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)。 |

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着“authorResult”结果映射来加载作者实例。

非常重要：id元素在嵌套结果映射中扮演着非常重要的角色。你应该总是指定一个或多个可以唯一标识结果的属性。实际上如果你不指定它的话，MyBatis仍然可以工作，但是会有严重的性能问题。在可以唯一标识结果的情况下，尽可能少的选择属性。主键是一个显而易见的选择(即时是复合主键)

现在，上面的示例用了外部的结果映射关联。这使得Author结果映射可以重用。然后，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>

如果blog有一个co-author怎么办？select语句将看起来这个样子：

<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>

再次调用Author的resultMap将定义如下：

<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>

因为结果中的列名与resultMap中的列名不同。你需要指定columnPrefix去重用映射co-author结果的resultMap。

<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<Post> 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="Post">

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，首先，让我们看看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** 这个对你所有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合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resultMap。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>

有三种自动映射等级：

* NONE – 禁用自动映射。仅设置手动映射属性。
* PARTIAL – 将自动映射结果除了那些有内部定义内嵌结果映射(joins)。
* FULL – 自动映射所有。

默认值是PARTIAL，这是有原因的。当使用FULL时，自动映射会在处理join结果时执行，并且join取得若干相同的不同实体数据，因此这可能导致非预期的映射。下面的例子将展示这种风险:

<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>

在结果中Blog和Author均将自动映射。但是注意Author有一个id属性，在ResultSet中有一个列名为id，所以Author的id将被填充为Blog的id，这不是你所期待的。所以需要谨慎使用FULL。

通过添加autoMapping属性可以忽略自动映射等级配置，你可以启用或者禁用自动映射指定的ResultMap。

<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，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他们。
* 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。

默认是LRU。

flushInterval(刷新间隔)可以被设置为任意的正整数，而且他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它第三方缓存方案创建适配器来完成覆盖缓存行为。

<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框架中很复杂的借口之一，但是简单给定它做什么就行啦。

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，简单和共有的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成为“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。And you can specify a placeholder(e.g. ${cache.file})to replace value defined at [configuration properties](http://www.mybatis.org/mybatis-3/zh/configuration.html#properties).

从3.4.2版本开始，MyBatis已经支持在所有属性设置完毕以后可以调用一个初始化方法。如果你想要使用这个特性，请在你的自定义缓存类里实现org.apache.ibatis.builder.InitializingObject接口。

public interface InitializingObject {

void initialize() throws Exception;

}

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元素来引用另一个缓存。

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

# 动态SQL

MyBatis 的强大特性之一便是它的动态 SQL。如果你有使用 JDBC 或其它类似框架的经验，你就能体会到根据不同条件拼接 SQL 语句的痛苦。例如拼接时要确保不能忘记添加必要的空格，还要注意去掉列表最后一个列名的逗号。利用动态 SQL 这一特性可以彻底摆脱这种痛苦。

虽然在以前使用动态 SQL 并非一件易事，但正是 MyBatis 提供了可以被用在任意 SQL 映射语句中的强大的动态 SQL 语言得以改进这种情形。

动态 SQL 元素和 JSTL 或基于类似 XML 的文本处理器相似。在 MyBatis 之前的版本中，有很多元素需要花时间了解。MyBatis 3 大大精简了元素种类，现在只需学习原来一半的元素便可。MyBatis 采用功能强大的基于 OGNL 的表达式来淘汰其它大部分元素。

* if
* choose (when, otherwise)
* trim (where, set)
* foreach

## if

动态 SQL 通常要做的事情是根据条件包含 where 子句的一部分。比如：

<select id="findActiveBlogWithTitleLike"

resultType="Blog">

SELECT \* FROM BLOG

WHERE state = ‘ACTIVE’

<if test="title != null">

AND title like #{title}

</if>

</select>

这条语句提供了一种可选的查找文本功能。如果没有传入“title”，那么所有处于“ACTIVE”状态的BLOG都会返回；反之若传入了“title”，那么就会对“title”一列进行模糊查找并返回 BLOG 结果（细心的读者可能会发现，“title”参数值是可以包含一些掩码或通配符的）。

如果希望通过“title”和“author”两个参数进行可选搜索该怎么办呢？首先，改变语句的名称让它更具实际意义；然后只要加入另一个条件即可。

<select id="findActiveBlogLike"

resultType="Blog">

SELECT \* FROM BLOG WHERE state = ‘ACTIVE’

<if test="title != null">

AND title like #{title}

</if>

<if test="author != null and author.name != null">

AND author\_name like #{author.name}

</if>

</select>

## choose, when, otherwise

有时我们不想应用到所有的条件语句，而只想从中择其一项。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“title”就按“title”查找，提供了“author”就按“author”查找的情形，若两者都没有提供，就返回所有符合条件的 BLOG（实际情况可能是由管理员按一定策略选出 BLOG 列表，而不是返回大量无意义的随机结果）。

<select id="findActiveBlogLike"

resultType="Blog">

SELECT \* FROM BLOG WHERE state = ‘ACTIVE’

<choose>

<when test="title != null">

AND title like #{title}

</when>

<when test="author != null and author.name != null">

AND author\_name like #{author.name}

</when>

<otherwise>

AND featured = 1

</otherwise>

</choose>

</select>

## trim, where, set

前面几个例子已经合宜地解决了一个臭名昭著的动态 SQL 问题。现在回到“if”示例，这次我们将“ACTIVE = 1”也设置成动态的条件，看看会发生什么。

如果这些条件没有一个能匹配上会发生什么？最终这条 SQL 会变成这样：

SELECT \* FROM BLOG

WHERE

这会导致查询失败。如果仅仅第二个条件匹配又会怎样？这条 SQL 最终会是这样:

SELECT \* FROM BLOG

WHERE

AND title like ‘someTitle’

这个查询也会失败。这个问题不能简单地用条件句式来解决，如果你也曾经被迫这样写过，那么你很可能从此以后都不会再写出这种语句了。

MyBatis 有一个简单的处理，这在 90% 的情况下都会有用。而在不能使用的地方，你可以自定义处理方式来令其正常工作。一处简单的修改就能达到目的：

<select id="findActiveBlogLike"

resultType="Blog">

SELECT \* FROM BLOG

<where>

<if test="state != null">

state = #{state}

</if>

<if test="title != null">

AND title like #{title}

</if>

<if test="author != null and author.name != null">

AND author\_name like #{author.name}

</if>

</where>

</select>

*where* 元素只会在至少有一个子元素的条件返回 SQL 子句的情况下才去插入“WHERE”子句。而且，若语句的开头为“AND”或“OR”，*where* 元素也会将它们去除。

如果 *where* 元素没有按正常套路出牌，我们可以通过自定义 trim 元素来定制 *where* 元素的功能。比如，和 *where* 元素等价的自定义 trim 元素为：

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

...

</trim>

*prefixOverrides* 属性会忽略通过管道分隔的文本序列（注意此例中的空格也是必要的）。它的作用是移除所有指定在 *prefixOverrides* 属性中的内容，并且插入 *prefix* 属性中指定的内容。

类似的用于动态更新语句的解决方案叫做 *set*。*set* 元素可以用于动态包含需要更新的列，而舍去其它的。比如：

<update id="updateAuthorIfNecessary">

update Author

<set>

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

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

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

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

</set>

where id=#{id}

</update>

这里，*set* 元素会动态前置 SET 关键字，同时也会删掉无关的逗号，因为用了条件语句之后很可能就会在生成的 SQL 语句的后面留下这些逗号。（译者注：因为用的是“if”元素，若最后一个“if”没有匹配上而前面的匹配上，SQL 语句的最后就会有一个逗号遗留）

若你对 *set* 元素等价的自定义 trim 元素的代码感兴趣，那这就是它的真面目：

<trim prefix="SET" suffixOverrides=",">

...

