目录

MyBatis 分享

简介

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。

功能

我们把Mybatis的功能架构分为三层：

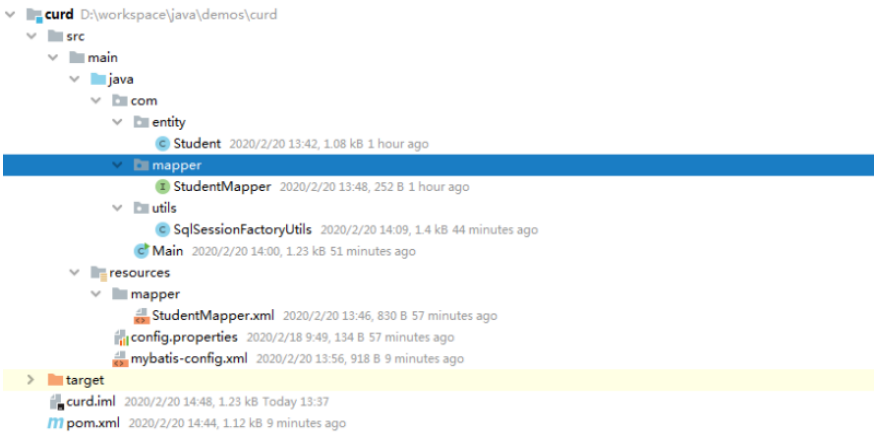
API接口层：提供给外部使用的接口API，开发人员通过这些本地API来操纵数据库。接口层一接收到调用请求就会调用数据处理层来完成具体的数据处理。

数据处理层：负责具体的SQL查找、SQL解析、SQL执行和执行结果映射处理等。它主要的目的是根据调用的请求完成一次数据库操作。

基础支撑层：负责最基础的功能支撑，包括连接管理、事务管理、配置加载和缓存处理，这些都是共用的东西，将他们抽取出来作为最基础的组件。为上层的数据处理层提供最基础的支撑。

项目结构

项目是纯MyBatis项目， 不带Spring的代码。



入门

安装

使用maven构建项目，使用下面的依赖配置即可

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

构建

每个基于MyBatis的应用都有一个SqlSessionFactory为核心的实例，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中构建SqlSessionFactory

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 SqlSessionFactoryBuild
```

从 SqlSessionFactory 中获取 SqlSession

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

SQL语句映射

下面是MyBatis SQL的XML定义

```
<?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>
```

使用命名空间调用

```
Blog blog = (Blog) session.selectOne("org.mybatis.example.BlogMa
```

使用映射器类

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

直接使用映射器类

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

生命周期

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。

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

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）
- settings（设置）
- typeAliases（类型别名）
- typeHandlers（类型处理器）
- objectFactory（对象工厂）
- plugins（插件）
- environments（环境配置）
  - environment（环境变量）
    - transactionManager（事务管理器）
    - dataSource（数据源）
- databaseIdProvider（数据库厂商标识)
- mappers（映射器）
```

而且这个配置文件必须按照这个顺序，如果顺序不对就报错

properties （属性）

这些属性可以在外部进行配置，并可以进行动态替换。你既可以在典型的 Java 属性文件中配置这些属性，也可以在 properties 元素的子元素中设置。例如：

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

设置好的属性可以在整个配置文件中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>
```

SqlSessionFactoryBuilder.build() 方法中也可以传入property的属性只

```
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build
// ... 或者 ...
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build
```

property加载顺序

1. 首先读取在 properties 元素体内指定的属性。
2. 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。
3. 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。

优先级从大到小是: 方法 > 文件 > properties

property默认值

```
<properties resource="org/mybatis/example/config.properties">
  <!-- 启用默认值特性 -->
  <property name="org.apache.ibatis.parsing.PropertyParser.enabl
  <!-- 如果属性 'username' 没有被配置，'username' 属性的值将为 'ut_
  <property name="username" value="${username:ut_user}"/>
</properties>
```

property特殊符号

```
<properties resource="org/mybatis/example/config.properties">
 <!-- 修改默认值的分隔符 -->
 <property name="org.apache.ibatis.parsing.PropertyParser.defaul
 <!-- 如果属性 'username' 没有被配置，'username' 属性的值将为 'ut_u
 <property name="username" value="${db:username?:ut_user}"/>
</properties>
```

settings （设置）

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设置的含义、默认值等。下面给出了一个完整的配置，更多详细配置请参考MyBatis

