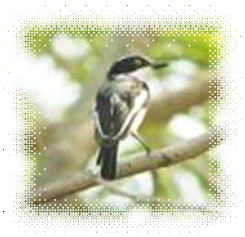

MyBatis 3

用户指南

V 1.0

罗利辉 译

2010.08.01

关于从本文档复制代码的警告

其实，这不是一个什么法律警告，这仅仅是一个提示。现代文字处理器做了出色的工作，以非常优美的方式使文本具有可读性和具有良好格式化。然而，有时看起来是一个您完全想要的代码示例，也往往完全毁于插入了特殊字符。 “引号”和连字符就是一个很好的例子，在文档中看到引号和连字符，拷贝到 IDE 或编辑器中却不能很好工作，至少不是您期望的方式。

因此，阅读这份文档指南并享受它，希望它有助于您。当您拷贝的代码来自本文档代码示例时，最好找出附带下载的例子（包括单元测试等），或是网站或邮件列表的例子。

如何更好地使用本文档…

如果您发现这个文档存在任何不足，或者缺少了对某个特性的描述，最好的方式是先学习它，然后自己写一份文档。

我们接受公众撰写的文档，并通过下列网址上传：

http://opensource.atlassian.com/confluence/oss/display/IBATIS/Contribute+Documentation

如果您的文档写的很棒，人们将会喜欢您并阅读您的文档!

关于本文

本文是《MyBatis3 用户指南》中文版，为学习与研究从《MyBatis-3-User-Guide》翻译而来，仅供大家参考，最权威的应以官方英文文档为准。如果发现翻译有错误，欢迎指正，请发送邮件到 lihui.luo@163.com。谢谢。

翻译过程中，直接对英文版进行了勘误，或者添加了一些帮助理解的注释。同时，文档最后也增加了几节附录的内容，希望对学习 MyBatis3有用。

Contents

MyBatis 是什么?

MyBatis 是一款一流的支持自定义 SQL、存储过程和高级映射的持久化框架。MyBatis 几乎消除了所有的 JDBC 代码，也基本不需要手工去设置参数和获取检索结果。MyBatis 能够使用简单的 XML 格式或者注解进行来配置，能够映射基本数据元素、Map 接口和 POJOs（普通 java 对象）到数据库中的记录。

准备开始

所有的 MyBatis 应用都以 SqlSessionFactory 实例为中心。SqlSessionFactory 实例通过 SqlSessionFactoryBuilder 来获得，SqlSessionFactoryBuilder 能够从 XML 配置文件或者通过自定义编写的配置类（Configuration class），来创建一个 SqlSessionFactory 实例。

从 XML 中创建 SqlSessionFactory 实例

从 XML 中创建 SqlSessionFactory 实例非常简单。建议您使用类资源路径（classpath resource）来加载配置文件，但是您也能够使用任何方式，包括文本文件路径或者以 file:// 开头 URL 的方式。MyBatis 包括一个叫做 Resources 的工具类（utility class），其中包含了一系列方法，使之能简单地从 classpath 或其它地方加载配置文件。

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

XML 配置文件包含 MyBatis 框架的核心设置，包括获取数据库连接的 DataSource 实例，和包括决定事务作用域范围和控制的事务管理等。您将能够在后面的章节中找到详细的 XML 配置，在这里我们先展示一个简单的例子：

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

```
  <mappers>
    <mapper resource="org/mybatis/example/BlogMapper.xml"/>
  </mappers>
</configuration>
```

虽然 XML 配置文件中还有很多其它的配置细节，但是，上面的示例显示了最重要的部分。注意 XML 配置文件的头部，会使用 DTD 验证文档来验证该 XML 配置文件。body 部分的 environment 元素，包含了事务管理和连接池配置。Mappers 元素指定了映射配置文件--包含 SQL 语句和映射定义的 XML 文件。

如何不使用 XML 来创建 SqlSessionFactory

如果您喜欢直接通过 java 代码而不是通过 XML 创建配置选项，或者想创建您自己的配置生成器。MyBatis 提供了一个完整的配置类（Configuration class），它提供了与 XML 文件相同的配置选项。

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

请注意，这种方式下的配置添加一个映射类（mapper class）。映射类是包含 SQL 映射注解的 Java 类，从而避免了使用 XML。但是，由于注解的一些局限性以及 MyBatis 映射的复杂性，XML 仍然是一些高级的映射功能（如嵌套连接映射，Nested Join Mapping）所必须的方式。基于这个原因，如果存在 XML 文件，MyBatis 自动寻找并加载这个 XML 文件。在这种情况下，BlogMapper.xml 将会被类路径下名称为 BlogMapper.class 的类加载。详述请见后面章节。

从 SqlSessionFactory 获取 SqlSession

现在您已经创建了一个 SqlSessionFactory（指上面的 sqlMapper），正如它名字暗示那样，您可以通过它来创建一个 SqlSession 实例。SqlSession 包含了所有执行数据库 SQL 语句的方法。您能够直接地通过 SqlSession 实例执行映射 SQL 语句。例如：

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

虽然这种方法很有效，MyBatis 以前版本的用户对此也可能很熟悉，但现在有一个更简便的方式，那就是对给定的映射语句，使用一个正确描述参数与返回值的接口（如 BlogMapper.class），您就能更清晰地执行类型安全的代码，从而避免错误和异常。 如：

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

现在，让我们一起探索它们究竟是如何执行的。

探索映射 SQL 语句

此时，您可能想知道 SqlSession 或者映射器类（Mapper class）是怎样执行的。映射 SQL 语句是一个很大的主题，该主题将可能占据本文档的大部分内容。但是，为了让您看到它是怎样运行的，这里举两个例子。

在上面的例子中，映射语句已经在 XML 配置文件或注解中定义。让我们首先来看看 XML 配置文件，所有 MyBatis 提供的功能特性都可以通过基于 XML 映射配置文件配置来实现。如果您以前使用过 MyBatis，对此就会很熟悉。但许多改进使 XML 映射文件变得更简洁清晰。下面是一个基于 XML 映射配置的例子，满足上述 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" parameterType="int" resultType="Blog">
    select * from Blog where id = #{id}
  </select>
</mapper>
```

这个简单的例子看起来有不少的开销，但它确实非常地轻巧，只要您喜欢，您可以在一个映射 XML 文件中定义许多映射语句。虽然会有一些 XML 头部和 DOCTYPE 声明，但该文件余下的部分是自解（self explanatory，可理解为不加解释就能明白）的。它定义了映射语句的名称“selectBlog”，在命名空间“org.mybatis.example.BlogMapper”，允许您通过指定完整类名“org.mybatis.example.BlogMapper”来访问上面的例子：

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

这非常类似 java 中通过完整类名来调用方法，而这样做是有原因的。这个名称可以直接映射到一个具在相同命名空间的映射类，这个映射类有一个方法的名称、参数及返回类型都与 select 映射语句相匹配。正如您前面看到的，这使您很简单地调用映射类里的方法，下面是另外的一个例子：

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

第二种方法有很多好处。第一，它不依赖于字符串，所以更安全。第二，如果您的 IDE 有自动完成功能，您可以利用这功能很快导航到您的映射 SQL 语句。第三，您不需要关注返回类型，不需要进行强制转换，因为使用 BlogMapper 接口已经限定了返回类型，它会安全地返回。

关于命名空间

➔ **命名空间**:在以前的版本中是可选的、复杂的且没多大用处。但在这个版本中，命名空间是必须的，并且不仅仅是简单地使用完整类名来隔离区分语句。

正如您看到的那样，命名空间能够进行接口绑定，即使您认为现在不会使用到它，但您应该按照这些准则来做。一旦使用了命名空间并且放入适当的 java 包命名空间中将会使您的代码清晰，并提高 MyBatis 的长期可用性。

➔ **名称解析**: 为了减少大量地输入， MyBatis 对所有的配置元素，包括 statements、result maps、 caches 等使用下面的名称解析规则:

- 完整类名 ：(例如: “com.mypackage.MyMapper.selectAllThings”)可用来直接查找并使用。
- 短名称： (例如: “selectAllThings”)可用来引用明确的实体对象。但是，如果出现有两个或更多（例如“com.foo.selectAllThings 和 com.bar.selectAllThings”）实体对象，您将收到一个错误报告（短名称含糊不清），因此，这时就必须使用完整类名。

对映射类还有一个更好的方法,就像前面的 BlogMapper。它们的映射语句不需要完全在 XML 中配置。相反，它们可以使用 Java 注解。例如上面的 XML 配置可以替换为：

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

对简单的映射语句，使用注解可以显得非常地清晰。但是 java 注解本身的局限难于应付更复杂的语句。如果您准备要做某些复杂的事情，最好使用 XML 文件来配置映射语句。

这将由您和您的项目小组来确定哪种方式是适合的，以一致的方式来定义映射语句是非常重要的。也就是说，您不会被限于仅用一种方式。您能够非常容易地从基于注解的映射语句移植到 XML 配置文件中，反之亦然。

作用域和生命周期

理解到目前为止所讨论的类的作用域和生命周期是非常重要的。如果使用不当可导致严重的并发性问题。

SqlSessionFactoryBuilder

这个类可以在任何时候被实例化、使用和销毁。一旦您创造了 SqlSessionFactory 就不需要再保留它了。所以 SqlSessionFactoryBuilder 实例的最好的作用域是方法体内（即一个本地方法变量）。您能重用 SqlSessionFactoryBuilder 创建多个 SqlSessionFactory 实例，但最好不要把时间、资源放在解析 XML 文件上，而是要从中解放出来做最重要事情。

SqlSessionFactory

一旦创建，SqlSessionFactory 将会存在于您的应用程序整个运行生命周期中。很少或根本没有理由去销毁它或重新创建它。最佳实践是不要在一个应用中多次创建 SqlSessionFactory。这样做会被视为“没品味”。所是 SqlSessionFactory 最好的作用域范围是一个应用的生命周期范围。这可以由多种方式来实现，最简单的方式是使用 Singleton 模式或静态 Singleton 模式。但这不是被广泛接受的最佳做法，相反，您可能更愿意使用像 Google Guice 或 Spring 的依赖注入方式。这些框架允许您创造一个管理器，用于管理 SqlSessionFactory 的生命周期。

SqlSession

每个线程都有一个 SqlSession 实例，SqlSession 实例是不被共享的，并且不是线程安全的。因此最好的作用域是 request 或者 method。决不要用一个静态字段或者一个类的实例字段来保存 SqlSession 实例引用。也不要用任何一个管理作用域，如 Servlet 框架中的 HttpSession，来保存 SqlSession 的引用。如果您正在用一个 WEB 框架，可以把 SqlSession 的作用域看作类似于 HTTP 的请求范围。也就是说，在收到一个 HTTP 请求，您可以打开一个 SqlSession，当您把 response 返回时，就可以把 SqlSession 关闭。关闭会话是非常重要的，您应该要确保会话在一个 finally 块中被关闭。

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

在您的代码里都使用这一模式将保证所有的数据库资源被正确地关闭（假如您没有把您自己的数据库连接传递给 MyBatis 管理，这就对 MyBatis 表明您希望自己管理连接）。

Mapper 实例

Mappers 是创建来绑定映射语句的接口，该 Mapper 实例是从 SqlSession 得到的。因此，所有 mapper 实例的作用域跟创建它的 SqlSession 一样。但是，mapper 实例最好的作用域是 method，也就是它们应该在方法内被调用，使用完即被销毁。并且 mapper 实例不用显式地被关闭。虽然把 mapper 实例保持在一个 request 范围（与 SqlSession 相似）不会产生太大的问题，但是您可能会发现，在这个层次上管理太多资源可能会失控。保持简单，就是让 Mappers 保持在一个方法内。下面的例子演示了这种做法。

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

Mapper XML 配置

MyBatis 的 XML 配置文件包含了设置和影响 MyBatis 行为的属性。XML 配置文件的层次结构如下：

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

- mappers

properties 元素

它们都是外部化，可替代的属性。可以配置在一个典型的 Java 属性文件中，或者通过 properties 元素的子元素进行配置。例如：

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

在整个配置文件中，这些属性能够被可动态替换（即使用占位符）的属性值引用，例如：

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

示例中的 username 和 password 将会被替换为配置在 properties 元素中的相应值。driver 和 url 属性则会被 config.properties 文件中的相应值替换。这里提供了大量的配置选项。

这些属性也可以传递给 sqlSessionFactoryBuilder.build()方法。例如：

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

// ... or ...

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

如果一个属性存在于多个地方，MyBatis 将使用下面的顺序加载：

- 首先读入 properties 元素主体中指定的属性。
- 然后会加载类路径或者 properties 元素中指定的 url 的资源文件属性。它会覆盖前面已经读入的重复属性。
- 通过方法参数来传递的属性将最后读取(即通过 sqlSessionFactoryBuilder.build)，同样也会覆盖从 properties 元素指定的和 resource/url 指定的重复属性。

因此最优先的属性是通过方法参数来传递的属性，然后是通过 resource/url 配置的属性，最后是在 MyBatis 的 Mapper 配置文件中，properties 元素主体中指定的属性。

Settings 元素

Setting 元素下是些非常重要的设置选项，用于设置和改变 MyBatis 运行中的行为。下面的表格列出了 Setting 元素支持的属性、默认值及其功能。

设置选项	描述	可用值	默认值
cacheEnabled	全局性地启用或禁用所有在 mapper 配置文件中配置的缓存。	true | false	true
lazyLoadingEnabled	全局性地启用或禁用延迟加载。当禁用时，所有关联的配置都会立即加载。	true | false	true
aggressiveLazyLoading	当启用后，一个有延迟加载属性的对象的任何一个延迟属性被加载时，该对象的所有的属性都会被加载。否则，所有属性都是按需加载。	true | false	true
multipleResultSetsEnabled	允许或禁止从单一的语句返回多个结果集（需要驱动程序兼容）。	true | false	true
useColumnLabel	使用列的标签而不是列的名称。在这方面，不同的驱动程序可能有不同的实现。参考驱动程序的文档或者进行测试来确定您所使用的驱动程的行为	true | false	true
useGeneratedKeys	允许 JDBC 自动生成主键。需要驱动程序兼容。如果设置为 true 则会强行自动生成主键，然而有些则不会自动生成主键（驱动程序不兼容），但依旧会工作（如 Derby）。	true | false	False
autoMappingBehavior	指定 MyBatis 是否以及如何自动将列映射到字段/属性。 PARTIAL：只是自动映射简单、非嵌套的结果集。 FULL：将会自动映射任何复杂的（嵌套或非嵌套）的结果集。	NONE, PARTIAL, FULL	PARTIAL
defaultExecutorType	配置默认的执行器（executor）。 SIMPLE ： 简单的执行器。 REUSE ： 重用 prepared statements 的执行器。 BATCH：重用 statements 并且进行批量更新的执行器。	SIMPLE REUSE BATCH	SIMPLE
defaultStatementTimeout	设置查询数据库超时时间。	任何正整数	Not Set (null)

一个 Settings 元素完整的配置例子如下：

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

typeAliases 元素

别名是一个较短的 Java 类型的名称。这只是与 XML 配置文件相关联，减少输入多余的完整类名。例如：

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

在这个配置中，您就可以在想要使用"domain.blog.Blog"的地方使用别名“Blog”了。

对常用的 java 类型，已经内置了一些别名支持。这些别名都是不区分大小写的。注意 java 的基本数据类型，它们进行了特别处理，加了“_”前缀。

别名	对应的 java 类型
_byte	byte
_long	long
_short	short
_int	int
_integer	int
_double	double
_float	float
_boolean	boolean
string	String
Byte	Byte
Long	Long
short	Short
Int	Integer
integer	Integer
double	Double
Float	Float
boolean	Boolean
Date	Date
decimal	BigDecimal

bigdecimal	BigDecimal
object	Object
Map	Map
hashmap	HashMap
List	List
arraylist	ArrayList
collection	Collection
iterator	Iterator

typeHandlers 元素

每当 MyBatis 设置参数到 PreparedStatement 或者从 ResultSet 结果集中取得值时，就会使用 TypeHandler 来处理数据库类型与 java 类型之间转换。下表描述了默认的 TypeHandlers。

Type Handler	Java Types	JDBC Types
BooleanTypeHandler	Boolean, boolean	任何兼容的 BOOLEAN
ByteTypeHandler	Byte, byte	任何兼容的 NUMERIC 或 BYTE
ShortTypeHandler	Short, short	任何兼容的 NUMERIC 或 SHORT INTEGER
IntegerTypeHandler	Integer, int	任何兼容的 NUMERIC 或 INTEGER
LongTypeHandler	Long, long	任何兼容的 NUMERIC 或 LONG INTEGER
FloatTypeHandler	Float, float	任何兼容的 NUMERIC 或 FLOAT
DoubleTypeHandler	Double, double	任何兼容的 NUMERIC 或 DOUBLE
BigDecimalTypeHandler	BigDecimal	任何兼容的 NUMERIC 或 DECIMAL
StringTypeHandler	String	CHAR, VARCHAR
ClobTypeHandler	String	CLOB, LONGVARCHAR
NStringTypeHandler	String	NVARCHAR, NCHAR
NClobTypeHandler	String	NCLOB
ByteArrayTypeHandler	byte[]	任何兼容的 byte stream type
BlobTypeHandler	byte[]	BLOB, LONGVARBINARY
DateTypeHandler	Date (java.util)	TIMESTAMP
DateOnlyTypeHandler	Date (java.util)	DATE
TimeOnlyTypeHandler	Date (java.util)	TIME
SqlTimestampTypeHandler	Timestamp (java.sql)	TIMESTAMP
SqlDateTypeHadler	Date (java.sql)	DATE
SqlTimeTypeHandler	Time (java.sql)	TIME
ObjectTypeHandler	Any	OTHER, or unspecified
EnumTypeHandler	Enumeration Type	VARCHAR - 任何与 string 兼容的类型，储存的是编码而不是索引。

您能够重写类型处理器（type handlers），或者创建您自己的类型处理器去处理没有被支持的或非标准的类型。要做到这一点，只要实现 TypeHandler 接口(org.mybatis.type)，并且将您的 TypeHandler 类映射到 java 类型和可选的 JDBC 类型即可。例如：

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

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

使用上面的 TypeHandler 将会重写已经存在的用来处理 java 的 String 属性、VARCHAR 参数和结果集的类型处理器。注意，MyBatis 并不会通过数据库的元数据来确认类型，所以您必须指定它的一个类型处理器，用于将 VARCHAR 字段的参数和结果映射到正确的类型上。这是因为 MyBatis 在语句的执行之前都不知道它要处理的数据类型是什么。

objectFactory 元素

MyBatis 每次创建一个结果对象实例都会使用 ObjectFactory 实例。使用默认的 ObjectFactory 与使用默认的构造函数（或含参数的构造函数）来实例化目标类没什么差别。如果您想重写 ObjectFactory 来改变其默认行为，那您能通过创造您自己的 ObjectFactory 来做到。看下面的例子：

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

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

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

ObjectFactory 接口非常简单，它包含两个 create 的方法，一个是默认构造器，还有一个是含参数的构造器。最后的 setProperties 方法用来配置 ObjectFactory。在初始化您自己的 ObjectFactory 实例之后，定义在 objectFactory 元素主体中的属性会以参数的形式传递给 setProperties 方法。

Plugins 元素

MyBatis 允许您在映射语句执行的某些点拦截方法调用。默认情况下，MyBatis 允许插件（plug-ins）拦截下面的方法：

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

通过寻找完整的方法特征能够发现这些类方法的细节，以及在每个 MyBatis 发布版本中的源代码中找到。假如您要做的不仅仅是监视方法的调用情况，您就应该清楚您将重写的方法的行为动作。如果您试图修改或者重写给定的方法，您很可能会改变 MyBatis 的核心行为。这些都是比较底层的类和方法，所以要小心使用插件。

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

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

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

上面定义的插件将会拦截所有对 Executor 实例 update 方法的调用。Executor 实例是一个负责底层映射语句执行的内部对象。

重写配置类（Configuration Class）

除了能用插件方式修改 MyBatis 的核心行为，您也可以完全重写配置类（Configuration Class）。简单地扩展它和重写内部的任何方法，然后作为参数传递给 sqlSessionFactoryBuilder.build(myConfig)方法。再次提醒，这可能对 MyBatis 行为产生严重影响，因此要小心。

Environments 元素

MyBatis 能够配置多套运行环境，这有助于将您的 SQL 映射到多个数据库上。例如，在您的开发、测试、生产环境中，您可能有不同的配置。或者您可能有多个共享同一 schema 的生产用数据库，或者您想将相同的 SQL 映射应用到两个数据库等等许多用例。

但是请记住：虽然您可以配置多个运行环境，但是每个 SqlSessionFactory 实例只能选择一个运行环境。

因此，如果您想连接两个数据库，就需要创建两个 SqlSessionFactory 实例，一个数据库对应一个 SqlSessionFactory 实例。如果是三个数据库，那就创建三个实例，如此类推。这真的非常容易记住：

⇨ 每个数据库对应一个 SqlSessionFactory 实例。

要指定哪个运行环境被创建，只需要简单地将运行环境作为可选参数传递给 SqlSessionFactoryBuilder，下面是两个接受运行环境的方法：

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

如果环境参数被忽略，那默认的环境配置将被加载，如下面：

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

environments 元素定义了运行环境是怎么配置的：

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

注意这里关键的部分：

- 默认的运行环境 ID，引用一个已经定义好的运行环境 ID（例如：default=“development”）
- 每个定义的运行环境 ID（例如：id=“development”）
- 事务管理器配置(例如： type=“JDBC”)
- 数据源配置(例如：type=“POOLED”)

默认的环境和环境 ID 是自解（self explanatory）的，只要您喜欢，就可以随意取一个名字，只要确保默认的运行环境引用一个已定义的运行环境就可以了。

译者注：

