

MyBatis笔记

目录

第一章、初始MyBatis

Java主流框架介绍

MyBatis简介

Mybatis的运行环境（jar包）下载

Mybatis的功能架构分为三层：

Mybatis原理图：

MyBatis的常用的API及方法：

原生态jdbc程序（单独使用jdbc开发）问题总结

持久化与ORM

什么是ORM？

什么是持久化？

为什么要做持久化和ORM设计？

ORM解决方案包含下面四个部分

Mybatis解决jdbc编程的问题

MyBatis框架的优缺点

mybatis与hibernate不同

MyBatis 和 Hibernate 本质区别和应用场景

MyBatis环境搭建步骤

MyBatis基本要素

MyBatis四大核心组件

SqlSessionFactoryBuilder

SqlSessionFactory

SqlSession

SqlSession的两种使用方式

selectOne和selectList区别：

Mapper动态代理方式

开发规范

Mapper接口开发需要遵循以下规范：

Mapper.xml(映射文件)

Mapper.java(接口文件)

加载UserMapper.xml文件

mybatis-config.xml配置文件元素

properties（属性）

settings（全局配置参数）

typeAliases（类型别名）（重点）

typeHandlers（类型处理器）

objectFactory（对象工厂）

mappers（映射器）

environments (配置环境)

注意这里的关键点:

第二章、SQL映射文件

SQL 映射文件有很少的几个顶级元素 (按照它们应该被定义的顺序) :

select

parameterType(输入类型)

- 1、#{ } 与 \${ }
- 2、传递简单类型
- 3、传递pojo对象(实体类)

resultType(输出类型)

- 1、输出简单类型
 - 2、输出pojo对象
 - 3、输出pojo列表
 - 4、输出hashmap
-

resultMap

自动映射

insert

insert, update 和 delete 语句的示例:

使用@Param注解传递多个参数

缓存 (cache)

什么是查询缓存？

为什么要用缓存？

一级缓存

二级缓存

第三章、动态SQL

什么是动态sql？

MyBatis中用于实现动态SQL的元素主要有：

使用if+where实现多条件查询

使用if+trim实现对条件查询

使用if+set实现更新操作

使用if+trim实现更新操作

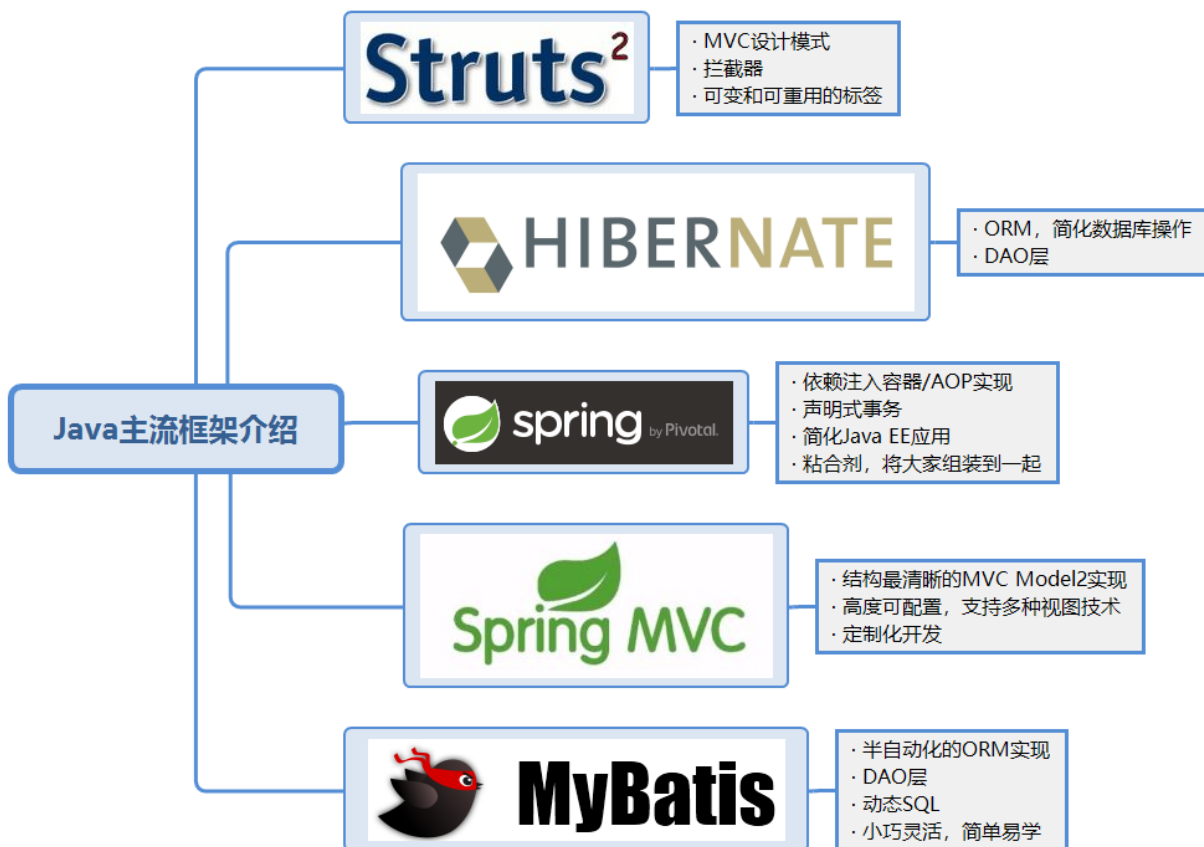
choose, when, otherwise （类似于Switch）

foreach

SQL片段

第一章、初始MyBatis

Java主流框架介绍



MyBatis简介

MyBatis的前身叫iBatis, 是一个持久层的框架, 是apache下的顶级项目。本是apache的一个开源项目, 2010年这个项目由apache software foundation 迁移到了google code, 并且改名为MyBatis。**MyBatis 是支持普通SQL查询, 存储过程和高级映射的优秀持久层框架。**

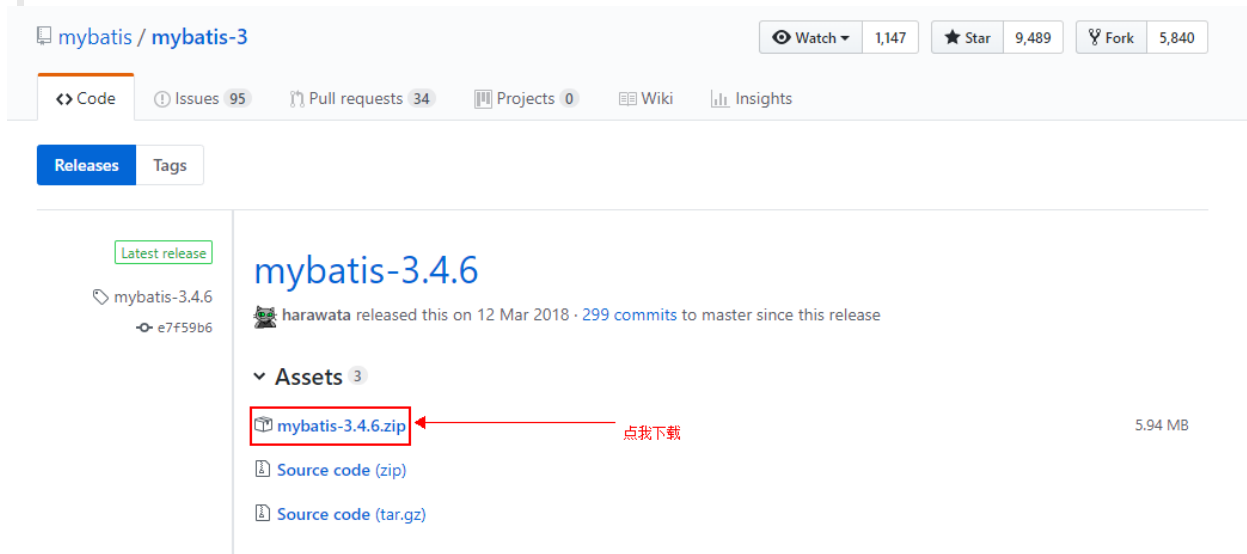
MyBatis 是一个可以自定义SQL、存储过程和高级映射的持久层框架, MyBatis可以将向 preparedStatement 中的输入参数自动进行输入映射, 将查询结果集灵活映射成java对象。(输出映射)。MyBatis 移除了大部分的JDBC代码、手工设置参数和结果集重获。MyBatis 只使用简单的XML 和注解来配置和映射基本数据类型、Map 接口和POJO 到数据库记录。相对Hibernate和Apache OJB等“一站式” ORM解决方案而言, Mybatis 是一种“半自动化”的ORM实现。

需要使用的Jar包:

mybatis-3.4.6.jar (mybatis核心包)
mybatis-spring-1.0.0.jar(与Spring结合包)

Mybatis的运行环境 (jar包) 下载

mybaits的代码由github.com管理, 地址: <https://github.com/mybatis/mybatis-3/releases>



下载完成mybatis-3.4.6.zip解压即可

mybatis-3.4.6.jar: mybatis的核心包

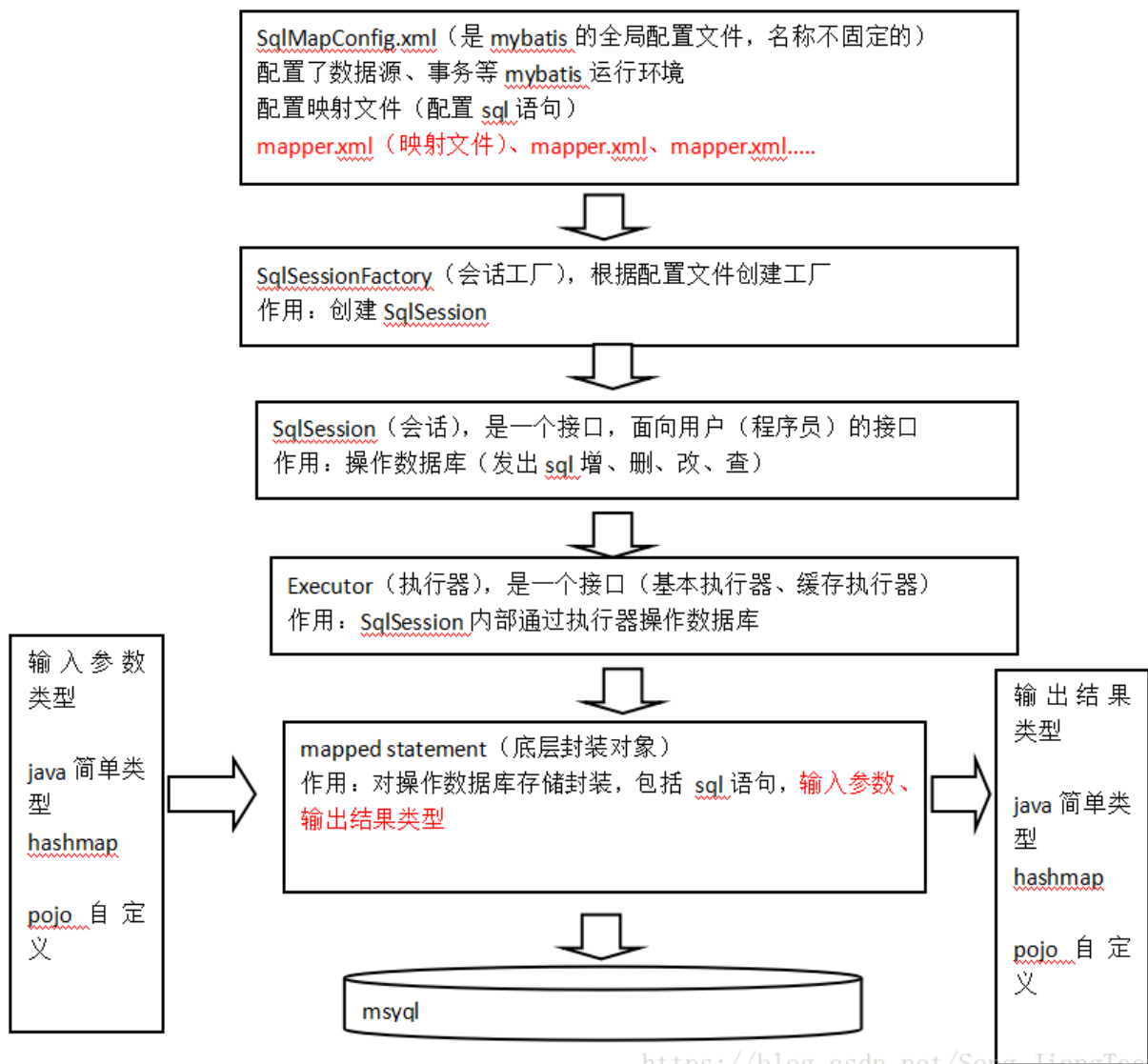
lib: mybatis的依赖包

mybatis-3.4.6.pdf: mybatis使用手册

Mybatis的功能架构分为三层:

- 1. API接口层:** 提供给外部使用的接口API, 开发人员通过这些本地API来操纵数据库。接口层一接收到调用请求就会调用数据处理层来完成具体的数据处理。
- 2. 数据处理层:** 负责具体的SQL查找、SQL解析、SQL执行和执行结果映射处理等。它主要的目的是根据调用的请求完成一次数据库操作。
- 3. 基础支撑层:** 负责最基础的功能支撑, 包括连接管理、事务管理、配置加载和缓存处理, 这些都是共用的东西, 将他们抽取出来作为最基础的组件。为上层的数据处理层提供最基础的支撑。

Mybatis原理图:



https://blog.csdn.net/Song_JiangTan

MyBatis的常用的API及方法:

1. **SqlSessionFactoryBuilder** 该对象根据MyBatis配置文件 **SqlMapConfig.xml** 构建 **SqlSessionFactory** 实例。
2. **SqlSessionFactory** 每一个MyBatis的应用程序都以一个 **SqlSessionFactory** 对象为核心。对该对象负责创建 **SqlSession** 对象实例。
3. **SqlSession** 该对象包含了所有执行SQL操作的方法，用于执行SQL操作的方法，用于执行以已映射的SQL语句。

原生态jdbc程序（单独使用jdbc开发）问题总结

先来看一部分代码：

```
1 Public static void main(String[] args) {  
2     Connection connection = null;  
3     PreparedStatement preparedStatement = null;
```

```

4  ResultSet resultSet = null;
5  try {
6      //加载数据库驱动
7      Class.forName("com.mysql.jdbc.Driver");
8      //通过驱动管理类获取数据库链接
9      connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8", "root", "mysql");
10     //定义sql语句 ?表示占位符
11     String sql = "select * from user where username = ?";
12     //获取预处理statement
13     preparedStatement = connection.prepareStatement(sql);
14     //设置参数，第一个参数为sql语句中参数的序号（从1开始），第二个参数为设置的参数值
15     preparedStatement.setString(1, "王五");
16     //向数据库发出sql执行查询，查询出结果集
17     resultSet = preparedStatement.executeQuery();
18     //遍历查询结果集
19     while(resultSet.next()){
20         System.out.println(resultSet.getString("id")+" "+resultSet.getString("username"));
21     }
22     } catch (Exception e) {
23         e.printStackTrace();
24     }finally{
25         //释放资源
26         if(resultSet!=null){
27             try {
28                 resultSet.close();
29             } catch (SQLException e) {
30                 e.printStackTrace();
31             }
32         }
33         if(preparedStatement!=null){
34             try {
35                 preparedStatement.close();
36             } catch (SQLException e) {
37                 // TODO Auto-generated catch block
38                 e.printStackTrace();
39             }

```



```
40 }
41 if(connection!=null){
42     try {
43         connection.close();
44     } catch (SQLException e) {
45         e.printStackTrace();
46     }
47 }
48 }
49 }
```

上面代码有如下几个问题：

- 数据库连接，使用时创建，不使用就关闭，对数据库进行频繁连接开启和关闭，造成数据库资源的浪费

解决：使用数据库连接池管理数据库连接

- 将sql 语句硬编码到Java代码中，如果sql语句修改，需要对java代码重新编译，不利于系统维护

解决：将sql语句设置在xml配置文件中，即使sql变化，也无需重新编译

- 向preparedStatement中设置参数，对占位符位置和设置参数值，硬编码到Java文件中，不利于系统维护

解决：将sql语句及占位符，参数全部配置在xml文件中

- 从resultSet中遍历结果集数据时，存在硬编码，将获取表的字段进行硬编码，不利于系统维护。

解决：将查询的结果集，自动映射成java对象

持久化与ORM

什么是ORM?

