链滴

MyBatis 笔记 - b 站

作者：Gao-Eason

原文链接：https://ld246.com/article/1649679722371

来源网站：链滴

1.什么是三层架构

在项目开发中,遵循的一种形式模式.分为三层.

** 1)界面层:用来接收客 户端的输入,调用业务逻辑层进行功能处理,返回结果给客户端.过去的servlet是界面层的功能.**

** 2)业务逻辑层:用来进行整个项目的业务逻辑处理,向上为界面层提供处理结果,向下问数据访问层要据.**

** 3)数据访问层:专门用来进行数据库的增删改查操作,向上为业务逻辑层提供数据.**

** 各层之间的调用顺序是固定的,不允许跨层访问.****

**** 界面层<------->业务逻辑层<------>数据访问层**

2.生活中的三层架构

3.常用的框架SSM.

** Spring:它是整合其它框架的框架.它的核心是IOC和AOP.它由20多个模块构成.在很多领域都提供很好的解决方案.是一个大佬级别的存在.****

**** SpringMVC:它是Spring家族的一员.专门用来优化控制器(Servlet)的.提供了极简单数据提交,数携带,页面跳转等功能.****

**** MyBatis:是持久化层的一个框架.用来进行数据库访问的优化.专注于sql语句.极大的简化了JDBC访问.**

4.什么是框架

** 它是一个半成品软件.将所有的公共的,重复的功能解决掉,帮助程序快速高效的进行开发.它是可复用可扩展的.**

**5.什么是MyBatis框架 **

** MyBatis 本是 apache 的一个开源项目iBatis, 2010 年这个项目由 apache software foundation 移到了 google code，并且改名为 MyBatis 。2013 年 11 月迁移到 Github,最新版本是 MyBatis 3.5 7 ，其发布时间是 2021 年 4月 7日。****

**** MyBatis完成数据访问层的优化.它专注于sql语句.简化了过去JDBC繁琐的访问机制. **

6.添加框架的步骤

1)添加依赖

2)添加配置文件

** 具体步骤:**

1.新建库建表

2.新建maven项目,选quickstart模板

3.修改目录,添加缺失的目录,修改目录属性

4.修改pom.xml文件,添加MyBatis的依赖,添加mysql的依赖

** **<!--添加MyBatis框架的依赖-->

```xml
<dependency>**
****     <groupId>org.mybatis</groupId>
****     <artifactId>mybatis</artifactId>
****     <version>3.5.6</version>
****   </dependency>
****   <!--添加mysql依赖-->
****   <dependency>
****     <groupId>mysql</groupId>
****     <artifactId>mysql-connector-java</artifactId>
****     <version>5.1.32</version>
****   </dependency>
****   ```**
```

5.修改pom.xml文件,添加资源文件指定

6.在idea中添加数据库的可视化

7.添加jdbc.properties属性文件(数据库的配置)

8.添加SqlMapConfig.xml文件,MyBatis的核心配置文件

9.创建实体类Student,用来封装数据

10.添加完成学生表的增删改查的功能的StudentMapper.xml文件

11.创建测试类,进行功能测试

** StudentMapper.xml文件**

```
  <mapper namespace="zar">
    <!--
     完成查询全部学生的功能
     List<Student> getALL();
        resultType:指定查询返回的结果集的类型,如果是集合,则必须是泛型的类型
        parameterType:如果有参数,则通过它来指定参数的类型
    -->
    <select id="getAll" resultType="com.bjpowernode.pojo.Student" >
      select id,name,email,age
      from student
    </select>

 <!--
  按主键id查询学生信息
  Student getById(Integer id);
 -->
 <select id="getById" parameterType="int" resultType="com.bjpowernode.pojo.Student">
    select id,name,email,age
    from student
    where id=#{id}
 </select>

 <!--
  按学生名称模糊查询
  List<Student> getByName(String name);
 -->
 <select id="getByName" parameterType="string" resultType="com.bjpowernode.pojo.Stude
t">
    select id,name,email,age
    from student
    where name like '%${name}%'
 </select>

 <!--
```

```
  增加学生
  int insert(Student stu);

  实体类:
    private Integer id;
    private String name;
    private String email;
    private Integer age;
 -->
 <insert id="insert" parameterType="com.bjpowernode.pojo.Student">
    insert into student (name,email ,age) values(#{name},#{email},#{age})
 </insert>

 <!--
  按主键删除学生
  int delete(Integer id);
 -->
 <delete id="delete" parameterType="int" >
    delete from student where id=#{id}
 </delete>

 <!--
  更新学生
  int update(Student stu);
 -->
 <update id="update" parameterType="com.bjpowernode.pojo.Student">
    update student set name=#{name},email=#{email},age=#{age}
    where id=#{id}
 </update>

** **</mapper>

** 测试类****
**** @Test****
****   public void testUpdate()throws IOException {****
****       //1.使用流读取核心配置文件****
****       InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml");****
****       //2.创建SqlSessionFactory工厂对象****
****       SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(in);****
****       //3.取出SqlSession****
****       SqlSession sqlSession = factory.openSession();****
****       //4.调用方法****
****       int num = sqlSession.update("zar.update",new Student(3,"hehe","hehe@126.com",30
);****
****       System.out.println(num);****
****       sqlSession.commit();****
****       sqlSession.close();****
****   }**
```

