

Mybatis

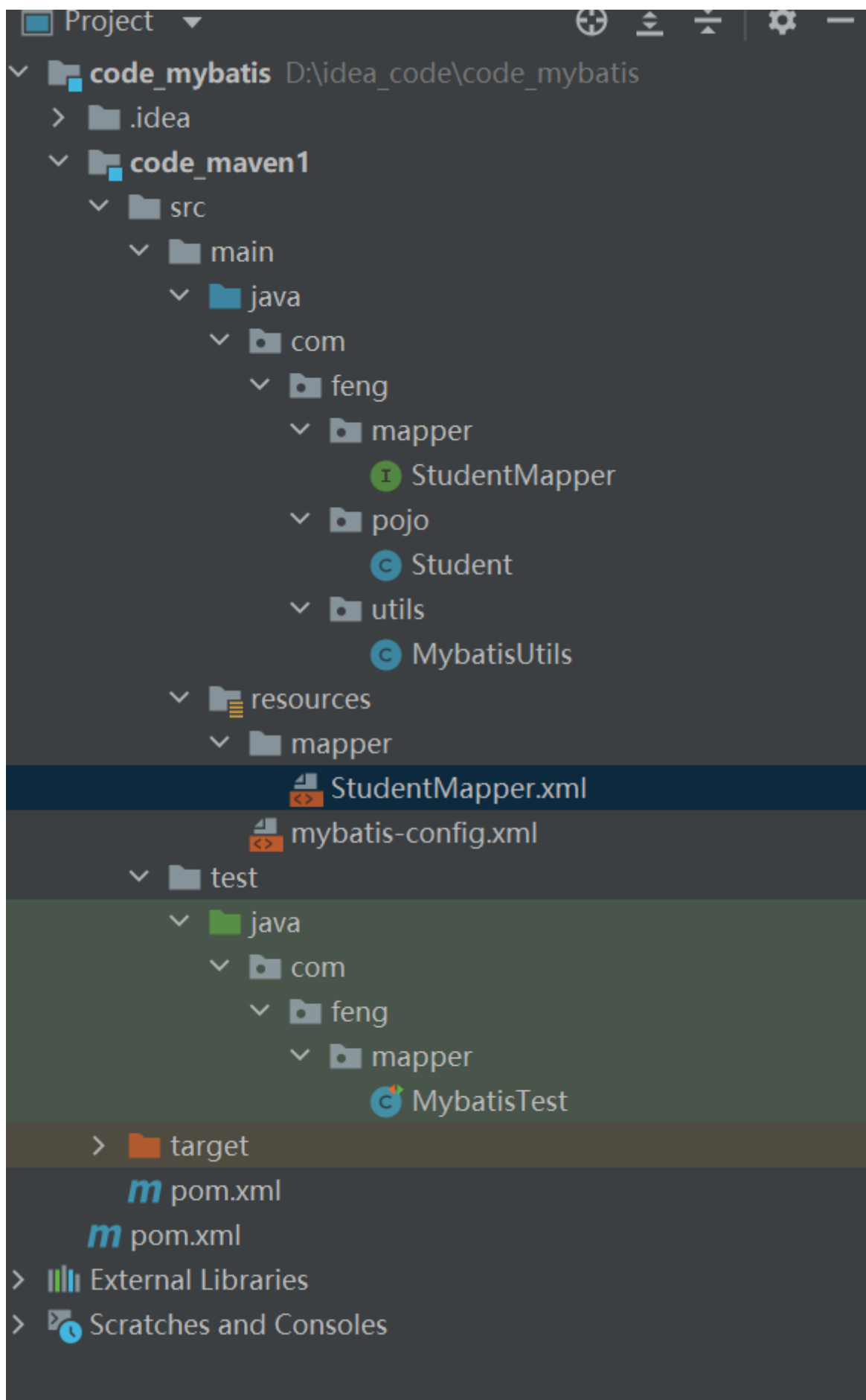
1.导入jar包

```
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>3.5.9</version>
</dependency>
```

2.第一个Mybatis程序 [Mybatis程序](#) [Mybatis文档](#)

2.0.文件的目录

这是两个maven项目融合在一起了



2.1.建立Mybatis环境

添加依赖，在maven仓库里找到Mybatis的dependency代码，还有junit和sql的依赖

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.28</version>
  </dependency>
  <dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.9</version>
  </dependency>
</dependencies>
```

2.2.建立mybatis-config.xml文件

通过官方文档，要先建立mybatis的configuration，这里有个注意的点，就是url的配置，还有mapper的映射

```
<?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>

</configuration>
```

```
jdbc:mysql://localhost:3306/user?
useSSL=true&useUnicode=true&characterEncoding=UTF-
8&serverTimezone=Asia/Shanghai
```

```
<?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>
<!-- 下面可以配置很多的环境，只是default时development-->
  <environments default="development">
    <environment id="development">
      <transactionManager type="JDBC"/>
      <dataSource type="POOLED">
        <property name="driver" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/user?useSSL=true&useUnicode=true&characterEncoding=UTF-8&autoReconnect=true"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
      </dataSource>
    </environment>
  </environments>
  <mappers>
    <mapper resource="mapper/StudentMapper.xml"/>
  </mappers>
</configuration>
```

2.3.建立工具类

工具类见名思意，也就是避免代码复用，也就是可以直接调用的类

```
package com.feng.utils;

