二

11 鸟瞰 MyBatis 初始化，把握 MyBatis 启动流程脉络（下）

在上一讲，我们深入分析了MyBatis 初始化过程中对 mybatis-config.xml 全局配置文件的解析，详细介绍了其中每个标签的解析流程以及涉及的经典设计模式——构造者模式。这一讲我们就紧接着上一讲的内容，继续介绍 MyBatis 初始化流程，重点介绍Mapper.xml 配置文件的解析以及 SQL 语句的处理逻辑。

Mapper.xml 映射文件解析全流程

在上一讲分析 mybatis-config.xml 配置文件解析流程的时候我们看到，在 mybatis-config.xml 配置文件中可以定义多个 `<mapper>` 标签指定 Mapper 配置文件的地址，**MyBatis 会为每个 Mapper.xml 映射文件创建一个 XMLMapperBuilder 实例完成解析。**

与 XMLConfigBuilder 类似，XMLMapperBuilder也是具体构造者的角色，继承了 BaseBuilder 这个抽象类，解析 Mapper.xml 映射文件的入口是 XMLMapperBuilder.parse() 方法，其核心步骤如下：

- 执行 configurationElement() 方法解析整个Mapper.xml 映射文件的内容；
- 获取当前 Mapper.xml 映射文件指定的 Mapper 接口，并进行注册；
- 处理 configurationElement() 方法中解析失败的 `<resultMap>` 标签；
- 处理 configurationElement() 方法中解析失败的 `<cache-ref>` 标签；
- 处理 configurationElement() 方法中解析失败的SQL 语句标签。

可以清晰地看到，**configurationElement() 方法才是真正解析 Mapper.xml 映射文件的地方**，其中定义了处理 Mapper.xml 映射文件的核心流程：

- 获取 `<mapper>` 标签中的 namespace 属性，同时会进行多种边界检查；
- 解析 `<cache>` 标签；
- 解析 `<cache-ref>` 标签；

- 解析 `<resultMap>` 标签；
- 解析 `<sql>` 标签；
- 解析 `<select>`、`<insert>`、`<update>`、`<delete>` 等 SQL 标签。

下面我们就按照顺序逐一介绍这些方法的核心实现。

1. 处理 `<cache>` 标签

我们知道 Cache 接口及其实现是MyBatis 一级缓存和二级缓存的基础，其中，一级缓存是默认开启的，而二级缓存默认情况下并没有开启，如有需要，**可以通过标签为指定的 namespace 开启二级缓存。**

XMLMapperBuilder 中解析 `<cache>` 标签的**核心逻辑位于 cacheElement() 方法**之中，其具体步骤如下：

- 获取 `<cache>` 标签中的各项属性（type、flushInterval、size 等属性）；
- 读取 `<cache>` 标签下的子标签信息，这些信息将用于初始化二级缓存；
- MapperBuilderAssistant 会根据上述配置信息，创建一个全新的Cache 对象并添加到 Configuration.caches 集合中保存。

也就是说，解析 `<cache>` 标签得到的所有信息将会传给 MapperBuilderAssistant 完成 Cache 对象的创建，创建好的Cache 对象会添加到 Configuration.caches 集合中，**这个 caches 字段是一个StrictMap 类型的集合**，其中的 Key是Cache 对象的唯一标识，默认值是Mapper.xml 映射文件的namespace，Value 才是真正的二级缓存对应的 Cache 对象。

这里我们简单介绍一下 StrictMap的特性。

StrictMap 继承了 HashMap，并且覆盖了 HashMap 的一些行为，例如，相较于 HashMap 的 put() 方法，StrictMap 的 put() 方法有如下几点不同：

- 如果检测到重复 Key 的写入，会直接抛出异常；
- 在没有重复 Key的情况下，会正常写入 KV 数据，与此同时，还会根据 Key产生一个 shortKey，shortKey 与完整 Key 指向同一个 Value 值；
- 如果 shortKey 已经存在，则将 value 修改成 Ambiguity 对象，Ambiguity 对象表示这个 shortKey 存在二义性，后续通过 StrictMap的get() 方法获取该 shortKey 的时候，会抛出异常。

了解了 StrictMap 这个集合类的特性之后，我们回到MapperBuilderAssistant 这个类继续分析，在它的 useNewCache() 方法中，会根据前面解析得到的配置信息，通过

