翻译：superkam　校对：numbbbbb

词法结构

本页包含内容：

Swift 的“词法结构（*lexical structure*）”描述了如何在该语言中用字符序列构建合法标记，组成该语言中最底层的代码块，并在之后的章节中用于描述语言的其他部分。

通常，标记在随后介绍的语法约束下，由 Swift 源文件的输入文本中提取可能的最长子串生成。这种方法称为“最长匹配项（*longest match*）”，或者“最大适合”（*maximal munch*）。

空白与注释

空白（*whitespace*）有两个用途：分隔源文件中的标记和区分运算符属于前缀还是后缀，（参见 运算符）在其他情况下则会被忽略。以下的字符会被当作空白：空格（*space*）（U+0020）、换行符（*line feed*）（U+000A）、回车符（*carriage return*）（U+000D）、水平 tab（*horizontal tab*）（U+0009）、垂直 tab（*vertical tab*）（U+000B）、换页符（*form feed*）（U+000C）以及空（*null*）（U+0000）。

注释（*comments*）被编译器当作空白处理。单行注释由 `//` 开始直到该行结束。多行注释由 `/*` 开始，以 `*/` 结束。可以嵌套注释，但注意注释标记必须匹配。

标识符

标识符（*identifiers*）可以由以下的字符开始：大写或小写的字母 `A` 到 `Z`、下划线 `_`、基本多语言面（*Basic Multilingual Plane*）中的 Unicode 非组合字符以及基本多语言面以外的非专用区（*Private Use Area*）字符。首字符之后，标识符允许使用数字和 Unicode 字符组合。

使用保留字（*reserved word*）作为标识符，需要在其前后增加反引号 `` ` ``。例如，`class` 不是合法的标识符，但可以使用 `` `class` ``。反引号不属于标识符的一部分，`` `x` `` 和 `x` 表示同一标识符。

闭包（*closure*）中如果没有明确指定参数名称，参数将被隐式命名为 `$0`、`$1`、`$2`... 这些命名在闭包作用域内是合法的标识符。

标识符语法

identifier → identifier-head identifier-characters *opt*

identifier → ` identifier-head identifier-characters *opt* `

identifier → implicit-parameter-name

identifier-list → identifier | identifier , identifier-list

identifier-head → A 到 Z 大写或小写字母

identifier-head → U+00A8, U+00AA, U+00AD, U+00AF, U+00B2–U+00B5, 或 U+00B7–U+00BA

identifier-head → U+00BC–U+00BE, U+00C0–U+00D6, U+00D8–U+00F6, 或 U+00F8–U+00FF

identifier-head → U+0100–U+02FF, U+0370–U+167F, U+1681–U+180D, 或 U+180F–U+1DBF

identifier-head → U+1E00–U+1FFF

identifier-head → U+200B–U+200D, U+202A–U+202E, U+203F–U+2040, U+2054, 或 U+2060–U+206F

identifier-head → U+2070–U+20CF, U+2100–U+218F, U+2460–U+24FF, 或 U+2776–U+2793

identifier-head → U+2C00–U+2DFF 或 U+2E80–U+2FFF

identifier-head → U+3004–U+3007, U+3021–U+302F, U+3031–U+303F, 或 U+3040–U+D7FF

identifier-head → U+F900–U+FD3D, U+FD40–U+FDCF, U+FDF0–U+FE1F, 或 U+FE30–U+FE44

identifier-head → U+FE47–U+FFFD

identifier-head → U+10000–U+1FFFD, U+20000–U+2FFFD, U+30000–U+3FFFD, 或 U+40000–U+4FFFD

identifier-head → U+50000–U+5FFFD, U+60000–U+6FFFD, U+70000–U+7FFFD, 或 U+80000–U+8FFFD

identifier-head → U+90000–U+9FFFD, U+A0000–U+AFFFD, U+B0000–U+BFFFD, 或 U+C0000–U+CFFFD

identifier-head → U+D0000–U+DFFFD 或 U+E0000–U+EFFFD

identifier-character → 数字 0 到 9

identifier-character → U+0300–U+036F, U+1DC0–U+1DFF, U+20D0–U+20FF, or U+FE20–U+FE2F

identifier-character → identifier-head

identifier-characters → identifier-character identifier-characters *opt*

implicit-parameter-name → $ decimal-digits

关键字

被保留的关键字（*keywords*）不允许用作标识符，除非被反引号转义，参见 标识符。

- **用作声明的关键字：** *class*、*deinit*、*enum*、*extension*、*func*、*import*、*init*、*let*、*protocol*、*static*、*struct*、*subscript*、*typealias*、*var*
- **用作语句的关键字：** *break*、*case*、*continue*、*default*、*do*、*else*、*fallthrough*、*if*、*in*、*for*、*return*、*switch*、*where*、*while*
- **用作表达和类型的关键字：** *as*、*dynamicType*、*is*、*new*、*super*、*self*、*Self*、*Type*、*__COLUMN__*、*__FILE__*、*__FUNCTION__*、*__LINE__*
- **特定上下文中被保留的关键字：** *associativity*、*didSet*、*get*、*infix*、*inout*、*left*、*mutating*、*none*、*nonmutating*、*operator*、*override*、*postfix*、*precedence*、*prefix*、*right*、*set*、*unowned*、*unowned(safe)*、*unowned(unsafe)*、*weak*、*willSet*，这些关键字在特定上下文之外可以被用于标识符。

