# MyBatis第二天

内容回顾：

1. mybatis-config.xml
   1. environments:transactionManager dataSource
   2. 引入外部属性资源文件
   3. setttings:开启驼峰匹配
   4. 配置别名
      1. 指定某个类的别名
      2. 扫描包
   5. 引入映射文件
2. 映射文件
   1. <select>:id resultType parameterType
   2. <insert>:id 开启主键回写
   3. <update>:id
   4. <delete>:id
3. reusltMap:自定义结果映射
   1. 解决列名和属性名不一致的问题
   2. 高级映射
4. parameterType:
   1. 基本数据类型
   2. 对象：#{属性名}
   3. hashmap:#{key}
5. 动态代理Mapper接口
   1. 映射文件的namespace和接口的全路径名一直
   2. 映射文件中id和方法名一致
   3. 映射文件中的reusltType和接口中方法的返回类型一直
   4. 映射文件中的parameterType和接口中方法的形参一直（不一定）

# 动态代理Mapper实现类

## 传统CRUD中存在的问题

1. 每个模块都需要编写一个接口及对应的实现类，并且还需要编写一个Mapper文件，麻烦；
2. 在实现类的每个方法中，使用Mybatis的方式很相似，都是增删改查；

我们能否只编写接口及mapper文件，不用写实现类呢？

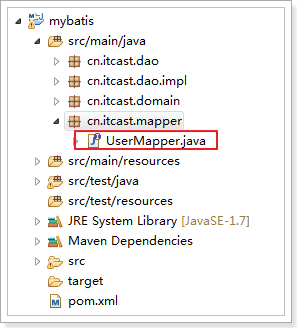
是可以的，mybatis提供了接口的动态代理。

Mapper接口的动态代理实现需要满足以下条件：

1. **映射文件中的namespace必须和接口的全路径名一致；**
2. **映射文件的statementId必须和接口中方法名一致；**
3. 映射文件中的resultType必须和接口中方法的返回类型一致；
4. 映射文件中的parameterType必须和接口中方法的形参类型一致；（不一定，因为parameterType可以省略）

## 使用动态代理改造CRUD

### 创建UserMapper接口



UserMapper:

**public** **interface** UserMapper {

/\*\*

\* 根据id查询用户

\*

\* **@param** id

\* **@return**

\*/

**public** User findUserById(Long id);

/\*\*

\* 查询所有用户

\*

\* **@return**

\*/

**public** List<User> findAllUsers();

/\*\*

\* 保存用户

\*

\* **@param** user

\*/

**public** **void** insertUser(User user);

/\*\*

\* 根据id删除用户

\*

\* **@param** id

\*/

**public** **void** deleteUser(Long id);

/\*\*

\* 更新用户

\*

\* **@param** user

\*/

**public** **void** updateUser(User user);

/\*\*

\* 登录

\* 使用动态代理，当接口中的某个方法有多个参数时，通过@Param注解指定参数名字

\* **@param** userName

\* **@param** password

\* **@return**

\*/

**public** User login(**@Param("userName")**String userName, **@Param("password")**String password);

}

### 改造UserMapper.xml

改造UserMapper.xml：

<?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=*"****cn.itcast.mapper.UserMapper****"*>

<resultMap type=*"User"* id=*"userMap"* autoMapping=*"true"*>

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

</resultMap>

<select id=*"findUserById"* resultMap=*"userMap"*>

select <include refid=*"CommonSql.commonSql"*></include> from tb\_user where id = #{id}

</select>

<select id=*"findAllUsers"* resultType=*"User"*>

select <include refid=*"CommonSql.commonSql"*></include> from tb\_user

</select>

<insert id=*"insertUser"* useGeneratedKeys=*"true"* keyColumn=*"id"* keyProperty=*"id"*>

INSERT INTO tb\_user (

user\_name,

password,

name,

age,

sex,

birthday,

created,

updated

)

VALUES

(

#{userName},

#{password},

#{name},

#{age},

#{sex},

#{birthday},

NOW(),

NOW()

)

</insert>

<update id=*"updateUser"*>

UPDATE tb\_user set

user\_name = #{userName},

password = #{password},

name = #{name},

age = #{age},

sex = #{sex},

birthday = #{birthday},

updated = NOW()

WHERE

id = #{id}

</update>

<delete id=*"deleteUser"*>

delete from tb\_user where

id = #{id}

</delete>

<!--

此处需要接收多个参数，有三种方式：

方式一：#{0} #{1}

方式二：#{param1} #{param2}

方式三：在接口方法的形参中加@Param注解指定名字，这里通过#{名字}来接收参数，如果只有

一个参数，#{名字}可以自定义

-->

<select id=*"login"* resultType=*"User"*>

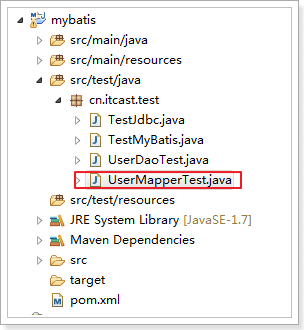
select \* from tb\_user where user\_name = #{userName} and password = #{password}

</select>

</mapper>

**注意：namespace要与接口的全路径名一致；接口中的方法名要与statementId一致。**

### 创建单元测试类



UserMapperTest:

**public** **class** UserMapperTest {

**private** UserMapper userMapper;

@Before

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

InputStream in = Resources.*getResourceAsStream*("mybatis-config.xml");

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

//事务自动提交

SqlSession sqlSession = sqlSessionFactory.openSession(**true**);

//获取UserMapper的代理类对象

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

}

@Test

**public** **void** testFindUserById() {

User user = userMapper.findUserById(1L);

System.***out***.println(user);

}

@Test

**public** **void** testFindAllUsers() {

List<User> users = userMapper.findAllUsers();

**for** (User user : users) {

System.***out***.println(user);

}

}

@Test

**public** **void** testSave() {

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

user.setUserName("zimuge");

user.setPassword("123456");

user.setName("字母哥");

userMapper.insertUser(user);

}

@Test

**public** **void** testDelete() {

userMapper.deleteUser(19L);

}

@Test

**public** **void** testUpdate() {

User user = userMapper.findUserById(1L);

user.setSex(30);

userMapper.updateUser(user);

