**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，你会立即发现省掉了将近 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（按照它们应该被定义的顺序）：

* cache – 给定命名空间的缓存配置。
* cache-ref – 其他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。
* 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

下一部分将从语句本身开始来描述每个元素的细节。

**select**

查询语句是 MyBatis 中最常用的元素之一，光能把数据存到数据库中价值并不大，如果还能重新取出来才有用，多数应用也都是查询比修改要频繁。对每个插入、更新或删除操作，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**>

这个语句被称作 selectPerson，接受一个 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，值便是结果行中的对应值。

注意参数符号：

**#{id}**

这就告诉 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 的代码来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 Attributes

| **属性** | **描述** |
| --- | --- |
| 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。 |
| 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 的命名引用。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，许多复杂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，它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 和 delete**

数据变更语句 insert，update 和 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 和 Delete 的属性

| **属性** | **描述** |
| --- | --- |
| id | 命名空间中的唯一标识符，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。 |
| 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：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：像 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。 |

下面就是 insert，update 和 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>

如前所述，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用来处理主键的生成，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（比如 MySQL 和 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 使用了自动生成的列类型，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**>

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 JDBC 驱动来说，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 的属性

| **属性** | **描述** |
| --- | --- |
| 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。如果希望作用于多个生成的列，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 BEFORE，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入索引调用。 |
| statementType | 与前面相同，MyBatis 支持 STATEMENT，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，可以包含在其他语句中。It can be statically (during load phase) parametrized. Different property values can vary in include instances. 比如：

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

这个 SQL 片段可以被包含在其他语句中，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**>

Property value can be also used in include refid attribute or property values inside include clause, for example:

<**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，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名参数映射。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。然而，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 的其他部分一样，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 通常可以从参数对象中来去确定，前提是只要对象不是一个 HashMap。那么 javaType 应该被确定来保证使用正确类型处理器。

NOTE 如果 null 被当作值来传递，对于所有可能为空的列，JDBC Type 是需要的。你可以自己通过阅读预处理语句的 setNull() 方法的 JavaDocs 文档来研究这种情况。

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，但实际上是很少去设置它们。

对于数值类型，还有一个小数保留位数的设置，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)，你必须指定一个 resultMap 来映射结果集到参数类型。要注意这里的 javaType 属性是可选的，如果左边的空白是 jdbcType 的 CURSOR 类型，它会自动地被设置为结果集。

**#{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}**

尽管所有这些强大的选项很多时候你只简单指定属性名，其他的事情 MyBatis 会自己去推断，最多你需要为可能为空的列名指定 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 创建预处理语句属性并安全地设置值（比如?）。这样做更安全，更迅速，通常也是首选做法，不过有时你只是想直接在 SQL 语句中插入一个不改变的字符串。比如，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。

NOTE 以这种方式接受从用户输出的内容并提供给语句中不变的字符串是不安全的，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。它就是让你远离 90%的需要从结果 集中取出数据的 JDBC 代码的那个东西, 而且在一些情形下允许你做一些 JDBC 不支持的事 情。 事实上, 编写相似于对复杂语句联合映射这些等同的代码, 也许可以跨过上千行的代码。 ResultMap 的设计就是简单语句不需要明确的结果映射,而很多复杂语句确实需要描述它们 的关系。

你已经看到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**>

这样一个语句简单作用于所有列被自动映射到 HashMap 的键上,这由 resultType 属性 指定。这在很多情况下是有用的,但是 HashMap 不能很好描述一个领域模型。那样你的应 用程序将会使用 JavaBeans 或 POJOs(Plain Old Java Objects,普通 Java 对象)来作为领域 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 和 hashedPassword。这些 在 select 语句中会精确匹配到列名。

这样的一个 JavaBean 可以被映射到结果集,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**>

要记住类型别名是你的伙伴。使用它们你可以不用输入类的全路径。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,基于属性名来映射列到 JavaBean 的属性上。如果列名没有精确匹配,你可以在列名上使用 select 字句的别名(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 最优秀的地方你已经了解了很多了,但是你还没有真正的看到一个。这些简 单的示例不需要比你看到的更多东西。 只是出于示例的原因, 让我们来看看最后一个示例中 外部的 resultMap 是什么样子的,这也是解决列名不匹配的另外一种方式。

