Go 编程语言规范

2014年11月11日版本

简介

这是关于 Go 语言的一个参考手册。如果想了解更多信息或是其他文档的话，可以去 http://golang.org 查看。

Go 是一门通用的编程语言，并时刻将系统编程铭记于心。它是强类型语言，带有垃圾回收机制，而且语言内在地支持并发编程。 程序由 包 组成，包的性质允许高效地管理它们之间的依赖。 已有的语言实现采用了传统的编译/链接模型，最终生成可执行的二进制代码。

Go 语言的语法紧凑，灰常有规则可循，可以很容易地被集成开发环境 (IDE)类的自动工具所分析。

记号

后面的语法使用扩展的巴克斯-诺尔范式 (EBNF)进行描述:

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

产生式由一些术语和下面的几个按优先级从低到高的操作符/运算符组成：

```
|   任选其一
()  一个整体
[]  可选/可有可无 (0 或是 1次)
{}  重复多次 (0 到 n 次)
```

小写的产生式的名字通常用于表示一个词法单元/符号；非终结符一般用驼峰式命名。词法单元/符号我们使用双引号 "" 或是反向单引号 `` 括住或是引住。

a … b 这种形式代表的是从 a 到 b 可选的字符集合。省略号 … 在本规范中也用于某些处的表示不完全枚举或是不再详细列出的代码分片。 字符 … (不同于三个字符的 ...)，它不是 Go 语言的一个词法单元/符号。

源代码表示

源文件是用 UTF-8 编码的 Unicode 文本。文本并不是规范化的，所以，一个加重音的代码点不同于重音再加一个字符，后面的认为是两个字符。 为了简化，本文档使用了并不是很规范的 字符 术语来指代源文本中的一个 Unicode 代码点。

每一个代码点都应该进行区分，比如说，大写字母和小写字母就是不同的字符。

实现限制: 为了和其他的工具兼容，一个编译器不允许在源文本中出现 NUL 字符(U+0000)。

字符

下面是一些比较特殊的 Unicode 字符类：

```
newline        = /* Unicode 代码点 U+000A */ .
unicode_char   = /* 除了 newline 之外的其他 Unicode 代码点 */ .
unicode_letter = /* Unicode 代码点中归为 "字母" 的字符*/ .
unicode_digit  = /* Unicode 代码点中归为 "十进制数字" 的字符 */ .
```

在 Unicode 标准 6.0 中， 4.5 章节 "通用分类" 定义了一系列的分类。Go 会认为在这些分类中的 Lu， Ll， Lt， Lm， or Lo 是 Unicode 字符， Nd 是 Unicode 数字。

字母和数字

下划线 _ (U+005F) 被认定为一个字母。

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

词法元素

注释

有两种形式的注释：

1. 行注释 从两斜杠 // 开始到这一行结束。一个行注释给人的感觉就是一个换行。
2. 通用注释(块注释) 从 /* 开始到 */ 结束。块注释中如果有行注释的话，那么它像是换行；其他情况下，它像是空白。

   注释不能嵌套。

词法单元/符号

词法单元/符号是 Go 语言的词汇表。它们分为四类： 标识符 、 关键字 、 操作符 和 分隔符 。由空格 (U+0020)、水平制表符 (U+0009)、回车符(U+000D) 和换行符 (U+000A)所组成的 空白 除了可能是用于组成符号之外，其他的时候用作分隔符，在分析阶段会被忽略掉。 还有就是，换行符或是页面结束可能导致 分号 的插入。在将代码文本分割成符号的过程中，下一个符号应该是能组成一个合法符号的最长字符序列。

分号

正式语法使用分号 ";" 作为一些产生式的分隔符。 但是 Go 程序可以基于下面两条规则省略多数时候的分号：

1. 当输入文本在被拆成了记号的时候，在一些情况下分号会自动被插入非空白行的尾部的记号流中去，但是需要这一行的最后一个记号是：

   - 标识符

- 整数 、 浮点数 、 虚数 、 分符 ，或是 字符串 值
- 关键字 break 、 continue 、 fallthrough ，或是 return
- 操作符和分隔符 ++ 、 -- ，) 、] ，或是 }

2. 为了允许复杂的语句能够挤在一行中，所以在")" or "}"前面的分号可能省略掉。

   为了反映通常的使用习惯，本文档中的代码例子通常使用这些规则而省略了分号。

标识符

标识符用来命名变量、类型等程序实体。一个标识符实际上就是一个或是多个字母/数字序列，不过第一个字符应该是字母而不能是数字。

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

```
a
_x9
ThisVariableIsExported
αβ
```

有一些标识符是 预声明的 。

关键字

下面的关键字被保留了因而不能作为标识符使用：

```
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/运算符和间隔符

下面的一些字符序列被当做 操作符/运算符 、分隔符或是其他一些特殊的符号：

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

整数值

一个整数值实际就是一串数字组成的 整数常量 。一个可选的前缀表明了这个整数值的基数: 0 表示八进制， 0x 或者 0X 代表十六进制。在十六进制表示中， a-f 或是 A-F 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
```

浮点数值

一个浮点值实际就是由十进制数字所组成的 浮点常量 。它有一个整数部分、小数点、小数部分和指数部分。整数部分和小数部分还是由十进制数字组成； 指数部分是一个 e 或是 E 后面跟一个可选的有符号十进制指数。整数部分或是小数部分二者可以省略其一；小数点和指数部分也可以省略其一。

```
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
```

虚数值

一个(纯)虚数是一个 复数常量， 只不过它只有虚数部分，而虚数部分是用十进制数字表示。它实际就是由一个 浮点常量 或是一个十进制整数后面跟一个小写的字母 i 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
```

分符值

一个分符值就是一个 分符常量， 实际上是一个能标识一个 Unicode 代码点的整数值。一个分符值用一个单引号引住的一个或是多个字符来表示； 在引号中，除了单引号和换行是不允许的，其他的都可以。一个单引号引住的单个字符

也表示这个字符本身的 Unicode 值，在用可变格式中使用反斜杠开始的多字节序列来表示值。

最简单地表示单个字符就是用单引号引住；因为 Go 源文本使用 UTF-8 编码的 Unicode字符，所以多个 UTF-8 字节可能只代表一个整数值。比如说，'a' 它就只有一个字节，表示字符 a（Unicode U+0061），值是 0x61；而'ä' 则用两个字节(0xc3 0xa4)表示带分音的字符 a（U+00E4）， 值是 0xe4。

表示 ASCII 文本的时候可以使用反斜杠来转义值。有四种表示一个整数值，也是整数常量，的方法：\x 后面跟两个十六进制的数字，是两个，不能多也不能少； \u 后面跟四个十六进制数字；\U 后面跟八个十六进制数字；一个普通的反斜杠 \ 后面跟三个八进制数字。不管哪种形式表示，值都是这种表示 所对应的字符的值。

尽管这些表示结果都是一个整数，但是它们之间却有着不同的表示范围。八进制的值能表示 0 到 255 之间的数。十六进制表示必须满足前面说的构造限制。 使用 \u 和 \U 进行转移表示的 Unicode 代码点，在它们中有一些值是不合法的，特别是对于超过 0x10FFFF 的和surrogate halves。

在反斜杠后面某些固定的字符代表一些特殊的值：

```
\a   U+0007 响铃符
\b   U+0008 后退符
\f   U+000C form feed
\n   U+000A 换行符
\r   U+000D 回车符
\t   U+0009 水平制表符
\v   U+000b 垂直制表符
\\   U+005c 反斜杠
\'   U+0027 单引号 （只在分符值中才是合法的转义）
\"   U+0022 双引号 （只在字符串值中是合法的转义）
```

所有其他的在分符值中的以反斜杠开始的转义都是不合法的。

```
char_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'
'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
```

字符串值

一个字符串值就是一系列的字符连接在一起的一个 字符串常量 。它有两种形式：一种是元字符串，一种是解释性字符串。

元字符串就是包括在两个方向单引号`` 内的字符序列。在引号内，除了反向单引号外其他字符都可以包括。一个元字符串的值就是将引号内的所有字符都不加解释滴看成是字符（UTF-8 字符） 而形成的字符串；比如说，不会将反斜杠看成特殊的字符；再者，里面可以包含换行符。元字符串中的回车符在字符串求值的过程中会被忽略掉。

解释性字符串是包括在双引号"" 内的字符序列。双引号内不能包括换行，引号内文本会像分符值一样对反斜杠进行转义，当然限制也一样，就是\' 和 \"这里也是合法的。3 个八进制\ nnn 或是两个十六进制\x nn 都是对单个 字节 的转移表示；而其他的转义形式都指的是（可能是多个字节的） UTF-8编码的单个 字符 。所以，在字符串值中的\377 和 \xFF 都表示的是单个字节，值是 0xFF =255；而 ÿ 、 \u00FF 、 \U000000FF 和两个字节 0xc3 和 0xbf 表示的\xc3\xbf，其实都是对 UTF-8 字符 U+00FF 的表示。

```
string_lit             = raw_string_lit | interpreted_string_lit .
raw_string_lit         = "`" { unicode_char | newline } "`" .
interpreted_string_lit = `"` { unicode_value | byte_value } `"` .
```

```
`abc`  // 等同于 "abc"
`\n
\n`    // 等同于 "\\n\n\\n"
"\n"
""
"Hello, world!\n"
"日本語"
"\u65e5本\U00008a9e"
"\xff\u00FF"
"\uD800"       // illegal: surrogate half
"\U00110000"   // illegal: invalid Unicode code point
```

下面的例子的表示实际上是表示的一个东西：

```
"日本語"                                 // UTF-8 文本
`日本語`                                 // UTF-8 元字分符本
"\u65e5\u672c\u8a9e"                    // 明确的 Unicode 代码点
"\U000065e5\U0000672c\U00008a9e"        // 明确的 Unicode 代码点
"\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e"  // 明确的 Unicode 字节
```

如果源文本使用两个代码点表示一个字符，比如组合一个重音和一个字母，那么如果将这个字符放入到符文值中会有问题（因为它不是一个代码点），而如果放在字符串中它占据两个代码点。

常量

常量有 布尔常量 、 分符常量 、 整型常量 、 浮点常量 、 复数常量 ，和 字符串常量 之分。 字符、整数、浮点数和复数常量 统称数值常量。

常量值会出现在很多地方，比如 文符值 、 整型值 、 浮点值 、 虚数值 ，或是 字符串值 、指代常量的表达式、 常

量表达式 、 常量结果的 转换 ，或是一些内置函数比如 unsafe.Sizeof （可适用于任意类型）、 cap/len （用于 一些表达式 ）、 real/imag （用于复数常量和虚数常量）的返回结果。布尔值可以用预声明的常量 true 和 false 表示，预声明的标识符 iota 也用来表示常量。

一般来说，复数常量是 常量表达式 的一种，所以，偶们在那里讨论。

数值常量表示任意精度的值，不会溢出。

常量可能是 有类型的 或是无类型的。值常量、 true 、 false 、 iota ，和一些只包含无符号操作数的 常量表达式 都是无符号的。

一个常量可能明确地来自于 常量声明 或是某个 转换 ，也可能隐式地出现在 变量声明 中，或是在一个 表达式 中被 赋值 或是作为操作数。如果一个常量不能用它对应的类型来表示，这么这就是个错误。比如说 3.0 既可以当做整数也可以当做浮点数，然而 2147483648.0 (等于 1<<31)可以是 float32 、 float64 或是 uint32 类型，就是不能是 int32 或是 string 类型。

并没有常量能代表 IEEE-754 的无穷大和非数值这两个值，但是 math 包 中的 Inf 、 NaN 、 IsInf 、和 IsNaN 函数可以在运行时来返回或是测试这些值。

实现限制: 尽管我们说数值常量在语言中是任意精度的，但是一个编译器可能在实现的时候在内部只是用有限的精度来表示。不管怎么说，每一个实现必须满足：

- 整型常量至少 256 比特位；
- 浮点常量，也包括复数常量的一部分的浮点常量，至少应该有 256 比特位的尾数和一个 32 比特位的有符号指数部分；
- 如果不能精确地表示某个整数常量，就给出一个错误；
- 如果因为溢出不能表示一个浮点数或是复数常量，就给出一个错误；
- 如果因为精度原因不能表示一个浮点数或是复数常量，那么就四舍五入到最近的那个能表示的常量。

这些需求适用于值常量和 常量表达式 的求值结果。

类型

一种类型决定了一个值的可能的取值范围以及能对这个值所进行的操作。 一个类型由(可能要 限定 的) 类型名 (§ 类型声明) 或是一个 类型文字 指定，它们本身又是由预声明的类型进行构造而得。

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

一些已经命名了的类型，比如布尔类型、数值类型和字符串类型，这些都是 预声明 好的。 复合类型 — 比如数组、结构体、指针、函数、接口、分片、映射和管道类型 — 需要使用已有的类型进行构造。

所谓的一个变量的 静态类型 (或是简单地说 类型)就是它声明时候的类型。接口类型对应的变量有种特殊的 动态类型 之说，也就是说它的实际类型是在运行时候根据存的值所决定的。 动态类型在程序执行的过程中类型可能改变，只要它对于接口定义时所指定的静态类型是 可赋值的 就可以。

每一个类型 T 都有一个 底层类型 ：如果类型 T 是个预定义的布尔、数值、字符串类型或是类型字面量，那么它对应的底层类型就是 T 本身。 否则， T 的底层类型就是它在 类型声明 时所指定的类型的底层类型。

