mybatis第二天 高级映射 查询缓存 和spring整合

# 复习

mybatis是什么？

mybatis是一个持久层框架，是apache下的开源项目，前身是itbatis，是一个不完全的ORM框架，mybatis提供输入和输出的映射，需要程序员自己写sql语句，mybatis重点对 sql语句的灵活操作。

适合用于：需求变化频繁， 数据模型不固定的项目，例如：互联网项目。

mybatis架构？

SqlMapConfig.xml（名称不固定），配置内容：数据源、事务、properties、typeAliases、settings、mapper配置。

SqlSessionFactory--会话工厂，作用是创建SqlSession，实际开发中以单例模式管理 SqlSessionFactory。

SqlSession--会话，是一个面向用户（程序员）的接口，使用mapper代理方法开发是不需要程序员直接调用sqlSession的方法。是线程不安全，最佳适用场合方法体内。

mybatis开发dao的方法：

1、原始dao开发方法，需要程序员编写dao接口和实现类，此方法在当前企业中还有使用，因为ibatis使用的就是原始dao开发方法。

2、mapper代理方法，程序员只需要写mapper接口（相当于dao接口），mybatis自动根据mapper接口和mapper接口对应的statement自动生成代理对象（接口实现类对象）。

开发需要遵循规则：

1）mapper.xml中namespace是mapper接口的全限定名

2）mapper.xml中statement的id为mapper接口方法名

3）mapper.xml中statement的输入映射类型（parameterType）和mapper接口方法输入参数类型一致

4）mapper.xml中statement的输出映射类型（resultType）和mapper接口方法返回结果类型一致

resultType和resultMap都可以完成输出映射：

resultType映射要求sql查询的列名和输出映射pojo类型的属性名一致

resultMap映射时对sql查询的列名和输出映射pojo类型的属性名作一个对应关系。

动态sql：

#{}和${}完成输入参数的属性值获取，通过OGNL获取parameterType指定pojo的属性名。

#{}：占位符号，好处防止sql注入

${}：sql拼接符号

if

where

foreach

# 课程计划

1、使用resultMap完成高级映射(重点)

学习商品订单数据模型（一对一、一对多、多对多）

resultMap实现一对一、一对多、多对多

延迟加载

2、查询缓存(重点)

一级缓存

二级缓存

3、mbatis和spring整合(重点)

4、mybatis逆向工程(常用)

# 商品订单数据模型



学会在企业中如何去分析陌生表的数据模型：

1、学习单表记录了什么东西（去学习理解需求）

2、学习单表重要字段的意义（优先学习不能为空的字段）

3、学习表与表之间的关系（一对一、一对多、多对多）

通过表的外键分析表之间的关系

注意：分析表与表之间的关系时是要建立在具体 的业务意义基础之上

**user**

**orders**

user\_id：用户id

user\_id

**orderdetail**

order\_id：订单id

item\_id：商品id

order\_id

**product**

item\_id

一对一

一对多

一对多

一对一

一对一

一对多

orders--> product

通过orderdetail

建立 一对多

product -->orders

通过orderdetail

建立 一对多

总之：多对多

用户和商品是多对多

用户表user：

记录了购买商品的用户

订单表orders：

记录了用户所创建的订单信息

订单明细表orderdetail：

记录了用户创建订单的详细信息

商品信息表product：

记录了商家提供的商品信息

分析表与表之间的关系：

用户user和订单orders：

user---->orders：一个用户可以创建多个订单 一对多

orders-->user：一个订单只能由一个用户创建 一对一

订单orders和订单明细orderdetail：

orders-->orderdetail：一个订单可以包括多个订单明细 一对多

orderdetail-->orders：一个订单明细只属于一个订单 一对一

订单明细orderdetail和商品信息product：

orderdetail-->product：一个订单明细对应一个商品信息 一对一

product--> orderdetail：一个商品对应多个订单明细 一对多

# 一对一查询

## 需求

查询订单信息关联查询用户信息

## sql语句

查询语句：

先确定主查询表：订单信息表

再确定关联查询表：用户信息

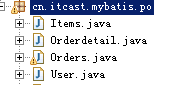
通过orders关联查询用户使用user\_id一个外键，只能关联查询出一条用户记录就可以使用内连接

<**select id="findOrderUserList" resultType="orderCustom"**>  
 SELECT orders.*\**, user.username, user.sex  
 FROM orders, USER  
 WHERE orders.user\_id = user.id  
</**select**>

## 使用resultType实现

### 创建po类

基础的单表的 po类：



### 一对一查询映射的pojo

创建pojo包括 订单信息和用户信息，resultType才可以完成映射。

创建OrderCustom作为自定义pojo，继承sql查询列多的po类。

*/\*\*  
 \* 订单扩展对象 用来完成订单和用户查询结果的映射<br>  
 \* create:2016-10-18 20:10  
 \*  
 \** ***@author*** *李欣  
 \** ***@versin*** *1.0  
 \*/***public class** OrderCustom **extends** Orders {  
  
 *// 补充用户信息* **private** String **username**;  
 **private** String **sex**;  
 **private** String **address**;