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 { //这个是工具类，也就是可以直接用的，避免代码复用
    //提升作用域来使得这些变量都可以使用，工厂可用来生产SqlSession 单对一的生产
    private static SqlSessionFactory sqlSessionFactory;

    static {
        InputStream inputStream = null;
        try {
            String resource = "mybatis-config.xml";
            inputStream = Resources.getResourceAsStream(resource);
            sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static SqlSession getSqlSession(){
        SqlSession sqlSession = sqlSessionFactory.openSession();
        return sqlSession;
    }
}
```

2.4.实体类

通过之前建立表的Student，建立相关的实体类，也就是Student类

```
public class Student {  
    private int id;  
    private String name;  
    private String pwd;  
}
```

2.5.接口类

通过实体类Student来建立相关的接口类

```
package com.feng.mapper;  
  
import com.feng.pojo.Student;  
  
import java.util.List;  
  
public interface StudentMapper {  
    List<Student> getStudentList();  
}
```

2.5.配置相关的接口.xml文件来实现sql语句的功能

```
<?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="接口类">  
  
</mapper>
```

这里注意这里的路径一定是完整路径，而非名字，就是namespace和resultType要完整路径

```
mybatis-config.xml x MybatisUtils.java x MybatisTest.java x Student.java
<?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.feng.mapper.StudentMapper">
  <select id="getStudentList" resultType="com.feng.pojo.Student">
    select * from student
    -- 这里面指定sql语句
  </select>
</mapper>
```

2.6.设置相关的过滤器

将过滤resources的功能恢复，这个在pom.xml中配置

```
<!--设置资源过滤-->
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <includes>
        <include>**/*.properties</include>
        <include>**/*.xml</include>
      </includes>
      <filtering>true</filtering>
    </resource>
    <resource>
      <directory>src/main/java</directory>
      <includes>
        <include>**/*.properties</include>
        <include>**/*.xml</include>
      </includes>
      <filtering>true</filtering>
    </resource>
  </resources>
</build>
```

2.7.测试 @Test

这里有两种方式来实现sql语句的功能，注意这里的步骤

```

public class MybatisTest {
    @Test
    public void test(){
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        //方式一
        StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
        List<Student> studentList = mapper.getStudentList();
        //
        List<Student> selectList = sqlSession.selectList("com.feng.mapper.StudentMapper.getStudentList");
        for(Student i:studentList){
            System.out.println(i);
        }
        sqlSession.close();
        //方式二
    }
}

```

3.实现增删改查

这里注意要提交事务，也就是sqlSession.commit()，这个操作要执行

0.开启mybatis的自动提交

commit自动调用

```

public static SqlSession getSqlSession(){
    return sqlSessionFactory.openSession(b: true);
}

```

3.1 增加用户

1.设置接口方法

```
void InsertStudent(Student student);
```

2.配置sql.xml文件

配置该文件来实现注册相应的操作

```

<insert id="InsertStudent" parameterType="com.feng.pojo.Student">
    insert into student(id,name,pwd) values ({id},{name},{pwd})
</insert>

```

3.配置相应的方法

```

@Test
public void test4(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
    mapper.InsertStudent(new Student( id: 4, name: "哇哇", pwd: "312643"));
    sqlSession.commit();
    sqlSession.close();
}

```

3.2删除用户

过程都一样，这里只把sql.xml配置的文件放出

```
</insert>
<delete id="DeleteStudent" parameterType="int">
    delete from student where id=#{id};
</delete>
```

3.3修改用户信息

```
<update id="UpdateStudent" parameterType="com.feng.pojo.Student">
    update student set pwd=#{pwd} where name=#{name}
</update>
```

3.4查找用户

```
<select id="getStudentById" parameterType="int" resultType="com.feng.pojo.Student">
    select * from student where id=#{id}
</select>
```

4.Map的妙用

通过map的使用，就可以不用再new出来一个新的对象，万一新的对象有很多的参数的话，new一个对象会显得很麻烦，map可以存入我们需要的参数并返回参数，这样参数的名字也可以修改

4.1配置接口方法

```
void addStudent(Map<String,Object> map);
```

4.2配置sql.xml文件的配置

这里通过map的使用成功实现了关键词改名的可能

```
</delete>
<insert id="addStudent" parameterType="map">
    insert into student(id,name,pwd) values (#{StudentId},#{StudentName},#{StudentPwd});
</insert>
```

4.3配置测试类


```

@Test
public void test6(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
    HashMap<String, Object> stringObjectHashMap = new HashMap<>();
    stringObjectHashMap.put("StudentId",4);
    stringObjectHashMap.put("StudentName","王五");
    stringObjectHashMap.put("StudentPwd","317458");
    mapper.addStudent(stringObjectHashMap);
    sqlSession.commit();
    sqlSession.close();
}

```

5.模糊查询

所谓模糊查询就是like关键词的使用，like的语法是“%中间匹配的对象%”

5.1配置接口方法

```
Student getStudentById(String value);
```

5.2配置sql.xml文件

方式一：

```

<select id="getStudentById" parameterType="String" resultType="student">
    select * from student where name like "%#{value}%";
</select>

```

方式二：

```

</select>
<select id="getStudentById" parameterType="String" resultType="student">
    select * from student where name like #{value};
</select>

```

5.3配置测试类

方式一：

```

@Test
public void test2(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
    Student studentById = mapper.getStudentById(value: "张");
    System.out.println(studentById);
    sqlSession.close();
}

```

方式二：

