# Mybatis概述

## 1.1原生jdbc

在传统java程序中，我们使用jdbc技术来查询数据库

使用jdbc需要下载jdbc驱动，不同的数据库有不同的jdbc驱动，这里我们使用jdbc for mysql的驱动，其maven代码如下：

<dependency>

<groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>8.0.9-rc</version>  
</dependency>

**1.加载jdbc驱动**

使用maven下载jar包后，通过下述代码来加载jdbc驱动，成功加载后，会将Driver类的实例注册到DriverManager类中。

Class.*forName*("com.mysql.jdbc.Driver");

**2.建立数据库连接**

加载完jdbc驱动以后，我们通过DriverManager类的getConnection获取Connection连接，api如下：

DriverManager.getConnection(url,username,password);

url的格式如下：jdbc协议:子协议://ip:端口号/数据库名

eg:

jdbc:mysql://127.0.0.1:80/myDatebase

**3.创建Sql语句**

接下来，创建sql语句

要创建SQL语句，必须获得java.sql.Statement实例，Statement实例分为以下3 种类型：

1、执行静态SQL语句。通常通过Statement实例实现。   
2、执行动态SQL语句。通常通过PreparedStatement实例实现。   
3、执行数据库存储**(不推荐使用存储过程)**过程。通常通过CallableStatement实例实现。

PreparedStatement继承自Statement，都是接口，但是PreparedStatement可以使用占位符，是**预编译**的，批处理比Statement效率高

eg：

//Statement stmt = con.createStatement() ;

PreparedStatement pstmt = con.prepareStatement("UPDATE EMPLOYEES SALARY = ? WHERE ID = ?");

//sql语句中的第一个占位符‘？’填入1533.00  
pstmt.setBigDecimal(1, 1533.00)

//sql语句中的第二个占位符‘？’填入1102  
pstmt.setInt(2, 1102)

//执行sql语句，打印执行结果

System.*out*.println(pstmt.execute());

而statement就不能使用占位符，同时预编译的SQL语句不是有具体数值的语句，而是用（？）来代替具体数据，然后在执行的时候再调用setXX（）方法把具体的数据传入。同时，这个语句只在第一次执行的时候编译一次，然后保存在缓存中。之后执行时，只需从缓存中抽取编译过了的代码以及新传进来的具体数据，即可获得完整的sql命令。这样一来就省下了后面每次执行时语句的编译时间。

**因此，PreparedStatement比 Statement 更快，SQL语句会预编译并缓存在数据库系统中，直接跳过数据库对SQL语句的分析，编译，优化等过程。其次，PreparedStatement的占位符还可以防止SQL注入攻击**

**4.执行**

Statement接口提供了三种执行SQL语句的方法：executeQuery 、executeUpdate和execute

**1. ResultSet executeQuery(String sqlString)**：执行SELECT语句，返回一个结果集（ResultSet）对象。

eg:

Statement stmt = conn.createStatement(ResultSet.TYPE\_SCROLL\_INSENSITIVE,ResultSet.CONCUR\_READ\_ONLY);

sql = "select \* from base\_of\_ad.ad\_list";  
ResultSet rs = stmt.executeQuery(sql);  
System.*out*.println(rs.getRow()); //rs.getRow()获得结果集的当前行号（1、2、3等），初始为0   
while(rs.next()){  
 name = rs.getString(3); //取得所在行第三列的String值   
 System.*out*.println("name="+name);  
}

**2. int executeUpdate(String sqlString)**：用于执行INSERT、UPDATE或 DELETE语句**(返回受影响行数)**以及SQL DDL语句，如：CREATE TABLE和DROP TABLE等**(操作成功返回0)**，**返回值为-1代表操作不成功**

eg:

Statement stmt = conn.createStatement(ResultSet.TYPE\_SCROLL\_INSENSITIVE,ResultSet.CONCUR\_READ\_ONLY);

sql = "update ad\_list set name = '小红' where id=1";  
//sql = "select \* from mysql.user";   
int result = stmt.executeUpdate(sql);  
if(result != -1){  
 System.*out*.println("更新 ok");  
}

**3. execute(sqlString)**:用于执行返回多个结果集、多个更新计数或二者组合的语句

**5.释放**

在jdbc中Connection是非常稀有的资源，使用完之后应当马上释放，甚至还要以”迟占用，早释放“的原则使用，也就是说在使用的过程中尽可能地要用的时候才创建Connection一用完应当立刻关闭释放。

我们需要通过close方法关闭Connection(连接)、Statement(语句)、ResultSet(结果集)这三个对象

但是需要注意的是，请在释放资源前，判断其对象是否为null，比如：**if (connect != null) connect.close();**这是必要的，如果getConnection方法没有正常获取connect对象(connect==null)，报错，被catch捕获处理，处理完成后进入finally释放资源，但此时connect对象为null，就会报空指针异常(对null调用了close方法)

**因此一个完整的jdbc案例如下：**

**需要注意的是：由于mysql版本问题(版本高了)，因此在连接数据库时，需要指定时区(serverTimezone)，并关闭SSL链接(useSSL=false)**

private String url =

"jdbc:mysql://localhost:3306/cal?serverTimezone=Asia/Shanghai&useSSL=false&characterEncoding=utf-8;  
private String username = "root";  
private String password = "123456";  
  
private Connection connect = null;  
private PreparedStatement state = null;  
private ResultSet rs = null;

public void handleRequest(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse) throws ServletException, IOException {  
 String name = null;  
 try {  
 httpServletRequest.setAttribute("name", "3333");  
 Class.*forName*("com.mysql.cj.jdbc.Driver");  
  
 connect = DriverManager.*getConnection*(url, username, password);  
 state = connect.prepareStatement("select \* from brand where typeid=?");  
 state.setInt(1, 3);  
  
 rs = state.executeQuery();  
  
 while (rs.next()) {  
 name += rs.getString(2); //取得所在行第三列的String值  
 }  
 httpServletRequest.setAttribute("brandName", name);  
 httpServletRequest.getRequestDispatcher("/WEB-INF/jsp/index.jsp").forward(httpServletRequest, httpServletResponse);  
 } catch (Exception ex) {  
 System.*out*.println("错误：" + ex.getMessage());  
 ex.getStackTrace();  
 } finally {  
 try {  
 if (state != null) state.close();  
 if (rs != null) rs.close();  
 if (connect != null) connect.close();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 }  
}

## 1.2原生jdbc 的问题

随着时间的推移，传统jdbc暴露出越来越多的问题：

**1.**传统jdbc，在使用数据库时，创建数据库连接，不使用时立即释放。似乎看起来没什么问题，但频繁地对数据库进行连接会造成大量性能损耗，浪费数据库资源

**解决方案：使用数据库连接池来管理数据库连接，当不使用数据库连接时，并不释放，而是存放到连接池中进行管理**

**2.**传统jdbc将sql语句硬编码 到java代码中，如果想更换sql语句，需要重新编写并编译java代码，不利于后期系统维护

**解决方案：将sql语句配置在xml文件中，即使sql语句变化，也不用对java代码进行重新编译**

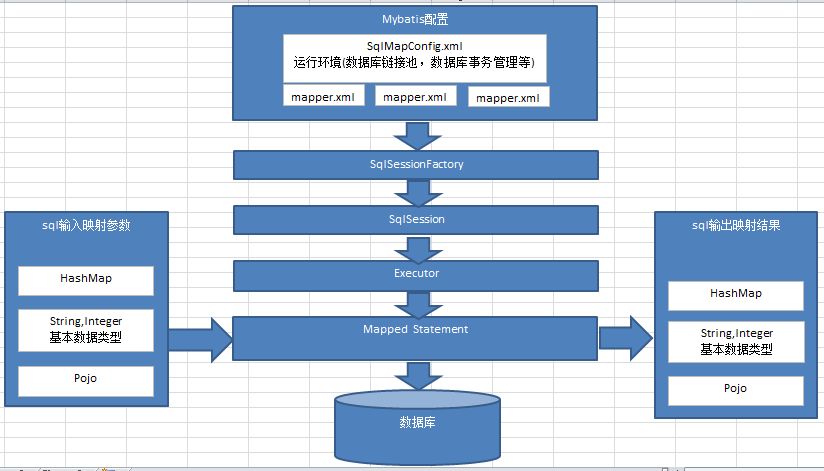
**3.** 在给**PreparedStatement**语句设置参数时，对于占位符位置和设置的参数值存在硬编码问题

**解决方案：同样将sql语句的参数值和参数位置配置在xml文件中**

**4.** 在从**ResultSet**结果集对象中遍历查询结果时，将要取出的表的字段名**硬编码**在代码中

**解决方案：将查询结果集自动映射为java bean对象**

## 1.3生命周期



## 1.4环境配置

<properties>

<mybatisVersion>3.4.4</mybatisVersion>  
 <asmVersion>3.3.1</asmVersion>  
 <cglibVersion>3.2.5</cglibVersion>  
</properties>

<dependencies>

<dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis</artifactId>  
 <version>${mybatisVersion}</version>  
</dependency>  
<dependency>  
 <groupId>asm</groupId>  
 <artifactId>asm</artifactId>  
 <version>${asmVersion}</version>  
</dependency>  
<dependency>  
 <groupId>cglib</groupId>  
 <artifactId>cglib</artifactId>  
 <version>${cglibVersion}</version>  
</dependency>

<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>8.0.9-rc</version>  
</dependency>

</dependencies>

## 1.5 log4j日志管理

日志管理在一个项目中是非常重要的环节，日志可以帮我们快速定位问题发生的位置，而Apache的log4j作为一款日志管理框架可以帮助我们轻松地进行日志管理，它能够：

**1：能够控制日志信息想往哪里打就往哪里打，比如：控制台、文件、邮箱、数据库等等**

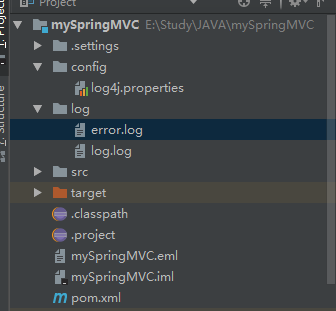
**2：能够控制日志信息想怎么打就怎么打，比如：我想要打印时间、程序的名称、程序的方法名、程序的行号、线程的名称等等**

**3：能够控制日志信息想打什么打什么，不想打的就不打，日志信息是分级别的，有时候我只想看错误的信息或者警告的信息，有时候我想看到所有的信息我想调试程序等等**

### 1.5.1配置文件

Log4J的配置文件(log4j.properties)就是用来设置记录器的级别、存放器和布局的，它可接受key=value格式的设置或xml格式的设置信息。通过配置，可以创建出Log4J的运行环境。

在项目根目录下新建文件夹config，然后在其中新建log4j.properties文件，然后在根目录下继续创建log文件夹用于存放日志，在log文件夹下，新建log.log文件(普通日志)和errorlog.log文件(错误日志)，eg：



**log4j需要配置三方面内容：**

1. **根目录（日志信息级别和目的地） ；**
2. **日志写入的目的地（写入到控制台、文件等等） ；**
3. **日志输出样式**
4. **log4j.properties配置文件的格式如下：**

#配置根Logger  
http://www.blogjava.net/Images/OutliningIndicators/None.giflog4j.rootLogger  =   **[** **level** **]**   , **[ appenderName1 ]** ,  **[ appenderName2 ]** ,  …  
http://www.blogjava.net/Images/OutliningIndicators/None.gif  
http://www.blogjava.net/Images/OutliningIndicators/None.gif#配置日志信息输出目的地Appender  
http://www.blogjava.net/Images/OutliningIndicators/None.giflog4j.appender. appenderName1 = **[ appender  ]**  
http://www.blogjava.net/Images/OutliningIndicators/None.gif　　log4j.appender. appenderName1.option1  =  value1   
http://www.blogjava.net/Images/OutliningIndicators/None.gif　　**…**  
http://www.blogjava.net/Images/OutliningIndicators/None.gif　　log4j.appender.appenderName.optionN  =  valueN   
http://www.blogjava.net/Images/OutliningIndicators/None.gif  
http://www.blogjava.net/Images/OutliningIndicators/None.gif#配置日志信息的格式（布局）  
http://www.blogjava.net/Images/OutliningIndicators/None.giflog4j.appender.appenderName.layout  =  fully.qualified.name.of.layout.class   
http://www.blogjava.net/Images/OutliningIndicators/None.gif　　log4j.appender.appenderName.layout.option1  =  value1   
http://www.blogjava.net/Images/OutliningIndicators/None.gif　　…   
http://www.blogjava.net/Images/OutliningIndicators/None.gif　　log4j.appender.appenderName.layout.optionN  =  valueN

**Logger** - 日志写出器，供程序员输出日志信息，其第一位**[** **level** **]**为日志优先级(**优先级：ALL < DEBUG < INFO <WARN < ERROR < FATAL < OFF)**，如果**[** **level** **]**配置 OFF 则不打出任何信息，如果配置为 INFO 这样只显示 INFO、WARN、ERROR 的 log 信息，而 DEBUG 信息不会被显示。

**appenderName** -Appender的名字，名字自己随便取，在下面对自定义的appenderName进行配置  
    **Appender** - 日志目的地，把格式化好的日志信息输出到指定的地方去。有如下三种常用的Appender

