

Golang

sanfusu <sanfusu@foxmail.com>

2018 年 7 月 9 日

第一章 Introduction

这是一份关于 Go 编程语言的参考指南。更多信息以及其余文档，请查看 golang.org

Go 是以系统语言为宗旨而设计的通用目的语言。Go 具有强类型和垃圾回收并且带有对并发编程的显式支持。程序通过 *packages* 构造，其属性可以有效的管理依赖。

语法方面，紧凑且有规律，这允许便捷的使用类似集成开发环境的自动化工具进行分析。

第二章 Notation

语法格式通过扩展的巴克斯范式¹表示。

```
Production = production_name "=" [ Expression ] "." .
Expression = Alternative { "|" ALternative } .
Alternative = Term { Term } .
Term        = production_name | token [ "..." token ] | Group | Option |
              Repetition .
Group       = "(" Expression ")" .
Option      = "[" Expression "]" .
Repetition = "{" Expression "}" .
```

Productions 是由 terms 以及下列操作符构成的表达式，按递增优先级如下：

- | alternation
- () grouping
- [] option (0 or 1 times)
- { } repetition (0 to n times)

小写的产生式名被用来标识词法记号。非终止符使用峰驼命名法。词法记号被包含在双引号””或者反撇号 `` 中。

a ... b 形式标识 a 到 b 的可替代集合。水平省略号 ... 在本规范中用来表示各种枚举或代码片段。字符 ...（与三个字符 ... 相反）并不是 Go 语言中的记号。

¹Extended Backus-Naur Form(EBNF)

第三章 Source code representation

原代码被编码成 UTF-8 格式的 Unicode 文本。这些文本并不是正规化的，所以单个重音编码点与那些通过重音符合字母构成的相同字符并不一样，组合出来的字符被视为两个编码点。简单的来说，本文档将使用未加任何限制的术语 *character*（字符）来指示源文本中的 Unicode 编码点。

区分每一个编码点，比如，大写字母和小写字母是不同的字符。

实现上的限制：为了和其他工具兼容，编译器可能不允许 NUL 字符（U+0000）出现在源文本中。

同样是实现上的限制：为了兼容其他工具，如果 UTF-8 的字节序记号（U+FEFF）位于源文本的第一个 Unicode 编码点，则编译器可能会忽视该记号。字节序记号可能会被禁止出现在源文本的其他任何地方。

† I Characters

下列术语被用来标识特定的 Unicode 字符类：

```
newline      =  
/* the Unicode code point U+000A */ .  
unicode_char =  
/* an arbitrary Unicode code point except newline */ .  
unicode_letter =  
/* a Unicode code point classified as "Letter" */ .  
unicode_digit =  
/* a Unicode code point classified as "Number, decimal digit" */ .
```

在 Unicode 8.0 标准中，第 4.5 节“General Category”中，定义了一组字符类别。Go 将字母分类 Lu, Ll, Lt, Lm 或者 Lo 中的所有字符均作为 Unicode 字母，并将 Number 类别中的 Nd 作为 Unicode 数字。

† II Letters and digits

下划线字符 _ (U+005F) 被视为一个字母。

```
letter      = unicode_letter | "_" .  
decimal_digit = "0" ... "9" .
```

```
octal_digit = "0" ... "7" .  
hex_digit  = "0" ... "9" | "A" ... "F" | "a" ... "f" .
```

第四章 Lexical elements

† I 注释

注释充当程序文档。有两种形式：

1. 行注释以字符序列 `//` 开头，并且结束于行尾。
2. 通用注释起始于 `/*` 并且结束于随后所遇到的第一个字符序列 `*/`。

注释不能起始于一个 rune 或者字符串文字，或者一个注释内部。不包含换行的通用注释和空格的作用一样。任何其他注释都和换行的作用一致。

† II 记号

记号形成 Go 语言的词汇。有四种类别的记号：标识符，关键词，操作符和标点符，以及字面量。*White space* 由空格 (U+0020)，水平制表 (U+0009)，回车 (U+000D) 和换行 (U+000A) 组成，如果空白字符没起到分割记号的作用，则将会被忽视。另外，换行和文件结尾会触发分号的插入。当输入分解为记号时，下一个记号将会是形成有效记号的最长字符序列。