CacheBuilder 创建 Cache 对象。

通过名字你就能猜测到 CacheBuilder 是 Cache 的构造者，**CacheBuilder 中最核心的方法是build() 方法，其中会根据传入的配置信息创建底层存储数据的 Cache 对象以及相关的 Cache 装饰器**，具体实现如下：

```
public Cache build() {
    // 将implementation默认值设置为PerpetualCache，在decorators集合中默认添加LruCache装
    // 都是在setDefaultImplementations()方法中完成的
    setDefaultImplementations();
    // 通过反射，初始化implementation指定类型的对象
    Cache cache = newBaseCacheInstance(implementation, id);
    // 创建Cache关联的MetaObject对象，并根据properties设置Cache中的各个字段
    setCacheProperties(cache);
    // 根据上面创建的Cache对象类型，决定是否添加装饰器
    if (PerpetualCache.class.equals(cache.getClass())) {
        // 如果是PerpetualCache类型，则为其添加decorators集合中指定的装饰器
        for (Class<? extends Cache> decorator : decorators) {
            // 通过反射创建Cache装饰器
            cache = newCacheDecoratorInstance(decorator, cache);
            // 依赖MetaObject将properties中配置信息设置到Cache的各个属性中，同时调用Cac
            setCacheProperties(cache);
        }
        // 根据readWrite、blocking、clearInterval等配置，
        // 添加SerializedCache、ScheduledCache等装饰器
        cache = setStandardDecorators(cache);
    } else if (!LoggingCache.class.isAssignableFrom(cache.getClass())) {
        // 如果不是PerpetualCache类型，就是其他自定义类型的Cache，则添加一个LoggingCach
        cache = new LoggingCache(cache);
    }
```

```
    return cache;
}
```

2. 处理 <cache-ref> 标签

通过上述介绍我们知道，可以通过 <cache> 标签为每个 namespace 开启二级缓存，同时还会将 namespace 与关联的二级缓存 Cache对象记录到 Configuration.caches 集合中，也就是说二级缓存是 namespace 级别的。但是，在有的场景中，我们会需要在多个 namespace 共享同一个二级缓存，也就是**共享同一个 Cache 对象。**

为了解决这个需求，MyBatis提供了 <cache-ref> 标签来引用另一个 namespace 的二级缓存。cacheRefElement() 方法是处理 <cache-ref> 标签的核心逻辑所在，在 Configuration 中维护了一个 cacheRefMap 字段（HashMap<String,String> 类型），其中的 Key 是 <cache-ref> 标签所属的namespace 标识，Value 值是 <cache-ref> 标签引用的 namespace 值，这样的话，就可以将两个namespace 关联起来了，即这两个 namespace 共用一个 Cache对象。

这里会使用到一个叫 CacheRefResolver 的 Cache 引用解析器。**CacheRefResolver 中记录了被引用的 namespace以及当前 namespace 关联的MapperBuilderAssistant 对象。**前面在解析 <cache> 标签的时候我们介绍过，MapperBuilderAssistant 会在 useNewCache() 方法中通过 CacheBuilder 创建新的 Cache 对象，并记录到 currentCache 字段。而这里解析 <cache-ref> 标签的时候，MapperBuilderAssistant 会通过 useCacheRef() 方法从 Configuration.caches 集合中，根据被引用的namespace 查找共享的 Cache 对象来初始化 currentCache，而不再创建新的Cache 对象，从而实现二级缓存的共享。

3. 处理 <resultMap> 标签

有关系型数据库使用经验的同学应该知道，select 语句执行得到的结果集实际上是一张二维表，而 Java 是一门面向对象的程序设计语言，在使用 JDBC 的时候，我们需要手动写代码将select 语句的结果集转换成 Java 对象，这是一项重复性很大的操作。

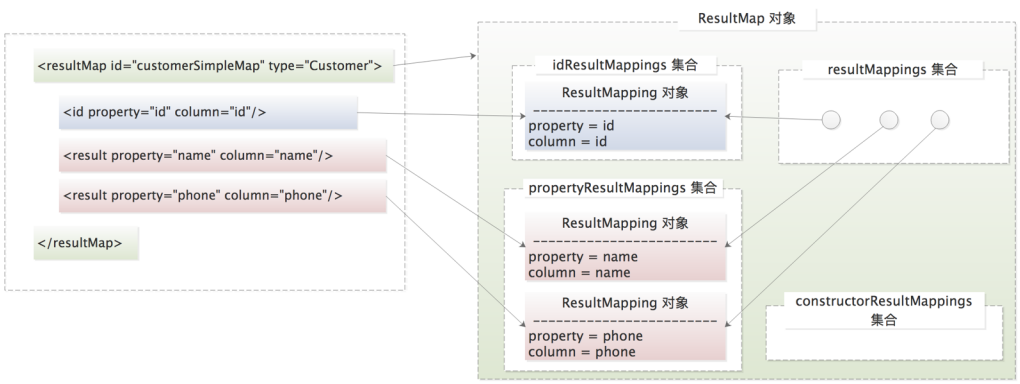
为了将 Java 开发者从这种重复性的工作中解脱出来，MyBatis 提供了 标签来定义结果集与 Java 对象之间的映射规则。