ORM 对象关系映射(Object Relational Mapping, 简称ORM, 或O/RM, 或O/R mapping), 是一种程序技术, 用于实现面向对象编程语言里不同类型系统的数据之间的转换。从效果上说, 它其实是创建了一个可在编程语言里使用的--“虚拟对象数据库”。

对象关系映射, 是随着面向对象的软件开发方法发展而产生的。用来把对象模型表示的对象映射到基于S Q L 的关系模型数据库结构中去。这样, 我们在具体的操作实体对象的时候

候，就不需要再去和复杂的 SQL 语句打交道，只需简单的操作实体对象的属性和方法

什么是持久化？

持久（Persistence），即把数据（如内存中的对象）保存到可永久保存的存储设备中（如磁盘）。持久化的主要应用是将内存中的数据存储到关系型的数据库中，当然也可以存储在磁盘文件中、XML数据文件中等等。

为什么要做持久化和ORM设计？

在目前的企业应用系统设计中，MVC，即 Model（模型） - View（视图） - Control（控制）为主要的系统架构模式。MVC 中的 Model 包含了复杂的业务逻辑和数据逻辑，以及数据存取机制（如 JDBC的连接、SQL生成和Statement创建、还有ResultSet结果集的读取等）等。将这些复杂的业务逻辑和数据逻辑分离，以将系统的紧耦合关系转化为松耦合关系（即解耦合），是降低系统耦合度迫切要做的，也是持久化要做的工作。MVC 模式实现了架构上将表现层（即View）和数据处理层（即Model）分离的解耦合，而**持久化的设计则实现了数据处理层内部的业务逻辑和数据逻辑分离的解耦合**。而 ORM 作为持久化设计中的最重要也最复杂的技术，也是目前业界热点技术。

ORM解决方案包含下面四个部分

1. 在持久化对象上执行基本的增、删、改、查操作。
2. 对持久化对象提供一种查询语言或API
3. 对象映射工具
4. 提供与事物对象交互、执行检查、延迟加载等其他优化功能

Mybatis解决jdbc编程的问题

1、数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。

解决：在SqlMapConfig.xml中配置数据链接池，使用连接池管理数据库链接。

2、Sql语句写在代码中造成代码不易维护，实际应用sql变化的可能较大，sql变动需要改变java代码。

解决：将Sql语句配置在XXXXmapper.xml文件中与java代码分离。

3、向sql语句传参数麻烦，因为sql语句的where条件不一定，可能多也可能少，占位符需要和参数一一对应。

解决：Mybatis自动将java对象映射至sql语句，通过statement中的parameterType定义输入参数的类型。

4、对结果集解析麻烦，sql变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成pojo对象解析比较方便。

解决：Mybatis自动将sql执行结果映射至java对象，通过statement中的resultType定义输出结果的类型。

MyBatis框架的优缺点

MyBatis框架的优点：

1. 与JDBC相比，减少了50%以上的代码量。
2. MyBatis是最简单的持久化框架，小巧并且简单易学。
3. MyBatis相当灵活，不会对应用程序或者数据库的现有设计强加任何影响，SQL写在XML里，从程序代码中彻底分离，降低耦合度，便于统一管理和优化，并可重用。
4. 提供XML标签，支持编写动态SQL语句。
5. 提供映射标签，支持对象与数据库的ORM字段关系映射。

MyBatis框架的缺点：

1. SQL语句的编写工作量较大，尤其是字段多、关联表多时，更是如此，对开发人员编写SQL语句的功底有一定要求。
2. SQL语句依赖于数据库，导致数据库移植性差，不能随意更换数据库。

mybatis与hibernate不同

Mybatis和hibernate不同，它不完全是一个ORM框架，因为MyBatis需要程序员自己编写Sql语句，不过mybatis可以通过XML或注解方式灵活配置要运行的sql语句，并将java对象和sql语句映射生成最终执行的sql，最后将sql执行的结果再映射生成java对象。

Mybatis学习门槛低，简单易学，程序员直接编写原生态sql，可严格控制sql执行性能，灵活度高，非常适合对关系数据模型要求不高的软件开发，例如互联网软件、企业运营类软件等，因为这类软件需求变化频繁，一旦需求变化要求成果输出迅速。但是灵活的前提是mybatis无法做到数据库无关性，如果需要实现支持多种数据库的软件则需要自定义多套sql映射文件，工作量大。

Hibernate对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件（例如需求固定的定制化软件）如果用hibernate开发可以节省很多代码，提高效率。但是Hibernate的学习门槛高，要精通门槛更高，而且怎么设计O/R映射，在性能和对象模型之间如何权衡，以及怎样用好Hibernate需要具有很强的经验和能力才行。

总之，按照用户的需求在有限的资源环境下只要能做出维护性、扩展性良好的软件架构都是好架构，所以框架只有适合才是最好。

MyBatis 和 Hibernate 本质区别和应用场景

Hibernate: 是一个标准ORM框架（对象关系映射）。入门门槛较高的，不需要程序写sql，sql语句自动生成了。对sql语句进行优化、修改比较困难的。

应用场景：

适用与需求变化不多的中小型项目，比如：后台管理系统，erp、orm、oa。。

MyBatis: 专注是sql本身，需要程序员自己编写sql语句，sql修改、优化比较方便。mybatis是一个不完全的ORM框架，虽然程序员自己写sql，mybatis也可以实现映射（输入映射、输出映射）。

应用场景：

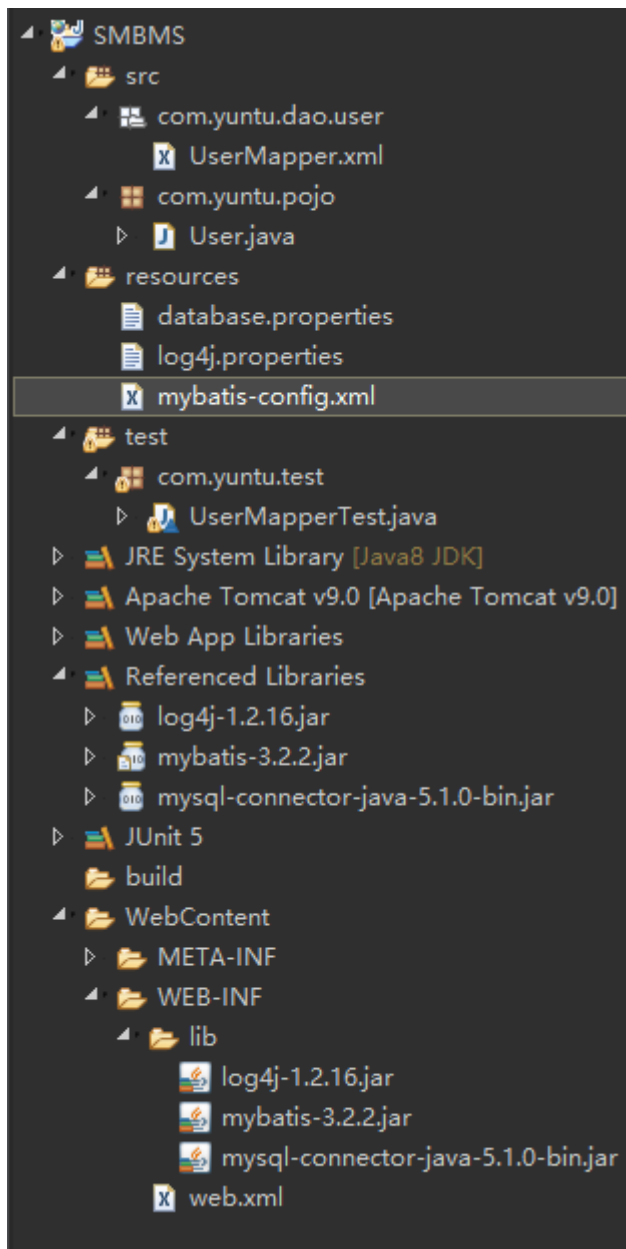
MyBatis专注于SQL本身，是一个足够灵活的DAO层解决方案。对性能的要求很高，或者需求变化较多的项目，如互联网项目，MyBatis将是不错的选择。

MyBatis环境搭建步骤

所需jar包：

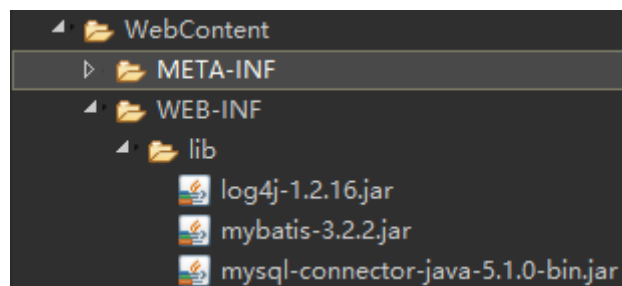
log4j-1.2.16.jar
mybatis-3.2.2.jar
mysql-connector-java-5.1.0-bin.jar

项目结构：



1、新建JavaWeb项目

2、导入jar包，WebContent/WEB-INF/lib文件夹下，然后全部选中所有jar包，右键build path构建一下即可。



如果你是使用maven构建工程的话，那就对应在pom.xml文件中添加对应依赖即可。

3、项目 → 右键 → new → source folder 创建包资源管理器 → 这里创建的是 resources

4、在 **resources** 文件夹下创建文件名为 log4j.properties 的文件 复制如下代码，保存：

```
1 # Global logging configuration
2 # 开发环境下，日志级别要设置成DEBUG或者ERROR
3 log4j.rootLogger=DEBUG, stdout
4 # Console output...
5 log4j.appender.stdout=org.apache.log4j.ConsoleAppender
6 log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
7 log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

mybatis默认使用log4j作为输出日志信息。

5、在 **resources** 文件夹下创建文件名为 mybatis-config.xml 的核心配置文件：

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE configuration
3 PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4 "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6 <configuration>
7 <!-- 和spring整合后 environments配置将废除-->
8 <!-- 引入 database.properties 文件 -->
9 <properties resource="database.properties"></properties>
10 <!-- 配置log4j，配置后后台会自动对SQL语句输入 -->
11 <settings>
12 <setting name="logImpl" value="LOG4J"/>
13 </settings>
14
15 <environments default="development">
16 <environment id="development">
17 <!-- 配置事务管理，此处采用JDBC的事务管理 -->
18 <transactionManager type="JDBC"/>
19 <!-- 数据库连接池，配置数据库连接信息 -->
20 <dataSource type="POOLED">
21 <property name="driver" value="${driver}"/>
22 <property name="url" value="${url}"/>
```

```

23 <property name="username" value="${user}"/>
24 <property name="password" value="${password}"/>
25 </dataSource>
26 </environment>
27 </environments>
28
29 <mappers>
30 <!-- resource: dao层映射的路径 -->
31 <mapper resource="com/yuntu/dao/user/UserMapper.xml"/>
32 </mappers>
33 </configuration>

```

mybatis-config.xml是mybatis核心配置文件，上边文件的配置内容为数据源、事务管理。

6、在 **resources** 文件夹下创建文件名为 database.properties 的数据库配置文件

```

1 driver=com.mysql.jdbc.Driver
2 #在和mysql传递数据的过程中，使用unicode编码格式，并且字符集设置为utf-8
3 url=jdbc:mysql://127.0.0.1:3306/SMBMS?useUnicode=true&characterEncoding=utf-8
4 user=root
5 password=123456

```

7、创建实体类 (POJO)

8、创建Dao层，新建SQL映射文件 (UserMapper.xml)

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper
3 PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6 <!--
7 namespace: 命名空间，作用为，对sql进行分类化管理，理解sql隔离，
8 注意：使用mapper代理方法开发，namespace有特殊作用
9 -->
10 <mapper namespace="test">

```

```

11 <!--在映射文件中配置sql-->
12 <!--
13 通过select执行数据库查询
14 id: 标识映射文件的sql
15 将sql语句封装到mappedStatement对象中，所以将id称为Statement的id
16 resultType: 指定sql输出结果的所映射的Java对象类型，select指定的resultType表示将单条记录映射成的Java对象
17 -->
18 <!-- 查询用户总数 -->
19 <select id="count" resultType="int">
20 select count(1) as count from smbms_user
21 </select>
22 </mapper>

```

9、创建测试类 (JUnit Test单元测试)

实现思路：

1. 定义mybatis配置文件名字的变量

```

1 String resource = "mybatis-config.xml";
2 InputStream inputStream =
  Resources.getResourceAsStream(resource);

```

2. 创建会话工厂

```

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

```

3. 通过工厂得到SqlSession

```

1 SqlSession sqlSession = sqlSessionFactory.openSession();

```

4. 通过SqlSession操作数据库

```

1 //第一个参数：映射文件中的statement的id，等于namespace+"."+statement的id
2 //第二个参数：指定和映射文件中所匹配的所有parameterType的类型
3 //sqlSession.selectOne()的结果是映射文件中所匹配的resultType类型的对象
4 User user = sqlSession.selectOne("test.findUserId",1);
5 System.out.println(user);
6 //释放资源
7 sqlSession.close();

```


示例:

```
1 package com.yuntu.test;
2 import static org.junit.jupiter.api.Assertions.*;
3 import java.io.IOException;
4 import java.io.InputStream;
5 import org.apache.ibatis.io.Resources;
6 import org.apache.ibatis.session.SqlSession;
7 import org.apache.ibatis.session.SqlSessionFactory;
8 import org.apache.ibatis.session.SqlSessionFactoryBuilder;
9 import org.apache.log4j.Logger;
10 import org.junit.jupiter.api.Test;
11
12 class UserMapperTest {
13     private Logger logger =
14         Logger.getLogger(UserMapperTest.class);
15     @Test
16     void test() {
17         //mybatis配置文件
18         String resource="mybatis-config.xml";
19         SqlSession sqlSession = null;
20         int count=0;//用来接收结果
21         try {
22             //1.根据 Resources (mybatis配置文件名) 获取 mybatis-config.xml
23             //的输入流
24             InputStream is = Resources.getResourceAsStream(resource);
25             //2.创建核心对象 SqlSessionFactory 对象，来完成配置文件的读取
26             SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(is);
27             //3.创建SqlSession对象
28             sqlSession=factory.openSession();
29             //4.调用mapper文件对数据进行操作 ps: 检查mapper是否在mybatis-config.xml中进行配置
30             count = sqlSession.selectOne("com.yuntu.dao.user.UserMapper.count");
31             logger.debug("当前系统一共有"+count+"用户");
32         } catch (IOException e) {
33             e.printStackTrace();
34         }
35     }
36 }
```

```

30 } catch (IOException e) {
31     e.printStackTrace();
32 } finally {
33     sqlSession.close();
34 }
35 }
36 }

```

MyBatis基本要素

MyBatis基本要素包括以下三个部分：

1. MyBatis的核心接口和类：

- SqlSessionFactoryBuilder(构造器)
- SqlSessionFactory (工厂接口)
- SqlSession(会话)

2. mybatis-config.xml 系统核心配置文件

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE configuration
3     PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-config.dtd">
5 <configuration>
6     <properties /> <!--属性：可以配置在Java属性配置文件中-->
7     <settings /> <!--设置：修改MyBatis在运行时的行为方式-->
8     <typeAliases /> <!--类型别名：为Java类型命名别名（简称）-->
9     <typeHandlers /> <!--类型处理器-->
10    <objectFactory /> <!--对象工厂-->
11    <environments> <!--配置环境-->
12        <environment> <!--环境配置-->
13            <transactionManager> </transactionManager> <!--事务管理器-->
14            <dataSource></dataSource> <!--数据源-->
15        </environment>
16    </environments>
17    <databaseIdProvider /> <!--数据库厂商-->
18    <mappers /> <!--映射器-->

```

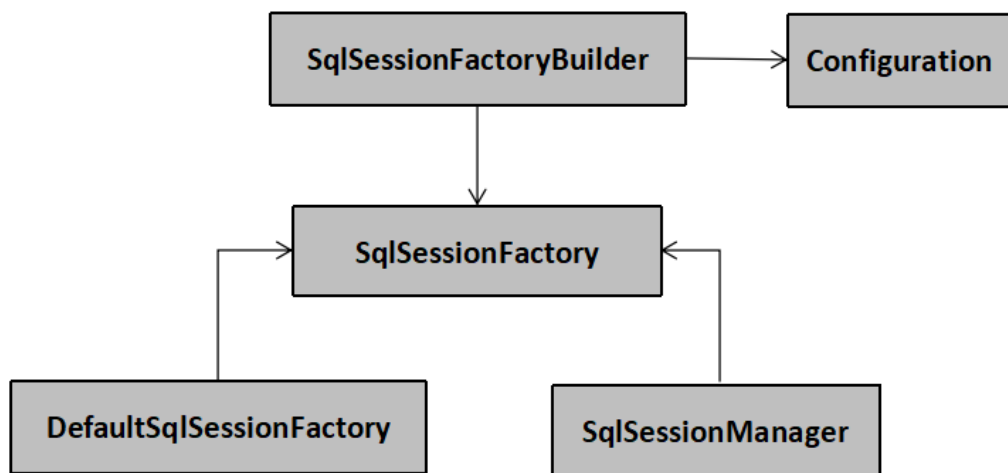
3. mapper.xmlSQL映射文件

MyBatis四大核心组件

核心接口和类包括以下三个部分：

- 1、SqlSessionFactoryBuilder (构造器)
- 2、SqlSessionFactory (工厂接口)
- 3、SqlSession (会话)

三者关系如下：



https://blog.csdn.net/Hello_Peter_Chan

注：

- 1、SqlSessionFactory是我们MyBatis整个应用程序的中心；整个程序都是以SqlSessionFactory的实例为核心的。
- 2、SqlSessionFactory对象是由SqlSessionFactoryBuilder对象创建而来的。
- 3、SqlSessionFactoryBuilder是通过xml配置文件或者configuration类的实例去构造这个对象。然后调用build()方法去构造SqlSessionFactory对象。
- 4、使用SqlSessionFactory对象的openSession()方法来获取SqlSession对象，有了SqlSession对象，我们就可以去进行数据库操作了，因为SqlSession里面包含了以数据库为背景所有执行sql操作的方法。

SqlSessionFactoryBuilder

根据配置或代码生成SqlSessionFactory,采用分步构建的Builder模式，创建成功SqlSessionFactory后就失去了作用。

- 用过即丢，其生命周期只存在于方法体内
- 可重用其来创建多个SqlSessionFactory实例
- 负责构建SqlSessionFactory，并提供多个build方法的重载

```
build(InputStream inputStream, String environment, Properties properties)
build(Reader reader, String environment, Properties properties)
build(Configuration config)

配置信息以三种形式提供给SqlSessionFactory的build方法：
InputStream(字节流)、Reader(字符流)、Configuration(类)
```

SqlSessionFactory

SqlSessionFactory是一个接口，接口中定义了openSession的不同重载方法，SqlSessionFactory的最佳使用范围是整个应用运行期间，一旦创建后可以重复使用，通常以单例模式管理SqlSessionFactory。

■ SqlSessionFactory

- ◆ SqlSessionFactory是每个MyBatis应用的核心
- ◆ 作用：创建SqlSession实例

```
SqlSession session = sqlSessionFactory.openSession(boolean autoCommit);
```

- ◆ 作用域：Application
- ◆ 生命周期与应用的生命周期相同
- ◆ 单例
 - 存在于整个应用运行时，并且同时只存在一个对象实例

注：

1、在使用openSession()方法创建一个SqlSession对象的时候传了一个布尔型的参数

```
SqlSession session = sqlSessionFactory.openSession(boolean autoCommit);
```

作用域：Application
生命周期与应用的生命周期相同
单例

true：关闭事务控制（默认）
false：开启事务控制

问题：获取SqlSessionFactory的代码是否可以优化？

可以使用静态代码块，以保证SqlSessionFactory只被创建一次。

创建一个工具类如下：

```
1 package com.yuntu.util;
2
3 import java.io.IOException;
4 import java.io.InputStream;
```

```
5 import org.apache.ibatis.io.Resources;
6 import org.apache.ibatis.session.SqlSession;
7 import org.apache.ibatis.session.SqlSessionFactory;
8 import org.apache.ibatis.session.SqlSessionFactoryBuilder;
9
10 /**
11  * 创建会话工厂
12  * @author 阿华
13  */
14 public class MybatisUtil {
15     static String resource="mybatis-config.xml";
16     private static SqlSessionFactory factory = null;
17     static {
18         try {
19             //1.根据 Resources 获取 mybatis-config.xml 的输入流
20             InputStream is = Resources.getResourceAsStream(resource);
21             //2.创建核心对象 SqlSessionFactory 对象，来完成配置文件的读取
22             factory = new SqlSessionFactoryBuilder().build(is);
23         } catch (IOException e) {
24             // TODO Auto-generated catch block
25             e.printStackTrace();
26         }
27     }
28     /**
29     * 创建会话工厂
30     * @return
31     */
32     public static SqlSession creatSqlSession() {
33         return factory.openSession();
34     }
35
36     /**
37     * 关闭会话工厂
38     * @param sqlSession
39     */
}
```

```

40 public static void closeSqlSession(SqlSession sqlSession) {
41     if (null != sqlSession) {
42         sqlSession.close();
43     }
44 }
45 }

