MyBatis 3

# Mapper XML文件

MyBatis的真正强大在于它的映射语句，也是它的魔力所在。由于它的异常强大，映射器的XML文件就显得相对简单。如果拿它跟具有相同功能的JDBC代码进行对比，你会发现省掉了将近95%的代码。MyBatis就是针对SQL构建的，并且比普通的方法做的更好。

SQL映射文件有很少的几个顶级元素(按照他们应该被定义的顺序)：

▪ cache – 给定命名空间的缓存配置；

▪ cache-ref – 其它命名空间缓存配置的引用；

▪ resultMap – 是最复杂也是最强大的元素，用来描述如何从数据库结果集中来加载对象；

▪ ~~parameterMap – 已废弃！老式风格的参数映射。内联参数是首选，这个元素可能在将来被移除，这里不会记录。~~

▪ sql – 可被其他语句引用的可重用语句块；

▪ 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。

## select

查询语句是MyBatis中最常见的元素之一，光能把数据存到数据库中的价值并不大，如果还能重新取出来才有用，多数应用也都是查询比修改更频繁。对每个插入、更新、删除操作，通常对应多个查询操作。这是MyBatis的基本原则之一，也是将焦点和努力放到查询和结果映射的原因。简单查询的select元素是非常简单的。比如：

<select id="selectPerson" parameterType="int" resultType="hashmap">

SELECT \* FROM PERSON WHERE ID = #{id}

</select>

这个语句被称为“selectPerson”，接收一个int(或Integer)类型的参数，并返回一个HashMap类型的对象，其中的键是列名，值便是结果行中对应的值。

注意参数符号：

#{id}

意思是告诉MyBatis创建一个预处理语句参数，通过JDBC，这样的一个参数在SQL中会由一个“?”来标识，并被传递到一个新的预处理语句中，就像这样：

// Similar JDBC code, NOT MyBatis…

String selectPerson = "SELECT \* FROM PERSON WHERE ID=?";

PreparedStatement ps = conn.prepareStatement(selectPerson);

ps.setInt(1,id);

当然，这需要很多单独的JDBC的代码啦提取结果并将它们映射到对象实例中，这就是MyBatis节省你时间的地方。我们需要深入了解参数和结果映射，细节部分我们下面来了解。

select元素有很多属性允许你配置，来决定每条语句的作用细节。

<select

id="selectPerson"

parameterType="int"

parameterMap="deprecated"

resultType="hashmap"

resultMap="personResultMap"

flushCache="false"

useCache="true"

timeout="10000"

fetchSize="256"

statementType="PREPARED"

resultSetType="FORWARD\_ONLY">

|  |  |
| --- | --- |
| select属性 | |
| id | 在命名空间中唯一的标识符，可以被用来引用这条语句 |
| parameterType | 将会传入这条语句的参数类的完全限定名或别名。这个属性是可选的，因为MyBatis可以通过TypeHandler推断出具体传入语句的参数，默认值为unset。 |
| ~~parameterMap~~ | ~~这是引用外部parameterMap的已经被废弃的方法。使用内联参数映射和parameterType属性。~~ |
| resultType | 从这条语句中返回的期望类型的类的完全限定名或别名。注意如果是集合情形，那应该是集合可以包含的类型，而不能是集合本身。使用resultType或resultMap，但不能同时使用。 |
| resultMap | 外部resultMap的命名应用。结果集的映射是MyBatis最强大的特性，对其有一个很好的理解的话，许多负载映射的情形都能迎刃而解。使用resultMap或resultType，但不能同时使用。 |
| flushCache | 将其设置为true，任何时候只要语句被调用，都会导致本地缓存和二级缓存都会被清空，默认值：false。 |
| useCache | 将其设置为true，将会导致本条语句的结果被二级缓存，默认值：对select元素为true。 |
| timeout | 这个设置是在抛出异常之前，驱动程序等待数据库返回请求结果的秒数。默认值为unset(依赖驱动) |
| fetchSize | 这是尝试影响驱动程序每次批量返回的结果行数和这个设置值相等。默认值为：unset(依赖驱动) |
| statementType | STATEMENT,PREPARED或CALLABLE的一个。这会让MyBatis分别使用Statement、PreparedStatement或CallableStatement，默认值：PREPARED。 |
| resultSetType | FORWARD\_ONLY,SCROLL\_SENSITIVE或SCROLL\_INSENSITIVE中的一个，默认值为:unset(依赖驱动)。 |
| databaseId | 如果配置了databaseIDProvider，MyBatis会加载所有的不带databaseId或匹配当前databaseId的语句；如果带或者不带的语句都有，则不带的会被忽略。 |
| resultOrdered | 这个设置仅针对嵌套结果select语句适用：如果为true，就是假设包含了嵌套结果集或是分组了，这样的话当返回一个主结果行的时候，就不会发生有对前面结果集的引用的情况。这就使得在获取嵌套的结果集的时候不至于导致内存不够用。默认值：false |
| resultSets | 这个设置仅对多结果集的情况适用，他将列出语句执行后返回的结果集并每个结果集给一个名称，名称是逗号分隔的。 |

