MyBatis运行流程

使用MyBatis的步骤

1. 编写MyBatis配置文件

```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">
<!--对应Configuration类，由XMLConfigBuilder解析-->
<configuration>
    <!--放在Properties类型的variables属性里卖弄用来替换xml文件里面的${}占位符的-->
    <properties>
        <property name="jdbc.username" value="{username}"/>
    </properties>
    <settings>
        <setting name="cacheEnabled" value="true"/>
        <!--日志打印到控制台-->
        <setting name="logImpl" value="org.apache.ibatis.logging.stdout.StdOutImpl"/>

        <!--开启主键自增-->
        <setting name="useGeneratedKeys" value="true"/>

        <!--全局启用懒加载-->
        <setting name="lazyLoadingEnabled" value="true"/>
        <!--激进懒加载，
        为true时，对对象任一属性的读、写操作，都会触发该对象所有懒加载属性的加载
        为false时，对对象的某一个懒加载属性的读操作会触发该属性的加载-->
        <setting name="aggressiveLazyLoading" value="false"/>
    </settings>
    <!--对应TypeAliasRegistry类-->
    <typeAliases>
        <typeAlias type="com.chenx.learning.pojo.Address" alias="address"/>
        <package name="com.chenx.learning"/>    <!--由TypeAliasRegistry解析-->
    </typeAliases>
    <!--对应TypeHandlerRegistry类-->
    <typeHandlers>
        <typeHandler handler="com.chenx.learning.pojo.XxxHandler"/>
    </typeHandlers>
    <!--对应ObjectFactory类-->
    <objectFactory type="com.chenx.learning.XxxObjectFactory"/>
    <!--对应ObjectWrapperFactory类-->
    <objectWrapperFactory type="com.chenx.learning.Xxx"/>
    <!--对应ReflectorFactory类-->
    <reflectorFactory type="Xxx"/>
    <plugins>
        <plugin interceptor="com.chenx.learning.Xxxxx"> <!--对应Interceptor类-->
```

```
            <property name="dialect" value="mysql"/>
        </plugin>
    </plugins>

    <environments default="dev">
        <!--对应Environment类-->
        <environment id="dev">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="url" value="db_path"/>
                <property name="username" value="${jdbc.username}"/>
                <property name="password" value="${jdbc.password}"/>
            </dataSource>
        </environment>
    </environments>
    <!--对应DatabaseIdProvider类-->
    <databaseIdProvider type="com.chenx.learning.TestDatabaseIdProvider"/>
    <mappers>
        <mapper resource="com/chenx/learning/XxxDao.xml"/> <!--由XMlMapper Builder解析-->
        <package name="com.chenx.learning"/>
    </mappers>
</configuration>
```

2. 编写DAO层接口

```java
package com.chenx.learning.dao;


import com.chenx.learning.dao.provider.CustomerMapperProvider;
import com.chenx.learning.pojo.Customer;
import org.apache.ibatis.annotations.Param;
import org.apache.ibatis.annotations.SelectProvider;

import java.util.List;

public interface CustomerMapper {
    // 根据Id查询Customer（不查询Address）
    // @*Provider允许指定一个返回字符串的方法，来提供对应的sql
    @SelectProvider(type = CustomerMapperProvider.class, method = "findCustomerById")
    Customer find(@Param("id") long id);

    // 根据Id查询Customer（包含Address）
    Customer findWithAddress(long id);

    // 根据orderId查询Customer
    Customer findByOrderId(long orderId);

    // 持久化Customer对象
    int save(Customer customer);

    // 获取所有Customer
    List<Customer> findAllCustomer();

    // 获取指定customer，用于测试懒加载
    List<Customer> findCustomerLazyLoading(@Param("name") String name);

    /**
     * 获取Customer信息，用于测试整个查询流程，测试代码见{@link com.chenx.learning.exploreprocess.TestProcess#testSelectProcess()}
     */
    Customer selectCustomerWithAddress(@Param("customerId") Long customerId);
}

