mybatis

# mybatis入门

## mybatis介绍

mybatis原名是ibatis，是apache的开源项目。ibatis改成mybatis不仅仅是名称的改变，mybatis在功能上也有升级。

mybatis是一个持久层的框架，mybatis框架和关系数据库进行交互，将数据持久化到关系数据中。

mybatis目前 在github（https://github.com/mybatis/mybatis-3/releases）托管。

java中通过jdbc对数据库，jdbc是一套数据库访问接口。mybatis持久层框架对jdbc的封装。

mybatis需要程序员自己编写sql，对于一些ORM框架（对象关系映射）一般是不需要程序员来编写sql。

mybatis的重点让程序员自己去灵活掌握sql，要对 sql进行优化比较容易让程序员自己掌握。

mybatis同时也对jdbc访问数据库的过程 进行了封装，对创建connection、创建statement、手动设置参数、结果集检索等进行封装。

mybatis可以通过 xml或注解（不推荐）将statement进行配置，主要完成两个重要核心 功能：

1、输入映射，将java对象映射到statement中即sql语句的输入参数。

2、输出映射，将sql语句执行的结果映射成java对象。

## 单独使用jdbc编程中问题总结

目的：

1、mybatis等持久层框架都是对jdbc的封装，为了更好的去学习mybatis，对单独使用jdbc编程中问题进行总结。

2、企业开发中要进行技术选型，对于小型 的项目来说，单独使用jdbc或对 jdbc简单的封装也在技术选型范围考虑之内。企业在技术选型主要考虑人力资源等，以节省成本及创建利益最大化为目标。

问题总结：

1、程序中存在硬编码（数据驱动加载，创建数据连接、sql语句），不利于系统维护。

设想解决硬编码：可以sql语句等信息配置在xml中。

2、数据库连接使用时打开不时立即关闭，频繁开关连接对数据库资源是一种浪费。

设想解决方案：使用数据库连接池。

3、向preparedStatement设置参数时 将参数下标 号（从1开始）硬编码在代码，并且将向占位符号设置的参数也硬编码了。

设想解决方案：可以自动通过程序将java类型对象映射到preparedStatement中。

4、从Resultset中取出结果集进行遍历，将列名硬编码

设想解决方案：可以自动通过程序将sql查询结果集映射到时java对象中，自动完成将sql查询的列值构造成一个java对象。

## mybatis框架原理

SqlMapConfig.xml(mybatis全局配置文件)

Mapper.xml（映射文件是多个）、Mapper.xml.....

SqlSessionFactory（会话工厂）

创建SqlSession会话

SqlSession（会话）

通过sqlSession接口操作数据库

Executor（底层执行器）

底层操作数据库对象

MappedStatement（mybatis底层封装对象）

封装了sql语句、输入参数类型、输出结果类型等信息

Mysql数据库

输入参数

java基本类型(int/string...)

自定义pojo

hashmap

输入映射

输出结果

java基本类型(int/string...)

自定义pojo

hashmap

输出映射

1、SqlMapConfig.xml，mybatis全局配置文件，名称不固定的。

配置了：数据源、事务等运行环境，最重要的是配置mybatis运行所需要的一些配置（Mapper.xml映射配置、全局运行参数、别名。。。）

mybatis运行先找SqlMapConfig.xml，由SqlMapConfig.xml找Mapper.xml映射配置。

Mapper.xml：mybatis的映射文件，配置了sql语句及sql 语句输入参数类型、及sql语句结果集所映射的类型

在mapper.xml中配置了statement（支持各种statement）了。

2、SqlSessionFactory，通过SqlMapconfig.xml等配置文件构造 一个会话工厂对象，要操作数据库通过SqlSessionFactory创建SqlSession会话。

作用：创建SqlSession会话

3、SqlSession，mybatis操作数据库使用的会话接口，定义了操作数据库方法（添加、删除、修改、更新）

SqlSession是一个接口，是一个面向程序员的接口。

4、Executor，是一个接口，是一个操作数据库的底层接口，其实现类有基本执行器、缓存执行器两种

SqlSession需要通过Executor操作数据库。

5、mybatis底层封装对象，封装了sql语句、输入参数类型、输出结果类型等信息

## 搭建环境

* 下载mybatis

mybaits的代码由github.com管理，地址：https://github.com/mybatis/mybatis-3/releases



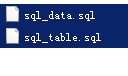
mybatis-3.2.7.jar----mybatis的核心包

lib----mybatis的依赖包

mybatis-3.2.7.pdf----mybatis使用手册

* 创建mysql数据库

在mysql中创建数据库，导入以下脚本：



sql\_data.sql：数据脚本，正常企业开在产品化需要提供初始化脚本（初始化数据）

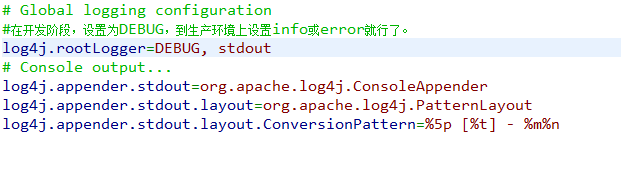
sql\_table.sql：表结构脚本，存储了表、存储过程、触发器等。



* 创建java工程：

使用eclipse创建java工程，jdk使用1.7.0\_72。

* log4j.properties



## 需求

实现以下功能：

根据用户id查询一个用户信息

根据用户名称模糊查询用户信息列表

添加用户

更新用户

删除用户

需求分析：

根据用户id查询一个用户信息：

SELECT \* FROM USER WHERE id=1

根据用户名称模糊查询用户信息列表：

SELECT \* FROM USER WHERE username LIKE '%张%'

## 根据用户id查询一个用户信息

### SqlMapConfig.xml

此方法为mybatis的全局配置文件，配置了数据源、事务等运行环境，加载配置mapper.xml映射。

使用mybatis-3-config.dtd约束xml

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

<!DOCTYPE configuration

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

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

<configuration>

<!-- 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>

<!-- 加载mapper.xml -->

<!-- <mappers></mappers> -->

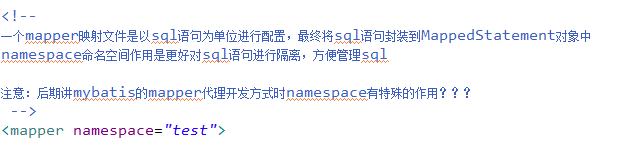
</configuration>

### mapper.xml映射文件

早期ibatis的映射文件以表名命名，后期mybatis 映射文件命名方式为：表名Mapper.xml。。

命名方式是不固定的。

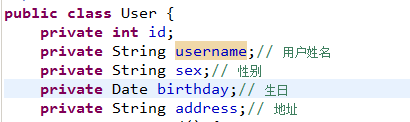
映射文件使用mybatis-3-mapper.dtd



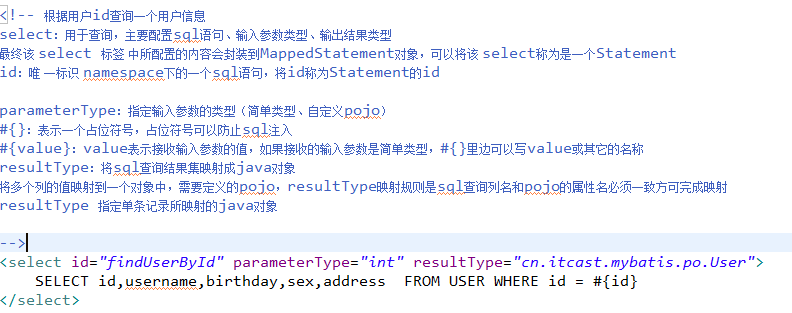
#### pojo

resultType：将sql查询结果集映射成java对象

将多个列的值映射到一个对象中，需要定义的pojo，resultType映射规则是sql查询列名和pojo的属性名必须一致方可完成映射。

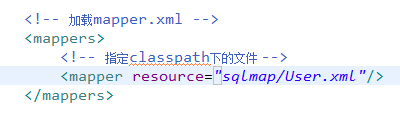


#### mapper.xml



### 在SqlMapConfig.xml加载映射文件

编辑SqlMapConfig.xml，加载User.xml映射文件



### 代码实现

@Test

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

//mybatis全局配置文件

String resource = "SqlMapConfig.xml";

//根据mybatis的全局配置文件构造 一个流

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

//创建SqlSessionFactory

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

//创建SqlSession会话

SqlSession sqlSession = sqlSessionFactory.openSession();

//使用sqlSession查询用户

//第一个参数：statement的id，前边要加namespace

//第二个参数：输入的参数对象值

//selectOne用于返回单个对象，如果sql查询返回一个列表（多个对象），如果使用selectOne报错

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

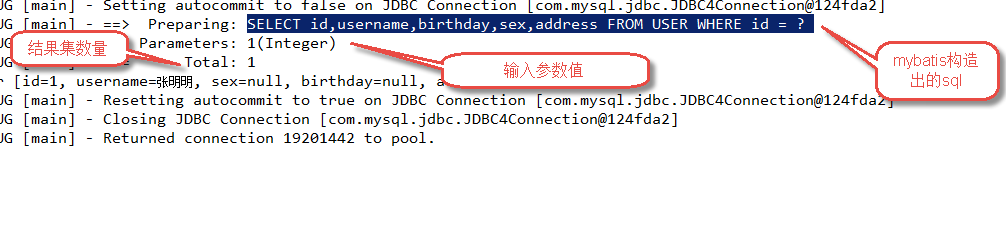
System.*out*.println(user);

//关闭sqlsession

sqlSession.close();

}

### 调试

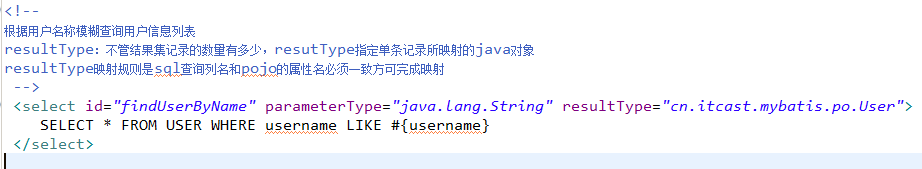


## 根据用户名称模糊查询用户信息列表

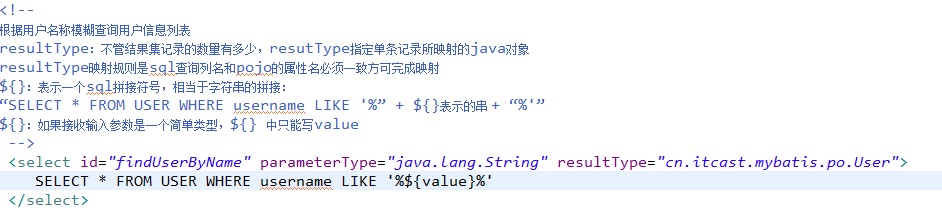
### SqlMapConfig.xml

SqlMapConfig.xml是mybatis的全局配置文件，一个系统编写一个即可。

### mapper.xml映射文件



使用${}拼接符号：



在实际开发中建议使用#{}占位符方式。

### 代码实现

// 根据用户名称模糊查询用户信息

@Test

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

// mybatis全局配置文件

String resource = "SqlMapConfig.xml";

