1.MyBatis介绍

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（DAO）

下载地址: MyBatis下载地址：https://github.com/mybatis/mybatis-3/releases
使用版本:3.4.5

```
Hibernate 冬眠    全自动框架   SQL语句可以自动生成，不用人工书写SQL！ 笨重

MyBatis          半自动   SQL语句还是需要自己书写，后期有一些插件可以自动生成SQL！ 灵活   定制SQL！
MyBatis Plus
```

ORM概念：Object Ralation Mapping 对象关系映射 框架

> *数据库类和程序中的实体类有对应关系(映射关系)的框架，叫做ORM框架(对象关系映射)*

```
        数据库表------>实体类
•       数据库表中字段----->实体类的属性
•       数据库表中字段的类型----->实体类中属性的类型

数据库表和程序实体类有对应关系的持久层框架  就叫ORM框架！
```

常见的**ORM**框架哪些：

```
MyBatis
Hibernate
Spring Data  JPA
......
```

2.MyBatis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基本上可以实现我们不使用数据访问框架可以实现的所有功能，或许更多。

3.MyBatis基础应用

3.1 搭建MyBatis环境

3.1.1 环境准备

- Jdk环境：jdk1.8
- Ide环境：IDEA
- 数据库环境：MySQL 5.1
- Mybatis：3.4.5

3.1.2 下载MyBatis

mybaits的代码由github.com管理，下载地址：https://github.com/mybatis/mybatis-3/releases
Mybatis-3.4.5.jar：mybatis的核心包

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.bruceliu.mybatis</groupId>
    <artifactId>mybatis-20190902</artifactId>
    <version>1.0-SNAPSHOT</version>

    <!--导入MyBatis开发环境的依赖-->
    <dependencies>

        <!-- myBatis -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.4.5</version>
        </dependency>

        <!-- mysql驱动包 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.38</version>
        </dependency>

        <!-- Junit测试 -->
        <dependency>
```

```
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>

        <!--Lombok依赖-->
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.16.18</version>
            <scope>provided</scope>
        </dependency>

    </dependencies>



</project>
```

3.1.4 准备数据库

```
SET FOREIGN_KEY_CHECKS=0;

-- ----------------------------
-- Table structure for `user`
-- ----------------------------
DROP TABLE IF EXISTS `user`;
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(32) NOT NULL COMMENT '用户名称',
  `birthday` date DEFAULT NULL COMMENT '生日',
  `sex` char(1) DEFAULT NULL COMMENT '性别',
  `address` varchar(256) DEFAULT NULL COMMENT '地址',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=27 DEFAULT CHARSET=utf8;

-- ----------------------------
-- Records of user
-- ----------------------------
INSERT INTO `user` VALUES ('1', '王五', '2018-09-06', '1',
'四川成都');
INSERT INTO `user` VALUES ('10', '张三', '2014-07-10',
'1', '北京市');
INSERT INTO `user` VALUES ('16', '张小明', '2018-09-06',
'1', '河南郑州');
INSERT INTO `user` VALUES ('22', '陈小明', '2018-09-05',
'1', '河南郑州');
INSERT INTO `user` VALUES ('24', '张三丰', '2018-09-13',
'1', '河南郑州');
INSERT INTO `user` VALUES ('25', '陈小明', '2018-09-12',
'1', '河南郑州');
INSERT INTO `user` VALUES ('26', '王五', '2018-09-05',
'0', '河南郑州');
```

3.1.5.创建主配置文件：mybatis-config.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>

    <!-- 引入外部配置文件 -->
    <properties resource="jdbc.properties"/>

    <!--配置Mybatis的参数-->
    <settings>
    <!--调试的时候方便 输出MyBatis的底层SQL语句-->
        <setting name="logImpl" value="STDOUT_LOGGING"/>
    </settings>

    <!-- 配置开发环境 -->
    <environments default="development">
        <!-- 这里可以配置多个多个environment，每个environment
的id不同，可以在environments标签中选择不同environment的id -->
```

```
        <environment id="development">
            <!-- 配置事务管理方式：JDBC，将事务交给JDBX管理（推
荐） -->
            <transactionManager type="JDBC" />
            <!-- 配置数据源，即连接池方式：
JNDI/POOLED/UNPOOLED 后期可以配置为德鲁伊 -->
            <!--数据源的定义可以去看xmind中的：Java核心技术卷II
第五章 数据库编程.xmind中-->
            <dataSource type="POOLED">
                <property name="driver"
value="${jdbc.driver}" />
                <property name="url" value="${jdbc.url}"
/>
                <property name="username"
value="${jdbc.username}" />
                <property name="password"
value="${jdbc.password}" />
            </dataSource>
        </environment>
    </environments>


    <!--映射Mapper文件-->
     <!--引入映射文件-->
    <mappers>
        <mapper
resource="com/bruceliu/mapper/UserMappper.xml"></mapper>
    </mappers>

</configuration>
```

