Mybatis系列【公众号：马士兵老师】.md

## Mybatis系列第1篇：mybatis未出世之前我们那些痛苦的经历

### 本篇内容

1. java操作数据库相关的各种技术介绍
2. 这么多技术，如何选择？
3. 这么多技术，为什么我们选择的是mybatis

不知道大家是否还记得使用jdbc如何操作数据库？ 加载驱动、获取连接、拼接sql、执行sql、获取结果、解析结果、关闭数据库，这些操作是纯jdbc的方式必经的一些过程，每次操作数据库都需要写这些，但是真正和开发相关有：拼接sql、执行sql、解析结果，这几个可能不同，而其他的几个步骤是所有db操作共用的一些步骤，所以纯jdbc的方式重复性的代码太多了，如果每个操作都使用这种方式会耗费大量的时间。 纯jdbc的方式，sql多数都是写在java代码中，我还记得刚开始工作的时候接触的第一个项目就是jdbc写的，当时jdbc是写在jsp中，调试和维护起来是相当麻烦的。 还有就是动态sql方面：面对于各种复杂的条件查询，需要在代码中写很多判断，拼接sql，这种情况估计大家都有经历过，应该是有同感的，各种判断各种拼接没有统一的规范，各种写法也是千奇百怪，最终导致系统难以维护。 上面这些问题使系统开发缓慢、难以维护，后面就出现了各种优秀的框架来解决这些问题，下面我们就来看一下常见的db框架及他们的优缺点，以及我们应该如何选择？

### Hibernate

这个框架学过，但是实际工作中所有接触的项目都未曾使用过hibernate。

介绍hibernate之前，我们先了解一下什么是ORM？

ORM（Object Relational Mapping）：对象关系映射，简单点说就是将数据库中的表和java中的对象建立映射关系，可以让我们操作对象来间接的操作数据库。

ORM最好的框架就是hibernate，hibernate可以让你通过java对象来间接的操作数据库，java对象就相当于数据库中表的代理一样，当你想删除表的数据的时候，不需要自己写delete语句发给数据库，只需要对hibernate说我需要删除哪个java对象就行了，hibernate自动根据你的操作去生成db需要的sql然后发给db去执行，对于开发者来说隐藏了底层jdbc和db的交互过程，可能开发者不需要掌握数据库技术，就可以通过这个框架直接操作数据库，比如对数据库进行增、删、改、查可能需要开发者会写各种sql脚本，还有每种数据库的sql脚本语法也不一样，刚开始项目使用的是mysql，你按照mysql的语法写的，后面可能要切换到oracle，语法上面需要变动，但是如果你使用hibernate，这些都不是问题，你不会sql也没关系，而你只需要像操作对象一样去操作数据库，hibernate会根据你的操作和db的类型自动生成操作所需要的sql，一种写法能够跨多种数据库运行。

#### 优点

1. 简化了整个jdbc操作过程
2. 对于开发者来说不需要关心sql了，只需要去操作对象就可以了，hibernate可以帮我们自动生成所需要的sql
3. 代码移植性比较好，通过hibernate操作db都是通过操作对象来进行的，而hibernate会根据我们的操作和db的类型生成符合各种db要求的sql，如果我们需要切换db的类型，hibernate会自动适应，对于开发者业务代码来说不需要做任何业务代码上的调整
4. 开发效率比较高

#### 缺点

1. sql优化比较艰难，各种操作最终发给db的sql是由hibernate自动生成的，对于开发者来说如果想干预最终需要执行的sql，相对来说比较困难
2. hibernate入门比较容易，但是想成为高手学习成本比较高
3. 对于复杂的动态sql，代码中也需要写很多判断进行组装，动态sql这块的支持比较欠缺

如果做一些简单的系统，开发周期也比较紧急，对sql的优化要求也不是很高，可以使用hibernate。

### JdbcTemplate

jdbctemplate是在spring框架的基础上开发的一个jdbc框架，所以对spring是有依赖的，它对jdbc做了封装，隐藏了各种重复的操作，使用时只需传入：需要执行的sql、参数以及对于结果如何解析的程序就可以了，使用起来还是很方便的，但是面对与动态sql，它也是无能为力了。整体上来说，jdbctemplate相对于纯jdbc隐藏了很多重复性的操作，对于sql的写法和结果的组装上完全交给了开发者自己去控制，在系统中使用也可以帮助我们节约很多时间，而且学习相当简单，花2个小时就学会了，也是很优秀的一个框架。

### Mybatis

mybatis相对于纯jdbc来说，也隐藏了重复性的工作，mybatis是一个半自动化的orm框架，为什么说是半自动化的呢，因为他需要我们自己去写sql，而他做的更好的地方就是动态sql的支持上面，而上面说的各种技术，面对与动态sql只能自己写很多判断去组装sql，而这些判断和组装在mybatis中实现起来就非常简单了，完全由mybatis去帮我们实现了。mybatis将sql交由开发者去控制，所以在sql的优化方面，开发者可以随心所欲，也就是说mybatis将重复性的工作优化到了极致：操作db的过程、动态sql的拼装、结果和对象的映射，这些mybatis都帮我们实现的很好，而让我们将更多的经历花在sql的写法和优化上面，所以毫无疑问mybatis使用人数和公司也是最多的，大部分互联网公司基本上都会使用mybatis，其他2种可以不会，但是mybatis你必须要会。

### 几种技术的对比

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 纯JDBC | Hibernate | JdbcTemplate | Mybatis |
| 代码重复度 | 高 | 低 | 低 | 低 |
| 动态sql支持度 | 低 | 低 | 低 | 高 |
| sql控制度 | 高 | 低 | 低 | 高 |
| 学习成本 | 低 | 高 | 低 | 低 |
| 开发速度 | 慢 | 高 | 高 | 高 |
| 使用的公司 | 少 | 比较多 | 比较多 | 最多 |

**后面的文章我们将正式详解mybatis所有知识点，理论+实战的方式，带领大家成为mybatis高手，请大家关注！**

## Mybatis系列第2篇：mybatis入门，你确定mybatis你玩的很溜？

### 本篇技术栈

1. mysql5.7.25
2. maven3.6.1
3. jdk1.8
4. idea

### 本篇主要内容

1. 通过一个案例感受一下mybatis的强大之处
2. mybatis开发项目的具体步骤
3. 介绍mybatis中主要的几个对象

我们先来一个案例，让大家感受一下mybatis是多么的牛逼，我相信大家看了案例之后，会强烈的激发大家学习mybatis的兴趣。

### 案例：原来ibatis是这么强大

下面的案例，大家先不用关系代码为什么这么写，先感受一下效果，后面我们再来细说代码，**案例代码文章尾部有获取方式**。

#### 准备数据库

mysql中运行下面脚本：

/\*创建数据库javacode2018\*/  
DROP DATABASE IF EXISTS `javacode2018`;  
CREATE DATABASE `javacode2018`;  
USE `javacode2018`;  
  
/\*创建表结构\*/  
DROP TABLE IF EXISTS `t\_user`;  
CREATE TABLE t\_user (  
 id BIGINT AUTO\_INCREMENT PRIMARY KEY COMMENT '主键，用户id，自动增长',  
 `name` VARCHAR(32) NOT NULL DEFAULT '' COMMENT '姓名',  
 `age` SMALLINT NOT NULL DEFAULT 1 COMMENT '年龄',  
 `salary` DECIMAL(12,2) NOT NULL DEFAULT 0 COMMENT '薪水'  
) COMMENT '用户表';  
  
SELECT \* FROM t\_user;

上面脚本中，创建了一个javacode2018数据库，然后创建了一个用户表，里面有4个字段，id为主键自动增长。

#### 我们的需求

对t\_user表，我们有以下这些需求：

1. 实现一个通用的插入操作：支持动态插入，可以根据传入的字段的值，动态生成所需要的各种insert语句
2. 批量插入功能
3. 实现一个通用的更新操作：支持动态更新操作，可以根据传入的字段，动态生成所需要的各种update语句
4. 实现一个通用的查询操作：支持各种组合条件查询、支撑排序、分页、支持返回列的控制等各种复杂的查询需求

下面我们就来一个案例，将上面这些需求通过mybatis实现，先见证一下mybatis的强大之处。

#### 创建maven项目

idea中创建maven项目，这个操作在maven系列中已经说过很多次了，以后大部分项目都是通过maven来玩的，maven这方面玩的不是太溜的，可以跳到文章尾部去看一下maven系列的文章。

项目采用maven中聚合及继承的方式来管理。

##### 创建父项目

先创建父项目mybatis-series，父项目的坐标信息：

<groupId>com.javacode2018</groupId>  
<artifactId>mybatis-series</artifactId>  
<version>1.0-SNAPSHOT</version>

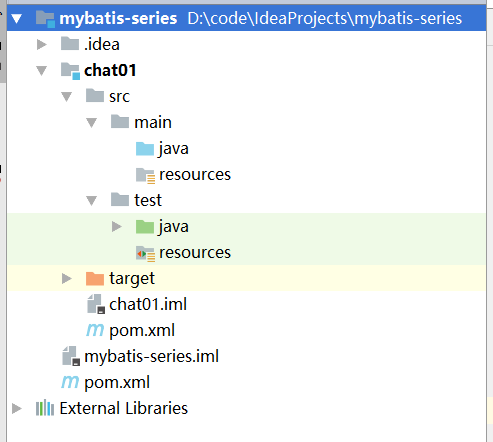
##### 创建子项目

创建一个子模块chat01，子模块的坐标信息：

<groupId>com.javacode2018</groupId>  
<artifactId>chat01</artifactId>  
<version>1.0-SNAPSHOT</version>

##### 项目结构

如下图：



#### 引入mybatis依赖

mybatis-series/pom.xml内容如下：

<?xml version="1.0" encoding="UTF-8"?>  
<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>com.javacode2018</groupId>  
 <artifactId>mybatis-series</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 <packaging>pom</packaging>  
  
 <modules>  
 <module>chat01</module>  
 </modules>  
  
 <properties>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 <!-- 配置maven编译的时候采用的编译器版本 -->  
 <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>  
 <!-- 指定源代码是什么版本的，如果源码和这个版本不符将报错，maven中执行编译的时候会用到这个配置，默认是1.5，这个相当于javac命令后面的-source参数 -->  
 <maven.compiler.source>1.8</maven.compiler.source>  
 <!-- 该命令用于指定生成的class文件将保证和哪个版本的虚拟机进行兼容，maven中执行编译的时候会用到这个配置，默认是1.5，这个相当于javac命令后面的-target参数 -->  
 <maven.compiler.target>1.8</maven.compiler.target>  
 <mybatis.version>3.5.3</mybatis.version>  
 <mysql.version>5.1.47</mysql.version>  
 <lombok.version>1.18.10</lombok.version>  
 </properties>  
  
 <dependencyManagement>  
 <dependencies>  
 <dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis</artifactId>  
 <version>${mybatis.version}</version>  
 </dependency>  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>${mysql.version}</version>  
 </dependency>  
 <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <version>${lombok.version}</version>  
 <scope>provided</scope>  
 </dependency>  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.12</version>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>ch.qos.logback</groupId>  
 <artifactId>logback-classic</artifactId>  
 <version>1.2.3</version>  
 <scope>test</scope>  
 </dependency>  
 </dependencies>  
 </dependencyManagement>  
  
</project>

chat01/pom.xml内容如下：

<?xml version="1.0" encoding="UTF-8"?>  
<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">  
 <parent>  
 <artifactId>mybatis-series</artifactId>  
 <groupId>com.javacode2018</groupId>  
 <version>1.0-SNAPSHOT</version>  
 </parent>  
 <modelVersion>4.0.0</modelVersion>  
 <artifactId>chat01</artifactId>  
  
 <dependencies>  
 <dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>ch.qos.logback</groupId>  
 <artifactId>logback-classic</artifactId>  
 </dependency>  
 </dependencies>  
  
</project>

上面我们引入了mybatis需要的包、mysql jdbc驱动、lombok、单元测试需要的junit包、日志输出需要的logback包。

这里lombok可能大家没有用过，这个东西可以自动帮我们生成javabean的一些代码，比如get、set方法，可以节省开发编写代码的量，这个以后有空了写一篇文章来介绍。

#### 配置logback

mybatis在运行过程中会输出一些日志，比如sql信息、sql的参数信息、执行的结果等信息，mybatis中会通过logback输出出来。

在chat01/src/main/resources目录中新建文件logback.xml，内容如下：

<?xml version="1.0" encoding="UTF-8"?>  
<configuration>  
 <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
 <encoder>  
 <pattern>%d{mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>  
 </encoder>  
 </appender>  
  
 <logger name="com.javacode2018" level="debug" additivity="false">  
 <appender-ref ref="STDOUT" />  
 </logger>  
</configuration>

#### 创建mybatis相关文件

##### user.xml

chat01/src/main/resources目录中新建user.xml，内容如下：

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper namespace="com.javacode2018.mybatis.chat01.UserMapper">  
  
 <!-- 插入 -->  
 <insert id="insert" parameterType="com.javacode2018.mybatis.chat01.UserModel" keyProperty="id" useGeneratedKeys="true">  
 <![CDATA[ INSERT INTO `t\_user` ]]>  
 <trim prefix="(" suffix=")" suffixOverrides=",">  
 <if test="id!=null">  
 <![CDATA[ `id`, ]]>  
 </if>  
 <if test="name!=null">  
 <![CDATA[ `name`, ]]>  
 </if>  
 <if test="age!=null">  
 <![CDATA[ `age`, ]]>  
 </if>  
 <if test="salary!=null">  
 <![CDATA[ `salary`, ]]>  
 </if>  
 </trim>  
 <![CDATA[ VALUES ]]>  
 <trim prefix="(" suffix=")" suffixOverrides=",">  
 <if test="id!=null">  
 <![CDATA[ #{id}, ]]>  
 </if>  
 <if test="name!=null">  
 <![CDATA[ #{name}, ]]>  
 </if>  
 <if test="age!=null">  
 <![CDATA[ #{age}, ]]>  
 </if>  
 <if test="salary!=null">  
 <![CDATA[ #{salary}, ]]>  
 </if>  
 </trim>  
 </insert>  
  
<!-- 批量插入 -->  
<insert id="insertBatch" parameterType="map">  
 <![CDATA[ INSERT INTO `t\_user` (`id`, `name`, `age`, `salary`) VALUES ]]>  
 <foreach collection="list" separator="," item="item">  
 (#{item.id}, #{item.name}, #{item.age}, #{item.salary})  
 </foreach>  
</insert>  
  
 <!-- 更新 -->  
 <update id="update" parameterType="com.javacode2018.mybatis.chat01.UserModel">  
 <![CDATA[ UPDATE `t\_user` ]]>  
 <set>  
 <if test="name!=null">  
 <![CDATA[ `name` = #{name}, ]]>  
 </if>  
 <if test="age!=null">  
 <![CDATA[ `age` = #{age}, ]]>  
 </if>  
 <if test="salary!=null">  
 <![CDATA[ `salary` = #{salary}, ]]>  
 </if>  
 </set>  
 <where>  
 <if test="id!=null">  
 <![CDATA[ AND `id` = #{id} ]]>  
 </if>  
 </where>  
 </update>  
  
 <!-- 更新 -->  
 <update id="updateByMap" parameterType="map">  
 <![CDATA[ UPDATE `t\_user` ]]>  
 <set>  
 <if test="name!=null">  
 <![CDATA[ `name` = #{name}, ]]>  
 </if>  
 <if test="age!=null">  
 <![CDATA[ `age` = #{age}, ]]>  
 </if>  
 <if test="salary!=null">  
 <![CDATA[ `salary` = #{salary}, ]]>  
 </if>  
 </set>  
 <where>  
 <if test="id!=null">  
 <![CDATA[ AND `id` = #{id} ]]>  
 </if>  
 </where>  
 </update>  
  
 <!-- 删除 -->  
 <delete id="delete" parameterType="map">  
 <![CDATA[  
 DELETE FROM `t\_user`  
 ]]>  
 <where>  
 <if test="id!=null">  
 <![CDATA[ AND `id` = #{id} ]]>  
 </if>  
 </where>  
 </delete>  
  
  
 <!-- 查询记录 -->  
 <select id="getModelList" parameterType="map" resultType="com.javacode2018.mybatis.chat01.UserModel">  
 <![CDATA[  
 SELECT  
 ]]>  
 <choose>  
 <when test="tableColumnList!=null and tableColumnList.size() >= 1">  
 <foreach collection="tableColumnList" item="item" separator=",">  
 <![CDATA[ ${item} ]]>  
 </foreach>  
 </when>  
 <otherwise>  
 <![CDATA[  
 `id`,  
 `name`,  
 `age`,  
 `salary`  
 ]]>  
 </otherwise>  
 </choose>  
 <![CDATA[  
 FROM  
 `t\_user` a  
 ]]>  
 <where>  
 <if test="id!=null and id.toString()!=''">  
 <![CDATA[ AND a.`id` = #{id} ]]>  
 </if>  
 <if test="idList!=null and idList.size() >= 1">  
 <![CDATA[ AND a.`id` IN ]]>  
 <foreach collection="idList" item="item" open="(" separator="," close=")">  
 <![CDATA[ #{item} ]]>  
 </foreach>  
 </if>  
 <if test="name!=null and name.toString()!=''">  
 <![CDATA[ AND a.`name` = #{name} ]]>  
 </if>  
 <if test="age!=null and age.toString()!=''">  
 <![CDATA[ AND a.`age` = #{age} ]]>  
 </if>  
 <if test="salary!=null and salary.toString()!=''">  
 <![CDATA[ AND a.`salary` = #{salary} ]]>  
 </if>  
 <if test="nameLike!=null and nameLike.toString()!=''">  
 <![CDATA[ AND a.`name` like '%${nameLike}%' ]]>  
 </if>  
 <if test="salaryGte!=null and salaryGte.toString()!=''">  
 <![CDATA[ AND a.`salary` >= #{salaryGte} ]]>  
 </if>  
 </where>  
 <if test="sort!=null and sort.toString()!=''">  
 <![CDATA[ order by ${sort} ]]>  
 </if>  
 <if test="skip!=null and pageSize!=null">  
 <![CDATA[ LIMIT #{skip},#{pageSize} ]]>  
 </if>  
 </select>  
  
</mapper>

##### mybatis-config.xml

chat01/src/main/resources目录中新建mybatis-config.xml，内容如下：

<?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>  
 <properties>  
 <property name="driver" value="com.mysql.jdbc.Driver"/>  
 <property name="url" value="jdbc:mysql://localhost:3306/javacode2018?characterEncoding=UTF-8"/>  
 <property name="username" value="root"/>  
 <property name="password" value="root123"/>  
 </properties>  
 <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="mapper/user.xml"/>  
 </mappers>  
</configuration>

##### UserMapper接口

package com.javacode2018.mybatis.chat01;  
  
import java.util.List;  
import java.util.Map;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
public interface UserMapper {  
  
 /\*\*  
 \* 插入用户信息  
 \*  
 \* @param userModel  
 \* @return  
 \*/  
 void insert(UserModel userModel);  
  
 /\*\*  
 \* 批量插入用户信息  
 \*  
 \* @param userModelList  
 \*/  
 void insertBatch(List<UserModel> userModelList);  
  
 /\*\*  
 \* 更新用户信息  
 \*  
 \* @param userModel  
 \* @return  
 \*/  
 int update(UserModel userModel);  
  
 /\*\*  
 \* 通过map来更新用户记录  
 \*  
 \* @param map  
 \* @return  
 \*/  
 int updateByMap(Map<String, Object> map);  
  
 /\*\*  
 \* 通过map来删除用户记录  
 \*  
 \* @param map  
 \* @return  
 \*/  
 int delete(Map<String, Object> map);  
  
 /\*\*  
 \* 查询用户列表  
 \*  
 \* @param map  
 \* @return  
 \*/  
 List<UserModel> getModelList(Map<String, Object> map);  
  
}

##### UserModel类

package com.javacode2018.mybatis.chat01;  
  
import lombok.\*;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
@Getter  
@Setter  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
@ToString  
public class UserModel {  
 private Long id;  
 private String name;  
 private Integer age;  
 private Double salary;  
}

这个类上面的注解都是都是lombok中的，通过这些注解，lombok可以帮助我们自动生成上面4个字段的get方法、set方法、无参构造方法、有参有参构造方法、builder模式构建对象的代码、重写toString方法，这些都在代码编译为字节码之前会写进去，通过lombok代码是不是精简了很多，最后生成的代码大家可以反编译一下UserModel.class去看一下，感受一下，此处我们就不贴出来了。

##### UserUtil类

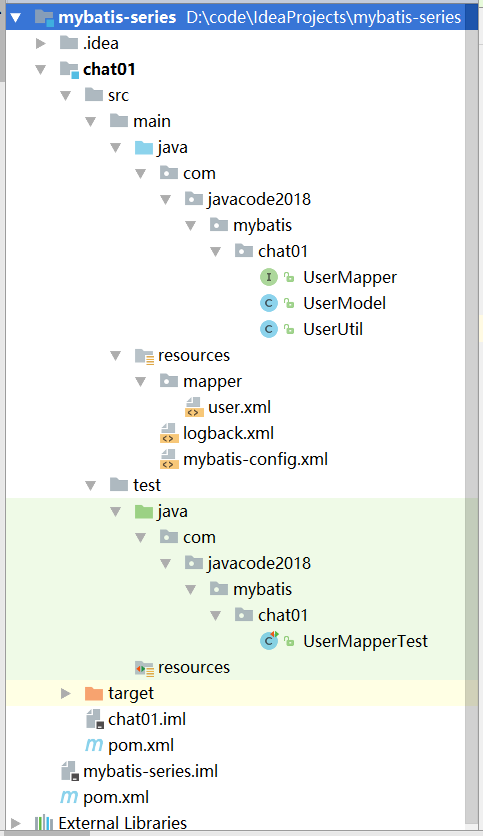
package com.javacode2018.mybatis.chat01;  
  
import lombok.extern.slf4j.Slf4j;  
import org.apache.ibatis.io.Resources;  
import org.apache.ibatis.session.SqlSession;  
import org.apache.ibatis.session.SqlSessionFactory;  
import org.apache.ibatis.session.SqlSessionFactoryBuilder;  
  
import java.io.IOException;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
@Slf4j  
public class UserUtil {  
 private static SqlSessionFactory sqlSessionFactory = build();  
  
 public static SqlSessionFactory build() {  
 try {  
 return new SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("mybatis-config.xml"));  
 } catch (IOException e) {  
 log.error(e.getMessage(), e);  
 throw new RuntimeException(e);  
 }  
 }  
  
 @FunctionalInterface  
 public interface SessionCall<O> {  
 O call(SqlSession session) throws Exception;  
 }  
  
 @FunctionalInterface  
 public interface MapperCall<T, O> {  
 O call(T mapper) throws Exception;  
 }  
  
 public static <T, O> O callMapper(Class<T> tClass, MapperCall<T, O> mapper) throws Exception {  
 return call(session -> mapper.call(session.getMapper(tClass)));  
 }  
  
 public static <O> O call(SessionCall<O> sessionCall) throws Exception {  
 try (SqlSession session = sqlSessionFactory.openSession(true);) {  
 return sessionCall.call(session);  
 }  
 }  
}

#### 创建单元测试类UserMapperTest

chat01\src\test\java\com\javacode2018\mybatis\chat01中创建UserMapperTest，代码如下：

package com.javacode2018.mybatis.chat01;  
  
import lombok.extern.slf4j.Slf4j;  
import org.junit.Test;  
  
import java.util.\*;  
import java.util.stream.Collectors;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
@Slf4j  
public class UserMapperTest {  
  
 //动态插入  
 @Test  
 public void insert() throws Exception {  
 UserModel userModel1 = UserModel.builder().name("马士兵老师").build();  
 UserUtil.callMapper(UserMapper.class, mapper -> {  
 mapper.insert(userModel1);  
 return null;  
 });  
 log.info("插入结果：{}", this.getModelById(userModel1.getId()));  
 log.info("---------------------");  
 UserModel userModel2 = UserModel.builder().name("路人").age(30).salary(50000.00).build();  
 UserUtil.callMapper(UserMapper.class, mapper -> {  
 mapper.insert(userModel2);  
 return null;  
 });  
 log.info("插入结果：{}", this.getModelById(userModel2.getId()));  
 }  
  
 //批量插入  
 @Test  
 public void insertBatch() throws Exception {  
 List<UserModel> userModelList = new ArrayList<>();  
 for (int i = 1; i <= 5; i++) {  
 userModelList.add(UserModel.builder().name("马士兵老师-" + i).age(30 + i).salary(10000.00 \* i).build());  
 userModelList.add(UserModel.builder().name("javacode2018-" + i).age(30 + i).salary(10000.00 \* i).build());  
 }  
 UserUtil.callMapper(UserMapper.class, mapper -> {  
 mapper.insertBatch(userModelList);  
 return null;  
 });  
  
 List<UserModel> userModelList1 = UserUtil.callMapper(UserMapper.class, mapper -> mapper.getModelList(null));  
 log.info("结果:{}", userModelList1);  
 }  
  
 //根据用户id删除数据  
 @Test  
 public void delete() throws Exception {  
 Map<String, Object> map = new HashMap<>();  
 //需要删除的用户id  
 map.put("id", 1);  
 Integer count = UserUtil.callMapper(UserMapper.class, mapper -> mapper.delete(map));  
 log.info("删除行数：{}", count);  
 }  
  
 //动态更新  
 @Test  
 public void update() throws Exception {  
 //将userId=2的name修改为：路人  
 Long userId1 = 2L;  
 Integer count = UserUtil.callMapper(UserMapper.class, mapper -> mapper.update(UserModel.builder().id(userId1).name("ready").build()));  
 log.info("更新行数：{}", count);  
  
 log.info("---------------------");  
 //将userId=3的name修改为：路人,薪水为：1000.88  
 Long userId2 = 3L;  
 count = UserUtil.callMapper(UserMapper.class, mapper -> mapper.update(UserModel.builder().id(userId2).name("ready").salary(1000.88D).build()));  
 log.info("更新行数：{}", count);  
 }  
  
 //按用户id查询  
 public UserModel getModelById(Long userId) throws Exception {  
 //查询指定id的数据  
 Map<String, Object> map = new HashMap<>();  
 map.put("id", userId);  
 return UserUtil.callMapper(UserMapper.class, mapper -> {  
 List<UserModel> userModelList = mapper.getModelList(map);  
 if (userModelList.size() == 1) {  
 return userModelList.get(0);  
 }  
 return null;  
 });  
 }  
  
 //查询所有数据  
 @Test  
 public void getModelList1() throws Exception {  
 List<UserModel> userModelList = UserUtil.callMapper(UserMapper.class, mapper -> mapper.getModelList(null));  
 log.info("结果:{}", userModelList);  
 }  
  
 //查询多个用户id对应的数据  
 @Test  
 public void getModelListByIds() throws Exception {  
 List<Integer> idList = Arrays.asList(2, 3, 4).stream().collect(Collectors.toList());  
 Map<String, Object> map = new HashMap<>();  
 map.put("idList", idList);  
  
 List<UserModel> userModelList = UserUtil.callMapper(UserMapper.class, mapper -> mapper.getModelList(map));  
 log.info("结果:{}", userModelList);  
 }  
  
 //多条件 & 指定返回的列  
 @Test  
 public void getModelList2() throws Exception {  
 //查询姓名中包含马士兵老师以及薪资大于3万的用户id、姓名  
 Map<String, Object> map = new HashMap<>();  
 map.put("nameLike", "马士兵老师");  
 map.put("salaryGte", 30000.00D);  
 //需要返回的列  
 List<String> tableColumnList = new ArrayList<>();  
 tableColumnList.add("id");  
 tableColumnList.add("name");  
 map.put("tableColumnList", tableColumnList);  
  
 List<UserModel> userModelList = UserUtil.callMapper(UserMapper.class, mapper -> mapper.getModelList(map));  
 log.info("结果:{}", userModelList);  
 }  
  
 //条件过滤 & 排序 & 分页查询数据 & 只返回用户id、salary  
 @Test  
 public void getPage() throws Exception {  
 //查询姓名中包含马士兵老师以及薪资大于3万的用户id，按照薪资倒叙，每页5条取第1页  
 Map<String, Object> map = new HashMap<>();  
 map.put("nameLike", "马士兵老师");  
 map.put("salaryGte", 30000.00D);  
  
 //加入排序参数  
 map.put("sort", "salary desc");  
  
 //加入分页参数  
 int page = 1;  
 int pageSize = 5;  
 map.put("skip", (page - 1) \* pageSize);  
 map.put("pageSize", pageSize);  
  
 //加入需要返回的列  
 List<String> tableColumnList = new ArrayList<>();  
 tableColumnList.add("id");  
 tableColumnList.add("salary");  
 map.put("tableColumnList", tableColumnList);  
  
 List<UserModel> userModelList = UserUtil.callMapper(UserMapper.class, mapper -> mapper.getModelList(map));  
 log.info("结果:{}", userModelList);  
 }  
}

#### 项目最终结构如下



#### 用例：动态插入

运行UserMapperTest#insert，输出如下：

37:58.556 [main] DEBUG c.j.mybatis.chat01.UserMapper.insert - ==> Preparing: INSERT INTO `t\_user` ( `name` ) VALUES ( ? )   
37:58.605 [main] DEBUG c.j.mybatis.chat01.UserMapper.insert - ==> Parameters: 马士兵老师(String)  
37:58.613 [main] DEBUG c.j.mybatis.chat01.UserMapper.insert - <== Updates: 1  
37:58.641 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - ==> Preparing: SELECT `id`, `name`, `age`, `salary` FROM `t\_user` a WHERE a.`id` = ?   
37:58.641 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - ==> Parameters: 1(Long)  
37:58.663 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - <== Total: 1  
37:58.664 [main] INFO c.j.mybatis.chat01.UserMapperTest - 插入结果：UserModel(id=1, name=马士兵老师, age=1, salary=0.0)  
37:58.667 [main] INFO c.j.mybatis.chat01.UserMapperTest - ---------------------  
37:58.668 [main] DEBUG c.j.mybatis.chat01.UserMapper.insert - ==> Preparing: INSERT INTO `t\_user` ( `name`, `age`, `salary` ) VALUES ( ?, ?, ? )   
37:58.675 [main] DEBUG c.j.mybatis.chat01.UserMapper.insert - ==> Parameters: 路人(String), 30(Integer), 50000.0(Double)  
37:58.679 [main] DEBUG c.j.mybatis.chat01.UserMapper.insert - <== Updates: 1  
37:58.681 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - ==> Preparing: SELECT `id`, `name`, `age`, `salary` FROM `t\_user` a WHERE a.`id` = ?   
37:58.681 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - ==> Parameters: 2(Long)  
37:58.683 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - <== Total: 1  
37:58.683 [main] INFO c.j.mybatis.chat01.UserMapperTest - 插入结果：UserModel(id=2, name=路人, age=30, salary=50000.0)

UserMapperTest#insert这个方法主要有4步操作：

步骤1：插入一条用户记录，用户记录只有name字段有值

步骤2：去db中查询步骤1中插入的记录

步骤3：插入一条用户记录，这次插入的记录所有字段都指定了值

步骤4：去db中查询步骤3中插入的记录

**重点来了：**大家认真看一下UserMapperTest#insert方法的代码，两个插入调用都是mapper.insert方法，传入的都是UserModel对象，唯一不同的是这个对象构建的时候字段的值不一样，最后再认真看一下上面输出的sql，产生的2个insert也是不一样的，这个mapper.insert方法可以根据UserModel对象字段是否有值来组装我们需要的sql，是不是很牛逼，这就是动态插入。

#### 用例：批量插入

运行UserMapperTest#insertBatch，输出如下：

38:12.425 [main] DEBUG c.j.m.chat01.UserMapper.insertBatch - ==> Preparing: INSERT INTO `t\_user` (`id`, `name`, `age`, `salary`) VALUES (?, ?, ?, ?) , (?, ?, ?, ?) , (?, ?, ?, ?) , (?, ?, ?, ?) , (?, ?, ?, ?) , (?, ?, ?, ?) , (?, ?, ?, ?) , (?, ?, ?, ?) , (?, ?, ?, ?) , (?, ?, ?, ?)   
38:12.476 [main] DEBUG c.j.m.chat01.UserMapper.insertBatch - ==> Parameters: null, 马士兵老师-1(String), 31(Integer), 10000.0(Double), null, javacode2018-1(String), 31(Integer), 10000.0(Double), null, 马士兵老师-2(String), 32(Integer), 20000.0(Double), null, javacode2018-2(String), 32(Integer), 20000.0(Double), null, 马士兵老师-3(String), 33(Integer), 30000.0(Double), null, javacode2018-3(String), 33(Integer), 30000.0(Double), null, 马士兵老师-4(String), 34(Integer), 40000.0(Double), null, javacode2018-4(String), 34(Integer), 40000.0(Double), null, 马士兵老师-5(String), 35(Integer), 50000.0(Double), null, javacode2018-5(String), 35(Integer), 50000.0(Double)  
38:12.484 [main] DEBUG c.j.m.chat01.UserMapper.insertBatch - <== Updates: 10  
38:12.502 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - ==> Preparing: SELECT `id`, `name`, `age`, `salary` FROM `t\_user` a   
38:12.502 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - ==> Parameters:   
38:12.521 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - <== Total: 12  
38:12.521 [main] INFO c.j.mybatis.chat01.UserMapperTest - 结果:[UserModel(id=1, name=马士兵老师, age=1, salary=0.0), UserModel(id=2, name=路人, age=30, salary=50000.0), UserModel(id=3, name=马士兵老师-1, age=31, salary=10000.0), UserModel(id=4, name=javacode2018-1, age=31, salary=10000.0), UserModel(id=5, name=马士兵老师-2, age=32, salary=20000.0), UserModel(id=6, name=javacode2018-2, age=32, salary=20000.0), UserModel(id=7, name=马士兵老师-3, age=33, salary=30000.0), UserModel(id=8, name=javacode2018-3, age=33, salary=30000.0), UserModel(id=9, name=马士兵老师-4, age=34, salary=40000.0), UserModel(id=10, name=javacode2018-4, age=34, salary=40000.0), UserModel(id=11, name=马士兵老师-5, age=35, salary=50000.0), UserModel(id=12, name=javacode2018-5, age=35, salary=50000.0)]

这次批量插入了10条用户记录，可以看到有这样的输出：

40:40.727 [main] DEBUG c.j.m.chat01.UserMapper.insertBatch - <== Updates: 10

上面这个表示插入影响的行数，10表示插入了10行。

批量插入之后，又执行了全表查询，这次插入了10条，加上前面的2个单条插入，表中总计12条记录。

#### 用例：根据用户id删除数据

运行UserMapperTest#delete，输出如下：

38:36.498 [main] DEBUG c.j.mybatis.chat01.UserMapper.delete - ==> Preparing: DELETE FROM `t\_user` WHERE `id` = ?   
38:36.551 [main] DEBUG c.j.mybatis.chat01.UserMapper.delete - ==> Parameters: 1(Integer)  
38:36.560 [main] DEBUG c.j.mybatis.chat01.UserMapper.delete - <== Updates: 1  
38:36.561 [main] INFO c.j.mybatis.chat01.UserMapperTest - 删除行数：1

#### 用例：动态更新

运行UserMapperTest#update，输出如下：

38:51.289 [main] DEBUG c.j.mybatis.chat01.UserMapper.update - ==> Preparing: UPDATE `t\_user` SET `name` = ? WHERE `id` = ?   
38:51.347 [main] DEBUG c.j.mybatis.chat01.UserMapper.update - ==> Parameters: ready(String), 2(Long)  
38:51.355 [main] DEBUG c.j.mybatis.chat01.UserMapper.update - <== Updates: 1  
38:51.356 [main] INFO c.j.mybatis.chat01.UserMapperTest - 更新行数：1  
38:51.358 [main] INFO c.j.mybatis.chat01.UserMapperTest - ---------------------  
38:51.359 [main] DEBUG c.j.mybatis.chat01.UserMapper.update - ==> Preparing: UPDATE `t\_user` SET `name` = ?, `salary` = ? WHERE `id` = ?   
38:51.360 [main] DEBUG c.j.mybatis.chat01.UserMapper.update - ==> Parameters: ready(String), 1000.88(Double), 3(Long)  
38:51.363 [main] DEBUG c.j.mybatis.chat01.UserMapper.update - <== Updates: 1  
38:51.364 [main] INFO c.j.mybatis.chat01.UserMapperTest - 更新行数：1

UserMapperTest#update方法，大家也认真看一下，2个更新，调用都是mapper.update方法，传入的都是UserModel类型的参数，只是2个UserModel对象的字段值不一样，最后产生的2个update语句也是不一样的，这个update语句是mybatis动态组装的，mybatis可以根据UserModel中字段是否为NULL，来拼装sql，这个更新是不是很强大。

#### 用例：动态查询

##### 查询所有数据

运行UserMapperTest#getModelList1，输出如下：

39:10.552 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - ==> Preparing: SELECT `id`, `name`, `age`, `salary` FROM `t\_user` a   
39:10.611 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - ==> Parameters:   
39:10.639 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - <== Total: 11  
39:10.639 [main] INFO c.j.mybatis.chat01.UserMapperTest - 结果:[UserModel(id=2, name=ready, age=30, salary=50000.0), UserModel(id=3, name=ready, age=31, salary=1000.88), UserModel(id=4, name=javacode2018-1, age=31, salary=10000.0), UserModel(id=5, name=马士兵老师-2, age=32, salary=20000.0), UserModel(id=6, name=javacode2018-2, age=32, salary=20000.0), UserModel(id=7, name=马士兵老师-3, age=33, salary=30000.0), UserModel(id=8, name=javacode2018-3, age=33, salary=30000.0), UserModel(id=9, name=马士兵老师-4, age=34, salary=40000.0), UserModel(id=10, name=javacode2018-4, age=34, salary=40000.0), UserModel(id=11, name=马士兵老师-5, age=35, salary=50000.0), UserModel(id=12, name=javacode2018-5, age=35, salary=50000.0)]

可以看到sql是没有查询条件的。

##### 查询多个用户id对应的数据

运行UserMapperTest#getModelListByIds，输出如下：

39:38.000 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - ==> Preparing: SELECT `id`, `name`, `age`, `salary` FROM `t\_user` a WHERE a.`id` IN ( ? , ? , ? )   
39:38.064 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - ==> Parameters: 2(Integer), 3(Integer), 4(Integer)  
39:38.096 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - <== Total: 3  
39:38.097 [main] INFO c.j.mybatis.chat01.UserMapperTest - 结果:[UserModel(id=2, name=ready, age=30, salary=50000.0), UserModel(id=3, name=ready, age=31, salary=1000.88), UserModel(id=4, name=javacode2018-1, age=31, salary=10000.0)]

上面这个按照id列表查询也是比较常用的，比如我们在电商中查询订单列表，还需要查询每个订单对应的商品，此时可以先查询订单列表，然后在通过订单列表拿到所有的商品id集合，然后通过商品id集合去通过上面的方式检索商品信息，只需要2次查询就可以查询出订单及商品的信息了。

##### 多条件 & 指定返回的列

运行UserMapperTest#getModelList2，查询姓名中包含马士兵老师以及薪资大于3万的用户id、姓名，输出如下：

41:12.185 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - ==> Preparing: SELECT id , name FROM `t\_user` a WHERE a.`name` like '%马士兵老师%' AND a.`salary` >= ?   
41:12.275 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - ==> Parameters: 30000.0(Double)  
41:12.311 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - <== Total: 3  
41:12.312 [main] INFO c.j.mybatis.chat01.UserMapperTest - 结果:[UserModel(id=7, name=马士兵老师-3, age=null, salary=null), UserModel(id=9, name=马士兵老师-4, age=null, salary=null), UserModel(id=11, name=马士兵老师-5, age=null, salary=null)]

看一下上面select语句，select后面只有id,name2个字段，where后面有多个条件，这种查询也是比较常用的，有些表可能有几十个字段，可能我们只需要几个字段，就可以使用上面这种查询。

##### 条件过滤 & 排序 & 分页查询数据 & 只返回用户id、salary

运行UserMapperTest#getModelList3，查询姓名中包含马士兵老师以及薪资大于3万的用户id，按照薪资倒叙，每页5条取第1页，输出如下：

44:00.719 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - ==> Preparing: SELECT id , salary FROM `t\_user` a WHERE a.`name` like '%马士兵老师%' AND a.`salary` >= ? order by salary desc LIMIT ?,?   
44:00.775 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - ==> Parameters: 30000.0(Double), 0(Integer), 5(Integer)  
44:00.805 [main] DEBUG c.j.m.chat01.UserMapper.getModelList - <== Total: 3  
44:00.806 [main] INFO c.j.mybatis.chat01.UserMapperTest - 结果:[UserModel(id=11, name=null, age=null, salary=50000.0), UserModel(id=9, name=null, age=null, salary=40000.0), UserModel(id=7, name=null, age=null, salary=30000.0)]

大家主要看一下输出的sql，如下：

SELECT id , salary FROM `t\_user` a WHERE a.`name` like '%马士兵老师%' AND a.`salary` >= ? order by salary desc LIMIT ?,?