† III 分号

形式化语法使用分号 “;” 作为产生式的终止符。Go 程序可以通过以下两个规则省略大部分分号：

1. 当输入分解为语言符号时，如果一行的最后一个语言符号为以下几种情况，则会自动插入到行末。
 - 标志符
 - 整型，浮点，虚数，rune，或者字符串文字
 - 关键词 `break`, `continue`, `fallthrough`, 或者 `return` 中的一个。
 - `++`, `--`, `)`, `]` 或 `}` 运算符和标点符号中的一个。
2. 为了能让复合语句占据单行，`)` 和 `}` 前的分号可以省略。

为了反映惯用的用法，本文档中的代码会使用以上规则来省略掉分号。

† IV 标识符

标识符命名程序条目，比如变量和类型。一个标识符是一个或者多个字母和数字的序列。标识符中的第一个字符必须是一个字母。

```
identifier = letter { letter | unicode_digit } .
```

```
a
_x9
ThisVariableIsExported
αβ
```

部分标识符是预先声明的。

† V 关键词

以下保留的关键词，并且不能用作标识符。

```
break      default      func         interface   select
case       defer        go           map         struct
chan       else         goto        package    switch
const      fallthrough if           range      type
continue   for          import      return     var
```

† VI 运算符和标点符号

以下字符序列标识运算符（包括赋值操作符）以及标点符号：

```
+  &   +=   &=   &&   ==   !=   (   )
-  |   -=   |=   ||   <   <=  [   ]
*  ^   *=   ^=   <-  >   >=  {   }
/  <<  /=   <<=  ++   =   :=   ,   ;
%  >>  %=   >>=  --   !   ...  .   :
&^  &^=
```

† VII 整型字面量

一个整型字面量是标识整型常量的一个数字序列。可选前缀可以设置一个非十进制基：0 为八进制，0x 或者 0X 为十六进制。在十六进制字面量中，字母 a-f 和 A-F 代表 10 到 15。

```
int_lit     = decimal_lit | octal_lit | hex_lit .
decimal_lit = ( "1" ... "9" ) { decimal_digit } .
octal_lit   = "0" { octal_digit } .
hex_lit     = "0" ( "x" | "X" ) hex_digit { hex_digit } .
```

```
42
0600
OxBadFace
170141183460469231731687303715884105727
```

† VIII 浮点字面量

浮点字面量是浮点常量的十进制表示。它具有一个整数部分，一个十进制小数点，一个小数部分，和一个指数部分。整数和小数部分有十进制数组成；指数部分为 e 或者 E 后面紧跟着一个可选的有符号十进制指数。可以省略整数部分或者小数部分中的一个；类似小数点或者指数中的一个可以被省略。

```
float_lit = decimals "." [ decimals ] [ exponent ] |
           decimals exponent |
           "." decimals [ exponent ] .
decimals  = decimal_digit { decimal_digit } .
exponent  = ( "e" | "E" ) [ "+" | "-" ] decimals .
```

```
0.
72.40
072.40 // == 72.40
2.71828
1.e+0
6.67428e-11
1E6
.25
.12345E+5
```

† IX 虚数字面量

虚数字面量是复数常量虚部的十进制表示。他由后面跟着字母 i 的浮点字面量或十进制整数组成。

```
imaginary_lit = (decimals | float_lit) "i" .
```

```
0i
011i // == 11i
0.i
2.71828i
1.e+0i
6.67428e-11i
1E6i
.25i
```



```
.12345E+5i
```

† X Rune literals

一个符文字面量表示一个符文常量，一个用来标识 Unicode 编码点的整数值。一个符文字面量使用单引号包含一个或多个字符来表达，如 'x' 或者 '\n'。除换行和未转义的单引号外，任何字符都可以出现在单引号内。使用单引号包含的字符代表了字符本身的 Unicode 值，但是以反斜杠开头的多字符序列会以各种格式对值进行编码。

符文字面量的最简单形式是在引号内表示单个字符；由于 Go 源文本是 UTF-8 编码的 Unicode 字符，所以多个 UTF-8 编码的字节可以使用单个整数值表示。比如字面值 'a' 拥有表示字面量 a 的单个字节，Unicode 编码点为 U+0061，值为 0x61，但是 'ä' 则拥有两个字节，用来表示 a 的分音符号，U+00E4，值为 0xe4。