```
<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="WARNIN
  <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,has
</settings>
```

typeAliases （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。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。见下面的例子：

```
@Alias("author")
public class Author {
    ...
}
```

Java 类型内建的类型别名

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 （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。

几乎所有的类型都有默认的类型处理器BooleanTypeHandler，ByteTypeHandler，ShortTypeHandler等。

objectFactory （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。比如

```
public class StudentFactory extends DefaultObjectFactory {
    //处理默认构造方法
    public <T> T create(Class<T> type) {
        return super.create(type);
    }

    //处理有参构造方法，这里可以添加一些处理逻辑
    public <T> T create(Class<T> type, List<Class<?>> constructo
        T ret = super.create(type, constructorArgTypes, construc
        if (Student.class.isAssignableFrom(type)) {
            Student stu = (Student) ret;
            stu.ini();
        }
        return ret;
    }

    //处理参数
    public void setProperties(Properties properties) {
        System.out.printf("StudentFactory 处理参数:  %s\n", prope
        super.setProperties(properties);
    }

    //判断集合类型参数
    public <T> boolean isCollection(Class<T> type) {
        return Collection.class.isAssignableFrom(type);
    }
}
```

配置文件 mybatis-config.xml

```xml
<!-- mybatis-config.xml -->
<objectFactory type="com.factory.StudentFactory">
    <!-- mybatis初始化后生效，全局变量 -->
    <property name="fname" value="student factory"/>
</objectFactory>
```

plugins （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，只需实现 Interceptor 接口，并指定想要拦截的方法签名即可实现拦截，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)

```java
@Intercepts({
        @Signature(
                type = Executor.class,
                method = "query",
                args = {MappedStatement.class, Object.class, Row
        )
})
public class ExamplePlugin implements Interceptor {


    //拦截逻辑，参数是代理类
    public Object intercept(Invocation invocation) throws Throwa
        System.out.printf("ExamplePlugin 处理中 ....\n");
        return invocation.proceed();
    }

    // 加载插件，一般使用Plugin.wrap(target, this);加载当前插件
    public Object plugin(Object target) {
        return Plugin.wrap(target, this);
    }

    // 初始化属性
    public void setProperties(Properties properties) {
        properties.setProperty("newValue", "200");
    }
}
```

配置

```
<!-- mybatis-config.xml -->
<plugins>
    <plugin interceptor="com.plugin.ExamplePlugin">
        <property name="newValue" 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 是负责执行底层映射语句的内部对象。

environments （环境配置）

显示开发有多个环境，MyBatis也可以配置多个不同的环境，但是每个SqlSessionFactory 只能有一个环境

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

            <property name="driver" value="${database.driver}"/>
            <property name="url" value="${database.url}"/>
            <property name="username" value="${database.username
            <property name="password" value="${database.password
        </dataSource>
    </environment>


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

选择某一个环境变量，这里的product，可以使用系统环境变量，灵活选择对应的环境

```
sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputSt

System.out.printf("env: %s\n", sqlSessionFactory.getConfiguratio
```

transactionManager（事务管理器）

MyBatis有两种事务管理器，如果是使用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。

- JDBC – 这个配置直接使用了 JDBC 的提交和回滚设施，它依赖从数据源获得的连接来管理事务作用域，这也是MyBatis的默认值
- MANAGED – 这个配置几乎没做什么。它从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连接被关闭，因此需要将 closeConnection 属性设置为 false 来阻止默认的关闭行为。例如:

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

dataSource （数据源）

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。有三种内建的数据源类型（也就是 type="[UNPOOLED|POOLED|JNDI]"）：

UNPOOLED

这个数据源的实现会每次请求时打开和关闭连接

- `driver` – 这是 JDBC 驱动的 Java 类全限定名（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。
- `defaultNetworkTimeout` – 等待数据库操作完成的默认网络超时时间（单位：毫秒）
- driver.encoding=UTF8（可选）

