

你知道MyBatis执行过程之初始化是如何执行的吗？

题目标签

学习时长：20分钟

题目难度：中等

知识点标签：MyBatis、执行过程

题目描述

你知道MyBatis执行过程之初始化是如何执行的吗？

题目解决

在了解MyBatis架构以及核心内容分析后，我们可以研究MyBatis执行过程，包括

- **MyBatis初始化**
- **SQL执行过程**

而且在面试会问到一下关于MyBatis初始化的问题，比如：

- **Mybatis需要初始化哪些？**
- **MyBatis初始化的过程？**

认准一手QQ3195303913微信wxywd8

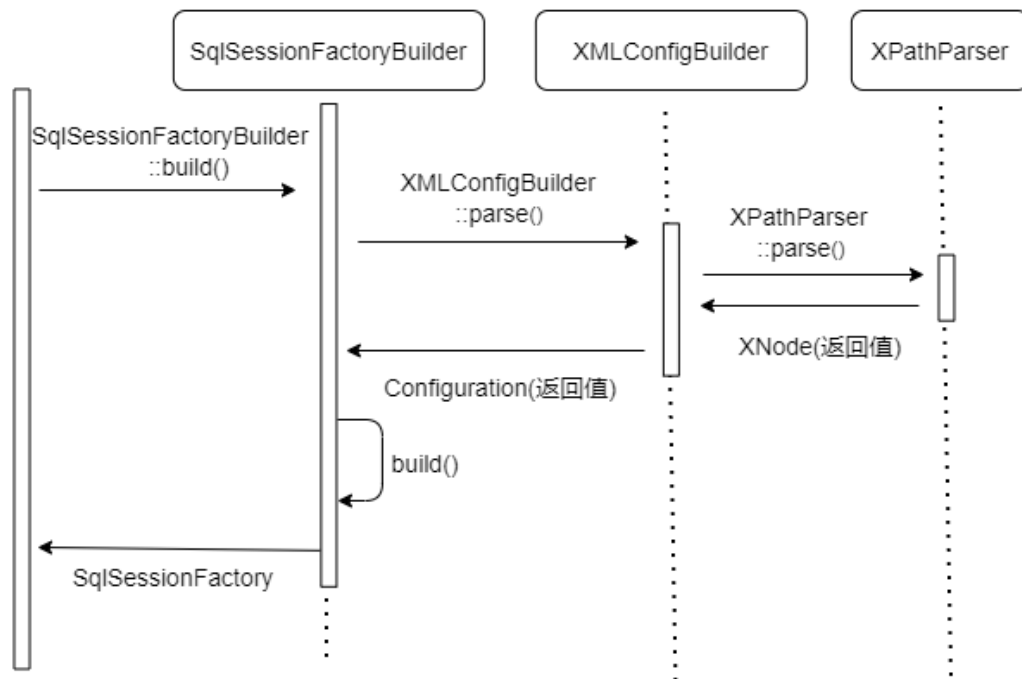
MyBatis初始化

在 MyBatis 初始化过程中，会加载 `mybatis-config.xml` 配置文件、`Mapper.xml` 映射配置文件以及 Mapper 接口中的注解信息，解析后的配置信息会形成相应的对象并保存到 `Configuration` 对象中。初始化过程可以分成三部分：

- 解析 `mybatis-config.xml` 配置文件
 - `SqlSessionFactoryBuilder`
 - `XMLConfigBuilder`
 - `Configuration`
- 解析 `Mapper.xml` 映射配置文件
 - `XMLMapperBuilder::parse()`
 - `XMLStatementBuilder::parseStatementNode()`
 - `XMLLanguageDriver`
 - `SqlSource`
 - `MappedStatement`
- 解析Mapper接口中的注解
 - `MapperRegistry`
 - `MapperAnnotationBuilder::parse()`

解析 `mybatis-config.xml` 配置文件

MyBatis 的初始化流程的入口是 `SqlSessionFactoryBuilder::build(Reader reader, String environment, Properties properties)` 方法, 看看具体流程图:



```
public SqlSessionFactory build(InputStream inputStream, String environment,
Properties properties) {
    try {
        XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment,
properties);
        return build(parser.parse());
    } catch (Exception e) {
        throw ExceptionFactory.wrapException("Error building SqlSession.", e);
    } finally {
        ErrorContext.instance().reset();
        try {
            inputStream.close();
        } catch (IOException e) {
            // Intentionally ignore. Prefer previous error.
        }
    }
}
```

