# 课程计划

1. 高级结果映射（一对一、一对多、多对多）（重点）
2. 延迟加载
3. 查询缓存
4. Spring和mybatis的整合（重点）
5. 逆向工程

# 高级结果映射

## 数据模型分析

1. 明确每张表存储的信息
2. 明确每张表中关键字段（主键、外键、非空）
3. 明确数据库中表与表之间的外键关系
4. 明确业务中表与表的关系（建立在具体的业务）

用户和订单关系：

用户到订单：一个用户可以创建多个订单 一对多关系

订单到用户：一个订单只能由一个用户创建一对一关系

订单和订单明细关系：

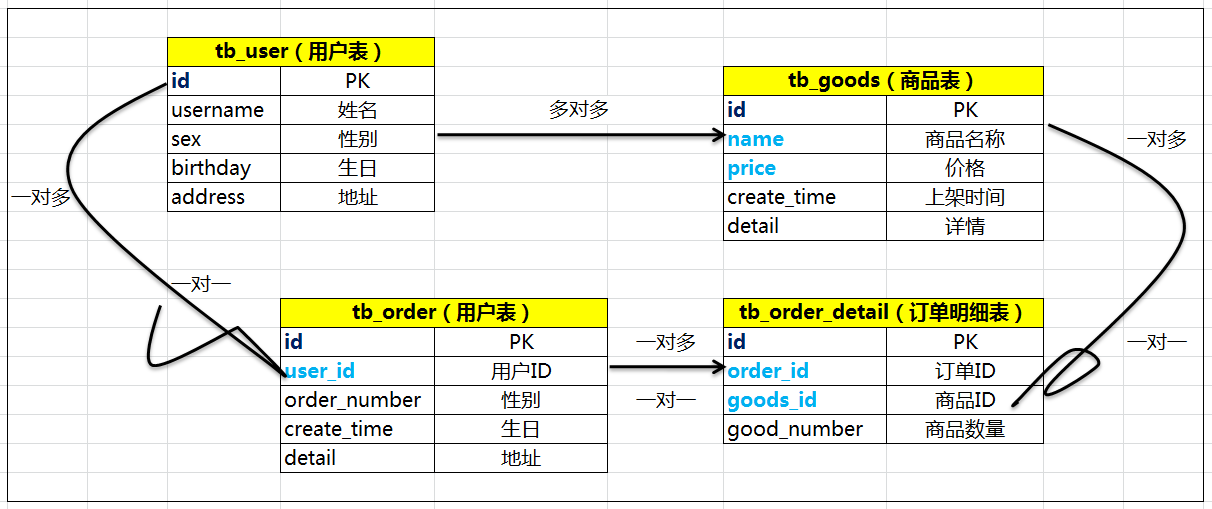
订单到订单明细：一个订单可以包括多个订单明细，因为一个订单可以购买多个商品，每个商品的购买信息在orderdetail记录，一对多关系

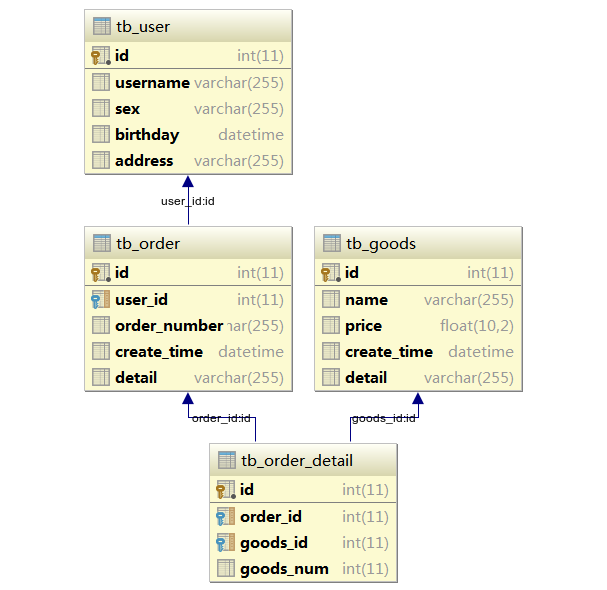
订单明细到订单：一个订单明细只能包括在一个订单中，一对一关系

订单明细和商品关系：

订单明细到商品：一个订单明细只对应一个商品信息，一对一关系

商品到订单明细:一个商品可以包括在多个订单明细，一对多关系





## 一对一映射

### 需求

查询订单信息，关联查询用户信息

### sql

主信息：tb\_order

从信息：tb\_user

**SELECT**tb\_order.id,  
tb\_order.user\_id,  
tb\_order.order\_number,  
tb\_user.username,  
tb\_user.sex  
**FROM** tb\_order, tb\_user  
**WHERE** tb\_order.user\_id = tb\_user.id

### resultType

#### 创建订单类

public class Order {  
private Integer id;  
 private Integer userId;  
 private String orderNumber;  
 private Date createTime;  
 private String detail;

#### 创建订单查询类

public class OrderQuery01 extends Order {  
private String username;  
 private String sex;  
  
 public String getUsername() {  
return username;  
}  
  
public void setUsername(String username) {  
this.username = username;  
}  
  
public String getSex() {  
return sex;  
}  
  
public void setSex(String sex) {  
this.sex = sex;  
}  
@Override  
public String toString() {  
return super.toString() + "OrderQuery{" +  
"username='" + username + '\'' +  
", sex='" + sex + '\'' +  
'}';  
}  
}

#### 创建OrderMapper接口

public interface OrderMapper {  
*/\*\*  
 \* 查询订单信息，关联查询用户信息  
\** ***@return*** *返回实体集合  
\*/*List<OrderQuery>findOrderAndUser();  
}

#### 创建OrderMapper.xml映射文件

**注意：如果使用resultType作为输出映射，数据库表字段和实体对象属性名称不一致的将获取不到值，如下：**

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
  
<mapper namespace="com.zhiyou100.mapper.OrderMapper">  
<!--一对一映射 返回类型为resultType-->  
<select id="findOrderAndUser" resultType="com.zhiyou100.model.OrderQuery">  
SELECT  
 tb\_order.id,  
 tb\_order.user\_id,  
 tb\_order.order\_number,  
 tb\_user.username,  
 tb\_user.sex  
 FROM tb\_order, tb\_user  
 WHERE tb\_order.user\_id = tb\_user.id  
</select>  
</mapper>

#### 创建mybatis-config.xml文件

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-config.dtd">  
<configuration>  
<!-- 加载属性文件 -->  
<properties resource="jdbc.properties"/>  
<!-- 自定义别名 -->  
<typeAliases>  
<!--单个别名定义-->  
<!-- <typeAlias type="com.zhiyou100.model.User" alias="User"/> -->  
<!-- 批量定义别名 (建议使用)-->  
<!-- package:指定包名称来为该包下的model类声明别名，默认的别名就是类型，首字符大小写均可-->  
<package name="com.zhiyou100.model"/>  
</typeAliases>  
<environments default="development">  
<environment id="development">  
<!-- 使用jdbc管理事务mybatis -->  
<transactionManager type="JDBC"/>  
<dataSource type="POOLED">  
<property name="driver" value="${jdbc.driverClassName}"/>  
<property name="url" value="${jdbc.url}"/>  
<property name="username" value="${jdbc.username}"/>  
<property name="password" value="${jdbc.password}"/>  
</dataSource>  
</environment>  
</environments>  
<!-- 加载映射文件 -->  
<mappers>  
<mapper resource="mapper/OrderMapper.xml"/>  
</mappers>  
</configuration>

#### 测试代码

public class OrderMapperTest {  
  
private SqlSessionFactory sqlSessionFactory;  
@Before  
public void setUp(){  
try {  
 InputStream inputStream = Resources.*getResourceAsStream*("mybatis-config.xml");  
sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
} catch (IOException e) {  
 e.printStackTrace();  
}  
 }  
  
@Test  
public void findOrderAndUser() throws Exception {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
OrderMapper orderMapper = sqlSession.getMapper(OrderMapper.class);  
List<OrderQuery01> list = orderMapper.findOrderAndUser();  
System.*out*.println(list);  
}  
}

#### 小结

使用resultType来进行一对一结果映射，查询出的列的个数和映射的属性的个数要一致。而且映射的属性要存在与一个大的对象中，它是一种平铺式的映射，即数据库查询出多少条记录，则映射成多少个对象。

