**MyBatis XML配置**

由 毛泽萝拉 创建， 最后一次修改 2016-08-12

**XML 映射配置文件**

MyBatis 的配置文件包含了影响 MyBatis 行为甚深的设置（settings）和属性（properties）信息。

**properties**

这些属性都是可外部配置且可动态替换的，既可以在典型的 Java 属性文件中配置，亦可通过 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来替换需要动态配置的属性值。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 元素中设置的相应值来替换。 driver 和 url 属性将会由 config.properties 文件中对应的值来替换。这样就为配置提供了诸多灵活选择。

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：

* 在 properties 元素体内指定的属性首先被读取。
* 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，并覆盖已读取的同名属性。
* 最后读取作为方法参数传递的属性，并覆盖已读取的同名属性。

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，最低优先级的是 properties 属性中指定的属性。

**settings**

这是 MyBatis 中极为重要的调整设置，它们会改变 MyBatis 的运行时行为。下表描述了设置中各项的意图、默认值等。

| **设置参数** | **描述** | **有效值** | **默认值** |
| --- | --- | --- | --- |
| cacheEnabled | 该配置影响的所有映射器中配置的缓存的全局开关。 | true,false | true |
| lazyLoadingEnabled | 延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。 特定关联关系中可通过设置fetchType属性来覆盖该项的开关状态。 | true,false | false |
| aggressiveLazyLoading | 当启用时，对任意延迟属性的调用会使带有延迟加载属性的对象完整加载；反之，每种属性将会按需加载。 | true,false,true |  |
| multipleResultSetsEnabled | 是否允许单一语句返回多结果集（需要兼容驱动）。 | true,false | true |
| useColumnLabel | 使用列标签代替列名。不同的驱动在这方面会有不同的表现， 具体可参考相关驱动文档或通过测试这两种不同的模式来观察所用驱动的结果。 | true,false | true |
| useGeneratedKeys | 允许 JDBC 支持自动生成主键，需要驱动兼容。 如果设置为 true 则这个设置强制使用自动生成主键，尽管一些驱动不能兼容但仍可正常工作（比如 Derby）。 | true,false | False |
| autoMappingBehavior | 指定 MyBatis 应如何自动映射列到字段或属性。 NONE 表示取消自动映射；PARTIAL 只会自动映射没有定义嵌套结果集映射的结果集。 FULL 会自动映射任意复杂的结果集（无论是否嵌套）。 | NONE, PARTIAL, FULL | PARTIAL |
| defaultExecutorType | 配置默认的执行器。SIMPLE 就是普通的执行器；REUSE 执行器会重用预处理语句（prepared statements）； BATCH 执行器将重用语句并执行批量更新。 | SIMPLE REUSE BATCH | SIMPLE |
| defaultStatementTimeout | 设置超时时间，它决定驱动等待数据库响应的秒数。 | Any positive integer | Not Set (null) |
| safeRowBoundsEnabled | 允许在嵌套语句中使用分页（RowBounds）。 | true,false | False |
| mapUnderscoreToCamelCase | 是否开启自动驼峰命名规则（camel case）映射，即从经典数据库列名 A\_COLUMN 到经典 Java 属性名 aColumn 的类似映射。 | true, false | False |
| localCacheScope | MyBatis 利用本地缓存机制（Local Cache）防止循环引用（circular references）和加速重复嵌套查询。 默认值为 SESSION，这种情况下会缓存一个会话中执行的所有查询。 若设置值为 STATEMENT，本地会话仅用在语句执行上，对相同 SqlSession 的不同调用将不会共享数据。 | SESSION,STATEMENT | SESSION |
| jdbcTypeForNull | 当没有为参数提供特定的 JDBC 类型时，为空值指定 JDBC 类型。 某些驱动需要指定列的 JDBC 类型，多数情况直接用一般类型即可，比如 NULL、VARCHAR 或 OTHER。 | JdbcType enumeration. Most common are: NULL, VARCHAR and OTHER | OTHER |
| lazyLoadTriggerMethods | 指定哪个对象的方法触发一次延迟加载。 | A method name list separated by commas | equals,clone,hashCode,toString |
| defaultScriptingLanguage | 指定动态 SQL 生成的默认语言。 | A type alias or fully qualified class name. | org.apache.ibatis.scripting.xmltags.XMLDynamicLanguageDriver |
| callSettersOnNulls | 指定当结果集中值为 null 的时候是否调用映射对象的 setter（map 对象时为 put）方法，这对于有 Map.keySet() 依赖或 null 值初始化的时候是有用的。注意基本类型（int、boolean等）是不能设置成 null 的。 | true,false | false |
| logPrefix | 指定 MyBatis 增加到日志名称的前缀。 | Any String | Not set |
| logImpl | 指定 MyBatis 所用日志的具体实现，未指定时将自动查找。 | SLF4J,LOG4J,LOG4J2,JDK\_LOGGING,COMMONS\_LOGGING,STDOUT\_LOGGING,NO\_LOGGING | Not set |
| proxyFactory | 指定 Mybatis 创建具有延迟加载能力的对象所用到的代理工具。 | CGLIB JAVASSIST | CGLIB |

