更多Golang资料包：https://github.com/0voice/Introduction-to-Golang

介绍

这是一个 Go 语言的参考手册，你也可以访问golang.org获取更多信息和其他文档。

Go 是在设计时考虑了系统编程的通用型编程语言。它是强类型，有垃圾回收机制并原生支持并发编程。Go 程序由一个或多个 package 组成，这样可以高效的管理依赖。

Go 的语法简洁且有规则，这让自动化工具可以很容易的分析代码，例如：集成开发环境。

标记

语法采用扩展巴科斯范式。

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

产生式是由词法单元和以下操作符构成的表达式（优先级依次递增）：

```
|   或
()  分组
[]  可选（出现 0 或 1 次）
{}  可重复（出现 0 到 n 次）
```

小写的产生式名称用来与词法单元区分。非终结符采用驼峰式。词法单元由双引号或反引号组成。

`a...b` 表示从 `a` 到 `b` 之间的任意字符。省略号 `...` 也可以在规范中表示对更详细的枚举和代码片段的省略。字符 `...` 不是 Go 语言的词法单元。

更多Golang资料包：https://github.com/0voice/Introduction-to-Golang

源码表示法

Go 的源代码使用 UTF-8 编码的 Unicode 文本。不过它并不是完全规范化的，单重音的代码点与由相同字符和音标组成的代码点是不同的；前者我们认为它是两个代码点。简单来讲，文档会在源代码文本中使用非规范的术语字符来表示一个 Unicode 代码点。

每个代码点都是不同的；相同字符的大写和小写形式表示不同的字符。

实现限制：为了兼容其他工具，编译器不允许出现 Utf-8 编码的源文本中的 NUL 字符（U+0000）。

实现限制：为了兼容其他工具，如果源文本中是以Utf-8 编码的字节序标记（U+FEFF）为起始代码点。编译器会忽略它。字节序标记不应出现在源文本的任何位置。

字符

这些单词表示 Unicode 字符的类别：

```
newline        = /* Unicode 代码点 U+000A */ .
unicode_char   = /* 排除换行以外的任意 Unicode 代码点 */ .
unicode_letter = /* 一个字母（"Letter"）类型的 Unicode 代码点  */ .
unicode_digit  = /* 一个数字（"Number, decimal digit"）类型的 Unicode 代码点  */ .
```

在 Unicode8.0 标准中，第 4.5 章节 “一般类别” 中定义了字符的类别。Go 能够处理任何字符集，包括 Lu，Li，Lt，Lm 或 Lo 作为 Unicode 字母，还可以把数字字符集 Nd 当作 Unicode 数字处理。

字母和数字

我们认为下划线 `_` （U+005F）是一个字母：

```
letter        = unicode_letter | "_" .
decimal_digit = "0" … "9" .
octal_digit   = "0" … "7" .
hex_digit     = "0" … "9" | "A" … "F" | "a" … "f" .
```

词汇元素

注释

注释是程序的说明文档。在 Go 中有两种形式：

- 单行注释从 `//` 开始直到行末结束。
- 通用注释从 `/*` 开始直到 `*/` 结束。

注释不能嵌套在其他注释、字符串和 rune 的字面值中。不包含换行符的通用注释之间通过空格符连接，其他情况下每段注释都会另起一行。

词汇元素

词汇元素构成了 Go 语言的词汇表。它有四种类型：标识符、关键字、操作符/标点符号、字面值。空白符可以是空格（U+0020）、水平制表符（U+0009）、换行符（U+000D）或换行符（U+000A）。它本身会被忽略，一般用来区分不同的词汇元素。换行符或文件终止符（EOF）还可能触发编译程序在源代码的行末或文件末尾追加分号。在分解源代码的词汇元素的过程中，会把当前可以形成有效词汇元素的最长字符序列作为下一个词汇元素。

分号

正规语法在很多产生式中使用分号 ";" 作为终结符。Go 程序中遵循下面两条规则省略了大部分的分号：

1. 当某行的最后一个词汇元素是以下元素时自动补全分号：

- 一个标识符。
- 一个整数，浮点数，虚数，rune 或字符串字面值。

- 关键字 `break`、`continue`、`fallthrough` 和 `return` 其中之一。
- 操作符/标点符号 `++`，`--`，`)`，`]` 和 `}` 其中之一。

1. 为了支持独占一行的复杂语句，会省略与 ")" 或 "}" 相邻的分号。

为了反应惯用用途，本篇文档的所有例子都基于以上规则省略分号。

标识符

标识符表示程序实体单元，例如：变量、类型。一个标识符由一个或多个字母和数字组成。标识符的首字符必须为字母。

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

```
a
_x9
ThisVariableIsExported
αβ
```

Go 已经预定义了一些标识符。

关键字

以下关键字是预留的，它们不能作为标识符：

```
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
```

操作符和标点符号

以下字符序列用于表示操作符（包括赋值运算符）和标点符号：

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

整型字面值

整型字面值是一个数字序列，相当于整型常量。可以使用前缀指定非小数进制：0 表示八进制，0x/0X 表示十六进制。在十六进制字面值中，字母 a-f 和 A-F 都表示数字 10-15。

```
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
0xBadFace
170141183460469231731687303715884105727
```

浮点字面值

浮点字面值是一个小数，相当于浮点数常量。它由整数部分，小数点，小数部分和指数部分构成。整数部分和小数部分用小数点链接；指数部分由 `e` / `E` 字符后接一个有符号指数构成。整数部分和小数部分可以省略其一；小数点和指数部分可以省略其一。

```
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
```

虚数字面值

虚数字面值是一个小数，相当于复数常量中的虚数部分。它由浮点数或者整数后接小写字母 i 构成。

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

```
0i
011i  // == 11i
0.i
2.71828i
1.e+0i
6.67428e-11i
1E6i
.25i
.12345E+5i
```

Rune 字面值

rune 类型字面值相当于一个 rune 常量。它是一个表示 Unicode 代码点的整数。rune 类型字面值表示为用单引号包裹的一个或多个字符，像 'x' 或 '\n'。在单引号中除了换行符和未转义的单引号其他的字符都可以直接显示。单引号包裹的字符的值和字符在 Unicode 编码中的值相等，而以反斜线开头的多字符序列会把值翻译成多种格式。

使用引号表示单字符是最简单的方式；因为 Go 的源文本是 UTF-8 编码，一个整数可能代表多个 UTF-8 字节。例如， 'a' 可以使用单字节表示字符 a，Unicode 编码 U+0061，值 0x61，而 'ä' 是两字节表示分音符的 a，Unicode 编码 U+00E4，值 0xe4。

反斜线能将任意值编码成 ASCII 文本。有四种方式将整数值表示为数字常量：`\x` 后接两个十六进制数；`\u` 后接四个十六进制数；`\U` 后接八个十六进制数。 `\` 后接三个八进制数。每种情况下都使用相应进制来表示字面量的整数值。

虽然这四种方式都以整数表示，但它们的有效区间并不相同。八进制只能表示 0 - 255 以内的整数。十六进制满可以满足需求。`\u` 和 `\U` 都可以表示 Unicode 代码点，不过其中的一些值是无效的，特别是 0x10FFFF 以上的值。

反斜线结合以下字符具有特殊含义：

```
\a   U+0007 alert or bell
\b   U+0008 退格符
\f   U+000C form feed
\n   U+000A line feed or newline
\r   U+000D carriage return
\t   U+0009 水平制表符
\v   U+000b 垂直制表符
\\   U+005c 反斜线
\'   U+0027 单引号  (只在 rune 字面值中有效)
\"   U+0022 双引号  (只在字符串字面值中有效)
```

其他所有以反斜线开头的序列在 rune 的规则中都是非法的。

```
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" | "b" | "f" | "n" | "r" | "t" | "v" | `\` | "'" |
`"` ) .
```

```
'a'
'ä'
'本'
'\t'
'\000'
'\007'
'\377'
'\x07'
'\xff'
'\u12e4'
'\U00101234'
'\''         // 包含单引号的 rune 字面值
'aa'         // 无效: 太多字符
'\xa'        // 无效: 缺少十六进制数
'\0'         // 无效: 缺少八进制数
```

```
'\uDFFF'     // 无效: surrogate half
'\U00110000' // 无效: 非法的 Unicode 代码点
```

字符串字面量

字符串字面量表示从字符序列中获取的字符串常量。它有两种格式：原始字符串字面量和解释型字符串字面量。

原始字符串是由反引号包裹（`foo`）。字符串中除反引号以外的其他字符都会显示出来。原生字符串由反引号之间的（默认 UTF-8 编码）的字符组成。它的值为引号内未经解释（默认 UTF-8 编码）所有字符；尤其是，反斜线再字符串中没有特殊意义并且字符串中保留换行符。在原始字符串的值中会丢弃回车键返回 '\r' 字符。

解释型字符串由双引号之间的字符组成（"bar"）。除了换行符和双引号其他字符都会显示出来。双引号之间的文本组成字面量的值。反斜线的转义规则与 rune 字面量基本相同（不同的是 \' 非法，而 " 合法）。三位八进制数（\nnn）和两位十六进制数（\xnn）换码符的值表示相应字符串的字节。其他的换码符都表示字符各自的 UTF-8 编码（可能是多字节）。因此字符串 \377 和 \xFF 都表示值为 0xFF=255 的单个字节，而 `ÿ`, `\u00FF`, `\U000000FF` 和 `\xc3\xbf` 表示 UTF-8 编码字符 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`                // 等价于 "abc"
`\n
\n`                  // 等价于 "\\n\n\\n"
"\n"
"\""                 // 等价于 `"`
"Hello, world!\n"
"日本語"
"\u65e5本\U00008a9e"
"\xff\u00FF"
"\uD800"             // 无效: surrogate half
"\U00110000"         // 无效: 无效的 Unicode 代码点
```

这些例子都表示相同的字符串：

```
"日本語"                                 // UTF-8 文本
`日本語`                                 // UTF-8 文本作为原生字面值
"\u65e5\u672c\u8a9e"                     // 确定的 Unicode 代码点
"\U000065e5\U0000672c\U00008a9e"         // 确定的 Unicode 代码点
"\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e"   // 确定的 UTF-8 字节
```

如果源代码中使用两个代码点表示一个字符，例如带音标的字母，把它放在 rune 中会报错（它不是单代码点）。并且在字符串中会显示两个代码点。

更多Golang资料包：https://github.com/0voice/Introduction-to-Golang

常量

常量分为：布尔型，rune型，整型，浮点型，复数型，字符串型。其中 rune，整型，浮点型，复数型统称为数字常量。

常量的值可以表示为一个 rune字面量，整数字面量，浮点数字面量，虚数字面量，字符串字面量，表示常量的标识符，常量表达式，一个转换结果为常量的类型转换，和一些返回值为常量的内置函数(接受任何值的 `unsafe.Sizeof`，接受部分表达式的 `cap` 或 `len`，接受虚数常量的 `real` 和 `imag`，接受数字常量的 `complex`)。布尔类型的值为预定义常量 `true` 或 `false`，预定义的标识符 `iota` 表示一个整型常量。

一般情况下复数常量是常量表达式的一种形式。会在常量表达式章节详细讨论。

数字常量可以表示任意精度的确定值而且不会溢出。因此，没有常量可以表示非 0，无穷大和非数字值。

常量可以指定类型也可以不指定类型。字面值常量，`true`，`false`，`iota`，和只包含无类型常量操作的常量表达式是无类型的。

常量可以通过常量声明和转换时显式的指定具体类型，也可以隐式的在变量声明、赋值或作为表达式操作元时隐式的指定具体类型。如果常量的值和他的类型不匹配，会报错。

无类型常量由一个默认的类型，这个类型会根据使用常量时的上下文进行隐式转换。例如：短变量声明 `i := 0` 没有指定 i 的类型。无类型常量的默认类型可以是：`bool`，`rune`，`int`，`float64`，`complex128` 或者 `string`，具体选择哪种类型由常量的值决定。

实现限制：虽然数字常量在 Go 中是任意精度，不过编译器在实现时会在内部限制精度。这意味着每个编译器实现都要：

- 至少保证整形常量有 256 位
- 浮点数常量（包括复数常量）都要保证至少 256 位的主体部分和至少 16 位的有符号指数部分
- 如果不能表示给定整数的精度抛出错误
- 如果浮点数或复数溢出抛出错误
- 如果由于精度限制不能表示浮点数或者复数进行舍入

这些要求同时作用于字面量常量额和常量表达式的结果。

变量

变量是一个用来储存值的位置。根据不同的变量类型，可以保存不同的值。

变量声明，函数参数和返回值，声明的函数签名，函数字面值都会为命名变量预留储存空间。调用内置的 `new` 函数或获取复合字面值的地址都会在运行时为变量分配存储空间。这种匿名变量是通过（可能是隐式的）指针间接引用的。

像数组，切片和结构体类型的变量，它们内部都包含很多元素或字段，而且这些元素和字段都可以直接被访问。数组和切片中的每个元素的行为和单独的变量基本相同。

变量的静态类型可以通过变量声明、提供给 `new` 的类型、复合字面值、结构体变量声明的元素类型以上几种方式确定。通过new或者类型初始化。接口类型的变量也有一个明确的动态类型，这个动态类型是在运行时赋值给变量的具体值类型（特例：预声明的 nil 是无类型的）。动态类型在程序的执行过程中可能并不相同，但是接口变量的值是可以分配给相同静态类型的变量。

```
var x interface{}  // x 的静态类型为 interface{} 值为 nil
var v *T           // v 的静态类型为 *T 值为 nil
x = 42             // x 的动态类型为 int 值为 42
x = v              // x 动态类型为 *T 值为 (*T)(nil)
```

在表达式中使用变量可以取出变量的值；这个值就是变量最近一次被赋予的值。如果没有对变量赋过值，那么他的值是该类型的零值。

类型

类型是一个集合，集合包括值和针对值的操作&方法。一个类型可以使用类型名来表示。类型有多种表现形式：如果存在类型名，可以使用类型名表示，或者也可以使用根据已有类型组合成的类型字面值。

```
Type      = TypeName | TypeLit | "(" Type ")" .
TypeName  = identifier | QualifiedIdent .
TypeLit   = ArrayType | StructType | PointerType | FunctionType | InterfaceType
|
        SliceType | MapType | ChannelType .
```

Go 已经预先声明了某些类型的名称。并引入了类型声明。复合类型（数组、结构体、指针、函数、接口、切片、map、channel）可以使用他们的类型字面值。

每个类型T都有一个底层类型。如果T是预定义类型或者类型字面值。那么底层类型就是他自身。否则，T的底层类型就是它再类型声明时引用到的类型。

```
type (
    A1 = string
    A2 = A1
)

type (
    B1 string
    B2 B1
    B3 []B1
    B4 B3
)
```

`string`，`A1`，`A2`，`B1`，`B2` 的底层类型是 `string`。`[]B1`，`B3`，`B4` 的下游类型是[]B1。

方法集

类型可能会有一个与之关联的方法集。接口类型的方法集就可以使用自身表示。对于其他类型，类型 T 的方法集由所有接收者类型为 T 的方法组成。而对应指针类型 *T 的方法集由所有接收者类型为 T 或 *T 的方法组成。如果是结构体类型且含有嵌入字段，那么方法集中可能还会包含更多的方法，具体请看结构体类型章节。其他类型的方法集都为空。方法集中的每个方法都有唯一且不为空的方法名。

类型的方法集用来确定类型实现的接口和以类型作为接收者能够调用的方法。

布尔类型

布尔类型表示预定义常量 `true` 和 `false` 表示布尔真实值的集合。预定义的布尔类型为 `bool`；它是通过类型声明创建的。

数字类型

一个数字类型相当于整型和浮点型的所有值的集合。预定义的数字类型包括：

