybatis开发过程小结

1、编写SqlMapConfig.xml

2、编写mapper.xml

定义了statement

3、编程通过配置文件创建SqlSessionFactory

4、通过SqlSessionFactory获取SqlSession

5、通过SqlSession操作数据库

如果执行添加、更新、删除需要调用SqlSession.commit()

6、SqlSesion使用完成要关闭

## 用户添加

向用户表插入一条记录。

### User.xml

*<!-- 添加用户  
parameterType：输入 参数的类型，User对象 包括 username,birthday,sex,address  
  
#{}接收pojo数据，可以使用OGNL解析出pojo的属性值  
#{username}表示从parameterType中获取pojo的属性值*<**insert id="insertUser" parameterType="cn.qdgxy.mybatis.po.User"**>  
 INSERT INTO USER(username,birthday,sex,address) VALUES(#{username},#{birthday},#{sex},#{address})  
</**insert**>

### 编码

*/\*\*  
 \* 插入用户  
 \*/*@Test  
**public void** testInsertUser() **throws** Exception {  
 *// 通过sqlSession操作数据库  
 // 插入用户* User user = **new** User();  
 user.setUsername(**"张三"**);  
 user.setAddress(**"北京"**);  
 user.setBirthday(**new** Date());  
 user.setSex(**"1"**);  
  
 **try** (SqlSession sqlSession = **sqlSessionFactory**.openSession()) {  
 sqlSession.insert(**"test.insertUser"**, user);  
  
 *// 需要提交事务* sqlSession.commit();  
 }  
  
 *// 打印ID* System.***out***.println(user);  
}

### 主键返回

需求：user对象插入到数据库后，新记录的主键要通过user对象返回，通过user获取主键值。

解决思路：

通过LAST\_INSERT\_ID()获取刚插入记录的自增主键值，在insert语句执行后，执行select LAST\_INSERT\_ID()就可以获取自增主键。

User.xml修改：

*<!-- 添加用户  
parameterType：输入 参数的类型，User对象 包括 username,birthday,sex,address  
  
#{}接收pojo数据，可以使用OGNL解析出pojo的属性值  
#{username}表示从parameterType中获取pojo的属性值  
  
selectKey：用于进行主键返回，定义了获取主键值的sql  
order：设置selectKey中sql执行的顺序，相对于insert语句来说  
keyProperty：将主键值设置到哪个属性  
resultType：select LAST\_INSERT\_ID()的结果 类型  
 -->*<**insert id="insertUser" parameterType="cn.qdgxy.mybatis.po.User"**>  
 <**selectKey keyProperty="id" order="AFTER" resultType="int"**>  
 select LAST\_INSERT\_ID()  
 </**selectKey**>  
  
 INSERT INTO USER(username,birthday,sex,address) VALUES(#{username},#{birthday},#{sex},#{address})  
</**insert**>

**使用mysql的uuid机制生成主键**：

使用uuid生成主键的好处是不考虑数据库移植后主键冲突问题。

实现思路：

先查询uuid得到主键，将主键设置到user对象中，将user对象插入数据库。

*<!-- mysql的uuid生成主键 -->*<**insert id="insertUser" parameterType="cn.qdgxy.mybatis.po.User"**>  
 <**selectKey keyProperty="id" order="BEFORE" resultType="string"**>  
 select uuid()  
 </**selectKey**>  
  
 INSERT INTO USER(id,username,birthday,sex,address) VALUES(#{id},#{username},#{birthday},#{sex},#{address})  
 </**insert**>

**实现oracle数据库主键返回，如何做？？**

oracle没有自增主键机制，使用序列完成主键生成。

实现思路：

先查询序列得到主键，将主键设置到user对象中，将user对象插入数据库。

*<!-- oracle  
在执行insert之前执行select 序列.nextval() from dual取出序列最大值，将值设置到user对象 的id属性  
 -->*<**insert id="insertUser" parameterType="cn.qdgxy.mybatis.po.User"**>  
 <**selectKey keyProperty="id" order="BEFORE" resultType="int"**>  
 select 序列.nextval() from dual  
 </**selectKey**>  
  