7.3MyBatis对象分析

** 1)Resources类****

**** 就是解析SqlMapConfig.xml文件,创建出相应的对象****

**** InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml");**

** 2)SqlSessionFactory接口****

**** 使用ctrl+h快捷键查看本接口的子接口及实现类****

**** DefaultSqlSessionFactory是实现类****

**** SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(in);**

** 3)SqlSession接口****

**** DefaultSqlSession实现类**

8.为实体类注册别名

** 1)单个注册**

```xml
<typeAlias type="com.bjpowernode.pojo.Student" alias="student"></typeAlias>
**** ```
```

** 2)批量注册**

```xml
** <!--<typeAlias type="com.bjpowernode.pojo.Student" alias="student"></typeAlias>-
>
****      <!--批量注册别名
          别名是类名的驼峰命名法(规范)
          -->
****   <package name="com.bjpowernode.pojo"></package>****
****   ```**
```

9.设置日志输出

```xml
  <!--设置日志输出底层执行的代码-->
   <settings>
      <setting name="logImpl" value="STDOUT_LOGGING"/>
   </settings>
```

10.动态代理存在意义

** 在三层架构中,业务逻辑层要通过接口访问数据访问层的功能.动态代理可以实现.**

** 动态代理的实现规范:**

** 1)UsersMapper.xml文件与UsersMapper.java的接口必须同一个目录下.**

** 2)UsersMapper.xml文件与UsersMapper.java的接口的文件名必须一致,后缀不管.**

** 3)UserMapper.xml文件中标签的id值与与UserMapper.java的接口中方法的名称完全一致.**

** 4)UserMapper.xml文件中标签的parameterType属性值与与UserMapper.java的接口中方法的数类型完全一致.**

** 5)UserMapper.xml文件中标签的resultType值与与UserMapper.java的接口中方法的返回值类型全一致.**

** 6)UserMapper.xml文件中namespace属性必须是接口的完全限定名称com.bjpowernode.mappe.UsersMapper**

** 7)在SqlMapConfig.xml文件中注册mapper文件时,使用class=接口的完全限定名称com.bjpowerode.mapper.UsersMapper.**

11.动态代理访问的步骤

** 1)建表Users****

**** 2)新建maven工程,刷新可视化****

**** 3)修改目录****

**** 4)修改pom.xml文件,添加依赖****

**** 5)添加jdbc.propertis文件到resources目录下****

**** 6)添加SqlMapConfig.xml文件****

**** 7)添加实体类****

**** 8)添加mapper文件夹,新建UsersMapper接口****

**** 9)在mapper文件夹下,新建UsersMapper.xml文件,完成增删改查功能****

**** 10)添加测试类,测试功能**

12.优化mapper.xml文件注册

```
<!--注册mapper.xml文件-->
  <mappers>
     <!--绝对路径注册-->
     <mapper url="/////"></mapper>
     <!--非动态代理方式下的注册-->
     <mapper resource="StudentMapper.xml"></mapper>
     <!--动态代理方式下的单个mapper.xml文件注册-->
     <mapper class="com.bjpowernode.mapper.UsersMapper"></mapper>
     <!--批量注册-->
     <package name="com.bjpowernode.mapper"></package>
  </mappers>
```

13.#{}占位符

** 传参大部分使用#{}传参,它的底层使用的是PreparedStatement对象,是安全的数据库访问 ,防止sq

注入.****

**** #{}里如何写,看parameterType参数的类型**

1)如果parameterType的类型是简单类型(8种基本(封装)+String),则#{}里随便写.