首先， <resultMap> 标签下的每一个子标签，例如， <column> 、 <id> 等，都被解析一个 ResultMapping 对象，其中维护了数据库表中一个列与对应 Java 类中一个属性之间的映射关系。

下面是 ResultMapping 中核心字段的含义。

- column（String 类型）：当前标签中指定的 column 属性值，指向的是数据库表中的一个列名（或是别名）。
- property（String 类型）：当前标签中指定的 property 属性值，指向的是与 column 列对应的属性名称。
- javaType（Class<?> 类型）、jdbcType（JdbcType 类型）：当前标签指定的 javaType 属性值和 jdbcType 属性值，指定了 property 字段的 Java 类型以及对应列的 JDBC 类型。
- typeHandler（TypeHandler<?> 类型）：当前标签的 typeHandler 属性值，这里指定的 TypeHandler 会覆盖默认的类型处理器。
- nestedResultMapId（String类型）：当前标签的 resultMap 属性值，通过该属性我们可以引用另一个 `<resultMap>` 标签的id，然后由这个被引用的 `<resultMap>` 标签映射结果集中的一部分列。这样，我们就可以将一个查询结果集映射成多个对象，同时确定这些对象之间的关联关系。
- nestedQueryId（String 类型）：当前标签的select 属性，我们可以通过该属性引用另一个 `<select>` 标签中的select 语句定义，它会将当前列的值作为参数传入这个 select 语句。由于当前结果集可能查询出多行数据，那么可能就会导致 select 属性指定的 SQL 语句会执行多次，也就是著名的 N+1 问题。
- columnPrefix（String 类型）：当前标签的 columnPrefix 属性值，记录了表中列名的公共前缀。
- resultSet（String 类型）：当前标签的 resultSet 属性值。
- lazy（boolean 类型）：当前标签的fetchType 属性，表示是否延迟加载当前标签对应的列。

介绍完 ResultMapping 对象（即 `<resultMap>` 标签下各个子标签的解析结果）之后，我们再来看 `<resultMap>` 标签如何被解析。整个 `<resultMap>` 标签最终会被解析成 ResultMap 对象，它与 ResultMapping 之间的映射关系如下图所示：



ResultMap 结构图

通过上图我们可以看出，ResultMap 中有四个集合与 ResultMapping 紧密相连。

- resultMappings 集合，维护了整个 `<resultMap>` 标签解析之后得到的全部映射关系，也就是全部 ResultMapping 对象。
- idResultMappings 集合，维护了与唯一标识相关的映射，例如，`<id>` 标签、`<constructor>` 标签下的 `<idArg>` 子标签解析得到的 ResultMapping 对象。如果没有定义 `<id>` 等唯一性标签，则由 resultMappings 集合中全部映射关系来确定一条记录的唯一性，即 idResultMappings 集合与 resulMappings 集合相同。
- constructorResultMappings 集合，维护了 `<constructor>` 标签下全部子标签定义的映射关系。
- propertyResultMappings 集合，维护了不带 Constructor 标志的映射关系。

除了上述四个 ResultMapping 集合，ResultMap 中还维护了下列核心字段。

- id（String 类型）：当前 `<resultMap>` 标签的 id 属性值。
- type（Class 类型）：当前 `<resultMap>` 的 type 属性值。
- mappedColumns（Set `<String>` 类型）：维护了所有映射关系中涉及的 column 属性值，也就是所有的列名（或别名）。
- hasNestedResultMaps（boolean 类型）：当前 `<resultMap>` 标签是否嵌套了其他 `<resultMap>` 标签，即这个映射关系中指定了 resultMap属性，且未指定 resultSet 属性。
- hasNestedQueries（boolean 类型）：当前 `<resultMap>` 标签是否含有嵌套查询。也就是说，这个映射关系中是否指定了 select 属性。
- autoMapping（Boolean 类型）：当前 ResultMap 是否开启自动映射的功能。
- discriminator（Discriminator 类型）：对应 `<discriminator>` 标签。