}

### mapper.xml

定义mapper.xml文件，

*<!-- 【一对一查询】使用reusltType完成  
 查询订单关联查询用户信息 -->*<**select id="findOrderUserList" resultType="orderCustom"**>  
 SELECT user.username, user.sex,  
 orders.*\** FROM user, orders  
 WHERE user.id=orders.user\_id  
</**select**>

### mapper.java

*// 一对一查询，查询订单关联查询用户，使用resultType***public** List<OrderCustom> findOrderUserList() **throws** Exception;

## 使用resultMap实现一对一

### resultMap映射思路

resultMap提供一对一关联查询的映射和一对多关联查询映射，一对一映射思路：将关联查询的信息映射到pojo中，如下：

在Orders类中创建一个User属性，将关联查询的信息映射到User属性中。

**public class** Orders {  
  
 **private** Integer **id**;  
 **private** Integer **userId**;  
 **private** String **number**;  
 **private** Date **createTime**;  
 **private** String **note**;  
  
 *// 关联用户信息* **private** User **user**;

}

### mapper.xml

*<!-- 【一对一查询】使用reusltMap完成  
 查询订单关联查询用户信息 -->*<**select id="findOrderUserListResultMap" resultMap="ordersUserResultMap"**>  
 SELECT user.username, user.sex,  
 orders.*\** FROM user, orders  
 WHERE user.id=orders.user\_id  
</**select**>

### resultMap定义

*<!-- 【一对一查询】resultMap-->*<**resultMap type="orders" id="ordersUserResultMap"**>  
 *<!-- 完成了订单信息的映射配置 -->  
 <!-- id：订单关联用户查询的唯 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"**/>  
  
 *<!-- 下边完成关联信息的映射  
 association：用于对关联信息映射到单个pojo  
 property： 要将关联信息映射到orders的哪个属性中  
 javaType： 关联信息映射到orders的属性的类型，是user的类型  
 -->* <**association property="user" javaType="User"**>  
 *<!-- id： 关联信息的唯 一标识 -->  
 <!-- property： 要映射到user的哪个属性中-->* <**id column="user\_id" property="id"**/>  
 *<!-- result就是普通列的映射 -->* <**result column="username" property="username"**/>  
 <**result column="sex" property="sex"**/>  
 </**association**>  
</**resultMap**>

### mapper.java

*// 一对一查询，使用resultMap***public** List<Orders> findOrderUserListResultMap() **throws** Exception;

## 小结

**resultType：**要自定义pojo 保证sql查询列和pojo的属性对应，这种方法相对较简单，所以应用广泛。

**resultMap：**使用association完成一对一映射需要配置一个resultMap，过程有点复杂，如果要实现延迟加载就只能用resultMap实现 ，如果为了方便对关联信息进行解析，也可以用association将关联信息映射到pojo中方便解析。

# 一对多查询

## 需求

查询所有订单信息及订单下的订单明细信息。

## sql语句

主查询表：订单表

关联查询表：订单明细

SELECT user.username, user.sex ,  
 orders.*\**,  
 orderdetail.id orderDetail\_id, orderdetail.orders\_id, orderdetail.product\_id, orderdetail.product\_num  
FROM user, orders, orderdetail  
WHERE user.id=orders.user\_id AND orders.id = orderdetail.orders\_id

## resultMap进行一对多映射思路

resultMap 提供collection完成关联信息映射到集合对象中。

在orders类中创建集合属性：

**public class** Orders {  
  
 **private** Integer **id**;  
 **private** Integer **userId**;  
 **private** String **number**;  
 **private** Date **createTime**;  
 **private** String **note**;  
  
 *// 关联用户信息* **private** User **user**;  
  
 ***// 订单明细* List<OrderDetail> orderDetails;**

}

## mapper.xml

*<!-- 【一对多查询】使用reusltMap完成  
 查询订单关联查询订单明细 -->*<**select id="findOrderAndOrderDetails" resultMap="orderAndOrderDetails"**>  
 SELECT user.username, user.sex ,  
 orders.*\**,  
 orderdetail.id orderDetail\_id, orderdetail.orders\_id, orderdetail.product\_id, orderdetail.product\_num  
 FROM user, orders, orderdetail  
 WHERE user.id=orders.user\_id AND orders.id = orderdetail.orders\_id  
</**select**>

## resultMap定义

*<!-- 【一对多】，查询订单及订单明细 -->*<**resultMap type="orders" id="orderAndOrderDetails" extends="ordersUserResultMap"**>  
 *<!-- 映射订单信息，和用户信息，这里使用继承ordersUserResultMap -->  
  
 <!-- 映射订单明细信息  
 property： 要将关联信息映射到orders的哪个属性中  
 ofType： 集合中pojo的类型 -->* <**collection property="orderDetails" ofType="cn.qdgxy.mybatis.po.OrderDetail"**>  
 *<!-- id： 关联信息订单明细的唯一标识  
 property： OrderDetail的属性名 -->* <**id column="orderDetail\_id" property="id"**/>  
 <**result column="orders\_id" property="ordersId"**/>  
 <**result column="product\_id" property="productId"**/>  
 <**result column="product\_num" property="productNum"**/>  
 </**collection**>  
</**resultMap**>