</trim>

注意这里我们删去的是后缀值，同时添加了前缀值。

## foreach

动态 SQL 的另外一个常用的操作需求是对一个集合进行遍历，通常是在构建 IN 条件语句的时候。比如：

<select id="selectPostIn" resultType="domain.blog.Post">

SELECT \*

FROM POST P

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，它允许你指定一个集合，声明可以在元素体内使用的集合项（item）和索引（index）变量。它也允许你指定开头与结尾的字符串以及在迭代结果之间放置分隔符。这个元素是很智能的，因此它不会偶然地附加多余的分隔符。

**注意** 你可以将任何可迭代对象（如 List、Set 等）、Map 对象或者数组对象传递给 *foreach*作为集合参数。当使用可迭代对象或者数组时，index 是当前迭代的次数，item 的值是本次迭代获取的元素。当使用 Map 对象（或者 Map.Entry 对象的集合）时，index 是键，item 是值。

到此我们已经完成了涉及 XML 配置文件和 XML 映射文件的讨论。下一章将详细探讨 Java API，这样就能提高已创建的映射文件的利用效率。

## bind

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。比如：

<select id="selectBlogsLike" resultType="Blog">

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

SELECT \* FROM BLOG

WHERE title LIKE #{pattern}

</select>

## 多数据库支持

一个配置了“\_databaseId”变量的 databaseIdProvider 可用于动态代码中，这样就可以根据不同的数据库厂商构建特定的语句。比如下面的例子：

<insert id="insert">

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

<if test="\_databaseId == 'oracle'">

select seq\_users.nextval from dual

</if>

<if test="\_databaseId == 'db2'">

select nextval for seq\_users from sysibm.sysdummy1"

</if>

</selectKey>

insert into users values (#{id}, #{name})

</insert>

## 动态 SQL 中的可插拔脚本语言

MyBatis 从 3.2 开始支持可插拔脚本语言，这允许你插入一种脚本语言驱动，并基于这种语言来编写动态 SQL 查询语句。

可以通过实现以下接口来插入一种语言：

public interface LanguageDriver {

ParameterHandler createParameterHandler(MappedStatement mappedStatement, Object parameterObject, BoundSql boundSql);

SqlSource createSqlSource(Configuration configuration, XNode script, Class<?> parameterType);

SqlSource createSqlSource(Configuration configuration, String script, Class<?> parameterType);

}

一旦设定了自定义语言驱动，你就可以在 mybatis-config.xml 文件中将它设置为默认语言：

<typeAliases>

<typeAlias type="org.sample.MyLanguageDriver" alias="myLanguage"/>

</typeAliases>

<settings>

<setting name="defaultScriptingLanguage" value="myLanguage"/>

</settings>

除了设置默认语言，你也可以针对特殊的语句指定特定语言，可以通过如下的 lang 属性来完成：

<select id="selectBlog" lang="myLanguage">

SELECT \* FROM BLOG

</select>

或者，如果你使用的是映射器接口类，在抽象方法上加上 @Lang 注解即可：

public interface Mapper {

@Lang(MyLanguageDriver.class)

@Select("SELECT \* FROM BLOG")

List<Blog> selectBlog();

}

**注意** 可以将 Apache Velocity 作为动态语言来使用，更多细节请参考 MyBatis-Velocity 项目。

你前面看到的所有 xml 标签都是由默认 MyBatis 语言提供的，而它由别名为 xml 的语言驱动器 org.apache.ibatis.scripting.xmltags.XmlLanguageDriver 所提供。

# SQL语句构建器类

## 问题

Java程序员面对的最痛苦的事情之一就是在Java代码中嵌入SQL语句。这么来做通常是由于SQL语句需要动态来生成-否则可以将它们放到外部文件或者存储过程中。正如你已经看到的那样，MyBatis在它的XML映射特性中有一个强大的动态SQL生成方案。但有时在Java代码内部创建SQL语句也是必要的。此时，MyBatis有另外一个特性可以帮到你，在减少典型的加号,引号,新行,格式化问题和嵌入条件来处理多余的逗号或 AND 连接词之前。事实上，在Java代码中来动态生成SQL代码就是一场噩梦。例如：

String sql = "SELECT P.ID, P.USERNAME, P.PASSWORD, P.FULL\_NAME, "

"P.LAST\_NAME,P.CREATED\_ON, P.UPDATED\_ON " +

"FROM PERSON P, ACCOUNT A " +

"INNER JOIN DEPARTMENT D on D.ID = P.DEPARTMENT\_ID " +

"INNER JOIN COMPANY C on D.COMPANY\_ID = C.ID " +

"WHERE (P.ID = A.ID AND P.FIRST\_NAME like ?) " +

"OR (P.LAST\_NAME like ?) " +

"GROUP BY P.ID " +

"HAVING (P.LAST\_NAME like ?) " +

"OR (P.FIRST\_NAME like ?) " +

"ORDER BY P.ID, P.FULL\_NAME";

## 解决办法

MyBatis 3提供了方便的工具类来帮助解决该问题。使用SQL类，简单地创建一个实例来调用方法生成SQL语句。上面示例中的问题就像重写SQL类那样：

private String selectPersonSql() {

return new SQL() {{

SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FULL\_NAME");

SELECT("P.LAST\_NAME, P.CREATED\_ON, P.UPDATED\_ON");

FROM("PERSON P");

FROM("ACCOUNT A");

INNER\_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT\_ID");

INNER\_JOIN("COMPANY C on D.COMPANY\_ID = C.ID");

WHERE("P.ID = A.ID");

WHERE("P.FIRST\_NAME like ?");

OR();

WHERE("P.LAST\_NAME like ?");

GROUP\_BY("P.ID");

HAVING("P.LAST\_NAME like ?");

OR();

HAVING("P.FIRST\_NAME like ?");

ORDER\_BY("P.ID");

ORDER\_BY("P.FULL\_NAME");

}}.toString();

}

该例中有什么特殊之处？当你仔细看时，那不用担心偶然间重复出现的"AND"关键字，或者在"WHERE"和"AND"之间的选择，抑或什么都不选。该SQL类非常注意"WHERE"应该出现在何处，哪里又应该使用"AND"，还有所有的字符串链接。

## SQL类

这里给出一些示例：

// Anonymous inner class

public String deletePersonSql() {

return new SQL() {{

DELETE\_FROM("PERSON");

WHERE("ID = #{id}");

}}.toString();

}

// Builder / Fluent style

public String insertPersonSql() {

String sql = new SQL()

.INSERT\_INTO("PERSON")

.VALUES("ID, FIRST\_NAME", "#{id}, #{firstName}")

.VALUES("LAST\_NAME", "#{lastName}")

.toString();

return sql;

}

// With conditionals (note the final parameters, required for the anonymous inner class to access them)

public String selectPersonLike(final String id, final String firstName, final String lastName) {

return new SQL() {{

SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FIRST\_NAME, P.LAST\_NAME");

FROM("PERSON P");

if (id != null) {

WHERE("P.ID like #{id}");

}

if (firstName != null) {

WHERE("P.FIRST\_NAME like #{firstName}");

}

if (lastName != null) {

WHERE("P.LAST\_NAME like #{lastName}");

}

ORDER\_BY("P.LAST\_NAME");

}}.toString();

}

public String deletePersonSql() {

return new SQL() {{

DELETE\_FROM("PERSON");

WHERE("ID = #{id}");

}}.toString();

}

public String insertPersonSql() {

return new SQL() {{

INSERT\_INTO("PERSON");

VALUES("ID, FIRST\_NAME", "#{id}, #{firstName}");

VALUES("LAST\_NAME", "#{lastName}");

}}.toString();

}

public String updatePersonSql() {

return new SQL() {{

UPDATE("PERSON");

SET("FIRST\_NAME = #{firstName}");

WHERE("ID = #{id}");

}}.toString();

}

| **方法** | **描述** |
| --- | --- |
| * SELECT(String) * SELECT(String...) | 开始或插入到 SELECT子句。 可以被多次调用，参数也会添加到 SELECT子句。 参数通常使用逗号分隔的列名和别名列表，但也可以是数据库驱动程序接受的任意类型。 |
| * SELECT\_DISTINCT(String) * SELECT\_DISTINCT(String...) | 开始或插入到 SELECT子句， 也可以插入DISTINCT关键字到生成的查询语句中。 可以被多次调用，参数也会添加到 SELECT子句。 参数通常使用逗号分隔的列名和别名列表，但也可以是数据库驱动程序接受的任意类型。 |
| * FROM(String) * FROM(String...) | 开始或插入到 FROM子句。 可以被多次调用，参数也会添加到 FROM子句。 参数通常是表名或别名，也可以是数据库驱动程序接受的任意类型。 |
| * JOIN(String) * JOIN(String...) * INNER\_JOIN(String) * INNER\_JOIN(String...) * LEFT\_OUTER\_JOIN(String) * LEFT\_OUTER\_JOIN(String...) * RIGHT\_OUTER\_JOIN(String) * RIGHT\_OUTER\_JOIN(String...) | 基于调用的方法，添加新的合适类型的 JOIN子句。 参数可以包含由列命和join on条件组合成标准的join。 |
| * WHERE(String) * WHERE(String...) | 插入新的 WHERE子句条件， 由AND链接。可以多次被调用，每次都由AND来链接新条件。使用OR() 来分隔OR。 |
| OR() | 使用OR来分隔当前的 WHERE子句条件。 可以被多次调用，但在一行中多次调用或生成不稳定的SQL。 |
| AND() | 使用AND来分隔当前的 WHERE子句条件。 可以被多次调用，但在一行中多次调用或生成不稳定的SQL。因为 WHERE 和 HAVING 二者都会自动链接AND, 这是非常罕见的方法，只是为了完整性才被使用。 |
| * GROUP\_BY(String) * GROUP\_BY(String...) | 插入新的 GROUP BY子句元素，由逗号连接。 可以被多次调用，每次都由逗号连接新的条件。 |
| * HAVING(String) * HAVING(String...) | 插入新的 HAVING子句条件。 由AND连接。可以被多次调用，每次都由AND来连接新的条件。使用OR() 来分隔OR. |
| * ORDER\_BY(String) * ORDER\_BY(String...) | 插入新的 ORDER BY子句元素， 由逗号连接。可以多次被调用，每次由逗号连接新的条件。 |
| DELETE\_FROM(String) | 开始一个delete语句并指定需要从哪个表删除的表名。通常它后面都会跟着WHERE语句！ |
| INSERT\_INTO(String) | 开始一个insert语句并指定需要插入数据的表名。后面都会跟着一个或者多个VALUES() or INTO\_COLUMNS() and INTO\_VALUES()。 |
| * SET(String) * SET(String...) | 针对update语句，插入到"set"列表中 |
| UPDATE(String) | 开始一个update语句并指定需要更新的表明。后面都会跟着一个或者多个SET()，通常也会有一个WHERE()。 |
| VALUES(String, String) | 插入到insert语句中。第一个参数是要插入的列名，第二个参数则是该列的值。 |
| INTO\_COLUMNS(String...) | Appends columns phrase to an insert statement. This should be call INTO\_VALUES() with together. |
| INTO\_VALUES(String...) | Appends values phrase to an insert statement. This should be call INTO\_COLUMNS() with together. |

