MyBatis

一、框架概述

1 软件开发常用结构

1. 三层架构

   三层架构包含的三层：

   - 用户界面层（User Interface layer）　　controller包（XXXServlet类）
   - 业务逻辑层（Business Logic Layer）　　service包（XXXService类）
   - 数据访问层（Data access layer）　　dao包（XXXDao类）

   三层的职责：

   1. 用户界面层（表示层，视图层）：主要功能是接受用户的数据，显示请求的处理结果。使用 web 页面和用户交互，手机 app 也就是表示层的，用户在 app 中操作，业务逻辑在服务器端处理。
   2. 业务逻辑层：接收表示层传递过来的数据，检查数据，计算业务逻辑，调用数据访问层获取数据。
   3. 数据访问层：与数据库打交道。主要实现对数据的增、删、改、查。将存储在数据库中的数据提交给业务层，同时将业务层处理的数据保存到数据库。

   三层的处理请求的交互：

   用户界面层 ---> 业务逻辑层 ---> 数据访问层（持久层） ---> 数据库（MySql）

   三层对应的处理框架

   - 界面层 --- servlet --- springmvc（框架）
   - 业务逻辑层 --- service类 --- spring（框架）
   - 数据访问层 --- dao类 --- mybatis（框架）

   为什么要使用三层?

   1. 结构清晰、耦合度低,、各层分工明确；
   2. 可维护性高，可扩展性高；
   3. 有利于标准化；
   4. 开发人员可以只关注整个结构中的其中某一层的功能实现；
   5. 有利于各层逻辑的复用。

2. 常用框架

   常见的 J2EE 中开发框架：

   - MyBatis 框架：

     MyBatis 是一个优秀的基于 java 的持久层框架，内部封装了 jdbc，开发者只需要关注 sql 语句本身，而不需要处理加载驱动、创建连接、创建 statement、关闭连接，资源等繁杂的过程。

     MyBatis 通过 xml 或注解两种方式将要执行的各种 sql 语句配置起来，并通过 java 对象和 sql 的动态参数进行映射生成最终执行的 sql 语句，最后由 mybatis 框架执行 sql 并将结果映射为 java 对象并返回。

   - Spring 框架：

     Spring 框架为了解决软件开发的复杂性而创建的。Spring 使用的是基本的 JavaBean 来完成以前非常复杂的企业级开发。Spring 解决了业务对象，功能模块之间的耦合，不仅在 javase、web 中使用， 大部分 Java 应用都可以从 Spring 中受益。

     Spring 是一个轻量级控制反转(IoC)和面向切面(AOP)的容器。

   - SpringMVC 框架：

     Spring MVC 属于 SpringFrameWork 3.0 版本加入的一个模块，为 Spring 框架提供了构建 Web 应用程序的能力。现在可以Spring 框架提供的 SpringMVC 模块实现 web 应用开发，在 web 项目中可以无缝使用 Spring 和 Spring MVC 框架。

2 框架是什么

1. 框架定义

   - 框架（Framework）是整个或部分系统的可重用设计，表现为一组抽象构件及构件实例间交互的方法；另一种认为，框架是可被应用开发者定制的应用骨架、模板。
   - 简单的说，框架其实是半成品软件，就是一组组件，供你使用完成你自己的系统。从另一个角度来说框架是一个舞台，你在舞台上做表演。在框架基础上加入你要完成的功能。
   - 框架是安全的，可复用的，不断升级的软件。

2. 框架解决的问题

   框架要解决的最重要的一个问题是技术整合，在 J2EE 的 框架中，有着各种各样的技术，不同的应用，系统使用不同的技术解决问题。需要从 J2EE 中选择不同的技术，而技术自身的复杂性，有导致更大的风险。企业在开发软件项目时，主要目的是解决业务问题。即要求企业负责技术本身，又要求解决业务问题。这是大多数企业不能完成的。框架把相关的技术融合在一起，企业开发可以集中在业务领域方面。

   另一个方面可以提供开发的效率。

3. 框架特点

   - 框架一般不是全能的， 不能做所有事情
   - 框架是针对某一个领域有效。 特长在某一个方面，比如mybatis做数据库操作强，但是他不能做其它的。
   - 框架是一个软件

3 JDBC编程

1. 使用 JDBC 编程的回顾

```
public class JDBCTest05 {
    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        ResultSet resultSet = null;
        try {
            //1、注册驱动
            Class.forName("com.mysql.jdbc.Driver");
            //2、获取连接
            conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/bjpowernode","root","111");
            //3、获取数据库操作对象
            stmt = conn.createStatement();
            //4、执行SQL
            String sql = "select empno as a, ename, sal from emp";
            resultSet = stmt.executeQuery(sql);//专门执行DQL语句的方法
            //5、处理查询结果集
            while(resultSet.next()){
                int empno = resultSet.getInt("a");
                String ename = resultSet.getString("ename");
                double sal = resultSet.getDouble("sal");
                System.out.println(empno + "," + ename + "," + sal);
            }
        }catch (SQLException e){
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } finally {
            //6、释放资源
            if(resultSet != null){
                try {
                    resultSet.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
            if(stmt != null){
                try {
                    stmt.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
            if(conn != null){
                try {
                    conn.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

2. 使用 JDBC 的缺陷

- 代码比较多，开发效率低
- 需要关注 Connection，Statement，ResultSet 对象的创建和销毁
- 对 ResultSet 查询的结果，需要自己封装为 List
- 重复的代码比较多些
- 业务代码和数据库的操作混在一起

4 MyBatis框架概述

MyBatis 框架：

MyBatis 本是 apache 的一个开源项目 iBatis，2010 年这个项目由 apache software foundation 迁移到了 google code，并且改名为 MyBatis 。2013 年 11 月迁移到 Github。

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

MyBatis 解决的主要问题：

减轻使用 JDBC 的复杂性，不用编写重复的创建 Connetion，Statement；不用编写关闭资源代码。直接使用 java 对象，表示结果数据。让开发者专注 SQL 的处理。 其他分心的工作由 MyBatis 代劳。

Mybatis可以完成：

1. 注册数据库的驱动，例如 Class.forName(“com.mysql.jdbc.Driver”))
2. 创建 JDBC 中必须使用的 Connection，Statement，ResultSet 对象
3. 从 xml 中获取 sql，并执行 sql 语句，把 ResultSet 结果转换 java 对象(List集合)

```
List<Student> list = new ArrayLsit<>();
ResultSet rs = state.executeQuery(“select * from student”);
while(rs.next()){
    Student student = new Student();
    student.setName(rs.getString(“name”));
    student.setAge(rs.getInt(“age”));
    list.add(student);
}
```

4. 关闭资源

   ResultSet.close()，Statement.close()，Conenection.close()

mybatis 是 MyBatis SQL Mapper Framework for Java （sql映射框架）

sql mapper：sql映射，可以把数据库表中的一行数据映射为 一个java对象。
一行数据可以看做是一个java对象。操作这个对象，就相当于操作表中的数据

Data Access Objects（DAOs） ：数据访问，对数据库执行增删改查。

开发人员只需要做的是：提供 sql 语句

最后是：开发人员提供sql语句 ---> mybatis处理sql ---> 开发人员得到List集合或java对象（表中的数据）

总结：
mybatis是一个sql映射框架，提供的数据库的操作能力。增强的JDBC，使用mybatis让开发人员集中精神写sql就可以了，不必关心Connection，Statement，ResultSet
的创建，销毁，sql的执行。

二、MyBatis 快速入门

1 入门案例

```
实现步骤：
1.新建student表
2.加入maven的mybatis坐标，mysql驱动的坐标
3.创建实体类，Student -- 保存表中的一行数据
4.创建持久层的dao接口，定义操作数据库的方法
5.创建一个mybatis使用的配置文件
  就做sql映射文件：写sql语句的，一般一个表一个sql映射文件。这个文件是xml文件
  1.在dao接口所在的目录中
  2.文件名称和接口保持一致