## mapper.java

*// 一对多查询，使用resultMap***public** List<Orders> findOrderAndOrderDetails() **throws** Exception;

# 一对多查询(复杂)

## 需求

查询所有用户信息，关联查询订单及订单明细信息及商品信息，订单明细信息中关联查询商品信息

## sql

主查询表：用户信息

关联查询：订单、订单明细，商品信息

SELECT user.username, user.sex ,  
 orders.*\**,  
 orderdetail.id orderdetail\_id, orderdetail.product\_id, orderdetail.product\_num  
FROM user, orders, orderdetail  
WHERE user.id=orders.user\_id AND orders.id = orderdetail.orders\_id

## pojo定义

在user.java中创建映射的属性：集合 List<Orders> orderlist

在Orders中创建映射的属性：集合List<Orderdetail> orderdetails

在Orderdetail中创建商品属性：pojo product product

**public class** User {  
  
 **private int id**;  
 **private** String **username**;  
 **private** String **sex**;  
 **private** Date **birthday**;  
 **private** String **address**;  
  
 *// 多个订单* **private** List<Orders> **orderList**;

}

**public class** Orders {  
  
 **private** Integer **id**;  
 **private** Integer **userId**;  
 **private** String **number**;  
 **private** Date **createTime**;  
 **private** String **note**;  
  
 *// 关联用户信息* **private** User **user**;  
  
 *// 订单明细* List<OrderDetail> **orderDetails**;

}

**public class** OrderDetail {  
  
 **private** Integer **id**;  
 **private** Integer **ordersId**;  
 **private** Integer **productId**;  
 **private** Integer **productNum**;  
  
 *// 商品信息* **private** Product **product**;

}

## mapper.xml

*<!-- 【一对多查询】使用reusltMap完成  
 查询用户及订单和订单明细，关联商品的信息 -->*<**select id="findUserOrderDetail" resultMap="userOrderDetailResultMap"**>  
 SELECT user.username, user.sex,  
 orders.*\**,  
 orderdetail.id orderdetail\_id, orderdetail.product\_num, orderdetail.product\_id,  
 product.name product\_name, product.detail product\_detail  
 FROM user, orders, orderdetail, product  
 WHERE user.id=orders.user\_id  
 AND orders.id = orderdetail.orders\_id  
 AND orderdetail.product\_id=product.id  
</**select**>

## resultMap

*<!-- 一对多查询，查询用户及订单明细和商品信息 -->*<**resultMap type="user" id="userOrderDetailResultMap"**>  
 *<!-- 用户信息映射 -->* <**id column="user\_id" property="id"**/>  
 <**result column="username" property="username"**/>  
 <**result column="sex" property="sex"**/>  
  
 *<!-- 订单信息 -->* <**collection property="orderList" ofType="cn.qdgxy.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"**/>  
  
 *<!-- 订单明细映射 -->* <**collection property="orderDetails" ofType="cn.qdgxy.mybatis.po.OrderDetail"**>  
 *<!-- id： 关联信息订单明细的唯 一标识  
 property： OrderDetail的属性名  
 -->* <**id column="orderDetail\_id" property="id"**/>  
 <**result column="product\_num" property="productNum"**/>  
 <**result column="product\_id" property="productId"**/>  
  
 *<!-- 商品信息 -->* <**association property="product" javaType="cn.qdgxy.mybatis.po.Product"**>  
 <**id column="item\_id" property="id"**/>  
 <**result column="product\_name" property="name"**/>  
 <**result column="product\_detail" property="detail"**/>  
 </**association**>  
 </**collection**>  
 </**collection**>  
</**resultMap**>

## mapper.java

*// 一对多查询，使用resultMap***public** List<User> findUserOrderDetail() **throws** Exception;

# 多对多查询（自己实现 ）

一对多是多对多的特例。

需求1：

查询显示字段：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：

查询显示字段：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。

# 延迟加载

## 使用延迟加载意义

在进行数据查询时，**为了提高数据库查询性能**，尽量使用单表查询，因为单表查询比多表关联查询速度要快。

如果查询单表就可以满足需求，一开始先查询单表，当需要关联信息时，再关联查询，当需要关联信息再查询这个叫延迟加载。

mybatis中resultMap提供延迟加载功能，通过resultMap配置延迟加载。

## 配置mybatis支持延迟加载

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

*<!-- 全局配置参数 -->*<**settings**>  
 *<!-- 延迟加载总开关 -->* <**setting name="lazyLoadingEnabled" value="true"** />  
 *<!-- 设置按需加载 -->* <**setting name="aggressiveLazyLoading" value="false"** />  
</**settings**>

## 延迟加载实现

### 实现思路

需求：

查询订单及用户的信息，一对一查询。

刚开始只查询订单信息

当需要用户时调用 Orders类中的getUser()方法执行延迟加载 ，向数据库发出sql。

### mapper.xml

*<!-- 【一对一查询延迟加载】  
 开始只查询订单，对用户信息进行延迟加载 -->*<**select id="findOrderUserListLazyLoading" resultMap="orderCustomLazyLoading"**>  
 SELECT orders.*\** FROM orders  
</**select**>