**1.org.apache.log4j.ConsoleAppender** - 目的地为控制台的 Appender,其option和value如下：

**Threshold=WARN**:指定日志消息的输出最低层次。  
**ImmediateFlush=true**:默认值是 true,意谓着所有的消息都会被立即输出。  
 **Target=System.err**：默认情况下value是：System.out,指定输出控制台

**2.org.apache.log4j.FileAppender** - 目的地为文件的 Appender

**Threshold=WARN**:指定日志消息的输出最低层次。  
        **ImmediateFlush=true**:默认值是 true,意谓着所有的消息都会被立即输出。  
        **File=mylog.txt**:指定消息输出到 mylog.txt 文件。  
        **Append=false:**默认值是 true,即将消息增加到指定文件中，false 指将消息覆盖指定的文件内容。

**3.org.apache.log4j.DailyRollingFileAppender** -每天产生一个日志文件的 Appender

**Threshold=WARN**:指定日志消息的输出最低层次。  
        **ImmediateFlush=true:**默认值是 true,意谓着所有的消息都会被立即输出。  
        **File=mylog.txt:**指定消息输出到 mylog.txt 文件。  
        **Append=false:**默认值是 true,即将消息增加到指定文件中，false 指将消息覆盖指定的文件内容。  
        **DatePattern=''.''yyyy-ww:**每周滚动一次文件，即每周产生一个新的文件。  
  
        当然也可以指定按月、周、天、时和分。即对应的格式如下：  
        1)**''.''yyyy-MM**: 每月  
        2)**''.''yyyy-ww**: 每周  
        3)**''.''yyyy-MM-dd**: 每天  
        4)**''.''yyyy-MM-dd-a**: 每天两次  
        5)**''.''yyyy-MM-dd-HH**: 每小时  
        6)**''.''yyyy-MM-dd-HH-mm**: 每分钟

**4.org.apache.log4j.RollingFileAppender** - 文件大小到达指定尺寸的时候产生一个新的文件

**Threshold=WARN**:指定日志消息的输出最低层次。  
       **ImmediateFlush=true**:默认值是 true,意谓着所有的消息都会被立即输出。  
        **File=mylog.log**:指定消息输出到 mylog. log文件。  
        **Append=false**:默认值是 true,即将消息增加到指定文件中，false 指将消息覆盖指定的文件内容。  
        **MaxFileSize=100KB**: 后缀可以是 KB, MB 或者是 GB. 在日志文件到达该大小时，将会自动滚动，即将原来的内容移到 mylog.log.1 文件。  
        **MaxBackupIndex=2**:指定可以产生的滚动文件的最大数。

**5.org.apache.log4j.WriterAppender** - 将日志信息以流格式发送到任意指定的地方

**Layout** - 日志格式化器，用来把程序员的 logging request 格式化成字符串

**1) HTMLLayout  选项**  
       **LocationInfo=true**:默认值是 false,输出 java 文件名称和行号  
       **Title=my app file**: 默认值是 Log4J Log Messages.  
**2) PatternLayout  选项**  
       **ConversionPattern=%m%n** :指定怎样格式化指定的消息。  
  
       **3) XMLLayout  选项**

**LocationInfo=true**:默认值是 false,输出 java 文件和行号   
       Log4J 采用类似 C 语言中的 printf 函数的打印格式格式化日志信息，打印参数如下：  
       og4j.appender.A1.layout.ConversionPattern=%-4r %-5p %d{yyyy-MM-dd HH:mm:ssS} %c %m%n

**PatternLayout** - 用指定的 pattern 格式化 logging request 的 Layout

**下面是log4j.properties配置文件的一个例子**

### 设置日志等级 ###

log4j.rootLogger = debug , stdout , log , errorLog  
  
### 输出到控制台 ###  
log4j.appender.stdout = org.apache.log4j.ConsoleAppender  
log4j.appender.stdout.Target = System.out  
log4j.appender.stdout.layout = org.apache.log4j.PatternLayout  
log4j.appender.stdout.layout.ConversionPattern = %d{ABSOLUTE} %5p %c{1}:%L - %m%n  
  
### 输出到日志文件 ###  
log4j.appender.log = org.apache.log4j.DailyRollingFileAppender  
log4j.appender.log.File = log/log.log  
log4j.appender.log.Append = true  
## 输出DEBUG级别以上的日志  
log4j.appender.log.Threshold = DEBUG  
log4j.appender.log.layout = org.apache.log4j.PatternLayout  
log4j.appender.log.layout.ConversionPattern = %-d{yyyy-MM-dd HH:mm:ss} [ %t:%r ] - [ %p ] %m%n  
  
### 保存异常信息到单独文件 ###  
log4j.appender.errorLog = org.apache.log4j.DailyRollingFileAppender  
## 异常日志文件名  
log4j.appender.errorLog.File = log/error.log  
log4j.appender.errorLog.Append = true  
### 只输出ERROR级别以上的日志!!!  
log4j.appender.errorLog.Threshold = ERROR  
log4j.appender.errorLog.layout = org.apache.log4j.PatternLayout  
log4j.appender.errorLog.layout.ConversionPattern = %-d{yyyy-MM-dd HH:mm:ss} [ %t:%r ] - [ %p ] %m%n

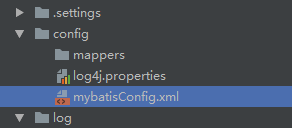
### 1.5.2使用

import org.apache.log4j.Logger;  
  
public class TestLog4J {  
 public static void main(String[] args) {  
 Logger logger = Logger.getLogger(TestLog4J.class);  
 logger.info("我是信息");  
 }  
  
}

# Mybatis使用

## 2.1 配置文件SqlMapConfig.xml

SqlMapConfig.xml是mybatis的配置文件，我们将其放置在根目录的config文件夹下，文件名可以随意取，在config文件夹下再创建一个mappers文件夹，用于存放mapper.xml，如下图



SqlMapConfig.xml有八项配置内容，其**配置顺序不可以打乱**，依次为：

1. properties(加载属性文件)
2. settings(配置全局参数)
3. typeAiases(设置类型别名)
4. typeHandlers(设置类型处理器)
5. objectFactory(设置对象工厂)
6. plugins(设置插件)
7. environments(设置mybatis环境集合属性对象)
   1. environment(环境子属性对象)
   2. transactionManager(事物管理)
   3. datesource(数据源)
8. mappers(配置映射文件)

需要注意：<environments>是用来配置mybatis的环境的，在和[spring](http://lib.csdn.net/base/17)整合前是由mybatis自己管理的，但是和Spring整合后，这些配置都将被废弃掉。<objectFactory>和<plugins>一般我们不用。<typeHandlers>类型处理器是mybatis中用来完成jdbc类型和[Java](http://lib.csdn.net/base/17)类型之间转换的，mybatis默认支持的类型处理器基本够我们使用了，一般不需要自定义。也就是说我们需要重点关注的是**1,2,3,8**

<?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>  
 <!-- &lt;!&ndash; 加载java的配置文件或者声明属性信息 &ndash;&gt;  
 <properties resource="db.properties"/>-->  
 <settings><!-- -->  
 <!-- 懒加载设置为 true -->  
 <setting name="lazyLoadingEnabled" value="true"/>  
 <!-- 积极加载设置为false -->  
 <setting name="aggressiveLazyLoading" value="false"/>  
 <setting name="lazyLoadTriggerMethods" value="equals,clone,hashCode"/>  
 <setting name="cacheEnabled" value="true"/>  
 </settings>  
 <!-- 自定义别名 -->  
 <typeAliases>  
 <package name="com.po"/>  
 </typeAliases>  
  
 <!-- 配置mybatis的环境信息，与spring整合，该信息由spring来管理 -->  
 <!-- <environments default="druidEnvironment">  
 <environment id="druidEnvironment">  
 &lt;!&ndash; 配置JDBC事务控制，和spring整合后，事务配置无效，会被mybatis-spring包中的事务配置覆盖 &ndash;&gt;  
 <transactionManager type="JDBC"/>  
 &lt;!&ndash; 配置数据源，采用druid连接池 &ndash;&gt;  
 <dataSource type="com.DruidPool">  
 <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>-->  
  
 <!-- 加载映射文件 -->  
 <mappers>  
 <!-- <mapper class="com.mapper.StudentMapper"/>-->  
 <!-- 批量加载映射文件 -->  
 <mapper resource="sqlMap/StudentMapper.xml"/>  
 <package name="com.mapper"/>  
 </mappers>  
</configuration>

### 2.1.1 properties加载属性文件

我们之前对于数据库连接的一些配置如数据库ip地址，账号，密码等信息都是在<environments>中写死的，但在实际开发中，为了保证灵活性，我们需要将数据库连接参数单独配在db.properties文件中，然后在SqlMapConfig.xml中加载db.properties文件中的属性值即可，这就用到了<properties>节点了

在**db.properties文件**中我们这样配置（请注意，如果是配置在xml文件中，&符号应用&amp;来替代）：

jdbc.driver=com.mysql.cj.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/cal?serverTimezone=Asia/Shanghai&useSSL=false&characterEncoding=utf-8  
jdbc.username=root  
jdbc.password=123456

然后在mybatisConfig.xml中这样配置(所有要配置的选项都是配置在<configuration>下的)：

<?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>  
  **<!-- 加载数据库文件db.properties -->**  
 <properties resource="db.properties">  
  **<!-- properties中还可以配置一些属性名和属性值,但为避免搞乱，不推荐 -->  
 <!-- <property name="driver" value=""/> -->**  
 </properties>  
</configuration>

需要注意的是：

属性的加载顺序为：最先加载**properties**元素体内的属性，然后加载**properties**文件中的属性**properties**的属性会覆盖**properties**元素体内的属性，最后，mapper.xml文件中的sql语句中如果$占位符有对应的属性，该$占位符会被替换成属性值，eg：

jdbc.driver=com.mysql.cj.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/laravel?serverTimezone=Asia/Shanghai&useSSL=false&characterEncoding=utf-8  
jdbc.username=root  
jdbc.password=123456  
item=Value

<delete id="deleteStudent" parameterType="int" >  
 DELETE FROM `student` WHERE ${item}=#{id}  
</delete>

最后执行的sql语句为：



因此**jdbc.**还是非常必要的

### 2.1.2 settings全局参数配置

mybatis框架运行时可以调整一些运行参数。比如，开启二级缓存，开启延迟加载等等。全局参数会影响mybatis的运行行为。



打印sql日志到控制台：

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-config.dtd">  
<configuration>  
 <settings>  
 <!--打印查询语句-->  
 <setting name="logImpl" value="STDOUT\_LOGGING" />  
 </settings>

</configuration>

<bean id="assetSqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  
 <property name="dataSource" ref="asset"/>  
 <property name="configLocation" value="classpath:mapper/configuration/mybatis-configuration.xml"/>  
 <property name="mapperLocations" value="classpath:mapper/\*.xml"></property>  
</bean>

### 2.1.3 typeAliases类型别名(重点内容)

　 在mapper.xml中，定义很多的sql语句statement，statement需要parameterType指定输入参数的类型、需要resultType指定输出结果的映射类型。

如果在指定类型时输入类型全路径，不方便进行开发，可以针对parameterType或resultType指定的类型定义一些别名，在mapper.xml中通过别名定义，方便开发。

#### 2.1.3.1 mybatis默认支持的别名

mybatis对基本数据类型指定了默认别名：

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

#### 2.1.3.2 自定义别名

基本类型mybatis已经为我们定义好了别名，但是pojo类型就要由我们自行定义别名了，自定义别名也有两种方式：单个定义和批量定义

**1.单个定义**

<!-- 自定义别名 -->  
<typeAliases>  
 <!-- 单个别名定义 -->  
 <typeAlias type="com.test.mybatis.pojo.User" alias="user"/>

<typeAlias type="com.test.mybatis.pojo.Visitor" alias="visitor"/>  
</typeAliases>

**2.批量定义(推荐)**

<!-- 自定义别名 -->  
<typeAliases>  
 <!-- package：指定包名称来为该包下的所有pojo类声明别名，默认的别名就是类名（首字母大小写都可） -->  
 <package name="com.test.mybatis.pojo"/>  
</typeAliases>

通过如上方式定义别名后，当输入参数或者输出结果为**com.test.mybatis.pojo.User**时，就可以用user来代替了，如下：

<select id="findUserById" parameterType="int" resultType="user" >  
　　select \* from t\_user where id=#{id}  
</select>

等价于

<select id="findUserById" parameterType="int" resultType="com.test.mybatis.pojo.User" >  
　　select \* from t\_user where id=#{id}  
</select>

### 2.1.4 typeHandlers类型处理器

mybatis通过typeHandlers来完成jdbc类型(数据库的数据类型)和java类型间的转换，一般情况下，mybatis默认支持的类型处理器基本够我们使用了，一般不需要自定义。

### 2.1.5 environments数据源配置

<environments>下面可以配置多个<environment>元素节点，而每个<environment>节点我们可以配置两个东西，一个是事务管理器配置<transactionManager>，另一个是数据源配置<dataSource>。

<!-- 配置mybatis的环境信息，与spring整合，该信息由spring来管理 -->  
<environments default="development">  
 <environment id="development">  
 <!-- 配置JDBC事务控制，由mybatis进行管理 -->  
 <transactionManager type="JDBC"/>  
 <!-- 配置数据源，采用mybatis连接池 -->  
 <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>