```

3. 编写DAO层接口对应映射文件

```xml
<?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="com.chenx.learning.dao.CustomerMapper">
    <!--自定义映射规则-->
    <resultMap id="customerSimpleMap" type="Customer">
        <!--主键映射-->
        <id column="id" property="id"/>
        <!--属性映射-->
        <result column="name" property="name"/>
        <result column="phone" property="phone"/>
    </resultMap>
    <resultMap id="customerMap" type="Customer">
        <!--主键映射-->
        <id column="id" property="id"/>
        <!--属性映射-->
        <id column="name" property="name"/>
        <id column="phone" property="phone"/>
        <!--映射Address集合-->
        <collection property="addresses" javaType="list" ofType="Address">
            <id column="address_id" property="id"/>
            <result column="street" property="street"/>
            <result column="city" property="city"/>
            <result column="country" property="country"/>
        </collection>
    </resultMap>

    <resultMap id="customerLazyLoadingMap" type="Customer">
        <id column="id" property="id"/>
        <!--属性映射-->
        <result column="name" property="name"/>
        <result column="phone" property="phone"/>
        <!--这个column表示查询参数-->
        <association property="addresses" column="id" select="com.chenx.learning.dao.AddressMapper.find"/>
    </resultMap>

    <sql id="demoSqlFragment">
        where id =
        #{id}
    </sql>
    <!--自定义SQL语句-->
    <!--    <select id="find" resultMap="customerSimpleMap">-->
    <!--        select * from t_customer-->
    <!--        /*测试include是怎么被sql标签替换的*/-->
    <!--        <include refid="demoSqlFragment"/>-->
```

```
    <!--    </select>-->
    <select id="findWithAddress" resultMap="customerMap">
        SELECT c.*, a.id as address_id, a.*
        FROM t_customer as c
                 join t_address as a
                      on c.id = a.customer_id
        WHERE c.id = #{id}
    </select>
    <!-- CustomerMapper接口中的findByOrderId()方法会执行该SQL,
        查询结果通过customerSimpleMap这个映射生成Customer对象-->
    <select id="findByOrderId" resultMap="customerSimpleMap">
        SELECT *
        FROM t_customer as c
                 join t_order as t
                      on c.id = t.customer_id
        WHERE t.customer_id = #{id}
    </select>
    <!-- 定义insert语句, CustomerMapper接口中的save()方法会执行该SQL,
        数据库生成的自增id会自动填充到传入的Customer对象的id字段中-->
    <insert id="save" keyProperty="id" useGeneratedKeys="true">
        insert into t_customer (id, name, phone)
        values (#{id}, #{name}, #{phone})
    </insert>

    <select id="findAllCustomer" resultType="Customer">
        select *
        from t_customer
    </select>

    <select id="findCustomerLazyLoading" resultMap="customerLazyLoadingMa
p">
        select *
        from t_customer
        where name = #{name}
    </select>
    <!--开启二级缓存-->
    <cache/>

    <select id="selectCustomerWithAddress" resultMap="customerMap">
        SELECT cust.*, addr.id, addr.street, addr.city, addr.country
        FROM `t_customer` as cust
                 left join t_address as addr on cust.id = addr.customer_id
        where cust.id = #{customerId}
    </select>
</mapper>
```

4. 使用SqlSession获取Mapper，调用对应接口操作数据库

```java
    @Test
    public void testSelectProcess() {
        String resource = "mybatis-config-sample.xml";
        try (
                InputStream configResource = Resources.getResourceAsStream(resource);
                SqlSession sqlSession = new SqlSessionFactoryBuilder().build(configResource).openSession();
        ) {
            CustomerMapper mapper = sqlSession.getMapper(CustomerMapper.class);
            Customer customer = mapper.selectCustomerWithAddress(1L);
            System.out.println(customer);
        } catch (IOException e) {
            // pass
        }
    }
```

运行流程

1. SqlSessionFactoryBuilder#builder(InputStream/Reader) 创建SqlSessionFactory。**所有xml文件的解析也是在这一步完成的。**

首先，SqlSessionFactoryBuilder的builder方法，会创建XMLConfigBuilder来解析MyBatis配置文件。XmlConfigBuilder类内聚合了一个XPathParser作为属性，用于解析xml文件。最终，解析配置文件的入口是XMLConfigBuilder的parse方法，该方法返回一个Configuration对象，SqlSessionFactoryBuilder的builder方法基于这个Configuration对象创建一个DefaultSqlSessionFactory对象。

所以整个XML配置文件、映射文件的解析，核心在于XMLConfigBuilder#parse方法。

在parse中，首先通过XMLConfigBuilder的parsed属性，防止配置文件重复解析。再调用聚合的XPathParser属性的evalNode方法，将配置文件的configuration节点解析成一个XNode对象。parse方法在调用parseConfiguration方法，来处理刚才XPathParser解析出来的XNode对象。

XMLConfigBuilder#parseConfiguration方法，是具体解析配置文件、映射文件的核心方法。该方法内调用每一个标签对应的处理方法。下面逐个解释该方法内调用的每一个方法。

