Mybatis

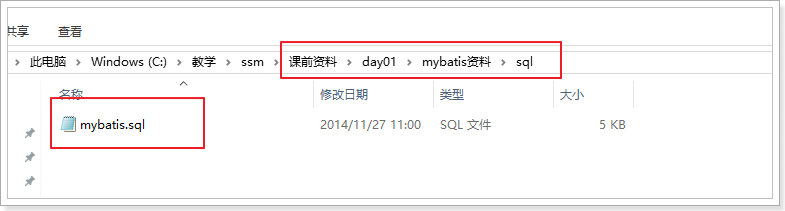
目录：

1. jdbc回顾
2. mybatis介绍
3. 快速入门
4. 使用mybatis进行基本的crud的入门
5. Mapper的动态代理
6. 全局配置文件的配置
7. 映射文件的配置
8. 动态sql的编写
9. 缓存（了解）
10. 高级查询
11. 延迟加载

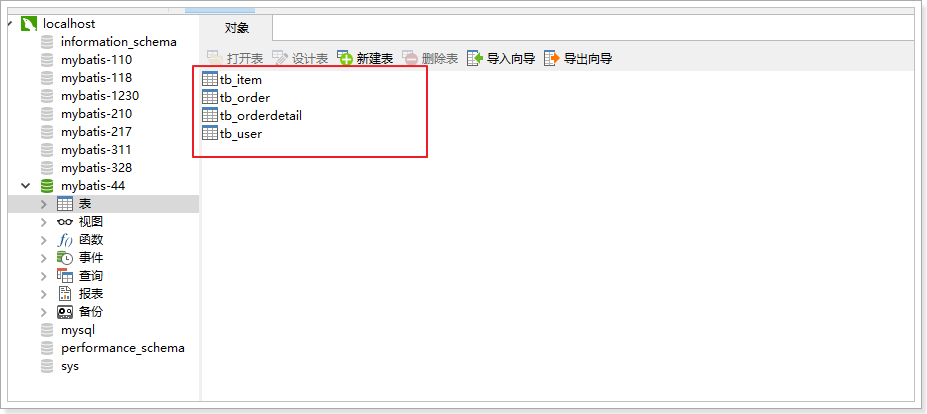
# JDBC回顾

## 创建mybatis库，执行sql

SQL文件：

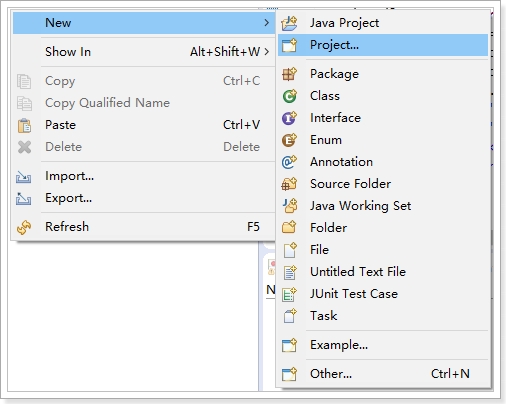


将sql文件直接拉进去或者右键运行sql文件都可。

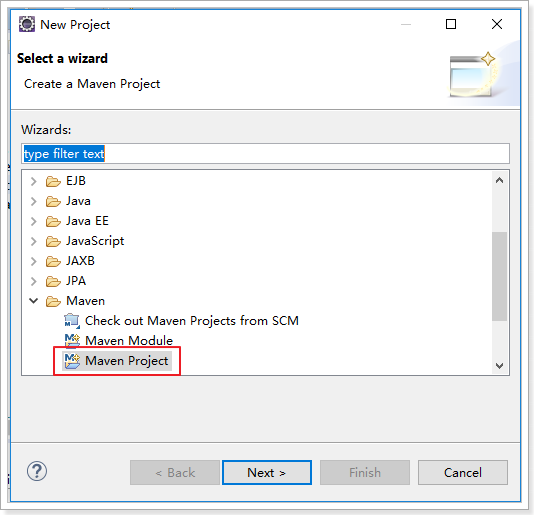


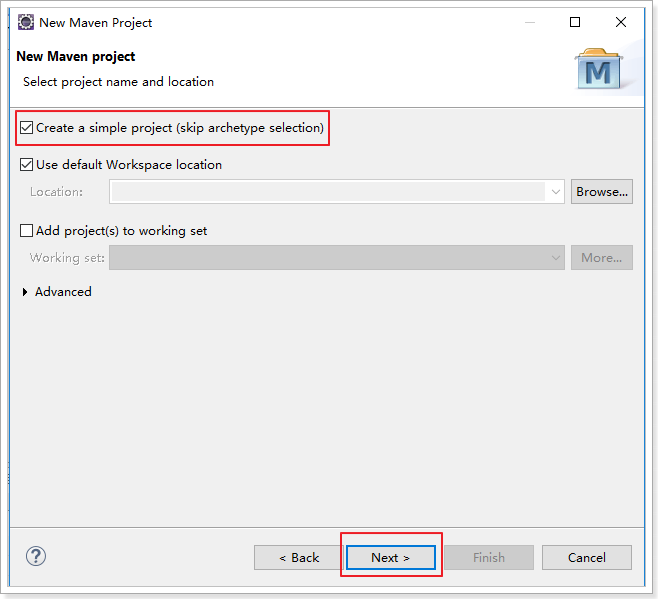
## 创建maven工程

右键—>new-->Project

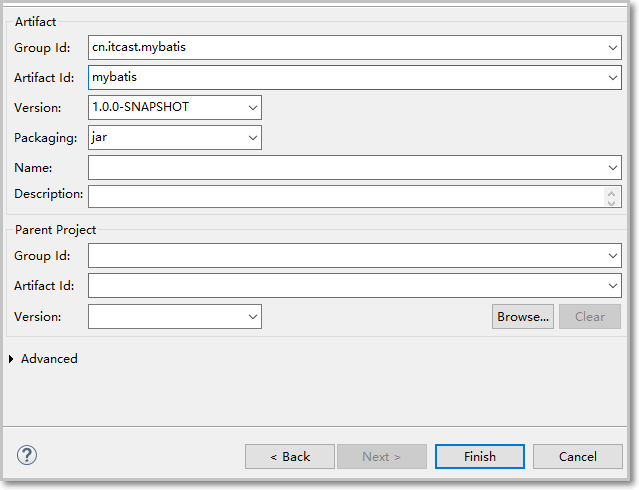


选择：Maven—>Maven Project





坐标：

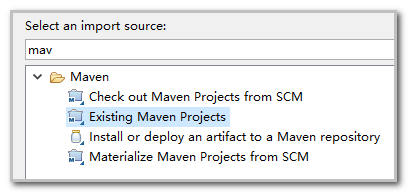


## 引入mysql依赖包

在企业开发中，一个项目可能会有很多人很多个小组一起开发，而为了统一关联各个jar包的版本，因此可以使用一个父工程来统一管理。

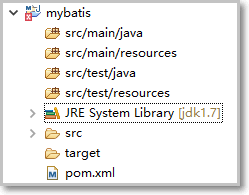
### 步骤一：导入itcast-parent

将资料中的itcast-parent复制到当前工作空间中，然后导入到eclipse中



### 步骤二：在mybatis中引入父工程坐标

创建一个mybatis的maven工程，导入itcast\_parent坐标后报错：更新一下即可



<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.itcast.mybatis</groupId>

<artifactId>mybatis-day01</artifactId>

<version>1.0.0-SNAPSHOT</version>

</project>

### 步骤三：引入mysql依赖包

注意：在引入mysql依赖的时候不需要版本号，因为版本都统一交给父工程进行管理了。

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.itcast.mybatis.day01</groupId>

<artifactId>mybatis-day01</artifactId>

<version>1.0.0-SNAPSHOT</version>

<dependencies>

<!-- MySql -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

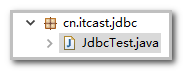
</dependencies>

</project>

## 需求

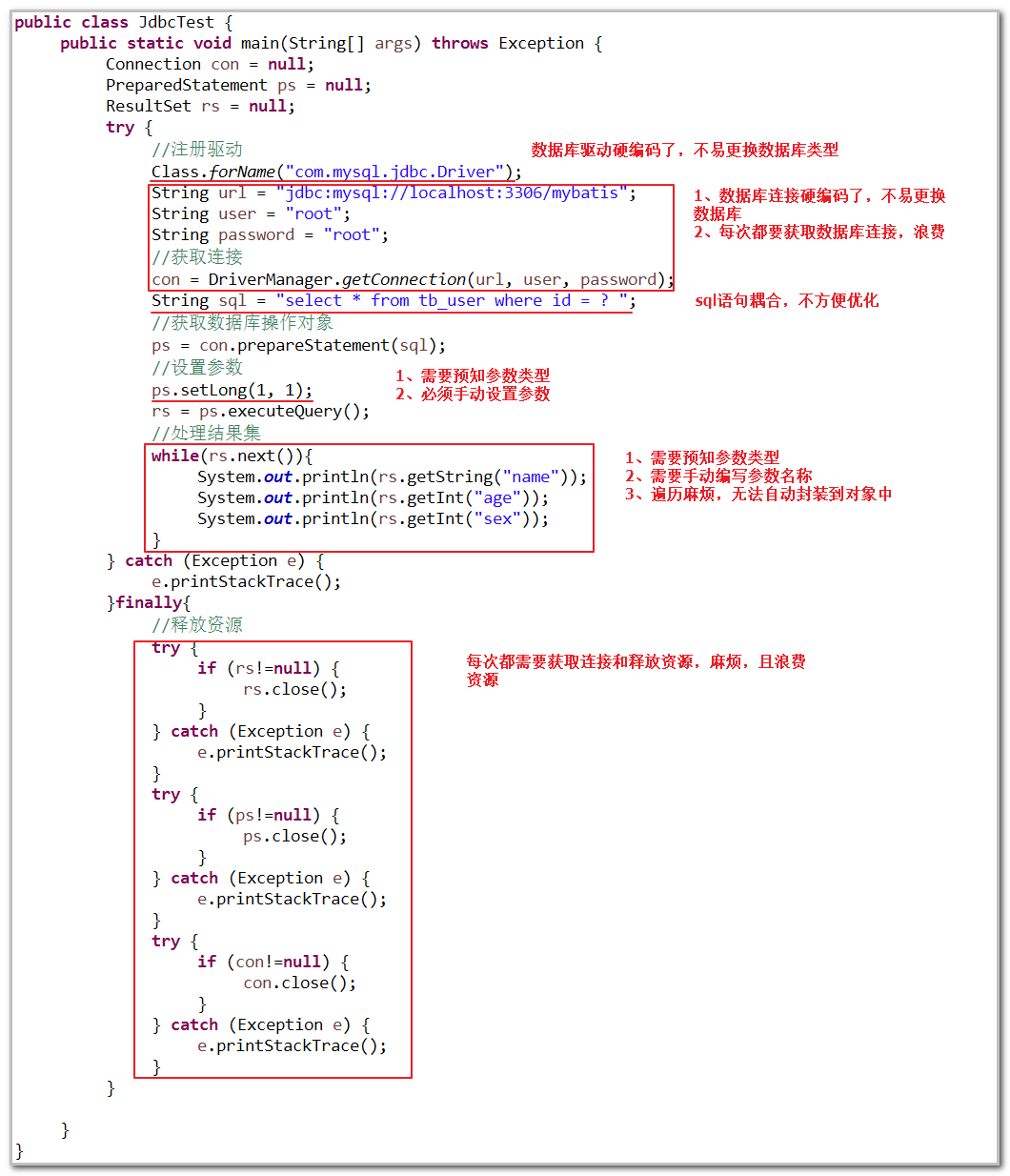
根据id查询用户信息

### 编写测试方法



|  |
| --- |
| **public** **class** JdbcTest {  **public** **static** **void** main(String[] args) **throws** Exception {  Connection con = **null**;  PreparedStatement ps = **null**;  ResultSet rs = **null**;  **try** {  //注册驱动  Class.*forName*("com.mysql.jdbc.Driver");  String url = "jdbc:mysql://localhost:3306/mybatis";  String user = "root";  String password = "root";  //获取连接  con = DriverManager.*getConnection*(url, user, password);  String sql = "select \* from tb\_user where id = ? ";  //获取数据库操作对象  ps = con.prepareStatement(sql);  //设置参数  ps.setLong(1, 1);  rs = ps.executeQuery();  //处理结果集  **while**(rs.next()){  System.***out***.println(rs.getString("name"));  System.***out***.println(rs.getInt("age"));  System.***out***.println(rs.getInt("sex"));  }  } **catch** (Exception e) {  e.printStackTrace();  }**finally**{  //释放资源  **try** {  **if** (rs!=**null**) {  rs.close();  }  } **catch** (Exception e) {  e.printStackTrace();  }  **try** {  **if** (ps!=**null**) {  ps.close();  }  } **catch** (Exception e) {  e.printStackTrace();  }  **try** {  **if** (con!=**null**) {  con.close();  }  } **catch** (Exception e) {  e.printStackTrace();  }  }    }  } |

### 缺点及解决方案



# MyBatis介绍

1. 框架

框架是可以重复使用的一些或一整套代码，通常与具体业务无关，也可以认为是软件的半成品。

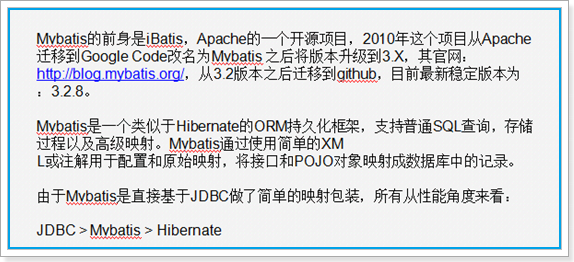
框架的好处：规范开发流程。

1. MVC

全名是Model View Controller，是模型(model)－视图(view)－控制器(controller)的缩写，是一种软件设计模式，或软件设计思想。

1. 视图-view：负责数据的显示。
2. 模型-model：负责数据的处理。
3. 控制器-controller：负责调度不同的逻辑代码（model）并显示到视图（view）。

## 简介



## 官方文档



见mybatis资料中的中文官方文档

## 特点

Mybatis：

1. 支持自定义SQL、存储过程、及高级映射
2. 实现自动对SQL的参数设置
3. 实现自动对结果集进行解析和封装
4. 通过XML或者注解进行配置和映射，大大减少代码量
5. 数据源的连接信息通过配置文件进行配置

可以发现，MyBatis是对JDBC进行了简单的封装，帮助用户进行SQL参数的自动设置，以及结果集与Java对象的自动映射。与Hibernate相比，配置更加简单、灵活、执行效率高。但是正因为此，所以是半自动化的ORM框架，需要手写SQL，这是优点也是缺点。

因此，对性能要求较高的电商类项目，一般会使用MyBatis，而对与业务逻辑复杂，不太在乎执行效率的传统行业，一般会使用Hibernate

## ※Mybaits整体架构



1. 配置文件

全局配置文件（核心配置文件）：mybatis-config.xml，作用：配置数据源（配置数据库连接信息），引入映射文件

映射文件：XxMapper.xml，作用：配置sql语句、参数、结果集封装类型等

1. SqlSessionFactory

作用：获取SqlSession

通过newSqlSessionFactoryBuilder().build(inputStream)来构建，inputStream：读取配置文件的IO流

1. SqlSession

作用：执行CRUD操作

它是线程不安全的。因此最佳的使用范围是请求或方法范围。

1. Executor

执行器，SqlSession通过调用它来完成具体的CRUD

它是一个接口，提供了两种实现：缓存的实现、数据库的实现

1. Mapped Statement

在映射文件里面配置，包含3部分内容：

具体的sql，sql执行所需的参数类型，sql执行结果的封装类型

参数类型和结果集封装类型包括3种：

HashMap，基本数据类型，pojo

# 快速入门（Mybatis第一个程序）

需求：通过id查询user对象

## 步骤一：添加依赖

在中文官方文档中，见入门：



因此在pom.xml中添加mybatis的依赖：

<project xmlns=*"http://maven.apache.org/POM/4.0.0"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0*

*http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.itcast.mybatis</groupId>

<artifactId>mybatis\_day01</artifactId>

<version>1.0.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

</dependency>

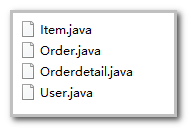
</dependencies>

</project>

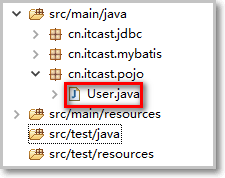
## 步骤二：编写实体类

既然要通过id查询user对象，必须添加一个user的实体类来封装数据。

pojo中的User.java参照资料中的pojo目录：



将其复制到工程中即可：

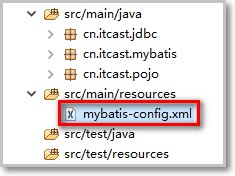


## 步骤三：编写全局配置文件

从mybatis的整体架构中可以看出，最终我们需要读取一个全局配置文件来生成sqlSessionFactory,进而获取sqlSession来操作数据库。

从官方文档可以看到从 XML 中构建 SqlSessionFactory的案例。

在resources下创建mybatis的全局配置文件：



<?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>

<!-- 配置环境：可以配置多个环境，default：配置某一个环境的唯一标识，表示默认使用哪个环境 -->

<environments default=*"development"*>

<!-- 配置环境,id:环境的唯一标识 -->

<environment id=*"development"*>

<!-- 事务管理器，type:使用jdbc的事务管理器 -->

<transactionManager type=*"JDBC"*/>

<!-- 数据源，type:数据源类型，池类型的数据源 -->

<dataSource type=*"POOLED"*>

<!-- 配置连接信息 -->

<property name=*"driver"* value=*"com.mysql.jdbc.Driver"*/>

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/mybatis"*/>

<property name=*"username"* value=*"root"*/>

<property name=*"password"* value=*"root"*/>

</dataSource>

</environment>

</environments>

<!-- 配置映射文件：用来配置sql语句和结果集类型等 -->

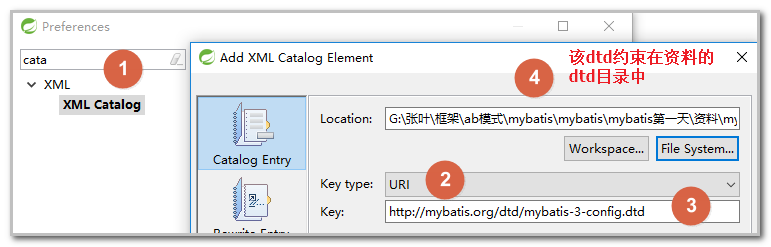
<mappers>

<mapper resource=*"UserMapper.xml"*/>

</mappers>

</configuration>

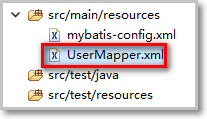
配置约束：



## 步骤四：编写映射文件

发现全局配置文件中需要管理映射文件，映射文件的配置可参考中文官方文档：

在resources目录下配置映射文件：



<?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标签:配置各类声明

namespace：名称空间，由于映射文件有多个，为了防止crud语句的唯一标识被重复，可以设置空间名称。

-->

<mapper namespace=*"UserMapper"*>

<!—

select:查询的statement（声明），用来编写查询语句

id:语句的唯一标识

resultType：配置返回的结果集类型

parameterType：传递的参数类型，可以省略

-->

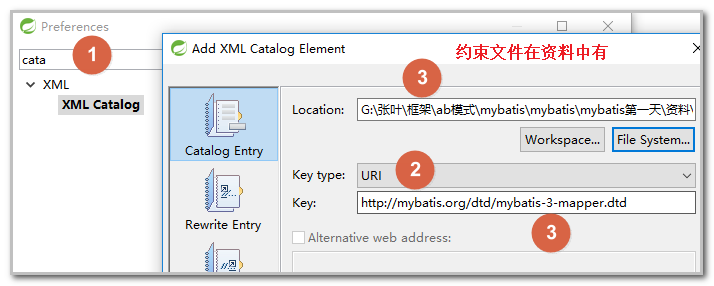
<select id=*"queryUserById"* resultType=*"User"*>

select \* from tb\_user where id = #{id}

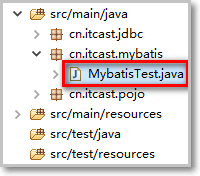
</select>

</mapper>

配置约束：



## 步骤五：编写测试类



内容如下:

**public** **class** MybatisTest {

**public** **static** **void** main(String[] args) **throws** IOException {

//获取全局配置文件输入流

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

//加载全局配置文件

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

//获取sqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

//第一个参数：名称空间.语句的唯一标识

//第二个参数：sql语句传递的参数

User user = sqlSession.selectOne("UserMapper.queryUserById",1L);