```

}
@Test
public void test2(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
    Student studentById = mapper.getStudentById( value: "%张%");
    System.out.println(studentById);
    sqlSession.close();
}

```

6.config文件的参数

6.1参数

- configuration (配置)
 - properties (属性)
 - settings (设置)
 - typeAliases (类型别名)
 - typeHandlers (类型处理器)
 - objectFactory (对象工厂)
 - plugins (插件)
 - environments (环境配置)
 - environment (环境变量)
 - transactionManager (事务管理器)
 - dataSource (数据源)
 - databaseIdProvider (数据库厂商标识)
 - mappers (映射器)

这里注意xml文件中这些参数的位置

```

❌ The content of element type "configuration" must match "(properties?,settings?,typeAliases?,typeHandlers?,objectFactory?,objectWrapperFactory?,reflectorFactory?,plugins?,environments?,databaseIdProvider?,mappers?)".
❌ The content of element type "configuration" must match "(properties?,settings?,typeAliases?,typeHandlers?,objectFactory?,objectWrapperFactory?,reflectorFactory?,plugins?,environments?,databaseIdProvider?,mappers?)".

```

6.2properties

1.配置properties文件

```

mybatisTest.java  db.properties  StudentMapper.java  StudentMapper.xml  mybatis-config.xml  MybatisUtils.java
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/user?useSSL=true&useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai
username=root
password=root

```

2.在config.xml文件中配置properties的属性

在properties中还可以配置property的新属性，注意在单个配置中直接一行结尾

```
<properties resource="properties/db.properties">
    <property name="username" value="root"/>
</properties>
```

```
<transactionManager type="JDBC"/>
<dataSource type="POOLED">
    <property name="driver" value="${driver}"/>
    <property name="url" value="${url}"/>
    <property name="username" value="${username}"/>
    <property name="password" value="${password}"/>
</dataSource>
```

6.3起别名---typeAliases

1.在config.xml文件中配置别名

```
<typeAliases>
    <typeAlias type="com.feng.pojo.Student" alias="student"/>
</typeAliases>
```

2.在sql.xml中的使用

```
<select id="getStudentById" parameterType="String" resultType="student">
    select * from student where name like #{value};
</select>
```

7.ResultMap的使用

7.1前提：引入as的关键词的使用

1.在实体类修改

```
public class Student1 {
    private int id;
    private String name;
    private String password;
```

2.在sql.xml文件中修改

将pwd用as关键词修改成password迎合student类的修改

```
<select id="getStudentList" resultType="com.feng.pojo.Student1">
    select id,name,pwd as password from student
    这里面指定sql语句
</select>
```

7.2ResultMap的使用

ResultMap是映射集

1.在sql.xml中配置ResultMap

注意这里的column是database中的字段，property是实体中的属性，这个操作使得pwd直接映射成实体类的password

```
<resultMap id="Student1Map" type="com.feng.pojo.Student1">
    <result column="pwd" property="password"/>
</resultMap>
<select id="getStudentList" resultMap="Student1Map">
    select id,name,pwd from student
    这里面指定sql语句
</select>
```

2.实体类的改变

```
public class Student1 {
    private int id;
    private String name;
    private String password;
```

3.查询结果的改变

```
Loading class `com.mysql.jdbc.Driver'. This is
Student1{id=1, name='张三', password='125450'}
Student1{id=2, name='李四', password='12345'}|
Student1{id=3, name='王五', password='12345'}
Student1{id=4, name='王五', password='317458'}
```

8.日志工厂

1.log4j

其中有STDOUT_LOGGING和log4j

2.STDOUT_LOGGING

在config.xml中配置，不过得注意顺序，看前面提到的

```
</properties>
<settings>
    <setting name="logImpl" value="STDOUT_LOGGING"/>
</settings>
<typeAliases>
```

效果

```
Loading class com.mysql.jdbc.Driver : This is deprecated. The new driver class is com.mysql.cj.jdbc.Driver
Created connection 1107217291.
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@41fecb8b]
==> Preparing: select * from student -- 上面的这个句子等价于select id,name,pwd from student -- 这里面指定sql语句
==> Parameters:
<==      Columns: id, name, pwd
<==      Row: 1, 张三, 125450
<==      Row: 2, 李四, 12345
<==      Row: 3, 王五, 12345
<==      Row: 4, 哇哇, 312643
<==      Total: 4
Student1{id=1, name='张三', password='125450'}
Student1{id=2, name='李四', password='12345'}
Student1{id=3, name='王五', password='12345'}
Student1{id=4, name='哇哇', password='312643'}
Resetting autocommit to true on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@41fecb8b]
Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@41fecb8b]
Returned connection 1107217291 to pool.
```

3.LOG4J

和上面的一样，在config.xml中配置，一样是注意顺序，注意这个properties文件的位置

这个是log4j的配置文件，log4j.properties

```
#将等级为DEBUG的日志信息输出到console和file这两个目的地，console和file的定义在下面的代码
log4j.rootLogger=DEBUG,console,file
```

```
#控制台输出的相关设置
```

```
log4j.appender.console = org.apache.log4j.ConsoleAppender
log4j.appender.console.Target = System.out
log4j.appender.console.Threshold=DEBUG
log4j.appender.console.layout = org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=[%c]-%m%n
```

```
#文件输出的相关设置
```

```
log4j.appender.file = org.apache.log4j.RollingFileAppender
log4j.appender.file.File=./log/shun.log
log4j.appender.file.MaxFileSize=10mb
log4j.appender.file.Threshold=DEBUG
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=[%p] [%d{yy-MM-dd}] [%c] %m%n
```

