## [MyBatis的动态SQL](http://www.mybatis.cn/archives/52.html)

**MyBatis还有一个强大特性就是它的动态SQL。在实际项目开发中，经常需要根据不同条件拼接SQL语句，拼接时还要确保不能忘了必要的空格，有时候还要注意省掉列名列表最后的逗号，等等。在使用JDBC 或其他类似持久层框架操作数据库时，处理这种情况是非常麻烦的，甚至可以用痛苦来形容，而在MyBatis中利用动态SQL这一特性可以很简单地解决这个问题。**

**动态SQL元素和使用JSTL或其他类似基于XML的文本处理器相似，MyBatis采用功能强大的基于OGNL的表达式来完成动态SQL。OGNL 的表达式可以被用在任意的SQL 映射语句中。**

**常用的动态SQL 元素包括:**

**if**

**choose (when、otherwise)**

**where**

**set**

**foreach**

**bind**

**if 语句**

动态SQL通常会做的事情是有条件地包含where子句的一部分。比如:

<mapper namespace="cn.mybatis.mapper.EmployeeMapper">

<select id="selectEmployeeByIdLike" resultTypem"cn.mybatis.domain.Employee">

SELECT \* FROM tb\_employee WHERE state = 'ACTIVE'

<!-- 可选条件，如果传进来的参数有id属性，则加上id查询条件-->

<if test="id != null">

and id = #{id}

</if>

</select>

</mapper>

以上语句提供了一个可选的根据id查找Employee的功能。如果没有传入id,那么所有处于“ACTIVE”状态的Employee都会被返回; 反之若传入了id，那么就会把查找id内容的Employee结果返回。

如果想通过两个条件搜索该怎么办呢? 很简单，只要多加入一个条件即可。

<select id="selectEmgloyeeByLoginLike" resultType="cn.mybatis.domain.Employee">

SELECT \* FROM tb\_employee WHERE state = 'ACTIVE'

<!-- 两个可选条件，例如登录功能的登录名和密码查询-->

<if test="loginname != null and password != null">

and loginname = #{loginname} and password = #{password)

</if>

</select>

**choose(when、otherwise)语句**

有些时候，我们不想用所有的条件语句，而只想从中择其一二。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，但是这次变为提供了id就按id查找，提供了loginname 和password就按loginnam 和password查找，若两者都没有提供，就返回所有sex等于男的Employee。

<select id="selectEmployeeChoose" parameterType="hashmap" resultType="my.mybatis.domain.Employee">

SELECT \* FROM tb employee WHERE state = 'ACTIVE'

<!--- 如果传入了id，就根据id查询，没有传入id就根据loginname和password查询，否则查询sex等于男的数据-->

<choose>

<when test="id != null">

and id= #{id}

</when>

<when test="loginname != null and password != null">

loginname = #{loginname} and password = #{password}

and

</when>

<otherwise>

and sex = '男'

</otherwise>

</choose>

</select>

**where语句**

前面几个例子已经很好地解决了动态SQL问题。现在回到之前的if示例，这次我们将state="ACTIVE"也设置成动态的条件，看看会发生什么。

<select id="selectEmployeeByIdLike" resultType="cn.mybatis.domain.Emplyee">

SELECT \* FROM tb\_employee WHERE

<if test="state != null ">

state= #{state}

</if>

<if test="id != null ">

and id = #{id}

</if>

</select>

如果传入state参数，则执行正常。如果没有传入参数，则会执行sql语句:  
SELECT \* EROM tb\_employee WHERE  
如果只是传入id，则会执行sql语句:  
SELECT \* FROM tb\_ employee WHERE and id = ?  
也就是说，如果没有传入state参数，会导致执行失败。这个问题不能简单地用条件语句来解决。MyBatis有一个简单的处理方法，只要简单地修改就能得到想要的效果:

<select id="selectEmployeeLike" resultType="cn.mybatis.domain.Employee">

SELECT \* EROM tb\_employee

<where>

<if test="state != null ">

state = #{state}

</if>

<if test="id != null ">

and id = #{id}

</if>

<if test="loginname != null and password != null">

and loginname = #{loginname} and password = #{password}

</if>

</where>

</select>

where元素知道只有在一个以上的if条件有值的情况下才去插入WHERE子句。而且，若最后的内容是“AND”或“OR”开头，则where元素也知道如何将它们去除。

**set元素**

关于动态更新语句还可以使用set元素。set元素可以被用于动态包含需要更新的列，而舍去其他的。

<!-- 根据id查询员工信息-->

<select id="selectEmployeewithId" parameterType="int" resultType="cn.mybatis.domain.Employee">

