## 1. 对原生态jdbc程序中问题总结

### 1.1 环境

java环境：jdk1.7.0\_72

eclipse：indigo

mysql：5.1

### 1.2 创建mysql数据

导入下边的脚本：



sql\_table.sql：记录表结构

sql\_data.sql：记录测试数据，在实际企业开发中，**最后提供一个初始化数据脚本**



### 1.3 jdbc程序

使用jdbc查询mysql数据库中用户表的记录。

创建java工程，加入jar包：

数据库驱动包（mysql5.1）



上边的是mysql驱动。

下边的是oracle的驱动。

程序代码：



### 1.4 问题总结

1、数据库连接，使用时就创建，不使用立即释放，对数据库进行频繁连接开启和关闭，造成数据库资源浪费，影响 数据库性能。

设想：使用数据库连接池管理数据库连接。

2、将sql语句硬编码到java代码中，如果sql 语句修改，需要重新编译java代码，不利于系统维护。

设想：将sql语句配置在xml配置文件中，即使sql变化，不需要对java代码进行重新编译。

3、向preparedStatement中设置参数，对占位符号位置和设置参数值，硬编码在java代码中，不利于系统维护。

设想：将sql语句及占位符号和参数全部配置在xml中。

4、从resutSet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。

设想：将查询的结果集，自动映射成java对象。

## 2. Mybatis框架基本配置

### 2.1 mybatis是什么

mybatis是一个持久层的框架，是apache下的顶级项目。

mybatis托管到goolecode下，再后来托管到github下(https://github.com/mybatis/mybatis-3/releases)。

mybatis让程序将主要精力放在sql上，通过mybatis提供的映射方式，自由灵活生成（半自动化，大部分需要程序员编写sql）满足需要sql语句。

mybatis可以将向 preparedStatement中的输入参数自动进行输入映射，将查询结果集灵活映射成java对象。（输出映射）

### 2.2 mybatis框架

SqlMapConfig.xml（是mybatis的全局配置文件，名称不固定的）

配置了数据源、事务等mybatis运行环境

配置映射文件（配置sql语句）

mapper.xml（映射文件）、mapper.xml、mapper.xml.....

SqlSessionFactory（会话工厂），根据配置文件创建工厂

作用：创建SqlSession

msyql

SqlSession（会话），是一个接口，面向用户（程序员）的接口

作用：操作数据库（发出sql增、删、改、查）

Executor（执行器），是一个接口（基本执行器、缓存执行器）

作用：SqlSession内部通过执行器操作数据库

mapped statement（底层封装对象）

作用：对操作数据库存储封装，包括 sql语句，输入参数、输出结果类型

输入参数类型

java简单类型

hashmap

pojo自定义

输出结果 类型

java简单类型

hashmap

pojo自定义

1. mybatis配置

SqlMapConfig.xml，此文件作为mybatis的全局配置文件，配置了mybatis的运行环境等信息。

mapper.xml文件即sql映射文件，文件中配置了操作数据库的sql语句。此文件需要在SqlMapConfig.xml中加载。

1. 通过mybatis环境等配置信息构造SqlSessionFactory即会话工厂
2. 由会话工厂创建sqlSession即会话，操作数据库需要通过sqlSession进行。
3. mybatis底层自定义了Executor执行器接口操作数据库，Executor接口有两个实现，一个是基本执行器、一个是缓存执行器。
4. Mapped Statement也是mybatis一个底层封装对象，它包装了mybatis配置信息及sql映射信息等。mapper.xml文件中一个sql对应一个Mapped Statement对象，sql的id即是Mapped statement的id。
5. Mapped Statement对sql执行输入参数进行定义，包括HashMap、基本类型、pojo，Executor通过Mapped Statement在执行sql前将输入的java对象映射至sql中，输入参数映射就是jdbc编程中对preparedStatement设置参数。
6. Mapped Statement对sql执行输出结果进行定义，包括HashMap、基本类型、pojo，Executor通过Mapped Statement在执行sql后将输出结果映射至java对象中，输出结果映射过程相当于jdbc编程中对结果的解析处理过程。

### 2.3 mybatis所需jar包



mybatis-3.2.7.jar----mybatis的核心包

lib----mybatis的依赖包



### 2.4 log4.propertyies

在classpath下创建log4j.properties如下：

# 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

mybatis默认使用log4j作为输出日志信息。

开发测试为**DEBUG**

软件发布改为**Info**

### 2.5 SqlMapConfig.xml

在classpath下创建SqlMapConfig.xml，如下：

<?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>

<!-- 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=*"com.mysql.jdbc.Driver"* />

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8"* />

<property name=*"username"* value=*"root"* />

<property name=*"password"* value=*"mysql"* />

</dataSource>

</environment>

</environments>

</configuration>

SqlMapConfig.xml是mybatis核心配置文件，上边文件的配置内容为数据源、事务管理。

### 2.6 映射文件配置

在classpath下的sqlmap目录下创建sql映射文件Users.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"*>

</mapper>

namespace ：命名空间，用于隔离sql语句，后面会讲另一层非常重要的作用。

### 2.7 加载映射文件

mybatis框架需要加载映射文件，将Users.xml添加在SqlMapConfig.xml，如下：

<mappers>

<mapper resource=*"sqlmap/User.xml"*/>

</mappers>

## 3. mybatis的增删查改

### 3.1 查询Sql语句:

|  |
| --- |
| <?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">  <!-- 命名空间,作用就是对sql进行分类化管理,理解sql隔离 -->  <mapper namespace=*"test"*>  <!-- id: 标识映射文件中的sql, 将sql语句封装到mappedStatement对象中,所以id被称为statement的id  parameterType: 传进来的参数类型  resultType: 定义sql语句执行后的返回类型  #{}: 与preparedStatement的占位符一样  #{id}: 其中的id表示接收输入的参数,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value或者其他名称  -->  <select id=*"findUserByUsername"* parameterType=*"String"* resultType=*"int"*>  select count(\*) from student where id=#{id}  </select>  </mapper> |

parameterType：定义输入到sql中的映射类型，#{id}表示使用preparedstatement设置占位符号并将输入变量id传到sql。

resultType：定义结果映射类型。

#### 加载映射文件:

mybatis框架需要加载映射文件，将Users.xml添加在SqlMapConfig.xml，如下：

<mappers>

<mapper resource=*"sqlmap/User.xml"*/>

</mappers>

#### Mybatis测试:

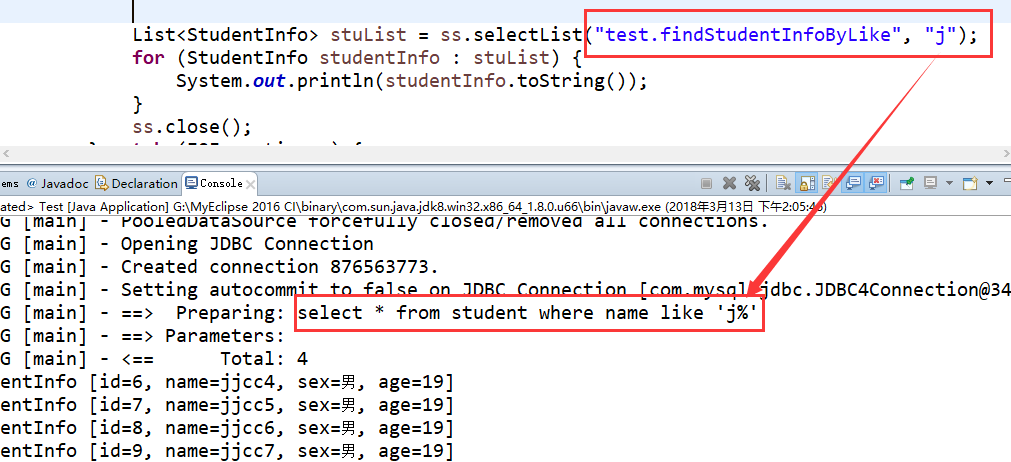
|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** IOException {  // mybatis配置文件  String resource = "SqlMapConfig.xml";    // 得到文件配置流  InputStream inputStream = Resources.*getResourceAsStream*(resource);    // 创建会话工厂,传入mybatis的配置文件信息  SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);    // 通过工厂得到SqlSession  SqlSession sqlSession = sqlSessionFactory.openSession();    //通过SqlSession操作数据库  **int** si = sqlSession.selectOne("test.findUserByUsername", "阿斯顿");  System.***out***.println(si);    // 释放资源  sqlSession.close();  } |

