MyBaits

MyBaits初步

mybatis

最近更新: 07 十月 2020 | 版本: 3.5.6

1.1简介

什么是 MyBatis?

- MyBatis 是一款优秀的**持久层框架**，
- 它支持自定义 SQL、存储过程以及高级映射。
- MyBatis免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作。MyBatis可以通过简单的 XML 或注解来配置和映射原始类型、接口和 Java POJO（Plain Old Java Objects, 普通老式 Java 对象）为数据库中的记录。
- 总的来说，简化了对数据库的操作！！！！！！ ###########

如何获取MyBatis

- maven

```
<!-- https://mvnrepository.com/artifact/org.mybatis/mybatis -->
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.6</version>
</dependency>
```

- 中文文档：https://mybatis.org/mybatis-3/zh/index.html
- Github: 去github找吧，我这里就不放了

1.2持久化

数据持久化

- 持久化就是将程序的数据在持久状态和瞬时状态转化的过程
- 内存：**断电即失**
- 数据库(jdbc)，io文件持久化
- 生活中：（持久化）冷藏、罐头。。。。。。。

为什么需要持久化?

- 有一些对象，不能让它丢掉
- 内存太贵了

1.3持久层

Dao层，service层，contrller层

- 完成持久化工作的代码块
- 层界限明显

1.4为什么需要Mybatis

- 方便，简化
- 传统的jdbc太复杂了
    - 传统jdbc六步骤：1加载驱动，2获取链接，3获取操作数据库的对象
    - 4执行sql，5处理查询结果集(如果有）6释放资源
- 不用MyBatis也可，更容易上手，但是没技术啊

1.5特点

- 简单易学：本身就很小且简单。没有任何第三方依赖，最简单安装只要两个jar文件+配置几个sql映射文件易于学习，易于使用，通过文档和源代码，可以比较完全的掌握它的设计思路和实现。
- 灵活：mybatis不会对应用程序或者数据库的现有设计强加任何影响。 sql写在xml里，便于统一管理和优化。通过sql语句可以满足操作数据库的所有需求。
- 解除sql与程序代码的耦合：通过提供DAO层，将业务逻辑和数据访问逻辑分离，使系统的设计更清晰，更易维护，更易单元测试。**sql和代码的分离**，提高了可维护性。
- 提供映射标签，支持对象与数据库的orm字段关系映射
- 提供对象关系映射标签，支持对象关系组建维护
- 提供xml标签，支持编写动态sql。

进入Mybatis

2.1第一个Mybatis程序

思路：搭建环境---->导入Mybatis----->编写代码------>测试

2.1.1搭建环境

- 搭建数据库（自己手动建一个都行）
-

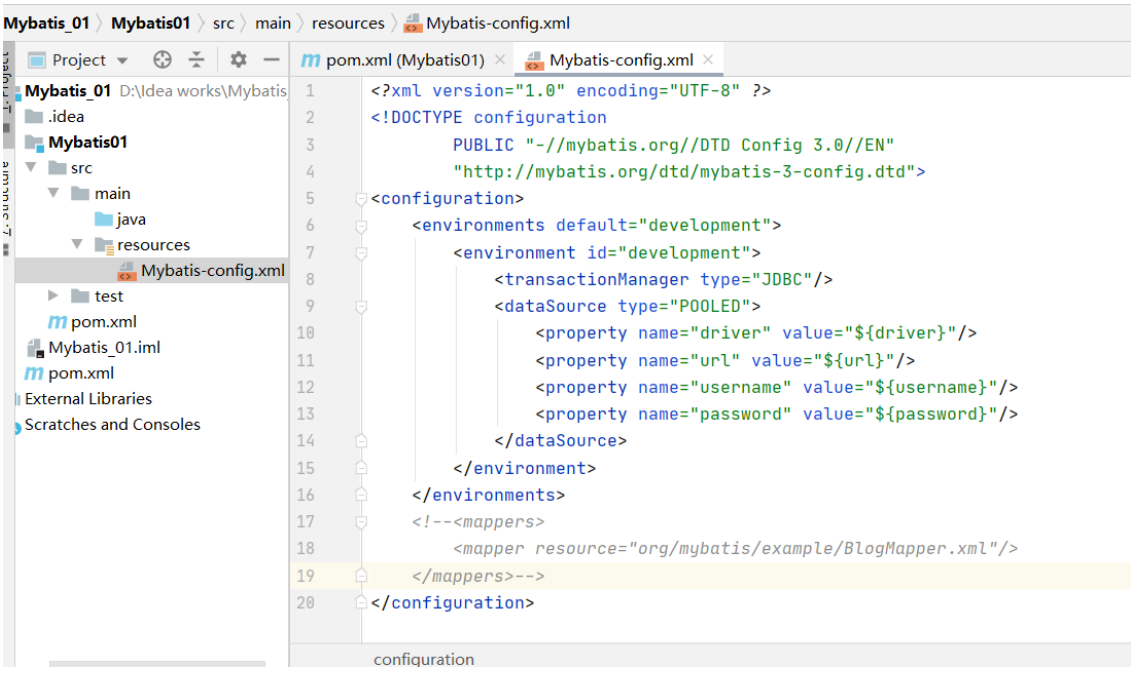
```
自动完成: [Tab]-> 下一个标签, [Ctrl+Space]-> 列出所有标签, [Ctrl+Enter]-> 列出匹配标签
1  CREATE DATABASE `mybatis`;
2
3  USE `mybatis`;
4
5  CREATE TABLE `user`(
6    `id` INT(20) NOT NULL PRIMARY KEY,
7    `name` VARCHAR(30) DEFAULT NULL,
8    `pwd` VARCHAR(30) DEFAULT NULL
9  )ENGINE=INNODB DEFAULT CHARSET=utf8;
10
11 INSERT INTO `user`(`id`,`name`,`pwd`) VALUES
12 (1,'狂神','123456'),
13 (2,'张三','123456'),
14 (3,'李四','123890')
```

- 新建一个普通的maven项目
- 导入maven依赖

```
<dependencies>
        <!--MySQL驱动-->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.23</version>
        </dependency>

        <!--Mybatis-->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.5.6</version>
        </dependency>

        <!--junit-->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.2</version>
            <scope>test</scope>
        </dependency>

    </dependencies>
```

2.1.2创建一个模块

- 编写Mybatis核心配置文件
- 编写Mybatis工具类




```
<?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="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="${driver}"/>
                <property name="url" value="${url}"/>
                <property name="username" value="${username}"/>
                <property name="password" value="${password}"/>
            </dataSource>
        </environment>
    </environments>
    <!--<mappers>
        <mapper resource="org/mybatis/example/BlogMapper.xml"/>
    </mappers>-->
</configuration>
//分界线@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@以下是MySQL8.0.23的，我自己的
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/mybatis?
useSSL=true&amp;serverTimezone=UTC&amp;useUnicode=true&amp;characterEncoding=UTF-
8"/>
                <property name="username" value="root"/>
                <property name="password" value="123456"/>
            </dataSource>
        </environment>
    </environments>

    <mappers>
        <mapper resource="com/feng/dao/UserMapper.xml"/>
    </mappers>



</configuration>
```

```
//编写Mybatis工具类（获取SqlSessionFactory对象）
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 {

    private static SqlSessionFactory sqlSessionFactory;
    static {
        try {
            //第一步：获取SqlSessionFactory对象
            String resource =  "Mybatis-config.xml";
            InputStream inputStream = Resources.getResourceAsStream(resource);
            sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);


        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    //第二步：既然有了 SqlSessionFactory，顾名思义，我们可以
    // 从中获得 SqlSession 的实例。SqlSession 提供了在数据库执行 SQL 命令所需的所有方法。
    // 你可以通过 SqlSession 实例来直接执行已映射的 SQL 语句。
    public static SqlSession SqlSession(){
        SqlSession sqlSession = sqlSessionFactory.openSession();
        return sqlSession;
    }
}
```

