Mybatis

Ssm：springmvc spring mybatis

# JDBC回顾

## 创建mybatis库，执行sql

----> Navicat

1. mysql

create table `mybatis-912`; -- 注意是 `` 不是 ''

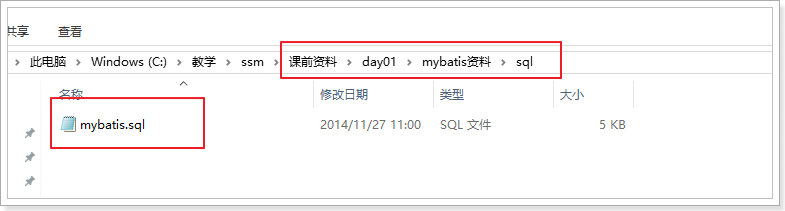
2. 表的导入:

打开数据库 mybatis-912

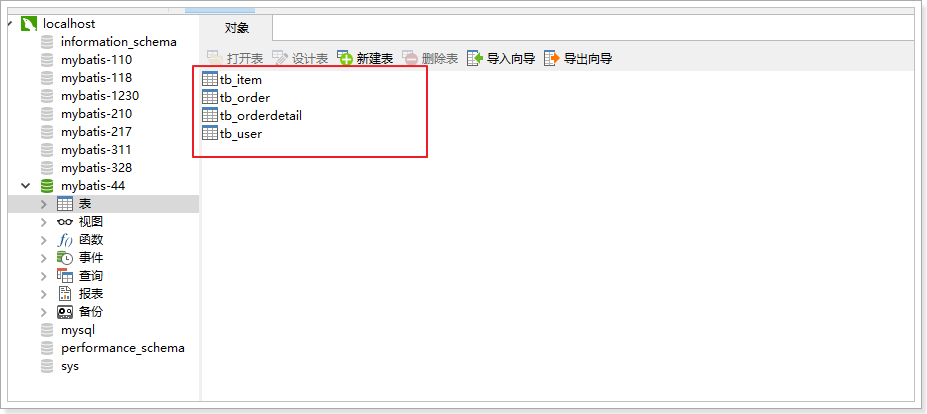
点击表

将文件拖到中间的区域即可

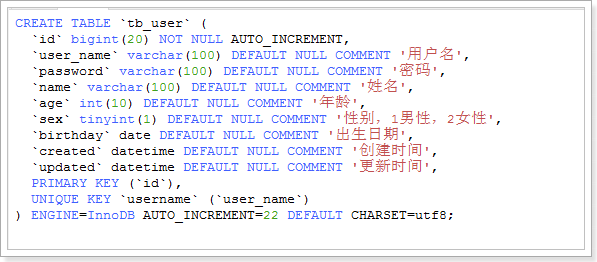
SQL文件：

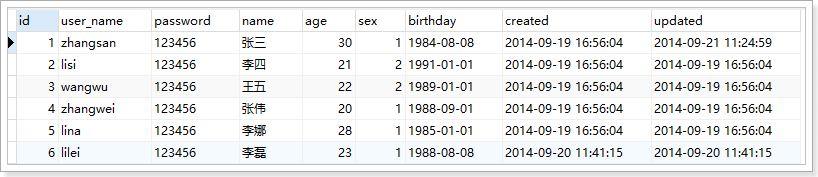


执行后：



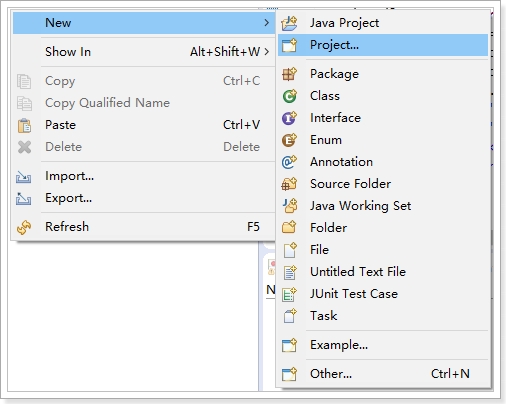
tb\_user表：



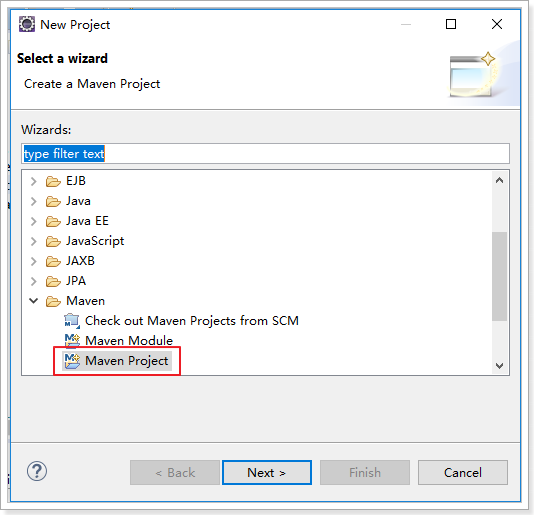


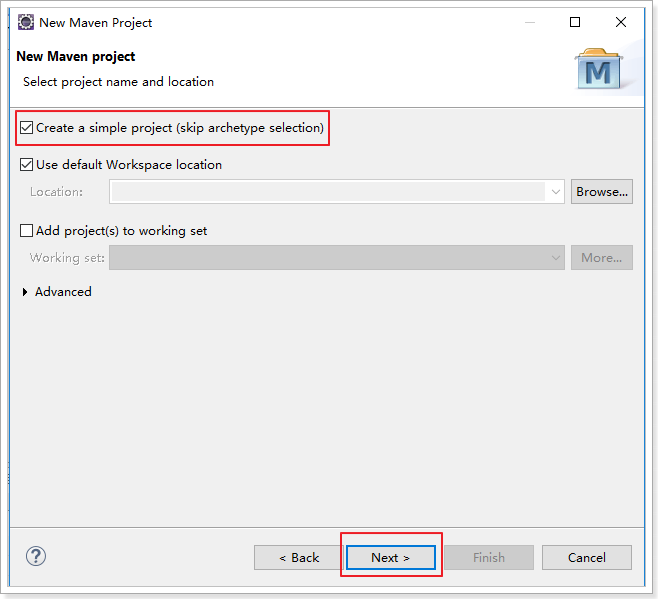
## 创建maven工程

右键—>new-->Project

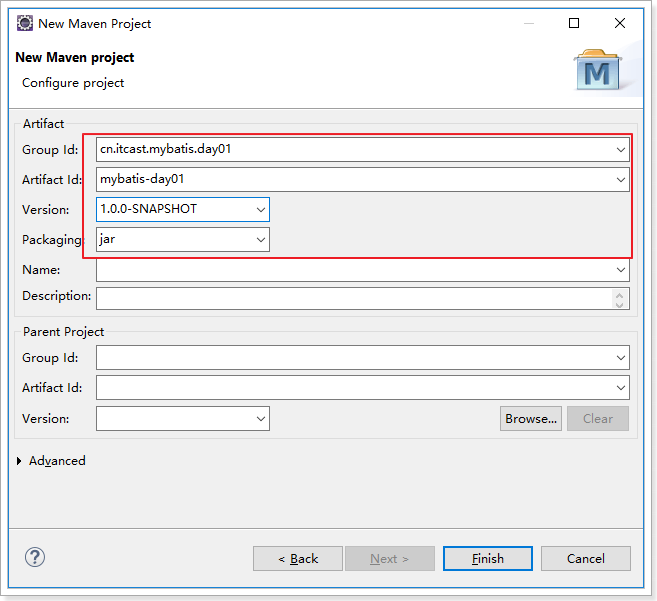


选择：Maven—>Maven Project

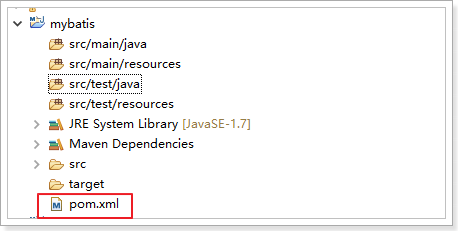




坐标：



## 引入mysql依赖包



Pom.xml内容：

<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>

<!-- 导入父工程 坐标 -->

<parent>

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

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

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

<artifactId>mybatis</artifactId>

<version>1.0.0-SNAPSHOT</version>

<dependencies>

<!-- MySql -->

<dependency>

<groupId>mysql</groupId>

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

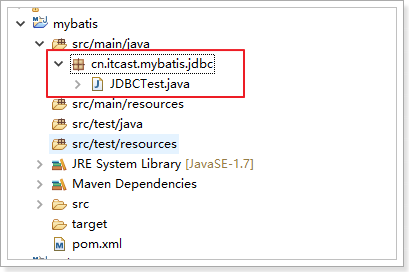
</dependency>

</dependencies>

</project>

## 需求 --> 根据id查询用户信息

## JDBCTest



1. 加载驱动
2. 获取连接
3. 获取prepareStatement 对象
4. 设置查询参数
5. 执行查询 , 获取结果集
6. 操作结果集
7. 关闭连接

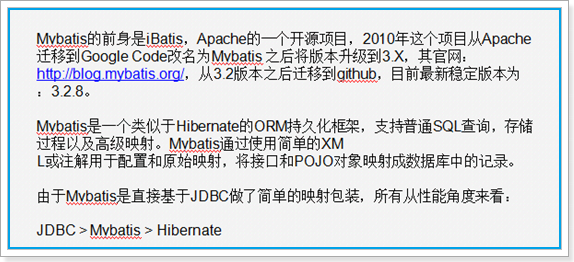
|  |
| --- |
| package cn.itcast.mybatis.jdbc;  import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.PreparedStatement;  import java.sql.ResultSet;  public class JDBCTest {  public static void main(String[] args) throws Exception {  Connection conn = null;  PreparedStatement ps = null;  ResultSet rs = null;  try {  // 加载驱动  Class.forName("com.mysql.jdbc.Driver");  // 获取连接  String url = "jdbc:mysql://127.0.0.1:3306/mybatis-44";  String user = "root";  String password = "root";  conn = DriverManager.getConnection(url, user, password);  // 获取statement，preparedStatement  String sql = "select \* from tb\_user where id=?";  ps = conn.prepareStatement(sql);  // 设置参数  ps.setLong(1, 1l);  // 执行查询，获取结果集  rs = ps.executeQuery();  // 处理结果集  while(rs.next()){  System.out.println(rs.getString("user\_name"));  System.out.println(rs.getString("name"));  System.out.println(rs.getInt("age"));  }  } finally {  // 关闭连接，释放资源  if(rs!=null){  rs.close();  }  if(ps!=null){  ps.close();  }  if(conn!=null){  conn.close();  }  }  }  } |

## 缺点及解决方案



# MyBatis介绍

## 简介



## 官方文档



## 特点

Mybatis：

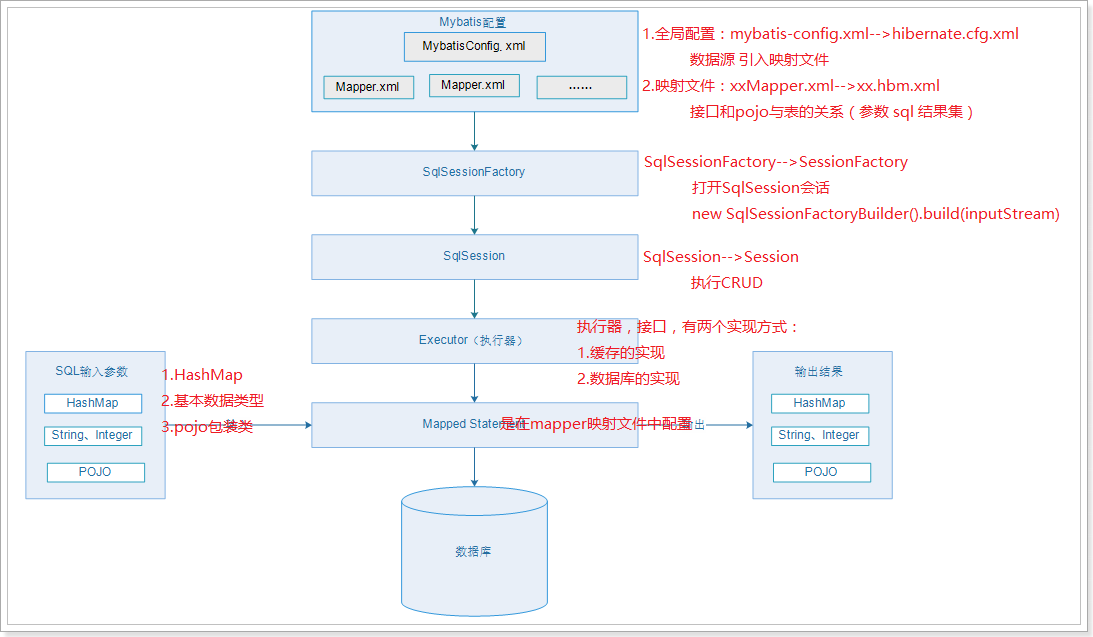
1. 支持自定义SQL、存储过程、及高级映射
2. 实现自动对SQL的参数设置
3. 实现自动对结果集进行解析和封装
4. 通过XML或者注解进行配置和映射，大大减少代码量
5. 数据源的连接信息通过配置文件进行配置

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，所以没有实现完全自动化，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：mybatis-config.xml🡪hibernate.cfg.xml，作用：配置数据源，引入映射文件

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

1. SqlSessionFactory

相当于Hibernate的SessionFactory，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

相当于Hibernate的Session，作用：执行CRUD操作

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

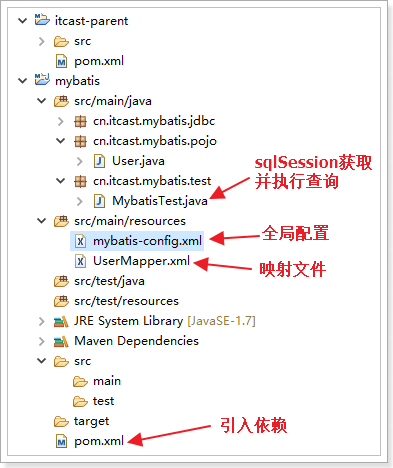
1. Mapped Statement

在映射文件里面配置，包含3部分内容： 1>具体的sql 2>sql执行所需的参数类型 3>sql执行结果的封装类型

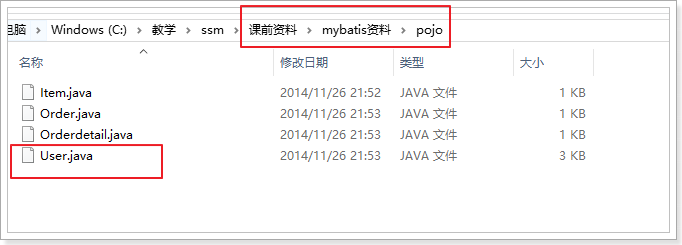
参数类型和结果集封装类型包括3种： 1>HashMap 2>基本数据类型 3>pojo

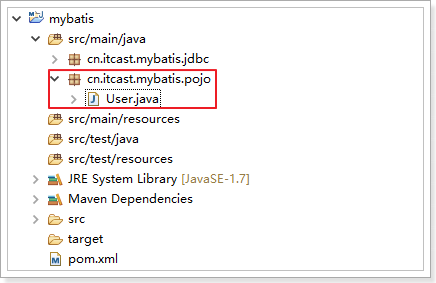
# 快速入门（Mybatis第一个程序）

## 目录结构



pojo中的User.java参照课前资料：





## 引入依赖（pom.xml）

参考itcast-parent工程

<!-- Mybatis -->

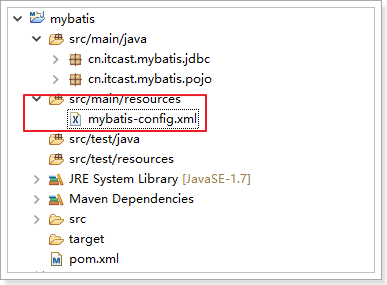
<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

</dependency>

## 全局配置文件（mybatis-config.xml）



Mybatis-config.xml内容，参照官方文档：

<?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>

<!-- 环境：说明可以配置多个，default:指定生效的环境 -->

<environments default=*"development"*>

<!-- id:环境的唯一标识 -->

<environment id=*"development"*>

<!-- 事务管理器，type：类型 -->

<transactionManager type=*"JDBC"* />

<!-- 数据源：type-池类型的数据源 -->

<dataSource type=*"POOLED"*>

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

<property name=*"url"* value=*"jdbc:mysql://127.0.0.1:3306/mybatis-49"* />

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

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

</dataSource>

</environment>

</environments>

<!-- 映射文件 -->

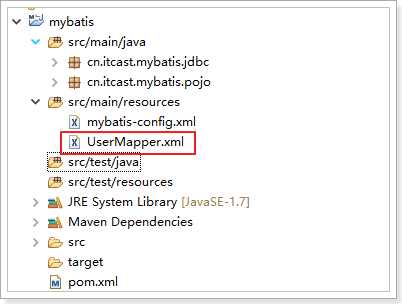
<mappers>

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

</mappers>

</configuration>

## 映射文件（UserMapper.xml）



UserMapper.xml映射文件内容，参照官方文档：

<?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">

<!-- namespace(命名空间)：映射文件的唯一标识 -->

<mapper namespace=*"UserMapper"*>

<!-- 查询的statement，id：在同一个命名空间下的唯一标识，resultType：sql语句的结果集封装类型 -->

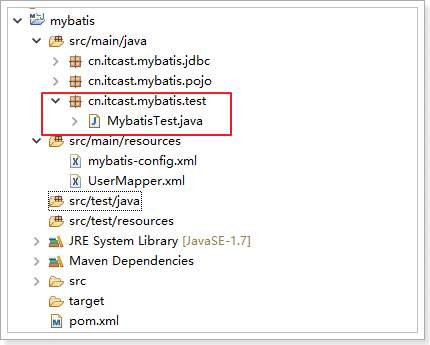
<select id=*"queryUserById"* resultType=*"User"*>

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

</select>

</mapper>

## 编写mybatis程序（MybatisTest.java）



MybatisTest.java内容：

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

SqlSession sqlSession = **null**;