jdbc.properties：（MySQL8+）

```
jdbc.driver=com.mysql.cj.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/travel?
useSSL=false&allowPublicKeyRetrieval=true&serverTimezone=U
TC
jdbc.username=root
jdbc.password=271441
```

3.2 实现MyBatis的查询

3.2.1 获取SqlSession对象(核心对象)

MyBatis框架中涉及到的几个API
SqlSessionFactoryBuilder：该对象负责根据MyBatis配置文件mybatis-config.xml构建SqlSessionFactory实例。

SqlSessionFactory：每一个MyBatis的应用程序都以一个SqlSessionFactory对象为核心。该对象负责创建SqlSession对象实例。

> *它是单个数据库映射关系经过编译后的内存映像，其主要作用是创建SqlSessioin。这个类是线程安全的，它一旦被创建，在整个应用执行期间都会存在。如果我们在一个应用执行期间多次创建同一个数据库的SqlSessionFactory，那么此数据库的资源将容易被耗尽。为了解决这种问题通常一个数据库都只对应一个SqlSessionFactory，所以在构造SqlSessionFactory实例的时候建议使用单例模式。*

SqlSession：该对象包含了所有执行SQL操作的方法，用于执行已映射的SQL语句。

> *它是应用程序与持久层之间进行交互操作的一个单线程对象，其主要作用是执行持久化操作。SqlSession对象包含了数据库中所有执行SQL操作的方法，由于其底层封装了JDBC连接，所以可以直接使用其实例来执行已映射的SQL语句。*
>
> *每一个线程都应该有一个自己的SqlSession实例，并且该实例是不能共享的。同时SqlSession实例也是线程不安全的，因此其使用范围最好在一次请求或一个方法中，绝对不能将其放在一个类的静态字段、实例字段或任何类型的管理范围（如Servlet的HttpSession）中使用。*
>
> *使用完SqlSession对象之后要即使的关闭它，通常可以将其放在finally块中关闭。*

```
//1、读取配置文件
String resource = "SqlMapConfig.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
//2、根据配置文件创建SqlSessionFactory
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
//3、SqlSessionFactory创建SqlSession
SqlSession sqlSession = sqlSessionFactory.openSession();
```

图中字符串resource应为：mybatis-config.xml

3.2.2 创建实体类 User

3.2.2.1 实体类

```
public class User {

    private Integer id;
    private String username;
    private Date birthday;
    private String sex;
    private String address;

    public Integer getId() {
        return id;
    }

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

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public String getSex() {
        return sex;
    }

    public void setSex(String sex) {
        this.sex = sex;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;

    }

    @Override
    public String toString() {
        return "User [id=" + id + ", username=" + username
+ ", birthday=" + birthday + ", sex=" + sex + ", address="
+ address + "]";
    }
}
```

3.2.2.2 数据库字段和程序POJO类的属性不一致问题

在使用SQL语句查询出数据之后，Mybatis底层会使用反射给Map或者List中的Map赋值（在6.1.1.1节中说了，一个自定义的POJO类的实例对象也可以理解为一个Map）。