// 根据mybatis的全局配置文件构造 一个流

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

// 创建SqlSessionFactory

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder()

.build(inputStream);

// 创建SqlSession会话

SqlSession sqlSession = sqlSessionFactory.openSession();

//根据用户名称模糊查询

//selectList用于 将sql查询结果映射成java对象 ，返回一个List<Object>

//1参数：statement的id（等于namespace+statement的id）

//2参数：输入参数值

List<User> list = sqlSession.selectList("test.findUserByName", "张");

System.*out*.println(list);

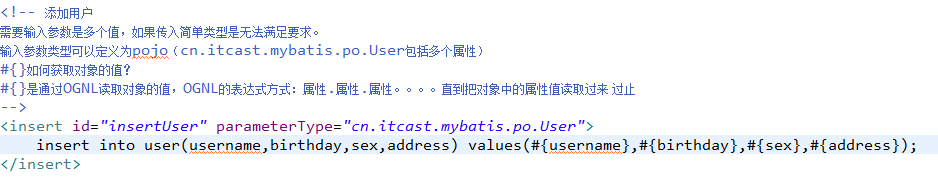
// 关闭sqlsession

sqlSession.close();

}

## 添加用户

### mapper.xml



### 代码实现

// 插入用户

@Test

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

// mybatis全局配置文件

String resource = "SqlMapConfig.xml";

// 根据mybatis的全局配置文件构造 一个流

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

// 创建SqlSessionFactory

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder()

.build(inputStream);

// 创建SqlSession会话

SqlSession sqlSession = sqlSessionFactory.openSession();

//构造 输入参数,就是插入的用户信息

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

user.setUsername("赵六");

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

user.setAddress("北京");

//调用sqlSession插入用户

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

//提交

sqlSession.commit();

// 关闭sqlsession

sqlSession.close();

}

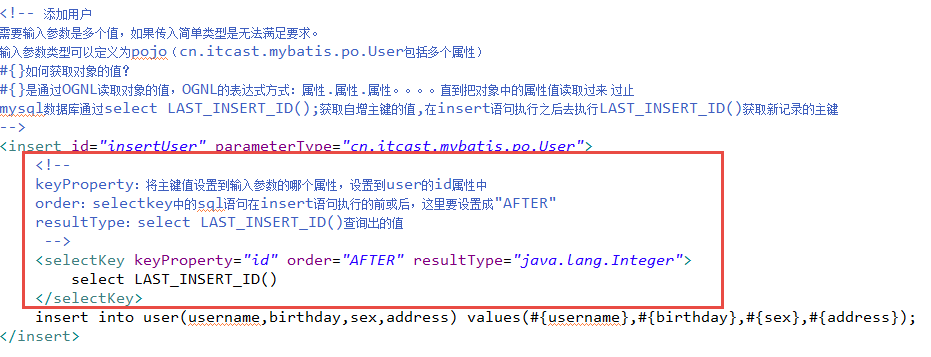
## 添加用户主键返回

### 需求

当新记录插入之后需要取出新记录的主键，如果主键是由程序代码生成的获取主键比较方便，如果主键是数据库自动生成的，需要通过程序代码获取新记录的主键。

mybatis要实现执行insert之后将主键通过输入参数对象返回。

### 数据库自增主键的返回



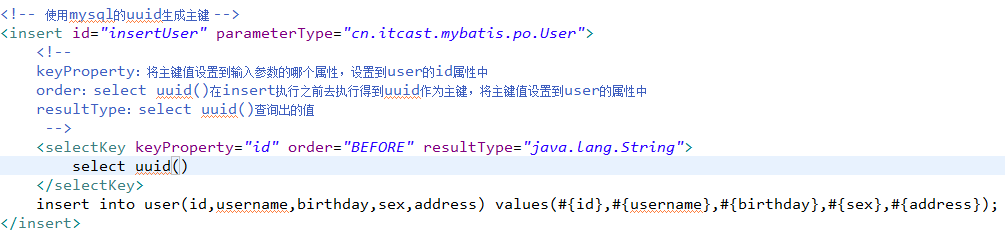
### uuid()

除了自增主键生成方式之外，还可以生成uuid的唯一序列号。

使用uuid好处：有利于数据库迁移，扩展。

缺点：对uuid数据作索引性能没有对数字型索引高。

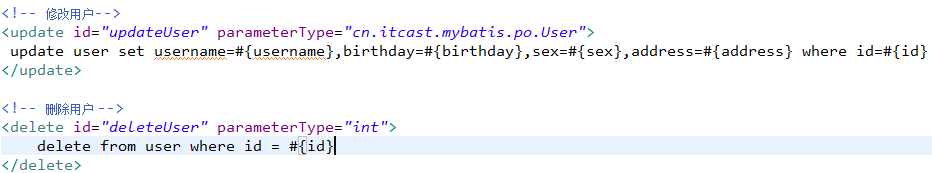
主键返回流程：



### 思考oracle的序列作主键如何返回？

## 用户修改和删除

### mapper.xml



### 代码

// 更新用户

@Test

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

// mybatis全局配置文件

String resource = "SqlMapConfig.xml";

// 根据mybatis的全局配置文件构造 一个流

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

// 创建SqlSessionFactory

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder()

.build(inputStream);

// 创建SqlSession会话

SqlSession sqlSession = sqlSessionFactory.openSession();

// 构造 输入参数,就是更新的用户信息

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

// 更新的用户信息中必须要有id

user.setId(62);

user.setUsername("赵六3");

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

user.setAddress("郑州");

// 调用sqlSession插入用户

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

// 提交

sqlSession.commit();

// 关闭sqlsession

sqlSession.close();

}

// 删除用户

@Test

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

// mybatis全局配置文件

String resource = "SqlMapConfig.xml";

// 根据mybatis的全局配置文件构造 一个流

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

// 创建SqlSessionFactory

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder()

.build(inputStream);

// 创建SqlSession会话

SqlSession sqlSession = sqlSessionFactory.openSession();

sqlSession.delete("test.deleteUser", 60);

// 提交

sqlSession.commit();

// 关闭sqlsession

sqlSession.close();

}

## 总结：

### #{}和${}

#{}：表示一个占位符号，很好去避免sql注入

原理：将占位符位置的整个参数和sql语句两部分提交给数据库，数据库去执行sql语句，去表中匹配所有的记录是否和整个参数是否一致。

#{}要获取输入参数的值：

如果输入参数是简单类型，#{}中可以写value或其它名称

如果是pojo对象类型，#{]通过OGNL方式去获取，表达式就是：属性.属性.属性。。。方式

${}表示一个sql 拼接符号

原理：是在向数据库发出sql之前去拼接好sql再提交给数据库执行。

${}要获取输入参数的值：

如果输入参数是简单类型，${}中只能写value

如果是pojo对象类型，${]通过OGNL方式去获取，表达式就是：属性.属性.属性。。。方式

一般情况下建议使用#{}，特殊情况下必须要用${}，比如：

动态拼接sql中动态组成排序字段， 要通过${}将排序字段传入sql中。

动态拼接sql中动态组成表名，要通过${}将表名传入sql中。

### 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。

## mybatis和hibernate的本质 区别

mybatis：学习成本低，入门门槛低，mybatis需要程序员自己写sql，对sql修改和优化就比较灵活。

mybatis是不完全的 ORM，mybatis需要程序员编写sql，但是mybatis也存在映射（输入映射、输出映射）

适用场景：需求变化较快的项目开发，比如：互联网项目、电商

hibernate：学习成本高，入门门槛高，hibernate是ORM框架，不需要程序员编写sql，自动根据对象映射生成sql。

适用场景：需求固定的中小型项目，OA系统、ERP系统。

企业在技术选型考虑各各技术框架的特点去进行选型 。

企业要根据人力、物力等资源去衡量，以节省成本利润最大化为目标进行选型。

# mybatis开发dao方法

## SqlSession在代码中的使用场合

使用场合：sqlSession在使用时由SqlSessionFactory生成创建一个Sqlsession会话对象，对象在代码中可以定义为成员变量、方法形参、方法的局部变量。

### SqlSessionFactory

SqlSessionFactory作为会话工厂，创建sqlSession，可以将SqlSessionFactory以单例方式。

### SqlSessionFactoryBuilder

SqlSessionFactoryBuilder用于创建SqlSessionFactory，又由于SqlSessionFactory以单例方式，可以将SqlSessionFactoryBuilder简单的当成一个工具类来使用即可。

### SqlSession

sqlSession是一个面向程序员接口。

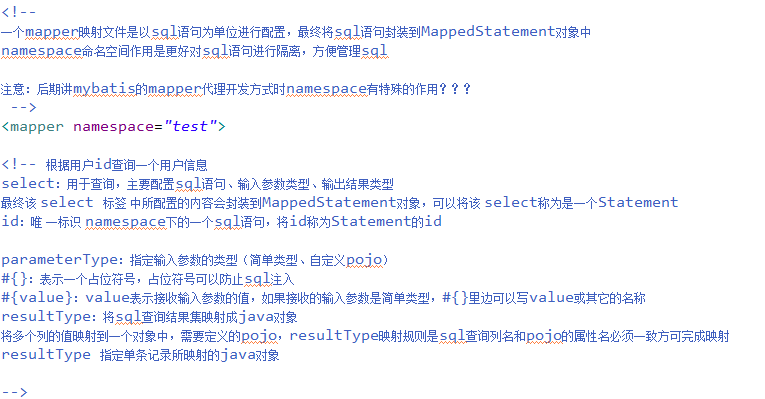
SqlSession作用是操作数据库，sqlSession对象要存储数据库连接、事务、一级缓存结构等。

sqlSession对象是线程不安全的（多线程访问系统，当多线程同时使用一个sqlsession对象时会造成数据冲突问题）

由于sqlSession对象是线程不安全的，sqlSession最佳使用场合在方法体内作为局部变量来使用。

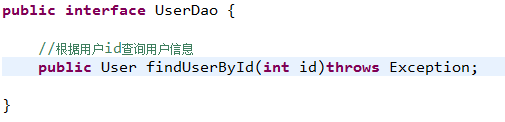
## 开发原始dao方法

### Used.xml映射文件

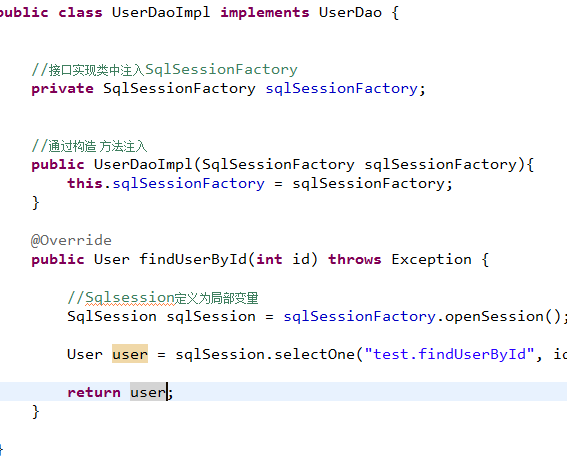


### dao接口和dao实现类

程序员需要编写dao接口及dao实现类。



接口实现类中注入SqlSessionFactory。





### 存在问题

1、dao接口实现类方法中存在大量的重复代码，这些重复的代码就是模板代码。

模板代码：

先创建sqlsession

再调用sqlsession的方法

再提交sqlsession

再关闭sqlsession

2、在调用sqlSession的方法时存在硬编码

调用sqlsession方法第一个参数是 statement 的id就是硬编码。

## mapper代理开发方法

程序员只需要编写dao接口（即mapper接口），由mybatis 框架自动生成接口实现的代理对象

### 分析

mapper代理方法要生成一个代理对象，代理对象内部执行的方法内容类似原始dao开发的dao实现类。

mapper代理类方法主要是对调用sqlsession的方法进行封装，如下：

sqlSession.selectOne("test.findUserById", id)

sqlSession.selectList("test.findUserByName", "张")