**try** {

// 指定mybatis的全局配置文件

String resource = "mybatis-config.xml";

// 读取mybatis-config.xml配置文件

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

// 构建sqlSessionFactory

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

// 获取sqlSession回话

sqlSession = sqlSessionFactory.openSession();

// 执行查询操作，获取结果集。参数：1-命名空间（namespace）+“.”+statementId,2-sql的占位符参数

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

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

} **finally** {

// 关闭连接

**if** (sqlSession != **null**) {

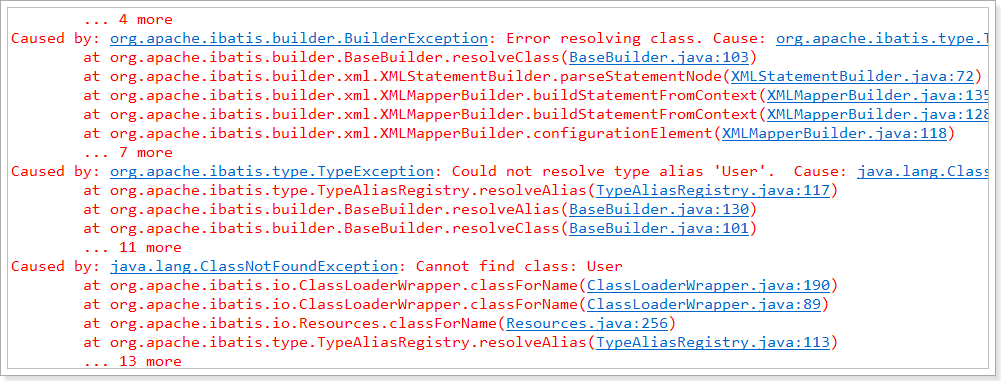
sqlSession.close();

}

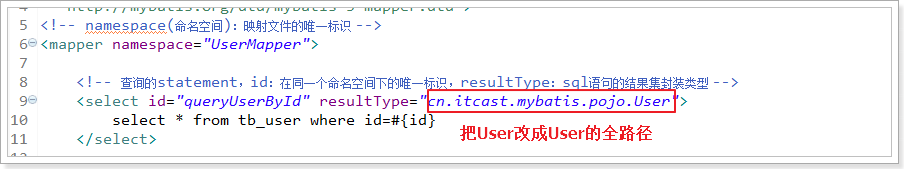
}

}

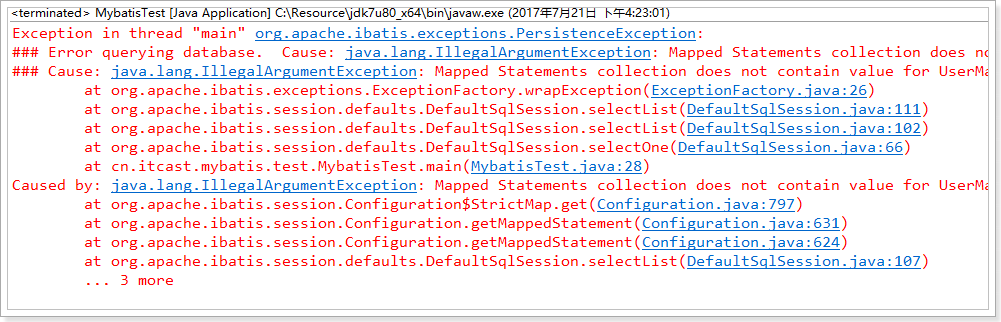
## 测试运行报错



原因：



## 其他典型出错



原因：



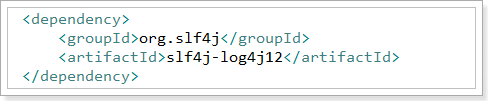
# 引入log日志

打印日志2个步骤：

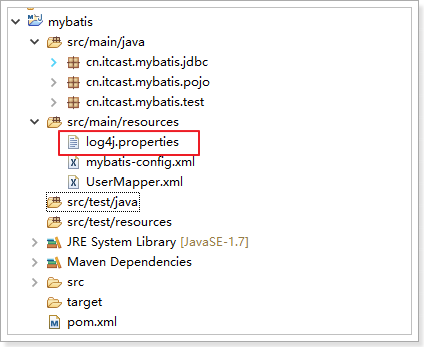
1. 在pom.xml中，引入slf4j的依赖
2. 在src/main/resources目录下添加log4j.properties文件

## 引入日志依赖包（pom.xml）

参照父工程



## 添加log4j.properties



内容如下： level --> log4j-1.2.16.jar org.apache.log4j.Level

|  |
| --- |
| log4j.rootLogger=DEBUG,A1  log4j.logger.org.apache=DEBUG  log4j.appender.A1=org.apache.log4j.ConsoleAppender  log4j.appender.A1.layout=org.apache.log4j.PatternLayout  log4j.appender.A1.layout.ConversionPattern=%-d{yyyy-MM-dd HH:mm:ss,SSS} [%t] [%c]-[%p] %m%n |

Appender --> 接口 --> 定义了 输出的 (控制台,file ...)

%-d 输出时间,左对齐

[%t] --> 线程

[%c] --> 类

[%p] --> 输出级别

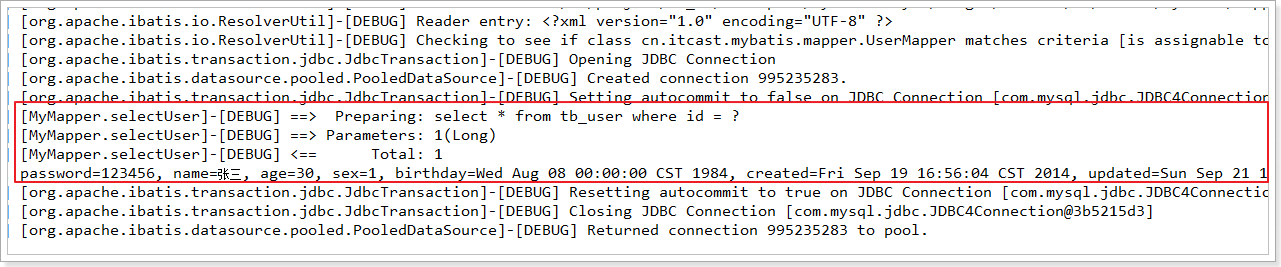
%m%n 输出message 和 换行

%-d [%t] [%c] [%p] %m%n

2017-09-12 19:58:08,739 [main] [org.apache.ibatis.datasource.pooled.PooledDataSource]-[DEBUG] Returned connection 25117323 to pool.

## 日志输出

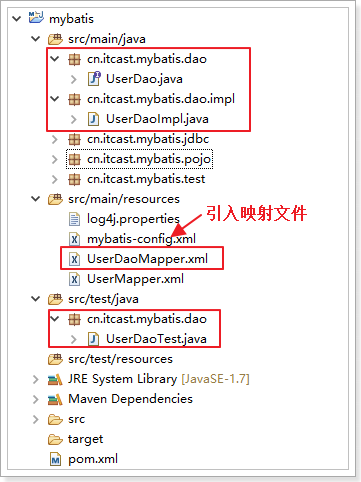
控制台截图：



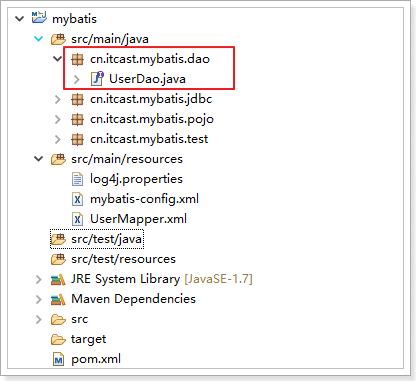
## MyBatis使用步骤总结

1. 配置mybatis-config.xml 全局的配置文件 (1、数据源，2、外部的mapper)
2. 创建SqlSessionFactory
3. 通过SqlSessionFactory创建SqlSession对象
4. 通过SqlSession操作数据库 CRUD
5. 调用session.commit()提交事务 (增删改)
6. 调用session.close()关闭会话

# 完整的CRUD操作



## 创建UserDao接口



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> queryUserAll();

/\*\*

\* 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

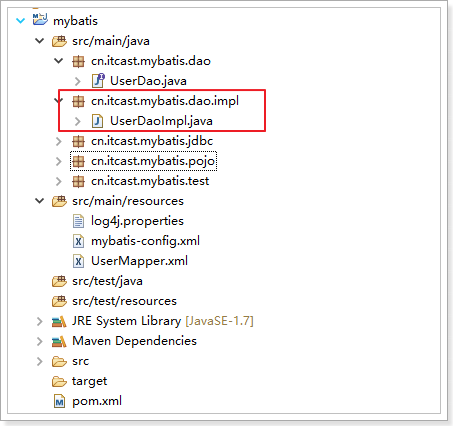
\* **@param** id

\*/

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

}

## 创建UserDaoImpl



实现UserDao接口：

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

**private** SqlSession sqlSession;

**public** UserDaoImpl(SqlSession sqlSession){

**this**.sqlSession = sqlSession;

}

@Override

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

**return** **this**.sqlSession.selectOne("UserDaoMapper.queryUserById", id);

}

@Override

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

**return** **this**.sqlSession.selectList("UserDaoMapper.queryUserAll");

}

@Override

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

**this**.sqlSession.insert("UserDaoMapper.insertUser", user);

**this**.sqlSession.commit(); // 不提交就会失效

}

@Override

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

**this**.sqlSession.update("UserDaoMapper.updateUser", user);

**this**.sqlSession.commit();

}

@Override

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

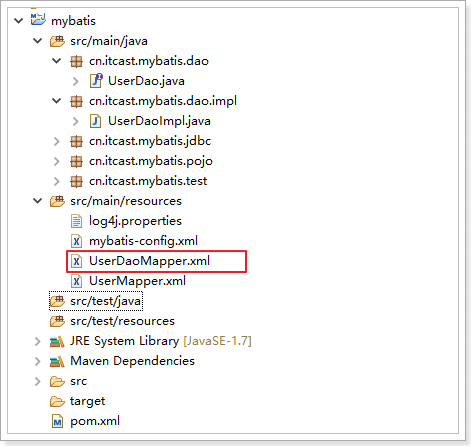
**this**.sqlSession.delete("UserDaoMapper.deleteUserById", id);

**this**.sqlSession.commit();

}

}

## 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 namespace=*"UserDaoMapper"*>

<!-- 一般id和方法名一致 -->

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

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

</select>

<!-- 这里mybatis会自动根据返回的是一个还是多个 来封装为list<user> 还是 user,所以返回类型resultType 还是 User -->

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

select \* from tb\_user

</select>

<insert id=*"insertUser"* parameterType=*"cn.itcast.mybatis.pojo.User"*>

INSERT INTO tb\_user (

user\_name,

password,

name,

age,

sex,

birthday,

created,

updated

)

VALUES

(

#{userName},

#{password},

#{name},

#{age},

#{sex},

#{birthday},

NOW(),

NOW() <!-- select now() 数据库查询当前时间 -->

);

</insert>

<update id=*"updateUser"* parameterType=*"cn.itcast.mybatis.pojo.User"*>

UPDATE tb\_user

SET

user\_name = #{userName},

password = #{password},

name = #{name},

age = #{age},

sex = #{sex},

birthday = #{birthday},

updated = NOW()

WHERE

(id = #{id});

</update>

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

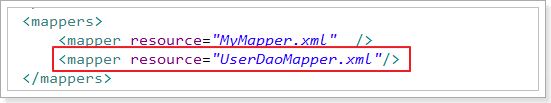
delete from tb\_user where id=#{id}

</delete>

</mapper>

## 引入UserDaoMapper.xml

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



如果

### Error building SqlSession.

### Cause: org.apache.ibatis.builder.BuilderException: Error creating document instance.

Cause: org.xml.sax.SAXParseException; lineNumber: 35; columnNumber: 10; 元素类型 "mapper" 的结束标记必须以 '>' 分隔符结束。

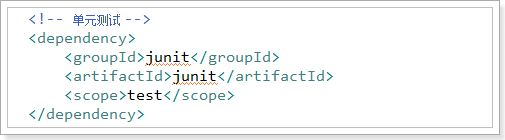
是这种错误 , 那么就是

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

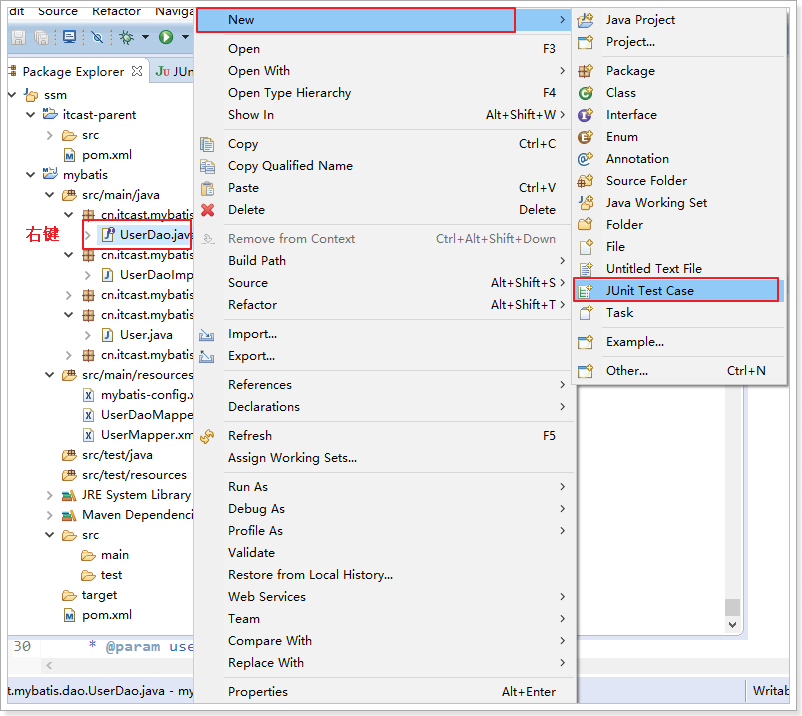
## 测试UserDao

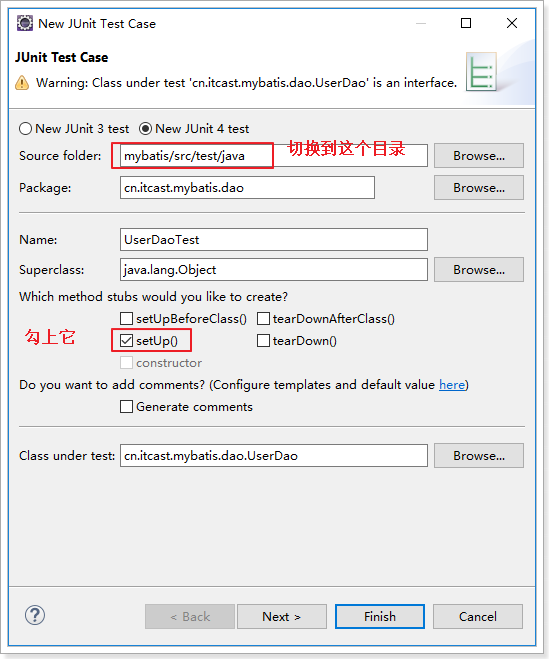
针对UserDao的测试，咱们使用Junit进行测试。

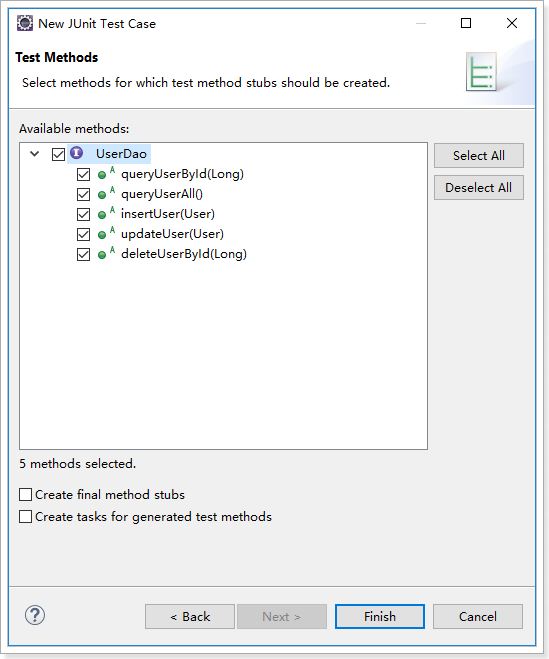
1. 引入Junit依赖：参照父工程，在mybatis工程中的pom.xml中引入junit的依赖



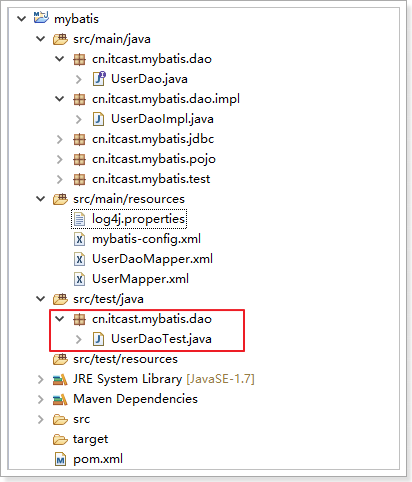
1. 创建junit测试用例：右键UserDao—>New—>Junit Test Case







创建完成之后的目录结构：



1. 编写测试用例：

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

**private** UserDao userDao;

@Before

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

String resource = "mybatis-config.xml";

// 读取配置文件

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

// 构建sqlSessionFactory

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

// 获取sqlSession

SqlSession sqlSession = sqlSessionFactory.openSession();

// 初始化userDao

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

}

@Test

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

User user = **this**.userDao.queryUserById(1l);

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

}

@Test

**public** **void** testQueryUserAll() {

List<User> users = **this**.userDao.queryUserAll();

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

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

}

}

@Test

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

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

user.setAge(18);

user.setName("柳岩");

user.setPassword("123456");

user.setUserName("yanyan");

user.setSex(3);

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

**this**.userDao.insertUser(user);

}

@Test

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

// 查询

User user = **this**.userDao.queryUserById(7l);

// 更新

user.setAge(28);

user.setPassword("111111");