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

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

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

<**result** property="password" column="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:数据库不用永远是你想要的或需要它们是什么样的。而我们 最喜欢的数据库最好是第三范式或 BCNF 模式,但它们有时不是。如果可能有一个单独的 数据库映射,所有应用程序都可以使用它,这是非常好的,但有时也不是。结果映射就是 MyBatis 提供处理这个问题的答案。

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,包含一个作者写的博客,有很多的博文,每 篇博文有零条或多条的评论和标签。 下面是一个完整的复杂结果映射例子 (假设作者, 博客, 博文, 评论和标签都是类型的别名) 我们来看看, 。 但是不用紧张, 我们会一步一步来说明。 当天最初它看起来令人生畏,但实际上非常简单。

<!-- Very Complex 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 - 类在实例化时,用来注入结果到构造方法中
  + idArg - ID 参数;标记结果作为 ID 可以帮助提高整体效能
  + arg - 注入到构造方法的一个普通结果
* id – 一个 ID 结果;标记结果作为 ID 可以帮助提高整体效能
* result – 注入到字段或 JavaBean 属性的普通结果
* association – 一个复杂的类型关联;许多结果将包成这种类型
  + 嵌入结果映射 – 结果映射自身的关联,或者参考一个
* collection – 复杂类型的集
  + 嵌入结果映射 – 结果映射自身的集,或者参考一个
* discriminator – 使用结果值来决定使用哪个结果映射
  + case – 基于某些值的结果映射
    - 嵌入结果映射 – 这种情形结果也映射它本身,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结果映射。

ResultMap Attributes

| **Attribute** | **Description** |
| --- | --- |
| id | A unique identifier in this namespace that can be used to reference this result map. |
| type | A fully qualified Java class name, or a type alias (see the table above for the list of built-in type aliases). |
| autoMapping | If present, MyBatis will enable or disable the automapping for this ResultMap. This attribute overrides the global autoMappingBehavior. Default: unset. |

最佳实践 通常逐步建立结果映射。单元测试的真正帮助在这里。如果你尝试创建 一次创建一个向上面示例那样的巨大的结果映射, 那么可能会有错误而且很难去控制它 来工作。开始简单一些,一步一步的发展。而且要进行单元测试!使用该框架的缺点是 它们有时是黑盒(是否可见源代码) 。你确定你实现想要的行为的最好选择是编写单元 测试。它也可以你帮助得到提交时的错误。

下面一部分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内容。id 和 result 都映射一个单独列的值到简单数据类型(字符 串,整型,双精度浮点数,日期等)的单独属性或字段。

这两者之间的唯一不同是 id 表示的结果将是当比较对象实例时用到的标识属性。这帮 助来改进整体表现,特别是缓存和嵌入结果映射(也就是联合映射) 。

每个都有一些属性:

Id and Result Attributes

| **属性** | **描述** |
| --- | --- |
| property | 映射到列结果的字段或属性。如果匹配的是存在的,和给定名称相同 的 JavaBeans 的属性,那么就会使用。否则 MyBatis 将会寻找给定名称 property 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" 。 |
| column | 从数据库中得到的列名,或者是列名的重命名标签。这也是通常和会 传递给 resultSet.getString(columnName)方法参数中相同的字符串。 |
| 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 来保证所需的行为。 |
| jdbcType | 在这个表格之后的所支持的 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 jdbcType 的需要,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。 |

**支持的 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 | BLOG | NVARCHAR |
| SMALLINT | DOUBLE | LONGVARCHAR | VARBINARY | CLOB | NCHAR |
| INTEGER | NUMERIC | DATE | LONGVARBINARY | BOOLEAN | NCLOB |
| BIGINT | DECIMAL | TIME | NULL | CURSOR | ARRAY |

**构造方法**

<**constructor**>

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

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

</**constructor**>

对于大多数数据传输对象(Data Transfer Object,DTO)类型,属性可以起作用,而且像 你绝大多数的领域模型, 指令也许是你想使用一成不变的类的地方。 通常包含引用或查询数 据的表很少或基本不变的话对一成不变的类来说是合适的。 构造方法注入允许你在初始化时 为类设置属性的值,而不用暴露出公有方法。MyBatis 也支持私有属性和私有 JavaBeans 属 性来达到这个目的,但是一些人更青睐构造方法注入。构造方法元素支持这个。

看看下面这个构造方法:

**public** **class** **User** {

//...

