# 新三大框架之 MyBatis

**内参保密不得外传**

**达内Java培优方向**

**陈子枢 主讲**

**QQ: 1920-9520-78**

**http://dobetter.tedu.cn**

**2018**

# 第一天：MyBatis基础+动态SQL

## 介绍

### 前世今生



MyBatis的前身就是iBatis,iBatis本是apache的一个开源项目，2010年5月这个项目由apahce sofeware foundation 迁移到了google code，并且改名为MyBatis。

iBATIS 一词来源于“internet”和“abatis”（铁丝网）的组合，是一个基于Java的持久层框架。iBATIS提供的持久层框架包括SQL Maps和Data Access Objects（DAO）。

MyBatis 是支持普通 SQL 查询,存储过程和高级映射的优秀持久层框架。MyBatis 消除 了几乎所有的 JDBC 代码和参数的手工设置以及结果集的检索。MyBatis 使用简单的 XML 或注解用于配置和原始映射,将接口和 Java 的 POJOs(Plan Old Java Objects,普通的 Java 对象)映射成数据库中的记录。

【面试】MyBatis和hibernate对比，MyBatis更基础，要求使用者自己控制的东西更多。mybatis完成了基本的一些ORM概念，但是没有Hibernate那么完善。要使用mybatis，程序员的关注点更集中于SQL和数据库结构设计。mybatis没有hibernate使用起来那么面向对象，所以，在使用mybatis的时候，hibernate的一些思想和设计需要改变。

MyBatis的好处：更底层，对性能控制更有优势。

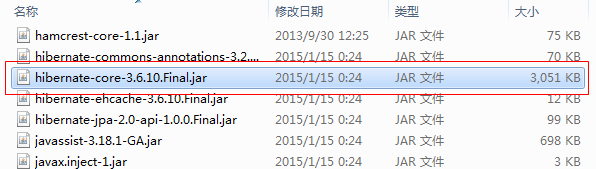
### MyBatis官网下载jar包

[**http://mybatis.github.io**](http://mybatis.github.io)

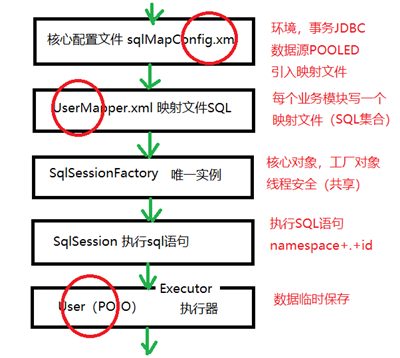
最新版本：

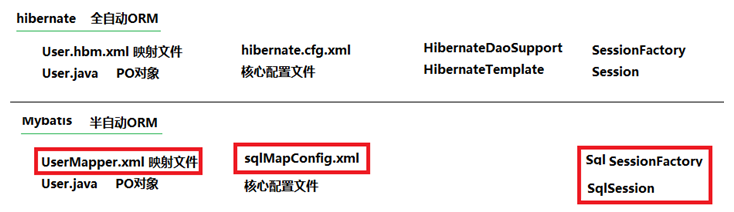
解压到D:\javaenv\all\_jar\mybatis-3.2.2

mybatis-3.2.2.jar 核心文件才684k



### MyBatis结构图





1、 mybatis配置

SqlMapConfig.xml，此文件作为mybatis的全局配置文件，配置了mybatis的运行环境等信息。

mapper.xml文件即sql映射文件，文件中配置了操作数据库的sql语句。此文件需要在SqlMapConfig.xml中加载。

2、 通过mybatis环境等配置信息构造SqlSessionFactory即会话工厂

3、 由会话工厂创建sqlSession即会话，操作数据库需要通过sqlSession进行。

4、 mybatis底层自定义了Executor接口操作数据库，Executor接口有两个实现，一个是基本实现、一个是缓存实现。

5、 Mapped Statement也是mybatis一个底层对象，它包装了mybatis配置信息及sql映射信息等。mapper.xml文件中一个sql对应一个Mapped Statement对象，sql的id即是Mapped statement的id。

6、 Mapped Statement对sql执行输入参数进行定义，包括HashMap、基本类型、pojo，Executor通过 Mapped Statement在执行sql前将输入的java对象映射至sql中，输入参数映射就是jdbc编程中对preparedStatement设置参数。

7、 Mapped Statement对sql执行输出结果进行定义，包括HashMap、基本类型、pojo，Executor通过 Mapped Statement在执行sql后将输出结果映射至java对象中，输出结果映射过程相当于jdbc编程中对结果的解析处理过程。

## 开发环境

### 创建JavaProject工程，导入jar包

方便引入log4j打印mybatis执行sql的日志信息

创建lib目录，拷入下列jar包，build path引入jar支持。

mybatis-3.2.2.jar mybatis核心包

mysql-connector-java-5.1.28-bin.jar mysql jdbc驱动

junit-4.9.jar 单元测试

# mybatis依赖包，在mybatis.zip中提供

asm-3.3.1.jar

cglib-2.2.2.jar

commons-logging-1.1.1.jar

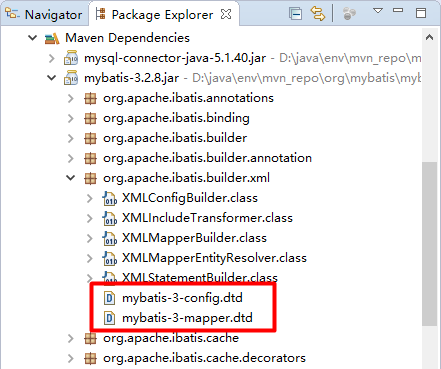
javassist-3.17.1-GA.jar

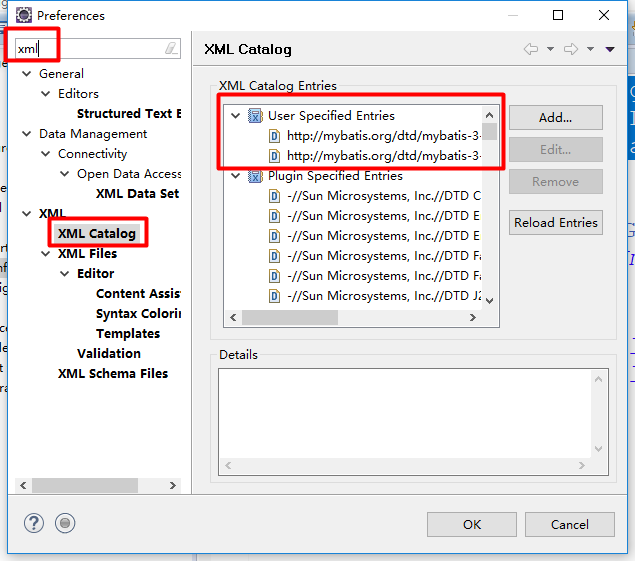
log4j-1.2.17.jar

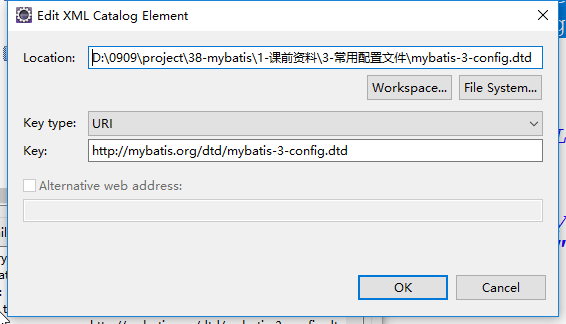
slf4j-api-1.7.5.jar

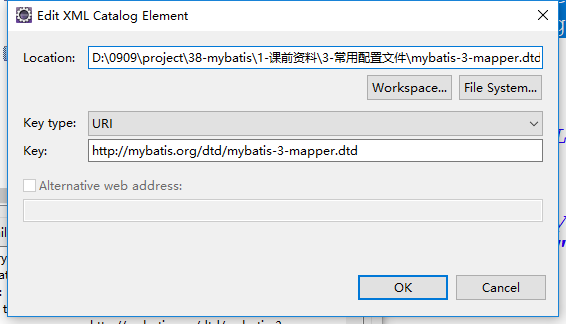
slf4j-log4j12-1.7.5.jar

### 引入DTD文件提示支持









### 引入log4j打印SQL语句

在classpath中建立一个名叫log4j.properties的文件，内容如下：

# Global logging configuration

log4j.rootLogger=DEBUG, stdout

# Console output...

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

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

log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n

再次运行：

DEBUG [main] - ooo Using Connection [com.mysql.jdbc.JDBC4Connection@17671]

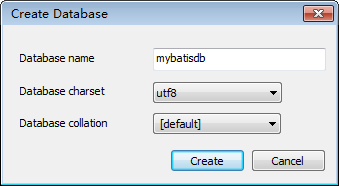
DEBUG [main] - ==> Preparing: select \* from user

DEBUG [main] - ==> Parameters:

可以很清楚的看到执行的sql和执行sql使用的参数。

### 创建mybatisdb数据库，创建user表

创建数据库



创建测试表

CREATE TABLE USER

(

ID INT NOT NULL,

NAME VARCHAR(30),

BIRTHDAY DATETIME,

ADDRESS VARCHAR(200),

PRIMARY KEY (ID)

);

测试数据：

INSERT INTO `user`(`id`,`name`,`birthday`,`address`)

VALUES ('1','夏言','1573-01-01','桂州村'),

('2','严嵩','1587-01-01','分宜县城介桥村'),

('3','徐阶','1580-01-01','明松江府华亭县'),

('4','高拱','1566-01-01','河南省新郑市高老庄村'),

('5','张居正','1558-01-01','江陵');

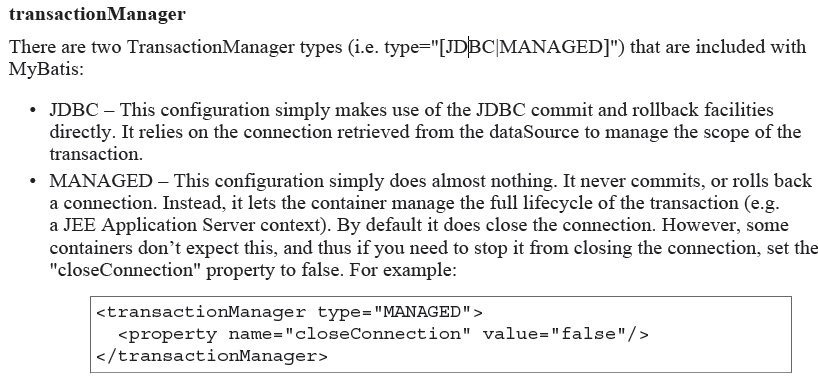
## 第一个MyBatis示例

### 需求

实现用户信息查询查询、新增、修改、删除操作

### 创建核心配置文件sqlMapConfig.xml

1、配置事务

JDBC直接使用jdbc的事务

MANAGED 容器去管理事务，mybatis什么也不做

2、数据源DataSource

UNPOOLED 非池化的

POOLED 池化的

JNDI 方式

3、核心配置文件 sqlMapConfig.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>

<environments default="test">

<environment id="test">

<transactionManager type="JDBC"/>

<dataSource type="POOLED">

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

<property name="url" value="jdbc:mysql://localhost:3306/mybatisdb?characterEncoding=utf8"/>

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

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

</dataSource>

</environment>

</environments>

</configuration>

这就是mybatis的一个典型的配置文件。在这个配置文件中，我们可以清楚得看到一些关键信息。第一，有一个environments这个元素，在这个元素里面配置的是一个一个的environment元素，每一个environment元素有一个id做为名字，而environments元素上面有一个default属性，引用environment元素的名称，很容易想象出他们之间的关系。在environments里面可以定义多个enviroment元素，使用default属性指定一个默认的environment。而在environment里面，定义了一组数据库相关的东西，包括transactionManager,这个元素代表着怎么去管理事务，后面还会详细讲到。而下面的dataSource就很明显了，代表连接数据库的相关信息。在这里，dataSource有一个属性type，这里我们写的是POOLED，很明显，使用连接池来做mybatis的连接提供。在dataSource里面，配置了四个属性，做为数据库连接信息配置。

### 创建User.java pojo对象

package cn.tedu.mybatis.domain;

public class User {

private Integer id;

private String name;

private java.util.Date birthday;

private String address;

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public java.util.Date getBirthday() {

return birthday;

}

public void setBirthday(java.util.Date birthday) {

this.birthday = birthday;

