**动态 SQL**

根据条件的不同, SQL 语句也会随之动态的改变. MyBatis 中,提供了一组标签用于实现动态 SQL.

**1.1 <if>**

用于进行条件判断, test 属性用于指定判断条件. 为了拼接条件, 在 SQL 语句后强行添加 1=1 的恒成立条件.

|  |
| --- |
| <**select id="selectBy" resultType="Users"**>  select *\** from users where 1=1  <**if test="uid != 0"**>  and uid = #{uid}  </**if**>  <**if test="username != null and username != ''"**>  and username = #{username}  </**if**>  <**if test="password != null and password !='' "**>  and password =#{password}  </**if**>  <**if test="city != null and city !='' "**>  and city =#{city}  </**if**> </**select**> |
| @Test  **public void** selBy(){  Users u = **new** Users();  u.setUid(22); *// u.setUsername("陈坤"); // u.setPassword("123"); // u.setCity("上海");* List<Users> users = **userMapper**.selectBy(u);  System.***out***.println(users);  **session**.close();  } |

**1.2 <where>**

用于管理 where 子句. 有如下功能:

1. 如果没有条件, 不会生成 where 关键字
2. 如果有条件, 会自动添加 where 关键字
3. 如果条件中有 and, 自动去除之,但不会自动加

|  |
| --- |
| <**select id="selectBy2" resultType="Users"**>  select *\** from users  <**where**>  <**if test="uid != 0"**>  and uid = #{uid}  </**if**>  <**if test="username != null and username != ''"**>  and username = #{username}  </**if**>  <**if test="password != null and password !='' "**>  and password =#{password}  </**if**>  <**if test="city != null and city !='' "**>  and city =#{city}  </**if**>  </**where**> </**select**> |
| @Test  **public void** selBy2(){  Users u = **new** Users(); *// u.setUid(22);* u.setUsername(**"陈坤"**);  u.setPassword(**"123"**); *// u.setCity("上海");* List<Users> users = **userMapper**.selectBy2(u);  System.***out***.println(users);  **session**.close();  } |

* 1. **<choose><when><otherwise>(了解)**

**单条件查询(只取其一)**

这是一套标签, 功能类似于 if..else if...else if...else

<select id=*""*>

<where>

<choose>

<when test=*""*>

</when>

<when test=*""*>

</when>

<otherwise>

</otherwise>

</choose>

</where>

</select>

when中取其中的一个,不会追加多个条件,如果有多个条件,只取最前面的那个,如果都不满足不查数据了

|  |
| --- |
| <**select id="selectBy3" resultType="Users"**>  select *\** from users  <**where**>  <**choose**>  <**when test="uid != 0"**>  uid=#{uid}  </**when**>  <**when test="username != null and username != ''"**>  username = #{username}  </**when**>  <**when test="password != null and password != ''"**>  password = #{password}  </**when**>  <**otherwise**>  city = #{city}  </**otherwise**>  </**choose**>  </**where**> </**select**> |
| @Test  **public void** selBy3(){  Users u = **new** Users(); *// u.setUid(22);* u.setUsername(**"陈坤"**);  u.setPassword(**"123"**); *// u.setCity("上海");* List<Users> users = **userMapper**.selectBy3(u);  System.***out***.println(users);  **session**.close();  } |

**1.2 <set>**

用于维护 update 语句中的 set 子句. 功能如下:

1. 满足条件时, 会自动添加 set 关键字
2. 会去除 set 子句中多余的逗号
3. 不满足条件时, 不会生成 set 关键字

<update>

<set>

<if test=*""*>

</if>

<if test=*""*>

</if>

</set>

</update>

|  |
| --- |
| <**update id="updateBy"**>  update users  <**set**>  <**if test="username != null and username != ''"**>  username = #{username},  </**if**>  <**if test="password != null and password !='' "**>  password =#{password},  </**if**>  <**if test="city != null and city !='' "**>  city =#{city}  </**if**>  </**set**>  where uid = #{uid} </**update**> |

**1.2 <bind>**

用于对数据进行再加工, 用于模糊查询

|  |
| --- |
| *<!--模糊查询方式一:使用bind标签对city中的数据作字符串拼接加工-->* <**select id="selectLike" resultType="Users"**>  <**bind name="city" value="'%'+city+'%'"**/>  select *\** from users where city like #{city} </**select**>  *<!--模糊查询方式二-->* <**select id="selectLike2" resultType="Users"**>  select \* from users where city like concat('%',#{city},'%') </**select**> |
| concat(,,)是一个字符串处理函数 |

**<foreach>**

**在一个范围中查询**

用于在 SQL 语句中遍历集合参数, 在 in 查询中使用

1. collection: 待遍历的集合
2. open: 设置开始符号
3. separator: 项目分隔符
4. close: 设置结束符号
5. item: 迭代变量