**注意：如果使用resultType作为输出映射，数据库表字段和实体对象属性名称不一致的将获取不到值**

### resultMap

使用resultMap来进行一对一结果映射，它是将关联对象添加到主信息的对象中，具体说是对象嵌套对象的一种映射方式。

#### 创建OrderQuery02

public class OrderQuery02 extends Order {  
  
private User user;  
  
 public User getUser() {  
return user;  
}  
public void setUser(User user) {  
this.user = user;  
}  
@Override  
public String toString() {  
return super.toString() + "OrderQuery02{" +  
"user=" + user +  
'}';  
}  
}

#### 映射文件

<!--将数据库表字段和实体对象字段进行映射-->  
<resultMap id="orderAndUserResultMap" type="com.zhiyou100.model.OrderQuery02">  
<!-- 订单信息-->  
<id column="order\_id" property="id"/>  
<result column="user\_id" property="userId"/>  
<result column="order\_number" property="orderNumber"/>  
<result column="create\_time" property="createTime"/>  
<result column="detail" property="detail"/>  
<!-- 用户信息 一对一 -->  
<association property="user" javaType="com.zhiyou100.model.User">  
<!--注意 id 要和订单中的user\_id 对应-->  
<id column="user\_id" property="id"/>  
<result column="username" property="username"/>  
<result column="sex" property="sex"/>  
<result column="birthday" property="birthday"/>  
<result column="address" property="address"/>  
</association>  
</resultMap>  
<!--一对一映射 返回类型为resultMap-->  
<select id="findOrderAndUserRsMap" resultMap="orderAndUserResultMap">  
SELECT  
 tb\_order.id order\_id,  
 tb\_order.user\_id,  
 tb\_order.order\_number,  
 tb\_order.detail,

tb\_user.username,  
 tb\_user.sex,  
 tb\_user.birthday,  
 tb\_user.address

FROM tb\_order, tb\_user  
 WHERE tb\_order.user\_id = tb\_user.id  
</select>

#### Mapper接口

public interface OrderMapper {  
*/\*\*  
 \* 查询订单信息，关联查询用户信息  
\** ***@return*** *返回实体集合  
\*/*List<OrderQuery01>findOrderAndUser();  
  
*/\*\*  
 \* 查询订单信息，关联查询用户信息  
\** ***@return*** *返回实体集合  
\*/*List<OrderQuery02>findOrderAndUserRsMap();  
}

#### 测试代码

@Test  
public void findOrderAndUserRsMap() throws Exception {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
OrderMapper orderMapper = sqlSession.getMapper(OrderMapper.class);  
List<OrderQuery02> list = orderMapper.findOrderAndUserRsMap();  
System.*out*.println(list);  
}

### 小结

在一对一结果映射时，使用resultType更加简单方便，如果有特殊要求（对象嵌套对象）时，需要使用resultMap进行映射，比如：查询订单列表，然后在点击列表中的查看订单明细按钮，这个时候就需要使用resultMap进行结果映射。而resultType更适应于查询明细信息，比如，查询订单明细列表。

## 一对多映射

### 需求

查询订单信息，关联查询订单明细信息及用户信息

### Sql

主信息：tb\_order

从信息：tb\_order\_detail、tb\_user

**SELECT**tb\_order.id order\_id,  
tb\_order.user\_id,  
tb\_order.order\_number,  
tb\_order.create\_time,  
tb\_order.detail,

tb\_user.username,  
tb\_user.sex,  
tb\_user.birthday,  
tb\_user.address,

tb\_order\_detail.id detail\_id,  
tb\_order\_detail.goods\_id,  
tb\_order\_detail.goods\_num

**FROM** tb\_order, tb\_user, tb\_order\_detail  
**WHERE** tb\_order.user\_id = tb\_user.id  
**AND** tb\_order.id = tb\_order\_detail.order\_id

### 创建OrderQuery03类

public class OrderQuery03 extends Order {  
//用户信息  
private User user;  
//订单明细  
List<OrderDetail>orderDetailList;  
  
 public User getUser() {  
return user;  
}  
  
public void setUser(User user) {  
this.user = user;  
}  
  
public List<OrderDetail>getOrderDetailList() {  
return orderDetailList;  
}  
  
public void setOrderDetailList(List<OrderDetail> orderDetailList) {  
this.orderDetailList = orderDetailList;  
}  
  
@Override  
public String toString() {  
return super.toString() + "OrderQuery03{" +  
"user=" + user +  
", orderDetailList=" + orderDetailList +  
'}';  
}  
}

### 映射文件

<!--将数据库表字段和实体对象字段进行映射-->  
<resultMap id="orderAndDetailResultMap" type="com.zhiyou100.model.OrderQuery03">  
<!-- 订单信息-->  
<id column="order\_id" property="id"/>  
<result column="user\_id" property="userId"/>  
<result column="order\_number" property="orderNumber"/>  
<result column="create\_time" property="createTime"/>  
<result column="detail" property="detail"/>  
<!-- 用户信息 一对一 -->  
<association property="user" javaType="com.zhiyou100.model.User">  
<!--注意 id 要和订单中的user\_id 对应-->  
<id column="user\_id" property="id"/>  
<result column="username" property="username"/>  
<result column="sex" property="sex"/>  
<result column="birthday" property="birthday"/>  
<result column="address" property="address"/>  
</association>  
<collection property="orderDetailList" ofType="com.zhiyou100.model.OrderDetail">  
<id column="detail\_id" property="id"/>  
<result column="goods\_id" property="goodsId"/>  
<result column="order\_id" property="orderId"/>  
<result column="goods\_num" property="goodsNum"/>  
</collection>  
</resultMap>  
  
<select id="findOrderAndDetailResultMap" resultMap="orderAndDetailResultMap">  
SELECT  
 tb\_order.id order\_id,  
 tb\_order.user\_id,  
 tb\_order.order\_number,  
 tb\_order.create\_time,  
 tb\_order.detail,  
 tb\_user.username,  
 tb\_user.sex,  
 tb\_user.birthday,  
 tb\_user.address,  
 tb\_order\_detail.id detail\_id,  
 tb\_order\_detail.goods\_id,  
 tb\_order\_detail.goods\_num  
 FROM tb\_order, tb\_user, tb\_order\_detail  
 WHERE tb\_order.user\_id = tb\_user.id  
 AND tb\_order.id = tb\_order\_detail.order\_id  
</select>

### Mapper接口

*/\*\*  
 \* 查询订单信息，关联查询订单明细和用户信息  
\** ***@return*** *返回实体集合  
\*/*List<OrderQuery03>findOrderAndDetailResultMap();

### 测试代码

@Test  
public void findOrderAndDetailResultMap() throws Exception {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
OrderMapper orderMapper = sqlSession.getMapper(OrderMapper.class);  
List<OrderQuery03> list = orderMapper.findOrderAndDetailResultMap();  
System.*out*.println(list);  
}

## 多对多映射

多对多映射是一对多映射的特例

### 需求

查询用户信息，关联查询该用户购买的商品信息

### Sql

主信息：tb\_user

从信息：tb\_goods、tb\_order、tb\_order\_detail

**SELECT**tb\_user.id user\_id,  
tb\_user.username,  
tb\_user.sex,  
tb\_user.birthday,  
tb\_user.address,  
  
tb\_order.id order\_id,  
tb\_order.order\_number,  
tb\_order.create\_time,  
tb\_order.detail,  
  
tb\_order\_detail.id detail\_id,  
tb\_order\_detail.goods\_id,  
tb\_order\_detail.goods\_num,  
  
tb\_goods.name,  
tb\_goods.price,  
tb\_goods.create\_time goods\_create\_time,  
tb\_goods.detail goods\_detail  
  
**FROM** tb\_order, tb\_user, tb\_order\_detail, tb\_goods  
**WHERE** tb\_order.user\_id = tb\_user.id  
**AND** tb\_order.id = tb\_order\_detail.order\_id  
**AND** tb\_order\_detail.id = tb\_goods.id

### 修改User类

在User类中添加List<Order> order;

public class User {  
  
private Long id;  
 private String username;  
 private String sex;  
 private Date birthday;  
 private String address;