```
<environments default="test">
    <environment id="development">
      ……
    </environment>
     <environment id="production">
```

```
      ……
    </environment>
     <environment id="test">
      ……
    </environment>

  </environments>
```

上面例子中，<environments default="test">配置表明，目前使用的是 test 的运行环境。当然，您也可以修改为使用 production 的运行环境：<environments default="production">。

事务管理器

MyBatis 有两种事务管理类型（即 type="[JDBC|MANAGED]"）：

- JDBC - 这个配置直接使用 JDBC 的提交和回滚功能。它依赖于从数据源获得连接来管理事务的生命周期。
- MANAGED - 这个配置基本上什么都不做。它从不提交或者回滚一个连接的事务。而是让容器（例如：Spring 或者 J2EE 应用服务器）来管理事务的生命周期 。默认情况下，它会关闭连接，但是一些容器并不会如此，因此，如果您需要通过关闭连接来停止事务，将属性 closeConnection 设置为 false。例如：

```
<transactionManager type="MANAGED">
      <property name="closeConnection" value="false"/>
</transactionManager>
```

这两个事务管理类型都不需要任何属性。然而它们都是类型别名，换句话说，您可以设置成指向已实现了 TransactionFactory 接口的完整类名或者别名。

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

实例化后，任何在 XML 文件配置的属性都将传递给 setProperties()方法。在您的实现中还需要创建一个非常简单的 Transaction 接口的实现：

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

通过这两个接口，您能够完全自定义 MyBatis 如何来处理事务。

dataSource 元素

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

⇨ 大部分 MyBatis 应用都像上面例子那样配置一个数据源，但这不是必须的。需要认清的是，只有使用了延迟加载才需要数据源。

MyBatis 内置了三种数据源类型（如： type=”????”）:

UNPOOLED - 这个类型的数据源实现只是在每次需要的时候简单地打开和关闭连接。虽然有点慢，但是对于不需要立即响应的简单的应用来说，不失为一种好的选择。不同的数据库在性能方面也会有所不同，因此对于一些数据库，不使用连接池时，这个配置就是比较理想的。
UNPOOLED 数据源有四个配置属性：

- driver - 指定 JDBC 驱动器。
- url - 连接数据库实例的 JDBC URL。
- username -登陆数据库的用户名。
- password - 登陆数据库的密码。
- defaultTransactionIsolationLevel - 指定连接的默认事务隔离级别。

另外，您也可以通过在属性前加前缀“driver”的方式，把属性传递给数据库驱动器，例如：

- driver.encoding=UTF8

这将会通过 DriverManager.getConnection(url, driverProperties) 方法，将值是“UTF8”的属性“encoding”传递给数据库驱动器。

POOLED - 这个数据源的实现缓存了 JDBC 连接对象，用于避免每次创建新的数据库连接时都初始化和进行认证，加快程序响应。并发 WEB 应用通常通过这种做法来获得快速响应。

另外，除了上面(UNPOOLED)的属性外，对 POOLED 数据源，还有很多属性可以设置。

- poolMaximumActiveConnections - 在任何特定的时间内激活（能够被使用）的连接数量，默认是 10。
- poolMaximumIdleConnections -在任何特定的时间内空闲的连接数。

- **poolMaximumCheckoutTime** - 在连接池被强行返回前，一个连接能够“检出”的总时间。默认是 20000ms（20 秒）。
- **poolTimeToWait** - 这是一上比较底层的设置，如果连接占用了很长时间，能够给连接池一个机会去打印日志，并重新尝试连接。默认是 20000ms（20 秒）。
- **poolPingQuery** -Ping Query 是发送给数据库的 Ping 信息，测试数据库连接是否良好和是否准备好了接受请求。默认值是“NO PING QUERY SET”，让大部分数据库都不使用 Ping，返回一个友好的错误信息（译者注：MyBatis 通过向数据执行 SQL 语句来确定与数据库连接状况）。
- **poolPingEnabled** - 这是允许或者禁 ping query 的开关。如果允许，您同时也要用一条可用的（并且应该是最高效的）SQL 语句来设置 poolPingQuery 属性的值。默认是：false（即禁止）。
- **poolPingConnectionsNotUsedFor** - 这个属性配置执行 poolPingQuery 的间隔时间。通常设置为与数据库连接的超时时间，来避免不必要的 pings 。默认是：0（允许所有连接随时进行 ping 测试，当然只有 poolPingEnabled 设置为 true 才会生效）。

JNDI - 这个数据源的配置是为了准备与像 Spring 或应用服务器能够在外部或者内部配置数据源的容器一起使用，然后在 JNDI 上下文中引用它。这个数据源只需配置两个属性：

- **initial_context** - 这个属性被用来从 InitialContext 中查找一个上下文。如：

  ```
  initialContext.lookup(initial_context)
  ```

  这个属性是可选的，如果忽略，那么数据源就会直接从 InitialContext 中查找。

- **data_source** - 这个属性是引用一个能够被找到的数据源实例的上下文路径。它会查找根据 initial_context 从 initialContext 中搜寻返回的上下文。或者在 initial_context 没有提供的情况下直接在 InitialContext 中进行查找。

```
译者注：

Context context = (Context) initialContext.lookup(initial_context);//返回一个上下文

//Context context = (Context)initContext.lookup("java:/comp/env");

DataSource ds = (DataSource) context.lookup(data_source); //返回一个数据源

Connection conn = ds.getConnection();

//DataSource ds = (DataSource)context.lookup("jdbc/myoracle");
```

如果 initial_context 没有配置，那么数据源就会直接从 InitialContext 进行查找，如：

```
DataSource ds = (DataSource) initialContext.lookup(data_source);
```

跟数据源的其它属性配置一样，可以通过在属性名前加“env.”的方式直接传递给 InitialContext。例如：

- env.encoding=UTF8

这将会把属性“encoding”及它的值“UTF8”在 InitialContext 实例化的时候传递给它的构造器。

Mappers 元素

现在，MyBatis 的行为属性都已经在上面的配置元素中配置好了，接下来开始定义映射 SQL 语句。但首先，我们需要告诉 MyBatis 在哪里能够找到我们定义的映射 SQL 语句。在这方面，JAVA 自动发现没有提供好的方法，因此最好的方法是告诉 MyBatis 在哪里能够找到这些映射文件。您可以使用类资源路径或者 URL（包括 file:/// URLs），例如：

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

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

这些配置告诉 MyBatis 在哪里找到 SQL 映射文件。而其它的更详细的信息配置在每一个 SQL 映射文件里。这些将在下一章节里讨论。

SQL 映射 XML 文件

MyBatis 真正强大之处就在这些映射语句，也就是它的魔力所在。对于它的强大功能，SQL 映射文件的配置却非常简单。如果您比较 SQL 映射文件配置与 JDBC 代码，您很快可以发现，使用 SQL 映射文件配置可以节省 95%的代码量。MyBatis 被创建来专注于 SQL，但又给您自己的实现极大的空间。

SQL 映射 XML 文件只有一些基本的元素需要配置，并且要按照下面的顺序来定义(in the order that they should be defined)：

- cache – 在特定的命名空间配置缓存。
- cache-ref – 引用另外一个命名空间配置的缓存.
- resultMap – 最复杂也是最强大的元素，用来描述如何从数据库结果集里加载对象。
- ~~parameterMap – 不推荐使用！在旧的版本里使用的映射配置，这个元素在将来可能会被删除，因此不再进行描述。~~
- sql – 能够被其它语句重用的 SQL 块。
- insert – INSERT 映射语句
- update – UPDATE 映射语句
- delete – DELEETE 映射语句
- select – SELECT 映射语句

接下来的章节将会对每一个元素进行描述。

Select 元素

Select 是 MyBatis 中最常用的元素之一。除非您把数据取回来，否则把数据放在数据库是没什么意义的，因此大部分的应用查询数据远多于修改数据。对每一次插入、修改、删除数据都可能伴有大量的查询。这是 MyBatis 基本设计原则之一，也就是为什么把这么多的焦点和努力放在查询和结果集映射上。对简单的查询，select 元素的配置是相当简单的，如：

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

这条语句叫做 selectPerson，以 int 型（或者 Integer 型）作为参数，并返回一个以数据库列名作为键值的 HashMap。

注意这个参数的表示方法：#{id}
它告诉 MyBatis 生成 PreparedStatement 参数。对于 JDBC，像这个参数会被标识为“?”，然后传递给 PreparedStatement，像这样：

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

当然，如果单独使用JDBC 去提取这个结果集并把结果集映射到对象上的话，则需要更多的代

码，而这些，MyBatis 都已经为您做到了。关于参数和结果集映射，还有许多要学的，以后有专门的章节来讨论。

select 语句有很多的属性允许您详细配置每一条语句。

```
<select
        id=" selectPerson"
        parameterType=" int"
        parameterMap=" deprecated"
        resultType=" hashmap"
        resultMap=" personResultMap"
        flushCache=" false"
        useCache=" true"
        timeout=" 10000"
        fetchSize=" 256"
        statementType=" PREPARED"
        resultSetType=" FORWARD_ONLY"
        >
```

Attribute	Description
id	在这个命名空间下唯一的标识符，可被其它语句引用
parameterType	传给此语句的参数的完整类名或别名
~~parameterMap~~	不推荐使用。这个参数将来可能被删除。
resultType	语句返回值类型的完整类名或别名。注意，如果返回的是集合（collections），那么应该是集合所包含的具体子类型，而不是集合本身。resultType 与 resultMap 不能同时使用
resultMap	引用的外部定义的 resultMap。结果集映射是 MyBatis 中最强大的特性，同时又非常好理解。许多复杂的映射都可以轻松解决。resultType 与 resultMap 不能同时使用
flushCache	如果设为 true，则会在每次语句调用的时候就会清空缓存。select 语句默认设为 false
useCache	如果设为 true，则语句的结果集将被缓存，select 语句默认设为 false
timeout	设置超时时间，默认没有设置，由驱动器自己决定
fetchSize	设置从数据库获得记录的条数，默认没有设置，由驱动器自己决定
statementType	可设置为STATEMENT，PREPARED 或CALLABLE 中的任意一个，告诉MyBatis 分别使用Statement，PreparedStatement或者CallableStatement。默认：PREPARED
resultSetType	FORWARD_ONLY 、SCROLL_SENSITIVE 、SCROLL_INSENSITIVE 三个中的任意一个。默认没有设置，由驱动器自己决定

Insert、 update、 delete 元素

数据修改语句 insert、update 和 delete 的配置使用都非常相似：

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

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

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

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

Attribute	Description
id	在这个命名空间下唯一的标识符，可被其它语句引用。
parameterType	传给此语句的参数的完整类名或别名。
~~parameterMap~~	不推荐使用，将来可能删除。
flushCache	如果设为 true，则会在每次语句调用的时候就会清空缓存。select 语句默认设为 false
timeout	设置超时时间，默认没有设置，由驱动器自己决定。
statementType	可设置为 STATEMENT，PREPARED 或 CALLABLE 中的任意一个告诉 MyBatis 分别使用 Statement，PreparedStatement 或者 CallableStatement。默认：PREPARED
useGeneratedKeys	(仅限 insert 语句时使用)告诉 MyBatis 使用 JDBC 的 getGeneratedKeys 方法来获取数据库自动生成主键（如：MySQL、SQLSERVER 等关系型数据库会有自增的字段）。默认：false
keyProperty	(仅限 insert 语句时使用)设置自动生成主键的字段，这个字段的值由 getGeneratedKeys 方法返回，或者由 insert 元素的 selectKey 子元素返回。默认不设置。

下面是一些 insert、update 和 delete 语句例子。

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

<update id="updateAuthor" parameterType="domain.blog.Author">
  update Author set
        username = #{username},
        password = #{password},
```

```
        email = #{email},
        bio = #{bio}
    where id = #{id}
</update>

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

正如您看到，insert 元素有更多一些的扩展属性及子元素，允许您使用多种方式自动生成主键。

首先，如果您使用的数据库支持自动生成主键（如：MySQL 和 SQL Server），那么您就可以简单地将 useGeneratedKeys 设置为”true”，然后使用 keyProperty 设置您希望自动生成主键的字段就可以了。例如，如果上面提到的 Author 表使用一个字段自动生成主键，那么配置语句就可以修改为：

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

MyBatis 还有另外一种方式为不支持自动生成主键的数据库及 JDBC 驱动来生成键值。

下面展示一个能够随机生成 ID 的例子（也许您不会这么做，这仅仅是演示 MyBatis 的功能）：

```
<insert id="insertAuthor" parameterType="domain.blog.Author">
  <selectKey keyProperty="id" resultType="int" order="BEFORE">
    select CAST(RANDOM()*1000000 as INTEGER) a from SYSIBM.SYSDUMMY1
  </selectKey>
  insert into Author
        (id, username, password, email,bio, favourite_section)
  values
        (#{id}, #{username}, #{password}, #{email}, #{bio},
        #{favouriteSection,jdbcType=VARCHAR}
  )
</insert>
```

在上面的例子中，selectKey 语句首先会执行，Author 表的 ID 首先会被设值，然后才会调用 insert 语句。这相当于在您的数据库中自动生成键值，不需要编写复杂的 java 代码。

selectKey 元素描述如下：

```
<selectKey
        keyProperty="id"
        resultType="int"
```

```
        order="BEFORE"
        statementType="PREPARED">
```

Attribute	Description
keyProperty	设置需要自动生成键值的列。
resultType	结果类型，MyBatis 通常可以自己检测到，但这并不影响给它一个确切的类型。MyBatis 允许使用任何基本的数据类型作为键值，也包括 String 类型。
order	可以设成 BEFORE 或者 AFTER，如果设为 BEFORE，那它会先选择主键，然后设置 keyProperty，再执行 insert 语句；如果设为 AFTER，它就先执行 insert 语句再执行 selectKey 语句，像数据库如 Oracle 那样在 insert 语句中调用内嵌的序列机制一样。
statementType	像前面一样，MyBatis 支持 STATEMENT、PREPARED 和 CALLABLE 语句类型，分别对应 Statement, PreparedStatement 和 CallableStatement 。

Sql 元素

这个元素用来定义能够被其它语句引用的可重用 SQL 语句块。例如：

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

这个 SQL 语句块能够被其它语句引用，如：

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

参数（Parameters）

在前面的语句中，我们可以看到一些例子中简单的参数（用于代入 SQL 语句中的可替换变量，如#{id}）。参数是 MyBatis 中非常强大的配置属性，基本上，90%的情况都会用到，如：

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

上面的例子演示了一个非常简单的命名参数映射，parameterType 被设置为“int”。参数可以设置为任何类型。像基本数据类型或者像 Integer 和 String 这样的简单的数据对象，（因为没有相关属性）将使用全部参数值。而如果传递的是复杂对象（一般是指 JavaBean），那情况就有所不同。例如：：

```
<insert id=”insertUser” parameterType=”User” >
        insert into users (id, username, password)
```