6.创建mybatis的主配置文件
  一个项目就一个主配置文件。
  主配置文件提供了数据库的连接信息和sql映射文件的位置信息
7.创建使用mybatis类
  通过mybatis访问数据库
```

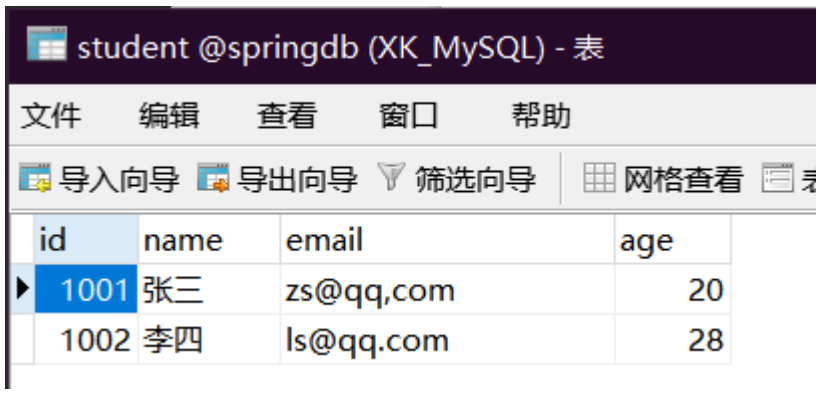
1.1 使用 MyBatis 准备

下载MyBatis：https://github.com/mybatis/mybatis-3/releases

1.2 搭建 MyBatis 开发环境

1. 创建 mysql 数据库和表

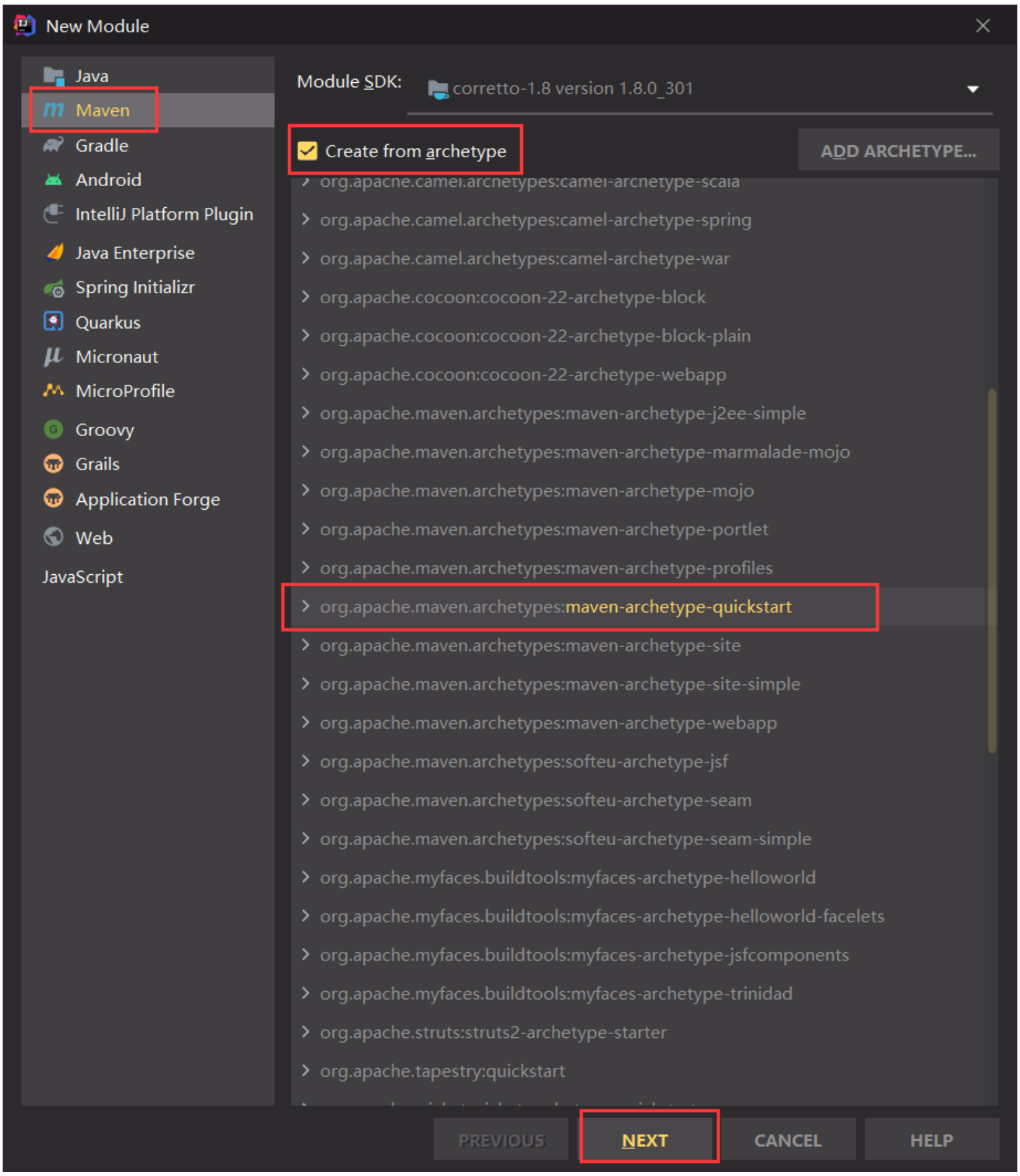
   数据库名springdb：表名student



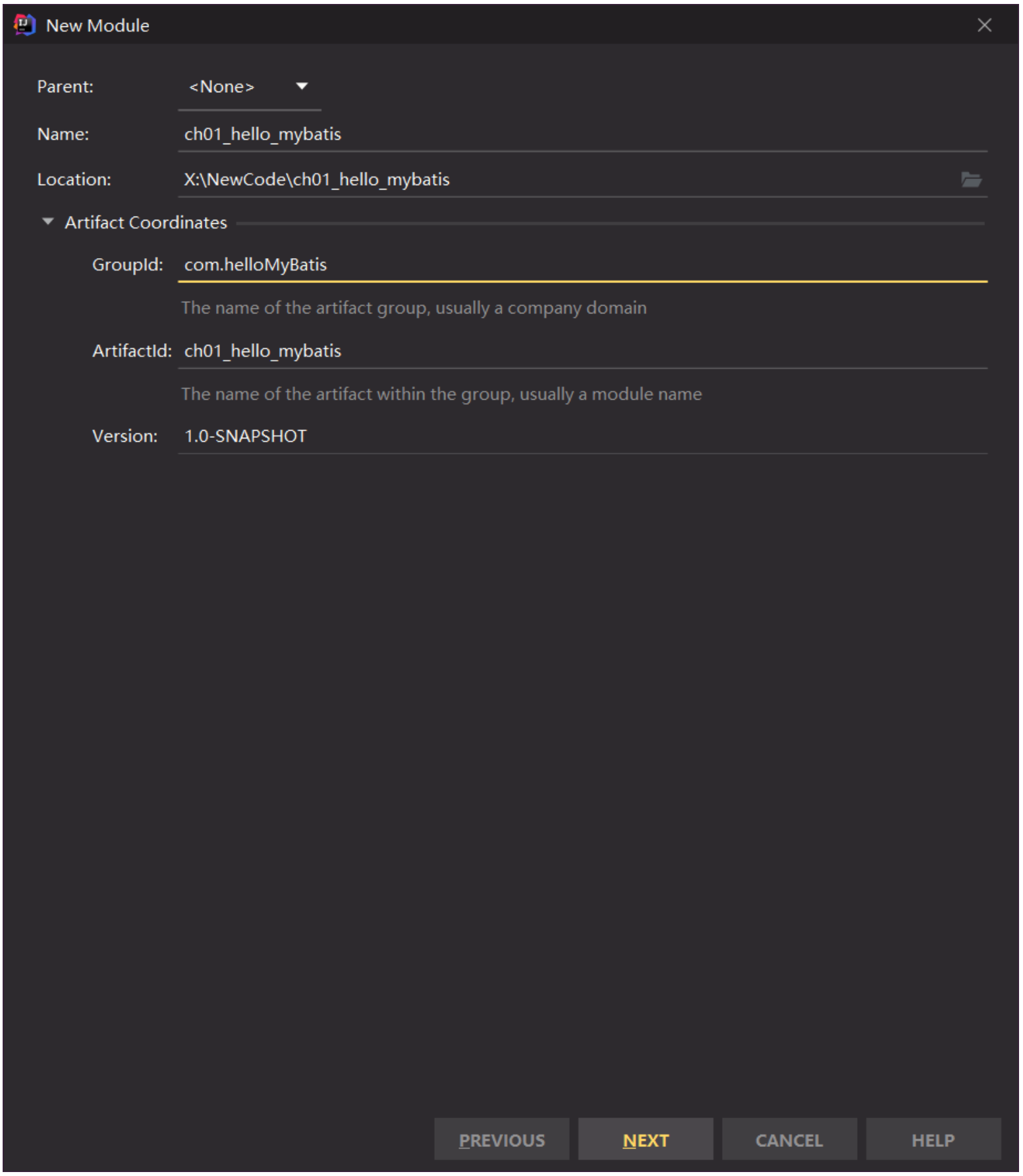
```sql
CREATE TABLE `student` (
 `id` int(11) NOT NULL ,
 `name` varchar(255) DEFAULT NULL,
 `email` varchar(255) DEFAULT NULL,
 `age` int(11) DEFAULT NULL,
 PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

2. 创建Maven工程

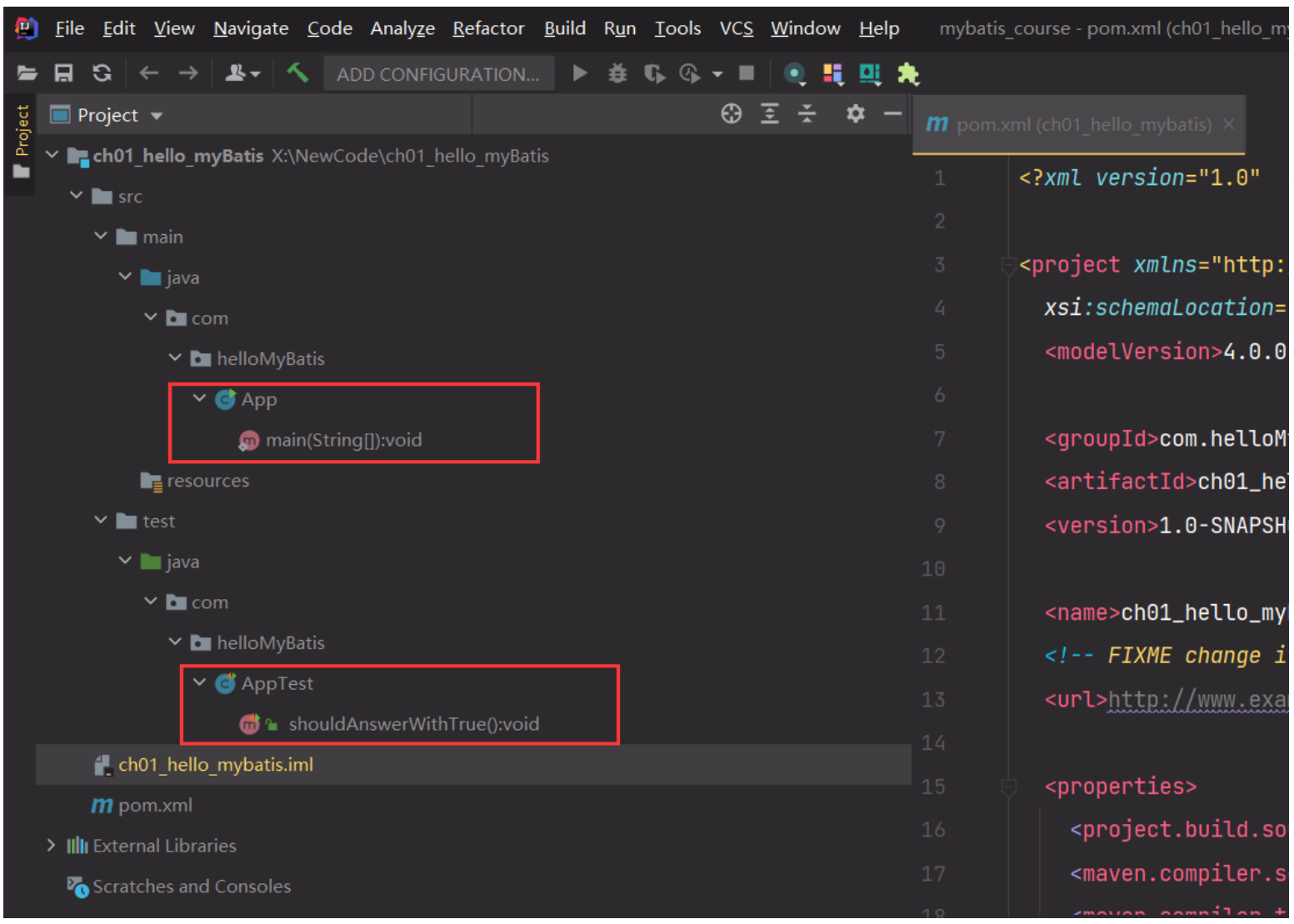
创建Maven工程，信息如下：



工程坐标：



3. 删除默认的App类文件



4. 加入 maven 坐标

pom.xml 加入 maven 坐标：

```
<dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
    <!--mybatis依赖-->
    <dependency>
      <groupId>org.mybatis</groupId>
      <artifactId>mybatis</artifactId>
      <version>3.5.9</version>
    </dependency>
    <!--mysql驱动-->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.9</version>
    </dependency>
  </dependencies>
```

