Golang

sanfusu <sanfusu@foxmail.com> 译校

2018 年 8 月 6 日

第一章　介绍

这是一份关于 Go 语言的参考指南。更多信息和文档，请查看 golang.org

Go 是以系统语言为宗旨而设计的通用目的语言。Go 具有强类型和垃圾回收并且支持原生并发编程。程序由 *packages* 组建。包的属性可以有效的进行依赖管理。

语法方面，紧凑且常规，因而可以很方便的使用类似集成开发环境的自动化工具进行分析。

第二章　记号

语法格式通过扩展的巴克斯范式（EBNF）表示。

```
1 Production  = production_name "=" [ Expression ] "." .
2 Expression  = Alternative { "|" ALternative } .
3 Alternative = Term { Term } .
4 Term        = production_name | token [ "…" token ] | Group | Option | Repetition
      .
5 Group       = "(" Expression ")" .
6 Option      = "[" Expression "]" .
7 Repetition  = "{" Expression "}" .
```

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

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

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

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

第三章　源码表示

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

区分每一个编码点，比如，大写字母和小写字母是不同的字符。

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

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

† 3.1　字符

下列术语被用来标识特定的 Unicode 字符类：

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

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

† 3.2　字母和数字

下划线字符 _(U+005F) 被视为一个字母。

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

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

第四章　词法元素

† 4.1　注释

注释充当程序文档。有两种形式：

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

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

† 4.2　记号

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

† 4.3　分号

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

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

为了反映惯用的用法，本文档中的代码会使用以上规则来省略掉分号。

† 4.4　标识符

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

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

```
a
_x9
ThisVariableIsExported
αβ
```

部分标识符是预先声明的。

† 4.5　关键词

以下为保留的关键词，不能用作标识符。

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

† 4.6　运算符和标点符号

以下字符序列表示运算符（包括赋值操作符）以及标点符号：

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

† 4.7　整型字面量

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

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

```
42
0600
0xBadFace
170141183460469231731687303715884105727
```

† 4.8　浮点字面量

浮点字面量是浮点常量的十进制表示。具有一个整数部分，一个十进制小数点，一个小数部分，和一个指数部分。整数和小数部分由十进制数组成；指数部分，e 或者 E 后面紧跟着一个可选的有符号十进制指数。可以省略整数部分或者小数部分中的一个；类似的，小数点或者指数中的一个可以被省略。

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

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

† 4.9　虚数字面量

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

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

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

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

† 4.10　Rune literals

一个符文字面量表示一个符文常量，一个用来标识 Unicode 编码点的整数值。一个符文字面量使用单引号包含的一个或多个字符来表达，如 `'x'` 或者 `'\n'`。除换行和未转义的单引号外，任何字符都可以出现在单引号内。使用单引号包含的字符代表了字符本身的 Unicode 值，但是以反斜杠开头的多字符序列会以各种格式对值进行编码。

符文字面量的最简单形式是在引号内表示单个字符；由于 Go 源文本是 UTF-8 编码的 Unicode 字符，所以多个 UTF-8 编码的字节可以使用单个整数值表示。比如字面值 `'a'` 拥有表示字面量 `a` 的单个字节，Unicode 编码点为 U+0061，值为 `0x61`，但是 `'ä'` 则拥有两个（0xc3, 0xa4）用来表示 a 的分音符号的字节，U+00E4，值为 `0xe4`。

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

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

反斜杠后面跟上特定的单字符表示特殊的值：

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

在符文字面量里的其余以反斜杠开始的序列均是非法的。

```
rune_lit         = "'" ( unicode_value | byte_value ) "'" .
unicode_value    = unicode_char | little_u_value | big_u_value | escaped_char .
byte_value       = octal_byte_value | hex_byte_value .
octal_byte_value = `\` octal_digit octal_digit octal_digit .
hex_byte_value   = `\` "x" hex_digit hex_digit .
little_u_value   = `\` "u" hex_digit hex_digit hex_digit hex_digit .
big_u_value      = `\` "U" hex_digit hex_digit hex_digit hex_digit
                           hex_digit hex_digit hex_digit hex_digit .
escaped_char     = `\` ( "a" | "b" | "f" | "n" | "r" | "t" | "v" | `\` | "'" | `"`
    ) .
```

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

† 4.11　字符串字面量

字符串字面量表示从相邻的字符序列中所获取的字符串常量。字符串字面量有两种形式：原始字符串字面量和解释型字符串字面量。

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

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

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

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

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

下面示例均表示相同的字符串：

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

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

第五章　常量

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

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

一般来说，复数常量是常量表达式的一种形式，将在相应的章节中讨论。

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

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

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

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

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

这些要求同时适用于字面常量和常量表达式的计算结果。

第六章　变量

一个变量是用来保存值的存储位置。变量所允许的值由其类型来决定。

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

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

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

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

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

第七章　类型

一个类型决定了一组值和特定于这些值的操作以及方法。一个类型可以通过类型名来表示，如果有的话也可以通过*类型字面量*来指定。类型字面量从现有类型中组成一个类型。

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

本语言预先声明了一些特定的类型名。其余由类型声明来引入。*复合类型*—数组，结构体，指针，函数，接口，切片，图，通道类型 — 可以通过类型字面量构造。

每一个类型 `T` 都有一个底层类型：如果 `T` 时预声明的布尔，数值，字符串类型或类型字面量中的一个，则相应的底层类型时其自身。否则，`T` 的底层类型是 `T` 在类型声明中引用到的类型的底层类型。

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

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

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

† 7.1　方法集

一个类型可能会有与其相关联的方法集。一个接口类型的方法集是其接口。其他任何类型 `T` 的方法集由所有使用类型 `T` 作为接收器声明的方法组成。指针类型 `*T` 相应的方法集为所有使用接收器 `*T` 或 `T` 声明的方法的集合（也就是说，包含 `T` 的方法集）。此外，如结构体类型章节中所说，这些规则同样适用于包含嵌入字段的结构体。任何其他类型则具有空方法集。在一个方法集中，每一个方法必须有一个独一无二的非空方法名。