}

public String getAddress() {

return address;

}

public void setAddress(String address) {

this.address = address;

}

}

### 创建映射文件 UserMapper.xml

接下来，就是为我们的User对象完成映射了。前面说了，在mybatis中，需要自己去控制sql，所以我们的配置文件会像下面这样。在User的同级包下，添加一个XML文件，起名为UserMapper.xml，代表这个文件是User对象的映射文件：

<?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="cn.tedu.mybatis.domain">

<select id="find" resultType="cn.tedu.mybatis.domain.User">

select \* from user\_c

</select>

</mapper>

### 在sqlMapConfig.xml中声明 UserMapper.xml

<mappers>

<mapper resource="cn/tedu/mybatis/domain/UserMapper.xml"/>

</mappers>

### 创建测试类

package test;

import java.io.IOException;

import java.io.InputStream;

import java.util.List;

import org.apache.ibatis.io.Resources;

import org.apache.ibatis.session.SqlSession;

import org.apache.ibatis.session.SqlSessionFactory;

import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import org.junit.Before;

import org.junit.Test;

import cn.tedu.mybatis.domain.User;

public class TestMyBatis {

private SqlSessionFactory sqlSessionFactory; //线程安全，单例

@Before

public void init() throws IOException{

InputStream is = Resources.getResourceAsStream("sqlMapConfig.xml");

sqlSessionFactory = new SqlSessionFactoryBuilder().build(is);

}

@Test

public void testFind(){

//线程非安全

SqlSession sqlSession = sqlSessionFactory.openSession();

List<User> userList = sqlSession.selectList("cn.tedu.mybatis.domain.find");

for(User u : userList){

System.out.println(u);

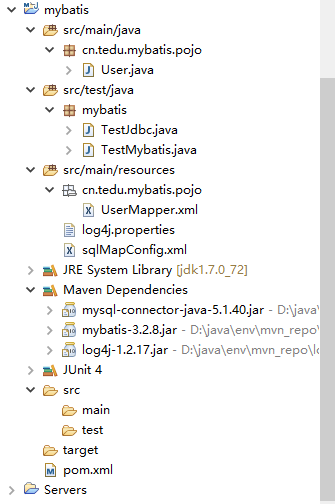
}

}

}

## Maven工程

### 工程结构



### pom.xml

<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>

<groupId>cn.tedu</groupId>

<artifactId>mybatis1803</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>mybatis1803</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<!--添加mysql驱动程序依赖 -->

<dependency>

<groupId>mysql</groupId>

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

<version>5.1.40</version>

</dependency>

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>3.2.8</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.17</version>

</dependency>

</dependencies>

</project>

### sqlMapConfig.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>

<environments default="test">

<environment id="test">

<transactionManager type="JDBC"/>

<dataSource type="POOLED">

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

<property name="url" value="jdbc:mysql://localhost:3306/mybatisdb?characterEncoding=utf-8"/>

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

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

</dataSource>

</environment>

</environments>

<mappers>

<mapper resource="cn/tedu/mybatis/pojo/UserMapper.xml"/>

</mappers>

</configuration>

### pojo

package cn.tedu.mybatis.pojo;

public class User {

private Integer id;

private String name;

private java.util.Date birthday;

private String address;

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public java.util.Date getBirthday() {

return birthday;

}

public void setBirthday(java.util.Date birthday) {

this.birthday = birthday;

}

public String getAddress() {

return address;

}

@Override

public String toString() {

return "User [id=" + id + ", name=" + name + ", birthday=" + birthday + ", address=" + address + "]";

}

public void setAddress(String address) {

this.address = address;

}

}

### 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 -->

<mapper namespace="cn.tedu.mybatis.pojo.UserMapper">

<!-- 查询所有，resultType返回值，标识返回类型List<> -->

<select id="find" resultType="cn.tedu.mybatis.pojo.User">

select id,name,birthday,address from user

</select>

<!-- 只返回一个值，返回一个整数值 -->

<select id="count" resultType="integer">

SELECT COUNT(\*) FROM user

</select>

<!-- 获取一个用户的信息（一条记录）#{id}参数，解析获取id值 -->

<select id="get" parameterType="integer" resultType="cn.tedu.mybatis.pojo.User">

SELECT \* FROM USER WHERE id = #{id}

</select>

<!-- 新增 -->

<insert id="insert">

INSERT INTO USER

(id,NAME,birthday,address)

VALUES(6,'tony','2018-06-19','北京')

</insert>

<!-- 修改 -->

<update id="update">

UPDATE USER

SET NAME='tony' WHERE id=6

</update>

<!-- 删除 -->

<delete id="delete" parameterType="integer">

DELETE FROM USER WHERE id=#{pid}

</delete>

<!-- 排序 -->

<select id="orderby" parameterType="map"

resultType="cn.tedu.mybatis.pojo.User">

SELECT \* FROM USER ${ob}

</select>

</mapper>

### 测试类

package mybatis1803;

import java.io.IOException;

import java.io.InputStream;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import org.apache.ibatis.io.Resources;

import org.apache.ibatis.session.SqlSession;

import org.apache.ibatis.session.SqlSessionFactory;

import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import org.junit.Before;

import org.junit.Test;

import cn.tedu.mybatis.pojo.User;

public class TestMybatis {

//成员变量，线程安全问题：

private SqlSessionFactory factory; //线程安全

@Before //junit，在其他方法执行前执行

public void init() throws IOException{

InputStream is = Resources.getResourceAsStream("sqlMapConfig.xml");

factory = new SqlSessionFactoryBuilder().build(is);

}

@Test

public void orderby(){

SqlSession session = factory.openSession();

String statement = "cn.tedu.mybatis.pojo.UserMapper.orderby";

//万能参数

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

map.put("ob", " order by birthday asc");

List<User> userList = session.selectList(statement, map);

for(User u:userList){

System.out.println(u);

}

}

@Test

public void delete(){

//SqlSession不是一个线程安全类

SqlSession session = factory.openSession();

String statement = "cn.tedu.mybatis.pojo.UserMapper.delete";

session.delete(statement, 6);

session.commit();

}

@Test //查询所有记录

public void find() throws IOException{

//读取核心配置文件，路径要求：

InputStream is = Resources.getResourceAsStream("sqlMapConfig.xml");

//获取到工厂对象

SqlSessionFactory factory =

new SqlSessionFactoryBuilder().build(is);

//创建SqlSession对象

SqlSession session = factory.openSession();

//查询所有的记录，namespace+"."+id

String statement = "cn.tedu.mybatis.pojo.UserMapper.find";

//selectList查询，找到映射文件，找到其中一个方法

//session.selectOne(statement);会报错，多个结果不能放到一个对象中

List<User> userList = session.selectList(statement);

//遍历对象

for(User u:userList){

System.out.println(u);

}

}

@Test

public void count(){

SqlSession session = factory.openSession();

//返回一条记录，而必须返回一条记录

String statement = "cn.tedu.mybatis.pojo.UserMapper.count";

Integer i = session.selectOne(statement);

System.out.println("有"+i+"条记录。");

}

@Test

public void get(){

SqlSession session = factory.openSession();

String statement = "cn.tedu.mybatis.pojo.UserMapper.get";

User user = session.selectOne(statement, 5);

System.out.println(user);

}

@Test

public void insert() throws IOException{

InputStream is = Resources.getResourceAsStream("sqlMapConfig.xml");

SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(is);

SqlSession session = factory.openSession();

String statement = "cn.tedu.mybatis.pojo.UserMapper.insert";

session.insert(statement);

session.commit(); //mybatis事务，默认不是自动提交，手动提交

}

@Test

public void update() throws IOException{

InputStream is = Resources.getResourceAsStream("sqlMapConfig.xml");

SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(is);

SqlSession session = factory.openSession();

String statement = "cn.tedu.mybatis.pojo.UserMapper.update";

session.update(statement);

session.commit();

}

}

## 常见操作

### 【查询】操作

映射文件：

<select id="find" resultType="cn.tedu.mybatis.domain.User">

select \* from user

</select>

测试类：

@Test

public void testFind(){

//线程非安全

SqlSession sqlSession = sqlSessionFactory.openSession();

List<User> userList = sqlSession.selectList("cn.tedu.mybatis.domain.find");

for(User u : userList){

System.out.println(u);

}

}

### 【新增】操作

映射文件：

<select id="insert" parameterType="cn.tedu.mybatis.domain.User">

insert into user (id,name,birthday,address)

values(#{name},#{birthday},#{address})

</select>

测试类：

@Test

public void testFind(){

//线程非安全

SqlSession sqlSession = sqlSessionFactory.openSession();

sqlSession.insert("cn.tedu.mybatis.domain.insert");

sqlSession.commit();

}

### 【修改】操作

在UserMapper.xml中加入：

<update id="update" parameterType="cd.tedu.mybatis.domain.User">

UPDATE user SET name = #{name},hiredate = #{hireDate} WHERE id = #{id}

</update>

1， update：代表当前定义了一个update操作（sql的update语句）；

a) id：为当前操作定义了一个名字，用于调用

b) parameterType：update调用需要一个cd.tedu.mybatis.domain.User的类型实例做为参数。

2， UPDATE user SET name = #{name},hiredate = #{hireDate} WHERE id = #{id}：即完整的UPDATE语句，其中的#{name},#{hireDate},#{id}和insert语句中的完全一致。

修改测试：

@Test

public void testUpdate(){

SqlSession session = sqlSessionFactory.openSession();

try{

User u = new User();

u.setId(1L);

u.setName("杨元庆");

u.setHireDate(new Date());

session.update("cn.tedu.mybatis.domain.UserMapper.update", u);

session.commit();

}catch(Exception e){

e.printStackTrace();

session.rollback();

}finally{

session.close();

}

}

对照保存测试，很好理解，关键的一句：

session.update("cd.tedu.mybatis.domain.UserMapper.update", u)：即调用在cd.tedu.mybatis.domain.UserMapper中定义的名字为update的Update语句，传入一个cd.tedu.mybatis.domain.User的对象实例。

运行测试，console打印：

DEBUG - ooo Using Connection [com.mysql.jdbc.JDBC4Connection@14cf7d6]

DEBUG - ==> Preparing: update user set name=?, hireDate=? where id=?

DEBUG - ==> Parameters: 杨元庆(String), 2013-12-31 17:02:14.225(Timestamp), 1(Long)

### 【获取】一条

在UserMapper.xml中添加如下配置：

<select id="get" parameterType="long" resultType="cn.tedu.mybatis.domain.User">

select \* from user where id=#{id}

</select>

1， select：代表这是一个SELECT语句

a) parameterType：代表要执行这个select语句需要传入一个类型为long的参数，即User对象的id

b) resultType：非常重要的东西，即完成ORM的映射关系所在。这里指定的cd.tedu.mybatis.domain.User代表，把结果集转换成一个User对象实例。注意，在这里有一个要求就是，属性名称和列的名称必须一致，才能完成转型。否则就需要自定义属性到字段的映射规则。

2， SELECT \* FROM user WHERE id = #{id}：即查询语句。非常重要的一点，这个#{id}是mybatis在使用查询单个对象的时候默认使用的参数的名称。即传入的类型为long的值，mybatis就用#{id}来代替。

Get测试：

@Test

public void testGet(){

SqlSession session = sqlSessionFactory.openSession();

try{

User u = session.selectOne("cn.tedu.mybatis.domain.UserMapper.get", 1);

System.out.println(u);

}catch(Exception e){

e.printStackTrace();

}finally{

session.close();

}

}

其中很重要的一点：

User u = session.selectOne("cd.tedu.mybatis.domain.UserMapper.get",1l);

调用的是cd.tedu.mybatis.domain.UserMapper中定义的get对应的SELECT语句。注意这里使用的是session.selectOne，代表，选择的是一个。那么mybatis就会把查询到的结果集中的第一行数据按照列名和属性名一一对应的关系，把结果包装成一个User类型的实例。

从这里也能得出结论：在mapper文件中定义的sql语句的类型和在session中使用的调用方法是对应的。

运行测试：

DEBUG - ooo Using Connection [com.mysql.jdbc.JDBC4Connection@1b39df1]

DEBUG - ==> Preparing: select \* from user where id=?