### resultMap

*<!-- 一对一查询延迟加载 的配置 -->*<**resultMap type="orders" id="orderCustomLazyLoading"**>  
 *<!-- 完成了订单信息的映射配置 -->  
 <!-- id：订单关联用户查询的唯 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"**/>  
 *<!-- 配置用户信息的延迟加载  
 select： 延迟加载执行的sql所在的statement的id，如果不在同一个namespace需要加namespace  
 sql： 根据用户id查询用户信息  
 column： 关联查询的列  
 property:将关联查询的用户信息设置到Orders的哪个属性 -->* <**association property="user" select="cn.qdgxy.mybatis.mapper.UserMapper.findUserById" column="user\_id"**/>  
</**resultMap**>

### mapper.java

*//一对一查询，延迟加载***public** List<Orders> findOrderUserListLazyLoading() **throws** Exception;

### 测试代码

@Test  
**public void** findOrderUserListLazyLoading() **throws** Exception {  
 SqlSession sqlSession = **sqlSessionFactory**.openSession();  
 *// 创建mapper代理对象* OrdersMapperCustom ordersMapperCustom = sqlSession.getMapper(OrdersMapperCustom.**class**);  
  
 List<Orders> orderList = ordersMapperCustom.findOrderUserListLazyLoading();  
  
 *// 这里执行延迟加载，要发出sql* User user = orderList.get(0).getUser();  
 System.***out***.println(user);  
}

## 一对多延迟加载

一对多延迟加载的方法同一对一延迟加载，在collection标签中配置select内容。

本部分内容自学。同一对一。

# 总结resultType、resultMap、延迟加载使用场景

**延迟加载：**

延迟加载实现的方法多种多样，在只查询单表就可以满足需求，为了提高数据库查询性能使用延迟加载，再查询关联信息。

**mybatis提供延迟加载的功能用于service层。**

**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，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。

association：

作用：

将关联查询信息映射到一个pojo类中。

场合：

为了**方便获取关联信息**可以使用association将关联订单映射为pojo，比如：查询订单及关联用户信息。

collection：

作用：

将关联查询信息映射到一个list集合中。

场合：

为了**方便获取关联信息**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和功能，可使用collection将模块和功能列表映射到list中。

# 查询缓存

## 缓存的意义

将用户经常查询的数据放在缓存（内存）中，用户去查询数据就不用从磁盘上(关系型数据库数据文件)查询，从缓存中查询，从而提高查询效率，解决了高并发系统的性能问题。

用户、程序

**系统持久层(mybatis)**

**缓存区域(数据库取出数据缓存**)

数据库

缓存数据

系统业务层(spring)

缓存区域（业务数据进行缓存）

系统控制层(springmvc)

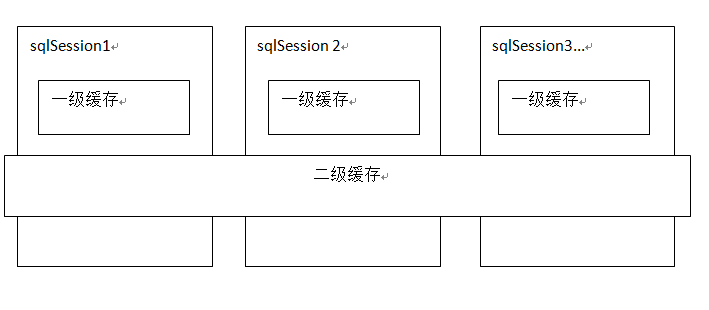
缓存区域(响应的jsp页面进行缓存)

缓存数据

缓存数据

## mybatis持久层缓存

mybatis提供一级缓存和二级缓存

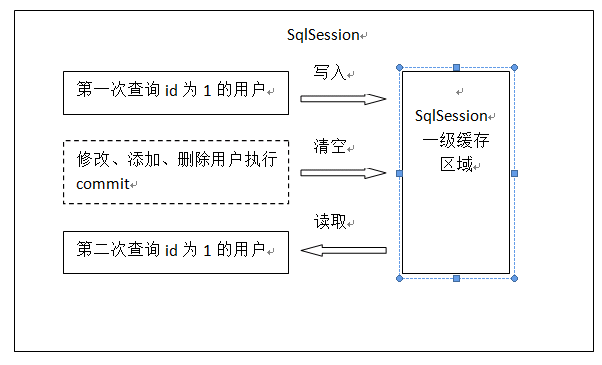


**一级缓存**是一个SqlSession级别，sqlsession只能访问自己的一级缓存的数据。

**二级缓存**是跨sqlSession，是mapper级别的缓存，对于mapper级别的缓存不同的sqlsession是可以共享的。

## 一级缓存

### 原理



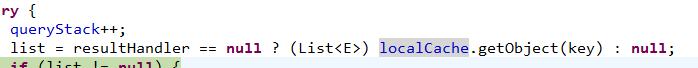
第一次发出一个查询sql，sql查询结果写入sqlsession的一级缓存中，缓存使用的数据结构是一个map<key,value>

key： hashcode+sql+sql输入参数+输出参数（sql的唯一标识）

value： 用户信息

同一个sqlsession再次发出相同的sql，就从缓存中取不走数据库。如果两次中间出现commit操作（修改、添加、删除），本sqlsession中的一级缓存区域全部清空，下次再去缓存中查询不到所以要从数据库查询，从数据库查询到再写入缓存。