```
        values (#{id}, #{username}, #{password})
    </insert>
```

如果参数对象 User 被传递给 SQL 语句，那么将会搜寻 PreparedStatement 里的 id,username 和 password 属性，并被 User 对象里相应的属性值替换。

这种传递参数到语句的方式非常优雅和简单。但参数映射还在很多特性。

首先，像 MyBatis 其它部分一样，参数可以指定许多的数据类型。

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

像 MyBatis 的其它部分一样，这个 javaType 是由参数对象决定，除了 HashMap 以外。然后这个 javaType 应该确保指定正确的 TypeHandler 被使用。

➔ Note:如果传递了一个空值，那这个JDBC Type 对于所有JDBC 允许为空的列来说是必须的。您可以研读一下关于PreparedStatement.setNull()的JavaDocs 文档。

对于需要自定义类型处理，您也可以指定一个特殊的 TypeHandler 类或者别名，如：

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

当然，这看起来更加复杂了，不过，这种情况比较少见。

对于数据类型，可以使用 numericScale 来指定小数位的长度。

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

最后，mode属性允许您指定IN、 OUT 或 INOUT 参数。如果参数是OUT 或 INOUT，参数对象属的实际值将会改变，正如您希望调用一个输出参数。如果mode=OUT (或者 INOUT)，并且 jdbcType=CURSOR (如Oracle 的REFCURSOR)，您必须指定一个resultMap映射结果集给这个参数类型。注意这里的javaType 类型是可选的，如果为空值而jdbcType =CURSOR 的话，则会自动地设给ResultSet。

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

MyBatis 也支持像 structs 这样高级的数据类型，但当您把 mode 设置为 out 的时候，您必须把类型名告诉执行语句。例如：

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

尽管有这些强大的选项，但是大多数情况下您只需指定属性名，MyBatis 将会识别其它部分的设置。最多就是给可以为 null 值的列指定 jdbcType：

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

字符串替换

默认的情况下，使用#{}语法会促使MyBatis 生成PreparedStatement并且安全地设置PreparedStatement 参数（=？）值。尽量这是安全、快捷并且是经常使用的，但有时候您可能想直接未更改的字符串代入到SQL 语句中。比如说，对于ORDER BY，您可以这样使用

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

这样 MyBatis 就不会修改这个字符串了。

➔ **警告：** 这种不加修改地接收用户输入并应用到语句的方式，是非常不安全的。这使用户能够进行SQL注入，破坏代码。所以，要么这些字段不允许用户输入，要么用户每次输入后都进行检测和规避。

resultMap 元素

resultMap元素是MyBatis中最重要最强大的元素。与使用JDBC从结果集获取数据相比，它可以省掉90%的代码，也可以允许您做一些JDBC不支持的事。事实上，要写一个类似于连结映射（join mapping）这样复杂的交互代码，可能需要上千行的代码。设计ResultMaps 的目的，就是只使用简单的配置语句而不需要详细地处理结果集映射，对更复杂的语句除了使用一些必须的语句描述以外，就不需要其它的处理了。

您可能已经看到过这样简单映射语句，它并没有使用 resultMap，例如：

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

像上面的语句，所有结果集将会自动地映射到以列表为key 的HasMap（由resultType指定）中。虽然这对许多场合下有用，但是HashMap 却不是非常好的域模型。更多的情况是使用JavaBeans或者POJOs作为域模型。MyBatis支持这两种域模型。考虑下面的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 JavaBeans 规范，上面的类有 3 个属性： id、username,和 hashedPassword 。这 3 个属性对应 select 语句的列名。

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

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

别忘了别名是您的朋友，使用别名您不用输入那长长的类路径。例如：

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

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

这种情况下，MyBatis 在后台自动生成 ResultMap，将列名映射到 JavaBean 的相应属性。如果列名与属性名不匹配，可以使用 select 语法（标准的 SQL 特性）中的将列名取一个别名的方式来进行匹配。例如

```
<select id="selectUsers" parameterType="int" resultType="User">
```

```
        select
               user_id             as “id”,
               user_name           as “userName”,
               hashed_password     as “hashedPassword”
        from some_table
        where id = #{id}
    </sql>
```

ResultMaps 的知识您可能已经学到了许多，但还有一个您从没见到过。为了举例，让我们看看最后一个例子，作为另一种解决列名不匹配的方法。

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

这个语句将会被 resultMap 属性引用（注意，我们没有使用 resultType）。如：

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

一切就是这么简单！

高级结果映射

MyBatis的创建基于这样一个思想：数据库并不是您想怎样就怎样的。虽然我们希望所有的数据库遵守第三范式或BCNF（修正的第三范式），但它们不是。如果有一个数据库能够完美映射到所有应用数据模型，也将是非常棒的，但也没有。结果集映射就是MyBatis为解决这些问题而提供的解决方案。例如，我们如何映射下面这条语句？

```
<!-- Very Complex Statement -->
<select id="selectBlogDetails" parameterType="int" resultMap="detailedBlogResultMap">
    select
            B.id as blog_id,
            B.title as blog_title,
            B.author_id as blog_author_id,
            A.id as author_id,
            A.username as author_username,
            A.password as author_password,
            A.email as author_email,
            A.bio as author_bio,
            A.favourite_section as author_favourite_section,
            P.id as post_id,
            P.blog_id as post_blog_id,
            P.author_id as post_author_id,
            P.created_on as post_created_on,
            P.section as post_section,
            P.subject as post_subject,
```

```
        P.draft as draft,
        P.body as post_body,
        C.id as comment_id,
        C.post_id as comment_post_id,
        C.name as comment_name,
        C.comment as comment_text,
        T.id as tag_id,
        T.name as tag_name
    from Blog B
        left outer join Author A on B.author_id = A.id
        left outer join Post P on B.id = P.blog_id
        left outer join Comment C on P.id = C.post_id
        left outer join Post_Tag PT on PT.post_id = P.id
        left outer join Tag T on PT.tag_id = T.id
    where B.id = #{id}
  </select>
```

您可能想要把它映射到一个智能的对象模型，包括由一个作者写的一个博客，有许多文章（Post，帖子），每个文章由0个或者多个评论和标签。下面是一个复杂ResultMap 的完整例子（假定作者、博客、文章、评论和标签都是别名）。仔细看看这个例子，但是不用太担心，我们会一步步地来分析，一眼看上去可能让人沮丧，但是实际上非常简单的

```
<!-- Very Complex Result Map -->
<resultMap id="detailedBlogResultMap" type="Blog">
  <constructor>
    <idArg column="blog_id" javaType="int"/>
  </constructor>
  <result property="title" column="blog_title"/>
  <association property="author" column="blog_author_id" javaType=" Author">
    <id property="id" column="author_id"/>
    <result property="username" column="author_username"/>
    <result property="password" column="author_password"/>
    <result property="email" column="author_email"/>
    <result property="bio" column="author_bio"/>
    <result property="favouriteSection" column="author_favourite_section"/>
  </association>
  <collection property="posts" ofType="Post">
    <id property="id" column="post_id"/>
    <result property="subject" column="post_subject"/>
    <association property="author" column="post_author_id" javaType="Author"/>
    <collection property="comments" column="post_id" ofType=" Comment">
      <id property="id" column="comment_id"/>
    </collection>
    <collection property="tags" column="post_id" ofType=" Tag" >
      <id property="id" column="tag_id"/>
    </collection>
    <discriminator javaType="int" column="draft">
      <case value="1" resultType="DraftPost"/>
    </discriminator>
  </collection>
</resultMap>
```

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

个resultMap的元素。

resultMap
- constructor - 实例化的时候通过构造器将结果集注入到类中
  - idArg - ID 参数; 将结果集标记为 ID，以方便全局调用
  - arg - 注入构造器的结果集
- id -结果集 ID，将结果集标记为 ID，以方便全局调用
- result - 注入一个字段或者 javabean 属性的结果
- association - 复杂类型联合; 许多查询结果合成这个类型
  - *嵌套结果映射* - associations 能引用自身，或者从其它地方引用
- collection - 复杂类型集合
  - *嵌套结果映射*- collections 能引用自身，或者从其它地方引用
- discriminator -使用一个结果值以决定使用哪个 resultMap
  - case - 基于不同值的结果映射
    - *嵌套结果映射* -case 也能引用它自身，所以也能包含这些同样的元素。它也可以从外部引用 resultMap

➔ **最佳实践:** 逐步地生成 resultMap，单元测试对此非常有帮助。如果您尝试一下子就生成像上面这样巨大的 resultMap，可能会出错，并且工作起来非常吃力。从简单地开始，再一步步地扩展，并且进行单元测试。使用框架开发有一个缺点，它们有时像是一个黑盒子。为了确保达到您所预想的行为，最好的方式就是进行单元测试。这对提交 bugs 也非常有用。

下一节，我们一步步地查看这些细节。

id, result 元素

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

这是最基本的结果集映射。*id* 和 *result* 将列映射到属性或简单的数据类型字段 (String, int, double, Date 等)。

这两者唯一不同的是，在比较对象实例时 *id* 作为结果集的标识属性。这有助于提高总体性能，特别是应用缓存和嵌套结果映射的时候。

id、result 属性如下:

Attribute	Description
property	映射数据库列的字段或属性。如果JavaBean 的属性与给定的名称匹配，就会使用匹配的名字。否则，MyBatis 将搜索给定名称的字段。两种情况下您都可以使用逗点的属性形式。比如，您可以映射到“username”，也可以映射到“address.street.number”。
column	数据库的列名或者列标签别名。与传递给 resultSet.getString(columnName)的参数名称相同。
javaType	完整 java 类名或别名 (参考上面的内置别名列表)。如果映射到一个 JavaBean，那 MyBatis 通常会自行检测到。然而，如果映射到一个 HashMap，那您应该明确

	指定 javaType 来确保所需行为。
jdbcType	这张表下面支持的 JDBC 类型列表列出的 JDBC 类型。这个属性只在 insert，update 或 delete 的时候针对允许空的列有用。JDBC 需要这项，但 MyBatis 不需要。如果您直接编写 JDBC 代码，在允许为空值的情况下需要指定这个类型。
typeHandler	我们已经在文档中讨论过默认类型处理器。使用这个属性可以重写默认类型处理器。它的值可以是一个 TypeHandler 实现的完整类名，也可以是一个类型别名。

支持的 JDBC 类型

MyBatis 支持如下的 JDBC 类型：

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

Constructor 元素

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

当属性与DTO，或者与您自己的域模型一起工作的时候，许多场合要用到不变类。通常，包含引用，或者查找的数据很少或者数据不会改变的的表，适合映射到不变类中。构造器注入允许您在类实例化后给类设值，这不需要通过public方法。MyBatis 同样也支持private 属性和 JavaBeans 的私有属性达到这一点，但是一些用户可能更喜欢使用构造器注入。构造器元素可以做到这点。

考虑下面的构造器：

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

为了将结果注入构造器，MyBatis 需要使用它的参数类型来标记构造器。Java 没有办法通过参数名称来反射获得。因此当创建 constructor 元素，确保参数是按顺序的并且指定了正确的类型。

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

其它的属性与规则与 id、result 元素的一样。

Attribute	Description
column	数据库的列名或者列标签别名。与传递给 resultSet.getString(columnName)的参数名称相同。
javaType	完整 java 类名或别名 (参考上面的内置别名列表)。如果映射到一个 JavaBean，那 MyBatis 通常会自行检测到。然而，如果映射到一个 HashMap，那您应该明确指定 javaType 来确保所需行为。
jdbcType	支持的 JDBC 类型列表中列出的 JDBC 类型。这个属性只在 insert,update 或 delete 的时候针对允许空的列有用。JDBC 需要这项，但 MyBatis 不需要。如果您直接编写 JDBC 代码，在允许为空值的情况下需要指定这个类型。
typeHandler	我们已经在文档中讨论过默认类型处理器。使用这个属性可以重写默认类型处理器。它的值可以是一个 TypeHandler 实现的完整类名，也可以是一个类型别名。

Association 元素

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

Association 元素处理“has-one”（一对一）这种类型关系。比如在我们的例子中，一个 Blog 有一个 Author。联合映射与其它的结果集映射工作方式差不多，指定 property、column、javaType（通常 MyBatis 会自动识别）、jdbcType（如果需要）、typeHandler。

不同的地方是您需要告诉 MyBatis 如何加载一个联合查询。MyBatis 使用两种方式来加载：

- **Nested Select:** 通过执行另一个返回预期复杂类型的映射 SQL 语句（即引用外部定义好的 SQL 语句块）。
- **Nested Results:** 通过嵌套结果映射（nested result mappings）来处理联接结果集（joined results）的重复子集。

首先，让我们检查一下元素属性。正如您看到的，它不同于普通只有 select 和 resultMap 属性的结果映射。

Attribute	Description
property	映射数据库列的字段或属性。如果 JavaBean 的属性与给定的名称匹配，就会使用匹配的名字。否则，MyBatis 将搜索给定名称的字段。两种情况下您都可以使用逗点的属性形式。比如，您可以映射到”username”，也可以映射到更复杂点的”address.street.number”。
column	数据库的列名或者列标签别名。与传递给 resultSet.getString(columnName)的参数名称相同。 注意： 在处理组合键时， 您可以使用 column= “{prop1=col1,prop2=col2}”这样的语法，设置多个列名传入到嵌套查询语句。这就会把 prop1 和 prop2 设置

	到目标嵌套选择语句的参数对象中。
javaType	完整 java 类名或别名 (参考上面的内置别名列表)。如果映射到一个 JavaBean，那 MyBatis 通常会自行检测到。然而，如果映射到一个 HashMap，那您应该明确指定 javaType 来确保所需行为。
jdbcType	支持的 JDBC 类型列表中列出的 JDBC 类型。这个属性只在 insert,update 或 delete 的时候针对允许空的列有用。JDBC 需要这项，但 MyBatis 不需要。如果您直接编写 JDBC 代码，在允许为空值的情况下需要指定这个类型。
typeHandler	我们已经在文档中讨论过默认类型处理器。使用这个属性可以重写默认类型处理器。它的值可以是一个 TypeHandler 实现的完整类名，也可以是一个类型别名。

联合嵌套选择（Nested Select for Association）

select	通过这个属性，通过 ID 引用另一个加载复杂类型的映射语句。从指定列属性中返回的值，将作为参数设置给目标 select 语句。表格下方将有一个例子。注意：在处理组合键时，您可以使用 column="{prop1=col1,prop2=col2}" 这样的语法，设置多个列名传入到嵌套语句。这就会把 prop1 和 prop2 设置到目标嵌套语句的参数对象中。

例如:

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

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

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

我们使用两个 select 语句：一个用来加载 Blog，另一个用来加载 Author。Blog 的 resultMap 描述了使用“selectAuthor”语句来加载 author 的属性。

如果列名和属性名称相匹配的话，所有匹配的属性都会自动加载。

> 译者注：
> 上面的例子，首先执行<select id=“selectBlog”>，执行结果存放到<resultMap id=“blogResult”>结果映射中。“blogResult”是一个 Blog 类型，从<select id=“selectBlog”>查出的数据都会自动赋值给”blogResult”的与列名匹配的属性，这时 blog_id，title 等就被赋值了。同时“blogResult”还有一个关联属性"Author"，执行嵌套查询 select=” selectAuthor”后，Author 对象的属性 id，username，password，email，bio 也被赋于与数据库匹配的值。
>
> Blog
> {

```
   blog_id;
   title;
   Author author
   {
        id;
        username;
        password;
        email;
        bio;

    }

}
```

建议不要使用 Batatis 的自动赋值，这样不能够清晰地知道要映射哪些属性，并且有时候还不能保证正确地映射数据库检索结果。

虽然这个方法简单，但是对于大数据集或列表查询，就不尽如人意了。这个问题被称为“N+1 选择问题”（N+1 Selects Problem）。概括地说，N+1选择问题是这样产生的：

- 您执行单条 SQL 语句去获取一个列表的记录（“+1”）。
- 对列表中的每一条记录，再执行一个联合 select 语句来加载每条记录更加详细的信息（“N”）。

译者注：
如：执行一条 SQL 语句获得了 10 条记录，这 10 条记录的每一条再执行一条 SQL 语句去加载更详细的信息，这就执行了 10+1 次查询。

这个问题会导致成千上万的 SQL 语句的执行，因此并非总是可取的。

上面的例子，MyBatis 可以使用延迟加载这些查询，因此这些查询立马可节省开销。然而，如果您加载一个列表后立即迭代访问嵌套的数据，这将会调用所有的延迟加载，因此性能会变得非常糟糕。

鉴于此，这有另外一种方式。
联合嵌套结果集（Nested Results for Association）

resultMap	一个可以映射联合嵌套结果集到一个适合的对象视图上的 ResultMap 。这是一个替代的方式去调用另一个 select 语句。它允许您去联合多个表到一个结果集里。这样的结果集可能包括冗余的、重复的需要分解和正确映射到一个嵌套对象视图的数据组。简言之，MyBatis 让您把结果映射‘链接’到一起，用来处理嵌套结果。举个例子会更好理解，例子在表格下方。

您已经在上面看到了一个非常复杂的嵌套联合的例子，接下的演示的例子会更简单一些。我们把 Blog 和 Author 表联接起来查询，而不是执行分开的查询语句：

```
<select id="selectBlog" parameterType="int" resultMap="blogResult">
    select
            B.id                as blog_id,
            B.title             as blog_title,
            B.author_id         as blog_author_id,
            A.id                as author_id,
            A.username  as author_username,
            A.password  as author_password,
            A.email             as author_email,
            A.bio               as author_bio
    from Blog B left outer join Author A on B.author_id = A.id
    where B.id = #{id}
  </select>
```

注意到这个连接（join），要确保所有的别名都是唯一且无歧义的。这使映射容易多了，现在我们来映射结果集：

```
<resultMap id="blogResult" type="Blog">
  <id property=" id"  column=" blog_id" />
  <result property="title" column="blog_title"/>
  <association property="author" column="blog_author_id" javaType="Author"
        resultMap=" authorResult" />
</resultMap>

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

在上面的例子中，您会看到Blog 的作者（“author”）联合一个“authorResult”结果映射来加载Author 实例。

重点提示： *id* 元素在嵌套结果映射中扮演了非常重要的角色，您应该总是指定一个或多个属性来唯一标识这个结果集。事实上，如果您没有那样做，MyBatis 也会工作，但是会导致严重性能开销。选择尽量少的属性来唯一标识结果，而使用主键是最明显的选择（即使是复合主键）。