 INSERT INTO USER(id,username,birthday,sex,address) VALUES(#{id},#{username},#{birthday},#{sex},#{address})  
</**insert**>

## 用户删除和更新

*<!-- 用户删除 -->*<**delete id="deleteUser" parameterType="int"**>  
 delete from user where id=#{id}  
 </**delete**>  
  
*<!-- 用户更新  
要求：传入的user对象中包括 id属性值  
-->*<**update id="updateUser" parameterType="cn.qdgxy.mybatis.po.User"**>  
 update user set username=#{username},birthday=#{birthday},sex=#{sex},address=#{address} where id=#{id}  
</**update**>

### 编码

*/\*\*  
 \* 删除用户  
 \*/*@Test  
**public void** testdeleteUser() {  
 *// 通过sqlSessionFactory创建sqlSession* **try** (SqlSession sqlSession = **sqlSessionFactory**.openSession()) {  
 sqlSession.delete(**"test.deleteUser"**, 28);  
  
 *// 需要提交事务* sqlSession.commit();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
}  
  
*/\*\*  
 \* 更新用户  
 \*/*@Test  
**public void** updateUser() {  
 User user = **new** User();  
 user.setId(26);  
 user.setUsername(**"王五"**);  
 user.setAddress(**"上海"**);  
 user.setBirthday(**new** Date());  
 user.setSex(**"1"**);  
  
 *// 通过sqlSessionFactory创建sqlSession* **try** (SqlSession sqlSession = **sqlSessionFactory**.openSession()) {  
 sqlSession.update(**"test.updateUser"**, user);  
  
 *// 需要提交事务* sqlSession.commit();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
}

## Mybatis解决jdbc编程的问题

1. 数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。

解决：在SqlMapConfig.xml中配置数据链接池，使用连接池管理数据库链接。

1. Sql语句写在代码中造成代码不易维护，实际应用sql变化的可能较大，sql变动需要改变java代码。

解决：将Sql语句配置在XXXXmapper.xml文件中与java代码分离。

1. 向sql语句传参数麻烦，因为sql语句的where条件不一定，可能多也可能少，占位符需要和参数一一对应。

解决：Mybatis自动将java对象映射至sql语句，通过statement中的parameterType定义输入参数的类型。

1. 对结果集解析麻烦，sql变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成pojo对象解析比较方便。

解决：Mybatis自动将sql执行结果映射至java对象，通过statement中的resultType定义输出结果的类型。

## mybatis与hibernate重要区别

企业开发进行技术选型 ，考虑mybatis与hibernate适用场景。

**mybatis：**

① 入门简单，程序容易上手开发，节省开发成本 。

② mybatis需要程序员自己编写sql语句，是一个不完全的ORM框架，对sql修改和优化非常容易实现 。

③ mybatis适合开发需求变更频繁的系统，比如：互联网项目。

**hibernate：**

① 入门门槛高，如果用hibernate写出高性能的程序不容易实现。

② hibernate不用写sql语句，是一个 ORM框架。

③ hibernate适合需求固定，对象数据模型稳定，中小型项目，比如：企业OA系统。

总之，企业在技术选型时根据项目实际情况，以降低成本和提高系统可维护性为出发点进行技术选型。

## 总结

### SqlMapConfig.xml

是mybatis全局配置文件，只有一个，名称不固定的，主要mapper.xml，mapper.xml中配置 sql语句

### mapper.xml

mapper.xml是以statement为单位进行配置。（把一个sql称为一个statement），satatement中配置sql语句、parameterType输入参数类型（完成输入映射）、resultType输出结果类型（完成输出映射）。

