案例：<https://gitee.com/zx19890628/spring-boot-example/tree/master/lab_012_db_mybatis>

反向工程案例：

生产可用：<https://gitee.com/zx19890628/spring-boot-example/tree/master/lab_019_db_mybatis_generator>

官网中文文档：<http://www.mybatis.org/mybatis-3/zh/index.html>

原理： <https://mp.weixin.qq.com/s/BcnO8m8cBpYa5ODuSMSiMA>

mybatis 第一天 mybatis的基础知识

官方文档地址：http://www.mybatis.org/mybatis-3/zh/index.html

课程安排：

mybatis和springmvc通过订单商品 案例驱动

第一天：基础知识（重点，内容量多）

对原生态jdbc程序（单独使用jdbc开发）问题总结

mybatis框架原理 （掌握）

mybatis入门程序

用户的增、删、改、查

mybatis开发dao两种方法：

原始dao开发方法（程序需要编写dao接口和dao实现类）（掌握）

mybaits的mapper接口（相当于dao接口）代理开发方法（掌握）

mybatis配置文件SqlMapConfig.xml

mybatis核心：

mybatis输入映射（掌握）

mybatis输出映射（掌握）

mybatis的动态sql（掌握）

第二天：高级知识

订单商品数据模型分析

高级结果集映射（一对一、一对多、多对多）

mybatis延迟加载

mybatis查询缓存（一级缓存、二级缓存）

mybaits和spring进行整合（掌握）

mybatis逆向工程

# 对原生态jdbc程序中问题总结

## 环境

java环境：jdk1.7.0\_72

eclipse：indigo

mysql：5.1

## 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**



## 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：



## 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。

# mybatis框架

## 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）

## mybatis程序执行流程（原理）



# 入门程序

## 需求

根据用户id（主键）查询用户信息

根据用户名称模糊查询用户信息

添加用户

删除 用户

更新用户

## 环境

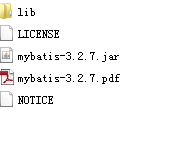
java环境：jdk1.7.0\_72

eclipse：indigo

mysql：5.1

mybatis运行环境（jar包）：

从https://github.com/mybatis/mybatis-3/releases下载，3.2.7版本



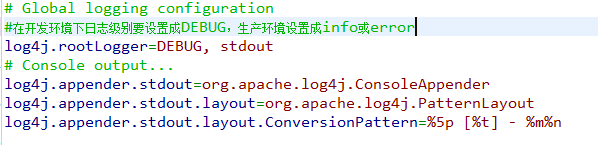
lib下：依赖包

mybatis-3.2.7.jar：核心 包

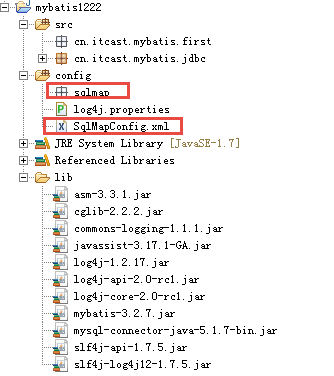
mybatis-3.2.7.pdf，操作指南

加入mysql的驱动包

## log4j.properties



## 工程结构



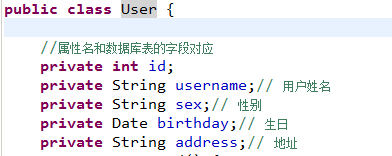
## SqlMapConfig.xml

配置mybatis的运行环境，数据源、事务等。



## 根据用户id（主键）查询用户信息

### 创建po类

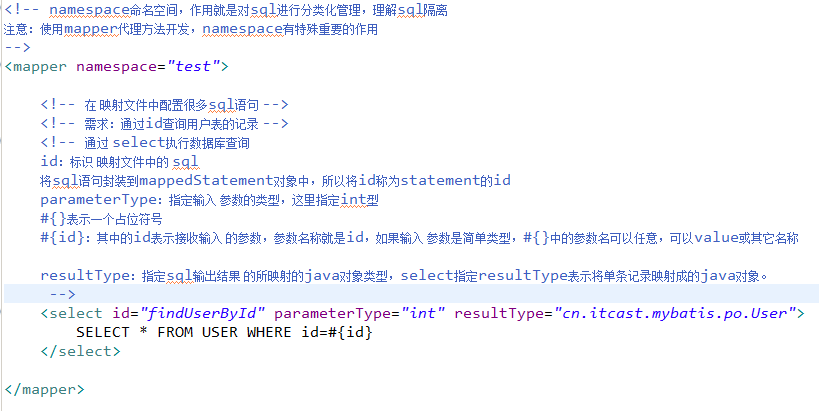


### 映射文件

映射文件命名：

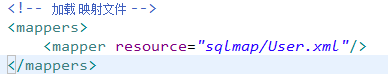
User.xml（原始ibatis命名），mapper代理开发映射文件名称叫XXXMapper.xml，比如：UserMapper.xml、ItemsMapper.xml

在映射文件中配置sql语句。



### 在SqlMapConfig.xml加载映射文件

在sqlMapConfig.xml中加载User.xml:



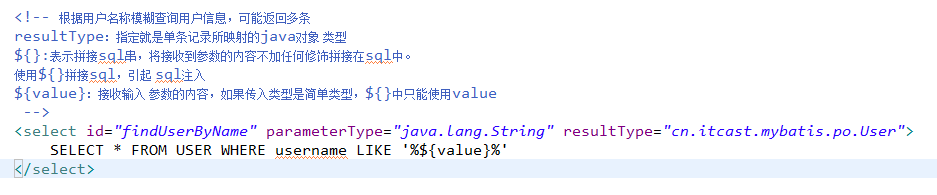
### 程序编写



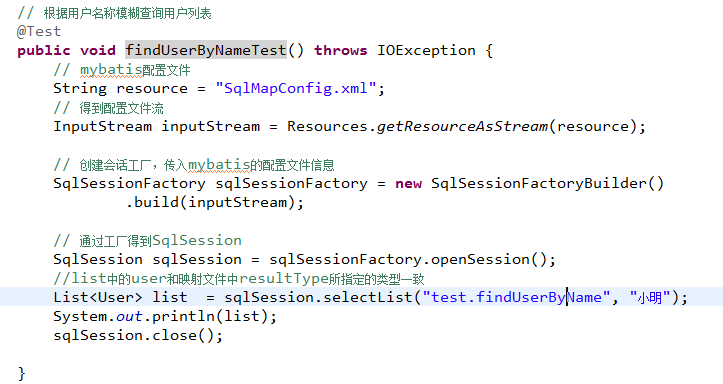
## 根据用户名称模糊查询用户信息

### 映射文件

使用User.xml，添加根据用户名称模糊查询用户信息的sql语句。



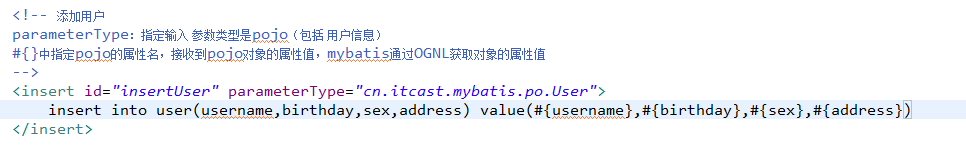
### 程序代码



## 添加用户

### 映射文件

在 User.xml中配置添加用户的Statement



### 程序代码



### 自增主键返回

mysql自增主键，执行insert提交之前自动生成一个自增主键。

通过mysql函数获取到刚插入记录的自增主键：

LAST\_INSERT\_ID()

是insert之后调用此函数。

修改insertUser定义：



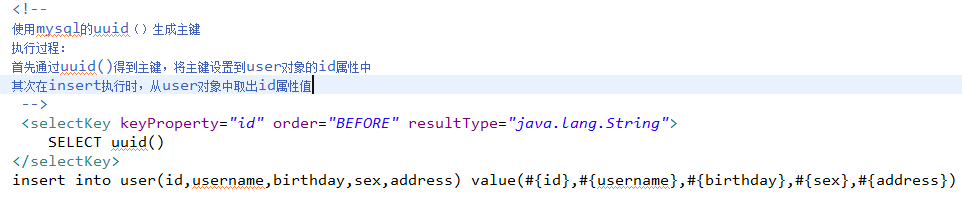
### 非自增主键返回(使用uuid())

使用mysql的uuid()函数生成主键，需要修改表中id字段类型为string，长度设置成35位。

执行思路：

先通过uuid()查询到主键，将主键输入 到sql语句中。

执行uuid()语句顺序相对于insert语句之前执行。



通过oracle的序列生成主键：

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

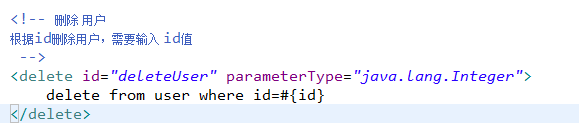
SELECT 序列名.nextval()

</selectKey>

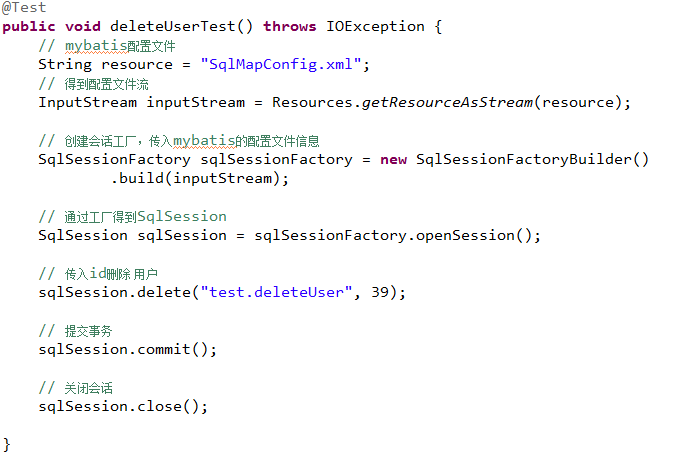
insert into user(id,username,birthday,sex,address) value(#{id},#{username},#{birthday},#{sex},#{address})