```
uint8       8 位无符号整数集合 (0 to 255)
uint16      16 位无符号整数集合 (0 to 65535)
uint32      32 位无符号整数集合 (0 to 4294967295)
uint64      64 位无符号整数集合 (0 to 18446744073709551615)

int8        8 位有符号整数集合 (-128 to 127)
int16       16 位有符号整数集合 (-32768 to 32767)
int32       32 位有符号整数集合 (-2147483648 to 2147483647)
int64       64 位有符号整数集合 (-9223372036854775808 to 9223372036854775807)

float32     IEEE-754 32 位浮点数集合
float64     IEEE-754 64 位浮点数集合

complex64   实部虚部都为 float32 的复数集合
complex128  实部虚部都为 float64 的复数集合

byte        uint8 的别名
rune        int32 的别名
```

n 位整数的值具有 n 比特的宽度并用补码表示。

以下几种预定义类型由具体平台实现指定长度：

```
uint     32 或 64 位
int      和 uint 位数相同
uintptr  能够容纳指针值的无符号整数
```

为了避免移植性问题，除了被 uint8 的别名 byte 和 int32 的别名 rune，其他所有的数字类型都是通过类型声明定义。当在表达式中使用不同的数字类型需要进行类型转换。例如：int32 和 int 不是相同的类型，即使他们在指定的平台上是相等的。

字符串类型

字符串类型表示字符串的值类型。字符串的值是一个字节序列（有可能为空）。字符串一旦创建就无法修改它的值。预定义的字符串类型是 `string`，它是通过类型声明定义的。

可以使用内置函数 `len` 获取字符串长度。如果字符串是常量那么它的长度在编译时也为常量。可以通过数字下标 0 ~ len(s)-1 访问字符串字节。获取字符串的地址是非法操作；如果 `s[i]` 是字符串的第 i 个字节，那么 `&s[i]` 是无效的。

数组类型

数组是一定数量的单一类型元素序列，而这个单一类型叫做元素类型。元素的个数表示元素的长度，它永远不是负数。

```
ArrayType   = "[" ArrayLength "]" ElementType .
ArrayLength = Expression .
ElementType = Type .
```

长度是数组类型的一部分；它是一个类型为 int 的非负常量。可以用内置函数 `len` 获取数组的长度。元素可以通过下标 `0~len(a)-1` 访问。数组一般都是一维的，不过也可以是多维的。

```
[32]byte
[2*N] struct { x, y int32 }
[1000]*float64
[3][5]int
[2][2][2]float64  // same as [2]([2]([2]float64))
```

切片类型

切片描述了底层数组的一个连续片段并提供对连续片段内元素的访问。切片类型表示元素类型的数组的所有切片的集合。没有被初始化的切片用 nil 表示。

```
SliceType = "[" "]" ElementType .
```

与数组一样，切片的可以使用索引访问并且有长度，切片的长度可以通过内置的 `len` 函数获取；与数组不同的是它的长度在运行时是可以变化的。我们可以通过下标 `0~len(s)-1` 来访问切片内的元素。切片的索引可能会小于相同元素再底层数组的索引。

切片一旦初始化，那么就有一个与之对应的底层数组保存切片中的元素。切片和底层的数组还有其他指向该数组的切片共享相同的储存空间；而不同的数组总是有着不同的存储空间。

切片的底层数组可能会延伸到切片末尾以外，切片的容积等于切片现在的长度加上数组中切片还没使用的长度；可以从原始切片中切出一个长度与容量相等的切片。切片的容量可以通过内置的 `cap(a)` 函数来获取。可以通过函数`make`来创建一个T类型的新切片。

使用内置函数 `make` 可以出实话给定元素类型 T 的切片。`make` 函数接收三个参数：切片类型、切片长度、切片容积，其中切片容积是可选参数。`make` 创建的切片会在底层分配一个切片所引用的新数组。

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

`make` 的作用就是创建新数组并切分它，所以下面两种写法是等价的：

```
make([]int, 50, 100)
new([100]int)[0:50]
```

与数组相同，切片一般是一维的，不过也可以复合成多维。数组中的数组都必须是相同的长度，但是切片中的切片长度是动态变化的，不过切片中的切片需要单独初始化。

结构体类型

结构体是一个命名元素序列，命名元素也叫做字段，每个字段都对应一个名称和类型，字段的名字可以是显式指定的（标识符列表）也可以是隐式的（嵌入字段）。在结构体中非空字段具有唯一性。

```
StructType    = "struct" "{" { FieldDecl ";" } "}" .
FieldDecl     = (IdentifierList Type | EmbeddedField) [ Tag ] .
EmbeddedField = [ "*" ] TypeName .
Tag           = string_lit .
```

```
// 空结构体.
struct {}

// 6个字段的结构体.
struct {
    x, y int
    u float32
    _ float32  // padding
    A *[]int
    F func()
}
```

一个指定了类型而没有指定名称的字段叫做嵌入字段，嵌入字段必须指定类型名 T 或指向非接口类型的指针类型 *T，其中 T 不能为指针类型。或者一个非接口类型的指针。并且T本身不能为指针类型。这种情况下会把类型名作为字段的名字。

```
// 一个包含 4 个嵌入字段 T1, *T2, P.T3 和 *P.T4 的结构体
struct {
    T1        // 字段名为 T1
    *T2       // 字段名为 T2
    P.T3      // 字段名为 T3
    *P.T4     // 字段名为 T4
    x, y int  // 字段名为 x 和 y
}
```

以下声明是错误的因为字段名称必须唯一。

```
struct {
    T     // 嵌入字段 *T 与 *P.T 冲突
    *T    // 嵌入字段 T 与 *P.T 冲突
    *P.T  // 嵌入字段 T 与 *T 冲突
}
```

如果 `x.f` 是表示该字段或方法 `f` 的合法选择器，则会调用结构 `x` 中嵌入字段的字段或方法 `f`。

从嵌入字段组合来的字段与结构体原来的字段行为基本相同，只是不能在结构体的复合字面值中直接使用。

给定一个结构体 S 和一个类型 T，依据以下规则生成组合后的方法集：

- 如果 S 包含嵌入字段 T，则 S 和 *S 的方法集包括接收者为 T 的方法集，而 *S 包括 接收者为 *T 的方法集。

- 如果 S 包含字段 *T*。那么*S*和S均包含接收者为 T 和 *T 的所有方法集。

声明字段时可以给该字段添加一个字符串的 tag。这个 tag 将会成为它所对应字段的一个属性。空 tag 和缺省 tag 是相同的。tag 的值可以通过反射的接口获取，可以作为类型结构体的类型定义的一部分，也可以忽略。

```
struct {
    x, y float64 ""  // 空 tag 和缺省 tag 相同
    name string  "any string is permitted as a tag"
    _    [4]byte "ceci n'est pas un champ de structure"
}

// 结构体对应一个 TimeStamp 的 protocol buffer.
// tag 字符串中定义了 protocol buffer 字段对应的数字;
// 一般使用 reflect 包读取他们.
struct {
    microsec  uint64 `protobuf:"1"`
    serverIP6 uint64 `protobuf:"2"`
}
```

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。这个指定的类型叫做指针的基础类型。没有初始化的指针值为nil。

```
PointerType = "*" BaseType .
BaseType    = Type .
```

```
*Point
*[4]int
```

函数类型

函数类型可以表示所有具有相同参数类型和返回值类型的函数。未初始化的函数类型值为 nil。

```
FunctionType   = "func" Signature .
Signature      = Parameters [ Result ] .
Result         = Parameters | Type .
Parameters     = "(" [ ParameterList [ "," ] ] ")" .
ParameterList  = ParameterDecl { "," ParameterDecl } .
ParameterDecl  = [ IdentifierList ] [ "..." ] Type .
```

在参数和返回值列表中，标识符列表必须同时存在或缺省。如果存在，那么每个名字都表示指定类型的一个参数/返回值，这些标识符必须非空并且不能重复。如果缺省，指定类型的参数/返回值使用对应的类型表示。参数列表和返回值列表一般都是需要加括号，不过在只有一个缺省返回值时，它可以不使用括号。

函数的最后一个参数可以添加前缀 `...`。包含这种参数的函数叫做变参函数，它可以接收零个或多个参数。

```
func()
func(x int) int
func(a, _ int, z float32) bool
func(a, b int, z float32) (bool)
func(prefix string, values ...int)
func(a, b int, z float64, opt ...interface{}) (success bool)
func(int, int, float64) (float64, *[]int)
func(n int) func(p *T)
```

接口类型

接口类型指定了一个方法集。一个接口类型变量可以保存任何方法集是该接口超集的类型。我们可以认为类型实现了接口。没有初始化的接口类型值为 nil。

```
InterfaceType      = "interface" "{" { MethodSpec ";" } "}" .
MethodSpec         = MethodName Signature | InterfaceTypeName .
MethodName         = identifier .
InterfaceTypeName  = TypeName .
```

在接口类型的方法集中，每个方法的名称必须是非空且唯一。

```
// A simple File interface
interface {
    Read(b Buffer) bool
    Write(b Buffer) bool
    Close()
}
```

接口可以由多个类型实现，例如：类型 `S1` 和类型 `S2` 都有以下方法集：

```
func (p T) Read(b Buffer) bool { return … }
func (p T) Write(b Buffer) bool { return … }
func (p T) Close() { … }
```

（这里的类型 T 可以表示 `S1` 也可以表示 `S2` ） `S1` 和 `S2` 都实现了接口 `File`，而不用管类型是否还有其他方法。

一个类型实现了任何方法集的为其子集的接口。因此它可能实现了多个不同接口。例如：所有的类型都实现了空接口：

```
interface{}
```

与之相似，思考下面这个定义为 `Locker` 的接口：

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

如果 `S1` 和 `S2` 也实现了它：

```
func (p T) Lock() { … }
func (p T) Unlock() { … }
```

那它们就实现了两个接口 `Locker` 和 `File`。

一个接口 T 可以使用另一个接口 E 来指定方法。这种方式叫做将接口 E 嵌入进接口 T。它把 E 中所有的方法（包括导出和未导出的方法）全部添加进接口 T。

```
type ReadWriter interface {
    Read(b Buffer) bool
    Write(b Buffer) bool
}

type File interface {
    ReadWriter  // 与添加 ReadWriter 接口中的方法是等价的
    Locker      // 与添加 Locker 接口中的方法是等价的
    Close()
}

type LockedFile interface {
    Locker
    File        // 无效: Lock, Unlock 不是唯一的
    Lock()      // 无效: Lock 不是唯一的
}
```

接口 T 不能递归的嵌入进自己或已经嵌入过它的接口。

```
// 无效: Bad 不能嵌入它自己
type Bad interface {
    Bad
}

// 无效: Bad1 不能嵌入已经引用它的 Bad2
type Bad1 interface {
    Bad2
}
type Bad2 interface {
    Bad1
}
```

Map类型

map 类型是一种以唯一值作为键的无序集合。

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

map的键类型必须能使用比较运算符 `==` 和 `!=` 进行比较。因此它的键类型不能是函数，map，或者切片。如果键是接口类型，那么比较运算符必须能比较他的动态值。如果不能会抛出一个运行时错误。

```
map[string]int
map[*T]struct{ x, y float64 }
map[string]interface{}
```

map中元素的个数叫做它的长度。对于一个map `m`。它的长度可以通过内置函数 `len` 获得，而且它的长度可能再运行时发生变化。map 可以再运行时添加和取回元素，页可以使用内置函数 `delete` 移除元素。

可以使用内置函数 `make` 初始化一个新的且为空的 map。它能指定 map 的类型和预留的空间：

```
make(map[string]int)
make(map[string]int, 100)
```

map 的预留空间不会固定住 map 的长度；它可以通过添加一定数量的元素来增加自己的长度（nil map 不能添加元素）。nil map 和空 map 是相等的，只是 nil map 不能添加元素。

Channel类型

channel提供一种手段在并发执行的函数间发送和接收指定类型的值。没有初始化的 channel 是nil。

```
ChannelType = ( "chan" | "chan" "<-" | "<-" "chan" ) ElementType .
```

操作符 `<-` 可以指定 channel 的数据流动方向。如果没有指定方向，channel 默认是双向的。channel 可以通过转换和赋值来限制只读和只写。

```
chan T          // 可以接收和发送 T 类型的数据
chan<- float64  // 只能发送 float64 类型的值
<-chan int      // 只能接收
```

`<-` 与最左侧的 `chan` 关联：

```
chan<- chan int    // 等价于 chan<- (chan int)
chan<- <-chan int  // 等价于 chan<- (<-chan int)
<-chan <-chan int  // 等价于 <-chan (<-chan int)
chan (<-chan int)
```

可以通过内置的 `make` 函数初始化 channel。`make` 函数可以指定channel的类型和容量。

```
make(chan int, 100)
```

容量是设置了最大能缓存元素的数量。如果没有设置容量或值为 0，channel 就是没有缓存的，这时只有当发送者和接收者都准备好后才会传输数据。而带缓存的 channel 在缓存没有满的时候依然可以成功发送数据，当缓存不为空的时候可以成功接收到数据，值为 nil 的 channel 不能传输数据。

可以通过内置函数 `close` 关闭 channel。在接收端的第二个返回值可以用来提示接收者在关闭的 channel 是否还包含数据。

channel 可以在发送语句，接收操作中使用。可以不考虑同步性直接在多个 goroutine 中对 channel 调用内置函数 `len` 和 `cap` 。channel 的行为和 FIFO 队列相同。举个例子，一个 goruntine 发送数据，另一个 goruntine 接收他们，接收数据的顺序和发送数据的顺序是相同的。

类型的属性和值

类型标识

两个类型可能相同也可能不同。

定义的类型都是不同类型。如果两个类型的底层类型在结构上是相同的，那它们也是相等的。总的来说：

- 2 个数组的长度和元素类型相同，那么它们就是相同类型。
- 如果两个切片的元素类型相同那么它们就是相同类型。
- 如果两个结构体字段顺序相同，并且字段名称、字段类型和 tag 都相同那么它们就是相等的。非导出字段的字段名在不同的包中总是不同的。
- 如果两个指针的基础类型相同那么他们具有相同类型。

- 如果两个函数具有相同的参数和返回值列表，并且他们的类型相同那么他们就是相同的，参数的名称不一定要相同。
- 如果两个接口的方法集完全相同（方法的顺序）。
- 如果两个 map 类型的键类型和值类型相同那它们就是相等的。
- 如果两个 channel 类型包含的对象类型和 channel 的方向都是相同的那它们就是相同的。

给出下列声明：

```
type (
    A0 = []string
    A1 = A0
    A2 = struct{ a, b int }
    A3 = int
    A4 = func(A3, float64) *A0
    A5 = func(x int, _ float64) *[]string
)

type (
    B0 A0
    B1 []string
    B2 struct{ a, b int }
    B3 struct{ a, c int }
    B4 func(int, float64) *B0
    B5 func(x int, y float64) *A1
)

type    C0 = B0
```

这些类型是相等的：

```
A0, A1, and []string
A2 and struct{ a, b int }
A3 and int
A4, func(int, float64) *[]string, and A5