接下来我们开始深入分析 `<resultMap>` 标签解析的流程。XMLMapperBuilder的 resultMapElements() 方法负责解析 Mapper 配置文件中的全部 `<resultMap>` 标签，其中会通过 resultMapElement() 方法解析单个 `<resultMap>` 标签。

下面是 resultMapElement() 方法解析 `<resultMap>` 标签的核心流程。

- 获取 `<resultMap>` 标签的type 属性值，这个值表示结果集将被映射成 type 指定类型的对象。如果没有指定 type 属性的话，会找其他属性值，优先级依次是：type、

ofType、resultType、javaType。在这一步中会确定映射得到的对象类型，这里支持别名转换。

- 解析 `<resultMap>` 标签下的各个子标签，每个子标签都会生成一个ResultMapping 对象，这个 ResultMapping 对象会被添加到resultMappings 集合（List `<ResultMapping>` 类型）中暂存。这里会涉及 `<id>` 、 `<result>` 、 `<association>` 、 `<collection>` 、 `<discriminator>` 等子标签的解析。
- 获取 `<resultMap>` 标签的id 属性，默认值会拼装所有父标签的id、value 或 property 属性值。
- 获取 `<resultMap>` 标签的extends、autoMapping 等属性。
- 创建 ResultMapResolver 对象，ResultMapResolver 会根据上面解析到的 ResultMappings 集合以及 `<resultMap>` 标签的属性构造 ResultMap 对象，并将其添加到 Configuration.resultMaps 集合（StrictMap 类型）中。

（1）解析 `<id>` 、 `<result>` 、 `<constructor>` 标签

在 resultMapElement() 方法中获取到 id 属性和 type 属性值之后，会调用 buildResultMappingFromContext() 方法解析上述标签得到 ResultMapping 对象，其核心逻辑如下：

- 获取当前标签的property的属性值作为目标属性名称（如果 `<constructor>` 标签使用的是 name 属性）；
- 获取 column、javaType、typeHandler、jdbcType、select 等一系列属性，与获取 property 属性的方式类似；
- 根据上面解析到的信息，调用 MapperBuilderAssistant.buildResultMapping() 方法创建 ResultMapping 对象。

正如 resultMapElement() 方法核心步骤描述的那样，经过解析得到 ResultMapping 对象集合之后，会记录到resultMappings 这个临时集合中，然后由 ResultMapResolver 调用 MapperBuilderAssistant.addResultMap() 方法创建 ResultMap 对象，将resultMappings 集合中的全部 ResultMapping 对象添加到其中，然后将ResultMap 对象记录到 Configuration.resultMaps 集合中。

下面是 MapperBuilderAssistant.addResultMap() 的具体实现：

```
public ResultMap addResultMap(
        String id,
        Class<?> type,
```

```
        String extend,
        Discriminator discriminator,
        List<ResultMapping> resultMappings,
        Boolean autoMapping) {
    // ResultMap的完整id是"namespace.id"的格式
    id = applyCurrentNamespace(id, false);
    // 获取被继承的ResultMap的完整id，也就是父ResultMap对象的完整id
    extend = applyCurrentNamespace(extend, true);
    if (extend != null) {  // 针对extend属性的处理
        // 检测Configuration.resultMaps集合中是否存在被继承的ResultMap对象
        if (!configuration.hasResultMap(extend)) {
            throw new IncompleteElementException("Could not find a parent resultmap
        }
        // 获取需要被继承的ResultMap对象，也就是父ResultMap对象
        ResultMap resultMap = configuration.getResultMap(extend);
        // 获取父ResultMap对象中记录的ResultMapping集合
        List<ResultMapping> extendedResultMappings = new ArrayList<>(resultMap.getR
        // 删除需要覆盖的ResultMapping集合
        extendedResultMappings.removeAll(resultMappings);
        // 如果当前<resultMap>标签中定义了<constructor>标签，则不需要使用父ResultMap中i
        // 的相应<constructor>标签，这里会将其对应的ResultMapping对象删除
        boolean declaresConstructor = false;
        for (ResultMapping resultMapping : resultMappings) {
            if (resultMapping.getFlags().contains(ResultFlag.CONSTRUCTOR)) {
                declaresConstructor = true;
                break;
            }
        }
```

```
            if (declaresConstructor) {
                extendedResultMappings.removeIf(resultMapping -> resultMapping.getFlags
            }
            // 添加需要被继承下来的ResultMapping对象记录到resultMappings集合中
            resultMappings.addAll(extendedResultMappings);
        }
        // 创建ResultMap对象，并添加到Configuration.resultMaps集合中保存
        ResultMap resultMap = new ResultMap.Builder(configuration, id, type, resultMapp
                .discriminator(discriminator)
                .build();
        configuration.addResultMap(resultMap);
        return resultMap;
    }
```