}

@Test

**public** **void** testLogin() {

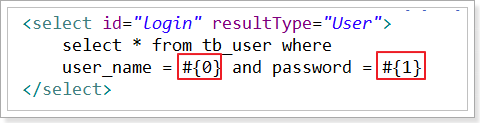
userMapper.login("zhangsan", "12345");

}

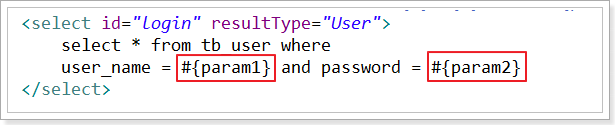
}

### Map接口的方法中有多个参数的情况

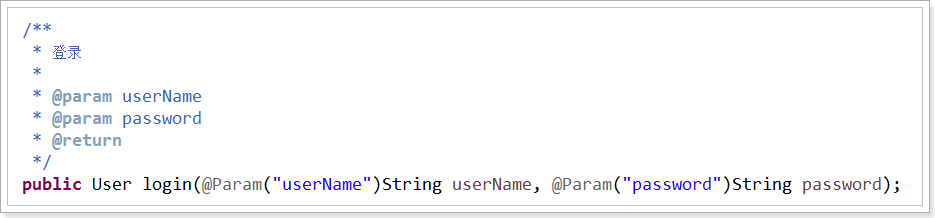
解决方案一：#{0} #{1}

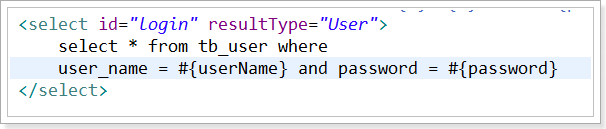


解决方案二：#{param1} #{param2}



解决方案三：在接口方法的形参前加@Param注解指定参数名



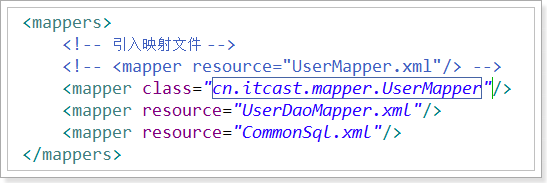


小结：不管map接口中的方法有几个参数（简单参数）建议都加@Param

# 动态代理优化

## 配置接口全路径

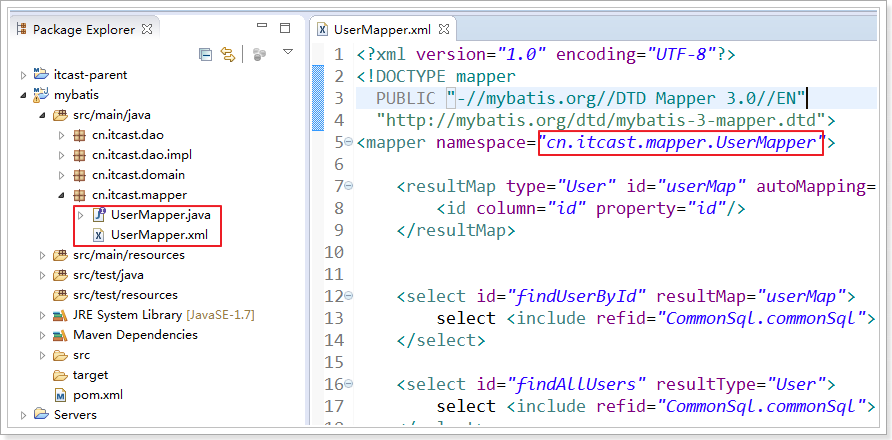
在mybatis-config.xml中配置接口的全路径名：



这样配置，并没有指定接口对应的映射文件的路径；为了能够找到映射文件，需要满足一定的条件：

1. 映射文件和接口在同一个目录；
2. 映射文件的名字必须和接口的名字保持一致；
3. 映射文件中的namespace和接口的全路径名一致；

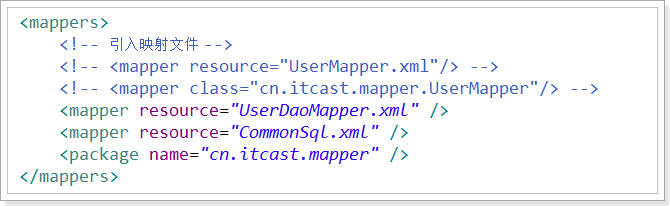
目录结构：



弊端：

1. 每增加一个mapper文件，都要在mybatis-config.xml中引入；
2. Java文件和mapper文件耦合；

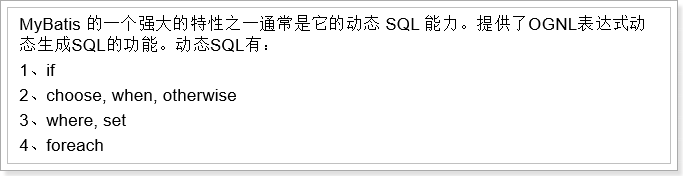
## 配置package



弊端：Java文件和mapper文件耦合。

最优化的写法在spring整合mybatis会讲解。

# 动态sql



## if

作用：判断语句

案例：查询男性用户，如果输入了用户名，就按照用户名模糊查询。

### 定义接口

**public** **interface** UserMapper {

/\*\*

\* 根据id查询用户

\*

\* **@param** id

\* **@return**

\*/

**public** User findUserById(Long id);

/\*\*

\* 查询所有用户

\*

\* **@return**

\*/

**public** List<User> findAllUsers();

/\*\*

\* 保存用户

\*

\* **@param** user

\*/

**public** **void** insertUser(User user);

/\*\*

\* 根据id删除用户

\*

\* **@param** id

\*/

**public** **void** deleteUser(Long id);

/\*\*

\* 更新用户

\*

\* **@param** user

\*/

**public** **void** updateUser(User user);

/\*\*

\* 登录

\*

\* **@param** userName

\* **@param** password

\* **@return**

\*/

**public** User login(@Param("userName")String userName, @Param("password")String password);

/\*\*

\* 根据用户名模糊查询

\* **@param** userName

\* **@return**

\*/

**public** List<User> findUsersLikeUserName(@Param("userName")String userName);

}

### 编写mapper.xml

在UserMapper.xml中添加语句：

<select id=*"findUsersLikeUserName"* resultType=*"User"*>

select <include refid=*"CommonSql.commonSql"*></include> from tb\_user where sex = 1

<if test=*"userName!=null and userName.trim()!=''"*>

and user\_name like '%' #{userName} '%'

