第 3 章

词法单元

3.1 编译过程

典型的编译过程一般分为 5 个阶段，分别是词法分析、语法分析、语义分析与中间代码生成、代码优化及最终的目标代码生成，详细过程如图 3-1 所示。

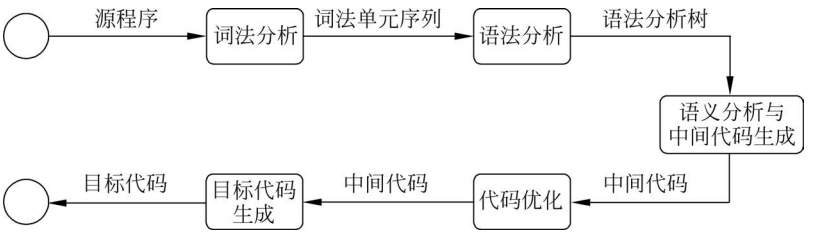

图 3-1 编译过程

在这个过程中，词法分析器对构成源程序的字符串进行扫描和解析，根据词法规则，识别出一个个的单词符号，如关键字、标识符、字面量、操作符等，这些单词符号构成了词法单元序列，在此基础上，再进行语法分析及代码生成等编译步骤，所以，词法单元是后续操作的基础对象，了解词法单元的用法是进行元编程的基础。

3.2 Token

在仓颉语言里，词法单元被称为“令牌”，使用 Token 类型表示，由仓颉标准库 ast 包提供。

3.2.1　成员变量

Token 类型包含如下 3 个变量：

```
public let kind: TokenKind
public let value: String
public let pos: Position
```

- kind

kind 表示令牌的类型，它的变量类型是 TokenKind。TokenKind 是一个包括所有令牌类型的枚举类型，例如，表示加法的枚举为 TokenKind.ADD，3.3 节会详细介绍该类型。

- value

value 表示构成令牌的字符串，是令牌的字面量。

- pos

pos 表示令牌在源代码文件中的位置信息，类型为 Position，包括 fileID（源代码文件 ID）、line（行号）和 column（列号）3 个成员变量。

3.2.2　构造函数

Token 的构造函数如下所示。

- public init()

使用默认构造函数创建 Token。

- public init(k：TokenKind)

使用 TokenKind 类型的参数 k 创建一个新的 Token。

- public init(k：TokenKind，v：String)

使用 TokenKind 类型的参数 k、String 类型的参数 v，创建一个新的 Token。

下面是一个创建 Token 并输出字面量的示例：

```
//Chapter3/token_sample/src/token_sample.cj

from std import ast.*

main() {
    //创建表示加号的令牌
    let addToken = Token(TokenKind.ADD)
```

```
    //创建名称为 cangjie 的标识符
    let idToken = Token(TokenKind.IDENTIFIER, "cangjie")

    println(addToken.value)
    println(idToken.value)
}
```

编译后运行该示例,输出如下:

```
cjc token_sample.cj
main.exe
+
cangjie
```

在这个示例中,创建了两个 Token,第 1 个 Token 使用 TokenKind.ADD 作为参数,所以打印输出的字面量是加号;第 2 个 Token 表示一个叫 cangjie 的标识符,所以打印出字符串 cangjie。

3.2.3 常用函数

Token 的常用函数是 dump,定义如下:

```
public func dump(): Unit
```

dump 函数用来输出 Token 的详细信息,这里把 3.2.2 节的示例改进一下,使用 dump 代替原来的 println 方法,示例如下:

```
//Chapter3/token_dump/src/token_dump.cj

from std import ast.*

main() {
    let addToken = Token(TokenKind.ADD)
    let idToken = Token(TokenKind.IDENTIFIER, "cangjie")

    addToken.dump()
    idToken.dump()
}
```