** <select id="getById" parameterType="int" resultType="users"> ==>入参类型是简类型****

**** select id,username,birthday,sex,address****

**** from users****

**** where id=#{zar} =**=**>随便写****

**** ****</select>**** **

**2)parameterType的类型是实体类的类型,则#{}里只能是类中成员变量的名称而且区分大小写. **

```
<insert id="insert" parameterType="users" > ===>入参是实体类
    insert into users (username, birthday, sex, address) values(#{userName},#{birthday},#{sex,#{address}) ==>成员变量名称
</insert>
```

14.**{}里随便写,但是分版本,如果是3.5.1及以下的版本,只以写value.**

```
<select id="getByName" parameterType="string" resultType="users"> ===>入参是简单类型
  select id,username,birthday,sex,address
  from users
  where username like '%${zar}%' ===>随便写
</select>
```

** B. 如果parameterType的类型是实体类的类型,则${}里只能是类中成员变量的名称.(现在已经少用)*

C. 优化后的模糊查询(以后都要使用这种方式)

```
<select id="getByNameGood" parameterType="string" resultType="users">
    select id,username,birthday,sex,address
    from users
    where username like concat('%',#{name},'%')
</select>
```

2)字符串替换

** 需求:模糊地址或用户名查询****

**** select * from users where username like '%小%';****

**** select * from users where address like '%市%'**

```
<!--
  //模糊用户名和地址查询
  //如果参数超过一个,则parameterType不写
  List<Users> getByNameOrAddress(
```

```
        @Param("columnName")  ===>为了在sql语句中使用的名称
        String columnName,
        @Param("columnValue")  ===>为了在sql语句中使用的名称
        String columnValue);
    -->

<select id="getByNameOrAddress" resultType="users">
    select id,username,birthday,sex,address
    from users
    where ${columnName} like concat('%',#{columnValue},'%')  ==>此处使用的是@Param注解
的名称
</select>
```

15.返回主键值

** 在插入语句结束后, 返回自增的主键值到入参的users对象的id属性中.**

```
<insert id="insert" parameterType="users" >
        <selectKey  keyProperty="id" resultType="int" order="AFTER">
            select last_insert_id()
        </selectKey>
        insert into users (username, birthday, sex, address) values(#{userName},#{birthday},#{sex}
#{address})
    </insert>
```

** **<selectKey>标签的参数详解:**

**** **keyProperty: users对象的哪个属性来接返回的主键值**

**** resultType:返回的主键的类型****

**** order:在插入语句执行前,还是执行后返回主键的值**

16.UUID

** 这是一个全球唯一随机字符串,由36个字母数字中划线组.****

**** UUID uuid = UUID.randomUUID();****

**** System.out.println(uuid.toString().replace("-","").substring(20));**

17.什么是动态sql

** 可以定义代码片断,可以进行逻辑判断,可以进行循环处理(批量处理),使条件判断更为简单.****

**** 1)**<sql>**:用来定义代码片断,可以将所有的列名,或复杂的条件定义为代码片断,供使用时调用.****

**** 2)**<include>**:用来引用****<sql>****定义的代码片断.**

```
<!--定义代码片断-->
    <sql id="allColumns">
        id,username,birthday,sex,address
    </sql>
```

** //引用定义好的代码片断**

```
select        from users
```

3)<if>:进行条件判断

** test条件判断的取值可以是实体类的成员变量,可以是map的key,可以是@Param注解的名称.****

**** 4)<where>:进行多条件拼接,在查询,删除,更新中使用.**

```
<select id="getByCondition" parameterType="users" resultType="users">
   select <include refid="allColumns"></include>
   from users
    <where>
       <if test="userName != null and userName != ''">
         and username like concat('%',#{userName},'%')
       </if>
       <if test="birthday != null">
         and birthday = #{birthday}
       </if>
       <if test="sex != null and sex != ''">
         and sex = #{sex}
       </if>
       <if test="address != null and address != ''">
          and address like concat('%',#{address},'%')
       </if>
    </where>
</select>
```

** 5)<set>:有选择的进行更新处理,至少更新一列.能够保证如果没有传值进来,则数据库中的数据保不变.**

```
<update id="updateBySet" parameterType="users">
       update users
       <set>
          <if test="userName != null and userName != ''">
            username = #{userName},
          </if>
          <if test="birthday != null">
             birthday = #{birthday},
          </if>
          <if test="sex != null and sex != ''">
             sex = #{sex},
          </if>
          <if test="address != null and address != ''">
             address =#{address} ,
          </if>
       </set>
       where id = #{id}
    </update>
```