2.1.3编写代码

- 实体类

```
package com.feng.pojo;

public class user {
    private int id;
    private String name;
    private int age;

    public user() {
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
```

```
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public user(int id, String name, int age) {
        this.id = id;
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "user{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

- Dao接口

```
package com.feng.dao;

import com.feng.pojo.user;

import java.util.List;

public interface UserDao {
    List<user> getUserList();
}
```

- 接口实现类(现在用Mybatis是用的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">
<!--namespace 绑定一个Dao、Mapper接口-->
<mapper namespace="com.feng.dao.UserDao">
    <!--SQL查询语句-->
    <select id="getUserList" resultType="com.feng.pojo.user">
        select * from mybatis.stu
    </select>
</mapper>
```

2.1.4测试

可能会出现的问题：绑定接口错误或者未绑定；方法名不对；返回类型不对；Maven导出资源问题；配置文件没有注册

```
异常：Exception in thread "main" org.apache.ibatis.binding.BindingException:
<mappers>
        <mapper resource="com/feng/dao/UserMapper.xml"/>
    </mappers>
 在UserMapper.xml文件里面少了绑定
```

```
异常二:(资源项目导出失败)需要在pom.xml文件里面加入这个
    <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>
```

CRUD增删改查

1、namespace：

增删改查:(其中**增删改**需要**提交事务**)

```
package com.feng.Dao;

import com.feng.dao.UserDao;
import com.feng.pojo.user;
import com.feng.utils.MybatisUtils;
import org.apache.ibatis.session.SqlSession;

import java.util.List;

public class UserDaoTest {
    private static user user01;

    /*@Test
        public void test01(){
```

```
        //1.获取sqlSession对象
        SqlSession sqlSession = MybatisUtils.getSqlSession();

        //2.执行sql
        //方式一getMapper(推荐使用的)
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<user> userList = userDao.getUserList();

        for (user use: userList) {
            System.out.println(use);
        }

        //3.释放资源
        sqlSession.close();
    }*/
/*01
public static void main(String[] args) {
    //1.获取sqlSession对象
    SqlSession sqlSession = MybatisUtils.getSqlSession();

    //2.执行sql
    //方式一getMapper(推荐使用的)
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    List<user> userList = userDao.getUserList();

    for (user use: userList) {
        System.out.println(use);
    }

    //3.释放资源
    sqlSession.close();
}*/

/*02
public static void main(String[] args) {
    //1.获取sqlSession对象
    SqlSession sqlSession = MybatisUtils.getSqlSession();

    //2.执行sql
    //方式一getMapper(推荐使用的)
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    user01 = userDao.getuserByID(1);

    System.out.println(user01);

    //3.释放资源
    sqlSession.close();
}*/

/*public static void main(String[] args) {
    SqlSession sqlSession = MybatisUtils.getSqlSession();

    UserDao mapper = sqlSession.getMapper(UserDao.class);
    int i = mapper.insertUser(new user(4, "气味儿", 19));
    int j = mapper.insertUser(new user(3, "蔷薇儿", 19));
    if ( i >= 1) {
        System.out.println("气味儿插入成功");
```

```
            //增删改提交事务**********************
            sqlSession.commit();
        }
        if ( j >= 1 ) {
            System.out.println("蔷薇儿进来了");
            sqlSession.commit();
        }
        else {
            ;
        }
        sqlSession.close();
    }*/
    public static void main(String[] args) {
        SqlSession sqlSession = MybatisUtils.getSqlSession();

        UserDao mapper = sqlSession.getMapper(UserDao.class);
        int i = mapper.deleteUser(2);
        System.out.println(i);

        sqlSession.commit();
        sqlSession.close();
    }
}



//接口
package com.feng.dao;

import com.feng.pojo.user;

import javax.jws.soap.SOAPBinding;
import java.util.List;

public interface UserDao {
    //获取全部的用户
    List<user> getUserList();

    //根据ID查询用户
    user getuserByID( int id );

    //插入用户
    int insertUser( user a );

    //修改用户
    int updateUser( user a );

    //删除一个用户
    int deleteUser( int id );
}


//xml
<?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.dao.UserDao">
    <select id="getUserList" resultType="com.feng.pojo.user">
        select * from mybatis.stu
    </select>

    <select id="getuserByID" parameterType="int"
resultType="com.feng.pojo.user">
        select * from mybatis.stu where id = #{id}
    </select>

    <insert id="insertUser" parameterType="com.feng.pojo.user" >
        insert into mybatis.stu (id ,name, age ) values (#{id}, #{name}, #
{age});
    </insert>

    <update id="updateUser" parameterType="com.feng.pojo.user">
        update mybatis.stu set name = #{name}, age = #{age} where id = #{id}
    </update>

    <delete id="deleteUser" parameterType="int">
        delete from mybatis.stu where id = #{id};
    </delete>
</mapper>

// 注意一一对应的关系 ,
```

模糊查询

```
public static void main(String[] args) {
        SqlSession sqlSession = MybatisUtils.getSqlSession();

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

        String value = "%田%";
        List<user> userLike = mapper.getUserLike("%田%");

        for (user u : userLike) {
            System.out.println(u);
        }

        sqlSession.close();
    }


//userDao

    //模糊查询
    List<user> getUserLike(String value);


//UserMapper.xml         这个里面最好不要写中文注释 utf-8序列异常
    <!--
        sql注入问题:
        我们写的，？是让用户传进去的：select * from mybatis.stu where id = ?
        假如有聪明的用户：  select * from mybatis.stu where id = 1 or 1=1
        这样就会查出所有的了，这就是SQL注入问题
        List<user> userLike = mapper.getUserLike("田");
```

```
        select * from mybatis.stu where name like "%"#{value}"%";
    -->
    <select id="getUserLike" resultType="com.feng.pojo.user">
        select * from mybatis.stu where name like #{value};
    </select>
```

配置解析

1.基本配置

MyBatis 的配置文件包含了会深深影响 MyBatis 行为的设置和属性信息。 配置文档的顶层结构如下：

```
    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）
```