还提供了parameterMap配置输入参数类型（过期了，不推荐使用了）

还提供resultMap配置输出结果类型（完成输出映射），明天重点讲通过resultMap完成复杂数据类型的映射（一对多，多对多映射）

### #{}

表示一个占位符，向占位符输入参数，mybatis自动进行java类型和jdbc类型的转换。

程序员不需要考虑参数的类型，比如：传入字符串，mybatis最终拼接好的sql就是参数两边加单引号。

#{}接收pojo数据，可以使用OGNL解析出pojo的属性值

### ${}

表示sql的拼接，通过${}接收参数，将参数的内容不加任何修饰拼接在sql中。

${}也可以接收pojo数据，可以使用OGNL解析出pojo的属性值

缺点：不能防止sql注入。

### selectOne

用于查询单条记录，不能用于查询多条记录，否则异常：

org.apache.ibatis.exceptions.TooManyResultsException: Expected one result (or null) to be returned by selectOne(), but found: 4

## selectList

用于查询多条记录，可以用于查询单条记录的。

# mybatis开发dao的方法

## SqlSession作用范围

是使用局部变量、成员变量。。。。？？？

### SqlSessionFactoryBuilder

SqlSessionFactoryBuilder是以工具类方式来使用，需要创建sqlSessionFactory就new一个SqlSessionFactoryBuilder。

### sqlSessionFactory

正常开发时，以单例方式管理sqlSessionFactory，整个系统运行过程中sqlSessionFactory只有一个实例，将来和spring整合后由spring以单例方式管理sqlSessionFactory。

### SqlSession

sqlSession是一个面向用户（程序员）的接口，程序员调用sqlSession的接口方法进行操作数据库。

sqlSession能否以单例方式使用？？

**由于sqlSession是线程不安全，所以sqlSession最佳应用范围在方法体内，在方法体内定义局部变量使用sqlSession。**

## 原始dao开发方式

程序员需要写dao接口和dao 的实现 类

### dao接口

/*/\*\*  
 \* 用户Dao 接口<br>  
 \* create:2016-10-16 17:43  
 \*  
 \** ***@author*** *李欣  
 \** ***@version*** *1.0  
 \*/***public interface** UserDao {  
  
 *//根据id查询用户信息* **public** User findUserById(**int** id) **throws** Exception;  
  
}

### dao接口实现

*/\*\*  
 \* 用户Dao实现  
 \* create:2016-10-16 17:46  
 \*  
 \** ***@author*** *李欣  
 \** ***@version*** *1.0  
 \*/***public class** UserDaoImpl **implements** UserDao {  
  
 *// 将sqlSessionFactory注入* **private** SqlSessionFactory **sqlSessionFactory**;  
  
 **public** UserDaoImpl(SqlSessionFactory sqlSessionFactory) {  
 **this**.**sqlSessionFactory** = sqlSessionFactory;  
 }  
  
 @Override  
 **public** User findUserById(**int** id) **throws** Exception {  
 *//创建sqlSession* SqlSession sqlSession = **sqlSessionFactory**.openSession();  
 User user = sqlSession.selectOne(**"test.findUserById"**, id);  
 sqlSession.close();  
 **return** user;  
 }  
  
}

### 测试代码

*/\*\*  
 \* ${DESCRIPTION}  
 \* create:2016-10-16 17:50  
 \*  
 \** ***@author*** *李欣  
 \** ***@version*** *${VERSION}  
 \*/***public class** UserDaoImplTest {  
  
 **private** SqlSessionFactory **sqlSessionFactory**;  
  
 @Before  
 **public void** init() **throws** IOException {  
 *// 配置文件* String resource = **"SqlMapConfig.xml"**;  
  
 *// 加载配置文件到输入流* InputStream inputStream = Resources.*getResourceAsStream*(resource);  
  
 *// 创建会话工厂* **sqlSessionFactory** = **new** SqlSessionFactoryBuilder().build(inputStream);  
 }

}

## mapper代理的方式

程序员只需要写dao接口，dao接口实现对象由mybatis自动生成代理对象。

本身dao在三层架构中就是一个通用的接口。

### 上边原始dao开发方式的问题

1 . dao的实现类中存在重复代码，整个mybatis操作的过程代码模板重复（先创建sqlsession、调用sqlsession的方法、关闭sqlsession）