字面量

字面值表示整型、浮点型数字或文本类型的值，举例如下：

```
42                  // 整型字面量
3.14159             // 浮点型字面量
"Hello, world!"     // 文本型字面量
```

字面量语法

literal → integer-literal | floating-point-literal | string-literal

整型字面量

整型字面量（*integer literals*）表示未指定精度整型数的值。整型字面量默认用十进制表示，可以加前缀来指定其他的进制，二进制字面量加 `0b`，八进制字面量加 `0o`，十六进制字面量加 `0x`。

十进制字面量包含数字 `0` 至 `9`。二进制字面量只包含 `0` 或 `1`，八进制字面

量包含数字 `0` 至 `7`，十六进制字面量包含数字 `0` 至 `9` 以及字母 `A` 至 `F` （大小写均可）。

负整数的字面量在数字前加减号 `-`，比如 `-42`。

允许使用下划线 `_` 来增加数字的可读性，下划线不会影响字面量的值。整型字面量也可以在数字前加 `0`，同样不会影响字面量的值。

```
1000_000      // 等于 1000000
005           // 等于 5
```

除非特殊指定，整型字面量的默认类型为 Swift 标准库类型中的 `Int`。Swift 标准库还定义了其他不同长度以及是否带符号的整数类型，请参考整数类型。

整型字面量语法

integer-literal → binary-literal

integer-literal → octal-literal

integer-literal → decimal-literal

integer-literal → hexadecimal-literal

binary-literal → **0b** binary-digit binary-literal-characters *opt*

binary-digit → 数字 0 或 1

binary-literal-character → binary-digit | _

binary-literal-characters → binary-literal-character binary-literal-characters *opt*

octal-literal → **0o** octal-digit octal-literal-characters *opt*

octal-digit → 数字 0 至 7

octal-literal-character → octal-digit | _

octal-literal-characters → octal-literal-character octal-literal-characters *opt*

decimal-literal → decimal-digit decimal-literal-characters *opt*

decimal-digit → 数字 0 至 9

decimal-digits → decimal-digit decimal-digits *opt*

decimal-literal-character → decimal-digit | _

decimal-literal-characters → decimal-literal-character decimal-literal-characters *opt*

hexadecimal-literal → **0x** hexadecimal-digit hexadecimal-literal-characters *opt*

hexadecimal-digit → 数字 0 到 9, a 到 f, 或 A 到 F

hexadecimal-literal-character → hexadecimal-digit | _

hexadecimal-literal-characters → hexadecimal-literal-character hexadecimal-literal-characters *opt*

浮点型字面量

浮点型字面量（*floating-point literals*）表示未指定精度浮点数的值。

浮点型字面量默认用十进制表示（无前缀），也可以用十六进制表示（加前缀 `0x`）。

十进制浮点型字面量（*decimal floating-point literals*）由十进制数字串后跟小数部分或指数部分（或两者皆有）组成。十进制小数部分由小数点 `.` 后跟十进制数字串组成。指数部分由大写或小写字母 `e` 后跟十进制数字串组成，这串数字表示 `e` 之前的数量乘以 10 的几次方。例如：`1.25e2` 表示 `1.25 × 10^2`，也就是 `125.0`；同样，`1.25e-2` 表示 `1.25 × 10^-2`，也就是 `0.0125`。

十六进制浮点型字面量（*hexadecimal floating-point literals*）由前缀 `0x` 后跟可选的十六进制小数部分以及十六进制指数部分组成。十六进制小数部分由小数点后跟十六进制数字串组成。指数部分由大写或小写字母 `p` 后跟十进制数字串组成，这串数字表示 `p` 之前的数量乘以 2 的几次方。例如：`0xFp2` 表示 `15 × 2^2`，也就是 `60`；同样，`0xFp-2` 表示 `15 × 2^-2`，也

就是 `3.75`。

与整型字面量不同，负的浮点型字面量由一元运算符减号 `-` 和浮点型字面量组成，例如 `-42.0`。这代表一个表达式，而不是一个浮点整型字面量。

允许使用下划线 `_` 来增强可读性，下划线不会影响字面量的值。浮点型字面量也可以在数字前加 `0`，同样不会影响字面量的值。

```
10_000.56      // 等于 10000.56
005000.76      // 等于 5000.76
```

除非特殊指定，浮点型字面量的默认类型为 Swift 标准库类型中的 `Double`，表示64位浮点数。Swift 标准库也定义 `Float` 类型，表示32位浮点数。