```
#日志输出级别
```

```
log4j.logger.org.mybatis=DEBUG
```

```
log4j.logger.java.sql=DEBUG
log4j.logger.java.sql.Statement=DEBUG
log4j.logger.java.sql.ResultSet=DEBUG
log4j.logger.java.sql.PreparedStatement=DEBUG
```

config.xml中的配置

```
<settings>
    <setting name="logImpl" value="LOG4J"/>
</settings>
```

函数体

```
public void test0() { ... }

@Test
public void test7() {
    static Logger logger = Logger.getLogger(StudentMapper.class);
    logger.info("info:进入了");
    logger.warn("warn:进入了");
    logger.trace("trace:进入了");
    logger.fatal("fatal:进入了");
    logger.error("error:错误了");
    logger.debug("debug:测试中");
}
```

输出文件的日志，也是效果

```
[INFO][22-03-08][com.feng.mapper.StudentMapper]info:进入了
[ERROR][22-03-08][com.feng.mapper.StudentMapper]error:错误了
[DEBUG][22-03-08][com.feng.mapper.StudentMapper]debug:测试中
```

9.Limit实现分页

1.语法

```
select * from table limit startIndex,pageSize;
```

2.接口类配置

```
Student selectStudentByLimit(Map<String, Object> map);
```

3.sql.xml配置

```
<select id="selectStudentByLimit" resultType="student" parameterType="map">
    select * from student limit #{startIndex},#{pageSize};
</select>
```

4.函数体的实现

```
}
@Test
public void test8(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
    HashMap<String, Object> stringObjectHashMap = new HashMap<>();
    stringObjectHashMap.put("startIndex",0);
    stringObjectHashMap.put("pageSize",2);
    List<Student> students = mapper.selectStudentByLimit(stringObjectHashMap);
    for (Student student : students) {
        System.out.println(student);
    }
    sqlSession.close();
}
```

这些操作和之前的一样

10.注解sql

1.语法

1.1先在接口类中配置sql语句

```
@Select("select * from student")
Student selectStudents();
```

1.2再在config.xml文件中配置接口类对象，也就是对什么对象操作，对象mapper的配置

```
<mappers>
    这个有很多的方法，但是我的这个文件布局就只能是这样的resource-->
    <mapper resource="mapper/StudentMapper.xml"/>
    <mapper class="com.feng.pojo.Student"/>
</mappers>
```

1.3测试类中的配置

```

@Test
public void test9(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
    List<Student> students = mapper.selectStudents();
    for (Student student : students) {
        System.out.println(student);
    }
    sqlSession.close();
}

```

2.实现增删改查

在这个方法下只需要修改接口类中的语句，config.xml不需要修改

1.增

```

@Insert("insert into student(id,name,pwd) values(#{StudentId},#{StudentName},#{StudentPwd})")
void insertStudent(Map<String,Object> map);

```

2.删

```

@Delete("delete from student where id=#{id}")
void deleteStudent(int id);

```

3.改

```

@Update("update student set name=#{name} where id=#{id}")
void updateStudent(Map<String,Object> map);

```

4.查

```


@Select("select * from student")
List<Student> selectStudents();

```

补充一个知识点

@Param的用法

```

public interface TeacherMapper {
     @Select("select * from user.teacher where id=#{tid}")
    Teacher getTeacher(@Param("tid") int id);
}

```

这里的Param中的参数为主要参数

注意

利用注解Sql语句来实现功能时，config.xml中要同时注册resources和class文件

11.关联和集合

通过老师和学生的关联来体现这节知识点

1.association--关联 多对一

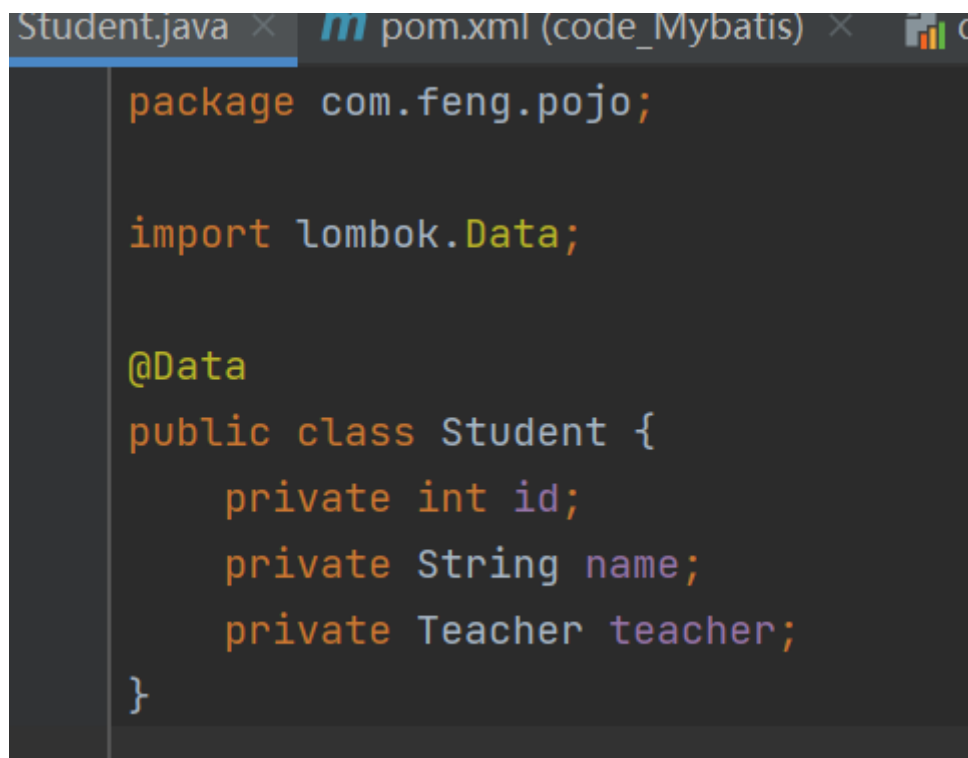
这里有两种方法

方式一：子查询

这里的语句规范就是这样的，下面的getTeacher方法是上一个查询自己调用的，上面不是有association中select的名字吗，下面的id与之对应，这里要十分注意，这是在sql.xml中配置的，这里通过子查询将查询结果返回到大查询中，并方便下次查询