```
type T1 string
type T2 T1
type T3 []T1
type T4 T3
```

string 、 T1 和 T2 的底层类型是 string 。而 []T1 、 T3 和 T4 的底层类型是 []T1 。

方法集

每一种类型都有其对应的 方法集 (§ 接口类型 ， § 方法声明)。 接口类型 的方法集就是它的接口。而任一其他类型，比如 T 的方法集就是所有 T 类型作为接收器的方法的集合。所有指针类型，比如 *T 的方法集 是 *T 或是 T 作为接收器的所有方法的集合，这也说明，指针的方法集包括它的基类型 T 的方法集。 适用于包含匿名字段的结构体的一些规则，窝们在 结构体类型 那里描述。其他类型则有一个空的方法集。在一个方法集中，每一个方法必须有 唯一的 非 空 方法名 。

一个类型的方法集决定了这种类型需要 实现 的接口以及通过这种类型的接收器可以 调用 的方法。

布尔类型

布尔类型 只能在预声明的常量 true 和 false 中取值，它对应的预声明类型是 bool 。

数值类型

数值 type 包括整数值和浮点数值。预声明的与机器无关的数值类型有下面这些：

```
uint8       无符号的 8 位整数 (0 to 255)
uint16      无符号的 16 位整数 (0 to 65535)
uint32      无符号的 32 位整数 (0 to 4294967295)
uint64      无符号的 64 位整数 (0 to 18446744073709551615)

int8        带符号的 8 位整数 (-128 to 127)
int16       带符号的 16 位整数 (-32768 to 32767)
int32       带符号的 32 位整数 (-2147483648 to 2147483647)
int64       带符号的 64 位整数 (-9223372036854775808 to 9223372036854775807)

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

complex64   由 float32 实部和虚部所能组成的复数
complex128  由 float64 实部和虚部所能组成的复数

byte        和 uint8 一样
rune        和 int32 一样
```

一个有 n 位的整数意思是有 n 个比特位宽，并且采用 二进制补码 表示。

下面还有一些预声明的类型，但是它们的长度/尺寸与具体实现有关：

```
uint     32 位或是 64位
int      和 uint 长度一样
uintptr  一个无符号整数类型，它的长度可以容纳下一个指针值
```

为了避免移植性的问题，所以，除了byte和uint8类型一样、 rune和int32类型一样之外，其他的任意两种类型都是互相区分的。 所以，在表达式中或是赋值的时候，只要类型不同就要使用转换；比如说，即使在一个机器上的实现中int32和int实际上是有相同的长度也都是有符号的， 但是在使用的时候也必须进行转换。

字符串类型

字符串类型 代表的是字符串值。字符串用起来像是分片，但是它们是不能修改的；也就是说一个字符串一旦创建，它的内容就是不变的了。 预声明的字符串类型名字是string。

字符串的元素的类型是byte，我们可以通过 索引/下标操作 来访问它们。对字符串的某个元素取地址是不允许的，比如s[i]是第i个元素 ，但&s[i]这样是不行的。字符串s的长度可以使用内置的函数len获得。一个字符串值的长度在编译时实际就已确定了。

数组类型

数组就是某种类型的一个序列，只不过序列中的每个元素都有一个编号。序列的元素的个数叫做长度，不能为负。

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

长度是数组类型的一部分，而且必须是能求出非负整数值的 常量表达式 。数组a的长度可以通过内置函数 len(a) 求得。数组的元素下标从 0 开始计算一直到len(a)-1 (§ 索引)。 一个数组通常是一维的，但是也可以组成多维。

```
[32]byte
[2*N] struct { x, y int32 }
[1000]*float64
[3][5]int
[2][2][2]float64  // 和 [2]([2]([2]float64)) 一个意思
```

分片类型

一个分片就是对一个数组上的连续一段的引用，它也是一个有编号的序列，元素取自某个数组。 一个分片类型指代的是对应元素类型的数组的所有可能的分片。一个未初始化的分片的值是nil.

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

如同数组，分片有个长度，也是通过下标索引访问。分片s 的长度可以通过内置的函数 len(s) 取得；和数组不同的是，长度在执行的过程中可以改变。 元素可以通过下标索引访问，索引的范围从 0 到len(s)-1 (§ 索引)。 对于同一个元素来说，在分片中的索引可能会比对应的底层数组的索引要小。

一个分片，一旦初始化以后，它总是关联着一个容纳其元素的底层数组。所以一个分片和它的数组共享存储，当然也和该数组的其他分片共享；相对的是，两个不同的数组总是代表不同的存储。

一个分片对应的底层数组可以能超出分片的范围。 容量 可以用来说明这种扩展：超过分片的范围但是又在数组范围以内的部分；可以在原来分片(§ 分片)的基础上通过“再分片” 获取一个扩大到数组容量的分片。一个分片a的容量可以通过内置的函数 cap(a) 取得。

一个新的但未初始化的T类型的分片，可以使用内置函数 make 取值，make 带有一个分片的类型和指定长度和容量的参数，容量参数是可选的：

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

对 make 的调用会创建一个新的隐含的数组，分片就是引用的这个数组。这也意味着，执行

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

和下面的执行都分配一个数组并在基础上生成一个分片，生成的两个分片是相同的：

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

如同数组，分片通常是一维的，但是也可以复合构造更高维的对象。对于数组的数组来说，内层的数组的长度在构造的时候总是一样的，然而分片的分片(分片的数组)的长度却是可以变化的。 更进步一地说，内层的分片是单独创建的(使用 make)。

结构体类型

一个结构体就是一个命名的元素序列，每个元素又叫做字段，每个字段都有一个类型和名字。字段的名字可以是明确指定的（标识符列表），也可以能是隐含的（匿名字段）。 在一个结构体内部，只要是非 空白 字段的名字就必须是 唯一的 。

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

```
// 空结构体
struct {}

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

一个字段声明的时候只有类型却没有名字，我们叫它为 匿名字段，或是 嵌入 字段或是类型嵌入。一个嵌入的类型必须指定一个类型名 T 或是一个非接口的指针 *T，而且 T 本身不能是指针类型。这些嵌入类型的类型名作为对应的字段名。

```
// 带有四个匿名字段 T1, *T2, P.T3 和 *P.T4 的结构体
struct {
        T1        // field name is T1
        *T2       // field name is T2
        P.T3      // field name is T3
        *P.T4     // field name is T4
        x, y int  // field names are x and y
}
```

下面的声明是不合法的，因为字段名在结构体中并不唯一：

```
struct {
        T     // 和匿名字段 *T , *P.T 冲突
        *T    // 和匿名字段 T ,*P.T 冲突
        *P.T  // 和匿名字段 T , *T 冲突
}
```

对于结构体 x 的一个匿名字段的字段或是 方法 f，如果 x.f 是一个合法的 选择子 （可能是一个字段或是一个方法 f ），我们就说，它被 提升 了。

提升后的字段用起来就赶脚是结构体的普通字段，只不过它们在结构体的 复合表示 中不能用作字段名。

给一个结构体类型 S 和一个命名类型 T，提升了的方法按照下面所说的包括在结构体的方法集中：

- 如果 S 包括一个匿名字段 T，那么 S 和 *S 的 方法集 都包括以 T 作为接收器而提升的方法. 而 *S 的方法集又包括以 *T 为接收器提升的方法。
- 如果 S 包括一个匿名字段 *T，那么 S 和 *S 的 方法集 都包括以 T 或是 *T 作为接收器而提升的方法.

一个字段的声明中可以跟着一个可选的字符串 标签 ，它在相应的字段声明中会算做字段的一种属性/性质。这些标签在 反射接口 和 类型一致 那里中是可见的，其他的时候可以认为是忽略不计的。

```
// 一个用于时间戳协议缓冲区的结构体
// 标签字符串定义了协议缓冲区字段号
struct {
        microsec  uint64 "field 1"
        serverIP6 uint64 "field 2"
        process   string "field 3"
}
```

指针类型

一个指针就是可以指向其他类型变量的变量。被指向的变量的类型叫做指针的 基类型 ；如果么有初始化的话，指针值是 nil 。

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

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

函数类型

一个函数类型指代的是带有相同参数和返回值类型的一类函数。一个未初始化的函数变量的值是 nil 。

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

在函数的参数/结果列表中，名字（标识符列表）可以都有也可以都么有。如果有的话，一个名字代表对应类型的一项（参数/结果）；如果没有，一个类型代表该类型的一项。参数/结果列表通常 用小括号括起来，不过当只有一个返回

值且没有名字的情况下，这个括号可以省略掉。

一个函数原型/签名的最后一个参数可能以... 为前缀，这样的函数我们叫 可变 函数，它们在调用的时候对于那个参数可以传递 0 或是多个实际值。

```
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

一个接口类型指定了一个称为 接口 的 方法集 。一个接口类型的变量可以存某个类型的值，只要这种类型的方法集是接口方法集的超集；这样的一种类型，我们说它 实现了接口 .一个无初始化的接口的值为nil 。

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

在一个接口类型中对于所有的方法集，每一个方法必须有 唯一的 名字。

```
// 一个简单的文件接口
interface {
        Read(b Buffer) bool
        Write(b Buffer) bool
        Close()
}
```

可以有多种类型都实现了一个接口。比如，如果 S1 和 S2 都有方法集：

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

(其中 T 代表 S1 或是 S2)那么， File 接口就是被 S1 和 S2 都实现了，而我们并不再去关心 S1 和 S2 是否是有其他方法或是共享了其他神马方法。

一个类型实现了一个接口，只要它的所有方法中的一个子集是接口的方法即可；所以，一种类型可能实现了多个接口。比如说，所有的类型都实现了 空接口 :

```
interface{}
```

类似的，考虑下面的接口说明，这个说明出现在 类型声明 中，这个类型声明定义了一个叫做 Lock 的接口:

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

如果S1和S2都实现了

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

那么，它们就实现了Lock接口，当然它们也实现了File接口（看上面）。

一个接口可以使用一个接口类型T来代替一系列方法说明。这样做等价于，我们在接口中一一枚举T类型中的方法。

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


type File interface {
        ReadWrite  // 和一一枚举 ReadWrite 中的方法效果一样
        Lock       // 和一一枚举 Lock 中的方法效果一样
        Close()
}
```

一个接口类型T不能自身嵌套在自身之中，或是递归地嵌套一个包含它自身T的接口。

```
// 非法：不能自身嵌套
type Bad interface {
        Bad
}


// 非法：Bad1 不能通过 Bad2 来嵌套自身
type Bad1 interface {
        Bad2
}
type Bad2 interface {
        Bad1
}
```

映射类型

一个 map/映射是一群无序的元素构成的组，这些元素的类型以一种类型的值作为唯一的索引 key 然后访问到另一种类型的某个值。一个未初始化的映射变量的值为nil。

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

对于 kye/关键字类型的比较运算符== and !=(§ 比较运算符) 必须是完整定义的；于是 key 的类型不能是函数、映射或是分片。 如果 key 的类是一个接口的话，这两个比较运算符应该对动态的两个 key 值是完整定义的；失败会引起一个 运行时问题 。

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

map 元素的个数叫做它的长度。对于一个 map m我们可以通过一个内置函数 len(m) 访问它的长度， 不过它的长度在执行过程中可能会发生变化。元素可以在 赋值 的时候进行添加，可以通过使用 索引/下标 表达式来获取； 我们也可以

使用内置的函数 delete 来删除元素。

一个新的空的 map 值可以使用内置的函数 make 来构造，这个函数带有 map 的类型和一个可选的容量长度作为参数：

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

一个初始过的 map 的容量不受尺寸的限制：map 根据存的东西的多少会自我调整，当然有个例外就是 nil 。 nil map 等价于一个空 map，只不过它还不能添加任何元素。 may be added.