DEBUG - ==> Parameters: 1(Integer)

User [id=1, name=柳传志, hireDate=Tue Dec 31 00:00:00 CST 2013]

### 【查询】多条

在UserMapper.xml中加入：

<select id="list" resultType="cn.tedu.mybatis.domain.User">

select \* from user

</select>

1，非常重要的一点，这次返回的结果应该是一个List<User>，但是配置的resultType仍然是cd.tedu.mybatis.domain.User，mybatis强大的一点，对于select，其实我们配置的都是把每一条结果集转成的对象类型。

2，SELECT \* FROM user：即查询语句。因为不需要参数了，所以也没有定义paramterType。

查询测试：

@Test

public void testList(){

SqlSession session = sqlSessionFactory.openSession();

try{

List<User> userList = session.selectList("cn.tedu.mybatis.domain.UserMapper.list", 1);

for(User u : userList){

System.out.println(u);

}

}catch(Exception e){

e.printStackTrace();

}finally{

session.close();

}

}

1， 使用的是session.selectList，代表这次调用的select语句需要把结果集包装成一个对象列表。

DEBUG - ooo Using Connection [com.mysql.jdbc.JDBC4Connection@16ac534]

DEBUG - ==> Preparing: select \* from user

DEBUG - ==> Parameters:

User [id=1, name=柳传志, hireDate=Tue Dec 31 00:00:00 CST 2013]

User [id=2, name=杨元庆, hireDate=Wed Jan 01 00:00:00 CST 2014]

得到正确的结果。

### 【删除】操作

在UserMapper.xml中添加：

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

delete from user

where id=#{pid}

</delete>

使用delete标识这是一个删除语句。参数parameterType=long表明删除数据的参数。同select一样，如果传入的参数就是一个id，那么在sql语句中使用#{id}来引用这个参数。

测试用例：

@Test

public void testDelete(){

SqlSession session = sqlSessionFactory.openSession();

try{

session.delete("cn.tedu.mybatis.domain.UserMapper.delete", 2);

session.commit();

}catch(Exception e){

e.printStackTrace();

session.rollback();

}finally{

session.close();

}

}

使用session.delete方法指定调用的删除SQL和传入参数。运行结果：

DEBUG - ooo Using Connection [com.mysql.jdbc.JDBC4Connection@1a431fd]

DEBUG - ==> Preparing: delete from user where id=?

DEBUG - ==> Parameters: 1(Integer)

### 异常回滚

测试事务是否起作用，在delete后，commit前，加下面语句

System.out.println(1/0)

使其出错，

运行结果，报错，

DEBUG - ooo Using Connection [com.mysql.jdbc.JDBC4Connection@1be02b6]

DEBUG - ==> Preparing: delete from user where id=?

DEBUG - ==> Parameters: 2(Integer)

java.lang.ArithmeticException: / by zero

说明事务起作用。

到这里，mybatis的CRUD基本演示完毕。从这些方法里面大概能够感受到mybatis的一些基本的使用方式。

### 【记录数】操作

在UserMapper.xml中添加：

<!-- count(\*) 返回单值 -->

<select id="count" resultType="int">

select count(\*) from user

</select>

//测试返回记录总数

@Test

public void testCount(){

SqlSession session = sqlSessionFactory.openSession(); //从工厂中获取sqlSession

int i = session.selectOne("cn.tedu.mybatis.mapper.UserMapper.count");

System.out.println(i);

}

### SQL中有特殊字符

当SQL中有特殊字符，mybatis不能正常解析时，用CDATA括起来就解决了

<![CDATA[

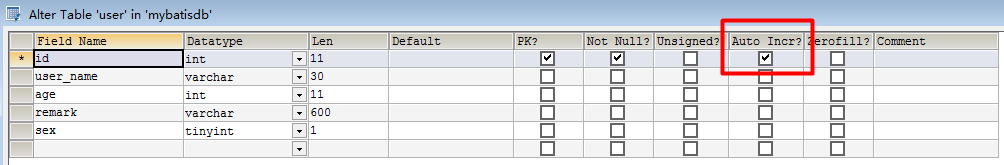
and age<=#{age}

]]>

## 自增主键

### 自增主键方式一

首先必须数据库支持，mysql,sqlserver支持自增主键



<insert id="insert" parameterType="cn.tedu.mybatis.domain.User">

insert into user\_c(name,birthday,address)

values(#{name},#{birthday},#{address})

</insert>

注意必须数据库支持自增主键mysql,sqlserver都支持，同时在insert语句时，就无需写id自增字段，数据库会在插入时自动设置这个值

### 自增主键方式二

配置自增的属性，注意第一种方式不能写id字段，下面的方式必须写id字段

useGeneratedKeys是否使用自增主键

keyProperty指定自增主键的列

<insert id="save"

keyProperty="id" useGeneratedKeys="true"

parameterType="user">

insert into user (name,hireDate) values(#{name},#{hireDate})

</insert>

### 自增主键方式三（不推荐）

通过修改sql映射文件，可以将mysql自增主键返回：

<insert id="insert" parameterType="cn.tedu.mybatis.domain.User">

<!-- selectKey将主键返回，需要再返回，mysql的底层函数 -->

<selectKey keyProperty="id" order="AFTER" resultType="int">

select LAST\_INSERT\_ID()

</selectKey>

insert into user\_c(id,name,birthday,address)

values(#{id},#{name},#{birthday},#{address})

</insert>

添加selectKey实现将主键返回

keyProperty:返回的主键存储在pojo中的哪个属性

order：selectKey的执行顺序，是相对与insert语句来说，由于mysql的自增原理执行完insert语句之后才将主键生成，所以这里selectKey的执行顺序为after

resultType:返回的主键是什么类型

LAST\_INSERT\_ID():是mysql的函数

### Oracle可采用序列完成

首先自定义一个序列且于生成主键，selectKey使用如下：

<selectKey resultType="java.lang.Integer" order="BEFORE"

keyProperty="id">

SELECT 自定义序列.NEXTVAL FROM DUAL

</selectKey>

注意这里使用的order是“BEFORE”

## 小结

### sqlMapConfig.xml

* 文件名可以随意命名，习惯名称sqlMapConfig.xml
* 配置3个内容：基础配置，数据源信息，映射文件

### 映射文件UserMapper.xml

* 文件名可以随意命名，习惯使用 PO对象名+Mapper.xml
* 命名空间<mapper namespace="cn.tedu.mybatis.mapper.UserMapper">
* 命名空间可以没有 <mapper namespace="">
* <select 底层不关心标签的命名，随意。但是底层对映射文件是有校验。

MyBatis底层实现简单，核心就是实现拼接sql语句。

### #{}获取参数和${}的区别

MyBatis中使用parameterType向SQL语句传参，parameterType后的类型可以是基本类型int,String,HashMap和java自定义类型。

在SQL中引用这些参数的时候，可以使用两种方式#{parameterName}或者${parameterName}，

首先，我们说一下这两种引用参数时的区别，使用#{parameterName}引用参数的时候，Mybatis会把这个参数认为是一个字符串，例如传入参数是“Smith”，那么在SQL（Select \* from emp where name = #{employeeName})使用的时候就会转换为Select \* from emp where name = 'Smith';同时在SQL（Select \* from emp where name = ${employeeName}）使用的时候就会转换为Select \* from emp where name = Smith。

再次，从安全性上考虑，能使用#尽量使用#来传参，因为这样可以有效防止SQL注入的问题。

# 第二天：接口开发+关联关系

## 【重点】动态SQL语句

Mybatis提供使用ognl表达式动态生成sql的功能。

### <sql><include>

可以在配置文件中调用的地方都进行include引用

<sql id="cols">

id,name,hireDate

</sql>

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

select <include refid="cols"/> from user

</select>

### <if>

### <where>

<select id="find" parameterType="map" resultType="User">

select <include refid="userCols"/> from user\_c

<where>

<if test="address != null">address like #{address}</if>

<![CDATA[

or age>#{ageStart}

or age<#{ageEnd}

]]>

</where>

</select>

### <set>

<!-- 修改 -->

<update id="update" parameterType="cn.mybatis.domain.User">

update user\_c

<set>

<if test="name!=null">name = #{name},</if>

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

<if test="address!=null">address=#{address},</if>

</set>

where id = #{id}

</update>

## 集合参数<foreach>

### 数组array

<!-- 批量删除，参数：数组 -->

<delete id="deleteArray" parameterType="integer">

delete from user

where id in

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

#{id}

</foreach>

</delete>

array是mybatis提供的类型，integer是集合元素的类型

### List集合

<!-- 批量删除，参数：LIST -->

<delete id="deleteList" parameterType="integer">

delete from user

where id in

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

#{id}

</foreach>

</delete>

list是mybatis提供的类型，integer是集合元素的类型

### Map/HashMap

<!-- 批量删除，参数：Map -->

<delete id="deleteMap" parameterType="map">

delete from user

where id in

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

#{id}

</foreach>

</delete>

ids为map中的key名称，key=ids，value={1,2,3}

## 优化，使代码更加简洁

前面讲了CRUD的全部测试，下面是对CRUD示例的优化：

### 别名

typeAliases：在示例中的UserMapper.xml文件中可以看到，凡是使用到User类型的时候，都需要写User的类的全限定名。在sqlMapConfig.xml中，可以使用typeAliases元素来简化这个操作：

在sqlMapConfig.xml中添加：

<typeAliases>

<typeAlias type="cn.tedu.mybatis.domain.User" alias="user"/>

</typeAliases>

或者

<typeAliases><!-- 类型的别名，大写User/user都行 -->

<package name="cn.tedu.mybatis.pojo"/>

</typeAliases>

UserMapper.xml中，修改，不论是参数，还是返回值，都可以简化

<?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.tedu.mybatis.domain.UserMapper">

<insert id="save" keyProperty="id" useGeneratedKeys="true" parameterType="user">

insert into user (name,hireDate) values(#{name},#{hireDate})

</insert>

<update id="update" parameterType="user">

update user set name=#{name}, hireDate=#{hireDate}

where id=#{id}

</update>

<select id="get" parameterType="long" resultType="user">

select \* from user where id=#{pid}

</select>

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

select \* from user

</select>

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

delete from user

where id=#{pid}

</delete>

</mapper>

### ResultMap映射不规范字段

<?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.tedu.mybatis.pojo.PersonMapper">

<!-- 最强大对象ResultMap，type结果封装到哪个pojo对象，type就是谁 -->

<resultMap type="Person" id="personRM">

<!-- 主键,property对应pojo属性，column表的字段 -->

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

<!-- 普通字段 -->

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

</resultMap>

<!-- 查询所有 -->

<select id="find" parameterType="Person"

resultMap="personRM"

>

SELECT id,user\_name FROM person

WHERE user\_name LIKE #{userName}

</select>

</mapper>

### 自动匹配规范驼峰规则

数据库中我们习惯使用全大写，多个单词用下划线隔开，而po对象，习惯使用java驼峰规则。那一个一个手工写resultMap字段，浪费开发时间。Mybatis可以配置mapUnderscoreToCamelCase，实现自动映射。这个值默认为true。

1）全局配置：

<?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>

<settings>

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

</settings>

</configuration>

2）resultMap配置autoMapping="true"

<resultMap type="cn.tedu.jk.domain.Contract" id="contractRM" autoMapping="true">

<id property="id" column="CONTRACT\_ID"/>

</resultMap>

注意：主键需要单独写，其它字段就可以直接利用驼峰规则自动映射。

## db.properties

我们说过数据库的连接信息一般要放到一个额外的.properties文件中，mybatis允许我们这样做。

首先，修改sqlMapConfig.xml文件：

<dataSource type="POOLED">

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

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

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

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

</dataSource>

联想spring的dataSource配置方式，不难理解。

### 方式一

在SqlSessionFactoryBuilder的build方法中，还提供了额外传入Properties对象的方法：

public SqlSessionFactory build(InputStream inputStream, Properties properties)

这个方法后面的Properties对象就可以做为sqlMapConfig.xml中的参数来源。所以，我们可以这样来使用：

Properties p=new Properties();

p.load(Resources.getResourceAsStream("db.properties"));