5. 加入 maven 插件

```
<build>
    <resources>
      <resource>
        <directory>src/main/java</directory><!--所在的目录-->
        <includes><!--包括目录下的.properties,.xml 文件都会扫描到-->
          <include>**/*.properties</include>
          <include>**/*.xml</include>
        </includes>
        <filtering>false</filtering>
      </resource>
    </resources>

    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

6. 编写 Student 实体类

创建包 com.helloMyBatis.entity, 包中创建 Student 类

```
package com.helloMyBatis.entity;
//推荐和表名一样，容易记忆
public class Student {
    private Integer id;
    private String name;
    private String email;
    private Integer age;
    //get set toString
}
```

7. 编写 Dao 接口 StudentDao

创建 com.helloMyBatis.dao 包，创建 StudentDao 接口

```
package com.helloMyBatis.dao;

import com.helloMyBatis.entity.Student;
import java.util.List;

//接口操作student表
public interface StudentDao {
    //查询student表中所有的数据
    List<Student> selectStudents();
}
```

8. 编写 Dao 接口 Mapper 映射文件 StudentDao.xml

要求：

1. 在 dao 包中创建文件 StudentDao.xml

2. 要 StudentDao.xml 文件名称和接口 StudentDao.java 一样，区分大小写。

```
<?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.helloMyBatis.dao.StudentDao">
<!--
    sql映射文件（sql mapper）：写sql语句的，mybatis会执行这些sql
    1.指定约束文件
        <!DOCTYPE mapper
            PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
            "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

        mybatis-3-mapper.dtd是约束文件的名称， 扩展名是dtd的。
    2.约束文件作用：限制，检查在当前文件中出现的标签，属性必须符合mybatis的要求。

    3.mapper 是当前文件的根标签，必须的。
    namespace：叫做命名空间，唯一值的，可以是自定义的字符串。
           要求你使用dao接口的全限定名称。

    4.在当前文件中，可以使用特定的标签，表示数据库的特定操作。
    <select>:表示执行查询，select语句
    <update>:表示更新数据库的操作， 就是在<update>标签中 写的是update sql语句
    <insert>:表示插入， 放的是insert语句
    <delete>:表示删除， 执行的delete语句
-->

    <!--
       select:表示查询操作。
       id: 你要执行的sql语法的唯一标识，mybatis会使用这个id的值来找到要执行的sql语句
           可以自定义，但是要求你使用接口中的方法名称。

       resultType:表示结果类型的， 是sql语句执行后得到ResultSet,遍历这个ResultSet得到java对象的类型。
       值写的类型的全限定名称
    -->
    <select id="selectStudents" resultType="com.helloMyBatis.entity.Student" >
        select id,name,email,age from student order by id
    </select>
</mapper>
```

9. 创建 MyBatis 主配置文件

项目 src/main 下创建 resources 目录，设置 resources 目录为 resources Root

创建主配置文件：名称为 mybatis.xml

说明：主配置文件名称是自定义的，

内容如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!--settings：控制 mybatis全局行为-->
    <settings>
        <!--设置mybatis输出日志-->
        <setting name="logImpl" value="STDOUT_LOGGING" />
    </settings>

    <!--环境配置： 数据库的连接信息
        default:必须和某个environment的id值一样。
        告诉mybatis使用哪个数据库的连接信息。也就是访问哪个数据库
    -->
    <environments default="mydev">
        <!--
            environment : 一个数据库信息的配置， 环境
            id:一个唯一值，自定义，表示环境的名称。
        -->
        <environment id="mydev">
            <!--
               transactionManager : mybatis的事务类型
               type: JDBC(表示使用jdbc中的Connection对象的commit，rollback做事务处理)
            -->
            <transactionManager type="JDBC"/>
            <!--
               dataSource:表示数据源，连接数据库的
               type：表示数据源的类型， POOLED表示使用连接池
            -->
            <dataSource type="POOLED">
                <!--
                   driver, user, username, password 是固定的，不能自定义。
```

```
                -->
                <!--数据库的驱动类名-->
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <!--连接数据库的url字符串-->
                <property name="url" value="jdbc:mysql://localhost:3306/springdb"/>
                <!--访问数据库的用户名-->
                <property name="username" value="root"/>
                <!--密码-->
                <property name="password" value="111"/>
            </dataSource>
        </environment>

        <!--表示线上的数据库，是项目真实使用的库-->
        <environment id="online">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/onlinedb"/>
                <property name="username" value="root"/>
                <property name="password" value="fhwertwr"/>
            </dataSource>
        </environment>
    </environments>

    <!-- sql mapper(sql映射文件)的位置-->
    <mappers>
        <!--一个mapper标签指定一个文件的位置。
            从类路径开始的路径信息。   target/clasess(类路径)
            可多次
        -->
        <mapper resource="com/helloMyBatis/dao/StudentDao.xml"/>
        <!--<mapper resource="com/helloMyBatis/dao/SchoolDao.xml" />-->
    </mappers>
</configuration>
<!--
mybatis的主配置文件： 主要定义了数据库的配置信息， sql映射文件的位置
1. 约束文件
    <!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">

     mybatis-3-config.dtd：约束文件的名称

2. configuration 根标签。
-->
```

10. 创建测试类 MyApp

```
package com.helloMyBatis;

import com.helloMyBatis.entity.Student;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.IOException;
import java.io.InputStream;
import java.util.List;

public class MyApp {
    public static void main(String[] args) throws IOException {
        //访问mybatis读取student数据
        //1.定义mybatis主配置文件的名称，从类路径的根开始（target/clasess）
        String config="mybatis.xml";
        //2.读取这个config表示的文件
        InputStream in = Resources.getResourceAsStream(config);
        //3.创建了SqlSessionFactoryBuilder对象
        SqlSessionFactoryBuilder builder  = new SqlSessionFactoryBuilder();
        //4.创建SqlSessionFactory对象
        SqlSessionFactory factory = builder.build(in);
        //5.获取SqlSession对象，从SqlSessionFactory中获取SqlSession
        SqlSession sqlSession = factory.openSession();
        //6.【重要】指定要执行的sql语句的标识。sql映射文件中的namespace + "." + 标签的id值(方法名)
        String sqlId = "com.helloMyBatis.dao.StudentDao" + "." + "selectStudents";
        //String sqlId = "com.helloMyBatis.dao.StudentDao.selectStudents";
        //7. 重要】执行sql语句，通过sqlId找到语句
        List<Student> studentList = sqlSession.selectList(sqlId);
        //8.输出结果
        //studentList.forEach( stu -> System.out.println(stu));
        for(Student stu : studentList){
            System.out.println("查询的学生="+stu);
        }
```

```
        //9.关闭SqlSession对象
        sqlSession.close();
    }
}
/*
Logging initialized using 'class org.apache.ibatis.logging.stdout.StdOutImpl' adapter.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
Opening JDBC Connection
Created connection 1731722639.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@6737fd8f]
==>  Preparing: select id,name,email,age from student order by id
==> Parameters:
<==    Columns: id, name, email, age
<==        Row: 1001, 张三, zs@qq,com, 20
<==        Row: 1002, 李四, ls@qq.com, 28
<==      Total: 2
查询的学生=Student{id=1001, name='张三', email='zs@qq,com', age=20}
查询的学生=Student{id=1002, name='李四', email='ls@qq.com', age=28}
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@6737fd8f]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@6737fd8f]
Returned connection 1731722639 to pool.

Process finished with exit code 0
*/
```

11. 配置日志功能

    mybatis.xml 文件加入日志配置，可以在控制台输出执行的 sql 语句和参数

```
<!--settings：控制 mybatis全局行为-->
<settings>
    <!--设置mybatis输出日志-->
    <setting name="logImpl" value="STDOUT_LOGGING" />
</settings>
```

2 基本的CRUD

2.1 insert

```
//StudentDao.java 接口中增加方法
//插入方法
//参数： student ,表示要插入到数据库的数据
//返回值： int ， 表示执行insert操作后的 影响数据库的行数
int insertStudent(Student student);
```

```
<!--StudentDao.xml 加入sql语句 -->
<!--插入操作-->
<insert id="insertStudent">
    insert into student values(#{id},#{name},#{email},#{age})
</insert>
```

```
    //insert 测试方法
    @Test
    public void testInsert() throws IOException {
        //访问mybatis读取student数据
        //1.定义mybatis主配置文件的名称，从类路径的根开始（target/clasess）
        String config="mybatis.xml";
        //2.读取这个config表示的文件
        InputStream in = Resources.getResourceAsStream(config);
        //3.创建了SqlSessionFactoryBuilder对象
        SqlSessionFactoryBuilder builder  = new SqlSessionFactoryBuilder();
        //4.创建SqlSessionFactory对象
        SqlSessionFactory factory = builder.build(in);
        //5.获取SqlSession对象，从SqlSessionFactory中获取SqlSession
        SqlSession sqlSession = factory.openSession();
        //6.【重要】指定要执行的sql语句的标识。sql映射文件中的namespace + "." + 标签的id值
        String sqlId = "com.helloMyBatis.dao.StudentDao" + "." + "insertStudent";
        //7. 重要】执行sql语句，通过sqlId找到语句
        Student student = new Student();
        student.setId(1004);
        student.setName("刘六");
        student.setEmail("ll@163.com");
        student.setAge(20);
        int nums = sqlSession.insert(sqlId, student);
        //mybatis默认不是自动提交事务的， 所以在insert ，update ，delete后要手工提交事务
        sqlSession.commit();
        //8.输出结果
```

```
        System.out.println("执行insert结果: " + nums);
        //9.关闭SqlSession对象
        sqlSession.close();
    }
```

2.2 update

```
//StudentDao.java 接口中增加方法
//更新方法
//参数:  student ,表示要更新到数据库的数据
//返回值:  int ,  表示执行update操作后的 影响数据库的行数
int updateStudent(Student student);
```

```
<!--StudentDao.xml 加入sql语句 -->
<!--更新操作-->
<update id="updateStudent">
    update student set age=#{age} where id=#{id}
</update>
```

```
    //update测试方法
    @Test
    public void testUpdate() throws IOException {
        //访问mybatis读取student数据
        //1.定义mybatis主配置文件的名称, 从类路径的根开始（target/clasess）
        String config="mybatis.xml";
        //2.读取配置文件
        InputStream in = Resources.getResourceAsStream(config);
        //3.创建了SqlSessionFactoryBuilder对象
        SqlSessionFactoryBuilder builder  = new SqlSessionFactoryBuilder();
        //4.创建SqlSessionFactory对象
        SqlSessionFactory factory = builder.build(in);
        //5.获取SqlSession对象，从SqlSessionFactory中获取SqlSession
        SqlSession sqlSession = factory.openSession();
        //6.【重要】指定要执行的sql语句的标识。sql映射文件中的namespace + "." + 标签的id值
        String sqlId = "com.helloMyBatis.dao.StudentDao" + "." + "updateStudent";
        //7. 重要】执行sql语句，通过sqlId找到语句
        Student student = new Student();
        student.setId(1004);
        student.setAge(18);
        int nums = sqlSession.update(sqlId, student);
        //手工提交事务
        sqlSession.commit();
        //8.输出结果
        System.out.println("执行update结果: " + nums);
        //9.关闭SqlSession对象
        sqlSession.close();
    }
```