一个类型的方法集决定了该类型实现的接口和可通过该类型接收器所调用的方法。

† 7.2　布尔类型

一个布尔类型代表通过预声明的常量 `true` 和 `false` 表示的布尔真值集合。预声明的布尔类型为 `bool`；这是一个定义的类型。

† 7.3　数值类型

一个数值类型表示整数或浮点值的集合。预声明的依赖于体系结构的数值类型为：

uint8	the set of all unsigned 8-bit integers (0 to 255)
uint16	the set of all unsigned 16-bit integers (0 to 65535)
uint32	the set of all unsigned 32-bit integers (0 to 4294967295)
uint64	the set of all unsigned 64-bit integers (0 to 18446744073709551615)
int8	the set of all signed 8-bit integers (-128 to 127)
int16	the set of all signed 16-bit integers (-32768 to 32767)
int32	the set of all signed 32-bit integers (-2147483648 to 2147483647)
int64	the set of all signed 64-bit integers (-9223372036854775808 to 9223372036854775807)
float32	the set of all IEEE-754 32-bit floating-point numbers
float64	the set of all IEEE-754 64-bit floating-point numbers
complex64	the set of all complex numbers with float32 real and imaginary parts
complex128	the set of all complex numbers with float64 real and imaginary parts
byte	alias for uint8
rune	alias for int32

n-bit 整数的值具有 n 个 bit 位，并使用二的补码算术表示。

一些预声明的数值类型具有特定于实现的大小：

uint	32 bit 或 64 bit
int	和 uint 具有相同大小
uintptr	足以存储一个未释义的指针值的所有 bit 的无符号整型。

为了避免移植性问题，所有的数值类型都是定义的类型，也就是各不相同，但除了 `byte` 是 `rune` 的别名，以及 `rune` 是 `int32` 的别名。如果一个表达式或赋值中混合了不同的数值类型，则需要进行转换。比如 `int32` 和 `int` 尽管在特定体系结构中具有相同大小，但并不是相同类型。

† 7.4　字符串类型

一个字符串类型代表字符串值的集合。一个字符串是一个字节序列（可能为空序列）。字符串是不可修改的：一旦创建，便无法改变字符串的内容。预声明的字符串类型为 `string`，为定义的类型。

字符串 s 的长度可以通过内置函数 len 来求得。如果字符串是常量，则长度会在编译期间计算得出。字符串中的字节可通过 0 到 len(s)-1 的整型索引访问。对字符串中的字节元素取地址是非法的；如果 s[i] 是字符串的第 i 个字节，则 &[si] 是无效的。

† 7.5　数组类型

一个数组一个编号的单类型元素（被称为元素类型）序列。元素的数量被称为长度，并且永远不为负。

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

长度作为数组类型的一部分，值必须为类型 int 的非负常量。数组 a 的长度可以通过内置函数 len 计算得出。数组元素可以通过 0 到 len(a)-1 的整型下标寻址。数组类型永远是一维的，但可以组合形成多维类型。

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

† 7.6　切片类型

切片是用来提供编号的序列元素访问的底层元素中连续片段的描述符。一个切片类型表示其元素类型数组的所有切片的集合。未初始化切片的值是 nil。

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

和数组类似，切片是可索引的并且具有长度。切片 s 的长度可以通过内置函数 len 来计算得出；和数组不同的是，可以在执行期间改变。元素可以通过 0 到 len(s)-1 的整型下标寻址。一个给定元素的切片索引可能会小于底层数组中相同元素的索引。

一个切片，一旦初始化后，便永远的和保存其元素的底层数组相关联。一个切片因此会和其数组以及相同数组的切片共享存储；相反，不同的数组永远具有不同的存储。

底层数组可能会超过切片的结尾。切片的容量为该区间的度量：切片的长度加上数组超过切片的长度。长度为其容量的切片可以通过从原有切片中重新划分。切片 a 的容量可以通过内置函数 cap(a) 计算得出。

通过内置函数 make 创建一个给定的初始化过的切片值，该函数需切片类型和指定长度的参数以及可选的容量参数。通过 make 创建的切片永远会申请一个新的隐藏的数组，返回的切片将会指向这个数组。也就是说，执行

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

会和申请一个数组并分片提供一样的切片，因此下面两个表达式是等价的：

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

和数组一样，切片永远是一维的，但是可以组合成更高维的对象。使用数组的数组是，内部数组永远具有相同的长度；但是在使用切片的切片时（或者切片的数组），内部长度可以动态的变化。此外，内部切片必须独立的初始化。

† 7.7　结构体类型

一个结构体是一个被称为字段的命名元素序列，每一个元素都有一个名字和类型。字段名既可以显式的指明（标识符列表），也可以隐式的指明（嵌入字段）。在结构体中，非空白字段名必须独一无二的。

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

```
// An empty struct.
struct {}

// A struct with 6 fields.
struct {
    x, y  int
    u  float32
    _  float32  // padding
    A *[] int
    F func()
}
```

一个字段使用一个类型进行声明但没有显式的字段名则称为嵌入的字段。一个嵌入的字段必须指定为类型名 `T` 或者为一个指向非接口类型的指针 `*T`，并且 `T` 本身不能是一个指针类型。无限制的类型名可以作为字段名。

```
// A struct with four embedded fields of types T1, *T2, P.T3 and *P.T4
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
}
```

下面声明是非法的，应为字段名在结构体类型中必须是独一无二的。

```
struct {
    T     // conflicts with embedded field *T and *P.T
    *T    // conflicts with embedded field T and *P.T
    *P.T  // conflicts with embedded field T and *T
}
```