B0, B0, and C0
[]int and []int
struct{ a, b *T5 } and struct{ a, b *T5 }
func(x int, y float64) *[]string, func(int, float64) (result *[]string), and A5
```

B0 和 B1 不是一种类型因为它们是通过类型定义方式分别定义的；`func(int, float64) *B0` 和 `func(x int, y float64) *[]string` 是不同的，因为 B0 和 []string 不是相同类型。

可分配性

在以下情况下，可以将 x 分配给类型为 T 的变量（把 x 分配给 T）：

- x 的类型为 T
- x 的类型 V 和 T 有相同的底层类型并且类型 T 或 V 至少一个定义的类型
- T 是一个接口类型并且 x 实现了 T
- x 是一个 channel，并且 T 是channel类型，类型V和类型T有相同的元素类型，并且 2 种类型至少有一种不是定义的类型
- x 等于 nil 并且 T 是一个指针，函数，切片，map，channel 或接口类型
- x 是一个可以表示 T 类型值的无类型常量

代表性

满足以下条件时可以用 T 类型的值表示常量 x：

- T 值的集合包括 x
- T 是浮点型，而 x 在没有溢出的情况下能够近似成 T 类型。近似规则使用 `IEEE 754 round-to-even`，负零和无符号的零相同。需要注意的是，常量的值不会为负零，NaN，或无限值。
- T 为复数类型，并且 x 的 `real(x)` 和 `imag(x)` 部分由复数类型对应的浮点类型（`float32` 或 `float64`）组成。

```
x                   T           x 可以表示 T 的值，因为：

'a'                 byte        97 在 byte 类型值的集合中
97                  rune        rune 是 int32 的别名，97 在 32 位整型值的集合中
"foo"               string      "foo" 在字符串值的集合中
1024                int16       1024 在 16 位整型值的集合中
42.0                byte        42 在 8 位无符号整型值的集合中
1e10                uint64      10000000000 在 64 位无符号整型值的集合中
2.718281828459045   float32     2.718281828459045 的近似值 2.7182817 在 float32 类型值的集合中
-1e-1000            float64     -1e-1000 的近视值 IEEE -0.0，等于 0
0i                  int         0 是整型值
(42 + 0i)           float32     42.0 (0 虚部) 在 float32 类型值的集合中
```

```
x                   T           x 不能表示 T 的值，因为：

0                   bool        0 不在布尔值的集合中
'a'                 string      'a' 是 rune 类型，它不在字符串类型的值集合中
1024                byte        1024 不在 8 位无符号整型值的集合中
-1                  uint16      -1 不在 16 位无符号整型值的集合中
1.1                 int         1.1 不是整型值
42i                 float32     (0 + 42i) 不在 float32 类型值的集合中
1e1000              float64     1e1000 取近似值时会溢出成 IEEE
```

代码块

代码块是用大括号括起来的声明和语句。

```
Block = "{" StatementList "}" .
StatementList = { Statement ";" } .
```

除了源码中显式的代码块，也有一些隐式的代码块。

- 包含所有的Go代码的全局代码块。
- 包含所有包的代码的包代码块。
- 包含文件内的所有代码的文件代码块。
- 每个 if，switch和 for 的范围都会形成隐式的块。
- 每个 switch 和 select 条件都有自己的代码块。

代码块可以嵌套并且影响作用域。

声明和作用域

一段声明可以给常量，类型，变量，函数，标签，和包绑定标识符。程序中每个标识符都需要声明。相同标识符不能在同一个代码块中声明2次。并且相同标识符不能同时在文件和 package 代码块中声明。

空标识符可以和其他标识符一样在声明中使用。不过它不绑定标识符，等于没有声明。在 package 代码块中 `init` 标识符只能用做 `init` 函数的标识符，就像空标识符一样，它不会引入新的绑定。

```
Declaration   = ConstDecl | TypeDecl | VarDecl .
TopLevelDecl  = Declaration | FunctionDecl | MethodDecl .
```

声明过的标识符的作用域就是声明标识符所在的作用域。

go使用块来规定词汇的方位：

- 预定义的标识符具有全局作用域。
- 所有定义的顶级标识符具有包作用域。
- import进来的包的名字标识符具有文件作用域。
- 方法的接收者，函数参数，返回值变量具有函数作用域。
- 函数内定义的参量和变量标识符的作用域是标识符被声明到容纳他的块结束。

一个代码块中声明的标识符可以在它内部的代码块中重新声明。在内部代码块的作用域中标识符表示在内部代码块中声明的实体。

pakcage 语句不属于声明。包名不会出现在任何的作用域中。它的作用只是用来标识属于相同包的多个文件并在导入时指定默认包名。

标签的作用域

可以使用标签语句来声明标签，并且可以在 `break`，`continue`，`goto` 语法中使用。如果只声明但没有使用标签时非法的。标签的作用域只有定义时的函数体，早递归函数体中没有作用。

空标识符

空标识符使用下划线 `_` 代表。与一般的非空标识符不同，它作为匿名标识符在声明，运算元和赋值语句中都有特殊含义。

预定义的标识符

以下标识符已经在全局作用域中预先声明：

```
Types:
    bool byte complex64 complex128 error float32 float64
    int int8 int16 int32 int64 rune string
```

```
    uint uint8 uint16 uint32 uint64 uintptr

Constants:
    true false iota

Zero value:
    nil

Functions:
    append cap close complex copy delete imag len
    make new panic print println real recover
```

导出标识符

标识符可以导出供其他包使用。在以下两种情况同时满足时标识符是导出的：

- 标识符的首字母是大写（Unicode 的 `Lu` 类）
- 标识符声明在包作用域或者它是字段名/方法名。

其他任何标识符都不是导出的。

标识符的唯一性

给定一个标识符集合，一个标识符与集合中的每个标识符都不相同，那就认为这个标识符是唯一的。假设有两个标识符，如果它们的拼写不同，或者它们在不同的包中并没有导出，那它们就是不同标识符。相反，其他情况下都认为标识符是相同的。

常量声明

常量声明使用常量表达式绑定一系列标识符。标识符的数量必须等于表达式的数量。左侧第 n 个标识符绑定右侧第 n 个表达式的值。

```
ConstDecl      = "const" ( ConstSpec | "(" { ConstSpec ";" } ")" ) .
ConstSpec      = IdentifierList [ [ Type ] "=" ExpressionList ] .

IdentifierList = identifier { "," identifier } .
ExpressionList = Expression { "," Expression } .
```

如果给定类型，常量会指定类型，并且表达式的值必须能对这个类型进行赋值。

如果没有给定类型。常量会转换成相应的表达式类型。如果表达式的值是无类型常量，那么声明的常量也是无类型的，并且常量的标识符代表常量的值。例如：即使小数部分是 0，只要表达式是浮点数字面值，常量标识符也表示为浮点数常量。

```
const Pi float64 = 3.14159265358979323846
const zero = 0.0         // 无类型浮点数常量
const (
    size int64 = 1024
    eof        = -1  // 无类型整型常量
)
const a, b, c = 3, 4, "foo"  // a = 3, b = 4, c = "foo", 无类型整型和字符串常量
const u, v float32 = 0, 3    // u = 0.0, v = 3.0
```

括号内的常量声明列表的表达式除了第一个必须声明其他表达式可以不写。空的表达式列表的值和类型都和前面的非空表达式相同。缺省的表达式列表等价于重复之前的表达式。标识符的数量必须等于表达式的数量。`iota`常量生成器是一个可以快速生成序列值的机制。

```
const (
    Sunday = iota
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Partyday
    numberOfDays  // 非导出常量
)
```

Iota

在常量声明中，预定义的标识符`iota`表示连续的无类型整型常量。它的值为常量声明中每个常量定义的位置（从零开始）。它能够用来生成一个关联常量集合：

```
const ( // iota is reset to 0
    c0 = iota  // c0 == 0
    c1 = iota  // c1 == 1
    c2 = iota  // c2 == 2
)

const ( // iota is reset to 0
    a = 1 << iota  // a == 1
    b = 1 << iota  // b == 2
    c = 3          // c == 3  (没有使用 iota 不过它的值依然递增)
    d = 1 << iota  // d == 8
)
```

```
const ( // iota is reset to 0
    u         = iota * 42  // u == 0     (无类型整型常量)
    v float64 = iota * 42  // v == 42.0  (float64 类型常量)
    w         = iota * 42  // w == 84    (无类型整型常量)
)

const x = iota  // x == 0  (iota 被重置)
const y = iota  // y == 0  (iota 被重置)
```

根据定义，在同一个常量定义中多次使用 `iota` 会得到相同的值：

```
const (
    bit0, mask0 = 1 << iota, 1<<iota - 1  // bit0 == 1, mask0 == 0  (iota == 0)
    bit1, mask1                           // bit1 == 2, mask1 == 1  (iota == 1)
    _, _                                  //                        (iota == 2,
unused)
    bit3, mask3                           // bit3 == 8, mask3 == 7  (iota == 3)
)
```

最后一个例子利用了最后一个非空表达式列表的隐式重复。

类型声明

类型声明为类型绑定一个标识符。类型声明有2种方式：类型声明和别名声明。

```
TypeDecl = "type" ( TypeSpec | "(" { TypeSpec ";" } ")" ) .
TypeSpec = AliasDecl | TypeDef .
```

Alias声明

别名声明给指定类型绑定一个标识符名称。

```
AliasDecl = identifier "=" Type .
```

在标识符作用域内，它作为类型的别名。

```
type (
    nodeList = []*Node  // nodeList 和 []*Node 是相同类型
    Polar    = polar    // Polar 和 polar 表示相同类型
)
```

Type 定义

类型定义会创建一个新类型并绑定一个标识符，新类型与给定类型具有相同的底层类型和操作。

```
TypeDef = identifier Type .
```

这个类型叫做定义类型，它和其他所有类型都不相同，包括创建它的类型。

```
type (
    Point struct{ x, y float64 }  // Point 和 struct{ x, y float64 } 是不同类型
    polar Point                   // polar 和 Point 表示不同类型
)

type TreeNode struct {
    left, right *TreeNode
    value *Comparable
}

type Block interface {
    BlockSize() int
    Encrypt(src, dst []byte)
    Decrypt(src, dst []byte)
}
```

定义类型可以关联该类型的方法。它不会继承原来类型的任何方法。但是接口类型的方法集和类型的结构没有改变。

```
// Mutex 是一个拥有 Lock 和 Unlock 两个方法的数据类型。
type Mutex struct         { /* Mutex fields */ }
func (m *Mutex) Lock()    { /* Lock implementation */ }
func (m *Mutex) Unlock()  { /* Unlock implementation */ }

// NewMutex 与 Mutex 结构相同不过方法集为空。
type NewMutex Mutex

// PtrMutex 的底层类型 *Mutex 的方法集没有改变，
// 但是 PtrMutex 的方法集为空。
type PtrMutex *Mutex
```

```
// *PrintableMutex 包含嵌入字段 Mutex 的 Lock 和 Unlock 方法。
type PrintableMutex struct {
    Mutex
}

// MyBlock 是与 Block 有相同方法集的接口类型
type MyBlock Block
```

类型定义可以定义方法集不同的布尔值、数字和字符串类型：

```
type TimeZone int

const (
    EST TimeZone = -(5 + iota)
    CST
    MST
    PST
)

func (tz TimeZone) String() string {
    return fmt.Sprintf("GMT%+dh", tz)
}
```

变量声明

变量声明可以创建一个或多个变量，并绑定对应的标识符、指定类型和初始值。

```
VarDecl     = "var" ( VarSpec | "(" { VarSpec ";" } ")" ) .
VarSpec     = IdentifierList ( Type [ "=" ExpressionList ] | "=" ExpressionList
) .
```

```
var i int
var U, V, W float64
var k = 0
var x, y float32 = -1, -2
var (
    i       int
    u, v, s = 2.0, 3.0, "bar"
)
var re, im = complexSqrt(-1)
var _, found = entries[name]  // map lookup; only interested in "found"
```

如果给定一个表达式列表。变量会根据赋值规则使用表达式进行初始化。否则，每个变量都会初始化成变量类型的零值。

如果指定类型，变量会为指定类型。如果没有指定类型，变量会使用分配的初始值类型。如果初始值为无类型常量，它会转换成初始值的默认类型。如果是一个无类型布尔值，那么变量的类型就是 `bool`。值 `nil` 不能给没有指定类型的变量赋值。

```
var d = math.Sin(0.5)  // d is float64
var i = 42             // i is int
var t, ok = x.(T)      // t is T, ok is bool
var n = nil            // illegal
```

实现的限制：在函数体内声明的变量如果没有使用过编译器需要报错。

短变量声明

短变量声明的语法:

```
ShortVarDecl = IdentifierList ":=" ExpressionList .
```

它比正常使用初始化表达式进行变量声明的方式要短，而且不指定类型：

```
"var" IdentifierList = ExpressionList .
```

```
i, j := 0, 10
f := func() int { return 7 }
ch := make(chan int)
r, w := os.Pipe(fd)  // os.Pipe() 返回两个值
_, y, _ := coord(p)  // coord() 返回三个值，我们只关注 y
```

和常规变量声明不同，即使之前在相同代码块中声明过的变量，也可以在短变量重新声明相同类型的变量，并且保证至少会有一个新的非空变量。总之，只应该在多变量短声明的时候重新声明变量，重新声明并不会使用新的变量，而是给变量分配新值。

```
field1, offset := nextField(str, 0)
field2, offset := nextField(str, offset)  // 重新声明 offset
a, a := 1, 2                              // 非法：声明了 a 两次并且没有新的变量
```

短变量声明只能在函数中使用，例如在`if`、`for`、`switch`语句的上下文中声明临时变量。

函数声明

函数声明为函数绑定标识符。

```
FunctionDecl = "func" FunctionName Signature [ FunctionBody ] .
FunctionName = identifier .
FunctionBody = Block .
```

如果函数指定了返回参数。函数体的语句必须以终止语句结束。

```
func IndexRune(s string, r rune) int {
   for i, c := range s {
    if c == r {
        return i
    }
   }
   // 无效：缺少 return 语句
}
```

函数声明可以没有函数体。这样的声明提供一个函数声明，并由其他外部实现，例如汇编脚本。

```
func min(x int, y int) int {
    if x < y {
        return x
    }
    return y
}

func flushICache(begin, end uintptr)  // 由外部实现
```

方法声明

方法是一个带接收者的函数，方法声明为方法绑定标识符作为方法名并指定方法对应的接收者类型。

```
MethodDecl = "func" Receiver MethodName Signature [ FunctionBody ] .
Receiver   = Parameters .
```

接收者通过在方法增加一个额外的参数来指定。这个参数必须是一个非可变参数。它的类型必须是 T 或者 T 的指针（可能包含括号）。T 被称作接收者的基础类型；它不能是指针或接口类型，并且只能在同一个包中定义方法。声明后，我们认为方法绑定了基础类型，并且可以通过 T 或 *T 选择器访问方法名。

非空的接收者标识符在方法签名中必须是唯一的。如果接收者的值没有在该方法中使用，那么接收者标识符可以省略。函数和方法的参数也是一样。

对于一个基础类型。绑定的非空的方法名必须是唯一的。如果基础类型是一个结构体，非空的方法名也不能与结构体字段重复。

给定一个`Point`类型。声明：

```
func (p *Point) Length() float64 {
    return math.Sqrt(p.x * p.x + p.y * p.y)
}

func (p *Point) Scale(factor float64) {
    p.x *= factor
    p.y *= factor
}
```

为类型`*Point`绑定了2个方法`Length`和`Scale`。

方法的类型就是以接收者作为第一个参数的函数类型，例如`Scale`方法：

```
func(p *Point, factor float64)
```

但是以这种方式声明的函数并不是方法。

更多Golang资料包：https://github.com/0voice/Introduction-to-Golang

表达式

表达式通过针对运算元使用运算符和函数来获取计算值。

运算元

运算元代表表达式中的一个简单的。运算元可以是字面值，非空标识符。或括号表达式。

空标识符只能出现在赋值声明的左侧。