## insert、update、delete

数据变更语句insert、update、delete的实现非常接近：

<insert

id="insertAuthor"

parameterType="domain.blog.Author"

flushCache="true"

statementType="PREPARED"

keyProperty=""

keyColumn=""

useGeneratedKeys=""

timeout="20">

<update

id="updateAuthor"

parameterType="domain.blog.Author"

flushCache="true"

statementType="PREPARED"

timeout="20">

<delete

id="deleteAuthor"

parameterType="domain.blog.Author"

flushCache="true"

statementType="PREPARED"

timeout="20">

|  |  |
| --- | --- |
| insert、update、delete属性 | |
| id | 命名空间中的唯一标识，可被用来代表这条语句 |
| parameterType | 将要传入语句的参数的完全限定类名或别名。这个属性是可选的，因为MyBatis可以通过TypeHandler推断出具体传入语句的参数，默认值为：unset |
| ~~parameterMap~~ | ~~这是引用外部parameterMap的已经被废弃的方法。使用内联参数映射和parameterType属性。~~ |
| flushCache | 将其设置为true，任何时候只要语句被调用，都会导致本地缓存和二级缓存被清空，默认值：true(对应插入、更新和删除语句) |
| timeout | 这个设置是在抛出异常之前，驱动程序等待数据库返回请求结果的秒数。默认值为：unset(依赖驱动) |
| statementType | STATEMENT,PREPARED或CALLABLE的一个。这会让MyBatis分别使用Statement、PreparedStatement或CallableStatement，默认值：PREPARED。 |
| useGeneratedKeys | (仅对insert和update有用)这会令MyBatis使用JDBC的getGeneratedKeys方法来取出由数据库内部生成的主键(比如：向MySQL和SQL Server这样的关系数据库管理系统的自动递增字段)，默认值：false |
| keyProperty | (仅对insert和update有用)唯一标记一个属性，MyBatis会通过getGeneratedKeys的返回值或者通过insert语句的selectKey子元素设置它的键值，默认：unset。如果希望得到多个生成的列，也可以是逗号分隔的属性名称列表 |
| keyColumn | (仅对insert和update有用)通过生成的键值设置表中的列名，这个设置仅在某些数据库(像PostgreSQL)是必须的，当主键列不是表中的第一列的时候需要设置。如果希望得到多个生成的列，也可以是逗号分隔的属性名称列表 |
| databaseId | 如果配置了databaseIdProvider，MyBatis会加载所有的不带databaseId或匹配当前databaseId的语句；如果带或者不带的语句都有，则不带的会被忽略。 |

下面是insert、update、delete语句的示例：

<insert id="insertAuthor">

insert into Author (id,username,password,email,bio)

values (#{id},#{username},#{password},#{email},#{bio})

</insert>

<update id="updateAuthor">

update Author set

username = #{username},

password = #{password},

email = #{email},

bio = #{bio}

where id = #{id}

</update>

<delete id="deleteAuthor">

delete from Author where id = #{id}

</delete>

如上所述，插入语句的配置规则更加丰富，在插入语句里面有一些额外的属性和子元素来处理主键的生成，而且有多种生成方式。

首先，如果你的数据库支持自动生成主键的字段(如：MySQL、SQL Server)，那么你可以设置useGeneratedKeys=“true”，然后再把keyProperty设置到目标属性上就OK了。例如，如果上面的Author表已经对id使用了自动生成列类型，那么语句可以修改为：

<insert id="insertAuthor" useGeneratedKeys="true"

keyProperty="id">

insert into Author (username,password,email,bio)

values (#{username},#{password},#{email},#{bio})

</insert>

如果你的数据库还支持多行插入，你可以传入一个Author数组和集合，并返回自动生成的主键：

<insert id="insertAuthor" useGeneratedKeys="true"

keyProperty="id">

insert into Author (username, password, email, bio) values

<foreach item="item" collection="list" separator=",">

(#{item.username}, #{item.password}, #{item.email}, #{item.bio})