#### 模糊查询:

##### 映射文件配置:

|  |
| --- |
| <!-- 模糊查询 -->  <!-- ${value}: 表示拼接sql串,通过${}可以将parameterType传入的内容拼接在sql中且不进行jdbc类型转换 -->  <select id=*"findStudentInfoByLike"* parameterType=*"String"* resultType=*"com.zsl.model.bean.StudentInfo"*>  select \* from student where name like '${value}%'  </select> |

##### Java代码编写:



#### #{}和${}的区别:

#{}表示一个占位符号，通过#{}可以实现preparedStatement向占位符中设置值，自动进行java类型和jdbc类型转换，#{}可以有效防止sql注入。 #{}可以接收简单类型值或pojo属性值。 如果parameterType传输单个简单类型值，#{}括号中可以是value或其它名称。

${}表示拼接sql串，通过${}可以将parameterType 传入的内容拼接在sql中且不进行jdbc类型转换， ${}可以接收简单类型值或pojo属性值，如果parameterType传输单个简单类型值，${}括号中只能是value。

#### parameterType和resultType:

parameterType：指定输入参数类型，mybatis通过ognl从输入对象中获取参数值拼接在sql中。

resultType：指定输出结果类型，mybatis将sql查询结果的一行记录数据映射为resultType指定类型的对象。

#### selectOne和selectList:

selectOne查询一条记录，如果使用selectOne查询多条记录则抛出异常：

org.apache.ibatis.exceptions.TooManyResultsException: Expected one result (or null) to be returned by selectOne(), but found: 3

at org.apache.ibatis.session.defaults.DefaultSqlSession.selectOne(DefaultSqlSession.java:70)

selectList可以查询一条或多条记录。

### 3.2添加Sql语句

#### 映射文件:

<!-- 添加用户 -->

<insert id=*"insertUser"* parameterType=*"cn.itcast.mybatis.po.User"*>

<selectKey keyProperty=*"id"* order=*"AFTER"* resultType=*"java.lang.Integer"*>

select LAST\_INSERT\_ID()

</selectKey>

insert into user(username,birthday,sex,address)

values(#{username},#{birthday},#{sex},#{address})

</insert>

#### 测试程序:

// 添加用户信息

@Test

**public** **void** testInsert() {

// 数据库会话实例

SqlSession sqlSession = **null**;

**try** {

// 创建数据库会话实例sqlSession

sqlSession = sqlSessionFactory.openSession();

// 添加用户信息

User user = **new User()**;

user.setUsername("张小明");

user.setAddress("河南郑州");

user.setSex("1");

user.setPrice(1999.9f);

sqlSession.insert("test.insertUser", user);

//提交事务

sqlSession.commit();

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

**if** (sqlSession != **null**) {

sqlSession.close();

}

}

}

#### mysql自增主键返回:

通过修改sql映射文件，可以将mysql自增主键返回：

<insert id=*"insertUser"* parameterType=*"cn.itcast.mybatis.po.User"*>

<!-- selectKey将主键返回，需要再返回 -->

<selectKey keyProperty=*"id"* order=*"AFTER"* resultType=*"java.lang.Integer"*>

select LAST\_INSERT\_ID()

</selectKey>

insert into user(username,birthday,sex,address)

values(#{username},#{birthday},#{sex},#{address});

</insert>

添加selectKey实现将主键返回

keyProperty:返回的主键存储在pojo中的哪个属性

order：selectKey的执行顺序，是相对与insert语句来说，由于mysql的自增原理执行完insert语句之后才将主键生成，所以这里selectKey的执行顺序为after

resultType:返回的主键是什么类型

LAST\_INSERT\_ID():是mysql的函数，返回auto\_increment自增列新记录id值。

#### Mysql使用 uuid实现主键:

需要增加通过select uuid()得到uuid值

<insert id="insertUser" parameterType="cn.itcast.mybatis.po.User">

<selectKey resultType="java.lang.String" order="BEFORE"

keyProperty="id">

select uuid()

</selectKey>

insert into user(id,username,birthday,sex,address)

values(#{id},#{username},#{birthday},#{sex},#{address})

</insert>

注意这里使用的order是“BEFORE”

#### Oracle使用序列化生成主键:

首先自定义一个序列且用于生成主键，selectKey使用如下：

<insert id="insertUser" parameterType="cn.itcast.mybatis.po.User">

<selectKey resultType="java.lang.Integer" order="BEFORE" keyProperty="id">

SELECT 自定义序列.NEXTVAL FROM DUAL

</selectKey>

insert into user(id,username,birthday,sex,address)

values(#{id},#{username},#{birthday},#{sex},#{address})

</insert>

注意这里使用的order是“BEFORE”

### 3.3. 删除Sql语句

#### 映射文件:

<!-- 删除用户 -->

<delete id=*"deleteUserById"* parameterType=*"int"*>

delete from user where id=#{id}

</delete>

#### 测试程序:

// 根据id删除用户

@Test

**public** **void** testDelete() {

// 数据库会话实例

SqlSession sqlSession = **null**;

**try** {

// 创建数据库会话实例sqlSession

sqlSession = sqlSessionFactory.openSession();

// 删除用户

sqlSession.delete("test.deleteUserById",18);

// 提交事务

sqlSession.commit();

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

**if** (sqlSession != **null**) {

sqlSession.close();

}

}

}

### 3.4. 修改Sql语句:

#### 映射文件:

<!-- 更新用户 -->

<update id=*"updateUser"* parameterType=*"cn.itcast.mybatis.po.User"*>

update user set username=#{username},birthday=#{birthday},sex=#{sex},address=#{address}

where id=#{id}

</update>

#### 测试程序:

// 更新用户信息

@Test

**public** **void** testUpdate() {

// 数据库会话实例

SqlSession sqlSession = **null**;

**try** {

// 创建数据库会话实例sqlSession

sqlSession = sqlSessionFactory.openSession();

// 添加用户信息

User user = **new User()**;

user.setId(16);

user.setUsername("张小明");

user.setAddress("河南郑州");

user.setSex("1");

user.setPrice(1999.9f);

sqlSession.update("test.updateUser", user);

// 提交事务

sqlSession.commit();

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

**if** (sqlSession != **null**) {

sqlSession.close();

}

}

}

## 4. Dao开发

使用mybatis开发Dao, 通常有两个方法,即原始Dao开发方法和Mapper接口开发方法.

### 4.1 SqlSession的使用范围

SqlSession中封装了对数据库的操作，如：查询、插入、更新、删除等。

通过SqlSessionFactory创建SqlSession，而SqlSessionFactory是通过SqlSessionFactoryBuilder进行创建。

#### 4.1.1 SqlSessionFactoryBuilder

SqlSessionFactoryBuilder用于创建SqlSessionFactory，SqlSessionFactory一旦创建完成就不需要SqlSessionFactoryBuilder了，因为SqlSession是通过SqlSessionFactory生产，所以可以将SqlSessionFactoryBuilder当成一个工具类使用，最佳使用范围是方法范围即方法体内局部变量。

#### 4.1.2 SqlSessionFactory

SqlSessionFactory是一个接口，接口中定义了openSession的不同重载方法，SqlSessionFactory的最佳使用范围是整个应用运行期间，一旦创建后可以重复使用，通常以单例模式管理SqlSessionFactory。

#### 4.1.3 SqlSession

SqlSession是一个面向用户的接口， sqlSession中定义了数据库操作，默认使用DefaultSqlSession实现类。

执行过程如下：

1. 加载数据源等配置信息

Environment environment = configuration.getEnvironment();

1. 创建数据库链接
2. 创建事务对象
3. 创建Executor，SqlSession所有操作都是通过Executor完成，mybatis源码如下：

**if** (ExecutorType.*BATCH* == executorType) {

executor = **new**BatchExecutor(**this**, transaction);

} **elseif** (ExecutorType.*REUSE* == executorType) {

executor = **new** ReuseExecutor(**this**, transaction);

} **else** {

executor = **new** SimpleExecutor(**this**, transaction);

}

**if** (cacheEnabled) {

executor = **new** CachingExecutor(executor, autoCommit);

}

1. SqlSession的实现类即DefaultSqlSession，此对象中对操作数据库实质上用的是Executor

结论：

每个线程都应该有它自己的SqlSession实例。SqlSession的实例不能共享使用，它也是线程不安全的。因此最佳的范围是请求或方法范围。绝对不能将SqlSession实例的引用放在一个类的静态字段或实例字段中。

打开一个 SqlSession；使用完毕就要关闭它。通常把这个关闭操作放到 finally 块中以确保每次都能执行关闭。如下：

SqlSession session = sqlSessionFactory.openSession();

try {

// do work

} finally {

session.close();