一个配置完整的 settings 元素的示例如下：

<**settings**>

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

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

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

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

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

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

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

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

<**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**>

**typeAliases**

类型别名是为 Java 类型设置一个短的名字。它只和 XML 配置有关，存在的意义仅在于用来减少类完全限定名的冗余。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>

当这样配置时，Blog可以用在任何使用domain.blog.Blog的地方。

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

<typeAliases>

<**package** name="domain.blog"/>

</**typeAliases**>

每一个在包 domain.blog 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，则别名为其注解值。看下面的例子：

@Alias("author")

**public** **class** **Author** {

...

}

已经为许多常见的 Java 类型内建了相应的类型别名。它们都是大小写不敏感的，需要注意的是由基本类型名称重复导致的特殊处理。

| **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（PreparedStatement）中设置一个参数时，还是从结果集中取出一个值时， 都会用类型处理器将获取的值以合适的方式转换成 Java 类型。下表描述了一些默认的类型处理器。

| **类型处理器** | **Java 类型** | **JDBC 类型** |
| --- | --- | --- |
| BooleanTypeHandler | java.lang.Boolean, boolean | 数据库兼容的 BOOLEAN |
| ByteTypeHandler | java.lang.Byte, byte | 数据库兼容的 NUMERIC 或 BYTE |
| ShortTypeHandler | java.lang.Short, short | 数据库兼容的 NUMERIC 或 SHORT INTEGER |
| IntegerTypeHandler | java.lang.Integer, int | 数据库兼容的 NUMERIC 或 INTEGER |
| LongTypeHandler | java.lang.Long, long | 数据库兼容的 NUMERIC 或 LONG INTEGER |
| FloatTypeHandler | java.lang.Float, float | 数据库兼容的 NUMERIC 或 FLOAT |
| DoubleTypeHandler | java.lang.Double, double | 数据库兼容的 NUMERIC 或 DOUBLE |
| BigDecimalTypeHandler | java.math.BigDecimal | 数据库兼容的 NUMERIC 或 DECIMAL |
| StringTypeHandler | java.lang.String | CHAR, VARCHAR |
| ClobTypeHandler | java.lang.String | CLOB, LONGVARCHAR |
| NStringTypeHandler | java.lang.String | NVARCHAR, NCHAR |
| NClobTypeHandler | java.lang.String | NCLOB |
| ByteArrayTypeHandler | byte[] | 数据库兼容的字节流类型 |
| BlobTypeHandler | byte[] | BLOB, LONGVARBINARY |
| DateTypeHandler | java.util.Date | TIMESTAMP |
| DateOnlyTypeHandler | java.util.Date | DATE |
| TimeOnlyTypeHandler | java.util.Date | TIME |
| SqlTimestampTypeHandler | java.sql.Timestamp | TIMESTAMP |
| SqlDateTypeHandler | java.sql.Date | DATE |
| SqlTimeTypeHandler | java.sql.Time | TIME |
| ObjectTypeHandler | Any | OTHER 或未指定类型 |
| EnumTypeHandler | Enumeration Type | VARCHAR-任何兼容的字符串类型，存储枚举的名称（而不是索引） |
| EnumOrdinalTypeHandler | Enumeration Type | 任何兼容的 NUMERIC 或 DOUBLE 类型，存储枚举的索引（而不是名称）。 |

