简介

什么是 MyBatis ？

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信息，将接口和 Java 的 POJOs(Plain Old Java Objects,普通的 Java 对象)映射成数据库中的记录。

帮助改进文档...

不管你以何种方式发现了文档的不足，或是丢失对某一特性的描述，那么你能做的最好的事情莫过于去研究它并把文档写出来。

该文档 xdoc 格式的源码文件可通过项目的 Git 代码库来获取。Fork 该源码库，做出更新，然后提交一个 pull request 吧。

你将成为本文档的最佳作者，MyBatis 的用户定会过来查阅的。

当前的国际化版本

MyBatis 的其他语言版本：

- English
- Español
- 日本語
- ____
- 简体中文

入门

安装

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

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>
```

从 XML 中构建 SqlSessionFactory

每个基于 MyBatis 的应用都是以一个 SqlSessionFactory 的实例为中心的。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。

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);
```

XML 配置文件（configuration XML）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>
```

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，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 mapper 映射器（这些 mapper 的 XML 文件包含了 SQL 代码和映射定义信息）。

不使用 XML 构建 SqlSessionFactory

如果你更愿意直接从 Java 程序而不是 XML 文件中创建 configuration，或者创建你自己的 configuration 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", transactionFactor
y, dataSource);

Configuration configuration = new Configuration(environment);

configuration.addMapper(BlogMapper.class);
```

```
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build
(configuration);
```

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了 XML 文件的依赖。不过，由于 Java 注解的一些限制加之某些 MyBatis 映射的复杂性，XML 映射对于大多数高级映射（比如：嵌套 Join 映射）来说仍然是必须的。有鉴于此，如果存在一个对等的 XML 配置文件的话，MyBatis 会自动查找并加载它（这种情况下， BlogMapper.xml 将会基于类路径和 BlogMapper.class 的类名被加载进来）。具体细节稍后讨论。

从 SqlSessionFactory 中获取 SqlSession

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 = (Blog) session.selectOne("org.mybatis.example.BlogMapper.selec
tBlog", 101);

} finally {

  session.close();

}
```

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直白的方式。使用对于给定语句能够合理描述参数和返回值的接口（比如说 BlogMapper.class），你现在不但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。

例如：

```
SqlSession session = sqlSessionFactory.openSession();

try {

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

  Blog blog = mapper.selectBlog(101);

} finally {
```

```
  session.close();

}
```

现在我们来探究一下这里到底是怎么执行的。

探究已映射的 SQL 语句

现在，或许你很想知道 SqlSession 和 Mapper 到底执行了什么操作，而 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。

在上面提到的两个例子中，一个语句应该是通过 XML 定义，而另外一个则是通过注解定义。先看 XML 定义这个，事实上 MyBatis 提供的全部特性可以利用基于 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，这个概念应该会比较熟悉。不过 XML 映射文件已经有了很多的改进，随着文档的进行会愈发清晰。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>
```

对于这个简单的例子来说似乎有点小题大做了，但实际上它是非常轻量级的。在一个 XML 映射文件中，你想定义多少个映射语句都是可以的，这样下来，XML 头部和文档类型声明占去的部分就显得微不足道了。文件的剩余部分具有很好的自解释性。在命名空间“org.mybatis.example.BlogMapper”中定义了一个名为“selectBlog”的映射语句，这样它就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，就像上面的例子中做的那样：

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

你可能注意到这和使用完全限定名调用 Java 对象的方法是相似的，之所以这样做是有原因的。这个命名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。这样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。不过让我们再看一遍下面的例子：

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

Blog blog = mapper.selectBlog(101);
```

第二种方法有很多优势，首先它不是基于字符串常量的，就会更安全；其次，如果你的 IDE 有代码补全功能，那么你可以在有了已映射 SQL 语句的基础之上利用这个功能。

提示命名空间的一点注释

命名空间（Namespaces）在之前版本的 MyBatis 中是可选的，这样容易引起混淆因此毫无益处。现在命名空间则是必须的，且意于简单地用更长的完全限定名来隔离语句。

命名空间使得你所见到的接口绑定成为可能，尽管你觉得这些东西未必用得上，你还是应该遵循这里的规定以防哪天你改变了主意。出于长远考虑，使用命名空间，并将它置于合适的 Java 包命名空间之下，你将拥有一份更加整洁的代码并提高了 MyBatis 的可用性。

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。

- 完全限定名（比如“com.mypackage.MyMapper.selectAllThings”）将被直接查找并且找到即用。
- 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”），那么使用时就会收到错误报告说短名称是不唯一的，这种情况下就必须使用完全限定名。

对于像 BlogMapper 这样的映射器类（Mapper class）来说，还有另一招来处理。它们的映射的语句可以不需要用 XML 来做，取而代之的是可以使用 Java 注解。比如，上面的 XML 示例可被替换如下：

```
package org.mybatis.example;
```

```java
public interface BlogMapper {

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

  Blog selectBlog(int id);

}
```

对于简单语句来说，注解使代码显得更加简洁，然而 Java 注解对于稍微复杂的语句就会力不从心并且会显得更加混乱。因此，如果你需要做很复杂的事情，那么最好使用 XML 来映射语句。

选择何种方式以及映射语句的定义的一致性对你来说有多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。

作用域（Scope）和生命周期

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。

提示 对象生命周期和依赖注入框架

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（mapper）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

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 解析资源开放给更重要的事情。

SqlSessionFactory

SqlSessionFactory 一旦被创建就应该在应用的运行期间一直存在，没有任何理由对它进行清除或重建。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

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管理作用域中，比如 Servlet 架构中的 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

} finally {

  session.close();

}
```

在你的所有的代码中一致性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。

映射器实例（Mapper Instances）

映射器是一个你创建来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，用过之后即可废弃。并不需要显式地关闭映射器实例，尽管在整个请求作用域（request scope）保持映射器实例也不会有什么问题，但是很快你会发现，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。所以要保持简单，最好把映射器放在方法作用域（method scope）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

} finally {
```

```
  session.close();

}
```

XML 映射配置文件

MyBatis 的配置文件包含了会深深影响 MyBatis 行为的设置（settings）和属性（properties）信息。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
    - 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

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>
```

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>
```

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。

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

```
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, pr
ops);



// ... or ...



SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, en
vironment, props);
```

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：

- 在 properties 元素体内指定的属性首先被读取。
- 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。
- 最后读取作为方法参数传递的属性，并覆盖已读取的同名属性。

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。

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' -->

</dataSource>
```

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 -->

</properties>
```

提示 你可以使用 `":"` 作为属性键(e.g. `db:username`) 或者你也可以在 sql 定义中使用 OGNL 表达式的三元运算符(e.g. `${tableName != null ? tableName : 'global_constants'}`)， 你应该通过增加一个指定的属性来改变分隔键和默认值的字符。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 -->

</properties>

<dataSource type="POOLED">

  <!-- ... -->

  <property name="username" value="${db:username?:ut_user}"/>

</dataSource>
```

settings

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。

设置参数	描述
cacheEnabled	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。

设置参数	描述
lazyLoadingEnabled	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 属性来覆盖该项的开关状态。
aggressiveLazyLoading	当开启时，任何方法的调用都会加载该对象的所有属性。否 `lazyLoadTriggerMethods`).
multipleResultSetsEnabled	是否允许单一语句返回多结果集（需要兼容驱动）。
useColumnLabel	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 种不同的模式来观察所用驱动的结果。
useGeneratedKeys	允许 JDBC 支持自动生成主键，需要驱动兼容。 如果设置为 true 尽管一些驱动不能兼容但仍可正常工作（比如 Derby）。
autoMappingBehavior	指定 MyBatis 应如何自动映射列到字段或属性。 NONE 表示取消自 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
autoMappingUnknownColumnBehavior	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.AutoMappingUnknow` 必须设置为 `WARN`) • `FAILING`: 映射失败 (抛出 `SqlSessionException`)
defaultExecutorType	配置默认的执行器。SIMPLE 就是普通的执行器；REUSE 执行器会重 BATCH 执行器将重用语句并执行批量更新。
defaultStatementTimeout	设置超时时间，它决定驱动等待数据库响应的秒数。
defaultFetchSize	为驱动的结果集获取数量（fetchSize）设置一个提示值。此参数只可
safeRowBoundsEnabled	允许在嵌套语句中使用分页（RowBounds）。如果允许使用则设置为
safeResultHandlerEnabled	允许在嵌套语句中使用分页（ResultHandler）。如果允许使用则设置

设置参数	描述
mapUnderscoreToCamelCase	是否开启自动驼峰命名规则（camel case）映射，即从经典数据库列 aColumn 的类似映射。
localCacheScope	MyBatis 利用本地缓存机制（Local Cache）防止循环引用（circular re 认值为 SESSION，这种情况下会缓存一个会话中执行的所有查询。 仅用在语句执行上，对相同 SqlSession 的不同调用将不会共享数据
jdbcTypeForNull	当没有为参数提供特定的 JDBC 类型时，为空值指定 JDBC 类型。 多数情况直接用一般类型即可，比如 NULL、VARCHAR 或 OTHER。
lazyLoadTriggerMethods	指定哪个对象的方法触发一次延迟加载。
defaultScriptingLanguage	指定动态 SQL 生成的默认语言。
defaultEnumTypeHandler	指定 Enum 使用的默认 `TypeHandler` 。 (从 3.4.5 开始)
callSettersOnNulls	指定当结果集中值为 null 的时候是否调用映射对象的 setter（m Map.keySet() 依赖或 null 值初始化的时候是有用的。注意基本类型 的。
returnInstanceForEmptyRow	当返回行的所有列都是空时，MyBatis 默认返回 `null`。 当开启这个 请注意，它也适用于嵌套的结果集 (i.e. collectioin and association)。
logPrefix	指定 MyBatis 增加到日志名称的前缀。
logImpl	指定 MyBatis 所用日志的具体实现，未指定时将自动查找。
proxyFactory	指定 Mybatis 创建具有延迟加载能力的对象所用到的代理工具。
vfsImpl	指定 VFS 的实现
useActualParamName	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， 加上`-parameters` 选项。(从 3.4.1 开始)
configurationFactory	指定一个提供 `Configuration` 实例的类。 这个被返回的 Configu

设置参数	描述
	的懒加载属性值。这个类必须包含一个签名方法 `static Configu` (从 3.2.3 版本开始)

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,toStr
ing"/>

</settings>
```

typeAliases

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>
```

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

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>
```

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。看下面的例子：

```
@Alias("author")

public class Author {

    ...

}
```

这是一些为常见的 Java 类型内建的相应的类型别名。它们都是大小写不敏感的，需要注意的是由基本类型名称重复导致的特殊处理。

别名
_byte
_long
_short
_int
_integer
_double
_float
_boolean
string
byte
long
short
int
integer
double

别名
float
boolean
date
decimal
bigdecimal
object
map
hashmap
list
arraylist
collection
iterator

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。

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

类型处理器	Java 类型
`BooleanTypeHandler`	`java.lang.Boolean`, `boolean`
`ByteTypeHandler`	`java.lang.Byte`, `byte`
`ShortTypeHandler`	`java.lang.Short`, `short`
`IntegerTypeHandler`	`java.lang.Integer`, `int`
`LongTypeHandler`	`java.lang.Long`, `long`
`FloatTypeHandler`	`java.lang.Float`, `float`
`DoubleTypeHandler`	`java.lang.Double`, `double`
`BigDecimalTypeHandler`	`java.math.BigDecimal`
`StringTypeHandler`	`java.lang.String`
`ClobReaderTypeHandler`	`java.io.Reader`
`ClobTypeHandler`	`java.lang.String`
`NStringTypeHandler`	`java.lang.String`
`NClobTypeHandler`	`java.lang.String`
`BlobInputStreamTypeHandler`	`java.io.InputStream`
`ByteArrayTypeHandler`	`byte[]`

类型处理器	Java 类型
`BlobTypeHandler`	`byte[]`
`DateTypeHandler`	`java.util.Date`
`DateOnlyTypeHandler`	`java.util.Date`
`TimeOnlyTypeHandler`	`java.util.Date`
`SqlTimestampTypeHandler`	`java.sql.Timestamp`
`SqlDateTypeHandler`	`java.sql.Date`
`SqlTimeTypeHandler`	`java.sql.Time`
`ObjectTypeHandler`	Any
`EnumTypeHandler`	Enumeration Type
`EnumOrdinalTypeHandler`	Enumeration Type
`InstantTypeHandler`	`java.time.Instant`
`LocalDateTimeTypeHandler`	`java.time.LocalDateTime`
`LocalDateTypeHandler`	`java.time.LocalDate`
`LocalTimeTypeHandler`	`java.time.LocalTime`
`OffsetDateTimeTypeHandler`	`java.time.OffsetDateTime`

类型处理器	Java 类型
`OffsetTimeTypeHandler`	`java.time.OffsetTime`
`ZonedDateTimeTypeHandler`	`java.time.ZonedDateTime`
`YearTypeHandler`	`java.time.Year`
`MonthTypeHandler`	`java.time.Month`
`YearMonthTypeHandler`	`java.time.YearMonth`
`JapaneseDateTypeHandler`	`java.time.chrono.JapaneseDate`

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 parame
ter, 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>
```