}

### 4.2 原始Dao开发方式

原始Dao开发方法需要程序员编写Dao接口和Dao实现类。

#### 4.2.1 映射文件

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace=*"test"*>

<!-- 根据id获取用户信息 -->

<select id=*"findUserById"* parameterType=*"int"* resultType=*"cn.itcast.mybatis.po.User"*>

select \* from user where id = #{id}

</select>

<!-- 添加用户 -->

<insert id=*"insertUser"* parameterType=*"cn.itcast.mybatis.po.User"*>

<selectKey keyProperty=*"id"* order=*"AFTER"* resultType=*"java.lang.Integer"*>

select LAST\_INSERT\_ID()

</selectKey>

insert into user(username,birthday,sex,address)

values(#{username},#{birthday},#{sex},#{address})

</insert>

</mapper>

#### 4.2.2 Dao接口

**Public interface** UserDao {

**public** User getUserById(**int** id) **throws** Exception;

**public void** insertUser(User user) **throws** Exception;

}

**Public class** UserDaoImpl **implements** UserDao {

//注入SqlSessionFactory

**public** UserDaoImpl(SqlSessionFactory sqlSessionFactory){

**this**.setSqlSessionFactory(sqlSessionFactory);

}

**private** SqlSessionFactory sqlSessionFactory;

@Override

**public User** getUserById(**int** id) **throws** Exception {

SqlSession session = sqlSessionFactory.openSession();

User user = **null**;

**try** {

//通过sqlsession调用selectOne方法获取一条结果集

//参数1：指定定义的statement的id,参数2：指定向statement中传递的参数

user = session.selectOne("test.findUserById", 1);

System.out.println(user);

} **finally**{

session.close();

}

**return** user;

}

@Override

**Public void** insertUser(User user) **throws** Exception {

SqlSession sqlSession = sqlSessionFactory.openSession();

**try** {

sqlSession.insert("insertUser", user);

sqlSession.commit();

} **finally**{

session.close();

}

}

}

### 4.3 Mapper动态代理方式

#### 4.3.1 实现原理

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

Mapper接口开发需要遵循以下规范：

1. Mapper.xml文件中的namespace与mapper接口的类路径相同。
2. Mapper接口方法名和Mapper.xml中定义的每个statement的id相同
3. Mapper接口方法的输入参数类型和mapper.xml中定义的每个sql 的parameterType的类型相同
4. Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的resultType的类型相同

#### 4.3.2 Mapper.xml(映射文件)

定义mapper映射文件UserMapper.xml（内容同Users.xml），需要修改namespace的值为 UserMapper接口路径。将UserMapper.xml放在classpath 下mapper目录 下。

<?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=*"cn.itcast.mybatis.mapper.UserMapper"*>

<!-- 根据id获取用户信息 -->

<select id=*"findUserById"* parameterType=*"int"* resultType=*"cn.itcast.mybatis.po.User"*>

select \* from user where id = #{id}

</select>

<!-- 自定义条件查询用户列表 -->

<select id=*"findUserByUsername"* parameterType=*"java.lang.String"*

resultType=*"cn.itcast.mybatis.po.User"*>

select \* from user where username like '%${value}%'

</select>

<!-- 添加用户 -->

<insert id=*"insertUser"* parameterType=*"cn.itcast.mybatis.po.User"*>

<selectKey keyProperty=*"id"* order=*"AFTER"* resultType=*"java.lang.Integer"*>

select LAST\_INSERT\_ID()

</selectKey>

insert into user(username,birthday,sex,address)

values(#{username},#{birthday},#{sex},#{address})

</insert>

</mapper>

#### 4.3.3 Mapper.java(java文件)

/\*\*

\* 用户管理mapper

\*/

**Public interface** UserMapper {

//根据用户id查询用户信息

**public** User findUserById(**int** id) **throws** Exception;

//查询用户列表

**public** List<User> findUserByUsername(String username) **throws** Exception;

//添加用户信息

**public** **void** insertUser(User user)**throws** Exception;

}

接口定义有如下特点：

1. Mapper接口方法名和Mapper.xml中定义的statement的id相同
2. Mapper接口方法的输入参数类型和mapper.xml中定义的statement的parameterType的类型相同
3. Mapper接口方法的输出参数类型和mapper.xml中定义的statement的resultType的类型相同

#### 4.3.4加载UserMapper.xml文件

修改SqlMapConfig.xml文件：

<!-- 加载映射文件 -->

<mappers>

<mapper resource=*"mapper/UserMapper.xml"*/>

</mappers>

#### 4.3.5测试

**Public class** UserMapperTest **extends** TestCase {

**private** SqlSessionFactory sqlSessionFactory;

**protected void** setUp() **throws** Exception {

//mybatis配置文件

String resource = "sqlMapConfig.xml";

InputStream inputStream = Resources.*getResourceAsStream*(resource);

//使用SqlSessionFactoryBuilder创建sessionFactory

sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

}

**Public void** testFindUserById() **throws** Exception {

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获取mapper接口的代理对象

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//调用代理对象方法

User user = userMapper.findUserById(1);

System.out.println(user);

//关闭session

session.close();

}

@Test

**public** **void** testFindUserByUsername() **throws** Exception {

SqlSession sqlSession = sqlSessionFactory.openSession();

UserMapper userMapper = sqlSession.getMapper(UserMapper.**class**);

List<User> list = userMapper.findUserByUsername("张");

System.*out*.println(list.size());

}

**Public void** testInsertUser() **throws** Exception {

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获取mapper接口的代理对象

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//要添加的数据

User user = **new User()**;

user.setUsername("张三");

user.setBirthday(**new** Date());

user.setSex("1");

user.setAddress("北京市");

//通过mapper接口添加用户

userMapper.insertUser(user);

//提交

session.commit();

//关闭session

session.close();

}

}

#### 4.3.6 总结

* selectOne和selectList

动态代理对象调用sqlSession.selectOne()和sqlSession.selectList()是根据mapper接口方法的返回值决定，如果返回list则调用selectList方法，如果返回单个对象则调用selectOne方法。

* namespace

mybatis官方推荐使用mapper代理方法开发mapper接口，程序员不用编写mapper接口实现类，使用mapper代理方法时，输入参数可以使用pojo包装对象或map对象，保证dao的通用性。

## 5. SqlMapConfig.xml

mybatis的全局配置文件SqlMapConfig.xml，配置内容如下：

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）

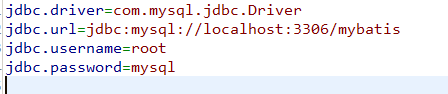
### 5.1 propertyies标签

需求：

将数据库连接参数单独配置在db.properties中，只需要在SqlMapConfig.xml中加载db.properties的属性值。

在SqlMapConfig.xml中就不需要对数据库连接参数硬编码。

将数据库连接参数只配置在db.properties中，原因：方便对参数进行统一管理，其它xml可以引用该db.properties。



在sqlMapConfig.xml加载属性文件：



properties特性：

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：

不要在properties元素体内添加任何属性值，只将属性值定义在properties文件中。

在properties文件中定义属性名要有一定的特殊性，如：XXXXX.XXXXX.XXXX

### 5.2 settings全局参数配置

mybatis框架在运行时可以调整一些运行参数。

比如：开启二级缓存、开启延迟加载。。

全局参数将会影响mybatis的运行行为。







### 5.3 typeAliases(别名)

#### 5.3.1需求:

在mapper.xml中，定义很多的statement，statement需要parameterType指定输入参数的类型、需要resultType指定输出结果的映射类型。

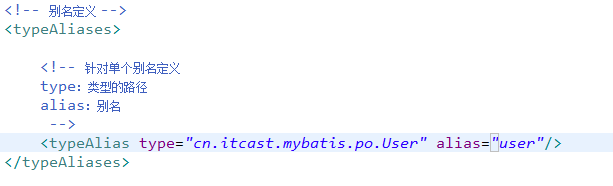
如果在指定类型时输入类型全路径，不方便进行开发，可以针对parameterType或resultType指定的类型定义一些别名，在mapper.xml中通过别名定义，方便开发。

#### 5.3.2 默认支持的别名 :

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

#### 5.3.3 自定义别名:

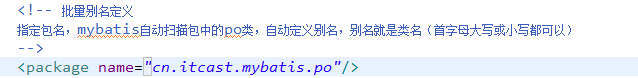
##### 单个别名定义:



引用别名：



##### 批量定义别名(常用):



### 5.4 typeHandler(类型处理器)

mybatis中通过typeHandlers完成jdbc类型和java类型的转换。

通常情况下，mybatis提供的类型处理器满足日常需要，不需要自定义.