所以当数据库中的字段和POJO类的属性不一致的时候会有问题，解决这种问题的方式：

1. 在映射文件中人工配置ResultMapper标签
2. 在SQL语句中使用as子句给表中的该字段起一个别名，让这个别名和POJO类中的属性的名称一致

3.2.3 创建接口层 UserMapper

与这个类的继承体系相关的定义都是在搭建Servlet中说的三层架构中的模型层的部分（数据访问层），业务逻辑层（也就是service层）没有搭建

```
/**
 * @ClassName: UserMapper
 * @author: BRUCELIU
 * @date:  2018年9月28日 下午5:17:06
 * @Description:TODO
 */
public interface UserMapper {

    public List<User> getList();
}
```

3.2.4 创建接口实现层 UserMapper实现层实现层(可以不写，另加配置)

> *这个实现类可以被简化，因为如果有多个类似于UserMapper的接口，那么就需要写多个类似于UserMapperImpl的实现类，而这些实现类中的很多代码是重复的，仅需要修改session对象的方法调用操作。*

```
/**
 * @ClassName: UserMapperImpl
 * @author: BRUCELIU
 * @date: 2018年9月28日 下午5:17:46
 * @Description:TODO
 */
public class UserMapperImpl implements UserMapper {

    public List<User> getList() {
        InputStream inputStream = null;
        SqlSessionFactory sqlSessionFactory = null;
        SqlSession session = null;
        try {
            String resource = "mybatis-config.xml"; // 配
置文件
            inputStream =
Resources.getResourceAsStream(resource);
            //获取SqlSessionFactory 数据库连接工厂
            sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
            //获取session，数据库连接对象，类似于Connection
            session = sqlSessionFactory.openSession();
            //执行参数指定位置的SQL语句，namespace+id唯一确定
SQL语句
            List<User> list =
session.selectList("com.bruceliu.dao.UserMapper.getList");
            return list;
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                session.close();
                inputStream.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return null;
    }
}
```

3.2.5 创建接口Mapper文件（映射文件）

```
<?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.bruceliu.dao.UserMapper">

    <!-- 返回值类型 -->
    <select id="getList"
resultType="com.bruceliu.bean.User">
        select * from user
    </select>

</mapper>
```

3.2.6 测试类

```
public class TestMyBatis {

    UserMapperImpl um = new UserMapperImpl();

    @Test
    public void test1() {
        List<User> list = um.getList();
        for (User u : list) {
            System.out.println(u);
        }
    }
}
```

3.2.7.Mybatis使用步骤总结：

1) 创建SqlSessionFactory
2) 通过SqlSessionFactory创建SqlSession对象
3) 通过SqlSession操作数据库
4) 调用SqlSession.commit()提交事务
5) 调用SqlSession.close()关闭会话

封装获取MyBatis中Session的工具类

```
/**
 * @author bruceliu
 * @create 2019-09-02 15:37
 * @description 获取sesison和关闭session的工具类
 */
public class MyBatisUtils {

    /**
     * 01-获取SqlSession
     * @return
     */
    public static SqlSession getSession(){
        SqlSession session=null;
        InputStream inputStream=null;
        try {
            //配置文件的路径
            String resource = "mybatis-config.xml";
            //加载配置文件，得到一个输入流
            inputStream =
Resources.getResourceAsStream(resource);
            //获取MyBatis的Session工厂
            SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
            //通过session工厂获取到一个session（此session非
Servlet中Session，这个Session表示MyBatis框架和数据库的会话信息）
            //获取到session就表示MyBatis连接上数据库啦！！类似于
JDBC中 Connection对象
            session =
sqlSessionFactory.openSession(true);//自动提交事务
            //调用session的查询集合方法
            return session;
        } catch (IOException e) {
            e.printStackTrace();
        }finally {
            try {
                inputStream.close();
            } catch (IOException e) {
```

```
                e.printStackTrace();
            }
        }
        return null;
    }

    /**
     * 02-关闭SqlSession
     * @param session
     */
    public static void closeSession(SqlSession session){
        if(session!=null){
            session.close();
        }
    }
}
```