**this**.userDao.updateUser(user);

}

@Test

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

**this**.userDao.deleteUserById(7l);

}

}

## 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，引入UserDaoMapper.xml

4、 编写UserDao的Junit Test Case测试用例

## 解决UserName为null

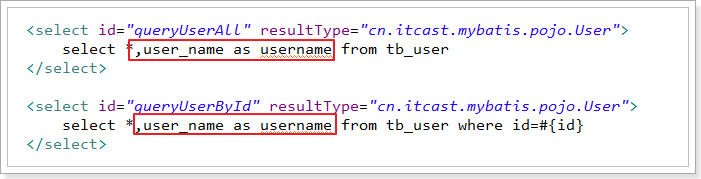
查询数据的时候，查不到userName的信息，原因：数据库的字段名是user\_name

POJO中的属性名是userName

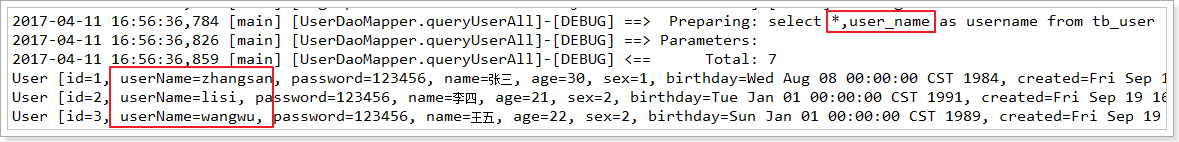
两端不一致，造成mybatis无法填充对应的字段信息。修改方法：在sql语句中使用别名

解决方案1：在sql语句中使用别名

修改映射文件（UserDaoMapper.xml）中的sql语句



控制台Log日志：



# 动态代理Mapper实现类

## 思考CRUD中的问题

1、接口->实现类->mapper.xml。

2、实现类中，使用mybatis的方式非常类似。

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接口的动态代理实现，需要满足以下条件：

1. 映射文件中的命名空间与Mapper接口的全路径一致
2. 映射文件中的statementId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）

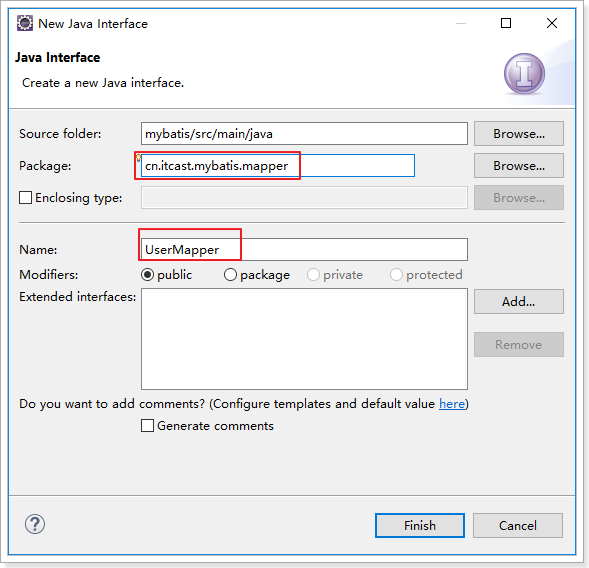
## 使用动态代理改造CRUD

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）。

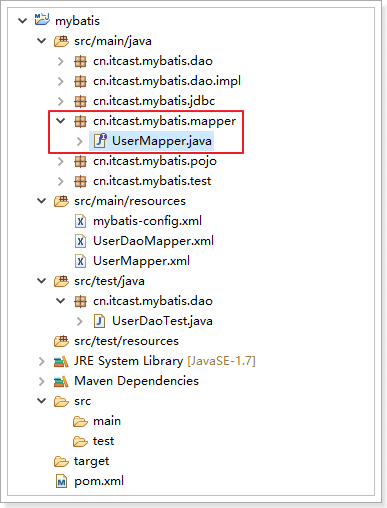
当然，仍有少部分项目会保留Dao的命名，以后见到Mapper或者Dao的命名都不要奇怪。

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

### 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 {

/\*\*

\* 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> queryUserAll();

/\*\*

\* 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

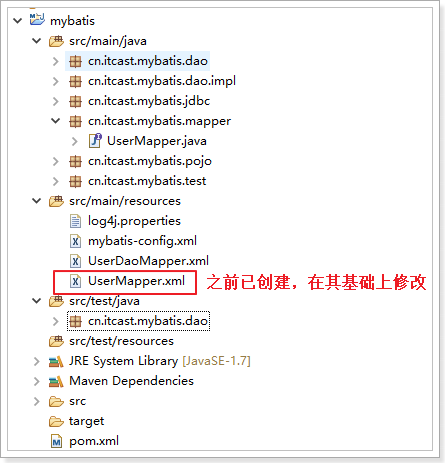
\* **@param** id

\*/

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

}

### UserMapper映射文件



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

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

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

</select>

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

select \* from tb\_user

</select>

<insert id=*"insertUser"* parameterType=*"cn.itcast.mybatis.pojo.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 id=*"updateUser"* parameterType=*"cn.itcast.mybatis.pojo.User"*>

UPDATE tb\_user

SET

user\_name = #{userName},

password = #{password},

name = #{name},

age = #{age},

sex = #{sex},

birthday = #{birthday},

updated = NOW()

WHERE

(id = #{id});

</update>

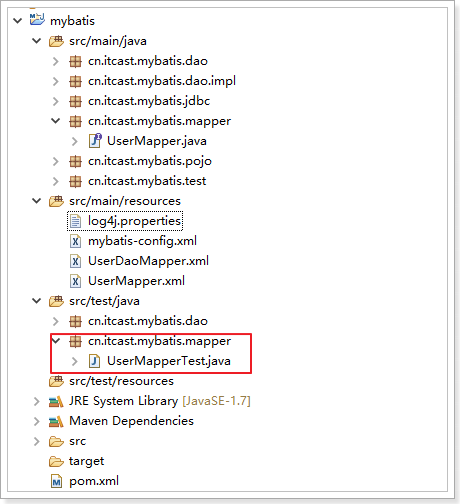
<delete id=*"deleteUserById"* parameterType=*"java.lang.Long"*>

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

</delete>

</mapper>

### UserMapperTest测试



添加UserMapper.java的测试用例。只需要修改一句代码即可

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

**private** UserMapper userMapper;

@Before

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

// 读取mybatis的全局配置文件

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

// 构建sqlSessionFactory

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

// 获取sqlSession会话

SqlSession sqlSession = sqlSessionFactory.openSession();

// 初始化userDao

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

}

@Test

**public** **void** testQueryUserAll() {

List<User> userList = **this**.userMapper.queryUserAll();

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

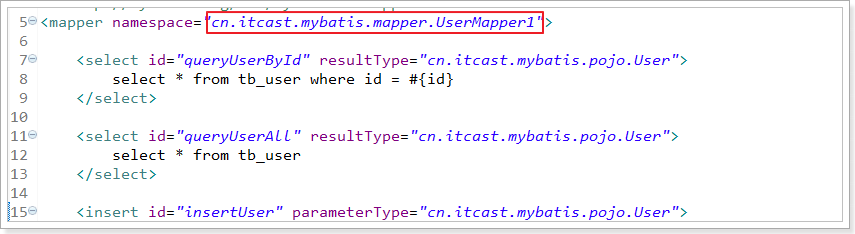
System.***out***.println(user);

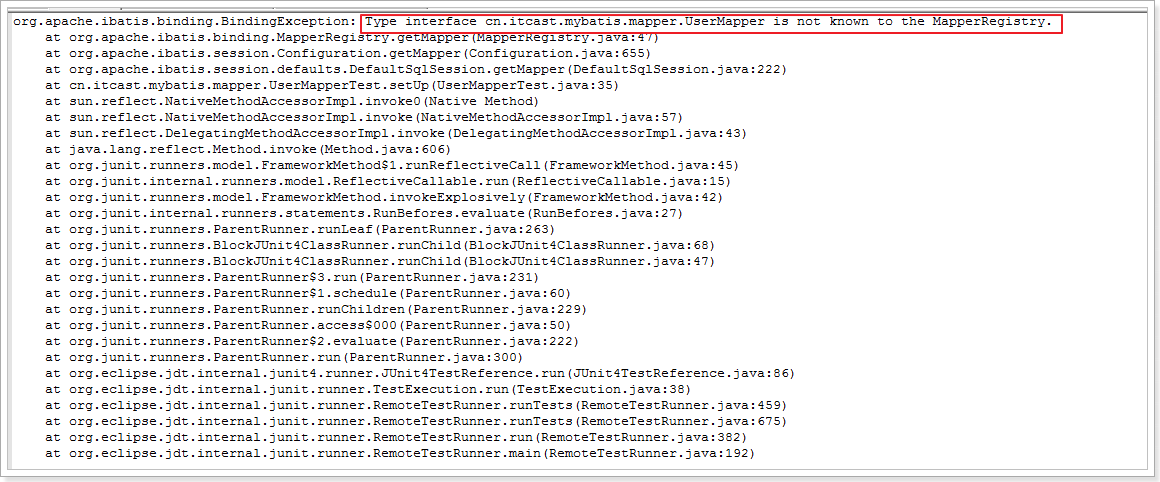
}

}

}

1. 如果名称空间不和mapper接口的全路径保持一致





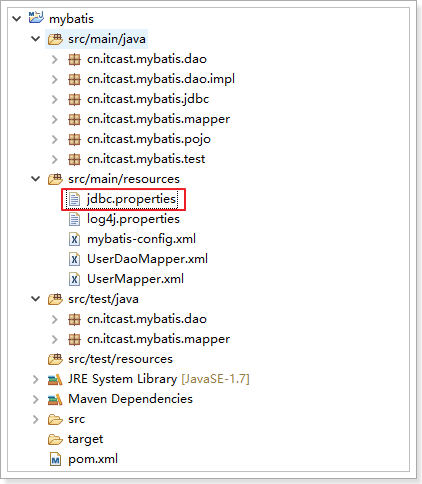
# mybatis-config.xml配置

mybatis-config.xml讲究严格的顺序，具体顺序遵循文档的顺序



## properties属性读取外部资源

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-44

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：相对路径 classpath的路径 ，url：绝对路径 比如 c:/xx-->

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

<!-- 环境：说明可以配置多个，default:指定生效的环境 -->

<environments default=*"development"*>

<!-- id:环境的唯一标识 -->

<environment id=*"development"*>

<!-- 事务管理器，type：类型 -->

<transactionManager type=*"JDBC"* />

<!-- 数据源：type-池类型的数据源 -->

<dataSource type=*"POOLED"*>

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

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

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

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

</dataSource>

</environment>

</environments>

<!-- 映射文件 -->

<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，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">

<!-- 注意顺序问题 , 有严格 的顺序 -->

<configuration>

<!-- 引入外部资源文件，resource：classpath路径，url：绝对路径（不建议使用） -->

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

<settings>

<!-- 开启驼峰匹配：经典的数据库列名（多个单词下划线连接）映射到经典的java属性名（多个单词驼峰连接） -->

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

</settings>

<environments default=*"development"*>

<environment id=*"development"*>

<transactionManager type=*"JDBC"* />

<dataSource type=*"POOLED"*>

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

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

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

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

</dataSource>

</environment>

</environments>

<mappers>

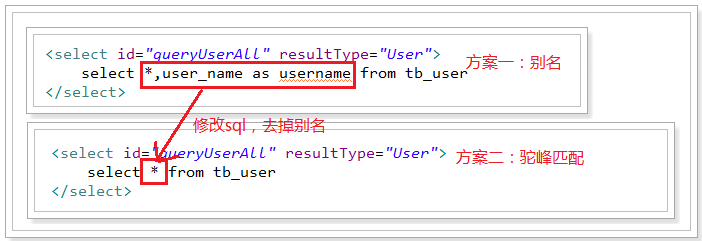
<mapper resource=*"UserMapper.xml"* />

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

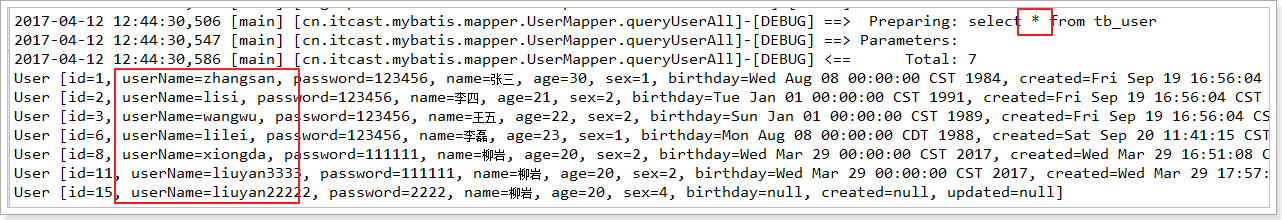
</mappers>

</configuration>

在UserMapper.xml中修改sql语句删除别名



测试日志截图，如下：



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

<?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：classpath路径，url：绝对路径（不建议使用） -->

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

<settings>

<!-- 开启驼峰匹配：经典的数据库列名（多个单词下划线连接）映射到经典的java属性名（多个单词驼峰连接） -->

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

</settings>

<typeAliases>

<!-- 类型别名：type-pojo类的全路径，alias-别名名称（可随便写，推荐和类名一致） -->

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

</typeAliases>

<environments default=*"development"*>

<environment id=*"development"*>

<transactionManager type=*"JDBC"* />

<dataSource type=*"POOLED"*>

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

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

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

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

</dataSource>

</environment>

</environments>

<mappers>

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

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

</mappers>

</configuration>

--------- 注意 使用到类别名

<!-- 使用了类别名 , 返回类型对大小写不敏感 , user,User都可以 -->

<!-- <select id="queryUserAll" resultType="cn.sp.mybatis.pojo.User"> -->

<select id="queryUserAll" resultType="user">

select \* from tb\_user

</select>

缺点：每个pojo类都要去配置。

### 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。

<typeAliases>

<!-- 类型别名：type-pojo类的全路径，alias-别名名称（可随便写，推荐和类名一致） -->

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

<!-- 开启别名包扫描，name：包路径，扫描的别名就是类名，并且大小写不敏感 -->

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

</typeAliases>

---------- xxxmapper.xml的使用和别名一样

注意:这里的 resultType="User" , 不是user ,否则找不到类,这个是区别

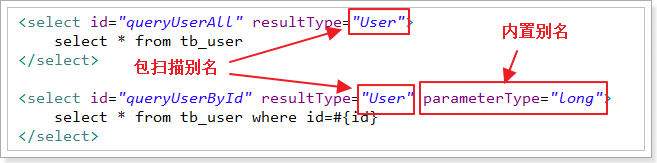
<select id="queryUserAll" resultType="user">

select \* from tb\_user

</select>

-------> 总结: 两种别名使用 , 跟类名一致(包括首字母大小写)

在映射文件中使用类型别名：



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**，需要注意的是由于重载原始类型的名称所做的特殊处理。

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

## typeHandlers（类型处理器）

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。

|  |  |  |
| --- | --- | --- |
| **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）。 |

这个是系统自动调用的,但是 :

你可以重写类型处理器或创建你自己的类型处理器来处理不支持的或非标准的类型。具体做法为：实现 org.apache.ibatis.type.TypeHandler 接口，或继承一个很便利的类 org.apache.ibatis.type.BaseTypeHandler，然后可以将它映射到一个 JDBC 类型。比如：

// ExampleTypeHandler.java

@MappedJdbcTypes(JdbcType.VARCHAR)

public class ExampleTypeHandler extends BaseTypeHandler<String> {

@Override

public void setNonNullParameter(PreparedStatement ps, int i, String parameter, JdbcType jdbcType) throws SQLException {

ps.setString(i, parameter);

}

@Override

public String getNullableResult(ResultSet rs, String columnName) throws SQLException {

return rs.getString(columnName);

}

@Override

public String getNullableResult(ResultSet rs, int columnIndex) throws SQLException {

return rs.getString(columnIndex);

}

@Override

public String getNullableResult(CallableStatement cs, int columnIndex) throws SQLException {

return cs.getString(columnIndex);

}

}

<!-- mybatis-config.xml -->

<typeHandlers>

<typeHandler handler="org.mybatis.example.ExampleTypeHandler"/>

</typeHandlers>

使用这个的类型处理器将会覆盖已经存在的处理 Java 的 String 类型属性和 VARCHAR 参数及结果的类型处理器。要注意 MyBatis 不会窥探数据库元信息来决定使用哪种类型，所以你必须在参数和结果映射中指明那是 VARCHAR 类型的字段，以使其能够绑定到正确的类型处理器上。这基于一个事实：MyBatis 直到语句被执行才会去关心数据类型。

MyBatis 会通过窥探属性的原始类型（generic type）来推断由类型处理器处理的 Java 类型，不过这种行为可以通过两种方法改变：

* 在类型处理器的配置元素（typeHandler element）上增加一个 javaType 属性（比如：javaType="String"）；
* 在类型处理器的类上（TypeHandler class）增加一个 @MappedTypes 注解来指定与其关联的 Java 类型列表。 如果在 javaType 属性中也同时指定，则注解方式将被忽略。

可以通过两种方式来指定被关联的 JDBC 类型：

* 在类型处理器的配置元素上增加一个 javaType 属性（比如：javaType="VARCHAR"）；
* 在类型处理器的类上（TypeHandler class）增加一个 @MappedJdbcTypes 注解来指定与其关联的 JDBC 类型列表。 如果在 javaType 属性中也同时指定，则注解方式将被忽略。

最后，可以让 MyBatis 为你查找类型处理器：

<!-- mybatis-config.xml -->

<typeHandlers>

<package name="org.mybatis.example"/>

</typeHandlers>

注意在使用自动检索（autodiscovery）功能的时候，只能通过注解方式来指定 JDBC 的类型。

你能创建一个原始的类型处理器，它可以处理多于一个类。为达到此目的，需要增加一个接收该类作为参数的构造器，这样在构造一个类型处理器的时候 MyBatis 就会传入一个具体的类。

//GenericTypeHandler.java

public class GenericTypeHandler<E extends MyObject> extends BaseTypeHandler<E> {

private Class<E> type;

public GenericTypeHandler(Class<E> type) {

if (type == null) throw new IllegalArgumentException("Type argument cannot be null");

this.type = type;

}

...