首先获取**<environments>**节点的default属性，拿到默认的environment的id后，载入对应的数据库配置(一个environment就是一个数据源配置，尽管可以配置多个environment，但mybatis只能选择一个)

#### 2.1.5.1 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:

<transactionManager type="MANAGED">  
 <property name="closeConnection" value="false"/>  
</transactionManager>

不过**如果使用spring整合mybatis，事务管理器就没有配置的必要了**，因为 Spring 模块会使用自带的管理器来覆盖前面的配置，整合时只需要在spring的配置文件applicationContext.xml中配置数据源即可

#### 2.1.5.2 数据源

Mybatis中支持三种形式数据源的配置，分别为：UNPOOLED、POOLED和JNDI

1. **UNPOOLED**    不使用连接池的数据源，适用于一些对数据库连接可用性方面没有太高要求的简单应用程序

该配置有以下属性：

(1)driver – 这是 JDBC 驱动的 Java 类的完全限定名（并不是 JDBC 驱动中可能包含的数据源类）。

(2)url – 这是数据库的 JDBC URL 地址。

(3)username – 登录数据库的用户名。

(4)password – 登录数据库的密码。

(5)defaultTransactionIsolationLevel – 默认的连接事务隔离级别。

eg：

<!-- 配置数据源，不使用连接池-->  
<dataSource type="UNPOOLED">  
 <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>

2.**POOLED**      使用连接池的数据源，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。

该配置除了要配置**UNPOOLED**中的属性，还可以配置如下属性：

 (1)poolMaximumActiveConnections – 在任意时间可以存在的活动（也就是正在使用）连接数量，默认值：10

 (2)poolMaximumIdleConnections – 任意时间可能存在的空闲连接数。

 (3)poolMaximumCheckoutTime – 在被强制返回之前，池中连接被检出（checked out）时间，默认值：20000 毫秒（即 20 秒）

 (4)poolTimeToWait – 这是一个底层设置，如果获取连接花费了相当长的时间，连接池会打印状态日志并重新尝试获取一个连接（避免在误配置的情况下一直安静的失败），默认值：20000 毫秒（即 20 秒）。

 (5)poolMaximumLocalBadConnectionTolerance – 这是一个关于坏连接容忍度的底层设置， 作用于每一个尝试从缓存池获取连接的线程. 如果这个线程获取到的是一个坏的连接，那么这个数据源允许这个线程尝试重新获取一个新的连接，但是这个重新尝试的次数不应该超过 poolMaximumIdleConnections 与 poolMaximumLocalBadConnectionTolerance 之和。 默认值：3 (新增于 3.4.5)

 (6)poolPingQuery – 发送到数据库的侦测查询，用来检验连接是否正常工作并准备接受请求。默认是“NO PING QUERY SET”，这会导致多数数据库驱动失败时带有一个恰当的错误消息。

 (7)poolPingEnabled – 是否启用侦测查询。若开启，需要设置 poolPingQuery 属性为一个可执行的 SQL 语句（最好是一个速度非常快的 SQL 语句），默认值：false。

 (8)poolPingConnectionsNotUsedFor – 配置 poolPingQuery 的频率。可以被设置为和数据库连接超时时间一样，来避免不必要的侦测，默认值：0（即所有连接每一时刻都被侦测 — 当然仅当 poolPingEnabled 为 true 时适用）

3.**JNDI**            使用JNDI实现的数据源，该配置能在如 EJB 或应用服务器这类容器中使用，容器可以集中或在外部配置数据源，然后放置一个 JNDI 上下文的引用。这种数据源配置只需要两个属性：

(1)**initial\_context** – 这个属性用来在 InitialContext 中寻找上下文（即，initialContext.lookup(initial\_context)）。这是个可选属性，如果忽略，那么 data\_source 属性将会直接从 InitialContext 中寻找。

(2)**data\_source** – 这是引用数据源实例位置的上下文的路径。提供了 initial\_context 配置时会在其返回的上下文中进行查找，没有提供时则直接在 InitialContext 中查找。

#### 2.1.5.3 数据库连接池原理

数据库连接池在初始化时将创建一定数量的数据库连接放到连接池中，数据库池连接数量一直保持一个不少于最小连接数的数量，每次连接数量不够时，数据库都会创建新的连接，直到一个最大连接数，之后如果再出现连接数量不够用的情况，就会将多出来的连接请求加入队列中等待。

**数据库连接池初始化时，数据库连接的数量是由最小数据库连接数来设定的**，无论这些数据库连接是否被使用，连接池都将一直保证**至少**拥有这么多的连接数量。以后当连接不够用时且需要的连接数未超过最大连接数，就会创建新的数据库连接，连接使用完毕后也会加入连接池中，**最大数据库连接数**限定了这个连接池能持有的最大连接数，当应用程序向连接池请求的连接数超过最大连接数量时，这些请求将被加入到等待队列中。

下面将介绍mybatis使用数据库连接池的过程

**MyBatis数据源对象DataSource的创建发生在MyBatis配置文件初始化的过程中。**通过配置文件创建对应的DataSource对象，即：

type=”POOLED”  MyBatis会创建PooledDataSource实例

type=”UNPOOLED” MyBatis会创建UnpooledDataSource实例

type=”JNDI”     MyBatis会从JNDI服务上查找DataSource实例，然后返回使用

MyBatis创建了对应的DataSource实例后，会将其放到Configuration对象内的Environment对象中， 供以后使用。

那么什么时候MyBatis才会去调用DataSource对象来创建java.sql.Connection对象呢？答案是当执行sql语句的时候，也就是说只有执行到如sqlSession.selectList()之类的方法时，才会触发MyBatis创建java.sql.Connection对象，对应于不同的DataSource对象，其获取java.sql.Connection对象的方式也是不同的：

**UnpooledDataSource** 该对象的getConnection方法，每调用一次都会产生一个**新的**Connection实例对象。执行一次sql语句就要创建一个connection对象来连接数据库，用完释放掉，下次用再创建，用完再释放，这种方式是极度消耗性能的，甚至可能比sql语句的执行时间都要长

**PooledDataSource** 该对象的getConnection方法，先是调用类PooledDataSource的popConnection()方法返回了一个PooledConnection对象，然后调用了PooledConnection的getProxyConnection()来返回Connection对象。**而在使用完connection对象后，并不释放，而是将其放到数据库连接池中进行维护，等待下次使用**

#### 2.1.5.3 配置第三方数据库连接池druid

mybatis可以帮助我们管理数据库连接池，但我们一般不用，常用的第三方数据库连接池有：dbcp(性能不佳，bug多)、c3p0(性能差、代码老，扩展性差)、protocol、Druid(阿里开源，据说是目前最好的数据库连接池)

druid的maven代码如下：

<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>druid</artifactId>  
 <version>${druidVersion}</version>  
</dependency>

这里提供未与spring整合时的druid配置

我们需要将DataSource的type属性设置为DataSource工厂，但因为**com.alibaba.druid.pool.DruidDataSourceFactory**对象和**org.apache.ibatis.datasource.DataSourceFactory**接口不一致，因此我们需要包装一下：

package com;  
import com.alibaba.druid.pool.DruidDataSourceFactory;  
import org.apache.ibatis.datasource.DataSourceFactory;  
import javax.sql.DataSource;  
import java.util.Properties;  
  
public class DruidPool extends DruidDataSourceFactory implements DataSourceFactory{  
 protected Properties properties;  
 public void setProperties(Properties properties) {  
 this.properties=properties;  
 }  
  
 public DataSource getDataSource() {  
 try {  
 return *createDataSource*(this.properties);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 return null;  
 }  
}

然后配置：

<!-- 配置mybatis的环境信息，与spring整合，该信息由spring来管理 -->  
<environments default="druidEnvironment">  
 <environment id="druidEnvironment">  
 <!-- 配置JDBC事务控制，和spring整合后，事务配置无效，会被mybatis-spring包中的事务配置覆盖 -->  
 <transactionManager type="JDBC"/>  
 <!-- 配置数据源，采用druid连接池，DruidPool是我们包装好的类 -->  
 <dataSource type="com.DruidPool">  
 <property name="driverClassName" value="${jdbc.driver}"/>  
 <property name="url" value="${jdbc.url}"/>  
 <property name="username" value="${jdbc.username}"/>  
 <property name="password" value="${jdbc.password}"/>  
 </dataSource>  
 </environment>  
</environments>

除了上例中的url、maxActive等属性，druid还提供其他的属性配置

|  |  |  |
| --- | --- | --- |
| 配置 | 缺省值 | 说明 |
| name |  | 配置这个属性的意义在于，如果存在多个数据源， 监控的时候可以通过名字来区分开来。 如果没有配置，将会生成一个名字， 格式是：”DataSource-” + System.identityHashCode(this) |
| jdbcUrl |  | 连接数据库的url，不同数据库不一样。例如：  mysql : jdbc:mysql://10.20.153.104:3306/druid2  oracle : jdbc:oracle:thin:@10.20.149.85:1521:ocnauto |
| username |  | 连接数据库的用户名 |
| password |  | 连接数据库的密码。如果你不希望密码直接写在配置文件中， 可以使用ConfigFilter。详细看这里：https://github.com/alibaba/druid/wiki/%E4%BD%BF%E7%94%A8ConfigFilter |
| driverClassName | 根据url自动识别 | 这一项可配可不配，如果不配置druid会根据url自动识别dbType， 然后选择相应的driverClassName(建议配置下) |
| initialSize | 0 | 初始化时建立物理连接的个数。初始化发生在显示调用init方法，或者第一次getConnection时 |
| maxActive | 8 | 最大连接池数量，如果当前连接池中连接不够用，且未达到最大连接数，则会创建新连接，如果超过最大连接数，则不会创建新连接，而是将其加入队列中等待 |
| maxIdle | 8 | 已经不再使用，配置了也没效果 |
| minIdle |  | 最小连接池数量，与initialSize不同，启动服务后不会立刻创建minIdle个，而是创建initialSize个。但随着请求增加，等连接数超过minIdle个以后，就不在会低于minIdle，一旦低了就会创建新的补足。 |
| maxWait |  | 获取连接时最大等待时间，单位毫秒。 配置了maxWait之后，缺省启用公平锁，并发效率会有所下降， 如果需要可以通过配置useUnfairLock属性为true使用非公平锁。 |
| poolPreparedStatements | false | 是否缓存preparedStatement，也就是PSCache。 PSCache对支持游标的数据库性能提升巨大， 比如说oracle。在mysql下建议关闭。 |
| maxOpenPreparedStatements | -1 | 要启用PSCache，必须配置大于0，当大于0时， poolPreparedStatements自动触发修改为true。 在Druid中，不会存在Oracle下PSCache占用内存过多的问题， 可以把这个数值配置大一些，比如说100 |
| validationQuery |  | 用来检测连接是否有效的sql，要求是一个查询语句。 如果validationQuery为null，testOnBorrow、testOnReturn、 testWhileIdle都不会其作用。 |
| testOnBorrow | true | 申请连接时执行validationQuery检测连接是否有效， 做了这个配置会降低性能。 |
| testOnReturn | false | 归还连接时执行validationQuery检测连接是否有效， 做了这个配置会降低性能 |
| testWhileIdle | false | 建议配置为true，不影响性能，并且保证安全性。申请连接的时候 检测，如果空闲时间大于timeBetweenEvictionRunsMillis， 执行validationQuery检测连接是否有效。 |
| timeBetweenEvictionRunsMillis |  | 有两个含义：  1) 配置Destroy线程检测需要关闭的空闲连接的间隔时间，单位毫秒  2) testWhileIdle的判断依据，详细看testWhileIdle属性的说明 |
| numTestsPerEvictionRun |  | 不再使用，一个DruidDataSource只支持一个EvictionRun |
| minEvictableIdleTimeMillis |  |  |
| connectionInitSqls |  | 物理连接初始化的时候执行的sql |
| exceptionSorter | 根据dbType自动识别 | 当数据库抛出一些不可恢复的异常时，抛弃连接 |
| filters |  | 属性类型是字符串，通过别名的方式配置扩展插件， 常用的插件有：  监控统计用的filter:stat日志用的filter:log4j 防御sql注入的filter:wall |
| proxyFilters |  | 类型是List<com.alibaba.druid.filter.Filter>， 如果同时配置了filters和proxyFilters，是组合关系，并非替换关系 |

### 2.1.6 mappers映射器

mapper映射文件用于配置sql语句，我们可以配置多个mapper.xml文件

有以下几种方式可以帮助我们配置mapper映射器

#### 2.1.6.1 使用相对于类路径的资源

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

#### 2.1.6.2 使用硬盘绝对路径

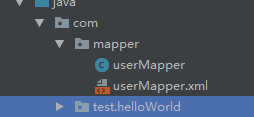
<mappers>  
 <mapper url="file:///E:\Study\JAVA\mySpringMVC\config\mappers/userMapper.xml"></mapper>  
</mappers>

#### 2.1.6.3 使用mapper接口加载(详见2.5.2)

通过mapper接口来加载mapper.xml映射文件，需要遵循以下规范：

1.使用mapper代理的方式开发DAO(详见2.5.2)