这个sql会根据查询条件，自动构建出我们需要的sql，这点上面是最厉害的。

#### 案例总结

上面列举的一些用例基本上包含了我们对db所需的大部分操作，动态sql处理方面体现的最为强劲，如果让我们自己写，我们需要写很多判断，而用mybatis这么简单就实现了，我们在java代码中没有看到一个判断拼接语句，而这些sql的判断拼接都在一个文件中：user.xml中，这个就是mybatis中核心的文件，我们需要写的sql及判断逻辑基本上都在这个xml中，大家可以认真去看一下这个xml文件。

### mybatis开发项目的具体步骤

#### 项目中引入mybatis maven配置

<dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis</artifactId>  
 <version>${mybatis.version}</version>  
</dependency>

上面的mybatis.version版本，大家可以在maven社区中央仓库中去查找最新的，目前最新的是3.5.3

#### 创建mybatis配置文件

mybatis配置文件为xml格式，可以放在resource目录下面，如上面案例中的mybatis-config.xml，内容如下：

<?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>  
 <properties>  
 <property name="driver" value="com.mysql.jdbc.Driver"/>  
 <property name="url" value="jdbc:mysql://localhost:3306/javacode2018?characterEncoding=UTF-8"/>  
 <property name="username" value="root"/>  
 <property name="password" value="root123"/>  
 </properties>  
 <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="mapper/user.xml"/>  
 </mappers>  
</configuration>

这个文件主要是对mybatis进行全局配置，比如数据源、事务的配置，如上面的datasource元素用来配置数据源，数据源中就需要指定数据库的一些配置信息；还有其他更多的配置，此处先不做具体说明，后面我们慢慢来，整个系列完成之后，这些配置大家都会懂的。

#### 创建mapper xml文件

如上面案例中的user.xml，大家在打开看看，我们需要对t\_user表所有操作sql就写在这个文件中，下一篇文章我们会详细介绍mapper xml文件的各种写法，user.xml文件是对t\_user表的所有操作一般都会放在这个里面，mybatis如何使用到这个文件呢，我们需要在上面的mybatis配置文件中引入这个mapper文件，如案例中在mybatis-config.xml有下面这样的内容：

<mappers>  
 <mapper resource="mapper/user.xml"/>  
</mappers>

mappers元素中可以有多个mapper文件，我们开发的项目中可能有很多表需要操作，那么对应会有很多mapper xml文件，我们都需要在mappers元素中进行引入，然后mybatis才会使用到。

#### 创建Mapper接口

开发者如何去调用user.xml中的各种操作去执行sql呢，这时我们就需要一个Mapper接口了，Mapper接口会和mapper xml建立映射关系，当我们调用Mapper接口中的方法的时候，会间接的调用到mapper xml中的各种数据的sql操作，Mapper接口如何和Mapper xml文件关联的呢？

大家去看一下user.xml文件中有个这样的一个配置：

<mapper namespace="com.javacode2018.mybatis.chat01.UserMapper">

注意上面的namespace的值，对应的是UserMapper这个接口完整的引用，通过这个namespace，UserMapper接口就可以user.xml建立了映射关系。

user.xml中又有很多db操作，这些操作会和UserMapper接口中的方法建立映射关系，当调用UserMapper中的方法的时候，间接的会调用到user.xml中对应的操作。

如user.xml中有下面一段配置：

<!-- 批量插入 -->  
<insert id="insertBatch" parameterType="map">  
 <![CDATA[ INSERT INTO `t\_user` (`id`, `name`, `age`, `salary`) VALUES ]]>  
 <foreach collection="list" separator="," item="item">  
 (#{item.id}, #{item.name}, #{item.age}, #{item.salary})  
 </foreach>  
</insert>

而UserMapper中有个insertBatch方法和上面这个insert批量插入对应，如下：

/\*\*  
 \* 批量插入用户信息  
 \*  
 \* @param userModelList  
 \*/  
void insertBatch(List<UserModel> userModelList);

所以当我们调用UserMapper中的insertBatch方法的时候，会间接调用到user.xml中的 id="insertBatch"这个操作。

提示一下：接口和mapper xml映射起来间接调用，是通过java动态代理实现的，后面我们会详解如何实现的。

下面我们就可以使用mybatis来操作db了。

#### 通过mybatis获取Mapper接口执行对db的操作

上面我们说了，我们可以通过mapper接口来执行对db的操作，获取Mapper的主要代码如下：

SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();  
SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(Resources.getResourceAsStream("mybatis-config.xml"));  
SqlSession sqlSession = sqlSessionFactory.openSession(true);  
UserMapper mapper = sqlSession.getMapper(UserMapper.class);

上面代码中使用到了mybatis中的核心组件，我们具体来看一下。

#### Mybatis核心对象介绍

##### SqlSessionFactoryBuilder

这个是一个构建器，通过名字大家也可以感觉到SqlSessionFactoryBuilder构建器，是用来构建SqlSessionFactory对象的，SqlSessionFactoryBuilder可以通过读取mybatis的配置文件，然后构建一个SqlSessionFactory对象，一个项目中有很多mapper xml文件，如果每次操作都去重新解析是非常慢的，那么怎么办？

能不能第一次解析好然后放在内存中，以后直接使用，SqlSessionFactoryBuilder就是搞这个事情的，将mybatis配置文件、mapper xml文件、mapper xml文件和Mapper 接口的映射关系，这些都先给解析好，然后放在java对象中，java对象存在于内存中，内存中访问会非常快的，那么我们每次去用的时候就不需要重新去解析xml了，SqlSessionFactoryBuilder解析配置之后，生成的对象就是SqlSessionFactory，这个是一个重量级的对象，创建他是比较耗时的，所以一般一个db我们会创建一个SqlSessionFactory对象，然后在系统运行过程中会一直存在，而SqlSessionFactoryBuilder用完了就可以释放了。

##### SqlSessionFactory

通过名字可以知道，这个是一个工厂，是用来创建SqlSession的工厂，SqlSessionFactory是一个重量级的对象，一般一个db对应一个SqlSessionFactory对象，系统运行过程中会一直存在。

SqlSessionFactory是一个接口，这个接口有2个实现DefaultSqlSessionFactory和SqlSessionManager，一般都是通过SqlSessionFactoryBuilder来创建SqlSessionFactory对象。

通过SqlSessionFactoryBuilder来创建SqlSessionFactory对象主要有2种方式，一种通过读取mybatis配置文件的方式，另外一种是硬编码的方式，这个后面会专门抽一篇文件介绍这块，springboot中会使用到硬编码的方式，所以这块会详细介绍。

##### SqlSession

我们通过jdbc操作数据库需要先获取一个Connection连接，然后拿着这个连接去对db进行操作，在mybatis中SqlSession就类似于jdbc中Connection连接对象，在mybatis中叫做Sql会话对象，一般我们一个db操作使用一个SqlSession对象，所以这个对象一般是方法级别的，方法结束之后，这个对象就销毁了，这个对象可以调用sqlSessionFactory.openSession的方法来进行获取。

我们可以直接通过SqlSession对象来调用mapper xml中各种db操作，需要指定具体的操作的id，id的格式为namespace.操作的id。

##### Mapper接口

我们可以通过SqlSession直接调用mapper xml中的db操作，不过更简单的以及推荐的方式是使用Mapper接口，Mapper接口中的方法和mapper xml文件中的各种db操作建立了映射关系，是通过Mapper接口完整名称+方法名称和mapper xml中的namespace+具体操作的id来进行关联的，然后我们直接调用Mapper接口中的方法就可以间接的操作db了，使用想当方便，Mapper接口需要通过SqlSession获取，传入Mapper接口对应的Class对象，然后会返回这个接口的实例，如：

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

### 总结

本篇文章主要通过一个案例来感受一下mybatis可以干什么，以及他的强大之处，还需要大家掌握mybatis开发项目的具体步骤，后面的文章将对mybatis中具体的知识点做详细介绍，让大家成为mybatis高手。

### MyBatis系列

1. MyBatis系列第1篇：MyBatis未出世之前我们那些痛苦的经历

## Mybatis系列第3篇：Mybatis使用详解(-)

### 主要内容

1. **快速入门**
   * **准备数据库**
   * **我们的需求**
   * **使用idea创建项目**
   * **pom.xml中引入mybatis依赖**
   * **配置mybatis全局配置文件**
   * **创建Mapper xml文件**
   * **mybatis全局配置文件中引入Mapper xml文件**
   * **构建SqlSessionFactory对象**
   * **构建SqlSession对象**
   * **引入lombok（非必须）**
   * **引入logback支持（非必须）**
   * **写一个测试用例**
2. **使用SqlSesion执行sql操作**
   * **SqlSession常见的用法**
   * **新增操作**
   * **执行删除**
   * **执行修改**
   * **执行查询**
3. **Mapper接口的使用**
   * **为什么需要Mapper接口**
   * **Mapper接口的用法**
   * **案例：使用Mapper接口来实现增删改查**
   * **Mapper接口使用时注意的几点**
   * **Mapper接口的原理**

### 快速入门

#### 准备数据库

mysql中执行下面sql：

/\*创建数据库javacode2018\*/  
DROP DATABASE IF EXISTS `javacode2018`;  
CREATE DATABASE `javacode2018`;  
USE `javacode2018`;  
  
/\*创建表结构\*/  
DROP TABLE IF EXISTS `t\_user`;  
CREATE TABLE t\_user (  
 id BIGINT AUTO\_INCREMENT PRIMARY KEY COMMENT '主键，用户id，自动增长',  
 `name` VARCHAR(32) NOT NULL DEFAULT '' COMMENT '姓名',  
 `age` SMALLINT NOT NULL DEFAULT 1 COMMENT '年龄',  
 `salary` DECIMAL(12,2) NOT NULL DEFAULT 0 COMMENT '薪水',  
 `sex` TINYINT NOT NULL DEFAULT 0 COMMENT '性别,0:未知,1:男,2:女'  
) COMMENT '用户表';  
  
SELECT \* FROM t\_user;

上面我们创建了一个数据库：javacode2018，一个用户表t\_user。

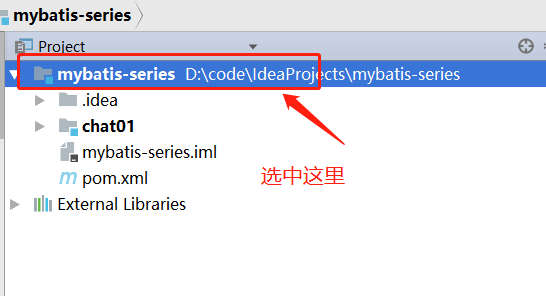
#### 我们的需求

使用mybatis来实现对t\_user表增删改查。

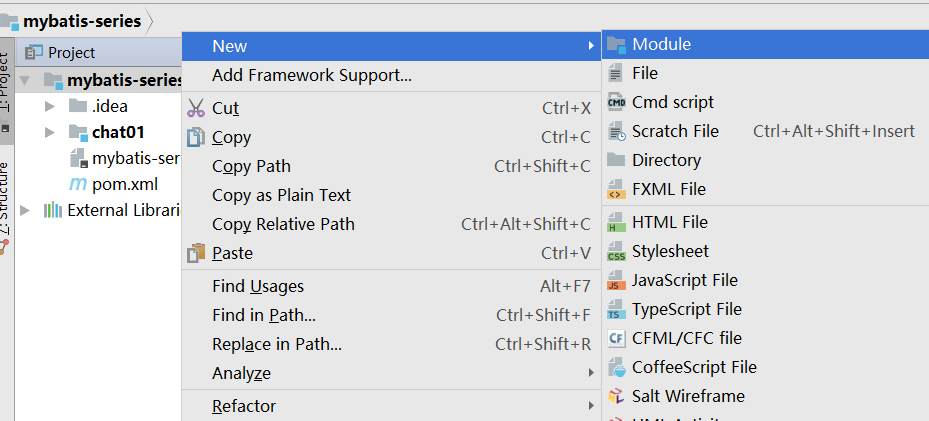
#### 使用idea创建项目

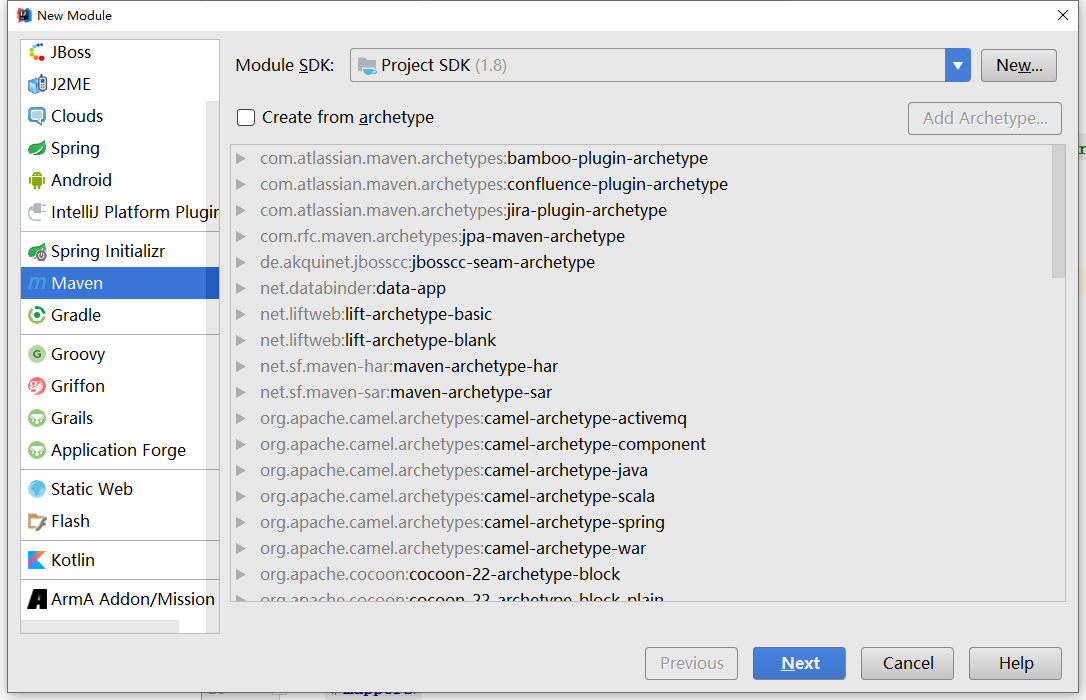
我们在上一篇文章mybatis-series项目中创建另外一个模块chat02，过程如下：

选中mybatis-series，如下图：

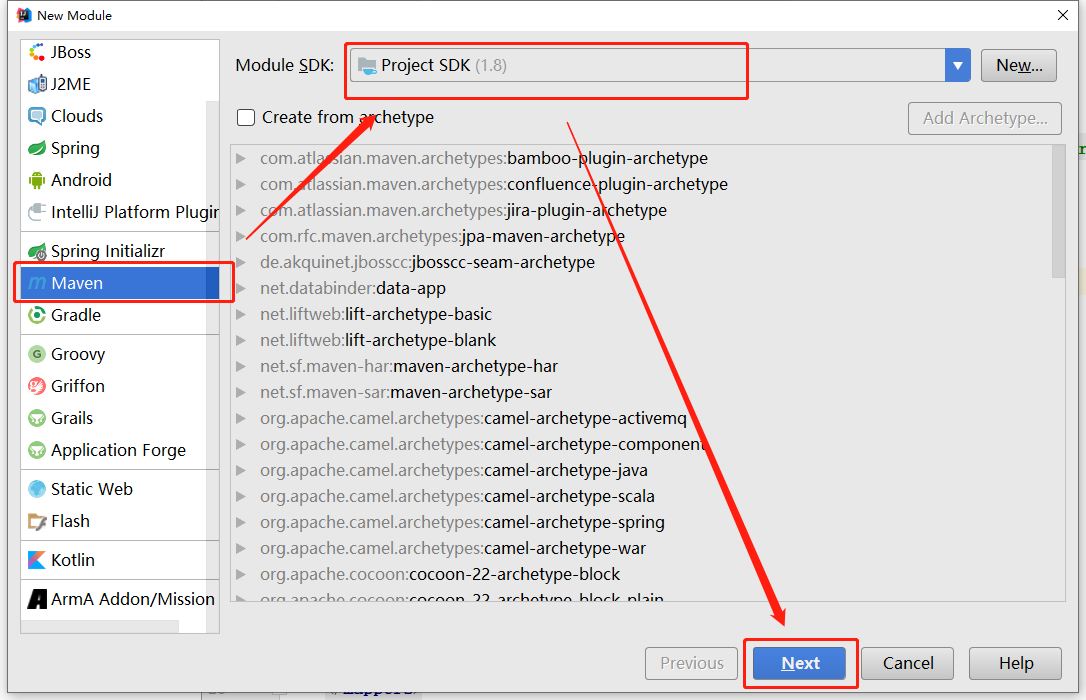


点击右键->New->Module，如下图：

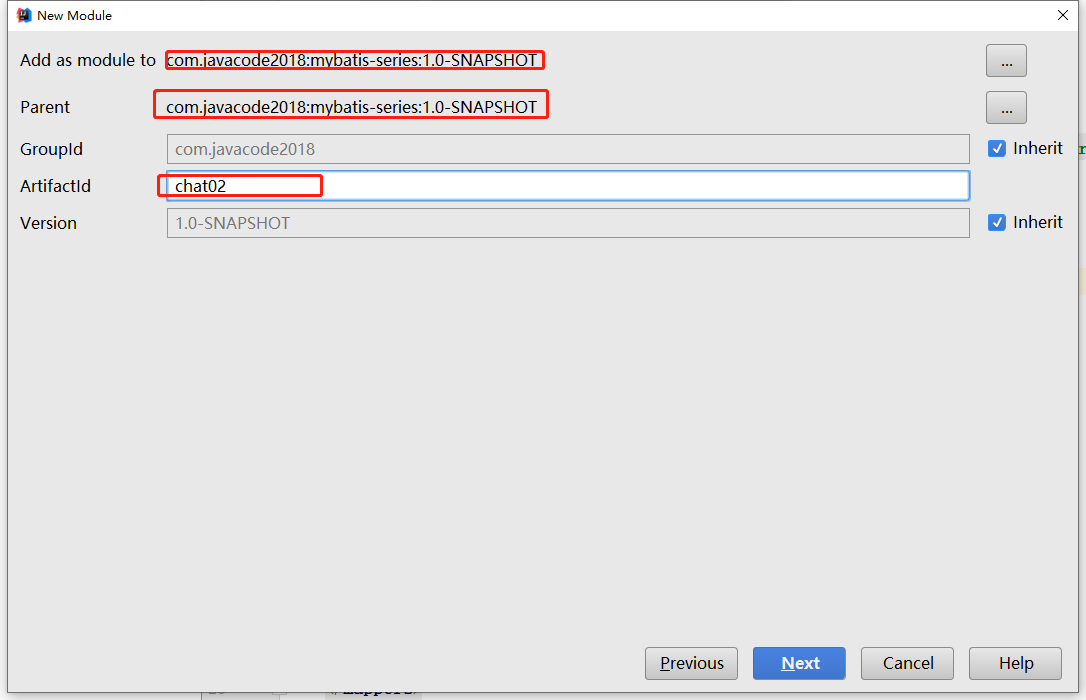




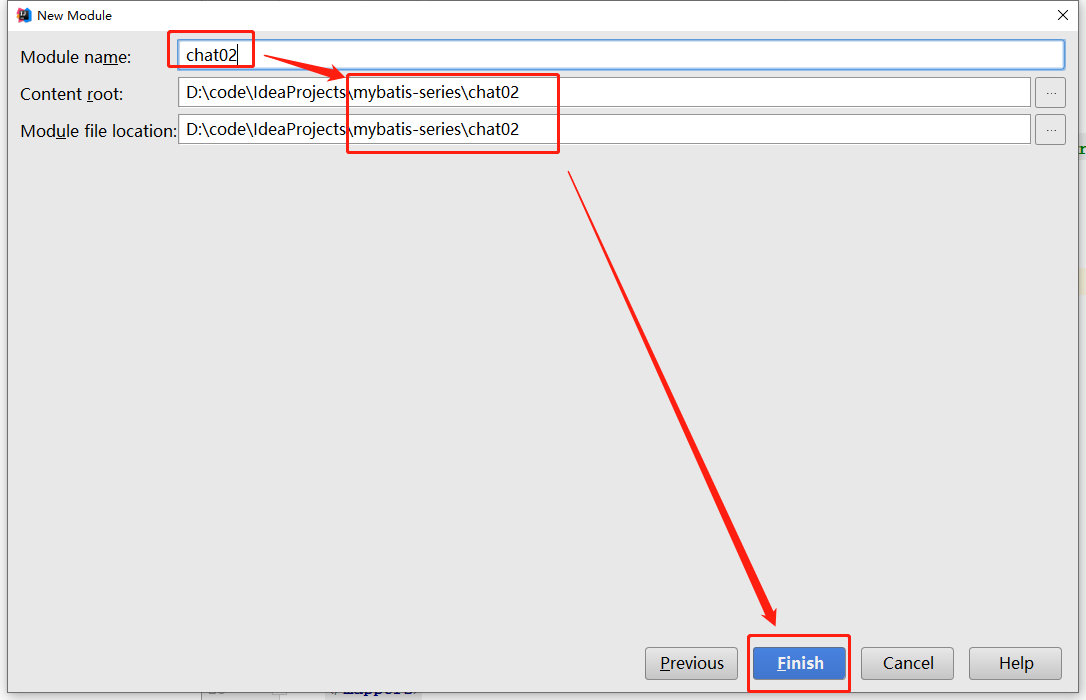
选中上图中的Maven，点击Next，如下图：



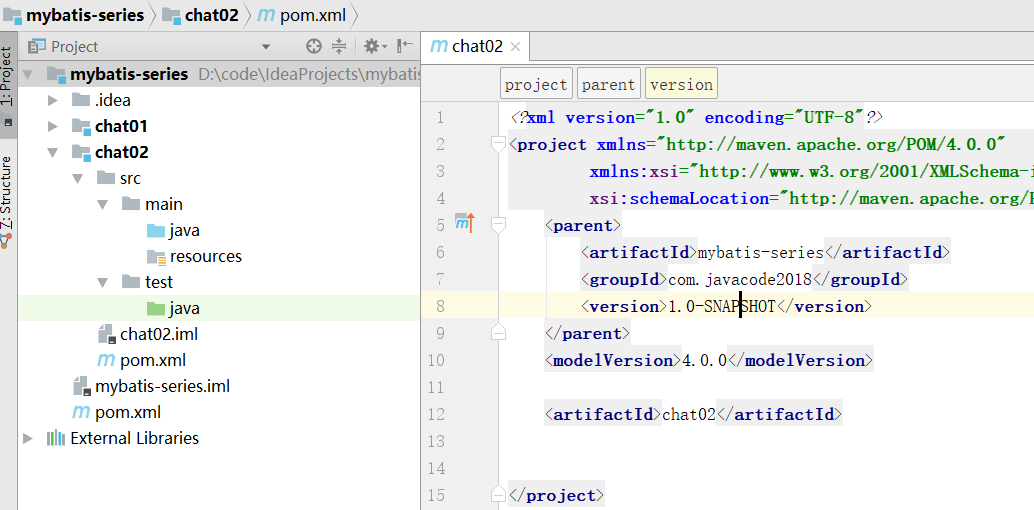
出现下面窗口：



上图中输入ArtifactId为chat02，点击Next，如下图：



点击上图中的Finish完成chat02模块的创建，项目结构如下图：



#### pom.xml中引入mybatis依赖

<dependencies>  
 <!-- mybatis依赖 -->  
 <dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis</artifactId>  
 </dependency>  
 <!-- mysql 驱动 -->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 </dependency>  
 <!-- lombok支持 -->  
 <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 </dependency>  
 <!-- 单元测试junit支持 -->  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 </dependency>  
 <!-- 引入logback用来输出日志 -->  
 <dependency>  
 <groupId>ch.qos.logback</groupId>  
 <artifactId>logback-classic</artifactId>  
 </dependency>  
</dependencies>

上面我们引入了依赖mybatis、mysql驱动、lombok支持、junit、logback支持，其实运行mybatis只需要引入下面这一个构件就行了：

<dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis</artifactId>  
</dependency>

注意：上面pom引入的构建中没有写版本号，是因为构件的版本号在父pom.xml中已经声明了，所以chat03/pom.xml中就不需要再去写了。

#### 配置mybatis全局配置文件

使用mybatis操作数据库，那么当然需要配置数据库相关信息，这个需要在mybatis全局配置文件中进行配置。

mybatis需提供一个全局配置的xml文件，可以在这个配置文件中对mybatis进行配置，如事务的支持，数据源的配置等等，这个属于配置文件，我们一般放在main/resource中。

在chat03/src/main/resource中创建mybatis-config.xml文件，内容如下：

<?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>  
 <!-- 环境配置，可以配置多个环境 -->  
 <environments default="chat03">  
 <!--   
 environment用来对某个环境进行配置  
 id：环境标识，唯一  
 -->  
 <environment id="chat03">  
 <!-- 事务管理器工厂配置 -->  
 <transactionManager type="org.apache.ibatis.transaction.jdbc.JdbcTransactionFactory"/>  
 <!-- 数据源工厂配置，使用工厂来创建数据源 -->  
 <dataSource type="org.apache.ibatis.datasource.pooled.PooledDataSourceFactory">  
 <property name="driver" value="com.mysql.jdbc.Driver"/>  
 <property name="url" value="jdbc:mysql://localhost:3306/javacode2018?characterEncoding=UTF-8"/>  
 <property name="username" value="root"/>  
 <property name="password" value="root123"/>  
 </dataSource>  
 </environment>  
 </environments>  
</configuration>

我们做一下解释。

##### configuration元素

这个是mybatis全局配置文件的根元素，每个配置文件只有一个

##### environments元素

用来配置mybatis的环境信息，什么是环境？比如开发环境、测试环境、线上环境，这3个环境中的数据库可能是不一样的，可能还有更多的环境。

environments元素中用来配置多个环境的，具体的一个环境使用environment元素进行配置，environment元素有个id用来标识某个具体的环境。

配置了这么多环境，那么mybatis具体会使用哪个呢？

environments元素有个default属性，用来指定默认使用哪个环境，如上面默认使用的是chat03。

##### environment元素

用来配置具体的环境信息，这个元素下面有两个子元素：**transactionManager和dataSource**

* transactionManager元素
* 用来配置事务工厂的，有个type属性，type的值必须是org.apache.ibatis.transaction.TransactionFactory接口的实现类，TransactionFactory看名字就知道是一个工厂，用来创建事务管理器org.apache.ibatis.transaction.Transaction对象的，TransactionFactory接口默认有2个实现：
* org.apache.ibatis.transaction.managed.ManagedTransactionFactory  
  org.apache.ibatis.transaction.jdbc.JdbcTransactionFactory
* 一般情况下我们使用org.apache.ibatis.transaction.jdbc.JdbcTransactionFactory这个，mybatis和其他框架集成，比如和spring集成，事务交由spring去控制，spring中有TransactionFactory接口的一个实现org.mybatis.spring.transaction.SpringManagedTransactionFactory，有兴趣的朋友可以去研究一下，这个到时候讲到spring的使用会详细说。
* dataSource元素
* 这个用来配置数据源的，type属性的值必须为接口org.apache.ibatis.datasource.DataSourceFactory的实现类，DataSourceFactory也是一个工厂，用来创建数据源javax.sql.DataSource对象的，mybatis中这个接口默认有3个实现类：
* org.apache.ibatis.datasource.jndi.JndiDataSourceFactory  
  org.apache.ibatis.datasource.pooled.PooledDataSourceFactory  
  org.apache.ibatis.datasource.unpooled.UnpooledDataSourceFactory
* 我们使用第2个org.apache.ibatis.datasource.pooled.PooledDataSourceFactory，这个用来创建一个数据库连接池类型的数据源，可以实现数据库连接共用，减少连接重复创建销毁的时间。
* 配置数据源需要指定数据库连接的属性信息，比如：驱动、连接db的url、用户名、密码，这个在dataSource元素下面的property中配置，property元素的格式：
* <property name="属性名称" value="值"/>

#### 创建Mapper xml文件

我们需要对t\_user表进行操作，需要写sql，sql写在什么地方呢？

在mybatis中一般我们将一个表的所有sql操作写在一个mapper xml中，一般命名为XXXMapper.xml格式。

创建文件chat02/src/main/resource/mapper/UserMapper.xml，内容如下：

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper namespace="com.javacode2018.chat02.UserMapper">  
</mapper>

mapper xml根元素为mapper，这个元素有个namespace属性，系统中会有很多表，每个表对应一个Mapper xml，为了防止mapper文件重复，我们需要给每个mapper xml文件需要指定一个namespace，通过这个可以区分每个mapper xml文件，上面我们指定为com.javacode2018.chat02.UserMapper。

一会对t\_user表的所有操作相关的sql，我们都会写在上面这个xml中。

#### mybatis全局配置文件中引入Mapper xml文件

UserMapper.xml我们写好了，如何让mybatis知道这个文件呢，此时我们需要在mybatis-config.xml全局配置文件中引入UserMapper.xml，在mybatis-config.xml加入下面配置：

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

mappers元素下面有多个mapper元素，通过mapper元素的resource属性可以引入Mapper xml文件，resource是相对于classes的路径。

上面说的都是一些配置文件，配置文件都ok了，下面我们就需要将mybatis跑起来了，此时需要使用到mybatis中的一些java对象了。

#### 构建SqlSessionFactory对象

//指定mybatis全局配置文件  
String resource = "mybatis-config.xml";  
//读取全局配置文件  
InputStream inputStream = Resources.getResourceAsStream(resource);  
//构建SqlSessionFactory对象  
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

SqlSessionFactory是一个接口，是一个重量级的对象，SqlSessionFactoryBuilder通过读取全局配置文件来创建一个SqlSessionFactory，创建这个对象是比较耗时的，主要耗时在对mybatis全局配置文件的解析上面，全局配置文件中包含很多内容，SqlSessionFactoryBuilder通过解析这些内容，创建了一个复杂的SqlSessionFactory对象，这个对象的生命周期一般和应用的生命周期是一样的，随着应用的启动而创建，随着应用的停止而结束，所以一般是一个全局对象，一般情况下一个db对应一个SqlSessionFactory对象。

#### 构建SqlSession对象

SqlSession相当于jdbc中的Connection对象，相当于数据库的一个连接，可以用SqlSession来对db进行操作：**如执行sql、提交事务、关闭连接等等**，需要通过SqlSessionFactory来创建SqlSession对象，SqlSessionFactory中常用的有2个方法来创建SqlSession对象，如下：

//创建一个SqlSession，默认不会自动提交事务  
SqlSession openSession();  
//创建一个SqlSession,autoCommit：指定是否自动提交事务  
SqlSession openSession(boolean autoCommit);

SqlSession接口中很多方法，直接用来操作db，方法清单如下，大家眼熟一下：

<T> T selectOne(String statement);  
<T> T selectOne(String statement, Object parameter);  
<E> List<E> selectList(String statement);  
<E> List<E> selectList(String statement, Object parameter);  
<E> List<E> selectList(String statement, Object parameter, RowBounds rowBounds);  
<K, V> Map<K, V> selectMap(String statement, String mapKey);  
<K, V> Map<K, V> selectMap(String statement, Object parameter, String mapKey);  
<K, V> Map<K, V> selectMap(String statement, Object parameter, String mapKey, RowBounds rowBounds);  
<T> Cursor<T> selectCursor(String statement);  
<T> Cursor<T> selectCursor(String statement, Object parameter);  
<T> Cursor<T> selectCursor(String statement, Object parameter, RowBounds rowBounds);  
void select(String statement, Object parameter, ResultHandler handler);  
void select(String statement, ResultHandler handler);  
void select(String statement, Object parameter, RowBounds rowBounds, ResultHandler handler);  
int insert(String statement);  
int insert(String statement, Object parameter);  
int update(String statement);  
int update(String statement, Object parameter);  
int delete(String statement);  
int delete(String statement, Object parameter);  
void commit();  
void commit(boolean force);  
void rollback();  
void rollback(boolean force);  
List<BatchResult> flushStatements();  
void close();  
void clearCache();  
Configuration getConfiguration();  
<T> T getMapper(Class<T> type);  
Connection getConnection();

上面以select开头的可以对db进行查询操作，insert相关的可以对db进行插入操作，update相关的可以对db进行更新操作。

#### 引入lombok支持（非必须）

**声明一下：lombok不是mybatis必须的，为了简化代码而使用的，以后我们会经常使用。**

Lombok能以简单的注解形式来简化java代码，提高开发人员的开发效率。例如开发中经常需要写的javabean，都需要花时间去添加相应的getter/setter，也许还要去写构造器、equals等方法，而且需要维护，当属性多时会出现大量的getter/setter方法，这些显得很冗长也没有太多技术含量，一旦修改属性，就容易出现忘记修改对应方法的失误。

Lombok能通过注解的方式，在编译时自动为属性生成构造器、getter/setter、equals、hashcode、toString方法。出现的神奇就是在源码中没有getter和setter方法，但是在编译生成的字节码文件中有getter和setter方法。这样就省去了手动重建这些代码的麻烦，使代码看起来更简洁些。

##### lombok的使用步骤

1. 先在idea中安装lombok插件

* 打开idea，点击File->Settings->plugins，然后搜索Lombok Plugin，点击安装就可以了。

1. maven中引入lombok支持

* <dependency>  
   <groupId>org.projectlombok</groupId>  
   <artifactId>lombok</artifactId>  
   <version>1.18.10</version>  
   <scope>provided</scope>  
  </dependency>

1. 代码中使用lombok相关功能

#### 引入logback（非必须）

**声明一下：日志框架mybatis中也不是必须的，不用配置也可以正常运行。**

为了方便查看mybatis运行过程中产生的日志，比如：执行的sql、sql的参数、sql的执行结果等等调试信息，我们需要引入日志框架的支持，logback是一个很好的日志框架，此处我们就使用这个

##### mybatis中集成logback步骤

1. maven中引入logback支持

* <dependency>  
   <groupId>ch.qos.logback</groupId>  
   <artifactId>logback-classic</artifactId>  
   <version>1.2.3</version>  
  </dependency>

1. src/main/resources中创建logback.xml文件：

* <?xml version="1.0" encoding="UTF-8"?>  
  <configuration>  
   <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
   <encoder>  
   <pattern>%d{mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>  
   </encoder>  
   </appender>  
    
   <logger name="com.javacode2018" level="debug" additivity="false">  
   <appender-ref ref="STDOUT" />  
   </logger>  
    
  </configuration>
* logback.xml具体的写法不是本文讨论的范围，有兴趣的朋友可以去研究一下logback具体的用法。
* 上面xml中配置了com.javacode2018包中所有的类，使用logback输出日志的时候，debug级别及以上级别的日志会输出到控制台，方便我们查看。

#### 写一个测试用例

在chat02/src/test下创建一个类:

com.javacode2018.chat02.UserTest

内容如下：

package com.javacode2018.chat02;  
  
import lombok.extern.slf4j.Slf4j;  
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 java.io.IOException;  
import java.io.InputStream;  
  
@Slf4j  
public class UserTest {  
  
 private SqlSessionFactory sqlSessionFactory;  
  
 @Before  
 public void before() throws IOException {  
 //指定mybatis全局配置文件  
 String resource = "mybatis-config.xml";  
 //读取全局配置文件  
 InputStream inputStream = Resources.getResourceAsStream(resource);  
 //构建SqlSessionFactory对象  
 SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
 this.sqlSessionFactory = sqlSessionFactory;  
 }  
  
 @Test  
 public void sqlSession() {  
 SqlSession sqlSession = this.sqlSessionFactory.openSession();  
 log.info("{}", sqlSession);  
 }  
  
}

上面代码中有个@Slf4j注解，这个是lombok提供的，可以在这个类中生成下面代码：

private static final org.slf4j.Logger log = org.slf4j.LoggerFactory.getLogger(UserTest.class);

运行一下上面的用例：sqlSession方法，输出如下：

45:51.289 [main] INFO com.javacode2018.chat02.UserTest - org.apache.ibatis.session.defaults.DefaultSqlSession@1f021e6c

### 使用SqlSesion执行sql操作

#### SqlSession常见的用法

SqlSession相当于一个连接，可以使用这个对象对db执行增删改查操作，操作完毕之后需要关闭，使用步骤：

1.获取SqlSession对象：通过该sqlSessionFactory.openSession方法获取SqlSession对象  
2.对db进行操作：使用SqlSession对象进行db操作  
3.关闭SqlSession对象：sqlSession.close();

常见的使用方式如下：

//获取SqlSession  
SqlSession sqlSession = this.sqlSessionFactory.openSession();  
try {  
 //执行业务操作，如：增删改查  
} finally {  
 //关闭SqlSession  
 sqlSession.close();  
}

上面我们将SqlSession的关闭放在finally块中，确保close()一定会执行。更简单的方式是使用java中的try()的方式，如下：

try (SqlSession sqlSession = this.sqlSessionFactory.openSession();) {  
 //执行业务操作，如：增删改查  
}

#### 新增操作

需求：传入UserModel对象，然后将这个对象的数据插入到t\_user表中。

##### 创建一个UserModel

新建一个com.javacode2018.chat02.UserModel类，代码如下：

package com.javacode2018.chat02;  
  
import lombok.\*;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
@Getter  
@Setter  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
@ToString  
public class UserModel {  
 private Long id;  
 private String name;  
 private Integer age;  
 private Double salary;  
 private Integer sex;  
}

这个类的字段和t\_user表对应。

##### UserMapper.xml中定义插入操作

我们说过了，对t\_user表的所有sql操作，我们都放在UserMapper.xml中，我们在UserMapper.xml中加入下面配置，使用insert元素定义插入操作：

<!-- insert用来定义一个插入操作  
 id：操作的具体标识  
 parameterType：指定插入操作接受的参数类型  
 -->  
<insert id="insertUser" parameterType="com.javacode2018.chat02.UserModel">  
 <![CDATA[  
 INSERT INTO t\_user (id,name,age,salary,sex) VALUES (#{id},#{name},#{age},#{salary},#{sex})  
 ]]>  
</insert>

insert元素用来定义了一个对db的insert操作

**id**：是这个操作的一个标识，一会通过mybatis执行操作的时候会通过这个namespace和id引用到这个insert操作，

**parameterType**：用来指定这个insert操作接受的参数的类型，可以是：**各种javabean、map、list、collection类型的java对象**，我们这个插入接受的是UserModel对象。

insert元素内部定义了具体的sql，可以看到是一个insert的sql，向t\_user表插入数据。

需要插入的值从UserModel对象中获取，取UserModel对象的的字段，使用**#{字段}**这种格式可以获取到UserModel中字段的值。

##### 调用SqlSession.insert方法执行插入操作

t\_user插入的sql我们已经在UserMapper中写好，此时我们怎么调用呢？

需要调用SqlSession.insert方法：

int insert(String statement, Object parameter)

这个方法有2个参数:

**statement**：表示那个操作，值为Mapper xml的namespace.具体操作的id，如需要调用UserMapper.xml中的insertUser操作，这个值就是：

com.javacode2018.chat02.UserMapper.insertUser

**parameter**：insert操作的参数，和Mapper xml中的insert中的parameterType指定的类型一致。

返回值为插入的行数。

UserTest类中新增一个测试用例：

@Test  
public void insertUser() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(false);) {  
 //创建UserModel对象  
 UserModel userModel = UserModel.builder().id(2L).name("javacode2018").age(30).salary(50000D).sex(1).build();  
 //执行插入操作  
 int result = sqlSession.insert("com.javacode2018.chat02.UserMapper.insertUser", userModel);  
 log.info("插入影响行数：{}", result);  
 //提交事务  
 sqlSession.commit();  
 }  
}

运行输出如下：

01:46.683 [main] DEBUG c.j.chat02.UserMapper.insertUser - ==> Preparing: INSERT INTO t\_user (id,name,age,salary,sex) VALUES (?,?,?,?,?)   
01:46.745 [main] DEBUG c.j.chat02.UserMapper.insertUser - ==> Parameters: 2(Long), javacode2018(String), 30(Integer), 50000.0(Double), 1(Integer)  
01:46.751 [main] DEBUG c.j.chat02.UserMapper.insertUser - <== Updates: 1  
01:46.751 [main] INFO com.javacode2018.chat02.UserTest - 影响行数：1

输出中打印了详细的sql语句，以及sql的参数信息，可以看到Mapper xml中的#{}被替换为了?，这个使用到了jdbc中的PreparedStatement来对参数设置值。

输出中的第二行详细列出了参数的值以及每个值的类型。

第三行输出了insert的结果为1，表示插入成功了1行记录。

去db中看一下，如下，插入成功：

mysql> SELECT \* FROM t\_user;  
+----+---------------+-----+----------+-----+  
| id | name | age | salary | sex |  
+----+---------------+-----+----------+-----+  
| 1 | 马士兵老师 | 30 | 50000.00 | 1 |  
+----+---------------+-----+----------+-----+  
1 row in set (0.00 sec)

上面代码中创建SqlSession，我们使用的是sqlSessionFactory.openSession()创建的，这个方法创建的SqlSession，内部事务是非自动提交的方式，所以需要我们手动提交：

sqlSession.commit();

如果想自动提交事务，可以将上面的测试用例改成下面这样：

@Test  
public void insertUser() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 //创建UserModel对象  
 UserModel userModel = UserModel.builder().id(1L).name("马士兵老师").age(30).salary(50000D).sex(1).build();  
 //执行插入操作  
 int result = sqlSession.insert("com.javacode2018.chat02.UserMapper.insertUser", userModel);  
 log.info("影响行数：{}", result);  
 }  
}

上面在创建SqlSession的时候调用了sqlSessionFactory.openSession(true)，指定事务为自动提交模式，所以最后我们不需要手动提交事务了。

#### 更新操作

需求：传入UserModel对象，然后通过id更新数据。

##### UserMapper.xml中定义Update操作

使用update定义更新操作：

<!-- update用来定义一个更新操作  
 id：操作的具体标识  
 parameterType：指定操作接受的参数类型  
 -->  
<update id="updateUser" parameterType="com.javacode2018.chat02.UserModel">  
 <![CDATA[  
 UPDATE t\_user SET name = #{name},age = #{age},salary = #{salary},sex = #{sex} WHERE id = #{id}  
 ]]>  
</update>

写法和insert操作的写法类似，指定id标识、parameterType指定操作的参数类型，元素体中是具体的sql语句。

##### 调用SqlSession.update方法执行更新操作

需要调用SqlSession.update方法：

int update(String statement, Object parameter)

这个方法有2个参数:

**statement**：表示哪个操作，值为Mapper xml的namespace.具体操作的id，如需要调用UserMapper.xml中的updateUser操作，这个值就是：

com.javacode2018.chat02.UserMapper.updateUser

**parameter**：update操作的参数，和Mapper xml中的update中的parameterType指定的类型一致。

返回值为update影响行数。

UserTest类中新增一个测试用例：

@Test  
public void updateUser() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 //创建UserModel对象  
 UserModel userModel = UserModel.builder().id(1L).name("马士兵老师，你好").age(18).salary(5000D).sex(0).build();  
 //执行更新操作  
 int result = sqlSession.update("com.javacode2018.chat02.UserMapper.updateUser", userModel);  
 log.info("影响行数：{}", result);  
 }  
}

运行输出：

14:09.051 [main] DEBUG c.j.chat02.UserMapper.updateUser - ==> Preparing: UPDATE t\_user SET name = ?,age = ?,salary = ?,sex = ? WHERE id = ?   
14:09.095 [main] DEBUG c.j.chat02.UserMapper.updateUser - ==> Parameters: 马士兵老师，你好(String), 18(Integer), 5000.0(Double), 0(Integer), 1(Long)  
14:09.100 [main] DEBUG c.j.chat02.UserMapper.updateUser - <== Updates: 1  
14:09.101 [main] INFO com.javacode2018.chat02.UserTest - 影响行数：1

db中去看一下：

mysql> SELECT \* FROM t\_user;  
+----+------------------------+-----+----------+-----+  
| id | name | age | salary | sex |  
+----+------------------------+-----+----------+-----+  
| 1 | 马士兵老师，你好 | 18 | 5000.00 | 0 |  
| 2 | javacode2018 | 30 | 50000.00 | 1 |  
+----+------------------------+-----+----------+-----+  
2 rows in set (0.00 sec)

#### 删除操作

需求：根据用户的id删除对应的用户记录

##### UserMapper.xml中定义Delete操作

使用update元素定义删除操作：

<!-- update用来定义一个删除操作  
 id：操作的具体标识  
 parameterType：指定操作接受的参数类型  
 -->  
<update id="deleteUser" parameterType="java.lang.Long">  
 <![CDATA[  
 DELETE FROM t\_user WHERE id = #{id}  
 ]]>  
</update>

写法和update操作的写法类似，指定id标识、parameterType指定操作的参数类型，用户id为Long类型的，元素体中是具体的delete语句。

##### 调用SqlSession.update方法执行更新操作

需要调用SqlSession.delete方法：

int delete(String statement, Object parameter)

这个方法有2个参数:

**statement**：表示哪个操作，值为Mapper xml的namespace.具体操作的id，如需要调用UserMapper.xml中的deleteUser操作，这个值就是：

com.javacode2018.chat02.UserMapper.

**parameter**：delete操作的参数，和Mapper xml中的delete中的parameterType指定的类型一致。

返回值为delete影响行数。

UserTest类中新增一个测试用例：

@Test  
public void deleteUser() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 //定义需要删除的用户id  
 Long userId = 1L;  
 //执行删除操作  
 int result = sqlSession.delete("com.javacode2018.chat02.UserMapper.deleteUser", userId);  
 log.info("影响行数：{}", result);  
 }  
}

运行输出：

24:45.427 [main] DEBUG c.j.chat02.UserMapper.deleteUser - ==> Preparing: DELETE FROM t\_user WHERE id = ?   
24:45.476 [main] DEBUG c.j.chat02.UserMapper.deleteUser - ==> Parameters: 1(Long)  
24:45.485 [main] DEBUG c.j.chat02.UserMapper.deleteUser - <== Updates: 1  
24:45.485 [main] INFO com.javacode2018.chat02.UserTest - 影响行数：1

#### 执行查询

需求：查询所有用户信息

##### UserMapper.xml中定义Select操作

<!-- select用来定义一个查询操作  
 id：操作的具体标识  
 resultType：指定查询结果保存的类型  
 -->  
<select id="getUserList" resultType="com.javacode2018.chat02.UserModel">  
 <![CDATA[  
 SELECT \* FROM t\_user  
 ]]>  
</select>

写法和update操作的写法类似，指定id标识、parameterType指定操作的参数类型，resultType指定查询结果的类型，元素体中是具体的select语句。

##### 调用SqlSession.select方法执行更新操作

UserTest添加一个用例：

@Test  
public void getUserList() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 //执行查询操作  
 List<UserModel> userModelList = sqlSession.selectList("com.javacode2018.chat02.UserMapper.getUserList");  
 log.info("结果：{}", userModelList);  
 }  
}

多插入几行，然后运行上面的用例，输出如下：

36:39.015 [main] DEBUG c.j.chat02.UserMapper.getUserList - ==> Preparing: SELECT \* FROM t\_user   
36:39.048 [main] DEBUG c.j.chat02.UserMapper.getUserList - ==> Parameters:   
36:39.066 [main] DEBUG c.j.chat02.UserMapper.getUserList - <== Total: 3  
36:39.067 [main] INFO com.javacode2018.chat02.UserTest - UserModel(id=2, name=javacode2018, age=30, salary=50000.0, sex=1)  
36:39.069 [main] INFO com.javacode2018.chat02.UserTest - UserModel(id=1575621274235, name=马士兵老师, age=30, salary=50000.0, sex=1)  
36:39.069 [main] INFO com.javacode2018.chat02.UserTest - UserModel(id=1575621329823, name=马士兵老师, age=30, salary=50000.0, sex=1)

### Mapper接口的使用

#### 为什么需要Mapper接口

上面我们讲解了对一个表的增删改查操作，都是通过调用SqlSession中的方法来完成的，大家再来看一下SqlSession接口中刚才用到的几个方法的定义：

int insert(String statement, Object parameter);  
int update(String statement, Object parameter);  
int delete(String statement, Object parameter);  
<E> List<E> selectList(String statement);

这些方法的特点我们来看一下：

1. 调用这些方法，需要明确知道statement的值，statement的值为namespace.具体操作的id，这些需要打开Mapper xml中去查看了才知道，写起来不方便
2. parameter参数都是Object类型的，我们根本不知道这个操作具体类型是什么，需要查看Mapper xml才知道，随便传递个值，可能类型不匹配，但是只有在运行的时候才知道有问题
3. selectList方法返回的是一个泛型类型的，通过这个方法我们根本不知道返回的结果的具体类型，也需要去查看Mapper xml才知道

以上这几点使用都不是太方便，有什么方法能解决上面这些问题么？

有，这就是mybatis中的Mapper接口，我们可以定义一个interface，然后和Mapper xml关联起来，Mapper xml中的操作和Mapper接口中的方法会进行绑定，当我们调用Mapper接口的方法的时候，会间接调用到Mapper xml中的操作，接口的完整类名需要和Mapper xml中的namespace一致。

#### Mapper接口的用法（三步）

##### 步骤1：定义Mapper接口

去看一下，UserMapper.xml中的namespace，是：

<mapper namespace="com.javacode2018.chat02.UserMapper">

我们创建的接口完整的名称需要和上面的namespace的值一样，下面我们创建一个接口com.javacode2018.chat02.UserMapper，如下：

package com.javacode2018.chat02;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
public interface UserMapper {  
}

UserMapper.xml中有4个操作，我们需要在UserMapper接口中也定义4个操作，和UserMapper.xml的4个操作对应，如下：

package com.javacode2018.chat02;  
  
import java.util.List;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
public interface UserMapper {  
  
 int insertUser(UserModel model);  
  
 int updateUser(UserModel model);  
  
 int deleteUser(Long userId);  
  
 List<UserModel> getUserList();  
}

UserMapper接口中定义了4个方法，方法的名称需要和UserMapper.xml具体操作的id值一样，这样调用UserMapper接口中的方法的时候，才会对应的找到UserMapper.xml中具体的操作。

比如调用UserMapper接口中的insertUser方法，mybatis查找的规则是：通过接口完整名称.方法名称去Mapper xml中找到对应的操作。

##### 步骤2：通过SqlSession获取Mapper接口对象

SqlSession中有个getMapper方法，可以传入接口的类型，获取具体的Mapper接口对象，如下：

/\*\*  
 \* Retrieves a mapper.  
 \* @param <T> the mapper type  
 \* @param type Mapper interface class  
 \* @return a mapper bound to this SqlSession  
 \*/  
 <T> T getMapper(Class<T> type);

如获取UserMapper接口对象：

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

##### 步骤3：调用Mapper接口的方法对db进行操作

如调用UserMapper接口的insert操作：

@Test  
public void insertUser() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 //创建UserModel对象  
 UserModel userModel = UserModel.builder().id(System.currentTimeMillis()).name("马士兵老师").age(30).salary(50000D).sex(1).build();  
 //执行插入操作  
 int insert = mapper.insertUser(userModel);  
 log.info("影响行数：{}", insert);  
 }  
}

#### 案例：使用Mapper接口来实现增删改查

chat02/src/test/java中创建一个测试类，代码如下：

package com.javacode2018.chat02;  
  
import lombok.extern.slf4j.Slf4j;  
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 java.io.IOException;  
import java.io.InputStream;  
import java.util.List;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
@Slf4j  
public class UserMapperTest {  
 private SqlSessionFactory sqlSessionFactory;  
  
 @Before  
 public void before() throws IOException {  
 //指定mybatis全局配置文件  
 String resource = "mybatis-config.xml";  
 //读取全局配置文件  
 InputStream inputStream = Resources.getResourceAsStream(resource);  
 //构建SqlSessionFactory对象  
 SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
 this.sqlSessionFactory = sqlSessionFactory;  
 }  
  
 @Test  
 public void insertUser() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 //创建UserModel对象  
 UserModel userModel = UserModel.builder().id(System.currentTimeMillis()).name("马士兵老师").age(30).salary(50000D).sex(1).build();  
 //执行插入操作  
 int insert = mapper.insertUser(userModel);  
 log.info("影响行数：{}", insert);  
 }  
 }  
  
 @Test  
 public void updateUser() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 //创建UserModel对象  
 UserModel userModel = UserModel.builder().id(1L).name("马士兵老师，你好").age(18).salary(5000D).sex(0).build();  
 //执行更新操作  
 int result = mapper.updateUser(userModel);  
 log.info("影响行数：{}", result);  
 }  
 }  
  
 @Test  
 public void deleteUser() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 //定义需要删除的用户id  
 Long userId = 1L;  
 //执行删除操作  
 int result = mapper.deleteUser(userId);  
 log.info("影响行数：{}", result);  
 }  
 }  
  
 @Test  
 public void getUserList() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 //执行查询操作  
 List<UserModel> userModelList = mapper.getUserList();  
 userModelList.forEach(item -> {  
 log.info("{}", item);  
 });  
 }  
 }  
}