管道类型

管道提供了一种两个并发执行的函数进行同步执行或是通信的机制。管道里面只能传输某一种指定类型的值，初始化的管道的值是 nil 。

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

<- 运算符指定了管道中输出传输的 方向 ： 发送 或是 接收 。如果管道的方向并没有指定的话，那么就认为是 双向的 。 管道经过 装换 或是 赋值 后可能就变成只能发送或是只能接收的了。

```
chan T          // 可以发送或是接收 T 类型的数据
chan<- float64  // 只能发送 float64 数据
<-chan int      // 只能接收 int 数据
```

<- 尽可能地左结合 chan：

```
chan<- chan int    // 等同于 chan<- (chan int)
chan<- <-chan int  // 等同于 chan<- (<-chan int)
<-chan <-chan int  // 等同于 <-chan (<-chan int)
chan (<-chan int)
```

一个新的未初始化的管道可以使用 make 进行构造，构造的时候需要指定管道中数据的类型，而管道的容量则是可选的， 也就是可以指定可以不指定：

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

容量 —— 也就是管道中元素的个数，指定了管道中的缓冲区的大小。如果容量大于 0，那么管道就是异步的，也就是说只有满的时候阻塞发送、空的时候阻塞接收，而其他的时候不阻塞； 当然，元素的接收顺序和发送的顺序一致。如果容量是 0 或是不指定，那么，只有在发送和接收都准备好的时候，通信正常进行，否则都进行阻塞。 一个 nil 管道不能进行通信。

管道可以通过内置的函数 close 进行关闭；而对管道是否关闭的测试可以通过 接收操作符 的多值赋值来实现。

类型以及值的属性

类型一致

两种类型要么它们是 一样的 ，要么它们是 不同的 。

两个命名类型只有它们是相同的 类型说明 。 一个命名类型和一个无名类型总是不同的类型。两个无名类型如果它们对应的类型字面量是一致的，也就是它们要有相同的结构以及每一部分都是相同的，那么它们就是相同类型

。 详述如下：

- 两个数组类型当它们的元素类型以及长度相同时，那么它们是相同类型。
- 两个分片类型只要它们的元素类型相同，那么它们就是相同类型。
- Two struct types are identical if they have the same sequence of fields， and if corresponding fields have the same names， and identical types， and identical tags. Two anonymous fields are considered to have the same name. Lower-case field names from different packages are always different.
- 两个指针类型当它们的基类型是相同的，那么它们就是相同的。
- Two function types are identical if they have the same number of parameters and result values， corresponding parameter and result types are identical， and either both functions are variadic or neither is. Parameter and result names are not required to match.
- Two interface types are identical if they have the same set of methods with the same names and identical function types. Lower-case method names from different packages are always different. The order of the methods is irrelevant.
- 两个map类型当它们有相同的key和value类型，那么它们是相同类型。
- 两个管道类型当它们有相同的值类型以及方向，那么它们是相同类型

给一些声明：

```
type (
        T0 []string
        T1 []string
        T2 struct{ a， b int }
        T3 struct{ a， c int }
        T4 func(int， float64) *T0
        T5 func(x int， y float64) *[]string
)
```

下面这些类型是的等价的：

```
T0 和 T0
[]int 和 []int
struct{ a， b *T5 } 和 struct{ a， b *T5 }
func(x int， y float64) *[]string 和 func(int， float64) (result *[]string)
```

T0 和 T1 是不同的类型，因为它们是不同声明中的命名类型。func(int， float64) *T0 和 func(x int， y float64) *[]string 是不同类型，因为 T0 的类型不同于 []string。

可赋值

只有在下面的情况下，一个值 x 才可以 赋值 给一个 T 类型的变量，或是说 x 对于 T 是可赋值的：

- x 的类型和 T 的类型一样；
- x 的类型 V 和 T 有一样的 底层类型 ，并且 V 和 T 至少有一个不是有名类型；
- T 是一个接口类型，而 x 实现了 接口 T；
- x 是双向管道，而 T 是个管道类型，x 的类型 V 和 T 有相同的元素类型，并且 V 和 T 至少有一个不是有名类型；

- x 是预声明的值 nil ，而 T 是指针、函数、分片、映射、管道或是接口类型；
- x 是一个无类型的 constant ， 可以表示 T 类型的值。

任何一个值都可以赋值给 空标识符 。

块

一个 块 就是放置在一对大括号内的一系列声明和语句。

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

除了源文本中明确的块之外，还有一些不显眼的块：

1. 包围着 Go 源文本的 整体块 ；
2. 任何一个 package 都有一个将包中的所有源文本包住的 包块 。
3. 任何一个文件都有一个将文件中的所有 Go 源文本包住的 文件块 。
4. 每一个 if 、 for 和 switch 语句都认为它们在一个隐含的块之中。
5. 在 switch 或是 select 语句中的每个子句都像是个隐式块。

块可以嵌套而且影响 作用域 。

声明和作用域

A declaration binds a non- blank identifier to a constant， type， variable， function， or package. Every identifier in a program must be declared. No identifier may be declared twice in the same block， and no identifier may be declared in both the file and package block.

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

The scope of a declared identifier is the extent of source text in which the identifier denotes the specified constant， type， variable， function， or package.

Go is lexically scoped using blocks:

1. The scope of a predeclared identifier is the universe block.
2. The scope of an identifier denoting a constant， type， variable， or function (but not method) declared at top level (outside any function) is the package block.
3. The scope of an imported package identifier is the file block of the file containing the import declaration.
4. The scope of an identifier denoting a function parameter or result variable is the function body.
5. The scope of a constant or variable identifier declared inside a function begins at the end of the ConstSpec or VarSpec (ShortVarDecl for short variable declarations) and ends at the end of the innermost containing block.
6. The scope of a type identifier declared inside a function begins at the identifier in the TypeSpec and ends at the end of the innermost containing block.

An identifier declared in a block may be redeclared in an inner block. While the identifier of the inner declaration is in scope， it denotes the entity declared by the inner declaration.

The package clause is not a declaration; the package name does not appear in any scope. Its purpose is to identify the files belonging to the same package and to specify the default package name for import declarations.

Label scopes

Labels are declared by labeled statements and are used in the `break`， `continue`， and `goto` statements (§ Break statements， § Continue statements， § Goto statements). It is illegal to define a label that is never used. In contrast to other identifiers， labels are not block scoped and do not conflict with identifiers that are not labels. The scope of a label is the body of the function in which it is declared and excludes the body of any nested function.

空标识符/通配符

空标识符/通配符， 使用下划线表示 _，它可以像其他标识符一样用在声明之中，不过空标识符在声明中并不会将名字和值的绑定。

预声明的标识符

下面的一些标识符在 通用块 中预声明了:

```
类型:
        bool byte complex64 complex128 error float32 float64
        int int8 int16 int32 int64 rune string
        uint uint8 uint16 uint32 uint64 uintptr

常量:
        true false iota

0 值:
        nil

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

导出的标识符

一个标志符被 导出 后就可以在其他包中使用，但是必须满足下面两个条件：

1. 标识符的首字母是 Unicode 大写字母 (Unicode "Lu" 类); 而且
2. 标识符要在 包块 中进行了声明，或是它是个 字段名 / 方法名 。

而其他所有的标识符都不是导出的。

唯一的标识符

给定一些标识符，如果在这些中某一个 不同 于其他一个，我们就说它是 唯一的 。如果两个标识符拼写都不一样，那么肯定是不同的，或者是它们处于不同的 包 之内而又没有被 导出 ，这也是不同的；除此之外的，就认为是相同的标识符。

常量声明

A constant declaration binds a list of identifiers (the names of the 常量) to the values of a list of 常量表达式. The number of identifiers must be equal to the number of expressions， and the n th identifier on the left is bound to the value of the n th expression on the right.

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

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

If the type is present， all 常量 take the type specified， and the expressions must be assignable to that type. If the type is omitted， the 常量 take the individual types of the corresponding expressions. If the expression values are untyped 常量， the declared 常量 remain untyped and the constant identifiers denote the constant values. For instance， if the expression is a floating-point literal， the constant identifier denotes a floating-point constant， even if the literal's fractional part is zero.

```
const Pi float64 = 3.14159265358979323846
const zero = 0.0         // untyped floating-point constant
const (
        size int64 = 1024
        eof        = -1  // untyped integer constant
)
const a, b, c = 3, 4, "foo"  // a = 3, b = 4, c = "foo", untyped integer and string 常量
const u, v float32 = 0, 3    // u = 0.0, v = 3.0
```

Within a parenthesized `const` declaration list the expression list may be omitted from any but the first declaration. Such an empty list is equivalent to the textual substitution of the first preceding non-empty expression list and its type if any. Omitting the list of expressions is therefore equivalent to repeating the previous list. The number of identifiers must be equal to the number of expressions in the previous list. Together with the `iota` constant generator this mechanism permits light-weight declaration of sequential values:

```
const (
        Sunday = iota
        Monday
        Tuesday
        Wednesday
        Thursday
        Friday
        Partyday
        numberOfDays  // this constant is not exported
)
```

Iota

在 常量声明 中， 预定义标识符 iota 代表了连续的无类型整数 常量. 当在源代码中一个遇到 常量声明 的保留字 const 它就会被置为0，然后依次增加。 它可以用来构造一系列常量：

```
const ( // iota 重置为 0
        c0 = iota  // c0 == 0
        c1 = iota  // c1 == 1
        c2 = iota  // c2 == 2
)

const (
        a = 1 << iota  // a == 1 (iota 又被重置)
        b = 1 << iota  // b == 2
        c = 1 << iota  // c == 4
)

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

在常量列表中， 每一个iota的值是相同的，因为前面说了， 它只会在常量声明之后增加：

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

上面这个例子利用了隐式地最后一个非空表达式的重复。

类型声明

A type declaration binds an identifier， the 类型名， to a new type that has the same underlying type as an existing type. The new type is different from the existing type.

```
TypeDecl     = "type" ( TypeSpec | "(" { TypeSpec ";" } ")" ) .
TypeSpec     = identifier  Type .
```

```
type IntArray [16]int

type (
        Point struct{ x,  y float64 }
        Polar Point
)

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

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

The declared type does not inherit any methods bound to the existing type， but the method set of an interface type or of elements of a composite type remains unchanged:

```
// A Mutex is a data type with two methods,  Lock and Unlock.
type Mutex struct         { /* Mutex fields */ }
func (m *Mutex) Lock()    { /* Lock implementation */ }
func (m *Mutex) Unlock()  { /* Unlock implementation */ }

// NewMutex has the same composition as Mutex but its method set is empty.
type NewMutex Mutex

// The method set of the base type of PtrMutex remains unchanged,
// but the method set of PtrMutex is empty.
type PtrMutex *Mutex

// The method set of *PrintableMutex contains the methods
// Lock and Unlock bound to its anonymous field Mutex.
type PrintableMutex struct {
        Mutex
}

// MyBlock is an interface type that has the same method set as Block.
type MyBlock Block
```

A type declaration may be used to define a different boolean， 数值， or string type and attach methods to it:

```
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

一个变量声明创建一个变量，并绑定一个标识符到该变量；声明的时候需要指定类型，而初始值则是可选的。

```
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 查找；这里只对 "found" 感兴趣
```

If a list of expressions is given， the variables are initialized by assigning the expressions to the variables (§ Assignments) in order; all expressions must be consumed and all variables initialized from them. Otherwise， each variable is initialized to its zero value .