2. dao的实现 类中存在硬编码，调用sqlsession方法时将statement的id硬编码。

### mapper开发规范

要想让mybatis自动创建dao接口实现类的代理对象，必须遵循一些规则：

1、mapper.xml中namespace指定为mapper接口的全限定名

*<!-- namespace命名空间，为了对sql语句进行隔离，方便管理 ，mapper开发dao方式，使用namespace有特殊作用  
mapper代理开发时将namespace指定为mapper接口的全限定名  
 -->*<**mapper namespace="cn.qdgxy.mybatis.mapper.UserMapper"**>

此步骤目的：通过mapper.xml和mapper.java进行关联。

2、mapper.xml中statement的id就是mapper.java中方法名

3、mapper.xml中statement的parameterType和mapper.java中方法输入参数类型一致

4、mapper.xml中statement的resultType和mapper.java中方法返回值类型一致.

### mapper.xml（映射文件）

mapper映射文件的命名方式建议：表名Mapper.xml

namespace指定为mapper接口的全限定名

*!-- 根据id查询用户信息 -->  
<!--  
 id：唯一标识 一个statement  
 #{}：表示 一个占位符，如果#{}中传入简单类型的参数，#{}中的名称随意  
 parameterType：输入 参数的类型，通过#{}接收parameterType输入 的参数  
 resultType：输出结果 类型，不管返回是多条还是单条，指定单条记录映射的pojo类型  
 -->*<**select id="findUserById" parameterType="int" resultType="user"**>  
 SELECT *\** FROM USER WHERE id= #{id}  
</**select**>

### mapper接口

mybatis提出了mapper接口，相当 于dao 接口。

mapper接口的命名方式建议：表名Mapper

*/\*\*  
 \* 用户Dao  
 \* create:2016-10-16 17:43  
 \*  
 \** ***@author*** *李欣  
 \** ***@version*** *1.0  
 \*/***public interface** UserMapper {  
  
 *//根据id查询用户信息* User findUserById(**int** id) **throws** Exception;  
  
}

### 将mapper.xml在SqlMapConfig.xml中加载

<**mappers**>  
 *<!-- 通过resource引用mapper的映射文件 -->* <**mapper resource="sqlmap/User.xml"**/>  
 <**mapper resource="mapper/UserMapper.xml"**/></**mappers**>

### mapper接口返回单个对象和集合对象

不管查询记录是单条还是多条，在 statement中resultType定义一致，都是单条记录映射的pojo类型。

mapper接口方法返回值，如果是返回的单个对象，返回值类型是pojo类型，生成的代理对象内部通过selectOne获取记录，如果返回值类型是集合对象，生成的代理对象内部通过selectList获取记录。**public interface** UserMapper {  
  
 *//根据id查询用户信息* User findUserById(**int** id) **throws** Exception;  
  
 *//根据用户名称模糊查询用户列表* List<User> findUserByName(String username) **throws** Exception;

}

### 问题

#### 返回值的问题

如果方法调用的statement，返回是多条记录，而mapper.java方法的返回值为pojo，此时代理对象通过selectOne调用，由于返回多条记录，所以报错：

org.apache.ibatis.exceptions.TooManyResultsException: Expected one result (or null) to be returned by selectOne(), but found: 4