并在classpath下定义一个db.properties文件：

driverClass=com.mysql.jdbc.Driver

url=jdbc:mysql://localhost:3306/mybatisdb?characterEncoding=utf-8

username=root

password=root

### 方式二

在sqlMapConfig.xml中有properties这样一个标签，那么我们可以在sqlMapConfig.xml中定义：

<properties>

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

<property name="url" value="jdbc:mysql:///mybatisdb?characterEncoding=utf-8"/>

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

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

</properties>

### 方式三

在sqlMapConfig.xml中的properties元素中，引入外部的properties文件：

<properties resource="db.properties" />

并在classpath中添加db.properties文件即可。

第三种方式和第二种方式可以混用，即：

db.properties文件中：

driverClass=com.mysql.jdbc.Driver

url=jdbc:mysql://localhost:3306/mybatisdb?characterEncoding=utf-8

sqlMapConfig.xml文件中：

<properties resource="db.properties">

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

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

</properties>

**三者的优先级为：**

**代码传入的Properties > resource加载的Properties > properties元素中定义的property。**

## Mapper接口的实现

在上面的测试用例中，在调用session的方法的时候，都会传入要调用的SQL的namespace+id名称，这不是必须的。可以只传入id即可。但是，如果在mybatis的环境中有多个相同id的映射名称，就会报错。所以，一般情况下，调用方法最好还是使用namespace+id。但是，namespace+id的使用方式很容易报错，因为是string类型的，没有检查。所以，mybatis提供了一种非常好的设计方式来避免这种问题，即Mapper接口。

**接口定义有如下特点：**

1、 Mapper接口方法名和mapper.xml中定义的每个sql的id相同

2、 Mapper接口方法的输入参数类型和mapper.xml中定义的每个sql 的parameterType的类型相同

3、 Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的resultType的类型相同

### 第一步：创建Mapper接口

对照UserMapper.xml的定义，我们只需要创建一个接口，注意，接口的名字和包必须和mapper.xml定义的namespace一致，接口的方法名必须和mapp.xml中定义的id名称一致。

即创建一个接口：cn.tedu.mybatis.domain.UserMapper：

package cn.tedu.mybatis.domain;

import java.util.List;

public interface UserMapper {

//对应<insert id="save" keyProperty="id" useGeneratedKeys="true" parameterType="user">

public void save(User u);

//对应<update id="update" parameterType="user">

public void update(User u);

//对应<delete id="delete" parameterType="long">

public void delete(Long id);

//对应<select id="get" parameterType="long" resultType="user">

public User get(Long id);

//对应<select id="list" resultType="user">

public List<User> list();

}

### 第二步：<mapper class=" " /> 在sqlMapConfig.xml中声明

使用mapper接口类路径

如：<mapper class="cn.tedu.mybatis.mapper.UserMapper"/>

接口方式就无需xml配置访问

注意：此种方法要求mapper接口名称和mapper映射文件名称相同，且放在同一个目录中。

### 第三步：创建测试类

UserMapper mapper = session.getMapper(UserMapper.class);

对照每一个方法的注释，应该是很好理解的。

使用Mapper接口。有了Mapper接口，并且Mapper接口放在了指定的位置之后，我们的测试就可以写成：

@Test

public void testSave(){

SqlSession session = factory.openSession();

try{

User u = new User();

u.setName("柳传志");

u.setHireDate(new Date());

UserMapper mapper = session.getMapper(UserMapper.class);

mapper.save(u);

session.commit();

}catch(Exception e){

e.printStackTrace();

session.rollback();

}finally{

session.close();

}

}

代码变得非常清晰，并且类型都使用接口强制性的完成。

**Mybatis怎么做的？**

@Test

public void testMapper(){

SqlSession session = MyBatisUtil.openSession();

try{

UserMapper mapper = session.getMapper(UserMapper.class);

System.out.println(mapper.getClass().getName());

}finally{

session.close();

}

}

打印结果：

$Proxy4

很简单了，mybatis为接口做了一个动态代理。在执行UserMapper接口上面的方法时，参考接口的全限定名，即可找到对应的UserMapper.xml，在执行接口上面的每一个方法的时候，实际上就是在执行namespace+id，mybatis在根据定义的方法的元素，选择调用合适的session的方法来执行，并传入参数可以。

使用Mapper接口的方式，在集成Spring+MyBatis也非常方便。因为我们可以直接把Mapper接口看作DOMAIN的DAO接口了。

# 第三天：高级特性+自动生成代码

## 关联关系

Mybatis表现关联关系比hibernate简单，没有分那么细致one-to-many、many-to-one、one-to-one。而是只有两种association（一）、collection（多），表现很简洁

关联出的结果集不能有同名字段，否则mybatis无法正确映射。

如果同名，Mybatis会自动第一次出现的值再次放到对象的属性中，这样业务逻辑就将错误。

解决办法：

如果同名必须重命名，可以通过resultMap映射时column属性写结果集的字段名称，而property属性仍然写POJO对象的属性。例如：结果集中User.id，UserInfo.id。User.id就无需修改，而UserInfo.id修改为USER\_INFO\_ID

<result property=”id” column=”USER\_INFO\_ID”/>

那为何proerty中就不怕同名呢？因为它会配置在不同的result下，例如UserInfo在assocaition下。

### 一对一

需求：

用户和用户扩展信息一对一关联；查询用户和用户的扩展信息。



下面代码只拿部分字段举例

**第一步：创建用户User实体PO对象**

单向关联，需求一般只是按用户来查询用户的扩展信息，因此只在User对象中配置。

private UserInfo userInfo;

package cn.tedu.mybatis.domain;

import java.util.Date;

public class User {

private UserInfo userInfo;

public UserInfo getUserInfo() {

return userInfo;

}

public void setUserInfo(UserInfo userInfo) {

this.userInfo = userInfo;

}

private Integer id;

private String name;

private Date birthday;

private String address;

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Date getBirthday() {

return birthday;

}

public void setBirthday(Date birthday) {

this.birthday = birthday;

}

public String getAddress() {

return address;

}

public void setAddress(String address) {

this.address = address;

}

@Override

public String toString() {

return "User [id=" + id + ", name=" + name + ", birthday=" + birthday

+ ", address=" + address + "]";

}

}

**第二步：创建用户扩展信息UserInfo实体PO对象**

package cn.tedu.mybatis.domain;

public class UserInfo {

private Integer id;

private String station;

private Integer salary;

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getStation() {

return station;

}

public void setStation(String station) {

this.station = station;

}

public Integer getSalary() {

return salary;

}

public void setSalary(Integer salary) {

this.salary = salary;

}

}

**第三步：增加ResultMap映射，继承用户的ResultMap userRM**

resultMap的属性：

|  |  |
| --- | --- |
| 属性 | 作用 |
| type | 配置查询出的结果集要封装到哪个实体对象中 |
| id | 主键；  property POJO对象的属性；  column 查询结果集对应的字段名 |
| result | 普通字段  property POJO对象的属性  column 查询结果集对应的字段名 |
| extends | 可以继承，这样可以复用，方便维护 |
| association | 对一关联，内部结构和单个对象的resultMap结构相同，可以复制 |
| collection | 对多关联，内部结构和单个对象的resultMap结构相同，可以复制 |

<!-- resultMap配置，非常强大，它是Mybatis中最强悍地方，封装复杂对象的关系 -->

<resultMap type="cn.tedu.mybatis.domain.User" id="userRM">

<!-- 主键映射 property:实体的属性；column:结果集的字段名称 -->

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

<!-- 普通属性映射，当结果集中的字段名称和实体属性名称不同时，必须通过下面的映射来设置 -->

<result property="name" column="USER\_NAME"/>

<result property="birthday" column="BIRTHDAY"/>

<result property="address" column="ADDRESS"/>

</resultMap>

<!-- 查询用户信息，包括扩展信息，一对一关联 -->

<resultMap type="User" id="userInfoRM" extends="userRM">

<!-- 关联关系 -->

<association property="userInfo" javaType="cn.tedu.mybatis.domain.UserInfo">

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

<result property="station" column="STATION"/>

<result property="salary" column="SALARY"/>

</association>

</resultMap>

**第四步：增加关联查询SQL**

要查询出所需的所有对象的内容，形成一个超大的结果集

<!-- 关联关系查询，在这个结果集中要得到所有关联对象的数据 -->

<select id="findUserInfo" resultMap="userInfoRM">

SELECT

u.id,u.user\_name,u.birthday,u.address,

info.id,info.station,info.salary

FROM usermap\_c u,userinfo\_c info

WHERE u.id=info.id

</select>

**第五步：调用**

@Test //查询关联关系

public void TestFindUserInfo(){

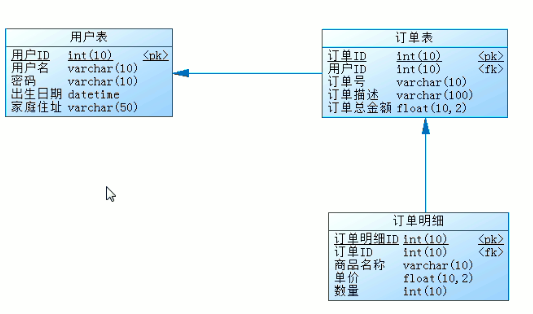
SqlSession session = factory.openSession();

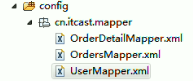
List<User> userList = session.selectList("cn.tedu.mybatis.domain.UserMapper.findUserInfo");

System.out.println(userList.size());

}

### 一对多







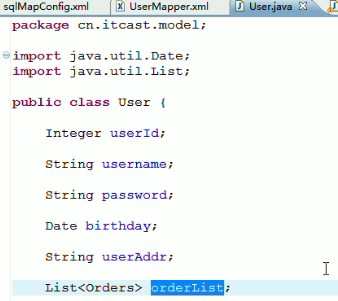
采用的内连接



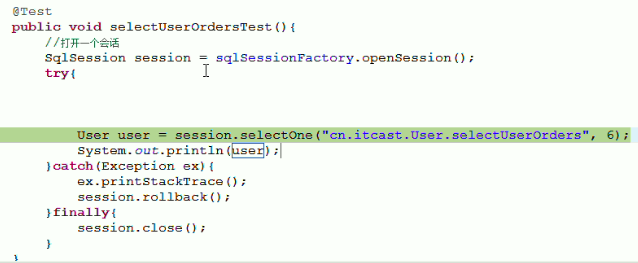
字段名相同，会自动重命名，其内容不同

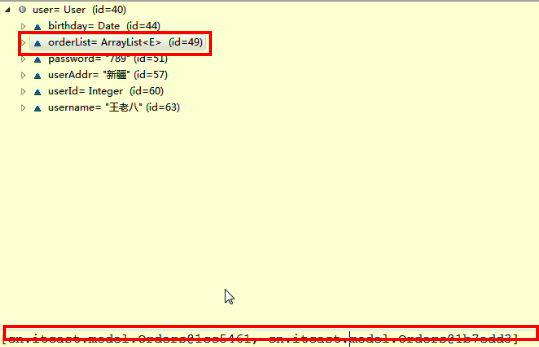
通常采用外连接











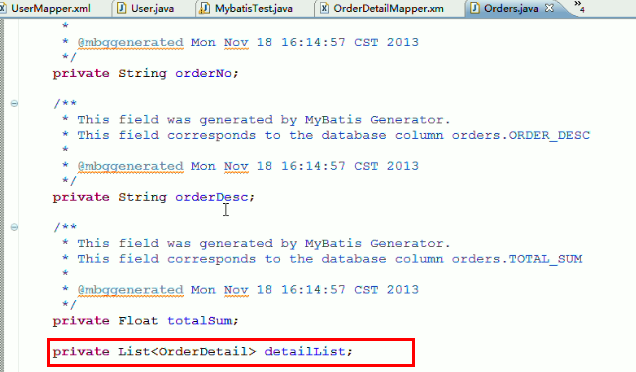
一个User对应多个orders，一个orders对应多个orderDetail。两层一对多