大家认真看一下上面的代码，这次我们使用了UserMapper来间接调用UserMapper.xml中对应的操作，可以去运行一下感受一下效果。

#### Mapper接口使用时注意的几点

1. Mapper接口的完整类名必须和对应的Mapper xml中的namespace的值一致
2. Mapper接口中方法的名称需要和Mapper xml中具体操作的id值一致
3. Mapper接口中方法的参数、返回值可以不和Mapper xml中的一致

#### Mapper接口的原理

这个使用java中的动态代理实现的，mybatis启动的时候会加载全局配置文件mybatis-config.xml，然后解析这个文件中的mapper元素指定的UserMapper.xml，会根据UserMapper.xml的namespace的值创建这个接口的一个动态代理，具体可以去看一下mybatis的源码，主要使用java中的Proxy实现的，使用java.lang.reflect.Proxy类中的newProxyInstance方法，我们可以创建任意一个接口的一个代理对象：

public static Object newProxyInstance(ClassLoader loader,  
 Class<?>[] interfaces,  
 InvocationHandler h)

我们使用Proxy来模仿Mapper接口的实现：

package com.javacode2018.chat02;  
  
import lombok.extern.slf4j.Slf4j;  
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 java.io.IOException;  
import java.io.InputStream;  
import java.lang.reflect.InvocationHandler;  
import java.lang.reflect.Method;  
import java.lang.reflect.Proxy;  
import java.util.List;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
@Slf4j  
public class ProxyTest {  
  
 public static class UserMapperProxy implements InvocationHandler {  
 private SqlSession sqlSession;  
  
 private Class<?> mapperClass;  
  
 public UserMapperProxy(SqlSession sqlSession, Class<?> mapperClass) {  
 this.sqlSession = sqlSession;  
 this.mapperClass = mapperClass;  
 }  
  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 log.debug("invoke start");  
 String statement = mapperClass.getName() + "." + method.getName();  
 List<Object> result = sqlSession.selectList(statement);  
 log.debug("invoke end");  
 return result;  
 }  
 }  
  
 private SqlSessionFactory sqlSessionFactory;  
  
 @Before  
 public void before() throws IOException {  
 //指定mybatis全局配置文件  
 String resource = "mybatis-config.xml";  
 //读取全局配置文件  
 InputStream inputStream = Resources.getResourceAsStream(resource);  
 //构建SqlSessionFactory对象  
 SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
 this.sqlSessionFactory = sqlSessionFactory;  
 }  
  
 @Test  
 public void test1() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper userMapper = (UserMapper) Proxy.newProxyInstance(ProxyTest.class.getClassLoader(), new Class[]{UserMapper.class}, new UserMapperProxy(sqlSession, UserMapper.class));  
 log.info("{}", userMapper.getUserList());  
 }  
 }  
}

上面代码中：UserMapper是没有实现类的，可以通过Proxy.newProxyInstance给UserMapper接口创建一个代理对象，当调用UserMapper接口的方法的时候，会调用到UserMapperProxy对象的invoke方法。

运行一下test1用例，输出如下：

16:34.288 [main] DEBUG com.javacode2018.chat02.ProxyTest - invoke start  
16:34.555 [main] DEBUG c.j.chat02.UserMapper.getUserList - ==> Preparing: SELECT \* FROM t\_user   
16:34.580 [main] DEBUG c.j.chat02.UserMapper.getUserList - ==> Parameters:   
16:34.597 [main] DEBUG c.j.chat02.UserMapper.getUserList - <== Total: 4  
16:34.597 [main] DEBUG com.javacode2018.chat02.ProxyTest - invoke end  
16:34.597 [main] INFO com.javacode2018.chat02.ProxyTest - [UserModel(id=2, name=javacode2018, age=30, salary=50000.0, sex=1), UserModel(id=1575621274235, name=马士兵老师, age=30, salary=50000.0, sex=1), UserModel(id=1575621329823, name=马士兵老师, age=30, salary=50000.0, sex=1), UserModel(id=1575623283897, name=马士兵老师, age=30, salary=50000.0, sex=1)]

注意上面输出的invoke start和invoke end，可以看到我们调用userMapper.getUserList时候，被UserMapperProxy#invoke方法处理了。

Mybatis中创建Mapper接口代理对象使用的是下面这个类，大家可以去研究一下：

public class MapperProxyFactory<T> {  
  
 private final Class<T> mapperInterface;  
 private final Map<Method, MapperMethod> methodCache = new ConcurrentHashMap<Method, MapperMethod>();  
  
 public MapperProxyFactory(Class<T> mapperInterface) {  
 this.mapperInterface = mapperInterface;  
 }  
  
 public Class<T> getMapperInterface() {  
 return mapperInterface;  
 }  
  
 public Map<Method, MapperMethod> getMethodCache() {  
 return methodCache;  
 }  
  
 @SuppressWarnings("unchecked")  
 protected T newInstance(MapperProxy<T> mapperProxy) {  
 return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new Class[] { mapperInterface }, mapperProxy);  
 }  
  
 public T newInstance(SqlSession sqlSession) {  
 final MapperProxy<T> mapperProxy = new MapperProxy<T>(sqlSession, mapperInterface, methodCache);  
 return newInstance(mapperProxy);  
 }  
  
}

### MyBatis系列

1. MyBatis系列第1篇：MyBatis未出世之前我们那些痛苦的经历
2. MyBatis系列第2篇：入门篇，带你感受一下mybatis独特的魅力！

## Mybatis系列第4篇：Mybatis使用详解(2)

### 主要内容

1. idea创建本篇案例
   * 建库建表
   * 创建项目
2. 别名使用详解(typeAliases)
   * 为什么需要使用别名
   * 别名3种用法详解
   * 方式1：使用typeAlias元素注册别名
   * 方式2：使用package元素批量注册别名
   * 方式3：使用package结合@Alias批量注册并指定别名的名称
   * 别名不区分大小写
   * mybatis内置的别名
   * 别名的原理
   * 别名使用建议
3. 属性配置详解(properties)
   * 属性配置的3种方式
   * 方式1：通过propertie元素配置属性
   * 方式2：方式通过resource引用classpath中的属性配置文件
   * 方式3：通过url引用外部属性配置文件
   * 使用建议
   * 相关问题
4. mybatis中引入mapper的3种方式
   * 方式1：通过mapper元素resource属性的方式注册Mapper xml文件和Mapper接口
   * 方式2：通过mapper元素class属性的方式注册Mapper接口和Mapper xml文件
   * 方式3：通过package元素批量注册Mapper接口和Mapper xml文件
   * 源码解释
   * 使用注意

### idea创建案例

#### 建库建表

/\*创建数据库javacode2018\*/  
DROP DATABASE IF EXISTS `javacode2018`;  
CREATE DATABASE `javacode2018`;  
USE `javacode2018`;  
  
/\*创建表结构\*/  
DROP TABLE IF EXISTS `t\_user`;  
CREATE TABLE t\_user (  
 id BIGINT AUTO\_INCREMENT PRIMARY KEY COMMENT '主键，用户id，自动增长',  
 `name` VARCHAR(32) NOT NULL DEFAULT '' COMMENT '姓名',  
 `age` SMALLINT NOT NULL DEFAULT 1 COMMENT '年龄',  
 `salary` DECIMAL(12,2) NOT NULL DEFAULT 0 COMMENT '薪水',  
 `sex` TINYINT NOT NULL DEFAULT 0 COMMENT '性别,0:未知,1:男,2:女'  
) COMMENT '用户表';  
  
DROP TABLE IF EXISTS `t\_order`;  
CREATE TABLE t\_order (  
 id BIGINT AUTO\_INCREMENT PRIMARY KEY COMMENT '主键，订单id，自动增长',  
 `user\_id` BIGINT NOT NULL DEFAULT 0 COMMENT '用户id',  
 `price` DECIMAL(12,2) NOT NULL DEFAULT 0 COMMENT '订单金额'  
) COMMENT '订单表';  
  
/\*插入几条测试数据\*/  
INSERT INTO t\_user (`name`,`age`,`salary`,`sex`)  
VALUES  
 ('马士兵老师',30,50000,1),  
 ('javacode2018',30,50000,1),  
 ('张学友',56,500000,1),  
 ('林志玲',45,88888.88,2);  
  
  
INSERT INTO t\_order (`user\_id`,`price`)  
VALUES  
 (1,88.88),  
 (2,666.66);  
  
SELECT \* FROM t\_user;  
SELECT \* FROM t\_order;

#### 创建工程

整个mybatis系列的代码采用maven模块的方式管理的，可以在文章底部获取，本次我们还是在上一篇的mybatis-series中进行开发，在这个项目中新建一个模块chat03，模块坐标如下：

<groupId>com.javacode2018</groupId>  
<artifactId>chat03</artifactId>  
<version>1.0-SNAPSHOT</version>

下面我们通过mybatis快速来实现对t\_user表增删改查，这个在上一篇的chat02中已经详细讲解过了。

##### 创建mybatis配置文件

chat03\src\main\resources\demo1目录创建，mybatis-config.xml，如下：

<?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>  
 <typeAliases></typeAliases>  
 <!-- 环境配置，可以配置多个环境 -->  
 <environments default="chat04-demo1">  
 <!--   
 environment用来对某个环境进行配置  
 id：环境标识，唯一  
 -->  
 <environment id="chat04-demo1">  
 <!-- 事务管理器工厂配置 -->  
 <transactionManager type="org.apache.ibatis.transaction.jdbc.JdbcTransactionFactory"/>  
 <!-- 数据源工厂配置，使用工厂来创建数据源 -->  
 <dataSource type="org.apache.ibatis.datasource.pooled.PooledDataSourceFactory">  
 <property name="driver" value="com.mysql.jdbc.Driver"/>  
 <property name="url" value="jdbc:mysql://localhost:3306/javacode2018?characterEncoding=UTF-8"/>  
 <property name="username" value="root"/>  
 <property name="password" value="root123"/>  
 </dataSource>  
 </environment>  
 </environments>  
  
 <mappers>  
 <mapper resource="demo1/mapper/UserMapper.xml"/>  
 </mappers>  
</configuration>

##### 创建UserMapper.xml文件

chat03\src\main\resources\demo1\mapper目录创建，UserMapper.xml，如下：

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper namespace="com.javacode2018.chat03.demo1.UserMapper">  
 <!-- insert用来定义一个插入操作  
 id：操作的具体标识  
 parameterType：指定插入操作接受的参数类型  
 -->  
 <insert id="insertUser" parameterType="com.javacode2018.chat03.demo1.UserModel" useGeneratedKeys="true">  
 <![CDATA[  
 INSERT INTO t\_user (id,name,age,salary,sex) VALUES (#{id},#{name},#{age},#{salary},#{sex})  
 ]]>  
 </insert>  
  
 <!-- update用来定义一个更新操作  
 id：操作的具体标识  
 parameterType：指定操作接受的参数类型  
 -->  
 <update id="updateUser" parameterType="com.javacode2018.chat03.demo1.UserModel">  
 <![CDATA[  
 UPDATE t\_user SET name = #{name},age = #{age},salary = #{salary},sex = #{sex} WHERE id = #{id}  
 ]]>  
 </update>  
  
 <!-- update用来定义一个删除操作  
 id：操作的具体标识  
 parameterType：指定操作接受的参数类型  
 -->  
 <update id="deleteUser" parameterType="java.lang.Long">  
 <![CDATA[  
 DELETE FROM t\_user WHERE id = #{id}  
 ]]>  
 </update>  
  
 <!-- select用来定义一个查询操作  
 id：操作的具体标识  
 resultType：指定查询结果保存的类型  
 -->  
 <select id="getUserList" resultType="com.javacode2018.chat03.demo1.UserModel">  
 <![CDATA[  
 SELECT \* FROM t\_user  
 ]]>  
 </select>  
  
</mapper>

##### 创建UserModel类

chat03\src\main\java\com\javacode2018\chat03\demo1目录创建UserModel.java，如下：

package com.javacode2018.chat03.demo1;  
  
import lombok.\*;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
@Getter  
@Setter  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
@ToString  
public class UserModel {  
 private Long id;  
 private String name;  
 private Integer age;  
 private Double salary;  
 private Integer sex;  
}

##### 创建UserMapper接口

chat03\src\main\java\com\javacode2018\chat03\demo1目录创建UserMapper.java，如下：

package com.javacode2018.chat03.demo1;  
  
import java.util.List;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
public interface UserMapper {  
  
 int insertUser(UserModel model);  
  
 int updateUser(UserModel model);  
  
 int deleteUser(Long userId);  
  
 List<UserModel> getUserList();  
}

##### 引入logback日志支持

chat03\src\main\resources目录创建logback.xml，如下：

<?xml version="1.0" encoding="UTF-8"?>  
<configuration>  
 <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
 <encoder>  
 <pattern>%d{mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>  
 </encoder>  
 </appender>  
  
 <logger name="com.javacode2018" level="debug" additivity="false">  
 <appender-ref ref="STDOUT" />  
 </logger>  
  
</configuration>

##### 创建测试用例UserMapperTest

chat03\src\test\java\com\javacode2018\chat03\demo1目录创建UserMapperTest.java，如下：

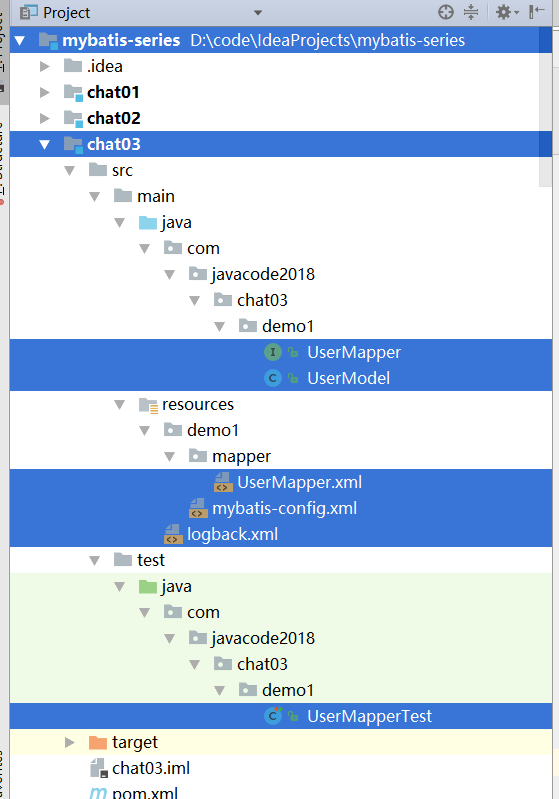
package com.javacode2018.chat03.demo1;  
  
import lombok.extern.slf4j.Slf4j;  
import org.apache.ibatis.io.Resources;  
import org.apache.ibatis.session.SqlSession;  
import org.apache.ibatis.session.SqlSessionFactory;  
import org.apache.ibatis.session.SqlSessionFactoryBuilder;  
import org.junit.Before;  
import org.junit.Test;  
  
import java.io.IOException;  
import java.io.InputStream;  
import java.util.List;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
@Slf4j  
public class UserMapperTest {  
 private SqlSessionFactory sqlSessionFactory;  
  
 @Before  
 public void before() throws IOException {  
 //指定mybatis全局配置文件  
 String resource = "demo1/mybatis-config.xml";  
 //读取全局配置文件  
 InputStream inputStream = Resources.getResourceAsStream(resource);  
 //构建SqlSessionFactory对象  
 SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
 this.sqlSessionFactory = sqlSessionFactory;  
 }  
  
 @Test  
 public void getUserList() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 //执行查询操作  
 List<UserModel> userModelList = mapper.getUserList();  
 userModelList.forEach(item -> {  
 log.info("{}", item);  
 });  
 }  
 }  
  
}

代码解释一下：

上面的before()方法上面有个@Before注解，这个是junit提供的一个注解，通过junit运行每个@Test标注的方法之前，会先运行被@before标注的方法，before()方法中我们创建了SqlSessionFactory对象，所以其他的@Test标注的方法中可以直接使用sqlSessionFactory对象了。

##### 项目结构如下图

注意项目结构如下图，跑起来有问题的可以对照一下。



##### 运行一下测试用例看效果

运行一下UserMapperTest.getUserList()方法，输出如下：

32:21.991 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - ==> Preparing: SELECT \* FROM t\_user   
32:22.028 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - ==> Parameters:   
32:22.052 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - <== Total: 4  
32:22.053 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)  
32:22.056 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=2, name=javacode2018, age=30, salary=50000.0, sex=1)  
32:22.056 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=3, name=张学友, age=56, salary=500000.0, sex=1)  
32:22.056 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=4, name=林志玲, age=45, salary=88888.88, sex=2)

上面是mybatis开发项目的一个玩转的步骤，希望大家都能够熟练掌握，下面我们来在这个示例的基础上讲解本章的知识点。

### 别名

#### 为什么需要使用别名?

大家打开chat03\src\main\resources\demo1\mapper\UserMapper.xml文件看一下，是不是有很多下面这样的代码：

parameterType="com.javacode2018.chat03.demo1.UserModel"  
resultType="com.javacode2018.chat03.demo1.UserModel"

parameterType是指定参数的类型，resultType是指定查询结果返回值的类型，他们的值都是UserModel类完整的类名，比较长，mybatis支持我们给某个类型起一个别名，然后通过别名可以访问到指定的类型。

#### 别名的用法

使用别名之前需要先在mybatis中注册别名，我们先说通过mybatis全局配置文件中注册别名，通过mybatis配置文件注册别名有3种方式。

#### 方式1

##### 使用typeAlias元素进行注册

如下：

<typeAliases>  
 <typeAlias type="玩转的类型名称" alias="别名" />  
</typeAliases>

**typeAliases元素中可以包含多个typeAlias子元素，每个typeAlias可以给一个类型注册别名，有2个属性需要指定：**

**type：完整的类型名称**

**alias：别名**

如上面给UserModel起了一个别名为user。

##### 案例

给UserModel注册一个别名user

chat03\src\main\resources\demo1\mapper\UserMapper.xml中加入下面配置：

<typeAliases>  
 <typeAlias type="com.javacode2018.chat03.demo1.UserModel" alias="user" />  
</typeAliases>

UserMapper.xml中使用别名，将chat03\src\main\resources\demo1\mapper\UserMapper.xml中getUserList的resultType的值改为user，如下：

<select id="getUserList" resultType="user">  
 <![CDATA[  
 SELECT \* FROM t\_user  
 ]]>  
</select>

运行com.javacode2018.chat03.demo1.UserMapperTest#getUserList，如下：

07:35.477 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - ==> Preparing: SELECT \* FROM t\_user   
07:35.505 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - ==> Parameters:   
07:35.527 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - <== Total: 4  
07:35.527 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)  
07:35.529 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=2, name=javacode2018, age=30, salary=50000.0, sex=1)  
07:35.529 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=3, name=张学友, age=56, salary=500000.0, sex=1)  
07:35.529 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=4, name=林志玲, age=45, salary=88888.88, sex=2)

看到了么，getUserList中我们使用的别名，运行是正常的，说明可以通过别名user直接访问UserModel。

#### 方式2

##### 通过packege元素批量注册

上面我们通过typeAlias元素可以注册一个别名，如果我们有很多类需要注册，需要写很多typeAlias配置。

mybatis为我们提供了批量注册别名的方式，通过package元素，如下：

<typeAliases>  
 <package name="需要扫描的包"/>  
</typeAliases>

这个也是在typeAliases元素下面，不过这次使用的是package元素，package有个name属性，可以指定一个包名，mybatis会加载这个包以及子包中所有的类型，给这些类型都注册别名，别名名称默认会采用类名小写的方式，如UserModel的别名为usermodel

##### 案例

下面我们将demo1/mybatis-config.xml中typeAliases元素的值改为下面这样：

<typeAliases>  
 <package name="com.javacode2018.chat03.demo1"/>  
</typeAliases>

mybatis会给com.javacode2018.chat03.demo1包及子包中的所有类型注册别名，UserModel类在这个包中，会被注册，别名为usermodel

UserMapper.xml中使用别名，将chat03\src\main\resources\demo1\mapper\UserMapper.xml中getUserList的resultType的值改为usermodel，如下：

<select id="getUserList" resultType="usermodel">  
 <![CDATA[  
 SELECT \* FROM t\_user  
 ]]>  
</select>

上面我们将返回值的类型resultType的值改为了usermodel

我们来运行com.javacode2018.chat03.demo1.UserMapperTest#getUserList，如下：

26:08.267 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - ==> Preparing: SELECT \* FROM t\_user   
26:08.296 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - ==> Parameters:   
26:08.318 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - <== Total: 4  
26:08.319 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)  
26:08.320 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=2, name=javacode2018, age=30, salary=50000.0, sex=1)  
26:08.320 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=3, name=张学友, age=56, salary=500000.0, sex=1)  
26:08.320 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=4, name=林志玲, age=45, salary=88888.88, sex=2)

看到了么，getUserList中我们使用的别名usermodel，运行也是正常的。

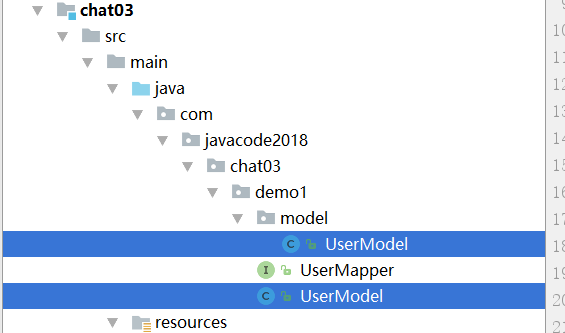
#### 方式3

##### package结合@Alias批量注册并指定别名

方式2中通过package可以批量注册别名，如果指定的包中包含了多个类名相同的类，会怎么样呢？

我们在com.javacode2018.chat03.demo1.model包中创建一个和UserModel同名的类，如下：

package com.javacode2018.chat03.demo1.model;  
  
import lombok.\*;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
@Getter  
@Setter  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
@ToString  
public class UserModel {  
 private Long id;  
 private String name;  
 private Integer age;  
 private Double salary;  
 private Integer sex;  
}



现在com.javacode2018.demo1包中有2个UserModel类了

运行com.javacode2018.chat03.demo1.UserMapperTest#getUserList，如下：

org.apache.ibatis.exceptions.PersistenceException:   
### Error building SqlSession.  
### The error may exist in SQL Mapper Configuration  
### Cause: org.apache.ibatis.builder.BuilderException: Error parsing SQL Mapper Configuration. Cause: org.apache.ibatis.type.TypeException: The alias 'UserModel' is already mapped to the value 'com.javacode2018.chat03.demo1.model.UserModel'.  
  
 at org.apache.ibatis.exceptions.ExceptionFactory.wrapException(ExceptionFactory.java:30)  
 at org.apache.ibatis.session.SqlSessionFactoryBuilder.build(SqlSessionFactoryBuilder.java:80)  
 at org.apache.ibatis.session.SqlSessionFactoryBuilder.build(SqlSessionFactoryBuilder.java:64)  
 at com.javacode2018.chat03.demo1.UserMapperTest.before(UserMapperTest.java:29)  
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)  
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)  
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)  
 at java.lang.reflect.Method.invoke(Method.java:498)  
 at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:50)  
 at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)  
 at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:47)  
 at org.junit.internal.runners.statements.RunBefores.evaluate(RunBefores.java:24)  
 at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:325)  
 at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:78)  
 at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:57)  
 at org.junit.runners.ParentRunner$3.run(ParentRunner.java:290)  
 at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:71)  
 at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:288)  
 at org.junit.runners.ParentRunner.access$000(ParentRunner.java:58)  
 at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:268)  
 at org.junit.runners.ParentRunner.run(ParentRunner.java:363)  
 at org.junit.runner.JUnitCore.run(JUnitCore.java:137)  
 at com.intellij.junit4.JUnit4IdeaTestRunner.startRunnerWithArgs(JUnit4IdeaTestRunner.java:68)  
 at com.intellij.rt.execution.junit.IdeaTestRunner$Repeater.startRunnerWithArgs(IdeaTestRunner.java:51)  
 at com.intellij.rt.execution.junit.JUnitStarter.prepareStreamsAndStart(JUnitStarter.java:242)  
 at com.intellij.rt.execution.junit.JUnitStarter.main(JUnitStarter.java:70)  
Caused by: org.apache.ibatis.builder.BuilderException: Error parsing SQL Mapper Configuration. Cause: org.apache.ibatis.type.TypeException: The alias 'UserModel' is already mapped to the value 'com.javacode2018.chat03.demo1.model.UserModel'.  
 at org.apache.ibatis.builder.xml.XMLConfigBuilder.parseConfiguration(XMLConfigBuilder.java:121)  
 at org.apache.ibatis.builder.xml.XMLConfigBuilder.parse(XMLConfigBuilder.java:98)  
 at org.apache.ibatis.session.SqlSessionFactoryBuilder.build(SqlSessionFactoryBuilder.java:78)  
 ... 24 more  
Caused by: org.apache.ibatis.type.TypeException: The alias 'UserModel' is already mapped to the value 'com.javacode2018.chat03.demo1.model.UserModel'.  
 at org.apache.ibatis.type.TypeAliasRegistry.registerAlias(TypeAliasRegistry.java:157)  
 at org.apache.ibatis.type.TypeAliasRegistry.registerAlias(TypeAliasRegistry.java:147)  
 at org.apache.ibatis.type.TypeAliasRegistry.registerAliases(TypeAliasRegistry.java:136)  
 at org.apache.ibatis.type.TypeAliasRegistry.registerAliases(TypeAliasRegistry.java:125)  
 at org.apache.ibatis.builder.xml.XMLConfigBuilder.typeAliasesElement(XMLConfigBuilder.java:164)  
 at org.apache.ibatis.builder.xml.XMLConfigBuilder.parseConfiguration(XMLConfigBuilder.java:109)  
 ... 26 more

报错了，2个类的类名一样了，默认都会使用usermodel作为别名，别名重复了mybatis会报错，那么此时我们怎么办呢？

package方式批量注册别名的时候，我们可以给类中添加一个@Alias注解来给这个类指定别名：

@Alias("user")  
public class UserModel {  
}

当mybatis扫描类的时候，发现类上有Alias注解，会取这个注解的value作为别名，如果没有这个注解，会将类名小写作为别名，如同方式2。

##### 案例

我们在com.javacode2018.chat03.demo1.UserModel类上加上下面注解：

@Alias("use")  
public class UserModel {  
}

修改demo1/mapper/UserMapper.xml，将resultType的值设置为user：

<select id="getUserList" resultType="user">  
 <![CDATA[  
 SELECT \* FROM t\_user  
 ]]>  
</select>

再来运行com.javacode2018.chat03.demo1.UserMapperTest#getUserList，如下：

18:51.219 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - ==> Preparing: SELECT \* FROM t\_user   
18:51.250 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - ==> Parameters:   
18:51.271 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - <== Total: 4  
18:51.272 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)  
18:51.274 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=2, name=javacode2018, age=30, salary=50000.0, sex=1)  
18:51.274 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=3, name=张学友, age=56, salary=500000.0, sex=1)  
18:51.274 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=4, name=林志玲, age=45, salary=88888.88, sex=2)

输出正常。

#### 别名不区分大小写

我们可以将上面UserMapper.xml中的use别名改成大写的：USER，如下：

<select id="getUserList" resultType="USER">  
 <![CDATA[  
 SELECT \* FROM t\_user  
 ]]>  
</select>

然后再运行一下com.javacode2018.chat03.demo1.UserMapperTest#getUserList，如下：

42:49.474 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - ==> Preparing: SELECT \* FROM t\_user   
42:49.509 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - ==> Parameters:   
42:49.527 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - <== Total: 4  
42:49.528 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)  
42:49.530 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=2, name=javacode2018, age=30, salary=50000.0, sex=1)  
42:49.530 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=3, name=张学友, age=56, salary=500000.0, sex=1)  
42:49.531 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=4, name=林志玲, age=45, salary=88888.88, sex=2)

也是正常的，**说明别名使用时是不区分大小写的。**

#### mybatis内置的别名

mybatis默认为很多类型提供了别名，如下：

|  |  |
| --- | --- |
| 别名 | 对应的实际类型 |
| \_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 |

上面这些默认都是在org.apache.ibatis.type.TypeAliasRegistry类中进行注册的，这个类就是mybatis注册别名使用的，别名和具体的类型关联是放在这个类的一个map属性（typeAliases）中，贴一部分代码大家感受一下：

public class TypeAliasRegistry {  
  
 private final Map<String, Class<?>> typeAliases = new HashMap<>();  
  
 public TypeAliasRegistry() {  
 registerAlias("string", String.class);  
  
 registerAlias("byte", Byte.class);  
 registerAlias("long", Long.class);  
 registerAlias("short", Short.class);  
 registerAlias("int", Integer.class);  
 registerAlias("integer", Integer.class);  
 registerAlias("double", Double.class);  
 registerAlias("float", Float.class);  
 registerAlias("boolean", Boolean.class);  
  
 registerAlias("byte[]", Byte[].class);  
 registerAlias("long[]", Long[].class);  
 registerAlias("short[]", Short[].class);  
 registerAlias("int[]", Integer[].class);  
 registerAlias("integer[]", Integer[].class);  
 registerAlias("double[]", Double[].class);  
 registerAlias("float[]", Float[].class);  
 registerAlias("boolean[]", Boolean[].class);  
  
 registerAlias("\_byte", byte.class);  
 registerAlias("\_long", long.class);  
 registerAlias("\_short", short.class);  
 registerAlias("\_int", int.class);  
 registerAlias("\_integer", int.class);  
 registerAlias("\_double", double.class);  
 registerAlias("\_float", float.class);  
 registerAlias("\_boolean", boolean.class);  
  
 registerAlias("\_byte[]", byte[].class);  
 registerAlias("\_long[]", long[].class);  
 registerAlias("\_short[]", short[].class);  
 registerAlias("\_int[]", int[].class);  
 registerAlias("\_integer[]", int[].class);  
 registerAlias("\_double[]", double[].class);  
 registerAlias("\_float[]", float[].class);  
 registerAlias("\_boolean[]", boolean[].class);  
  
 registerAlias("date", Date.class);  
 registerAlias("decimal", BigDecimal.class);  
 registerAlias("bigdecimal", BigDecimal.class);  
 registerAlias("biginteger", BigInteger.class);  
 registerAlias("object", Object.class);  
  
 registerAlias("date[]", Date[].class);  
 registerAlias("decimal[]", BigDecimal[].class);  
 registerAlias("bigdecimal[]", BigDecimal[].class);  
 registerAlias("biginteger[]", BigInteger[].class);  
 registerAlias("object[]", Object[].class);  
  
 registerAlias("map", Map.class);  
 registerAlias("hashmap", HashMap.class);  
 registerAlias("list", List.class);  
 registerAlias("arraylist", ArrayList.class);  
 registerAlias("collection", Collection.class);  
 registerAlias("iterator", Iterator.class);  
  
 registerAlias("ResultSet", ResultSet.class);  
 }  
}

mybatis启动的时候会加载全局配置文件，会将其转换为一个org.apache.ibatis.session.Configuration对象，存储在内存中，Configuration类中也注册了一些别名，代码如下：

typeAliasRegistry.registerAlias("JDBC", JdbcTransactionFactory.class);  
typeAliasRegistry.registerAlias("MANAGED", ManagedTransactionFactory.class);  
  
typeAliasRegistry.registerAlias("JNDI", JndiDataSourceFactory.class);  
typeAliasRegistry.registerAlias("POOLED", PooledDataSourceFactory.class);  
typeAliasRegistry.registerAlias("UNPOOLED", UnpooledDataSourceFactory.class);  
  
typeAliasRegistry.registerAlias("PERPETUAL", PerpetualCache.class);  
typeAliasRegistry.registerAlias("FIFO", FifoCache.class);  
typeAliasRegistry.registerAlias("LRU", LruCache.class);  
typeAliasRegistry.registerAlias("SOFT", SoftCache.class);  
typeAliasRegistry.registerAlias("WEAK", WeakCache.class);  
  
typeAliasRegistry.registerAlias("DB\_VENDOR", VendorDatabaseIdProvider.class);  
  
typeAliasRegistry.registerAlias("XML", XMLLanguageDriver.class);  
typeAliasRegistry.registerAlias("RAW", RawLanguageDriver.class);  
  
typeAliasRegistry.registerAlias("SLF4J", Slf4jImpl.class);  
typeAliasRegistry.registerAlias("COMMONS\_LOGGING", JakartaCommonsLoggingImpl.class);  
typeAliasRegistry.registerAlias("LOG4J", Log4jImpl.class);  
typeAliasRegistry.registerAlias("LOG4J2", Log4j2Impl.class);  
typeAliasRegistry.registerAlias("JDK\_LOGGING", Jdk14LoggingImpl.class);  
typeAliasRegistry.registerAlias("STDOUT\_LOGGING", StdOutImpl.class);  
typeAliasRegistry.registerAlias("NO\_LOGGING", NoLoggingImpl.class);  
  
typeAliasRegistry.registerAlias("CGLIB", CglibProxyFactory.class);  
typeAliasRegistry.registerAlias("JAVASSIST", JavassistProxyFactory.class);

上面有2行如下：

typeAliasRegistry.registerAlias("JDBC", JdbcTransactionFactory.class);  
typeAliasRegistry.registerAlias("POOLED", PooledDataSourceFactory.class);

上面这2行，注册了2个别名，别名和类型映射关系如下：

JDBC -> JdbcTransactionFactory  
POOLED -> PooledDataSourceFactory

上面这2个对象，大家应该比较熟悉吧，mybatis全局配置文件（chat03\src\main\resources\demo1\mybatis-config.xml）中我们用到过，我们再去看一下，如下：



上面2个红框的是不是就是上面注册的2个类型，上面xml中我们写的是完整类型名称，我们可以将其改为别名的方式也是可以的，如下：



我们来运行com.javacode2018.chat03.demo1.UserMapperTest#getUserList，看一下能否正常运行，输出如下：

44:10.886 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - ==> Preparing: SELECT \* FROM t\_user   
44:10.929 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - ==> Parameters:   
44:10.947 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - <== Total: 4  
44:10.948 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)  
44:10.950 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=2, name=javacode2018, age=30, salary=50000.0, sex=1)  
44:10.950 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=3, name=张学友, age=56, salary=500000.0, sex=1)  
44:10.950 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=4, name=林志玲, age=45, salary=88888.88, sex=2)

很好，一切正常的。

#### 别名的原理

mybatis允许我们给某种类型注册一个别名，别名和类型之间会建立映射关系，这个映射关系存储在一个map对象中，key为别名的名称，value为具体的类型，当我们通过一个名称访问某种类型的时候，mybatis根据类型的名称，先在别名和类型映射的map中按照key进行查找，如果找到了直接返回对应的类型，如果没找到，会将这个名称当做完整的类名去解析成Class对象，如果这2步解析都无法识别这种类型，就会报错。

**mybatis和别名相关的操作都位于org.apache.ibatis.type.TypeAliasRegistry类中，包含别名的注册、解析等各种操作。**

我们来看一下别名解析的方法，如下：

public <T> Class<T> resolveAlias(String string) {  
 try {  
 if (string == null) {  
 return null;  
 }  
 // issue #748  
 String key = string.toLowerCase(Locale.ENGLISH);  
 Class<T> value;  
 if (typeAliases.containsKey(key)) {  
 value = (Class<T>) typeAliases.get(key);  
 } else {  
 value = (Class<T>) Resources.classForName(string);  
 }  
 return value;  
 } catch (ClassNotFoundException e) {  
 throw new TypeException("Could not resolve type alias '" + string + "'. Cause: " + e, e);  
 }  
 }

有一个typeAliases对象，我们看一下其定义：

private final Map<String, Class<?>> typeAliases = new HashMap<>();

这个对象就是存放别名和具体类型映射关系的，从上面代码中可以看出，通过传入的参数解析对应的类型的时候，会先从typeAliases中查找，如果找不到会调用下面代码：

value = (Class<T>) Resources.classForName(string);

上面这个方法里面具体是使用下面代码去通过名称解析成类型的：

Class.forName(类名完整名称)

Class.forName大家应该是很熟悉的，可以获取一个字符串对应的Class对象，如果找不到这个对象，会报错。

#### 别名使用建议

别名的方式可以简化类型的写法，原本很长一串的UserModel对象，现在只用写个user就行了，用起来是不是挺爽的？

从写法上面来说，确实少帮我们省了一些代码，但是从维护上面来讲，不是很方便。

如Mapper xml直接写别名，看代码的时候，很难知道这个别名对应的具体类型，还需要我们去注册的地方找一下，不是太方便，如果我们在idea中写完整的类名，还可以按住Ctrl健，然后用鼠标左键点击类型直接可以跳到对应的类定义中去，如果使用别名是无法导航过去的。

整体上来说开发和看代码都不是太方便，只是写法上比价简单。

**所以建议自定义的类尽量别使用别名，而对mybatis中内置的一些别名我们需要知道。**

### 属性配置文件详解

大家看一下chat03\src\main\resources\demo1\mybatis-config.xml中下面这一部分的配置：

<dataSource type="POOLED">  
 <property name="driver" value="com.mysql.jdbc.Driver"/>  
 <property name="url" value="jdbc:mysql://localhost:3306/javacode2018?characterEncoding=UTF-8"/>  
 <property name="username" value="root"/>  
 <property name="password" value="root123"/>  
</dataSource>

这个连接数据库的配置，我们是直接写在mybatis全局配置文件中的，上面这是我们本地测试库的db信息，上线之后，需要修改为线上的db配置信息，db配置信息一般由运维去修改，让运维去修改这个xml配置文件？

这样不是太好，我们通常将一些需要运维修改的配置信息（如：db配置、邮件配置、redis配置等等各种配置）放在一个properties配文件中，然后上线时，只需要运维去修改这个配置文件就可以了，根本不用他们去修改和代码相关的文件。

mybatis也支持我们通过外部properties文件来配置一些属性信息。

mybatis配置属性信息有3种方式。

#### 方式1：property元素中定义属性

##### 属性定义

mybatis全局配置文件中通过properties元素来定义属性信息，如下：

<configuration>  
 <properties>  
 <property name="属性名称" value="属性对应的值"/>  
 </properties>  
</configuration>

上面通过property元素的方式进行配置属性信息：

name：属性的名称

value：属性的值。

如：

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

##### 使用${属性名称}引用属性的值

属性已经定义好了，我们可以通过${属性名称}引用定义好的属性的值，如：

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

##### 案例

我们在demo1/mapper/mybatis-config.xml的configuration元素中加入下面配置：

<properties>  
 <property name="jdbc.driver" value="com.mysql.jdbc.Driver"/>  
 <property name="jdbc.url" value="jdbc:mysql://localhost:3306/javacode2018?characterEncoding=UTF-8"/>  
 <property name="jdbc.username" value="root"/>  
 <property name="jdbc.password" value="root123"/>  
</properties>

修改datasource的配置：

<dataSource type="POOLED">  
 <property name="driver" value="${jdbc.driver}"/>  
 <property name="url" value="${jdbc.url}"/>  
 <property name="username" value="${jdbc.username}"/>  
 <property name="password" value="${jdbc.password}"/>  
</dataSource>

运行com.javacode2018.chat03.demo1.UserMapperTest#getUserList，如下：

40:22.274 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - ==> Preparing: SELECT \* FROM t\_user   
40:22.307 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - ==> Parameters:   
40:22.330 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - <== Total: 4  
40:22.331 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)  
40:22.332 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=2, name=javacode2018, age=30, salary=50000.0, sex=1)  
40:22.332 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=3, name=张学友, age=56, salary=500000.0, sex=1)  
40:22.332 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=4, name=林志玲, age=45, salary=88888.88, sex=2)

运行正常。

#### 方式2：resource引入配置文件

方式1中，我们的配置文件还是写在全局配置文件中，mybatis支持从外部引入配置文件，可以把配置文件写在其他外部文件中，然后进行引入。

##### 引入classes路径中的配置文件

<configuration>  
 <properties resource="配置文件路径"/>  
</configuration>

properties元素有个resource属性，值为配置文件相对于classes的路径，配置文件我们一般放在src/main/resource目录，这个目录的文件编译之后会放在classes路径中。

##### 案例

下面我们将上面db的配置放在外部的config.properties文件中。

在chat03\src\main\resources\demo1目录新建一个配置文件config.properties，内容如下：

jdbc.driver=com.mysql.jdbc.Driver  
jdbc.url=jdbc:mysql://localhost:3306/javacode2018?characterEncoding=UTF-8  
jdbc.username=root  
jdbc.password=root123

demo1/mapper/mybatis-config.xml中引入上面配置文件：

<!-- 引入外部配置文件 -->  
<properties resource="demo1/mapper/config.properties"/>

目前demo1/mapper/mybatis-config.xml文件内容如下：

<?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>  
 <!-- 引入外部配置文件 -->  
 <properties resource="demo1/config.properties"/>  
 <typeAliases>  
 <package name="com.javacode2018.chat03.demo1"/>  
 </typeAliases>  
 <!-- 环境配置，可以配置多个环境 -->  
 <environments default="chat04-demo1">  
 <!--   
 environment用来对某个环境进行配置  
 id：环境标识，唯一  
 -->  
 <environment id="chat04-demo1">  
 <!-- 事务管理器工厂配置 -->  
 <transactionManager type="JDBC"/>  
 <!-- 数据源工厂配置，使用工厂来创建数据源 -->  
 <dataSource type="POOLED">  
 <property name="driver" value="${jdbc.driver}"/>  
 <property name="url" value="${jdbc.url}"/>  
 <property name="username" value="${jdbc.username}"/>  
 <property name="password" value="${jdbc.password}"/>  
 </dataSource>  
 </environment>  
 </environments>  
  
 <mappers>  
 <mapper resource="demo1/mapper/UserMapper.xml"/>  
 </mappers>  
</configuration>

运行com.javacode2018.chat03.demo1.UserMapperTest#getUserList，如下：

57:40.405 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - ==> Preparing: SELECT \* FROM t\_user   
57:40.436 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - ==> Parameters:   
57:40.454 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - <== Total: 4  
57:40.455 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)  
57:40.457 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=2, name=javacode2018, age=30, salary=50000.0, sex=1)  
57:40.457 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=3, name=张学友, age=56, salary=500000.0, sex=1)  
57:40.457 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=4, name=林志玲, age=45, salary=88888.88, sex=2)

运行正常。

#### 方式3：url的方式引入远程配置文件

mybatis还提供了引入远程配置文件的方式，如下：

<properties url="远程配置文件的路径" />

这次还是使用properties元素，不过使用的是url属性，如：

<properties url="http://mashibing.com/properties/config.properties" />

这种方式的案例就不提供了，有兴趣的可以自己去玩玩。

#### 属性配置文件使用建议

上面我们说了3中方式，第2中方式是比较常见的做法，建议大家可以使用第二种方式来引入外部资源配置文件。

#### 问题

**如果3种方式如果我们都写了，mybatis会怎么走？**

下面我们修改一下resources/demo1/mybatis-config.xml，使用第一种方式定义属性，如下：

<properties>  
 <property name="jdbc.driver" value="com.mysql.jdbc.Driver"/>  
 <property name="jdbc.url" value="jdbc:mysql://localhost:3306/javacode2018?characterEncoding=UTF-8"/>  
 <property name="jdbc.username" value="root"/>  
 <property name="jdbc.password" value="root"/>  
</properties>

将password的值改为了root，正确的是root123，运行测试用例，报错如下：

org.apache.ibatis.exceptions.PersistenceException:   
### Error querying database. Cause: java.sql.SQLException: Access denied for user 'root'@'localhost' (using password: YES)  
### The error may exist in demo1/mapper/UserMapper.xml  
### The error may involve com.javacode2018.chat03.demo1.UserMapper.getUserList  
### The error occurred while executing a query  
### Cause: java.sql.SQLException: Access denied for user 'root'@'localhost' (using password: YES)  
  
 at org.apache.ibatis.exceptions.ExceptionFactory.wrapException(ExceptionFactory.java:30)  
 at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:149)  
 at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:140)  
 at org.apache.ibatis.binding.MapperMethod.executeForMany(MapperMethod.java:147)  
 at org.apache.ibatis.binding.MapperMethod.execute(MapperMethod.java:80)  
 at org.apache.ibatis.binding.MapperProxy.invoke(MapperProxy.java:93)  
 at com.sun.proxy.$Proxy6.getUserList(Unknown Source)  
 at com.javacode2018.chat03.demo1.UserMapperTest.getUserList(UserMapperTest.java:38)  
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)  
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)  
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)  
 at java.lang.reflect.Method.invoke(Method.java:498)  
 at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:50)  
 at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)  
 at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:47)  
 at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)  
 at org.junit.internal.runners.statements.RunBefores.evaluate(RunBefores.java:26)  
 at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:325)  
 at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:78)  
 at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:57)  
 at org.junit.runners.ParentRunner$3.run(ParentRunner.java:290)  
 at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:71)  
 at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:288)  
 at org.junit.runners.ParentRunner.access$000(ParentRunner.java:58)  
 at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:268)  
 at org.junit.runners.ParentRunner.run(ParentRunner.java:363)  
 at org.junit.runner.JUnitCore.run(JUnitCore.java:137)  
 at com.intellij.junit4.JUnit4IdeaTestRunner.startRunnerWithArgs(JUnit4IdeaTestRunner.java:68)  
 at com.intellij.rt.execution.junit.IdeaTestRunner$Repeater.startRunnerWithArgs(IdeaTestRunner.java:51)  
 at com.intellij.rt.execution.junit.JUnitStarter.prepareStreamsAndStart(JUnitStarter.java:242)  
 at com.intellij.rt.execution.junit.JUnitStarter.main(JUnitStarter.java:70)  
Caused by: java.sql.SQLException: Access denied for user 'root'@'localhost' (using password: YES)

提示密码错误。

下面我们将第2种方式也加入，修改配置：

<properties resource="demo1/config.properties">  
 <property name="jdbc.driver" value="com.mysql.jdbc.Driver"/>  
 <property name="jdbc.url" value="jdbc:mysql://localhost:3306/javacode2018?characterEncoding=UTF-8"/>  
 <property name="jdbc.username" value="root"/>  
 <property name="jdbc.password" value="root"/>  
</properties>

再运行一下测试用例，如下：

18:59.436 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - ==> Preparing: SELECT \* FROM t\_user   
18:59.462 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - ==> Parameters:   
18:59.481 [main] DEBUG c.j.c.demo1.UserMapper.getUserList - <== Total: 4  
18:59.482 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)  
18:59.485 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=2, name=javacode2018, age=30, salary=50000.0, sex=1)  
18:59.485 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=3, name=张学友, age=56, salary=500000.0, sex=1)  
18:59.485 [main] INFO c.j.chat03.demo1.UserMapperTest - UserModel(id=4, name=林志玲, age=45, salary=88888.88, sex=2)

这次正常了。

**可以看出方式1和方式2都存在的时候，方式2的配置会覆盖方式1的配置。**

mybatis这块的源码在org.apache.ibatis.builder.xml.XMLConfigBuilder#propertiesElement方法中，如下：

private void propertiesElement(XNode context) throws Exception {  
 if (context != null) {  
 Properties defaults = context.getChildrenAsProperties();  
 String resource = context.getStringAttribute("resource");  
 String url = context.getStringAttribute("url");  
 if (resource != null && url != null) {  
 throw new BuilderException("The properties element cannot specify both a URL and a resource based property file reference. Please specify one or the other.");  
 }  
 if (resource != null) {  
 defaults.putAll(Resources.getResourceAsProperties(resource));  
 } else if (url != null) {  
 defaults.putAll(Resources.getUrlAsProperties(url));  
 }  
 Properties vars = configuration.getVariables();  
 if (vars != null) {  
 defaults.putAll(vars);  
 }  
 parser.setVariables(defaults);  
 configuration.setVariables(defaults);  
 }  
 }