** 6)<foreach>:用来进行循环遍历,完成循环条件查询,批量删除,批量增加,批量更新.****

**** 查询实现****

**** **<select id="getByIds" resultType="users">**

**** select **<include refid="allColumns">**</include>****

**** from users****

**** where id in****

**** <foreach collection="array" item="id" separator="," open="(" close=")">

**** #{id}****

**** </foreach>

**** </select>

**** <foreach>**参数详解:**

**** **collection:用来指定入参的类型,如果是List集合,则为list,如果是Map集合,则为map,如果数组,则为array.**

**** item:每次循环遍历出来的值或对象****

**** separator:多个值或对象或语句之间的分隔符****

**** open:整个循环外面的前括号 ****

**** close:整个循环外面的后括号**

批量删除实现

```
<delete id="deleteBatch" >
   delete from users
   where id in
   <foreach collection="array" item="id" separator="," open="(" close=")">
      #{id}
   </foreach>
</delete>
```

批量增加实现

```
<insert id="insertBatch">
   insert into users(username, birthday, sex, address) values
   <foreach collection="list" item="u" separator="," >
      (#{u.userName},#{u.birthday},#{u.sex},#{u.address})
   </foreach>
</insert>
```

18.指定参数位置

** 如果入参是多个,可以通过指定参数位置进行传参. 是实体包含不住的条件.实体类只能封装住成员量的条件.如果某个成员变量要有区间范围内的判断,或者有两个值进行处理,则实体类包不住.****

**** 例如:查询指定日期范围内的用户信息.****

**** <!--

//查询指定日期范围内的用户

List<Users> getByBirthday(Date begin, Date end);

-->

**** <select id="getByBirthday" resultType="users">

**** select <include refid="allColumns"></include>****

**** from users****

**** where birthday between #{arg0} and #{arg1}****

**** </select> **

**19.入参是map(重点掌握) **

** 如果入参超过一个以上,使用map封装查询条件,更有语义,查询条件更明确.****

**** **<!--
//入参是map
List<Users> getByMap(Map map);
#{birthdayBegin}:就是map中的key
-->

```
<select id="getByMap" resultType="users" >
  select <include refid="allColumns"></include>
  from users
  where birthday between #{birthdayBegin} and #{birthdayEnd}
</select>
```

** 测试类中**

```java
@Test**
**** public void testGetByMap() throws ParseException {****
****  Date begin = sf.parse("1999-01-01");****
****  Date end = sf.parse("1999-12-31");****
****  Map map = new HashMap<>();****
****  map.put("birthdayBegin",begin);****
****  map.put("birthdayEnd", end);****
****  List****<Users>**** list = uMapper.getByMap(map);****
****  list.forEach(users -> System.out.println(users));****
**** }****
**** ```**
```

20.返回值是map

** 如果返回的数据实体类无法包含,可以使用map返回多张表中的若干数据.返回后这些数据之间没有何关系.就是Object类型.返回的map的key就是列名或别名.****

```
**** <!--
//返回值是map(一行)
Map getReturnMap(Integer id);
-->
**** <select id="getReturnMap" parameterType="int" resultType="map">
****  select username nam,address a****
****  from users****
****  where id=#{id}****
**** </select>
```

```
<!--
  //返回多行的map
List<Map> getMulMap();
-->
<select id="getMulMap" resultType="map">
    select username,address
    from users
</select>
```

