
扫一扫二维码关注公众号
关注后，可以看到
「RocketMQ」
「MyCAT」
所有源码解析文章
近期更新「Sharding-JDBC」中
你有233个小伙伴已经关注

微信公众号福利：芋艿的后端小屋

0. 阅读源码葵花宝典

1. RocketMQ / MyCAT / Sharding-JDBC 详细中文注释源码

2. 您对于源码的疑问每条留言都将得到认真回复

3. 新的源码解析文章实时收到通知，每周六十点更新

4. 认真的源码交流微信群

分类

分享

Sharding-JDBC 源码分析 —— SQL 解析（一）之词法解析

2017-07-23 更新日期:2017-07-31 总阅读量:172次

文章目录

---

1. 概述

SQL 解析引擎，数据库中间件必备的功能和流程。Sharding-JDBC 在 `1.5.0.M1` 正式发布时，将 SQL 解析引擎从 Druid 替换成了自研的。**新引擎仅解析分片上下文，对于 SQL 采用”半理解”理念，进一步提升性能和兼容性，同时降低了代码复杂度**（不理解没关系，我们后续会更新文章解释该优点）。 国内另一款数据库中间件 MyCAT SQL 解析引擎也是 Druid，目前也在开发属于自己的 SQL 解析引擎。

可能有同学看到**SQL 解析**会被吓到，请淡定，耐心往下看。《SQL 解析》内容我们会分成 5 篇相对简短的文章，让大家能够相对轻松愉快的去理解：

1. 词法解析
2. 插入 SQL 解析
3. 查询 SQL 解析
4. 更新 SQL 解析
5. 删除 SQL 解析

maven模块	模块	子模块	子子模块	用途	模块代码量	子模块代码量	
core	api				1750		
		rule		分表规则接口			
		strategry		分表策略接口			
	config			配置	178		
	constant			枚举	136		
	exception			异常	37		
	executor			执行器	1070		
	hint			分片键			
	jdbc				3955		
		adapter		适配？			
		core		jdbc 核心			
		unsupported		未实现			
	keygen			id生成	248		
	merger			结果合并	1656		
	metrics			指标			
	parsing			SQL 解析	6262		
		lexer		语法		1404	
			analyzer	分析器			483
			dialect	方言			437
			token	语法标记			348
		parser				4692	

		context			762
		dialect			1519
		exception			69
		expression			200
		statement			1392
		token			
rewrite			SQL 重写	298	
routing				1499	
	router		路由执行器		
	stragegy		路由策略		
	type		路由类型		
util			工具	234	

SQL 解析引擎在 `parsing` 包下，如上图所见包含两大组件：

1. Lexer：**词法**解析器。
2. Parser：**SQL**解析器。

两者都是解析器，区别在于 Lexer 只做词法的解析，不关注上下文，将字符串拆解成 N 个词法。而 Parser 在 Lexer 的基础上，还需要理解 SQL 。打个比方：

```
SQL : SELECT * FROM t_user
Lexer : [SELECT] [ * ] [FROM] [t_user]
Parser : 这是一条 [SELECT] 查询表为 [t_user] ，并且返回 [ * ] 所有字段的 SQL。
```