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。除了上面的属性，还有一些针对连接池的属性

- `poolMaximumActiveConnections` – 在任意时间可存在的活动（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，如果获取连接花费了相当长的时间，连接池会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失败且不打印日志），默认值：20000 毫秒（即 20 秒）。
- `poolMaximumLocalBadConnectionTolerance` – 这是一个关于坏连接容忍度的底层设置， 作用于每一个尝试从缓存池获取连接的线程。 如果这个线程获取到的是一个坏的连接，那么这个数据源允许这个线程尝试重新获取一个新的连接，但是这个重新尝试的次数不应该超过 `poolMaximumIdleConnections` 与 `poolMaximumLocalBadConnectionTolerance` 之和。 默认值：3（新增于 3.4.5）
- `poolPingQuery` – 发送到数据库的侦测查询，用来检验连接是否正常工作并准备接受请求。默认是“NO PING QUERY SET”，这会导致多数数据库驱动出错时返回恰当的错误消息。
- `poolPingEnabled` – 是否启用侦测查询。若开启，需要设置 `poolPingQuery` 属性为一个可执行的 SQL 语句（最好是一个速度非常快的 SQL 语句），默认值：false。
- `poolPingConnectionsNotUsedFor` – 配置 poolPingQuery 的频率。可以被设置为和数据库连接超时时间一样，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

个数据源实现是为了能在如 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 上下文的数据源引用。

- `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，那么将会直接从 InitialContext 中寻找 data_source 属性。
- `data_source` – 这是引用数据源实例位置的上下文路径。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。
- env.encoding=UTF8 （可选）

databaseIdProvider （数据库厂商标识)

下面的配置，会在后面的动态SQL中得到应用

```
<!--   位于environments之后 -->
<databaseIdProvider type="DB_VENDOR">
    <property name="MySQL" value="mysql"/>
    <property name="Oracle" value="oracle" />
</databaseIdProvider>
```

mappers （映射器）

既然 MyBatis 的行为已经由上述元素配置完了，我们现在就要来定义 SQL 映射语句了。 但首先，我们需要告诉 MyBatis 到哪里去找到这些语句。 在自动查找资源方面，Java 并没有提供一个很好的解决方案，所以最好的办法是直接告诉 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:///` 形式的 URL），或类名和包名等。例如：

```
<!-- 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映射文件

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映射文件主要是下面的几个元素

1. select
2. insert
3. delete
4. update
5. sql
6. resultMap
7. cache
8. cache-ref

select ,insert, update,delete

```
<!--    根据ID查询-->
<select id="get" parameterType="long" resultType="com.entity.Stu
    select id,name,age from student where id=#{id}
</select>
<!--    更新-->
<update id="update">
    update student set name=#{name} where id=#{id}
</update>
<!--    新增-->
<insert id="add" useGeneratedKeys="true" keyProperty="id">
    insert into student ( name ,age) values (#{name}, #{age});
</insert>
<!--    删除-->
<delete id="delete" parameterType="int">
    delete from student where  id=#{id};
</delete>
```

主键自动生成

insert和update元素中，MyBatis 会使用 getGeneratedKeys 的返回值或 insert 语句的 selectKey 子元素设置它的值，默认值：未设置（ `unset` ）。如果生成列不止一个，可以用逗号分隔多个属性名称。

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

使用selectKey, 这里使用了一个random函数生成一个随机的ID(不建议这么使用）

```
<insert id="insertAuthor">
  <selectKey keyProperty="id" resultType="int" order="BEFORE">
     <!--   SELECT LAST_INSERT_ID()-->
    select CAST(RANDOM()*1000000 as INTEGER) a from SYSIBM.SYSDU
  </selectKey>
  insert into Author
    (id, username, password, email,bio, favourite_section)
  values
    (#{id}, #{username}, #{password}, #{email}, #{bio}, #{favour
</insert>
```

select 元素常用属性

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。已被废弃
`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。

属性	描述
`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,delete元素常用属性

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。
`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：（对 insert、update 和 delete 语句）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）。
`statementType`	可选 STATEMENT，PREPARED 或 CALLABLE。这会让 MyBatis 分别使用 Statement，PreparedStatement 或 CallableStatement，默认值：PREPARED。
`useGeneratedKeys`	（仅适用于 insert 和 update）这会令 MyBatis 使用 JDBC 的 getGeneratedKeys 方法来取出由数据库内部生成的主键（比如：像 MySQL 和 SQL Server 这样的关系型数据库管理系统的自动递增字段），默认值：false。
`keyProperty`	（仅适用于 insert 和 update）指定能够唯一识别对象的属性，MyBatis 会使用 getGeneratedKeys 的返回值或 insert 语句的 selectKey 子元素设置它的值，默认值：未设置（ `unset` ）。如果生成列不止一个，可以用逗号分隔多个属性名称。
`keyColumn`	（仅适用于 insert 和 update）设置生成键值在表中的列名，在某些数据库（像 PostgreSQL）中，当主键列不是表中的第一列的时候，是必须设置的。如果生成列不止一个，可以用逗号分隔多个属性名称。
`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。

SQL

SQL是定义的一个代码片段，可以在其他语句使用