如果 x.f 可以合法的表示一个字段或者方法 f，而字段或方法 f 是结构体 x 中的嵌入字段，则称之为被提升。

除了不能再结构体复合字面量中作为字段名外，提升的字段和普通字段一样。

给定一个结构体类型 S 和定义的类型 T，以下情况中，提升的方法会被包含在结构体的方法集中：

- 如果 S 包含嵌入的字段 T，则 S 和 *S d 的方法集都包含接收器为 T 的提升方法。*S 的方法集中也会包含带有接收器 *T 的提升方法。
- 如果 S 包含嵌入字段 *T，则 S 和 *S 同时包含带有接收器 T 或 *T 的提升方法。

一个字段声明后面可以跟上可选的字符串字面量标签，该标签将成为相应字段声明中所有字段的属性 (attribute)。一个空标签字符串等价于一个缺省标签。一个标签可以通过反射接口可见，并参与结构体的类型识别，其他情况下会被标签忽略掉。

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

// A struct corresponding to a TimeStamp protocol buffer.
// The tag strings define the protocol buffer field numbers;
// they follow the convention outlined by the reflect package.
struct {
    microsec   uint64 `protobuf:"1"`
    serverIP6  uint64 `protobuf:"2"`
}
```

† 7.8　指针类型

一个指针类型表示指向给定类型变量的所有指针集合，该给定类型被称为指针的基类型。未初始化指针的值为 nil

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

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

† 7.9　函数类型

一个函数类型表示所有带有相同类型的参数和结果的函数。未初始化的函数类型变量的值为 `nil`。

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

在参数列表或结果中，参数名和结果民（标识符列表）必须全部呈现或者全部省略。如果呈现，每一个名字代表一个特定类型的参数或结果，签字中所有非空白名必须是独一无二的。如果缺省，每一个类型代表给类型的一个参数或结果。参数和结果列表必须使用圆括弧包含，除非只有一个未命名结果。

函数签字里面最后的输入参数类型可以有一个 … 前缀。具有这种参数的函数被称为可变参数函数[1]，这种参数可以视为 0 个或多个参数。

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

† 7.10　接口类型

一个接口类型表示一个被称为接口的方法集。一个接口类型变量可以存储任何方法集为接口超集的类型的变量。这种类型被称为*接口的实现*。未初始化的接口类型变量的值为 `nil`。

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

一个接口类型的方法集中，每一个方法必须具有独一无二的非空白名。

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

[1] variadic

```
    Close()
}
```

可以有多个类型实现同一个接口。比如，两个类型 S1 和 S2 可以具有方法集

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

（其中 T 既可以代表 S1 也可以代表 S2），同时被 S1 和 S2 实现，S1 和 S2 也可以有其余（共享）的方法。

一个类型可以实现任何由其方法子集组成的接口，因此一个类型可以实现几个不同的接口。比如，所有的类型都会实现一个空接口。

```
interface{}
```

于此类似，下面出现在一个类型声明中的接口说明，定义了一个名为 Locker 的接口：

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

如果 S1 和 S2 也实现了

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

则他们在实现 File 接口的同时实现了 Locker 接口。

一个接口 T 可以使用名为 E 的接口类型来代替方法说明。这称为将接口 E 嵌入到 T 中，这种方式会将 E 的所有方法（包括导出和未导出的）加入到接口 T 中。

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

type File interface {
    ReadWriter  // same as adding the methods of ReadWriter
    Locker      // same as adding the methods of Locker
    Close()
}

type LockedFile interface {
    Locker
    File        // illegal: Lock, Unlock not unique
    Lock()      // illegal: Lock not unique
}
```

一个接口类型 T 无法递归的嵌入自身或者任何嵌入 T 的接口。

```
// illegal: Bad cannot embed itself
type Bad interface {
    Bad
}

// illegal: Bad1 cannot embed itself using Bad2
type Bad1 interface {
    Bad2
}
type Bad2 interface {
    Bad1
}
```

† 7.11　图类型

一个图是一组无序的单类型元素；该类型被称为元素类型。可以通过其他类型的唯一键来索引，即键类型。未初始化的图的值为 nil。

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

必须为 key 类型操作数完全定义比较操作符 == 和 !=；因此 key 类型不能为函数、图、或者切片。如果 key 类型是一个接口类型，必须为这些动态 key 值定义比较操作符；失败将会导致 run-time panic。

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

map 元素的数量被称为长度。对于 map m，可以通过内置函数 len 来获取长度，并可能会在执行时更改。元素可以在执行时使用赋值来添加，并且通过索引表达式来获取；他们可以通过内置函数 delete 来删除。

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

初始容量并不会限制 map 的大小：map 会根据其存储的元素数量来增加大小。一个 nil map 等价于一个空 map，但无法向其中添加元素。

† 7.12　通道类型

一个通道通过使用发送和接受特定元素类型的值为并发执行函数提供通讯机制。未初始化的通道的值为 nil

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

可选的 <- 运算符表好似通道方向，发送还是接收。如果没有给定方向，则通道是双向的。一个通道可以通过转换或赋值限制为只用作发送或者只用作接收。

```
chan T          // can be used to send and receive values of type T
chan<- float64  // can only be used to send float64s
<-chan int      // can only be used to receive ints
```

<- 会和最左边的 chan 关联：

```
chan<- chan int    // same as chan<- (chan int)
chan<- <-chan int  // same as chan<- (<-chan int)
<-chan <-chan int  // same as <-chan (<-chan int)
chan (<-chan int)
```

一个新的，且初始化过的通道值可以通过内置函数 make 来创建，make 函数需要通道类型和可选的容量作为参数：

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

容量即元素数量，用来设置通道中的缓冲区大小。如果容量是零或者缺省，则通道无缓冲并且仅当接收器和发送器都准备好时，才会成功。若有缓冲区，则会在发送时未满，接收时未空，才会无阻塞的通讯成功。一个 nil 通道永远不会处于准备通讯状态。