## 删除用户

### 映射文件

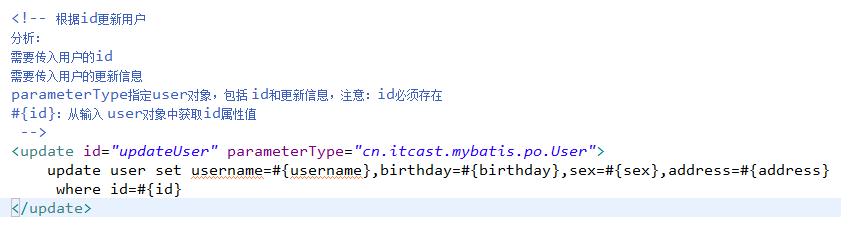


### 代码：



## 更新用户

### 映射文件



### 代码



## 总结

### parameterType

在映射文件中通过parameterType指定输入 参数的类型。

### resultType

在映射文件中通过resultType指定输出结果的类型。

### #{}和${}

#{}表示一个占位符号，#{}接收输入参数，类型可以是简单类型，pojo、hashmap。

如果接收简单类型，#{}中可以写成value或其它名称。

#{}接收pojo对象值，通过OGNL读取对象中的属性值，通过属性.属性.属性...的方式获取对象属性值。

${}表示一个拼接符号，会引用sql注入，所以不建议使用${}。

${}接收输入参数，类型可以是简单类型，pojo、hashmap。

如果接收简单类型，${}中只能写成value。

${}接收pojo对象值，通过OGNL读取对象中的属性值，通过属性.属性.属性...的方式获取对象属性值。

### selectOne和selectList

selectOne表示查询出一条记录进行映射。如果使用selectOne可以实现使用selectList也可以实现（list中只有一个对象）。

selectList表示查询出一个列表（多条记录）进行映射。如果使用selectList查询多条记录，不能使用selectOne。

如果使用selectOne报错：

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

## mybatis和hibernate本质区别和应用场景

hibernate：是一个标准ORM框架（对象关系映射）。入门门槛较高的，不需要程序写sql，sql语句自动生成了。

对sql语句进行优化、修改比较困难的。

应用场景：

适用与需求变化不多的中小型项目，比如：后台管理系统，erp、orm、oa。。

mybatis：专注是sql本身，需要程序员自己编写sql语句，sql修改、优化比较方便。mybatis是一个不完全 的ORM框架，虽然程序员自己写sql，mybatis 也可以实现映射（输入映射、输出映射）。

应用场景：

适用与需求变化较多的项目，比如：互联网项目。

企业进行技术选型，以低成本 高回报作为技术选型的原则，根据项目组的技术力量进行选择。

# mybatis开发dao的方法

## SqlSession使用范围

### SqlSessionFactoryBuilder

通过SqlSessionFactoryBuilder创建会话工厂SqlSessionFactory

将SqlSessionFactoryBuilder**当成一个工具类使用即可**，不需要使用单例管理SqlSessionFactoryBuilder。

在需要创建SqlSessionFactory时候，只需要new一次SqlSessionFactoryBuilder即可。

### SqlSessionFactory

通过SqlSessionFactory创建SqlSession，使用单例模式管理sqlSessionFactory（工厂一旦创建，使用一个实例）。

将来mybatis和spring整合后，使用单例模式管理sqlSessionFactory。

### SqlSession

SqlSession是一个面向用户（程序员）的接口。

SqlSession中提供了很多操作数据库的方法：如：selectOne(返回单个对象)、selectList（返回单个或多个对象）、。

SqlSession是线程不安全的，在SqlSesion实现类中除了有接口中的方法（操作数据库的方法）还有数据域属性。

**SqlSession最佳应用场合在方法体内，定义成局部变量使用。**

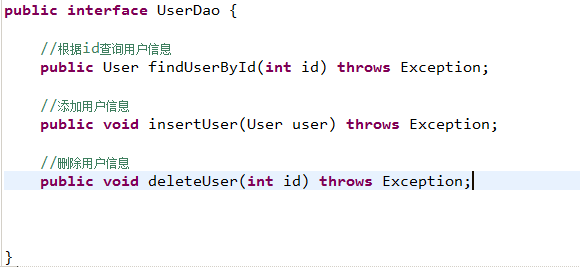
## 原始dao开发方法（程序员需要写dao接口和dao实现类）

### 思路

程序员需要写dao接口和dao实现类。

需要向dao实现类中注入SqlSessionFactory，在方法体内通过SqlSessionFactory创建SqlSession

### dao接口



### dao接口实现类

**public** **class** UserDaoImpl **implements** UserDao {

// 需要向dao实现类中注入SqlSessionFactory

// 这里通过构造方法注入

**private** SqlSessionFactory sqlSessionFactory;

**public** UserDaoImpl(SqlSessionFactory sqlSessionFactory) {

**this**.sqlSessionFactory = sqlSessionFactory;

}

@Override

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

SqlSession sqlSession = sqlSessionFactory.openSession();

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

// 释放资源

sqlSession.close();

**return** user;

}

@Override

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

SqlSession sqlSession = sqlSessionFactory.openSession();

//执行插入操作

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

// 提交事务

sqlSession.commit();

// 释放资源

sqlSession.close();

}

@Override

**public** **void** deleteUser(**int** id) **throws** Exception {

SqlSession sqlSession = sqlSessionFactory.openSession();

//执行插入操作

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

// 提交事务

sqlSession.commit();

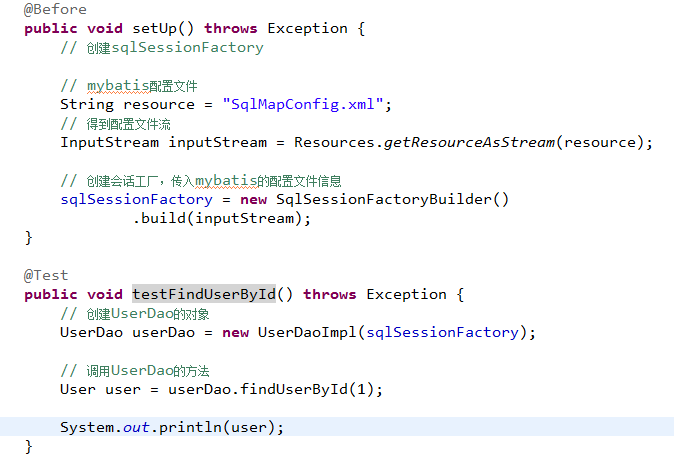
// 释放资源

sqlSession.close();

}

}

### 测试代码：



### 总结原始 dao开发问题

1、dao接口实现类方法中存在大量模板方法，设想能否将这些代码提取出来，大大减轻程序员的工作量。

2、调用sqlsession方法时将statement的id硬编码了

3、调用sqlsession方法时传入的变量，由于sqlsession方法使用泛型，即使变量类型传入错误，在编译阶段也不报错，不利于程序员开发。

## mapper代理方法（程序员只需要mapper接口（相当 于dao接口））

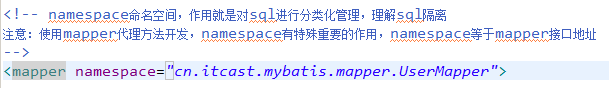
### 思路（mapper代理开发规范）

程序员还需要编写mapper.xml映射文件

程序员编写mapper接口需要遵循一些开发规范，mybatis可以自动生成mapper接口实现类代理对象。

开发规范：

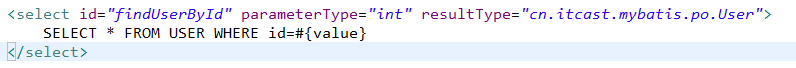
1、在mapper.xml中namespace等于mapper接口地址



2、mapper.java接口中的方法名和mapper.xml中statement的id一致

3、mapper.java接口中的方法输入参数类型和mapper.xml中statement的parameterType指定的类型一致。

4、mapper.java接口中的方法返回值类型和mapper.xml中statement的resultType指定的类型一致。





总结：

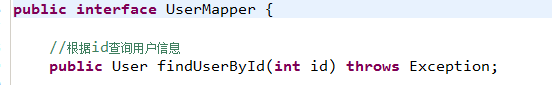
以上开发规范主要是对下边的代码进行统一生成：

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

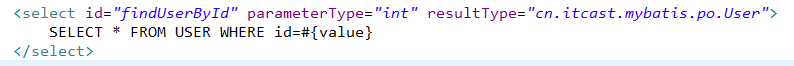
sqlSession.insert("test.insertUser", user);

。。。。

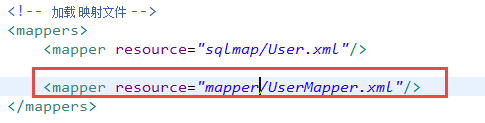
### mapper.java



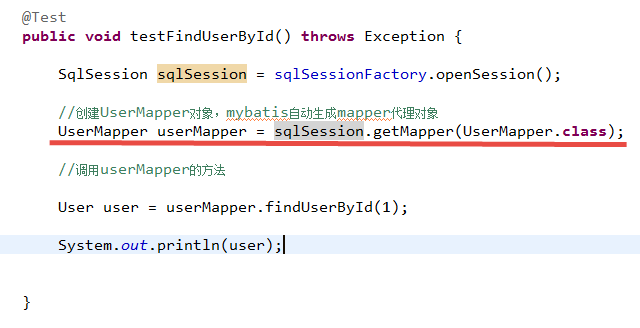
### mapper.xml



### 在SqlMapConfig.xml中加载mapper.xml



### 测试



### 一些问题总结

#### 代理对象内部调用selectOne或selectList

如果mapper方法返回单个pojo对象（非集合对象），代理对象内部通过selectOne查询数据库。

如果mapper方法返回集合对象，代理对象内部通过selectList查询数据库。

#### mapper接口方法参数只能有一个是否影响系统 开发

mapper接口方法参数只能有一个，系统是否不利于扩展维护。

**系统 框架中，dao层的代码是被业务层公用的。**

**即使mapper接口只有一个参数，可以使用包装类型的pojo满足不同的业务方法的需求。**

注意：持久层方法的参数可以包装类型、map。。。，service方法中建议不要使用包装类型（不利于业务层的可扩展）。

# 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）

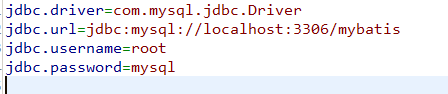
## properties属性

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

## 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。

详细参见“学习资料/mybatis-settings.xlsx”文件







## typeAliases（别名）重点

### 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。

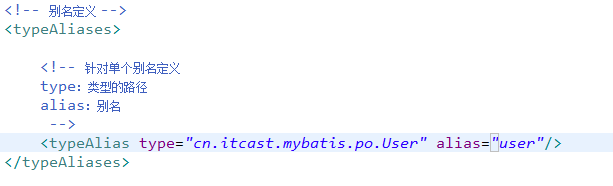
如果在指定类型时输入类型全路径，不方便进行开发，可以针对parameterType或resultType指定的类型定义一些别名，在mapper.xml中通过别名定义，方便开发。