如果类型指定，那么每个变量都是那种类型。否则的话，类型通过对表达式列表进行推断而得。

如果类型没有指定而且对应的表达式求得的是一个 常量 ，那么这个声明的变量取§ 赋值 这里描述的类型。

实现限制: 一个编译器可以不允许在 函数体 内声明变量但是却从来不适用的情况。

短变量声明

可以使用 短变量声明 ：

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

It is a shorthand for a regular variable declaration with initializer expressions but no types:

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

```
i,  j := 0,  10
f := func() int { return 7 }
ch := make(chan int)
r,  w := os.Pipe(fd)  // os.Pipe() returns two values
_,  y,  _ := coord(p)  // coord() returns three values; only interested in y coordinate
```

Unlike regular variable declarations， a short variable declaration may redeclare variables provided they were originally declared in the same block with the same type， and at least one of the non- blank variables is new. As a consequence， redeclaration can only appear in a multi-variable short declaration. Redeclaration does not introduce a new variable; it just assigns a new value to the original.

```
field1,  offset := nextField(str,  0)
field2,  offset := nextField(str,  offset)  // redeclares offset
```

Short variable declarations may appear only inside functions. In some contexts such as the initializers for `if`， `for`， or `switch` statements， they can be used to declare local temporary variables (§ Statements).

函数声明

一个函数声明实际就是将一个标识符，或是说 函数名 和一个具体的函数绑定到一起。

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

一个函数声明可以没有函数体，这样的函数声明实际上只是提供了一个函数调用时的原型/签名，而真正的函数甚至可以是 Go 语言之外的实现，比如一个汇编例程。

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

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

方法声明

A method is a function with a receiver . A method declaration binds an identifier， the method name ， to a method. It also associates the method with the receiver's base type .

```
MethodDecl   = "func" Receiver  MethodName  Signature  [ Body ] .
Receiver     = "(" [ identifier ] [ "*" ] BaseTypeName ")" .
BaseTypeName = identifier .
```

The receiver type must be of the form `T` or `*T` where `T` is a type name. The type denoted by `T` is called the receiver base type ; it must not be a pointer or interface type and it must be declared in the same package as the method. The method is said to be bound to the base type and the method name is visible only within selectors for that type.

For a base type， the non- blank names of methods bound to it must be unique . If the base type is a struct type ， the non-blank method and field names must be distinct.

Given type `Point`， the declarations

```
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
}
```

bind the methods `Length` and `Scale`， with receiver type `*Point`， to the base type `Point`.

If the receiver's value is not referenced inside the body of the method， its identifier may be omitted in the declaration. The same applies in general to parameters of functions and methods.

The type of a method is the type of a function with the receiver as first argument. For instance， the method `Scale` has type

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

这样的函数声明时不能算作一个方法。

表达式

表达式实际指定的是将一系列操作数使用运算符或是函数进行计算求值。

操作符/运算符

Operands denote the elementary values in an expression. An operand may be a literal， a (possibly qualified) identifier denoting a 常量， 变量， or 函数， a 方法表达式 yielding a function， or a parenthesized expression.

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

Qualified identifiers

A qualified identifier is an identifier qualified with a package name prefix. Both the package name and the identifier must not be blank .

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

A qualified identifier accesses an identifier in a different package， which must be imported . The identifier must be exported and declared in the package block of that package.

```
math.Sin       // denotes the Sin function in package math
```

Composite literals

Composite literals construct values for structs， arrays， slices， and maps and create a new value each time they are evaluated. They consist of the type of the value followed by a brace-bound list of composite elements. An element may be a single expression or a key-value pair.

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

The LiteralType must be a struct， array， slice， or map type (the grammar enforces this constraint except when the type is given as a TypeName). The types of the expressions must be assignable to the respective field， element， and key types of the LiteralType; there is no additional conversion. The key is interpreted as a field name for struct literals， an index expression for array and slice literals， and a key for map literals. For map literals， all elements must have a key. It is an error to specify multiple elements with the same field name or constant key value.

For struct literals the following rules apply:

- A key must be a field name declared in the LiteralType.
- A literal that does not contain any keys must list an element for each struct field in the order in which the

fields are declared.

- If any element has a key， every element must have a key.
- A literal that contains keys does not need to have an element for each struct field. Omitted fields get the zero value for that field.
- A literal may omit the element list; such a literal evaluates to the zero value for its type.
- It is an error to specify an element for a non-exported field of a struct belonging to a different package.

Given the declarations

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

one may write

```
origin := Point3D{}                            // zero value for Point3D
line := Line{origin， Point3D{y: -4， z: 12.3}}  // zero value for line.q.x
```

For array and slice literals the following rules apply:

- Each element has an associated integer index marking its position in the array.
- An element with a key uses the key as its index; the key must be a constant integer expression.
- An element without a key uses the previous element's index plus one. If the first element has no key， its index is zero.

Taking the address of a composite literal (§ Address operators) generates a pointer to a unique instance of the literal's value.

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

The length of an array literal is the length specified in the LiteralType. If fewer elements than the length are provided in the literal， the missing elements are set to the zero value for the array element type. It is an error to provide elements with index values outside the index range of the array. The notation `...` specifies an array length equal to the maximum element index plus one.

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

A slice literal describes the entire underlying array literal. Thus， the length and capacity of a slice literal are the maximum element index plus one. A slice literal has the form

```
[]T{x1， x2， … xn}
```

and is a shortcut for a slice operation applied to an array:

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

Within a composite literal of array， slice， or map type `T`， elements that are themselves composite literals may elide the respective literal type if it is identical to the element type of `T`. Similarly， elements that are addresses of composite literals may elide the `&T` when the element type is `*T`.

```
[...]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{{1.5,  -3.5},  {0,  0}}  // same as [...]*Point{&Point{1.5,  -3.5},  &Point{0,  0}}
```

A parsing ambiguity arises when a composite literal using the TypeName form of the LiteralType appears between the keyword and the opening brace of the block of an "if"， "for"， or "switch" statement， because the braces surrounding the expressions in the literal are confused with those introducing the block of statements. To resolve the ambiguity in this rare case， the composite literal must appear within parentheses.

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

Examples of valid array， slice， and map literals:

```
// 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

一个函数字面值代表一个匿名函数。它包括一个函数类型的说明以及一个函数体。

```
FunctionLit  =  FunctionType  Body .
```

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

一个函数字面值可以赋值给一个变量，也可以指直接调用。

```
f := func(x,  y int) int { return x + y }
func(ch chan int) { ch <- ACK }(replyChan)
```

函数字面量可以是闭包 closures : 它访问它周围函数中的变量。这些变量既可以在周围的函数中使用，也可以在这个函数字面量中使用，只要它们在使用着，它们就存在着，也就是说，它们的 生存期可以延长。

主表达式

主表达式指的是那些一元、二元运算符表达式:

```
PrimaryExpr  =
        Operand  |
        Conversion  |
        BuiltinCall |
        PrimaryExpr Selector |
        PrimaryExpr Index |
        PrimaryExpr Slice |
        PrimaryExpr TypeAssertion |
        PrimaryExpr Call .

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

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

选择子

对于 主表达式 x （不是 包名 ）来说， 选择子表达式

```
x.f
```

代表的是x（有时候可能会是*x；见下面）的字段或是方法f。标识符f，不管是字段或是方法，我们都叫它 选择子 ;它必须 不能是 空标识符 。选择子表达式的类型就是f的类型。如果x是个包名的话，你还是看 限定标识符 这里吧。

选择子f可能指代的就是类型T的字段或是方法f，它也可能指代的是T的一个 匿名字段 的字段或是方法f。 访问到f所要经过的匿名字段的数量叫做在T中的 深度 。如果字段f在T中声明，那么它的深度就是0。在T中的字段或是方法 f的深度是f在匿名字段A（在T声明）中的深度再加 1。

对选择子有下面一些规则：

1. 对已一个T类型或是一个指针*T （T不是接口类型）的值x， x.f代表的是在T中的最浅层次的字段或是方法，它们之中有一个f。如果在最浅层次上没有精确的 一个f ，那么选择子表达式就是不合法的。
2. 对于一个I类型（I是一个接口）的值x 那么 x.f指代的是赋值给x的实际的f名字的方法。如果在I的 方法集中没有一个名字为 method set 的方法，那么这个选择子表达式就是不合法的。
3. 其他情况下， x.f都是不合法的。
4. 如果x是个指针或是接口类型，但是值却是nil，不管是赋值、计算值或是调用x.f都引起一个 运行时问题 。

选择子会自动 解析 指向结构体的指针。如果x是个一个结构体指针，那么x.y就代表(*x).y;如果y还是一个 结构体指针，那么x.y.z代表(*(*x).y).z，以此类推。如果x包括一个匿名字段类型*A，而A又是一个结构体类型，那么 x.f代表的是(*x.A).f。

举个例子，给个声明：

```
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
```

下面的都是合法的：

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

但下面的是不合法的：

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

索引/下标

一个有下面形式的主表达式：

```
a[x]
```

用来访问数组、分片、字符串或是映射 map a 中的以 x 为下标/索引的元素。值 x 被叫做下标/索引 index 或是 map 键 。对于它们有下面的规则：

假设 A 是一个 数组类型 ，那么对于 A 或是 *A 的 a ，或是对于是 分片类型 S 的变量 a ，那么

- x 的值必须是整型，且 0 <= x < len(a)
- a[x] 是下标位于 x 的元素的值， a[x] 的类型是 A 的元素类型
- 如果 a 是 nil 或是下标 x 越界，那么会有一个 运行时问题 出现

如果 T 是一个 string类型 ，那么对于 T 类型的 a 来说：

- x 的值必须是整型，且 0 <= x < len(a)
- a[x] 是下表是 x 的字节， a[x] 类型是 byte
- a[x] 是不能赋值的
- 如果下标 x 越界，那么会有一个 运行时问题 出现

如果 M 是 map 类型 ，那么对于 M 类型的 a 来说：

- x 对于 M 的key类型来说必须是 可赋值的
- 如果map结构包含key x ，那么 a[x] 是key x 对应的value值， a[x] 类型是 M 的value类型
- 如果 nil 或是不包含对应的key，那么 a[x] 是 M 对应value类型的 0 值

除了上面的情况之外， a[x] 都是不合法的。

对于类型 map[K]V 的map a 来说，下标表达式可以以一种特殊的形式用在赋值或是初始化中：

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

在这种形式中，下表表达式的结果是一个类型为 (V, bool) pair对的值。 如果key x 是存在的，那么 ok 的值是 true ，否则是 false ； v 的值就是 a[x] 作为单个结果的值。

对于一个 nil map的元素的赋值，会引起 运行时错误 。

分片

对于一个string、数组、数组指针或是一个分片 a ，主表达式

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

实际上构造了一个子string或是分片。下标表达式 low 和 high 决定了哪些元素出现在结果中。结果是索引或是下标从 low 开始长度为 high - low 。在堆数组 a 进行分片之后

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

分片 s 类型 []int ，长度为3，容量为4， 对应的元素是：

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

为了方便起见，下标表达式的前后两部分都可以省略，缺省的 low 默认是0，缺省的 high 默认是分片的长度：

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

对于数组或是字符串来说，下标 low 和 high 必须满足 0 <= low <= high <= 长度; 对于分片来说，访问上界不是长度而是容量大小。

如果分片的对象是字符串或是分片类型，那么分片的结果还是对应的字符串或是分片类型；如果分片的对象是数组，那么它必须是 可寻址的 ，分片的结果是一个分片类型，元素的类型和数组的元素类型一致。

类型断言

For an expression x of interface type and a type T， the primary expression

```
x.(T)
```

asserts that x is not nil and that the value stored in x is of type T. The notation x.(T) is called a type assertion .

More precisely， if T is not an interface type， x.(T) asserts that the dynamic type of x is identical to the type T. If T is an interface type， x.(T) asserts that the dynamic type of x implements the interface T (§ Interface types).

If the type assertion holds， the value of the expression is the value stored in x and its type is T. If the type assertion is false， a run-time panic occurs. In other words， even though the dynamic type of x is known only at run-time， the type of x.(T) is known to be T in a correct program.