SELECT \* FROM tb\_employee where id = #{id}

</select>

<!-- 动态更新员工信息-->

<update id="updateEmployeeIfNecessary" parameterType="cn.mybatis.domain.Employee">

update tb\_employee

<set>

<if test="loginname != null">loginname=#{loginname} ,</if>

<if test="password != null">password=#{password} ,</if>

<if test="name != null">name=#{name},</if>

<if test="sex != null">sex=#{sex},</if>

<if test="age != null">age=#{age},</if>

<if test="phone != null">phone=#{phone} ,</if>

<if test="sal != null">sal=#{sal},</if>

<if test="state != null">state=#{state}</if>

</set>

where id=#{id}

</update>

set元素会动态前置SET关键字，同时也会消除无关的逗号，因为使用了条件语句之后很可能就会在生成的赋值语句的后面留下这些逗号。

**foreach元素**

关于动态SQL另外一个常用的操作就是需要对一个集合进行遍历，通常发生在构建IN条件语句时。

<select id="selectEmployeeIn" resultType="cn.mybatis.domain.Employee">

SELECT \* FROM tb\_employee WHERE ID in

<foreach item="item" index="index" collection="list"

open="(" separator="," close=") ">

#{item}

</foreach>

</select>

foreach元素的功能非常强大，它允许指定一个集合，声明可以用在元素体内的集合项和索引变量。它也允许指定开闭匹配的字符串以及在迭代中间放置分隔符。这个元素是很智能的，

因此它不会随机地附加多余的分隔符。

**bind元素**

bind元素可以从OGNL表达式中创建一个变量并将其绑定到上下文。

<select id="selectEmployeeLikeName" resultType="cn.mybati.domain.Employee">

<bind name="pattern" value="'%'+\_parameter.getName() + '%'" />

SELECT \* EROM tb\_employee

WHERE loginname LIKE #{pattern}

</select>

**trim元素**

trim标记是一个格式化的标记，可以完成select，update，insert语句的格式化操作。trim元素的主要功能有四个：

（1）可以在包含的内容前加上某些前缀，与之对应的属性是prefix；

（2）也可以在包含的内容后加上某写后缀，与之对应的属性是suffix；

（3）可以把包含内容的首部某些内容覆盖，即忽略，对应的属性是prefixOverrides

（4）也可以把包含内容的尾部的某些内容覆盖，对应的属性是suffixOverrides。

以下举例说明一下trim元素的应用场景：

例子1：

select \* from user

<trim prefix="WHERE" prefixoverride="AND|OR">

　　<if test="name != null and name.length()>0"> AND name=#{name}</if>

　　<if test="gender != null and gender.length()>0"> AND gender=#{gender}</if>

</trim>

假如说name和gender的值都不为null的话打印的SQL为：

select \* from user where name = 'xx' and gender = 'xx'

where后不存在and，这是因为prefixOverrides="AND|OR"代表去掉第一个and或者是or。

例子2：

update user

<trim prefix="set" suffixoverride="," suffix=" where id = #{id} ">

　　<if test="name != null and name.length()>0"> name=#{name} , </if>

　　<if test="gender != null and gender.length()>0"> gender=#{gender} , </if>

</trim>

假如说name和gender的值都不为null的话打印的SQL为：

update user set name='xx', gender='xx' where id='x'

在gender='xx'后面不存在逗号，而且自动加了一个set前缀和where后缀。

例子3：

INSERT INTO S\_NOTICE

<trim prefix="(" suffix=")" suffixOverrides=",">

ID,

<if test="title != null">TITLE,</if>

<if test="content != null">CONTENT,</if>

<if test="noticeStatus != null">NOTICE\_STATUS,</if>

<if test="createdBy != null">CREATED\_BY,</if>

CREATED\_TS,

<if test="lastUpdBy != null">LAST\_UPD\_BY,</if>

LAST\_UPD\_TS,

</trim>

<trim prefix="values (" suffix=")" suffixOverrides=",">

SYS\_GUID(),

<if test="title != null">#{title,jdbcType=VARCHAR},</if>

<if test="content != null">#{content,jdbcType=VARCHAR},</if>

<if test="noticeStatus != null">#{noticeStatus,jdbcType=VARCHAR},</if>

<if test="createdBy != null">#{createdBy,jdbcType=VARCHAR},</if>

systimestamp,

<if test="lastUpdBy != null">#{lastUpdBy,jdbcType=VARCHAR},</if>

systimestamp,

</trim>

## [MyBatis的关联关系](http://www.mybatis.cn/archives/40.html)

**一对一关联关系**

在实际项目开发中，经常存在一对一关系，比如一个人只能有一个身份证，一个身份证只能给一个人使用，这就是一对一的关系。一对一关系推荐使用唯一主外键关联，即两张表使用外键关联关系，由于是一对一关联，因此还需要给外键列增加unique 唯一约束。