□不完全懂？没关系，本文的主角是 Lexer，我们通过源码一点一点理解。一共 1400 行左右代码左右，还包含注释等等，实际更少噢。

分享

2. Lexer 词法解析器

Lexer 原理：**顺序顺序顺序** 解析 SQL，将字符串拆解成 N 个词法。

核心代码如下：

```
// Lexer.java
public class Lexer {
    /**
     * 输出字符串
     * 比如：SQL
     */
    @Getter
    private final String input;
    /**
     * 词法标记字典
     */
    private final Dictionary dictionary;
    /**
     * 解析到 SQL 的 offset
     */
    private int offset;
    /**
     * 当前 词法标记
     */
    @Getter
    private Token currentToken;
    /**
     * 分析下一个词法标记.
```

```
     *
     * @see #currentToken
     * @see #offset
     */
    public final void nextToken() {
        skipIgnoredToken();
        if (isVariableBegin()) { // 变量
            currentToken = new Tokenizer(input, dictionary, offset).scanVariable();
        } else if (isNCharBegin()) { // N\
            currentToken = new Tokenizer(input, dictionary, ++offset).scanChars();
        } else if (isIdentifierBegin()) { // Keyword + Literals.IDENTIFIER
            currentToken = new Tokenizer(input, dictionary, offset).scanIdentifier();
        } else if (isHexDecimalBegin()) { // 十六进制
            currentToken = new Tokenizer(input, dictionary, offset).scanHexDecimal();
        } else if (isNumberBegin()) { // 数字（整数+浮点数）
            currentToken = new Tokenizer(input, dictionary, offset).scanNumber();
        } else if (isSymbolBegin()) { // 符号
            currentToken = new Tokenizer(input, dictionary, offset).scanSymbol();
        } else if (isCharsBegin()) { // 字符串，例如："abc"
            currentToken = new Tokenizer(input, dictionary, offset).scanChars();
        } else if (isEnd()) { // 结束
            currentToken = new Token(Assist.END, "", offset);
        } else { // 分析错误，无符合条件的词法标记
            currentToken = new Token(Assist.ERROR, "", offset);
        }
        offset = currentToken.getEndPosition();
        // System.out.println("| " + currentToken.getLiterals() + " | " + currentToken.getType() + " |
    }
    /**
```

```
     * 跳过忽略的词法标记
     * 1. 空格
     * 2. SQL Hint
     * 3. SQL 注释
     */
    private void skipIgnoredToken() {
        // 空格
        offset = new Tokenizer(input, dictionary, offset).skipWhitespace();
        // SQL Hint
        while (isHintBegin()) {
            offset = new Tokenizer(input, dictionary, offset).skipHint();
            offset = new Tokenizer(input, dictionary, offset).skipWhitespace();
        }
        // SQL 注释
        while (isCommentBegin()) {
            offset = new Tokenizer(input, dictionary, offset).skipComment();
            offset = new Tokenizer(input, dictionary, offset).skipWhitespace();
        }
    }
}
```

通过 `#nextToken()` 方法，不断解析出 Token(*词法标记*)。我们来执行一次，看看 SQL 会被拆解成哪些 Token。

```
SQL : SELECT i.* FROM t_order o JOIN t_order_item i ON o.order_id=i.order_id WHERE o.user_id=? AND o.o
```

literals	TokenType类	TokenType值	endPosition

SELECT	DefaultKeyword	SELECT	6
i	Literals	IDENTIFIER	8
.	Symbol	DOT	9
*	Symbol	STAR	10
FROM	DefaultKeyword	FROM	15
t_order	Literals	IDENTIFIER	23
o	Literals	IDENTIFIER	25
JOIN	DefaultKeyword	JOIN	30
t_order_item	Literals	IDENTIFIER	43
i	Literals	IDENTIFIER	45
ON	DefaultKeyword	ON	48
o	Literals	IDENTIFIER	50
.	Symbol	DOT	51
order_id	Literals	IDENTIFIER	59