#### 输入参数的问题

使用mapper代理的方式开发，mapper接口方法输入 参数只有一个，可扩展性是否很差？？

可扩展性没有问题，因为dao层就是通用的，可以通过扩展pojo（定义pojo包装类型）将不同的参数（可以是pojo也可以简单类型）传入进去。

# sqlMapConfig.xml

SqlMapConfig.xml中配置的内容和顺序如下：

**properties（属性）**

settings（全局配置参数）

**typeAliases（类型别名）**

typeHandlers（类型处理器）

objectFactory（对象工厂）

plugins（插件）

environments（环境集合属性对象）

environment（环境子属性对象）

transactionManager（事务管理）

dataSource（数据源）

**mappers（映射器）**

## properties属性定义

可以把一些通用的属性值配置在属性文件中，加载到mybatis运行环境内。

比如：创建db.properties配置数据库连接参数。

<**properties resource="dbconfig.properties"**/>  
  
*<!-- 和spring整合后 environments配置将废除-->*<**environments default="development"**>  
 <**environment id="development"**>  
 *<!-- 使用jdbc事务管理-->* <**transactionManager type="JDBC"**/>  
 *<!-- 数据库连接池-->* <**dataSource type="POOLED"**>  
 <**property name="driver" value="${jdbc.driver}"**/>  
 <**property name="url" value="${jdbc.url}"**/>  
 <**property name="username" value="${jdbc.username}"**/>  
 <**property name="password" value="${jdbc.password}"**/>  
 </**dataSource**>  
 </**environment**>  
</**environments**>

注意： MyBatis 将按照下面的顺序来加载属性：

* 在 properties 元素体内定义的属性首先被读取。
* 然后会读取properties 元素中resource或 url 加载的属性，它会覆盖已读取的同名属性。
* 最后读取parameterType传递的属性，它会覆盖已读取的同名属性。

建议使用properties，不要在properties中定义属性，只引用定义的properties文件中属性，并且properties文件中定义的key要有一些特殊的规则。

## settings全局参数配置

mybatis运行时可以调整一些全局参数（相当于软件的运行参数），参考：mybatis-settings.xlsx

根据使用需求进行参数配置。

注意：小心配置，配置参数会影响mybatis的执行。

ibatis的全局配置参数中包括很多的性能参数（最大线程数，最大待时间。。。），通过调整这些性能参数使ibatis达到高性能的运行，mybatis没有这些性能参数，由mybatis自动调节。

## typeAliases(常用)

可以将parameterType、resultType中指定的类型 通过别名引用。

### mybaits提供了很多别名

|  |  |
| --- | --- |
| 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

*<!-- 定义 别名 -->*<**typeAliases**>  
 *<!--  
 【单个别名的定义】  
 alias：别名，type：别名映射的类型 -->  
 <!--<typeAlias type="cn.qdgxy.mybatis.po.User" alias="user"/>-->  
 <!-- 【批量别名定义】  
 指定包路径，自动扫描包下边的pojo，定义别名，别名默认为类名（首字母小写或大写）  
 -->* <**package name="cn.qdgxy.mybatis.po"**/>  
</**typeAliases**>

### 使用别名

在parameterType、resultType中使用别名：

<**select id="findUserById" parameterType="int" resultType="user"**>  
 SELECT *\** FROM USER WHERE id= #{id}  
</**select**>

### typeHandlers

类型处理器将java类型和jdbc类型进行映射。

mybatis默认提供很多类型处理器，一般情况下够用了。

### mappers

*<!--加载mapper映射  
 如果将和spring整合后，可以使用整合包中提供的mapper扫描器，此处的mappers不用配置了。  
 -->* <**mappers**>  
 *<!-- 【通过resource引用mapper的映射文件、 -->* <**mapper resource="sqlmap/User.xml"**/>  
 *<!--<mapper resource="mapper/UserMapper.xml"/>-->  
  
 <!-- 【通过class引用mapper接口】  
 class：配置mapper接口全限定名  
 要求：需要mapper.xml和mapper.java同名并且在一个目录中  
 -->  
 <!--* <mapper class="cn.qdgxy.mybatis.mapper.UserMapper"/>*-->  
  
 <!-- 【批量mapper配置】  
 通过package进行自动扫描包下边的mapper接口，  
 要求：需要mapper.xml和mapper.java同名并且在一个目录 中  
 -->* <**package name="cn.qdgxy.mybatis.mapper"**/>  
 </**mappers**>

# 输入和输出映射

通过parameterType完成输入映射，通过resultType和resultMap完成输出映射。

## parameterType传递pojo包装对象

可以定义pojo包装类型扩展mapper接口输入参数的内容。

需求：

自定义查询条件查询用户信息，需要向statement输入查询条件，查询条件可以有user信息、商品信息。。。。

### 包装类型

*/\*\*  
 \* pojo包装对象，将来在使用时从页面传到Controller、Service、Mapper<br>  
 \* create:2016-10-17 8:32  
 \*  
 \** ***@author*** *李欣  
 \** ***@version*** *1.0  
 \*/***public class** UserQueryVo {  
  
