MyBatis 3

用户指南

从文档中复制代码的警告

对，这不是一个法律上的警告，但是它可以帮助你保持清醒的认识。从美学上来讲，现代的文字处理工具在制作可读性强和格式良好的文本上做了大量的工作。然而，它们也往往会由于插入特殊字符而完全破坏代码示例，有时看起来和你想要的是一模一样的。"引号"和连字符就是一个很好的例子-在 IDE 环境或文本编辑器中，左边的那个符号就不会正常起作用，至少不会是你想要的那个效果。

阅读本文档，就要享受它，希望它能对你有帮助。当遇到代码段示例的时候，可以寻找示例和下载（包括单元测试等），或是来自网站和邮件列表的示例。

帮助我们把文档做得更好...

如果你发现了本文档的遗漏之处，或者丢失 MyBatis 特性的说明时，那么最好的方法就是了解一下这个遗漏之处然后把它记录下来。

我们在 wiki 接收公共的文档贡献：

http://code.google.com/p/mybatis/wiki/Welcome

你也是本文档的最佳作者，其他用户也会来阅读它的。

关于翻译

MyBatis 3 的用户指南翻译由南磊完成，若对翻译质量有任何意见和建议，请联系 nanlei1987@gmail.com，愿和大家共同提高，共同进步。

目录

什么是 MyBatis？

MyBatis 是支持普通 SQL 查询，存储过程和高级映射的优秀持久层框架。MyBatis 消除了几乎所有的 JDBC 代码和参数的手工设置以及结果集的检索。MyBatis 使用简单的 XML 或注解用于配置和原始映射，将接口和 Java 的 POJOs（Plan Old Java Objects，普通的 Java 对象）映射成数据库中的记录。

入门

每一个 MyBatis 的应用程序都以一个 SqlSessionFactory 对象的实例为核心。SqlSessionFactory 对象的实例可以通过 SqlSessionFactoryBuilder 对象来获得。SqlSessionFactoryBuilder 对象可以从 XML 配置文件，或从 Configuration 类的习惯准备的实例中构建 SqlSessionFactory 对象。

从 XML 中构建 SqlSessionFactory

从 XML 文件中构建 SqlSessionFactory 的实例非常简单。这里建议你使用类路径下的资源文件来配置，但是你可以使用任意的 Reader 实例，这个实例包括由文字形式的文件路径或 URL 形式的文件路径 file://来创建。MyBatis 包含了一些工具类，称作为资源，这些工具类包含一些方法，这些方法使得从类路径或其他位置加载资源文件更加简单。

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

XML 配置文件包含对 MyBatis 系统的核心设置，包含获取数据库连接实例的数据源和决定事务范围和控制的事务管理器。关于 XML 配置文件的详细内容可以在文档后面找到，这里给出一个简单的示例：

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

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

当然，在XML配置文件中还有很多可以配置的，上面的示例指出的则是最关键的部分。要注意 XML 头部的声明，需要用来验证 XML 文档正确性。environment 元素体中包含对事务管理和连接池的环境配置。mappers 元素是包含所有 mapper（映射器）的列表，这些 mapper 的 XML 文件包含 SQL 代码和映射定义信息。

不使用 XML 构建 SqlSessionFactory

如果你喜欢从 Java 程序而不是 XML 文件中直接创建配置实例，或创建你自己的配置构建器，MyBatis 也提供完整的配置类，提供所有从 XML 文件中加载配置信息的选项。

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

注意这种情况下配置是添加映射类。映射类是 Java 类，这些类包含 SQL 映射语句的注解从而避免了 XML 文件的依赖，XML 映射仍然在大多数高级映射（比如：嵌套 Join 映射）时需要。出于这样的原因，如果存在 XML 配置文件的话，MyBatis 将会自动查找和加载一个对等的 XML 文件（这种情况下，基于类路径下的 BlogMapper.class 类的类名，那么 BlogMapper.xml 将会被加载）。后面我们会了解更多。

从 SqlSessionFactory 中获取 SqlSession

现在，我们已经知道如何获取 SqlSessionFactory 对象了，基于同样的启示，我们就可以获得 SqlSession 的实例了。SqlSession 对象完全包含以数据库为背景的所有执行 SQL 操作的方法。你可以用 SqlSession 实例来直接执行已映射的 SQL 语句。例如：

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

这种方法起到的作用，和我们使用之前的 MyBatis 版本是相似的，现在有一种更简洁的方法。使用合理描述参数和 SQL 语句返回值的接口（比如 BlogMapper.class），这样现在就可以至此那个更简单，更安全的代码，没有容易发生的字符串文字和转换的错误。

例如：

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

现在我们来探究一下这里到底执行了什么。

探究已映射的 SQL 语句

这里你也许想知道通过 SqlSession 和 Mapper 对象到底执行了什么操作。已映射的 SQL 语句是一个很大的主题，而且这个主题会贯穿本文档的大部分内容。为了给出一个宏观的概念，这里有一些示例。

上面提到的任何一个示例，语句是通过 XML 或注解定义的。我们先来看看 XML。使用基于 XML 的映射语言，在过去的几年中使得 MyBatis 非常流行，他为 MyBatis 提供所有的特性设置。如果你以前用过 MyBatis，这个概念应该很熟悉了，但是 XML 映射文件也有很多的改进，后面我们会详细来说。这里给出一个基于 XML 映射语句的示例，这些语句应该可以满足上述示例中 SqlSession 对象的调用。

```
<?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>
```

这个简单的例子中看起来有很多额外的东西，但是也相当简洁了。你可以在一个单独的 XML 映射文件中定义很多的映射语句，除 XML 头部和文档类型声明之外，你可以得到很多方便之处。在文件的剩余部分是很好的自我解释。在命名空间“com.mybatis.example.BlogMapper”中，它定义了一个名为“selectBlog”的映射语句，这样它允许你使用完全限定名“org.mybatis.example.BlogMapper.selectBlog”来调用映射语句，我们下面示例中所有的写法也是这样的。

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

要注意这个使用完全限定名调用 Java 对象的方法是相似的，这样做是有原因的。这个命名可以直接给相同命名空间下的的映射类，使用一个名称，参数和返回值和已映射的查询语句都一样的方法即可。这就允许你非常容易地调用映射器接口中的方法，这和你前面看到的是一样的，下面这个示例中它又出现了。

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

第二种方式有很多有点，首先它不是基于文字的，那就更安全了。第二，如果你的 IDE 有代码补全功能，那么你可以利用它来操纵已映射的 SQL 语句。第三，不需要强制类型转

换，同时 BlogMapper 接口可以保持简洁，返回值类型很安全（参数类型也很安全）。

命名空间的一点注释

命名空间在之前版本的 MyBatis 中是可选项，非常混乱也没有帮助。现在，命名空间是必须的，而且有一个目的，它使用更长的完全限定名来隔离语句。

命名空间使得接口绑定成为可能，就像你看到的那样，如果之前不了解，那么现在你就会使用它们了，你应该按照下面给出示例的来练习，以免改变自己的想法。使用命名空间，并将它放在合适的 Java 包空间之下，将会使你的代码变得简洁，在很长的时间内提高 MyBatis 的作用。

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

- 直接查找完全限定名（比如“com.mypackage.MyMapper.selectAllThings”），如果发现就使用。
- 短名称（比如“selectAllThings”）可以用来引用任意含糊的对象。而如果有两个或两个以上的（比如“com.foo.selectAllThings”和“com.bar.selectAllThings”），那么就会得到错误报告，说短名称是含糊的，因此就必须使用完全限定名。

如 BlogMapper 这样的映射器类来说，还有一个妙招。它们中间映射的语句可以不需要在 XML 中来写，而可以使用 Java 注解来替换。比如，上面的 XML 示例可以如下来替换：

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

对于简单语句来说，使用注解代码会更加清晰，然而 Java 注解对于复杂语句来说就会混乱，应该限制使用。因此，如果你不得不做复杂的事情，那么最好使用 XML 来映射语句。

当然这也取决于你和你的项目团队的决定，看哪种更适合你来使用，还有以长久方式来使用映射语句的重要性。也就是说，不要将自己局限在一种方式中。你可以轻松地将注解换成 XML 映射语句，反之亦然。

范围和生命周期

理解我们目前已经讨论过的不同范围和生命周期类是很重要的。不正确的使用它们会导致严重的并发问题。

SqlSessionFactoryBuilder

这个类可以被实例化，使用和丢弃。一旦你创建了 SqlSessionFactory 后，这个类就不需要存在了。因此 SqlSessionFactoryBuilder 实例的最佳范围是方法范围（也就是本地方法变量）。你可以重用 SqlSessionFactoryBuilder 来创建多个 SqlSessionFactory 实例，但是最好的方式是不需要保持它一直存在来保证所有 XML 解析资源，因为还有更重要的事情要做。

SqlSessionFactory

一旦被创建，SqlSessionFactory 应该在你的应用执行期间都存在。没有理由来处理或重新创建它。使用 SqlSessionFactory 的最佳实践是在应用运行期间不要重复创建多次。这样的操作将被视为是非常糟糕的。因此 SqlSessionFactory 的最佳范围是应用范围。有很多方法可以做到，最简单的就是使用单例模式或者静态单例模式。然而这两种方法都不认为是最佳实践。这样的话，你可以考虑依赖注入容器，比如 Google Guice 或 Spring。这样的框架允许你创建支持程序来管理单例 SqlSessionFactory 的生命周期。

SqlSession

每个线程都应该有它自己的 SqlSession 实例。SqlSession 的实例不能被共享，也是线程不安全的。因此最佳的范围是请求或方法范围。绝对不能将 SqlSession 实例的引用放在一个类的静态字段甚至是实例字段中。也绝不能将 SqlSession 实例的引用放在任何类型的管理范围中，比如 Serlvet 架构中的 HttpSession。如果你现在正用任意的 Web 框架，要考虑 SqlSession 放在一个和 HTTP 请求对象相似的范围内。换句话说，基于收到的 HTTP 请求，你可以打开了一个 SqlSession，然后返回响应，就可以关闭它了。关闭 Session 很重要，你应该确保使用 finally 块来关闭它。下面的示例就是一个确保 SqlSession 关闭的基本模式：