**从上面代码中也可以看出，如果方式2和方式3都存在的时候，方式3会失效，mybatis会先读取方式1的配置，然后读取方式2或者方式3的配置，会将1中相同的配置给覆盖。**

### mybatis中引入mapper的3种方式

mapper xml文件是非常重要的，我们写的sql基本上都在里面，使用mybatis开发项目的时候，和mybatis相关的大部分代码就是写sql，基本上都是和mapper xml打交道。

编写好的mapper xml需要让mybatis知道，我们怎么让mybatis知道呢？

可以通过mybatis全局配置文件进行引入，主要有3种方式。

#### 方式1：使用mapper resouce属性注册mapper xml文件

目前我们所涉及到的各种例子都是采用的这种方式，使用下面的方法进行引入：

<mappers>  
 <mapper resource="Mapper xml的路径（相对于classes的路径）"/>  
</mappers>

再来说一下这种方式的一些注意点：

1. 一般情况下面我，我们会创建一个和Mapper xml中namespace同名的Mapper接口，Mapper接口会和Mapper xml文件进行绑定
2. mybatis加载mapper xml的时候，会去查找namespace对应的Mapper接口，然后进行注册，我们可以通过Mapper接口的方式去访问Mapper xml中的具体操作
3. Mapper xml和Mapper 接口配合的方式是比较常见的做法，也是强烈建议大家使用的

#### 方式2：使用mapper class属性注册Mapper接口

##### 引入Mapper接口

mybatis全局配置文件中引入mapper接口，如下：

<mappers>  
 <mapper class="接口的完整类名" />  
</mappers>

**这种情况下，mybais会去加载class对应的接口，然后还会去加载和这个接口同一个目录的同名的xml文件。**

如：

<mappers>  
 <mapper class="com.javacode2018.chat03.demo1.UserMapper" />  
</mappers>

上面这种写法，mybatis会自动去注册UserMapper接口，还会去查找下面的文件：

com/javacode2018/chat03/demo1/UserMapper.xml

大家以后开发项目的时候估计也会看到这种写法，Mapper接口和Mapper xml文件放在同一个包中。

##### 案例

下面我们重新创建一个案例，都放在demo2包中。

新建com.javacode2018.chat03.demo2.UserModel，如下：

package com.javacode2018.chat03.demo2;  
  
import lombok.\*;  
import org.apache.ibatis.type.Alias;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
@Getter  
@Setter  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
@ToString  
public class UserModel {  
 private Long id;  
 private String name;  
 private Integer age;  
 private Double salary;  
 private Integer sex;  
}

新建com.javacode2018.chat03.demo2.UserMapper，如下：

package com.javacode2018.chat03.demo2;  
  
import java.util.List;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
public interface UserMapper {  
  
 List<UserModel> getUserList();  
}

chat03\src\main\java\com\javacode2018\chat03\demo2中创建UserMapper.xml，如下：

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper namespace="com.javacode2018.chat03.demo2.UserMapper">  
 <!-- select用来定义一个查询操作  
 id：操作的具体标识  
 resultType：指定查询结果保存的类型  
 -->  
 <select id="getUserList" resultType="com.javacode2018.chat03.demo2.UserModel">  
 <![CDATA[  
 SELECT \* FROM t\_user  
 ]]>  
 </select>  
  
</mapper>

下面重点来了。

创建mybatis全局配置文件，在chat03\src\main\resources\demo2目录中创建mybatis-config.xml，如下：

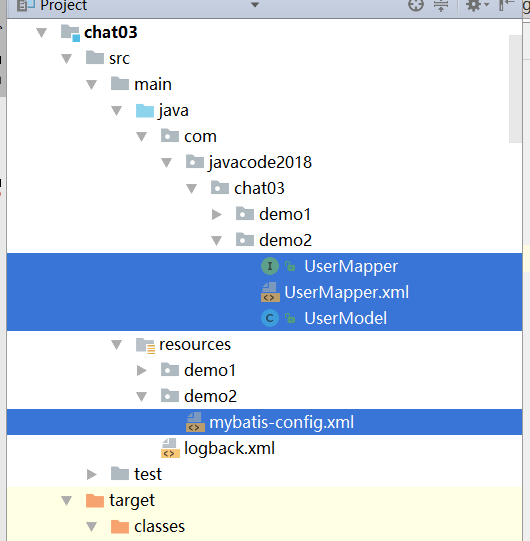
<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE configuration  
 PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-config.dtd">  
<configuration>  
 <properties>  
 <property name="jdbc.driver" value="com.mysql.jdbc.Driver"/>  
 <property name="jdbc.url" value="jdbc:mysql://localhost:3306/javacode2018?characterEncoding=UTF-8"/>  
 <property name="jdbc.username" value="root"/>  
 <property name="jdbc.password" value="root123"/>  
 </properties>  
 <!-- 环境配置，可以配置多个环境 -->  
 <environments default="chat04-demo2">  
 <!--   
 environment用来对某个环境进行配置  
 id：环境标识，唯一  
 -->  
 <environment id="chat04-demo2">  
 <!-- 事务管理器工厂配置 -->  
 <transactionManager type="JDBC"/>  
 <!-- 数据源工厂配置，使用工厂来创建数据源 -->  
 <dataSource type="POOLED">  
 <property name="driver" value="${jdbc.driver}"/>  
 <property name="url" value="${jdbc.url}"/>  
 <property name="username" value="${jdbc.username}"/>  
 <property name="password" value="${jdbc.password}"/>  
 </dataSource>  
 </environment>  
 </environments>  
   
</configuration>

chat03\src\test\java目录创建测试用例com.javacode2018.chat03.demo2.UserMapperTest，如下：

package com.javacode2018.chat03.demo2;  
  
import lombok.extern.slf4j.Slf4j;  
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 java.io.IOException;  
import java.io.InputStream;  
import java.util.List;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
@Slf4j  
public class UserMapperTest {  
 private SqlSessionFactory sqlSessionFactory;  
  
 @Before  
 public void before() throws IOException {  
 //指定mybatis全局配置文件  
 String resource = "demo2/mybatis-config.xml";  
 //读取全局配置文件  
 InputStream inputStream = Resources.getResourceAsStream(resource);  
 //构建SqlSessionFactory对象  
 SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
 this.sqlSessionFactory = sqlSessionFactory;  
 }  
  
 @Test  
 public void getUserList() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 //执行查询操作  
 List<UserModel> userModelList = mapper.getUserList();  
 userModelList.forEach(item -> {  
 log.info("{}", item);  
 });  
 }  
 }  
  
}

注意这次上面使用的是demo2/mybatis-config.xml配置文件。

我们先来看一下项目结构，4个文件：



注意一下UserMapper接口所在的包中有个同名的UserMapper.xml文件，这个如果按照方式2中所说的，会自动加载。

下面我们来运行一下com.javacode2018.chat03.demo2.UserMapperTest#getUserList，输出：

org.apache.ibatis.binding.BindingException: Type interface com.javacode2018.chat03.demo2.UserMapper is not known to the MapperRegistry.  
  
 at org.apache.ibatis.binding.MapperRegistry.getMapper(MapperRegistry.java:47)  
 at org.apache.ibatis.session.Configuration.getMapper(Configuration.java:779)  
 at org.apache.ibatis.session.defaults.DefaultSqlSession.getMapper(DefaultSqlSession.java:291)  
 at com.javacode2018.chat03.demo2.UserMapperTest.getUserList(UserMapperTest.java:36)  
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)  
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)  
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)  
 at java.lang.reflect.Method.invoke(Method.java:498)  
 at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:50)  
 at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)  
 at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:47)  
 at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)  
 at org.junit.internal.runners.statements.RunBefores.evaluate(RunBefores.java:26)  
 at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:325)  
 at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:78)  
 at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:57)  
 at org.junit.runners.ParentRunner$3.run(ParentRunner.java:290)  
 at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:71)  
 at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:288)  
 at org.junit.runners.ParentRunner.access$000(ParentRunner.java:58)  
 at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:268)  
 at org.junit.runners.ParentRunner.run(ParentRunner.java:363)  
 at org.junit.runner.JUnitCore.run(JUnitCore.java:137)  
 at com.intellij.junit4.JUnit4IdeaTestRunner.startRunnerWithArgs(JUnit4IdeaTestRunner.java:68)  
 at com.intellij.rt.execution.junit.IdeaTestRunner$Repeater.startRunnerWithArgs(IdeaTestRunner.java:51)  
 at com.intellij.rt.execution.junit.JUnitStarter.prepareStreamsAndStart(JUnitStarter.java:242)  
 at com.intellij.rt.execution.junit.JUnitStarter.main(JUnitStarter.java:70)

从输出中可以看到，UserMapper找不到。

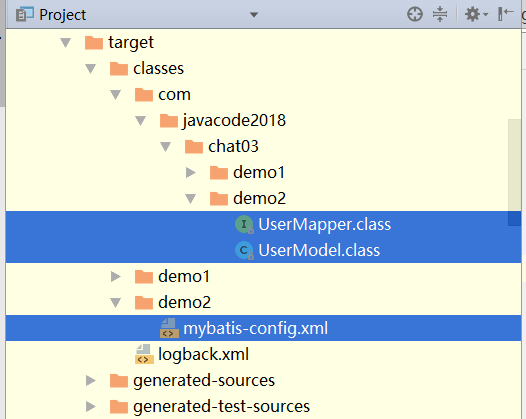
我们去看一下demo2/mybatis-config.xml这个配置文件，这个文件中需要使用方式2引入UserMapper接口，在demo2/mybatis-config.xml中加入下面配置：

<mappers>  
 <mapper class="com.javacode2018.chat03.demo2.UserMapper" />  
</mappers>

再运行一下，还是报错，如下，还是找不到对应的UserMapper：

org.apache.ibatis.binding.BindingException: Invalid bound statement (not found): com.javacode2018.chat03.demo2.UserMapper.getUserList  
  
 at org.apache.ibatis.binding.MapperMethod$SqlCommand.<init>(MapperMethod.java:235)  
 at org.apache.ibatis.binding.MapperMethod.<init>(MapperMethod.java:53)

还是有问题，我们看一下target/classes中demo2包的内容，如下图：



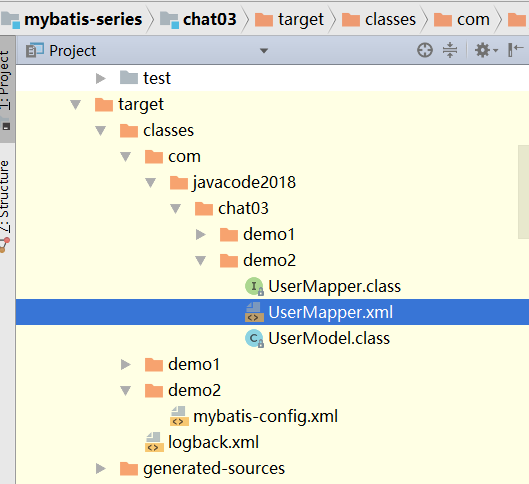
编译之后的文件中少了UserMapper.xml，这个和maven有关，maven编译src/java代码的时候，默认只会对java文件进行编译然后放在target/classes目录，需要在chat03/pom.xml中加入下面配置：

<build>  
 <resources>  
 <resource>  
 <directory>${project.basedir}/src/main/java</directory>  
 <includes>  
 <include>\*\*/\*.xml</include>  
 </includes>  
 </resource>  
 <resource>  
 <directory>${project.basedir}/src/main/resources</directory>  
 <includes>  
 <include>\*\*/\*</include>  
 </includes>  
 </resource>  
 </resources>  
</build>

最终chat03/pom.xml内容如下：

<?xml version="1.0" encoding="UTF-8"?>  
<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">  
 <parent>  
 <artifactId>mybatis-series</artifactId>  
 <groupId>com.javacode2018</groupId>  
 <version>1.0-SNAPSHOT</version>  
 </parent>  
 <modelVersion>4.0.0</modelVersion>  
  
 <artifactId>chat03</artifactId>  
  
 <dependencies>  
 <!-- mybatis依赖 -->  
 <dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis</artifactId>  
 </dependency>  
 <!-- mysql 驱动 -->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 </dependency>  
 <!-- lombok支持 -->  
 <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 </dependency>  
 <!-- 单元测试junit支持 -->  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 </dependency>  
 <!-- 引入logback用来输出日志 -->  
 <dependency>  
 <groupId>ch.qos.logback</groupId>  
 <artifactId>logback-classic</artifactId>  
 </dependency>  
 </dependencies>  
  
 <build>  
 <resources>  
 <resource>  
 <directory>${project.basedir}/src/main/java</directory>  
 <includes>  
 <include>\*\*/\*.xml</include>  
 </includes>  
 </resource>  
 <resource>  
 <directory>${project.basedir}/src/main/resources</directory>  
 <includes>  
 <include>\*\*/\*</include>  
 </includes>  
 </resource>  
 </resources>  
 </build>  
  
</project>

加了这个之后UserMapper.xml就会被放到target的classes中去了，如下图：



为什么maven中需要加上面配置，这块大家可以去看公众号中maven系列的文章，里面有详细介绍，maven的相关东西，后面还会经常用到，对这块不熟悉的，建议尽快把maven系列的所有文章都看一遍，以免后面学习的过程中掉队。

我们再次运行一下测试用例com.javacode2018.chat03.demo2.UserMapperTest#getUserList，效果如下：

24:37.814 [main] DEBUG c.j.c.demo2.UserMapper.getUserList - ==> Preparing: SELECT \* FROM t\_user   
24:37.852 [main] DEBUG c.j.c.demo2.UserMapper.getUserList - ==> Parameters:   
24:37.875 [main] DEBUG c.j.c.demo2.UserMapper.getUserList - <== Total: 4  
24:37.876 [main] INFO c.j.chat03.demo2.UserMapperTest - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)  
24:37.879 [main] INFO c.j.chat03.demo2.UserMapperTest - UserModel(id=2, name=javacode2018, age=30, salary=50000.0, sex=1)  
24:37.879 [main] INFO c.j.chat03.demo2.UserMapperTest - UserModel(id=3, name=张学友, age=56, salary=500000.0, sex=1)  
24:37.879 [main] INFO c.j.chat03.demo2.UserMapperTest - UserModel(id=4, name=林志玲, age=45, salary=88888.88, sex=2)

这次正常了。

##### 源码

方式2对应的源码大家可以去看下面这个方法：

org.apache.ibatis.builder.xml.XMLConfigBuilder#mapperElement

方法中会去加载mapper元素中class属性指定的Mapper接口，然后进行注册，随后会在接口同目录中查找同名的mapper xml文件，将解析这个xml文件，如果mapper xml文件不存在，也不会报错，源码还是比较简单的，大家可以去看一下，加深理解。

#### 方式3：使用package元素批量注册Mapper接口

##### 批量注册Mapper接口

上面说2种方式都是一个个注册mapper的，如果我们写了很多mapper，是否能够批量注册呢？

mybatis提供了扫描包批量注册的方式，需要在mybatis全局配置文件中加入下面配置：

<mappers>  
 <package name="需要扫描的包" />  
</mappers>

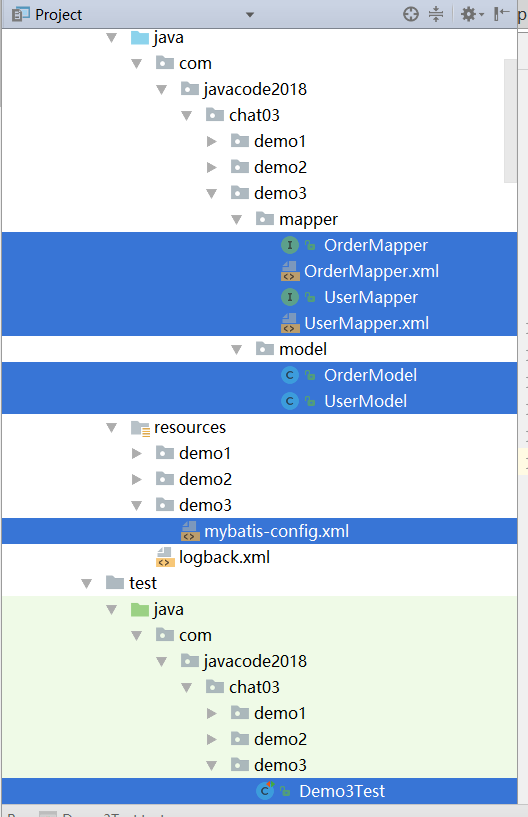
mybatis会扫描package元素中name属性指定的包及子包中的所有接口，将其当做Mapper 接口进行注册，所以一般我们会创建一个mapper包，里面放Mapper接口和同名的Mapper xml文件。

大家来看一个案例，理解一下。

##### 案例

这个案例中将对t\_user、t\_order两个表进行查询操作，采用方式3中的package批量引入mapper 接口和xml文件。

所有代码放在demo3包中，大家先看下文件所在的目录：



创建UserModel类，如下：

package com.javacode2018.chat03.demo3.model;  
  
import lombok.\*;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
@Getter  
@Setter  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
@ToString  
public class UserModel {  
 private Long id;  
 private String name;  
 private Integer age;  
 private Double salary;  
 private Integer sex;  
}

创建OrderModel类，如下：

package com.javacode2018.chat03.demo3.model;  
  
import lombok.\*;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
@Getter  
@Setter  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
@ToString  
public class OrderModel {  
 private Long id;  
 private Long user\_id;  
 private Double price;  
}

创建UserMapper接口，如下：

package com.javacode2018.chat03.demo3.mapper;  
  
import com.javacode2018.chat03.demo3.model.UserModel;  
  
import java.util.List;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
public interface UserMapper {  
  
 List<UserModel> getList();  
}

创建OrderMapper接口，如下：

package com.javacode2018.chat03.demo3.mapper;  
  
import com.javacode2018.chat03.demo3.model.OrderModel;  
import com.javacode2018.chat03.demo3.model.UserModel;  
  
import java.util.List;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
public interface OrderMapper {  
  
 List<OrderModel> getList();  
}

创建UserMapper.xml，如下：

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper namespace="com.javacode2018.chat03.demo3.mapper.OrderMapper">  
 <!-- select用来定义一个查询操作  
 id：操作的具体标识  
 resultType：指定查询结果保存的类型  
 -->  
 <select id="getList" resultType="com.javacode2018.chat03.demo3.model.OrderModel">  
 <![CDATA[  
 SELECT \* FROM t\_order  
 ]]>  
 </select>  
  
</mapper>

上面我们写了一个查询t\_user数据的sql

创建OrderMapper.xml，如下：

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper namespace="com.javacode2018.chat03.demo3.mapper.OrderMapper">  
 <!-- select用来定义一个查询操作  
 id：操作的具体标识  
 resultType：指定查询结果保存的类型  
 -->  
 <select id="getList" resultType="com.javacode2018.chat03.demo3.model.OrderModel">  
 <![CDATA[  
 SELECT \* FROM t\_order  
 ]]>  
 </select>  
  
</mapper>

上面我们写了一个查询t\_order数据的sql

创建resources/demo3/mybatis-config.xml配置文件，如下：

<?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>  
 <properties>  
 <property name="jdbc.driver" value="com.mysql.jdbc.Driver"/>  
 <property name="jdbc.url" value="jdbc:mysql://localhost:3306/javacode2018?characterEncoding=UTF-8"/>  
 <property name="jdbc.username" value="root"/>  
 <property name="jdbc.password" value="root123"/>  
 </properties>  
 <!-- 环境配置，可以配置多个环境 -->  
 <environments default="chat04-demo3">  
 <!--   
 environment用来对某个环境进行配置  
 id：环境标识，唯一  
 -->  
 <environment id="chat04-demo3">  
 <!-- 事务管理器工厂配置 -->  
 <transactionManager type="JDBC"/>  
 <!-- 数据源工厂配置，使用工厂来创建数据源 -->  
 <dataSource type="POOLED">  
 <property name="driver" value="${jdbc.driver}"/>  
 <property name="url" value="${jdbc.url}"/>  
 <property name="username" value="${jdbc.username}"/>  
 <property name="password" value="${jdbc.password}"/>  
 </dataSource>  
 </environment>  
 </environments>  
  
 <mappers>  
 <package name="com.javacode2018.chat03.demo3.mapper"/>  
 </mappers>  
</configuration>

注意这次我们使用package来让mybatis加载com.javacode2018.chat03.demo3.mapper包下面所有的Mapper接口和Mapper xml文件。

创建测试用例Demo3Test，如下：

package com.javacode2018.chat03.demo3;  
  
import com.javacode2018.chat03.demo3.mapper.OrderMapper;  
import com.javacode2018.chat03.demo3.mapper.UserMapper;  
import com.javacode2018.chat03.demo3.model.OrderModel;  
import com.javacode2018.chat03.demo3.model.UserModel;  
import lombok.extern.slf4j.Slf4j;  
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 java.io.IOException;  
import java.io.InputStream;  
import java.util.List;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
@Slf4j  
public class Demo3Test {  
 private SqlSessionFactory sqlSessionFactory;  
  
 @Before  
 public void before() throws IOException {  
 //指定mybatis全局配置文件  
 String resource = "demo3/mybatis-config.xml";  
 //读取全局配置文件  
 InputStream inputStream = Resources.getResourceAsStream(resource);  
 //构建SqlSessionFactory对象  
 SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
 this.sqlSessionFactory = sqlSessionFactory;  
 }  
  
 @Test  
 public void test() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
 //执行查询操作  
 List<UserModel> userModelList = userMapper.getList();  
 userModelList.forEach(item -> {  
 log.info("{}", item);  
 });  
  
 log.info("----------------------------------");  
 OrderMapper orderMapper = sqlSession.getMapper(OrderMapper.class);  
 //执行查询操作  
 List<OrderModel> orderModelList = orderMapper.getList();  
 orderModelList.forEach(item -> {  
 log.info("{}", item);  
 });  
 }  
 }  
  
}

运行com.javacode2018.chat03.demo3.Demo3Test#test，输出如下：

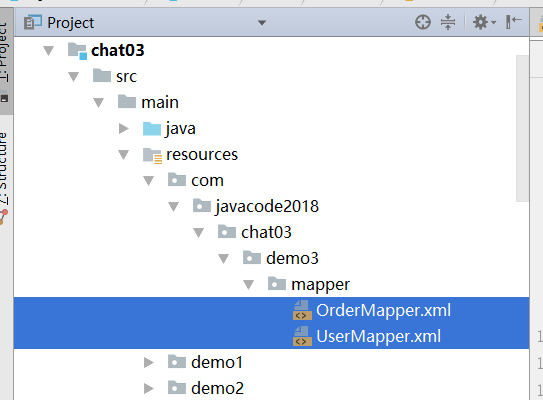
48:39.280 [main] DEBUG c.j.c.d.mapper.UserMapper.getList - ==> Preparing: SELECT \* FROM t\_user   
48:39.315 [main] DEBUG c.j.c.d.mapper.UserMapper.getList - ==> Parameters:   
48:39.339 [main] DEBUG c.j.c.d.mapper.UserMapper.getList - <== Total: 4  
48:39.340 [main] INFO c.j.chat03.demo3.Demo3Test - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)  
48:39.343 [main] INFO c.j.chat03.demo3.Demo3Test - UserModel(id=2, name=javacode2018, age=30, salary=50000.0, sex=1)  
48:39.343 [main] INFO c.j.chat03.demo3.Demo3Test - UserModel(id=3, name=张学友, age=56, salary=500000.0, sex=1)  
48:39.343 [main] INFO c.j.chat03.demo3.Demo3Test - UserModel(id=4, name=林志玲, age=45, salary=88888.88, sex=2)  
48:39.343 [main] INFO c.j.chat03.demo3.Demo3Test - ----------------------------------  
48:39.344 [main] DEBUG c.j.c.d.mapper.OrderMapper.getList - ==> Preparing: SELECT \* FROM t\_order   
48:39.345 [main] DEBUG c.j.c.d.mapper.OrderMapper.getList - ==> Parameters:   
48:39.351 [main] DEBUG c.j.c.d.mapper.OrderMapper.getList - <== Total: 2  
48:39.351 [main] INFO c.j.chat03.demo3.Demo3Test - OrderModel(id=1, user\_id=1, price=88.88)  
48:39.351 [main] INFO c.j.chat03.demo3.Demo3Test - OrderModel(id=2, user\_id=2, price=666.66)

这种批量的方式是不是用着挺爽的，不过有点不是太好，mapper xml和mapper接口放在了一个目录中，目录中既有java代码又有xml文件，看起来也挺别扭的，其实你们可以这样：

一般我们将配置文件放在resource目录，我们可以在resource目录中创建下面子目录：

com/javacode2018/chat03/demo3/mapper

然后将com.javacode2018.chat03.demo3.mapper中的2个xml文件移到上面新创建的目录中去，如下图：



在去运行一下com.javacode2018.chat03.demo3.Demo3Test#test，输出如下：

56:22.669 [main] DEBUG c.j.c.d.mapper.UserMapper.getList - ==> Preparing: SELECT \* FROM t\_user   
56:22.700 [main] DEBUG c.j.c.d.mapper.UserMapper.getList - ==> Parameters:   
56:22.721 [main] DEBUG c.j.c.d.mapper.UserMapper.getList - <== Total: 4  
56:22.722 [main] INFO c.j.chat03.demo3.Demo3Test - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)  
56:22.725 [main] INFO c.j.chat03.demo3.Demo3Test - UserModel(id=2, name=javacode2018, age=30, salary=50000.0, sex=1)  
56:22.725 [main] INFO c.j.chat03.demo3.Demo3Test - UserModel(id=3, name=张学友, age=56, salary=500000.0, sex=1)  
56:22.725 [main] INFO c.j.chat03.demo3.Demo3Test - UserModel(id=4, name=林志玲, age=45, salary=88888.88, sex=2)  
56:22.725 [main] INFO c.j.chat03.demo3.Demo3Test - ----------------------------------  
56:22.727 [main] DEBUG c.j.c.d.mapper.OrderMapper.getList - ==> Preparing: SELECT \* FROM t\_order   
56:22.727 [main] DEBUG c.j.c.d.mapper.OrderMapper.getList - ==> Parameters:   
56:22.732 [main] DEBUG c.j.c.d.mapper.OrderMapper.getList - <== Total: 2  
56:22.732 [main] INFO c.j.chat03.demo3.Demo3Test - OrderModel(id=1, user\_id=1, price=88.88)  
56:22.732 [main] INFO c.j.chat03.demo3.Demo3Test - OrderModel(id=2, user\_id=2, price=666.66)

也是可以的。

#### 源码

方式3的源码和方式2的源码在一个地方：

org.apache.ibatis.builder.xml.XMLConfigBuilder#mapperElement

方法中会去扫描指定的包中所有的接口，会将接口作为Mapper接口进行注册，然后还会找这些接口同名的Xml文件，将其注册为Mapper xml文件，相对于对方式2循环的方式。

#### 使用注意

**方式3会扫描指定包中所有的接口，把这些接口作为Mapper接口进行注册，扫描到的类型只要是接口就会被注册，所以指定的包中通常我们只放Mapper接口，避免存放一些不相干的类或者接口。**

### 关于配置和源码

本次讲解到的一些配置都是在mybatis全局配置文件中进行配置的，这些元素配置是有先后顺序的，具体元素是在下面的dtd文件中定义的：

http://mybatis.org/dtd/mybatis-3-config.dtd

建议大家去看一下这个dtd配置文件。

Mybatis解析这个配置文件的入口是在下面的方法中：

org.apache.ibatis.builder.xml.XMLConfigBuilder#parseConfiguration

代码的部分实现如下：

private void parseConfiguration(XNode root) {  
 try {  
 //issue #117 read properties first  
 propertiesElement(root.evalNode("properties"));  
 Properties settings = settingsAsProperties(root.evalNode("settings"));  
 loadCustomVfs(settings);  
 loadCustomLogImpl(settings);  
 typeAliasesElement(root.evalNode("typeAliases"));  
 pluginElement(root.evalNode("plugins"));  
 objectFactoryElement(root.evalNode("objectFactory"));  
 objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));  
 reflectorFactoryElement(root.evalNode("reflectorFactory"));  
 settingsElement(settings);  
 // read it after objectFactory and objectWrapperFactory issue #631  
 environmentsElement(root.evalNode("environments"));  
 databaseIdProviderElement(root.evalNode("databaseIdProvider"));  
 typeHandlerElement(root.evalNode("typeHandlers"));  
 mapperElement(root.evalNode("mappers"));  
 } catch (Exception e) {  
 throw new BuilderException("Error parsing SQL Mapper Configuration. Cause: " + e, e);  
 }  
 }

可以看到mybatis启动的时候会按顺序加载上面的标签，大家可以去看一下源码，研究一下，下篇继续深入mybatis其他知识点。

### 总结

1. 掌握别名注册的3种方式，建议大家尽量少使用自定义别名
2. 掌握属性配置3种方式
3. 掌握mapper注册的3种方式及需要注意的地方

### MyBatis系列

1. MyBatis系列第1篇：MyBatis未出世之前我们那些痛苦的经历
2. MyBatis系列第2篇：入门篇，带你感受一下mybatis独特的魅力！
3. MyBatis系列第3篇：Mybatis使用详解（1）

## Mybatis系列第5篇：详解传参的几种方式、原理、源码解析

### 主要内容

本篇详解mapper接口传参的各种方式。

* 传递一个参数
* 传递一个Map参数
* 传递一个javabean参数
* 多参数中用@param指定参数名称
* java编译中参数名称的处理
* mapper接口传参源码分析
* 传递1个Collection参数
* 传递1个List参数
* 传递1个数组参数
* mybatis对于集合处理源码分析
* ResultHandler作为参数的用法

**本篇文章的案例在上一篇chat03模块上进行开发，大家可以到文章的尾部获取整个mybatis系列的案例源码。**

**mybatis系列的文章前后都是有依赖的，请大家按顺序去看，尽量不要跳着去看，这样不会出现看不懂的情况，建议大家系统化的学习知识，基础打牢，慢慢才能成为高手。**

使用mybatis开发项目的中，基本上都是使用mapper接口的方式来执行db操作，下面我们来看一下mapper接口传递参数的几种方式及需要注意的地方。

### 传递一个参数

#### 用法

Mapper接口方法中只有一个参数，如：

UserModel getByName(String name);

Mapper xml引用这个name参数：

#{任意合法名称}

如：#{name}、#{val}、${x}等等写法都可以引用上面name参数的值。

#### 案例

创建UserModel类，如下：

package com.javacode2018.chat03.demo4.model;  
  
import lombok.\*;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
@Getter  
@Setter  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
@ToString  
public class UserModel {  
 private Long id;  
 private String name;  
 private Integer age;  
 private Double salary;  
 private Integer sex;  
}

创建Mapper接口UserMapper，如下：

package com.javacode2018.chat03.demo4.mapper;  
  
import com.javacode2018.chat03.demo4.model.UserModel;  
  
import java.util.List;  
import java.util.Map;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
public interface UserMapper {  
 /\*\*  
 \* 通过name查询  
 \*  
 \* @param name  
 \* @return  
 \*/  
 UserModel getByName(String name);  
}

注意上面有个getByName方法，这个方法传递一个参数。

创建Mapper xml文件UserMapper.xml,mybatis-series\chat03\src\main\resources\com\javacode2018\chat03\demo4\mapper目录创建UserMapper.xml，如下：

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper namespace="com.javacode2018.chat03.demo4.mapper.UserMapper">  
  
 <!-- 通过name查询 -->  
 <select id="getByName" resultType="com.javacode2018.chat03.demo4.model.UserModel">  
 <![CDATA[  
 SELECT \* FROM t\_user WHERE name = #{value} LIMIT 1  
 ]]>  
 </select>  
  
</mapper>

上面有个getByName通过用户名查询，通过#{value}引用传递进来的name参数，当一个参数的时候#{变量名称}中变量名称可以随意写，都可以取到传入的参数。

创建属性配置文件，mybatis-series\chat03\src\main\resources目录创建jdbc.properties，如下：

jdbc.driver=com.mysql.jdbc.Driver  
jdbc.url=jdbc:mysql://localhost:3306/javacode2018?characterEncoding=UTF-8  
jdbc.username=root  
jdbc.password=root123

上面是我本地db配置，大家可以根据自己db信息做对应修改。

创建mybatis全局配置文件，mybatis-series\chat03\src\main\resources\demo4目录创建mybatis-config.xml，如下：

<?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jdbc配置 -->  
 <properties resource="jdbc.properties"/>  
 <!-- 环境配置，可以配置多个环境 -->  
 <environments default="demo4">  
 <environment id="demo4">  
 <!-- 事务管理器工厂配置 -->  
 <transactionManager type="JDBC"/>  
 <!-- 数据源工厂配置，使用工厂来创建数据源 -->  
 <dataSource type="POOLED">  
 <property name="driver" value="${jdbc.driver}"/>  
 <property name="url" value="${jdbc.url}"/>  
 <property name="username" value="${jdbc.username}"/>  
 <property name="password" value="${jdbc.password}"/>  
 </dataSource>  
 </environment>  
 </environments>  
  
 <mappers>  
 <package name="com.javacode2018.chat03.demo4.mapper"/>  
 </mappers>  
</configuration>

上面通过properties的resource属性引入了jdbc配置文件。

package属性的name指定了mapper接口和mapper xml文件所在的包，mybatis会扫描这个包，自动注册mapper接口和mapper xml文件。

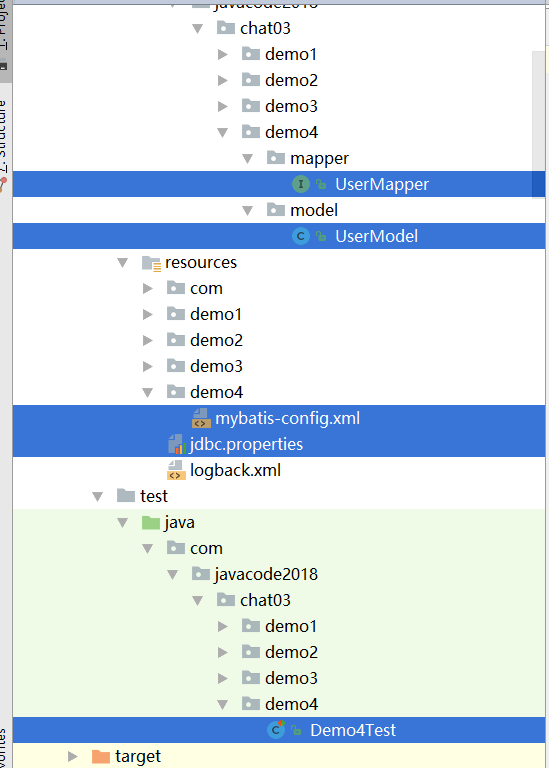
创建测试用例Demo4Test，如下：

package com.javacode2018.chat03.demo4;  
  
import com.javacode2018.chat03.demo4.mapper.UserMapper;  
import com.javacode2018.chat03.demo4.model.UserModel;  
import lombok.extern.slf4j.Slf4j;  
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 java.io.IOException;  
import java.io.InputStream;  
import java.util.HashMap;  
import java.util.List;  
import java.util.Map;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
@Slf4j  
public class Demo4Test {  
 private SqlSessionFactory sqlSessionFactory;  
  
 @Before  
 public void before() throws IOException {  
 //指定mybatis全局配置文件  
 String resource = "demo4/mybatis-config.xml";  
 //读取全局配置文件  
 InputStream inputStream = Resources.getResourceAsStream(resource);  
 //构建SqlSessionFactory对象  
 SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
 this.sqlSessionFactory = sqlSessionFactory;  
 }  
  
 /\*\*  
 \* 通过map给Mapper接口的方法传递参数  
 \*/  
 @Test  
 public void getByName() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
 UserModel userModel = userMapper.getByName("马士兵老师");  
 log.info("{}", userModel);  
 }  
 }  
  
}

注意上面的getByName方法，会调用UserMapper接口的getByName方法通过用户名查询用户信息，我们运行一下这个方法，输出如下：

44:55.747 [main] DEBUG c.j.c.d.mapper.UserMapper.getByName - ==> Preparing: SELECT \* FROM t\_user WHERE name = ? LIMIT 1   
44:55.779 [main] DEBUG c.j.c.d.mapper.UserMapper.getByName - ==> Parameters: 马士兵老师(String)  
44:55.797 [main] DEBUG c.j.c.d.mapper.UserMapper.getByName - <== Total: 1  
44:55.798 [main] INFO c.j.chat03.demo4.Demo4Test - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)

这个案例中我们新增的几个文件结构如下：



### 传递一个Map参数

#### 用法

如果我们需要传递的参数比较多，参数个数是动态的，那么我们可以将这些参数放在一个map中，key为参数名称，value为参数的值。

Mapper接口中可以这么定义，如：

List<UserModel> getByMap(Map<String,Object> map);

如我们传递：

Map<String, Object> map = new HashMap<>();  
 map.put("id", 1L);  
 map.put("name", "张学友");

对应的mapper xml中可以通过#{map中的key}可以获取key在map中对应的value的值作为参数，如：

SELECT \* FROM t\_user WHERE id=#{id} OR name = #{name}

#### 案例

下面我们通过map传递多个参数来按照id或者用户名进行查询。

com.javacode2018.chat03.demo4.mapper.UserMapper中新增一个方法，和上面UserMapper.xml中的对应，如下：

/\*\*  
 \* 通过map查询  
 \* @param map  
 \* @return  
 \*/  
List<UserModel> getByMap(Map<String,Object> map);

注意上面的方法由2个参数，参数名称分别为id、name，下面我们在对应的mapper xml中写对应的操作

chat03\src\main\resources\com\javacode2018\chat03\demo4\mapper\UserMapper.xml中新增下面代码：

<!-- 通过map查询 -->  
<select id="getByMap" resultType="com.javacode2018.chat03.demo4.model.UserModel">  
 <![CDATA[  
 SELECT \* FROM t\_user WHERE id=#{id} OR name = #{name}  
 ]]>  
</select>

大家注意一下上面的取值我们是使用#{id}取id参数的值，#{name}取name参数的值，下面我们创建测试用例，看看是否可以正常运行？

Demo4Test中新增下面方法：

/\*\*  
 \* 通过map给Mapper接口的方法传递参数  
 \*/  
@Test  
public void getByName() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
 UserModel userModel = userMapper.getByName("马士兵老师");  
 log.info("{}", userModel);  
 }  
}

运行一下上面的这个测试用例，输出：

01:28.242 [main] DEBUG c.j.c.d.mapper.UserMapper.getByMap - ==> Preparing: SELECT \* FROM t\_user WHERE id=? OR name = ?   
01:28.277 [main] DEBUG c.j.c.d.mapper.UserMapper.getByMap - ==> Parameters: 1(Long), 张学友(String)  
01:28.296 [main] DEBUG c.j.c.d.mapper.UserMapper.getByMap - <== Total: 2  
01:28.297 [main] INFO c.j.chat03.demo4.Demo4Test - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)  
01:28.298 [main] INFO c.j.chat03.demo4.Demo4Test - UserModel(id=3, name=张学友, age=56, salary=500000.0, sex=1)

### 传递一个java对象参数

当参数比较多，但是具体有多少个参数我们是确定的时候，我们可以将这些参数放在一个javabean对象中。

如我们想通过userId和userName查询，可以定义一个dto对象，属性添加对应的get、set方法，如：

@Getter  
@Setter  
@ToString  
@Builder  
@NoArgsConstructor  
@AllArgsConstructor  
public class UserFindDto {  
 private Long userId;  
 private String userName;  
}

注意上面的get、set方法我们通过lombok自动生成的。

UserMapper中新增一个方法，将UserFindDto作为参数：

/\*\*  
 \* 通过UserFindDto进行查询  
 \* @param userFindDto  
 \* @return  
 \*/  
List<UserModel> getListByUserFindDto(UserFindDto userFindDto);

对应的UserMapper.xml中这么写，如下：

<!-- 通过map查询 -->  
<select id="getListByUserFindDto" parameterType="com.javacode2018.chat03.demo4.dto.UserFindDto" resultType="com.javacode2018.chat03.demo4.model.UserModel">  
 <![CDATA[  
 SELECT \* FROM t\_user WHERE id=#{userId} OR name = #{userName}  
 ]]>  
</select>

Demo4Test中创建一个测试用例来调用一下新增的这个mapper接口中的方法，如下：

@Test  
public void getListByUserFindDto() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
 UserFindDto userFindDto = UserFindDto.builder().userId(1L).userName("张学友").build();  
 List<UserModel> userModelList = userMapper.getListByUserFindDto(userFindDto);  
 userModelList.forEach(item -> {  
 log.info("{}", item);  
 });  
 }  
}

上面我们通过传递一个userFindDto对象进行查询，运行输出：

20:59.454 [main] DEBUG c.j.c.d.m.U.getListByUserFindDto - ==> Preparing: SELECT \* FROM t\_user WHERE id=? OR name = ?   
20:59.487 [main] DEBUG c.j.c.d.m.U.getListByUserFindDto - ==> Parameters: 1(Long), 张学友(String)  
20:59.508 [main] DEBUG c.j.c.d.m.U.getListByUserFindDto - <== Total: 2  
20:59.509 [main] INFO c.j.chat03.demo4.Demo4Test - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)  
20:59.511 [main] INFO c.j.chat03.demo4.Demo4Test - UserModel(id=3, name=张学友, age=56, salary=500000.0, sex=1)

**传递java对象的方式相对于map的方式更清晰一些，可以明确知道具体有哪些参数，而传递map，我们是不知道这个map中具体需要哪些参数的，map对参数也没有约束，参数可以随意传，建议多个参数的情况下选择通过java对象进行传参。**

### 传递多个参数

上面我们介绍的都是传递一个参数，那么是否可以传递多个参数呢？我们来试试吧。

#### 案例

我们来新增一个通过用户id或用户名查询的操作。

com.javacode2018.chat03.demo4.mapper.UserMapper中新增一个方法，和上面UserMapper.xml中的对应，如下：

/\*\*  
 \* 通过id或者name查询  
 \*  
 \* @param id  
 \* @param name  
 \* @return  
 \*/  
UserModel getByIdOrName(Long id, String name);

注意上面的方法由2个参数，参数名称分别为id、name，下面我们在对应的mapper xml中写对应的操作

chat03\src\main\resources\com\javacode2018\chat03\demo4\mapper\UserMapper.xml中新增下面代码：

<!-- 通过id或者name查询 -->  
<select id="getByIdOrName" resultType="com.javacode2018.chat03.demo4.model.UserModel">  
 <![CDATA[  
 SELECT \* FROM t\_user WHERE id=#{id} OR name = #{name} LIMIT 1  
 ]]>  
</select>

**大家注意一下上面的取值我们是使用#{id}取id参数的值，#{name}取name参数的值，下面我们创建测试用例，看看是否可以正常运行？**

Demo4Test中新增下面方法：

/\*\*  
 \* 通过map给Mapper接口的方法传递参数  
 \*/  
@Test  
public void getByIdOrName() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
 UserModel userModel = userMapper.getByIdOrName(1L, "马士兵老师");  
 log.info("{}", userModel);  
 }  
}

运行一下上面的这个测试用例，报错了，我们截取部分主要的错误信息，如下：

org.apache.ibatis.exceptions.PersistenceException:   
### Error querying database. Cause: org.apache.ibatis.binding.BindingException: Parameter 'id' not found. Available parameters are [arg1, arg0, param1, param2]  
### Cause: org.apache.ibatis.binding.BindingException: Parameter 'id' not found. Available parameters are [arg1, arg0, param1, param2]  
  
 at org.apache.ibatis.exceptions.ExceptionFactory.wrapException(ExceptionFactory.java:30)  
 at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:149)  
 at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:140)  
 at org.apache.ibatis.session.defaults.DefaultSqlSession.selectOne(DefaultSqlSession.java:76)

上面报错，给大家解释一下，sql中我们通过#{id}去获取id参数的值，但是mybatis无法通过#{id}获取到id的值，所以报错了，从上错误中我们看到有这样的一段：

Available parameters are [arg1, arg0, param1, param2]

上面表示可用的参数名称列表，我们可以在sql中通过#{参数名称}来引参数列表中的参数，先不说这4个参数具体怎么来的，那么我们将sql改成下面这样试试：

SELECT \* FROM t\_user WHERE id=#{arg0} OR name = #{arg1} LIMIT 1

再运行一下测试用例，看看效果：

46:07.533 [main] DEBUG c.j.c.d.m.UserMapper.getByIdOrName - ==> Preparing: SELECT \* FROM t\_user WHERE id=? OR name = ? LIMIT 1   
46:07.566 [main] DEBUG c.j.c.d.m.UserMapper.getByIdOrName - ==> Parameters: 1(Long), 马士兵老师(String)  
46:07.585 [main] DEBUG c.j.c.d.m.UserMapper.getByIdOrName - <== Total: 1  
46:07.586 [main] INFO c.j.chat03.demo4.Demo4Test - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)

这下正常了。

我们将sql再修改一下，修改成下面这样：

SELECT \* FROM t\_user WHERE id=#{param1} OR name = #{param2} LIMIT 1

运行一下测试用例，输出：

47:19.935 [main] DEBUG c.j.c.d.m.UserMapper.getByIdOrName - ==> Preparing: SELECT \* FROM t\_user WHERE id=? OR name = ? LIMIT 1   
47:19.966 [main] DEBUG c.j.c.d.m.UserMapper.getByIdOrName - ==> Parameters: 1(Long), 马士兵老师(String)  
47:19.984 [main] DEBUG c.j.c.d.m.UserMapper.getByIdOrName - <== Total: 1  
47:19.985 [main] INFO c.j.chat03.demo4.Demo4Test - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)

也是正常的。

我们来分析一下mybatis对于这种多个参数是如何处理的？

#### 多参数mybatis的处理

mybatis处理多个参数的时候，会将多个参数封装到一个map中，map的key为参数的名称，java可以通过反射获取方法参数的名称，下面这个方法：

UserModel getByIdOrName(Long id, String name);

编译之后，方法参数的名称通过反射获取的并不是id、name，而是arg0、arg1，也就是说编译之后，方法真实的参数名称会丢失，会变成arg+参数下标的格式。

所以上面传递的参数相当于传递了下面这样的一个map：

Map<String,Object> map = new HashMap<>();  
map.put("arg0",id);  
map.put("arg1",name);