### mybatis默认支持别名

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

### 自定义别名

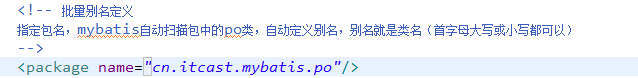
#### 单个别名定义



引用别名：



#### 批量定义别名（常用）



## typeHandlers（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）。 |

## mappers（映射配置）

### 通过resource加载单个映射文件



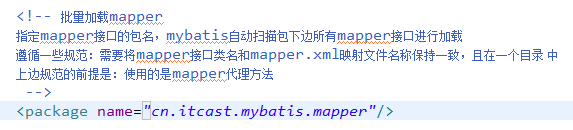
### 通过mapper接口加载单个mapper



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



### 批量加载mapper(推荐使用)



# 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

。

## 传递pojo的包装对象

### 需求

完成用户信息的综合查询，需要传入查询条件很复杂（可能包括用户信息、其它信息，比如商品、订单的）

### 定义包装类型pojo

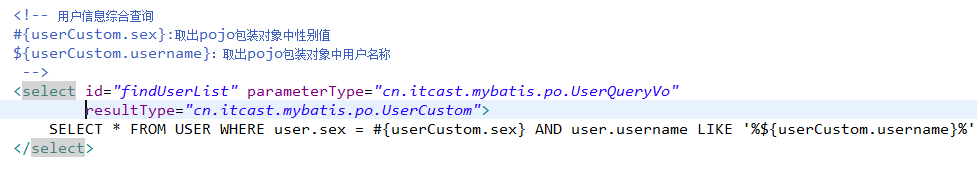
针对上边需求，建议使用自定义的包装类型的pojo。

在包装类型的pojo中将复杂的查询条件包装进去。



### mapper.xml

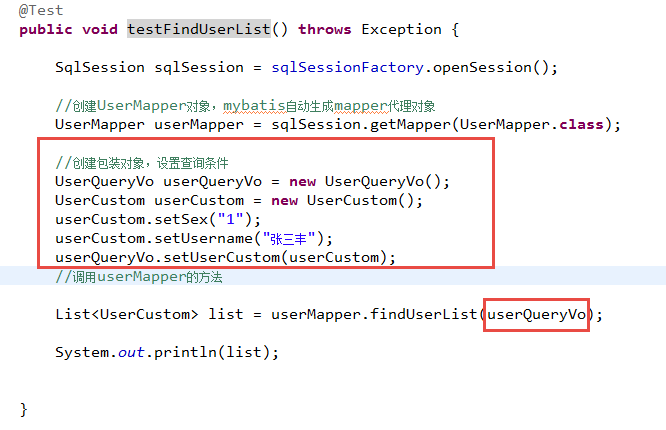
在UserMapper.xml中定义用户信息综合查询（查询条件复杂，通过高级查询进行复杂关联查询）。



### mapper.java



### 测试代码



# 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。

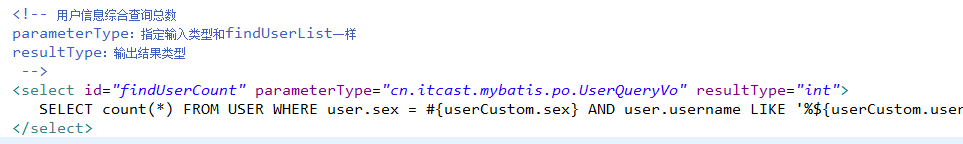
只要查询出来的列名和pojo中的属性有一个一致，就会创建pojo对象。

### 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。

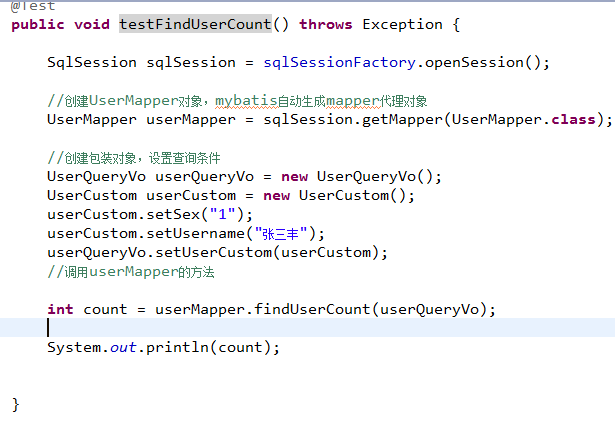
#### mapper.xml



#### mapper.java



#### 测试代码



#### 小结

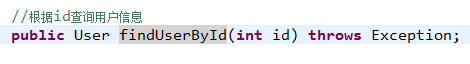
查询出来的结果集只有一行且一列，可以使用简单类型进行输出映射。

### 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

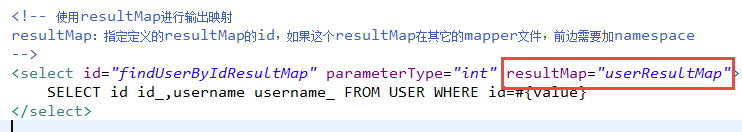
SELECT id id\_,username username\_ FROM USER WHERE id=#{value}

User类中属性名和上边查询列名不一致。

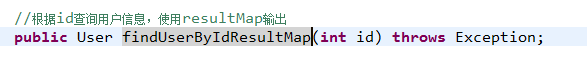
#### 定义reusltMap



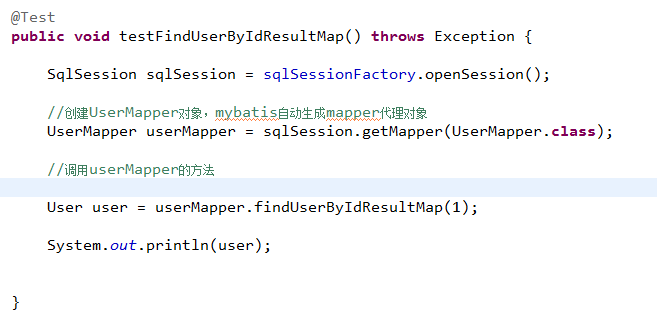
#### 使用resultMap作为statement的输出映射类型



#### 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。**

# 动态sql

## 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。

## 需求

用户信息综合查询列表和用户信息查询列表总数这两个statement的定义使用动态sql。

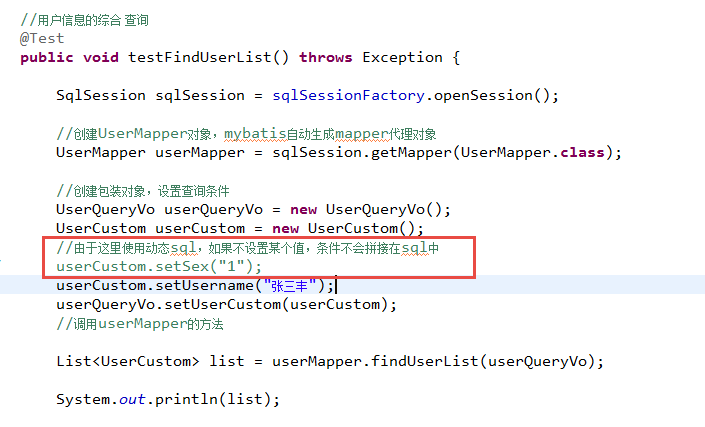
对查询条件进行判断，如果输入参数不为空才进行查询条件拼接。

## mapper.xml





## 测试代码



## sql片段

### 需求

将上边实现的动态sql判断代码块抽取出来，组成一个sql片段。其它的statement中就可以引用sql片段。

方便程序员进行开发。

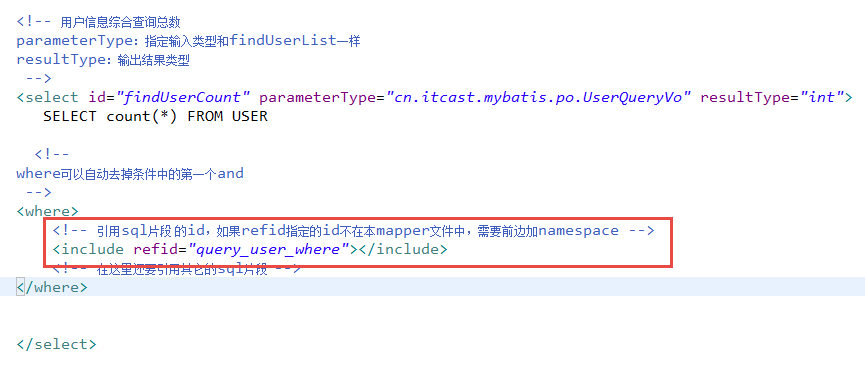
### 定义sql片段



### 引用sql片段

在mapper.xml中定义的statement中引用sql片段：





## foreach

向sql传递数组或List，mybatis使用foreach解析

### 需求

在用户查询列表和查询总数的statement中增加多个id输入查询。

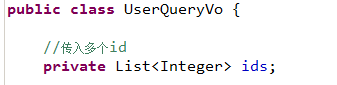
sql语句如下：

两种方法：

SELECT \* FROM USER WHERE id=1 OR id=10 OR id=16

SELECT \* FROM USER WHERE id IN(1,10,16)

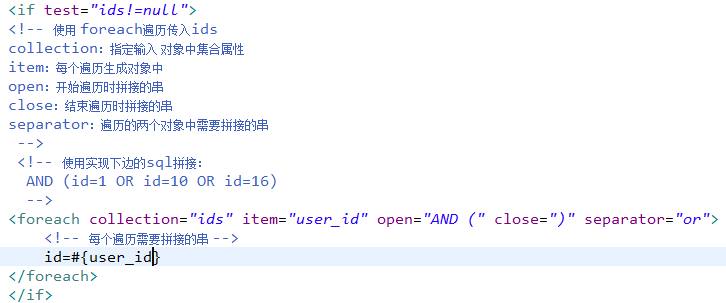
### 在输入参数类型中添加List<Integer> ids传入多个id



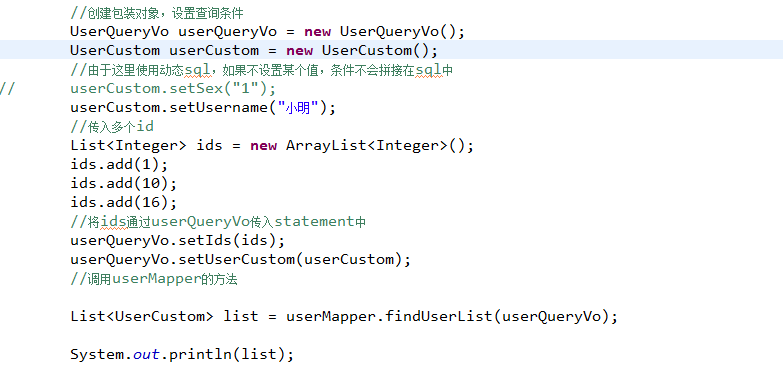
### 修改mapper.xml

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

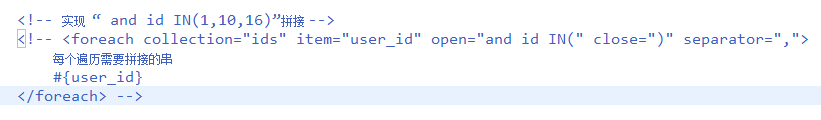
在查询条件中，查询条件定义成一个sql片段，需要修改sql片段。



### 测试代码



### 另外一个sql的实现：



mybatis第二天 高级映射 查询缓存 和spring整合

课程复习：

mybatis是什么？

mybatis是一人持久层框架，mybatis是一个不完全的ORM框架。sql语句需要程序员自己去编写，但是mybatis也有映射（输入参数映射、输出结果映射）。