2. mapper接口的实现类的类名和mapper.xml映射文件名称保持一致，且在一个目录中



<mapper class="com.mapper.userMapper"/>

#### 2.1.6.4 批量加载mapper(推荐)

批量加载也必须按照2.1.6.3的规范来

<!-- 批量加载映射配置文件,mybatis自动扫描包下面的mapper接口进行加载  
 遵循一定的规范：需要将mapper接口类名和mapper.xml映射文件名称保持一致，且在一个目录中；  
 上边规范的前提是：使用的是mapper代理方法; -->  
<package name="com.mapper"/>

一定要注意，如果希望同时使用package和mapper标签加载mapper.xml文件，一定要mapper在前，package在后，否则报错，这是由于xml的DTD约束为：

<!ELEMENT mappers (mapper\*,package\*)>

正确的使用方式为：

<mappers>  
 <mapper resource="sqlMap/StudentMapper.xml"/>

<!-- 批量加载映射文件 -->  
 <package name="com.mapper"/>  
</mappers>

## 2.2 映射文件mapper.xml

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

<!DOCTYPE mapper  
 PUBLIC "-//ibatis.apache.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
  
<!-- mapper 为根元素节点， 一个namespace对应一个dao -->  
<mapper namespace="com.test.dao.UserDao">  
</mapper>

我们的<select>、<update>、<insert>、<delete>等标签都是写在里面的

### 2.2.1 select(查询)

<select id="selectPerson" parameterType="int" resultType="hashmap">  
 SELECT \* FROM PERSON WHERE ID = #{id}  
</select>

比如上例就是创建了一个id为**selectPerson**的查询语句，该语句传入参数为**int**类型，输出参数为**hashmap**类型，**hashmap**中的**key**是列名，**value**便是值，**#{id}**符号是告诉mybatis创建一个预处理语句参数，在jdbc中，该符号用?来标识，上述语句等价为：

String selectPerson = "SELECT \* FROM PERSON WHERE ID=?";  
PreparedStatement ps = conn.prepareStatement(selectPerson);  
ps.setInt(1,id);

|  |  |
| --- | --- |
| **属性** | **描述** |
| **id** | 在命名空间中唯一的标识符，可以被用来引用这条语句。 |
| **parameterType** | 将会传入这条语句的参数的类型名或别名。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。 |
| **resultType** | 从这条语句中返回的结果集的类型名或别名。**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，也就是说查询多条记录和单条记录该属性值应当是一样的**。使用 resultType 或 resultMap，但不能同时使用。 |
| **resultMap** | 外部 resultMap 的命名引用。使用 resultMap 或 resultType，但不能同时使用。 |
| **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：false。 |
| **useCache** | 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。 |
| **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）。 |
| **fetchSize** | 这是尝试影响驱动程序每次批量返回的结果行数和这个设置值相等。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。 |
| **resultSetType** | FORWARD\_ONLY，SCROLL\_SENSITIVE 或 SCROLL\_INSENSITIVE 中的一个，默认值为 unset （依赖驱动）。 |
| **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。 |
| **resultOrdered** | 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。 |
| **resultSets** | 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。 |

### 2.2.2 insert、update、delete

数据变更语句 insert，update 和 delete 的实现非常接近：

|  |  |
| --- | --- |
| **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** | 将会传入这条语句的参数的类型名或别名。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。 |
| **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）当设置为 true 时，表示如果插入的表以自增列为主键，则允许 JDBC 支持自动生成主键，并可将自动生成的主键返回。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(配合**useGeneratedKeys**使用，一般填入表的主键) 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 id="insertAuthor" useGeneratedKeys="true" keyProperty="id">  
 insert into Author (username,password,email,bio)  
 values (#{username},#{password},#{email},#{bio})  
</insert>

如果希望进行多行插入

<insert id="insertAuthor" useGeneratedKeys="true" keyProperty="id">  
 insert into Author (username, password, email, bio) values  
 <foreach item="item" collection="list" separator=",">  
 (#{item.username}, #{item.password}, #{item.email}, #{item.bio})  
 </foreach>  
</insert>

在insert或者update时，有时希望将刚刚插入或者修改的字段的主键返回到pojo对象中比如上面的insert语句，我们new了一个pojo对象，该对象有**username,password,email,bio**等属性，将它插入表中得到一条新的数据，数据库会自动为我们设置主键(如果主键自增的话)，然而pojo对象的id属性却没有新增记录的主键值，而mybatis提供了一个selectKey帮助我们解决这一问题

<insert id="insertAuthor" useGeneratedKeys="true" keyProperty="id">  
 insert into Author (username,password,email,bio)  
 values (#{username},#{password},#{email},#{bio})  
 <selectKey keyProperty="id" order="AFTER" resultType="java.lang.Integer">  
 SELECT LAST\_INSERT\_ID()  
 </selectKey>  
</insert>

它通过**LAST\_INSERT\_ID()**函数查询最后一次插入的id，并将其填入**keyProperty**字段中，这里请注意，last\_insert\_id函数的返回值不是基于整个数据表的插入语句，而是单个连接客户端之间所执行的insert语句的最后一条，因此不需要担心并发问题

请注意！order必须为**AFTER**！这意味着selectKey会在insert后执行，这样才能获取到id

https://segmentfault.com/q/1010000011126611/a-1020000011133986

### 2.2.3 sql片段

这个元素可以被用来定义可重用的 SQL 代码段，可以被包含在其他语句中。它可以被静态地参数化。

<sql id="userColumns"> ${alias}.id,${alias}.username,${alias}.password </sql>  
<select id="selectUsers" resultType="map">  
 select  
 <include refid="userColumns"><property name="alias" value="table\_1"/></include>,  
 <include refid="userColumns"><property name="alias" value="table\_2"/></include>  
 from table\_1 left join table\_2 on table\_1.id=table\_2.uid  
</select>

甚至还可以有多条sql嵌套使用，$占位符也可以替换refid的属性值：

<sql id="sometable">  
 ${prefix}Table  
</sql>

<sql id="someinclude">  
 from  
 <include refid="${include\_target}"/>  
</sql>  
<select id="select" resultType="map">  
 select  
 field1, field2, field3  
 <include refid="someinclude">  
 <property name="prefix" value="Some"/>  
 <property name="include\_target" value="sometable"/>  
 </include>  
</select>

### 2.2.4 #{ }和${ }的区别

<select id="findStudentById\_1" parameterType="String" resultType="Student">  
 SELECT \* FROM Student WHERE name=#{name}  
</select>  
<select id="findStudentById\_2" parameterType="String" resultType="Student">  
 SELECT \* FROM Student WHERE name=${value}   
</select>

上述代码展示了两种写法($和#)，那么这两种写法有什么区别呢？

动态 SQL 是 mybatis 的强大特性之一，在动态解析sql时：

**1.#{ }**

#{} 在预处理时，会把参数部分用一个占位符 ? 代替，eg：

**SELECT \* FROM Student WHERE name=#{name}会被解析成为SELECT \* FROM Student WHERE name=?**

**2.${ }**

${}在预处理时，会将我们传入的参数当做String字符串填充到我们的语句中，只是做一个简单的字符串替换，eg：

**SELECT \* FROM Student WHERE name=#{value}会被解析成为SELECT \* FROM Student WHERE name="李四"**

**1.#方式能够很大程度防止sql注入攻击，而$方式无法防止Sql注入。eg：**

**select \* from ${value} where name = #{name}**

**如果给${value}填入 user; delete user; -- ，那么sql语句就会变成两句：**

**select \* from  user; delete user; --  where name = "李四"**

**第一句查询user的所有用户，第二句则直接将user表删除**

**2.$方式一般用于传入数据库对象，最常见的是传入表名**

**3.千万注意，$和#都能接受简单类型、pojo类型和hashmap**

**#{} 在接收简单类型时，可以使用#{value}或#{任意名称}**

**在接收pojo对象时，若pojo对象的属性仍为pojo对象，可通过 . 符号来访问内层pojo的属性，如#{class.teacher.name}就是访问班级(class)pojo的属性老师(teacher)pojo的name属性**

**${} 在接收简单类型时，只能使用${value}**

**在接收pojo对象时，也可以使用 . 访问内层pojo属性，但不推荐$接收pojo对象**

**4.尽量的使用#方式，一般能用#的就别用$**

### 2.2.5 resultType和resultMap

在MyBatis进行查询映射时，其实查询出来的每一个属性都是放在一个对应的Map里面的，其中键是属性名，值则是其对应的值

**1. resultType**

当提供的返回类型属性是resultType时，MyBatis会将Map里面的键值对取出赋给resultType所指定的对象对应的属性。因此要求resultType所指定的类其属性名必须与查询的字段保持一致。否则无法正常赋值(映射)

**2. resultMap**

mybatis可以通过resultMap完成高级的结果映射(一对一、一对多、多对多)

如果查询出的列名和要映射的pojo对象属性名不一致，可以通过resultMap进行映射

我们可以在mapper.xml映射文件的mapper节点下添加**<resultMap>**节点(和select，insert等节点同级)来定义查询结果列名和pojo对象的属性名之间的关系，eg：

type属性指定要映射的pojo对象类型

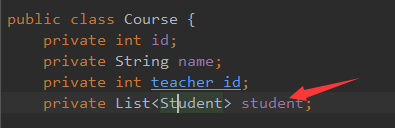
id标签指定表的主键和pojo属性的关联(如果查询结果中没有主键，可以不写)

result标签则是普通列名和pojo属性的关联

colum是查询结果的列名，property是pojo属性

<resultMap id="myResultMap" type="Student">  
 <id column="id" property="id"/>  
 <result column="name\_" property="name"/>  
 <result column="sex\_" property="sex"/>  
</resultMap>  
  
<select id="findStudentById" resultMap="myResultMap">  
 SELECT name as name\_,sex as sex\_ FROM Student WHERE name=#{myName} AND num=#{myNum}  
</select>

resultMap下还有一个collection节点，该节点用于将sql查询到的多条记录映射到一个带集合属性的类中去，eg：



property属性是要映射的类的属性名，javaType是集合的类型，ofType是集合中元素的类型

<resultMap id="courseStudent" type="Course">  
 <id column="course\_id" property="id"></id>  
 <result column="course\_name" property="name"></result>  
 <result column="teacher\_id" property="teacher\_id"></result>  
 <collection property="student" javaType="List" ofType="Student">  
 <id column="student\_id" property="id"></id>  
 <result column="num" property="num"></result>  
 <result column="age" property="age"></result>  
 <result column="student\_name" property="name"></result>  
 <result column="sex" property="sex"></result>  
 </collection>  
</resultMap>

### 2.2.6 动态sql

在复杂的业务需求中，我们常常需要根据不同条件来拼接sql语句，试想一下，我们在进行多条件查询，多个条件也就是需要传入多个参数，如果某个参数为null，则不查询该参数对应的字段，这样一来，多个参数可能会出现多种情况，比如要传入name和sex参数，如果name和sex非空，则sql为SELECT \* FROM Student WHERE name=#{myName} AND sex=#{mySex}，如果传入的sex参数为null，我们则希望sql变为：SELECT \* FROM Student WHERE name=#{myName}，动态sql可以帮助我们解决这一问题

#### 2.1.6.1 if

**<if>**标签可以帮助我们对条件进行判断

<select id="searchStudent" parameterType="String" resultType="Student">  
 SELECT \* FROM Student WHERE name = '张亭'  
 <if test="sex != null and sex != ''">  
 AND sex = #{sex}  
 </if>  
</select>

#### 2.1.6.2 where

上面的<if>标签已经解决了一个麻烦的sql拼接问题，但仍然不够完善，假设name这一条件也需要判断

<select id="searchStudent" parameterType="String" resultType="Student">

SELECT \* FROM Student WHERE  
 <if test="name != null and name != ''">  
 AND name = #{name}  
 </if>

<if test="sex != null and sex != ''">  
 AND sex = #{sex}  
 </if>  
</select>

那么sql语句有可能变为**SELECT \* FROM Student WHERE AND name = #{name}**，这可不是我们想要的结果。所幸，mybatis已经考虑到了这一点

**where 元素只会在至少有一个子元素的条件返回 SQL 子句的情况下才去插入“WHERE”子句。而且，若语句的开头为“AND”或“OR”，where 元素也会将它们去除。**

因此，上例可以改造成：

<select id="searchStudent" parameterType="String" resultType="Student">

SELECT \* FROM Student WHERE

<where>  
 <if test="name != null and name != ''">  
 AND name = #{name}  
 </if>

<if test="sex != null and sex != ''">  
 AND sex = #{sex}  
 </if>

</where>

### 2.2.7 foreach

<insert id="insertStudent" parameterType="java.util.List">

INSERT INTO `student` (`num`, `name`, `age`, `sex`, `classNum`)  
 <foreach collection="list" item="item" separator="," open="VALUES " open="">

(#{item.num}, #{item.name},#{item.age}, #{item.sex}, #{item.classNum})

</foreach>  
 <selectKey keyProperty="id" order="AFTER" resultType="java.lang.Integer">  
 SELECT LAST\_INSERT\_ID()  
 </selectKey>  
</insert>

foreach标签一共有如下几个属性：

1.collection：传入参数类型，有三种情况

1.如果**parameterType**传入参数为List<?>，**collection="list"**

2.如果**parameterType**传入参数为数组，**collection="array"**

3.如果**parameterType**传入参数为map，这意味着除了foreach需要传参，其他部位也需要参数，为了避免混淆，需要使用map键值对传递参数，此时，**collection=**要获取的参数(必需是一个集合或数组)在map中的key值

