入门

安装

要使用 MyBatis， 只需将 mybatis-x.x.x.jar (https://github.com/mybatis/mybatis-3/releases)　文件置于 classpath 中即可。

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

```
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>x.x.x</version>
</dependency>
```

从 **XML** 中构建 **SqlSessionFactory**

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

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

```
String resource = "org/mybatis/example/mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
```

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

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

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

不使用 XML 构建 SqlSessionFactory

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

```java
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);
```

注意该例中，configuration 添加了一个映射器类（mapper class）。映射器类是 Java 类，它们包含 SQL 映射语句的注解从而避免了 XML 文件的依赖。不过，由于 Java 注解的一些限制加之某些 MyBatis 映射的复杂性，XML 映射对于大多数高级映射（比如：嵌套 Join 映射）来说仍然是必须的。有鉴于此，如果存在一个对等的 XML 配置文件的话，MyBatis 会自动查找并加载它（这种情况下， BlogMapper.xml 将会基于类路径和 BlogMapper.class 的类名被加载进来）。具体细节稍后讨论。

从 SqlSessionFactory 中获取 SqlSession

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

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

诚然这种方式能够正常工作，并且对于使用旧版本 MyBatis 的用户来说也比较熟悉，不过现在有了一种更直白的方式。使用对于给定语句能够合理描述参数和返回值的接口（比如说BlogMapper.class），你现在不但可以执行更清晰和类型安全的代码，而且还不用担心易错的字符串字面值以及强制类型转换。

例如：

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

现在我们来探究一下这里到底是怎么执行的。

探究已映射的 SQL 语句

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

在上面提到的两个例子中，一个语句应该是通过 XML 定义，而另外一个则是通过注解定义。先看 XML 定义这个，事实上 MyBatis 提供的全部特性可以利用基于 XML 的映射语言来实现，这使得 MyBatis 在过去的数年间得以流行。如果你以前用过 MyBatis，这个概念应该会比较熟悉。不过 XML 映射文件已经有了很多的改进，随着文档的进行会愈发清晰。这里给出一个基于 XML 映射语句的示例，它应该可以满足上述示例中 SqlSession 的调用。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="org.mybatis.example.BlogMapper">
  <select id="selectBlog" resultType="Blog">
    select * from Blog where id = #{id}
  </select>
</mapper>
```

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

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

你可能注意到这和使用完全限定名调用 Java 对象的方法是相似的，之所以这样做是有原因的。这个命名可以直接映射到在命名空间中同名的 Mapper 类，并将已映射的 select 语句中的名字、参数和返回类型匹配成方法。这样你就可以像上面那样很容易地调用这个对应 Mapper 接口的方法。不过让我们再看一遍下面的例子：

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

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

提示 命名空间的一点注释

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

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

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

- 完全限定名（比如“com.mypackage.MyMapper.selectAllThings”）将被直接查找并且找到即用。
- 短名称（比如“selectAllThings”）如果全局唯一也可以作为一个单独的引用。如果不唯一，有两个或两个以上的相同名称（比如“com.foo.selectAllThings ”和“com.bar.selectAllThings”），那么使用时就会收到错误报告说短名称是不唯一的，这种情况下就必须使用完全限定名。

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

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

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

选择何种方式以及映射语句的定义的一致性对你来说有多重要这些完全取决于你和你的团队。换句话说，永远不要拘泥于一种方式，你可以很轻松的在基于注解和 XML 的语句映射方式间自由移植和切换。

作用域（**Scope**）和生命周期

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

提示 对象生命周期和依赖注入框架

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

SqlSessionFactoryBuilder

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

SqlSessionFactory

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

SqlSession

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

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

在你的所有的代码中一致性地使用这种模式来保证所有数据库资源都能被正确地关闭。

映射器实例（Mapper Instances）

映射器是一个你创建来绑定你映射的语句的接口。映射器接口的实例是从 SqlSession 中获得的。因此从技术层面讲，任何映射器实例的最大作用域是和请求它们的 SqlSession 相同的。尽管如此，映射器实例的最佳作用域是方法作用域。也就是说，映射器实例应该在调用它们的方法中被请求，用过之后即可废弃。并不需要显式地关闭映射器实例，尽管在整个请求作用域（request scope）保持映射器实例也不会有什么问题，但是很快你会发现，像 SqlSession 一样，在这个作用域上管理太多的资源的话会难于控制。所以要保持简单，最好把映射器放在方法作用域（method scope）内。下面的示例就展示了这个实践：

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