mybatis第二天 高级知识

# 复习第一天内容

## mybatis开发步骤

1、配置SqlMapConfig.xml(运行环境、setting运行参数、别名、mapper加载)

2、创建mapper.xml映射文件（一般情况一张表对应一个映射文件）

mapper.xml中定义很多的statement（sql语句、输入映射参数类型、输出映射的参数类型）

3、mapper.xml映射文件在SqlMapConfig.xml中加载

4、使用SqlSessionFacoryBuilder创建SqlSessionFactory

5、通过SqlSessionFactory创建SqlSession(重点)

sqlSession的使用场合是方法体内，因为多个线程对方法体内的局部变量使用不同的内存空间。

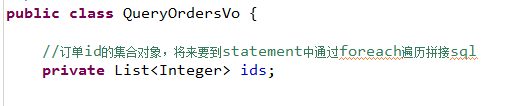
6、调用SqlSession的方法操作数据库(程序员需要根据需求调用不同的操作数据库方法)

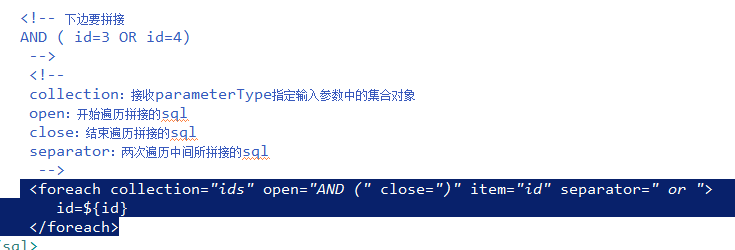
7、如果是插入、删除、更新需要调用SqlSession.commit()方法提交事务

8、关闭SqlSession

## foreach

输入参数中如果有list、数组等可以使用foreach遍历集合对象，从而动态拼接sql。





# 课程计划

mybatis的高级映射，使用resultMap完成高级映射：（掌握）

分析订单商品数据模型（表及表之间的关系）

一对一查询

一对多查询

多对多查询

mybatis的高级映射之延迟加载(了解思想)

mybatis查询缓存(了解思想)：

一级缓存

二级缓存

分布式缓存

mybatis和spring整合(重点掌握)

mybatis逆向工程(代码生成器)(会用即可)