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

对以上代码进行哪些封装？

什么时候调用sqlSession 的selectOne方法、什么时候调用sqlSession 的selecList方法，什么时候 调用insert。。。。

解决方法：可以根据映射文件中标签类型一决定是执行select方法还是执行insert方法等。。

硬编码的statement的id如何封装。

解决方法：要解决映射文件中statement和mapper接口的方法的对应关系，在生成的代理对象方法中就知道该 代理对象的方法对应去调用哪个映射文件中statement。

### 开发规范

mybatis要解决映射文件中statement和mapper接口的方法对应起来。

可以遵循一个开发规范让statement和mapper接口的方法对应起来：

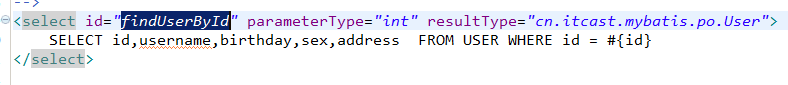
1、映射文件和mapper接口类的对应

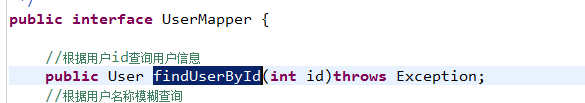
映射文件中namespace等于mapper接口类路径



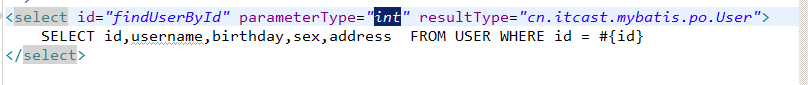
2、statement和mapper接口的方法对应

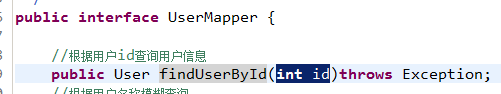
将映射文件中statement的id和mapper中方法名一致



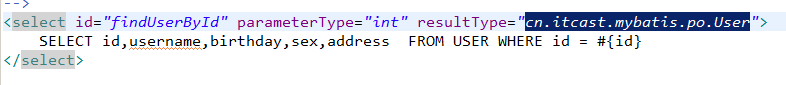


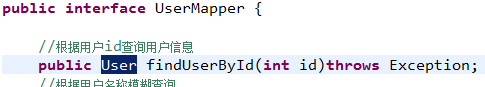
3、让mapper的接口方法的输入参数类型和statement的parameterType指定的参数类型保持一致





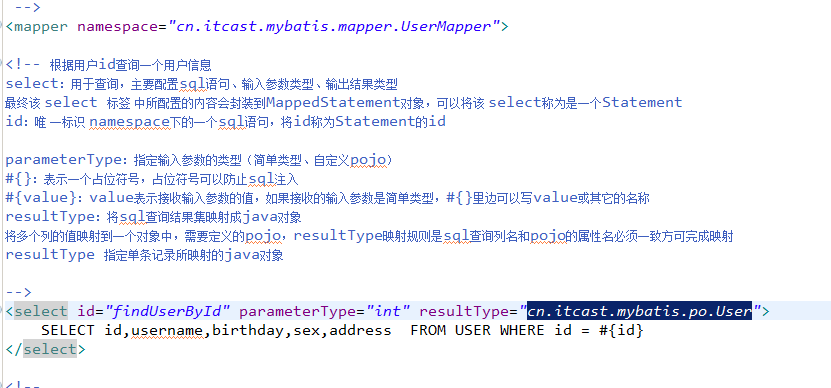
4、让mapper的接口方法返回值类型和statement的resultType指定的类型保持一致。



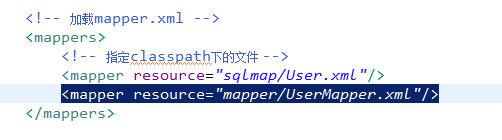


### 代码实现

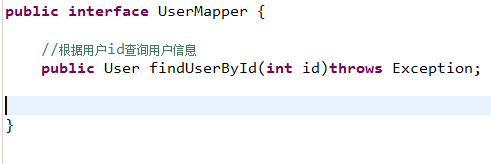
#### mapper.xml



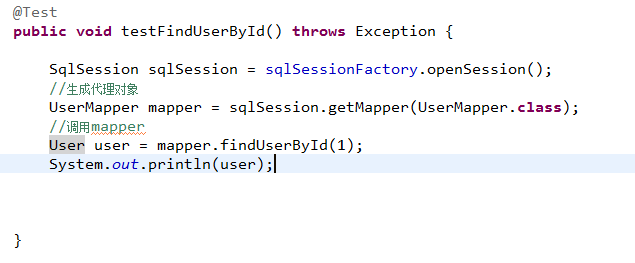
将mapper.xml在SqlMapConfig.xml加载



#### mapper.java

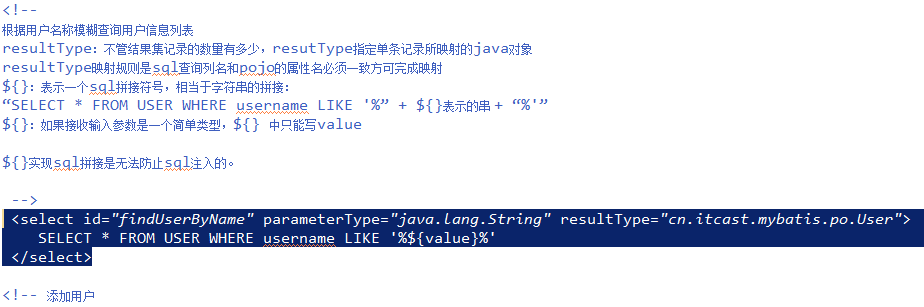


#### 测试



### 根据用户名称模糊查询

#### mapper.xml



#### mapper.java

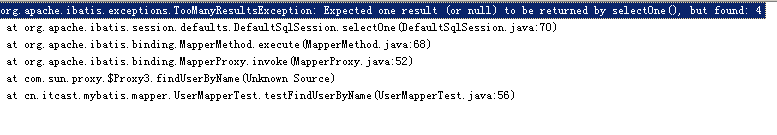


### 总结

* 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。

selectOne如果返回多条记录报错：



* 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。

# SqlMapConfig.xml

## 介绍

SqlMapconfig.xml是mybatis的全局配置文件，mybatis通过它创建SqlSessionFactory。

配置内容：

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）

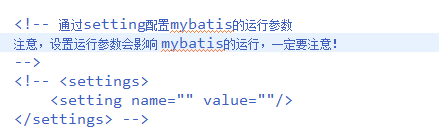
mappers（映射器）

## 属性



## settings

参考：mybatis-settings.xlsx



## typeAliases（重点）

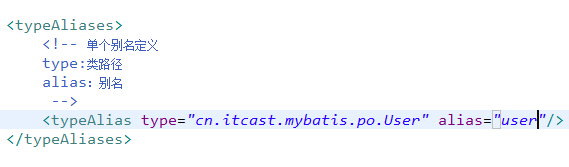
mybatis提供别名的定义，在mappe.xml中，指定输入参数类的全限定名不太方便，就可以定义一个别名代表参数类的全限定名。

mybatis提供了很多别名：

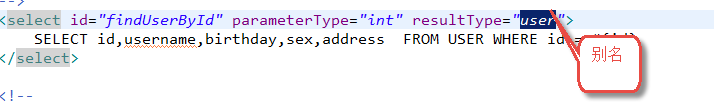
|  |  |
| --- | --- |
| 别名 | 映射的类型 |
| \_byte | byte |
| \_long | long |
| \_short | short |
| \_int | int |
| \_integer | int |
| \_double | double |
| \_float | float |
| \_boolean | boolean |
| string | String |
| byte | Byte |
| long | Long |
| short | Short |
| int | Integer |
| integer | Integer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

别名的定义：

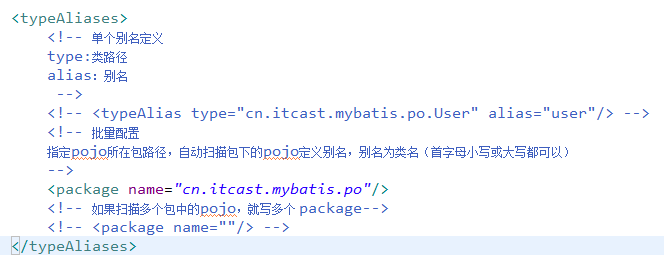
单个别名的定义



使用别名：



批量定义别名：



## mapper配置（重点）



# 输入映射

## 传递pojo包装对象

### 需求

用户综合条件查询功能。

使用mybatis实现用户查询列表，有很多查询条件（比如：商品信息、订单信息）。

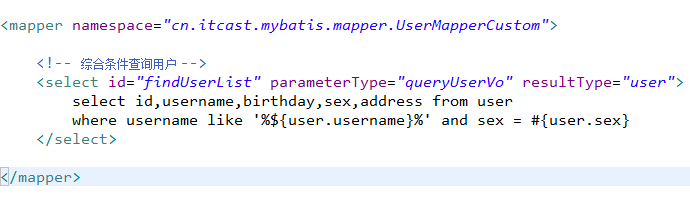
如何将综合查询条件传到mybatis的statement中，要使用包装对象传这些参数。

### 定义一个包装对象

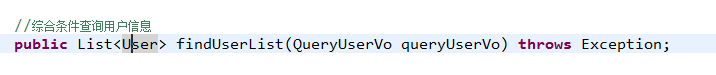
简单理解为：pojo包装了pojo。



### mapper.xml



### mapper.java



## 传递hashmap

Sql映射文件定义如下：

<!-- 传递hashmap综合查询用户信息 -->

<select id=*"findUserByHashmap"* parameterType=*"hashmap"* resultType=*"user"*>

select \* from user where id=#{id} and username like '%${username}%'

</select>

上边红色标注的是hashmap的key。

测试：

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

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

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

//构造查询条件Hashmap对象

HashMap<String, Object> map = **new** HashMap<String, Object>();

map.put("id", 1);

map.put("username", "管理员");

//传递Hashmap对象查询用户列表

List<User>list = userMapper.findUserByHashmap(map);

//关闭session

session.close();