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

在你的代码中一贯地使用这种模式，将会保证所有数据库资源都正确地关闭（假设你没有通过你自己的连接关闭，这会给 MyBatis 造成一种迹象表明你要自己管理连接资源）。

Mapper 实例

映射器是你创建绑定映射语句的接口。映射器接口的实例可以从 SqlSession 中获得。那么从技术上来说，当被请求时，任意映射器实例的最宽范围和 SqlSession 是相同的。然而，映射器实例的最佳范围是方法范围。也就是说，它们应该在使用它们的方法中被请求，然后就抛弃掉。它们不需要明确地关闭，那么在请求对象中保留它们也就不是什么问题了，这和 SqlSession 相似。你也许会发现，在这个水平上管理太多的资源的话会失控。保持简单，将映射器放在方法范围内。下面的示例就展示了这个实践：

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

XML 映射配置文件

MyBatis 的 XML 配置文件包含了影响 MyBatis 行为甚深的设置和属性信息。XML 文档的高层级结构如下：

- configuration 配置
  - properties 属性
  - settings 设置
  - typeAliases 类型命名
  - typeHandlers 类型处理器
  - objectFactory 对象工厂
  - plugins 插件
  - environments 环境
    - environment 环境变量
      - transactionManager 事务管理器
      - dataSource 数据源
  - 映射器

properties

这些是外部化的，可替代的属性，这些属性也可以配置在典型的 Java 属性配置文件中，或者通过 properties 元素的子元素来传递。例如：

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

其中的属性就可以在整个配置文件中使用，使用可替换的属性来实现动态配置。比如：

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

这个例子中的 username 和 password 将会由 properties 元素中设置的值来替换。driver 和 url 属性将会从包含进来的 config.properties 文件中的值来替换。这里提供很多配置的选项。

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

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

如果在这些地方，属性多于一个的话，MyBatis 按照如下的顺序加载它们：

- 在 properties 元素体内指定的属性首先被读取。

- 从类路径下资源或 properties 元素的 url 属性中加载的属性第二被读取，它会覆盖已经存在的完全一样的属性。
- 作为方法参数传递的属性最后被读取，它也会覆盖任一已经存在的完全一样的属性，这些属性可能是从 properties 元素体内和资源/url 属性中加载的。

因此，最高优先级的属性是那些作为方法参数的，然后是资源/url 属性，最后是 properties 元素中指定的属性。

Settings

这些是极其重要的调整，它们会修改 MyBatis 在运行时的行为方式。下面这个表格描述了设置信息，它们的含义和默认值。

设置参数	描述	有效值	默认值
cacheEnabled	这个配置使全局的映射器启用或禁用缓存。	true | false	true
lazyLoadingEnabled	全局启用或禁用延迟加载。当禁用时，所有关联对象都会即时加载。	true | false	true
aggressiveLazyLoading	当启用时，有延迟加载属性的对象在被调用时将会完全加载任意属性。否则，每种属性将会按需要加载。	true | false	true
multipleResultSetsEnabled	允许或不允许多种结果集从一个单独的语句中返回（需要适合的驱动）。	true | false	true
useColumnLabel	使用列标签代替列名。不同的驱动在这方便表现不同。参考驱动文档或充分测试两种方法来决定所使用的驱动。	true | false	true
useGeneratedKeys	允许 JDBC 支持生成的键。需要适合的驱动。如果设置为 true 则这个设置强制生成的键被使用，尽管一些驱动拒绝兼容但仍然有效（比如 Derby）。	true | false	false
autoMappingBehavior	指定 MyBatis 如何自动映射列到字段/属性。PARTIAL 只会自动映射简单，没有嵌套的结果。FULL 会自动映射任意复杂的结果（嵌套的或其他情况）。	NONE, PARTIAL, FULL	PARTIAL
defaultExecutorType	配置默认的执行器。SIMPLE 执行器没有什么特别之处。REUSE 执行器重用预处理语句。BATCH 执行器重用语句和批量更新	SIMPLE, REUSE, BATCH	SIMPLE
defaultStatementTimeout	设置超时时间，它决定驱动等待一个数据库响应的时间。	Any positive integer	Not Set (null)

一个设置信息元素的示例，完全的配置如下所示：

```
<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

类型别名是为 Java 类型命名一个短的名字。它只和 XML 配置有关，只用来减少类完全限定名的多余部分。例如：

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

使用这个配置，“Blog”可以任意用来替代“domain.blog.Blog”所使用的地方。

对于普通的 Java 类型，有许多内建的类型别名。它们都是大小写不敏感的，由于重载的名字，要注意原生类型的特殊处理。

别名	映射的类型
_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

无论是 MyBatis 在预处理语句中设置一个参数，还是从结果集中取出一个值时，类型处理器被用来将获取的值以合适的方式转换成 Java 类型。下面这个表格描述了默认的类型处理器。

类型处理器	Java 类型	JDBC 类型
BooleanTypeHandler	Boolean，boolean	任何兼容的布尔值
ByteTypeHandler	Byte，byte	任何兼容的数字或字节类型
ShortTypeHandler	Short，short	任何兼容的数字或短整型
IntegerTypeHandler	Integer，int	任何兼容的数字和整型
LongTypeHandler	Long，long	任何兼容的数字或长整型
FloatTypeHandler	Float，float	任何兼容的数字或单精度浮点型
DoubleTypeHandler	Double，double	任何兼容的数字或双精度浮点型
BigDecimalTypeHandler	BigDecimal	任何兼容的数字或十进制小数类型
StringTypeHandler	String	CHAR 和 VARCHAR 类型
ClobTypeHandler	String	CLOB 和 LONGVARCHAR 类型
NStringTypeHandler	String	NVARCHAR 和 NCHAR 类型
NClobTypeHandler	String	NCLOB 类型
ByteArrayTypeHandler	byte[]	任何兼容的字节流类型
BlobTypeHandler	byte[]	BLOB 和 LONGVARBINARY 类型
DateTypeHandler	Date（java.util）	TIMESTAMP 类型
DateOnlyTypeHandler	Date（java.util）	DATE 类型
TimeOnlyTypeHandler	Date（java.util）	TIME 类型
SqlTimestampTypeHandler	Timestamp（java.sql）	TIMESTAMP 类型
SqlDateTypeHandler	Date（java.sql）	DATE 类型
SqlTimeTypeHandler	Time（java.sql）	TIME 类型
ObjectTypeHandler	Any	其他或未指定类型
EnumTypeHandler	Enumeration 类型	VARCHAR-任何兼容的字符串类型，作为代码存储（而不是索引）。

你可以重写类型处理器或创建你自己的类型处理器来处理不支持的或非标准的类型。要这样做的话，简单实现 TypeHandler 接口（org.mybatis.type），然后映射新的类型处理器类到 Java 类型，还有可选的一个 JDBC 类型。例如：

// ExampleTypeHandler.java

```
public class ExampleTypeHandler implements TypeHandler {
```

```
    public void setParameter(PreparedStatement ps, int i, Object
    parameter,JdbcType jdbcType) throws SQLException {
        ps.setString(i, (String) parameter);
    }
    public Object getResult(ResultSet rs, String columnName)
        throws SQLException {
        return rs.getString(columnName);
    }
    public Object getResult(CallableStatement cs, int columnIndex)
        throws SQLException {
        return cs.getString(columnIndex);
    }
}
// MapperConfig.xml
<typeHandlers>
    <typeHandler javaType="String" jdbcType="VARCHAR"
        handler="org.mybatis.example.ExampleTypeHandler"/>
</typeHandlers>
```

使用这样的类型处理器将会覆盖已经存在的处理 Java 的 String 类型属性和 VARCHAR 参数及结果的类型处理器。要注意 MyBatis 不会审视数据库元信息来决定使用哪种类型，所以你必须在参数和结果映射中指定那是 VARCHAR 类型的字段，来绑定到正确的类型处理器上。这是因为 MyBatis 直到语句被执行都不知道数据类型的这个现实导致的。

objectFactory

MyBatis 每次创建结果对象新的实例时，它使用一个 ObjectFactory 实例来完成。如果参数映射存在，默认的 ObjectFactory 不比使用默认构造方法或带参数的构造方法实例化目标类做的工作多。如果你想重写默认的 ObjectFactory，你可以创建你自己的。比如：

```
// ExampleObjectFactory.java
public class ExampleObjectFactory extends DefaultObjectFactory {
    public Object create(Class type) {
        return super.create(type);
    }
    public Object create(Class type,List<Class> constructorArgTypes,
        List<Object> constructorArgs) {
        return super.create(type, constructorArgTypes,
    constructorArgs);
    }
    public void setProperties(Properties properties) {
        super.setProperties(properties);
    }
}
// MapperConfig.xml
<objectFactory type="org.mybatis.example.ExampleObjectFactory">
```

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

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

plugins

MyBatis 允许你在某一点拦截已映射语句执行的调用。默认情况下，MyBatis 允许使用插件来拦截方法调用：

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

这些类中方法的详情可以通过查看每个方法的签名来发现，而且它们的源代码在 MyBatis 的发行包中有。你应该理解你覆盖方法的行为，假设你所做的要比监视调用要多。如果你尝试修改或覆盖一个给定的方法，你可能会打破 MyBatis 的核心。这是低层次的类和方法，要谨慎使用插件。

使用插件是它们提供的非常简单的力量。简单实现拦截器接口，要确定你想拦截的指定签名。

// ExamplePlugin.java

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

// MapperConfig.xml

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