2. item：集合中每一个元素进行foreach迭代时的别名

3. index：用于表示在迭代过程中，每次迭代到的位置索引

4.separator：表示在每次进行迭代之间以什么符号作为分隔符

5.open：指定foreach循环结束后，在循环结果最左边要拼接的sql串

6.open：指定foreach循环结束后，在循环结果最右边要拼接的sql串

## 2.3 build path

在build path下的类可以被编译运行，在build path下的配置文件可以在类中以相对路径的形式直接读写。

因此我们需要将config文件夹设为Source root文件夹在IDEA中，右键config文件夹，make directory as->source root即可

## java代码

public class MyJunitTestCase {  
 @Test  
 public void findStudentById(){  
 SqlSession sqlSession=null;  
 try {  
 String resource = "mybatisConfig.xml";  
 InputStream in = Resources.*getResourceAsStream*(resource);  
 SqlSessionFactory ssf = new SqlSessionFactoryBuilder().build(in);  
 sqlSession = ssf.openSession();  
 Student student = sqlSession.selectOne("mapper.findStudentById", 2);  
 System.*out*.println(student);  
 }catch (IOException ex){  
 ex.printStackTrace();  
 }finally {  
 if(sqlSession!=null){  
 sqlSession.close();  
 }  
 }  
 }  
}

以上是一个简单地使用mybatis访问数据库的案例

### 2.4.1 SqlSessionFactoryBuilder

这个类可以被实例化、使用和丢弃，最好将其包装到一个工具类(util)中来使用，一旦创建了 SqlSessionFactory，就不再需要它了。因此 SqlSessionFactoryBuilder 实例的最佳范围是**方法范围（**也就是局部方法变量）。我们最好**不要让其一直存在，**以保证所有的 XML 解析资源开放给更重要的事情。

*/\*\*  
 \*通过MyBatis操作数据库用到的是一个叫SqlSession的类，   
 \*  
 \*这个类是通过SqlSessionFactory产生的，一般建议在全局维护一个SqlSessionFactory就可以了。   
 \*/*public class MyBatisUtil {  
  
 private final static SqlSessionFactory *sqlSessionFactory*;  
 //静态代码块在整个web程序执行前，jvm加载类时执行，执行顺序最为靠前  
 static{  
 String resource = "mybatis-config.xml";  
 Reader reader = null;  
 try {  
 reader = Resources.getResourceAsReader(resource);  
 } catch (IOException e) {  
 System.*out*.println(e.getMessage());  
 }  
 *sqlSessionFactory* = new SqlSessionFactoryBuilder().build(reader);  
 }  
  
 public static SqlSessionFactory getSqlsessionfactory() {  
 return *sqlSessionFactory*;  
 }  
}

### 2.4.2 SqlSessionFactory

SqlSessionFactory是工厂类，用于创建SqlSession，应当是**web应用范围**的。因此使用**单例模式**对其进行管理(只创建一次，以后一直使用，与Spring整合后通过Spring的bean的singleton实现单例)

### 2.4.3 SqlSession

每个线程都应该有它自己的 SqlSession 实例。SqlSession 的实例不是线程安全的，因此是不能被共享的，所以它的最佳的范围是**请求或方法范围，**其生命周期最长不应该大于一个HttpRequest请求的生命周期。最后使用结束后不要忘记关闭它释放资源。绝对不能将 SqlSession 实例的引用放在一个类的静态域，甚至一个类的实例变量也不行。最常见的使用是每次收到的 HTTP 请求，就可以打开一个 SqlSession，每次返回一个响应，就关闭它。这个关闭操作是很重要的，应该把这个关闭操作放到 finally 块中以确保每次都能执行关闭。

**1.selectOne(String s)方法**

该方法用于查询一条记录

**2.selectList(String s)方法**

该方法用于查询多条记录

**3.insert(String s)方法**

该方法用于执行插入语句，返回插入记录条数

**4.update(String s)方法**

该方法用于执行更新语句

**5.delete(String s)方法**

该方法用于执行删除语句

## Dao

DAO是数据访问接口，数据访问：顾名思义就是操作数据库的接口。夹在业务逻辑层(service)与数据库资源中间(DB)，DAO层可以将数据访问逻辑和service层的业务逻辑分开，让程序员能够更加专注于编写数据访问代码，DAO层的主要作用就是将访问数据库获取数据，并将其交付service层进行业务逻辑处理

也就是说，在service层不要出现操作数据库的代码，如selectOne，selectList，insert等，而是将这些操作封装成DAO层方法进行调用

### 2.5.1 原始Dao开发

原始的Dao开发，需要编写Dao接口，以及DAO实现类

eg：

**student.xml文件**

<!-- mapper 为根元素节点， 一个namespace对应一个dao -->  
<mapper namespace="mapper">  
 <select id="findStudentById" parameterType="int" resultType="Student">  
 SELECT \* FROM Student WHERE id=#{id}  
 </select>  
 <insert id="insertStudent" parameterType="java.util.List">  
 INSERT INTO `student` (`num`, `name`, `age`, `sex`, `classNum`) VALUES  
 <foreach collection="list" item="item" separator=",">

(#{item.num}, #{item.name},#{item.age}, #{item.sex}, #{item.classNum})

</foreach>  
 <selectKey keyProperty="id" order="AFTER" resultType="java.lang.Integer">  
 SELECT LAST\_INSERT\_ID()  
 </selectKey>  
 </insert>  
 <delete id="deleteStudent" parameterType="int">  
 DELETE FROM `student` WHERE id=#{id}  
 </delete>  
</mapper>

**StudentDao.java DAO接口**

package com.dao;  
import com.orm.pojo.Student;  
import java.util.List;  
  
public interface StudentDao {  
 public Student getStudentById(int id) throws Exception;  
 public int createBatchStudent(List<Student> student) throws Exception;  
 public int deleteStudentById(int id) throws Exception;  
}

**StudentDaoImp.java DAO实现类**

package com.dao;  
import com.orm.pojo.Student;  
import org.apache.ibatis.session.SqlSession;  
import org.apache.ibatis.session.SqlSessionFactory;  
import java.util.List;  
  
public class StudentDaoImp implements StudentDao {  
 private SqlSessionFactory sqlSessionFactory;  
  
 //和spring整合后可通过注解注入SqlSessionFactory  
 public StudentDaoImp(SqlSessionFactory sqlSessionFactory) {  
 this.sqlSessionFactory = sqlSessionFactory;  
 }  
  
 public Student getStudentById(int id) throws Exception {  
 SqlSession sqlSession = this.sqlSessionFactory.openSession();  
 Student student = sqlSession.selectOne("mapper.findStudentById", id);  
 sqlSession.close();  
 return student;  
 }  
  
 public int createBatchStudent(List<Student> student) throws Exception {  
 SqlSession sqlSession = this.sqlSessionFactory.openSession();  
 int a=sqlSession.insert("mapper.insertStudent", student);  
 System.*out*.println("insert返回值："+a);  
 sqlSession.commit();  
 sqlSession.close();  
 return a;  
 }  
  
 public int deleteStudentById(int id) throws Exception {  
 SqlSession sqlSession = this.sqlSessionFactory.openSession();  
 int a=sqlSession.delete("mapper.deleteStudent", id);  
 System.*out*.println("delete返回值："+a);  
 sqlSession.commit();  
 sqlSession.close();  
 return a;  
 }  
}

然而原始DAO编写方式，存在一些问题：

1.DAO接口实现类的方法中，存在大量的重复代码，可以将这些重复代码提取

2.在调用sqlSession的方法时，将sql语句的id硬编码在代码中eg：**mapper.deleteStudent**

3.在调用sqlSession的方法时，由于方法第二个形参使用泛型，即使sql语句的参数类型传入错误，编译器也不会报错

### 2.5.2 mapper代理

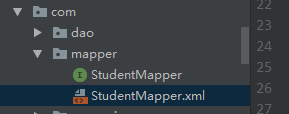
通过mapper代理进行开发时，程序员只需要编写mapper接口即可，mybatis会自动帮我们mapper接口的实现类作为代理对象。因此，在编写mapper接口时，我们不能像原始DAO开发中编写DAO接口一样随意，而是应当按如下规范进行：

**1.mapper.xml(sql语句映射文件)的namespace属性应当与mapper接口的包地址保持一致**

**2.mapper接口中的抽象方法名必须与mapper.xml中sql语句的id名保持一致**

**3.mapper接口中的抽象方法的参数必须与mapper.xml中sql语句的parameterType类型一致**

**4. mapper接口中的抽象方法的反回值必须与mapper.xml中sql语句的resultType类型一致，没有返回值，填入void**



**一般将mapper接口和mapper.xml放在同一目录下，但请注意，由于IDEA不对src下的xml文件进行编译，需要在pom.xml下添加：**

<build>  
 <finalName>mySpringMVC</finalName>  
 <!--注意,如果将静态资源放在src/main/java中,那么编译时将被maven忽略,在target目录下将没有这些资源,此处,我们将mybatis的mapper.xml放在了com.mapper包中,方便编程查看,所以需要将该目录添加为资源目录. -->  
 <resources>  
 <resource>  
 <directory>src/main/java</directory>  
 <includes>  
 <include>\*\*/\*.xml</include>  
 </includes>  
 </resource>  
 </resources>  
</build>

**StudentMapper.xml**

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper PUBLIC "-//ibatis.apache.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
  
<!-- mapper 为根元素节点， 一个namespace对应一个dao -->  
<mapper namespace="com.mapper.StudentMapper">  
 <select id="findStudentById" parameterType="int" resultType="Student">  
 SELECT \* FROM Student WHERE id=#{id}  
 </select>  
 <insert id="insertStudent" parameterType="java.util.List">  
 INSERT INTO `student` (`num`, `name`, `age`, `sex`, `classNum`)VALUES  
 <foreach collection="list" item="item" separator=",">

(#{item.num}, #{item.name},#{item.age}, #{item.sex}, #{item.classNum})

</foreach>  
 <selectKey keyProperty="id" order="AFTER" resultType="java.lang.Integer">  
 SELECT LAST\_INSERT\_ID()  
 </selectKey>  
 </insert>  
 <delete id="deleteStudent" parameterType="int">  
 DELETE FROM `student` WHERE id=#{id}  
 </delete>  
</mapper>

**StudentMapper.java**

package com.mapper;  
import com.orm.pojo.Student;  
import java.util.List;  
  
public interface StudentMapper {  
 public Student findStudentById(int id)throws Exception;  
 public void insertStudent(List<Student> students)throws Exception;  
 public void deleteStudent(int id)throws Exception;  
}

public class myJunit {  
 private SqlSessionFactory sqlSessionFactory;  
 @Before  
 public void setUp(){  
 String config="mybatisConfig.xml";  
 try {  
 InputStream resourceIo=Resources.*getResourceAsStream*(config);  
 this.sqlSessionFactory=new SqlSessionFactoryBuilder().build(resourceIo);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 @Test  
 public void findStudentById() throws Exception {

SqlSession sqlSession=this.sqlSessionFactory.openSession();

StudentMapper studentMapper=sqlSession.getMapper(StudentMapper.class);

Student student=studentMapper.findStudentById("王三","女");

sqlSession.close();

}

**执行完毕别忘记SqlSession的commit和close**

### 2.5.3 传入多个参数

上述方式有一个缺陷，就是只能给sql语句传入一个参数

#### 2.5.3.1 相同类型的参数

如果参数的类型相同，可以通过arg0，arg1…来传入参数

<select id="findStudentById" parameterType="java.lang.String" resultType="Student">  
 SELECT \* FROM Student WHERE name=#{arg0} AND sex=#{arg1}  
</select>

public Student findStudentById(String name,String sex)throws Exception;

上例中，参数类型都是String，因此可以通过**#{arg0}**和**#{arg1}**来传入参数。

当然，如果参数类型不相同，则不能使用该方式

#### 2.5.3.2 封装成pojo对象

由于给sql语句只能传入一个参数，我们可以将多个参数封装成一个pojo进行传递，该方法最常见与新增记录

<insert id="insertStudent" parameterType="java.util.List">  
 INSERT INTO `student` (`num`, `name`, `age`, `sex`, `classNum`) VALUES  
 <foreach collection="students" item="item" separator=",">

(#{item.num}, #{item.name},#{item.age}, #{item.sex}, #{item.classNum})

</foreach>  
 <selectKey keyProperty="id" order="AFTER" resultType="java.lang.Integer">  
 SELECT LAST\_INSERT\_ID()  
 </selectKey>  
</insert>

public void insertStudent(List<Student> students)throws Exception;

上例就是封装成pojo对象的list集合

#### 2.5.3.3 封装成map

<select id="findStudentById" parameterType="java.util.Map" resultType="Student">

SELECT \* FROM Student WHERE name=#{name} AND num=#{num}  
</select>

public Student findStudentById(Map map)throws Exception;

public class myJunit {

private SqlSessionFactory sqlSessionFactory;  
 @Before  
 public void setUp(){  
 String config="mybatisConfig.xml";  
 try {  
 InputStream resourceIo=Resources.*getResourceAsStream*(config);  
 this.sqlSessionFactory=new SqlSessionFactoryBuilder().build(resourceIo);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 @Test  
 public void testfindStudentById() throws Exception {  
 SqlSession sqlSession=this.sqlSessionFactory.openSession();  
 StudentMapper studentMapper=sqlSession.getMapper(StudentMapper.class);  
 HashMap<String,Object> paramMap=new HashMap<String,Object>();  
 paramMap.put("name","王三");  
 paramMap.put("num",123456);  
 Student student=studentMapper.findStudentById(paramMap);  
 System.*out*.println("结果："+student);  
 sqlSession.close();