}

# 输出映射

## resultType和resultMap介绍

* resultType：

指定输出映射类型，包括：java简单类型 string/int/float...、包括自定义的pojo。

如果指定自定义pojo，表示单条记录所映射pojo类型。

映射规则要求：sql查询列名和pojo的属性名一致方可映射成功。

通过测试发现：当所有列名和pojo属性名不一致，映射的对象为null

只要有一个 列名和pojo属性名一致，映射的对象不为null。

重点掌握：只要一个列名和一个属性名一致，该列就可以映射成功。

* resultMap

当列名和pojo属性名不一致时，可以通过resultMap定义列名和属性名的一个对应关系，完成映射。

## resultType

### 输出简单类型

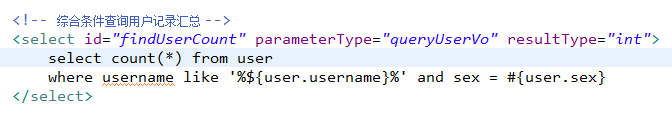
需求：

用户综合查询功能，要分页显示需要获取符合查询条件的记录总数

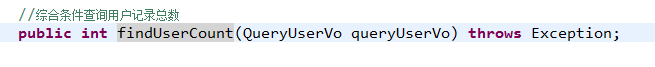
实现mybatis查询结果映射成int型。

当查询结果集只有一条记录且一列时可以将结果集映射为一个java简单基本类型。

mapper.xml



mapper.java



### 总结

输出pojo对象和输出pojo列表在sql中定义的resultType是一样的。

返回单个pojo对象要保证sql查询出来的结果集为单条，内部使用session.selectOne方法调用，mapper接口使用pojo对象作为方法返回值。

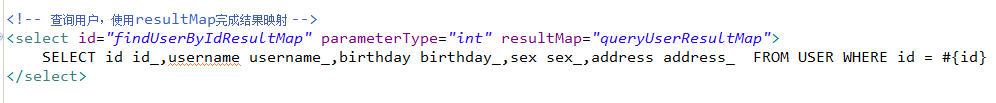
返回pojo列表表示查询出来的结果集可能为多条，内部使用session.selectList方法，mapper接口使用List<pojo>对象作为方法返回值。

## resultMap

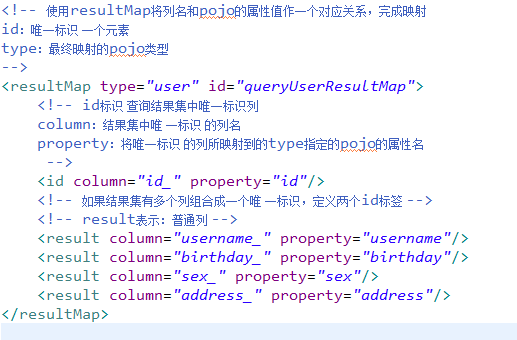
当列名和pojo属性名不一致时，可以通过resultMap定义列名和属性名的一个对应关系，完成映射。

resultMap还可以实现一些高级映射（一对一、一对多。）。

### mapper.xml



### resultMap定义



注意：定义resultMap也要指定一个映射的pojo类型。

### mapper.java



# 动态sql

## 什么是动态sql

mybatis重点是对sql语句是灵活的组织，由程序员编写sql语句，通过配置方式完成输入、输出映射。

mybatis通过标签拼接sql语句，去组织sql 语句。

## 需求

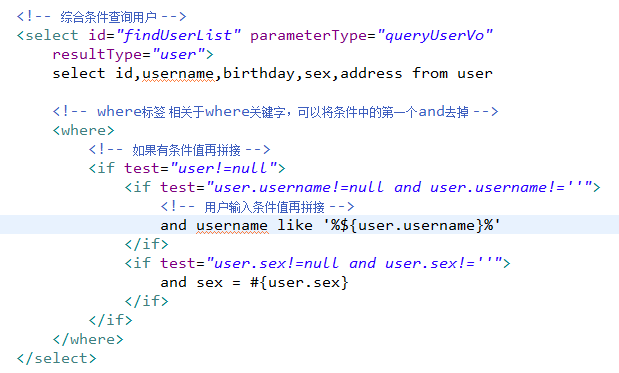
对用户综合查询statement定义进行优化，使用动态sql。

1、用户查询条件可以灵活拼接，当用户输入查询条件再拼接到sql中。

2、对mapper.xml中重复的sql抽取统一维护。

## if判断

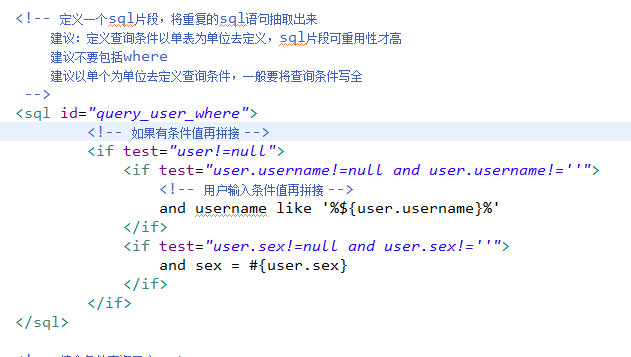
解决：用户查询条件可以灵活拼接，当用户输入查询条件再拼接到sql中



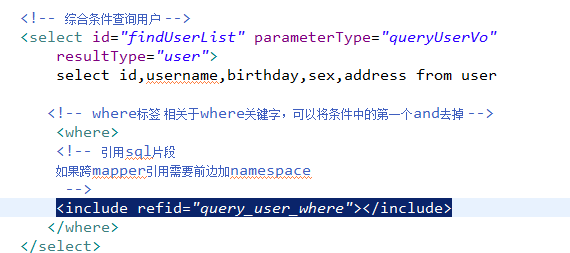
## sql片段

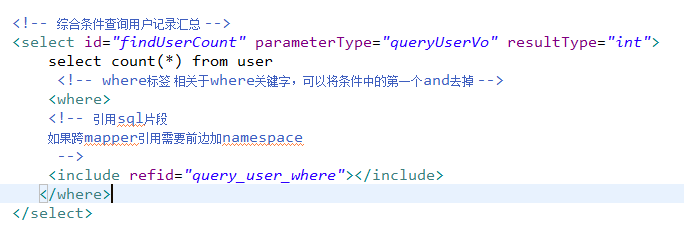
解决：对mapper.xml中重复的sql抽取统一维护

### 定义



### 引用 sql片段





## foreach

mybatis使用foreach循环遍历输入的参数。

### 需求

根据用户多个id查询用户信息。

实现方式：

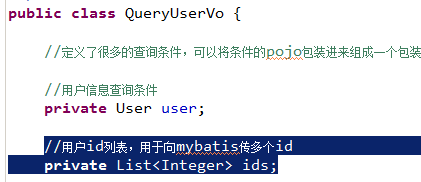
SELECT \* FROM USER WHERE sex = '1' AND id IN (1,10,16)

也可以：

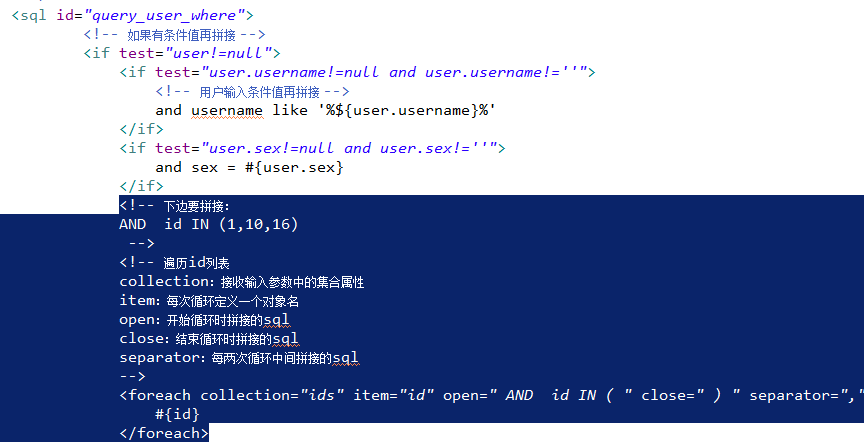
SELECT \* FROM USER WHERE sex = '1' AND (id=1 OR id=10 OR id=16)

### po类修改

使用queryUserVo包括对象使用List传递多个id



### mapper.xml



# 关联查询

mybatis实现表与表之间的查询，将查询结果集进行映射，重点讲resultMap完成高级映射，大家了解resultMap的使用方法。

## 数据模型

订单商品模型：



目标：分析表与表之间的关系。

分析步骤：

1、按模块去分析表，不要将系统全部表一次性分析

2、了解每个表存储了什么业务数据

3、了解表中关键字段（主键、外键、索引字段、非空字段）

4、了解表与表之间数据库级别的关系（外键关系）

5、表与表之间的业务关系（一对一、一对多、多对多）

注意：分析表与表之间的业务关系时一定要建立 在某种业务意义基础上

user表

id：主键（没有业务意义）

orders订单表

user\_id：用户id( 外键)

user\_id

一对一

orderdetail

订单明细表

orders\_id：订单id

items\_id：商品id

销售价

购买数量

orders\_id

外键

items商品表

name：商品名称

price：定价

items\_id外键

一对多

一对多

一对一

一对多

一对一

由于订单和订单明细的存在将用户和商品建立 关系

从用户到商品是一对多

从商品到用户是一对多

总：用户和商品之间是多对多

user表：存储了购买用户信息

网民，在电商系统中注册成为购买用户

orders表：存储了用户创建的订单信息

用户购买东西，一次不管购买多少商品只会创建一个订单

比如：张三一次购买手机、电脑，在orders表插入一条订单记录

orderdetail表：存储用户创建订单的详细信息，记录当时用户购买商品的购买信息

比如：张三一次购买手机、电脑，在orders表插入一条订单记录，在订单明细表插入两条记录，订单明细表每条记录对应一个商品信息

items商品表：存储了电商系统中的商品信息，用户在网站查看的商品信息就是来源与这张表

比如：网站销售手机，在这个表中插入很多的不同型号手机商品信息

orders--->user：

一个订单只能由一个用户创建，一对一关系

user--->orders：

一个用户可以创建多个订单，一对多关系

## 一对一查询

### 需求

查询订单关联查询用户信息

### 分析

使用mybatis开发持久层的dao接口时，要分析的重点就是sql语句如何实现？

主查询表：orders订单表

关联查询表：user用户信息

确定关联查询方法：包括内链接、外链接

由于orders表有外键user\_id，根据user\_id去查询user只能查询到一条记录的。就可以使用内链接。

SELECT

orders.\*,

user.username,

user.address

FROM

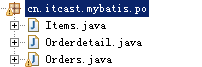
orders,

USER

WHERE orders.user\_id = user.id

### 基本数据模型po

一个表对应一个po类，表名和po类名一致，表的属性和po的属性一致。

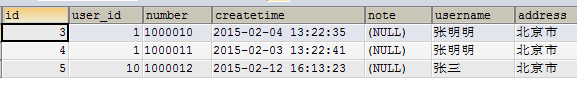


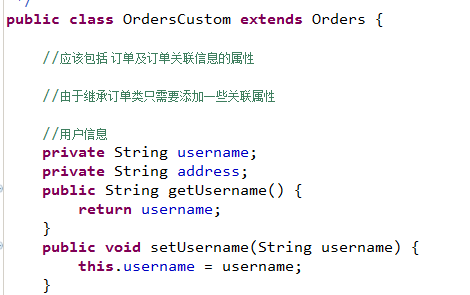
### resultType实现

resultType映射规则：sql查询出几条记录，只要保证sql列名和resulttype指定的pojo类型的属性一致，可以映射成功，sql查询出几条记录最终映射生成的List<pojo>就有几个对象。