使用这个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 直到语句被执行才清楚数据类型。

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：

- 在类型处理器的配置元素（typeHandler element）上增加一个 `javaType` 属性（比如：`javaType="String"`）；
- 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。

可以通过两种方式来指定被关联的 JDBC 类型：

- 在类型处理器的配置元素上增加一个 `jdbcType` 属性（比如：`jdbcType="VARCHAR"`）；
- 在类型处理器的类上（TypeHandler class）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。

当决定在 `ResultMap` 中使用某一 TypeHandler 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=[TheJavaType], jdbcType=null` 的组合来选择一个 TypeHandler。 这意味着使用`@MappedJdbcTypes` 注解可以*限制*TypeHandler 的范围，同时除非显式的设置，否则 TypeHandler 在 `ResultMap` 中将是无效的。 如果希望在 `ResultMap` 中使用 TypeHandler，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 TypeHandler 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`）。

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>
```

注意在使用自动检索（autodiscovery）功能的时候，只能通过注解方式来指定 JDBC 的类型。

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 cann
ot be null");

    this.type = type;

  }

  ...
```

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

处理枚举类型

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

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。

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。

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。

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

<typeHandlers>

  <typeHandler handler="org.apache.ibatis.type.EnumOrdinalTypeHandler" java
Type="java.math.RoundingMode"/>

</typeHandlers>
```

但是怎样能将同样的 `Enum` 既映射成字符串又映射成整形呢？

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。

（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="use
rmap">

                <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="use
rmap2">

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

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

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

                <result column="roundingMode" property="roundingMode" typeH
andler="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>
```

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

对象工厂（objectFactory）

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<Obj
ect> 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>
```

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。

插件（plugins）

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)

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。

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>
```

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。

提示 **覆盖配置类**

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。

配置环境（environments）

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；或者共享相同 Schema 的多个生产数据库， 想使用相同的 SQL 映射。许多类似的用例。

不过要记住：尽管可以配置多个环境，每个 SqlSessionFactory 实例只能选择其一。

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：

- **每个数据库对应一个 SqlSessionFactory 实例**

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, en
vironment);
```

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

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);
```

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>
```

注意这里的关键点:

- 默认的环境 ID（比如:default="development"）。
- 每个 environment 元素定义的环境 ID（比如:id="development"）。
- 事务管理器的配置（比如:type="JDBC"）。
- 数据源的配置（比如:type="POOLED"）。

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。

事务管理器（transactionManager）

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

- 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 – 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>
```

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。

这两种事务管理器类型都不需要任何属性。它们不过是类型别名，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, TransactionIsolationLev
el level, boolean autoCommit);

}
```

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;

}
```

使用这两个接口，你可以完全自定义 MyBatis 对事务的处理。

数据源（dataSource）

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。

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

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

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

- `driver` – 这是 JDBC 驱动的 Java 类的完全限定名（并不是 JDBC 驱动中可能包含的数据源类）。
- `url` – 这是数据库的 JDBC URL 地址。
- `username` – 登录数据库的用户名。
- `password` – 登录数据库的密码。
- `defaultTransactionIsolationLevel` – 默认的连接事务隔离级别。

作为可选项，你也可以传递属性给数据库驱动。要这样做，属性的前缀为“driver.”，例如：

- `driver.encoding=UTF8`

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

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

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

- `poolMaximumActiveConnections` – 在任意时间可以存在的活动（也就是正在使用）连接数量，默认值：10
- `poolMaximumIdleConnections` – 任意时间可能存在的空闲连接数。

- `poolMaximumCheckoutTime` – 在被强制返回之前，池中连接被检出（checked out）时间，默认值：20000 毫秒（即 20 秒）
- `poolTimeToWait` – 这是一个底层设置，如果获取连接花费了相当长的时间，连接池会打印状态日志并重新尝试获取一个连接（避免在误配置的情况下一直安静的失败），默认值：20000 毫秒（即 20 秒）。
- `poolMaximumLocalBadConnectionTolerance` – 这是一个关于坏连接容忍度的底层设置， 作用于每一个尝试从缓存池获取连接的线程. 如果这个线程获取到的是一个坏的连接，那么这个数据源允许这个线程尝试重新获取一个新的连接，但是这个重新尝试的次数不应该超过 `poolMaximumIdleConnections` 与 `poolMaximumLocalBadConnectionTolerance` 之和。 默认值：3 (新增于 3.4.5)
- `poolPingQuery` – 发送到数据库的侦测查询，用来检验连接是否正常工作并准备接受请求。默认是“NO PING QUERY SET”，这会导致多数数据库驱动失败时带有一个恰当的错误消息。
- `poolPingEnabled` – 是否启用侦测查询。若开启，需要设置 `poolPingQuery` 属性为一个可执行的 SQL 语句（最好是一个速度非常快的 SQL 语句），默认值：false。
- `poolPingConnectionsNotUsedFor` – 配置 poolPingQuery 的频率。可以被设置为和数据库连接超时时间一样，来避免不必要的侦测，默认值：0（即所有连接每一时刻都被侦测 — 当然仅当 poolPingEnabled 为 true 时适用）。

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

- `initial_context` – 这个属性用来在 InitialContext 中寻找上下文（即，initialContext.lookup(initial_context)）。这是个可选属性，如果忽略，那么 data_source 属性将会直接从 InitialContext 中寻找。
- `data_source` – 这是引用数据源实例位置的上下文的路径。提供了 initial_context 配置时会在其返回的上下文中进行查找，没有提供时则直接在 InitialContext 中查找。

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

- `env.encoding=UTF8`

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

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

```
public interface DataSourceFactory {

  void setProperties(Properties props);

  DataSource getDataSource();
```

```
}
```

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

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

import com.mchange.v2.c3p0.ComboPooledDataSource;



public class C3P0DataSourceFactory extends UnpooledDataSourceFactory {



  public C3P0DataSourceFactory() {

    this.dataSource = new ComboPooledDataSource();

  }

}
```

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

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

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

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

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

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

</dataSource>
```

databaseIdProvider

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

```
<databaseIdProvider type="DB_VENDOR" />
```

这里的 DB_VENDOR 会通过 `DatabaseMetaData#getDatabaseProductName()` 返回的字符串进行设置。 由于通常情况下这个字符串都非常长而且相同产品的不同版本会返回不同的值，所以最好通过设置属性别名来使其变短，如下：

```
<databaseIdProvider type="DB_VENDOR">

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

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

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

</databaseIdProvider>
```

在提供了属性别名时，DB_VENDOR databaseIdProvider 将被设置为第一个能匹配数据库产品名称的属性键对应的值，如果没有匹配的属性将会设置为 “null”。 在这个例子中，如果 `getDatabaseProductName()` 返回“Oracle (DataDirect)”，databaseId 将被设置为“oracle”。

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

```
public interface DatabaseIdProvider {

  void setProperties(Properties p);

  String getDatabaseId(DataSource dataSource) throws SQLException;

}
```

映射器（mappers）

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

```
<!-- 使用相对于类路径的资源引用 -->

<mappers>

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

```
  <mapper resource="org/mybatis/builder/BlogMapper.xml"/>
  <mapper resource="org/mybatis/builder/PostMapper.xml"/>
</mappers>
<!-- 使用完全限定资源定位符（URL） -->
<mappers>
  <mapper url="file:///var/mappers/AuthorMapper.xml"/>
  <mapper url="file:///var/mappers/BlogMapper.xml"/>
  <mapper url="file:///var/mappers/PostMapper.xml"/>
</mappers>
<!-- 使用映射器接口实现类的完全限定类名 -->
<mappers>
  <mapper class="org.mybatis.builder.AuthorMapper"/>
  <mapper class="org.mybatis.builder.BlogMapper"/>
  <mapper class="org.mybatis.builder.PostMapper"/>
</mappers>
<!-- 将包内的映射器接口实现全部注册为映射器 -->
<mappers>
  <package name="org.mybatis.builder"/>
</mappers>
```

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

Mapper XML 文件

MyBatis 的真正强大在于它的映射语句，也是它的魔力所在。由于它的异常强大，映射器的 XML 文件就显得相对简单。如果拿它跟具有相同功能的 JDBC 代码进行对比，

你会立即发现省掉了将近 95% 的代码。MyBatis 就是针对 SQL 构建的，并且比普通的方法做的更好。

SQL 映射文件有很少的几个顶级元素（按照它们应该被定义的顺序）：

- `cache` – 给定命名空间的缓存配置。
- `cache-ref` – 其他命名空间缓存配置的引用。
- `resultMap` – 是最复杂也是最强大的元素，用来描述如何从数据库结果集中来加载对象。
- ~~`parameterMap` – 已废弃！老式风格的参数映射。内联参数是首选,这个元素可能在将来被移除，这里不会记录。~~
- `sql` – 可被其他语句引用的可重用语句块。
- `insert` – 映射插入语句
- `update` – 映射更新语句
- `delete` – 映射删除语句
- `select` – 映射查询语句

下一部分将从语句本身开始来描述每个元素的细节。

select

查询语句是 MyBatis 中最常用的元素之一，光能把数据存到数据库中价值并不大，如果还能重新取出来才有用，多数应用也都是查询比修改要频繁。对每个插入、更新或删除操作，通常对应多个查询操作。这是 MyBatis 的基本原则之一，也是将焦点和努力放到查询和结果映射的原因。简单查询的 select 元素是非常简单的。比如：

```
<select id="selectPerson" parameterType="int" resultType="hashmap">

  SELECT * FROM PERSON WHERE ID = #{id}

</select>
```

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

注意参数符号：

```
#{id}
```

这就告诉 MyBatis 创建一个预处理语句参数，通过 JDBC，这样的一个参数在 SQL 中会由一个“?”来标识，并被传递到一个新的预处理语句中，就像这样：

```
// Similar JDBC code, NOT MyBatis…

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

PreparedStatement ps = conn.prepareStatement(selectPerson);
```

```
ps.setInt(1,id);
```

当然，这需要很多单独的 JDBC 的代码来提取结果并将它们映射到对象实例中，这就是 MyBatis 节省你时间的地方。我们需要深入了解参数和结果映射，细节部分我们下面来了解。

select 元素有很多属性允许你配置，来决定每条语句的作用细节。

```
<select

 id="selectPerson"

 parameterType="int"

 parameterMap="deprecated"

 resultType="hashmap"

 resultMap="personResultMap"

 flushCache="false"

 useCache="true"

 timeout="10000"

 fetchSize="256"

 statementType="PREPARED"

 resultSetType="FORWARD_ONLY">
```

属性	描述
id	在命名空间中唯一的标识符，可以被用来引用这条语句。
parameterType	将会传入这条语句的参数类的完全限定名或别名。这个属性是可选的，因为 MyBatis 可

属性	描述
~~parameterMap~~	~~这是引用外部 parameterMap 的已经被废弃的方法。使用内联参数映射和 parameterTy~~
`resultType`	从这条语句中返回的期望类型的类的完全限定名或别名。注意如果是集合情形，那应该
`resultMap`	外部 resultMap 的命名引用。结果集的映射是 MyBatis 最强大的特性，对其有一个很好
`flushCache`	将其设置为 true，任何时候只要语句被调用，都会导致本地缓存和二级缓存都会被清空
`useCache`	将其设置为 true，将会导致本条语句的结果被二级缓存，默认值：对 select 元素为 tru
`timeout`	这个设置是在抛出异常之前，驱动程序等待数据库返回请求结果的秒数。默认值为 unse
`fetchSize`	这是尝试影响驱动程序每次批量返回的结果行数和这个设置值相等。默认值为 unset（
`statementType`	STATEMENT，PREPARED 或 CALLABLE 的一个。这会让 MyBatis 分别使用 Statement,
`resultSetType`	FORWARD_ONLY，SCROLL_SENSITIVE 或 SCROLL_INSENSITIVE 中的一个，默认值为 u
`databaseId`	如果配置了 databaseIdProvider，MyBatis 会加载所有的不带 databaseId 或匹配当前 d
`resultOrdered`	这个设置仅针对嵌套结果 select 语句适用：如果为 true，就是假设包含了嵌套结果集或 的时候不至于导致内存不够用。默认值：`false`。
`resultSets`	这个设置仅对多结果集的情况适用，它将列出语句执行后返回的结果集并每个结果集给

insert, update 和 delete

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