mybatis入门门槛不高，学习成本低，让程序员把精力放在sql语句上，对sql语句优化非常方便，适用与需求变化较多项目，比如互联网项目。

mybatis框架执行过程：

1、配置mybatis的配置文件，SqlMapConfig.xml（名称不固定）

2、通过配置文件，加载mybatis运行环境，创建SqlSessionFactory会话工厂

SqlSessionFactory在实际使用时按单例方式。

3、通过SqlSessionFactory创建SqlSession

SqlSession是一个面向用户接口（提供操作数据库方法），实现对象是线程不安全的，建议sqlSession应用场合在方法体内。

4、调用sqlSession的方法去操作数据。

如果需要提交事务，需要执行SqlSession的commit()方法。

5、释放资源，关闭SqlSession

mybatis开发dao的方法：

1、原始dao 的方法

需要程序员编写dao接口和实现类

需要在dao实现类中注入一个SqlSessionFactory工厂。

2、mapper代理开发方法（建议使用）

只需要程序员编写mapper接口（就是dao接口）

程序员在编写mapper.xml(映射文件)和mapper.java需要遵循一个开发规范：

1、mapper.xml中namespace就是mapper.java的类全路径。

2、mapper.xml中statement的id和mapper.java中方法名一致。

3、mapper.xml中statement的parameterType指定输入参数的类型和mapper.java的方法输入 参数类型一致。

4、mapper.xml中statement的resultType指定输出结果的类型和mapper.java的方法返回值类型一致。

SqlMapConfig.xml配置文件：可以配置properties属性、别名、mapper加载。。。

输入映射：

parameterType：指定输入参数类型可以简单类型、pojo、hashmap。。

对于综合查询，建议parameterType使用包装的pojo，有利于系统 扩展。

输出映射：

resultType：

查询到的列名和resultType指定的pojo的属性名一致，才能映射成功。

reusltMap：

可以通过resultMap 完成一些高级映射。

如果查询到的列名和映射的pojo的属性名不一致时，通过resultMap设置列名和属性名之间的对应关系（映射关系）。可以完成映射。

高级映射：

将关联查询的列映射到一个pojo属性中。（一对一）

将关联查询的列映射到一个List<pojo>中。（一对多）

动态sql：（重点）

if判断（掌握）

where

foreach

sql片段（掌握）

课程安排：

对订单商品数据模型进行分析。

高级映射：（了解）

实现一对一查询、一对多、多对多查询。

延迟加载

查询缓存

一级缓存

二级缓存（了解mybatis二级缓存使用场景）

mybatis和spirng整合（掌握）

逆向工程（会用）

# 订单商品数据模型



## 数据模型分析思路

1、每张表记录的数据内容

分模块对每张表记录的内容进行熟悉，相当 于你学习系统 需求（功能）的过程。

2、每张表重要的字段设置

非空字段、外键字段

3、数据库级别表与表之间的关系

外键关系

4、表与表之间的业务关系

在分析表与表之间的业务关系时一定要建立 在某个业务意义基础上去分析。

## 数据模型分析

用户表：user

id：自增主键

订单表：orders

number：订单号

user\_id（外键，用户id）

订单明细表：orderdetail

orders\_id（外键，订单id）

items\_id（外键，商品id）

商品表：items

user\_id外键

orders\_id

外键

items\_id外键

一对一

一对多

一对一

一对多

一对一

一对多

orders--> items

一对多

items-->orders：

一对多

总之：orders和items是多对多关系

user和items是多对多关系

用户表user：

记录了购买商品的用户信息

订单表：orders

记录了用户所创建的订单（购买商品的订单）

订单明细表：orderdetail：

记录了订单的详细信息即购买商品的信息

商品表：items

记录了商品信息

表与表之间的业务关系：

在分析表与表之间的业务关系时需要建立 在某个业务意义基础上去分析。

先分析数据级别之间有关系的表之间的业务关系：

usre和orders：

user---->orders：一个用户可以创建多个订单，一对多

orders--->user：一个订单只由一个用户创建，一对一

orders和orderdetail：

orders---》orderdetail：一个订单可以包括 多个订单明细，因为一个订单可以购买多个商品，每个商品的购买信息在orderdetail记录，一对多关系

orderdetail--> orders：一个订单明细只能包括在一个订单中，一对一

orderdetail和itesm：

orderdetail---》itesms：一个订单明细只对应一个商品信息，一对一

items--> orderdetail:一个商品可以包括在多个订单明细 ，一对多

再分析数据库级别没有关系的表之间是否有业务关系：

orders和items：

orders和items之间可以通过orderdetail表建立 关系。

# 一对一查询

## 需求

查询订单信息，关联查询创建订单的用户信息

## resultType

### sql语句

确定查询的主表：订单表

确定查询的关联表：用户表

关联查询使用内链接？还是外链接？

由于orders表中有一个外键（user\_id），通过外键关联查询用户表只能查询出一条记录，可以使用内链接。

SELECT

orders.\*,

USER.username,

USER.sex,

USER.address

FROM

orders,

USER

WHERE orders.user\_id = user.id

### 创建pojo

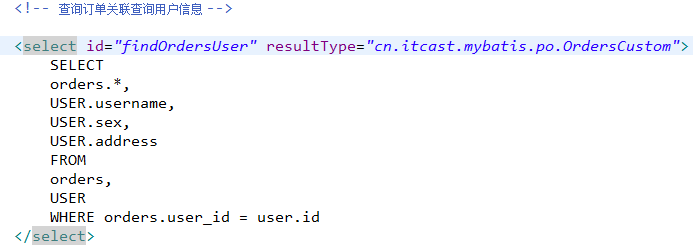
将上边sql查询的结果映射到pojo中，pojo中必须包括所有查询列名。

原始的Orders.java不能映射全部字段，需要新创建的pojo。

创建 一个pojo继承包括查询字段较多的po类。



### mapper.xml



### mapper.java



## resultMap

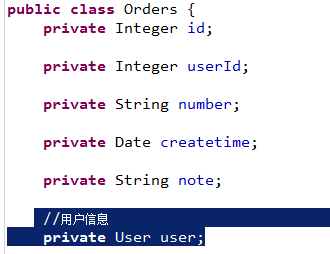
### sql语句

同resultType实现的sql

### 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。

### 需要Orders类中添加user属性



### mapper.xml

#### 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"*/>

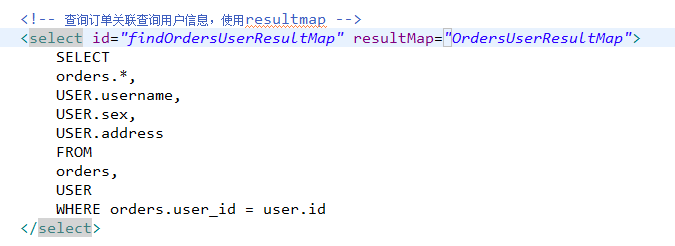
<result column=*"sex"* property=*"sex"*/>

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

</association>

</resultMap>

#### statement定义



### mapper.java



## resultType和resultMap实现一对一查询小结

实现一对一查询：

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。

# 一对多查询

## 需求

查询订单及订单明细的信息。

## 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,

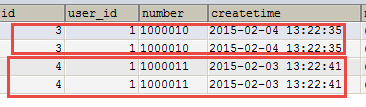
USER,

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。

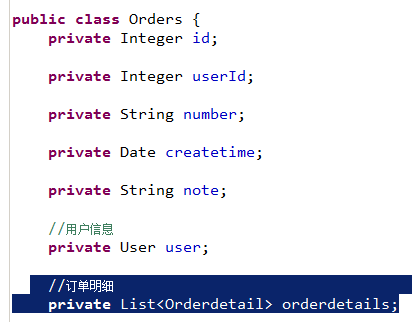
最终会将订单信息映射到orders中，订单所对应的订单明细映射到orders中的orderDetails属性中。



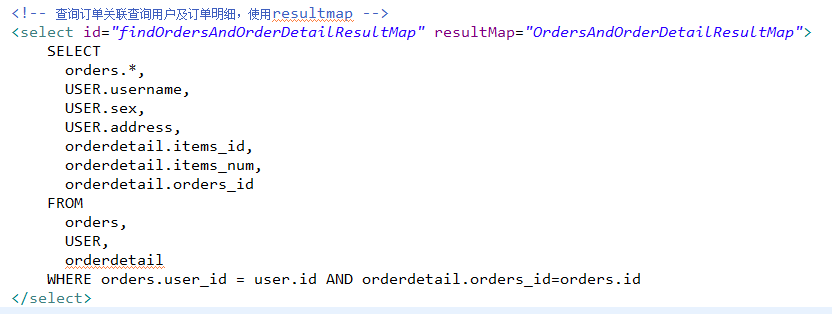
映射成的orders记录数为两条（orders信息不重复）

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，不用在中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，不用在中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。

# 多对多查询

## 需求

查询用户及用户购买商品信息。

## sql语句

查询主表是：用户表

关联表：由于用户和商品没有直接关联，通过订单和订单明细进行关联，所以关联表：

orders、orderdetail、items

SELECT

orders.\*,

USER.username,

USER.sex,

USER.address,

orderdetail.id orderdetail\_id,

orderdetail.items\_id,

orderdetail.items\_num,

orderdetail.orders\_id,

items.name items\_name,

items.detail items\_detail,

items.price items\_price

FROM

orders,

USER,

orderdetail,

items

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

## 映射思路

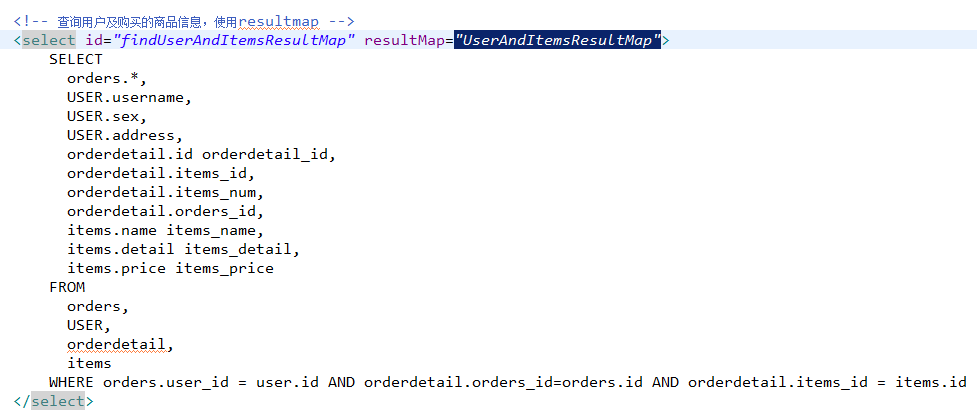
将用户信息映射到user中。

在user类中添加订单列表属性List<Orders> orderslist，将用户创建的订单映射到orderslist

在Orders中添加订单明细列表属性List<OrderDetail>orderdetials，将订单的明细映射到orderdetials

在OrderDetail中添加Items属性，将订单明细所对应的商品映射到Items

## mapper.xml



## resultMap定义

<!-- 查询用户及购买的商品 -->

<resultMap type=*"cn.itcast.mybatis.po.User"* id=*"UserAndItemsResultMap"*>

<!-- 用户信息 -->

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

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

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

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

<!-- 订单信息

一个用户对应多个订单，使用collection映射

-->

<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

一个订单包括 多个明细

-->

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

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

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

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

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

<!-- 商品信息

一个订单明细对应一个商品

-->

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

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

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

<result column=*"items\_detail"* property=*"detail"*/>

<result column=*"items\_price"* property=*"price"*/>

</association>

</collection>

</collection>

</resultMap>

## mapper.java



## 多对多查询总结

将查询用户购买的商品信息明细清单，（用户名、用户地址、购买商品名称、购买商品时间、购买商品数量）

针对上边的需求就使用resultType将查询到的记录映射到一个扩展的pojo中，很简单实现明细清单的功能。

一对多是多对多的特例，如下需求：

查询用户购买的商品信息，用户和商品的关系是多对多关系。

需求1：

查询字段：用户账号、用户名称、用户性别、商品名称、商品价格(最常见)