有几个反斜杠转义允许将任意值编码为 ASCII 文本。有四种方式将整型值表示为数字常量：\x 后面跟上两个十六进制数；\u 后面跟上四个十六进制数；\U 后面跟上八个十六进制数；\ 后面跟上三个八进制数。每一种表示方法中，字面量的值为数字使用相应的进制表示的值。

尽管这些表示的结果均为整型，但是他们拥有不同的有效范围。八进制转义表示的值必须在 0 到 255 范围内（包含 0 和 255）。十六进制转义通过构造来满足该条件。转义 \u 和 \U 表示 Unicode 编码点，因此他们所表示的部分值可能是非法的，特别是大于 0x10FFFF 的值以及 surrogate halves。

反斜杠后面跟上特定的单字符表示特殊的值：

```
\a  U+0007 alert or bell
\b  U+0008 backspace
\f  U+000C form feed
\n  U+000A line feed or newline
\r  U+000D carriage return
\t  U+0009 horizontal tab
\v  U+000b vertical tab
\\  U+005c backslash
\'  U+0027 single quote (valid escape only within rune literals)
\"  U+0022 double quote (valid escape only within string literals)
```

在符文字面量里的其余以反斜杠开始的序列均是非法的。

```
rune_lit      = "'" ( unicode_value | byte_value ) "'" .
unicode_value = unicode_char | little_u_value | big_u_value |
    escaped_char .
byte_value    = octal_byte_value | hex_byte_value .
octal_byte_value = "` octal_digit octal_digit octal_digit .
hex_byte_value  = "` "x" hex_digit hex_digit .
little_u_value  = "` "u" hex_digit hex_digit hex_digit hex_digit .
big_u_value     = "` "U" hex_digit hex_digit hex_digit hex_digit
    hex_digit hex_digit hex_digit hex_digit .
```

```
escaped_char = `\

```

```
'a'
'ä'
'本'
'\t'
'\000'
'\007'
'\377'
'\x07'
'\xff'
'\u12e4'
'\U00101234'
'' // rune literal containing single quote character
'aa' // illegal: too many characters
'xa' // illegal: too few hexadecimal digits
'0' // illegal: too few octal digits
'\uDFFF' // illegal: surrogate half
'\U00110000' // illegal: invalid Unicode code point
```

† XI 字符串字面量

字符串字面量表示从相连的字符序列中所获取的字符串常量。字符串字面量有两种形式：原始字符串字面量和已翻译的字符串字面量。

原始字符串字面量是两个反撇号之间的字符序列，比如 ``foo``。除了反撇号自身，任何字符均可以出现在反撇号里。原始字符串字面量的值是由反撇号之间未翻译过（默认是 UTF-8 编码）的字符序列所组成的字符串；值得注意的是，反斜杠没有特殊意义，并且字符串可以包含换行。原始字符串字面量中的回车字符（``\r``）会被舍弃。

翻译过的字符串字面量为双引号之间的字符序列，比如 `"bar"`。除了双引号自身和换行之外任何字符都可以出现在双引号内。双引号之间的文本形成字面量的值，但其中的反斜杠转义会根据符文字面量中的限制来解释（除了 ``\`` 非法，而 ``\"`` 结果字符串中的合法之外）。三个八进制数字（`\nnn` 和两个十六进制数转义表示结果字符串中的个体字节，其他所有的转义均表示个体字符的 UTF-8 编码（也可能是多字节）。因此字符串字面量里面的 `\377` 和 `\xFF` 表示值为 `0xFF=255` 的单字节，而 `ÿ`, `\u00FF`, `\U000000FF` 和 `\xc3\xbf` 表示 UTF-8 编码的字符 `U+00FF` 的两个字节 `0xc3 0xbf`。

```
string_lit = raw_string_lit | interpreted_string_lit .
raw_string_lit = "`" { unicode_char | newline } "`" .
interpreted_string_lit = `\"_{unicode_value_|_byte_value_}_` .
```

```
`abc` // same as "abc"
`\n`
```

```

\n`           // same as "\\n\n\n"
"\n"
"\`"         // same as ` `
"Hello, \uworld!\n"
"日本語"
"\u65e5本\u00008a9e"
"\xff\u00FF"
"\uD800"     // illegal: surrogate half
"\U00110000" // illegal: invalid Unicode code point