123456789101112131415

首先会使用 `XMLConfigBuilder::parse()` 解析 `mybatis-config.xml` 配置文件,

- 先解析标签 `configuration` 内的数据封装成 `XNode`, `configuration` 也是 MyBatis 中最重要的一个标签
- 根据 `XNode` 解析 `mybatis-config.xml` 配置文件的各个标签转变为各个对象

```
private void parseConfiguration(XNode root) {
    try {
        //issue #117 read properties first
        propertiesElement(root.evalNode("properties"));
        Properties settings = settingsAsProperties(root.evalNode("settings"));
        loadCustomVfs(settings);
    }
```

```

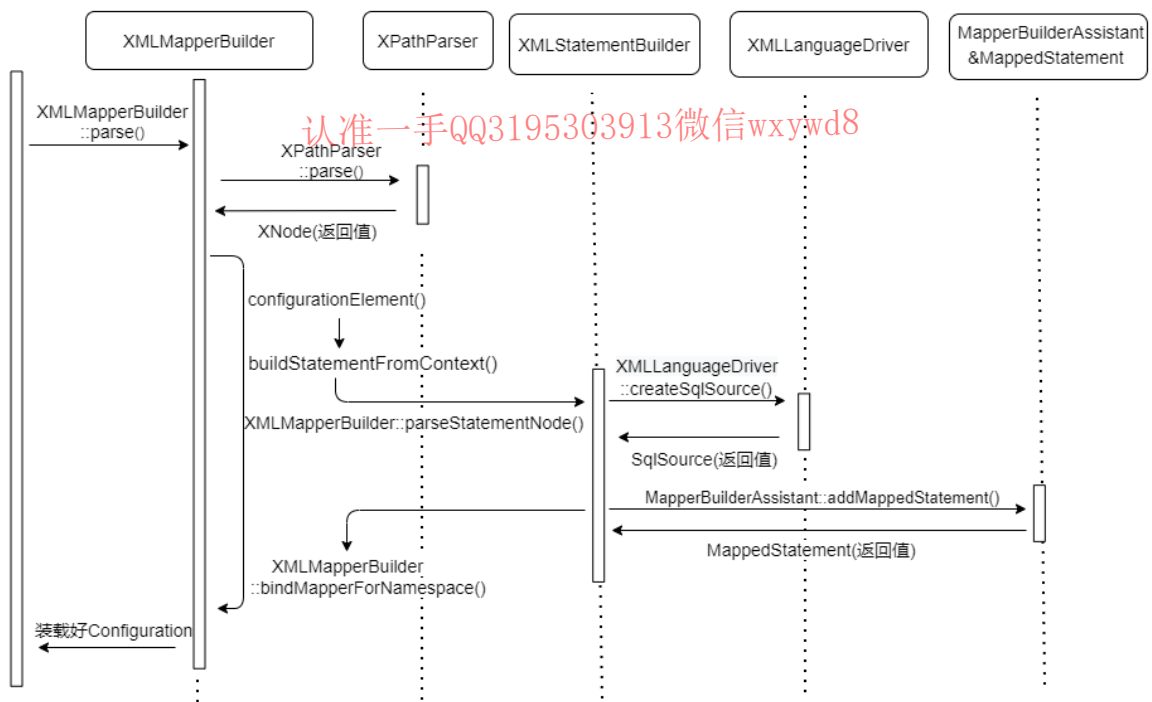
typeAliasesElement(root.evalNode("typeAliases"));
pluginElement(root.evalNode("plugins"));
objectFactoryElement(root.evalNode("objectFactory"));
objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
reflectorFactoryElement(root.evalNode("reflectorFactory"));
settingsElement(settings);
// read it after objectFactory and objectWrapperFactory issue #631
environmentsElement(root.evalNode("environments"));
databaseIdProviderElement(root.evalNode("databaseIdProvider"));
typeHandlerElement(root.evalNode("typeHandlers"));
mapperElement(root.evalNode("mappers"));
} catch (Exception e) {
    throw new BuilderException("Error parsing SQL Mapper Configuration. Cause:
" + e, e);
}
}
123456789101112131415161718192021