浮点型字面量语法

floating-point-literal → decimal-literal decimal-fraction *opt* decimal-exponent *opt*

floating-point-literal → hexadecimal-literal hexadecimal-fraction *opt* hexadecimal-exponent

decimal-fraction → . decimal-literal

decimal-exponent → floating-point-e sign *opt* decimal-literal

hexadecimal-fraction → . hexadecimal-literal *opt*

hexadecimal-exponent → floating-point-p sign *opt* hexadecimal-literal

floating-point-e → **e** | **E**

floating-point-p → **p** | **P**

sign → **+** | **-**

文本型字面量

文本型字面量（*string literal*）由双引号中的字符串组成，形式如下：

```
"characters"
```

文本型字面量中不能包含未转义的双引号 `"`、未转义的反斜线`\`、回车符（*carriage return*）或换行符（*line feed*）。

可以在文本型字面量中使用的转义特殊符号如下：

- 空字符（Null Character）`\0`
- 反斜线（Backslash）`\\`
- 水平 Tab （Horizontal Tab）`\t`
- 换行符（Line Feed）`\n`
- 回车符（Carriage Return）`\r`
- 双引号（Double Quote）`\"`
- 单引号（Single Quote）`\'`

字符也可以用以下方式表示：

- `\x` 后跟两位十六进制数字
- `\u` 后跟四位十六进制数字
- `\U` 后跟八位十六进制数字

后跟的数字表示一个 Unicode 码点。

文本型字面量允许在反斜线小括号 `\()` 中插入表达式的值。插入表达式（*interpolated expression*）不能包含未转义的双引号 `"`、反斜线 `\`、回车符或者换行符。表达式值的类型必须在 *String* 类中有对应的初始化方法。

例如，以下所有文本型字面量的值相同：

```
"1 2 3"
"1 2 \(3)"
"1 2 \(1 + 2)"
var x = 3; "1 2 \(x)"
```

文本型字面量的默认类型为 `String`。组成字符串的字符类型为 `Character`。更多有关 `String` 和 `Character` 的信息请参照 字符串和字符。

文本型字面量语法

string-literal → " quoted-text "

quoted-text → quoted-text-item quoted-text *opt*

quoted-text-item → escaped-character

quoted-text-item → (expression)

quoted-text-item → 除 `"`、`\`、`U+000A` 或 `U+000D` 以外的任何 Unicode 扩展字符集

escaped-character → **\0** | **** | **\t** | **\n** | **\r** | **\"** | **\'**

escaped-character → **\x** hexadecimal-digit hexadecimal-digit

escaped-character → **\u** hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

escaped-character → **\U** hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

运算符

Swift 标准库定义了许多可供使用的运算符，其中大部分在 基础运算符 和 高级运算符 中进行了阐述。这里将描述哪些字符能用作运算符。

运算符由一个或多个以下字符组成：`/`、`=`、`-`、`+`、`!`、`*`、`%`、`<`、`>`、`&`、`|`、`^`、`~`、`.`。也就是说，标记 `=`, `->`、`//`、`/*`、`*/`、`.` 以及一元前缀运算符 `&` 属于保留字，这些标记不能被重写或用于自定义运算符。

运算符两侧的空白被用来区分该运算符是否为前缀运算符（*prefix operator*）、后缀运算符（*postfix operator*）或二元运算符（*binary operator*）。规则总结如下：

- 如果运算符两侧都有空白或两侧都无空白，将被看作二元运算符。例如：`a+b` 和 `a + b` 中的运算符 `+` 被看作二元运算符。
- 如果运算符只有左侧空白，将被看作前缀一元运算符。例如 `a ++b` 中的 `++` 被看作前缀一元运算符。
- 如果运算符只有右侧空白，将被看作后缀一元运算符。例如 `a++ b`

中的 `++` 被看作后缀一元运算符。

- 如果运算符左侧没有空白并紧跟 `.`，将被看作后缀一元运算符。例如 `a++.b` 中的 `++` 被看作后缀一元运算符（同理，`a++ . b` 中的 `++` 是后缀一元运算符而 `a ++ .b` 中的 `++` 不是）.

鉴于这些规则，运算符前的字符 `(`、`[` 和 `{`；运算符后的字符 `)`、`]` 和 `}` 以及字符 `,`、`;` 和 `:` 都将用于空白检测。

以上规则需注意一点，如果运算符 `!` 或 `?` 左侧没有空白，则不管右侧是否有空白都将被看作后缀运算符。如果将 `?` 用作可选类型（*optional type*）修饰，左侧必须无空白。如果用于条件运算符 `? :`，必须两侧都有空白。

在特定构成中，以 `<` 或 `>` 开头的运算符会被分离成两个或多个标记，剩余部分以同样的方式会被再次分离。因此，在 `Dictionary<String, Array<Int>>` 中没有必要添加空白来消除闭合字符 `>` 的歧义。在这个例子中，闭合字符 `>` 被看作单字符标记，而不会被误解为移位运算符 `>>`。

要学习如何自定义新的运算符，请参考 自定义操作符 和 运算符声明。学习如何重写现有运算符，请参考 运算符方法。

运算符语法

operator → operator-character operator *opt*

operator-character → / | = | - | + | ! | * | % | < | > | & | | | ^ | ~ | .

binary-operator → operator

prefix-operator → operator

postfix-operator → operator