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。 具体做法为：实现 org.apache.ibatis.type.TypeHandler 接口， 或继承一个很便利的类 org.apache.ibatis.type.BaseTypeHandler， 然后可以选择性地将它映射到一个 JDBC 类型。比如：

// ExampleTypeHandler.java

@MappedJdbcTypes(JdbcType.VARCHAR)

**public** **class** **ExampleTypeHandler** **extends** **BaseTypeHandler** {

@Override

**public** **void** **setNonNullParameter**(PreparedStatement ps, **int** i, String parameter, JdbcType jdbcType) **throws** SQLException {

ps.setString(i, parameter);

}

@Override

**public** String **getNullableResult**(ResultSet rs, String columnName) **throws** SQLException {

**return** rs.getString(columnName);

}

@Override

**public** String **getNullableResult**(ResultSet rs, **int** columnIndex) **throws** SQLException {

**return** rs.getString(columnIndex);

}

@Override

**public** String **getNullableResult**(CallableStatement cs, **int** columnIndex) **throws** SQLException {

**return** cs.getString(columnIndex);

}

}

<!-- mybatis-config.xml -->

<**typeHandlers**>

<**typeHandler** handler="org.mybatis.example.ExampleTypeHandler"/>

</**typeHandlers**>

使用这个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 不会窥探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那是 VARCHAR 类型的字段， 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：

* 在类型处理器的配置元素（typeHandler element）上增加一个 javaType 属性（比如：javaType="String"）；
* 在类型处理器的类上（TypeHandler class）增加一个 @MappedTypes 注解来指定与其关联的 Java 类型列表。 如果在 javaType 属性中也同时指定，则注解方式将被忽略。

可以通过两种方式来指定被关联的 JDBC 类型：

* 在类型处理器的配置元素上增加一个 javaType 属性（比如：javaType="VARCHAR"）；
* 在类型处理器的类上（TypeHandler class）增加一个 @MappedJdbcTypes 注解来指定与其关联的 JDBC 类型列表。 如果在 javaType 属性中也同时指定，则注解方式将被忽略。

最后，可以让 MyBatis 为你查找类型处理器：

<!-- mybatis-config.xml -->

<**typeHandlers**>

<**package** name="org.mybatis.example"/>

</**typeHandlers**>

注意在使用自动检索（autodiscovery）功能的时候，只能通过注解方式来指定 JDBC 的类型。

你能创建一个泛型类型处理器，它可以处理多于一个类。为达到此目的， 需要增加一个接收该类作为参数的构造器，这样在构造一个类型处理器的时候 MyBatis 就会传入一个具体的类。

//GenericTypeHandler.java

**public** **class** **GenericTypeHandler** **extends** **BaseTypeHandler** {

**private** Class type;

**public** **GenericTypeHandler**(Class type) {

**if** (type == **null**) **throw** **new** IllegalArgumentException("Type argument cannot be null");

**this**.type = type;

}

...

EnumTypeHandler 和 EnumOrdinalTypeHandler 都是泛型类型处理器（generic TypeHandlers）， 我们将会在接下来的部分详细探讨。

**处理枚举类型**

若想映射枚举类型 Enum，则需要从 EnumTypeHandler 或者 EnumOrdinalTypeHandler 中选一个来使用。

比如说我们想存储取近似值时用到的舍入模式。默认情况下，MyBatis 会利用 EnumTypeHandler 来把 Enum 值转换成对应的名字。

**注意 EnumTypeHandler 在某种意义上来说是比较特别的，其他的处理器只针对某个特定的类，而它不同，它会处理任意继承了 Enum 的类。**

不过，我们可能不想存储名字，相反我们的 DBA 会坚持使用整形值代码。那也一样轻而易举： 在配置文件中把 EnumOrdinalTypeHandler 加到 typeHandlers 中即可， 这样每个 RoundingMode 将通过他们的序数值来映射成对应的整形。

<!-- mybatis-config.xml -->

<**typeHandlers**>

<**typeHandler** handler="org.apache.ibatis.type.EnumOrdinalTypeHandler" javaType="java.math.RoundingMode"/>