企业开发中常见明细列表，用户购买商品明细列表，

使用resultType将上边查询列映射到pojo输出。

需求2：

查询字段：用户账号、用户名称、购买商品数量、商品明细（鼠标移上显示明细）

使用resultMap将用户购买的商品明细列表映射到user对象中。

总结：

使用resultMap是针对那些对查询结果映射有特殊要求的功能，，比如特殊要求映射成list中包括 多个list。

# resultMap总结

resultType：

作用：

将查询结果按照sql列名pojo属性名一致性映射到pojo中。

场合：

常见一些明细记录的展示，比如用户购买商品明细，将关联查询信息全部展示在页面时，此时可直接使用resultType将每一条记录映射到pojo中，在前端页面遍历list（list中是pojo）即可。

resultMap：

使用association和collection完成一对一和一对多高级映射（对结果有特殊的映射要求）。

association：

作用：

将关联查询信息映射到一个pojo对象中。

场合：

为了方便查询关联信息可以使用association将关联订单信息映射为用户对象的pojo属性中，比如：查询订单及关联用户信息。

使用resultType无法将查询结果映射到pojo对象的pojo属性中，根据对结果集查询遍历的需要选择使用resultType还是resultMap。

collection：

作用：

将关联查询信息映射到一个list集合中。

场合：

为了方便查询遍历关联信息可以使用collection将关联信息映射到list集合中，比如：查询用户权限范围模块及模块下的菜单，可使用collection将模块映射到模块list中，将菜单列表映射到模块对象的菜单list属性中，这样的作的目的也是方便对查询结果集进行遍历查询。

如果使用resultType无法将查询结果映射到list集合中。

# 延迟加载

## 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。

## 使用association实现延迟加载

### 需求

查询订单并且关联查询用户信息

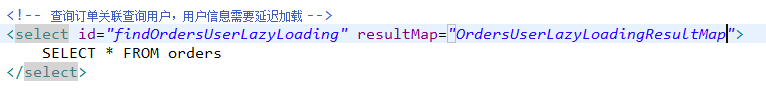
### mapper.xml

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。

### 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>

### mapper.java



### 测试

#### 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。

#### 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：

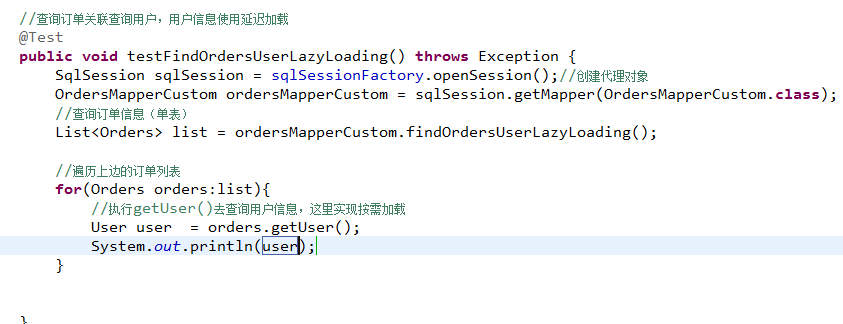
lazyLoadingEnabled、aggressiveLazyLoading

|  |  |  |  |
| --- | --- | --- | --- |
| 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：



#### 测试代码



### 延迟加载思考

不使用mybatis提供的association及collection中的延迟加载功能，如何实现延迟加载？？

实现方法如下：

定义两个mapper方法：

1、查询订单列表

2、根据用户id查询用户信息

实现思路：

先去查询第一个mapper方法，获取订单信息列表

在程序中（service），按需去调用第二个mapper方法去查询用户信息。

总之：

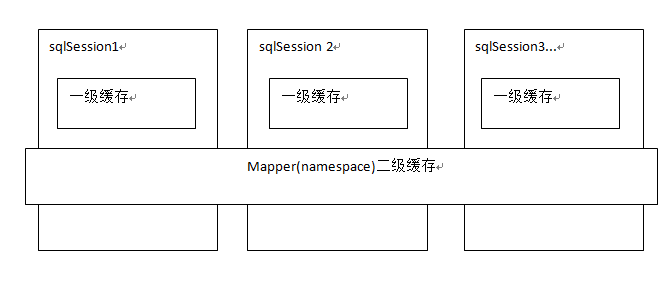
使用延迟加载方法，先去查询**简单的sql（**最好单表，也可以关联查询），再去按需要加载关联查询的其它信息。

# 查询缓存

## 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。

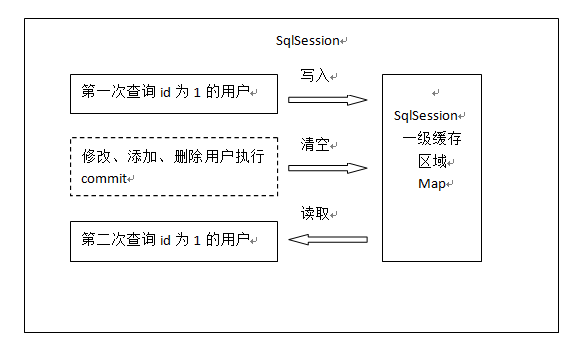
二级缓存是mapper级别的缓存，多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的。

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。

## 一级缓存

### 一级缓存工作原理



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。

### 一级缓存测试

mybatis默认支持一级缓存，不需要在配置文件去配置。

按照上边一级缓存原理步骤去测试。

@Test

**public** **void** testCache1() **throws** Exception{

SqlSession sqlSession = sqlSessionFactory.openSession();//创建代理对象

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

//下边查询使用一个SqlSession

//第一次发起请求，查询id为1的用户

User user1 = userMapper.findUserById(1);

System.*out*.println(user1);

// 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。

//更新user1的信息

user1.setUsername("测试用户22");

userMapper.updateUser(user1);

//执行commit操作去清空缓存

sqlSession.commit();

//第二次发起请求，查询id为1的用户

User user2 = userMapper.findUserById(1);

System.*out*.println(user2);

sqlSession.close();

}

### 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。

## 二级缓存

### 原理



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。

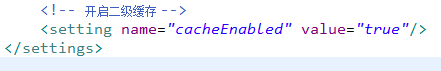
### 开启二级缓存

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。

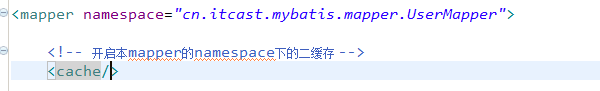
在核心配置文件SqlMapConfig.xml中加入

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

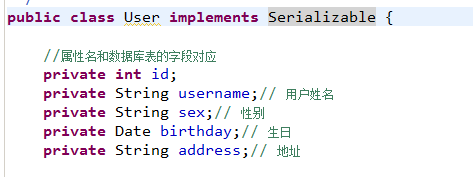
|  |  |  |  |
| --- | --- | --- | --- |
|  | 描述 | 允许值 | 默认值 |
| cacheEnabled | 对在此配置文件下的所有cache 进行全局性开/关设置。 | true false | true |



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）。



### 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。

### 测试方法

// 二级缓存测试

@Test

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

SqlSession sqlSession1 = sqlSessionFactory.openSession();

SqlSession sqlSession2 = sqlSessionFactory.openSession();

SqlSession sqlSession3 = sqlSessionFactory.openSession();

// 创建代理对象

UserMapper userMapper1 = sqlSession1.getMapper(UserMapper.**class**);

// 第一次发起请求，查询id为1的用户

User user1 = userMapper1.findUserById(1);

System.*out*.println(user1);

//这里执行关闭操作，将sqlsession中的数据写到二级缓存区域

sqlSession1.close();

//使用sqlSession3执行commit()操作

UserMapper userMapper3 = sqlSession3.getMapper(UserMapper.**class**);

User user = userMapper3.findUserById(1);

user.setUsername("张明明");

userMapper3.updateUser(user);

//执行提交，清空UserMapper下边的二级缓存

sqlSession3.commit();

sqlSession3.close();

UserMapper userMapper2 = sqlSession2.getMapper(UserMapper.**class**);

// 第二次发起请求，查询id为1的用户

User user2 = userMapper2.findUserById(1);

System.*out*.println(user2);

sqlSession2.close();

}

### 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。

### 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。

## mybatis整合ehcache

ehcache是一个分布式缓存框架。

### 分布缓存

我们系统为了提高系统并发，性能、一般对系统进行分布式部署（集群部署方式）

系统工程

springmvc

mybatis

服务器1

系统工程

springmvc

mybatis

服务器2

对缓存数据进行集中管理（redis集群）

使用分布式缓存框架

redis、memcached、ehcache。。

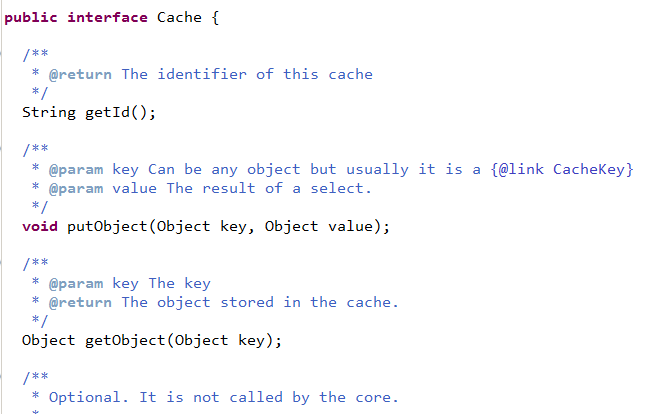
不使用分布缓存，缓存的数据在各各服务单独存储，不方便系统 开发。所以要使用分布式缓存对缓存数据进行集中管理。

mybatis无法实现分布式缓存，需要和其它分布式缓存框架进行整合。

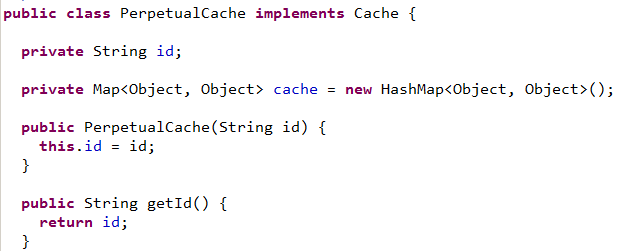
### 整合方法(掌握)

mybatis提供了一个cache接口，如果要实现自己的缓存逻辑，实现cache接口开发即可。

mybatis和ehcache整合，mybatis和ehcache整合包中提供了一个cache接口的实现类。



mybatis默认实现cache类是：

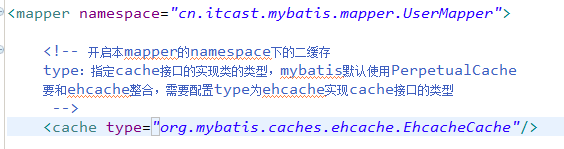


### 加入ehcache包



### 整合ehcache

配置mapper中cache中的type为ehcache对cache接口的实现类型。



### 加入ehcache的配置文件

在classpath下配置ehcache.xml



## 二级应用场景

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。

实现方法如下：通过设置刷新间隔时间，由mybatis每隔一段时间自动清空缓存，根据数据变化频率设置缓存刷新间隔flushInterval，比如设置为30分钟、60分钟、24小时等，根据需求而定。