上面的例子使用一个扩展的resultMap 元素来联合映射。这可使Author结果映射可重复使用。然后，如果您不需要重用它，您可以直接嵌套这个联合结果映射。下面例子就是使用这样的方式：

```
        <resultMap id="blogResult" type="Blog">
          <id property=" blog_id"  column="id" />
          <result property="title" column="blog_title"/>
          <association property="author" column="blog_author_id" javaType="Author">
                <id property="id" column="author_id"/>
                <result property="username" column="author_username"/>
                <result property="password" column="author_password"/>
                <result property="email" column="author_email"/>
```

```
        <result property="bio" column="author_bio"/>
      </association>
    </resultMap>
```

在上面的例子中您已经看到如果处理“一对一”（“ has one”）类型的联合查询。但是对于“一对多”（“has many”）的情况如果处理呢？这个问题在下一节讨论。

Collection 元素

```
<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>
```

collection 元素的作用差不多和 association 元素的作用一样。事实上，它们非常相似，以至于再对相似点进行描述会显得冗余，因此我们只关注它们的不同点。

继续我们上面的例子，一个 Blog 只有一个 Author。但一个 Blog 有许多帖子（文章）。在 Blog 类中，会像下面这样定义相应属性：

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

映射一个嵌套结果集到一个列表，我们使用 *collection* 元素。就像 *association* 元素那样，我们使用嵌套查询，或者从连接中嵌套结果集。

集合嵌套选择 (Nested Select for Collection)

首先我们使用嵌套选择来加载 Blog 的文章。

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

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

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

一看上去这有许多东西需要注意，但大部分看起与我们在*associa*tion元素中学过的相似。首先，您会注意到我们使用了collection元素，然后会注意到一个新的属性“ofType”。这个元素是用来区别JavaBean属性（或者字段）类型和集合所包括的类型。因此您会读到下面这段代码。

```
<collection property="posts" javaType=" ArrayList"  column="blog_id"
```

```
ofType="Post" select=" selectPostsForBlog" />
```

➔ 理解为： “一个名为 posts，类型为 Post 的 ArrayList 集合（ A collection of posts in an ArrayList of type Post）” 。

javaType 属性不是必须的，通常 MyBatis 会自动识别，所以您通常可以简略地写成：

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

集合的嵌套结果集（Nested Results for Collection）

这时候，您可能已经猜出嵌套结果集是怎样工作的了，因为它与 association 非常相似，只不过多了一个属性 *“ofType”* 。

让我们看下这个 SQL：

```
<select id="selectBlog" parameterType="int" resultMap="blogResult">
    select
            B.id as blog_id,
            B.title as blog_title,
            B.author_id as blog_author_id,
            P.id as post_id,
            P.subject as post_subject,
            P.body as post_body,
    from Blog B
            left outer join Post P on B.id = P.blog_id
    where B.id = #{id}
  </select>
```

同样，我们把 Blog 和 Post 两张表连接在一起，并且也保证列标签名在映射的时候是唯一且无歧义的。现在将 Blog 和 Post 的集合映射在一起是多么简单：

```
<resultMap id="blogResult" type="Blog">
  <id property=" id"  column="blog_id" />
  <result property="title" column="blog_title"/>
  <collection property="posts" ofType="Post">
    <id property="id" column="post_id"/>
    <result property="subject" column="post_subject"/>
    <result property="body" column="post_body"/>
  </collection>
</resultMap>
```

再次强调一下，*id* 元素是非常重要的。如果您忘了或者不知道 *id* 元素的作用，请先读一下上面 *association* 一节。

如果希望结果映射有更好的可重用性，您可以使用下面的方式：

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

```
  <result property="title" column="blog_title"/>
  <collection property="posts" ofType="Post" resultMap=" blogPostResult" />
</resultMap>

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

➔ Note: 在您的映射中没有深度、宽度、联合和集合数目的限制。但应该谨记，在进行映射的时候也要考虑性能的因素。应用程序的单元测试和性能测试帮助您发现最好的方式可能要花很长时间。但幸运的是，MyBatis 允许您以后可以修改您的想法，这时只需要修改少量代码就行了。

关于高级联合和集合映射是一个比较深入的课题，文档只能帮您了解到这里，多做一些实践，一切将很快变得容易理解。

Discriminator 元素

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

有时候一条数据库查询可能会返回包括各种不同的数据类型的结果集。*Discriminator（识别器）*元素被设计来处理这种情况，以及其它像类继承层次情况。识别器非常好理解，它就像 java 里的 switch 语句。

Discriminator定义要指定*column*和*javaType*属性。列是MyBatis 将要取出进行比较的值，javaType 用来确定适当的测试是否正确运行（即使是String 在大部分情况下也可以工作），例：

```
<resultMap id="vehicleResult" type="Vehicle">
        <id property=" id"  column="id" />
        <result property="vin" column="vin"/>
        <result property="year" column="year"/>
        <result property="make" column="make"/>
        <result property="model" column="model"/>
        <result property="color" column="color"/>
        <discriminator javaType="int" column="vehicle_type">
          <case value="1" resultMap="carResult"/>
          <case value="2" resultMap="truckResult"/>
          <case value="3" resultMap="vanResult"/>
          <case value="4" resultMap="suvResult"/>
        </discriminator>
</resultMap>
```

在这个例子中，MyBatis 将会从结果集中取出每条记录，然后比较它的 vehicle type 的值。如果匹配任何 discriminator 中的 case，它将使用由 case 指定的 resultMap。这是排它性的，换句话说，其它的 case 的 resultMap 将会被忽略（除非使用我们下面说到的 extended）。如果没

有匹配到任何 case，MyBatis 只是简单的使用定义在 discriminator 块外面的 resultMap。所以，如果 carResult 像下面这样定义：

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

那么，只有doorCount属性会被加载。这样做是为了与识别器cases群组完全独立开来，哪怕它与上一层的resultMap 一点关系都没有。在刚才的例子里我们当然知道cars和vehicles的关系，a Car is-a Vehicle。因此，我们也要把其它属性加载进来。我们要稍稍改动一下 resultMap：

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

现在，vehicleResult 和 carResult 的所有属性都会被加载。

可能有人会认为这样扩展映射定义有一点单调了，所以还有一种可选的更加简单明了的映射风格语法。例如：

```
<resultMap id="vehicleResult" type="Vehicle">
        <id property=" id"  column="id" />
        <result property="vin" column="vin"/>
        <result property="year" column="year"/>
        <result property="make" column="make"/>
        <result property="model" column="model"/>
        <result property="color" column="color"/>
        <discriminator javaType="int" column="vehicle_type">
          <case value="1" resultType="carResult">
            <result property=" doorCount"  column="door_count" />
          </case>
          <case value="2" resultType="truckResult">
            <result property=" boxSize"  column="box_size" />
            <result property=" extendedCab"  column="extended_cab" />
          </case>
          <case value="3" resultType="vanResult">
            <result property=" powerSlidingDoor"  column="power_sliding_door" />
          </case>
          <case value="4" resultType="suvResult">
            <result property=" allWheelDrive"  column="all_wheel_drive" />
          </case>
        </discriminator>
</resultMap>
```

➔ **记住：**对于这么多的结果映射，如果您不指定任何的结果集，那么MyBatis 会自动地将列名与属性相匹配。所以上面所举的例子比实际中需要的要详细。尽管如此，大部分数据库有点复杂，并且它并不是所有情况都是完全可以适用的。

Cache 元素

MyBatis 包含一个强大的、可配置、可定制的查询缓存机制。MyBatis 3 的缓存实现有了许

多改进，使它更强大更容易配置。默认的情况，缓存是没有开启的，除了会话缓存以外，会话缓存可以提高性能，且能解决循环依赖。开启二级缓存，您只需要在SQL映射文件中加入简单的一行：

```
<cache/>
```

这句简单的语句作用如下：

- 所有映射文件里的 select 语句的结果都会被缓存。
- 所有映射文件里的 **insert、 update** 和 **delete** 语句执行都会清空缓存 。
- 缓存使用最近最少使用算法(LRU)来回收。
- 缓存不会被设定的时间所清空。
- 每个缓存可以存储 1024 个列表或对象的引用（不管查询方法返回的是什么）。
- 缓存将作为“读/写”缓存，意味着检索的对象不是共享的且可以被调用者安全地修改，而不会被其它调用者或者线程干扰。

所有这些特性都可以通过 cache 元素进行修改。例如：

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

这种高级的配置创建一个每60秒刷新一次的FIFO 缓存，存储512个结果对象或列表的引用，并且返回的对象是只读的。因此在不用的线程里的调用者修改它们可能会引用冲突。

可用的回收算法如下：

- **LRU** – 最近最少使用：移出最近最长时间内都没有被使用的对象。
- **FIFO** –先进先出：移除最先进入缓存的对象。
- **SOFT** – 软引用: 基于垃圾回收机制和软引用规则来移除对象（空间内存不足时才进行回收）。
- **WEAK** – 弱引用: 基于垃圾回收机制和弱引用规则（垃圾回收器扫描到时即进行回收）。

默认使用 LRU。

flushInterval ：设置任何正整数，代表一个以毫秒为单位的合理时间。默认是没有设置，因此没有刷新间隔时间被使用，在语句每次调用时才进行刷新。

Size：属性可以设置为一个正整数，您需要留意您要缓存对象的大小和环境中可用的内存空间。默认是 1024。

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

使用自定义缓存

除了上面已经定义好的缓存方式，您能够通过您自己的缓存实现来完全重写缓存行为，或者通过创建第三方缓存解决方案的适配器。

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

这个例子演示了如果自定义缓存实现。由 type 指定的类必须实现 org.mybatis.cache.Cache 接口。这个接口是 MyBatis 框架比较复杂的接口之一，先给个示例：

```
public interface Cache {
  String getId();
  int getSize();
  void putObject(Object key, Object value);
  Object getObject(Object key);
  boolean hasKey(Object key);
  Object removeObject(Object key);
  void clear();
  ReadWriteLock getReadWriteLock();
}
```

要配置您的缓存，简单地添加一个公共的JavaBeans 属性到您的缓存实现中，然后通过cache 元素设置属性进行传递，下面示例，将在您的缓存实现上调用一个setCacheFile(String file)方法。

```
<cache type=”com.domain.something.MyCustomCache”>
  <property name=”cacheFile” value=”/tmp/my-custom-cache.tmp”/>
</cache>
```

您可以使用所有简单的 JavaBeans 属性，MyBatis 会自动进行转换。

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

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

因为有默认值，所以您不需要使用这种方式明确地配置这些语句。如果您想改变默认的动作，只需要设置flushCache和useCache 属性即可。举个例子来说，在许多的场合下您可能排除缓存中某些特定的select语句。或者您想用select语句清空缓存。同样的，您也可能有一些update语句在执行的时候不需要清空缓存。

cache-ref 元素

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

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

动态 SQL（Dynamic SQL）

MyBatis 最强大的特性之一就是它的动态语句功能。如果您以前有使用JDBC或者类似框架的经历，您就会明白把SQL语句条件连接在一起是多么的痛苦，要确保不能忘记空格或者不要在columns列后面省略一个逗号等。动态语句能够完全解决掉这些痛苦。

尽管与动态SQL一起工作不是在开一个party，但是MyBatis确实能通过在任何映射SQL语句中使用强大的动态SQL来改进这些状况。

动态SQL元素对于任何使用过JSTL或者类似于XML之类的文本处理器的人来说，都是非常熟悉的。在上一版本中，需要了解和学习非常多的元素，但在MyBatis 3 中有了许多的改进，现在只剩下差不多二分之一的元素。MyBatis使用了基于强大的OGNL表达式来消除了大部分元素。

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

if 元素

动态 SQL 最常做的事就是有条件地包括 where 子句。例如：

```
<select id=”findActiveBlogWithTitleLike”
        parameterType=”Blog” resultType=”Blog”>
  SELECT * FROM BLOG
  WHERE state = ‘ACTIVE’
```

```
  <if test=" title != null" >
    AND title like #{title}
  </if>
</select>
```

这条语句提供一个带功能性的可选的文字。如果您没有传入标题，那么将返回所有激活的Blog。如果您传入了一个标题，那它就会查找与这个标题匹配的Blog（在这种情况下，您的参数值可能需要包括任何masking或者通配符）。

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

```
<select id=" findActiveBlogLike"
       parameterType=" Blog"  resultType=" Blog" >
  SELECT * FROM BLOG WHERE state =  'ACTIVE'
  <if test=" title != null" >
    AND title like #{title}
  </if>
  <if test=" author != null and author.name != null" >
    AND title like #{author.name}
  </if>
</select>
```

choose, when, otherwise 元素

有时候我们不想应用所有的条件，而是想从多个选项中选择一个。与 java 中的 switch 语句相似，MyBatis 提供了一个 choose 元素。

让我们继续使用上面的例子，但这次我们只搜索有提供查询标题的，或者只搜索有提供查询作者的数据。如果两者都没有提供，那只返回加精的 Blog（可能是管理员有选择性的查询，而不是返回大量无意义的随机的 Blog）。

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

trim, where, set 元素

考虑一下我们上面提到的‘if ’的例子中，如果现在我们把‘ACTIVE=1’也做为条件，会发生什么情况。

```
<select id=”findActiveBlogLike”
       parameterType=”Blog” resultType=”Blog”>
  SELECT * FROM BLOG
  WHERE
  <if test=”state != null”>
    state = #{state}
  </if>
  <if test=”title != null”>
    AND title like #{title}
  </if>
  <if test=”author != null and author.name != null”>
    AND title like #{author.name}
  </if>
</select>
```

如果我们一个条件都不设置，会发生什么呢？语句最终可能会变成这个样子：

```
SELECT * FROM BLOG
WHERE
```

这将会执行失败。如果只有第二个条件满足呢？语句最终会变成这样：

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

这同样会执行失败。这个问题仅用条件很难简单地解决，如果您已经这么写了，那您可能以后永远都不想犯这样的错了。

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

```
<select id=”findActiveBlogLike”
       parameterType=”Blog” resultType=”Blog”>
  SELECT * FROM BLOG
  <where>
       <if test=”state != null”>
              state = #{state}
       </if>
       <if test=”title != null”>
              AND title like #{title}
       </if>
       <if test=”author != null and author.name != null”>
              AND title like #{author.name}
       </if>
```

```
  </where>
</select>
```

where 元素知道插入“where”如果它包含的标签中有内容返回的话。此外，如果返回的内容以“AND” 或者 “OR”开头，它会把“AND” 或者 “OR”去掉。

如果 *where* 元素的行为并没有完全按您想象的那样，您还可以使用 trim 元素来自定义。例如，下面的 trim 与 where 元素实现相同功能：

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

overrides属性使用了管道分隔的文本列表来覆写，而且它的空白也不能忽略的。这样的结果是移出了指定在overrides 属性里字符，而在开头插入prefix属性中指定的字符。

译者注，下面的两种配置方法效果是一样的：

```
<select id=”findActiveBlogLike”
      parameterType=”Blog” resultType=”Blog”>
  SELECT * FROM BLOG
  <where>
      <if test=”state != null”>
          state = #{state}
      </if>
      <if test=”title != null”>
          AND title like #{title}
      </if>
      <if test=”author != null and author.name != null”>
          AND title like #{author.name}
      </if>
  </where>
</select>
```

```
<select id=”findActiveBlogLike”
      parameterType=”Blog” resultType=”Blog”>
  SELECT * FROM BLOG
  < trim prefix="WHERE" prefixOverrides="AND |OR ">
      <if test=”state != null”>
          state = #{state}
      </if>
      <if test=”title != null”>
          AND title like #{title}
      </if>
      <if test=”author != null and author.name != null”>
          AND title like #{author.name}
```

```
        </if>
    </ trim >
</select>
```

下面的使用 SET 元素也类似。

在动态update语句里相似的解决方式叫做set，这个set元素能够动态地更新列。例如：

```
<update id="updateAuthorIfNecessary"
          parameterType="domain.blog.Author">
    update Author
      <set>
        <if test="username != null">username=#{username},</if>
        <if test="password != null">password=#{password},</if>
        <if test="email != null">email=#{email},</if>
        <if test="bio != null">bio=#{bio}</if>
      </set>
    where id=#{id}
  </update>
```

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

您想知道等同的 trim 元素该怎么写吧，它就像这样 ：

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

注意这种情况，我们剔除了一个后缀， 同时追加了一个前缀。

Foreach 元素

另一个动态 SQL 经常使用到的功能是集合迭代，通常用在 IN 条件句。例如：

```
<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>
```

译者注：

对上面这个映射 SQL 语句的 java 调用代码示例如下：

```
List<Integer> authorIdList = new ArrayList<Integer>();
postList.add(2);
postList.add(3);
postList.add(4);
List<Post> postList = (List<Post>)session.selectList(
          "selectPostIn", postList);
//将会查询出 ID 是 2、3、4 的文章。
```

foreach 元素非常强大，允许您指定一个集合，申明能够用在元素体内的项和索引变量。也允许您指定开始和结束的字符，也可以加入一个分隔符到迭代器之间。这个元素的聪明之处在于它不会意外地追加额外的分隔符。

⇨ 注意：您可以把一个 List 实例或者一个数组作为一个参数对象传递给 MyBatis。如果您这么做，MyBatis 会自动将它包装成一个 Map，并以名字作为 key。List 实例会以“list”作为 key，array 实例会以“ array” 作为 key。

关于 XML 配置文件及 XML 映射文件就讨论到这了，下一章节我们将详细讨论 Java API。

Java API

现在您知道如何配置 MyBatis 和生成映射，您已经收获良多。MyBatis 的 Java API 让您的努力获得回报。正如您将看到的，相比 JDBC，MyBatis 极大地简化了您的代码，并使您的代码保持清晰、容易理解和维护。MyBatis3 推出了一系列重大的改进来使 SQL 映射更好地工作。

目录结构

在我们深入Java API 之前，理解目录结构的最佳实践是非常重要的。MyBatis 非常灵活，您可以对您的文件做任何事，但是做为一个框架，总有一个首选的方式。

让我们来看一下典型的应用目录结构：