```

工具类创建完成之后，就可以修改之前的测试代码了，如下：

```

1 package com.yuntu.test;
2
3 import static org.junit.jupiter.api.Assertions.*;
4 import org.apache.ibatis.session.SqlSession;
5 import org.apache.log4j.Logger;
6 import org.junit.jupiter.api.Test;
7 import com.yuntu.util.MybatisUtil;
8
9 class UserMapperTest {
10     private Logger logger =
11         Logger.getLogger(UserMapperTest.class);
12     @Test
13     void test() {
14         String resource="mybatis-config.xml";
15         SqlSession sqlSession = null;
16         int count=0;
17
18         try {
19             //3.创建SqlSession
20             sqlSession=MybatisUtil.creatSqlSession();
21             //4.调用mapper文件对数据进行操作 ps: 检查mapper是否在mybatis-conf
22             g.xml中进行配置
23             count = sqlSession.selectOne("com.yuntu.dao.user.UserMapper.co
24             unt");
25             logger.debug("当前系统一共有"+count+"用户");
26         } catch (Exception e) {
27             e.printStackTrace();
28         } finally {
29             MybatisUtil.closeSqlSession(sqlSession);
30         }
31     }
32 }

```

```
27 }
28 }
29 }
```

SqlSession

SqlSession是一个面向用户的接口， sqlSession中定义了数据库操作方法。

每个线程都应该有它自己的SqlSession实例。SqlSession的实例不能共享使用，它也是线程不安全的。因此最佳的范围是请求或方法范围。绝对不能将SqlSession实例的引用放在一个类的静态字段或实例字段中。

- 包含了自行SQL所需的所有方法
- 对应一次数据库会话，会话结束必须关闭
- 线程级别，不能共享

注意：打开一个 SqlSession；使用完毕就要 **关闭** 它，否则会**报错**。通常把这个关闭操作放到 finally 块中以确保每次都能执行关闭。如下：

```
1 SqlSession session = sqlSessionFactory.openSession();
2 try {
3
4 } finally {
5     session.close();
6 }
```

SqlSession的获取方式：

■ SqlSession的获取方式

```
String resource = "mybatis-config.xml";
InputStream is = Resources.getResourceAsStream(resource);
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(is);
SqlSession sqlSession = factory.openSession();
```

SqlSession的两种使用方式：

■ SqlSession的两种使用方式

- ◆ 通过SqlSession实例直接运行映射的SQL语句
- ◆ 基于Mapper接口方式操作数据

第一种方式：

```
1 count = sqlSession.selectOne("cn.smbms.dao.provider.providerCount");
```

就是直接运行映射的SQL语句。

第二种方式：

基于mapper接口方式来操作数据，也是MyBatis比较推崇的方式。

SqlSession的两种使用方式

id	userCode	userName	userPassword	gender	birthday	phone	address	userRole	createdBy	creationDate	modifyBy	modifyDate
1	admin	系统管理员	1234567		1 1983-10-10	13688889999	北京市海淀区成府路207号	1	1	2013-03-21 16:52:07	(Null) (Null)	
2	liming	李明	0000000		2 1983-12-10	13688884457	北京市东城区前门东大街5	2	1	0000-00-00 00:00:00	(Null) (Null)	
5	hanlubiao	韩路彪	0000000		2 1984-06-05	18567542321	北京市朝阳区北辰中心12-	2	1	2014-12-31 19:52:09	(Null) (Null)	
6	zhanghua	张华	0000000		1 1983-06-15	13544561111	北京市海淀区学院路61号	3	1	2013-02-11 10:51:17	(Null) (Null)	
7	wangyang	王洋	0000000		2 1982-12-31	13444561124	北京市海淀区西二旗辉煌	3	1	2014-06-11 19:09:07	(Null) (Null)	
8	zhaoyan	赵燕	0000000		1 1986-03-07	18098764545	北京市海淀区回龙观小区1	3	1	2016-04-21 13:54:07	(Null) (Null)	
10	sunlei	孙磊	0000000		2 1981-01-04	13387676765	北京市朝阳区管庄新月小	3	1	2015-05-06 10:52:07	(Null) (Null)	
11	sunxing	孙兴	0000000		2 1978-03-12	13367890900	北京市朝阳区建国门南大	3	1	2016-11-09 16:51:17	(Null) (Null)	
12	zhangchen	张晨	0000000		1 1986-03-28	18098765434	朝阳区管庄路口北柏林爱	3	1	2016-08-09 05:52:37	1	2016-04-14 14:15:36
13	dengchao	邓超	0000000		2 1981-11-04	13689674534	北京市海淀区北航家属院1	3	1	2016-07-11 08:02:47	(Null) (Null)	
14	yangguo	杨过	0000000		2 1980-01-01	13388886623	北京市朝阳区北苑家园莱	3	1	2015-02-01 03:52:07	(Null) (Null)	
15	zhaomin	赵敏	0000000		1 1987-12-04	18099897657	北京市昌平区天通苑3区1	2	1	2015-09-12 12:02:12	(Null) (Null)	

需求说明：使用SqlSession的两种使用方式实现用户表的查询操作。

分析：

- 调用 sqlSession.selectList() 执行查询操作
- 调用 sqlSession.getMapper(Mapper.class) 执行DAO接口方法来实
现对数据的查询操作

1. 调用sqlSession.selectList()执行查询操作

首先，在UserMapper.xml文件中增加新的select标签：

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper
3 PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5 <!-- mapper标签里面只有一个属性namespace（其实是本xml文件的路径
6 名），其作用是用来区分其他不同的mapper，来达到全局唯一-->
7
8 <!-- 因为每一个实体类都有一个mapper.xml文件，所以最后会有很多个mappe
9 -->
10
11 <!-- 查询用户表的记录数 -->
12
13 <!-- 每一个子节点元素都会有一个id，id属性是表示该命名空间下的唯一标
14 识符
15
16 resultType属性是指返回的结果类型
17
18 -->
19
20 <select id="count" resultType="int">
21 select count(1) as count from smbms_user
```



```

15 </select>
16
17 <!-- 查询用户列表 -->
18 <select id="getUserList" resultType="cn.smbms.pojo.User">
19     select * from smbms_user
20 </select>
21 </mapper>

```

然后，增加新的单元测试的方法：

```

1 @Test
2 public void testGetuserList() {
3     List<User> userList = null;
4     SqlSession sqlSession = null;
5     try {
6         sqlSession = MyBatisUtil.createSqlSession();
7         // 4、调用mapper文件来对数据进行操作,操作之前必须将mapper文件引入到
            abatis-config.xml中
8         userList = sqlSession.selectList("cn.smbms.dao.user.UserMapper.
            getUserList");
9         // userList = sqlSession.getMapper(UserMapper.class).getUserList
            t();
10
11     } catch (Exception e) {
12         e.printStackTrace();
13     } finally {
14         MyBatisUtil.closeSqlSession(sqlSession);
15     }
16     for (User user:userList) {
17         logger.debug("testGetUserList userCode:" + user.getUserCode()
18             + "and userName" + user.getUserName() + user.getModifyDate());
19     }
20 }

```

2.调用sqlSession.getmapper(Mapper.class)执行DAO接口方法来实现对数据的查询操作
 在增加完UserMapper.xml中的标签后，在cn.smbms.dao.user下重新建立一个接口文件：

```

1 package cn.smbms.dao.user;

```

```

2 import java.util.List;
3 import cn.smbms.pojo.User;
4
5 public interface UserMapper {
6     public List<User> geAll();
7 }

```

增加测试单元如下:

```

1 package com.yuntu.test;
2
3 import static org.junit.jupiter.api.Assertions.*;
4 import java.util.ArrayList;
5 import java.util.List;
6 import org.apache.ibatis.session.SqlSession;
7 import org.apache.log4j.Logger;
8 import org.junit.jupiter.api.Test;
9 import com.yuntu.dao.user.UserMapper;
10 import com.yuntu.pojo.User;
11 import com.yuntu.util.MybatisUtil;
12
13 class UserMapperTest {
14     private Logger logger =
15         Logger.getLogger(UserMapperTest.class);
16     @Test
17     void test() {
18         String resource="mybatis-config.xml";
19         SqlSession sqlSession = null;
20         List<User> userList = null;
21
22         try {
23             //3.创建SqlSession
24             sqlSession=MybatisUtil.creatSqlSession();
25             //4.调用mapper文件对数据进行操作 ps: 检查mapper是否在mybatis-conf
26             g.xml中进行配置
27             // userList = sqlSession.selectList("com.yuntu.dao.user.UserMap
28             per.getAll");
29         } catch (Exception e) {
30             logger.error(e.getMessage());
31         }
32     }
33 }

```

```

27  userList = sqlSession.getMapper(UserMapper.class).getAll();
28  } catch (Exception e) {
29      e.printStackTrace();
30  } finally {
31      MybatisUtil.closeSqlSession(sqlSession);
32  }
33  for (User user : userList) {
34      logger.debug(user.getUserCode() + "---" + user.getUserName() +
35                  "---" + user.getUserPassword());
36  }
37  }
38

```

注：1、接口的方法的名字必须同xml文件中的id一致。

selectOne和selectList区别：

selectOne表示查询出一条记录进行映射。如果使用selectOne可以实现使用selectList也可以实现（list中只有一个对象）。

selectList表示查询出一个列表（多条记录）进行映射。如果使用selectList查询多条记录，不能使用selectOne。

如果使用selectOne报错：

org.apache.ibatis.exceptions_TOO_MANY_RESULTS: Expected one result (or null) to be returned by selectOne(), but found: 4

Mapper动态代理方式

开发规范

Mapper接口开发方法只需要程序员编写Mapper接口（相当于Dao接口），由Mybatis框架根据接口定义创建接口的动态代理对象，代理对象的方法体同上边Dao接口实现类方法。

Mapper接口开发需要遵循以下规范：

- 1、Mapper.xml文件中的namespace与mapper接口的类路径相同。
- 2、Mapper接口方法名和Mapper.xml中定义的每个statement的id相同

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

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

Mapper.xml(映射文件)

定义mapper映射文件UserMapper.xml（内容同Users.xml），需要修改 namespace的值为 UserMapper 接口路径，路径为当前映射文件所在绝对路径（如：src/com/yuntu/dao/user/UserMapper.xml）。

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper
3 PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5 <!--
6 mapper接口代理实现编写规则：
7   1. 映射文件中namespace要等于接口的全路径名称
8   2. 映射文件中sql语句id要等于接口的方法名称
9   3. 映射文件中传入参数类型要等于接口方法的传入参数类型
10  4. 映射文件中返回结果集类型要等于接口方法的返回值类型
11 -->
12 <mapper namespace="cn.zy.mapper.UserMapper">
13 <!--
14   id:sql语句唯一标识
15   parameterType:指定传入参数类型
16   resultType:返回结果集类型
17   #{ } 占位符:起到占位作用,如果传入的是基本类型(string,long,double,int,boolean,float等),那么#{ }中的变量名称可以随意写.
18 -->
19 <select id="findUserById" parameterType="int" resultType="cn.i
20 theima.pojo.User">
21   select * from user where id=#{id}
22 </select>
23 <!--
```

24 如果返回结果为集合,可以调用`selectList`方法,这个方法返回的结果就是一个集合,所以映射文件中应该配置成集合泛型的类型

25 `${}`拼接符:字符串原样拼接,如果传入的参数是基本类型(`string`,`long`,`double`,`int`,`boolean`,`float`等),那么`${}`中的变量名称必须是`value`

26 注意:拼接符有sql注入的风险,所以慎重使用

27 -->

28 `<select id="findUserByUserName" parameterType="string" resultType="user">`

29 `select * from user where username like '%${value}%'`

30 `</select>`

31

32 `<!--`

33 `#{}:`如果传入的是`pojo`类型,那么`#{}:`中的变量名称必须是`pojo`中对应的属性.属性.属性.....

34 如果要返回数据库自增主键:可以使用`select LAST_INSERT_ID()`

35 -->

36 `<insert id="insertUser" parameterType="cn.itheima.pojo.User" >`

37 `<!-- 执行 select LAST_INSERT_ID()数据库函数,返回自增的主键`

38 `keyProperty:`将返回的主键放入传入参数的`Id`中保存.

39 `order:`当前函数相对于`insert`语句的执行顺序,在`insert`前执行是`before`,在`insert`后执行是`AFTER`

40 `resultType:``id`的类型,也就是`keyproperties`中属性的类型

41 -->

42 `<selectKey keyProperty="id" order="AFTER" resultType="java.lang.Integer">`

43 `select LAST_INSERT_ID()`

44 `</selectKey>`

45 `insert into user (username,birthday,sex,address) values(#{username},#{birthday},#{sex},#{address})`

46 `</insert>`

47 `</mapper>`

Mapper.java(接口文件)

```
1 public interface UserMapper {
2     public User findUserById(Integer id);
3     //动态代理形式中,如果返回结果集为List,那么mybatis会在生成实现类的时候会自动调用selectList方法
```

```
4 public List<User> findUserByUserName(String userName);  
5 public void insertUser(User user);  
6 }
```

接口定义有如下特点:

- 1、Mapper接口方法名和Mapper.xml中定义的statement的id相同
- 2、Mapper接口方法的输入参数类型和mapper.xml中定义的statement的parameterType的类型相同
- 3、Mapper接口方法的输出参数类型和mapper.xml中定义的statement的结果Type的类型相同

加载UserMapper.xml文件

修改mybatis-config.xml文件:

```
1 <!-- 加载映射文件 -->  
2 <mappers>  
3 <mapper resource="mapper/UserMapper.xml"/>  
4 </mappers>
```

mybatis-config.xml配置文件元素

配置的内容和顺序如下:

- properties (属性)
- settings (全局配置参数)
- typeAliases (类型别名)
- typeHandlers (类型处理器)
- objectFactory (对象工厂)
- plugins (插件)
- environments (环境集合属性对象)
- environment (环境子属性对象)
- transactionManager (事务管理)
- dataSource (数据源)
- mappers (映射器)

properties (属性)

将数据库连接的参数单独配置在, database.properties中, 只需要在mybatis-config.xml中加载database.properties的属性值。在mybatis-config.xml中就不需要对数据库连接参数硬编码。

好处: 方便对参数进行统一管理, 其它xml可以引用该db.properties

特性： MyBatis 将按照下面的顺序来加载属性：

- 在 properties 元素体内定义的属性首先被读取。
- 然后会读取properties 元素中resource或 url 加载的属性，它会覆盖已读取的同名属性。
- 最后读取parameterType传递的属性，它会覆盖已读取的同名属性。

建议：

- 不要在properties元素体内添加任何属性值，只将属性值定义在properties文件中。
- 在properties文件中定义属性名要有一定的特殊性，如
XXXXXX.XXXXXX.XXXX

```
1 <!-- 通过porpert的子元素来进行数据库连接的相关配置-->
2 <properties>
3   <property name="driver" value="com.mysql.jdbc.Driver" />
4   <property name="url" value="jdbc:mysql://localhost:3306/mybatis_
   _studydb" />
5   <property name="username" value="root" />
6   <property name="password" value="123456" />
7 </properties>
```

为了方便管理，我们一般使用properties文件进行数据库的连接配置。创建jdbc.properties文件，放置classpath路径下。方便我们配置数据库。

```
1 driver=com.mysql.jdbc.Driver
2 url=jdbc:mysql://127.0.0.1:3306/SMBMS?useUnicode=true&characterE
  ncoding=utf-8
3 user=root
4 password=123456
```

接下来我们就可以把properties和它的子元素进行替换即可

```
1 <!-- 引入 database.properties(数据库配置文件) 文件 -->
2 <properties resource="database.properties"></properties>!
```

操作示例：

```
database.properties x mybatis-config.xml
1 driver=com.mysql.jdbc.Driver
2 url=jdbc:mysql://127.0.0.1:3306/SMBMS?useUnicode=true&characterEncoding=utf-8
3 user=root
4 password=123456
```

```
database.properties x mybatis-config.xml x
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE configuration
3 PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4 "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6 <configuration>
7   <!-- 引入 database.properties 文件 -->
8   <properties resource="database.properties"></properties>
9   <!-- 配置log4j，配置后后台会自动对SQL语句输入 -->
10  <settings>
11    <setting name="LogImpl" value="LOG4J"/>
12  </settings>
13
14  <environments default="development">
15    <environment id="development">
16      <!-- 配置事务管理，此处采用JDBC的事务管理 -->
```

settings (全局配置参数)

settings是 MyBatis 中极为重要的调整设置可以调整一些运行参数，它们会改变 MyBatis 的运行时行为。大部分情况下保持默认值运行即可，比如：开启二级缓存、开启延迟加载。

一个配置完整的 settings 元素的示例如下：

```
1 <settings>
2   <setting name="cacheEnabled" value="true"/>
3   <setting name="lazyLoadingEnabled" value="true"/>
4   <setting name="multipleResultSetsEnabled" value="true"/>
5   <setting name="useColumnLabel" value="true"/>
6   <setting name="useGeneratedKeys" value="false"/>
7   <setting name="autoMappingBehavior" value="PARTIAL"/>
8   <setting name="autoMappingUnknownColumnBehavior"
9     value="WARNING"/>
10  <setting name="defaultExecutorType" value="SIMPLE"/>
11  <setting name="defaultStatementTimeout" value="25"/>
12  <setting name="defaultFetchSize" value="100"/>
```



```

12 <setting name="safeRowBoundsEnabled" value="false"/>
13 <setting name="mapUnderscoreToCamelCase" value="false"/>
14 <setting name="localCacheScope" value="SESSION"/>
15 <setting name="jdbcTypeForNull" value="OTHER"/>
16 <setting name="lazyLoadTriggerMethods" value="equals,clone,has
hCode,toString"/>
17 </settings>