## 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。

# spring和mybatis整合

## 整合思路

需要spring通过单例方式管理SqlSessionFactory。

spring和mybatis整合生成代理对象，使用SqlSessionFactory创建SqlSession。（spring和mybatis整合自动完成）

持久层的mapper都需要由spring进行管理。

## 整合环境

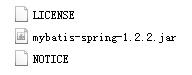
创建一个新的java工程（接近实际开发的工程结构）

jar包：

mybatis3.2.7的jar包

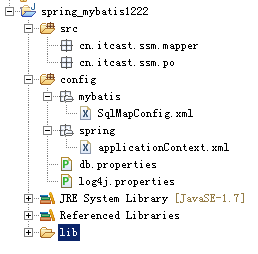
spring3.2.0的jar包

mybatis和spring的整合包：早期ibatis和spring整合是由spring官方提供，mybatis和spring整合由mybatis提供。



全部jar包





## sqlSessionFactory

在applicationContext.xml配置sqlSessionFactory和数据源

sqlSessionFactory在mybatis和spring的整合包下。

<!-- 加载配置文件 -->

<context:property-placeholder location=*"classpath:db.properties"* />

<!-- 数据源，使用dbcp -->

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*

destroy-method=*"close"*>

<property name=*"driverClassName"* value=*"${jdbc.driver}"* />

<property name=*"url"* value=*"${jdbc.url}"* />

<property name=*"username"* value=*"${jdbc.username}"* />

<property name=*"password"* value=*"${jdbc.password}"* />

<property name=*"maxActive"* value=*"10"* />

<property name=*"maxIdle"* value=*"5"* />

</bean>

<!-- sqlSessinFactory -->

<bean id=*"sqlSessionFactory"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>

<!-- 加载mybatis的配置文件 -->

<property name=*"configLocation"* value=*"mybatis/SqlMapConfig.xml"* />

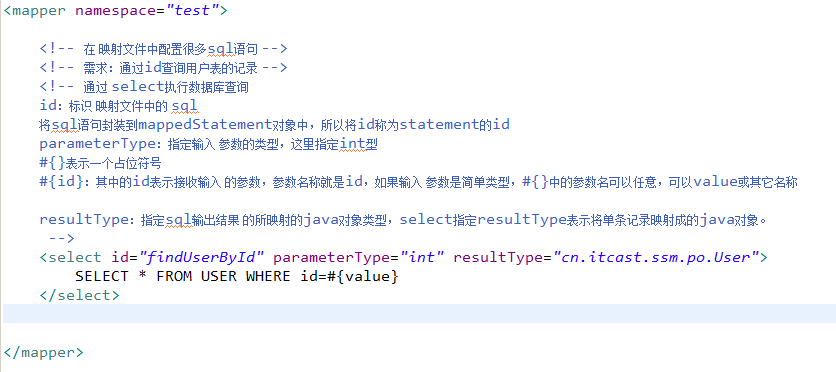
<!-- 数据源 -->

<property name=*"dataSource"* ref=*"dataSource"* />

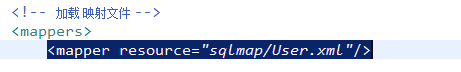
</bean>

## 原始dao开发（和spring整合后）

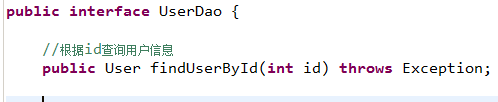
### User.xml



在SqlMapconfig.xml中加载User.xml



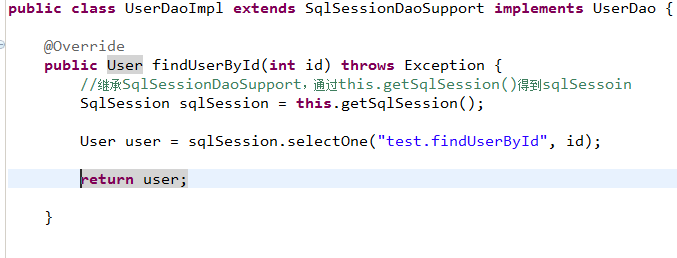
### dao(实现类继承SqlSessionDaoSupport)



dao接口实现类需要注入SqlSessoinFactory，通过spring进行注入。

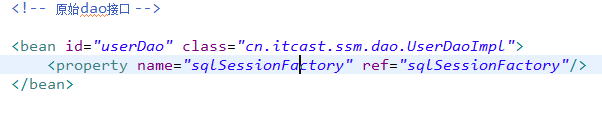
这里spring声明配置方式，配置dao的bean：

**让UserDaoImpl实现类继承SqlSessionDaoSupport**

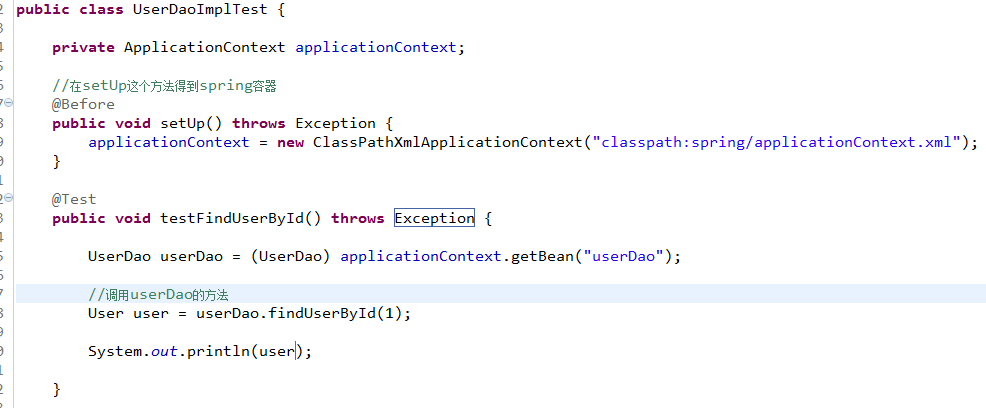


### 配置dao

在applicationContext.xml中配置dao。



### 测试程序

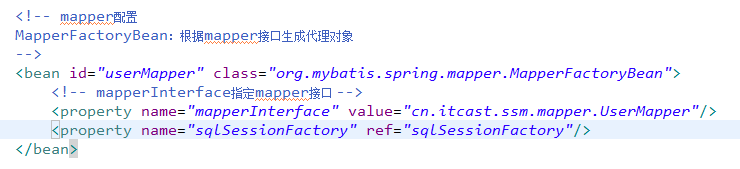


## mapper代理开发

### mapper.xml和mapper.java



### 通过MapperFactoryBean创建代理对象



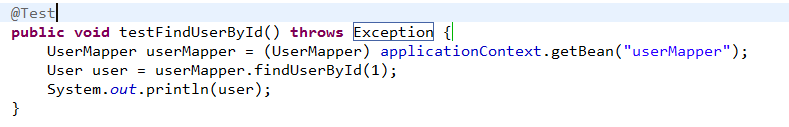
此方法问题：

需要针对每个mapper进行配置，麻烦。

### 通过MapperScannerConfigurer进行mapper扫描（建议使用）



### 测试代码



# 逆向工程

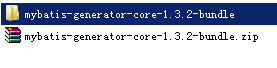
## 什么是逆向工程

mybaits需要程序员自己编写sql语句，mybatis官方提供逆向工程 可以针对单表自动生成mybatis执行所需要的代码（mapper.java,mapper.xml、po..）

企业实际开发中，常用的逆向工程方式：

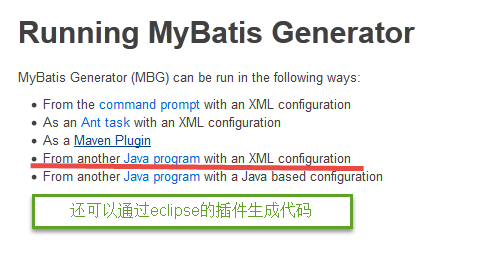
由于数据库的表生成java代码。

## 下载逆向工程



## 使用方法（会用）

### 运行逆向工程



建议使用java程序方式，不依赖开发工具。

### 生成代码配置文件

<?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.ssm.po"*

targetProject=*".\src"*>

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

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

<!-- 从数据库返回的值被清理前后的空格 -->

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

</javaModelGenerator>

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

<sqlMapGenerator targetPackage=*"cn.itcast.ssm.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.ssm.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>

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

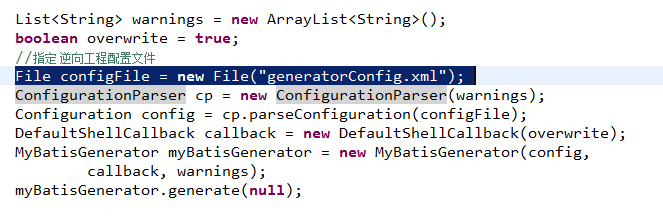
<table tableName=*"orderdetail"*></table>

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

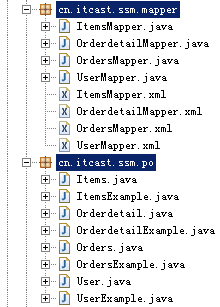
</context>

</generatorConfiguration>

### 执行生成程序



生成后的代码：



### 使用生成的代码

需要将生成工程中所生成的代码拷贝到自己的工程中。

测试ItemsMapper中的方法

//自定义条件查询

@Test

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

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

//通过criteria构造查询条件

ItemsExample.Criteria criteria = itemsExample.createCriteria();

criteria.andNameEqualTo("笔记本3");

//可能返回多条记录

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);

}

//插入

@Test

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

//构造 items对象

Items items = **new** Items();

items.setName("手机");

items.setPrice(999f);

itemsMapper.insert(items);

}

//更新数据

@Test

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

//对所有字段进行更新，需要先查询出来再更新

Items items = itemsMapper.selectByPrimaryKey(1);

items.setName("水杯");

itemsMapper.updateByPrimaryKey(items);