2.3 delete

```
//StudentDao.java 接口中增加方法
//删除方法
//参数:  id ,表示要删除数据的学生id
//返回值:  int ,  表示执行delete操作后的 影响数据库的行数
int deleteStudent(int id);
```

```
<!--StudentDao.xml 加入sql语句 -->
<!--更新操作-->
<delete id="deleteStudent">
    delete from student where id=#{id};
</delete>
```

```
    //delete测试方法
    @Test
    public void testDelete() throws IOException {
        //访问mybatis读取student数据
        //1.定义mybatis主配置文件的名称, 从类路径的根开始（target/classes）
        String config="mybatis.xml";
        //2.读取配置文件
        InputStream in = Resources.getResourceAsStream(config);
        //3.创建了SqlSessionFactoryBuilder对象
        SqlSessionFactoryBuilder builder  = new SqlSessionFactoryBuilder();
        //4.创建SqlSessionFactory对象
        SqlSessionFactory factory = builder.build(in);
        //5.获取SqlSession对象，从SqlSessionFactory中获取SqlSession
        SqlSession sqlSession = factory.openSession();
        //6.【重要】指定要执行的sql语句的标识。sql映射文件中的namespace + "." + 标签的id值
        String sqlId = "com.helloMyBatis.dao.StudentDao" + "." + "deleteStudent";
        //7. 重要】执行sql语句，通过sqlId找到语句
```

```
        int id = 1002;//要删除的学生id
        int nums = sqlSession.delete(sqlId, id);
        //手工提交事务
        sqlSession.commit();
        //8.输出结果
        System.out.println("执行update结果: " + nums);
        //9.关闭SqlSession对象
        sqlSession.close();
    }
```

3 MyBatis对象分析

3.1 对象使用

1. Resources 类：Resources 类，顾名思义就是资源，用于读取资源文件。其有很多方法通过加载并解析资源文件，返回不同类型的 IO 流对象。

```
InputStream in = Resources.getResourceAsStream("mybatis.xml");
```

2. SqlSessionFactoryBuilder 类：SqlSessionFactory 的创建，需要使用 SqlSessionFactoryBuilder 对象的build() 方法。由于 SqlSessionFactoryBuilder 对象在创建完工厂对象后，就完成了其历史使命，即可被销毁。所以，一般会将该 SqlSessionFactoryBuilder 对象创建为一个方法内的局部对象，方法结束，对象销毁。

```
SqlSessionFactoryBuilder builder  = new SqlSessionFactoryBuilder();
//创建SqlSessionFactory对象
SqlSessionFactory factory = builder.build(in);
```

3. SqlSessionFactory 接口：SqlSessionFactory 接口对象是一个重量级对象（系统开销大的对象），是线程安全的，所以一个应用只需要一个该对象即可。创建 SqlSession 需要使用 SqlSessionFactory 接口的 openSession()方法。
   - openSession(true)：创建一个有自动提交功能的 SqlSession
   - openSession(false)：创建一个非自动提交功能的 SqlSession，需手动提交
   - openSession()：同 openSession(false)

```
SqlSession sqlSession = factory.openSession();
```

4. SqlSession 接口

   SqlSession 接口对象用于执行持久化操作。一个 SqlSession 对应着一次数据库会话，一次会话以 SqlSession 对象的创建开始，以 SqlSession 对象的关闭结束。

   SqlSession 接口对象是线程不安全的，所以每次数据库会话结束前，需要马上调用其 close()方法，将 其关闭。再次需要会话，再次创建。 SqlSession 在方法内部创建，使用完毕后关闭。

   定义了操作数据的方法 例如 selectOne()、selectList()、insert()、update()、delete()、commit()、rollback()

3.2 创建工具类

```
package com.helloMyBatis.util;

import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.IOException;
import java.io.InputStream;

public class MyBatisUtils {
    private static SqlSessionFactory factory = null;
    //静态代码块，类加载的时候执行
    static {
        String config = "mybatis.xml";
        try {
            InputStream in = Resources.getResourceAsStream(config);
            //创建SqlSessionFactory对象，使用SqlSessionFactoryBuilder
            factory = new SqlSessionFactoryBuilder().build(in);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    //获取SqlSession的方法
    public static SqlSession getSqlSession(){
        SqlSession sqlSession = null;
        if(factory != null){
            sqlSession = factory.openSession();//非自动提交事务
        }
        return sqlSession;
    }
```

```
}
```

```
//测试工具类
package com.helloMyBatis;

import com.helloMyBatis.entity.Student;
import com.helloMyBatis.util.MyBatisUtils;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.IOException;
import java.io.InputStream;
import java.util.List;

public class MyApp2 {
    public static void main(String[] args) throws IOException {
        //获取SqlSession对象，从SqlSessionFactory中获取SqlSession
        SqlSession sqlSession = MyBatisUtils.getSqlSession();
        //指定要执行的sql语句的标识。sql映射文件中的namespace + "." + 标签的id值
        String sqlId = "com.helloMyBatis.dao.StudentDao" + "." + "selectStudents";
        //【重要】执行sql语句，通过sqlId找到语句
        List<Student> studentList = sqlSession.selectList(sqlId);
        //输出结果
        studentList.forEach( stu -> System.out.println(stu));
        //提交事务
        sqlSession.commit();
        //关闭SqlSession对象
        sqlSession.close();
    }
}
```

4 MyBatis 使用传统 Dao 开发方式

4.1 Dao开发

```
//创建Dao接口
package com.xukang.dao;

import com.xukang.entity.Student;

import java.util.List;

public interface StudentDao {
    List<Student> selectStudents();
}
```

```
<!--StudentDao.xml 文件-->
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.xukang.dao.StudentDao">
    <select id="selectStudents" resultType="com.xukang.entity.Student">
        select id,name,email,age from student order by id
    </select>
</mapper>
```

```
//创建 Dao 接口实现类
package com.xukang.dao.impl;

import com.xukang.dao.StudentDao;
import com.xukang.entity.Student;
import com.xukang.util.MyBatisUtil;
import org.apache.ibatis.session.SqlSession;

import java.util.List;

public class StudentDaoImpl implements StudentDao {
    @Override
    public List<Student> selectStudents() {
        //获取SqlSession对象
        SqlSession sqlSession = MyBatisUtil.getSqlSession();
        String sqlId = "com.xukang.dao.StudentDao.selectStudents";
        //执行sql语句，使用SqlSession类的方法
        List<Student> students = sqlSession.selectList(sqlId);
        students.forEach(stu -> System.out.println(stu));
        //提交事务
```

```
        sqlSession.commit();
        //关闭资源
        sqlSession.close();
        return students;
    }
}
```

```
<!--myBatis.xml 配置文件-->
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <environments default="mydev">
        <environment id="mydev">
            <!--
               transactionManager ：mybatis的事务类型
               type: JDBC(表示使用jdbc中的Connection对象的commit，rollback做事务处理)
            -->
            <transactionManager type="JDBC"/>
            <!--
               dataSource:表示数据源，连接数据库的
               type：表示数据源的类型， POOLED表示使用连接池
            -->
            <dataSource type="POOLED">
                <!--数据库的驱动类名-->
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <!--连接数据库的url字符串-->
                <property name="url" value="jdbc:mysql://localhost:3306/springdb"/>
                <!--访问数据库的用户名-->
                <property name="username" value="root"/>
                <!--密码-->
                <property name="password" value="111"/>
            </dataSource>
        </environment>
    </environments>

    <!-- sql mapper(sql映射文件)的位置-->
    <mappers>
        <mapper resource="com/xukang/dao/StudentDao.xml"/>
    </mappers>
</configuration>
```

```
//测试类
public class TestMyBatis {
    @Test
    public void testSelect(){
        StudentDao dao = new StudentDaoImpl();
        List<Student> students = dao.selectStudents();
        System.out.println("================================");
        students.forEach(stu -> System.out.println(stu));
    }
}
```

4.2 传统 Dao 开发方式的分析

在前面例子中自定义 Dao 接口实现类时发现一个问题：Dao 的实现类其实并没有干什么实质性的工作，它仅仅就是通过 SqlSession 的相关 API 定位到映射文件 mapper 中相应 id 的 SQL 语句，真正对 DB 进行操作的工作其实是由框架通过 mapper 中的 SQL 完成的。

所以，MyBatis 框架就抛开了 Dao 的实现类，直接定位到映射文件 mapper 中的相应 SQL 语句，对 DB 进行操作。这种对 Dao 的实现方式称为 Mapper 的动态代理方式。