```
Operand     = Literal | OperandName | MethodExpr | "(" Expression ")" .
Literal     = BasicLit | CompositeLit | FunctionLit .
BasicLit    = int_lit | float_lit | imaginary_lit | rune_lit | string_lit .
OperandName = identifier | QualifiedIdent.
```

修饰标识符

修饰标识符是以包名作为前缀修饰的标识符。包名和标识符都不能为空。

```
QualifiedIdent = PackageName "." identifier .
```

修饰标识符可以用来访问不同包（需要先导入）中的标识符。标识符必须是导出的并在包级代码块声明才能够被访问。

```
math.Sin    // 表示 math 包中的 Sin 函数
```

复合字面值

复合字面值能为结构体、数组、切片和 map 初始化值。它每次只能创建一个值。字面值由一个字面值类型和使用括号括起来的元素列表组成。元素前也可以声明元素对应的键。

```
CompositeLit  = LiteralType LiteralValue .
LiteralType   = StructType | ArrayType | "[" "..." "]" ElementType |
                SliceType | MapType | TypeName .
LiteralValue  = "{" [ ElementList [ "," ] ] "}" .
ElementList   = KeyedElement { "," KeyedElement } .
KeyedElement  = [ Key ":" ] Element .
Key           = FieldName | Expression | LiteralValue .
FieldName     = identifier .
Element       = Expression | LiteralValue .
```

字面值类型的底层类型必须是一个结构体，数组，切片或 map 类型（如果没有指定类型名就会强制执行这个约束）。元素的类型和键都必须能够分配给相应的字段的元素和键类型；没有额外的类型转换。键可以表示结构体的字段名，切片和数组的索引，map 类型的键。对于 map 字面值，所有的元素都必须有键。如果相同字段名或常量值的键对应多个元素就会报错。如果 map 类型的键为非常量类型，请看求值顺序章节。

结构体字面值遵循以下规则：

- 在结构体中，键必须是它的字段名。
- 不包含任何键的元素列表的顺序需要与结构体字段的声明顺序相同。
- 如果一个元素指定了键，那么所有的元素都必须指定键。
- 包含键的元素列表不需要指定结构体的每个字字段，缺省字段会使用字段类型的零值。
- 字面值可以不指定元素；这样的字面值等于该类型的零值。
- 指定非本包的非导出字段会报错。

给定声明：

```
type Point3D struct { x, y, z float64 }
type Line struct { p, q Point3D }
```

我们可以使用这种写法：

```
origin := Point3D{}                            // Point3D 的零值
line := Line{origin, Point3D{y: -4, z: 12.3}}  // line.q.x 的零值
```

数组和切片遵循以下规则：

- 每个元素都关联一个数字索引标记元素再数组中的位置。

- 给元素指定的键会作为它的索引。键必须是能够表示非负的 `int` 类型值的常量；如果是指定类型的常量，那么常量必须是整型。
- 元素没有指定键时会使用之前的索引加一。如果第一个元素没有指定键，它的索引为零。

对复合字面值取址会生成指向由字面量初始化的变量的指针。

```
var pointer *Point3D = &Point3D{y: 1000}
```

数组字面值需要在类型中指定数组的长度。如果提供的元素少于数组的长度，那么缺少元素的位置将会使用元素类型的零值替代。如果索引超过数组的长度会报错。`...` 表示数组的长度等于最大元素索引加一。

```
buffer := [10]string{}             // len(buffer) == 10
intSet := [6]int{1, 2, 3, 5}       // len(intSet) == 6
days := [...]string{"Sat", "Sun"}  // len(days) == 2
```

切片字面值底层其实就是数组字面值。因此它的长度和容量都是元素的最大索引加一。切片字面值的格式为：

```
[]T{x1, x2, … xn}
```

可以在数组上进行切片操作从而获得切片：

```
tmp := [n]T{x1, x2, … xn}
tmp[0 : n]
```

在一个数组、切片或 map 类型 T 中。元素或者 map 的键可能有自己的字面值类型，如果字面值类型和元素或者键类型相同，那么对应的类型标识符可以省略。与之类似，如果元素或键的类型为 `*T`，那么它们的 `&T` 也可以省略。

```
[...]Point{{1.5, -3.5}, {0, 0}}     // same as [...]Point{Point{1.5, -3.5},
Point{0, 0}}
[][]int{{1, 2, 3}, {4, 5}}          // same as [][]int{[]int{1, 2, 3}, []int{4,
5}}
[][]Point{{{0, 1}, {1, 2}}}         // same as [][]Point{[]Point{Point{0, 1},
Point{1, 2}}}
map[string]Point{"orig": {0, 0}}    // same as map[string]Point{"orig": Point{0,
0}}
map[Point]string{{0, 0}: "orig"}    // same as map[Point]string{Point{0, 0}:
"orig"}

type PPoint *Point
[2]*Point{{1.5, -3.5}, {}}          // same as [2]*Point{&Point{1.5, -3.5},
&Point{}}
[2]PPoint{{1.5, -3.5}, {}}          // same as [2]PPoint{PPoint(&Point{1.5,
-3.5}), PPoint(&Point{})}
```

当复合字面值使用字面值类型的类型名格式出现在 `if`、`for` 或 `switch` 语句的关键字和括号之间并且没有使用圆括号包裹的时候，会引发语法歧义。在这种特殊的情况下字面值的括号会被认为是语句的代码块。为了避免歧义，复合字面值必须用括号括起来。

```
if x == (T{a,b,c}[i]) { … }
if (x == T{a,b,c}[i]) { … }
```

下面是合法的数组、切片和 map 的例子：

```
// list of prime numbers
primes := []int{2, 3, 5, 7, 9, 2147483647}

// vowels[ch] is true if ch is a vowel
vowels := [128]bool{'a': true, 'e': true, 'i': true, 'o': true, 'u': true, 'y':
true}

// the array [10]float32{-1, 0, 0, 0, -0.1, -0.1, 0, 0, 0, -1}
filter := [10]float32{-1, 4: -0.1, -0.1, 9: -1}

// frequencies in Hz for equal-tempered scale (A4 = 440Hz)
noteFrequency := map[string]float32{
    "C0": 16.35, "D0": 18.35, "E0": 20.60, "F0": 21.83,
    "G0": 24.50, "A0": 27.50, "B0": 30.87,
}
```

函数字面值

函数字面值表示一个匿名函数。

```
FunctionLit = "func" Function .
```

```
func(a, b int, z float64) bool { return a*b < int(z) }
```

函数字面值能分配给变量或直接调用。

函数字面值是一个闭包。它可以引用包裹函数中的变量，这些变量在包裹函数和函数字面值之间是共享的。并且它会一直存在直到生命周期结束。

主要表达式

主要表达式是一元和二元表达式的运算元。

```
PrimaryExpr =
    Operand |
    Conversion |
    PrimaryExpr Selector |
    PrimaryExpr Index |
    PrimaryExpr Slice |
    PrimaryExpr TypeAssertion |
    PrimaryExpr Arguments .

Selector       = "." identifier .
Index          = "[" Expression "]" .
Slice          = "[" [ Expression ] ":" [ Expression ] "]" |
                 "[" [ Expression ] ":" Expression ":" Expression "]" .
TypeAssertion  = "." "(" Type ")" .
Arguments      = "(" [ ( ExpressionList | Type [ "," ExpressionList ] ) [ "..."
] [ "," ] ] ")" .
```

```
x
2
(s + ".txt")
f(3.1415, true)
Point{1, 2}
m["foo"]
s[i : j + 1]
obj.color
f.p[i].x()
```

选择器

对于一个 x 不是包名的主要表达式，选择器表达式：

```
x.f
```

表示 x 的字段或方法 f（有时为 *x）。标识符 f 叫做（字段/方法）选择器。它不能是空标识符。选择器表达式的类型就是 f 的类型。如果 x 是包名。请参考修饰标识符。

选择器 f 可以表示类型 T 的方法或字段 f。也可以表示类型 T 的嵌入方法或字段 f。访问 f 所需穿过的嵌套层数叫做它在类型 T 中的深度。声明在 T 中的字段或方法的深度为 0。声明在 T 的嵌入字段 A 中的方法或字段的深度等于 f 在 A 中的深度加一。

选择器遵循以下原则：

- 对于非指针/接口类型 T/*T 的值 x，x.f 表示第一层的方法/字段。如果在第一层没有对应的 f，选择器表达式就是非法的。
- 对于接口类型 I 的值 x，`x.f` 表示动态值 x 的方法名 f。如果接口 I 的方法集中没有 f 方法，选择器就是非法的。
- 作为例外，如果 x 是一个指针类型并且 *(x).f* 是合法的选择器表达式（只能表示字段，不能表示方法）。那么(x).f 可以简写成 x.f。
- 在其他情况下，x.f 都是非法的。
- 如果x是指针类型，并且值为 nil，其中 f 为结构体字段。赋值或取值 x.f 会引起运行时恐慌。
- 如果x是接口类型，并且值为 nil。调用 x.f 会引起运行时恐慌。

例如给定声明：

```
type T0 struct {
    x int
```

```
}

func (*T0) M0()

type T1 struct {
    y int
}

func (T1) M1()

type T2 struct {
    z int
    T1
    *T0
}

func (*T2) M2()

type Q *T2

var t T2     // with t.T0 != nil
var p *T2    // with p != nil and (*p).T0 != nil
var q Q = p
```

结果：

```
t.z          // t.z
t.y          // t.T1.y
t.x          // (*t.T0).x

p.z          // (*p).z
p.y          // (*p).T1.y
p.x          // (*(*p).T0).x

q.x          // (*(*q).T0).x        (*q).x is a valid field selector

p.M0()       // ((*p).T0).M0()      M0 expects *T0 receiver
p.M1()       // ((*p).T1).M1()      M1 expects T1 receiver
p.M2()       // p.M2()              M2 expects *T2 receiver
t.M2()       // (&t).M2()           M2 expects *T2 receiver, see section on
Calls
```

但是下面这种方式是不合法的：

```
q.M0()       // (*q).M0 is valid but not a field selector
```

方法表达式

如果 M 在类型 T 的方法集中。那么 T.M 就是能够正常调用的函数。使用与 M 相同的参数只是在参数列表的最前面增加了接收者参数。

```
MethodExpr    = ReceiverType "." MethodName .
ReceiverType  = TypeName | "(" "*" TypeName ")" | "(" ReceiverType ")" .
```

假设结构体 T 有两个方法。接收者类型为 T 的 Mv 方法和接收者类型为 *T 的 Mp 方法：

```
type T struct {
    a int
}
func (tv  T) Mv(a int) int         { return 0 }  // value receiver
func (tp *T) Mp(f float32) float32 { return 1 }  // pointer receiver

var t T
```

表达式

```
T.Mv
```

将会生成一个等价于 Mv 方法只是第一个参数显式声明接受者的函数。它的签名为：

```
func(tv T, a int) int
```

这个函数能够通过接收者正常调用，以下5种方式是等价的：

```
t.Mv(7)
T.Mv(t, 7)
(T).Mv(t, 7)
f1 := T.Mv; f1(t, 7)
f2 := (T).Mv; f2(t, 7)
```

与之类似：

```
(*T).Mp
```

生成表示 Mp 的函数签名：

```
func(tp *T, f float32) float32
```

对于一个把值作为接收者的方法，我们可以显式的从指针接收者获得函数：

```
(*T).Mv
```

生成表示 Mv 的函数签名：

```
func(tv *T, a int) int
```

这样的函数会通过接收者间接的创建一个值作为接收者传入底层方法中。方法内不能修改接收者的值，因为它的地址是在函数的调用栈里面。

最后一个例子。把值作为接收者函数当做指针作为接收者的方法是非法的，因为指针接收者的方法集中不包含值类型的方法集。

通过函数调用语法从方法中获取函数的值。接收者作为调用函数的第一个参数。给定 `f :=T.Mv`，f 作为 `f(t,7)` 进行调用而不是 `t.f(7)`。想创建一个绑定接收者的函数可以使用函数字面值或者方法值。

在接口类型中定义函数获取函数值是合法的。最终的函数调用会使用接口类型作为接收者。

方法值

如果表达式 x 拥有静态类型 T 并且 M 在类型 T 的方法集中。x.M 叫做方法值。方法值 `x.M` 是一个函数值，这个函数和 x.M 拥有相同的参数列表。表达式 x 在计算方法值时会被保存和计算，这个拷贝的副本会作为任何接下来调用的接收者。

类型 T 可能是接口类型也可能不是接口类型。

与方法表达式中讲过的一样，假设类型 T 有两个方法：接收者类型为 T 的 Mv 和接受者类型为 *T 的 Mp：

```
type T struct {
    a int
}
func (tv  T) Mv(a int) int         { return 0 }  // value receiver
func (tp *T) Mp(f float32) float32 { return 1 }  // pointer receiver

var t T
var pt *T
func makeT() T
```

表达式：

```
t.Mv
```

生成一个类型的函数值：

```
func(int) int
```

以下两种调用是等价的：

```
t.Mv(7)
f := t.Mv; f(7)
```

相似的，表达式：

```
pt.Mp
```

生成一个类型的函数值：

```
func(float32) float32
```

与选择器相同，使用指针调用以值作为接收者的非接口方法会自动将指针解引用：`pt.Mv` 等价于 `(*pt).Mv`。

与方法调用相同，使用值调用以指针作为接收者的非接口方法会自动对值取址：`pt.Mv` 等价于 `(&pt).Mv`。

```
f := t.Mv; f(7)   // like t.Mv(7)
f := pt.Mp; f(7)  // like pt.Mp(7)
f := pt.Mv; f(7)  // like (*pt).Mv(7)
f := t.Mp; f(7)   // like (&t).Mp(7)
f := makeT().Mp   // invalid: result of makeT() is not addressable
```

尽管上面使用的都是非接口类型的例子，不过对于接口类型同样适用。

```
var i interface { M(int) } = myVal
f := i.M; f(7)  // like i.M(7)
```

index表达式

主要表达式格式：

```
a[x]
```

可以表示数组元素、数组的指针、切片、字符串或 map 类型 a 索引 x 对应的值。x 称作索引或者 map 的键。遵循以下规则：

如果a不是 map 类型：

- 索引 x 必须是整型或无类型常量。
- 常量索引必须是非负数且可以使用 int 类型表示。
- 无类型的常量索引会作为 int 型的值。
- 索引 x 的范围在 `0<=x<len(a)` 内，否则就是越界。

对于数组类型 A：

- 常量索引必须在合法范围内。
- 如果 x 在运行时越界会引起运行时恐慌。
- `a[x]` 表示数组在索引 x 处的元素。`a[x]` 的类型就是 A 的元素类型。

对于数组的指针类型：

- 可以使用 `a[x]` 表示 `(*a)[x]`。

对于切片类型 S：

- 如果 x 在运行时越界会引起运行时恐慌。
- `a[x]` 表示切片在索引 x 处的元素。`a[x]` 的类型就是 S 的元素类型。

对于字符串类型：

- 如果字符串 a 为常量，那么常量索引必须在合法范围内。
- 如果 x 在运行时越界会引起运行时恐慌。
- a[x] 表示索引 x 处的非常量字节，它是byte类型。
- 不能对 a[x] 分配值。

对于 map 类型 M：

- 必须保证 x 的类型能够给 M 的键分配值。
- 如果map包含键为 x 的值，a[x] 就是 map 中键 x 对应的值，它的类型就是 M 的元素类型。
- 如果 map 值为 nil 或不包含这个实体，那么 a[x] 为 M 元素类型的零值。

否则 a[x] 就是非法的。

基于 `map[K]V` 类型 a 的索引表达式可以使用特殊格式的赋值和初始化语法。