Since version 3.4.2, you can use variable-length arguments as follows:

public String selectPersonSql() {

return new SQL()

.SELECT("P.ID", "A.USERNAME", "A.PASSWORD", "P.FULL\_NAME", "D.DEPARTMENT\_NAME", "C.COMPANY\_NAME")

.FROM("PERSON P", "ACCOUNT A")

.INNER\_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT\_ID", "COMPANY C on D.COMPANY\_ID = C.ID")

.WHERE("P.ID = A.ID", "P.FULL\_NAME like #{name}")

.ORDER\_BY("P.ID", "P.FULL\_NAME")

.toString();

}

public String insertPersonSql() {

return new SQL()

.INSERT\_INTO("PERSON")

.INTO\_COLUMNS("ID", "FULL\_NAME")

.INTO\_VALUES("#{id}", "#{fullName}")

.toString();

}

public String updatePersonSql() {

return new SQL()

.UPDATE("PERSON")

.SET("FULL\_NAME = #{fullName}", "DATE\_OF\_BIRTH = #{dateOfBirth}")

.WHERE("ID = #{id}")

.toString();

}

## ~~SqlBuilder~~ 和 ~~SelectBuilder~~ (已经废弃)

在3.2版本之前，我们使用了一点不同的做法，通过实现ThreadLocal变量来掩盖一些导致Java DSL麻烦的语言限制。但这种方式已经废弃了，现代的框架都欢迎人们使用构建器类型和匿名内部类的想法。因此，SelectBuilder 和 SqlBuilder 类都被废弃了。

下面的方法仅仅适用于废弃的SqlBuilder 和 SelectBuilder 类。

| **方法** | **描述** |
| --- | --- |
| BEGIN()  /  RESET() | 这些方法清空SelectBuilder类的ThreadLocal状态，并且准备一个新的构建语句。开始新的语句时， BEGIN()读取得最好。 由于一些原因（在某些条件下，也许是逻辑需要一个完全不同的语句），在执行中清理语句 RESET()读取得最好。 |
| SQL() | 返回生成的 SQL() 并重置 SelectBuilder 状态 (好像 BEGIN() 或 RESET()被调用了). 因此，该方法只能被调用一次！ |

SelectBuilder 和 SqlBuilder 类并不神奇，但是知道它们如何工作也是很重要的。 SelectBuilder 使用 SqlBuilder 使用了静态导入和ThreadLocal变量的组合来开启整洁语法，可以很容易地和条件交错。使用它们，静态导入类的方法即可，就像这样(一个或其它，并非两者):

import static org.apache.ibatis.jdbc.SelectBuilder.\*;

import static org.apache.ibatis.jdbc.SqlBuilder.\*;

这就允许像下面这样来创建方法：

/\* DEPRECATED \*/

public String selectBlogsSql() {

BEGIN(); // Clears ThreadLocal variable

SELECT("\*");

FROM("BLOG");

return SQL();

}

/\* DEPRECATED \*/

private String selectPersonSql() {

BEGIN(); // Clears ThreadLocal variable

SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FULL\_NAME");

SELECT("P.LAST\_NAME, P.CREATED\_ON, P.UPDATED\_ON");

FROM("PERSON P");

FROM("ACCOUNT A");

INNER\_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT\_ID");

INNER\_JOIN("COMPANY C on D.COMPANY\_ID = C.ID");

WHERE("P.ID = A.ID");

WHERE("P.FIRST\_NAME like ?");

OR();

WHERE("P.LAST\_NAME like ?");

GROUP\_BY("P.ID");

HAVING("P.LAST\_NAME like ?");

OR();

HAVING("P.FIRST\_NAME like ?");

ORDER\_BY("P.ID");

ORDER\_BY("P.FULL\_NAME");

return SQL();

}

# 日志

Mybatis 的内置日志工厂提供日志功能，内置日志工厂将日志交给以下其中一种工具作代理：

* SLF4J
* Apache Commons Logging
* Log4j 2
* Log4j
* JDK logging

# MyBatis 工作流程

1. 加载配置并初始化

▪ 触发条件：加载配置文件

▪ 配置来源于两个地方，一处是配置文件，一处是Java代码的注解，将SQL的配置信息加载成为一个个MappedStatement对象(包括了传入参数映射配置、执行的SQL语句、结果映射配置)存储在内存中。

1. 接收调用请求

▪ 触发条件：调用MyBatis提供的API

▪ 传入参数：为SQL的ID和传入参数对象

▪ 处理过程：将请求传递给下层的请求处理层进行处理

1. 处理操作请求 触发条件：API接口层传递请求过来

▪ 传入参数：为SQL的ID和传入参数对象

▪ 处理过程：

* 1. 根据SQL的ID查找对应的MappedStatement 对象
  2. 根据传入参数对象解析iMappedStatement对象，得到最终要执行的SQL和执行传入参数
  3. 获取数据库连接，根据得到的最终SQL语句和执行传入参数到数据库执行，并得到执行结果
  4. 根据MappedStatement对象中的结果映射配置对得到的执行结果进行转换处理，并得到最终的处理结果。
  5. 释放连接资源

1. 返回处理结果将最终的处理结果返回

orm工具的基本思想：

无论是用过的hibernate、mybatis，你都可以发现它们有一个共同点：

▪ 从配置文件(通常是XML配置文件中)得到SessionFactory

▪ 由SessionFactory产生session

▪ 在session中完成对数据的增删改查和事务提交等

▪ 在用完之后关闭session

▪ 在java对象和数据库之间有做mapping的配置文件，也通常是xml文件。

# 功能架构



MyBatis的功能架构分为三层：

1. API接口层：提供给外部使用的接口API，开发人员通过这些本地API来操纵数据库。接口层接收到调用请求就会调用数据处理层来完成具体数据处理

2. 数据处理层：负责具体的SQL查找、SQL解析、SQL执行和执行结果映射处理等。它主要的目的是根据调用的请求完成一次数据库操作

3. 基础支撑层：负责最基础的功能支撑，包括连接管理、事务管理、配置加载和缓存处理，这些都是共用的东西，将他们抽取出来作为最基础的组件。为上层的数据处理层提供最基础的支撑。