**public** **User**(**int** id, String username) {

//...

}

//...

}

为了向这个构造方法中注入结果,MyBatis 需要通过它的参数的类型来标识构造方法。 Java 没有自查(反射)参数名的方法。所以当创建一个构造方法元素时,保证参数是按顺序 排列的,而且数据类型也是确定的。

<**constructor**>

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

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

</**constructor**>

剩余的属性和规则和固定的 id 和 result 元素是相同的。

| **属性** | **描述** |
| --- | --- |
| column | 来自数据库的类名,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。 |
| 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 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。 |
| select | The ID of another mapped statement that will load the complex type required by this property mapping. The values retrieved from columns specified in the column attribute will be passed to the target select statement as parameters. See the Association element for more. |
| resultMap | This is the ID of a ResultMap that can map the nested results of this argument into an appropriate object graph. This is an alternative to using a call to another select statement. It allows you to join multiple tables together into a single ResultSet. Such a ResultSet will contain duplicated, repeating groups of data that needs to be decomposed and mapped properly to a nested object graph. To facilitate this, MyBatis lets you "chain" result maps together, to deal with the nested results. See the Association element below for more. |

**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:通过执行另外一个 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联合结果的子集。首先,然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** | **描述** |
| --- | --- |
| property | 映射到列结果的字段或属性。如果匹配的是存在的,和给定名称相同的 property JavaBeans 的属性, 那么就会使用。 否则 MyBatis 将会寻找给定名称的字段。 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 | 来自数据库的类名,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 | Optional. Valid values are lazy and eager. If present, it supersedes the global configuration parameter lazyLoadingEnabled for this mapping. |

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,另外一个来加载作者,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 查询问题可以是这样引起的:

* 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 | When joining multiple tables, you would have to use column alias to avoid duplicated column names in the ResultSet. Specifying columnPrefix allows you to map such columns to an external resultMap. Please see the example explained later in this section. |
| notNullColumn | By default a child object is created only if at least one of the columns mapped to the child's properties is non null. With this attribute you can change this behaviour by specifiying which columns must have a value so MyBatis will create a child object only if any of those columns is not null. Multiple column names can be specified using a comma as a separator. Default value: unset. |
| autoMapping | If present, MyBatis will enable or disable auto-mapping when mapping the result to this property. This attribute overrides the global autoMappingBehavior. Note that it has no effect on an external resultMap, so it is pointless to use it with select or resultMap attribute. Default value: unset. |

在上面你已经看到了一个非常复杂的嵌套关联的示例。 下面这个是一个非常简单的示例 来说明它如何工作。代替了执行一个分离的语句,我们联合博客表和作者表在一起,就像:

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

**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

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

**where** B.id = #{id}

</**select**>

注意这个联合查询, 以及采取保护来确保所有结果被唯一而且清晰的名字来重命名。 这使得映射非常简单。现在我们可以映射这个结果:

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

<**id** property="id" column="blog\_id" />

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

<**association** property="author" column="blog\_author\_id" javaType="Author" resultMap="authorResult"/>

</**resultMap**>

<**resultMap** id="authorResult" type="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"/>

</**resultMap**>

在上面的示例中你可以看到博客的作者关联代表着"authorResult"结果映射来加载作 者实例。

非常重要: 在嵌套据诶过映射中 id 元素扮演了非常重要的角色。应应该通常指定一个 或多个属性,它们可以用来唯一标识结果。实际上就是如果你离开她了,但是有一个严重的 性能问题时 MyBatis 仍然可以工作。选择的属性越少越好,它们可以唯一地标识结果。主键 就是一个显而易见的选择(尽管是联合主键)。