环境配置（environments）

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

不过要记住：尽管可以配置多个环境，但每个 SqlSessionFactory实例只能选择一种环境。

所以，如果你想连接两个数据库，就需要创建两个 SqlSessionFactory 实例，每个数据库对应一个。而如果是三个数据库，就需要三个实例，依此类推，记起来很简单: **每个数据库对应一个 SqlSessionFactory 实例**

```
//environments 元素定义了如何配置环境。
<environments default="test">  //默认使用的环境
  <environment id="development">
    <transactionManager type="JDBC">
      <property name="..." value="..."/>
    </transactionManager>
    <dataSource type="POOLED">
      <property name="driver" value="${driver}"/>
      <property name="url" value="${url}"/>
      <property name="username" value="${username}"/>
```

```
      <property name="password" value="${password}"/>
    </dataSource>
  </environment>

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

注意一些关键点:

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

默认环境和环境 ID 顾名思义。 环境可以随意命名，但务必保证默认的环境 ID 要匹配其中一个环境 ID。

事务管理器（transactionManager）

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

- JDBC – 这个配置直接使用了 JDBC 的提交和回滚设施，它依赖从数据源获得的连接来管理事务作用域。()
- MANAGED – 这个配置几乎没做什么(**了解就可以**)。它从不提交或回滚一个连接，而是让容器来管理事务的整个生命周期（比如 JEE 应用服务器的上下文）。 默认情况下它会关闭连接。然而一些容器并不希望连接被关闭，因此需要将 closeConnection 属性设置为 false 来阻止默认的关闭行为。例如:

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

数据源（dataSource）

dataSource 元素使用标准的 JDBC 数据源接口来配置 JDBC 连接对象的资源。大多数 MyBatis应用程序会按示例中的例子来配置数据源。虽然数据源配置是可选的，但如果要启用延迟加载特性，就必须配置数据源。

有三种内建的数据源类型（也就是 type="[UNPOOLED|POOLED|JNDI]"）：

UNPOOLED– 这个数据源的实现会每次请求时打开和关闭连接。虽然有点慢，但对那些数据库连接可用性要求不高的简单应用程序来说，是一个很好的选择。 性能表现则依赖于使用的数据库，对某些数据库来说，使用连接池并不重要，这个配置就很适合这种情形。UNPOOLED 类型的数据源仅仅需要配置以下 5 种属性：

- `driver` – 这是 JDBC 驱动的 Java 类全限定名（并不是 JDBC 驱动中可能包含的数据源类）。

- `url` – 这是数据库的 JDBC URL 地址。
- `username` – 登录数据库的用户名。
- `password` – 登录数据库的密码。
- `defaultTransactionIsolationLevel` – 默认的连接事务隔离级别。
- `defaultNetworkTimeout` – 等待数据库操作完成的默认网络超时时间（单位：毫秒）。查看 `java.sql.Connection#setNetworkTimeout()` 的 API 文档以获取更多信息。

POOLED（默认的） – 这种数据源的实现利用“池”的概念将 JDBC 连接对象组织起来，避免了创建新的连接实例时所必需的初始化和认证时间。 这种处理方式很流行，能使并发 Web 应用快速响应请求。

除了上述提到 UNPOOLED 下的属性外，还有更多属性用来配置 POOLED 的数据源：

- `poolMaximumActiveConnections` – 在任意时间可存在的活动（正在使用）连接数量，默认值：10
- `poolMaximumIdleConnections` – 任意时间可能存在的空闲连接数。
- `poolMaximumCheckoutTime` – 在被强制返回之前，池中连接被检出（checked out）时间，默认值：20000 毫秒（即 20 秒）
- `poolTimeToWait` – 这是一个底层设置，如果获取连接花费了相当长的时间，连接池会打印状态日志并重新尝试获取一个连接（避免在误配置的情况下一直失败且不打印日志），默认值：20000 毫秒（即 20 秒）。
- `poolMaximumLocalBadConnectionTolerance` – 这是一个关于坏连接容忍度的底层设置，作用于每一个尝试从缓存池获取连接的线程。 如果这个线程获取到的是一个坏的连接，那么这个数据源允许这个线程尝试重新获取一个新的连接，但是这个重新尝试的次数不应该超过 `poolMaximumIdleConnections` 与 `poolMaximumLocalBadConnectionTolerance` 之和。默认值：3（新增于 3.4.5）
- `poolPingQuery` – 发送到数据库的侦测查询，用来检验连接是否正常工作并准备接受请求。默认是“NO PING QUERY SET”，这会导致多数数据库驱动出错时返回恰当的错误消息。
- `poolPingEnabled` – 是否启用侦测查询。若开启，需要设置 `poolPingQuery` 属性为一个可执行的 SQL 语句（最好是一个速度非常快的 SQL 语句），默认值：false。
- `poolPingConnectionsNotUsedFor` – 配置 poolPingQuery 的频率。可以被设置为和数据库连接超时时间一样，来避免不必要的侦测，默认值：0（即所有连接每一时刻都被侦测 — 当然仅当 poolPingEnabled 为 true 时适用）。

JNDI – 这个数据源实现是为了能在如 EJB 或应用服务器这类容器中使用，容器可以集中或在外部配置数据源，然后放置一个 JNDI 上下文的数据源引用。

属性(properties)

我们可以用properties属性来引用其他的properties文件

这些属性可以在外部进行配置，并可以进行动态替换。你既可以在典型的 Java 属性文件中配置这些属性，也可以在 properties 元素的子元素中设置。

编写一个properties文件(db.properties)

```
driver=com.mysql.cj.jdbc.Driver
url=jdbc:mysql://localhost:3306/mybatis?
useSSL=true&serverTimezone=UTC&useUnicode=true&characterEncoding=UTF-8
username=root //或者这里不写
password=123456   //或者这里不写
```

在核心配置文件中引入

注意一个问题：

```
    </mappers>

    <properties></properties>


</configuration>
```

The content of element type "configuration" must match "(properties?,settings?,typeAliases?,typeHandlers?,objectFactory?,objectWrapperFactory?,reflectorFactory?

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

    <!--引入外部配置文件-->
    <properties resource="db.properties">
        <!--上面不写这两个属性，这里写上
        但是如果都写了，优先选择properties配置文件里面的
        <property name="username" value="root"/>
        <property name="passowrd" value="123456"/>
        -->
    </properties>

    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="${driver}"/>
                <property name="url" value="${url}"/>
                <property name="username" value="${username}"/>
                <property name="password" value="${password}"/>
            </dataSource>
        </environment>
    </environments>

    <mappers>
        <mapper resource="com/feng/dao/UserMapper.xml"/>
    </mappers>



</configuration>
```

类型别名（typeAliases）

类型别名可为 Java 类型设置一个缩写名字。

它仅用于 XML 配置，意在降低冗余的全限定类名书写。

类（可以自定义别名）

这是**原来的**:

```xml
    <select id="getUserList" resultType="com.feng.pojo.user">
        select * from mybatis.stu
    </select>

    <select id="getuserByID" parameterType="int" resultType="com.feng.pojo.user">
        select * from mybatis.stu where id = #{id}
    </select>

    <insert id="insertUser" parameterType="com.feng.pojo.user" >
        insert into mybatis.stu (id ,name, age ) values (#{id}, #{name}, #{age});
    </insert>

    <update id="updateUser" parameterType="com.feng.pojo.user">
        update mybatis.stu set name = #{name}, age = #{age} where id = #{id}
    </update>

    <delete id="deleteUser" parameterType="int">
        delete from mybatis.stu where id = #{id};
    </delete>
```

加上

```xml
    <properties resource="db.properties">
        <property name="username" value="root"/>
        <property name="passowrd" value="123456"/>
    </properties>

    <!--起别名-->
    <typeAliases>
        <typeAlias type="com.feng.pojo.user" alias="user"></typeAlias>
    </typeAliases>

    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="${driver}"/>
```

configuration › typeAliases › typeAlias

现在的

```xml
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.feng.dao.UserDao">

    <select id="getUserList" resultType="user">
        select * from mybatis.stu
    </select>

    <select id="getuserByID" parameterType="int" resultType="user">
        select * from mybatis.stu where id = #{id}
    </select>

    <insert id="insertUser" parameterType="com.feng.pojo.user" >
        insert into mybatis.stu (id ,name, age ) values (#{id}, #{name}, #{age});
    </insert>

    <update id="updateUser" parameterType="com.feng.pojo.user">
        update mybatis.stu set name = #{name}, age = #{age} where id = #{id}
```

也可以指定一个包名，MyBatis 会在包名下面搜索需要的 Java Bean，(不能自定义别名，如果非要改，则要在实体类上加上注解：@Alias("别名"))

```
<typeAliases>
        <package name="com.feng.pojo"/>
    </typeAliases>
```

假如原来的包中的类名叫：Student ,默认别名就是student **首字母小写**！！

设置(settings)

映射器（mappers)