4.编写DAO的实现类发现的问题

- 冗余的代码多；
- 调用**Statement**不方便；
- 通用性不好，实现方法非常的类似；

4.1 使用动态代理实现接口的实现类（不需要Mapper实现类）

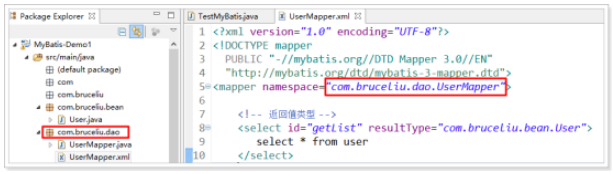
如何得到动态代理：

```
/**
 * @author bruceliu
 * @create 2019-09-02 10:58
 * @description
 */
public class TestMybatis {

    SqlSession session=null;
    UserMappper um=null;

    //在每次调用测试方法之前，先运行@Before修饰的init()方法
    @Before
    public void init(){
        //MyBatis在底层使用动态代理(反射)自动生成Mapper实现类，不
需要人工写实现类!
        session = MyBatisUtils.getSession();
        //um就是Mapper的实现类
        um = session.getMapper(UserMappper.class);
    }

    @Test
    public void testQuery(){
        List<User> list = um.findList();
        for (User user : list) {
            System.out.println(user);
        }
    }

    /*
     * MyBatis增删改 需要手动提交事务
     */
    @Test
    public void testAdd(){
        User u=new User("刘老师",new Date(),"女","日本东
京");
        int count = um.addUser(u);
        System.out.println(count>0?"新增成功":"新增失败");
    }


    //在运行单元测试方法之后谜，运行@After修饰的destory()方法
    @After
    public void destory(){
        MyBatisUtils.closeSession(session);
    }
```

```
}
```

注意：

1、保证命名空间和接口的全路径一致：



2、Statement的id和接口中的方法名一致

3、加入到mybatis-config.xml中

4.2 使用动态代理总结

使用mapper接口不用写接口实现类即可完成数据库操作，使用非常简单，也是官方所推荐的使用方法。

使用mapper接口的必须具备以几个条件：

1) Mapper的namespace必须和mapper接口的全路径一致。

2) Mapper接口的方法名必须和sql定义的id一致。

3) Mapper接口中方法的输入参数类型必须和sql定义的parameterType一致。

4) Mapper接口中方法的输出参数类型必须和sql定义的resultType一致。

5.Mybatis-Config的核心配置（在mybatis-config.xml文件中）

- 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
    - environment 环境变量
        - transactionManager 事务管理器
        - dataSource 数据源
- mappers 映射器

Mybatis的配置文件中配置项是有顺序的，即按照上面的顺序；

5.1 Properties

```
<!-- 引入外部配置文件 -->
<properties resource="jdbc.properties"></properties>
```

5.2.typeAliases（别名）

类型别名是为 Java 类型命名一个短的名字。 它只和 XML 配置有关, 只用来减少类完全限定名的多余部分。

自定义别名：

```
<!--实体类取别名-->
<typeAliases>
    <!--直接给所有的实体类取别名。默认的实体类的别名就是类名(不区分小大写,意思是在写有sql语句的映射文件中引用这个类的时候，使用的别名不区分大小写)
        User实体类：User、user、USER
    -->
    <package name="com.bruceliu.bean"/>
</typeAliases>
```

注意：

使用定义的别名是不区分大小写的，但一般按java规则去使用即可，即user或者User

5.3.mappers

mapper映射文件的引入到Mybatis的核心配置文件mybatis-config.xml中有3种方式：
□路径相对于资源目录跟路径：

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

□使用完整的文件路径：

```
    <mappers>
        <mapper class="com.bruceliu.dao.UserMapper"/>
    </mappers>
```

