# 简介

## 什么是 MyBatis？

MyBatis 是一款优秀的持久层框架，它支持自定义 SQL、存储过程以及高级映射。MyBatis 免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作。MyBatis 可以通过简单的 XML 或注解来配置和映射原始类型、接口和 Java POJO（Plain Old Java Objects，普通老式 Java 对象）为数据库中的记录。

# 入门

## 使用的Jar

1. mybatis-xx.xx.jar,将jar包放到classpath路径下
2. 使用maven，groupid🡪org.mybatis artfactId mybatis 指定version

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>x.x.x</version>

</dependency>

除了mybatis自身的jar之外还需要jdbc相关的jar，mysql-connector-java-5.1.8-bin.jar（版本自选）

## 从XML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。

示例SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream); inputStream是配置文件的输入流

## 从 SqlSessionFactory 中获取 SqlSession

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：

SqlSession session = sqlSessionFactory.openSession()

另一种更简洁的方式——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 = sqlSessionFactory.openSession()) {

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

Blog blog = mapper.selectBlog(101);

}

## 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。

## 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

}

# XML配置

MyBatis 的配置文件包含了会深深影响 MyBatis 行为的设置和属性信息。 配置文档的顶层结构如下：

* configuration（配置）
  + [properties（属性）](https://mybatis.org/mybatis-3/zh/configuration.html#properties)
  + [settings（设置）](https://mybatis.org/mybatis-3/zh/configuration.html#settings)
  + [typeAliases（类型别名）](https://mybatis.org/mybatis-3/zh/configuration.html#typeAliases)
  + [typeHandlers（类型处理器）](https://mybatis.org/mybatis-3/zh/configuration.html#typeHandlers)
  + [objectFactory（对象工厂）](https://mybatis.org/mybatis-3/zh/configuration.html#objectFactory)
  + [plugins（插件）](https://mybatis.org/mybatis-3/zh/configuration.html#plugins)
  + [environments（环境配置）](https://mybatis.org/mybatis-3/zh/configuration.html#environments)
    - environment（环境变量）
      * transactionManager（事务管理器）
      * dataSource（数据源）
  + [databaseIdProvider（数据库厂商标识）](https://mybatis.org/mybatis-3/zh/configuration.html#databaseIdProvider)
  + [mappers（映射器）](https://mybatis.org/mybatis-3/zh/configuration.html#mappers)

## 属性

Properties标签

Value的来源：

1. property的value属性直接赋值

<properties >

<property name="username" value="dev\_user"/>

<property name="password" value="F2Fa3!33TYyg"/>

</properties>

1. 从properties文件中获取value

通过properties的resource加载配置文件

<properties resource="org/mybatis/example/config.properties">

</properties>

使用时用 ${key}来获取值

1. 通过SqlSessionFactoryBuilder.build() 方法中传入属性值；SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, props);

定义properties类，设置属性，通过SqlSessionFactoryBuilder的build方法传入属性

如果一个属性在不只一个地方进行了配置，那么，MyBatis 将按照下面的顺序来加载：

首先读取在 properties 元素体内指定的属性。

然后根据 properties 元素中的 resource 属性读取类路径下属性文件，或根据 url 属性指定的路径读取属性文件，并覆盖之前读取过的同名属性。

最后读取作为方法参数传递的属性，并覆盖之前读取过的同名属性。

因此，通过方法参数传递的属性具有最高优先级，resource/url 属性中指定的配置文件次之，最低优先级的则是 properties 元素中指定的属性。

## 设置

用来调整mybatis的属性，可以改变mybatis的行为

<settings>

<setting name="cacheEnabled" value="true"/>

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

<setting name="multipleResultSetsEnabled" value="true"/>

<setting name="useColumnLabel" value="true"/>

<setting name="useGeneratedKeys" value="false"/>

<setting name="autoMappingBehavior" value="PARTIAL"/>

<setting name="autoMappingUnknownColumnBehavior" value="WARNING"/>

<setting name="defaultExecutorType" value="SIMPLE"/>

<setting name="defaultStatementTimeout" value="25"/>

<setting name="defaultFetchSize" value="100"/>

<setting name="safeRowBoundsEnabled" value="false"/>

<setting name="mapUnderscoreToCamelCase" value="false"/>

<setting name="localCacheScope" value="SESSION"/>

<setting name="jdbcTypeForNull" value="OTHER"/>

<setting name="lazyLoadTriggerMethods" value="equals,clone,hashCode,toString"/>

</settings>

## 类型别名

类型别名可为 Java 类型设置一个缩写名字。 它仅用于 XML 配置，意在降低冗余的全限定类名书写

格式

<typeAliases>

<typeAlias alias="别名" type="全类名"/>

</typeAliases>

也可以指定一个包名，MyBatis 会在包名下面搜索需要的 Java Bean

<typeAliases>

<package name="domain.blog"/>

</typeAliases>

每一个在包 中的 Java Bean，在没有注解的情况下，会使用 Bean 的首字母小写的非限定类名来作为它的别名。 比如 domain.blog.Author 的别名为 author；若有注解@Alias(“”)，则别名为其注解值。