既然 MyBatis 的行为已经由上述元素配置完了，我们现在就要来定义 SQL 映射语句了。 但首先，我们需要告诉 MyBatis 到哪里去找到这些语句。 在自动查找资源方面，Java 并没有提供一个很好的解决方案，所以最好的办法是直接告诉 MyBatis 到哪里去找映射文件。你可以使用相对于类路径的资源引用，或完全限定资源定位符（包括 `file:///` 形式的 URL），或类名和包名等。

```
<!-- 使用相对于类路径的资源引用 -->
<mappers>
  <mapper resource="org/mybatis/builder/AuthorMapper.xml"/>
  <mapper resource="org/mybatis/builder/BlogMapper.xml"/>
  <mapper resource="org/mybatis/builder/PostMapper.xml"/>
</mappers>

<!-- 使用完全限定资源定位符（URL） 建议了解这个就行了，不要去用-->
<mappers>
  <mapper url="file:///var/mappers/AuthorMapper.xml"/>
  <mapper url="file:///var/mappers/BlogMapper.xml"/>
  <mapper url="file:///var/mappers/PostMapper.xml"/>
</mappers>

<!-- 使用映射器接口实现类的完全限定类名 -->
<!--1.接口和他的Mapper配置文件必须同名(例如：接口叫UserMapper，配置文件必须叫
UserMapperxml)-->
<!--2.接口和他的Mapper配置文件必须在同一个包下-->
<mappers>
  <mapper class="org.mybatis.builder.AuthorMapper"/>
  <mapper class="org.mybatis.builder.BlogMapper"/>
  <mapper class="org.mybatis.builder.PostMapper"/>
</mappers>

<!-- 将包内的映射器接口实现全部注册为映射器 -->
<!--1.接口和他的Mapper配置文件必须同名(例如：接口叫UserMapper，配置文件必须叫
UserMapperxml)-->
<!--2.接口和他的Mapper配置文件必须在同一个包下-->
<mappers>
  <package name="org.mybatis.builder"/>
</mappers>
```

生命周期

作用域和生命周期类别是至关重要的，因为错误的使用会导致非常严重的并发问题。

SqlSessionFactoryBuilder

这个类可以被实例化、使用和丢弃，**一旦创建SqlSessionFactory，就不再需要它了。** 因此 SqlSessionFactoryBuilder 实例的最佳作用域是方法作用域（也就是**局部方法变量**）。你可以重用 SqlSessionFactoryBuilder 来创建多个 SqlSessionFactory 实例，但最好还是不要一直保留着它，以保证所有的 XML 解析资源可以被释放给更重要的事情。

SqlSessionFactory

SqlSessionFactory **一旦被创建就应该在应用的运行期间一直存在，没有任何理由丢弃它或重新创建另一个实例。**使用 SqlSessionFactory 的最佳实践是在应用运行期间不要重复创建多次，多次重建 SqlSessionFactory 被视为一种代码“坏习惯”。因此 SqlSessionFactory 的最佳作用域是**应用作用域。**有很多方法可以做到，最简单的就是使用单例模式或者静态单例模式。

可以想象成数据库连接池

SqlSession

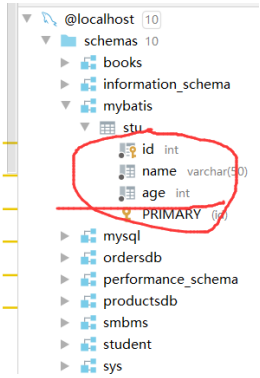
每个线程都应该有它自己的 SqlSession 实例。SqlSession的实例不是线程安全的，因此是不能被共享的，所以它的最佳的作用域是**请求或方法作用域。绝对不能将 SqlSession 实例的引用放在一个类的静态域，甚至一个类的实例变量也不行。也绝不能将 SqlSession 实例的引用放在任何类型的托管作用域中，**比如 Servlet 框架中的 HttpSession。如果你现在正在使用一种 Web 框架，考虑将 SqlSession 放在一个和 HTTP 请求相似的作用域中。换句话说，每次收到 HTTP 请求，就可以打开一个 SqlSession，返回一个响应后，就关闭它。这个关闭操作很重要，为了确保每次都能执行关闭操作，你应该把这个关闭操作放到 finally 块中.

连接到连接池中的一个请求，需要开启和关闭

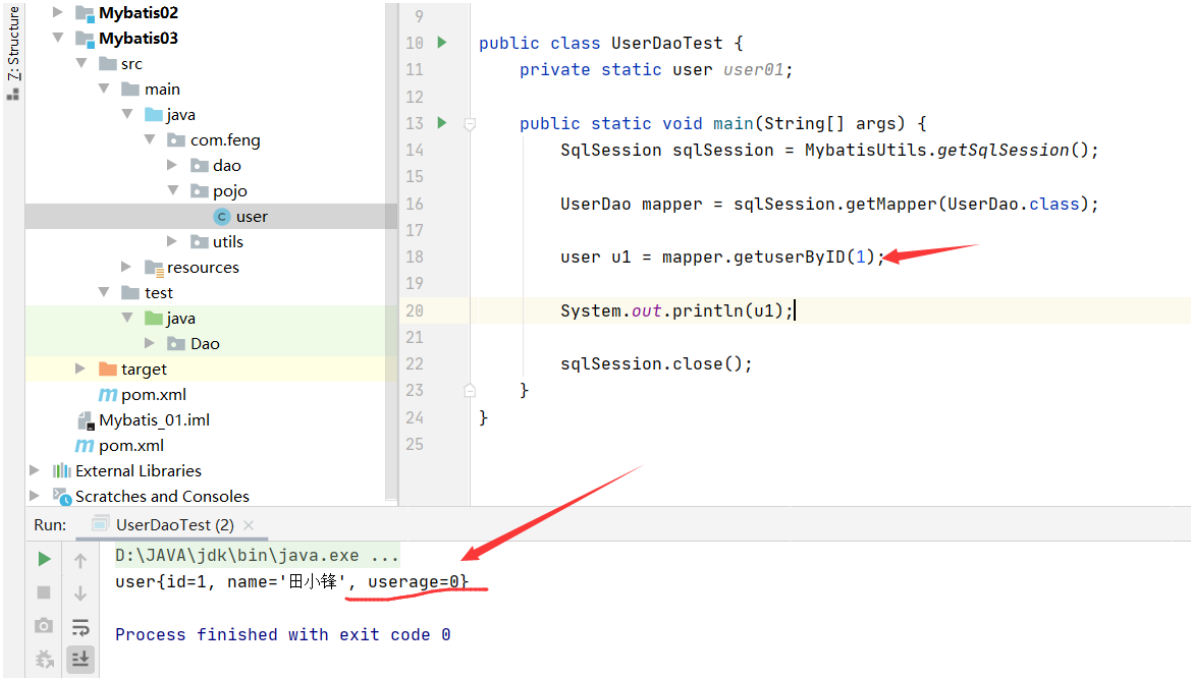
用完之后赶紧关闭，否则资源被占用

ResultMap结果集映射

```java
public class user {
    private int id;
    private String name;
    private int userage;


    public user() {

    }

    public user(int id, String name, int userage) {
        this.id = id;
        this.name = name;
        this.userage = userage;
    }
```



实体类属性名和数据库字段名不一致



查询结果出现问题！！！！！！！！！！！！

解决方法1：起别名

```
<select id="getuserByID" parameterType="int" resultType="user">
        //原来是select *  ..............
        select id,name,age as userage from mybatis.stu where id = #{id}
</select>
```

解决方法2：

column数据库字段名 对应 property实体类属性名 结果集映射

- `resultMap` 元素是 MyBatis 中最重要最强大的元素。
- 引用它的语句中设置 `resultMap` 属性就行了（注意我们**去掉了** `resultType` **属性**）
- ResultMap 的设计思想是，对简单的语句做到零配置，对于复杂一点的语句，只需要描述语句之间的关系就行了。
- **如果**这个世界**总是这么简单**就好了。

日志

有时候需要**排错**！！！！！！

logImpl	指定 MyBatis 所用日志的具体实现，未指定时将自动查找。	SLF4J | LOG4J | LOG4J2 | JDK_LOGGING | COMMONS_LOGGING | STDOUT_LOGGING | NO_LOGGING	未设置	默认未设置

设置（settings）

这是 MyBatis 中极为重要的调整设置，它们会改变 MyBatis 的运行时行为。下表描述了设置中各项设置的含义、默认值等。

指定 MyBatis 所用日志的具体实现，未指定时将自动查找。	**SLF4J 、LOG4J 、LOG4J2 、JDK_LOGGING、COMMONS_LOGGING、STDOUT_LOGGING（标准日志输出） 、NO_LOGGING**				