- ⭐**propertiesElement(XNode)**：解析properties标签的方法。内部通过XNode#getChildrenAsProperties，将properties标签的每个子标签的name、value属性组成一个键值对存入Properties对象。最后将该Properties对象，存入Configuration对象的variables属性，同时也存入了XPathParser的variables属性中，用于在后续解析中用来替换配置文件的${}占位符。
- ⭐**settingsAsProperties(XNode)**：解析settings标签的方法。也是通过

XNode#getChildrenAsProperties方法，将所有子标签setting解析成Properties对象，通过判断Configuration是否有“键”对应的setter方法，来判断该setting是否存在，不存在则抛出异常。最后将Properties对象返回，用于后续操作使用。

- **loadCustomVfs(Properties)**：根据settings标签解析的具体setting节点，加载自定义虚拟文件系统的方法，参数就是上一步解析的settings标签解析出来的Properties对象。该方法做法就是将Configuration的vfsImpl属性设置为类似 `<setting name="vfsImpl" value="com.example.MyCustomVfs"/>` 指定的类并加到VFS的USER_IMPLEMENTATIONS属性这个列表中。这一块功能本人目前没有实际使用过。
- **loadCustomLogImpl(Properties)**：根据settings标签中是否配置name为logImpl的属性，决定使用哪一个作为日志对象。会将Configuration的logImpl属性设置为该标签value属性对应的类，然后将LogFactory类中用于创建具体的日志打印对象的logConstructor属性赋值为该类型的构造器。在这个标签中，value可以使用已注册的类型别名，处理过程中会进行别名解析。
- ⭐**typeAliasesElement(XNode)**：解析typeAliases标签。typeAliases标签下可能存在package或typeAlias两种标签。解析时会依次处理每一个子标签。对于package标签 `<package name="com.chenx"/>` ，name属性指定的包下的JavaBean，使用类注解@Alias指定别名，或默认使用首字母小写作为别名。对于typeAlias标签，使用alias属性作为type属性指定的类型的别名。最终都是将类型和别名注册到XMLConfigBuilder的父类的typeAliasRegistry属性中，该属性是一个TypeAliasRegistry类型，该类中有一个Map<String, Class<?>>类型的typeAliases，该属性是存储所有别名的地方。
- pluginElement(XNode)：解析plugins标签。依次处理每一个子标签节点。（用的不多，暂且省略）。
- **objectFactoryElement(XNode)**：处理objectFactory标签。可以通过该标签配置自定义的ObjectFactory的实现，用作对象工厂，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。（用的不多）
- **objectWrapperFactoryElement(XNode)**：处理objectFactoryWrapper标签。改变前用作对对象进行包装后处理。默认的ObjectWrapperFactory实现类DefaultObjectWrapperFactory没有实现任何功能。（用的不多）
- reflectorFactoryElement(XNode)：解析reflectorFactory标签。用于用户自定义ReflectorFactory的实现，替代默认实现DefaultReflectorFactory。是用于缓存Reflector，提高Reflector初始化速度的。
- ⭐**settingsElement(Properties)**：上面settingsAsProperties方法解析了settings标签，将其全部子标签转为了Properties对象，也就是本方法的入参。本方法依据settings中是否配置了设置，对Configuration对象中很大一部分属性赋值，如果配置了值就用用户配置的值，如果没有配置，就用默认的值。

- ⭐**environmentsElement(XNode)**：解析environments标签。如果在创建XMLConfigBuilder对象的时候，没有通过构造器指定environment属性的值，environments标签的default属性指定的值作为环境。依次处理每一个environment标签，如果标签的id和父标签的default属性指定的值一样，解析其中的transactionManager、dataSource标签，来构建对应的TransactionFactory、DataSourceFactory，最终存入Environment类实例中，设置为Configuration的environment属性。
- **databaseIdProviderElement(XNode)**：解析<databaseIdProvider>标签。该标签用于提供数据库标识，是Mybatis用于支持多厂商的， MyBatis 会加载带有匹配当前数据库 `databaseId` 属性和所有不带 `databaseId` 属性的语句。 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。（用的不多）。
- **typeHandlerElement(XNode)**：解析typeHandler标签。用来注册自定义类型处理器。也可以通过package指定包下有注解的某类。最终注册的自定义处理器和默认的自定义类型处理器会一起存放在XMLConfigBuilder的父类BaseBuilder的typeHandlerRegister属性中。
- ⭐**mapperElement(XNode)**：解析mappers标签，同时内部会通过XMLMapperBuilder加载映射文件。子标签可能是package或mapper，如果是package，就将该标签name属性对应的包下所有的映射器接口全部注册为映射器。如果是mapper标签，就根据其resource/class/url属性指定的映射文件加载，这三个属性只能使用一个，否则会抛出异常，一般用的是resource指定xml映射文件。方法内执行的操作，就是通过Resources#getResourceAsStream加载resource属性指定的映射文件。再根据当前的映射文件的流、全局配置对象、映射文件路径等创建XMLMapperBuilder类对象，并通过该对象的parse方法解析映射文件。下面详细介绍XMLMapperBuilder是如何解析映射文件的。