Mapper 动态代理方式无需程序员实现 Dao 接口。接口是由 MyBatis 结合映射文件自动生成的动态代理实现的。

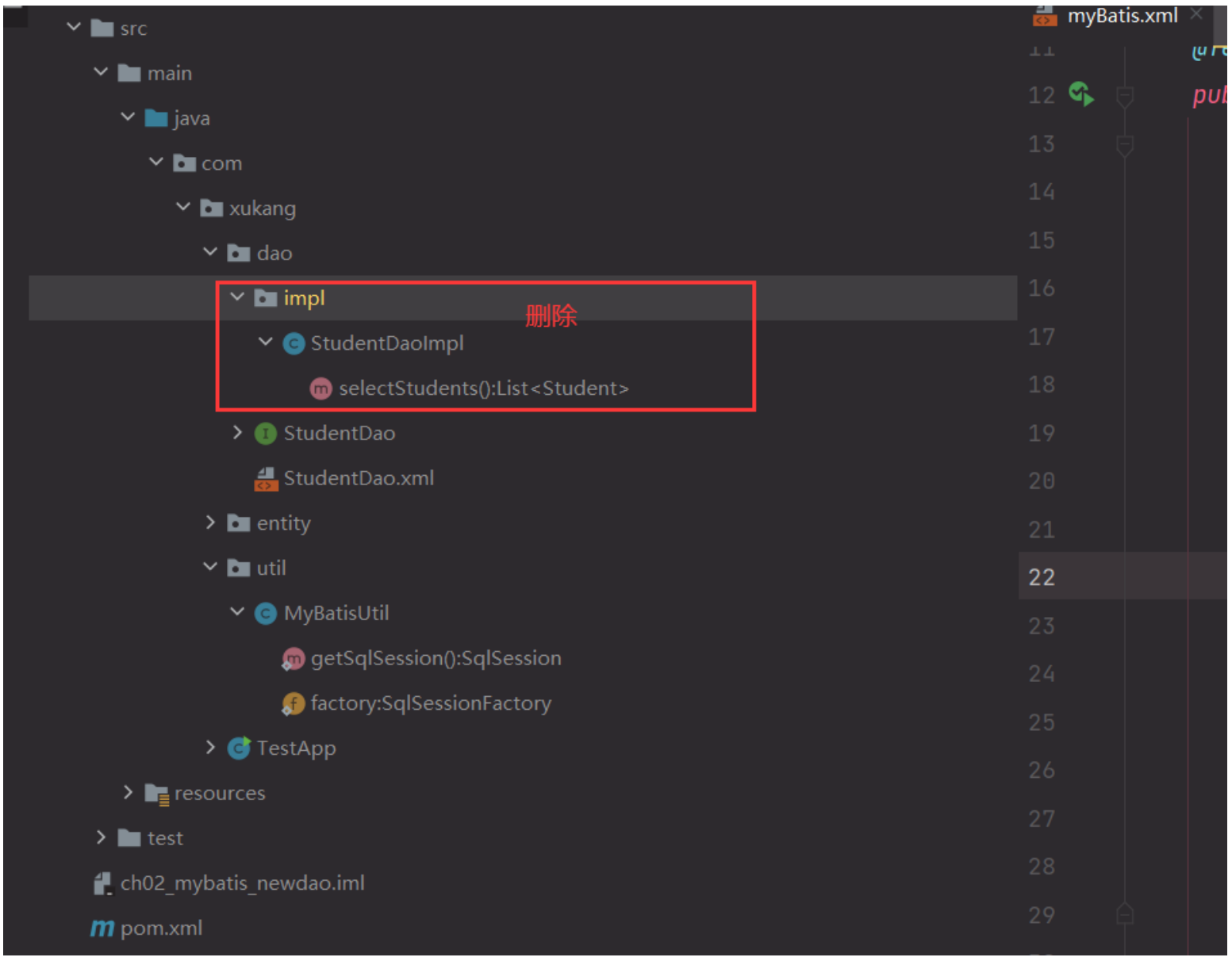
```
分析
List<Student> studentList  = dao.selectStudents(); //调用
/*
1.dao对象，类型是StudentDao，全限定名称是：com.bjpowernode.dao.StudentDao
  全限定名称 和 namespace 是一样的。

2.方法名称， selectStudents， 这个方法就是 mapper文件中的 id值 selectStudents

3.通过dao中方法的返回值也可以确定MyBatis要调用的SqlSession的方法
如果返回值是List ，调用的是SqlSession.selectList()方法。
如果返回值 int ，或是非List的， 看mapper文件中的 标签是<insert>，<update> 就会调用
SqlSession的insert， update等方法

mybatis的动态代理： mybatis根据dao的方法调用，获取执行sql语句的信息。
mybatis根据你的dao接口，创建出一个dao接口的实现类，并创建这个类的对象。
完成SqlSession调用方法，访问数据库。
*/
```

三、MyBatis 框架 Dao 代理

1 Dao 代理实现 CRUD

步骤:

1. 去掉 Dao 接口实现类



2. getMapper 获取代理对象

   只需调用 SqlSession 的 getMapper()方法，即可获取指定接口的实现类对象。该方法的参数为指定 Dao 接口类的 class 值。目的是获取这个Dao接口的对象

```
/**
* 使用mybatis的动态代理机制， 使用SqlSession.getMapper(dao接口)
* getMapper能获取dao接口对于的实现类对象。
*/
SqlSession sqlSession = MyBatisUtil.getSqlSession();
StudentDao dao  =  sqlSession.getMapper(StudentDao.class);

System.out.println("dao="+dao.getClass().getName());//com.sun.proxy.$Proxy2 : jdk的动态代理
//调用dao的方法， 执行数据库的操作
List<Student> students = dao.selectStudents();
for(Student stu: students){
   System.out.println("学生="+stu);
}
sqlSession.commit();
sqlSession.close();
```

2 深入理解参数

2.1 parameterType

parameterType: 接口中方法参数的类型， 类型的完全限定名或别名。这个属性是可选的，因为 MyBatis 可以推断出具体传入语句的参数，默认值为未设置（unset）。接口中方法的参数从 java 代码传入到 mapper 文件的 sql 语句。

```
    <!--
      parameterType :  dao接口中方法参数的数据类型。
        parameterType它的值是java的数据类型全限定名称或者是mybatis定义的别名
        例如:  parameterType="java.lang.Integer"
               parameterType="int"

        注意：parameterType不是强制的，mybatis通过反射机制能够发现接口参数的数类型。
        所以可以没有。 一般我们也不写。
    -->
    <select id="selectStudentById" parameterType="int" resultType="com.bjpowernode.domain.Student">
        select id,name, email,age from student where id=${studentId}
```

```
    </select>

<!--
       使用#{}之后， mybatis执行sql是使用的jdbc中的PreparedStatement对象
        由mybatis执行下面的代码：
         1. mybatis创建Connection ， PreparedStatement对象
            String sql="select id,name, email,age from student where id=?";
            PreparedStatement pst = conn.preparedStatement(sql);
            pst.setInt(1,1001);

         2. 执行sql封装为resultType="com.bjpowernode.domain.Student"这个对象
            ResultSet rs = ps.executeQuery();
            Student student = null;
            while(rs.next()){
               //从数据库取表的一行数据， 存到一个java对象属性中
               student = new Student();
               student.setId(rs.getInt("id"));
               student.setName(rs.getString("name"));
               student.setEmail(rs.getString("email"));
               student.setAge(rs.getInt("age"));
            }

           return student;  //给了dao方法调用的返回值
-->
```

别名	数据类型
_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

2.2 MyBatis 传递参数

1. 一个简单类型参数

   Dao 接口中方法的参数只有一个简单类型（java 基本类型和 String），占位符 #{ 任意字符 }，和方 法的参数名无关。

```
    /**
     * 一个简单类型的参数：
     *   简单类型： mybatis把java的基本数据类型和String都叫简单类型。
     *  在mapper文件获取简单类型的一个参数的值，使用 #{任意字符}
     */
    public Student selectStudentById(Integer id);
```

```
<select id="selectStudentById" parameterType="int" resultType="com.xukang.domain.Student">
   select id,name, email,age from student where id = #{studentId}
</select>
#{studentId} , studentId 是自定义的变量名称，和方法参数名无关。
```

2. 多个参数-使用@Param

   当 Dao 接口方法多个参数，需要通过名称使用参数。在方法形参前面加入@Param(“自定义参数名”)， mapper 文件使用 #{自定义参数名}。

```
    /**
     * 多个参数：命名参数，在形参定义的前面加入 @Param("自定义参数名称")
     */
    List<Student> selectMultiParam(@Param("myname") String name,
                                   @Param("myage") Integer age);
```

```
    <!--多个参数，使用@Param命名-->
    <select id="selectMultiParam" resultType="com.xukang.domain.Student">
         select id,name,email,age from student where name=#{myname} and age=#{myage}
    </select>
```

3. 多个参数-使用对象

   使用 java 对象传递参数， java 的属性值就是 sql 需要的参数值。 每一个属性就是一个参数。

```
package com.bjpowernode.vo;
public class QueryParam {
    private String paramName;
    private Integer paramAge;

    public String getParamName() {
        return paramName;
    }
    public void setParamName(String paramName) {
        this.paramName = paramName;
    }
    public Integer getParamAge() {
        return paramAge;
    }
    public void setParamAge(Integer paramAge) {
        this.paramAge = paramAge;
    }
}
```

```
    /**
     * 多个参数，使用java对象作为接口中方法的参数
     */
    List<Student> selectMultiObject(QueryParam param);
```

```
<!--多个参数， 使用java对象的属性值，作为参数实际值
   使用对象语法： #{属性名,javaType=类型名称,jdbcType=数据类型} 很少用。
                javaType:指java中的属性数据类型。
                jdbcType:在数据库中的数据类型。
                例如： #{paramName, javaType=java.lang.String, jdbcType=VARCHAR}

   我们使用的简化方式： #{属性名}，javaType, jdbcType的值mybatis反射能获取。不用提供
-->
<select id="selectMultiObject" resultType="com.xukang.domain.Student">
     select id,name, email,age from student where
     name=#{paramName, javaType=java.lang.String, jdbcType=VARCHAR}
     or age=#{paramAge, javaType=java.lang.Integer, jdbcType=INTEGER}
</select>
OR
<select id="selectMultiObject" resultType="com.xukang.domain.Student">
     select id,name,email,age from student where
     name=#{paramName}   or age=#{paramAge}
</select>
```

```java
@Test
public void testSelectMultiObject(){
    SqlSession sqlSession = MyBatisUtils.getSqlSession();
    StudentDao dao = sqlSession.getMapper(StudentDao.class);

    QueryParam param = new QueryParam();
    param.setParamName("张三");
    param.setParamAge(28);
    List<Student> students = dao.selectMultiObject(param);

    for(Student stu: students){
        System.out.println("学生="+stu);
    }
    sqlSession.close();
}
```

4. 多个参数-按位置

参数位置从 0 开始， 引用参数语法 #{ arg 位置 } ， 第一个参数是#{arg0}, 第二个是#{arg1} 注意：mybatis-3.3 版本和之前的版本使用#{0}, #{1}方式， 从 mybatis3.4 开始使用#{arg0}方式。

```java
/**
 * 多个参数-简单类型的，按位置传值，
 * mybatis.3.4之前，使用 #{0} ，#{1}
 * mybatis。3.4之后 ，使用 #{arg0} ,#{arg1}
 */
List<Student> selectMultiPosition( String name,Integer age);
```

```xml
<!--多个参数使用位置-->
<select id="selectMultiPosition" resultType="com.bjpowernode.domain.Student">
      select id,name, email,age from student where
      name = #{arg0} or age=#{arg1}
</select>
```

```java
@Test
public void testSelectMultiPosition(){
    SqlSession sqlSession = MyBatisUtils.getSqlSession();
    StudentDao dao = sqlSession.getMapper(StudentDao.class);

    List<Student> students = dao.selectMultiPosition("李四",20);

    for(Student stu: students){
        System.out.println("学生="+stu);
    }
    sqlSession.close();
}
```

5. 多个参数-使用 Map

Map 集合可以存储多个值，使用Map向 mapper 文件一次传入多个参数。Map 集合使用 String的 key， Object 类型的值存储参数。mapper 文件使用 # { key } 引用参数值。

```java
/**
 * 多个参数，使用Map存放多个值
 */
List<Student> selectMultiByMap(Map<String,Object> map);
```

```xml
<!--多个参数，使用Map ，使用语法 #{map的key}-->
<select id="selectMultiByMap" resultType="com.bjpowernode.domain.Student">
      select id,name, email,age from student where
      name = #{myname} or age=#{age1}
</select>
```

```java
@Test
public void testSelectMultiByMap(){
    SqlSession sqlSession = MyBatisUtils.getSqlSession();
    StudentDao dao = sqlSession.getMapper(StudentDao.class);

    Map<String,Object> data = new HashMap<>();
    data.put("myname","张三");
    data.put("age1",28);

    List<Student> students = dao.selectMultiByMap(data);

    for(Student stu: students){
        System.out.println("学生="+stu);
    }
    sqlSession.close();
}
```