EnumTypeHandler 和 EnumOrdinalTypeHandler 都是原始的类型处理器（generic TypeHandlers），我们将会在接下来的部分详细探讨。

### **处理枚举类型**

若想映射枚举类型 Enum，则需要从 EnumTypeHandler 或者 EnumOrdinalTypeHandler 中选一个来使用。

比如说我们想存储用来四舍五入一些数字时用到的舍入模式。默认情况下，MyBatis 会利用 EnumTypeHandler 来把 Enum 值转换成对应的名字。

**注意 EnumTypeHandler 在某种意义上来说是比较特别的，其他的处理器只针对某个特定的类，而它不同，它会处理任意继承了 Enum 的类。**

不过，我们可能不会去存储名字，相反我们的 DBA 会坚持使用整形值代码。那也一样轻而易举：在配置文件中把 EnumOrdinalTypeHandler 加到 typeHandlers 中即可，这样每个 RoundingMode 将通过他们的序数值来映射成对应的整形。

<!-- mybatis-config.xml -->

<typeHandlers>

<typeHandler handler="org.apache.ibatis.type.EnumOrdinalTypeHandler" javaType="java.math.RoundingMode"/>

</typeHandlers>

但是怎样能将同样的 Enum 既映射成字符串又映射成整形呢？

自动映射器（auto-mapper）会自动地选用 EnumOrdinalTypeHandler 来处理，所以如果我们想用回旧的序数型的 EnumTypeHandler，就非要为那些 SQL 语句显式地设置要用到的类型处理器不可。

（下一节才开始讲映射器文件，所以如果是首次阅读该文档，你可能需要先越过这一步，过会再来看。）

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="org.apache.ibatis.submitted.rounding.Mapper">

<resultMap type="org.apache.ibatis.submitted.rounding.User" id="usermap">

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

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

<result column="funkyNumber" property="funkyNumber"/>

<result column="roundingMode" property="roundingMode"/>

</resultMap>

<select id="getUser" resultMap="usermap">

select \* from users

</select>

<insert id="insert">

insert into users (id, name, funkyNumber, roundingMode) values (

#{id}, #{name}, #{funkyNumber}, #{roundingMode}

)

</insert>

<resultMap type="org.apache.ibatis.submitted.rounding.User" id="usermap2">

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

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

<result column="funkyNumber" property="funkyNumber"/>

<result column="roundingMode" property="roundingMode" typeHandler="org.apache.ibatis.type.EnumTypeHandler"/>

</resultMap>

<select id="getUser2" resultMap="usermap2">

select \* from users2

</select>

<insert id="insert2">

insert into users2 (id, name, funkyNumber, roundingMode) values (

#{id}, #{name}, #{funkyNumber}, #{roundingMode, typeHandler=org.apache.ibatis.type.EnumTypeHandler}

)

</insert>

</mapper>

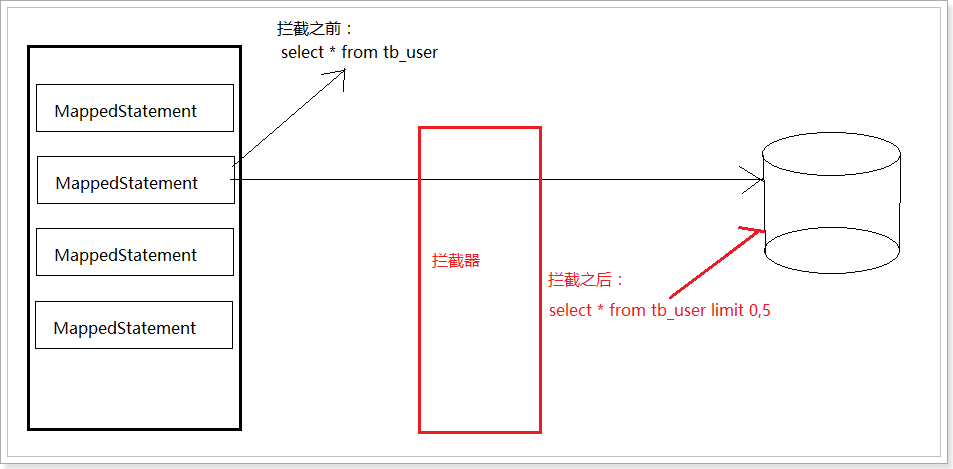
注意，这里的 select 语句强制使用 resultMap 来代替 resultType。

## plugins(插件又称拦截器)

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 允许使用插件来拦截的方法调用包括：

注意: 不要拦截自己定义的方法

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





## environments(环境)

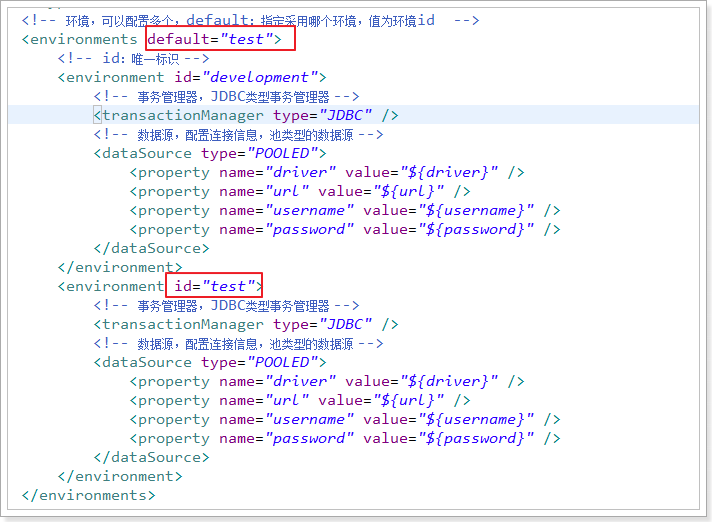
MyBatis 可以配置成适应多种环境，例如，开发、测试和生产环境需要有不同的配置；

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。

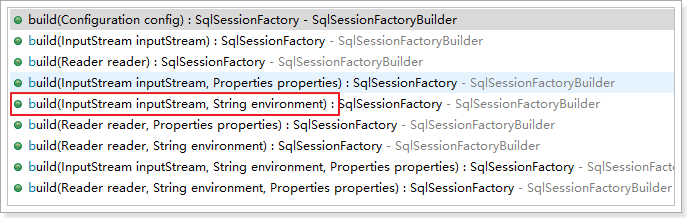
### 方法一：default

添加一个test（测试）环境，并在default参数中指向test环境。

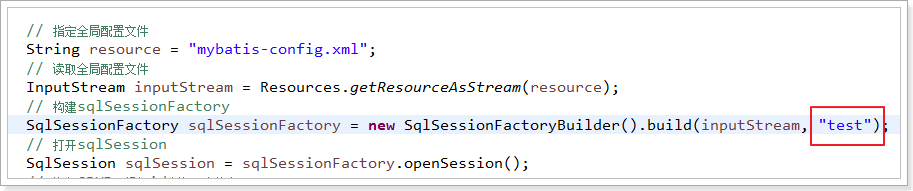


### 方法二：build方法

通过build方法的重载方法



使用：

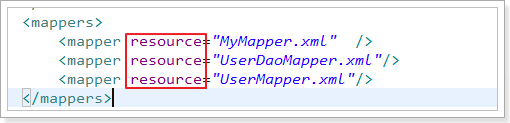


## Mappers

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 : 推荐

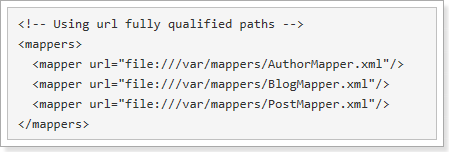
在mybatis-config.xml引入项目目录下的映射文件：



缺点：每次都要在mybatis-config.xml中引入映射文件，麻烦

### 方式二：url /file （不采用）

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

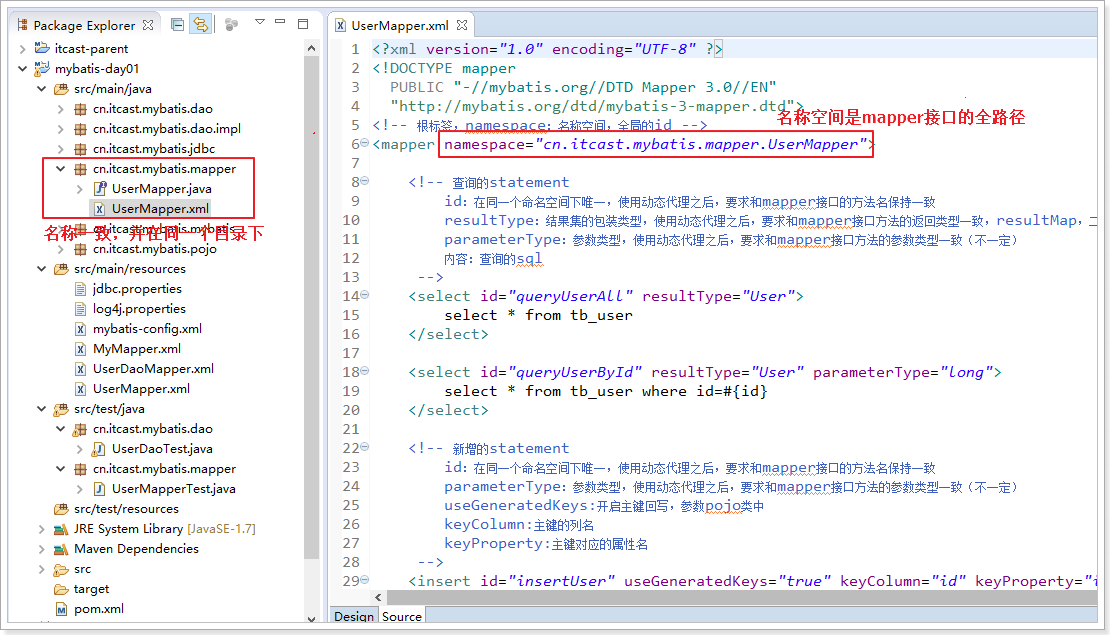
在mybatis-config.xml配置mapper接口的全路径：



这种配置方式，在全局配置文件中配置了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(包下)
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。

缺点：

1、如果包的路径有很多

2、mapper.xml和mapper.java没有分离。

1. properties 引入外部资源文件

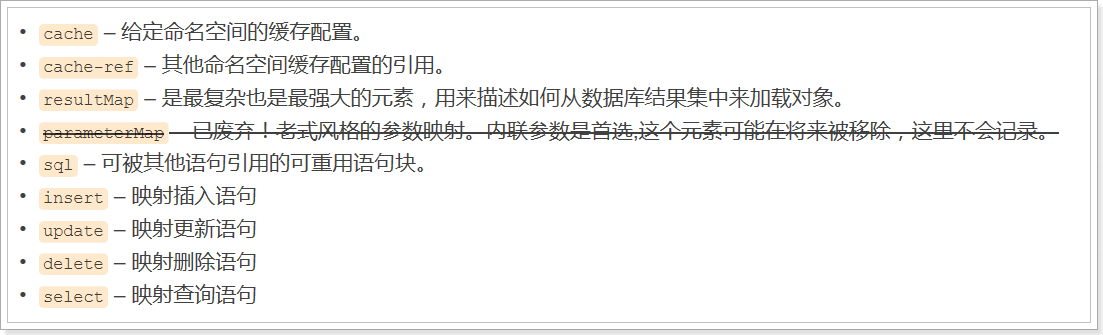
2. Settings 行为参数 (驼峰匹配)

3. typeAliases 类型别名 (package)

4. environments (default | test)

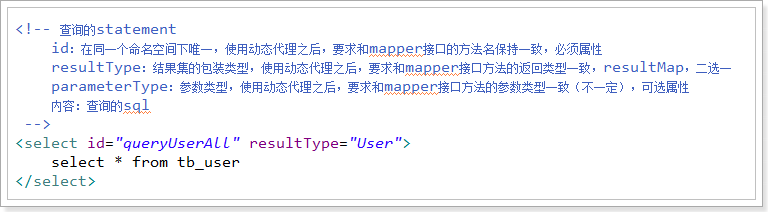
5. mapper xxxMapper.xml的加载

# Mapper XML 文件（映射文件）

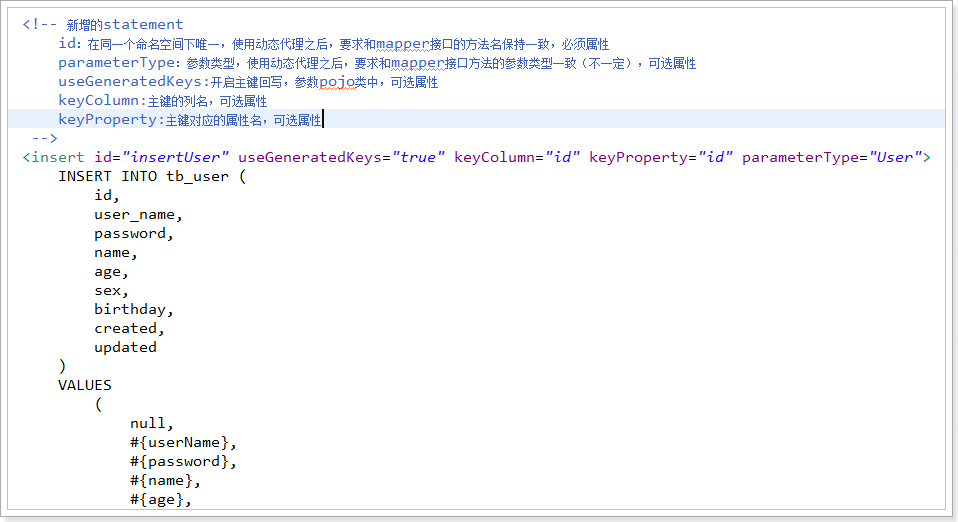


## CRUD标签 select , insert ,update , delete

### select



### insert



注意: 一个有趣的知识点:

1. <insert id="insertUser" useGeneratedKeys="true" keyColumn="id" keyProperty="id" parameterType="cn.sp.mybatis.pojo.User">

User user = new User();

user.setAge(20);

user.setBirthday(new Date());

user.setName("罗玉凤");

user.setPassword("111111");

user.setSex(2);

user.setUserName("luoyufeng");

this.userDelegateMapper.insertUser(user);

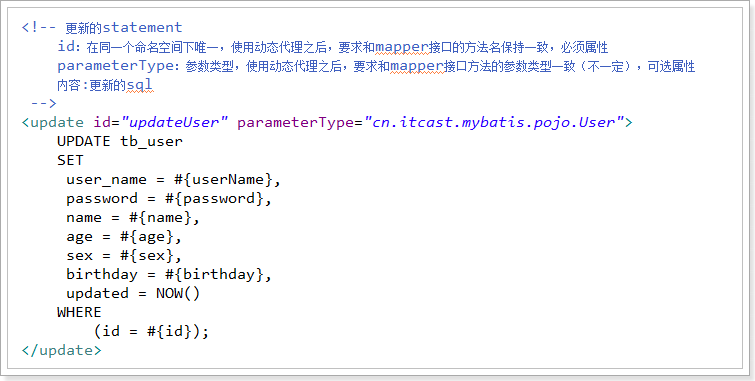
System.out.println(user.getId());// 16

2. <insert id="insertUser" useGeneratedKeys="true" keyColumn="user\_name" keyProperty="userName" parameterType="cn.sp.mybatis.pojo.User">

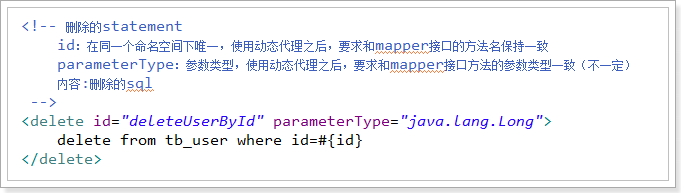
System.out.println(user.getUserName()); // 16 --> 注意 是 user.getUserName() = 16 , 类型改变了

System.out.println(user.getId());//null

### update



### delete



## 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

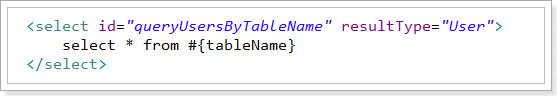
场景：数据库有两个一模一样的表。历史表，当前表

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。

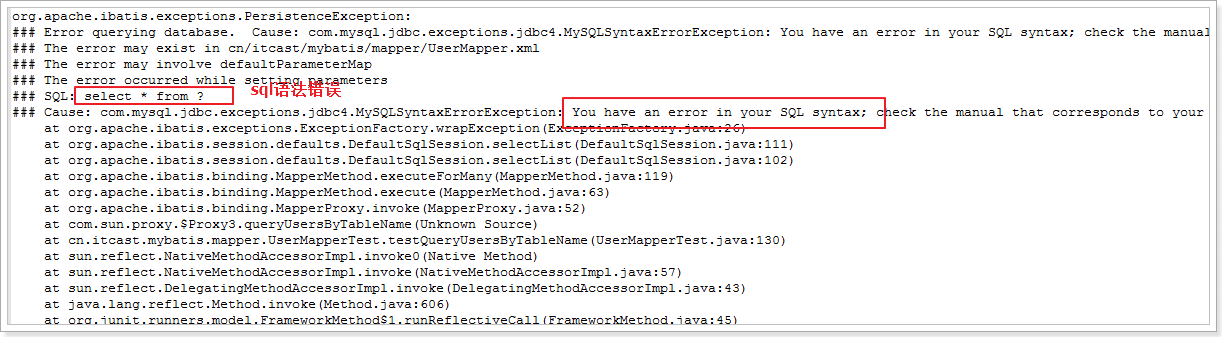
在UserMapper接口中，添加根据表名查询用户信息的方法：



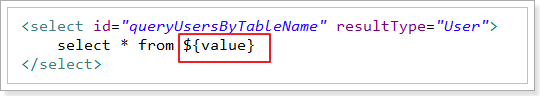
在UserMapper映射文件中，添加方法对应的statement：



输出（报错）：



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，这显然是不行的，所以你必须改成非预编译的，也就是这样：