If a type assertion is used in an assignment or initialization of the form

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

the result of the assertion is a pair of values with types (T， bool). If the assertion holds， the expression returns the pair (x.(T)， true); otherwise， the expression returns (Z， false) where Z is the zero value for type T. No run-time panic occurs in this case. The type assertion in this construct thus acts like a function call returning a value and a boolean indicating success. (§ Assignments)

Calls

一个函数类型 F 的一个表达式 f，

```
f(a1， a2， … an)
```

调用 f ，带有参数 a1， a2， … an 。除了一个特殊的情况，参数都是单值表达式，可以 赋值 给 F 的参数，而且是在函数调用之前求值。 表达式的类型就是 F 的返回值类型。一个方法的调用也是类似的，只不过方法要指定一个选择子作为该方法的接收器。

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

In a function call, the function value and arguments are evaluated in the usual order . After they are evaluated, the parameters of the call are passed by value to the function and the called function begins execution. The return parameters of the function are passed by value back to the calling function when the function returns.

Calling a `nil` function value causes a run-time panic .

As a special case, if the return parameters of a function or method `g` are equal in number and individually assignable to the parameters of another function or method `f`, then the call `f(g(parameters_of_g))` will invoke `f` after binding the return values of `g` to the parameters of `f` in order. The call of `f` must contain no parameters other than the call of `g`. If `f` has a final `...` parameter, it is assigned the return values of `g` that remain after assignment of regular parameters.

```
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")
}
```

A method call `x.m()` is valid if the method set of (the type of) `x` contains `m` and the argument list can be assigned to the parameter list of `m`. If `x` is addressable and `&x`'s method set contains `m`, `x.m()` is shorthand for `(&x).m()`:

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

There is no distinct method type and there are no method literals.

Passing arguments to `...` parameters

If `f` is variadic with final parameter type `...T`, then within the function the argument is equivalent to a parameter of type `[]T`. At each call of `f`, the argument passed to the final parameter is a new slice of type `[]T` whose successive elements are the actual arguments, which all must be assignable to the type `T`. The length of the slice is therefore the number of arguments bound to the final parameter and may differ for each call site.

Given the function and call

```
func Greeting(prefix string, who ...string)
Greeting("hello:", "Joe", "Anna", "Eileen")
```

within `Greeting`, `who` will have the value `[]string{"Joe", "Anna", "Eileen"}`

If the final argument is assignable to a slice type `[]T`, it may be passed unchanged as the value for a `...T` parameter if the argument is followed by `...`. In this case no new slice is created.

Given the slice s and call

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

within Greeting, who will have the same value as s with the same underlying array.

Operators

Operators combine operands into expressions.

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

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

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

Comparisons are discussed elsewhere . For other binary operators, the operand types must be identical unless the operation involves shifts or untyped 常量 . For operations involving 常量 only, see the section on constant expressions .

Except for shift operations, if one operand is an untyped constant and the other operand is not, the constant is converted to the type of the other operand.

The right operand in a shift expression must have unsigned integer type or be an untyped constant that can be converted to unsigned integer type. If the left operand of a non-constant shift expression is an untyped constant, the type of the constant is what it would be if the shift expression were replaced by its left operand alone; the type is int if it cannot be determined from the context (for instance, if the shift expression is an operand in a comparison against an untyped constant).

```
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
var n = 1.0<<s != 0    // 1.0 has type int; n == false if ints are 32bits in size
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 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
```

Operator precedence

Unary operators have the highest precedence. As the ++ and -- operators form statements, not expressions, they fall outside the operator hierarchy. As a consequence, statement *p++ is the same as (*p)++.

There are five precedence levels for binary operators. Multiplication operators bind strongest, followed by addition operators， comparison operators， `&&` (logical and)， and finally `||` (logical or):

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

Binary operators of the same precedence associate from left to right. For instance， `x / y * z` is the same as `(x / y) * z`.

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

Arithmetic operators

Arithmetic operators apply to 数值 values and yield a result of the same type as the first operand. The four standard arithmetic operators (`+`， `-`， `*`， `/`) apply to integer， floating-point， and complex types; `+` also applies to strings. All other arithmetic operators apply to integers only.

`+`	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	

Strings can be concatenated using the `+` operator or the `+=` assignment operator:

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

String addition creates a new string by concatenating the operands.

For two integer values `x` and `y`， the integer quotient `q = x / y` and remainder `r = x % y` satisfy the following relationships:

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

with `x / y` truncated towards zero ("truncated division").

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

As an exception to this rule， if the dividend `x` is the most negative value for the int type of `x`， the quotient `q = x / -1` is equal to `x` (and `r = 0`).

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

If the divisor is zero， a run-time panic occurs. If the dividend is positive and the divisor is a constant power of 2， the division may be replaced by a right shift， and computing the remainder may be replaced by a bitwise "and" operation:

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

The shift operators shift the left operand by the shift count specified by the right operand. They implement arithmetic shifts if the left operand is a signed integer and logical shifts if it is an unsigned integer. There is no upper limit on the shift count. Shifts behave as if the left operand is shifted `n` times by 1 for a shift count of `n`. As a result， `x << 1` is the same as `x*2` and `x >> 1` is the same as `x/2` but truncated towards negative infinity.

For integer operands， the unary operators `+`， `-`， and `^` are defined as follows:

```
+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
```

For floating-point and complex numbers， `+x` is the same as `x`， while `-x` is the negation of `x`. The result of a floating-point or complex division by zero is not specified beyond the IEEE-754 standard; whether a run-time panic occurs is implementation-specific.

Integer overflow

For unsigned integer values， the operations `+`， `-`， `*`， and `<<` are computed modulo 2^n， where n is the bit width of the unsigned integer's type (§ Numeric types). Loosely speaking， these unsigned integer operations discard high bits upon overflow， and programs may rely on ``wrap around".

For signed integers， the operations `+`， `-`， `*`， and `<<` may legally overflow and the resulting value exists and is deterministically defined by the signed integer representation， the operation， and its operands. No exception is raised as a result of overflow. A compiler may not optimize code under the assumption that overflow does not occur. For instance， it may not assume that `x < x + 1` is always true.

Comparison operators

Comparison operators compare two operands and yield a boolean value.

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

In any comparison， the first operand must be assignable to the type of the second operand， or vice versa.

The equality operators `==` and `!=` apply to operands that are comparable . The ordering operators `<`， `<=`， `>`， and `>=` apply to operands that are ordered . These terms and the result of the comparisons are defined as follows:

- Boolean values are comparable. Two boolean values are equal if they are either both `true` or both `false`.
- Integer values are comparable and ordered， in the usual way.
- Floating point values are comparable and ordered， as defined by the IEEE-754 standard.
- Complex values are comparable. Two complex values `u` and `v` are equal if both `real(u) == real(v)` and `imag(u) == imag(v)`.
- String values are comparable and ordered， lexically byte-wise.
- Pointer values are comparable. Two pointer values are equal if they point to the same variable or if both have value `nil`. Pointers to distinct zero-size variables may or may not be equal.
- Channel values are comparable. Two channel values are equal if they were created by the same call to `make` (§ Making slices， maps， and channels) or if both have value `nil`.
- Interface values are comparable. Two interface values are equal if they have identical dynamic types and equal dynamic values or if both have value `nil`.
- A value `x` of non-interface type `X` and a value `t` of interface type `T` are comparable when values of type `X` are comparable and `X` implements `T`. They are equal if `t`'s dynamic type is identical to `X` and `t`'s dynamic value is equal to `x`.
- Struct values are comparable if all their fields are comparable. Two struct values are equal if their corresponding non- blank fields are equal.
- Array values are comparable if values of the array element type are comparable. Two array values are equal if their corresponding elements are equal.

A comparison of two interface values with identical dynamic types causes a run-time panic if values of that type are not comparable. This behavior applies not only to direct interface value comparisons but also when comparing arrays of interface values or structs with interface-valued fields.

Slice， map， and function values are not comparable. However， as a special case， a slice， map， or function value may be compared to the predeclared identifier `nil`. Comparison of pointer， channel， and interface values to `nil` is also allowed and follows from the general rules above.

The result of a comparison can be assigned to any boolean type. If the context does not demand a specific boolean type， the result has type `bool`.

```
type MyBool bool
```

```
var x,  y int
var (
        b1 MyBool = x == y // result of comparison has type MyBool
        b2 bool   = x == y // result of comparison has type bool
        b3        = x == y // result of comparison has type bool
)
```

Logical operators

Logical operators apply to boolean values and yield a result of the same type as the operands. The right operand is evaluated conditionally.

```
&&    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"
```

Address operators

For an operand `x` of type `T`， the address operation `&x` generates a pointer of type `*T` to `x`. The operand must be addressable， that is， either a variable， pointer indirection， or slice indexing operation; or a field selector of an addressable struct operand; or an array indexing operation of an addressable array. As an exception to the addressability requirement， `x` may also be a composite literal.

For an operand `x` of pointer type `*T`， the pointer indirection `*x` denotes the value of type `T` pointed to by `x`. If `x` is `nil`， an attempt to evaluate `*x` will cause a run-time panic.

```
&x
&a[f(2)]
*p
*pf(x)
```

Receive operator

For an operand `ch` of channel type， the value of the receive operation `<-ch` is the value received from the channel `ch`. The type of the value is the element type of the channel. The expression blocks until a value is available. Receiving from a `nil` channel blocks forever. Receiving from a closed channel always succeeds， immediately returning the element type's zero value.

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

A receive expression used in an assignment or initialization of the form

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

yields an additional result of type `bool` reporting whether the communication succeeded. The value of `ok` is `true` if the value received was delivered by a successful send operation to the channel， or `false` if it is a zero value generated because the channel is closed and empty.

Method expressions

If `M` is in the method set of type `T`, `T.M` is a function that is callable as a regular function with the same arguments as `M` prefixed by an additional argument that is the receiver of the method.

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

Consider a struct type `T` with two methods, `Mv`, whose receiver is of type `T`, and `Mp`, whose receiver is of type `*T`.

```
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
```

The expression

```
T.Mv
```

yields a function equivalent to `Mv` but with an explicit receiver as its first argument; it has signature

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

That function may be called normally with an explicit receiver, so these three invocations are equivalent:

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

Similarly, the expression

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

yields a function value representing `Mp` with signature

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

For a method with a value receiver, one can derive a function with an explicit pointer receiver, so

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

yields a function value representing `Mv` with signature

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

Such a function indirects through the receiver to create a value to pass as the receiver to the underlying method; the method does not overwrite the value whose address is passed in the function call.

The final case, a value-receiver function for a pointer-receiver method, is illegal because pointer-receiver methods are not in the method set of the value type.

Function values derived from methods are called with function call syntax; the receiver is provided as the first argument to the call. That is, given `f := T.Mv`, `f` is invoked as `f(t, 7)` not `t.f(7)`. To construct a function that binds the receiver, use a closure.

It is legal to derive a function value from a method of an interface type. The resulting function takes an explicit receiver of that interface type.

Conversions

Conversions are expressions of the form `T(x)` where `T` is a type and `x` is an expression that can be converted to type `T`.

```
Conversion = Type "(" Expression ")" .
```

If the type starts with an operator it must be parenthesized:

```
*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)
```

A constant value `x` can be converted to type `T` in any of these cases:

- `x` is representable by a value of type `T`.
- `x` is an integer constant and `T` is a string type. The same rule as for non-constant `x` applies in this case (§ Conversions to and from a string type).

Converting a constant yields a typed constant as result.

```
uint(iota)               // iota value of type uint
float32(2.718281828)     // 2.718281828 of type float32
complex128(1)            // 1.0 + 0.0i of type complex128
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, 数值, 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
```

A non-constant value `x` can be converted to type `T` in any of these cases:

- `x` is assignable to `T`.
- `x`'s type and `T` have identical underlying types.
- `x`'s type and `T` are unnamed pointer types and their pointer base types have identical underlying types.
- `x`'s type and `T` are both integer or floating point types.
- `x`'s type and `T` are both complex types.
- `x` is an integer or has type `[]byte` or `[]rune` and `T` is a string type.
- `x` is a string and `T` is `[]byte` or `[]rune`.