3 # 和 $ 占位符的比较

#：占位符，告诉 mybatis 使用实际的参数值代替。并使用 PrepareStatement 对象执行 sql 语句，#{...} 代替 sql 语句的 “?”。这样做更安全，更迅速，通常也是首选做法，

```
mapper 文件：
    <select id="selectById" resultType="com.bjpowernode.domain.Student">
        select id,name,email,age from student where id=#{studentId}
    </select>

转为 MyBatis 的执行是：
    String sql=” select id,name,email,age from student where id=?”;
    PreparedStatement ps = conn.prepareStatement(sql);
    ps.setInt(1,1005);

解释：
    where id=? 就是 where id=#{studentId}
    ps.setInt(1,1005) , 1005 会替换掉 #{studentId}
```

$：字符串替换，告诉 mybatis 使用$包含的“字符串”替换所在位置。使用 Statement 把 sql 语句和${}的 内容连接起来。主要用在替换表名，列名，不同列排序等操作。

```
select id,name, email,age from student where id=#{studentId}
# 的结果： select id,name, email,age from student where id=?


select id,name, email,age from student where id=${studentId}
$ 的结果：select id,name, email,age from student where id=1001

String sql="select id,name,email,age from student where id=" + "1001";
使用的Statement对象执行sql，效率比PreparedStatement低。
```

```
List<Student> selectUse$Order(@Param("colName") String colName);
```

```
    <!--
       $替换列名
    -->
    <select id="selectUse$Order" resultType="com.bjpowernode.domain.Student">
         select * from student order by ${colName}
    </select>
```

```
    @Test
    public void testSelectUse$Order(){
        SqlSession sqlSession = MyBatisUtils.getSqlSession();
        StudentDao dao = sqlSession.getMapper(StudentDao.class);

        List<Student> students = dao.selectUse$Order("age");

        for(Student stu: students){
            System.out.println("学生="+stu);
        }
        sqlSession.close();
    }
```

"#" 和 "$" 区别

- #使用？ 在sql语句中做占位的，使用PreparedStatement执行sql，效率高
- #能够避免sql注入，更安全。
- $不使用占位符，是字符串连接方式，使用Statement对象执行sql，效率低
- $有sql注入的风险，缺乏安全性。
- $:可以替换表名或者列名

4 封装 MyBatis 的输出结果

4.1 resultType

resultType：

```
指sql语句执行完毕后，数据转为的java对象，java类型是任意的。
resultType结果类型它的值
    1.类型的全限定名称
    2.类型的别名，例如 java.lang.Integer 别名是 int
执行 sql 得到 ResultSet 转换的类型，使用类型的完全限定名或别名。注意如果返回的是集合，那应该设置为集合包含的类型，而不是集合本身。resultType 和 resultMap，不能同时使用。

处理方式：
1. mybatis执行sql语句，然后mybatis调用类的无参数构造方法，创建对象。
2. mybatis把ResultSet指定列值付给同名的属性。
```

```
<select id="selectMultiPosition" resultType="com.bjpowernode.domain.Student">
   select id,name, email,age from student
</select>

对等的jdbc
ResultSet rs = executeQuery(" select id,name, email,age from student" )
while(rs.next()){
    Student student = new Student();
    student.setId(rs.getInt("id"));
    student.setName(rs.getString("name"));
}
```

- 简单类型

```
//接口方法
int countStudent();
```

```
<!-- mapper文件 -->
<select id="countStudent" resultType="int">
    select count(*) from student
</select>
```

```
    //测试代码
    @Test
    public void testCountStudent(){
        SqlSession sqlSession = MyBatisUtil.getSqlSession();
        StudentDao mapper = sqlSession.getMapper(StudentDao.class);
        int counts = mapper.countStudent();
        System.out.println("学生的数量为" + counts + "位。");
    }
```

- 对象类型

```
//接口方法
Student selectAllStudent();
```

```
<!-- mapper文件 -->
<select id="selectAllStudent" resultType="org.example.entity.Student">
   select id,name,email,age from student
</select>
```

```
    //测试代码
    @Test
    public void testSelectById(){
        SqlSession sqlSession = MyBatisUtil.getSqlSession();
        StudentDao mapper = sqlSession.getMapper(StudentDao.class);
        Student students = mapper.selectAllStudent();
        students.forEach(s -> System.out.println("student" + s));
        sqlSession.close();
    }
```

框架的处理： 使用构造方法创建对象。调用 setXXX 给属性赋值。 Student student = new Student();

sql语句列	java对象方法
id	setId(rs.getInt(“id”))
name	setName(rs.getString(“name”))
email	setEmail(rs.getString(“email”))
age	setAge(rs.getInt(“age”))

调用列名对应的 set 方法

注意：Dao 接口方法返回是集合类型，需要指定集合中的类型，不是集合本身。

- Map

```
//接口方法
//定义方法返回Map
Map<Object,Object> selectMapById(Integer id);
```

```
<!-- mapper文件 -->
<!--返回Map
   1）列名是map的key，列值是map的value
   2)只能最多返回一行记录。多余一行是错误
-->
<select id="selectMapById" resultType="java.util.HashMap">
    select id,name,email from student where id=#{stuid}
</select>
```

```
    //测试代码
    //返回Map
    @Test
    public void testSelecMap(){
        SqlSession sqlSession = MyBatisUtil.getSqlSession();
        StudentDao dao  =  sqlSession.getMapper(StudentDao.class);

        Map<Object,Object> map = dao.selectMapById(1001);
        System.out.println("map=="+map);//map=={name=张三, id=1001, email=zs@qq,com}
        sqlSession.close();
    }
```

4.2 resultMap

resultMap 可以自定义 sql 的结果和 java 对象属性的映射关系。更灵活的把列值赋值给指定属性。常用在列名和 java 对象属性名不一样的情况。

使用方式：

1. 先定义 resultMap，指定列名和属性的对应关系。
2. 在中把 resultType 替换为 resultMap。

```
resultMap:结果映射， 指定列名和java对象的属性对应关系。
    1.你自定义列值赋值给哪个属性
    2.当你的列名和属性名不一样时，一定使用resultMap

注意：resultMap和resultType不要一起用，二选一
```

```
    /*
    *  使用resultMap定义映射关系
    * */
    List<Student> selectAllStudents();
```

```
    <!--使用resultMap
        1)先定义resultMap
        2)在select标签，使用resultMap来引用1定义的。
    -->
    <!--定义resultMap
        id:自定义名称，表示你定义的这个resultMap
        type：java类型的全限定名称
    -->
    <resultMap id="studentMap" type="org.example.entity.Student">
        <!--列名和java属性的关系-->
        <!--主键列，使用id标签
            column :列名
            property:java类型的属性名
        -->
        <id column="id" property="id" />
        <!--非主键列，使用result-->
        <result column="name" property="name" />
        <result column="email" property="email" />
        <result column="age" property="age" />
    </resultMap>

    <select id="selectAllStudents" resultMap="studentMap">
        select id,name, email , age from student
    </select>
```

```java
    @Test
    public void testSelectAllStudents(){
        SqlSession sqlSession = MyBatisUtil.getSqlSession();
        StudentDao dao = sqlSession.getMapper(StudentDao.class);
        List<Student> students = dao.selectAllStudents();
        for(Student stu: students){
            System.out.println("学生="+stu);
        }
        sqlSession.close();
    }
```

4.3 实体类属性名和列名不同的处理方式

```java
public class MyStudent {
    private Integer stuid;
    private String stuname;
    private String stuemail;
    private Integer stuage;
    //get set toString
}
```

1. 使用 列别名 和 resultType

```java
//StudentDao
List<MyStudent> selectDiffColProperty();
```

```xml
<!--
   列名和属性名不一样
   resultType的默认原则是 同名的列值赋值给同名的属性， 使用列别名(java对象的属性名)
-->
<select id="selectDiffColProperty" resultType="org.example.entity.MyStudent">
   select id as stuid ,name as stuname, email as stuemail, age as stuage from student
</select>
```

```java
@Test
public void testSelectDiffColProperty(){
    SqlSession sqlSession = MyBatisUtil.getSqlSession();
    StudentDao dao = sqlSession.getMapper(StudentDao.class);
    List<MyStudent> students = dao.selectDiffColProperty();
    for(MyStudent stu: students){
        System.out.println("学生="+stu);
    }
    sqlSession.close();
}
```

2. 使用 resultMap

```java
List<MyStudent> selectMyStudent();
```

```xml
    <resultMap id="myStudentMap" type="org.example.entity.MyStudent">
        <!--列名和java属性的关系-->
        <id column="id" property="stuid" />
        <!--非主键列，使用result-->
        <result column="name" property="stuname" />
        <result column="email" property="stuemail" />
        <result column="age" property="stuage" />
    </resultMap>
    <select id="selectMyStudent" resultMap="myStudentMap">
        select id,name, email , age from student
    </select>
```

```java
    @Test
    public void testSelectAllStudents2(){
        SqlSession sqlSession = MyBatisUtil.getSqlSession();
        StudentDao dao = sqlSession.getMapper(StudentDao.class);
        List<MyStudent> students = dao.selectMyStudent();
        for(MyStudent stu: students){
            System.out.println("学生="+stu);
        }
        sqlSession.close();
    }
```

4.4 定义别名

```
定义自定义类型的别名
    1.在mybatis主配置文件中定义，使用<typeAlias>定义别名
    2.可以在resultType中使用自定义别名
```

```
    <!--mybatis.xml-->
    <!--定义别名-->
    <typeAliases>
        <!--
            第一种方式：
            可以指定一个类型一个自定义别名
            type:自定义类型的全限定名称
            alias:别名（短小，容易记忆的）
        -->
        <typeAlias type="com.bjpowernode.domain.Student" alias="stu" />
        <typeAlias type="com.bjpowernode.vo.ViewStudent" alias="vstu" />

        <!--
          第二种方式
          <package> name是包名，这个包中的所有类，类名就是别名（类名不区分大小写）
        -->
        <package name="com.bjpowernode.domain"/>
        <package name="com.bjpowernode.vo"/>
    </typeAliases>
```

5 模糊 like

模糊查询的实现有两种方式

1. java 代码中给查询数据加上“%”;

```
//接口方法:
/*第一种模糊查询，在java代码指定 like 的内容*/
List<Student> selectLikeOne(String name);
```

```
<!--mapper 文件-->
<!--第一种 like: java代码指定 like的内容-->
<select id="selectLikeOne" resultType="org.example.entity.Student">
    select id,name,email,age from student where name like #{name}
</select>
```

```
//测试方法
@Test
public void testSelectLikeOne(){
    SqlSession sqlSession = MyBatisUtil.getSqlSession();
    StudentDao dao = sqlSession.getMapper(StudentDao.class);
    //准备好like的内容
    String name = "%李%";
    List<Student> students = dao.selectLikeOne(name);
    for(Student stu: students){
        System.out.println("#######学生="+stu);
    }
    sqlSession.close();
}
```