现在,上面的示例用了外部的结果映射元素来映射关联。这使得 Author 结果映射可以 重用。然而,如果你不需要重用它的话,或者你仅仅引用你所有的结果映射合到一个单独描 述的结果映射中。你可以嵌套结果映射。这里给出使用这种方式的相同示例:

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

<**id** property="id" column="blog\_id" />

<**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"/>

</**association**>

</**resultMap**>

What if the blog has a co-author? The select statement would look like:

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

**select**

B.id **as** blog\_id,

B.title **as** blog\_title,

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,

CA.id **as** co\_author\_id,

CA.username **as** co\_author\_username,

CA.password **as** co\_author\_password,

CA.email **as** co\_author\_email,

CA.bio **as** co\_author\_bio

**from** Blog B

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

left outer **join** Author CA on B.co\_author\_id = CA.id

**where** B.id = #{id}

</**select**>

Recall that the resultMap for Author is defined as follows.

<**resultMap** id="authorResult" type="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"/>

</**resultMap**>

Because the column names in the results differ from the columns defined in the resultMap, you need to specify columnPrefix to reuse the resultMap for mapping co-author results.

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

<**id** property="id" column="blog\_id" />

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

<**association** property="author"

resultMap="authorResult" />

<**association** property="coAuthor"

resultMap="authorResult"

columnPrefix="co\_" />

</**resultMap**>

上面你已经看到了如何处理"有一个"类型关联。但是"有很多个"是怎样的?下面这 个部分就是来讨论这个主题的。

**集合**

<**collection** property="posts" ofType="domain.blog.Post">

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

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

<**result** property="body" column="post\_body"/>

</**collection**>

集合元素的作用几乎和关联是相同的。实际上,它们也很相似,文档的异同是多余的。 所以我们更多关注于它们的不同。

我们来继续上面的示例,一个博客只有一个作者。但是博客有很多文章。在博客类中, 这可以由下面这样的写法来表示:

**private** **List** posts;

要映射嵌套结果集合到 List 中,我们使用集合元素。就像关联元素一样,我们可以从 连接中使用嵌套查询,或者嵌套结果。

**集合的嵌套查询**

首先,让我们看看使用嵌套查询来为博客加载文章。

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

<**collection** property="posts" javaType="ArrayList" column="id" ofType="Post" select="selectPostsForBlog"/>

</**resultMap**>

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

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

</**select**>

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

SELECT \* FROM POST WHERE BLOG\_ID = #{id}

</**select**>

这里你应该注意很多东西,但大部分代码和上面的关联元素是非常相似的。首先,你应 该注意我们使用的是集合元素。然后要注意那个新的"ofType"属性。这个属性用来区分 JavaBean(或字段)属性类型和集合包含的类型来说是很重要的。所以你可以读出下面这个 映射:

<**collection** property="posts" javaType="ArrayList" column="id" ofType="Post" select="selectPostsForBlog"/>

读作: "在 Post 类型的 ArrayList 中的 posts 的集合。"

javaType 属性是不需要的,因为 MyBatis 在很多情况下会为你算出来。所以你可以缩短 写法:

<**collection** property="posts" column="id" ofType="Post" select="selectPostsForBlog"/>

**集合的嵌套结果**

至此,你可以猜测集合的嵌套结果是如何来工作的,因为它和关联完全相同,除了它应 用了一个"ofType"属性

First, let's look at the SQL:

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

**select**

B.id **as** blog\_id,

B.title **as** blog\_title,

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

P.id **as** post\_id,

P.subject **as** post\_subject,

P.body **as** post\_body,

**from** Blog B

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

**where** B.id = #{id}

</**select**>

我们又一次联合了博客表和文章表,而且关注于保证特性,结果列标签的简单映射。现 在用文章映射集合映射博客,可以简单写为:

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

<**id** property="id" column="blog\_id" />

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

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

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

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

<**result** property="body" column="post\_body"/>

</**collection**>

</**resultMap**>

同样,要记得 id 元素的重要性,如果你不记得了,请阅读上面的关联部分。

同样, 如果你引用更长的形式允许你的结果映射的更多重用, 你可以使用下面这个替代 的映射:

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

<**id** property="id" column="blog\_id" />

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

<**collection** property="posts" ofType="Post" resultMap="blogPostResult" columnPrefix="post\_"/>

</**resultMap**>

<**resultMap** id="blogPostResult" type="Post">

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

<**result** property="subject" column="subject"/>

<**result** property="body" column="body"/>

</**resultMap**>

注意 这个对你所映射的内容没有深度,广度或关联和集合相联合的限制。当映射它们 时你应该在大脑中保留它们的表现。 你的应用在找到最佳方法前要一直进行的单元测试和性 能测试。好在 myBatis 让你后来可以改变想法,而不对你的代码造成很小(或任何)影响。

高级关联和集合映射是一个深度的主题。文档只能给你介绍到这了。加上一点联系,你 会很快清楚它们的用法。

**鉴别器**

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

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

</**discriminator**>

有时一个单独的数据库查询也许返回很多不同 (但是希望有些关联) 数据类型的结果集。 鉴别器元素就是被设计来处理这个情况的, 还有包括类的继承层次结构。 鉴别器非常容易理 解,因为它的表现很像 Java 语言中的 switch 语句。

定义鉴别器指定了 column 和 javaType 属性。 列是 MyBatis 查找比较值的地方。 JavaType 是需要被用来保证等价测试的合适类型(尽管字符串在很多情形下都会有用)。比如:

<**resultMap** id="vehicleResult" type="Vehicle">

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

<**result** property="vin" column="vin"/>

<**result** property="year" column="year"/>

<**result** property="make" column="make"/>

<**result** property="model" column="model"/>

<**result** property="color" column="color"/>

<**discriminator** javaType="int" column="vehicle\_type">

<**case** value="1" resultMap="carResult"/>

<**case** value="2" resultMap="truckResult"/>

<**case** value="3" resultMap="vanResult"/>

<**case** value="4" resultMap="suvResult"/>

</**discriminator**>

</**resultMap**>

在这个示例中, MyBatis 会从结果集中得到每条记录, 然后比较它的 vehicle 类型的值。 如果它匹配任何一个鉴别器的实例,那么就使用这个实例指定的结果映射。换句话说,这样 做完全是剩余的结果映射被忽略(除非它被扩展,这在第二个示例中讨论) 。如果没有任何 一个实例相匹配,那么 MyBatis 仅仅使用鉴别器块外定义的结果映射。所以,如果 carResult 按如下声明:

<resultMap id="carResult" type="Car">

<**result** property="doorCount" column="door\_count" />

</**resultMap**>

那么只有 doorCount 属性会被加载。这步完成后完整地允许鉴别器实例的独立组,尽管 和父结果映射可能没有什么关系。这种情况下,我们当然知道 cars 和 vehicles 之间有关系, 如 Car 是一个 Vehicle 实例。因此,我们想要剩余的属性也被加载。我们设置的结果映射的 简单改变如下。

<resultMap id="carResult" type="Car" extends="vehicleResult">

<**result** property="doorCount" column="door\_count" />

</**resultMap**>

现在 vehicleResult 和 carResult 的属性都会被加载了。

尽管曾经有些人会发现这个外部映射定义会多少有一些令人厌烦之处。 因此还有另外一 种语法来做简洁的映射风格。比如:

<**resultMap** id="vehicleResult" type="Vehicle">

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

<**result** property="vin" column="vin"/>

<**result** property="year" column="year"/>

<**result** property="make" column="make"/>

<**result** property="model" column="model"/>

<**result** property="color" column="color"/>

<**discriminator** javaType="int" column="vehicle\_type">

<**case** value="1" resultType="carResult">

<**result** property="doorCount" column="door\_count" />

</**case**>

<**case** value="2" resultType="truckResult">

<**result** property="boxSize" column="box\_size" />

<**result** property="extendedCab" column="extended\_cab" />

</**case**>

<**case** value="3" resultType="vanResult">

<**result** property="powerSlidingDoor" column="power\_sliding\_door" />

</**case**>

<**case** value="4" resultType="suvResult">

<**result** property="allWheelDrive" column="all\_wheel\_drive" />

</**case**>

</**discriminator**>

</**resultMap**>

要记得 这些都是结果映射, 如果你不指定任何结果, 那么 MyBatis 将会为你自动匹配列 和属性。所以这些例子中的大部分是很冗长的,而其实是不需要的。也就是说,很多数据库 是很复杂的,我们不太可能对所有示例都能依靠它。