每次查询都先从缓存中查询：



如果缓存中查询到则将缓存数据直接返回。

如果缓存中查询不到就从数据库查询：



### 一级缓存配置

mybatis默认支持一级缓存不需要配置。

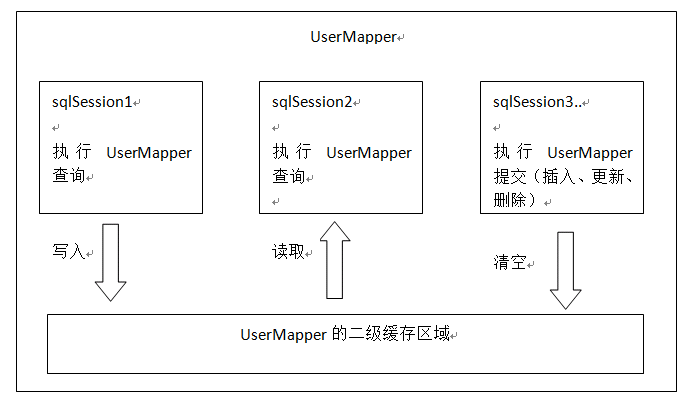
**注意：mybatis和spring整合后进行mapper代理开发，不支持一级缓存，mybatis和spring整合，spring按照mapper的模板去生成mapper代理对象，模板中在最后统一关闭sqlsession。**

### 一级缓存测试

*//一级缓存*@Test  
**public void** testCache1() **throws** Exception {  
 SqlSession sqlSession = **sqlSessionFactory**.openSession();  
 UserMapper userMapper = sqlSession.getMapper(UserMapper.**class**);  
  
 *//第一次查询用户id为1的用户* User user = userMapper.findUserById(1);  
 System.***out***.println(user);  
  
 *//中间修改用户要清空缓存，目的防止查询出脏数据* user.setUsername(**"测试用户2"**);  
 userMapper.updateUser(user);  
 sqlSession.commit();  
  
 *//第二次查询用户id为1的用户* User user2 = userMapper.findUserById(1);  
 System.***out***.println(user2);  
  
 sqlSession.close();  
}

## 二级缓存

### 原理



二级缓存的范围是mapper级别（mapper同一个命名空间），mapper以命名空间为单位创建缓存数据结构，结构是map<key、value>。

每次查询先看是否开启二级缓存，如果开启从二级缓存的数据结构中取缓存数据，



如果从二级缓存没有取到，再从一级缓存中找，如果一级缓存也没有，从数据库查询。

### mybatis二级缓存配置

1. 在核心配置文件SqlMapConfig.xml中加入

*<!-- 【开启二级缓存】 -->*<**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 |

1. 要在你的Mapper映射文件中添加一行： <cache /> ，表示此mapper开启二级缓存。

*<!-- 【开启二级缓存】 -->*<**cache**/>

### 查询结果映射的pojo序列化

mybatis**二级缓存需要将查询结果映射的pojo实现 java.io.serializable接口**，如果不实现则抛出异常：

org.apache.ibatis.cache.CacheException: Error serializing object. Cause: java.io.NotSerializableException: cn.itcast.mybatis.po.User

二级缓存可以将内存的数据写到磁盘，存在对象的序列化和反序列化，所以要实现java.io.serializable接口。

如果结果映射的pojo中还包括了pojo，都要实现java.io.serializable接口。

**public class** User **implements** Serializable {  
  
 **private int id**;  
 **private** String **username**;  
 **private** String **sex**;  
 **private** Date **birthday**;  
 **private** String **address**;  
  
 *// 多个订单* **private** List<Orders> **orderList**;

}

.................. 里面包括的类也要实现

### 二级缓存禁用

对于变化频率较高的sql，需要禁用二级缓存：

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

### 刷新缓存

如果sqlsession操作commit操作，对二级缓存进行刷新（全局清空）。

设置statement的flushCache是否刷新缓存，默认值是true。

<**update id="updateUser" parameterType="User" flushCache="false"**>  
 update user set username=#{username},birthday=#{birthday},sex=#{sex},address=#{address} where id=#{id}  
</**update**>

### 测试代码

*//二级缓存的测试*@Test  
**public void** testCache2() **throws** Exception {  
 SqlSession sqlSession1 = **sqlSessionFactory**.openSession();  
 SqlSession sqlSession2 = **sqlSessionFactory**.openSession();  
 SqlSession sqlSession3 = **sqlSessionFactory**.openSession();  
 UserMapper userMapper1 = sqlSession1.getMapper(UserMapper.**class**);  
 UserMapper userMapper2 = sqlSession2.getMapper(UserMapper.**class**);  
 UserMapper userMapper3 = sqlSession3.getMapper(UserMapper.**class**);  
  
 *//第一次查询用户id为1的用户* User user = userMapper1.findUserById(1);  
 System.***out***.println(user);  
 sqlSession1.close();  
  