}

}

通过map虽然能便捷地将参数封装传递，但是缺点也很明显，不够直观，**findStudentById**方法的形参只提示map，无法通过其看出map中具体需要哪些参数

#### 2.5.3.4 通过注解传参(推荐)

上述几种传参方式多多少少有不足之处，下面将介绍一种传参方式：通过注解传递参数

mybatis提供了**@param**注解，该注解的作用是给参数命名，参数命名后就能根据DAO接口的抽象方法中形参的名字映射到sql语句的对应参数中

<select id="findStudentById" resultType="Student">  
 SELECT \* FROM Student WHERE name=#{myName} AND num=#{myNum}  
</select>

public Student findStudentById(@Param("myName")String name,@Param("myNum")int num)throws Exception;

**请注意：使用这种方式，sql语句一般省去parameterType属性**

### 2.5.4 设置返回值

某些情况下，我们也想让mybatis返回适当格式的数据类型

#### 2.5.3.1 返回值为map

Dao方法返回值为map有两种情况，一种是从数据库查出pojo后，任意选取pojo中的一个字段作为key，整个pojo作为value。另一种是返回值为map(查询一条数据)或List<map>(多条数据)的情况，这种情况下是以字段名作为key，字段值作为value的，一个map一条数据

##### 2.5.3.1 pojo中的任意字段作为key

只需要借助一个注解@MapKey即可实现

<select id="getServiceProviderAndEnterPriseRelationByEnterpriseIdList" parameterType="String"

resultMap="BaseResultMap">

select enterprise\_id,service\_provider\_id from r\_energy\_service\_provider\_enterprise  
 where sys\_deleted=0 and enterprise\_id in  
 <trim prefix="(" suffix=")">  
 <foreach collection="enterpriseIdList" item="id" separator=",">  
 #{id}  
 </foreach>  
 </trim>  
</select>

**resultMap正常填就好，只当作是为了查出ServiceProviderEnterprise这个pojo类而不是为了查出Map**

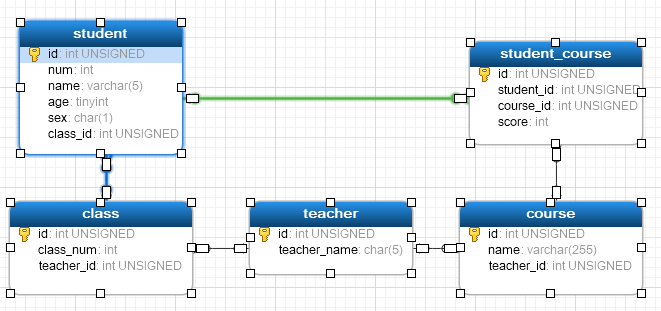
@MapKey("enterpriseId")

Map<String,ServiceProviderEnterprise> getServiceProviderAndEnterPriseRelationByEnterpriseIdList(@Param("enterpriseIdList")List<String> enterpriseIdList);

**@MapKey中填入pojo的一个属性，该属性将作为map的key**

## 关联查询

下面就用一个实例来演示关联查询。现有表：



student 学生表

student\_course 学生选课表

class 班级表

teacher 教师表

course 课程表

### 2.6.1 po对象创建

首先我们需要几个po类来承载数据，最开始应当是一个表创建一个po类，一个字段对应一个属性，创建student、student\_course、class、teacher、course五个类

这里不在一一举例，只列出student.java作为示例：

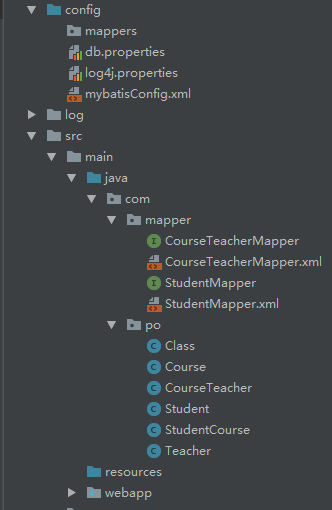
package com.po;  
  
public class Teacher {  
 private int id;  
 private String teacher\_name;  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getTeacher\_name() {  
 return teacher\_name;  
 }  
  
 public void setTeacher\_name(String teacher\_name) {  
 this.teacher\_name = teacher\_name;  
 }  
}

但是光靠这几个po对象是不够的，如果连表查询，查询结果就是两张(或以上)表的字段，则需要对这几个po对象进行扩展，扩展原则是尽量继承字段多的po对象

### 2.6.2 一对一查询

需求：给定课程名，根据课程名查询任课老师的信息及课程信息

任课老师信息和课程信息，两张表的内容，现有po对象不满足需求，因此需要扩展现有po对象（尽量继承字段多的po对象）：



**CourseTeacher.java**

package com.po;  
  
public class CourseTeacher extends Course{  
 private String teacher\_name;  
  
 public String getTeacher\_name() {  
 return teacher\_name;  
 }  
  
 public void setTeacher\_name(String teacher\_name) {  
 this.teacher\_name = teacher\_name;  
 }  
  
 @Override  
 public String toString() {  
 return new StringBuilder("课程编号：").append(this.getId())  
 .append(",课程名：").append(this.getName())  
 .append(",教师编号：").append(this.getTeacher\_id())  
 .append(",教师名：").append(this.teacher\_name).toString();  
 }  
}

在mybatis配置文件中，配置po别名，加载mapper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>  
 <!-- 加载java的配置文件或者声明属性信息 -->  
 <properties resource="db.properties"/>  
 <!-- 自定义别名 -->  
 <typeAliases>  
 <package name="com.po"/>  
 </typeAliases>  
  
 <!-- 配置mybatis的环境信息，与spring整合，该信息由spring来管理 -->  
 <environments default="development">  
 <environment id="development">  
 <!-- 配置JDBC事务控制，由mybatis进行管理 -->  
 <transactionManager type="JDBC"/>  
 <!-- 配置数据源，采用mybatis连接池 -->  
 <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>  
  
 <!-- 加载映射文件 -->  
 <mappers>  
 <!-- 批量加载映射文件 -->  
 <package name="com.mapper"/>  
 </mappers>  
</configuration>

编写mapper文件CourseTeacherMapper.xml

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper  
 PUBLIC "-//ibatis.apache.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper namespace="com.mapper.CourseTeacherMapper">  
 <select id="findCourseInfo" parameterType="String" resultType="CourseTeacher">  
 SELECT course.\*,teacher.teacher\_name FROM course LEFT JOIN teacher on teacher.id=teacher\_id  
 <where>  
 course.name=#{name}  
 </where>  
 </select>  
</mapper>

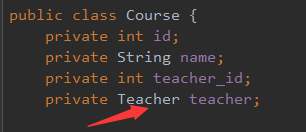
编写mapper接口CourseTeacherMapper.java，使用mapper代理开发，需与CourseTeacherMapper.xml放在同一目录下

package com.mapper;  
  
import com.po.CourseTeacher;  
  
public interface CourseTeacherMapper {  
 public CourseTeacher findCourseInfo(String name) throws Exception;  
}

测试：

public class myJunit {  
 private SqlSessionFactory sqlSessionFactory;  
 @Before  
 public void setUp(){  
 String config="mybatisConfig.xml";  
 try {  
 InputStream resourceIo=Resources.*getResourceAsStream*(config);  
 this.sqlSessionFactory=new SqlSessionFactoryBuilder().build(resourceIo);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 @Test  
 public void test() throws Exception {  
 SqlSession sqlSession=this.sqlSessionFactory.openSession();  
 CourseTeacherMapper courseTeacherMapper=sqlSession.getMapper(CourseTeacherMapper.class);  
 CourseTeacher courseTeacher=courseTeacherMapper.findCourseInfo("大学物理");  
 System.*out*.println("结果："+courseTeacher);  
 }  
}

如果想要使用resultMap，在course中添加一个teacher属性(**不使用CourseTeacher类承载查询结果，转而使用course类**)



resultMap有子节点association来帮助我们进行一对一查询

则CourseTeacherMapper.xml为：

<resultMap id="courseTeacherResultMap" type="Course">

<id column="id" property="id"></id>  
 <result column="name" property="name"></result>  
 <result column="teacher\_id" property="teacher\_id"></result>  
 <association property="teacher" javaType="Teacher">  
 <id column="teacherId" property="id"></id>  
 <result column="teacher\_name" property="teacher\_name"></result>  
 </association>  
</resultMap>  
<select id="findCourseInfoByResultMap" parameterType="String" resultMap="courseTeacherResultMap">  
 SELECT course.\*,teacher.id as teacherId,teacher.teacher\_name FROM course LEFT JOIN teacher on teacher.id=teacher\_id  
 <where>  
 course.name=#{name}  
 </where>  
</select>

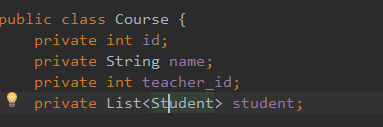
### 2.6.2 一对多查询

**参考下一节一对多查询**

### 2.6.3 多对多查询

需求：给定课程名，根据课程名查询课程信息以及选择该课程的学生信息

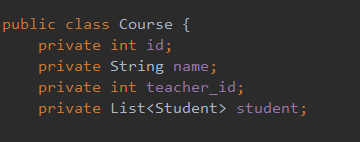
在上例中，会查出多条记录，其中多条记录的课程信息会重复，如果希望课程信息不重复，我们可以这样设计po类



然后通过resultMap的collection子节点配置student属性的映射，其他部分和一对一查询并无区别

<resultMap id="courseStudent" type="Course">

<id column="course\_id" property="id"></id>  
 <result column="course\_name" property="name"></result>  
 <result column="teacher\_id" property="teacher\_id"></result>  
 <collection property="student" javaType="List" ofType="Student">  
 <id column="student\_id" property="id"></id>  
 <result column="num" property="num"></result>  
 <result column="age" property="age"></result>  
 <result column="student\_name" property="name"></result>  
 <result column="sex" property="sex"></result>  
 </collection>  
</resultMap>  
<select id="findStudentsSelectedCourse" parameterType="String" resultMap="courseStudent">  
 SELECT course.id as course\_id,student.id as student\_id,student.num,student.name as student\_name,student.age,student.sex,student.class\_id,course.name as course\_name,course.teacher\_id  
 FROM course LEFT JOIN student\_course ON student\_course.course\_id=course.id  
 LEFT JOIN student ON student\_course.student\_id=student.id  
 <where>  
 course.name=#{courseName}  
 </where>  
</select>



这就要求mapper文件中的sql语句必须使用resultMap进行代理开发

## 延迟加载

### 2.5.1 什么是延迟加载

有些时候，我们值用到一张表中的信息，此时我们做单表查询，当我们用到关联表的属性时，再去查询对应关联表的信息。这种对用户需求的按需查询就叫延时加载

resultMap的association和collection标签就支持延迟加载功能

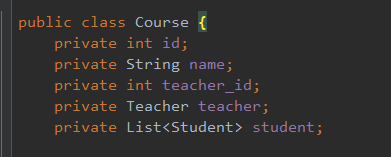
我们需要在mybatisConfig文件中开启懒加载

<settings><!-- -->  
 <!-- 懒加载设置为 true -->  
 <setting name="lazyLoadingEnabled" value="true"/>  
 <!-- 积极加载设置为false -->  
 <setting name="aggressiveLazyLoading" value="false"/>

<setting name="lazyLoadTriggerMethods" value=""/>  
</settings>

### 2.5.1 配置延迟加载

在association和collection标签中有两个属性select和column，可以帮助我们配置延迟加载，select属性指定了要二次查询的select标签，column指定了第一次查询出的字段中要传递给第二次查询作为参数的字段



CourseTeacherMapper.xml

<resultMap id="findCourseDeatil" type="Course">  
 <id property="id" column="id"></id>  
 <result property="name" column="name"></result>  
 <result property="teacher\_id" column="teacher\_id"></result>  
 <association property="teacher" javaType="Teacher" select="com.mapper.TeacherMapper.findTeacherById" column="teacher\_id">

<id property="id" column="id"></id>  
 <result property="teacher\_name" column="teacher\_name"></result>  
 </association>  
</resultMap>  
<select id="findCourseById" parameterType="int" resultMap="findCourseDeatil">  
 select \* FROM course WHERE id=#{id}  
</select>

StudentMapper.xml

<mapper namespace="com.mapper.TeacherMapper">  
 <select id="findTeacherById" parameterType="int" resultType="Teacher">  
 SELECT\* FROM teacher WHERE id=#{id}  
 </select>  
</mapper>

由于二者不在同一个文件。因此select属性中加上了namespace (**com.mapper.TeacherMapper.findTeacherById**)，将第一次**select \* FROM course WHERE id=#{id}**查询到**teacher\_id**传入第二次查询，如果最终的Course对象用到了teacher属性，则进行第二次查询，否则只进行第一次的单表查询，提高了sql性能