```
</plugins>
```

上面的插件将会拦截在 Executor 实例中所有的“update”方法调用，它也是负责低层次映射语句执行的内部对象。

覆盖配置类

除了用插件来修改 MyBatis 核心行为之外，你也可以完全覆盖配置类。简单扩展它，然后覆盖其中的任意方法，之后传递它到 sqlSessionFactoryBuilder.build(myConfig)方法的调用。这可能会严重影响 MyBatis 的行为，所以要小心。

environments

MyBatis 可以配置多种环境。这会帮助你将 SQL 映射应用于多种数据库之中。例如，你也许为开发要设置不同的配置，测试和生产环境。或者你可能有多种生产级数据库却共享相同的模式，所以你会想对不同数据库使用相同的 SQL 映射。这种用例是很多的。

一个很重要的问题要记得：你可以配置多种环境，但你只能为每个 SqlSessionFactory 实例选择一个。

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

- **每个数据库对应一个 SqlSessionFactory**

为了明确创建哪种环境，你可以将它作为可选的参数传递给 SqlSessionFactoryBuilder。可以接受环境配置的两个方法签名是：

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

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

如果环境被忽略，那么默认环境将会被加载，如下进行：

SqlSessionFactory factory = sqlSessionFactoryBuilder.build(reader);

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

环境元素定义了如何配置环境。

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

注意这里的键：

- 默认的环境 ID（比如：default=”development”）。
- 每个 environment 元素定义的环境 ID（比如：id=”development”）。

- 事务管理器的配置（比如：type=”JDBC”）。
- 数据源的配置（比如：type=”POOLED”）。

默认的环境和环境 ID 是自我解释的。你可以使用你喜欢的名称来命名，只要确定默认的要匹配其中之一。

transactionManager

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

- **JDBC** – 这个配置直接简单使用了 JDBC 的提交和回滚设置。它依赖于从数据源得到的连接来管理事务范围。
- **MANAGED** – 这个配置几乎没做什么。它从来不提交或回滚一个连接。而它会让容器来管理事务的整个生命周期（比如 Spring 或 JEE 应用服务器的上下文）。默认情况下它会关闭连接。然而一些容器并不希望这样，因此如果你需要从连接中停止它，将 closeConnection 属性设置为 false。例如：

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

这两种事务管理器都不需要任何属性。然而它们都是类型别名，要替换使用它们，你需要放置将你自己的类的完全限定名或类型别名，它们引用了你对 TransacFactory 接口的实现类。

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

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

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

使用这两个接口，你可以完全自定义 MyBatis 对事务的处理。

dataSsource

dataSource 元素使用基本的 JDBC 数据源接口来配置 JDBC 连接对象的资源。

- 许多 MyBatis 的应用程序将会按示例中的例子来配置数据源。然而它并不是必须的。要知道为了方便使用延迟加载，数据源才是必须的。

有三种内建的数据源类型（也就是 type=”???”）：

UNPOOLED – 这个数据源的实现是每次被请求时简单打开和关闭连接。它有一点慢，这是对简单应用程序的一个很好的选择，因为它不需要及时的可用连接。不同的数据库对这个的表现也是不一样的，所以对某些数据库来说配置数据源并不重要，这个配置也是闲置的。

UNPOOLED 类型的数据源仅仅用来配置以下 4 种属性：

- **driver** – 这是 JDBC 驱动的 Java 类的完全限定名（如果你的驱动包含，它也不是数据源类）。
- **url** – 这是数据库的 JDBC URL 地址。
- **username** – 登录数据库的用户名。
- **password** – 登录数据库的密码。
- **defaultTransactionIsolationLevel** – 默认的连接事务隔离级别。

作为可选项，你可以传递数据库驱动的属性。要这样做，属性的前缀是以“driver.”开头的，例如：

- **driver.encoding=UTF8**

这样就会传递以值“UTF8”来传递属性“encoding”，它是通过 DriverManager.getConnection(url,driverProperties)方法传递给数据库驱动。

POOLED – 这是 JDBC 连接对象的数据源连接池的实现，用来避免创建新的连接实例时必要的初始连接和认证时间。这是一种当前 Web 应用程序用来快速响应请求很流行的方法。

除了上述（UNPOOLED）的属性之外，还有很多属性可以用来配置 POOLED 数据源：

- **poolMaximumActiveConnections** – 在任意时间存在的活动（也就是正在使用）连接的数量。默认值：10
- **poolMaximumIdleConnections** – 任意时间存在的空闲连接数。
- **poolMaximumCheckoutTime** – 在被强制返回之前，池中连接被检查的时间。默认值：20000 毫秒（也就是 20 秒）
- **poolTimeToWait** – 这是给连接池一个打印日志状态机会的低层次设置，还有重新尝试获得连接，这些情况下往往需要很长时间（为了避免连接池没有配置时静默失败）。默认值：20000 毫秒（也就是 20 秒）
- **poolPingQuery** – 发送到数据的侦测查询，用来验证连接是否正常工作，并且准备接受请求。默认是“NO PING QUERY SET”，这会引起许多数据库驱动连接由一个错误信息而导致失败。
- **poolPingEnabled** – 这是开启或禁用侦测查询。如果开启，你必须用一个合法的 SQL 语句（最好是很快速的）设置 poolPingQuery 属性。默认值：false。
- **poolPingConnectionsNotUsedFor** – 这是用来配置 poolPingQuery 多次时间被用一次。这可以被设置匹配标准的数据库连接超时时间，来避免不必要的侦测。默认值：0（也就是所有连接每一时刻都被侦测-但仅仅当 poolPingEnabled 为 true 时适用）。

JNDI – 这个数据源的实现是为了使用如 Spring 或应用服务器这类的容器，容器可以集中或在外部配置数据源，然后放置一个 JNDI 上下文的引用。这个数据源配置只需要两个属性：

- **initial_context** – 这个属性用来从初始上下文中寻找环境（也就是 initialContext.lookup（initial——context））。这是个可选属性，如果被忽略，那么 data_source 属性将会直接以 initialContext 为背景再次寻找。
- **data_source** – 这是引用数据源实例位置的上下文的路径。它会以由 initial_context 查询返回的环境为背景来查找，如果 initial_context 没有返回结果时，直接以初始上下文为环境来查找。

和其他数据源配置相似，它也可以通过名为“env.”的前缀直接向初始上下文发送属性。比如：

- **env.encoding=UTF8**

在初始化之后，这就会以值“UTF8”向初始上下文的构造方法传递名为“encoding”的属性。

mappers

既然 MyBatis 的行为已经由上述元素配置完了，我们现在就要定义 SQL 映射语句了。但是，首先我们需要告诉 MyBatis 到哪里去找到这些语句。Java 在这方面没有提供一个很好的方法，所以最佳的方式是告诉 MyBatis 到哪里去找映射文件。你可以使用相对于类路径的资源引用，或者字符表示，或 url 引用的完全限定名（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>
```

这些语句简单告诉了 MyBatis 去哪里找映射文件。其余的细节就是在每个 SQL 映射文件中了，下面的部分我们来讨论 SQL 映射文件。

SQL 映射的 XML 文件

MyBatis 真正的力量是在映射语句中。这里是奇迹发生的地方。对于所有的力量，SQL 映射的 XML 文件是相当的简单。当然如果你将它们和对等功能的 JDBC 代码来比较，你会发现映射文件节省了大约 95%的代码量。MyBatis 的构建就是聚焦于 SQL 的，使其远离于普通的方式。

SQL 映射文件有很少的几个顶级元素（按照它们应该被定义的顺序）：

- cache - 配置给定命名空间的缓存。
- cache-ref – 从其他命名空间引用缓存配置。
- resultMap – 最复杂，也是最有力量的元素，用来描述如何从数据库结果集中来加载你的对象。
- parameterMap – 已经被废弃了！老式风格的参数映射。内联参数是首选，这个元素可能在将来被移除。这里不会记录。
- sql – 可以重用的 SQL 块，也可以被其他语句引用。
- insert – 映射插入语句
- update – 映射更新语句
- delete – 映射删除语句
- select – 映射查询语句

下一部分将从语句本身开始来描述每个元素的细节。

select

查询语句是使用 MyBatis 时最常用的元素之一。直到你从数据库取出数据时才会发现将数据存在数据库中是多么的有价值，所以许多应用程序查询要比更改数据多的多。对于每次插入，更新或删除，那也会有很多的查询。这是 MyBatis 的一个基本原则，也是将重心和努力放到查询和结果映射的原因。对简单类别的查询元素是非常简单的。比如：

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

这个语句被称作 selectPerson，使用一个 int（或 Integer）类型的参数，并返回一个 HashMap 类型的对象，其中的键是列名，值是列对应的值。