至于 `<constructor>` 标签的流程，是由XMLMapperBuilder 中的 processConstructorElement() 方法实现，其中会先获取 `<constructor>` 标签的全部子标签，然后为每个标签添加 CONSTRUCTOR 标志（为每个 `<idArg>` 标签添加额外的ID标志），最后通过 buildResultMappingFromContext()方法创建 ResultMapping对象并记录到 resultMappings 集合中暂存，这些 ResultMapping 对象最终也会添加到前面介绍的 ResultMap 对象。

（2）解析 `<association>` 和 `<collection>` 标签

接下来，我们来介绍解析 `<association>` 和 `<collection>` 标签的核心流程，两者解析的过程基本一致。前面介绍的 buildResultMappingFromContext() 方法不仅完成了 `<id>`、`<result>` 等标签的解析，还完成了 `<association>` 和 `<collection>` 标签的解析，其中相关的代码片段如下：

```
    private ResultMapping buildResultMappingFromContext(XNode context, Class<?> resultT
        ... // <association>标签中其他属性的解析与<result>、<id>标签类似，这里不再展开
        // 如果<association>标签没有指定resultMap属性，那么就是匿名嵌套映射，需要通过
        //  processNestedResultMappings()方法解析该匿名的嵌套映射
```

```
    String nestedResultMap = context.getStringAttribute("resultMap", () ->
            processNestedResultMappings(context, Collections.emptyList(), resultTyp
    ... // <association>标签中其他属性的解析与<result>、<id>标签类似，这里不再展开
    // 根据上面解析到的属性值，创建ResultMapping对象
    return builderAssistant.buildResultMapping(resultType, property, column, javaTy
}
```

这里的 processNestedResultMappings() 方法会递归执行resultMapElement() 方法解析 `<association>` 标签和 `<collection>` 标签指定的匿名嵌套映射，得到一个完整的 ResultMap 对象，并添加到Configuration.resultMaps集合中。

(3) 解析 `<discriminator>` 标签

最后一个要介绍的是 `<discriminator>` 标签的解析过程，我们将 `<discriminator>` 标签与 `<case>` 标签配合使用，根据结果集中某列的值改变映射行为。从 resultMapElement() 方法的逻辑我们可以看出， `<discriminator>` 标签是由 processDiscriminatorElement() 方法专门进行解析的，具体实现如下：

```
private Discriminator processDiscriminatorElement(XNode context, Class<?> resultTyp
    // 从<discriminator>标签中解析column、javaType、jdbcType、typeHandler四个属性的逻辑
    Map<String, String> discriminatorMap = new HashMap<>();
    // 解析<discriminator>标签的<case>子标签
    for (XNode caseChild : context.getChildren()) {
        String value = caseChild.getStringAttribute("value");
        // 通过前面介绍的processNestedResultMappings()方法，解析<case>标签，
        // 创建相应的嵌套ResultMap对象
        String resultMap = caseChild.getStringAttribute("resultMap",
                processNestedResultMappings(caseChild, resultMappings, resultType))
        // 记录该列值与对应选择的ResultMap的Id
        discriminatorMap.put(value, resultMap);
    }
    // 创建Discriminator对象
```

```
    return builderAssistant.buildDiscriminator(resultType, column, javaTypeClass, j
}
```

SQL 语句解析全流程

在 Mapper.xml 映射文件中，除了上面介绍的标签之外，还有一类比较重要的标签，那就是 `<select>`、`<insert>`、`<delete>`、`<update>` 等 SQL 语句标签。虽然定义在 Mapper.xml 映射文件中，但是**这些标签是由 XMLStatementBuilder 进行解析的**，而不再由 XMLMapperBuilder 来完成解析。

在开始介绍 XMLStatementBuilder 解析 SQL 语句标签的具体实现之前，我们先来了解一下 MyBatis 在内存中是如何表示这些 SQL 语句标签的。在内存中，MyBatis 使用 SqlSource 接口来表示解析之后的 SQL 语句，其中的 SQL 语句只是一个中间态，可能包含动态 SQL 标签或占位符等信息，无法直接使用。SqlSource 接口的定义如下：

```
public interface SqlSource {
    // 根据Mapper文件或注解描述的SQL语句，以及传入的实参，返回可执行的SQL
    BoundSql getBoundSql(Object parameterObject);
}
```