 *//中间修改用户要清空缓存，目的防止查询出脏数据* user.setUsername(**"测试用户2"**);  
 userMapper3.updateUser(user);  
 sqlSession3.commit();  
 sqlSession3.close();  
  
  
 *//第二次查询用户id为1的用户* User user2 = userMapper2.findUserById(1);  
 System.***out***.println(user2);  
 sqlSession2.close();  
}

### mybatis的cache参数（了解）

mybatis的cache参数只适用于mybatis维护缓存。

**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和ehcache缓存框架整合

mybatis二级缓存通过ehcache维护缓存数据。

### 分布缓存

将缓存数据数据进行分布式管理。

系统1

springmvc

mybatis

系统2

springmvc

mybatis

用户请求

缓存框架（远程服务器集群）

redis、memcached、ehcache

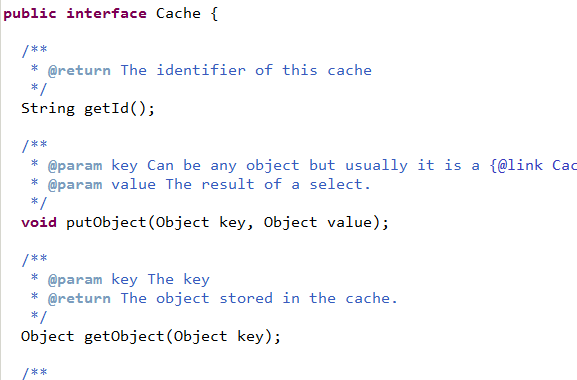
存取缓存数据

存取缓存数据

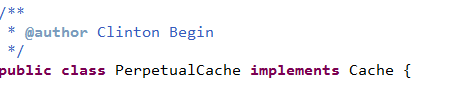
### mybatis和ehcache思路

通过mybatis和ehcache框架进行整合，就可以把缓存数据的管理托管给ehcache。

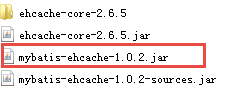
在mybatis中提供一个cache接口，只要实现cache接口就可以把缓存数据灵活的管理起来。



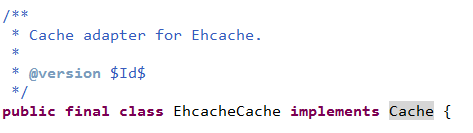
mybatis中默认实现：



### 下载和ehcache整合的jar包



ehcache对cache接口的实现类：



### 配置ehcache.xml

<**ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:noNamespaceSchemaLocation="/ehcache.xsd"  
 updateCheck="true" monitoring="autodetect"  
 dynamicConfig="true"**>  
  
 *<!-- 硬盘中的临时存放目录 -->* <**diskStore path="F:/EHCache"**/>  
   
 *<!--   
 name ：cache唯一标识   
 eternal ：缓存是否永久有效   
 maxElementsInMemory ：内存中最大缓存对象数   
 overflowToDisk(true,false)：缓存对象达到最大数后，将缓存写到硬盘中   
 diskPersistent ：硬盘持久化   
 timeToIdleSeconds ：缓存清除时间   
 timeToLiveSeconds ：缓存存活时间   
 memoryStoreEvictionPolicy ：缓存清空策略   
 1. FIFO ：first in first out 先讲先出   
 2. LFU ： Less Frequently Used 一直以来最少被使用的   
 3. LRU ：Least Recently Used 最近最少使用的   
 -->* <**defaultCache  
 maxEntriesLocalHeap="10000"  
 eternal="false"  
 timeToIdleSeconds="120"  
 timeToLiveSeconds="120"  
 diskSpoolBufferSizeMB="30"  
 maxEntriesLocalDisk="10000000"  
 diskExpiryThreadIntervalSeconds="120"  
 memoryStoreEvictionPolicy="LRU"**>  
 <**persistence strategy="localTempSwap"**/>  
 </**defaultCache**>  
</**ehcache**>

### 整合测试

在mapper.xml添加ehcache配置：

*<!-- 【开启二级缓存】 -->  
<!-- 单位：毫秒 -->*<**cache type="org.mybatis.caches.ehcache.EhcacheCache"**>  
 *<!-- 覆盖ehcacche.xml中的配置 -->* <**property name="timeToIdleSeconds" value="12000"**/>  
 <**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**>

## 二级缓存的应用场景

对查询频率高，变化频率低的数据建议使用二级缓存。

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和spring整合

## mybaits和spring整合的思路

1、让spring管理SqlSessionFactory

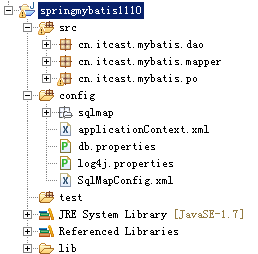
2、让spring管理mapper对象和dao。

使用spring和mybatis整合开发mapper代理及原始dao接口。

自动开启事务，自动关闭 sqlsession.