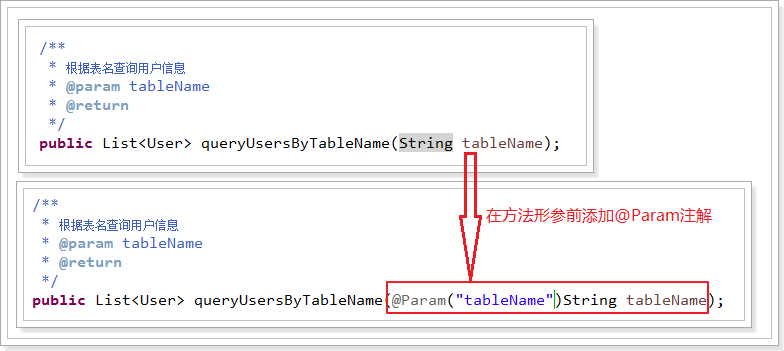


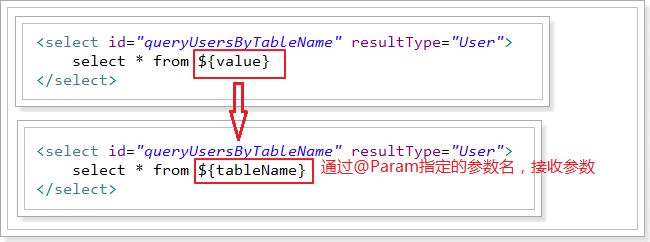
注意：

如果是${tableName} --> error : 没有tableName 这个字段 --> 所以: 1. 使用${} 去接收参数信息，2. 在一个参数时 --> 默认情况下 **必须** 使用${value}获取参数值，

而#{} 只是表示占位，与参数的名字无关，如果只有一个参数，可以使用任意参数名接收参数值，会自动对应。

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，在使用#{}传参时，可以通过任意参数名接收参数； #{xxx},等不会报错

<select id="queryUserById" resultType="cn.sp.mybatis.pojo.User.User">

<!-- select \* from tb\_user where id = #{id} -->

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

</select>

而${}，默认必须通过value来接收参数。 --> 可以通过@Param注解指定参数名

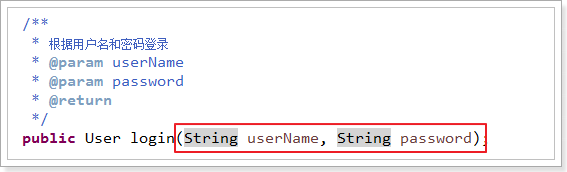
### 多个参数

当mapper接口要传递多个参数时，有两种传递参数的方法：

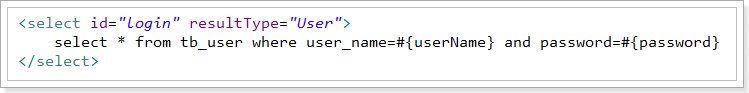
1. 默认规则获取参数{0,1,param1,param2}
2. 使用@Param注解指定参数名

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

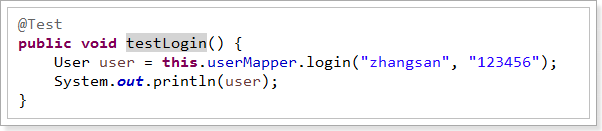
在UserMapper接口中，添加登陆方法：



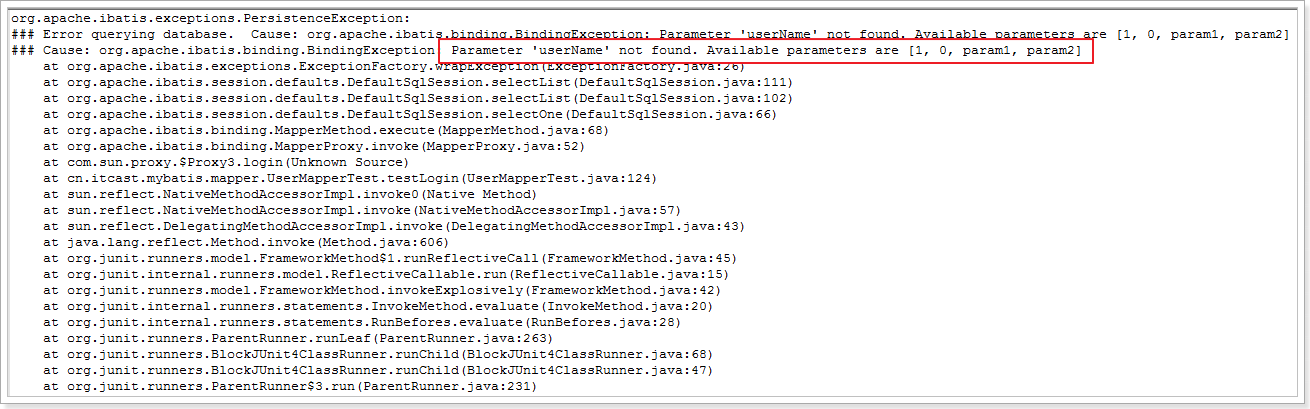
在UserMapper.xml配置中，添加登陆方法对应的Statement配置：



在UserMapperTest测试用例中，添加测试方法：



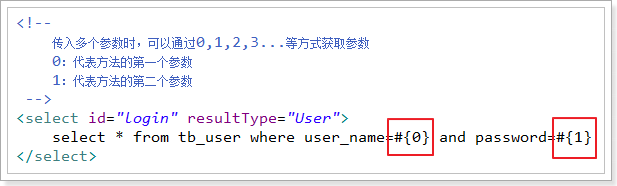
运行报错：



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：

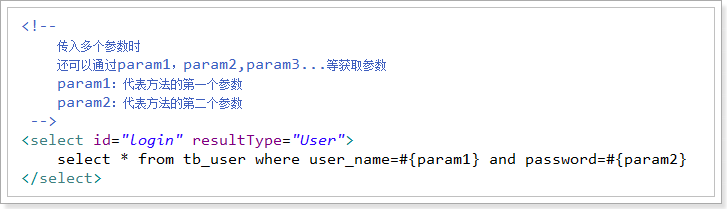
解决方案一：

在映射文件里，通过#{0}，#{1}获取参数



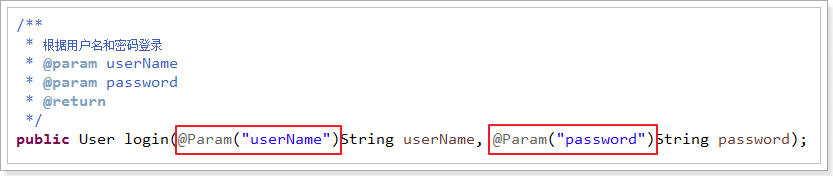
解决方案二：

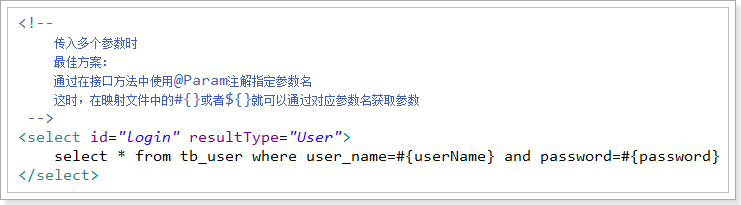
在映射文件里，通过param1，param2获取参数



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。

能够使用 #{ } 尽量使用, 不能使用 则 使用${ } , #{ } ,预编译 --> 提高效率等

### HashMap

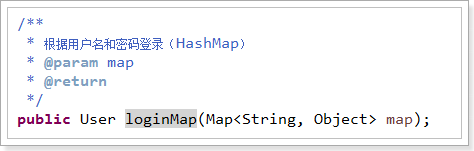
parameterType有三种类型的输入参数：

1. 基本数据类型
2. hashMap
3. pojo包装类

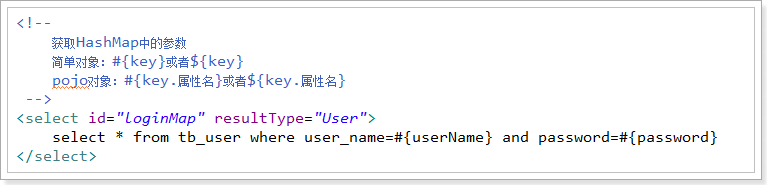
前面已经使用了基本数据类型和pojo类型的参数，那么hashMap这种类型的参数怎么传递参数呢？

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}

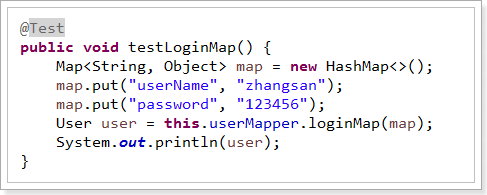
UserMapper接口方法：



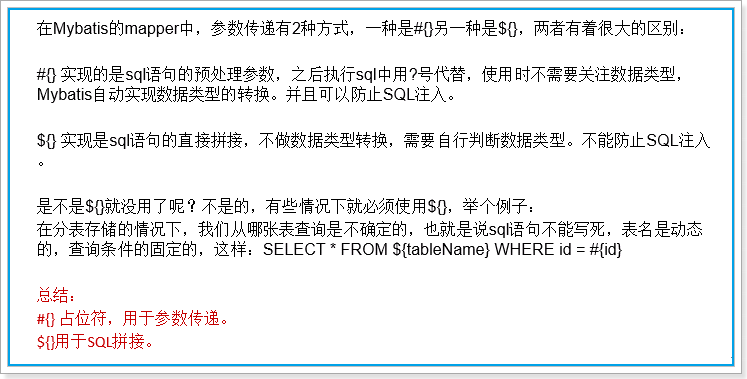
UserMapper配置文件：



测试用例：

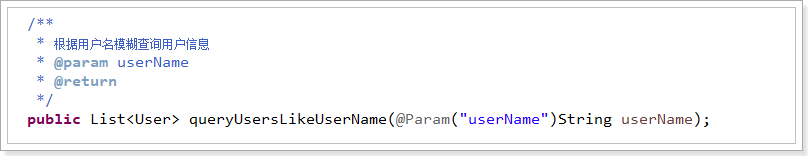


### 面试题（#、$区别）

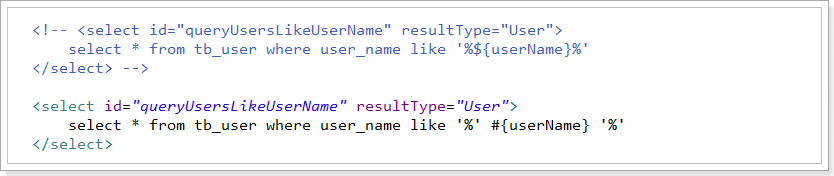


练习：根据用户名进行模糊查询

UserMapper接口：



UserMapper映射文件：



扩展:

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

<!-- 方法1 : -->

<!-- select \* from tb\_user where user\_name like '%${userName}%' -->

<!-- 方法2: -->

<!-- select \* from tb\_user where user\_name like '%' ? '%' -->

<!-- error : select \* from tb\_user where user\_name like '%'?'%' ? 两边必须有空格 -->

<!-- select \* from tb\_user where user\_name like '%' #{userName} '%' -->

<!-- 方法3: -->

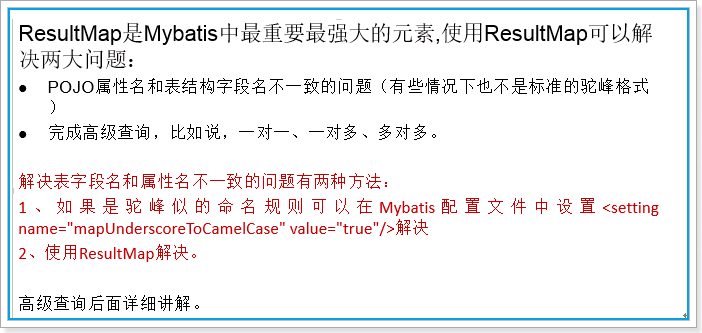
select \* from tb\_user where user\_name like concat ('%',#{userName} ,'%' )

</select>

#{} 预编译 编译成占位符？ 防止sql注入 不需要关注数据类型 一个参数时，可以使用任意参数名接收参数

${} 非预编译 直接的字符串的拼接 不可以防止sql注入 需要关注数据类型 一个参数，必须使用value接收参数 @Param

## resultMap



### 解决列名和属性名不一致

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

解决方案3：resultMap自定义映射

在UserMapper.xml中配置resultMap



在UserMapper.xml中使用resultMap：



### 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，字段名称匹配的字段要不要配置那？

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。

为false时：只针对resultMap中已经配置的字段作映射。

并且resultMap会自动映射单表查询的结果集

## sql片段

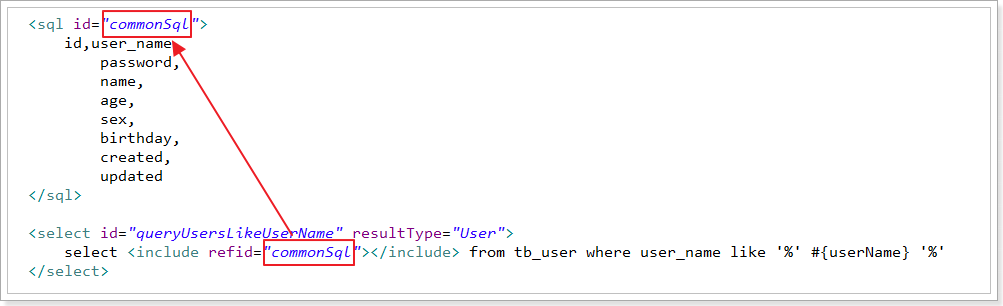
在java代码中，为了提高代码的重用性，针对一些出现频率较高的代码，抽离出来一个共同的方法或者类

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？

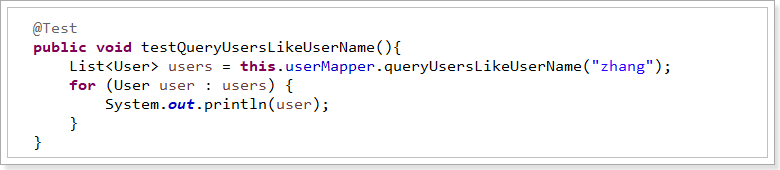
Mybatis当然想到了这一点，它就是sql标签。

### 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，如改造根据表名查询用户信息：



测试：



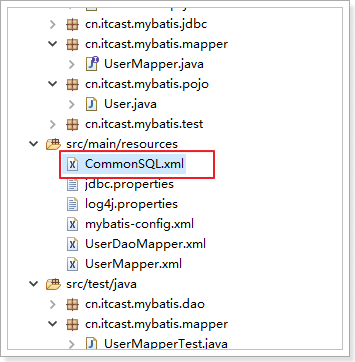
测试结果：

|  |
| --- |
| 2017-07-26 10:18:08,622 [main] [org.apache.ibatis.datasource.pooled.PooledDataSource]-[DEBUG] PooledDataSource forcefully closed/removed all connections.  2017-07-26 10:18:08,622 [main] [org.apache.ibatis.datasource.pooled.PooledDataSource]-[DEBUG] PooledDataSource forcefully closed/removed all connections.  2017-07-26 10:18:08,622 [main] [org.apache.ibatis.datasource.pooled.PooledDataSource]-[DEBUG] PooledDataSource forcefully closed/removed all connections.  2017-07-26 10:18:08,854 [main] [org.apache.ibatis.transaction.jdbc.JdbcTransaction]-[DEBUG] Opening JDBC Connection  2017-07-26 10:18:09,130 [main] [org.apache.ibatis.datasource.pooled.PooledDataSource]-[DEBUG] Created connection 802102645.  2017-07-26 10:18:09,132 [main] [cn.itcast.mybatis.mapper.UserMapper.queryUsersLikeUserName]-[DEBUG] ==> Preparing: select id,user\_name, password, name, age, sex, birthday, created, updated from tb\_user where user\_name like '%' ? '%'  2017-07-26 10:18:09,185 [main] [cn.itcast.mybatis.mapper.UserMapper.queryUsersLikeUserName]-[DEBUG] ==> Parameters: zhang(String)  2017-07-26 10:18:09,213 [main] [cn.itcast.mybatis.mapper.UserMapper.queryUsersLikeUserName]-[DEBUG] <== Total: 2  User [id=1, userName=zhangsan, password=123456, name=张三, age=30, sex=1, birthday=Wed Aug 08 00:00:00 CST 1984, created=Fri Sep 19 16:56:04 CST 2014, updated=Sun Sep 21 11:24:59 CST 2014]  User [id=4, userName=zhangwei, password=123456, name=张伟, age=20, sex=1, birthday=Thu Sep 01 00:00:00 CDT 1988, created=Fri Sep 19 16:56:04 CST 2014, updated=Fri Sep 19 16:56:04 CST 2014] |

### 用法二

很多时候同一个sql片段，可能在很多映射文件里都有使用，这就需要添加一个映射文件，用来统一定义sql片段。

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">

<!-- namespace(命名空间)：映射文件的唯一标识 -->

<mapper namespace=*"CommonSQL"*>

<sql id=*"commonSql"*>

id,user\_name,

password,

name,

age,

sex,

birthday,

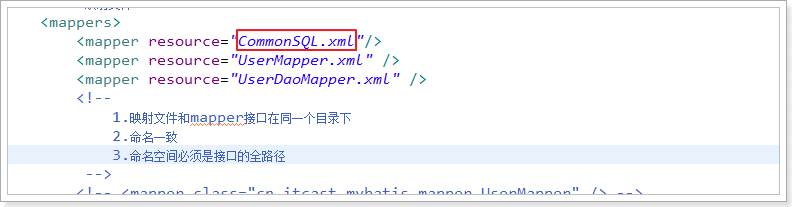
created,

updated

</sql>

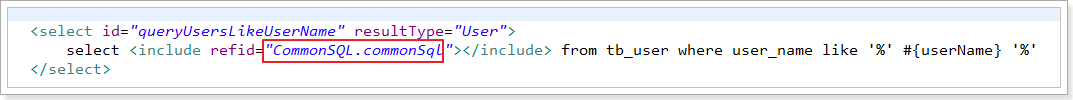
</mapper>

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）：

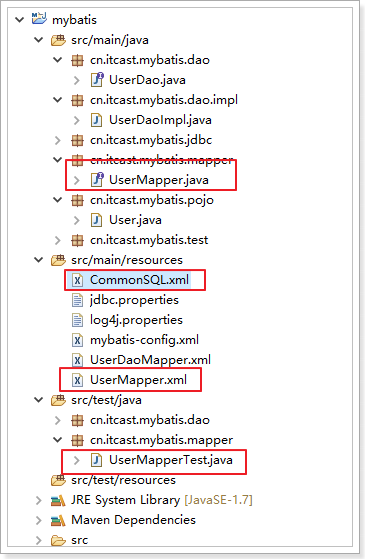


最后在需要使用该sql片段的地方通过include标签的refId属性引用该sql片段：<include refId=”名称空间.sql片段的id” />

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



## 本章代码汇总



### UserMapper.java

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

/\*\*

\* 根据用户名模糊查询用户信息

\* **@param** userName

\* **@return**

\*/

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

/\*\*

\* 登陆（Map的方式）

\* **@param** map

\* **@return**

\*/

**public** User loginByMap(Map<String,Object> map);

/\*\*

\* 根据用户名和密码登陆

\* **@param** userName

\* **@param** password

\* **@return**

\*/

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

/\*\*

\* 根据表名查询用户信息

\* **@param** tableName

\* **@return**

\*/

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

/\*\*

\* 根据id查询用户信息

\* **@param** id

\* **@return**

\*/

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

/\*\*

\* 查询所有用户信息

\* **@return**

\*/

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

/\*\*

\* 新增用户信息

\* **@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);