mybatis支持类型处理器：

|  |  |  |
| --- | --- | --- |
| 类型处理器 | **Java**类型 | **JDBC**类型 |
| BooleanTypeHandler | Boolean，boolean | 任何兼容的布尔值 |
| ByteTypeHandler | Byte，byte | 任何兼容的数字或字节类型 |
| ShortTypeHandler | Short，short | 任何兼容的数字或短整型 |
| IntegerTypeHandler | Integer，int | 任何兼容的数字和整型 |
| LongTypeHandler | Long，long | 任何兼容的数字或长整型 |
| FloatTypeHandler | Float，float | 任何兼容的数字或单精度浮点型 |
| DoubleTypeHandler | Double，double | 任何兼容的数字或双精度浮点型 |
| BigDecimalTypeHandler | BigDecimal | 任何兼容的数字或十进制小数类型 |
| StringTypeHandler | String | CHAR和VARCHAR类型 |
| ClobTypeHandler | String | CLOB和LONGVARCHAR类型 |
| NStringTypeHandler | String | NVARCHAR和NCHAR类型 |
| NClobTypeHandler | String | NCLOB类型 |
| ByteArrayTypeHandler | byte[] | 任何兼容的字节流类型 |
| BlobTypeHandler | byte[] | BLOB和LONGVARBINARY类型 |
| DateTypeHandler | Date（java.util） | TIMESTAMP类型 |
| DateOnlyTypeHandler | Date（java.util） | DATE类型 |
| TimeOnlyTypeHandler | Date（java.util） | TIME类型 |
| SqlTimestampTypeHandler | Timestamp（java.sql） | TIMESTAMP类型 |
| SqlDateTypeHandler | Date（java.sql） | DATE类型 |
| SqlTimeTypeHandler | Time（java.sql） | TIME类型 |
| ObjectTypeHandler | 任意 | 其他或未指定类型 |
| EnumTypeHandler | Enumeration类型 | VARCHAR-任何兼容的字符串类型，作为代码存储（而不是索引）。 |

### 5.5 mappers（映射配置）

#### 5.5.1 通过resource加载单个映射文件



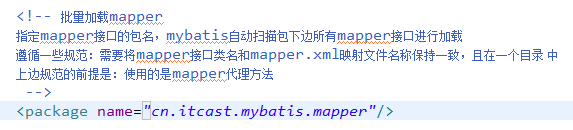
#### 5.5.2 通过mapper接口加载单个映射文件



按照上边的规范，将mapper.java和mapper.xml放在一个目录 ，且同名。



#### 5.5.3 通过package批量加载映射文件



## 6. 输入映射

通过parameterType指定输入参数的类型，类型可以是简单类型、hashmap、pojo的包装类型

### 定义包装类型pojo

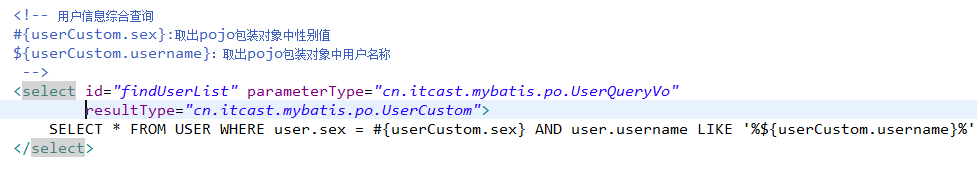
针对上边需求，建议使用自定义的包装类型的pojo。

在包装类型的pojo中将复杂的查询条件包装进去。



### mapper.xml

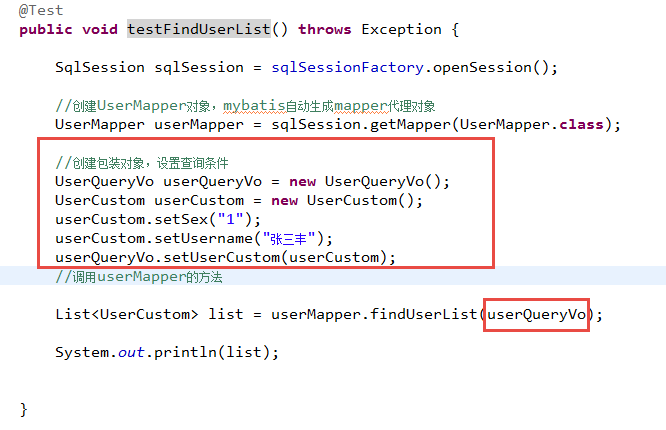
在UserMapper.xml中定义用户信息综合查询（查询条件复杂，通过高级查询进行复杂关联查询）。



### mapper.java



### 测试代码



## 7. 输出映射

### resultType

**使用resultType进行输出映射，只有查询出来的列名和pojo中的属性名一致，该列才可以映射成功。**

如果查询出来的列名和pojo中的属性名全部不一致，没有创建pojo对象。

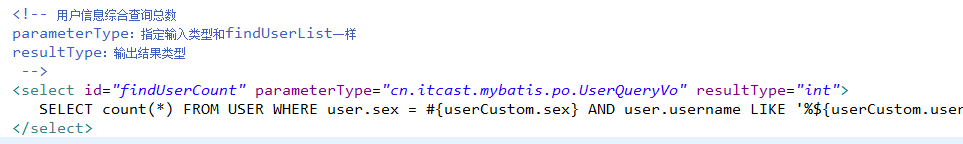
只要查询出来的列名和pojo中的属性有一个一致，就会创建pojo对象。

### 输出简单类型

#### 需求

用户信息的综合查询列表总数，通过查询总数和上边用户综合查询列表才可以实现分页。

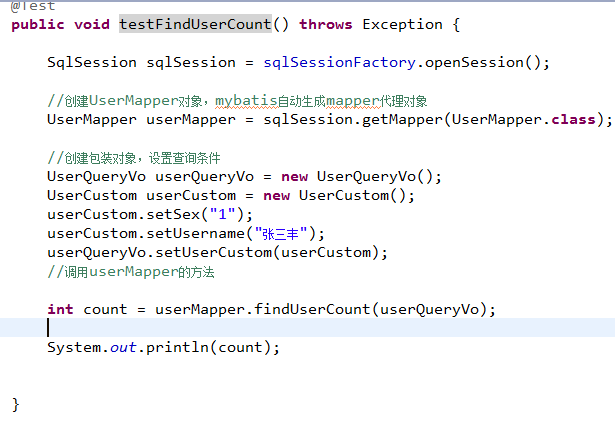
#### mapper.xml



#### mapper.java



#### 测试代码



#### 小结

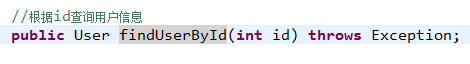
查询出来的结果集只有一行且一列，可以使用简单类型进行输出映射。

### 输出pojo对象和pojo列表

不管是输出的pojo单个对象还是一个列表（list中包括pojo），在mapper.xml中resultType指定的类型是一样的。

在mapper.java指定的方法返回值类型不一样：

1、输出单个pojo对象，方法返回值是单个对象类型



2、输出pojo对象list，方法返回值是List<Pojo>



生成的动态代理对象中是根据mapper方法的返回值类型确定是调用selectOne(返回单个对象调用)还是selectList （返回集合对象调用 ）.

### resultMap

mybatis中使用resultMap完成高级输出结果映射。

### resultMap使用方法

如果查询出来的列名和pojo的属性名不一致，通过定义一个resultMap对列名和pojo属性名之间作一个映射关系。

1、定义resultMap

2、使用resultMap作为statement的输出映射类型

### 将下边的sql使用User完成映射

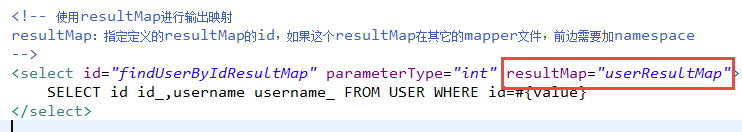
SELECT id id\_,username username\_ FROM USER WHERE id=#{value}