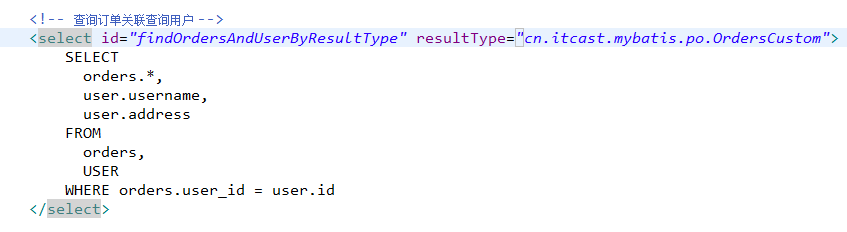
#### 自定义pojo

由于与表对应的po类的属性不能够全部包括下边列名，所以自定义pojo，属性名和下边的列名保持一致

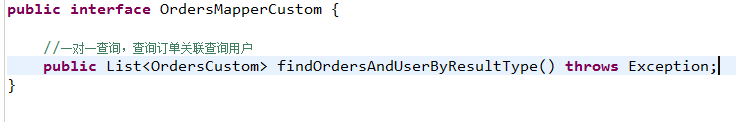




#### mapper.xml



#### mapper.java

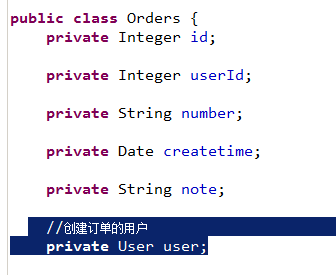


### resultMap实现

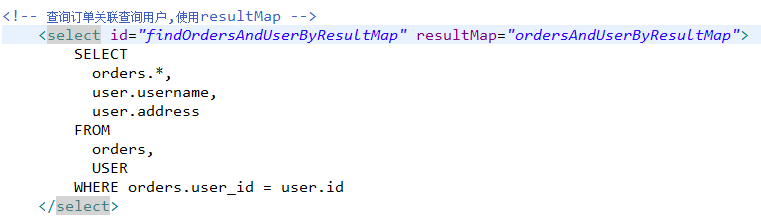
#### 需求

主查询是订单信息，要将主查询订单信息映射到orders对象中，在orders中创建user属性，让关联查询出来的用户信息映射到orders对象中的user属性中。其主要目标为了获取信息方便，如果想获取用户的信息，调用orders对象的getUser()方法即可。

#### 在orders中创建user属性



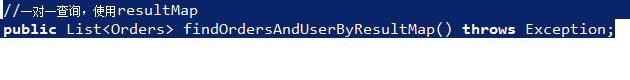
#### mapper.xml



#### resultMap



#### mapper.java



### 小结

如果没有特殊的要求，使用resultType是比较方便，只要能够保证sql查询的列和pojo属性名一致可以映射成功，如果pojo中属性少了，添加属性即可。

特殊的要求：

如果需要将关联查询的信息映射到pojo的pojo属性中， 为了获取数据方便，可以使用resultMap。

如果使用延迟加载，只能使用resultMap。

## 一对多查询

### 需求

查询订单关联查询明细信息

### 分析

主查询表：orders订单表

关联查询表：orderdetail订单明细表

使用内链接

SELECT

orders.\*,

user.username,

user.address,

orderdetail.id orderdetail\_id,

orderdetail.items\_id,

orderdetail.items\_num

FROM

orders,

USER ,

orderdetail

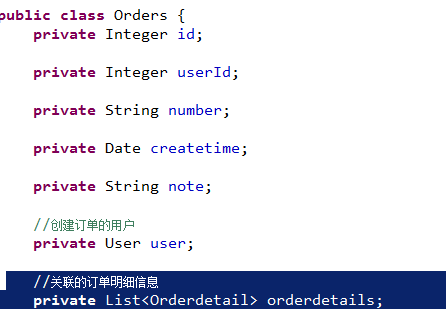
WHERE orders.user\_id = user.id AND orderdetail.orders\_id = orders.id

### 映射需求

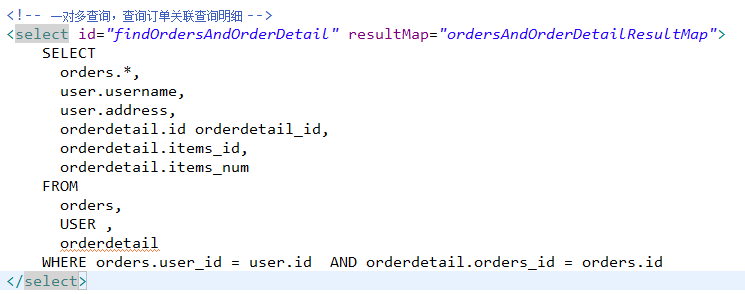
将主查询的订单信息映射到orders对象中，将订单关联的明细信息映射到orders对象中List<Orderdetail>

根据此需求分析使用resultType不能实现，只能使用resultMap

### 修改orders类添加List属性



### mapper.xml



### resultMap



### mapper.java



## association和collection小结

association：用于将关联查询信息映射到单个对象中。

标签注意使用javaType指定关联映射的对象类型

collection：用于将关联查询信息映射到集合对象中。

标签注意使用ofType指定关联映射的List中pojo的类型。

## 多对多查询

### 需求

查询用户信息及用户购买的商品信息

### 分析

主查询表：user用户信息

关联查询表：orders订单、orderdetail订单明细、items商品信息

采用内链接关联。

SELECT

orders.\*,

user.username,

user.address,

orderdetail.id orderdetail\_id,

orderdetail.items\_id,

orderdetail.items\_num,

items.name items\_name,

items.pic items\_pic

FROM

orders,

USER,

orderdetail,

items

WHERE orders.user\_id = user.id

AND orderdetail.orders\_id = orders.id

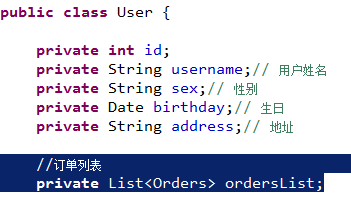
AND items.id = orderdetail.items\_id

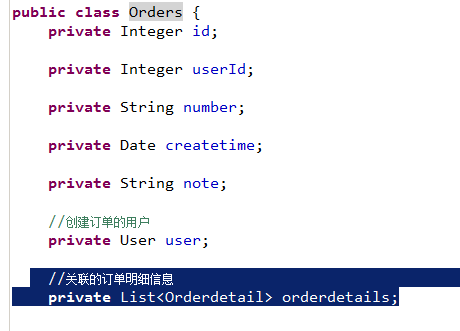
### 映射需求

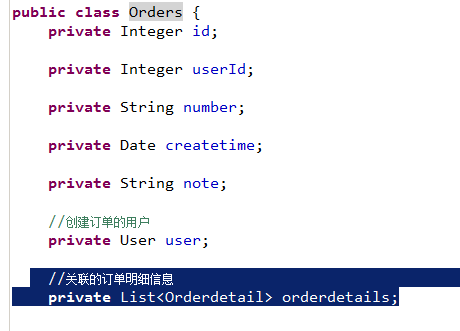
在user中设置一个List<Orders> orders属性，在order对象中设置List<Orderdetail> orderdeils 属性，在Orderdetail中设置Items属性。

最终查询出来的用户列表，用户信息映射到List<User>中，用户创建的订单信息隐射到user中的List<Orders> orders属性，订单下的明细映射到order对象中List<Orderdetail> orderdeils 属性，订单明细所对应的商品信息映射到Orderdetail中Items属性

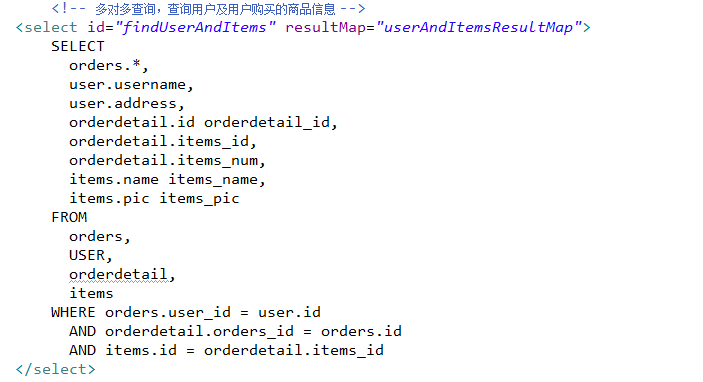
### 修改po类







### mapper.xml



### resultMap

<!-- 查询用户及用户购买的商品信息

最终要将查询结果映射成List<User>

-->

<resultMap type=*"user"* id=*"userAndItemsResultMap"*>

<!-- 用户信息 -->

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

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

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

<!-- 订单信息

一个用户可以创建多个订单

-->

<collection property=*"ordersList"* ofType=*"cn.itcast.mybatis.po.Orders"*>

<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"* />

<!-- 订单明细信息

一个订单包括多个明细信息

-->

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

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

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

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

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

<!-- 商品信息

一个明细对应一个商品

-->

<association property=*"items"* javaType=*"cn.itcast.mybatis.po.Items"*>

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

<result column=*"items\_name"* property=*"name"*/>

<result column=*"items\_pic"* property=*"pic"*/>

</association>

</collection>

</collection>

</resultMap>

### mapper.java



## resultMap总结

resultMap实现高级映射，如果对结果有高级映射的需求可以使用reusltMap，需求比如：将一个列表数据映射到pojo的list属性中采用resultmap。

如果仅仅是将查询列表简单映射为List<pojo>方式，列表中每个pojo对应一条记录，采用resultType即可。

## 延迟加载

如果刚开始仅仅查询一些简单的数据能够 满足用户的需求时，只需要查询简单的数据（尽量从单表中取数据），当用户需要查询关联信息时，此时我们再关联查询关联信息，叫延迟加载，对关联查询的信息进行延迟加载。

延迟加载其目的为了提高查询性能，减轻数据的压力。

### 需求

查询订单关联查询用户信息，对关联查询用户进行延迟加载。

### 实现方式1

定义两个mapper：

1、查询订单列表（只从订单表查询），单表查询

2、根据用户id查询用户信息。

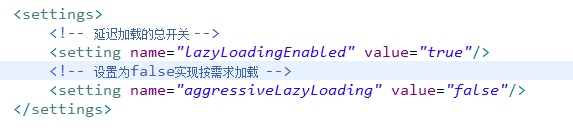
service调用第一个mapper查询出订单信息在页面显示出来

当用户在页面点击“查看用户”链接时通过service调用第二个mapper查询用户信息。

### resultMap实现延迟加载

#### 打开延迟加载的开关

在SqlMapConfig.xml中配置：



#### sql语句

查询订单的sql：

单表查询：

select \* from orders;

查询用户信息：

根据用户id查询用户信息

select \* from user where id=?