# 分析订单商品数据模型

新的项目，如何分析项目中使用的表及表之间的关系：

1、按业务模块去分析，在有限的表中去分析，不要一次性去分析所有表

2、搞清楚该 业务模块下每张表存储了什么内容（接触业务）

3、搞清楚每张表的主要字段（主键、外键、非空字段、索引字段）

4、搞清楚表与表之间数据库级别的关系（外键关系）

5、分析表与表之间的业务关系



例子：

张三在网站购买商品，他购买了两个商品（三星手机、ibm电脑）

张三在user表存在一条记录

购买商品创建一个订单，在orders表有一条订单记录

一个订单包括两个购买信息（三星手机和ibm电脑）：

购买信息：

购买的商品是哪一个

当时的购买价

购买数量

每个购买信息在orderdetail表中对应一条记录

一个购买信息指定购买的是哪一个具体的商品

三星手机在items表中有一条记录

ibm电脑在items表中有一条记录

# 一对一查询

## 需求

查询订单关联查询用户信息

## sql语句

编写关联查询sql经验：

1、确定主查询表

2、确定关联查询表，确定关联方式（内连接、外连接）

如果通过外键关联查询子表，使用内连接决对没有问题。

SELECT

orders.\*,

username,

sex,

address

FROM

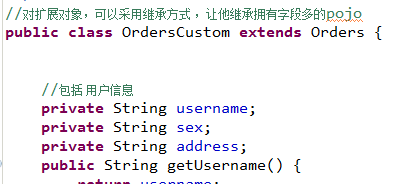
orders,

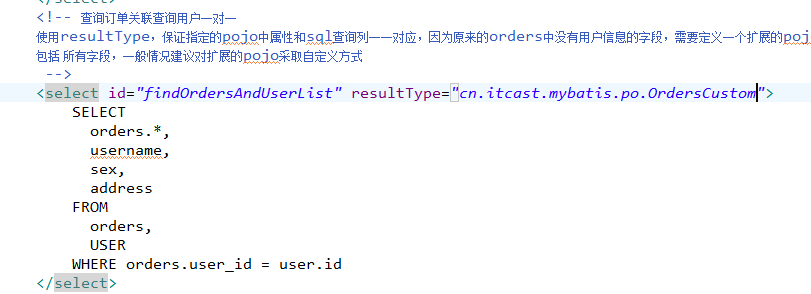
USER

WHERE orders.user\_id = user.id

## 使用resultType实现

如果sql查询列在原来的pojo中少的话，需要自定义一个pojo包括所有列：





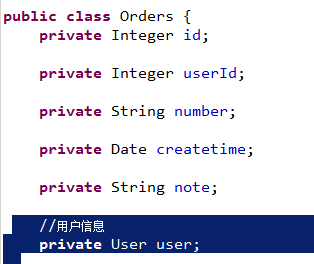


## 使用resultMap

### 需求

在Orders类中创建一个User属性，将订单信息映射到order对象中，关联查询出来的用户信息自动映射到user属性中。

### 在orders类添加user属性



### mapper.xml

resultMap:

<!-- 定义一个resultMap，完成订单关联查询用户信息的映射 -->

<resultMap type=*"orders"* id=*"OrdersAndUserListResultMap"*>

<!-- 订单信息映射

id：订单信息的唯一标识

-->

<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"*/>

<!-- 用户信息映射 -->

<!--

从订单查询用户是一对一关联查询，这里使用association

association：完成关联查询一个对象，对这一个对象进行映射

property：将用户信息映射到orders对象 的哪个属性中

javaType：user属性的类型

-->

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

<!-- id：表示用户信息的唯一标识

property：将user\_id列的值映射到user的哪个属性中

-->

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

<!-- result：用户信息的 普通列映射 -->

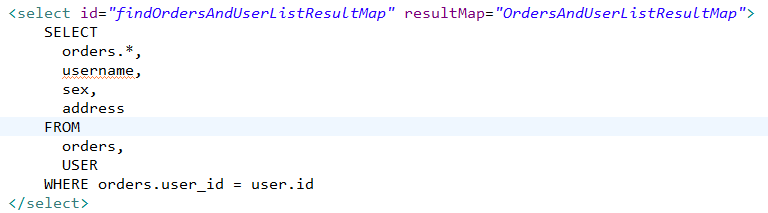
<result column=*"username"* property=*"username"*/>

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

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

</association>

</resultMap>



## 小结

如果没有对结果集的特殊映射要求，建议实际开发中使用resultType进行一对一查询。

特殊映射要求必须要用resultMap：比如将关联查询信息映射到pojo属性中、使用延迟加载。

# 一对多查询

## 需求

业务需求：查询订单关联查询订单明细信息

映射需求：将sql查询结果集通过mybatis映射成java对象，将订单信息映射到orders对象中，将关联查询出来的订单明细信息映射到orders对象中List<Orderdetail>属性中

## sql语句

SELECT

orders.\*,

orderdetail.id orderdetail\_id,

orderdetail.items\_num,

orderdetail.items\_id,

username,

sex,

address

FROM

orders,

USER,

orderdetail

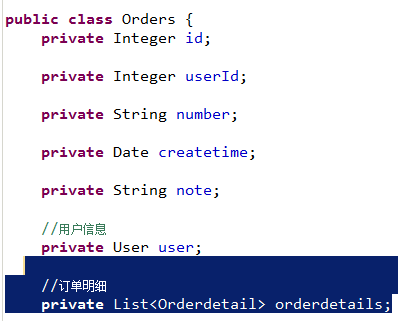
WHERE orders.user\_id = user.id AND orders.id = orderdetail.orders\_id

## 分析

如果使用resultType实现上边的映射需求，只是将查询结果集映射到多个pojo对象中（一个记录映射一个pojo对象）。

使用resultMap提供collection实现将多条记录聚合到一个List中。

## 在orders中添加List<orderdetail>属性



## mapper.xml

定义resultMap

<!-- 订单关联订单明细resultMap -->

<resultMap type=*"orders"* id=*"OrdersAndDetailListResultMap"* extends=*"OrdersAndUserListResultMap"*>

<!-- 订单信息映射、用户信息映射由于使用继承，不用在此配置了 -->

<!-- 明细信息映射

property：将明细信息映射到orders的哪个属性

ofType：orderdetails中的pojo类型

-->

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

<!--

id：订单明细唯一标识

property：将orderdetail\_id列的值映射到ofType指定的pojo的哪个属性中

-->

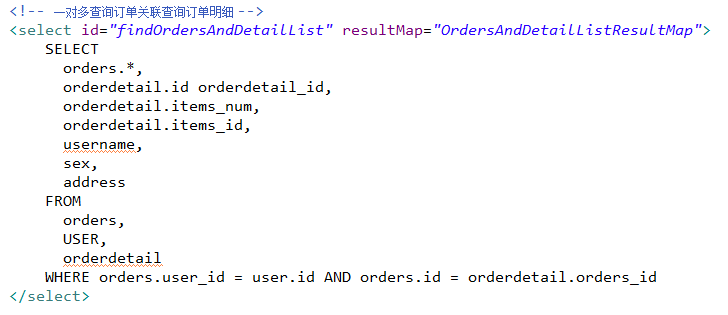
<id column=*"orderdetail\_id"* property=*"id"*/>