注意参数注释：

```
#{id}
```

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

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

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

select 元素有很多属性允许你配置，来决定每条语句的作用细节。

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

属性	描述
id	在命名空间中唯一的标识符，可以被用来引用这条语句。
parameterType	将会传入这条语句的参数类的完全限定名或别名。
~~parameterMap~~	~~这是引用外部 parameterMap 的已经被废弃的方法。使用内联参数映射和 parameterType 属性。~~

resultType	从这条语句中返回的期望类型的类的完全限定名或别名。注意集合情形，那应该是集合可以包含的类型，而不能是集合本身。使用 resultType 或 resultMap，但不能同时使用。
resultMap	命名引用外部的 resultMap。返回 map 是 MyBatis 最具力量的特性，对其有一个很好的理解的话，许多复杂映射的情形就能被解决了。使用 resultMap 或 resultType，但不能同时使用。
flushCache	将其设置为 true，不论语句什么时候被带哦用，都会导致缓存被清空。默认值：false。
useCache	将其设置为 true，将会导致本条语句的结果被缓存。默认值：true。
timeout	这个设置驱动程序等待数据库返回请求结果，并抛出异常时间的最大等待值。默认不设置（驱动自行处理）。
fetchSize	这是暗示驱动程序每次批量返回的结果行数。默认不设置（驱动自行处理）。
statementType	STATEMENT,PREPARED 或 CALLABLE 的一种。这会让 MyBatis 使用选择使用 Statement，PreparedStatement 或 CallableStatement。默认值：PREPARED。
resultSetType	FORWARD_ONLY|SCROLL_SENSITIVE|SCROLL_INSENSITIVE 中的一种。默认是不设置（驱动自行处理）。

insert，update，delete

数据变更语句 insert，update 和 delete 在它们的实现中非常相似：

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

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

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

```
timeout="20000">
```

属性	描述
id	在命名空间中唯一的标识符，可以被用来引用这条语句。
parameterType	将会传入这条语句的参数类的完全限定名或别名。
~~parameterMap~~	~~这是引用外部 parameterMap 的已经被废弃的方法。使用内联参数映射和 parameterType 属性。~~
flushCache	将其设置为 true，不论语句什么时候被带哦用，都会导致缓存被清空。默认值：false。
timeout	这个设置驱动程序等待数据库返回请求结果，并抛出异常时间的最大等待值。默认不设置（驱动自行处理）。
statementType	STATEMENT,PREPARED 或 CALLABLE 的一种。这会让 MyBatis 使用选择使用 Statement，PreparedStatement 或 CallableStatement。默认值：PREPARED。
useGeneratedKeys	（仅对 insert 有用）这会告诉 MyBatis 使用 JDBC 的 getGeneratedKeys 方法来取出由数据（比如：像 MySQL 和 SQL Server 这样的数据库管理系统的自动递增字段）内部生成的主键。默认值：false。
keyProperty	（仅对 insert 有用）标记一个属性，MyBatis 会通过 getGeneratedKeys 或者通过 insert 语句的 selectKey 子元素设置它的值。默认：不设置。

下面就是 insert，update 和 delete 语句的示例：

```
<insert id="insertAuthor" parameterType="domain.blog.Author">
    insert into Author (id,username,password,email,bio)
    values (#{id},#{username},#{password},#{email},#{bio})
</insert>
<update id="updateAuthor" parameterType="domain.blog.Author">
    update Author set
    username = #{username},
    password = #{password},
    email = #{email},
    bio = #{bio}
    where id = #{id}
</update>
<delete id="deleteAuthor" parameterType="int">
    delete from Author where id = #{id}
</delete>
```

如前所述，插入语句有一点多，它有一些属性和子元素用来处理主键的生成。

首先，如果你的数据库支持自动生成主键的字段（比如 MySQL 和 SQL Server），那么你可以设置 useGeneratedKeys="true"，而且设置 keyProperty 到你已经做好的目标属性上。例如，如果上面的 Author 表已经对 id 使用了自动生成的列类型，那么语句可以修改为：

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

MyBatis 有另外一种方法来处理数据库不支持自动生成类型，或者可能 JDBC 驱动不支持自动生成主键时的主键生成问题。

这里有一个简单（甚至很傻）的示例，它可以生成一个随机 ID（可能你不会这么做，但是这展示了 MyBatis 处理问题的灵活性，因为它并不真的关心 ID 的生成）：

```
<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>
```

在上面的示例中，selectKey 元素将会首先运行，Author 的 id 会被设置，然后插入语句会被调用。这给你了一个简单的行为在你的数据库中来处理自动生成的主键，而不需要使你的 Java 代码变得复杂。

selectKey 元素描述如下：

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

属性	描述
keyProperty	selectKey 语句结果应该被设置的目标属性。
resultType	结果的类型。MyBatis 通常可以算出来，但是写上也没有问题。MyBatis 允许任何简单类型用作主键的类型，包括字符串。
order	这可以被设置为 BEFORE 或 AFTER。如果设置为 BEFORE，那么它会首先选择主键，设置 keyProperty 然后执行插入语句。如果设置为 AFTER，那么先执行插入语句，然后是 selectKey 元素-这和如 Oracle 数据库相似，可以在插入语句中嵌入序列调用。
statementType	和前面的相同，MyBatis 支持 STATEMENT，PREPARED 和 CALLABLE 语句的映射类型，分别代表 PreparedStatement 和 CallableStatement 类型。

sql

这个元素可以被用来定义可重用的 SQL 代码段，可以包含在其他语句中。比如：

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

这个 SQL 片段可以被包含在其他语句中，例如：

```
<select id="selectUsers" parameterType="int" resultType="hashmap">
    select <include refid="userColumns"/>
    from some_table
```

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

Parameters

在之前的语句中，你已经看到了一些简单参数的示例。在 MyBatis 中参数是非常强大的元素。对于简单的做法，大概 90%的情况，是不用太多的，比如：

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

上面的这个示例说明了一个非常简单的命名参数映射。参数类型被设置为“int”，因此这个参数可以被设置成任何内容。原生的类型或简单数据类型，比如整型和没有相关属性的字符串，因此它会完全用参数来替代。然而，如果你传递了一个复杂的对象，那么 MyBatis 的处理方式就会有一点不同。比如：

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

如果 User 类型的参数对象传递到了语句中，id、username 和 password 属性将会被查找，然后它们的值就被传递到预处理语句的参数中。

这点对于传递参数到语句中非常好。但是对于参数映射也有一些其他的特性。

首先，像 MyBatis 的其他部分，参数可以指定一个确定的数据类型。

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

像 MyBatis 的剩余部分，javaType 通常可以从参数对象中来去顶，除非对象是一个 HashMap。那么 javaType 应该被确定来保证使用正确类型处理器。

注意：如果 null 被当作值来传递，对于所有可能为空的列，JDBC Type 是需要的。以可以自己通过阅读预处理语句的 setNull()方法的 JavaDocs 文档来研究这个。

为了自定义类型处理器，你可以指定一个确定的类型处理器类（或别名），比如：

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

尽管它看起来繁琐，但是实际上是你很少设置它们其中之一。

对于数值类型，对于决定有多少数字是相关的，有一个数值范围。

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

最后，mode 属性允许你指定 IN，OUT 或 INOUT 参数。如果参数为 OUT 或 INOUT，参数对象属性的真实值将会被改变，就像你期望你需要你个输出参数。如果 mode 为 OUT（或 INOUT），而且 jdbcType 为 CURSOR（也就是 Oracle 的 REFCURSOR），你必须指定一个 resultMap 来映射结果集到参数类型。要注意这里的 javaType 属性是可选的，如果左边的空白是 jdbcType 的 CURSOR 类型，它会自动地被设置为结果集。

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

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

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

尽管所有这些强大的选项很多时候你只简单指定属性名，MyBatis 会自己计算剩余的。最多的情况是你为 jdbcType 指定可能为空的列名。

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

字符串替换

默认情况下，使用#{}格式的语法会导致 MyBatis 创建预处理语句属性并以它为背景设置安全的值（比如?）。这样做很安全，很迅速也是首选做法，有时你只是想直接在 SQL 语句中插入一个不改变的字符串。比如，像 ORDER BY，你可以这样来使用：

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

这里 MyBatis 不会修改或转义字符串。

重要：接受从用户输出的内容并提供给语句中不变的字符串，这样做是不安全的。这会导致潜在的 SQL 注入攻击，因此你不应该允许用户输入这些字段，或者通常自行转义并检查。

resultMap

resultMap 元素是 MyBatis 中最重要最强大的元素。它就是让你远离 90%的需要从结果集中取出数据的 JDBC 代码的那个东西，而且在一些情形下允许你做一些 JDBC 不支持的事情。事实上，编写相似于对复杂语句联合映射这些等同的代码，也许可以跨过上千行的代码。ResultMap 的设计就是简单语句不需要明确的结果映射，而很多复杂语句确实需要描述它们的关系。

你已经看到简单映射语句的示例了，但没有明确的 resultMap。比如：

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

这样一个语句简单作用于所有列被自动映射到 HashMap 的键上，这由 resultType 属性指定。这在很多情况下是有用的，但是 HashMap 不能很好描述一个领域模型。那样你的应用程序将会使用 JavaBeans 或 POJOs（Plain Old Java Objects，普通 Java 对象）来作为领域模型。MyBatis 对两者都支持。看看下面这个 JavaBean：

```
package com.someapp.model;
public class User {
    private int id;
    private String username;
    private String hashedPassword;
```

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

基于 JavaBean 的规范，上面这个类有 3 个属性：id，username 和 hashedPassword。这些在 select 语句中会精确匹配到列名。

这样的一个 JavaBean 可以被映射到结果集，就像映射到 HashMap 一样简单。

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

要记住类型别名是你的伙伴。使用它们你可以不用输入类的全路径。比如：

```
<!-- 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}
</select>
```

这些情况下，MyBatis 会在幕后自动创建一个 ResultMap，基于属性名来映射列到 JavaBean 的属性上。如果列名没有精确匹配，你可以在列名上使用 select 字句的别名（一个基本的 SQL 特性）来匹配标签。比如：

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

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

ResultMap 最优秀的地方你已经了解了很多了，但是你还没有真正的看到一个。这些简单的示例不需要比你看到的更多东西。只是出于示例的原因，让我们来看看最后一个示例中外部的 resultMap 是什么样子的，这也是解决列名不匹配的另外一种方式。

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

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

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

如果世界总是这么简单就好了。

高级结果映射

MyBatis 创建的一个想法：数据库不用永远是你想要的或需要它们是什么样的。而我们最喜欢的数据库最好是第三范式或 BCNF 模式，但它们有时不是。如果可能有一个单独的数据库映射，所有应用程序都可以使用它，这是非常好的，但有时也不是。结果映射就是 MyBatis 提供处理这个问题的答案。

