

# 入门

## 安装

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

如果使用 Maven 来构建项目，则需将下面的依赖代码置于 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 的实例为核心的。SqlSessionFactory 的实例可以通过 SqlSessionFactoryBuilder 获得。而 SqlSessionFactoryBuilder 则可以从 XML 配置文件或一个预先配置的 Configuration 实例来构建出 SqlSessionFactory 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。但也可以使用任意的输入流 (InputStream) 实例，比如用文件路径字符串或 file:// URL 构造的输入流。MyBatis 包含一个名叫 Resources 的工具类，它包含一些实用方法，使得从类路径或其它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 配置文件中包含了对 MyBatis 系统的核心设置，包括获取数据库连接实例的数据源 (DataSource) 以及决定事务作用域和控制方式的事务管理器 (TransactionManager)。后面会再探讨 XML 配置文件的详细内容，这里先给出一个简单的示例：

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

当然，还有很多可以在 XML 文件中配置的选项，上面的示例仅罗列了最关键的部分。注意 XML 头部的声明，它用来验证 XML 文档的正确性。environment 元素体中包含了事务管理和连接池的配置。mappers 元素则包含了一组映射器（mapper），这些映射器的 XML 映射文件包含了 SQL 代码和映射定义信息。

## 不使用 XML 构建 SqlSessionFactory

如果你更愿意直接从 Java 代码而不是 XML 文件中创建配置，或者想要创建你自己的配置建造器，MyBatis 也提供了完整的配置类，提供了所有与 XML 文件等价的配置项。

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

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 映射注解从而避免依赖 XML 文件。不过，由于 Java 注解的一些限制以及某些 MyBatis 映射的复杂性，要使用大多数高级映射（比如：嵌套联合映射），仍然需要使用 XML 配置。有鉴于此，如果存在一个同名 XML 配置文件，MyBatis 会自动查找并加载它（在这个例子中，基于类路径和 BlogMapper.class 的类名，会加载 BlogMapper.xml）。具体细节稍后讨论。

## 从 SqlSessionFactory 中获取 SqlSession

既然有了 SqlSessionFactory，顾名思义，我们可以从中获得 SqlSession 的实例。SqlSession 提供了在数据库执行 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：

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

诚然，这种方式能够正常工作，对使用旧版本 MyBatis 的用户来说也比较熟悉。但现在有了一种更简洁的方式——使用和指定语句的参数和返回值相匹配的接口（比如 `BlogMapper.class`），现在你的代码不仅更清晰，更加类型安全，还不用担心可能出错的字符串字面值以及强制类型转换。

例如：

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

现在我们来探究一下这段代码究竟做了些什么。

## 探究已映射的 SQL 语句

现在你可能很想知道 `SqlSession` 和 `Mapper` 到底具体执行了些什么操作，但 SQL 语句映射是个相当广泛的话题，可能会占去文档的大部分篇幅。但为了让你能够了解个大概，这里会给出几个例子。