编译后运行该示例,输出如下:

```
cjc token_dump.cj
main.exe
```

```
description: add, token_id: 12, token_literal_value: + , fileID: 1, line: 4, column: 20
description: identifier, token_id: 133, token_literal_value: cangjie, fileID: 1, line: 5,
column: 19
```

dump 函数输出信息的说明如下。

(1) description：对 Token 类型的说明。

(2) token_id：Token 的唯一标识。

(3) token_literal_value：Token 的字面量表示形式。

(4) fileID：Token 所在源文件的文件标识，本例只有一个文件，所以 ID 为 1。

(5) line：Token 所在源文件的行号，本例分别在第 4 行和第 5 行。

(6) column：Token 所在源文件的列号，针对本例是每个 Token 构造函数开始的位置。

3.3　TokenKind

TokenKind 是 Token 类型的枚举，主要包括关键字、标识符、字面量、操作符等类型，详细的枚举描述信息如表 3-1 所示。

表 3-1　TokenKind 枚举

分　类	枚　举	字　面　量	说　明
操作符	ADD	+	—
操作符	ADD_ASSIGN	+=	—
操作符	AND	&&	—
操作符	AND_ASSIGN	&&=	—
操作符	ARROW	->	—
操作符	ASSIGN	=	—
操作符	AT	@	—
操作符	BITAND	&	—
操作符	BITAND_ASSIGN	&=	—
操作符	BITNOT	~	—
操作符	BITOR	|	—
操作符	BITOR_ASSIGN	|=	—
操作符	BITXOR	^	—
操作符	BITXOR_ASSIGN	^=	—
操作符	CLOSEDRANGEOP	..=	—

续表

分　类	枚　举	字 面 量	说　明
操作符	COALESCING	??	—
操作符	COMPOSITION	~>	—
操作符	DECR	--	—
操作符	DIV	/	—
操作符	DIV_ASSIGN	/=	—
操作符	DOT	.	—
操作符	DOUBLE_ARROW	=>	—
操作符	ELLIPSIS	...	—
操作符	EQUAL	==	—
操作符	EXP	**	—
操作符	EXP_ASSIGN	**=	—
操作符	GE	>=	—
操作符	GT	>	—
操作符	INCR	++	—
操作符	LE	<=	—
操作符	LSHIFT	<<	—
操作符	LSHIFT_ASSIGN	<<=	—
操作符	LT	<	—
操作符	MOD	%	—
操作符	MOD_ASSIGN	%=	—
操作符	MUL	*	—
操作符	MUL_ASSIGN	*=	—
操作符	NOT	!	—
操作符	NOTEQ	!=	—
操作符	OR	||	—
操作符	OR_ASSIGN	||=	—
操作符	PIPELINE	|>	—
操作符	QUEST	?	—
操作符	RANGEOP	..	—
操作符	RSHIFT	>>	—
操作符	RSHIFT_ASSIGN	>>=	—
操作符	SUB	-	—
操作符	SUB_ASSIGN	-=	—
分隔符	COLON	:	—
分隔符	COMMA	,	—
分隔符	LCURL	{	—
分隔符	LPAREN	(	—
分隔符	LSQUARE	[	—
分隔符	RCURL	}	—

续表

分　类	枚　举	字 面 量	说　明
分隔符	RPAREN	)	—
分隔符	RSQUARE	]	—
分隔符	SEMI	;	—
关键字	ABSTRACT	abstract	—
关键字	AS	as	也是操作符
关键字	BOOLEAN	Bool	—
关键字	BREAK	break	—
关键字	CASE	case	—
关键字	CATCH	catch	—
关键字	CFUNC	CFunc	—
关键字	CHAR	Char	—
关键字	CLASS	class	—
关键字	CONTINUE	continue	—
关键字	DO	do	—
关键字	ELSE	else	—
关键字	ENUM	enum	—
关键字	EXTEND	extend	—
关键字	FINALLY	finally	—
关键字	FLOAT16	Float16	—
关键字	FLOAT32	Float32	—
关键字	FLOAT64	Float64	—
关键字	FOR	for	—
关键字	FOREIGN	foreign	—
关键字	FROM	from	—
关键字	FUNC	func	—
关键字	IF	if	—
关键字	IMPORT	import	—
关键字	IN	in	—
关键字	INIT	init	—
关键字	INT16	Int16	—
关键字	INT32	Int32	—
关键字	INT64	Int64	—
关键字	INT8	Int8	—
关键字	INTERFACE	interface	—
关键字	INTNATIVE	IntNative	—
关键字	IS	is	也是操作符
关键字	LET	let	—
关键字	MACRO	macro	—
关键字	MAIN	main	—