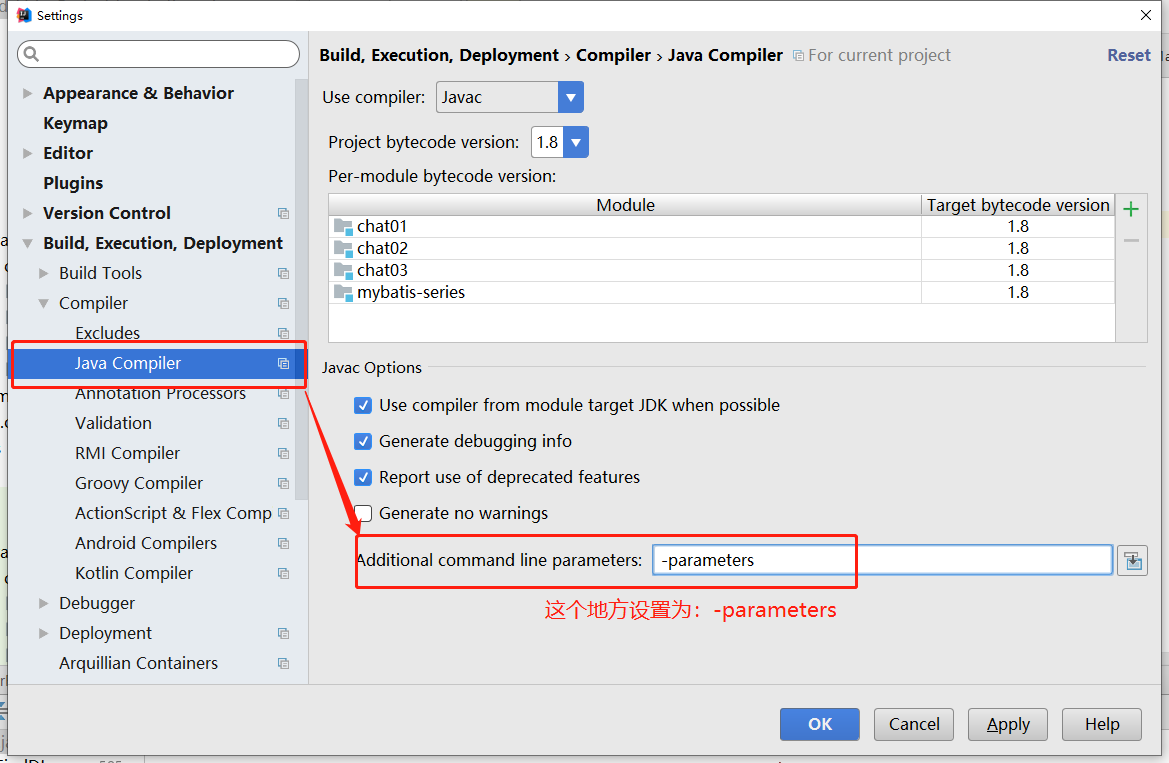
那么参数中的param1、param2又是什么呢？

上面的map中会放入按照参数名称->参数的值的方式将其放入map中，通过反射的方式获取的参数名称是可能会发生变化的，我们编译java代码使用javac命令，javac命令有个-parameters参数，当编译代码的时候加上这个参数，方法的实际名称会被编译到class字节码文件中，当通过反射获取方法名称的时候就不是arg0、arg1这种格式了，而是真实的参数名称：id、name了，我们来修改一下maven的配置让maven编译代码的时候加上这个参数，修改chat03/pom.xml中的build元素，这个元素中加入下面代码：

<plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-compiler-plugin</artifactId>  
 <version>3.3</version>  
 <configuration>  
 <compilerArgs>  
 <arg>-parameters</arg>  
 </compilerArgs>  
 </configuration>  
 </plugin>  
</plugins>

idea中编译代码也加一下这个参数，操作如下：

点击File->Settings->Build,Execution,Deployment->Java Compiler，如下图：



下面我们将demo4/UserMapper.xml中的getByIdOrName对应的sql修改成下面这样：

SELECT \* FROM t\_user WHERE id=#{arg0} OR name = #{arg1} LIMIT 1

使用maven命令重新编译一下chat03的代码，cmd命令中mybatis-series/pom.xml所在目录执行下面命令，如下：

D:\code\IdeaProjects\mybatis-series>mvn clean compile -pl :chat03  
[INFO] Scanning for projects...  
[INFO]  
[INFO] ----------------------< com.javacode2018:chat03 >-----------------------  
[INFO] Building chat03 1.0-SNAPSHOT  
[INFO] --------------------------------[ jar ]---------------------------------  
[INFO]  
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ chat03 ---  
[INFO] Deleting D:\code\IdeaProjects\mybatis-series\chat03\target  
[INFO]  
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ chat03 ---  
[INFO] Using 'UTF-8' encoding to copy filtered resources.  
[INFO] Copying 1 resource  
[INFO] Copying 11 resources  
[INFO]  
[INFO] --- maven-compiler-plugin:3.3:compile (default-compile) @ chat03 ---  
[INFO] Changes detected - recompiling the module!  
[INFO] Compiling 13 source files to D:\code\IdeaProjects\mybatis-series\chat03\target\classes  
[INFO] ------------------------------------------------------------------------  
[INFO] BUILD SUCCESS  
[INFO] ------------------------------------------------------------------------  
[INFO] Total time: 3.532 s  
[INFO] Finished at: 2019-12-10T13:41:05+08:00  
[INFO] ------------------------------------------------------------------------

再运行一下com.javacode2018.chat03.demo4.Demo4Test#getByIdOrName，输出如下：

org.apache.ibatis.exceptions.PersistenceException:   
### Error querying database. Cause: org.apache.ibatis.binding.BindingException: Parameter 'arg0' not found. Available parameters are [name, id, param1, param2]  
### Cause: org.apache.ibatis.binding.BindingException: Parameter 'arg0' not found. Available parameters are [name, id, param1, param2]  
  
 at org.apache.ibatis.exceptions.ExceptionFactory.wrapException(ExceptionFactory.java:30)  
 at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:149)  
 at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:140)  
 at org.apache.ibatis.session.defaults.DefaultSqlSession.selectOne(DefaultSqlSession.java:76)

又报错了，这次错误信息变了，注意有这么一行：

Parameter 'arg0' not found. Available parameters are [name, id, param1, param2]

参数名称变成了真实的名称了，但是还是有param1、param2，方法参数名称不管怎么变，编译方式如何变化，param1, param2始终在这里，这个param1, param2就是为了应对不同的编译方式导致参数名称而发生变化的，mybatis内部除了将参数按照名称->值的方式放入map外，还会按照参数的顺序放入一些值，这些值的key就是param+参数位置，这个位置从1开始的，所以id是第一个参数，对应的key是param1，name对应的key是param2，value对应的还是参数的值，所以mybatis对于参数的处理相当于下面过程：

Map<String,Object> map = new HashMap<>();  
map.put("反射获取的参数id的名称",id);  
map.put("反射获取的参数name的名称",name);  
map.put("param1",id);  
map.put("param2",name);

我们将demo4/UserMaper.xml中的getByIdOrName对应的sql改成param的方式，如下：

SELECT \* FROM t\_user WHERE id=#{param1} OR name = #{param2} LIMIT 1

再运行一下com.javacode2018.chat03.demo4.Demo4Test#getByIdOrName，输出如下，正常了：

51:12.588 [main] DEBUG c.j.c.d.m.UserMapper.getByIdOrName - ==> Preparing: SELECT \* FROM t\_user WHERE id=? OR name = ? LIMIT 1   
51:12.619 [main] DEBUG c.j.c.d.m.UserMapper.getByIdOrName - ==> Parameters: 1(Long), 马士兵老师(String)  
51:12.634 [main] DEBUG c.j.c.d.m.UserMapper.getByIdOrName - <== Total: 1  
51:12.635 [main] INFO c.j.chat03.demo4.Demo4Test - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)

#### 使用注意

1. **使用参数名称的方式对编译环境有很强的依赖性，如果编译中加上了-parameters参数，参数实际名称可以直接使用，如果没有加，参数名称就变成arg下标的格式了，这种很容易出错**
2. **sql中使用param1、param2、paramN这种方式来引用多参数，对参数的顺序依赖性特别强，如果有人把参数的顺序调整了或者调整了参数的个数，后果就是灾难性的，所以这种方式不建议大家使用。**

### 多参数中用@param指定参数名称

刚才上面讲了多参数传递的使用上面，对参数名称和顺序有很强的依赖性，容易导致一些严重的错误。

mybatis也为我们考虑到了这种情况，可以让我们自己去指定参数的名称，通过@param(“参数名称”)来给参数指定名称。

com.javacode2018.chat03.demo4.mapper.UserMapper#getByIdOrName做一下修改：

/\*\*  
 \* 通过id或者name查询  
 \*  
 \* @param id  
 \* @param name  
 \* @return  
 \*/  
UserModel getByIdOrName(@Param("userId") Long id, @Param("userName") String name);

上面我们通过@Param注解给两个参数明确指定了名称，分别是userId、userName，对应的UserMapper.xml中也做一下调整，如下：

<!-- 通过id或者name查询 -->  
<select id="getByIdOrName" resultType="com.javacode2018.chat03.demo4.model.UserModel">  
 <![CDATA[  
 SELECT \* FROM t\_user WHERE id=#{userId} OR name = #{userName} LIMIT 1  
 ]]>  
</select>

运行com.javacode2018.chat03.demo4.Demo4Test#getByMap，输出：

13:25.431 [main] DEBUG c.j.c.d.mapper.UserMapper.getByMap - ==> Preparing: SELECT \* FROM t\_user WHERE id=? OR name = ?   
13:25.460 [main] DEBUG c.j.c.d.mapper.UserMapper.getByMap - ==> Parameters: null, 张学友(String)  
13:25.477 [main] DEBUG c.j.c.d.mapper.UserMapper.getByMap - <== Total: 1  
13:25.478 [main] INFO c.j.chat03.demo4.Demo4Test - UserModel(id=3, name=张学友, age=56, salary=500000.0, sex=1)

### mybatis参数处理相关源码

上面参数的解析过程代码在org.apache.ibatis.reflection.ParamNameResolver类中，主要看下面的2个方法：

public ParamNameResolver(Configuration config, Method method)  
public Object getNamedParams(Object[] args)

这2个方法建议大家都设置一下断点细看一下整个过程，方法的实现不复杂，大家花半个小时去看一下加深一下理解。

下面我们继续说其他方式的传参。

### 传递1个Collection参数

当传递的参数类型是java.util.Collection的时候，会被放在map中，key为collection，value为参数的值，如下面的查询方法：

/\*\*  
 \* 查询用户id列表  
 \*  
 \* @param idCollection  
 \* @return  
 \*/  
List<UserModel> getListByIdCollection(Collection<Long> idCollection);

上面的查询方法，mybatis内部会将idList做一下处理：

Map<String,Object> map = new HashMap<>();  
map.put("collection",idCollection)

所以我们在mapper xml中使用的使用，需要通过collection名称来引用idCollection参数，如下：

<!-- 通过用户id列表查询 -->  
<select id="getListByIdCollection" resultType="com.javacode2018.chat03.demo4.model.UserModel">  
 <![CDATA[  
 SELECT \* FROM t\_user WHERE id IN (#{collection[0]},#{collection[1]})  
 ]]>  
</select>

com.javacode2018.chat03.demo4.Demo4Test中写个测试用例getListByIdList，查询2个用户信息，如下：

@Test  
public void getListByIdCollection() {  
 log.info("----------");  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
 List<Long> userIdList = Arrays.asList(1L, 3L);  
 List<UserModel> userModelList = userMapper.getListByIdCollection(userIdList);  
 userModelList.forEach(item -> {  
 log.info("{}", item);  
 });  
 }  
}

运行输出：

26:15.774 [main] INFO c.j.chat03.demo4.Demo4Test - ----------  
26:16.055 [main] DEBUG c.j.c.d.m.U.getListByIdCollection - ==> Preparing: SELECT \* FROM t\_user WHERE id IN (?,?)   
26:16.083 [main] DEBUG c.j.c.d.m.U.getListByIdCollection - ==> Parameters: 1(Long), 3(Long)  
26:16.102 [main] DEBUG c.j.c.d.m.U.getListByIdCollection - <== Total: 2  
26:16.103 [main] INFO c.j.chat03.demo4.Demo4Test - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)  
26:16.105 [main] INFO c.j.chat03.demo4.Demo4Test - UserModel(id=3, name=张学友, age=56, salary=500000.0, sex=1)

### Mybatis中集合参数处理了源码解析

集合参数，mybatis会进行一些特殊处理，代码在下面的方法中：

org.apache.ibatis.session.defaults.DefaultSqlSession#wrapCollection

这个方法的源码如下：

private Object wrapCollection(final Object object) {  
 if (object instanceof Collection) {  
 StrictMap<Object> map = new StrictMap<>();  
 map.put("collection", object);  
 if (object instanceof List) {  
 map.put("list", object);  
 }  
 return map;  
 } else if (object != null && object.getClass().isArray()) {  
 StrictMap<Object> map = new StrictMap<>();  
 map.put("array", object);  
 return map;  
 }  
 return object;  
 }

源码解释：

判断参数是否是java.util.Collection类型，如果是，会放在map中，key为collection。

如果参数是java.util.List类型的，会在map中继续放一个list作为key来引用这个对象。

如果参数是数组类型的，会通过array来引用这个对象。

传递1个List参数

从上面源码中可知，List类型的参数会被放在map中，可以通过2个key（collection和list）都可以引用到这个List对象。

com.javacode2018.chat03.demo4.mapper.UserMapper中新增一个方法：

/\*\*  
 \* 查询用户id列表  
 \*  
 \* @param idList  
 \* @return  
 \*/  
List<UserModel> getListByIdList(List<Long> idList);

对应的demo4/UserMaper.xml中增加一个操作，如下：

<!-- 通过用户id列表查询 -->  
<select id="getListByIdList" resultType="com.javacode2018.chat03.demo4.model.UserModel">  
 <![CDATA[  
 SELECT \* FROM t\_user WHERE id IN (#{list[0]},#{collection[1]})  
 ]]>  
</select>

注意上面我们使用了2中方式获取参数，通过list、collection都可以引用List类型的参数。

新增一个测试用例com.javacode2018.chat03.demo4.Demo4Test#getListByIdList，如下：

@Test  
public void getListByIdList() {  
 log.info("----------");  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
 List<Long> userIdList = Arrays.asList(1L, 3L);  
 List<UserModel> userModelList = userMapper.getListByIdList(userIdList);  
 userModelList.forEach(item -> {  
 log.info("{}", item);  
 });  
 }  
}

运行输出：

33:17.871 [main] INFO c.j.chat03.demo4.Demo4Test - ----------  
33:18.153 [main] DEBUG c.j.c.d.m.UserMapper.getListByIdList - ==> Preparing: SELECT \* FROM t\_user WHERE id IN (?,?)   
33:18.185 [main] DEBUG c.j.c.d.m.UserMapper.getListByIdList - ==> Parameters: 1(Long), 3(Long)  
33:18.207 [main] DEBUG c.j.c.d.m.UserMapper.getListByIdList - <== Total: 2  
33:18.208 [main] INFO c.j.chat03.demo4.Demo4Test - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)  
33:18.210 [main] INFO c.j.chat03.demo4.Demo4Test - UserModel(id=3, name=张学友, age=56, salary=500000.0, sex=1)

### 传递1个数组参数

数组类型的参数从上面源码中可知，sql中需要通过array来进行引用，这个就不写了，案例中也是有的，大家可以去看一下com.javacode2018.chat03.demo4.Demo4Test#getListByIdArray这个方法。

### ResultHandler作为参数

#### 用法

查询的数量比较大的时候，返回一个List集合占用的内存还是比较多的，比如我们想导出很多数据，实际上如果我们通过jdbc的方式，遍历ResultSet的next方法，一条条处理，而不用将其存到List集合中再取处理。

mybatis中也支持我们这么做，可以使用ResultHandler对象，犹如其名，这个接口是用来处理结果的，先看一下其定义：

public interface ResultHandler<T> {  
  
 void handleResult(ResultContext<? extends T> resultContext);  
  
}

里面有1个方法，方法的参数是ResultContext类型的，这个也是一个接口，看一下源码：

public interface ResultContext<T> {  
  
 T getResultObject();  
  
 int getResultCount();  
  
 boolean isStopped();  
  
 void stop();  
  
}

4个方法：

* getResultObject：获取当前行的结果
* getResultCount：获取当前结果到第几行了
* isStopped：判断是否需要停止遍历结果集
* stop：停止遍历结果集

ResultContext接口有一个实现类org.apache.ibatis.executor.result.DefaultResultContext，mybatis中默认会使用这个类。

#### 案例

我们遍历t\_user表的所有记录，第2条遍历结束之后，停止遍历，实现如下：

新增一个方法com.javacode2018.chat03.demo4.mapper.UserMapper#getList，如下：

void getList(ResultHandler<UserModel> resultHandler);

对应的UserMapper.xml新增sql操作，如下：

<select id="getList" resultType="com.javacode2018.chat03.demo4.model.UserModel">  
 <![CDATA[  
 SELECT \* FROM t\_user  
 ]]>  
</select>

新增测试用例com.javacode2018.chat03.demo4.Demo4Test#getList，如下：

@Test  
public void getList() {  
 log.info("----------");  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
 userMapper.getList(context -> {  
 //将context参数转换为DefaultResultContext对象  
 DefaultResultContext<UserModel> defaultResultContext = (DefaultResultContext<UserModel>) context;  
 log.info("{}", defaultResultContext.getResultObject());  
 //遍历到第二条之后停止  
 if (defaultResultContext.getResultCount() == 2) {  
 //调用stop方法停止遍历，stop方法会更新内部的一个标志，置为停止遍历  
 defaultResultContext.stop();  
 }  
 });  
 }  
}

运行输出：

07:05.561 [main] INFO c.j.chat03.demo4.Demo4Test - ----------  
07:05.816 [main] DEBUG c.j.c.d.mapper.UserMapper.getList - ==> Preparing: SELECT \* FROM t\_user   
07:05.845 [main] DEBUG c.j.c.d.mapper.UserMapper.getList - ==> Parameters:   
07:05.864 [main] INFO c.j.chat03.demo4.Demo4Test - UserModel(id=1, name=马士兵老师, age=30, salary=50000.0, sex=1)  
07:05.867 [main] INFO c.j.chat03.demo4.Demo4Test - UserModel(id=2, name=javacode2018, age=30, salary=50000.0, sex=1)

**本文的内容希望大家都能掌握。**

### MyBatis系列

1. MyBatis系列第1篇：MyBatis未出世之前我们那些痛苦的经历
2. MyBatis系列第2篇：入门篇，带你感受一下mybatis独特的魅力！
3. MyBatis系列第3篇：Mybatis使用详解（1）
4. MyBatis系列第4篇：Mybatis使用详解（2）

## Mybatis系列第6篇：增删改知识点汇总及主键获取3种方式详解

### 主要内容

* 建库建表
* mybatis增删改返回值说明及源码解析
* jdbc获取自增值的3种方式详解
* mybatis获取自增值的3种方式详解

### 建库建表

/\*创建数据库javacode2018\*/  
DROP DATABASE IF EXISTS `javacode2018`;  
CREATE DATABASE `javacode2018`;  
USE `javacode2018`;  
  
/\*创建表结构\*/  
DROP TABLE IF EXISTS `t\_user`;  
CREATE TABLE t\_user (  
 id BIGINT AUTO\_INCREMENT PRIMARY KEY COMMENT '主键，用户id，自动增长',  
 `name` VARCHAR(32) NOT NULL DEFAULT '' COMMENT '姓名',  
 `age` SMALLINT NOT NULL DEFAULT 1 COMMENT '年龄',  
 `salary` DECIMAL(12,2) NOT NULL DEFAULT 0 COMMENT '薪水',  
 `sex` TINYINT NOT NULL DEFAULT 0 COMMENT '性别,0:未知,1:男,2:女'  
) COMMENT '用户表';  
  
SELECT \* FROM t\_user;

### 增删改返回值说明

mybatis中对db执行增删改操作，不管是新增、删除、还是修改，最后都会去调用jdbc中对应的方法，要么是调用java.sql.Statement的executeUpdate的方法，要么是调用java.sql.PreparedStatement的executeUpdate方法，这2个类的方法名称都是executeUpdate，他们的参数可能不一样，但是他们的返回值都是int，说明增删改的返回值都是int类型的，表示影响的行数，比如插入成功1行返回结果就是1，删除了10行记录，返回就是10，更新了5行记录，返回的就是5。

那么我们通过Mybatis中的Mapper接口来对db增删改的时候，mybatis的返回值支持哪些类型呢？

int类型那肯定是支持的，jdbc执行增删改默认返回int类型，那mybatis当然也支持这个类型。

但是mybatis的返回值比jdbc更强大，对于增删改还支持下面几种类型：

int  
Integer  
long   
Long  
boolean  
Boolean  
void

mapper的增删改方法返回值必须为上面的类型，mybatis内部将jdbc返回的int类型转换为上面列表中指定的类型，我们来看一下mybatis这块的源码，源码在下面的方法中：

org.apache.ibatis.binding.MapperMethod#rowCountResult

我们来看一下这个方法的源码：

private Object rowCountResult(int rowCount) {  
 final Object result;  
 if (method.returnsVoid()) {  
 result = null;  
 } else if (Integer.class.equals(method.getReturnType()) || Integer.TYPE.equals(method.getReturnType())) {  
 result = rowCount;  
 } else if (Long.class.equals(method.getReturnType()) || Long.TYPE.equals(method.getReturnType())) {  
 result = (long)rowCount;  
 } else if (Boolean.class.equals(method.getReturnType()) || Boolean.TYPE.equals(method.getReturnType())) {  
 result = rowCount > 0;  
 } else {  
 throw new BindingException("Mapper method '" + command.getName() + "' has an unsupported return type: " + method.getReturnType());  
 }  
 return result;  
 }

mybatis中会使用上面这个方法最后会对jdbc 增删改返回的int结果进行处理，处理为mapper接口中增删改方法返回值的类型。

**int、Integer、long、Long我们就不说了，主要说一下返回值是boolean、Boolean类型，如果影响的行数大于0了，将返回true。**

下面我们来创建一个工程感受一下增删改各种返回值。

### 创建案例

整个mybatis系列的代码采用maven模块的方式管理的，可以在文章底部获取，本次我们还是在上一篇的mybatis-series中进行开发，在这个项目中新建一个模块chat04，模块坐标如下：

<groupId>com.javacode2018</groupId>  
<artifactId>chat04</artifactId>  
<version>1.0-SNAPSHOT</version>

下面我们通过mybatis快速来实现对t\_user表增删改。

#### 创建UserModel类

mybatis-series\chat04\src\main\java\com\javacode2018\chat04\demo1\model目录创建UserModel.java，如下：

package com.javacode2018.chat04.demo1.model;  
  
import lombok.\*;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
@Getter  
@Setter  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
@ToString  
public class UserModel {  
 private Long id;  
 private String name;  
 private Integer age;  
 private Double salary;  
 private Integer sex;  
}

#### 创建UserMapper接口

mybatis-series\chat04\src\main\java\com\javacode2018\chat04\demo1\mapper目录创建UserMapper.java，如下：

package com.javacode2018.chat04.demo1.mapper;  
  
import com.javacode2018.chat04.demo1.model.UserModel;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
public interface UserMapper {  
  
 /\*\*  
 \* 插入用户信息，返回影响行数  
 \*  
 \* @param model  
 \* @return  
 \*/  
 int insertUser(UserModel model);  
  
 /\*\*  
 \* 更新用户信息，返回影响行数  
 \*  
 \* @param model  
 \* @return  
 \*/  
 long updateUser(UserModel model);  
  
 /\*\*  
 \* 根据用户id删除用户信息，返回删除是否成功  
 \*  
 \* @param userId  
 \* @return  
 \*/  
 boolean deleteUser(Long userId);  
  
}

注意上面3个操作的返回类型，我们体验一下int、long、boolean类型的返回值。

#### 创建UserMapper.xml文件

mybatis-series\chat04\src\main\resources\demo1目录创建，UserMapper.xml，如下：

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper namespace="com.javacode2018.chat04.demo1.mapper.UserMapper">  
  
 <insert id="insertUser" parameterType="com.javacode2018.chat04.demo1.model.UserModel">  
 <![CDATA[  
 INSERT INTO t\_user (id,name,age,salary,sex) VALUES (#{id},#{name},#{age},#{salary},#{sex})  
 ]]>  
 </insert>  
  
 <update id="updateUser" parameterType="com.javacode2018.chat04.demo1.model.UserModel">  
 <![CDATA[  
 UPDATE t\_user SET name = #{name},age = #{age},salary = #{salary},sex = #{sex} WHERE id = #{id}  
 ]]>  
 </update>  
  
 <update id="deleteUser" parameterType="java.lang.Long">  
 <![CDATA[  
 DELETE FROM t\_user WHERE id = #{id}  
 ]]>  
 </update>  
  
</mapper>

#### 创建属性配置文件

mybatis-series\chat04\src\main\resources目录中创建jdbc.properties，如下：

jdbc.driver=com.mysql.jdbc.Driver  
jdbc.url=jdbc:mysql://localhost:3306/javacode2018?characterEncoding=UTF-8  
jdbc.username=root  
jdbc.password=root123

#### 创建mybatis全局配置文件

mybatis-series\chat04\src\main\resources\demo1目录创建，mybatis-config.xml，如下：

<?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jdbc配置 -->  
 <properties resource="jdbc.properties"/>  
 <!-- 环境配置，可以配置多个环境 -->  
 <environments default="demo4">  
 <environment id="demo4">  
 <!-- 事务管理器工厂配置 -->  
 <transactionManager type="JDBC"/>  
 <!-- 数据源工厂配置，使用工厂来创建数据源 -->  
 <dataSource type="POOLED">  
 <property name="driver" value="${jdbc.driver}"/>  
 <property name="url" value="${jdbc.url}"/>  
 <property name="username" value="${jdbc.username}"/>  
 <property name="password" value="${jdbc.password}"/>  
 </dataSource>  
 </environment>  
 </environments>  
  
 <mappers>  
 <mapper resource="demo1/UserMapper.xml" />  
 </mappers>  
</configuration>

#### 引入logback日志支持

chat04\src\main\resources目录创建logback.xml，如下：

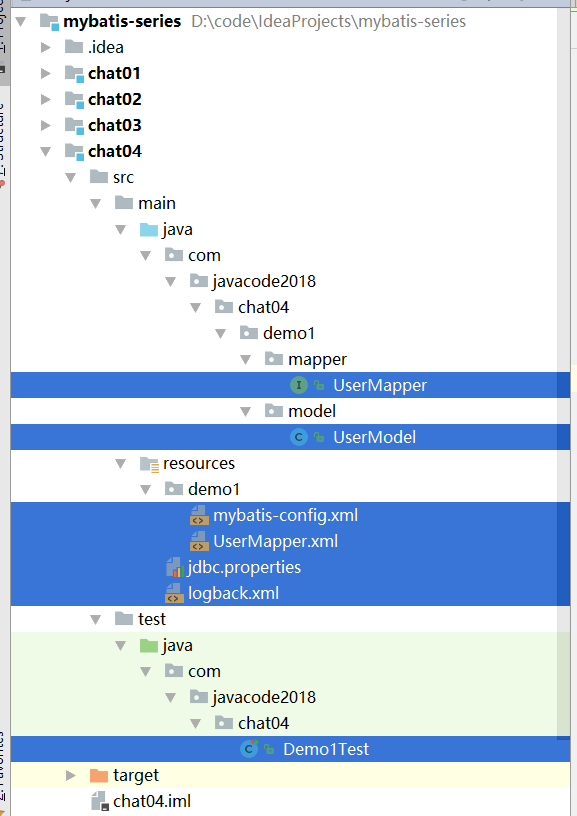
<?xml version="1.0" encoding="UTF-8"?>  
<configuration>  
 <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
 <encoder>  
 <pattern>%d{mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>  
 </encoder>  
 </appender>  
  
 <logger name="com.javacode2018" level="debug" additivity="false">  
 <appender-ref ref="STDOUT" />  
 </logger>  
  
</configuration>

#### 创建测试用例Demo1Test

mybatis-series\chat04\src\test\java\com\javacode2018\chat04目录创建Demo1Test.java，如下：

package com.javacode2018.chat04;  
  
import com.javacode2018.chat04.demo1.mapper.UserMapper;  
import com.javacode2018.chat04.demo1.model.UserModel;  
import lombok.extern.slf4j.Slf4j;  
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 java.io.IOException;  
import java.io.InputStream;  
  
/\*\*  
 \* 公众号：马士兵老师，工作10年的前阿里P7分享Java、算法、数据库方面的技术干货！坚信用技术改变命运，让家人过上更体面的生活!  
 \*/  
@Slf4j  
public class Demo1Test {  
 private SqlSessionFactory sqlSessionFactory;  
  
 @Before  
 public void before() throws IOException {  
 //指定mybatis全局配置文件  
 String resource = "demo1/mybatis-config.xml";  
 //读取全局配置文件  
 InputStream inputStream = Resources.getResourceAsStream(resource);  
 //构建SqlSessionFactory对象  
 SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
 this.sqlSessionFactory = sqlSessionFactory;  
 }  
  
 @Test  
 public void insertUser() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 //创建UserModel对象  
 UserModel userModel = UserModel.builder().id(1L).name("马士兵老师").age(30).salary(50000D).sex(1).build();  
 //执行插入操作  
 int insert = mapper.insertUser(userModel);  
 log.info("影响行数：{}", insert);  
 }  
 }  
  
 @Test  
 public void updateUser() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 //创建UserModel对象  
 UserModel userModel = UserModel.builder().id(1L).name("马士兵老师，你好").age(18).salary(5000D).sex(0).build();  
 //执行更新操作  
 long result = mapper.updateUser(userModel);  
 log.info("影响行数：{}", result);  
 }  
 }  
  
 @Test  
 public void deleteUser() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 //定义需要删除的用户id  
 Long userId = 1L;  
 //执行删除操作  
 boolean result = mapper.deleteUser(userId);  
 log.info("第1次删除:id={},返回值:{}", userId, result);  
 result = mapper.deleteUser(userId);  
 log.info("第2次删除:id={},返回值:{}", userId, result);  
 }  
 }  
}

#### 项目结构如下图



注意项目结构如下图，跑起来有问题的可以对照一下。

#### 运行测试用例

##### 测试int类型返回值

运行com.javacode2018.chat04.Demo1Test#insertUser，插入一条用户信息，输出如下：

16:35.821 [main] DEBUG c.j.c.d.mapper.UserMapper.insertUser - ==> Preparing: INSERT INTO t\_user (id,name,age,salary,sex) VALUES (?,?,?,?,?)   
16:35.858 [main] DEBUG c.j.c.d.mapper.UserMapper.insertUser - ==> Parameters: 1(Long), 马士兵老师(String), 30(Integer), 50000.0(Double), 1(Integer)  
16:35.865 [main] DEBUG c.j.c.d.mapper.UserMapper.insertUser - <== Updates: 1  
16:35.865 [main] INFO com.javacode2018.chat04.Demo1Test - 影响行数：1

##### 测试long类型返回值

运行com.javacode2018.chat04.Demo1Test#updateUser，通过用户id更新用户信息，输出如下：

17:49.084 [main] DEBUG c.j.c.d.mapper.UserMapper.updateUser - ==> Preparing: UPDATE t\_user SET name = ?,age = ?,salary = ?,sex = ? WHERE id = ?   
17:49.127 [main] DEBUG c.j.c.d.mapper.UserMapper.updateUser - ==> Parameters: 马士兵老师，你好(String), 18(Integer), 5000.0(Double), 0(Integer), 1(Long)  
17:49.135 [main] DEBUG c.j.c.d.mapper.UserMapper.updateUser - <== Updates: 1  
17:49.135 [main] INFO com.javacode2018.chat04.Demo1Test - 影响行数：1

##### 测试boolean类型返回值

运行com.javacode2018.chat04.Demo1Test#deleteUser，根据用户id删除用户信息，删除2次，输出如下：

20:37.745 [main] DEBUG c.j.c.d.mapper.UserMapper.deleteUser - ==> Preparing: DELETE FROM t\_user WHERE id = ?   
20:37.785 [main] DEBUG c.j.c.d.mapper.UserMapper.deleteUser - ==> Parameters: 1(Long)  
20:37.790 [main] DEBUG c.j.c.d.mapper.UserMapper.deleteUser - <== Updates: 0  
20:37.791 [main] INFO com.javacode2018.chat04.Demo1Test - 第1次删除:id=1,返回值:false  
20:37.793 [main] DEBUG c.j.c.d.mapper.UserMapper.deleteUser - ==> Preparing: DELETE FROM t\_user WHERE id = ?   
20:37.794 [main] DEBUG c.j.c.d.mapper.UserMapper.deleteUser - ==> Parameters: 1(Long)  
20:37.795 [main] DEBUG c.j.c.d.mapper.UserMapper.deleteUser - <== Updates: 0  
20:37.795 [main] INFO com.javacode2018.chat04.Demo1Test - 第2次删除:id=1,返回值:false

第一次删除成功，再次删除数据已经不存在了，返回false。

### jdbc获取主键的几种方式

上面的案例中inserUser会向t\_user表插入数据，t\_user表的id是自动增长的，插入数据的时候我们不指定id的值，看看插入成功之后userModel对象和db中插入的记录是什么样的。

com.javacode2018.chat04.Demo1Test#insertUser代码改成下面这样：

@Test  
public void insertUser() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 //创建UserModel对象  
 UserModel userModel = UserModel.builder().name("郭富城").age(30).salary(50000D).sex(1).build();  
 //执行插入操作  
 int insert = mapper.insertUser(userModel);  
 log.info("影响行数：{}", insert);  
 log.info("{}", userModel);  
 }  
}

执行一下，输出：

36:10.673 [main] DEBUG c.j.c.d.mapper.UserMapper.insertUser - ==> Preparing: INSERT INTO t\_user (id,name,age,salary,sex) VALUES (?,?,?,?,?)   
36:10.715 [main] DEBUG c.j.c.d.mapper.UserMapper.insertUser - ==> Parameters: null, 郭富城(String), 30(Integer), 50000.0(Double), 1(Integer)  
36:10.721 [main] DEBUG c.j.c.d.mapper.UserMapper.insertUser - <== Updates: 1  
36:10.722 [main] INFO com.javacode2018.chat04.Demo1Test - 影响行数：1  
36:10.723 [main] INFO com.javacode2018.chat04.Demo1Test - UserModel(id=null, name=郭富城, age=30, salary=50000.0, sex=1)

输出中插入成功1行，最后一行日志中输出了userModel对象所有属性信息，id是null的，我们去db中看一下这条记录：

mysql> SELECT \* FROM t\_user;  
+----+-----------+-----+----------+-----+  
| id | name | age | salary | sex |  
+----+-----------+-----+----------+-----+  
| 2 | 郭富城 | 30 | 50000.00 | 1 |  
+----+-----------+-----+----------+-----+  
1 row in set (0.00 sec)

db中插入的这条郭富城的id是2，当我们没有指定id，或者指定的id为null的时候，mysql会自动生成id的值。

那么我们如何mysql中获取这个自动增长的值呢？我们先看看jdbc是如何实现的

#### 方式1：jdbc内置的方式

##### 用法

jdbc的api中为我们提供了获取自动生成主键的值，具体看这个方法：

java.sql.Statement#getGeneratedKeys

看一下这个方法的定义：

/\*\*  
\* Retrieves any auto-generated keys created as a result of executing this  
\* <code>Statement</code> object. If this <code>Statement</code> object did  
\* not generate any keys, an empty <code>ResultSet</code>  
\* object is returned.  
\*  
\*<p><B>Note:</B>If the columns which represent the auto-generated keys were not specified,  
\* the JDBC driver implementation will determine the columns which best represent the auto-generated keys.  
\*  
\* @return a <code>ResultSet</code> object containing the auto-generated key(s)  
\* generated by the execution of this <code>Statement</code> object  
\* @exception SQLException if a database access error occurs or  
\* this method is called on a closed <code>Statement</code>  
\* @throws SQLFeatureNotSupportedException if the JDBC driver does not support this method  
\* @since 1.4  
\*/  
ResultSet getGeneratedKeys() throws SQLException;

这个方法会返回一个结果集，从这个结果集中可以获取自增主键的值。

不过使用这个方法有个前提，执行sql的时候需要做一个设置。

如果是通过java.sql.Statement执行sql，需要调用下面这个方法：

int executeUpdate(String sql, int autoGeneratedKeys) throws SQLException

注意上面这个方法的第二个参数需要设置为java.sql.Statement.RETURN\_GENERATED\_KEYS，表示需要返回自增列的值。

不过多数情况下，我们会使用java.sql.PreparedStatement对象来执行sql，如果想获取自增值，创建这个对象需要设置第2个参数的值，如下：

PreparedStatement preparedStatement = connection.prepareStatement(sql, Statement.RETURN\_GENERATED\_KEYS);

然后我们就可以通过getGeneratedKeys返回的ResultSet对象获取自动增长的值了，如下：

ResultSet generatedKeys = preparedStatement.getGeneratedKeys();  
if (generatedKeys!=null && generatedKeys.next()) {  
 log.info("自增值为：{}", generatedKeys.getInt(1));  
}

##### 案例

com.javacode2018.chat04.Demo1Test中新增一个测试用例，如下代码：

private String jdbcDriver = "com.mysql.jdbc.Driver";  
private String jdbcUrl = "jdbc:mysql://localhost:3306/javacode2018?characterEncoding=UTF-8";  
private String jdbcUserName = "root";  
private String jdbcPassword = "root123";  
  
@Test  
public void jdbcInsertUser1() throws Exception {  
 Connection connection = null;  
 PreparedStatement preparedStatement = null;  
 ResultSet generatedKeys = null;  
 try {  
  
 UserModel userModel = UserModel.builder().name("黎明").age(30).salary(50000D).sex(1).build();  
 //执行jdbc插入数据操作  
 Class.forName(jdbcDriver);  
 connection = DriverManager.getConnection(jdbcUrl, jdbcUserName, jdbcPassword);  
 //注意创建PreparedStatement的时候，使用prepareStatement方法的第二个参数需要指定Statement.RETURN\_GENERATED\_KEYS  
 preparedStatement = connection.prepareStatement("INSERT INTO t\_user (name,age,salary,sex) VALUES (?,?,?,?)", Statement.RETURN\_GENERATED\_KEYS);  
 int parameterIndex = 1;  
 preparedStatement.setString(parameterIndex++, userModel.getName());  
 preparedStatement.setInt(parameterIndex++, userModel.getAge());  
 preparedStatement.setDouble(parameterIndex++, userModel.getSalary());  
 preparedStatement.setInt(parameterIndex++, userModel.getSex());  
 int count = preparedStatement.executeUpdate();  
 log.info("影响行数：{}", count);  
  
 //获取自增值  
 generatedKeys = preparedStatement.getGeneratedKeys();  
 if (generatedKeys != null && generatedKeys.next()) {  
 log.info("自增值为：{}", generatedKeys.getInt(1));  
 }  
 } finally {  
 if (generatedKeys != null && generatedKeys.isClosed()) {  
 generatedKeys.close();  
 }  
 if (preparedStatement != null && preparedStatement.isClosed()) {  
 preparedStatement.close();  
 }  
 if (connection != null && connection.isClosed()) {  
 connection.close();  
 }  
 }  
}

上面代码中我们插入了一条用户的信息，没有指定用户的id，执行输出：

21:22.410 [main] INFO com.javacode2018.chat04.Demo1Test - 影响行数：1  
21:22.414 [main] INFO com.javacode2018.chat04.Demo1Test - 自增值为：5

我们去db中看一下这个记录的id，如下，确实是5：

mysql> SELECT \* FROM t\_user;  
+----+--------+-----+----------+-----+  
| id | name | age | salary | sex |  
+----+--------+-----+----------+-----+  
| 5 | 黎明 | 30 | 50000.00 | 1 |  
+----+--------+-----+----------+-----+  
1 row in set (0.00 sec)

#### 方式2：插入之后查询获取

##### 用法

mysql中插入一条数据之后，可以通过下面的sql获取最新插入记录的id的值：

SELECT LAST\_INSERT\_ID()

那么我们可以在插入之后，立即使用当前连接发送上面这条sql去获取自增列的值就可以。

##### 案例

创建测试用例com.javacode2018.chat04.Demo1Test#jdbcInsertUser2,代码如下：

@Test  
public void jdbcInsertUser2() throws Exception {  
 Connection connection = null;  
 PreparedStatement preparedStatement = null;  
 ResultSet rs = null;  
 try {  
 UserModel userModel = UserModel.builder().name("梁朝伟").age(30).salary(50000D).sex(1).build();  
 //执行jdbc插入数据操作  
 Class.forName(jdbcDriver);  
 connection = DriverManager.getConnection(jdbcUrl, jdbcUserName, jdbcPassword);  
 //注意创建PreparedStatement的时候，使用prepareStatement方法的第二个参数需要指定Statement.RETURN\_GENERATED\_KEYS  
 preparedStatement = connection.prepareStatement("INSERT INTO t\_user (name,age,salary,sex) VALUES (?,?,?,?)", Statement.RETURN\_GENERATED\_KEYS);  
 int parameterIndex = 1;  
 preparedStatement.setString(parameterIndex++, userModel.getName());  
 preparedStatement.setInt(parameterIndex++, userModel.getAge());  
 preparedStatement.setDouble(parameterIndex++, userModel.getSalary());  
 preparedStatement.setInt(parameterIndex++, userModel.getSex());  
 int count = preparedStatement.executeUpdate();  
 log.info("影响行数：{}", count);  
  
 //通过查询获取自增值  
 rs = connection.prepareStatement("SELECT LAST\_INSERT\_ID()").executeQuery();  
 if (rs != null && rs.next()) {  
 log.info("自增值为：{}", rs.getInt(1));  
 }  
 } finally {  
 if (rs != null && rs.isClosed()) {  
 rs.close();  
 }  
 if (preparedStatement != null && preparedStatement.isClosed()) {  
 preparedStatement.close();  
 }  
 if (connection != null && connection.isClosed()) {  
 connection.close();  
 }  
 }  
}

运行输出：

26:55.407 [main] INFO com.javacode2018.chat04.Demo1Test - 影响行数：1  
26:55.414 [main] INFO com.javacode2018.chat04.Demo1Test - 自增值为：6

db中我们去看一下，梁朝伟的id是6，如下：

mysql> SELECT \* FROM t\_user;  
+----+-----------+-----+----------+-----+  
| id | name | age | salary | sex |  
+----+-----------+-----+----------+-----+  
| 5 | 黎明 | 30 | 50000.00 | 1 |  
| 6 | 梁朝伟 | 30 | 50000.00 | 1 |  
+----+-----------+-----+----------+-----+  
2 rows in set (0.00 sec)

#### 方式3：插入之前获取

oracle不知道大家有没有玩过，oracle中没有mysql中自动增长列，但是oracle有个功能可以实现自动增长，这个功能就是序列，序列就相当于一个自增器一样，有个初始值，每次递增的步长，当然这个序列提供了一些功能给我们使用，可以获取序列的当前值、下一个值，使用方式如下：

1.先定义一个序列  
2.获取下一个值：SELECT 序列名.NEXTVAL FROM dual;

这个案例我只说一下具体步骤，代码就不写了，步骤：

1.通过jdbc执行`SELECT 序列名.NEXTVAL FROM dual`获取序列的下一个值，如nextId  
2.在代码中使用nextId的值

上面就是jdbc获取值增值的几种方式，jdbc中的这3中方式，mybatis中都提供了对应的 支持，下面我们来看mybatis中是如何实现的。

### mybatis获取主键的3种方式

#### 方式1：内部使用jdbc内置的方式

##### 用法

mybatis这个方式内部采用的是上面说的jdbc内置的方式。

我们需要在Mapper xml中进行配置，如：

<insert id="insertUser1" parameterType="com.javacode2018.chat04.demo1.model.UserModel" useGeneratedKeys="true" keyProperty="id">  
 <![CDATA[  
 INSERT INTO t\_user (name,age,salary,sex) VALUES (#{name},#{age},#{salary},#{sex})  
 ]]>  
</insert>

有2个关键参数必须要设置：

* useGeneratedKeys：设置为true
* keyProperty：参数对象中的属性名称，最后插入成功之后，mybatis会通过反射将自增值设置给keyProperty指定的这个属性

##### 案例

mybatis-series\chat04\src\main\resources\demo1\UserMapper.xml中新增代码：

<!-- 插入的时候获取值增值，必须需指定2个属性  
 useGeneratedKeys：设置为true  
 keyProperty：参数对象中的属性，插入成功之后会将值增值设置给这个属性  
 -->  
<insert id="insertUser1" parameterType="com.javacode2018.chat04.demo1.model.UserModel" useGeneratedKeys="true" keyProperty="id">  
 <![CDATA[  
 INSERT INTO t\_user (name,age,salary,sex) VALUES (#{name},#{age},#{salary},#{sex})  
 ]]>  
</insert>

Mapper接口中也新增代码，com.javacode2018.chat04.demo1.mapper.UserMapper中新增一个方法，如下：

int insertUser1(UserModel userModel);

创建测试用例方法com.javacode2018.chat04.Demo1Test#insertUser1，如下：

@Test  
public void insertUser1() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 //创建UserModel对象  
 UserModel userModel = UserModel.builder().name("陈宝国").age(30).salary(50000D).sex(1).build();  
 //执行插入操作  
 int insert = mapper.insertUser1(userModel);  
 log.info("影响行数：{}", insert);  
 log.info("{}", userModel);  
 }  
}

注意上面的userModel对象，id没有设置值，运行输出：

59:44.412 [main] DEBUG c.j.c.d.m.UserMapper.insertUser1 - ==> Preparing: INSERT INTO t\_user (name,age,salary,sex) VALUES (?,?,?,?)   
59:44.444 [main] DEBUG c.j.c.d.m.UserMapper.insertUser1 - ==> Parameters: 陈宝国(String), 30(Integer), 50000.0(Double), 1(Integer)  
59:44.451 [main] DEBUG c.j.c.d.m.UserMapper.insertUser1 - <== Updates: 1  
59:44.453 [main] INFO com.javacode2018.chat04.Demo1Test - 影响行数：1  
59:44.455 [main] INFO com.javacode2018.chat04.Demo1Test - UserModel(id=8, name=陈宝国, age=30, salary=50000.0, sex=1)

看上面最后一行输出，id的值为8，去db中看一下，如下：

mysql> SELECT \* FROM t\_user;  
+----+-----------+-----+----------+-----+  
| id | name | age | salary | sex |  
+----+-----------+-----+----------+-----+  
| 5 | 黎明 | 30 | 50000.00 | 1 |  
| 6 | 梁朝伟 | 30 | 50000.00 | 1 |  
| 7 | 陈宝国 | 30 | 50000.00 | 1 |  
| 8 | 陈宝国 | 30 | 50000.00 | 1 |  
+----+-----------+-----+----------+-----+  
4 rows in set (0.00 sec)

#### 方式2：插入后查询获取主键

##### 用法

这个方式和上面介绍的jdbc的第二种方式一样，插入之后通过查询获取主键的值然后填充给指定的属性，mapper xml配置如下：

<insert id="insertUser2" parameterType="com.javacode2018.chat04.demo1.model.UserModel">  
 <selectKey keyProperty="id" order="AFTER" resultType="long">  
 <![CDATA[  
 SELECT LAST\_INSERT\_ID()  
 ]]>  
 </selectKey>  
 <![CDATA[  
 INSERT INTO t\_user (name,age,salary,sex) VALUES (#{name},#{age},#{salary},#{sex})  
 ]]>  
</insert>

关键代码是selectKey元素包含的部分，这个元素内部可以包含一个sql，这个sql可以在插入之前或者插入之后运行（之前还是之后通过order属性配置），然后会将sql运行的结果设置给keyProperty指定的属性，selectKey元素有3个属性需要指定：

* keyProperty：参数对象中的属性名称，最后插入成功之后，mybatis会通过反射将自增值设置给keyProperty指定的这个属性
* order：指定selectKey元素中的sql是在插入之前运行还是插入之后运行，可选值（BEFORE|AFTER），这种方式中我们选择AFTER
* resultType：keyProperty指定的属性对应的类型，如上面的id对应的类型是java.lang.Long，我们直接写的是别名long

##### 案例

mybatis-series\chat04\src\main\resources\demo1\UserMapper.xml中新增代码：

<insert id="insertUser2" parameterType="com.javacode2018.chat04.demo1.model.UserModel">  
 <selectKey keyProperty="id" order="AFTER" resultType="long">  
 <![CDATA[  
 SELECT LAST\_INSERT\_ID()  
 ]]>  
 </selectKey>  
 <![CDATA[  
 INSERT INTO t\_user (name,age,salary,sex) VALUES (#{name},#{age},#{salary},#{sex})  
 ]]>  
</insert>

Mapper接口中也新增代码，com.javacode2018.chat04.demo1.mapper.UserMapper中新增一个方法，如下：

int insertUser2(UserModel userModel);

创建测试用例方法com.javacode2018.chat04.Demo1Test#insertUser2，如下：

@Test  
public void insertUser2() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 //创建UserModel对象  
 UserModel userModel = UserModel.builder().name("周润发").age(30).salary(50000D).sex(1).build();  
 //执行插入操作  
 int insert = mapper.insertUser2(userModel);  
 log.info("影响行数：{}", insert);  
 log.info("{}", userModel);  
 }  
}

注意上面的userModel对象，id没有设置值，运行输出：

22:18.140 [main] DEBUG c.j.c.d.m.UserMapper.insertUser2 - ==> Preparing: INSERT INTO t\_user (name,age,salary,sex) VALUES (?,?,?,?)   
22:18.173 [main] DEBUG c.j.c.d.m.UserMapper.insertUser2 - ==> Parameters: 周润发(String), 30(Integer), 50000.0(Double), 1(Integer)  
22:18.180 [main] DEBUG c.j.c.d.m.UserMapper.insertUser2 - <== Updates: 1  
22:18.183 [main] DEBUG c.j.c.d.m.U.insertUser2!selectKey - ==> Preparing: SELECT LAST\_INSERT\_ID()   
22:18.183 [main] DEBUG c.j.c.d.m.U.insertUser2!selectKey - ==> Parameters:   
22:18.197 [main] DEBUG c.j.c.d.m.U.insertUser2!selectKey - <== Total: 1  
22:18.198 [main] INFO com.javacode2018.chat04.Demo1Test - 影响行数：1  
22:18.200 [main] INFO com.javacode2018.chat04.Demo1Test - UserModel(id=11, name=周润发, age=30, salary=50000.0, sex=1)

上面输出中执行了2条sql，先执行的插入，然后执行了一个查询获取自增值id，最后一行输出的id为11.