Specific rules apply to (non-constant) conversions between 数值 types or to and from a string type. These conversions may change the representation of `x` and incur a run-time cost. All other conversions only change the type but not the representation of `x`.

There is no linguistic mechanism to convert between pointers and integers. The package `unsafe` implements this functionality under restricted circumstances.

Conversions between 数值 types

For the conversion of non-constant 数值 values， the following rules apply:

1. When converting between integer types， if the value is a signed integer， it is sign extended to implicit infinite precision; otherwise it is zero extended. It is then truncated to fit in the result type's size. For example， if `v := uint16(0x10F0)`， then `uint32(int8(v)) == 0xFFFFFFF0`. The conversion always yields a valid value; there is no indication of overflow.
2. When converting a floating-point number to an integer， the fraction is discarded (truncation towards zero).
3. When converting an integer or floating-point number to a floating-point type， or a complex number to another complex type， the result value is rounded to the precision specified by the destination type. For instance， the value of a variable `x` of type `float32` may be stored using additional precision beyond that of an IEEE-754 32-bit number， but float32(x) represents the result of rounding `x`'s value to 32-bit precision. Similarly， `x + 0.1` may use more than 32 bits of precision， but `float32(x + 0.1)` does not.

In all non-constant conversions involving floating-point or complex values， if the result type cannot represent the value the conversion succeeds but the result value is implementation-dependent.

Conversions to and from a string type

1. Converting a signed or unsigned integer value to a string type yields a string containing the UTF-8 representation of the integer. Values outside the range of valid Unicode code points are converted to `"\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"
```

2. Converting a slice of bytes to a string type yields a string whose successive bytes are the elements of the slice. If the slice value is `nil`， the result is the empty string.

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

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

3. Converting a slice of runes to a string type yields a string that is the concatenation of the individual rune values converted to strings. If the slice value is `nil`， the result is the empty string.

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

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

4. Converting a value of a string type to a slice of bytes type yields a slice whose successive elements are the bytes of the string. If the string is empty， the result is `[]byte(nil)`.

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

5. Converting a value of a string type to a slice of runes type yields a slice containing the individual Unicode code points of the string. If the string is empty， the result is `[]rune(nil)`.

```
[]rune(MyString("白鵬翔"))  // []rune{0x767d, 0x9d6c, 0x7fd4}
MyRunes("白鵬翔")           // []rune{0x767d, 0x9d6c, 0x7fd4}
```

Constant expressions

Constant expressions may contain only constant operands and are evaluated at compile-time.

Untyped boolean， 数值， and string 常量 may be used as operands wherever it is legal to use an operand of boolean， 数值， or string type， respectively. Except for shift operations， if the operands of a binary operation are different kinds of untyped 常量， the operation and， for non-boolean operations， the result use the kind that appears later in this list: integer， rune， floating-point， complex. For example， an untyped integer constant divided by an untyped complex constant yields an untyped complex constant.

A constant comparison always yields an untyped boolean constant. If the left operand of a constant shift expression is an untyped constant， the result is an integer constant; otherwise it is a constant of the same type as the left operand， which must be of integer type (§ Arithmetic operators). Applying all other operators to untyped 常量 results in an untyped constant of the same kind (that is， a boolean， integer， floating-point， complex， or string constant).

```
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 == 1.5   (type float64)
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   // f == 0     (type 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)
```

Applying the built-in function `complex` to untyped integer， rune， or 浮点常量 yields an untyped complex constant.

```
const ic = complex(0,  c)   // ic == 3.75i (untyped complex constant)
const iΘ = complex(0,  Θ)   // iΘ == 1.5i  (type complex128)
```

Constant expressions are always evaluated exactly; intermediate values and the 常量 themselves may require precision significantly larger than supported by any predeclared type in the language. The following are legal declarations:

```
const Huge = 1 << 100
const Four int8 = Huge >> 98
```

The values of typed 常量 must always be accurately representable as values of the constant type. The following constant expressions are illegal:

```
uint(-1)     // -1 cannot be represented as a uint
int(3.14)    // 3.14 cannot be represented as an int
int64(Huge)  // 1<<100 cannot be represented as an int64
Four * 300   // 300 cannot be represented as an int8
Four * 100   // 400 cannot be represented as an int8
```

The mask used by the unary bitwise complement operator ^ matches the rule for non-常量: the mask is all 1s for unsigned 常量 and -1 for signed and untyped 常量.

```
^1         // untyped integer constant,  equal to -2
uint8(^1)  // error,  same as uint8(-2),  out of range
^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
```

Implementation restriction: A compiler may use rounding while computing untyped floating-point or complex constant expressions; see the implementation restriction in the section on 常量 . This rounding may cause a floating-point constant expression to be invalid in an integer context， even if it would be integral when calculated using infinite precision.

Order of evaluation

When evaluating the operands of an expression， assignment， or return statement， all function calls， method calls， and communication operations are evaluated in lexical left-to-right order.

For example， in the assignment

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

the function calls and communication happen in the order `f()`， `h()`， `i()`， `j()`， `<-c`， `g()`， and `k()`. However， the order of those events compared to the evaluation and indexing of `x` and the evaluation of `y` is not specified.

```
a := 1
f := func() int { a = 2; return 3 }
x := []int{a,  f()}  // x may be [1,  3] or [2,  3]: evaluation order between a and f() is not specified
```

Floating-point operations within a single expression are evaluated according to the associativity of the operators. Explicit parentheses affect the evaluation by overriding the default associativity. In the expression `x + (y + z)` the addition `y + z` is performed before adding `x`.

语句

语句控制了程序的执行。

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

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

空语句

空语句神马也不做。

```
EmptyStmt  = .
```

标号语句

一个标号语句可能会被 goto 、 break 或是 continue 使用到。

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

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

表达式语句

函数调用、方法调用和接收操作可以出现在语句中；有的时候可能会用到括号。

```
ExpressionStmt  =  Expression  .
```

```
h(x+y)
f.Close()
<-ch
(<-ch)
```

发送语句

一个发送语句向管道中送入一个值。管道表达式当然必须是 管道类型 ，而值必须对于管道中的元素类型来说是 可赋值的 。

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

管道和求值都是发生在通信之前。在发送可以开始之前，通信是处于阻塞状态；向一个无缓冲管道发送数据，只有接收准备就绪时发送才正常进行；而向一个有缓冲 管道发送数据，只要缓冲区有空间发送便可以进行。如果向一个已经关闭了的管道发送数据，会导致一个 运行时问题 。向 nil 中发送一个数据会导致永远阻塞。

```
ch <- 3
```

自加自减语句

"++"和"--"语句对操作数增加或是减少一个无类型的 常量 1。为了赋值，要求操作数必须是 可寻址的 或是一个 map 的下标表达式。

```
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 ] "=" .
```

每一个左边的操作数必须是 可寻址的 或是一个 map 的索引索引表达式，或是 空标识符 。操作数也许会带有括号。

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

An assignment operation `x op = y` where op is a binary arithmetic operation is equivalent to `x = x op y` but evaluates `x` only once. The op = construct is a single token. In assignment operations， both the left- and right-hand expression lists must contain exactly one single-valued expression.

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

A tuple assignment assigns the individual elements of a multi-valued operation to a list of variables. There are two forms. In the first， the right hand operand is a single multi-valued expression such as a function evaluation or channel or map operation or a type assertion . The number of operands on the left hand side must match the number of values. For instance， if `f` is a function returning two values，

```
x， y = f()
```

assigns the first value to `x` and the second to `y`. The blank identifier provides a way to ignore values returned by a multi-valued expression:

```
x， _ = f()  // ignore second value returned by f()
```

In the second form， the number of operands on the left must equal the number of expressions on the right， each of which must be single-valued， and the n th expression on the right is assigned to the n th operand on the left.

The assignment proceeds in two phases. First， the operands of index expressions and pointer indirections (including implicit pointer indirections in selectors) on the left and the expressions on the right are all evaluated in the usual order . Second， the assignments are carried out in left-to-right order.

```
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}
```

In assignments, each value must be assignable to the type of the operand to which it is assigned. If an untyped constant is assigned to a variable of interface type, the constant is converted to type `bool`, `rune`, `int`, `float64`, `complex128` or `string` respectively, depending on whether the value is a boolean, rune, integer, floating-point, complex, or string constant.

if 语句

if 语句会根据一个布尔表达式的结果进行条件执行；如果布尔表达式求得 true，那么 if 分支执行，否则的话，有 else 的话就执行 else 分支，么有的话就罢了。

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

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

表达式可能会带有一个前置语句，前置语句会在表达式之前进行计算。

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

switch 语句

"Switch" statements provide multi-way execution. An expression or type specifier is compared to the "cases" inside the "switch" to determine which branch to execute.

```
SwitchStmt = ExprSwitchStmt | TypeSwitchStmt .
```

There are two forms: expression switches and type switches. In an expression switch， the cases contain expressions that are compared against the value of the switch expression. In a type switch， the cases contain types that are compared against the type of a specially annotated switch expression.

表达式分支

In an expression switch， the switch expression is evaluated and the case expressions， which need not be 常量， are evaluated left-to-right and top-to-bottom; the first one that equals the switch expression triggers execution of the statements of the associated case; the other cases are skipped. If no case matches and there is a "default" case， its statements are executed. There can be at most one default case and it may appear anywhere in the "switch" statement. A missing switch expression is equivalent to the expression `true`.

```
ExprSwitchStmt = "switch" [ SimpleStmt ";" ] [ Expression ] "{" { ExprCaseClause } "}" .
ExprCaseClause = ExprSwitchCase ":" { Statement ";" } .
ExprSwitchCase = "case" ExpressionList | "default" .
```

In a case or default clause， the last statement only may be a "fallthrough" statement (§ Fallthrough statement) to indicate that control should flow from the end of this clause to the first statement of the next clause. Otherwise control flows to the end of the "switch" statement.

The expression may be preceded by a simple statement， which executes before the expression is evaluated.

```
switch tag {
default: s3()
case 0， 1， 2， 3: s1()
case 4， 5， 6， 7: s2()
}

switch x := f(); { // missing switch expression means "true"
case x < 0: return -x
default: return x
}

switch {
case x < y: f1()
case x < z: f2()
case x == 4: f3()
}
```

类型分支

A type switch compares types rather than values. It is otherwise similar to an expression switch. It is marked by a special switch expression that has the form of a type assertion using the reserved word `type` rather than an actual type. Cases then match literal types against the dynamic type of the expression in the type assertion.

```
TypeSwitchStmt  = "switch" [ SimpleStmt ";" ] TypeSwitchGuard "{" { TypeCaseClause } "}" .
TypeSwitchGuard = [ identifier ":=" ] PrimaryExpr "." "(" "type" ")" .
TypeCaseClause  = TypeSwitchCase ":" { Statement ";" } .
```

```
TypeSwitchCase   = "case" TypeList | "default" .
TypeList         = Type { "," Type } .
```

The TypeSwitchGuard may include a short variable declaration . When that form is used， the variable is declared at the beginning of the implicit block in each clause. In clauses with a case listing exactly one type， the variable has that type; otherwise， the variable has the type of the expression in the TypeSwitchGuard.

The type in a case may be `nil` (§ Predeclared identifiers); that case is used when the expression in the TypeSwitchGuard is a `nil` interface value.

Given an expression `x` of type `interface{}`， the following type switch:

```
switch i := x.(type) {
case nil:
        printString("x is nil")
case int:
        printInt(i)  // i is an int
case float64:
        printFloat64(i)  // i is a float64
case func(int) float64:
        printFunction(i)  // i is a function
case bool， string:
        printString("type is bool or string")  // i is an interface{}
default:
        printString("don't know the type")
}
```

could be rewritten:

```
v := x  // x is evaluated exactly once
if v == nil {
        printString("x is nil")
} else if i， isInt := v.(int); isInt {
        printInt(i)  // i is an int
} else if i， isFloat64 := v.(float64); isFloat64 {
        printFloat64(i)  // i is a float64
} else if i， isFunc := v.(func(int) float64); isFunc {
        printFunction(i)  // i is a function
} else {
        i1， isBool := v.(bool)
        i2， isString := v.(string)
        if isBool || isString {
                i := v
                printString("type is bool or string")  // i is an interface{}
        } else {
                i := v
                printString("don't know the type")  // i is an interface{}
        }
}
```

The type switch guard may be preceded by a simple statement， which executes before the guard is evaluated.

在类型分支中是不允许存在 fallthrough 语句的。

for 语句

A "for" statement specifies repeated execution of a block. The iteration is controlled by a condition， a "for" clause， or a "range" clause.

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

In its simplest form， a "for" statement specifies the repeated execution of a block as long as a boolean condition evaluates to true. The condition is evaluated before each iteration. If the condition is absent， it is equivalent to `true`.

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

A "for" statement with a ForClause is also controlled by its condition， but additionally it may specify an init and a post statement， such as an assignment， an increment or decrement statement. The init statement may be a short variable declaration， but the post statement must not.

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

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

If non-empty， the init statement is executed once before evaluating the condition for the first iteration; the post statement is executed after each execution of the block (and only if the block was executed). Any element of the ForClause may be empty but the semicolons are required unless there is only a condition. If the condition is absent， it is equivalent to `true`.

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

A "for" statement with a "range" clause iterates through all entries of an array， slice， string or map， or values received on a channel. For each entry it assigns iteration values to corresponding iteration variables and then executes the block.