//如果传入字段不空为才更新，在批量更新中使用此方法，不需要先查询再更新

//itemsMapper.updateByPrimaryKeySelective(record);

}

# 插件（拦截器）

MyBatis提供了一种插件(plugin)的功能，虽然叫做插件，但其实这是拦截器功能。那么拦截器拦截MyBatis中的哪些内容呢？

MyBatis 允许你在已映射语句执行过程中的某一点进行拦截调用。默认情况下，MyBatis允许使用插件来拦截的方法调用包括：

1. Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed) 拦截执行器的方法
2. ParameterHandler (getParameterObject, setParameters) 拦截参数的处理
3. ResultSetHandler (handleResultSets, handleOutputParameters) 拦截结果集的处理
4. StatementHandler (prepare, parameterize, batch, update, query) 拦截Sql语法构建的处理

Mybatis采用责任链模式，通过动态代理组织多个拦截器（插件），通过这些拦截器可以改变Mybatis的默认行为（诸如SQL重写之类的），由于插件会深入到Mybatis的核心，因此在编写自己的插件前最好了解下它的原理，以便写出安全高效的插件。

#1 拦截器的使用# ##1.1 拦截器介绍及配置## 首先我们看下MyBatis拦截器的接口定义：

public interface Interceptor {

Object intercept(Invocation invocation) throws Throwable;

Object plugin(Object target);

void setProperties(Properties properties);

}

比较简单，只有3个方法。 MyBatis默认没有一个拦截器接口的实现类，开发者们可以实现符合自己需求的拦截器。下面的MyBatis官网的一个拦截器实例：

@Intercepts({@Signature(type= Executor.class, method = "update", args = {MappedStatement.class,Object.class})})

public class ExamplePlugin implements Interceptor {

public Object intercept(Invocation invocation) throws Throwable {

return invocation.proceed();

}

public Object plugin(Object target) {

return Plugin.wrap(target, this);

}

public void setProperties(Properties properties) {

}

}

全局xml配置：

<plugins>

<plugin interceptor="org.format.mybatis.cache.interceptor.ExamplePlugin"></plugin></plugins>

这个拦截器拦截Executor接口的update方法（其实也就是SqlSession的新增，删除，修改操作），所有执行executor的update方法都会被该拦截器拦截到。 ##1.2 源码分析## **首先从源头->配置文件开始分析：**

1. XMLConfigBuilder解析MyBatis全局配置文件的pluginElement私有方法：

private void pluginElement(XNode parent) throws Exception {

if (parent != null) {

for (XNode child : parent.getChildren()) {

String interceptor = child.getStringAttribute("interceptor");

Properties properties = child.getChildrenAsProperties();

Interceptor interceptorInstance = (Interceptor) resolveClass(interceptor).newInstance();

interceptorInstance.setProperties(properties);

configuration.addInterceptor(interceptorInstance);

}

}

}

1. 具体的解析代码其实比较简单，就不贴了，主要就是通过反射实例化plugin节点中的interceptor属性表示的类。然后调用全局配置类Configuration的addInterceptor方法。

public void addInterceptor(Interceptor interceptor) {

interceptorChain.addInterceptor(interceptor);

}

1. 这个interceptorChain是Configuration的内部属性，类型为InterceptorChain，也就是一个拦截器链，我们来看下它的定义：

public class InterceptorChain {

private final List<Interceptor> interceptors = new ArrayList<Interceptor>();

public Object pluginAll(Object target) {

for (Interceptor interceptor : interceptors) {

target = interceptor.plugin(target);

}

return target;

}

public void addInterceptor(Interceptor interceptor) {

interceptors.add(interceptor);

}

public List<Interceptor> getInterceptors() {

return Collections.unmodifiableList(interceptors);

}

}

1. 现在我们理解了拦截器配置的解析以及拦截器的归属，现在我们回过头看下为何拦截器会拦截这些方法（Executor，ParameterHandler，ResultSetHandler，StatementHandler的部分方法）：

public ParameterHandler newParameterHandler(MappedStatement mappedStatement, Object parameterObject, BoundSql boundSql) {

ParameterHandler parameterHandler = mappedStatement.getLang().createParameterHandler(mappedStatement, parameterObject, boundSql);

parameterHandler = (ParameterHandler) interceptorChain.pluginAll(parameterHandler);

return parameterHandler;

}

public ResultSetHandler newResultSetHandler(Executor executor, MappedStatement mappedStatement, RowBounds rowBounds, ParameterHandler parameterHandler, ResultHandler resultHandler, BoundSql boundSql) {

ResultSetHandler resultSetHandler = new DefaultResultSetHandler(executor, mappedStatement, parameterHandler, resultHandler, boundSql, rowBounds);

resultSetHandler = (ResultSetHandler) interceptorChain.pluginAll(resultSetHandler);

return resultSetHandler;

}

public StatementHandler newStatementHandler(Executor executor, MappedStatement mappedStatement, Object parameterObject, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) {

StatementHandler statementHandler = new RoutingStatementHandler(executor, mappedStatement, parameterObject, rowBounds, resultHandler, boundSql);

statementHandler = (StatementHandler) interceptorChain.pluginAll(statementHandler);

return statementHandler;

}

public Executor newExecutor(Transaction transaction, ExecutorType executorType, boolean autoCommit) {

executorType = executorType == null ? defaultExecutorType : executorType;

executorType = executorType == null ? ExecutorType.SIMPLE : executorType;

Executor executor;

if (ExecutorType.BATCH == executorType) {

executor = new BatchExecutor(this, transaction);

} else if (ExecutorType.REUSE == executorType) {

executor = new ReuseExecutor(this, transaction);

} else {

executor = new SimpleExecutor(this, transaction);

}

if (cacheEnabled) {

executor = new CachingExecutor(executor, autoCommit);

}

executor = (Executor) interceptorChain.pluginAll(executor);

return executor;

}

以上4个方法都是Configuration的方法。这些方法在MyBatis的一个操作(新增，删除，修改，查询)中都会被执行到，执行的先后顺序是Executor，ParameterHandler，ResultSetHandler，StatementHandler(其中ParameterHandler和ResultSetHandler的创建是在创建StatementHandler[3个可用的实现类CallableStatementHandler,PreparedStatementHandler,SimpleStatementHandler]的时候，其构造函数调用的[这3个实现类的构造函数其实都调用了父类BaseStatementHandler的构造函数])。

这4个方法实例化了对应的对象之后，都会调用interceptorChain的pluginAll方法，InterceptorChain的pluginAll刚才已经介绍过了，就是遍历所有的拦截器，然后调用各个拦截器的plugin方法。注意：拦截器的plugin方法的返回值会直接被赋值给原先的对象。

由于可以拦截StatementHandler，这个接口主要处理sql语法的构建，因此比如分页的功能，可以用拦截器实现，只需要在拦截器的plugin方法中处理StatementHandler接口实现类中的sql即可，可使用反射实现。

MyBatis还提供了@Intercepts和 @Signature关于拦截器的注解。官网的例子就是使用了这2个注解，还包括了Plugin类的使用：

@Overridepublic Object plugin(Object target) {

return Plugin.wrap(target, this);

}

#2 代理链的生成# Mybatis支持对Executor、StatementHandler、ParameterHandler和ResultSetHandler进行拦截，也就是说会对这4种对象进行代理。通过查看Configuration类的源代码我们可以看到，每次都对目标对象进行代理链的生成。

下面以Executor为例。Mybatis在创建Executor对象时会执行下面一行代码：

executor =(Executor) interceptorChain.pluginAll(executor);

InterceptorChain里保存了所有的拦截器，它在mybatis初始化的时候创建。上面这句代码的含义是调用拦截器链里的每个拦截器依次对executor进行plugin（插入？）代码如下：

/\*\*

\* 每一个拦截器对目标类都进行一次代理

\* @param target

\* @return 层层代理后的对象

\*/

public Object pluginAll(Object target) {

for(Interceptor interceptor : interceptors) {

target= interceptor.plugin(target);

}

return target;

}

下面以一个简单的例子来看看这个plugin方法里到底发生了什么：

@Intercepts({@Signature(type = Executor.class, method ="update", args = {MappedStatement.class, Object.class})})

public class ExamplePlugin implements Interceptor {

@Override

public Object intercept(Invocation invocation) throws Throwable {

return invocation.proceed();

}

@Override

public Object plugin(Object target) {

return Plugin.wrap(target, this);

}

@Override

public void setProperties(Properties properties) {

}

}

**每一个拦截器都必须实现上面的三个方法，其中：**

Object intercept(Invocation invocation)是实现拦截逻辑的地方，内部要通过invocation.proceed()显式地推进责任链前进，也就是调用下一个拦截器拦截目标方法。

Object plugin(Object target)就是用当前这个拦截器生成对目标target的代理，实际是通过Plugin.wrap(target,this)来完成的，把目标target和拦截器this传给了包装函数。

setProperties(Properties properties)用于设置额外的参数，参数配置在拦截器的Properties节点里。

注解里描述的是指定拦截方法的签名 [type,method,args] （即对哪种对象的哪种方法进行拦截），它在拦截前用于决断。

定义自己的Interceptor最重要的是要实现plugin方法和intercept方法，在plugin方法中我们可以决定是否要进行拦截进而决定要返回一个什么样的目标对象。而intercept方法就是要进行拦截的时候要执行的方法。

对于plugin方法而言，其实Mybatis已经为我们提供了一个实现。Mybatis中有一个叫做Plugin的类，里面有一个静态方法wrap(Object target,Interceptor interceptor)，通过该方法可以决定要返回的对象是目标对象还是对应的代理。这里我们先来看一下Plugin的源码：

package org.apache.ibatis.plugin;

import java.lang.reflect.InvocationHandler;import java.lang.reflect.Method;import java.lang.reflect.Proxy;import java.util.HashMap;import java.util.HashSet;import java.util.Map;import java.util.Set;

import org.apache.ibatis.reflection.ExceptionUtil;

//这个类是Mybatis拦截器的核心,大家可以看到该类继承了InvocationHandler//又是JDK动态代理机制public class Plugin implements InvocationHandler {

//目标对象

private Object target;

//拦截器

private Interceptor interceptor;

//记录需要被拦截的类与方法

private Map<Class<?>, Set<Method>> signatureMap;

private Plugin(Object target, Interceptor interceptor, Map<Class<?>, Set<Method>> signatureMap) {

this.target = target;

this.interceptor = interceptor;

this.signatureMap = signatureMap;

}

//一个静态方法,对一个目标对象进行包装，生成代理类。

public static Object wrap(Object target, Interceptor interceptor) {

//首先根据interceptor上面定义的注解 获取需要拦截的信息

Map<Class<?>, Set<Method>> signatureMap = getSignatureMap(interceptor);

//目标对象的Class

Class<?> type = target.getClass();

//返回需要拦截的接口信息

Class<?>[] interfaces = getAllInterfaces(type, signatureMap);

//如果长度为>0 则返回代理类 否则不做处理

if (interfaces.length > 0) {

return Proxy.newProxyInstance(

type.getClassLoader(),

interfaces,

new Plugin(target, interceptor, signatureMap));