人和身份证是一对一的关系，即一个人只有一个身份证。在Person类中定义了一个card属性，该属性是一个Card类型，用来映射一对一的关联关系，表示这个人的身份证。

再接下来是XML映射文件。

<mapper namespace="cn.mybatis.mapper.CardMapper">

<!-- 根据id查询Card,返回Card对象-->

<select id="selectCardById" parameterType="int" resultType="cn.mybatis.domain.Card">

SELECT \* from tb\_card where id = #{id}

</select>

</mapper>

<mapper namespace="cn.mybatis.mapper.PersonMapper">

<!-- 根据id查询Person,返回resultMap-->

<select id="selectPersonById" parameterType="int" resultMap="personMapper">

SELECT \* from tb\_person where id = #{id}

</select>

<!-- 映射Peson 对象的resultMap-->

<resultMap id="personMapper" type="cn.mybatis.domain.Person">

<id property="id" column="id"/>

<result property="name" column= "name" />

<result property="sex" column="sex"/>

<result property="age" column="age"/>

< !--一对一关联映射:association-->

<association property="card" column="card\_id" select="cn.mybatis.mapper.CardMapper.selectCardById"

javaType="cn.mybatis.domain.Card"/>

</resultMap>

</mapper>

在PersonMapper.xml中定义了一个<select.../>，其根据id查询Person信息，由于Person类除了简单的属性id、name、sex和age之外，还有一个关联对象card，所以返回的是一个名为personMapper的resultMap。personMapper中使用了<association .../>元素映射一对一的关联关系，

select属性表示会使用column属性的card\_id值作为参数执行CardMapper中定义的selectCardById查询对应的Card数据，查询出的数据将被封装到property表示的card对象当中。

我们通常都是使用SqlSession对象调用insert，update，delete和select方法进行测试。实际上，Mybatis官方手册建议通过mapper接口的代理对象访问mybatis，该对象关联了SqlSession对象，开发者可以通过该对象直接调用方法操作数据库。下面定义一个mapper接口对象，需要注意的是，mapper接口对象的类名必须和之前的XML文件中的mapper的namespace一致，而方法名和参数也必须和XML文件中的<select.../>元素的id属性和parameterType属性一致。

<mpper namespace="cn.mybatis.mapper.PersonMapper">

<select id="selectPersonById" parameterType="int".../>

< /mapper>

**一对多关联关系**

在实际项目开发中，一对多是非常常见的关系，比如，一个班级可以有多个学生，一个学生只能属于一个班级，班级和学生是一对多的关系，而学生和班级是多对一的关系。数据库中一对多关系通常使用主外键关联，外键列应该在多方，即多方维护关系

学生和班级是多对一的关系，即一个学生只属于一个班级。在Student类当中定义了一个clazz属性，该属性是一个Clazz类型，用来映射多对一的关联关系，表示该学生所属的班级。

再接下来是XML映射文件。

<mapper namespace="cn.mybatis.mapper.clazzMapper">

<!-- 根据id查询班级信息，返回resultMap-->

<select id="selectClazzById" parameterType="int" resultMap="clazzResultMap">

SELECT \* FROM tb\_clazz WHERE id = #{id}

</select>

<!-- 映射Clazz对象的resultMap-->

<resuItMap id="clazzResultMap" type="cn.mybatis.domain.Clazz">

<id property="id" column="id" />

<result property="code" column= "name" />

<result property="name" column= "name" />

<!--一对多关联映射:collection fetchType="lazy"表示懒加载-->

<collection property="students" javaType="ArrayList"

column="id" ofType="cn.mybatis.domain.Student"

select="cn.mybatis.mapper.StudentMapper.selectStudentByClazzId" fetchType="lazy">

<id property="id" column="id" />

<result property="name" column= "name" />

<reault property="sex" column="sex"/>

<result property="age" column="age"/>

</collection>

</resultMap>

</mapper>

ClazzMapper.xml中定义了一个<select.../>,其根据id查询班级信息。由于Clazz类除了简单的属性id、code、name 之外，还有一个关联对象students，所以返回的是一个名为clazzResultMap的resultMap。由于student是一个List集合，所以clazzResultMap中使用了<collection.../>元素映射一对多的关联关系，select属性表示会使用column属性的id值作为参数执行StudentMapper中定义的selectStudentByClazz查询该班级对应的所有学生数据，查询出的数据将被封装到property表示的students对象当中。