比如，我们如何映射下面这个语句？

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

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

你可能想把它映射到一个智能的对象模型，包含一个作者写的博客，有很多的博文，每篇博文有零条或多条的评论和标签。下面是一个完整的复杂结果映射例子(假设作者,博客，博文，评论和标签都是类型的别名)。我们来看看，但是不用紧张，我们会一步一步来说明。当天最初它看起来令人生畏，但实际上非常简单。

```
<!-- 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>
```

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

resultMap

- **constructor** – 类在实例化时，用来注入结果到构造方法中
  - **idArg** – ID 参数；标记结果作为 ID 可以帮助提高整体效能
  - **arg** – 注入到构造方法的一个普通结果
- **id** – 一个 ID 结果；标记结果作为 ID 可以帮助提高整体效能
- **result** – 注入到字段或 JavaBean 属性的普通结果
- **association** – 一个复杂的类型关联；许多结果将包成这种类型
  - 嵌入结果映射 – 结果映射自身的关联，或者参考一个
- **collection** – 复杂类型的集
  - 嵌入结果映射 – 结果映射自身的集，或者参考一个
- **discriminator** – 使用结果值来决定使用哪个结果映射
  - **case** – 基于某些值的结果映射
    - 嵌入结果映射 – 这种情形结果也映射它本身，因此可以包含很多相同的元素，或者它可以参照一个外部的结果映射。

最佳实践：通常逐步建立结果映射。单元测试的真正帮助在这里。如果你尝试创建一次创建一个向上面示例那样的巨大的结果映射，那么可能会有错误而且很难去控制它来工作。开始简单一些，一步一步的发展。而且要进行单元测试！使用该框架的缺点是它们有时是黑盒（是否可见源代码）。你确定你实现想要的行为的最好选择是编写单元测试。它也可以你帮助得到提交时的错误。

下面一部分将详细说明每个元素。

id，result

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

这些是结果映射最基本内容。id 和 result 都映射一个单独列的值到简单数据类型（字符串，整型，双精度浮点数，日期等）的单独属性或字段。

这两者之间的唯一不同是 id 表示的结果将是当比较对象实例时用到的标识属性。这帮助来改进整体表现，特别是缓存和嵌入结果映射（也就是联合映射）。

每个都有一些属性：

属性	描述
property	映射到列结果的字段或属性。如果匹配的是存在的，和给定名称相同的 JavaBeans 的属性，那么就会使用。否则 MyBatis 将会寻找给定名称的字段。这两种情形你可以使用通常点式的复杂属性导航。比如，你可以这样映射一些东西：“username”，或者映射到一些复杂的东西：“address.street.number”。
column	从数据库中得到的列名，或者是列名的重命名标签。这也是通常和会传递给 resultSet.getString(columnName)方法参数中相同的字符串。
javaType	一个 Java 类的完全限定名，或一个类型别名（参加上面内建类型别名的列表）。如果你映射到一个 JavaBean，MyBatis 通常可以断定类型。然而，如果你映射到的是 HashMap，那么你应该明确地指定 javaType 来保证所需的行为。
jdbcType	在这个表格之后的所支持的 JDBC 类型列表中的类型。JDBC 类型是仅仅需要对插入，更新和删除操作可能为空的列进行处理。这是 JDBC 的需要，而不是 MyBatis 的。如果你直接使用 JDBC 编程，你需要指定这个类型-但仅仅对可能为空的值。
typeHandler	我们在前面讨论过默认的类型处理器。使用这个属性，你可以覆盖默认的类型处理器。这个属性值是类的完全限定名或者是一个类型处理器的实现，或者是类型别名。

支持的 JDBC 类型

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

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

构造方法

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

对于大多数数据传输对象(Data Transfer Object，DTO)类型，属性可以起作用，而且像你绝大多数的领域模型，指令也许是你想使用一成不变的类的地方。通常包含引用或查询数据的表很少或基本不变的话对一成不变的类来说是合适的。构造方法注入允许你在初始化时为类设置属性的值，而不用暴露出公有方法。MyBatis 也支持私有属性和私有 JavaBeans 属性来达到这个目的，但是一些人更青睐构造方法注入。构造方法元素支持这个。

看看下面这个构造方法：

```
public class User {
    //…
```

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

为了向这个构造方法中注入结果，MyBatis 需要通过它的参数的类型来标识构造方法。Java 没有自查（反射）参数名的方法。所以当创建一个构造方法元素时，保证参数是按顺序排列的，而且数据类型也是确定的。

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

剩余的属性和规则和固定的 id 和 result 元素是相同的。

属性	描述
column	来自数据库的类名，或重命名的列标签。这和通常传递给 resultSet.getString(columnName)方法的字符串是相同的。
javaType	一个 Java 类的完全限定名，或一个类型别名（参加上面内建类型别名的列表）。如果你映射到一个 JavaBean，MyBatis 通常可以断定类型。然而，如果你映射到的是 HashMap，那么你应该明确地指定 javaType 来保证所需的行为。
jdbcType	在这个表格之前的所支持的 JDBC 类型列表中的类型。**JDBC 类型是仅仅需要对插入，更新和删除操作可能为空的列进行处理。**这是 JDBC 的需要，而不是 MyBatis 的。如果你直接使用 JDBC 编程，你需要指定这个类型-但仅仅对可能为空的值。
typeHandler	我们在前面讨论过默认的类型处理器。使用这个属性，你可以覆盖默认的类型处理器。这个属性值是类的完全限定名或者是一个类型处理器的实现，或者是类型别名。

关联

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

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

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

- **嵌套查询**：通过执行另外一个 SQL 映射语句来返回预期的复杂类型。
- **嵌套结果**：使用嵌套结果映射来处理重复的联合结果的子集。

首先，然让我们来查看这个元素的属性。所有的你都会看到，它和普通的只由 select 和

resultMap 属性的结果映射不同。

属性	描述
property	映射到列结果的字段或属性。如果匹配的是存在的，和给定名称相同的 JavaBeans 的属性，那么就会使用。否则 MyBatis 将会寻找给定名称的字段。这两种情形你可以使用通常点式的复杂属性导航。比如，你可以这样映射一些东西：“username”，或者映射到一些复杂的东西：“address.street.number”。
column	来自数据库的类名，或重命名的列标签。这和通常传递给 resultSet.getString(columnName)方法的字符串是相同的。 **注意：要处理复合主键，你可以指定多个列名通过 column=”{prop1=col1,prop2=col2}”这种语法来传递给嵌套查询语句。这会引起 prop1 和 prop2 以参数对象形式来设置给目标嵌套查询语句。**
javaType	一个 Java 类的完全限定名，或一个类型别名（参加上面内建类型别名的列表）。如果你映射到一个 JavaBean，MyBatis 通常可以断定类型。然而，如果你映射到的是 HashMap，那么你应该明确地指定 javaType 来保证所需的行为。
jdbcType	在这个表格之前的所支持的 JDBC 类型列表中的类型。**JDBC 类型是仅仅需要对插入，更新和删除操作可能为空的列进行处理。**这是 JDBC 的需要，而不是 MyBatis 的。如果你直接使用 JDBC 编程，你需要指定这个类型-但仅仅对可能为空的值。
typeHandler	我们在前面讨论过默认的类型处理器。使用这个属性，你可以覆盖默认的类型处理器。这个属性值是类的完全限定名或者是一个类型处理器的实现，或者是类型别名。

关联的嵌套查询

select	另外一个映射语句的 ID，可以加载这个属性映射需要的复杂类型。获取的在列属性中指定的列的值将被传递给目标 select 语句作为参数。表格后面有一个详细的示例。 **注意：要处理复合主键，你可以指定多个列名通过 column=”{prop1=col1,prop2=col2}”这种语法来传递给嵌套查询语句。这会引起 prop1 和 prop2 以参数对象形式来设置给目标嵌套查询语句。**

示例：

```
<resultMap id=”blogResult” type=”Blog”>
    <association property="author" column="blog_author_id"
        javaType="Author" select=”selectAuthor”/>
</resultMap>
<select id=”selectBlog” parameterType=”int” resultMap=”blogResult”>
    SELECT * FROM BLOG WHERE ID = #{id}
</select>
<select id=”selectAuthor” parameterType=”int” resultType="Author">
    SELECT * FROM AUTHOR WHERE ID = #{id}
</select>
```

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

其他所有的属性将会被自动加载，假设它们的列和属性名相匹配。

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

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

这个问题会导致成百上千的 SQL 语句被执行。这通常不是期望的。

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

所以还有另外一种方法。

关联的嵌套结果

resultMap	这是结果映射的 ID，可以映射关联的嵌套结果到一个合适的对象图中。这是一种替代方法来调用另外一个查询语句。这允许你联合多个表来合成到一个单独的结果集。这样的结果集可能包含重复，数据的重复组需要被分解，合理映射到一个嵌套的对象图。为了使它变得容易，MyBatis 让你“链接”结果映射，来处理嵌套结果。一个例子会很容易来仿照，这个表格后面也有一个示例。

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

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

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

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

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

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

非常重要：在嵌套据诶过映射中 id 元素扮演了非常重要的角色。应应该通常指定一个或多个属性，它们可以用来唯一标识结果。实际上就是如果你离开她了，但是有一个严重的性能问题时 MyBatis 仍然可以工作。选择的属性越少越好，它们可以唯一地标识结果。主键就是一个显而易见的选择（尽管是联合主键）。

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

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

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

集合

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

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

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

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

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

集合的嵌套查询

首先，让我们看看使用嵌套查询来为博客加载文章。

```
<resultMap id=”blogResult” type=”Blog”>
    <collection property="posts" javaType=”ArrayList” column="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>
```