21.表之间的关联关系

**** 关联关系是有方向的.****

**** 1)一对多关联:一个老师可以教多个学生,多个学生只有一个老师来教,站在老师方,就是一对多关联.***

**** 2)多对一关联:一个老师可以教多个学生,多个学生只有一个老师来教,站在学生方,就是多对一关联.***

**** 3)一对一关联:一个老师辅导一个学生,一个学生只请教一个老师.学生和老师是一对一.****

**** 4)多对多关联:园区划线的车位和园区的每一辆车.任意一个车位可以停任意一辆车.任意一车辆车以停在任意一个车位上.**

22.一对多关联关系

**** 客户和订单就是典型的一对多关联关系.****

**** 一个客户名下可以有多个订单.****

**** 客户表是一方,订单表是多方.客户一中持有订单的集合.****

**** 使用一对多的关联关系,可以满足查询客户的同时查询该客户名下的所有订单.****

```
<!--多出来的一咕噜绑定ordersList


Orders实体类:
private Integer id;
private String orderNumber;
private Double orderPrice;
-->
<collection property="ordersList" ofType="orders">
    <!--主键绑定-->
    <id property="id" column="oid"></id>
    <!--非主键绑定-->
    <result property="orderNumber" column="orderNumber"></result>
    <result property="orderPrice" column="orderPrice"></result>
</collection>

</resultMap>
<select id="getById" parameterType="int" resultMap="customermap">
    select c.id cid,name,age,o.id oid,orderNumber,orderPrice,customer_id
    from customer c left  join orders o on c.id = o.customer_id
    where c.id=#{id}
</select>
**</mapper>**
```

23.多对一关联关系.

**** 订单和客户就是多对一关联.****

**** 站在订单的方向查询订单的同时将客户信息查出.****

**** 订单是多方,会持有一方的对象.客户是一方.****

**** <mapper namespace="com.bjpowernode.mapper.OrdersMapper">

**** **<!--
//根据主键查询订单,并同时查询下此订单的客户信息
Orders getById(Integer id);
-->

```
<!--
 手工绑定数据
 实体类
   private Integer id;
   private String orderNumber;
   private Double orderPrice;

   //关联下此订单的客户信息,多方持有一方的对象
   private Customer customer;
-->
<resultMap id="ordersmap" type="orders">
   <!--主键绑定-->
   <id property="id" column="oid"></id>
   <!--非主键绑定-->
   <result property="orderNumber" column="orderNumber"></result>
   <result property="orderPrice" column="orderPrice"></result>
   <!--多出来的一咕噜绑定
      private Integer id;
      private String name;
      private Integer age;

      //该客户名下的所有订单的集合,一方持有多方的集合
      private List<Orders> ordersList; //不用管
   -->
   <association property="customer" javaType="customer">
      <id property="id" column="cid"></id>
      <result property="name" column="name"></result>
      <result property="age" column="age"></result>

   </association>
</resultMap>
<select id="getById" parameterType="int" resultMap="ordersmap">
   select o.id oid,orderNumber,orderPrice,customer_id,c.id cid,name,age
   from orders o inner join customer c on o.customer_id = c.id
   where o.id=#{id}
</select>
```

** </mapper**

24.一对一关联

25.多对多关联

总结：无论是什么关联关系，如果某方持有另一方的集合，则使用<collection>标签完成射，如果某方持有另一方的对象，则使用<association>**标签完成映射。**

26.事务

** 多个操作同时完成,或同时失败称为事务处理.****

**** 事务有四个特性:一致性,持久性,原子性,隔离性.**

** 下订单的业务:****

**** 1)订单表中完成增加一条记录的操作****

**** 2)订单明细表中完成N条记录的增加****

**** 3)商品数据更新(减少)****

**** 4)购物车中已支付商品删除****

**** 5)用户积分更新(增加)**

** 在MyBatis框架中设置事务****

**** <transactionManager type="JDBC"></transactionManager>**** ====>程序员自己控处理的提交和回滚**

** 可设置为自动提交****

**** sqlSession = factory.openSession(); ==>默认是手工提交事务,设置为false也是手工提交事务如果设置为true,则为自动提交.****

**** sqlSession = factory.openSession(true); =**=**>设置为自动提交,在增删改后不需要commit();**

27.缓存

** MyBatis框架提供两级缓存,一级缓存和二级缓存.默认开启一级缓存.**

** 缓存就是为了提交查询效率.**

** 使用缓存后,查询的流程:****

**** 查询时先到缓存里查,如果没有则查询数据库,放缓存一份,再返回客户端.下次再查询的时候直接缓存返回,不再访问数据库.如果数据库中发生commit()操作,则清空缓存.**

** 一级缓存使用的是SqlSession的作用域,同一个sqlSession共享一级缓存的数据.****

**** 二级缓存使用的是mapper的作用域,不同的sqlSession只要访问的同一个mapper.xml文件,则共二级缓存作用域.**

28.什么是ORM

ORM(Object Relational Mapping):对象关系映射

MyBatis框架是ORM非常优秀的框架.

java语言中以对象的方式操作数据,存到数据库中是以表的方式进行存储,对象中的成员变量与表中的之间的数据互换称为映射.整个这套操作就是ORM.

持久化的操作：将对象保存到关系型数据库中 ,将关系型数据库中的数据读取出来以对象的形式封装

MyBatis是持久化层优秀的框架.