通道可以使用内置函数 close 关闭。接收运算符的多值赋值形式可以用来报告，接受的值是否在通道关闭之前发送。

单通道可能会被用在发送语句,接收操作,和被任意数量的无跟多的同步操作的 goroutinges 调用 cap 和 len 内置函数。通道的表现形式为先进先出的队列。比如一个 goroutine 向一个通道发送值，而第二个 goroutine 用于接收值，那么这些值会按照发送的顺序接收。

第八章　类型和值的属性

† 8.1　类型特征

两个类型要么相等要么不等。

一个定义的类型永远和其他任何类型不同。非定义类型在其底层类型字面量结构相等时相等；也就是说具有相同的字面结构以及相应的组件具有相同的类型。具体为：

- 如果两个数组类型具有相同的元素类型和数组长度，则该两个数组相等。
- 如果两个切片类型具有相同的元素类型，则相等。
- 如果两个结构体具有相同的字段序列，并且相应的字段具有相同的名字和相等的类型以及相等的标签。不同包中的未导出的字段名永远不相同。
- 两个指针如果具有相等的基类型，则相等。
- 如果两个函数类型具有相同数量的参数和结果值，并且相应的参数和结果类型相等（不用管函数是不是可变参数函数），则两个函数类型相等。参数和结果名不需要匹配。
- 两个接口类型在其具有相同名字的方法集和等价的函数类型时相等。不同包中的非导出方法名永远不同。方法顺序不相干。
- 两个 map 类型在其具有相等的 key 和元素类型时相等。
- 两个通道类型在他们具有相等的元素类型和相同的方向时相等。

以下声明中

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

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

```
    B5 func(x  int, y  float64) *A1
)

type    C0 = B0
```

等价的类型为：

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

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

B0 和 B1 是通过不同的类型定义创建的新类型，因此这两个类型并不相同；func (int, float64) *B0 和 func(x int, y float64) *[] string 不同是因为 B0 不同于 [] string。

† 8.2　可赋值性

如果满足以下条件，则值 x 可以赋值给类型 T 的变量（“x 是可赋值给 T 的”）：

- x 的类型和 T 完全相同。
- x 的类型 V 和 T 具有相同的底层类型，并且 V 和 T 中至少有一个不是定义的类型。
- T 是一个借口类型，并且 x 实现了 T。
- x 是一个双向通道值，T 是一个通道类型，x 的类型 V 和 T 具有完全相同的元素类型，并且这两个类型中至少有一个不是定义的类型。
- x 是预声明标识符 nil 并且 T 是一个指针，函数，切片，图，通道，或者借口类型。
- x 是一个可被类型 T 表示的无类型常量。

† 8.3　可表达性

当满足下列条件之一时，常量 x 可被表示为类型 T 的值：

- x 在由 T 决定的值得集合中。
- T 是一个浮点类型，并且 x 可以无溢出的四舍五入到 T 的精度范围内。使用 IEEE 754 的偶数约算规则进行四舍五入，但是 IEEE 的非负零简化成一个无符号零。注意常量值永远不可能是一个 IEEE 非负零，NAN，或者无穷。
- T 是一个复数类型，x 的组件 real(x) 和 imag(x) 可以由 T 的组件类型（float32 或 float64 的值表示。

x	T	x 能够被 T 的值表示的原因是
'a'	byte	97 在字节类型的值得集合中
97	rune	rune 是 int32 的别名，并且 97 在 32-bit 整型值得集合中
"foo"	string	"foo" 在字符串值得集合中
1024	int16	1024 在 16-bit 整型值的集合中
42.0	byte	42 在无符号 8-bit 整型值得集合中
1e10	uint64	10000000000 在 64-bit 无符号整型值的集合中
2.718281828459045	float32	2.718281828459045 四舍五入到 2.7182817 后再 float32 值得集合中。
-1e-1000	float64	-1e-1000 四舍五入到 IEEE -0.0 进一步简化到 0.0
0i	int	0 是一个整型值
(42 + 0i)	float32	42.0 (虚部为 0) 在 float32 值得集合中

x	T	x 无法被 T 表示的原因
0	bool	0 不在布尔类型值的集合中
'a'	string	'a' 是一个 rune，不在字符串值的集合中
1024	byte	1024 不在无符号 8-bit 整型值的集合中
-1	uint16	-1 不在无符号 16-bit 整型值的集合中
1.1	int	1.1 不是一个整型值
42i	float32	0 + 42i 不在 float32 值的集合中
1e1000	float64	1e1000 在约数后溢出为 IEEE 的正无穷（+Inf）

第九章　块 Blocks

一个块域可以是匹配的花括弧内连续的空声明和语句序列。

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

除了源代码中显式的块域，还有以下隐式的块域：

1. 包含所有 Go 源文本的全局块。
2. 每一个包中都有一个包含该包中所有代码文本的 package block
3. 每一个文件都有一个文件块包含该文件内的所有 Go 源文本。
4. 每一个 “if”，“for”，“switch” 语句都有一个自己的隐式块。
5. “switch” 或 “select” 语句中的每一个子句都表现为一个隐式块。

块可以嵌套并会影响作用域。

第十章　声明和作用域

一个声明将一个非空白标识符绑定到一个常量、类型、变量、函数、标签或者包。程序中的每一个标识符必须被声明。相同块中不可以将重复声明一个标识符，并且标识符不可同时声明在文件块和包块中。

空白标识符可以和声明中的任何其他标识符一样使用，但是不会引入一个绑定，因而不是一个声明。在包块中，标识符 `init` 可能会用作 init 函数声明，和空白标识符一样也不会引入一个新的绑定。