□可直接配个扫描包：

注意：此种方法要求**mapper**接口名称和**mapper**映射文件名称相同，且放在同一个目录中

```
 <!--引入映射文件-->
    <mappers>
        <!-- <mapper
resource="com/bruceliu/mapper/UserMappper.xml"></mapper>--
>
        <!--<mapper
class="com.bruceliu.mapper.UserMappper"></mapper>-->

        <!--直接映射包的名字，那么这个包下面所有的Mapper接口全部映
射！-->
        <package name="com.bruceliu.mapper"/>

    </mappers>
```

6.Mapper XML 文件

Mapper映射文件是在实际开发过程中使用最多的，也是我们学习的重点。
Mapper文件中包含的元素有：
□cache – 配置给定命名空间的缓存。
□cache-ref – 从其他命名空间引用缓存配置。
□resultMap – 映射复杂的结果对象。
□sql – 可以重用的 SQL 块,也可以被其他语句引用。
□insert – 映射插入语句
□update – 映射更新语句
□delete – 映射删除语句
□select – 映射查询语句

6.1.CRUD

6.1.1.select

```
    <select id="getById" parameterType="int"
resultType="User">
          select * from user where id=#{id}
    </select>
```

select标签叫Statement
id，必要属性，在当前的命名空间下不能重复。
指定输出类型，resultType
parameterType（不是必须）如果不指定，自动识别。

6.1.1.1 查询结果总结

查询操作得到的结果一般是一个Map或者一个List，因为查询出来的数据只能是表中的一行数据或者多行数据，而存储一行数据使用的形式就是Map，以列名为键，以该列下这行数据对应的值为值组成一个键值对，如果得到的结果是表中的多行，那么就是得到多个Map，可以组成一个List。

我们在程序中自定义的一个POJO工具类的实例对象（记作Q）也可以理解为一个Map，Q对象中的字段的名称对应于Map中的键，Q对象中的字段的值对应于Map中的值。

6.1.2. insert

```
    <!-- 增删改返回的都是int值 不用写返回值 -->
    <insert id="addUser">
        INSERT INTO USER VALUES (null,#{username},#
{birthday},#{sex},#{address})
    </insert>
```

insert标签叫Statement
id，必要属性，在当前的命名空间下不能重复。
parameterType （不是必须）如果不指定，自动识别。

6.1.2.1如何获得到自增id

方式一：在insert语句中添加属性useGeneratedKeys开启自增ID、和属性keyProperty执行（详细内容在学校的教材JavaEE企业级应用开发教程第七章中有）

```
    <!-- 增删改返回的都是int值 不用写返回值 -->
    <insert id="addUser" parameterType="User"
useGeneratedKeys="true" keyProperty="id">
        INSERT INTO USER VALUES (null,#{username},#
{birthday},#{sex},#{address})
    </insert>
```

useGeneratedKeys：开启自增长映射
keyProperty：指定id所对应对象中的属性名

当执行完addUser()方法后，其返回值依然是执行sql影响的行数，并不是要获取的自增ID！Mybatis会自动将自增的主键值赋值给对象User的属性id（keyProperty属性指定的），因此你可以通过属性的get方法获得插入的主键值： System.out.println(User.getId())；

方式二：使用标签（详细内容在学校的教材JavaEE企业级应用开发教程第七章中有）
在插入操作完成之前或之后，可以配置标签获得生成的主键的值，获得插入之前还是之后的值，可以通过配置order属性来指定。
LAST_INSERT_ID：该函数是mysql的函数，获取自增后主键的ID，它必须配合insert语句一起使用

```
<!-- 增删改返回的都是int值 不用写返回值 -->
    <insert id="addUser" parameterType="User">

        <selectKey keyProperty="id" resultType="int"
order="AFTER">
            SELECT LAST_INSERT_ID()
        </selectKey>

        INSERT INTO USER VALUES (null,#{username},#
{birthday},#{sex},#{address})
    </insert>
```