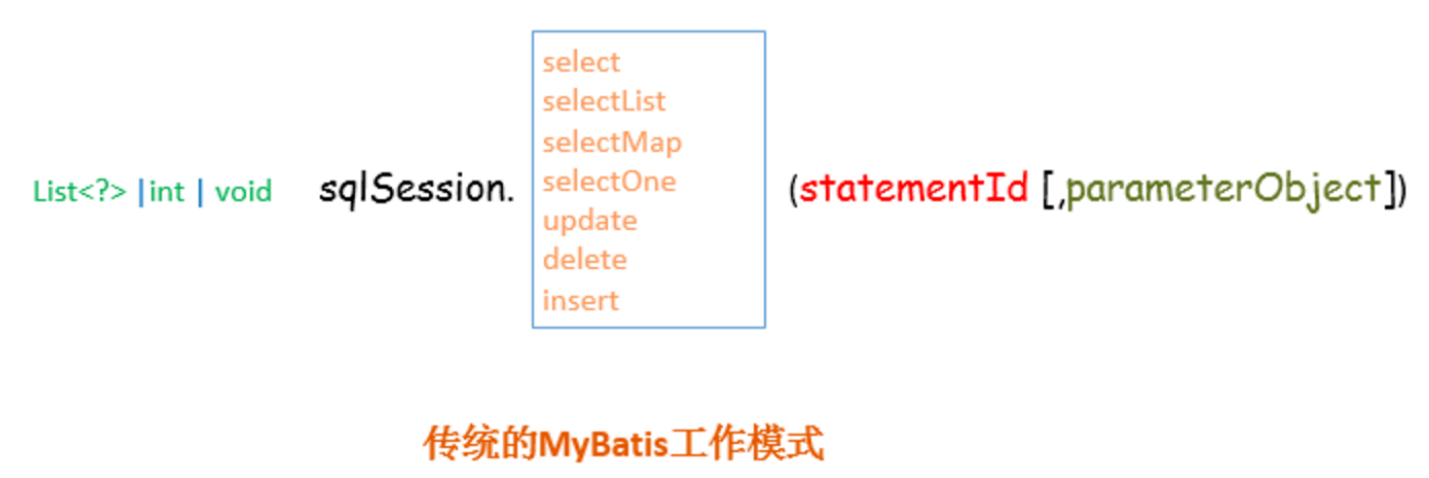
## 接口层-和数据库交互的方式

MyBatis和数据库的交互有两种方式：

* 使用传统的MyBatis提供的API；
* 使用Mapper接口；

### 使用传统的MyBatis提供的API

这是传统的传递Statement Id和查询参数给SqlSession对象，使用SqlSession对象完成和数据库的交互；MyBatis提供了非常方便和简单的API，供用户实现对数据库的增删改查操作，以及对数据库连接信息和MyBatis自身配置的维护操作。

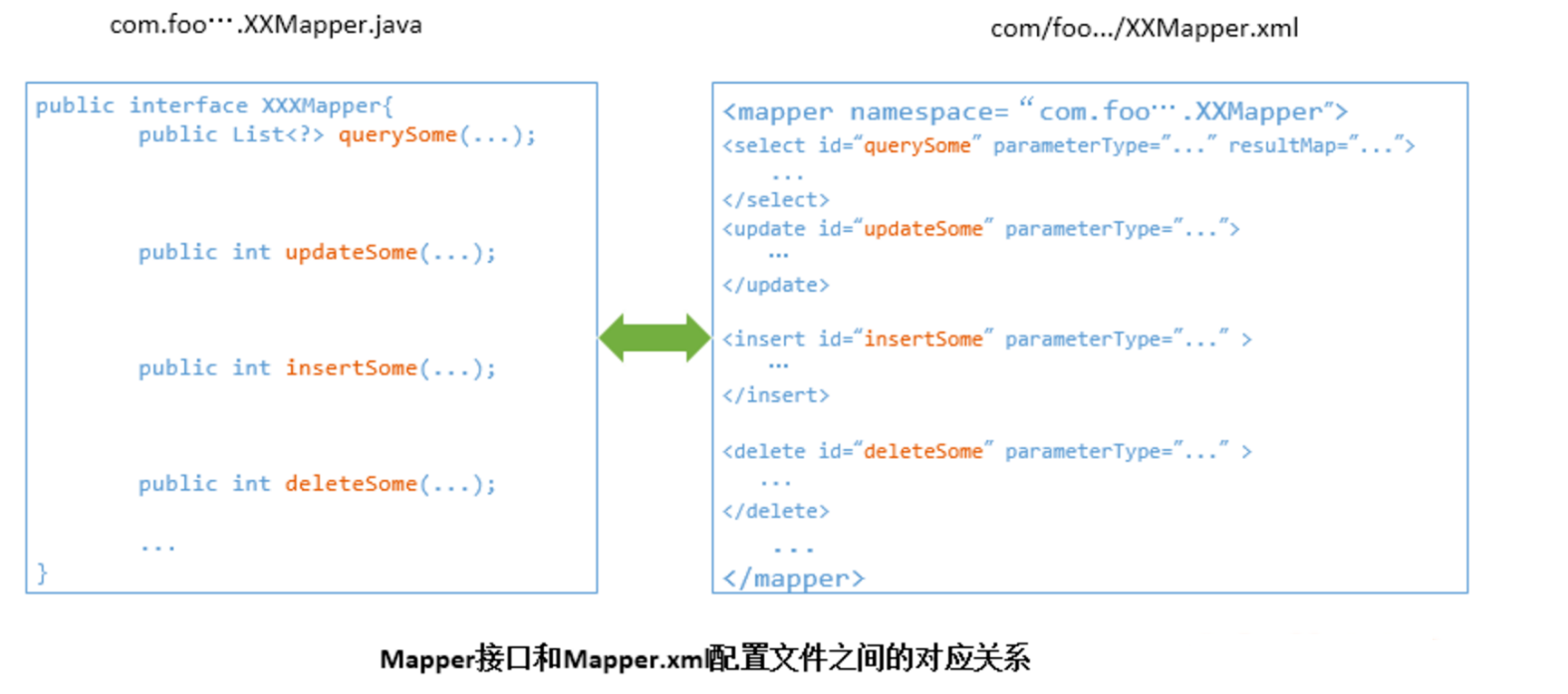


上图使用MyBatis的方法，是创建一个和数据库打交道的SqlSession对象，然后根据Statement Id和参数来操作数据库，这种方式固然很简单和使用实用，但是它不符合面向对象语言的概念和面向接口编程的编程习惯。由于面向接口的编程是面向对象的大趋势，MyBatis为了适应这一趋势，增加了第二种使用MyBatis支持接口(interface)调用方式。

### 使用Mapper接口

MyBatis将配置文件中的每一个<mapper>节点抽象为一个Mapper接口：

*这个接口中声明的方法和<mapper>节点中的<select|update|delete|insert>节点项对应，即<select|update|delete|insert>节点的id值为Mapper接口中方法名称，parameterType值表示Mapper对应方法的入参类型，而resultMap值则对应了Mapper接口表示的返回值类型或者返回结果集的元素类型。*



*根据MyBatis的配置规范配置好后，通过SqlSession.getMapper(XXXMapper.class)方法，MyBatis会根据相应的接口声明的方法信息，通过动态代理机制生成一个Mapper实例，我们使用Mapper接口的某一个方法时，MyBatis会根据这个方法名和参数类型，确定Statement Id，底层还是通过SqlSession.select(“statementId”,parameterObject);或者SqlSession.update(“statementId”,parameterObject);等等来实现对数据库的操作，MyBatis引用Mapper接口这种调用方式，纯粹是为了满足面向接口编程的需要。(其实还有一个原因是，面向接口的编程，使得用户在接口上可以使用注解来配置SQL语句，这样就可以脱离XML配置文件，实现“0配置”)*

## 数据处理层

数据处理层可以说是MyBatis的核心，从大的方面上讲，它要完成两个功能：

1. 通过传入参数构建动态SQL语句；

2. SQL语言的执行以及封装查询结果成List<E>；

### 参数映射和动态SQL语句生成

动态语句生成可以说是MyBatis框架非常优雅的设计，MyBatis通过传入的参数值，使用OGNL来动态构造SQL语句，使得MyBatis有很强的灵活性和扩展性。

参数映射指的是对于java数据类型和jdbc数据类型之间的转换：这里包括两个过程：

1. 查询阶段：将java类型的数据转换为jdbc类型的数据，通过PreparedStatement.setXXX()来设置；

2. resultset查询结果集的jdbcType数据转换成java数据类型。

### SQL语句的执行以及封装查询结果集成List<E>

动态SQL语句生成之后，MyBatis将执行SQL语句，并将可能返回的结果集转换成List<E>列表。MyBatis在对结果集的处理中，支持结果集关系一对多和多对一的转换，并且有两种支持方式，一种为嵌套查询语句的查询，一种是嵌套结果集的查询。

## 框架支承层

### 事务管理机制

事务管理机制对于ORM框架而言是不可缺少的一部分，事务管理机制的质量也是考量一个ORM框架是否优秀的一个标准。

### 连接池管理机制

由于创建一个数据库连接所占用的资源比较大，对于数据吞吐量和访问量非常大的应用而言，连接池的设计就显得非常重要。

### 缓存机制

为了提高数据利用率和减小服务器和数据库的压力，MyBatis会对于一些查询提供会话级别的数据缓存，会将对某一次查询，放置到SqlSession中，在允许的时间间隔内，对于完全相同的查询，MyBatis会直接将缓存结果返回给用户，而不同再到数据库中查找。

### SQL语句的配置方式

传统的MyBatis配置SQL语句方式就是使用XML文件进行配置的，但是这种方式不能很好的支持面向接口编程的理念，为了支持面向接口的编程，MyBatis引入了Mapper接口的概念，面向接口的引入，对使用注解来配置SQL语句成为可能，用户只需要在接口上添加必要的注解即可，不用再去配置XML文件了，但是，目前的MyBatis知识对注解配置SQL语句提供了有限的支持，某些高级功能还是要依赖XML配置文件SQL语句。