```
v, ok = a[x]
v, ok := a[x]
var v, ok = a[x]
```

它会额外生成一个无类型的布尔值。如果 ok 是 true，那么代表在map中有该键，如果没有 ok 为 false。

给一个值为 nil 的 map 类型变量赋值会导致运行时恐慌。

切片表达式

切片表达式可以基于字符串、数组、数组指针、切片创建字符串子串或切片。它有两种变体，一种是简单的格式是指定开始和结束位置，完全格式的语法还可以指定容量。

####### 简单切片表达式

对于数组、字符串、指针数组、切片 a，主要表达式：

```
a[low:high]
```

可以构造字符串子串或切片。索引 `low` 和 `high` 决定结果切片中的元素。结果切片的索引从 0 开始，长度为 `high - low`。从数组切分出的切片 s 拥有类型 `[]int`，长度为 3，容积为 4。

```
a := [5]int{1, 2, 3, 4, 5}
s := a[1:4]
```

```
s[0] == 2
s[1] == 3
s[2] == 4
```

为了方便起见，索引值都可以缺省。当 low 缺省时默认从 0 开始。当缺 high 缺省时默认的取切片的长度。

```
a[2:]  // same as a[2 : len(a)]
a[:3]  // same as a[0 : 3]
a[:]   // same as a[0 : len(a)]
```

如果 a 是一个数组指针，那么 `a[low:high]` 可以表示 `(*a)[low : high]`。

对于数组或者字符串，索引的范围是`0<=low<=high<=len(a)`。对于切片，最大的索引值可以为切片的容量，而不是切片的长度。常量索引必须为非负数，且能够转换成 int 类型。对于数组或者常量字符串。常量索引值必须在合法范围内。如果2个索引都是常量。low 必须小于 high。如果索引在运行时访问了非法内存，程序会发生运行时恐慌。

除了无类型字符串，对于切片和字符串的操作结果是非常量类型的值，它的类型与运算元相同。如果运算元为无类型字符串，那么结果类型会为 string。如果把数组作为运算元，它必须是可寻址的，并且获得的切片和原数组具有同一元素类型。

如果切片运算元为 nil，那么结果也是 nil。否则结果切片会和运算元共享相同的底层无类型数组。

完全切片表达式

对于数组，数组指针或非字符串切片，主要表达式为：

```
a[low : high : max]
```

它会构造一个同类型切片，并具有与简单切片表达式的 `a[low:high]` 相同的长度和元素。另外，它还可以把切片的容量设置为 `max - low`。这时只有第一个索引可以为缺省值，默认为零。从数组中获得切片以后：

```
a := [5]int{1, 2, 3, 4, 5}
t := a[1:3:5]
```

切片 t 为 []int 类型，长度为 2，容量为 4，并且元素为：

```
t[0] == 2
t[1] == 3
```

和简单切片表达式一样，如果 a 是数组指针 ，那么 `a[low:high:max]` 可以简写为 `(*a)[low:high:max]`。如果切分操作元是数组，那么这个数组必须是可以寻址的。

如果索引必须在 `0 <= low <= high <= max <= cap(a)` 范围内。常量索引不能是负数并且能够使用 `int` 类型表示；对于数组，索引必须在合法范围内。如果有多个索引都是常量的，那么所有索引都需要在合法范围内。如果索引是非法的，会引起运行时恐慌。

类型断言

对于接口类型 x 和类型 T，主要表达式：

```
x.(T)
```

可以断言 x 不是 nil 且 x 的值是 T 类型。标记 x.(T) 叫做类型断言。

更确切的说，如果 T 不是接口类型，那么 x.(T) 将会断言动态类型 x 的类型是不是 T。

这时，T 必须实现了 x 的（接口）类型。否则断言会是非法的因为 x 不能保存 T 类型的值。如果 T 是接口类型，那么可以断言动态类型 x 是否实现了 T 接口。

如果类型断言成功，表达式的值为 x 的值，但它的类型是T。如果断言失败，将会导致运行时恐慌。换句话说，即使 x 是运行时确定的，x.(T) 也必须是编程时就确认存在的。

```
var x interface{} = 7           // x 拥有动态类型 int 值为 7
i := x.(int)                    // i 为 int 类型值为 7

type I interface { m() }

func f(y I) {
   s := y.(string)         // 非法: 字符串没有实现接口 I （缺少 m 方法）
   r := y.(io.Reader)      // r 拥有接口 io.Reader 所以 y 的动态类型必须同时实现 I 和
io.Reader
   …
}
```

类型断言可以使用特定格式的赋值和初始化语句。

```
v, ok = x.(T)
v, ok := x.(T)
var v, ok = x.(T)
var v, ok T1 = x.(T)
```

这时将会额外生成一个无类型的布尔值。如果断言成功，ok返回 `true`，否则是 `false`。并且 v 会是 T 类型的零值。这时不会有恐慌发生。

调用

给定函数类型为 F 的表达式 f:

```
f(a1, a2, … an)
```

可以使用 a1,a2...an 来调用函数 f。除一种特殊情况之外，函数参数必须是对应 F 函数参数类型的单值表达式，且在函数调用前就已经完成求值。表达式的结果类型是 f 的结果类型。函数调用和方法调用相似，只是方法额外需要一个接收者类型。

```
math.Atan2(x, y)  // function call
var pt *Point
pt.Scale(3.5)     // method call with receiver pt
```

在函数调用中，函数的值和参数是按照顺序求值的。在计算之后作为参数会传进函数，函数开始执行。当函数执行完成后返回的参数将会返回给函数的调用者。

调用值为 nil 的函数会导致运行时恐慌。

作为特例，如果函数或者方法的返回值等于参数列表的个数，那么会嵌套调用。这将把返回值直接赋值给下一次调用函数的参数。

```
func Split(s string, pos int) (string, string) {
    return s[0:pos], s[pos:]
}

func Join(s, t string) string {
    return s + t
}

if Join(Split(value, len(value)/2)) != value {
    log.Panic("test fails")
}
```

如果 x 的方法集中包含 m 那么 `x.m()` 是合法的。并且参数列表和 m 的参数列表相同。如果x是可寻址的，那么那么x指针的方法集 `(&x).m()` 可以简写成 `x.m()`。

```
var p Point
p.Scale(3.5)
```

没有方法类型，也没有方法字面值。

更多Golang资料包：https://github.com/0voice/Introduction-to-Golang

通过 `...` 来传递参数

如果 f 的最后一个参数 p 的类型是 `...T`。那么在函数内部 p 参数的类型就是 `[]T`。如果 f 调用时没有传入 p 对应的参数，那么p为 nil。否则这些参数会以切片方式传入，在新的底层切片中。切片中的类型都是能赋值给类型 T 的值。这个切片的长度和容量在不同的调用中有所不同。

给定函数调用：

```
func Greeting(prefix string, who ...string)
Greeting("nobody")
Greeting("hello:", "Joe", "Anna", "Eileen")
```

在 `Greeting` 中，第一次调用时，`who` 是 nil 类型。而在第二次调用时是 `[]string{"Joe", "Anna", "Eileen"}`。

如果在调用的时候的最后一个参数是[]T，那么我们可以使用 `...` 来将切片中的值依次赋值给参数列表。

给定切片s并且调用：

```
s := []string{"James", "Jasmine"}
Greeting("goodbye:", s...)
```

z 在 `Greeting`。中 `who` 会和切片 s 共享相同的底层数组。

操作符

操作符用来连接运算元。

```
Expression = UnaryExpr | Expression binary_op Expression .
UnaryExpr  = PrimaryExpr | unary_op UnaryExpr .

binary_op  = "||" | "&&" | rel_op | add_op | mul_op .
rel_op     = "==" | "!=" | "<" | "<=" | ">" | ">=" .
add_op     = "+" | "-" | "|" | "^" .
mul_op     = "*" | "/" | "%" | "<<" | ">>" | "&" | "&^" .

unary_op   = "+" | "-" | "!" | "^" | "*" | "&" | "<-" .
```

比较运算符在此处讨论。对于其他二元操作符，两个操作元的类型必须是相同的，除了位移和无类型常量。针对常量的操作，请看常量表达式章节。

除了位移操作，如果其中一个操作符是无类型常量，而另个不是，那么无类型的常量会转换成另一个运算元的类型。

在右移表达式中的运算元必须是无符号的整数或者可以转换成 uint 的无类型的常量。如果左移一个无类型常量那么结果依然是无类型的。他首先会转换成指定类型。

```
var s uint = 33
var i = 1<<s           // 1 has type int
var j int32 = 1<<s     // 1 has type int32; j == 0
var k = uint64(1<<s)   // 1 has type uint64; k == 1<<33
var m int = 1.0<<s     // 1.0 has type int; m == 0 if ints are 32bits in size
var n = 1.0<<s == j    // 1.0 has type int32; n == true
var o = 1<<s == 2<<s   // 1 and 2 have type int; o == true if ints are 32bits in
size
var p = 1<<s == 1<<33  // illegal if ints are 32bits in size: 1 has type int,
but 1<<33 overflows int
var u = 1.0<<s         // illegal: 1.0 has type float64, cannot shift
var u1 = 1.0<<s != 0   // illegal: 1.0 has type float64, cannot shift
var u2 = 1<<s != 1.0   // illegal: 1 has type float64, cannot shift
var v float32 = 1<<s   // illegal: 1 has type float32, cannot shift
var w int64 = 1.0<<33  // 1.0<<33 is a constant shift expression
```

运算符优先级

一元运算符拥有最高优先级。++ 和 -- 是语句而不是表达式，他们在运算符的优先级之外。所以 (*p)++ 和 *p++ 是一样的。

二元运算符有 5 个优先级。乘法运算符在最高级，紧接着是加法运算符。比较运算符，&& 运算符，最后是 ||。

```
Precedence    Operator
    5             *  /  %  <<  >>  &  &^
    4             +  -  |  ^
    3             ==  !=  <  <=  >  >=
    2             &&
    1             ||
```

相同优先级的二元运算符的执行顺序是由左到右。例如 `x/y*z` 和 `(x/y)*z` 是一样的。

```
+x
23 + 3*x[i]
x <= f()
^a >> b
f() || g()
x == y+1 && <-chanPtr > 0
```

算数运算符

算数运算符应用在 2 个数字值之间，别切生成一个相同类型的值作为第一个运算元。四种算数运算符 (+,-,*,/)应用在数字，浮点，复合类型之中。+ 也可以用于字符串。位运算和位移运算只适用于整数。

```
+    sum                    integers, floats, complex values, strings
-    difference             integers, floats, complex values
*    product                integers, floats, complex values
/    quotient               integers, floats, complex values
%    remainder              integers

&    bitwise AND            integers
|    bitwise OR             integers
^    bitwise XOR            integers
&^   bit clear (AND NOT)    integers

<<   left shift             integer << unsigned integer
>>   right shift            integer >> unsigned integer
```

数字运算符

对于两个整数 x 和 y。整数商 `q=x/y` 和余数 `r=x%y` 遵循以下规律。

```
x = q*y + r  and  |r| < |y|
```

`x/y` 截断为 0。

```
x     y     x / y     x % y
 5     3       1         2
-5     3      -1        -2
 5    -3      -1         2
-5    -3       1        -2
```

作为这个规则的例外情况，如果 x 非常大，那么 `q=x/-1` 等于 x。

```
x, q
int8                    -128
int16                 -32768
int32            -2147483648
int64    -9223372036854775808
```

如果除数是一个常量。那么它不能是 0，如果除数在运行时为 0，会导致运行时恐慌。如果除数是负数并且除数是：

```
x     x / 4     x % 4     x >> 2     x & 3
 11      2         3         2          3
-11     -2        -3        -3          1
```

位移运算符移动左侧运算元右侧元算元指定的位数。如果左侧是有符号整型，那它就实现了位移运算，如果是无符号整数使用逻辑位移。位移运算没有上限，位移操作让左边运算元位移 n 个 1。 `x<<1` 和 `x*2` 是相等的。并且 `x>>1` 和 `x/2` 是相同的。

对于整数运算元，一元运算符+-^定义如下：

```
+x                          is 0 + x
-x    negation              is 0 - x
^x    bitwise complement    is m ^ x  with m = "all bits set to 1" for unsigned
x
                                      and  m = -1 for signed x
```

整型溢出

对于无符号的值，运算符+-*和<<都是2禁止运算。这里的n是无符号类型的宽度，无符号整型将会丢弃溢出的位，并且程序将会返回`wrap around`。

对于有符号的整数，操作符+=*<<都会溢出并且值存在，并且代表相应的有符号的值。在运算时不会抛出异常。标一起不会报错。所以不是所有情况下x<x+1都成立。

浮点数运算符

对于浮点数和其他复杂数字，+x和x是一样的，-x是x的对立面。除了IEEE-754还没有指定浮点数除0或者复数的结果。是否抛出异常将会依赖其具体实现。

一种实现可以合并多个浮点操作进一个操作，有可能是夸语句的，并且他的结果可能和依次单独执行的结果不一样。1个浮点数类型将会转变成目标的精度，防止四舍五入的融合。

```
// FMA allowed for computing r, because x*y is not explicitly rounded:
r  = x*y + z
r  = z;   r += x*y
t  = x*y; r = t + z
*p = x*y; r = *p + z
r  = x*y + float64(z)

// FMA disallowed for computing r, because it would omit rounding of x*y:
r  = float64(x*y) + z
r  = z; r += float64(x*y)
t  = float64(x*y); r = t + z
```

字符串

字符串可以使用+和+=操作符。

```
s := "hi" + string(c)
s += " and good bye"
```

字符串想家将会创建一个新的字符串。

比较运算符

比较运算符比较连个运算元，并且生成一个无类型的布尔值。

```
==    equal
!=    not equal
<     less
<=    less or equal
>     greater
>=    greater or equal
```

在任何比较运算元中2种类型必须是可以分配的。

使用等于运算符`==`和`!=`的运算元必须是可比较的。使用顺序运算符`<`,`<=`,`>`和`>=`必须是可比较的。这些限制导致比较运算符被定义成以下的方式。

- 布尔值是可比较的，两个布尔值当他们同为`true`或者false的使用是相等的
- 整数值是可比较和排序的
- 浮点数是可比较和排序的，具体定义在IEEE-754标准中。
- 复数是可比较的，2个复数当实部和虚部都相等时就是相等的。
- 字符串是可以比较和排序的。是按照字节顺序排序。
- 指针式可以排序的，连个指针当指向相同变量时是相同的，或者他们2个都是nil。指向一个为非配的变量的结果是未定义的。
- channel是可比较的。当两个管道是用同一个make出来的，或者都是nil时时相等的。
- 接口值时可以比较的，2个接口值时相等的如果2个标识符的动态类型是一样的或者他们都是nil。
- 一个非接口类型的值x和一个接口类型的值T在非接口类型是可以比较的并且非接口类型实现了接口是是可以比较的。当他们的动态类型类型相同时时相等的。
- 当结构体内的所有字段都是可以比较的时候，他是可以比较的。连个结构体的值当非空字段都相等时他们是相等的。
- 数组类型的值时可比较的，如果数组的原属时可以比较的，那么当数组的所有值是相等的时候他们就是相等的。

使用两个动态类型的标识符来比较接口的值。如果这个类型的值时不可比较的，那么将会引起一个panic。这个行为不仅仅时接口，数组结构体接口字段都有这个问题。

切片，map，和函数值都是不可比较的，然而，作为一个特殊的例子，切片，map和函数的值的nil时可以比较的，指针，channel和接口的值nil也是可以比较的。