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

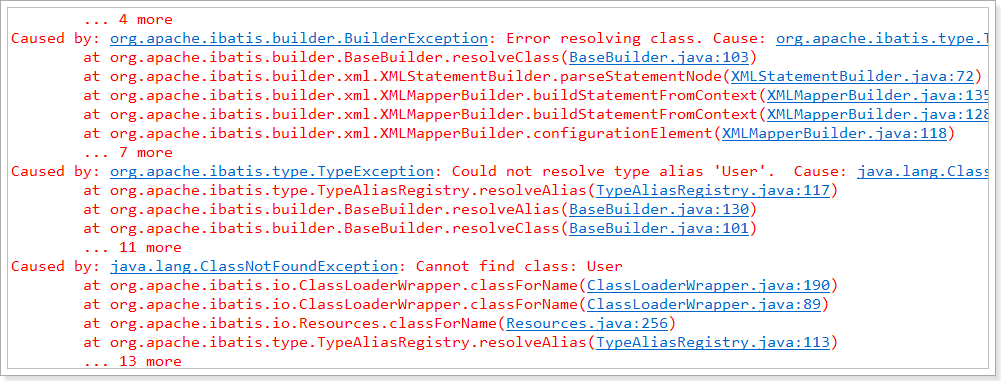
//释放资源

sqlSession.close();

}

}

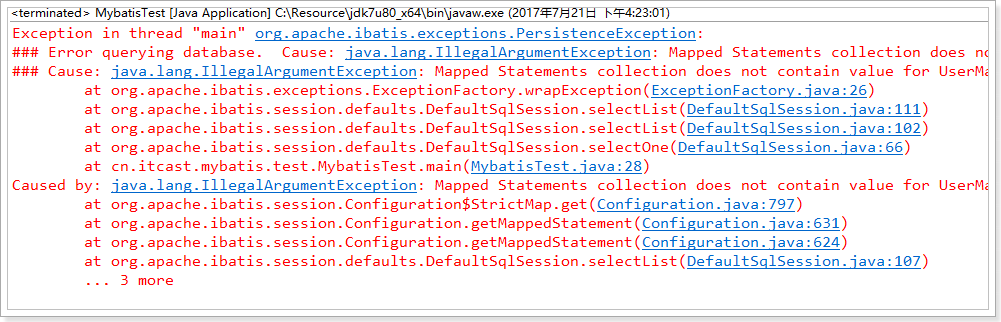
## 步骤六：测试运行报错



原因：



## 其他典型出错

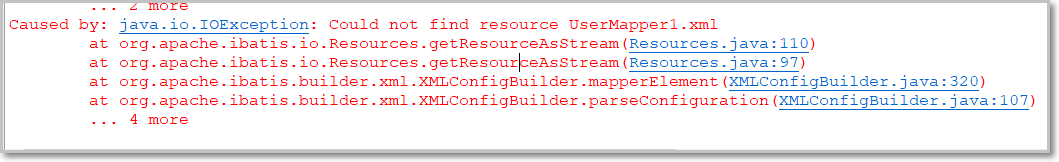


原因：



为了可以查看mybatis中发送的具体的sql，可以在工程中添加日志。

错误原因:可能是映射文件名称有问题，找不到。



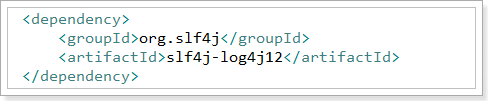
# ※引入log日志

打印日志2个步骤：

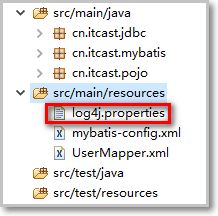
1. 在pom.xml中，引入slf4j的依赖
2. 在src/main/resources目录下添加log4j.properties文件

## 引入日志依赖包（pom.xml）

参照父工程



## 添加log4j.properties



内容如下：

### direct log messages to stdout ###

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.Target=System.out

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%-d{yyyy-MM-dd HH:mm:ss,SSS} [%t] [%c]-[%p] %m%n

### direct messages to file mylog.log ###

log4j.appender.file=org.apache.log4j.FileAppender

log4j.appender.file.File=f:/mylog.log

log4j.appender.file.layout=org.apache.log4j.PatternLayout

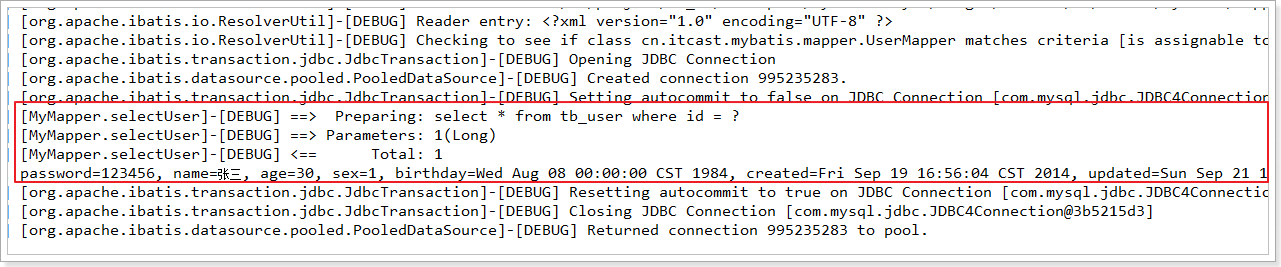
log4j.appender.file.layout.ConversionPattern=%-d{yyyy-MM-dd HH:mm:ss,SSS} [%t] [%c]-[%p] %m%n

### set log levels - for more verbose logging change 'info' to 'debug' ###

log4j.rootLogger=debug, stdout,file

## 日志输出

控制台截图：



## MyBatis使用步骤总结

配置步骤：

1. 配置全局配置文件

设置环境（事务、数据源）

管理映射文件XxxMapper.xml

1. 配置映射文件

Mapper标签：用来配置不同的statement

Namespace属性:用来配置名称空间，来区分不同的映射文件中的statement

编写crud的statement:select 、update 、insert 、delete

Select：专门用于查询的statement,可以编写查询语句

Id属性：该statement的唯一标识，一般用来被引用

resultType:结果集类型

paramterType:参数类型，可以省略,一般都省略

使用步骤：

1、获取全局配置文件的数据流，加载全局配置文件

2、通过全局配置文件的输入流创建SqlSessionFactory

3、通过SqlSessionFactory获取SqlSession

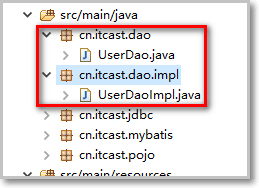
4、通过SqlSession操作crud

5、关闭SqlSession，释放资源。

# 完整的CRUD操作

接下来我们编写一个dao接口和实现类，来测试crud。

## 步骤一：创建接口和实现类



UserDao接口的内容：

**public** **interface** UserDao {

/\*\*

\* 根据id获取用户信息

\* **@param** id

\* **@return**

\*/

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

/\*\*

\* 查询所有用户

\* **@return**

\*/

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

/\*\*

\* 新增用户

\* **@param** user

\*/

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

/\*\*

\* 更新用户信息

\* **@param** user

\*/

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

/\*\*

\* 根据id删除用户信息

\* **@param** id

\*/

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

}

userDaoImp内容：

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

**private** SqlSession sqlSession;

**public** UserDaoImpl(SqlSession sqlSession) {

**this**.sqlSession = sqlSession;

}

@Override

**public** User queryUserById(Long id) {

User user = sqlSession.selectOne("UserDaoMapper.queryUserById", 1l);

**return** user;

}

@Override

**public** List<User> queryUserList() {

List<User> list = sqlSession.selectList("UserDaoMapper.queryUserList");

**return** list;

}

@Override

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

//返回值是该方法操作的记录数

**int** count = sqlSession.insert("UserDaoMapper.addUser", user);

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

sqlSession.commit();//提交事务

}

@Override

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

//返回值是该方法操作的记录数

**int** count = sqlSession.update("UserDaoMapper.updateUser", user);

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

sqlSession.commit();//提交事务

}

@Override

**public** **void** deleteUserById(Long id) {

//返回值是该方法操作的记录数

**int** count = sqlSession.delete("UserDaoMapper.deleteUserById", id);

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

sqlSession.commit();//提交事务

}

}

## 步骤二：编写UserDaoMapper.xml

编写对应的映射文件：

<?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标签:配置各类映射

namespace：名称空间，由于映射文件有多个，为了防止crud语句的唯一标识被重复，可以设置空间名称。

-->

<mapper namespace=*"UserDaoMapper"*>

<!--

select:编写查询语句

id:语句的唯一标识

resultType：配置返回的结果集类型

-->

<select id=*"queryUserById"* resultType=*"cn.itcast.pojo.User"*>

select \* from tb\_user where id = #{id}

</select>

<select id=*"queryUserList"* resultType=*"cn.itcast.pojo.User"*>

select \* from tb\_user

</select>

<!--

insert：插入的statement，编写插入语句

id:插入语句的唯一标识

parameterType：插入语句的参数，可以省略

-->

<insert id=*"insertUser"*>

INSERT INTO tb\_user (

user\_name,

password,

name,

age,

sex,

birthday,

created,

updated

)

VALUES

(

#{userName},

#{password},

#{name},

#{age},

#{sex},

#{birthday},

now(),

now()

);

</insert>

<!--

update:更新的statement,用来编写更新语句

id:语句的唯一标识

parameterType：语句的参数，可以省略

-->

<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:删除的statement,用来编写删除语句

id:语句的唯一标识

parameterType：语句的参数，可以省略

-->

<delete id=*"deleteUserById"*>

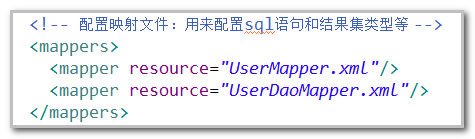
delete from tb\_user where id = #{id};

</delete>

</mapper>

## 步骤三：引入UserDaoMapper.xml

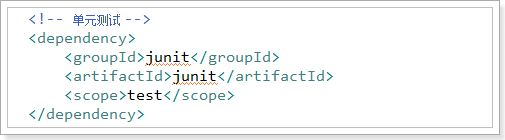
在mybatis-config.xml中引入UserDaoMapper.xml映射文件：



## 步骤四：添加junit依赖

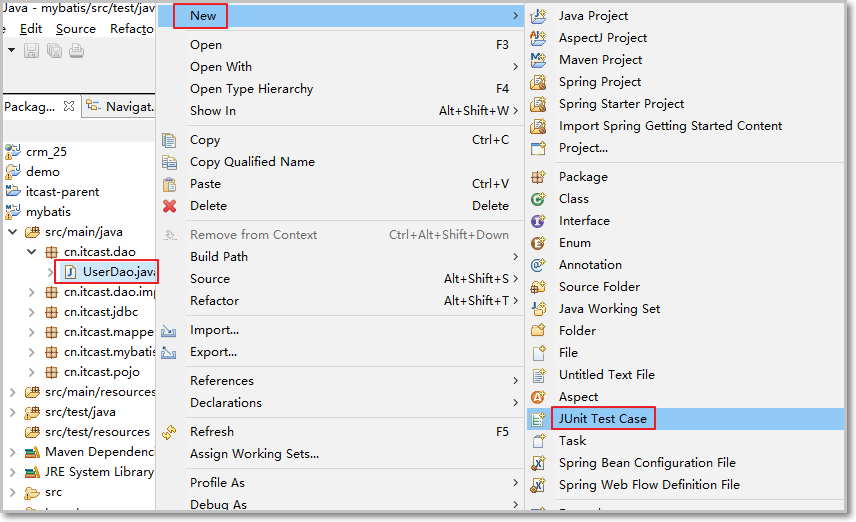
针对UserDao的测试，咱们使用Junit进行测试。

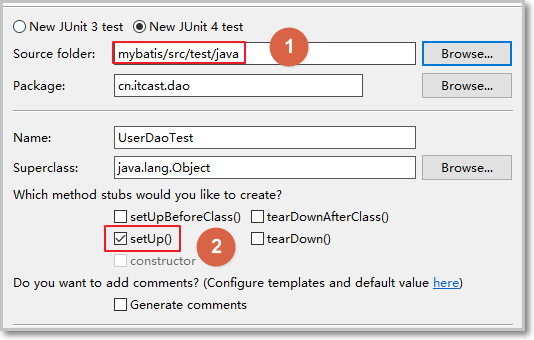
1. 引入Junit依赖：参照父工程，在mybatis工程中的pom.xml中引入junit的依赖

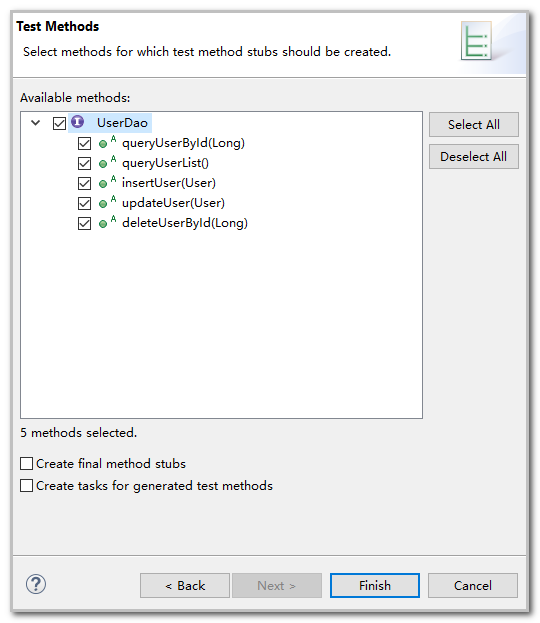


## 步骤五：测试UserDao

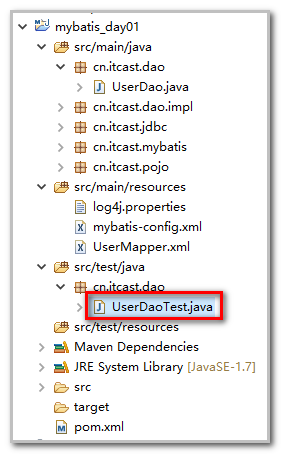
1. 创建junit测试用例：右键UserDao—>New—>Junit Test Case







创建完成之后的目录结构：



内如如下：

**public** **class** UserDaoTest {

**private** UserDao userDao;

@Before

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

//获取全局配置文件的输入流

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

//加载全局配置文件获取sqlSessionFactory

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

//获取sqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

userDao = **new** UserDaoImpl(sqlSession);

}

@Test

**public** **void** testQueryUserById() {

User user = userDao.queryUserById(1L);

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

}

@Test

**public** **void** testQueryUserList() {

List<User> list = userDao.queryUserList();

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

}

@Test

**public** **void** testInsertUser() {

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

user.setAge(16);

user.setBirthday(**new** Date());

user.setName("柳岩");

user.setPassword("123");

user.setSex(1);

user.setUserName("liuyan");

userDao.insertUser(user);

}

@Test

**public** **void** testUpdateUser() {

//先查询

User user = userDao.queryUserById(8L);

user.setAge(30);

//再更新

userDao.updateUser(user);

}

@Test

**public** **void** testDeleteUserById() {

userDao.deleteUserById(8L);

}

}

## 事务问题

如果在进行增删改的操作时，发现没有任何作用，那是因为增删改操作需要事务的提交。

因此我们可以在UserDaoImpl中设置事务的提交。

将UserDaoImpl修改如下:见红色代码

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

**private** SqlSession sqlSession;

**public** UserDaoImpl(SqlSession sqlSession) {

**this**.sqlSession = sqlSession;

}

@Override

**public** User queryUserById(Long id) {

User user = sqlSession.selectOne("UserDaoMapper.queryUserById", 1l);

**return** user;

}

@Override

**public** List<User> queryUserList() {

List<User> list = sqlSession.selectList("UserDaoMapper.queryUserList");

**return** list;

}

@Override

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

//返回值是该方法操作的记录数

**int** count = sqlSession.insert("UserDaoMapper.addUser", user);

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

sqlSession.commit();//提交事务

}

@Override

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

//返回值是该方法操作的记录数

**int** count = sqlSession.update("UserDaoMapper.updateUser", user);

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

sqlSession.commit();//提交事务

}

@Override

**public** **void** deleteUserById(Long id) {

//返回值是该方法操作的记录数

**int** count = sqlSession.delete("UserDaoMapper.deleteUserById", id);

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

sqlSession.commit();//提交事务

}

}

## 总结

//

1、 编写UserDao接口

2、 编写UserDao的实现类UserDaoImpl及映射文件UserDaoMapper.xml

3、 修改全局配置文件mybatis，引入UserDaoMapper.xml

4、 编写UserDao的Junit Test Case测试用例

## ※解决UserName为null

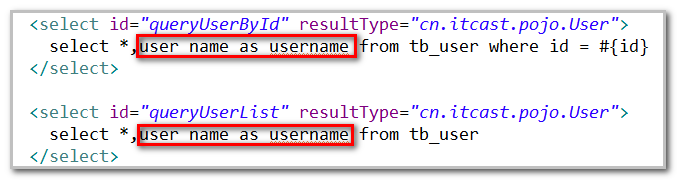
查询数据的时候，查不到userName的信息，原因：数据库的字段名是user\_name

POJO中的属性名是userName

两端不一致，造成mybatis无法填充对应的字段信息。

解决方案：在sql语句中使用别名

修改映射文件（UserDaoMapper.xml）中的sql语句，使用pojo对象中的属性名称作为别名，大小写无所谓，数据库默认不区分大小写。



# Mapper动态代理方式

## 思考CRUD中的问题

1、接口->实现类

2、每次CRUD实现类中需要传递SqlSession

3、sql statement 硬编码到java代码中。

思考：能否只写接口，不书写实现类，只编写Mapper.xml即可

因为在dao（mapper）的实现类中对sqlsession的使用方式很类似。mybatis提供了接口的动态代理

Mapper接口开发方法只需要程序员Mapper接口（相当于dao接口），由Mybatis框架根据接口定义创建接口的动态代理对象（相当于dao实现类），然后可以通过该代理对象进行增删改查的操作。区别是传统的编写方式dao的实现类我们需要自己创建和编写，而使用Mybatis框架的动态代理方式，我们只需要编写接口，实现类由Mabits为我们自动生成（即动态代理对象）。

Mapper接口的动态代理实现，需要遵循以下规范：

1. 映射文件中的命名空间（名称控制）与Mapper接口的全路径一致
2. 映射文件中的statement的Id与Mapper接口的方法名保持一致
3. 映射文件中的statement的ResultType必须和mapper接口方法的返回类型一致（即使不采用动态代理，也要一致）
4. 映射文件中的statement的parameterType必须和mapper接口方法的参数类型一致（不一定，该参数可省略）

## 使用动态代理改造CRUD

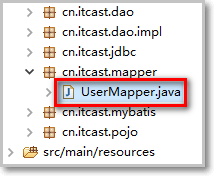
在mybatis中，持久层的XxxDao通常习惯上命名为XxxMapper（例如：以前命名为UserDao的接口，现在命名为UserMapper）。

当然，仍有少部分项目会保留Dao的命名，以后见到Mapper或者Dao的命名都不要奇怪。

采用动态代理之后，只剩下UserMapper接口、UserMapper.xml映射文件以及UserMapper接口的测试文件，即可以少写一个接口的实现类

### 步骤一：创建UserMapper接口

添加后：



UserMapper.java的内容参考UserDao.java的内容。（可以从UserDao直接Copy过来）

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

/\*\*

\* 通过id查询user对象

\* **@param** id

\* **@return**

\*/

User queryUserById(Long id);

/\*\*

\* 查询所有用户

\* **@return**

\*/

List<User> queryUserList();

/\*\*

\* 插入用户

\* **@param** user

\*/

**void** insertUser(User user);

/\*\*

\* 更新用户

\* **@param** user

\*/

**void** updateUser(User user);

/\*\*

\* 通过id删除用户

\* **@param** id

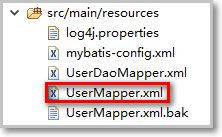
\*/

**void** deleteUserById(Long id);

}

### 步骤二：UserMapper映射文件

在之前的UserMapper.xml上直接修改即可：



对应的UserMapper映射文件，之前咱们已创建，但是内容参考UserDaoMapper。

名称空间必须改成UserMapper接口的全路径，StatementId必须和接口方法名一致，结果集的封装类型已经和方法的返回类型一致

<?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标签:配置各类映射

namespace：名称空间，由于映射文件有多个，为了防止crud语句的唯一标识被重复，可以设置空间名称。

-->

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

<!--

select:查询的statement,用来编写查询语句

id:语句的唯一标识

resultType：配置返回的结果集类型

-->

<select id=*"queryUserById"* resultType=*"cn.itcast.pojo.User"*>

select \*,user\_name as username from tb\_user where id = #{id}

</select>

<select id=*"queryUserList"* resultType=*"cn.itcast.pojo.User"*>

select \*,user\_name as username from tb\_user

</select>

<!--

insert：编写插入语句

id:插入语句的唯一标识

parameterType：插入语句的参数

-->

<insert id=*"insertUser"*>

INSERT INTO tb\_user (

user\_name,

password,

name,

age,

sex,

birthday,

created,

updated

)