```

1	Setting (设置)	Description (描述)	Valid Values (验证值组)	Default (默认值)
2	cacheEnabled	在全局范围内启用或禁用缓存配置任何映射器在此配置下。	true false	TRUE
3	lazyLoadingEnabled	在全局范围内启用或禁用延迟加载。禁用时，所有协会将热加载。	true false	TRUE
4	aggressiveLazyLoading	启用时，有延迟加载属性的对象将被完全加载后调用懒惰的任何属性。否则，每一个属性是按需加载。	true false	TRUE
5	multipleResultSetsEnabled	允许或不允许从一个单独的语句（需要兼容的驱动程序）要返回多个结果集。	true false	TRUE
6	useColumnLabel	使用列标签，而不是列名。在这方面，不同的驱动有不同的行为。参考驱动文档或测试两种方法来决定你的驱动程序的行为如何。	true false	TRUE
7	useGeneratedKeys	允许JDBC支持生成的密钥。兼容的驱动程序是必需的。此设置强制生成的键被使用，如果设置为true，一些驱动会不兼容性，但仍然可以工作。	true false	FALSE
8	autoMappingBehavior	指定MyBatis的应如何自动映射列到字段/属性。NONE自动映射。PARTIAL只会自动映射结果没有嵌套结果映射定义里面。FULL会自动映射的结果映射任何复杂的（包含嵌套或其他）。	NONE, PARTIAL, FULL https://blog.csdn.net/Song_jiangTao	PARTIAL

9	defaultExecutorType	配置默认执行人。SIMPLE执行人确实没有什么特别的。REUSE执行器重用准备好的语句。BATCH执行器重用语句和批处理更新。	SIMPLE REUSE BATCH	SIMPLE
10	defaultStatementTimeout	设置驱动程序等待一个数据库响应的秒数。	Any positive integer	Not Set (null)
11	safeRowBoundsEnabled	允许使用嵌套的语句RowBounds。	true false	FALSE
12	mapUnderscoreToCamelCase	从经典的数据库列名A_COLUMN启用自动映射到骆驼标识的经典的Java属性名aColumn。	true false	FALSE
13	localCacheScope	MyBatis的使用本地缓存，以防止循环引用，并加快反复嵌套查询。默认情况下（SESSION）会话期间执行的所有查询缓存。如果localCacheScope=STATEMENT本地会话将被用于语句的执行，只是没有将数据共享之间的两个不同的调用相同的SqlSession。	SESSION STATEMENT	SESSION
14	jdbcTypeForNull	指定为空值时，没有特定的JDBC类型的参数的JDBC类型。有些驱动需要指定列的JDBC类型，但其他像NULL，VARCHAR或OTHER的工作与通用值。	JdbcType enumeration. Most common are: NULL, VARCHAR and OTHER	OTHER
15	lazyLoadTriggerMethods	指定触发延迟加载的对象的方法。	A method name list separated by commas https://blog.csdn.net/Song_jiangTao	equals, clone, hashCode, toString

16	defaultScriptingLanguage	指定所使用的语言默认为动态SQL生成。	A type alias or fully qualified class name.	org.apache.ibatis.scripting.xmltags.XMLDynamicLanguage
17	callSettersOnNulls	指定如果setter方法或地图的put方法时，将调用检索到的值是null。它是有用的，当你依靠Map.keySet()或null初始化。注意原语（如整型，布尔等）不会被设置为null。	true false	FALSE
18	logPrefix	指定的前缀字符串，MyBatis将会增加记录器的名称。	Any String	Not set
19	logImpl	指定MyBatis的日志实现使用。如果此设置是不存在的记录的实施将自动查找。	SLF4J LOG4J LOG4J2 JDK_LOGGING COMMONS_LOGGING STDOUT_LOGGING	Not set
20	proxyFactory	指定代理工具，MyBatis将会使用创建懒加载能力的对象。	CGLIB JAVASSIST	https://blog.csdn.net/Song_JiangTao

typeAliases (类型别名) (重点)

由于类的完全限定名称比较长，大量使用时十分不方便。MyBatis通过别名的方式来代表它，在MyBatis中**别名是不区分大小写**。别名又分为系统定义别名和自定义别名。

配置自定义别名：

```

1 <typeAliases>
2 <!--全写-->
3 <typeAlias type="com.lzx.entity.Author" alias="author" />
4 <typeAlias type="com.lzx.entity.Blog" alias="blog" />
5 <!--简写，MyBatis自动扫描这个包下面的所有类，把第一字母变成小写作为别名-->
6 <package name="com.lzx.entity" />
7 </typeAliases>

```

使用注解配置别名：

```

1 @Alias("blog")
2 public class Blog {
3     ...
4 }

```

- 单个定义

```

<!--setting, 需要时配置-->
<!--<setting></setting>-->
<!--别名定义-->
<typeAliases>
    <!--type:类型路径, alias: 别名-->
    <typeAlias type="com.nuc.mybatis.po.User" alias="user"></typeAlias>
</typeAliases>

```

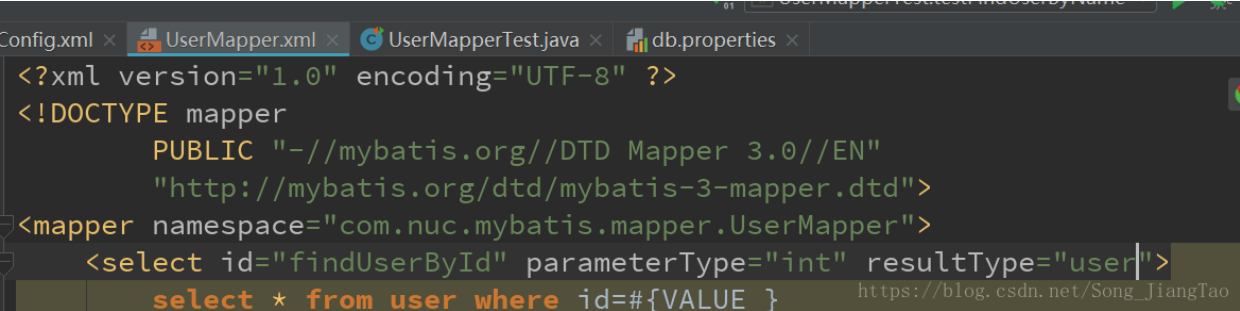
https://blog.csdn.net/Song_JiangTao

- 批量定义（常用）

```
<!--<setting></setting>-->
<!--别名定义-->
<typeAliases>
    <!--批量定义:mybatis自动扫描包中的类，别名就是类名（首字母大小写都可以）-->
    <package name="com.nuc.mybatis.po"></package>
</typeAliases>
```

https://blog.csdn.net/Song_JiangTao

这样在其他地方就可以使用，例如：



```
<?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.nuc.mybatis.mapper.UserMapper">
    <select id="findUserById" parameterType="int" resultType="user">
        select * from user where id=#{VALUE }
```

https://blog.csdn.net/Song_JiangTao

mybatis默认支持的别名

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

typeHandlers (类型处理器)

在JDBC中，需要在PreparedStatement中处理预编译的SQL语句中的参数，执行完毕SQL后，通过ResultSet对象获得数据库中的数据，这些数据类型在MyBatis中通过typeHandler实现。在typeHandler中分为jdbcType和javaType。jdbcType定义数据库类型，javaType定义java类型。typeHandler的作用是承担jdbcType和javaType之间的相互转换。一般来说，系统定义的类型Handler就足够我们使用，我们还可以自定义typeHandler来处理我们需要满足的转换规则。

- mybatis中通过typeHandlers完成jdbc类型和java类型的转换。通常情况下，mybatis提供的类型处理器满足日常需要，不需要自定义。
- mybatis支持的类型处理器

系统定义的typeHandler:

类型处理器	Java类型	JDBC类型
BooleanTypeHandler	java.lang.Boolean,boolean	数据库兼容的BOOLEAN
ByteTypeHandler	java.lang.Byte,byte	数据库兼容的NUMERIC或BYTE
ShortTypeHandler	java.lang.Short,short	数据库兼容的NUMERIC或SHORTINTEGER
IntegerTypeHandler	java.lang.Integer,int	数据库兼容的NUMERIC或INTEGER
LongTypeHandler	java.lang.Long,long	数据库兼容的NUMERIC或LONGINTEGER
FloatTypeHandler	java.lang.Float,float	数据库兼容的NUMERIC或FLOAT
DoubleTypeHandler	java.lang.Double,double	数据库兼容的NUMERIC或DOUBLE
BigDecimalTypeHandler	java.math.BigDecimal	数据库兼容的NUMERIC或DECIMAL
StringTypeHandler	java.lang.String	CHAR,VARCHAR
ClobReaderTypeHandler	java.io.Reader	-
ClobTypeHandler	java.lang.String	CLOB,LONGVARCHAR
NStringTypeHandler	java.lang.String	NVARCHAR,NCHAR
NClobTypeHandler	java.lang.String	NCLOB
BlobInputStreamTypeHandler	java.io.InputStream	-
ByteArrayTypeHandler	byte[]	数据库兼容的字节流类型
BlobTypeHandler	byte[]	BLOB,LONGVARBINARY
DateTypeHandler	java.util.Date	TIMESTAMP
DateOnlyTypeHandler	java.util.Date	DATE
TimeOnlyTypeHandler	java.util.Date	TIME
SqlTimestampTypeHandler	java.sql.Timestamp	TIMESTAMP
SqlDateTypeHandler	java.sql.Date	DATE
SqlTimeTypeHandler	java.sql.Time	TIME
ObjectTypeHandler	Any	OTHER或未指定类型
EnumTypeHandler	EnumerationType	VARCHAR-任何兼容的字符串类型，存储枚举的名称（而不是索引）
EnumOrdinalTypeHandler	EnumerationType	任何兼容的NUMERIC或DOUBLE类型，存储枚举的索引（而不是名称）。
InstantTypeHandler	java.time.Instant	TIMESTAMP
LocalDateTimeTypeHandler	java.time.LocalDateTime	TIMESTAMP
LocalDateTypeHandler	java.time.LocalDate	DATE
LocalTimeTypeHandler	java.time.LocalTime	TIME
OffsetDateTimeTypeHandler	java.time.OffsetDateTime	TIMESTAMP
OffsetTimeTypeHandler	java.time.OffsetTime	TIME
ZonedDateTimeTypeHandler	java.time.ZonedDateTime	TIMESTAMP
YearTypeHandler	java.time.Year	INTEGER
MonthTypeHandler	java.time.Month	INTEGER
YearMonthTypeHandler	java.time.YearMonth	VARCHAR或LONGVARCHAR
JapaneseDateTypeHandler	java.time.chrono.JapaneseDate	DATE

objectFactory (对象工厂)

这个自行查看下载mybatis时附带的pdf文件，用的不多

plugins (插件)

environments (环境集合属性对象)

environment (环境子属性对象)

transactionManager (事务管理)

dataSource (数据源)

mappers (映射器)

既然 MyBatis 的行为已经由上述元素配置完了，我们现在就要定义 SQL 映射语句了。但是首先我们需要告诉 MyBatis 到哪里去找到这些语句。Java 在自动查找这方面没有提供一个很好的方法，所以最佳的方式是告诉 MyBatis 到哪里去找映射文件。你可以使用相对于类路径的资源引用，或完全限定资源定位符（包括 `file:///` 的 URL），或类名和包名等。例如：

```
1 <!-- 使用相对于类路径的资源引用 -->
2 <mappers>
3   <mapper resource="org/mybatis/builder/AuthorMapper.xml"/>
4   <mapper resource="org/mybatis/builder/BlogMapper.xml"/>
5   <mapper resource="org/mybatis/builder/PostMapper.xml"/>
6 </mappers>
7
8 <!-- 使用完全限定资源定位符（URL） -->
9 <mappers>
10  <mapper url="file:///var/mappers/AuthorMapper.xml"/>
11  <mapper url="file:///var/mappers/BlogMapper.xml"/>
12  <mapper url="file:///var/mappers/PostMapper.xml"/>
13 </mappers>
14
15 <!-- 使用映射器接口实现类的完全限定类名 -->
16 <mappers>
17  <mapper class="org.mybatis.builder.AuthorMapper"/>
18  <mapper class="org.mybatis.builder.BlogMapper"/>
19  <mapper class="org.mybatis.builder.PostMapper"/>
20 </mappers>
21
```

```

22 <!-- 将包内的映射器接口实现全部注册为映射器 -->
23 <mappers>
24   <package name="org.mybatis.builder"/>
25 </mappers>

```

- 通过resource

```
<!--加载映射文件-->
```

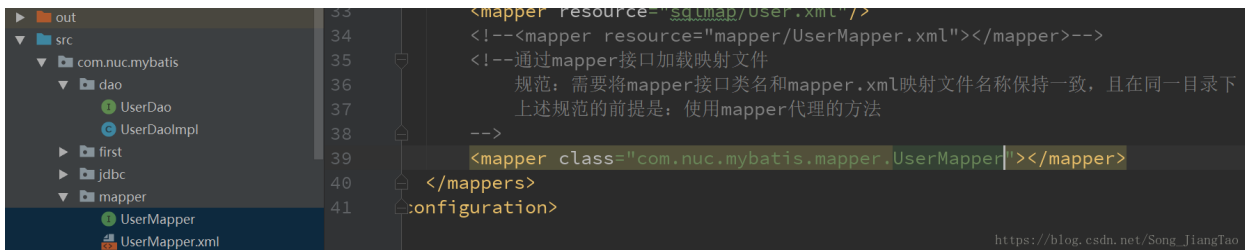
```

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

```

https://blog.csdn.net/Song_JiangTao

- 通过class



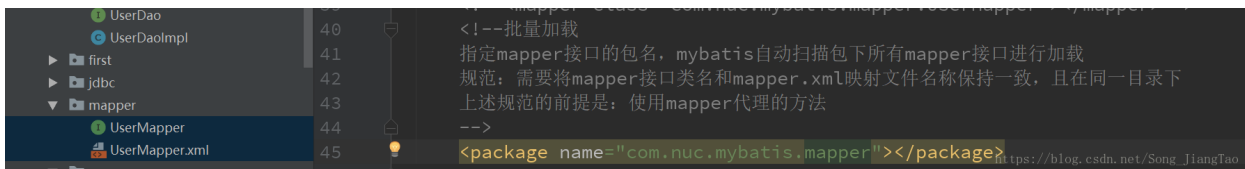
```

33 <!--加载映射文件-->
34 <mappers>
35   <mapper resource="sqlmap/User.xml"/>
36   <!--通过mapper接口加载映射文件
37   规范：需要将mapper接口类名和mapper.xml映射文件名称保持一致，且在同一目录下
38   上述规范的前提是：使用mapper代理的方法
39   -->
40   <mapper class="com.nuc.mybatis.mapper.UserMapper"></mapper>
41 </mappers>
42 </configuration>

```

https://blog.csdn.net/Song_JiangTao

- 通过package(推荐使用)



```

40 <!--批量加载
41 指定mapper接口的包名，mybatis自动扫描包下所有mapper接口进行加载
42 规范：需要将mapper接口类名和mapper.xml映射文件名称保持一致，且在同一目录下
43 上述规范的前提是：使用mapper代理的方法
44 -->
45 <package name="com.nuc.mybatis.mapper"></package>

```

https://blog.csdn.net/Song_JiangTao

environments (配置环境)

MyBatis 可以配置成适应多种环境，这种机制有助于将 SQL 映射应用于多种数据库之中，现实情况下有多种理由需要这么做。例如，开发、测试和生产环境需要有不同的配置；或者共享相同 Schema 的多个生产数据库，想使用相同的 SQL 映射。许多类似的用例。

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

```

1 environments default="development">
2   <environment id="development">
3     <transactionManager type="JDBC">
4       <property name="..." value="..."/>
5     </transactionManager>
6     <dataSource type="POOLED">

```



```

7 <property name="driver" value="${driver}"/>
8 <property name="url" value="${url}"/>
9 <property name="username" value="${username}"/>
10 <property name="password" value="${password}"/>
11 </dataSource>
12 </environment>
13 </environments>

```

注意这里的关键点:

- 默认的环境 ID (比如:default="development") 。
- 每个 environment 元素定义的环境 ID (比如:id="development") 。
- 事务管理器的配置 (比如:type="JDBC") 。
- 数据源的配置 (比如:type="POOLED") 。

默认的环境和环境 ID 是自解释的, 因此一目了然。你可以对环境随意命名, 但一定要保证默认的环境 ID 要匹配其中一个环境 ID。

transactionManager (事务管理器)

在 MyBatis 中有两种类型的事务管理器 (也就是 type="[JDBC|MANAGED]") :

- JDBC – 这个配置就是直接使用了 JDBC 的提交和回滚设置, 它依赖于从数据源得到的连接来管理事务作用域。
- MANAGED – 这个配置几乎没做什么。它从来不提交或回滚一个连接, 而是让容器来管理事务的整个生命周期 (比如 JEE 应用服务器的上下文)。默认情况下它会关闭连接, 然而一些容器并不希望这样, 因此需要将 closeConnection 属性设置为 false 来阻止它默认的关闭行为。例如:

```

1 <transactionManager type="MANAGED">
2   <property name="closeConnection" value="false"/>
3 </transactionManager>
4
5 <!-- 配置事务管理, 此处采用JDBC的事务管理 -->
6 <transactionManager type="JDBC"/>

```

如果你正在使用 Spring + MyBatis, 则没有必要配置事务管理器, 因为 Spring 模块会使用自带的管理器来覆盖前面的配置。

POOLED: 这种数据源的实现利用“池”的概念将 JDBC 连接对象组织起来, 避免了创建新的连接实例时所必需的初始化和认证时间。这是一种使得并发 Web 应用快速响应请求的流行处理方式。

第二章、SQL映射文件

MyBatis 的真正强大在于它的映射语句，也是它的魔力所在。由于它的异常强大，映射器的 XML 文件就显得相对简单。

Mapper.xml映射文件中定义了操作数据库的sql，每个sql是一个statement，映射文件是mybatis的核心。

映射文件中有很多属性，常用的就是parameterType(输入类型)、resultType(输出类型)、resultMap () 、rparameterMap () 。