```
/my_application
        /bin
        /devlib
        /lib                                    ← MyBatis *.jar 文件存放在这里。
        /src
                /org/myapp/
                        /action
                        /data                   ← MyBatis 物件放在这里。如： 映射器类（Mapper Classes），XML 配置文件，XML 映射文件。
                                /SqlMapConfig.xml
                                /BlogMapper.java
                                /BlogMapper.xml
                        /model
                        /service
                        /view
                /properties                     ← Properties 存放您自己的属性配置文件
        /test
                /org/myapp/
                        /action
                        /data
                        /model                  记住，这只是推荐，并不是必须的，但使用这样通用的目录结构，其它开
                        /service
                        /view
```

```
        /properties                                    发人员将会感激您。
/web
        /WEB-INF
                /web.xml
```

这章节的例子都是假定您按照上面的目录结构来存放文件。

SqlSessions

SqlSession 是与 MyBatis 一起工作的基本 java 接口。通过这个接口，您可以执行命令、获得映射和管理事务。我们很快就会讨论 SqlSession，但首先我们要了解如果获得一个 SqlSession 实例。SqlSessions 是由 SqlSessionFactory 实例创建的。SqlSessionFactory 包含从不同的方式创建 SqlSessions 实例的方法。而 SqlSessionFactory 又是 SqlSessionFactoryBuilder 从 XML 文件，注解或者手动编写 java 配置代码中创建的。

SqlSessionFactoryBuilder

SqlSessionFactoryBuilder 有五个 build() 方法，每个方法允许您从不同来源中创建 SqlSession

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

前四个方法较为常用，它们使用一个引用 XML 文件的 *Reader* 实例，或者更具体地说是上面讨论的 SqlMapConfig.xml 文件。可选参数是 *environment* 和 *properties*。Environment 决定加载的环境（包括数据源和事务管理）。例如：

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

```
</environments>
```

如果您调用一个传递 *environment* 参数的 build 方法，MyBatis 将使用所传递的环境的配置。当然，如果您指定一个不可用的环境，您将收到一个错误。如果您调用的 build 方法没有传递 *environment* 参数，将使用默认的环境（像上面的例子中由 default=“development”指定）。

如果您调用一个传递 properties 实例的方法，MyBatis 将会加载传递进来的属性，并使这些属性在配置文件中生效。这些属性能够应用于配置文件中使用${propName}语法的地方。

回想一下，属性可以从 SqlMapConfig.xml 文件中被引用，也可以直接指定它。因此很重要的是要理解它们的优先顺序。在文档前面已经提到过，这里再次引用以供参考：

如果属性存在于多个地方，MyBatis 按照下面的顺序来加载：

- 首先读入 properties 元素主体中指定的属性。
- 然后会加载类路径或者 properties 元素中指定的 url 的资源文件属性。它会覆盖前面已经读入的重复属性。
- 通过方法参数来传递的属性将最后读取(即通过 sqlSessionFactoryBuilder.build)，同样也会覆盖从 properties 元素指定的和 resource/url 指定的重复属性。

因此优先级最高的属性是通过方法参数来传递的属性，然后是通过 resource/url 配置的属性，最后是在 MyBatis 的映射配置文件中，properties 元素主体中指定的属性。

总得来说，前面四个方法大致相同，通过重载的方式允许您可选地指定 environment 和/或者 properties。这里是一个从 SqlMapConfig.xml 文件中创建 SqlSessionFactory 的例子：

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

注意，我们使用了*Resources*工具类，*Resources*工具类放在org.mybatis.io包中。*Resources*类，正如它的名字暗示，帮助我们从类路径、文件系统或者WEB URL加载资源。浏览一下这个类的源代码，或者使用您的IDE 查看，就会显露一系列有用的方法，简单列表如下：

```
URL getResourceURL(String resource)
URL getResourceURL(ClassLoader loader, String resource)
InputStream getResourceAsStream(String resource)
InputStream getResourceAsStream(ClassLoader loader, String resource)
Properties getResourceAsProperties(String resource)
Properties getResourceAsProperties(ClassLoader loader, String resource)
Reader getResourceAsReader(String resource)
Reader getResourceAsReader(ClassLoader loader, String resource)
File getResourceAsFile(String resource)
```

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

最后一个 build 方法传递一个 Configuration 的实例。Configuration 类包含您需要了解的关于 SqlSessionFactory 实例的所有事情。Configuration 类对于内省的配置很有用，包括发现和操作 SQL 映射。Configuration 类有您已经学过的所有配置开关，像 java API 那样提供方法暴露出来。下面是一个例子，演示如何操作 Configuration 实例，如何把这个实例传递给 build() 方法来创建 SqlSessionFactory。

```
DataSource dataSource = BaseDataTest.createBlogDataSource();
TransactionFactory transactionFactory = new JdbcTransactionFactory();

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

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

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

现在您拥有了一个 SqlSessionFactory，您就能用它来创建 SqlSession 实例了。

SqlSessionFactory

SqlSessionFactory 有六个方法用来创建 SqlSession 实例。在一般情况下，选择其中一个方法要考虑下面几个方面：

- **事务（Transaction）：** 您是否想为会话使用事务作用域，或者自动提交（通常是指数据库或者 JDBC 驱动没有事务的情况下）
- **连接（Connection）：** 您想从配置数据源获得一个连接，还是想自己提供一个？
- **执行（Execution）：** 您想让 MyBatis 重复使用用 PreparedStatements 还是希望批量更新（包括插入和删除）？

重载的 openSession() 方法集，允许您选择任何一种有意义的组合。

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

默认的 openSession()方法不需要参数，创建的 SqlSession 有以下特征：

- 将会启用一个事务作用域 （即不会自动提交）
- 一个连接对象将从正在生效的运行环境所配置的数据源实例中获得
- 事务隔离级别是由驱动或数据源使用的默认级别
- PreparedStatements 不会被重用，也不会进行批量更新

大部分方法自身都是很容易理解的。要启用自动提交，传入“true”值给可选参数 *autoCommit*。要使用自己的连接，可以传入一个连接实例给参数 *connection*。注意，没有提供重载的方法同时传入 Connection 和 autoCommit 参数，因为 MyBatis 将会使用所提供的连接对象正在使用的设置。MyBatis 使用 java 枚举包装器作为事务隔离级别 TransactionIsolationLevel，并有 5 个 JDBC 支持的级别：NONE、READ_UNCOMMITTED、READ_COMMITTED、REPEATABLE_READ、SERIALIZABLE，如果不设置，它们按默认方式工作。

还有一个前面没有遇到过的新参数 ExecutorType，这个枚举定义了三个值：

ExecutorType.SIMPLE
这个类型不做特殊的事情，它只为每个执行语句创建一个 PreparedStatement。

ExecutorType.REUSE

这个类型的执行器（**Executor**）重用 PreparedStatements。

ExecutorType.BATCH
这个执行器（Executor）将会批量执行更新语句，如果 SELECT 在更新语句中被执行，要根据需要进行区分，确保动作易于理解。

译者注：

1. **使用 ExecutorType.SIMPLE**

```
session = factory.openSession(ExecutorType.SIMPLE);
```

对于 XML 映射配置文件中定义的所有 SQL 语句在执行时都由 Connection 创建一个 PreparedStatement 对象来执行。即使在执行同一条 SQL 语句，也要创建一个新的 PreparedStatement 对象。如：

```
<select id="selectBlog" parameterType="int" resultType="Blog">
  select * from Blog where id = #{id}
  </select>
```

执行 N 次 session.selectOne（"selectBlog", #{id}），就要创建 N 个 PreparedStatement 对象，了解一下数据库引擎工作原理，就知道这样会带来不小的开销。

2. **使用 ExecutorType.REUSE**

```
session = factory.openSession(ExecutorType.REUSE);
```

对于 XML 映射配置文件中定义的所有 SQL 语句在执行时都由 Connection 创建一个 PreparedStatement 对象来执行。但 MyBatis 会把每条 SQL 语句的 PreparedStatement 对象缓存起来，等到下次再执行相同的 SQL 语句，则使用缓存的 PreparedStatement 对象。如，执行 N 次 session.selectOne（"selectBlog", #{id}），只是创建一个 PreparedStatement 对象。

3. **使用 ExecutorType.BATCH**

执行批量更新，把要进行批量更新的 SQL 语句作为整体进行打包、编译、优化等，可减少网络流量等。

➔ 注意：SqlSessionFactory 中还有一个方法我们没有提到，那就是getConfiguration()。这个方法返回一个能够在运行期通过反射的方式得到的Configuration 实例。

➔ 注意：如果您已经使用过以前的MyBatis 版本，您可能会回想起会话、事务和批量处理都是分开的。现在不再是这个样子，这三者都完美地包括在一个sesson 作用域里。您不需要分开处理事务和批量处理就能获得它们的全部好处。

SqlSession

正如前面提到的，SqlSession 实例是 MyBatis 里最强大的类。SqlSession 实例里您会找到所有的执行语句、提交或者回滚事务、获得 mapper 实例的方法。

在 SqlSession 类里有超过 20 个方法，现在让我们把它们拆分成容易理解的分组。

语句执行方法组（Statement Execution Methods）

这些方法用来执行定义在 SQL 映射 XML 文件中的 select ， insert,update 和 delete 语句。它们都很好理解，执行时使用语句的 ID 和并传入参数对象（基本类型，javaBean，POJO 或者 Map）。

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

selectOne 方法和 selectList 方法的不同之处是 selectOne 必须返回一个对象，如果返回一个以上的对象或者返回 0 个对象，将会抛出异常。如果不能预知会返回多少对象，最好使用 selectList 方法。如果您只是检测一下是否存在一个对象，可以使用 SQL 语法中的 COUNT 返回一个计数（0 或者 1）。因为不是所有语句都需要传入参数的，因此这些方法可以重载为不需要参数的版本。

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

最后，它们还有三个 select 方法的高级版本，允许您限定返回行记录的范围或者提供自定义的结果处理逻辑。这三个方法一般用来处理大量的数据集

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

RowBounds 参数促使 MyBatis 跳过指定数量的记录，或者返回指定数量的结果集。RowBounds 类有一个构造函数，设定偏移量或者指定条数，如果没进行设置，那就没有限制了。

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

不同的驱动程序在这方面可以达到的效率是不同的。为了得到最好性能，使用 SCROLL_SENSITIVE 或者 SCROLL_INSENSITIVE 参数设置来返回结果集（换名话说，不要使用 FORWARD_ONLY）。

（译者注）默认情况下获得的结果集是不能更新的，且只有一个向前移动的光标。下面

是一个可更新可滚动的结果集：

```
Statement stmt = con.createStatement(
                                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                                    ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT a, b FROM TABLE2");
```

ResultHandler 参数允许您按照自己喜欢的方式对每条记录进行操作。您能把每条记录加入一个 list、Map 或者 Set，或者只是进行数量统计。您能让 ResultHandler 做许多事情，它同样被 MyBatis 用来生成结果集列表（result set lists）。

这个接口非常简单：

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

ResultContext 参数允许您访问结果对象本身，创建一定数量的结果对象，或者使用返回值是 Boolean 的 stop()方法来停止加载更多的结果集。

事务控制方法组（Transaction Control Methods）

有四个控制事务作用域的方法，当然，如果您使用了自动提交或者正在使用的是外部事务管理器，那这四个方法就没什么作用。然而，如果您使用由 Connection 实例管理的 JDBC 的事务管理器，那这四个方法就非常管用：

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

默认地 MyBatis 不会自动提交，除非它检测到数据库被 insert，update 或者 delete 改变。如果在哪里作了修改而没有使用这些方法，那么，您需要传入 true 到 commit 和 rollback 方法，以保证它会提交。注意：您依然不能把一个会话或者一个使用外部事务的管理器强制设成自动提交模式。大部分时间，您不需要调用 rollback()方法，因为如果您不调用 commit 方法的话，MyBatis 会为您进行回滚。然而，如果您需要在一个有多个提交或者多个回滚可能的会话中获得更好（更细粒度）的控制，那么可以使用 rollback 选项来实现。

清除会话层缓存（Clearing the Session Level Cache）

```
void clearCache()
```

SqlSession 实例有一个本地缓存，这个缓存在每次提交，回滚和关闭时进行清除。如果不想在每次提交或者回滚时都清空缓存，可以明确地调用 clearCache()方法来关闭。

确保 SqlSession 已经关闭（Ensuring that SqlSession is Closed）

```
void close()
```

最重要的事情是要确保关闭已经打开的会话，而最好的方式来保证是用下面的模式：

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

➔ 注意： 就像 SqlSessionFactory，您能够使用 SqlSession 的 getConfiguration()方法来获得 Configuration 的实例。

```
Configuration getConfiguration()
```

使用 Mappers

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

虽然前面提到的 insert, update, delete 和 select 方法很强大，但同时，它们也有点冗长，且不是类型安全的，IDE 或者单元测试也帮助发现不了其中的错误。我们在文章最开始就看到过使用 Mappers 的例子，我们可以回顾一下。

因此，一个最常用的方式是使用 Mapper 接口来执行映射语句。一个 Mapper 接口定义的方法要与 SqlSession 执行的方法相匹配，即 Mapper 接口方法名与映射 SQL 文件中的映射语句 ID 相同。下面例子演示了这种用法。

```
public interface AuthorMapper {
  // (Author) selectOne(“selectAuthor”,5);
  Author selectAuthor(int id);
  // (List<Author>) selectList(“selectAuthors”)
  List<Author> selectAuthors();
```

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

总体来说，每个Mapper接口方法签名，都要与一个SqlSession 的方法相对应，方法名也要与映射语句的ID相对应。

另外，返回的类型要与期望的结果类型一致。支持返回所有类型，包括：基本数据类型，Maps，POJOs 和 JavaBeans。

➔ Mapper 接口不需要实现任何接口或者继承任何类，只要方法签名能够唯一地与映射语句相对应就可以了。

➔ Mapper 接口可以继承其它接口。当您使用 XML 绑定 Mapper 接口时，要确保在同一命名空间里有您使用的语句。唯一的限制是不能同时有相同的方法签名在两个不同层次的接口。

您能够传递多个参数给 mapper 方法，如果这么做，要它们默认的参数列表位置来命名，例如这样：#{1}, #{2}等，如果您想改变参数的名字（多参数情况下），那您可以使用 @Param(“paramName”) 注解的方式来传入参数。

您同样也可以传递一个 RowBounds 实例到方法，用来限定查询结果范围。

Mapper 注解

从很早的时候，MyBatis 就使用 XML 驱动的框架，基于 XML 配置，映射语句都定义在 XML 中。在 MyBatis3 中，还有新的可选方案。MyBatis3 建立于广泛且强大的 java 配置 API 之上。java 配置 API 是基于 XML 的 MyBatis 配置的基础，同时也是基于注解的配置基础。注解提供了一个简单的方式来执行简单映射语句而不引入大量的开销。

➔ 注意: 很不幸，java 注解在表现力与灵活性上是有限的。尽管花了很多时间来研究，设计与试验，但是强大的 MyBatis 映射不能够建立在注解之上。C#属性则不会有这种限制。虽然如此，基于注解的配置并非没有好处的。

注解配置属性如下表：

注解	使用范围	等同 XML	描述
@CacheNamespace	Class	<cache>	在指定的命令空间里配置缓存，如类里。 属性有： implementation, eviction,

			flushInterval, size 和 readWrite.
@CacheNamespaceRef	Class	<cacheRef>	引用另一个命名空间的缓存。 属性:value：命名空间名（也就是完全类路径名).
@ConstructorArgs	Method	<constructor>	收集一组结果集传入到一个结果对象的构造器。 属性：value：参数对象的数组。
@Arg	Method	<arg> <idArg>	单个构造器参数，是 ConstructorArgs 集合的一部分。 属性： Id， column， javaType，jdbcType，typeHandler。 这个 id 属性值是一个字符串类型的值，用于标识这个属性，并被用来进行比较。类似于<idArg>这个 XML 元素（原文：The id attribute is a **boolean** value that identifies the property to be used for comparisons, similar to the <idArg> XML element.说是 boolean 值应该有误）。
@TypeDiscriminator	Method	<discriminator>	一组 cases 值，用来决定哪一个结果会被映射。 属性： column, javaType, jdbcType, typeHandler, cases。 cases 属性是一个 Cases 数组 。
@Case	Method	<case>	单个值的 case ，与映射对应。 属性：value， type，results。 Results 属性是一个结果集数组，因此 Case 注解与 ResultMap 很相似，由下面的 Results 注解指定。
@Results	Method	<resultMap>	结果映射集。其中包含如何将结果列映射到属性或者字段的详细信息。 属性： Value：一个结果集注解的数组。
@Result	Method	<result> <id>	一个列和属性或字段之间的单个结果映射。 属性：id， column， property，javaType， jdbcType，typeHandler，one， many。 这个 id 属性值是一个字符串类型的值，用于标识这个属性，并被用来进行比较。类似于<idArg>这个 XML 元素。one 属性是单个 association，与<association>元素相似。它们的命名要避免与类名相冲突。

@One	Method	<association>	映射到单个复杂类型属性。 属性：select：映射语句全称（也就是映射方法），能够加载一个相应类型的实例。 注意：您会发现连接映射没有被注解 API 支持。这就是注解的局限性，它不允许循环引用。
@Many	Method	<collection>	映射到一个复杂类型的集合属性。 属性：select：映射语句全称（也就是映射方法），能够加载一个相应集合的实例。 注意：您会发现连接映射没有被注解 API 支持。这就是注解的局限性，它不允许循环引用。
@Options	Method	映射语句属性	这个注解提供访问各种开关和配置选项。Options 注解提供一致的和清晰的方式来访问这些开关与配置。 属性：useCache=true，flushCache=false，resultSetType=FORWARD_ONLY，statementType=PREPARED，fetchSize=-1， timeout=-1，useGeneratedKeys=false，keyProperty=“id”。 理解 java 注解是很重要的，它没有指定“null”作为一个值的方式。因此，一旦使用 Options 注解，您的语句将使用所有默认选项值。注意默认的值以避免出现不是您所期待的行为结果。
@Insert **@Update** **@Delete** **@Select**	Method	<insert> <update> <delete> <select>	其中的每一个注解都代表着要被执行的实际 SQL。它们都可以传入一个字符串数组（或者单个字符串）。如果传递了一个字符串数组，它们就会使用一个空格将这些字符串分开，然后连接在一起。这样帮助在 java 代码中生成 SQL 语句时避免空格丢失问题。当然如果您喜欢，传入一条字符串也可以。 属性：value：传入的字符串数组。这些数组将形成单一 SQL 语句。
@InsertProvider **@UpdateProvider** **@DeleteProvider** **@SelectProvider**	Method	<insert> <update> <delete> <select> 允许动态生成 SQL。	这些可选的 SQL 注解允许您在运行的过程中，执行指定类和方法返回的 SQL 。当执行映射语句时，MyBatis 将会实例化 provider 指定的类，执行其中的方法。这方法可选地接受参数对象作为它唯一的参数，但必须只指定一个参数或