```
RangeClause  = Expression [ ", " Expression ] ( "=" | ":=" ) "range" Expression .
```

The expression on the right in the "range" clause is called the range expression， which may be an array， pointer to an array， slice， string， map， or channel. As with an assignment， the operands on the left must be addressable or map index expressions; they denote the iteration variables. If the range expression is a channel， only one iteration variable is permitted， otherwise there may be one or two. If the second iteration variable is the blank identifier， the range clause is equivalent to the same clause with only the first variable present.

The range expression is evaluated once before beginning the loop except if the expression is an array， in which case， depending on the expression， it might not be evaluated (see below). Function calls on the left are evaluated once per iteration. For each iteration， iteration values are produced as follows:

```
Range expression                          1st value          2nd value (if 2nd variable is present)

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                 element  e  E
```

1. For an array, pointer to array, or slice value `a`, the index iteration values are produced in increasing order, starting at element index 0. As a special case, if only the first iteration variable is present, the range loop produces iteration values from 0 up to `len(a)` and does not index into the array or slice itself. For a `nil` slice, the number of iterations is 0.
2. For a string value, the "range" clause iterates over the Unicode code points in the string starting at byte index 0. On successive iterations, the index value will be the index of the first byte of successive UTF-8-encoded code points in the string, and the second value, of type `rune`, will be the value of the corresponding code point. If the iteration encounters an invalid UTF-8 sequence, the second value will be `0xFFFD`, the Unicode replacement character, and the next iteration will advance a single byte in the string.
3. The iteration order over maps is not specified and is not guaranteed to be the same from one iteration to the next. If map entries that have not yet been reached are deleted during iteration, the corresponding iteration values will not be produced. If map entries are inserted during iteration, the behavior is implementation-dependent, but the iteration values for each entry will be produced at most once. If the map is `nil`, the number of iterations is 0.
4. For channels, the iteration values produced are the successive values sent on the channel until the channel is closed. If the channel is `nil`, the range expression blocks forever.

The iteration values are assigned to the respective iteration variables as in an assignment statement.

The iteration variables may be declared by the "range" clause using a form of short variable declaration (`:=`). In this case their types are set to the types of the respective iteration values and their scope ends at the end of the "for" statement; they are re-used in each iteration. If the iteration variables are declared outside the "for" statement, after execution their values will be those of the last iteration.

```
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
m := map[string]int{"mon":0, "tue":1, "wed":2, "thu":3, "fri":4, "sat":5, "sun":6}
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 {}  // value type of m is assignable to val
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)
}
```

go 语句

一个 “go” 语句在一个独立的控制线程中执行一个函数或是方法调用；这个线程或是叫做 goroutine ，它和原来的线程在同一地址空间中。

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

表达式必须是能够调用的。在调用的 go 例程中函数值和参数都会被正常 求值 ；不过不像常规的函数调用，这个 go 例程不会等待调用函数的结束；相反的，函数在一个新的例程中 开始执行。在函数终止的时候，这个 go 例程也就终止了；如果函数有返回值，那它们会被忽略掉。

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

select 语句

A "select" statement chooses which of a set of possible communications will proceed. It looks similar to a "switch" statement but with the cases all referring to communication operations.

```
SelectStmt  = "select" "{" { CommClause } "}" .
CommClause  = CommCase  ":" { Statement  ";" } .
CommCase    = "case" ( SendStmt  |  RecvStmt  ) | "default" .
RecvStmt    = [ Expression  [ ", "  Expression  ] ( "=" | ":=" ) ]  RecvExpr  .
RecvExpr    =  Expression  .
```

RecvExpr must be a 接收语句 . For all the cases in the "select" statement， the channel expressions are evaluated in top-to-bottom order， along with any expressions that appear on the right hand side of send statements. A channel may be `nil`， which is equivalent to that case not being present in the select statement except， if a send， its expression is still evaluated. If any of the resulting operations can proceed， one of those is chosen and the corresponding communication and statements are evaluated. Otherwise， if there is a default case， that executes; if there is no default case， the statement blocks until one of the communications can complete. If there are no cases with non-`nil` channels， the statement blocks forever. Even if the statement blocks， the channel and send expressions are evaluated only once， upon entering the select statement.

Since all the channels and send expressions are evaluated， any side effects in that evaluation will occur for all the communications in the "select" statement.

If multiple cases can proceed， a uniform pseudo-random choice is made to decide which single communication will execute.

The receive case may declare one or two new variables using a short variable declaration .

```
var c， c1， c2， c3 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")
        }
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

A "return" statement terminates execution of the containing function and optionally provides a result value or values to the caller.

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

In a function without a result type， a "return" statement must not specify any result values.

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

There are three ways to return values from a function with a result type:

1. The return value or values may be explicitly listed in the "return" statement. Each expression must be single-valued and assignable to the corresponding element of the function's result type.

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

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

2. The expression list in the "return" statement may be a single call to a multi-valued function. The effect is as if each value returned from that function were assigned to a temporary variable with the type of the respective value， followed by a "return" statement listing these variables， at which point the rules of the previous case apply.

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

3. The expression list may be empty if the function's result type specifies names for its result parameters (§ Function Types). The result parameters act as ordinary local variables and the function may assign values to them as necessary. The "return" statement returns the values of these variables.

```
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
}
```

Regardless of how they are declared， all the result values are initialized to the zero values for their type (§ The zero value) upon entry to the function.

Break statements

A "break" statement terminates execution of the innermost "for"， "switch" or "select" statement.

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

If there is a label， it must be that of an enclosing "for"， "switch" or "select" statement， and that is the one whose execution terminates (§ For statements ， § Switch statements ， § Select statements).

```
L:
        for i < n {
                switch i {
                case 5:
                        break L
                }
        }
```

Continue statements

A "continue" statement begins the next iteration of the innermost "for" loop at its post statement (§ For statements).

```
ContinueStmt = "continue" [ Label ] .
```

If there is a label， it must be that of an enclosing "for" statement， and that is the one whose execution advances (§ For statements).

goto 语句

A "goto" statement transfers control to the statement with the corresponding label.

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

```
goto Error
```

Executing the "goto" statement must not cause any variables to come into scope that were not already in scope at the point of the goto. For instance， this example:

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

is erroneous because the jump to label `L` skips the creation of `v`.

A "goto" statement outside a block cannot jump to a label inside that block. For instance， this example:

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

is erroneous because the label `L1` is inside the "for" statement's block but the `goto` is not.

fallthrough 语句

A "fallthrough" statement transfers control to the first statement of the next case clause in a expression "switch" statement (§ Expression switches). It may be used only as the final non-empty statement in a case or default clause in an expression "switch" statement.

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

defer 语句

A "defer" statement invokes a function whose execution is deferred to the moment the surrounding function returns.

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

The expression must be a function or method call. Each time the "defer" statement executes， the function value and parameters to the call are evaluated as usual and saved anew but the actual function is not invoked. Instead， deferred calls are executed in LIFO order immediately before the surrounding function returns， after the return values， if any， have been evaluated， but before they are returned to the caller. For instance， if the deferred function is a function literal and the surrounding function has named result parameters that are in scope within the literal， the deferred function may access and modify the result parameters before they are returned. If the deferred function has any return values， they are discarded when the function completes.

```
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
}
```

内置函数

内置函数都是 预声明的 。它们可以像其他函数一样被调用，不过有些函数接受一个类型而不是一个值作为第一个参数。

内置函数并没有标准的 Go 类型，所以它们只可以出现在 调用表达式 中，而不能当函数值。

```
BuiltinCall = identifier "(" [ BuiltinArgs [ "," ] ] ")" .
BuiltinArgs = Type [ "," ExpressionList ] | ExpressionList .
```

关闭

对一个管道 c 来说，内置函数 close(c) 说明不再往管道中发送数据。如果 c 只是个接收管道，那这就是一个错误。向一个关闭的管道发送数据或是 再次关闭都会引起一个 run-time panic 。关闭 nil 管道同样引起 run-time panic 。调用 close， 并且所有先前发送的数据接收完毕之后，接收操作会根据管道的类型返回一个 0 值，但不会引起阻塞。使用多值 接收操作 可以得到一个测试管道是否关闭的标志。

长度和容量

内置函数 len and cap 接收多种类型作为参数，返回一个 int 类型的值。实现保证返回的结果可以适合 int 。

```
Call      Argument type    Result

len(s)    string type      字符串的字节长度
          [n]T， *[n]T      数组长度 (== n)
          []T            分片长度
          map[K]T        map 长度（key 的数量）
          chan T         管道缓冲区中派对元素的数量

cap(s)    [n]T， *[n]T   数组长度 (== n)
          []T            分片容量
          chan T         管道缓冲区容量
```

一个分片的容量就是它底层的数组为它提供的元素个数。任何时候都必须满足一下关系：

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

nil 分片、map 或是管道的长度和容量都是 0 。

当 s 是一个字符串常量的时候， len(s) 表达式也是 常量 。只要 s 的类型是数组类型或是指向数组的指针类型， 并且 s 表达式不包括 管道接收 和 函数调用 ，那么 len(s) 和 cap(s) 都是常量， 并不用去计算 s 。而其他情况下，对 len 和 cap 的调用就不是常量，需要计算 s 而得。

分配空间

内置函数 new 接受一个类型参数 T 然后返回 *T 类型的一个值。存储空间会按初始化(§ 0 值)那里说明的对值进行初始化。

```
new(T)
```

举个例子：

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

会动态地为 S 类型的变量分配空间，将值初始化为(a=0， b=0.0)，然后返回一个 *S 的值，这个值是分配空间的地址。

构造分片、map 和管道

分片、map 和管道都是引用类型，所以不需要使用 new 来分配间址访问的空间。内置函数 make 带有一个类型 T，必须是分片、map或是管道类型， 后面跟着可选的跟类型有关的表达式。它返回的值的类型是 T (而不是 *T)。存储空间也会按照初始化(§ 0 值)那里说明的对值进行初始化。

```
调用             类型T      结果

make(T, n)        slice       长度可容量都是 n 的分片
make(T, n, m)     slice       长度是 n 容量是 m 的分片

make(T)          map        T 类型的 map
make(T, n)       map        T 类型的 map，有 n 个经过初始化的元素

make(T)          channel    T 类型的同步管道
make(T, n)       channel    带有长度为 n 的缓冲去的 T 类型的异步管道
```

参数 n 和 m 必须是整型类型。如果 n 是个负数或是比 m 还大，亦或是 n 或是 m 不能用 int 表示，那么会有一个 运行时问题 出现。

```
s := make([]int, 10, 100)       // len(s) == 10, cap(s) == 100 的分片
s := make([]int, 10)           // len(s) == cap(s) == 10 的分片
c := make(chan int, 10)        // 缓冲区长度为 10 的管道
m := make(map[string]int, 100) // 有 100 个初始化元素的 map
```

Appending to and copying slices

Two built-in functions assist in common slice operations.