```

再基于 Configuration 使用 `SqlSessionFactoryBuilder::build()` 生成 `DefaultSqlSessionFactory` 供给后续执行使用。

解析 Mapper.xml 映射配置文件

首先使用 `XMLMapperBuilder::parse()` 解析 Mapper.xml，看看加载流程图来分析分析



通过 `XPathParser::evalNode` 将 `mapper` 标签中内容解析到 `XNode`

```

public void parse() {
    if (!this.configuration.isResourceLoaded(this.resource)) {
        this.configurationElement(this.parser.evalNode("/mapper"));
        this.configuration.addLoadedResource(this.resource);
        this.bindMapperForNamespace();
    }

    this.parsePendingResultMaps();
    this.parsePendingCacheRefs();
    this.parsePendingStatements();
}
1234567891011

```

再由 configurationElement() 方法去解析 xNode 中的各个标签：

- namespace
- parameterMap
- resultMap
- select|insert|update|delete

```

private void configurationElement(XNode context) {
    try {
        String namespace = context.getStringAttribute("namespace");
        if (namespace == null || namespace.equals("")) {
            throw new BuilderException("Mapper's namespace cannot be empty");
        }
        builderAssistant.setCurrentNamespace(namespace);
        cacheRefElement(context.evalNode("cache-ref"));
        cacheElement(context.evalNode("cache"));
        parameterMapElement(context.evalNodes("/mapper/parameterMap"));
        resultMapElements(context.evalNodes("/mapper/resultMap"));
        sqlElement(context.evalNodes("/mapper/sql"));
        //解析MapperState

        buildStatementFromContext(context.evalNodes("select|insert|update|delete"));
    } catch (Exception e) {
        throw new BuilderException("Error parsing Mapper XML. The XML location is '"
            + resource + "'. Cause: " + e, e);
    }
}
123456789101112131415161718

```

其中，基于 XMLMapperBuilder::buildStatementFromContext()，遍历 `<cache-ref>`、`<cache>` 节点们，逐个创建 XMLStatementBuilder 对象，执行解析，通过 XMLStatementBuilder::parseStatementNode() 解析，

- parameterType
- resultType
- selectKey 等

并会通过 LanguageDriver::createSqlSource() (默认 XmlLanguageDriver) 解析动态sql生成 SqlSource (详细内容请看下个小节)，

- 使用 GenericTokenParser::parser() 负责将 SQL 语句中的 `#{}` 替换成相应的 `?` 占位符，并获取该 `?` 占位符对应的