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

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

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

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

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

集合的嵌套结果

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

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

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

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

同样，要记得 id 元素的重要性，如果你不记得了，请阅读上面的关联部分。

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

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

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

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

鉴别器

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

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

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

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

在这个示例中，MyBatis 会从结果集中得到每条记录，然后比较它的 vehicle 类型的值。

如果它匹配任何一个鉴别器的实例，那么就使用这个实例指定的结果映射。换句话说，这样做完全是剩余的结果映射被忽略（除非它被扩展，这在第二个示例中讨论）。如果没有任何一个实例相匹配，那么 MyBatis 仅仅使用鉴别器块外定义的结果映射。所以，如果 carResult 按如下声明：

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

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

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

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

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

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

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

缓存

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

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

```
<cache/>
```

字面上看就是这样。这个简单语句的效果如下：

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

所有的这些属性都可以通过缓存元素的属性来修改。比如：

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

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

可用的收回策略有：

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

默认的是 LRU。

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

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

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

使用自定义缓存

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

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

这个示例展示了如何使用一个自定义的缓存实现。type 属性指定的类必须实现 org.mybatis.cache.Cache 接口。这个接口是 MyBatis 框架中很多复杂的接口之一，但是简单给定它做什么就行。

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

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

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

你可以使用所有简单类型作为 JavaBeans 的属性，MyBatis 会进行转换。

记得缓存配置和缓存实例是绑定在 SQL 映射文件的命名空间是很重要的。因此，所有在相同命名空间的语句正如绑定的缓存一样。语句可以修改和缓存交互的方式，或在语句的语句的基础上使用两种简单的属性来完全排除它们。默认情况下，语句可以这样来配置：

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

因为那些是默认的，你明显不能明确地以这种方式来配置一条语句。相反，如果你想改变默认的行为，只能设置 flushCache 和 useCache 属性。比如，在一些情况下你也许想排除从缓存中查询特定语句结果，或者你也许想要一个查询语句来刷新缓存。相似地，你也许有一些更新语句依靠执行而不需要刷新缓存。

参照缓存

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

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

动态 SQL

MyBatis 的一个强大的特性之一通常是它的动态 SQL 能力。如果你有使用 JDBC 或其他

相似框架的经验，你就明白条件地串联 SQL 字符串在一起是多么的痛苦，确保不能忘了空格或在列表的最后省略逗号。动态 SQL 可以彻底处理这种痛苦。

通常使用动态 SQL 不可能是独立的一部分，MyBatis 当然使用一种强大的动态 SQL 语言来改进这种情形，这种语言可以被用在任意映射的 SQL 语句中。

动态 SQL 元素和使用 JSTL 或其他相似的基于 XML 的文本处理器相似。在 MyBatis 之前的版本中，有很多的元素需要来了解。MyBatis 3 大大提升了它们，现在用不到原先一半的元素就能工作了。MyBatis 采用功能强大的基于 OGNL 的表达式来消除其他元素。

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

if

在动态 SQL 中所做的最通用的事情是包含部分 where 字句的条件。比如：

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

这条语句会提供一个可选的文本查找功能。如果你没有传递 title，那么所有激活的博客都会被返回。但是如果你传递了 title，那么就会查找相近的 title（对于敏锐的检索，这中情况下你的参数值需要包含任意的遮掩或通配符）的博客。

假若我们想可选地搜索 title 和 author 呢？首先，要改变语句的名称让它有意义。然后简单加入另外的一个条件。

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

choose, when, otherwise

有时我们不想应用所有的条件，相反我们想选择很多情况下的一种。和 Java 中的 switch 语句相似，MyBatis 提供 choose 元素。

我们使用上面的示例，但是现在我们来搜索当 title 提供时仅有 title 条件，当 author 提供时仅有 author 条件。如果二者都没提供，只返回 featured blogs（也许是由管理员策略地选择的结果列表，而不是返回大量没有意义的随机博客结果列表）。

```
<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

前面的例子已经方便地处理了一个臭名昭著的动态 SQL 问题。要考虑我们回到“if”示例后会发生什么，但是这次我们将“ACTIVE = 1”也设置成动态的条件。

```
<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>
```

如果这些条件都没有匹配上将会发生什么？这条 SQL 结束时就会成这样：

```
SELECT * FROM BLOG
WHERE
```

这会导致查询失败。如果仅仅第二个条件匹配是什么样的？这条 SQL 结束时就会是这样：

```
SELECT * FROM BLOG
WHERE
```

AND title like ‘someTitle’

这个查询也会失败。这个问题不能简单的用条件来解决，如果你从来没有这样写过，那么你以后也不会这样来写。

MyBatis 有一个简单的处理，这在 90%的情况下都会有用。而在不能使用的地方，你可以自定义处理方式。加上一个简单的改变，所有事情都会顺利进行：

```
<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 元素知道如果由被包含的标记返回任意内容，就仅仅插入“WHERE”。而且，如果以“AND”或“OR”开头的内容，那么就会跳过 WHERE 不插入。

如果 *where* 元素没有做出你想要的，你可以使用 *trim* 元素来自定义。比如，和 *where* 元素相等的 *trim* 元素是：

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

overrides 属性采用管道文本分隔符来覆盖，这里的空白也是重要的。它的结果就是移除在 *overrides* 属性中指定的内容，插入在 *with* 属性中的内容。

和动态更新语句相似的解决方案是 *set*。*set* 元素可以被用于动态包含更新的列，而不包含不需更新的。比如：

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

这里，*set* 元素会动态前置 SET 关键字，而且也会消除任意无关的逗号，那也许在应用条件之后来跟踪定义的值。

如果你对和这相等的 *trim* 元素好奇，它看起来就是这样的：

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

注意这种情况下我们覆盖一个后缀，而同时也附加前缀。

foreach

另外一个动态SQL通用的必要操作是迭代一个集合，通常是构建在IN条件中的。比如：

```
<select id="selectPostIn" resultType="domain.blog.Post">
    SELECT *
    FROM POST P
    WHERE ID in
    <foreach item="item" index="index" collection="list"
        open="(" separator="," close=")">
        #{item}
    </foreach>
</select>
```

foreach 元素是非常强大的，它允许你指定一个集合，声明集合项和索引变量，它们可以用在元素体内。它也允许你指定开放和关闭的字符串，在迭代之间放置分隔符。这个元素是很智能的，它不会偶然地附加多余的分隔符。

注意：你可以传递一个 List 实例或者数组作为参数对象传给 MyBatis。当你这么做的时候，MyBatis 会自动将它包装在一个 Map 中，用名称在作为键。List 实例将会以“list”作为键，而数组实例将会以“array”作为键。

这个部分是对关于 XML 配置文件和 XML 映射文件的而讨论的。下一部分将详细讨论 Java API，所以你可以得到你已经创建的最有效的映射。

Java API

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

应用目录结构

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

让我们看一下典型应用的目录结构：

```
/my_application
    /bin
```

```
/devlib
/lib                                          ←MyBatis *.jar文件在这里。
/src
   /org/myapp/
      /action
      /data                                   ←MyBatis配置文件在这里，
         /SqlMapConfig.xml                      包括映射器类，XML配置，
         /BlogMapper.java                       XML映射文件。
         /BlogMapper.xml
      /model
      /service
      /view
   /properties                                ←在你XML中配置的属性
/test                                           文件在这里。
   /org/myapp/
      /action
      /data
      /model
      /service
      /view
   /properties
/web                                          要记得这只是参照，而不是
   /WEB-INF                                    要求，可能其他人会感谢你
      /web.xml                                 使用了通用的目录结构。
```

这部分内容剩余的示例将假设你使用了这种目录结构。

SqlSessions

使用 MyBatis 的主要 Java 接口就是 SqlSession。尽管你可以使用这个接口执行命令，获取映射器和管理事务。我们会讨论 SqlSession 本身更多，但是首先我们还是要了解如果获取一个 SqlSession 实例。SqlSessions 是由 SqlSessionFactory 实例创建的。SqlSessionFactory 对象包含创建 SqlSession 实例的所有方法。而 SqlSessionFactory 本身是由 SqlSessionFactoryBuilder 创建的，它可以从 XML 配置，注解或手动配置 Java 来创建 SqlSessionFactory。

SqlSessionFactoryBuilder

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

```
SqlSessionFactory build(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)
```

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

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

如果你调用了一个使用 *environment* 参数的 build 方法，那么 MyBatis 将会使用 configuration 对象来配置这个 environment。当然，如果你指定了一个不合法的 environment，你会得到错误提示。如果你调用了其中之一没有 *environment* 参数的 build 方法，那么就使用默认的 environment（在上面的示例中就会指定为 default=”development”）。

如果你调用了使用 properties 实例的方法，那么 MyBatis 就会加载那些 properties（属性配置文件），并你在你配置中可使用它们。那些属性可以用${propName}语法形式多次用在配置文件中。

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

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

- 在 properties 元素体中指定的属性首先被读取，
- 从 properties 元素的类路径 resource 或 url 指定的属性第二个被读取，可以覆盖已经指定的重复属性，
- 作为方法参数传递的属性最后被读取，可以覆盖已经从 properties 元素体和 resource/url 属性中加载的任意重复属性。

因此，最高优先级的属性是通过方法参数传递的，之后是 resource/url 属性指定的，最后是在 properties 元素体中指定的属性。

总结一下，前四个方法很大程度上是相同的，但是由于可以覆盖，就允许你可选地指定 environment 和/或 properties。这里给出一个从 SqlMapConfig.xml 文件创建 SqlSessionFactory 的示例：

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

注意这里我们使用了 *Resources* 工具类，这个类在 org.mybatis.io 包中。Resources 类正如其名，会帮助你从类路径下，文件系统或一个 web URL 加载资源文件。看一下这个类的

