**Introduction 简介**

**What is MyBatis? 什么是 MyBatis？**

MyBatis is a first class persistence framework with support for custom SQL, stored procedures and advanced mappings. MyBatis eliminates almost all of the JDBC code and manual setting of parameters and retrieval of results. MyBatis can use simple XML or Annotations for configuration and map primitives, Map interfaces and Java POJOs (Plain Old Java Objects) to database records.

MyBatis 是一款优秀的持久层框架，它支持定制化 SQL、存储过程以及高级映射。MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。MyBatis 可以使用简单的 XML 或注解来配置和映射原生类型、接口和 Java 的 POJO（Plain Old Java Objects，普通老式 Java 对象）为数据库中的记录。

**Help make this documentation better…**

帮助改进文档...

If you find this documentation lacking in any way, or missing documentation for a feature, then the best thing to do is learn about it and then write the documentation yourself!

如果你发现文档有任何的缺失，或者缺少某一个功能点的说明，最好的解决办法是先自己学习，并且为缺失的部份补上相应的文档。

Sources of this manual are available in xdoc format at project's Git Fork the repository, update them and send a pull request.

该文档 xdoc 格式的源码文件可通过项目的 Git 代码库来获取。Fork 该源码库，作出更新，并提交 Pull Request 吧。

You’re the best author of this documentation, people like you have to read it!

还有其他像你一样的人都需要阅读这份文档，而你，就是这份文档最好的作者。

**Translations 文档的翻译版本**

Users can read about MyBatis in following translations:

您可以阅读 MyBatis 文档的其他语言版本：

* English
* Español
* 日本語
* 한국어
* 简体中文

Do you want to read about MyBatis in your own native language? File an issue providing patches with your mother tongue documentation!

想用你的母语来了解 MyBatis 吗？那就将文档翻译成你的母语并提供给我们吧！

**Getting started 入门**

**Installation 安装**

To use MyBatis you just need to include the mybatis-x.x.x.jar file in the classpath.

要使用 MyBatis， 只需将 mybatis-x.x.x.jar 文件置于 classpath 中即可。

If you are using Maven just add the following dependency to your pom.xml:

如果使用 Maven 来构建项目，则需将下面的 dependency 代码置于 pom.xml 文件中：

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>x.x.x</version>

</dependency>

**Building SqlSessionFactory from XML**

从 XML 中构建 SqlSessionFactory

Every MyBatis application centers around an instance of SqlSessionFactory. A SqlSessionFactory instance can be acquired by using the SqlSessionFactoryBuilder. SqlSessionFactoryBuilder can build a SqlSessionFactory instance from an XML configuration file, or from a custom prepared instance of the Configuration class.

每个基于 MyBatis 的应用都是以一个 SqlSessionFactory 的实例为核心的。SqlSessionFactory 的实例可以通过 SqlSessionFactoryBuilder 获得。而 SqlSessionFactoryBuilder 则可以从 XML 配置文件或一个预先定制的 Configuration 的实例构建出 SqlSessionFactory 的实例。

Building a SqlSessionFactory instance from an XML file is very simple. It is recommended that you use a classpath resource for this configuration, but you could use any InputStream instance, including one created from a literal file path or a file:// URL. MyBatis includes a utility class, called Resources, that contains a number of methods that make it simpler to load resources from the classpath and other locations.

从 XML 文件中构建 SqlSessionFactory 的实例非常简单，建议使用类路径下的资源文件进行配置。 但是也可以使用任意的输入流（InputStream）实例，包括字符串形式的文件路径或者 file:// 的 URL 形式的文件路径来配置。MyBatis 包含一个名叫 Resources 的工具类，它包含一些实用方法，可使从 classpath 或其他位置加载资源文件更加容易。

String resource = "org/mybatis/example/mybatis-config.xml";

InputStream inputStream = Resources.getResourceAsStream(resource);

SqlSessionFactory sqlSessionFactory =

new SqlSessionFactoryBuilder().build(inputStream);

The configuration XML file contains settings for the core of the MyBatis system, including a DataSource for acquiring database Connection instances, as well as a TransactionManager for determining how transactions should be scoped and controlled. The full details of the XML configuration file can be found later in this document, but here is a simple example:

XML 配置文件中包含了对 MyBatis 系统的核心设置，包含获取数据库连接实例的数据源（DataSource）和决定事务作用域和控制方式的事务管理器（TransactionManager）。 XML 配置文件的详细内容后面再探讨，这里先给出一个简单的示例：

<?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>

While there is a lot more to the XML configuration file, the above example points out the most critical parts. Notice the XML header, required to validate the XML document. The body of the environment element contains the environment configuration for transaction management and connection pooling. The mappers element contains a list of mappers – the XML files and/or annotated Java interface classes that contain the SQL code and mapping definitions.

当然，还有很多可以在 XML 文件中进行配置，上面的示例指出的则是最关键的部分。 要注意 XML 头部的声明，它用来验证 XML 文档正确性。environment 元素体中包含了事务管理和连接池的配置。mappers 元素则是包含一组映射器（mapper），这些映射器的 XML 映射文件包含了 SQL 代码和映射定义信息。

**Building SqlSessionFactory without XML**

不使用 XML 构建 SqlSessionFactory

If you prefer to directly build the configuration from Java, rather than XML, or create your own configuration builder, MyBatis provides a complete Configuration class that provides all of the same configuration options as the XML file.

如果你更愿意直接从 Java 代码而不是 XML 文件中创建配置，或者想要创建你自己的配置构建器，MyBatis 也提供了完整的配置类，提供所有和 XML 文件相同功能的配置项。

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);

Notice in this case the configuration is adding a mapper class. Mapper classes are Java classes that contain SQL Mapping Annotations that avoid the need for XML. However, due to some limitations of Java Annotations and the complexity of some MyBatis mappings, XML mapping is still required for the most advanced mappings (e.g. Nested Join Mapping). For this reason, MyBatis will automatically look for and load a peer XML file if it exists (in this case, BlogMapper.xml would be loaded based on the classpath and name of BlogMapper.class). More on this later.

注意该例中，configuration 添加了一个映射器类（mapper class）。映射器类是 Java 类，它们包含 SQL 映射语句的注解从而避免依赖 XML 文件。不过，由于 Java 注解的一些限制以及某些 MyBatis 映射的复杂性，要使用大多数高级映射（比如：嵌套联合映射），仍然需要使用 XML 配置。有鉴于此，如果存在一个同名 XML 配置文件，MyBatis 会自动查找并加载它（在这个例子中，基于类路径和 BlogMapper.class 的类名，会加载 BlogMapper.xml）。具体细节稍后讨论。

**Acquiring a SqlSession from SqlSessionFactory**

从 SqlSessionFactory 中获取 SqlSession

Now that you have a SqlSessionFactory, as the name suggests, you can acquire an instance of SqlSession. The SqlSession contains absolutely every method needed to execute SQL commands against the database. You can execute mapped SQL statements directly against the SqlSession instance. For example:

既然有了 SqlSessionFactory，顾名思义，我们就可以从中获得 SqlSession 的实例了。SqlSession 完全包含了面向数据库执行 SQL 命令所需的所有方法。你可以通过 SqlSession 实例来直接执行已映射的 SQL 语句。例如：

SqlSession session = sqlSessionFactory.openSession();

try {

Blog blog = session.selectOne(

"org.mybatis.example.BlogMapper.selectBlog", 101);

} finally {

session.close();

}

While this approach works, and is familiar to users of previous versions of MyBatis, there is now a cleaner approach. Using an interface (e.g. BlogMapper.class) that properly describes the parameter and return value for a given statement, you can now execute cleaner and more type safe code, without error prone string literals and casting.

诚然，这种方式能够正常工作，并且对于使用旧版本 MyBatis 的用户来说也比较熟悉。不过现在有了一种更简洁的方式 ——使用正确描述每个语句的参数和返回值的接口（比如 BlogMapper.class），你现在不仅可以执行更清晰和类型安全的代码，而且还不用担心易错的字符串字面值以及强制类型转换。

For example: 例如：

SqlSession session = sqlSessionFactory.openSession();

try {

BlogMapper mapper = session.getMapper(BlogMapper.class);

Blog blog = mapper.selectBlog(101);

} finally {

session.close();

}

Now let's explore what exactly is being executed here.

现在我们来探究一下这里到底是怎么执行的。

**Exploring Mapped SQL Statements**

探究已映射的 SQL 语句

At this point you may be wondering what exactly is being executed by the SqlSession or Mapper class. The topic of Mapped SQL Statements is a big one, and that topic will likely dominate the majority of this documentation. But to give you an idea of what exactly is being run, here are a couple of examples.

现在你可能很想知道 SqlSession 和 Mapper 到底执行了什么操作，但 SQL 语句映射是个相当大的话题，可能会占去文档的大部分篇幅。 不过为了让你能够了解个大概，这里会给出几个例子。

In either of the examples above, the statements could have been defined by either XML or Annotations. Let's take a look at XML first. The full set of features provided by MyBatis can be realized by using the XML based mapping language that has made MyBatis popular over the years. If you've used MyBatis before, the concept will be familiar to you, but there have been numerous improvements to the XML mapping documents that will become clear later. Here is an example of an XML based mapped statement that would satisfy the above SqlSession calls.

在上面提到的例子中，一个语句既可以通过 XML 定义，也可以通过注解定义。我们先看看 XML 定义语句的方式，事实上 MyBatis 提供的全部特性都可以利用基于 XML 的映射语言来实现，这使得 MyBatis 在过去的数年间得以流行。如果你以前用过 MyBatis，你应该对这个概念比较熟悉。 不过自那以后，XML 的配置也改进了许多，我们稍后还会再提到。这里给出一个基于 XML 映射语句的示例，它应该可以满足上述示例中 SqlSession 的调用。

<?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" resultType="Blog">

select \* from Blog where id = #{id}

</select>

</mapper>

While this looks like a lot of overhead for this simple example, it is actually very light. You can define as many mapped statements in a single mapper XML file as you like, so you get a lot of mileage out of the XML header and doctype declaration. The rest of the file is pretty self explanatory. It defines a name for the mapped statement “selectBlog”, in the namespace “org.mybatis.example.BlogMapper”, which would allow you to call it by specifying the fully qualified name of “org.mybatis.example.BlogMapper.selectBlog”, as we did above in the following example:

为了这个简单的例子，我们似乎写了不少配置，但实际上它并不多。在一个 XML 映射文件中，可以定义无数个映射语句，这样一来，XML 头部和文档类型声明占去的部分就显得微不足道了。而文件的剩余部分具备自解释性，很容易理解。 在命名空间 “org.mybatis.example.BlogMapper” 中定义了一个名为 “selectBlog” 的映射语句，允许你使用指定的完全限定名 “org.mybatis.example.BlogMapper.selectBlog” 来调用映射语句，就像上面例子中那样：

Blog blog = session.selectOne(

"org.mybatis.example.BlogMapper.selectBlog", 101);

Notice how similar this is to calling a method on a fully qualified Java class, and there's a reason for that. This name can be directly mapped to a Mapper class of the same name as the namespace, with a method that matches the name, parameter, and return type as the mapped select statement. This allows you to very simply call the method against the Mapper interface as you saw above, but here it is again in the following example:

你可能注意到这和使用完全限定名调用 Java 对象的方法类似。这样，该命名就可以直接映射到在命名空间中同名的 Mapper 类，并将已映射的 select 语句中的名字、参数和返回类型匹配成方法。 因此你就可以像上面那样很容易地调用这个对应 Mapper 接口的方法，就像下面这样：

BlogMapper mapper = session.getMapper(BlogMapper.class);

Blog blog = mapper.selectBlog(101);

The second approach has a lot of advantages. First, it doesn't depend on a string literal, so it's much safer. Second, if your IDE has code completion, you can leverage that when navigating your mapped SQL statements.

第二种方法有很多优势，首先它不依赖于字符串字面值，会更安全一点； 其次，如果你的 IDE 有代码补全功能，那么代码补全可以帮你快速选择已映射的 SQL 语句。

**NOTE** A note about namespaces.

**提示**对命名空间的一点说明

**Namespaces** were optional in previous versions of MyBatis, which was confusing and unhelpful. Namespaces are now required and have a purpose beyond simply isolating statements with longer, fully-qualified names.

在之前版本的 MyBatis 中，命名空间（Namespaces）的作用并不大，是可选的。 但现在，随着命名空间越发重要，你必须指定命名空间。

Namespaces enable the interface bindings as you see here, and even if you don’t think you’ll use them today, you should follow these practices laid out here in case you change your mind. Using the namespace once, and putting it in a proper Java package namespace will clean up your code and improve the usability of MyBatis in the long term.

命名空间的作用有两个，一个是利用更长的完全限定名来将不同的语句隔离开来，同时也实现了你上面见到的接口绑定。就算你觉得暂时用不到接口绑定，你也应该遵循这里的规定，以防哪天你改变了主意。 长远来看，只要将命名空间置于合适的 Java 包命名空间之中，你的代码会变得更加整洁，也有利于你更方便地使用 MyBatis。

**Name Resolution:** To reduce the amount of typing, MyBatis uses the following name resolution rules for all named configuration elements, including statements, result maps, caches, etc.

命名解析：为了减少输入量，MyBatis 对所有的命名配置元素（包括语句，结果映射，缓存等）使用了如下的命名解析规则。

* Fully qualified names (e.g. “com.mypackage.MyMapper.selectAllThings”) are looked up directly and used if found.
* 完全限定名（比如 “com.mypackage.MyMapper.selectAllThings）将被直接用于查找及使用。
* Short names (e.g. “selectAllThings”) can be used to reference any unambiguous entry. However if there are two or more (e.g. “com.foo.selectAllThings and com.bar.selectAllThings”), then you will receive an error reporting that the short name is ambiguous and therefore must be fully qualified.
* 短名称（比如 “selectAllThings”）如果全局唯一也可以作为一个单独的引用。 如果不唯一，有两个或两个以上的相同名称（比如 “com.foo.selectAllThings” 和 “com.bar.selectAllThings”），那么使用时就会产生“短名称不唯一”的错误，这种情况下就必须使用完全限定名。

There's one more trick to Mapper classes like BlogMapper. Their mapped statements don't need to be mapped with XML at all. Instead they can use Java Annotations. For example, the XML above could be eliminated and replaced with:

对于像 BlogMapper 这样的映射器类来说，还有另一种方法来处理映射。 它们映射的语句可以不用 XML 来配置，而可以使用 Java 注解来配置。比如，上面的 XML 示例可被替换如下：

package org.mybatis.example;

public interface BlogMapper {

@Select("SELECT \* FROM blog WHERE id = #{id}")

Blog selectBlog(int id);

}

The annotations are a lot cleaner for simple statements, however, Java Annotations are both limited and messier for more complicated statements. Therefore, if you have to do anything complicated, you're better off with XML mapped statements.

使用注解来映射简单语句会使代码显得更加简洁，然而对于稍微复杂一点的语句，Java 注解就力不从心了，并且会显得更加混乱。 因此，如果你需要完成很复杂的事情，那么最好使用 XML 来映射语句。

It will be up to you and your project team to determine which is right for you, and how important it is to you that your mapped statements be defined in a consistent way. That said, you're never locked into a single approach. You can very easily migrate Annotation based Mapped Statements to XML and vice versa.

选择何种方式来配置映射，以及认为映射语句定义的一致性是否重要，这些完全取决于你和你的团队。 换句话说，永远不要拘泥于一种方式，你可以很轻松的在基于注解和 XML 的语句映射方式间自由移植和切换。

**Scope and Lifecycle**

作用域（Scope）和生命周期

It's very important to understand the various scopes and lifecycles classes we've discussed so far. Using them incorrectly can cause severe concurrency problems.

理解我们目前已经讨论过的不同作用域和生命周期类是至关重要的，因为错误的使用会导致非常严重的并发问题。

**NOTE Object lifecycle and Dependency Injection Frameworks**

提示 对象生命周期和依赖注入框架

Dependency Injection frameworks can create thread safe, transactional SqlSessions and mappers and inject them directly into your beans so you can just forget about their lifecycle. You may want to have a look at MyBatis-Spring or MyBatis-Guice sub-projects to know more about using MyBatis with DI frameworks.

依赖注入框架可以创建线程安全的、基于事务的 SqlSession 和映射器，并将它们直接注入到你的 bean 中，因此可以直接忽略它们的生命周期。 如果对如何通过依赖注入框架来使用 MyBatis 感兴趣，可以研究一下 MyBatis-Spring 或 MyBatis-Guice 两个子项目。

**SqlSessionFactoryBuilder**

This class can be instantiated, used and thrown away. There is no need to keep it around once you've created your SqlSessionFactory. Therefore the best scope for instances of SqlSessionFactoryBuilder is method scope (i.e. a local method variable). You can reuse the SqlSessionFactoryBuilder to build multiple SqlSessionFactory instances, but it's still best not to keep it around to ensure that all of the XML parsing resources are freed up for more important things.

这个类可以被实例化、使用和丢弃，一旦创建了 SqlSessionFactory，就不再需要它了。 因此 SqlSessionFactoryBuilder 实例的最佳作用域是方法作用域（也就是局部方法变量）。 你可以重用 SqlSessionFactoryBuilder 来创建多个 SqlSessionFactory 实例，但是最好还是不要让其一直存在，以保证所有的 XML 解析资源可以被释放给更重要的事情。

**SqlSessionFactory**

Once created, the SqlSessionFactory should exist for the duration of your application execution. There should be little or no reason to ever dispose of it or recreate it. It's a best practice to not rebuild the SqlSessionFactory multiple times in an application run. Doing so should be considered a “bad smell”. Therefore the best scope of SqlSessionFactory is application scope. This can be achieved a number of ways. The simplest is to use a Singleton pattern or Static Singleton pattern.

SqlSessionFactory 一旦被创建就应该在应用的运行期间一直存在，没有任何理由丢弃它或重新创建另一个实例。 使用 SqlSessionFactory 的最佳实践是在应用运行期间不要重复创建多次，多次重建 SqlSessionFactory 被视为一种代码“坏味道（bad smell）”。因此 SqlSessionFactory 的最佳作用域是应用作用域。 有很多方法可以做到，最简单的就是使用单例模式或者静态单例模式。

**SqlSession**

Each thread should have its own instance of SqlSession. Instances of SqlSession are not to be shared and are not thread safe. Therefore the best scope is request or method scope. Never keep references to a SqlSession instance in a static field or even an instance field of a class. Never keep references to a SqlSession in any sort of managed scope, such as HttpSession of the Servlet framework. If you're using a web framework of any sort, consider the SqlSession to follow a similar scope to that of an HTTP request. In other words, upon receiving an HTTP request, you can open a SqlSession, then upon returning the response, you can close it. Closing the session is very important. You should always ensure that it's closed within a finally block. The following is the standard pattern for ensuring that SqlSessions are closed:

每个线程都应该有它自己的 SqlSession 实例。SqlSession 的实例不是线程安全的，因此是不能被共享的，所以它的最佳的作用域是请求或方法作用域。 绝对不能将 SqlSession 实例的引用放在一个类的静态域，甚至一个类的实例变量也不行。 也绝不能将 SqlSession 实例的引用放在任何类型的托管作用域中，比如 Servlet 框架中的 HttpSession。 如果你现在正在使用一种 Web 框架，要考虑 SqlSession 放在一个和 HTTP 请求对象相似的作用域中。 换句话说，每次收到的 HTTP 请求，就可以打开一个 SqlSession，返回一个响应，就关闭它。 这个关闭操作是很重要的，你应该把这个关闭操作放到 finally 块中以确保每次都能执行关闭。 下面的示例就是一个确保 SqlSession 关闭的标准模式：

SqlSession session = sqlSessionFactory.openSession();

try {

// do work // 你的应用逻辑代码

} finally {

session.close();

}

Using this pattern consistently throughout your code will ensure that all database resources are properly closed.

在你的所有的代码中一致地使用这种模式来保证所有数据库资源都能被正确地关闭。

**Mapper Instances 映射器实例**

Mappers are interfaces that you create to bind to your mapped statements. Instances of the mapper interfaces are acquired from the SqlSession. As such, technically the broadest scope of any mapper instance is the same as the SqlSession from which they were requested. However, the best scope for mapper instances is method scope. That is, they should be requested within the method that they are used, and then be discarded. They do not need to be closed explicitly. While it's not a problem to keep them around throughout a request, similar to the SqlSession, you might find that managing too many resources at this level will quickly get out of hand. Keep it simple, keep Mappers in the method scope. The following example demonstrates this practice.

映射器是一些由你创建的、绑定你映射的语句的接口。映射器接口的实例是从 SqlSession 中获得的。因此从技术层面讲，任何映射器实例的最大作用域是和请求它们的 SqlSession 相同的。尽管如此，映射器实例的最佳作用域是方法作用域。 也就是说，映射器实例应该在调用它们的方法中被请求，用过之后即可丢弃。 并不需要显式地关闭映射器实例，尽管在整个请求作用域保持映射器实例也不会有什么问题，但是你很快会发现，像 SqlSession 一样，在这个作用域上管理太多的资源的话会难于控制。 为了避免这种复杂性，最好把映射器放在方法作用域内。下面的示例就展示了这个实践：

SqlSession session = sqlSessionFactory.openSession();

try {

BlogMapper mapper = session.getMapper(BlogMapper.class);

// do work 你的应用逻辑代码

} finally {

session.close();

}

**Configuration XML**

**Configuration 配置**

The MyBatis configuration contains settings and properties that have a dramatic effect on how MyBatis behaves. The high level structure of the document is as follows:

MyBatis 的配置文件包含了会深深影响 MyBatis 行为的设置和属性信息。 配置文档的顶层结构如下：

* configuration（配置）
  + properties（属性）
  + settings（设置）
  + typeAliases（类型别名）
  + typeHandlers（类型处理器）
  + objectFactory（对象工厂）
  + plugins（插件）
  + environments（环境配置）
    - environment（环境变量）
      * transactionManager（事务管理器）
      * dataSource（数据源）
  + databaseIdProvider（数据库厂商标识）
  + mappers（映射器）

**properties 属性**

These are externalizable, substitutable properties that can be configured in a typical Java Properties file instance, or passed in through sub-elements of the properties element. For example:

这些属性都是可外部配置且可动态替换的，既可以在典型的 Java 属性文件中配置，亦可通过 properties 元素的子元素来传递。例如：

<properties resource="org/mybatis/example/config.properties">

<property name="username" value="dev\_user"/>

<property name="password" value="F2Fa3!33TYyg"/>

</properties>

The properties can then be used throughout the configuration files to substitute values that need to be dynamically configured. For example:

然后其中的属性就可以在整个配置文件中被用来替换需要动态配置的属性值。比如:

<dataSource type="POOLED">

<property name="driver" value="${driver}"/>

<property name="url" value="${url}"/>

<property name="username" value="${username}"/>

<property name="password" value="${password}"/>

</dataSource>

The username and password in this example will be replaced by the values set in the properties elements. The driver and url properties would be replaced with values contained from the config.properties file. This provides a lot of options for configuration.

这个例子中的 username 和 password 将会由 properties 元素中设置的相应值来替换。 driver 和 url 属性将会由 config.properties 文件中对应的值来替换。这样就为配置提供了诸多灵活选择。

Properties can also be passed into the SqlSessionFactoryBuilder.build() methods. For example:

属性也可以被传递到 SqlSessionFactoryBuilder.build()方法中。例如：

SqlSessionFactory factory =

sqlSessionFactoryBuilder.build(reader, props);

// ... or ... ... 或者 ...

SqlSessionFactory factory =

new SqlSessionFactoryBuilder.build(reader, environment, props);

If a property exists in more than one of these places, MyBatis loads them in the following order:

如果属性在不只一个地方进行了配置，那么 MyBatis 将按照下面的顺序来加载：

* Properties specified in the body of the properties element are read first,
* 在 properties 元素体内指定的属性首先被读取。
* Properties loaded from the classpath resource or url attributes of the properties element are read second, and override any duplicate properties already specified,
* 然后根据 properties 元素中的 resource 属性读取类路径下属性文件或根据 url 属性指定的路径读取属性文件，并覆盖已读取的同名属性。
* Properties passed as a method parameter are read last, and override any duplicate properties that may have been loaded from the properties body and the resource/url attributes.
* 最后读取作为方法参数传递的属性，并覆盖已读取的同名属性。

Thus, the highest priority properties are those passed in as a method parameter, followed by resource/url attributes and finally the properties specified in the body of the properties element.

因此，通过方法参数传递的属性具有最高优先级，resource/url 属性中指定的配置文件次之，最低优先级的是 properties 属性中指定的属性。

Since the MyBatis 3.4.2, your can specify a default value into placeholder as follow:

从 MyBatis 3.4.2 开始，你可以为占位符指定一个默认值。例如：

<dataSource type="POOLED">

<!-- ... -->

<property name="username" value="${username:ut\_user}"/> <!-- If 'username' property not present, username become 'ut\_user' 如果属性 'username' 没有被配置，'username' 属性的值将为 'ut\_user' -->

</dataSource>

This feature is disabled by default. If you specify a default value into placeholder, you should be enable this feature by adding a special property as follow:

这个特性默认是关闭的。如果你想为占位符指定一个默认值， 你应该添加一个指定的属性来开启这个特性。例如：

<properties resource="org/mybatis/example/config.properties">

<!-- ... -->

<property name="org.apache.ibatis.parsing.PropertyParser.enable-default-value" value="true"/> <!-- Enable this feature 启用默认值特性 -->

</properties>

NOTE Also If you are used already the ":" as property key(e.g. db:username) or you are used already the ternary operator of OGNL expression(e.g. ${tableName != null ? tableName : 'global\_constants'}) on your sql definition, you should be change the character that separate key and default value by adding a special property as follow:

提示 如果你已经使用 ":" 作为属性的键（如：db:username） ，或者你已经在 SQL 定义中使用 OGNL 表达式的三元运算符（如： ${tableName != null ? tableName : 'global\_constants'}），你应该通过设置特定的属性来修改分隔键名和默认值的字符。例如：

<properties resource="org/mybatis/example/config.properties">

<!-- ... -->

<property name="org.apache.ibatis.parsing.PropertyParser.default-value-separator" value="?:"/> <!-- Change default value of separator 修改默认值的分隔符 -->

</properties>

<dataSource type="POOLED">

<!-- ... -->

<property name="username" value="${db:username?:ut\_user}"/>

</dataSource>

**settings 设置**

These are extremely important tweaks that modify the way that MyBatis behaves at runtime. The following table describes the settings, their meanings and their default values.

这是 MyBatis 中极为重要的调整设置，它们会改变 MyBatis 的运行时行为。 下表描述了设置中各项的意图、默认值等。

* **cacheEnabled** Globally enables or disables any caches configured in any mapper under this configuration.

全局地开启或关闭配置文件中的所有映射器已经配置的任何缓存。

* + **Valid Values:** true | false
  + **Default:** true
* **lazyLoadingEnabled** Globally enables or disables lazy loading. When enabled, all relations will be lazily loaded. This value can be superseded for an specific relation by using the fetchType attribute on it.

延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。 特定关联关系中可通过设置 fetchType 属性来覆盖该项的开关状态。

* + **Valid Values:** true | false
  + **default:** false
* **aggressiveLazyLoading** When enabled, any method call will load all the lazy properties of the object. Otherwise, each property is loaded on demand (see also lazyLoadTriggerMethods).

当开启时，任何方法的调用都会加载该对象的所有属性。 否则，每个属性会按需加载（参考 lazyLoadTriggerMethods)。

* + **Valid Values:** true | false
  + **Default:** false (true in ≤3.4.1)
* **multipleResultSetsEnabled** Allows or disallows multiple ResultSets to be returned from a single statement (compatible driver required).

是否允许单一语句返回多结果集（需要驱动支持）。

* + **Valid Values:** true | false
  + **Default:** true
* **useColumnLabel** Uses the column label instead of the column name. Different drivers behave differently in this respect. Refer to the driver documentation, or test out both modes to determine how your driver behaves.

使用列标签代替列名。不同的驱动在这方面会有不同的表现，具体可参考相关驱动文档或通过测试这两种不同的模式来观察所用驱动的结果。

* + **Valid Values:** true | false
  + **Default:** true
* **useGeneratedKeys** Allows JDBC support for generated keys. A compatible driver is required. This setting forces generated keys to be used if set to true, as some drivers deny compatibility but still work (e.g. Derby).

允许 JDBC 支持自动生成主键，需要驱动支持。 如果设置为 true 则这个设置强制使用自动生成主键，尽管一些驱动不能支持但仍可正常工作（比如 Derby）。

* + **Valid Values:** true | false
  + **Default:** False
* **autoMappingBehavior** Specifies if and how MyBatis should automatically map columns to fields/properties. NONE disables auto-mapping. PARTIAL will only auto-map results with no nested result mappings defined inside. FULL will auto-map result mappings of any complexity (containing nested or otherwise).

指定 MyBatis 应如何自动映射列到字段或属性。 NONE 表示取消自动映射；PARTIAL 只会自动映射没有定义嵌套结果集映射的结果集。 FULL 会自动映射任意复杂的结果集（无论是否嵌套）。

* + **Valid Values:** NONE, PARTIAL, FULL
  + **Default:** PARTIAL
* **autoMappingUnknownColumnBehavior** Specify the behavior when detects an unknown column (or unknown property type) of automatic mapping target.NONE: Do nothing，WARNING: Output warning log (The log level of 'org.apache.ibatis.session.AutoMappingUnknownColumnBehavior' must be set to WARN) ，FAILING: Fail mapping (Throw SqlSessionException)

指定发现自动映射目标未知列（或者未知属性类型）的行为。NONE: 不做任何反应;WARNING: 输出提醒日志 ;('org.apache.ibatis.session.AutoMappingUnknownColumnBehavior' 的日志等级必须设置为 WARN);FAILING: 映射失败 (抛出 SqlSessionException)

* + **Valid Values:** NONE, WARNING, FAILING
  + **Default:** NONE
* **defaultExecutorType** Configures the default executor. SIMPLE executor does nothing special. REUSE executor reuses prepared statements. BATCH executor reuses statements and batches updates.

配置默认的执行器。SIMPLE 就是普通的执行器；REUSE 执行器会重用预处理语句（prepared statements）； BATCH 执行器将重用语句并执行批量更新。

* + **Valid Values:** SIMPLE REUSE BATCH
  + **Default:** SIMPLE
* **defaultStatementTimeout** Sets the number of seconds the driver will wait for a response from the database.

设置超时时间，它决定驱动等待数据库响应的秒数。

* + **Valid Values:** Any positive integer 任意正整数
  + **Default:** Not Set (null)
* **defaultFetchSize** Sets the driver a hint as to control fetching size for return results. This parameter value can be override by a query setting.

为驱动的结果集获取数量（fetchSize）设置一个提示值。此参数只可以在查询设置中被覆盖。

* + **Valid Values:** Any positive integer 任意正整数
  + **Default:** Not Set (null)
* **safeRowBoundsEnabled** Allows using RowBounds on nested statements. If allow, set the false.

允许在嵌套语句中使用分页（RowBounds）。如果允许使用则设置为 false。

* + **Valid Values:** true | false
  + **Default:** False
* **safeResultHandlerEnabled** Allows using ResultHandler on nested statements. If allow, set the false.

允许在嵌套语句中使用分页（ResultHandler）。如果允许使用则设置为 false。

* + **Valid Values:** true | false
  + **Default:** True
* **mapUnderscoreToCamelCase** Enables automatic mapping from classic database column names A\_COLUMN to camel case classic Java property names aColumn.

是否开启自动驼峰命名规则（camel case）映射，即从经典数据库列名 A\_COLUMN 到经典 Java 属性名 aColumn 的类似映射。

* + **Valid Values:** true | false
  + **Default:** False
* **localCacheScope** MyBatis uses local cache to prevent circular references and speed up repeated nested queries. By default (SESSION) all queries executed during a session are cached. If localCacheScope=STATEMENT local session will be used just for statement execution, no data will be shared between two different calls to the same SqlSession.

MyBatis 利用本地缓存机制（Local Cache）防止循环引用（circular references）和加速重复嵌套查询。 默认值为 SESSION，这种情况下会缓存一个会话中执行的所有查询。 若设置值为 STATEMENT，本地会话仅用在语句执行上，对相同 SqlSession 的不同调用将不会共享数据。

* + **Valid Values:** SESSION | STATEMENT
  + **Default:** SESSION
* **jdbcTypeForNull** Specifies the JDBC type for null values when no specific JDBC type was provided for the parameter. Some drivers require specifying the column JDBC type but others work with generic values like NULL, VARCHAR or OTHER.

当没有为参数提供特定的 JDBC 类型时，为空值指定 JDBC 类型。 某些驱动需要指定列的 JDBC 类型，多数情况直接用一般类型即可，比如 NULL、VARCHAR 或 OTHER。

* + **Valid Values:** JdbcType enumeration. Most common are: NULL, VARCHAR and OTHER JdbcType 常量，常用值：NULL, VARCHAR 或 OTHER。
  + **Default:** OTHER
* **lazyLoadTriggerMethods** Specifies which Object's methods trigger a lazy load

指定哪个对象的方法触发一次延迟加载。

* + **Valid Values:** A method name list separated by commas 用逗号分隔的方法列表。
  + **Default:** equals,clone,hashCode,toString
* **defaultScriptingLanguage** Specifies the language used by default for dynamic SQL generation.

指定动态 SQL 生成的默认语言。

* + **Valid Values:** A type alias or fully qualified class name. 一个类型别名或完全限定类名
  + **Default:** org.apache.ibatis.scripting.xmltags.XMLLanguageDriver
* **defaultEnumTypeHandler** Specifies the TypeHandler used by default for Enum. (Since: 3.4.5)

指定 Enum 使用的默认 TypeHandler 。（新增于 3.4.5）

* + **Valid Values:** A type alias or fully qualified class name. 一个类型别名或完全限定类名
  + **Default:** org.apache.ibatis.type.EnumTypeHandler
* **callSettersOnNulls** Specifies if setters or map's put method will be called when a retrieved value is null. It is useful when you rely on Map.keySet() or null value initialization. Note primitives such as (int,boolean,etc.) will not be set to null.

指定当结果集中值为 null 的时候是否调用映射对象的 setter（map 对象时为 put）方法，这在依赖于 Map.keySet() 或 null 值初始化的时候比较有用。注意基本类型（int、boolean 等）是不能设置成 null 的。

* + **Valid Values:** true | false
  + **Default:** false
* **returnInstanceForEmptyRow** MyBatis, by default, returns null when all the columns of a returned row are NULL. When this setting is enabled, MyBatis returns an empty instance instead. Note that it is also applied to nested results (i.e. collectioin and association). Since: 3.4.2

当返回行的所有列都是空时，MyBatis默认返回 null。 当开启这个设置时，MyBatis会返回一个空实例。 请注意，它也适用于嵌套的结果集 （如集合或关联）。（新增于 3.4.2）

* + **Valid Values:** true | false
  + **Default:** false
* **logPrefix** Specifies the prefix string that MyBatis will add to the logger names. 指定 MyBatis 增加到日志名称的前缀。
  + **Valid Values:** Any String 任何字符串
  + **Default:** Not set
* **logImpl** Specifies which logging implementation MyBatis should use. If this setting is not present logging implementation will be autodiscovered.

指定 MyBatis 所用日志的具体实现，未指定时将自动查找。

* + **Valid Values:** SLF4J | LOG4J | LOG4J2 | JDK\_LOGGING | COMMONS\_LOGGING | STDOUT\_LOGGING | NO\_LOGGING
  + **Default:** Not set
* **proxyFactory** Specifies the proxy tool that MyBatis will use for creating lazy loading capable objects.

指定 Mybatis 创建具有延迟加载能力的对象所用到的代理工具。

* + **Valid Values:** CGLIB | JAVASSIST
  + **Default:** JAVASSIST (MyBatis 3.3 or above)
* **vfsImpl** Specifies VFS implementations

指定 VFS 的实现

* + **Valid Values:** Fully qualified class names of custom VFS implementation separated by commas. 自定义 VFS 的实现的类全限定名，以逗号分隔。
  + **Default:** Not set 未设置
* **useActualParamName** Allow referencing statement parameters by their actual names declared in the method signature. To use this feature, your project must be compiled in Java 8 with -parameters option. (Since: 3.4.1)

允许使用方法签名中的名称作为语句参数名称。 为了使用该特性，你的项目必须采用 Java 8 编译，并且加上 -parameters 选项。（新增于 3.4.1）

* + **Valid Values:** true | false
  + **Default:** true
* **configurationFactory** Specifies the class that provides an instance of Configuration. The returned Configuration instance is used to load lazy properties of deserialized objects. This class must have a method with a signature static Configuration getConfiguration(). (Since: 3.2.3)

指定一个提供 Configuration 实例的类。 这个被返回的 Configuration 实例用来加载被反序列化对象的延迟加载属性值。 这个类必须包含一个签名为static Configuration getConfiguration() 的方法。（新增于 3.2.3）

* + **Valid Values:** A type alias or fully qualified class name. 类型别名或者全类名.
  + **Default:** Not set 未设置

An example of the settings element fully configured is as follows:

一个配置完整的 settings 元素的示例如下：

<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="autoMappingBehavior" value="PARTIAL"/>

<setting name="autoMappingUnknownColumnBehavior" value="WARNING"/>

<setting name="defaultExecutorType" value="SIMPLE"/>

<setting name="defaultStatementTimeout" value="25"/>

<setting name="defaultFetchSize" value="100"/>

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

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

<setting name="localCacheScope" value="SESSION"/>

<setting name="jdbcTypeForNull" value="OTHER"/>

<setting name="lazyLoadTriggerMethods"

value="equals,clone,hashCode,toString"/>

</settings>

**typeAliases 类型别名**

A type alias is simply a shorter name for a Java type. It's only relevant to the XML configuration and simply exists to reduce redundant typing of fully qualified classnames. For example:

类型别名是为 Java 类型设置一个短的名字。 它只和 XML 配置有关，存在的意义仅在于用来减少类完全限定名的冗余。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>

With this configuration, Blog can now be used anywhere that domain.blog.Blog could be.

当这样配置时，Blog 可以用在任何使用 domain.blog.Blog 的地方。

You can also specify a package where MyBatis will search for beans. For example:

也可以指定一个包名，MyBatis 会在包名下面搜索需要的 Java Bean，比如：

<typeAliases>

<package name="domain.blog"/>

</typeAliases>

Each bean found in domain.blog , if no annotation is found, will be registered as an alias using uncapitalized non-qualified class name of the bean. That is domain.blog.Author will be registered as author. If the @Alias annotation is found its value will be used as an alias. See the example below:

每一个在包 domain.blog 中的 Java Bean，在没有注解的情况下，会使用 Bean 的首字母小写的非限定类名来作为它的别名。 比如 domain.blog.Author 的别名为 author；若有注解，则别名为其注解值。见下面的例子：

@Alias("author")

public class Author {

...

}

There are many built-in type aliases for common Java types. They are all case insensitive, note the special handling of primitives due to the overloaded names.

这是一些为常见的 Java 类型内建的相应的类型别名。它们都是不区分大小写的，注意对基本类型名称重复采取的特殊命名风格。

**typeHandlers 类型处理器**

Whenever MyBatis sets a parameter on a PreparedStatement or retrieves a value from a ResultSet, a TypeHandler is used to retrieve the value in a means appropriate to the Java type. The following table describes the default TypeHandlers.

无论是 MyBatis 在预处理语句（PreparedStatement）中设置一个参数时，还是从结果集中取出一个值时， 都会用类型处理器将获取的值以合适的方式转换成 Java 类型。下表描述了一些默认的类型处理器。

NOTE Since version 3.4.5, The MyBatis has been supported JSR-310(Date and Time API) by default.

提示 从 3.4.5 开始，MyBatis 默认支持 JSR-310（日期和时间 API） 。

You can override the type handlers or create your own to deal with unsupported or non-standard types. To do so, implement the interface org.apache.ibatis.type.TypeHandler or extend the convenience class org.apache.ibatis.type.BaseTypeHandler and optionally map it to a JDBC type. For example:

你可以重写类型处理器或创建你自己的类型处理器来处理不支持的或非标准的类型。 具体做法为：实现 org.apache.ibatis.type.TypeHandler 接口， 或继承一个很便利的类 org.apache.ibatis.type.BaseTypeHandler， 然后可以选择性地将它映射到一个 JDBC 类型。比如：

// ExampleTypeHandler.java

@MappedJdbcTypes(JdbcType.VARCHAR)

public class ExampleTypeHandler extends BaseTypeHandler<String> {

@Override

public void setNonNullParameter(PreparedStatement ps, int i,

String parameter, JdbcType jdbcType) throws SQLException {

ps.setString(i, parameter);

}

@Override

public String getNullableResult(ResultSet rs, String columnName)

throws SQLException {

return rs.getString(columnName);

}

@Override

public String getNullableResult(ResultSet rs, int columnIndex)

throws SQLException {

return rs.getString(columnIndex);

}

@Override

public String getNullableResult(CallableStatement cs, int columnIndex)

throws SQLException {

return cs.getString(columnIndex);

}

}

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

<typeHandlers>

<typeHandler handler="org.mybatis.example.ExampleTypeHandler"/>

</typeHandlers>

Using such a TypeHandler would override the existing type handler for Java String properties and VARCHAR parameters and results. Note that MyBatis does not introspect upon the database metadata to determine the type, so you must specify that it’s a VARCHAR field in the parameter and result mappings to hook in the correct type handler. This is due to the fact that MyBatis is unaware of the data type until the statement is executed.

使用上述的类型处理器将会覆盖已经存在的处理 Java 的 String 类型属性和 VARCHAR 参数及结果的类型处理器。 要注意 MyBatis 不会通过窥探数据库元信息来决定使用哪种类型，所以你必须在参数和结果映射中指明那是 VARCHAR 类型的字段， 以使其能够绑定到正确的类型处理器上。这是因为 MyBatis 直到语句被执行时才清楚数据类型。

MyBatis will know the the Java type that you want to handle with this TypeHandler by introspecting its generic type, but you can override this behavior by two means:

通过类型处理器的泛型，MyBatis 可以得知该类型处理器处理的 Java 类型，不过这种行为可以通过两种方法改变：

* Adding a javaType attribute to the typeHandler element (for example: javaType="String")
* 在类型处理器的配置元素（typeHandler 元素）上增加一个 javaType 属性（比如：javaType="String"）；
* Adding a @MappedTypes annotation to your TypeHandler class specifying the list of java types to associate it with. This annotation will be ignored if the javaType attribute as also been specified.
* 在类型处理器的类上（TypeHandler class）增加一个 @MappedTypes 注解来指定与其关联的 Java 类型列表。 如果在 javaType 属性中也同时指定，则注解方式将被忽略。

Associated JDBC type can be specified by two means:

可以通过两种方式来指定被关联的 JDBC 类型：

* Adding a jdbcType attribute to the typeHandler element (for example: jdbcType="VARCHAR").
* 在类型处理器的配置元素上增加一个 jdbcType 属性（比如：jdbcType="VARCHAR"）；
* Adding a @MappedJdbcTypes annotation to your TypeHandler class specifying the list of JDBC types to associate it with. This annotation will be ignored if the jdbcType attribute as also been specified.
* 在类型处理器的类上增加一个 @MappedJdbcTypes 注解来指定与其关联的 JDBC 类型列表。 如果在 jdbcType 属性中也同时指定，则注解方式将被忽略。

When deciding which TypeHandler to use in a ResultMap, the Java type is known (from the result type), but the JDBC type is unknown. MyBatis therefore uses the combination javaType=[TheJavaType], jdbcType=null to choose a TypeHandler. This means that using a @MappedJdbcTypes annotation restricts the scope of a TypeHandler and makes it unavailable for use in ResultMaps unless explicity set. To make a TypeHandler available for use in a ResultMap, set includeNullJdbcType=true on the @MappedJdbcTypes annotation. Since Mybatis 3.4.0 however, if a single TypeHandler is registered to handle a Java type, it will be used by default in ResultMaps using this Java type (i.e. even without includeNullJdbcType=true).

当在 ResultMap 中决定使用哪种类型处理器时，此时 Java 类型是已知的（从结果类型中获得），但是 JDBC 类型是未知的。 因此 Mybatis 使用 javaType=[Java 类型], jdbcType=null 的组合来选择一个类型处理器。 这意味着使用 @MappedJdbcTypes 注解可以限制类型处理器的范围，同时除非显式的设置，否则类型处理器在 ResultMap 中将是无效的。 如果希望在 ResultMap 中使用类型处理器，那么设置 @MappedJdbcTypes 注解的 includeNullJdbcType=true 即可。 然而从 Mybatis 3.4.0 开始，如果只有一个注册的类型处理器来处理 Java 类型，那么它将是 ResultMap 使用 Java 类型时的默认值（即使没有 includeNullJdbcType=true）。

And finally you can let MyBatis search for your TypeHandlers:

最后，可以让 MyBatis 为你查找类型处理器：

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

<typeHandlers>

<package name="org.mybatis.example"/>

</typeHandlers>

Note that when using the autodiscovery feature JDBC types can only be specified with annotations.

注意在使用自动发现功能的时候，只能通过注解方式来指定 JDBC 的类型。

You can create a generic TypeHandler that is able to handle more than one class. For that purpose add a constructor that receives the class as a parameter and MyBatis will pass the actual class when constructing the TypeHandler.

你可以创建一个能够处理多个类的泛型类型处理器。为了使用泛型类型处理器， 需要增加一个接受该类的 class 作为参数的构造器，这样在构造一个类型处理器的时候 MyBatis 就会传入一个具体的类。

//GenericTypeHandler.java

public class GenericTypeHandler<E extends MyObject> extends BaseTypeHandler<E> {

private Class<E> type;

public GenericTypeHandler(Class<E> type) {

if (type == null) throw new IllegalArgumentException("Type argument cannot be null");

this.type = type;

}

...

EnumTypeHandler and EnumOrdinalTypeHandler are generic TypeHandlers. We will learn about them in the following section.

EnumTypeHandler 和 EnumOrdinalTypeHandler 都是泛型类型处理器，我们将会在接下来的部分详细探讨。

**Handling Enums 处理枚举类型**

If you want to map an Enum, you'll need to use either EnumTypeHandler or EnumOrdinalTypeHandler.

若想映射枚举类型 Enum，则需要从 EnumTypeHandler 或者 EnumOrdinalTypeHandler 中选一个来使用。

For example, let's say that we need to store the rounding mode that should be used with some number if it needs to be rounded. By default, MyBatis uses EnumTypeHandler to convert the Enum values to their names.

比如说我们想存储取近似值时用到的舍入模式。默认情况下，MyBatis 会利用 EnumTypeHandler 来把 Enum 值转换成对应的名字。

Note EnumTypeHandler is special in the sense that unlike other handlers, it does not handle just one specific class, but any class that extends Enum

注意 EnumTypeHandler 在某种意义上来说是比较特别的，其他的处理器只针对某个特定的类，而它不同，它会处理任意继承了 Enum 的类。

However, we may not want to store names. Our DBA may insist on an integer code instead. That's just as easy: add EnumOrdinalTypeHandler to the typeHandlers in your config file, and now each RoundingMode will be mapped to an integer using its ordinal value.

不过，我们可能不想存储名字，相反我们的 DBA 会坚持使用整形值代码。那也一样轻而易举： 在配置文件中把 EnumOrdinalTypeHandler 加到 typeHandlers 中即可， 这样每个 RoundingMode 将通过他们的序数值来映射成对应的整形数值。

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

<typeHandlers>

<typeHandler handler="org.apache.ibatis.type.EnumOrdinalTypeHandler"

javaType="java.math.RoundingMode"/>

</typeHandlers>

But what if you want to map the same Enum to a string in one place and to integer in another?

但是怎样能将同样的 Enum 既映射成字符串又映射成整形呢？

The auto-mapper will automatically use EnumOrdinalTypeHandler, so if we want to go back to using plain old ordinary EnumTypeHandler, we have to tell it, by explicitly setting the type handler to use for those SQL statements.

自动映射器（auto-mapper）会自动地选用 EnumOrdinalTypeHandler 来处理， 所以如果我们想用普通的 EnumTypeHandler，就必须要显式地为那些 SQL 语句设置要使用的类型处理器。

(Mapper files aren't covered until the next section, so if this is your first time reading through the documentation, you may want to skip this for now and come back to it later.)

（下一节才开始介绍映射器文件，如果你是首次阅读该文档，你可能需要先跳过这里，过会再来看。）

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="org.apache.ibatis.submitted.rounding.Mapper">

<resultMap type="org.apache.ibatis.submitted.rounding.User" id="usermap">

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

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

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

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

</resultMap>

<select id="getUser" resultMap="usermap">

select \* from users

</select>

<insert id="insert">

insert into users (id, name, funkyNumber, roundingMode) values (

#{id}, #{name}, #{funkyNumber}, #{roundingMode}

)

</insert>

<resultMap type="org.apache.ibatis.submitted.rounding.User" id="usermap2">

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

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

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

<result column="roundingMode" property="roundingMode"

typeHandler="org.apache.ibatis.type.EnumTypeHandler"/>

</resultMap>

<select id="getUser2" resultMap="usermap2">

select \* from users2

</select>

<insert id="insert2">

insert into users2 (id, name, funkyNumber, roundingMode) values (

#{id}, #{name}, #{funkyNumber}, #{roundingMode, typeHandler=org.apache.ibatis.type.EnumTypeHandler}

)

</insert>

</mapper>

Note that this forces us to use a resultMap instead of a resultType in our select statements.

注意，这里的 select 语句强制使用 resultMap 来代替 resultType。

**objectFactory 对象工厂**

Each time MyBatis creates a new instance of a result object, it uses an ObjectFactory instance to do so. The default ObjectFactory does little more than instantiate the target class with a default constructor, or a parameterized constructor if parameter mappings exist. If you want to override the default behaviour of the ObjectFactory, you can create your own. For example:

MyBatis 每次创建结果对象的新实例时，它都会使用一个对象工厂（ObjectFactory）实例来完成。 默认的对象工厂需要做的仅仅是实例化目标类，要么通过默认构造方法，要么在参数映射存在的时候通过参数构造方法来实例化。 如果想覆盖对象工厂的默认行为，则可以通过创建自己的对象工厂来实现。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, constructorArgs);

}

public void setProperties(Properties properties) {

super.setProperties(properties);

}

public <T> boolean isCollection(Class<T> type) {

return Collection.class.isAssignableFrom(type);

}}

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

<objectFactory type="org.mybatis.example.ExampleObjectFactory">

<property name="someProperty" value="100"/>

</objectFactory>

The ObjectFactory interface is very simple. It contains two create methods, one to deal with the default constructor, and the other to deal with parameterized constructors. Finally, the setProperties method can be used to configure the ObjectFactory. Properties defined within the body of the objectFactory element will be passed to the setProperties method after initialization of your ObjectFactory instance.

ObjectFactory 接口很简单，它包含两个创建用的方法，一个是处理默认构造方法的，另外一个是处理带参数的构造方法的。 最后，setProperties 方法可以被用来配置 ObjectFactory，在初始化你的 ObjectFactory 实例后， objectFactory 元素体中定义的属性会被传递给 setProperties 方法。

**plugins 插件**

MyBatis allows you to intercept calls to at certain points within the execution of a mapped statement. By default, MyBatis allows plug-ins to intercept method calls of:

MyBatis 允许你在已映射语句执行过程中的某一点进行拦截调用。默认情况下，MyBatis 允许使用插件来拦截的方法调用包括：

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

The details of these classes methods can be discovered by looking at the full method signature of each, and the source code which is available with each MyBatis release. You should understand the behaviour of the method you’re overriding, assuming you’re doing something more than just monitoring calls. If you attempt to modify or override the behaviour of a given method, you’re likely to break the core of MyBatis. These are low level classes and methods, so use plug-ins with caution.

这些类中方法的细节可以通过查看每个方法的签名来发现，或者直接查看 MyBatis 发行包中的源代码。 如果你想做的不仅仅是监控方法的调用，那么你最好相当了解要重写的方法的行为。 因为如果在试图修改或重写已有方法的行为的时候，你很可能在破坏 MyBatis 的核心模块。 这些都是更低层的类和方法，所以使用插件的时候要特别当心。

Using plug-ins is pretty simple given the power they provide. Simply implement the Interceptor interface, being sure to specify the signatures you want to intercept.

通过 MyBatis 提供的强大机制，使用插件是非常简单的，只需实现 Interceptor 接口，并指定想要拦截的方法签名即可。

// 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) {

}

}

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

<plugins>

<plugin interceptor="org.mybatis.example.ExamplePlugin">

<property name="someProperty" value="100"/>

</plugin>

</plugins>

The plug-in above will intercept all calls to the "update" method on the Executor instance, which is an internal object responsible for the low level execution of mapped statements.

上面的插件将会拦截在 Executor 实例中所有的 “update” 方法调用， 这里的 Executor 是负责执行低层映射语句的内部对象。

NOTE Overriding the Configuration Class

提示 覆盖配置类

In addition to modifying core MyBatis behaviour with plugins, you can also override the Configuration class entirely. Simply extend it and override any methods inside, and pass it into the call to the SqlSessionFactoryBuilder.build(myConfig) method. Again though, this could have a severe impact on the behaviour of MyBatis, so use caution.

除了用插件来修改 MyBatis 核心行为之外，还可以通过完全覆盖配置类来达到目的。只需继承后覆盖其中的每个方法，再把它传递到 SqlSessionFactoryBuilder.build(myConfig) 方法即可。再次重申，这可能会严重影响 MyBatis 的行为，务请慎之又慎。

**environments 环境配置**

MyBatis can be configured with multiple environments. This helps you to apply your SQL Maps to multiple databases for any number of reasons. For example, you might have a different configuration for your Development, Test and Production environments. Or, you may have multiple production databases that share the same schema, and you’d like to use the same SQL maps for both. There are many use cases.

MyBatis 可以配置成适应多种环境，这种机制有助于将 SQL 映射应用于多种数据库之中， 现实情况下有多种理由需要这么做。例如，开发、测试和生产环境需要有不同的配置；或者想在具有相同 Schema 的多个生产数据库中 使用相同的 SQL 映射。有许多类似的使用场景。

**One important thing to remember though: While you can configure multiple environments, you can only choose ONE per SqlSessionFactory instance.**

不过要记住：尽管可以配置多个环境，但每个 SqlSessionFactory 实例只能选择一种环境。

So if you want to connect to two databases, you need to create two instances of SqlSessionFactory, one for each. For three databases, you’d need three instances, and so on. It’s really easy to remember:

所以，如果你想连接两个数据库，就需要创建两个 SqlSessionFactory 实例，每个数据库对应一个。而如果是三个数据库，就需要三个实例，依此类推，记起来很简单：

* **One SqlSessionFactory instance per database** 每个数据库对应一个 SqlSessionFactory 实例

To specify which environment to build, you simply pass it to the SqlSessionFactoryBuilder as an optional parameter. The two signatures that accept the environment are:

为了指定创建哪种环境，只要将它作为可选的参数传递给 SqlSessionFactoryBuilder 即可。可以接受环境配置的两个方法签名是：

SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, environment);

SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, environment, properties);

If the environment is omitted, then the default environment is loaded, as follows:

如果忽略了环境参数，那么默认环境将会被加载，如下所示：

SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader);

SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, properties);

The environments element defines how the environment is configured.

环境元素定义了如何配置环境。

<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>

Notice the key sections here:

注意这里的关键点:

* The default Environment ID (e.g. default="development"). 默认使用的环境 ID（比如：default="development"）。
* The Environment ID for each environment defined (e.g. id="development"). 每个 environment 元素定义的环境 ID（比如：id="development"）。
* The TransactionManager configuration (e.g. type="JDBC") 事务管理器的配置（比如：type="JDBC"）。
* The DataSource configuration (e.g. type="POOLED") 数据源的配置（比如：type="POOLED"）。

The default environment and the environment IDs are self explanatory. Name them whatever you like, just make sure the default matches one of them.

默认的环境和环境 ID 是自解释的，因此一目了然。 你可以对环境随意命名，但一定要保证默认的环境 ID 要匹配其中一个环境 ID。

**transactionManager** 事务管理器

There are two TransactionManager types (i.e. type="[JDBC|MANAGED]") that are included with MyBatis:

在 MyBatis 中有两种类型的事务管理器（也就是 type=”[JDBC|MANAGED]”）：

* JDBC – This configuration simply makes use of the JDBC commit and rollback facilities directly. It relies on the connection retrieved from the dataSource to manage the scope of the transaction.
* JDBC – 这个配置就是直接使用了 JDBC 的提交和回滚设置，它依赖于从数据源得到的连接来管理事务作用域。
* MANAGED – This configuration simply does almost nothing. It never commits, or rolls back a connection. Instead, it lets the container manage the full lifecycle of the transaction (e.g. a JEE Application Server context). By default it does close the connection. However, some containers don’t expect this, and thus if you need to stop it from closing the connection, set the "closeConnection" property to false. For example:
* MANAGED – 这个配置几乎没做什么。它从来不提交或回滚一个连接，而是让容器来管理事务的整个生命周期（比如 JEE 应用服务器的上下文）。 默认情况下它会关闭连接，然而一些容器并不希望这样，因此需要将 closeConnection 属性设置为 false 来阻止它默认的关闭行为。例如:

<transactionManager type="MANAGED">

<property name="closeConnection" value="false"/>

</transactionManager>

**NOTE** If you are planning to use MyBatis with Spring there is no need to configure any TransactionManager because the Spring module will set its own one overriding any previously set configuration.

提示如果你正在使用 Spring + MyBatis，则没有必要配置事务管理器， 因为 Spring 模块会使用自带的管理器来覆盖前面的配置。

Neither of these TransactionManager types require any properties. However, they are both Type Aliases, so in other words, instead of using them, you could put your own fully qualified class name or Type Alias that refers to your own implementation of the TransactionFactory interface.

这两种事务管理器类型都不需要设置任何属性。它们其实是类型别名，换句话说，你可以使用 TransactionFactory 接口的实现类的完全限定名或类型别名代替它们。

public interface TransactionFactory {

void setProperties(Properties props);

Transaction newTransaction(Connection conn);

Transaction newTransaction(DataSource dataSource, TransactionIsolationLevel level, boolean autoCommit);

}

Any properties configured in the XML will be passed to the setProperties() method after instantiation. Your implementation would also need to create a Transaction implementation, which is also a very simple interface:

任何在 XML 中配置的属性在实例化之后将会被传递给 setProperties() 方法。你也需要创建一个 Transaction 接口的实现类，这个接口也很简单：

public interface Transaction {

Connection getConnection() throws SQLException;

void commit() throws SQLException;

void rollback() throws SQLException;

void close() throws SQLException;

Integer getTimeout() throws SQLException;

}

Using these two interfaces, you can completely customize how MyBatis deals with Transactions.

使用这两个接口，你可以完全自定义 MyBatis 对事务的处理。

**dataSource** 数据源

The dataSource element configures the source of JDBC Connection objects using the standard JDBC DataSource interface.

dataSource 元素使用标准的 JDBC 数据源接口来配置 JDBC 连接对象的资源。

* Most MyBatis applications will configure a dataSource as in the example. However, it’s not required. Realize though, that to facilitate Lazy Loading, this dataSource is required.

许多 MyBatis 的应用程序会按示例中的例子来配置数据源。虽然这是可选的，但为了使用延迟加载，数据源是必须配置的。

There are three build-in dataSource types (i.e. type=[UNPOOLED|POOLED|JNDI]"):

有三种内建的数据源类型（也就是 type=”[UNPOOLED|POOLED|JNDI]”）：

**UNPOOLED** – This implementation of DataSource simply opens and closes a connection each time it is requested. While it’s a bit slower, this is a good choice for simple applications that do not require the performance of immediately available connections. Different databases are also different in this performance area, so for some it may be less important to pool and this configuration will be ideal. The UNPOOLED DataSource is configured with only five properties:

UNPOOLED– 这个数据源的实现只是每次被请求时打开和关闭连接。虽然有点慢，但对于在数据库连接可用性方面没有太高要求的简单应用程序来说，是一个很好的选择。 不同的数据库在性能方面的表现也是不一样的，对于某些数据库来说，使用连接池并不重要，这个配置就很适合这种情形。UNPOOLED 类型的数据源仅仅需要配置以下 5 种属性：

* driver – This is the fully qualified Java class of the JDBC driver (NOT of the DataSource class if your driver includes one).
* driver – 这是 JDBC 驱动的 Java 类的完全限定名（并不是 JDBC 驱动中可能包含的数据源类）。
* url – This is the JDBC URL for your database instance. url – 这是数据库的 JDBC URL 地址。
* username – The database username to log in with. url – 这是数据库的 JDBC URL 地址。
* password - The database password to log in with. 登录数据库的密码。
* defaultTransactionIsolationLevel – The default transaction isolation level for connections. 默认的连接事务隔离级别。

Optionally, you can pass properties to the database driver as well. To do this, prefix the properties with driver., for example:

作为可选项，你也可以传递属性给数据库驱动。只需在属性名加上“driver.”前缀即可，例如：

* driver.encoding=UTF8

This will pass the property encoding, with the value UTF8, to your database driver via the DriverManager.getConnection(url, driverProperties) method.

这将通过 DriverManager.getConnection(url,driverProperties) 方法传递值为 UTF8 的 encoding 属性给数据库驱动。

**POOLED** – This implementation of DataSource pools JDBC Connection objects to avoid the initial connection and authentication time required to create a new Connection instance. This is a popular approach for concurrent web applications to achieve the fastest response.

POOLED– 这种数据源的实现利用“池”的概念将 JDBC 连接对象组织起来，避免了创建新的连接实例时所必需的初始化和认证时间。 这是一种使得并发 Web 应用快速响应请求的流行处理方式。

In addition to the (UNPOOLED) properties above, there are many more properties that can be used to configure the POOLED datasource:

除了上述提到 UNPOOLED 下的属性外，还有更多属性用来配置 POOLED 的数据源：

* poolMaximumActiveConnections – This is the number of active (i.e. in use) connections that can exist at any given time. Default: 10 在任意时间可以存在的活动（也就是正在使用）连接数量，默认值：10
* poolMaximumIdleConnections – The number of idle connections that can exist at any given time. 任意时间可能存在的空闲连接数。
* poolMaximumCheckoutTime – This is the amount of time that a Connection can be "checked out" of the pool before it will be forcefully returned. Default: 20000ms (i.e. 20 seconds) 在被强制返回之前，池中连接被检出（checked out）时间，默认值：20000 毫秒（即 20 秒）
* poolTimeToWait – This is a low level setting that gives the pool a chance to print a log status and re-attempt the acquisition of a connection in the case that it’s taking unusually long (to avoid failing silently forever if the pool is misconfigured). Default: 20000ms (i.e. 20 seconds)
* 这是一个底层设置，如果获取连接花费了相当长的时间，连接池会打印状态日志并重新尝试获取一个连接（避免在误配置的情况下一直安静的失败），默认值：20000 毫秒（即 20 秒）。
* poolMaximumLocalBadConnectionTolerance – This is a low level setting about tolerance of bad connections got for any thread. If a thread got a bad connection, it may still have another chance to re-attempt to get another connection which is valid. But the retrying times should not more than the sum of poolMaximumIdleConnections and poolMaximumLocalBadConnectionTolerance. Default: 3 (Since: 3.4.5)
* 这是一个关于坏连接容忍度的底层设置， 作用于每一个尝试从缓存池获取连接的线程。 如果这个线程获取到的是一个坏的连接，那么这个数据源允许这个线程尝试重新获取一个新的连接，但是这个重新尝试的次数不应该超过 poolMaximumIdleConnections 与 poolMaximumLocalBadConnectionTolerance 之和。 默认值：3 （新增于 3.4.5）
* poolPingQuery – The Ping Query is sent to the database to validate that a connection is in good working order and is ready to accept requests. The default is "NO PING QUERY SET", which will cause most database drivers to fail with a decent error message.
* 发送到数据库的侦测查询，用来检验连接是否正常工作并准备接受请求。默认是“NO PING QUERY SET”，这会导致多数数据库驱动失败时带有一个恰当的错误消息。
* poolPingEnabled – This enables or disables the ping query. If enabled, you must also set the poolPingQuery property with a valid SQL statement (preferably a very fast one). Default: false.
* 是否启用侦测查询。若开启，需要设置 poolPingQuery 属性为一个可执行的 SQL 语句（最好是一个速度非常快的 SQL 语句），默认值：false。
* poolPingConnectionsNotUsedFor – This configures how often the poolPingQuery will be used. This can be set to match the typical timeout for a database connection, to avoid unnecessary pings. Default: 0 (i.e. all connections are pinged every time – but only if poolPingEnabled is true of course).
* 配置 poolPingQuery 的频率。可以被设置为和数据库连接超时时间一样，来避免不必要的侦测，默认值：0（即所有连接每一时刻都被侦测 — 当然仅当 poolPingEnabled 为 true 时适用）。

**JNDI** – This implementation of DataSource is intended for use with containers such as EJB or Application Servers that may configure the DataSource centrally or externally and place a reference to it in a JNDI context. This DataSource configuration only requires two properties:

JNDI – 这个数据源的实现是为了能在如 EJB 或应用服务器这类容器中使用，容器可以集中或在外部配置数据源，然后放置一个 JNDI 上下文的引用。这种数据源配置只需要两个属性：

* initial\_context – This property is used for the Context lookup from the InitialContext (i.e. initialContext.lookup(initial\_context)). This property is optional, and if omitted, then the data\_source property will be looked up against the InitialContext directly.
* 这个属性用来在 InitialContext 中寻找上下文（即，initialContext.lookup(initial\_context)）。这是个可选属性，如果忽略，那么将会直接从 InitialContext 中寻找 data\_source 属性。
* data\_source – This is the context path where the reference to the instance of the DataSource can be found. It will be looked up against the context returned by the initial\_context lookup, or against the InitialContext directly if no initial\_context is supplied.
* 这是引用数据源实例位置的上下文的路径。提供了 initial\_context 配置时会在其返回的上下文中进行查找，没有提供时则直接在 InitialContext 中查找。

Similar to the other DataSource configurations, it’s possible to send properties directly to the InitialContext by prefixing those properties with env., for example:

和其他数据源配置类似，可以通过添加前缀“env.”直接把属性传递给初始上下文。比如：

* env.encoding=UTF8

This would send the property encoding with the value of UTF8 to the constructor of the InitialContext upon instantiation.

这就会在初始上下文（InitialContext）实例化时往它的构造方法传递值为 UTF8 的 encoding 属性。

You can plug any 3rd party DataSource by implementing the interface org.apache.ibatis.datasource.DataSourceFactory:

你可以通过实现接口 org.apache.ibatis.datasource.DataSourceFactory 来使用第三方数据源：

public interface DataSourceFactory {

void setProperties(Properties props);

DataSource getDataSource();

}

org.apache.ibatis.datasource.unpooled.UnpooledDataSourceFactory can be used as super class to build new datasource adapters. For example this is the code needed to plug C3P0:

org.apache.ibatis.datasource.unpooled.UnpooledDataSourceFactory 可被用作父类来构建新的数据源适配器，比如下面这段插入 C3P0 数据源所必需的代码：

import org.apache.ibatis.datasource.unpooled.UnpooledDataSourceFactory;

import com.mchange.v2.c3p0.ComboPooledDataSource;

public class C3P0DataSourceFactory extends UnpooledDataSourceFactory {

public C3P0DataSourceFactory() {

this.dataSource = new ComboPooledDataSource();

}

}

To set it up, add a property for each setter method you want MyBatis to call. Follows below a sample configuration which connects to a PostgreSQL database:

为了令其工作，记得为每个希望 MyBatis 调用的 setter 方法在配置文件中增加对应的属性。 下面是一个可以连接至 PostgreSQL 数据库的例子：

<dataSource type="org.myproject.C3P0DataSourceFactory">

<property name="driver" value="org.postgresql.Driver"/>

<property name="url" value="jdbc:postgresql:mydb"/>

<property name="username" value="postgres"/>

<property name="password" value="root"/>

</dataSource>

**databaseIdProvider 数据库厂商标识**

MyBatis is able to execute different statements depending on your database vendor. The multi-db vendor support is based on the mapped statements databaseId attribute. MyBatis will load all statements with no databaseId attribute or with a databaseId that matches the current one. In case the same statement is found with and without the databaseId the latter will be discarded. To enable the multi vendor support add a databaseIdProvider to mybatis-config.xml file as follows:

MyBatis 可以根据不同的数据库厂商执行不同的语句，这种多厂商的支持是基于映射语句中的 databaseId 属性。 MyBatis 会加载不带 databaseId 属性和带有匹配当前数据库 databaseId 属性的所有语句。 如果同时找到带有 databaseId 和不带 databaseId 的相同语句，则后者会被舍弃。 为支持多厂商特性只要像下面这样在 mybatis-config.xml 文件中加入 databaseIdProvider 即可：

<databaseIdProvider type="DB\_VENDOR" />

The DB\_VENDOR implementation databaseIdProvider sets as databaseId the String returned by DatabaseMetaData#getDatabaseProductName(). Given that usually that string is too long and that different versions of the same product may return different values, you may want to convert it to a shorter one by adding properties like follows:

DB\_VENDOR 对应的 databaseIdProvider 实现会将 databaseId 设置为 DatabaseMetaData#getDatabaseProductName() 返回的字符串。 由于通常情况下这些字符串都非常长而且相同产品的不同版本会返回不同的值，所以你可能想通过设置属性别名来使其变短，如下：

<databaseIdProvider type="DB\_VENDOR">

<property name="SQL Server" value="sqlserver"/>

<property name="DB2" value="db2"/>

<property name="Oracle" value="oracle" />

</databaseIdProvider>

When properties are provided, the DB\_VENDOR databaseIdProvider will search the property value corresponding to the first key found in the returned database product name or "null" if there is not a matching property. In this case, if getDatabaseProductName() returns "Oracle (DataDirect)" the databaseId will be set to "oracle".

在提供了属性别名时，DB\_VENDOR 的 databaseIdProvider 实现会将 databaseId 设置为第一个数据库产品名与属性中的名称相匹配的值，如果没有匹配的属性将会设置为 “null”。 在这个例子中，如果 getDatabaseProductName() 返回“Oracle (DataDirect)”，databaseId 将被设置为“oracle”。

You can build your own DatabaseIdProvider by implementing the interface org.apache.ibatis.mapping.DatabaseIdProvider and registering it in mybatis-config.xml:

你可以通过实现接口 org.apache.ibatis.mapping.DatabaseIdProvider 并在 mybatis-config.xml 中注册来构建自己的 DatabaseIdProvider：

public interface DatabaseIdProvider {

void setProperties(Properties p);

String getDatabaseId(DataSource dataSource) throws SQLException;

}

**mappers 映射器**

Now that the behavior of MyBatis is configured with the above configuration elements, we’re ready to define our mapped SQL statements. But first, we need to tell MyBatis where to find them. Java doesn’t really provide any good means of auto-discovery in this regard, so the best way to do it is to simply tell MyBatis where to find the mapping files. You can use classpath relative resource references, fully qualified url references (including file:/// URLs), class names or package names. For example:

既然 MyBatis 的行为已经由上述元素配置完了，我们现在就要定义 SQL 映射语句了。 但是首先我们需要告诉 MyBatis 到哪里去找到这些语句。 Java 在自动查找这方面没有提供一个很好的方法，所以最佳的方式是告诉 MyBatis 到哪里去找映射文件。 你可以使用相对于类路径的资源引用， 或完全限定资源定位符（包括 file:/// 的 URL），或类名和包名等。例如：

<!-- Using classpath relative resources 使用相对于类路径的资源引用-->

<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 使用完全限定资源定位符（URL）-->

<mappers>

<mapper url="file:///var/mappers/AuthorMapper.xml"/>

<mapper url="file:///var/mappers/BlogMapper.xml"/>

<mapper url="file:///var/mappers/PostMapper.xml"/>

</mappers>

<!-- Using mapper interface classes 使用映射器接口实现类的完全限定类名-->

<mappers>

<mapper class="org.mybatis.builder.AuthorMapper"/>

<mapper class="org.mybatis.builder.BlogMapper"/>

<mapper class="org.mybatis.builder.PostMapper"/>

</mappers>

<!-- Register all interfaces in a package as mappers 将包内的映射器接口实现全部注册为映射器-->

<mappers>

<package name="org.mybatis.builder"/>

</mappers>

These statement simply tell MyBatis where to go from here. The rest of the details are in each of the SQL Mapping files, and that’s exactly what the next section will discuss.

这些配置会告诉了 MyBatis 去哪里找映射文件，剩下的细节就应该是每个 SQL 映射文件了，也就是接下来我们要讨论的。

**Mapper XML Files XML 映射文件**

The true power of MyBatis is in the Mapped Statements. This is where the magic happens. For all of their power, the Mapper XML files are relatively simple. Certainly if you were to compare them to the equivalent JDBC code, you would immediately see a savings of 95% of the code. MyBatis was built to focus on the SQL, and does its best to stay out of your way.

MyBatis 的真正强大在于它的映射语句，这是它的魔力所在。由于它的异常强大，映射器的 XML 文件就显得相对简单。如果拿它跟具有相同功能的 JDBC 代码进行对比，你会立即发现省掉了将近 95% 的代码。MyBatis 为聚焦于 SQL 而构建，以尽可能地为你减少麻烦。

The Mapper XML files have only a few first class elements (in the order that they should be defined):

SQL 映射文件只有很少的几个顶级元素（按照应被定义的顺序列出）：

* **cache** – Configuration of the cache for a given namespace. 对给定命名空间的缓存配置。
* **cache-ref** – Reference to a cache configuration from another namespace. 对其他命名空间缓存配置的引用。
* **resultMap** – The most complicated and powerful element that describes how to load your objects from the database result sets. 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~~**~~– Deprecated! Old-school way to map parameters. Inline parameters are preferred and this element may be removed in the future. Not documented here.~~ 已被废弃！老式风格的参数映射。更好的办法是使用内联参数，此元素可能在将来被移除。文档中不会介绍此元素。
* **sql** – A reusable chunk of SQL that can be referenced by other statements. 可被其他语句引用的可重用语句块。
* **insert** – A mapped INSERT statement. 映射插入语句
* **update** – A mapped UPDATE statement.
* **delete** – A mapped DELETE statement.
* **select** – A mapped SELECT statement.

The next sections will describe each of these elements in detail, starting with the statements themselves.

下一部分将从语句本身开始来描述每个元素的细节。

**select**

The select statement is one of the most popular elements that you'll use in MyBatis. Putting data in a database isn't terribly valuable until you get it back out, so most applications query far more than they modify the data. For every insert, update or delete, there are probably many selects. This is one of the founding principles of MyBatis, and is the reason so much focus and effort was placed on querying and result mapping. The select element is quite simple for simple cases. For example:

查询语句是 MyBatis 中最常用的元素之一，光能把数据存到数据库中价值并不大，只有还能重新取出来才有用，多数应用也都是查询比修改要频繁。对每个插入、更新或删除操作，通常间隔多个查询操作。这是 MyBatis 的基本原则之一，也是将焦点和努力放在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>

This statement is called selectPerson, takes a parameter of type int (or Integer), and returns a HashMap keyed by column names mapped to row values.

这个语句被称作 selectPerson，接受一个 int（或 Integer）类型的参数，并返回一个 HashMap 类型的对象，其中的键是列名，值便是结果行中的对应值。

Notice the parameter notation:

注意参数符号：

#{id}

This tells MyBatis to create a PreparedStatement parameter. With JDBC, such a parameter would be identified by a "?" in SQL passed to a new PreparedStatement, something like this:

这就告诉 MyBatis 创建一个预处理语句（PreparedStatement）参数，在 JDBC 中，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… 近似的 JDBC 代码，非 MyBatis 代码...

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

PreparedStatement ps = conn.prepareStatement(selectPerson);

ps.setInt(1,id);

Of course, there's a lot more code required by JDBC alone to extract the results and map them to an instance of an object, which is what MyBatis saves you from having to do. There's a lot more to know about parameter and result mapping. Those details warrant their own section, which follows later in this section.

当然，使用 JDBC 意味着需要更多的代码来提取结果并将它们映射到对象实例中，而这就是 MyBatis 节省你时间的地方。参数和结果映射还有更深入的细节。这些细节会分别在后面单独的小节中呈现。

The select element has more attributes that allow you to configure the details of how each statement should behave.

select 元素允许你配置很多属性来配置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="10"

fetchSize="256"

statementType="PREPARED"

resultSetType="FORWARD\_ONLY">

* **id** A unique identifier in this namespace that can be used to reference this statement.
* 在命名空间中唯一的标识符，可以被用来引用这条语句。
* **parameterType** The fully qualified class name or alias for the parameter that will be passed into this statement. This attribute is optional because MyBatis can calculate the TypeHandler to use out of the actual parameter passed to the statement. Default is unset.
* 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 可以通过类型处理器（TypeHandler） 推断出具体传入语句的参数，默认值为未设置（unset）。
* **parameterMap** This is a deprecated approach to referencing an external parameterMap. Use inline parameter mappings and the parameterType attribute.
* 这是引用外部 parameterMap 的已经被废弃的方法。请使用内联参数映射和 parameterType 属性。
* **resultType** - The fully qualified class name or alias for the expected type that will be returned from this statement. Note that in the case of collections, this should be the type that the collection contains, not the type of the collection itself. Use resultType OR resultMap, not both.
* 从这条语句中返回的期望类型的类的完全限定名或别名。 注意如果返回的是集合，那应该设置为集合包含的类型，而不是集合本身。可以使用 resultType 或 resultMap，但不能同时使用。
* **resultMap** - A named reference to an external resultMap. Result maps are the most powerful feature of MyBatis, and with a good understanding of them, many difficult mapping cases can be solved. Use resultMap OR resultType, not both.
* 外部 resultMap 的命名引用。结果集的映射是 MyBatis 最强大的特性，如果你对其理解透彻，许多复杂映射的情形都能迎刃而解。可以使用 resultMap 或 resultType，但不能同时使用。
* **flushCache** - Setting this to true will cause the local and 2nd level caches to be flushed whenever this statement is called. Default: false for select statements.
* 将其设置为 true 后，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：false。
* **useCache** - Setting this to true will cause the results of this statement to be cached in 2nd level cache. Default: true for select statements.
* 将其设置为 true 后，将会导致本条语句的结果被二级缓存缓存起来，默认值：对 select 元素为 true。
* **timeout** - This sets the number of seconds the driver will wait for the database to return from a request, before throwing an exception. Default is unset (driver dependent).
* 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。默认值为未设置（unset）（依赖驱动）。
* **fetchSize** - This is a driver hint that will attempt to cause the driver to return results in batches of rows numbering in size equal to this setting. Default is unset (driver dependent).
* 这是一个给驱动的提示，尝试让驱动程序每次批量返回的结果行数和这个设置值相等。 默认值为未设置（unset）（依赖驱动）。
* **statementType** - Any one of STATEMENT, PREPARED or CALLABLE. This causes MyBatis to use Statement, PreparedStatement or CallableStatement respectively. Default: PREPARED.
* STATEMENT，PREPARED 或 CALLABLE 中的一个。这会让 MyBatis 分别使用 Statement，PreparedStatement 或 CallableStatement，默认值：PREPARED。
* **resultSetType** - Any one of FORWARD\_ONLY|SCROLL\_SENSITIVE|SCROLL\_INSENSITIVE|DEFAULT(same as unset). Default is unset (driver dependent).
* FORWARD\_ONLY，SCROLL\_SENSITIVE, SCROLL\_INSENSITIVE 或 DEFAULT（等价于 unset） 中的一个，默认值为 unset （依赖驱动）。
* **databaseId** - In case there is a configured databaseIdProvider, MyBatis will load all statements with no databaseId attribute or with a databaseId that matches the current one. If case the same statement if found with and without the databaseId the latter will be discarded.
* 如果配置了数据库厂商标识（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** - This is only applicable for nested result select statements: If this is true, it is assumed that nested results are contained or grouped together such that when a new main result row is returned, no references to a previous result row will occur anymore. This allows nested results to be filled much more memory friendly. Default: false.
* 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，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** - This is only applicable for multiple result sets. It lists the result sets that will be returned by the statement and gives a name to each one. Names are separated by commas.
* 这个设置仅对多结果集的情况适用。它将列出语句执行后返回的结果集并给每个结果集一个名称，名称是逗号分隔的。

**insert, update and delete**

The data modification statements insert, update and delete are very similar in their implementation:

数据变更语句 insert，update 和 delete 的实现非常接近：

<insert

id="insertAuthor"

parameterType="domain.blog.Author"

flushCache="true"

statementType="PREPARED"

keyProperty=""

keyColumn=""

useGeneratedKeys=""

timeout="20">

<update

id="updateAuthor"

parameterType="domain.blog.Author"

flushCache="true"

statementType="PREPARED"

timeout="20">

<delete

id="deleteAuthor"

parameterType="domain.blog.Author"

flushCache="true"

statementType="PREPARED"

timeout="20">

* **id** A unique identifier in this namespace that can be used to reference this statement.
* 命名空间中的唯一标识符，可被用来代表这条语句。
* **parameterType** - The fully qualified class name or alias for the parameter that will be passed into this statement. This attribute is optional because MyBatis can calculate the TypeHandler to use out of the actual parameter passed to the statement. Default is unset.
* 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 可以通过类型处理器推断出具体传入语句的参数，默认值为未设置（unset）。
* **parameterMap** - This is a deprecated approach to referencing an external parameterMap. Use inline parameter mappings and the parameterType attribute.
* 这是引用外部 parameterMap 的已经被废弃的方法。请使用内联参数映射和 parameterType 属性。
* **flushCache** - Setting this to true will cause the 2nd level and local caches to be flushed whenever this statement is called. Default: true for insert, update and delete statements.
* 将其设置为 true 后，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（对于 insert、update 和 delete 语句）。
* **timeout** - This sets the maximum number of seconds the driver will wait for the database to return from a request, before throwing an exception. Default is unset (driver dependent).
* 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。默认值为未设置（unset）（依赖驱动）。
* **statementType** - Any one of STATEMENT, PREPARED or CALLABLE. This causes MyBatis to use Statement, PreparedStatement or CallableStatement respectively. Default: PREPARED.
* STATEMENT，PREPARED 或 CALLABLE 的一个。这会让 MyBatis 分别使用 Statement，PreparedStatement 或 CallableStatement，默认值：PREPARED。
* **useGeneratedKeys** - (insert and update only) This tells MyBatis to use the JDBC getGeneratedKeys method to retrieve keys generated internally by the database (e.g. auto increment fields in RDBMS like MySQL or SQL Server). Default: false.
* （仅对 insert 和 update 有用）这会令 MyBatis 使用 JDBC 的 getGeneratedKeys 方法来取出由数据库内部生成的主键（比如：像 MySQL 和 SQL Server 这样的关系数据库管理系统的自动递增字段），默认值：false。
* **keyProperty** - (insert and update only) Identifies a property into which MyBatis will set the key value returned by getGeneratedKeys, or by a selectKey child element of the insert statement. Default: unset. Can be a comma separated list of property names if multiple generated columns are expected.
* （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，默认值：未设置（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** - (insert and update only) Sets the name of the column in the table with a generated key. This is only required in certain databases (like PostgreSQL) when the key column is not the first column in the table. Can be a comma separated list of columns names if multiple generated columns are expected.
* （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。如果希望使用多个生成的列，也可以设置为逗号分隔的属性名称列表。
* **databaseId** - In case there is a configured databaseIdProvider, MyBatis will load all statements with no databaseId attribute or with a databaseId that matches the current one. If case the same statement if found with and without the databaseId the latter will be discarded.
* 如果配置了数据库厂商标识（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。

The following are some examples of insert, update and delete statements.

下面就是 insert，update 和 delete 语句的示例：

<insert id="insertAuthor">

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

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

</insert>

<update id="updateAuthor">

update Author set

username = #{username},

password = #{password},

email = #{email},

bio = #{bio}

where id = #{id}

</update>

<delete id="deleteAuthor">

delete from Author where id = #{id}

</delete>

As mentioned, insert is a little bit more rich in that it has a few extra attributes and sub-elements that allow it to deal with key generation in a number of ways.

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

First, if your database supports auto-generated key fields (e.g. MySQL and SQL Server), then you can simply set useGeneratedKeys="true" and set the keyProperty to the target property and you're done. For example, if the Author table above had used an auto-generated column type for the id, the statement would be modified as follows:

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

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

keyProperty="id">

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

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

</insert>

If your database also supports multi-row insert, you can pass a list or an array of Authors and retrieve the auto-generated keys.

如果你的数据库还支持多行插入, 你也可以传入一个 Author 数组或集合，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>

MyBatis has another way to deal with key generation for databases that don't support auto-generated column types, or perhaps don't yet support the JDBC driver support for auto-generated keys.

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。

Here's a simple (silly) example that would generate a random ID (something you'd likely never do, but this demonstrates the flexibility and how MyBatis really doesn't mind):

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>

In the example above, the selectKey statement would be run first, the Author id property would be set, and then the insert statement would be called. This gives you a similar behavior to an auto-generated key in your database without complicating your Java code.

在上面的示例中，selectKey 元素中的语句将会首先运行，Author 的 id 会被设置，然后插入语句会被调用。这可以提供给你一个与数据库中自动生成主键类似的行为，同时保持了 Java 代码的简洁。

The selectKey element is described as follows:

selectKey 元素描述如下：

<selectKey

keyProperty="id"

resultType="int"

order="BEFORE"

statementType="PREPARED">

* **keyProperty** - The target property where the result of the selectKey statement should be set. Can be a comma separated list of property names if multiple generated columns are expected.
* 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** - The column name(s) in the returned result set that match the properties. Can be a comma separated list of column names if multiple generated columns are expected.
* 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** - The type of the result. MyBatis can usually figure this out, but it doesn't hurt to add it to be sure. MyBatis allows any simple type to be used as the key, including Strings. If you are expecting multiple generated columns, then you can use an Object that contains the expected properties, or a Map.
* 结果的类型。MyBatis 通常可以推断出来，但是为了更加精确，写上也不会有什么问题。MyBatis 允许将任何简单类型用作主键的类型，包括字符串。如果希望作用于多个生成的列，则可以使用一个包含期望属性的 Object 或一个 Map。
* **order** - This can be set to BEFORE or AFTER. If set to BEFORE, then it will select the key first, set the keyProperty and then execute the insert statement. If set to AFTER, it runs the insert statement and then the selectKey statement – which is common with databases like Oracle that may have embedded sequence calls inside of insert statements.
* 这可以被设置为 BEFORE 或 AFTER。如果设置为 BEFORE，那么它会首先生成主键，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 中的语句 - 这和 Oracle 数据库的行为相似，在插入语句内部可能有嵌入索引调用。
* **statementType** - Same as above, MyBatis supports STATEMENT, PREPARED and CALLABLE statement types that map to Statement, PreparedStatement and CallableStatement respectively.
* 与前面相同，MyBatis 支持 STATEMENT，PREPARED 和 CALLABLE 语句的映射类型，分别代表 PreparedStatement 和 CallableStatement 类型。

**sql**

This element can be used to define a reusable fragment of SQL code that can be included in other statements. It can be statically (during load phase) parametrized. Different property values can vary in include instances. For example:

这个元素可以被用来定义可重用的 SQL 代码段，这些 SQL 代码可以被包含在其他语句中。它可以（在加载的时候）被静态地设置参数。 在不同的包含语句中可以设置不同的值到参数占位符上。比如：

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

The SQL fragment can then be included in another statement, for example:

这个 SQL 片段可以被包含在其他语句中，例如：

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

select

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

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

from some\_table t1

cross join some\_table t2

</select>

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

属性值也可以被用在 include 元素的 refid 属性里或 include 元素的内部语句中，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>

**Parameters 参数**

In all of the past statements, you've seen examples of simple parameters. Parameters are very powerful elements in MyBatis. For simple situations, probably 90% of the cases, there's not much to them, for example:

你之前见到的所有语句中，使用的都是简单参数。实际上参数是 MyBatis 非常强大的元素。对于简单的使用场景，大约 90% 的情况下你都不需要使用复杂的参数，比如：

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

select id, username, password

from users

where id = #{id}

</select>

The example above demonstrates a very simple named parameter mapping. The parameterType is set to int, so therefore the parameter could be named anything. Primitive or simple data types such as Integer and String have no relevant properties, and thus will replace the full value of the parameter entirely. However, if you pass in a complex object, then the behavior is a little different. For example:

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

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

insert into users (id, username, password)

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

</insert>

If a parameter object of type User was passed into that statement, the id, username and password property would be looked up and their values passed to a PreparedStatement parameter.

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。

That's nice and simple for passing parameters into statements. But there are a lot of other features of parameter maps.

对向语句中传递参数来说，这真是既简单又有效。不过参数映射的功能远不止于此。

First, like other parts of MyBatis, parameters can specify a more specific data type.

首先，像 MyBatis 的其他部分一样，参数也可以指定一个特殊的数据类型。

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

Like the rest of MyBatis, the javaType can almost always be determined from the parameter object, unless that object is a HashMap. Then the javaType should be specified to ensure the correct TypeHandler is used.

像 MyBatis 的其它部分一样，javaType 几乎总是可以根据参数对象的类型确定下来，除非该对象是一个 HashMap。这个时候，你需要显式指定 javaType 来确保正确的类型处理器（TypeHandler）被使用。

**NOTE** The JDBC Type is required by JDBC for all nullable columns, if null is passed as a value. You can investigate this yourself by reading the JavaDocs for the PreparedStatement.setNull() method.

提示 JDBC 要求，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 JavaDoc 文档来获取更多信息。

To further customize type handling, you can also specify a specific TypeHandler class (or alias), for example:

要更进一步地自定义类型处理方式，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}

So already it seems to be getting verbose, but the truth is that you'll rarely set any of these.

尽管看起来配置变得越来越繁琐，但实际上，很少需要如此繁琐的配置。

For numeric types there's also a numericScale for determining how many decimal places are relevant.

对于数值类型，还有一个小数保留位数的设置，来指定小数点后保留的位数。

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

Finally, the mode attribute allows you to specify IN, OUT or INOUT parameters. If a parameter is OUT or INOUT, the actual value of the parameter object property will be changed, just as you would expect if you were calling for an output parameter. If the mode=OUT (or INOUT) and the jdbcType=CURSOR (i.e. Oracle REFCURSOR), you must specify a resultMap to map the ResultSet to the type of the parameter. Note that the javaType attribute is optional here, it will be automatically set to ResultSet if left blank with a CURSOR as the jdbcType.

最后，mode 属性允许你指定 IN，OUT 或 INOUT 参数。如果参数的 mode 为 OUT 或 INOUT，就像你在指定输出参数时所期望的行为那样，参数对象的属性实际值将会被改变。 如果 mode 为 OUT（或 INOUT），而且 jdbcType 为 CURSOR（也就是 Oracle 的 REFCURSOR），你必须指定一个 resultMap 引用来将结果集 ResultMap 映射到参数的类型上。要注意这里的 javaType 属性是可选的，如果留空并且 jdbcType 是 CURSOR，它会被自动地被设为 ResultMap。

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

MyBatis also supports more advanced data types such as structs, but you must tell the statement the type name when registering the out parameter. For example (again, don't break lines like this in practice):

MyBatis 也支持很多高级的数据类型，比如结构体（structs），但是当使用 out 参数时，你必须显式设置类型的名称。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}

Despite all of these powerful options, most of the time you'll simply specify the property name, and MyBatis will figure out the rest. At most, you'll specify the jdbcType for nullable columns.

尽管所有这些选项很强大，但大多时候你只须简单地指定属性名，其他的事情 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}

**String Substitution 字符串替换**

By default, using the #{} syntax will cause MyBatis to generate PreparedStatement properties and set the values safely against the PreparedStatement parameters (e.g. ?). While this is safer, faster and almost always preferred, sometimes you just want to directly inject an unmodified string into the SQL Statement. For example, for ORDER BY, you might use something like this:

默认情况下,使用 #{} 格式的语法会导致 MyBatis 创建 PreparedStatement 参数占位符并安全地设置参数（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。 比如，像 ORDER BY，你可以这样来使用：

ORDER BY ${columnName}

Here MyBatis won't modify or escape the string.

这里 MyBatis 不会修改或转义字符串。

String Substitution can be very useful when the metadata(i.e. table name or column name) in the sql statement is dynamic, for example, if you want to select from a table by any one of its columns, instead of writing code like:

当 SQL 语句中的元数据（如表名或列名）是动态生成的时候，字符串替换将会非常有用。 举个例子，如果你想通过任何一列从表中 select 数据时，不需要像下面这样写：

@Select("select \* from user where id = #{id}")

User findById(@Param("id") long id);

@Select("select \* from user where name = #{name}")

User findByName(@Param("name") String name);

@Select("select \* from user where email = #{email}")

User findByEmail(@Param("email") String email);

// and more "findByXxx" method

you can just write: 可以只写这样一个方法：

@Select("select \* from user where ${column} = #{value}")

User findByColumn(@Param("column") String column, @Param("value") String value);

in which the ${column} will be substituted directly and the #{value} will be "prepared". Thus you can just do the same work by:

其中 ${column} 会被直接替换，而 #{value} 会被使用 ? 预处理。 因此你就可以像下面这样来达到上述功能：

User userOfId1 = userMapper.findByColumn("id", 1L);

User userOfNameKid = userMapper.findByColumn("name", "kid");

User userOfEmail = userMapper.findByColumn("email", "noone@nowhere.com");

This idea can be applied to substitute the table name as well.

这个想法也同样适用于用来替换表名的情况。

**NOTE** It's not safe to accept input from a user and supply it to a statement unmodified in this way. This leads to potential SQL Injection attacks and therefore you should either disallow user input in these fields, or always perform your own escapes and checks.

提示 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 结果映射**

The resultMap element is the most important and powerful element in MyBatis. It's what allows you to do away with 90% of the code that JDBC requires to retrieve data from ResultSets, and in some cases allows you to do things that JDBC does not even support. In fact, to write the equivalent code for something like a join mapping for a complex statement could probably span thousands of lines of code. The design of the ResultMaps is such that simple statements don't require explicit result mappings at all, and more complex statements require no more than is absolutely necessary to describe the relationships.

resultMap 元素是 MyBatis 中最重要最强大的元素。它可以让你从 90% 的 JDBC ResultSets 数据提取代码中解放出来，并在一些情形下允许你进行一些 JDBC 不支持的操作。实际上，在为一些比如连接的复杂语句编写映射代码的时候，一份 resultMap 能够代替实现同等功能的长达数千行的代码。ResultMap 的设计思想是，对于简单的语句根本不需要配置显式的结果映射，而对于复杂一点的语句只需要描述它们的关系就行了。

You've already seen examples of simple mapped statements that don't have an explicit resultMap. For example:

你已经见过简单映射语句的示例了，但并没有显式指定 resultMap。比如：

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

select id, username, hashedPassword

from some\_table

where id = #{id}

</select>

Such a statement simply results in all columns being automatically mapped to the keys of a HashMap, as specified by the resultType attribute. While useful in many cases, a HashMap doesn't make a very good domain model. It's more likely that your application will use JavaBeans or POJOs (Plain Old Java Objects) for the domain model. MyBatis supports both. Consider the following JavaBean:

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 不是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，普通老式 Java 对象）作为领域模型。MyBatis 对两者都提供了支持。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;

}

}

Based on the JavaBeans specification, the above class has 3 properties: id, username, and hashedPassword. These match up exactly with the column names in the select statement.

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

Such a JavaBean could be mapped to a ResultSet just as easily as the HashMap.

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>

And remember that TypeAliases are your friends. Use them so that you don't have to keep typing the fully qualified path of your class out. For example:

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 Config XML file -->

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

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

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

select id, username, hashedPassword

from some\_table

where id = #{id}

</select>

In these cases MyBatis is automatically creating a ResultMap behind the scenes to auto-map the columns to the JavaBean properties based on name. If the column names did not match exactly, you could employ select clause aliases (a standard SQL feature) on the column names to make the labels match. For example:

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>

The great thing about ResultMaps is that you've already learned a lot about them, but you haven't even seen one yet! These simple cases don't require any more than you've seen here. Just for example sake, let's see what this last example would look like as an external resultMap, as that is another way to solve column name mismatches.

ResultMap 最优秀的地方在于，虽然你已经对它相当了解了，但是根本就不需要显式地用到他们。 上面这些简单的示例根本不需要下面这些繁琐的配置。 但出于示范的原因，让我们来看看最后一个示例中，如果使用外部的 resultMap 会怎样，这也是解决列名不匹配的另外一种方式。

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

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

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

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

</resultMap>

And the statement that references it uses the resultMap attribute to do so (notice we removed the resultType attribute). For example:

而在引用它的语句中使用 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>

Now if only the world was always that simple.

如果世界总是这么简单就好了。

**Advanced Result Maps 高级结果映射**

MyBatis was created with one idea in mind: Databases aren't always what you want or need them to be. While we'd love every database to be perfect 3rd normal form or BCNF, they aren't. And it would be great if it was possible to have a single database map perfectly to all of the applications that use it, it's not. Result Maps are the answer that MyBatis provides to this problem.

MyBatis 创建时的一个思想是：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，可惜它们不总都是这样。 如果能有一种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。 而 ResultMap 就是 MyBatis 对这个问题的答案。

For example, how would we map this statement?

比如，我们如何映射下面这个语句？

<!-- Very Complex Statement 非常复杂的语句 -->

<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>

You'd probably want to map it to an intelligent object model consisting of a Blog that was written by an Author, and has many Posts, each of which may have zero or many Comments and Tags. The following is a complete example of a complex ResultMap (assume Author, Blog, Post, Comments and Tags are all type aliases). Have a look at it, but don't worry, we're going to go through each step. While it may look daunting at first, it's actually very simple.

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，每篇博文有零或多条的评论和标签。 我们来看看下面这个完整的例子，它是一个非常复杂的结果映射（假设作者，博客，博文，评论和标签都是类型别名）。 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。

<!-- Very Complex Result Map 非常复杂的结果映射 -->

<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>

The resultMap element has a number of sub-elements and a structure worthy of some discussion. The following is a conceptual view of the resultMap element.

resultMap 元素有很多子元素和一个值得深入探讨的结构。 下面是resultMap 元素的概念视图。

**resultMap 结果映射**

* **constructor** - used for injecting results into the constructor of a class upon instantiation 用于在实例化类时，注入结果到构造方法中
  + **idArg** - ID argument; flagging results as ID will help improve overall performance ID 参数；标记出作为 ID 的结果可以帮助提高整体性能
  + **arg** - a normal result injected into the constructor 将被注入到构造方法的一个普通结果
* **id** – an ID result; flagging results as ID will help improve overall performance 一个 ID 结果；标记出作为 ID 的结果可以帮助提高整体性能
* **result** – a normal result injected into a field or JavaBean property 注入到字段或 JavaBean 属性的普通结果
* **association** – a complex type association; many results will roll up into this type 一个复杂类型的关联；许多结果将包装成这种类型
  + **nested** result mappings – associations are resultMaps themselves, or can refer to one 嵌套结果映射 – 关联本身可以是一个 resultMap 元素，或者从别处引用一个
* **collection** – a collection of complex types 一个复杂类型的集合
  + **nested** result mappings – collections are resultMaps themselves, or can refer to one 嵌套结果映射 – 集合本身可以是一个 resultMap 元素，或者从别处引用一个
* **discriminator** – uses a result value to determine which resultMap to use 使用结果值来决定使用哪个 resultMap
  + **case** – a case is a result map based on some value 基于某些值的结果映射
    - **nested** result mappings – a case is also a result map itself, and thus can contain many of these same elements, or it can refer to an external resultMap.
    - 嵌套结果映射 – case 本身可以是一个 resultMap 元素，因此可以具有相同的结构和元素，或者从别处引用一个

**id** A unique identifier in this namespace that can be used to reference this result map.

当前命名空间中的一个唯一标识，用于标识一个结果映射。

**type** A fully qualified Java class name, or a type alias (see the table above for the list of built-in type aliases).

类的完全限定名, 或者一个类型别名（关于内置的类型别名，可以参考上面的表格）。

**autoMapping** If present, MyBatis will enable or disable the automapping for this ResultMap. This attribute overrides the global autoMappingBehavior. Default: unset.

如果设置这个属性，MyBatis将会为本结果映射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：未设置（unset）。

**Best Practice** Always build ResultMaps incrementally. Unit tests really help out here. If you try to build a gigantic resultMap like the one above all at once, it's likely you'll get it wrong and it will be hard to work with. Start simple, and evolve it a step at a time. And unit test! The downside to using frameworks is that they are sometimes a bit of a black box (open source or not). Your best bet to ensure that you're achieving the behaviour that you intend, is to write unit tests. It also helps to have them when submitting bugs.

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，逐步迭代。而且别忘了单元测试！ 使用框架的缺点是有时候它们看上去像黑盒子（无论源代码是否可见）。 为了确保你实现的行为和想要的一致，最好的选择是编写单元测试。提交 bug 的时候它也能起到很大的作用。

The next sections will walk through each of the elements in more detail.

下一部分将详细说明每个元素。

**id & result**

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

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

These are the most basic of result mappings. Both id and result map a single column value to a single property or field of a simple data type (String, int, double, Date, etc.).

这些是结果映射最基本的内容。id 和 result 元素都将一个列的值映射到一个简单数据类型（String, int, double, Date 等）的属性或字段。

The only difference between the two is that id will flag the result as an identifier property to be used when comparing object instances. This helps to improve general performance, but especially performance of caching and nested result mapping (i.e. join mapping).

这两者之间的唯一不同是，id 元素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，尤其是进行缓存和嵌套结果映射（也就是连接映射）的时候。

Each has a number of attributes:

两个元素都有一些属性：

**property** - The field or property to map the column result to. If a matching JavaBeans property exists for the given name, then that will be used. Otherwise, MyBatis will look for a field of the given name. In both cases you can use complex property navigation using the usual dot notation. For example, you can map to something simple like: username, or to something more complicated like: address.street.number.

映射到列结果的字段或属性。如果用来匹配的 JavaBean 存在给定名字的属性，那么它将会被使用。否则 MyBatis 将会寻找给定名称的字段。 无论是哪一种情形，你都可以使用通常的点式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上：“address.street.number”。

**column** - The column name from the database, or the aliased column label. This is the same string that would normally be passed to resultSet.getString(columnName).

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** - A fully qualified Java class name, or a type alias (see the table above for the list of built-in type aliases). MyBatis can usually figure out the type if you're mapping to a JavaBean. However, if you are mapping to a HashMap, then you should specify the javaType explicitly to ensure the desired behaviour.

一个 Java 类的完全限定名，或一个类型别名（关于内置的类型别名，可以参考上面的表格）。 如果你映射到一个 JavaBean，MyBatis 通常可以推断类型。然而，如果你映射到的是 HashMap，那么你应该明确地指定 javaType 来保证行为与期望的相一致。

**jdbcType** - The JDBC Type from the list of supported types that follows this table. The JDBC type is only required for nullable columns upon insert, update or delete. This is a JDBC requirement, not a MyBatis one. So even if you were coding JDBC directly, you'd need to specify this type – but only for nullable values.

JDBC 类型，所支持的 JDBC 类型参见这个表格之后的“支持的 JDBC 类型”。 只需要在可能执行插入、更新和删除的且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，你需要对可能存在空值的列指定这个类型。

**typeHandler** - We discussed default type handlers previously in this documentation. Using this property you can override the default type handler on a mapping-by-mapping basis. The value is either a fully qualified class name of a TypeHandler implementation, or a type alias.

我们在前面讨论过默认的类型处理器。使用这个属性，你可以覆盖默认的类型处理器。 这个属性值是一个类型处理器实现类的完全限定名，或者是类型别名。

**Supported JDBC Types 支持的 JDBC 类型**

For future reference, MyBatis supports the following JDBC Types via the included JdbcType enumeration.

为了以后可能的使用场景，MyBatis 通过内置的 jdbcType 枚举类型支持下面的 JDBC 类型。

**constructor 构造方法**

While properties will work for most Data Transfer Object (DTO) type classes, and likely most of your domain model, there may be some cases where you want to use immutable classes. Often tables that contain reference or lookup data that rarely or never changes is suited to immutable classes. Constructor injection allows you to set values on a class upon instantiation, without exposing public methods. MyBatis also supports private properties and private JavaBeans properties to achieve this, but some people prefer Constructor injection. The constructor element enables this.

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（Data Transfer 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。 一般来说，很少改变或基本不变的包含引用或数据的表，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，而不用暴露出公有方法。MyBatis 也支持私有属性和私有 JavaBean 属性来完成注入，但有一些人更青睐于通过构造方法进行注入。 constructor 元素就是为此而生的。

Consider the following constructor:

看看下面这个构造方法:

public class User {

//...

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

//...

}

//...

}

In order to inject the results into the constructor, MyBatis needs to identify the constructor for somehow. In the following example, MyBatis searches a constructor declared with three parameters: java.lang.Integer, java.lang.String and int in this order.

为了将结果注入构造方法，MyBatis 需要通过某种方式定位相应的构造方法。 在下面的例子中，MyBatis 搜索一个声明了三个形参的的构造方法，参数类型以 java.lang.Integer, java.lang.String 和 int 的顺序给出。

<constructor>

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

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

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

</constructor>

When you are dealing with a constructor with many parameters, maintaining the order of arg elements is error-prone.Since 3.4.3, by specifying the name of each parameter, you can write arg elements in any order. To reference constructor parameters by their names, you can either add @Param annotation to them or compile the project with '-parameters' compiler option and enable useActualParamName (this option is enabled by default). The following example is valid for the same constructor even though the order of the second and the third parameters does not match with the declared order.

当你在处理一个带有多个形参的构造方法时，很容易搞乱 arg 元素的顺序。 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。下面是一个等价的例子，尽管函数签名中第二和第三个形参的顺序与 constructor 元素中参数声明的顺序不匹配。

<constructor>

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

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

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

</constructor>

javaType can be omitted if there is a property with the same name and type.

如果存在名称和类型相同的属性，那么可以省略 javaType 。

The rest of the attributes and rules are the same as for the regular id and result elements.

剩余的属性和规则和普通的 id 和 result 元素是一样的。

**column** - The column name from the database, or the aliased column label. This is the same string that would normally be passed to resultSet.getString(columnName).

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** - A fully qualified Java class name, or a type alias (see the table above for the list of built-in type aliases). MyBatis can usually figure out the type if you're mapping to a JavaBean. However, if you are mapping to a HashMap, then you should specify the javaType explicitly to ensure the desired behaviour.

一个 Java 类的完全限定名，或一个类型别名（关于内置的类型别名，可以参考上面的表格）。 如果你映射到一个 JavaBean，MyBatis 通常可以推断类型。然而，如果你映射到的是 HashMap，那么你应该明确地指定 javaType 来保证行为与期望的相一致。

**jdbcType** - The JDBC Type from the list of supported types that follows this table. The JDBC type is only required for nullable columns upon insert, update or delete. This is a JDBC requirement, not an MyBatis one. So even if you were coding JDBC directly, you'd need to specify this type – but only for nullable values.

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、更新和删除的且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，你需要对可能存在空值的列指定这个类型。

**typeHandler** - We discussed default type handlers previously in this documentation. Using this property you can override the default type handler on a mapping-by-mapping basis. The value is either a fully qualified class name of a TypeHandler implementation, or a type alias.

我们在前面讨论过默认的类型处理器。使用这个属性，你可以覆盖默认的类型处理器。 这个属性值是一个类型处理器实现类的完全限定名，或者是类型别名。

**select** - The ID of another mapped statement that will load the complex type required by this property mapping. The values retrieved from columns specified in the column attribute will be passed to the target select statement as parameters. See the Association element for more.

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。具体请参考关联元素。

**resultMap** - This is the ID of a ResultMap that can map the nested results of this argument into an appropriate object graph. This is an alternative to using a call to another select statement. It allows you to join multiple tables together into a single ResultSet. Such a ResultSet will contain duplicated, repeating groups of data that needs to be decomposed and mapped properly to a nested object graph. To facilitate this, MyBatis lets you "chain" result maps together, to deal with the nested results. See the Association element below for more.

结果映射的 ID，可以将嵌套的结果集映射到一个合适的对象树中。 它可以作为使用额外 select 语句的替代方案。它可以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。为了将结果集正确地映射到嵌套的对象树中，MyBatis 允许你 “串联”结果映射，以便解决嵌套结果集的问题。想了解更多内容，请参考下面的关联元素。

**name** - The name of the constructor parameter. Specifying name allows you to write arg elements in any order. See the above explanation. Since 3.4.3.

构造方法形参的名字。从 3.4.3 版本开始，通过指定具体的参数名，你可以以任意顺序写入 arg 元素。参看上面的解释。

**association 关联**

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

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

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

</association>

The association element deals with a "has-one" type relationship. For example, in our example, a Blog has one Author. An association mapping works mostly like any other result. You specify the target property, the javaType of the property (which MyBatis can figure out most of the time), the jdbcType if necessary and a typeHandler if you want to override the retrieval of the result values.

关联（association）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。关联结果映射和其它类型的映射工作方式差不多。 你需要指定目标属性名以及属性的javaType（很多时候 MyBatis 可以自己推断出来），在必要的情况下你还可以设置 JDBC 类型，如果你想覆盖获取结果值的过程，还可以设置类型处理器。

Where the association differs is that you need to tell MyBatis how to load the association. MyBatis can do so in two different ways:

关联的不同之处是，你需要告诉 MyBatis 如何加载关联。MyBatis 有两种不同的方式加载关联：

* Nested Select: By executing another mapped SQL statement that returns the complex type desired.
* 嵌套 Select 查询：通过执行另外一个 SQL 映射语句来加载期望的复杂类型。
* Nested Results: By using nested result mappings to deal with repeating subsets of joined results.
* 嵌套结果映射：使用嵌套的结果映射来处理连接结果的重复子集。

First, let's examine the properties of the element. As you'll see, it differs from a normal result mapping only by the select and resultMap attributes.

首先，先让我们来看看这个元素的属性。你将会发现，和普通的结果映射相比，它只在 select 和 resultMap 属性上有所不同。

**property** - The field or property to map the column result to. If a matching JavaBeans property exists for the given name, then that will be used. Otherwise, MyBatis will look for a field of the given name. In both cases you can use complex property navigation using the usual dot notation. For example, you can map to something simple like: username, or to something more complicated like: address.street.number.

映射到列结果的字段或属性。如果用来匹配的 JavaBean 存在给定名字的属性，那么它将会被使用。否则 MyBatis 将会寻找给定名称的字段。 无论是哪一种情形，你都可以使用通常的点式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上：“address.street.number”。

**javaType** - A fully qualified Java class name, or a type alias (see the table above for the list of built- in type aliases). MyBatis can usually figure out the type if you're mapping to a JavaBean. However, if you are mapping to a HashMap, then you should specify the javaType explicitly to ensure the desired behaviour.

一个 Java 类的完全限定名，或一个类型别名（关于内置的类型别名，可以参考上面的表格）。 如果你映射到一个 JavaBean，MyBatis 通常可以推断类型。然而，如果你映射到的是 HashMap，那么你应该明确地指定 javaType 来保证行为与期望的相一致。

**jdbcType** - The JDBC Type from the list of supported types that follows this table. The JDBC type is only required for nullable columns upon insert, update or delete. This is a JDBC requirement, not an MyBatis one. So even if you were coding JDBC directly, you'd need to specify this type – but only for nullable values.

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、更新和删除的且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，你需要对可能存在空值的列指定这个类型。

**typeHandler** - We discussed default type handlers previously in this documentation. Using this property you can override the default type handler on a mapping-by-mapping basis. The value is either a fully qualified class name of a TypeHandler implementation, or a type alias.

我们在前面讨论过默认的类型处理器。使用这个属性，你可以覆盖默认的类型处理器。 这个属性值是一个类型处理器实现类的完全限定名，或者是类型别名。

**Nested Select for Association 关联的嵌套 Select 查询**

**column** - The column name from the database, or the aliased column label that holds the value that will be passed to the nested statement as an input parameter. This is the same string that would normally be passed to resultSet.getString(columnName). Note: To deal with composite keys, you can specify multiple column names to pass to the nested select statement by using the syntax column="{prop1=col1,prop2=col2}". This will cause prop1 and prop2 to be set against the parameter object for the target nested select statement.

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。 注意：在使用复合主键的时候，你可以使用 column="{prop1=col1,prop2=col2}" 这样的语法来指定多个传递给嵌套 Select 查询语句的列名。这会使得 prop1 和 prop2 作为参数对象，被设置为对应嵌套 Select 语句的参数。

**select** - The ID of another mapped statement that will load the complex type required by this property mapping. The values retrieved from columns specified in the column attribute will be passed to the target select statement as parameters. A detailed example follows this table. Note: To deal with composite keys, you can specify multiple column names to pass to the nested select statement by using the syntax column="{prop1=col1,prop2=col2}". This will cause prop1 and prop2 to be set against the parameter object for the target nested select statement.

用于加载复杂类型属性的映射语句的 ID，它会从 column 属性指定的列中检索数据，作为参数传递给目标 select 语句。 具体请参考下面的例子。注意：在使用复合主键的时候，你可以使用 column="{prop1=col1,prop2=col2}" 这样的语法来指定多个传递给嵌套 Select 查询语句的列名。这会使得 prop1 和 prop2 作为参数对象，被设置为对应嵌套 Select 语句的参数。

**fetchType** - Optional. Valid values are lazy and eager. If present, it supersedes the global configuration parameter lazyLoadingEnabled for this mapping.

可选的。有效值为 lazy 和 eager。 指定属性后，将在映射中忽略全局配置参数 lazyLoadingEnabled，使用属性的值。

For example: 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>

That's it. We have two select statements: one to load the Blog, the other to load the Author, and the Blog's resultMap describes that the selectAuthor statement should be used to load its author property.

就是这么简单。我们有两个 select 查询语句：一个用来加载博客（Blog），另外一个用来加载作者（Author），而且博客的结果映射描述了应该使用 selectAuthor 语句加载它的 author 属性。

All other properties will be loaded automatically assuming their column and property names match.

其它所有的属性将会被自动加载，只要它们的列名和属性名相匹配。

While this approach is simple, it will not perform well for large data sets or lists. This problem is known as the "N+1 Selects Problem". In a nutshell, the N+1 selects problem is caused like this:

这种方式虽然很简单，但在大型数据集或大型数据表上表现不佳。这个问题被称为“N+1 查询问题”。 概括地讲，N+1 查询问题是这样子的：

* You execute a single SQL statement to retrieve a list of records (the "+1"). 你执行了一个单独的 SQL 语句来获取结果的一个列表（就是“+1”）。
* For each record returned, you execute a select statement to load details for each (the "N"). 对列表返回的每条记录，你执行一个 select 查询语句来为每条记录加载详细信息（就是“N”）。

This problem could result in hundreds or thousands of SQL statements to be executed. This is not always desirable.

这个问题会导致成百上千的 SQL 语句被执行。有时候，我们不希望产生这样的后果。

The upside is that MyBatis can lazy load such queries, thus you might be spared the cost of these statements all at once. However, if you load such a list and then immediately iterate through it to access the nested data, you will invoke all of the lazy loads, and thus performance could be very bad.

好消息是，MyBatis 能够对这样的查询进行延迟加载，因此可以将大量语句同时运行的开销分散开来。 然而，如果你加载记录列表之后立刻就遍历列表以获取嵌套的数据，就会触发所有的延迟加载查询，性能可能会变得很糟糕。

And so, there is another way.

所以还有另外一种方法。

**Nested Results for Association 关联的嵌套结果映射**

**resultMap** - This is the ID of a ResultMap that can map the nested results of this association into an appropriate object graph. This is an alternative to using a call to another select statement. It allows you to join multiple tables together into a single ResultSet. Such a ResultSet will contain duplicated, repeating groups of data that needs to be decomposed and mapped properly to a nested object graph. To facilitate this, MyBatis lets you "chain" result maps together, to deal with the nested results. An example will be far easier to follow, and one follows this table.

结果映射的 ID，可以将此关联的嵌套结果集映射到一个合适的对象树中。 它可以作为使用额外 select 语句的替代方案。它可以将多表连接操作的结果映射成一个单一的 ResultSet。这样的 ResultSet 有部分数据是重复的。 为了将结果集正确地映射到嵌套的对象树中, MyBatis 允许你“串联”结果映射，以便解决嵌套结果集的问题。使用嵌套结果映射的一个例子在表格以后。

**columnPrefix** - When joining multiple tables, you would have to use column alias to avoid duplicated column names in the ResultSet. Specifying columnPrefix allows you to map such columns to an external resultMap. Please see the example explained later in this section.

当连接多个表时，你可能会不得不使用列别名来避免在 ResultSet 中产生重复的列名。指定 columnPrefix 列名前缀允许你将带有这些前缀的列映射到一个外部的结果映射中。 详细说明请参考后面的例子。

**notNullColumn** - By default a child object is created only if at least one of the columns mapped to the child's properties is non null. With this attribute you can change this behaviour by specifiying which columns must have a value so MyBatis will create a child object only if any of those columns is not null. Multiple column names can be specified using a comma as a separator. Default value: unset.

默认情况下，在至少一个被映射到属性的列不为空时，子对象才会被创建。 你可以在这个属性上指定非空的列来改变默认行为，指定后，Mybatis 将只在这些列非空时才创建一个子对象。可以使用逗号分隔来指定多个列。默认值：未设置（unset）。

**autoMapping** - If present, MyBatis will enable or disable automapping when mapping the result to this property. This attribute overrides the global autoMappingBehavior. Note that it has no effect on an external resultMap, so it is pointless to use it with select or resultMap attribute. Default value: unset.

如果设置这个属性，MyBatis 将会为本结果映射开启或者关闭自动映射。 这个属性会覆盖全局的属性 autoMappingBehavior。注意，本属性对外部的结果映射无效，所以不能搭配 select 或 resultMap 元素使用。默认值：未设置（unset）。

You've already seen a very complicated example of nested associations above. The following is a far simpler example to demonstrate how this works. Instead of executing a separate statement, we'll join the Blog and Author tables together, like so:

之前，你已经看到了一个非常复杂的嵌套关联的例子。 下面的例子则是一个非常简单的例子，用于演示嵌套结果映射如何工作。 现在我们将博客表和作者表连接在一起，而不是执行一个独立的查询语句，就像这样：

<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>

Notice the join, as well as the care taken to ensure that all results are aliased with a unique and clear name. This makes mapping far easier. Now we can map the results:

注意查询中的连接，以及为确保结果能够拥有唯一且清晰的名字，我们设置的别名。 这使得进行映射非常简单。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" 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>

In the example above you can see at the Blog's "author" association delegates to the "authorResult" resultMap to load the Author instance.

在上面的例子中，你可以看到，博客（Blog）作者（author）的关联元素委托名为 “authorResult” 的结果映射来加载作者对象的实例。

Very Important: id elements play a very important role in Nested Result mapping. You should always specify one or more properties that can be used to uniquely identify the results. The truth is that MyBatis will still work if you leave it out, but at a severe performance cost. Choose as few properties as possible that can uniquely identify the result. The primary key is an obvious choice (even if composite).

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。 虽然，即使不指定这个属性，MyBatis 仍然可以工作，但是会产生严重的性能问题。 只需要指定可以唯一标识结果的最少属性。显然，你可以选择主键（复合主键也可以）。

Now, the above example used an external resultMap element to map the association. This makes the Author resultMap reusable. However, if you have no need to reuse it, or if you simply prefer to co-locate your result mappings into a single descriptive resultMap, you can nest the association result mappings. Here's the same example using this approach:

现在，上面的示例使用了外部的结果映射元素来映射关联。这使得 Author 的结果映射可以被重用。 然而，如果你不打算重用它，或者你更喜欢将你所有的结果映射放在一个具有描述性的结果映射元素中。 你可以直接将结果映射作为子元素嵌套在内。这里给出使用这种方式的等效例子：

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

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

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

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

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

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

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

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

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

</association>

</resultMap>

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

那如果博客（blog）有一个共同作者（co-author）该怎么办？select 语句看起来会是这样的：

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

select

B.id as blog\_id,

B.title as blog\_title,

A.id as author\_id,

A.username as author\_username,

A.password as author\_password,

A.email as author\_email,

A.bio as author\_bio,

CA.id as co\_author\_id,

CA.username as co\_author\_username,

CA.password as co\_author\_password,

CA.email as co\_author\_email,

CA.bio as co\_author\_bio

from Blog B

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

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

where B.id = #{id}

</select>

Recall that the resultMap for Author is defined as follows.

回忆一下，Author 的结果映射定义如下：

<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>

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

由于结果中的列名与结果映射中的列名不同。你需要指定 columnPrefix 以便重复使用该结果映射来映射 co-author 的结果。

<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>

**Multiple ResultSets for Association 关联的多结果集（ResultSet）**

**column** - When using multiple resultset this attribute specifies the columns (separated by commas) that will be correlated with the foreignColumn to identify the parent and the child of a relationship.

当使用多个结果集时，该属性指定结果集中用于与 foreignColumn 匹配的列（多个列名以逗号隔开），以识别关系中的父类型与子类型。

**foreignColumn** - Identifies the name of the columns that contains the foreign keys which values will be matched against the values of the columns specified in the column attibute of the parent type.

指定外键对应的列名，指定的列将与父类型中 column 的给出的列进行匹配。

**resultSet** - Identifies the name of the result set where this complex type will be loaded from.

指定用于加载复杂类型的结果集名字。

Starting from version 3.2.3 MyBatis provides yet another way to solve the N+1 problem.

从版本 3.2.3 开始，MyBatis 提供了另一种解决 N+1 查询问题的方法。

Some databases allow stored procedures to return more than one resultset or execute more than one statement at once and return a resultset per each one. This can be used to hit the database just once and return related data without using a join.

某些数据库允许存储过程返回多个结果集，或一次性执行多个语句，每个语句返回一个结果集。 我们可以利用这个特性，在不使用连接的情况下，只访问数据库一次就能获得相关数据。

In the example, the stored procedure executes the following queries and returns two result sets. The first will contain Blogs and the second Authors.

在例子中，存储过程执行下面的查询并返回两个结果集。第一个结果集会返回博客（Blog）的结果，第二个则返回作者（Author）的结果。

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

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

A name must be given to each result set by adding a resultSets attribute to the mapped statement with a list of names separated by commas.

在映射语句中，必须通过 resultSets 属性为每个结果集指定一个名字，多个名字使用逗号隔开。

<select id="selectBlog" resultSets="blogs,authors" resultMap="blogResult" statementType="CALLABLE">

{call getBlogsAndAuthors(#{id,jdbcType=INTEGER,mode=IN})}

</select>

Now we can specify that the data to fill the "author" association comes in the "authors" result set:

现在我们可以指定使用 “authors” 结果集的数据来填充 “author” 关联：

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

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

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

<association property="author" javaType="Author" resultSet="authors" column="author\_id" foreignColumn="id">

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

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

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

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

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

</association>

</resultMap>

You've seen above how to deal with a "has one" type association. But what about "has many"? That's the subject of the next section.

你已经在上面看到了如何处理“有一个”类型的关联。但是该怎么处理“有很多个”类型的关联呢？这就是我们接下来要介绍的。

**collection 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>

The collection element works almost identically to the association. In fact, it's so similar, to document the similarities would be redundant. So let's focus on the differences.

集合元素和关联元素几乎是一样的，它们相似的程度之高，以致于没有必要再介绍集合元素的相似部分。 所以让我们来关注它们的不同之处吧。

To continue with our example above, a Blog only had one Author. But a Blog has many Posts. On the blog class, this would be represented by something like:

我们来继续上面的示例，一个博客（Blog）只有一个作者（Author)。但一个博客有很多文章（Post)。 在博客类中，这可以用下面的写法来表示：

private List<Post> posts;

To map a set of nested results to a List like this, we use the collection element. Just like the association element, we can use a nested select, or nested results from a join.

要像上面这样，映射嵌套结果集合到一个 List 中，可以使用集合元素。 和关联元素一样，我们可以使用嵌套 Select 查询，或基于连接的嵌套结果映射集合。

**Nested Select for Collection 集合的嵌套 Select 查询**

First, let's look at using a nested select to load the Posts for the Blog.

首先，让我们看看如何使用嵌套 Select 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>

There are a number things you'll notice immediately, but for the most part it looks very similar to the association element we learned about above. First, you'll notice that we're using the collection element. Then you'll notice that there's a new "ofType" attribute. This attribute is necessary to distinguish between the JavaBean (or field) property type and the type that the collection contains. So you could read the following mapping like this:

你可能会立刻注意到几个不同，但大部分都和我们上面学习过的关联元素非常相似。 首先，你会注意到我们使用的是集合元素。 接下来你会注意到有一个新的 “ofType” 属性。这个属性非常重要，它用来将 JavaBean（或字段）属性的类型和集合存储的类型区分开来。 所以你可以按照下面这样来阅读映射：

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

Read as: "A collection of posts in an ArrayList of type Post."

读作： “posts 是一个存储 Post 的 ArrayList 集合”

The javaType attribute is really unnecessary, as MyBatis will figure this out for you in most cases. So you can often shorten this down to simply:

在一般情况下，MyBatis 可以推断 javaType 属性，因此并不需要填写。所以很多时候你可以简略成：

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

**Nested Results for Collection 集合的嵌套结果映射**

By this point, you can probably guess how nested results for a collection will work, because it's exactly the same as an association, but with the same addition of the ofType attribute applied.

现在你可能已经猜到了集合的嵌套结果映射是怎样工作的——除了新增的 “ofType” 属性，它和关联的完全相同。

First, let's look at the SQL:

首先, 让我们看看对应的 SQL 语句：

<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>

Again, we've joined the Blog and Post tables, and have taken care to ensure quality result column labels for simple mapping. Now mapping a Blog with its collection of Post mappings is as simple as:

我们再次连接了博客表和文章表，并且为每一列都赋予了一个有意义的别名，以便映射保持简单。 要映射博客里面的文章集合，就这么简单：

<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>

Again, remember the importance of the id elements here, or read the association section above if you haven't already.

再提醒一次，要记得上面 id 元素的重要性，如果你不记得了，请阅读关联部分的相关部分。

Also, if you prefer the longer form that allows for more reusability of your result maps, you can use the following alternative mapping:

如果你喜欢更详略的、可重用的结果映射，你可以使用下面的等价形式：

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

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

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

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

</resultMap>

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

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

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

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

</resultMap>

**Multiple ResultSets for Collection 集合的多结果集**

As we did for the association, we can call a stored procedure that executes two queries and returns two result sets, one with Blogs and another with Posts:

像关联元素那样，我们可以通过执行存储过程实现，它会执行两个查询并返回两个结果集，一个是博客的结果集，另一个是文章的结果集：

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

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

A name must be given to each result set by adding a resultSets attribute to the mapped statement with a list of names separated by commas.

在映射语句中，必须通过 resultSets 属性为每个结果集指定一个名字，多个名字使用逗号隔开。

<select id="selectBlog" resultSets="blogs,posts" resultMap="blogResult">

{call getBlogsAndPosts(#{id,jdbcType=INTEGER,mode=IN})}

</select>

We specify that the "posts" collection will be filled out of data contained in the result set named "posts":

我们指定 “posts” 集合将会使用存储在 “posts” 结果集中的数据进行填充：

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

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

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

<collection property="posts" ofType="Post" resultSet="posts" column="id" foreignColumn="blog\_id">

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

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

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

</collection>

</resultMap>

**NOTE** There's no limit to the depth, breadth or combinations of the associations and collections that you map. You should keep performance in mind when mapping them. Unit testing and performance testing of your application goes a long way toward discovering the best approach for your application. The nice thing is that MyBatis lets you change your mind later, with very little (if any) impact to your code.

注意 对关联或集合的映射，并没有深度、广度或组合上的要求。但在映射时要留意性能问题。 在探索最佳实践的过程中，应用的单元测试和性能测试会是你的好帮手。 而 MyBatis 的好处在于，可以在不对你的代码引入重大变更（如果有）的情况下，允许你之后改变你的想法。

Advanced association and collection mapping is a deep subject. Documentation can only get you so far. With a little practice, it will all become clear very quickly.

高级关联和集合映射是一个深度话题。文档的介绍只能到此为止。配合少许的实践，你会很快了解全部的用法。

**discriminator 鉴别器**

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

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

</discriminator>

Sometimes a single database query might return result sets of many different (but hopefully somewhat related) data types. The discriminator element was designed to deal with this situation, and others, including class inheritance hierarchies. The discriminator is pretty simple to understand, as it behaves much like a switch statement in Java.

有时候，一个数据库查询可能会返回多个不同的结果集（但总体上还是有一定的联系的）。 鉴别器（discriminator）元素就是被设计来应对这种情况的，另外也能处理其它情况，例如类的继承层次结构。 鉴别器的概念很好理解——它很像 Java 语言中的 switch 语句。

A discriminator definition specifies column and javaType attributes. The column is where MyBatis will look for the value to compare. The javaType is required to ensure the proper kind of equality test is performed (although String would probably work for almost any situation). For example:

一个鉴别器的定义需要指定 column 和 javaType 属性。column 指定了 MyBatis 查询被比较值的地方。 而 javaType 用来确保使用正确的相等测试（虽然很多情况下字符串的相等测试都可以工作）。例如：

<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>

In this example, MyBatis would retrieve each record from the result set and compare its vehicle type value. If it matches any of the discriminator cases, then it will use the resultMap specified by the case. This is done exclusively, so in other words, the rest of the resultMap is ignored (unless it is extended, which we talk about in a second). If none of the cases match, then MyBatis simply uses the resultMap as defined outside of the discriminator block. So, if the carResult was declared as follows:

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 type 值。 如果它匹配任意一个鉴别器的 case，就会使用这个 case 指定的结果映射。 这个过程是互斥的，也就是说，剩余的结果映射将被忽略（除非它是扩展的，我们将在稍后讨论它）。 如果不能匹配任何一个 case，MyBatis 就只会使用鉴别器块外定义的结果映射。 所以，如果 carResult 的声明如下：

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

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

</resultMap>

Then ONLY the doorCount property would be loaded. This is done to allow completely independent groups of discriminator cases, even ones that have no relationship to the parent resultMap. In this case we do of course know that there's a relationship between cars and vehicles, as a Car is-a Vehicle. Therefore, we want the rest of the properties loaded too. One simple change to the resultMap and we're set to go.

那么只有 doorCount 属性会被加载。这是为了即使鉴别器的 case 之间都能分为完全独立的一组，尽管和父结果映射可能没有什么关系。在上面的例子中，我们当然知道 cars 和 vehicles 之间有关系，也就是 Car 是一个 Vehicle。因此，我们希望剩余的属性也能被加载。而这只需要一个小修改。

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

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

</resultMap>

Now all of the properties from both the vehicleResult and carResult will be loaded.

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

Once again though, some may find this external definition of maps somewhat tedious. Therefore there's an alternative syntax for those that prefer a more concise mapping style. For example:

可能有人又会觉得映射的外部定义有点太冗长了。 因此，对于那些更喜欢简洁的映射风格的人来说，还有另一种语法可以选择。例如：

<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>

**NOTE** Remember that these are all Result Maps, and if you don't specify any results at all, then MyBatis will automatically match up columns and properties for you. So most of these examples are more verbose than they really need to be. That said, most databases are kind of complex and it's unlikely that we'll be able to depend on that for all cases.

提示 请注意，这些都是结果映射，如果你完全不设置任何的 result 元素，MyBatis 将为你自动匹配列和属性。所以上面的例子大多都要比实际的更复杂。 这也表明，大多数数据库的复杂度都比较高，我们不太可能一直依赖于这种机制。

**Auto-mapping 自动映射**

As you have already seen in the previous sections, in simple cases MyBatis can auto-map the results for you and in others you will need to build a result map. But as you will see in this section you can also mix both strategies. Let's have a deeper look at how auto-mapping works.

正如你在前面一节看到的，在简单的场景下，MyBatis 可以为你自动映射查询结果。但如果遇到复杂的场景，你需要构建一个结果映射。 但是在本节中，你将看到，你可以混合使用这两种策略。让我们深入了解一下自动映射是怎样工作的。

When auto-mapping results MyBatis will get the column name and look for a property with the same name ignoring case. That means that if a column named ID and property named id are found, MyBatis will set the id property with the ID column value.

当自动映射查询结果时，MyBatis 会获取结果中返回的列名并在 Java 类中查找相同名字的属性（忽略大小写）。 这意味着如果发现了 ID 列和 id 属性，MyBatis 会将列 ID 的值赋给 id 属性。

Usually database columns are named using uppercase letters and underscores between words and java properties often follow the camelcase naming covention. To enable the auto-mapping between them set the setting mapUnderscoreToCamelCase to true.

通常数据库列使用大写字母组成的单词命名，单词间用下划线分隔；而 Java 属性一般遵循驼峰命名法约定。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。

Auto-mapping works even when there is an specific result map. When this happens, for each result map, all columns that are present in the ResultSet that have not a manual mapping will be auto-mapped, then manual mappings will be processed. In the following sample id and userName columns will be auto-mapped and hashed\_password column will be mapped.

甚至在提供了结果映射后，自动映射也能工作。在这种情况下，对于每一个结果映射，在 ResultSet 出现的列，如果没有设置手动映射，将被自动映射。在自动映射处理完毕后，再处理手动映射。 在下面的例子中，id 和 userName 列将被自动映射，hashed\_password 列将根据配置进行映射。

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

select

user\_id as "id",

user\_name as "userName",

hashed\_password

from some\_table

where id = #{id}

</select>

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

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

</resultMap>

There are three auto-mapping levels: 有三种自动映射等级：

* NONE - disables auto-mapping. Only manually mapped properties will be set. 禁用自动映射。仅对手动映射的属性进行映射。
* PARTIAL - will auto-map results except those that have nested result mappings defined inside (joins). 对除在内部定义了嵌套结果映射（也就是连接的属性）以外的属性进行映射
* FULL - auto-maps everything. 自动映射所有属性。

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

默认值是 PARTIAL，这是有原因的。当对连接查询的结果使用 FULL 时，连接查询会在同一行中获取多个不同实体的数据，因此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>

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

在该结果映射中，Blog 和 Author 均将被自动映射。但是注意 Author 有一个 id 属性，在 ResultSet 中也有一个名为 id 的列，所以 Author 的 id 将填入 Blog 的 id，这可不是你期望的行为。 所以，要谨慎使用 FULL。

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

无论设置的自动映射等级是哪种，你都可以通过在结果映射上设置 autoMapping 属性来为指定的结果映射设置启用/禁用自动映射。

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

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

</resultMap>

**cache 缓存**

MyBatis includes a powerful transactional query caching feature which is very configurable and customizable. A lot of changes have been made in the MyBatis 3 cache implementation to make it both more powerful and far easier to configure.

MyBatis 内置了一个强大的事务性查询缓存机制，它可以非常方便地配置和定制。 为了使它更加强大而且易于配置，我们对 MyBatis 3 中的缓存实现进行了许多改进。

By default, just local session caching is enabled that is used solely to cache data for the duration of a session. To enable a global second level of caching you simply need to add one line to your SQL Mapping file:

默认情况下，只启用了本地的会话缓存，它仅仅对一个会话中的数据进行缓存。 要启用全局的二级缓存，只需要在你的 SQL 映射文件中添加一行：

<cache/>

Literally that's it. The effect of this one simple statement is as follows:

基本上就是这样。这个简单语句的效果如下:

* All results from select statements in the mapped statement file will be cached.
* 映射语句文件中的所有 select 语句的结果将会被缓存。
* All insert, update and delete statements in the mapped statement file will flush the cache.
* 映射语句文件中的所有 insert、update 和 delete 语句会刷新缓存。
* The cache will use a Least Recently Used (LRU) algorithm for eviction.
* 缓存会使用最近最少使用算法（LRU, Least Recently Used）算法来清除不需要的缓存。
* The cache will not flush on any sort of time based schedule (i.e. no Flush Interval).
* 缓存不会定时进行刷新（也就是说，没有刷新间隔）。
* The cache will store 1024 references to lists or objects (whatever the query method returns).
* 缓存会保存列表或对象（无论查询方法返回哪种）的 1024 个引用。
* The cache will be treated as a read/write cache, meaning objects retrieved are not shared and can be safely modified by the caller, without interfering with other potential modifications by other callers or threads.
* 缓存会被视为读/写缓存，这意味着获取到的对象并不是共享的，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。

**NOTE** The cache will only apply to statements declared in the mapping file where the cache tag is located. If you are using the Java API in conjunction with the XML mapping files, then statements declared in the companion interface will not be cached by default. You will need to refer to the cache region using the @CacheNamespaceRef annotation.

提示 缓存只作用于 cache 标签所在的映射文件中的语句。如果你混合使用 Java API 和 XML 映射文件，在共用接口中的语句将不会被默认缓存。你需要使用 @CacheNamespaceRef 注解指定缓存作用域。

All of these properties are modifiable through the attributes of the cache element. For example:

这些属性可以通过 cache 元素的属性来修改。比如：

<cache

eviction="FIFO"

flushInterval="60000"

size="512"

readOnly="true"/>

This more advanced configuration creates a FIFO cache that flushes once every 60 seconds, stores up to 512 references to result objects or lists, and objects returned are considered read-only, thus modifying them could cause conflicts between callers in different threads.

这个更高级的配置创建了一个 FIFO 缓存，每隔 60 秒刷新，最多可以存储结果对象或列表的 512 个引用，而且返回的对象被认为是只读的，因此对它们进行修改可能会在不同线程中的调用者产生冲突。

The available eviction policies available are:

可用的清除策略有：

* LRU – Least Recently Used: Removes objects that haven't been used for the longst period of time.
* 最近最少使用：移除最长时间不被使用的对象。
* FIFO – First In First Out: Removes objects in the order that they entered the cache.
* 先进先出：按对象进入缓存的顺序来移除它们。
* SOFT – Soft Reference: Removes objects based on the garbage collector state and the rules of Soft References.
* 软引用：基于垃圾回收器状态和软引用规则移除对象。
* WEAK – Weak Reference: More aggressively removes objects based on the garbage collector state and rules of Weak References.
* 弱引用：更积极地基于垃圾收集器状态和弱引用规则移除对象。

The default is LRU.

默认的清除策略是 LRU。

The flushInterval can be set to any positive integer and should represent a reasonable amount of time specified in milliseconds. The default is not set, thus no flush interval is used and the cache is only flushed by calls to statements.

flushInterval（刷新间隔）属性可以被设置为任意的正整数，设置的值应该是一个以毫秒为单位的合理时间量。 默认情况是不设置，也就是没有刷新间隔，缓存仅仅会在调用语句时刷新。

The size can be set to any positive integer, keep in mind the size of the objects your caching and the available memory resources of your environment. The default is 1024.

size（引用数目）属性可以被设置为任意正整数，要注意欲缓存对象的大小和运行环境中可用的内存资源。默认值是 1024。

The readOnly attribute can be set to true or false. A read-only cache will return the same instance of the cached object to all callers. Thus such objects should not be modified. This offers a significant performance advantage though. A read-write cache will return a copy (via serialization) of the cached object. This is slower, but safer, and thus the default is false.

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。这就提供了可观的性能提升。而可读写的缓存会（通过序列化）返回缓存对象的拷贝。 速度上会慢一些，但是更安全，因此默认值是 false。

**NOTE** Second level cache is transactional. That means that it is updated when a SqlSession finishes with commit or when it finishes with rollback but no inserts/deletes/updates with flushCache=true where executed.

提示 二级缓存是事务性的。这意味着，当 SqlSession 完成并提交时，或是完成并回滚，但没有执行 flushCache=true 的 insert/delete/update 语句时，缓存会获得更新。

**Using a Custom Cache 使用自定义缓存**

In addition to customizing the cache in these ways, you can also completely override the cache behavior by implementing your own cache, or creating an adapter to other 3rd party caching solutions.

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

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

This example demonstrates how to use a custom cache implementation. The class specified in the type attribute must implement the org.apache.ibatis.cache.Cache interface and provide a constructor that gets an String id as an argument. This interface is one of the more complex in the MyBatis framework, but simple given what it does.

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，且提供一个接受 String 参数作为 id 的构造器。 这个接口是 MyBatis 框架中许多复杂的接口之一，但是行为却非常简单。

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();

}

To configure your cache, simply add public JavaBeans properties to your Cache implementation, and pass properties via the cache Element, for example, the following would call a method called setCacheFile(String file) on your Cache implementation:

为了对你的缓存进行配置，只需要简单地在你的缓存实现中添加公有的 JavaBean 属性，然后通过 cache 元素传递属性值，例如，下面的例子将在你的缓存实现上调用一个名为 setCacheFile(String file) 的方法：

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

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

</cache>

You can use JavaBeans properties of all simple types, MyBatis will do the conversion. And you can specify a placeholder(e.g. ${cache.file}) to replace value defined at configuration properties.

你可以使用所有简单类型作为 JavaBean 属性的类型，MyBatis 会进行转换。 你也可以使用占位符（如 ${cache.file}），以便替换成在配置文件属性中定义的值。

Since 3.4.2, the MyBatis has been supported to call an initialization method after it's set all properties. If you want to use this feature, please implements the org.apache.ibatis.builder.InitializingObject interface on your custom cache class.

从版本 3.4.2 开始，MyBatis 已经支持在所有属性设置完毕之后，调用一个初始化方法。 如果想要使用这个特性，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;

}

**NOTE** Settings of cache (like eviction strategy, read write..etc.) in section above are not applied when using Custom Cache.

提示 上一节中对缓存的配置（如清除策略、可读或可读写等），不能应用于自定义缓存。

It's important to remember that a cache configuration and the cache instance are bound to the namespace of the SQL Map file. Thus, all statements in the same namespace as the cache are bound by it. Statements can modify how they interact with the cache, or exclude themselves completely by using two simple attributes on a statement-by-statement basis. By default, statements are configured like this:

请注意，缓存的配置和缓存实例会被绑定到 SQL 映射文件的命名空间中。 因此，同一命名空间中的所有语句和缓存将通过命名空间绑定在一起。 每条语句可以自定义与缓存交互的方式，或将它们完全排除于缓存之外，这可以通过在每条语句上使用两个简单属性来达成。 默认情况下，语句会这样来配置：

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

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

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

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

Since that's the default, you obviously should never explicitly configure a statement that way. Instead, only set the flushCache and useCache attributes if you want to change the default behavior. For example, in some cases you may want to exclude the results of a particular select statement from the cache, or you might want a select statement to flush the cache. Similarly, you may have some update statements that don't need to flush the cache upon execution.

鉴于这是默认行为，显然你永远不应该以这样的方式显式配置一条语句。但如果你想改变默认的行为，只需要设置 flushCache 和 useCache 属性。比如，某些情况下你可能希望特定 select 语句的结果排除于缓存之外，或希望一条 select 语句清空缓存。类似地，你可能希望某些 update 语句执行时不要刷新缓存。

**cache-ref**

Recall from the previous section that only the cache for this particular namespace will be used or flushed for statements within the same namespace. There may come a time when you want to share the same cache configuration and instance between namespaces. In such cases you can reference another cache by using the cache-ref element.

回想一下上一节的内容，对某一命名空间的语句，只会使用该命名空间的缓存进行缓存或刷新。 但你可能会想要在多个命名空间中共享相同的缓存配置和实例。要实现这种需求，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"/>

**Dynamic SQL 动态 SQL**

One of the most powerful features of MyBatis has always been its Dynamic SQL capabilities. If you have any experience with JDBC or any similar framework, you understand how painful it is to conditionally concatenate strings of SQL together, making sure not to forget spaces or to omit a comma at the end of a list of columns. Dynamic SQL can be downright painful to deal with.

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。

While working with Dynamic SQL will never be a party, MyBatis certainly improves the situation with a powerful Dynamic SQL language that can be used within any mapped SQL statement.

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。

The Dynamic SQL elements should be familiar to anyone who has used JSTL or any similar XML based text processors. In previous versions of MyBatis, there were a lot of elements to know and understand. MyBatis 3 greatly improves upon this, and now there are less than half of those elements to work with. MyBatis employs powerful OGNL based expressions to eliminate most of the other elements:

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**

The most common thing to do in dynamic SQL is conditionally include a part of a where clause. For example:

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>

This statement would provide an optional text search type of functionality. If you passed in no title, then all active Blogs would be returned. But if you do pass in a title, it will look for a title like that (for the keen eyed, yes in this case your parameter value would need to include any masking or wildcard characters).

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）。

What if we wanted to optionally search by title and author? First, I’d change the name of the statement to make more sense. Then simply add another condition.

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**

Sometimes we don’t want all of the conditionals to apply, instead we want to choose only one case among many options. Similar to a switch statement in Java, MyBatis offers a choose element.

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。

Let’s use the example above, but now let’s search only on title if one is provided, then only by author if one is provided. If neither is provided, let’s only return featured blogs (perhaps a strategically list selected by administrators, instead of returning a huge meaningless list of random blogs).

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**

The previous examples have been conveniently dancing around a notorious dynamic SQL challenge. Consider what would happen if we return to our "if" example, but this time we make "ACTIVE = 1" a dynamic condition as well.

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。

<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>

</select>

What happens if none of the conditions are met? You would end up with SQL that looked like this:

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

This would fail. What if only the second condition was met? You would end up with SQL that looked like this:

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’

This would also fail. This problem is not easily solved with conditionals, and if you’ve ever had to write it, then you likely never want to do so again.

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 has a simple answer that will likely work in 90% of the cases. And in cases where it doesn’t, you can customize it so that it does. With one simple change, everything works fine:

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>

The where element knows to only insert "WHERE" if there is any content returned by the containing tags. Furthermore, if that content begins with "AND" or "OR", it knows to strip it off.

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。

If the where element does not behave exactly as you like, you can customize it by defining your own trim element. For example, the trim equivalent to the where element is:

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>

The prefixOverrides attribute takes a pipe delimited list of text to override, where whitespace is relevant. The result is the removal of anything specified in the prefixOverrides attribute, and the insertion of anything in the prefix attribute.

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。

There is a similar solution for dynamic update statements called set. The set element can be used to dynamically include columns to update, and leave out others. For example:

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>

Here, the set element will dynamically prepend the SET keyword, and also eliminate any extraneous commas that might trail the value assignments after the conditions are applied.

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）

If you’re curious about what the equivalent trim element would look like, here it is:

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>

Notice that in this case we’re overriding a suffix, while we’re still appending a prefix.

注意这里我们删去的是后缀值，同时添加了前缀值。

**foreach**

Another common necessity for dynamic SQL is the need to iterate over a collection, often to build an IN condition. For example:

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>

The foreach element is very powerful, and allows you to specify a collection, declare item and index variables that can be used inside the body of the element. It also allows you to specify opening and closing strings, and add a separator to place in between iterations. The element is smart in that it won’t accidentally append extra separators.

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。

**NOTE** You can pass any Iterable object (for example List, Set, etc.), as well as any Map or Array object to foreach as collection parameter. When using an Iterable or Array, index will be the number of current iteration and value item will be the element retrieved in this iteration. When using a Map (or Collection of Map.Entry objects), index will be the key object and item will be the value object.

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。

This wraps up the discussion regarding the XML configuration file and XML mapping files. The next section will discuss the Java API in detail, so that you can get the most out of the mappings that you’ve created.

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**

The bind element lets you create a variable out of an OGNL expression and bind it to the context. For example:

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>

**Multi-db vendor support 多数据库支持**

If a databaseIdProvider was configured a "\_databaseId" variable is available for dynamic code, so you can build different statements depending on database vendor. Have a look at the following example:

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>

**Pluggable Scripting Languages For Dynamic SQL 动态 SQL 中的可插拔脚本语言**

Starting from version 3.2 MyBatis supports pluggable scripting languages, so you can plug a language driver and use that language to write your dynamic SQL queries.

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。

You can plug a language by implementing the following interface:

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);

}

Once you have your custom language driver you can set it to be the default by configuring it in the mybatis-config.xml file:

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>

Instead of changing the default, you can specify the language for an specific statement by adding the lang attribute as follows:

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>

Or, in the case you are using mappers, using the @Lang annotation:

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();

}

**NOTE** You can use Apache Velocity as your dynamic language. Have a look at the MyBatis-Velocity project for the details.

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。

All the xml tags you have seen in the previous sections are provided by the default MyBatis language that is provided by the driver org.apache.ibatis.scripting.xmltags.XmlLanguageDriver which is aliased as xml.

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。