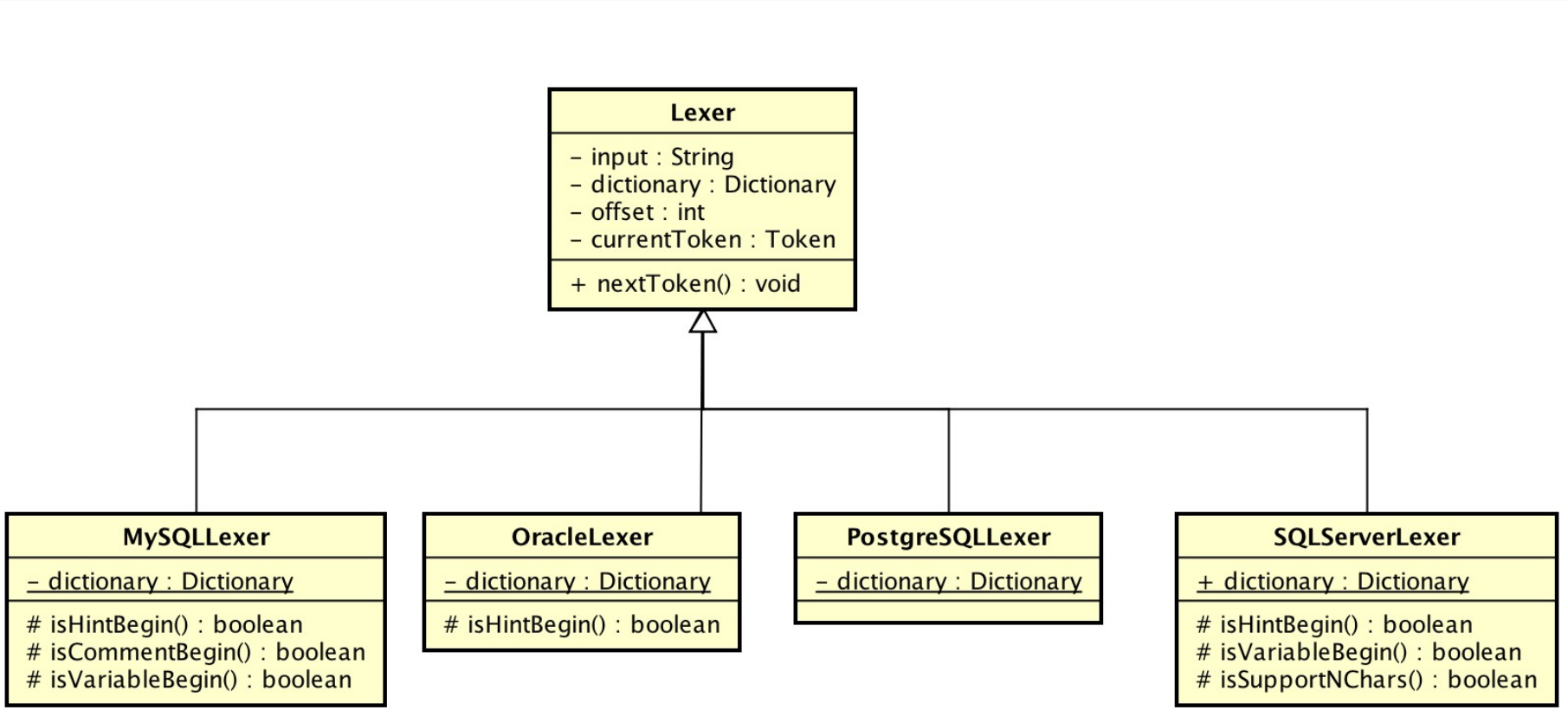
=	Symbol	EQ	60
i	Literals	IDENTIFIER	61
.	Symbol	DOT	62
order_id	Literals	IDENTIFIER	70
WHERE	DefaultKeyword	WHERE	76
o	Literals	IDENTIFIER	78
.	Symbol	DOT	79
user_id	Literals	IDENTIFIER	86
=	Symbol	EQ	87
?	Symbol	QUESTION	88
AND	DefaultKeyword	AND	92
o	Literals	IDENTIFIER	94
.	Symbol	DOT	95
order_id	Literals	IDENTIFIER	103

=	Symbol	EQ	104
?	Symbol	QUESTION	105
	Assist	END	105

眼尖的同学可能看到了 Tokenizer。对的，它是 Lexer 的好基佬，负责**分词**。

*我们来总结下，`Lexer#nextToken()` 方法里，使用 `#skipIgnoredToken()` 方法跳过忽略的 Token，通过 `#isXXXX()` 方法判断好下一个 Token 的类型后，**交给 Tokenizer 进行分词返回 Token**。!!此处可以考虑做个优化，不需要每次都 `new Tokenizer(...)` 出来，一个 Lexer 搭配一个 Tokenizer。*