如果希望给第二次查询传入多个参数，应将二次查询的**parameterType**设为Map属性，然后通过**column="{key1=value1,key2=value2}"**的方式传入参数，eg：

<association property="actualWorkTime" column="{reqId=id,endDate=endDateStr,startDate=startDateStr}" select="selectById"></association>

<select id="selectById" parameterType="java.util.Map" resultType="java.lang.Double">  
 select \* from Table\_XXX  
 where reqId=#{reqId}  
 and DATE\_FORMAT(endDate,'%Y-%m-%d') <= DATE\_FORMAT(#{endDate},'%Y-%m-%d')  
 and DATE\_FORMAT(startDate,'%Y-%m-%d') >= DATE\_FORMAT(#{startDate},'%Y-%m-%d')  
</select>

对collection同理：

<collection property="student" javaType="List" ofType="Student" select="com.mapper.StudentMapper.findStudentByCourse" column="id"></collection>

## 查询缓存

### 2.5.1简介

mybatis提供了查询缓存来缓解数据库压力。

在操作数据库时，需要构建sqlSession对象，在对象中有一个HashMap用于存储缓存数据。因此，不同的sqlSession之间的缓存互不影响，这就是一级缓存，它是sqlSession级别的缓存。

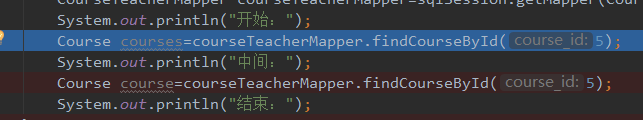
二级缓存是多个sqlSession共享的，是mapper级别的，如果有多个sqlSession去操作同一个sql语句，则取出的数据是存在二级缓存中的

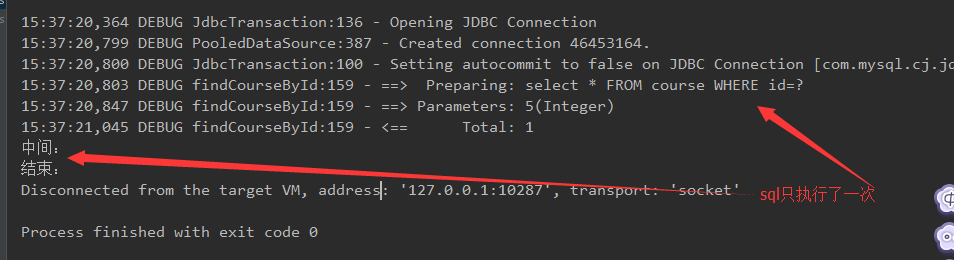


### 2.5.2一级缓存

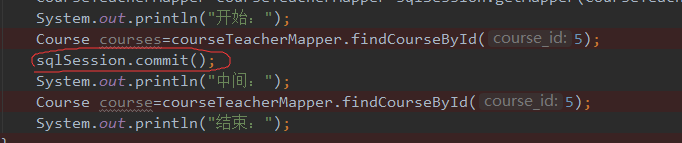
sqlSession会将用户查询结果缓存到HashMap中，当下次查询的时候，如果判断先前有个完全一样的查询，会直接从缓存中直接将结果取出，返回给用户，不需要再进行一次数据库查询了。如果sqlSession执行了commit操作(更新、插入、删除)，会清空sqlSession的一级缓存避免读到脏数据

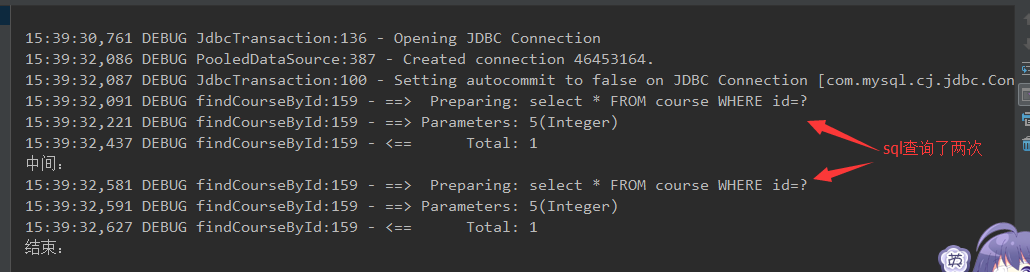
**mybatis默认开启一级缓存**





如果中间有commit操作，就会清空缓存





**对于某个sqlSession执行update、insert或delete操作，会清空缓存**

在整合spring之后：

* 1. **如果没有开启事务，一级缓存失效**，这是因为不开启事务的话，每次查询都会新建一个sqlSession，整个方法结束后，sqlSession将会自动关闭
  2. **如果开启了事务，一级缓存仍然生效**，因为开启事务后，将会自动获取当前线程上绑定的sqlSession，前后使用的仍是同一个sqlSession，并且方法结束后，也不会关闭sqlSeesion(因为每个线程都会对应绑定一个sqlSession)

一级缓存实际上没有什么存在的实际意义，对于多个sqlsession共存或者分布式环境下，一级缓存极有可能读出脏数据，单独使用时建议关闭，和spring整合后就没必要关闭了，这是因为整合spring后若开启，则事务一级缓存生效，出现的问题是读取脏数据，即一个sqlSeesion的两次读取之间，数据被另一个sqlSession修改，这是不可重复的现象，这一点由mysql的隔离级别处理了(mysql默认是可重复读级别，即保证两次查询的数据一致)，不管是否主动关闭缓存都不会查询最新的更改后的数据，因此没必要主动关闭，不整合的话尽量还是关闭一下

### 2.5.3二级缓存

由于sqlSession在service方法执行结束后一般会使用close方法关闭(sqlSession是方法级别的)，一旦sqlSession被关闭，其数据结构就被销毁，因此一般情况下，不同的service层方法之间的一级缓存不共享(因为它们是不同的sqlSession)，但在开发中，我们往往有在不同sqlSession中共享缓存的需求，这就用到了二级缓存

如果开启了二级缓存，不同的sqlSession在执行相同的sql语句所查询到的数据都会存储在二级缓存中。同一个namespqce下的mapper中的sql语句共享同一个二级缓存区域

**二级缓存默认不开启，除了需要在mybatisConfig文件中开启总开关，还需要在mapper文件中进行配置**

**mybatisConfig.xml**

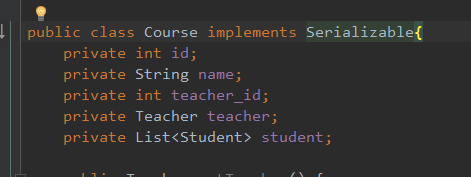
<setting name="cacheEnabled" value="true"/>

然后还需要在对应的mapper文件中开启二级缓存

**CourseTeacherMapper.xml**

<mapper namespace="com.mapper.CourseTeacherMapper">  
 <!--开启本namespqce下的二级缓存-->  
 <cache />

由于二级缓存的存储介质不一定是在内存中，也可能是在硬盘中，我们取出缓存时可能需要对其进行反序列化。因此，使用二级缓存的sql语句的结果类型，其pojo对象应实现序列化接口Serializable



@Test  
public void test() throws Exception {  
 SqlSession sqlSession\_1=this.sqlSessionFactory.openSession();  
 SqlSession sqlSession\_2=this.sqlSessionFactory.openSession();  
 CourseTeacherMapper courseTeacherMapper\_1=sqlSession\_1.getMapper(CourseTeacherMapper.class);  
 CourseTeacherMapper courseTeacherMapper\_2=sqlSession\_2.getMapper(CourseTeacherMapper.class);  
 System.*out*.println("sqlSession\_1：");  
 Course courses=courseTeacherMapper\_1.findCourseById(5);  
 //close操作会将sqlSession的数据写入到二级缓存区域  
 sqlSession\_1.close();  
 System.*out*.println("sqlSession\_2：");  
 Course course=courseTeacherMapper\_2.findCourseById(5);  
 sqlSession\_2.close();  
 System.*out*.println("结束：");  
}

**如果sqlSession\_1不执行close关闭操作，则不会将数据写入二级缓存，第二次查询还是去数据库查询！同样的，如果有commit操作，为了避免读取到脏数据，会重新进行查询**

**对于某个mapper执行update、insert或delete操作，会清空缓存**

也可以给某条sql语句禁用二级缓存，只要设置**useCache="false"**即可，**对于联表查询的sql，务必禁用二级缓存，二级缓存请在单表查询中使用**，这是因为由于MyBatis的二级缓存是基于mapper文件的，多表查询语句所在的mapper无法感应到其他mapper中的语句对多表查询中涉及的表进行的修改，进而引发脏数据问题。eg：

<select id="findCourseById" parameterType="int" resultMap="findCourseDeatil" **useCache="false"**>  
 select \* FROM course WHERE id=#{id}  
</select>

### 2.5.4二级缓存的局限性和刷新缓存

对查询频率高，变化频率低的数据建议使用二级缓存。

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。

对于sqlSession或者mapper执行insert、update、delete等操作可以清空一级缓存或二级缓存，但如果将flushCache设为false(默认为true)，则不会清空缓存，一般情况下不需要设置该属性

既然可以刷新缓存，那么可以定时进行刷新吗？答案是肯定的，我们在mapper映射文件中添加<cache/>来表示给该namespace下的mapper文件开启缓存，那接下来，只需要我们配置刷新间隔时间flushinterval，通过设置flushinterval，让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。

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

**flushInterval** 代表刷新间隔时间，单位是毫秒

**size** 缓存会存储列表集合或对象(无论查询方法返回什么)的引用个数，默认1024

**readOnly** 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。

**eviction** 设置缓存淘汰算法

**type** 设置缓存实现类，实现了cache接口的实例，默认为PerpetualCache，这是mybatis自己的缓存机制，也可以配置第三方框架的缓存，前提是实现cache类

二级缓存也有相当的局限性，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，假设此时二级缓存中存有一万个商品的缓存数据，一旦一个商品数据改变，那这一万个商品的缓存数据都将全部清空，也就是说如果使用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。

**Mybatis本身的缓存，无论是一级缓存还是二级缓存，对于分布式应用都是没有意义的，因为他们的缓存只能缓存在一台服务器的内存中，在分布式环境中极易出现脏读。在缓存技术已相当成熟的当下，没必要使用mybatis自带的缓存**

## 逆向工程

mybatis提供了逆向工程帮助我们生成代码，可以针对数据库的单表自动生成mybatis执行所需要的代码（包括mapper.xml、mapper.java接口、po类..），如果希望使用逆向工程来生成代码，需要配置逆向工程的jar包，其maven如下：

<dependency>  
 <groupId>org.mybatis.generator</groupId>  
 <artifactId>mybatis-generator-core</artifactId>  
 <version>${generatorVersion}</version>  
</dependency>

通过xml配置逆向工程

# Mybatis整合其他框架

## 3.1 Mybatis 整合spring

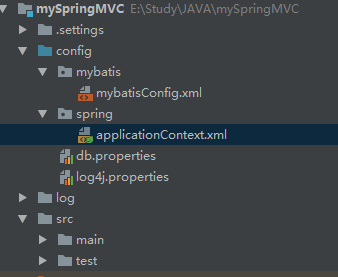
### 3.1.1 思路和环境搭建

SqlSessionFactory是工厂类，用于创建SqlSession，应当是**web应用范围**的。只创建一次，以后一直使用，因此我们需要通过**单例模式**进行管理，spring帮助我们实现了这一点

spring整合mybatis需要第三方jar包，其maven代码为：

<dependency>

<groupId>org.mybatis</groupId>  
 <artifactId>mybatis-spring</artifactId>  
 <version>1.3.1</version>  
</dependency>



### 3.1.2 spring中配置sqlSessionFactory

我们需要在spring配置文件中获取sqlSessionFactory的bean实例，这里我们从**mybatis-spring.jar**中获取对应的类**org.mybatis.spring.SqlSessionFactoryBean**

由于spring可以帮助我们创建DataSource的bean对象，我们不再需要DataSource工厂类，因此这里配置druid就不需要像2.1.5.3中一样包装一下了，直接使用类**com.alibaba.druid.pool.DruidDataSource**即可

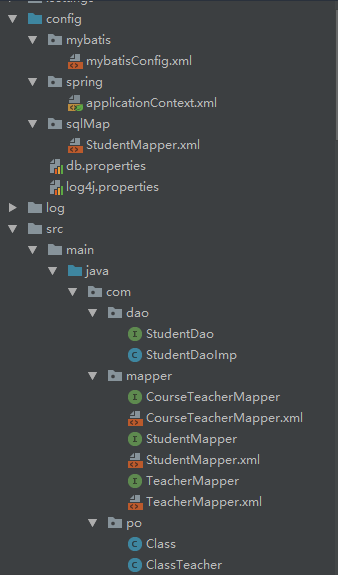
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/context  
 http://www.springframework.org/schema/context/spring-context.xsd">  
 <!--扫描加载classpath路径下的所有properties文件,classpath路径就是target/classes文件夹下的路径-->  
 <context:property-placeholder location="classpath:\*.properties" />  
 <!--配置数据源，数据库连接池使用阿里巴巴的druid连接池，指定初始化方法和销毁方法-->  
 <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init"  
 destroy-method="close">  
 <property name="driverClassName" value="${jdbc.driver}"></property>  
 <property name="url" value="${jdbc.url}"></property>  
 <property name="username" value="${jdbc.username}"></property>  
 <property name="password" value="${jdbc.password}"></property>  
 <!--最大连接数-->  
 <property name="maxActive" value="60"></property>  
 <!--最小连接数-->  
 <property name="minIdle" value="10"></property>  
 <property name="testWhileIdle" value="true"></property>  
 </bean>  
 <!--在这个bean中配置sqlSeesionFactory-->  
 <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  
 <!--加载mybatis配置文件-->  
 <!--注入依赖,value填入mybatis配置文件的classpath路径即可-->  
 <property name="configLocation" value="mybatis/mybatisConfig.xml"></property>  
 <!--引用配置好的数据源DataSource的bean-->  
 <property name="dataSource" ref="dataSource"></property>  
 </bean>  
</beans>