</foreach>

</insert>

对于不支持自动生成类型的数据库或可能不支持自动生成主键的JDBC驱动，MyBatis有另外一种方法来生成主键。

这里有一个简单(甚至很傻)的示例，它可以生成一个随机ID(你最好不要这么做，但这里展示了MyBatis处理问题的灵活性及其所关心的广度)：

<insert id="insertAuthor">

<selectKey keyProperty="id" resultType="int" order="BEFORE">

select CAST(RANDOM()\*1000000 as INTEGER) a from SYSIBM.SYSDUMMY1

</selectKey>

insert into Author

(id, username, password, email,bio, favourite\_section)

values

(#{id}, #{username}, #{password}, #{email}, #{bio}, #{favouriteSection,jdbcType=VARCHAR})

</insert>

在上面的示例中，selectKey元素将会首先运行，Author的id会被设置，然后插入语句会被调用。这给你了一个和数据库中来处理自动生成的主键类似的行为，避免了使Java代码变得复杂。

selectKey元素描述如下：

<selectKey

keyProperty="id"

resultType="int"

order="BEFORE"

statementType="PREPARED">

|  |  |
| --- | --- |
| selectKey属性 | |
| keyProperty | selectKey语句结果应该被设置的目标属性。如果希望得到多个生成的列，也可以是逗号分隔的属性名称列表。 |
| keyColumn | 匹配属性的返回结果集中的列名称。如果希望得到多个生成的列，也可以是逗号分隔的属性名称列表 |
| resultType | 结果的类型。MyBatis通常可以推算出来，但是为了更加确定写上也不会有什么问题。MyBatis允许任何简单类型用作主键的类型，包括字符串。如果希望作用域多个生成的列，则可以使用一个包含期望属性的Object或一个Map |
| order | 这可以被设置为BEFORE或AFTER。如果设置为BEFORF，那么它会首先选择主键，设置keyProperty然后执行插入语句。如果设置为AFTER，那么先执行插入语句，然后是selectKey元素 – 这和像Oracle的数据库相似，在插入语句内部可能有嵌套索引调用 |
| statementType | 与前面相同，MyBatis支持STATEMETN,PREPARED和CALLABLE语句的映射类型，分别代表PreparedStatement和CallableStatement类型 |

## sql

这个元素可以被用来定义可重用的SQL代码段，可以包含在其它语句中。它可以被静态地(在加载参数)参数化。不同的属性值通过包含的实例变化。例如：

<sql id="userColumns">${alias}.id,${alias}.username,${alias}.password</sql>

这个SQL片段可以被包含在其它语句中，例如：

<select id="selectUsers" resultType="map">

select

<include refid="userColumns"><property name="alias" value="t1"/></include>,

<include refid="userColumns"><property name="alias" value="t2"/></include>

from some\_table t1

cross join some\_table t2

</select>

属性值也可以被用在include元素的refid属性里(

<include refid="${include\_target}"/>

)或include内部语句中(

${prefix}Table

),例如：

<sql id="sometable">

${prefix}Table

</sql>

<sql id="someinclude">

from

<include refid="${include\_target}"/>

</sql>

<select id="select" resultType="map">

select

field1, field2, field3

<include refid="someinclude">

<property name="prefix" value="Some"/>

<property name="include\_target" value="sometable"/>

</include>

</select>

## 参数(Parameters)

前面的所有语句中你所见到的都是简单参数的例子，实际上参数是MyBatis非常强大的元素，对于简单的做法，大概90%的情况参数都很少见，比如：

<select id="selectUsers" resultType="User">

select id, username, password

from users

where id = #{id}

</select>

上面的这个示例说明了一个非常简单的命令参数映射。参数类型被设置为int，这样这个参数就可以被设置成任何内容。原生的类型或简单数据类型(比如整型和字符串)因为没有相关属性，它会完全用参数值来替代。然后，如果传入一个复杂的对象，行为就会有一点不同了。比如：

<insert id="insertUser" parameterType="User">

insert into users (id, username, password)

values (#{id}, #{username}, #{password})

</insert>

如果User类型的参数对象传递到了语句中，id、username和password属性将会被查找，然后将它们的值传入预处理语句的参数中。

这点相对于向语句中，传参是比较好的，而且又简单，不过参数映射的功能远不止于此。

首先，像MyBatis的其它部分一样，参数也可以指定一个特殊的数据类型。

#{property,javaType=int,jdbcType=NUMERIC}

像MyBatis的剩余部分一样，JavaType通常可以有参数对象确定，除非该对象是一个HashMap。这时所使用的TypeHandler应该明确指明JavaType。

注意：如果一个列允许null值，并且会传递值null的参数，就必须要指定JDBC Type。阅读PreparedStatement.setNull()的JavaDocs文档来获取更多信息。