在配置文件里面加上这个（注意里面的顺序!!!）

```
<settings>
    <!--value的 值必须和原来的一模一样，空格都不能有-->
        <setting name="logImpl" value="STDOUT_LOGGING"/>
    </settings>
```

```
PooledDataSource forcefully closed/removed all connections.
Opening JDBC Connection
Created connection 262457445.
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@fa4c865]
==>  Preparing: select * from mybatis.stu where id = ?
==> Parameters: 1(Integer)
<==    Columns: id, name, age
<==        Row: 1, 田小锋, 20
<==      Total: 1
user{id=1, name='田小锋', userage=20}
Resetting autocommit to true on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@fa4c865]
Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@fa4c865]
Returned connection 262457445 to pool.

Process finished with exit code 0
```

LOG4J

什么是LOG4J

- Log4j是Apache的一个开源项目，通过使用Log4j，我们可以控制日志信息输送的目的地是控制台、文件、GUI组件，甚至是套接口服务器、NT的事件记录器、UNIX Syslog守护进程等
- 我们也可以控制每一条日志的输出格式
- 通过定义每一条日志信息的级别，我们能够更加细致地控制日志的生成过程
- 可以通过一个配置文件来灵活地进行配置，而不需要修改应用的代码。

使用LOG4J01

- 先导包

```
<!-- https://mvnrepository.com/artifact/log4j/log4j -->
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

- 写log4j.properties配置文件

```
#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
log4j.rootLogger=DEBUG,console,file

#控制台输出相关设置
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

#文件输出相关设置
log4j.appender.file = org.apache.log4j.RollingFileAppender
log4j.appender.file.File = ./log/feng.log
log4j.appender.file.MaxFileSize=10mb
log4j.appender.file.Threshold=DENUG
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=[%p][%d{yy-MM-dd}][%c]%m%n

#日志相关设置
log4j.logger.org.mtbatis=DEBUG
log4j.logger.java.sql=DEBUG
log4j.logger.java.sql.Statement=DEBUG
log4j.logger.java.sql.ResultSet=DEBUG
log4j.logger.java.sql.PreparedStstement=DEBUG
```

- 核心配置文件(Mybatis-config.xml)里面

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

使用log4j02

导包：import org.apache.log4j.Logger;

```
package Dao;

import com.feng.dao.UserDao;
import com.feng.pojo.user;
import com.feng.utils.MybatisUtils;
import org.apache.ibatis.session.SqlSession;
import org.apache.log4j.Logger;
```

```
import java.util.List;

public class UserDaoTest {
    private static user user01;
    static Logger logger = Logger.getLogger(UserDaoTest.class);

    /*public static void main(String[] args) {
        SqlSession sqlSession = MybatisUtils.getSqlSession();

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

        user u1 = mapper.getuserByID(1);

        System.out.println(u1);

        sqlSession.close();
    }*/
    public static void main(String[] args) {
        logger.info("info进来了");
        logger.debug("debug进来了");
        logger.error("error进来了");
    }
}
```

Limit分页

为什么需要分页

减少数据的处理量

```
select * from user limit start,end ;
```

Mybatis实现分页

- 接口

```
//分页查询
    List<user> getUserListBylimit(Map<String, Integer> map);
```

- ***Mapper.xml

```
<select id="getUserListBylimit" parameterType="map" resultMap="userMap">
        select * from mybatis.stu limit #{startIndex}, #{pageSize}
    </select>
```

- 测试

```
public static void main(String[] args) {
        SqlSession sqlSession = MybatisUtils.getSqlSession();

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

        HashMap<String, Integer> map = new HashMap<String, Integer>();
        map.put("startIndex", 0);
        map.put("pageSize", 2);

        List<user> userListBylimit = mapper.getUserListBylimit(map);

        for (user u1 : userListBylimit) {
            System.out.println(u1);
        }

        sqlSession.close();
    }
```

注解开发

对于像 BlogMapper 这样的映射器类来说，还有另一种方法来完成语句映射。 它们映射的语句可以不用 XML 来配置，而可以使用 Java 注解来配置。比如，上面的 XML示例可以被替换成如下的配置：

```
package org.mybatis.example;
public interface BlogMapper {
  @Select("SELECT * FROM blog WHERE id = #{id}")
  Blog selectBlog(int id);
}
```

使用注解来映射简单语句会使代码显得更加简洁，但对于稍微复杂一点的语句，Java 注解不仅力不从心，还会让你本就复杂的 SQL 语句更加混乱不堪。 因此，如果你需要做一些很复杂的操作，最好用 XML 来映射语句。

选择何种方式来配置映射，以及认为是否应该要统一映射语句定义的形式，完全取决于你和你的团队。 换句话说，永远不要拘泥于一种方式，你可以很轻松的在基于注解和 XML的语句映射方式间自由移植和切换。

使用注解开发：

1.注解在接口上实现

```
@Select("select * from stu")
    List<user> getList();
```

2.在核心配置文件里面绑定接口

```
<mappers>
        <mapper class="com.feng.dao.UserDao"></mapper>
    </mappers>
```

3.测试

本质：反射机制实现

底层：动态代理

可以自动提交事物：(在工具类里面这么写，true)

```
public static SqlSession getSqlSession(){
        SqlSession sqlSession = sqlSessionFactory.openSession(true); //true自动提交事务
        return sqlSession;
    }
```

接口

```
package com.feng.dao;

import com.feng.pojo.user;
import org.apache.ibatis.annotations.*;

import java.util.List;
import java.util.Map;

public interface UserDao {

    @Select("select * from stu")
    List<user> getAll();

    //有多个参数时，最好在参数前面加上注解，如下(SQL语句中以注解中的名字为主)
    @Select("select * from stu where id = #{id}")
    user getUserByID( @Param("id") int id );

    @Insert("insert stu (id, name, age) values (#{id},#{name},#{userage})")
    int InsertUser(user u);

    @Update("update stu set name = #{name}, age=#{userage} where id = #{id}")
    int Update(user u);

    @Delete("delete from stu where id = #{uid}")
    int DeleteUser(@Param("uid") int id);
}
```

测试类

```
import com.feng.dao.UserDao;
import com.feng.pojo.user;
import com.feng.utils.MybatisUtils;
import org.apache.ibatis.session.SqlSession;

import java.util.List;
```

```
public class MapperTest {
    public static void main(String[] args) {
        SqlSession sqlSession = MybatisUtils.getSqlSession();

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

//        List<user> list = mapper.getAll();
//
//        for (user useru1 : list) {
//            System.out.println(useru1);
//        }

//        user userByID = mapper.getUserByID(1);
//        System.out.println(userByID);
//        sqlSession.close();

//        int i = mapper.InsertUser(new user(6, "嘻嘻嘻", 20));
//        System.out.println(i);