```
const c = 3 < 4            // c is the untyped boolean constant true

type MyBool bool
var x, y int
var (
    // The result of a comparison is an untyped boolean.
    // The usual assignment rules apply.
    b3        = x == y // b3 has type bool
    b4 bool   = x == y // b4 has type bool
    b5 MyBool = x == y // b5 has type MyBool
)
```

逻辑操作符

逻辑运算符使用布尔值值，并且生成一个相同类型的结果值作为操作元。右面的操作元计算是有条件的。

```
&&    conditional AND    p && q  is  "if p then q else false"
||    conditional OR     p || q  is  "if p then true else q"
!     NOT                !p      is  "not p"
```

地址操作符

以类型 T 的 x 作为运算元，取址操作 `&x` 会生成一个类型为 `*T` 并指向 x 的指针。运算元必须是能够取址的，它可以是一个变量，指针，切片的取值操作；或是一个可取址结构体的字段选择器；或是对于可取址数组的索引取值操作。作为寻址能力的例外，x 可能是一个复合字面值。如果对 x 进行取址操作将会 panic，`&x` 也会 panic。

对于一个 `*T` 类型的运算元 x，指针解引用 `*x` 表示 x 指向的 T 类型。如果 x 为 nil，那么解引用 `*x` 会 panic。

```
&x
&a[f(2)]
&Point{2, 3}
*p
*pf(x)

var x *int = nil
*x   // causes a run-time panic
&*x  // causes a run-time panic
```

接收操作符

对于管道类型的运算元 `ch`，接收操作 `<-ch` 返回值是管道 `ch` 接收到的值。带方向的管道需要有接受权限，接收操作的类型也是通道的元素类型。表达式会一直阻塞直到接收到返回值。从 nil 通道接收值会一直阻塞。从一个已经关闭的通道接收数据会在其他数据都被接收以后生成该通道元素类型的零值。

```
v1 := <-ch
v2 = <-ch
f(<-ch)
<-strobe  // wait until clock pulse and discard received value
```

接收数据的表达式可以使用赋值表达式。

```
x, ok = <-ch
x, ok := <-ch
var x, ok = <-ch
var x, ok T = <-ch
```

它还可以生成一个额外的无类型布尔值来表示通道是否关闭。如果 ok 为 `true` 说明获取到的是发送到通道内的数据，而 `false` 它就返回一个零值因为通道内没有元素且已经关闭。

类型转换

类型转换表达式 `T(x)` 其中 T 代表类型，x 代表可以转换成 T 类型的表达式。

```
Conversion = Type "(" Expression [ "," ] ")" .
```

如果类型是以 `*` 或 `<-` 开头，或以关键字 `func` 开头并且没有返回值列表，那么它必须用括号括起来避免歧义：

```
*Point(p)        // same as *(Point(p))
(*Point)(p)      // p is converted to *Point
<-chan int(c)    // same as <-(chan int(c))
(<-chan int)(c)  // c is converted to <-chan int
func()(x)        // function signature func() x
(func())(x)      // x is converted to func()
(func() int)(x)  // x is converted to func() int
func() int(x)    // x is converted to func() int (unambiguous)
```

常量 x 可以在可以用类型 T 表示时自动转换。作为一个特例，整数常量 x 可以转换成字符串类型就和非常量 x 一样。

对常量的转换会生成一个指定类型的常量。

```
uint(iota)               // iota value of type uint
float32(2.718281828)     // 2.718281828 of type float32
complex128(1)            // 1.0 + 0.0i of type complex128
float32(0.49999999)      // 0.5 of type float32
float64(-1e-1000)        // 0.0 of type float64
string('x')              // "x" of type string
string(0x266c)           // "♬" of type string
MyString("foo" + "bar")  // "foobar" of type MyString
string([]byte{'a'})      // not a constant: []byte{'a'} is not a constant
(*int)(nil)              // not a constant: nil is not a constant, *int is not a
boolean, numeric, or string type
int(1.2)                 // illegal: 1.2 cannot be represented as an int
string(65.0)             // illegal: 65.0 is not an integer constant
```

非常量 x 可以在以下情况下转换成类型 T：

- x 可以给类型 T 赋值
- 忽略的结构体标签，x 的类型和 T 具有相同的底层类型
- 忽略的结构体标签，x 的类型和 T 都是指针类型，并且指针所指的类型具有相同的底层类型
- x 的类型和 T 都是整数或者浮点数类型
- x 的类型和 T 都是复数类型
- x 是一个字符串而 T 时字节切片或者 rune 切片

在比较两个结构体类型的时候会忽略结构体标签：

```
type Person struct {
    Name    string
    Address *struct {
        Street string
        City   string
    }
}

var data *struct {
    Name    string `json:"name"`
    Address *struct {
        Street string `json:"street"`
        City   string `json:"city"`
    } `json:"address"`
}

var person = (*Person)(data)  // ignoring tags, the underlying types are
identical
```

这个规则也适用于数字类型与字符串类型间的相互转换。这个转换可能会改变 x 的值并且会增加运行时消耗。包 unsafe 实现了这个功能底层的限制。

数字之间的转换

对于非常量的数字转换，需要遵守以下规则：

- 在转换整型数字时，如果是一个有符号整型，它是继承有符号的无限精度；否则就不用继承符号。转换时会截断数字以适应类型的大小。例如：如果 `v:=uint16(0x10F0)`，然后 ``uint32(int8(v)) == 0xFFFFFFF0 。类型转换总是生成有效值，并且永远不会溢出。
- 如果要将浮点数转换成整型，会丢弃小数部分（截断为零）。
- 如果要将整型或浮点型转换成浮点数类型，或或者一个复数转换成其他复数类型，结果会四舍五入成指定精度。例如： 可以使用超出IEEE-754 32位数的附加精度来存储float32类型的变量x的值，但float32（x）表示将x的值舍入为32位精度的结果。x + 0.1 会使用超过 32 位的精度，而 `float32(x+0.1)` 不会。

在所有浮点数和复数的非常量转换中，如果结构类型不能成功表示数据，那么结果将会依赖于具体平台实现。

字符串的类型转换

1. 转换一个有符号或者无符号的整型值会转换成对应的 UTF-8 表示整型值。不在范围内的 Unicode 代码点会转换成 "\uFFFD"。

```
string('a')       // "a"
string(-1)        // "\ufffd" == "\xef\xbf\xbd"
string(0xf8)      // "\u00f8" == "ø" == "\xc3\xb8"
type MyString string
MyString(0x65e5)  // "\u65e5" == "日" == "\xe6\x97\xa5"
```

1. 将字节切片转换成字符串类型会生成一个由切片元素组成的字符串

```
string([]byte{'h', 'e', 'l', 'l', '\xc3', '\xb8'})   // "hellø"
string([]byte{})                                     // ""
string([]byte(nil))                                  // ""

type MyBytes []byte
string(MyBytes{'h', 'e', 'l', 'l', '\xc3', '\xb8'})  // "hellø"
```

1. 将 rune 切片转换成字符串类型会生成一个由切片元素组成的字符串

```
string([]rune{0x767d, 0x9d6c, 0x7fd4})   // "\u767d\u9d6c\u7fd4" == "白鹏翔"
string([]rune{})                         // ""
string([]rune(nil))                      // ""

type MyRunes []rune
string(MyRunes{0x767d, 0x9d6c, 0x7fd4})  // "\u767d\u9d6c\u7fd4" == "白鹏翔"
```

1. 将字符串转换成字节切片会生成由字符串中每个字节组成的切片

```
[]byte("hellø")   // []byte{'h', 'e', 'l', 'l', '\xc3', '\xb8'}
[]byte("")        // []byte{}

MyBytes("hellø")  // []byte{'h', 'e', 'l', 'l', '\xc3', '\xb8'}
```

1. 将字符串转换成 rune 切片会生成由字符串中每个 Unicode 代码点组成的切片

```
[]rune(MyString("白鹏翔"))  // []rune{0x767d, 0x9d6c, 0x7fd4}
[]rune("")                 // []rune{}

MyRunes("白鹏翔")           // []rune{0x767d, 0x9d6c, 0x7fd4}
```

常量表达式

常量表达式只包含常量运算元并且在编译程序时就已经计算完成。

无类型布尔值，数值和字符串常量都可以当作运算元。除了位置操作符，如果二元运算符右不同类型的常量，操作元，和非布尔值，和即将在接下来出现的：整型，rune，浮点数和复数类型。例如：一个无类型整型常量减去无类型复数常量，结果为复数常量。

一个常量的比较运算会生成无类型的布尔常量。如果左移运算是一个无类型常量，结果会是一个整型常量。它会和原来常量为相同类型。其他与无类型常量的运算都会生成相同类型的结果（布尔值，整型，浮点数，复数，字符串常量）。

```
const a = 2 + 3.0          // a == 5.0   (untyped floating-point constant)
const b = 15 / 4           // b == 3     (untyped integer constant)
const c = 15 / 4.0         // c == 3.75  (untyped floating-point constant)
const Θ float64 = 3/2      // Θ == 1.0   (type float64, 3/2 is integer division)
const Π float64 = 3/2.     // Π == 1.5   (type float64, 3/2. is float division)
const d = 1 << 3.0         // d == 8     (untyped integer constant)
const e = 1.0 << 3         // e == 8     (untyped integer constant)
const f = int32(1) << 33   // illegal    (constant 8589934592 overflows int32)
const g = float64(2) >> 1  // illegal    (float64(2) is a typed floating-point
constant)
const h = "foo" > "bar"    // h == true  (untyped boolean constant)
const j = true             // j == true  (untyped boolean constant)
const k = 'w' + 1          // k == 'x'   (untyped rune constant)
const l = "hi"             // l == "hi"  (untyped string constant)
const m = string(k)        // m == "x"   (type string)
const Σ = 1 - 0.707i       //            (untyped complex constant)
const Δ = Σ + 2.0e-4       //            (untyped complex constant)
const Φ = iota*1i - 1/1i   //            (untyped complex constant)
```

对一个无类型整数，rune，或浮点数应用内置的 `complex` 函数会生成无类型的复数常量。

```
const ic = complex(0, c)   // ic == 3.75i  (untyped complex constant)
const iΘ = complex(0, Θ)   // iΘ == 1i     (type complex128)
```

常量表达式总是一个明确的值；中间值和常量自己可以比语言所支持的精度更高，下面的声明是合法的：

```
const Huge = 1 << 100         // Huge == 1267650600228229401496703205376
(untyped integer constant)
const Four int8 = Huge >> 98  // Four == 4                                  (type
int8)
```

常量的除法的除数不能为 0:

```
3.14 / 0.0   // illegal: division by zero
```

定义了类型的常量的精度必须根据常量类型定义。所以下面的常量表达式是非法的:

```
uint(-1)     // -1 cannot be represented as a uint
int(3.14)    // 3.14 cannot be represented as an int
int64(Huge)  // 1267650600228229401496703205376 cannot be represented as an
int64
Four * 300   // operand 300 cannot be represented as an int8 (type of Four)
Four * 100   // product 400 cannot be represented as an int8 (type of Four)
```

补码使用的一元操作符 ^ 对于非常量的匹配模式：补码对于无符号常量为 1，对于有符号和无类型常量为 -1。

```
^1         // untyped integer constant, equal to -2
uint8(^1)  // illegal: same as uint8(-2), -2 cannot be represented as a uint8
^uint8(1)  // typed uint8 constant, same as 0xFF ^ uint8(1) = uint8(0xFE)
int8(^1)   // same as int8(-2)
^int8(1)   // same as -1 ^ int8(1) = -2
```

实现限制：编译器在处理无类型浮点数和复数时会取近似值；具体请看常量章节。这个取近似值的操作在浮点数在整数上下文时会产生无效值，即使在计算过后是一个整型。

运算优先级

在包级别，初始化的依赖性由变量声明的初始化表达式顺序决定。否则，当计算表达式内的操作数时，赋值，返回语句，所有函数调用，方法调用，和通信操作都会由左向右计算。

例如，在函数作用域中的赋值:

```
y[f()], ok = g(h(), i()+x[j()], <-c), k()
```

函数调用和通信的发生顺序为：`f()`，`h()`，`i()`，`j()`，`<-c`，`g()` 和 `k()`。但是对 y 和 x 的取值操作没有指定。

```
a := 1
f := func() int { a++; return a }
x := []int{a, f()}            // x may be [1, 2] or [2, 2]: evaluation order
between a and f() is not specified
m := map[int]int{a: 1, a: 2}  // m may be {2: 1} or {2: 2}: evaluation order
between the two map assignments is not specified
n := map[int]int{a: f()}      // n may be {2: 3} or {3: 3}: evaluation order
between the key and the value is not specified
```

在包级别，依赖的初始化顺序会覆盖这个从左向右的规则：

```
var a, b, c = f() + v(), g(), sqr(u()) + v()

func f() int        { return c }
func g() int        { return a }
func sqr(x int) int { return x*x }

// functions u and v are independent of all other variables and functions
```

语句

语句控制程序的执行。

```
Statement =
    Declaration | LabeledStmt | SimpleStmt |
    GoStmt | ReturnStmt | BreakStmt | ContinueStmt | GotoStmt |
    FallthroughStmt | Block | IfStmt | SwitchStmt | SelectStmt | ForStmt |
    DeferStmt .