#### mapper.xml



#### resultMap定义延迟加载



#### mapper.java



#### 测试延迟加载



#### 总结

resultMap提供延迟加载，通过association可以延迟加载一个对象，通过collection可以延迟加载多个对象即集合对象。

延迟加载要注意实现的方法多种多样，主要目的是提高查询性能，减轻数据库压力，先查询简单数据，按 需查询关联数据。

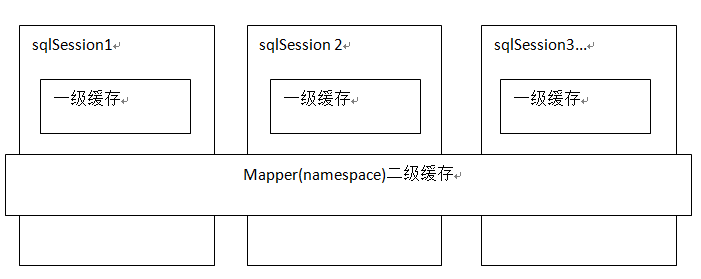
# mybatis查询缓存

什么是查询缓存？

将从数据库查询的数据存储到内存中缓存起来，这样就不用从数据库中查询数据而从缓存中查询，提高查询的速度，减少对数据库的访问。

mybatis提供查询缓存包括一级缓存、和二级缓存。

## 一级缓存和二级缓存原理



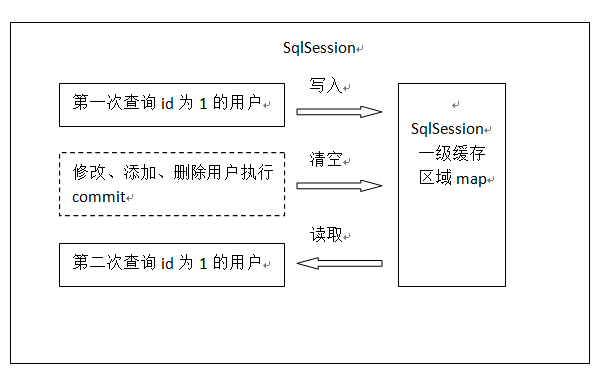
一级缓存是针对每一个sqlSession进行缓存。每个sqlSession对象中使用Map存储一级缓存数据，sqlSession对象销毁其中一级缓存数据不存在了。sqlSession与SqlSession之间的一级缓存互相不影响。

map中存储了sql执行查询的结果集（java对象）。

二级缓存是针对每个mapper相同 的namespace进行缓存。每个SqlSession都要调用mapper下的sql语句，在mapper级别设置了二级缓存的数据结构map，每个mapper对应一个map数据结构，map中存储了二级缓存的数据，存储了sql执行查询的结果集（java对象）。

每个SqlSession都可以访问到二级缓存中的数据，sqlsession对象销毁mapper中的二级缓存数据仍然存在。

一级缓存运行原理：

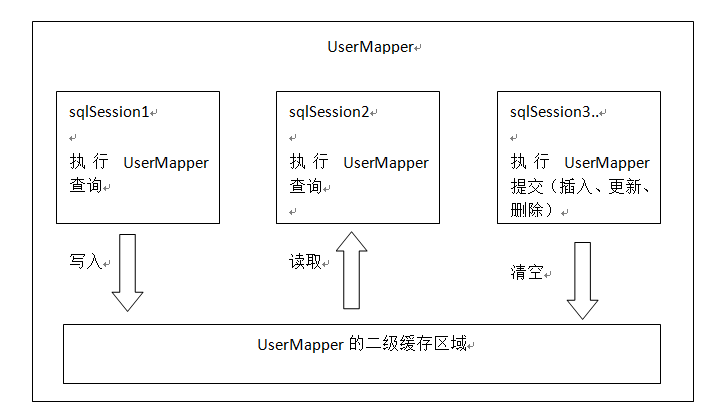


第一次查询先去缓存中找是否有缓存数据，发现没有，查询数据库，将查询到的数据写入sqlsession的一级缓存区域。

第二次查询先去缓存中找是否有缓存数据，发现有，直接从缓存区域中取出数据返回。

如果 执行sqlsession的添加、修改、删除等操作，会执行commit，最终会清空缓存。

二级缓存原理：



不同的sqlsession都要调用mapper下的sql语句发起数据库请求。

sqlsession1执行UserMapper下的查询用户请求先从二级缓存中查找有没有数据，如果没有就从数据库中查询，并且将查询到数据存储二级缓存中。

sqlsession2执行UserMapper下的同一个查询用户请求，先从二级缓存中查找有没有数据，如果有就从二级缓存中查询数据，返回。

如果有一个sqlsession3执行UserMapper下添加、修改、删除语句，执行commit操作后，将UserMapper下的所有缓存数据全部清空。

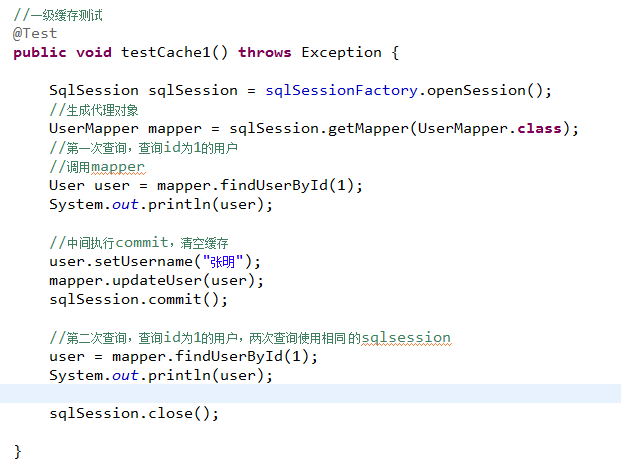
## 一级缓存测试

当两次查询相同 的sql时，第二次查询要缓存中查询，而不从数据库查询。

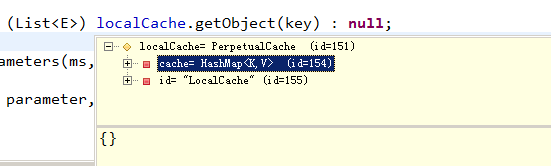
注意：

1、相同 的sql定义：相同 mapper namespace下的相同 的statement下的sql，且sql的输入参数必须一致。

2、两次查询使用相同 的sqlsession对象。



跟踪源代码，一级缓存的区域：



## 二级缓存测试

当两次查询相同 的sql时，第二次查询要缓存中查询，而不从数据库查询。

注意：

1、相同 的sql定义：相同 mapper namespace下的相同 的statement下的sql，且sql的输入参数必须一致。

2、两次查询可以使用不同 的sqlsession对象。

### 开启二级缓存

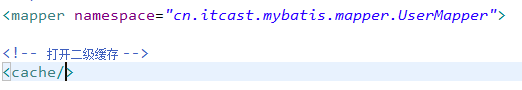
mybatis对一级缓存是默认支持，对二级缓存需要开启。

打开总开关：

在核心配置文件SqlMapConfig.xml中加入

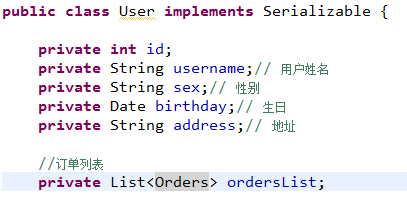
<setting name=*"cacheEnabled"* value=*"true"*/>

打开mapper的二级缓存开关：



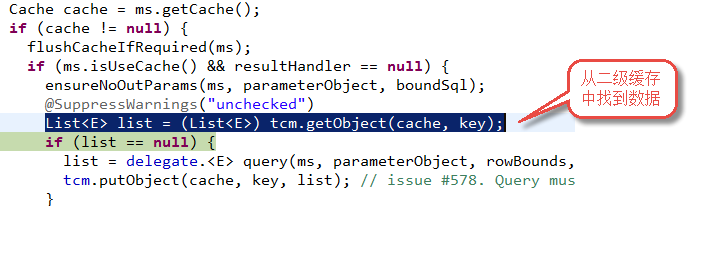
### 对象序列化

mybatis支持使用第三方缓存框架实现二级缓存数据的存储，要求对象必须实现序列化。



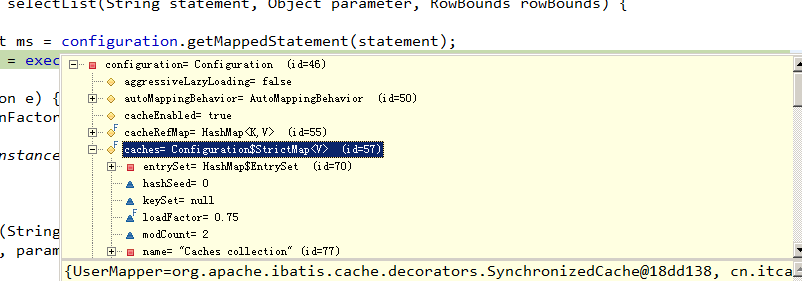
### 测试代码

每次查询都会去先查找二级缓存中是否有数据：

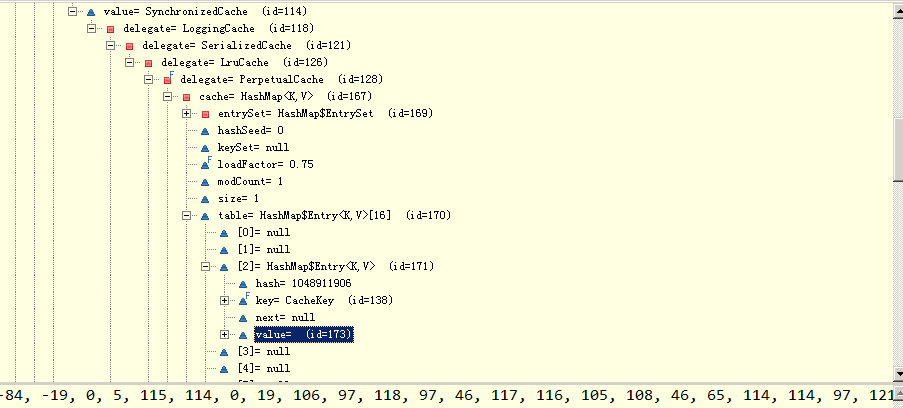


通过源码跟踪，二级缓存的数据结构，是一个全局的对象，不受sqlsession的影响，mybatis程序运行起来二级缓存的数据结构就创建出来，直到mybatis程序停止二级缓存数据结构销毁：

二级缓存在一个全局对象中即configruation中：



详细map数据结构如下：



测试代码：

// 二级缓存测试

@Test

**public** **void** testCache2() **throws** Exception {

//二级缓存是跨sqlsession的是基于mapper级别的缓存，sqlsession可以使用多个对象，多个对象必须要访问同一个mapper下的sql语句

SqlSession sqlSession1 = sqlSessionFactory.openSession();

SqlSession sqlSession2 = sqlSessionFactory.openSession();