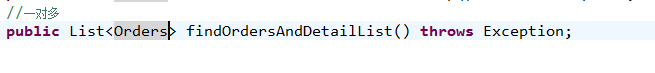
<result column=*"items\_num"* property=*"itemsNum"*/>

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

</collection>

</resultMap>





## 思考

用resultType如何实现

1、将查询出来的所有记录映射到List<pojo>中，到这mybatis执行完了

2、在service代码中，对上边查询出来的List进行汇总处理将明细信息设置到orders中的List<orderdetail>属性中。

实际开发中：

如果只是将sql查询出来的记录信息原原本本显示在jsp中，使用resultType达到目标。

如果对结果集有特殊映射要求，使用resultMap将信息聚合到List属性中：

特殊映射要求：比如：页面左侧二级菜单，通过resultMap 在持久层将数据封装成List<pojo>，pojo中也存在一个list，方便在页面进行循环遍历。

# 多对多查询

## 需求

业务需求：查询用户及用户购买的商品信息

## sql语句

主查询表：用户信息

关联查询表：订单、订单明细、商品信息

SELECT

orders.\*,

orderdetail.id orderdetail\_id,

orderdetail.items\_num,

orderdetail.items\_id,

username,

sex,

address,

items.name,

items.price

FROM

orders,

USER,

orderdetail,

items

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

## 映射分析

根据需求确定映射方法：

如果只是将sql查询出来的列内容原本的显示在jsp中，直接使用reusltType完成。

现在的需求是：

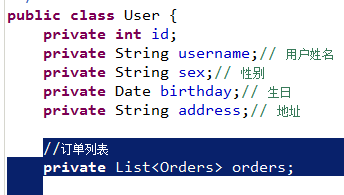
将用户信息映射到user对象中，最终所有用户生成一个List<User>

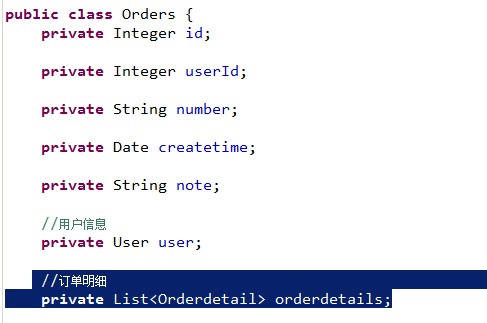
用户创建的订单映射到user对象的List<Orders>属性中

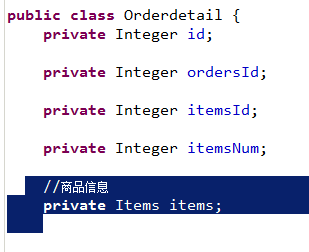
将订单所包括订单明细映射到orders中的List<orderdetail>属性中。