```
<sql id="userColumns"> ${alias}.id,${alias}.username,${alias}.pa

<select id="selectUsers" resultType="map">
  select
    <include refid="userColumns"><property name="alias" value="t
    <include refid="userColumns"><property name="alias" value="t
  from some_table t1
    cross join some_table t2
</select>
```

参数

Select元素的参数和Insert元素的参数是不同的，Select不需要多复杂的元素，Insert需要一个hashMap参数，还有可能存hashmap和基本元素并存的情况

指定类型简单参数

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

hashmap

下面的insert语句中，参数是一个User对象，这个对象有三个属性，id，username，password

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

混合

有时候参数可能有几个，但不是一个对象，这个时候无法使用username指定了，但是可以用 #{arg0}， #{arg1}顺序指定，但是有更好的指定方式

```
<select id="get" resultMap="userResultMapper">
    select id,user_id,name,age from user_#{tableIndex} where use
</select>
```

Java代码中这样写

```
public User get(@Param("userId") long userId, @Param("tableIndex
```

字符串替换

如果遇到更灵活的，重复很高的查询语句，比如：

```
@Select("select * from user where id = #{id}")
User findById(@Param("id") long id);

@Select("select * from user where name = #{name}")
User findByName(@Param("name") String name);

@Select("select * from user where email = #{email}")
User findByEmail(@Param("email") String email);

// 其它的 "findByXxx" 方法
```

可以使用下面的一个方法

```
@Select("select * from user where ${column} = #{value}")
User findByColumn(@Param("column") String column, @Param("value"
```

Java代码可以这样写

```
User userOfId1 = userMapper.findByColumn("id", 1L);
User userOfNameKid = userMapper.findByColumn("name", "kid");
User userOfEmail = userMapper.findByColumn("email", "noone@nowhe
```

结果映射

resultMap元素是MyBatis中最强大，也是比较复杂的元素

```
<select id="get" parameterType="long" resultType="com.entity.Stu
      select id,name,age from student where id=#{id}
</select>
```

上面的查询语句resultType指定了一个Student接收查询结果，MyBatis会在幕后自动创建一个 `ResultMap` ，再根据属性名来映射列到JavaBean 的属性上

显示resultMap

```
<resultMap id="userResultMapper" type="com.entity.Student">
    <id property="id" column="id"></id>
    <result property="age" column="age"></result>
    <result property="name" column="name"></result>
</resultMap>

<select id="queryByPage" resultMap="userResultMapper">
    select id,name,age from student limit #{arg0}, #{arg1}
</select>
```

autoMapping

上面的映射有三个等级，默认值是 PARTIAL

- NONE

  禁用自动映射。仅对手动映射的属性进行映射。

- PARTIAL

  对除在内部定义了嵌套结果映射（也就是连接的属性）以外的属性进行映射

- FULL

  自动映射所有属性

缓存

MyBatis 内置了一个强大的事务性查询缓存机制，它可以非常方便地配置和定制。默认情况下，只启用了本地的会话缓存，它仅仅对一个会话中的数据进行缓存。 要启用全局的二级缓存，只需要在你的 SQL 映射文件中添加一行：