而且通过 `MapperBuilderAssistant::addMappedStatement()` 生成 `MappedStatement`

```
public void parseStatementNode() {
    //获得 id 属性, 编号
    String id = context.getStringAttribute("id");
    String databaseId = context.getStringAttribute("databaseId");
    // 判断 databaseId 是否匹配
    if (!databaseIdMatchesCurrent(id, databaseId, this.requiredDatabaseId)) {
        return;
    }
    //解析获得各种属性
    Integer fetchSize = context.getIntAttribute("fetchSize");
    Integer timeout = context.getIntAttribute("timeout");
    String parameterMap = context.getStringAttribute("parameterMap");
    String parameterType = context.getStringAttribute("parameterType");
    Class<?> parameterTypeClass = resolveClass(parameterType);
    String resultMap = context.getStringAttribute("resultMap");
    String resultType = context.getStringAttribute("resultType");
    String lang = context.getStringAttribute("lang");
    //获得 lang 对应的 LanguageDriver 对象
    LanguageDriver langDriver = getLanguageDriver(lang);
    //获得 resultType 对应的类
    Class<?> resultTypeClass = resolveClass(resultType);
    String resultSetType = context.getStringAttribute("resultSetType");
    //获得 resultSetType 对应的枚举值
    StatementType statementType =
        StatementType.valueOf(context.getStringAttribute("statementType",
            StatementType.PREPARED.toString()));
    //获得 resultSet 对应的枚举值
    ResultSetType resultSetTypeEnum = resolveResultSetType(resultSetType);

    String nodeName = context.getNode().getNodeName();
    //获得 SQL 对应的 SqlCommandType 枚举值
    SqlCommandType sqlCommandType =
        SqlCommandType.valueOf(nodeName.toUpperCase(Locale.ENGLISH));
    boolean isSelect = sqlCommandType == SqlCommandType.SELECT;
    //解析获得各种属性
    boolean flushCache = context.getBooleanAttribute("flushCache", !isSelect);
    boolean useCache = context.getBooleanAttribute("useCache", isSelect);
    boolean resultOrdered = context.getBooleanAttribute("resultOrdered", false);

    //创建 XMLIncludeTransformer 对象, 并替换 <include /> 标签相关的内容
    XMLIncludeTransformer includeParser = new XMLIncludeTransformer(configuration,
        builderAssistant);
    includeParser.applyIncludes(context.getNode());

    //解析 <selectKey /> 标签
    processSelectKeyNodes(id, parameterTypeClass, langDriver);

    //创建 SqlSource生成动态sql
    SqlSource sqlSource = langDriver.createSqlSource(configuration, context,
        parameterTypeClass);
    String resultSets = context.getStringAttribute("resultSets");
    String keyProperty = context.getStringAttribute("keyProperty");
    String keyColumn = context.getStringAttribute("keyColumn");
    KeyGenerator keyGenerator;
    String keyStatementId = id + SelectKeyGenerator.SELECT_KEY_SUFFIX;
```

```

keyStatementId = builderAssistant.applyCurrentNamespace(keyStatementId, true);
if (configuration.hasKeyGenerator(keyStatementId)) {
    keyGenerator = configuration.getKeyGenerator(keyStatementId);
} else {
    keyGenerator = context.getBooleanAttribute("useGeneratedKeys",
        configuration.isUseGeneratedKeys() &&
        SqlCommandType.INSERT.equals(sqlCommandType))
        ? Jdbc3KeyGenerator.INSTANCE : NoKeyGenerator.INSTANCE;
}
//创建 MappedStatement 对象
this.builderAssistant.addMappedStatement(id, sqlSource, statementType,
    sqlCommandType, fetchSize, timeout, parameterMap,
    parameterTypeClass, resultMap, resultTypeClass, resultSetTypeEnum, flushCache,
    useCache, resultOrdered, (KeyGenerator)keyGenerator,
    keyProperty, keyColumn, databaseId, langDriver, resultSets);
}
12345678910111213141516171819202122232425262728293031323334353637383940414243444
546474849505152535455565758596061626364

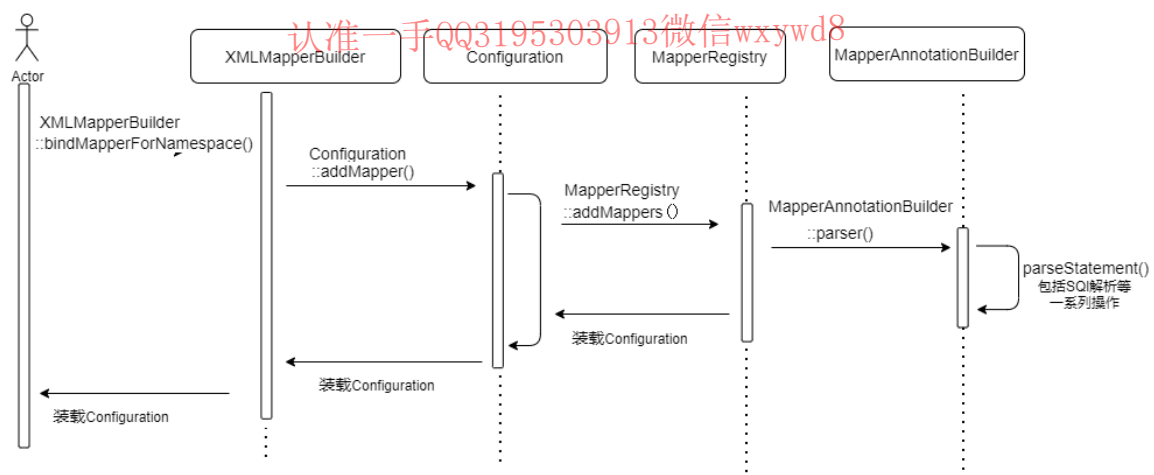
```