3、让spring管理数据源( 数据库连接池)

## 创建整合工程



## 加入jar包

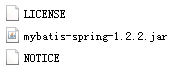
1、mybatis3.2.7本身的jar包

2、数据库驱动包

3、spring3.2.0

4、spring和mybatis整合包

从mybatis的官方下载spring和mybatis整合包



## log4j.properties

## SqlMapconfig.xml

mybatis配置文件：别名、settings，数据源不在这里配置

## applicationContext.xml

1、数据源（dbcp连接池）

2、SqlSessionFactory

3、mapper或dao

## 整合开发原始dao接口

### 配置SqlSessionFactory

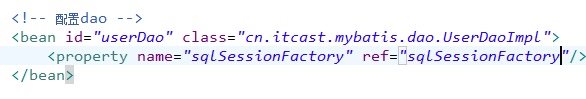
在 applicationContext.xml配置SqlSessionFactory

*<!-- 加载配置文件 -->*<**context:property-placeholder location="classpath:dbconfig.properties"**/>  
*<!-- 数据库连接池 -->*<**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}"**/>  
  
 *<!--initialSize 0 初始化连接:连接池启动时创建的初始化连接数量-->* <**property name="initialSize" value="5"**/>  
 *<!--maxActive 8 最大活动连接:连接池在同一时间能够分配的最大活动连接的数量-->* <**property name="maxActive" value="10"**/>  
 *<!--8 最大空闲连接:连接池中容许保持空闲状态的最大连接数量,超过的空闲连接将被释放-->* <**property name="maxIdle" value="5"**/>  
</**bean**>  
  
*<!-- sqlSessionFactory 整合Mybatis -->*<**bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean"**>  
 *<!-- 数据源 -->* <**property name="dataSource" ref="dataSource"**/>  
 *<!-- mybatis配置文件 -->* <**property name="configLocation" value="classpath:Mybatis/SqlMapConfig.xml"**/>  
</**bean**>

### 开发dao

*/\*\*  
 \* 用户Dao实现  
 \* create:2016-10-16 17:46  
 \*  
 \** ***@author*** *李欣  
 \** ***@version*** *1.0  
 \*/*@Repository  
**public class** UserDaoImpl **implements** UserDao {  
  
 @Resource  
 **private** SqlSessionTemplate **sqlSessionTemplate**;  
  
 **public** UserDaoImpl() {  
 }  
  
 **public** User findUserById(**int** id) {  
 **return sqlSessionTemplate**.selectOne(**"cn.qdgxy.mybatis.mapper.UserMapper.findUserById"**, id);  
 }  
  
}

### 配置 dao



使用注解，如11.7.2。

### 测试dao接口

*/\*\*  
 \* ${DESCRIPTION}<br>  
 \* create:2016-10-21 13:30  
 \*  
 \** ***@author*** *李欣  
 \** ***@version*** *${VERSION}  
 \*/*@RunWith(SpringJUnit4ClassRunner.**class**) *//表示继承了SpringJUnit4ClassRunner类*@ContextConfiguration(locations = {**"classpath:Spring/applicationContext.xml"**})  
**public class** UserServiceImplTest {  
  
 @Resource  
 **private** UserService **userService**;  
  
 @Test  
 **public void** findUserById() **throws** Exception {  
 System.***out***.println(**userService**.findUserById(1));  
 }  
  
}

## 整合开发mapper代理方法

### 开发mapper.xml和mapper.java



### 使用MapperFactoryBean

*<!-- 配置mapper：麻烦 -->*<**bean id="userMapper" class="org.mybatis.spring.mapper.MapperFactoryBean"**>  
 *<!-- MapperFactoryBean：用于生成mapper代理对象，指定接口类型 -->* <**property name="mapperInterface" value="cn.qdgxy.mybatis.mapper.UserMapper"**/>  
 <**property name="sqlSessionFactory" ref="sqlSessionFactory"**/>  
</**bean**>

使用此方法对于每个mapper都需要配置，比较繁琐。

### 使用MapperScannerConfigurer（扫描mapper）

*<!-- 扫描mapper：方便  
 MapperScannerConfigurer：mapper的扫描器，将包下边的mapper接口自动创建代理对象，  
 自动创建到spring容器中，bean的id是mapper的类名（首字母小写） -->*<**bean class="org.mybatis.spring.mapper.MapperScannerConfigurer"**>  
 *<!-- 配置扫描包的路径  
 如果要扫描多个包，中间使用半角逗号分隔 -->* <**property name="basePackage" value="cn.qdgxy.mybatis.mapper"**/>  
 *<!-- 使用sqlSessionFactoryBeanName -->* <**property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"**/>  
</**bean**>

使用扫描器自动扫描mapper，生成代理对象，比较方便。

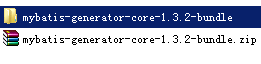
### 测试mapper接口

*/\*\*  
 \* ${DESCRIPTION}<br>  
 \* create:2016-10-21 19:50  
 \*  
 \** ***@author*** *李欣  
 \** ***@version*** *${VERSION}  
 \*/*@RunWith(SpringJUnit4ClassRunner.**class**) *//表示继承了SpringJUnit4ClassRunner类*@ContextConfiguration(locations = {**"classpath:Spring/applicationContext.xml"**})  
**public class** UserServiceByMapperImplTest {  
  