User类中属性名和上边查询列名不一致。

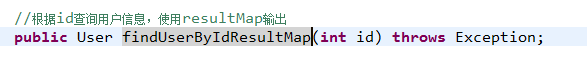
#### 定义reusltMap



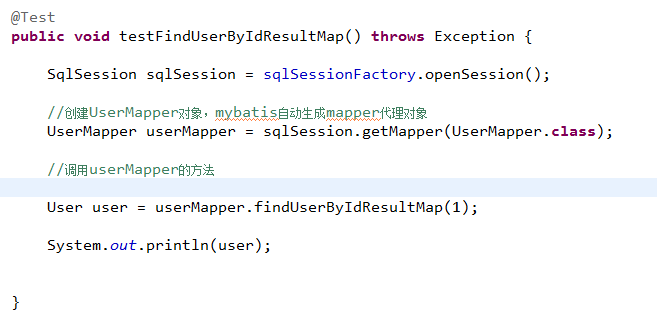
#### 使用resultMap作为statement的输出映射类型



#### mapper.java



#### 测试



### 小结

**使用resultType进行输出映射，只有查询出来的列名和pojo中的属性名一致，该列才可以映射成功。**

**如果查询出来的列名和pojo的属性名不一致，通过定义一个resultMap对列名和pojo属性名之间作一个映射关系。**

## 8. 动态sql

### 8.1 什么是动态sql

mybatis核心 对sql语句进行灵活操作，通过表达式进行判断，对sql进行灵活拼接、组装。

### 8.2 if表达式

#### 8.2.1 定义sql片段





##### 传输进来的是简单数据类型或者String

mybatis在<if>中传入一个简单类型参数的两种写法。  
方法一:  使用 注释

**[java]** [view plain](http://blog.csdn.net/u010003835/article/details/51155623" \t "_blank" \o "view plain) [copy](http://blog.csdn.net/u010003835/article/details/51155623" \t "_blank" \o "copy)

1. **public** **long** getAllCustomerQuantity(@Param("mobileFlag") **boolean** mobileFlag);

Mapper.xml

**[html]** [view plain](http://blog.csdn.net/u010003835/article/details/51155623" \t "_blank" \o "view plain) [copy](http://blog.csdn.net/u010003835/article/details/51155623" \t "_blank" \o "copy)

1. **<select** id="getAllCustomerQuantity" resultType="java.lang.Long" parameterType="boolean"**>**
2. SELECT COUNT(\*) FROM CUSTOMER AS cutomer WHERE cutomer.USE\_FLAG = 1;
3. **<if** test="mobileFlag == true"**>**
4. AND cutomer.STATUS\_CODE='2'
5. **</if>**
6. **</select>**

方法二 :  使用  \_parameter:

Mapper.java

**[java]** [view plain](http://blog.csdn.net/u010003835/article/details/51155623" \t "_blank" \o "view plain) [copy](http://blog.csdn.net/u010003835/article/details/51155623" \t "_blank" \o "copy)

1. **public** **long** getAllCustomerQuantity(**boolean** mobileFlag);

Mapper.xml

**[html]** [view plain](http://blog.csdn.net/u010003835/article/details/51155623" \t "_blank" \o "view plain) [copy](http://blog.csdn.net/u010003835/article/details/51155623" \t "_blank" \o "copy)

1. **<select** id="getAllCustomerQuantity" resultType="java.lang.Long" parameterType="boolean"**>**
2. SELECT COUNT(\*) FROM CUSTOMER AS cutomer WHERE cutomer.USE\_FLAG = 1;
3. **<if** test="\_parameter == true"**>**
4. AND cutomer.STATUS\_CODE='2'
5. **</if>**
6. **</select>**

最后，如果不是在<if>标签中，也可以按下面的方式写：  
这时候mobileFlag这个参数叫什么名称都无所谓了。

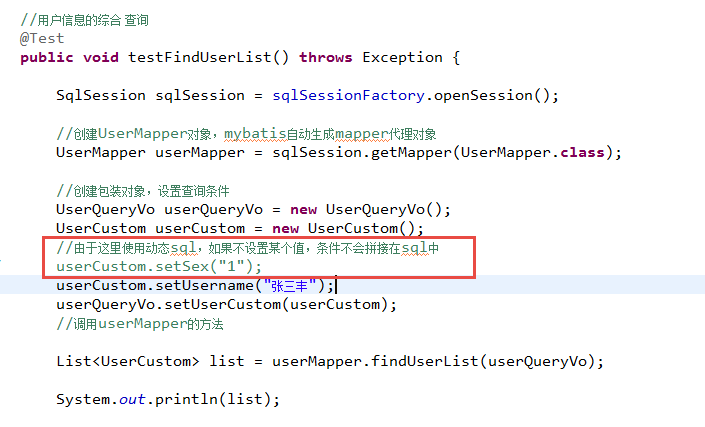
**[java]** [view plain](http://blog.csdn.net/u010003835/article/details/51155623" \t "_blank" \o "view plain) [copy](http://blog.csdn.net/u010003835/article/details/51155623" \t "_blank" \o "copy)

1. **public** **long** getAllCustomerQuantity(**boolean** mobileFlag);

**[html]** [view plain](http://blog.csdn.net/u010003835/article/details/51155623" \t "_blank" \o "view plain) [copy](http://blog.csdn.net/u010003835/article/details/51155623" \t "_blank" \o "copy)

1. **<select** id="getAllCustomerQuantity" resultType="java.lang.Long" parameterType="boolean"**>**
2. SELECT COUNT(\*) FROM CUSTOMER AS cutomer WHERE cutomer.USE\_FLAG = 1;
3. AND cutomer.STATUS\_CODE=#{mobileFlag}
4. **</select>**

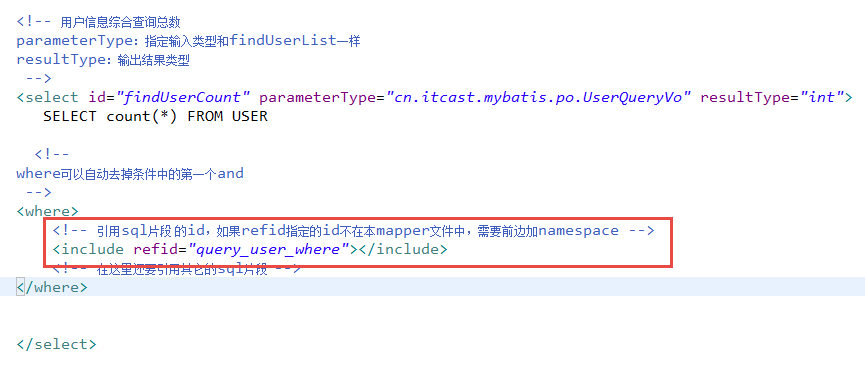
#### 8.2.2 测试代码



### 8.3引用sql片段

在mapper.xml中定义的statement中引用sql片段：

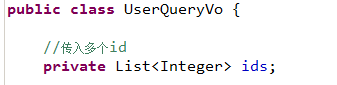




### 8.4 foreach

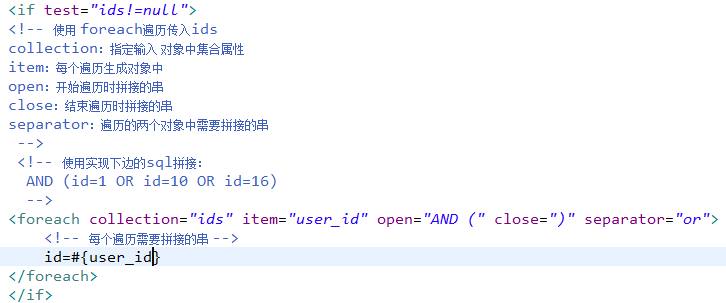
向sql传递数组或List，mybatis使用foreach解析

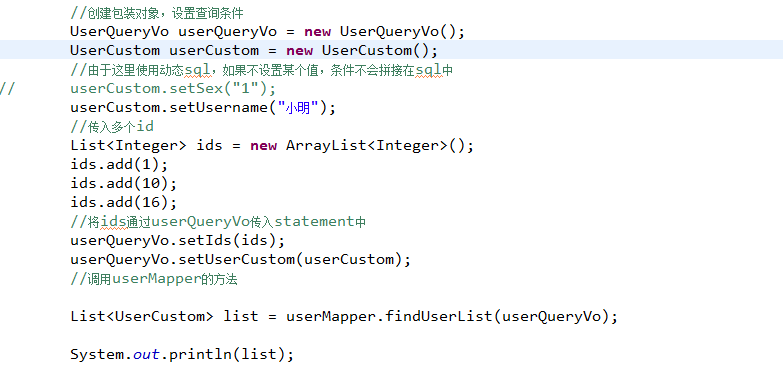
#### 8.4.1在输入参数类型中添加List<Integer> ids传入多个id

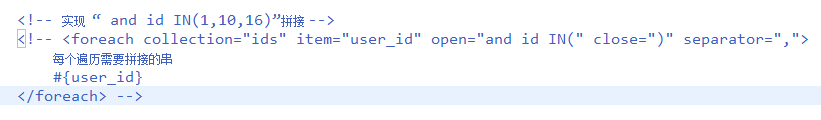


WHERE id=1 OR id=10 OR id=16

在查询条件中，查询条件定义成一个sql片段，需要修改sql片段。







#### 8.4.2 直接参数类型中添加List<Integer>

1.单参数List的类型：

1 <select id="dynamicForeachTest" parameterType="java.util.List" resultType="Blog">