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

被声明的标识符的作用域是标识符表示特定常量，类型，变量，函数，标签，或包的源文本区间。

Go 使用块进行此法分区：

1. 预声明标识符作用域为全局块。
2. 在最顶层（所有函数之外）声明的用来表示一个常量，类型，变量，或者函数（非方法）的标识符的作用域为包块。
3. 一个导入的包名的作用域为包含导入声明的文件的文件域。
4. 一个表示一个方法接收器，函数参数或结果变量的标识符的作用域为函数体。
5. 函数内声明的常量或变量标识符的作用域起始于 ConstSpec 或 Var Spec（ShortVarDecl 表示短变量声明）结尾以及最内层包含快结尾。
6. 函数内声明的类型标识符作用域起始于 TypeSpec 中的标识符并结束于最内层包含块的结尾。

一个区块中声明的标识符可以其内部区块重新声明。但内部区块中声明的标识符在作用域范围内表示内部声明所声明的条目。

包子句部分并不是一个声明；包名不会出现在任何作用域内。其目的只是为了将一些文件归纳于同一个包中，并且为导入声明指定默认的包名。

† 10.1　标签作用域

标签通过标签化语句声明，并用在 break，continue 和 goto 语句中。定义一个从未用过的标签是非法的。和其他标识符相反，标签无区块作用域并且不会和非标签的标识符冲突。标签的作用域为声明所在的函数体内，并且不包含任何嵌套的函数体。

† 10.2　空白标识符

空白标识符通过下划线字符 _ 表示。空白标识符用作匿名占位符并在声明、操作数和赋值中具有特殊意义，

† 10.3　预声明标识符

下面标识符隐式的声明在全局区块中：

```
Types:  bool byte complex64 complex128 error float32 float64 int int8
        int16 int32 int64 rune string uint uint8 uint16 uint32 uint64
        uintptr
Constants:
        true false iota
Zero value:
        nil
Functions:
        append cap close complex copy delete imag len make new panic print
        println real recover
```

† 10.4　导出的标识符

一个标识符可以被导出，从而允许在其他包里访问。满足以下条件的标识符会被导出：

1. 标识符名的首字符为大写的 Unicode 字母（Unicode 类别为 “Lu”）；并且
2. 标识符声明在包区块中或者是一个字段名或是一个方法名。

所有其他标识符不会导出。

† 10.5　标识符的唯一性

在给定的标识符集合中，如果一个标识符不同于集合中任何其他标识符，则称之为唯一。两个标识符的不同点在于拼写，或出现在不同的包中并且没有被导出。其他情况下，认为标识符相同。

† 10.6　常量声明

一个常量声明将一组标识符（常量的名字）绑定到一组常量表达式的值。标识符的数量必须和表达式的数量相等，并且左边第 n 个标识符被绑定到右边第 n 个表达式的值。

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

```
3
4  IdentifierList = identifier { "," identifier } .
5  ExpressionList = Expression { "," Expression } .
```

如果出现类型，则所有常量会采用指定的类型，并且表达式必须能够赋值给该类型。如果省略类型，常量则会采用相应表达式的类型。如果表达式的值为无类型常量，声明的常量依旧保留无类型，并且该常量标识符表示该常量值。比如，若表达式是一个浮点字面量，常量标识符则会表示一个浮点常量，即便字面量的小数部分是 0。

```
1  const Pi  float64 = 3.14159265358979323846
2  const zero = 0.0         // untyped floating-point constant
3  const (
4      size  int64 = 1024
5      eof         = -1     // untyped integer constant
6  )
7  /* a = 3, b = 4, c = "foo", untyped integer and string constants */
8  const a, b, c = 3, 4, "foo"
9  const u, v  float32 = 0, 3    // u = 0.0, v = 3.0
```

在使用原括弧包含的 `const` 声明列表中，除了第一个 ConstSpec 其余的表达式列表均可以省略。这种空列表等价于前面第一个非空表达式列表和其类型（如果有的话）的文本替换。故，省略表达式列表等于重复前一个列表。标识符的数量必须等于前一个列表中表达式的数量。连同 iota 常提供量产生器一起可以提供轻量级的序列值声明：

```
1  const (
2      Sunday =  iota
3      Monday
4      Tuesday
5      Wednesday
6      Thursday
7      Friday
8      Partyday
9      numberOfDays  // this constant is not exported
10 )
```

† 10.7　Iota

在常量声明中，预声明标识符 `iota` 表示连续无类型常量。`iota` 的值为常量声明中相应的 ConstSpec 的索引（从零开始）。可以用来构造一组相关的常量：

```
1  const (
2      c0 =  iota  // c0 == 0
3      c1 =  iota  // c1 == 1
4      c2 =  iota  // c2 == 2
5  )
```

```
6
7  const (
8      a = 1 <<  iota  // a == 1  (iota == 0)
9      b = 1 <<  iota  // b == 2  (iota == 1)
10     c = 3           // c == 3  (iota == 2, unused)
11     d = 1 <<  iota  // d == 8  (iota == 3)
12 )
13
14 const (
15     u         =  iota * 42  // u == 0     (untyped integer constant)
16     v  float64 =  iota * 42  // v == 42.0  (float64 constant)
17     w         =  iota * 42  // w == 84    (untyped integer constant)
18 )
19
20 const x =  iota  // x == 0
21 const y =  iota  // y == 0
```

根据定义，在同一个 ConstSpec 中的多个 `iota` 具有相同的值：

```
1  const (
2      bit0, mask0 = 1 <<  iota, 1<< iota - 1  // bit0 == 1, mask0 == 0  (iota == 0)
3      bit1, mask1                             // bit1 == 2, mask1 == 1  (iota == 1)
4      _, _                                    //                        (iota == 2,
       unused)
5      bit3, mask3                             // bit3 == 8, mask3 == 7  (iota == 3)
6  )
```

最后一个示例中利用了隐式的重复最后一个非空表达式列表。

† 10.8　类型声明

一个类型声明将一个标识符即类型名绑定到一个类型。类型声明具有两种形式：别名声明和类型定义。

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

• 10.8.1　别名声明

一个别名声明将一个标识符绑定到一个给定的类型上。

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

在该标识符的作用域内，提供类型别名的服务。

```
1  type (
2      nodeList = []*Node  // nodeList and []*Node are identical types
```

```
    Polar    = polar    // Polar and polar denote identical types
)
```