//用户对应的订单信息 一对多  
List<Order>order;  
  
 public Long getId() {  
return id;  
}  
  
public void setId(Long id) {  
this.id = id;  
}  
  
public String getUsername() {  
return username;  
}  
  
public void setUsername(String username) {  
this.username = username;  
}  
  
public String getSex() {  
return sex;  
}  
  
public void setSex(String sex) {  
this.sex = sex;  
}  
  
public Date getBirthday() {  
return birthday;  
}  
  
public void setBirthday(Date birthday) {  
this.birthday = birthday;  
}  
  
public String getAddress() {  
return address;  
}  
  
public void setAddress(String address) {  
this.address = address;  
}  
  
public List<Order>getOrder() {  
return order;  
}  
  
public void setOrder(List<Order> order) {  
this.order = order;  
}  
  
@Override  
public String toString() {  
return "User{" +  
"id=" + id +  
", username='" + username + '\'' +  
", sex='" + sex + '\'' +  
", birthday=" + birthday +  
", address='" + address + '\'' +  
", order=" + order +  
'}';  
}  
}

### 修改Order类

在Order类中添加List<OrderDetail> detailList;

public class Order {  
private Integer id;  
 private Integer userId;  
 private String orderNumber;  
 private Date createTime;  
 private String detail;

//订单对应的订单明细信息 一对多  
List<OrderDetail>detailList;  
  
 public Integer getId() {  
return id;  
}  
  
public void setId(Integer id) {  
this.id = id;  
}  
  
public Integer getUserId() {  
return userId;  
}  
  
public void setUserId(Integer userId) {  
this.userId = userId;  
}  
  
public String getOrderNumber() {  
return orderNumber;  
}  
  
public void setOrderNumber(String orderNumber) {  
this.orderNumber = orderNumber;  
}  
  
public Date getCreateTime() {  
return createTime;  
}  
  
public void setCreateTime(Date createTime) {  
this.createTime = createTime;  
}  
  
public String getDetail() {  
return detail;  
}  
  
public void setDetail(String detail) {  
this.detail = detail;  
}  
  
public List<OrderDetail>getDetailList() {  
return detailList;  
}  
  
public void setDetailList(List<OrderDetail> detailList) {  
this.detailList = detailList;  
}  
  
@Override  
public String toString() {  
return "Order{" +  
"id=" + id +  
", userId=" + userId +  
", orderNumber='" + orderNumber + '\'' +  
", createTime=" + createTime +  
", detail='" + detail + '\'' +  
", detailList=" + detailList +  
'}';  
}  
}

### 修改OrderDetail类

在OrderDetail中添加Goodsgoods；

public class OrderDetail {  
private Integer id;  
 private Integer orderId;  
 private Integer goodsId;  
 private Integer goodsNum;  
  
//订单明细对应的商品信息 一对一  
private Goods goods;  
  
 public Integer getId() {  
return id;  
}  
  
public void setId(Integer id) {  
this.id = id;  
}  
  
public Integer getOrderId() {  
return orderId;  
}  
  
public void setOrderId(Integer orderId) {  
this.orderId = orderId;  
}  
  
public Integer getGoodsId() {  
return goodsId;  
}  
  
public void setGoodsId(Integer goodsId) {  
this.goodsId = goodsId;  
}  
  
public Integer getGoodsNum() {  
return goodsNum;  
}  
  
public void setGoodsNum(Integer goodsNum) {  
this.goodsNum = goodsNum;  
}  
  
public Goods getGoods() {  
return goods;  
}  
  
public void setGoods(Goods goods) {  
this.goods = goods;  
}  
  
@Override  
public String toString() {  
return "OrderDetail{" +  
"id=" + id +  
", orderId=" + orderId +  
", goodsId=" + goodsId +  
", goodsNum=" + goodsNum +  
", goods=" + goods +  
'}';  
}  
}

### 创建UserMapper接口

public interface UserMapper {  
*/\*\*  
 \* 查询用户信息，关联查询该用户购买的商品信息  
\** ***@return****\*/*List<User>findUserAndGoodsResultMap();  
}

### 创建UserMapper.xml映射文件

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper namespace="com.zhiyou100.mapper.UserMapper">  
<resultMap id="UserResultMap" type="com.zhiyou100.model.User">  
<!--用户信息-->  
<id column="user\_id" property="id"/>  
<result column="username" property="username"/>  
<result column="sex" property="sex"/>  
<result column="birthday" property="birthday"/>  
<result column="address" property="address"/>  
<!-- 订单信息 -->  
<collection property="order" ofType="com.zhiyou100.model.Order">  
<id column="order\_id" property="id"/>  
<result column="user\_id" property="userId"/>  
<result column="order\_number" property="orderNumber"/>  
<result column="create\_time" property="createTime"/>  
<result column="order\_detail" property="detail"/>  
<!--订单明细-->  
<collection property="detailList" ofType="com.zhiyou100.model.OrderDetail">  
<id column="detail\_id" property="id"/>  
<result column="goods\_id" property="goodsId"/>  
<result column="order\_id" property="orderId"/>  
<result column="goods\_num" property="goodsNum"/>  
<!--商品信息-->  
<association property="goods" javaType="com.zhiyou100.model.Goods">  
<id column="detail\_id" property="id"/>  
<result column="name" property="name"/>  
<result column="price" property="price"/>  
<result column="goods\_create\_time" property="createTime"/>  
<result column="goods\_detail" property="detail"/>  
</association>  
</collection>  
</collection>  
</resultMap>  
<select id="findUserAndGoodsResultMap" resultMap="UserResultMap">  
SELECT  
 tb\_user.id user\_id,  
 tb\_user.username,  
 tb\_user.sex,  
 tb\_user.birthday,  
 tb\_user.address,  
  
 tb\_order.id order\_id,  
 tb\_order.order\_number,  
 tb\_order.create\_time,  
 tb\_order.detail order\_detail,  
  
 tb\_order\_detail.id detail\_id,  
 tb\_order\_detail.goods\_id,  
 tb\_order\_detail.goods\_num,  
  
 tb\_goods.name,  
 tb\_goods.price,  
 tb\_goods.create\_time goods\_create\_time,  
 tb\_goods.detail goods\_detail  
  
 FROM tb\_order, tb\_user, tb\_order\_detail, tb\_goods  
 WHERE tb\_order.user\_id = tb\_user.id  
 AND tb\_order.id = tb\_order\_detail.order\_id  
 AND tb\_order\_detail.id = tb\_goods.id  
</select>  
</mapper>

### 添加映射文件到mybatis-config.xml

<!-- 加载映射文件 -->  
<mappers>  
<mapper resource="mapper/OrderMapper.xml"/>

<!--添加USER的映射文件-->  
<mapper resource="mapper/UserMapper.xml"/>  
</mappers>

### 测试代码

public class UserMapperImplTest {  
  
private SqlSessionFactory sqlSessionFactory;  
@Before  
public void setUp() {  
try {  
 InputStream inputStream = Resources.*getResourceAsStream*("mybatis-config.xml");  
sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
} catch (IOException e) {  
 e.printStackTrace();  
}  
 }  
  
@Test  
public void findUserAndGoodsResultMap() throws Exception {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
List<User> list = userMapper.findUserAndGoodsResultMap();  
System.*out*.println(list);  
sqlSession.close();  
}  
}

# 延迟加载

## 什么是延迟加载

延迟加载又叫懒加载，也叫按需加载。也就是说先加载主信息，在需要的时候，再去加载从信息。

在mybatis中，resultMap标签的association标签和collection标签具有延迟加载的功能。

## 需求

查询订单信息，关联查询用户信息

1. 创建一个statement来查询用户信息
2. 创建一个statement来查询订单信息

## 创建OrderQuery04类

public class OrderQuery04 extends Order {  
  
//用户信息  
private User user;  
  
 public User getUser() {  
return user;  
}  
  
public void setUser(User user) {  
this.user = user;  
}  
  
@Override  
public String toString() {  
return "OrderQuery04{" +  
"user=" + user +  
'}';  
}  
}

## Mapper接口

1. UserMapper接口添加根据用户id查询用户接口

public interface UserMapper {  
*/\*\*  
 \* 查询用户信息，关联查询该用户购买的商品信息  
\** ***@return****\*/*List<User>findUserAndGoodsResultMap();  
  
*/\*\*  
 \* 根据用户id查询用户  
\** ***@param*** *id  
\** ***@return****\*/*User findUserById(Long id);  
}