**自动映射**

正如你在前面一节看到的，在简单的场景下，MyBatis可以替你自动映射查询结果。 如果遇到复杂的场景，你需要构建一个result map。 但是在本节你将看到，你也可以混合使用这两种策略。 让我们到深一点的层面上看看自动映射是怎样工作的。

当自动映射查询结果时，MyBatis会获取sql返回的列名并在java类中查找相同名字的属性（忽略大小写）。 这意味着如果Mybatis发现了\_ID\_列和\_id\_属性，Mybatis会将\_ID\_的值赋给*id*。

通常数据库列使用大写单词命名，单词间用下划线分隔；而java属性一般遵循驼峰命名法。 为了在这两种命名方式之间启用自动映射，需要将 mapUnderscoreToCamelCase设置为true。

自动映射甚至在特定的result map下也能工作。在这种情况下，对于每一个result map,所有的ResultSet提供的列， 如果没有被手工映射，则将被自动映射。自动映射处理完毕后手工映射才会被处理。 在接下来的例子中， *id* 和 \_userName\_列将被自动映射， \_hashed*password* 列将根据配置映射。

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

**select**

user\_id **as** "id",

user\_name **as** "userName",

hashed\_password

**from** some\_table

**where** id = #{id}

</**select**>

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

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

</**resultMap**>

There are three auto-mapping levels:

* NONE - disables auto-mapping. Only manually mapped properties will be set.
* PARTIAL - will auto-map results except those that have nested result mappings defined inside (joins).
* FULL - auto-maps everything.

The default value is PARTIAL, and it is so for a reason. When FULL is used auto-mapping will be performed when processing join results and joins retrieve data of several different entities in the same row hence this may result in undesired mappings. To understand the risk have a look at the following sample:

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

**select**

B.id,

B.title,

A.username,

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

**where** B.id = #{id}

</**select**>

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

<**association** property="author" resultMap="authorResult"/>

</**resultMap**>

<resultMap id="authorResult" type="Author">

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

</**resultMap**>

With this result map both *Blog* and *Author* will be auto-mapped. But note that *Author* has an *id* property and there is a column named *id* in the ResultSet so Author's id will be filled with Blog's id, and that is not what you were expecting. So use the FULL option with caution.

Regardless of the auto-mapping level configured you can enable or disable the automapping for an specific ResultMap by adding the attribute autoMapping to it:

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

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

</**resultMap**>

**缓存**

MyBatis 包含一个非常强大的查询缓存特性,它可以非常方便地配置和定制。MyBatis 3 中的缓存实现的很多改进都已经实现了,使得它更加强大而且易于配置。

默认情况下是没有开启缓存的,除了局部的 session 缓存,可以增强变现而且处理循环 依赖也是必须的。要开启二级缓存,你需要在你的 SQL 映射文件中添加一行:

<**cache**/>

字面上看就是这样。这个简单语句的效果如下:

* 映射语句文件中的所有 select 语句将会被缓存。
* 映射语句文件中的所有 insert,update 和 delete 语句会刷新缓存。
* 缓存会使用 Least Recently Used(LRU,最近最少使用的)算法来收回。
* 根据时间表(比如 no Flush Interval,没有刷新间隔), 缓存不会以任何时间顺序 来刷新。
* 缓存会存储列表集合或对象(无论查询方法返回什么)的 1024 个引用。
* 缓存会被视为是 read/write(可读/可写)的缓存,意味着对象检索不是共享的,而 且可以安全地被调用者修改,而不干扰其他调用者或线程所做的潜在修改。