2. 在 mapper 文件 sql 语句的 条件位置加上“%”

```
//接口方法:
/*name就是李值，在mapper中拼接  like "%" 李 "%" */
List<Student> selectLikeTwo(String name);
```

```
<!--mapper 文件-->
<!--第二种 like: 在mapper文件中拼接 like的内容-->
<select id="selectLikeTwo" resultType="org.example.entity.Student">
    select id,name,email,age from student where name  like "%" #{name} "%"
</select>
```

```java
//测试方法
@Test
public void testSelectLikeOne(){
    SqlSession sqlSession = MyBatisUtil.getSqlSession();
    StudentDao dao = sqlSession.getMapper(StudentDao.class);
    //准备好like的内容
    String name = "李";
    List<Student> students = dao.selectLikeOne(name);
    for(Student stu: students){
        System.out.println("#######学生="+stu);
    }
    sqlSession.close();
}
```

四、MyBatis 框架动态 SQL

```
动态 SQL，通过 MyBatis 提供的各种标签对条件作出判断以实现动态拼接 SQL 语句。这里的条件判
断使用的表达式为 OGNL 表达式。常用的动态 SQL 标签有<if>、<where>、<choose>、<foreach>等。
MyBatis 的动态 SQL 语句，与 JSTL 中的语句非常相似。

动态 SQL，主要用于解决查询条件不确定的情况：在程序运行期间，根据用户提交的查询条件进行
查询。提交的查询条件不同，执行的 SQL 语句不同。若将每种可能的情况均逐一列出，对所有条件进行
排列组合，将会出现大量的 SQL 语句。此时，可使用动态 SQL 来解决这样的问题
```

1 环境准备

创建新的 maven 项目，加入 mybatis，mysql 驱动依赖

创建实体类 Student，StudentDao 接口，StudentDao.xml，mybatis.xml，测试类

使用之前的表 student

在 mapper 的动态 SQL 中若出现大于号（>）、小于号（<）、大于等于号（>=）、小于等于号（<=）等符号，最好将其转换为实体符号。否则，XML 可能会出现解析出错问题。

特别是对于小于号（<），在 XML 中是绝不能出现的。否则解析 mapper 文件会出错。

```
<       小于        &lt;
>       大于        &gt;
<=      小于等于     &lt;=
>=      大于等于     &gt;=
```

2 动态SQL之 if

```
对于该标签的执行，当 test 的值为 true 时，会将其包含的 SQL 片断拼接到其所在的 SQL 语句中。
语法：
<if test=”条件”>
    sql 语句的部分
</if>
```

```java
//接口方法
//动态sql，使用Java对象作为参数
List<Student> selectStudentIf(Student student);
```

```xml
    <!--mapper文件-->
    <select id="selectStudentIf" resultType="org.example.entity.Student">
        select id, name, email, age from student
        where id > 0
        <if test="name != null and name != ''">
            name = #{name}
        </if>
        <if test="age > 0">
            or age > #{age}
        </if>
    </select>
```

```
    @Test
    public void testSelectStudentIf(){
        SqlSession sqlSession = MyBatisUtil.getSqlSession();
        StudentDao mapper = sqlSession.getMapper(StudentDao.class);
        Student student = new Student();
        student.setName("张三");
        student.setAge(18);
        List<Student> students = mapper.selectStudentIf(student);
        students.forEach(stu -> System.out.println("student : " + stu));
        sqlSession.close();
    }
```

3 动态SQL之 where

if 标签中存在一个比较麻烦的地方：需要在 where 后手工添加 1=1 或者 id>0 的子句。因为，若 where 后的所有的 if 条件均为 false，而 where 后若又没有 1=1 子句，则 SQL 中就会只剩下一个空的 where，SQL 出错。所以，在 where 后，需要添加**永为真**子句 1=1，以防止这种情况的发生。但当数据量很大时，会严重影响查询效率。

使用 where 标签，在有查询条件时，可以自动添加上 where 子句；没有查询条件时，不会添加 where 子句。需要注意的是，第一个 if 标签中的 SQL 片断，可以不包含 and。不过，写上 and 也不错， 系统会将多出的 and 去掉。但其它 if 中 SQL 片断的 and，必须要求写上。否则 SQL 语句将拼接出错 。

```
语法：<where> 其他动态sql </where>
```

```
//动态sql，where使用
List<Student> selectStudentWhere(Student student);
```

```
    <!--
        where:
            <where>
                <if>...</if>
                <if>...</if>
                <if>...</if>
            </where>
    -->
    <select id="selectStudentWhere" resultType="org.example.entity.Student">
        select id,name,email,age from student
        <where>
            <if test="name != null and name != ''">
                name = #{name}
            </if>
            <if test="age > 0">
                and age = #{age}
            </if>
        </where>
    </select>
```

```
    @Test
    public void testSelectStudentWhere(){
        SqlSession sqlSession = MyBatisUtil.getSqlSession();
        StudentDao mapper = sqlSession.getMapper(StudentDao.class);
        Student student = new Student();
        student.setName("张三");
        student.setAge(20);
        List<Student> students = mapper.selectStudentWhere(student);
        students.forEach(stu -> System.out.println("student : " + stu));
        sqlSession.close();
    }
```

4 动态SQL之 foreach

foreach 标签用于实现对于数组与集合的遍历。对其使用，需要注意：

- collection 表示要遍历的集合类型，list，array 等。
- open、close、separator 为对遍历内容的 SQL 拼接。

```
语法：
<foreach collection="集合类型" open="开始的字符" close="结束的字符"
        item="集合中的成员" separator="集合成员之间的分隔符">
    #{item 的值}
</foreach>
```

```
主要用在sql的in语句中。
    学生id是 1001,1002,1003的三个学生
    select * from student where id in (1001,1002,1003)

public List<Student> selectFor(List<Integer> idlist)

List<Integer> list = new ArrayList<>();
```

```
list.add(1001);
list.add(1002);
list.add(1003);
dao.selectFor(list)

<foreach collection="" item="xxx" open="" close="" separator="">
    #{xxx}
</foreach>
```

1. 遍历 List<简单类型>

```
    //foreach 用法1 遍历 List<简单类型>
    List<Student> selectForEachOne(List<Integer> idList);
```

```
    <!-- foreach 使用1，List<Integer> -->
    <select id="selectForEachOne" resultType="org.example.entity.Student">
        select * from student where id in
        <foreach collection="list" item="myId" open="(" close=")" separator=",">
            #{myId}
        </foreach>
        <!--
            collection:表示接口中的方法参数的类型，如果是数组使用array，如果是list集合使用list
            item:自定义的，表示数组和集合成员的变量
            open:循环开始是的字符
            close:循环结束时的字符
            separator:集合成员之间的分隔符
         -->
    </select>
```

```
    @Test
    public void testSelectForEachOne(){
        SqlSession sqlSession = MyBatisUtil.getSqlSession();
        StudentDao mapper = sqlSession.getMapper(StudentDao.class);
        List<Integer> list = new ArrayList<>();
        list.add(1001);
        list.add(1003);
        list.add(1009);
        List<Student> students = mapper.selectForEachOne(list);
        students.forEach(stu -> System.out.println("student : " + stu));
        sqlSession.close();
    }
/*
Created connection 1475491159.
==>  Preparing: select * from student where id in ( ? , ? , ? )
==> Parameters: 1001(Integer), 1003(Integer), 1009(Integer)
<==    Columns: id, name, email, age
<==        Row: 1001, 张三, zs@qq,com, 20
<==        Row: 1003, 王五, ww@163.com, 30
<==        Row: 1009, 刘备, lb@163.com, 55
<==      Total: 3
student : Student{id=1001, name='张三', email='zs@qq,com', age=20}
student : Student{id=1003, name='王五', email='ww@163.com', age=30}
student : Student{id=1009, name='刘备', email='lb@163.com', age=55}
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@57f23557]
*/
```

2. 遍历 List<对象类型>

```
    //foreach 用法2 遍历 List<对象类型>
    List<Student> selectForEachTwo(List<Student> idList);
```

```
    <!-- foreach 使用2，List<Object> -->
    <select id="selectForEachTwo" resultType="org.example.entity.Student">
        select * from student where id in
        <foreach collection="list" item="stu" open="(" close=")" separator=",">
            #{stu.id}
        </foreach>
    </select>
```

```
    @Test
    public void testSelectForEachTwo(){
        SqlSession sqlSession = MyBatisUtil.getSqlSession();
        StudentDao mapper = sqlSession.getMapper(StudentDao.class);
        List<Student> studentList = new ArrayList<>();
        Student s1 = new Student();
        Student s2 = new Student();
        s1.setId(1001);
        s2.setId(1003);
```

```
        studentList.add(s1);
        studentList.add(s2);
        List<Student> students = mapper.selectForEachTwo(studentList);
        students.forEach(stu -> System.out.println("student : " + stu));
        sqlSession.close();
    }
/*
Created connection 1024429571.
==>  Preparing: select * from student where id in ( ? , ? )
==> Parameters: 1001(Integer), 1003(Integer)
<==    Columns: id, name, email, age
<==        Row: 1001, 张三, zs@qq,com, 20
<==        Row: 1003, 王五, ww@163.com, 30
<==      Total: 2
student : Student{id=1001, name='张三', email='zs@qq,com', age=20}
student : Student{id=1003, name='王五', email='ww@163.com', age=30}
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@3d0f8e03]
*/
```

5 动态 SQL 之代码片段

sql 标签用于定义 SQL 片断，以便其它 SQL 标签复用。而其它标签使用该 SQL 片断，需要使用 include 子标签。该 sql 标签可以定义 SQL 语句中的任何部分，所以 include 子标签可以放在动态 SQL 的任何位置。

```
步骤：
    1.先定义 <sql id="自定义名称唯一">  sql语句，  表名，字段等 </sql>
    2.再使用，  <include refid="id的值" />
```

```
    <!-- 定义sql片段 -->
    <sql id="selectStu">
        select id, name, email, age from student
    </sql>

    <select id="selectStudents" resultType="org.example.entity.Student">
        <include refid="selectStu"></include>
    </select>

    <select id="selectStudentIf" resultType="org.example.entity.Student">
        <include refid="selectStu" />
        where id > 0
        <if test="name != null and name != ''">
            name = #{name}
        </if>
        <if test="age > 0">
            and age = #{age}
        </if>
    </select>
```

五、MyBatis 配置文件

1 主配置文件

项目中使用的 mybatis.xml 是主配置文件。

主配置文件特点：

- xml 文件，需要在头部使用约束文件

```
<?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">
```