去db中看一下，如下：

mysql> SELECT \* FROM t\_user order by id desc limit 1;  
+----+-----------+-----+----------+-----+  
| id | name | age | salary | sex |  
+----+-----------+-----+----------+-----+  
| 11 | 周润发 | 30 | 50000.00 | 1 |  
+----+-----------+-----+----------+-----+  
1 row in set (0.00 sec)

#### 方式2：插入前查询获取主键

##### 用法

这个方式和上面介绍的jdbc的第3种方式一样，会在插入之前先通过一个查询获取主键的值然后填充给指定的属性，然后在执行插入，mapper xml配置如下：

<insert id="insertUser3" parameterType="com.javacode2018.chat04.demo1.model.UserModel">  
 <selectKey keyProperty="id" order="BEFORE" resultType="long">  
 <![CDATA[ 获取主键的select语句 ]]>  
 </selectKey>  
 <![CDATA[  
 INSERT INTO t\_user (name,age,salary,sex) VALUES (#{name},#{age},#{salary},#{sex})  
 ]]>  
</insert>

关键代码是selectKey元素包含的部分，这个元素内部可以包含一个sql，这个sql可以在插入之前或者插入之后运行（之前还是之后通过order属性配置），然后会将sql运行的结果设置给keyProperty指定的属性，selectKey元素有3个属性需要指定：

* keyProperty：参数对象中的属性名称，最后插入成功之后，mybatis会通过反射将自增值设置给keyProperty指定的这个属性
* order：指定selectKey元素中的sql是在插入之前运行还是插入之后运行，可选值（BEFORE|AFTER），这种方式中我们选择BEFORE
* resultType：keyProperty指定的属性对应的类型，如上面的id对应的类型是java.lang.Long，我们直接写的是别名long

##### 案例

这个案例我就不写了，大家可以拿oracle的序列去练习一下这个案例。

#### 源码

mybatis处理自动生产主键值的代码，主要看下面这个接口：

org.apache.ibatis.executor.keygen.KeyGenerator

看一下这个接口的定义：

public interface KeyGenerator {  
  
 void processBefore(Executor executor, MappedStatement ms, Statement stmt, Object parameter);  
  
 void processAfter(Executor executor, MappedStatement ms, Statement stmt, Object parameter);  
  
}

有2个方法，根据方法名称就可以知道，一个是插入sql执行之前调用的，一个是之后调用的，通过这2个方法mybatis完成了获取主键的功能。

这个接口默认有3个实现类：

org.apache.ibatis.executor.keygen.Jdbc3KeyGenerator  
org.apache.ibatis.executor.keygen.SelectKeyGenerator  
org.apache.ibatis.executor.keygen.NoKeyGenerator

mybatis中获取主键的第一种方式就是在Jdbc3KeyGenerator类中实现的，其他2种方式是在第2个类中实现的，大家可以去看一下代码，设置断点感受一下，第3个类2个方法是空实现。

### MyBatis系列

1. MyBatis系列第1篇：MyBatis未出世之前我们那些痛苦的经历
2. MyBatis系列第2篇：入门篇，带你感受一下mybatis独特的魅力！
3. MyBatis系列第3篇：Mybatis使用详解（1）
4. MyBatis系列第4篇：Mybatis使用详解（2）
5. Mybatis系列第5篇：Mapper接口多种方式传参详解、原理、源码解析

## Mybatis系列第7篇：各种查询详解

### 主要内容

* 单表查询3种方式详解
* 一对一关联查询（4种方式）详解
* 一对多查询（2种方式）详解
* 综合案例
* 总结
* 建议
* 源码

### 建库建表

创建一个db:javacode2018

4张表:

t\_user(用户表)

t\_goods(商品表)

t\_order(订单表)

t*order*detail(订单明细表)

表之间的关系：

t*order和t*user是一对一的关系，一条订单关联一个用户记录

t*order和t*order\_detail是一对多关系，每个订单中可能包含多个子订单，每个子订单对应一个商品

DROP DATABASE IF EXISTS `javacode2018`;  
CREATE DATABASE `javacode2018`;  
  
USE `javacode2018`;  
  
DROP TABLE IF EXISTS t\_user;  
CREATE TABLE t\_user(  
 id int AUTO\_INCREMENT PRIMARY KEY COMMENT '用户id',  
 name VARCHAR(32) NOT NULL DEFAULT '' COMMENT '用户名'  
) COMMENT '用户表';  
INSERT INTO t\_user VALUES (1,'张学友'),(2,'马士兵老师');  
  
DROP TABLE IF EXISTS t\_goods;  
CREATE TABLE t\_goods(  
 id int AUTO\_INCREMENT PRIMARY KEY COMMENT '商品id',  
 name VARCHAR(32) NOT NULL DEFAULT '' COMMENT '商品名称',  
 price DECIMAL(10,2) NOT NULL DEFAULT 0 COMMENT '商品价格'  
) COMMENT '商品信息表';  
INSERT INTO t\_goods VALUES (1,'Mybatis系列',8.88),(2,'maven高手系列',16.66);  
  
DROP TABLE IF EXISTS t\_order;  
CREATE TABLE t\_order(  
 id int AUTO\_INCREMENT PRIMARY KEY COMMENT '订单id',  
 user\_id INT NOT NULL DEFAULT 0 COMMENT '用户id，来源于t\_user.id',  
 create\_time BIGINT NOT NULL DEFAULT 0 COMMENT '订单创建时间(时间戳，秒)',  
 up\_time BIGINT NOT NULL DEFAULT 0 COMMENT '订单最后修改时间(时间戳，秒)'  
) COMMENT '订单表';  
INSERT INTO t\_order VALUES (1,2,unix\_timestamp(now()),unix\_timestamp(now())),(2,1,unix\_timestamp(now()),unix\_timestamp(now()));  
  
DROP TABLE IF EXISTS t\_order\_detail;  
CREATE TABLE t\_order\_detail(  
 id int AUTO\_INCREMENT PRIMARY KEY COMMENT '订单明细id',  
 order\_id INT NOT NULL DEFAULT 0 COMMENT '订单id，来源于t\_order.id',  
 goods\_id INT NOT NULL DEFAULT 0 COMMENT '商品id，来源于t\_goods.id',  
 num INT NOT NULL DEFAULT 0 COMMENT '商品数量',  
 total\_price DECIMAL(12,2) NOT NULL DEFAULT 0 COMMENT '商品总金额'  
) COMMENT '订单表';  
INSERT INTO t\_order\_detail VALUES (1,1,1,2,17.76),(2,1,1,1,16.66),(3,2,1,1,8.88);  
  
select \* from t\_user;  
select \* from t\_goods;  
select \* from t\_order;  
select \* from t\_order\_detail;

### 单表查询(3种方式)

#### 需求

需要按照订单id查询订单信息。

#### 方式1

##### 创建每个表对应的Model

db中表的字段是采用下划线分割的，model中我们是采用骆驼命名法来命名的，如OrderModel:

package com.javacode2018.chat05.demo1.model;  
  
import lombok.\*;  
  
import java.util.List;  
  
@Getter  
@Setter  
@Builder  
@ToString  
@NoArgsConstructor  
@AllArgsConstructor  
public class OrderModel {  
 private Integer id;  
 private Integer userId;  
 private Long createTime;  
 private Long upTime;  
}

其他几个Model也类似。

##### Mapper xml

<select id="getById" resultType="com.javacode2018.chat05.demo1.model.OrderModel">  
 <![CDATA[  
 SELECT a.id,a.user\_id as userId,a.create\_time createTime,a.up\_time upTime FROM t\_order a WHERE a.id = #{value}  
 ]]>  
</select>

注意上面的resultType，标识结果的类型。

##### Mapper接口方法

OrderModel getById(int id);

##### mybatis全局配置文件

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE configuration  
 PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-config.dtd">  
<configuration>  
 <!-- 引入外部jdbc配置 -->  
 <properties resource="jdbc.properties"/>  
 <!-- 环境配置，可以配置多个环境 -->  
 <environments default="demo4">  
 <environment id="demo4">  
 <!-- 事务管理器工厂配置 -->  
 <transactionManager type="JDBC"/>  
 <!-- 数据源工厂配置，使用工厂来创建数据源 -->  
 <dataSource type="POOLED">  
 <property name="driver" value="${jdbc.driver}"/>  
 <property name="url" value="${jdbc.url}"/>  
 <property name="username" value="${jdbc.username}"/>  
 <property name="password" value="${jdbc.password}"/>  
 </dataSource>  
 </environment>  
 </environments>  
  
 <mappers>  
 <mapper resource="demo1/mapper/UserMapper.xml" />  
 <mapper resource="demo1/mapper/GoodsMapper.xml" />  
 <mapper resource="demo1/mapper/OrderMapper.xml" />  
 <mapper resource="demo1/mapper/OrderDetailMapper.xml" />  
 </mappers>  
</configuration>

##### 测试用例

com.javacode2018.chat05.demo1.Demo1Test#getById  
  
@Before  
public void before() throws IOException {  
 //指定mybatis全局配置文件  
 String resource = "demo1/mybatis-config.xml";  
 //读取全局配置文件  
 InputStream inputStream = Resources.getResourceAsStream(resource);  
 //构建SqlSessionFactory对象  
 SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
 this.sqlSessionFactory = sqlSessionFactory;  
}  
  
@Test  
public void getById() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);  
 OrderModel orderModel = mapper.getById(1);  
 log.info("{}", orderModel);  
 }  
}

##### 运行输出

35:59.211 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById - ==> Preparing: SELECT a.id,a.user\_id as userId,a.create\_time createTime,a.up\_time upTime FROM t\_order a WHERE a.id = ?   
35:59.239 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById - ==> Parameters: 1(Integer)  
35:59.258 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById - <== Total: 1  
35:59.258 [main] INFO c.j.chat05.demo1.Demo1Test - OrderModel(id=1, userId=2, createTime=1577947790, upTime=1577947790)

##### 原理

sql中我们使用了别名，将t\_order中的字段转换成了和OrderModel中字段一样的名称，最后mybatis内部会通过反射，将查询结果按照名称到OrderModel中查找同名的字段，然后进行赋值。

#### 方式2

若我们项目中表对应的Model中的字段都是采用骆驼命名法，mybatis中可以进行一些配置，可以使表中的字段和对应Model中骆驼命名法的字段进行自动映射。

需要在mybatis全局配置文件中加入下面配置：

<settings>  
 <!-- 是否开启自动驼峰命名规则映射，及从xx\_yy映射到xxYy -->  
 <setting name="mapUnderscoreToCamelCase" value="true"/>  
</settings>

##### Mapper xml

<select id="getById1" resultType="com.javacode2018.chat05.demo1.model.OrderModel">  
 <![CDATA[  
 SELECT a.id,a.user\_id,a.create\_time,a.up\_time FROM t\_order a WHERE a.id = #{value}  
 ]]>  
</select>

注意上面的sql，我们没有写别名了，由于我们开启了自动骆驼命名映射，所以查询结果会按照下面的关系进行自动映射：

|  |  |
| --- | --- |
| sql对应的字段 | OrderModel中的字段 |
| id | id |
| user\_id | userId |
| create\_time | createTime |
| up\_time | upTime |

##### Mapper接口

OrderModel getById1(int id);

##### 测试用例

com.javacode2018.chat05.demo1.Demo1Test#getById1  
  
@Test  
public void getById1() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);  
 OrderModel orderModel = mapper.getById1(1);  
 log.info("{}", orderModel);  
 }  
}

##### 运行输出

59:44.884 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - ==> Preparing: SELECT a.id,a.user\_id,a.create\_time,a.up\_time FROM t\_order a WHERE a.id = ?   
59:44.917 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - ==> Parameters: 1(Integer)  
59:44.935 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - <== Total: 1  
59:44.935 [main] INFO c.j.chat05.demo1.Demo1Test - OrderModel(id=1, userId=2, createTime=1577947790, upTime=1577947790)

输出中可以看出，sql中的字段是下划线的方式，OrderModel中的字段是骆驼命名法，结果也自动装配成功，这个就是开启mapUnderscoreToCamelCase产生的效果。

#### 方式3

mapper xml中有个更强大的元素resultMap，通过这个元素可以定义查询结果的映射关系。

##### Mapper xml

<resultMap id="orderModelMap2" type="com.javacode2018.chat05.demo1.model.OrderModel">  
 <id column="id" property="id" />  
 <result column="user\_id" property="userId" />  
 <result column="create\_time" property="createTime" />  
 <result column="up\_time" property="upTime" />  
</resultMap>  
  
<select id="getById2" resultMap="orderModelMap2">  
 <![CDATA[  
 SELECT a.id,a.user\_id,a.create\_time,a.up\_time FROM t\_order a WHERE a.id = #{value}  
 ]]>  
</select>

上面resultMap有2个元素需要指定：

* id：resultMap标识
* type：将结果封装成什么类型，此处我们需要将结果分装为OrderModel

注意上面的select元素，有个resultMap，标识查询结果使用哪个resultMap进行映射，此处我们使用的是orderModelMap2，所以查询结果会按照orderModelMap2关联的resultMap进行映射。

##### Mapper接口

OrderModel getById2(int id);

##### 测试用例

com.javacode2018.chat05.demo1.Demo1Test#getById2  
  
@Test  
public void getById2() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);  
 OrderModel orderModel = mapper.getById2(1);  
 log.info("{}", orderModel);  
 }  
}

运行输出

14:12.518 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - ==> Preparing: SELECT a.id,a.user\_id,a.create\_time,a.up\_time FROM t\_order a WHERE a.id = ?   
14:12.546 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - ==> Parameters: 1(Integer)  
14:12.564 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - <== Total: 1  
14:12.564 [main] INFO c.j.chat05.demo1.Demo1Test - OrderModel(id=1, userId=2, createTime=1577947790, upTime=1577947790)

### 一对一关联查询(4种方式)

#### 需求

通过订单id查询订单的时候，将订单关联的用户信息也返回。

我们修改一下OrderModel代码，内部添加一个UserModel，如下：

package com.javacode2018.chat05.demo2.model;  
  
import lombok.\*;  
  
import java.util.List;  
  
@Getter  
@Setter  
@Builder  
@ToString  
@NoArgsConstructor  
@AllArgsConstructor  
public class OrderModel {  
 private Integer id;  
 private Integer userId;  
 private Long createTime;  
 private Long upTime;  
  
 //下单用户信息  
 private UserModel userModel;  
}

UserModel内容：

package com.javacode2018.chat05.demo2.model;  
  
import lombok.\*;  
  
@Getter  
@Setter  
@Builder  
@ToString  
@NoArgsConstructor  
@AllArgsConstructor  
public class UserModel {  
 private Integer id;  
 private String name;  
}

#### 方式1

##### Mapper xml

<resultMap id="orderModelMap1" type="com.javacode2018.chat05.demo2.model.OrderModel">  
 <id column="id" property="id" />  
 <result column="user\_id" property="userId"/>  
 <result column="create\_time" property="createTime"/>  
 <result column="up\_time" property="upTime"/>  
 <result column="user\_id" property="userModel.id"/>  
 <result column="name" property="userModel.name"/>  
</resultMap>  
  
<select id="getById1" resultMap="orderModelMap1">  
 <![CDATA[  
 SELECT  
 a.id,  
 a.user\_id,  
 a.create\_time,  
 a.up\_time,  
 b.name  
 FROM  
 t\_order a,  
 t\_user b  
 WHERE  
 a.user\_id = b.id  
 AND a.id = #{value}  
 ]]>  
</select>

注意重点在于上面的这两行：

<result column="user\_id" property="userModel.id"/>  
<result column="name" property="userModel.name"/>

这个地方使用到了级联赋值，多级之间用.进行引用，此处我们只有一级，可以有很多级。

##### Mapper 接口

OrderModel getById1(int id);

##### 测试用例

com.javacode2018.chat05.demo2.Demo2Test#getById1  
  
@Before  
public void before() throws IOException {  
 //指定mybatis全局配置文件  
 String resource = "demo2/mybatis-config.xml";  
 //读取全局配置文件  
 InputStream inputStream = Resources.getResourceAsStream(resource);  
 //构建SqlSessionFactory对象  
 SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
 this.sqlSessionFactory = sqlSessionFactory;  
}  
  
@Test  
public void getById1() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);  
 OrderModel orderModel = mapper.getById1(1);  
 log.info("{}", orderModel);  
 }  
}

##### 运行输出

24:20.811 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - ==> Preparing: SELECT a.id, a.user\_id, a.create\_time, a.up\_time, b.name FROM t\_order a, t\_user b WHERE a.user\_id = b.id AND a.id = ?   
24:20.843 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - ==> Parameters: 1(Integer)  
24:20.861 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - <== Total: 1  
24:20.861 [main] INFO c.j.chat05.demo2.Demo2Test - OrderModel(id=1, userId=2, createTime=1577947790, upTime=1577947790, userModel=UserModel(id=2, name=马士兵老师))

#### 方式2

这次我们需要使用mapper xml中另外一个元素association，这个元素可以配置关联对象的映射关系，看示例。

##### Mapper xml

<resultMap id="orderModelMap2" type="com.javacode2018.chat05.demo2.model.OrderModel">  
 <id column="id" property="id" />  
 <result column="user\_id" property="userId"/>  
 <result column="create\_time" property="createTime"/>  
 <result column="up\_time" property="upTime"/>  
 <association property="userModel">  
 <id column="user\_id" property="id"/>  
 <result column="name" property="name" />  
 </association>  
</resultMap>  
  
<select id="getById2" resultMap="orderModelMap2">  
 <![CDATA[  
 SELECT  
 a.id,  
 a.user\_id,  
 a.create\_time,  
 a.up\_time,  
 b.name  
 FROM  
 t\_order a,  
 t\_user b  
 WHERE  
 a.user\_id = b.id  
 AND a.id = #{value}  
 ]]>  
</select>

注意上面下面这部分代码：

<association property="userModel">  
 <id column="user\_id" property="id"/>  
 <result column="name" property="name" />  
</association>

注意上面的property属性，这个就是配置sql查询结果和OrderModel.userModel对象的映射关系，将user\_id和userModel中的id进行映射,name和userModel中的name进行映射。

##### Mapper接口

OrderModel getById2(int id);

##### 测试用例

@Test  
public void getById2() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);  
 OrderModel orderModel = mapper.getById2(1);  
 log.info("{}", orderModel);  
 }  
}

##### 运行结果

51:44.896 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - ==> Preparing: SELECT a.id, a.user\_id, a.create\_time, a.up\_time, b.name FROM t\_order a, t\_user b WHERE a.user\_id = b.id AND a.id = ?   
51:44.925 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - ==> Parameters: 1(Integer)  
51:44.941 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - <== Total: 1  
51:44.942 [main] INFO c.j.chat05.demo2.Demo2Test - OrderModel(id=1, userId=2, createTime=1577947790, upTime=1577947790, userModel=UserModel(id=2, name=马士兵老师))

从结果的最后一行可以看出，所有字段的值映射都是ok的。

#### 方式3

先按照订单id查询订单数据，然后在通过订单中user\_id去用户表查询用户数据，通过两次查询，组合成目标结果，mybatis已经内置了这种操作，如下。

##### UserMapper.xml

我们先定义一个通过用户id查询用户信息的select元素，如下

<select id="getById" resultType="com.javacode2018.chat05.demo2.model.UserModel">  
 <![CDATA[  
 SELECT id,name FROM t\_user where id = #{value}  
 ]]>  
</select>

##### OrderModel.xml

<resultMap id="orderModelMap3" type="com.javacode2018.chat05.demo2.model.OrderModel">  
 <id column="id" property="id" />  
 <result column="user\_id" property="userId"/>  
 <result column="create\_time" property="createTime"/>  
 <result column="up\_time" property="upTime"/>  
 <association property="userModel" select="com.javacode2018.chat05.demo2.mapper.UserMapper.getById" column="user\_id" />  
</resultMap>  
  
<select id="getById3" resultMap="orderModelMap3">  
 <![CDATA[  
 SELECT  
 a.id,  
 a.user\_id,  
 a.create\_time,  
 a.up\_time  
 FROM  
 t\_order a  
 WHERE  
 a.id = #{value}  
 ]]>  
</select>

OrderModel.userModel属性的值来在于另外一个查询，这个查询是通过association元素的select属性指定的，此处使用的是

com.javacode2018.chat05.demo2.mapper.UserMapper.getById

这个查询是有条件的，条件通过association的column进行传递的，此处传递的是getById3查询结果中的user\_id字段。

##### Mapper接口

OrderModel getById3(int id);

##### 测试用例

com.javacode2018.chat05.demo2.Demo2Test#getById3  
@Test  
public void getById3() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);  
 OrderModel orderModel = mapper.getById3(1);  
 log.info("{}", orderModel);  
 }  
}

##### 运行输出

07:12.569 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById3 - ==> Preparing: SELECT a.id, a.user\_id, a.create\_time, a.up\_time FROM t\_order a WHERE a.id = ?   
07:12.600 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById3 - ==> Parameters: 1(Integer)  
07:12.619 [main] DEBUG c.j.c.d.mapper.UserMapper.getById - ====> Preparing: SELECT id,name FROM t\_user where id = ?   
07:12.620 [main] DEBUG c.j.c.d.mapper.UserMapper.getById - ====> Parameters: 2(Integer)  
07:12.625 [main] DEBUG c.j.c.d.mapper.UserMapper.getById - <==== Total: 1  
07:12.625 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById3 - <== Total: 1  
07:12.625 [main] INFO c.j.chat05.demo2.Demo2Test - OrderModel(id=1, userId=2, createTime=1577947790, upTime=1577947790, userModel=UserModel(id=2, name=马士兵老师))

从输出中可以看出有2次查询，先按照订单id查询订单，然后通过订单记录中用户id去用户表查询用户信息，最终执行了2次查询。

方式4

方式3中给第二个查询传递了一个参数，如果需要给第二个查询传递多个参数怎么办呢？可以这么写

<association property="属性" select="查询对应的select的id" column="{key1=父查询字段1,key2=父查询字段2,key3=父查询字段3}" />

这种相当于给子查询传递了一个map，子查询中 需要用过map的key获取对应的条件，看案例：

OrderMapper.xml

<resultMap id="orderModelMap4" type="com.javacode2018.chat05.demo2.model.OrderModel">  
 <id column="id" property="id" />  
 <result column="user\_id" property="userId"/>  
 <result column="create\_time" property="createTime"/>  
 <result column="up\_time" property="upTime"/>  
 <association property="userModel" select="com.javacode2018.chat05.demo2.mapper.UserMapper.getById1" column="{uid1=user\_id,uid2=create\_time}" />  
</resultMap>  
  
<select id="getById4" resultMap="orderModelMap4">  
 <![CDATA[  
 SELECT  
 a.id,  
 a.user\_id,  
 a.create\_time,  
 a.up\_time  
 FROM  
 t\_order a  
 WHERE  
 a.id = #{value}  
 ]]>  
</select>

UserMapper.xml

<select id="getById1" resultType="com.javacode2018.chat05.demo2.model.UserModel">  
 <![CDATA[  
 SELECT id,name FROM t\_user where id = #{uid1} and id = #{uid2}  
 ]]>  
</select>

Mapper接口

OrderModel getById4(int id);

测试用例

com.javacode2018.chat05.demo2.Demo2Test#getById4  
@Test  
public void getById4() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);  
 OrderModel orderModel = mapper.getById4(1);  
 log.info("{}", orderModel);  
 }  
}

运行输出

19:59.881 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById4 - ==> Preparing: SELECT a.id, a.user\_id, a.create\_time, a.up\_time FROM t\_order a WHERE a.id = ?   
19:59.914 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById4 - ==> Parameters: 1(Integer)  
19:59.934 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - ====> Preparing: SELECT id,name FROM t\_user where id = ? and id = ?   
19:59.934 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - ====> Parameters: 2(Integer), 1577947790(Long)  
19:59.939 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - <==== Total: 0  
19:59.939 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById4 - <== Total: 1  
19:59.939 [main] INFO c.j.chat05.demo2.Demo2Test - OrderModel(id=1, userId=2, createTime=1577947790, upTime=1577947790, userModel=null)

输出中看一下第二个查询的条件，传过来的是第一个查询的user\_id和create\_time。

### 一对多查询(2种方式)

#### 需求

根据订单id查询出订单信息，并且查询出订单明细列表。

先修改一下OrderModel代码，如下：

package com.javacode2018.chat05.demo3.model;  
  
import lombok.\*;  
  
import java.util.List;  
  
@Getter  
@Setter  
@Builder  
@ToString  
@NoArgsConstructor  
@AllArgsConstructor  
public class OrderModel {  
 private Integer id;  
 private Integer userId;  
 private Long createTime;  
 private Long upTime;  
 //订单详情列表  
 private List<OrderDetailModel> orderDetailModelList;  
}

OrderModel中添加了一个集合orderDetailModelList用来存放订单详情列表。

#### 方式1

##### OrderMapper.xml

<resultMap id="orderModelMap1" type="com.javacode2018.chat05.demo3.model.OrderModel">  
 <id column="id" property="id"/>  
 <result column="user\_id" property="userId"/>  
 <result column="create\_time" property="createTime"/>  
 <result column="up\_time" property="upTime"/>  
 <collection property="orderDetailModelList" ofType="com.javacode2018.chat05.demo3.model.OrderDetailModel">  
 <id column="orderDetailId" property="id"/>  
 <result column="order\_id" property="orderId"/>  
 <result column="goods\_id" property="goodsId"/>  
 <result column="num" property="num"/>  
 <result column="total\_price" property="totalPrice"/>  
 </collection>  
</resultMap>  
  
<select id="getById1" resultMap="orderModelMap1">  
 <![CDATA[  
 SELECT  
 a.id ,  
 a.user\_id,  
 a.create\_time,  
 a.up\_time,  
 b.id orderDetailId,  
 b.order\_id,  
 b.goods\_id,  
 b.num,  
 b.total\_price  
 FROM  
 t\_order a,  
 t\_order\_detail b  
 WHERE  
 a.id = b.order\_id  
 AND a.id = #{value}  
 ]]>  
</select>

注意上面的getById1中的sql，这个sql中使用到了t\_order和t\_order\_detail连接查询，这个查询会返回多条结果，但是最后结果按照orderModelMap1进行映射，最后只会返回一个OrderModel对象，关键在于collection元素，这个元素用来定义集合中元素的映射关系，有2个属性需要注意：

* property：对应的属性名称
* ofType：集合中元素的类型，此处是OrderDetailModel

原理是这样的，注意orderModelMap1中有个

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

查询出来的结果会按照这个配置中指定的column进行分组，即按照订单id进行分组，每个订单对应多个订单明细，订单明细会按照collection的配置映射为ofType元素指定的对象。

实际resultMap元素中的id元素可以使用result元素代替，只是用id可以提升性能，mybatis可以通过id元素配置的列的值判断唯一一条记录，如果我们使用result元素，那么判断是否是同一条记录的时候，需要通过所有列去判断了，所以通过id可以提升性能，使用id元素在一对多中可以提升性能，在单表查询中使用id元素还是result元素，性能都是一样的。

##### Mapper接口

OrderModel getById1(Integer id);

##### 测试用例

com.javacode2018.chat05.demo3.Demo3Test#getById1  
@Before  
public void before() throws IOException {  
 //指定mybatis全局配置文件  
 String resource = "demo3/mybatis-config.xml";  
 //读取全局配置文件  
 InputStream inputStream = Resources.getResourceAsStream(resource);  
 //构建SqlSessionFactory对象  
 SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
 this.sqlSessionFactory = sqlSessionFactory;  
}  
  
@Test  
public void getById1() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);  
 Integer id = 1;  
 OrderModel orderModel = mapper.getById1(id);  
 log.info("{}", orderModel);  
 }  
}

##### 运行输出

03:52.092 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - ==> Preparing: SELECT a.id , a.user\_id, a.create\_time, a.up\_time, b.id orderDetailId, b.order\_id, b.goods\_id, b.num, b.total\_price FROM t\_order a, t\_order\_detail b WHERE a.id = b.order\_id AND a.id = ?   
03:52.124 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - ==> Parameters: 1(Integer)  
03:52.148 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - <== Total: 2  
03:52.148 [main] INFO c.j.chat05.demo3.Demo3Test - OrderModel(id=1, userId=2, createTime=1577947790, upTime=1577947790, orderDetailModelList=[OrderDetailModel(id=1, orderId=1, goodsId=1, num=2, totalPrice=17.76), OrderDetailModel(id=2, orderId=1, goodsId=1, num=1, totalPrice=16.66)])

注意最后一条输出，和期望的结果一致。

#### 方式2

通过2次查询，然后对结果进行分装，先通过订单id查询订单信息，然后通过订单id查询订单明细列表，然后封装结果。mybatis中默认支持这么玩，还是通过collection元素来实现的。

##### OrderDetailMapper.xml

<select id="getListByOrderId1" resultType="com.javacode2018.chat05.demo3.model.OrderDetailModel" parameterType="int">  
 <![CDATA[  
 SELECT  
 a.id,  
 a.order\_id AS orderId,  
 a.goods\_id AS goodsId,  
 a.num,  
 a.total\_price AS totalPrice  
 FROM  
 t\_order\_detail a  
 WHERE  
 a.order\_id = #{value}  
 ]]>  
</select>

##### OrderMapper.xml

<resultMap id="orderModelMap2" type="com.javacode2018.chat05.demo3.model.OrderModel">  
 <id column="id" property="id"/>  
 <result column="user\_id" property="userId"/>  
 <result column="create\_time" property="createTime"/>  
 <result column="up\_time" property="upTime"/>  
 <collection property="orderDetailModelList" select="com.javacode2018.chat05.demo3.mapper.OrderDetailMapper.getListByOrderId1" column="id"/>  
</resultMap>  
  
<select id="getById2" resultMap="orderModelMap2">  
 <![CDATA[  
 SELECT  
 a.id ,  
 a.user\_id,  
 a.create\_time,  
 a.up\_time  
 FROM  
 t\_order a  
 WHERE  
 a.id = #{value}  
 ]]>  
</select>

重点在于下面这句配置：

<collection property="orderDetailModelList" select="com.javacode2018.chat05.demo3.mapper.OrderDetailMapper.getListByOrderId1" column="id"/>

表示orderDetailModelList属性的值通过select属性指定的查询获取，即：

com.javacode2018.chat05.demo3.mapper.OrderDetailMapper.getListByOrderId1

查询参数是通过column属性指定的，此处使用getById2 sql中的id作为条件，即订单id。

##### Mapper接口

OrderModel getById2(int id);

##### 测试用例

com.javacode2018.chat05.demo3.Demo3Test#getById2  
  
@Test  
public void getById2() {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);  
 OrderModel orderModel = mapper.getById2(1);  
 log.info("{}", orderModel);  
 }  
}

##### 运行输出

10:07.087 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - ==> Preparing: SELECT a.id , a.user\_id, a.create\_time, a.up\_time FROM t\_order a WHERE a.id = ?   
10:07.117 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - ==> Parameters: 1(Integer)  
10:07.135 [main] DEBUG c.j.c.d.m.O.getListByOrderId1 - ====> Preparing: SELECT a.id, a.order\_id AS orderId, a.goods\_id AS goodsId, a.num, a.total\_price AS totalPrice FROM t\_order\_detail a WHERE a.order\_id = ?   
10:07.136 [main] DEBUG c.j.c.d.m.O.getListByOrderId1 - ====> Parameters: 1(Integer)  
10:07.141 [main] DEBUG c.j.c.d.m.O.getListByOrderId1 - <==== Total: 2  
10:07.142 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - <== Total: 1  
10:07.142 [main] INFO c.j.chat05.demo3.Demo3Test - OrderModel(id=1, userId=2, createTime=1577947790, upTime=1577947790, orderDetailModelList=[OrderDetailModel(id=1, orderId=1, goodsId=1, num=2, totalPrice=17.76), OrderDetailModel(id=2, orderId=1, goodsId=1, num=1, totalPrice=16.66)])

输出中有2次查询，先通过订单id查询订单信息，然后通过订单id查询订单明细，mybatis内部对结果进行了组装。

### 综合案例

入口

com.javacode2018.chat05.demo4.Demo4Test

这个案例中将上面多种查询混合在一起了，有兴趣的可以去看看，加深理解。

### 总结

1. mybatis全局配置文件中通过mapUnderscoreToCamelCase可以开启sql中的字段和javabean中的骆驼命名法的字段进行自动映射
2. 掌握resultMap元素常见的用法
3. 一对一关联查询使用resultMap->association元素（2种方式）
4. 一对多查询使用resultMap->collection元素（2种方式）
5. resultMap中使用id元素主要在复杂的关联查询中可以提升效率，可以通过这个来判断记录的唯一性，如果没有这个，需要通过所有的result相关的列才能判断记录的唯一性

### 建议

mybatis为我们提供了强大的关联查询，不过个人建议尽量少用，最好是采用单表的方式查询，在程序中通过多次查询，然后自己对结果进行组装。

Model中最好只定义一些和单表字段关联的属性，不要掺杂着其他对象 的引用。

### 案例代码

链接：https://pan.baidu.com/s/1vt-MAX3oJOu9gyxZAhKkbg   
提取码：i8op

测试用例为代码的入口，下面目录中的所有类：

mybatis-series\chat05\src\test\java\com\javacode2018\chat05

### MyBatis系列

1. MyBatis系列第1篇：MyBatis未出世之前我们那些痛苦的经历
2. MyBatis系列第2篇：入门篇，带你感受一下mybatis独特的魅力！
3. MyBatis系列第3篇：Mybatis使用详解（1）
4. MyBatis系列第4篇：Mybatis使用详解（2）
5. Mybatis系列第5篇：Mapper接口多种方式传参详解、原理、源码解析
6. Mybatis系列第6篇：恕我直言，mybatis增删改你未必玩得转!

## Mybatis系列第8篇：自动映射，使用需谨慎

### 案例代码

链接：https://pan.baidu.com/s/1vt-MAX3oJOu9gyxZAhKkbg   
提取码：i8op

本文案例代码入口，配合源码看案例，效果更好。

mybatis-series\chat05\src\test\java\com\javacode2018\chat05\demo7\Demo7Test.java

### 什么是自动映射？

介绍自动映射之前先看一下手动映射，如下：

<resultMap id="orderModelMap1" type="com.javacode2018.chat05.demo7.model.OrderModel">  
 <id column="id" property="id"/>  
 <result column="userId" property="userId" />  
 <result column="createTime" property="createTime" />  
 <result column="upTime" property="upTime" />  
</resultMap>  
  
<select id="getById1" resultMap="orderModelMap1">  
 <![CDATA[  
 SELECT  
 a.id,  
 a.user\_id userId,  
 a.create\_time createTime,  
 a.up\_time upTime  
 FROM  
 t\_order a  
 WHERE  
 a.id = #{value}  
 ]]>  
</select>

注意上面的resultMap元素中有4行配置，如下：

<id column="id" property="id"/>  
<result column="userId" property="userId" />  
<result column="createTime" property="createTime" />  
<result column="upTime" property="upTime" />

这4行代码用于配置sql结果的列和OrderModel对象中字段的映射关系。

大家有没有注意到，映射规则中column和property元素的值都是一样，mybatis中支持自动映射配置，当开启自动映射之后，当sql的列名和Model中的字段名称是一样的时候（不区分大小写），mybatis内部会进行自动映射，不需要我们手动去写上面的4行映射规则。

下面我们将上面的示例改成自动映射的方式，如下：

<resultMap id="orderModelMap2" type="com.javacode2018.chat05.demo7.model.OrderModel" autoMapping="true">  
</resultMap>  
  
<select id="getById2" resultMap="orderModelMap2">  
 <![CDATA[  
 SELECT  
 a.id,  
 a.user\_id userId,  
 a.create\_time createTime,  
 a.up\_time upTime  
 FROM  
 t\_order a  
 WHERE  
 a.id = #{value}  
 ]]>  
</select>

**注意上面的resultMap中的autoMapping属性，是否开启自动映射，我们设置为true，这样mybatis会自动按照列名和Model中同名的字段进行映射赋值。**

上面两个配置最后查询结果是一样的，都会将查询结果对应的4个字段的值自动赋值给OrderModel中同名的属性。

### 自动映射开关

mybatis中自动映射主要有2种配置，一种是全局的配置，对应用中所有的resultMap起效，这个是在mybatis配置文件中进行设置的；另外一种是通过resultMap的autoMapping属性进行配置。

mybatis判断某个resultMap是否开启自动映射配置的时候，会先查找自身的autoMapping属性，如果这个属性设置值了，就直接用这个属性的值，如果resultMap元素的autoMapping属性没有配置，则走全局配置的自动映射规则。

下面我们来详解介绍一下这款的内容。

### mybatis自动映射全局配置

在mybatis全局配置文件中加入下面配置：

<settings>  
 <setting name="autoMappingBehavior" value="自动映射规则"/>  
</settings>

autoMappingBehavior值来源于枚举：org.apache.ibatis.session.AutoMappingBehavior，源码：

public enum AutoMappingBehavior {  
  
 /\*\*  
 \* Disables auto-mapping.  
 \*/  
 NONE,  
  
 /\*\*  
 \* Will only auto-map results with no nested result mappings defined inside.  
 \*/  
 PARTIAL,  
  
 /\*\*  
 \* Will auto-map result mappings of any complexity (containing nested or otherwise).  
 \*/  
 FULL  
}

* NONE：关闭全局映射开关
* PARTIAL：对除在内部定义了嵌套结果映射（也就是连接的属性）以外的属性进行映射，这个也是默认值。
* FULL：自动映射所有属性。

小提示：settings元素中有很多配置，这些配置最后都会被解析成org.apache.ibatis.session.Configuration的属性，源码位于org.apache.ibatis.builder.xml.XMLConfigBuilder#settingsElement方法中。

下面我们来演示一下autoMappingBehavior每种配置的效果。

#### NONE

##### mybatis-config.xml加入配置

<settings>  
 <!-- 关闭自动映射开关 -->  
 <setting name="autoMappingBehavior" value="NONE"/>  
</settings>

##### OrderMapper.xml

<resultMap id="orderModelMap4" type="com.javacode2018.chat05.demo7.model.OrderModel">  
</resultMap>  
  
<select id="getById4" resultMap="orderModelMap4">  
 <![CDATA[  
 SELECT  
 a.id,  
 a.user\_id userId,  
 a.create\_time createTime,  
 a.up\_time upTime  
 FROM  
 t\_order a  
 WHERE  
 a.id = #{value}  
 ]]>  
</select>

##### OrderMapper.java加入

OrderModel getById4(int id);

##### 测试用例

com.javacode2018.chat05.demo7.Demo7Test#getById4  
  
@Test  
public void getById4() throws IOException {  
 this.before("demo7/mybatis-config1.xml");  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);  
 OrderModel orderModel = mapper.getById4(2);  
 log.info("{}", orderModel);  
 }  
}

##### 运行结果

21:58.821 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById4 - ==> Preparing: SELECT a.id, a.user\_id userId, a.create\_time createTime, a.up\_time upTime FROM t\_order a WHERE a.id = ?   
21:58.850 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById4 - ==> Parameters: 2(Integer)  
21:58.868 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById4 - <== Total: 1  
21:58.868 [main] INFO c.j.chat05.demo7.Demo7Test - null

从输出中可以看到最后一样输出结果为null，sql实际上返回的是有结果的，但是结果映射的时候返回的是空。

##### 结果解释

由于mybatis全局配置中将autoMappingBehavior的值置为了NONE,表示全局自动映射被关闭了，而resultMapper中的orderModelMap4没有配置autoMapping属性，所以最终这个查询结果不会自动映射，所以最后查询结果为null。

#### PARTIAL

对除在内部定义了嵌套结果映射（也就是连接的属性）以外的属性进行映射，这个也是autoMappingBehavior的默认值。

##### mybatis-config.xml加入配置

<settings>  
 <!-- 对除在内部定义了嵌套结果映射（也就是连接的属性）以外的属性进行映射，这个也是autoMappingBehavior的默认值。 -->  
 <setting name="autoMappingBehavior" value="PARTIAL"/>  
</settings>

##### OrderMapper.xml

<resultMap id="orderModelMap5" type="com.javacode2018.chat05.demo7.model.OrderModel">  
</resultMap>  
  
<select id="getById5" resultMap="orderModelMap5">  
 <![CDATA[  
 SELECT  
 a.id,  
 a.user\_id userId,  
 a.create\_time createTime,  
 a.up\_time upTime  
 FROM  
 t\_order a  
 WHERE  
 a.id = #{value}  
 ]]>  
</select>

##### OrderMapper.java加入

OrderModel getById5(int id);

##### 测试用例

com.javacode2018.chat05.demo7.Demo7Test#getById5  
  
@Test  
public void getById5() throws IOException {  
 this.before("demo7/mybatis-config2.xml");  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);  
 OrderModel orderModel = mapper.getById5(2);  
 log.info("{}", orderModel);  
 }  
}

##### 运行结果

28:32.612 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById5 - ==> Preparing: SELECT a.id, a.user\_id userId, a.create\_time createTime, a.up\_time upTime FROM t\_order a WHERE a.id = ?   
28:32.648 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById5 - ==> Parameters: 2(Integer)  
28:32.664 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById5 - <== Total: 1  
28:32.665 [main] INFO c.j.chat05.demo7.Demo7Test - OrderModel(id=2, userId=1, createTime=1577947790, upTime=1577947790, userModel=null)

OrderModel中的4个属性被自动映射成功了。

##### 结果解释

orderModelMap5中没有指定autoMapping属性，所以自动映射会走全局配置的规则，即PARTIAL，会进行自动映射。

我们再来看看PARTIAL的解释：对除在内部定义了嵌套结果映射（也就是连接的属性）以外的属性进行映射。这句话是什么意思？

有些复杂的查询映射会在resultMap中嵌套一些映射（如：association，collection），当使用PARTIAL的时候，如果有嵌套映射，则这个嵌套映射不会进行自动映射了。

通过订单id查询出订单以及订单用户的信息，sqlmap如下：

<resultMap id="orderModelMap6" type="com.javacode2018.chat05.demo7.model.OrderModel">  
 <association property="userModel">  
 </association>  
</resultMap>  
  
<select id="getById6" resultMap="orderModelMap6">  
 <![CDATA[  
 SELECT  
 a.id,  
 a.user\_id userId,  
 a.create\_time createTime,  
 a.up\_time upTime  
 b.id as user\_id,  
 b.name  
 FROM  
 t\_order a,t\_user b  
 WHERE  
 a.user\_id = b.id  
 AND a.id = #{value}  
 ]]>  
</select>

##### OrderModel.java

package com.javacode2018.chat05.demo7.model;  
  
import lombok.\*;  
  
import java.util.List;  
  
@Getter  
@Setter  
@Builder  
@ToString  
@NoArgsConstructor  
@AllArgsConstructor  
public class OrderModel {  
 private Integer id;  
 private Integer userId;  
 private Long createTime;  
 private Long upTime;  
 private UserModel userModel;  
}

内部有个userModel属性引用用户对象。

##### UserModel.java

package com.javacode2018.chat05.demo7.model;  
  
import lombok.\*;  
  
@Getter  
@Setter  
@Builder  
@ToString  
@NoArgsConstructor  
@AllArgsConstructor  
public class UserModel {  
 private Integer id;  
 private String name;  
}

##### OrderMapper.java中加入

OrderModel getById6(int id);

测试用例

com.javacode2018.chat05.demo7.Demo7Test#getById6  
@Test  
public void getById6() throws IOException {  
 this.before("demo7/mybatis-config2.xml");  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);  
 OrderModel orderModel = mapper.getById6(2);  
 log.info("{}", orderModel);  
 }  
}

##### 运行输出

52:49.037 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById6 - ==> Preparing: SELECT a.id, a.user\_id userId, a.create\_time createTime, a.up\_time upTime, b.id as user\_id, b.name FROM t\_order a,t\_user b WHERE a.user\_id = b.id AND a.id = ?   
52:49.066 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById6 - ==> Parameters: 2(Integer)  
52:49.087 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById6 - <== Total: 1  
52:49.088 [main] INFO c.j.chat05.demo7.Demo7Test - null

sql查询实际上是有一条记录的，但是最后返回的是null，说明没有进行自动映射。

#### FULL

自动映射所有属性。

这次Mapper我们不动，还是下面这样，没有手动指定映射规则。

<resultMap id="orderModelMap6" type="com.javacode2018.chat05.demo7.model.OrderModel">  
 <association property="userModel">  
 </association>  
</resultMap>  
  