 @Resource(name = **"userServiceByMapperImpl"**)  
 **private** UserService **userService**;  
  
 @Test  
 **public void** findUserById() **throws** Exception {  
 System.***out***.println(**userService**.findUserById(1));  
 }  
  
}

# mybatis逆向工程 （MyBatis Generator）

## 什么是mybatis的逆向工程

mybatis官方为了提高开发效率，提高自动对单表生成sql，包括 ：mapper.xml、mapper.java、表名.java(po类)



在企业开发中通常是在设计阶段对表进行设计 、创建。

在开发阶段根据表结构创建对应的po类。

mybatis逆向工程的方向：由数据库表----》java代码

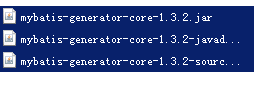
## 逆向工程 使用配置

运行逆向工程 方法：



本教程使用通过java程序运行逆向工程 。

逆向工程 运行所需要的jar包：



数据库驱动包

### xml配置

需要使用用配置的地方：

1、连接数据库的地址和驱动

*<!--1. 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://10.10.39.123:3306/mybatis"  
 userId="root"  
 password="1234"**>  
</**jdbcConnection**>  
  
*<!-- Oracle  
<jdbcConnection driverClass="oracle.jdbc.OracleDriver"  
 connectionURL="jdbc:oracle:thin:@127.0.0.1:1521:yycg"  
 userId="yycg"  
 password="yycg">  
</jdbcConnection> -->*

2、需要配置po类的包路径

*<!-- 2. targetProject：生成PO类的位置 -->*<**javaModelGenerator targetPackage="cn.qdgxy.mybatis.po" targetProject=".\src\main\java"**>  
 *<!-- 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**>

3、需要配置mapper包的路径

*<!-- 3.1 targetProject：mapper映射文件生成的位置 -->*<**sqlMapGenerator targetPackage="cn.qdgxy.mybatis.mapper" targetProject=".\src\main\java"**>  
 *<!-- enableSubPackages：是否让schema作为包的后缀 -->* <**property name="enableSubPackages" value="false"**/>  
</**sqlMapGenerator**>  
  
*<!-- 3.2 targetPackage：mapper接口生成的位置 -->*<**javaClientGenerator type="XMLMAPPER"  
 targetPackage="cn.qdgxy.mybatis.mapper"  
 targetProject=".\src\main\java"**>  
 *<!-- enableSubPackages：是否让schema作为包的后缀 -->* <**property name="enableSubPackages" value="false"**/>  
</**javaClientGenerator**>

4、指定数据表

*<!-- 4. 指定数据库表 -->*<**table tableName="user"**/>  
<**table tableName="orderDetail"**/>  
<**table tableName="product"**/>  
<**table tableName="orders"**/>  
  
*<!-- <table schema="" tableName="sys\_user"/>  
 <table schema="" tableName="sys\_role"/>  
 <table schema="" tableName="sys\_permission"/>  
 <table schema="" tableName="sys\_user\_role"/>  
 <table schema="" tableName="sys\_role\_permission"/>  
  
 &lt;!&ndash;有些表的字段需要指定java类型&ndash;&gt;  
 <table schema="" tableName="">  
 <columnOverride column="" javaType=""/>  
 </table>-->*

### java程序

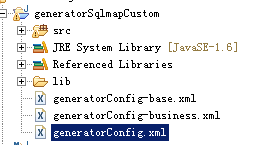
通过java程序生成mapper类、po类。。

**public class** GeneratorSqlmap {  
  
 **public void** generator() **throws** Exception {  
  
 List<String> warnings = **new** ArrayList<>();  
 **final boolean** overwrite = **true**;  
 *//指定逆向工程配置文件* URL url = GeneratorSqlmap.**class**.getClassLoader().getResource(**"Mybatis/generatorConfig.xml"**);  
 File configFile = **new** File(url.getFile());  
 ConfigurationParser cp = **new** ConfigurationParser(warnings);  
 Configuration config = cp.parseConfiguration(configFile);  
 DefaultShellCallback callback = **new** DefaultShellCallback(overwrite);  
 MyBatisGenerator myBatisGenerator = **new** MyBatisGenerator(config,  
 callback, warnings);  
 myBatisGenerator.generate(**null**);  
  
 }  
  
 **public static void** main(String[] args) **throws** Exception {  
 **try** {  
 GeneratorSqlmap generatorSqlmap = **new** GeneratorSqlmap();  
 generatorSqlmap.generator();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
}

## 使用逆向工程生成的代码

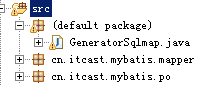
### 第一步配置generatorConfig.xml

参考：

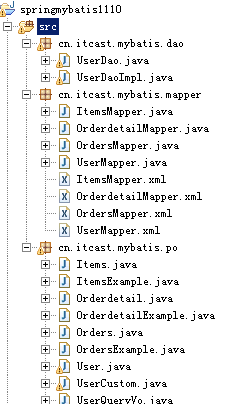


### 第二步配置执行java程序

执行java程序后，所生成的代码已经生成到工程中。



### 第三步将生成的代码拷贝到工程中



## 测试生成的代码

参考：