```
<insert

  id="insertAuthor"

  parameterType="domain.blog.Author"

  flushCache="true"

  statementType="PREPARED"

  keyProperty=""

  keyColumn=""

  useGeneratedKeys=""

  timeout="20">



<update

  id="updateAuthor"

  parameterType="domain.blog.Author"

  flushCache="true"

  statementType="PREPARED"

  timeout="20">



<delete

  id="deleteAuthor"

  parameterType="domain.blog.Author"

  flushCache="true"

  statementType="PREPARED"

  timeout="20">
```

属性	描述
id	命名空间中的唯一标识符，可被用来代表这条语句。
parameterType	将要传入语句的参数的完全限定类名或别名。这个属性是可选的，因为 MyBatis 可
~~parameterMap~~	~~这是引用外部 parameterMap 的已经被废弃的方法。使用内联参数映射和 paramet~~
flushCache	将其设置为 true，任何时候只要语句被调用，都会导致本地缓存和二级缓存都会被
timeout	这个设置是在抛出异常之前，驱动程序等待数据库返回请求结果的秒数。默认值为
statementType	STATEMENT，PREPARED 或 CALLABLE 的一个。这会让 MyBatis 分别使用 Statem
useGeneratedKeys	（仅对 insert 和 update 有用）这会令 MyBatis 使用 JDBC 的 getGeneratedKeys 值：false。
keyProperty	（仅对 insert 和 update 有用）唯一标记一个属性，MyBatis 会通过 getGenerated 以是逗号分隔的属性名称列表。
keyColumn	（仅对 insert 和 update 有用）通过生成的键值设置表中的列名，这个设置仅在某 逗号分隔的属性名称列表。
databaseId	如果配置了 databaseIdProvider，MyBatis 会加载所有的不带 databaseId 或匹配当

下面就是 insert，update 和 delete 语句的示例：

```
<insert id="insertAuthor">

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

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

</insert>
```

```
<update id="updateAuthor">
  update Author set
    username = #{username},
    password = #{password},
    email = #{email},
    bio = #{bio}
  where id = #{id}
</update>

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

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

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

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

如果你的数据库还支持多行插入，你也可以传入一个 `Authors` 数组或集合，并返回自动生成的主键。

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

    keyProperty="id">

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

  <foreach item="item" collection="list" separator=",">

    (#{item.username}, #{item.password}, #{item.email}, #{item.bio})

  </foreach>

</insert>
```

对于不支持自动生成类型的数据库或可能不支持自动生成主键的 JDBC 驱动，MyBatis 有另外一种方法来生成主键。

这里有一个简单（甚至很傻）的示例，它可以生成一个随机 ID（你最好不要这么做，但这里展示了 MyBatis 处理问题的灵活性及其所关心的广度）：

```
<insert id="insertAuthor">

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

    select CAST(RANDOM()*1000000 as INTEGER) a from SYSIBM.SYSDUMMY1

  </selectKey>

  insert into Author

    (id, username, password, email,bio, favourite_section)

  values

    (#{id}, #{username}, #{password}, #{email}, #{bio}, #{favouriteSection,j
dbcType=VARCHAR})

</insert>
```

在上面的示例中，selectKey 元素将会首先运行，Author 的 id 会被设置，然后插入语句会被调用。这给你了一个和数据库中来处理自动生成的主键类似的行为，避免了使 Java 代码变得复杂。

selectKey 元素描述如下：

```
<selectKey
```

```
  keyProperty="id"

  resultType="int"

  order="BEFORE"

  statementType="PREPARED">
```

se

属性	描述
keyProperty	selectKey 语句结果应该被设置的目标属性。如果希望得到多个生成的列，也可以是逗号
keyColumn	匹配属性的返回结果集中的列名称。如果希望得到多个生成的列，也可以是逗号分隔的
resultType	结果的类型。MyBatis 通常可以推算出来，但是为了更加确定写上也不会有什么问题。Object 或一个 Map。
order	这可以被设置为 BEFORE 或 AFTER。如果设置为 BEFORE，那么它会首先选择主键，设数据库相似，在插入语句内部可能有嵌入索引调用。
statementType	与前面相同，MyBatis 支持 STATEMENT，PREPARED 和 CALLABLE 语句的映射类型，

sql

这个元素可以被用来定义可重用的 SQL 代码段，可以包含在其他语句中。它可以被静态地(在加载参数) 参数化. 不同的属性值通过包含的实例变化. 比如：

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

这个 SQL 片段可以被包含在其他语句中，例如：

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

  select
```

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

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

  from some_table t1

    cross join some_table t2

</select>
```

属性值也可以被用在 include 元素的 refid 属性里（

```
<include refid="${include_target}"/>
```

）或 include 内部语句中（

```
${prefix}Table
```

），例如：

```
<sql id="sometable">

  ${prefix}Table

</sql>



<sql id="someinclude">

  from

    <include refid="${include_target}"/>

</sql>



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

  select

    field1, field2, field3

  <include refid="someinclude">
```

```
    <property name="prefix" value="Some"/>
    <property name="include_target" value="sometable"/>
  </include>
</select>
```

参数（Parameters）

前面的所有语句中你所见到的都是简单参数的例子，实际上参数是 MyBatis 非常强大的元素，对于简单的做法，大概 90% 的情况参数都很少，比如：

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

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

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

如果 User 类型的参数对象传递到了语句中，id、username 和 password 属性将会被查找，然后将它们的值传入预处理语句的参数中。

这点相对于向语句中传参是比较好的，而且又简单，不过参数映射的功能远不止于此。

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

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

像 MyBatis 的剩余部分一样，javaType 通常可以由参数对象确定，除非该对象是一个 HashMap。这时所使用的 `TypeHandler` 应该明确指明 javaType。

NOTE 如果一个列允许 null 值，并且会传递值 null 的参数，就必须要指定 JDBC Type。阅读 `PreparedStatement.setNull()` 的 JavaDocs 文档来获取更多信息。

为了以后定制类型处理方式，你也可以指定一个特殊的类型处理器类（或别名），比如：

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

尽管看起来配置变得越来越繁琐，但实际上，很少需要去设置它们。

对于数值类型，还有一个小数保留位数的设置，来确定小数点后保留的位数。

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

最后，mode 属性允许你指定 IN，OUT 或 INOUT 参数。如果参数为 OUT 或 INOUT，参数对象属性的真实值将会被改变，就像你在获取输出参数时所期望的那样。如果 mode 为 OUT（或 INOUT），而且 jdbcType 为 CURSOR(也就是 Oracle 的 REFCURSOR)，你必须指定一个 `resultMap` 来映射结果集 `ResultMap` 到参数类型。要注意这里的 `javaType` 属性是可选的，如果留空并且 jdbcType 是 `CURSOR`，它会被自动地被设为 `ResultMap`。

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

MyBatis 也支持很多高级的数据类型，比如结构体，但是当注册 out 参数时你必须告诉它语句类型名称。比如（再次提示，在实际中要像这样不能换行）：

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

尽管所有这些选项很强大，但大多时候你只须简单地指定属性名，其他的事情 MyBatis 会自己去推断，顶多要为可能为空的列指定 `jdbcType`。

```
#{firstName}

#{middleInitial,jdbcType=VARCHAR}

#{lastName}
```

字符串替换

默认情况下,使用 `#{}` 格式的语法会导致 MyBatis 创建 `PreparedStatement` 参数并安全地设置参数（就像使用 ? 一样）。这样做更安全，更迅速，通常也是首选做法，不

过有时你就是想直接在 SQL 语句中插入一个不转义的字符串。比如，像 ORDER BY，你可以这样来使用：

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

这里 MyBatis 不会修改或转义字符串。

NOTE 用这种方式接受用户的输入，并将其用于语句中的参数是不安全的，会导致潜在的 SQL 注入攻击，因此要么不允许用户输入这些字段，要么自行转义并检验。

Result Maps

`resultMap` 元素是 MyBatis 中最重要最强大的元素。它可以让你从 90% 的 JDBC `ResultSets` 数据提取代码中解放出来，并在一些情形下允许你做一些 JDBC 不支持的事情。 实际上，在对复杂语句进行联合映射的时候，它很可能可以代替数千行的同等功能的代码。 ResultMap 的设计思想是，简单的语句不需要明确的结果映射，而复杂一点的语句只需要描述它们的关系就行了。

你已经见过简单映射语句的示例了,但没有明确的 resultMap。比如:

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

  select id, username, hashedPassword

  from some_table

  where id = #{id}

</select>
```

上述语句只是简单地将所有的列映射到 HashMap 的键上，这由 resultType 属性指定。 虽然在大部分情况下都够用，但是 HashMap 不是一个很好的领域模型。 你的程序更可能会使用 JavaBean 或 POJO(Plain Old Java Objects，普通 Java 对象)作为领域模型。 MyBatis 对两者都支持。看看下面这个 JavaBean:

```
package com.someapp.model;

public class User {

  private int id;

  private String username;

  private String hashedPassword;
```

```
  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getUsername() {
    return username;
  }
  public void setUsername(String username) {
    this.username = username;
  }
  public String getHashedPassword() {
    return hashedPassword;
  }
  public void setHashedPassword(String hashedPassword) {
    this.hashedPassword = hashedPassword;
  }
}
```

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

这样的一个 JavaBean 可以被映射到 `ResultSet`，就像映射到 `HashMap` 一样简单。

```
<select id="selectUsers" resultType="com.someapp.model.User">
  select id, username, hashedPassword
```

```
  from some_table
  where id = #{id}
</select>
```

类型别名是你的好帮手。使用它们，你就可以不用输入类的完全限定名称了。比如：

```
<!-- In mybatis-config.xml file -->
<typeAlias type="com.someapp.model.User" alias="User"/>

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

这些情况下，MyBatis 会在幕后自动创建一个 `ResultMap`，再基于属性名来映射列到 JavaBean 的属性上。如果列名和属性名没有精确匹配，可以在 SELECT 语句中对列使用别名（这是一个 基本的 SQL 特性）来匹配标签。比如：

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

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

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

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

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

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

</resultMap>
```

引用它的语句使用 resultMap 属性就行了（注意我们去掉了 resultType 属性）。比如:

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

  select user_id, user_name, hashed_password

  from some_table

  where id = #{id}

</select>
```

如果世界总是这么简单就好了。

高级结果映射

MyBatis 创建的一个想法是：数据库不可能永远是你所想或所需的那个样子。 我们希望每个数据库都具备良好的第三范式或 BCNF 范式，可惜它们不总都是这样。 如果有一个独立且完美的数据库映射模式，所有应用程序都可以使用它，那就太好了，但可惜也没有。 ResultMap 就是 MyBatis 对这个问题的答案。

比如，我们如何映射下面这个语句？

```
<!-- Very Complex Statement -->

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

  select

       B.id as blog_id,
```

```
     B.title as blog_title,

     B.author_id as blog_author_id,

     A.id as author_id,

     A.username as author_username,

     A.password as author_password,

     A.email as author_email,

     A.bio as author_bio,

     A.favourite_section as author_favourite_section,

     P.id as post_id,

     P.blog_id as post_blog_id,

     P.author_id as post_author_id,

     P.created_on as post_created_on,

     P.section as post_section,

     P.subject as post_subject,

     P.draft as draft,

     P.body as post_body,

     C.id as comment_id,

     C.post_id as comment_post_id,

     C.name as comment_name,

     C.comment as comment_text,

     T.id as tag_id,

     T.name as tag_name

from Blog B
```

```
      left outer join Author A on B.author_id = A.id
      left outer join Post P on B.id = P.blog_id
      left outer join Comment C on P.id = C.post_id
      left outer join Post_Tag PT on PT.post_id = P.id
      left outer join Tag T on PT.tag_id = T.id
  where B.id = #{id}
</select>
```

你可能想把它映射到一个智能的对象模型，这个对象表示了一篇博客，它由某位作者所写， 有很多的博文，每篇博文有零或多条的评论和标签。 我们来看看下面这个完整的例子，它是一个非常复杂的 ResultMap （假设作者,博客,博文,评论和标签都是类型的别名）。 不用紧张，我们会一步一步来说明。 虽然它看起来令人望而生畏，但其实非常简单。