由于不同数据库遵守 SQL 规范略有不同，所以不同的数据库对应不同的 Lexer。



子 Lexer 通过重写方法实现自己独有的 SQL 语法。

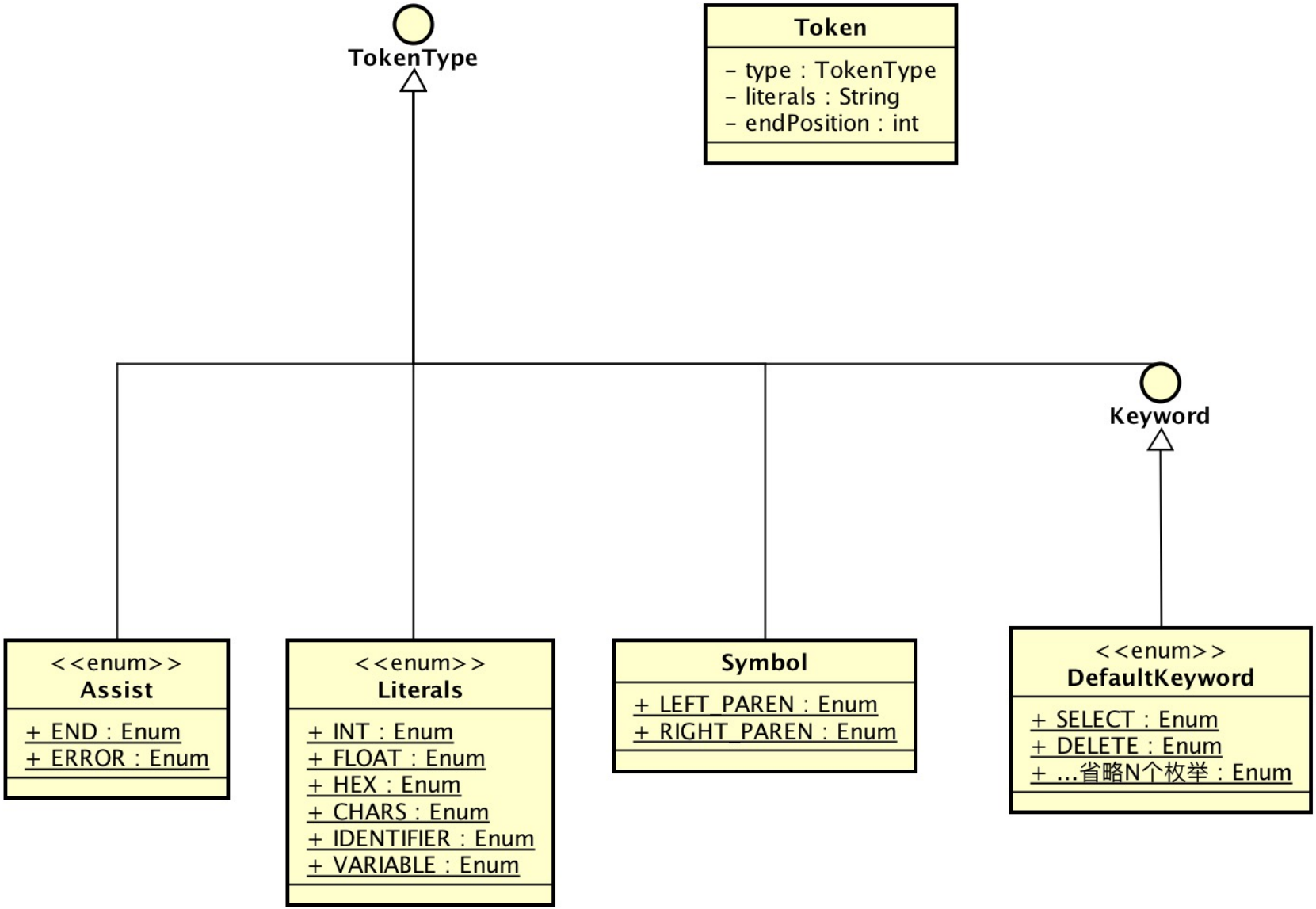
3. Token 词法标记

上文我们已经看过 Token 的例子，一共有 3 个属性：