</**typeHandlers**>

但是怎样能将同样的 Enum 既映射成字符串又映射成整形呢？

自动映射器（auto-mapper）会自动地选用 EnumOrdinalTypeHandler 来处理， 所以如果我们想用普通的 EnumTypeHandler，就非要为那些 SQL 语句显式地设置要用到的类型处理器不可。

（下一节才开始讲映射器文件，所以如果是首次阅读该文档，你可能需要先越过这一步，过会再来看。）

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<**mapper** namespace="org.apache.ibatis.submitted.rounding.Mapper">

<**resultMap** type="org.apache.ibatis.submitted.rounding.User" id="usermap">

<**id** column="id" property="id"/>

<**result** column="name" property="name"/>

<**result** column="funkyNumber" property="funkyNumber"/>

<**result** column="roundingMode" property="roundingMode"/>

</**resultMap**>

<**select** id="getUser" resultMap="usermap">

select \* from users

</**select**>

<**insert** id="insert">

insert into users (id, name, funkyNumber, roundingMode) values (

#{id}, #{name}, #{funkyNumber}, #{roundingMode}

)

</**insert**>

<**resultMap** type="org.apache.ibatis.submitted.rounding.User" id="usermap2">

<**id** column="id" property="id"/>

<**result** column="name" property="name"/>

<**result** column="funkyNumber" property="funkyNumber"/>

<**result** column="roundingMode" property="roundingMode" typeHandler="org.apache.ibatis.type.EnumTypeHandler"/>

</**resultMap**>

<**select** id="getUser2" resultMap="usermap2">

select \* from users2

</**select**>

<**insert** id="insert2">

insert into users2 (id, name, funkyNumber, roundingMode) values (

#{id}, #{name}, #{funkyNumber}, #{roundingMode, typeHandler=org.apache.ibatis.type.EnumTypeHandler}

)

</**insert**>

</**mapper**>

注意，这里的 select 语句强制使用 resultMap 来代替 resultType。

**对象工厂（objectFactory）**

MyBatis 每次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。 默认的对象工厂需要做的仅仅是实例化目标类，要么通过默认构造方法，要么在参数映射存在的时候通过参数构造方法来实例化。 如果想覆盖对象工厂的默认行为，则可以通过创建自己的对象工厂来实现。比如：

// ExampleObjectFactory.java

**public** **class** **ExampleObjectFactory** **extends** **DefaultObjectFactory** {

**public** Object **create**(Class type) {

**return** **super**.create(type);

}

**public** Object **create**(Class type, List constructorArgTypes, List constructorArgs) {

**return** **super**.create(type, constructorArgTypes, constructorArgs);

}

**public** **void** **setProperties**(Properties properties) {

**super**.setProperties(properties);

}

**public** **boolean** **isCollection**(Class type) {

**return** Collection.class.isAssignableFrom(type);

}}

<!-- mybatis-config.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的构造方法的。 最后，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（plugins）**

MyBatis 允许你在已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)

这些类中方法的细节可以通过查看每个方法的签名来发现，或者直接查看 MyBatis 的发行包中的源代码。 假设你想做的不仅仅是监控方法的调用，那么你应该很好的了解正在重写的方法的行为。 因为如果在试图修改或重写已有方法的行为的时候，你很可能在破坏 MyBatis 的核心模块。 这些都是更低层的类和方法，所以使用插件的时候要特别当心。

通过 MyBatis 提供的强大机制，使用插件是非常简单的，只需实现 Interceptor 接口，并指定了想要拦截的方法签名即可。

// 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) {

}

}

<!-- mybatis-config.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， 这里的 Executor 是负责执行低层映射语句的内部对象。

NOTE **覆盖配置类**

除了用插件来修改 MyBatis 核心行为之外，还可以通过完全覆盖配置类来达到目的。只需继承后覆盖其中的每个方法，再把它传递到 sqlSessionFactoryBuilder.build(myConfig) 方法即可。再次重申，这可能会严重影响 MyBatis 的行为，务请慎之又慎。

**配置环境（environments）**

MyBatis 可以配置成适应多种环境，这种机制有助于将 SQL 映射应用于多种数据库之中， 现实情况下有多种理由需要这么做。例如，开发、测试和生产环境需要有不同的配置；或者共享相同 Schema 的多个生产数据库， 想使用相同的 SQL 映射。许多类似的用例。