对于bean sqlSessionFactory的配置，根据DAO开发方式的不同，有两种配置方式：

#### 3.1.2.1 原始DAO开发

参照2.5.1节的原始DAO开发，我们将其整合到spring中



首先我们来看看mybatisConfig.xml文件的配置：

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

<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-config.dtd">  
<configuration>  
 <settings>  
 <!-- 懒加载设置为 true -->  
 <setting name="lazyLoadingEnabled" value="true"/>  
 <!-- 积极加载设置为false -->  
 <setting name="aggressiveLazyLoading" value="false"/>  
 <setting name="lazyLoadTriggerMethods" value="equals,clone,hashCode"/>  
 <setting name="cacheEnabled" value="true"/>  
 </settings>  
 <!-- 自定义别名 -->  
 <typeAliases>  
 <package name="com.po"/>  
 </typeAliases>  
 <!-- 加载映射文件 -->  
 <mappers>  
 <mapper resource="sqlMap/StudentMapper.xml"/>  
 <!-- 批量加载映射文件 -->  
 <package name="com.mapper"/>  
 </mappers>  
</configuration>

和之前的配置的文件相比，我们发现，少了加载属性文件的**<properties>**标签和配置数据库连接的**<environments>**标签，这些都被集成到了spring配置文件中，由spring进行管理

接下来看看spring的配置文件

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

<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/context  
 http://www.springframework.org/schema/context/spring-context.xsd">  
 <!--扫描加载classpath路径下的所有properties文件,classpath路径就是target/classes文件夹下的路径-->  
 <context:property-placeholder location="classpath\*:\*.properties" />  
 <!--配置数据源，数据库连接池使用阿里巴巴的druid连接池-->  
 <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init"  
 destroy-method="close">  
 <property name="driverClassName" value="${jdbc.driver}"></property>  
 <property name="url" value="${jdbc.url}"></property>  
 <property name="username" value="${jdbc.username}"></property>  
 <property name="password" value="${jdbc.password}"></property>  
 <!--最大连接数-->  
 <property name="maxActive" value="60"></property>  
 <!--最小连接数-->  
 <property name="minIdle" value="10"></property>  
 <property name="testWhileIdle" value="true"></property>  
 </bean>  
 <!--在这个bean中配置sqlSeesionFactory-->  
 <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  
 <!--加载mybatis配置文件-->  
 <!--注入依赖,value填入mybatis配置文件的classpath路径即可-->  
 <property name="configLocation" value="mybatis/mybatisConfig.xml"></property>  
 <!--引用配置好的数据源DataSource的bean-->  
 <property name="dataSource" ref="dataSource"></property>  
 </bean>  
 <!--原始Dao开发-->  
 <bean id="studentDaoImp" class="com.dao.StudentDaoImp">  
 <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>  
 </bean>  
</beans>

从spring配置文件中我们可以看到，首先我们扫描并加载了properties文件，然后我们使用druid连接池配置了数据源的bean，该bean被注入到**org.mybatis.spring.SqlSessionFactoryBean** 对象中作为全局的sqlSessionFactory使用，有了sqlSessionFactory后，一切就好办了，接下来我们编写mapper.xml文件

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

<!DOCTYPE mapper  
 PUBLIC "-//ibatis.apache.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper namespace="originalDao">  
 <select id="findInstructorsByStudentInfo" parameterType="Student" resultType="ClassTeacher">  
 SELECT \* FROM class  
 LEFT JOIN teacher ON class.teacher\_id=teacher.id  
 LEFT JOIN student ON student.class\_id=class.id  
 WHERE student.name=#{name} and sex=#{sex}  
 </select>  
</mapper>

入参是Student类型的po对象，出参是ClassTeacher类型的po对象，namespace是**originalDao**(原始DAO开发不对namespace命名规范做要求)

package com.dao;

import com.po.ClassTeacher;  
import com.po.Student;  
public interface StudentDao {

public ClassTeacher findInstructorsByStudentInfo(Student student) throws Exception;  
}

StudentDao接口如上

package com.dao;

import com.po.ClassTeacher;  
import com.po.Student;  
import org.apache.ibatis.session.SqlSession;  
import org.mybatis.spring.support.SqlSessionDaoSupport;  
  
public class StudentDaoImp extends SqlSessionDaoSupport implements StudentDao{  
 public ClassTeacher findInstructorsByStudentInfo(Student student) throws Exception{  
 SqlSession sqlSession=this.getSqlSession();  
 ClassTeacher classTeacher=sqlSession.selectOne("originalDao.findInstructorsByStudentInfo",student);  
 //这里的sqlSession是一个代理类SqlSessionTemplate，内部他会为每次请求创建线程安全的sqlsession,并与Spring进行集成.在方法调用完毕以后会自动关闭。  
 return classTeacher;  
 }  
}

DAO实现类就需要重点注意了，这里我们除了要实现DAO接口外，还需要继承整合包中的**org.mybatis.spring.support.SqlSessionDaoSupport**类，该类为我们提供了两个方法setSqlSessionFactory(虽然该类没有sqlSessionFactory属性)和getSqlSession，让我们看看上面spring配置文件中是如何配置DAO实现类的bean对象的

<!--原始Dao开发-->

<bean id="studentDaoImp" class="com.dao.StudentDaoImp">  
 <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>  
</bean>

从property标签我们可以看到，这样的配置意味着将会调用setSqlSessionFactory方法给**StudentDaoImp**注入sqlSessionFactory，而**StudentDaoImp**的基类**SqlSessionDaoSupport**中并未持有sqlSessionFactory对象，而是持有sqlSession对象，那sqlSessionFactory方法到底做了什么呢？通过查看源码我们可以得知

public abstract class SqlSessionDaoSupport extends DaoSupport {

private SqlSession sqlSession;  
 private boolean externalSqlSession;  
  
 public SqlSessionDaoSupport() {  
 }  
  
 public void setSqlSessionFactory(SqlSessionFactory sqlSessionFactory) {  
 if (!this.externalSqlSession) {  
 this.sqlSession = new SqlSessionTemplate(sqlSessionFactory);  
 }  
 }

public SqlSession getSqlSession() {  
 return this.sqlSession;  
 }

未赋值的boolean值默认为false，因此第一次将会执行if代码块，用注入的sqlSessionFactory创建sqlSessionTemplate对象，SqlSessionTemplate 是 MyBatis-Spring 整合包的核心，它实现了sqlSession接口，可以被当做sqlSession使用

并且SqlSessionTemplate不需要close关闭sqlSession，这采用了aop编程的技术通过spring的动态代理的方式实现，比如调用SqlSessionTemplate的insert方法进行插入操作实际上是调用sqlSessionProxy代理对象进行的

public int insert(String statement) {

return this.sqlSessionProxy.insert(statement);  
}

而代理对象的调用处理器是SqlSessionInterceptor，该处理器的invoke方法有一个trycatch块，在finally中执行了closeSqlSession方法，因此SqlSessionTemplate的方法执行完毕后会自动关闭sqlSession，不需要我们手动关闭

#### 3.1.2.2 mapper代理开发

参照2.5.2节的原始mapper代理开发，我们将其整合到spring中

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

<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/context  
 http://www.springframework.org/schema/context/spring-context.xsd">  
 <!--扫描加载classpath路径下的所有properties文件,classpath路径就是target/classes文件夹下的路径-->  
 <context:property-placeholder location="classpath\*:\*.properties" />  
 <!--配置数据源，数据库连接池使用阿里巴巴的druid连接池-->  
 <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init"  
 destroy-method="close">  
 <property name="driverClassName" value="${jdbc.driver}"></property>  
 <property name="url" value="${jdbc.url}"></property>  
 <property name="username" value="${jdbc.username}"></property>  
 <property name="password" value="${jdbc.password}"></property>  
 <!--最大连接数-->  
 <property name="maxActive" value="60"></property>  
 <!--最小连接数-->  
 <property name="minIdle" value="10"></property>  
 <property name="testWhileIdle" value="true"></property>  
 </bean>  
 <!--在这个bean中配置sqlSeesionFactory-->  
 <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  
 <!--加载mybatis配置文件-->  
 <!--注入依赖,value填入mybatis配置文件的classpath路径即可-->  
 <property name="configLocation" value="mybatis/mybatisConfig.xml"></property>  
 <!--引用配置好的数据源DataSource的bean-->  
 <property name="dataSource" ref="dataSource"></property>  
 </bean>  
 <!--mapper代理开发-->  
 <bean id="studentMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">  
 <property name="mapperInterface" value="com.mapper.StudentMapper"></property>  
 <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>  
 </bean>  
</beans>

需要给mapper代理bean指定mapper接口和sqlSessionFactory，剩下的如DAO开发中一样调用即可

但这样需要对每个mapper文件都配置一个对应的bean，很麻烦，可以通过mapper批量扫描进行配置，它会默认注册id为mapper文件首字母小写的mapper的bean，如StudentMapper.xml会注册名为studentMapper的bean，同时mybatis配置文件中的扫描mapper的package标签也可以去掉，如果希望扫描多个包，使用逗号分隔

public void test() {

ApplicationContext ac = new ClassPathXmlApplicationContext("spring/applicationContext.xml");  
 StudentMapper studentMapper = (StudentMapper) ac.getBean("studentMapper");  
 try {  
 List<Student> students = studentMapper.findStudentByCourse(1);  
 for (Student student : students) {  
 System.*out*.println(student.getName());  
 }  
  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }

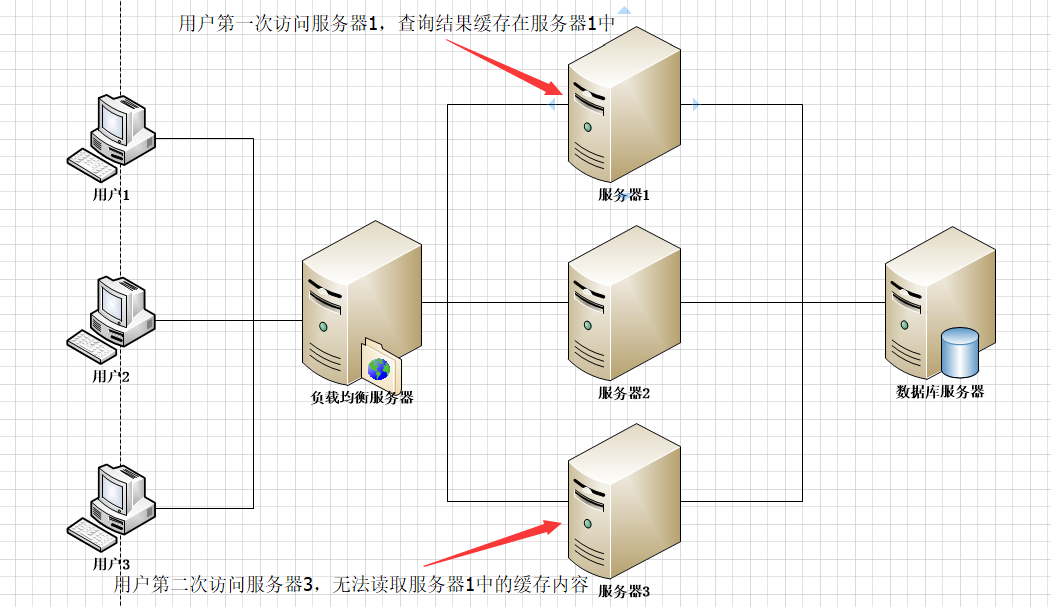
### 3.1.3 SSM整合

参考<http://how2j.cn/k/idea/idea-maven-idea-ssm-create/1397.html>

## 3.2 Mybatis 整合redis

### 3.2.1 环境配置

mybatis的缓存虽然好用，但由于mybaits默认缓存是缓存在服务器内存中的，这在分布式系统上存在极大的问题：



为了解决这一问题，我们将用到缓存技术，这里介绍如何使用redis作为mybatis的二级缓存，关于redis的下载安装配置参见redis学习笔记

如果想要在java中使用redis，应当导入jedis包，其maven代码为：

<dependency>  
 <groupId>redis.clients</groupId>  
 <artifactId>jedis</artifactId>  
 <version>${jedisVersion}</version>  
</dependency>

如果需要与spring整合，则还需要添加对应的jar包

<dependency>  
 <groupId>org.springframework.data</groupId>  
 <artifactId>spring-data-redis</artifactId>  
 <version>${springRedis}</version>  
</dependency>

### 3.2.2 环境配置