```
<cache/>
```

基本上就是这样。这个简单语句的效果如下:

- 映射语句文件中的所有 select 语句的结果将会被缓存。
- 映射语句文件中的所有 insert、update 和 delete 语句会刷新缓存。
- 缓存会使用最近最少使用算法（LRU, Least Recently Used）算法来清除不需要的缓存。
- 缓存不会定时进行刷新（也就是说，没有刷新间隔）。
- 缓存会保存列表或对象（无论查询方法返回哪种）的 1024 个引用。
- 缓存会被视为读/写缓存，这意味着获取到的对象并不是共享的，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。

动态 SQL

动态 SQL 是 MyBatis 的强大特性之一，如果你在代码中使用过原生的SQL，那么肯定感受过处理SQL拼接中空格，逗号等痛苦。动态SQL可以帮你不需要关注这些细枝末节

由于MyBatis3精简了元素的种类，现在需要学习的很少，主要是有下面的几类

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

if

```
<select id="query" resultType="com.entity.Student">
    select id,name,age from student
    <where>
        <if test="name!=null">
            AND name like '%#{name}%'
        </if>
        <if test="age>0">
            AND age=#{age}
        </if>
    </where>
</select>
```

choose (when, otherwise)

```
<select id="pick" resultType="com.entity.Student">
    select id,name,age from student
    <choose>
        <when test="name!=null">
            AND name like '%#{name}%'
        </when>
        <when test="age>0">
            AND age=#{age}
        </when>
        <otherwise>
            AND age>20
        </otherwise>
    </choose>
</select>
```

trim (where, set)

Where

```
<select id="findActiveBlogLike"
     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 author_name like #{author.name}
    </if>
  </where>
</select>
```

trim

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

set

```
<update id="update" parameterType="com.entity.Student">
    update student
    <set>
        <if test="name!=null">name=#{name},</if>
        <if test="age>0">age=#{age}</if>
    </set>
    where id=#{id}
</update>
```

foreach

foreach用来方便的描述，select * from student where id in (2,3,4);

```
<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>
```

script

要在带注解的映射器接口类中使用动态 SQL，可以使用 script 元素。比如:

```
@Update({"<script>",
      "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}",
      "</script>"})
    void updateAuthorValues(Author author);
```

但是这里使用的很不方便，建议还是在XML文件中使用比较好

bind

`bind` 元素允许你在 OGNL（对象图导航语言） 表达式以外创建一个变量，并将其绑定到当前的上下文。比如：

```
<select id="selectBlogsLike" resultType="Blog">
  <bind name="pattern" value="'%' + _parameter.getTitle() + '%'"
  SELECT * FROM BLOG
  WHERE title LIKE #{pattern}
</select>
```

多数据库支持

如果配置了 databaseIdProvider，你就可以在动态代码中使用名为 “_databaseId” 的变量来为不同的数据库构建特定的语句。比如下面的例子：

```
<!--   位于environments之后 -->
<databaseIdProvider type="DB_VENDOR">
    <property name="MySQL" value="mysql"/>
    <property name="Oracle" value="oracle" />
</databaseIdProvider>
```

```
<insert id="insert">
  <selectKey keyProperty="id" resultType="int" order="BEFORE">
    <if test="_databaseId == 'oracle'">
      select seq_users.nextval from dual
    </if>
    <if test="_databaseId == 'db2'">
      select nextval for seq_users from sysibm.sysdummy1"
    </if>
  </selectKey>
  insert into users values (#{id}, #{name})
</insert>
```

多数据库使用起来还是比较的混乱的，不建议使用

动态 SQL 中的插入脚本语言

MyBatis 从 3.2 版本开始支持插入脚本语言，这允许你插入一种语言驱动，并基于这种语言来编写动态 SQL 查询语句。详情可以查看MyBatis

API

MyBatis的API的使用就是MyBatis使用的一些类，在JAVA项目代码中，程序员可以使用这些API实现自定义功能

SqlSession

SqlSession由SqlSessionFactory创建， SqlSessionFactory由SqlSessionFactoryBuilde创建。创建SqlSessionFactory有五种方法

```
SqlSessionFactory build(InputStream inputStream)
SqlSessionFactory build(InputStream inputStream, String environm
SqlSessionFactory build(InputStream inputStream, Properties prop
SqlSessionFactory build(InputStream inputStream, String env, Pro
SqlSessionFactory build(Configuration config)
```

SqlSessionFactory

获取一个session的方法有更多，获取一个session通常要考虑，事务处理，数据库连接，语句执行等