}

### UserMapper.xml

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- namespace(命名空间)：映射文件的唯一标识 -->

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

<!-- <sql id="commonSql">

id,user\_name,

password,

name,

age,

sex,

birthday,

created,

updated

</sql> -->

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

select <include refid=*"CommonSQL.commonSql"*></include> from tb\_user where user\_name like '%' #{userName} '%'

</select>

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

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

</select>

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

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

</select>

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

select \* from ${tableName}

</select>

<!-- resultMap:自定义映射关系

属性：type-结果集的封装类型，id-唯一标识，autoMapping-开启自动匹配，如果开启了驼峰匹配，就以驼峰匹配的形式进行匹配

id:指定主键映射的，不要省。提高性能

result：其他的非主键普通字段

子标签的属性：Column-表中的字段名，property-对应的java属性名

-->

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

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

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

</resultMap>

<!-- statement：查询的statement

id:在该映射文件下的唯一标识。在使用动态代理之后，必须和接口的方法名一致。必须属性

resultType:结果集的映射类型。在使用动态里之后，必须和接口方法的返回值类型一致。必须属性

parameterType:参数类型。可省略

-->

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

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

</select>

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

select \* from tb\_user

</select>

<!-- 插入的statement

id:在该映射文件下的唯一标识。在使用动态代理之后，必须和接口的方法名一致。必须属性

parameterType:参数类型。可省略

useGeneratedKeys:开启主键回写,回写到参数的pojo对象里

keyColumn:主键列名

keyProperty：主键对应的属性名

-->

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

INSERT INTO tb\_user (

user\_name,

password,

name,

age,

sex,

birthday,

created,

updated

)

VALUES

(

#{userName},

#{password},

#{name},

#{age},

#{sex},

#{birthday},

now(),

now()

);

</insert>

<!-- 更新的statement

id：在该映射文件下的唯一标识。在使用动态代理之后，必须和接口的方法名一致。必须属性

parameterType:参数类型。可省略

-->

<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>

<!-- 删除的statement

id：在该映射文件下的唯一标识。在使用动态代理之后，必须和接口的方法名一致。必须属性

parameterType:参数类型。可省略

-->

<delete id=*"deleteUserById"*>

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

</delete>

</mapper>

### CommonSQL.xml

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- namespace(命名空间)：映射文件的唯一标识 -->

<mapper namespace=*"CommonSQL"*>

<sql id=*"commonSql"*>

id,user\_name,

password,

name,

age,

sex,

birthday,

created,

updated

</sql>

</mapper>

### Mybatis-config.xml

<?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：相对路径，url：绝对路径 -->

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

<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"*/>

</settings>

<typeAliases>

<!-- 类型别名：type：类型的全路径，alias：别名名称，推荐和class类名一致 -->

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

<!-- 别名扫描，name:包的路径 -->

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

</typeAliases>

<!-- 环境：说明可以配置多个，default:指定生效的环境 -->

<environments default=*"test"*>

<!-- id:环境的唯一标识 -->

<environment id=*"development"*>

<!-- 事务管理器，type：类型 -->

<transactionManager type=*"JDBC"* />

<!-- 数据源：type-池类型的数据源 -->

<dataSource type=*"POOLED"*>

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

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

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

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

</dataSource>

</environment>

<environment id=*"test"*>

<!-- 事务管理器，type：类型 -->

<transactionManager type=*"JDBC"* />

<!-- 数据源：type-池类型的数据源 -->

<dataSource type=*"POOLED"*>

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

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

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

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

</dataSource>

</environment>

</environments>

<!-- 映射文件 -->

<mappers>

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

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

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

<!--

1.映射文件和mapper接口在同一个目录下

2.命名一致

3.命名空间必须是接口的全路径

-->

<!-- <mapper class="cn.itcast.mybatis.mapper.UserMapper" /> -->

<!-- <package name="cn.itcast.mybatis.mapper"/> -->

</mappers>

</configuration>

### UserMapperTest

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

**private** UserMapper userMapper;

@Before

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

String resource = "mybatis-config.xml";

// 读取配置文件

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

// 构建sqlSessionFactory

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

// 获取sqlSession

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

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

}

@Test

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

List<User> users = **this**.userMapper.queryUsersLikeUserName("zhang");

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

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

}

}

@Test

**public** **void** testMap(){

Map<String,Object> map = **new** HashMap<>();

map.put("userName", "zhangsan");

map.put("password", "123456");

System.***out***.println(**this**.userMapper.loginByMap(map));

}

@Test

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

System.***out***.println(**this**.userMapper.login("zhangsan", "123456"));

}

@Test

**public** **void** testQueryUsersByTableName(){

List<User> users = **this**.userMapper.queryUsersByTableName("tb\_user");

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

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

}

}

@Test

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

System.***out***.println(**this**.userMapper.queryUserById(1l));

}

@Test

**public** **void** testQueryUserAll() {

}

@Test

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

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

user.setAge(18);

user.setName("柳岩");

user.setPassword("123456");

user.setUserName("yanyan1");

user.setSex(3);

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

**this**.userMapper.insertUser(user);

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

}

@Test

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

}

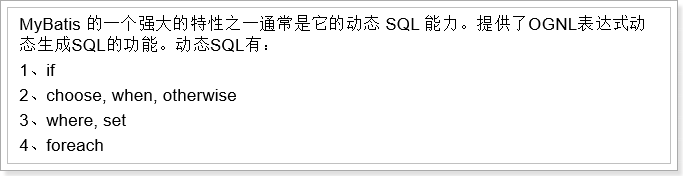
@Test

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

}

}

# 动态sql



## if

判断语句

案例：查询男性用户，如果输入了用户名，按用户名模糊查询

### 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> queryUserListLikeUserName(@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=*"queryUserListLikeUserName"* resultType=*"User"*>

select \* from tb\_user where sex=1

<!-- if:判断

test：OGNL表达式

-->

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

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

</if>

</select>

### 测试

在UserMapperTest测试类中，添加测试用例

@Test

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

List<User> users = **this**.userMapper.queryUserListLikeUserName("zhang");

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

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

}

}

## choose when otherwise

查询男性用户，如果输入了用户名则按照用户名模糊查找，否则如果输入了年龄则按照年龄查找，否则查找用户名为“zhangsan”的用户。

### 定义接口

在UserMapper接口中，定义接口方法：

/\*\*

\* 查询男性用户，如果输入了用户名则按照用户名模糊查找，否则如果输入了年龄则按照年龄查找，否则查找用户名为“zhangsan”的用户。

\* **@param** userName

\* **@param** age

\* **@return**

\*/

**public** List<User> queryUserListLikeUserNameOrAge(@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

Sql: select \* from tb\_user where

and user\_name like *'*%zhang%*'*

*and age < 30*

*and user\_name = 'z*hangsan*'*

*Sql中的 and 是 与的连接*

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

select \* from tb\_user where sex=1

<!-- choose:条件选择

when：test-判断条件，一旦有一个when成立，后续的when都不再执行

otherwise：所有的when都不成立时，才会执行

注意: < 是 非法字符 可以用 &lt; 代替 或 可以用 > 代替 或

-->

<choose>

<when test=*"userName!=null and userName.trim()!=''"*>and user\_name like '%' #{userName} '%'</when>

<when test=*"age != null"*>and age = #{age}</when>

<otherwise>and user\_name = 'zhangsan' </otherwise>

</choose>

</select>

### 测试

在UserMapperTest测试类中，添加测试用例

@Test

**public** **void** testQueryUserListLikeUserNameOrAge(){

List<User> users = **this**.userMapper.queryUserListLikeUserNameOrAge(**null**, **null**);

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

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

}

}

## where

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，按照年龄进行查询，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：

/\*\*

\* 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，按照年龄进行查询，如果两者都输入，两个条件都要成立。

\* **@param** userName

\* **@param** age

\* **@return**

\*/

**public** List<User> queryUserListLikeUserNameAndAge(@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

Sql: select \* from tb\_user where 1=1

and user\_name like *'*%zhang%*'*

*and age < 30*

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

<!-- 第一种方案 -->

<!--

select \* from tb\_user where 1=1

<if test=*"userName!=null and userName.trim()!=''"*>user\_name like '%' #{userName} '%'</if>

<if test=*"age!=null"*>and age = #{age}</if>

-->

<!-- 第二种方案 -->

select \* from tb\_user

<!--

自动添加where关键字

有一定的纠错功能：去掉sql语句块之前多余的一个and 或 or , 只能去掉一个 多余的and 或 or , 两个 and and 就无法成功

通常结合if或者choose使用

-->

<where>

<if test=*"userName!=null and userName.trim()!=''"*>user\_name like '%' #{userName} '%'</if>

<if test=*"age!=null"*>and age = #{age}</if>

</where>

</select>

### 测试

在UserMapperTest测试类中，添加测试用例

@Test

**public** **void** testQueryUserListLikeUserNameAndAge(){

List<User> users = **this**.userMapper.queryUserListLikeUserNameAndAge(**null**, 30);

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

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

}

}

## set

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。

### 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"* >

UPDATE tb\_user

<!--

set自动添加set关键字

也有一定的纠错功能：自动去掉sql语句块之后多余的一个逗号

-->

<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>

<if test=*"sex!=null"*>sex = #{sex}</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.setAge(18);

user.setName("柳岩");

user.setPassword("123456");

user.setUserName("yanyan2");

// user.setSex(3);

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

user.setId(12l);

**this**.userMapper.updateUserSelective(user);

}

## foreach

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

Sql: select \* from tb\_user where id in(1,2,3,4)

查询的结果是 4个 , 包含了 id= 1,2,3,4 的结果集

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

select \* from tb\_user where id in

<!-- 传递的是数组

foreach:遍历集合

collection：接收的集合参数

item：遍历的集合中的一个元素

separator:分隔符

open:以什么开始

close：以什么结束

-->

<!-- 写法1 -->

<!--

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

#{id}

</foreach>)

-->

<!-- 写法2 -->

<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 = **this**.userMapper.queryUserListByIds(**new** Long[]{1l,2l,3l,4l});

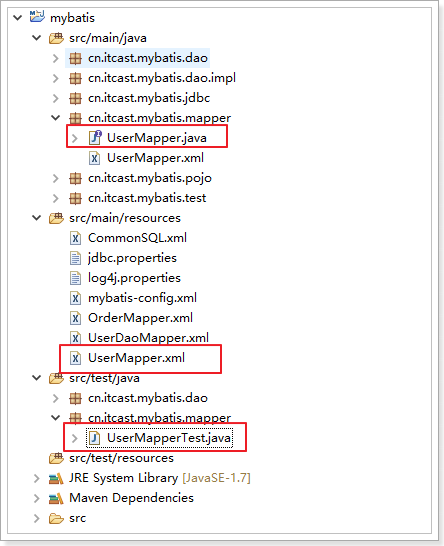
**for** (User user : users) {

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

}

}

## 本章代码汇总



### UserMapper.xml

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

select \* from tb\_user where sex &lt; 2

<!-- if:判断

test：OGNL表达式或者简单的java代码

-->

<if test=*"userName!=null &amp;&amp; !''.equals(userName)"*>

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

</if>

</select>

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

select \* from tb\_user where sex=1

<!-- choose:条件选择

when：test-条件，使用方式参考if的test属性，一旦有一个成立，后续的when都不再执行

otherwise：所有的when都不成立时，才会执行

-->

<choose>

<when test=*"userName!=null and userName.trim()!=''"*>and user\_name like '%' #{userName} '%'</when>

<when test=*"age != null"*>and age = #{age}</when>

<otherwise>and user\_name = 'zhangsan' </otherwise>

</choose>

</select>

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

select \* from tb\_user

<!--

自动添加where关键字

有一定的纠错功能：去掉sql语句块之前多余的一个and|or

通常结合if或者choose使用

-->

<where>

<if test=*"userName!=null and userName.trim()!=''"*>user\_name like '%' #{userName} '%'</if>

<if test=*"age!=null"*>and age = #{age}</if>

</where>

</select>

<update id=*"updateUserSelective"* >

UPDATE tb\_user

<!--

set自动添加set关键字

也有一定的纠错功能：自动去掉sql语句块之后多余的一个逗号

-->

<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>

<if test=*"sex!=null"*>sex = #{sex}</if>

</set>

WHERE

(id = #{id});

</update>

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

select \* from tb\_user where id in

<!--

foreach:遍历集合

collection：接收的集合参数

item：遍历的集合中的一个元素

separator: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>

### UserMapper.java

/\*\*

\* 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> queryUserListLikeUserName(@Param("userName")String userName);

/\*\*

\* 查询男性用户，如果输入了姓名则按照姓名模糊查找，否则如果输入了年龄则按照年龄查找，否则查找姓名为“张三”的用户。

\* **@param** userName

\* **@param** age

\* **@return**

\*/

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

/\*\*

\* 查询所有用户，如果输入了姓名按照姓名进行模糊查询，如果输入年龄，按照年龄进行查询，如果两者都输入，两个条件都要成立。

\* **@param** userName

\* **@param** age

\* **@return**

\*/

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

/\*\*

\* 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);

/\*\*

\* 根据多个id查询用户信息

\* **@param** ids

\* **@return**

\*/

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

### UserMapperTest.java

@Test

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

List<User> users = **this**.userMapper.queryUserListByIds(**new** Long[]{1l,2l,3l,4l});

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

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

}

}

@Test

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

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

user.setAge(18);

user.setName("柳岩");

user.setPassword("123456");

user.setUserName("yanyan2");

// user.setSex(3);

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

user.setId(12l);

**this**.userMapper.updateUserSelective(user);

}

@Test

**public** **void** testQueryUserListLikeUserNameAndAge(){

List<User> users = **this**.userMapper.queryUserListLikeUserNameAndAge(**null**, 30);

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

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

}

}

@Test

**public** **void** testQueryUserListLikeUserNameOrAge(){

List<User> users = **this**.userMapper.queryUserListLikeUserNameOrAge(**null**, **null**);

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

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

}

}

@Test

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

List<User> users = **this**.userMapper.queryUserListLikeUserName("zhang");

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

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

}

}

# 缓存

执行相同的sql语句和参数，mybatis不进行执行sql，而是从缓存中命中返回。

## 一级缓存

在mybatis中，一级缓存默认是开启的，并且一直无法关闭，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** testCache(){

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

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

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

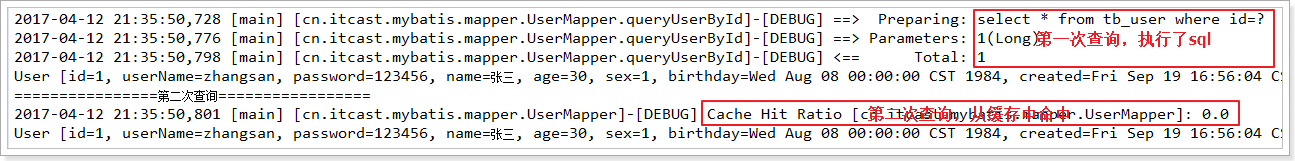
User user2 = **this**.userMapper.queryUserById(1l);

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：



使用：sqlSession.clearCache();可以强制清除缓存

在测试方法中清空一级缓存：

@Test

**public** **void** testCache(){

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

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

**this**.sqlSession.clearCache();

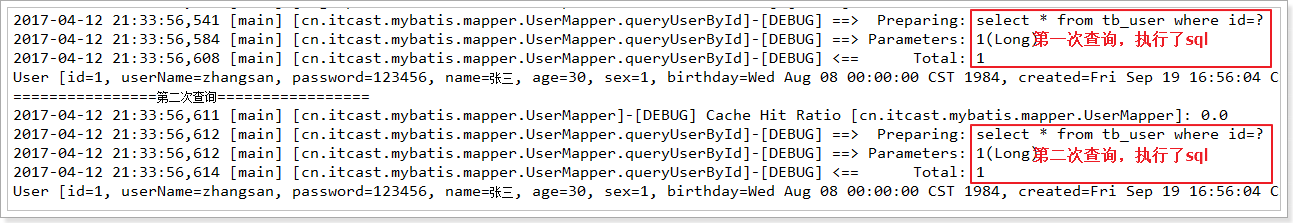
System.***out***.println("=================第二次查询======================");

User user2 = **this**.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。



执行update、insert、delete的时候，会清空缓存

在测试方法中添加更新的操作：

@Test

**public** **void** testCache(){

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

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

// this.sqlSession.clearCache();

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

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

user.setAge(18);

user.setName("柳岩");

user.setPassword("123456");

user.setUserName("yanyan2");

// user.setSex(3);

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

user.setId(12l);