SqlSession sqlSession3 = sqlSessionFactory.openSession();

SqlSession sqlSession4 = sqlSessionFactory.openSession();

// 生成代理对象

UserMapper mapper1 = sqlSession1.getMapper(UserMapper.**class**);

UserMapper mapper2 = sqlSession2.getMapper(UserMapper.**class**);

UserMapper mapper3 = sqlSession3.getMapper(UserMapper.**class**);

UserMapper mapper4 = sqlSession4.getMapper(UserMapper.**class**);

// 第一次查询，查询id为1的用户

// 调用mapper

User user = mapper1.findUserById(1);

System.*out*.println(user);

//第一次查询用户列表

List<User> list = mapper4.findUserByName("张");

System.*out*.println(list);

//关闭sqlsession1，此时将数据写入二级缓存

sqlSession1.close();

sqlSession4.close();

// 中间执行commit，清空缓存，将userMapper下手所有缓存数据全部

user.setUsername("张明明");

mapper3.updateUser(user);

sqlSession3.commit();

sqlSession3.close();

// 第二次查询，查询id为1的用户，两次查询使用相同 的sqlsession

user = mapper2.findUserById(1);

System.*out*.println(user);

List<User> list2 = mapper2.findUserByName("张");

System.*out*.println(list2);

sqlSession2.close();

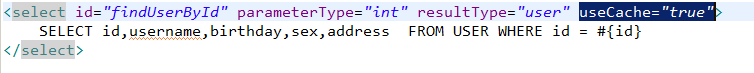
sqlSession3.close();

}

## 二级缓存深入学习

### 设置statement是否二级缓存

可以针对每一个statement进行测试是否二级缓存，默认是进行二级缓存。



useCache=true表示要二级缓存，一般要对查询进行二级缓存。

useCache=false，查询该 statement不进行二级缓存，针对那些查询非常频繁的语句，信息的变化性较高，可以将useCache设置false不进行二级缓存。

### 刷新缓存

针对insert、update、delete这些statement设置flushCache="true" 表示执行commit就清空二级缓存，如果设置flushCache="false"，执行操作也不清空缓存。

一般情况下需要设置flushCache="true" ，为了避免查询出脏数据。

### 其它的参数

flushInterval（刷新间隔）可以被设置为任意的正整数，而且它们代表一个合理的毫秒形式的时间段。默认情况是不设置，也就是没有刷新间隔，缓存仅仅调用语句时刷新。

size（引用数目）可以被设置为任意正整数，要记住你缓存的对象数目和你运行环境的可用内存资源数目。默认值是1024。

readOnly（只读）属性可以被设置为true或false。只读的缓存会给所有调用者返回缓存对象的相同实例。因此这些对象不能被修改。这提供了很重要的性能优势。可读写的缓存会返回缓存对象的拷贝（通过序列化）。这会慢一些，但是安全，因此默认是false。

如下例子：

<cache  eviction="FIFO"  flushInterval="60000"  size="512"  readOnly="true"/>

这个更高级的配置创建了一个 FIFO 缓存,并每隔 60 秒刷新,存数结果对象或列表的 512 个引用,而且返回的对象被认为是只读的,因此在不同线程中的调用者之间修改它们会导致冲突。可用的收回策略有, 默认的是 LRU:

1. LRU – 最近最少使用的:移除最长时间不被使用的对象。
2. FIFO – 先进先出:按对象进入缓存的顺序来移除它们。
3. SOFT – 软引用:移除基于垃圾回收器状态和软引用规则的对象。
4. WEAK – 弱引用:更积极地移除基于垃圾收集器状态和弱引用规则的对象。

### mybatis整合ehcache

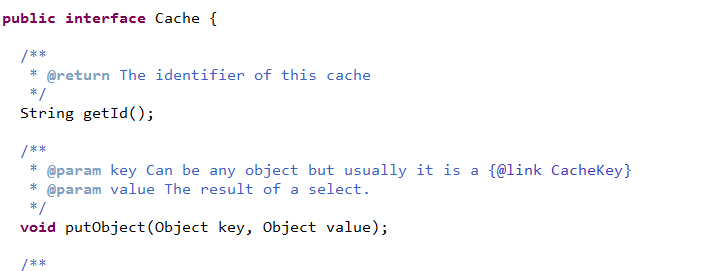
mybaits是一个持久框架，对缓存数据处理没有第三方专门做缓存框架优秀。

mybatis提供一个Cache接口供第三方框架整合。

建议使用第三方缓存框架和mybatis整合，比如：ehcache、redis、memcache。

mybatis提供二级缓存Cache接口，如下：



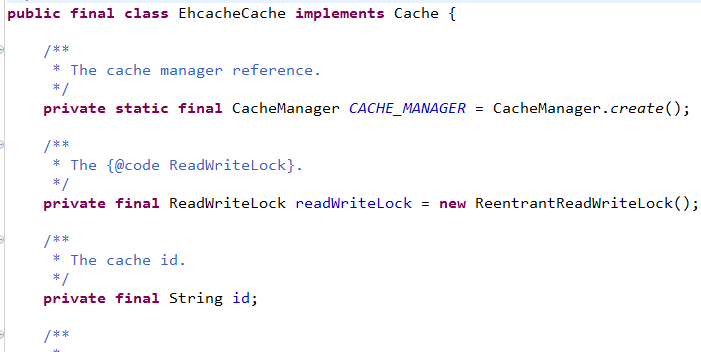


它的默认实现类：



#### 第一步：引入ehcache缓存的依赖包





#### 第二步，配置ehcache的配置文件



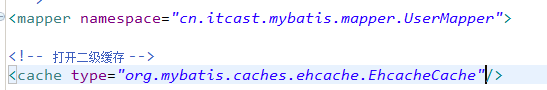
#### 第三步，开启ehcache缓存

实现让echcache托管mybaits二级缓存数据的存取。

修改mapper.xml文件，在cache中指定EhcacheCache。

<cache type=*"org.mybatis.caches.ehcache.**EhcacheCache"*/>

配置如下：



根据需求调整缓存参数：

<cache type=*"org.mybatis.caches.ehcache.EhcacheCache"* >

<property name=*"timeToIdleSeconds"* value=*"3600"*/>

<property name=*"timeToLiveSeconds"* value=*"3600"*/>

<!-- 同ehcache参数maxElementsInMemory -->

<property name=*"maxEntriesLocalHeap"* value=*"1000"*/>

<!-- 同ehcache参数maxElementsOnDisk -->

<property name=*"maxEntriesLocalDisk"* value=*"10000000"*/>

<property name=*"memoryStoreEvictionPolicy"* value=*"LRU"*/>

</cache>

### mybatis二级缓存应用场景

对于访问多的查询请求且用户对查询结果实时性要求不高，此时可采用mybatis二级缓存技术降低数据库访问量，提高访问速度，业务场景比如：耗时较高的统计分析sql、电话账单查询sql等。

### mybatis二级缓存局限性

mybatis二级缓存对细粒度的数据级别的缓存实现不好，比如如下需求：对商品信息进行缓存，由于商品信息查询访问量大，但是要求用户每次都能查询最新的商品信息，此时如果使用mybatis的二级缓存就无法实现当一个商品变化时只刷新该商品的缓存信息而不刷新其它商品的信息，因为mybaits的二级缓存区域以mapper为单位划分，当一个商品信息变化会将所有商品信息的缓存数据全部清空。解决此类问题需要在业务层根据需求对数据有针对性缓存。

# mybatis与spring整合

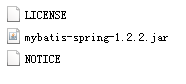
在实际开发中需要将mybaits与spring进行整合开发，让spring管理mybatis的SqlSessionFactory及dao（原始dao实现类及mapper接口代理对象）

## 搭建整合环境

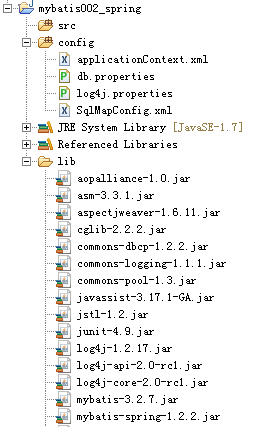
spring：3.2.0

mybatis：3.2.7

mybaits-spring的整合包(由mybatis提供)：



数据驱动包。。。



## 原始dao开发

### SqlMapConfig.xml

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

<!DOCTYPE configuration

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

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

<configuration>

<!-- 通过setting配置mybatis的运行参数

注意，设置运行参数会影响 mybatis的运行，一定要注意！

-->

<settings>

<!-- 延迟加载的总开关 -->

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

<!-- 设置为false实现按需求加载 -->

<setting name=*"aggressiveLazyLoading"* value=*"false"*/>

<!-- 开启二级缓存 -->

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

</settings>

<!-- 定义别名 -->

<typeAliases>

<!-- 单个别名定义

type:类路径

alias：别名

-->

<!-- <typeAlias type="cn.itcast.mybatis.po.User" alias="user"/> -->

<!-- 批量配置

指定pojo所在包路径，自动扫描包下的pojo定义别名，别名为类名（首字母小写或大写都可以）

-->

<package name=*"cn.itcast.mybatis.po"*/>

<!-- 如果扫描多个包中的pojo，就写多个 package-->

<!-- <package name=""/> -->

</typeAliases>

<!-- 加载mapper.xml -->

<mappers>

<!-- 指定classpath下的映射文件 -->

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

<!-- <mapper resource="mapper/UserMapper.xml" /> -->

<!-- 通过mapper.java进行加载映射

class指定mapper接口 的路径

要求：mapper.java文件和mapper.xml同名且在同一个目录

-->

<!-- <mapper class="cn.itcast.mybatis.mapper.UserMapper"/> -->

<!-- 包扫描方式

指定mapper接口所在包，自动扫描mapper接口进行加载

要求：mapper.java文件和mapper.xml同名且在同一个目录

-->

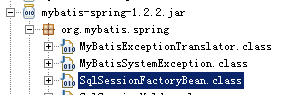
<package name=*"cn.itcast.mybatis.mapper"*/>