SimpleStmt = EmptyStmt | ExpressionStmt | SendStmt | IncDecStmt | Assignment |
ShortVarDecl .
```

终止语句

终止语句会阻止相同代码块中下面所有语句的执行。以下语句属于终止语句：

1. `return` 和 `goto` 语句
2. 对内置 `panic` 函数的调用

1. 代码块结束
2. `if` 语句中：

1. else 分支
2. 所有分支末尾

1. `for`语句中：

1. `break` 语句和循环结束

1. `switch` 语句：

1. 在 `switch` 语句中没有 `break` 语句，
2. 有一个默认的 case

1. 语句列表中的每个 case 语句和有可能存在的 `fallthrough` 语句

1. `select` 语句中：

1. 没有 break 语句
2. 每个 case 中的语句列表，如果包含默认 case

所有其他语句都不是中断语句。

如果语句序列不为空并且最后一个非空语句是终止语句，那么语句序列就以终结语句结尾。

空语句

空语句不做任何事情。

```
EmptyStmt = .
```

标签语句

标签语句可以作为 `goto`，`break` 和 `continue` 语句的目标。

```
LabeledStmt = Label ":" Statement .
Label       = identifier .
```

```
Error: log.Panic("error encountered")
```

表达式语句

除了特定的内置函数，一般的函数、方法和接收操作都可以出现在表达式语句的上下文中。这些语句可以使用括号括起来。

```
ExpressionStmt = Expression .
```

下面的内置函数不允许出现在语句的上下文中：

```
append cap complex imag len make new real
unsafe.Alignof unsafe.Offsetof unsafe.Sizeof
```

```
h(x+y)
f.Close()
<-ch
(<-ch)
len("foo")  // illegal if len is the built-in function
```

发送语句

发送语句可以向通道发送一个值。通道表达式必须是通道类型，通道方向必须允许发送操作，并且值类型是可以分配给通道元素通道类型。

```
SendStmt = Channel "<-" Expression .
Channel  = Expression .
```

通道类型和值表达式会在发送之前求值。发送操作会一致阻塞，直到可以进行发送操作。如果接收者已经准备好向没有缓存的通道发送值可以立即执行。如果通道内还有缓存空间，向通道内发送值也会立即执行。向关闭的通道发送数据会导致运行时恐慌。像值为 nil 的通道发送数据会一直阻塞。

```
ch <- 3  // send value 3 to channel ch
```

递增/递减语句

“++” 和 “--” 语句可以递增或者递减运算元一个无类型常量 1。作为一个赋值语句，运算元必须是可寻址的或者 map 的索引表达式。

```
IncDecStmt = Expression ( "++" | "--" ) .
```

下面的赋值语句在语义上是等价的：

```
IncDec statement    Assignment
x++                 x += 1
x--                 x -= 1
```

赋值

```
Assignment = ExpressionList assign_op ExpressionList .

assign_op = [ add_op | mul_op ] "=" .
```

所有左侧运算元都必须是可寻址的、map 索引表达式或空标识符其中之一。运算元可以用括号括起来。

```
x = 1
*p = f()
a[i] = 23
(k) = <-ch  // same as: k = <-ch
```

对于赋值操作 `x op= y` 其中 `op` 为二元运算符，它和 `x=x op (y)` 是等价的，不过它只计算一次 x。`op=` 是单独的一个词汇单元，在赋值操作中左侧表达式和右侧表达式必须都是单值表达式，并且左侧表达式不能是空白标识符。

```
a[i] <<= 2
i &^= 1<<n
```

元祖赋值语句会把运算返回的多个值分别分配给变量列表。它有两种格式，第一种：它是返回多值的表达式，例如函数调用、通道和 map 运算、类型断言。左侧运算元的数量必须等于返回值的数量。如果函数返回两个值：

```
x, y = f()
```

它会将第一个返回值分配给 x，把第二个返回值分配给 y。第二种格式中，左侧运算元的数量必须等于右侧运算元的数量。每个表达式都只能返回单一值，右侧第 n 个值会赋值给左侧第 n 个变量。

```
one, two, three = '一', '二', '三'
```

空标识符可以在分配时忽略一个右面位置的表达式：

```
_ = x       // evaluate x but ignore it
x, _ = f()  // evaluate f() but ignore second result value
```

赋值分为两个阶段。首先会计算左侧运算元的索引表达式和指针的解引用工作并以一定顺序计算右侧表达式的值。

然后依次对左侧运算元赋值。

```
a, b = b, a  // exchange a and b

x := []int{1, 2, 3}
i := 0
i, x[i] = 1, 2  // set i = 1, x[0] = 2

i = 0
x[i], i = 2, 1  // set x[0] = 2, i = 1

x[0], x[0] = 1, 2  // set x[0] = 1, then x[0] = 2 (so x[0] == 2 at end)

x[1], x[3] = 4, 5  // set x[1] = 4, then panic setting x[3] = 5.

type Point struct { x, y int }
var p *Point
```

```
x[2], p.x = 6, 7  // set x[2] = 6, then panic setting p.x = 7

i = 2
x = []int{3, 5, 7}
for i, x[i] = range x {  // set i, x[2] = 0, x[0]
    break
}
// after this loop, i == 0 and x == []int{3, 5, 3}
```

在赋值语句中每个值都必须能分配给左侧指定类型的值。除了以下特例：

1. 任何类型都能分配给空标识符。
2. 如果把无类型常量分配给接口类型或者空标识符，它会转换成默认类型。

1. 如果无类型的布尔值分配给了接口类型或者空标识符，它会先转换成 `bool` 类型。

if 语句

`if` 语句根据布尔值表达式的值来决定执行条件分支的代码。如果表达式为真，就执行 `if` 分支内的代码，否则执行 `else` 分支的代码。

```
IfStmt = "if" [ SimpleStmt ";" ] Expression Block [ "else" ( IfStmt | Block ) ]
.
```

```
if x > max {
    x = max
}
```

表达式可能先于普通语句，它会在表达式求值之前发生。

```
if x := f(); x < y {
    return x
} else if x > z {
    return z
} else {
    return y
}
```

更多Golang资料包：https://github.com/0voice/Introduction-to-Golang

switch 语句

for 语句

for 语句可以用来重复执行一段代码。它有三种格式：迭代器可以是单一条件、for 分句或者 range 语句。

```
ForStmt = "for" [ Condition | ForClause | RangeClause ] Block .
Condition = Expression .
```

单一条件的 for 语句

这种情况下 for 会在条件为 true 时一直重复。条件会在每次迭代时都重新计算。如果没有指定条件，默认一直为 true。

```
for a < b {
    a *= 2
}
```

带分句的 for 语句

带分句的 for 语句也是由条件控制，只是它有一个初始化和寄送的过程。例如赋值、递增或者递减语句。初始化语句可以是短变量声明，但是寄送语句不能。在初始化语句中声明的变量可以在迭代过程中使用。

```
ForClause = [ InitStmt ] ";" [ Condition ] ";" [ PostStmt ] .
InitStmt = SimpleStmt .
PostStmt = SimpleStmt .
```

```
for i := 0; i < 10; i++ {
    f(i)
}
```

如果初始化语句非空，它会在进入迭代前执行一次；post 语句在每次循环后都会执行一次。在只有条件的情况下可以省略分号。如果缺省条件语句，默认为 true。

```
for cond { S() }    is the same as    for ; cond ; { S() }
for      { S() }    is the same as    for true     { S() }
```

带 range 分句的 for 语句

带 range 分句的 for 语句可以访问数组、切片、字符串、map 的所有元素，还可以从通道中接收值。迭代获得元素分配给了相应的迭代变量并执行代码块。

```
RangeClause = [ ExpressionList "=" | IdentifierList ":=" ] "range" Expression .
```

右侧的 range 分句表达式叫做 range 表达式，它可能是数组、数组的指针、切片、字符串、map 或通道接收者类型。在分配时，左侧运算元必须是可寻址的或者 map 的索引表达式；它们作为迭代变量。如果 range 表达式是一个通道类型，至少需要有一个变量，它也可以有两个变量。如果迭代变量是空标识符，就代表在分句中不存在该标识符。

```
Range expression                          1st value          2nd value

array or slice  a  [n]E, *[n]E, or []E    index    i  int    a[i]       E
string          s  string type            index    i  int    see below  rune
map             m  map[K]V                key      k  K      m[k]       V
channel         c  chan E, <-chan E       element  e  E
```

```
var testdata *struct {
    a *[7]int
}
for i, _ := range testdata.a {
    // testdata.a is never evaluated; len(testdata.a) is constant
    // i ranges from 0 to 6
    f(i)
}

var a [10]string
for i, s := range a {
    // type of i is int
    // type of s is string
    // s == a[i]
    g(i, s)
}

var key string
var val interface {}  // element type of m is assignable to val
```

```
m := map[string]int{"mon":0, "tue":1, "wed":2, "thu":3, "fri":4, "sat":5,
"sun":6}
for key, val = range m {
    h(key, val)
}
// key == last map key encountered in iteration
// val == map[key]

var ch chan Work = producer()
for w := range ch {
    doWork(w)
}

// empty a channel
for range ch {}
```

Go 语句

`go` 语句会开始在相同地址空间中的单独 goroutine 中调用函数。

```
GoStmt = "go" Expression .
```

表达式必须是函数或者方法调用；它不能使用括号括起来，调用内置函数有表达式语句的限制。

函数的值和参数会按顺序在调用的 goroutine 中求值。不像普通的函数调用，程序不会等待函数调用完成，而是直接开启一个新的 goroutine 执行函数。函数退出时，goroutine 也会退出。函数的任何返回值都会被丢弃。

```
go Server()
go func(ch chan<- bool) { for { sleep(10); ch <- true }} (c)
```

select 语句

select 语句会在接收/发送操作集中选择一个执行。它看起来和 switch 很像，只不过是专门针对通信操作的。

```
SelectStmt = "select" "{" { CommClause } "}" .
CommClause = CommCase ":" StatementList .
CommCase   = "case" ( SendStmt | RecvStmt ) | "default" .
RecvStmt   = [ ExpressionList "=" | IdentifierList ":=" ] RecvExpr .
RecvExpr   = Expression .
```

接收表达式可以将接收表达式的值分配给一个或两个变量。接收表达式必须是一个接收运算元（可以使用括号括起来）。它最多允许有一个 default 语句。

select 语句执行以下几个步骤：

1. 对于 select 语句的所有分句，接收操作的通道运算元、通道、发送语句的右侧表达式都会执行一次操作。
2. 如果一个或多个通信同时发生，它会通过一致性随机选择一个执行。如果没有 default 语句，select 语句会一直阻塞。

1. 除了 default 分句，其他分句只有在开始进行通信的时候才会执行。
2. 如果 select 分句是一个接收语句，它可以给变量分配值。

1. 执行 select 分句内的内容。

如果向 nil 通道发送信息在没有 default 分句的情况下会一直阻塞。

```
var a []int
var c, c1, c2, c3, c4 chan int
var i1, i2 int
select {
case i1 = <-c1:
    print("received ", i1, " from c1\n")
case c2 <- i2:
    print("sent ", i2, " to c2\n")
case i3, ok := (<-c3):  // same as: i3, ok := <-c3
    if ok {
        print("received ", i3, " from c3\n")
    } else {
        print("c3 is closed\n")
    }
case a[f()] = <-c4:
    // same as:
    // case t := <-c4
    //  a[f()] = t
default:
    print("no communication\n")
}

for {  // send random sequence of bits to c
    select {
    case c <- 0:  // note: no statement, no fallthrough, no folding of cases
    case c <- 1:
```

```
    }
}

select {}  // block forever
```

return 语句

`return` 语句会终止函数 F 的执行并可选的返回一个或多个返回值。所有的滞后函数都会在 F 返回到它的调用者之前执行。

```
ReturnStmt = "return" [ ExpressionList ] .
```

如果函数没有返回值类型，return 不能返回任何值。

```
func noResult() {
    return
}
```

有三种方式能够返回指定类型的值：

1. 返回值可以直接在 return 语句中列出。每个表达式都必须返回一个值并且能够分配给相应的返回值类型。

```
func simpleF() int {
    return 2
}

func complexF1() (re float64, im float64) {
    return -7.0, -4.0
}
```

1. return 语句的表达式列表可以是一个返回多值的函数调用。这时会使用临时变量来获取函数调用的返回值并直接将其作为 return 语句的表达式列表。

```
func complexF2() (re float64, im float64) {
    return complexF1()
}
```

1. 如果制定了返回值的标识符那么 return 的表达式列表可以为空。返回值参数会作为普通的本地变量按需分配。return 语句会直接返回它们。

```
func complexF3() (re float64, im float64) {
    re = 7.0
    im = 4.0
    return
}

func (devnull) Write(p []byte) (n int, _ error) {
    n = len(p)
    return
}
```

不管如何声明，所有的返回值都会在进入函数前提前初始化成类型的零值。return 语句会在所有 defer 函数之前指定返回值。

实现限制：编译器不允许在覆盖了命名返回值的作用域中直接返回。

```
func f(n int) (res int, err error) {
    if _, err := f(n-1); err != nil {
        return  // invalid return statement: err is shadowed
    }
    return
}
```

break 语句

`break` 语句会在 `for`、`switch` 或 `select` 语句内部退出到相同函数的某个位置。

```
BreakStmt = "break" [ Label ] .
```

如果想指定标签，它必须出现在它所中止的 `for`、`switch` 或 `select` 语句旁。

```
OuterLoop:
    for i = 0; i < n; i++ {
        for j = 0; j < m; j++ {
            switch a[i][j] {
            case nil:
                state = Error
                break OuterLoop
            case item:
                state = Found
                break OuterLoop
            }
        }
    }
```

continue 语句

`continue` 语句会提前 `for` 语句的下一次迭代。`for` 语句必须和 `continue` 在相同函数中。

```
RowLoop:
    for y, row := range rows {
        for x, data := range row {
            if data == endOfRow {
                continue RowLoop
            }
            row[x] = data + bias(x, y)
        }
    }
```

goto 语句

`goto` 会将程序跳转到相同函数的指定标签处。

```
GotoStmt = "goto" Label .
```

```
goto Error
```

goto 语句不允许跳过作用域内程序变量的初始化工作。

```
goto L  // BAD
    v := 3
L:
```

上面的程序是错误的，因为它跳过了变量 v 的初始化过程。

```
if n%2 == 1 {
    goto L1
}
for n > 0 {
    f()
    n--
L1:
    f()
    n--
}
```

标签作用域外的 goto 语句不能跳转到标签处，所以上面的代码是错误的。

Fallthrough 语句

`fallthrough` 语句会跳转到 `switch` 语句中的下一个 case 分句中。它应该只在最后一个非空分句中使用。

```
FallthroughStmt = "fallthrough" .
```

Defer 语句

`defer` 语句会在包裹函数返回后触发函数调用。这里的返回泛指函数因为 return 语句终止、到达函数末尾或者当前 goroutine 触发运行时恐慌。

```
DeferStmt = "defer" Expression .
```

表达式必须是函数或者方法调用；它不能使用括号括起来，调用内置函数会有一些限制。

每次执行 defer 语句执行时都会计算函数的参数和值，但是并不会调用函数。相反，函数的调用是在包裹函数返回后进行，它们的执行顺序与声明顺序正好相反。如果 defer 对应的函数值为 nil，会在调用函数的时候导致运行时恐慌而不是声明 defer 语句的时候。

例如：当 defer 函数为函数字面值且包裹函数具有命名结果值，此时，我们在defer 函数中可以访问和修改命名的结果值。defer 函数的所有返回值都会被忽略。

```
lock(l)
defer unlock(l)  // unlocking happens before surrounding function returns

// prints 3 2 1 0 before surrounding function returns
for i := 0; i <= 3; i++ {
    defer fmt.Print(i)
}

// f returns 1
func f() (result int) {
    defer func() {
        result++
    }()
    return 0
}
```

更多Golang资料包：https://github.com/0voice/Introduction-to-Golang

内置函数

内置函数是预定义的。调用他们和其他函数一样只是他们接受一个类型而不是一个表达式。

内置函数没有标准的 Go 类型，所以他们只能作为调用表达式；而不能作为函数的值。

Close

对于管道类型 `c`，内置函数 `close(c)` 意味着不在有数据插入到管道中。如果 `c` 是一个只接收数据的管道，会发生错误。向已经关闭的发送数据或者重复关闭已经关闭的管道会导致运行时恐慌。关闭 nil 管道会引起运行时恐慌。调用 close 后所有之前发送的数据都能接收到，并且在最后不会阻塞而返回零值。多值的接收操作能够返回接收到的数据和表示管道是否关闭的布尔值。

长度和容积

内置函数 `len` 和 `cap` 可以接收多种类型的参数，并且返回一个 int 类型结果值。函数的实现能够确保结果值不会溢出。

```
Call      Argument type    Result

len(s)    string type      string length in bytes
          [n]T, *[n]T      array length (== n)
          []T              slice length
          map[K]T          map length (number of defined keys)
          chan T           number of elements queued in channel buffer

cap(s)    [n]T, *[n]T      array length (== n)
          []T              slice capacity
          chan T           channel buffer capacity
```

切片的容积底层数组包含的元素个数。在任何情况下都有以下关系：

```
0 <= len(s) <= cap(s)
```

`nil` 切片，map，或者 channel 的长度都为 0。`nil` 切片，管道的容积都为 0。

表达式 `len(x)` 在 `s` 是字符串常量时也为常量。如果 `s` 为数组或者指向数组的指针并且表达式 s 不包含 `channel` 接收器或者函数调用那么 `len(s)` 和 `cap(s)` 也是常量；在这个情况下 s 时不能求值的。其他情况下 `len` 和 `cap` 不是常量并且 `s` 是可以求值的。