</if>

</select>

### 测试

在UserMapperTest中增加测试方法：

@Test

**public** **void** testFindUsersLikeUserName() {

List<User> list = userMapper.findUsersLikeUserName("z");

**for** (User user : list) {

System.***out***.println(user);

}

}

## choose when otherwise

作用：choose条件选择，一旦有一个when成立，后续的when都不会执行，如果所有的when都不成立，那就执行otherwise。

案例：查询男性用户，如果输了用户名，就按照用户名模糊查询；否则，如果输入了年龄则按照年龄查询；否则，查询用户名为张三的用户。

### 定义接口

**public** **interface** UserMapper {

/\*\*

\* 根据id查询用户

\*

\* **@param** id

\* **@return**

\*/

**public** User findUserById(Long id);

/\*\*

\* 查询所有用户

\*

\* **@return**

\*/

**public** List<User> findAllUsers();

/\*\*

\* 保存用户

\*

\* **@param** user

\*/

**public** **void** insertUser(User user);

/\*\*

\* 根据id删除用户

\*

\* **@param** id

\*/

**public** **void** deleteUser(Long id);

/\*\*

\* 更新用户

\*

\* **@param** user

\*/

**public** **void** updateUser(User user);

/\*\*

\* 登录

\*

\* **@param** userName

\* **@param** password

\* **@return**

\*/

**public** User login(@Param("userName") String userName, @Param("password") String password);

/\*\*

\* 根据用户名模糊查询

\*

\* **@param** userName

\* **@return**

\*/

**public** List<User> findUsersLikeUserName(@Param("userName") String userName);

/\*\*

\* 查询男性用户，如果用户名不为空就按照用户名模糊查询；否则，如果年龄不为空则按照年龄查询；否则，查询用户名为张三的用户

\* **@param** userName

\* **@param** age

\* **@return**

\*/

**public** List<User> findUsersLikeUserNameOrAge(@Param("userName") String userName, @Param("age") Integer age);

}

### 编写mapper.xml

<select id=*"findUsersLikeUserNameOrAge"* resultType=*"User"*>

select <include refid=*"CommonSql.commonSql"*></include> from tb\_user where sex = 1

<choose>

<when test=*"userName!=null and userName.trim()!=''"*>

and user\_name like '%' #{userName} '%'

</when>

<when test=*"age!=null"*>

and age = #{age}

</when>

<otherwise>

and user\_name = 'zhangsan'

</otherwise>

</choose>

</select>

### 测试

在UserMapperTest中增加测试方法：

@Test

**public** **void** testFindUsersLikeUserNameOrAge() {

List<User> list = userMapper.findUsersLikeUserNameOrAge("", 20);

**for** (User user : list) {

System.***out***.println(user);

}

}

## where

作用：自动添加where关键字，具备一定的纠错功能，会自动去掉sql语句块前多余的and或or，一般和if、choose使用。

案例：查询所用用户，如果用户名不为空就按照用户名模糊查询；如果年龄不为空则按照年龄查询；如果两个都不为空，就按照用户名和年龄来查询。

### 定义接口

**public** **interface** UserMapper {

/\*\*

\* 根据id查询用户

\*

\* **@param** id

\* **@return**

\*/

**public** User findUserById(Long id);

/\*\*

\* 查询所有用户

\*

\* **@return**

\*/

**public** List<User> findAllUsers();

/\*\*

\* 保存用户

\*

\* **@param** user

\*/

**public** **void** insertUser(User user);

/\*\*

\* 根据id删除用户

\*

\* **@param** id

\*/

**public** **void** deleteUser(Long id);

/\*\*

\* 更新用户

\*

\* **@param** user

\*/

**public** **void** updateUser(User user);

/\*\*

\* 登录

\*

\* **@param** userName

\* **@param** password

\* **@return**

\*/

**public** User login(@Param("userName") String userName, @Param("password") String password);

/\*\*

\* 根据用户名模糊查询

\*

\* **@param** userName

\* **@return**

\*/

**public** List<User> findUsersLikeUserName(@Param("userName") String userName);

/\*\*

\* 查询男性用户，如果用户名不为空就按照用户名模糊查询；否则，如果年龄不为空则按照年龄查询；否则，查询用户名为张三的用户

\* **@param** userName

\* **@param** age

\* **@return**

\*/

**public** List<User> findUsersLikeUserNameOrAge(@Param("userName") String userName, @Param("age") Integer age);

/\*\*

\* 查询所用用户，如果用户名不为空就按照用户名模糊查询；如果年龄不为空则按照年龄查询；如果两个都不为空，就按照用户名和年龄来查询

\* **@param** userName

\* **@param** age

\* **@return**

\*/

**public** List<User> findUsersLikeUserNameAndAge(@Param("userName") String userName, @Param("age") Integer age);

}

### 编写mapper.xml

<!--

自动添加where关键字，具备一定的纠错功能，会自动去掉sql片段前多余的and或or

一般和if、choose使用

-->

<select id=*"findUsersLikeUserNameAndAge"* resultType=*"User"*>

select <include refid=*"CommonSql.commonSql"*></include> from tb\_user

<where>

<if test=*"userName!=null and userName.trim()!=''"*>

user\_name like '%' #{userName} '%'

</if>

<if test=*"age!=null"*>

and age = #{age}

</if>

</where>

</select>

### 测试

在UserMapperTest中增加测试方法：

@Test

**public** **void** testFindUsersLikeUserNameAndAge() {

List<User> list = userMapper.findUsersLikeUserNameAndAge("zhangwei", 20);

**for** (User user : list) {

System.***out***.println(user);