```
<!-- 超复杂的 Result Map -->
<resultMap id="detailedBlogResultMap" type="Blog">
  <constructor>
    <idArg column="blog_id" javaType="int"/>
  </constructor>
  <result property="title" column="blog_title"/>
  <association property="author" javaType="Author">
    <id property="id" column="author_id"/>
    <result property="username" column="author_username"/>
    <result property="password" column="author_password"/>
    <result property="email" column="author_email"/>
    <result property="bio" column="author_bio"/>
    <result property="favouriteSection" column="author_favourite_section"/>
```

```
  </association>

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

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

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

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

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

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

    </collection>

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

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

    </collection>

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

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

    </discriminator>

  </collection>

</resultMap>
```

`resultMap` 元素有很多子元素和一个值得讨论的结构。 下面是 `resultMap` 元素的概念视图。

resultMap

- `constructor` - 用于在实例化类时，注入结果到构造方法中
  - `idArg` - ID 参数;标记出作为 ID 的结果可以帮助提高整体性能
  - `arg` - 将被注入到构造方法的一个普通结果
- `id` – 一个 ID 结果;标记出作为 ID 的结果可以帮助提高整体性能
- `result` – 注入到字段或 JavaBean 属性的普通结果
- `association` – 一个复杂类型的关联;许多结果将包装成这种类型
  - 嵌套结果映射 – 关联可以指定为一个 `resultMap` 元素，或者引用一个
- `collection` – 一个复杂类型的集合

    - 嵌套结果映射 – 集合可以指定为一个 `resultMap` 元素，或者引用一个
- `discriminator` – 使用结果值来决定使用哪个 `resultMap`
    - `case` – 基于某些值的结果映射
        - 嵌套结果映射 – 一个 `case` 也是一个映射它本身的结果,因此可以包含很多相同的元素，或者它可以参照一个外部的 `resultMap`。

ResultMap 的&

属性	描述
`id`	当前命名空间中的一个唯一标识，用于标识一个 result map.
`type`	类的完全限定名，或者一个类型别名 (内置的别名可以参考上面的表格).
`autoMapping`	如果设置这个属性，MyBatis 将会为这个 ResultMap 开启或者关闭自动映射。这个属

最佳实践 最好一步步地建立结果映射。单元测试可以在这个过程中起到很大帮助。如果你尝试一次创建一个像上面示例那样的巨大的结果映射， 那么很可能会出现错误而且很难去使用它来完成工作。 从最简单的形态开始，逐步进化。而且别忘了单元测试！使用框架的缺点是有时候它们看上去像黑盒子(无论源代码是否可见)。 为了确保你实现的行为和想要的一致，最好的选择是编写单元测试。提交 bug 的时候它也能起到很大的作用。

下一部分将详细说明每个元素。

id & result

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

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

这些是结果映射最基本的内容。id 和 result 都将一个列的值映射到一个简单数据类型(字符串,整型,双精度浮点数,日期等)的属性或字段。

这两者之间的唯一不同是， id 表示的结果将是对象的标识属性，这会在比较对象实例时用到。 这样可以提高整体的性能，尤其是缓存和嵌套结果映射(也就是联合映射)的时候。

两个元素都有一些属性:

属性	描述
property	映射到列结果的字段或属性。如果用来匹配的 JavaBeans 存在给定名字的属性，那么它将会 复杂属性导航。比如,你可以这样映射一些简单的东西: “username” ,或者映射到一些复杂的东
column	数据库中的列名,或者是列的别名。一般情况下，这和 传递给 resultSet.getString(
javaType	一个 Java 类的完全限定名,或一个类型别名(参考上面内建类型别名 的列表) 。如果你映射 期望的行为。
jdbcType	JDBC 类型，所支持的 JDBC 类型参见这个表格之后的“支持的 JDBC 类型”。 只需要在可能 编程,你需要对可能为 null 的值指定这个类型。
typeHandler	我们在前面讨论过的默认类型处理器。使用这个属性,你可以覆盖默 认的类型处理器。这个

支持的 JDBC 类型

为了未来的参考,MyBatis 通过包含的 jdbcType 枚举型,支持下面的 JDBC 类型。

BIT	FLOAT	CHAR
TINYINT	REAL	VARCHAR
SMALLINT	DOUBLE	LONGVARCHAR
INTEGER	NUMERIC	DATE
BIGINT	DECIMAL	TIME

构造方法

通过修改对象属性的方式，可以满足大多数的数据传输对象(Data Transfer Object,DTO)以及绝大部分领域模型的要求。 但有些情况下你想使用不可变类。 通常来说，很少或基本不变的、包含引用或查询数 据的表，很适合使用不可变类。 构造方法注入允许你在初始化时 为类设置属性的值，而不用暴露出公有方法。MyBatis 也支持私有属性和私有 JavaBeans 属 性来达到这个目的，但有一些人更青睐于构造方法注入。 constructor 元素就是为此而生的。

看看下面这个构造方法:

```
public class User {

   //...

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

     //...

  }

//...

}
```

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

```
<constructor>

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

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

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

</constructor>
```

当你在处理一个带有多个形参的构造方法时，很容易在保证 arg 元素的正确顺序上出错。 从版本 3.4.3 开始，可以在指定参数名称的前提下，以任意顺序编写 arg 元素。为了通过名称来引用构造方法参数，你可以添加 `@Param` 注解，或者使用 '-parameters' 编译选项并启用 `useActualParamName` 选项（默认开启）来编译项目。 下面的例子对于同一个构造方法依然是有效的，尽管第二和第三个形参顺序与构造方法中声明的顺序不匹配。

```
<constructor>
   <idArg column="id" javaType="int" name="id" />
   <arg column="age" javaType="_int" name="age" />
   <arg column="username" javaType="String" name="username" />
</constructor>
```

如果类中存在名称和类型相同的属性，那么可以省略 `javaType` 。

剩余的属性和规则和普通的 id 和 result 元素是一样的。

属性	描述
`column`	数据库中的列名,或者是列的别名。一般情况下，这和 传递给 `resultSet.getString(`
`javaType`	一个 Java 类的完全限定名,或一个类型别名(参考上面内建类型别名的列表)。 如果你映射到 望的 行为。
`jdbcType`	JDBC 类型，所支持的 JDBC 类型参见这个表格之前的“支持的 JDBC 类型”。 只需要在可能 编程,你需要对可能为 null 的值指定这个类型。
`typeHandler`	我们在前面讨论过的默认类型处理器。使用这个属性,你可以覆盖默 认的类型处理器。这个
`select`	用于加载复杂类型属性的映射语句的 ID,它会从 column 属性中指定的列检索数据，作为参
`resultMap`	ResultMap 的 ID，可以将嵌套的结果集映射到一个合适的对象树中，功能和 select 属性相 复的结果集正确的映射到嵌套的对象树中。为了实现它, MyBatis 允许你 “串联” ResultMap,
`name`	构造方法形参的名字。从 3.4.3 版本开始，通过指定具体的名字，你可以以任意顺序写入 ar

关联

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

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

</association>
```

关联元素处理“有一个”类型的关系。比如,在我们的示例中,一个博客有一个用户。 关联映射就工作于这种结果之上。你指定了目标属性,来获取值的列,属性的 java 类型(很 多情况下 MyBatis 可以自己算出来) ,如果需要的话还有 jdbc 类型,如果你想覆盖或获取的 结果值还需要类型控制器。

关联中不同的是你需要告诉 MyBatis 如何加载关联。MyBatis 在这方面会有两种不同的 方式:

- 嵌套查询:通过执行另外一个 SQL 映射语句来返回预期的复杂类型。
- 嵌套结果:使用嵌套结果映射来处理重复的联合结果的子集。首先,然让我们来查看这个元素的属性。所有的你都会看到,它和普通的只由 select 和

resultMap 属性的结果映射不同。

属性	描述
property	映射到列结果的字段或属性。如果用来匹配的 JavaBeans 存在给定名字的属性，那么它将会 这样映射 一 些 东 西 :“ username ”, 或 者 映 射 到 一 些 复 杂 的 东 西 : “address.s
javaType	一个 Java 类的完全限定名,或一个类型别名(参考上面内建类型别名的列 表) 。如果你映射到 来保证所需的 行为。
jdbcType	在这个表格之前的所支持的 JDBC 类型列表中的类型。JDBC 类型是仅仅 需要对插入，更新 指定这个类型-但 仅仅对可能为空的值。
typeHandler	我们在前面讨论过默认的类型处理器。使用这个属性,你可以覆盖默认的 typeHandler 类型

关联的嵌套查询

属性	描述
column	来自数据库的列名,或重命名的列标签。这和通常传递给 resultSet.getString(columnName)方 {prop1=col1,prop2=col2} ” 这种语法来传递给嵌套查询语 句。这会引起 prop1 和 prop2 以

属性	描述
`select`	另外一个映射语句的 ID,可以加载这个属性映射需要的复杂类型。获取的 在列属性中指定的列 以 指 定 多 个 列 名 通 过 column= ”{prop1=col1,prop2=col2}” 这种语法来传递给嵌套查
`fetchType`	可选的。有效值为 `lazy` 和 `eager`。 如果使用了， 它将取代全局配置参数 `lazyLoadingE`

示例:

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

  <association property="author" column="author_id" javaType="Author" select
="selectAuthor"/>

</resultMap>



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

  SELECT * FROM BLOG WHERE ID = #{id}

</select>



<select id="selectAuthor" resultType="Author">

  SELECT * FROM AUTHOR WHERE ID = #{id}

</select>
```

我们有两个查询语句:一个来加载博客,另外一个来加载作者,而且博客的结果映射描 述了“selectAuthor”语句应该被用来加载它的 author 属性。

其他所有的属性将会被自动加载,假设它们的列和属性名相匹配。

这种方式很简单， 但是对于大型数据集合和列表将不会表现很好。 问题就是我们熟知的“N+1 查询问题”。概括地讲,N+1 查询问题可以是这样引起的:

- 你执行了一个单独的 SQL 语句来获取结果列表(就是“+1”)。
- 对返回的每条记录,你执行了一个查询语句来为每个加载细节(就是“N”)。

这个问题会导致成百上千的 SQL 语句被执行。这通常不是期望的。

MyBatis 能延迟加载这样的查询就是一个好处,因此你可以分散这些语句同时运行的消耗。然而,如果你加载一个列表,之后迅速迭代来访问嵌套的数据,你会调用所有的延迟加载,这样的行为可能是很糟糕的。

所以还有另外一种方法。

关联的嵌套结果

属性	描述
`resultMap`	这是结果映射的 ID,可以映射关联的嵌套结果到一个合适的对象图中。这 是一种替代方 的重复组需要被分 解,合理映射到一个嵌套的对象图。为了使它变得容易,MyBatis 让你"
`columnPrefix`	当连接多表时，你将不得不使用列别名来避免 ResultSet 中的重复列名。指定 columnPre
`notNullColumn`	默认情况下，子对象仅在至少一个列映射到其属性非空时才创建。 通过对这个属性指定 隔。默认值：未设置(unset)。
`autoMapping`	如果使用了，当映射结果到当前属性时，Mybatis 将启用或者禁用自动映射。 该属性覆盖 未设置(unset)。

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

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

 select

   B.id            as blog_id,

   B.title         as blog_title,

   B.author_id     as blog_author_id,

   A.id            as author_id,

   A.username      as author_username,

   A.password      as author_password,

   A.email         as author_email,
```

```
    A.bio          as author_bio

  from Blog B left outer join Author A on B.author_id = A.id

  where B.id = #{id}

</select>
```

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

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

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

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

  <association property="author" column="blog_author_id" javaType="Author" r
esultMap="authorResult"/>

</resultMap>



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

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

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

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

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

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

</resultMap>
```

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

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

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

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

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

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

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

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

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

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

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

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

  </association>

</resultMap>
```

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

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

  select

    B.id            as blog_id,

    B.title         as blog_title,

    A.id            as author_id,

    A.username      as author_username,

    A.password      as author_password,

    A.email         as author_email,

    A.bio           as author_bio,
```

```
    CA.id           as co_author_id,
    CA.username     as co_author_username,
    CA.password     as co_author_password,
    CA.email        as co_author_email,
    CA.bio          as co_author_bio
  from Blog B
  left outer join Author A on B.author_id = A.id
  left outer join Author CA on B.co_author_id = CA.id
  where B.id = #{id}
</select>
```

再次调用 Author 的 resultMap 将定义如下：