<select id="getById6" resultMap="orderModelMap6">  
 <![CDATA[  
 SELECT  
 a.id,  
 a.user\_id userId,  
 a.create\_time createTime,  
 a.up\_time upTime  
 b.id as user\_id,  
 b.name  
 FROM  
 t\_order a,t\_user b  
 WHERE  
 a.user\_id = b.id  
 AND a.id = #{value}  
 ]]>  
</select>

修改一下autoMappingBehavior的值为FULL，看看效果。

##### mybatis配置

<settings>  
 <!-- 自动映射所有属性 -->  
 <setting name="autoMappingBehavior" value="FULL"/>  
</settings>

##### 测试用例

com.javacode2018.chat05.demo7.Demo7Test#getById6\_0  
  
@Test  
public void getById6\_0() throws IOException {  
 this.before("demo7/mybatis-config3.xml");  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);  
 OrderModel orderModel = mapper.getById6(2);  
 log.info("{}", orderModel);  
 }  
}

##### 运行输出

56:05.127 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById6 - ==> Preparing: SELECT a.id, a.user\_id userId, a.create\_time createTime, a.up\_time upTime, b.id as user\_id, b.name FROM t\_order a,t\_user b WHERE a.user\_id = b.id AND a.id = ?   
56:05.155 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById6 - ==> Parameters: 2(Integer)  
56:05.186 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById6 - <== Total: 1  
56:05.186 [main] INFO c.j.chat05.demo7.Demo7Test - OrderModel(id=2, userId=1, createTime=1577947790, upTime=1577947790, userModel=UserModel(id=2, name=张学友))

输出中可以看到OrderModel所有属性都是有值的，userModel的2个属性也有值，userModel.id是2，我们运行一下sql看看，用户id是多少，如下：

mysql> SELECT a.id, a.user\_id userId, a.create\_time createTime, a.up\_time upTime, b.id as user\_id, b.name FROM t\_order a,t\_user b WHERE a.user\_id = b.id AND a.id = 2;  
+----+--------+------------+------------+---------+-----------+  
| id | userId | createTime | upTime | user\_id | name |  
+----+--------+------------+------------+---------+-----------+  
| 2 | 1 | 1577947790 | 1577947790 | 1 | 张学友 |  
+----+--------+------------+------------+---------+-----------+  
1 row in set (0.00 sec)

user\_id实际上是1，mybatis中按照sql字段和model结果字段同名进行自动映射，所以将订单的id赋值给userModel的id属性了。

此时需要我们orderModelMap6的配置，手动指定一下user\_id和userModel.id的映射规则，如下：

<resultMap id="orderModelMap6" type="com.javacode2018.chat05.demo7.model.OrderModel">  
 <association property="userModel">  
 <id column="user\_id" property="id"/>  
 </association>  
</resultMap>

##### 再次运行测试用例

com.javacode2018.chat05.demo7.Demo7Test#getById6\_0

##### 输出

15:02.751 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById6 - ==> Preparing: SELECT a.id, a.user\_id userId, a.create\_time createTime, a.up\_time upTime, b.id as user\_id, b.name FROM t\_order a,t\_user b WHERE a.user\_id = b.id AND a.id = ?   
15:02.783 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById6 - ==> Parameters: 2(Integer)  
15:02.801 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById6 - <== Total: 1  
15:02.801 [main] INFO c.j.chat05.demo7.Demo7Test - OrderModel(id=2, userId=1, createTime=1577947790, upTime=1577947790, userModel=UserModel(id=1, name=张学友))

这次userModel中的id正确了。

### autoMapping使用

上面我们有说过，当在resultMap中指定了autoMapping属性之后，这个resultMap的自动映射就受autoMapping属性的控制，和mybatis中全局映射配置（autoMappingBehavior）行为无关了。

#### 案例1

##### 这个核心配置主要在sqlmap中，如下：

<resultMap id="orderModelMap7" type="com.javacode2018.chat05.demo7.model.OrderModel" autoMapping="true">  
 <association property="userModel" autoMapping="true">  
 <id column="user\_id" property="id"/>  
 </association>  
</resultMap>  
  
<select id="getById7" resultMap="orderModelMap7">  
 <![CDATA[  
 SELECT  
 a.id,  
 a.user\_id userId,  
 a.create\_time createTime,  
 a.up\_time upTime,  
 b.id as user\_id,  
 b.name  
 FROM  
 t\_order a,t\_user b  
 WHERE  
 a.user\_id = b.id  
 AND a.id = #{value}  
 ]]>  
</select>

##### 对应测试用例

com.javacode2018.chat05.demo7.Demo7Test#getById7  
  
@Test  
public void getById7() throws IOException {  
 this.before("demo7/mybatis-config1.xml");  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);  
 OrderModel orderModel = mapper.getById7(2);  
 log.info("{}", orderModel);  
 }  
}

##### 运行输出

24:37.544 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById7 - ==> Preparing: SELECT a.id, a.user\_id userId, a.create\_time createTime, a.up\_time upTime, b.id as user\_id, b.name FROM t\_order a,t\_user b WHERE a.user\_id = b.id AND a.id = ?   
24:37.589 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById7 - ==> Parameters: 2(Integer)  
24:37.610 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById7 - <== Total: 1  
24:37.610 [main] INFO c.j.chat05.demo7.Demo7Test - OrderModel(id=2, userId=1, createTime=1577947790, upTime=1577947790, userModel=UserModel(id=1, name=张学友))

OrderModel中所有属性都自动映射成功。

自动装配并不是那么好玩，玩不转可能带来一些隐患，我们看一个案例，见下面的示例2。

#### 示例2

根据订单编号，查询出订单信息，顺便查询出订单明细列表。这个我们使用mybatis中的一对多查询。

##### OrderDetaiMapper.xml加入

<select id="getListByOrderId1" resultType="com.javacode2018.chat05.demo7.model.OrderDetailModel">  
 <![CDATA[  
 SELECT  
 a.id,  
 a.order\_id AS orderId,  
 a.goods\_id AS goodsId,  
 a.num,  
 a.total\_price AS totalPrice  
 FROM  
 t\_order\_detail a  
 WHERE  
 a.order\_id = #{value}  
 ]]>  
</select>

这个可以根据订单的id，查询出订单关联的明细列表。

##### OrderMapper.xml加入

<resultMap id="orderModelMap8" type="com.javacode2018.chat05.demo7.model.OrderModel" autoMapping="true">  
 <collection property="orderDetailModelList" select="com.javacode2018.chat05.demo7.mapper.OrderDetailMapper.getListByOrderId1" column="id"/>  
</resultMap>  
  
<select id="getById8" resultMap="orderModelMap8">  
 <![CDATA[  
 SELECT  
 a.id,  
 a.user\_id userId,  
 a.create\_time createTime,  
 a.up\_time upTime  
 FROM  
 t\_order a  
 WHERE a.id = #{value}  
 ]]>  
</select>

##### 测试用例

com.javacode2018.chat05.demo7.Demo7Test#getById8  
  
@Test  
public void getById8() throws IOException {  
 this.before("demo7/mybatis-config.xml");  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);  
 OrderModel orderModel = mapper.getById8(1);  
 log.info("{}", orderModel);  
 }  
}

##### 运行输出

11:06.193 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById8 - ==> Preparing: SELECT a.id, a.user\_id userId, a.create\_time createTime, a.up\_time upTime FROM t\_order a WHERE a.id = ?   
11:06.229 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById8 - ==> Parameters: 1(Integer)  
11:06.250 [main] DEBUG c.j.c.d.m.O.getListByOrderId1 - ====> Preparing: SELECT a.id, a.order\_id AS orderId, a.goods\_id AS goodsId, a.num, a.total\_price AS totalPrice FROM t\_order\_detail a WHERE a.order\_id = ?   
11:06.251 [main] DEBUG c.j.c.d.m.O.getListByOrderId1 - ====> Parameters: 1(Integer)  
11:06.255 [main] DEBUG c.j.c.d.m.O.getListByOrderId1 - <==== Total: 2  
11:06.256 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById8 - <== Total: 1  
11:06.256 [main] INFO c.j.chat05.demo7.Demo7Test - OrderModel(id=null, userId=2, createTime=1577947790, upTime=1577947790, userModel=null, orderDetailModelList=[OrderDetailModel(id=1, orderId=1, goodsId=1, num=2, totalPrice=17.76), OrderDetailModel(id=2, orderId=1, goodsId=1, num=1, totalPrice=16.66)])

注意输出中OrderModel的id属性，怎么是null值？主要是下面这行配置导致的

<collection property="orderDetailModelList" select="com.javacode2018.chat05.demo7.mapper.OrderDetailMapper.getListByOrderId1" column="id"/>

上面这个配置中有个column属性，指定的是id，此时mybatis认为你对id字段手动指定了映射关系，就跳过了对id字段到OrderModel.id属性的自动映射，所以导致OrderModel对象的id属性没有赋值，此时需要我们在orderModelMap8手动指定id的映射规则，如下：

<resultMap id="orderModelMap8" type="com.javacode2018.chat05.demo7.model.OrderModel" autoMapping="true">  
 <id column="id" property="id" />  
 <collection property="orderDetailModelList" select="com.javacode2018.chat05.demo7.mapper.OrderDetailMapper.getListByOrderId1" column="id"/>  
</resultMap>

再去运行测试用例就正常了。

### 总结一下

对于咱们开发来说，自动映射确实可以帮助我们节省一些代码，不过也存在一些隐患，我们希望自己开发的系统是健壮的，建议大家写mapper xml的时候，还是花点时间将映射的配置都给写上去，这样能够杜绝一些隐患，使我们的系统更稳定。

### MyBatis系列

1. MyBatis系列第1篇：MyBatis未出世之前我们那些痛苦的经历
2. MyBatis系列第2篇：入门篇，带你感受一下mybatis独特的魅力！
3. MyBatis系列第3篇：Mybatis使用详解（1）
4. MyBatis系列第4篇：Mybatis使用详解（2）
5. Mybatis系列第5篇：Mapper接口多种方式传参详解、原理、源码解析
6. Mybatis系列第6篇：恕我直言，mybatis增删改你未必玩得转!
7. Mybatis系列第7篇：各种查询详解

## Mybatis系列第9篇：延迟加载、鉴别器、继承

### 建库建表

创建一个db:javacode2018

4张表:

t\_user(用户表)

t\_goods(商品表)

t\_order(订单表)

t*order*detail(订单明细表)

表之间的关系：

t*order和t*user是一对一的关系，一条订单关联一个用户记录

t*order和t*order\_detail是一对多关系，每个订单中可能包含多个子订单，每个子订单对应一个商品

DROP DATABASE IF EXISTS `javacode2018`;  
CREATE DATABASE `javacode2018`;  
  
USE `javacode2018`;  
  
DROP TABLE IF EXISTS t\_user;  
CREATE TABLE t\_user(  
 id int AUTO\_INCREMENT PRIMARY KEY COMMENT '用户id',  
 name VARCHAR(32) NOT NULL DEFAULT '' COMMENT '用户名'  
) COMMENT '用户表';  
INSERT INTO t\_user VALUES (1,'张学友'),(2,'马士兵老师');  
  
DROP TABLE IF EXISTS t\_goods;  
CREATE TABLE t\_goods(  
 id int AUTO\_INCREMENT PRIMARY KEY COMMENT '商品id',  
 name VARCHAR(32) NOT NULL DEFAULT '' COMMENT '商品名称',  
 price DECIMAL(10,2) NOT NULL DEFAULT 0 COMMENT '商品价格'  
) COMMENT '商品信息表';  
INSERT INTO t\_goods VALUES (1,'Mybatis系列',8.88),(2,'maven高手系列',16.66);  
  
DROP TABLE IF EXISTS t\_order;  
CREATE TABLE t\_order(  
 id int AUTO\_INCREMENT PRIMARY KEY COMMENT '订单id',  
 user\_id INT NOT NULL DEFAULT 0 COMMENT '用户id，来源于t\_user.id',  
 create\_time BIGINT NOT NULL DEFAULT 0 COMMENT '订单创建时间(时间戳，秒)',  
 up\_time BIGINT NOT NULL DEFAULT 0 COMMENT '订单最后修改时间(时间戳，秒)'  
) COMMENT '订单表';  
INSERT INTO t\_order VALUES (1,2,unix\_timestamp(now()),unix\_timestamp(now())),(2,1,unix\_timestamp(now()),unix\_timestamp(now())),(3,1,unix\_timestamp(now()),unix\_timestamp(now()));  
  
DROP TABLE IF EXISTS t\_order\_detail;  
CREATE TABLE t\_order\_detail(  
 id int AUTO\_INCREMENT PRIMARY KEY COMMENT '订单明细id',  
 order\_id INT NOT NULL DEFAULT 0 COMMENT '订单id，来源于t\_order.id',  
 goods\_id INT NOT NULL DEFAULT 0 COMMENT '商品id，来源于t\_goods.id',  
 num INT NOT NULL DEFAULT 0 COMMENT '商品数量',  
 total\_price DECIMAL(12,2) NOT NULL DEFAULT 0 COMMENT '商品总金额'  
) COMMENT '订单表';  
INSERT INTO t\_order\_detail VALUES (1,1,1,2,17.76),(2,1,1,1,16.66),(3,2,1,1,8.88),(4,3,1,1,8.88);  
  
select \* from t\_user;  
select \* from t\_goods;  
select \* from t\_order;  
select \* from t\_order\_detail;

### 延迟加载

延迟加载其实就是将数据加载时机推迟，比如推迟嵌套查询的执行时机，在mybatis中经常用到关联查询，但是并不是任何时候都需要立即返回关联查询结果。比如查询订单信息，并不一定需要及时返回订单对应的用户信息或者订单详情信息等，这种情况需要一种机制，当需要查看关联的数据时，再去执行对应的查询，返回需要的结果，这种需求在mybatis中可以使用延迟加载机制来实现。

#### 延迟加载2种设置方式

1. 全局配置的方式
2. sqlmap中配置的方式

方式1中会对所有关联查询起效，而方式2只会对相关设置的查询起效。

### 全局配置延迟加载

mybatis配置文件中通过下面两个属性来控制延迟加载：

<settings>  
 <!--打开延迟加载的开关 -->  
 <setting name="lazyLoadingEnabled" value="true"/>  
 <!-- 当为true的时候，调用任意延迟属性，会去加载所有延迟属性，如果为false，则调用某个属性的时候，只会加载指定的属性 -->  
 <setting name="aggressiveLazyLoading" value="false"/>  
</settings>

lazyLoadingEnabled：这个属性比较好理解，是否开启延迟加载，默认为false，如果需要开启延迟加载，将其设置为true

aggressiveLazyLoading：当为true的时候，调用任意延迟属性，会去加载所有延迟属性，如果为false，则调用某个属性的时候，只会加载指定的属性

下面我们来个案例感受一下效果。

#### 需求

通过订单id查询订单信息、订单用户信息、订单明细列表，而订单用户信息、订单明细列表采用延迟加载的方式获取。

#### mybatis配置

<settings>  
 <!--打开延迟加载的开关 -->  
 <setting name="lazyLoadingEnabled" value="true"/>  
 <!-- 当为true的时候，调用任意延迟属性，会去加载所有延迟属性，如果为false，则调用某个属性的时候，只会加载指定的属性 -->  
 <setting name="aggressiveLazyLoading" value="true"/>  
</settings>

#### OrderMapper.xml

<resultMap id="orderModelMap1" type="com.javacode2018.chat05.demo5.model.OrderModel">  
 <id column="id" property="id"/>  
 <result column="user\_id" property="userId"/>  
 <result column="create\_time" property="createTime"/>  
 <result column="up\_time" property="upTime"/>  
 <!-- 通过订单中user\_id作为条件，查询下单人信息 -->  
 <association property="userModel" select="com.javacode2018.chat05.demo5.mapper.UserMapper.getById1" column="user\_Id"/>  
 <!-- 通过订单id作为条件，查询详情列表 -->  
 <collection property="orderDetailModelList" select="com.javacode2018.chat05.demo5.mapper.OrderDetailMapper.getListByOrderId1" column="id"/>  
</resultMap>  
  
<select id="getById1" resultMap="orderModelMap1">  
 <![CDATA[  
 SELECT  
 a.id ,  
 a.user\_id,  
 a.create\_time,  
 a.up\_time  
 FROM  
 t\_order a  
 WHERE  
 a.id = #{value}  
 ]]>  
</select>

上面的orderModelMap1元素下面有两个关联查询，我们也写一下。

#### UserMapper.xml

<!-- 根据用户id查询用户信息 -->  
<select id="getById1" resultType="com.javacode2018.chat05.demo5.model.UserModel">  
 <![CDATA[  
 SELECT id,name FROM t\_user where id = #{user\_id}  
 ]]>  
</select>

#### OrderDetailMapper.xml

<!-- 根据订单di查询订单明细列表 -->  
<select id="getListByOrderId1" resultType="com.javacode2018.chat05.demo5.model.OrderDetailModel">  
 <![CDATA[  
 SELECT  
 a.id,  
 a.order\_id AS orderId,  
 a.goods\_id AS goodsId,  
 a.num,  
 a.total\_price AS totalPrice  
 FROM  
 t\_order\_detail a  
 WHERE  
 a.order\_id = #{order\_id}  
 ]]>  
</select>

#### 对应的3个Model

@Getter  
@Setter  
@Builder  
@ToString  
@NoArgsConstructor  
@AllArgsConstructor  
public class OrderModel {  
 private Integer id;  
 private Integer userId;  
 private Long createTime;  
 private Long upTime;  
 private UserModel userModel;  
 //订单详情列表  
 private List<OrderDetailModel> orderDetailModelList;  
}  
  
@Getter  
@Setter  
@Builder  
@ToString  
@NoArgsConstructor  
@AllArgsConstructor  
public class UserModel {  
 private Integer id;  
 private String name;  
}  
  
@Getter  
@Setter  
@Builder  
@ToString  
@NoArgsConstructor  
@AllArgsConstructor  
public class OrderDetailModel {  
 private Integer id;  
 private Integer orderId;  
 private Integer goodsId;  
 private Integer num;  
 private Double totalPrice;  
}

#### 测试用例

com.javacode2018.chat05.demo5.Demo5Test#getById1  
  
@Test  
public void getById1() throws IOException {  
 //指定mybatis全局配置文件  
 mybatisConfig = "demo5/mybatis-config.xml";  
 this.before();  
 OrderModel orderModel = null;  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);  
 orderModel = mapper.getById1(1);  
 }  
 log.info("-------分割线--------");  
 log.info("{}", orderModel.getUserModel());  
}

#### 运行输出

01:55.343 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - ==> Preparing: SELECT a.id , a.user\_id, a.create\_time, a.up\_time FROM t\_order a WHERE a.id = ?   
01:55.372 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - ==> Parameters: 1(Integer)  
01:55.431 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - <== Total: 1  
01:55.431 [main] INFO c.j.chat05.demo5.Demo5Test - -------分割线--------  
01:55.432 [main] DEBUG c.j.c.d.m.O.getListByOrderId1 - ==> Preparing: SELECT a.id, a.order\_id AS orderId, a.goods\_id AS goodsId, a.num, a.total\_price AS totalPrice FROM t\_order\_detail a WHERE a.order\_id = ?   
01:55.432 [main] DEBUG c.j.c.d.m.O.getListByOrderId1 - ==> Parameters: 1(Integer)  
01:55.435 [main] DEBUG c.j.c.d.m.O.getListByOrderId1 - <== Total: 2  
01:55.439 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - ==> Preparing: SELECT id,name FROM t\_user where id = ?   
01:55.439 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - ==> Parameters: 2(Integer)  
01:55.441 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - <== Total: 1  
01:55.441 [main] INFO c.j.chat05.demo5.Demo5Test - UserModel(id=2, name=马士兵老师)

从日志中可以看出，总共有3次查询，后面2次查询在分割线之后出现的，说明是调用了orderModel.getUserModel()触发后面2次查询动作。

代码中我们调用的是获取用户信息，而订单列表信息也被加载了，这个主要是由于aggressiveLazyLoading被设置为true了，当使用到一个延迟加载的属性时，其他的延迟加载的属性也会被一起加载，所以触发了2个关联的查询。

下面我们看看将aggressiveLazyLoading设置为false的效果

<settings>  
 <!--打开延迟加载的开关 -->  
 <setting name="lazyLoadingEnabled" value="true"/>  
 <!-- 当为true的时候，调用任意延迟属性，会去加载所有延迟属性，如果为false，则调用某个属性的时候，只会加载指定的属性 -->  
 <setting name="aggressiveLazyLoading" value="false"/>  
</settings>

再次运行测试用例输出

12:19.236 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - ==> Preparing: SELECT a.id , a.user\_id, a.create\_time, a.up\_time FROM t\_order a WHERE a.id = ?   
12:19.268 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - ==> Parameters: 1(Integer)  
12:19.336 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - <== Total: 1  
12:19.337 [main] INFO c.j.chat05.demo5.Demo5Test - -------分割线--------  
12:19.338 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - ==> Preparing: SELECT id,name FROM t\_user where id = ?   
12:19.338 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - ==> Parameters: 2(Integer)  
12:19.340 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - <== Total: 1  
12:19.341 [main] INFO c.j.chat05.demo5.Demo5Test - UserModel(id=2, name=马士兵老师)

### sqlmap中设置延迟加载

全局的方式会对所有的关联查询起效，影响范围比较大，mybatis也提供了在关联查询中进行设置的方式，只会对当前设置的关联查询起效。

关联查询，一般我们使用association、collection，这两个元素都有个属性fetchType，通过这个属性可以指定关联查询的加载方式。

fetchType值有2种，eager：立即加载；lazy：延迟加载。

下面我们来实现一个需求：还是通过订单id查询订单信息，并获取关联的用户信息、订单详细列表，用户信息我们要求立即加载，而订单详情我们要求延迟加载。

#### mapper xml如下

<resultMap id="orderModelMap2" type="com.javacode2018.chat05.demo5.model.OrderModel">  
 <id column="id" property="id"/>  
 <result column="user\_id" property="userId"/>  
 <result column="create\_time" property="createTime"/>  
 <result column="up\_time" property="upTime"/>  
 <!-- 通过订单中user\_id作为条件，查询下单人信息 -->  
 <association property="userModel" fetchType="eager" select="com.javacode2018.chat05.demo5.mapper.UserMapper.getById1" column="user\_Id"/>  
 <!-- 通过订单id作为条件，查询详情列表 -->  
 <collection property="orderDetailModelList" fetchType="lazy" select="com.javacode2018.chat05.demo5.mapper.OrderDetailMapper.getListByOrderId1" column="id"/>  
</resultMap>  
  
<select id="getById2" resultMap="orderModelMap2">  
 <![CDATA[  
 SELECT  
 a.id ,  
 a.user\_id,  
 a.create\_time,  
 a.up\_time  
 FROM  
 t\_order a  
 WHERE  
 a.id = #{value}  
 ]]>  
</select>

重点注意上面配置中association、collection这2个元素的fetchType属性，eager表示立即加载，lazy表示延迟加载。

#### 测试用例

com.javacode2018.chat05.demo5.Demo5Test#getById2  
  
@Test  
public void getById2() throws IOException {  
 //指定mybatis全局配置文件  
 mybatisConfig = "demo5/mybatis-config2.xml";  
 this.before();  
 OrderModel orderModel = null;  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);  
 orderModel = mapper.getById2(1);  
 }  
 log.info("-------分割线--------");  
 log.info("{}", orderModel.getOrderDetailModelList());  
}

#### 运行输出

36:54.284 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - ==> Preparing: SELECT a.id , a.user\_id, a.create\_time, a.up\_time FROM t\_order a WHERE a.id = ?   
36:54.321 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - ==> Parameters: 1(Integer)  
36:54.385 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - ====> Preparing: SELECT id,name FROM t\_user where id = ?   
36:54.385 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - ====> Parameters: 2(Integer)  
36:54.387 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - <==== Total: 1  
36:54.389 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - <== Total: 1  
36:54.390 [main] INFO c.j.chat05.demo5.Demo5Test - -------分割线--------  
36:54.392 [main] DEBUG c.j.c.d.m.O.getListByOrderId1 - ==> Preparing: SELECT a.id, a.order\_id AS orderId, a.goods\_id AS goodsId, a.num, a.total\_price AS totalPrice FROM t\_order\_detail a WHERE a.order\_id = ?   
36:54.392 [main] DEBUG c.j.c.d.m.O.getListByOrderId1 - ==> Parameters: 1(Integer)  
36:54.397 [main] DEBUG c.j.c.d.m.O.getListByOrderId1 - <== Total: 2  
36:54.398 [main] INFO c.j.chat05.demo5.Demo5Test - [OrderDetailModel(id=1, orderId=1, goodsId=1, num=2, totalPrice=17.76), OrderDetailModel(id=2, orderId=1, goodsId=1, num=1, totalPrice=16.66)]

注意输出中的分割线，可以分析得出，用户信息是和订单信息一起立即查出来的，而订单详情，是在我们调用orderModel.getOrderDetailModelList()获取订单列表的时候，采取懒加载的。

### 鉴别器(discriminator)

有时候，一个数据库查询可能会返回多个不同的结果集（但总体上还是有一定的联系的）， 鉴别器（discriminator）元素就是被设计来应对这种情况的，鉴别器的概念很好理解——它很像 Java 语言中的 switch 语句。

discriminator标签常用的两个属性如下：

* column：该属性用于设置要进行鉴别比较值的列。
* javaType：该属性用于指定列的类型，保证使用相同的java类型来比较值。

discriminator标签可以有1个或多个case标签，case标签有一个比较重要的属性：

* value：该值为discriminator指定column用来匹配的值，当匹配的时候，结果会走这个case关联的映射。

我们使用鉴别器实现一个功能：通过订单id查询订单信息，当传入的订单id为1的时候，获取订单信息及下单人信息；当传入的订单id为2的时候，获取订单信息、下单人信息、订单明细信息；其他情况默认只查询订单信息。

#### OrderMapper.xml

<resultMap id="orderModelMap1" type="com.javacode2018.chat05.demo6.model.OrderModel">  
 <id column="id" property="id"/>  
 <result column="user\_id" property="userId"/>  
 <result column="create\_time" property="createTime"/>  
 <result column="up\_time" property="upTime"/>  
 <!-- 鉴别器 -->  
 <discriminator javaType="int" column="id">  
 <case value="1">  
 <!--通过用户id查询用户信息-->  
 <association property="userModel" select="com.javacode2018.chat05.demo6.mapper.UserMapper.getById1" column="user\_Id"/>  
 </case>  
 <case value="2">  
 <!--通过用户id查询用户信息-->  
 <association property="userModel" select="com.javacode2018.chat05.demo6.mapper.UserMapper.getById1" column="user\_Id"/>  
 <!--通过订单id查询订单列表-->  
 <collection property="orderDetailModelList" select="com.javacode2018.chat05.demo6.mapper.OrderDetailMapper.getListByOrderId1" column="id"/>  
 </case>  
 </discriminator>  
</resultMap>  
  
<select id="getById1" resultMap="orderModelMap1">  
 <![CDATA[  
 SELECT  
 a.id ,  
 a.user\_id,  
 a.create\_time,  
 a.up\_time  
 FROM  
 t\_order a  
 WHERE  
 a.id = #{value}  
 ]]>  
</select>

注意上面的discriminator，这部分是关键，discriminator内部的case会和每行查询结果中的id字段进行匹配，匹配成功了case内部的关联查询会被执行，未匹配上的，只会走discriminator外部默认配置的映射映射规则。

#### UserMapper.xml

<!-- 通过用户id查询用户信息 -->  
<select id="getById1" resultType="com.javacode2018.chat05.demo6.model.UserModel">  
 <![CDATA[  
 SELECT id,name FROM t\_user where id = #{user\_id}  
 ]]>  
</select>

#### OrderDetailMapper.xml

<!-- 通过订单id查询订单明细列表 -->  
<select id="getListByOrderId1" resultType="com.javacode2018.chat05.demo6.model.OrderDetailModel">  
 <![CDATA[  
 SELECT  
 a.id,  
 a.order\_id AS orderId,  
 a.goods\_id AS goodsId,  
 a.num,  
 a.total\_price AS totalPrice  
 FROM  
 t\_order\_detail a  
 WHERE  
 a.order\_id = #{order\_id}  
 ]]>  
</select>

#### 对应的3个Model类

@Getter  
@Setter  
@Builder  
@ToString  
@NoArgsConstructor  
@AllArgsConstructor  
public class OrderModel {  
 private Integer id;  
 private Integer userId;  
 private Long createTime;  
 private Long upTime;  
 //用户信息  
 private UserModel userModel;  
 //订单详情列表  
 private List<OrderDetailModel> orderDetailModelList;  
}  
  
  
@Getter  
@Setter  
@Builder  
@ToString  
@NoArgsConstructor  
@AllArgsConstructor  
public class UserModel {  
 private Integer id;  
 private String name;  
}  
  
@Getter  
@Setter  
@Builder  
@ToString  
@NoArgsConstructor  
@AllArgsConstructor  
public class OrderDetailModel {  
 private Integer id;  
 private Integer orderId;  
 private Integer goodsId;  
 private Integer num;  
 private Double totalPrice;  
}

#### 测试用例

com.javacode2018.chat05.demo6.Demo6Test#getById1  
  
@Test  
public void getById1() throws IOException {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);  
 //查询订单为1的  
 OrderModel orderModel = mapper.getById1(1);  
 log.info("{}", orderModel);  
 log.info("------------------------------------------------------------");  
 //查询订单为2的  
 orderModel = mapper.getById1(2);  
 log.info("{}", orderModel);  
 log.info("------------------------------------------------------------");  
 //查询订单为3的  
 orderModel = mapper.getById1(3);  
 log.info("{}", orderModel);  
 }  
}

#### 运行输出

58:16.413 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - ==> Preparing: SELECT a.id , a.user\_id, a.create\_time, a.up\_time FROM t\_order a WHERE a.id = ?   
58:16.457 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - ==> Parameters: 1(Integer)  
58:16.481 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - ====> Preparing: SELECT id,name FROM t\_user where id = ?   
58:16.481 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - ====> Parameters: 2(Integer)  
58:16.488 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - <==== Total: 1  
58:16.489 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - <== Total: 1  
58:16.489 [main] INFO c.j.chat05.demo6.Demo6Test - OrderModel(id=1, userId=2, createTime=1578368161, upTime=1578368161, userModel=UserModel(id=2, name=马士兵老师), orderDetailModelList=null)  
58:16.491 [main] INFO c.j.chat05.demo6.Demo6Test - ------------------------------------------------------------  
58:16.491 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - ==> Preparing: SELECT a.id , a.user\_id, a.create\_time, a.up\_time FROM t\_order a WHERE a.id = ?   
58:16.492 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - ==> Parameters: 2(Integer)  
58:16.493 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - ====> Preparing: SELECT id,name FROM t\_user where id = ?   
58:16.493 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - ====> Parameters: 1(Integer)  
58:16.494 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - <==== Total: 1  
58:16.495 [main] DEBUG c.j.c.d.m.O.getListByOrderId1 - ====> Preparing: SELECT a.id, a.order\_id AS orderId, a.goods\_id AS goodsId, a.num, a.total\_price AS totalPrice FROM t\_order\_detail a WHERE a.order\_id = ?   
58:16.495 [main] DEBUG c.j.c.d.m.O.getListByOrderId1 - ====> Parameters: 2(Integer)  
58:16.505 [main] DEBUG c.j.c.d.m.O.getListByOrderId1 - <==== Total: 1  
58:16.505 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - <== Total: 1  
58:16.506 [main] INFO c.j.chat05.demo6.Demo6Test - OrderModel(id=2, userId=1, createTime=1578368161, upTime=1578368161, userModel=UserModel(id=1, name=张学友), orderDetailModelList=[OrderDetailModel(id=3, orderId=2, goodsId=1, num=1, totalPrice=8.88)])  
58:16.506 [main] INFO c.j.chat05.demo6.Demo6Test - ------------------------------------------------------------  
58:16.506 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - ==> Preparing: SELECT a.id , a.user\_id, a.create\_time, a.up\_time FROM t\_order a WHERE a.id = ?   
58:16.506 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - ==> Parameters: 3(Integer)  
58:16.508 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById1 - <== Total: 1  
58:16.509 [main] INFO c.j.chat05.demo6.Demo6Test - OrderModel(id=3, userId=1, createTime=1578368161, upTime=1578368161, userModel=null, orderDetailModelList=null)

输出中可以看出，订单1查询了2次，订单2查询了3次，订单3查询了1次；鉴别器算是一个不错的功能。

### 继承(extends)

继承在java是三大特性之一，可以起到重用代码的作用，而mybatis也有继承的功能，和java中的继承的作用类似，主要在resultMap中使用，可以重用其他resultMap中配置的映射关系。

#### 用法

<resultMap extends="被继承的resultMap的id"></resultMap>

#### 案例

下面我们使用继承来对上面的鉴别器的案例改造一下，优化一下代码

##### OrderMapper.xml

<resultMap id="orderModelMap2" type="com.javacode2018.chat05.demo6.model.OrderModel">  
 <id column="id" property="id"/>  
 <result column="user\_id" property="userId"/>  
 <result column="create\_time" property="createTime"/>  
 <result column="up\_time" property="upTime"/>  
 <!-- 鉴别器 -->  
 <discriminator javaType="int" column="id">  
 <case value="1" resultMap="orderModelMap3" />  
 <case value="2" resultMap="orderModelMap4" />  
 </discriminator>  
</resultMap>  
  
<resultMap id="orderModelMap3" type="com.javacode2018.chat05.demo6.model.OrderModel" extends="orderModelMap2">  
 <!--通过用户id查询用户信息-->  
 <association property="userModel" select="com.javacode2018.chat05.demo6.mapper.UserMapper.getById1" column="user\_Id"/>  
</resultMap>  
  
<resultMap id="orderModelMap4" type="com.javacode2018.chat05.demo6.model.OrderModel" extends="orderModelMap3">  
 <!--通过订单id查询订单列表-->  
 <collection property="orderDetailModelList" select="com.javacode2018.chat05.demo6.mapper.OrderDetailMapper.getListByOrderId1" column="id"/>  
</resultMap>  
  
<select id="getById2" resultMap="orderModelMap2">  
 <![CDATA[  
 SELECT  
 a.id ,  
 a.user\_id,  
 a.create\_time,  
 a.up\_time  
 FROM  
 t\_order a  
 WHERE  
 a.id = #{value}  
 ]]>  
</select>

重点在于上面两个extends属性，上面orderModelMap3继承了orderModelMap2中配置的映射关系（除鉴别器之外），自己又加入了一个association去查询用户信息；orderModelMap4继承了orderModelMap3，自己又加入了一个查询订单列表的collection元素。上面使用extends做到了代码重用，其实和下面这块代码写法效果一样：

<resultMap id="orderModelMap2" type="com.javacode2018.chat05.demo6.model.OrderModel">  
 <id column="id" property="id"/>  
 <result column="user\_id" property="userId"/>  
 <result column="create\_time" property="createTime"/>  
 <result column="up\_time" property="upTime"/>  
 <!-- 鉴别器 -->  
 <discriminator javaType="int" column="id">  
 <case value="1" resultMap="orderModelMap3" />  
 <case value="2" resultMap="orderModelMap4" />  
 </discriminator>  
</resultMap>  
  
<resultMap id="orderModelMap3" type="com.javacode2018.chat05.demo6.model.OrderModel">  
 <id column="id" property="id"/>  
 <result column="user\_id" property="userId"/>  
 <result column="create\_time" property="createTime"/>  
 <result column="up\_time" property="upTime"/>  
 <!--通过用户id查询用户信息-->  
 <association property="userModel" select="com.javacode2018.chat05.demo6.mapper.UserMapper.getById1" column="user\_Id"/>  
</resultMap>  
  
<resultMap id="orderModelMap4" type="com.javacode2018.chat05.demo6.model.OrderModel">  
 <id column="id" property="id"/>  
 <result column="user\_id" property="userId"/>  
 <result column="create\_time" property="createTime"/>  
 <result column="up\_time" property="upTime"/>  
 <!--通过用户id查询用户信息-->  
 <association property="userModel" select="com.javacode2018.chat05.demo6.mapper.UserMapper.getById1" column="user\_Id"/>  
 <!--通过订单id查询订单列表-->  
 <collection property="orderDetailModelList" select="com.javacode2018.chat05.demo6.mapper.OrderDetailMapper.getListByOrderId1" column="id"/>  
</resultMap>  
  
<select id="getById2" resultMap="orderModelMap2">  
 <![CDATA[  
 SELECT  
 a.id ,  
 a.user\_id,  
 a.create\_time,  
 a.up\_time  
 FROM  
 t\_order a  
 WHERE  
 a.id = #{value}  
 ]]>  
</select>

##### 测试用例

com.javacode2018.chat05.demo6.Demo6Test#getById2  
  
@Test  
public void getById2() throws IOException {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);  
 //查询订单为1的  
 OrderModel orderModel = mapper.getById2(1);  
 log.info("{}", orderModel);  
 log.info("------------------------------------------------------------");  
 //查询订单为2的  
 orderModel = mapper.getById2(2);  
 log.info("{}", orderModel);  
 log.info("------------------------------------------------------------");  
 //查询订单为3的  
 orderModel = mapper.getById2(3);  
 log.info("{}", orderModel);  
 }  
}

##### 运行输出

39:55.936 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - ==> Preparing: SELECT a.id , a.user\_id, a.create\_time, a.up\_time FROM t\_order a WHERE a.id = ?   
39:55.969 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - ==> Parameters: 1(Integer)  
39:55.986 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - ====> Preparing: SELECT id,name FROM t\_user where id = ?   
39:55.987 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - ====> Parameters: 2(Integer)  
39:55.992 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - <==== Total: 1  
39:55.993 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - <== Total: 1  
39:55.993 [main] INFO c.j.chat05.demo6.Demo6Test - OrderModel(id=1, userId=2, createTime=1578368161, upTime=1578368161, userModel=UserModel(id=2, name=马士兵老师), orderDetailModelList=null)  
39:55.994 [main] INFO c.j.chat05.demo6.Demo6Test - ------------------------------------------------------------  
39:55.994 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - ==> Preparing: SELECT a.id , a.user\_id, a.create\_time, a.up\_time FROM t\_order a WHERE a.id = ?   
39:55.995 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - ==> Parameters: 2(Integer)  
39:55.995 [main] DEBUG c.j.c.d.m.O.getListByOrderId1 - ====> Preparing: SELECT a.id, a.order\_id AS orderId, a.goods\_id AS goodsId, a.num, a.total\_price AS totalPrice FROM t\_order\_detail a WHERE a.order\_id = ?   
39:55.996 [main] DEBUG c.j.c.d.m.O.getListByOrderId1 - ====> Parameters: 2(Integer)  
39:56.000 [main] DEBUG c.j.c.d.m.O.getListByOrderId1 - <==== Total: 1  
39:56.001 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - ====> Preparing: SELECT id,name FROM t\_user where id = ?   
39:56.004 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - ====> Parameters: 1(Integer)  
39:56.005 [main] DEBUG c.j.c.d.mapper.UserMapper.getById1 - <==== Total: 1  
39:56.005 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - <== Total: 1  
39:56.005 [main] INFO c.j.chat05.demo6.Demo6Test - OrderModel(id=2, userId=1, createTime=1578368161, upTime=1578368161, userModel=UserModel(id=1, name=张学友), orderDetailModelList=[OrderDetailModel(id=3, orderId=2, goodsId=1, num=1, totalPrice=8.88)])  
39:56.005 [main] INFO c.j.chat05.demo6.Demo6Test - ------------------------------------------------------------  
39:56.005 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - ==> Preparing: SELECT a.id , a.user\_id, a.create\_time, a.up\_time FROM t\_order a WHERE a.id = ?   
39:56.006 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - ==> Parameters: 3(Integer)  
39:56.007 [main] DEBUG c.j.c.d.mapper.OrderMapper.getById2 - <== Total: 1  
39:56.007 [main] INFO c.j.chat05.demo6.Demo6Test - OrderModel(id=3, userId=1, createTime=1578368161, upTime=1578368161, userModel=null, orderDetailModelList=null)

### 案例代码

链接：https://pan.baidu.com/s/1vt-MAX3oJOu9gyxZAhKkbg   
提取码：i8op

### MyBatis系列

1. MyBatis系列第1篇：MyBatis未出世之前我们那些痛苦的经历
2. MyBatis系列第2篇：入门篇，带你感受一下mybatis独特的魅力！
3. MyBatis系列第3篇：Mybatis使用详解（1）
4. MyBatis系列第4篇：Mybatis使用详解（2）
5. Mybatis系列第5篇：Mapper接口多种方式传参详解、原理、源码解析
6. Mybatis系列第6篇：恕我直言，mybatis增删改你未必玩得转!
7. Mybatis系列第7篇：各种查询详解
8. Mybatis系列第8篇：自动映射，使用需谨慎！

## Mybatis系列第10篇：动态SQL，这么多种你都会么？

mybatis中一个比较强大的功能就是动态sql，记得在刚开始工作那会，当时使用jdbc开发系统，在java代码中搞了很多判断去拼接sql，代码看起来比较乱，也不方便维护和扩展。

mybatis在处理sql的拼接这块简直是我们的福音，基本上sql拼接的所有的痛点，mybatis都帮我们解决了。下面我们来学一下mybatis中各种动态sql的用法。

### 案例sql脚本

DROP DATABASE IF EXISTS `javacode2018`;  
CREATE DATABASE `javacode2018`;  
  
USE `javacode2018`;  
  
DROP TABLE IF EXISTS t\_user;  
CREATE TABLE t\_user(  
 id int AUTO\_INCREMENT PRIMARY KEY COMMENT '用户id',  
 name VARCHAR(32) NOT NULL DEFAULT '' COMMENT '用户名',  
 age SMALLINT NOT NULL DEFAULT 1 COMMENT '年龄'  
) COMMENT '用户表';  
INSERT INTO t\_user VALUES (1,'马士兵老师',30),(2,'张学友',50),(3,'刘德华',50);

### if元素

相当于java中的if判断，语法：

<if test="判断条件">  
需要追加的sql  
</if>

test的值为一个判断表达式，写法上采用OGNL表达式的方式，OGNL在struts2中用的比较多，本文暂时对ognl不做详细介绍，有兴趣的可以去查一下相关资料。

当test成立的时候，if体内部的sql会被拼接上。

如：

<select id="getList1" resultType="com.javacode2018.chat05.demo8.model.UserModel" parameterType="map">  
 SELECT id,name,age FROM t\_user  
 WHERE 1 = 1  
 <if test="id!=null">  
 AND id = #{id}  
 </if>  
 <if test="name!=null and name.toString()!=''">  
 AND name = #{name}  
 </if>  
 <if test="age!=null">  
 AND age = #{age}  
 </if>  
</select>

上面查询用户列表，参数为一个map，当map中id不为空的时候，将其作为条件查询，如果name不为空，将name也作为条件，如果age不为空，将age也作为条件进行查询

当只传入id的时候，sql如下：

SELECT id,name,age FROM t\_user WHERE 1 = 1 AND id = ?

当3个参数都传了，sql如下：

SELECT id,name,age FROM t\_user WHERE 1 = 1 AND id = ? AND name = ? AND age = ?

上面这种写法相对于java代码看起来是不是清爽了很多，也更方便维护，大家注意一下sql中有个WHERE 1=1，如果没有这个，上面单通过if元素就不好实现了，mybatis也有解决方案，稍后会说明。

### choose/when/otherwise元素

这个相当于java中的if..else if..else，语法：

<choose>  
 <when test="条件1">  
 满足条件1追加的sql  
 </when>  
 <when test="条件2">  
 满足条件2追加的sql  
 </when>  
 <when test="条件n">  
 满足条件n追加的sql  
 </when>  
 <otherwise>  
 都不满足追加的sql  
 </otherwise>  
</choose>

choose内部的条件满足一个，choose内部的sql拼接就会结束。

otherwise属于可选的，当所有条件都不满足的时候，otherwise将起效。

如：

传入id、name、age作为条件，按顺序进行判断，如果id不为空，将id作为条件，忽略其他条件，如果id为空，会判断name是否为空，name不为空将name作为条件，如果name为空，再看看age是否为空，如果age不为空，将age作为条件。

<select id="getList2" resultType="com.javacode2018.chat05.demo8.model.UserModel" parameterType="map">  
 SELECT id,name,age FROM t\_user  
 WHERE 1 = 1  
 <choose>  
 <when test="id!=null">  
 AND id = #{id}  
 </when>  
 <when test="name!=null and name.toString()!=''">  
 AND name = #{name}  
 </when>  
 <when test="age!=null">  
 AND age = #{age}  
 </when>  
 </choose>  
</select>

如果id、name、age都传了，sql如下：

SELECT id,name,age FROM t\_user WHERE 1 = 1 AND id = ?

如果值传递了name、age，sql如下：

SELECT id,name,age FROM t\_user WHERE 1 = 1 AND name = ?

name判断在age前面，所以name条件先匹配上了。

### where元素

上面2个案例的sql中都有where 1=1这部分代码，虽然可以解决问题，但是看起来不美观，如果将where 1=1中1=1这部分干掉，上面的两个案例都会出问题，where后面会多一个AND符号，mybatis中已经考虑到这种问题了，属于通用性的问题，mybatis中通过where 元素来解决，当使用where元素的时候，mybatis会将where内部拼接的sql进行处理，会将这部分sql前面的AND 或者 OR给去掉，并在前面追加一个where，我们使用where元素来对上面的案例1进行改造，如下：

<select id="getList1" resultType="com.javacode2018.chat05.demo8.model.UserModel" parameterType="map">  
 SELECT id,name,age FROM t\_user  
 <where>  
 <if test="id!=null">  
 AND id = #{id}  
 </if>  
 <if test="name!=null and name.toString()!=''">  
 AND name = #{name}  
 </if>  
 <if test="age!=null">  
 AND age = #{age}  
 </if>  
 </where>  
</select>

where 1=1被替换成了where 元素。

当传入id、name的时候，where内部的sql会变成这样：

AND id = ? AND name = ?

mybatis会对上面这个sql进行处理，将前面的AND给去掉，并在前面追加一个where，变成了下面这样

where id = ? AND name = ?

案例2也用where改造一下，变成了下面这样：

<select id="getList2" resultType="com.javacode2018.chat05.demo8.model.UserModel" parameterType="map">  
 SELECT id,name,age FROM t\_user  
 <where>  
 <choose>  
 <when test="id!=null">  
 AND id = #{id}  
 </when>  
 <when test="name!=null and name.toString()!=''">  
 AND name = #{name}  
 </when>  
 <when test="age!=null">  
 AND age = #{age}  
 </when>  
 </choose>  
 </where>  
</select>

这下看起来是不是舒服很多了。

### set元素

现在我们想通过用户id更新用户信息，参数为UserModel对象，对象中的属性如果不为空，就进行更新，我们可以这么写：

<update id="update1" parameterType="com.javacode2018.chat05.demo8.model.UserModel">  
 UPDATE t\_user SET  
 <if test="name!=null">  
 name = #{name},  
 </if>  
 <if test="age!=null">  
 age = #{age},  
 </if>  
 <where>  
 <if test="id!=null">  
 AND id = #{id}  
 </if>  
 </where>  
</update>

我们来看一下，当所有属性都传值了，sql变成了下面这样：

UPDATE t\_user SET name = ?, age = ?, where id = ?