}

return target;

}

//代理对象每次调用的方法

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

try {

//通过method参数定义的类 去signatureMap当中查询需要拦截的方法集合

Set<Method> methods = signatureMap.get(method.getDeclaringClass());

//判断是否需要拦截

if (methods != null && methods.contains(method)) {

return interceptor.intercept(new Invocation(target, method, args));

}

//不拦截 直接通过目标对象调用方法

return method.invoke(target, args);

} catch (Exception e) {

throw ExceptionUtil.unwrapThrowable(e);

}

}

//根据拦截器接口（Interceptor）实现类上面的注解获取相关信息

private static Map<Class<?>, Set<Method>> getSignatureMap(Interceptor interceptor) {

//获取注解信息

Intercepts interceptsAnnotation = interceptor.getClass().getAnnotation(Intercepts.class);

//为空则抛出异常

if (interceptsAnnotation == null) { // issue #251

throw new PluginException("No @Intercepts annotation was found in interceptor " + interceptor.getClass().getName());

}

//获得Signature注解信息

Signature[] sigs = interceptsAnnotation.value();

Map<Class<?>, Set<Method>> signatureMap = new HashMap<Class<?>, Set<Method>>();

//循环注解信息

for (Signature sig : sigs) {

//根据Signature注解定义的type信息去signatureMap当中查询需要拦截方法的集合

Set<Method> methods = signatureMap.get(sig.type());

//第一次肯定为null 就创建一个并放入signatureMap

if (methods == null) {

methods = new HashSet<Method>();

signatureMap.put(sig.type(), methods);

}

try {

//找到sig.type当中定义的方法 并加入到集合

Method method = sig.type().getMethod(sig.method(), sig.args());

methods.add(method);

} catch (NoSuchMethodException e) {

throw new PluginException("Could not find method on " + sig.type() + " named " + sig.method() + ". Cause: " + e, e);

}

}

return signatureMap;

}

//根据对象类型与signatureMap获取接口信息

private static Class<?>[] getAllInterfaces(Class<?> type, Map<Class<?>, Set<Method>> signatureMap) {

Set<Class<?>> interfaces = new HashSet<Class<?>>();

//循环type类型的接口信息 如果该类型存在与signatureMap当中则加入到set当中去

while (type != null) {

for (Class<?> c : type.getInterfaces()) {

if (signatureMap.containsKey(c)) {

interfaces.add(c);

}

}

type = type.getSuperclass();

}

//转换为数组返回

return interfaces.toArray(new Class<?>[interfaces.size()]);

}

}

**下面是俩个注解类的定义源码：**

package org.apache.ibatis.plugin;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)@Target(ElementType.TYPE)

public @interface Intercepts {

Signature[] value();

}

package org.apache.ibatis.plugin;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)@Target(ElementType.TYPE)

public @interface Signature {

Class<?> type();

String method();

Class<?>[] args();

}

#3 Plugin.wrap方法# 从前面可以看出，每个拦截器的plugin方法是通过调用Plugin.wrap方法来实现的。代码如下：

public static Object wrap(Object target, Interceptor interceptor) {

// 从拦截器的注解中获取拦截的类名和方法信息

Map<Class<?>, Set<Method>> signatureMap = getSignatureMap(interceptor);

Class<?> type = target.getClass();

// 解析被拦截对象的所有接口（注意是接口）

Class<?>[] interfaces = getAllInterfaces(type, signatureMap);

if(interfaces.length > 0) {

// 生成代理对象， Plugin对象为该代理对象的InvocationHandler （InvocationHandler属于java代理的一个重要概念，不熟悉的请参考相关概念）

return Proxy.newProxyInstance(type.getClassLoader(), interfaces, new Plugin(target,interceptor,signatureMap));

}

return target;

}

**这个Plugin类有三个属性：**

private Object target;// 被代理的目标类

private Interceptor interceptor;// 对应的拦截器

private Map<Class<?>, Set<Method>> signatureMap;// 拦截器拦截的方法缓存

**getSignatureMap方法：**

private static Map<Class<?>, Set<Method>> getSignatureMap(Interceptor interceptor) {

Intercepts interceptsAnnotation = interceptor.getClass().getAnnotation(Intercepts.class);

if (interceptsAnnotation == null) { // issue #251

throw new PluginException("No @Intercepts annotation was found in interceptor " + interceptor.getClass().getName());

}

Signature[] sigs = interceptsAnnotation.value();

Map<Class<?>, Set<Method>> signatureMap = new HashMap<Class<?>, Set<Method>>();

for (Signature sig : sigs) {

Set<Method> methods = signatureMap.get(sig.type());

if (methods == null) {

methods = new HashSet<Method>();

signatureMap.put(sig.type(), methods);

}

try {

Method method = sig.type().getMethod(sig.method(), sig.args());

methods.add(method);

} catch (NoSuchMethodException e) {

throw new PluginException("Could not find method on " + sig.type() + " named " + sig.method() + ". Cause: " + e, e);

}

}

return signatureMap;

}

\*\*getSignatureMap方法解释：\*\*首先会拿到拦截器这个类的 @Interceptors注解，然后拿到这个注解的属性 @Signature注解集合，然后遍历这个集合，遍历的时候拿出 @Signature注解的type属性(Class类型)，然后根据这个type得到带有method属性和args属性的Method。由于 @Interceptors注解的 @Signature属性是一个属性，所以最终会返回一个以type为key，value为Set<Method>的Map。

@Intercepts({@Signature(type= Executor.class, method = "update", args = {MappedStatement.class,Object.class})})

比如这个 @Interceptors注解会返回一个key为Executor，value为集合(这个集合只有一个元素，也就是Method实例，这个Method实例就是Executor接口的update方法，且这个方法带有MappedStatement和Object类型的参数)。这个Method实例是根据 @Signature的method和args属性得到的。如果args参数跟type类型的method方法对应不上，那么将会抛出异常。

**getAllInterfaces方法：**

private static Class<?>[] getAllInterfaces(Class<?> type, Map<Class<?>, Set<Method>> signatureMap) {

Set<Class<?>> interfaces = new HashSet<Class<?>>();

while (type != null) {

for (Class<?> c : type.getInterfaces()) {

if (signatureMap.containsKey(c)) {

interfaces.add(c);

}

}

type = type.getSuperclass();

}

return interfaces.toArray(new Class<?>[interfaces.size()]);

}

\*\*getAllInterfaces方法解释：\*\*根据目标实例target(这个target就是之前所说的MyBatis拦截器可以拦截的类，Executor,ParameterHandler,ResultSetHandler,StatementHandler)和它的父类们，返回signatureMap中含有target实现的接口数组。

所以Plugin这个类的作用就是根据 @Interceptors注解，得到这个注解的属性 @Signature数组，然后根据每个 @Signature注解的type，method，args属性使用反射找到对应的Method。最终根据调用的target对象实现的接口决定是否返回一个代理对象替代原先的target对象。

我们再次结合(Executor)interceptorChain.pluginAll(executor)这个语句来看，这个语句内部对executor执行了多次plugin,第一次plugin后通过Plugin.wrap方法生成了第一个代理类，姑且就叫executorProxy1，这个代理类的target属性是该executor对象。第二次plugin后通过Plugin.wrap方法生成了第二个代理类，姑且叫executorProxy2，这个代理类的target属性是executorProxy1...这样通过每个代理类的target属性就构成了一个代理链（从最后一个executorProxyN往前查找，通过target属性可以找到最原始的executor类）。

#4 代理链上的拦截# 代理链生成后，对原始目标的方法调用都转移到代理者的invoke方法上来了。Plugin作为InvocationHandler的实现类，他的invoke方法是怎么样的呢？

比如MyBatis官网的例子，当Configuration调用newExecutor方法的时候，由于Executor接口的update(MappedStatement ms, Object parameter)方法被拦截器被截获。因此最终返回的是一个代理类Plugin，而不是Executor。这样调用方法的时候，如果是个代理类，那么会执行：

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

try {

Set<Method> methods = signatureMap.get(method.getDeclaringClass());

if(methods != null && methods.contains(method)) {

// 调用代理类所属拦截器的intercept方法，

return interceptor.intercept(new Invocation(target, method, args));

}

return method.invoke(target, args);

} catch(Exception e) {

throw ExceptionUtil.unwrapThrowable(e);

}

}

没错，如果找到对应的方法被代理之后，那么会执行Interceptor接口的interceptor方法。

在invoke里，如果方法签名和拦截中的签名一致，就调用拦截器的拦截方法。我们看到传递给拦截器的是一个Invocation对象，这个对象是什么样子的，他的功能又是什么呢？

public class Invocation {

private Object target;

private Method method;

private Object[] args;

public Invocation(Object target, Method method, Object[] args) {

this.target =target;

this.method =method;

this.args =args;

}

...

public Object proceed() throws InvocationTargetException, IllegalAccessException {

return method.invoke(target, args);

}

}

可以看到，Invocation类保存了代理对象的目标类，执行的目标类方法以及传递给它的参数。

在每个拦截器的intercept方法内，最后一个语句一定是return invocation.proceed()（不这么做的话拦截器链就断了，你的mybatis基本上就不能正常工作了）。invocation.proceed()只是简单的调用了下target的对应方法，如果target还是个代理，就又回到了上面的Plugin.invoke方法了。这样就形成了拦截器的调用链推进。

public Object intercept(Invocation invocation) throws Throwable {

//完成代理类本身的逻辑

...

//通过invocation.proceed()方法完成调用链的推进

return invocation.proceed();

}

#5 总结# MyBatis拦截器接口提供的3个方法中，plugin方法用于某些处理器(Handler)的构建过程。interceptor方法用于处理代理类的执行。setProperties方法用于拦截器属性的设置。

其实MyBatis官网提供的使用 @Interceptors和 @Signature注解以及Plugin类这样处理拦截器的方法，我们不一定要直接这样使用。我们也可以抛弃这3个类，直接在plugin方法内部根据target实例的类型做相应的操作。

总体来说MyBatis拦截器还是很简单的，拦截器本身不需要太多的知识点，但是学习拦截器需要对MyBatis中的各个接口很熟悉，因为拦截器涉及到了各个接口的知识点。

**我们假设在MyBatis配置了一个插件，在运行时会发生什么？**

1. 所有可能被拦截的处理类都会生成一个代理
2. 处理类代理在执行对应方法时，判断要不要执行插件中的拦截方法
3. 执行插接中的拦截方法后，推进目标的执行

如果有N个插件，就有N个代理，每个代理都要执行上面的逻辑。这里面的层层代理要多次生成动态代理，是比较影响性能的。虽然能指定插件拦截的位置，但这个是在执行方法时动态判断，初始化的时候就是简单的把插件包装到了所有可以拦截的地方。

**因此，在编写插件时需注意以下几个原则：**

1. 不编写不必要的插件；
2. 实现plugin方法时判断一下目标类型，是本插件要拦截的对象才执行Plugin.wrap方法，否者直接返回目标本省，这样可以减少目标被代理的次数。