SQL 映射文件有很少的几个顶级元素（按照它们应该被定义的顺序）：

- **namespace** – 命名空间，即映射文件的路径，这里把mapper标签和接口联系在一起了，namespace=写接口路径，映射文件要和接口在同一目录下
- **cache** – 给定命名空间的缓存配置。
- **cache-ref** – 其他命名空间缓存配置的引用。
- **resultMap** – 是最复杂也是最强大的元素，用来描述如何从数据库结果集中来加载对象。
- ~~**parameterMap** – 已废弃！老式风格的参数映射。内联参数是首选,这个元素可能在将来被移除，这里不会记录。~~
- **sql** – 可被其他语句引用的可重用语句块。
- **insert** – 映射插入语句
- **update** – 映射更新语句
- **delete** – 映射删除语句
- **select** – 映射查询语句

select

查询语句是 MyBatis 中最常用的元素之一，光能把数据存到数据库中价值并不大，如果还能重新取出来才有用，多数应用也都是查询比修改要频繁。对每个插入、更新或删除操作，通常对应多个查询操作。这是 MyBatis 的基本原则之一，也是将焦点和努力放到查询和结果映射的原因。简单查询的 select 元素是非常简单的。比如：

id	userCode	userName	userPassword	gender	birthday	phone	address	userRole	createdBy	creationDate	modifyBy	modifyDate
1	admin	系统管理员	1234567		1 1983-10-10	13688889999	北京市海淀区成府路207号	1	1	2013-03-21 16:52:07	(Null)	(Null)
2	liming	李明	0000000		2 1983-12-10	13688884457	北京市东城区前门东大街5	2	1	2019-01-23 13:53:58	(Null)	(Null)
5	hanlubiao	韩路彪	0000000		2 1984-06-05	18567542321	北京市朝阳区北辰中心12-	2	1	2014-12-31 19:52:09	(Null)	(Null)
6	zhanghua	张华	0000000		1 1983-06-15	13544561111	北京市海淀区学院路61号	2	1	2013-02-11 10:51:17	(Null)	(Null)
7	wangyang	王洋	0000000		2 1982-12-31	13444561124	北京市海淀区西二旗辉煌	3	1	2014-06-11 19:09:07	(Null)	(Null)
8	zhaoyan	赵燕	0000000		1 1986-03-07	18098764545	北京市海淀区回龙观小区1	3	1	2016-04-21 13:54:07	(Null)	(Null)
10	sunlei	孙磊	0000000		2 1981-01-04	13387676765	北京市朝阳区管庄新月小	3	1	2015-05-06 10:52:07	(Null)	(Null)
11	sunxing	孙兴	0000000		2 1978-03-12	13367890900	北京市朝阳区建国门南大	3	1	2016-11-09 16:51:17	(Null)	(Null)
12	zhangchen	张晨	0000000		1 1986-03-28	18098765434	朝阳区管庄路口北柏林	3	1	2016-08-09 05:52:37	1	2016-04-14 14:15:36
13	dengchao	邓超	0000000		2 1981-11-04	13689674534	北京市海淀区北航家属院1	3	1	2016-07-11 08:02:47	(Null)	(Null)
14	yangguo	杨过	0000000		2 1980-01-01	13388886623	北京市朝阳区北苑家园美	3	1	2015-02-01 03:52:07	(Null)	(Null)
15	zhaomin	赵敏	0000000		1 1987-12-04	18099897657	北京市昌平区天通苑3区1	2	1	2015-09-12 12:02:12	(Null)	(Null)

```

1 <!-- 查询用户总数 -->
2 <select id="count" resultType="int">
3     select count(1) as count from smbms_user
4 </select>
5
6 <!-- 查询所有姓氏为张的用户 -->
7 <select id="getByName" resultType="user" parameterType="String">
8     SELECT * FROM smbms_user WHERE userName LIKE CONCAT('%',{userName},'%');
9     <!-- SELECT * FROM smbms_user WHERE userName LIKE '%${value}%' -->
10 </select>
11
12 <!-- 查询姓名带张和角色为经理的用户 -->
13 <select id="getByNameRol" resultType="user" parameterType="user"
14 >
15     SELECT * FROM smbms_user
16     WHERE userName LIKE CONCAT('%',{userName},'%')
17     AND userRole=#{userRole};
18 </select>

```

select 元素有很多属性允许你配置，来决定每条语句的作用细节。

```

1 <select
2     id="selectPerson"
3     parameterType="int"
4     parameterMap="deprecated"
5     resultType="hashmap"
6     resultMap="personResultMap"
7     flushCache="false"
8     useCache="true"

```

```

9   timeout="10000"
10  fetchSize="256"
11  statementType="PREPARED"
12  resultSetType="FORWARD_ONLY">

```

属性	描述
<code>id</code>	在命名空间中唯一的标识符，可以被用来引用这条语句。
<code>parameterType</code>	将会传入这条语句的参数类的完全限定名或别名。这个属性是可选的，因为 MyBatis 可以通过 TypeHandler 推断出具体传入语句的参数，默认值为 unset。
<code>parameterMap</code>	这是引用外部 <code>parameterMap</code> 的已经被废弃的方法。使用内联参数映射和 <code>parameterType</code> 属性。
<code>resultType</code>	从这条语句中返回的期望类型的类的完全限定名或别名。注意如果是集合情形，那应该是集合可以包含的类型，而不能是集合本身。使用 <code>resultType</code> 或 <code>resultMap</code> ，但不能同时使用。
<code>resultMap</code>	外部 <code>resultMap</code> 的命名引用。结果集的映射是 MyBatis 最强大的特性，对其有一个很好的理解的话，许多复杂映射的情形都能迎刃而解。使用 <code>resultMap</code> 或 <code>resultType</code> ，但不能同时使用。
<code>flushCache</code>	将其设置为 true，任何时候只要语句被调用，都会导致本地缓存和二级缓存都会被清空，默认值：false。
<code>useCache</code>	将其设置为 true，将会导致本条语句的结果被二级缓存，默认值：对 select 元素为 true。
<code>timeout</code>	这个设置是在抛出异常之前，驱动程序等待数据库返回请求结果的秒数。默认值为 unset（依赖驱动）。
<code>fetchSize</code>	这是尝试影响驱动程序每次批量返回的结果行数而这个设置值相等。默认值为 unset（依赖驱动）。
<code>statementType</code>	STATEMENT，PREPARED 或 CALLABLE 的一个。这会让 MyBatis 分别使用 Statement，PreparedStatement 或 CallableStatement，默认值：PREPARED。
<code>resultSetType</code>	FORWARD_ONLY，SCROLL_SENSITIVE 或 SCROLL_INSENSITIVE 中的一个，默认值为 unset（依赖驱动）。
<code>databaseId</code>	如果配置了 databaseIdProvider，MyBatis 会加载所有的不带 databaseId 或匹配当前 databaseId 的语句；如果带或者不带的语句都有，则不带的会被忽略。
<code>resultOrdered</code>	这个设置仅针对嵌套结果 select 语句适用：如果为 true，就是假设包含了嵌套结果集或是分组了，这样的话当返回一个主结果行的时候，就不会发生有对前面结果集的引用的情况。这就使得在获取嵌套的结果集的时候不至于导致内存不够用。默认值：false。
<code>resultSets</code>	这个设置仅对多结果集的情况适用，它将列出语句执行后返回的结果集并每个结果集给一个名称，名称是逗号分隔的。

parameterType(输入类型)

通过parameterType指定输入参数的类型，类型可以是简单类型、hashmap、pojo的包装类型。

1、#{ } 与 \${ }

#{}实现的是向prepareStatement中的预处理语句中设置参数值，sql语句中#{}表示一个占位符即？。

```
1 <!-- 根据id查询用户信息 -->
2 <select id="findUserById" parameterType="int" resultType="user">
3   select * from user where id = #{id}
4 </select>
```

使用占位符 #{} 可以有效防止sql注入，在使用时不需要关心参数值的类型，mybatis会自动进行java类型和jdbc类型的转换。#{}可以接收简单类型值或POJO属性值，如果 parameterType 传输单个简单类型值，#{}括号中可以是value或其它名称。

\${} 和 #{} 不同，通过\${}可以将parameterType 传入的内容拼接在sql中且不进行jdbc类型转换，\${}可以接收简单类型值或pojo属性值，如果 parameterType传输单个简单类型值，\${}括号中只能是value。使用\${}不能防止sql注入，但是有时用 \${} 会非常方便，如下的例子：

```
1 <!-- 根据名称模糊查询用户信息 -->
2 <select id="selectUserByName" parameterType="string"
  resultType="user">
3   select * from user where username like '%${value}%'
4 </select>
```

2、传递简单类型

传递简单类型只需要注意 #{} 与 \${} 的使用就可以。

3、传递pojo对象(实体类)

Mybatis使用ognl表达式解析对象字段的值，如下例子：

```
1 <!-- 传递pojo对象综合查询用户信息 -->
2 <select id="findUserByUser" parameterType="user" resultType="user">
3   select * from user where id=#{id} and username like '%${username}%'
4 </select>
```

传递hashmap

Sql映射文件定义如下：

```
1 <!-- 传递hashmap综合查询用户信息 -->
```

```

2  <select id="findUserByHashmap" parameterType="hashmap" resultType="user">
3    select * from user where id=#{id} and username like '%${username}%'
4  </select>
5  上边红色标注的是hashmap的key。
6
7  测试：
8  Public void testFindUserByHashmap()throws Exception{
9    //获取session
10   SqlSession session = sqlSessionSessionFactory.openSession();
11   //获取mapper接口实例
12   UserMapper userMapper = session.getMapper(UserMapper.class);
13   //构造查询条件HashMap对象
14   HashMap<String, Object> map = new HashMap<String, Object>();
15   map.put("id", 1);
16   map.put("username", "管理员");
17   //传递HashMap对象查询用户列表
18   List<User>list = userMapper.findUserByHashmap(map);
19   //关闭session
20   session.close();
21 }

```

异常测试：

传递的map中的key和sql中解析的key不一致。

测试结果没有报错，只是通过key获取值为空。

resultType(输出类型)

使用resultType进行输出映射，只有查询出来的列名和pojo中的属性名一致，该列才可以映射成功。如果查询出来的列名和pojo的属性名不一致，通过定义一个resultMap对列名和pojo属性名之间作一个映射关系。

1、输出简单类型

mapper.xml:

```

1 <!--
2  用户信息综合查询总数
3  parameterType: 指定输入类型和findUserList一样
4  resultType: 输出结果类型
5  -->
6 <select id="findUserCount" parameterType="com.iot.mybatis.po.UserQueryVo" resultType="int">
7   SELECT count(*) FROM user WHERE sex=#{userCustom.sex} AND
8   user.username LIKE '%${userCustom.username}%'
9 </select>

```

输出简单类型必须查询出来的结果集有一条记录，最终将第一个字段的值转换为输出类型。使用session的selectOne可查询单条记录。

2、输出pojo对象

映射文件:

```

1 //根据id查询用户信息
2 public User findUserById(int id) throws Exception;
3
4 <!-- 根据id查询用户信息 -->
5 <select id="findUserById" parameterType="int" resultType="user">
6   select * from user where id = #{id}
7 </select>

```

3、输出pojo列表

映射文件:

```

1 //根据用户名列查询用户列表
2 public List<User> findUserByName(String name) throws Exception;
3
4 <!-- 根据名称模糊查询用户信息 -->
5 <select id="findUserByUsername" parameterType="string" resultType="user">
6   select * from user where username like '%${value}%'
7 </select>

```

注意：MyBatis会根据Mapper接口方法的返回类型来选择调用selectOne(返回单个对象调用)还是selectList（返回集合对象调用）方法。

4、输出hashmap

输出pojo对象可以改用hashmap输出类型，将输出的字段名称作为map的key，value为字段值。

resultMap

mybatis中使用resultMap完成高级输出结果映射。

resultType可以指定pojo将查询结果映射为pojo，但需要pojo的属性名和sql查询的列名一致方可映射成功。**如果sql查询字段名和pojo的属性名不一致，可以通过resultMap将字段名和属性名作一个对应关系，resultMap实质上还需要将查询结果映射到pojo对象中。**

```
1 1.定义resultMap
2 <!-- 定义resultMap
3 将SELECT id id_,username username_ FROM USER 和User类中的属性作一个映射关系
4
5 type: resultMap最终映射的java对象类型,可以使用别名
6 id: 对resultMap的唯一标识
7 -->
8 <resultMap type="user" id="userResultMap">
9 <!--
10 id: 表示查询结果集中唯一标识
11 column: 查询出来的列名
12 property: type指定的pojo类型中的属性名
13 最终resultMap对column和property作一个映射关系 （对应关系）
14 -->
15 <id column="id_" property="id"/>
16
17 <!--
18 result: 对普通名映射定义
19 column: 查询出来的列名
20 property: type指定的pojo类型中的属性名
21 最终resultMap对column和property作一个映射关系 （对应关系）
22 -->
23 <result column="username_" property="username"/>
```



```

24 </resultMap>
25
26 2.使用resultMap作为statement的输出映射类型
27 <!--
28 使用resultMap进行输出映射
29 resultMap: 指定定义的resultMap的id, 如果这个resultMap在其它的mapper文件, 前边需要加namespace
30 -->
31 <select id="findUserMapById" parameterType="java.lang.Integer"
32 resultMap="userMap" >
33   select id id_,username username_ from user where id = #{id}
34 </select>
35
36 mapper.java代码:
37 //根据id查询用户信息, 使用resultMap输出
38 public User findUserByIdResultMap(int id) throws Exception;
39
40 测试代码:
41 @Test
42 public void testFindUserByIdResultMap() throws Exception {
43   SqlSession sqlSession = sqlSessionFactory.openSession();
44   //创建UserMapper对象, mybatis自动生成mapper代理对象
45   UserMapper userMapper =
46     sqlSession.getMapper(UserMapper.class);
47   //调用userMapper的方法
48   User user = userMapper.findUserByIdResultMap(1);
49   System.out.println(user);
50 }

```

association (一对一查询)

复杂的类型关联, 一对一查询, 映射到JavaBean 的某个“复杂类型”属性,其他JavaBean类.

- 内部嵌套
 - 映射一个JavaBean属性
- 属性
 - property: 映射数据库列的实体对象属性

- javaType: 完整Java类名或者别名
- resultMap: 引用外部resultMap
- 子元素
 - id: 表示查询结果集中唯一标识
 - resul: 查询出来的列名, 映射到JavaBean 的某个 “简单类型” 属性,String,int等
 - property: 映射数据库列的实体对象的属性, 指定的pojo类型中的属性名
 - column: 数据库列名或者别名

案例: 根据用户ID查询用户信息和角色信息。

用户表数据 (smbms_user) :

id	userCode	userName	userPassword	gender	birthday	phone	address	userRole	createdBy	creationDate	modifyBy	modifyDate
1	admin	系统管理员	1234567		1 1983-10-10	13688889999	北京市海淀区成府路207号	1	1	2013-03-21 16:52:07	(Null)	(Null)
2	liming	李明	0000000		2 1985-12-10	13688884457	北京市东城区前门东大街5	2	1	2019-01-23 13:53:58	(Null)	(Null)
5	hanlubiao	韩路彪	0000000		2 1984-06-05	18567542321	北京市朝阳区北辰中心12	2	1	2014-12-31 19:52:09	(Null)	(Null)
6	zhanghua	张华	0000000		1 1983-06-15	13544561111	北京市海淀区学院路61号	2	1	2013-02-11 10:51:17	(Null)	(Null)
7	wangyang	王洋	0000000		2 1996-12-31	13444561124	北京市海淀区西二旗邮编	3	1	2014-06-11 19:09:07	(Null)	(Null)
8	zhaoayan	赵燕	0000000		1 1986-03-07	18098764545	北京市海淀区回龙观小区1	3	1	2016-04-21 13:54:07	(Null)	(Null)
10	sunlei	孙磊	0000000		2 1981-01-04	13387676765	北京市朝阳区管庄新月小	3	1	2015-05-06 10:52:07	(Null)	(Null)
11	sunxing	孙兴	0000000		2 1978-03-12	13367890900	北京市朝阳区建国门南大	3	1	2016-11-09 16:51:17	(Null)	(Null)
12	zhangchen	张晨	0000000		1 1986-03-28	18098765434	朝阳区管庄路口北柏林爱	3	1	2016-08-09 05:52:37	1	2016-04-14 14:15:36
13	dengchao	邓超	0000000		2 1999-11-04	13689674534	北京市海淀区北航家属院1	3	1	2016-07-11 08:02:47	(Null)	(Null)
14	yangguo	杨过	0000000		2 1980-01-01	13388886623	北京市朝阳区北苑家园莱	3	1	2015-02-01 03:52:07	(Null)	(Null)
15	zhaomin	赵敏	0000000		1 1987-12-04	18099897657	北京市昌平区天通苑3区1	2	1	2015-09-12 12:02:12	(Null)	(Null)
16	qiaqian	倩倩	123456		1 1915-08-26	183069899457	北京市昌平区新苑小区3区	2	1	2019-01-21 16:02:12	(Null)	(Null)
25	cat	小猫咪	666666		(Null) (Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)
26	xiaohua	小花	888888		(Null) (Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)

角色表数据 (smbms_user) :

id	roleCode	roleName	createdBy	creationDate	modifyBy	modifyDate
1	SMBMS_ADMIN	系统管理员	1	2016-04-13 00:00:00	(Null)	(Null)
2	SMBMS_MANAGER	经理	1	2016-04-13 00:00:00	(Null)	(Null)
3	SMBMS_EMPLOYEE	普通员工	1	2016-04-13 00:00:00	(Null)	(Null)

```

1 1.编写角色表POJO类
2 public class Role {
3     //属性
4     private Integer id;//主键ID
5     private String roleCode;//角色编码
6     private String roleName;//角色名称
7     private Integer createdBy;//创建者
8     private Date creationDate;//创建时间
9     private Integer modifyBy;//修改者