}

}

## set

作用：自动添加set关键字，也具备一定的纠错功能，会自动去掉sql语句块后多余的逗号。

案例：修改用户信息，如果user对象中的某个属性为null，则不修改该属性。

### 定义接口

**public** **interface** UserMapper {

/\*\*

\* 根据id查询用户

\*

\* **@param** id

\* **@return**

\*/

**public** User findUserById(Long id);

/\*\*

\* 查询所有用户

\*

\* **@return**

\*/

**public** List<User> findAllUsers();

/\*\*

\* 保存用户

\*

\* **@param** user

\*/

**public** **void** insertUser(User user);

/\*\*

\* 根据id删除用户

\*

\* **@param** id

\*/

**public** **void** deleteUser(Long id);

/\*\*

\* 更新用户

\*

\* **@param** user

\*/

**public** **void** updateUser(User user);

/\*\*

\* 登录

\*

\* **@param** userName

\* **@param** password

\* **@return**

\*/

**public** User login(@Param("userName") String userName, @Param("password") String password);

/\*\*

\* 根据用户名模糊查询

\*

\* **@param** userName

\* **@return**

\*/

**public** List<User> findUsersLikeUserName(@Param("userName") String userName);

/\*\*

\* 查询男性用户，如果用户名不为空就按照用户名模糊查询；否则，如果年龄不为空则按照年龄查询；否则，查询用户名为张三的用户

\* **@param** userName

\* **@param** age

\* **@return**

\*/

**public** List<User> findUsersLikeUserNameOrAge(@Param("userName") String userName, @Param("age") Integer age);

/\*\*

\* 查询所用用户，如果用户名不为空就按照用户名模糊查询；如果年龄不为空则按照年龄查询；如果两个都不为空，就按照用户名和年龄来查询

\* **@param** userName

\* **@param** age

\* **@return**

\*/

**public** List<User> findUsersLikeUserNameAndAge(@Param("userName") String userName, @Param("age") Integer age);

/\*\*

\* 修改用户信息，如果user对象中的某个属性为null，则不修改该属性

\* **@param** user

\*/

**public** **void** updateUserSelective(User user);

}

### 编写mapper.xml

<update id=*"updateUserSelective"*>

update tb\_user

<set>

<if test=*"userName!=null and userName.trim()!=''"*>

user\_name = #{userName},

</if>

<if test=*"password!=null and password.trim()!=''"*>

password = #{password},

</if>

<if test=*"age!=null"*>

age = #{age},

</if>

<if test=*"sex!=null"*>

sex = #{sex}

</if>

</set>

where id = #{id}

</update>

### 测试

在UserMapperTest中增加测试方法：

@Test

**public** **void** testUpdateUserSelective() {

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

user.setId(17L);

user.setUserName("kevin1");

// user.setPassword("111111");

// user.setAge(28);

// user.setSex(1);

userMapper.updateUserSelective(user);

}

## foreach

作用：遍历集合。

案例：按照多个id查询用户信息。

### 定义接口

**public** **interface** UserMapper {

/\*\*

\* 根据id查询用户

\*

\* **@param** id

\* **@return**

\*/

**public** User findUserById(Long id);

/\*\*

\* 查询所有用户

\*

\* **@return**

\*/

**public** List<User> findAllUsers();

/\*\*

\* 保存用户

\*

\* **@param** user

\*/

**public** **void** insertUser(User user);

/\*\*

\* 根据id删除用户

\*

\* **@param** id

\*/

**public** **void** deleteUser(Long id);

/\*\*

\* 更新用户

\*

\* **@param** user

\*/

**public** **void** updateUser(User user);

/\*\*

\* 登录

\*

\* **@param** userName

\* **@param** password

\* **@return**

\*/

**public** User login(@Param("userName") String userName, @Param("password") String password);

/\*\*

\* 根据用户名模糊查询

\*

\* **@param** userName

\* **@return**

\*/

**public** List<User> findUsersLikeUserName(@Param("userName") String userName);

/\*\*

\* 查询男性用户，如果用户名不为空就按照用户名模糊查询；否则，如果年龄不为空则按照年龄查询；否则，查询用户名为张三的用户

\* **@param** userName

\* **@param** age

\* **@return**

\*/

**public** List<User> findUsersLikeUserNameOrAge(@Param("userName") String userName, @Param("age") Integer age);

/\*\*

\* 查询所用用户，如果用户名不为空就按照用户名模糊查询；如果年龄不为空则按照年龄查询；如果两个都不为空，就按照用户名和年龄来查询

\* **@param** userName

\* **@param** age

\* **@return**

\*/

**public** List<User> findUsersLikeUserNameAndAge(@Param("userName") String userName, @Param("age") Integer age);

/\*\*

\* 修改用户信息，如果user对象中的某个属性为null，则不修改该属性

\* **@param** user

\*/

**public** **void** updateUserSelective(User user);

/\*\*

\* 按照多个id查询用户信息

\* **@param** ids

\*/

**public** List<User> findUsersByIds(@Param("ids")Long[] ids);

}

### 编写mapper.xml

<!--

foreach遍历集合、数组

collection：要遍历的集合或数组，名字要与@Param指定的名字一致；

item：遍历集合、数组中的一个元素

open：以什么开始

close：以什么结束

separator：分隔符

-->

<select id=*"findUsersByIds"* resultType=*"User"*>

select <include refid=*"CommonSql.commonSql"*></include> from tb\_user where id in

<foreach collection=*"ids"* item=*"id"* open=*"("* close=*")"* separator=*","*>

#{id}

</foreach>

</select>

### 测试

@Test

**public** **void** testQueryUsersByIds() {

List<User> list = userMapper.findUsersByIds(**new** Long[]{1L,2L,3L});

**for** (User user : list) {

System.***out***.println(user);

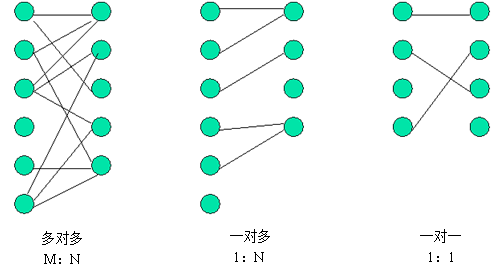
}

}

# 多表查询

## 表之间关系划分

数据库中多表之间存在着三种关系，也就是系统设计中的三种实体关系。如图所示。



Mybatis作为一个持久层框架，对sql的多表查询提供了支持。下面我们学习mybatis中的一对一、一对多、多对多的查询。

## 一对一查询

### 示例分析

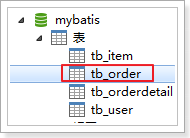
本小节，我们研究订单和用户的关系。

站在订单的角度来看，一个订单只属于一个用户，所以，订单和用户的关系是一对一。

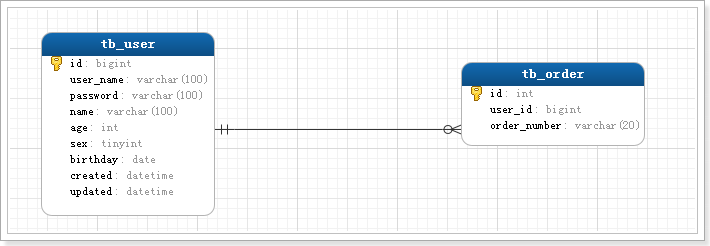
需求：查询订单，同时还要查询出订单所属的用户信息。

### 设计表结构

创建订单表：

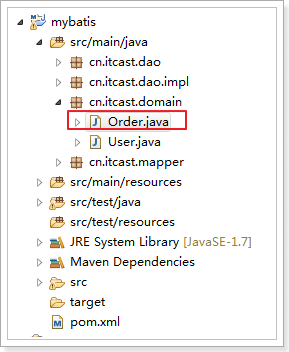


订单表和用户表之间的关系：



### 设计实体类

创建Order实体类：



Order:

**public** **class** Order **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** Integer id;

**private** String orderNumber;

**private** User user;//表示该订单所属的用户

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** String getOrderNumber() {

**return** orderNumber;

}

**public** **void** setOrderNumber(String orderNumber) {

**this**.orderNumber = orderNumber;

}

**public** User getUser() {

**return** user;

}

**public** **void** setUser(User user) {

**this**.user = user;

}

@Override

**public** String toString() {

**return** "Order [id=" + id + ", orderNumber=" + orderNumber + ", user=" + user + "]";

}

}

### 编写接口

在UserMapper中添加方法：

/\*\*

\* 根据订单号查询订单信息以及该订单所属的用户

\* **@param** number

\* **@return**

\*/

**public** Order findOrderUserByNumber(@Param("number")String number);

### 编写mapper文件

在UserMapper.xml中添加语句：

<!--

查询订单及对应用户的信息，mybatis不能直接把用户的信息封装到user对象中，需要自定义resultMap

-->

<resultMap type=*"Order"* id=*"orderUserMap"* autoMapping=*"true"*>

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

<!--

association：一对一映射

property属性：Order类中关联的另一方的属性名

javaType属性：Order类中关联的另一方的属性的类型

autoMapping属性：开启自动映射

-->

<association property=*"user"* javaType=*"User"* autoMapping=*"true"*>

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

</association>

</resultMap>

<select id=*"findOrderUserByNumber"* resultMap=*"orderUserMap"*>

select \* from tb\_order a left join tb\_user b on a.user\_id = b.id where a.order\_number = #{number}

</select>

### 测试

在UserMapperTest中添加测试方法：

@Test

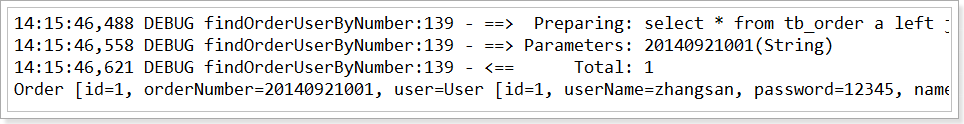
**public** **void** testFindOrderUserByNumber() {

Order order = userMapper.findOrderUserByNumber("20140921001");

System.***out***.println(order);

}

测试结果如下：



## 一对多查询

### 示例分析

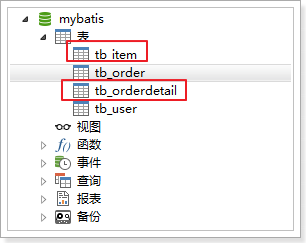
本小节，我们研究订单和订单详情的关系。

站在订单的角度来看，订单和订单详情的是一对多的关系。

需求：查询订单，同时还要查询出该订单下的订单详情信息。

### 设计表结构

创建商品表和订单详情表：



### 设计实体类

订单详情实体类OrderDetail：

**public** **class** OrderDetail **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** Integer id;

**private** Double totalPrice;

**private** Integer status;

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** Double getTotalPrice() {

**return** totalPrice;

}

**public** **void** setTotalPrice(Double totalPrice) {

**this**.totalPrice = totalPrice;

}

**public** Integer getStatus() {

**return** status;

}

**public** **void** setStatus(Integer status) {

**this**.status = status;

}

@Override

**public** String toString() {

**return** "OrderDetail [id=" + id + ", totalPrice=" + totalPrice + ", status=" + status + "]";

}

}

改造Order实体类：

**public** **class** Order **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** Integer id;

**private** String orderNumber;

**private** User user;//表示该订单所属的用户

**private** List<OrderDetail> orderDetails;//一个订单有多个订单详情

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** String getOrderNumber() {

**return** orderNumber;

}

**public** **void** setOrderNumber(String orderNumber) {

**this**.orderNumber = orderNumber;

}

**public** User getUser() {

**return** user;

}

**public** **void** setUser(User user) {

**this**.user = user;

}

**public** List<OrderDetail> getOrderDetails() {

**return** orderDetails;

}

**public** **void** setOrderDetails(List<OrderDetail> orderDetails) {

**this**.orderDetails = orderDetails;

}

@Override

**public** String toString() {

**return** "Order [id=" + id + ", orderNumber=" + orderNumber + ", user=" + user + ", orderDetails=" + orderDetails

+ "]";

}

}

### 编写接口

/\*\*

\* 根据订单号查询订单信息以及该订单所属的用户及该订单下的详细信息

\* **@param** number

\* **@return**

\*/

**public** Order findOrderUserDetailByNumber(@Param("number")String number);

### 编写mapper文件

在UserMapper.xml中添加语句：

<!--

查询订单及对应用户的信息，mybatis不能直接把用户的信息封装到user对象中，需要自定义resultMap

-->

<resultMap type=*"Order"* id=*"orderUserDetailMap"* autoMapping=*"true"*>

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

<!--

association：一对一映射

property属性：Order类中关联的另一方的属性名

javaType属性：Order类中关联的另一方的属性的类型

autoMapping属性：开启自动映射

-->

<association property=*"user"* javaType=*"User"* autoMapping=*"true"*>

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

</association>

<!--

collection：一对多映射

property属性：Order类中集合属性的名字

javaType属性：集合的类型

ofType属性：集合中元素的类型

-->

<collection property=*"orderDetails"* javaType=*"list"* ofType=*"OrderDetail"* autoMapping=*"true"*>

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

</collection>

</resultMap>

<!-- 需要给tb\_detail表中的id加别名，不然id这个列名会有冲突 -->

<select id=*"findOrderUserDetailByNumber"* resultMap=*"orderUserDetailMap"*>

select \*,**c.id as detail\_id** from tb\_order a left join tb\_user b on a.user\_id = b.id left join tb\_orderdetail c on a.id = c.order\_id where a.order\_number = #{number}