6.1.3.update

```
<update id="updateUser" >
        update user set username=#{username},birthday=#
{birthday},sex=#{sex},address=#{address} where id=#{id}
    </update>
```

update标签叫Statement
id，必要属性，在当前的命名空间下不能重复。
parameterType （不是必须）如果不指定，自动识别。

6.1.4.删除

```
    <delete id="deleteUser" parameterType="int">
        delete from user where id=#{id}
    </delete>
```

delete标签叫Statement
id，必要属性，在当前的命名空间下不能重复。
parameterType （不是必须）如果不指定，自动识别。

7. #和$区别(面试题)

在映射文件配置

使用#{xxx}用作占位符的方式：

```
<!--
根据id查询用户，User findById(int id)
    select：配置查询语句
        id：可以通过id找到执行的statement，statement唯一标识
        parameterType:输入参数类型
        resultType:输出结果类型
        #{}:相当于占位符
        #{id}：其中的id可以表示输入参数的名称，如果是简单类型名称可以任意
 -->
<select id="getById" parameterType="int" resultType="User" >
     select * from user where id=#{id}
</select>
```

使用${value}拼接Sql语句进行模糊匹配

```
<!--
根据用户名称来模糊查询用户信息列表：
    ${}:表示拼接sql语句
    ${value}：表示输入参数的名称，如果参数是简单类型，参数名称必须是value
 -->
 <select id="findByUsername" parameterType="java.lang.String"
    resultType="User">
    select * from user where username like '%${value}%'
 </select>
```

在Mybatis的mapper中，参数传递有2种方式，一种是#{}另一种是${}，两者有着很大的区别：

```
#{} 实现的是sql语句的预处理参数，之后执行sql中用?号代替，使用时不需要关注数据类型，Mybatis自动实现数据类型的转换。并且可以防止SQL注入。
${} 实现是sql语句的直接拼接，不做数据类型转换，需要自行判断数据类型。不能防止SQL注入。

总结：
#{} 占位符，用于参数传递。
${}用于SQL拼接。
```

> *SQL注入的场景：网页中的搜索框，用户输入想要搜索的内容就可以查看自己想要看到的东西，如果不对用户输入到搜索框中的内容加以限制，那么就会导致SQL注入的问题，例如：用户在搜索框中输入想要搜索的东西之后，添加一个分号，在添加一个删除数据库的语句。*

> *总结：有关于模糊匹配的方式：*
>
> 1. *在SQL语句中使用like '%${value}%';——>会导致SQL注入，并且没有特殊情况下中括号间的值必须为value（特殊情况指的是：使用8.4节的方式）*
> 2. *在SQL语句中使用like concat('%', #{value} , '%',);——>可以防止SQL注入（详细内容在学校的教材JavaEE企业级应用开发教程第六章中有），如果value是简单类型，那么这个SQL语句中的value也可以用别的标识符来替代*
> 3. *视频中补充的一种方式：*
>   *在SQL语句中使用like #{value};——>可以防止SQL注入，并且value的值的形式是："%一个字符串%"，如果value是简单类型，那么这个SQL语句中的value也可以用别的标识符来替代*

8.parameterType的传入参数

传入类型有三种：

1、简单类型，string、long、integer等
2、Pojo类型，User等，这种对象要有getter、setter方法
3、HashMap类型。

8.1 .传入参数是HashMap类型（传递的参数为一个Map类型的实例对象）

查询需求：

```
 <select id="getUsers" parameterType="map" resultType="User">
     select * from user where birthday between #{startdate} and #{enddate}
 </select>
```

查询测试：

```
@Test
public void test7(){
    HashMap<String, Object> map=new HashMap<String, Object>();
    map.put("startdate", "2018-09-07");
    map.put("enddate", "2018-09-25");
    //注意这里：在调用SQL语句的时候传递的参数是一
    //Map类型，而不是字符串startdate和enddate
    //如果是后者的话，那么在SQL语句中就不能使用这
    //两个字符串作为占位符，而是使用arg0，arg1之
    //类的(例如8.3节)，除非使用8.4节的方式传递参数
    List<User> users = mapper.getUsers(map);
    for (User user : users) {
        System.out.println(user);
    }
}
```