//        int i = mapper.Update(new user(6,"囉囉胡",22));

        int i = mapper.DeleteUser(6);

        sqlSession.close();
    }
}
```

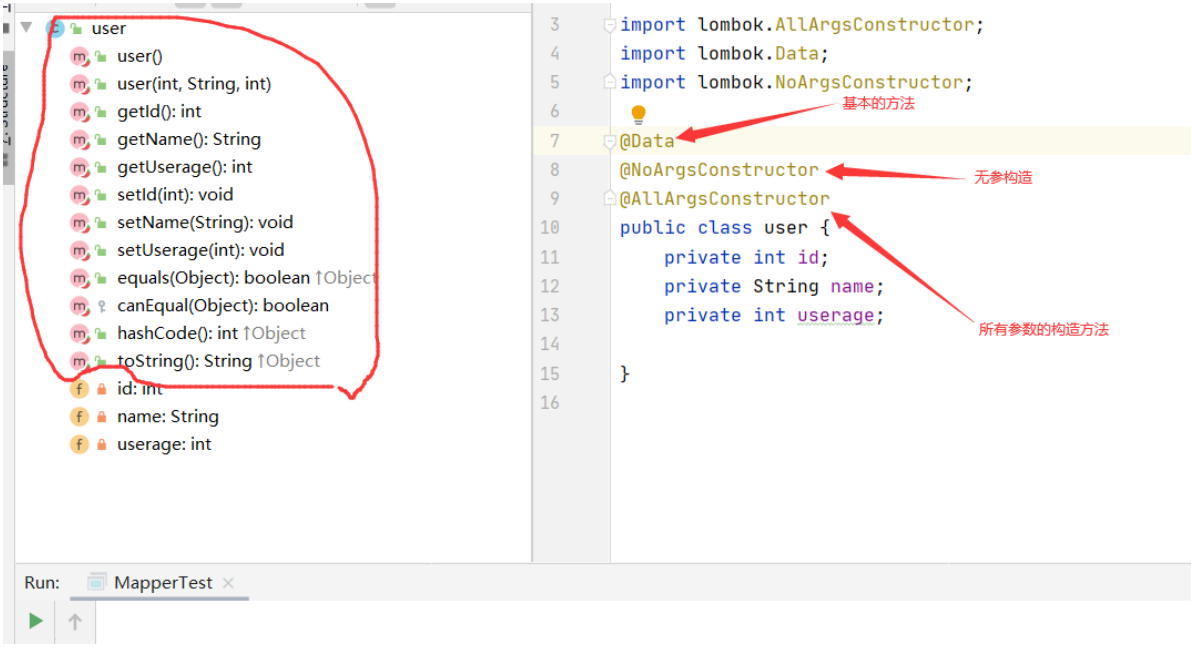
关于@Param()注解

- 基本类型的参数或者String类型，加上
- 引用类型不需要加
- 如果只有一个基本类型的话，可以忽略，最好加上
- 我们在SQL中引用的就是我们这里的@Param("**") 里面打星号的内容

Lombok

```
<dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.20</version>
        <scope>provided</scope>
    </dependency>
```

Maven依赖



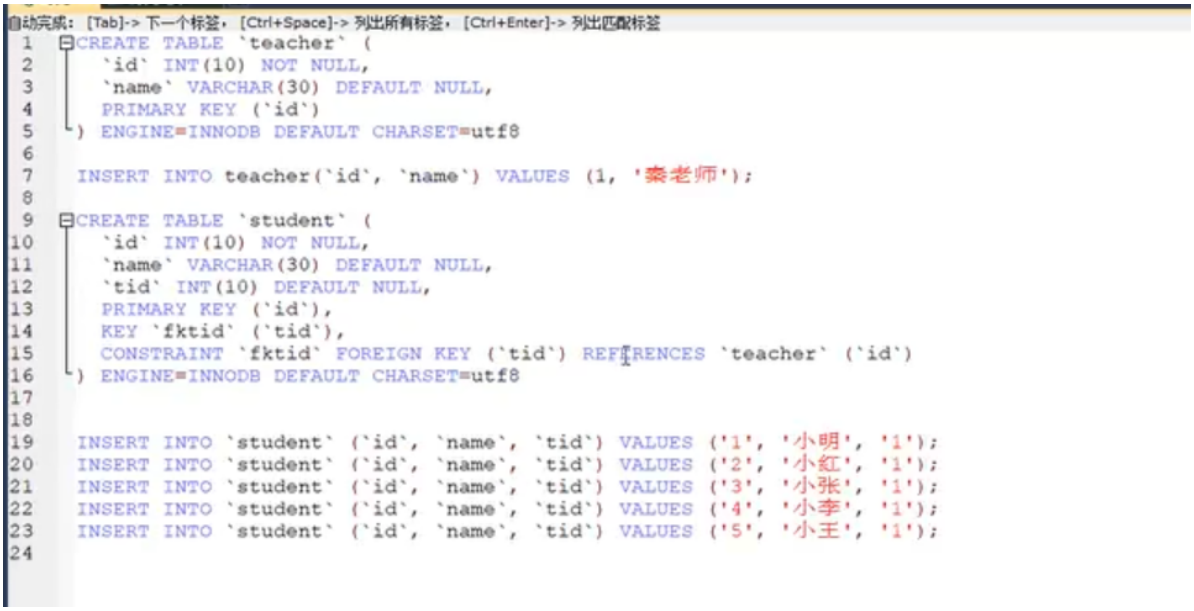
几个注解，这些方法全出来了！！！！！！！！！！！

偷懒工具！！！！！！！！！

多对一

- 例子：对于学生而言，多个学生对应一个老师
- 对于学生这边，【关联】多个学生关联一个老师**{多对一}**
- 对于老师这边，【集合】一个 老师教多个学生**{一对多}**

建立数据库：



```
自动完成: [Tab]-> 下一个标签, [Ctrl+Space]-> 列出所有标签, [Ctrl+Enter]-> 列出匹配标签
CREATE TABLE `teacher` (
  `id` INT(10) NOT NULL,
  `name` VARCHAR(30) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=INNODB DEFAULT CHARSET=utf8

INSERT INTO teacher(`id`, `name`) VALUES (1, '秦老师');

CREATE TABLE `student` (
  `id` INT(10) NOT NULL,
  `name` VARCHAR(30) DEFAULT NULL,
  `tid` INT(10) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `fktid` (`tid`),
  CONSTRAINT `fktid` FOREIGN KEY (`tid`) REFERENCES `teacher` (`id`)
) ENGINE=INNODB DEFAULT CHARSET=utf8


INSERT INTO `student` (`id`, `name`, `tid`) VALUES ('1', '小明', '1');
INSERT INTO `student` (`id`, `name`, `tid`) VALUES ('2', '小红', '1');
INSERT INTO `student` (`id`, `name`, `tid`) VALUES ('3', '小张', '1');
INSERT INTO `student` (`id`, `name`, `tid`) VALUES ('4', '小李', '1');
INSERT INTO `student` (`id`, `name`, `tid`) VALUES ('5', '小王', '1');
```

```
create table `teacher`(
      `id` int(10) not null,
      `name` varchar(30) default null,
      primary key(`id`)
) ENGINE=INNODB DEFAULT CHARSET=UTF8

insert into teacher (`id`, `name`) values ( 1, '田老师');
```

```sql
create table `student`(
     `id` int(10) not null,
     `name` varchar(30) default null,
     `tid` int(10) default null,
     primary key(`id`),
     key `fktid` (`tid`),
     constraint `fktid` foreign key (`tid`) references `teacher` (`id`)
) ENGINE=INNODB DEFAULT CHARSET=UTF8;

insert into `student` (`id`, `name`, `tid`) values ('1', '小明', '1');
insert into `student` (`id`, `name`, `tid`) values ('2', '小君', '1');
insert into `student` (`id`, `name`, `tid`) values ('3', '小o', '1');
insert into `student` (`id`, `name`, `tid`) values ('4', '小求', '1');
insert into `student` (`id`, `name`, `tid`) values ('5', '小阿', '1');
```

步骤:

1. 新建实体类Teacher，Student
2. 建立Mapper接口
3. 建立Mapper.xml文件
4. 在核心配置文件里面注册我们的Mapper接口或者文件
5. 测试

```java
package com.feng.pojo;
//学生类
public class Student {
    private int id;
    private String name;

    private Teacher teacher;

    public Student() {
    }

    public Student(int id, String name, Teacher teacher) {
        this.id = id;
        this.name = name;
        this.teacher = teacher;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
    public Teacher getTeacher() {
        return teacher;
    }

    public void setTeacher(Teacher teacher) {
        this.teacher = teacher;
    }

    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", teacher=" + teacher +
                '}';
    }
}

//老师表
package com.feng.pojo;

public class Teacher {
    private int id;
    private String name;

    public Teacher() {
    }

    public Teacher(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Teacher{" +
                "id=" + id +
                ", name='" + name + '\'' +
                '}';
    }
}
```