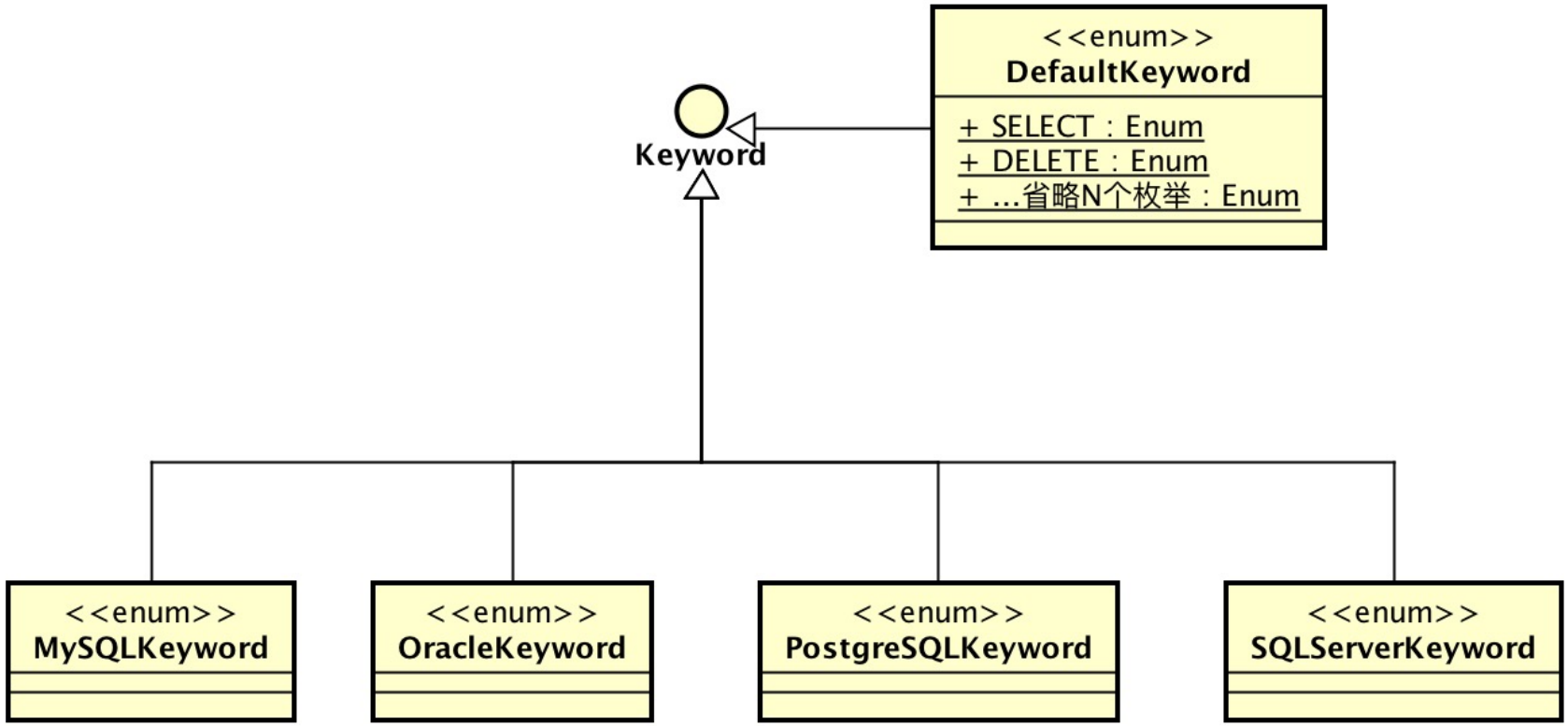
- TokenType type ：词法标记类型
- String literals ：词法字面量标记
- int endPosition ： `literals` 在 SQL 里的结束位置