			者不指定任何参数。 属性：type，method。 type 属性指定类的全名，method 属性指定这个类中的方法名称。注意：下一章中，我们会讨论 SelectBuilder 类，它可以用一种更清晰、更易读的方式生成动态 SQL。
@Param	Parameter	N/A	如果您的映射方法有多个参数，那这个注解能为方法的每个参数设定一个参数名称。否则，多个参数会使用它们的顺序号命名（不包括任何 RowBounds 参数），例如：#{1}，#{2}等等，这是默认的方式。如果使用@Param(“person“ ）， 则这个参数就被叫做 #{person}。

SelectBuilder

Java 开发人员最讨厌的事情就是不得不在 java 代码中嵌入 SQL 语句。通常这样做的原因是 SQL 必须动态生成，要不然，您可以把 SQL 定义在外部文件或者存储过程中。正如您所看到的，MyBatis 在它的 XML 映射功能中对动态 SQL 的生成有一个很强大的解决方案。然而，有时候还是不得不在 java 代码中生成 SQL 语句字符串，这种情况下，MyBatis 还有一个功能可以帮助到您，用于减少加号、引号、换行、格式问题和在嵌套条件中处理额外的逗号与 AND 连词……事实上，在 java 中动态生成 SQL 代码可以说是一场真正的噩梦。

MyBatis3 引入一个稍微有点不同的概念来处理这个问题。我们可以生成一个类的实例，然后调用这个实例的方法来一步生成一条 SQL 语句，最终的 SQL 看起来更像是 java 代码而不像 SQL 代码。我们正在尝试不同的东西，最终结果接近一种领域特定语言（DSL），这也是 java 目前形态要达到的目标。

SelectBuilder 的秘密

SelectBuilder 类并没有什么神奇的地方，如果您不了解它如何工作，它对我们也没任何好处。SelectBuilder 使用一组静态导入方法和一个 ThreadLocal 变量来启用一个能够很容易地组合条件并会注意所有 SQL 格式的语法。例如：

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

这只是一个您可能只选择静态生成 SQL 的简单例子，下面是一个更复杂的例子：

```
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();
  }
```

如果使用字符串连接，上面的 SQL 语句就会像下面这样来生成：

```
"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";
```

如果您喜欢上面这种语法，那您仍然可以使用它，但是它很容易出错，要注意在每行结束的地方都加一个空格。即使您喜欢上面的语法，下面的例子毫无争辩地比在 java 代码中连接字符串更简单：

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

```
    }
    if (p.lastName != null) {
      WHERE("P.LAST_NAME like #{lastName}");
    }
    ORDER_BY("P.LAST_NAME");
    return SQL();
  }
```

上面的例子有什么特别之处呢？如果您仔细观察的话，您会发现您不需要再担心是否会遇外地加入了重复的“AND”关键字，或者考虑要使用“WHERE”还是“AND”或者两者都使用。上面的例子生成一个查询所有PERSON记录的语句，查出与ID匹配的、或者与firstName匹配的、或者与lastName匹配的记录，或者是与上述三种任何组合条件相匹配的记录。SelectBuilder 会知道哪里需要添加“WHERE”，哪里需要使用“AND”，以及会注意字符串的连接。最重要的是，它几乎不管您调用这些方法的顺序（只有一个例外就是使用OR()方法）。

您可能已经关注到这两个方法：BEGIN() and SQL()。概括地说，每一个 SelectBuilder 方法都要以 BEGIN()开头，以 SQL()结束，这也是生成 SQL 的范围。在 BEGIN()和 SQL()之间，您可以提取 SelectBuilder 的方法来一步一步分解您的逻辑。BEGIN()方法清除了 ThreadLocal 变量以保证不会有先前残留的内容，SQL()方法装配自从调用 BEGIN()后所有 SelectBuilder 方法生成的 SQL 语句。注意：BEGIN()方法有一个同义词方法 RESET()，它们做相同的事，在某些情况下使用 RESET()可读性会更强。

要像上面那样使用 SelectBuilder 的方法，您简单地只要使用静态导入就可以了，如：

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

一旦被导入，您就能够使用 SelectBuilder 类的所有方法。SelectBuilder 类的方法如下：

Method	Description
BEGIN() / RESET()	这个方法清空 SelectBuilder 的 ThreadLocal 状态，并准备好生成一个新的语句。BEGIN()方法放在句子的开头可读性比较好。RESET()方法在执行过程中由于一些原因（可能在某些情况下，执行逻辑需要一个完全不同的语句）清空语句后使用，可读性比较好
SELECT(String)	开始或者追加一个 SELECT 子句。也可被多次调用，参数将会被追加到 SELECT 语句中。参数通常是用一个逗号分隔的列名和别名字符串，或者是任何驱动器支持的语法。
SELECT_DISTINCT(String)	开始或者追加一个SELECT 子句，在生成的查询语句中增加关键字“DISTINCT”。也可被多次调用，参数将会被追加到SELECT 语句中。参数通常是用一个逗号分隔的列名和别名字符串，或者是任何驱动器支持的语法。
FROM(String)	开始或者追加一个 FROMT 子句。也可被多次调用，参数将会被追加到 FROM 语句中。参数通常是一个 table 名或者别名，或者是驱动器支持的语法。
JOIN(String) INNER_JOIN(String) LEFT_OUTER_JOIN(String)	增加一个恰当类型的 JOIN 子句，这由调用的方法决定。参数是标准的由列和条件连接在一起的 JOIN 子句。

Method	Description
RIGHT_OUTER_JOIN(String)	
WHERE(String)	追加一个新的条件子句，可以被多次调用，每个子句使用 AND 来连结。如果使用 OR() ，则使用 OR 来分隔。
OR()	使用 OR 来分隔 WHERE 条件子句，可以被多次调用，但是对某一行调用超过一次则可能生成错误的 SQL 语句。
AND()	使用 AND 来分隔 WHERE 条件子句，可以被多次调用，但是对某一行调用超过一次会产生错误的 SQL 语句。因此 WHERE 和 HAVING 都会自动的添加 AND 来连结语句。这个方法的使用非常罕见，包含这个方法的原因可能就为了生成 SQL 语句语法上的完整性。
GROUP_BY(String)	追加一个 GROUP BY 子句，可以被多次调用，每个子句由逗号连接。
HAVING(String)	追加一个 HAING 条件子句，可以被多次调用，每个子句由 AND 连接。如果使用 OR() ，则使用 OR 来分隔。
ORDER_BY(String)	追加一个 ORDER BY 子句，可以被多次调用，每个子句由逗号分隔。
SQL()	SQL()方法返回生成的 SQL 语句，并且重新设置 SelectBuilder 状态（只有 BEGIN() 或者 RESET()已经被调用时）。因此，这个方法只能被调用一次。

SqlBuilder

与SelectBuilder类似，MyBatis 也包含了一个通用的SqlBuilder类，它包含了SelectBuilder的所有方法，同时也有一些针对inserts, updates, 和deletes 的方法。这个类在DeleteProvider 、nsertProvider和UpdateProvider （以及SelectProvider ）里生成SQL 语句时非常有用。

要像上面的例子那样使用 SqlBuilder，同样只要使用静态导入就可了，例如：

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

SqlBuilder 包含了 SelectBuilder 的所有方法, 同时还有一些额外的方法：

Method	Description
DELETE_FROM(String)	开始一个 delete 语句，同时指定要删除的表。通常它应该在 WHERE 语句前被调用。
INSERT_INTO(String)	开始一个插入语句，同时指定插入的表。它应该在一个或者多个 VALUES （）方法之前被调用。
SET(String)	追加“set”列表到更新语句中。
UPDATE(String)	开始一个更新语句，同时指定更新的表。在一个或者多个 SET （）方法前被调用，通常情况下，WHERE()方法连接在它后面。
VALUES(String, String)	追加一个插入语句，第一个参数是要插入的一列名，第二个参数是要插入的值。

下面是些例子：

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

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

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

结束语

MyBatis 是一款方便、简单又不失灵活的持久层框架。至此，官方英文版用户指南也结束了，感谢 GOOGLE，因为在翻译本文档时，GOOGLE 翻译帮了大忙。

但是，用户指南对 MyBatis 的介绍还远远不够。接下来要靠我们继续去学习、使用和挖掘了，这样才能真正地了解或掌握 MyBatis。

以后的附录章节是译者学习和使用 MyBatis 的一些笔记记录，供大家参考，与大家共勉。

附录 1 对象模型

为保持与官方用户指南的连贯性，并且可以使用文中的例子，帮助理解，仍然使用官方文档中的对象模型：包括由一个作者写的一个博客，一个博客有许多文章（Post，帖子），每个文章由 0 个或者多个评论和标签组成。

首先建立对象模型。

Author.java

```
package org.mybatis.model;

public class Author {

    private Integer id;

    private String username;

    private String password;

    private String email;

    private String bio;

    //省略get和set方法
}
```

Blog.java

```
package org.mybatis.model;

public class Blog {

    private Integer id;

    private String title;

    private Integer authorId;

    //省略get和set方法
}
```

Post.java

```
package org.mybatis.model;

public class Post {

    private Integer id;

    private Integer blogId;

    private Integer authorId;

    private String createdOn;

    private String section;

    private String subject;

    private String body;

   //省略get和set方法}
```

Comment.java

```
package org.mybatis.model;

public class Comment {

    private Integer id;

    private Integer postId;

    private String name;

    private String comment;

    //省略get和set方法
}
```

Tag.java

```
package org.mybatis.model;

public class Tag {
```

```
    private Integer id;

    private String name;

    public Integer getId() {

    //省略get和set方法
}
```

要使用 MyBatis，首先要有一个数据库，这里选用 Java6 自带的 JavaDB，也就是 derby 数据库。使用 JavaDB 内嵌模式，可以不需要独立维护一个数据库服务器。

附录 2 就是使用 JavaDB 来搭建数据库环境的过程。

附录 2 创建数据库

JavaDB 作为 JDK 6 内嵌的数据库，使得程序员不再需要耗费大量精力安装和配置数据库，就能进行安全、易用、标准、并且免费的数据库编程。

在安装 JDK6 的时候，也同时会安装 JavaDB，并包含了一些实用工具，我们可以直接使用 JavaDB 自带的工具来启动、查看数据库等。特别是 ij 工具，可以用来创建数据库、创建表、执行 SQL 语句等等操作。JavaDB 有两种工作模式：内嵌模式和网络服务器模式。这里选用网络服务器模式，独立启动 JavaDB，并用 ij 工具进行连接操作。但是在程序中使用 derby 数据库时是选用内嵌模式，这样就不用启动 derby 服务器了，请留意后面章节的连接数据库配置。

对于 JavaDB 的介绍可参考其它相关资料。下面开始创建数据库、创建表和插入准备数据……

第一步，双击 JavaDB\bin\startNetworkServer.bat，启动 JavaDB 数据库。

第二步，双击 JavaDB\bin\ ij.bat，启动 ij 工具。

第三步，连接并创建数据库。数据库名：blogDB。

```
ij> connect   'jdbc:derby://localhost:1527/blogDB; create=true';
```

使用参数 create=true 表明，连接数据库服务器并创建一个数据库。创建数据库以后连接就不再需要参数 create=true 了，但是使用也无妨。

```
ij> connect   'jdbc:derby://localhost:1527/blogDB';
```

第四步，执行 SQL 脚本文件，创建表、插入数据。

```
ij> connect 'jdbc:derby://localhost:1527/blogDB; user=lory';
```

这次使用了参数 user=lory，本次连接使用用户名 lory 来进行连接。指定用户名也就是指定了模式（schema），那么可以直接使用表名来访问，而不需要用 schema.tablename 的形式，例如，不指定用户名来连接服务器，要访问 lory 用户创建的 blog 表，就需要通过 lory.blog 的方式来访问。这里的用户名也是 MyBatis 配置数据库连接参数所指定的用户名。

执行 SQL 脚本。如执行事先准备好创建作者的脚本 Author.sql：

```
ij> run 'E:\blogDB\Author.sql';
```

Author.sql：

```
drop table author;

create table author
(
    id int not null,
    username varchar(10) not null,
    password  varchar(30) not null,
    email varchar(30),
    bio varchar(30)
);


insert into author(id,username,password,email,bio)
values(1,'user1','user1','user1@163.com','guy');

insert into author(id,username,password,email,bio)
values(2,'user2','user2','user2@163.com','guy');

insert into author(id,username,password,email,bio)
values(3,'user3','user3','user3@163.com','guy');

insert into author(id,username,password,email,bio)
values(4,'user4','user4','user4@163.com','guy');

insert into author(id,username,password,email,bio)
values(5,'user5','user5','user5@163.com','guy');

insert into author(id,username,password,email,bio)
values(6,'user6','user6','user6@163.com','guy');
```

同理，我们再创建四张表：blog，post，comment，tag。

Blog.sql:

```
drop table blog;

create table blog
(
 id     int not null primary key,
 title  varchar(30) default 'My Blog',
 author_id int not null
);


insert into blog(id,title,author_id) values(1,'just fun',1);
```

```
insert into blog(id,title,author_id) values(2,'just funny',2);
insert into blog(id,author_id) values(3,3);
insert into blog(id,author_id) values(4,4);
insert into blog(id,title,author_id) values(5,'hello one',5);
insert into blog(id,title,author_id) values(6,'hello two',6);
```

Post.sql:

```
drop table post;
create table post
(
    id int not null,
    blog_id int not null,
    author_id int not null,
    created_on varchar(30),
    section varchar(30),
    subject varchar(30),
    body varchar(100)
);

insert into post(id,blog_id,author_id,created_on,section,subject,body)
values(1,1,1,'2010-08-04','photo','ddd','nothing');
insert into post(id,blog_id,author_id,created_on,section,subject,body)
values(2,1,1,'2010-08-05','photo','hello','nothing too');
insert into post(id,blog_id,author_id,created_on,section,subject,body)
values(3,1,1,'2010-08-06','photo','ddfdfdd','also nothing');
insert into post(id,blog_id,author_id,created_on,section,subject,body)
values(4,2,2,'2010-08-06','photo','hi','nothing more');
```

Comment.sql:

```
drop table comment;
create table comment
```

```
(
    id int not null,
    post_id int not null,
    name varchar(30),
    comment varchar(50)
)
```

Tag.sql:

```
drop table tag;
create table tag
(
 id int not null,
 name varchar(30)
)
```

至此，一个可供使用的数据库已经建成。上面创建的表都比较简单，没有创建外键、索引等。接下来就开始 MyBatis 之旅。

附录 3 MyBatis 实例

- 简单 select

一切从简单开始，下面是一个 Mapper XML 配置，包含了基本的部分。

Sqlconfig.xml

```
<?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>
  <properties resource="org/mybatis/data/sqlconfig.properties" />
  <typeAliases>
    <typeAlias alias="Author" type="org.mybatis.model.Author"/>
    <typeAlias alias="Blog" type="org.mybatis.model.Blog"/>
    <typeAlias alias="Comment" type="org.mybatis.model.Comment"/>
    <typeAlias alias="Post" type="org.mybatis.model.Post"/>
    <typeAlias alias="Tag" type="org.mybatis.model.Tag"/>
  </typeAliases>

  <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/data/BlogMapper.xml"/>
    <mapper resource="org/mybatis/data/AuthorMapper.xml"/>
    <mapper resource="org/mybatis/data/PostMapper.xml"/>
   <!--  <mapper resource="org/mybatis/data/CommentMapper.xml"/>  -->
   <!-- <mapper resource="org/mybatis/data/TagMapper.xml"/> -->
  </mappers>
  </configuration>
```

sqlconfig.properties

```
#derby Network Server
#driver=org.apache.derby.jdbc.ClientDriver
#url=jdbc:derby://localhost:1527/blogDB;


#derby embedded model
driver=org.apache.derby.jdbc.EmbeddedDriver
#url=jdbc:derby:blogDB;
url=jdbc:derby:D:\\Program Files\\Java\\JavaDB\\bin\\blogDB;

username=lory
password=lorry
```

说明：这里使用 derby 数据库，它有两种配置方式。

网络服务器模式使用配置：

```
driver=org.apache.derby.jdbc.ClientDriver
url=jdbc:derby://localhost:1527/blogDB;
```

内嵌模式使用配置：

```
driver=org.apache.derby.jdbc.EmbeddedDriver
#url=jdbc:derby:blogDB;
url=jdbc:derby:D:\\Program Files\\Java\\JavaDB\\bin\\blogDB;
```

url直接指向本地已经建好的blogDB数据库。程序中使用内嵌模式就不需要独立启动derby数据库服务器了。此时的derby数据库只接受一个连接，但作为练习，是没有问题的。

BlogMapper.xml

```
<?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.model.BlogMapper">
  <select id="selectBlog_by_id" parameterType="int" resultType="Blog">
    select * from Blog where id = #{id}
  </select>

    <select id="selectBlog_by_id_Map" parameterType="HashMap" resultType="Blog">
    select * from Blog where id = #{id}
  </select>

    <select id="selectBlog_by_bean" parameterType="Blog" resultType="Blog">
    select * from Blog where id = #{id}
    </select>
</mapper>
```

接着创建 XML 配置管理器。

SqlMapperManager:

```
package org.mybatis.service;

import java.io.IOException;
import java.io.Reader;

import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

public class SqlMapperManager
{
	private static SqlSessionFactory factory = null;
	private static String fileName =  "org/mybatis/data/Sqlconfig.xml";

	private SqlMapperManager()
	{

	}

	public static void initMapper(String sqlMapperFileName)
	{
		fileName = sqlMapperFileName;
	}

	public static SqlSessionFactory getFactory()
	{
		try
		{
			if (factory == null)
			{
				Reader reader = Resources
				    .getResourceAsReader(fileName);
				SqlSessionFactoryBuilder builder =
				     new SqlSessionFactoryBuilder();
				factory = builder.build(reader);
				builder = null;
			}
		}
		catch (IOException e)
		{
			e.printStackTrace();
			return null;
		}
```

```
            return factory;
        }
}
```