为了以后定制类型处理方式，你也可以指定一个特殊的类型处理器类(或别名)，比如：

#{age,javaType=int,jdbcType=NUMERIC,typeHandler=MyTypeHandler}

尽管看起来配置变得越来越繁琐，但实际上，很少需要去设置他们。

对于数值类型，还有一个小数保留位的设置，来确定小数点后保留的位数。

#{height,javaType=double,jdbcType=NUMERIC,numericScale=2}

最后，mode属性允许你指定IN、OUT或INOUT参数。如果参数为OUT或INOUT，参数对象属性的真实值将会被改变，就像你在获取输出参数时所期望的那样。如果mode为OUT(或INOUT)，而且jdbcType为CURSOR(也就是Oracle的REFCURSOR)，你必须制定一个resultMap来映射结果集ResultMap到参数类型。要注意这里的JavaType属性是可选的，如果留空并且jdbcType是CURSOR，它会被自动被设为ResultMap。

#{department, mode=OUT, jdbcType=CURSOR, javaType=ResultSet, resultMap=departmentResultMap}

MyBatis也支持很多高级的数据类型，比如结构体，但是当注册out参数时你必须告诉它语句类型名称。比如(再次提示，在实际中要像这样不能换行)：

#{middleInitial, mode=OUT, jdbcType=STRUCT, jdbcTypeName=MY\_TYPE, resultMap=departmentResultMap}

尽管所有这些选项很强大，但大多时候你只需简单地指定属性名，其它的事情MyBatis会自己去推断，顶多要为可能为空的列指定jdbcType。

#{firstName}

#{middleInitial,jdbcType=VARCHAR}

#{lastName}

## 字符串替换

默认情况下，使用#{}格式的语法会导致MyBatis创建PrepareStatement参数并安全地设置参数(就像使用 ？一样)。这样做更安全，更迅速，通常也是首选做法，不过有时你就是想直接在SQL语句中插入一个不转义的字符串。日入，像ORDER BY，你可以这样来使用：

ORDER BY ${columnName}

这里MyBatis不会修改或转义字符串。

注意：用这种方式接受用户的输入，并将其用于语句中的参数是不安全的，会导致潜在的SQL注入攻击，因此要么不允许用户输入这些字段，要么自行转义并检验。

## Result Maps

resultMap 元素是MyBatis中最重要最强大的元素。它可以让你从90%的JDBCResultSets数据提取代码中解放出来，并在一些情形下允许你做一些JDBC不支持的事情。实际上，在对复杂语句进行联合映射的时候，他可能可以代替数千行的同等功能的代码。ResultMap的设计思想是，简单的语句不需要明确的结果映射，而复杂一点的语句只需要描述他们的关系就行了。

你已经见过简单映射语句的示例了，但没有明确的resultMap。比如：

<select id="selectUsers" resultType="map">

select id, username, hashedPassword

from some\_table

where id = #{id}

</select>

上述语句只是简单地将所有的列映射到HashMap的键上，这由resultType属性指定。虽然在大部分情况下都够用，但是HashMap不使一个很好的领域模型。你的程序更可能会使用JavaBean或POJO(Plain Old Java Objects,普通Java对象)作为领域模型。MyBatis对两者都支持。看看下面这个JavaBean：

package com.someapp.model;

public class User {

private int id;

private String username;

private String hashedPassword;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public String getHashedPassword() {

return hashedPassword;

}

public void setHashedPassword(String hashedPassword) {

this.hashedPassword = hashedPassword;

}

}

基于JavaBean的规范，上面这个类有3个属性：id,username,hashedPassword。这些属性会对应到select语句中的列名。

这样的一个JavaBean可以被映射到ResultSet，就像映射到HashMap一样简单。

<select id="selectUsers" resultType="com.someapp.model.User">

select id, username, hashedPassword

from some\_table

where id = #{id}

</select>

类型别名是你的好帮手。使用它们，你就可以不用输入类的完全限定名称了。比如：

<!-- In mybatis-config.xml file -->

<typeAlias type="com.someapp.model.User" alias="User"/>

<!-- In SQL Mapping XML file -->

<select id="selectUsers" resultType="User">

select id, username, hashedPassword

from some\_table

where id = #{id}

</select>

这些情况下，MyBatis会在幕后自动创建一个ResultMap，再基于属性名来映射列到JavaBean的属性上。如果列名和属性名没有精确匹配，可以在select语句中对列使用别名(这是一个基本的SQL 特性)来匹配标签。比如：