续表

分　类	枚　举	字 面 量	说　明
关键字	MATCH	match	—
关键字	MUT	mut	—
关键字	NOT_IN	!in	—
关键字	NOTHING	Nothing	—
关键字	OPEN	open	—
关键字	OPERATOR	operator	—
关键字	OVERRIDE	override	—
关键字	PACKAGE	package	—
关键字	PRIVATE	private	—
关键字	PROP	prop	—
关键字	PROTECTED	protected	—
关键字	PUBLIC	public	—
关键字	QUOTE	quote	—
关键字	REDEF	redef	—
关键字	RETURN	return	—
关键字	SPAWN	spawn	—
关键字	STATIC	static	—
关键字	STRUCT	struct	—
关键字	SUPER	super	—
关键字	SYNCHRONIZED	synchronized	—
关键字	THIS	this	—
关键字	THISTYPE	This	—
关键字	THROW	throw	—
关键字	TRY	try	—
关键字	TYPE	type	—
关键字	UINT16	UInt16	—
关键字	UINT32	UInt32	—
关键字	UINT64	UInt64	—
关键字	UINT8	UInt8	—
关键字	UINTNATIVE	UIntNative	—
关键字	UNIT	Unit	—
关键字	UNSAFE	unsafe	—
关键字	VAR	var	—
关键字	WHERE	where	—
关键字	WHILE	while	—
关键字	WITH	with	—
其他	ANNOTATION	无	声明，例如@when 等宏标识
其他	COMMENT	无	注释
其他	DOLLAR	$	插值字符串标识

续表

分　类	枚　举	字　面　量	说　明
其他	END	无	文件结束
其他	HASH	#	—
其他	IDENTIFIER	无	标识符
其他	ILLEGAL	无	不合法 token
其他	NL	无	新行
其他	SENTINEL	无	—
其他	UPPERBOUND	<:	继承
其他	WILDCARD	_	通配符模式
字面量	BOOL_LITERAL	无	布尔类型字面量，例如 true 或者 false
字面量	BYTE _ STRING _ ARRAY _ LITERAL	无	字节数组字面量，例如 b"abc"
字面量	CHAR_BYTE_LITERAL	无	字符字节字面量，例如 b'a'
字面量	CHAR_LITERAL	无	字符字面量，例如 a
字面量	DOLLAR_IDENTIFIER	无	元编程中的插值标识符
字面量	FLOAT_LITERAL	无	浮点字面量，例如 1.0
字面量	INTEGER_LITERAL	无	整型字面量，例如 1
字面量	MULTILINE_RAW_STRING	无	多行原始字符串字面量，例如 ##"abc"##
字面量	MULTILINE_STRING	无	多行字符串字面量，例如 """abc"""
字面量	STRING_LITERAL	无	字符串字面量，例如 abc
字面量	UNIT_LITERAL	无	Unit 字面量，例如()

说明：仓颉语言在不断发展和进化中，TokenKind 所包含的具体枚举值也会随之变化，本节表格列出的枚举值基于 0.37.2 版本提供，后续版本的枚举值可能会有微调。

3.4　Tokens

一段代码经过词法分析后可能会生成多个 Token 类型的对象，这些 Token 对象的序列被封装为一个 Tokens 类型的对象，Tokens 类型是仓颉元编程中主要的输入/输出类型。

3.4.1 构造函数

Tokens 的构造函数如下所示。

- public init()

使用默认构造函数创建 Tokens。

- public init(tokArr: Array < Token >)

使用 Token 数组类型的参数 tokArr 创建一个新的 Tokens。

- public init(buf: Array < UInt8 >)

使用 UInt8 类型的数组创建一个新的 Tokens。

- public init(tokArrList: ArrayList < Token >)