2 select \* from t\_blog where id in

3 <foreach collection="list" index="index" item="item" open="(" separator="," close=")">

4 #{item}

5 </foreach>

6 </select>

上述collection的值为list，对应的Mapper是这样的  
public List dynamicForeachTest(List ids);  
测试代码：

[复制代码](javascript:void(0);)

1 @Test

2 public void dynamicForeachTest() {

3 SqlSession session = Util.getSqlSessionFactory().openSession();

4 BlogMapper blogMapper = session.getMapper(BlogMapper.class);

5 List ids = new ArrayList();

6 ids.add(1);

7 ids.add(3);

8 ids.add(6);

9 List blogs = blogMapper.dynamicForeachTest(ids);

10 for (Blog blog : blogs)

11 System.out.println(blog);

12 session.close();

13 }

[复制代码](javascript:void(0);)

2.单参数array数组的类型：

1 <select id="dynamicForeach2Test" parameterType="java.util.ArrayList" resultType="Blog">

2 select \* from t\_blog where id in

3 <foreach collection="array" index="index" item="item" open="(" separator="," close=")">

4 #{item}

5 </foreach>

6 </select>

上述collection为array，对应的Mapper代码：  
public List dynamicForeach2Test(int[] ids);  
对应的测试代码：

[复制代码](javascript:void(0);)

1 @Test

2 public void dynamicForeach2Test() {

3 SqlSession session = Util.getSqlSessionFactory().openSession();

4 BlogMapper blogMapper = session.getMapper(BlogMapper.class);

5 int[] ids = new int[] {1,3,6,9};

6 List blogs = blogMapper.dynamicForeach2Test(ids);

7 for (Blog blog : blogs)

8 System.out.println(blog);

9 session.close();

10 }

[复制代码](javascript:void(0);)

3.自己把参数封装成Map的类型

1 <select id="dynamicForeach3Test" parameterType="java.util.HashMap" resultType="Blog">

2 select \* from t\_blog where title like "%"#{title}"%" and id in

3 <foreach collection="ids" index="index" item="item" open="(" separator="," close=")">

4 #{item}

5 </foreach>

6 </select>

上述collection的值为ids，是传入的参数Map的key，对应的Mapper代码：  
public List dynamicForeach3Test(Map params);  
对应测试代码：

[复制代码](javascript:void(0);)

@Test

public void dynamicForeach3Test() {

SqlSession session = Util.getSqlSessionFactory().openSession();

BlogMapper blogMapper = session.getMapper(BlogMapper.class);

final List ids = new ArrayList();

ids.add(1);

ids.add(2);

ids.add(3);

ids.add(6);

ids.add(7);

ids.add(9);

Map params = new HashMap();

params.put("ids", ids);

params.put("title", "中国");

List blogs = blogMapper.dynamicForeach3Test(params);

for (Blog blog : blogs)

System.out.println(blog);

session.close();

}

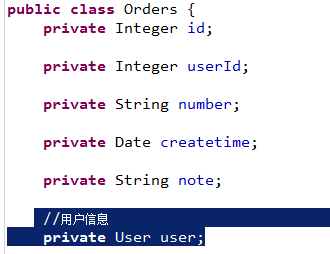
[复制代码](javascript:void(0);)

## 9. 一对一查询

### 9.1 使用resultMap映射的思路

使用resultMap将查询结果中的订单信息映射到Orders对象中，在orders类中添加User属性，将关联查询出来的用户信息映射到orders对象中的user属性中。

### 9.2 需要orders类中添加user属性



### 9.3 mapper.xml

#### 9.3.1 定义ResultMap

<!-- 订单查询关联用户的resultMap

将整个查询的结果映射到cn.itcast.mybatis.po.Orders中

-->

<resultMap type=*"cn.itcast.mybatis.po.Orders"* id=*"OrdersUserResultMap"*>

<!-- 配置映射的订单信息 -->

<!-- id：指定查询列中的唯 一标识，订单信息的中的唯 一标识，如果有多个列组成唯一标识，配置多个id

column：订单信息的唯 一标识 列

property：订单信息的唯 一标识 列所映射到Orders中哪个属性

-->

<id column=*"id"* property=*"id"*/>

<result column=*"user\_id"* property=*"userId"*/>

<result column=*"number"* property=*"number"*/>

<result column=*"createtime"* property=*"createtime"*/>

<result column=*"note"* property=*note*/>

<!-- 配置映射的关联的用户信息 -->

<!-- association：用于映射关联查询单个对象的信息

property：要将关联查询的用户信息映射到Orders中哪个属性

-->

<association property=*"user"* javaType=*"cn.itcast.mybatis.po.User"*>

<!-- id：关联查询用户的唯 一标识

column：指定唯 一标识用户信息的列

javaType：映射到user的哪个属性

-->

<id column=*"user\_id"* property=*"id"*/>

<result column=*"username"* property=*"username"*/>

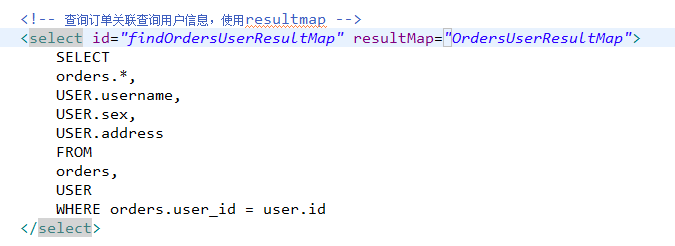
<result column=*"sex"* property=*"sex"*/>

<result column=*"address"* property=*"address"*/>

</association>

</resultMap>

#### 9.3.2 statement定义



### 9.4 mapper.java



### 9.5 resultType和resultMap实现一对一查询小结

resultType：使用resultType实现较为简单，如果pojo中没有包括查询出来的列名，需要增加列名对应的属性，即可完成映射。

如果没有查询结果的特殊要求建议使用resultType。

resultMap：需要单独定义resultMap，实现有点麻烦，如果对查询结果有特殊的要求，使用resultMap可以完成将关联查询映射pojo的属性中。

resultMap可以实现延迟加载，resultType无法实现延迟加载。

## 10. 一对多查询

### sql语句

确定主查询表：订单表

确定关联查询表：订单明细表

在一对一查询基础上添加订单明细表关联即可。

SELECT

orders.\*,

USER.username,

USER.sex,

USER.address,

orderdetail.id orderdetail\_id,

orderdetail.items\_id,

orderdetail.items\_num,

orderdetail.orders\_id

FROM

orders,

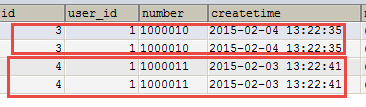
USER,

orderdetail

WHERE orders.user\_id = user.id AND orderdetail.orders\_id=orders.id

### 分析

使用resultType将上边的 查询结果映射到pojo中，订单信息的就是重复。



要求：

对orders映射不能出现重复记录。

在orders.java类中添加List<orderDetail> orderDetails属性。

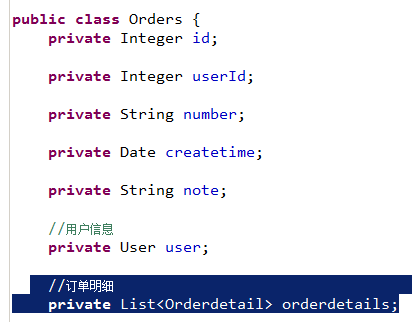
最终会将订单信息映射到orders中，订单所对应的订单明细映射到orders中的orderDetails属性中。



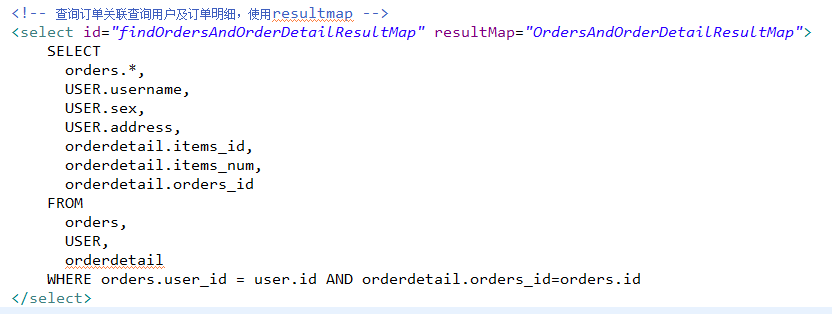
映射成的orders记录数为两条（orders信息不重复）

每个orders中的orderDetails属性存储了该 订单所对应的订单明细。

### 在orders中添加list订单明细属性



### mapper.xml



### resultMap定义

<!-- 订单及订单明细的resultMap

使用extends继承，不用在配置订单信息和用户信息的映射

-->

<resultMap type=*"cn.itcast.mybatis.po.Orders"* id=*"OrdersAndOrderDetailResultMap"* extends=*"OrdersUserResultMap"*>