```

```
10 private Date modifyDate;//修改时间
11 .....
12 }
13
14 2.在用户表POJO类添加角色表POJO类的附加属性或继承（Role类）
15 public class User {
16     //属性
17     private Integer id;//主键ID
18     private String userCode;//用户编码
19     private String userName;//用户名称
20     private String userPassword;//用户密码
21     private Integer gender;//性别（1:女、 2:男）
22     private Date birthday;//出生日期
23     private String phone;//手机
24     private String address;//地址
25     private Integer userRole;//用户角色（取自角色表-角色id）
26     private Integer createdBy;//创建者（userId）
27     private Date creationDate;//创建时间
28     private Integer modifyBy;//更新者（userId）
29     private Date modifyDate;//更新时间
30     //附加属性
31     private Integer age;//年龄
32     private Role role;
33 }
34
35 3.Mapper.xml
36 <!-- 根据用户角色id获取用户列表 -->
37 <resultMap type="user" id="userinfo">
38     <id property="id" column="id"/>
39     <result property="userCode" column="userCode"/>
40     <result property="userName" column="userName"/>
41     <result property="userPassword" column="userPassword"/>
42     <association property="role" javaType="Role" resultMap="roleinfo"></association>
43 </resultMap>
44
```

```

45 <!-- 定义resultMap，需要关联查询映射的是用户信息，使用association将
    用户信息映射到订单对象的用户属性中。 -->
46 <resultMap type="Role" id="roleinfo">
47 <id property="id" column="roleid" />
48 <result property="roleCode" column="roleCode"/>
49 <result property="roleName" column="roleName"/>
50 </resultMap>
51 <select id="getUserByRoleId" resultMap="userinfo" parameterType=
    "Integer">
52 SELECT u.*,r.id as roleid,r.roleCode,r.roleName FROM smbms_use
    r as u,smbms_role as r WHERE u.userRole = r.id AND r.id=#
    {role_id};
53 </select>
54
55 4.Mapper接口:
56 //根据用户角色id获取用户列表
57 public List<User> getUserByRoleId(@Param("role_id") Integer rol
    eid);
58
59 5.测试
60 @Test
61 void testgetUserByRoleId() {
62 String resource="mybatis-config.xml";
63 SqlSession sqlSession = null;
64 List<User> userList=null;
65 try {
66 sqlSession=MybatisUtil.creatSqlSession();
67 userList = sqlSession.getMapper(UserMapper.class).getUserByRol
    eId(2);
68 for (User user : userList) {
69 logger.debug("用户编号: " + user.getId() + ", 用户名: " + user.g
    etUserName()+", 角色id: "+user.getRole().getId()+", 角色名
    称: "+user.getRole().getRoleName());
70 }
71 } catch (Exception e) {
72 e.printStackTrace();
73 } finally {
74 MybatisUtil.closeSqlSession(sqlSession);

```

```

75     }
76 }

```

代码运行效果：

```

<terminated> UserMapperTest.getUserById [JUnit] C:\Program Files\Java\jdk-9\bin\java.exe (2019年1月22日 下午2:09:19)
org.apache.ibatis.datasource.pooled.PooledDataSource - Created connection 957465255.
org.apache.ibatis.transaction.jdbc.JdbcTransaction - Setting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a2b3c4d]
com.yuntu.dao.user.UserMapper.getUserById - ooo Using Connection [com.mysql.jdbc.JDBC4Connection@1a2b3c4d]
com.yuntu.dao.user.UserMapper.getUserById - ==> Preparing: SELECT u.*,r.id as roleid,r.roleCode as rolename FROM user u LEFT JOIN role r ON u.roleid=r.id WHERE u.id=?
com.yuntu.dao.user.UserMapper.getUserById - ==> Parameters: 2(Integer)
com.yuntu.test.UserMapperTest - 用户编号: 2, 用户名: 李明, 角色id: 2, 角色名称: 经理
com.yuntu.test.UserMapperTest - 用户编号: 5, 用户名: 韩路彪, 角色id: 2, 角色名称: 经理
com.yuntu.test.UserMapperTest - 用户编号: 6, 用户名: 张华, 角色id: 2, 角色名称: 经理
com.yuntu.test.UserMapperTest - 用户编号: 15, 用户名: 赵敏, 角色id: 2, 角色名称: 经理
com.yuntu.test.UserMapperTest - 用户编号: 16, 用户名: 倩倩, 角色id: 2, 角色名称: 经理
org.apache.ibatis.transaction.jdbc.JdbcTransaction - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a2b3c4d]
org.apache.ibatis.datasource.pooled.PooledDataSource - Returned connection 957465255 to pool.

```

collection (一对多查询)

复杂类型集合

案例：根据用户id查询用户信息和收货地址

用户表数据 (smbms_user)：

id	userCode	userName	userPassword	gender	birthday	phone	address	userRole	createdBy	creationDate	modifyBy	modifyDate
1	admin	系统管理员	1234567		1 1983-10-10	13688889999	北京市海淀区成府路207号	1		1 2013-03-21 16:52:07	(Null) (Null)	
2	liming	李明	0000000		2 1985-12-10	13688884457	北京市东城区前门东大街5号	2		1 2019-01-23 13:53:58	(Null) (Null)	
5	hanlubiao	韩路彪	0000000		2 1984-06-05	18567542321	北京市朝阳区北辰中心12-1	2		1 2014-12-31 19:52:09	(Null) (Null)	
6	zhanghua	张华	0000000		1 1983-06-15	13544561111	北京市海淀区学院路61号	2		1 2013-02-11 10:51:17	(Null) (Null)	
7	wangyang	王洋	0000000		2 1996-12-31	13444561124	北京市海淀区西二旗锦绣园	3		1 2014-06-11 19:09:07	(Null) (Null)	
8	zhaoyan	赵燕	0000000		1 1986-03-07	18098764545	北京市海淀区回龙观小区1	3		1 2016-04-21 13:54:07	(Null) (Null)	
10	sunlei	孙磊	0000000		2 1981-01-04	13387676765	北京市朝阳区管庄新月小区	3		1 2015-05-06 10:52:07	(Null) (Null)	
11	sunxing	孙兴	0000000		2 1978-03-12	13367890900	北京市朝阳区建国门南大街	3		1 2016-11-09 16:51:17	(Null) (Null)	
12	zhangchen	张晨	0000000		1 1986-03-28	18098765434	朝阳区管庄路口北柏林爱琴	3		1 2016-08-09 05:52:37	1 2016-04-14 14:15:36	
13	dengchao	邓超	0000000		2 1999-11-04	13689674534	北京市海淀区北航家属院1	3		1 2016-07-11 08:02:47	(Null) (Null)	
14	yangguo	杨过	0000000		2 1980-01-01	13388886623	北京市朝阳区北苑家园莱锦	3		1 2015-02-01 03:52:07	(Null) (Null)	
15	zhaomin	赵敏	0000000		1 1987-12-04	18099897657	北京市昌平区天通苑3区1	2		1 2015-09-12 12:02:12	(Null) (Null)	
16	qiaqian	倩倩	123456		1 1915-08-26	183069899457	北京市昌平区新苑小区3区	2		1 2019-01-21 16:02:12	(Null) (Null)	
25	cat	小猫咪	666666	(Null) (Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null) (Null)	(Null) (Null)	
26	xiaohua	小花	888888	(Null) (Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null) (Null)	(Null) (Null)	

用户地址表 (smbms_address)：

id	contact	addressDesc	postCode	tel	createdBy	creationDate	modifyBy	modifyDate	userId
1	王丽	北京市东城区东交民巷44号	100010	13678789999		1 2016-04-13 00:00:00	(Null) (Null)		1
2	张红丽	北京市海淀区丹棱街3号	100000	18567672312		1 2016-04-13 00:00:00	(Null) (Null)		1
3	任志强	北京市东城区美术馆后街23号	100021	13387906742		1 2016-04-13 00:00:00	(Null) (Null)		1
4	曹颖	北京市朝阳区朝阳门南大街14号	100053	13568902323		1 2016-04-13 00:00:00	(Null) (Null)		2
5	李慧	北京市西城区三里河路南三巷3号	100032	18032356666		1 2016-04-13 00:00:00	(Null) (Null)		3
6	王国强	北京市顺义区高丽营镇金马工业区18号	100061	13787882222		1 2016-04-13 00:00:00	(Null) (Null)		3

```

1 1.在User实体类中添加List<Address> addList 属性
2 public class User {
3     //属性
4     private Integer id;//主键ID
5     private String userCode;//用户编码
6     private String userName;//用户名称
7     private String userPassword;//用户密码
8     private Integer gender;//性别（1:女、 2:男）

```

```

9  private Date birthday;//出生日期
10 private String phone;//手机
11 private String address;//地址
12 private Integer userRole;//用户角色（取自角色表-角色id）
13 private Integer createdBy;//创建者（userId）
14 private Date creationDate;//创建时间
15 private Integer modifyBy;//更新者（userId）
16 private Date modifyDate;//更新时间
17
18 //附加属性
19 private Integer age;//年龄
20 private List<Address> addList;//收货地址
21 }
22
23 2.Mapper接口:
24 //根据用户id查询用户信息及地址
25 public User getUserAndAdd(@Param("id") Integer userid);
26
27 3.Mapper.xml
28 <!-- 根据用户id查询用户信息及地址 -->
29 <resultMap type="user" id="userAddInfo">
30 <id property="id" column="id"/>
31 <result property="userName" column="userName"/>
32 <result property="userPassword" column="userPassword"/>
33 <collection property="addList" ofType="Address">
34 <id property="id" column="add_id"/>
35 <result property="contact" column="contact"/>
36 <result property="addressDesc" column="addressDesc"/>
37 </collection>
38 </resultMap>
39
40 <select id="getUserAndAdd" parameterType="Integer"
    resultMap="userAddInfo">
41 select u.*,a.id as add_id,a.contact,a.addressDesc from smbms_u
    ser as u,smbms_address as a where u.id=a.userId and u.id=#{id};
42 </select>

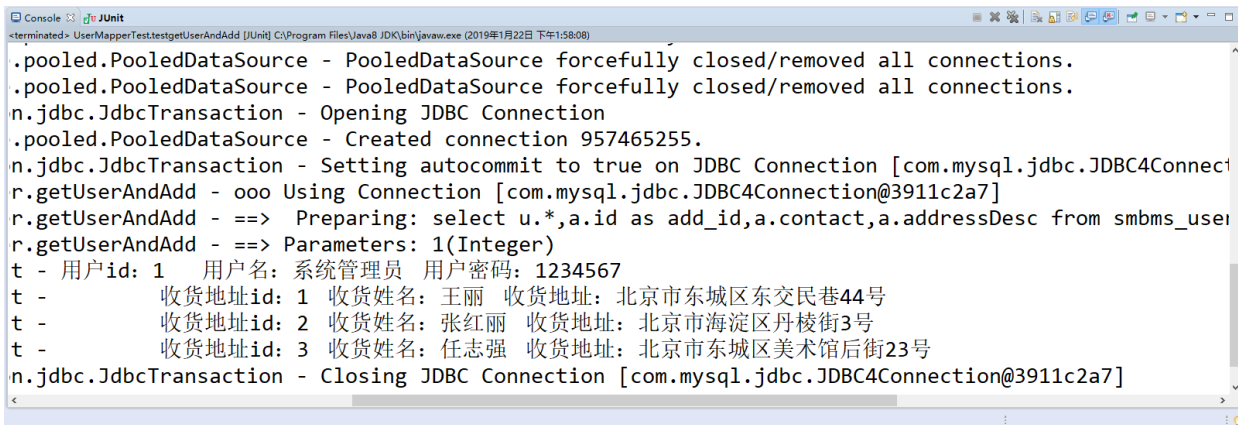
```

```

43 4.测试
44 @Test
45 void testgetUserAndAdd() {
46     String resource="mybatis-config.xml";
47     SqlSession sqlSession = null;
48     User user=null;
49     try {
50         sqlSession=MybatisUtil.creatSqlSession();
51         user =
sqlSession.getMapper(UserMapper.class).getUserAndAdd(1);
52         if (null!=user) {
53             logger.debug("用户id: "+user.getId()+" 用户名: "+user.getUserName()
e()+" 用户密码: "+user.getUserPassword());
54             List<Address> addList =user.getAddList();
55             for (Address address : addList) {
56                 logger.debug("\t收货地址id: "+address.getId()+" 收货姓名: "+address
ess.getContact()+" 收货地址: "+address.getAddressDesc());
57             }
58         }
59     } catch (Exception e) {
60         e.printStackTrace();
61     } finally {
62         MybatisUtil.closeSqlSession(sqlSession);
63     }
64 }

```

运行结果:



```

Console [JUnit]
<terminated> UserMapperTest.testgetUserAndAdd [JUnit] C:\Program Files\Java8\jdk\bin\javaw.exe (2019年1月22日 下午1:58:08)
.pooled.PooledDataSource - PooledDataSource forcefully closed/removed all connections.
.pooled.PooledDataSource - PooledDataSource forcefully closed/removed all connections.
n.jdbc.JdbcTransaction - Opening JDBC Connection
.pooled.PooledDataSource - Created connection 957465255.
n.jdbc.JdbcTransaction - Setting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@3911c2a7]
r.getUserAndAdd - ooo Using Connection [com.mysql.jdbc.JDBC4Connection@3911c2a7]
r.getUserAndAdd - ==> Preparing: select u.*,a.id as add_id,a.contact,a.addressDesc from smbms_user
r.getUserAndAdd - ==> Parameters: 1(Integer)
t - 用户id: 1 用户名: 系统管理员 用户密码: 1234567
t - 收货地址id: 1 收货姓名: 王丽 收货地址: 北京市东城区东交民巷44号
t - 收货地址id: 2 收货姓名: 张红丽 收货地址: 北京市海淀区丹棱街3号
t - 收货地址id: 3 收货姓名: 任志强 收货地址: 北京市东城区美术馆后街23号
n.jdbc.JdbcTransaction - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@3911c2a7]

```

resultMap总结

- **resultType:**

作用:

将查询结果按照sql列名pojo属性名一致性映射到pojo中。

场合:

常见一些明细记录的展示，比如用户购买商品明细，将关联查询信息全部展示在页面时，此时可直接使用resultType将每一条记录映射到pojo中，在前端页面遍历list（list中是pojo）即可。

- **resultMap:**

使用**association**和**collection**完成一对一和一对多高级映射（对结果有特殊的映射 要求）。

association:

作用:

将关联查询信息映射到一个pojo对象中。

场合:

为了方便查询关联信息可以使用association将关联订单信息映射为用户对象的pojo属性中，比如：查询订单及关联用户信息。

使用resultType无法将查询结果映射到pojo对象的pojo属性中，根据对结果集查询遍历的需要选择使用resultType还是resultMap。

collection:

作用:

将关联查询信息映射到一个list集合中。

场合:

为了方便查询遍历关联信息可以使用collection将关联信息映射到list集合中，比如：查询用户权限范围模块及模块下的菜单，可使用collection将模块映射到模块list中，将菜单列表映射到模块对象的菜单list属性中，这样的作的目的也是方便对查询结果集进行遍历查询。

如果使用resultType无法将查询结果映射到list集合中。

自动映射

在简单的场景下，MyBatis可以替你自动映射查询结果。如果遇到复杂的场景，你需要构建一个resultMap。

当自动映射查询结果时，MyBatis会获取sql返回的列名并在java类中查找相同名字的属性（忽略大小写）。这意味着如果Mybatis发现了ID列和id属性，Mybatis会将ID的值赋给id。

有三种自动映射等级：

- **NONE** - 禁止自动映射。仅设置手动映射属性。
- **PARTIAL** - 半自动映射(不含嵌套)。(默认值)
- **FULL** - 全自动映射所有(含嵌套)。

```
1 <settings>
2   <setting name="autoMappingBehavior" value="PARTIAL"/>
3 </settings>
```

insert

```
1 <insert
2   id="insertAuthor"
3   parameterType="domain.blog.Author"
4   flushCache="true"
5   statementType="PREPARED"
6   keyProperty=""
7   keyColumn=""
8   useGeneratedKeys=""
9   timeout="20">
10
11 <update
12   id="updateAuthor"
13   parameterType="domain.blog.Author"
14   flushCache="true"
15   statementType="PREPARED"
16   timeout="20">
17
18 <delete
19   id="deleteAuthor"
```

```

20 parameterType="domain.blog.Author"
21 flushCache="true"
22 statementType="PREPARED"
23 timeout="20">

```

属性	描述
id	命名空间中的唯一标识符，可被用来代表这条语句。
parameterType	将要传入语句的参数的完全限定类名或别名。这个属性是可选的，因为 MyBatis 可以通过 TypeHandler 推断出具体传入语句的参数，默认值为 unset。
parameterMap	这是引用外部 parameterMap 的已经被废弃的方法。使用内联参数映射和 parameterType 属性。
flushCache	将其设置为 true，任何时候只要语句被调用，都会导致本地缓存和二级缓存都会被清空，默认值：true（对应插入、更新和删除语句）。
timeout	这个设置是在抛出异常之前，驱动程序等待数据库返回请求结果的秒数。默认值为 unset（依赖驱动）。
statementType	STATEMENT, PREPARED 或 CALLABLE 的一个。这会让 MyBatis 分别使用 Statement, PreparedStatement 或 CallableStatement，默认值：PREPARED。
useGeneratedKeys	（仅对 insert 和 update 有用）这会令 MyBatis 使用 JDBC 的 getGeneratedKeys 方法来取出由数据库内部生成的主键（比如：像 MySQL 和 SQL Server 这样的关系数据库管理系统的自动递增字段），默认值：false。
keyProperty	（仅对 insert 和 update 有用）唯一标记一个属性，MyBatis 会通过 getGeneratedKeys 的返回值或者通过 insert 语句的 selectKey 子元素设置它的键值，默认：unset。如果希望得到多个生成的列，也可以是逗号分隔的属性名称列表。
keyColumn	（仅对 insert 和 update 有用）通过生成的键值设置表中的列名，这个设置仅在某些数据库（像 PostgreSQL）是必须的，当主键列不是表中的第一列的时候需要设置。如果希望得到多个生成的列，也可以是逗号分隔的属性名称列表。
databaseId	如果配置了 databaseIdProvider，MyBatis 会加载所有的不带 databaseId 或匹配当前 databaseId 的语句；如果带或者不带的语句都有，则不带的会被忽略。

insert, update 和 delete 语句的示例：

```

1  接口
2  //添加用户信息
3  public int addUserInfo(User user);
4  //修改用户
5  public int updateUser(User user);
6  //修改密码
7  public int updatePwd(@Param("id") Integer id,@Param("userPassword") String userPassword);
8  //删除用户
9  public int deleteUser(Integer id);
10
11 mapper.xml
12 <!-- 添加用户信息 -->
13 <insert id="addUserInfo" parameterType="User">
14     INSERT INTO smbms_user (userCode,userName,userPassword) VALUES
15     (#{userCode},#{userName},#{userPassword});
16 </insert>
17 <!-- 修改用户信息 -->