解析 Mapper 接口中的注解

当执行完 `XMLMapperBuilder::configurationElement()` 方法后,会调用

`XMLMapperBuilder::bindMapperForNamespace()` 会转换成对接口上注解进行扫描, 具体通过

`MapperRegistry::addMapper()` 调用 `MapperAnnotationBuilder` 实现的



`MapperAnnotationBuilder::parse()` 是注解构造器, 负责解析 `Mapper` 接口上的注解, 解析时需要注意避免和 `XMLMapperBuilder::parse()` 方法冲突, 重复解析, 最终使用 `parseStatement` 解析, 那怎么操作?

```

public void parse() {
    String resource = type.toString();
    //判断当前 Mapper 接口是否应加载过。
    if (!configuration.isResourceLoaded(resource)) {
        //加载对应的 XML Mapper, 注意避免和 `XMLMapperBuilder::parse()` 方法冲突
        loadXmlResource();
        //标记该 Mapper 接口已经加载过
        configuration.addLoadedResource(resource);
        assistant.setCurrentNamespace(type.getName());
        //解析 @CacheNamespace 注解
    }
}

```

```

parseCache();
parseCacheRef();
//遍历每个方法，解析其上的注解
Method[] methods = type.getMethods();
for (Method method : methods) {
    try {
        if (!method.isBridge()) {
            //执行解析
            parseStatement(method);
        }
    } catch (IncompleteElementException e) {
        configuration.addIncompleteMethod(new MethodResolver(this, method));
    }
}
}
//解析待定的方法
parsePendingMethods();
}
12345678910111213141516171819202122232425262728

```

那其中最重要的 `parseStatement()` 是怎么操作？其实跟解析 `Mapper.xml` 类型主要处理流程类似：

- 通过加载 `LanguageDriver`, `GenericTokenizer` 等为生成 `SqlSource` 动态sql作准备
- 使用 `MapperBuilderAssistant::addMappedStatement()` 生成注解
`@mapper`, `@CacheNamespace` 等的 `MappedStatement` 信息

```

void parseStatement(Method method) {
    //获取接口参数类型
    Class<?> parameterTypeClass = getParameterType(method);
    //加载语言处理器，默认XmlLanguageDriver
    LanguageDriver languageDriver = getLanguageDriver(method);
    //根据LanguageDriver, GenericTokenizer生成动态SQL
    SqlSource sqlSource = getSqlSourceFromAnnotations(method,
parameterTypeClass, languageDriver);
    if (sqlSource != null) {
        //获取其他属性
        Options options = method.getAnnotation(Options.class);
        final String mappedStatementId = type.getName() + "." +
method.getName();
        Integer fetchSize = null;
        Integer timeout = null;
        StatementType statementType = StatementType.PREPARED;
        ResultSetType resultSetType = null;
        SqlCommandType sqlCommandType = getSqlCommandType(method);
        boolean isSelect = sqlCommandType == SqlCommandType.SELECT;
        boolean flushCache = !isSelect;
        boolean useCache = isSelect;

        //获得 KeyGenerator 对象
        KeyGenerator keyGenerator;
        String keyProperty = null;
        String keyColumn = null;
        if (SqlCommandType.INSERT.equals(sqlCommandType) ||
SqlCommandType.UPDATE.equals(sqlCommandType)) { // 有
            // first check for SelectKey annotation - that overrides everything
        } else
            //如果有 @SelectKey 注解，则进行处理
    }
}