## 类型处理器

MyBatis 在设置预处理语句（PreparedStatement）中的参数或从结果集中取出一个值时， 都会用类型处理器将获取到的值以合适的方式转换成 Java 类型

自定义类型处理器类实现将获取到的值转换为指定的类型，具体做法：实现 org.apache.ibatis.type.TypeHandler 接口，或继承一个很便利的类 org.apache.ibatis.type.BaseTypeHandler， 并且可以（可选地）将它映射到一个 JDBC 类型 ；在自定义的类上使用@MappedJdbcTypes注解，指定要转换的jdbc类型（这个是可选择的）

要注意 MyBatis 不会通过检测数据库元信息来决定使用哪种类型，所以你必须在参数和结果映射中指明字段是 VARCHAR 类型， 以使其能够绑定到正确的类型处理器上。这是因为 MyBatis 直到语句被执行时才清楚数据类型。

通过类型处理器的泛型，MyBatis 可以得知该类型处理器处理的 Java 类型，不过这种行为可以通过两种方法改变：

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

<typeHandlers>

<typeHandler handler="mybatis.handler.TypeHandler" javaType="" jdbcType=""/>

</typeHandlers>

2.在类型处理器的类上增加一个 @MappedTypes 注解指定与其关联的 Java 类型列表。 如果在 javaType 属性中也同时指定，则注解上的配置将被忽略。

可以通过两种方式来指定关联的 JDBC 类型：

在类型处理器的配置元素上增加一个 jdbcType 属性（比如：jdbcType="VARCHAR"）；

在类型处理器的类上增加一个 @MappedJdbcTypes 注解指定与其关联的 JDBC 类型列表。 如果在 jdbcType 属性中也同时指定，则注解上的配置将被忽略。

## 对象工厂

每次 MyBatis 创建结果对象的新实例时，它都会使用一个对象工厂（ObjectFactory）实例来完成实例化工作。 默认的对象工厂需要做的仅仅是实例化目标类，要么通过默认无参构造方法，要么通过存在的参数映射来调用带有参数的构造方法。 如果想覆盖对象工厂的默认行为，可以通过创建自己的对象工厂来实现

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

## 插件

MyBatis 允许你在映射语句执行过程中的某一点进行拦截调用。默认情况下，MyBatis 允许使用插件来拦截的方法调用包括：

Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)

ParameterHandler (getParameterObject, setParameters)

ResultSetHandler (handleResultSets, handleOutputParameters)

StatementHandler (prepare, parameterize, batch, update, query)

实现插件只需要实现Interceptor接口，使用@Intercepts注解来指定拦截的方法（也就是接口Executor，ParameterHandler，ResultSetHandler，StatementHandler的实现类重写的方法）

例如:

@Intercepts(

{ @Signature(type = Executor.class, method = "query", args =

{ MappedStatement.class, Object.class, RowBounds.class, ResultHandler.class }) })

XML配置示例

<plugins>

<plugin interceptor="org.mybatis.example.ExamplePlugin">

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

</plugin>

</plugins>

## 环境配置

配置数据库环境，可以配置多个环境，有利于将sql映射应用于多种数据库，但是每个sqlsessionFactory只能对应一个环境

环境变量的配置

<!--默认环境的id default的值：development（开发模式）、work(工作模式) -->

<environments default=*"development"*>

<!-- 环境id，默认环境的id和环境id必须相同 -->

<environment id=*"development"*>

<!-- 配置控制事务的方式 ，有2中方式JDBC和MANAGED -->

<!-- jdbc 使用jdbc的提交和事物回滚方式-->

<!-- managed：这个配置是将事务管理交给其他容器来管理，比如spring -->

<!-- 默认情况下他会关闭连接，然而一些容器并不希望这样，因此需要将closeConnection属性设置为false来阻止它默认的关闭行为 -->

<transactionManager type=*"JDBC"*/>

<!-- 配置连接数据库实例的数据源 -->

<!-- 三种内建的数据源类型：type=[ UNPOOLED | POOLED | JNDI ] -->

<!-- UNPOOLED：表示每次请求时打开和关闭连接，适合比较简单的应用程序 -->

<!-- POOLED 类似连接池这个概念 -->

<!-- JDNI- 这个数据源的实现是为了能在如EJB或应用服务器这类容器中使用，-->

<!-- 容器可以集中在外部配置数据源，然后放置一个JDNI上下文的引用 -->

<!-- 这种数据源只需要两个属性：

initial\_context - 这个属性用来InitailContext中寻找上下文（即，initialContext.lookup(initial\_context)）。

这是个可选属性，如果忽略，那么 data\_source 属性将会直接从 InitialContext 中寻找

data\_source - 这是引用数据源实例位置的上下文的路径。

提供了 initial\_context 配置时会在其返回的上下文中进行查找，没有提供时则直接在 InitialContext 中查找

和其他数据源配置类似 -->

<dataSource type=*"POOLED"*>

<property name=*"driver"* value=*"${driver}"*/>

<property name=*"url"* value=*"${url}"*/>

<!-- <property name="driver" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql:///test"/>