<!-- 订单信息 -->

<!-- 用户信息 -->

<!-- 使用extends继承，不用在配置订单信息和用户信息的映射 -->

<!-- 订单明细信息

一个订单关联查询出了多条明细，要使用collection进行映射

collection：对关联查询到多条记录映射到集合对象中

property：将关联查询到多条记录映射到cn.itcast.mybatis.po.Orders哪个属性

ofType：指定映射到list集合属性中pojo的类型

-->

<collection property=*"orderdetails"* ofType=*"cn.itcast.mybatis.po.Orderdetail"*>

<!-- id：订单明细唯 一标识

property:要将订单明细的唯 一标识 映射到cn.itcast.mybatis.po.Orderdetail的哪个属性

-->

<id column=*"orderdetail\_id"* property=*"id"*/>

<result column=*"items\_id"* property=*"itemsId"*/>

<result column=*"items\_num"* property=*"itemsNum"*/>

<result column=*"orders\_id"* property=*"ordersId"*/>

</collection>

</resultMap>

### mapper.java



### 小结

mybatis使用resultMap的collection对关联查询的多条记录映射到一个list集合属性中。

使用resultType实现：

将订单明细映射到orders中的orderdetails中，需要自己处理，使用双重循环遍历，去掉重复记录，将订单明细放在orderdetails中。

## 12. 延迟加载

### 12.1 什么是延迟加载

resultMap可以实现高级映射（使用association、collection实现一对一及一对多映射），**association、collection具备延迟加载功能。**

需求：

如果查询订单并且关联查询用户信息。如果先查询订单信息即可满足要求，当我们需要查询用户信息时再查询用户信息。把对用户信息的按需去查询就是延迟加载。

延迟加载：先从单表查询、需要时再从关联表去关联查询，大大提高 数据库性能，因为查询单表要比关联查询多张表速度要快。

### 12.2使用association实现延迟加载

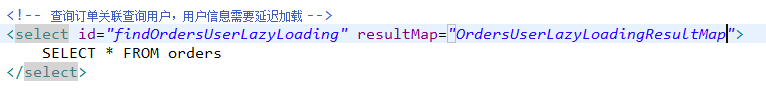
#### 12.2.1 mapper.xml

需要定义两个mapper的方法对应的statement。

1、只查询订单信息

SELECT \* FROM orders

在查询订单的statement中使用**association去延迟加载（执行）下边的satatement(**关联查询用户信息**)**



2、关联查询用户信息

通过上边查询到的订单信息中user\_id去关联查询用户信息

使用UserMapper.xml中的findUserById



上边先去执行findOrdersUserLazyLoading，当需要去查询用户的时候再去执行findUserById，通过resultMap的定义将延迟加载执行配置起来。

#### 12.2.2 延迟加载resultMap

使用association中的select指定延迟加载去执行的statement的id。

<!-- 延迟加载的resultMap -->

<resultMap type=*"cn.itcast.mybatis.po.Orders"* id=*"OrdersUserLazyLoadingResultMap"*>

<!--对订单信息进行映射配置 -->

<id column=*"id"* property=*"id"*/>

<result column=*"user\_id"* property=*"userId"*/>

<result column=*"number"* property=*"number"*/>

<result column=*"createtime"* property=*"createtime"*/>

<result column=*"note"* property=*"note"*/>

<!-- 实现对用户信息进行延迟加载

select：指定延迟加载需要执行的statement的id（是根据user\_id查询用户信息的statement）

要使用userMapper.xml中findUserById完成根据用户id(user\_id)用户信息的查询，如果findUserById不在本mapper中需要前边加namespace

column：订单信息中关联用户信息查询的列，是user\_id

关联查询的sql理解为：

SELECT orders.\*,

(SELECT username FROM USER WHERE orders.user\_id = user.id)username,

(SELECT sex FROM USER WHERE orders.user\_id = user.id)sex

FROM orders

-->

<association property=*"user"* javaType=*"cn.itcast.mybatis.po.User"*

select=*"cn.itcast.mybatis.mapper.UserMapper.findUserById"* column=*"user\_id"*>

<!-- 实现对用户信息进行延迟加载 -->

</association>

</resultMap>

#### 12.2.3 mapper.java



#### 12.2.4 测试思路

1、执行上边mapper方法（findOrdersUserLazyLoading），内部去调用cn.itcast.mybatis.mapper.OrdersMapperCustom中的findOrdersUserLazyLoading只查询orders信息（单表）。

2、在程序中去遍历上一步骤查询出的List<Orders>，当我们调用Orders中的getUser方法时，开始进行延迟加载。

3、延迟加载，去调用UserMapper.xml中findUserbyId这个方法获取用户信息。

#### 12.2.5 延迟加载配置

mybatis默认没有开启延迟加载，需要在SqlMapConfig.xml中setting配置。

在mybatis核心配置文件中配置：

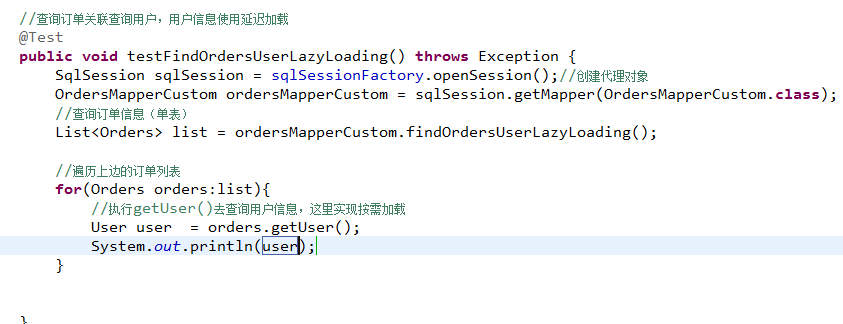
lazyLoadingEnabled、aggressiveLazyLoading

|  |  |  |  |
| --- | --- | --- | --- |
| 设置项 | 描述 | 允许值 | 默认值 |
| lazyLoadingEnabled | 全局性设置懒加载。如果设为‘false’，则所有相关联的都会被初始化加载。 | true | false | false |
| aggressiveLazyLoading | 当设置为‘true’的时候，懒加载的对象可能被任何懒属性全部加载。否则，每个属性都按需加载。 | true | false | true |

在SqlMapConfig.xml中配置：



#### 12.2.6 测试代码

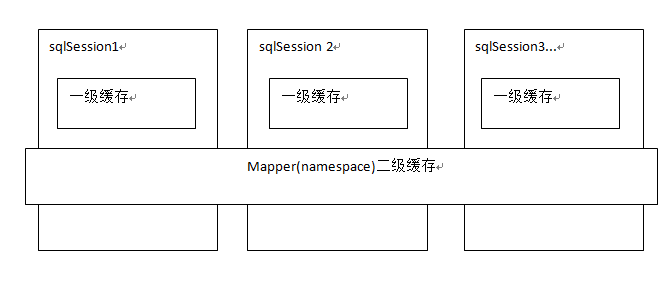


## 13. 查询缓存

### 13.1 什么是查询缓存

mybatis提供查询缓存，用于减轻数据压力，提高数据库性能。

mybaits提供一级缓存，和二级缓存。



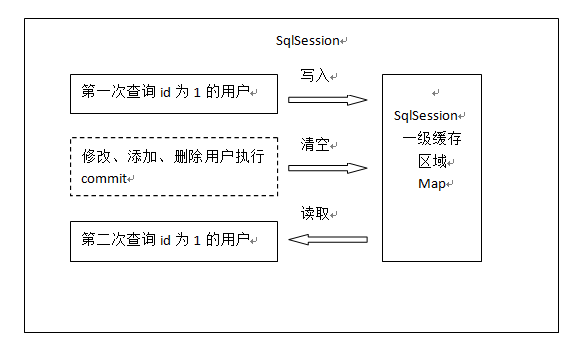
一级缓存是SqlSession级别的缓存。在操作数据库时需要构造 sqlSession对象，在对象中有一个数据结构（HashMap）用于存储缓存数据。不同的sqlSession之间的缓存数据区域（HashMap）是互相不影响的。

二级缓存是mapper级别的缓存，多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的。

为什么要用缓存？

如果缓存中有数据就不用从数据库中获取，大大提高系统性能。

### 13.2 一级缓存



第一次发起查询用户id为1的用户信息，先去找缓存中是否有id为1的用户信息，如果没有，从数据库查询用户信息。

得到用户信息，将用户信息存储到一级缓存中。

如果sqlSession去执行commit操作（执行插入、更新、删除），清空SqlSession中的一级缓存，这样做的目的为了让缓存中存储的是最新的信息，避免脏读。