```

```

        SelectKey selectKey = method.getAnnotation(SelectKey.class);
        if (selectKey != null) {
            keyGenerator = handleSelectKeyAnnotation(selectKey,
mappedStatementId, getParameterType(method), languageDriver);
            keyProperty = selectKey.keyProperty();
            //如果无 @Options 注解，则根据全局配置处理
        } else if (options == null) {
            keyGenerator = configuration.isUseGeneratedKeys() ?
Jdbc3KeyGenerator.INSTANCE : NoKeyGenerator.INSTANCE;
            // 如果有 @Options 注解，则使用该注解的配置处理
        } else {
            keyGenerator = options.useGeneratedKeys() ?
Jdbc3KeyGenerator.INSTANCE : NoKeyGenerator.INSTANCE;
            keyProperty = options.keyProperty();
            keyColumn = options.keyColumn();
        }
        // 无
    } else {
        keyGenerator = NoKeyGenerator.INSTANCE;
    }

    //初始化各种属性
    if (options != null) {
        if (FlushCachePolicy.TRUE.equals(options.flushCache())) {
            flushCache = true;
        } else if (FlushCachePolicy.FALSE.equals(options.flushCache())) {
            flushCache = false;
        }
        useCache = options.useCache();
        fetchSize = options.fetchSize() > -1 || options.fetchSize() ==
Integer.MIN_VALUE ? options.fetchSize() : null; //issue #348
        timeout = options.timeout() > -1 ? options.timeout() : null;
        statementType = options.statementType();
        resultSetType = options.resultSetType();
    }

    // 获得 resultMapId 编号字符串
    String resultMapId = null;
    //如果有 @ResultMap 注解，使用该注解为 resultMapId 属性
    ResultMap resultMapAnnotation = method.getAnnotation(ResultMap.class);
    if (resultMapAnnotation != null) {
        String[] resultMaps = resultMapAnnotation.value();
        StringBuilder sb = new StringBuilder();
        for (String resultMap : resultMaps) {
            if (sb.length() > 0) {
                sb.append(",");
            }
            sb.append(resultMap);
        }
        resultMapId = sb.toString();
    }
    // 如果无 @ResultMap 注解，解析其它注解，作为 resultMapId 属性
    } else if (isSelect) {
        resultMapId = parseResultMap(method);
    }
    //构建 MappedStatement 对象
    assistant.addMappedStatement(
        mappedStatementId,
        sqlSource,

```



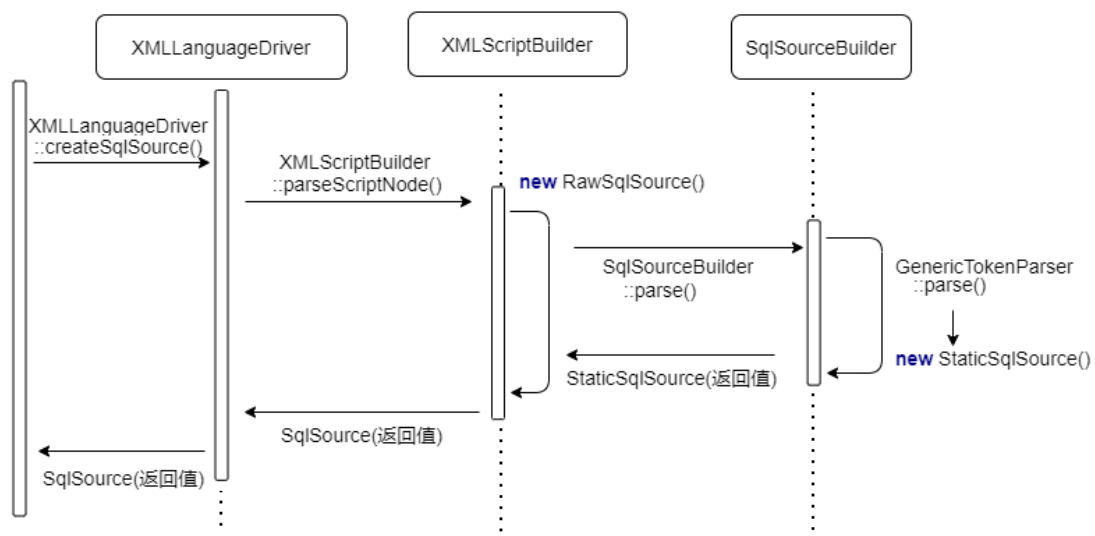
```

statementType,
sqlCommandType,
fetchSize,
timeout,
// ParameterMapID
null,
parameterTypeClass,
resultMapId,
getReturnType(method),
resultSetType,
flushCache,
useCache,
// TODO gcode issue #577
false,
keyGenerator,
keyProperty,
keyColumn,
// DatabaseID
null,
languageDriver,
// ResultSets
options != null ? nullOrEmpty(options.resultSets()) : null);
}
}
12345678910111213141516171819202122232425262728293031323334353637383940414243444
54647484950515253545556575859606162636465666768697071727374757677787980818283848
58687888990919293949596979899100101102103104105