创建测试类：

SimpleMapper.java

```
package org.mybatis.action;

import java.util.HashMap;

import org.apache.ibatis.session.SqlSession;
import org.apache.log4j.BasicConfigurator;
import org.mybatis.model.Blog;
import org.mybatis.service.SqlMapperManager;

public class SimpleMapper {

        /**
         * @param args
         */
        public static void main(String[] args)
        {
                BasicConfigurator.configure();
                SqlSession session = null;
                Blog blog = null;
                try
                {
                        SqlSessionFactory factory = SqlMapperManager.getFactory();
                        if (factory == null)
                        {
                                System.out.println("get SqlSessionFactory failed.");
                                return;
                        }
                    session = factory.openSession();
                        HashMap<String, Integer>  paramMap =
                                  new HashMap<String, Integer>();
                        paramMap.put("id", 2);

                        Blog myBlog = new Blog();
                        myBlog.setId(3);

                        blog = (Blog) session.selectOne(
                                        "selectBlog_by_id", 1);
```

```
                        pringBlog(blog);

                        blog = (Blog) session.selectOne(
                                        "selectBlog_by_id_Map", paramMap);
                        pringBlog(blog);

                        blog = (Blog) session.selectOne(
                                        "selectBlog_by_bean", myBlog);
                        pringBlog(blog);
                }
                catch(Exception e)
                {
                        e.printStackTrace();
                }
                finally
                {
                  session.close();
                }
        }

        public static void pringBlog(Blog blog)
        {
                if (blog != null)
                {
                        System.out.println("ID:" + blog.getId());
                        System.out.println("title:" + blog.getTitle());
                        System.out.println("authorID:" + blog.getAuthorId());
                }
                else
                {
                        System.out.println("blog=null");
                }
        }

}
```

运行 SimpleMapper.java 结果，控制台输出信息：

```
D:\Programs\workspace\MyBatis3
0 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - PooledDataSource
forcefully closed/removed all connections.
16 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - PooledDataSource
forcefully closed/removed all connections.
16 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - PooledDataSource
forcefully closed/removed all connections.
16 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - PooledDataSource forc
```

```
efully closed/removed all connections.
6787 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - Created connection
24585668.
6866 [main] DEBUG java.sql.Connection  - ooo Connection Opened
10158 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: select * from Blog where id
= ?
10158 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 2(Integer)
10517 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, TITLE, AUTHOR_ID
10517 [main] DEBUG java.sql.ResultSet  - <==        Row: 2, just funny, 2
ID:2
title:just funny
authorID:null
11266 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: select * from Blog where id
= ?
11266 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 1(Integer)
11266 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, TITLE, AUTHOR_ID
11266 [main] DEBUG java.sql.ResultSet  - <==        Row: 1, just fun, 1
ID:1
title:just fun
authorID:null
11283 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: select * from Blog where id
= ?
11283 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 3(Integer)
11283 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, TITLE, AUTHOR_ID
11298 [main] DEBUG java.sql.ResultSet  - <==        Row: 3, My Blog, 3
ID:3
title:My Blog
authorID:null
11298 [main] DEBUG java.sql.Connection  - xxx Connection Closed
11298 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - Returned
connection 24585668 to pool.
```

从上面可以看出，要传递什么参数有多种选择。如果是要传递多个参数，如 insert 或 update，在没有模型（javaBean）的情况下，使用 HashMap 应该是比较方便的。HashMap 同样可以保存映射结果。.虽然这对许多场合下有用，但是 HashMap 却不是非常好的域模型。

从结果可以看到，authorID:null，因为数据库中的author_id与Blog没有匹配的字段，因此结果为null，并且使用“*”来代替查询的字段也是很拙劣的做法。解决字段不匹配的问题有两种方式，如把BlogMapper.xml修改如下：

```
<?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.model.BlogMapper">
```

```
  <resultMap id="blogResultMap" type="Blog">
    <id property="id" column="id" />
    <result property="title" column="title" />
    <result property="authorId" column="author_id" />
  </resultMap>

    <select id="selectBlog_use_as" parameterType="HashMap" resultType="Blog">
     select id , title, author_id as authorid  from Blog where id = #{id}
    </select>

    <select id="selectBlog_use_resultMap" parameterType="HashMap" resultMap="blogResultMap">
        select  id , title, author_id from Blog where id = #{id}
    </select>
</mapper>
```

然后在代码里调用：

```
HashMap<String, Integer>  paramMap = new HashMap<String, Integer>();
paramMap.put("id", 2);

Blog myBlog = new Blog();
myBlog.setId(3);

blog = (Blog) session.selectOne("selectBlog_use_as", myBlog);
pringBlog(blog);

blog = (Blog) session.selectOne("selectBlog_use_resultMap", paramMap);
pringBlog(blog);
```

控制台输出结果：

```
9611 [main] DEBUG java.sql.Connection  - ooo Connection Opened
12903 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: select id , title, author_id
as authorid from Blog where id = ?
12903 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 9(Integer)
13169 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, TITLE, AUTHORID
13169 [main] DEBUG java.sql.ResultSet  - <==        Row: 9, I Love Photh too, 3
ID:9
title:I Love Photh too
authorID:3
13294 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: select id , title, author_id
from Blog where id = ?
13294 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 2(Integer)
13294 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, TITLE, AUTHOR_ID
```

```
13294 [main] DEBUG java.sql.ResultSet  - <==        Row: 2, just funny, 2
ID:2
title:just funny
authorID:2
13294 [main] DEBUG java.sql.Connection  - xxx Connection Closed
13294 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - Returned
connection 9936523 to pool.
```

其实，parameterType="HashMap"不是必须的，MyBatis 有足够的信息可以知道输入的对象是什么。注意 selectBlog_use_resultMap，虽然设置的是 parameterType="HashMap"，但实际传入的是一个 Blog 对象。

- update，delete，insert

对于 update，delete，insert，看下面的例子：

映射配置文件：

```
    <update id="updateBlog_use_bean" statementType="PREPARED" parameterType="Blog">
        update blog set title= #{title}, author_id=#{authorId} where id = #{id}
    </update>

    <delete id="deleteBlog_use_bean" statementType="PREPARED" parameterType="Blog">
        delete from blog where id = #{id}
    </delete>

    <insert id="insertBlog_user_bean" statementType="PREPARED" parameterType="Blog">
        insert into blog(id, title, author_id) values(#{id}, #{title}, #{authorId})
    </insert>
```

调用代码：

```
Blog myBlog = new Blog();
myBlog.setId(3);
myBlog.setTitle("I Love Photh");
myBlog.setAuthorId(3);

session.update("updateBlog_use_bean", myBlog);
session.delete("deleteBlog_use_bean", myBlog);
session.insert("insertBlog_user_bean", myBlog);
```

控制台输出结果：

```
2476 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: update blog set title= ?,
author_id=? where id = ?
2476 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: I Love Photh(String),
3(Integer), 3(Integer)
2731 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: delete from blog where id = ?
2731 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 3(Integer)
2792 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: insert into blog(id, title,
author_id) values(?, ?, ?)
2797 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 3(Integer), I Love
Photh(String), 3(Integer)
```

默认地 MyBatis 不会自动提交，最后还需要使用 session.commit()来提交修改。但可以调用下面的方法来自动提交。

```
SqlSession openSession(boolean autoCommit)
```

- 自动生成主键

下面是自动生成 blog 的 id 的值，生成规则是：当前表中最大的 id 值加 1。

映射配置文件：

```
<insert id="insertBlog_user_autokey" statementType="PREPARED" parameterType="Blog">
 <selectKey keyProperty="id" resultType="int" order="BEFORE">
 select max(id)+1 from blog
 </selectKey>
    insert into blog(id, title, author_id) values(#{id}, #{title}, #{authorId})
</insert>
```

调用代码：

```
Blog myBlog1 = new Blog();
myBlog1.setTitle("I Love Photh");
myBlog1.setAuthorId(3);
session.insert("insertBlog_user_autokey", myBlog1);
session.insert("insertBlog_user_autokey", myBlog1);
session.insert("insertBlog_user_autokey", myBlog1);
```

虽然插入同样的 myBlog1，但是 myBlog1.id 是不一样的。

控制台输出结果：

```
D:\Programs\workspace\MyBatis3
0 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - PooledDataSource
forcefully closed/removed all connections.
16 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - PooledDataSource
forcefully closed/removed all connections.
16 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - PooledDataSource
forcefully closed/removed all connections.
16 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - PooledDataSource
forcefully closed/removed all connections.
1560 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - Created connection
17007273.
1592 [main] DEBUG java.sql.Connection  - ooo Connection Opened
2014 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: select max(id)+1 from blog
2014 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters:
2435 [main] DEBUG java.sql.ResultSet  - <==    Columns: 1
2435 [main] DEBUG java.sql.ResultSet  - <==        Row: 7
2575 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: insert into blog(id, title,
author_id) values(?, ?, ?)
2575 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 7(Integer), I Love
Photh(String), 3(Integer)
2591 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: select max(id)+1 from blog
2591 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters:
2607 [main] DEBUG java.sql.ResultSet  - <==    Columns: 1
2607 [main] DEBUG java.sql.ResultSet  - <==        Row: 8
2607 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: insert into blog(id, title,
author_id) values(?, ?, ?)
2607 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 8(Integer), I Love
Photh(String), 3(Integer)
2607 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: select max(id)+1 from blog
2607 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters:
2623 [main] DEBUG java.sql.ResultSet  - <==    Columns: 1
2623 [main] DEBUG java.sql.ResultSet  - <==        Row: 9
2623 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: insert into blog(id, title,
author_id) values(?, ?, ?)
2623 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 9(Integer), I Love
Photh(String), 3(Integer)
2701 [main] DEBUG java.sql.Connection  - xxx Connection Closed
2701 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - Returned connection
17007273 to pool.
```

默认地 MyBatis 不会自动提交,最后还需要使用 session.commit()来提交修改。

➢ 处理 NULL 值

像上面的例子，如果myBlog.setTitle(null)情况会怎样呢？那么程序会报错：

```
org.apache.ibatis.exceptions.PersistenceException:
### Error updating database.  Cause: org.apache.ibatis.type.TypeException: Error setting null parameter.  Most JDBC drivers require that the JdbcType must be specified for all nullable parameters. Cause: java.sql.SQLDataException: 尝试从类型“OTHER”的数据值获取类型“VARCHAR”的数据值。
### The error may involve org.mybatis.model.BlogMapper.insertBlog_user_autokey-Inline
### The error occurred while setting parameters
```

➔ Note:如果传递了一个空值，那这个JDBC Type 对于所有JDBC 允许为空的列来说是必须的。您可以研读一下关于PreparedStatement.setNull()的JavaDocs 文档。

解决这个问题就需要在参数中指定 jdbcType 属性，这个属性只在 insert,update 或 delete 的时候针对允许空的列有用。如：

```
    <insert id="insertBlog_user_autokey" statementType="PREPARED" parameterType="Blog" flushCache="true">
     <selectKey keyProperty="id" resultType="int" order="BEFORE">
     select max(id)+1 from blog
     </selectKey>
        insert into blog(id, title, author_id) values(#{id}, #{title,jdbcType=VARCHAR}, #{authorId})
    </insert>
```

然后我们使用 ij 工具查看一下插入的数据，title 值被设置为 NULL：

```
ij 版本 10.5
ij> connect 'jdbc:derby:blogDB;user=lory';
ij> select * from blog;
ID         |TITLE                          |AUTHOR_ID
-----------------------------------------------------
1          |nothing title                  |1
2          |just funny                     |2
3          |My Blog                        |3
4          |My Blog                        |4
5          |hello one                      |5
6          |hello two                      |6
7          |I Love Photh                   |3
8          |I Love Photh                   |3
9          |I Love Photh too               |3
10         |NULL                           |3
11         |NULL                           |3
12         |NULL                           |3
```

```
已选择 12 行
ij>
```

- 使用接口映射类

对给定的映射语句，使用一个正确描述参数与返回值的接口（如 BlogMapper.class），您就能更清晰地执行类型安全的代码，从而避免错误和异常。下在是使用接口的例子：

定义接口映射类 BlogMapper.class：

```
package org.mybatis.model;

public interface BlogMapper
{
    //<select id="selectBlog_by_id">
        public Blog selectBlog_by_id(int id);

        //<select id="selectBlog_by_bean">
        public Blog selectBlog_by_bean(Blog blog);

        //<update id="updateBlog_use_bean">
        public void updateBlog_use_bean(Blog blog);

        //<insert id="insertBlog_user_bean">
        public void insertBlog_user_bean(Blog blog);

        //<insert id="insertBlog_user_autokey">
        public void insertBlog_user_autokey(Blog blog);
}
```

调用代码：

```
        /**
         * user mapper interface
         */
        public static void userMapper(SqlSession session)
        {
                Blog blog = new Blog();
                blog.setTitle("nothing title");
                blog.setId(1);
                blog.setAuthorId(1);

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

```
            Blog blog1 = blogMapper.selectBlog_by_id(1);
            pringBlog(blog1);

            blogMapper.updateBlog_use_bean(blog);

            blog1 = blogMapper.selectBlog_by_id(1);
            pringBlog(blog1);
        }
```

控制台输出结果：

```
D:\Programs\workspace\MyBatis3
0 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - PooledDataSource
forcefully closed/removed all connections.
15 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - PooledDataSource
forcefully closed/removed all connections.
15 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - PooledDataSource
forcefully closed/removed all connections.
15 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - PooledDataSource
forcefully closed/removed all connections.
1217 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - Created connection
20474136.
1233 [main] DEBUG java.sql.Connection  - ooo Connection Opened
1640 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: select * from Blog where id =
 ?
1640 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 1(Integer)
1796 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, TITLE, AUTHOR_ID
1796 [main] DEBUG java.sql.ResultSet  - <==        Row: 1, just fun, 1
ID:1
title:just fun
authorID:null
1842 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: update blog set title= ?,
author_id=? where id = ?
1842 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: nothing title(String),
1(Integer), 1(Integer)
1920 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: select * from Blog where id =
 ?
1920 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 1(Integer)
1920 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, TITLE, AUTHOR_ID
1920 [main] DEBUG java.sql.ResultSet  - <==        Row: 1, nothing title, 1
ID:1
title:nothing title
authorID:null
1920 [main] DEBUG java.sql.Connection  - xxx Connection Closed
1920 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - Returned connection
20474136 to pool.
```

要注意的是， BlogMapper.class 是 XML 映射配置文件 BlogMapper.xml 里命名空间指向的接口：<mapper namespace="*org.mybatis.model.**BlogMapper***">

一般来说，一个 XML 映射配置文件对应一个命名空间，而这个命名空间又对应一个接口，这样我们就可以定义这样一个接口，并类型安全地执行 SQL 语句，如：

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

Blog blog1 = blogMapper.selectBlog_by_id(1);
```

MyBatis 启动时首先会根据 XML 配置文件中的命名空间查找是否存在这个接口类，如果存在则保存在一个名为 knownMappers 的 HashSet 中。当我们使用 session 的方法：

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

的时候，如果 type 是已知的接口类(knownMappers.contains(type)==true)，OK，继续执行，如果不是已知接口类(knownMappers.contains(type)=false)，则抛出如下异常：

```
org.apache.ibatis.binding.BindingException: Type interface org.mybatis.model.BlogMapperr is not known to the MapperRegistry.
```

- 使用 Constructor 元素

接下来的几节都是 resultMap 元素下的子元素的应用例子，为了简单，不把所有子元素都应用在一起，而是一个子元素一个子元素地进行学习使用。

使用 Constructor 元素是将数据库查询的结果通过构造器注入到结果映射类（JavaBean）中，可以理解为 spring 中的构造器注入。实际上在一个映射结果类中很可能没有构造器，或者在一个 resultMap 中既有构造器映射，又有指定属性映射，会造成配置不统一，反而不清晰，因此这个元素可能会很少使用。

首先在 Blog 类中增加两个构造函数：

```
    public Blog()
    {

    }
    public Blog(Integer id, String title, Integer authorId)
    {
        this.id = id;
        this.title = title;
```

```
        this.authorId = authorId;
    }
```

BlogMapper.xml 进行如下配置:

```
<?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.model.BlogMapper">
  <resultMap id="blogResultMap" type="Blog">
   <constructor>
       <idArg column="id" javaType="int"/>
       <arg column="title" javaType="String"/>
       <arg column="author_id" javaType="int"/>
</constructor>

  </resultMap>

    <select id="selectBlog_use_constructor"  resultMap="blogResultMap">
        select  id , title, author_id from Blog where id = #{id}
    </select>
</mapper>
```

调用代码是:

```
Blog blog = (Blog)session.selectOne("selectBlog_use_constructor", 3);
pringBlog(blog);
```

控制台输出:

```
3089 [main] DEBUG java.sql.Connection  - ooo Connection Opened
4494 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: select id , title, author_id
from Blog where id = ?
4494 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 3(Integer)
4667 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, TITLE, AUTHOR_ID
4667 [main] DEBUG java.sql.ResultSet  - <==        Row: 3, My Blog, 3
ID:3
title:My Blog
authorID:3
4729 [main] DEBUG java.sql.Connection  - xxx Connection Closed
4729 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - Returned connection
```

```
15558189 to pool.
```

➢ 使用 Association 元素

一个作者有一个博客，这是种“has-a”的一对一关系，现在我们使用 Association 元素把博客及作者的信息查询出来。这里重现官方教程的例子：

修改 Blog 的域模型：

```
package org.mybatis.model;

public class Blog {
     private Integer id;

    private String title;

     private Author author; //将原来的authorId换成author

     //省略get和set方法
}
```

BlogMapper.xml 配置(1)：

```
<?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.model.BlogMapper">
  <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>

   <select id="selectBlog_use_association" parameterType="int" 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
    from Blog B left outer join Author A on B.author_id = A.id
    where B.id = #{id}
  </select>

</mapper>
```

BlogMapper.xml 配置(2):

```
<?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.model.BlogMapper">

       <resultMap id="blogResult" type="Blog">
       <id property="id" column="blog_id" />
       <result property="title" column="blog_title"/>
       <association property="author" column="blog_author_id" javaType="Author"
            resultMap="authorResult"/>
       </resultMap>

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


   <select id="selectBlog_use_association" parameterType="int" 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
    from Blog B left outer join Author A on B.author_id = A.id
    where B.id = #{id}
  </select>
```

```
</mapper>
```