使用 Token 数组列表类型的参数 tokArrList 创建一个新的 Tokens。

3.4.2 常用属性及函数

- public prop size: Int64

获取 Tokens 对象中包含 Token 的数量。

- public func get(index: Int64): Token

获取参数 index 处对应的 Token,当 index 超出 Tokens 的范围时,将抛出异常。

- public func concat(ts: Tokens): Tokens

将当前 Tokens 与传入的参数 ts 表示的 Tokens 进行拼接,返回拼接后的 Tokens。

- public func dump(): Unit

打印 Tokens 信息。

- public func toString(): String

将当前 Tokens 转换为 String 表示形式。

使用构造函数和常用函数的示例代码如下:

```
//Chapter3/tokens_demo/src/tokens_demo.cj

from std import ast.*
from std import collection.*

main() {
    //标识符 a
    let tokenA = Token(TokenKind.IDENTIFIER, "a")
```

```
    //标识符 b
    let tokenB = Token(TokenKind.IDENTIFIER, "b")

    //操作符 +
    let tokenAdd = Token(TokenKind.ADD)

    //创建 Token 数组
    let tokenArray = [tokenA, tokenAdd, tokenB]

    //使用 Token 数组创建 tokens
    let tokenExpList = Tokens(tokenArray)

    //打印 tokens 的字符串形式,也就是 a + b
    println(tokenExpList.toString())

    //输出 tokens 的详细信息
    tokenExpList.dump()

    //关键字 let
    let tokenLet = Token(TokenKind.LET)

    //标识符 c
    let tokenC = Token(TokenKind.IDENTIFIER, "c")

    //操作符 =
    let tokenAssign = Token(TokenKind.ASSIGN)

    //创建 Token 数组列表,然后加入 tokenLet、tokenC、tokenAssign
    let tokenArrayList = ArrayList<Token>()
    tokenArrayList.append(tokenLet)
    tokenArrayList.append(tokenC)
    tokenArrayList.append(tokenAssign)

    //使用 Token 数组列表创建 tokensDef
    let tokensDef = Tokens(tokenArrayList)

    //打印 tokensDef 的字符串形式,也就是 let c =
    println(tokensDef.toString())

    //拼接 tokensDef 和 tokenExpList
    let newTokens = tokensDef.concat(tokenExpList)

    //输出拼接后的 newTokens 字符串表示形式,也就是 let c = a + b
    println(newTokens.toString())
}
```

编译后运行该示例,命令及回显如下:

```
cjc tokens_demo.cj
main.exe
a + b
description: identifier, token_id: 133, token_literal_value: a, fileID: 1, line: 6, column: 18
description: add, token_id: 12, token_literal_value: +, fileID: 1, line: 12, column: 20
description: identifier, token_id: 133, token_literal_value: b, fileID: 1, line: 9, column: 18
let c =
let c = a + b
```

3.4.3 运算符重载函数

为了方便元编程对 Tokens 对象的操作,Tokens 类型提供了运算符重载函数。

- public operator func [](index: Int64): Token

获取参数 index 对应位置的 Token 对象。

- public operator func +(r: Tokens): Tokens

把参数 r 代表的 Tokens 对象拼接到当前 Tokens 后面并返回新的 Tokens。

- public operator func +(r: Token): Tokens

把参数 r 代表的 Token 对象拼接到当前 Tokens 后面并返回新的 Tokens。

Tokens 的运算符重载函数,示例代码如下:

```
//Chapter3/tokens_operator/src/tokens_operator_demo.cj

from std import ast.*

main() {
    //标识符 a
    let tokenA = Token(TokenKind.IDENTIFIER, "a")

    //构造 tokensExp 对象
    var tokensExp = Tokens([tokenA])

    //tokens 对象加上 token 对象
    tokensExp = tokensExp + Token(TokenKind.ADD)

    //构造 tokens 对象
    let tokens = Tokens([Token(TokenKind.IDENTIFIER, "b")])

    //两个 Tokens 对象相加
```

```
    tokensExp = tokensExp + tokens

    //打印 tokensExp 中的第 2 个 Token 字面量
    println(tokensExp[1].value)

    //打印 tokensExp 字符串表示形式
    println(tokensExp.toString())
}
```

编译后运行该示例，命令及回显如下：

```
cjc tokens_operator_demo.cj
main.exe
+
a + b
```

3.5　quote 表达式

在元编程中，很少直接通过构造函数创建 Tokens 对象，一般使用 quote 表达式把代码直接转换为 Tokens 对象，这样更直观、更方便。quote 表达式使用 quote 关键字，后面是一对圆括号，圆括号内是仓颉代码，一个典型的 quote 表达式的用法如下：

```
let tokensExp = quote(a + b)
```

这个 quote 表达式把 a＋b 这行代码转换为由标识符“a”、操作符“＋”、标识符“b”这 3 个 Token 组成的 Tokens。

quote 表达式支持多行代码，一个稍微复杂的使用 quote 表达式的示例代码如下：

```
//Chapter3/quote_demo/src/quote_demo.cj

from std import ast.*

main() {
    let tokens = quote(
    class pos {
        pos(let x: Int64, let y: Int64) {}
    }
```

```
    )

    for (token in tokens) {
        token.dump()
    }
}
```