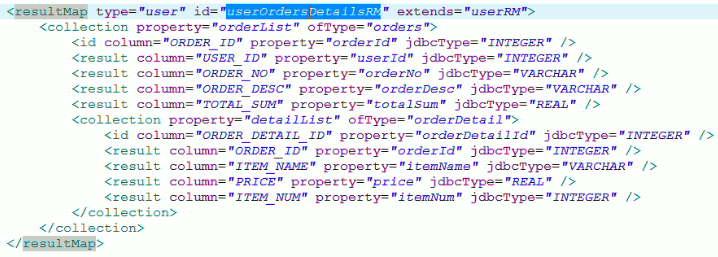
Orders映射中加

OrderDetail detailList

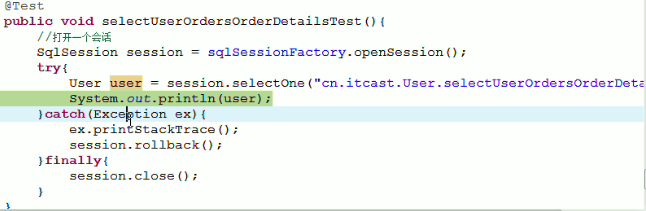


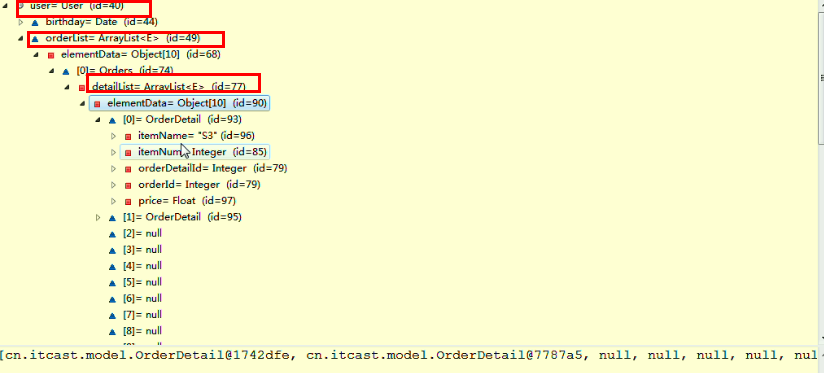
sqlMapConfig.xml中加aliases

ordersDetail





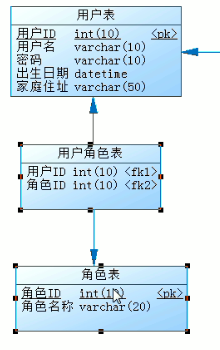


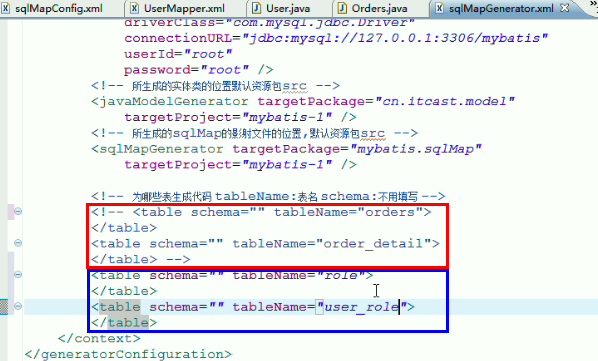


### 多对多

在数据库中多对多如何体现？三张表对吧

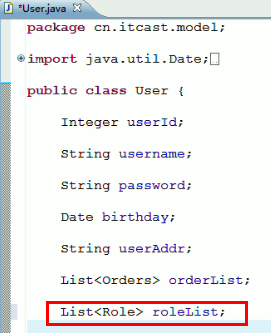
最典型的是哪个？应用场景？ 对，权限，角色和用户

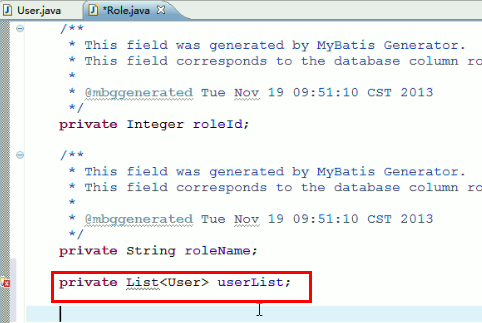




注释掉，以前生成的（再次生成代码时，会覆盖掉之前的内容）

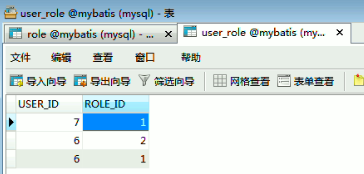
加上新的，生成后，删除example的

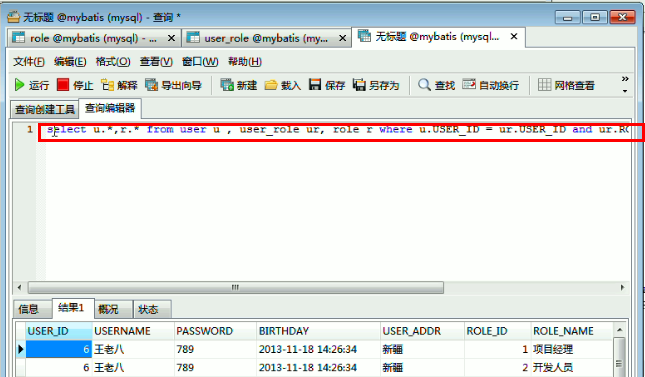




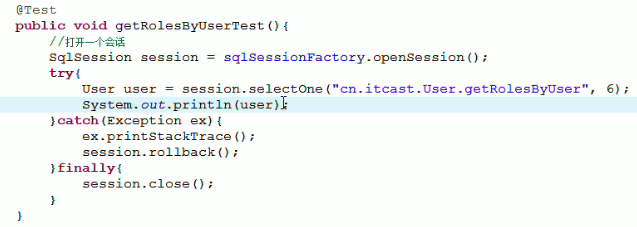


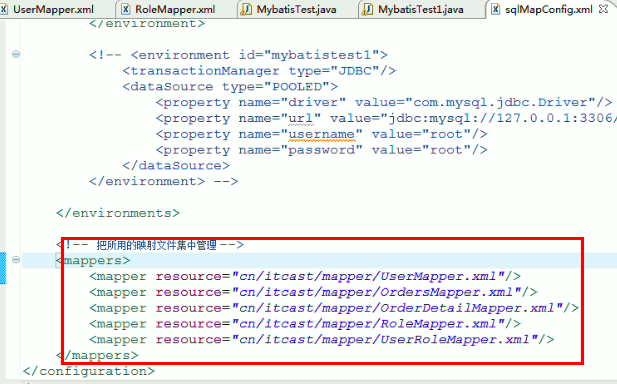


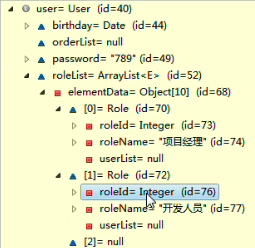












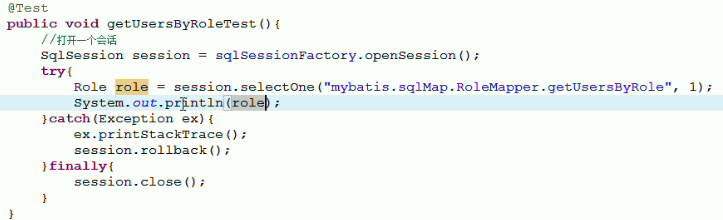
其实从用户这边来看，就是一对多；那我们从角色这边来看，也是一对多。

那这样也就实现了用户和角色多对多的情况

下面我们站在角色的角度来写下









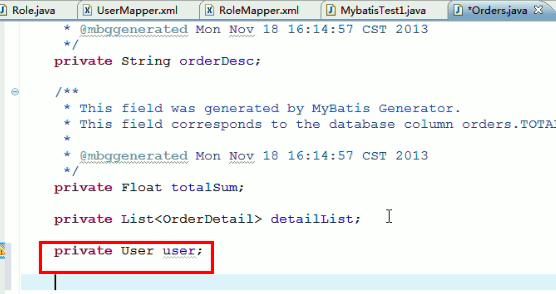
所以你们就理解了，多对多就是两层一对多。

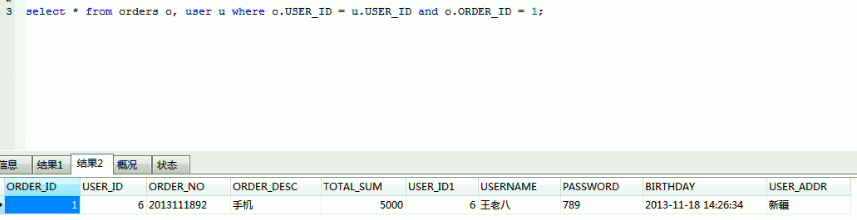
### 多对一

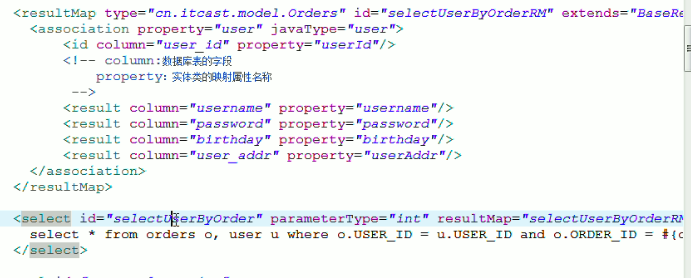


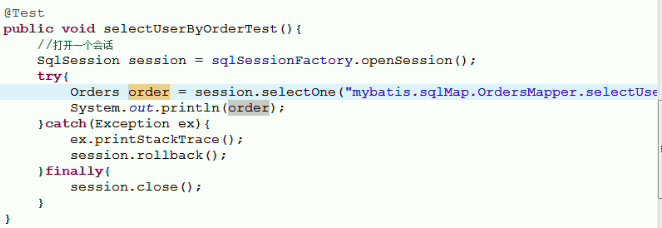
从用户表看订单，它们是一对多。从订单看用户是多对一。

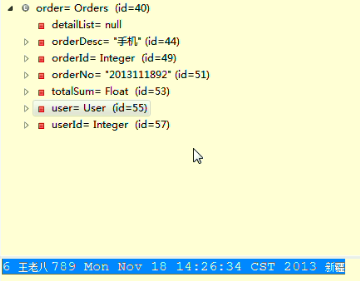
它们两个角度一样吗？不一样吧。而且实体展现形式也不同。

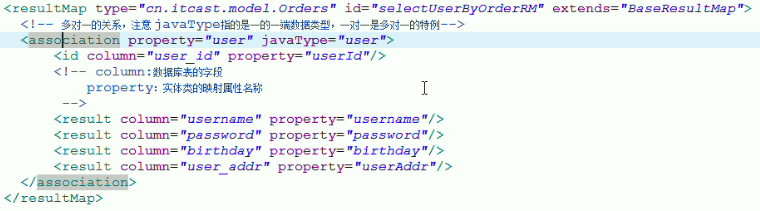


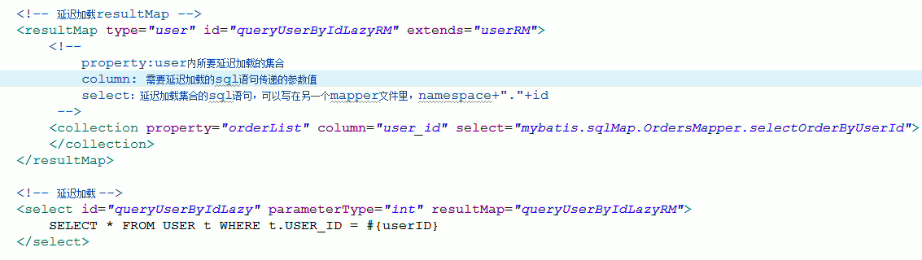












一对一呢？是不一对多的特例

### 【重点】resultType和resultMap

假设现在User对象不变，但是对应USER表中的name列的名称DBA重新设计成了username，那现在再来运行一遍测试：

可以看到这次的name属性就为null。这时候就体现出mybatis在直接使用resultType的局限性，要求属性名称必须和列的名称一致（大小写可以不一致）。在这种情况下，就必须要手动来完成数据表列和对象的映射关系了。首先来修改get方法：

<select id="get" parameterType="int" resultMap="usermapping">

SELECT \* FROM user WHERE id = #{id}

</select>

在这里，可以看到，去掉了resultType，因为resultType只能完成默认的对象类型的转换。在这里修改成了resultMap，其实这里叫做resultMapping对于学习了hibernate的童鞋来说更好理解。在这里我把结果集定义到了一个名字叫做usermapping的映射上。

下面定义usermapping：

<resultMap type="User" id="usermapping">

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

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

<result property="hireDate" column="hiredate"/>

</resultMap>

resultMap定义了一个ORM的具体映射方式。

1，type：代表O，即最终返回的对象类型

2，id：为该映射设置一个名称，这个名称就是在get或list中使用的resultMap对应的id

3，id/result：对应这属性的映射，可以参考hibernate的property。id和result的区别在于，id一般用于映射主键，可以提高速度，result一般对于普通的属性。