源代码或者通过你的 IDE 来查看，就会看到一整套有用的方法。这里给出一个简表：

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

最后一个 build 方法使用了一个 Configuration 实例。configuration 类包含你可能需要了解 SqlSessionFactory 实例的所有内容。Configuration 类对于配置的自查很有用，包含查找和操作 SQL 映射（不推荐使用，因为应用正接收请求）。configuration 类有所有配置的开关，这些你已经了解了，只在 Java API 中露出来。这里有一个简单的示例，如何手动配置 configuration 实例，然后将它传递给 build()方法来创建 SqlSessionFactory。

```
DataSource dataSource = BaseDataTest.createBlogDataSource();
TransactionFactory transactionFactory = new JdbcTransactionFactory();
Environment environment =
new Environment("development", transactionFactory, dataSource);
Configuration configuration = new Configuration(environment);
configuration.setLazyLoadingEnabled(true);
configuration.setEnhancementEnabled(true);
configuration.getTypeAliasRegistry().registerAlias(Blog.class);
configuration.getTypeAliasRegistry().registerAlias(Post.class);
configuration.getTypeAliasRegistry().registerAlias(Author.class);
configuration.addMapper(BoundBlogMapper.class);
configuration.addMapper(BoundAuthorMapper.class);
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(configuration);
```

现在你有一个 SqlSessionFactory，可以用来创建 SqlSession 实例。

SqlSessionFactory

SqlSessionFactory 有六个方法可以用来创建 SqlSession 实例。通常来说，如何决定是你选择下面这些方法时：

- **Transaction**（事务）：你想为 session 使用事务或者使用自动提交（通常意味着很多数据库和/或 JDBC 驱动没有事务）？
- **Connection**（连接）：你想 MyBatis 获得来自配置的数据源的连接还是提供你自己

定义的连接？

- **Execution**（执行）：你想 MyBatis 复用预处理语句和/或批量更新语句（包括插入和删除）？

重载的 openSession()方法签名设置允许你选择这些可选中的任何一个组合。

```
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()方法没有参数，它会创建有如下特性的 SqlSession：

- 会开启一个事务（也就是不自动提交）
- 连接对象会从由活动环境配置的数据源实例中得到。
- 事务隔离级别将会使用驱动或数据源的默认设置。
- 预处理语句不会被复用，也不会批量处理更新。

这些方法大都可以自我解释的。开启自动提交，传递“true”给可选的 *autoCommit* 参数。提供自定义的连接，传递一个 Connection 实例给 *connection* 参数。注意没有覆盖同时设置 Connection 和 autoCommit 两者的方法，因为 MyBatis 会使用当前 connection 对象提供的设置。MyBatis 为事务隔离级别调用使用一个 Java 枚举包装器，称为 TransactionIsolationLevel，否则它们按预期的方式来工作，并有 JDBC 支持的 5 级（NONE,READ_UNCOMMITTED,READ_COMMITTED,REPEATABLE_READ,SERIALIZABLE）

还有一个可能对你来说是新见到的参数，就是 ExecutorType。这个枚举类型定义了 3 个值：

ExecutorType.SIMPLE

这个执行器类型不做特殊的事情。它为每个语句的执行创建一个新的预处理语句。

ExecutorType.REUSE

这个执行器类型会复用预处理语句。

ExecutorType.BATCH

这个执行器会批量执行所有更新语句，如果 SELECT 在它们中间执行还会标定它们是必须的，来保证一个简单并易于理解的行为。

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

注意：如果你已经使用之前版本 MyBatis，你要回忆那些 session，transaction 和 batch 都是分离的。现在和以往不同了，这些都包含在 session 的范围内了。你需要处理分开处理事务或批量操作来得到它们的效果。

SqlSession

如上面所提到的，SqlSession 实例在 MyBatis 中是非常强大的一个类。在这里你会发现

所有执行语句的方法，提交或回滚事务，还有获取映射器实例。

在 SqlSession 类中有超过 20 个方法，所以将它们分开成易于理解的组合。

语句执行方法

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

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

selectOne 和 selectList 的不同仅仅是 selectOne 必须返回一个对象。如果多余一个，或者没有返回（或返回了 null），那么就会抛出异常。如果你不知道需要多少对象，使用 selectList。如果你想检查一个对象是否存在，那么最好返回统计数（0 或 1）。因为并不是所有语句都需要参数，这些方法都是有不同重载版本的，它们可以不需要参数对象。

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

最后，还有查询方法的三个高级版本，它们允许你限制返回行数的范围，或者提供自定义结果控制逻辑，这通常用于大量的数据集合。

```
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 参数会告诉 MyBatis 略过指定数量的记录，还有限制返回结果的数量。RowBounds 类有一个构造方法来接收 offset 和 limit，否则是不可改变的。

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

不同的驱动会实现这方面的不同级别的效率。对于最佳的表现，使用结果集类型的 SCROLL_SENSITIVE 或 SCROLL_INSENSITIVE（或句话说：不是 FORWARD_ONLY）。

ResultHandler 参数允许你按你喜欢的方式处理每一行。你可以将它添加到 List 中，创建 Map，Set 或抛出每个结果而不是只保留总计。你可以使用 ResultHandler 做很多漂亮的事，那就是 MyBatis 内部创建结果集列表。

它的接口很简单。

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

ResultContext 参数给你访问结果对象本身的方法，大量结果对象被创建，你可以使用布尔返回值的 stop()方法来停止 MyBatis 加载更多的结果。

事务控制方法

控制事务范围有四个方法。当然，如果你已经选择了自动提交或你正在使用外部事务管理器，这就没有任何效果了。然而，如果你正在使用 JDBC 事务管理员，由 Connection 实例来控制，那么这四个方法就会派上用场：

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

默认情况下 MyBatis 不会自动提交事务，除非它侦测到有插入，更新或删除操作改变了数据库。如果你已经做出了一些改变而没有使用这些方法，那么你可以传递 true 到 commit 和 rollback 方法来保证它会被提交（注意，你不能在自动提交模式下强制 session，或者使用了外部事务管理器时）。很多时候你不用调用 rollback()，因为如果你没有调用 commit 时 MyBatis 会替你完成。然而，如果你需要更多对多提交和回滚都可能的 session 的细粒度控制，你可以使用回滚选择来使它成为可能。

清理 Session 级的缓存

```
void clearCache()
```

SqlSession 实例有一个本地缓存在执行 update，commit，rollback 和 close 时被清理。要明确地关闭它（获取打算做更多的工作），你可以调用 clearCache()。

确保 SqlSession 被关闭

```
void close()
```

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

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

注意：就像 SqlSessionFactory，你可以通过调用 getConfiguration()方法获得 SqlSession 使用的 Configuration 实例

```
Configuration getConfiguration()
```

使用映射器

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

上述的各个 insert，update，delete 和 select 方法都很强大，但也有些繁琐，没有类型安全，对于你的 IDE 也没有帮助，还有可能的单元测试。在上面的入门章节中我们已经看到了一个使用映射器的示例。

因此，一个更通用的方式来执行映射语句是使用映射器类。一个映射器类就是一个简单的接口，其中的方法定义匹配于 SqlSession 方法。下面的示例展示了一些方法签名和它们是

如何映射到 SqlSession 的。

```
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);
}
```

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

此外，返回类型必须匹配期望的结果类型。所有常用的类型都是支持的，包括：原生类型，Map，POJO 和 JavaBean。

映射器接口不需要去实现任何接口或扩展任何类。只要方法前面可以被用来唯一标识对应的映射语句就可以了。

映射器接口可以扩展其他接口。当使用 XML 来构建映射器接口时要保证在合适的命名空间中有语句。而且，唯一的限制就是你不能在两个继承关系的接口中有相同的方法签名（这也是不好的想法）。

你可以传递多个参数给一个映射器方法。如果你这样做了，默认情况下它们将会以它们在参数列表中的位置来命名，比如：#{1},#{2}等。如果你想改变参数的名称（只在多参数情况下），那么你可以在参数上使用@Param("paramName")注解。

你也可以给方法传递一个 RowBounds 实例来限制查询结果。

映射器注解

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

注意：不幸的是，Java 注解限制了它们的表现和灵活。尽管很多时间都花调查，设计和实验上，最强大的 MyBatis 映射不能用注解来构建，那并不可笑。C#属性（做示例）就没有这些限制，因此 MyBatis.NET 将会比 XML 有更丰富的选择。也就是说，基于 Java 注解的配置离不开它的特性。

注解有下面这些：

注解	目标	相对应的 XML	描述
@CacheNamespace	类	<cache>	为给定的命名空间（比如类）配置缓存。属性：**implemetation,eviction, flushInterval,size** 和 **readWrite**。
@CacheNamespaceRef	类	<cacheRef>	参照另外一个命名空间的缓存来使用。属性：**value**，应该是一个名空姐的字符串值（也就是类的完全限定名）。

@ConstructorArgs	方法	<constructor>	收集一组结果传递给一个劫夺对象的构造方法。属性：**value**，是形式参数的数组。
@Arg	方法	<arg> <idArg>	单独的构造方法参数，是 ConstructorArgs 集合的一部分。属性：**id**，**column**，**javaType**，**typeHandler**。id 属性是布尔值，来标识用于比较的属性，和<idArg>XML 元素相似。
@TypeDiscriminator	方法	<discriminator>	一组实例值被用来决定结果映射的表现。属性：**column**，**javaType**，**jdbcType**，**typeHandler**，**cases**。cases 属性就是实例的数组。
@Case	方法	<case>	单独实例的值和它对应的映射。属性：**value**，**type**，**results**。Results 属性是结果数组，因此这个注解和实际的 ResultMap 很相似，由下面的 Results 注解指定。
@Results	方法	<resultMap>	结果映射的列表，包含了一个特别结果列如何被映射到属性或字段的详情。属性：**value**，是 Result 注解的数组。
@Result	方法	<result> <id>	在列和属性或字段之间的单独结果映射。属性：**id**，**column**，**property**，**javaType**，**jdbcType**，**typeHandler**，**one**，**many**。id 属性是一个布尔值，表示了应该被用于比较（和在 XML 映射中的<id>相似）的属性。one 属性是单独的联系，和<association>相似，而 many 属性是对集合而言的，和<collection>相似。它们这样命名是为了避免名称冲突。
@One	方法	<association>	复杂类型的单独属性值映射。属性：**select**，已映射语句（也就是映射器方法）的完全限定名，它可以加载合适类型的实例。**注意**：联合映射在注解 API 中是不支持的。这是因为 Java 注解的限制，不允许循环引用。
@Many	方法	<collection>	复杂类型的集合属性映射。属性：**select**，是映射语句（也就是映射器方法）的完全限定名，它可以加载合适类型的一组实例。**注意**：联合映射在 Java 注解中是不支持的。这是因为 Java 注解的限制，不允许循环引用。
@Options	方法	映射语句的属性	这个注解提供访问交换和配置选项的宽广范围，它们通常在映射语句上作为