## 引导层

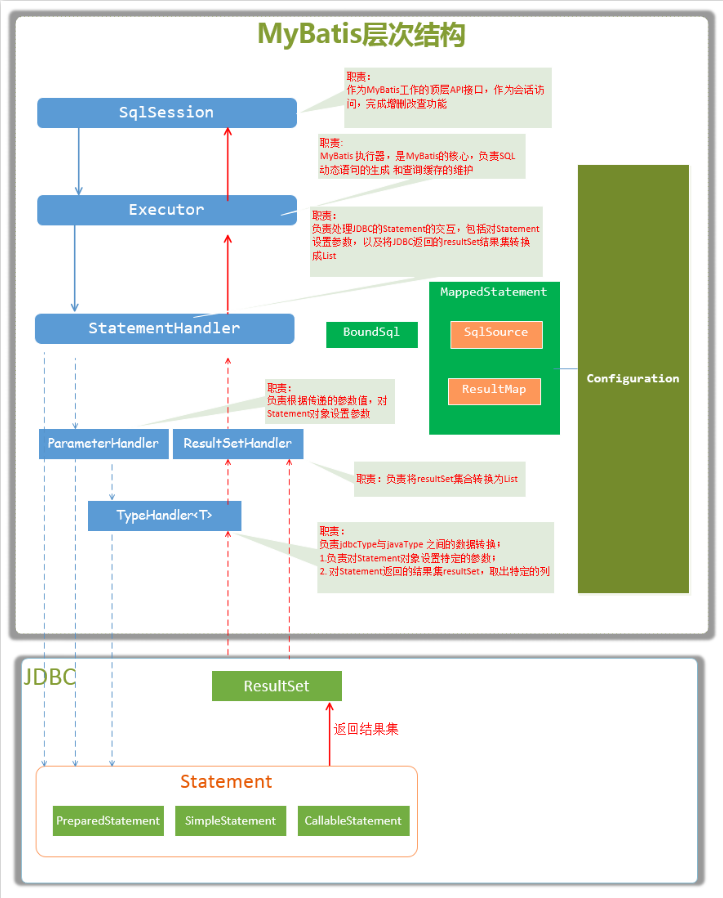
引导层是配置和启动MyBatis配置信息的方式。MyBatis提供两种方式来引导MyBatis：

1.基于XML配置文件的方式和基于Java API的方式。

## 主要构件及相互关系

|  |  |
| --- | --- |
| SqlSession | 作为MyBatis工作的主要顶层API，表示和数据库交互的会话，完成必要数据库增删改查功能 |
| Executor | MyBatis执行器，是MyBatis调度的核心，负责SQL语句的生成和查询缓存的维护 |
| StatementHandler | 封装了JDBC Statement操作，负责对JDBC statement的操作，如设置参数、将Statement结果集转换成List集合 |
| ParameterHandler | 负责对用户传递的参数转换成JDBC Statement所需要的参数 |
| ResultSetHandler | 负责将JDBC返回的ResultSet结果对象转换成List类型的集合 |
| TypeHandler | 负责java数据类型和jdbc数据类型之间的映射和转换 |
| MappedStatement | MappedStatement维护了一条<select|update|delete|insert>节点的封装 |
| SqlSource | 负责根据用户传递的parameterObject，动态地生成SQL语句，将信息封装到BoundSql对象中，并返回 |
| BoundSql | 表示动态生成的SQL语句以及相应的参数信息 |
| Configuration | MyBatis所有的配置信息都维护在Configuration对象之中 |

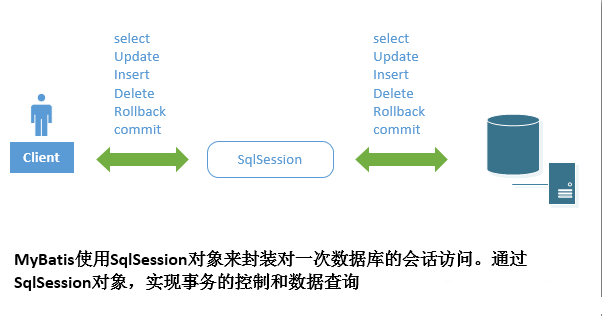
主要构件之间的关系：



### SqlSession工作过程分析

1 开启一个数据库访问会话----创建SqlSession对象

SqlSession sqlSession = factory.openSession();



2 为SqlSession传递一个配置的Sql语句的Statement Id和参数，然后返回结果：

List<Employee> result = sqlSession.selectList("com.louis.mybatis.dao.EmployeesMapper.selectByMinSalary",params);

上述的“com.louis.mybatis.dao.EmployeesMapper.selectByMinSalary”，是配置在EmployeesMapper.xml的Statement ID，params是传递的查询参数。

我们来看一下sqlSession.selectList()方法的定义：

public <E> List<E> selectList(String statement, Object parameter) {

return this.selectList(statement, parameter, RowBounds.DEFAULT);

}

public <E> List<E> selectList(String statement, Object parameter, RowBounds rowBounds) {

try {

//1.根据Statement Id，在mybatis 配置对象Configuration中查找和配置文件相对应的MappedStatement

MappedStatement ms = configuration.getMappedStatement(statement);

//2. 将查询任务委托给MyBatis 的执行器 Executor

List<E> result = executor.query(ms, wrapCollection(parameter), rowBounds, Executor.NO\_RESULT\_HANDLER);

return result;

} catch (Exception e) {

throw ExceptionFactory.wrapException("Error querying database. Cause: " + e, e);

} finally {

ErrorContext.instance().reset();

}

};

MyBatis在初始化的时候，会将MyBatis的配置信息全部加载到内存中，使用org.apache.ibatis.session.Configuration实例来维护。使用者可以使用sqlSession.getConfiguration()方法来获取。MyBatis的配置文件中配置信息的组织格式和内存中对象的组织格式集合完全对应的。

上述例子中的：

<select id="selectByMinSalary" resultMap="BaseResultMap" parameterType="java.util.Map" >

select

EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, SALARY

from LOUIS.EMPLOYEES

<if test="min\_salary != null">

where SALARY < #{min\_salary,jdbcType=DECIMAL}

</if>

</select>

加载到内存中会生成一个对应的MappedStatement对象，然后会以key=”com.Iouis.mybatis.dao.EmployeesMapper.selectByMinSalary”，value为MapperStatement对象的形式维护到Configuration的一个Map中。当以后需要使用的时候，只需要通过Id值来获取就可以了。

上述代码中我们可以看到SqlSession的职能是：SqlSession根据Statement ID，在mybatis配置对象Configuration中获取到对应的MappedStatement对象，然后调用mybatis执行器来执行具体的操作。

3 MyBatis执行器Executor根据SqlSession传递的参数执行query()方法(由于代码过长，读者只需要阅读我注释的地方即可)：

/\*\*

\* BaseExecutor 类部分代码

\*

\*/

public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler) throws SQLException {

// 1. 根据具体传入的参数，动态地生成需要执行的SQL语句，用BoundSql对象表示

BoundSql boundSql = ms.getBoundSql(parameter);

// 2. 为当前的查询创建一个缓存Key

CacheKey key = createCacheKey(ms, parameter, rowBounds, boundSql);

return query(ms, parameter, rowBounds, resultHandler, key, boundSql);

}

@SuppressWarnings("unchecked")

public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws SQLException {

ErrorContext.instance().resource(ms.getResource()).activity("executing a query").object(ms.getId());

if (closed) throw new ExecutorException("Executor was closed.");

if (queryStack == 0 && ms.isFlushCacheRequired()) {

clearLocalCache();

}

List<E> list;

try {

queryStack++;

list = resultHandler == null ? (List<E>) localCache.getObject(key) : null;

if (list != null) {

handleLocallyCachedOutputParameters(ms, key, parameter, boundSql);

} else {

// 3.缓存中没有值，直接从数据库中读取数据

list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key, boundSql);

}

} finally {

queryStack--;

}

if (queryStack == 0) {

for (DeferredLoad deferredLoad : deferredLoads) {

deferredLoad.load();

}

deferredLoads.clear(); // issue #601

if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {

clearLocalCache(); // issue #482

}

}

return list;

}

private <E> List<E> queryFromDatabase(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws SQLException {

List<E> list;

localCache.putObject(key, EXECUTION\_PLACEHOLDER);

try {

//4. 执行查询，返回List 结果，然后 将查询的结果放入缓存之中

list = doQuery(ms, parameter, rowBounds, resultHandler, boundSql);

} finally {

localCache.removeObject(key);

}

localCache.putObject(key, list);

if (ms.getStatementType() == StatementType.CALLABLE) {

localOutputParameterCache.putObject(key, parameter);

}

return list;

}

/\*\*

\*

\* SimpleExecutor类的doQuery()方法实现

\*

\*/

public <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) throws SQLException {

Statement stmt = null;

try {

Configuration configuration = ms.getConfiguration();