 *// 用户信息* **private** User **user**;  
  
 *// 自定义User扩展类型* **private** UserCustom **userCustom**;

}

### mapper.xml

*<!-- 自定义查询条件查询用户的信息  
 parameterType：指定包装类型  
 %${userCustom.username}%：userCustom是userQueryVo中的属性，通过OGNL获取属性的值  
-->*<**select id="findUserList" parameterType="userQueryVo" resultType="user"**>  
 select *\** from user where username like '%${userCustom.username}%'   
</**select**>

### mapper.java

*//自定义查询条件查询用户信息*List<User> findUserList(UserQueryVo userQueryVo) **throws** Exception;

### 测试

*/\*\*  
 \* 通过包装类型查询用户信息  
 \** ***@throws*** *Exception 异常  
 \*/*@Test  
**public void** findUserList() **throws** Exception {  
 SqlSession sqlSession = **sqlSessionFactory**.openSession();  
  
 *// 创建代理对象* UserMapper userMapper = sqlSession.getMapper(UserMapper.**class**);  
  
 *// 构造查询条件* UserQueryVo userQueryVo = **new** UserQueryVo();  
 UserCustom userCustom = **new** UserCustom();  
 userCustom.setUsername(**"小明"**);  
 userQueryVo.setUserCustom(userCustom);  
  
 List<User> users = userMapper.findUserList(userQueryVo);  
  
 **for** (User user : users) {  
 System.***out***.println(user);  
 }  
  
 sqlSession.close();  
}

### 异常

如果parameterType中指定属性错误，异常，找不到getter方法：

org.apache.ibatis.exceptions.PersistenceException:

### Error querying database. Cause: org.apache.ibatis.reflection.ReflectionException: There is no getter for property named 'userCustom2' in 'class cn.qdgxy.mybatis.po.UserQueryVo'

### Cause: org.apache.ibatis.reflection.ReflectionException: There is no getter for property named 'userCustom2' in 'class cn.qdgxy.mybatis.po.UserQueryVo'

注意：如果将来和spring整合后，不是通过调用getter方法来获取属性值，通过**反射强读取**pojo的属性值。

## resultType

指定输出结果的类型（pojo、简单类型、hashmap..），将sql查询结果映射为java对象 。

### 返回简单类型

mapper.xml

*<!-- 【输出简单类型】  
 功能：自定义查询条件，返回查询记录个数，通常用于实现 查询分页  
-->* <**select id="findUserCount" parameterType="userQueryVo" resultType="int"**>  
 select *count*(*\**) from user where username like '%${userCustom.username}%'  
 </**select**>

mapper.java

*//自定义查询条件查询用户数量***int** findUserCount(UserQueryVo userQueryVo) **throws** Exception;

测试

*/\*\*  
 \* 返回查询记录总数  
 \*  
 \** ***@throws*** *Exception 异常  
 \*/*@Test  
**public void** findUserCount() **throws** Exception {  
 SqlSession sqlSession = **sqlSessionFactory**.openSession();  
  
 *// 创建代理对象* UserMapper userMapper = sqlSession.getMapper(UserMapper.**class**);  
  
 *// 构造查询条件* UserQueryVo userQueryVo = **new** UserQueryVo();  
 UserCustom userCustom = **new** UserCustom();  
 userCustom.setUsername(**"小明"**);  
 userQueryVo.setUserCustom(userCustom);  
  