</mappers>

</configuration>

### 在spring容器中配置sqlSessionFactory

使用mybatis与spring整合包中的类创建sqlSesionFactory

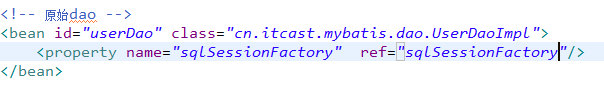




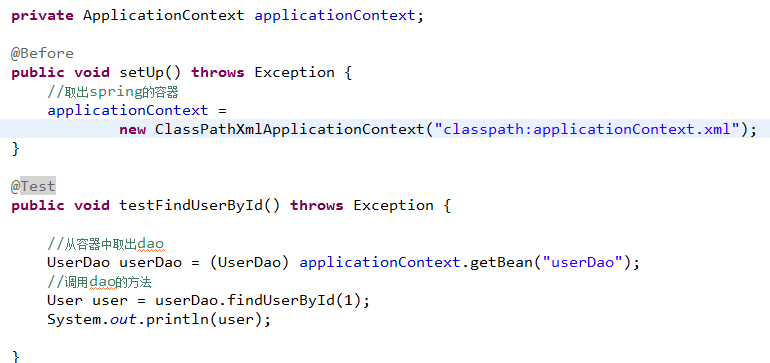
### Dao接口实现类继承SqlSessionDaoSupport



### 在spring容器中配置dao



### 从spring容器取出dao实例进行测试



## mapper代理开发

### SqlMapConfig.xml

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

<!DOCTYPE configuration

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

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

<configuration>

<!-- 通过setting配置mybatis的运行参数

注意，设置运行参数会影响 mybatis的运行，一定要注意！

-->

<settings>

<!-- 延迟加载的总开关 -->

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

<!-- 设置为false实现按需求加载 -->

<setting name=*"aggressiveLazyLoading"* value=*"false"*/>

<!-- 开启二级缓存 -->

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

</settings>

<!-- 定义别名 -->

<typeAliases>

<!-- 单个别名定义

type:类路径

alias：别名

-->

<!-- <typeAlias type="cn.itcast.mybatis.po.User" alias="user"/> -->

<!-- 批量配置

指定pojo所在包路径，自动扫描包下的pojo定义别名，别名为类名（首字母小写或大写都可以）

-->

<package name=*"cn.itcast.mybatis.po"*/>

<!-- 如果扫描多个包中的pojo，就写多个 package-->

<!-- <package name=""/> -->

</typeAliases>

<!-- 加载mapper.xml -->

<mappers>

<!-- 指定classpath下的映射文件 -->

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

<!-- <mapper resource="mapper/UserMapper.xml" /> -->

<!-- 通过mapper.java进行加载映射

class指定mapper接口 的路径

要求：mapper.java文件和mapper.xml同名且在同一个目录

-->

<!-- <mapper class="cn.itcast.mybatis.mapper.UserMapper"/> -->

<!-- 包扫描方式

指定mapper接口所在包，自动扫描mapper接口进行加载

要求：mapper.java文件和mapper.xml同名且在同一个目录

-->

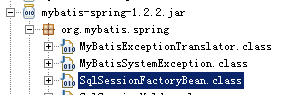
<package name=*"cn.itcast.mybatis.mapper"*/>

</mappers>

</configuration>

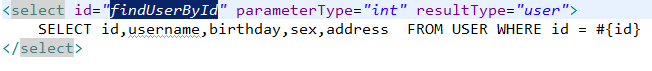
### 在spring容器中配置sqlSessionFactory

使用mybatis与spring整合包中的类创建sqlSesionFactory

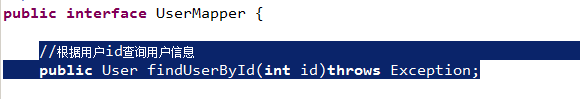




### mapper.xml

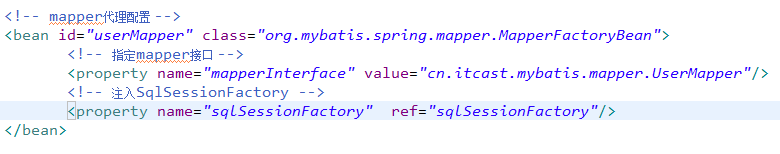


### mapper.java



### spring与mybatis整合生成代理对象方法1 MapperFactoryBean

使用org.mybatis.spring.mapper.MapperFactoryBean



使用此方法需要在spring容器中配置很多的mapper的bean，比较麻烦，不推荐使用。

### spring与mybatis整合生成代理对象方法2 扫描器



推荐使用mapper扫描器！！！

# mybatis逆向工程

## 什么是mybatis逆向工程

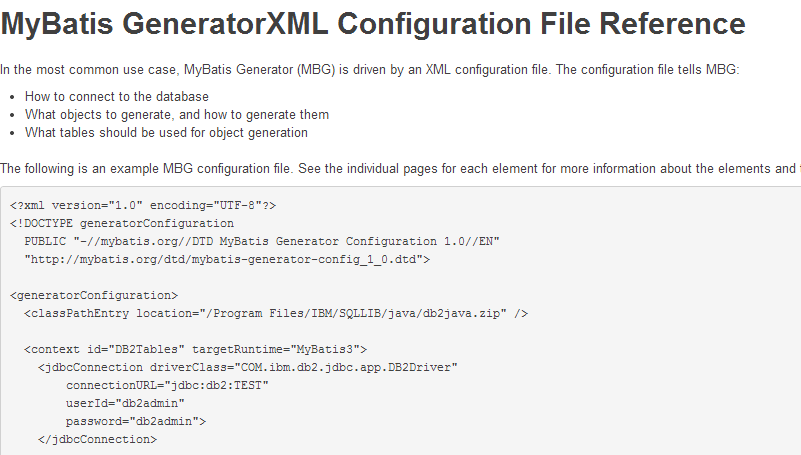
在使用mybatis时需要程序员自己编写sql语句，针对单表的sql语句量是很大的，mybatis官方提供了一种根据数据库表生成mybatis执行代码的工具，这个工具就是一个逆向工程。

逆向工程：针对数据库单表----》生成代码（mapper.xml、mapper.java、pojo。。）



mybatis-generator-core-1.3.2.jar---逆向工程运行所需要的jar核心 包

## 配置逆向工程的配置文件



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

<!DOCTYPE generatorConfiguration

PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"

"http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd">

<generatorConfiguration>

<context id=*"testTables"* targetRuntime=*"MyBatis3"*>

<commentGenerator>

<!-- 是否去除自动生成的注释 true：是 ： false:否 -->

<property name=*"suppressAllComments"* value=*"true"* />

</commentGenerator>

<!--数据库连接的信息：驱动类、连接地址、用户名、密码 -->

<jdbcConnection driverClass=*"com.mysql.jdbc.Driver"*

connectionURL=*"jdbc:mysql://localhost:3306/mybatis"* userId=*"root"*

password=*"mysql"*>

</jdbcConnection>

<!-- <jdbcConnection driverClass="oracle.jdbc.OracleDriver"

connectionURL="jdbc:oracle:thin:@127.0.0.1:1521:yycg"

userId="yycg"

password="yycg">

</jdbcConnection> -->

<!-- 默认false，把JDBC DECIMAL 和 NUMERIC 类型解析为 Integer，为 true时把JDBC DECIMAL 和

NUMERIC 类型解析为java.math.BigDecimal -->

<javaTypeResolver>

<property name=*"forceBigDecimals"* value=*"false"* />

</javaTypeResolver>

<!-- targetProject:生成PO类的位置 -->

<javaModelGenerator targetPackage=*"cn.itcast.mybatis.po"*

targetProject=*".\src"*>

<!-- enableSubPackages:是否让schema作为包的后缀 -->

<property name=*"enableSubPackages"* value=*"false"* />

<!-- 从数据库返回的值被清理前后的空格 -->

<property name=*"trimStrings"* value=*"true"* />

</javaModelGenerator>

<!-- targetProject:mapper映射文件生成的位置 -->

<sqlMapGenerator targetPackage=*"cn.itcast.mybatis.mapper"*

targetProject=*".\src"*>

<!-- enableSubPackages:是否让schema作为包的后缀 -->

<property name=*"enableSubPackages"* value=*"false"* />

</sqlMapGenerator>

<!-- targetPackage：mapper接口生成的位置 -->

<javaClientGenerator type=*"XMLMAPPER"*

targetPackage=*"cn.itcast.mybatis.mapper"*

targetProject=*".\src"*>

<!-- enableSubPackages:是否让schema作为包的后缀 -->

<property name=*"enableSubPackages"* value=*"false"* />

</javaClientGenerator>

<!-- 指定数据库表 -->

<table tableName=*"items"*></table>

<!-- 有些表的字段需要指定java类型

<table schema="" tableName="">

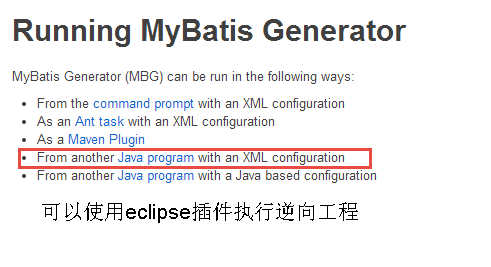
<columnOverride column="" javaType="" />

</table> -->

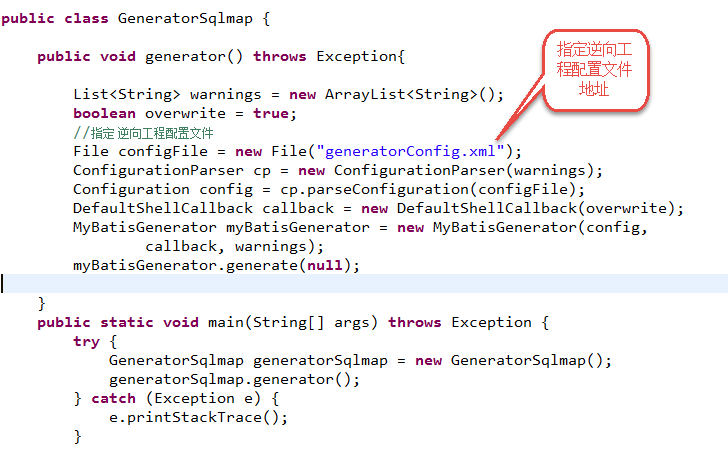
</context>

</generatorConfiguration>

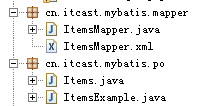
## 执行逆向工程生成代码



执行java类方法：



生成的代码如下：



## 将生成的代码拷贝到业务系统工程中测试

**public** **class** ItemsMapperTest {

**private** ApplicationContext applicationContext;

**private** ItemsMapper itemsMapper;

@Before

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

//取出spring的容器

applicationContext =

**new** ClassPathXmlApplicationContext("classpath:applicationContext.xml");

itemsMapper= (ItemsMapper) applicationContext.getBean("itemsMapper");

}

//根据主键删除

@Test

**public** **void** testDeleteByPrimaryKey() {

//根据主键删除

// itemsMapper.deleteByPrimaryKey(id);

//将符合条件的记录删除，参考下边的查询自定义条件example

// itemsMapper.deleteByExample(example);

}

//插入商品

@Test

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

// itemsMapper.insert(record);

}

//自定义条件查询

@Test

**public** **void** testSelectByExample() {

ItemsExample itemsExample = **new** ItemsExample();

ItemsExample.Criteria criteria = itemsExample.createCriteria();

//使用criteria自定义查询条件

criteria.andNameEqualTo("水杯");

criteria.andIdEqualTo(1);

List<Items> list = itemsMapper.selectByExample(itemsExample);

System.*out*.println(list);

}

//根据主键查询

@Test

**public** **void** testSelectByPrimaryKey() {

Items items = itemsMapper.selectByPrimaryKey(1);

System.*out*.println(items);

}

//更新

@Test

**public** **void** testUpdateByPrimaryKey() {

//根据主键更新，不管更新对象的属性是否为空全部更新

// itemsMapper.updateByPrimaryKey(record)

//根据主键更新，只更新不为空的属性，参考mapper.xml中的代码

// itemsMapper.updateByPrimaryKeySelective(record)

}