<select id="selectUsers" resultType="User">

select

user\_id as "id",

user\_name as "userName",

hashed\_password as "hashedPassword"

from some\_table

where id = #{id}

</select>

ResultMap最优秀的地方在于，虽然你已经对它相当了解，但是根本就不需要显示地用到他们。上面这些简单的示例根本不需要下面这些繁琐的配置。处于示范的原因，让我们来看看最后一个示例中，如果适用外部的resultMap会怎样，这也是解决列名不匹配的另一种方式。

<resultMap id="userResultMap" type="User">

<id property="id" column="user\_id" />

<result property="username" column="user\_name"/>

<result property="password" column="hashed\_password"/>

</resultMap>

引用它的语句使用resultMap属性就行了(注意我们去掉了resultType属性)。比如：

<select id="selectUsers" resultMap="userResultMap">

select user\_id, user\_name, hashed\_password

from some\_table

where id = #{id}

</select>

如果世界总是这么简单就好了。

## 高级结果映射

MyBatis创建的一个想法是：数据库不可能永远是你所想或所需的那个样子。我们希望每个数据库都具备良好的第三范式或BCNF范式，可惜他们不总都是这样。如果有一个独立且完美的数据库映射模式，所有应用程序都可以使用它，那就太好了，但可惜也没有。ResultMap就是MyBatis对这个问题的答案。

比如，我们如何映射下面这个语句？

<!-- Very Complex Statement -->

<select id="selectBlogDetails" resultMap="detailedBlogResultMap">

select

B.id as blog\_id,

B.title as blog\_title,

B.author\_id as blog\_author\_id,

A.id as author\_id,

A.username as author\_username,

A.password as author\_password,

A.email as author\_email,

A.bio as author\_bio,

A.favourite\_section as author\_favourite\_section,

P.id as post\_id,

P.blog\_id as post\_blog\_id,

P.author\_id as post\_author\_id,

P.created\_on as post\_created\_on,

P.section as post\_section,

P.subject as post\_subject,

P.draft as draft,

P.body as post\_body,

C.id as comment\_id,

C.post\_id as comment\_post\_id,

C.name as comment\_name,

C.comment as comment\_text,

T.id as tag\_id,

T.name as tag\_name

from Blog B

left outer join Author A on B.author\_id = A.id

left outer join Post P on B.id = P.blog\_id

left outer join Comment C on P.id = C.post\_id

left outer join Post\_Tag PT on PT.post\_id = P.id

left outer join Tag T on PT.tag\_id = T.id

where B.id = #{id}

</select>

你可能想把它映射到一个智能的对象模型，这个对象表示了一篇博客，它由某位作者所写，有很多的博文，每篇博文有零活多条的评论和标签。我们来看看下面这个完整的例子，它是一个非常复杂的ResultMap(假设作者，博客，博文，评论和标签都是类型的别名)。不用紧张，我们会一步一步来说明。虽然它看起来令人望而生畏，但其实非常简单。