所有的这些属性都可以通过缓存元素的属性来修改。比如:

<**cache**

eviction="FIFO"

flushInterval="60000"

size="512"

readOnly="true"/>

这个更高级的配置创建了一个 FIFO 缓存,并每隔 60 秒刷新,存数结果对象或列表的 512 个引用,而且返回的对象被认为是只读的,因此在不同线程中的调用者之间修改它们会 导致冲突。

可用的收回策略有:

* LRU – 最近最少使用的:移除最长时间不被使用的对象。
* FIFO – 先进先出:按对象进入缓存的顺序来移除它们。
* SOFT – 软引用:移除基于垃圾回收器状态和软引用规则的对象。
* WEAK – 弱引用:更积极地移除基于垃圾收集器状态和弱引用规则的对象。

默认的是 LRU。

flushInterval(刷新间隔)可以被设置为任意的正整数,而且它们代表一个合理的毫秒 形式的时间段。默认情况是不设置,也就是没有刷新间隔,缓存仅仅调用语句时刷新。

size(引用数目)可以被设置为任意正整数,要记住你缓存的对象数目和你运行环境的 可用内存资源数目。默认值是 1024。

readOnly(只读)属性可以被设置为 true 或 false。只读的缓存会给所有调用者返回缓 存对象的相同实例。因此这些对象不能被修改。这提供了很重要的性能优势。可读写的缓存 会返回缓存对象的拷贝(通过序列化) 。这会慢一些,但是安全,因此默认是 false。

**使用自定义缓存**

除了这些自定义缓存的方式, 你也可以通过实现你自己的缓存或为其他第三方缓存方案 创建适配器来完全覆盖缓存行为。

<**cache** type="com.domain.something.MyCustomCache"/>

这个示 例展 示了 如何 使用 一个 自定义 的缓 存实 现。type 属 性指 定的 类必 须实现 org.mybatis.cache.Cache 接口。这个接口是 MyBatis 框架中很多复杂的接口之一,但是简单 给定它做什么就行。

**public** **interface** **Cache** {

String **getId**();

**int** **getSize**();

**void** **putObject**(Object key, Object value);

Object **getObject**(Object key);

**boolean** **hasKey**(Object key);

Object **removeObject**(Object key);

**void** **clear**();

}

要配置你的缓存, 简单和公有的 JavaBeans 属性来配置你的缓存实现, 而且是通过 cache 元素来传递属性, 比如, 下面代码会在你的缓存实现中调用一个称为 "setCacheFile(String file)" 的方法:

<cache type="com.domain.something.MyCustomCache">

<**property** name="cacheFile" value="/tmp/my-custom-cache.tmp"/>

</**cache**>

你可以使用所有简单类型作为 JavaBeans 的属性,MyBatis 会进行转换。

记得缓存配置和缓存实例是绑定在 SQL 映射文件的命名空间是很重要的。因此,所有 在相同命名空间的语句正如绑定的缓存一样。 语句可以修改和缓存交互的方式, 或在语句的 语句的基础上使用两种简单的属性来完全排除它们。默认情况下,语句可以这样来配置:

<**select** ... flushCache="false" useCache="true"/>

<**insert** ... flushCache="true"/>

<**update** ... flushCache="true"/>

<**delete** ... flushCache="true"/>

因为那些是默认的,你明显不能明确地以这种方式来配置一条语句。相反,如果你想改 变默认的行为,只能设置 flushCache 和 useCache 属性。比如,在一些情况下你也许想排除 从缓存中查询特定语句结果,或者你也许想要一个查询语句来刷新缓存。相似地,你也许有 一些更新语句依靠执行而不需要刷新缓存。

**参照缓存**

回想一下上一节内容, 这个特殊命名空间的唯一缓存会被使用或者刷新相同命名空间内 的语句。也许将来的某个时候,你会想在命名空间中共享相同的缓存配置和实例。在这样的 情况下你可以使用 cache-ref 元素来引用另外一个缓存。

<cache-**ref** **namespace**="com.someone.application.data.SomeMapper"/>