MyBatis 3 用户指南中文版

MyBatis 3.0.1

曾令祝
2010.06.15

目录

第一章 什么是 MyBatis 3
第二章 入门 4
一、从 XML 中创造 SqlSessionFactory 4
二、不使用 XML 文件新建 SqlSessionFactory 5
三、使用 SqlSessionFactory 获取 SqlSession 5
四、探究 SQL 映射语句 6
五、作用域和生命周期 7
1. SqlSessionFactoryBuilder 7
2. SqlSessionFactory 7
3. SqlSession 7
4. Mapper 实例 8
第三章 Mapper 的 XML 配置文件 9
一、属性（properties） 9
二、设置（settings） 10
三、类型别名（typeAliases） 11
四、类型句柄（typeHandlers） 12
五、对象工厂（ObjectFactory） 13
六、插件（plugins） 14
七、环境（environments） 15
八、映射器（Mappers） 18
第四章 SQL 映射语句文件 19
一、select 19
二、insert, update, delete 20
三、SQL 23
四、参数（parameters） 23
五、resultMap 25
六、缓存（cache） 37
七、cache-ref 缓存引用 39
第五章 动态语句 40
if 40
choose, when, otherwise 41
trim, where, set 41
foreach 43
第六章 Java API 45
一、目录结构 45
二、SqlSessions 46
三、SqlSession 49
第七章 SelectBuilder 55
第八章 SqlBuilder 58
第九章 说明 59

第一章 什么是 MyBatis

MyBatis 世界上流行最广泛的 SQL 映射框架，由 Clinton Begin 在 2002 年创建，其后，捐献给了 Apache 基金会，成立了 iBatis 项目。2010 年 5 月，将代码库迁致 Google Code，并更名为 MyBatis。
关于以前的版本，请访问 http://ibatis.apache.org/
关于更名后最新的版本，请访问：http://code.google.com/p/mybatis/

MyBatis 是一个可以自定义 SQL、存储过程和高级映射的持久层框架。MyBatis 摒除了大部分的 JDBC 代码、手工设置参数和结果集重获。MyBatis 只使用简单的 XML 和注解来配置和映射基本数据类型、Map 接口和 POJO 到数据库记录。

第二章 入门

每一个 MyBatis 应该都是以一个 SqlSessionFactory 实例为中心。一个 SqlSessionFactory 实例可以使用 SqlSessionFactoryBuilder 来创造。从配置类中创造的定制 SqlSessionFactoryBuilder 实例，可以使用 XML 配置文件来生成一个 SqlSessionFactory 实例。

一、从 XML 中创造 SqlSessionFactory

从 XML 文件中创造 SqlSessionFactory 实例是非常简单的。推荐使用一个类路径资源来进行配置，你也可以使用一个 Reader 实例，甚至使用 URL 路径。

MyBatis 有一个 Resources 通用类，类中有许多方法可以简单地从类路径和其他地址中加载资源。

```
String resource = "org/mybatis/example/Configuration.xml";
Reader reader = Resources.getResourceAsReader(resource);
sqlMapper = new SqlSessionFactoryBuilder().build(reader);
```

XML 文件包含了许多 MyBatis 的核心设置，包括一个获取数据库连接（Connection）实例的数据源（DataSource），和一个决定事务作用域和操作的 TransactionManager。全部的 XML 配置文件的内容将在以后提到，先给出一个简单的样子。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
    PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  <environments default="development">
      <environment id="development">
      <transactionManager type="JDBC"/>
      <dataSource type="POOLED">
        <property name="driver" value="${driver}"/>
        <property name="url" value="${url}"/>
        <property name="username" value="${username}"/>
        <property name="password" value="${password}"/>
      </dataSource>
    </environment>
  </environments>
  <mappers>
    <mapper resource="org/mybatis/example/BlogMapper.xml"/>
  </mappers>
</configuration>
```

XML 配置文件中还有其它许多内容，上面的例子只是指出了最重要的部分。注意这个 XML 的标头，需要一个 DTD 验证文档。environment 项里包含了事务管理和连接池的环境配置。 mappers 项中包含了一系列 SQL 语句映射定义的 XML 文件。

二、不使用 XML 文件新建 SqlSessionFactory

如果你更想直接使用Java语言而不是XML来生成这些配置，更或者你想使用自己的配置生成器，MyBatis提供了一个完整的配置类来完成 XML 文件一样的配置。

```
DataSource dataSource = BlogDataSourceFactory.getBlogDataSource();
TransactionFactory transactionFactory = new JdbcTransactionFactory();
Environment environment =
  new Environment("development", transactionFactory, dataSource);
Configuration configuration = new Configuration(environment);
configuration.addMapper(BlogMapper.class);
SqlSessionFactory sqlSessionFactory =
  new SqlSessionFactoryBuilder().build(configuration);
```

这个配置里，加载了一个映射类。映射类是包含了 SQL 映射注解的 Java 类，可以用来取代 XML。然而，由于 Java 注解的一些限制和 MyBatis 映射的复杂性，一些高级的映射还是要用 XML 来配置，比如嵌套映射等。由于这个原因，MyBatis 会自动查找和加载已经存在的 XML。比如说上面的代码，BlogMapper.xml 将会被类路径中 BlogMapper.class 加载。以后会详细讨论这些。

三、使用 SqlSessionFactory 获取 SqlSession

假设你有一个 SqlSessionFactory，你就可以来获取一个 SqlSession 实例，SqlSession 包含了针对数据库执行语句的每一个方法。你可以直接使用 SqlSession 执行已经映射的每一个 SQL 语句。比如：

```
SqlSession session = sqlMapper.openSession();
try {
  Blog blog = (Blog) session.select(
    "org.mybatis.example.BlogMapper.selectBlog", 101);
} finally {
  session.close();
}
```

上述步骤对于使用 MyBatis 的上一个版本（即 iBatis 2）的用户来说比较熟悉。现在，有一个更加清晰的方式。使用一个有正确参数和返回值的接口，你就可以更加清晰和安全地编写代码，从而避免出错。像这样：

```
SqlSession session = sqlSessionFactory.openSession();
try {
  BlogMapper mapper = session.getMapper(BlogMapper.class);
  Blog blog = mapper.selectBlog(101);
} finally {
  session.close();
}
```

现在，让我们开始探究一下步骤的执行细节。

四、探究 SQL 映射语句

对于上面所说的，你可能很好奇 SqlSession 或 Mapper 类具体是什么执行的。这是一个很复杂的话题，如果要讨论，可能要用占据这个文档的绝大部分。为了给你一个执行过程的概括，现在给出两个例子。

在上面的例子中，语句已经由 XML 或注解所定义。我们先来看一下 XML，以前，MyBatis 提供的的所有特性，都是基于 XML 的映射语句来实现。如果你以前使用过 MyBatis，那你对这些概念会非常熟悉。但是 XML 的映射配置文档有了许多改进，以后将会变得越来越简单清晰。下面这个基于 XML 映射语句可以完成上面的 SqlSession 调用。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="org.mybatis.example.BlogMapper">
  <select id="selectBlog" parameterType="int" resultType="Blog">
    select * from Blog where id = #{id}
  </select>
</mapper>
```

虽然这个简单的例子有点生涩，但是却非常简约。你可以定义多个文件，也可以在一个 XML 文件里定义任意个映射语句，这样可以省去 XML 标头。文件的其它部分，都是自身的描述。它定义了一个 org.mybatis.example.BlogMapper 命名空间，在这个空间里再定义了一个 selectBlog 语句。也可以使用 org.mybatis.example.BlogMapper.selectBlog 全名称来调用。我们可以将这样来调用上面这个文件

```
Blog blog = (Blog) session.select(
    "org.mybatis.example.BlogMapper.selectBlog", 101);
```

这和调用一个普通的 JAVA 类非常相似。这个名字可以直接映射为一个与命名空间相同名称的 Mapper 类，语句名对应类的方法名，参数和返回值也相对应。你可以用下列语句简单地针对 Mapper 接口进行调用，代码如下：

```
BlogMapper mapper = session.getMapper(BlogMapper.class);
Blog blog = mapper.selectBlog(101);
```

第二种方式有许多优点。一、它不依赖字符串，可以减少出错。二、如果你的 IDE 有代码自动完成功能，你可以很快导航到你的 SQL 语句（因为已经转化为方法名）。三、你不再需要设定返回值类型，因为接口限定了返回值和参数。

还有一个关于 Mapper 类的技巧。它们的映射语句完全不需要使用 XML 来配置，可以使用 JAVA 注解方式来取代。比如，上面的 XML 语句可以替换为：

```
package org.mybatis.example;
public interface BlogMapper {
  @Select("SELECT * FROM blog WHERE id = #{id}")
  Blog selectBlog(int id);
}
```

注解是非常简单明了的，但是 JAVA 注解既有局限性，在语句比较复杂的情况下又比较容易混乱。所以，如果你的语句比较复杂，最好还是使用 XML 来映射语句。

这主要取决于你和你的项目团队，决定哪个更适合于你，主要还是以稳健为主。也就是说，你不需要制约于哪一种方式，你可以很容易的把注解转为 XML，也可以把 XML 转化为注解。

五、作用域和生命周期

理解作用域和生命周期类非常重要，如果使用不当，会造成各种各样的问题。

1. SqlSessionFactoryBuilder

这个类可以被初始、使用和丢弃，如果你已经创建好了一个 SqlSessionFactory 后就不用再保留它。因此，SqlSessionFactoryBuilder 的最好作用域是方法体内，比如说定义一个方法变量。你可以重复使用 SqlSessionFactoryBuilder 生成多个 SqlSessionFactory 实例，但是最好不要强行保留，因为 XML 的解析资源要用来做其它更重要的事。

2. SqlSessionFactory

一旦创建，SqlSessionFactory 就会在整个应用过程中始终存在。所以没有理由去销毁和再创建它，一个应用运行中也不建议多次创建 SqlSessionFactory。如果真的那样做，会显得很拙劣。因此 SqlSessionFactory 最好的作用域是 Application。可以有多种方法实现。最简单的方法是单例模式或者是静态单例模式。然而这既不是广泛赞成和好用的。反而，使用 Google Guice 或 Spring 来进行依赖反射会更好。这些框架允许你生成管理器来管理 SqlSessionFactory 的单例生命周期。

3. SqlSession

每个线程都有自己的 SqlSession 实例，SqlSession 实例是不能被共享，也是不是线程安全的。因此最好使用 Request 作用域或者方法体作用域。不要使用类的静态变量来引用一个 SqlSession 实例，甚至不要使用类的一个实例变更来引用。永远不要在一个被管理域中引用 SqlSession，比如说在 Servlet 中的 HttpSession 中。如果你正在使用 WEB 框架，应该让 SqlSession 跟随 HTTP 请求的相似作用域。也就是说，在收到一个 HTTP 请求过后，打开 SqlSession，等返回一个回应以后，立马关掉这个 SqlSession。关闭 SqlSession 是非常重要的。你必须要确保 SqlSession 在 finally 方法体中正常关闭。可以使用下面的标准方式来关闭：

```
SqlSession session = sqlSessionFactory.openSession();
try {
  // do work
} finally {
  session.close();
}
```

使用这种模式来贯穿你的所有代码，以确保所有数据库资源都被完全关闭。[这是假定不是使用你自己的数据库连接，而是使用 MyBatis 来管理你的数据库连接资源]

4. Mapper 实例

Mapper 是一种你创建的用于绑定映射语句的接口。Mapper 接口的实例是用 SqlSession 来获得的。同样，从技术上来说，最广泛的 Mapper 实例作用域像 SqlSession 一样，使用请求作用域。确切地说，在方法被调用的时候调用 Mapper 实例，然后使用后，就自动销毁掉。不需要使用明确的注销。当一个请求执行正确无误的时候，像 SqlSession 一样，你可以轻而易举地操控这一切。保持简单性，保持 Mapper 在方法体作用域内。下面演示了如果来操作：

```
SqlSession session = sqlSessionFactory.openSession();
try {
  BlogMapper mapper = session.getMapper(BlogMapper.class);
  // do work
} finally {
  session.close();
}
```

第三章 Mapper 的 XML 配置文件

Mapper 的 XML 配置文件包含一些设置和属性，用于增强 MyBatis 的动作。文档的深层次结果如下：

configuration
- properties
- settings
- typeAliases
- typeHandlers
- objectFactory
- plugins
- environments
- environment
- transactionManager
- dataSource
- mappers

一、属性（properties）

JAVA 属性文件就可以配置直观的、可代替的属性，或者是属性项的子项。比如：

```
<properties resource="org/mybatis/example/config.properties">
  <property name="username" value="dev_user"/>
  <property name="password" value="F2Fa3!33TYyg"/>
</properties>
```

通过动态配置，这些属性都可以用替换整个文件的值。例如：

```
<dataSource type="POOLED">
  <property name="driver" value="${driver}"/>
  <property name="url" value="${url}"/>
  <property name="username" value="${username}"/>
  <property name="password" value="${password}"/>
</dataSource>
```

例子中的 usrname 和 password 将被属性文件中设置的值所替换，driver 和 value 属性也将被 config.properties 文件中值所替换，这为配置提供了多种选择。

属性值也可以设入到 SqlSessionBuilder.build()方法中，例如：

```
SqlSessionFactory factory =
sqlSessionFactoryBuilder.build(reader, props);
// ... or ...
SqlSessionFactory factory =
sqlSessionFactoryBuilder.build(reader, environment, props);
```

如果一个属性项在多个地方出现，那 MyBatis 将按以下顺序加载：

- 属性文件中的属性项首先被读取
- 在类路径或 URL 资源中读取的属性项第二顺序加载，并且可以覆盖第一顺序加载的值
- 在方法体中给定的参数值最好加载，但是以后覆盖上述两种加载的值。

也就是说，最高级别的属性值是方法体中设定的参数值，接下来是类路径和 URL，最后才是属性文件

二、设置（settings）

这是 MyBatis 修改操作运行过程细节的重要的步骤。下方这个表格描述了这些设置项、含义和默认值。

设置项	描述	允许值	默认值
cacheEnabled	对在此配置文件下的所有 cache 进行全局性开/关设置。	true | false	true
lazyLoadingEnabled	全局性设置懒加载。如果设为‘关’，则所有相关联的都会被初始化加载。	true | false	true
aggressiveLazyLoading	当设置为‘开’的时候，懒加载的对象可能被任何懒属性全部加载。否则，每个属性都按需加载。	true | false	true
multipleResultSetsEnabled	允许和不允许单条语句返回多个数据集（取决于驱动需求）	true | false	true
useColumnLabel	使用列标签代替列名称。不用的驱动器有不同的作法。参考一下驱动器文档，或者用这两个不同的选项进行测试一下。	true | false	true
useGeneratedKeys	允许 JDBC 生成主键。需要驱动器支持。如果设为了 true，这个设置将强制使用被生成的主键，有一些驱动器不兼容不过仍然可以执行。	true | false	False
autoMappingBehavior	指定 MyBatis 是否并且如何来自动映射数据表字段与对象的属性。PARTIAL 将只自动映射简单的，没有嵌套的结果。FULL 将自动映射所有复杂的结果。	NONE, PARTIAL, FULL	PARTIAL
defaultExecutorType	配置和设定执行器，SIMPLE 执行器执行其它语句。REUSE 执行器可能重复使用 prepared statements 语句，BATCH 执行器可以重复执行语句和批量更新。	SIMPLE REUSE BATCH	SIMPLE
defaultStatementTimeout	设置一个时限，以决定让驱动器等待数据库回应的多长时间为超时	正整数	Not Set (null)

下面列出关于设置的完整例子：

```
<settings>
  <setting name="cacheEnabled" value="true"/>
  <setting name="lazyLoadingEnabled" value="true"/>
  <setting name="multipleResultSetsEnabled" value="true"/>
```

```
    <setting name="useColumnLabel" value="true"/>
    <setting name="useGeneratedKeys" value="false"/>
    <setting name="enhancementEnabled" value="false"/>
    <setting name="defaultExecutorType" value="SIMPLE"/>
    <setting name="defaultStatementTimeout" value="25000"/>
  </settings>
```

三、类型别名（typeAliases）

类型别名是 Java 类型的简称。
它仅仅只是关联到 XML 配置，简写冗长的 JAVA 类名。例如：

```
  <typeAliases>
    <typeAlias alias="Author" type="domain.blog.Author"/>
    <typeAlias alias="Blog" type="domain.blog.Blog"/>
    <typeAlias alias="Comment" type="domain.blog.Comment"/>
    <typeAlias alias="Post" type="domain.blog.Post"/>
    <typeAlias alias="Section" type="domain.blog.Section"/>
    <typeAlias alias="Tag" type="domain.blog.Tag"/>
  </typeAliases>
```

使用这个配置，“Blog”就能在任何地方代替“domain.blog.Blog”被使用。

还有一些与通用 JAVA 类型建立的别名。它们是大小写敏感的，注意 JAVA 的基本类型，使用了_来命名。

别名	映射类型
_byte	byte
_long	long
_short	short
_int	int
_integer	int
_double	double
_float	float
_boolean	boolean
string	String
byte	Byte
long	Long
short	Short
int	Integer
integer	Integer
double	Double
float	Float
boolean	Boolean

date	Date
decimal	BigDecimal
bigdecimal	BigDecimal
object	Object
map	Map
hashmap	HashMap
list	List
arraylist	ArrayList
collection	Collection
iterator	Iterator

四、类型句柄（typeHandlers）

当 MyBatis 对 PreparedStatement 设入一个参数或者是从 ResultSet 返回一个值的时候，类型句柄被用将值转转为相匹配的 JAVA 类型。这方表格描述了默认的类型句柄。

类型句柄	Java 类型	JDBC 类型
BooleanTypeHandler	Boolean, boolean	任何与 BOOLEAN 兼容的类型
ByteTypeHandler	Byte, byte	任何与 NUMERIC or BYTE 兼容的类型
ShortTypeHandler	Short, short	任何与 NUMERIC or SHORT INTEGER 兼容的类型
IntegerTypeHandler	Integer, int	任何与 NUMERIC or INTEGER 兼容的类型
LongTypeHandler	Long, long	任何与 NUMERIC or LONG INTEGER 兼容的类型
FloatTypeHandler	Float, float	任何与 NUMERIC or FLOAT 兼容的类型
DoubleTypeHandler	Double, double	任何与 NUMERIC or DOUBLE 兼容的类型
BigDecimalTypeHandler	BigDecimal	任何与 NUMERIC or DECIMAL 兼容的类型
StringTypeHandler	String	CHAR, VARCHAR
ClobTypeHandler	String	CLOB, LONGVARCHAR
NStringTypeHandler	String	NVARCHAR, NCHAR
NClobTypeHandler	String	NCLOB
ByteArrayTypeHandler	byte[]	任何与字节流兼容的类型
BlobTypeHandler	byte[]	BLOB, LONGVARBINARY
DateTypeHandler	Date (java.util)	TIMESTAMP
DateOnlyTypeHandler	Date (java.util)	DATE
TimeOnlyTypeHandler	Date (java.util)	TIME
SqlTimestampTypeHandler	Timestamp (java.sql)	TIMESTAMP
SqlDateTypeHadler	Date (java.sql)	DATE
SqlTimeTypeHandler	Time (java.sql)	TIME
ObjectTypeHandler	Any	其它未指定的类型
EnumTypeHandler	Enumeration Type	VARCHAR - 任何与 string 兼容的类型。存储的是枚举编码，而不是枚举索引

你可以重写（loverride）类型句柄或者是创建你自己的方式来处理不支持或者是非标准的类型。只需要简单地实现 org.mybatis.type 包里的 TypeHandler，并且映射你的新类型句柄类到一个 JAVA 类型，再选定一个 JDBC 类型。例如：

```
// ExampleTypeHandler.java
public class ExampleTypeHandler implements TypeHandler {
  public void setParameter(
  PreparedStatement ps, int i, Object parameter,JdbcType jdbcType)
throws SQLException {
    ps.setString(i, (String) parameter);
  }
  public Object getResult(
ResultSet rs, String columnName)
throws SQLException {
    return rs.getString(columnName);
  }
  public Object getResult(
CallableStatement cs, int columnIndex)
throws SQLException {
    return cs.getString(columnIndex);
  }
}

// MapperConfig.xml
<typeHandlers>
<typeHandler javaType="String" jdbcType="VARCHAR"
handler="org.mybatis.example.ExampleTypeHandler"/>
</typeHandlers>
```

使用像这样的类型句柄，将会覆盖现有的处理 JAVA String 属性与 VARCHAR 和返回值的类型句柄。注意，MyBatis 无法省查数据库的元数据从而决定类型，所以你必须指定参数它是一个 VARCHAR 类型，并且结果映射到正确的类型句柄上。这么做主要是由于 MyBatis 在没有执行语句之类，无法得知数据的类型。

五、对象工厂（ObjectFactory）

每次 MyBatis 为结果对象创建一个新实例，都会用到 ObjectFactory。默认的 ObjectFactory 与使用目标类的构造函数创建一个实例毫无区别，如果有已经映射的参数，那也可能使用带参数的构造函数。如果你重写 ObjectFactory 的默认操作，你可以创建一下你自己的。比如：

```
// ExampleObjectFactory.java
public class ExampleObjectFactory extends DefaultObjectFactory {
  public Object create(Class type) {
    return super.create(type);
```

```
  }
  public Object create(
    Class type,
    List<Class> constructorArgTypes,
    List<Object> constructorArgs) {
        return super.create(type, constructorArgTypes, constructorArg
  }
  public void setProperties(Properties properties) {
    super.setProperties(properties);
  }
}

// MapperConfig.xml
<objectFactory type="org.mybatis.example.ExampleObjectFactory">
  <property name="someProperty" value="100"/>
</objectFactory>
```

ObjectFactory 接口非常简单，只包含两个方法，一个是构造函数，一个是带参数的构造函数。最后，setProperties 方法也可以使用 ObjectFactory 来配置。可以在 ObjectFactory 实例化后，通过 setProperties 方法，在对象工厂中定义属性。

六、插件（plugins）

MyBatis 允许你在映射语句执行过程中某点上拦截调用。默认的，MyBatis 允许插件拦截以下调用：

- Executor
  (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)
- ParameterHandler
  (getParameterObject, setParameters)
- ResultSetHandler
  (handleResultSets, handleOutputParameters)
- StatementHandler
  (prepare, parameterize, batch, update, query)

这些类的细节在每个方法签名中均可以找到，源代码在 MyBatis 每次发布时都可以下载。如果你要做的事不仅仅是调用，而是重写（overriding）方法，那你需要了解你要重写的方法的动作。如果你试图修改或者重写既定方便的动作，你最好深入到 MyBatis 的核心。因为这些方法和类都底层的架构，所以使用插件时要格外小心。

使用插件是非常简单而又有用的。只需要简单地实现这个 Interceptor 接口，确定好你要拦截的标识即可。

```
// ExamplePlugin.java
@Intercepts({@Signature(
type= Executor.class,
```

```
    method = "update",
    args = {MappedStatement.class,Object.class})})
    public class ExamplePlugin implements Interceptor {
      public Object intercept(Invocation invocation) throws Throwable {
        return invocation.proceed();
      }
      public Object plugin(Object target) {
        return Plugin.wrap(target, this);
      }
      public void setProperties(Properties properties) {
      }
    }

    // MapperConfig.xml
    <plugins>
      <plugin interceptor="org.mybatis.example.ExamplePlugin">
        <property name="someProperty" value="100"/>
      </plugin>
    </plugins>
```

上面的这个插件可以在执行器上拦截所有“update”方法的调用，这里的执行器，是一个映射语句内部对象的深架构的执行器。

重写 Configuration 类

附加说明一下，对于使用插件修改 MyBatis 的内核动作，你也可以重写整个 Configuration 类。简单地继承或者重写所有的方法，然后把它做为参数代入 sqlSessionFactoryBuilder.build(myConfig)中。再次强调，这会引起 MyBatis 动作的严重冲突，慎用。

七、环境（environments）

MyBatis 可以配置多个环境。这可以帮助你 SQL 映射对应多种数据库等。比如说，你想为开发、测试、发布产品配置不用的环境。或者，你想为多个数据库产品共享相同的模式，或者也想使用相同的 SQL 映射。等等。

需要记住一个重要的事情：虽然你可以配置多重环境，你也可以只选择一对一 SqlSessionFactory 实例。

所以如果你想连接两个数据库，你需要使用 SqlSessionFactory 创建两个实例，每个数据库一个。如果要连三个数据库就创建三个，以此类推。记住：

一个 SqlSessionFactory 实例对应一个数据库。

想要指定生成哪个环境，只要简单地把它做了一个可选参数代入 SqlSessionFactoryBuilder。下面两种方式都可以：

```
SqlSessionFactory  factory = sqlSessionFactoryBuilder.build(reader, environment);
SqlSessionFactory  factory = sqlSessionFactoryBuilder.build(reader, environment,properties);
```

如果环境变更省略了，就会载入默认的环境变量。像这样：

```
SqlSessionFactory  factory = sqlSessionFactoryBuilder.build(reader);
SqlSessionFactory  factory = sqlSessionFactoryBuilder.build(reader,properties);
```

环境元素定义这些环境是如何被配置的。

```
<environments default="development">
 <environment id="development">
    <transactionManager type="JDBC">
      <property name="" value=""/>
    </transactionManager>
    <dataSource type="POOLED">
      <property name="driver" value="${driver}"/>
      <property name="url" value="${url}"/>
      <property name="username" value="${username}"/>
      <property name="password" value="${password}"/>
    </dataSource>
  </environment>
 </environments>
```

注意这些关键段：

- 设定一个默认环境 ID
- 这个环境 ID 对每个环境都起作用
- 配置事务管理器
- 配置数据源

默认的环境和环境 ID 是对自己起作用，你可以随意起你想叫的名字，只是他们是不重复的就可以了。

事务管理器

MyBatis 有两个事情管理类型：

- **JDBC** - 这个类型直接全部使用 JDBC 的提交和回滚功能。它依靠使用连接的数据源来管理事务的作用域。
- **MANAGED** - 这个类型什么不做，它从不提交、回滚和关闭连接。而是让窗口来管理事务的全部生命周期。（比如说 Spring 或者 JAVAEE 服务器）

它们俩都不需要任何的属性。然而，既然它们是类型别名，你就直接把你的类名称或者类型别名指向你的 TransactionFactory 接口实现类就可以了。

```
public interface TransactionFactory {
  void setProperties(Properties props);
  Transaction newTransaction(Connection conn, boolean autoCommit);
}
```

实例化后，在 XML 中已经被配置的任何属性都可以代入到 setProperties()方法中。你自己的实现形式也需要创建一个事务实现，其实接口是非常简单的：

```
public interface Transaction {
  Connection getConnection();
  void commit() throws SQLException;
  void rollback() throws SQLException;
  void close() throws SQLException;
}
```

通过使用这两个接口，你自己完全可以定制 MyBatis 如何来处理事务。

数据源

数据源元素是用来配置使用 JDBC 数据源接口的 JDBC 连接对象的源。

大部分的 MyBatis 应用像上面例子中那样配置数据源。但是，这并不是必须的。需要清楚的是：只要使用了懒加载，才必须使用数据源。

数据源类型有三种：UNPOOLED，POOLED，JNDI。

UNPOOLED - 这个数据源实现只是在每次请求的时候简单的打开和关闭一个连接。虽然这有点慢，但作为一些不需要性能和立即响应的简单应用来说，不失为一种好选择。不同的数据库在性能方面也有所不同，所以相对于连接池来说倒是不重要，这个配置倒是蛮理想。UNPOOLED 数据源有几个属性：

- driver - 指定 JDBC 驱动器的 JAVA 类，而不是数据类。
- url - 连接数据库实例的 URL 路径
- username - 登录数据库的用户名
- password - 登录数据库的密码
- defaultTransactionsolationLevel - 指定连接的默认事务隔离层

另外，你也可以为数据驱动器设置属性。只需要简单取‘driver.’开头就行了，比如说：

- driver.encoding=UTF8

这就会把属性为‘encoding’，值为‘UTF-8’，通过 DriverManager.getConnection(url, driverProperties) 方法传递能数据库驱动器。

POOLED - 这个数据源缓存 JDBC 连接对象用于避免每次都要连接和生成连接实例而需要的验证时间。对于并发 WEB 应用，这种方式非常流行因为它有最快的响应时间。

在 UNPOOLED 的属于之上，POOLED 数据还有许多其它许多配置属性

- poolMaximumActiveConnections - 特定时间里可同时使用的连接数
- poolMaximumIdleConnections - 特定时间里闲置的连接数
- poolMaximumCheckoutTime - 在连接池强行返回前，一个连接可以进行‘检出’的总计时间
- poolTimeToWait - 这是一个底层的设置，给连接一个机会去打印 log 状态，并重新尝试重新连接，免得长时间的等待。
- poolPingQuery - Ping Query 是发送给数据库的 Ping 信息，测试数据库连接是否良好和是否准备好了接受请求。默认值是“NO PING QUERY SET”，让大部分数据库都不使用 Ping，返回一个友好的错误信息。
- poolPingEnabled - 设置 PingQuery 是否可用。如果可用，你可以使用一个最简单的 SQL 语句测试一下。默认是：false
- poolPingConnectionsNotUsedFor - 配置 poolPingQuery 多长时间可以用。通常匹配数据库连接的超时，避免无谓的 ping。默认：0，表示随时允许 ping，当然，必须在 poolPingEnabled 设为 true 的前提下。

JNDI - 这个数据源实现是为了准备和 Spring 或应用服务一起使用，可以在外部也可以在内部配置这个数据源，然后在 JNDI 上下文中引用它。这个数据源配置只需要两上属性：

- initial_context - 这个属性是被用于上下文从 InitialContext 中（比如：initialContext.lookup(initial_context)）查找。这个属性是可选的，如果被省略，InitialContext 将会直接查找 data_source 属性。
- data_source - 这是数据源实例能搜索到的上下文路径。它会直接查找 initial_context 搜索返回的值，如果 initial_context 没有值的庆，直接使用 InitialContext 查找。

像数据源其它配置一样，可以使用以‘env.’属性直接设给 InitialContext，例如：

- env.encoding=UTF8

这样就可以把值为‘UTF8’的属性直接代入 InitialContext 实例化的构造器。

八、映射器（Mappers）

现在 MyBatis 的动作已经通过以上的步骤配置好了。我们现在开始定义我们的映射 SQL 语句。首先，我们需要告诉 MyBatis 去哪儿寻找。JAVA 在自动发现上没有什么好的方法，所以最好的方式就是直接告诉 MyBatis 去哪发现映射文件。你可以使用类路径中的资源引用，或者使用字符，输入确切的 URL 引用. 例如：

```
// Using classpath relative resources
<mappers>
    <mapper resource="org/mybatis/builder/AuthorMapper.xml"/>
    <mapper resource="org/mybatis/builder/BlogMapper.xml"/>
    <mapper resource="org/mybatis/builder/PostMapper.xml"/>
</mappers>

// Using url fully qualified paths
<mappers>
    <mapper url="file:///var/sqlmaps/AuthorMapper.xml"/>
    <mapper url="file:///var/sqlmaps/BlogMapper.xml"/>
    <mapper url="file:///var/sqlmaps/PostMapper.xml"/>
</mappers>
```

这些句子仅仅只是告诉 MyBatis 往哪去寻找，具体的关于每个 SQL 映射文件的细节，在下一章中详细讨论。

第四章 SQL 映射语句文件

MyBatis 真正强大的地主就是在映射语句，这也是魔力所在。就所有功能来说，SQL 映射 XML 文件相对来说比较简单。当然，如果你比较了文件与 JDBC 代码，你就会发现，映射 XML 文件节省了 95%的代码。MyBatis 为 SQL 而建，但却又给你极大的空间。

SQL 映射 XML 文件仅仅只有一些初级的元素：

- cache – 配置给定模式的缓存
- cache-ref – 从别的模式中引用一个缓存
- resultMap – 这是最复杂而却强大的一个元素了，它描述如何从结果集中加载对象
- sql – 一个可以被其他语句复用的 SQL 块
- insert – 映射 INSERT 语句
- update – 映射 UPDATE 语句
- delete – 映射 DELEETE 语句
- select – 映射 SELECT 语句

下一将将讲述每个元素的细节

一、select

select 语句可能是 MyBatis 中使用最多的元素了。通常都会把数据存储在数据库中，然后读取。所以大部分的应用中查询远远多于修改。对于每个过 insert、update 或者 delete，都会伴有众多 select。这是 MyBatis 基础原则之一，也是为什么要把更多的焦点和努力都放在查询和结果映射上。一个 select 元素非常简单。例如：

```
<select id=" selectPerson" parameterType=" int" resultType=" hashmap" >
  SELECT * FROM PERSON WHERE ID = #{id}
</select>
```

这条语句就叫做‘selectPerson’，有一个 int（或是 Integer），并返回一个以数据表列名和 key 的 HashMap 结果集。

注意参数的标识是：

```
#{id}
```

这告诉 MyBatis 生成一个 PreparedStatement 的参数。对于 JDBC，这样的参数可能是在 SQL 中用‘？’标识，传递给一个 PreparedStatement 语句。通常像这样：

```
// Similar JDBC code, NOT MyBatis…
String selectPerson = “SELECT * FROM PERSON WHERE ID=?”;
PreparedStatement ps = conn.prepareStatement(selectPerson);
ps.setInt(1,id);
```

当然，如果只用 JDBC 单独去解开这个结果集并映射到对象上的话，则需要非常多的代码，而这些，MyBatis 都已经为你做到了。关于参数和结果集映射，还有许多要学的，以后有专门的章节来讨论。

select 语句有众多的属性可以供你来配置语句的执行细节。

```
<select
```

```
id=”selectPerson”
parameterType=”int”
resultType=”hashmap”
resultMap=”personResultMap”
flushCache=”false”
useCache=”true”
timeout=”10000”
fetchSize=”256”
statementType=”PREPARED”
resultSetType=”FORWARD_ONLY”
>
```

属性	描述
id	在这个模式下唯一的标识符，可被其它语句引用
parameterType	传给此语句的参数的完整类名或别名
resultType	语句返回值类型的整类名或别名。注意，如果是集体，那么这里填写的是集合的项的整类名或别名，而不是集合本身的类名。resultType 与 resultMap 不能并用
resultMap	引用的外部 resultMap 名。结果集映射是 MyBatis 中最强大的特性，也非常好理解。许多复杂的映射都可以轻松解决。resultType 与 resultMap 不能并用
flushCache	如果设为 true，则会在每次语句调用的时候就会清空缓存。select 语句默认设为 false
useCache	如果设为 true，则语句的结果集将被缓存，select 语句默认设为 false
timeout	设置驱动器在抛出异常前等待回应的最长时间，默认为不值，由驱动器自己决定
fetchSize	设置一个值后，驱动器会在结果集数目达到此数值后，激发返回，默认为不值，由驱动器自己决定
statementType	STATEMENT，PREPARED 或 CALLABLE 中的任意一个，这就告诉 MyBatis 分别使用 Statement，PreparedStatement 或者 CallableStatement。默认：PREPARED
resultSetType	FORWARD_ONLY 、SCROLL_SENSITIVE 、SCROLL_INSENSITIVE 三个中的任意一个。默认是不设置，由驱动器决定

二、insert, update, delete

数据修改语句 insert，update，delete 都非常相似：

```
<insert
    id="insertAuthor"
    parameterType="domain.blog.Author"
    flushCache="true"
```

```
        statementType="PREPARED"
        keyProperty=""
        useGeneratedKeys=""
        timeout="20000">

    <update
        id="insertAuthor"
        parameterType="domain.blog.Author"
        flushCache="true"
        statementType="PREPARED"
        timeout="20000">

    <delete
        id="insertAuthor"
        parameterType="domain.blog.Author"
        flushCache="true"
        statementType="PREPARED"
        timeout="20000">
```

属性	描述
id	在这个模式下唯一的标识符，可被其它语句引用
parameterType	传给此语句的参数的完整类名或别名
flushCache	如果设为 true，则会在每次语句调用的时候就会清空缓存。select 语句默认设为 false
useCache	如果设为 true，则语句的结果集将被缓存，select 语句默认设为 false
timeout	设置驱动器在抛出异常前等待回应的最长时间，默认为不值，由驱动器自己决定
fetchSize	设置一个值后，驱动器会在结果集数目达到此数值后，激发返回，默认为不值，由驱动器自己决定
statementType	STATEMENT，PREPARED 或 CALLABLE 中的任意一个，这就告诉 MyBatis 分别使用 Statement，PreparedStatement 或者 CallableStatement。默认：PREPARED
useGeneratedKeys	（仅限 insert）告诉 MyBatis 使用 JDBC 的 getGeneratedKeys 方法来获取数据库自己生成的主键（MySQL、SQLSERVER 等关系型数据库会有自动生成的字段）。默认：false
keyProperty	（仅限 insert）标识一个将要被 MyBatis 设置进 getGeneratedKeys 的 key 所返回的值，或者为 insert 语句使用一个 selectKey 子元素。

下面是一些 insert，update，delete 语句的示例：

```
<insert id="insertAuthor" parameterType="domain.blog.Author">
    insert into Author (id,username,password,email,bio)
    values (#{id},#{username},#{password},#{email},#{bio})
```

```
</insert>

<update id="updateAuthor" parameterType="domain.blog.Author">
    update Author set
    username = #{username},
    password = #{password},
    email = #{email},
    bio = #{bio}
    where id = #{id}
</update>

<delete id="deleteAuthor" parameterType="int">
    delete from Author where id = #{id}
</delete>
```

关于 insert 需要多说几句，它会有一些更多的属性有子元素，用来可以使用多种方式生成主键处理。

首先，如果你使用的数据库支持自动生成主键，那么你就可以设置 useGeneratedKeys=”true”，然后把 keyProperty 设成对应的列，就搞定了。比如说上面的 Author 使用 auto-generated 为 id 列生成主键，语句可以修改如下：

```
<insert id="insertAuthor" parameterType="domain.blog.Author"
useGeneratedKeys=”true” keyProperty=”id”>
    insert into Author (username,password,email,bio)
    values (#{username},#{password},#{email},#{bio})
</insert>
```

MyBatis 还有另外一种方式来处理在数据库不支持自动生成主键的情况来生成主键，或者是为 JDBC 驱动不能自动生成主键的情况下生成主键。

下面简单的示例一下如果生成随机的 ID，也许你不需要这么做，仅仅只是演示一下 MyBatis 的复杂功能。

```
<insert id="insertAuthor" parameterType="domain.blog.Author">
  <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>
```

在上面例子中，selectKey 语句将会首先运行，Author 表的 id 列将会被设置，然后再调用这个 insert 语句，这相当于自动在你的数据库里生成主键而不需要写复杂的代码。

selectKey 元素描述如下：

```
<selectKey
```

```
keyProperty="id"
resultType="int"
order="BEFORE"
statementType="PREPARED">
```

属性	描述
keyProperty	selectKey 语句生成结果需要设给的目的列
resultType	结果类型，MyBatis 通常可以自己检测到，但这并不影响给它一个确实的设定。MyBatis 允许使用基本的数据类型，也包括 String 类型。
order	可以设成 BEFORE 或者 AFTER，如果设为 BEFORE，那它会先选择主键，然后设置 keyProperty，再执行 insert 语句；如果设为 AFTER，它就先运行 insert 语句再运行 selectKey 语句，通常是 insert 语句中内部调用数据库（像 Oracle）内嵌的序列机制。
statementType	像上面的那样，MyBatis 支持 STATEMENT，PREPARED 和 CALLABLE 的语句形式，对应 Statement，PreparedStatement 和 CallableStatement 响应

三、SQL

这个元素用来定义一个可以复用的 SQL 语句段，供其它语句调用。比如：

```
<sql id=”userColumns”> id,username,password </sql>
```

这个语句块，可以包含到别的语句中，比如：

```
<select id=”selectUsers” parameterType=”int” resultType=”hashmap”>
select <include refid=”userColumns” />
from some_table
where id = #{id}
</select>
```

四、参数（parameters）

从以上所以的语句中，我们都可以看到参数。参数在 MyBatis 中非常强大。90%的情况下，都会用到。比如：

```
<select id=”selectUsers” parameterType=”int” resultType=”User”>
select id, username, password
from users
where id = #{id}
</select>
```

这个示例显示了非常简单的命名参数映射。parameterType 设置为‘int’，因此参数可以被叫作任何值。基本的或者是简单的数据类型像 Integer 和 String 没有相关的属性，因此将为使用全部的参数值。然而，如果你代入一个复合的对象，情况会有所不同。比如

```
<insert id=”insertUser” parameterType=”User” >
insert into users (id, username, password)
values (#{id}, #{username}, #{password})
</insert>
```

如果是代入一个 User 类型做为语句的参数，那么这个 id,username 和 password 属性将于找到它对应的值，代入到 PreparedStatement 参数。

把参数代入语句是非常简单的，但是参数映射还有其它许多特性。

首先，像 MyBatis 其它部分一样，参数可以指定更多的数据类型。

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

像 MyBatis 的其它部分一样，这个 javaType 是由参数对象决定，队了 HashMap 以外。然后这个 javaType 被指定给正在使用的 TypeHandler。

注意：如果传递了一个空值，那这个 JDBC Type 对于所有 JDBC 允许为空的列来说是必须的。你可能研究一下 PreparedStatement.setNull()的 JavaDocs 文档。

对于更深的自定制类型处理，你也可以指定特殊的 TypeHandler 类或者别名。比如：

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

当然，这看起来有点复杂了，不过，这种情况比较少见。

对于数字类型，你可以使用 numericScale 来决定相关的小数位有多长。

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

最后，这个 mode 属性允许你设定 IN，OUT 或者 INOUT 参数，如果参数是 OUT 或者 INOUT，那实际的参数值可能会有变动，就你正在调用一个输出参数。如果 mode=OUT 或者 mode=INOUT，并且 jdbcType=CURSOR，你需要指定一个 resultMap 映射结果集给这个参数类型。注意这里的 javaType 类型是可选的，如果为空而 jdbcType 是 CURSOR 的话，则会自动地设给 ResultSet。

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

MyBatis 同样支持更多数据类型，比如 STRUCT 类型，但是当你使用 OUT 模式的时候，你必须告诉语句类型的名称。

```
#{middleInitial,
mode=OUT,
jdbcType=STRUCT,
jdbcTypeName=MY_TYPE,
resultMap=departmentResultMap}
```

尽管有这么多强大的选项，但是大部分的情况，你只选择使用简单的属性名称，MyBatis 会识别其它部分。顶多，在对应允许为空的列里指定一下 jdbcType 就可以了。

```
#{firstName}
```

```
#{middleInitial,jdbcType=VARCHAR}
#{lastName}
```

字符串代入法

默认的情况下，使用 #{}语法会促使 MyBatis 生成 PreparedStatement 属性并且使用 PreparedStatement 的参数（=？）来安全的设置值。尽量这些是快捷安全，也是经常使用的。但有时候你可能想直接未更改的字符串代入到 SQL 语句中。比如说，对于 ORDER BY，你可能会这样使用：

```
ORDER BY ${columnName}
```

但 MyBatis 不会修改和规避掉这个字符串。

注意：这样地接收和应用一个用户输入到未更改的语句中，是非常不安全的。这会让用户能植入破坏代码，所以，要么在这些字段你不要允许客户输入，要么你直接来检测和规避一下。

五、resultMap

resultMap 是 MyBatis 中最重要最强大的元素了。你可以让你比使用 JDBC 调用结果集省掉 90%的代码，也可以让你做许多 JDBC 不支持的事。现实上，要写一个等同类似于交互的映射这样的复杂语句，可能要上千行的代码。ResultMaps 的目的，就是这样简单的语句而不需要多余的结果映射，更多复杂的语句，除了只要一些绝对必须的语句描述关系以外，再也不需要其它的。

你可能已经看到这样的简单映射语句，而没有一个 resultMap，例如：

```
<select id=”selectUsers” parameterType=”int” resultType=”hashmap”>
select id, username, hashedPassword
from some_table
where id = #{id}
</sql>
```

像这样的语句简单的所有列结果将会自动地映射到以列表为 key 的 HasMap，使用 resultType 指定。虽然这对许多场合下有用，但是 HashMap 却不是非常好的领域建模。更多可能你的应用会使用 JavaBeans 或者 POJOs 来领域建模。MyBatis 对这两者都支持。

```
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 JavaBeans 规范，上面的类有三个属性：id, username 和 hashedPassword。它们对应着 select 语句的列名。

这样的 JavaBean 可以像 HashMap 一样简单地映射到 ResultSet。

```
<select id=”selectUsers” parameterType=”int”
resultType=”com.someapp.model.User”>
select id, username, hashedPassword
from some_table
where id = #{id}
</sql>
```

你当 TypeAliases 当成你朋友一样牢记。使用它，就可以避免敲长长的类的全路径名。比如说：

```
<!-- In Config XML file -->
<typeAlias type=”com.someapp.model.User” alias=”User”/>

<!-- In SQL Mapping XML file -->
<select id=”selectUsers” parameterType=”int”
resultType=”User”>
select id, username, hashedPassword
from some_table
where id = #{id}
</sql>
```

在这些情况下 MyBatis 自动在后台生成一个 ResultMap 映射列名到 JavaBean 的属性，如果列的名称与属性名确实不符，可以使用标准 SQL 的特性，生成别名，使用列名与属性匹配。例如：

```
<select id=”selectUsers” parameterType=”int” resultType=”User”>
select
user_id    as “id”,
user_name    as “userName”,
hashed_password  as “hashedPassword”
from some_table
where id = #{id}
</select>
```

ResultMaps 的许多优点可能你已经学到了许多，但还有一个你没有见过。上面的例子已经囊括了不省的

功能。做为一个例子的情况，我们来看一下最后一个例子，看起来像 resultMap 扩展功能，是另外一种解决列表不配对的情况。

```
<resultMap id="userResultMap" type="User">
    <id property="id" column="user_id" />
    <result property="username" column="username"/>
    <result property="password" column="password"/>
</resultMap>
```

这个语句将会被 resultMap 来引用。注意，不能再使用 resultType 属性。

```
<select id=”selectUsers” parameterType=”int” resultMap=”userResultMap”>
select user_id, user_name, hashed_password
from some_table
where id =#{id}
</select>
```

一切就是这么的简单。

高级结果映射

MyBatis 谨记一个信条：数据库并不是你想怎么就怎么的。我们也许总是希望总是只有一条简单的数据库映射就能完美的解决应用中的问题，但是它们不是。结果映射就是 MyBatis 为解决这些问题提供的答案。

举个例子，下面语句怎么映射？

```
<!-- Very Complex Statement -->
<select id="selectBlogDetails" parameterType="int"
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>
```

你可能想要把它映射到一个智能的对象模型，包括由一个作者写的一个博客，由多项交互，由 0 个或者多个评论和标签。下面是一个复杂 ResultMap 的完整例子，（假定作者、博客、评论和标签都是别名）。仔细看看这个例子，但是不用太担心，我们会一步步地来。跃然乍看让人沮丧，但是实际上是很简单的。

```
<!-- 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" column="blog_author_id" 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" column="post_author_id"
javaType="Author"/>
    <collection property="comments" column="post_id" ofType=" Comment">
      <id property="id" column="comment_id"/>
    </collection>
    <collection property="tags" column="post_id" ofType=" Tag" >
      <id property="id" column="tag_id"/>
```

```
    </collection>
    <discriminator javaType="int" column="draft">
      <case value="1" resultType="DraftPost"/>
    </discriminator>
  </collection>
</resultMap>
```

这个 resultMap 的元素的子元素比较多，讨论起来比较宽泛。下面我们从概念上概览一下这个 resultMap 的元素。

resultMap

- constructor - 用来将结果反射给一个实例化好的类的构造器
  - idArg - ID 参数；将结果集标记为 ID，以方便全局调用
  - arg - 反射到构造器的通常结果
- id - ID 结果，将结果集标记为 ID，以方便全局调用
- result - 反射到 JavaBean 属性的普通结果
- association - 复杂类型的结合；多个结果合成的类型
  - nested result mappings - 几 resultMap 自身嵌套关联，也可以引用到一个其它上
- collection - 复杂类型集合 a collection of complex types
  - nested result mappings - resultMap 的集合，也可以引用到一个其它上
- discriminator - 使用一个结果值以决定使用哪个 resultMap
  - case - 基本一些值的结果映射的 case 情形
    - nested result mappings - 一个 case 情形本身就是一个结果映射，因此也可以包括一些相同的元素，也可以引用一个外部 resultMap。

最佳实践：慢慢地来生成 resultMap，单元测试非常有用，如果你试用一下子就生成像上面这样巨大的 resultMap。可能你会出错，并且工作起来非常吃力。从简单地开始，再一步步地扩展，并且单元测试。使用框架开发，有点像在一个黑箱里面。如果要确定是否达到你所需要的行为，最好的方式就是写单元测试。这对提交 bugs 也非常有用。

下一节，我们一步步地查看这些细节。

id, result

```
<id property="id" column="post_id"/>
<result property="subject" column="post_subject"/>
```

这是结果映射最基础的部分，id，和映射到单列值到一个属性或字段的简单数据类型（String，int，double，Date，等等）

唯一的不同，是 id 是做为唯一的标识，当和其它对象实例对比的时候。这可以用平常地调用非常有用，尤其是应用到缓存和内嵌的结果映射

属于如下表：

属性	描述
property	这个属于映射一个结果的列。如果 JavaBean 的属性与给定的名称匹配，就

	会使用匹配的名字。否则，MyBatis 将搜索属性名。两种情况下你都可以使用逗号的属性形式。比如，你可以遇到到‘username’，也可以映射到‘address.street.number’
column	数据表的列名或者标签别名，通用是代入 resultSet.getString(columnName)这个名称。
javaType	一个完整的类名，或者是一个类型别名。如果你匹配的是一个 JavaBean，那 MyBatis 通常会自行检测到。然后，如果你是要映射到一个 HashMap，那你需要指定 javaType 明天要达到的目的。
jdbcType	数据表支持的类型列表。这个属性只在 insert,update 或 delete 的时候针对允许空的列有用。JDBC 需要这项，但 MyBatis 不需要。如果你是直接针对 JDBC 编码，且有允许空的列，而你要指定这项。
typeHandler	我们已经讨论过了这项。使用这个属性可以覆写类型处理器。这项值可以是一个完整的类名，也可以是一个类型别名。

支持的 JDBC 类型

为了将来的引用，MyBatis 支持下列 JDBC 类型，通过 JdbcType 枚举。

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	

构造器（constructor）

```
<constructor>
<idArg column="id" javaType="int"/>
<arg column=”username” javaType=”String”/>
</constructor>
```

当属性与 DTO，和你自己的领域模型操作的时候，许多场合要用到不变类。通常，数据表包含引用和查找数据很少甚至不会与不变类相匹配。构造器反射允许你给实例化后的类设置值，不用通过 public 方法。MyBatis 同样也支持 private 属性和 JavaBeans 的私有属性达到这一点，但是有一些用户可能更喜欢使用构造器反射。构造器元素可以做到这点。

考虑下面的构造器。

```
public class User {
    //…
    public User(int id, String username) {
        //…
    }
    //…
}
```

为了将值注入到构造器中，MyBatis 需要使用它的参数的类型来标识构造器。Java 没有办法通过参数名

称反射。所以当创建一个构造器元素时，要确定参数是否良好，类型是否指定。

```
<constructor>
<idArg column="id" javaType="int"/>
<arg column=”username” javaType=”String”/>
</constructor>
```

其它的属于和规则与 id 和 result 元素一模一样。

联合（association）

```
<association property="author" column="blog_author_id" javaType="
Author">
<id property="id" column="author_id"/>
<result property="username" column="author_username"/>
</association>
```

联合元素用来处理“一对一”的关系。比如说，在我们的例子中，一个博客对应一个作者。联系映射像其它结果一样的工作。你指定目标属性，要返回值的列，属性的 javaType（通常 MyBatis 自己会识别），如果需要设定一下 jdbcType，如果你想覆写的话返回结果的值，需要指定 typeHandler。

不同的地方是你需要告诉 MyBatis 如果加载一个联合。MyBatis 可以用两种方式加载：

- Nested Select: 执行一个其它映射的 SQL 语句返回一个期望的复杂类型
- Nested Results: 使用一个嵌套的结果映射来处理交合的结果的重复的子集

首先，让我们来查看一下元素的属性。如你所见，它不同于普通只有 select 和 resultMap 的结果集映射属性。

属性	描述
property	这个属于映射一个结果的列。如果 JavaBean 的属性与给定的名称匹配，就会使用匹配的名字。否则，MyBatis 将搜索属性名。两种情况下你都可以使用逗号的属性形式。比如，你可以遇到到‘username’，也可以映射到‘address.street.number’
column	数据表的列名或者标签别名，通用是代入 resultSet.getString(columnName)这个名称。**注意**：在处理组合键时，你可以使用 column=”{prop1=col1,prop2=col2}”这样的语法，设置多个列名传入到嵌套语句。
javaType	一个完整的类名，或者是一个类型别名。如果你匹配的是一个 JavaBean，那 MyBatis 通常会自行检测到。然后，如果你是要映射到一个 HashMap，那你需要指定 javaType 明天要达到的目的。
jdbcType	数据表支持的类型列表。这个属性只在 insert，update 或 delete 的时候针对允许空的列有用。JDBC 需要这项，但 MyBatis 不需要。如果你是直接针对 JDBC 编码，且有允许空的列，而你要指定这项。
typeHandler	我们已经讨论过了这项。使用这个属性可以覆写类型处理器。这项值可以是一个完整的类名，也可以是一个类型别名。

联合的嵌套选择

select	要被用来加载复杂类型的其它映射语句的 ID。从指定列属性中返回的值，将作为参数设置给目标 select 语句。表格下方将有一个例子。

	注意：在处理组合键时，你可以使用 column=”{prop1=col1,prop2=col2}”这样的语法，设置多个列名传入到嵌套语句。

示例：

```
<resultMap id=”blogResult” type=”Blog”>
  <association property="author" column="blog_author_id"
javaType="Author"
select=”selectAuthor” />
</resultMap>

<select id=”selectBlog” parameterType=”int” resultMap=”blogResult”>
  SELECT * FROM BLOG WHERE ID = #{id}
</select>

<select id=”selectAuthor” parameterType=”int” resultType="Author">
  SELECT * FROM AUTHOR WHERE ID = #{id}
</select>
```

就是这个样子，我们有两个 select 语句，一个用为加载博客，另外一个用来加载作者。博客这一项的映射结果局申明一个“selectAuthor”语句，来来加载它的作者属性。

如果它们的列名和属性名称相匹配的话，所有其它的属性都会自动加载。

这一系列都是比较简单，但是对于大数据集或列表，可能有点问题。这就是“N+1 Selects Problem”。概要地说，N+1 Selects Problem 会这样产生：

- 你执行一个单条语句去获取一个列表记录。(“+1”)
- 对每一条记录，再去执行一个 select 语句去加载每一条记录的详细（“N”）

这个问题会在成千上万条语句在发生，是无法预料的。

就上面的部分，MyBatis 可以使用懒加载这些查询，因而你可以立马节俭开销。然而发，如果你加载一个列表然后立马又迭代访问内嵌的数据，你再调用所以的懒加载，那么执行就非常糟糕了。

鉴于此，有另一个方式。

联合的嵌套结果集

resultMap	一个可以映射联合嵌套结果集到一个适合对象图元的 ResultMap 的 ID。这是一个替换的方式去调用另一个 select 语句。它允许你去结合多个表在一起到一个结果集里。因此这个结果集中包括多个可复杂的、重复的数据组，用来分解和映射属性到一个嵌套的对象图元。简言之，MyBatis 让你把结果映射‘链’到一起，用来处理嵌套结果。举个例子更为理解，例子在表格下方。

你已经在上面看到了一上很复杂的嵌套联合语句。下面这个例子就非常简单了，它演示了如何工作的。我们不执行分离的语句，而是把博客和作者表都连接在一起。像这样：

```
<select id="selectBlog" parameterType="int" 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>
```

注意这个连接，注意结果都被别名为一个唯一且明确的名称。这将使映射变得更多容易。我们可以这样做：

```
<resultMap id="blogResult" type="Blog">
  <id property=”blog_id” column="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>
```

从上面的例子中你可以看到博客的作者，减除到了“authorResult”的结果映射中，用来加载作者实例。

非常重要：id 元素是嵌套映射中非常重要的角色。你需要指定一个或更多的属性用来标记结果。如果你没有这么做，MyBatis 也可能会正常运行，但是却会付出惨重的性能损耗。尽量用数量少的列来唯一标识结果，使用主键是个好的选择。

上面的例子使用一个扩展的 resultMap 元素来联合映射。这可使作者结果映射可重复使用。然后，如果你不需要重用它，或者如果只是简单地想协同定位你的结果映射到一个描述性的结果映射中，你可以嵌套这个联合结果映射。下面例子就是使用这样的方式：

```
<resultMap id="blogResult" type="Blog">
  <id property=”blog_id” column="id" />
  <result property="title" column="blog_title"/>
```

```
  <association property="author" column="blog_author_id"
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>
```

你已经看到了如果处理“一对一”类型的联合，那么“一对多”怎么办呢？下一节我们就来讨论。

聚集（collection）

```
<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>
```

聚集元素作用和联合几乎一模一样，现实上，他们非常相似，阐述它们之间的相同没有什么意义，让我们来着重看一下他们的不同。

继续我们上面的例子，一个博客只有一个作者，但是一个博客却有许多的评论。在一个博客类里，可能会表述成这样：

```
private List<Post> posts;
```

要映射一个嵌套的结果集到一个像这样的列表中，我们可以使用聚集元素。就像联合元素一样，我们可以使用一个嵌套选择，或者是一个连接的嵌套的结果。

集合的嵌套选择

首先，让我来看一下嵌套选择来加载博客的评论。

```
<resultMap id=”blogResult” type=”Blog”>
  <collection property="posts" javaType=”ArrayList” column="blog_id"
ofType="Post" select=”selectPostsForBlog” />
</resultMap>

<select id=”selectBlog” parameterType=”int” resultMap=”blogResult”>
  SELECT * FROM BLOG WHERE ID = #{id}
</select>

<select id=”selectPostsForBlog” parameterType=”int” resultType="Author">
  SELECT * FROM POST WHERE BLOG_ID = #{id}
</select>
```

有许多事情你都需要注意，但是大部分我们都在上面的聚合元素中见过。首先，你一定要搞明白，我们正在使用的是聚集元素。你会注意到有一个“***ofType***”新元素。这个元素是用来区别 JavaBean（或者字

段）的属性类型和集合所包括的类型。所以，当你读到这段的时候：

```
<collection property="posts" javaType=”ArrayList” column="blog_id"
ofType="Post" select=”selectPostsForBlog” />
```

你应该理解为：“一组名为 posts 的 ArrayList 集合，它的类型是 Post。”

javaType 属于完全可以省略，MyBatis 会为你自动识别，所以你通常可以省略这样简写：

```
<collection property="posts" column="blog_id" ofType="Post" select=”
selectPostsForBlog” />
```

集合的嵌套结果

到这里，你可能已经猜出来嵌套结果集到一个集合中是如果工作的。因为它和联合是一模一样的，只是多了一个“***ofType***”，上面已经提到了它的作用。

首先，看一下这个 SQL：

```
<select id="selectBlog" parameterType="int" 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>
```

我们已经连接了博客和评论表，并且让列名标签也做了明确的处理，以方便映射。现在像这样映射一个博客和它的一组评论：

```
<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>
```

再次强调，这个 id 属于非常非常重要!!

同样，如果你更使用稍长一点的代码，而达到可重复使用。你可以使用这面这种方式：

```
<resultMap id="blogResult" type="Blog">
  <id property=”id” column="blog_id" />
  <result property="title" column="blog_title"/>
  <collection property="posts" ofType="Post" resultMap=”blogPostResult”
/>
```

```
</resultMap>

<resultMap id="blogPostResult" type="Post">
    <id property="id" column="post_id"/>
    <result property="subject" column="post_subject"/>
    <result property="body" column="post_body"/>
</resultMap>
```

注意：这里没有深度、宽度、联合和聚合数目的限制。但是一定要把性能牢记在心。单元测试和性能测试能帮助你调整你采取的方式。幸好，MyBatis 能允许你在以后还能修改你的想法，只需要修改一些少量的代码。

关于高级联合和集合映射是一个高深的课题，文档只能是帮你了解到这，多做一些实践，你就会很快理解透彻了。

识别器（discriminator）

```
<discriminator javaType="int" column="draft">
  <case value="1" resultType="DraftPost"/>
</discriminator>
```

有时候一条数据库查询可能会返回的结果可能可能包括各种不同的数据类型。识别器元素就是被用来处理这种事情的，还包括一些继承层次。识别器非常容易理解，它很像 JAVA 里的 switch 语句。
一个识别器定义指定列和 javaType 属性。列就是 MyBatis 将要取出进行比较的值。javaType 用来确定适当的测试是否运行正确，虽然 String 在大部分情况下都可以使用。示例：

```
<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>
```

这上示例中，MyBatis 将会从数据集中获取每条记录，并比较 vehicle type 的值。如果它匹配了识别器的条件，就会使用相对应的 resultMap。这种行为有排他性，也就是说，只要匹配到了一项，剩余的部分都不忽略不再进行比较。使用扩展属性除外，我们马上将会谈到。如果没有任何一项能匹配到，MyBatis 就会简单地使用识别器外面定义的 resultMap。因此如果像下面这样定义一个 carResult。

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

```
        <result property=”doorCount” column="door_count" />
    </resultMap>
```

那么仅仅只是加载了 doorCount 这个属性。这样做是为了完全独立的聚集识别器的选项，哪怕它与上一层的 resultMap 一点关系都没有。在刚才的例子里我们当然之知道 cars 和 vehicles 的关系。因此，我们也要把其它部分加载进来。我们要稍稍改动一下 resultMap。

```
    <resultMap id="carResult" type="Car" extends=”vehicleResult”>
        <result property=”doorCount” column="door_count" />
    </resultMap>
```

现在，vehicleResult 和 carResult 都将被加载。

可能有人会认为这样的扩展映射有一点单调了，所以还有另外一种可选的语法来更简明映射。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>
```

记住：对于这么多的结果映射，如果你不指定任何的结果，那么 MyBatis 会自动地将列名与属性相匹配。所以上面所举的例子比实际中需要的要详细。也就是说，大部分数据库有点复杂，并且它并不是所有情况都是完全可以依赖的。

六、缓存（cache）

MyBatis 包含一个强在的、可配置、可定制的缓存机制。MyBatis 3 的缓存实现有了许多改进，既强劲也更容易配置。

默认的情况，缓存是没有开启，除了会话缓存以外，它可以提高性能，且能解决全局依赖。开启二级缓存，你只需要在 SQL 映射文件中加入简单的一行：

```
<cache/>
```

这句简单的语句的作用如下：

- 所有在映射文件里的 select 语句都将被缓存。
- 所有在映射文件里 insert,update 和 delete 语句会清空缓存。
- 缓存使用“最近很少使用”算法来回收
- 缓存不会被设定的时间所清空。
- 每个缓存可以存储 1024 个列表或对象的引用（不管查询出来的结果是什么）。
- 缓存将作为“读/写”缓存，意味着获取的对象不是共享的且对调用者是安全的。不会有其它的调用者或线程潜在修改。

缓存元素的所有特性都可以通过属性来修改。比如：

```
<cache
eviction="FIFO"
flushInterval="60000"
size="512"
readOnly="true"/>
```

更多高级的配置创建一个 FIFO 缓存让 60 秒就清空一次，存储 512 个对象结果或列表引用，并且返回的结果是只读。因此在不用的线程里的两个调用者修改它们可能会引用冲突。

收到的方针如下：

- LRU - 最近最少使用法：移出最近较长周期内都没有被使用的对象。
- FIFI- 先进先出：移出队列里较早的对象
- SOFT - 软引用：基于软引用规则，使用垃圾回收机制来移出对象
- WEAK - 弱引用：基于弱引用规则，使用垃圾回收机制来强制性地移出对象
- 默认值是 LRU。

flushInterval 属性可以被设置为一个正整数，代表一个合理的毫秒总计时间。默认是不设置，因此使用无间隔清空即只能调用语句来清空。

size 属性可以设置为一个正整数，你需要留意你要缓存的对象和你的内在环境，默认值是 1024。

readOnly 属性可以被设置为 true 或 false。只读缓存将对所有调用者返回同一个实例。因此都不能被修改，这可以极大的提高性能。可写的缓存将通过序列化来返回一个缓存对象的拷贝。这会比较慢，但是比较安全。所以默认值是 false。

使用自定义缓存

这里来附加说一下自定义缓存，你完全可以重写缓存的动作去实现你自己的缓存。或者创建一个适配器去使用第三方的解决方案。

```
<cache type=”com.domain.something.MyCustomCache”/>
```

这个例子演示了如果使用一个定制的缓存实现。这个在 type 属性中指定的类，必须实现 org.mybatis.cache.Cache 接口。这个接口是 MyBatis 框架比较复杂的之一，先给个示例：

```
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();
  ReadWriteLock getReadWriteLock();
}
```

配置你的缓存，简单地添加一个公共的 JavaBeans 属性到你的缓存实现，然后通过 cache 元素设置属性，下面示例，将在你的实现上调用一个 setCacheFile(String file)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的 JavaBeans 属性，MyBatis 会自动转换。

需要牢记的是一个缓存配置和缓存实例都绑定到一个 SQL Map 文件命名空间。因此，所有的这个相同命名空间的语句也都和这个缓存绑定。语句可以修改如何与这个缓存相匹配，或者使用两个简单的属于一条语句接着一条语句地完全排除它们自己。默认情况下，语句像下面这样来配置：

```
<select ... flushCache="false" useCache="true" />
<insert ... flushCache="true" />
<update ... flushCache="true" />
<delete ... flushCache="true" />
```

因为有默认值，所以你不需要再确切地配置这些语句。如果你想改变默认的动作，只需要设置 flushCache 和 useCache 属性即可。举个例子来说，在许多的场合下你可能排除缓存中某些特定的 select 语句。或者你想用 select 语句清空缓存。同样的，你也可能有一些 update 语句在执行的时候不想清空缓存。这时，你就需要一一分别做出设定。

七、cache-ref 缓存引用

回想上一节，我们仅仅只是讨论在特定的某一个命名空间里，使用或者刷新缓存。但有可能你想要在不同的命名空间里共享同一个缓存配置或者实例。在这种情况下，你就可以使用 cache-ref 来引用另外一个缓存。

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

第五章 动态语句

MyBatis 最强大的特性之一是它的动态语句功能。如果你以前使用 JDBC 或者类似的框架，你就会明白把 SQL 语句条件连接在一起是多么的痛苦，一点都不能疏忽空格和逗号等。动态语句完全能解决这些烦恼。

尽管使用动态 SQL 不是开晚会，但是 MyBatis 确实能通过映射的 SQL 语句使用强大的动态 SQL 来解决许多的问题。

动态 SQL 元素对于任何使用过 JSTL 或者类似于 XML 之类的文本处理的人来说，都是非常熟悉的。在上一版本中（iBatis2），动态 SQL 有许多的元素需要学习和了解，但在 MyBatis 3 中，这些都有了许多的改进，现在只剩下了二份这一的元素。MyBatis 使用了基于强大的 OGNL（Object-Graph Navigation Language 的缩写，它是一种功能强大的表达式语言）表达式来避免了大部分其它的元素。

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

if

动态 SQL 中最常做的事情就是用条件地包含一个 where 子句。比如：

```
<select id=”findActiveBlogWithTitleLike”
parameterType=”Blog” resultType=”Blog”>
  SELECT * FROM BLOG
  WHERE state = ‘ACTIVE’
  <if test=”title != null”>
    AND title like #{title}
  </if>
</select>
```

这条语句提供一个带功能性的可选的文字。如果你没有传入标题，那么所以的博客都会被返回，如果你传入了一个标题，那就会寻找相似的标题。

如果我们想要可选地使用标题或者作者查询怎么办？首先，我把语句的名称稍稍改一下，使得看起来更直观。然后简单地加上另外一个条件。

```
<select id=”findActiveBlogLike”
parameterType=”Blog” 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 title like #{author.name}
  </if>
</select>
```

choose, when, otherwise

有时候我们并不想应用所有的条件，而只是想从多个选项中选择一个。类似于Java的switch语句，MyBatis提供了choose元素。

让我继续拿上面的例子来举例，只是现在我们只搜索有查询标题的，或者只搜索有查询作者的。如果都没有提交，则只选有特性的（比如说是管理员加精的或者是置顶的）。

```
<select id="findActiveBlogLike"
parameterType="Blog" 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 title like #{author.name}
    </when>
    <otherwise>
      AND featured = 1
    </otherwise>
  </choose>
</select>
```

trim, where, set

上面的示例已经围绕烦琐的动态SQL发出挑战。考虑一下我们上面提到的‘if’的例子中，如果现在我们把‘ACTIVE=1’也做为条件，会发生什么情况。

```
<select id="findActiveBlogLike"
parameterType="Blog" 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 title like #{author.name}
    </if>
  </select>
```

如果我们一个条件都不给出，会怎么样？语句可能会变成这个样子：

```
SELECT * FROM BLOG
WHERE
```

这会很惨。或者如果我们仅仅只获得了第二个条件，语句又会变成这样：

```
SELECT * FROM BLOG
WHERE
AND title like ‘someTitle’
```

同样会很糟糕。这个问题仅用条件很难简单地解决，如果你已经这么写了，那你可能以后永远都不想犯这样的错了。

MyBatis 有个简单的方案就能解决这里面 90%的问题。如果 where 没有出现的时候，你可以自定一个。稍稍修改一下，就能完全解决：

```
<select id=”findActiveBlogLike”
parameterType=”Blog” 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 title like #{author.name}
    </if>
  </where>
</select>
```

这个“where”元素会知道如果它包含的标签中有返回值的话，它就插入一个‘where’。此外，如果标签返回的内容是以 AND 或 OR 开头的，则它会剔除掉。

如果 where 元素并没有完全按你想要的那样，那你也可以使用 trim 元素自定义它。下面的 trim 也相同于 where：

```
<trim prefix="WHERE" prefixOverrides="AND |OR ">
…
</trim>
```

overrides 元素使用一个管理分隔符组成的文字来进行覆写，空白符也是不能忽略的。这样的结果是移出了所有指定在 overrides 里的字符，使用 with 属性里的字符来覆写。
在动态 update 语句里相似的情形叫做 set。这个 set 元素被用做动态囊括列名来更新，而忽略其它的。比如：

```
<update id="updateAuthorIfNecessary"
        parameterType="domain.blog.Author">
    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>
```

set 元素将动态的配置 SET 关键字，也用来剔除追加到条件末尾的任何不相关的逗号。

如果你非常想知道，等同的 trim 怎么写，是这样：

```
<trim prefix="SET" suffixOverrides=",">
…
</trim>
```

注意，我们只写了一个前缀，同样我们也可以追加一个后缀。

foreach

另外一个对于动态 SQL 非常必须的，主是要迭代一个集合，通常是用于 IN 条件。比如说：

```
<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 元素非常强大，允许你指定一个集合，申明一个项和一个索引变量，用在这个元素的方法体内。也允许你指定开始和结束的字符，也可以在两个迭代器之间加入一个分隔符。它的智能之处在于它不会偶尔追加额外的分隔符。

注意：你可以传入一个 List 实例或者一个数组给 MyBatis 作为一个参数对象。如果你这么做，MyBatis 会自动将它包装成一个 Map，使用它的名称做为 key。List 实例将使用“list”做为键，数组实例以“array”

作为键。

关于 XML 的配置文件和 XML 映射文件就讨论这么多。下一章我们将详细讨论 JAVA API，到这里，你已经学会了绝大部分关于映射的知识。

第六章 Java API

现在你知道了如何来配置 MyBatis 和生成映射，你已经具备了相当的技能。MyBatis Java API 将让你的努力得到回报。如你所见，相比起 JDBC，MyBatis 极大的化简了你的代码且保持清晰，易于理解和维护。MyBatis 3 有了许多显著的进步，SQL 映射也变得更加优秀。

一、目录结构

在我们深入 Java API 之前，理解最佳的目录结构非常重要。MyBatis 非常有弹性 ，你可以对你的文件做任何的手脚。但是做为一个框架，总有一个比较好的方式。

让我来看一下典型的应用目录结构

```
/my_application
    /bin
    /devlib
    /lib（存放 MyBatis 的*.jar 文件）
    /src
        /org/myapp/
            /action
            /data （MyBatis 的手编文件，包含映射类，XML 配置文件，XML 映射文件）
                /SqlMapConfig.xml
                /BlogMapper.java
                /BlogMapper.xml
            /model
            /service
            /view
        /properties （放置 XML 配置文件里的属性文件）
    /test
        /org/myapp/
            /action
            /data
            /model
            /service
            /view
        /properties
    /web
        /WEB-INF
            /web.xml
```

记住，这仅仅是建议，不是必须。但如果你按标准来放置文件，别的开发人员会非常熟悉。

以后的例子，都是假设你是按这样的目标结构来放置文件的

二、SqlSessions

SqlSession 是使用 MyBatis 用到的最主要的类。通过这个接口你可以执行命令、获取映射以及管理事务。我们等一会儿会详细地讨论 SqlSession，首先我们来学习一直如果获取一个 SqlSession 实例。SqlSessions 是通过 SqlSessionFactory 实例来生成的。SqlSessionFactory 包含的方法可能通过各种渠道来生成 SqlSessions。SqlSessionFactory 本身是被 SqlSessionFactoryBuilder 通过 XML、注解和手动 Java 配置生成。

SqlSessionFactoryBuilder

SqlSessionFactoryBuildere 有五个 build()方法，每一个都允许通过不能起点严重成 SqlSession。

```
SqlSessionFactory build(Reader reader)
SqlSessionFactory build(Reader reader, String environment)
SqlSessionFactory build(Reader reader, Properties properties)
SqlSessionFactory build(Reader reader, String env, Properties props)
SqlSessionFactory build(Configuration config)
```

前四个方法较为常用，他们使用一个 Reader 来读取一个 XML 文件。 SqlMapConfig.xml 我们已经学习过。可选参数是 environment 和 properties。Environment 决定哪有哪一些环境将被加载，包括数据源和事务管理器。示例：

```
<environments default="development">
    <environment id="development">
        <transactionManager type="JDBC">
        …
<dataSource type="POOLED">
        …
    </environment>
    <environment id="production">
        <transactionManager type="EXTERNAL">
        …
<dataSource type="JNDI">
        …
    </environment>
</environments>
```

如果你使用 environment 参数调用 build 方法，那 MyBatis 将会合适环境的配置。当然，如果你指定了不正确的环境，那你会出错。如果你没有使用 environment 参数调用 build 方法。MyBatis 会调用默认的环境。

如果使用一个属性实例调用一个方法，那 MyBatis 把这些属性变量传入到你的配置中。那些属性可以在绝大部分通过 ${属性名}来调用。

回想一下，属性可以从 SqlMapConfig.xml 文件中引用，或者直接指定。因此，搞懂它的优先级非常重要。我再已经在以前提及过，现在再再次重申一下：

如果一个属性在多个地方，MyBatis 会按下列顺序加载它们。

- 在属性元素方法体内指定的属性最先读取
- 从类路径加载的资源或者属性 url 属性其次读取，且会覆盖已经指定的重复属性。
- 通过方法参数传入的属性最好读取，且覆盖上面两种方式已经指定的重复属性。

因此，最好的优先级是通过方法会话的参数，其它是资源路径和 URL 方式，最好是属性方法体里属性元素。

总体来说，这前四个方法都大同小异，使用覆写能让你可选地指定环境和属性。下面是一个从 SqlMapConfig.xml 文件生成 SqlSessionFactory 的例子。

```
String resource = "org/mybatis/builder/MapperConfig.xml";
Reader reader = Resources.getResourceAsReader(resource);
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(reader);
```

注意我们使用的 Resources 工具类，它是在 org.mybatis.io 包里。这个 Resources 类，就像它的名称指的一样，可以帮助你从类路径、文件系统或者是 URL 中加载资源。浏览一下这个类的源代码，或者使用你的 IDE 查看一下，就会了解一系统有用的方法，简单列表如下：

```
URL getResourceURL(String resource)
URL getResourceURL(ClassLoader loader, String resource)
InputStream getResourceAsStream(String resource)
InputStream getResourceAsStream(ClassLoader loader, String resource)
Properties getResourceAsProperties(String resource)
Properties getResourceAsProperties(ClassLoader loader, String resource)
Reader getResourceAsReader(String resource)
Reader getResourceAsReader(ClassLoader loader, String resource)
File getResourceAsFile(String resource)
File getResourceAsFile(ClassLoader loader, String resource)
InputStream getUrlAsStream(String urlString)
Reader getUrlAsReader(String urlString)
Properties getUrlAsProperties(String urlString)
Class classForName(String className)
```

最后一个 build 方法使用一个 Configuration 实例，这个 Configuration 包含了所有你可能需要的了解的关于 SqlSessionFactory 对象。Configuration 类对于内省的配置非常有用，包括查询和操作 SQL 映射。Configuration 有你已经学过的每个配置开关，就像一个 Java API。下面是一个示例，演示了如果来手册改配置一个 Configuration 实例，然后再传入到 build 方法，用来舒服一个 SqlSessionFactory。

```
DataSource dataSource = BaseDataTest.createBlogDataSource();
```

```
TransactionFactory transactionFactory = new JdbcTransactionFactory();

Environment environment =
new Environment("development", transactionFactory, dataSource);

Configuration configuration = new Configuration(environment);
configuration.setLazyLoadingEnabled(true);
configuration.setEnhancementEnabled(true);
configuration.getTypeAliasRegistry().registerAlias(Blog.class);
configuration.getTypeAliasRegistry().registerAlias(Post.class);
configuration.getTypeAliasRegistry().registerAlias(Author.class);
configuration.addMapper(BoundBlogMapper.class);
configuration.addMapper(BoundAuthorMapper.class);

SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(configuration);
```

现在你已经有了一个 SqlSessionFactory，可以用来创建 SqlSession 实例了。

SqlSessionFactory

SqlSessionFactory 有六个方法可用来创建 SqlSession 实例，选择其中一个方法，通常要考虑下面几个方面：

- **事务**：你是否想为会话使用事务作用域，或者使用自动提交。（一般是在数据库或者 JDBC 驱动器没有事务的情况下）
- **连接**：你是否要 MyBatis 从配置的数据源中获取一个连接，或者使用自定义的连接。
- **执行**：你是否要 MyBatis 重复使用 PreparedStatements 或者批量更新，包括插入和删除。

重载了 openSession()的这一些方法允许你选择任何有关的选项，使得更有意义。

```
SqlSession openSession()
SqlSession openSession(boolean autoCommit)
SqlSession openSession(Connection connection)
SqlSession openSession(TransactionIsolationLevel level)
SqlSession openSession(ExecutorType execType,TransactionIsolationLevel
level)
SqlSession openSession(ExecutorType execType)
SqlSession openSession(ExecutorType execType, boolean autoCommit)
SqlSession openSession(ExecutorType execType, Connection connection)
Configuration getConfiguration();
```

默认的 openSession() 不使用参数，生成的 SqlSession 将有以下特性：

- 将会启动一个事务作用域（也就是不会自动提交）
- 从一个 DataSource 实例中获取一个连接对象，这个 DataSource 实例从现行的环境中配置生成。
- 事务的隔离级别，使用驱动器或者是数据源的默认级别。
- 没有 PreparedStatements 语句能重复使用，也不能执行批量更新。

大部分的方法本身就非常容易理解。如果要开启自动提交，传入一个‘true’值给 autoCommit 参数。如果要使用你自己的连接，传入一个连接实例给 connection 参数。注意，没有同时包括 Connection 和 autoCommit 两个参考的方法。因为 MyBatis 将使用任何设置都会对正在使用的对象起作用。，TransactionIsolationLevel 是指事务调用级别，MyBatis 使用枚举类型来包装，他们按预期方式工作和具有 5 个级别通过 JDBC 支持。 (NONE, READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, ERIALIZABLE)

还有一个新参数，就是 ExecutorType。这个枚举类型有三个值：

ExecutorType.SIMPLE

这个类型不做特殊的事情，它只为每个语句创建一个 PreparedStatement。

ExecutorType.REUSE

这种类型将重复使用 PreparedStatements。

ExecutorType.BATCH

这个类型批量更新，且必要地区别开其中的 select 语句，确保动作易于理解。

注意：SqlSessionFactory 中还有一个方法我们没有提到，那就是 getConfiguration()。这个方法返回一个 Configuration 实例，让你可用来与 MyBatis 运行里的进行自省。

注意：如果你已经使用过以前的 MyBatis 版本，你会回想起会话、事务和批量处理都是分开的。现在不再是这个样子，这三者都完美地包括在一个 sesson 作用域里。你不需要分开处理事务和批量处理了。

三、SqlSession

就像上面所提及的那样，SqlSession 实例是 MyBatis 最强大的类。类里有执行语句、提交或回滚事务、获取映射实例等所有方法。
SqlSession 类里有超过二十个方法，我们现在来拆分，分组讲解。

语句执行方法

这些方法用来执行定义在 SQL 映射 XML 文件里的 SELECT, INSERT, UPDATE 和 DELETE 语句。它们本身非常好理解，执行以 ID 为标识的语句，使用参数对象。也可以是 JAVA 的基本类型（自动装箱，包装类）、JavaBean、POJO 或是 Map。

```
Object selectOne(String statement, Object parameter)
List selectList(String statement, Object parameter)
int insert(String statement, Object parameter)
int update(String statement, Object parameter)
int delete(String statement, Object parameter)
```

selectOne 和 selectList 的不同之处是，selectOne 必须返回且只返回一条记录，如果是多条或者没有（null）都会抛出异常。如果你并不知道会选择出多少条记录，最好使用 selectList。如果你只是想验证是否有想要的记录存在，可以返回一个计数（0 或者 1）。因为并不是所有的语句都需要参数，所以这些方法都可以重写成其它不需要传入参数的版本。

```
Object selectOne(String statement)
```

```
List selectList(String statement)
int insert(String statement)
int update(String statement)
int delete(String statement)
```

最后，还有三个 select 方法的高级版本，永远你严格限定返回记录的范围，或者使用自定义的结果处理逻辑，通常是用来处理大数据集。

```
List selectList
(String statement, Object parameter, RowBounds rowBounds)
void select
(String statement, Object parameter, ResultHandler handler)
void select
(String statement, Object parameter, RowBounds rowBounds,
ResultHandler handler)
```

RowBounds 参数可让 MyBatis 跳过指定数目的记录，也可以限制只返回一定数据的数据结果。RowBounds 有一个构造函数可以设置一个偏移量和一个限制长度，RowBounds 是不可变的。

```
int offset = 100;
int limit = 25;
RowBounds rowBounds = new RowBounds(offset, limit);
```

不同的驱动器在这方面可以达到不能的效率级别。为了最好的性能，将返回结果类型标为 SCROLL_SENSITIVE 或者 SCROLL_INSENSITIVE，注意，没有 FORWARD_ONLY。

ResultHandler 参数允许你对每一条记录进行任何的操作，你可以把它添加进一个 List，生成一个 Map、Set，或者单独拿出进行统计和计算。你可以使用 ResultHandler 做任何的事情，MyBatis 也是使用它来生成结果的集合。

ResultHandler 接口非常简单：

```
package org.mybatis.executor.result;
public interface ResultHandler {
  void handleResult(ResultContext context);
}
```

ResultContext 允许你访问结果对象本身、结果对象的数目、一个 Boolean 类型的 stop()方法，以让你从停止 MyBatis 加载更多的结果。

事务控制方法

在事务里，有四个方法控制作用域。当然，如果你使用了自动提交或者使用了一个外部的事务管理器，那它就不会起作用。然后，如果你使用的是 JDBC 事务管理器，使用的是 Connection 实例来管理，那这四个方法就是手到擒来。

```
void commit()
void commit(boolean force)
void rollback()
void rollback(boolean force)
```

默认的情况下，MyBatis 并不真正的提交，除非它监测到数据库有了 insert, update or delete 操作。如果你没有调用这些方法而却做了一些改动，那你需要传入一个 true 到 commit 和 rollback 方法里，以确保它能提交。大部分的时候，你不需要调用 rollback()，因为如果你不调用提交方法的话，MyBatis 会为你去回滚。然后，如果你需要更好地在一个有多重提交和回滚的会话里控制粒度。你可以使用回滚的选项来操控。

清除会话层缓存

```
void clearCache()
```

SqlSession 实例持有的一个本地缓存，在执行 update, commit, rollback and close 之后会被清空。如果想明确地关闭清空，可以调用 clearCache()方法。

确保 SqlSession 已关闭

```
void close()
```

最重要的事情是要确保你打开的所有会话都已经关闭。确保的最好方式就是使用下面的工作模式。

```
SqlSession session = sqlSessionFactory.openSession();
try {
    // following 3 lines pseudocod for “doing some work”
    session.insert(…);
    session.update(…);
    session.delete(…);
    session.commit();
} finally {
    session.close();
}
```

注意：就像 SqlSessionFactory 一样，你可以得到一个 Configuration 实例，SqlSession 也可以调用 getConfiguration()方法来获取一个 Configuration 实例。

```
Configuration getConfiguration()
```

使用映射

```
<T> T getMapper(Class<T> type)
```

虽然各种各样的 insert, update, delete and select 方法非常强大，但是它们却也很冗长，不是类型安全的，也帮不了 IDE 和单元测试起作用。我们在最开始的时候，就看过这样的例子。

因此，执行映射语句更通常的方式。一个映射类是一个简单的接口，定义好与 SqlSession 里的方法相匹配的方法。下面的示例类示范了一些方法签名和如果映射到 SqlSession 中。

```
public interface AuthorMapper {
  // (Author) selectOne(“selectAuthor”,5);
  Author selectAuthor(int id);
  // (List<Author>) selectList(“selectAuthors”)
  List<Author> selectAuthors();
  // insert(“insertAuthor”, author)
  void insertAuthor(Author author);
  // updateAuthor(“updateAuhor”, author)
```

```
    void updateAuthor(Author author);
    // delete(“deleteAuthor”,5)
    void deleteAuthor(int id);
  }
```

总体来说，每个映射方法签名，都要与一个 SqlSession 的方法相对对应，方法也也要与映射文句的 ID 相对应。

另外，返回值的类型要与想要的类型想一致。支持所有的类型：基本类型，Maps, POJOs and JavaBeans。
映射接口并不需要实现任何接口或者继承任何的类。只要方法签名能被唯一对应的语句执行即可。
映射接口也可以继承其它的接口。当使用 XML 绑定到映射接口的时候，只要确定你在对应的命名空间里有语句。唯一的限制就是不能在同一层次的两个接口中有相同的方法。

你可以传递多个参数给映射方法。如果你传了多个，那他们将会按他们的次序依次排名称，像这样：#{1}, #{2} 等等。如果你想改变他们的名称，那你可以使用@Param(“paramName”) 注解来传入参数。

你也可以传入一个 RowBounds 实例来限制查询结果。

映射注解

从最开始的时候，MyBatis 曾是一个 XML 驱动的框架，配置文件是 XML，语句映射也是定义在 XML 文件。但在 MyBatis3 里，增添了许多项目。MyBatis 3 生成在一个宽泛且强大的基于 Java 的配置 API 之上。这个配置 API，是基于 XML 的 MyBatis 配置的基础，也是基于注解配置的基础。注解提供一种简便地方式来简单地映射语句。

注意：JAVA 注解在表达式和弹性上还是有一些限制的，尽管有花大量的时间来研究、设计和尝试，但强大的 MyBatis 映射还是无法用注解来生成。

注解元素如下表：

注解	目标	等同 XML	说明
@CacheNamespace	Class	<cache>	配置给定名称的缓存，属性有：implementation, eviction, flushInterval, size 和 readWrite。
@CacheNamespaceRef	Class	<cacheRef>	引用另一个命名空间的缓存。属性：value。是一个命名空间的的名称。
@ConstructorArgs	Method	<constructor>	收集一组结果传入到一个结果对象的构造器。属性：value。参数对象的数组
@Arg	Method	<arg> <idArg>	单个构造器参数是 ConstructorArgs 集合的一部分。属性：id, column, javaType, jdbcType, typeHandler。这个 id 是一个字符串类型的值，用于标识这个属性。类似于<idArg>这个 XML 元素
@TypeDiscriminator	Method	<discriminator>	一组值表达式，决定结果该怎么映射。属性：column, javaType, jdbcType, typeHandler, cases。cases 属性是一个表达式数组。
@Case	Method	<case>	一个值的单个表达式和它对应的映射。属性：

			value, type, results. 这个结果属性是一组结果数组，因此，Case 注解非常类似于真正的 ResultMap，通过下面的 Results 注解指定。
@Results	Method	<resultMap>	一个结果映射集合，包含的项是结果列名与字段的映射。属性：value，一个结果集注解的数组
@Result	Method	<result> <id>	单个的结果列名与字段的映射，属性： id, column, property, javaType, jdbcType, typeHandler, one, many。这个 id 是一个 boolean 类型的值，用于标识这个属性是否用来被比较。类似于 XML 映射中的<id>。one 属性是用于单个联合，类似于<association>，many 属性用于聚集，类似于<collection>。他们的命名，要避免与类名相冲突。
@One	Method	<association>	一个复杂类型的单个属性值的映射。属性：select，映射语句（也就是映射方法）的全名，用于加载一个相应类型的实例。注意：你可能已经注意到了不能使用注解来支持联接映射，这是由于 JAVA 注解不支持循环引用。
@Many	Method	<collection>	一个复杂类型的集合的映射。属性：select，映射语句（也就是映射方法）的全名，用于加载一个相应类型的实例。注意：你可能已经注意到了不能使用注解来支持联接映射，这是由于 JAVA 注解不支持循环引用。
@Options	Method	映射语句属性	这个注解提供广泛的切换访问和配置通常在映射语句里做了属性的选项。比起把每个语句错综在一起，Options 注解提供一个一致且清晰的方式去访问。属性：useCache=true, flushCache=false, resultSetType=FORWARD_ONLY, statementType=PREPARED, fetchSize=-1, timeout=-1, useGeneratedKeys=false, keyProperty=“id”。理解 JAVA 注解非常重要，要注意不能设置“null”作为值。因此。当你处理 Options 注解的时候，你的语句都要设置一个默认值，尽量把默认值设成你想要的结果。
@Insert @Update @Delete @Select	Method	<insert> <update> <delete> <select>	这里的每个注解，都代表一个可以执行的实际的 SQL 语句。他们可以是一组或者是一个字符串。如果是传入一个字符串数组，那么会使用一个空白符来连接这些字符串，拼成一条语句。直接传入一个字符串也行。属性：value。要传入的字符串数组。
@InsertProvider @UpdateProvider @DeleteProvider @SelectProvider	Method	<insert> <update> <delete> <select> 允许动态	这些可选的 SQL 注解，允许你在运行的过程中，指定类名和方法名返回 SQL 来执行。针对于执行映射语句，MyBatis 实例化类，执行方法。属性：type, method。type 属性指定类的全名，方法属性指定方法名称。注意：下一章中，我们会讨

		生成 SQL	论到 SelectBuilder 类，可以帮且我们更简单明了地生成动态 SQL。
@Param	Parameter		如果你的映射方法使用多个参数，那这个注解能为方法的每个参数设定一个参数名称。否则，多个参数会使用它们的序列号命名（RowBounds 参数除外），像：#{1}，#{2}等等。如果使用@Param(“person”)，则这个参数就被叫做#{person}。

第七章 SelectBuilder

对于一个 JAVA 开发者来说，最厌恶的事情就是在 Java 代码中嵌入 SQL 语句。通常这样做是因为 SQL 要动态的生成，要不然你可以把它放置在一个文件或者存储过程中。就像你所看到的，MyBatis 在 XML 映射特性中有一个强劲的方案用于 SQL 动态生成。然而，有时候却必须要在 JAVA 代码中也生成 SQL 语句。这种情况下，MyBatis 也有一个特性来帮你解决。将你从一大堆的加号、引号、错乱的格式和嵌套条件中解剖。

MyBatis 3 引入了一个稍微有点不同的概念。我们可以生成一个实例，然后使用这个实例一步一步的生成 SQL 语句。只是最终我看起来更像是 JAVA 代码而不是 SQL 代码。

SelectBuilder 的秘密

SelectBuilder 类也并不是巫术，如果不去了解它，它自己并不能完美的解决什么。现在让我们看看它能做些什么。SelectBuilder 使用一组静态导入，和一个 ThreadLocal 变量去启用一个清空语法以方便组合条件语句，且梳理好 SQL 的格式。它可以像这样创建：

```
public String selectBlogsSql() {
    BEGIN(); // Clears ThreadLocal variable
    SELECT("*");
    FROM("BLOG");
    return SQL();
}
```

这只是一个生成静态语句的最简单的例子，还可以像下面这样稍稍复杂一点的：

```
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();
```

```
    }
```

上面的方法将会生成下面这样的语句：

```
"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";
```

写成 SQL 语句，会比较复杂，尤其是要注意每一行最后都有添加一个空白符。这面这个例子，显然会比 Java 字符串连接更简单：

```
    private String selectPersonLike(Person p){
      BEGIN(); // Clears ThreadLocal variable
      SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FIRST_NAME, P.LAST_NAME");
      FROM("PERSON P");
      if (p.id != null) {
        WHERE("P.ID like #{id}");
      }
      if (p.firstName != null) {
        WHERE("P.FIRST_NAME like #{firstName}");
      }
      if (p.lastName != null) {
          WHERE("P.LAST_NAME like #{lastName}");
      }
      ORDER_BY("P.LAST_NAME");
      return SQL();
    }
```

这个例子有什么特别之处？如果你观察比较仔细地话，你就会发现，现也不用担心是不是会多出“AND”关键字，“WHERE”和“AND”之间是不是只是空白，等等，上面生成的语句将会生成一个查询 PERSON 所有记录的例子。可能是 ID 匹配参数的，也可能是 firstName 匹配参数的，也可能是 lastName 匹配参数的，也可能是三者都匹配的。SelectBuilder 会注意什么哪里需要添加“WHERE”，哪里需要添加“AND”，且不用关心它的调用顺序。

你可能已经留意到了两个方法： BEGIN() 和 SQL()。概括来说，每一个 SelectBuilder 里的方法，都是以 BEGIN()开头，以 SQL()结尾，在中间部分不按次序。BEGIN()清零 ThreadLocal 变量，SQL()方法收集从 BEGIN()开始处的所有语句。BEGIN()还且个同义词 RESET()，两者完全相同。

如果想像上例中那样使用 SelectBuilder，你需要静态导入：

```
import static org.mybatis.jdbc.SelectBuilder.*;
```

静态导入过后，你就可以直接使用 SelectBuilder 类的所有方法了。所有方法如下表：

方法	描述
BEGIN() / RESET()	这个方法清空 SelectBuilder 的 ThreadLocal 状态，准备好新建一个语句。BEGIN()这个方法名做一个句子的开头会好理解一点。 RESET()方法名，在执行过程中由于一些原因而要清空语句，会比较好理解。
SELECT(String)	开始或者追加一个 SELECT 子句。也可被多次调用，参数将会被追加到 SELECT 子句中。参数通常是用一个逗号分隔的列名和别名，或者是驱动器支持的其它类型。
SELECT_DISTINCT(String)	开始或者追加一个 SELECT 子句，同时，也会生成“DISTINCT”关键到生成的查询中。也可被多次调用，参数将会被追加到 SELECT 子句中。参数通常是用一个逗号分隔的列名和别名，或者是驱动器支持的其它类型。
FROM(String)	开始或者追加一个 FROMT 子句。也可被多次调用，参数将会被追加到 FROM 子句中。参数通常是用一个逗号分隔的列名和别名，或者是驱动器支持的其它类型。
JOIN(String) INNER_JOIN(String) LEFT_OUTER_JOIN(String) RIGHT_OUTER_JOIN(String)	添加一个适当类型的新 JOIN 子句，取决你调用哪个方法。参数可是一个标准联接的列名也可以是要联接的条件。
WHERE(String)	追加一个亲的的 WHERE 条件子句，使用 AND 来连接。可能被多次调用，每次都是使用 AND 来连接新条件，如果要使用 OR 的形式，使用 OR()方法。
OR()	使用 OR 形式把 WHERE 条件分隔开，也被重复调用，但如果对某一行进行重复调用的话，就会产生错误。
AND()	使用 AND 形式把 WHERE 条件分隔开，也被重复调用，但如果对某一行进行重复调用的话，就会产生错误。由于 WHERE 和 HAVING 都会自动生成 AND，所以这个方法极少用到。
GROUP_BY(String)	追加一个新的 GROUP BY 的子句元素，使用逗号连接。可以重复调用，每次调用都会使用逗号连接新的条件。
HAVING(String)	追加一个新的 HAVING 的子句条件，使用 AND 连接。可以重复调用，每次调用都会使用 AND 连接新的条件。可以使用 OR()来用 OR 连接。
ORDER_BY(String)	追加一个新的 GROUP BY 的子句元素，使用逗号连接。可以重复调用，每次调用都会使用逗号连接新的条件。
SQL()	返回生成 SQL()，重新设置 SelectBuilder 状态，也就是 BEGIN() 或者 RESET() 将被调用。因此，这个方法只能调用一次。

第八章 SqlBuilder

与 SelectBuilder 相似，MyBatis 也包含了一个通用的 SqlBuilder 类，它包含了 SelectBuilder 的所有方法，也有一些针对 inserts, updates, 和 deletes 的方法。这个类在在 DeleteProvider 、 nsertProvider, or UpdateProvider 和 SelectProvider 里生成 SQL 语句非常有用。

使用时必须导入包：

```
import static org.mybatis.jdbc.SqlBuilder.*;
```

除了拥有所有 SelectBuilder 里的方法以外（注意：源类并不是继承自 SelectBuilder），还有以下方法：

方法	描述
DELETE_FROM(String)	开始一个 delete 语句，且指定一个表名称。通常，最好给定一个 WHERE 语句
INSERT_INTO(String)	开始一个 insert 语句，且指定一个表名称。后来要紧接一个或多个 VALUES()调用。
SET(String)	追加到 update 语句的 SET 列表里
UPDATE(String)	开始一个 update 语句，且指定一个表名称。后来会紧接一个或多个 SET()调用，且有时候会紧接一个 WHERE()调用。
VALUES(String, String)	追加到一个 insert 语句，第一个参数是要插入的列名，第二个参数是要插入的值。

```
public String deletePersonSql() {
    BEGIN(); // Clears ThreadLocal variable
    DELETE_FROM("PERSON");
    WHERE("ID = ${id}");
    return SQL();
}

public String insertPersonSql() {
    BEGIN(); // Clears ThreadLocal variable
    INSERT_INTO("PERSON");
    VALUES("ID, FIRST_NAME", "${id}, ${firstName}");
    VALUES("LAST_NAME", "${lastName}");
    return SQL();
}

public String updatePersonSql() {
    BEGIN(); // Clears ThreadLocal variable
    UPDATE("PERSON");
    SET("FIRST_NAME = ${firstName}");
    WHERE("ID = ${id}");
    return SQL();
}
```

第九章 说明

在 MyBatis3.0.1 的发布包中，包名称还是使用 org.apache.ibatis.*的包名。如果你下载的版本有所不同，最好查看一下源代码文件，以便正确导入。

由于这是本人的第一次翻译文档，水平极其有限。仅供参照，以官方英文原稿语义为准。