//5. 根据既有的参数，创建StatementHandler对象来执行查询操作

StatementHandler handler = configuration.newStatementHandler(wrapper, ms, parameter, rowBounds, resultHandler, boundSql);

//6. 创建java.Sql.Statement对象，传递给StatementHandler对象

stmt = prepareStatement(handler, ms.getStatementLog());

//7. 调用StatementHandler.query()方法，返回List结果集

return handler.<E>query(stmt, resultHandler);

} finally {

closeStatement(stmt);

}

}

上述的Executor.query()方法几经转折，最后会创建一个StatementHandler对象，然后将必要的参数传递给StatementHandler，使用StatementHandler来完成对数据库的查询，最终返回List结果集。

从上面的代码中我们可以看出，Executor的功能和作用是：

a. 根据传入的参数，完成SQL语句的动态解析，生成BoundSql对象，供StatementHandler使用；

b. 为查询创建缓存，以提高性能；

c. 创建JDBC的Statement连接对象，传递给StatementHandler对象，返回List查询结果；

4 StatementHandler对象负责设置Statement对象中的查询参数、处理JDBC返回的resultSet，将resultSet加工为List集合返回：

接着上面的Executor第六步，看一下：prepareStatement()方法的实现：

/\*\*

\*

\* SimpleExecutor类的doQuery()方法实现

\*

\*/

public <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) throws SQLException {

Statement stmt = null;

try {

Configuration configuration = ms.getConfiguration();

StatementHandler handler = configuration.newStatementHandler(wrapper, ms, parameter, rowBounds, resultHandler, boundSql);

// 1.准备Statement对象，并设置Statement对象的参数

stmt = prepareStatement(handler, ms.getStatementLog());

// 2. StatementHandler执行query()方法，返回List结果

return handler.<E>query(stmt, resultHandler);

} finally {

closeStatement(stmt);

}

}

private Statement prepareStatement(StatementHandler handler, Log statementLog) throws SQLException {

Statement stmt;

Connection connection = getConnection(statementLog);

stmt = handler.prepare(connection);

//对创建的Statement对象设置参数，即设置SQL 语句中 ? 设置为指定的参数

handler.parameterize(stmt);

return stmt;

}

以上我们可以总结StatementHandler对象主要完成两个工作：

a 对于JDBC的PreparedStatement类型的对象，创建的过程中，我们使用的是SQL语句字符串会包含若干个？占位符，我们其后再对占位符进行设置。StatementHandler通过parameterize(statement)方法对Statement进行设值；

b StatementHandler通过List<E>query(Statement statement, ResultHandler resultHandler)方法来完成执行Statement，和将Statement对象返回的resultSet封装成List；

5 StatementHandler的parameterize(statement)方法的实现：

/\*\*

\* StatementHandler 类的parameterize(statement) 方法实现

\*/

public void parameterize(Statement statement) throws SQLException {

//使用ParameterHandler对象来完成对Statement的设值

parameterHandler.setParameters((PreparedStatement) statement);

}

/\*\*

\*

\* ParameterHandler类的setParameters(PreparedStatement ps) 实现

\* 对某一个Statement进行设置参数

\*/

public void setParameters(PreparedStatement ps) throws SQLException {

ErrorContext.instance().activity("setting parameters").object(mappedStatement.getParameterMap().getId());

List<ParameterMapping> parameterMappings = boundSql.getParameterMappings();

if (parameterMappings != null) {

for (int i = 0; i < parameterMappings.size(); i++) {

ParameterMapping parameterMapping = parameterMappings.get(i);

if (parameterMapping.getMode() != ParameterMode.OUT) {

Object value;

String propertyName = parameterMapping.getProperty();

if (boundSql.hasAdditionalParameter(propertyName)) { // issue #448 ask first for additional params

value = boundSql.getAdditionalParameter(propertyName);

} else if (parameterObject == null) {

value = null;

} else if (typeHandlerRegistry.hasTypeHandler(parameterObject.getClass())) {

value = parameterObject;

} else {

MetaObject metaObject = configuration.newMetaObject(parameterObject);

value = metaObject.getValue(propertyName);

}

// 每一个Mapping都有一个TypeHandler，根据TypeHandler来对preparedStatement进行设置参数

TypeHandler typeHandler = parameterMapping.getTypeHandler();

JdbcType jdbcType = parameterMapping.getJdbcType();

if (value == null && jdbcType == null) jdbcType = configuration.getJdbcTypeForNull();

// 设置参数

typeHandler.setParameter(ps, i + 1, value, jdbcType);

}

}

}

}

从上述的代码可以看到，StatementHandler的parameterize(Statement)方法调用了ParameterHandler的setParameters(statement)方法，ParameterHandler的setParameters(Statement)方法负责根据我们输入的参数，对statement对象的”?”占位符处进行赋值。

6 StatementHandler的List<E>query(Statement statement, ResultHandler resultHandler)方法的实现：

/\*\*

\* PreParedStatement类的query方法实现

\*/

public <E> List<E> query(Statement statement, ResultHandler resultHandler) throws SQLException {

//1.调用preparedStatemnt。execute()方法，然后将resultSet交给ResultSetHandler处理

PreparedStatement ps = (PreparedStatement) statement;

ps.execute();

//2. 使用ResultHandler来处理ResultSet

return resultSetHandler.<E> handleResultSets(ps);

}

从上述代码我们可以看出，StatementHandler的List<E>query(Statement statement, ResultHandler resultHandler)方法的实现，是调用了ResultSetHandler的handleResultSets(Statement)方法。ResultSetHandler的handleResultSets(Statement)方法会将Statement语句执行后生成的resultSet结果集转换成List<E>结果集：

/\*\*

\* ResultSetHandler类的handleResultSets()方法实现

\*

\*/

public List<Object> handleResultSets(Statement stmt) throws SQLException {

final List<Object> multipleResults = new ArrayList<Object>();

int resultSetCount = 0;

ResultSetWrapper rsw = getFirstResultSet(stmt);

List<ResultMap> resultMaps = mappedStatement.getResultMaps();

int resultMapCount = resultMaps.size();

validateResultMapsCount(rsw, resultMapCount);

while (rsw != null && resultMapCount > resultSetCount) {

ResultMap resultMap = resultMaps.get(resultSetCount);

//将resultSet

handleResultSet(rsw, resultMap, multipleResults, null);

rsw = getNextResultSet(stmt);

cleanUpAfterHandlingResultSet();

resultSetCount++;

}

String[] resultSets = mappedStatement.getResulSets();

if (resultSets != null) {

while (rsw != null && resultSetCount < resultSets.length) {

ResultMapping parentMapping = nextResultMaps.get(resultSets[resultSetCount]);

if (parentMapping != null) {

String nestedResultMapId = parentMapping.getNestedResultMapId();

ResultMap resultMap = configuration.getResultMap(nestedResultMapId);

handleResultSet(rsw, resultMap, null, parentMapping);

}

rsw = getNextResultSet(stmt);

cleanUpAfterHandlingResultSet();

resultSetCount++;

}

}

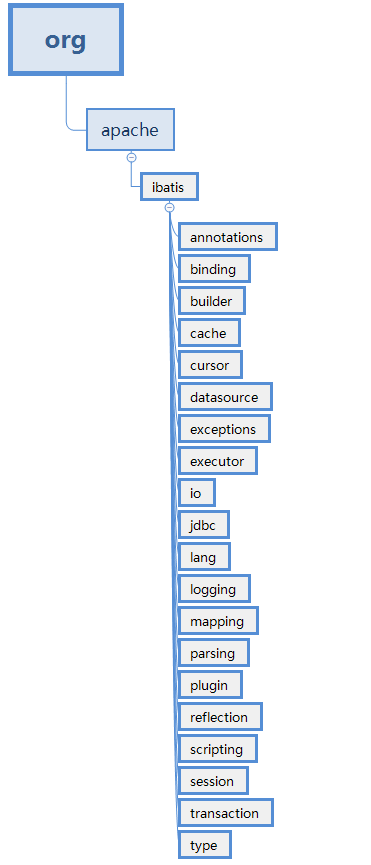
return collapseSingleResultList(multipleResults);

}

# MyBatis整个项目包结构

官方GitHub项目地址：

<https://github.com/mybatis/mybatis-3/tree/master/src/main/java/org/apache/ibatis>



①mybatis通过SqlSessionFactoryBuilder作为入口，通过传入配置文件，使用BaseBuilder实现类进行配置文件解析，具体实现类是XMLConfigBuilder，在这里mybatis对配置的项进行了全面解析，只不过不是所有的解析都放在了 XMLConfigBuilder，XMLConfigBuilder解析了二级节点，并作为一个总入口，还有另外集合类继承了BaseBuilder，用于解析不同的配置。而解析到的配置项信息，基本都保存在了Configuration这个类，可以看到多处地方依赖到它。