```

```
18 <update id="updateUser" parameterType="user">
19     UPDATE smbms_user SET userCode=#{userCode},userName=#{
    {userName}},userPassword=#{userPassword} WHERE id=#{id};
20 </update>
21
22 <!-- 修改密码 -->
23 <update id="updatePwd">
24     UPDATE smbms_user SET userPassword=#{userPassword} WHERE id=#{
    {id};
25 </update>
26
27 <!-- 删除用户信息 -->
28 <delete id="deleteUser" parameterType="Integer">
29     DELETE FROM smbms_user WHERE id=#{id};
30 </delete>
31
32 测试:
33 @Test
34 void testaddUserInfo() {
35     String resource="mybatis-config.xml";
36     SqlSession sqlSession = null;
37     int number = 0;
38     try {
39         sqlSession=MybatisUtil.creatSqlSession();
40         User user = new User();
41         user.setUserCode("xiaohua");
42         user.setUserName("小花");
43         user.setUserPassword("666448");
44         number = sqlSession.getMapper(UserMapper.class).addUserInfo(us
            er);
45         logger.debug("number:"+number);
46     } catch (Exception e) {
47         e.printStackTrace();
48     } finally {
49         MybatisUtil.closeSqlSession(sqlSession);
50     }
```

```
51 }
52
53 @Test
54 void testupdateUser() {
55     String resource="mybatis-config.xml";
56     SqlSession sqlSession = null;
57     int number = 0;
58     try {
59         sqlSession=MybatisUtil.creatSqlSession();
60         User user = new User();
61         user.setUserCode("cat");
62         user.setUserName("小猫咪");
63         user.setUserPassword("666666");
64         user.setId(25);
65         number = sqlSession.getMapper(UserMapper.class).updateUser(user);
66         logger.debug("number:"+number);
67     } catch (Exception e) {
68         e.printStackTrace();
69     } finally {
70         MybatisUtil.closeSqlSession(sqlSession);
71     }
72 }
73
74 @Test
75 void testupdatePwd() {
76     String resource="mybatis-config.xml";
77     SqlSession sqlSession = null;
78     int number = 0;
79     try {
80         sqlSession=MybatisUtil.creatSqlSession();
81         number = sqlSession.getMapper(UserMapper.class).updatePwd(26,
            "888888");
82         logger.debug("number:"+number);
83     } catch (Exception e) {
84         e.printStackTrace();
85     }
```

```

85     } finally {
86         MybatisUtil.closeSqlSession(sqlSession);
87     }
88 }
89
90 @Test
91 void testdeleteUser() {
92     String resource="mybatis-config.xml";
93     SqlSession sqlSession = null;
94     int number = 0;
95     try {
96         sqlSession=MybatisUtil.creatSqlSession();
97         number =
sqlSession.getMapper(UserMapper.class).deleteUser(26);
98         logger.debug("受影响行数: "+number);
99     } catch (Exception e) {
100         e.printStackTrace();
101     } finally {
102         MybatisUtil.closeSqlSession(sqlSession);
103     }
104 }

```

使用@Param注解传递多个参数

什么是注解 (Annotatation) ?

对于很多初次接触的开发者来说应该都有这个疑问?

Annotatation 是Java5开始引入的新特征，中文名称叫注解。它提供了一种安全的类似注释的机制，用来将任何的信息或元数据（metadata）与程序元素（类、方法、成员变量等）进行关联。为程序的元素（类、方法、成员变量）加上更直观更明了的说明，这些说明信息是与程序的业务逻辑无关，并且供指定的工具或框架使用。Annotatation像一种修饰符一样，应用于包、类型、构造方法、方法、成员变量、参数及本地变量的声明语句中。

Java注解是附加在代码中的一些元信息，用于一些工具在编译、运行时进行解析和使用，起到说明、配置的功能。注解不会也不能影响代码的实际逻辑，仅仅起到辅助性的作用。包含在 java.lang.annotation 包中。

注解的用处:

- 1、生成文档。这是最常见的，也是java 最早提供的注解。常用的有 **@param @return** 等
- 2、跟踪代码依赖性，实现替代配置文件功能。比如Dagger 2依赖注入，未来java开发，将大量注解配置，具有很大用处;
- 3、在编译时进行格式检查。如@Override 放在方法前，如果你这个方法并不是覆盖了超类方法，则编译时就能检查出。

注解的原理:

注解本质是一个继承了Annotation的特殊接口，其具体实现类是Java运行时生成的动态代理类。而我们通过反射获取注解时，返回的是Java运行时生成的动态代理对象\$Proxy1。通过代理对象调用自定义注解（接口）的方法，会最终调用AnnotationInvocationHandler的invoke方法。该方法会从memberValues这个Map中索引出对应的值。而memberValues的来源是Java常量池。

@Param 注解的使用

@Param注解 的作用是给参数命名，参数命名后就能根据名字得到参数值，正确的将参数传入sql语句中

```
1  接口代码:
2  //修改密码
3  public int updatePwd(@Param("id") Integer id,@Param("userPassword") String userPassword);
4
5  mapper.xml代码
6  <!-- 修改密码 -->
7  <update id="updatePwd">
8  UPDATE smbms_user SET userPassword=#{userPassword} WHERE id=#{id};
9  </update>
10
11  测试代码:
12  @Test
13  void testupdatePwd() {
14  String resource="mybatis-config.xml";
```

```

15  SqlSession sqlSession = null;
16  int number = 0;
17  try {
18      sqlSession=MybatisUtil.creatSqlSession();
19      number = sqlSession.getMapper(UserMapper.class).updatePwd(26,
        "888888");
20      logger.debug("受影响行数:"+number);
21  } catch (Exception e) {
22      e.printStackTrace();
23  } finally {
24      MybatisUtil.closeSqlSession(sqlSession);
25  }
26  }

```

在方法参数的前面写上@Param("参数名"),表示给参数命名,名称就是括号中的内容

缓存 (cache)

MyBatis 包含一个非常强大的查询缓存特性,它可以非常方便地配置和定制。MyBatis 3 中的缓存实现的很多改进都已经实现了,使得它更加强大而且易于配置。

什么是查询缓存?

mybatis提供查询缓存,用于减轻数据压力,提高数据库性能。

为什么要用缓存?

如果缓存中有数据就不用从数据库中获取,大大提高系统性能。

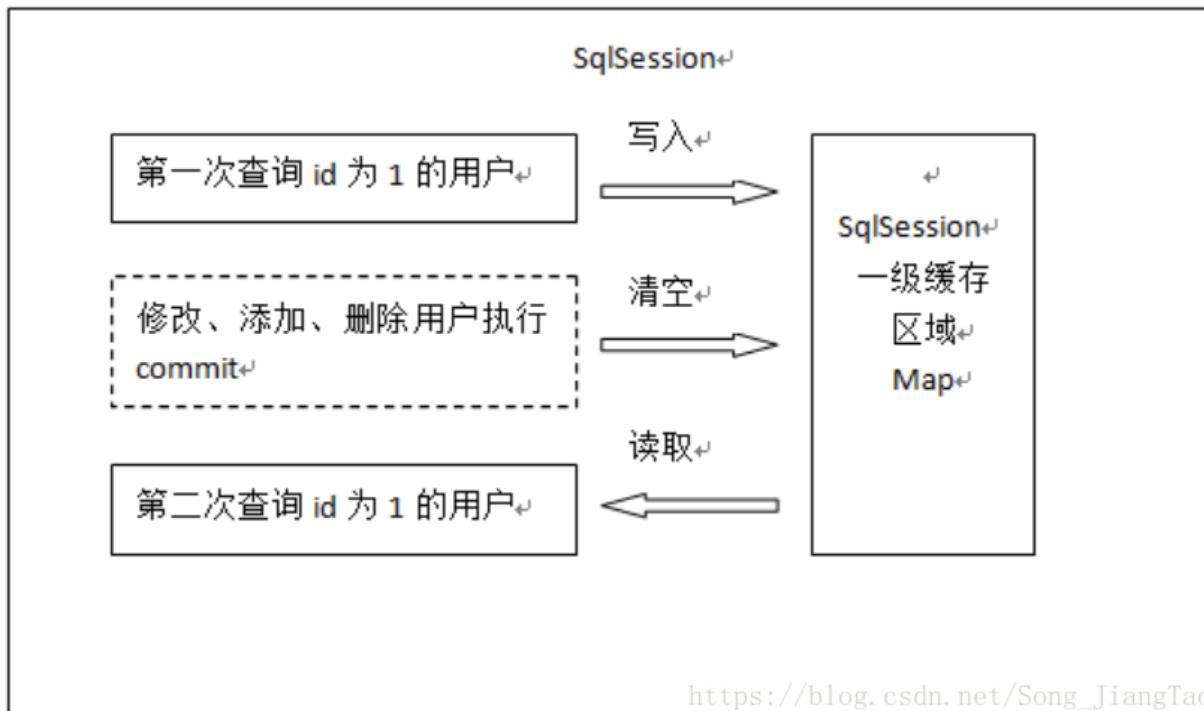
mybaits提供一级缓存,和二级缓存。



一级缓存

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。

工作原理



- 第一次发起查询用户id为1的用户信息，先去找缓存中是否有id为1的用户信息，如果没有，从数据库查询用户信息。得到用户信息，将用户信息存储到一级缓存中。
- 如果sqlSession去执行commit操作（执行插入、更新、删除），清空SqlSession中的一级缓存，这样做的目的为了让缓存中存储的是最新的信息，避免脏读。
- 第二次发起查询用户id为1的用户信息，先去找缓存中是否有id为1的用户信息，缓存中有，直接从缓存中获取用户信息。

测试：

```
1 //mybatis默认支持一级缓存，不需要在配置文件去配置。
2 @Test
3 public void testCache1() throws Exception{
```



```

4  SqlSession sqlSession = sqlSessionFactory.openSession();//创建代
   理对象
5  UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
6  //下边查询使用一个SqlSession
7  //第一次发起请求，查询id为1的用户
8  User user1 = userMapper.findUserById(1);
9  System.out.println(user1);
10 // 如果sqlSession去执行commit操作（执行插入、更新、删除），清空SqlS
   ession中的一级缓存，这样做的目的为了让缓存中存储的是最新的信息，避免脏
   读。
11 //更新user1的信息
12 user1.setUsername("测试用户22");
13 userMapper.updateUser(user1);
14 //执行commit操作去清空缓存
15 sqlSession.commit();
16 //第二次发起请求，查询id为1的用户
17 User user2 = userMapper.findUserById(1);
18 System.out.println(user2);
19 sqlSession.close();
20
21 }

```

一级缓存应用

正式开发，是将mybatis和spring进行整合开发，事务控制在service中。一个service方法中包括 很多mapper方法调用。

```

1  service{
2  //开始执行时，开启事务，创建SqlSession对象
3  //第一次调用mapper的方法findUserById(1)
4
5  //第二次调用mapper的方法findUserById(1)，从一级缓存中取数据
6  //方法结束，sqlSession关闭
7  }

```

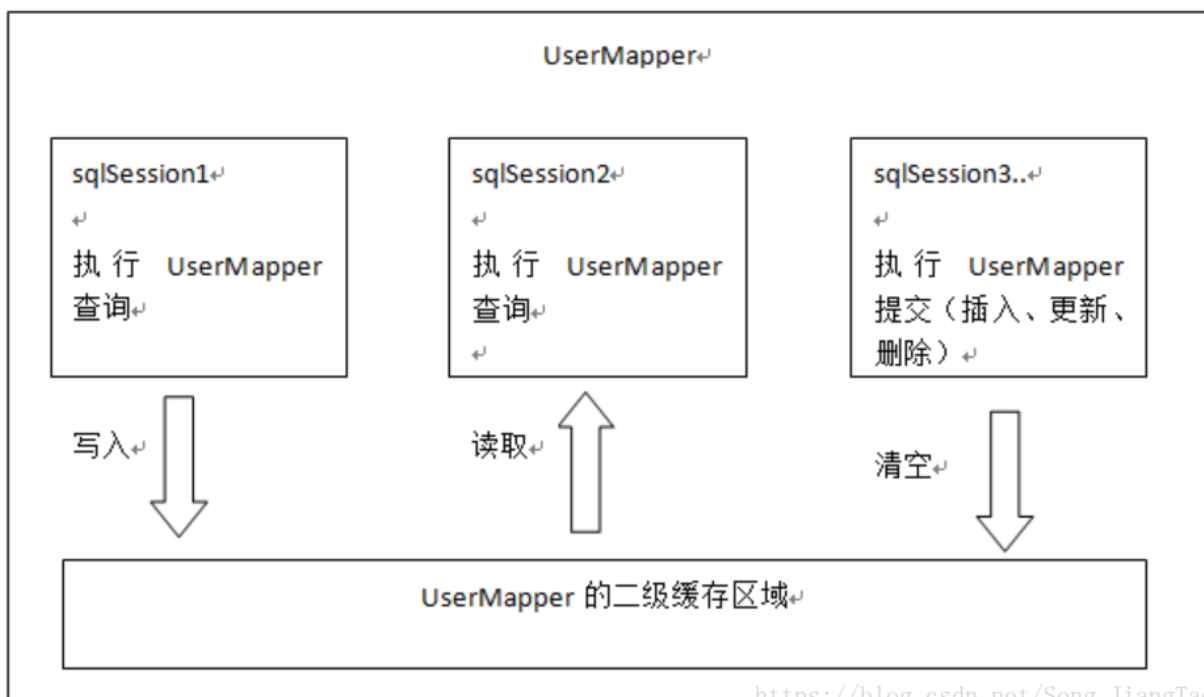
如果是执行两个service调用查询相同 的用户信息，不走一级缓存，因为 session方法结束， sqlSession就关闭，一级缓存就清空。

二级缓存

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。

默认情况下是没有开启缓存的,除了局部的 session 缓存,可以增强变现而且处理循环 依赖也是必须的。要开启二级缓存,你需要在你的 SQL 映射文件中添加一行:

原理



- 首先开启mybatis的二级缓存。
- sqlSession1去查询用户id为1的用户信息，查询到用户信息会将查询数据存储到二级缓存中。
- 如果SqlSession3去执行相同 mapper下sql，执行commit提交，清空该 mapper下的二级缓存区域的数据。
- sqlSession2去查询用户id为1的用户信息，去缓存中找是否存在数据，如果存在直接从缓存中取出数据。
- 二级缓存与一级缓存区别，二级缓存的范围更大，多个 sqlSession可以共享一个UserMapper的二级缓存区域。

- UserMapper有一个二级缓存区域（按namespace分），其它mapper也有自己的二级缓存区域（按namespace分）。
- 每一个namespace的mapper都有一个二缓存区域，两个mapper的namespace如果相同，这两个mapper执行sql查询到数据将存在相同的二级缓存区域中。

```
1 <cache/>
```

字面上看就是这样。这个简单语句的效果如下：

- 映射语句文件中的所有 select 语句将会被缓存。
- 映射语句文件中的所有 insert,update 和 delete 语句会刷新缓存。
- 缓存会使用 Least Recently Used(LRU,最近最少使用的)算法来收回。
- 根据时间表(比如 no Flush Interval,没有刷新闻隔), 缓存不会以任何时间顺序 来刷新。
- 缓存会存储列表集合或对象(无论查询方法返回什么)的 1024 个引用。
- 缓存会被视为是 read/write(可读/可写)的缓存,意味着对象检索不是共享的,而且可以安全地被调用者修改,而不干扰其他调用者或线程所做的潜在修改。

所有的这些属性都可以通过缓存元素的属性来修改。比如：

```
1 <cache
2   eviction="FIFO"
3   flushInterval="60000"
4   size="512"
5   readOnly="true"/>
6
```

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

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

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

这个更高级的配置创建了一个 FIFO 缓存,并每隔 60 秒刷新,存数结果对象或列表的 512 个引用,而且返回的对象被认为是只读的,因此在不同线程中的调用者

之间修改它们会 导致冲突。

可用的回收策略有:

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

开启二级缓存:

mybaits的二级缓存是mapper范围级别, 除了在SqlMapConfig.xml设置二级缓存的总开关, 还要在具体的mapper.xml中开启二级缓存。

```
<settings>
  <!--打开延迟加载的开关-->
  <setting name="lazyLoadingEnabled" value="true"></setting>
  <!--将积极加载改为消极加载, 即按需求加载-->
  <setting name="aggressiveLazyLoading" value="true"></setting>
  <!--开启二级缓存-->
  <setting name="cacheEnabled" value="true"></setting>
</settings>
```

https://blog.csdn.net/Song_JiangTao

```
<mapper namespace="com.nuc.mybatis.mapper.UserMapper">
  <!--开启本mapper的namespace下的二级缓存-->
  <cache></cache>
```

https://blog.csdn.net/Song_JiangTao

调用pojo类实现序列化接口

```
public class User implements Serializable {
```

https://blog.csdn.net/Song_JiangTao

测试

```
1 // 二级缓存测试
2 @Test
3 public void testCache2() throws Exception {
4     SqlSession sqlSession1 = sqlSessionFactory.openSession();
5     SqlSession sqlSession2 = sqlSessionFactory.openSession();
6     SqlSession sqlSession3 = sqlSessionFactory.openSession();
7     // 创建代理对象
```

```

8  UserMapper userMapper1 =
    sqlSession1.getMapper(UserMapper.class);
9  // 第一次发起请求，查询id为1的用户
10 User user1 = userMapper1.findUserById(1);
11 System.out.println(user1);
12
13 //这里执行关闭操作，将sqlSession中的数据写到二级缓存区域
14 sqlSession1.close();
15
16 //使用sqlSession3执行commit()操作
17 UserMapper userMapper3 = sqlSession3.getMapper(UserMapper.class);
18 User user = userMapper3.findUserById(1);
19 user.setUsername("张明明");
20 userMapper3.updateUser(user);
21 //执行提交，清空UserMapper下边的二级缓存
22 sqlSession3.commit();
23 sqlSession3.close();
24
25 UserMapper userMapper2 = sqlSession2.getMapper(UserMapper.class);
26 // 第二次发起请求，查询id为1的用户
27 User user2 = userMapper2.findUserById(1);
28 System.out.println(user2);
29 sqlSession2.close();
30 }

```

- 二级缓存应用场景:

对于访问多的查询请求且用户对查询结果实时性要求不高，此时可采用mybatis二级缓存技术降低数据库访问量，提高访问速度，业务场景比如：耗时较高的统计分析sql、电话账单查询sql等。

实现方法如下：通过设置刷新间隔时间，由mybatis每隔一段时间自动清空缓存，根据数据变化频率设置缓存刷新间隔flushInterval，比如设置为30分钟、60分钟、24小时等，根据需求而定。

- 二级缓存局限性

mybatis二级缓存对细粒度的数据级别的缓存实现不好，比如如下需求：对商品信息进行缓存，由于商品信息查询访问量大，但是要求用户每次都能查询最新的商品信息，此时如果使用mybatis的二级缓存就无法实现当一个商品变化时只刷新该商品的缓存信息而不刷新其它商品的信息，因为mybaits的二级缓存区域以mapper为单位划分，当一个商品信息变化会将所有商品信息的缓存数据全部清空。解决此类问题需要在业务层根据需求对数据有针对性缓存。

禁用二级缓存

在statement中设置useCache=false可以禁用当前select语句的二级缓存，即每次查询都会发出sql去查询，默认情况是true，即该sql使用二级缓存。

```
1 <select id="findOrderListResultMap" resultMap="ordersUserMap" useCache="false">
```

刷新缓存

在mapper的同一个namespace中，如果有其它insert、update、delete操作数据后需要刷新缓存，如果不执行刷新缓存会出现脏读。

设置statement配置中的flushCache="true" 属性，默认情况下为true即刷新缓存，如果改成false则不会刷新。使用缓存时如果手动修改数据库表中的查询数据会出现脏读。

如下：

```
1 <insert id="insertUser" parameterType="cn.itcast.mybatis.po.User" flushCache="true">
```

第三章、动态SQL

什么是动态sql?