```
<resultMap id="authorResult" type="Author">
  <id property="id" column="author_id"/>
  <result property="username" column="author_username"/>
  <result property="password" column="author_password"/>
  <result property="email" column="author_email"/>
  <result property="bio" column="author_bio"/>
</resultMap>
```

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

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

```
  <association property="author"
    resultMap="authorResult" />
  <association property="coAuthor"
    resultMap="authorResult"
    columnPrefix="co_" />
</resultMap>
```

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

集合

```
<collection property="posts" ofType="domain.blog.Post">
  <id property="id" column="post_id"/>
  <result property="subject" column="post_subject"/>
  <result property="body" column="post_body"/>
</collection>
```

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

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

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

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

集合的嵌套查询

首先,让我们看看使用嵌套查询来为博客加载文章。

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

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

</resultMap>



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

  SELECT * FROM BLOG WHERE ID = #{id}

</select>



<select id="selectPostsForBlog" resultType="Post">

  SELECT * FROM POST WHERE BLOG_ID = #{id}

</select>
```

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

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

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

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

```
<collection property="posts" column="id" ofType="Post" select="selectPostsF
orBlog"/>
```

集合的嵌套结果

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

首先, 让我们看看 SQL:

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

```
  select

  B.id as blog_id,

  B.title as blog_title,

  B.author_id as blog_author_id,

  P.id as post_id,

  P.subject as post_subject,

  P.body as post_body,

  from Blog B

  left outer join Post P on B.id = P.blog_id

  where B.id = #{id}

</select>
```

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

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

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

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

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

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

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

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

  </collection>

</resultMap>
```

同样,要记得 id 元素的重要性,如果你不记得了,请阅读上面的关联部分。

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

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

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

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

  <collection property="posts" ofType="Post" resultMap="blogPostResult" colu
mnPrefix="post_"/>

</resultMap>



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

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

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

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

</resultMap>
```

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

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

鉴别器

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

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

</discriminator>
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  </discriminator>

</resultMap>
```

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

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

  <result property="doorCount" column="door_count" />

</resultMap>
```

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

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

  <result property="doorCount" column="door_count" />

</resultMap>
```

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

尽管曾经有些人会发现这个外部映射定义会多少有一些令人厌烦之处。 因此还有另外一种语法来做简洁的映射风格。比如:

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

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

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

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

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

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

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

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

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

      <result property="doorCount" column="door_count" />

    </case>

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

      <result property="boxSize" column="box_size" />

      <result property="extendedCab" column="extended_cab" />

    </case>
```

```
    <case value="3" resultType="vanResult">
      <result property="powerSlidingDoor" column="power_sliding_door" />
    </case>
    <case value="4" resultType="suvResult">
      <result property="allWheelDrive" column="all_wheel_drive" />
    </case>
  </discriminator>
</resultMap>
```

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

自动映射

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

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

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

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

```
<select id="selectUsers" resultMap="userResultMap">
  select
    user_id             as "id",
    user_name           as "userName",
```

```
    hashed_password
  from some_table
  where id = #{id}
</select>
<resultMap id="userResultMap" type="User">
  <result property="password" column="hashed_password"/>
</resultMap>
```

有三种自动映射等级：

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

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

```
<select id="selectBlog" resultMap="blogResult">
  select
    B.id,
    B.title,
    A.username,
  from Blog B left outer join Author A on B.author_id = A.id
  where B.id = #{id}
</select>
<resultMap id="blogResult" type="Blog">
  <association property="author" resultMap="authorResult"/>
</resultMap>
```

```
<resultMap id="authorResult" type="Author">
  <result property="username" column="author_username"/>
</resultMap>
```

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

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

```
<resultMap id="userResultMap" type="User" autoMapping="false">
  <result property="password" column="hashed_password"/>
</resultMap>
```

缓存

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

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

```
<cache/>
```

字面上看就是这样。这个简单语句的效果如下:

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

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.

所有的这些属性都可以通过缓存元素的属性来修改。比如:

```
<cache

 eviction="FIFO"

 flushInterval="60000"

 size="512"

 readOnly="true"/>
```

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

可用的收回策略有:

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

默认的是 LRU。

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

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

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

使用自定义缓存

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

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

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

```
public interface Cache {

  String getId();

  int getSize();

  void putObject(Object key, Object value);

  Object getObject(Object key);

  boolean hasKey(Object key);

  Object removeObject(Object key);

  void clear();

}
```

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

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

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

</cache>
```

你可以使用所有简单类型作为 JavaBeans 的属性,MyBatis 会进行转换。 And you can specify a placeholder(e.g. `${cache.file}`) to replace value defined at configuration properties.

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

```
public interface InitializingObject {

  void initialize() throws Exception;

}
```

记得缓存配置和缓存实例是绑定在 SQL 映射文件的命名空间是很重要的。因此,所有在相同命名空间的语句正如绑定的缓存一样。 语句可以修改和缓存交互的方式，或在语

句的 语句的基础上使用两种简单的属性来完全排除它们。默认情况下,语句可以这样来配置:

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

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

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

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

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

参照缓存

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

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

动态 SQL

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

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

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

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

if

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

```
<select id="findActiveBlogWithTitleLike"

    resultType="Blog">

  SELECT * FROM BLOG

  WHERE state = ‘ACTIVE’

  <if test="title != null">

    AND title like #{title}

  </if>

</select>
```

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

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

```
<select id="findActiveBlogLike"

    resultType="Blog">

  SELECT * FROM BLOG WHERE state = ‘ACTIVE’

  <if test="title != null">

    AND title like #{title}

  </if>

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

```
    AND author_name like #{author.name}
  </if>
</select>
```

choose, when, otherwise

有时我们不想应用到所有的条件语句，而只想从中择其一项。针对这种情况，MyBatis 提供了 choose 元素，它有点像 Java 中的 switch 语句。

还是上面的例子，但是这次变为提供了“title”就按“title”查找，提供了“author”就按“author”查找的情形，若两者都没有提供，就返回所有符合条件的 BLOG（实际情况可能是由管理员按一定策略选出 BLOG 列表，而不是返回大量无意义的随机结果）。

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG WHERE state = ‘ACTIVE’
  <choose>
    <when test="title != null">
      AND title like #{title}
    </when>
    <when test="author != null and author.name != null">
      AND author_name like #{author.name}
    </when>
    <otherwise>
      AND featured = 1
    </otherwise>
  </choose>
```

```
</select>
```

trim, where, set

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

```
<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>
```

如果这些条件没有一个能匹配上会发生什么？最终这条 SQL 会变成这样：

```
SELECT * FROM BLOG

WHERE
```

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

```
SELECT * FROM BLOG
```

```
WHERE

AND title like ‘someTitle’
```

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

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

```
<select id="findActiveBlogLike"

    resultType="Blog">

  SELECT * FROM BLOG

  <where>

    <if test="state != null">

        state = #{state}

    </if>

    <if test="title != null">

       AND title like #{title}

    </if>

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

       AND author_name like #{author.name}

    </if>

  </where>

</select>
```

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

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

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

  ...

</trim>
```

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

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

```
<update id="updateAuthorIfNecessary">

  update Author

    <set>

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

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

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

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

    </set>

  where id=#{id}

</update>
```

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

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

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

  ...

</trim>
```

注意这里我们删去的是后缀值，同时添加了前缀值。

foreach

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

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

  SELECT *

  FROM POST P

  WHERE ID in

  <foreach item="item" index="index" collection="list"

      open="(" separator="," close=")">

        #{item}

  </foreach>

</select>
```

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

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

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

bind

`bind` 元素可以从 OGNL 表达式中创建一个变量并将其绑定到上下文。比如：

```
<select id="selectBlogsLike" resultType="Blog">
  <bind name="pattern" value="'%' + _parameter.getTitle() + '%'" />
  SELECT * FROM BLOG
  WHERE title LIKE #{pattern}
</select>
```

多数据库支持

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

```
<insert id="insert">
  <selectKey keyProperty="id" resultType="int" order="BEFORE">
    <if test="_databaseId == 'oracle'">
      select seq_users.nextval from dual
    </if>
    <if test="_databaseId == 'db2'">
      select nextval for seq_users from sysibm.sysdummy1"
    </if>
  </selectKey>
  insert into users values (#{id}, #{name})
```

```
</insert>
```

动态 SQL 中的可插拔脚本语言

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

可以通过实现以下接口来插入一种语言：

```
public interface LanguageDriver {

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

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

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

}
```

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

```
<typeAliases>

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

</typeAliases>

<settings>

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

</settings>
```

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

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

```
  SELECT * FROM BLOG

</select>
```

或者，如果你使用的是映射器接口类，在抽象方法上加上 `@Lang` 注解即可：

```
public interface Mapper {

  @Lang(MyLanguageDriver.class)

  @Select("SELECT * FROM BLOG")

  List<Blog> selectBlog();

}
```

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

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

Java API

既然你已经知道如何配置 MyBatis 和创建映射文件，你就已经准备好来提升技能了。MyBatis 的 Java API 就是你收获你所做的努力的地方。正如你即将看到的，和 JDBC 相比，MyBatis 很大程度简化了你的代码并保持代码简洁，容易理解并维护。MyBatis 3 已经引入了很多重要的改进来使得 SQL 映射更加优秀。

应用目录结构

在我们深入 Java API 之前，理解关于目录结构的最佳实践是很重要的。MyBatis 非常灵活，你可以用你自己的文件来做几乎所有的事情。但是对于任一框架，都有一些最佳的方式。

让我们看一下典型的应用目录结构：

```
/my_application

  /bin

  /devlib

  /lib                <-- MyBatis *.jar 文件在这里。
```

```
  /src
    /org/myapp/
      /action
      /data           <-- MyBatis 配置文件在这里，包括映射器类，XML 配置，XML 映射文件。
        /mybatis-config.xml
        /BlogMapper.java
        /BlogMapper.xml
      /model
      /service
      /view
    /properties       <-- 在你 XML 中配置的属性文件在这里。
  /test
    /org/myapp/
      /action
      /data
      /model
      /service
      /view
    /properties
  /web
    /WEB-INF
      /web.xml
```

当然这是推荐的目录结构，并非强制要求，但是使用一个通用的目录结构将更利于大家沟通。

这部分内容剩余的示例将假设你使用了这种目录结构。

SqlSessions

使用 MyBatis 的主要 Java 接口就是 SqlSession。你可以通过这个接口来执行命令，获取映射器和管理事务。我们会概括讨论一下 SqlSession 本身，但是首先我们还是要了解如何获取一个 SqlSession 实例。SqlSessions 是由 SqlSessionFactory 实例创建

的。SqlSessionFactory 对象包含创建 SqlSession 实例的所有方法。而 SqlSessionFactory 本身是由 SqlSessionFactoryBuilder 创建的，它可以从 XML、注解或手动配置 Java 代码来创建 SqlSessionFactory。

注意 当 Mybatis 与一些依赖注入框架（如 Spring 或者 Guice）同时使用时，SqlSessions 将被依赖注入框架所创建，所以你不需要使用 SqlSessionFactoryBuilder 或者 SqlSessionFactory，可以直接看 SqlSession 这一节。请参考 Mybatis-Spring 或者 Mybatis-Guice 手册了解更多信息。

SqlSessionFactoryBuilder

SqlSessionFactoryBuilder 有五个 build() 方法，每一种都允许你从不同的资源中创建一个 SqlSession 实例。

```
SqlSessionFactory build(InputStream inputStream)

SqlSessionFactory build(InputStream inputStream, String environment)

SqlSessionFactory build(InputStream inputStream, Properties properties)

SqlSessionFactory build(InputStream inputStream, String env, Properties prop
s)

SqlSessionFactory build(Configuration config)
```

第一种方法是最常用的，它使用了一个参照了 XML 文档或上面讨论过的更特定的 mybatis-config.xml 文件的 Reader 实例。可选的参数是 environment 和 properties。environment 决定加载哪种环境，包括数据源和事务管理器。比如：

```
<environments default="development">

  <environment id="development">

    <transactionManager type="JDBC">

        ...

    <dataSource type="POOLED">

        ...

  </environment>

  <environment id="production">
```

```
    <transactionManager type="MANAGED">
        ...
    <dataSource type="JNDI">
        ...
  </environment>
</environments>
```

如果你调用了参数有 environment 的 build 方法，那么 MyBatis 将会使用 configuration 对象来配置这个 environment。当然，如果你指定了一个不合法的 environment，你就会得到错误提示。如果你调用了不带 environment 参数的 build 方法，那么就使用默认的 environment（在上面的示例中指定为 default="development" 的代码）。