</select>

### 测试

在UserMapperTest中增加测试方法：

@Test

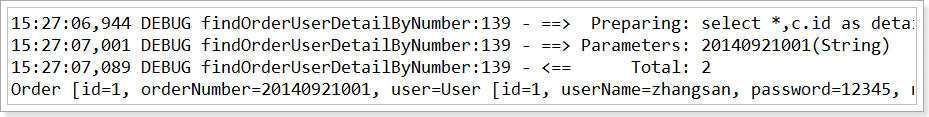
**public** **void** testFindOrderUserDetailByNumber() {

Order order = userMapper.findOrderUserDetailByNumber("20140921001");

System.***out***.println(order);

}

测试结果如下：



## 多对多查询

### 示例分析

本小节，我们研究订单和商品的关系，订单和商品的是多对多的关系：一个订单中有多个商品，一个商品也属于多个订单。

需求：查询订单，同时还要查询出订单下的商品信息。

### 设计实体类

创建商品实体类Item:

**public** **class** Item **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** Integer id;

**private** String itemName;

**private** Double itemPrice;

**private** String itemDetail;

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** String getItemName() {

**return** itemName;

}

**public** **void** setItemName(String itemName) {

**this**.itemName = itemName;

}

**public** Double getItemPrice() {

**return** itemPrice;

}

**public** **void** setItemPrice(Double itemPrice) {

**this**.itemPrice = itemPrice;

}

**public** String getItemDetail() {

**return** itemDetail;

}

**public** **void** setItemDetail(String itemDetail) {

**this**.itemDetail = itemDetail;

}

@Override

**public** String toString() {

**return** "Item [id=" + id + ", itemName=" + itemName + ", itemPrice=" + itemPrice + ", itemDetail=" + itemDetail

+ "]";

}

}

改造OrderDetail：

**public** **class** OrderDetail **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** Integer id;

**private** Double totalPrice;

**private** Integer status;

**private Item item;//该订单详情所属的商品**

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** Double getTotalPrice() {

**return** totalPrice;

}

**public** **void** setTotalPrice(Double totalPrice) {

**this**.totalPrice = totalPrice;

}

**public** Integer getStatus() {

**return** status;

}

**public** **void** setStatus(Integer status) {

**this**.status = status;

}

**public** Item getItem() {

**return** item;

}

**public** **void** setItem(Item item) {

**this**.item = item;

}

@Override

**public** String toString() {

**return** "OrderDetail [id=" + id + ", totalPrice=" + totalPrice + ", status=" + status + ", item=" + item + "]";

}

}

### 编写接口

/\*\*

\* 根据订单号查询订单信息以及该订单所属的用户及该订单下的详细信息以及商品信息

\* **@param** number

\* **@return**

\*/

**public** Order findOrderUserDetailItemByNumber(@Param("number")String number);

### 编写mapper文件

<resultMap type=*"Order"* id=*"orderUserDetailItemMap"* autoMapping=*"true"*>

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

<!--

association：一对一映射

property属性：Order类中关联的另一方的属性名

javaType属性：Order类中关联的另一方的属性的类型

autoMapping属性：开启自动映射

-->

<association property=*"user"* javaType=*"User"* autoMapping=*"true"*>

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

</association>

<!--

collection：一对多映射

property属性：Order类中集合属性的名字

javaType属性：集合的类型

ofType属性：集合中元素的类型

-->

<collection property=*"orderDetails"* javaType=*"list"* ofType=*"OrderDetail"* autoMapping=*"true"*>

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

<association property=*"item"* javaType=*"Item"* autoMapping=*"true"*>

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

</association>

</collection>

</resultMap>

<select id=*"findOrderUserDetailItemByNumber"* resultMap=*"orderUserDetailItemMap"*>

select \*,c.id as detail\_id from tb\_order a left join tb\_user b on a.user\_id = b.id left join tb\_orderdetail c on a.id = c.order\_id left join tb\_item d on c.item\_id = d.id where a.order\_number = #{number}

</select>

### 测试

在UserMapperTest中增加单元测试方法：

@Test

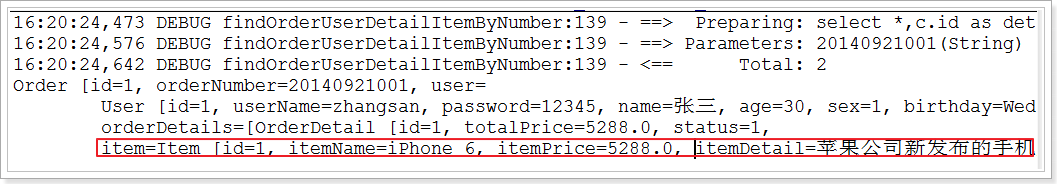
**public** **void** testFindOrderUserDetailItemByNumber() {

Order order = userMapper.findOrderUserDetailItemByNumber("20140921001");

System.***out***.println(order);

}

测试结果如下：



## resultMap的继承

有的时候，在一个Mapper文件中会有多个resultMap有着相同的配置，此时我们可以采用继承的形式减少代码的冗余。

<!--

查询订单及对应用户的信息，mybatis不能直接把用户的信息封装到user对象中，需要自定义resultMap

-->

<resultMap type=*"Order"* id=*"orderUserMap"* autoMapping=*"true"*>

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

<!-- association：一对一映射

property属性：Order类中关联的另一方的属性名

javaType属性：Order类中关联的另一方的属性的类型

autoMapping属性：开启自动映射 -->

<association property=*"user"* javaType=*"User"* autoMapping=*"true"*>

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

</association>

</resultMap>

<resultMap type=*"Order"* id=*"orderUserDetailMap"* autoMapping=*"true"* **extends=*"orderUserMap"***>

<collection property=*"orderDetails"* javaType=*"list"* ofType=*"OrderDetail"* autoMapping=*"true"*>

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

</collection>

</resultMap>

<resultMap type=*"Order"* id=*"orderUserDetailItemMap"* autoMapping=*"true"* **extends=*"orderUserMap"***>

<collection property=*"orderDetails"* javaType=*"list"* ofType=*"OrderDetail"* autoMapping=*"true"*>