将商品信息映射到orderdetail对象 的Item属性中。

## pojo中属性映射属性







## mapper.xml

resultMap

<!-- 用户关联商品的resultMap -->

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

<!-- 用户信息

id：用户信息唯一标识

-->

<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

一个用户对象多个订单，使用collection

property：订单信息映射到user中的哪个属性中

ofType：orders中的pojo类型

-->

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

<!-- id：订单的唯一标识 -->

<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

一个订单对应多个明细，使用collection

property：将订单明细映射到orders中的orderdetails

ofType：orderdetails中的pojo类型

-->

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

<!-- 这里边是明细的映射配置 -->

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

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

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

<!-- 商品信息

一个订单明细对应一个商品，使用association

property：将商品信息映射到明细对象orderdetail中items属性

javaType：items属性的类型

-->

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

<!-- 在这里是商品的映射配置 -->

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

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

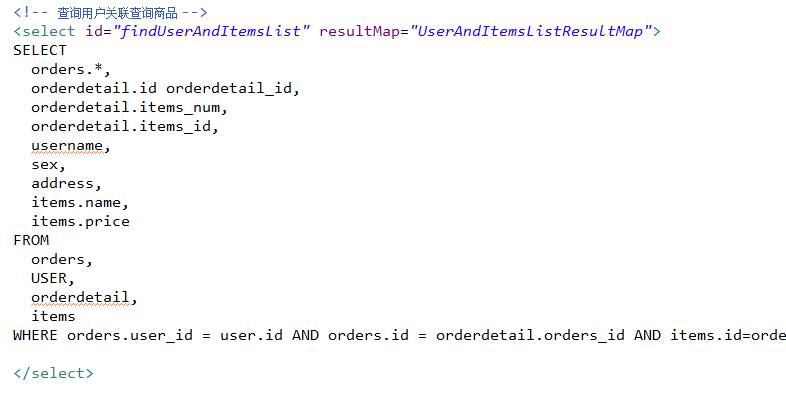
<result column=*"price"* property=*"price"*/>

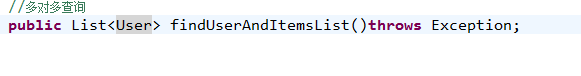
</association>

</collection>

</collection>

</resultMap>





# 上午课程小结

输入映射：将java对象输入到statement中，statement通过#{}和${}接收输入参数。通过基于 OGNL读取输入对象中的属性值。

使用parameterType指定输入参数的类型：简单类型、pojo自定义类型、hashmap。

使用什么类型根据输入参数值类型以及个数决定，如果输入单个值可以使用简单类型、pojo。

如果要输入多个值，只能使用pojo、hashmap，因为pojo中可以将多个属性值封装起来，传入 statement 中。

输出映射：将sql执行的结果集输出映射成java对象。

resultType：指定简单类型、pojo、hashmap，指定的是单条记录映射成的java对象类型。

如果想输出映射成一个pojo对象，resultType指定pojo类型，执行sqlSession查询时，使用selectOne方法。

如果相输出映射成一个List<pojo>，resultType指定pojo类型，执行sqlSession查询时，使用selectList方法

resultMap：

也是对查询结果集进行输出映射，根据自己需求可以通过resultMap将查询结果集映射到pojo中pojo属性中，还可以将多条记录结果集映射到pojo中的List<pojo>集合属性中。

一对多、多对多、一对多这是针对业务来说的，可以写成符合一对一、一对多、多对多业务的sql语句，sql语句的结果集的映射由mybatis完成。

写成一个复杂的多对多的sql语句，使用resultMap或resultType完成结果集映射。

# 延迟加载

## 什么是延迟加载

当我们进行复杂的关联查询时，如果只查询主要内容不查询关联信息就能满足用户的需求时，可以只查询出主要的内容返回用户，当用户需要查询关联信息时，再去查询关联信息，理解为延迟加载，好处是：提高数据库查询性能，因为主要信息不包括关联信息的很可能是单表，单表查询比多表关联查询快，所以提高 数据库查询性能。

## 需求

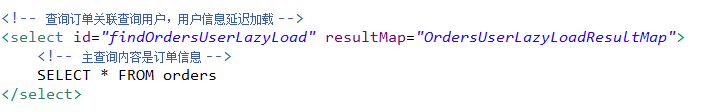
查询订单信息，关联查询用户信息，对关联用户信息按需进行加载。

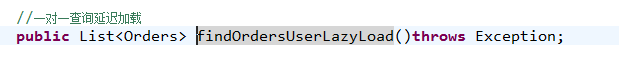
效果：当执行statement的查询时，只查询订单信息(单表)，当需要获取用户信息时去查询用户信息。