• 10.8.2　类型定义

类型定义会创建一个和给定类型拥有相同底层类型和操作，但截然不同的类型，并且将一个标识符绑定到新创建的类型。

```
TypeDef = identifier Type .
```

新类型被称为定义的类型。该类型不同于其他任何类型，包括它的创建来源。

```
type (
    Point struct{ x, y
    float64 }  // Point and struct{ x, y float64 } are different types
    polar Point                    // polar and Point denote different types
)

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

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

定义的类型可能会有与其相关联的方法。不会集成任何与给定类型绑定方法，但是接口类型或聚合类型的元素的方法集不会更改：

```
// 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 PtrMutex's underlying type *Mutex 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 embedded field Mutex.
type PrintableMutex struct {
```

```
    Mutex
}

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

类型定义可能会被用来定义不同的布尔，数值或字符串类型并且与其关联一些方法：

```
type TimeZone  int

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

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

† 10.9　变量声明

一个变量声明可以创建一个或多个变量，绑定到相应的标识符，并且给出每一个变量的类型和一个初始值。

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

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

如果给出表达式列表，变量则会通过赋值规则使用表达式来初始化变量。否则，每一个变量会被初始化为 0 值。

变量会优先使用给出的类型，否则使用初始化时所赋的值的类型。如果值为无类型常量，则会先转换为其默认类型；如果时一个无类型布尔值，则会先转换为 `bool` 类型。预声明的值 `nil` 必须使用显式的类型来初始化一个变量。

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

受限制于实现：编译器可能会认为在函数体内声明一个从未被使用的变量是非法声明。

† 10.10　短变量声明

短变量声明使用以下格式：

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

这是使用无类型初始化表达式声明变量的简写：

```
"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
```

和常规变量声明不同，短变量声明可以使用相同的类型重新声明在之前同一块中声明过的变量（或者块为函数体的参数列表），并且在重声明时至少有一个非空白变量是新的。相应的，重声明只能出现在多变量的短声明中。重声明不会引入新的变量，他只会将新值赋值给原有的变量中。

```
field1, offset := nextField(str, 0)
field2, offset := nextField(str, offset)  // redeclares offset
a, a := 1, 2    // illegal: double declaration of a
                // or no new variable if a was declared elsewhere
```

短变量声明只能出现在函数内，在一些环境中比如 `"if"`, `"for"`, `"switch"` 语句的初始器中，可以用来声明局部临时变量。

† 10.11　函数声明

一个函数声明会将一个标识符（即函数名）绑定到函数上。

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

如果函数签字声明了结果参数，函数体的语句列表必须使用一个终止语句结尾。

```
func IndexRune(s  string, r  rune)  int {
    for i, c := range s {
        if c == r {
            return i
        }
    }
    // invalid: missing return statement
}
```

一个函数声明可能会忽略函数体。这中声明为实现于 Go 以外的函数提供签字，比如汇编函数。

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

func flushICache(begin, end  uintptr)  // implemented externally
```

† 10.12　方法声明

一个方法是一个带有接收器的函数。一个方法声明会将一个标识符，即方法名，绑定到方法中，并使用接收器的基类型与方法进行关联。

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

通过在方法名前使用额外的参数部分来指定接收器。该参数部分必须声明为一个非可变参数，该参数为接收器。参数类型必须是 `T` 或 `*T` 这两个形式中的一种，`T` 为类型。由 `T` 表示的类型为接收器基类型；他不能是一个指针或这接口类型，并且必须定义在和方法所属的同一个包内。方法被称为绑定到一个基类型并且方法名只在类型 `T` 或 `*T` 的选择器中可见。

在方法签字中，非空白接收器标识符必须是独一无二的。如果方法体内没有引用接收器的值，则可以省略接收器标识符。同样的规则适用于函数和方法的参数。

对于基类型，绑定到该基类型中的非空白方法名必须是独一无二的。如果基类型是一个结构体类型，非空白方法和字段名必须不同。

给定类型 `Point`，以下声明：

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

func (p *Point) Scale(factor  float64) {
```

```
6     p.x *= factor
7     p.y *= factor
8 }
```

会使用接收器类型 *Point 将方法 Length 和 Scale 绑定到基类型 Point。

方法的类型为使用接收器作为第一个参数的函数的类型。比如，方法 Scale 具有类型：

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

但是，用这种方式声明的函数并不是一个方法。

第十一章　表达式

一个表达式通过对操作数施加运算符和函数来表示一个值的计算。

† 11.1　操作数

操作数表示一个表达式中的元素值。一个操作数可能是一个字面量，一个非空标识符（可以是被限定的）表示一个常量、变量、或这函数、或者一个参数化的表达式。

空白标识符只能在赋值表达式的左侧作为操作数出现。

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

† 11.2　限定的标识符

一个限定的标识符是一个使用包名作为前缀限定的标识符。包名和标识符均不能为空白标识符。

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

一个限定的标识符可以在不同的包中访问一个被导出的标识符。标识符必须被导出并声明在该包的包块中。

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

† 11.3　复合字面量

复合字面量用于构造结构体，数组，切片和映射，并且每次计算时都会创建一个新值。通过字面量的类型以及跟随其后的花括弧组成复合字面量。每一个元素前可以加上一个可选的键。

```
CompositeLit  = LiteralType LiteralValue .
LiteralType   = StructType | ArrayType | "[" "..." "]" ElementType |
                SliceType | MapType | TypeName .
LiteralValue  = "{" [ ElementList [ "," ] ] "}" .
```