MyBatis 在内存中使用 MappedStatement 对象表示上述 SQL 标签。在 MappedStatement 中的 sqlSource 字段记录了 SQL 标签中定义的 SQL 语句，sqlCommandType 字段记录了 SQL 语句的类型（INSERT、UPDATE、DELETE、SELECT 或 FLUSH 类型）。

介绍完表示 SQL 标签的基础类之后，我们来分析 XMLStatementBuilder 解析 SQL 标签的入口方法—— parseStatementNode() 方法，在该方法中首先会根据 id 属性和 databaseId 属性决定加载匹配的 SQL 标签，然后解析其中的 `<include>` 标签和 `<selectKey>` 标签，相关的代码片段如下：

```
public void parseStatementNode() {
    // 获取SQL标签的id以及databaseId属性
    String id = context.getStringAttribute("id");
    String databaseId = context.getStringAttribute("databaseId");
    // 若databaseId属性值与当前使用的数据库不匹配，则不加载该SQL标签
    // 若存在相同id且databaseId不为空的SQL标签，则不再加载该SQL标签
```

```
        if (!databaseIdMatchesCurrent(id, databaseId, this.requiredDatabaseId)) {
            return;
        }
        // 根据SQL标签的名称决定其SqlCommandType
        String nodeName = context.getNode().getNodeName();
        SqlCommandType sqlCommandType = SqlCommandType.valueOf(nodeName.toUpperCase(Loc
        // 获取SQL标签的属性值，例如，fetchSize、timeout、parameterType、parameterMap、
        // resultMap、resultType、lang、resultSetType、flushCache、useCache等。
        // 这些属性的具体含义在MyBatis官方文档中已经有比较详细的介绍了，这里不再赘述
        ... ...
        // 在解析SQL语句之前，先处理其中的<include>标签
        XMLIncludeTransformer includeParser = new XMLIncludeTransformer(configuration,
        includeParser.applyIncludes(context.getNode());
        // 获取SQL标签的parameterType、lang两个属性
        ... ...
        // 解析<selectKey>标签
        processSelectKeyNodes(id, parameterTypeClass, langDriver);
        // 暂时省略后面的逻辑
        ...
    }
```

1. 处理 `<include>` 标签

在实际应用中，我们会在 `<sql>` 标签中定义一些能够被重用的SQL 片段，在 XMLMapperBuilder.sqlElement() 方法中会根据当前使用的 DatabaseId 匹配 `<sql>` 标签，只有匹配的 SQL 片段才会被加载到内存。

在解析 SQL 标签之前，MyBatis 会先将 `<include>` 标签转换成对应的 SQL 片段（即定义在 `<sql>` 标签内的文本），这个转换过程是在 XMLIncludeTransformer.applyIncludes() 方法中实现的（其中不仅包含了 `<include>` 标签的处理，还包含了“${}”占位符的处理）。

针对 `<include>` 标签的处理如下：

- 查找 refid 属性指向的 `<sql>` 标签，得到其对应的 Node 对象；
- 解析 `<include>` 标签下的 `<property>` 标签，将得到的键值对添加到 variablesContext 集合（Properties 类型）中，并形成新的 Properties 对象返回，用于替换占位符；
- 递归执行 applyIncludes()方法，因为在 `<sql>` 标签的定义中可能会使用 `<include>` 引用其他 SQL 片段，在 applyIncludes()方法递归的过程中，如果遇到“${}”占位符，则使用 variablesContext 集合中的键值对进行替换；
- 最后，将 `<include>` 标签替换成 `<sql>` 标签的内容。

通过上面逻辑可以看出，`<include>` 标签和 `<sql>` 标签是可以嵌套多层的，此时就会涉及 applyIncludes()方法的递归，同时可以配合“${}”占位符，实现 SQL 片段模板化，更大程度地提高 SQL 片段的重用率。