VALUES

(

#{userName},

#{password},

#{name},

#{age},

#{sex},

#{birthday},

now(),

now()

);

</insert>

<!--

update:更新语句

id:语句的唯一标识

parameterType：语句的参数

-->

<update id=*"updateUser"*>

UPDATE tb\_user

SET id = #{id},

user\_name = #{userName},

password = #{password},

name = #{name},

age = #{age},

sex = #{sex},

birthday = #{birthday},

updated = now()

WHERE

(id = #{id});

</update>

<!--

delete:删除语句

id:语句的唯一标识

parameterType：语句的参数

-->

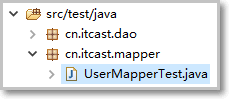
<delete id=*"deleteUserById"*>

delete from tb\_user where id = #{id};

</delete>

</mapper>

### 步骤三：UserMapperTest测试



添加UserMapper.java的测试用例。

sqlSession.getMapper(UserMapper.class)方法会通过动态代理生成一个代理的实现类。

仅仅时为了演示方法，以下方法的演示中就不再执行sqlSession.close()了：

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

**private** UserMapper userMapper;

@Before

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

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

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

SqlSession sqlSession = sqlSessionFactory.openSession();

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

}

@Test

**public** **void** testQueryUserById() {

User user = userMapper.queryUserById(1L);

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

}

@Test

**public** **void** testQueryUserList() {

List<User> list = userMapper.queryUserList();

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

}

@Test

**public** **void** testInsertUser() {

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

user.setAge(16);

user.setBirthday(**new** Date());

user.setCreated(user.getBirthday());

user.setName("范爷");

user.setPassword("123");

user.setSex(1);

user.setUpdated(user.getBirthday());

user.setUserName("fanye");

userMapper.insertUser(user);

}

@Test

**public** **void** testUpdateUser() {

//先查询

User user = userMapper.queryUserById(8L);

user.setAge(30);

//再更新

userMapper.updateUser(user);

}

@Test

**public** **void** testDeleteUserById() {

userMapper.deleteUserById(8L);

}

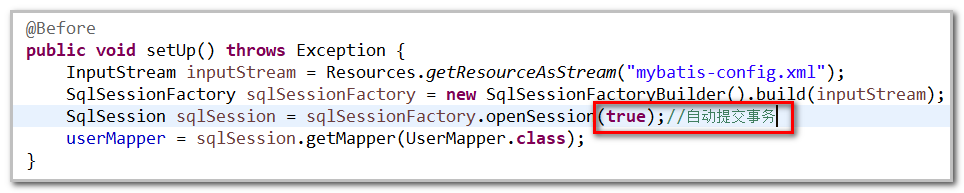
}

问题：发现在测试的时候，查询操作都没有什么问题，但是增删改没有反应。

原因：原来在UserDaoImpl中我们是手动提交事务的，但是用来动态代理生成的代理对象的话，并没有提交事务。

解决方案：设置为自动提交即可。

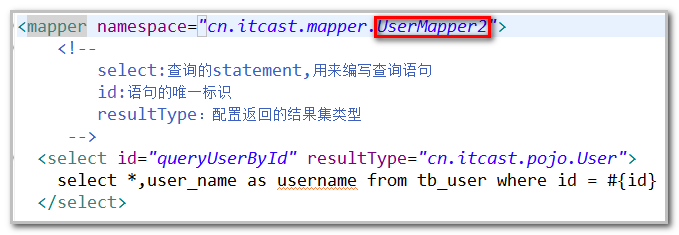
修改代码如下：



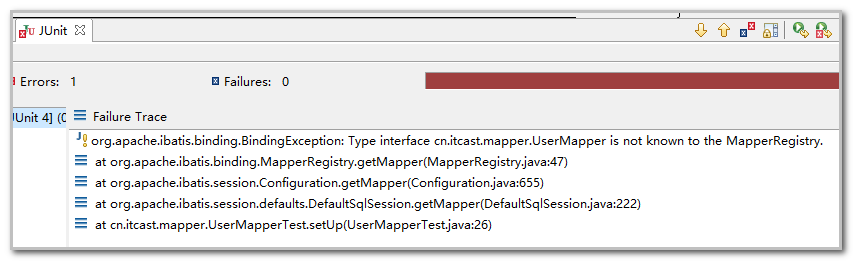
再次测试crud无误。

### 注意事项

如果名称空间不和mapper接口的全路径保持一致



报错如下：



### 问题思考

思考：接口中是否可以有重载的方法？

注意:由于使用的是接口方式进行查询（动态代理），而其它要求映射文件中的statement的Id与Mapper接口的方法名保持一致。

而statement的id由于是唯一的不能重复，因此也就意味着接口中的方法也不能重名，那也就是说不能有重载的方法了。

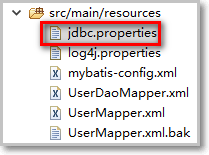
# mybatis-config.xml配置

mybatis-config.xml讲究严格的顺序，具体配置顺序遵循文档的顺序：



## properties属性读取外部资源

在src下添加jdbc.properties资源文件：



jdbc.properties资源文件内容：

driver=com.mysql.jdbc.Driver

url=jdbc:mysql://127.0.0.1:3306/mybatis

username=root

password=root

在Mybatis-config.xml中引入jdbc.properties资源文件：

<?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>

<!-- 引入外部资源文件

resource:默认引入classpath路径下的资源文件

-->

<properties resource=*"jdbc.properties"*></properties>

<!-- 配置环境：可以配置多个环境，default：配置某一个环境的唯一标识，表示默认使用哪个环境 -->

<environments default=*"development"*>

<!-- 配置环境,id:环境的唯一标识 -->

<environment id=*"development"*>

<!-- 事务管理器，type:使用jdbc的事务管理器 -->

<transactionManager type=*"JDBC"* />

<!-- 数据源，type:池类型的数据源 -->

<dataSource type=*"POOLED"*>

<!-- 配置连接信息 -->

<property name=*"driver"* value=*"${driverClass}"* />

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

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

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

</dataSource>

</environment>

</environments>

<!-- 配置映射文件：用来配置sql语句和结果集类型等 -->

<mappers>

<mapper resource=*"UserMapper.xml"* />

<mapper resource=*"UserDaoMapper.xml"* />

</mappers>

</configuration>

通过properties引入外部资源文件之后，就可以通过${xxx}的方式使用资源文件里的参数了。

## settings设置

settings参数有很多，咱们只需要学习以下4个参数就行了，而今天咱们先学习驼峰匹配：

|  |  |  |  |
| --- | --- | --- | --- |
| 设置参数 | 描述 | 有效值 | 默认值 |
| cacheEnabled | 该配置影响的所有映射器中配置的缓存的全局开关。 | true | false | true |
| lazyLoadingEnabled | 延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。特定关联关系中可通过设置fetchType属性来覆盖该项的开关状态。 | true | false | false |
| aggressiveLazyLoading | 当启用时，带有延迟加载属性的对象的加载与否完全取决于对任意延迟属性的调用；反之，每种属性将会按需加载。 | true | false | true |
| mapUnderscoreToCamelCase | 是否开启自动驼峰命名规则（camel case）映射，即从经典数据库列名 A\_COLUMN 到经典 Java 属性名 aColumn 的类似映射。 | true | false | False |

开启驼峰匹配：完成经典的数据库命名到java属性的映射

经典数据库命名：如果多个单词之间，通常使用下划线进行连接。

java中命名：第二个单词首字母大写。

驼峰匹配：相当于去掉数据中的名字的下划线，和java进行匹配

查询数据的时候，查不到userName的信息，原因：数据库的字段名是user\_name，POJO中的属性名字是userName，两端不一致，造成mybatis无法填充对应的字段信息。修改方法：在sql语句中使用别名

解决方案1：在sql语句中使用别名

解决方案2：参考驼峰匹配 --- mybatis-config.xml

mybatis-config.xml中开启驼峰匹配：

<?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>

<!-- 引入外部资源文件

resource:默认引入classpath路径下的资源文件

-->

<properties resource=*"jdbc.properties"*></properties>

<!-- 设置驼峰匹配:从数据库列名映射java属性名称的经典方式 -->

<settings>

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

</settings>

<!-- 配置环境：可以配置多个环境，default：配置某一个环境的唯一标识，表示默认使用哪个环境 -->

<environments default=*"development"*>

<!-- 配置环境,id:环境的唯一标识 -->

<environment id=*"development"*>

<!-- 事务管理器，type:使用jdbc的事务管理器 -->

<transactionManager type=*"JDBC"* />

<!-- 数据源，type:池类型的数据源 -->

<dataSource type=*"POOLED"*>

<!-- 配置连接信息 -->

<property name=*"driver"* value=*"${driverClass}"* />

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

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

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

</dataSource>

</environment>

</environments>

<!-- 配置映射文件：用来配置sql语句和结果集类型等 -->

<mappers>

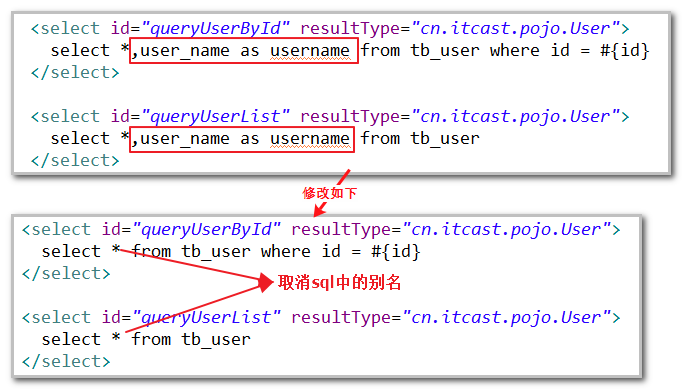
<mapper resource=*"UserMapper.xml"* />

<mapper resource=*"UserDaoMapper.xml"* />

</mappers>

</configuration>

在UserMapper.xml中修改sql语句删除别名



删除sql中的别名，开启驼峰匹配后，仍然能够获取用户名，说明驼峰匹配起到了作用。

## typeAliases

之前咱们在映射文件中用到java类型时，都是使用类的全路径，书写起来非常麻烦

解决方案：

类型别名是为 Java 类型命名的一个短的名字。它只和 XML 配置有关，存在的意义仅在于用来减少类完全限定名的冗余。（官方文档）

### 方式一：typeAlias

第一步：修改全局配置文件

<?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>

<!-- 引入外部资源文件

resource:默认引入classpath路径下的资源文件

url：引入物理路径下的资源文件（如：d:\\jdbc.properties）

-->

<properties resource=*"jdbc.properties"*></properties>

<!-- 设置参数 -->

<settings>

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

</settings>

<!-- 配置别名 -->

<typeAliases>

<!-- typeAlias:用来配置别名，方便映射文件使用，type:类的全限定类名,alias:别名 -->

<typeAlias type=*"cn.itcast.pojo.User"* alias=*"User"*/>

</typeAliases>

<!-- 配置环境：可以配置多个环境，default：配置某一个环境的唯一标识，表示默认使用哪个环境 -->

<environments default=*"development"*>

<!-- 配置环境,id:环境的唯一标识 -->

<environment id=*"development"*>

<!-- 事务管理器，type:使用jdbc的事务管理器 -->

<transactionManager type=*"JDBC"* />

<!-- 数据源，type:池类型的数据源 -->

<dataSource type=*"POOLED"*>

<!-- 配置连接信息 -->

<property name=*"driver"* value=*"${driverClass}"* />

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

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

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

</dataSource>

</environment>

</environments>

<!-- 配置映射文件：用来配置sql语句和结果集类型等 -->

<mappers>

<mapper resource=*"UserMapper.xml"* />

<mapper resource=*"UserDaoMapper.xml"* />

</mappers>

</configuration>

第二步：修改映射文件

<?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标签:配置各类映射

namespace：名称空间，由于映射文件有多个，为了防止crud语句的唯一标识被重复，可以设置空间名称。

-->

<mapper namespace=*"UserDaoMapper"*>

<!--

select:查询的statement,用来编写查询语句

id:语句的唯一标识

resultType：配置返回的结果集类型

-->

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

select \* from tb\_user where id = #{id}

</select>

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

select \* from tb\_user

</select>

<!--

insert：编写插入语句

id:插入语句的唯一标识

parameterType：插入语句的参数

-->

<insert id=*"insertUser"* parameterType=*"User"*>

INSERT INTO tb\_user (

user\_name,

password,

name,

age,

sex,

birthday,

created,

updated

)

VALUES

(

#{userName},

#{password},

#{name},

#{age},

#{sex},

#{birthday},

now(),

now()

);

</insert>

<!--

update:更新语句

id:语句的唯一标识

parameterType：语句的参数

-->

<update id=*"updateUser"* parameterType=*"User"*>

UPDATE tb\_user

SET id = #{id},

user\_name = #{userName},

password = #{password},

name = #{name},

age = #{age},

sex = #{sex},

birthday = #{birthday},

updated = now()

WHERE

(id = #{id});

</update>

<!--

delete:删除语句

id:语句的唯一标识

parameterType：语句的参数

-->

<delete id=*"deleteUserById"* parameterType=*"java.lang.Long"*>

delete from tb\_user where id = #{id};

</delete>

</mapper>

缺点：如果由多个pojo,那么每个pojo类都要去配置别名。

### 方式二：package

扫描指定包下的所有类，扫描之后的别名就是类名，大小写不敏感（不区分大小写），建议使用的时候和类名一致。

<!-- 配置别名 -->

<typeAliases>

<!-- typeAlias:用来配置别名，方便映射文件使用，type:类的全限定类名,alias:别名 -->

<!-- <typeAlias type="cn.itcast.pojo.User" alias="User"/> -->

<!-- 配置包扫描：解决多个pojo别名问题，默认将类名作为笔名，大小写不敏感 -->

<package name=*"cn.itcast.pojo"*/>

</typeAliases>

### ※内置别名

已经为普通的 Java 类型内建了许多相应的类型别名。它们都是大小写不敏感的：

|  |  |
| --- | --- |
| 别名 | 映射的类型 |
| \_byte | byte |
| \_long | long |
| \_short | short |
| \_int | int |
| \_integer | int |
| \_double | double |
| \_float | float |
| \_boolean | boolean |
| string | String |
| byte | Byte |
| long | Long |
| short | Short |
| int | Integer |
| integer | Integer |
| double | Double |
| float | Float |
| boolean | Boolean |
| date | Date |
| decimal | BigDecimal |
| bigdecimal | BigDecimal |
| object | Object |
| map | Map |
| hashmap | HashMap |
| list | List |
| arraylist | ArrayList |
| collection | Collection |
| iterator | Iterator |

因此，我们也可以将删除的映射文件修改如下:

<delete id=*"deleteUserById"* parameterType=*"long"*>

delete from tb\_user where id = #{id};

</delete>

## typeHandlers（类型处理器）-了解

无论是 MyBatis 在预处理语句（PreparedStatement）中设置一个参数时，还是从结果集中取出一个值时，都会用类型处理器将获取的值以合适的方式转换成 Java 类型。下表描述了一些默认的类型处理器。

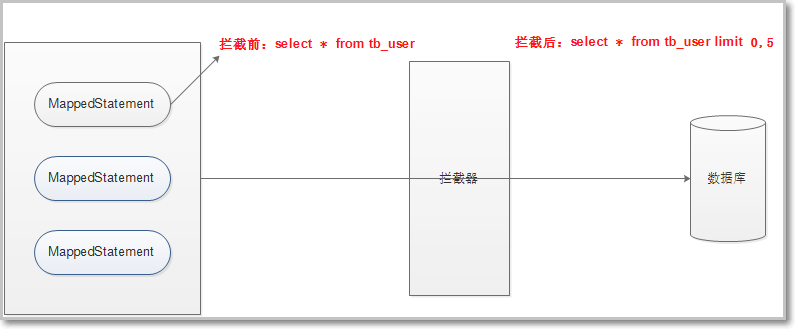
比如：数据库 中的id类型为bigint,而java对象中的id类型Long,之所以可以自动封装就是因为mybatis的类型处理器来帮助我们完成的。

|  |  |  |
| --- | --- | --- |
| 类型处理器 | Java 类型 | JDBC 类型 |
| BooleanTypeHandler | java.lang.Boolean, boolean | 任何兼容的布尔值 |
| ByteTypeHandler | java.lang.Byte, byte | 任何兼容的数字或字节类型 |
| ShortTypeHandler | java.lang.Short, short | 任何兼容的数字或短整型 |
| IntegerTypeHandler | java.lang.Integer, int | 任何兼容的数字和整型 |
| LongTypeHandler | java.lang.Long, long | 任何兼容的数字或长整型 |
| FloatTypeHandler | java.lang.Float, float | 任何兼容的数字或单精度浮点型 |
| DoubleTypeHandler | java.lang.Double, double | 任何兼容的数字或双精度浮点型 |
| BigDecimalTypeHandler | java.math.BigDecimal | 任何兼容的数字或十进制小数类型 |
| StringTypeHandler | java.lang.String | CHAR 和 VARCHAR 类型 |
| ClobTypeHandler | java.lang.String | CLOB 和 LONGVARCHAR 类型 |
| NStringTypeHandler | java.lang.String | NVARCHAR 和 NCHAR 类型 |
| NClobTypeHandler | java.lang.String | NCLOB 类型 |
| ByteArrayTypeHandler | byte[] | 任何兼容的字节流类型 |
| BlobTypeHandler | byte[] | BLOB 和 LONGVARBINARY 类型 |
| DateTypeHandler | java.util.Date | TIMESTAMP 类型 |
| DateOnlyTypeHandler | java.util.Date | DATE 类型 |
| TimeOnlyTypeHandler | java.util.Date | TIME 类型 |
| SqlTimestampTypeHandler | java.sql.Timestamp | TIMESTAMP 类型 |
| SqlDateTypeHandler | java.sql.Date | DATE 类型 |
| SqlTimeTypeHandler | java.sql.Time | TIME 类型 |
| ObjectTypeHandler | Any | 其他或未指定类型 |
| EnumTypeHandler | Enumeration Type | VARCHAR-任何兼容的字符串类型，作为代码存储（而不是索引） |
| EnumOrdinalTypeHandler | Enumeration Type | 任何兼容的 NUMERIC 或 DOUBLE 类型，作为位置存储（而不是代码本身）。 |

## plugins(插件又称拦截器)-了解

MyBatis 允许你在已映射语句执行过程中的某一点进行拦截调用。默认情况下，MyBatis支持使用插件对以下方法(四大对象)的拦截：

* Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)
* ParameterHandler (getParameterObject, setParameters)
* ResultSetHandler (handleResultSets, handleOutputParameters)
* StatementHandler (prepare, parameterize, batch, update, query)