<**foreach collection="" open="" separator="" close="" item=""**></**foreach**>

|  |
| --- |
| *<!--在一个条件范围中查询-->* <**select id="selIn" resultType="Users"**>  select *\** from users where uid in  <**foreach collection="list" open="(" separator="," close=")" item="i"**>  #{i}  </**foreach**> </**select**> |
| @Test **public void** selIn(){  ArrayList<Integer> list = **new** ArrayList<>();  list.add(16);  list.add(18);  list.add(21);  List<Users> users = **userMapper**.selIn(list);  System.***out***.println(users); } |

**批量插入数据**

***面试中可能会被问***

|  |
| --- |
| *<!-- 等同于 for(Users u:list){}-->* <insert id="addBash">  insert into users values  <foreach collection="list" separator="," item="u">  (null,#{u.username},#{u.password},#{u.city})  </foreach> </insert> |
| */\*\*  \* 批量插入数据\*\*\*\*\*(面试中可能会被问),mysql默认能够接收sql语句大小是1M左右,不能超过1M(51万个中文字符)  \*/* @Test public void addBash(){  List<Users> list = new ArrayList<>();   list.add(new Users("阿里","123","武汉"));  list.add(new Users("阿毛","321","孝感"));  list.add(new Users("阿东","345","苏州"));  list.add(new Users("阿莲","543","西安"));  userMapper.addBash(list);  sqlSession.commit();  sqlSession.close();  } |

**<sql><include>**

动态选择要查询的列

<sql>用于提取 SQL 语句, <include>用于引用 SQL 语句

<sql id=*""*>

列名1,列名2,列名3,...

</sql>

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

select

<include refid=*"列名2"*/>

from ...

</select>

|  |
| --- |
| <**sql id="ss"**>  username,password </**sql**> <**sql id="ss2"**>  city </**sql**> <**select id="selSame" resultType="Users"**>  select  <**include refid="ss2"**></**include**>  from users where uid = #{uid} </**select**> |
| @Test **public void** selSame(){  List<Users> users = **userMapper**.selSame(21);  System.***out***.println(users); } |

sql的注解开发

|  |
| --- |
| package com.mybatis.mapper;  import com.mybatis.pojo.Users; import org.apache.ibatis.annotations.Delete; import org.apache.ibatis.annotations.Insert; import org.apache.ibatis.annotations.Select; import org.apache.ibatis.annotations.Update;  import java.util.List;  */\*\*  \* @author: 享学课堂-SaiLing老师  \* @老师qq: 2408524688  \* @description:  \*/* public interface UserMapper {  */\*\*  \* 注解式的开始,推荐单表中使用  \* 在sql优化中,不提倡多表关联查  \* 性能不如单表查  \* @param users  \* @return  \*/* @Select("select \* from users where city=#{city}")  public List<Users> selBy2(Users users);   @Delete("delete from users where uid = #{param1}")  public int delete(long uid);   @Insert("insert into users values(null,#{username},#{password},#{city})")  public int addUsers(Users u);   @Update("update users set password = #{password} where uid = #{uid}")  public int update(Users u);  } |

**MyBatis 的缓存机制**

1. 缓存用于提高查询的效率.
2. MyBatis 的缓存是使用 SQL 标签的 ID 作为缓存的唯一标识的. 执行相同的标签可以使用缓存. 不同的标签不能使用缓存.
3. MyBatis 中有两种缓存机制.

**1.1** **一级缓存**

1. 默认开启. 线程级别的缓存, SqlSession 的缓存
2. 在一个 SqlSession 生命周期中有效. SqlSession 关闭,

缓存清空.

|  |
| --- |
| **public class** Demo {  SqlSession **session**;  UserMapper **userMapper**;  **public** Demo(){  **try** {  *//加载applicationContext-dao.xml文件* InputStream in = Resources.*getResourceAsStream*(**"applicationContext-dao.xml"**);  *//通过配置文件创建连接数据库的工厂类* SqlSessionFactory factory = **new** SqlSessionFactoryBuilder().build(in);*//底层由子类DefaultSqlSessionFactory创建  //通过factory获取发送sql语句的对象* **session** = factory.openSession();*//底层用DefaultSqlSession创建的* **userMapper** = **session**.getMapper(UserMapper.**class**);*//从这里传入字节码对象,可以看出,底层使用反射创建接口的对象* } **catch** (IOException e) {  e.printStackTrace();  }  }    @Test  **public void** selSame(){  List<Users> users = **userMapper**.selSame(24);  System.***out***.println(**"是从数据库中查询:"**+users);*//首次查询的数据会放入SqlSession对象中缓存起来 // session.close(); //关闭sessoin,缓存就会被清空* System.***out***.println(**"---------查询相同的数据时----------"**);  List<Users> users2 = **userMapper**.selSame(21);  System.***out***.println(**"从缓存中查的数据:"**+users2);  *//缓存的周期是SqlSession的会话周期* } |

**1.2** **二级缓存**

1. 进程级别的缓存, SqlSessionFactory 的缓存
2. 在一个 SqlSessionFactory 生命周期中有效. 可以在多个 SqlSession 生命中周期中共享.
3. 默认关闭, 需要使用的时候, 要为某个命名空间开启二级缓存(在 mapper.xml 中配置<cache>).