 **int** count = userMapper.findUserCount(userQueryVo);  
  
 System.***out***.println(count);  
  
 sqlSession.close();  
}

**注意：**

如果查询记录结果集为一条记录且一列再使用返回简单类型。

## resultMap(入门)

resultType ：指定输出结果的类型（pojo、简单类型、hashmap..），将sql查询结果映射为java对象 。

使用resultType注意：sql查询的列名要和resultType指定pojo的属性名相同，指定相同属性方可映射成功，如果sql查询的列名要和resultType指定pojo的属性名全部不相同，list中无法创建pojo对象的。

resultMap：将sql查询结果映射为java对象。

如果sql查询列名和最终要映射的pojo的属性名不一致，使用resultMap将列名和pojo的属性名做一个对应关系 （列名和属性名映射配置）

### resultMap配置

*<!-- 定义resultMap，列名和属性名映射配置  
 id：mapper.xml中的唯一标识  
 type：最终要映射的pojo类型  
-->* <**resultMap id="userListResultMap" type="user"**>  
 *<!-- 列名  
 id\_,username\_,birthday\_  
 id：要映射结果集的唯 一标识 ，称为主键  
 column：结果集的列名  
 property：type指定的哪个属性中  
 -->* <**id column="id\_" property="id"**/>  
 *<!-- result就是普通列的映射配置 -->* <**result column="username\_" property="username"**/>  
 <**result column="birthday\_" property="birthday"**/>  
 </**resultMap**>

### 使用resultMap

*<!-- 使用resultMap作结果映射  
 resultMap：如果引用resultMap的位置和resultMap的定义在同一个mapper.xml，  
 直接使用resultMap的id，如果不在同一个mapper.xml要在resultMap的id前边加namespace  
-->* <**select id="findUserListResultMap" parameterType="userQueryVo" resultMap="userListResultMap"**>  
 select id id\_,username username\_,birthday birthday\_ from user where username like '%${userCustom.username}%'  
 </**select**>

### mapper.java

*// 查询用户 使用resultMap*List<User> findUserListResultMap(**int** id) **throws** Exception;

# 动态sql(重点)

mybatis重点是对sql的灵活解析和处理。

## 需求

将自定义查询条件查询用户列表和查询用户列表总记录数改为动态sql

## if和where

<**select id="findUserList" parameterType="userQueryVo" resultType="user"**>  
 select id,username,birthday from user  
   
 *<!-- where标签相当 于where关键字，可以自动去除第一个and -->* <**where**>  
 *<!-- 如果 userQueryVo中传入查询条件，再进行sql拼接-->  
 <!-- test中userCustom.username表示从userQueryVo读取属性值 -->* <**if test="userCustom!=null"**>  
 <**if test="userCustom.username!=null and userCustom.username!=''"**>  
 and username like '%${userCustom.username}%'  
 </**if**>  
 <**if test="userCustom.sex!=null and userCustom.sex!=''"**>  
 and sex = #{userCustom.sex}  
 </**if**>  
 *<!-- 还有很多查询条件 -->* </**if**>  
 </**where**>  
</**select**>

## sql片段

通过sql片段可以将通用的sql语句抽取出来，单独定义，在其它的statement中可以引用sql片段。

通用的sql语句，常用：where条件、查询列

### sql片段的定义

*<!-- 【将用户查询条件定义为sql片段】* ***建议对单表的查询条件单独抽取sql片段，提高公用性******注意：不要将where标签放在sql片段****-->*<**sql id="query\_user\_where"**>  
 *<!-- 如果 userQueryVo中传入查询条件，再进行sql拼接-->  
 <!-- test中userCustom.username表示从userQueryVo读取属性值-->* <**if test="userCustom!=null"**>  
 <**if test="userCustom.username!=null and userCustom.username!=''"**>  
 and username like '%${userCustom.username}%'  
 </**if**>  
 <**if test="userCustom.sex!=null and userCustom.sex!=''"**>  
 and sex = #{userCustom.sex}  
 </**if**> *<!-- 还有很的查询条件 -->* </**if**>  
</**sql**>

### 引用sql片段

<**select id="findUserList" parameterType="userQueryVo" resultType="user"**>  
 select id,username,birthday from user  
 *<!-- where标签相当 于where关键字，可以自动去除第一个and -->* <**where**>  
 *<!-- 引用sql片段，如果sql片段和引用处不在同一个mapper必须前边加namespace -->* <**include refid="query\_user\_where"**></**include**>  
   