## environments(环境)

MyBatis 可以配置成适应多种环境，例如，开发、测试和生产环境需要有不同的配置；

尽管可以配置多个环境，每个 SqlSessionFactory 实例只能选择其一。

虽然，这种方式也可以做到很方便的分离多个环境，但是实际使用场景下，我们更多的是选择使用spring来管理数据源，来做到环境的分离。

### 方法一：default

添加一个test（测试）环境，并在default参数中指向test环境。

<?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>

<!-- 引入外部资源文件

resource:默认引入classpath路径下的资源文件

-->

<properties resource=*"jdbc.properties"*></properties>

<!-- 设置参数 -->

<settings>

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

</settings>

<!-- 配置别名 -->

<typeAliases>

<!-- typeAlias:用来配置别名，方便映射文件使用，type:类的全限定类名,alias:别名 -->

<!-- <typeAlias type="cn.itcast.pojo.User" alias="User"/> -->

<!-- 配置包扫描：解决多个pojo别名问题，默认将类名作为笔名，大小写不敏感 -->

<package name=*"cn.itcast.pojo"*/>

</typeAliases>

<!-- 配置环境：可以配置多个环境，default：配置某一个环境的唯一标识，表示默认使用哪个环境 -->

<environments default=*"test"*>

<!-- 配置环境,id:环境的唯一标识 -->

<environment id=*"development"*>

<!-- 事务管理器，type:使用jdbc的事务管理器 -->

<transactionManager type=*"JDBC"* />

<!-- 数据源，type:池类型的数据源 -->

<dataSource type=*"POOLED"*>

<!-- 配置连接信息 -->

<property name=*"driver"* value=*"${driverClass}"* />

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

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

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

</dataSource>

</environment>

<!-- 配置测试环境 -->

<environment id=*"test"*>

<!-- 事务管理器，type:使用jdbc的事务管理器 -->

<transactionManager type=*"JDBC"* />

<!-- 数据源，type:池类型的数据源 -->

<dataSource type=*"POOLED"*>

<!-- 配置连接信息 -->

<property name=*"driver"* value=*"${driverClass}"* />

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

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

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

</dataSource>

</environment>

</environments>

<!-- 配置映射文件：用来配置sql语句和结果集类型等 -->

<mappers>

<mapper resource=*"UserMapper.xml"* />

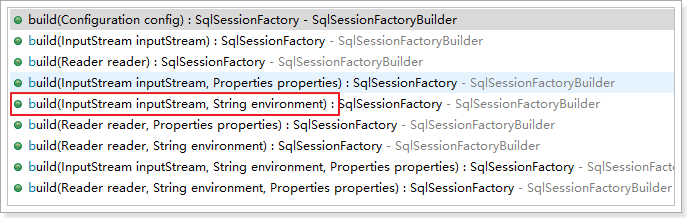
<mapper resource=*"UserDaoMapper.xml"* />

</mappers>

</configuration>

### 方法二：build方法

有时候为了测试方便，可以直接在代码中进行数据源的设置，通过build方法的重载方法：



setUp()方法修改如下：

@Before

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

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

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream,"test");

SqlSession sqlSession = sqlSessionFactory.openSession(**true**);//自动提交事务

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

}

## Mappers

既然 MyBatis 的行为已经由上述元素配置完了，我们现在就要定义 SQL 映射语句了。但是首先我们需要告诉 MyBatis 到哪里去找到这些语句。Java 在自动查找这方面没有提供一个很好的方法，所以最佳的方式是告诉 MyBatis 到哪里去找映射文件。

### 方式一：resource

在mybatis-config.xml引入项目目录下的映射文件：



缺点：有多少个映射文件就要配置多少次，麻烦。

### 方式二：url（不采用-了解）

将UserMapper.xml映射文件放入d盘下，然后在全局配置文件中引入该映射文件。

引入硬盘目录下的映射文件：

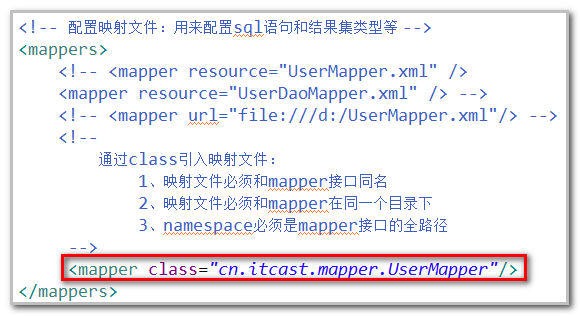


缺点：

1. 硬盘的位置可能随着项目的部署或迁移，路径发生变化
2. 每新增一个映射文件，就要在全局配置文件中引入

### 方式三：class

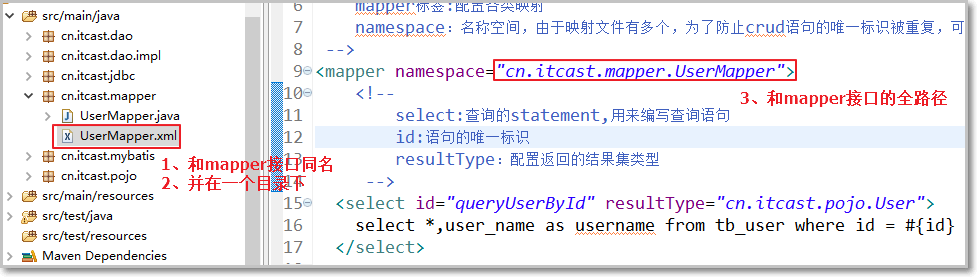
在mybatis-config.xml配置mapper接口的全路径：



这种配置方式，在mybatis-config.xml配置文件中配置了mapper接口的全路径，并没有配置mapper接口的映射文件的位置。如果要让mybatis找到对应的映射文件，则必须满足一定的条件或规则：

1. 映射文件和mapper接口在同一个目录下
2. 文件名必须一致
3. 映射文件的namespace必须和mapper接口的全路径保持一致

目录结构：



缺点：

1. java文件和xml映射文件耦合
2. 每新增一个映射文件，就要在全局配置文件中引入

### 方式四：package

在mybatis-config.xml中，开启包扫描：



原理：扫描目标包目录下的mapper接口，并按照class的方式找到接口对应的映射文件。

其条件和设置class的条件一样：

1. 映射文件和mapper接口在同一个目录下
2. 文件名必须一致

3、映射文件的namespace必须和mapper接口的全路径保持一致

缺点：

1、需要遵守的条件太多

2、mapper.xml和mapper.java没有分离。

## 总结//

Properties:引入外部资源文件

Settings:配置驼峰命名方式，类似经典的数据库字段名和java类中属性的映射关系。

typeAliases:配置别名

typeHandler:类型转换器-了解

plugings:插件，又称之为拦截器（分页助手、通用mapper）

environments:可以配置多个环境

environment:配置环境

transactionManager:配置事务管理器

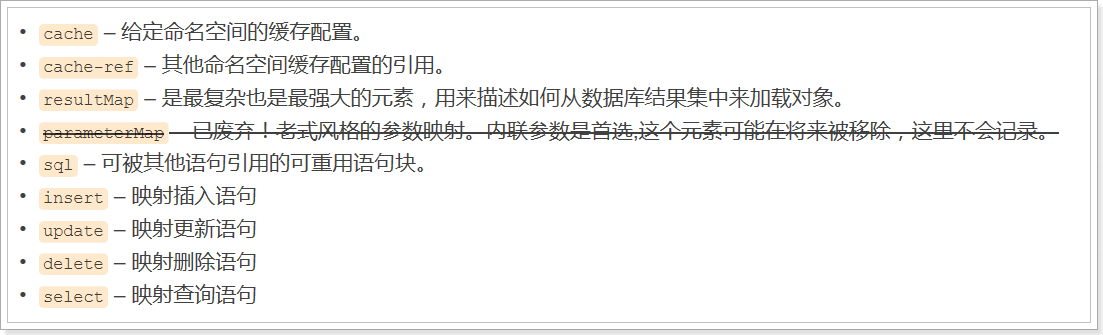
dataSource:配置数据源

mappers:配置映射文件

Package:通过包名配置映射文件

# Mapper XML 文件（映射文件）

Mapper映射文件中定义了操作数据库的sql,每一个sql都被包含在一个statement中。映射文件是mybatis操作数据库的核心。



## CRUD标签

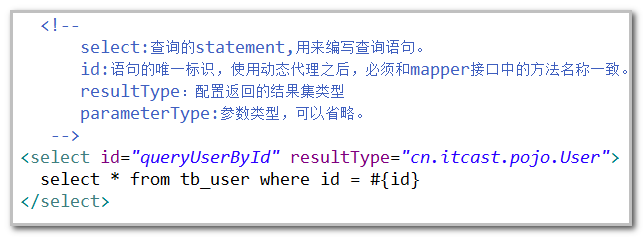
### Select

Select标签：用来编写查询语句的statement

id属性：唯一标识

resultType:查询语句返回的结果集类型

parameterType:参数类型，使用动态代理之后，需要和mapper接口中的参数类型一致，可以省略。



### Insert

Insert标签：编写新增语句的statement

insert：编写插入语句

id:插入语句的唯一标识

parameterType：插入语句的参数类型，使用动态代理之后，需要和mapper接口中的参数类型一致，可以省略。

useGeneratedKeys：开启主键自增回显，将自增长的主键值回显到形参中（即封装到User对象中）

keyColumn：数据库中主键的字段名称

keyProperty：pojo中主键对应的属性

<!--

insert：编写插入语句

id:插入语句的唯一标识

parameterType：插入语句的参数类型，可以省略。

useGeneratedKeys：开启主键自增回显，将自增长的主键值回显到形参中（即封装到User对象中）

keyColumn：数据库中主键的字段名称

keyProperty：pojo中主键对应的属性

-->

<insert id=*"insertUser"* parameterType=*"User"* 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>

修改新增的测试方法：

@Test

**public** **void** testInsertUser() {

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

user.setAge(16);

user.setBirthday(**new** Date());

user.setCreated(user.getBirthday());

user.setName("范海辛");

user.setPassword("123");

user.setSex(1);

user.setUpdated(user.getBirthday());

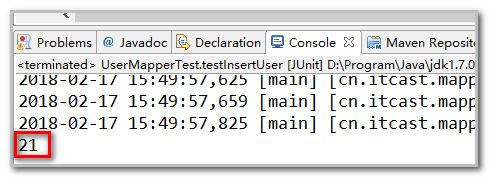
user.setUserName("fanhaixin");

userMapper.insertUser(user);

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

}

结果：发现id已经被回写。

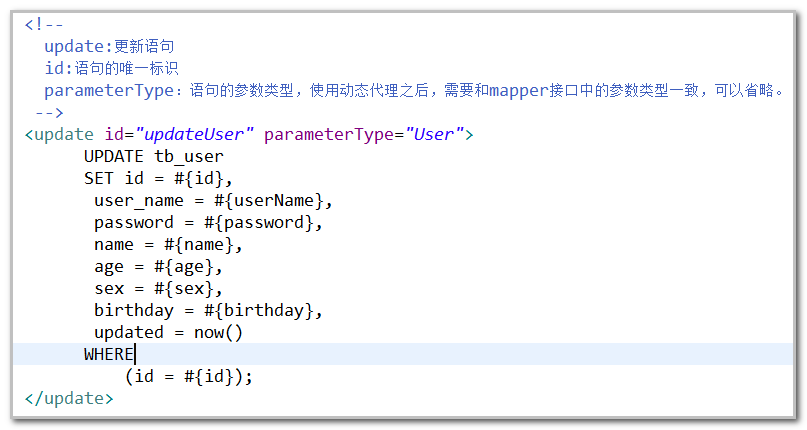


### Update

Update标签：编写更新语句的statement

id:语句的唯一标识

parameterType：语句的参数类型，使用动态代理之后，需要和mapper接口中的参数类型一致，可以省略。

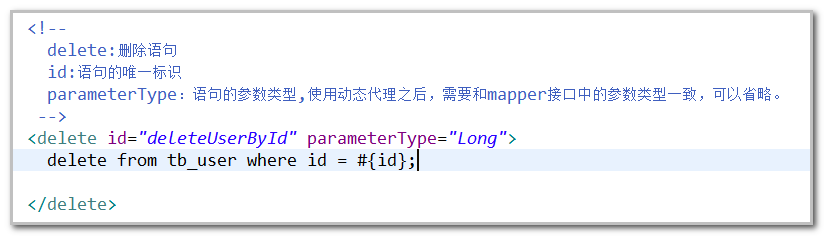


### Delete

delete标签：编写删除语句的statement

id:语句的唯一标识

parameterType：语句的参数类型,使用动态代理之后，需要和mapper接口中的参数类型一致，可以省略。



## parameterType传入参数

CRUD标签都有一个属性parameterType，statement通过它指定接收的参数类型。

接收参数的方式有两种：

1. #{}预编译

2、${}非预编译（直接的sql拼接，不能防止sql注入）

参数类型有三种：

1. 基本数据类型
2. HashMap（使用方式和pojo类似）
3. Pojo自定义包装类型

### ${}的用法

场景：数据库有两个一模一样的表。历史表，当前表

查询表中的信息，有时候从历史表中去查询数据，有时候需要去新的表去查询数据。希望使用1个方法来完成操作。

【示例】

第一步：添加一个接口

在UserMapper接口中，添加根据表名查询用户信息的方法：



第二步：编写一个查询的statement

在UserMapper映射文件中，添加方法对应的statement：

<!-- 通过表名查询数据 -->

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

select \* from ${tableName}

</select>

第三步：编写测试方法

@Test

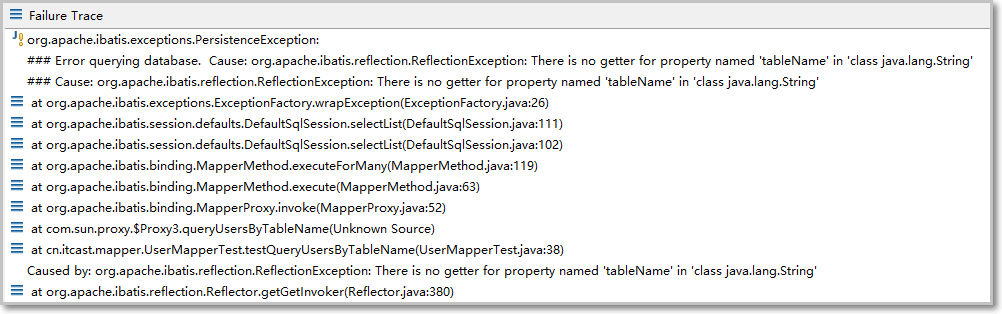
**public** **void** testQueryUserListByTableName(){

List<User> list = userMapper.queryUserListByTableName("tb\_user");

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

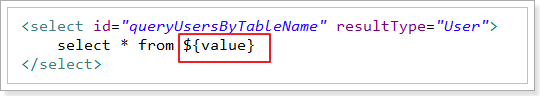
}

执行方法输出（报错）：只能使用${value}来传递参数，其他任何字符串都不行



如果你要动态传入的字段名是表名，并且sql执行是预编译的，这显然是不允许的，所以你必须改成非预编译的，也就是这样：

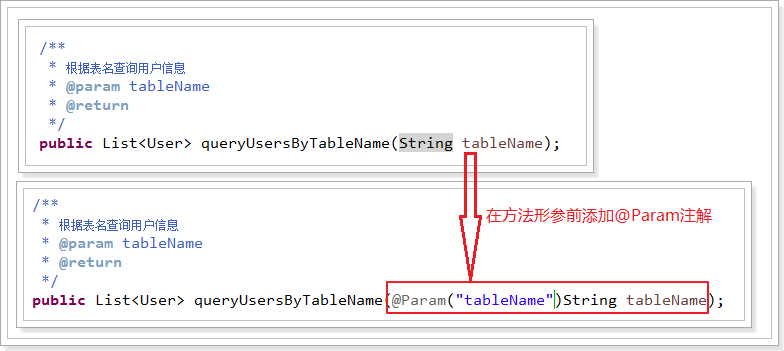
在一个参数的情况下必须，参数名称必须为value:即${value}

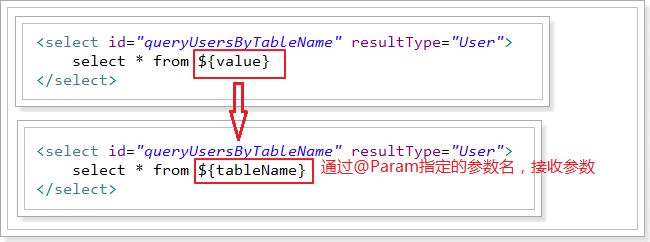


注意：

使用${} 去接收参数信息，在一个参数时，默认情况下必须使用${value}获取参数值，

但是这并不是一种稳妥的解决方案，推荐使用@Param注解指定参数名，所以以上接口及映射文件可以改成如下：





一个参数时， ${}默认必须通过value来接收参数。

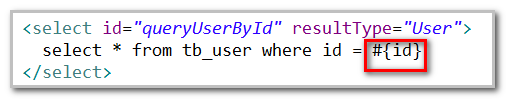
可以通过@Param注解指定参数名

### #{}的用法

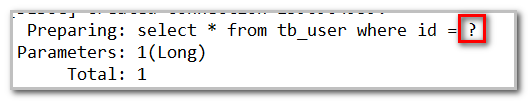
#{}类似于sql语句中的？，但是单个参数和多个参数的使用方式稍微有点区别。

#### 单个参数

参见之前的通过id查询数据的statement:

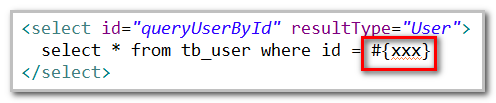


执行查询结果如下：#{}就相当于一个？，可以进行预编译的占位。



**注意：在一个参数的情况下#{id}中的id可以修改成任意字符。**

即使修改如下，也可以照常运行：



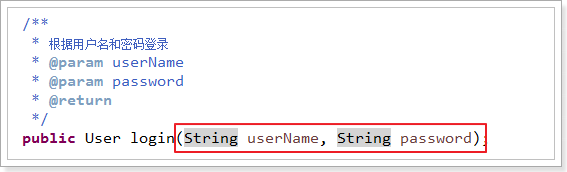
#### 多个参数

当mapper接口要传递多个参数时，有两种传递参数的方法：

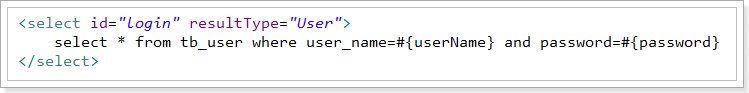
1. 默认规则获取参数{0,1,param1,param2}
2. 使用@Param注解指定参数名

案例：实现一个简单的用户登录，根据username和password验证用户信息

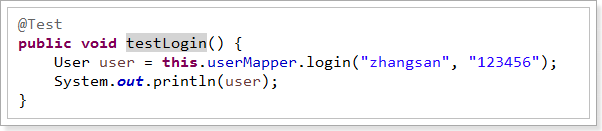
在UserMapper接口中，添加登陆方法：



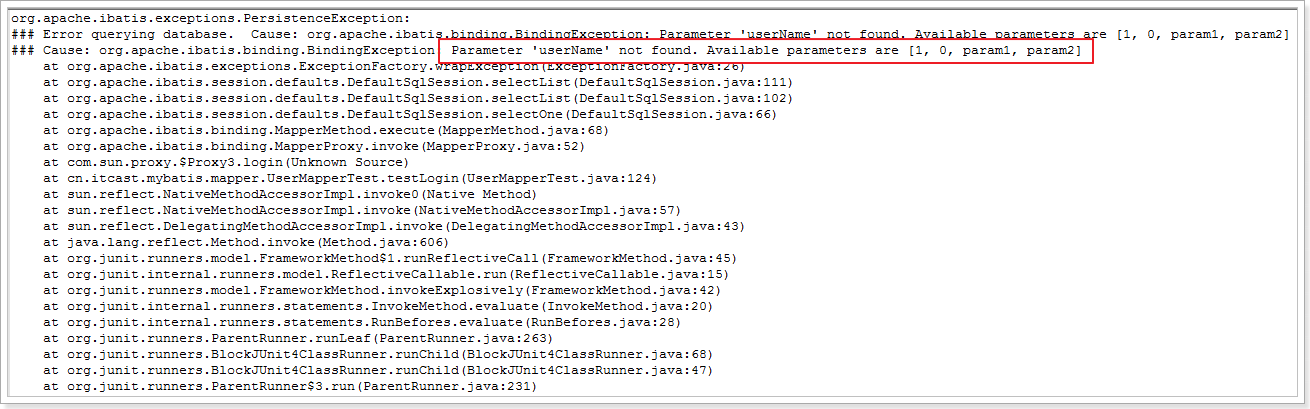
在UserMapper.xml配置中，添加登陆方法对应的Statement配置：