2. 处理 `<selectKey>` 标签

在有的数据库表设计场景中，我们会添加一个自增 ID 字段作为主键，例如，用户 ID、订单 ID 或者这个自增 ID 本身并没有什么业务含义，只是一个唯一标识而已。在某些业务逻辑里面，我们希望在执行 insert 语句的时候返回这个自增 ID 值，`<selectKey>` 标签就可以实现自增 ID 的获取。`<selectKey>` 标签不仅可以获取自增 ID，还可以指定其他 SQL 语句，从其他表或执行数据库的函数获取字段值。

parseSelectKeyNode() 方法是解析 标签的核心所在，其中会解析 `<selectKey>` 标签的各个属性，并根据这些属性值将其中的 SQL 语句解析成 MappedStatement 对象，具体实现如下：

```
private void parseSelectKeyNode(String id, XNode nodeToHandle, Class<?> parameterTy
    ... // 解析<selectKey>标签的resultType、statementType、keyProperty等属性
    // 通过LanguageDriver解析<selectKey>标签中的SQL语句，得到对应的SqlSource对象
    SqlSource sqlSource = langDriver.createSqlSource(configuration, nodeToHandle, p
    SqlCommandType sqlCommandType = SqlCommandType.SELECT;
    // 创建MappedStatement对象
    builderAssistant.addMappedStatement(id, sqlSource, statementType, sqlCommandTyp
            fetchSize, timeout, parameterMap, parameterTypeClass, resultMap, result
            resultSetTypeEnum, flushCache, useCache, resultOrdered,
```

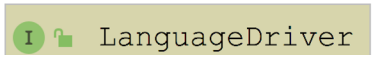
```
            keyGenerator, keyProperty, keyColumn, databaseId, langDriver, null);
    id = builderAssistant.applyCurrentNamespace(id, false);
    // 创建<selectKey>标签对应的KeyGenerator对象，这个KeyGenerator对象会添加到Configur
    MappedStatement keyStatement = configuration.getMappedStatement(id, false);
    configuration.addKeyGenerator(id, new SelectKeyGenerator(keyStatement, executeB
}
```

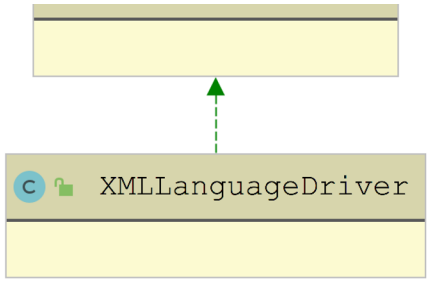
3. 处理 SQL 语句

经过 `<include>` 标签和 `<selectKey>` 标签的处理流程之后，XMLStatementBuilder 中的 parseStatementNode()方法接下来就要开始处理 SQL 语句了，相关的代码片段之前被省略了，这里我们详细分析一下：

```
public void parseStatementNode() {
    // 前面是解析<selectKey>和<include>标签的逻辑，这里不再展示
    // 当执行到这里的时候，<selectKey>和<include>标签已经被解析完毕，并删除掉了
    // 下面是解析SQL语句的逻辑，也是parseStatementNode()方法的核心
    // 通过LanguageDriver.createSqlSource()方法创建SqlSource对象
    SqlSource sqlSource = langDriver.createSqlSource(configuration, context, parame
    // 获取SQL标签中配置的resultSets、keyProperty、keyColumn等属性，以及前面解析<select
    // 这些信息将会填充到MappedStatement对象中
    // 根据上述属性信息创建MappedStatement对象，并添加到Configuration.mappedStatements
    builderAssistant.addMappedStatement(id, sqlSource, statementType, sqlCommandTyp
            fetchSize, timeout, parameterMap, parameterTypeClass, resultMap, result
            resultSetTypeEnum, flushCache, useCache, resultOrdered,
            keyGenerator, keyProperty, keyColumn, databaseId, langDriver, resultSet
}
```