## 实现

mybatis通过resultMap可以实现延迟加载，在进行association以及collection关联时可以使用延迟加载。

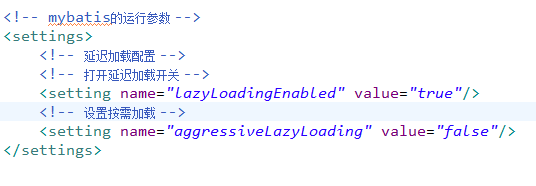




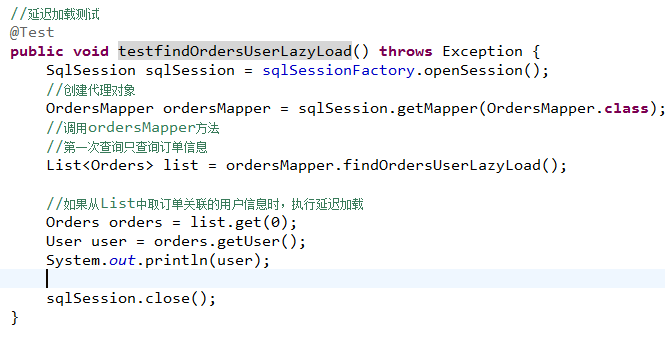


## 配置mybatis运行参数

打开延迟加载的开关，mybatis支持延迟加载。



## 测试



## collection一对多延迟加载

参数association自行练习。

## 思考

如何不使用mybaits的延迟加载如何实现？

定义两个statment（定义两个mapper接口），一个是订单信息、一个是根据用户id查询用户信息。

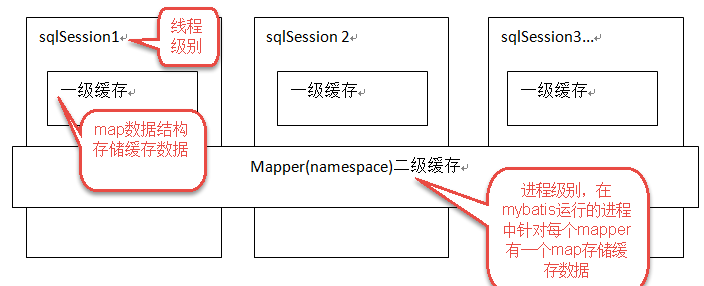
手动控制延迟加载，首先调用订单信息statment，在调用getUser()方法时调用根据用户id查询用户信息的mapper接口。

在实际企业开发中，用户不需要的东西不要去查询！！！！让用户自己去点击链接触发事件执行关联查询。

# 查询缓存

## mybatis的查询缓存

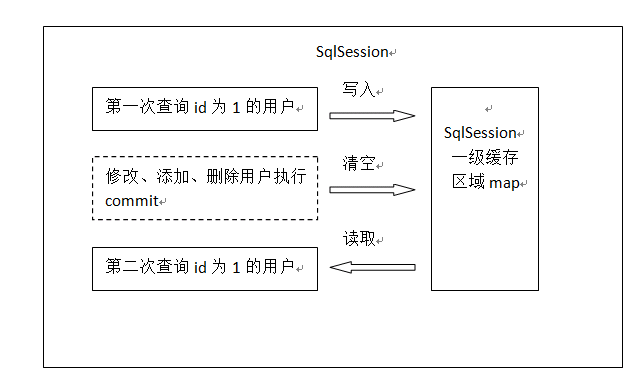
mybatis提供了一级缓存、二级缓存。



一级缓存：是sqlSession级别，是线程级别的缓存。不同的sqlsession的一级缓存map不能共用。

当sqlsession关闭，或commit，该 sqlsession的一级缓存数据清空，二级缓存map数据不清空。

**一级缓存的map的key就是hashcode+statement的id+sql语句+输入参数。。作为一个唯一标识 sql的key。**



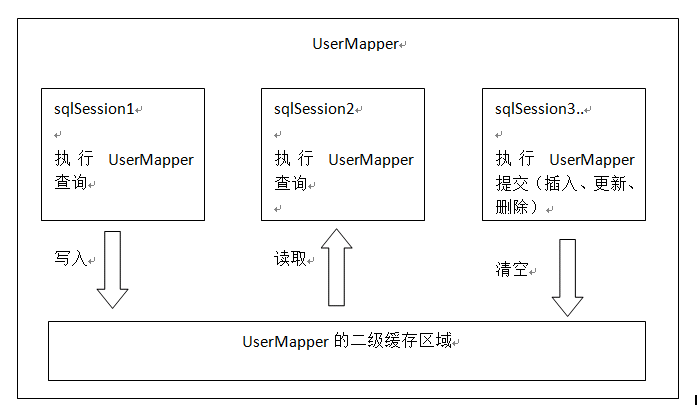
sqlsession去查询时，首先去缓存中查找，如果缓存中存在不再去数据库查询了，返回。

如果缓存中没有，查询数据库，返回给用户之前，将查询到数据存入缓存。

为了避免脏数据，当执行sqlsession的commit方法时将一级缓存数据清空。

二级缓存：是跨sqlsession，是进程级别的缓存。不同的sqlsession的二级缓存的map可以共用。二级缓存是以mapper为单位创建map数据结构，有一个mapper就有几个map。