注意：map的key要和sql中的占位符保持名字一致

8.2 分页查询（传递的参数为一个Map类型的实例对象）

查询需求：

```
<!-- 分页：map传参 -->
<select id="selectAuthorByPage" resultType="User">
    SELECT * FROM USER LIMIT #{offset}, #{pagesize}
</select>
```

接口：

```
    /**
     * 根据分页参数查询
     * @param paramList 分页参数
     * @return 分页后的用户列表
     */
    List<User> selectAuthorByPage(Map<String, Object> paramList);
```

测试：

```
    @Test
    public void testSelectAuthorByPage() {

        Map<String, Object> map = new HashMap<String, Object>();
        map.put("offset", 0);
        map.put("pagesize", 2);
        //传递的参数是Map类型的，如果传递的参数是字符串
        //offset和pagesize那么在SQL语句中使用的占位          //符类似于arg0、arg1之类的(例如8.3节)，除
        //非使用8.4节的方式传递参数
        List<User> authorList = mapper.selectAuthorByPage(map);
        for (int i = 0; i < authorList.size(); i++) {
            System.out.println(authorList.get(i));
        }
    }
```

8.3使用参数顺序（使用简单类型并且传递的参数>=2个）

注意：mapper文件中参数占位符的位置编号一定要和接口中参数的顺序保持一致
mapper：

```
    <!-- 分页：传参顺序 -->
    <select id="selectUserByPage3" parameterType="map" resultType="User">
        SELECT * FROM USER LIMIT #{param1},#{param2}
    </select>
```

接口：

```
/**
     * 根据分页参数查询
     * @param offset 偏移量
     * @param pagesize 每页条数
     * @return 分页后的用户列表
     */
    List<User> selectUserByPage3(int offset, int pagesize);
```

测试：

```
@Test
    public void testSelectAuthorByPage3() {
        List<User> users = mapper.selectUserByPage3(1, 1);
        for (int i = 0; i < users.size(); i++) {
            System.out.println(users.get(i));
            System.out.println("----------------------");
        }
    }
```

8.4使用注解（使用简单类型并且传递的参数>=2个）

注意：mapper文件中的参数占位符的名字一定要和接口中参数的注解保持一致
mapper：

```
    <!-- 分页：map传参 -->
    <select id="selectUserByPage2" resultType="User">
        SELECT * FROM USER LIMIT #{offset}, #{pagesize}
    </select>
```

接口：

```
    /**
     * 根据分页参数查询
     * @param offset 偏移量
     * @param pagesize  每页条数
     * @return 分页后的用户列表
     */
    List<User> selectUserByPage2(@Param(value = "offset") int offset, @Param(value = "pagesize") int pagesize);
```

测试：

```
    @Test
    public void testSelectAuthorByPage2() {

        List<User> authorList = mapper.selectUserByPage2(0, 2);

        for (int i = 0; i < authorList.size(); i++) {
            System.out.println(authorList.get(i));
            System.out.println("----------------------");
        }
    }
```

9.返回Map类型查询结果

Mybatis中查询结果集为Map的功能,只需要重写ResultHandler接口，,然后用SqlSession 的select方法,将xml里面的映射文件的返回值配置成 HashMap 就可以了。具体过程如下

9.1 xml文件配置

```
    <resultMap id="resultMap1" type="HashMap">
        <result property="key" column="r1" />
        <result property="value" column="r2" />
    </resultMap>

    <select id="getResult" resultMap="resultMap1">
        select count(*) r1, max(birthday) r2 from user
    </select>
```

接口：

```
public HashMap<String, Object> getResult();
```

测试：

```
    @Test
    public void test8(){
        HashMap<String, Object> result = mapper.getResult();
        System.out.println(result);
    }
```

返回多个值：

```
<select id="getResult" resultType="map">
        select count(*) r1, max(birthday) r2,min(id) r3 from user
</select>
```