<!-- 超复杂的 Result Map -->

<resultMap id="detailedBlogResultMap" type="Blog">

<constructor>

<idArg column="blog\_id" javaType="int"/>

</constructor>

<result property="title" column="blog\_title"/>

<association property="author" javaType="Author">

<id property="id" column="author\_id"/>

<result property="username" column="author\_username"/>

<result property="password" column="author\_password"/>

<result property="email" column="author\_email"/>

<result property="bio" column="author\_bio"/>

<result property="favouriteSection" column="author\_favourite\_section"/>

</association>

<collection property="posts" ofType="Post">

<id property="id" column="post\_id"/>

<result property="subject" column="post\_subject"/>

<association property="author" javaType="Author"/>

<collection property="comments" ofType="Comment">

<id property="id" column="comment\_id"/>

</collection>

<collection property="tags" ofType="Tag" >

<id property="id" column="tag\_id"/>

</collection>

<discriminator javaType="int" column="draft">

<case value="1" resultType="DraftPost"/>

</discriminator>

</collection>

</resultMap>

resultMap元素有很多子元素和一个值得讨论的结构，下面是resultMap元素的概念视图。

**resultMap**

▪ constructor – 用于在实例化类时，注入结果到构造方法中

♢ idArg – ID 参数；标记出作为ID的结果可以帮助提高整体性能

♢ arg – 将被注入到构造方法的一个普通结果

▪ id – 一个ID结果；标记出作为ID的结果可以帮助提供整体性能

▪ result – 注入到字段或JavaBean属性的普通结果

▪ association – 一个复杂类型的关联；许多结果将包装成这种类型

♢ 嵌套结果映射 – 关联可以指定为一个resultMap元素，或者引用一个

▪ collection – 一个复杂类型的集合

♢ 嵌套结果映射 – 集合可以指定为一个resultMap元素，或者引用一个

▪ discriminatory – 使用结果值来决定使用哪个resultMap

♢ case – 基于某些值的结果映射

▪ 嵌套结果映射 – 一个case也是一个映射它本身的结果，因此可以包含很多相同的元素，或者它可以参照一个外部的resultMap

|  |  |
| --- | --- |
| ResultMap | |
| 属性 | 描述 |
| id | 当前命名空间中的唯一标识，用于标识一个result map |
| type | 类的完全限定名，或者一个类型别名(内置的别名可以参考上面的表格) |
| autoMaping | 如果设置这个属性，MyBatis将会为这个ResultMap开启或者关闭自动映射。这个属性会覆盖全局的属性autoMappingBehavior。默认值为：unset |

**最佳实践** 最好一步步地建立结果映射。单元测试可以在这个过程中起到很大帮助。如果你尝试一次创建一个像上面示例那样的巨大的结果映射，那么很可能会出现错误而且很难去使用它来完成工作。从最简单的形态开始，逐步进化。而且别忘了单元测试！使用框架的缺点是有时候他们看上去像黑盒子(无论代码是否可见)。为了确保你实现的行为和想要的一致，最好的选择时编写单元测试。提交bug的时候它也能起到很大的作用。

下面将详细说明每个元素：

### id & result

<id property="id" column="post\_id"/>

<result property="subject" column="post\_subject"/>

这些是结果映射最基本的内容。id和result都将一个列的值映射到一个简单数据类型(字符串，整型，双精度浮点数，日期等)的属性或字段。

这两者之间的唯一不同是，id表示的结果将是对象的标识属性，这会在比较对象实例时用到。这样可以提高整体的性能，尤其是缓存和嵌套结果映射(也就是联合映射)的时候。

两个元素都有一些属性：

|  |  |
| --- | --- |
| id属性 | |
| 属性 | 描述 |
| property | 映射到列结果的字段或属性。如果用来匹配的JavaBeans存在给定名字的属性，那么它将会被使用。否则MyBatis将会寻找给定名称property的字段。无论是哪一种情形，你都可以使用通常的点分隔形式进行复杂属性导航。比如，你可以这样映射一些简单的东西：“username”，或者映射到一些复杂的东西：“address.street.number” |
| column | 数据库中的列名，或者是列的别名。一般情况下，这和传递给resultSet.getString(columnName)方法的参数一样。 |
| javaType | 一个Java类的完全限定名，或一个类型别名(参考上面内建类型别名的列表)。如果你映射到一个JavaBean，MyBatis通常可以断定类型。然后，如果映射到的是HashMap，那么你应该明确地指定JavaType来保证期望的行为。 |
| jdbcType | JDBC类型，所支持的JDBC类型参见这个表格之后的“支持JDBC类型”。只需要在可能执行插入、更新和删除的允许空值的列上指定JDBC类型。这是JDBC的要求而非MyBatis的要求。如果你直接面向JDBC编程，你需要对可能为null的值指定这个类型。 |
| TypeHandler | 我们在前面讨论过的默认类型处理器。使用这个属性，你可以覆盖默认的类型处理器。这个属性值是一个类型处理器实现类的玩全限定名，或者是类型别名 |

### 支持JDBC类型

为了未来的参考，MyBatis通过包含的jdbcType枚举型，支持下面的JDBC类型。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| BIT | FLOAT | CHAR | TIMESTAMP | OTHER | UNDEFINED |
| TINYINT | REAL | VARCHAR | BINARY | BLOB | NVARCHAR |
| SMALLINT | DOUBLE | LONGVARCHAR | VARBINARY | CLOB | NCHAR |
| INTEGER | NUMERIC | DATE | LONGVARBINARY | BOOLEAN | NCLOB |
| BIGINT | DECIMAL | TIME | NULL | CURSOR | ARRAY |

### 构造方法

通过修改对象属性的方式，可以满足大多数的数据传输对象(DataTransfer Object, DTO)以及绝大部分领域模型的要求。但有些情况下你想使用不可变类。通常来说，很少或者基本上不变的、包含引用或查询数据的表，很适合使用不可变类。构造方法注入允许你在初始化时为类设置属性的值，而不用暴露出共有方法。MyBatis也支持私有属性和私有JavaBeans属性来达到这个目的，但有一些人更青睐有构造方法注入。constructor元素就是为此而生的。

看看下面这个构造方法：

public class User {

//...

public User(Integer id, String username, int age) {

//...

}

//...