在这段代码的 quote 表达式里，定义了一个名称为 pos 的 class，编译后运行该示例，命令及回显如下：

```
cjc quote_demo.cj
main.exe
description: newline, token_id: 138, token_literal_value: \n, fileID: 1, line: 4, column: 24
description: class, token_id: 84, token_literal_value: class, fileID: 1, line: 5, column: 5
description: identifier, token_id: 133, token_literal_value: pos, fileID: 1, line: 5, column: 11
description: l_curl, token_id: 6, token_literal_value: {, fileID: 1, line: 5, column: 15
description: newline, token_id: 138, token_literal_value: \n, fileID: 1, line: 5, column: 16
description: identifier, token_id: 133, token_literal_value: pos, fileID: 1, line: 6, column: 9
description: l_paren, token_id: 2, token_literal_value: (, fileID: 1, line: 6, column: 12
description: let, token_id: 90, token_literal_value: let, fileID: 1, line: 6, column: 13
description: identifier, token_id: 133, token_literal_value: x, fileID: 1, line: 6, column: 17
description: colon, token_id: 28, token_literal_value: :, fileID: 1, line: 6, column: 18
description: Int64, token_id: 64, token_literal_value: Int64, fileID: 1, line: 6, column: 20
description: comma, token_id: 1, token_literal_value: ,, fileID: 1, line: 6, column: 25
description: let, token_id: 90, token_literal_value: let, fileID: 1, line: 6, column: 27
description: identifier, token_id: 133, token_literal_value: y, fileID: 1, line: 6, column: 31
description: colon, token_id: 28, token_literal_value: :, fileID: 1, line: 6, column: 32
description: Int64, token_id: 64, token_literal_value: Int64, fileID: 1, line: 6, column: 34
description: r_paren, token_id: 3, token_literal_value: ), fileID: 1, line: 6, column: 39
description: l_curl, token_id: 6, token_literal_value: {, fileID: 1, line: 6, column: 41
description: r_curl, token_id: 7, token_literal_value: }, fileID: 1, line: 6, column: 42
description: newline, token_id: 138, token_literal_value: \n, fileID: 1, line: 6, column: 43
description: r_curl, token_id: 7, token_literal_value: }, fileID: 1, line: 7, column: 5
description: newline, token_id: 138, token_literal_value: \n, fileID: 1, line: 7, column: 6
```

在这个示例中，要注意打印输出内容中的“换行”(newline，字面量输出为\n)，与“空格”等被忽略的空白字符不同，“换行”被识别为一个独立的 Token。另外，还要注意各个 token 的行号和列号也是各不相同的，这与 Token 对应的源码字符串在源文件中的行、列位置相匹配。

3.6　插值运算符

在 quote 表达式中，支持插值操作，使用的插值运算符为 $，$ 符号后跟圆括号，圆括号内是代表插值的表达式，这类似占位操作，在最终使用 quote 表达式时，插值表达式会被替换为实际的值。一个典型的插值表达式如下：

```
let tokenPlus = Token(TokenKind.ADD)
let exp = quote(a $(tokenPlus) b)
```

在这个示例中，先是定义了一个表示加号的 tokenPlus，然后把它作为插值表达式的一部分组合成了 exp 对象，这时，exp 对象和下面的 quote 表达式是等效的：

```
quote(a + b)
```

在默认情况下，插值运算符后面的表达式需要使用圆括号限定，如果该表达式只包括单个标识符，则可以省略圆括号，这样，上例的代码可以改写为如下形式：

```
let tokenPlus = Token(TokenKind.ADD)
let exp = quote(a $tokenPlus b)
```

需要注意的是，插值运算符后面的表达式需要实现 ast 包里的 ToTokens 接口，这是因为在最终替换插值表达式时，是通过调用表达式的 toTokens 函数实现的，也就是把 toTokens 函数返回的值作为替换后的值。