在UserMapperTest测试用例中，添加测试方法：



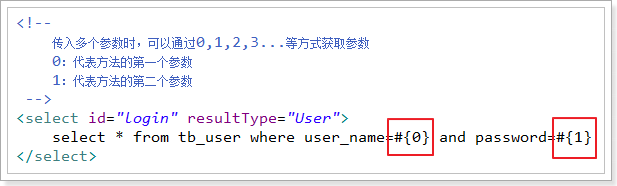
运行报错：



注意报错信息：没有找到参数userName，可用的参数是:[1,0,param1,param2]，所以可有以下解决方案：

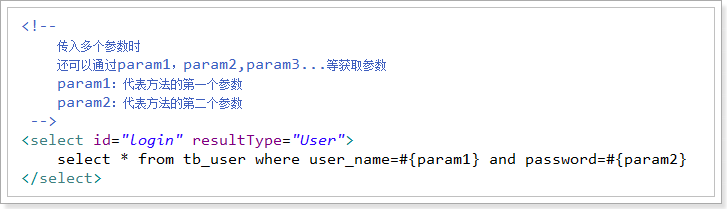
解决方案一：

在映射文件里，通过#{0}，#{1}获取参数



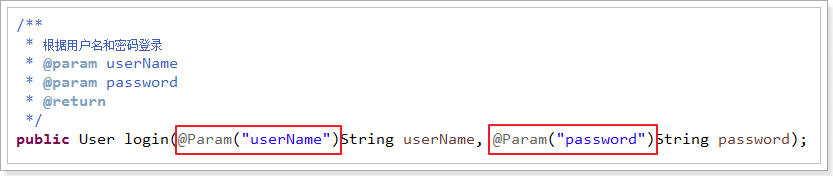
解决方案二：

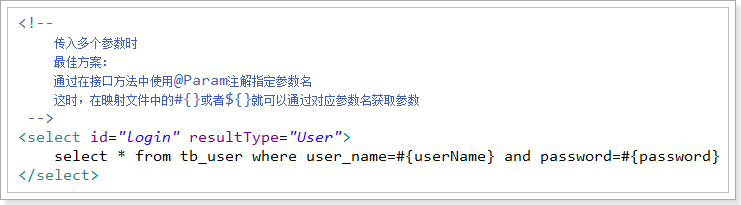
在映射文件里，通过param1，param2获取参数



最终解决方案：

在接口方法中的参数前，添加@Param注解指定参数名:





通常在方法的参数列表上加上一个注解@Param(“xxxx”) 表示参数的名字，然后通过${“xxxx”}或#{“xxxx”}获取参数

注意：

单个参数时，#{}与参数名无关的。

多个参数时，#{} ${}与参数名（@Param）有关。

什么时候需要加@Param注解？什么时候不加？

单个参数不加，多个参数加

终极解决方案：都加。

### HashMap

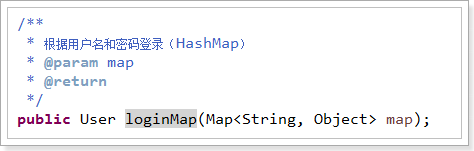
parameterType有三种类型的输入参数：

1. 基本数据类型
2. hashMap
3. pojo包装类

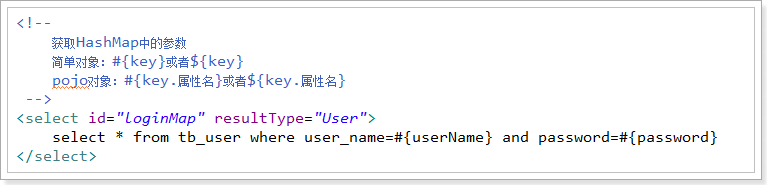
前面已经使用了基本数据类型和pojo类型的参数，那么hashMap这种类型的参数怎么传递参数呢？

其实，它的使用方式和pojo有点类似，简单类型通过#{key}或者${key}，复杂类型通过${key.属性名}或者#{key.属性名}

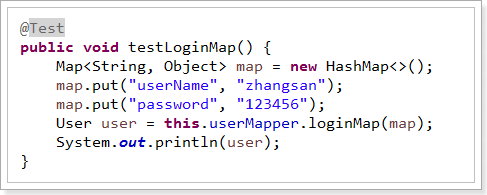
UserMapper接口方法：



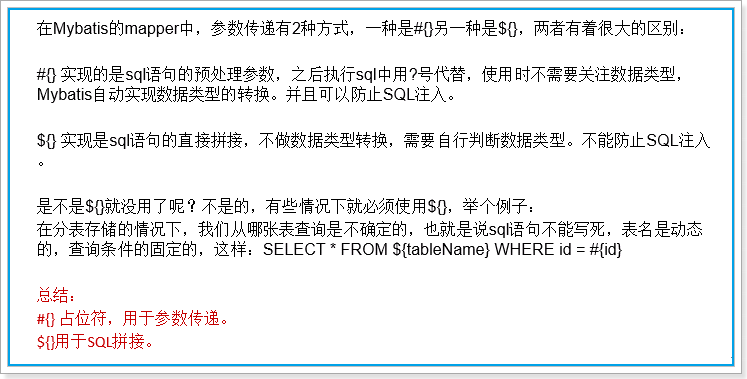
UserMapper配置文件：



测试用例：



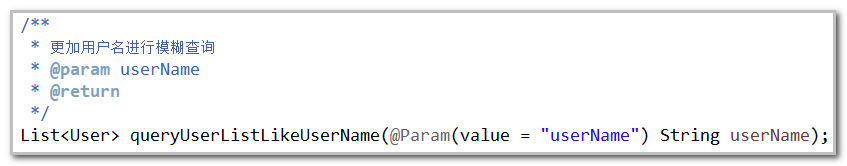
### 面试题（#、$区别）



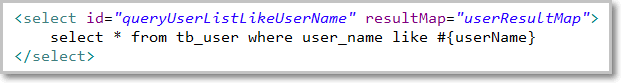
#### 区别案例一

练习：根据用户名进行模糊查询

UserMapper接口：



* 使用#{}查询：



编写测试方法：

@Test

**public** **void** testQueryUserListLikeUserName(){

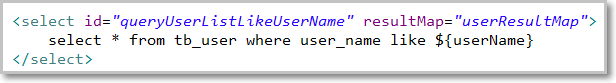
List<User> list = userMapper.queryUserListLikeUserName("%zhang%");

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

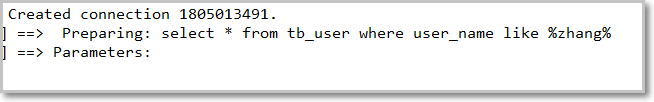
}

执行结果，都可以正常运行。

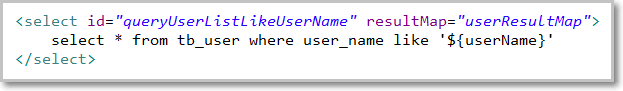
* 使用${}查询：



控制台语句的非预编译语句：是直接拼接的

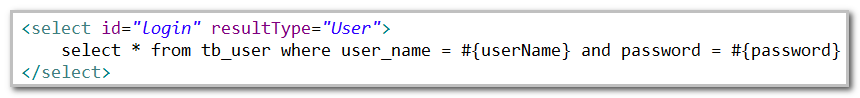


需要修改如下：



#### 区别案例二

再次执行之前的登陆方法：



执行测试：

@Test

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

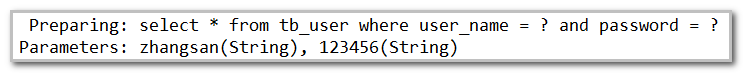
User user = userMapper.login("zhangsan", "123456");

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

}

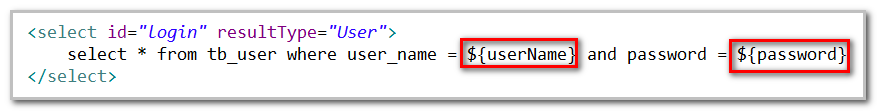
测试无误

执行查询测试后，控制台语句如下：



由于使用的是#{},在预编译时使用？进行了占位，而当将参数zhangsan和123456传递进去之后，会自动判断其参数类型为String类型，因此会在真正的sql语句中添加引号将两个参数包含在里面。

但是如果将#{}改成${}，如下:



再次执行测试会报错。

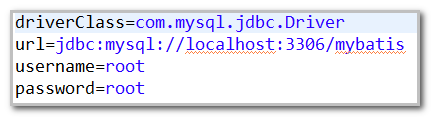
控制台语句如下：



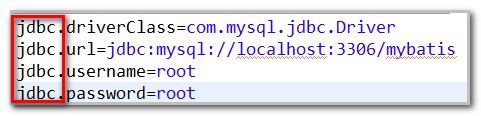
问题有两个：

1. 由于${}表示拼接，因此并不会自动判断参数类型，仅仅只是将参数直接拼接上去而已，少了引号。
2. Password参数的值明明是123456，却变成了root.

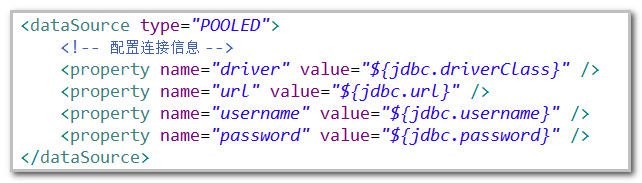
原因是在资源文件jdbc.properties中也有一个password=root，${}读取了资源文件中的password的值并将其作为参数值拼接到了sql中



问题2 的解决方案：在资源文件中的key前都添加一个前缀，让其不要重名即可。

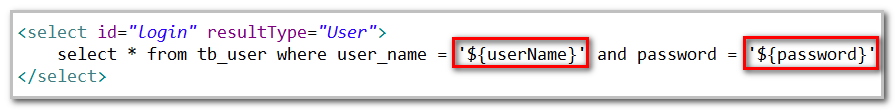


注意如果资源文件添加了前缀，那么在全局配置文件mybatis-config.xml中也需要将表达式进行相应的修改：



问题1的解决方案：

如果非要使用${}，那么就需要在statement的sql语句中手动添加引号，如下：



#### #{}和${}的区别总结

#{}：

1. 是预编译
2. 编译成占位符
3. 可以防止sql注入
4. 自动判断数据类型
5. 一个参数时，可以使用任意参数名称进行接收

${}:

1. 非预编译
2. sql的直接拼接
3. 不能防止sql注入
4. 需要判断数据类型，如果是字符串，需要手动添加引号。
5. 一个参数时，参数名称必须是value，才能接收参数。

## resultMap

ResultMap是mybatis中最重要最强大的元素，使用ResultMap可以解决两大问题：

* + POJO属性名和表结构字段名不一致的问题（有些情况下也不是标准的驼峰格式，比如id和userId）
  + 完成高级查询，比如说，一对一、一对多、多对多。(后面讲多表再讲)

### 解决列名（表中的字段名称）和实体类中的属性名不一致

查询数据的时候，查不到userName的信息，原因：数据库的字段名是user\_name，而POJO中的属性名字是userName

两端不一致，造成mybatis无法填充对应的字段信息。修改方法：在sql语句中使用别名

解决方案1：在sql语句中使用别名

解决方案2：参考驼峰匹配 --- mybatis-config.xml 的时候

<settings>

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

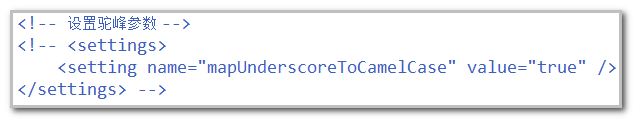
</settings>

解决方案3：resultMap自定义映射

示例：

#### 步骤一：将驼峰匹配注释掉

一旦注释掉驼峰匹配，那么再通过queryUserById查询的结果中，用户名就无法封装了，此时我们可以尝试使用ResultMap来解决这个问题。



#### 步骤二：在UserMapper.xml中配置resultMap

<!--

resultMap标签:可以自己配置对象属性和表字段的映射（不仅仅是驼峰规则的映射）

type属性：结果集的封装类型

id属性：唯一标识

autoMapping属性：如果不配置，默认为true。对其他属性进行自动映射。

-->

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

<!-- id标签：配置主键映射，column:表中的字段名称，property：属性名称 -->

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

<!-- 可以用来配置普通类型字段和属性映射（非复杂类型） -->

<result column=*"user\_name"* property=*"userName"*/>

</resultMap>

#### 步骤三：在statement中引用自定义resultMap

<!--

select:查询的statement,用来编写查询语句。

id:语句的唯一标识，使用动态代理之后，必须和mapper接口中的方法名称一致。

resultType：配置返回的结果集类型

parameterType:参数类型，可以省略。

-->

<!-- resultMap：引用自定义结果集的唯一标识 -->

<select id=*"queryUserById"* resultMap=*"userResultMap"*>

select \* from tb\_user where id = #{id}

</select>

测试无误

注意：测试完记得将驼峰命名的配置重新开启，因为其他的测试方法还要用。

### resultMap的自动映射

在上面讲到的resultMap中，主键需要通过id子标签配置，表字段和属性名不一致的普通字段需要通过result子标签配置。

那么，字段和属性名称都匹配的字段要不要配置？

这个取决于resultMap中的autoMapping属性的值：

为true时：resultMap中的没有配置的字段会自动对应。如果不配置，则默认为true。

为false时：只针对resultMap中已经配置的字段作映射。

## sql片段

在java代码中，为了提高代码的重用性，针对一些出现频率较高的代码，抽离出来一个共同的方法或者类

那么针对一些重复出现的sql片段，mybatis有没有一个比较好的解决方案呢？

Mybatis当然想到了这一点，它就是sql标签。

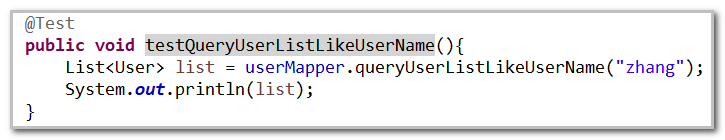
### 用法一

sql标签可以定义一个sql片段，在需要使用该sql片段的地方，通过include标签来使用。

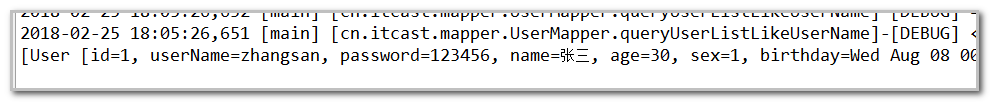
在有的时候为了提交sql语句的性能，就不能使用select \* 来进行查询，需要把需要显示的每一个字段都声明出来，如果这是查询语句笔记多的情况下，每次都手动编写比较麻烦，也容易写错，那么此时就需要定义一个片段来引用，这样就比较方便了。



测试：



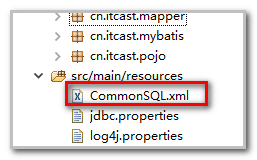
测试结果：



### 用法二

很多时候同一个sql片段，可能在多个映射文件中都有使用，如果每一个映射文件都编写一个相同的sql就比较麻烦，因此可以将通用的sql片段都定义在一个专门存放sql片段的映射文件中，然后有其他映射文件引用它即可。

如下，在resource目录下新增CommonSQL.xml文件：



内容如下：

<?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=*"CommonSQL"*>

<sql id=*"commonSql"*>

id,

user\_name,

password,

name,

age,

sex,

birthday,

created,

updated

</sql>

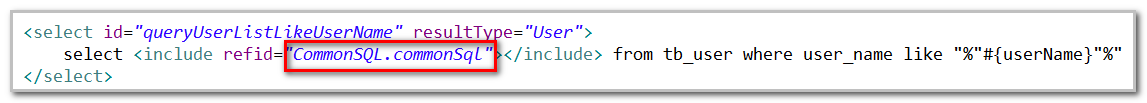
</mapper>

定义好sql片段的映射文件之后，接下来就该使用它了，首先应该把该映射文件引入到mybatis的全局配置文件中（mybatis-config.xml）：



最后在需要使用该sql片段的地方通过include标签的refId属性引用该sql片段：<include refId=”名称空间.sql片段的id” />

在UserMapper.xml的映射文件中，进一步改造根据用户名查询用户信息



## 总结//

Crud标签的使用：

Insert标签: 可以设置主键回显

userGeneratedKeys：开启主键回显的功能

keyColunm: 在数据库中主键对应的字段名称

keyproperty:在对象中主键对应的属性名称

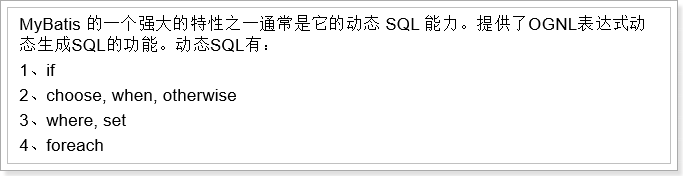
${}和#{}的5个区别：

1. 预编译和非预编译
2. 编译成占位符，直接拼接
3. 防sql注入和不防的区别
4. 自动判断参数类型，一个需要自行判断
5. 一个参数时#{}可以使用任意名称接收参数，${}只能使用value

resultMap: 可以手动配置表和对象之间的映射关系

sql片段： 可以抽取重复使用的sql片段，如果单独设置了一个映射文件，那么该映射文件需要被mybatis-config.xml管理，并且引用方式：namespace.id

# 动态sql



**MyBatis常用OGNL表达式**

1. e1 or e2
2. e1 and e2
3. e1 == e2,e1 eq e2
4. e1 != e2,e1 neq e2
5. e1 lt e2：小于
6. e1 lte e2：小于等于，其他gt（大于）,gte（大于等于）
7. e1 in e2
8. e1 not in e2
9. e1 + e2,e1 \* e2,e1/e2,e1 - e2,e1%e2
10. !e,not e：非，求反
11. e.method(args)调用对象方法
12. e.property对象属性值
13. e1[ e2 ]按索引取值，List,数组和Map
14. @class@method(args)调用类的静态方法
15. @class@field调用类的静态字段值

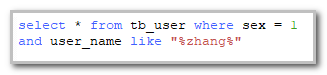
## if

* if标签：判断语句，用于进行逻辑判断的。

test属性：用来编写表达式，支持ognl.

案例：查询男性用户，如果输入了用户名，按用户名模糊查询,如果没有输入用户名，就查询所有男性用户。

正常的sql语句：查询男性并且用户名中包含zhang



但是要求时还要判断用户是否输入用户名来做不同的查询要求，而这里似乎没有办法判断是否输入了用户名，因此可以考虑使用动态sql来完成这个功能。

### 定义接口

在UserMapper接口中定义方法：

/\*\*

\* 查询男性用户，如果输入了用户名，按用户名模糊查询

\* **@param** userName

\* **@return**

\*/

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

### 编写mapper.xml

在UserMapper映射文件中，定义接口方法对应的Statement

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

select \* from tb\_user where sex = 1

<!--

if标签：用来进行判断

test属性：编写ognl表达式

-->

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

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

</if>

</select>

### ※测试

当用户名为zhang时

@Test

**public** **void** testQueryUsersLikeUserName(){

List<User> list = userMapper.queryUsersLikeUserName("zhang");

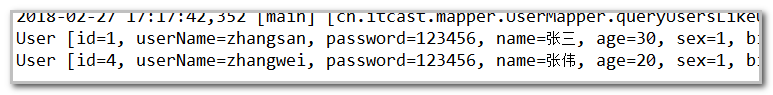
**for** (User user : list) {

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

}

}

查询结果：只查名字中带zhang的男性



## choose when otherwise

* choose标签：条件选择
* when子标签：编写条件，不管有多少个when条件，一旦其中一个条件成立，后面的when条件都不执行。

test属性：编写ognl表达式

* otherwise子标签：当所有条件都不满足时，才会执行该条件。

需求：

编写一个查询方法，设置两个参数，一个是用户名，一个是年龄。

根据用户名或者年龄查询所有男性用户，如果输入了用户名则按照用户名模糊查找，否则就按照年龄查找，两个条件只能成立一个，如果都不输入就查找用户名为“zhangsan”的用户。

### 定义接口

在UserMapper接口中，定义接口方法：

/\*\*

\* 根据用户名或者年龄查找男性用户，如果有用户名就按用户名查找，否则按年龄查找，如果都没有，就查询名字是zhangsan的男性用户

\* **@param** userName

\* **@param** age

\* **@return**

\*/

List<User> queryUserListByUserNameOrAge(@Param("userName")String userName,@Param("age")Integer age);

### 编写mapper.xml

在UserMapper.xml中，定义接口方法对应的Statement

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

select \* from tb\_user where sex = 1

<!--

choose标签：条件选择

when标签:编写条件，不管有多少个when条件，一旦其中一个条件成立，后面的when条件都不执行。

test属性：编写ognl表达式

otherwise标签：当所有条件都不满足时，才会执行该条件。

-->

<choose>

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

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

</when>

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

and age &lt; #{age}

</when>

<otherwise>

and user\_name = 'zhangsan'

</otherwise>

</choose>

</select>

注意：不能使用<，因为该符号在xml中表示一个标签的开始符号，因此会报错，需要进行转义。

可以转义为：

&lt;

也可以把条件反过来写：>不是标签的开始，因此不会做检查。

and #{age} > age

小于 &lt;

大于 &gt;

### 测试

当既有用户名又有年龄时

@Test

**public** **void** testQueryUserListByUserNameOrAge(){

List<User> list = userMapper.queryUserListByUserNameOrAge("zhang", 16);

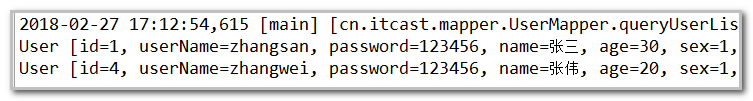
**for** (User user : list) {

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

}

}

测试结果：只按第一个条件进行查询，既只按用户名进行模糊查询。



## Where

* where标签：可以自动将动态sql中多出来的一个and或者or去除。

案例：查询所有用户，如果输入了用户名按照用户名进行模糊查询，如果输入年龄，按年龄进行查询，如果两者都输入，两个条件都要成立。

### 定义接口

在UserMapper接口中，定义接口方法：

/\*\*

\* 根据用户名和年龄查询用户，如果有用户名按用户名查，如果有年龄按年龄差，如果都有，两个条件都要成立。

\* **@param** userName

\* **@param** age

\* **@return**

\*/

List<User> queryUserListByUserNameAndAge(@Param("userName")String userName,@Param("age")Integer age);

### 编写mapper.xml

在UserMapper.xml中，定义接口方法对应的Statement.