如果你调用了参数有 properties 实例的方法，那么 MyBatis 就会加载那些 properties（属性配置文件），并在配置中可用。那些属性可以用${propName} 语法形式多次用在配置文件中。

回想一下，属性可以从 mybatis-config.xml 中被引用，或者直接指定它。因此理解优先级是很重要的。我们在文档前面已经提及它了，但是这里要再次重申：

如果一个属性存在于这些位置，那么 MyBatis 将会按照下面的顺序来加载它们：

- 首先读取在 properties 元素体中指定的属性；
- 其次，读取从 properties 元素的类路径 resource 或 url 指定的属性，且会覆盖已经指定了的重复属性；
- 最后，读取作为方法参数传递的属性，且会覆盖已经从 properties 元素体和 resource 或 url 属性中加载了的重复属性。

因此，通过方法参数传递的属性的优先级最高，resource 或 url 指定的属性优先级中等，在 properties 元素体中指定的属性优先级最低。

总结一下，前四个方法很大程度上是相同的，但是由于覆盖机制，便允许你可选地指定 environment 和/或 properties。以下给出一个从 mybatis-config.xml 文件创建 SqlSessionFactory 的示例：

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

InputStream inputStream = Resources.getResourceAsStream(resource);

SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();

SqlSessionFactory factory = builder.build(inputStream);
```

注意到这里我们使用了 Resources 工具类，这个类在 org.apache.ibatis.io 包中。Resources 类正如其名，会帮助你从类路径下、文件系统或一个 web URL 中加载资源文件。看一下这个类的源代码或者通过你的 IDE 来查看，就会看到一整套相当实用的方法。这里给出一个简表：

```
URL getResourceURL(String resource)

URL getResourceURL(ClassLoader loader, String resource)

InputStream getResourceAsStream(String resource)

InputStream getResourceAsStream(ClassLoader loader, String resource)

Properties getResourceAsProperties(String resource)

Properties getResourceAsProperties(ClassLoader loader, String resource)

Reader getResourceAsReader(String resource)

Reader getResourceAsReader(ClassLoader loader, String resource)

File getResourceAsFile(String resource)
```

```
File getResourceAsFile(ClassLoader loader, String resource)

InputStream getUrlAsStream(String urlString)

Reader getUrlAsReader(String urlString)

Properties getUrlAsProperties(String urlString)

Class classForName(String className)
```

最后一个 build 方法的参数为 Configuration 实例。configuration 类包含你可能需要了解 SqlSessionFactory 实例的所有内容。Configuration 类对于配置的自查很有用，它包含查找和操作 SQL 映射（当应用接收请求时便不推荐使用）。作为一个 Java API 的 configuration 类具有所有配置的开关，这些你已经了解了。这里有一个简单的示例，教你如何手动配置 configuration 实例，然后将它传递给 build() 方法来创建 SqlSessionFactory。

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

TransactionFactory transactionFactory = new JdbcTransactionFactory();



Environment environment = new Environment("development", transactionFactor
y, dataSource);



Configuration configuration = new Configuration(environment);

configuration.setLazyLoadingEnabled(true);

configuration.setEnhancementEnabled(true);

configuration.getTypeAliasRegistry().registerAlias(Blog.class);

configuration.getTypeAliasRegistry().registerAlias(Post.class);

configuration.getTypeAliasRegistry().registerAlias(Author.class);

configuration.addMapper(BoundBlogMapper.class);

configuration.addMapper(BoundAuthorMapper.class);
```

```
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();

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

现在你就获得一个可以用来创建 SqlSession 实例的 SqlSessionFactory 了！

SqlSessionFactory

SqlSessionFactory 有六个方法创建 SqlSession 实例。通常来说，当你选择这些方法时你需要考虑以下几点：

- **事务处理**：我需要在 session 使用事务或者使用自动提交功能（auto-commit）吗？（通常意味着很多数据库和/或 JDBC 驱动没有事务）
- **连接**：我需要依赖 MyBatis 获得来自数据源的配置吗？还是使用自己提供的配置？
- **执行语句**：我需要 MyBatis 复用预处理语句和/或批量更新语句（包括插入和删除）吗？

基于以上需求，有下列已重载的多个 openSession() 方法供使用。

```
SqlSession openSession()

SqlSession openSession(boolean autoCommit)

SqlSession openSession(Connection connection)

SqlSession openSession(TransactionIsolationLevel level)

SqlSession openSession(ExecutorType execType,TransactionIsolationLevel leve
l)

SqlSession openSession(ExecutorType execType)

SqlSession openSession(ExecutorType execType, boolean autoCommit)
```

```
SqlSession openSession(ExecutorType execType, Connection connection)

Configuration getConfiguration();
```

默认的 openSession()方法没有参数，它会创建有如下特性的 SqlSession：

- 会开启一个事务（也就是不自动提交）。

- 将从由当前环境配置的 DataSource 实例中获取 Connection 对象。
- 事务隔离级别将会使用驱动或数据源的默认设置。

- 预处理语句不会被复用，也不会批量处理更新。

这些方法大都是可读性强的。向 `autoCommit` 可选参数传递 `true` 值即可开启自动提交功能。若要使用自己的 `Connection` 实例，传递一个 `Connection` 实例给 `connection` 参数即可。注意并未覆写同时设置 `Connection` 和 `autoCommit` 两者的方法，因为 MyBatis 会使用正在使用中的、设置了 Connection 的环境。MyBatis 为事务隔离级别调用使用了一个 Java 枚举包装器，称为 `TransactionIsolationLevel`，若不使用它，将使用 JDBC 所支持五个隔离级（`NONE`、`READ_UNCOMMITTED`、`READ_COMMITTED`、`REPEATABLE_READ` 和 `SERIALIZABLE`），并按它们预期的方式来工作。

还有一个可能对你来说是新见到的参数，就是 `ExecutorType`。这个枚举类型定义了三个值:

- `ExecutorType.SIMPLE`：这个执行器类型不做特殊的事情。它为每个语句的执行创建一个新的预处理语句。
- `ExecutorType.REUSE`：这个执行器类型会复用预处理语句。

- `ExecutorType.BATCH`：这个执行器会批量执行所有更新语句，如果 SELECT 在它们中间执行，必要时请把它们区分开来以保证行为的易读性。

注意 在 SqlSessionFactory 中还有一个方法我们没有提及，就是 getConfiguration()。这 个方法会返回一个 Configuration 实例，在运行时你可以使用它来自检 MyBatis 的配置。

注意 如果你使用的是 MyBatis 之前的版本，你要重新调用 openSession，因为旧版本的 session、事务和批量操作是分离开来的。如果使用的是新版本，那么就不必这么做了，因为它们现在都包含在 session 的作用域内了。你不必再单独处理事务或批量操作就能得到想要的全部效果。

SqlSession

正如上面所提到的，SqlSession 实例在 MyBatis 中是非常强大的一个类。在这里你会看到所有执行语句、提交或回滚事务和获取映射器实例的方法。

在 SqlSession 类中有超过 20 个方法，所以将它们组合成易于理解的分组。

执行语句方法

这些方法被用来执行定义在 SQL 映射的 XML 文件中的 SELECT、INSERT、UPDATE 和 DELETE 语句。它们都会自行解释，每一句都使用语句的 ID 属性和参数对象，参数可以是原生类型（自动装箱或包装类）、JavaBean、POJO 或 Map。

```
<T> T selectOne(String statement, Object parameter)

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

<K,V> Map<K,V> selectMap(String statement, Object parameter, String mapKey)

int insert(String statement, Object parameter)

int update(String statement, Object parameter)

int delete(String statement, Object parameter)
```

selectOne 和 selectList 的不同仅仅是 selectOne 必须返回一个对象或 null 值。如果返回值多于一个，那么就会抛出异常。如果你不知道返回对象的数量，请使用 selectList。如果需要查看返回对象是否存在，可行的方案是返回一个值即可（0 或 1）。selectMap 稍微特殊一点，因为它会将返回的对象的其中一个属性作为 key 值，将对象作为 value 值，从而将多结果集转为 Map 类型值。因为并不是所有语句都需要参数，所以这些方法都重载成不需要参数的形式。

```
<T> T selectOne(String statement)
```

```
<E> List<E> selectList(String statement)

<K,V> Map<K,V> selectMap(String statement, String mapKey)

int insert(String statement)

int update(String statement)

int delete(String statement)
```

最后，还有 select 方法的三个高级版本，它们允许你限制返回行数的范围，或者提供自定义结果控制逻辑，这通常在数据集合庞大的情形下使用。

```
<E> List<E> selectList (String statement, Object parameter, RowBounds rowBou
nds)

<K,V> Map<K,V> selectMap(String statement, Object parameter, String mapKey,
RowBounds rowbounds)

void select (String statement, Object parameter, ResultHandler<T> handler)

void select (String statement, Object parameter, RowBounds rowBounds, Result
Handler<T> handler)
```

RowBounds 参数会告诉 MyBatis 略过指定数量的记录，还有限制返回结果的数量。RowBounds 类有一个构造方法来接收 offset 和 limit，另外，它们是不可二次赋值的。

```
int offset = 100;

int limit = 25;

RowBounds rowBounds = new RowBounds(offset, limit);
```

所以在这方面，不同的驱动能够取得不同级别的高效率。为了取得最佳的表现，请使用结果集的 SCROLL_SENSITIVE 或 SCROLL_INSENSITIVE 的类型(换句话说：不用 FORWARD_ONLY)。

ResultHandler 参数允许你按你喜欢的方式处理每一行。你可以将它添加到 List 中、创建 Map 和 Set，或者丢弃每个返回值都可以，它取代了仅保留执行语句过后的总结果

列表的死板结果。你可以使用 ResultHandler 做很多事，并且这是 MyBatis 自身内部会使用的方法，以创建结果集列表。

Since 3.4.6, ResultHandler passed to a CALLABLE statement is used on every REFCURSOR output parameter of the stored procedure if there is any.

它的接口很简单。

```
package org.apache.ibatis.session;

public interface ResultHandler<T> {

  void handleResult(ResultContext<? extends T> context);

}
```

ResultContext 参数允许你访问结果对象本身、被创建的对象数目、以及返回值为 Boolean 的 stop 方法，你可以使用此 stop 方法来停止 MyBatis 加载更多的结果。

使用 ResultHandler 的时候需要注意以下两种限制：

- 从被 ResultHandler 调用的方法返回的数据不会被缓存。

- 当使用结果映射集（resultMap）时，MyBatis 大多数情况下需要数行结果来构造外键对象。如果你正在使用 ResultHandler，你可以给出外键（association）或者集合（collection）尚未赋值的对象。

批量立即更新方法

有一个方法可以刷新（执行）存储在 JDBC 驱动类中的批量更新语句。当你将 `ExecutorType.BATCH` 作为 `ExecutorType` 使用时可以采用此方法。

```
List<BatchResult> flushStatements()
```

事务控制方法

控制事务作用域有四个方法。当然，如果你已经设置了自动提交或你正在使用外部事务管理器，这就没有任何效果了。然而，如果你正在使用 JDBC 事务管理器，由 Connection 实例来控制，那么这四个方法就会派上用场：

```
void commit()

void commit(boolean force)
```

```
void rollback()

void rollback(boolean force)
```

默认情况下 MyBatis 不会自动提交事务，除非它侦测到有插入、更新或删除操作改变了数据库。如果你已经做出了一些改变而没有使用这些方法，那么你可以传递 true 值到 commit 和 rollback 方法来保证事务被正常处理（注意，在自动提交模式或者使用了外部事务管理器的情况下设置 force 值对 session 无效）。很多时候你不用调用 rollback()，因为 MyBatis 会在你没有调用 commit 时替你完成回滚操作。然而，如果你需要在支持多提交和回滚的 session 中获得更多细粒度控制，你可以使用回滚操作来达到目的。

注意 MyBatis-Spring 和 MyBatis-Guice 提供了声明事务处理，所以如果你在使用 Mybatis 的同时使用了 Spring 或者 Guice，那么请参考它们的手册以获取更多的内容。

本地缓存

Mybatis 使用到了两种缓存：本地缓存（local cache）和二级缓存（second level cache）。

每当一个新 session 被创建，MyBatis 就会创建一个与之相关联的本地缓存。任何在 session 执行过的查询语句本身都会被保存在本地缓存中，那么，相同的查询语句和相同的参数所产生的更改就不会二度影响数据库了。本地缓存会被增删改、提交事务、关闭事务以及关闭 session 所清空。

默认情况下，本地缓存数据可在整个 session 的周期内使用，这一缓存需要被用来解决循环引用错误和加快重复嵌套查询的速度，所以它可以不被禁用掉，但是你可以设置 localCacheScope=STATEMENT 表示缓存仅在语句执行时有效。

注意，如果 localCacheScope 被设置为 SESSION，那么 MyBatis 所返回的引用将传递给保存在本地缓存里的相同对象。对返回的对象（例如 list）做出任何更新将会影响本地缓存的内容，进而影响存活在 session 生命周期中的缓存所返回的值。因此，不要对 MyBatis 所返回的对象作出更改，以防后患。

你可以随时调用以下方法来清空本地缓存：