```
SqlSession openSession()
SqlSession openSession(boolean autoCommit)
SqlSession openSession(Connection connection)
SqlSession openSession(TransactionIsolationLevel level)
SqlSession openSession(ExecutorType execType, TransactionIsolati
SqlSession openSession(ExecutorType execType)
SqlSession openSession(ExecutorType execType, boolean autoCommit
SqlSession openSession(ExecutorType execType, Connection connect
Configuration getConfiguration();
```

无参数的session的默认值如下

- 事务作用域将会开启（也就是不自动提交）。
- 将由当前环境配置的 DataSource 实例中获取 Connection 对象。
- 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。

ExecutorType的类型，描述的是session的预处理语句的生成方式

- `ExecutorType.SIMPLE`：该类型的执行器没有特别的行为。它为每个语句的执行创建一个新的预处理语句。
- `ExecutorType.REUSE`：该类型的执行器会复用预处理语句。
- `ExecutorType.BATCH`：该类型的执行器会批量执行所有更新语句，如果 SELECT 在多个更新中间执行，将在必要时将多条更新语句分隔开来，以方便理解。

生成Java代码 不生成XML-MyBatis3注释仅用于 生成的模型对象是“平面的”-没有单独的主键对象 生成的代码依赖于MyBatis动态SQL库 生成的代码量相对较小 生成的代码在查询构造方面具有极大的灵活性

SQL语句构建器

SQL语句构建器这个功能针对复杂的语句，我们目前的开发不建议使用复杂的语句，而且其使用也比较复杂

```
String sql = "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";
```

使用SQL语句构建器可以这样写

```
private String selectPersonSql() {
  return new SQL() {{
    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");
  }}.toString();
}
```

日志

```
<configuration>
  <settings>
    ...
    <setting name="logImpl" value="LOG4J"/>
    ...
  </settings>
</configuration>
```

日志包只知道使用，没有深入探究过

MyBatis Generator 分享

MyBatis Generator

MyBatisGenerator 是自动生成MyBatis代码的的工具

使用

此工具是代码生成工具，和业务代码没关系，所以在maven的配置是在build元素下面

```
<build>
    <plugins>
        <!-- mybatis代码生成插件 -->
        <plugin>
            <groupId>org.mybatis.generator</groupId>
            <artifactId>mybatis-generator-maven-plugin</artifact
            <version>1.3.2</version>
            <configuration>
                <!--配置文件的位置-->
                <configurationFile>src/main/resources/generatorC
                <verbose>true</verbose>
                <overwrite>true</overwrite>
            </configuration>
            <executions>
                <execution>
                    <id>Generate MyBatis Artifacts</id>
                    <goals>
                        <goal>generate</goal>
                    </goals>
                </execution>
            </executions>
            <dependencies>
                <dependency>
                    <groupId>org.mybatis.generator</groupId>
                    <artifactId>mybatis-generator-core</artifact
                    <version>1.3.2</version>
                </dependency>
            </dependencies>
        </plugin>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

下面是generatorConfig.xml配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
        PUBLIC "-//mybatis.org//DTD MyBatis Generator Configurat
        "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd

<generatorConfiguration>
    <!--mysql 连接数据库jar 这里选择自己本地位置;
    如果不知道maven本地仓库地址，可以使用EveryThing工具全局搜索mysql
    也可以手动下载一个jar放在指定位置，进行引用。
    -->
    <classPathEntry
            location="D:/apache/apache-maven-3.6.1/repos/mysql/m

    <context id="default" targetRuntime="MyBatis3">
        <commentGenerator>
            <!-- 是否去除自动生成的注释,true：是,false:否 -->
            <property name="suppressAllComments" value="true"/>
        </commentGenerator>

        <!--数据库连接的信息：驱动类、连接地址、用户名、密码 -->
        <jdbcConnection driverClass="com.mysql.jdbc.Driver"
                        connectionURL="jdbc:mysql://localhost:33
                        password="123456">
        </jdbcConnection>

        <!-- 默认false，把JDBC DECIMAL 和 NUMERIC 类型解析为 Integ
           NUMERIC 类型解析为java.math.BigDecimal -->
        <javaTypeResolver>
            <property name="forceBigDecimals" value="false"/>
        </javaTypeResolver>