**二级缓存的map的key就是mapper的namespace（一个mapper生成两个：UserMapper、cn.itcast.mybatis.mapper.UserMapper）作为一个唯一标识 sql的key。**



一个mapper对应一个map数据结构存储缓存数据。

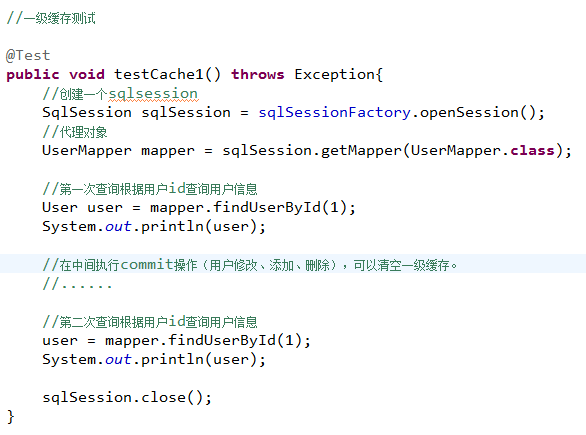
mapper中多个statement，当执行一个查询的statement时，先去二级缓存中找，如果缓存中有，不再从数据库查询数据，直接将二级缓存数据返回。

如果mapper中的其它statement执行commit操作，将该mapper下的二级缓存数据全部清空。

进程结束二级缓存的数据清空。

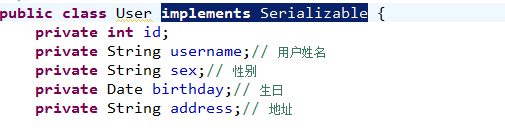
## 一级缓存测试

mybtis默认支持一级缓存。



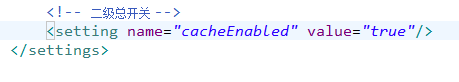
## 二级缓存测试

### 需要将查询结果映射的pojo序列化



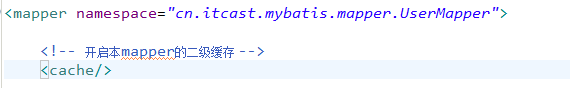
### 开启mybatis的二级缓存

需要开启二级缓存的总开关。



需要开启mapper.xml中的二级缓存开关。

需要在具体 的mapper.xml中添加cache



### 测试代码

//二级缓存测试

@Test

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

//创建sqlsession

SqlSession sqlSession1 = sqlSessionFactory.openSession();

SqlSession sqlSession2 = sqlSessionFactory.openSession();

SqlSession sqlSession3 = sqlSessionFactory.openSession();

//代理对象

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

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

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

//第一次查询根据用户id查询用户信息

User user1 = mapper1.findUserById(1);

System.*out*.println(user1);

//注意：将sqlSession1.close时将数据存入二级缓存

sqlSession1.close();

//使用sqlSession3进行用户添加、修改、删除执行commit，会清空二级缓存

//.....

//第二次查询根据用户id查询用户信息

User user2 = mapper2.findUserById(1);

System.*out*.println(user2);

sqlSession2.close();

}

### 总结

当二级缓存开启，先去二级缓存中查找，如果二级缓存没有，再去一级缓存查找。

## mybatis二级场景(理解)

对于经常查询，且变化较少的数据，可以使用二级缓存。

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。

## mybatis二级缓存局限性(理解)

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。

对于细粒度缓存建议在业务层有针对的去处理。

## 分布式缓存(理解)

mybatis+springmvc应用

tomcat1(进程 )

192.168.101.2

mybatis+springmvc应用

tomcat2(进程 )

192.168.101.3

负载均衡 lvs+nginx

分布式缓存应用（进程）

redis、memcache

集中对缓存数据进行统一管理

比如：商品信息

tcp

tcp

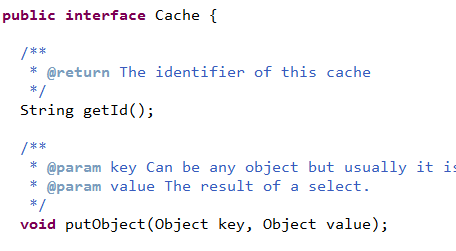
## mybatis如何整合第三方的缓存框架

思路：

使用redis统一管理缓存。

mybatis从数据库查询出数据，mybatis基于cache接口需要写缓存数据。

可以扩展cache接口，接口实现是将数据存到redis中。



整合ehcache参考教案自学。

# spring和mybatis整合

## 整合思路

通过spring以单例管理SqlSessionFactory.