TokenType 词法标记类型，一共分成 4 个大类：

- DefaultKeyword ：词法关键词
- Literals ：词法字面量标记
- Symbol ：词法符号标记
- Assist ：词法辅助标记



3.1 DefaultKeyword 词法关键词



不同数据库有自己独有的*词法关键词*，例如 MySQL 熟知的分页 Limit。

我们以 MySQL 举个例子，当创建 MySQLLexer 时，会加载 DefaultKeyword 和 MySQLKeyword（ *OracleLexer*、*PostgreSQLLexer*、*SQLServerLexer* 同 *MySQLLexer* ）。核心代码如下：

```
// MySQLLexer.java
public final class MySQLLexer extends Lexer {
    /**
     * 字典
     */
    private static Dictionary dictionary = new Dictionary(MySQLKeyword.values());
```

```
    public MySQLLexer(final String input) {
        super(input, dictionary);
    }
}
// Dictionary.java
public final class Dictionary {
    /**
     * 词法关键词Map
     */
    private final Map<String, Keyword> tokens = new HashMap<>(1024);

    public Dictionary(final Keyword... dialectKeywords) {
        fill(dialectKeywords);
    }
    /**
     * 装上默认词法关键词 + 方言词法关键词
     * 不同的数据库有相同的默认词法关键词，有有不同的方言关键词
     *
     * @param dialectKeywords 方言词法关键词
     */
    private void fill(final Keyword... dialectKeywords) {
        for (DefaultKeyword each : DefaultKeyword.values()) {
            tokens.put(each.name(), each);
        }
        for (Keyword each : dialectKeywords) {
            tokens.put(each.toString(), each);
        }
    }
```

```
}
```

Keyword 与 Literals.IDENTIFIER 是一起解析的，我们放在 Literals.IDENTIFIER 处一起分析。

3.2 Literals 词法字面量标记

Literals 词法字面量标记，一共分成 6 种：

- IDENTIFIER ：词法关键词
- VARIABLE ：变量
- CHARS ：字符串
- HEX ：十六进制
- INT ：整数
- FLOAT ：浮点数

3.2.1 Literals.IDENTIFIER 词法关键词

词法关键词。例如：表名，查询字段 等等。

解析 Literals.IDENTIFIER 与 Keyword 核心代码如下：

```
// Lexer.java
private boolean isIdentifierBegin() {
   return isIdentifierBegin(getCurrentChar(0));
}
private boolean isIdentifierBegin(final char ch) {
```

```
    return CharType.isAlphabet(ch) || '`' == ch || '_' == ch || '$' == ch;
}
// Tokenizer.java
/**
* 扫描标识符.
*
* @return 标识符标记
*/
public Token scanIdentifier() {
    // `字段`，例如：SELECT `id` FROM t_user 中的 `id`
    if ('`' == charAt(offset)) {
        int length = getLengthUntilTerminatedChar('`');
        return new Token(Literals.IDENTIFIER, input.substring(offset, offset + length), offset + length
    }
    int length = 0;
    while (isIdentifierChar(charAt(offset + length))) {
        length++;
    }
    String literals = input.substring(offset, offset + length);
    // 处理 order / group 作为表名
    if (isAmbiguousIdentifier(literals)) {
        return new Token(processAmbiguousIdentifier(offset + length, literals), literals, offset + leng
    }
    // 从 词法关键词 查找是否是 Keyword，如果是，则返回 Keyword，否则返回 Literals.IDENTIFIER
    return new Token(dictionary.findTokenType(literals, Literals.IDENTIFIER), literals, offset + length
}
/**
* 计算到结束字符的长度
*
```

```
     * @see #hasEscapeChar(char, int) 处理类似 SELECT a AS `b``c` FROM table。此处连续的 "``" 不是结尾，如果传
     * @param terminatedChar 结束字符
     * @return 长度
     */
    private int getLengthUntilTerminatedChar(final char terminatedChar) {
        int length = 1;
        while (terminatedChar != charAt(offset + length) || hasEscapeChar(terminatedChar, offset + length))
            if (offset + length >= input.length()) {
                throw new UnterminatedCharException(terminatedChar);
            }
            if (hasEscapeChar(terminatedChar, offset + length)) {
                length++;
            }
            length++;
        }
        return length + 1;
    }
    /**
     * 是否是 Escape 字符
     *
     * @param charIdentifier 字符
     * @param offset 位置
     * @return 是否
     */
    private boolean hasEscapeChar(final char charIdentifier, final int offset) {
        return charIdentifier == charAt(offset) && charIdentifier == charAt(offset + 1);
    }
    private boolean isIdentifierChar(final char ch) {
        return CharType.isAlphabet(ch) || CharType.isDigital(ch) || '_' == ch || '$' == ch || '#' == ch;
```

```
    }
    /**
     * 是否是引起歧义的标识符
     * 例如 "SELECT * FROM group"，此时 "group" 代表的是表名，而非词法关键词
     *
     * @param literals 标识符
     * @return 是否
     */
    private boolean isAmbiguousIdentifier(final String literals) {
        return DefaultKeyword.ORDER.name().equalsIgnoreCase(literals) || DefaultKeyword.GROUP.name().equals
    }
    /**
     * 获取引起歧义的标识符对应的词法标记类型
     *
     * @param offset 位置
     * @param literals 标识符
     * @return 词法标记类型
     */
    private TokenType processAmbiguousIdentifier(final int offset, final String literals) {
        int i = 0;
        while (CharType.isWhitespace(charAt(offset + i))) {
            i++;
        }
        if (DefaultKeyword.BY.name().equalsIgnoreCase(String.valueOf(new char[] {charAt(offset + i), charAt
            return dictionary.findTokenType(literals);
        }
        return Literals.IDENTIFIER;
    }
```