			属性出现。而不是将每条语句注解变复杂，Options 注解提供连贯清晰的方式来访问它们。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，因为没有办法来指定“null”作为值。因此，一旦你使用了 Options 注解，语句就受所有默认值的支配。要注意什么样的默认值来避免不期望的行为。
@Insert **@Update** **@Delete**	方法	<insert> <update> <delete>	这些注解中的每一个代表了执行的真实 SQL。它们每一个都使用字符串数组（或单独的字符串）。如果传递的是字符串数组，它们由每个分隔它们的单独空间串联起来。这就当用 Java 代码构建 SQL 时避免了“丢失空间”的问题。然而，如果你喜欢，也欢迎你串联单独的字符串。属性：**value**，这是字符串数组用来组成单独的 SQL 语句。
@InsertProvider **@UpdateProvider** **@DeleteProvider** **@SelectProvider**	方法	<insert> <update> <delete> <select> 允许创建动态 SQL。	这些可选的 SQL 注解允许你指定一个类名和一个方法在执行时来返回运行的 SQL。基于执行的映射语句，MyBatis 会实例化这个类，然后执行由 provider 指定的方法.这个方法可以选择性的接受参数对象作为它的唯一参数，但是必须只指定该参数或者没有参数。属性：**type**，**method**。type 属性是类的完全限定名。method 是该类中的那个方法名。**注意**：这节之后是对 SelectBuilder 类的讨论，它可以帮助你以干净，容于阅读的方式来构建动态 SQL。
@Param	参数	N/A	如果你的映射器的方法需要多个参数，这个注解可以被应用于映射器的方法参数来给每个参数一个名字。否则，多参数将会以它们的顺序位置来被命名（不包括任何 RowBounds 参数）。比如 #{1}，#{2}等，这是默认的。使用 @Param(“person”)，参数应该被命名为 #{person}。

SelectBuilder

一个 Java 程序员面对的最痛苦的事情之一就是在 Java 代码中嵌入 SQL 语句。通常这么做是因为 SQL 要动态的生成-否则你可以将它们放到外部的文件或存储过程中。正如你已经看到的，MyBatis 在它的 XML 映射特性中有处理生成动态 SQL 的很强大的方案。然而，有时必须在 Java 代码中创建 SQL 语句的字符串。这种情况下，MyBatis 有另外一种特性来帮助你，在减少典型的加号，引号，新行，格式化问题和嵌入条件来处理多余的逗号或 AND 连接词之前，事实上，在 Java 代码中动态生成 SQL 就是一个噩梦。

MyBatis 3 引入了一些不同的理念来处理这个问题，我们可以创建一个类的实例来调用其中的方法来一次构建 SQL 语句。但是我们的 SQL 结尾时看起来很像 Java 代码而不是 SQL 语句。相反，我们尝试了一些不同的做法。最终的结果是关于特定领域语言的结束，Java 也不断实现它目前的形式…

SelectBuilder 的秘密

SelectBuilder 类并不神奇，如果你不了解它的工作机制也不会有什么好的作用。别犹豫，让我们来看看它是怎么工作的。SelectBuilder 使用了静态引入和 TreadLocal 变量的组合来开启简洁的语法可以很容易地用条件进行隔行扫描，而且为你保护所有 SQL 的格式。它允许你创建这样的方法：

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

这是一个非常简单的示例，你也许会选择静态地来构建。所以这里给出一个复杂一点的示例：

```
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();
```

```
}
```

用字符串连接的方式来构建上面的 SQL 就会有一些繁琐了。比如：

```
"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";
```

如果你喜欢那样的语法，那么你就可以使用它。它很容易出错，要小心那些每行结尾增加的空间。现在，即使你喜欢这样的语法，下面的示例比 Java 中的字符串连接要简单也是没有疑问的：

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

这个例子有什么特殊之处？如果你看得仔细，那就不同担心偶然会重复的“AND”关键字，或在“WHERE”和“AND”或两者都没有中选择！上面的语句将会由例子对所有 PERSON 记录生成一个查询，有像参数一样的 ID 或 firstName 或 lastName-或这三者的任意组合。SelectBuilder 对理解哪里放置“WHERE”，哪里应该使用“AND”还有所有的字符串连接都是很小心的。最好的情况，无论你以何种顺序调用这些方法（只有一种例外使用 OR()方法）。

有两个方法会吸引你的眼球：BEGIN()和 SQL()。总之，每个 SelectBuilder 方法应该以调用 BEGIN（）开始，以调用 SQL()结束。当然你可以在中途提取方法来打断你执行的逻辑，但是 SQL 生成的范围应该以 BEGIN()方法开始而且以 SQL()方法结束。BEGIN()方法清理 ThreadLocal 变量，来确保你不会不小心执行了前面的状态，而且 SQL()方法会基于这些调用，从最后一次调用 BEGIN()开始组装你的 SQL 语句。注意 BEGIN()有一个称为 RESET()

的代替方法，它们所做的工作相同，只是 RESET()会在特定上下文中读取的更好。

要按照上面示例的方式使用 SelectBuilder，你应该静态引入如下内容：

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

只要这个被引入了，那么你使用的类就会拥有 SelectBuilder 的所有可用的方法。下表就是可用方法的完整列表：

方法	描述
BEGIN()/RESET()	这些方法清理 SelectBuilder 类的 ThreadLocal 的状态，而且准备构建新的语句。当开始一条新的语句时，BEGIN()读取得最好。当在执行中间因为某些原因（在某些条件下，也许处理逻辑需要一个完整的而且不同的语句）要清理一条语句时 RESET()读取的做好。
SELECT(String)	开始或附加一个 SELECT 子句。可以被多次调用，而且参数会被追加在 SELECT 子句后面。参数通常是逗号分隔的列名列表和别名，但要是驱动程序可以接受的东西。
SELECT_DISTINCT(String)	开始或附加一个 SELECT 子句，也在生成的查询语句中添加“DISTINCT”关键字。可以被多次调用，而且参数会被追加在 SELECT 子句后面。参数通常是逗号分隔的列名列表和别名，但要是驱动程序可以接受的东西。
FROM(String)	开始或附加一个 FROM 子句。可以被多次调用，而且参数会被追加在 FROM 子句后面。参数通常是表明或别名，或是驱动程序可以接受的任意内容。
JOIN(String) INNER_JOIN(String) LEFT_OUTER_JOIN(String) RIGHT_OUTER_JOIN(String)	基于调用的方法，添加一个合适类型的新的 JOIN 子句。参数可以包含列之间基本的 join 连接还有条件连接。
WHERE(String)	添加一个新的 WHERE 条件子句，由 AND 串联起来。可以被多次调用，由 AND 告诉它来串联一个新的条件。使用 OR()方法来分隔 OR 条件。
OR()	使用 OR 来分隔当前 WHERE 子句的条件。可以被多次调用，但是在一行上多次调用会生成不稳定的 SQL。
AND()	使用 AND 来分隔当前 WHERE 字句的条件。可以被多次调用，但是在一行上多次调用会生成不稳定的 SQL。因为 WHERE 和 HAVING 两者都自动串联 AND，这样使用是非常罕见的，包含它也仅仅是为了完整性。
GROUP_BY(String)	附加一个新的 GROUP BY 子句，由逗号串联起来。可以被多次调用，每次使用逗号来告诉它串联一个新的条件。
HAVING(String)	附加一个新的 HAVING 条件子句，由 AND 串联起来。可以被多次调用，每次使用 AND 来告诉它要串联新的条件。使用 OR()方法来分隔 OR 条件。
ORDER_BY(String)	附加一个新的 ORDER BY 子句，由逗号串联起来。可以被多次调用，每次使用逗号来告诉它串联新的条件。
SQL()	这会返回生成 SQL 而且重置 SelectBuilder 的状态（正如 BEGIN()或 RESET()方法被调用）。因此，这个方法只能被调用一次！

SqlBuilder

和 SelectBuilder 相似，MyBatis 也包含一个一般性的 SqlBuilder。它包含 SelectBuilder 的所有方法，还有构建 insert，update 和 delete 的方法。在 DeleteProvider，InsertProvider 或 UpdateProvider 中（还有 SelectProvider）构建 SQL 字符串时这个类就很有用。

在上述示例中要使用 SqlBuilder，你只需简单静态引入如下内容：

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

SqlBuilder 包含 SelectBuilder 中的所有方法，还有下面这些额外的方法：

方法	描述
DELETE_FROM(String)	开始一个 delete 语句，要指定删除的表。通常它后面要跟着一个 WHERE 语句！
INSERT_INTO(String)	开始一个 insert 语句，要指定插入的表。它的后面要跟着一个或多个 VALUES()调用。
SET(String)	为更新语句附加“set”内容的列表。
UPDATE(String)	开始一个 update 语句，要指定更新的表。它的后面要跟着一个或多个 SET()调用，通常需要一个 WHERE()调用。
VALUES(String,String)	附加到 insert 语句后。第一个参数是要插入的列名，第二个参数是要插入的值。

这里是一些示例：

```
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();
}
```