```
ElementList   = KeyedElement { "," KeyedElement } .
KeyedElement  = [ Key ":" ] Element .
Key           = FieldName | Expression | LiteralValue .
FieldName     = identifier .
Element       = Expression | LiteralValue .
```

LiteralType 的底层类型必须时一个结构体、数组、切片、或者映射类型（语法对此作出强制限制，除非该类型作为 TypeName 给出）。元素和键的类型必须能够直接（无需转化）赋值给相应的字段、元素和字面量类型的键类型。键会被解释为结构体中的字段名、数组和切片字面量的索引和映射字面量的键。对于映射字面量，所有的元素必须有一个键。如果多个元素具有相同的字段名或者常量键值，则会出错。对于非常量映射键，请查看计算顺序章节。

结构体字面量遵循下面规则：

- 键必须是声明在结构体类型中的字段名。
- 不包含任何键的元素列表必须按照字段声明顺序列出元素。
- 如果某个元素具有一个键，则每一个元素都必须有一个键。
- 一个包含键的元素列表不需要为每一个结构体字段都提供元素。省略掉的字段的值为 0。
- 一个子民啊量可能会省略元素列表；这样的字面量会被计算为其类型的零值。
- 为属于不同的包中的结构体中未导出的字段指定一个元素会导致错误。

给出下列声明：

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

可以写出下列字面量：

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

对于数组和切片字面量，适用下列规则：

- 每一个元素都有一个关联的整型索引，用来标记在数组中的位置。
- 具有键的元素会使用键作为索引。键必须是一个可以被类型 `int` 表示的非负常量；并且，如果有类型则必须是一个整型。
- 一个无键的元素的索引为前一个元素的索引加一。如果第一个元素无键，则索引为 0。

对复合字面量取地址，会产生一个指向使用该字面量值初始化的独一无二的变量的指针。

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

数组字面量的长度是字面量类型中指定的长度。如果提供的元素数量少于指定的长度，则缺失的元素会被置为类型为数组元素类型的零值。通过数组索引范围之外的索引来获取元素会导致错误。记号 ...表示一个等于最大元素索引加一的数组长度。

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

一个切片字面量秒速了整个底层数组字面量。因此，切片字面量的长度和容量是最大元素索引加一。一个切片字面量具有如下形式：

```
[]T{x1, x2, … xn}
```

同时是应用于一个数组的切片操作的简写：

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

在数组、切片或映射类型 T 的复合字面量中，为自身复合字面量的元素或映射键可以省略相应的字面量类型（如果该字面量类型等于T 的元素或键类型）。相似的，当元素或键类型为 *T，值为复合字面量地址的元素或键可以省略掉 &T。

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

当使用 LiteralType 中 TypeName 形式的复合字面量作为操作数出现在关键词和 “if”，“for”，或 “switch” 语句的开花括号之间，并且复合字面量没有被圆括号、方括弧，或者花括弧包含时，会导致解析歧义。在这种罕见的情况下，字面量的开括弧会被错误的解释为一个块语句的引入符。为了解决这种歧义，复合字面量必须使用原括弧包含。

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

下面时有效的数组、切片和映射字面量：

```
// 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}
```

```
9
10 // frequencies in Hz for equal-tempered scale (A4 = 440Hz)
11 noteFrequency := map[ string] float32{
12     "C0": 16.35, "D0": 18.35, "E0": 20.60, "F0": 21.83,
13     "G0": 24.50, "A0": 27.50, "B0": 30.87,
14 }
```

† 11.4　函数字面量

一个函数字面量表示一个匿名函数。

```
1 FunctionLit = "func" Signature FunctionBody .
```

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

一个函数字面量可以被赋值给一个变量或者直接调用。

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

函数字面量时闭包的：他们可以使用定义在外层函数内的变量。这些变量可以在外层函数和函数字面量之间共享，并且只要他们而被访问便可以继续存在。

† 11.5　主表达式

主表达式是用作一元和二元表达式的操作数。

```
1  PrimaryExpr =
2      Operand |
3      Conversion |
4      MethodExpr |
5      PrimaryExpr Selector |
6      PrimaryExpr Index |
7      PrimaryExpr Slice |
8      PrimaryExpr TypeAssertion |
9      PrimaryExpr Arguments .
10
11 Selector       = "." identifier .
12 Index          = "[" Expression "]" .
13 Slice          = "[" [ Expression ] ":" [ Expression ] "]" |
14                   "[" [ Expression ] ":" Expression ":" Expression "]" .
15 TypeAssertion  = "." "(" Type ")" .
16 Arguments      = "(" [ ( ExpressionList | Type [ "," ExpressionList ] ) [ "..." ] [
       "," ] ] ")" .
```

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

† 11.6　选择器

对于一个不是包名的组表达式 x，选择器表达式 x.f 表示值 x（有时候会是 *x）的字段或者方法 f。标识符 f 被称为（字段或方法）选择器，必须是一个非空白标识符。选择器表达式的类型是 f 的类型。如果 x 是一个包名，请查看限定标识符章节。

一个选择器 f 可以表示一个类型 T 的一个字段或方法 f，也可以表示 T 的一个嵌套的嵌入字段的字段或方法 f。为了访问 f 而遍历的嵌入字段的数量称为 f 在 T 中的深度。声明在 T 中的一个字段或方法 f 的深度为 0。声明在 T 中的嵌入字段 A 的字段或方法的 f 的深度为 f 在 A 中的深度加一。

下面的规则适用于选择器：

1. 对于类型 T 或 *T（T 不是一个指针或接口类型）的值，x.f 表示 T 最表层深度的 T。如果最表层深度下不只有一个 f 则该选择器表达式非法。
2. 对于接口类型 I 的值 x，x.f 表示动态值 x 中名为 f 的实际方法。在 f 的方法集中，如果没有名为 f 的方法，则该选择器表达式非法。
3. 作为例外，如果类型 x 是个定义的指针类型并且 (*x).f 是一个表示字段（非方法）的有效的选择器表达式，则 x.f 可以作为 (*x).f 的简写。
4. 所有其他情况下，x.f 是非法的。
5. 如果 x 是一个指针类型，并且值为 nil，这时使用 x.f 来表示一个结构体字段、赋值或计算会导致运行时错误。
6. 如果 x 为接口类型，并且值为 nil，则调用或计算方法 x.f 都会引发运行时错误。

比如，给出下列声明：