如果使用原始dao开发，通过spring管理dao。

如果使用mapper代理开发，通过spring管理mapper代理对象。

## 搭建环境

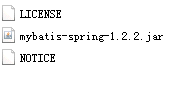
mysql：5.1

java：1.7

spring：3.2

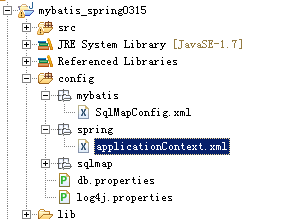
mybatis：3.2.7

mybatis-spring的整合包（mybatis管方出的整合包）





工程：



## 开发原始dao方法

### applicationContext.xml-sqlSessionFactory

配置数据源，本教程 使用dbcp。

以单例管理SqlSessionFactory



### applicationContext.xml-dao

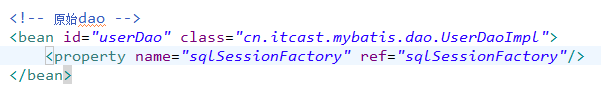
使用声明式配置dao。

dao实现类中注入SqlSessionFactory

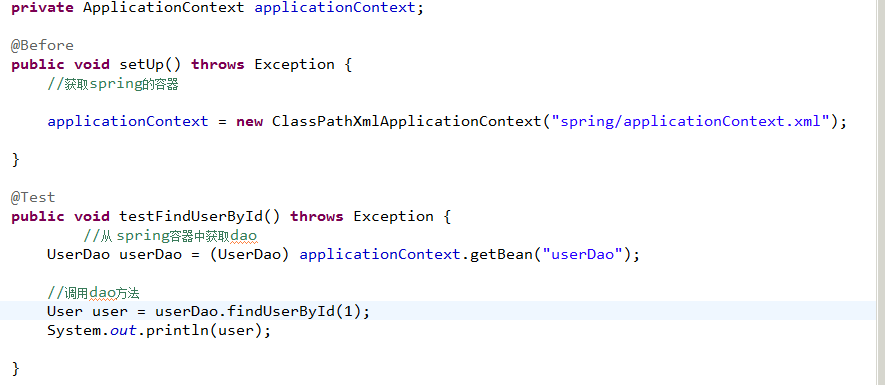
让dao实现类继承SqlSessionDaoSupport



在applicationContext.xml配置上边的dao类。



### 测试

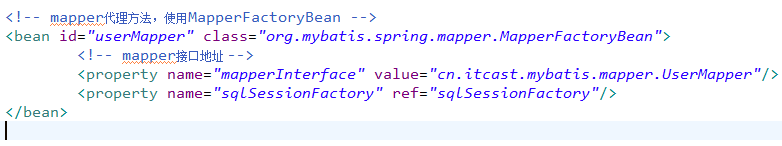


## 开发mapper代理方法

程序员只写mapper接口。

### 使用MapperFactoryBean

在 applicationContext.xml配置MAPPER：



如果需要创建多个mapper对象，需要在spring中配置多个，比较繁琐，不推荐使用。

### 使用MapperScannerConfigurer扫描mapper

在 applicationContext.xml配置mapper扫描器

<!-- 使用MapperScannerConfigurer扫描mapper

扫描器将mapper扫描出来自动 注册到spring容器，bean的id是类名(首字母小写)

-->

<bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>

<!-- 指定扫描的包

如果扫描多个包中间使用半角逗号分隔

如果使用扫描器，不用在sqlmapconfig.xml中去配置mapper的扫描了，如果使用mapper代理的开发，在SqlMapConfig.xml中不用配置mapper项了

-->

<property name=*"basePackage"* value=*"cn.itcast.mybatis.mapper"*/>

<!-- 使用sqlSessionFactoryBeanName注入sqlSessionFactory -->

<property name=*"sqlSessionFactoryBeanName"* value=*"sqlSessionFactory"*/>

<!--

注意： 这里使用sqlSessionFactoryBeanName而不使用sqlSessionFactory原因如下：

MapperScannerConigurer在扫描mapper时需要注入 sqlSessionFactory，如果使用

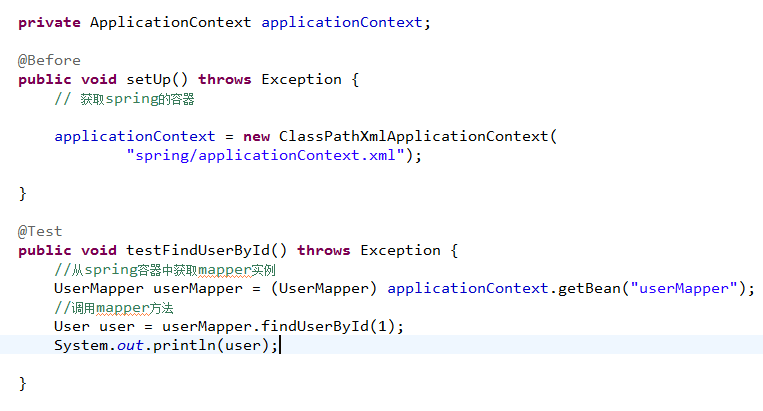
<property name="sqlSessionFactory" ref="sqlSessionFactory"/>