The variadic function `append` appends zero or more values `x` to `s` of type `S`, which must be a slice type, and returns the resulting slice, also of type `S`. The values `x` are passed to a parameter of type `...T` where `T` is the element type of `S` and the respective parameter passing rules apply. As a special case, `append` also accepts a first argument assignable to type `[]byte` with a second argument of string type followed by `...`. This form appends the bytes of the string.

```
append(s S, x ...T) S  // T is the element type of S
```

If the capacity of `s` is not large enough to fit the additional values, `append` allocates a new, sufficiently large slice that fits both the existing slice elements and the additional values. Thus, the returned slice may refer to a different underlying array.

```
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}

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' }
```

The function `copy` copies slice elements from a source `src` to a destination `dst` and returns the number of elements copied. Source and destination may overlap. Both arguments must have identical element type `T` and must be assignable to a slice of type `[]T`. The number of elements copied is the minimum of `len(src)` and `len(dst)`. As a special case, `copy` also accepts a destination argument assignable to type `[]byte` with a source argument of a string type. This form copies the bytes from the string into the byte slice.

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

Examples:

```
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")
```

map 元素的删除

内置函数delete可以从 map m中删除键值为k的元素，而k的类型对于m的key类型来说必须是 assignable 。

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

如果元素m[k]不存在的话， delete不执行其他操作；如果对 nil 进行delete调用引起一个 run-time panic 。

Manipulating complex numbers

Three functions assemble and disassemble complex numbers. The built-in function `complex` constructs a complex value from a floating-point real and imaginary part， while `real` and `imag` extract the real and imaginary parts of a complex value.

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

The type of the arguments and return value correspond. For `complex`， the two arguments must be of the same floating-point type and the return type is the complex type with the corresponding floating-point constituents: `complex64` for `float32`， `complex128` for `float64`. The `real` and `imag` functions together form the inverse， so for a complex value `z`， `z == complex(real(z), imag(z))`.

If the operands of these functions are all 常量， the return value is a constant.

```
var a = complex(2, -2)             // complex128
var b = complex(1.0, -1.4)         // complex128
x := float32(math.Cos(math.Pi/2))  // float32
var c64 = complex(5, -x)           // complex64
var im = imag(b)                   // float64
var rl = real(c64)                 // float32
```

处理问题

Two built-in functions， `panic` and `recover`， assist in reporting and handling run-time panics and program-defined error conditions.

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

When a function `F` calls `panic`， normal execution of `F` stops immediately. Any functions whose execution was deferred by the invocation of `F` are run in the usual way， and then `F` returns to its caller. To the caller， `F` then behaves like a call to `panic`， terminating its own execution and running deferred functions. This continues until all functions in the goroutine have ceased execution， in reverse order. At that point， the program is terminated and the error condition is reported， including the value of the argument to `panic`. This termination sequence is called panicking .

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

The `recover` function allows a program to manage behavior of a panicking goroutine. Executing a `recover` call inside a deferred function (but not any function called by it) stops the panicking sequence by restoring normal execution， and retrieves the error value passed to the call of `panic`. If `recover` is called outside the deferred function it will not stop a panicking sequence. In this case， or when the goroutine is not panicking， or if the argument supplied to `panic` was `nil`， `recover` returns `nil`.

The `protect` function in the example below invokes the function argument `g` and protects callers from run-time panics raised by `g`.

```
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()
}
```

引导

当前的实现提供了几个内置的有用的引导函数。为了完整性，我们在这里也对它们进行说明，然而不会保证语言中一直存在。它们不返回结果。

函数	行为
print	输出所有的参数；参数的格式化是跟实现有关的；
println	跟 print 函数类型，不过这里在参数之间加上空白以及在结束的时候添加换行；

包

Go 程序是由链接在一起的 包 构成的。而每一个包则是由一个或是多个源文件构建起来的，源文件中包含常量、类型、变量、函数声明以及一些其他属于包的可以包内访问的东西。 这些元素可以 导出 ，然后为另外一个包所使用。

源文件组织

每一个源文件由若干部分构成，首先是一个定义了该文件所属的包子句；其次是一系列的可以为空的包的导入声明，声明希望使用的包；紧接着可以是一些函数、类型、变量或是常量声明，不过也可以么有。

```
SourceFile       = PackageClause ";" { ImportDecl ";" } { TopLevelDecl ";" } .
```

包子句

每个源文件开始于一个包子句，该子句定义了该文件属于的包。

```
PackageClause  = "package" PackageName .
PackageName    = identifier .
```

PackageName 不能是 空白标识符 。

```
package math
```

多个文件可能共享同一个 PackageName，这时候它们构成包的一个实现。一个包的实现应满足所有的源文件位于相同的目录下。

导入声明

An import declaration states that the source file containing the declaration depends on functionality of the imported package (§Program initialization and execution) and it enables access to exported identifiers of that package. The import names an identifier (PackageName) to be used for access and an ImportPath that specifies the package to be imported.

```
ImportDecl       = "import" ( ImportSpec | "(" { ImportSpec ";" } ")" ) .
ImportSpec       = [ "." | PackageName ] ImportPath .
ImportPath       = string_lit .
```

The PackageName is used in qualified identifiers to access exported identifiers of the package within the importing source file. It is declared in the file block . If the PackageName is omitted， it defaults to the identifier specified in the package clause of the imported package. If an explicit period (.) appears instead of a name， all the package's exported identifiers declared in that package's package block will be declared in the importing source file's file block and can be accessed without a qualifier.

The interpretation of the ImportPath is implementation-dependent but it is typically a substring of the full file name of the compiled package and may be relative to a repository of installed packages.

Implementation restriction: A compiler may restrict ImportPaths to non-empty strings using only characters belonging to Unicode's L， M， N， P， and S general categories (the Graphic characters without spaces) and may also exclude the characters `!"#$%&'()*，:;<=>?[\]^`{|}` and the Unicode replacement character U+FFFD.

Assume we have compiled a package containing the package clause `package math`， which exports function `Sin`， and installed the compiled package in the file identified by `"lib/math"`. This table illustrates how `Sin` may be accessed in files that import the package after the various types of import declaration.

```
Import declaration          Local name of Sin


import   "lib/math"         math.Sin
import M "lib/math"         M.Sin
import . "lib/math"         Sin
```

An import declaration declares a dependency relation between the importing and imported package. It is illegal for a package to import itself or to import a package without referring to any of its exported identifiers. To import a package solely for its side-effects (initialization)， use the blank identifier as explicit package name:

```
import _ "lib/math"
```

一个包的例子

下面是一个实现了并发的素数筛的完整的 Go 包：

```
package main


import "fmt"


// Send the sequence 2， 3， 4， … to channel 'ch'.
func generate(ch chan<- int) {
        for i := 2; ; i++ {
                ch <- i  // Send 'i' to channel 'ch'.
        }
}

// Copy the values from channel 'src' to channel 'dst',
// removing those divisible by 'prime'.
func filter(src <-chan int, dst chan<- int, prime int) {
        for i := range src {  // Loop over values received from 'src'.
                if i%prime != 0 {
                        dst <- i  // Send 'i' to channel 'dst'.
                }
        }
}

// The prime sieve: Daisy-chain filter processes together.
func sieve() {
        ch := make(chan int)  // Create a new channel.
        go generate(ch)       // Start generate() as a subprocess.
        for {
                prime := <-ch
                fmt.Print(prime, "\n")
                ch1 := make(chan int)
                go filter(ch, ch1, prime)
                ch = ch1
        }
}

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

程序初始化和执行

0 值

不管是通过声明，还是make或是new，只要为了保存一个值创造了空间但是却没有显式地初始化，那么这些空间都有默认值。 这样的值的每一个元素都会根据它的类型 被 0 值 化：对布尔类型值是false，对整数值是0，对浮点数值是0.0，对字符串是""，其他剩下的nil 的指针、函数、 接口、分片、管道和映射等都是nil 。而且这个初始化是递归进行的，所以说，如果是个结构体数组的话，那么，里面的每一个元素都会被 0 值，只要它没有被指定。

下面的两个简单声明等价：

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

看下面

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

接下来有结果：

```
t.i == 0
```

```
t.f == 0.0
t.next == nil
```

当然，如果是下面还是一样：

```
var t T
```

Program execution

A package with no imports is initialized by assigning initial values to all its package-level variables and then calling any package-level function with the name and signature of

```
func init()
```

defined in its source. A package may contain multiple `init` functions， even within a single source file; they execute in unspecified order.

Within a package， package-level variables are initialized， and constant values are determined， in data-dependent order: if the initializer of `A` depends on the value of `B`， `A` will be set after `B`. It is an error if such dependencies form a cycle. Dependency analysis is done lexically: `A` depends on `B` if the value of `A` contains a mention of `B`， contains a value whose initializer mentions `B`， or mentions a function that mentions `B`， recursively. If two items are not interdependent， they will be initialized in the order they appear in the source. Since the dependency analysis is done per package， it can produce unspecified results if `A`'s initializer calls a function defined in another package that refers to `B`.

An `init` function cannot be referred to from anywhere in a program. In particular， `init` cannot be called explicitly， nor can a pointer to `init` be assigned to a function variable.

If a package has imports， the imported packages are initialized before initializing the package itself. If multiple packages import a package `P`， `P` will be initialized only once.

The importing of packages， by construction， guarantees that there can be no cyclic dependencies in initialization.

A complete program is created by linking a single， unimported package called the main package with all the packages it imports， transitively. The main package must have package name `main` and declare a function `main` that takes no arguments and returns no value.

```
func main() { … }
```

Program execution begins by initializing the main package and then invoking the function `main`. When the function `main` returns， the program exits. It does not wait for other (non-`main`) goroutines to complete.

Package initialization—variable initialization and the invocation of `init` functions—happens in a single goroutine， sequentially， one package at a time. An `init` function may launch other goroutines， which can run concurrently with the initialization code. However， initialization always sequences the `init` functions: it will not start the next `init` until the previous one has returned.

错误

预声明的类型error定义如下：

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

对于表示一个错误条件来说，这是个灰常方便的接口，如果没有错误的话就是 nil。比如说，一个从文件中读数据的函数可能是这样定义的：

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

运行时问题

在程序执行的过程中，如果出现访问数组越界这些错误就会触发一个 运行时问题 。不过也可以通过调用内置函数 `panic` 来实现， 这个函数带有一个实现定义的接口类型 `runtime.Error` 的值；这个值只要满足预声明的接口类型 `error` 就好。 而实际的表示不同运行问题信息的值则不做具体指定。

```
package runtime

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

系统考量

包 `unsafe`

The built-in package `unsafe`， known to the compiler， provides facilities for low-level programming including operations that violate the type system. A package using `unsafe` must be vetted manually for type safety. The package provides the following interface:

```
package unsafe

type ArbitraryType int  // shorthand for an arbitrary Go type; it is not a real type
type Pointer *ArbitraryType

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

Any pointer or value of underlying type `uintptr` can be converted into a `Pointer` and vice versa.

The function `Sizeof` takes an expression denoting a variable of any type and returns the size of the variable in bytes.

The function `Offsetof` takes a selector (§ Selectors) denoting a struct field of any type and returns the field offset in bytes relative to the struct's address. For a struct `s` with field `f`:

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

Computer architectures may require memory addresses to be aligned ; that is， for addresses of a variable to be a multiple of a factor， the variable's type's alignment . The function `Alignof` takes an expression denoting a variable of any type and returns the alignment of the (type of the) variable in bytes. For a variable `x`:

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

Calls to `Alignof`， `Offsetof`， and `Sizeof` are compile-time constant expressions of type `uintptr`.

大小以及对齐保证

对于(§ 数值类型)来说，下面的大小必须保证：

```
type                                 size in bytes


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

下面的最小对齐属性也必须保证：

1. 对于任意类型的变量 x ： `unsafe.Alignof(x)` 至少是 1。
2. 对于结构类型的变量 x ： `unsafe.Alignof(x)` 是 x 中的所有字段 f 的 `unsafe.Alignof(x.f)` 的最大值，至少是 1 。
3. 对于数组类型 x ： `unsafe.Alignof(x)` 和 `unsafe.Alignof(x[0])` 是一样的， 至少是 1 。

一个结构体或是数组类型，如果不含有尺寸大于0 的任何字段(或是说元素)，那么这种类型的大小是 0 。两个不同的 0 尺寸的变量可能在内存中会有相同的地址。