**this**.userMapper.updateUser (user);

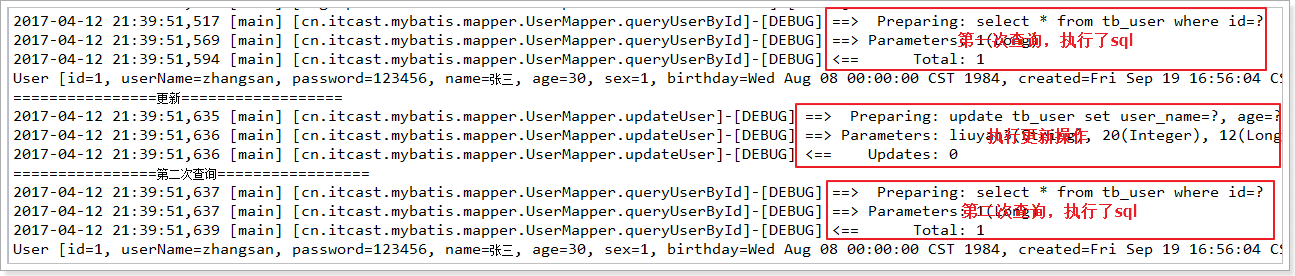
System.***out***.println("=================第二次查询======================");

User user2 = **this**.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。



## 二级缓存

mybatis 的二级缓存的作用域： 视频: mybatis-day2 13

1. 同一个mapper的namespace，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

怎么开启二级缓存：

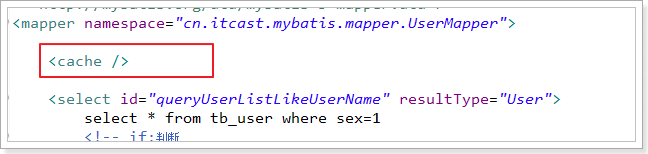
1. 在映射文件中，添加<cache />标签
2. 在全局配置文件中，设置cacheEnabled参数，默认已开启。

注意：

由于缓存数据是在sqlSession调用close方法时，放入二级缓存的，所以第一个sqlSession必须先关闭

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

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



在UserMapperTest中添加二级缓存的测试方法：

@Test

**public** **void** testCache2(){

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

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

// 注意：关闭第一个sqlSession ,不关闭则执行两次 , 因为第二次是通过 sqlSession2 取来的,不是通过二级缓存取来的

// 因为 当sqlSession调用close方法时，才放入二级缓存的

sqlSession.close();

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

// 重新打开一个sqlSession会话,这样userMapper 就是一个新的 userMapper

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

// 通过sqlSession2重新实例化UserMapper

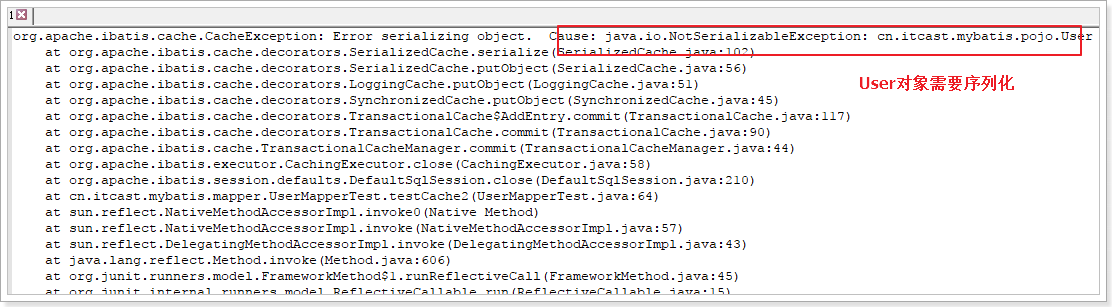
**this**.userMapper = sqlSession2.getMapper(UserMapper.**class**);

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

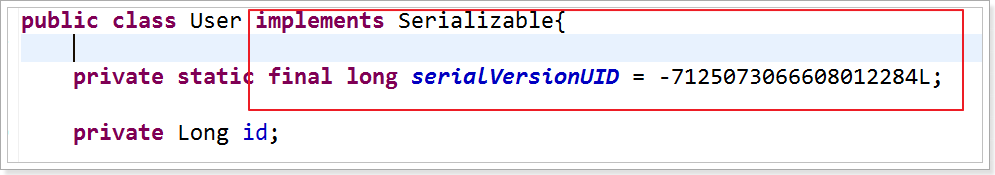
System.***out***.println(user2);

}

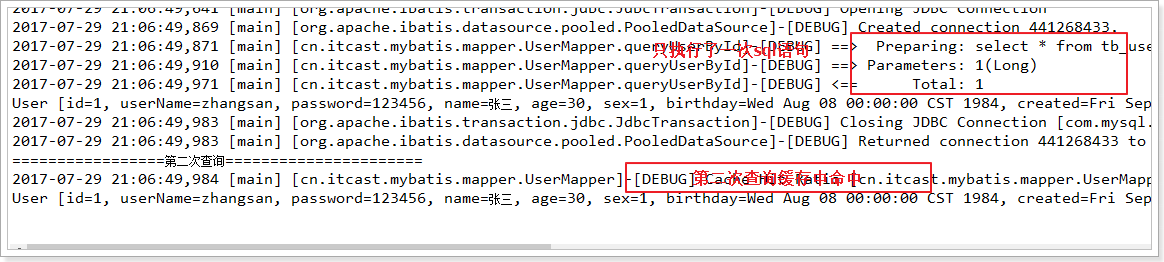
运行测试用例：



说明二级缓存，必须序列化，使User类实现序列化接口：



给User对象实现序列化接口后，重新运行测试用例，日志：



执行update、insert、delete的时候，会清空缓存

@Test

**public** **void** testCache2(){

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

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

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

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

user.setAge(18);

user.setName("柳岩");

user.setPassword("123456");

user.setUserName("yanyan2");

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

user.setId(12l);

**this**.userMapper.updateUserSelective(user);

// 注意：关闭sqlSession

sqlSession.close();

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

// 重新打开一个sqlSession会话

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

// 通过sqlSession2重新实例化UserMapper

**this**.userMapper = sqlSession2.getMapper(UserMapper.**class**);

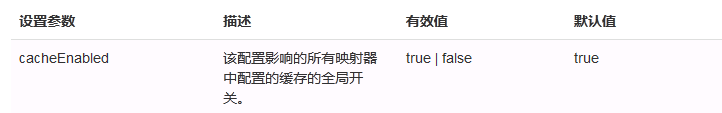
User user2 = **this**.userMapper.queryUserById(1l);

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

}

**关闭二级缓存：**

不开启，或者在全局的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 -->

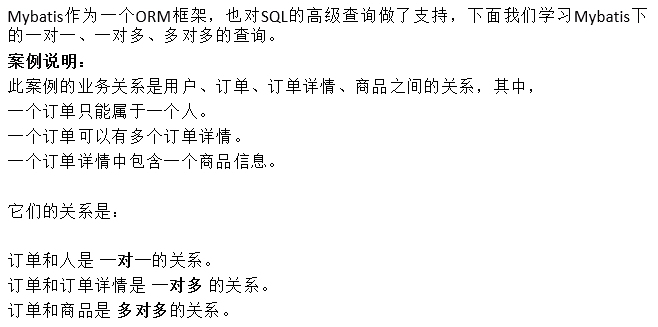
<setting name=*"cacheEnabled"* value=*"false"*/>

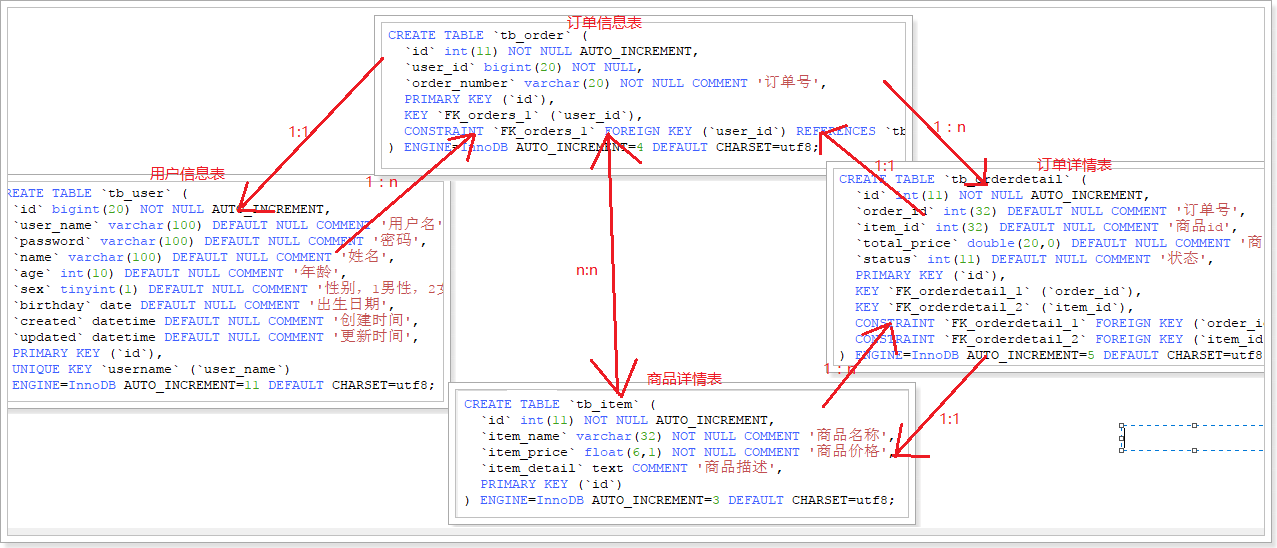
</settings>



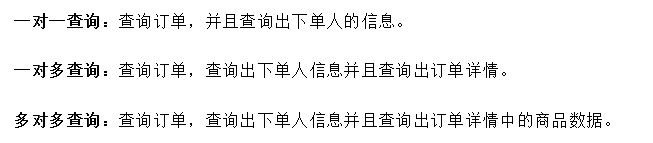
# 高级查询(OrderMapper.xml)

## 表关系说明

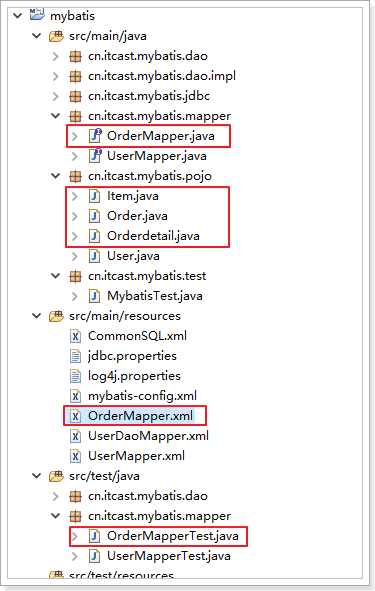




需求说明：



目录结构：

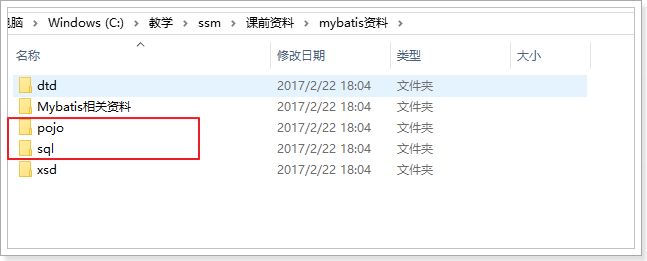


创建OrderMapper.java接口，参考UserMapper.java。

添加OrderMapper.xml映射文件，参考UserMapper.xml。

创建OrderMapperTest测试用例，参考UserMapperTest。

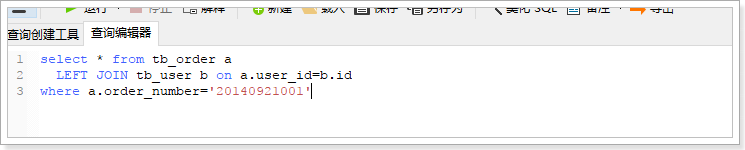
Pojo参考课前资料：



## 一对一查询

需求：查询出订单信息，并查询出下单人信息

Sql：



**Order.java**

private User user; // 下单人

**OrderMapper.java**

public interface OrderMapper {

/\*\*

\* 根据订单号 , 查询出订单信息，并查询出下单人信息

\* @param number

\* @return

\*/

public Order queryOrderWithUser(@Param("number")String number);

}

**mybatis-config.xml**

<mappers>

<mapper resource="OrderMapper.xml"/>

</mappers>

**OrderMapper.xml**

<!-- mybatis-config.xml的配置

1. 开启了package扫描 :resultType

2. 开启了驼峰匹配 : autoMapping

3. 没有autoMapping 那么 没有映射的列,就不会自动匹配 , 那么出来的数据就是null

-->

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

<!-- id : Order 表和pojo的 对应的 id名 -->

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

<!-- 1 vs 1

javaType="User" : 开启了 package扫描

-->

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

<!-- column 1 vs 1 的关联表中对应的 user的id的列名

(可能不止一个 在这里有 user\_id 和 id1 这两个,但是user\_id形象比较好,id1可能会变,因为在前面有一个id那是order的id)

property : user的pojo对应的id的名

-->

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

</association>

</resultMap>

<select id="queryOrderWithUser" resultMap="orderUserMap">

select \* from tb\_order o

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

where o.order\_number = #{number}

</select>

**OrderMapperTest.java**

@Test

public void testQueryOrderWithUser() {

Order order = this.orderMapper.queryOrderWithUser("20140921001");

System.out.println(order);

}

### 扩展Order类的实现方式（了解）

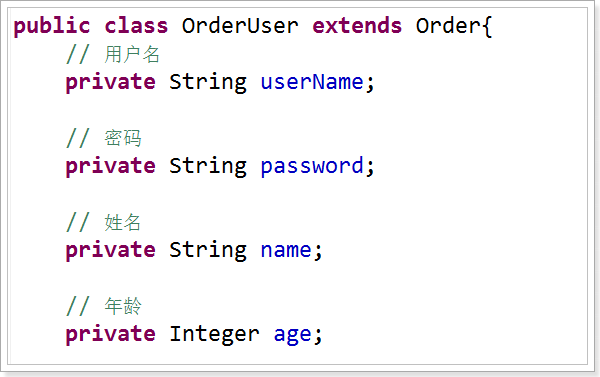
思路：

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。

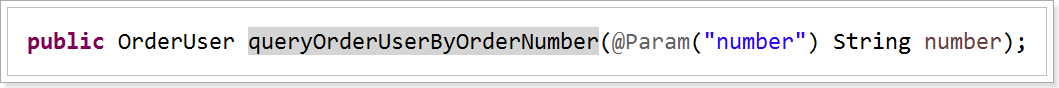
再把User对象中的属性copy到OrderUser对象中，这样OrderUser又有了User中的用户信息。

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。

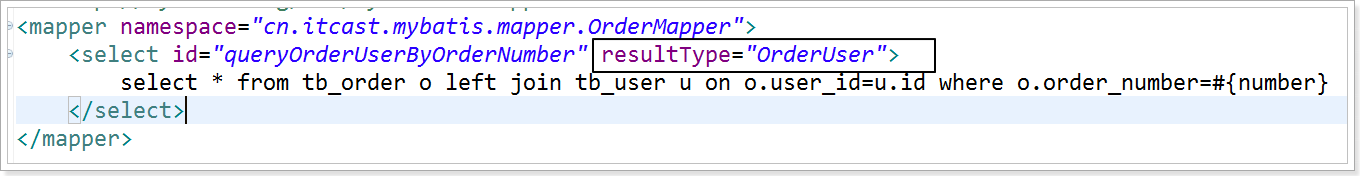
新建OrderUser实体类继承Order，在把User中的属性copy到OrderUser中，并添加get、set方法：



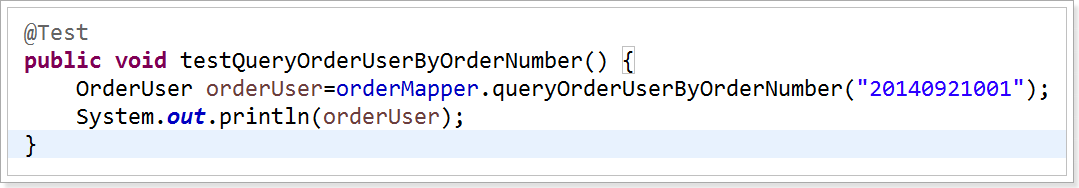
在OrderMapper接口中，添加查询方法：



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



在OrderMapperTest中添加测试方法：



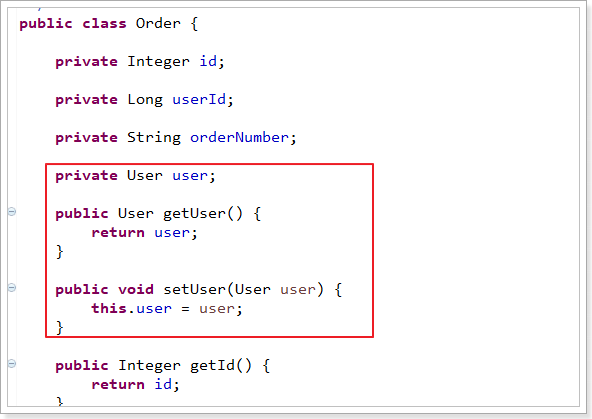
### 添加User属性的实现方式

思路：

从订单的角度来看，订单和用户的关系是一对一的关系，并且通过tb\_order表中的id进行关联。

用面向对象的思想实现，就是在Order对象中添加user属性。

在Order对象中添加User属性，并添加user的get、set方法：



在OrderMapper接口中，添加接口方法：

/\*\*

\* 根据订单号查询订单信息，并且查询出下单人信息

\* **@param** number

\* **@return**

\*/

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

使用resultType不能完成user对象的自动映射，需要手动完成结果集映射，即使用resultMap标签自定义映射。

在OrderMapper.xml中配置，结果集的映射，这时必须使用resultMap：

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

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

<!--

association:一对一的映射

property：java的属性名

javaType：属性名对应的java类型

autoMapping:开启自动映射

子标签：参照resultMap

-->

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

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

</association>

</resultMap>

<!-- resultType不能完成user信息的映射，必须使用resultMap，resultMap的值对应resultMap标签的id，resultMap和resultType必须二选一 -->

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

select \* from tb\_order a

LEFT JOIN tb\_user b on a.user\_id=b.id

where a.order\_number = #{number}

</select>

在OrderMapperTest添加测试：

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

**private** OrderMapper orderMapper;

@Before

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

String resource = "mybatis-config.xml";

// 读取配置文件

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

// 构建sqlSessionFactory

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

// 获取sqlSession

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

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

}

@Test

**public** **void** testQueryOrderWithUser() {

System.***out***.println(**this**.orderMapper.queryOrderWithUser("20140921001"));

}

}

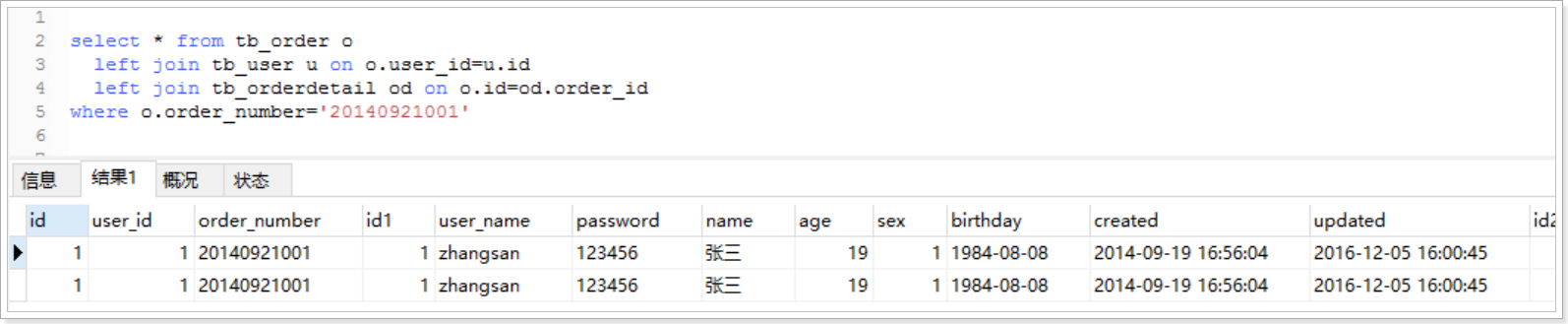
## 一对多查询

**一对多查询：**查询订单，查询出下单人信息并且查询出订单详情。

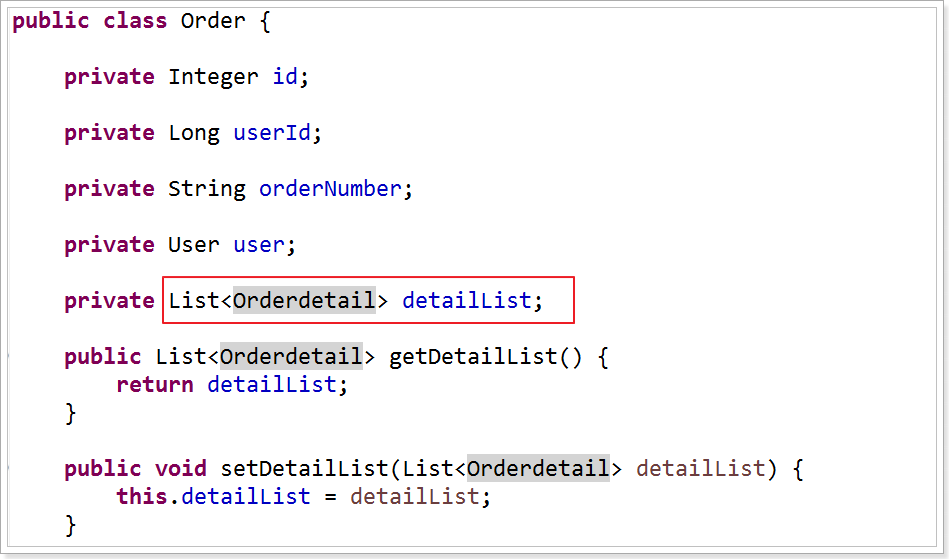
思路：

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：



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



OrderMapper接口：

/\*\*

\* 查询订单，查询出下单人信息并且查询出订单详情。

\* **@param** number

\* **@return**

\*/

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

OrderMapper映射：

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

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

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

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

</association>

<!--

collection:一对多的查询

property:属性名

javaType：集合类型

ofType：集合中的元素类型

autoMapping：开启自动映射

子标签：参照resultMap

-->

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

<!-- *detail\_id 关联表中对应的 c.id 的别名* -->

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

</collection>

</resultMap>

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

select \*,c.id as detail\_id from tb\_order a

LEFT JOIN tb\_user b on a.user\_id=b.id

LEFT JOIN tb\_orderdetail c on a.id=c.order\_id

where a.order\_number=#{number}

</select>

OrderMapperTest测试：

@Test

**public** **void** testQueryOrderWithUserDetail(){

System.***out***.println(**this**.orderMapper.queryOrderWithUserDetail("20140921001"));

}

## 多对多查询

**多对多查询：**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。

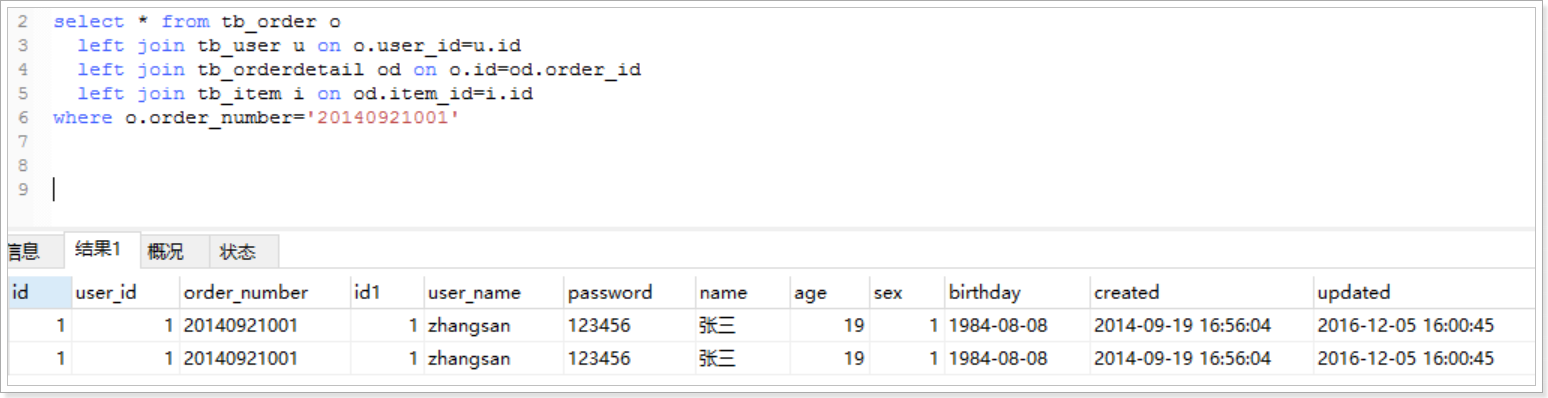
思路：

订单 : 商品 = n vs n --> 中间表 订单详情

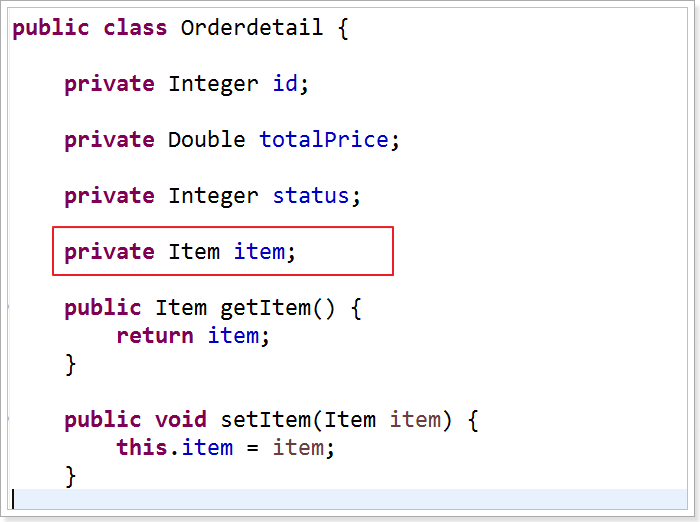
订单：订单详情 = 1 : n（体现在pojo对象中就是在Order对象中添加OrderDetail对象的集合）

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：



在Orderdetail添加Item属性，并添加get、set方法：



OrderMapper接口：

/\*\*

\* 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。

\* **@param** number

\* **@return**

\*/

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

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

<!-- 1 vs 1 -->

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

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

</association>

<!-- n vs n -->

<!-- 1 vs n -->

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

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

<!-- 1 vs 1 -->

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

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

</association>

</collection>

</resultMap>

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

select \*,c.id as detail\_id from tb\_order a

LEFT JOIN tb\_user b on a.user\_id=b.id

LEFT JOIN tb\_orderdetail c on a.id=c.order\_id

LEFT JOIN tb\_item d on c.item\_id=d.id

where a.order\_number=#{number}

</select>

OrderMapperTest测试：

@Test

**public** **void** testQueryOrderWithUserDetailItem(){

System.***out***.println(**this**.orderMapper.queryOrderWithUserDetailItem("20140921001"));

}

## resultMap的继承



## 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：一对一的映射

property 定义对象的属性名

javaType 属性的类型

autoMapping 开启自动映射

collection：一对多的映射

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** | 该配置影响的所有映射器中配置的缓存的全局开关(二级缓存)。 | 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 |

分析：

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。

## 改造一对一查询

在OrderMapper接口中，编写接口方法：

/\*\*

\* 测试延迟加载

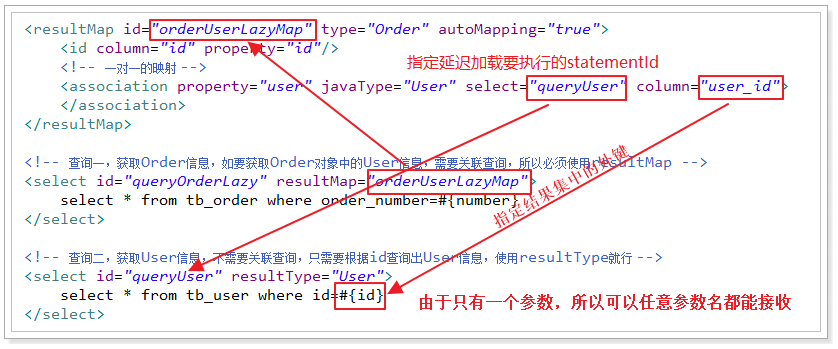
\* **@param** number

\* **@return**

\*/

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

OrderMapper配置：



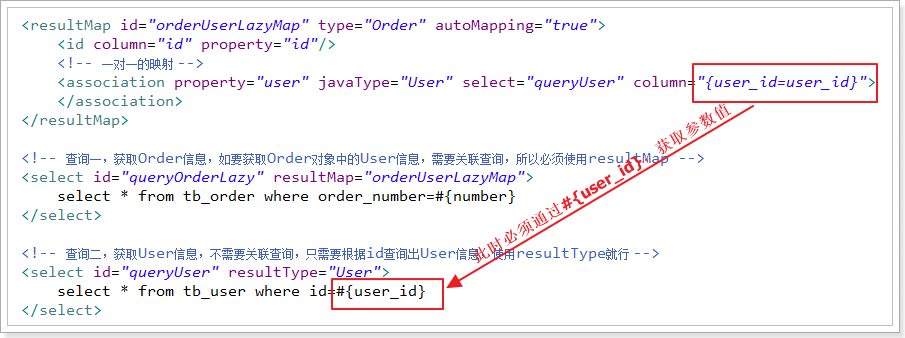
传递单个参数的时候 可以是 column=”user\_id” #{id}

或者是 column=”{user\_id=user\_id}” #{user\_id} --> {key=value} --> #{key}

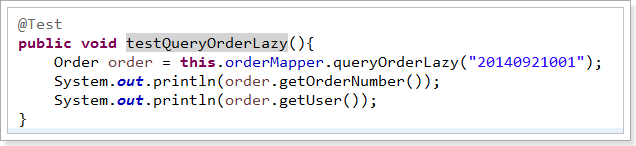
处理组合键时，需要传递多个参数，可以使用column=”{prop1=col1, prop2=col2, prop3=col3…}”，

设置多个列名传入到嵌套查询语句，mybatis会把prop1,prop2,prop3设置到目标嵌套的查询语句中的参数对象中。

子查询中，必须通过prop1,prop2,prop3获取对应的参数值，你也可以使用这种方式指定参数名例如：



测试：



日志信息：

|  |
| --- |
| 2017-04-13 00:11:14,982 [main] [cn.itcast.mybatis.mapper.OrderMapper.queryOrderLazy]-[DEBUG] ==> Preparing: select \* from tb\_order where order\_number=?  2017-04-13 00:11:15,028 [main] [cn.itcast.mybatis.mapper.OrderMapper.queryOrderLazy]-[DEBUG] ==> Parameters: 20140921001(String)  2017-04-13 00:11:15,049 [main] [cn.itcast.mybatis.mapper.OrderMapper.queryUser]-[DEBUG] ====> Preparing: select \* from tb\_user where id=?  2017-04-13 00:11:15,049 [main] [cn.itcast.mybatis.mapper.OrderMapper.queryUser]-[DEBUG] ====> Parameters: 1(Long)  2017-04-13 00:11:15,057 [main] [cn.itcast.mybatis.mapper.OrderMapper.queryUser]-[DEBUG] <==== Total: 1  2017-04-13 00:11:15,057 [main] [cn.itcast.mybatis.mapper.OrderMapper.queryOrderLazy]-[DEBUG] <== Total: 1  Order [id=1, userId=null, orderNumber=20140921001, user=User [id=1, userName=zhangsan, password=123456, name=张三, age=30, sex=1, birthday=Wed Aug 08 00:00:00 CST 1984, created=Fri Sep 19 16:56:04 CST 2014, updated=Sun Sep 21 11:24:59 CST 2014], detailList=null]  20140921001  User [id=1, userName=zhangsan, password=123456, name=张三, age=30, sex=1, birthday=Wed Aug 08 00:00:00 CST 1984, created=Fri Sep 19 16:56:04 CST 2014, updated=Sun Sep 21 11:24:59 CST 2014] |

分析：红色部分为查询Order信息的日志，绿色为查询Order中的User信息的日志。已经成功拆分出两个子查询，并且查询出了带有User信息的Order信息，说明改造OK。

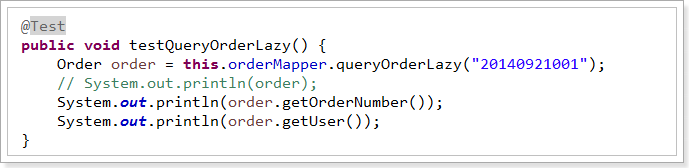
**--> 接下来开启延迟加载**

## 开启延迟加载

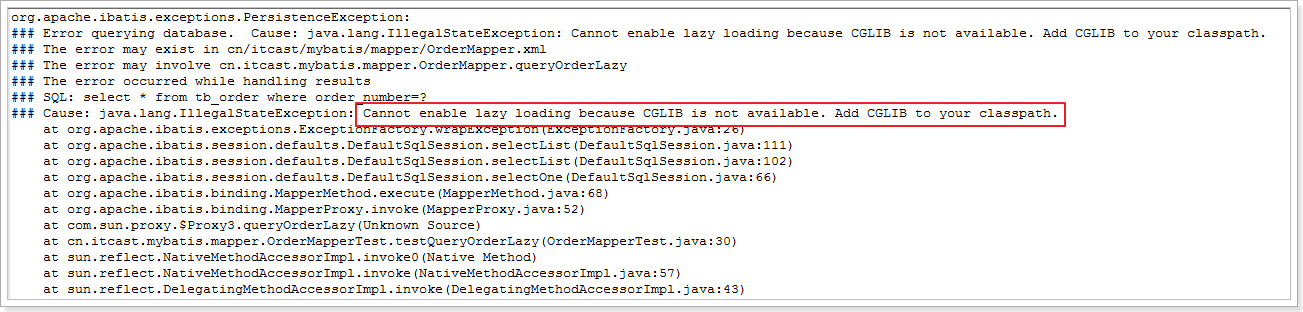
在mybatis-config.xml中配置行为参数：



修改测试用例，注释掉打印Order信息的这行代码：



执行，报错：

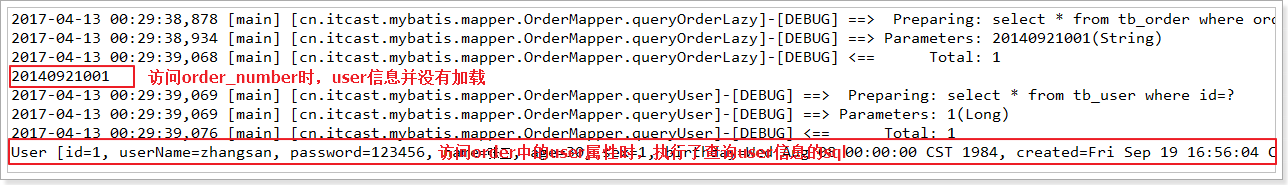


说明延迟加载需要cglib的支持。

在pom.xml中，添加cglib的依赖：

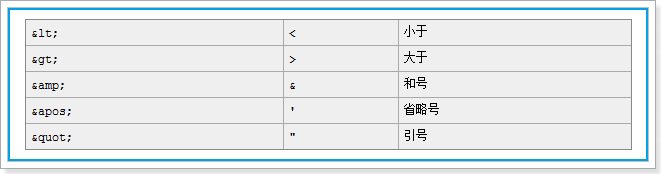
|  |
| --- |
| <dependency>  <groupId>cglib</groupId>  <artifactId>cglib</artifactId>  <version>3.1</version>  </dependency> |

执行：



# 如果sql语句中出现’<’的解决方案

1、使用xml中的字符实体



2、使用<![CDATA[ < ]]>

<select id="queryUserListLikeUserNameAndAge" resultType="User">

select \* from tb\_user

<!--

添加where关键字

有一定的纠错功能，自动去掉动态sql代码块之前多余的一个and|or

-->

<where>

<if test="userName!=null and userName.trim()!=''">or user\_name like '%' #{userName} '%'</if>

<if test="age!=null">and age <![CDATA[ < ]]> #{age}</if>

</where>

</select>