        <!-- 指定javaBean生成的位置
            targetPackage：生成的类要放的包，真实的包受enableSubPac
            targetProject：目标项目，指定一个存在的目录下，生成的内
         -->
        <javaModelGenerator targetPackage="com.gaodun.generator.
            <!-- 在targetPackage的基础上，根据数据库的schema再生成-
            <!--            <property name="enableSubPackages" v
            <!-- 设置是否在getter方法中，对String类型字段调用trim()
            <property name="trimStrings" value="true"/>
        </javaModelGenerator>

        <!--  指定mapper映射文件生成的位置
           targetPackage、targetProject同javaModelGenerator中作用
        <sqlMapGenerator targetPackage="mapper" targetProject="s
            <property name="enableSubPackages" value="false"/>
        </sqlMapGenerator>
```

```
        <!-- 指定mapper接口生成的位置
         targetPackage、targetProject同javaModelGenerator中作用一
         -->
        <javaClientGenerator type="XMLMAPPER" targetPackage="com
                             targetProject="src/main/java">
            <property name="enableSubPackages" value="false"/>
        </javaClientGenerator>

        <!-- 指定数据库表
        domainObjectName：生成的domain类的名字,当表名和domain类的名
        可以设置为somepck.domainName，那么会自动把domainName类再放
        -->
        <!--        <table tableName="student" domainObjectName=
        <!--               enableSelectByExample="false" selectB
        <table tableName="student" domainObjectName="Student">
            <generatedKey column="id" sqlStatement="Mysql"/>
        </table>
    </context>
</generatorConfiguration>
```

常见问题

下面介绍MyBatis使用的过程中遇到的一些问题

再次生成覆盖

```
<configuration>
       <!--配置文件的位置-->
       <configurationFile>src/main/resources/generatorConfig.xm
       <verbose>true</verbose>
       <overwrite>true</overwrite>
</configuration>
```

如果是完全覆盖，新增字段需要注意原有的逻辑，选择merge，要手动决绝冲突

只生成了insert和List方法

可能是主键问题, 网上有详细的解决办法

生成了很多方法

使用下面的配置去掉不想要的方法

```
enableUpdateByExample="false"
enableDeleteByExample="false"
enableSelectByExample="false"
```

MyBatis的targetRuntime

MyBatis3DynamicSql

- Generates Java code
- Does not generate XML - MyBatis3 annotations are used exclusively
- The generated model objects are "flat" - there is no separate primary key object
- The generated code is dependent on the MyBatis Dynamic SQL Library
- The amount of generated code is relatively small
- The generated code allows tremendous flexibility in query construction

MyBatis3Simple

- Generates Java code
- Generates MyBatis3 compatible XML and SQL or MyBatis3 compatible annotated interfaces with no XML
- The generated model objects are "flat" - there is no separate primary key object
- The generated code has no external dependencies
- The amount of generated code is relatively small
- No "by example" or "selective" methods are generated
- The generated code does not include methods for dynamic query construction and is difficult to extend

百度翻译一下，这个也是我们现在使用的

- 生成Java代码
- 生成与MyBatis3兼容的XML和SQL或与MyBatis3兼容的无XML注释接口
- 生成的模型对象是“平面的”-没有单独的主键对象
- 生成的代码没有外部依赖项
- 生成的代码量相对较小
- 不生成“按示例”或“选择性”方法

- 生成的代码不包含动态查询构造的方法，并且很难扩展

MyBatis3

- Generates Java code
- Generates MyBatis3 compatible XML and SQL or MyBatis3 compatible annotated interfaces with no XML
- The generated model objects may have a hierarchy with separate primary key objects and/or separate object with BLOB fields
- The generated code has no external dependencies
- The amount of generated code is very large
- The generated code has limited capabilities for query construction and is difficult to extend

MyBatis3Kotlin

- Generates Kotlin code
- Does not generate XML - MyBatis3 annotations are used exclusively
- The generated model objects are "flat" - there is no separate primary key object
- The generated code is dependent on the MyBatis Dynamic SQL Library
- The amount of generated code is relatively small
- The generated code allows tremendous flexibility in query construction

https://blog.csdn.net/weixin_41809435/article/details/85207563