1. OrderMapper接口添加查询订单关联查询用户接口（用户信息延迟加载）

public interface OrderMapper {  
*/\*\*  
 \* 查询订单信息，关联查询用户 用户信息延迟加载  
\** ***@return****\*/*List<OrderQuery04>findOrderAndUserLazyLoading();  
*/\*\*  
 \* 查询订单信息，关联查询用户信息  
\** ***@return*** *返回实体集合  
\*/*List<OrderQuery01>findOrderAndUser();  
*/\*\*  
 \* 查询订单信息，关联查询用户信息  
\** ***@return*** *返回实体集合  
\*/*List<OrderQuery02>findOrderAndUserRsMap();  
*/\*\*  
 \* 查询订单信息，关联查询订单明细和用户信息  
\** ***@return*** *返回实体集合  
\*/*List<OrderQuery03>findOrderAndDetailResultMap();  
}

## 映射文件

1. 在UserMapper.xml添加根据id查询用户的statement

<!--根据id查询用户信息-->  
<select id="findUserById" parameterType="java.lang.Long" resultType="com.zhiyou100.model.User">  
SELECT \* FROM tb\_user WHERE id = #{id}  
</select>

1. 在OrderMapper.xml添加查询订单延迟加载查询用户的statement

<!--延迟加载-->  
<resultMap id="lazyLoadingResultMap" type="com.zhiyou100.model.OrderQuery04">  
<!--订单信息-->  
<id column="id" property="id"/>  
<result column="user\_id" property="userId"/>  
<result column="order\_number" property="orderNumber"/>  
<result column="create\_time" property="createTime"/>  
<result column="detail" property="detail"/>  
<!--用户信息-->  
<!--  
 select：指定关联查询的查询statement (即查询用户的statement的id)

然后将查询结果封装到property属性指定的变量中  
 column：clolum指定列的值作为com.zhiyou100.mapper.UserMapper.findUserById()的查询参数  
-->  
<association property="user" select="com.zhiyou100.mapper.UserMapper.findUserById" column="user\_id"/>  
</resultMap>  
<select id="findOrderAndUserLazyLoading" resultMap="lazyLoadingResultMap">  
SELECT \* FROM tb\_order  
</select>

## 设置延迟加载

在mybatis-config.xml中，配置settings标签

<!-- 加载属性文件 -->  
<properties resource="jdbc.properties"/>  
<settings>  
<!-- 延迟加载 默认false -->  
<setting name="lazyLoadingEnabled" value="true" />  
<!-- 积极加载 默认true -->  
<setting name="aggressiveLazyLoading" value="false" />  
</settings>

## 测试代码

@Test  
public void findOrderAndUserLazyLoading() throws Exception {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
OrderMapper orderMapper = sqlSession.getMapper(OrderMapper.class);  
List<OrderQuery04> list = orderMapper.findOrderAndUserLazyLoading();  
 for (OrderQuery04 order : list){  
//按需加载 需要访问用户信息的时候才去数据库加载  
System.*out*.println(order.getUser());  
}  
 sqlSession.close();  
}

## 测试结果

1. 开启延迟加载

2016-12-02 14:18:39,641 [main] DEBUG [com.zhiyou100.mapper.OrderMapper.findOrderAndUserLazyLoading] - ==> Preparing: SELECT \* FROM tb\_order   
2016-12-02 14:18:39,687 [main] DEBUG [com.zhiyou100.mapper.OrderMapper.findOrderAndUserLazyLoading] - ==> Parameters:   
2016-12-02 14:18:39,819 [main] DEBUG [com.zhiyou100.mapper.OrderMapper.findOrderAndUserLazyLoading] - <== Total: 2  
2016-12-02 14:18:39,819 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Preparing: SELECT \* FROM tb\_user WHERE id = ?   
2016-12-02 14:18:39,820 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Parameters: 1(Long)  
2016-12-02 14:18:39,823 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - <== Total: 1  
User{id=1, username='张三', sex='男', birthday=Mon Nov 21 00:10:36 CST 2016, address='河南安阳', order=null}  
2016-12-02 14:18:39,826 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Preparing: SELECT \* FROM tb\_user WHERE id = ?   
2016-12-02 14:18:39,827 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Parameters: 2(Long)  
2016-12-02 14:18:39,828 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - <== Total: 1  
User{id=2, username='李四', sex='女', birthday=Tue Nov 22 14:22:11 CST 2016, address='河南安阳', order=null}

1. 关闭延迟加载

2016-12-02 14:23:50,710 [main] DEBUG [com.zhiyou100.mapper.OrderMapper.findOrderAndUserLazyLoading] - ==> Preparing: SELECT \* FROM tb\_order   
2016-12-02 14:23:50,788 [main] DEBUG [com.zhiyou100.mapper.OrderMapper.findOrderAndUserLazyLoading] - ==> Parameters:   
2016-12-02 14:23:50,819 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ====> Preparing: SELECT \* FROM tb\_user WHERE id = ?   
2016-12-02 14:23:50,819 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ====> Parameters: 1(Long)  
2016-12-02 14:23:50,819 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - <==== Total: 1  
2016-12-02 14:23:50,819 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ====> Preparing: SELECT \* FROM tb\_user WHERE id = ?   
2016-12-02 14:23:50,819 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ====> Parameters: 2(Long)  
2016-12-02 14:23:50,819 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - <==== Total: 1  
2016-12-02 14:23:50,819 [main] DEBUG [com.zhiyou100.mapper.OrderMapper.findOrderAndUserLazyLoading] - <== Total: 2  
User{id=1, username='张三', sex='男', birthday=Mon Nov 21 00:10:36 CST 2016, address='河南安阳', order=null}  
User{id=2, username='李四', sex='女', birthday=Tue Nov 22 14:22:11 CST 2016, address='河南安阳', order=null}

# 查询缓存

## Mybatis的缓存理解

sqlSession1

sqlSession 2

sqlSession3...

一级缓存

一级缓存

一级缓存

Mapper(namespace)二级缓存

Mybatis的缓存，包括一级缓存和二级缓存

一级缓存指的就是sqlsession，在sqlsession中有一个数据区域，是map结构，这个区域就是一级缓存区域。一级缓存中的key是由sql语句、条件、statement等信息组成一个唯一值。一级缓存中的value，就是查询出的结果对象。

二级缓存指的就是同一个namespace下的mapper，二级缓存中，也有一个map结构，这个区域就是一级缓存区域。一级缓存中的key是由sql语句、条件、statement等信息组成一个唯一值。一级缓存中的value，就是查询出的结果对象。

一级缓存是默认使用的。

二级缓存需要手动开启。

## 一级缓存

### 原理

第一次查询id为1的用户

SqlSession

一级缓存区域

Map

SqlSession

写入

修改、添加、删除用户执行commit

清空

第二次查询id为1的用户

读取

### 测试1

*/\*\*  
 \* 测试一级缓存  
\** ***@throws*** *Exception  
 \*/*@Test  
public void testOneLevelCache01() throws Exception {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
//第一次查询 从数据库读取  
System.*out*.println("第一次查询 从数据库读取");  
User user1 = userMapper.findUserById(1L);  
System.*out*.println(user1);  
  
//第二次查询 从缓存读取  
System.*out*.println("第二次查询 从缓存读取");  
User user2 = userMapper.findUserById(1L);  
System.*out*.println(user2);  
  
sqlSession.close();  
}

### 测试结果1

第一次查询 从数据库读取  
2016-12-02 14:29:39,557 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Opening JDBC Connection  
2016-12-02 14:29:39,838 [main] DEBUG [org.apache.ibatis.datasource.pooled.PooledDataSource] - Created connection 410080301.  
2016-12-02 14:29:39,838 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1871542d]  
2016-12-02 14:29:39,838 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Preparing: SELECT \* FROM tb\_user WHERE id = ?   
2016-12-02 14:29:39,885 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Parameters: 1(Long)  
2016-12-02 14:29:39,931 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - <== Total: 1  
User{id=1, username='张三', sex='男', birthday=Mon Nov 21 00:10:36 CST 2016, address='河南安阳', order=null}  
第二次查询 从缓存读取  
User{id=1, username='张三', sex='男', birthday=Mon Nov 21 00:10:36 CST 2016, address='河南安阳', order=null}

### 测试2

@Test  
public void testOneLevelCache02() throws Exception {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
//第一次查询 从数据库读取  
System.*out*.println("第一次查询 从数据库读取");  
User user1 = userMapper.findUserById(1L);  
System.*out*.println(user1);  
  
sqlSession.commit();//执行commit时，一级缓存将被清空  
  
//第二次查询 从缓存读取  
System.*out*.println("第二次查询 从数据库读取");  
User user2 = userMapper.findUserById(1L);  
System.*out*.println(user2);  
sqlSession.close();  
}