1.在applicationContext-dao.xml配置文件中加

<**setting name="cacheEnabled" value="true"**/> 也可以不加  
 2.在\*\*\*\*mapper.xml中**第一行**加<cache></cache>

|  |
| --- |
| @Test **public void** selSame(){  List<Users> users = **userMapper**.selSame(24);  System.***out***.println(**"是从数据库中查询:"**+users);*//首次查询的数据会放入SqlSession对象中缓存起来* **session**.close(); *//关闭sessoin,缓存就会被清空* System.***out***.println(**"---------查询相同的数据时----------"**);  SqlSession session2 = **factory**.openSession();  UserMapper userMapper2 = session2.getMapper(UserMapper.**class**);  List<Users> users2 = userMapper2.selSame(24);  System.***out***.println(**"从缓存中查的数据:"**+users2);  *//缓存的周期是SqlSession的会话周期* } |
| *<?***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**>  *<!--从外面src目录下加载含有数据库连接的配置参数文件-->* <**properties resource="db.properties"**></**properties**>  *<!--开启MyBatis的二级缓存-->* <**settings**>  <**setting name="cacheEnabled" value="true"**/>  </**settings**>   *<!--给\*\*\*\*Maper.xml文件中的返回值类型取简短的别面-->* <**typeAliases**>  <**typeAlias type="com.xiangxue.pojo.Users" alias="Users"**></**typeAlias**>  </**typeAliases**>  <**environments default="development"**>  <**environment id="development"**>  <**transactionManager type="JDBC"**/> *<!--当前的事务沿用JDBC的事务逻辑-->* <**dataSource type="POOLED"**> *<!--表示当前使用连接池技术-->* <**property name="driver" value="${driver}"**/>  <**property name="url" value="${url}"**/>  <**property name="username" value="${username}"**/>  <**property name="password" value="${password}"**/>  </**dataSource**>  </**environment**>   </**environments**>  *<!--关联存有sql语句的xml映射文件-->* <**mappers**>  <**mapper resource="com/xiangxue/mapper/UserMapper.xml"**/>  </**mappers**> </**configuration**> |
| *<?***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.xiangxue.mapper.UserMapper"**>  *<!--写在第一行-->*  **<cache></cache>**  <**select id="selSame" resultType="Users"**>  select  <**include refid="ss2"**></**include**>  from users where uid = #{uid}  </**select**> </**mapper**> |

微服务分布式系统中用redis(MongoDB)做缓存使用

**数据库中列名和pojo中属性名不一致问题**

如果查询时使用 resultType 属性, 表示采用 MyBatis 的Auto-Mapping(自动映射)机制, 即相同的列名和属性名会自动匹配. 因此, 当数据库表的列名和类的属性名不一致时,会导致查不到数据. 解决该问题可以有两种方式:

**1.1 列别名**

查询时, 可以通过列别名的方式将列名和属性名保持一致,继续使用自动映射, 从而解决该问题. 但是较为麻烦.

**1.2 使用<resultMap>**

<resultMap>用于自定义映射关系, 可以由程序员自主制定

列名和属性名的映射关系. 一旦使用 resultMap, 表示不再

采用自动映射机制.

|  |
| --- |
| *<?***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.xiangxue.mapper.UserMapper"**>  *<!-- id标签用来描述主键;result描述非主键;property是pojo实体类中的属性名;column是数据库中的列名-->* <**resultMap id="mm" type="Users"**>  <**id property="id" column="uid"**></**id**>  <**result property="uname" column="username"**></**result**>  <**result property="pwd" column="password"**></**result**>  *<!-- 如果有属性中存在与数据库列同名的,可以省略-->* <**result property="city" column="city"**></**result**>  </**resultMap**>  <**select id="selectBy2" resultMap="mm"**>  select *\** from users where uid = #{id}  </**select**>  </**mapper**> |
| **package** com.xiangxue.pojo;   **import** java.io.Serializable;  **public class** Users **implements** Serializable {  *//作为MyBatis查询结果映射的对象实体列(封装数据库中所有的列数据的类),属性名如果与数据库的列名不一致* **private long id**;  **private** String **uname**;  **private** String **pwd**;  **private** String **city**;    **public long** getId() {  **return id**;  }   **public void** setId(**long** id) {  **this**.**id** = id;  }   **public** String getUname() {  **return uname**;  }   **public void** setUname(String uname) {  **this**.**uname** = uname;  }   **public** String getPwd() {  **return pwd**;  }   **public void** setPwd(String pwd) {  **this**.**pwd** = pwd;  }   **public** String getCity() {  **return city**;  }   **public void** setCity(String city) {  **this**.**city** = city;  }   @Override  **public** String toString() {  **return "Users{"** +  **"id="** + **id** +  **", uname='"** + **uname** + **'\''** +  **", pwd='"** + **pwd** + **'\''** +  **", city='"** + **city** + **'\''** +  **'}'**;  } } |