第二次发起查询用户id为1的用户信息，先去找缓存中是否有id为1的用户信息，缓存中有，直接从缓存中获取用户信息。

### 13.2 一级缓存应用

正式开发，是将mybatis和spring进行整合开发，事务控制在service中。

一个service方法中包括 很多mapper方法调用。

service{

//开始执行时，开启事务，创建SqlSession对象

//第一次调用mapper的方法findUserById(1)

//第二次调用mapper的方法findUserById(1)，从一级缓存中取数据

//方法结束，sqlSession关闭

}

如果是执行两次service调用查询相同 的用户信息，不走一级缓存，因为session方法结束，sqlSession就关闭，一级缓存就清空。

## 14. 二级缓存



首先开启mybatis的二级缓存。

sqlSession1去查询用户id为1的用户信息，查询到用户信息会将查询数据存储到二级缓存中。

如果SqlSession3去执行相同 mapper下sql，执行commit提交，清空该 mapper下的二级缓存区域的数据。

sqlSession2去查询用户id为1的用户信息，去缓存中找是否存在数据，如果存在直接从缓存中取出数据。

二级缓存与一级缓存区别，二级缓存的范围更大，多个sqlSession可以共享一个UserMapper的二级缓存区域。

UserMapper有一个二级缓存区域（按namespace分） ，其它mapper也有自己的二级缓存区域（按namespace分）。

每一个namespace的mapper都有一个二缓存区域，两个mapper的namespace如果相同，这两个mapper执行sql查询到数据将存在相同 的二级缓存区域中。

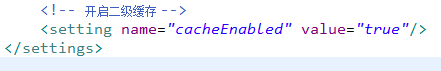
### 14.1开启二级缓存

mybaits的二级缓存是mapper范围级别，除了在SqlMapConfig.xml设置二级缓存的总开关，还要在具体的mapper.xml中开启二级缓存。

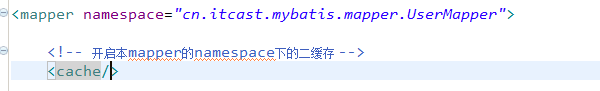
在核心配置文件SqlMapConfig.xml中加入

<setting name=*"cacheEnabled"* value=*"true"*/>

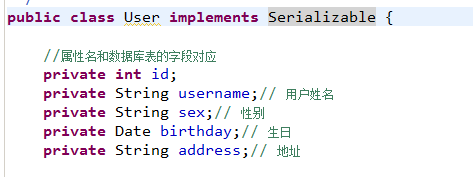
|  |  |  |  |
| --- | --- | --- | --- |
|  | 描述 | 允许值 | 默认值 |
| cacheEnabled | 对在此配置文件下的所有cache 进行全局性开/关设置。 | true false | true |



在UserMapper.xml中开启二缓存，UserMapper.xml下的sql执行完成会存储到它的缓存区域（HashMap）。



### 14.2调用pojo类实现序列化接口



为了将缓存数据取出执行反序列化操作，因为二级缓存数据存储介质多种多样，不一样在内存。

### 14.3 useCache配置

在statement中设置useCache=false可以禁用当前select语句的二级缓存，即每次查询都会发出sql去查询，默认情况是true，即该sql使用二级缓存。

<select id="findOrderListResultMap" resultMap="ordersUserMap" useCache="false">

总结：针对每次查询都需要最新的数据sql，要设置成useCache=false，禁用二级缓存。

### 14.4刷新缓存（就是清空缓存）

在mapper的同一个namespace中，如果有其它insert、update、delete操作数据后需要刷新缓存，如果不执行刷新缓存会出现脏读。

 设置statement配置中的flushCache="true" 属性，默认情况下为true即刷新缓存，如果改成false则不会刷新。使用缓存时如果手动修改数据库表中的查询数据会出现脏读。

如下：

<insert id="insertUser" parameterType="cn.itcast.mybatis.po.User" flushCache="true">

总结：一般下执行完commit操作都需要刷新缓存，flushCache=true表示刷新缓存，这样可以避免数据库脏读。

## 15. MyBatis注解Annotation介绍及Demo

MyBatis可以利用SQL映射文件来配置，也可以利用Annotation来设置。MyBatis提供的一些基本注解如下表所示。

这些注解都是运用到传统意义上映射器接口中的方法、类或者方法参数中的。

今天主要介绍两种使用注解的方式。

一种是直接在映射器接口中写SQL语句，一种是利用SqlBuilder来创建SQL再由映射器接口来调用

|  |  |  |  |
| --- | --- | --- | --- |
| 注解 | 目标 | 相应的XML | 描述 |
| @CacheNamespace | **类** | <cache> | 为给定的命名空间（比如类）配置缓存。属性：  implemetation,eviction,  flushInterval , size 和 readWrite 。 |
| @CacheNamespaceRef | **类** | <cacheRef> | 参照另外一个命名空间的缓存来使用。  属性：value，也就是类的完全限定名。 |
| @ConstructorArgs | 方法 | <constructor> | 收集一组结果传递给对象构造方法。  属性：value，是形式参数的数组 |
| @Arg | 方法 | <arg>  <idArg> | 单独的构造方法参数，是ConstructorArgs  集合的一部分。属性：id,column,javaType，typeHandler。  id属性是布尔值，来标识用于比较的属性，和<idArg>XML 元素相似 |
| @TypeDiscriminator | 方法 | <discriminator> | 一组实例值被用来决定结果映射的表  现。属性：Column, javaType ,  jdbcType typeHandler，cases。  cases属性就是实例的数组。 |
| @Case | 方法 | <case> | 单独实例的值和它对应的映射。属性：value  ，type，results 。  Results 属性是结果数组，因此这个注解和实际的ResultMap 很相似，由下面的  Results注解指定 |
| @Results | 方法 | <resultMap> | 结果映射的列表，包含了一个特别结果  列如何被映射到属性或字段的详情。  属性：value ，是Result注解的数组 |
| @Result | 方法 | <result>  <id> | 在列和属性或字段之间的单独结果映  射。属性：id ，column， property ，  javaType ，jdbcType ，type Handler ，  one，many。id属性是一个布尔值，表  示了应该被用于比较的属性。one 属性是单独的联系，和 <association> 相似，而many 属性是对集合而言的，和  <collection>相似。 |
| @One | 方法 | <association> | 复杂类型的单独属性值映射。属性：  select，已映射语句（也就是映射器方  法）的完全限定名，它可以加载合适类  型的实例。注意：联合映射在注解 API  中是不支持的。 |
| @Many | 方法 | <collection> | 复杂类型的集合属性映射。属性：  select，是映射器方法的完全限定名，它可加载合适类型的一组实例。注意：联合映射在 Java注解中是不支持的。 |
| @Options | 方法 | 映射语句的属性 | 这个注解提供访问交换和配置选项的宽广范围，它们通常在映射语句上作为属性出现。而不是将每条语句注解变复杂，Options注解提供连贯清晰的方式来访问它们。属性：useCache=true，  flushCache=false，  resultSetType=FORWARD\_ONLY，  statementType=PREPARED，  fetchSize= -1，timeout=-1 ，  useGeneratedKeys=false ，  keyProperty=”id“。  理解Java 注解是很重要的，因为没有办法来指定“null”作为值。因此，一旦你使用了 Options注解，语句就受所有默认值的支配。要注意什么样的默认值来避免不期望的行为 |
| @Insert  @Update  @Delete | 方法 | <insert>  <update>  <delete> | 这些注解中的每一个代表了执行的真实 SQL。它们每一个都使用字符串数组（或单独的字符串）。如果传递的是字符串数组，它们由每个分隔它们的单独空间串联起来。属性：value，这是字符串数组用来组成单独的SQL语句 |
| @InsertProvider  @UpdateProvider  @DeleteProvider  @SelectProvider | 方法 | <insert>  <update>  <delete>  <select>  允许创建动态  SQL。 | 这些可选的SQL注解允许你指定一个  类名和一个方法在执行时来返回运行  的SQL。基于执行的映射语句， MyBatis  会实例化这个类，然后执行由 provider  指定的方法. 这个方法可以选择性的接  受参数对象作为它的唯一参数，但是必  须只指定该参数或者没有参数。属性：  type，method。type属性是类的完全限定名。method  是该类中的那个方法名。 |
| @Param | 参数 | N/A | 当映射器方法需多个参数，这个注解可以被应用于映射器方法参数来给每个参数一个名字。否则，多参数将会以它们的顺序位置来被命名。比如  #{1}，#{2} 等，这是默认的。  使用@Param(“person”)，SQL中参数应该被命名为#{person}。 |