```
<?xml version="1.0" encoding="UTF-8"?>
<mapper namespace="com.feng.Mapper.StudentMapper">
  <select id="getStudent" resultMap="StudentTeacher">
    select * from student
  </select>
  <resultMap id="StudentTeacher" type="com.feng.pojo.Student">
    <association property="teacher" column="tid" javaType="com.feng.pojo.Teacher" select="getTeacher"/>
  </resultMap>
  <select id="getTeacher" resultType="com.feng.pojo.Teacher">
    select * from teacherSS
  </select>
</mapper>
```

附上类的截图



```
package com.feng.pojo;

import lombok.Data;

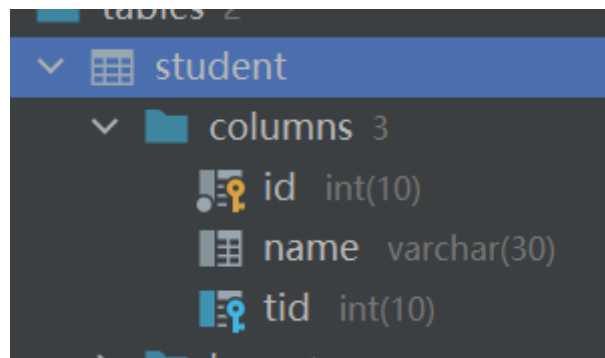
@Data
public class Student {
    private int id;
    private String name;
    private Teacher teacher;
}
```

```

@Data
public class Teacher {
    private int id;
    private String name;
}

```

还有数据库的截图



方式二：嵌套查询

这个方法符合sql的基本语法，是比较倾向推荐的，下面附上截图

resultmap就相当于给列取一个别名让别人来进行访问，注意下面的column的tname是和上面的改了的列名是一样的，注意select下的t.id teaid,t.name tname，这里不是必须改名字，但是后面的result中必须出现他们的id和name，还有注意select执行的时间顺序，先是判断where语句，后面才是查找语句，所以在select处改名字是不会影响where的判断的，这里的association就是一个集合将teacher中的元素封装起来，再放回

```

<select id="getStudent1" resultMap="StudentTeacher1">
    select s.id sid,s.name sname,t.id teaid,t.name tname
    from student s,teacher t
    where s.tid = t.id
</select>

<resultMap id="StudentTeacher1" type="com.feng.pojo.Student">
    <result property="id" column="sid"/>
    <result property="name" column="sname"/>
    <association property="teacher" javaType="com.feng.pojo.Teacher">
        <result property="id" column="teaid"/>
        <result property="name" column="tname"/>
    </association>
</resultMap>

```

2.collection--集合 一对多

嵌套查询

这点和上面的嵌套查询是一样的操作，但是注意这里的ofType的属性，还有这里的collection是关键

```
<select id="getTeacher" resultType="com.feng.pojo.Teacher1">
    select * from teacher
</select>
<select id="getTeacher1" resultMap="TeacherStudent">
    select s.id sid,s.name sname,t.name tname,t.id tid
    from student s,teacher t
    where s.tid=t.id
</select>
<resultMap id="TeacherStudent" type="com.feng.pojo.Teacher1">
    <result property="id" column="tid"/>
    <result property="name" column="tname"/>
    <collection property="students" ofType="com.feng.pojo.Student1">
        <result property="id" column="sid"/>
        <result property="name" column="sname"/>
        <result property="tid" column="tid"/>
    </collection>
</resultMap>
```

这里附上类的截图

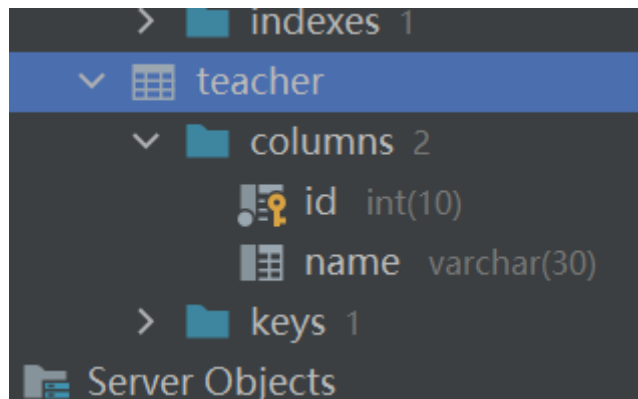
```
import java.util.List;

@Data
public class Teacher1 {
    private int id;
    private String name;
    private List<Student1> students;
}
```