 *<!-- 下边还有很其它的条件 -->  
 <!-- <include refid="其它的sql片段"></include> -->* </**where**>  
</**select**>

*<!-- 【输出简单类型】  
 功能：自定义查询条件，返回查询记录个数，通常用于实现 查询分页  
-->* <**select id="findUserCount" parameterType="userQueryVo" resultType="int"**>  
 select *count*(*\**) from user  
 *<!-- where标签相当 于where关键字，可以自动去除第一个and -->* <**where**>  
 *<!-- 引用sql片段，如果sql片段和引用处不在同一个mapper必须前边加namespace -->* <**include refid="query\_user\_where"** ></**include**>  
   
 *<!-- 下边还有很其它的条件 -->  
 <!-- <include refid="其它的sql片段"></include> -->* </**where**>  
 </**select**>

## foreach

在statement通过foreach遍历parameterType中的集合类型。

需求：

根据多个用户id查询用户信息。

### 在userQueryVo中定义list<Integer> ids

在userQueryvo中定义list<Integer> ids存储多个id

*/\*\*  
 \* pojo包装对象，将来在使用时从页面传到Controller、Service、Mapper<br>  
 \* create:2016-10-17 8:32  
 \*  
 \** ***@author*** *李欣  
 \** ***@version*** *1.0  
 \*/***public class** UserQueryVo {  
  
 *// 用户信息* **private** User **user**;  
  
 *// 自定义User扩展类型* **private** UserCustom **userCustom**;  
  
 *// id集合* **private** Integer **ids**;  
  
}

### 修改where语句

使用foreach遍历list：

<**sql id="query\_user\_where"**>  
 *<!-- 如果 userQueryVo中传入查询条件，再进行sql拼接-->  
 <!-- test中userCustom.username表示从userQueryVo读取属性值-->* <**if test="userCustom!=null"**>  
 <**if test="userCustom.username!=null and userCustom.username!=''"**>  
 and username like '%${userCustom.username}%'  
 </**if**>  
 <**if test="userCustom.sex!=null and userCustom.sex!=''"**>  
 and sex = #{userCustom.sex}  
 </**if**>  
  
 *<!-- 根据id集合查询用户信息 -->  
 <!-- 最终拼接的效果：  
 SELECT id ,username ,birthday FROM USER WHERE username LIKE '%小明%' AND id IN (16,22,25)  
 collection： 集合的属性  
 open： 开始循环拼接的串  
 close： 结束循环拼接的串  
 item： 每次循环取到的对象  
 separator： 每两次循环中间拼接的串，用来间隔item  
 -->* <**foreach collection="ids" open=" AND id IN ( " close=")" item="id" separator=","**>  
 #{id}  
 </**foreach**>  
  
 *<!-- 最终拼接的效果：  
 SELECT id ,username ,birthday FROM USER WHERE username LIKE '%小明%' AND (id = 16 OR id = 22 OR id = 25) -->  
 <!--<foreach collection="ids" open=" AND ( " close=")" item="id" separator="OR">  
 id = #{id}  
 </foreach>-->  
 <!-- 还有很的查询条件 -->* </**if**>  
</**sql**>

### 测试代码

@Test  
**public void** findUserList() **throws** Exception {  
 SqlSession sqlSession = **sqlSessionFactory**.openSession();  
  
 *// 创建代理对象* UserMapper userMapper = sqlSession.getMapper(UserMapper.**class**);  
  
 *// 构造查询条件* UserQueryVo userQueryVo = **new** UserQueryVo();  
 UserCustom userCustom = **new** UserCustom();  
 userCustom.setUsername(**"小明"**);  
 userCustom.setSex(**"1"**);  
 userQueryVo.setUserCustom(userCustom);  
 *// id集合* ArrayList<Integer> ids = **new** ArrayList<>();  
 ids.add(16);  
 ids.add(22);  
 userQueryVo.setIds(ids);  
  
 List<User> users = userMapper.findUserList(userQueryVo);  
  
 **for** (User user : users) {  
 System.***out***.println(user);  
 }  
  
 sqlSession.close();  
}