设置完成后，就可以将对象正常get了。

在mybatis中，其实如果仅仅只是使用了resultType，相当于mybatis自动的帮我们建立了一个resultMap，只是这个resultMap直接完成了属性和列相同名称的映射而已。所以，mybatis真正完成映射的地方就在resultMap。

之前咱们说过，如果说hibernate是面向对象为主，关系为辅，那么在mybatis中则是着重考虑的是关系模型，换句话说，如果对象模型设计的不好，就会很容易的感觉到实现的难度。

首先来看看最简单的单向many2one：

建立对象：

public class Customer {

private Long id;

private String name;

}

public class Orders {

private Long id;

private String sn;

private Double price;

private Customer customer;

}

很简单，Orders和Customer是一个单向的many2one的关系。和hibernate一样，我们先来配置简单的customer:

创建CustomerMapper.xml：

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

<!DOCTYPE mapper

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

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

<mapper namespace="cn.tedu.mybatis.domain.CustomerMapper">

<insert id="save" keyProperty="id" parameterType="customer" useGeneratedKeys="true">

insert into customer (name) values(#{name})

</insert>

<select id="get" resultType="customer" parameterType="int">

SELECT \* FROM customer WHERE id = #{id}

</select>

</mapper>

在sqlMapConfig.xml中加入

<typeAlias type="cn.tedu.mybatis.domain.Customer" alias="customer"/>

只是一个很简单的单对象映射操作。

在hibernate的many2one中我们知道，在many方的外键关联到one方的主键。在many方保存的时候，应该在对应外键保存关联one方的id。所以，many方的保存insert应该是这样：

OrdersMapper.xml：

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

<!DOCTYPE mapper

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

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

<mapper namespace="cn.tedu.mybatis.domain.OrdersMapper">

<insert id="save" keyProperty="id" parameterType="orders" useGeneratedKeys="true">

INSERT INTO orders(sn,price,customer\_id) values (#{sn},#{price},#{customer.id})

</insert>

</mapper>

在这里看到，我们在对应外键（customer\_id）的地方，使用#{customer.id}来代表orders的customer属性的id属性，mybatis会像EL那样自动帮我们得到值。所以，在这里一定要注意保存对象的顺序，必须是先保存one再保存many方。

在hibernate中，如果保存顺序有误，hibernate会在提交事务的时候同步脏数据，帮我们补一条update语句来保证对象的关系正确，但是在mybaits中，所有的sql都由我们来完成，所以没人会帮我们来完成update。所以，对应的保存测试应该是：

@Test

public void testSave(){

SqlSession session = MyBatisUtil.openSession();

try{

Customer c = new Customer();

c.setName("tedu");

Orders o = new Orders();

o.setPrice(800d);

o.setSn("001");

o.setCustomer(c);

session.insert("cn.tedu.mybatis.domain.CustomerMapper.save", c);

session.insert("cn.tedu.mybatis.domain.OrdersMapper.save", o);

session.commit();

}catch(Exception e){

e.printStackTrace();

session.rollback();

}finally{

session.close();

}

}

下面就是many方的获取了。如果只是简单的把orders配置成一个单对象，即使用resultType来映射：

<select id="get" parameterType="long" resultType="Orders">

SELECT \* FROM orders WHERE id = #{id}

</select>

如果是这样，在测试的时候：

@Test

public void testGet(){

SqlSession session = MyBatisUtil.openSession();

try{

Orders o = session.selectOne("cn.tedu.mybatis.domain.OrdersMapper.get", 1);

System.out.println(o);

Customer c=o.getCustomer();

System.out.println(c);

}catch(Exception e){

e.printStackTrace();

}finally{

session.close();

}

}

会看到，orders对应的customer是null，只能说明mybatis不能帮我们自动完成相关的关系映射。

DEBUG - ooo Using Connection [com.mysql.jdbc.JDBC4Connection@1442cd0]

DEBUG - ==> Preparing: SELECT \* FROM orders WHERE id = ?

DEBUG - ==> Parameters: 1(Integer)

cn.tedu.mybatis.domain.Orders@121f043

null

下面就来看下在mybatis中的关系映射：

第一种方式：使用额外的sql：

<?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.tedu.mybatis.domain.OrdersMapper">

<resultMap type="orders" id="ordersmap">

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

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

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

<association property="customer" column="customer\_id" javaType="customer"

select="cn.tedu.mybatis.CustomerMapper.get"/>

</resultMap>

<insert id="save" keyProperty="id" parameterType="orders" useGeneratedKeys="true">

INSERT INTO orders(sn,price,customer\_id) values (#{sn},#{price},#{customer.id})

</insert>

<select id="get" parameterType="long" resultMap="ordersmap">

SELECT \* FROM orders WHERE id = #{id}

</select>

</mapper>

注意<select id="get" parameterType="long" resultMap="ordersmap"> 原来的resultType要修改为resultMap

可能第一次看这个配置会觉得很奇怪，其实很好理解：

1，association：在mybatis中不像hibernate那样，把关系分的很细致，比如one-to-one,many-to-one，而只是把关系分成了两种：第一种是对单对象的关系，就用association，一种是集合的关系，这个待会再看。因为这里orders只是把customer对象作为自己的一个属性，所以使用association来表示这个关系；

2，property：和hibernate一样，定义这个关系在orders对象上面的对应属性。在完成关联对象的映射后，会使用这个属性把对象设置到orders对象上；

3，column：代表从查询结果集（select \* from orders where id = ?）中得到对关系维护的列，这里就指明的是外键customer\_id；

4，javaType：代表完成映射后应该返回的对象类型，这里肯定就是Orders

5，select：是这种映射方式的最关键的地方，这个select指代着cd.tedu.mybatis.customer.get这条SQL，这条sql即是得到customer的sql，而这条sql需要传入一个customer的id，我们前面已经通过column="customer\_id"指定了应该传给cd.tedu.mybatis.customer.get的值就是结果集中的customer\_id。

所以，可以看出，mybatis在这种映射的方式为：首先执行SELECT \* FROM orders WHERE o.id = #{id}，然后对id,sn,price属性直接从结果集中得到；接着，从结果集中得到customer\_id列对应的值，这个值即是orders对应customer对象的id，然后把这个值作为id交给cd.tedu.mybatis.customer.get去执行，而这个执行的结果会被拼装为一个Customer对象，接着这个对象再会被设置到orders的customer属性中。完成映射。

## 高级特性

### 延迟加载

**第一步：在mybatis核心配置文件中配置：**

lazyLoadingEnabled、aggressiveLazyLoading

|  |  |  |  |
| --- | --- | --- | --- |
| 设置项 | 描述 | 允许值 | 默认值 |
| lazyLoadingEnabled | 全局性设置懒加载。如果设为‘false’，则所有相关联的都会被初始化加载。 | true | false | false |
| aggressiveLazyLoading | 当设置为‘true’的时候，懒加载的对象可能被任何懒属性全部加载。否则，每个属性都按需加载。 | true | false | true |

**第二步：在sqlMapConfig.xml中配置**

<settings>

<!-- 打开延迟加载的开关 -->

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

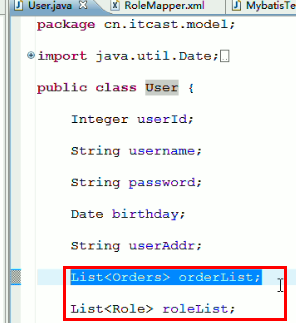
<!-- 将积极加载改为消息加载即按需加载 -->

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

</settings>

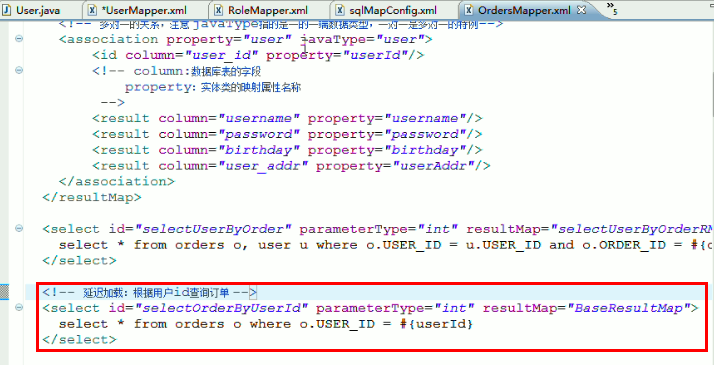
延迟加载是个什么样的概念？

用的时候，再来查



Mybatis中延迟加载比hibernate复杂，要写sql；两条SQL语句

虽然写两条，但都很简单

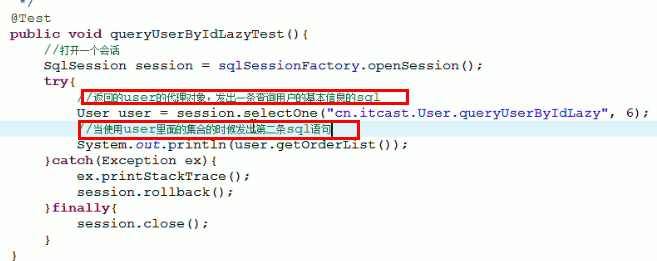




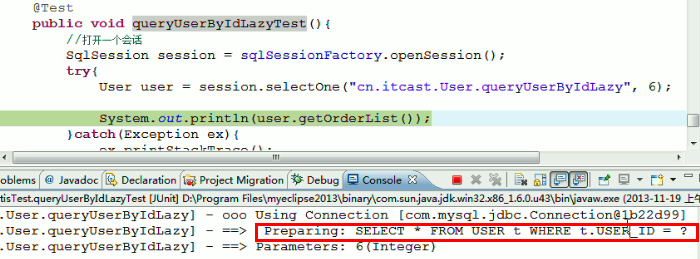
那这个collection中无需做映射了，因为在OrdersMapper.xml中已经继承了。