```

下面示例均表示相同的字符串：

```

"日本語"           // UTF-8 input text
`日本語`         // UTF-8 input text as a raw literal
"\u65e5\u672c\u8a9e" // the explicit Unicode code points
"\U000065e5\U0000672c\U00008a9e" // the explicit Unicode code points
"\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e" // the explicit UTF-8 bytes

```

如果源代码将一个字符表示为两个编码点，比如音调符号和字母的组合形式，若放在符文字面值中，则结果将会出错，若放置在字符串字面量中则会出现两个编码点。

第五章 常量

有布尔常量，符文常量，整型常量，浮点常量，复数常量和字符串常量。符文，整型，浮点，和复数常量通常为数值常量。

一个常量通过符文，整型，浮点，虚数或字符串字面量，也可以通过一个表示常量的标识符，常量表达式，结果为常量的转换来表示，或者使用一些内置函数，比如适用于任何值的`unsafe.Sizeof` 或者适用于一些表达式的 `cap` 或 `len`，`real` 和 `imag` 适用于复数常量，`complex` 适用于数值常量。布尔真值由欲声明常量 `true` 和 `false` 来表示。预声明标识符 `iota` 则表示一个整型常量。

一般来说，复数常量是常量表达式的一种形式，并且会在相应的章节中讨论。

数值常量表示任意精度的确切值，并且不会导致溢出。因此没有常量可以表示 IEEE-754 的负零，无穷，非数值。常量既可以具有类型也可无类型的。字面常量，`true`，`false`，`iota` 以及一些只包含无类型常量操作数的常量表达式属于无类型常量。

一个常量可以通过常量声明或者转换显式的给出其类型，也可以在使用变量声明或赋值亦或表达式中的一个操作数时隐式的给出类型。如果常量值无法表示相应类型的值，则视为错误。

一个无类型常量具有默认类型，该类型为常量在上下文环境中隐式转换的所需值的类型，比如，没有显式类型的短变量声明 `i := 0`。一个无类型常量的默认类型可以为 `bool`，`rune`，`int`，`float64`，`complex128` 或者 `string`，这取决于其是否是相应类型的常量。

实现上的约束：尽管数值常量在语言中具有任意的精度，但是编译器可能会使用具有受限精度的内部表示。也就是说，每一个实现必须：

- 至少使用 256 个 bit 来表示整型常量。
- 浮点常量，包括复数常量的各分部的尾数部分至少使用 256 bits 来表示，有符二进制指数则至少 16 bits
- 如果无法精确的表示整型常量，则视为错误。
- 如果由于溢出而无法表示一个浮点或复数常量，则视为出错。
- 如果受限于精度而无法表示一个浮点或复数常量，则取最近的可表示的常量。

这些要求同时适用于字面常量和常量表达式的计算结果。

第六章 变量

一个变量时用来保存值的存储位置。变量所允许的值有其类型来决定。

变量声明或者用作函数参数和结果时，以及函数声明的签字或者函数字面量都会为命名变量保留存储空间。通过内置函数 `new` 或者获取符合字面量的地址，则会在运行时为变量申请变量。这种匿名变量（可以隐式的）通过指针间接寻址来访问。

数组，切片，和结构体类型的结构化变量拥有可以独立寻址的元素和字段。每一个这种元素都表现为一个变量。

变量声明时给出的类型，使用 `new` 调用或者复合字面值亦或结构化变量的元素类型为变量的静态类型（简称为类型）。接口类型变量具有不同的动态类型，其类型为运行时所赋值给变量的值的具体类型（除非值为预声明标识符 `nil`，这时候没有类型）。在执行期间动态类型可能会不同，但是存储在接口变量中的值，永远可以赋值给变量的静态类型。

```
var x interface{} // x is nil and has static type interface{}
var v *T          // v has value nil, static type *T
x = 42           // x has value 42 and dynamic type int
x = v           // x has value (*T)(nil) and dynamic type *T
```

一个变量的值可以通过参考表达式中的变量来获取，其值为赋值给变量的最新值。如果变量还未被赋予一个值，则值为其类型的零值。

第七章 类型

† I rune 字面值

有点类似与 C 中的 char 型的 Unicode 版。同样是表示一个字符 C 中的 char 型只能表示 ASCII 范围的字符，而 Go 中的 rune 字面值可以表示 Unicode 中的字符，当然也可以支持转义字符了。