3.2.2 Literals.VARIABLE 变量

变量。例如：`SELECT @@VERSION` 。

解析核心代码如下：

```
// Lexer.java
/**
* 是否是 变量
* MySQL 与 SQL Server 支持
*
* @see Tokenizer#scanVariable()
* @return 是否
*/
protected boolean isVariableBegin() {
   return false;
}
// Tokenizer.java
/**
* 扫描变量.
* 在 MySQL 里，@代表用户变量；@@代表系统变量。
* 在 SQLServer 里，有 @@。
*
* @return 变量标记
*/
public Token scanVariable() {
   int length = 1;
   if ('@' == charAt(offset + 1)) {
       length++;
```

```
    }
    while (isVariableChar(charAt(offset + length))) {
        length++;
    }
    return new Token(Literals.VARIABLE, input.substring(offset, offset + length), offset + length);
}
```

3.2.3 Literals.CHARS 字符串

字符串。例如：`SELECT "123"` 。

解析核心代码如下：

```
// Lexer.java
/**
* 是否 N\
* 目前 SQLServer 独有：在 SQL Server 中處理 Unicode 字串常數時，必需為所有的 Unicode 字串加上前置詞 N
*
* @see Tokenizer#scanChars()
* @return 是否
*/
private boolean isNCharBegin() {
    return isSupportNChars() && 'N' == getCurrentChar(0) && '\'' == getCurrentChar(1);
}
private boolean isCharsBegin() {
    return '\'' == getCurrentChar(0) || '\"' == getCurrentChar(0);
}
// Tokenizer.java
/**
```

```
* 扫描字符串.
*
* @return 字符串标记
*/
public Token scanChars() {
    return scanChars(charAt(offset));
}
private Token scanChars(final char terminatedChar) {
    int length = getLengthUntilTerminatedChar(terminatedChar);
    return new Token(Literals.CHARS, input.substring(offset + 1, offset + length - 1), offset + length)
}
```

3.2.4 Literals.HEX 十六进制

```
// Lexer.java
/**
* 是否是 十六进制
*
* @see Tokenizer#scanHexDecimal()
* @return 是否
*/
private boolean isHexDecimalBegin() {
    return '0' == getCurrentChar(0) && 'x' == getCurrentChar(1);
}
// Tokenizer.java
/**
* 扫描十六进制数.
```

```
     *
     * @return 十六进制数标记
     */
    public Token scanHexDecimal() {
        int length = HEX_BEGIN_SYMBOL_LENGTH;
        // 负数
        if ('-' == charAt(offset + length)) {
            length++;
        }
        while (isHex(charAt(offset + length))) {
            length++;
        }
        return new Token(Literals.HEX, input.substring(offset, offset + length), offset + length);
    }
```

3.2.5 Literals.INT 整数

整数。例如：`SELECT * FROM t_user WHERE id = 1`。

Literals.INT 与 Literals.FLOAT 是一起解析的，我们放在 Literals.FLOAT 处一起分析。

3.2.6 Literals.FLOAT 浮点数

浮点数。例如：`SELECT * FROM t_user WHERE id = 1.0`。
浮点数包含几种：”1.0”，”1.0F”，”7.823E5”（科学计数法）。