```
void clearCache()
```

确保 SqlSession 被关闭

```
void close()
```

你必须保证的最重要的事情是你要关闭所打开的任何 session。保证做到这点的最佳方式是下面的工作模式:

```
SqlSession session = sqlSessionFactory.openSession();

try {

    // following 3 lines pseudocod for "doing some work"

    session.insert(...);

    session.update(...);

    session.delete(...);

    session.commit();

} finally {

    session.close();

}
```

还有，如果你正在使用 jdk 1.7 以上的版本还有 MyBatis 3.2 以上的版本，你可以使用 try-with-resources 语句：

```
try (SqlSession session = sqlSessionFactory.openSession()) {

    // following 3 lines pseudocode for "doing some work"

    session.insert(...);

    session.update(...);

    session.delete(...);

    session.commit();

}
```

注意 就像 SqlSessionFactory，你可以通过调用当前使用中的 SqlSession 的 getConfiguration 方法来获得 Configuration 实例。

```
Configuration getConfiguration()
```

使用映射器

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

上述的各个 insert、update、delete 和 select 方法都很强大，但也有些繁琐，可能会产生类型安全问题并且对于你的 IDE 和单元测试也没有实质性的帮助。在上面的入门章节中我们已经看到了一个使用映射器的示例。

因此，一个更通用的方式来执行映射语句是使用映射器类。一个映射器类就是一个仅需声明与 SqlSession 方法相匹配的方法的接口类。下面的示例展示了一些方法签名以及它们是如何映射到 SqlSession 上的。

```
public interface AuthorMapper {

  // (Author) selectOne("selectAuthor",5);

  Author selectAuthor(int id);

  // (List<Author>) selectList(“selectAuthors”)

  List<Author> selectAuthors();

  // (Map<Integer,Author>) selectMap("selectAuthors", "id")

  @MapKey("id")

  Map<Integer, Author> selectAuthors();

  // insert("insertAuthor", author)

  int insertAuthor(Author author);

  // updateAuthor("updateAuthor", author)

  int updateAuthor(Author author);

  // delete("deleteAuthor",5)

  int deleteAuthor(int id);

}
```

总之，每个映射器方法签名应该匹配相关联的 SqlSession 方法，而字符串参数 ID 无需匹配。相反，方法名必须匹配映射语句的 ID。

此外，返回类型必须匹配期望的结果类型，单返回值时为所指定类的值，多返回值时为数组或集合。所有常用的类型都是支持的，包括：原生类 型、Map、POJO 和 JavaBean。

注意 映射器接口不需要去实现任何接口或继承自任何类。只要方法可以被唯一标识对应的映射语句就可以了。

注意 映射器接口可以继承自其他接口。当使用 XML 来构建映射器接口时要保证语句被包含在合适的命名空间中。而且，唯一的限制就是你不能在两个继承关系的接口中拥有相同的方法签名（潜在的危险做法不可取）。

你可以传递多个参数给一个映射器方法。如果你这样做了，默认情况下它们将会以 "param" 字符串紧跟着它们在参数列表中的位置来命名，比如：#{param1}、#{param2} 等。如果你想改变参数的名称（只在多参数情况下），那么你可以在参数上使用 @Param("paramName") 注解。

你也可以给方法传递一个 RowBounds 实例来限制查询结果。

映射器注解

因为最初设计时，MyBatis 是一个 XML 驱动的框架。配置信息是基于 XML 的，而且映射语句也是定义在 XML 中的。而到了 MyBatis 3，就有新选择了。MyBatis 3 构建在全面且强大的基于 Java 语言的配置 API 之上。这个配置 API 是基于 XML 的 MyBatis 配置的基础，也是新的基于注解配置的基础。注解提供了一种简单的方式来实现简单映射语句，而不会引入大量的开销。

注意 不幸的是，Java 注解的的表达力和灵活性十分有限。尽管很多时间都花在调查、设计和试验上，最强大的 MyBatis 映射并不能用注解来构建——并不是在开玩笑，的确是这样。比方说，C#属性就没有这些限制，因此 MyBatis.NET 将会比 XML 有更丰富的选择。也就是说，基于 Java 注解的配置离不开它的特性。

注解如下表所示：

注解	使用对象	相对应的 XML	描述
`@CacheNamespace`	类	`<cache>`	为给定的命名空间（比如类）配置缓存。属性有
`@Property`	N/A	`<property>`	指定参数值或占位值（placeholder）（能被 `myba`
`@CacheNamespaceRef`	类	`<cacheRef>`	参照另外一个命名空间的缓存来使用。属性有：`v` 定命名空间（命名空间名就是指定的 Java 类型的
`@ConstructorArgs`	方法	`<constructor>`	收集一组结果传递给一个结果对象的构造方法。属
`@Arg`	N/A	• `<arg>` • `<idArg>`	单参数构造方法，是 ConstructorArgs 集合的一部 用于比较的属性，和`<idArg>` XML 元素相似。
`@TypeDiscriminator`	方法	`<discriminator>`	一组实例值被用来决定结果映射的表现。属性有
`@Case`	N/A	`<case>`	单独实例的值和它对应的映射。属性有：`value`,
`@Results`	方法	`<resultMap>`	结果映射的列表，包含了一个特别结果列如何被映
`@Result`	N/A	• `<result>` • `<id>`	在列和属性或字段之间的单独结果映射。属性有 XML 映射中的`<id>`相似）的属性。one 属性是单 免名称冲突。
`@One`	N/A	`<association>`	复杂类型的单独属性值映射。属性有：`selec` 数 `lazyLoadingEnabled`。注意联合映射在注
`@Many`	N/A	`<collection>`	映射到复杂类型的集合属性。属性有：`select` 数 `lazyLoadingEnabled`。注意 联合映射在注
`@MapKey`	方法		这是一个用在返回值为 Map 的方法上的注解。它 key 值。
`@Options`	方	映射语句的属性	这个注解提供访问大范围的交换和配置选项的入口

注解	使用对象	相对应的 XML	描述
	法		属性有：useCache=true, flushCache=1, timeout=-1, useGeneratedKeys=false 你使用了 Options 注解，你的语句就会被上述属 注意： keyColumn 属性只在某些数据库中
• @Insert • @Update • @Delete • @Select	方法	• <insert> • <update> • <delete> • <select>	这四个注解分别代表将会被执行的 SQL 语句。它的字符串。这有效避免了以 Java 代码构建 SQL 的字符串数组。
• @InsertProvider • @UpdateProvider • @DeleteProvider • @SelectProvider	方法	• <insert> • <update> • <delete> • <select>	允许构建动态 SQL。这些备选的 SQL 注解允许你返回类型返回值了。）当执行映射语句的时候，M interface type" 和 "Mapper method" 会经过 Pr 有： type, method。type 属性需填入类。met
@Param	参数	N/A	如果你的映射方法的形参有多个，这个注解使用在作前缀，再加上它们的参数位置作为参数别名。例
@SelectKey	方法	<selectKey>	这个注解的功能与 <selectKey> 标签完全一致个注解的方法上作 @SelectKey 注解则视为无效属性。属性有：statement 填入将会被执行的句应被在插入语句的之前还是之后执行。resu 的 STATEMENT、 PREPARED 或 CALLABLE 中
@ResultMap	方法	N/A	这个注解给 @Select 或者 @SelectProvider select 注解中还存在 @Results 或者 @Constru
@ResultType	方法	N/A	此注解在使用了结果处理器的情况下使用。在这种请使用 @ResultMap 注解。如果结果类型在 XM 用结果处理器的方法上，那么返回类型必须是 vo
@Flush	方法	N/A	如果使用了这个注解，定义在 Mapper 接口中的

映射申明样例

这个例子展示了如何使用 @SelectKey 注解来在插入前读取数据库序列的值：

```
@Insert("insert into table3 (id, name) values(#{nameId}, #{name})")

@SelectKey(statement="call next value for TestSequence", keyProperty="nameI
d", before=true, resultType=int.class)

int insertTable3(Name name);
```

这个例子展示了如何使用 @SelectKey 注解来在插入后读取数据库识别列的值：

```
@Insert("insert into table2 (name) values(#{name})")

@SelectKey(statement="call identity()", keyProperty="nameId", before=false,
 resultType=int.class)

int insertTable2(Name name);
```

这个例子展示了如何使用 `@Flush` 注解去调用 `SqlSession#flushStatements()`：

```
@Flush

List<BatchResult> flush();
```

这些例子展示了如何通过指定 @Result 的 id 属性来命名结果集：

```
@Results(id = "userResult", value = {

  @Result(property = "id", column = "uid", id = true),

  @Result(property = "firstName", column = "first_name"),

  @Result(property = "lastName", column = "last_name")

})

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

User getUserById(Integer id);



@Results(id = "companyResults")

@ConstructorArgs({
```

```
  @Arg(property = "id", column = "cid", id = true),
  @Arg(property = "name", column = "name")
})
@Select("select * from company where id = #{id}")
Company getCompanyById(Integer id);
```

这个例子展示了单一参数使用 @SqlProvider 注解：

```
@SelectProvider(type = UserSqlBuilder.class, method = "buildGetUsersByName
")
List<User> getUsersByName(String name);


class UserSqlBuilder {
  public static String buildGetUsersByName(final String name) {
    return new SQL(){{
      SELECT("*");
      FROM("users");
      if (name != null) {
        WHERE("name like #{value} || '%'");
      }
      ORDER_BY("id");
    }}.toString();
  }
}
```

这个例子展示了多参数使用 @SqlProvider 注解：

```
@SelectProvider(type = UserSqlBuilder.class, method = "buildGetUsersByName
")

List<User> getUsersByName(

    @Param("name") String name, @Param("orderByColumn") String orderByColum
n);



class UserSqlBuilder {



  // If not use @Param, you should be define same arguments with mapper meth
od

  public static String buildGetUsersByName(

      final String name, final String orderByColumn) {

    return new SQL(){{

      SELECT("*");

      FROM("users");

      WHERE("name like #{name} || '%'");

      ORDER_BY(orderByColumn);

    }}.toString();

  }



  // If use @Param, you can define only arguments to be used

  public static String buildGetUsersByName(@Param("orderByColumn") final Str
ing orderByColumn) {

    return new SQL(){{
```

```
      SELECT("*");

      FROM("users");

      WHERE("name like #{name} || '%'");

      ORDER_BY(orderByColumn);

    }}.toString();

  }

}
```

SQL 语句构建器类

问题

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

```
String sql = "SELECT P.ID, P.USERNAME, P.PASSWORD, P.FULL_NAME, "

"P.LAST_NAME,P.CREATED_ON, P.UPDATED_ON " +

"FROM PERSON P, ACCOUNT A " +

"INNER JOIN DEPARTMENT D on D.ID = P.DEPARTMENT_ID " +

"INNER JOIN COMPANY C on D.COMPANY_ID = C.ID " +

"WHERE (P.ID = A.ID AND P.FIRST_NAME like ?) " +

"OR (P.LAST_NAME like ?) " +

"GROUP BY P.ID " +

"HAVING (P.LAST_NAME like ?) " +
```

```
"OR (P.FIRST_NAME like ?) " +

"ORDER BY P.ID, P.FULL_NAME";
```

The Solution

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

```
private String selectPersonSql() {

  return new SQL() {{

    SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FULL_NAME");

    SELECT("P.LAST_NAME, P.CREATED_ON, P.UPDATED_ON");

    FROM("PERSON P");

    FROM("ACCOUNT A");

    INNER_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT_ID");

    INNER_JOIN("COMPANY C on D.COMPANY_ID = C.ID");

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

    WHERE("P.FIRST_NAME like ?");

    OR();

    WHERE("P.LAST_NAME like ?");

    GROUP_BY("P.ID");

    HAVING("P.LAST_NAME like ?");

    OR();

    HAVING("P.FIRST_NAME like ?");

    ORDER_BY("P.ID");