}

为了将结果注入构造方法，MyBatis需要通过某种方式定位响应的构造方法。在下面的例子中，MyBatis搜索一个生命了三个形参的构造方法，以java.lang.Integer，java.lang.String and int的顺序排列。

<constructor>

<idArg column="id" javaType="int"/>

<arg column="username" javaType="String"/>

<arg column="age" javaType="\_int"/>

</constructor>

当你处理一个带有多个形参的构造方法时，很容易在保证arg元素的正确顺序上出错。从版本3.4.3开始，可以在指定参数名称的前提下，以任意顺序编写arg元素。为了通过名称来引用构造方法参数，你可以添加@Param注解，或者使用’-parameters’编译选项并启用useActualParamName选项(默认开启)来编译项目。下面的例子对于同一个构造方法依然是有效的，尽管第二和第三形参顺序与构造方法中声明的顺序不匹配。

<constructor>

<idArg column="id" javaType="int" name="id" />

<arg column="age" javaType="\_int" name="age" />

<arg column="username" javaType="String" name="username" />

</constructor>

如果类中存在名称和类型相同的属性，那么可以省略JavaType。

剩余的属性和规则和普通的id和result元素是一样的。

|  |  |
| --- | --- |
| 属性 | |
| 属性 | 描述 |
| column | 数据库中的列名，或者是列的别名。一般情况下，这和传递给resultSet.getString(columnName)方法的参数一样。 |
| JavaType | 一个 Java 类的完全限定名,或一个类型别名(参考上面内建类型别名的列表)。 如果你映射到一个 JavaBean,MyBatis 通常可以断定类型。然而,如 果你映射到的是 HashMap,那么你应该明确地指定 javaType 来保证期望的 行为。 |
| jdbcType | JDBC 类型，所支持的 JDBC 类型参见这个表格之前的“支持的 JDBC 类型”。 只需要在可能执行插入、更新和删除的允许空值的列上指定 JDBC 类型。这是 JDBC 的要求而非 MyBatis 的要求。如果你直接面向 JDBC 编程,你需要对可能为 null 的值指定这个类型。 |
| typeHandler | 我们在前面讨论过的默认类型处理器。使用这个属性,你可以覆盖默 认的类型处理器。这个属性值是一个类型处理 器实现类的完全限定名，或者是类型别名。 |
| select | 用于加载复杂类型属性的映射语句的 ID,它会从 column 属性中指定的列检索数据，作为参数传递给此 select 语句。具体请参考 Association 标签。 |
| resultMap | ResultMap 的 ID，可以将嵌套的结果集映射到一个合适的对象树中，功能和 select 属性相似，它可以实现将多表连接操作的结果映射成一个单一的ResultSet。这样的ResultSet将会将包含重复或部分数据重复的结果集正确的映射到嵌套的对象树中。为了实现它, MyBatis允许你 “串联” ResultMap,以便解决嵌套结果集的问题。想了解更多内容，请参考下面的Association元素。 |
| name | 构造方法形参的名字。从3.4.3版本开始，通过指定具体的名字，你可以以任意顺序写入arg元素。参看上面的解释。 |

### 关联

<association property="author" column="blog\_author\_id" javaType="Author">

<id property="id" column="author\_id"/>

<result property="username" column="author\_username"/>

</association>

关联元素处理“有一个”类型的关系。比如，在我们的示例中，一个博客有一个用户。关联映射就工作于这种结果之上。你指定了目标属性，来获取值的列，属性的java类型(很多情况下MyBatis可以自己算出来)，如果需要的话还有jdbc类型，如果你想覆盖或获取的结果值还需要类型控制器。

关联中不同的是你需要告诉MyBatis如何加载关联。MyBatis在这方面会有两种不同的方式：

▪ 嵌套查询：通过执行另一个SQL映射语句来返回预期的复杂类型；

▪ 嵌套结果：使用嵌套结果映射来处理重复的关联结果的子集。首先，让我们来查看这个元素的属性。所有的你都会看到，它和普通的只由select和resultMap属性的结果映射不同。

|  |  |
| --- | --- |
| 属性 | |
| 属性 | 描述 |
| property | 映射到列结果的字段或属性。如果用来匹配的 JavaBeans 存在给定名字的属性，那么它将会被使用。 否则 MyBatis 将会寻找与给定名称相同的字段。 这两种情形你可以使用通常点式的复杂属性导航。比如,你可以这样映射 一 些 东 西 :“ username ”, 或 者 映 射 到 一 些 复 杂 的 东 西 : “address.street.number” 。 |
| javaType | 一个 Java 类的完全限定名,或一个类型别名(参考上面内建类型别名的列 表) 。如果你映射到一个 JavaBean,MyBatis 通常可以断定类型。然而,如 javaType 果你映射到的是 HashMap,那么你应该明确地指定 javaType 来保证所需的 行为。 |
| jdbcType | 在这个表格之前的所支持的 JDBC 类型列表中的类型。JDBC 类型是仅仅 需要对插入, 更新和删除操作可能为空的列进行处理。这是 JDBC 的需要, jdbcType 而不是 MyBatis 的。如果你直接使用 JDBC 编程,你需要指定这个类型-但 仅仅对可能为空的值。 |
| typeHandler | 我们在前面讨论过默认的类型处理器。使用这个属性,你可以覆盖默认的 typeHandler 类型处理器。 这个属性值是类的完全限定名或者是一个类型处理器的实现, 或者是类型别名。 |