上面这个sql是有问题的，where前面多了一个逗号，得想办法将这个逗号去掉，这个逗号属于最后一个需要更新的字段后面的逗号，属于多余的，mybatis中提供了set元素来解决这个问题，将上面的代码改成下面这样：

<update id="update1" parameterType="com.javacode2018.chat05.demo8.model.UserModel">  
 UPDATE t\_user  
 <set>  
 <if test="name!=null">  
 name = #{name},  
 </if>  
 <if test="age!=null">  
 age = #{age},  
 </if>  
 </set>  
 <where>  
 <if test="id!=null">  
 AND id = #{id}  
 </if>  
 </where>  
</update>

我们将sql中的set去掉了，加了个set元素，set元素会对其内部拼接的sql进行处理，**会将这部分sql前后的逗号给去掉并在前面加上set**。

当传入id和age的时候，生成的sql：

UPDATE t\_user SET age = ? where id = ?

### trim元素

这个元素的功能比较强大，先看一下他的语法：

<trim prefix="" prefixOverrides="" suffix="" suffixOverrides="">  
</trim>

trim元素内部可以包含各种动态sql，如where、chose、sql等各种元素，使用trim包含的元素，mybatis处理过程：

1. 先对trim内部的sql进行拼接，比如这部分sql叫做sql1
2. 将sql1字符串前面的部分中包含trim的prefixOverrides指定的部分给去掉，得到sql2
3. 将sql2字符串后面的部分中包含trim的suffixOverrides指定的部分给去掉，得到sql3
4. 在sql3前面追加trim中prefix指定的值，得到sql4
5. 在sql4后面追加trim中suffix指定的值，得到最终需要拼接的sql5

了解了这个过程之后，说明可以通过trim来代替where和set，我们使用trim来改造一下案例1，如下：

<select id="getList1" resultType="com.javacode2018.chat05.demo8.model.UserModel" parameterType="map">  
 SELECT id,name,age FROM t\_user  
 <trim prefix="where" prefixOverrides="and|or">  
 <if test="id!=null">  
 AND id = #{id}  
 </if>  
 <if test="name!=null and name.toString()!=''">  
 AND name = #{name}  
 </if>  
 <if test="age!=null">  
 AND age = #{age}  
 </if>  
 </trim>  
</select>

**注意上面的prefixOverrides的值的写法，如果有多个需要覆盖的之间用|进行分割，suffixOverrides写法和prefixOverrides的写法类似。**

我们在用trim来改造一下上面的update中的，如下：

<update id="update1" parameterType="com.javacode2018.chat05.demo8.model.UserModel">  
 UPDATE t\_user  
 <trim prefix="SET" prefixOverrides="," suffixOverrides=",">  
 <if test="name!=null">  
 name = #{name},  
 </if>  
 <if test="age!=null">  
 age = #{age},  
 </if>  
 </trim>  
 <where>  
 <if test="id!=null">  
 AND id = #{id}  
 </if>  
 </where>  
</update>

上面的prefixOverrides和suffixOverrides都设置的是逗号，表示trim内部的sql前后的逗号会被去掉，最后会在前面拼接一个prefix指定的set。

大家有兴趣的可以去看一下trim的java实现，代码下面这个类中：

org.apache.ibatis.scripting.xmltags.TrimSqlNode

实际上where和set的实现是继承了TrimSqlNode，where对应的java代码：

public class WhereSqlNode extends TrimSqlNode {  
  
 private static List<String> prefixList = Arrays.asList("AND ","OR ","AND\n", "OR\n", "AND\r", "OR\r", "AND\t", "OR\t");  
  
 public WhereSqlNode(Configuration configuration, SqlNode contents) {  
 super(configuration, contents, "WHERE", prefixList, null, null);  
 }  
  
}

set对应的 java代码：

public class SetSqlNode extends TrimSqlNode {  
  
 private static final List<String> COMMA = Collections.singletonList(",");  
  
 public SetSqlNode(Configuration configuration,SqlNode contents) {  
 super(configuration, contents, "SET", COMMA, null, COMMA);  
 }  
  
}

最后都是依靠TrimSqlNode来实现的。

### foreach元素

相当于java中的循环，可以用来遍历数组、集合、map等。

#### 语法

<foreach collection="需要遍历的集合" item="集合中当前元素" index="" open="" separator="每次遍历的分隔符" close="">  
动态sql部分  
</foreach>

* collection：可以是一个List、Set、Map或者数组
* item：集合中的当前元素的引用
* index：用来访问当前元素在集合中的位置
* separator：各个元素之间的分隔符
* open和close用来配置最后用什么前缀和后缀将foreach内部所有拼接的sql给包装起来。

#### 案例：in多值查询

我们对案例1做个改造，map中支持放入用户的id列表（ArrayList），对应的key为idList，然后支持多个用户id查询，此时我们需要用in来查询，实现如下：

<select id="getList1" resultType="com.javacode2018.chat05.demo8.model.UserModel" parameterType="map">  
 SELECT id,name,age FROM t\_user  
 <where>  
 <if test="id!=null">  
 AND id = #{id}  
 </if>  
 <if test="name!=null and name.toString()!=''">  
 AND name = #{name}  
 </if>  
 <if test="age!=null">  
 AND age = #{age}  
 </if>  
 <if test="idList!=null and idList.size()>=1">  
 <foreach collection="idList" item="id" open="AND id in (" separator="," close=")">  
 #{id}  
 </foreach>  
 </if>  
 </where>  
</select>

大家看一下上面idList那部分判断，判断这个参数不为空，并且size()大于1，表示这个集合不为空，然后会走if元素内部的foreach元素。

比如我们传递的idList对应的是[1,2]，最后产生的sql如下：

SELECT id,name,age FROM t\_user WHERE id in ( ? , ? )

#### 案例：批量插入

传入UserModel List集合，使用foreach实现批量插入，如下：

<insert id="insertBatch" parameterType="list">  
 INSERT INTO t\_user (id,name,age) VALUES  
 <foreach collection="collection" separator="," item="item">  
 (#{item.id}, #{item.name}, #{item.age})  
 </foreach>  
</insert>

测试用例

com.javacode2018.chat05.demo8.Demo8Test#insertBatch  
  
@Test  
public void insertBatch() throws IOException {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 List<UserModel> userModelList = new ArrayList<>();  
 for (int i = 0; i < 3; i++) {  
 userModelList.add(UserModel.builder().id(10 + i).name("mybatis-" + i).age(20 + i).build());  
 }  
 int count = mapper.insertBatch(userModelList);  
 log.info("{}", count);  
 }  
}

传入了3个用户信息，运行输出

39:52.241 [main] DEBUG c.j.c.d.m.UserMapper.insertBatch - ==> Preparing: INSERT INTO t\_user (id,name,age) VALUES (?, ?, ?) , (?, ?, ?) , (?, ?, ?)   
39:52.317 [main] DEBUG c.j.c.d.m.UserMapper.insertBatch - ==> Parameters: 10(Integer), mybatis-0(String), 20(Integer), 11(Integer), mybatis-1(String), 21(Integer), 12(Integer), mybatis-2(String), 22(Integer)  
39:52.327 [main] DEBUG c.j.c.d.m.UserMapper.insertBatch - <== Updates: 3  
39:52.327 [main] INFO c.j.chat05.demo8.Demo8Test - 3

### sql/include元素

这两2个元素一般进行配合使用，可以实现代码重用的效果。

sql元素可以用来定义一段动态sql，语法如下：

<sql id="sql片段id">  
各种动态sql  
</sql>

其他地方需要使用的时候需要通过include关键字进行引入：

<include refid="需要引入的sql片段的id"/>

**注意：refid值的写法，refid的值为mapper xml的namespace的值.sql的id，如果在同一个mapper中，namespace可以省略，直接写对应的sql的id就可以了，**如：

<include refid="findSql"/>  
<include refid="com.javacode2018.chat05.demo8.mapper.UserMapper.findSql"/>

来个案例

下面定义2个查询，他们的查询条件一样，最后将条件抽出来用sql元素定义了一个片段，然后进行共用。

<sql id="findSql">  
 <where>  
 <if test="id!=null">  
 AND id = #{id}  
 </if>  
 <if test="name!=null and name.toString()!=''">  
 AND name = #{name}  
 </if>  
 <if test="age!=null">  
 AND age = #{age}  
 </if>  
 <if test="idList!=null and idList.size()>=1">  
 <foreach collection="idList" item="id" open="AND id in (" separator="," close=")">  
 #{id}  
 </foreach>  
 </if>  
 </where>  
</sql>  
  
<select id="getList1" resultType="com.javacode2018.chat05.demo8.model.UserModel" parameterType="map">  
 SELECT id,name,age FROM t\_user  
 <include refid="com.javacode2018.chat05.demo8.mapper.UserMapper.findSql" />  
</select>  
  
<select id="getList1Count" resultType="com.javacode2018.chat05.demo8.model.UserModel" parameterType="map">  
 SELECT count(\*) FROM t\_user  
 <include refid="findSql" />  
</select>

### bind元素

bind元素允许我们通过ognl表达式在上下文中自定义一个变量，最后在动态sql中可以使用这个变量。

#### 语法

<bind name="变量名称" value="ognl表达式">

#### 案例

对sql、include中的案例进行扩展，添加一个按照用户名模糊查询，用户名在map中对应的key为likeName，主要修改上面sql片段部分，在sql中加入下面部分：

<if test="likeName!=null and likeName.trim()!=''">  
 <bind name="nameLike" value="'%'+likeName.trim()+'%'" />  
 AND name like #{nameLike}  
</if>

先判断传入的参数likeName是否不为空字符串，然后使用bind元素创建了一个变量nameLike，值为'%'+likeName.trim()+'%'。

对应的测试用例：

com.javacode2018.chat05.demo8.Demo8Test#getModelList  
@Test  
public void getModelList() throws IOException {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 Map<String, Object> paramMap = new HashMap<>();  
 paramMap.put("likeName","java");  
 List<UserModel> userModelList = mapper.getList1(paramMap);  
 log.info("{}", userModelList);  
 }  
}

运行输出：

06:25.633 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Preparing: SELECT id,name,age FROM t\_user WHERE name like ?   
06:25.671 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Parameters: %java%(String)  
06:25.690 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - <== Total: 1  
06:25.691 [main] INFO c.j.chat05.demo8.Demo8Test - [UserModel(id=1, name=马士兵老师, age=31)]

注意输出中的第二部分，参数的值为%java%。

### #和$

#和$一般都是结合变量来使用，如：#{}、${}这种来进行使用。

**#{}:为参数占位符？，即sql预编译，相当于使用jdbc中的PreparedStatement中的sql占位符，可以防止sql注入**

**${}:为字符串替换， 即字符串拼接，不能访问sql注入。**

#{}的用法上面已经有很多案例了，此处我们来一个${}的案例。

下面通过orderSql变量传入任意的排序sql，如下：

<select id="getList1" resultType="com.javacode2018.chat05.demo8.model.UserModel" parameterType="map">  
 SELECT id,name,age FROM t\_user  
 <if test="orderSql">  
 ${orderSql}  
 </if>  
</select>

传入值：

orderSql = "order by id asc,age desc"

最后运行产生的sql如下：

20:32.138 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Preparing: SELECT id,name,age FROM t\_user order by id asc,age desc   
20:32.173 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Parameters:   
20:32.196 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - <== Total: 6  
20:32.197 [main] INFO c.j.chat05.demo8.Demo8Test - [UserModel(id=1, name=马士兵老师, age=31), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=10, name=mybatis-0, age=20), UserModel(id=11, name=mybatis-1, age=21), UserModel(id=12, name=mybatis-2, age=22)]

mybatis会对$包含部分进行sql替换。

### 案例代码

链接：https://pan.baidu.com/s/1vt-MAX3oJOu9gyxZAhKkbg   
提取码：i8op

### MyBatis系列

1. MyBatis系列第1篇：MyBatis未出世之前我们那些痛苦的经历
2. MyBatis系列第2篇：入门篇，带你感受一下mybatis独特的魅力！
3. MyBatis系列第3篇：Mybatis使用详解（1）
4. MyBatis系列第4篇：Mybatis使用详解（2）
5. Mybatis系列第5篇：Mapper接口多种方式传参详解、原理、源码解析
6. Mybatis系列第6篇：恕我直言，mybatis增删改你未必玩得转!
7. Mybatis系列第7篇：各种查询详解
8. Mybatis系列第8篇：自动映射，使用需谨慎！
9. Mybatis系列第9篇：延迟加载、鉴别器、继承怎么玩？

## Mybatis系列第12篇：掌握缓存为查询提速!

### 什么是缓存？

缓存就是存储数据的一个地方（称作：Cache），当程序要读取数据时，会首先从缓存中获取，有则直接返回，否则从其他存储设备中获取，**缓存最重要的一点就是从其内部获取数据的速度是非常快的，通过缓存可以加快数据的访问速度。比如我们从db中获取数据，中间需要经过网络传输耗时，db server从磁盘读取数据耗时等，如果这些数据直接放在jvm对应的内存中，访问是不是会快很多。**

### mybatis中的缓存

通常情况下mybatis会访问数据库获取数据，中间涉及到网络通信，数据库从磁盘中读取数据，然后将数据返回给mybatis，总的来说耗时还是挺长的，mybatis为了加快数据查询的速度，在其内部引入了缓存来加快数据的查询速度。

mybatis中分为一级缓存和二级缓存。

一级缓存是SqlSession级别的缓存，在操作数据库时需要构造 sqlSession对象，在对象中有一个数据结构（HashMap）用于存储缓存数据，不同的sqlSession之间的缓存数据区域（HashMap）是互相不影响的。

二级缓存是mapper级别的缓存，多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的。

下面我们详细说一下一级缓存和二级缓存的各种用法和注意点。

### 一级缓存

**一级缓存是SqlSession级别的缓存，每个SqlSession都有自己单独的一级缓存，多个SqlSession之间的一级缓存是相互隔离的，互不影响，mybatis中一级缓存是默认自动开启的。**

**一级缓存工作原理：在同一个SqlSession中去多次去执行同样的查询，每次执行的时候会先到一级缓存中查找，如果缓存中有就直接返回，如果一级缓存中没有相关数据，mybatis就会去db中进行查找，然后将查找到的数据放入一级缓存中，第二次执行同样的查询的时候，会发现缓存中已经存在了，会直接返回。一级缓存的存储介质是内存，是用一个HashMap来存储数据的，所以访问速度是非常快的。**

#### 一级缓存案例

##### 案例sql脚本

DROP DATABASE IF EXISTS `javacode2018`;  
CREATE DATABASE `javacode2018`;  
  
USE `javacode2018`;  
  
DROP TABLE IF EXISTS t\_user;  
CREATE TABLE t\_user(  
 id int AUTO\_INCREMENT PRIMARY KEY COMMENT '用户id',  
 name VARCHAR(32) NOT NULL DEFAULT '' COMMENT '用户名',  
 age SMALLINT NOT NULL DEFAULT 1 COMMENT '年龄'  
) COMMENT '用户表';  
INSERT INTO t\_user VALUES (1,'马士兵老师',30),(2,'张学友',50),(3,'刘德华',50);

##### 下面是查询用户信息，返回一个list

<select id="getList1" resultType="com.javacode2018.chat05.demo9.model.UserModel" parameterType="map">  
 SELECT id,name,age FROM t\_user  
 <where>  
 <if test="id!=null">  
 AND id = #{id}  
 </if>  
 <if test="name!=null and name.toString()!=''">  
 AND name = #{name}  
 </if>  
 <if test="age!=null">  
 AND age = #{age}  
 </if>  
 </where>  
</select>

##### 对应的mapper接口方法

List<UserModel> getList1(Map<String, Object> paramMap);

##### 测试用例

com.javacode2018.chat05.demo9.Demo9Test#level1CacheTest1  
  
/\*\*  
 \* 一级缓存测试  
 \*  
 \* @throws IOException  
 \*/  
@Test  
public void level1CacheTest1() throws IOException {  
 String mybatisConfig = "demo9/mybatis-config.xml";  
 this.before(mybatisConfig);  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 //第一次查询  
 List<UserModel> userModelList1 = mapper.getList1(null);  
 log.info("{}", userModelList1);  
 //第二次查询  
 List<UserModel> userModelList2 = mapper.getList1(null);  
 log.info("{}", userModelList2);  
 log.info("{}", userModelList1 == userModelList2);  
 }  
}

**上面的代码在同一个SqlSession中去执行了2次获取用户列表信息，2次查询结果分别放在userModelList1和userModelList2，最终代码中也会判断这两个集合是否相等，下面我们运行一下看看会访问几次db？**

##### 运行输出

01:15.312 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Preparing: SELECT id,name,age FROM t\_user   
01:15.340 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Parameters:   
01:15.364 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - <== Total: 3  
01:15.364 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50)]  
01:15.367 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50)]  
01:15.367 [main] INFO c.j.chat05.demo9.Demo9Test - true

从输出中可以看出看到，sql只输出了一次，说明第一次会访问数据库，第二次直接从缓存中获取的，最后输出了一个true，也说明两次返回结果是同一个对象，第二次直接从缓存中获取数据的，加快了查询的速度。

### 清空一级缓存的3种方式

同一个SqlSession中查询同样的数据，mybatis默认会从一级缓存中获取，如果缓存中没有，才会访问db，那么我们如何去情况一级缓存呢，强制让查询去访问db呢？

**让一级缓存失效有3种方式：**

1. **SqlSession中执行增、删、改操作，此时sqlsession会自动清理其内部的一级缓存**
2. **调用SqlSession中的clearCache方法清理其内部的一级缓存**
3. **设置Mapper xml中select元素的flushCache属性值为true，那么执行查询的时候会先清空一级缓存中的所有数据，然后去db中获取数据**

上面方式任何一种都会让当前SqlSession中的以及缓存失效，进而去db中获取数据，下面我们来分别演示这3中情况。

#### 方式1：增删改让一级缓存失效

当执行增删改操时，mybatis会将当前SqlSession一级缓存中的所有数据都清除。

##### 案例代码：

com.javacode2018.chat05.demo9.Demo9Test#level1CacheTest2  
  
/\*\*  
 \* 增删改使一级缓存失效  
 \*  
 \* @throws IOException  
 \*/  
@Test  
public void level1CacheTest2() throws IOException {  
 String mybatisConfig = "demo9/mybatis-config.xml";  
 this.before(mybatisConfig);  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 //第一次查询  
 List<UserModel> userModelList1 = mapper.getList1(null);  
 log.info("{}", userModelList1);  
 //新增一条数据  
 mapper.insert1(UserModel.builder().id(100).name("路人").age(30).build());  
 //第二次查询  
 List<UserModel> userModelList2 = mapper.getList1(null);  
 log.info("{}", userModelList2);  
 log.info("{}", userModelList1 == userModelList2);  
 }  
}

上面同一个SqlSession中执行了3个操作，同样的查询执行了2次，2次查询中间夹了一个插入操作。

##### 运行输出

21:55.097 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Preparing: SELECT id,name,age FROM t\_user   
21:55.135 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Parameters:   
21:55.159 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - <== Total: 3  
21:55.159 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50)]  
21:55.161 [main] DEBUG c.j.c.d.mapper.UserMapper.insert1 - ==> Preparing: INSERT INTO t\_user (id,name,age) VALUES (?,?,?)   
21:55.162 [main] DEBUG c.j.c.d.mapper.UserMapper.insert1 - ==> Parameters: 100(Integer), 路人(String), 30(Integer)  
21:55.165 [main] DEBUG c.j.c.d.mapper.UserMapper.insert1 - <== Updates: 1  
21:55.166 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Preparing: SELECT id,name,age FROM t\_user   
21:55.166 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Parameters:   
21:55.167 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - <== Total: 4  
21:55.168 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=100, name=路人, age=30)]  
21:55.168 [main] INFO c.j.chat05.demo9.Demo9Test - false

从输出中可以看出2次查询都访问了db，并且两次查询的结果是不一样的，两个集合也不相等，插入数据让缓存失效是可以理解的，插入操作可能会改变数据库中的数据，所以如果再从缓存中去获取，可能获取到的数据和db中的数据不一致的情况，mybatis为了避免这种情况，在执行插入操作的时候，会将SqlSession中的一级缓存清空。当然删除和修改也同样会改变db中的数据，如果在同一个SqlSession中去执行删除或者修改数据的时候，mybatis也一样会清除一级缓存中的所有数据，删除和修改大家自己可以写2个例子试试，看看是否也会清理一级缓存中的数据。

#### 方式2：SqlSession.clearCache清理一级缓存

SqlSession.clearCache()方法会将当前SqlSession一级缓存中的所有数据清除。

##### 案例代码：

com.javacode2018.chat05.demo9.Demo9Test#level1CacheTest3  
  
/\*\*  
 \* 调用sqlSession.clearCache()清理一级缓存  
 \*  
 \* @throws IOException  
 \*/  
@Test  
public void level1CacheTest3() throws IOException {  
 String mybatisConfig = "demo9/mybatis-config.xml";  
 this.before(mybatisConfig);  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 //第一次查询  
 List<UserModel> userModelList1 = mapper.getList1(null);  
 log.info("{}", userModelList1);  
 //调用clearCache方法清理当前SqlSession中的缓存  
 sqlSession.clearCache();  
 //第二次查询  
 List<UserModel> userModelList2 = mapper.getList1(null);  
 log.info("{}", userModelList2);  
 log.info("{}", userModelList1 == userModelList2);  
 }  
}

##### 运行输出

31:21.937 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Preparing: SELECT id,name,age FROM t\_user   
31:21.966 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Parameters:   
31:21.985 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - <== Total: 4  
31:21.985 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=100, name=路人, age=30)]  
31:21.988 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Preparing: SELECT id,name,age FROM t\_user   
31:21.988 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Parameters:   
31:21.989 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - <== Total: 4  
31:21.989 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=100, name=路人, age=30)]  
31:21.990 [main] INFO c.j.chat05.demo9.Demo9Test - false

从输出中可以看出，2次同样的查询都访问了db。

#### 方式3：Select元素的flushCache置为true

将Mapper xml中select元素的flushCache属性置为true的时候，每次执行这个select元素对应的查询之前，mybatis会将当前SqlSession中一级缓存中的所有数据都清除。

##### 案例代码

新增一个select元素的查询，将flushCache元素置为true，注意：select元素这个属性的默认值是false。

<select id="getList2" flushCache="true" resultType="com.javacode2018.chat05.demo9.model.UserModel" parameterType="map">  
 SELECT id,name,age FROM t\_user  
 <where>  
 <if test="id!=null">  
 AND id = #{id}  
 </if>  
 <if test="name!=null and name.toString()!=''">  
 AND name = #{name}  
 </if>  
 <if test="age!=null">  
 AND age = #{age}  
 </if>  
 </where>  
</select>

##### 对应测试用例

com.javacode2018.chat05.demo9.Demo9Test#level1CacheTest4  
  
/\*\*  
 \* 将Mapper xml中select元素的flushCache属性置为true的时候，每次执行这个select元素对应的查询之前，mybatis会将当前SqlSession中一级缓存中的所有数据都清除。  
 \*  
 \* @throws IOException  
 \*/  
@Test  
public void level1CacheTest4() throws IOException {  
 String mybatisConfig = "demo9/mybatis-config.xml";  
 this.before(mybatisConfig);  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 //查询1：getList1  
 log.info("查询1 start");  
 log.info("查询1：getList1->{}", mapper.getList1(null));  
 //查询2：getList1  
 log.info("查询2 start");  
 log.info("查询2：getList1->{}", mapper.getList1(null));  
 //查询3：getList2  
 log.info("查询3 start");  
 log.info("查询3：getList2->{}", mapper.getList2(null));  
 //查询4：getList2  
 log.info("查询4 start");  
 log.info("查询4：getList2->{}", mapper.getList2(null));  
 //查询5：getList1  
 log.info("查询5 start");  
 log.info("查询5：getList1->{}", mapper.getList1(null));  
 }  
}

注意上面的代码，代码中有5次查询，第1次、第2次、第5次查询调用的都是getList1，这个查询对应的mapper xml中的select元素的flushCache属性没有设置，默认是false；而第3次和第4次查询调用的是getList2，getList2这个查询对应的mapper xml中的select(id=getList2)，它的flushCache属性设置的是true，说明第3和第4次查询会清空当前一级缓存中所有数据。

**最终效果应该是查询1访问db拿去数据，然后将其丢到一级缓存中，查询2会直接从一级缓存中拿到数据，而查询3走的是getList2，发现flushCache为true，会先清空一级缓存中所有数据，也就是此时查询1放入缓存的数据会被清理掉，然后查询3会访问db获取数据，然后丢到缓存中；而查询4走的是getList2，发现flushCache为true，会先清空缓存，所以3放入一级缓存的数据会被清空，然后导致查询4也会访问db，查询5去一级缓存中查询数据，因为查询1和2放入缓存的数据都被查询3清空了，所以导致查询5发现一级缓存中没有数据，也会访问db去获取数据。**

我们来运行一下看看看看是否和我们分析的一致。

##### 运行输出

20:10.872 [main] INFO c.j.chat05.demo9.Demo9Test - 查询1 start  
20:11.164 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Preparing: SELECT id,name,age FROM t\_user   
20:11.195 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Parameters:   
20:11.216 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - <== Total: 4  
20:11.216 [main] INFO c.j.chat05.demo9.Demo9Test - 查询1：getList1->[UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=100, name=路人, age=30)]  
20:11.218 [main] INFO c.j.chat05.demo9.Demo9Test - 查询2 start  
20:11.218 [main] INFO c.j.chat05.demo9.Demo9Test - 查询2：getList1->[UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=100, name=路人, age=30)]  
20:11.219 [main] INFO c.j.chat05.demo9.Demo9Test - 查询3 start  
20:11.219 [main] DEBUG c.j.c.d.mapper.UserMapper.getList2 - ==> Preparing: SELECT id,name,age FROM t\_user   
20:11.219 [main] DEBUG c.j.c.d.mapper.UserMapper.getList2 - ==> Parameters:   
20:11.222 [main] DEBUG c.j.c.d.mapper.UserMapper.getList2 - <== Total: 4  
20:11.222 [main] INFO c.j.chat05.demo9.Demo9Test - 查询3：getList2->[UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=100, name=路人, age=30)]  
20:11.222 [main] INFO c.j.chat05.demo9.Demo9Test - 查询4 start  
20:11.223 [main] DEBUG c.j.c.d.mapper.UserMapper.getList2 - ==> Preparing: SELECT id,name,age FROM t\_user   
20:11.223 [main] DEBUG c.j.c.d.mapper.UserMapper.getList2 - ==> Parameters:   
20:11.225 [main] DEBUG c.j.c.d.mapper.UserMapper.getList2 - <== Total: 4  
20:11.225 [main] INFO c.j.chat05.demo9.Demo9Test - 查询4：getList2->[UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=100, name=路人, age=30)]  
20:11.225 [main] INFO c.j.chat05.demo9.Demo9Test - 查询5 start  
20:11.225 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Preparing: SELECT id,name,age FROM t\_user   
20:11.225 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Parameters:   
20:11.230 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - <== Total: 4  
20:11.230 [main] INFO c.j.chat05.demo9.Demo9Test - 查询5：getList1->[UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=100, name=路人, age=30)]

大家认真看一下上面的输出，查询1/3/4/5访问了db，查询2从缓存中获取的，和我们上面分析的过程一致。

### 一级缓存使用总结

1. **一级缓存是SqlSession级别的，每个人SqlSession有自己的一级缓存，不同的SqlSession之间一级缓存是相互隔离的**
2. **mybatis中一级缓存默认是自动开启的**
3. **当在同一个SqlSession中执行同样的查询的时候，会先从一级缓存中查找，如果找到了直接返回，如果没有找到会去访问db，然后将db返回的数据丢到一级缓存中，下次查询的时候直接从缓存中获取**
4. **一级缓存清空的3种方式（1：SqlSession中执行增删改会使一级缓存失效；2：调用SqlSession.clearCache方法会使一级缓存失效；3：Mapper xml中的select元素的flushCache属性置为true，那么执行这个查询会使一级缓存失效）**

### 二级缓存

#### 二级缓存的使用

一级缓存使用上存在局限性，必须要在同一个SqlSession中执行同样的查询，一级缓存才能提升查询速度，如果想在不同的SqlSession之间使用缓存来加快查询速度，此时我们需要用到二级缓存了。

二级缓存是mapper级别的缓存，每个mapper xml有个namespace，二级缓存和namespace绑定的，每个namespace关联一个二级缓存，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的。

二级缓存默认是没有开启的，需要我们在mybatis全局配置文件中进行开启：

<settings>  
 <!-- 开启二级缓存 -->  
 <setting name="cacheEnabled" value="true"/>  
</settings>

上面配置好了以后，还需要在对应的mapper xml加上下面配置，表示这个mapper中的查询开启二级缓存：

<cache/>

配置就这么简单。

#### 一二级缓存共存时查询原理

一二级缓存如果都开启的情况下，数据查询过程如下：

1. **当发起一个查询的时候，mybatis会先访问这个namespace对应的二级缓存，如果二级缓存中有数据则直接返回，否则继续向下**
2. **查询一级缓存中是否有对应的数据，如果有则直接返回，否则继续向下**
3. **访问db获取需要的数据，然后放在当前SqlSession对应的二级缓存中，并且在本地内存中的另外一个地方存储一份（这个地方我们就叫TransactionalCache）**
4. **当SqlSession关闭的时候，也就是调用SqlSession的close方法的时候，此时会将TransactionalCache中的数据放到二级缓存中，并且会清空当前SqlSession一级缓存中的数据。**

#### 二级缓存案例

##### mybatis全局配置文件开启二级缓存配置

<settings>  
 <!-- 开启二级缓存 -->  
 <setting name="cacheEnabled" value="true"/>  
</settings>

##### mapper xml中使用cache元素开启二级缓存

<!-- 启用二级缓存 -->  
<cache/>  
  
<select id="getList1" resultType="com.javacode2018.chat05.demo9.model.UserModel" parameterType="map">  
 SELECT id,name,age FROM t\_user  
 <where>  
 <if test="id!=null">  
 AND id = #{id}  
 </if>  
 <if test="name!=null and name.toString()!=''">  
 AND name = #{name}  
 </if>  
 <if test="age!=null">  
 AND age = #{age}  
 </if>  
 </where>  
</select>

##### 测试用例

com.javacode2018.chat05.demo9.Demo9Test#level2CacheTest1  
  
/\*\*  
 \* 二级缓存测试  
 \*  
 \* @throws IOException  
 \*/  
@Test  
public void level2CacheTest1() throws IOException {  
 String mybatisConfig = "demo9/mybatis-config1.xml";  
 this.before(mybatisConfig);  
 for (int i = 0; i < 2; i++) {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 List<UserModel> userModelList1 = mapper.getList1(null);  
 log.info("{}", userModelList1);  
 }  
 }  
}

上面执行了2次查询，每次查询都是新的SqlSession，运行一下看看效果。

##### 执行输出

34:36.574 [main] DEBUG c.j.chat05.demo9.mapper.UserMapper - Cache Hit Ratio [com.javacode2018.chat05.demo9.mapper.UserMapper]: 0.0  
34:36.831 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Preparing: SELECT id,name,age FROM t\_user   
34:36.864 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Parameters:   
34:36.883 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - <== Total: 4  
34:36.883 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=100, name=路人, age=30)]  
34:36.894 [main] DEBUG c.j.chat05.demo9.mapper.UserMapper - Cache Hit Ratio [com.javacode2018.chat05.demo9.mapper.UserMapper]: 0.5  
34:36.895 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=100, name=路人, age=30)]

注意上面第一行日志输出：

Cache Hit Ratio [com.javacode2018.chat05.demo9.mapper.UserMapper]: 0.0

对这行做一个解释：com.javacode2018.chat05.demo9.mapper.UserMapper是上面查询访问的mapper xml的namesapce的值，去这个namespace对应的二级缓存中去查询数据，没有查询到，输出中的0.0表示命中率，这次没有命中，所以命中率为0

然后就去db中访问数据了，会将db中返回的数据放在一级缓存中，第一次运行完毕之后会自动调用SqlSession的close方法，然后db中返回的数据会被丢到二级缓存中，第二次查询的时候就直接从二级缓存中获取到数据返回了，所以第二次查询输出如下：

34:36.894 [main] DEBUG c.j.chat05.demo9.mapper.UserMapper - Cache Hit Ratio [com.javacode2018.chat05.demo9.mapper.UserMapper]: 0.5  
34:36.895 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=100, name=路人, age=30)]

2次查询都去访问了二级缓存，第二次命中了，命中率为1/2=0.5

### 清空或者跳过二级缓存的3种方式

当二级缓存开启的时候，在某个mapper xml中添加cache元素之后，这个mapper xml中所有的查询都默认开启了二级缓存，那么我们如何清空或者跳过二级缓存呢？3种方式如下：

1. **对应的mapper中执行增删改查会清空二级缓存中数据**
2. **select元素的flushCache属性置为true，会先清空二级缓存中的数据，然后再去db中查询数据，然后将数据再放到二级缓存中**
3. **select元素的useCache属性置为true，可以使这个查询跳过二级缓存，然后去查询数据**

下面我们来演示一下每种方式对应的效果。

#### 方式1：增删改会清除二级缓存中的数据

下面我们主要演示一下新增对二级缓存的影响。

##### 案例代码

com.javacode2018.chat05.demo9.Demo9Test#level2CacheTest2  
  
/\*\*  
 \* 增删改会清除二级缓存中的数据  
 \*  
 \* @throws IOException  
 \*/  
@Test  
public void level2CacheTest2() throws IOException {  
 String mybatisConfig = "demo9/mybatis-config1.xml";  
 this.before(mybatisConfig);  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 List<UserModel> userModelList1 = mapper.getList1(null);  
 log.info("{}", userModelList1);  
 }  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 //新增一条数据  
 mapper.insert1(UserModel.builder().id(Integer.valueOf(System.nanoTime() % 100000 + "")).name("路人").age(30).build());  
 }  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 List<UserModel> userModelList1 = mapper.getList1(null);  
 log.info("{}", userModelList1);  
 }  
}

上面使用了3个不同的SqlSession，第一次和第三次都调用了getList1执行查询，中间执行了一个插入操作，mybatis执行插入的时候，会先清除当前namespace对应的二级缓存中的数据，所以上面2次查询最终都会访问db，来运行一下看看效果。

##### 运行输出

23:00.620 [main] DEBUG c.j.chat05.demo9.mapper.UserMapper - Cache Hit Ratio [com.javacode2018.chat05.demo9.mapper.UserMapper]: 0.0  
23:00.900 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Preparing: SELECT id,name,age FROM t\_user   
23:00.924 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Parameters:   
23:00.948 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - <== Total: 3  
23:00.948 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50)]  
23:00.951 [main] DEBUG c.j.c.d.mapper.UserMapper.insert1 - ==> Preparing: INSERT INTO t\_user (id,name,age) VALUES (?,?,?)   
23:00.953 [main] DEBUG c.j.c.d.mapper.UserMapper.insert1 - ==> Parameters: 79600(Integer), 路人(String), 30(Integer)  
23:00.955 [main] DEBUG c.j.c.d.mapper.UserMapper.insert1 - <== Updates: 1  
23:00.959 [main] DEBUG c.j.chat05.demo9.mapper.UserMapper - Cache Hit Ratio [com.javacode2018.chat05.demo9.mapper.UserMapper]: 0.0  
23:00.959 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Preparing: SELECT id,name,age FROM t\_user   
23:00.959 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Parameters:   
23:00.961 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - <== Total: 4  
23:00.961 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=79600, name=路人, age=30)]  
23:00.961 [main] DEBUG c.j.c.d.mapper.UserMapper.insert1 - ==> Preparing: INSERT INTO t\_user (id,name,age) VALUES (?,?,?)   
23:00.962 [main] DEBUG c.j.c.d.mapper.UserMapper.insert1 - ==> Parameters: 94100(Integer), 路人(String), 30(Integer)  
23:00.967 [main] DEBUG c.j.c.d.mapper.UserMapper.insert1 - <== Updates: 1

从输出中可以看出，2次查询都访问了db。上面演示的是插入会清空二级缓存的数据，同样删除和修改也会先清除当前namespace对应的二级缓存中的数据。

#### 方式2：select元素的flushCache属性置为true

当将mapper xml中select元素的flushCache属性置为true，会先清空二级缓存中的数据，然后再去db中查询数据，然后将数据再放到二级缓存中。

##### 代码

<select id="getList2" flushCache="true" resultType="com.javacode2018.chat05.demo9.model.UserModel"  
 parameterType="map">  
 SELECT id,name,age FROM t\_user  
 <where>  
 <if test="id!=null">  
 AND id = #{id}  
 </if>  
 <if test="name!=null and name.toString()!=''">  
 AND name = #{name}  
 </if>  
 <if test="age!=null">  
 AND age = #{age}  
 </if>  
 </where>  
</select>

##### 测试用例

com.javacode2018.chat05.demo9.Demo9Test#level2CacheTest3  
  
/\*\*  
 \* 当将mapper xml中select元素的flushCache属性置为true，会先清空二级缓存中的数据，然后再去db中查询数据，然后将数据再放到二级缓存中  
 \*  
 \* @throws IOException  
 \*/  
@Test  
public void level2CacheTest3() throws IOException {  
 String mybatisConfig = "demo9/mybatis-config1.xml";  
 this.before(mybatisConfig);  
 //先查询2次getList1,getList1第二次会从二级缓存中拿到数据  
 log.info("getList1查询");  
 for (int i = 0; i < 2; i++) {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 List<UserModel> userModelList1 = mapper.getList1(null);  
 log.info("{}", userModelList1);  
 }  
 }  
 //getList2的flushCache为true，所以查询之前会先将对应的二级缓存中的所有数据清空，所以二次都会访问db  
 log.info("getList2查询");  
 for (int i = 0; i < 2; i++) {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 List<UserModel> userModelList1 = mapper.getList2(null);  
 log.info("{}", userModelList1);  
 }  
 }  
  
 //二级缓存中没有getList1需要查找的数据了，所以这次访问getList1会去访问db  
 log.info("getList1查询");  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 List<UserModel> userModelList1 = mapper.getList1(null);  
 log.info("{}", userModelList1);  
 }  
}

##### 运行输出

02:51.560 [main] INFO c.j.chat05.demo9.Demo9Test - getList1查询  
02:51.842 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Preparing: SELECT id,name,age FROM t\_user   
02:51.871 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Parameters:   
02:51.891 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - <== Total: 4  
02:51.892 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=5100, name=路人, age=30)]  
02:51.905 [main] DEBUG c.j.chat05.demo9.mapper.UserMapper - Cache Hit Ratio [com.javacode2018.chat05.demo9.mapper.UserMapper]: 0.5  
02:51.906 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=5100, name=路人, age=30)]  
02:51.906 [main] INFO c.j.chat05.demo9.Demo9Test - getList2查询  
02:51.906 [main] DEBUG c.j.chat05.demo9.mapper.UserMapper - Cache Hit Ratio [com.javacode2018.chat05.demo9.mapper.UserMapper]: 0.3333333333333333  
02:51.907 [main] DEBUG c.j.c.d.mapper.UserMapper.getList2 - ==> Preparing: SELECT id,name,age FROM t\_user   
02:51.907 [main] DEBUG c.j.c.d.mapper.UserMapper.getList2 - ==> Parameters:   
02:51.909 [main] DEBUG c.j.c.d.mapper.UserMapper.getList2 - <== Total: 4  
02:51.909 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=5100, name=路人, age=30)]  
02:51.910 [main] DEBUG c.j.chat05.demo9.mapper.UserMapper - Cache Hit Ratio [com.javacode2018.chat05.demo9.mapper.UserMapper]: 0.5  
02:51.913 [main] DEBUG c.j.c.d.mapper.UserMapper.getList2 - ==> Preparing: SELECT id,name,age FROM t\_user   
02:51.913 [main] DEBUG c.j.c.d.mapper.UserMapper.getList2 - ==> Parameters:   
02:51.914 [main] DEBUG c.j.c.d.mapper.UserMapper.getList2 - <== Total: 4  
02:51.914 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=5100, name=路人, age=30)]  
02:51.915 [main] INFO c.j.chat05.demo9.Demo9Test - getList1查询  
02:51.915 [main] DEBUG c.j.chat05.demo9.mapper.UserMapper - Cache Hit Ratio [com.javacode2018.chat05.demo9.mapper.UserMapper]: 0.4  
02:51.915 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Preparing: SELECT id,name,age FROM t\_user   
02:51.915 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Parameters:   
02:51.917 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - <== Total: 4  
02:51.917 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=5100, name=路人, age=30)]

第一次查询访问db，第二次查询从二级缓存中获取了数据，第3和第4查询访问的是getList2，这个查询会清空二级缓存中的数据，直接去db中查询，查询4执行完毕之后，二级缓存中只有第四次查询的数据，第5次查询去getList1中获取数据，此时二级缓存中没有，所以直接去db中获取了。

#### 方式3：select元素的useCache置为false跳过二级缓存，但是不会清空二级缓存数据

新增一个select3查询，将useCache置为false，如下：

<select id="getList3" useCache="false" resultType="com.javacode2018.chat05.demo9.model.UserModel"  
 parameterType="map">  
 SELECT id,name,age FROM t\_user  
 <where>  
 <if test="id!=null">  
 AND id = #{id}  
 </if>  
 <if test="name!=null and name.toString()!=''">  
 AND name = #{name}  
 </if>  
 <if test="age!=null">  
 AND age = #{age}  
 </if>  
 </where>  
</select>

对应的用例代码

/\*\*  
 \* select元素的useCache置为false跳过二级缓存，但是不会清空二级缓存数据  
 \*  
 \* @throws IOException  
 \*/  
@Test  
public void level2CacheTest4() throws IOException {  
 String mybatisConfig = "demo9/mybatis-config1.xml";  
 this.before(mybatisConfig);  
 //第一次查询访问getList1，会将数据丢到二级缓存中  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 List<UserModel> userModelList1 = mapper.getList1(null);  
 log.info("{}", userModelList1);  
 }  
  
 //getList3对应的select的useCache为false，会跳过二级缓存，所以会直接去访问db  
 for (int i = 0; i < 2; i++) {  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 List<UserModel> userModelList1 = mapper.getList3(null);  
 log.info("{}", userModelList1);  
 }  
 }  
  
 //下面的查询又去执行了getList1，由于上面的第一次查询也是访问getList1会将数据放在二级缓存中，所以下面的查询会从二级缓存中获取到数据  
 try (SqlSession sqlSession = this.sqlSessionFactory.openSession(true);) {  
 UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
 List<UserModel> userModelList1 = mapper.getList1(null);  
 log.info("{}", userModelList1);  
 }  
}

注意上面有4次查询，第一次查询访问getList1，会将数据丢到二级缓存中，而第二三次查询访问的是getList3，getList3对应的select的useCache为false，会跳过二级缓存，所以会直接去访问db，第四次查询也是访问getList1，由于第一次查询也是访问getList1会将数据放在二级缓存中，所以第4次查询直接从二级缓存中获取到了数据，运行输出：

13:38.454 [main] DEBUG c.j.chat05.demo9.mapper.UserMapper - Cache Hit Ratio [com.javacode2018.chat05.demo9.mapper.UserMapper]: 0.0  
13:38.852 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Preparing: SELECT id,name,age FROM t\_user   
13:38.898 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - ==> Parameters:   
13:38.929 [main] DEBUG c.j.c.d.mapper.UserMapper.getList1 - <== Total: 4  
13:38.930 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=5100, name=路人, age=30)]  
13:38.941 [main] DEBUG c.j.c.d.mapper.UserMapper.getList3 - ==> Preparing: SELECT id,name,age FROM t\_user   
13:38.942 [main] DEBUG c.j.c.d.mapper.UserMapper.getList3 - ==> Parameters:   
13:38.945 [main] DEBUG c.j.c.d.mapper.UserMapper.getList3 - <== Total: 4  
13:38.945 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=5100, name=路人, age=30)]  
13:38.946 [main] DEBUG c.j.c.d.mapper.UserMapper.getList3 - ==> Preparing: SELECT id,name,age FROM t\_user   
13:38.946 [main] DEBUG c.j.c.d.mapper.UserMapper.getList3 - ==> Parameters:   
13:38.952 [main] DEBUG c.j.c.d.mapper.UserMapper.getList3 - <== Total: 4  
13:38.952 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=5100, name=路人, age=30)]  
13:38.957 [main] DEBUG c.j.chat05.demo9.mapper.UserMapper - Cache Hit Ratio [com.javacode2018.chat05.demo9.mapper.UserMapper]: 0.5  
13:38.957 [main] INFO c.j.chat05.demo9.Demo9Test - [UserModel(id=1, name=马士兵老师, age=30), UserModel(id=2, name=张学友, age=50), UserModel(id=3, name=刘德华, age=50), UserModel(id=5100, name=路人, age=30)]

从输出中可以看出前面3次查询都访问db了，最后一次查询访问的是二级缓存命中了数据。4次查询，第1次和第4次会访问二级缓存，中间2次跳过了二级缓存，二级缓存命中了1次，所以最后一次输出的命中率是0.5

### 总结

1. **一二级缓存访问顺序：一二级缓存都存在的情况下，会先访问二级缓存，然后再访问一级缓存，最后才会访问db，这个顺序大家理解一下**
2. **将mapper xml中select元素的flushCache属性置为true，最终会清除一级缓存所有数据，同时会清除这个select所在的namespace对应的二级缓存中所有的数据**
3. **将mapper xml中select元素的useCache置为false，会使这个查询跳过二级缓存**
4. **总体上来说使用缓存可以提升查询效率，这块知识掌握了，大家可以根据业务自行选择**

### 案例代码

链接：https://pan.baidu.com/s/1vt-MAX3oJOu9gyxZAhKkbg   
提取码：i8op

### MyBatis系列

1. MyBatis系列第1篇：MyBatis未出世之前我们那些痛苦的经历
2. MyBatis系列第2篇：入门篇，带你感受一下mybatis独特的魅力！
3. MyBatis系列第3篇：Mybatis使用详解（1）
4. MyBatis系列第4篇：Mybatis使用详解（2）
5. Mybatis系列第5篇：Mapper接口多种方式传参详解、原理、源码解析
6. Mybatis系列第6篇：恕我直言，mybatis增删改你未必玩得转!
7. Mybatis系列第7篇：各种查询详解
8. Mybatis系列第8篇：自动映射，使用需谨慎！
9. Mybatis系列第9篇：延迟加载、鉴别器、继承怎么玩？
10. Mybatis系列第10篇：动态SQL，这么多种你都会？
11. Mybatis系列第11篇：类型处理器，这个你得会玩