**不过要记住：尽管可以配置多个环境，每个 SqlSessionFactory 实例只能选择其一。**

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

* **每个数据库对应一个 SqlSessionFactory 实例**

为了指定创建哪种环境，只要将它作为可选的参数传递给 SqlSessionFactoryBuilder 即可。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);

如果忽略了环境参数，那么默认环境将会被加载，如下所示：

**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**>

注意这里的关键点:

* 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 是一目了然的。随你怎么命名，只要保证默认环境要匹配其中一个环境ID。

**事务管理器（transactionManager）**

在 MyBatis 中有两种类型的事务管理器（也就是 type="[JDBC|MANAGED]"）：

* JDBC – 这个配置就是直接使用了 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，而是让容器来管理事务的整个生命周期（比如 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，因此需要将 closeConnection 属性设置为 false 来阻止它默认的关闭行为。例如:

<transactionManager type="MANAGED">

<**property** name="closeConnection" value="false"/>

</**transactionManager**>

NOTE如果你正在使用 Spring + MyBatis，则没有必要配置事务管理器， 因为 Spring 模块会使用自带的管理器来覆盖前面的配置。

这两种事务管理器类型都不需要任何属性。它们不过是类型别名，换句话说，你可以使用 TransactionFactory 接口的实现类的完全限定名或类型别名代替它们。

**public** **interface** **TransactionFactory** {

**void** **setProperties**(Properties props);

Transaction **newTransaction**(Connection conn);

Transaction **newTransaction**(DataSource dataSource, TransactionIsolationLevel level, **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。你也需要创建一个 Transaction 接口的实现类，这个接口也很简单：

**public** **interface** **Transaction** {

Connection **getConnection**() **throws** SQLException;

**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。

**数据源（dataSource）**

dataSource 元素使用标准的 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|POOLED|JNDI]"）：

**UNPOOLED**– 这个数据源的实现只是每次被请求时打开和关闭连接。虽然一点慢，它对在及时可用连接方面没有性能要求的简单应用程序是一个很好的选择。 不同的数据库在这方面表现也是不一样的，所以对某些数据库来说使用连接池并不重要，这个配置也是理想的。UNPOOLED 类型的数据源仅仅需要配置以下 5 种属性：

* driver – 这是 JDBC 驱动的 Java 类的完全限定名（并不是JDBC驱动中可能包含的数据源类）。
* 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，你也可以传递属性给数据库驱动。要这样做，属性的前缀为"driver."，例如：

* driver.encoding=UTF8

这将通过DriverManager.getConnection(url,driverProperties)方法传递值为 UTF8 的 encoding 属性给数据库驱动。

**POOLED**– 这种数据源的实现利用"池"的概念将 JDBC 连接对象组织起来，避免了创建新的连接实例时所必需的初始化和认证时间。 这是一种使得并发 Web 应用快速响应请求的流行处理方式。

除了上述提到 UNPOOLED 下的属性外，会有更多属性用来配置 POOLED 的数据源：

* poolMaximumActiveConnections – 在任意时间可以存在的活动（也就是正在使用）连接数量，默认值：10
* poolMaximumIdleConnections – 任意时间可能存在的空闲连接数。
* poolMaximumCheckoutTime – 在被强制返回之前，池中连接被检出（checked out）时间，默认值：20000 毫秒（即 20 秒）
* poolTimeToWait – 这是一个底层设置，如果获取连接花费的相当长的时间，它会给连接池打印状态日志并重新尝试获取一个连接（避免在误配置的情况下一直安静的失败），默认值：20000 毫秒（即 20 秒）。
* poolPingQuery – 发送到数据库的侦测查询，用来检验连接是否处在正常工作秩序中并准备接受请求。默认是"NO PING QUERY SET"，这会导致多数数据库驱动失败时带有一个恰当的错误消息。
* poolPingEnabled – 是否启用侦测查询。若开启，也必须使用一个可执行的 SQL 语句设置 poolPingQuery 属性（最好是一个非常快的 SQL），默认值：false。
* poolPingConnectionsNotUsedFor – 配置 poolPingQuery 的使用频度。这可以被设置成匹配具体的数据库连接超时时间，来避免不必要的侦测，默认值：0（即所有连接每一时刻都被侦测 — 当然仅当 poolPingEnabled 为 true 时适用）。