可以单独使用if标签完成：但是需要在语句中where 1 = 1

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

select \* from tb\_user where 1 = 1

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

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

</if>

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

and age = #{age}

</if>

</select>

可以使用where标签进行优化：可以将where 1 = 1优化掉

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

select \* from tb\_user

<!--

where标签的作用：

1、自动添加where关键字

2、自动去除动态sql前面多余的一个and或者or

-->

<where>

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

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

</if>

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

and age = #{age}

</if>

</where>

</select>

### 测试

两个条件都成立

在UserMapperTest测试类中，添加测试用例

@Test

**public** **void** testQueryUserListByUserNameAndAge(){

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

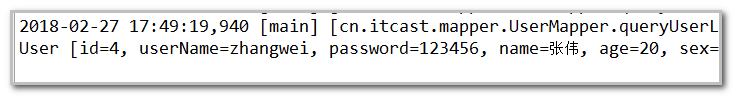
**for** (User user : list) {

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

}

}

测试结果：查询到名字中有zhang并且年龄是20的记录



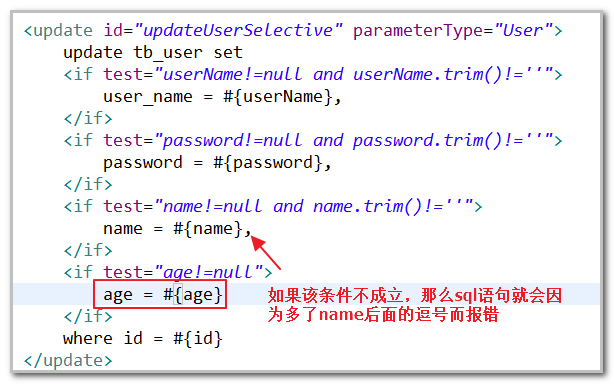
## Set

* set标签：

可以自动添加一个set关键字，并且会将动态sql最后多余的逗号去除。

案例：修改用户信息，如果参数user中的某个属性为null，则不修改。

如果在正常编写更新语句时，如下：



那么一旦在传递的参数中没有age，此时生成的sql机构就会因为多了一个逗号而报错。

### 定义接口

在UserMapper接口中，定义接口方法：

/\*\*

\* 修改用户信息，如果参数user中的某个属性为null，则不修改。

\* **@param** user

\*/

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

### 编写mapper.xml

在UserMapper.xml中，定义接口方法对应的Statement

<update id=*"updateUserSelective"* parameterType=*"User"*>

update tb\_user

<!--

set标签：可以将动态sql语句最后多余的逗号去除

-->

<set>

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

user\_name = #{userName},

</if>

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

password = #{password},

</if>

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

name = #{name},

</if>

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

age = #{age}

</if>

</set>

where id = #{id}

</update>

### 测试

在UserMapperTest测试类中，添加测试用例

@Test

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

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

user.setId(21l);

user.setUserName("张三丰");

user.setPassword("4321");

user.setName("zhangsanfeng");

//不设置age

userMapper.updateUserSelective(user);

}

注意：千万别写错成userMapper.updateUser(user);这个是之前的方法，别习惯性的调用错了。

## foreach

* foreach标签：遍历集合或者数组
* collection属性：接收的集合或者数组参数
* item属性：集合或者数组参数中的每一个元素
* separator属性：标签分隔符
* open属性：以什么开始
* close属性：以什么结束

案例：按照多个id查询用户信息

### 接口

在UserMapper接口中，定义接口方法：

/\*\*

\* 根据多个id查询用户信息

\* **@param** ids

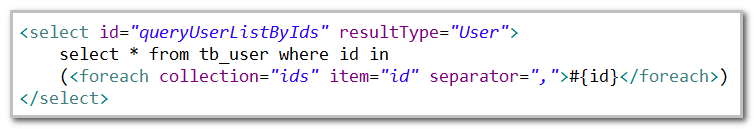
\* **@return**

\*/

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

### 配置

在UserMapper.xml中，定义接口方法对应的Statement



如果是这样子配置其实也行，不过总感觉两个()不太舒服

因此可以改造如下:

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

select \* from tb\_user where id in

<!--

foreach标签：遍历集合

collection属性：接收的集合参数

item属性：集合参数中的每一个元素

separator属性：标签逗号分隔符

open：以什么开始

close：以什么结束

-->

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

#{id}

</foreach>

</select>

### 测试

在UserMapperTest测试类中，添加测试用例

@Test

**public** **void** testQueryUserListByIds(){

List<User> users = userMapper.queryUserListByIds(**new** Long[]{1l,2l,3l});

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

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

}

}

## 总结//

If标签：条件判断

test属性：编写ognl表达式

choose标签：条件选择

when子标签：用于条件选择，一旦有一个when成立，后续不再执行。

otherwise子标签：所有条件不成立，执行该条件。

where标签：用于sql动态条件拼接，添加where关键字，可以将动态sql多余的第一个and或者or去除。

set标签: 用于更新语句的拼接，添加set关键字，并可以将动态sql中多余的逗号去除

foreach标签：用于遍历参数中的集合

collection属性：参数中的集合

item属性：表示集合中的某个元素

separator属性：分隔符

open:以什么开始

close:以什么结束

# 缓存（了解）

执行相同的sql语句和参数，mybatis不再重新执行sql，而是从缓存中命中返回。

## 一级缓存

### 测试一级缓存

在mybatis中，一级缓存默认是开启的（不管你用或不用，她一直都在），作用域：在同一个sqlSession下

在UserMapperTest中添加测试一级缓存的方法：

@Test

**public** **void** testfirstCache(){

User user1 = **this**.userMapper.queryUserById(1l);

System.***out***.println(user1);

System.***out***.println("=================第二次查询==================");

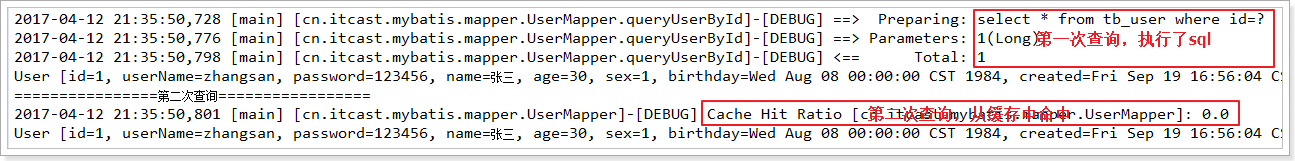
User user2 = **this**.userMapper.queryUserById(1l);

System.***out***.println(user2);

}

由于一级缓存的存在，此时在log日志中，应该只会在第一次查询是执行sql语句，第二次查询时直接从缓存中命中，即不再执行sql语句。

结果：



### 清除一级缓存

使用：sqlSession.clearCache();可以强制清除缓存

步骤一：修改加载配置文件的方法

**private** UserMapper userMapper;

**private** SqlSession sqlSession;

@Before

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

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

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream,"test");

sqlSession = sqlSessionFactory.openSession(**true**);//自动提交事务

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

}

步骤二：编写测试方法

@Test

**public** **void** testfirstCache(){

User user1 = userMapper.queryUserById(1l);

System.***out***.println(user1);

sqlSession.clearCache();//清空一级缓存

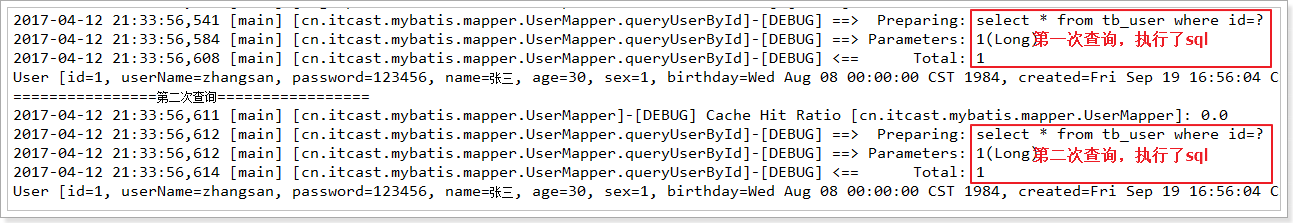
System.***out***.println("=================第二次查询==================");

User user2 = userMapper.queryUserById(1l);

System.***out***.println(user2);

}

在执行第二次查询之前清空缓存，再去执行查询。这时无法从缓存中命中，便会去执行sql从数据库中查询。



### ※一级缓存的问题

在执行update、insert、delete的时候,即使操作的不是和一级缓存中的是同一条记录，都会清空一级缓存。

在测试方法中添加更新的操作：

@Test

**public** **void** testfisrtCache2(){

User user1 = userMapper.queryUserById(1l);

System.***out***.println(user1);

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

user.setId(21l);//即使修改的其他的记录，也会将一级缓存中的数据进行清空。

user.setUserName("张无忌");

user.setPassword("4321");

user.setName("zhangwuji");

//不设置age

userMapper.updateUserSelective(user);

System.***out***.println("======================================");

User user2 = userMapper.queryUserById(1l);

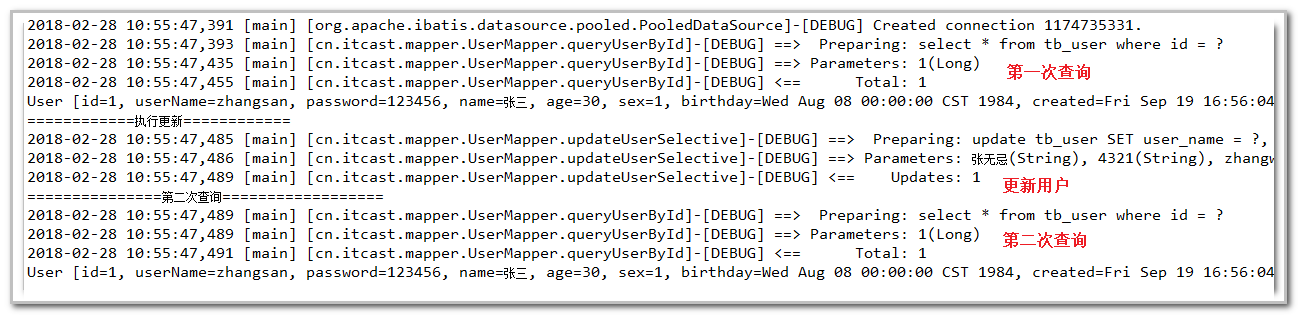
System.***out***.println(user2);

}

由于insert、update、delete会清空缓存，所以第二次查询时，依然会输出sql语句，即从数据库中查询。

当我们在一个网站浏览商品的时候，一旦对一个商品进行下单，那么所有的一级缓存就都被被清空。

因此mybatis自带的一级缓存其实并没有什么作用。



## 二级缓存(了解)

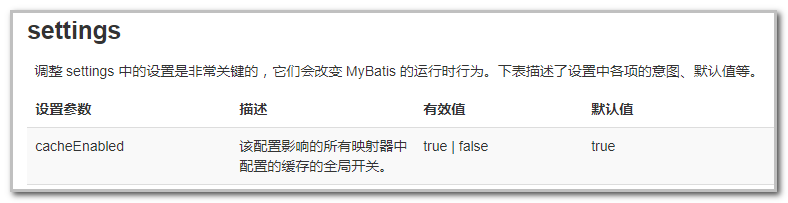
mybatis 的二级缓存的作用域：mapper范围的（即映射文件级别的，多个sqlSession可以共享二级缓存数据）

1. 同一个mapper的namespace，在同一个namespace中查询sql可以从缓存中命中。
2. 跨sqlSession，不同的SqlSession可以从二级缓存中命中

场景：在用户通过id查询一个商品信息，执行完查询方法之后会将sqlSession释放掉，当其他再次发送请求查询同一个商品信息时会再次获取一个sqlSession来执行查询，那么此时如果配置了二级缓存的话，就可以直接从缓存中获取信息，而不用再次去数据库查询了。

怎么开启二级缓存：

1. 在映射文件中，添加<cache />标签
2. 在全局配置文件中，设置cacheEnabled参数，默认已开启。



注意：

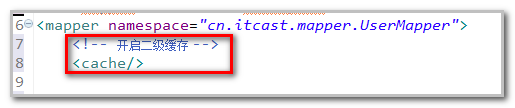
由于缓存数据是在sqlSession调用close方法时，放入二级缓存的，因此在测试二级缓存时必须先将第一个sqlSession关闭

二级缓存的对象必须序列化，例如：User对象必须实现Serializable接口。

### 测试二级缓存

步骤一：在映射文件中开启二级缓存

开启二级缓存，在映射文件(UserMapper.xml)中添加<cache />：



步骤二：修改setUp方法

为了获取第二个SqlSession方便，将SqlSessionFactory定义到成员变量位置。

**private** UserMapper userMapper;

**private** SqlSession sqlSession;

**private** SqlSessionFactory sqlSessionFactory;

@Before

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

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

sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream,"test");

sqlSession = sqlSessionFactory.openSession(**true**);//自动提交事务

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

}

步骤三：编写测试方法

@Test

**public** **void** testSecondCache(){

User user1 = userMapper.queryUserById(21l);

System.***out***.println(user1);

System.***out***.println("===============跨sqlSession查询================");

//获取第二个sqlSession

SqlSession sqlSession2 = sqlSessionFactory.openSession(**true**);

//通过第二个sqlSession中的接口执行查询

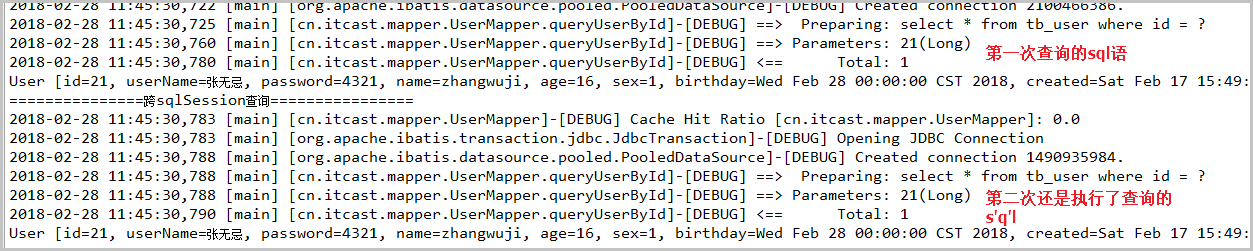
UserMapper userMapper2 = sqlSession2.getMapper(UserMapper.**class**);

User user2 = userMapper2.queryUserById(21l);

System.***out***.println(user2);

}

测试结果如下：发现二级缓存没有起作用，第二次查询还是发送了sql



原因：二级缓存只有在sqlSession被close()的时候才会将数据存入二级缓存中，因此我们需要手动关闭第一个sqlSession.

步骤四：手动关闭第一个sqlSession

@Test

**public** **void** testSecondCache(){

User user1 = userMapper.queryUserById(21l);

System.***out***.println(user1);

sqlSession.close();//关闭sqlSession，此时会将数据存入二级缓存中。

System.***out***.println("===============跨sqlSession查询================");

//获取第二个sqlSession

SqlSession sqlSession2 = sqlSessionFactory.openSession(**true**);

//通过第二个sqlSession中的接口执行查询

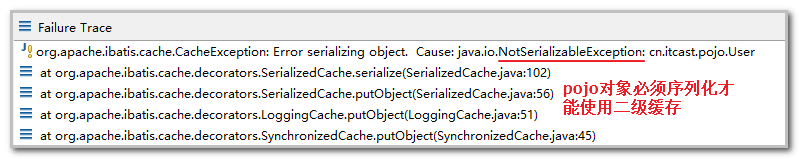
UserMapper userMapper2 = sqlSession2.getMapper(UserMapper.**class**);

User user2 = userMapper2.queryUserById(21l);

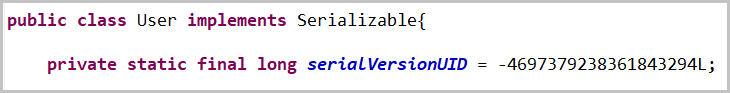
System.***out***.println(user2);

}

再次执行测试，发现又报错如下：原因是使用二级缓存要求对象必须序列化



将对象设置序列化：



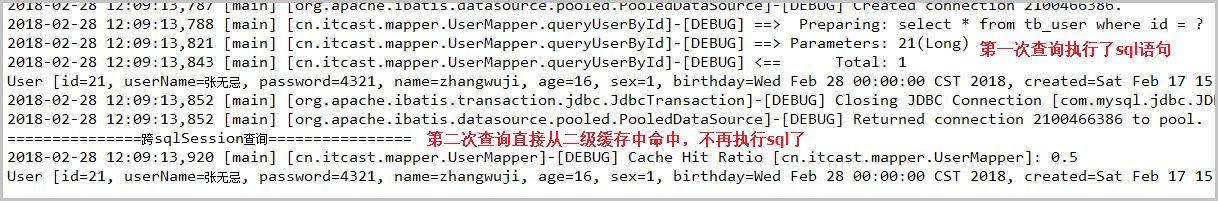
serialVersionUID的作用：

相当于设置了一个版本，.class文件中记录一个版本号，序列化出来的文件中有一个版本号，当修改对象中某个属性的时候，会检查两个版本号是否一直。

如果设置了该属性，那么一旦在修改对象中的某个属性后，版本号发生改变，一旦检查出版本号不一致，就不能再次序列化了。

如果不设置每次修改数据都可以序列化。

给User对象实现序列化接口后，重新运行测试用例，日志：



### 二级缓存的问题

二级缓存在执行update、insert、delete的时候，也同样会清空二级缓存中的内容。

@Test

**public** **void** testSecondCache(){

User user1 = userMapper.queryUserById(21l);

System.***out***.println(user1);

sqlSession.close();//关闭sqlSession，此时会将数据存入二级缓存中。

//获取第二个sqlSession

SqlSession sqlSession2 = sqlSessionFactory.openSession(**true**);

//通过第二个sqlSession中的接口执行查询

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

//在第二个sqlSession中执行更新

System.***out***.println("============执行更新============");

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

user.setId(21l);//即使修改的其他的记录，也会将一级缓存中的数据进行清空。

user.setUserName("张翠山");

user.setPassword("4321");

user.setName("zhang翠山");

//不设置age

userMapper2.updateUserSelective(user);

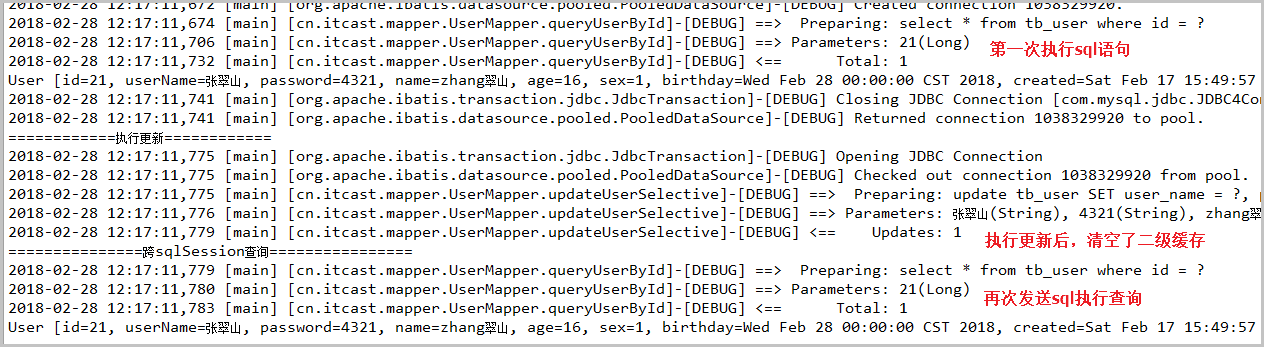
System.***out***.println("===============跨sqlSession查询================");

User user2 = userMapper2.queryUserById(21l);

System.***out***.println(user2);