```
const (
    c1 = imag(2i)                    // imag(2i) = 2.0 is a constant
    c2 = len([10]float64{2})         // [10]float64{2} contains no function
calls
    c3 = len([10]float64{c1})        // [10]float64{c1} contains no function
calls
    c4 = len([10]float64{imag(2i)})  // imag(2i) is a constant and no function
call is issued
    c5 = len([10]float64{imag(z)})   // invalid: imag(z) is a (non-constant)
function call
)
var z complex128
```

内存分配

内置函数 `new` 接收一个类型 T，它会在运行时给变量分配内存，并且返回一个指向类型 `T` 的 `*T` 类型指针。变量的初始化在初始化值章节中介绍。

```
new(T)
```

例如：

```
type S struct { a int; b float64 }
new(S)
```

给 S 类型的变量分配空间，并初始化它（`a=0`，`b=0.0`），并且返回一个 `*S` 类型值保存变量所在的位置。

创建切片，map 和 管道

内置函数 `make` 以一个类型作为参数，它必须是一个切片，map 或者管道类型，它返回一个 T 类型的值，而不是（*T）类型，它会按初始化值章节描述的方式进行初始化。

```
Call             Type T     Result

make(T, n)       slice      slice of type T with length n and capacity n
make(T, n, m)    slice      slice of type T with length n and capacity m

make(T)          map        map of type T
make(T, n)       map        map of type T with initial space for approximately n
elements

make(T)          channel    unbuffered channel of type T
make(T, n)       channel    buffered channel of type T, buffer size n
```

n 和 m 必须是整数类型或者无类型常量。一个常量参数不能为负数并且该值在 `int` 类型的范围内；如果它是无类型常量，会被转换成 `int` 类型。如果 n 和 m 都是常量，那么 n 必须大于 m。如果 n 是负数或者大于 m 会引发运行时 panic。

```
s := make([]int, 10, 100)       // slice with len(s) == 10, cap(s) == 100
s := make([]int, 1e3)           // slice with len(s) == cap(s) == 1000
s := make([]int, 1<<63)         // illegal: len(s) is not representable by a
value of type int
s := make([]int, 10, 0)         // illegal: len(s) > cap(s)
c := make(chan int, 10)         // channel with a buffer size of 10
m := make(map[string]int, 100)  // map with initial space for approximately 100
elements
```

使用 make 来指定大小初始化 map 类型将会创建一个预留 n 个元素空间的 map 类型。更详细的行为依赖于具体实现。

追加或者拷贝切片

内置函数 `append` 和 `copy` 可以进行切片的通用操作。对于这两个函数，一个是拷贝内存，一个是引用内存。

可变参数的函数 `append` 可以向切片 s 中追加一个或多个 x 值，并返回这个切片。传进 `...T` 的值会根据参数传值。作为特例，`append` 在 s 为 `[]byte` 切片时，可以使用字符串后面跟 `...` 作为参数。

如果 s 的容积容纳不下这些元素，那么 `append` 会分配一个新的足够大的数组。否则会使用原来的底层数组。

```
s0 := []int{0, 0}
s1 := append(s0, 2)                // append a single element     s1 == []int{0,
0, 2}
s2 := append(s1, 3, 5, 7)          // append multiple elements    s2 == []int{0,
0, 2, 3, 5, 7}
s3 := append(s2, s0...)            // append a slice              s3 == []int{0,
0, 2, 3, 5, 7, 0, 0}
s4 := append(s3[3:6], s3[2:]...)   // append overlapping slice    s4 == []int{3,
5, 7, 2, 3, 5, 7, 0, 0}

var t []interface{}
t = append(t, 42, 3.1415, "foo")   //                            t ==
[]interface{}{42, 3.1415, "foo"}

var b []byte
b = append(b, "bar"...)            // append string contents      b ==
[]byte{'b', 'a', 'r' }
```

`copy` 函数从 `src` 拷贝原属到 `dst` 并且返回拷贝元素的个数。参数中所有的元素类型必须是 T 类型或者能转换成 T 的类型。拷贝元素的数量是 `len(src)` 和 `len(dst)` 中的较小值。作为特例，`copy` 可以从 string 类型拷贝元素到 `[]byte` 类型。这会把字符串中的元素拷贝到字节切片中。

```
copy(dst, src []T) int
copy(dst []byte, src string) int
```

例：

```
var a = [...]int{0, 1, 2, 3, 4, 5, 6, 7}
var s = make([]int, 6)
var b = make([]byte, 5)
n1 := copy(s, a[0:])            // n1 == 6, s == []int{0, 1, 2, 3, 4, 5}
n2 := copy(s, s[2:])            // n2 == 4, s == []int{2, 3, 4, 5, 4, 5}
n3 := copy(b, "Hello, World!")  // n3 == 5, b == []byte("Hello")
```

删除 map 中的元素

内置函数 `delete` 移除 map 类型 m 中的键值 k。k 的类型必须是能够转换成 m 键类型的类型。

```
delete(m, k)  // remove element m[k] from map m
```

如果 map 类型 m 是 nil 或者 m[k] 不存在，那么 `delete` 函数不做任何事情。

操作复数

有三个函数可以组装或者分解复数。内置函数 `complex` 会构造一个复数，`real` 和 `imag` 会分解出复数的实部和虚部。

```
complex(realPart, imaginaryPart floatT) complexT
real(complexT) floatT
imag(complexT) floatT
```

参数的类型和返回值类型是对应的。对于 `complex`，两个参数必须是相同的浮点类型，并返回由相同浮点数组成的复数类型。`complex64` 是 `float32` 对应的类型，`complex128` 是 `float64` 对应的参数类型。如果参数是一个无类型常量，它会转换成另一个参数的类型。如果两个参数都是无类型常量，他们必须实数或者虚数部分为零，并且它会返回一个无类型的复数常量。

`real` 和 `imag` 函数和 `complex` 正好相反的，所以对于一个值复数类型 Z 的值 z，`z==Z(complex(real(z),imag(z)))`。

如果这么操作都是常量，那么返回的值也是常量。

```
var a = complex(2, -2)             // complex128
const b = complex(1.0, -1.4)       // untyped complex constant 1 - 1.4i
x := float32(math.Cos(math.Pi/2))  // float32
var c64 = complex(5, -x)           // complex64
var s uint = complex(1, 0)         // untyped complex constant 1 + 0i can be
converted to uint
_ = complex(1, 2<<s)               // illegal: 2 assumes floating-point type,
cannot shift
var rl = real(c64)                 // float32
var im = imag(a)                   // float64
const c = imag(b)                  // untyped constant -1.4
_ = imag(3 << s)                   // illegal: 3 assumes complex type, cannot
shift
```

处理 panic

两个内置函数 `panic` 和 `recover`，可以抛出和处理运行时 `panic` 和程序的错误条件。

```
func panic(interface{})
func recover() interface{}
```

当执行 F 函数时，显式的调用 `panic` 或者运行时 panic 都会中断 F 的执行。但是 F 中的延迟函数还会执行。接下来调用 F 函数处的延迟函数也会执行，一直到顶级的延迟函数。鉴于这点，程序关闭并且错误条件可以抛出。包括 `panic` 中的值。这个顺序叫做 `panicking`。

```
panic(42)
panic("unreachable")
panic(Error("cannot parse"))
```

`recover` 函数允许程序从一个 panicking 中恢复执行。假设函数 G 延迟执行函数 D，在 D 中调用 `recover` 这时如果在 G 执行时发生 panic 会在 D 中恢复。当函数执行到 D，`recover` 的返回值会返回 `panic` 对应的错误，并且终止 `panicking`。在这个情况下 G 函数和 `panic` 之间的代码不会执行。任何在 D 中 G 之前的延迟函数会返回到调用者。

在下面两种情况下 `recover` 会返回 nil：

- panic 的参数为 nil
- 携程里没有发生 panic
- recover 不是在延迟函数中执行

本例中的 `protect` 函数会在 g 发生 panic 的时候恢复执行。

```
func protect(g func()) {
    defer func() {
        log.Println("done")  // Println executes normally even if there is a
panic
        if x := recover(); x != nil {
            log.Printf("run time panic: %v", x)
        }
    }()
    log.Println("start")
    g()
}
```

初始化

这个实现提供了多个内置函数来帮助进行初始化。这些函数用来输出信息但是不确定会一直存在于语言中，他们都没有返回值。

```
Function   Behavior

print      prints all arguments; formatting of arguments is implementation-
specific
println    like print but prints spaces between arguments and a newline at the
end
```

实现限制：`print` 和 `println` 不接受除了布尔值，数字，字符串以外的其他类型。

程序的初始化和执行

零值

当为变量分配内存空间时，不管是声明还是调用 `new` 或者使用字面值和 `make` 初始化，只要创建了一个新值变量都会有一个默认值。这样的元素和值会使用它类型的零值：`false` 是布尔值的零值，0 为数值类型零值，"" 为字符串零值，nil 为指针，函数，接口，切片，频道，字典。初始化会递归完成，所以结构体里的数组中的元素也都会有它自己的零值。

下面两个声明时相等的：

```
var i int
var i int = 0
```

请看下面的声明：

```
type T struct { i int; f float64; next *T }
t := new(T)
t.i == 0
t.f == 0.0
t.next == nil
```

这和下面的声明时同等效果的：

```
var t T
```

包的初始化

保级变量会按声明的顺序进行初始化，如果依赖其他变量，则会在其他变量之后进行初始化。

更确切的说，如果包级变量还没初始化并且没有初始化表达式或者表达式中不包含对其他未初始化变量的依赖，那么会认为它正在等待初始化。初始化过程会从最早声明的变量开始向下一个包级变量重复，直到没有需要初始化的变量。

如果在初始化过程完成后还有未初始化的变量，那么这些变量可能是循环初始化了，这事程序不是合法的。

在多个文件中变量的声明顺序会依据编译时文件出现的顺序：声明在第一个文件中的变量优先于第二个文件中声明的变量，依此类推。

对依赖关系的分析不会根据变量的具体值，它只分析在源码中是否引用了其他变量。例如，如果变量 x 的初始化表达式引用了变量 y 那么 x 就依赖于 y：

- 引用一个变量或者函数中用到了一个变量
- 引用了一个方法值 `m` 或者方法表达式 `t.m` (这里的静态类型 `t` 不是借口类型，并且方法 `m` 是 `t` 方法集中的方法)。`t.m` 的返回值不会在此时影响。
- 变量，函数，或者方法 x 依赖变量 y

依赖分析会在每个包中执行；他只考虑当前包中的析变量，函数，和方法。

例如，给定声明：

```
var (
    a = c + b
    b = f()
    c = f()
    d = 3
)

func f() int {
    d++
    return d
}
```

初始化顺序为 d，b，c，a。

变量可以在包中声明的初始化函数 `init` 中进行初始化，它没有参数和返回值。

```
func init() {}
```

可以为每个包定义多个该函数，甚至在一个文件中也可以。并且不会声明该该标识符。因此 init 函数不能在程序中调用。

还未导入的包会先初始化包级的变量然后按照 `init` 函数在源码中的顺序调用，它可能在包的多个文件中。如果需要导入一个包，它会在初始化自己之前先初始化这个需要导入的包。如果导入一个包多次，那这个包只会初始化一次。导入的包不能存在循环引用。

包的初始化——变量初始化和对 init 函数的调用会按顺序发生在同一个 goroutine 中。 `init` 函数可能会启动其他 goroutine。不过一般 `init` 函数都是按序进行初始化的：它只在上一步已经执行完成时才会调用下一个步骤。

确保初始化行为是可以复现的，构建系统鼓励在同一个包中包含多个文件这些文件在编译器中会以字母排序。

程序执行

一个完整的程序由一个 `main` 包导入所有需要的包。 `main` 包必须以 `main` 作为包名并且声明一个没有参数和返回值的 `main` 函数。

```
func main() {}
```

程序先初始化 `main` 包然后调用 `main` 函数。当 `main` 函数返回时，程序就会退出。它不会等待其他 goroutines 完成。

更多Golang资料包：https://github.com/0voice/Introduction-to-Golang

错误

预定义的错误类型为：

```
type error interface {
    Error() string
}
```

它是表示错误信息的常规接口，nil 代表没有发生错误。例如，在文件中读取数据可以定义为：

```
func Read(f *File, b []byte) (n int, err error)
```

运行时恐慌

运行时错误（例如数组的越界访问）会造成运行时恐慌，它和以 `runtime.Error` 接口实现调用内置的 `panic` 函数一样。`runtime.Error` 满足预定义的 `error` 接口。不同的错误值代表不同的运行时错误条件。

```
package runtime

type Error interface {
    error
    // and perhaps other methods
}
```

系统相关

unsafe 包

`unsafe` 是编译器已知的内置包，可以通过导入路径 `unsafe` 访问包内容，提供 `unsafe` 包目的是支持底层编程（包括操作非 Go 类型的数据结构）。使用 `unsafe` 包必须自己保证类型安全而且它有可能破坏程序的移植性。`unsafe` 包提供了以下接口：

```
package unsafe

type ArbitraryType int  // 任意一个 Go 类型；它不是一个具体的类型。
type Pointer *ArbitraryType

func Alignof(variable ArbitraryType) uintptr
func Offsetof(selector ArbitraryType) uintptr
func Sizeof(variable ArbitraryType) uintptr
```

`Pointer` 是一个指针类型，但是不能解引用 `Pointer` 的值。所有底层类型 `uintptr` 的指针和值都能转换成 `Pointer` 类型，反之亦然。`Pointer` 和 `uintptr` 之间的转换效果由具体实现定义。

```
var f float64
bits = *(*uint64)(unsafe.Pointer(&f))

type ptr unsafe.Pointer
bits = *(*uint64)(ptr(&f))

var p ptr = nil
```

假设变量 v 由 `var v = x` 定义。`Alignof` 以表达式 x 作为参数并返回 x 的对齐字节数。`Sizeof` 以表达式 x 作为参数并返回 x 的大小。

函数 `Offsetof` 以选择器 `s.f`（s 或者 *s 结构体中的 f 字段）作为参数，返回字段相对结构体首地址的位置。如果 f 是一个嵌入字段，那 f 必须可以直接访问（不能通过指针进行间接访问）。对于结构体 s 的 f 字段：

```
uintptr(unsafe.Pointer(&s)) + unsafe.Offsetof(s.f) ==
uintptr(unsafe.Pointer(&s.f))
```

计算机的体系结构要求对齐内存地址（对于一个变量的地址有多种因素影响对齐）。`Alignof` 函数获取一个人和类型的表达式并返回变量对齐的字节数。对于变量 x：

```
uintptr(unsafe.Pointer(&x)) % unsafe.Alignof(x) == 0
```

编译时 `uintptr` 类型常量表达式会调用 `Alignof`，`Offsetof`，和 `Sizeof`。

确定的大小和对齐字节数

对于数字类型，确定有以下尺寸：

```
type                                size in bytes

byte, uint8, int8                    1
uint16, int16                        2
uint32, int32, float32               4
uint64, int64, float64, complex64    8
complex128                          16
```

Go 中规定的最小对齐特性：

1. 对于任意变量类型 x：`unsafe.Alignof(x)` 至少为 1。
2. 对于结构体类型：`unsafe.Alignof(x)` 是所有内部字段 `unsafe.Alignof(x.f)` 的最大值，并且至少为 1。

1. 对于数组类型：`unsafe.Alignof(x)` 和数组元素类型的 alignment 相同。

结构体（数组）在内部没有字段（元素）的时候大小为 0。两个所占空间大小为 0 的不同变量可能在内存中拥有相同地址。

更多Golang资料包：https://github.com/0voice/Introduction-to-Golang