```

生成动态 SqlSource 认准一手QQ3195303913微信wxywd8

当在执行 `langDriver::createSqlSource(configuration, context, parameterTypeClass)` 中的时候，是怎样从 `Mapper XML` 或方法注解上读取 `SQL` 内容生成动态 `SqlSource` 的呢？现在来一探究竟，



首先需要获取 `langDriver` 实现 `XMLLanguageDriver / RawLanguageDriver`，现在使用默认的 `XMLLanguageDriver::createSqlSource(configuration, context, parameterTypeClass)` 开启创建，再使用 `XMLScriptBuilder::parseScriptNode()` 解析生成 `SqlSource`

- `DynamicSqlSource`：动态的 `SqlSource` 实现类，适用于使用了 `OGNL` 表达式，或者使用了 `${}` 表达式的 `SQL`

- `RawSqlSource`：原始的 `SqlSource` 实现类，适用于仅使用 `#{} 表达式`，或者不使用任何表达式的情况

```
public SqlSource parseScriptNode() {
    MixedSqlNode rootSqlNode = this.parseDynamicTags(this.context);
    Object sqlSource;
    if (this.isDynamic) {
        sqlSource = new DynamicSqlSource(this.configuration, rootSqlNode);
    } else {
        sqlSource = new RawSqlSource(this.configuration, rootSqlNode,
this.parameterType);
    }

    return (SqlSource)sqlSource;
}
1234567891011
```

那就选择其中一种来分析一下 `RawSqlSource`，怎么完成构造的呢？看看 `RawSqlSource` 构造函数：

```
public RawSqlSource(Configuration configuration, String sql, Class<?>
parameterType) {
    SqlSourceBuilder sqlSourceParser = new SqlSourceBuilder(configuration);
    Class<?> clazz = parameterType == null ? Object.class : parameterType;
    this.sqlSource = sqlSourceParser.parse(sql, clazz, new HashMap());
}
12345
```

使用 `SqlSourceBuilder::parse()` 去解析SQL，里面又有什么神奇的地方呢？

```
public SqlSource parse(String originalSql, Class<?> parameterType, Map<String,
Object> additionalParameters) {
    SqlSourceBuilder.ParameterMappingTokenHandler handler = new
SqlSourceBuilder.ParameterMappingTokenHandler(this.configuration, parameterType,
additionalParameters);
    //创建基于#{的GenericTokenParser
    GenericTokenParser parser = new GenericTokenParser("#{", "}", handler);
    String sql = parser.parse(originalSql);
    return new StaticSqlSource(this.configuration, sql,
handler.getParameterMappings());
}
1234567
```

`ParameterMappingTokenHandler` 是 `SqlSourceBuilder` 的内部私有静态类，`ParameterMappingTokenHandler`，负责将匹配到的 `#{` 和 `}` 对，替换成相应的 `?` 占位符，并获取该 `?` 占位符对应的 `org.apache.ibatis.mapping.ParameterMapping` 对象。

并基于 `ParameterMappingTokenHandler` 使用 `GenericTokenParser::parse()` 将SQL中的 `#{` 转化占位符 `?` 占位符后创建一个 `StaticSqlSource` 返回。

总结

在 MyBatis 初始化过程中，会加载 `mybatis-config.xml` 配置文件、`Mapper.xml` 映射配置文件以及 `Mapper` 接口中的注解信息，解析后的配置信息会形成相应的对象并全部保存到 `Configuration` 对象中，并创建 `DefaultSqlSessionFactory` 供SQL执行过程创建出顶层接口 `SqlSession` 供给用户进行操作。

认准一手QQ3195303913微信wxywd8