在上面提到的例子中，一个语句既可以通过 XML 定义，也可以通过注解定义。我们先看看 XML 定义语句的方式，事实上 MyBatis 提供的所有特性都可以利用基于 XML 的映射语言来实现，这使得 MyBatis 在过去的数年间得以流行。如果你用过旧版本的 MyBatis，你应该对这个概念比较熟悉。但相比于之前的版本，新版本改进了许多 XML 的配置，后面我们会提到这些改进。这里给出一个基于 XML 映射语句的示例，它应该可以满足上个示例中 `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>
```

为了这个简单的例子，我们似乎写了不少配置，但其实并不多。在一个 XML 映射文件中，可以定义无数个映射语句，这样一来，XML 头部和文档类型声明部分就显得微不足道了。文档的其它部分很直白，容易理解。它在命名空间 “`org.mybatis.example.BlogMapper`” 中定义了一个名为 “`selectBlog`” 的映射语句，这样你就可以用全限定名 “`org.mybatis.example.BlogMapper.selectBlog`” 来调用映射语句了，就像上面例子中那样：

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

你可能会注意到，这种方式和用全限定名调用 Java 对象的方法类似。这样，该命名就可以直接映射到在命名空间中同名的映射器类，并将已映射的 `select` 语句匹配到对应名称、参数和返回类型的方法。因此你就可以像上面那样，不费吹灰之力地在对应的映射器接口调用方法，就像下面这样：

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

第二种方法有很多优势，首先它不依赖于字符串字面值，会更安全一点；其次，如果你的 IDE 有代码补全功能，那么代码补全可以帮你快速选择到映射好的 SQL 语句。

### 提示 对命名空间的一点补充

在之前版本的 MyBatis 中，**命名空间 (Namespaces)** 的作用并不大，是可选的。但现在，随着命名空间越发重要，你必须指定命名空间。

命名空间的作用有两个，一个是利用更长的全限定名来将不同的语句隔离开来，同时也实现了你上面见到的接口绑定。就算你觉得暂时用不到接口绑定，你也应该遵循这里的规定，以防哪天你改变了主意。长远来看，只要将命名空间置于合适的 Java 包命名空间之中，你的代码会变得更加整洁，也有利于你更方便地使用 MyBatis。

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

- 全限定名（比如 “com.mypackage.MyMapper.selectAllThings”）将被直接用于查找及使用。
- 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”），那么使用时就会产生“短名称不唯一”的错误，这种情况下就必须使用全限定名。

对于像 BlogMapper 这样的映射器类来说，还有另一种方法来完成语句映射。它们映射的语句可以不用 XML 来配置，而可以使用 Java 注解来配置。比如，上面的 XML 示例可以被替换成如下的配置：

```
package org.mybatis.example;

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

使用注解来映射简单语句会使代码显得更加简洁，但对于稍微复杂一点的语句，Java 注解不仅力不从心，还会让你本就复杂的 SQL 语句更加混乱不堪。因此，如果你需要做一些很复杂的操作，最好用 XML 来映射语句。

选择何种方式来配置映射，以及认为是否应该要统一映射语句定义的形式，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

理解我们之前讨论过的不同作用域和生命周期类别是至关重要的，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，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，但最好还是不要一直保留着它，以保证所有的 XML 解析资源可以被释放给更重要的事情。

## SqlSessionFactory

SqlSessionFactory 一旦被创建就应该在应用的运行期间一直存在，没有任何理由丢弃它或重新创建另一个实例。使用 SqlSessionFactory 的最佳实践是在应用运行期间不要重复创建多次，多次重建 SqlSessionFactory 被视为一种代码“坏习惯”。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托管作用域中，比如 `Servlet` 框架中的 `HttpSession`。如果你现在正在使用一种 `Web` 框架，考虑将 `SqlSession` 放在一个和 `HTTP` 请求相似的作用域中。换句话说，每次收到 `HTTP` 请求，就可以打开一个 `SqlSession`，返回一个响应后，就关闭它。这个关闭操作很重要，为了确保每次都能执行关闭操作，你应该把这个关闭操作放到 `finally` 块中。下面的示例就是一个确保 `SqlSession` 关闭的标准模式：

```
try (SqlSession session = sqlSessionFactory.openSession()) {  
    // 你的应用逻辑代码  
}
```

在所有代码中都遵循这种使用模式，可以保证所有数据库资源都能被正确地关闭。

## 映射器实例

映射器是一些绑定映射语句的接口。映射器接口的实例是从 `SqlSession` 中获得的。虽然从技术层面上来讲，任何映射器实例的最大作用域与请求它们的 `SqlSession` 相同。但方法作用域才是映射器实例的最合适的作用域。也就是说，映射器实例应该在调用它们的方法中被获取，使用完毕之后即可丢弃。映射器实例并不需要被显式地关闭。尽管在整个请求作用域保留映射器实例不会有什么问题，但是你很快会发现，在这个作用域上管理太多像 `SqlSession` 的资源会让你忙不过来。因此，最好将映射器放在方法作用域内。就像下面的例子一样：

```
try (SqlSession session = sqlSessionFactory.openSession()) {  
    BlogMapper mapper = session.getMapper(BlogMapper.class);  
    // 你的应用逻辑代码  
}
```