- 根元素 configuration
- 主要包含内容：定义别名；数据源；mapper 文件

2 dataSource 标签

```
<!--
Mybatis 中访问数据库，可以连接池技术，但它采用的是自己的连接池技术。
在 Mybatis 的 mybatis.xml配置文件中，通过<dataSource type="pooled">
来实现 Mybatis 中连接池的配置
-->
<dataSource type="POOLED">
    <property name="driver" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/springdb"/>
    <property name="username" value="root"/>
    <property name="password" value="111"/>
</dataSource>
```

1. dataSource 类型

   Mybatis 将数据源分为三类：

   - UNPOOLED 不使用连接池的数据源
   - POOLED 使用连接池的数据源
   - JNDI 使用 JNDI 实现的数据源

   其中 UNPOOLED，POOLED 数据源实现了 javax.sq.DataSource 接口，JNDI 和前面两个实现方式不同，了解可以。

2. dataSource 配置

   在 MyBatis.xml 主配置文件，配置 dataSource

   ```
   <dataSource type="POOLED">
       <property name="driver" value="com.mysql.jdbc.Driver"/>
       <property name="url" value="jdbc:mysql://localhost:3306/springdb"/>
       <property name="username" value="root"/>
       <property name="password" value="111"/>
   </dataSource>
   ```

   MyBatis 在初始化时，根据 dataSource 的 type 属性来创建相应类型的的数据源 DataSource，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

3 事务

1. 默认需要手动提交事务

   Mybatis 框架是对 JDBC 的封装，所以 Mybatis 框架的事务控制方式，本身也是用 JDBC 的 Connection 对象的 commit(), rollback()

   Connection 对象的 setAutoCommit() 方法来设置事务提交方式的：自动提交和手工提交

   ```
   <transactionManager type="JDBC"/>
   ```

   该标签用于指定 MyBatis 所使用的事务管理器。MyBatis 支持两种事务管理器类型：JDBC 与 MANAGED。

   1. JDBC：使用 JDBC 的事务管理机制。即通过 Connection 的 commit() 方法提交，通过 rollback() 方法回滚。但默认情况下，MyBatis 将自动提交功能关闭了，改为了手动提交。即程序中需要显式的对事务进行提交或回滚。
   2. MANAGED：由容器来管理事务的整个生命周期（如 Spring 容器）。

2. 自动提交事务

   设置自动提交的方式，factory 的 openSession() 分为有参数和无参数的。

   ```
   SqlSession openSession();
   SqlSession openSession(boolean var1);
   /*
   openSession(true)：创建一个有自动提交功能的 SqlSession
   openSession(false)：创建一个非自动提交功能的 SqlSession，需手动提交
   openSession()：同 openSession(false)
   */
   ```

4 使用数据库属性配置文件

为了方便对数据库连接的管理，DB 连接四要素数据一般都是存放在一个专门的属性文件中的。MyBatis 主配置文件需要从这个属性文件中读取这些数据。步骤如下：

1. 在 classpath 路径下，创建 properties 文件

   在 resources 目录创建 jdbc.properties 文件，文件名称自定义。

   ```
   jdbc.driver=com.mysql.jdbc.Driver
   jdbc.url=jdbc:mysql://localhost:3306/springdb
   jdbc.user=root
   jdbc.passwd=111
   ```

2. 使用 properties 标签

修改主配置文件，文件开始位置加入：

```
<?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>
    <!--指定properties文件的位置，从类路径根开始找文件-->
    <properties resource="jdbc.properties" />
```

3. 使用 key 指定值

```
<dataSource type="POOLED">
    <!--数据库的驱动类名-->
    <property name="driver" value="${jdbc.driver}"/>
    <!--连接数据库的url字符串-->
    <property name="url" value="${jdbc.url}"/>
    <!--访问数据库的用户名-->
    <property name="username" value="${jdbc.user}"/>
    <!--密码-->
    <property name="password" value="${jdbc.passwd}"/>
</dataSource>
```

5 typeAliases (类型别名)

Mybatis 支持默认别名，我们也可以采用自定义别名方式来开发，主要使用在 mybatis.xml 主配置文件定义别名：

```
    <!--mybatis.xml-->
    <!--定义别名-->
    <typeAliases>
        <!--
            第一种方式：
            可以指定一个类型一个自定义别名
            type:自定义类型的全限定名称
            alias:别名（短小，容易记忆的）
        -->
        <typeAlias type="com.xukang.entity.Student" alias="stu" />
        <typeAlias type="com.xukang.vo.ViewStudent" alias="vstu" />

        <!--
          第二种方式
          <package> name是包名，这个包中的所有类，类名就是别名（类名不区分大小写）
        -->
        <package name="com.xukang.entity"/>
        <package name="com.xukang.vo"/>
    </typeAliases>
```

mapper.xml 文件，使用别名表示类型

```
<select id="selectStudents" resultType="stu">
    select id, name, email, age from student
</select>
```

6 mappers (映射器)

```
<mapper resource=" " />
使用相对于类路径的资源,从 classpath 路径查找文件
例如：<mapper resource="org/example/dao/StudentDao.xml" />
```

```
<package name=" "/>
指定包下的所有 Dao 接口
如：<package name="org.example.dao" />
注意：此种方法要求 Dao 接口名称和 mapper 映射文件名称相同，且在同一个目录中。
```

7 小结

```
#jdbc.properties
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/springdb
jdbc.user=root
jdbc.passwd=123456
```

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!--指定properties文件的位置，从类路径根开始找文件-->
    <properties resource="jdbc.properties" />

    <!--settings：控制mybatis全局行为-->
    <settings>
        <!--设置mybatis输出日志-->
        <setting name="logImpl" value="STDOUT_LOGGING"/>
    </settings>

    <!--定义别名-->
    <typeAliases>
        <!--
            第一种方式：
            可以指定一个类型一个自定义别名
            type:自定义类型的全限定名称
            alias:别名（短小，容易记忆的）
        -->
        <typeAlias type="com.bjpowernode.domain.Student" alias="stu" />
        <typeAlias type="com.bjpowernode.vo.ViewStudent" alias="vstu" />

        <!--
          第二种方式
          <package> name是包名，  这个包中的所有类，类名就是别名（类名不区分大小写）
        -->
        <package name="com.bjpowernode.domain"/>
        <package name="com.bjpowernode.vo"/>
    </typeAliases>

    <!--配置插件-->
    <plugins>
        <plugin interceptor="com.github.pagehelper.PageInterceptor" />
    </plugins>

    <environments default="mydev">
        <environment id="mydev">
            <!--
              transactionManager:mybatis提交事务，回顾事务的方式
                 type: 事务的处理的类型
                     1）JDBC ： 表示mybatis底层是调用JDBC中的Connection对象的，commit，  rollback
                     2）MANAGED ： 把mybatis的事务处理委托给其它的容器（一个服务器软件，一个框架（spring））
            -->
            <transactionManager type="JDBC"/>
            <!--
               dataSource:表示数据源，java体系中，规定实现了javax.sql.DataSource接口的都是数据源。
                          数据源表示Connection对象的。

               type:指定数据源的类型
                  1）POOLED: 使用连接池，  mybatis会创建PooledDataSource类
                  2）UPOOLED: 不使用连接池，  在每次执行sql语句，先创建连接，执行sql，在关闭连接
                             mybatis会创建一个UnPooledDataSource，管理Connection对象的使用
                  3）JNDI: java命名和目录服务（windows注册表）
            -->
            <dataSource type="POOLED">
                <!--数据库的驱动类名-->
                <property name="driver" value="${jdbc.driver}"/>
                <!--连接数据库的url字符串-->
                <property name="url" value="${jdbc.url}"/>
                <!--访问数据库的用户名-->
                <property name="username" value="${jdbc.user}"/>
                <!--密码-->
                <property name="password" value="${jdbc.passwd}"/>
            </dataSource>
        </environment>
    </environments>

    <!-- sql mapper(sql映射文件)的位置-->
    <mappers>
        <!--第一种方式：指定多个mapper文件-->
        <!--<mapper resource="com/bjpowernode/dao/StudentDao.xml"/>
        <mapper resource="com/bjpowernode/dao/OrderDao.xml" />-->

        <!--第二种方式：  使用包名
            name: xml文件（mapper文件）所在的包名，这个包中所有xml文件一次都能加载给mybatis
            使用package的要求：
             1. mapper文件名称需要和接口名称一样，  区分大小写的一样
             2. mapper文件和dao接口需要在同一目录
        -->
        <package name="com.bjpowernode.dao"/>
```

```
        <!-- <package name="com.bjpowernode.dao2"/>
         <package name="com.bjpowernode.dao3"/>-->
    </mappers>
</configuration>
```

六、扩展

PageHelper 分页

实现步骤：

1. 在 pom.xml 文件中加入 maven 坐标依赖

```
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper</artifactId>
    <version>5.1.10</version>
</dependency>
```

2. 在 mybatis.xml主配置文件中加入 plugin 配置

```
在<environments>之前加入
<plugins>
    <plugin interceptor="com.github.pagehelper.PageInterceptor" />
</plugins>
```

3. PageHelper 对象

   查询语句之前调用 PageHelper.startPage 静态方法。

   除了 PageHelper.startPage 方法外，还提供了类似用法的 PageHelper.offsetPage 方法。

   在你需要进行分页的 MyBatis 查询方法前调用 PageHelper.startPage 静态方法即可，紧跟在这个方法后的第一个 MyBatis 查询方法会被进行分页。

```
    List<Student> selectStudents();
```

```
    <!-- 定义sql片段 -->
    <sql id="selectStu">
        select id, name, email, age from student
    </sql>

    <select id="selectStudents" resultType="org.example.entity.Student">
        <include refid="selectStu"></include> order by id
    </select>
```

```
     @Test
     public void testPageHelper(){
         SqlSession sqlSession = MyBatisUtil.getSqlSession();
         StudentDao mapper = sqlSession.getMapper(StudentDao.class);
         // 加入PageHelper的方法，分页
         // pageNum: 第几页，从1开始
         // pageSize: 一页中有多少行数据
         PageHelper.startPage(1,2);
         List<Student> students = mapper.selectStudents();
         students.forEach(stu -> System.out.println("student : " + stu));
         sqlSession.close();
     }
/*
Checked out connection 194706439 from pool.
==>  Preparing: SELECT count(0) FROM student
==> Parameters:
<==    Columns: count(0)
<==        Row: 3
<==      Total: 1
==>  Preparing: select id, name, email, age from student order by id LIMIT ?
==> Parameters: 2(Integer)
<==    Columns: id, name, email, age
<==        Row: 1001, 张三, zs@qq,com, 20
<==        Row: 1003, 王五, ww@163.com, 30
<==      Total: 2
student : Student{id=1001, name='张三', email='zs@qq,com', age=20}
student : Student{id=1003, name='王五', email='ww@163.com', age=30}
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@b9afc07]
Returned connection 194706439 to pool.
*/
```