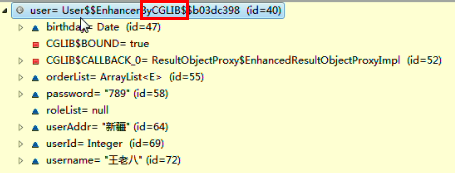






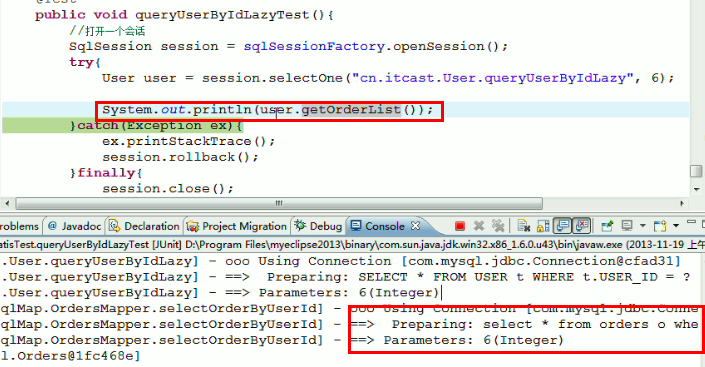
主要通过日志信息来看





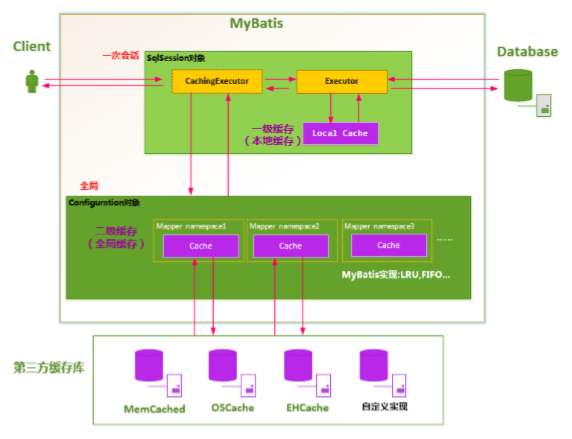
可以看出是个代理对象吧，使用CGLIB实现

这就查询了user，当F6执行下一步，也就是调用时，才执行下面SQL



一对多的延迟加载，那还有多对一的延迟加载，道理是差不多的

### 缓存架构



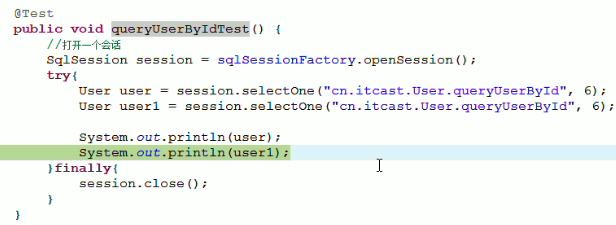
### 一级缓存

Hibernate有几个缓存啊？

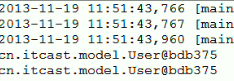
三个，一级、二级、查询

Hibernate一级缓存是Session级缓存

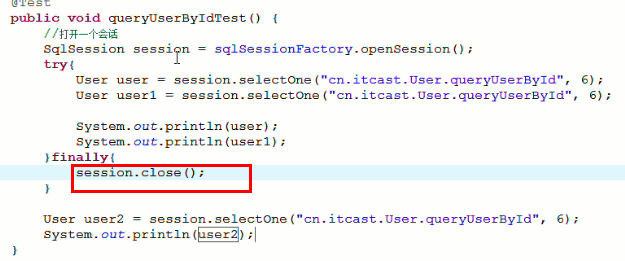
Mybatis也有类似的缓存，sqlSession级的



执行第一条，打印sql；执行第二条，没打印。说明缓存生效



对象地址相同，那说明我们使用了缓存，对吧



我们下面user2使用了session。可上面就关闭了，执行是不就报错了。

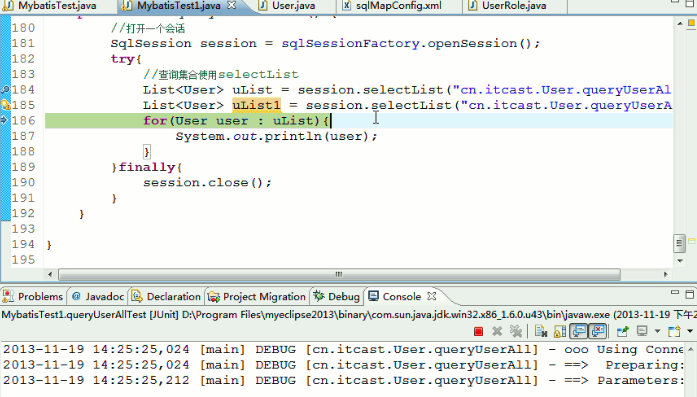


那我们再打开，你说会怎么执行？

第二句会不会发SQL

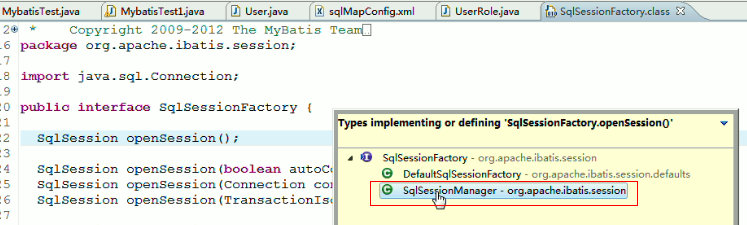


上面是单个对象，下面我们测试下list

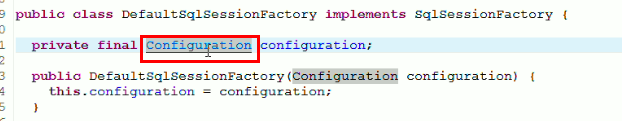


执行第一条，打印SQL；第二条不打印，说明它去缓存中取了

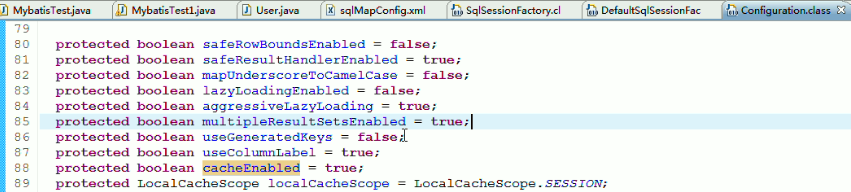
看看源码



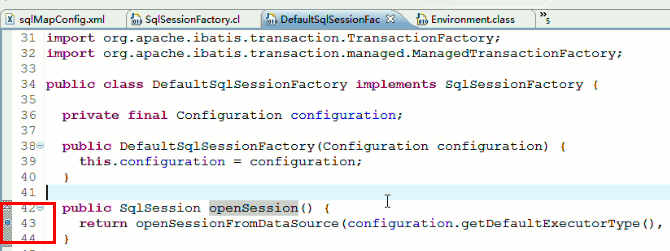
看SqlSession的实现类。

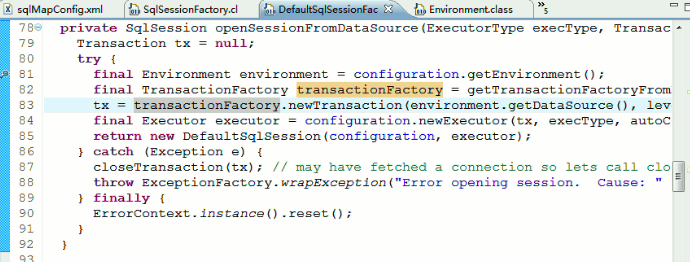


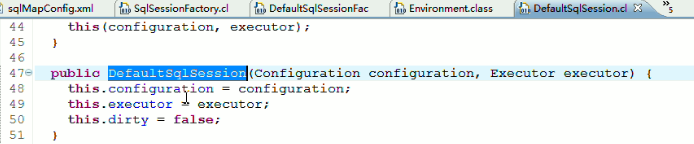
加载项

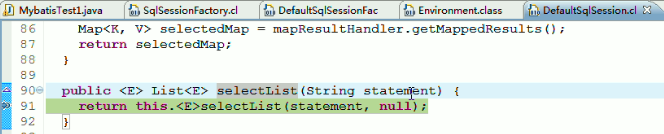


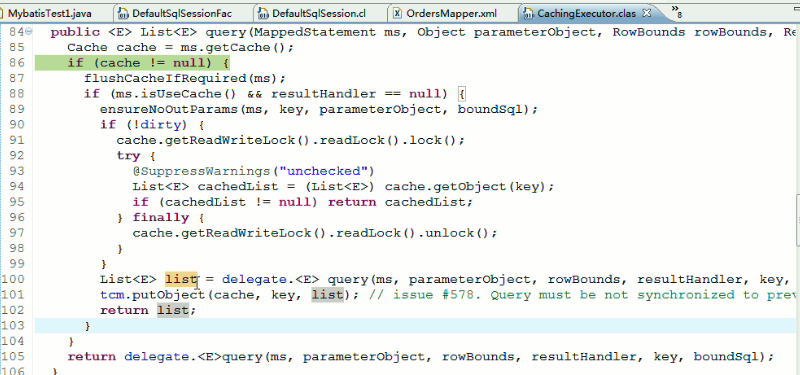
初始化sqlMapCongif.xml时，就全部进行初始化



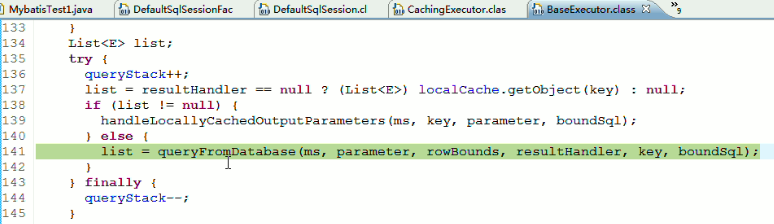








你看它这有个缓存判断，如果有就从缓存中读取。



真正是执行这句，从数据库获取数据

一级缓存不用开启，默认就在。

### 二级缓存

Hibernate中二级缓存是SessionFactory级别， 那mybatis也一样，也是SessionFactory级别的。一个项目中肯定会存在很多共用的查询数据，对于这一部分的数据，没必要

每一个用户访问时都去查询数据库，因此配置二级缓存将是非常必要的。

更准确的说Mybatis二级缓存是在同一个namespace下共享。

注意：PO对象必须序列化

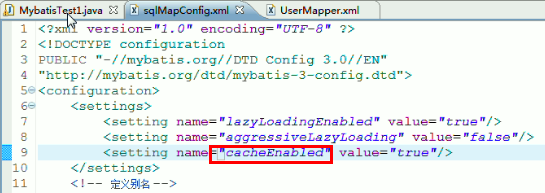
当内存不够用时，二级缓存支持写磁盘，所以PO类必须序列化。如果没有序列化，它底层会报错。

Mybatis的二级缓存配置相当容易，要开启二级缓存：

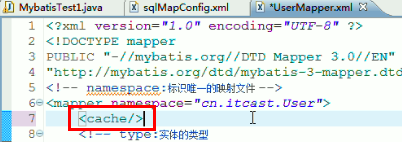
**第一步：在核心配置文件中加入**

<setting name="cacheEnabled" value="true"/>

默认开启



**第二步：要在你的Mapper映射文件中添加一行：  <cache />**



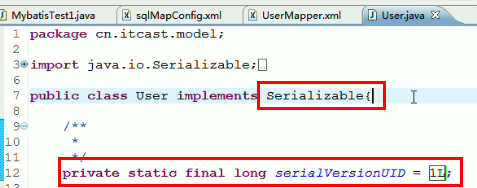
二级缓存可以精确到每一条SQL语句上

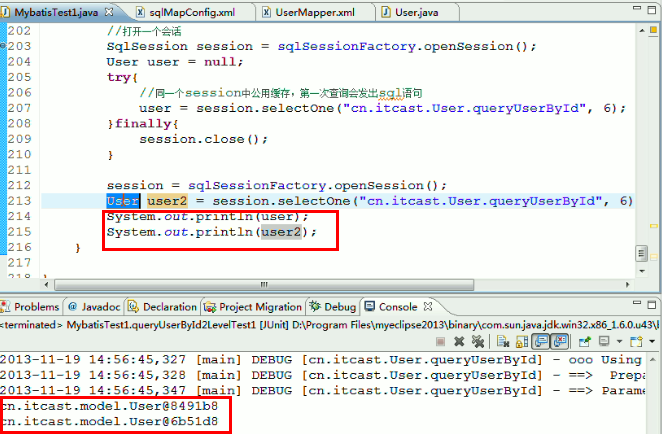
**第三步：在select语句中useCache=false**

可以禁用当前的语句的二级缓存，即每次查询夸session 的查询都会发出sql去查询，默认情况是true，即该sql使用二级缓存。

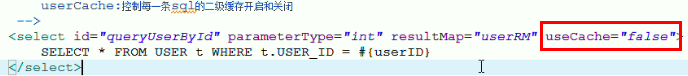
二级缓存实例化接口







二级缓存生效，第二句没有打印SQL。但由于是不同的sqlsession，产生对象不同，但数据相同。和hibernate不同。Hibernate是同一个对象。这跟二级缓存实现机制也相关，ehcache就地址相同。



执行第一条查询数据库

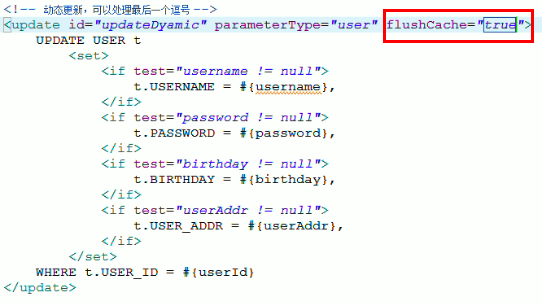
然后利用客户端修改数据

第二条它从缓存中拿，造成数据脏读。

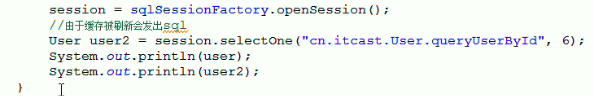
这就是缓存的弊端。

那客户端会有这样的隐患，那程序呢？mybatis呢？

下面就是解决方案







那上面如果使用一个session

也就是使用一级缓存

那先查询，执行SQL；

修改数据，执行SQL；

再次查询，执行SQL；

它们使用一样的，唯一不同是执行范围不同。



4. 映射文件中所有的insert、update和delete语句将刷新缓存，三个语句中有flushCache=*"true"* 属性，默认情况下为true，即同个sessionFactory中当有sql语句更新时缓存自动被刷新以保证数据的实时性，如果改成false则不会刷新。使用缓存时如果手动修改数据库表中的查询数据会出现脏读， 缓存将使用LRU（Least Recently Used）最近最少使用策略算法来回