BlogMapper.xml 配置(3)：

```
<?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.model.BlogMapper">

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

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

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

</mapper>
```

调用代码：

```
Blog blog = (Blog)session.selectOne("selectBlog_use_association", 3);
printBlogAuthor(blog);

public static void printBlogAuthor(Blog blog)
{
        System.out.println("ID:" + blog.getId());
        System.out.println("title:" + blog.getTitle());
        System.out.println("authorID:" + blog.getAuthor().getId());
        System.out.println("authorName:" + blog.getAuthor().getUsername());
        System.out.println("authorPassword:" + blog.getAuthor().getPassword());
        System.out.println("authorEmail:" + blog.getAuthor().getEmail());
        System.out.println("authorBio:" + blog.getAuthor().getBio());
}
```

控制台输出信息：

```
ID:3
title:My Blog
```

```
authorID:3
authorName:user3
authorPassword:user3
authorEmail:user3@163.com
authorBio:guy
```

BlogMapper.xml 配置(1)和 BlogMapper.xml 配置(2)的配置是等效的，谁优谁劣见仁见智，或者看具体的项目要求。BlogMapper.xml 配置(3)使用了两条查询语句，加载博客信息后再加载作者信息。

对于配置(3)，如果"selectBlog_use_association"查询返回 N 条博客记录，如修改一下 SQL 语句：

```
<select id="selectBlog_use_association" resultMap="blogResult">
      <![CDATA[SELECT * FROM BLOG WHERE ID > 0 and ID < 7]]>
</select>
```

注：如果 SQL 语句有特殊符号，需要用<![CDATA[]]>括起来。这里的小于号<被认为是特殊符号，如果不用<![CDATA[]]>括起来是执行不了的。而调用代码也相应的改变为：

```
List<Blog> blogList = (List<Blog>)session.selectList("selectBlog_use_association");
printBlogAuthorList(blogList);

public static void printBlogAuthorList(List<Blog> blogList)
{
    for (Blog blog : blogList)
    {
        System.out.println("==========================");
        System.out.println("ID:" + blog.getId());
        System.out.println("title:" + blog.getTitle());
        System.out.println("authorID:" + blog.getAuthor().getId());
        System.out.println("authorName:" + blog.getAuthor().getUsername());
        System.out.println("authorPassword:" + blog.getAuthor().getPassword());
        System.out.println("authorEmail:" + blog.getAuthor().getEmail());
        System.out.println("authorBio:" + blog.getAuthor().getBio());
    }
}
```

控制台输出结果：

```
3510 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - Created connection
22377952.
3510 [main] DEBUG java.sql.Connection  - ooo Connection Opened
4931 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: SELECT * FROM BLOG WHERE ID >
0 and ID < 7
4946 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters:
```

```
5149 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, TITLE, AUTHOR_ID
5149 [main] DEBUG java.sql.ResultSet  - <==        Row: 1, nothing title, 1
5227 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: SELECT * FROM AUTHOR WHERE ID
= ?
5227 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 1(Integer)
5227 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, USERNAME, PASSWORD, EMAIL, BIO
5227 [main] DEBUG java.sql.ResultSet  - <==        Row: 1, user1, user1, user1@163.com, guy
5258 [main] DEBUG java.sql.ResultSet  - <==        Row: 2, just funny, 2
5258 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: SELECT * FROM AUTHOR WHERE ID
= ?
5258 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 2(Integer)
5258 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, USERNAME, PASSWORD, EMAIL, BIO
5258 [main] DEBUG java.sql.ResultSet  - <==        Row: 2, user2, user2, user2@163.com, guy
5274 [main] DEBUG java.sql.ResultSet  - <==        Row: 3, My Blog, 3
5274 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: SELECT * FROM AUTHOR WHERE ID
= ?
5274 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 3(Integer)
5274 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, USERNAME, PASSWORD, EMAIL, BIO
5274 [main] DEBUG java.sql.ResultSet  - <==        Row: 3, user3, user3, user3@163.com, guy
5274 [main] DEBUG java.sql.ResultSet  - <==        Row: 4, My Blog, 4
5274 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: SELECT * FROM AUTHOR WHERE ID
= ?
5274 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 4(Integer)
5289 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, USERNAME, PASSWORD, EMAIL, BIO
5289 [main] DEBUG java.sql.ResultSet  - <==        Row: 4, user4, user4, user4@163.com, guy
5289 [main] DEBUG java.sql.ResultSet  - <==        Row: 5, hello one, 5
5289 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: SELECT * FROM AUTHOR WHERE ID
= ?
5289 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 5(Integer)
5289 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, USERNAME, PASSWORD, EMAIL, BIO
5289 [main] DEBUG java.sql.ResultSet  - <==        Row: 5, user5, user5, user5@163.com, guy
5289 [main] DEBUG java.sql.ResultSet  - <==        Row: 6, hello two, 6
5289 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: SELECT * FROM AUTHOR WHERE ID
= ?
5289 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 6(Integer)
5289 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, USERNAME, PASSWORD, EMAIL, BIO
5289 [main] DEBUG java.sql.ResultSet  - <==        Row: 6, user6, user6, user6@163.com, guy
==========================
ID:1
title:nothing title
authorID:1
authorName:user1
authorPassword:user1
authorEmail:user1@163.com
authorBio:guy
==========================
ID:2
title:just funny
authorID:2
```

```
authorName:user2
authorPassword:user2
authorEmail:user2@163.com
authorBio:guy
==========================
ID:3
title:My Blog
authorID:3
authorName:user3
authorPassword:user3
authorEmail:user3@163.com
authorBio:guy
==========================
ID:4
title:My Blog
authorID:4
authorName:user4
authorPassword:user4
authorEmail:user4@163.com
authorBio:guy
==========================
ID:5
title:hello one
authorID:5
authorName:user5
authorPassword:user5
authorEmail:user5@163.com
authorBio:guy
==========================
ID:6
title:hello two
authorID:6
authorName:user6
authorPassword:user6
authorEmail:user6@163.com
authorBio:guy
5305 [main] DEBUG java.sql.Connection  - xxx Connection Closed
```

修改后的结果是"selectAuthor"的 SQL 语句会被执行 6 次来加载每个博客的作者信息，这样就产生 N+1 问题。但恰恰，配置(1)或配置(2)就是解决这个 N+1 问题的一种方案，这叫联合嵌套结果集（Nested Results for Association）。

另外一种方案就是使用延迟加载，在 SQL 配置文件中设置：

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

再次执行的结果是：

```
3371 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - Created connection
701508.
3386 [main] DEBUG java.sql.Connection  - ooo Connection Opened
4745 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: SELECT * FROM BLOG WHERE ID >
0 and ID < 7
4745 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters:
4886 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, TITLE, AUTHOR_ID
4886 [main] DEBUG java.sql.ResultSet  - <==        Row: 1, nothing title, 1
5120 [main] DEBUG java.sql.ResultSet  - <==        Row: 2, just funny, 2
5120 [main] DEBUG java.sql.ResultSet  - <==        Row: 3, My Blog, 3
5120 [main] DEBUG java.sql.ResultSet  - <==        Row: 4, My Blog, 4
5136 [main] DEBUG java.sql.ResultSet  - <==        Row: 5, hello one, 5
5136 [main] DEBUG java.sql.ResultSet  - <==        Row: 6, hello two, 6
============================
5214 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: SELECT * FROM AUTHOR WHERE ID
= ?
5214 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 1(Integer)
5214 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, USERNAME, PASSWORD, EMAIL, BIO
5214 [main] DEBUG java.sql.ResultSet  - <==        Row: 1, user1, user1, user1@163.com, guy
ID:1
title:nothing title
authorID:1
authorName:user1
authorPassword:user1
authorEmail:user1@163.com
authorBio:guy
============================
5307 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: SELECT * FROM AUTHOR WHERE ID
= ?
5307 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 2(Integer)
5323 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, USERNAME, PASSWORD, EMAIL, BIO
5323 [main] DEBUG java.sql.ResultSet  - <==        Row: 2, user2, user2, user2@163.com, guy
ID:2
title:just funny
authorID:2
authorName:user2
authorPassword:user2
authorEmail:user2@163.com
authorBio:guy
============================
5323 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: SELECT * FROM AUTHOR WHERE ID
= ?
5323 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 3(Integer)
5323 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, USERNAME, PASSWORD, EMAIL, BIO
5323 [main] DEBUG java.sql.ResultSet  - <==        Row: 3, user3, user3, user3@163.com, guy
ID:3
title:My Blog
```

```
authorID:3
authorName:user3
authorPassword:user3
authorEmail:user3@163.com
authorBio:guy
==========================
5323 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: SELECT * FROM AUTHOR WHERE ID
= ?
5338 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 4(Integer)
5338 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, USERNAME, PASSWORD, EMAIL, BIO
5338 [main] DEBUG java.sql.ResultSet  - <==        Row: 4, user4, user4, user4@163.com, guy
ID:4
title:My Blog
authorID:4
authorName:user4
authorPassword:user4
authorEmail:user4@163.com
authorBio:guy
==========================
5338 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: SELECT * FROM AUTHOR WHERE ID
= ?
5338 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 5(Integer)
5338 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, USERNAME, PASSWORD, EMAIL, BIO
5338 [main] DEBUG java.sql.ResultSet  - <==        Row: 5, user5, user5, user5@163.com, guy
ID:5
title:hello one
authorID:5
authorName:user5
authorPassword:user5
authorEmail:user5@163.com
authorBio:guy
==========================
5338 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: SELECT * FROM AUTHOR WHERE ID
= ?
5338 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 6(Integer)
5354 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, USERNAME, PASSWORD, EMAIL, BIO
5354 [main] DEBUG java.sql.ResultSet  - <==        Row: 6, user6, user6, user6@163.com, guy
ID:6
title:hello two
authorID:6
authorName:user6
authorPassword:user6
authorEmail:user6@163.com
authorBio:guy
5354 [main] DEBUG java.sql.Connection  - xxx Connection Closed
```

MyBatis 首先只是加载了 Blog 信息，Author 信息并没有加载。但我们执行下面的方法：

```
printBlogAuthorList(blogList);
```

Author 信息才被加载进来，并且是处理一条才加载一条。正如官方指南说的：MyBatis 可以使用延迟加载这些查询，因此这些查询立马可节省开销。然而，如果您加载一个列表后立即迭代访问嵌套的数据，这将会调用所有的延迟加载，因此性能会变得非常糟糕。

一般地，获得一个数据库列表后，都需要进行迭代处理，要不然获得数据就没有太大意义了，因此使用延迟加载在总的时间上并没有节省。

因此，在“has-a”一对一情况下，使用联合嵌套结果集是较好的做法，但可能会产生能大量冗余的、重复的数据。

- 使用 Collection 元素

Collection 元素用来处理“一对多”的数据模型，例如，一个博客有许多文章（Posts）。在博客类里，应该有一个文章的列表，其属性定义如下：

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

这个例子，查询 id 排在前 3 位的博客，并列出这 3 个博客的所有文章。

重新修改 Blog 域模型：

```
package org.mybatis.model;

public class Blog {
     private Integer id;

    private String title;

     private Integer authorId;

     private List<Post> posts;

     //省略get和set方法

}
```

BlogMapper.xml 配置(1)，使用集合嵌套选择（Nested Select for Collection）

```
<?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.model.BlogMapper">
       <resultMap id="blogResult" type="Blog">
           <id property="id" column="id"/>
           <result property="title" column="title"/>
           <result property="authorId" column="authorid"/>
              <collection property="posts" javaType="ArrayList" column="id"
                     ofType="Post" select="selectPostsForBlog"/>
       </resultMap>

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


       <select id="selectBlog_use_collection" resultMap="blogResult">
              <![CDATA[SELECT id , title, author_id as authorid FROM BLOG WHERE ID > 0 and ID
< 4]]>
       </select>

</mapper>
```

BlogMapper.xml配置(2)，使用集合的嵌套结果集（Nested Results for Collection）

```
<?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.model.BlogMapper">
       <resultMap id="blogResult" type="Blog">
         <id property="id" column="blog_id" />
         <result property="title" column="blog_title"/>
         <result property="authorId" column="authorid"/>
         <collection property="posts" ofType="Post">
           <id property="id" column="post_id"/>
           <result property="subject" column="post_subject"/>
           <result property="section" column="post_section"/>
           <result property="body" column="post_body"/>
         </collection>
       </resultMap>

       <select id="selectBlog_use_collection"  resultMap="blogResult">
        <![CDATA[
              select
                  B.id as blog_id,
```

```
                B.title as blog_title,
                B.author_id as authorid,
                P.id as post_id,
                P.subject as post_subject,
                P.section as post_section,
                P.body as post_body
            from Blog B
            left outer join Post P on B.id = P.blog_id
            where B.id > 0 and B.id < 4]]>
        </select>
</mapper>
```

调用代码：

```
List<Blog> blogList = (List<Blog>)session.selectList("selectBlog_use_collection");
printBlogPosts(blogList);



public static void printBlogPosts(List<Blog> blogList)
{
        for (Blog blog : blogList)
        {
                System.out.println("\n=========================");
                System.out.println("ID:" + blog.getId());
                System.out.println("blog_title:" + blog.getTitle());
                System.out.println("authorID:" + blog.getAuthorId());
                System.out.println("==========posts============");

                for (Post post : blog.getPosts())
                {
                        System.out.println("subject:" + post.getSubject());
                        System.out.println("section:" + post.getSection());
                        System.out.println("body:" + post.getBody());
                }
        }
}
```

BlogMapper.xml 配置(1)和 BlogMapper.xml 配置(2)打印的结果都是一样的，只是查询的方式不一样：

```
3340 [main] DEBUG java.sql.Connection  - ooo Connection Opened
4620 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: SELECT id , title, author_id
as authorid FROM BLOG WHERE ID > 0 and ID < 4
```

```
4620 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters:
4777 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, TITLE, AUTHORID
4777 [main] DEBUG java.sql.ResultSet  - <==        Row: 1, nothing title, 1
4995 [main] DEBUG java.sql.ResultSet  - <==        Row: 2, just funny, 2
4995 [main] DEBUG java.sql.ResultSet  - <==        Row: 3, My Blog, 3

==========================
5058 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: SELECT * FROM POST WHERE
BLOG_ID = ?
5058 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 1(Integer)
5058 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, BLOG_ID, AUTHOR_ID, CREATED_ON,
SECTION, SUBJECT, BODY
5058 [main] DEBUG java.sql.ResultSet  - <==        Row: 1, 1, 1, 2010-08-04, photo, ddd,
nothing
5182 [main] DEBUG java.sql.ResultSet  - <==        Row: 2, 1, 1, 2010-08-05, photo, df,
nothing too
5182 [main] DEBUG java.sql.ResultSet  - <==        Row: 3, 1, 1, 2010-08-06, photo, ddfdfdd,
also nothing
ID:1
blog_title:nothing title
authorID:1
==========posts=============
subject:ddd
section:photo
body:nothing
subject:df
section:photo
body:nothing too
subject:ddfdfdd
section:photo
body:also nothing

==========================
5182 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: SELECT * FROM POST WHERE
BLOG_ID = ?
5182 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 2(Integer)
5198 [main] DEBUG java.sql.ResultSet  - <==    Columns: ID, BLOG_ID, AUTHOR_ID, CREATED_ON,
SECTION, SUBJECT, BODY
5198 [main] DEBUG java.sql.ResultSet  - <==        Row: 4, 2, 2, 2010-08-06, photo, fd,
nothing more
ID:2
blog_title:just funny
authorID:2
==========posts=============
subject:fd
section:photo
body:nothing more

==========================
```

```
5198 [main] DEBUG java.sql.PreparedStatement  - ==>  Executing: SELECT * FROM POST WHERE
BLOG_ID = ?
5198 [main] DEBUG java.sql.PreparedStatement  - ==> Parameters: 3(Integer)
ID:3
blog_title:My Blog
authorID:3
==========posts============
5198 [main] DEBUG java.sql.Connection  - xxx Connection Closed
5198 [main] DEBUG org.apache.ibatis.datasource.pooled.PooledDataSource  - Returned connection
3403998 to pool.
```

从输出结果看，可以知道是使用了延迟加载。对于 collection 元素的学习暂时到这里。如果一篇文章还有 N 个评论，那还可以继续嵌套下去，嵌套的深度是没有限制的，但要考虑一下性能的因素。

附录 4 XML 中的特殊字符

如果 MyBatis 使用 XML 配置，那不可避免地会遇到一些对 XML 来说是特殊的字符。如小于号“<”，因为 XML 解析器会认为是一个新元素的开始，因此要进行转义。这里有两个方法：

- 使用转义实体

下面是五个在 XML 文档中预定义好的转义实体:

< → < 小于号

> → > 大于号

& → & 和

' → ’ 单引号

" → ″ 双引号

如果是小于等于“<=”，其转义实体为：<=

同理，大小等于“>=”，其转义实体为：>=

- 使用 CDATA 部件

一个 CDATA 部件以"<![CDATA[" 标记开始，以"]]>"标记结束。在"<![CDATA["和"]]>"之间的特殊字符的意义都不起作用，而转变为普通字符串内容。

一般地，在 MyBatis 的 XML 映射语句配置文件中，如果 SQL 语句有特殊字符，那么使用 CDTA 部件括起来，如：

```
<select id="selectBlog_use_collection" resultMap="blogResult">
   <![CDATA[
         SELECT id , title, author_id as authored
         FROM BLOG
         WHERE ID > 0 and ID < 10
        ]]>
 </select>
```

当然使用转义实体也行：

```
<select id="selectBlog_use_collection" resultMap="blogResult">
       SELECT id , title, author_id as authorid
       FROM BLOG
       WHERE ID &gt; 0 and ID &lt; 10
</select>
```

而在动态 SQL 各元素的测试语句中，在元素的属性中不能再嵌套其它元素或包含 CDATA 部件，因此只能使用转义实体，如：

```
<select id="selectAuthor_use_where" parameterType="Blog" resultType="Author">
     select * from author
     <where>
        <if test="authorId != null
                    and authorId &gt;= 1
                    and authorId &lt;= 5">
            id = #{authorId}
        </if>
      </where>
</select>
```