```java
//StudentMapper接口
package com.feng.dao;

import com.feng.pojo.Student;

import java.util.List;

public interface StudentMapper {
    //查询所有学生的老师
    public List<Student> getStudent();
    public List<Student> getStudent2();
}
```

```xml
<!--StudentMapper.xml
查询嵌套处理 子查询
-->
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.feng.dao.StudentMapper">
    <select id="getStudent" resultMap="StudentTeacher">
        select * from student;
    </select>

    <resultMap id="StudentTeacher" type="Student">
        <result property="id" column="id"/>
        <result property="name" column="name"/>

        <!--复杂的属性，单独处理 association:对象  collection:集合-->
        <association property="teacher" column="tid" javaType="Teacher"
select="getTeacher" />

    </resultMap>

    <select id="getTeacher" resultType="Teacher">
        select * from teacher where id = #{id}
    </select>
</mapper>

<!--StudentMapper.xml
结果嵌套处理  连表查询
-->
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.feng.dao.StudentMapper">

    <select id="getStudent2" resultMap="StudentTeacher2">
        select s.id sid, s.name sname, t.name tname
```

```
        from student s, teacher t
        where s.tid = t.id
    </select>
    <resultMap id="StudentTeacher2" type="Student">
        <result property="id" column="sid"/>
        <result property="name" column="sname"/>

        <association property="teacher" javaType="Teacher">
            <result property="name" column="tname"/>
        </association>
    </resultMap>
</mapper>
学生表：id   name    tid
       1    小明      1
       2    b        1
        ......
老师表：id   name
       1    田老师
学生表里面的tid对应老师表的id,
上述xml是两表联合查询，在实体类中，学生里面有私有属性Teacher，teacher不是普通属性，是复杂属性！！！
```

```
public static void main(String[] args) {
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
        List<Student> stu = mapper.getStudent();
        for (Student student : stu) {
            System.out.println(student);
        }
        sqlSession.close();
    }

结果:
Student{id=1, name='小明', teacher=Teacher{id=1, name='田老师'}}
Student{id=2, name='小君', teacher=Teacher{id=1, name='田老师'}}
Student{id=3, name='小o', teacher=Teacher{id=1, name='田老师'}}
Student{id=4, name='小求', teacher=Teacher{id=1, name='田老师'}}
Student{id=5, name='小阿', teacher=Teacher{id=1, name='田老师'}}
```

一对多

一个老师有多个学生，对于老师而言，就是一对多

步骤和上面一样的

```
//实体类
package com.feng.pojo;

public class Student {
    private int id;
    private String name;
    private int tid;

    public Student() {
```

```
    }

    public Student(int id, String name, int tid) {
        this.id = id;
        this.name = name;
        this.tid = tid;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getTid() {
        return tid;
    }

    public void setTid(int tid) {
        this.tid = tid;
    }

    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", tid=" + tid +
                '}';
    }
}


package com.feng.pojo;

import java.util.List;

public class Teacher {
    private int id;
    private String name;
    private List<Student> students;

    public Teacher() {
    }

    public Teacher(int id, String name, List<Student> students) {
        this.id = id;
        this.name = name;
```

```
        this.students = students;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public List<Student> getStudents() {
        return students;
    }

    public void setStudents(List<Student> students) {
        this.students = students;
    }

    @Override
    public String toString() {
        return "Teacher{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", students=" + students +
                '}';
    }
}
```

```
//接口
package com.feng.dao;

import com.feng.pojo.Teacher;
import org.apache.ibatis.annotations.Param;
import org.apache.ibatis.annotations.Select;

import java.util.List;

public interface TeacherMapper {

    //获取老师下的所有学生以及学生对应的老师
    Teacher getTeacher(@Param("tid") int id);

    Teacher getTeacher2(@Param("tid") int id);
}
```

```xml
<!--TeacherMapper.xml-->
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.feng.dao.TeacherMapper">

    <!--结果嵌套处理：推荐使用的哟，-->
    <select id="getTeacher" 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 and t.id=#{tid}
    </select>
    <resultMap id="TeacherStudent" type="Teacher">
        <result property="id" column="tid"/>
        <result property="name" column="tname"/>

        <!--javaType:指定属性的类型
            ofType:集合中的泛型信息
        -->
        <collection property="students" ofType="Student" >
            <result property="id" column="sid"/>
            <result property="name" column="sname"/>
            <result property="tid" column="tid"/>
        </collection>
    </resultMap>

    <!--查询嵌套处理-->
    <select id="getTeacher2" resultMap="TeacherStudent2">
        select * from teacher where id = #{tid}
    </select>
    <resultMap id="TeacherStudent2" type="Teacher">
        <collection property="students" javaType="ArrayList" ofType="Student"
select="getStudentByTeacherId" column="id"/>
    </resultMap>
    <select id="getStudentByTeacherId" resultType="Student">
        select * from student where tid = #{tid}
    </select>
</mapper>
```

```java
//测试
import com.feng.dao.TeacherMapper;
import com.feng.pojo.Teacher;
import com.feng.utils.MybatisUtils;
import org.apache.ibatis.session.SqlSession;

import java.util.List;

public class Test02 {
    public static void main(String[] args) {
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        TeacherMapper mapper = sqlSession.getMapper(TeacherMapper.class);
        Teacher teacher = mapper.getTeacher(1);
        System.out.println(teacher);

        sqlSession.close();
    }
```

```
}


Teacher{id=1, name='田老师', students=[Student{id=1, name='小明', tid=1},
Student{id=2, name='小君', tid=1}, Student{id=3, name='小o', tid=1},
Student{id=4, name='小求', tid=1}, Student{id=5, name='小阿', tid=1}]}
```

面试高频：MySQL引擎，INNODB底层原理、索引、索引优化！！！！！！！

动态SQL

根据不同的环境生成不同的sql语句,**SQL拼接**

搭建环境(数据库名字是mybatis,上面都是这个数据库)

所谓的动态SQL，本质还是SQL，只是在SQL层面，执行一些逻辑代码而已

```
create table `blog`(
    `id` varchar(50) not null comment '博客ID',
    `title` varchar(100) not null comment '博客标题',
    `anthor` varchar(30) not null comment '博客作者',
    `create_time` datetime not null comment '创建时间',
    `views` int(30) not null comment '浏览量'
)ENGINE=INNODB DEFAULT CHARSET=UTF8;
```

```
//实体类Blog
package com.feng.pojo;

import java.util.Date;

public class Blog {
    private int id;
    private String title;
    private String anthor;
    private Date createTime;
    private int views;

    public Blog() {
    }

    public Blog(int id, String title, String anthor, Date createTime, int views)
{
        this.id = id;
        this.title = title;
        this.anthor = anthor;
        this.createTime = createTime;
        this.views = views;
    }

    public int getId() {
        return id;
    }
```

```
    public void setId(int id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getAnthor() {
        return anthor;
    }

    public void setAnthor(String anthor) {
        this.anthor = anthor;
    }

    public Date getCreateTime() {
        return createTime;
    }

    public void setCreateTime(Date createTime) {
        this.createTime = createTime;
    }

    public int getViews() {
        return views;
    }

    public void setViews(int views) {
        this.views = views;
    }

    @Override
    public String toString() {
        return "Blog{" +
                "id=" + id +
                ", title='" + title + '\'' +
                ", anthor='" + anthor + '\'' +
                ", createTime=" + createTime +
                ", views=" + views +
                '}';
    }
}
```

```
<!--mybatis-config.xml里面-->
<settings>
        <setting name="logImpl" value="STDOUT_LOGGING"/>
        <setting name="mapUnderscoreToCamelCase" value="true"/>
    </settings>
```

```xml
<!--BlogMapper.xml-->
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.feng.dao.BlogMapper">
    <select id="addBlog" parameterType="blog">
        insert into mybatis.blog  ( id, title, anthor, create_time, views)
        values (#{id}, #{title}, #{anthor}, #{createTime}, #{views});
    </select>
</mapper>
```