```

    @Data
    public class Student1 {
        private int id;
        private String name;
        private int tid;
    }
```

数据库的截图



3.多对多查询

多对多查询中，需要一个表做中间表链接操作，Inner join，left join，right join

[MyBatis多对多关联查询](#)

综上所述，选择嵌套查询的效果最好，格式也是固定的

javaType和ofType的区别

javaType用来指定对象所属的java数据类型，也就是private Listposts 的ArrayList类型，ofType用来指定对象的所属javaBean类,也就是尖括号的泛型private Listposts

ofType可以指出collection的list中的类型， javaType可以指出当前使用的java的类型

12.动态sql语句

0.这个用例的实体类和数据库的数据类型

```

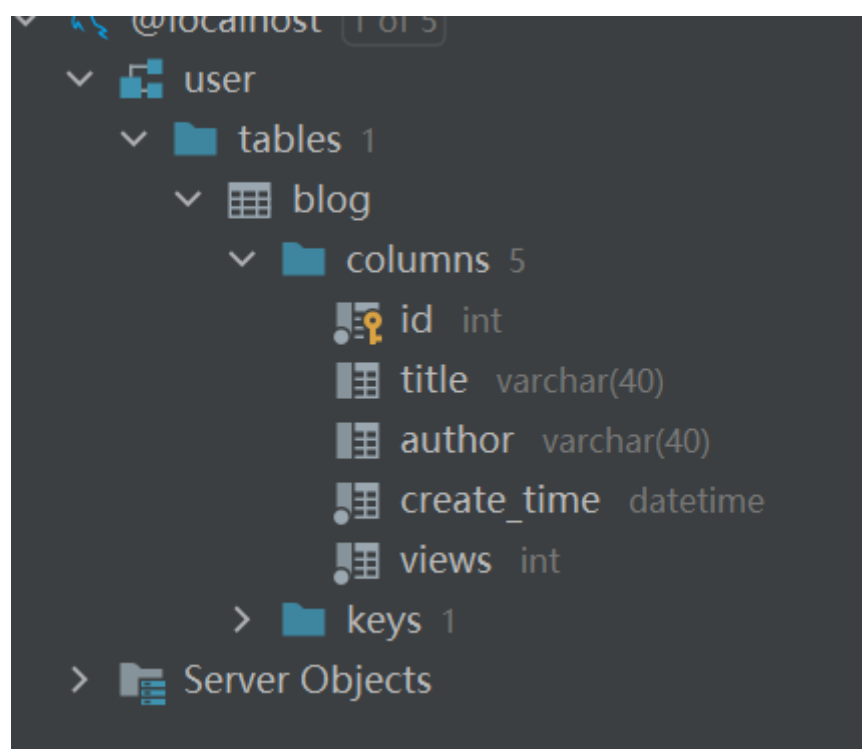
package com.feng.pojo;

import lombok.Data;

import java.util.Date;

@Data
public class Blog {
    private int id;
    private String title;
    private String author;
    private Date createTime;
    private int views;
}

```



1.if

下面的这个语句自带一个where标签，所以后面的只需要加入and就行，但是实际开发中是没有这样的

```

<select id="getBlogs" resultType="com.feng.pojo.Blog" parameterType="map">
    select * from blog where 1 = 1
<if test="title!=null">
    and title=#{title}
</if>
<if test="views!=null">
    and views>#{views}
</if>
</select>

```

所以调用下面的标签

```

<select id="getBlogs" resultType="com.feng.pojo.Blog" parameterType="map">
    select * from blog
    <where>
        <if test="title!=null">
            and title=#{title}
        </if>
        <if test="views!=null">
            and views>#{views}
        </if>
    </where>
</select>

```

这里的标签可以实现当前面第一个有where的时候就会加入and，没有的时候不会加入and

2.choose when otherwise

这个标签就相当于java中的switch语句，这要你满足了when中的条件，就可以通过这个when条件来查询记录，最后的otherwise是之前的when条件不满足的时候就会调用otherwise中的标签

```

<select id="getBlogsByChoose" parameterType="map" resultType="com.feng.pojo.Blog">
    select * from blog
    <where>
        <choose>
            <when test="title!=null">
                title = #{title}
            </when>
            <when test="views!=null">
                and views>#{views}
            </when>
            <otherwise>
                and id = #{id}
            </otherwise>
        </choose>
    </where>
</select>

```

测试类的写法

```

@Test
public void test3(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
    HashMap<String, Object> stringObjectHashMap = new HashMap<>();
    //stringObjectHashMap.put("title","java真好玩");
    //stringObjectHashMap.put("views",100);
    stringObjectHashMap.put("id",123);
    List<Blog> blogsByChoose = mapper.getBlogsByChoose(stringObjectHashMap);
    for (Blog blog : blogsByChoose) {
        System.out.println(blog);
    }
    sqlSession.close();
}

```

3.set

set标签可以实现末尾去","和动态插入set关键词，可以和其他的标签实现组合功能

```

<update id="updateBlog" parameterType="map">
    update blog
    <set>
        <if test="title!=null">title=#{title},</if>
        <if test="views!=null">views=#{views},</if>
    </set>
    <where>
        <if test="author!=null">and author = #{author} </if>
        <if test="id!=null">and id = #{id}</if>
    </where>
</update>

```

测试类的写法