  首先，XMLMapperBuilder和XMLConfigBuilder一样，都有一个XPathParser类型的属性并且继承BaseBuilder类，XPathParser用于解析XML文件，最后会给出xml文件对应的XNode对象。而XMLMapperBuilder类还有一个MapperBuilderAssistant类型的属性，该类是一个辅助类，提供了一些方便使用的方法。下面介绍解析映射文件的入口，XMLMapperBuilder#parse()。

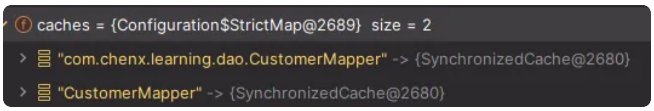
  首先会判断文件是否已经解析过，如果没有解析过才进入解析流程，否则只会执行处理上一次执行中出现的暂时性错误。如果为解析过，先调用configurationElement方法，解析映射文件的根节点mapper节点。

  在configurationElement方法中，首先会获取mapper根节点的namespace属性，该属性指定本映射文件对应的映射接口。然后解析调用cacheRefElement方法解析cache-ref节点，该节点表示引用其他映射文件的缓存，用于缓存共享。

  在cacheRefElement方法里，将本映射文件和cacheRef标签指定的缓存共享的命名空间添加到Configuration类的cacheRefMap属性中，实现绑定。下一个调用的方法是cacheElement，用于解析cache标签，也就是二级缓存相关的。

  在cacheElement方法中，如果映射文件包含了cache标签，就根据该标签中声明的属性对应的缓存配置和默认配置，调用MapperBuilderAssistant#useNewCache来初始化二级缓存，而

useNewCache方法中，就是根据标签中自定义的配置和默认配置来创建对应Cache对象的具体实现。并通过不同的装饰器器对该对象进行装饰，提供更丰富的缓存功能。创建该Cache对象是通过CacheBuilder建造者类实现的，该类的build方法中，调用了setStandardDecorators方法，在该方法中，通过cache标签的属性配置来判断，决定是否对Cache对象装饰更多的装饰器。其中LoggingCache和SynchronizedCache两个装饰器是一定加上的。最后，该Cache对象会被放入Configuration对象的caches属性中，该属性的类型是HashMap的子类，在添加元素是，会根据插入的键值对的键"."分割后的最后一段字符串存一个键值对，也就是说，存入caches属性的是xml映射文件的命名空间和命名空间对应映射接口名称两个键值对。如下图所示。



cacheElement方法调用完成后，调用parameterMapElement方法，解析parameterMap节点，官网上声明该节点是老式风格的参数映射，现在已经被废弃，所以不在介绍。

后面调用resultMapElements方法解析resultMap节点。resultMap节点用来自定义映射规则，并且提供了丰富、强大的拓展功能。resultMapElements方法会逐个解析resultMap节点，最后可以追踪到`resultMapElement(XNode resultMapNode, List<ResultMapping> additionalResultMappings, Class<?> enclosingType)`方法。该方法较为复杂，所以下面详细介绍一下。

首先，通过获取type、ofType、resultType、javaType的值，来找到对应的Java类型，四个属性都可以声明，优先级顺序如上述顺序。接下来，获取到该属性声明的类型对应的Class对象，然后逐个处理resultMap的子标签，可能是id、result、constructor、discriminator等。construct中的属性和普通的result属性的区别就是属性赋值是通过构造器还是setter。最终解析的所有标签都被封装成ResultMapping对象，存在一个名为resultMappings的List中。