插值运算符在仓颉元编程中，特别是在后续章节介绍实现自定义宏时，使用非常方便，例如一个实现自定义 class 名称的示例如下：

```
//Chapter3/interpolation_demo/src/interpolation_demo.cj

from std import ast.*

main() {
    //表示类名称的令牌
    let tokenName = Token(TokenKind.IDENTIFIER, "pos")

    //使用插值表达式创建表示类的 Tokens
```

```
    let tokens = quote(class $tokenName {
        $tokenName(let x: Int64, let y: Int64) {}
    }
    )

    //输出每个令牌,如果令牌是换行就输出换行
    for (token in tokens) {
        if (token.kind == TokenKind.NL) {
            println()
        } else {
            print(token.value)
            print(" ")
        }
    }
}
```

编译后运行该示例,命令及回显如下:

```
cjc interpolation_demo.cj
main.exe
class pos {
pos ( let x : Int64 , let y : Int64 ) { }
}
```

从输出可以看到,这里生成了一个名称为 pos 的 class,其实,如果有必要,则可以把类名称改成其他的名字。仓颉的插值运算符有非常灵活的运用方式,它可以在既有的代码基础上对代码进行修改,从而形成新的代码,最终完成一段代码到另一段代码的转换,这也是仓颉宏编程的常用实现方式,后续会详细介绍。

3.7 词法解析函数

除了 quote 表达式外,仓颉还提供了词法解析函数 cangjieLex,用来对源码字符串进行解析并得到 Tokens,该函数的定义如下:

- public func cangjieLex(code: String): Tokens

对参数 code 代表的字符串进行词法解析,返回词法解析后得到的 Tokens。

下面通过一个示例演示词法解析函数的用法,该示例会解析一个名称为 Point 的 class

定义，然后输出该定义所有的 Token 对象信息，示例代码如下：

```
//Chapter3/lex_demo/src/lex_demo.cj

from std import ast.*

main(): Unit {
    //定义变量 code，code 中保存表示类 Point 定义的字符串
    let code = """
        class Point {
            Point(let x: Int64, let y: Int64) {}
        }"""

    //对 code 中的字符串进行解析，得到解析后的 Tokens
    let tokens = cangjieLex(code)

    //输出 tokens 的内容
    for (token in tokens) {
        token.dump()
    }
}
```

编译后运行该示例，命令及回显如下：

```
cjc lex_demo.cj
main.exe
description: class, token_id: 84, token_literal_value: class, fileID: 0, line: 1, column: 9
description: identifier, token_id: 133, token_literal_value: Point, fileID: 0, line: 1,
column: 15
description: l_curl, token_id: 6, token_literal_value: {, fileID: 0, line: 1, column: 21
description: newline, token_id: 138, token_literal_value: \n, fileID: 0, line: 1, column: 22
description: identifier, token_id: 133, token_literal_value: Point, fileID: 0, line: 2,
column: 13
description: l_paren, token_id: 2, token_literal_value: (, fileID: 0, line: 2, column: 18
description: let, token_id: 90, token_literal_value: let, fileID: 0, line: 2, column: 19
description: identifier, token_id: 133, token_literal_value: x, fileID: 0, line: 2, column: 23
description: colon, token_id: 28, token_literal_value: :, fileID: 0, line: 2, column: 24
description: Int64, token_id: 64, token_literal_value: Int64, fileID: 0, line: 2, column: 26
description: comma, token_id: 1, token_literal_value: ,, fileID: 0, line: 2, column: 31
description: let, token_id: 90, token_literal_value: let, fileID: 0, line: 2, column: 33
description: identifier, token_id: 133, token_literal_value: y, fileID: 0, line: 2, column: 37
description: colon, token_id: 28, token_literal_value: :, fileID: 0, line: 2, column: 38
description: Int64, token_id: 64, token_literal_value: Int64, fileID: 0, line: 2, column: 40
```

```
description: r_paren, token_id: 3, token_literal_value: ), fileID: 0, line: 2, column: 45
description: l_curl, token_id: 6, token_literal_value: {, fileID: 0, line: 2, column: 47
description: r_curl, token_id: 7, token_literal_value: }, fileID: 0, line: 2, column: 48
description: newline, token_id: 138, token_literal_value: \n, fileID: 0, line: 2, column: 49
description: r_curl, token_id: 7, token_literal_value: }, fileID: 0, line: 3, column: 9
description: end, token_id: 139, token_literal_value: , fileID: 0, line: 3, column: 10
```