之后通过配置获取到SqlSessionFactory，可以看到SqlSessionFactoryBuilder提供了一个build方法，就是返回SqlSessionFactory的。

②而SqlSessionFactory只是一个接口，默认实现由DefaultSqlSessionFactory。通过SqlSessionFactory.openSession的几个多态方法，可以获取到SqlSession。

③而SqlSession也只是一个接口，怎么理解SqlSession呢，其实Session即是一次会话，加入我们把某次请求与数据库连接Connection的交互，看成是一次会话，那就可以理解SqlSession了。SqlSession是请求与数据库的一次交互，交互完成之后，Session则结束，但Session结束不代表数据库连接也关闭，知识这次会话终止了，但是数据库连接以旧是返回给连接池。

这里的SqlSession即是对一次交互信息的封装，请求可以通过SqlSession的方法进行操作，而数据库连接，以旧由SqlSessionFactory注入。

④SqlSessionFactory获取到TransactionFactory，一个事务的工厂类接口，而通过这个工厂具体实现类可以产生多个事务Transaction(具体参考ManagedTransactionFactory),Transaction抽象了一个方法getConnection()，通过这个方法我们可以获取到不同事务级别的connection。mybatis的代码是这样的：

|  |
| --- |
| @Override **public** Connection getConnection() **throws** SQLException {  **if** (**this**.**connection** == **null**) {  openConnection();  }  **return this**.**connection**; }  **protected void** openConnection() **throws** SQLException {  **if** (***log***.isDebugEnabled()) {  ***log***.debug(**"Opening JDBC Connection"**);  }  **this**.**connection** = **this**.**dataSource**.getConnection();  **if** (**this**.**level** != **null**) {  **this**.**connection**.setTransactionIsolation(**this**.**level**.getLevel());  } } |

⑤这个时候可能大家觉得万事俱备了，SqlSession都定义好了CRUD方法了，那它也会帮忙将语句扔给Connection操作，其实不然。这里mybatis还抽象了接口Execute。

Execute才是真正将sql与Connection打交道的地方。

# MyBatis初始化

任何框架的初始化，无非是加载自己运行时所需要的配置信息。MyBatis的配置信息，大概包含以下信息，其高层级结构如下：

* configuration配置

▪ properties属性

▪ settings设置

▪ typeAliases类型命名

▪ typeHandlers类型处理器

▪ objectFactory对象工厂

▪ plugins插件

▪ environments环境

environment环境变量

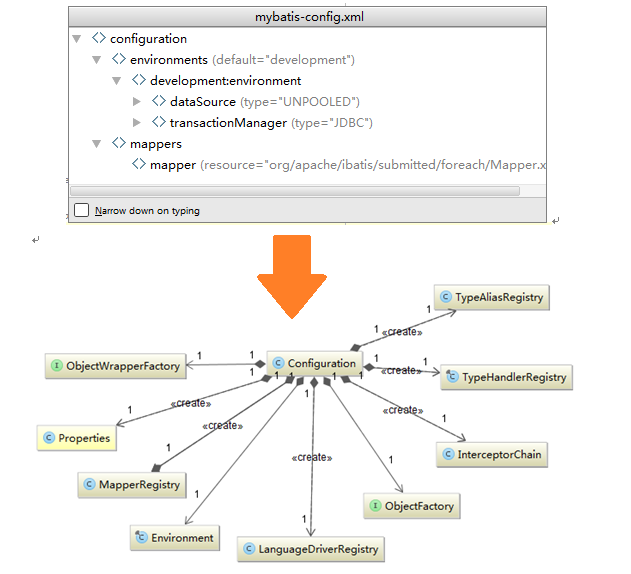
transactionManager事务管理器

dataSource数据源

* 映射器

MyBatis的上述配置信息会配置在XML配置文件中，那么，这些信息被加载进入MyBatis内部，MyBatis是怎样维护的呢？

使用org.apache.ibatis.session.Configuration对象作为一个所有配置信息的容器，Configuration对象的组织结构和XML配置文件的组织结构几乎完全一样(Configuration对象的功能并不限于此，它还负责创建一些MyBatis内部使用的对象，如Executor等)。如下图所示：



MyBatis根据初始化好Configuration信息，这时候用户就可以使用MyBatis进行数据库操作了。

简单的说，MyBatis初始化的过程，就是创建Configuration对象的过程。

## MyBatis初始化的两种初始化方式

* 基于XML配置文件：基于XML配置文件的方式是将MyBatis的所有配置信息放在XML文件中，MyBatis通过加载XML配置文件，将配置文件信息组装成内部的Configuration对象
* 基于Java API：这种方式不使用XML配置文件，需要MyBatis使用者在Java代码中，手动创建Configuration对象，然后将配置参数set进入Configuration对象中

### MyBatis基于XML配置文件创建Configuration对象的过程

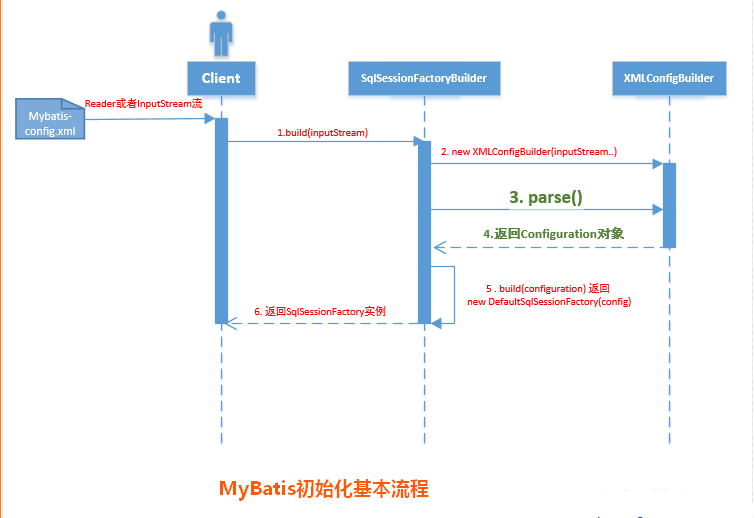
废话少说，直接看一个使用Mybatis的小例子：

1. String resource = "mybatis-config.xml";
2. InputStream inputStream = Resources.getResourceAsStream(resource);
3. SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);
4. SqlSession sqlSession = sqlSessionFactory.openSession();
5. List list = sqlSession.selectList("com.foo.bean.BlogMapper.queryAllBlogInfo");

上述代码执行了一个查询并返回一个List结果集，此过程中经历了：mybatis初始化->创建SqlSession->执行SQL语句并返回结果三个过程。

其中mybatis的初始化就发生在第三句。

SqlSessionFactoryBuilder根据传入的数据流生成Configuration对象，然后根据Configuration对象创建默认的SqlSessionFactory实例。



1. 调用SqlSessionFactoryBuilder对象的builder(inputStream)方法；

2. SqlSessionFactoryBuilder会根据输入流InputStream等信息创建XMLConfigBuilder对象；

3. SqlSessionFactoryBuilder调用XMLConfigBuilder对象的parse()方法；

4. XMLConfigBuilder对象返回Configuration对象；

5. SqlSessionFactoryBuilder根据Configuration对象创建一个DefaultSessionFactory对象；

6. SqlSessionFactoryBuilder返回DefaultSessionFactory对象给Client，供Client使用。

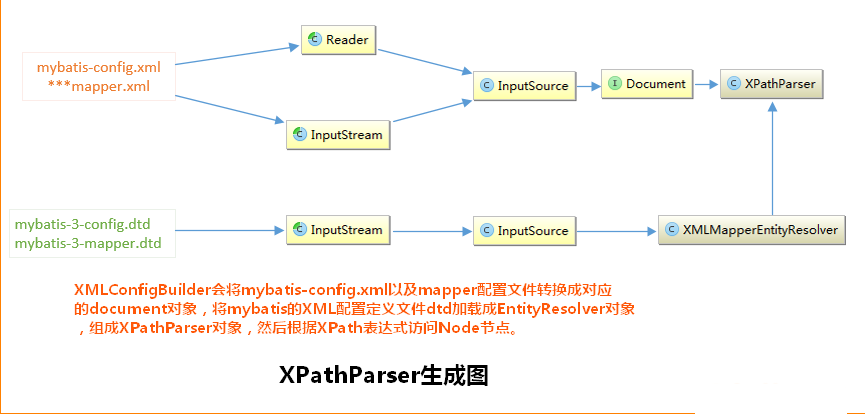
上述的初始化过程中，涉及到一下几个对象：

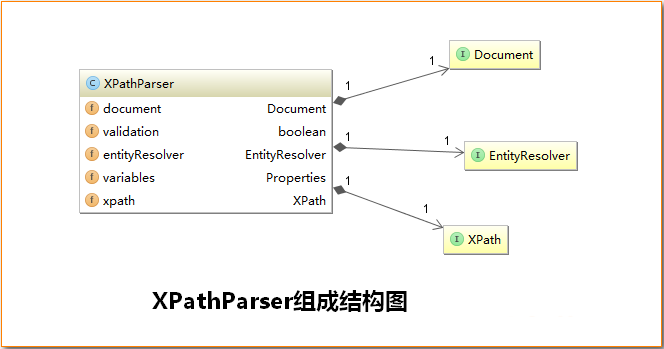
* SqlSessionFactoryBuilder：SqlSessionFactory的构造器，用于创建SqlSessionFactory，采用了Builder设计模式
* Configuration：该对象是mybatis-config.xml文件中所有mybatis配置信息
* SqlSessionFactory：SqlSession工厂类，以工厂形式创建SqlSession对象，采用了Factory工厂设计模式
* XmlConfigParser：负责将mybatis-config.xml配置文件解析成Configuration对象，共SqlSessionFactoryBuilder使用，创建SqlSessionFactory。

### 创建Configuration对象的过程

上文MyBatis初始化基本过程中，当SqlSessionFactoryBuilder执行build()方法，调用了XMLConfigBuilder的parse()方法，然后返回了Configuration对象。那么parse()方法是如何处理XML文件，生成Configuration对象呢？

1. XMLConfigBuilder会将XML配置文件的信息转换为Document对象，而XML配置定义文件DTD转换成XMLMapperEntiryResolver对象，然后将二者封装到XpathParsar对象中，XpathParse的作用是提供根据Xpath表达式获取基本的DOM节点Node信息的操作。如下图所示：



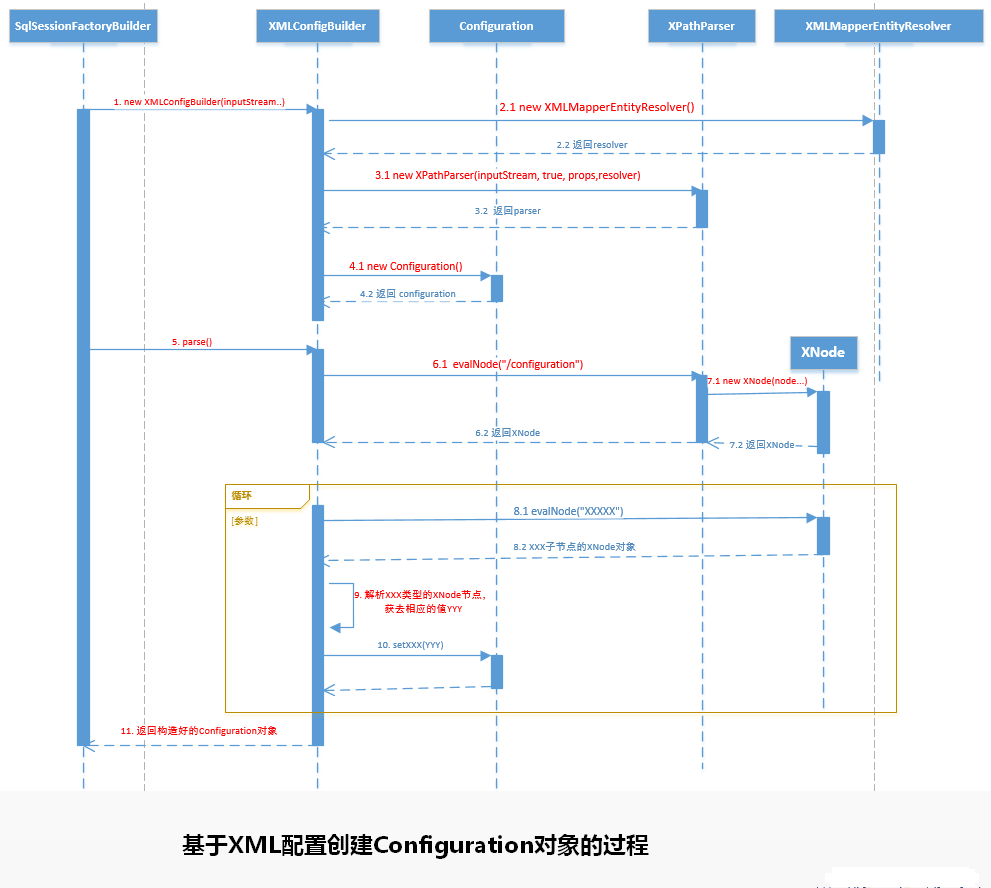


2. 之后XMLConfigBuilder调用parse()方法：会从XPathParser中取出<configuration>节点对应的Node对象，然后解析此Node节点的子Node：properties，settings,typeAliases,typeHandlers,objectFactory,objectWrapperFactory,plugins,environments,databaseIdProvider,mappers

3. 然后将这些值解析出来设置到Configuration对象中

4. 返回Configuration对象

上述MyBatis初始化基本过程的序列图细化如下：

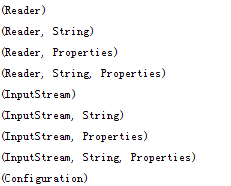


## 涉及到的设计模式

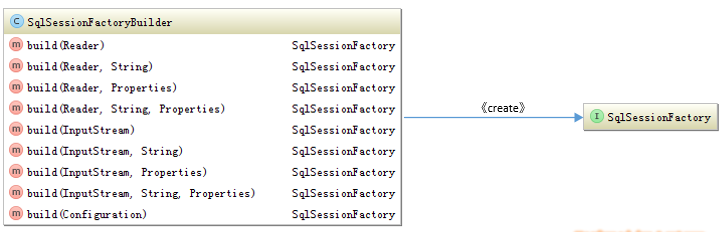
初始化的过程涉及到创建各种对象，所以会使用一些创建型的设计模式。在初始化的过程中，Builder模式运行的比较多。

### Builder模式应用1：SqlSessionFactory创建

对于创建SqlSessionFactory时，会根据情况提供不同的参数，其参数组合可以有一下几种：



由于构造时参数不定，可以为其创建一个构造器Builder，将SqlSessionFactory的构建过程和表示分开：



MyBatis将SqlSessionFactoryBuilder和SqlSessionFactory互相独立。

### Builder模式应用2：数据库连接环境Environment对象的创建

在构建Configuration对象的过程中，XMLConfigBuilder解析mybatis XML配置文件节点<environment>节点时，会有以下相应的代码：

|  |
| --- |
| **private void** environmentsElement(XNode context) **throws** Exception {  **if** (context != **null**) {  **if** (**environment** == **null**) {  **environment** = context.getStringAttribute(**"default"**);  }  **for** (XNode child : context.getChildren()) {  String id = child.getStringAttribute(**"id"**);  **if** (isSpecifiedEnvironment(id)) {  TransactionFactory txFactory = transactionManagerElement(child.evalNode(**"transactionManager"**));  DataSourceFactory dsFactory = dataSourceElement(child.evalNode(**"dataSource"**));  DataSource dataSource = dsFactory.getDataSource();  Environment.Builder environmentBuilder = **new** Environment.Builder(id)  .transactionFactory(txFactory)  .dataSource(dataSource);  **configuration**.setEnvironment(environmentBuilder.build());  }  }  } } |

在Environment内部，定义了静态内部Builder类：

|  |
| --- |
| **public final class** Environment {  **private final** String **id**;  **private final** TransactionFactory **transactionFactory**;  **private final** DataSource **dataSource**;   **public** Environment(String id, TransactionFactory transactionFactory, DataSource dataSource) {  **if** (id == **null**) {  **throw new** IllegalArgumentException(**"Parameter 'id' must not be null"**);  }  **if** (transactionFactory == **null**) {  **throw new** IllegalArgumentException(**"Parameter 'transactionFactory' must not be null"**);  }  **this**.**id** = id;  **if** (dataSource == **null**) {  **throw new** IllegalArgumentException(**"Parameter 'dataSource' must not be null"**);  }  **this**.**transactionFactory** = transactionFactory;  **this**.**dataSource** = dataSource;  }   **public static class** Builder {  **private** String **id**;  **private** TransactionFactory **transactionFactory**;  **private** DataSource **dataSource**;   **public** Builder(String id) {  **this**.**id** = id;  }   **public** Builder transactionFactory(TransactionFactory transactionFactory) {  **this**.**transactionFactory** = transactionFactory;  **return this**;  }   **public** Builder dataSource(DataSource dataSource) {  **this**.**dataSource** = dataSource;  **return this**;  }   **public** String id() {  **return this**.**id**;  }   **public** Environment build() {  **return new** Environment(**this**.**id**, **this**.**transactionFactory**, **this**.**dataSource**);  }   }   **public** String getId() {  **return this**.**id**;  }   **public** TransactionFactory getTransactionFactory() {  **return this**.**transactionFactory**;  }   **public** DataSource getDataSource() {  **return this**.**dataSource**;  }  } |