会存在PropertyPlaceholderConfigurer还没来得及替换dataSource定义中的${jdbc.driver}等数据源变量就注入到了MapperScannerConigurer中，将导致数据库连接不上，如果改为如下方式可以解决问题：

-->

</bean>

### 测试

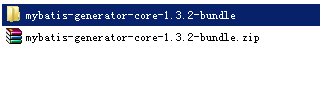


# mybatis的逆向工程

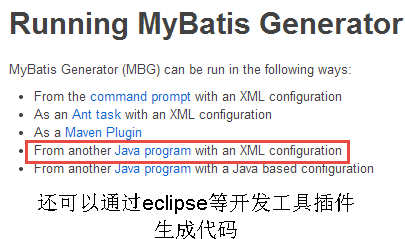
## 需求

使用mybatis需要手动写很多sql，mybatis官方推出自动代码生成器。

在企业实际开发中一般情况 ，先有的数据库表，再根据数据库表创建pojo类、mapper.xml，mybatis代码生成器根据数据库表生成java代码（mapper.java、mapper.xml、pojo）



## 运行生成器



建议使用java程序运行。

## 生成配置文件generatorConfig.xml

<?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>

<!-- <table tableName="orders"></table>

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

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

<!-- <table schema="" tableName="sys\_user"></table>

<table schema="" tableName="sys\_role"></table>

<table schema="" tableName="sys\_permission"></table>

<table schema="" tableName="sys\_user\_role"></table>

<table schema="" tableName="sys\_role\_permission"></table> -->

<!-- 有些表的字段需要指定java类型

<table schema="" tableName="">

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

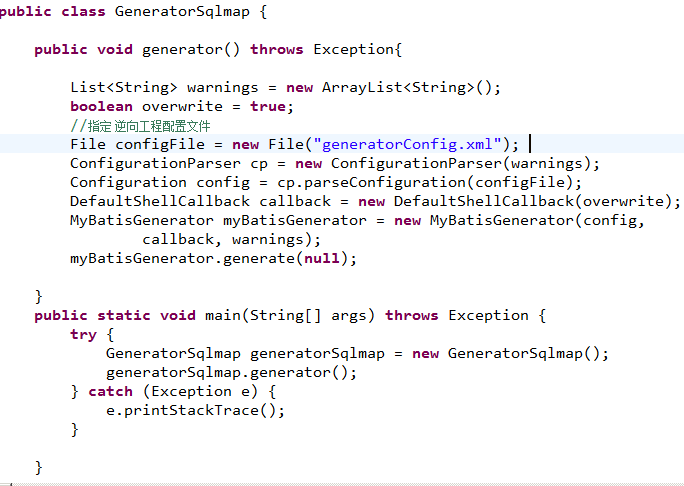
</table> -->

</context>

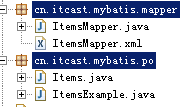
</generatorConfiguration>

## 开始生成代码

运行以下：



生成后的代码：



## 使用生成的代码操作数据库

要将生成的代码拷贝到工程中，才能使用。

**private** ApplicationContext applicationContext;

**private** ItemsMapper itemsMapper;

@Before

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

//获取spring的容器

applicationContext = **new** ClassPathXmlApplicationContext("spring/applicationContext.xml");

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

}

@Test

**public** **void** testDeleteByExample() {

}

@Test

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

*fail*("Not yet implemented");

}

@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();

//拼接商品id等于？？某

criteria.andIdEqualTo(1);

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

//查询时将大文本的列也查询出来

// itemsMapper.selectByExampleWithBLOBs(example);

System.*out*.println(list);

}

//根据主键查询

@Test

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

Items items = itemsMapper.selectByPrimaryKey(1);

System.*out*.println(items);

}

@Test

**public** **void** testUpdateByPrimaryKeySelective() {

*fail*("Not yet implemented");

}

@Test

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

//updateByPrimaryKey不管输入 的参数是否为空都更新，所以实际开中，先从数据库查询出来，再设置更新的值。

//itemsMapper.updateByPrimaryKey(record);

//输入的参数不为空再更新

// itemsMapper.updateByPrimaryKeySelective(record)

}