还使用了一个新的属性fetchType，该属性的取值有eager和lazy，eager表示立即加载，即查询Clazz对象的时候，会立即执行关联的selectStudentByClazzId中定义的SQL语句去查询班级的所有学生；lazy表示懒加载，其不会立即发送SQL语句去查询班级的所有学生，而是等到需要使用到班级的students属性时，才会发送SQL语句去查询班级的所有学生。fetch机制更多的是为了性能考虑，如果查询班级时确定会访问班级的所有学生，则该属性应该设置为eager；如果查询班级时只是查询班级信息，有可能不会访问班级的所有学生，则该属性应该设置为lazy。正常情况下，一对多所关联的集合对象，都应该被设置成lazy。

使用懒加载还需要在mybatis-config.xm 中增加如下配置:

<settings>

<!-- 要使延迟加载生效必须配置下面两个属性-->

<setting name="lazyLoadingEnabled" value="true"/>

<setting name="aggressiveLazyLoading" value="false" />

</settings>

lazyLoadingEnabled属性表示延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。默认为false.

agressiveLazyLoading属性启用时，会使带有延迟加载属性的对象立即加载; 反之，每种属性将会按需加载。默认为true,所以这里需要设置成false。

<mapper namespace="cn.mybatis.mapper.studentMapper">

<!-- 根据id查询学生信息，多表连接，返回reaultMap-->

<select id="selectStudentById" parameterType="int" resultMap="studentResuItMap" >

SELECT \* FROM tb\_clazz c,tb\_student s

WHERE c.id = s.clazz\_id

AND s.id = #{id}

</select>

<!-- 根据班级id查询学生信息，返回resultMap-->

<select id="selectstudentByClazzId" parameterType="int" resultMap="studentResultMap">

SELECT \* FROM tb\_student WHERE clazz\_id = #{id}

</select>

<!-- 映射Student对象的resultMap-->

<resultMap id="studentResultMap" type="cn.mybatis.domain.Student" >

<id property = "id" column="id" />

<result property="sex" column="sex" />

<result property="age" column="age" />

<!-- 多对一关联映射:association -->

<association property="clazz" javaType="cn.mybatis.domain.Clazz">

<id property = "id" column="id" />

<result property="code" column="code" />

<result property="name" column="name" />

</association>

</resultMap>

</mapper>

StudentMapper.xm 中定义了一个<select id="selectStudentById".../> 其会根据学生id查询学生信息，由于Student类除了简单的属性id、name、sex和age之外，还有一个关联对象clazz，所以它返回的是一个名为clazzResultMap的resultMap。clazzResutMap中使用了<association.../>元素映射多对一的关联关系，因为<select id="selectStudenById".../>的SQL 语句是一条多表连接，关联tb\_clazz表的同时查询了班级数据，所以<association.../>只是简单地装载数据。

在实际开发中，一对多关系通常映射为集合对象，而由于多方的数据量可能很大，所以通常使用懒加载; 而多对一只是关联到一个对象，所以通常使用多表连接直接把数据提取出来。

StudentMapper.xml 中还定义了一个<select id="selectStudentByClazzId".../>其会根据班级id查询所有学生信息，该查询用于ClazzMapper.xml 中的关联查询。

**多对多关联关系**

在实际项目开发中，多对多关系也是非常常见的关系，比如，一个购物系统中，一个用户可以有多个订单，这是一对多的关系；一个订单中可以购买多种商品，一种商品也可以属于多个不同的订单，订单和商品就是多对多的关系。对于数据库中多对多关系建议使用一个中间表来维护关系，中间表中的订单d作为外键参照订单表的id，商品id作为外键参照商品表的id。

下面我们就用一个简单示例来看看MyBatis怎么处理多对多关系。

商品和订单是多对多的关系，即一种商品可以出现在多个订单中。在Article 类中定义了一个orders 属性，该属性是一个List集合，用来映射多对多的关联关系，表示该商品关联的多个订单。