}

测试结果：发现一旦进行了增删改操作，二级缓存也被清空了。



### 关闭二级缓存

如何关闭二级缓存：

1. 不要在映射文件中开启就行
2. 设置全局的全局配置文件mybatis-config.xml 中去关闭二级缓存

在mybatis-config.xml配置中：

<settings>

<!-- 行为参数，name:参数名，value：参数值，默认为false，true：开启驼峰匹配，即从经典的数据库列名到经典的java属性名 -->

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

<!-- 关闭二级缓存，默认是开启，false：关闭 -->

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

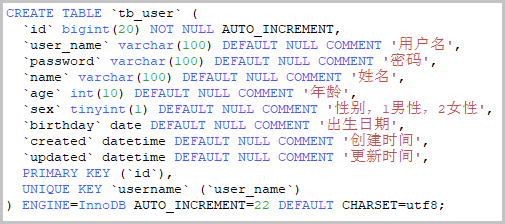
</settings>

# 高级查询(OrderMapper.xml)

## 表介绍

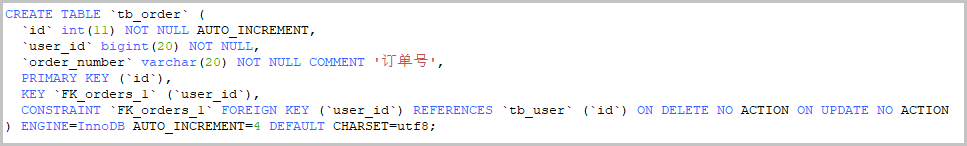
tb\_user:用户表

该表用于存放用户信息



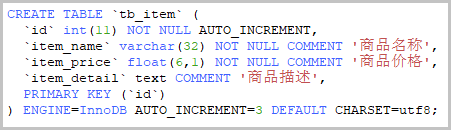
tb\_order:订单表

用于存放订单编号和用户id



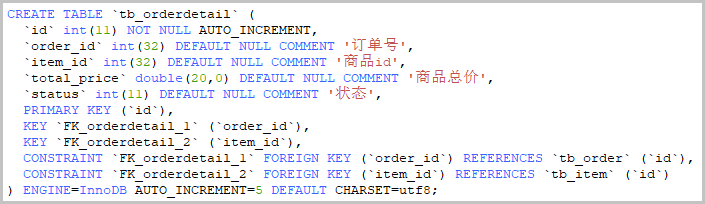
tb\_item:商品表

该表用于存放商品信息

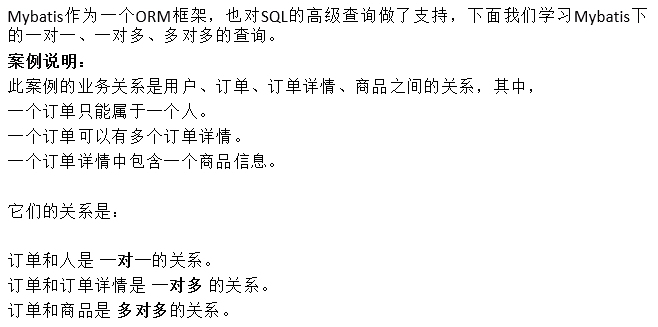


tb\_orderdetail：订单详情表

用于存放订单的详细信息，如订单编号、商品编号、商品价格等。



## 表关系说明

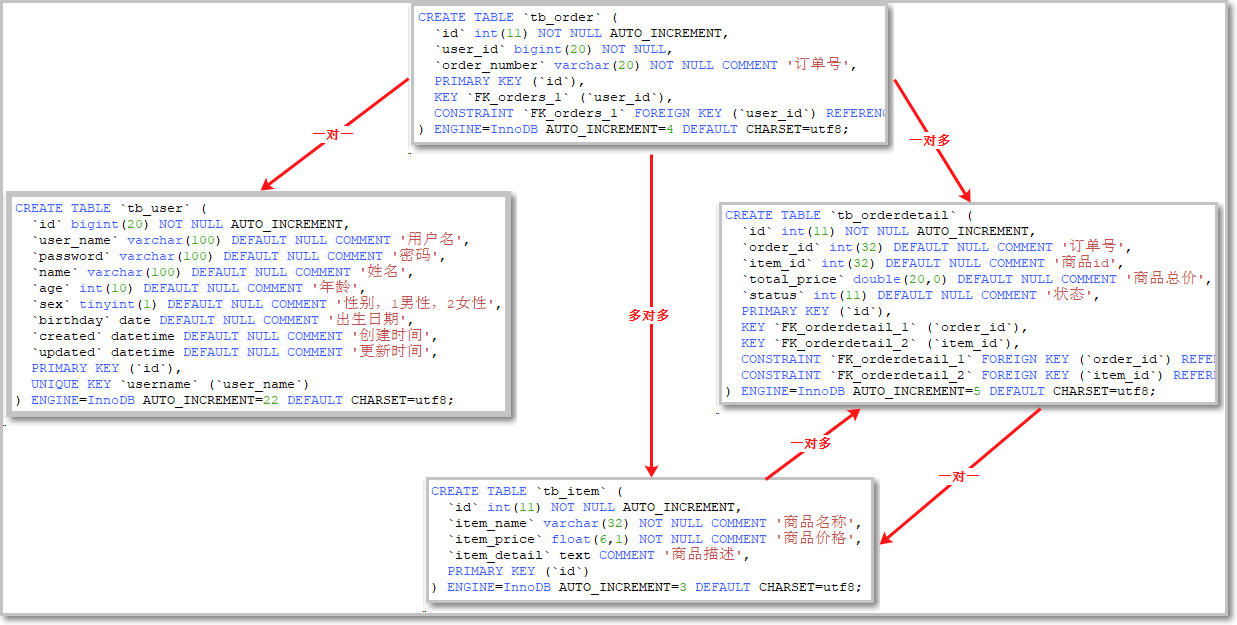


站在订单的角度，一个订单只有一个用户（一对一）。

一个订单中有两个订单详情（一对多），一个订单详情只有一个商品（一对一），一个订单可以有多个商品，一个商品可以有多个订单（多对多）



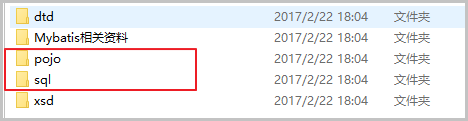
表关系:

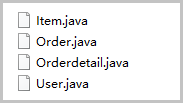


## 数据准备

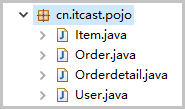
将资料中的的其他pojo对象复制到工程中即可。

Pojo参考课前资料：





目录结构：

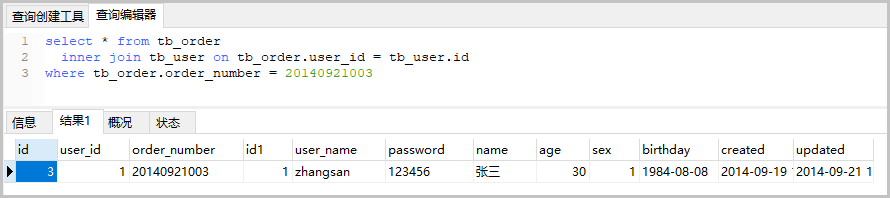


## 一对一查询

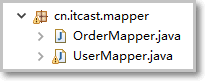
### 添加User属性方式

需求：通过订单编号20140921003查询出订单信息，并查询出下单人信息。

查询语句以及查询结果：



#### 步骤一：编写接口



接口如下：

**public** **interface** OrderMapper {

Order queryOrderAndUserByOrderNumber(@Param("number") String number);

}

#### 步骤二：编写映射文件

映射文件如下：

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

<select id=*"queryOrderAndUserByOrderNumber"* resultType=*"Order"*>

select \* from tb\_order o inner join tb\_user u on o.user\_id = u.id where o.order\_number = #{number}

</select>

</mapper>

#### 步骤三：全局配置文件管理映射文件

在mybatis-config.xml中添加代码如下:

<mapper resource=*"OrderMapper.xml"*/>

#### 步骤四：编写测试类

**public** **class** OrderMapperTest {

**private** OrderMapper orderMapper;

@Before

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

//加载全局配置文件

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

//获取SqlSessionFactory

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

//获取SqlSession

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

//获取orderMapper

orderMapper = sqlSession.getMapper(OrderMapper.**class**);

}

@Test

**public** **void** testQueryOrderAndUserByOrderNumber(){

Order order = orderMapper.queryOrderAndUserByOrderNumber("20140921003");

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

}

}

执行后测试结果如下：只查询出Order对象，并没有User对象的数据。



原因:Order对象中并没有相关User对象的属性，当然就无法将user对象的数据显示出来。

解决方案：如果是通过在查询Order的时候查询User对象，可以在Order对象中添加一个User对象的属性。

#### 步骤五：添加user属性到Order对象中

**public** **class** Order {

**private** Integer id;

**private** Long userId;

**private** String orderNumber;

**private** User user;

**public** User getUser() {

**return** user;

}

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

**this**.user = user;

}

**public** Integer getId() {

**return** id;

}

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

**this**.id = id;

}

**public** Long getUserId() {

**return** userId;

}

**public** **void** setUserId(Long userId) {

**this**.userId = userId;

}

**public** String getOrderNumber() {

**return** orderNumber;

}

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

**this**.orderNumber = orderNumber;

}

@Override

**public** String toString() {

**return** "Order [id=" + id + ", userId=" + userId + ", orderNumber="

+ orderNumber + ", user=" + user + "]";

}

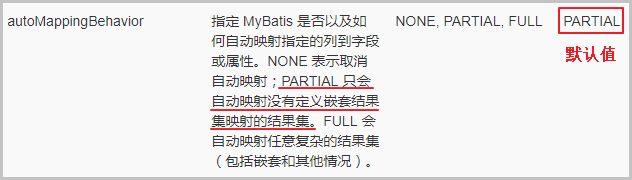
}

#### 步骤六：再次执行查询

查询结果：发现user对象中的数据还是没有封装到Order中



原因：在mybatis-config.xml中的setting配置中有如下属性：



在查询Order的时候把User查询出来，其实这就是一个嵌套查询。

而在使用嵌套查询的时候，那么普通结果集就无法满足我们的查询要求，因此要使用到ResultMap标签了

#### 步骤七：修改OrderMapper.xml

注意：一旦涉及到嵌套映射，一定要设置手动设置为自动映射。不管是Order的自动映射还是User的自动映射都需要手动设置为true。

<?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.OrderMapper"*>

<!--

使用resultMap设置结果集

type:返回的结果集类型

id：resultMap的唯一标识

autoMapping:添加自动映射，一旦涉及到嵌套查询，需要手动设置为true

-->

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

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

<!--

association：用于对一的映射

property：类中的关联属性的名称

javaType：属性对应的类型

autoMapping:添加自动映射，一旦涉及到嵌套查询，需要手动设置为true

-->

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

<!--

id:关联对象的主键

property:关联对象的主键名称

column:关联对象在表中对应的字段名称，如果重名，可以使用别名

-->

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

</association>

</resultMap>

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

select \*,u.id as uid from tb\_order o inner join tb\_user u on o.user\_id = u.id where o.order\_number = #{number}

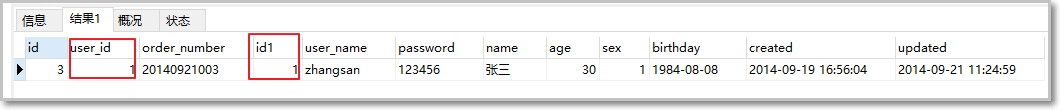
</select>

</mapper>

也可以不用设置别名，直接使用user\_id也行：不过还是建议使用别名

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

因为user\_id的值是和user表中的id的值是一样的，但千万不能使用id1，这个值其实本质上也是id，只不过如果显示id就和order中id重名了，因此navicat为了帮我们区分不同的id，将user表中的id显示成id1了而已。



### ※扩展Order类的实现方式（了解）

思路：

创建新的pojo对象OrderUser继承Order对象，这样OrderUser就从Order中继承了订单信息。

再把User对象中的属性copy到OrderUser对象中，这样OrderUser又有了User中的用户信息。

把sql语句的结果集封装成OrderUser对象，那么结果中就既包含了订单信息又包含了下单人信息。

问题：每次嵌套查询都需要编写一个新的类，太麻烦。不推荐

第一步：

新建OrderUser实体类继承Order，在把User中除Id以外的其他属性copy到OrderUser中，并添加get、set方法：

**public** **class** OrderUser **extends** Order{

// 用户名

**private** String userName;

// 密码

**private** String password;

// 姓名

**private** String name;

// 年龄

**private** Integer age;

// 性别，1男性，2女性

**private** Integer sex;

// 出生日期

**private** Date birthday;

// 创建时间

**private** Date created;

// 更新时间

**private** Date updated;

**public** String getUserName() {

**return** userName;

}

**public** **void** setUserName(String userName) {

**this**.userName = userName;

}

**public** String getPassword() {

**return** password;

}

**public** **void** setPassword(String password) {

**this**.password = password;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** Integer getAge() {

**return** age;

}

**public** **void** setAge(Integer age) {

**this**.age = age;

}

**public** Integer getSex() {

**return** sex;

}

**public** **void** setSex(Integer sex) {

**this**.sex = sex;

}

**public** Date getBirthday() {

**return** birthday;

}

**public** **void** setBirthday(Date birthday) {

**this**.birthday = birthday;

}

**public** Date getCreated() {

**return** created;

}

**public** **void** setCreated(Date created) {

**this**.created = created;

}

**public** Date getUpdated() {

**return** updated;

}

**public** **void** setUpdated(Date updated) {

**this**.updated = updated;

}

@Override

**public** String toString() {

**return** "OrderUser [userName=" + userName + ", password=" + password

+ ", name=" + name + ", age=" + age + ", sex=" + sex

+ ", birthday=" + birthday + ", created=" + created

+ ", updated=" + updated + "]";

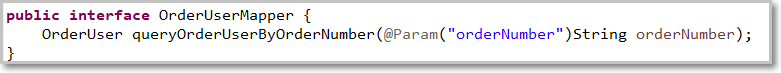
}

}

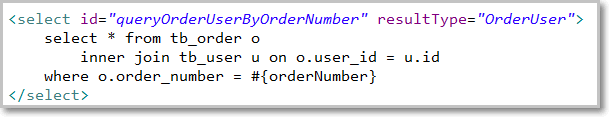
第二步：

编写OrderUserMapper的接口和OrderUserMapper.xml的映射文件：

在OrderUserMapper接口中，添加查询方法(复制即可)：



在OrderUserMapper.xml中，配置对应的Statement，并把结果集封装成OrderUser对象：



第三步：

在全局配置文件mybatis-config.xml中配置关联OrderUserMapper.xml的映射文件：

<mapper resource=*"OrderUserMapper.xml"*/>

第四步：

**public** **class** OrderUserMapperTest {

**private** OrderUserMapper orderUserMapper;

@Before

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

String resource = "mybatis-config.xml";

InputStream inputStream = Resources.*getResourceAsStream*(resource);

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

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

orderUserMapper = sqlSession.getMapper(OrderUserMapper.**class**);

}

@Test

**public** **void** testQueryOrderUserByOrderNumber() {

OrderUser orderUser = orderUserMapper.queryOrderUserByOrderNumber("20140921003");

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

}

}

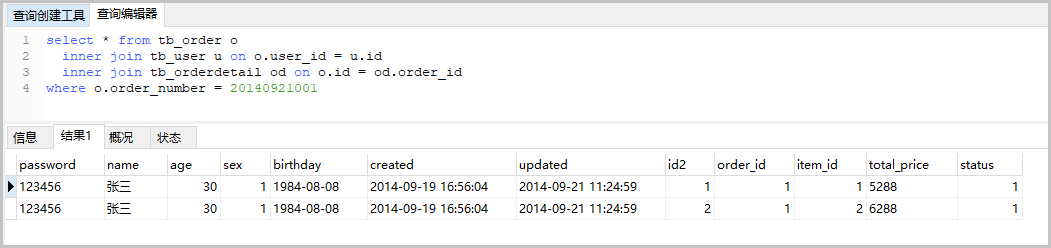
## 一对多查询

**一对多查询：**通过订单编号20140921001查询订单，并查询出下单人信息以及查询出订单详情。

思路：

订单 : 订单详情 = 1 : n（体现在pojo对象中，就是在Order对象中添加OrderDetail对象的集合）

SQL：查询出两条记录，除了item\_id和total\_price不同外，其他都一样（截图的原因，无法截全，因此只截图后边不一样的部分）。



### 步骤一：修改Order

在Order类添加List<OrderDetial>属性，并添加get、set方法和toString方法：

**public** **class** Order {

**private** Integer id;

**private** Long userId;

**private** String orderNumber;

**private** User user;//一对一

**private** List<Orderdetail> detailList;

**public** List<Orderdetail> getDetailList() {

**return** detailList;

}

**public** **void** setDetailList(List<Orderdetail> detailList) {

**this**.detailList = detailList;

}

**public** User getUser() {

**return** user;

}

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

**this**.user = user;

}

**public** Integer getId() {

**return** id;

}

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

**this**.id = id;

}

**public** Long getUserId() {

**return** userId;

}

**public** **void** setUserId(Long userId) {

**this**.userId = userId;

}

**public** String getOrderNumber() {

**return** orderNumber;

}

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

**this**.orderNumber = orderNumber;

}

@Override

**public** String toString() {

**return** "Order [id=" + id + ", userId=" + userId + ", orderNumber="

+ orderNumber + ", user=" + user + ", detailList=" + detailList

+ "]";

}

}

### 步骤二：编写接口方法

编写接口方法：

**public** **interface** OrderMapper {

Order queryOrderAndUserByOrderNumber(@Param("number") String number);

Order queryOrderAndUserAndOrderdetailByOrderNumber(@Param("number") String number);

}

### 步骤三：编写statement

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

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

<!-- 一对一映射 -->

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

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

</association>

<!-- 订单和订单详情的一对多映射

collection标签:配置对多的集合映射

property：Order中多方的属性名称

javaType:集合类型

ofType:集合中的对象类型

autoMapping:自动映射

-->

<collection property=*"detailList"* javaType=*"List"* ofType=*"Orderdetail"* autoMapping=*"true"*>

<!--

property:类中的对应主键的属性名称

column：表中的字段名称，可以使用自己在sql语句中设置的别名，注意不要使用结果中有mysql自动设置的别名，如id2

-->

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

</collection>

</resultMap>

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

select \*,od.id as detail\_id,u.id as uid from tb\_order o

inner join tb\_user u on o.user\_id = u.id

inner join tb\_orderdetail od on o.id = od.order\_id

where o.order\_number = #{number}

</select>

### 步骤四：测试

@Test

**public** **void** testQueryOrderAndUserAndOrderdetailByOrderNumber(){

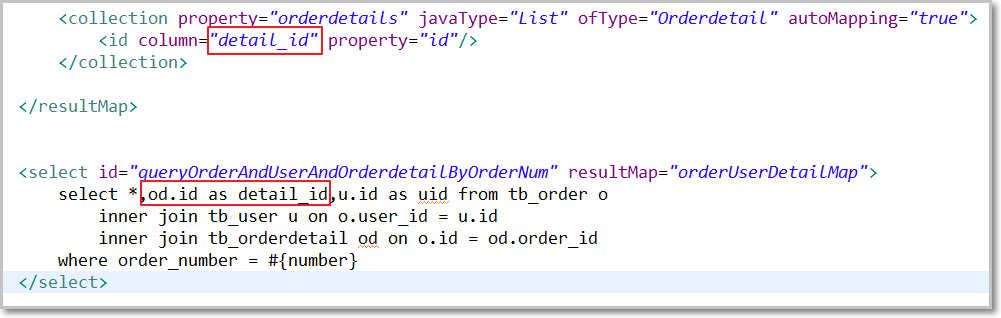
Order order = orderMapper.queryOrderAndUserAndOrderdetailByOrderNumber("20140921001");