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

<property name="password" value="12345"/> -->

<property name=*"username"* value=*"${username}"*/>

<property name=*"password"* value=*"${password}"*/>

</dataSource>

</environment>

</environments>

当使用spring+mybatis时，就没有必要配置环境信息，这个可以在spring中统一配置

## 数据库厂商标识

## 映射器

映射器就是将定义好的映射文件告诉mybatis去找到想要的sql,下面就是使用方式

<!-- 使用相对于类路径的资源引用 -->

<mappers>

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

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

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

</mappers>

<!-- 使用完全限定资源定位符（URL） -->

<mappers>

<mapper url="file:///var/mappers/AuthorMapper.xml"/>

<mapper url="file:///var/mappers/BlogMapper.xml"/>

<mapper url="file:///var/mappers/PostMapper.xml"/>

</mappers>

<!-- 使用映射器接口实现类的完全限定类名 -->

<mappers>

<mapper class="org.mybatis.builder.AuthorMapper"/>

<mapper class="org.mybatis.builder.BlogMapper"/>

<mapper class="org.mybatis.builder.PostMapper"/>

</mappers>

<!-- 将包内的映射器接口实现全部注册为映射器 -->

<mappers>

<package name="org.mybatis.builder"/>

</mappers>

# XML 映射器

MyBatis 的真正强大在于它的语句映射，这是它的魔力所在。由于它的异常强大，映射器的 XML 文件就显得相对简单。如果拿它跟具有相同功能的 JDBC 代码进行对比，你会立即发现省掉了将近 95% 的代码。MyBatis 致力于减少使用成本，让用户能更专注于 SQL 代码。

SQL 映射文件只有很少的几个顶级元素（按照应被定义的顺序列出）：

cache – 该命名空间的缓存配置。

cache-ref – 引用其它命名空间的缓存配置。

resultMap – 描述如何从数据库结果集中加载对象，是最复杂也是最强大的元素。

parameterMap – 老式风格的参数映射。此元素已被废弃，并可能在将来被移除！请使用行内参数映射。文档中不会介绍此元素。

sql – 可被其它语句引用的可重用语句块。

insert – 映射插入语句。

update – 映射更新语句。

delete – 映射删除语句。

select – 映射查询语句。

## Select

查询语句是 MyBatis 中最常用的元素之一

| **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的类全限定名或别名。这个属性是可选的，  因为 MyBatis 可以通过类型处理器（TypeHandler）推断出具体传入语句的  参数，默认值为未设置（unset）。 |
| ~~parameterMap~~ | ~~用于引用外部 parameterMap 的属性，目前已被废弃。请使用行内参数~~  ~~映射和 parameterType 属性。~~ |
| resultType | 期望从这条语句中返回结果的类全限定名或别名。 注意，如果返回的是集合，  那应该设置为集合包含的类型，而不是集合本身的类型。  resultType 和 resultMap 之间只能同时使用一个。 |
| resultMap | 对外部 resultMap 的命名引用。结果映射是 MyBatis 最强大的特性，  如果你对其理解透彻，许多复杂的映射问题都能迎刃而解。  resultType 和 resultMap 之间只能同时使用一个。 |
| flushCache | 将其设置为 true 后，只要语句被调用，都会导致本地缓存和二级缓存被清空，  默认值：false。 |
| useCache | 将其设置为 true 后，将会导致本条语句的结果被二级缓存缓存起来，默认值：  对 select 元素为 true。 |
| timeout | 这个设置是在抛出异常之前，驱动程序等待数据库返回请求结果的秒数。  默认值为未设置（unset）（依赖数据库驱动）。 |
| fetchSize | 这是一个给驱动的建议值，尝试让驱动程序每次批量返回的结果行数等于这个设置值。  默认值为未设置（unset）（依赖驱动）。 |
| statementType | 可选 STATEMENT，PREPARED 或 CALLABLE。这会让 MyBatis 分别使用  Statement，PreparedStatement 或 CallableStatement，默认值：PREPARED。 |
| resultSetType | FORWARD\_ONLY，SCROLL\_SENSITIVE, SCROLL\_INSENSITIVE 或 DEFAULT  （等价于 unset） 中的一个，默认值为 unset （依赖数据库驱动）。 |
| databaseId | 如果配置了数据库厂商标识（databaseIdProvider），MyBatis 会加载所有不带  databaseId 或匹配当前 databaseId 的语句；如果带和不带的语句都有，则不带的会被忽略。 |
| resultOrdered | 这个设置仅针对嵌套结果 select 语句：如果为 true，将会假设包含了嵌套结果集  或是分组，当返回一个主结果行时，就不会产生对前面结果集的引用。  这就使得在获取嵌套结果集的时候不至于内存不够用。默认值：false。 |
| resultSets | 这个设置仅适用于多结果集的情况。它将列出语句执行后返回的结果集并  赋予每个结果集一个名称，多个名称之间以逗号分隔。 |

## insert, update 和 delete

## sql

## 参数

## 结果映射