10.解决数据库字段和实体类属性不同

在平时的开发中，我们表中的字段名和表对应实体类的属性名称不一定都是完全相同的，下面来演示一下这种情况下的如何解决字段名与实体类属性名不相同的冲突。

上面的测试代码演示当实体类中的属性名和表中的字段名不一致时，使用MyBatis进行查询操作时无法查询出相应的结果的问题以及针对问题采用的两种办法：

- 解决办法一: 通过在查询的sql语句中定义字段名的别名，让字段名的别名和实体类的属性名一致，这样就可以表的字段名和实体类的属性名一一对应上了，这种方式是通过在sql语句中定义别名来解决字段名和属性名的映射关系的。

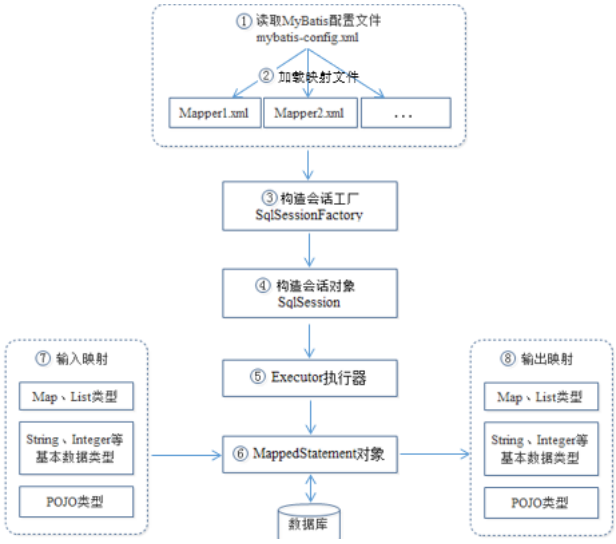
```
<!-- 1.查询所有 -->
<select id="getStudents" resultType="Student">
    SELECT b_id id,b_name b_name,b_sex sex,b_birthday birthday FROM Student
</select>
```

- 解决办法二: 通过来映射字段名和实体类属性名的一一对应关系。这种方式是使用MyBatis提供的解决方式来解决字段名和属性名的映射关系的。

```
 <resultMap type="Student" id="map2">
      <id column="b_id" property="id"/>
      <result column="b_name" property="b_name" />
      <result column="b_sex" property="sex" />
      <result column="b_birthday" property="birthday" />
 </resultMap>

<!-- 1.查询所有 -->
<select id="getStudents" resultMap="map2">
    SELECT * FROM Student
</select>
```


11.MyBatis整体架构

Mybatis是一个类似于Hibernate的ORM持久化框架，支持普通SQL查询，存储过程以及高级映射。Mybatis通过使用简单的XML或注解用于配置和原始映射，将接口和POJO对象映射成数据库中的记录。
由于Mybatis是直接基于JDBC做了简单的映射包装，所有从性能角度来看：
JDBC > Mybatis > Hibernate



1、配置2类配置文件，其中一类是：Mybatis-Config.xml (名字不是写死，随便定义)，另一类：Mapper.xml(多个)，定义了sql片段；
2、通过配置文件得到SqlSessionFactory
3、通过SqlSessionFactory得到SqlSession（操作数据库）
4、通过底层的Executor（执行器）执行sql，Mybatis提供了2种实现，一种是基本实现，另一种带有缓存功能的实现；（这里简化了Executor的详细过程）

12.补充

Maven默认不编译xml文件，所以当项目中xml文件修改之后可能会出现没有效果的情况，这个时候可以在pom.xml文件中的poject标签中增加下面的代码：

```
<build>
    <!--maven编译的时候 顺便把src/main/java包下的xml也编译一下-->
    <resources>
        <resource>
            <directory>src/main/java</directory>
            <includes>
                <include>**/*.xml</include>
            </includes>
        </resource>
        <resource>
            <directory>src/main/resources</directory>
        </resource>
    </resources>
</build>
```