### 关联的嵌套查询

|  |  |
| --- | --- |
| 属性 | 描述 |
| column | 来自数据库的列名,或重命名的列标签。这和通常传递给 resultSet.getString(columnName)方法的字符串是相同的。 column 注 意 : 要 处 理 复 合 主 键 , 你 可 以 指 定 多 个 列 名 通 过 column= ” {prop1=col1,prop2=col2} ” 这种语法来传递给嵌套查询语 句。这会引起 prop1 和 prop2 以参数对象形式来设置给目标嵌套查询语句。 |
| select | 另外一个映射语句的 ID,可以加载这个属性映射需要的复杂类型。获取的 在列属性中指定的列的值将被传递给目标 select 语句作为参数。表格后面 有一个详细的示例。 select 注 意 : 要 处 理 复 合 主 键 , 你 可 以 指 定 多 个 列 名 通 过 column= ” {prop1=col1,prop2=col2} ” 这种语法来传递给嵌套查询语 句。这会引起 prop1 和 prop2 以参数对象形式来设置给目标嵌套查询语句。 |
| fetchType | 可选的。有效值为 lazy和eager。 如果使用了，它将取代全局配置参数lazyLoadingEnabled。 |

示例：

<resultMap id="blogResult" type="Blog">

<association property="author" column="author\_id" javaType="Author" select="selectAuthor"/>

</resultMap>

<select id="selectBlog" resultMap="blogResult">

SELECT \* FROM BLOG WHERE ID = #{id}

</select>

<select id="selectAuthor" resultType="Author">

SELECT \* FROM AUTHOR WHERE ID = #{id}

</select>

我们有两个查询语句：一个来加载博客，另一个来加载作者，而且博客的结果映射描述了“selectAuthor”语句应该被用来加载它的author属性。

其他所有的属性将会被自动加载，假设它们的列和属性名相匹配。

这种方式很简单，但是对于大型数据集合和列表将不会表现很好。问题就是我们熟知的“N+1查询问题”。概括地讲，N+1查询问题可以这样引用的：

▪ 你执行了一个单独的SQL语句来获取结果列表(就是“+1”)；

▪ 对返回的每条记录，你执行了一个查询语句来为每个加载细节(就是“N”)。

这个问题会导致成百上千的SQL语句被执行。这通常不是期望的。

MyBatis能延迟加载这样的查询就是一个好处，因此你可以分散这些语句同时运行的消耗。然而，如果你加载一个列表，之后迅速迭代来访问嵌套的数据，你会调用所有的延迟加载，这样的行为可能是很糟糕的。

所以还有另外一种方法。

### 关联的嵌套结果

|  |  |
| --- | --- |
| 属性 | 描述 |
| resultMap | 这是结果映射的 ID,可以映射关联的嵌套结果到一个合适的对象图中。这 是一种替代方法来调用另外一个查询语句。这允许你联合多个表来合成到 resultMap 一个单独的结果集。这样的结果集可能包含重复,数据的重复组需要被分 解,合理映射到一个嵌套的对象图。为了使它变得容易,MyBatis 让你“链 接”结果映射,来处理嵌套结果。一个例子会很容易来仿照,这个表格后 面也有一个示例。 |
| columnPrefix | 当连接多表时，你将不得不使用列别名来避免ResultSet中的重复列名。指定columnPrefix允许你映射列名到一个外部的结果集中。 请看后面的例子。 |
| notNullColumn | 默认情况下，子对象仅在至少一个列映射到其属性非空时才创建。 通过对这个属性指定非空的列将改变默认行为，这样做之后Mybatis将仅在这些列非空时才创建一个子对象。 可以指定多个列名，使用逗号分隔。默认值：未设置(unset)。 |
| autoMapping | 如果使用了，当映射结果到当前属性时，Mybatis将启用或者禁用自动映射。 该属性覆盖全局的自动映射行为。 注意它对外部结果集无影响，所以在selector resultMap属性中这个是毫无意义的。 默认值：未设置(unset)。 |