<cache  eviction="FIFO"  flushInterval="60000"  size="512"  readOnly="true"/>

这个更高级的配置创建了一个 FIFO 缓存,并每隔 60 秒刷新,存数结果对象或列表的 512 个引用,而且返回的对象被认为是只读的,因此在不同线程中的调用者之间修改它们会 导致冲突。可用的收回策略有, 默认的是 LRU:

1. LRU – 最近最少使用的:移除最长时间不被使用的对象。
2. FIFO – 先进先出:按对象进入缓存的顺序来移除它们。
3. SOFT – 软引用:移除基于垃圾回收器状态和软引用规则的对象。
4. WEAK – 弱引用:更积极地移除基于垃圾收集器状态和弱引用规则的对象。

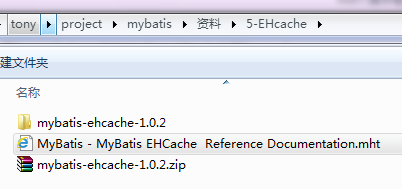
flushInterval(刷新间隔)可以被设置为任意的正整数,而且它们代表一个合理的毫秒 形式的时间段。默认情况是不设置,也就是没有刷新间隔,缓存仅仅调用语句时刷新。

size(引用数目)可以被设置为任意正整数,要记住你缓存的对象数目和你运行环境的 可用内存资源数目。默认值是1024。

readOnly(只读)属性可以被设置为 true 或 false。只读的缓存会给所有调用者返回缓存对象的相同实例。因此这些对象不能被修改。这提供了很重要的性能优势。可读写的缓存 会返回缓存对象的拷贝(通过序列化) 。这会慢一些,但是安全,因此默认是 false。

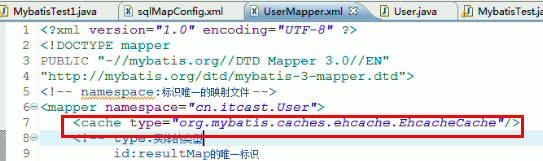
readOnly=true可以不序列化，=false必须序列化

### EHcache第三方缓存插件



加入mybatis-ehcache-1.0.2.jar和lib/ehcache-core-2.6.5.jar文件

复制\lib\ehcache-core-2.6.5下的ehcache-failsafe.xml到calsspath下，修改文件名为ehcache.xml，因为默认加载为此名称。





检查它有无发送两条SQL语句就OK了

ehcache.xml

defaultCache配置说明：

maxElementsInMemory 内存中最大缓存对象数.当超过最大对象数的时候,ehcache会按指定的策略去清理内存

eternal 缓存对象是否永久有效,一但设置了,timeout将不起作用.

timeToIdleSeconds 设置Element在失效前的允许闲置时间.仅当element不是永久有效时使用,可选属性,默认值是0,也就是可闲置时间无穷大.

timeToLiveSeconds：设置Element在失效前允许存活时间.最大时间介于创建时间和失效时间之间.仅当element是永久有效时使用,默认是0.,也就是element存活时间无穷大.

overflowToDisk 配置此属性,当内存中Element数量达到maxElementsInMemory时,Ehcache将会Element写到磁盘中.

diskSpoolBufferSizeMB 这个参数设置DiskStore(磁盘缓存)的缓存区大小.默认是30MB.每个Cache都应该有自己的一个缓冲区.

maxElementsOnDisk 磁盘中最大缓存对象数,若是0表示无穷大.

diskPersistent 是否在重启服务的时候清楚磁盘上的缓存数据.true不清除.

diskExpiryThreadIntervalSeconds 磁盘失效线程运行时间间隔.

memoryStoreEvictionPolicy：当达到maxElementsInMemory限制时,Ehcache将会根据指定的策略去清理内存.默认策略是LRU(最近最少使用).你可以设置为FIFO(先进先出)或是LFU(较少使用).

### mybatis拦截器

mapperedstatement，执行器executor，转换参数，拼接成sql，然后执行；执行后返回结果集，将结果集封装到pojo对象中。

在执行sql前，和封装到pojo对象前

好处：

1. 性能日志，SQL执行的性能

真实项目中性能瓶颈在哪里？

1. 业务逻辑，大量计算 springMVC/struts2 拦截器性能监控
2. 数据库查询统计SQL mybatis拦截器性能监控（主要：关联层级（打断，冗余）+表数据过多（分表））

**第一步：在sqlMapConfig.xml中声明拦截器插件**

<plugins>

<plugin interceptor="cn.tedu.interceptor.ExamplePlugin">

</plugin>

</plugins>

**第二步：实现interceptor接口**

拦截SQL执行，对SQL进行加工；实际项目开发中比较少用。主要用于通用分页（支持多种数据库（Oracle/MySQL/SqlServer），利用SQL进行分页，提高性能），SQL监控。

例如：监控SQL的执行时间，发现SQL执行瓶颈进行优化。

package cn.tedu.mybatis.interceptor;

import java.text.DateFormat;

import java.util.Date;

import java.util.List;

import java.util.Locale;

import java.util.Properties;

import org.apache.ibatis.executor.Executor;

import org.apache.ibatis.mapping.BoundSql;

import org.apache.ibatis.mapping.MappedStatement;

import org.apache.ibatis.mapping.ParameterMapping;

import org.apache.ibatis.plugin.Interceptor;

import org.apache.ibatis.plugin.Intercepts;

import org.apache.ibatis.plugin.Invocation;

import org.apache.ibatis.plugin.Plugin;

import org.apache.ibatis.plugin.Signature;

import org.apache.ibatis.reflection.MetaObject;

import org.apache.ibatis.session.Configuration;

import org.apache.ibatis.session.ResultHandler;

import org.apache.ibatis.session.RowBounds;

import org.apache.ibatis.type.TypeHandlerRegistry;

@Intercepts({

@Signature(type = Executor.class, method = "update", args = {

MappedStatement.class, Object.class }),

@Signature(type = Executor.class, method = "query", args = {

MappedStatement.class, Object.class, RowBounds.class,

ResultHandler.class }) })

public class SqlTimeInterceptor implements Interceptor {

private Properties properties;

public Object intercept(Invocation invocation) throws Throwable {

MappedStatement mappedStatement = (MappedStatement) invocation

.getArgs()[0];

Object parameter = null;

if (invocation.getArgs().length > 1) {

parameter = invocation.getArgs()[1];

}

String sqlId = mappedStatement.getId();

BoundSql boundSql = mappedStatement.getBoundSql(parameter);

Configuration configuration = mappedStatement.getConfiguration();

Object returnValue = null;

long start = System.currentTimeMillis();

returnValue = invocation.proceed();

long end = System.currentTimeMillis();

long time = (end - start);

if (time > 1) {

String sql = getSql(configuration, boundSql, sqlId, time);

System.err.println(sql);

}

return returnValue;

}

public static String getSql(Configuration configuration, BoundSql boundSql,

String sqlId, long time) {

String sql = showSql(configuration, boundSql);

StringBuilder str = new StringBuilder(100);

str.append(sqlId);

str.append(":");

str.append(sql);

str.append(":");

str.append(time);

str.append("ms");

return str.toString();

}

private static String getParameterValue(Object obj) {

String value = null;

if (obj instanceof String) {

value = "'" + obj.toString() + "'";

} else if (obj instanceof Date) {

DateFormat formatter = DateFormat.getDateTimeInstance(

DateFormat.DEFAULT, DateFormat.DEFAULT, Locale.CHINA);

value = "'" + formatter.format(new Date()) + "'";

} else {

if (obj != null) {

value = obj.toString();

} else {

value = "";

}

}

return value;

}

public static String showSql(Configuration configuration, BoundSql boundSql) {

Object parameterObject = boundSql.getParameterObject();

List<ParameterMapping> parameterMappings = boundSql

.getParameterMappings();

String sql = boundSql.getSql().replaceAll("[\\s]+", " ");

if (parameterMappings.size() > 0 && parameterObject != null) {

TypeHandlerRegistry typeHandlerRegistry = configuration

.getTypeHandlerRegistry();

if (typeHandlerRegistry.hasTypeHandler(parameterObject.getClass())) {

sql = sql.replaceFirst("\\?",

getParameterValue(parameterObject));

} else {

MetaObject metaObject = configuration

.newMetaObject(parameterObject);

for (ParameterMapping parameterMapping : parameterMappings) {

String propertyName = parameterMapping.getProperty();

if (metaObject.hasGetter(propertyName)) {

Object obj = metaObject.getValue(propertyName);

sql = sql.replaceFirst("\\?", getParameterValue(obj));

} else if (boundSql.hasAdditionalParameter(propertyName)) {

Object obj = boundSql

.getAdditionalParameter(propertyName);

sql = sql.replaceFirst("\\?", getParameterValue(obj));

}

}

}

}

return sql;

}

public Object plugin(Object target) {

return Plugin.wrap(target, this);

}

public void setProperties(Properties properties0) {

this.properties = properties0;

}

}

## 工具：自动生成代码-eclipse插件

jdbcType 当这个值为null时，告诉jdbc驱动程序，这个值当null时应该按什么类型来处理。

Mysql驱动不需要，Oracle驱动底层不能识别，所以必须指定jdbcType，不指定报错。

只有在修改时，才需要指定。

利用这个工具产生insert的全字段的代码；update使用动态SQL语句的；select一般直接从insert的语句中拷贝字段；delete语句直接拷贝

### 第一步：安装插件

将给定的plugins和feaures目录直接拷贝到D:\java\ide\eclipse\dropins

重启eclipse。

### 第二步：配置文件

制定mysql的驱动包的路径 千万别放中文路径下

配置数据源和生成的代码所存放的位置

创建一个配置文件sqlMapGeneractor.xml

这个文件名称随便起，它会自动识别你文件的头信息

<!DOCTYPE generatorConfiguration。

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

<!DOCTYPE generatorConfiguration PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN" "http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd" >

<generatorConfiguration>

<!-- 配置属性文件，这样有变更只需改配置文件 -->

<properties resource="generatorConfig.properties"/>

<!-- 制定mysql的驱动包的路径 千万别放中文路径下 -->

<classPathEntry location="D:\javaenv\all\_jar\database\mysql-connector-java-5.1.26.jar" />

<!-- 配置数据源和生成的代码所存放的位置 -->

<context id="tedu">

<!-- 是否生成注释 true不生成 false生成 -->

<commentGenerator>

<property name="suppressAllComments" value="${suppressAllComments}"/>

</commentGenerator>

<jdbcConnection driverClass="${driverClass}" connectionURL="${url}" userId="${username}" password="${password}" />

<!-- 所生成的实体类的位置默认资源包src -->

<javaModelGenerator targetPackage="${modeltargetPackage}" targetProject="${targetProject}" />

<!-- 所生成的sqlMap的影射文件的位置,默认资源包src -->

<sqlMapGenerator targetPackage="${sqltargetPackage}" targetProject="${targetProject}" />

<javaClientGenerator targetPackage="${clienttargetPackage}" targetProject="${targetProject}" type="XMLMAPPER" />

<!-- 为哪些表生成代码 tableName:表名 schema:不用填写 -->

<table schema="" tableName="orders" />

<table schema="" tableName="department" />

<table schema="" tableName="employee" />

</context>

</generatorConfiguration>

配置属性文件，控制生成的代码，方便更改配置generatorConfig.properties

driverClass=com.mysql.jdbc.Driver

url=jdbc:mysql://localhost:3306/mybatisdb?characterEncoding=utf-8

username=root

password=root

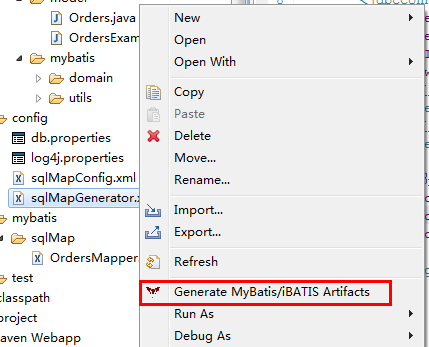
suppressAllComments=true

targetProject=cd

modeltargetPackage=cn.tedu.model

sqltargetPackage=cn.tedu.sqlMap

clienttargetPackage=cn.tedu.client



点击菜单运行即可。

我们可能会担心一旦重新执行generate的时候，我们自己编写的代码会不会丢失，不会的，插件不会修改或丢弃我们自己编写的代码。

一旦掌握了插件如何使用，重要的工作就是如何使用XXXExample类了。这种方式，完全不用编写繁琐的mapper xml文件。