    ORDER_BY("P.FULL_NAME");
```

```
  }}.toString();

}
```

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

SQL 类

这里给出一些示例：

```
// Anonymous inner class

public String deletePersonSql() {

  return new SQL() {{

    DELETE_FROM("PERSON");

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

  }}.toString();

}



// Builder / Fluent style

public String insertPersonSql() {

  String sql = new SQL()

    .INSERT_INTO("PERSON")

    .VALUES("ID, FIRST_NAME", "#{id}, #{firstName}")

    .VALUES("LAST_NAME", "#{lastName}")

    .toString();

  return sql;

}
```

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

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

  return new SQL() {{

    SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FIRST_NAME, P.LAST_NAME");

    FROM("PERSON P");

    if (id != null) {

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

    }

    if (firstName != null) {

      WHERE("P.FIRST_NAME like #{firstName}");

    }

    if (lastName != null) {

      WHERE("P.LAST_NAME like #{lastName}");

    }

    ORDER_BY("P.LAST_NAME");

  }}.toString();

}



public String deletePersonSql() {

  return new SQL() {{

    DELETE_FROM("PERSON");
```

```
    WHERE("ID = #{id}");
  }}.toString();
}

public String insertPersonSql() {
  return new SQL() {{
    INSERT_INTO("PERSON");
    VALUES("ID, FIRST_NAME", "#{id}, #{firstName}");
    VALUES("LAST_NAME", "#{lastName}");
  }}.toString();
}

public String updatePersonSql() {
  return new SQL() {{
    UPDATE("PERSON");
    SET("FIRST_NAME = #{firstName}");
    WHERE("ID = #{id}");
  }}.toString();
}
```

方法	描述
• `SELECT(String)` • `SELECT(String...)`	开始或插入到 `SELECT` 子句。 可以被多次调用，参数也会添加

方法	描述
• `SELECT_DISTINCT(String)` • `SELECT_DISTINCT(String...)`	开始或插入到 `SELECT` 子句， 也可以插入 `DISTINCT` 关键字到 可以是数据库驱动程序接受的任意类型。
• `FROM(String)` • `FROM(String...)`	开始或插入到 `FROM` 子句。 可以被多次调用，参数也会添加到
• `JOIN(String)` • `JOIN(String...)` • `INNER_JOIN(String)` • `INNER_JOIN(String...)` • `LEFT_OUTER_JOIN(String)` • `LEFT_OUTER_JOIN(String...)` • `RIGHT_OUTER_JOIN(String)` • `RIGHT_OUTER_JOIN(String...)`	基于调用的方法，添加新的合适类型的 `JOIN` 子句。 参数可以
• `WHERE(String)` • `WHERE(String...)`	插入新的 `WHERE` 子句条件， 由 `AND` 链接。可以多次被调用，
`OR()`	使用 `OR` 来分隔当前的 `WHERE` 子句条件。 可以被多次调用，但
`AND()`	使用 `AND` 来分隔当前的 `WHERE` 子句条件。 可以被多次调用， 了完整性才被使用。
• `GROUP_BY(String)` • `GROUP_BY(String...)`	插入新的 `GROUP BY` 子句元素，由逗号连接。 可以被多次调用
• `HAVING(String)` • `HAVING(String...)`	插入新的 `HAVING` 子句条件。 由 AND 连接。可以被多次调用，
• `ORDER_BY(String)` • `ORDER_BY(String...)`	插入新的 `ORDER BY` 子句元素， 由逗号连接。可以多次被调用
`DELETE_FROM(String)`	开始一个 delete 语句并指定需要从哪个表删除的表名。通常它后
`INSERT_INTO(String)`	开始一个 insert 语句并指定需要插入数据的表名。后面都会跟着

方法	描述
• `SET(String)` • `SET(String...)`	针对 update 语句，插入到"set"列表中
`UPDATE(String)`	开始一个 update 语句并指定需要更新的表明。后面都会跟着一
`VALUES(String, String)`	插入到 insert 语句中。第一个参数是要插入的列名，第二个参数
`INTO_COLUMNS(String...)`	Appends columns phrase to an insert statement. This should be
`INTO_VALUES(String...)`	Appends values phrase to an insert statement. This should be cal

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

```
public String selectPersonSql() {

  return new SQL()

    .SELECT("P.ID", "A.USERNAME", "A.PASSWORD", "P.FULL_NAME", "D.DEPARTMENT
_NAME", "C.COMPANY_NAME")

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

    .INNER_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT_ID", "COMPANY C on D.CO
MPANY_ID = C.ID")

    .WHERE("P.ID = A.ID", "P.FULL_NAME like #{name}")

    .ORDER_BY("P.ID", "P.FULL_NAME")

    .toString();

}



public String insertPersonSql() {

  return new SQL()

    .INSERT_INTO("PERSON")

    .INTO_COLUMNS("ID", "FULL_NAME")
```

```
    .INTO_VALUES("#{id}", "#{fullName}")
    .toString();
}

public String updatePersonSql() {
  return new SQL()
    .UPDATE("PERSON")
    .SET("FULL_NAME = #{fullName}", "DATE_OF_BIRTH = #{dateOfBirth}")
    .WHERE("ID = #{id}")
    .toString();
}
```

SqlBuilder 和 SelectBuilder (已经废弃)

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

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

方法	描述
`BEGIN()` /`RESET()`	这些方法清空 SelectBuilder 类的 ThreadLocal 状态，并且准备一个新的构建语句。开始 理语句 `RESET()` 读取得最好。
`SQL()`	返回生成的 `SQL()` 并重置 `SelectBuilder` 状态 (好像 `BEGIN()` 或 `RESET()` 被

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

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

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

这就允许像下面这样来创建方法：

```
/* DEPRECATED */

public String selectBlogsSql() {

  BEGIN(); // Clears ThreadLocal variable

  SELECT("*");

  FROM("BLOG");

  return SQL();

}



/* DEPRECATED */

private String selectPersonSql() {

  BEGIN(); // Clears ThreadLocal variable

  SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FULL_NAME");

  SELECT("P.LAST_NAME, P.CREATED_ON, P.UPDATED_ON");

  FROM("PERSON P");

  FROM("ACCOUNT A");

  INNER_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT_ID");

  INNER_JOIN("COMPANY C on D.COMPANY_ID = C.ID");

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

  WHERE("P.FIRST_NAME like ?");

  OR();

  WHERE("P.LAST_NAME like ?");
```

```
  GROUP_BY("P.ID");

  HAVING("P.LAST_NAME like ?");

  OR();

  HAVING("P.FIRST_NAME like ?");

  ORDER_BY("P.ID");

  ORDER_BY("P.FULL_NAME");

  return SQL();

}
```

日志

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

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

MyBatis 内置日志工厂基于运行时自省机制选择合适的日志工具。它会使用第一个查找得到的工具（按上文列举的顺序查找）。如果一个都未找到，日志功能就会被禁用。

不少应用服务器（如 Tomcat 和 WebShpere）的类路径中已经包含 Commons Logging，所以在这种配置环境下的 MyBatis 会把它作为日志工具，记住这点非常重要。这将意味着，在诸如 WebSphere 的环境中，它提供了 Commons Logging 的私有实现，你的 Log4J 配置将被忽略。MyBatis 将你的 Log4J 配置忽略掉是相当令人郁闷的（事实上，正是因为在这种配置环境下，MyBatis 才会选择使用 Commons Logging 而不是 Log4J）。如果你的应用部署在一个类路径已经包含 Commons Logging 的环境中，而你又想使用其它日志工具，你可以通过在 MyBatis 配置文件 mybatis-config.xml 里面添加一项 setting 来选择别的日志工具。

```
<configuration>
```

```
  <settings>

    ...

    <setting name="logImpl" value="LOG4J"/>

    ...

  </settings>

</configuration>
```

logImpl 可选的值有：SLF4J、LOG4J、LOG4J2、JDK_LOGGING、COMMONS_LOGGING、STDOUT_LOGGING、NO_LOGGING，或者是实现了接口 `org.apache.ibatis.logging.Log` 的，且构造方法是以字符串为参数的类的完全限定名。（译者注：可以参考 org.apache.ibatis.logging.slf4j.Slf4jImpl.java 的实现）

你也可以调用如下任一方法来使用日志工具：

```
org.apache.ibatis.logging.LogFactory.useSlf4jLogging();

org.apache.ibatis.logging.LogFactory.useLog4JLogging();

org.apache.ibatis.logging.LogFactory.useJdkLogging();

org.apache.ibatis.logging.LogFactory.useCommonsLogging();

org.apache.ibatis.logging.LogFactory.useStdOutLogging();
```

如果你决定要调用以上某个方法，请在调用其它 MyBatis 方法之前调用它。另外，仅当运行时类路径中存在该日志工具时，调用与该日志工具对应的方法才会生效，否则 MyBatis 一概忽略。如你环境中并不存在 Log4J，你却调用了相应的方法，MyBatis 就会忽略这一调用，转而以默认的查找顺序查找日志工具。

关于 SLF4J、Apache Commons Logging、Apache Log4J 和 JDK Logging 的 API 介绍不在本文档介绍范围内。不过，下面的例子可以作为一个快速入门。关于这些日志框架的更多信息，可以参考以下链接：

- Apache Commons Logging
- Apache Log4j

- JDK Logging API

日志配置

你可以对包、映射类的全限定名、命名空间或全限定语句名开启日志功能来查看 MyBatis 的日志语句。

再次说明下，具体怎么做，由使用的日志工具决定，这里以 Log4J 为例。配置日志功能非常简单：添加一个或多个配置文件（如 log4j.properties），有时需要添加 jar 包（如 log4j.jar）。下面的例子将使用 Log4J 来配置完整的日志服务，共两个步骤：

步骤 1：添加 Log4J 的 jar 包

因为我们使用的是 Log4J，就要确保它的 jar 包在应用中是可用的。要启用 Log4J，只要将 jar 包添加到应用的类路径中即可。Log4J 的 jar 包可以在上面的链接中下载。

对于 web 应用或企业级应用，则需要将 `log4j.jar` 添加到 `WEB-INF/lib` 目录下；对于独立应用，可以将它添加到 JVM 的 `-classpath` 启动参数中。

步骤 2：配置 Log4J

配置 Log4J 比较简单，假如你需要记录这个映射器接口的日志：

```
package org.mybatis.example;

public interface BlogMapper {

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

  Blog selectBlog(int id);

}
```

在应用的类路径中创建一个名称为 `log4j.properties` 的文件，文件的具体内容如下：

```
# Global logging configuration

log4j.rootLogger=ERROR, stdout
```

```
# MyBatis logging configuration...

log4j.logger.org.mybatis.example.BlogMapper=TRACE

# Console output...

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

添加以上配置后，Log4J 就会记录 `org.mybatis.example.BlogMapper` 的详细执行操作，且仅记录应用中其它类的错误信息（若有）。

你也可以将日志的记录方式从接口级别切换到语句级别，从而实现更细粒度的控制。如下配置只对 `selectBlog` 语句记录日志：

```
log4j.logger.org.mybatis.example.BlogMapper.selectBlog=TRACE
```

与此相对，可以对一组映射器接口记录日志，只要对映射器接口所在的包开启日志功能即可：

```
log4j.logger.org.mybatis.example=TRACE
```

某些查询可能会返回庞大的结果集，此时只想记录其执行的 SQL 语句而不想记录结果该怎么办？为此，Mybatis 中 SQL 语句的日志级别被设为 DEBUG（JDK 日志设为 FINE），结果的日志级别为 TRACE（JDK 日志设为 FINER)。所以，只要将日志级别调整为 DEBUG 即可达到目的：

```
log4j.logger.org.mybatis.example=DEBUG
```

要记录日志的是类似下面的映射器文件而不是映射器接口又该怎么做呢？

```
<?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>
```

如需对 XML 文件记录日志，只要对命名空间增加日志记录功能即可：

```
log4j.logger.org.mybatis.example.BlogMapper=TRACE
```

要记录具体语句的日志可以这样做：

```
log4j.logger.org.mybatis.example.BlogMapper.selectBlog=TRACE
```

你应该注意到了，为映射器接口和 XML 文件添加日志功能的语句毫无差别。

注意 如果你使用的是 SLF4J 或 Log4j 2，MyBatis 将以 MYBATIS 这个值进行调用。

配置文件 `log4j.properties` 的余下内容是针对日志输出源的，这一内容已经超出本文档范围。关于 Log4J 的更多内容，可以参考 Log4J 的网站。不过，你也可以简单地做做实验，看看不同的配置会产生怎样的效果。