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

}

### 注意问题

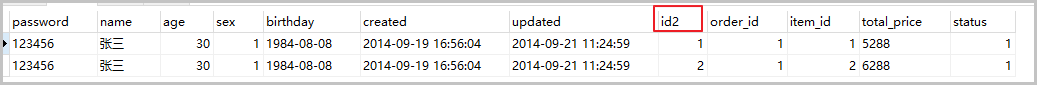
注意在使用别名查询来映射主键的时候，一定要手动在sql语句中自行定义别名。



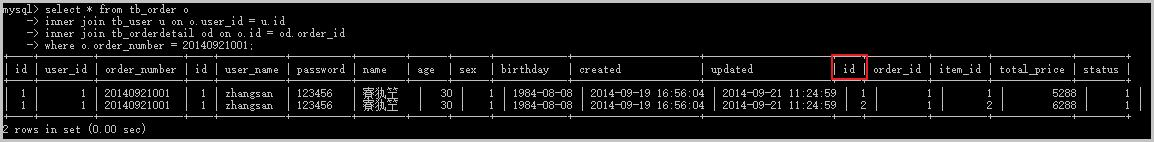
不要使用数据库可视化工具生成的别名，会封装错误。

比如在查询的时候，由于有多个id同名，因此navicate在显示结果的时候自动设置了id2作为Orderdetail的id主键别名。

但是这个id2本质上还是id,只是为了看起来清晰才显示了id2。

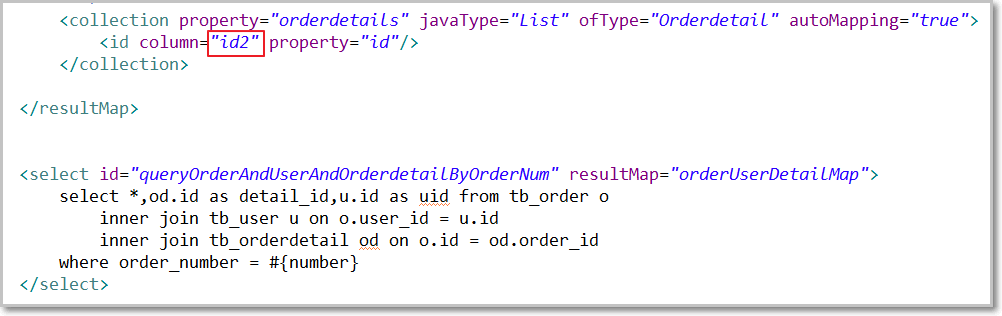


如果使用mysql进行查询，那么其显示结果还是id:



如果在statement的集合映射中设置的是这个id2的话就可能在注入id的时候将数据封装错误。

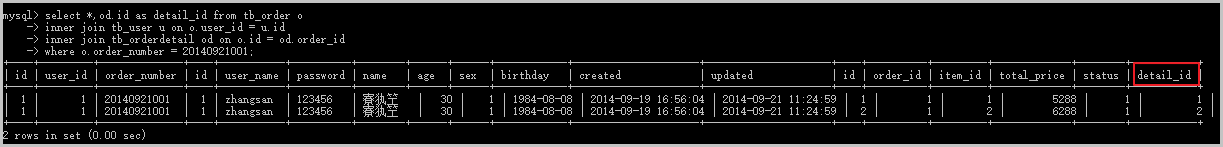
如果结果集配置为id2:



那么控制台打印结果时发现，Order对象中的关联属性detailList中的两条数据，id重复了。



而如果使用别名查询，那么查询结果中是确实有别名列存在的：因此不会发生封装错误数据的问题。



## 多对多查询

**多对多查询：**通过订单号20140921001查询订单，查询出下单人信息并且查询出订单详情以及商品数据。

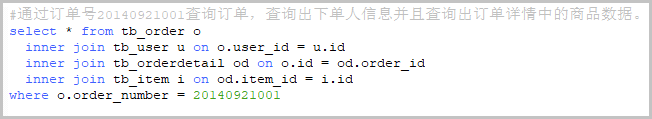
思路：

订单：订单详情 = 1 : n（体现在pojo对象中就是在Order对象中添加OrderDetail对象的集合）

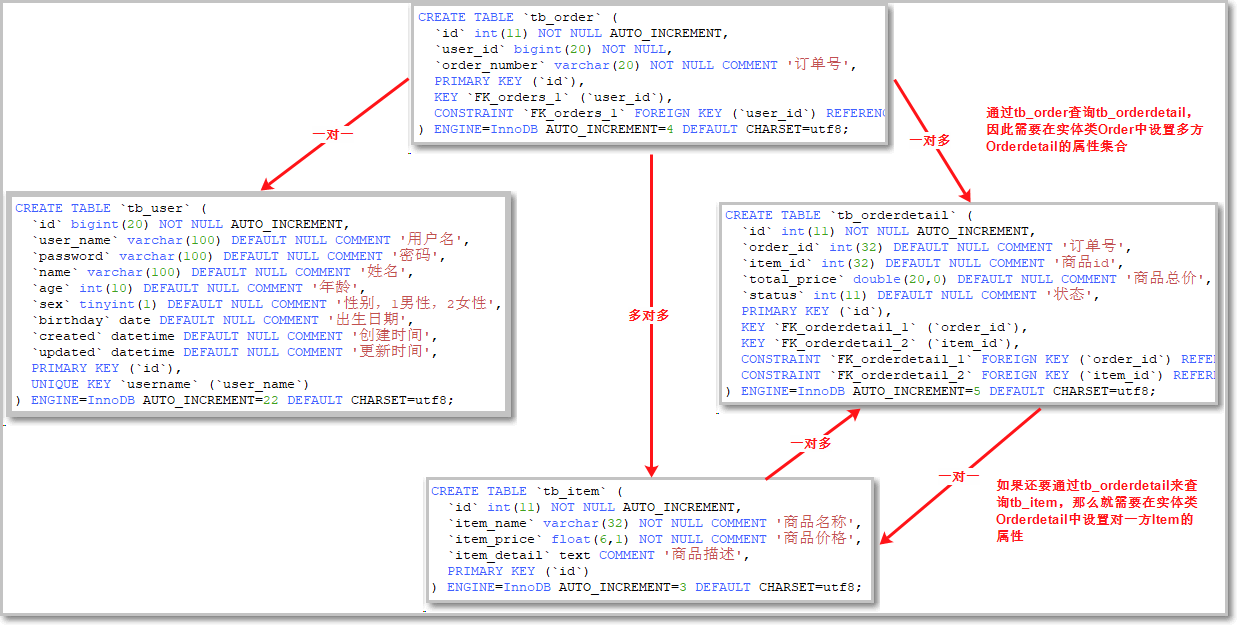
订单详情：商品 = 1 : 1（体现在pojo对象中就是在OrderDetail对象中添加Item对象）

Sql：

#通过订单号20140921001查询订单，查询出下单人信息并且查询出订单详情中的商品数据。

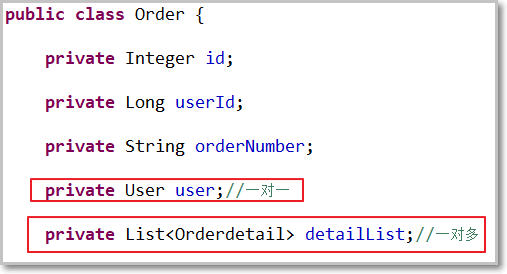


分析：

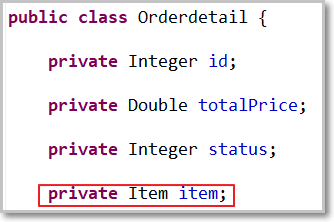


通过订单号20140921001查询订单，查询出下单人信息并且查询出订单详情中的商品数据。

1、由于是通过Order对一查询User,因此要在Order的实体中设置对一User的属性配置，然后还要对多去查询Orderdetail,因此要在Order实体类中设置对多detailList的属性来进行查询，并添加get、set方法：之前已设置过。



2、还要通过Orderdetail去对一查询Item,因此要在Orderdetail中去设置对一Item的属性，并添加get、set方法：



### 步骤一：修改Orderdetail

在Orderdetail中添加item属性：

**public** **class** Orderdetail {

**private** Integer id;

**private** Double totalPrice;

**private** Integer status;

**private** Item item;

**public** Item getItem() {

**return** item;

}

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

**this**.item = item;

}

**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** Integer getId() {

**return** id;

}

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

**this**.id = id;

}

@Override

**public** String toString() {

**return** "Orderdetail [id=" + id + ", totalPrice=" + totalPrice

+ ", status=" + status + ", item=" + item + "]";

}

}

### 步骤二：编写接口方法

**public** **interface** OrderMapper {

Order queryOrderAndUserByOrderNumber(@Param("number") String number);

Order queryOrderAndUserAndOrderdetailByOrderNumber(@Param("number") String number);

Order queryOrderAndUserAndOrderdetailAndItemByOrderNumber(@Param("number") String number);

}

### 步骤三：编写statement

OrderMapper配置（通过在collection标签中嵌套使用association标签）：

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

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

<!-- 对一查询User -->

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

<!--

property="id"：关联对象的主键的属性名称

column：关联对象在当前表中的外键的字段名称，如果重复，可以使用别名或者外键名称

-->

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

</association>

<!-- 对多查询Orderdetail -->

<collection property=*"detailList"* javaType=*"List"* ofType=*"Orderdetail"*>

<!--

property="id"：关联对象的主键的属性名称

column：关联对象在表中的主键名称,如果重复，可以使用别名

-->

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

<!-- 通过Orderdetail对一查询Item -->

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

<!--

property：item的主键的属性名称

column：表中主键的字段名称，如果重名，可以使用别名，或者外键名称

-->

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

</association>

</collection>

</resultMap>

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

select \*,od.id as detail\_id,u.id as uid,i.id as iid from tb\_order o

inner join tb\_user u on o.user\_id = u.id

inner join tb\_orderdetail od on o.id = od.order\_id

inner join tb\_item i on od.item\_id = i.id

where o.order\_number = #{number}

</select>

### ※步骤四：测试

@Test

**public** **void** testqueryOrderAndUserAndOrderdetailAndItemByOrderNumber(){

Order order = orderMapper.queryOrderAndUserAndOrderdetailAndItemByOrderNumber("20140921001");

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

}

## resultMap的继承

如果说有的resultMap中的数据其他地方也要用到，那么可以使用继承的方法来复用即可：



## 高级查询的整理

//

resutlType无法帮助我们自动的去完成映射，所以只有使用resultMap手动的进行映射

resultMap:

type 结果集对应的数据类型

id 唯一标识，被引用的时候，进行指定

autoMapping 开启自动映射

extends 继承

子标签：

association：配置对一的映射

property 定义对象的属性名

javaType 属性的类型

autoMapping 开启自动映射

collection：配置对多的映射

property 定义对象的属性名

javaType 集合的类型

ofType 集合中的元素类型

autoMapping 开启自动映射

# 延迟加载

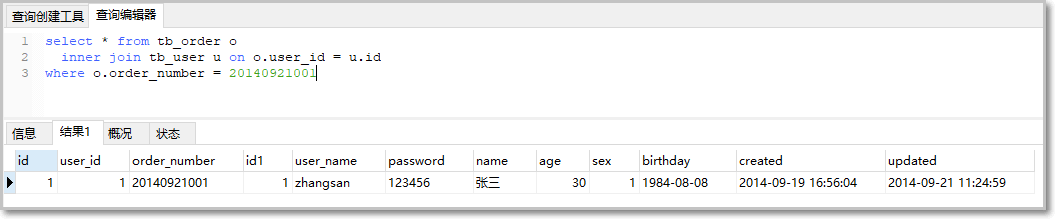
|  |  |  |  |
| --- | --- | --- | --- |
| 设置参数 | 描述 | 有效值 | 默认值 |
| cacheEnabled | 该配置影响这所有映射器中配置的缓存的全局开关。 | true | false | true |
| lazyLoadingEnabled | 延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。特定关联关系中可通过设置fetchType属性来覆盖该项的开关状态。 | true | false | false |
| aggressiveLazyLoading | 当启用时，带有延迟加载属性的对象的加载与否完全取决于对任意延迟属性的调用；反之，每种属性将会按需加载。 | true | false | true |
| mapUnderscoreToCamelCase | 是否开启自动驼峰命名规则（camel case）映射，即从经典数据库列名 A\_COLUMN 到经典 Java 属性名 aColumn 的类似映射。 | true | false | False |

采用之前的配置方式（参考高级查询），肯定是不能做到延迟加载的，因为咱是通过一个查询sql直接查询出所有的数据。为了测试延迟加载的效果，必须改造高级查询的配置，使Order的查询和User或者OrderDetail的查询分开。只有当我们访问Order对象的User或者OrderDetail属性时，才去执行User或者OrderDetail的查询。

## 分析

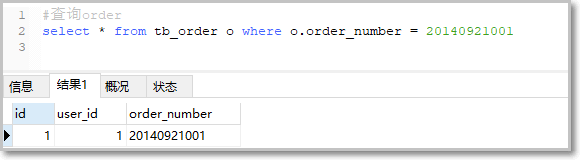
延迟加载需求：通过订单编号20140921001查询order并延迟加载user

如果通过订单编号查询order并且查询user信息，在正常情况下的查询语句应该是：

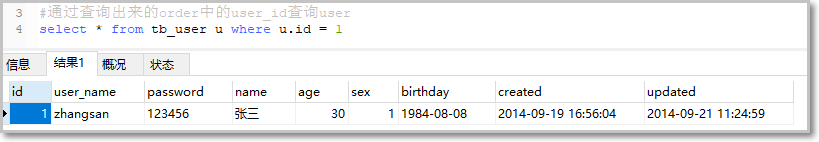


如果改成延迟加载，也就意味着，先查询order,等需要的时候再去查询user,那就相当于将上面的一条语句变成了两条语句：

1、通过订单编号查询order



2、通过查询出来的order中的user\_id查询user



## 步骤一：编写接口方法

在接口中添加方法如下：

Order queryOrderUserLazy(@Param("orderNum")String orderNum);

## 步骤二：编写statement

两个statement就分别相当于两条查询order和查询user的sql语句，然后通过resultMap将延迟加载的查询user的statement关联到查询查询order的statement上。使得在查询order之后，可以延迟加载user.

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

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

<!-- 关联查询user

select：延迟加载user,通过调用另外的user的statement来发送第二个语句查询user

column:查询user的参数，该id的信息其实是在查询出来的order中的user\_id

如果参数有多个，可以使用{user\_id=id,user\_name = userName}的方式来传参

-->

<association property=*"user"* javaType=*"User"* select=*"queryUserByUserIdOfOrder"* column=*"user\_id"*></association>

</resultMap>

<!-- 通过order\_number查询order -->

<select id=*"queryOrderUserLazy"* resultMap=*"orderUserLazyMap"*>

select \* from tb\_order where order\_number = #{orderNumber}

</select>

<!-- 通过order中的user\_id查询user -->

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

select \* from tb\_user where id = #{id}

</select>

## 步骤三：测试

@Test

**public** **void** testQueryOrderUserLazy(){

Order order = orderMapper.queryOrderUserLazy("20140921001");

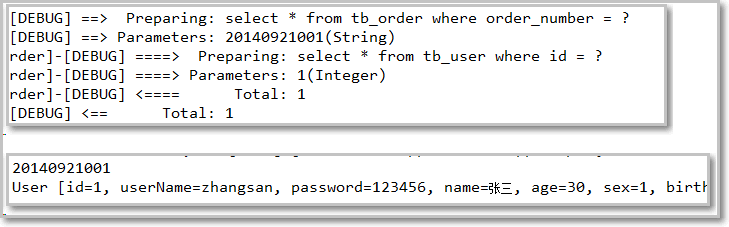
System.***out***.println(order.getOrderNumber());

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

}

测试结果：发现并没有延迟加载，还是一起查询了。

原因：没有开启延迟加载的开关



## 步骤四：开启延迟加载的开关

在全局配置文件中开启延迟加载的开关：

<settings>

<!-- 设置驼峰参数 -->

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

<!-- 关闭二级缓存，默认是开启，false：关闭 -->

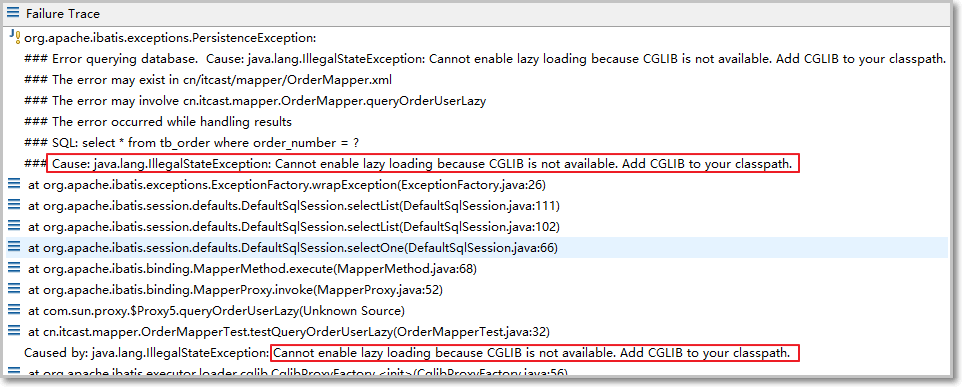
<setting name=*"cacheEnabled"* value=*"false"*/>

<!-- 开启延迟加载 -->

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

</settings>

报错如下：发现需要cglib的动态代理的jar包



## 步骤五：添加cglib的依赖

在pom.xml中添加cglib的依赖：

<!-- cglib -->

<dependency>

<groupId>cglib</groupId>

<artifactId>cglib</artifactId>

<version>3.1</version>

</dependency>

## 步骤六：再次测试

再次测试：可以正常查询，但是任然没有延迟加载。

@Test

**public** **void** testQueryOrderUserLazy(){

Order order = orderMapper.queryOrderUserLazy("20140921001");

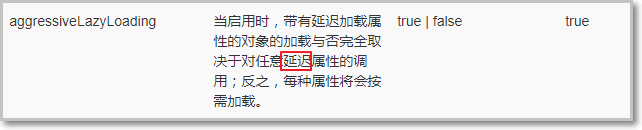
System.***out***.println(order.getOrderNumber());

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

}

发现：当执行完黄色代码后，就发送了第一条查询order的语句，而到执行完红色代码后，获取order中的任意属性（包括非延迟属性）时，就发送了第二条查询user的语句。而如果真的是延迟加载，那么应该在最后一条绿色语句执行时才会发送sql查询user，显然延迟加载并没有成功。

原因是，在settings中还有一个延迟加载相关的配置如下：



当该配置默认为true是，当获取order中任意属性的时候都会立即加载user对象，注意是任意属性，而不是任意延迟属性，这个文档的翻译有问题。

因此只有当在settings中设置该属性为false时，才会按需进行延迟加载。

在settings中配置aggressiveLazyLoading属性为false.

<settings>

<!-- 设置驼峰参数 -->

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

<!-- 关闭二级缓存，默认是开启，false：关闭 -->

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

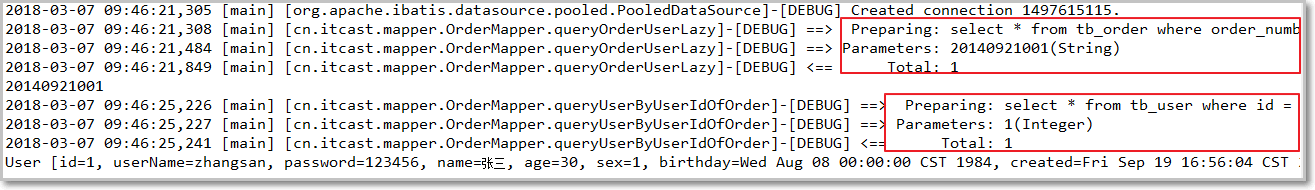
<!-- 开启延迟加载 -->

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

<setting name=*"aggressiveLazyLoading"* value=*"false"*/>

</settings>

再次测试：发现延迟加载成功



注意：延迟加载时第一个sql语句查询出来的order对象中的userId是没有值，可以认为这是一个mybatis的bug

## 延迟加载分析图例

注意：association标签总的colunm属性，如果传递的参数只有一个的话，直接使用user\_id即可，如果是多个可以使用{user\_id=id,user\_nam=userName}来传递。



也可以修改如下：



# ※如果sql语句中出现’<’的解决方案

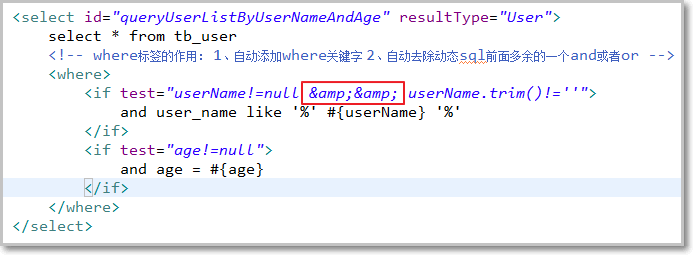
1、使用xml中的字符实体



【示例】



修改如下即可:



2、使用<![CDATA[ < ]]>

CDATA 内部的所有东西都会被解析器忽略,因此只要将字符实体编写在其中，就不需要进行转义。

如下:

