## 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](https://github.com/mybatis/mybatis-3/tree/master/src/site)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](https://github.com/mybatis/mybatis-3/tree/master/src/site)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](http://www.mybatis.org/mybatis-3/getting-started.html)
* [Español](http://www.mybatis.org/mybatis-3/es/index.html)
* [日本語](http://www.mybatis.org/mybatis-3/ja/index.html)
* [한국어](http://www.mybatis.org/mybatis-3/ko/index.html)
* [简体中文](http://www.mybatis.org/mybatis-3/zh/index.html)

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](https://github.com/mybatis/mybatis-3/releases) file in the classpath.

要使用 MyBatis， 只需将 [mybatis-x.x.x.jar](https://github.com/mybatis/mybatis-3/releases) 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**

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 ...

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.

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。

| **Setting（设置属性）** | **Description（描述）** | **Valid Values(有效值)** | **Default（默认）** |
| --- | --- | --- | --- |
| 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。 | true | false | 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。 | true | false | 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)。 | true | false | 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）。 | true | false | 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。 | true | false | 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）。 | true | false | 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）。 | NONE, PARTIAL, FULL | 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) | NONE, WARNING, FAILING | 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。 | SIMPLE REUSE BATCH | 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。 | Any positive integer  任意整数 | 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。 | Any positive integer  任意整数 | 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。 | true | false | 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。 | true | false | 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。 | true | false | 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。 | SESSION | STATEMENT | 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。 | JdbcType enumeration. Most common are: NULL, VARCHAR and OTHER | 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。 | A method name list separated by commas | 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。 | A type alias or fully qualified class name. | 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） | A type alias or fully qualified class name. | 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。 | true | false | 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） | true | false | 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。 | Any String | 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。 | SLF4J | LOG4J | LOG4J2 | JDK\_LOGGING | COMMONS\_LOGGING | STDOUT\_LOGGING | NO\_LOGGING | 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。 | CGLIB | JAVASSIST | JAVASSIST (MyBatis 3.3 or above) |
| vfsImpl | Specifies VFS implementations  指定 VFS 的实现 | 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。 | Not set |
| 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） | true | false | 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） | A type alias or fully qualified class name. | Not set |

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 isdomain.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。

| **Alias** | **Mapped Type** |
| --- | --- |
| \_byte | byte |
| \_long | long |
| \_short | short |
| \_int | int |
| \_integer | int |
| \_double | double |
| \_float | float |
| \_boolean | boolean |
| string | String |
| byte | Byte |
| long | Long |
| short | Short |
| int | Integer |
| integer | Integer |
| double | Double |
| float | Float |
| boolean | Boolean |
| date | Date |
| decimal | BigDecimal |
| bigdecimal | BigDecimal |
| object | Object |
| map | Map |
| hashmap | HashMap |
| list | List |
| arraylist | ArrayList |
| collection | Collection |
| iterator | Iterator |

**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） 。

| **Type Handler** | **Java Types** | **JDBC Types** |
| --- | --- | --- |
| BooleanTypeHandler | java.lang.Boolean, boolean | Any compatible BOOLEAN  数据库兼容的 |
| ByteTypeHandler | java.lang.Byte, byte | Any compatible NUMERIC or BYTE |
| ShortTypeHandler | java.lang.Short, short | Any compatible NUMERIC or SMALLINT |
| IntegerTypeHandler | java.lang.Integer, int | Any compatible NUMERIC or INTEGER |
| LongTypeHandler | java.lang.Long, long | Any compatible NUMERIC or BIGINT |
| FloatTypeHandler | java.lang.Float, float | Any compatible NUMERIC or FLOAT |
| DoubleTypeHandler | java.lang.Double, double | Any compatible NUMERIC or DOUBLE |
| BigDecimalTypeHandler | java.math.BigDecimal | Any compatible NUMERIC or DECIMAL |
| StringTypeHandler | java.lang.String | CHAR, VARCHAR |
| ClobReaderTypeHandler | java.io.Reader | - |
| ClobTypeHandler | java.lang.String | CLOB, LONGVARCHAR |
| NStringTypeHandler | java.lang.String | NVARCHAR, NCHAR |
| NClobTypeHandler | java.lang.String | NCLOB |
| BlobInputStreamTypeHandler | java.io.InputStream | - |
| ByteArrayTypeHandler | byte[] | Any compatible byte stream type |
| BlobTypeHandler | byte[] | BLOB, LONGVARBINARY |
| DateTypeHandler | java.util.Date | TIMESTAMP |
| DateOnlyTypeHandler | java.util.Date | DATE |
| TimeOnlyTypeHandler | java.util.Date | TIME |
| SqlTimestampTypeHandler | java.sql.Timestamp | TIMESTAMP |
| SqlDateTypeHandler | java.sql.Date | DATE |
| SqlTimeTypeHandler | java.sql.Time | TIME |
| ObjectTypeHandler | Any | OTHER, or unspecified  OTHER 或未指定类型 |
| EnumTypeHandler | Enumeration Type | VARCHAR any string compatible type, as the code is stored (not index).  VARCHAR 或任何兼容的字符串类型，用以存储枚举的名称（而不是索引值） |
| EnumOrdinalTypeHandler | Enumeration Type | Any compatible NUMERIC or DOUBLE, as the position is stored (not the code itself).  任何兼容的 NUMERIC 或 DOUBLE 类型，存储枚举的序数值（而不是名称）。 |
| SqlxmlTypeHandler | java.lang.String | SQLXML |
| InstantTypeHandler | java.time.Instant | TIMESTAMP |
| LocalDateTimeTypeHandler | java.time.LocalDateTime | TIMESTAMP |
| LocalDateTypeHandler | java.time.LocalDate | DATE |
| LocalTimeTypeHandler | java.time.LocalTime | TIME |
| OffsetDateTimeTypeHandler | java.time.OffsetDateTime | TIMESTAMP |
| OffsetTimeTypeHandler | java.time.OffsetTime | TIME |
| ZonedDateTimeTypeHandler | java.time.ZonedDateTime | TIMESTAMP |
| YearTypeHandler | java.time.Year | INTEGER |
| MonthTypeHandler | java.time.Month | INTEGER |
| YearMonthTypeHandler | java.time.YearMonth | VARCHAR or LONGVARCHAR |
| JapaneseDateTypeHandler | java.time.chrono.JapaneseDate | DATE |

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 jdbcTypeattribute 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.

There are three build-in dataSource types (i.e. 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:

* driver – This is the fully qualified Java class of the JDBC driver (NOT of the DataSource class if your driver includes one).
* url – This is the JDBC URL for your database instance.
* username – The database username to log in with.
* password - The database password to log in with.
* defaultTransactionIsolationLevel – The default transaction isolation level for connections.

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

* driver.encoding=UTF8

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

**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.

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

* poolMaximumActiveConnections – This is the number of active (i.e. in use) connections that can exist at any given time. Default: 10
* poolMaximumIdleConnections – The number of idle connections that can exist at any given time.
* 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)
* 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)
* 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 poolMaximumIdleConnectionsand poolMaximumLocalBadConnectionTolerance. Default: 3 (Since: 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.
* 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.
* 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).

**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:

* 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.
* 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.

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

* env.encoding=UTF8

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

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

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:

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:

<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**

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:

<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:

<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".

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

public interface DatabaseIdProvider {

void setProperties(Properties p);

String getDatabaseId(DataSource dataSource) throws SQLException;

}

**mappers**

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:

<!-- Using classpath relative resources -->

<mappers>

<mapper resource="org/mybatis/builder/AuthorMapper.xml"/>

<mapper resource="org/mybatis/builder/BlogMapper.xml"/>

<mapper resource="org/mybatis/builder/PostMapper.xml"/>

</mappers>

<!-- Using url fully qualified paths -->

<mappers>

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

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

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

</mappers>

<!-- Using mapper interface classes -->

<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 -->

<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.

## Mapper XML Files

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.

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

* cache – Configuration of the cache for a given namespace.
* cache-ref – Reference to a cache configuration from another namespace.
* resultMap – The most complicated and powerful element that describes how to load your objects from the database result sets.
* ~~parameterMap – Deprecated! Old-school way to map parameters. Inline parameters are preferred and this element may be removed in the future. Not documented here.~~
* sql – A reusable chunk of SQL that can be referenced by other statements.
* insert – A mapped INSERT statement.
* 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.

### 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:

<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.

Notice the parameter notation:

#{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:

// Similar JDBC code, NOT MyBatis…

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.

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

<select

id="selectPerson"

parameterType="int"

parameterMap="deprecated"

resultType="hashmap"

resultMap="personResultMap"

flushCache="false"

useCache="true"

timeout="10"

fetchSize="256"

statementType="PREPARED"

resultSetType="FORWARD\_ONLY">

| Select Attributes | |
| --- | --- |
| **Attribute** | **Description** |
| id | A unique identifier in this namespace that can be used to reference this statement. |
| 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. |
| ~~parameterMap~~ | ~~This is a deprecated approach to referencing an external parameterMap. Use inline parameter mappings and the parameterType attribute.~~ |
| 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. |
| 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. |
| 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. |
| useCache | Setting this to true will cause the results of this statement to be cached in 2nd level cache. Default: true for select statements. |
| 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). |
| 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). |
| statementType | Any one of STATEMENT, PREPARED or CALLABLE. This causes MyBatis to use Statement, PreparedStatement or CallableStatement respectively. Default: PREPARED. |
| resultSetType | Any one of FORWARD\_ONLY|SCROLL\_SENSITIVE|SCROLL\_INSENSITIVE|DEFAULT(same as unset). Default is unset (driver dependent). |
| 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. |
| 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. |
| 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. |

### insert, update and delete

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

<insert

id="insertAuthor"

parameterType="domain.blog.Author"

flushCache="true"

statementType="PREPARED"

keyProperty=""

keyColumn=""

useGeneratedKeys=""

timeout="20">

<update

id="updateAuthor"

parameterType="domain.blog.Author"

flushCache="true"

statementType="PREPARED"

timeout="20">

<delete

id="deleteAuthor"

parameterType="domain.blog.Author"

flushCache="true"

statementType="PREPARED"

timeout="20">

| Insert, Update and Delete Attributes | |
| --- | --- |
| **Attribute** | **Description** |
| id | A unique identifier in this namespace that can be used to reference this statement. |
| 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. |
| ~~parameterMap~~ | ~~This is a deprecated approach to referencing an external parameterMap. Use inline parameter mappings and the parameterType attribute.~~ |
| 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. |
| 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). |
| statementType | Any one of STATEMENT, PREPARED or CALLABLE. This causes MyBatis to use Statement, PreparedStatement or CallableStatementrespectively. Default: 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. |
| 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. |
| 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. |
| 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. |

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

<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.

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:

<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.

<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.

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):

<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.

The selectKey element is described as follows:

<selectKey

keyProperty="id"

resultType="int"

order="BEFORE"

statementType="PREPARED">

| selectKey Attributes | |
| --- | --- |
| **Attribute** | **Description** |
| 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. |
| 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. |
| 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. |
| 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. |
| statementType | Same as above, MyBatis supports STATEMENT, PREPARED and CALLABLE statement types that map to Statement, PreparedStatement and CallableStatement respectively. |

### 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:

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

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

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

select

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

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

from some\_table t1

cross join some\_table t2

</select>

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

<sql id="sometable">

${prefix}Table

</sql>

<sql id="someinclude">

from

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

</sql>

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

select

field1, field2, field3

<include refid="someinclude">

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

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

</include>

</select>

### Parameters

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:

<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:

<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.

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

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

#{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.

**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.

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

#{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.

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

#{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.

#{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):

#{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.

#{firstName}

#{middleInitial,jdbcType=VARCHAR}

#{lastName}

#### String Substitution

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:

ORDER BY ${columnName}

Here MyBatis won't modify or escape the string.

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:

@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:

@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:

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.

**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.

### Result Maps

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.

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

<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:

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.

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

<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:

<!-- In Config XML file -->

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

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

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

select id, username, hashedPassword

from some\_table

where id = #{id}

</select>

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:

<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 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:

<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.

#### Advanced Result Maps

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.

For example, how would we map this statement?

<!-- Very Complex Statement -->

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

select

B.id as blog\_id,

B.title as blog\_title,

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

A.id as author\_id,

A.username as author\_username,

A.password as author\_password,

A.email as author\_email,

A.bio as author\_bio,

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

P.id as post\_id,

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

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

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

P.section as post\_section,

P.subject as post\_subject,

P.draft as draft,

P.body as post\_body,

C.id as comment\_id,

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

C.name as comment\_name,

C.comment as comment\_text,

T.id as tag\_id,

T.name as tag\_name

from Blog B

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

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

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

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

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

where B.id = #{id}

</select>

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.

<!-- Very Complex Result Map -->

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

<constructor>

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

</constructor>

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

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

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

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

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

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

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

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

</association>

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

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

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

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

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

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

</collection>

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

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

</collection>

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

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

</discriminator>

</collection>

</resultMap>

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

* constructor - used for injecting results into the constructor of a class upon instantiation
  + idArg - ID argument; flagging results as ID will help improve overall performance
  + arg - a normal result injected into the constructor
* id – an ID result; flagging results as ID will help improve overall performance
* result – a normal result injected into a field or JavaBean property
* association – a complex type association; many results will roll up into this type
  + nested result mappings – associations are resultMaps themselves, or can refer to one
* collection – a collection of complex types
  + nested result mappings – collections are resultMaps themselves, or can refer to one
* discriminator – uses a result value to determine which resultMap to use
  + case – a case is a result map based on some value
    - 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.

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

**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.

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

#### 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.).

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).

Each has a number of attributes:

| Id and Result Attributes | |
| --- | --- |
| **Attribute** | **Description** |
| 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. |
| 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). |
| 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. |
| 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. |
| 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. |

#### Supported JDBC Types

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

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

#### constructor

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.

Consider the following constructor:

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.

<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.

<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.

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

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

#### association

<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.

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

* Nested Select: By executing another mapped SQL statement that returns the complex type desired.
* Nested Results: By using nested result mappings to deal with repeating subsets of joined results.

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.

| **Attribute** | **Description** |
| --- | --- |
| 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. |
| 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. |
| 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. |
| 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. |

#### Nested Select for Association

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

For example:

<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.

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

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:

* You execute a single SQL statement to retrieve a list of records (the "+1").
* For each record returned, you execute a select statement to load details for each (the "N").

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

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.

And so, there is another way.

#### Nested Results for Association

| **Attribute** | **Description** |
| --- | --- |
| 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. |
| columnPrefix | When joining multiple tables, you would have to use column alias to avoid duplicated column names in the ResultSet. Specifying columnPrefix allows you to map such columns to an external resultMap. Please see the example explained later in this section. |
| notNullColumn | By default a child object is created only if at least one of the columns mapped to the child's properties is non null. With this attribute you can change this behaviour by specifiying which columns must have a value so MyBatis will create a child object only if any of those columns is not null. Multiple column names can be specified using a comma as a separator. Default value: unset. |
| autoMapping | If present, MyBatis will enable or disable 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. |

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:

<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:

<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.

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).

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:

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

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

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

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

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

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

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

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

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

</association>

</resultMap>

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

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

select

B.id as blog\_id,

B.title as blog\_title,

A.id as author\_id,

A.username as author\_username,

A.password as author\_password,

A.email as author\_email,

A.bio as author\_bio,

CA.id as co\_author\_id,

CA.username as co\_author\_username,

CA.password as co\_author\_password,

CA.email as co\_author\_email,

CA.bio as co\_author\_bio

from Blog B

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

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

where B.id = #{id}

</select>

Recall that the resultMap for Author is defined as follows.

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

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

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

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

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

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

</resultMap>

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

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

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

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

<association property="author"

resultMap="authorResult" />

<association property="coAuthor"

resultMap="authorResult"

columnPrefix="co\_" />

</resultMap>

#### Multiple ResultSets for Association

| **Attribute** | **Description** |
| --- | --- |
| 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. |
| 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 columnattibute of the parent type. |
| resultSet | Identifies the name of the result set where this complex type will be loaded from. |

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

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.

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

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.

<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:

<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.

#### collection

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

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

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

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

</collection>

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.

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:

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.

#### Nested Select for Collection

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

<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:

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

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

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:

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

#### Nested Results for Collection

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 ofTypeattribute applied.

First, let's look at the SQL:

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

select

B.id as blog\_id,

B.title as blog\_title,

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

P.id as post\_id,

P.subject as post\_subject,

P.body as post\_body,

from Blog B

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

where B.id = #{id}

</select>

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:

<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.

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

<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

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:

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.

<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":

<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.

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.

#### discriminator

<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.

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:

<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:

<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.

<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.

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:

<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.

### Auto-mapping

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.

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.

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.

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.

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

select

user\_id as "id",

user\_name as "userName",

hashed\_password

from some\_table

where id = #{id}

</select>

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

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

</resultMap>

There are three auto-mapping levels:

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

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

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

select

B.id,

B.title,

A.username,

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

where B.id = #{id}

</select>

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

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

</resultMap>

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

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

</resultMap>

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

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

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

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

</resultMap>

### cache

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.

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:

<cache/>

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

* All results from select statements in the mapped statement file will be cached.
* All insert, update and delete statements in the mapped statement file will flush the cache.
* The cache will use a Least Recently Used (LRU) algorithm for eviction.
* The cache will not flush on any sort of time based schedule (i.e. no Flush Interval).
* The cache will store 1024 references to lists or objects (whatever the query method returns).
* 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.

**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.

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

<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.

The available eviction policies available are:

* LRU – Least Recently Used: Removes objects that haven't been used for the longst period of time.
* FIFO – First In First Out: Removes objects in the order that they entered the cache.
* SOFT – Soft Reference: Removes objects based on the garbage collector state and the rules of Soft References.
* WEAK – Weak Reference: More aggressively removes objects based on the garbage collector state and rules of Weak References.

The default is 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.

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.

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.

**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.

#### Using a Custom Cache

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.

<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.

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:

<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](http://www.mybatis.org/mybatis-3/configuration.html#properties).

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.

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.

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:

<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.

#### 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.

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

**Dynamic 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.

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.

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:

* 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:

<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).

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.

<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.

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).

<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.

<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:

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:

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.

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:

<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.

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:

<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.

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:

<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.

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

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

...

</trim>

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

**foreach**

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

<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.

**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.

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.

**bind**

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

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

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

SELECT \* FROM BLOG

WHERE title LIKE #{pattern}

</select>

**Multi-db vendor support**

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:

<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**

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.

You can plug a language by implementing the following interface:

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:

<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:

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

SELECT \* FROM BLOG

</select>

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

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.

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