### 测试结果2

第一次查询 从数据库读取  
2016-12-02 14:34:59,663 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Opening JDBC Connection  
2016-12-02 14:34:59,913 [main] DEBUG [org.apache.ibatis.datasource.pooled.PooledDataSource] - Created connection 926939869.  
2016-12-02 14:34:59,913 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@373ffadd]  
2016-12-02 14:34:59,913 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Preparing: SELECT \* FROM tb\_user WHERE id = ?   
2016-12-02 14:34:59,960 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Parameters: 1(Long)  
2016-12-02 14:34:59,991 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - <== Total: 1  
User{id=1, username='张三', sex='男', birthday=Mon Nov 21 00:10:36 CST 2016, address='河南安阳', order=null}  
执行 sqlSession.commit() 一级缓存被清空  
第二次查询 还是从数据库读取  
2016-12-02 14:34:59,991 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Preparing: SELECT \* FROM tb\_user WHERE id = ?   
2016-12-02 14:34:59,991 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Parameters: 1(Long)  
2016-12-02 14:34:59,991 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - <== Total: 1  
User{id=1, username='张三', sex='男', birthday=Mon Nov 21 00:10:36 CST 2016, address='河南安阳', order=null}

## 二级缓存

### 原理



### 开启二级缓存

1. 开启二级缓存的总开关

<settings>  
<!-- 延迟加载 默认false -->  
<setting name="lazyLoadingEnabled" value="true" />  
<!-- 积极加载 默认true -->  
<setting name="aggressiveLazyLoading" value="false" />  
<!--二级缓存的总开关 默认为false-->  
<setting name="cacheEnabled" value="true"/>  
</settings>

1. 在mapper映射文件中开启二级缓存

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper namespace="com.zhiyou100.mapper.UserMapper">  
<cache/>  
<!--根据id查询用户信息-->  
<select id="findUserById" parameterType="java.lang.Long" resultType="com.zhiyou100.model.User">  
SELECT \* FROM tb\_user WHERE id = #{id}  
</select>

### 序列化

使用二级缓存需要对查询映射对象实现序列化

public class User implements Serializable {  
private Long id;  
 private String username;  
 private String sex;  
 private Date birthday;  
 private String address;  
//用户对应的订单信息 一对多  
List<Order>order;

### 测试1

@Test  
public void testTwoLevelCache01() throws Exception {  
 SqlSession sqlSession1 = sqlSessionFactory.openSession();  
SqlSession sqlSession2 = sqlSessionFactory.openSession();  
UserMapper userMapper1 = sqlSession1.getMapper(UserMapper.class);  
UserMapper userMapper2 = sqlSession2.getMapper(UserMapper.class);  
//第一次查询 从数据库读取  
System.*out*.println("第一次查询 从数据库读取");  
User user1 = userMapper1.findUserById(1L);  
System.*out*.println(user1);  
//在close()的时候才会把缓存写入到二级缓存  
sqlSession1.close();  
  
//第二次查询 从缓存读取  
System.*out*.println("第二次查询 从缓存读取");  
User user2 = userMapper2.findUserById(1L);  
System.*out*.println(user2);  
sqlSession2.close();  
}

### 测试结果1

第一次查询 从数据库读取  
2016-12-02 14:45:05,056 [main] DEBUG [com.zhiyou100.mapper.UserMapper] - Cache Hit Ratio [com.zhiyou100.mapper.UserMapper]: 0.0  
2016-12-02 14:45:05,056 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Opening JDBC Connection  
2016-12-02 14:45:05,368 [main] DEBUG [org.apache.ibatis.datasource.pooled.PooledDataSource] - Created connection 667467684.  
2016-12-02 14:45:05,383 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@27c8bfa4]  
2016-12-02 14:45:05,383 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Preparing: SELECT \* FROM tb\_user WHERE id = ?   
2016-12-02 14:45:05,430 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Parameters: 1(Long)  
2016-12-02 14:45:05,493 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - <== Total: 1  
User{id=1, username='张三', sex='男', birthday=Mon Nov 21 00:10:36 CST 2016, address='河南安阳', order=null}  
2016-12-02 14:45:05,493 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@27c8bfa4]  
2016-12-02 14:45:05,508 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@27c8bfa4]  
2016-12-02 14:45:05,508 [main] DEBUG [org.apache.ibatis.datasource.pooled.PooledDataSource] - Returned connection 667467684 to pool.  
第二次查询 从缓存读取  
2016-12-02 14:45:05,508 [main] DEBUG [com.zhiyou100.mapper.UserMapper] - Cache Hit Ratio [com.zhiyou100.mapper.UserMapper]: 0.5  
User{id=1, username='张三', sex='男', birthday=Mon Nov 21 00:10:36 CST 2016, address='河南安阳', order=null}

### 测试2

**自行添加一个updateUser()接口到UserMapper 并配置对应的SQL**

@Test  
public void testTwoLevelCache02() 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 从数据库读取  
System.*out*.println("第一次查询 从数据库读取");  
User user1 = userMapper1.findUserById(1L);  
System.*out*.println(user1);  
//在close()的时候才会把缓存写入到二级缓存  
sqlSession1.close();  
  
User user3 = userMapper3.findUserById(2L);  
userMapper3.updateUser(user3);  
sqlSession3.commit();  
sqlSession3.close();  
System.*out*.println("执行 update() 二级缓存被清空");  
  
//第二次查询 从缓存读取  
System.*out*.println("第二次查询 缓存命中率为0 从数据库读取");  
User user2 = userMapper2.findUserById(1L);  
System.*out*.println(user2);  
sqlSession2.close();  
}

### 测试结果2

第一次查询 从数据库读取  
2016-12-02 15:01:51,878 [main] DEBUG [com.zhiyou100.mapper.UserMapper] - Cache Hit Ratio [com.zhiyou100.mapper.UserMapper]: 0.0  
2016-12-02 15:01:51,883 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Opening JDBC Connection  
2016-12-02 15:01:52,167 [main] DEBUG [org.apache.ibatis.datasource.pooled.PooledDataSource] - Created connection 1169856126.  
2016-12-02 15:01:52,168 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@45ba967e]  
2016-12-02 15:01:52,172 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Preparing: SELECT \* FROM tb\_user WHERE id = ?   
2016-12-02 15:01:52,237 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Parameters: 1(Long)  
2016-12-02 15:01:52,267 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - <== Total: 1  
User{id=1, username='张三', sex='男', birthday=Mon Nov 21 00:10:36 CST 2016, address='河南安阳', order=null}  
2016-12-02 15:01:52,285 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@45ba967e]  
2016-12-02 15:01:52,299 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@45ba967e]  
2016-12-02 15:01:52,299 [main] DEBUG [org.apache.ibatis.datasource.pooled.PooledDataSource] - Returned connection 1169856126 to pool.  
2016-12-02 15:01:52,299 [main] DEBUG [com.zhiyou100.mapper.UserMapper] - Cache Hit Ratio [com.zhiyou100.mapper.UserMapper]: 0.0  
2016-12-02 15:01:52,299 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Opening JDBC Connection  
2016-12-02 15:01:52,299 [main] DEBUG [org.apache.ibatis.datasource.pooled.PooledDataSource] - Checked out connection 1169856126 from pool.  
2016-12-02 15:01:52,299 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@45ba967e]  
2016-12-02 15:01:52,301 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Preparing: SELECT \* FROM tb\_user WHERE id = ?   
2016-12-02 15:01:52,301 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Parameters: 2(Long)  
2016-12-02 15:01:52,313 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - <== Total: 1  
2016-12-02 15:01:52,314 [main] DEBUG [com.zhiyou100.mapper.UserMapper.updateUser] - ==> Preparing: UPDATE tb\_user SET id = ?, username = ?, sex = ?, birthday = ?, address = ? WHERE id = ?   
2016-12-02 15:01:52,319 [main] DEBUG [com.zhiyou100.mapper.UserMapper.updateUser] - ==> Parameters: 2(Long), 李四(String), 女(String), 2016-11-22 14:22:11.0(Timestamp), 河南安阳(String), 2(Long)  
2016-12-02 15:01:52,319 [main] DEBUG [com.zhiyou100.mapper.UserMapper.updateUser] - <== Updates: 1  
2016-12-02 15:01:52,319 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Committing JDBC Connection [com.mysql.jdbc.JDBC4Connection@45ba967e]  
2016-12-02 15:01:52,323 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@45ba967e]  
2016-12-02 15:01:52,324 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@45ba967e]  
2016-12-02 15:01:52,324 [main] DEBUG [org.apache.ibatis.datasource.pooled.PooledDataSource] - Returned connection 1169856126 to pool.  
执行 update() 二级缓存被清空  
第二次查询 缓存命中率为0 从数据库读取  
2016-12-02 15:01:52,325 [main] DEBUG [com.zhiyou100.mapper.UserMapper] - Cache Hit Ratio [com.zhiyou100.mapper.UserMapper]: 0.0  
2016-12-02 15:01:52,325 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Opening JDBC Connection  
2016-12-02 15:01:52,325 [main] DEBUG [org.apache.ibatis.datasource.pooled.PooledDataSource] - Checked out connection 1169856126 from pool.  
2016-12-02 15:01:52,325 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@45ba967e]  
2016-12-02 15:01:52,326 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Preparing: SELECT \* FROM tb\_user WHERE id = ?   
2016-12-02 15:01:52,326 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Parameters: 1(Long)  
2016-12-02 15:01:52,328 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - <== Total: 1  
User{id=1, username='张三', sex='男', birthday=Mon Nov 21 00:10:36 CST 2016, address='河南安阳', order=null}

### 禁用缓存

默认值是true

<!--根据id查询用户信息-->  
<select id="findUserById" parameterType="java.lang.Long" resultType="com.zhiyou100.model.User" useCache="false">  
SELECT \* FROM tb\_user WHERE id = #{id}  
</select>

### 刷新缓存

<!--根据id查询用户信息-->  
<!--  
 flushCache 在select中默认为false 在增 删 改中默认是ture  
-->  
<select id="findUserById" parameterType="java.lang.Long" resultType="com.zhiyou100.model.User" flushCache="false">  
SELECT \* FROM tb\_user WHERE id = #{id}  
</select>

### 整合ehcache

Mybatis本身是一个持久层框架，它不是专门的缓存框架，所以它对缓存的实现不够好，不能支持分布式。

Ehcache是一个分布式的缓存框架。

#### 什么是分布式

系统为了提高性能，通常会对系统采用分布式部署（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。

#### 整合思路

Cache是一个接口，它的默认实现是mybatis的PerpetualCache。如果想整合mybatis的二级缓存，那么实现Cache接口即可。

public interface Cache {  
  
*/\*\*  
 \** ***@return*** *The identifier of this cache  
 \*/*String getId();  
  
*/\*\*  
 \** ***@param*** *key Can be any object but usually it is a {****@link*** *CacheKey}  
 \** ***@param*** *value The result of a select.  
 \*/*void putObject(Object key, Object value);  
  
*/\*\*  
 \** ***@param*** *key The key  
 \** ***@return*** *The object stored in the cache.  
 \*/*Object getObject(Object key);  
  
*/\*\*  
 \* As of 3.3.0 this method is only called during a rollback   
 \* for any previous value that was missing in the cache.  
 \* This lets any blocking cache to release the lock that   
 \* may have previously put on the key.  
 \* A blocking cache puts a lock when a value is null   
 \* and releases it when the value is back again.  
 \* This way other threads will wait for the value to be   
 \* available instead of hitting the database.  
 \*  
 \*   
 \** ***@param*** *key The key  
 \** ***@return*** *Not used  
 \*/*Object removeObject(Object key);  
  
*/\*\*  
 \* Clears this cache instance  
 \*/*void clear();  
  
*/\*\*  
 \* Optional. This method is not called by the core.  
 \*   
 \** ***@return*** *The number of elements stored in the cache (not its capacity).  
 \*/*int getSize();  
  
*/\*\*   
 \* Optional. As of 3.2.6 this method is no longer called by the core.  
 \*   
 \* Any locking needed by the cache must be provided internally by the cache provider.  
 \*   
 \** ***@return*** *A ReadWriteLock   
 \*/*ReadWriteLock getReadWriteLock();  
  
}

#### 添加jar包

<!-- ehcache -->  
<dependency>  
<groupId>net.sf.ehcache</groupId>  
<artifactId>ehcache</artifactId>  
<version>2.0.0</version>  
</dependency>  
<dependency>  
<groupId>org.mybatis</groupId>  
<artifactId>mybatis-ehcache</artifactId>  
<version>1.0.0</version>  
</dependency>

#### 设置映射文件中cache标签的type值为ehcache的实现类

<cache type="org.mybatis.caches.ehcache.EhcacheCache"/>  
<!--根据id查询用户信息-->  
<!--  
 flushCache 在select中默认为false 在增 删 改中默认是ture  
-->  
<select id="findUserById" parameterType="java.lang.Long" resultType="com.zhiyou100.model.User" flushCache="false">  
SELECT \* FROM tb\_user WHERE id = #{id}  
</select>

#### 添加ehcache的配置文件

在src/main/resources下创建cache文件夹，在文件夹下创建ehcache.xml

<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../config/ehcache.xsd">  
  
<diskStore path="java.io.tmpdir"/>  
  
<!--  
 Mandatory Default Cache configuration. These settings will be applied to caches  
 created programmtically using CacheManager.add(String cacheName)  
 -->  
<!--  
 name:缓存名称。  
 maxElementsInMemory：缓存最大个数。  
 eternal:对象是否永久有效，一但设置了，timeout将不起作用。  
 timeToIdleSeconds：设置对象在失效前的允许闲置时间（单位：秒）。仅当eternal=false对象不是永久有效时使用，可选属性，默认值是0，也就是可闲置时间无穷大。  
 timeToLiveSeconds：设置对象在失效前允许存活时间（单位：秒）。最大时间介于创建时间和失效时间之间。仅当eternal=false对象不是永久有效时使用，默认是0.，也就是对象存活时间无穷大。  
 overflowToDisk：当内存中对象数量达到maxElementsInMemory时，Ehcache将会对象写到磁盘中。  
 diskSpoolBufferSizeMB：这个参数设置DiskStore（磁盘缓存）的缓存区大小。默认是30MB。每个Cache都应该有自己的一个缓冲区。  
 maxElementsOnDisk：硬盘最大缓存个数。  
 diskPersistent：是否缓存虚拟机重启期数据 Whether the disk store persists between restarts of the Virtual Machine. The default value is false.  
 diskExpiryThreadIntervalSeconds：磁盘失效线程运行时间间隔，默认是120秒。  
 memoryStoreEvictionPolicy：当达到maxElementsInMemory限制时，Ehcache将会根据指定的策略去清理内存。默认策略是LRU（最近最少使用）。你可以设置为FIFO（先进先出）或是LFU（较少使用）。  
 clearOnFlush：内存数量最大时是否清除。  
-->  
<defaultCache  
maxElementsInMemory="10000"  
eternal="false"  
timeToIdleSeconds="120"  
timeToLiveSeconds="120"  
overflowToDisk="true"  
maxElementsOnDisk="10000000"  
diskPersistent="false"  
diskExpiryThreadIntervalSeconds="120"  
memoryStoreEvictionPolicy="LRU"  
/>  
</ehcache>

#### 测试ehcache的二级缓存

@Test  
public void testTwoLevelCache01() throws Exception {  
 SqlSession sqlSession1 = sqlSessionFactory.openSession();  
SqlSession sqlSession2 = sqlSessionFactory.openSession();  
UserMapper userMapper1 = sqlSession1.getMapper(UserMapper.class);  
UserMapper userMapper2 = sqlSession2.getMapper(UserMapper.class);  
//第一次查询 从数据库读取  
System.*out*.println("第一次查询 从数据库读取");  
User user1 = userMapper1.findUserById(1L);  
System.*out*.println(user1);  
//在close()的时候才会把缓存写入到二级缓存  
sqlSession1.close();  
  
//第二次查询 从缓存读取  
System.*out*.println("第二次查询 从缓存读取");  
User user2 = userMapper2.findUserById(1L);  
System.*out*.println(user2);  
sqlSession2.close();  
}

#### 测试结果

第一次查询 从数据库读取  
2016-12-02 15:23:29,802 [main] DEBUG [net.sf.ehcache.Cache] - com.zhiyou100.mapper.UserMapperCache: com.zhiyou100.mapper.UserMapperMemoryStore miss for -1890793718  
2016-12-02 15:23:29,802 [main] DEBUG [net.sf.ehcache.Cache] - com.zhiyou100.mapper.UserMapper cache - Miss  
2016-12-02 15:23:29,802 [main] DEBUG [com.zhiyou100.mapper.UserMapper] - Cache Hit Ratio [com.zhiyou100.mapper.UserMapper]: 0.0  
2016-12-02 15:23:29,808 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Opening JDBC Connection  
2016-12-02 15:23:30,124 [main] DEBUG [org.apache.ibatis.datasource.pooled.PooledDataSource] - Created connection 1996740648.  
2016-12-02 15:23:30,124 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@7703d828]  
2016-12-02 15:23:30,127 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Preparing: SELECT \* FROM tb\_user WHERE id = ?   
2016-12-02 15:23:30,177 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - ==> Parameters: 1(Long)  
2016-12-02 15:23:30,199 [main] DEBUG [com.zhiyou100.mapper.UserMapper.findUserById] - <== Total: 1  
User{id=1, username='张三', sex='男', birthday=Mon Nov 21 00:10:36 CST 2016, address='河南安阳', order=null}  
2016-12-02 15:23:30,208 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@7703d828]  
2016-12-02 15:23:30,209 [main] DEBUG [org.apache.ibatis.transaction.jdbc.JdbcTransaction] - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@7703d828]  
2016-12-02 15:23:30,209 [main] DEBUG [org.apache.ibatis.datasource.pooled.PooledDataSource] - Returned connection 1996740648 to pool.  
第二次查询 从缓存读取  
2016-12-02 15:23:30,210 [main] DEBUG [net.sf.ehcache.Cache] - com.zhiyou100.mapper.UserMapperCache: com.zhiyou100.mapper.UserMapperMemoryStore hit for -1890793718  
2016-12-02 15:23:30,210 [main] DEBUG [com.zhiyou100.mapper.UserMapper] - Cache Hit Ratio [com.zhiyou100.mapper.UserMapper]: 0.5  
User{id=1, username='张三', sex='男', birthday=Mon Nov 21 00:10:36 CST 2016, address='河南安阳', order=null}

使用场景：

**对于访问响应速度要求高，但是实时性不高的查询，可以采用二级缓存技术**。

注意：在使用二级缓存的时候，要设置一下刷新间隔（cache标签中有一个flashInterval属性）来定时刷新二级缓存，这个刷新间隔根据具体需求来设置，比如设置30分钟、60分钟等，单位为毫秒。

### 局限性

**Mybatis二级缓存对细粒度的数据，缓存实现不好。**

场景：对商品信息进行缓存，由于商品信息查询访问量大，但是要求用户每次查询都是最新的商品信息，此时如果使用二级缓存，就无法实现当一个商品发生变化只刷新该商品的缓存信息而不刷新其他商品缓存信息，因为二级缓存是mapper级别的，当一个商品的信息发送更新，所有的商品信息缓存数据都会清空。

解决此类问题，需要在业务层根据需要对数据有针对性的缓存。

比如可以对经常变化的数据操作单独放到另一个namespace的mapper中。

# 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会话。（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

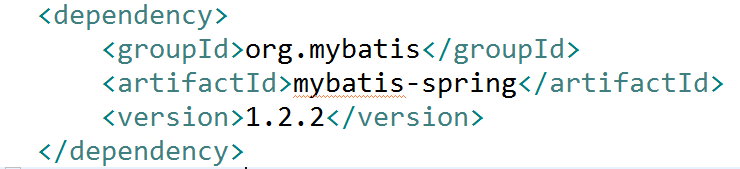
新建java project

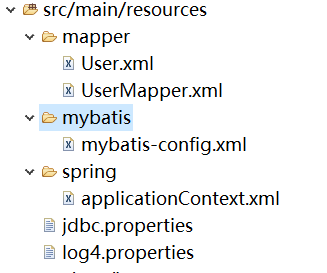
Jar包:

Mybatis的jar包，参看上面项目的依赖

Spring 4.3.8的 jar包

Mybatis和Spring的整合包：mybatis-spring





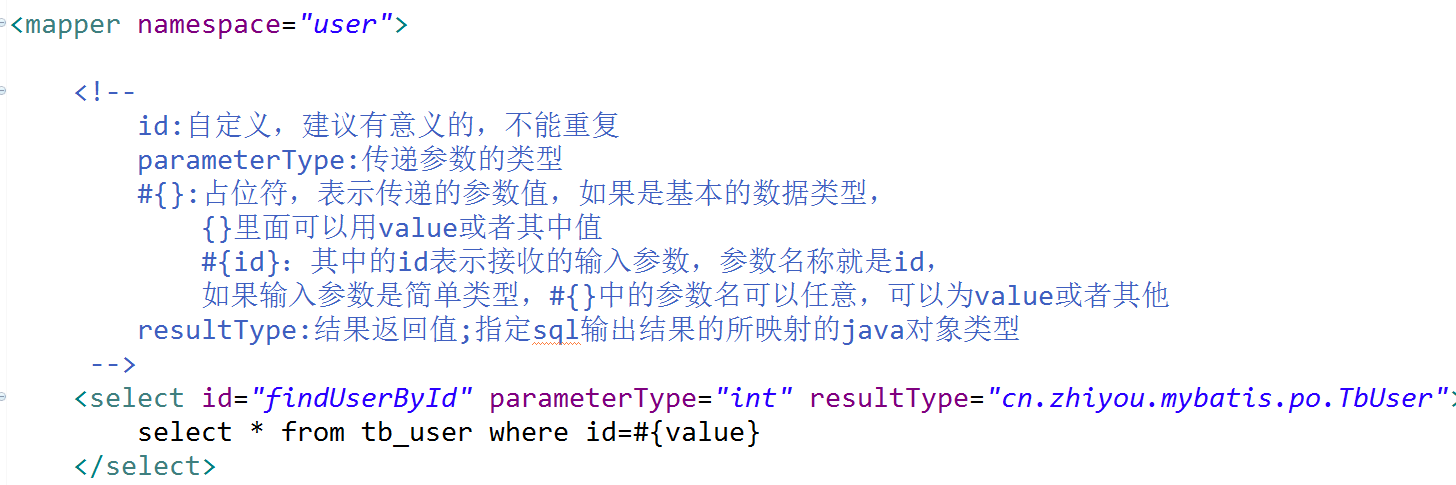
## SqlSessionFactory配置

在spring的核心配置文件中(applicationContext.xml)配置SqlSessionFactory，在mybatis与spring的整合包里

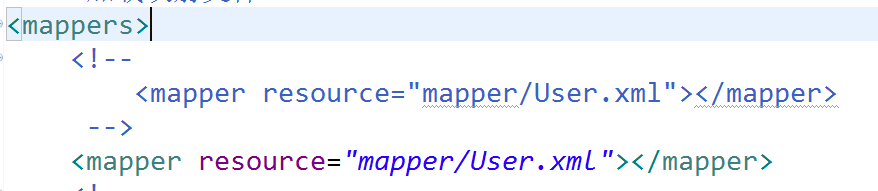


## 原始dao开发（和spring整合后）

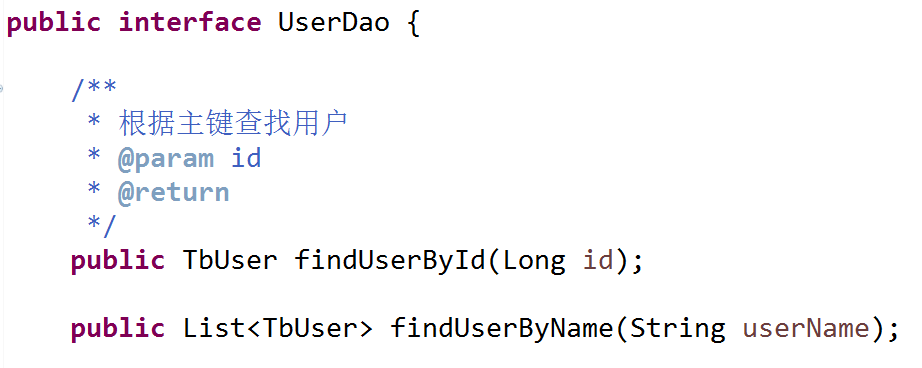
### 创建User.xml



在mybatis-config.xml中加载User.xml映射文件



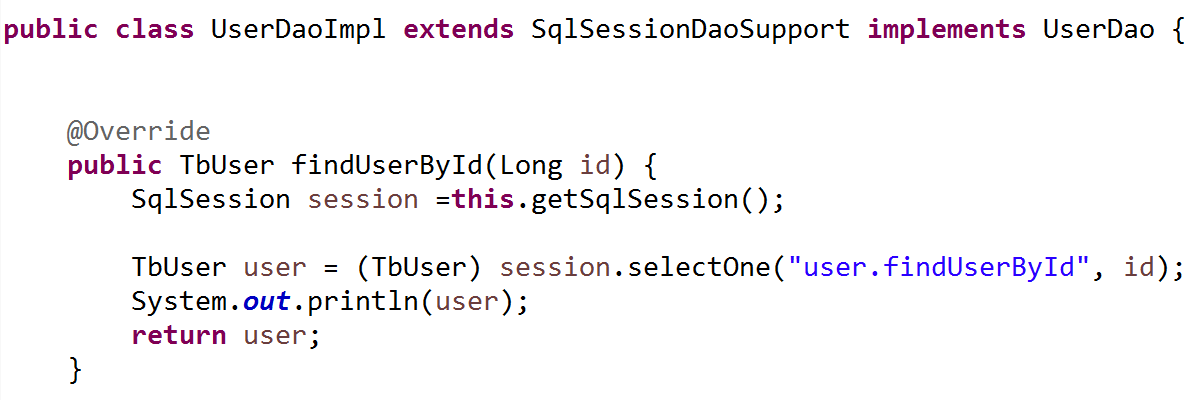
### Dao



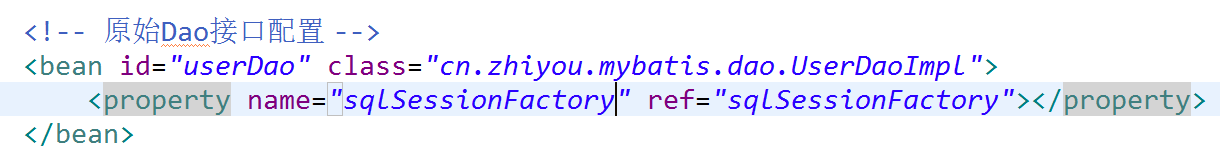
UserDaoImpl实现类中需要注入SqlsessionFactory，通过spring进行注入。

这里spring声明配置方式，配置dao的bean:

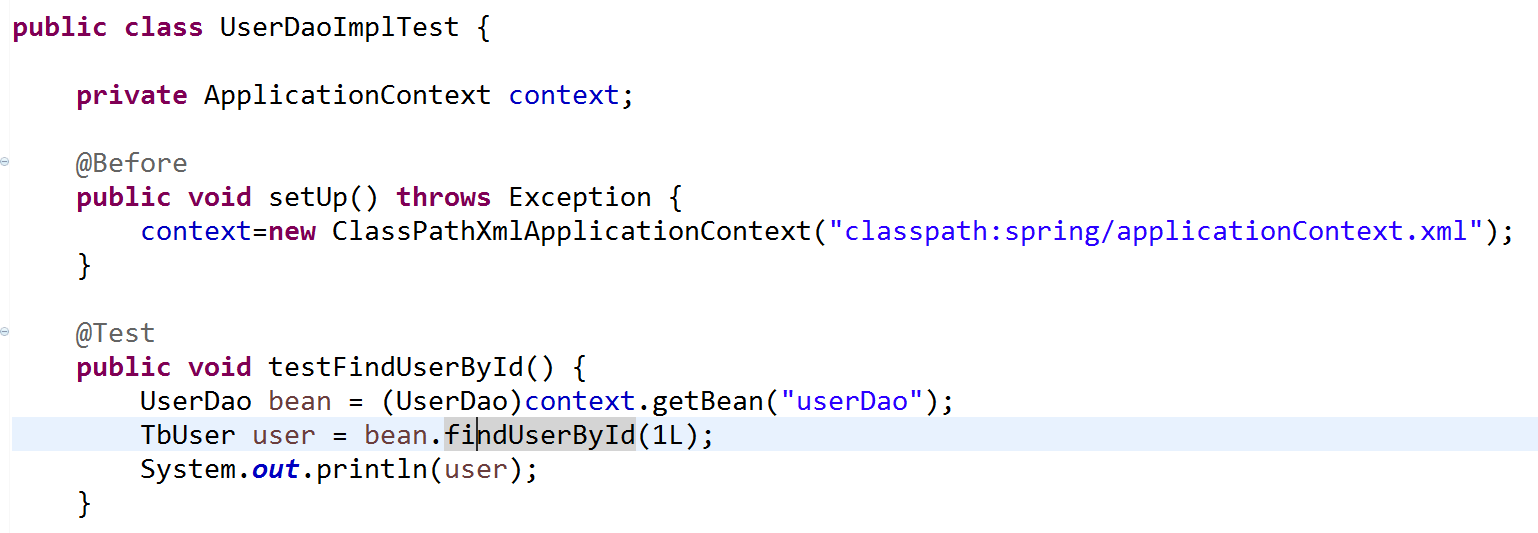
让UserDaoImpl继承SqlSessionDaoSupport



### 配置Dao

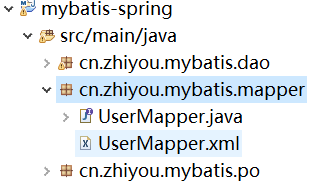


### 测试程序

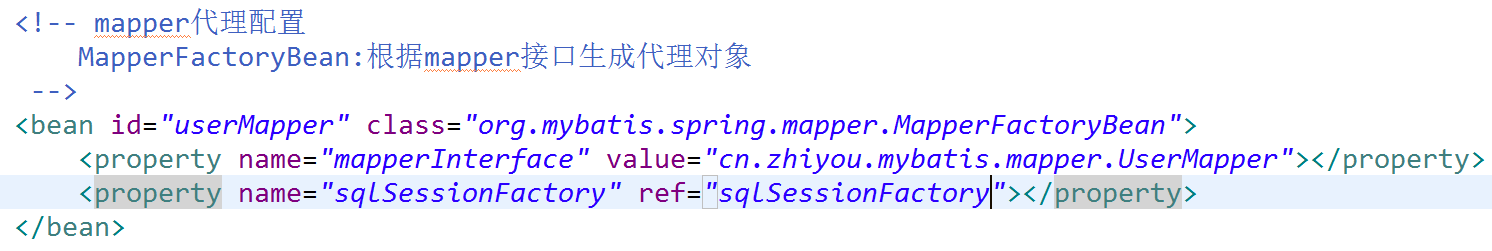


## Mapper代理方式开发

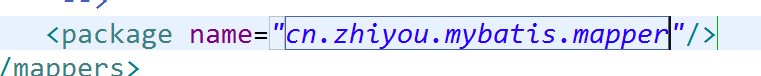
### Mapper.xml和mapper.java



### 通过MapperFactoryBean生成代理对象



并导入映射文件

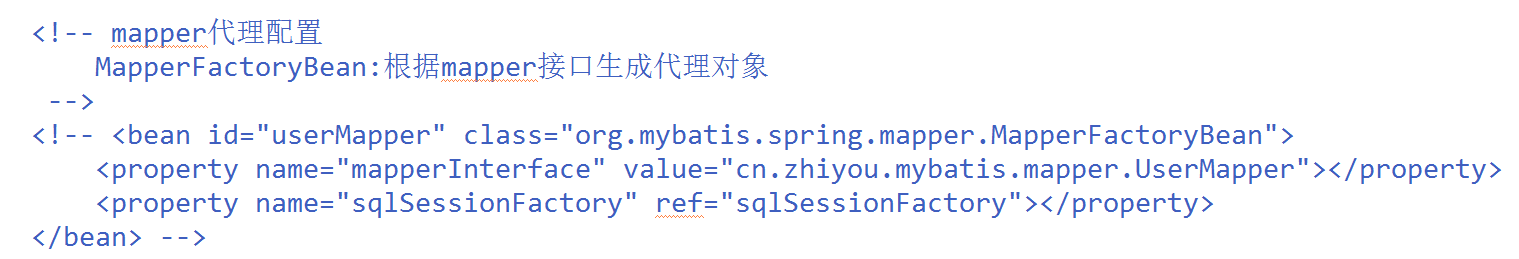


此方法问题：如果模块多，即需要配置多个mapper。

### 通过MapperScannerConfigurer 进行mapper的批量扫描



原先配置的mapper对象可以注释掉



原先mybatis-config.xml配置的包扫描也可以注释掉



### 测试代码