**JNDI**– 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，容器可以集中或在外部配置数据源，然后放置一个 JNDI 上下文的引用。这种数据源配置只需要两个属性：

* initial\_context – 这个属性用来在 InitialContext 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，可以通过添加前缀"env."直接把属性传递给初始上下文。比如：

* env.encoding=UTF8

这就会在初始上下文（InitialContext）实例化时往它的构造方法传递值为 UTF8 的 encoding 属性。

通过需要实现接口 org.apache.ibatis.datasource.DataSourceFactory ， 也可使用任何第三方数据源，：

**public** **interface** **DataSourceFactory** {

**void** **setProperties**(Properties props);

DataSource **getDataSource**();

}

org.apache.ibatis.datasource.unpooled.UnpooledDataSourceFactory 可被用作父类来构建新的数据源适配器，比如下面这段插入 C3P0 数据源所必需的代码：

**import** org.apache.ibatis.datasource.unpooled.UnpooledDataSourceFactory;

**import** com.mchange.v2.c3p0.ComboPooledDataSource;

**public** **class** **C3P0DataSourceFactory** **extends** **UnpooledDataSourceFactory** {

**public** **C3P0DataSourceFactory**() {

**this**.dataSource = **new** ComboPooledDataSource();

}

}

为了令其工作，为每个需要 MyBatis 调用的 setter 方法中增加一个属性。下面是一个可以连接至 PostgreSQL 数据库的例子：

<**dataSource** type="org.myproject.C3P0DataSourceFactory">

<**property** name="driver" value="org.postgresql.Driver"/>

<**property** name="url" value="jdbc:postgresql:mydb"/>

<**property** name="username" value="postgres"/>

<**property** name="password" value="root"/>

</**dataSource**>

**databaseIdProvider**

MyBatis 可以根据不同的数据库厂商执行不同的语句，这种多厂商的支持是基于映射语句中的 databaseId 属性。 MyBatis 会加载不带 databaseId 属性和带有匹配当前数据库 databaseId 属性的所有语句。 如果同时找到带有 databaseId 和不带 databaseId 的相同语句，则后者会被舍弃。 为支持多厂商特性只要像下面这样在 mybatis-config.xml 文件中加入 databaseIdProvider 即可：

<**databaseIdProvider** type="DB\_VENDOR" />

这里的 DB\_VENDOR 会通过 DatabaseMetaData#getDatabaseProductName() 返回的字符串进行设置。 由于通常情况下这个字符串都非常长而且相同产品的不同版本会返回不同的值，所以最好通过设置属性别名来使其变短，如下：

<**databaseIdProvider** type="DB\_VENDOR">

<**property** name="SQL Server" value="sqlserver"/>

<**property** name="DB2" value="db2"/>

<**property** name="Oracle" value="oracle" />

</**databaseIdProvider**>

在有 properties 时，DB\_VENDOR databaseIdProvider 的将被设置为第一个能匹配数据库产品名称的属性键对应的值，如果没有匹配的属性将会设置为 "null"。 在这个例子中，如果 getDatabaseProductName() 返回"Oracle (DataDirect)"，databaseId 将被设置为"oracle"。

你可以通过实现接口 org.apache.ibatis.mapping.DatabaseIdProvider 并在 mybatis-config.xml 中注册来构建自己的 DatabaseIdProvider：

**public** **interface** **DatabaseIdProvider** {

**void** **setProperties**(Properties p);

String **getDatabaseId**(DataSource dataSource) **throws** SQLException;

}

**映射器（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 在自动查找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， 或完全限定资源定位符（包括 file:/// 的 URL），或类名和包名等。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/mappers/AuthorMapper.xml"/>

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

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

</**mappers**>

<!-- Using mapper interface classes -->

<**mappers**>

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

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

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

</**mappers**>

<!-- Register all interfaces in a package as mappers -->

<**mappers**>

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

</**mappers**>

这些配置会告诉了 MyBatis 去哪里找映射文件，剩下的细节就应该是每个 SQL 映射文件了，也就是接下来我们要讨论的。