† II 常量

数值常量必须要能够表示任意精度的实际数值，并且不能溢出。所以常量无法直视 IEEE-754 负零、无穷和非数值。常量既可以是 typed 也可以是 untyped。事实上无类型常量具有默认类型，该类型由隐式转换时上下文所需的类型决定。

7.2.1 常量约束

数值常量具体精度由编译器决定，但 Go spec 对编译器做出了最低约束：

- 至少能表示 256 bit 的整型常量。
- 浮点常量（包括复数常量），尾数部分至少 256 bit，有符二进制指数部分至少 16 bit。
- 如果无法准确的表示一个整型常量应当给出错误信息。
- 如果应为溢出而无法表示一个浮点或者复数常量，也应当给出错误信息。
- 如果无法在限定的精度范围内表示一个浮点或复数常量，应当取最接近原始值的可表示的近似值。

† III 变量

一个变量占用一定的存储位置用来保存一个值。变量所允许的值有其变量类型来决定。结构化变量中的元素如果可独立寻址，则亦可作为变量来看待。变量声明会为命名变量申请存储空间。使用 new 或者使用符合字面值的地址会在运行时为一个变量申请存储空间，这种匿名变量可以通过间接指针来访问。

† IV 类型

类型决定了值和可以施加到该值上的操作和方法。复合类型 — array, struct, pointer, function, interface, slice, map, channel。每一个类型 T 都具有一个底层类型¹，底层的类型的底层类型是其本身。使用关键字 `type` 可以进行类型声明。

```
type {
  A1 = string
  A2 = A1
}
```

连个类型如果其底层类型字面结构相同则认为这两个类型相等。

7.4.1 方法集

类型可能会有与其相关联的方法集。一个接口类型的方法集合是其接口。

7.4.2 字符串

字符串类型的值是一个字节序列（该序列可能为空）。字符串类型是不可修改的。一旦创建后便无法在原有基础上修改内容。

7.4.3 切片

和数组类似，切片 (slice) 可索引，并且拥有一个长度。和数组不同的是，slice 类型可以在运行时更改。slice 初始化后回一个底层的数组相关联，该数组用来保存 slice 的元素。因此 slice 会和相同数组的其他 slice 共享存储。可以通过内置函数 `make` 初始化一个给定类型的 slice。关于 Go 中的 slice 可以参照 python 中的概念。

```
make([]T, length, capacity)
```

7.4.4 指针类型

指针类型表示所有指向给定类型变量的指针集合，该类型叫做指针的基类型。未被初始化的指针的值为 `nil`。

7.4.5 接口

A variable of interface type can store a value of any type with a method set that is any superset of the interface.

一个接口类型变量可以存储任何类型为接口方法集超集的值。这种类型被称为实现一个接口²。未初始化的接口类型变量的值为 `nil`。使用 `type` 来定义一个名为 `Locker` 的接口：

¹布尔，数值，字符串，类型字面值

²规范里面说的很拗口，实际上和其他语言的接口类似—要实现一个接口必须实现该接口的全部方法，同时可以添加自己的方法

```
type Locker interface {
    Lock()
    Unlock()
}
```

所有的类型都会实现一个空接口：`interface{}`。接口可以嵌入³，即可定义如下接口类型：

```
type File interface {
    Read(b Buffer) bool
    Write(b Buffer) bool
    Locker // 上文定义的 Locker 接口
    Close()
}
```

7.4.6 Map 类型

Map 由一组单类型元素组成的无序集合，可以通过另一种类型的独一无二的 key 来索引。可以看出 Map 类型由两种类型复合而成—元素类型和索引类型。未初始化的 map 的值为 `nil`。Map 的 EBNF 定义如下：

```
MapType = "map" "[" KeyType "]" ElementType .
```

```
KeyType = Type .
```

示例：

```
map[string]int
```

通过内置函数 `make` 创建空 map，示例：

```
string
make(map[string]int, 100) // 100 为 map 的容量
```

`nil map` 和空 map 类似，但是无法向其中添加元素。Map 的初始容量并不和其大小绑定，map 会根据存储的元素数量来自动扩充。

7.4.7 Channel 类型

Channel 类型通过发送和接受特定元素类型的值为并发执行函数提供通信机制。具体使用方法见官方 spec⁴，重点在于定义时可确定方向（发送方和接受方）。

³类似于 C 的面向对象设计，但要注意无法嵌套父

⁴手酸，懒得解释