解析核心代码如下：

```
// Lexer.java
/**
* 是否是 数字
* '-' 需要特殊处理。".2" 被处理成省略0的小数，"-.2" 不能被处理成省略的小数，否则会出问题。
* 例如说，"SELECT a-.2" 处理的结果是 "SELECT" / "a" / "-" / ".2"
*
* @return 是否
*/
private boolean isNumberBegin() {
    return CharType.isDigital(getCurrentChar(0)) // 数字
            || ('.' == getCurrentChar(0) && CharType.isDigital(getCurrentChar(1)) && !isIdentifierBegin
            || ('-' == getCurrentChar(0) && ('.' == getCurrentChar(0) || CharType.isDigital(getCurrentC
}
// Tokenizer.java
/**
* 扫描数字.
* 解析数字的结果会有两种：整数 和 浮点数.
*
* @return 数字标记
*/
public Token scanNumber() {
    int length = 0;
    // 负数
    if ('-' == charAt(offset + length)) {
        length++;
    }
    // 浮点数
    length += getDigitalLength(offset + length);
    boolean isFloat = false;
```

```
    if ('.' == charAt(offset + length)) {
        isFloat = true;
        length++;
        length += getDigitalLength(offset + length);
    }
    // 科学计数表示，例如：SELECT 7.823E5
    if (isScientificNotation(offset + length)) {
        isFloat = true;
        length++;
        if ('+' == charAt(offset + length) || '-' == charAt(offset + length)) {
            length++;
        }
        length += getDigitalLength(offset + length);
    }
    // 浮点数，例如：SELECT 1.333F
    if (isBinaryNumber(offset + length)) {
        isFloat = true;
        length++;
    }
    return new Token(isFloat ? Literals.FLOAT : Literals.INT, input.substring(offset, offset + length),
}
```

这里要特别注意下：“-“。在数字表达实例，可以判定为 负号 和 减号（不考虑科学计数法）。

- “.2” 解析结果是 “.2”
- “-.2” 解析结果不能是 “-.2”，而是 “-“ 和 “.2”。

3.3 Symbol 词法符号标记

词法符号标记。例如：”{“, “}”, “>=” 等等。

解析核心代码如下：

```
// Lexer.java
/**
* 是否是 符号
*
* @see Tokenizer#scanSymbol()
* @return 是否
*/
private boolean isSymbolBegin() {
   return CharType.isSymbol(getCurrentChar(0));
}
// CharType.java
/**
* 判断是否为符号.
*
* @param ch 待判断的字符
* @return 是否为符号
*/
public static boolean isSymbol(final char ch) {
   return '(' == ch || ')' == ch || '[' == ch || ']' == ch || '{' == ch || '}' == ch || '+' == ch || '
           || '>' == ch || '<' == ch || '~' == ch || '!' == ch || '?' == ch || '&' == ch || '|' == ch
}
// Tokenizer.java
/**
* 扫描符号.
```

```
     *
     * @return 符号标记
     */
    public Token scanSymbol() {
        int length = 0;
        while (CharType.isSymbol(charAt(offset + length))) {
            length++;
        }
        String literals = input.substring(offset, offset + length);
        // 倒序遍历，查询符合条件的 符号。例如 literals = ";;"，会是拆分成两个 ";"。如果基于正序，literals = "<="
        Symbol symbol;
        while (null == (symbol = Symbol.literalsOf(literals))) {
            literals = input.substring(offset, offset + --length);
        }
        return new Token(symbol, literals, offset + length);
    }
```

3.4 Assist 词法辅助标记

Assist 词法辅助标记，一共分成 2 种：

- END ：分析结束
- ERROR ：分析错误。

4. 彩蛋

老铁，是不是比想象中简单一些？！继续加油写 Parser 相关的文章！来一波微信公众号关注吧。

Sharding-JDBC 正在收集使用公司名单：传送门。 你的登记，会让更多人参与和使用 Sharding-JDBC。Sharding-JDBC 也会因此，能够覆盖更广的场景。登记吧，少年！

我创建了一个微信群【源码圈】，希望和大家分享交流读源码的经验。
读源码先难后易，掌握方法后，可以做更有深度的学习。
而且掌握方法并不难噢。
加群方式：微信公众号发送关键字【qun】。

Sharding-JDBC

PREVIOUS:
« Sharding-JDBC 源码分析 —— SQL 解析（二）之SQL解析
NEXT:
» MyCAT 源码分析 —— SQL ON MongoDB