再接下来是XML映射文件。

<mapper namespace="cn.mybatis.mapper.UserMapper">

<resultMap type="cn.mybatis.domain.User" id="userResultMap">

<id porperty="id" column="id"/ >

<result property="username" column="username"/>

<result property="loginname" column="loginname"/>

<result property="password" column="password"/>

<result property="phone" column="phone"/>

<result property="address" column="address"/>

<!--一对多关联映射：collection-->

<collection property="orders" javaType="ArrayList"

column="id" ofType="cn.mybatis.domain.User"

select="cn.mybatis.mapper.ordeMapper.selectOrderById" fetchType="lazy">

<id porperty="id" column="oid"/ >

<result property="code" column="code"/>

<result property="total" column="total"/>

</collection>

</resultMap>

<select id="selectUserById" parameterType="int" resultMap="userResultMap">

SEIECT 由FROM tb user WHERE id = #{id}

</select>

</mapper>

UserMapper.xml中定义了一一个<select.../>，其根据id查询用户信息。由于User 类除了简单的属性id、usemame、loginame、password和address之外，还有一个关联对象orders，所以返回的是一个名为userResultMap的resultMap。由于orders是一个List集合，因此userResultMap 中使用了<collection.../>元素映射一对多的关联关系，select属性表示会使用columm属性的id值作为参数执行OrderMapper中定义的selectOrderByUserId查询该用户所下的所有订单，查询出的数据将被封装到property表示的orders对象当中。注意，一对多使用的都是lazy(懒加载)。

<mapper namespace="cn.mybatis.mapper.OrderMapper">

<resultMap id="orderResultMapper" type="cn.mybatis.domain.Order">

<id property="id" column="id"/>

<result property="code" column= "code" />

<result property="total" column="total"/>

<!--多对一关联映射:association-->

<association property="user" javaType="cn.mybatis.domain.User"/>

<id property="id" column="id"/>

<result property="username" column="username" />

<result property="loginname" column="loginname"/>

<result property="password" column="password"/>

<result property="phone" column="phone"/>

<result property="address" column="address"/>

</association>

<!--多对多关联映射:collection -->

<collection property="articles" javaType="ArrayList"

column="id" ofType="cn.mybatis.domain.Article"

select="cn.mybatis.mapper.ArticleMapper.selectArticleByOrderId" fetchType="lazy">

<id property="id" column="id" />

<result property="name" column= "name" />

<reault property="price" column="price"/>

<result property="remark" column="remark"/>

</collection>

</resultMap>

<!-- 注意，如果查询出来的列同名，例如tb\_user表的id和tb\_order表的id都是id，同名。则需要使用别名区分-->

< select id="selectOrderById" parameterType="int" resultMap="orderResultMap">

SELECT u.\*,o.id AS oid,code,total,user\_ id

FROM tb\_user u,tb\_ order o

WHERE u.id = o.user\_id

AND o.id = #{id}

</select>

<!-- 根据userid查询订单-->

<select id="selectOrderByUserId parameterType="int" resultType="cn.mybatis.domain.Order"

SELECT \* FROM tb\_order WHERE user\_id = #{id}

</select>

</mapper>

OrderMapper.xml中定义了一个<select id="selectOrderByUserId.../>其根据用户id查询订单信息，返回的是简单的Order对象。还定义了一<select id="selectOrderById.../>,其根据订单id 查询订单信息，由于 Order类和用户是多对一关系，和商品是多对多关系，而多对一通常都是立即加载，因此SQL语句是一条关联了tb\_user和tb\_order的多表查询语句。查询结果返回一个名为orderResultMap的resultMap。orderResultMap中使用了< association>元素映射多对一的关联关系，其将查询到的用户信息装载到Order 对象的user属性当中;orderResutMap中还使用了<collection..>元素映射多对多的关联关系，select属性表示会使用column 属性的oid 值作为参数执行ArticleMapper 中定义的selectArticleByOrderd 查询该订单中的所有商品，查询出的数据将被封装到property表示的articles对象当中。注意，一对多使用的都是lazy(懒加载)。

提示:

因为多表查询返回的结果集中tb user有个id列，tb\_order也有个id列，当列同名时，MyBatis使用的元素中的column属性如果是id，则MyBatis会默认使用查询出的第一个id列。为了区分同名的列，最好的方法是给列取一个别名。SQL 语句中的o.id AS o.oid，resultMap中的column="oid"就是指使用的是tb\_order 表的id:

<mapper namespace="cn.mybatis.mapper.ArticleMapper">

<select id="selectArticleByOrderId" parameterType="int" resultType="cn.mybatis.domain.Article>

SELECT \* FROM tb\_article WHERE id IN (

SELECT article.id FROM tb\_item WHERE order\_id = #(id}

</select>

</mapper>

ArticleMape.xml 中定义了一个<select id="selectArticleByOrderId.../>其根据订单id查询订单关联的所有商品，由于订单和商品是多对多的关系，数据库使用了一一个中间表tb\_item维护多对多的关系，故此处使用了一个子查询，首先根据订单id定位到中间表中查询出所有的商品，之后根据所有商品的id 查询出所有的商品信息，并将这些信息封装到Atrticle对象当中。