```

@Test
public void test4(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
    HashMap<String, Object> stringObjectHashMap = new HashMap<>();
    stringObjectHashMap.put("title","Mybatis真好玩");
    stringObjectHashMap.put("author","feng");
    stringObjectHashMap.put("views",98);
    stringObjectHashMap.put("id",123);
    mapper.updateBlog(stringObjectHashMap);
    sqlSession.commit();
    sqlSession.close();
}

```

4.sql语句

sql语句可以实现代码复用的功能，后面要调用的时候要调用include标签来调用sql语句

```
</insert>
<sql id="if-title-views">
    <if test="title!=null">
        and title=#{title}
    </if>
    <if test="views!=null">
        and views>#{views}
    </if>
</sql>
<select id="getBlogsByIf" resultType="com.feng.pojo.Blog" parameterType="map">
    select * from blog
    <where>
        <include refid="if-title-views"></include>
    </where>
</select>
```

5.foreach语句

foreach语句就相当于是sql语句中的in的关键词

```
</update>
<select id="getBlogsByForEach" parameterType="map" resultType="com.feng.pojo.Blog">
    select * from blog
    <where>
        <foreach collection="ids" item="id" open="(" separator="or" close=")">
            id = #{id}
        </foreach>
    </where>
</select>
```

测试类的写法

这里注意ArrayList的细节实现，foreach的写法要ArrayList

```
@Test
public void test5(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
    HashMap<String, Object> stringObjectHashMap = new HashMap<>();
    ArrayList<Integer> integers = new ArrayList<>();
    integers.add(123);
    integers.add(124);
    stringObjectHashMap.put("ids", integers);
    List<Blog> blogsByForEach = mapper.getBlogsByForEach(stringObjectHashMap);
    for (Blog byForEach : blogsByForEach) {
        System.out.println(byForEach);
    }
    sqlSession.close();
}
```


6.trim标签

这个语句可以自定义一个where的标签，也就是标签的自动添加和去除的功能

如果 *where* 元素与你期望的不太一样，你也可以通过自定义 *trim* 元素来定制 *where* 元素的功能。比如，和 *where* 元素等价的自定义 *trim* 元素为：

```
<trim prefix="WHERE" prefixOverrides="AND |OR ">
...
</trim>
```

prefixOverrides 属性会忽略通过管道符分隔的文本序列（注意此例中的空格是必要的）。上述例子会移除所有 *prefixOverrides* 属性中指定的内容，并且插入 *prefix* 属性中指定的内容。

同理，也可以自定义一个标签的添加set关键词和去掉最后一个','

这个例子中，*set* 元素会动态地在行首插入 SET 关键字，并会删掉额外的逗号（这些逗号是在使用条件语句给列赋值时引入的）。

来看看与 *set* 元素等价的自定义 *trim* 元素吧：

```
<trim prefix="SET" suffixOverrides=",">
...
</trim>
```

注意，我们覆盖了后缀值设置，并且自定义了前缀值。

13.缓存

1.简介

缓存就是将查询结果放在一个地方，访问的时候就可以直接访问缓存，缓存可以有效避免多用户高并发的问题，mybatis系统中内置了两级缓存，**一级缓存**和**二级缓存**，默认的情况下，只有一级缓存开启着，二级缓存需要手动开启和配置，他是基于namespace级别的缓存，也就是一个namespace，对应一个二级缓存地址，为了提高拓展性，mybatis定义了缓存接口cache，我们可以通过实现Cache接口来定义二级缓存

2.一级缓存

一级缓存是默认开启的，下面这个结果是true的，是通过一级缓存缓存下来的，再次查询是相同的

```
@Test
public void test6(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
    Blog blogs = mapper.getBlogs( id: 123);
    Blog blogs1 = mapper.getBlogs( id: 123);
    System.out.println(blogs);
    System.out.println(blogs1);
    System.out.println(blogs1==blogs);
    sqlSession.close();
}
```

注意下面的情况是会使sqlsession再次链接，导致缓存失去，一级缓存消失，导致结果为假

默认情况下，只启用了本地的会话缓存，它仅仅对一个会话中的数据进行缓存。要启用全局的二级缓存，只需要在你的 SQL 映射文件中添加一行：

```
<cache/>
```

基本上就是这样。这个简单语句的效果如下：

- 映射语句文件中的所有 select 语句的结果将会被缓存。
- 映射语句文件中的所有 insert、update 和 delete 语句会刷新缓存。
- 缓存会使用最近最少使用算法（LRU, Least Recently Used）算法来清除不需要的缓存。
- 缓存不会定时进行刷新（也就是说，没有刷新间隔）。
- 缓存会保存列表或对象（无论查询方法返回哪种）的 1024 个引用。
- 缓存会被视为读/写缓存，这意味着获取到的对象并不是共享的，可以安全地被调用者修改，而不干扰其他调用者或线程所做的潜在修改。

亲自清除缓存，需要java语句，sqlsession.clearCache()，下面的结果为假，因为Cache已经被清除了

```

    }
    @Test
    public void test6(){
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
        Blog blogs = mapper.getBlogs(id: 123);
        sqlSession.clearCache();
        Blog blogs1 = mapper.getBlogs(id: 123);
        System.out.println(blogs);
        System.out.println(blogs1);
        System.out.println(blogs1==blogs);
        sqlSession.close();
    }
}

```

3.二级缓存

在二级缓存中缓存只是关于namespace,一个namespace管理一个二级缓存,实体类还要序列化

1.在config.xml中设置settings, 在settings中设置cacheEnabled为true, 二级缓存的来源是来自死去的一级缓存

cacheEnabled	全局性地开启或关闭所有映射器配置文件中已配置的任何缓存。	true false	true
--------------	------------------------------	--------------	------