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

<association property=*"item"* javaType=*"Item"* autoMapping=*"true"*>

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

</association>

</collection>

</resultMap>

# 延迟加载

## 什么是延迟加载

立即加载：

不管用不用，马上查询。

延迟加载：

等到用的时候才真正发起查询。

采用之前的配置方式肯定是做不到延迟加载的，因为我们是通过一个sql语句查询所有的信息。为了测试延迟加载的效果，我们必须把Order查询和User或OrderDetai查询分开。只有当我们需要用到Order中的User或OrderDetail时，才会去查询User或OrderDetail。

## 改造一对一查询

### 编写接口

/\*\*

\* 根据订单号查询订单信息，延迟加载订单所属的用户

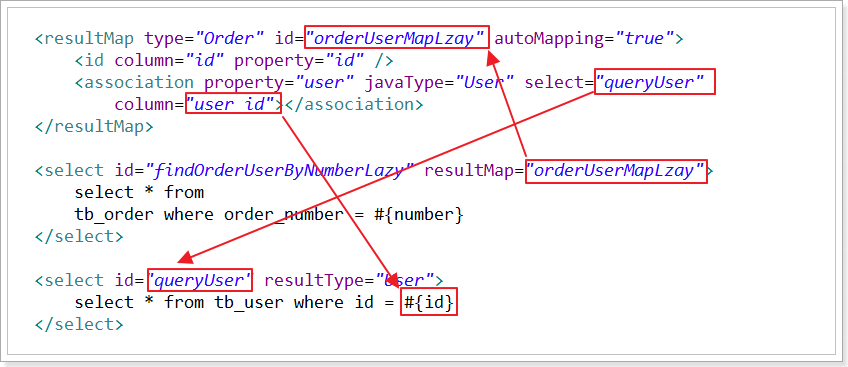
\* **@param** number

\* **@return**

\*/

**public** Order findOrderUserByNumberLazy(@Param("number")String number);

### 编写mapper文件



### 测试

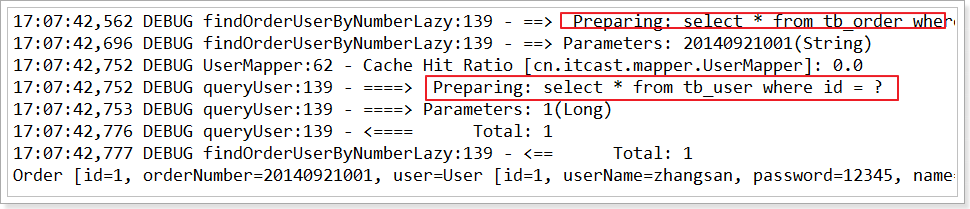
在UserMapperTest中增加测试方法：

@Test

**public** **void** testFindOrderUserByNumberLazy() {

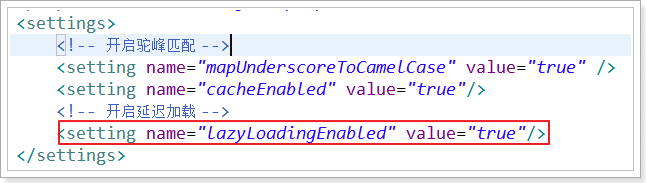
Order order = userMapper.findOrderUserByNumberLazy("20140921001");

}

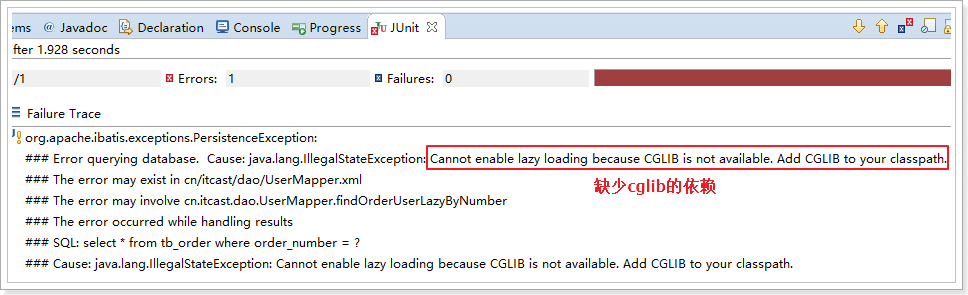


## 开启延迟加载

在mybatis-config.xml中开启延迟加载：



测试报错：



引入cglib的依赖：

<dependency>

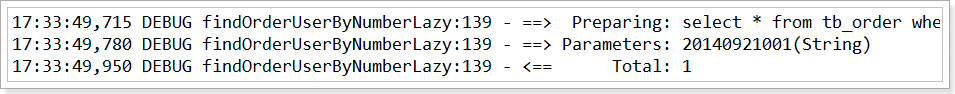
<groupId>cglib</groupId>

<artifactId>cglib</artifactId>

<version>3.1</version>

</dependency>

测试结果如下：只查询了tb\_order表：



修改测试方法：

@Test

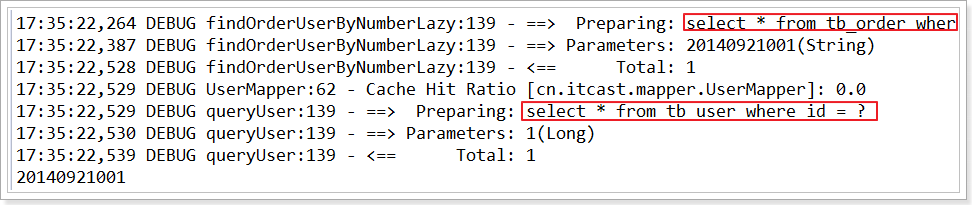
**public** **void** testFindOrderUserByNumberLazy() {

Order order = userMapper.findOrderUserByNumberLazy("20140921001");

System.***out***.println(order.getOrderNumber());

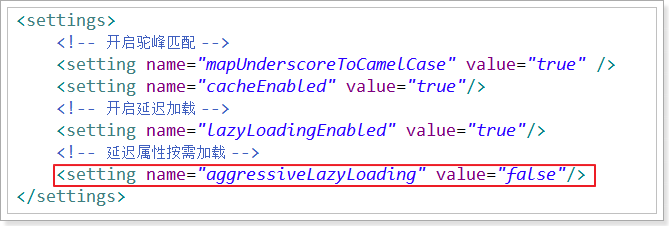
}

测试结果如下:

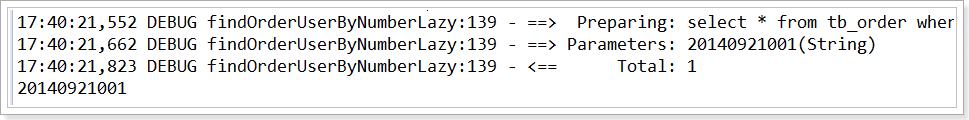


我只打印了订单编号，但是依然会发出查询user表的sql语句，这不满足我们的需求，怎么办？

在mybatis-config.xml中把aggressiveLazyLoading的值设置为false，表示按需加载，什么时候用，什么时候查询



测试结果如下：



修改测试代码：

@Test

**public** **void** testFindOrderUserByNumberLazy() {

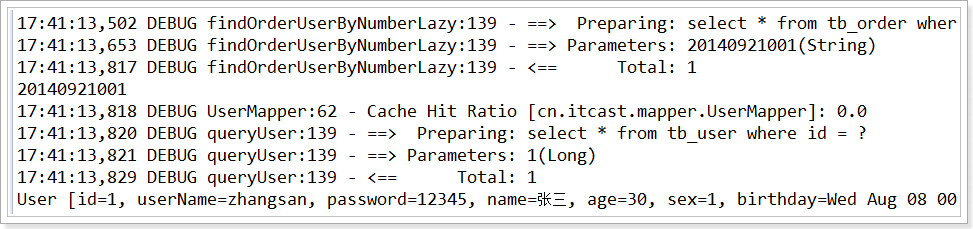
Order order = userMapper.findOrderUserByNumberLazy("20140921001");

System.***out***.println(order.getOrderNumber());

System.***out***.println(order.getUser());

}

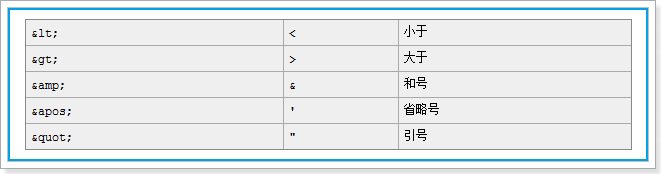
测试结果如下：

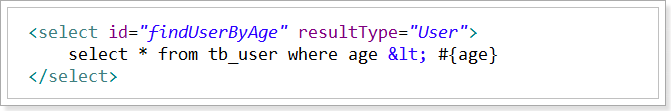


# 如果sql语句中出现<的解决方案

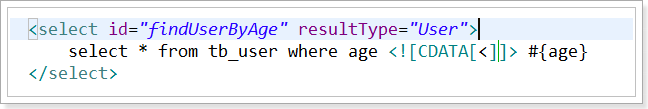
如果在mapper文件中出现<,不符合xml的规范，会报错，可以采用以下两种方式解决：

1、使用xml中的字符实体





1. 使用<![CDATA[<]]>



# 面试题：#和$符号的区别

## $符号的用法

场景：用户数据特别多，需要把用户数据分表存储，user1表和user2表；查询表中的信息，有时需要从user1表中查，有时需要从user2表中查，现在想用一个方法完成。

在UserMapper接口中增加一个根据表名查询用户的方法：

/\*\*

\* 根据表名查询用户信息

\* **@param** tableName

\* **@return**

\*/

**public** List<User> findUserByTableName(@Param("tableName") String tableName);

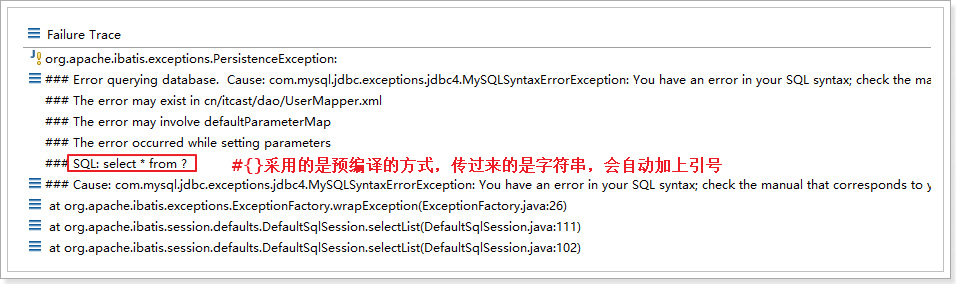
在UserMapper.xml中增加一条sql语句：

<select id=*"findUserByTableName"* resultType=*"User"*>

select \* from #{tableName}

</select>

测试，发现报错：



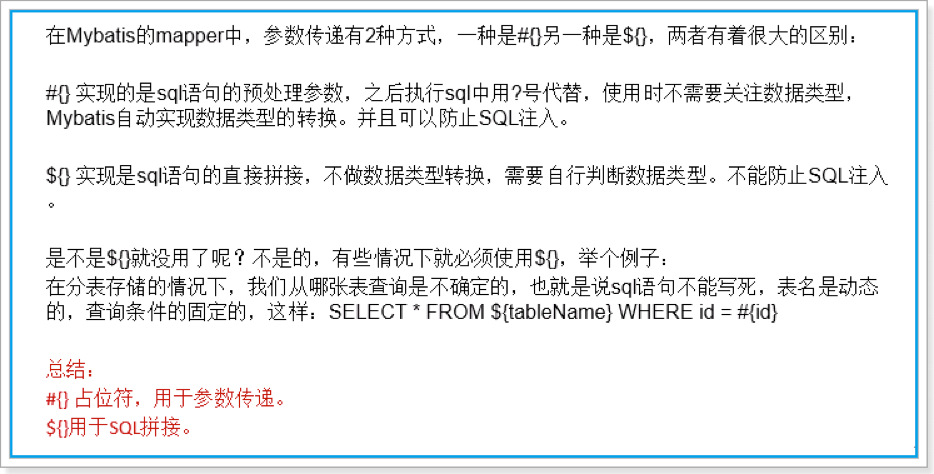
需要改成${}来接收参数，${}是非预编译的方式来接收参数

<select id=*"findUserByTableName"* resultType=*"User"*>

select \* from **${tableName}**

</select>

## 区别



#和$的区别：

1. #是预编译的方式，$是直接拼接；
2. #不需要关注数据类型，mybatis实现自动数据类型转换；$不做数据类型转换，需要自行判断数据类型；
3. #可以防止sql注入；$不能防止sql注入；
4. 如果只有一个参数，默认情况下，#{}中可以写任意的名字；${}中只能用value来接收。