MyBatis 核心对sql语句进行灵活操作，通过表达式进行判断，对sql进行灵活拼接、组装。

动态 SQL 元素和 JSTL 或基于类似 XML 的文本处理器相似。在 MyBatis 之前的版本中，有很多元素需要花时间了解。MyBatis 3 大大精简了元素种类，现在只需学习原来一半的元素便可。MyBatis 采用功能强大的基于 OGNL 的表达式来淘汰其它大部分元素。

MyBatis中用于实现动态SQL的元素主要有：

- 1、if ： 利用if实现简单的条件选择
- 2、choose (when, otherwise) ： 相当于java中的switch语句，通常与when和otherwise搭配
- 3、trim： 可以灵活地去除多余的关键字
- 4、set： 解决动态更新语句
- 5、foreach： 迭代一个集合，通常用于in条件
- 6、where： 简化SQL语句中where的条件判断

使用if+where实现多条件查询

```
1 <select id="getUserList" resultType="User">
2   select * from user
3   <where>
4     <if test="username!=null and userName!=''">
5       and userName like CONCAT('%',{userName},'%')
6     </if>
7   </where>
8 </select>
```

上述代码就是一个最简单的if+where的SQL映射语句，where元素标签会自动识别其标签内是否有返回值，若有，就插入一个where关键字，此外，若该标签返回的内容是以and或者or开头的，where元素会将其自动剔除，if元素标签里主要的属性就是test属性，test后面跟的是一个表达式，返回true或者false，以此来进行判断。

使用if+trim实现对条件查询

```
1 <select id="getUserList" resultType="User">
2   select * from user
3   <trim prefix="where" prefixOverride="and|or">
4     <if test="username!=null and userName!=''">
5       and userName like CONCAT('%',{userName},'%')
6     </if>
7   </trim>
8 </select>
```

从上述代码中可以看出trim和where元素标签的用法差不多，就trim标签中多了几个元素，那多了啥元素呢？

- prefix: 前缀, 作用是在trim包含的内容上加上前缀。
- suffix: 后缀, 作用是在trim包含的内容上加上后缀。
- prefixOverride: 对于trim包含内容的**首部**进行指定内容的忽略（如此处的 "and|or "）的忽略。
- suffixOverride: 对于trim包含内容的**尾部**进行指定内容的忽略。

使用if+set实现更新操作

```

1 <update id="modify" parameterType="AppInfo">
2   update app_info
3   <set>
4     <if test="logoPicPath != null">logoPicPath=#{logoPicPath},</if>
5     <if test="logoLocPath != null">logoLocPath=#{logoLocPath},</if>
6     <if test="modifyBy != null">modifyBy=#{modifyBy},</if>
7     <if test="modifyDate != null">modifyDate=#{modifyDate},</if>
8   </set>
9   where id=#{id}
10 </update>

```

上述代码就是一个最简单的if+set的动态SQL，从上面的代码中能看出其所做的更新操作是动态的，意思就是说你这个值为不为空，不为空就给你更新，要是为空就不管它，emmmm，看样子它的设计还是很人性化的。

使用if+trim实现更新操作

```

1 <update id="modify" parameterType="AppInfo">
2   update app_info
3   <trim prefix="set" suffixOverride="," suffix="where id=#{id}">
4     <if test="logoPicPath != null">logoPicPath=#{logoPicPath},</if>
5     <if test="logoLocPath != null">logoLocPath=#{logoLocPath},</if>
6     <if test="modifyBy != null">modifyBy=#{modifyBy},</if>
7     <if test="modifyDate != null">modifyDate=#{modifyDate},</if>
8   </trim>
9 </update>

```

choose, when, otherwise (类似于Switch)

```

1 <!-- 根据供应商编码或供应商名称查询供应商信息 if+where -->

```



```

2  <select id="getProInfoList" resultType="Provider"
   parameterType="Provider">
3    SELECT * FROM smbms_provider
4    <where>
5    <choose>
6    <when test="proCode!=null">
7      proCode LIKE concat('%',{proCode},'%')
8    </when>
9
10   <when test="proName!=null">
11     AND proName LIKE concat('%',{proName},'%')
12   </when>
13   <otherwise>
14     id=1;
15   </otherwise>
16   </choose>
17   </where>
18 </select>
19
20 测试代码:
21  @Test
22  void testgetProInfoList() {
23    sqlSession=MybatisUtil.creatSqlSession();
24    proList = sqlSession.getMapper(ProviderMapper.class).getProInfoList(null, null);
25    for (Provider provider : proList) {
26      logger.debug(provider.toString());
27    }
28  }

```

代码运行结果:

```

Console 33  JUnit  Git Staging
<terminated> ProviderMapperTest.testgetProInfoList [JUnit] C:\Program Files\Java\jdk-8.0.60\bin\javaw.exe (2019年1月24日 上午9:37:25)
bc.JdbcTransaction - Opening JDBC Connection
led.PooledDataSource - Created connection 440737101.
bc.JdbcTransaction - Setting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1a451d4d]
apper.getProInfoList - ooo Using Connection [com.mysql.jdbc.JDBC4Connection@1a451d4d]
apper.getProInfoList - ==> Preparing: SELECT * FROM smbms_provider WHERE id=1;
apper.getProInfoList - ==> Parameters:
t - Provider [id=1, proCode=BJ_GYS001, proName=北京三木堂商贸有限公司, proDesc=长期合作伙伴, 主营产品:茅

```

foreach

foreach主要用于在构建in条件中，它可以在SQL语句中迭代一个集合。

属性：

item：表示集合中每一个元素进行迭代的别名(如此处的roleIds)。

index：指定一个名称，用于表示在迭代过程中，每次迭代到的位置。

open：表示该语句以什么开始（既然是in条件语句，所以必然是以"**(**"开始）

separator：表示在每次进行迭代之间以什么符号作为分隔符（既然是in条件语句，所以必然以"**,**"作为分隔符）。

close：表示该语句以什么结束（既然是in条件语句，所以必然是以"**)**"结束）。

collection：最关键并最容易出错的属性，需格外注意，该属性必须指定，不同情况下，该属性的值是不一样的。主要有三种情况

- 若入参为单参数且参数类型是一个**list**的时候，collection属性值为list。
- 若入参为单参数且参数类型是一个**数组**的时候，collection属性值为array（此处传入参数Integer[] roleIds为数组类型，故此处collection属性值设为"array"）。
- 若传入参数为多参数，就需要把它们封装为一个Map进行处理。

```
1 //根据角色编号，查询所有的用户-数组方式
2 public List<User> getUserInfoByRole(Integer[] roleIds);
3 //根据角色编号，查询所有的用户-list方式
4 public List<User> getUserInfoByRole2(List<Integer> roleIds);
5
6 <!-- 根据角色编号，查询所有的用户信息 -->
7 <select id="getUserInfoByRole" resultType="user">
8   SELECT * FROM smbms_user WHERE userRole IN
9   <!-- 数组方式 -->
10   <foreach collection="array" item="roleId" open="("
11     separator="," close=")">
12     #{roleId}
13   </foreach>
```

```
14  <!-- list方式 -->
15  <foreach collection="list" item="roleId" open="("
    separator="," close=")">
16    #{roleId}
17  </foreach>
18 </select>
19
20 @Test
21 void testgetUserInfoByRole() {
22     String resource="mybatis-config.xml";
23     SqlSession sqlSession = null;
24     List<User> userList=null;
25     Integer [] roleIds= {2,3};
26     try {
27         sqlSession=MybatisUtil.creatSqlSession();
28         userList = sqlSession.getMapper(UserMapper.class).getUserInfoByRole(roleIds);
29     } catch (Exception e) {
30         e.printStackTrace();
31     } finally {
32         MybatisUtil.closeSqlSession(sqlSession);
33         logger.debug("userList的行数: " + userList.size());
34         for (User user : userList) {
35             logger.debug(user.toString());
36         }
37     }
38 }
39
40 @Test
41 void testgetUserInfoByRole2() {
42     String resource="mybatis-config.xml";
43     SqlSession sqlSession = null;
44     List<User> userList=null;
45     List<Integer> roleIds=new ArrayList<Integer>();
46     try {
47         sqlSession=MybatisUtil.creatSqlSession();
```

```

48  roleIds.add(2);
49  roleIds.add(3);
50  userList = sqlSession.getMapper(UserMapper.class).getUserInfoByRole2(roleIds);
51  } catch (Exception e) {
52  e.printStackTrace();
53  } finally {
54  MybatisUtil.closeSqlSession(sqlSession);
55  logger.debug("userList的行数: " + userList.size());
56  for (User user : userList) {
57  logger.debug(user.toString());
58  }
59  }
60  }

```

代码运行效果:

userList的行数: 13

```

User [id=2, userCode=liming, userName=李明, userPassword=0000000, gender=2, birthday=Tue Dec 10 00
User [id=5, userCode=hanlubiao, userName=韩路彪, userPassword=0000000, gender=2, birthday=Tue Jun 0
User [id=6, userCode=zhanghua, userName=张华, userPassword=0000000, gender=1, birthday=Wed Jun 15
User [id=7, userCode=wangyang, userName=王洋, userPassword=0000000, gender=2, birthday=Tue Dec 31
User [id=8, userCode=zhaoyan, userName=赵燕, userPassword=0000000, gender=1, birthday=Fri Mar 07 0
User [id=10, userCode=sunlei, userName=孙磊, userPassword=0000000, gender=2, birthday=Sun Jan 04 0
User [id=11, userCode=sunxing, userName=孙兴, userPassword=0000000, gender=2, birthday=Sun Mar 12
User [id=12, userCode=zhangchen, userName=张晨, userPassword=0000000, gender=1, birthday=Fri Mar 2
User [id=13, userCode=dengchao, userName=邓超, userPassword=0000000, gender=2, birthday=Thu Nov 04
User [id=14, userCode=yangguo, userName=杨过, userPassword=0000000, gender=2, birthday=Tue Jan 01
User [id=15, userCode=zhaomin, userName=赵敏, userPassword=0000000, gender=1, birthday=Fri Dec 04
User [id=16, userCode=qiaqian, userName=倩倩, userPassword=123456, gender=1, birthday=Thu Aug 26 0
User [id=25, userCode=hln, userName=哈琳娜, userPassword=6565465, gender=1, birthday=Thu Aug 26 00

```

map方式

```

1  //根据角色编号或姓名，查询用户信息
2  public List<User> getUserInfoByRole3(Map<String, Object>
    params);
3
4  <select id="getUserInfoByRole3" resultType="user">
5  SELECT * FROM smbms_user WHERE userName like concat('%',#
    {name},'%') and userRole IN
6  <!-- map -->
7  <foreach collection="rolIdsKey" item="roleId" open="(" separator="," close=")">
8  #{roleId}
9  </foreach>
10 </select>

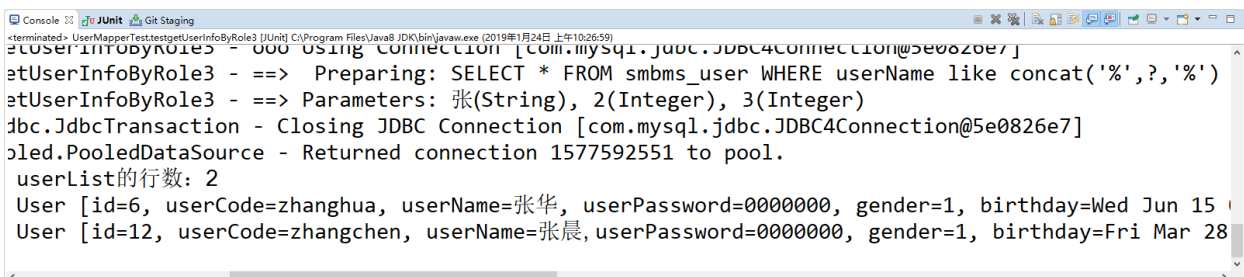
```

```

11
12 @Test
13 void testgetUserInfoByRole3() {
14     String resource="mybatis-config.xml";
15     SqlSession sqlSession = null;
16     List<User> userList=null;
17     List<Integer> roleIds=new ArrayList<Integer>();
18     roleIds.add(2);
19     roleIds.add(3);
20     Map<String, Object> params = new HashMap<String, Object>();
21     params.put("roleIdsKey", roleIds);
22     params.put("name", "张");
23
24     try {
25         sqlSession=MybatisUtil.creatSqlSession();
26         userList = sqlSession.getMapper(UserMapper.class).getUserInfoByRole3(params);
27     } catch (Exception e) {
28         e.printStackTrace();
29     } finally {
30         MybatisUtil.closeSqlSession(sqlSession);
31         logger.debug("userList的行数: " + userList.size());
32         for (User user : userList) {
33             logger.debug(user.toString());
34         }
35     }
36 }

```

代码运行效果：



```

<terminated> UserMapperTest.testgetUserInfoByRole3 [JUnit] C:\Program Files\Java8\jdk\bin\javaw.exe (2019年1月24日 上午10:26:59)
etUserInfoByRole3 - ==> Using Connection [com.mysql.jdbc.JDBC4Connection@5e0826e7]
etUserInfoByRole3 - ==> Preparing: SELECT * FROM smbms_user WHERE userName like concat('%',?, '%')
etUserInfoByRole3 - ==> Parameters: 张(String), 2(Integer), 3(Integer)
jdbc.JdbcTransaction - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@5e0826e7]
oled.PooledDataSource - Returned connection 1577592551 to pool.
userList的行数: 2
User [id=6, userCode=zhanghua, userName=张华, userPassword=0000000, gender=1, birthday=Wed Jun 15
User [id=12, userCode=zhangchen, userName=张晨, userPassword=0000000, gender=1, birthday=Fri Mar 28

```

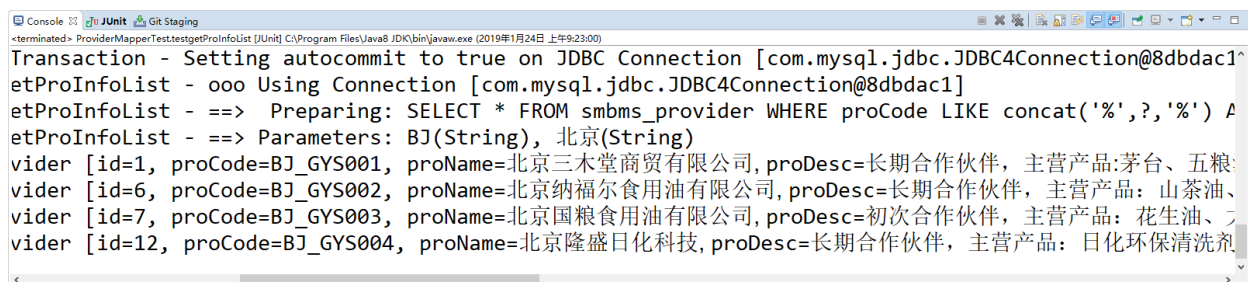
SQL片段

作用：将实现的动态sql判断代码块抽取出来，组成一个sql片段。其它的statement（声明）中就可以引用sql片段。方便程序员进行开发

先在mapper.xml中定义一个sql片段：

```
1 <!-- 根据供应商编码或供应商名称查询供应商信息 if+where -->
2 <!-- 定义sql片段
3 id: sql片段的唯一标识
4 经验：是基于单表来定义sql片段，这样的话这个sql片段可重用性才高
5 在sql片段中不要包括 where
6 -->
7 <sql id="query_getProInfoWhere">
8   <if test="proCode!=null">
9     proCode LIKE concat('%',{proCode},'%')
10  </if>
11  <if test="proName!=null">
12    AND proName LIKE concat('%',{proName},'%')
13  </if>
14 </sql>
15
16 <select id="getProInfoList" resultType="Provider"
17   parameterType="Provider">
18   SELECT * FROM smbms_provider
19   <where>
20     <!-- 引用sql片段 的id，如果refid指定的id不在本mapper文件中，需要前
21     边加namespace -->
22     <include refid="query_getProInfoWhere"></include>
23   </where>
24 </select>
```

代码运行结果：



```
<terminated> ProviderMapperTest.testGetProInfoList [JUnit] C:\Program Files\Java8\jdk\bin\javaw.exe (2019年1月24日 上午9:23:00)
Transaction - Setting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@8dbdac1]
etProInfoList - ooo Using Connection [com.mysql.jdbc.JDBC4Connection@8dbdac1]
etProInfoList - ==> Preparing: SELECT * FROM smbms_provider WHERE proCode LIKE concat('%',{proCode},'%') A
etProInfoList - ==> Parameters: BJ(String), 北京(String)
vider [id=1, proCode=BJ_GYS001, proName=北京三木堂商贸有限公司, proDesc=长期合作伙伴, 主营产品:茅台、五粮
vider [id=6, proCode=BJ_GYS002, proName=北京纳福尔食用油有限公司, proDesc=长期合作伙伴, 主营产品:山茶油、
vider [id=7, proCode=BJ_GYS003, proName=北京国粮食用油有限公司, proDesc=初次合作伙伴, 主营产品:花生油、
vider [id=12, proCode=BJ_GYS004, proName=北京隆盛日化科技, proDesc=长期合作伙伴, 主营产品:日化环保清洗剂
```

Copyright © 曹帅华 All Rights Reserved