这里解析 SQL 语句**使用的是 LanguageDriver 接口**，其核心实现是 XMLLanguageDriver，继承关系如下图所示：





LanguageDriver 继承关系图

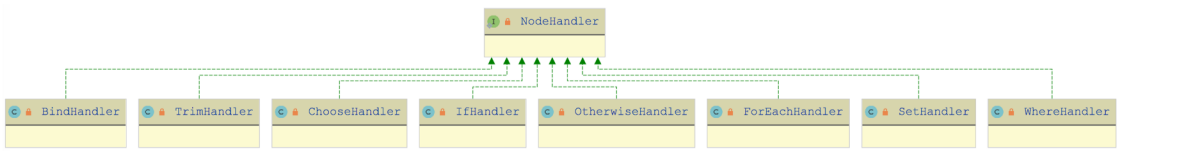
在 createSqlSource() 方法中，XMLLanguageDriver 会依赖 XMLScriptBuilder 创建 SqlSource 对象，XMLScriptBuilder 首先会判断 SQL 语句是否为动态SQL，判断的核心逻辑在 parseDynamicTags()方法中，核心实现如下：

```
protected MixedSqlNode parseDynamicTags(XNode node) {
    List<SqlNode> contents = new ArrayList<>(); // 解析后的SqlNode结果集合
    NodeList children = node.getNode().getChildNodes();
    // 获取SQL标签下的所有节点，包括标签节点和文本节点
    for (int i = 0; i < children.getLength(); i++) {
        XNode child = node.newXNode(children.item(i));
        if (child.getNode().getNodeType() == Node.CDATA_SECTION_NODE ||
                child.getNode().getNodeType() == Node.TEXT_NODE) {
            // 处理文本节点，也就是SQL语句
            String data = child.getStringBody("");
            TextSqlNode textSqlNode = new TextSqlNode(data);
            // 解析SQL语句，如果含有未解析的"${}"占位符，则为动态SQL
            if (textSqlNode.isDynamic()) {
                contents.add(textSqlNode);
                isDynamic = true; // 标记为动态SQL语句
            } else {
                contents.add(new StaticTextSqlNode(data));
            }
        } else if (child.getNode().getNodeType() == Node.ELEMENT_NODE) {
```

```
                // 如果解析到一个子标签，那么一定是动态SQL
                // 这里会根据不同的标签，获取不同的NodeHandler，然后由NodeHandler进行后续解析
                String nodeName = child.getNode().getNodeName();
                NodeHandler handler = nodeHandlerMap.get(nodeName);
                if (handler == null) {
                    throw new BuilderException("Unknown element <" + nodeName + "> in S
                }
                // 处理动态SQL语句，并将解析得到的SqlNode对象记录到contents集合中
                handler.handleNode(child, contents);
                isDynamic = true;
            }
        }
        // 解析后的SqlNode集合将会被封装成MixedSqlNode返回
        return new MixedSqlNode(contents);
    }
```

这里使用 SqlNode 接口来表示一条 SQL 语句的不同部分，其中，TextSqlNode 表示的是 SQL 语句的文本（可能包含“${}”占位符），StaticTextSqlNode 表示的是不包含占位符的 SQL 语句文本。

另外一个新接口是NodeHandler，它有很多实现类，如下图所示：



NodeHandler 继承关系图

NodeHandler接口负责解析动态 SQL 内的标签，生成相应的 SqlNode 对象，通过 NodeHandler 实现类的名称，我们就可以大概猜测到其解析的标签名称。以 IfHandler 为例，它解析的就是 `<if>` 标签，其核心实现如下：

```
    private class IfHandler implements NodeHandler {
```

```
        public void handleNode(XNode nodeToHandle, List<SqlNode> targetContents) {
            // 通过parseDynamicTags()方法，解析<if>标签下嵌套的动态SQL
            MixedSqlNode mixedSqlNode = parseDynamicTags(nodeToHandle);
            // 获取<if>标签判断分支的条件
            String test = nodeToHandle.getStringAttribute("test");
            // 创建IfNode对象(也是SqlNode接口的实现)，并将其保存下来
            IfSqlNode ifSqlNode = new IfSqlNode(mixedSqlNode, test);
            targetContents.add(ifSqlNode);
        }
    }
```

完成了对 SQL 语句的解析，得到了相应的 MixedSqlNode对象之后，XMLScriptBuilder 会根据 SQL 语句的类型生成不同的 SqlSource 实现：

```
    public SqlSource parseScriptNode() {
        // 对SQL语句进行解析
        MixedSqlNode rootSqlNode = parseDynamicTags(context);
        SqlSource sqlSource;
        if (isDynamic) { // 根据该SQL是否为动态SQL，创建不同的SqlSource实现
            sqlSource = new DynamicSqlSource(configuration, rootSqlNode);
        } else {
            sqlSource = new RawSqlSource(configuration, rootSqlNode, parameterType);
        }
        return sqlSource;
    }
```

总结

这一讲我们重点介绍了 MyBatis 在初始化过程中对 Mapper.xml 映射文件的解析。

首先，我们着重介绍了 Mapper.xml 映射文件中对 `<cache>` 标签、 `<cache-ref>` 标签以及 `<resultMap>` 标签（包括它的各个子标签）的解析流程，让我们知道 MyBatis是如何正确理解二级缓存的配置信息以及我们定义的各种映射规则。

然后，我们详细分析了 MyBatis 对 Mapper.xml 映射文件中 SQL 语句标签的解析，其中涉及 `<include>` 、 `<selectKey>` 等标签的处理逻辑。

上一页 下一页