```

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

```

2.在mapper的设置

```

"http://mybatis.org/dtd/mybatis-3-mapper.dtd"
<mapper namespace="com.feng.Mapper.BlogMapper">
    <cache/>
    <insert id="addBlog" parameterType="com.feng.pojo

```

在该标签下可以设置cache的属性

这些属性可以通过 cache 元素的属性来修改。比如：

```

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

```

在有些标签下, 还可以设置useCache的真值

```
<select id="getBlogsByIf" resultType="com.feng.pojo.Blog" parameterType="map" useCache="false">
    select * from blog
    <where>
        <include refid="if-title-views"></include>
    </where>
</select>
```

实体类的设置，序列化，implements继承接口类

```
@Data
public class Blog implements Serializable {
    private int id;
    private String title;
    private String author;
    private Date createTime;
    private int views;
}
```

3.自定义缓存Ehcache

在pom.xml中导入Ehcache

```
<dependency>
    <groupId>org.mybatis.caches</groupId>
    <artifactId>mybatis-ehcache</artifactId>
    <version>1.2.1</version>
</dependency>
```

在mapper.xml中配置cache

```
<cache type="org.mybatis.caches.ehcache.EhcacheCache"/>
```

resources中ehcache.xml中的配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
    updateCheck="false">

    <diskStore path="./tmpdir/Tmp_EhCache"/>

    <defaultCache
        eternal="false"
        maxElementsInMemory="10000"
        overflowToDisk="false"
        diskPersistent="false"
        timeToIdleSeconds="1800"
        timeToLiveSeconds="259200"
```

```
memoryStoreEvictionPolicy="LRU"/>

<cache
  name="cloud_user"
  eternal="false"
  maxElementsInMemory="5000"
  overflowToDisk="false"
  diskPersistent="false"
  timeToIdleSeconds="1800"
  timeToLiveSeconds="1800"
  memoryStoreEvictionPolicy="LRU"/>
</ehcache>
```

慢SQL 1S 1000S

面试高频

- Mysql引擎
- InnoDB底层原理
- 索引
- 索引优化!

14.JDBC

0.导入jar包

导入mysql-connector-java.jar包，这个可以从maven的仓库中获取

1.第一个JDBC程序

resultset中有很多的内置函数，就相当是LinkedList的对象一样，上网可搜

```

public class jdbc {
    public static void main(String[] args) throws Exception {
        String url = "jdbc:mysql://localhost:3306/user?useUnicode=true&characterEncoding=utf-8";
        String username = "root";
        String password = "root";
        Class.forName("com.mysql.jdbc.Driver");
        Connection connection = DriverManager.getConnection(url, username, password);
        Statement statement = connection.createStatement();
        String sql = "select * from users";

        PreparedStatement preparedStatement = connection.prepareStatement(sql);
        String sql = "insert into users(id, name, password, email, birthday) values (?, ?, ?, ?, ?)";
        PreparedStatement preparedStatement = connection.prepareStatement(sql);
        preparedStatement.setObject(); //这个下标是从1开始的, 注意这个细节
        ResultSet resultSet = statement.executeQuery(sql);
        while(resultSet.next()){
            System.out.println("id="+resultSet.getInt( columnLabel: "id"));
            System.out.println("name="+resultSet.getString( columnLabel: "name"));
            System.out.println("password="+resultSet.getString( columnLabel: "password"));
            System.out.println("email="+resultSet.getString( columnLabel: "email"));
            System.out.println("birthday"+resultSet.getObject( columnLabel: "birthday"));
        }
        resultSet.close();
        statement.close();
        connection.close();
    }
}

```

这个是事务回滚 try catch 的实现
这个是事务提交

2.增删改查

0.注意这里的preparedStatement

这里的preparedStatement是预编译的语句，下面的代码都是预编译来实现的

```
PreparedStatement preparedStatement = connection.prepareStatement(sql);
```

1.查

```

String statement = "select * from biao where id = ? and name = ?";
preparedStatement.setObject(1,id);
preparedStatement.setObject(2,name);
preparedStatement.excuteQuery();

```

这里的问号代表占位符，是从1开始的，该测试类是需要传入参数id和name的，从而来实现查询的功能，这里的preparedStatement是预编译产生的，这里的excute是执行sql语句的

2.删

```

String statement = "delete from biao where id = ? and name = ?";
preparedStatement.setObject(1,id);
preparedStatement.setObject(2,name);
preparedStatement.excuteUpdate();

```

这里的原理也是和上面的是一样的，注意connection.commit()，记得事务提交

3.改

```

String statement = "update biao set name = ? ";
preparedStatement.setObject(1,name);
preparedStatement.excuteUpdate();

```

这里的核心也是和上面的一样，注意connection.commit()，记得事务提交

4.增

```
String statement = "insert into biao(id,name) values(?,?)";
preparedStatement.setObject(1,id);
preparedStatement.setObject(2,name);
preparedStatement.executeUpdate();
```

这里的核心也是和上面的一样，注意connection.commit()，记得事务提交

5.拓展

```
connection.rollback();    //事务回滚
connection.commit();      //事务提交
connection.setAutoCommit(); //事务自动提交
```

6.createStatement版本

是一样的，只是sql语句是通过字符串拼接起来的，这里里面的参数通过传参来进行查询和拼接，下面放下链接，来体验一下

[createStatement和PreparedStatement来实现增删查改操作的实例](#)