```java
import com.feng.dao.BlogMapper;
import com.feng.pojo.Blog;
import com.feng.utils.IDutils;
import com.feng.utils.MybatisUtils;
import org.apache.ibatis.session.SqlSession;

import java.util.Date;

public class Test03 {
    public static void main(String[] args) {
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);

        Blog blog = new Blog();
        blog.setId(IDutils.getId());
        blog.setTitle("Mybatis");
        blog.setAnthor("田小锋");
        blog.setCreateTime(new Date());
        blog.setViews(9999);
        mapper.addBlog(blog);

        blog.setId(IDutils.getId());
        blog.setTitle("JAVA");
        mapper.addBlog(blog);

        blog.setId(IDutils.getId());
        blog.setTitle("SpringMVC");
        mapper.addBlog(blog);

        blog.setId(IDutils.getId());
        blog.setTitle("微服务");
        mapper.addBlog(blog);
        sqlSession.close();
    }
}
```

IF标签

```
<!--BlogMapper.xml里面-->
<select id="QueryBlogIF" parameterType="map" resultType="blog">
        select * from mybatis.blog where 1 = 1
        <if test="title != null">
            and title=#{title}
        </if>
        <if test="anthor != null">
            and anthor=#{anthor}
        </if>
    </select>
```

choose(when, otherwize)

choose、when、otherwise

有时候，我们不想使用所有的条件，而只是想从多个条件中选择一个使用。针对这种情况，MyBatis 提供了 choose 元素，它有点像 Java 中的 switch 语句。

还是上面的例子，但是策略变为：传入了 “title” 就按 “title” 查找，传入了 “author” 就按 “author” 查找的情形。若两者都没有传入，就返回标记为 featured 的 BLOG（这可能是管理员认为，与其返回大量的无意义随机 Blog，还不如返回一些由管理员精选的 Blog）。

```
<select id="findActiveBlogLike"
    resultType="Blog">
  SELECT * FROM BLOG WHERE state = ‘ACTIVE’
  <choose>
    <when test="title != null">
      AND title like #{title}
    </when>
    <when test="author != null and author.name != null">
      AND author_name like #{author.name}
    </when>
    <otherwise>
      AND featured = 1
    </otherwise>
  </choose>
</select>
```

```
<select id="QueryBlogChoose" parameterType="map" resultType="blog">
        select * from mybatis.blog
        <where>
            <choose>
                <when test="title != null">
                    title = #{title}
                </when>
                <when test="anthor != null">
                    and anthor = #{anthor}
                </when>
                <otherwise>
                    and views = #{views}
                </otherwise>
            </choose>
        </where>
    </select>
```

where

前面几个例子已经方便地解决了一个臭名昭著的动态 SQL 问题。现在回到之前的 “if” 示例，这次我们将 “state = ‘ACTIVE’” 设置成动态条件，看看会发生什么。

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG
  WHERE
  <if test="state != null">
    state = #{state}
  </if>
  <if test="title != null">
    AND title like #{title}
  </if>
  <if test="author != null and author.name != null">
    AND author_name like #{author.name}
  </if>
</select>
```

如果没有匹配的条件会怎么样？最终这条 SQL 会变成这样：

```
SELECT * FROM BLOG
WHERE
```

这会导致查询失败。如果匹配的只是第二个条件又会怎样？这条 SQL 会是这样：

```
SELECT * FROM BLOG
WHERE
AND title like ‘someTitle’
```

这个查询也会失败。这个问题不能简单地用条件元素来解决。这个问题是如此的难以解决，以至于解决过的人不会再想碰到这种问题。

MyBatis 有一个简单且适合大多数场景的解决办法。而在其他场景中，可以对其进行自定义以符合需求。而这，只需要一处简单的改动：

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG
  <where>
    <if test="state != null">
         state = #{state}
    </if>
    <if test="title != null">
        AND title like #{title}
    </if>
    <if test="author != null and author.name != null">
        AND author_name like #{author.name}
    </if>
  </where>
</select>
```

where 元素只会在子元素返回任何内容的情况下才插入 “WHERE” 子句。而且，若子句的开头为 “AND” 或 “OR”，*where* 元素也会将它们去除。

```
<!--对于上面的例子-->
<select id="QueryBlogIF" parameterType="map" resultType="blog">
        select * from mybatis.blog
        <where>
            <if test="title != null">
                and title=#{title}
            </if>
            <if test="anthor != null">
                and anthor=#{anthor}
            </if>
        </where>
    </select>
```

set

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。

用于动态更新语句的类似解决方案叫做 *set*。*set* 元素可以用于动态包含需要更新的列，忽略其它不更新的列。比如：

```
<update id="updateAuthorIfNecessary">
  update Author
    <set>
      <if test="username != null">username=#{username},</if>
      <if test="password != null">password=#{password},</if>
      <if test="email != null">email=#{email},</if>
      <if test="bio != null">bio=#{bio}</if>
    </set>
  where id=#{id}
</update>
```

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。

代码就不演示了，只不过这个是在update里面使用的，和where类似的。。。。。

trim

就是where，set的爹

foreach

```
//查询id=1,2,3....
    List<Blog> QueryBlogForeach(Map map);
```

```
<!--
        select * from mybatis.blog where 1=1 and( id = 1 or id = 2 or id = 3 )
    -->
    <select id="QueryBlogForeach" parameterType="map" resultType="blog">
        select * from mybatis.blog
        <where>
            <foreach collection="ids" item="id" open="and (" close=")"
separator="or">
                id = #{id}
            </foreach>
        </where>
    </select>
```

```
public static void main(String[] args) {
        SqlSession sqlSession = MybatisUtils.getSqlSession();
        BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);

        ArrayList arrayList = new ArrayList();
        arrayList.add(new Integer(1));
        arrayList.add(new Integer(2));
        //arrayList.add(new Integer(3));

        HashMap map = new HashMap();
        map.put("ids",arrayList);

        List<Blog> blogs = mapper.QueryBlogForeach(map);

        sqlSession.close();
    }
```

```
<==    Columns: id, title, anthor, create_time, views
<==        Row: 1, Mybatis, 田小锋, 2021-04-10 13:06:36, 9999
<==        Row: 2, JAVA, 田小锋, 2021-04-10 13:06:36, 1000
<==        Row: 3, SpringMVC, 田小锋, 2021-04-10 13:06:36, 9999
```

```
        Row: 1, Mybatis, 田小锋, 2021-04-10 13:06:36, 9999
        Row: 2, JAVA, 田小锋, 2021-04-10 13:06:36, 1000
```

SQL片段

有的时候，我们的sql语句存在大量的重复，我们需要将它们抽取出来，做成单独的片段，避免了重写的繁琐！！！！

最好是基于单表的@@@@@@@@@@@@

不要存在where标签@@@@@@@@@@@@@@

```
<sql id="if-title-anthor">
        <if test="title != null">
            and title=#{title}
        </if>
        <if test="anthor != null">
            and anthor=#{anthor}
        </if>
</sql>

    <select id="QueryBlogIF" parameterType="map" resultType="blog">
        select * from mybatis.blog
        <where>
            <include refid="if-title-anthor"></include> <!--引用SQL片段-->
        </where>
    </select>


<!--原来的,假如有很多sql里面都需要那一段SQL的话，就需要写很多次了-->
<select id="QueryBlogIF" parameterType="map" resultType="blog">
        select * from mybatis.blog
        <where>
            <if test="title != null">
                and title=#{title}
            </if>
            <if test="anthor != null">
                and anthor=#{anthor}
            </if>
        </where>
    </select>
```

总结

动态SQL就是在拼接SQL语句，我们只要保证sql语句的正确性和规范性，去排列组合就行了

建议：先写出SQL语句，再对应去修改成动态SQL！！！

缓存