重点看result标签是怎么解析的。因为id标签和result标签是一起解析的，所以在解析所有非constructor和discriminator标签时，会创建一个ArrayList来保存标志，如果是id标签，就向这个List中添加一个ResultFlag.ID，用来判断是不是constructor标签里的，如果是就用name获取属性名，如果不是就用property获取。然后调用buildResultMappingFromContext()方法，将返回的ResultMapping对象存放到上面创建的名为resultMappings的List中。下面看一下buildResultMappingFromContext是如何将每个子标签解析成ResultMapping对象的。

首先，获取子标签的property属性，作为目标属性名称，然后获取标签可能存在的各个属性值，包括column、javaType、jdbcType、select、fetchType(用于配置懒加载)等。然后通过MapperBuilderAssistant#buildResultMapping方法和获取的属性值，创建ResultMapping对象。最终每个子节点对应生成的ResultMapping对象都被放入哪个名为resultMappings的List中。最后，借助ResultMapResolver#resolve方法，创建ResultMap对象、处理继承关系，并将创建好的ResultMap对象添加到Configuration的resultMaps属性中。

至此，XMLMapperBuilder中的resultMapElements方法调用结束，后面调用sqlElement方法。

sqlElement方法处理sql标签，该标签用于提出sql片段，在数据库操作标签节点内部通过include标签引用，实现复用。这里不多介绍。

sqlElement方法调用结束后，调用buildStatementFromContext处理各个数据库操作语句。

buildStatementFromContext方法中，针对每一个数据库操作节点，通过XMLStatementBuilder类来处理，实际处理的方法是XMLStatementBuilder#parseStatementNode。

在此方法中，先通过databaseId和id两个标签属性的值，来实现兼容多种数据库的功能。然后获取标签的部分属性，再处理标签中可能存在的include标签引用sql片段情况，接着处理可能存在的selectKey标签(用于处理Oracle不支持自增id)。到这里，自增id和sql片段引用就处理完了。后面再通过XMLLanguageDriver创建SqlSource对象，SqlSource对象就是标签节点中的sql语句文本对应的对象，SqlSource提供了一个getBoundSql方法，该方法返回BoundSql对象，BoundSql对象就是参数绑定完成后的sql语句。再通过标签的属性和MapperBuilderAssistant#addMappedStatement方法，创建MappedStatement对象，并将其添加到Configuration的mappedStatement集合中。

至此，XMLMapperBuilder#configurationElement执行完成，接着将解析的该resource添加到Configuration的loadedResources中，防止资源重复解析，再调用bindMapperForNamespace方法，将mapper添加到Configuration的mapperRegistry属性中的knownMappers这个Map中，键为映射文件命名空间对应的类型的Class对象，值为根据该类型封装的MapperProxyFactory对象。

至此，XMLMapperBuilder的parse方法重要部分完成，该方法剩余部分会处理解析过程中的暂时性错误，是处理无序性依赖的。

SqlSessionFactoryBuilder#build方法执行结束。也意味着Mybatis解析配置文件和配置的映射文件完成。

2. SqlSessionFactory#openSession()获取SqlSession。

上一步根据Configuration对象创建的DefaultSqlSessionFactory的openSession会创建SqlSession对象。具体的创建逻辑如下。

首先，获取Configuration对象中的environment属性，类型是Environment。从environment中可以获取到事务工厂，通过事务工厂生成事务对象，再根据事务对象创建执行器Executor。最后根据Executor执行器、全局配置configuration和默认autocommit=false，创建DefaultSqlSession的对象。

这就是openSession方法拿到的SqlSession的具体实现类对象实例。

3. SqlSession#getMapper(Class<T>)获取XxxMapper的代理类实例。

SqlSession的getMapper方法实际就是调用Configuration#getMapper，最终是调用的MapperRegistry#getMapper。具体的执行逻辑是，从MapperRegistry的knownMappers这个Map中，根据传入的类型获取对应的MapperProxyFactory对象。这个MapeprProxyFactory的生成介绍，就是上

面黄色高亮的部分。**拿到这个MapperProxyFactory对象后通过他的newInstance(SqlSession)方法，生成MapperProxy对象，再根据这个MapperProxy对象，调用JDK动态代理，以该MapperProxy作为代理类，MapperProxyFactory中保存的映射接口类型，生成映射接口的代理对象，我们通过自动注入或者通过getMapper获取的映射接口的实例，就是这个代理对象**。

那么，重点就在于该MapperProxy实现InvocationHandler接口后，重写的invoke方法逻辑了。而最终跟踪到的方法，是MapperMethod#execute方法，这里根据sql的类型，真正的去执行sql

4. XxxMapper#Xxx()调用数据库操作。
5. 关闭流、会话。