```
type T0 struct {
    x  int
}

func (*T0) M0()

type T1 struct {
    y  int
}
```

```
10
11 func (T1) M1()
12
13 type T2 struct {
14     z  int
15     T1
16     *T0
17 }
18
19 func (*T2) M2()
20
21 type Q *T2
22
23 var t T2     // with t.T0 != nil
24 var p *T2    // with p != nil and (*p).T0 != nil
25 var q Q = p
```

可以写出下面的选择器：

```
1  t.z          // t.z
2  t.y          // t.T1.y
3  t.x          // (*t.T0).x
4
5  p.z          // (*p).z
6  p.y          // (*p).T1.y
7  p.x          // (*(*p).T0).x
8
9  q.x          // (*(*q).T0).x        (*q).x is a valid field selector
10
11 p.M0()       // ((*p).T0).M0()      M0 expects *T0 receiver
12 p.M1()       // ((*p).T1).M1()      M1 expects T1 receiver
13 p.M2()       // p.M2()              M2 expects *T2 receiver
14 t.M2()       // (&t).M2()           M2 expects *T2 receiver, see section on Calls
```

但是下面的选择器表达式是无效的：

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

† 11.7　方法表达式

如果 `M` 是类型 `T` 的方法集，`T.M` 相当于一个和 `M` 使用相同参数，但会前置一个值为方法接收器的额外参数的函数。

```
1  MethodExpr    = ReceiverType "." MethodName .
2  ReceiverType  = Type .
```

试想一个没有方法的结构体类型 `T`，接收器类型为 `T` 的 `Mv` 和接收器类型为 `*T` 的 `Mp`。

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

var t T
```

表达式 `T.Mv` 会提供一个等价于 `Mv` 的函数，但是会带有显式的接收器作为第一个参数；具有如下签字：

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

该函数可以显式的使用接收器进行正常调用，因此下面五个调用是等价的：

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

相似的，表达式 `(*T).Mp` 提供一个使用如下签字表示的 `Mp` 函数值：

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

对于具有值接收器的方法，可以使用显式指针接收器派生一个函数，因此 `(*T).Mv` 提供了一个使用如下签字表示的函数值：

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

这种函数间接的通过接收器创建一个值作为接收器传递给底层方法；方法不会重写传递给函数调用的值。

最后，将值接收器函数用作一个指针接收器方法是非法的，因为指针接收器方法不在值类型的方法集中。

派生自方法的函数值可以使用函数调用语法格式进行调用，接收器会作为调用的第一个参数。也就是，给出 `f := T.Mv`，`f` 会作为 `f(t, 7)` 调用，而不是 `t.f(7)`。为了构造一个绑定接收器的函数，需要使用函数字面量或这方法值。

从一个接口类型的方法中派生出一个函数值是合法的。结果函数会带有一个具有该接口类型的显式接收器。

† 11.8　方法值

如果表达式 `x` 具有静态类型 `T`，并且 `M` 存在于类型 `T` 的方法集中，则 `x.M` 被称为一个方法值。方法值 `x.M` 是一个可使用于 `x.M` 方法调用一样的参数进行调用的函数值。表达式 `x` 可以在方法值计算的过程中计算并保存；被保存的拷贝值可以在任何调用做为接收器，以便在后续执行中使用。

类型 T 可以是一个接口或非接口类型。

如上所讨论的方法表达式，试想一个有两个方法的结构体类型 T，Mv 的接收器类型为 T，Mp 的接收器类型为 *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
var pt *T
func makeT() T
```

表达式 t.Mv 提供一个类型为 func(int) int 的函数值。下面两个调用是等价的：

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

相似的，表达式 pt.Mp 提供一个类型为 func(float32) float32 的函数值。

和选择器一样，使用指针对带有值接收器的非接口方法进行引用会自动反引用该指针：pt.Mv 等价于 (*pt).Mv。

和方法调用一样，使用一个可取地址的值对带有指针接收器的非接口方法进行引用会自动获取该值的地址：t.Mp 等价于 (&t).Mp。

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

尽管上述示例中使用了非接口类型，但仍可以合法的从接口类型值中创建一个方法值。

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

† 11.9　索引表达式

具有 a[x] 形式的主表达式表示由 x 索引的数组、数组指针、切片、字符串或映射 a 的元素。值 x 被相应的称为索引或映射键。适用于下列规则：

如果 a 不是一个映射：

- 索引必须是一个整型或者是一个无类型常量
- 一个常量索引必须是一个非负且能被类型 int 表示的值
- 一个无类型常量索引会被认为是 int 类型
- 索引 x 需要在 $0 \le x < len(a)$ 范围内，否在将超出范围。

对于数组类型 A：

- 一个常量索引必须在范围内
- 如果 x 在运行时超出范围，将发生运行时错误
- a[x] 为在索引 x 处的数组元素，并且 a[x] 的类型是 A 的元素类型

对于指向数组类型的指针（数组指针）a：

- a[x] 是 (*a)[x] 的简写

对于切片类型 S 的 a：

- 如果 x 运行时超出范围，将会发送运行时错误
- a[x] 时在索引 x 处的切片元素，并且 a[x] 的类型是 S 的元素类型

对于字符串类型 a

- 如果字符串 a 是常量，则常量索引也必须在范围内
- 如果 x 在运行时超出范围，将导致运行时错误发生
- a[x] 是一个在索引 x 处的非常量字节，并且 a[x] 的类型为 byte
- a[x] 无法被赋值

对于类型为映射类型 M 的 a：

- x 的类型必须可以被赋值给 M 的键类型
- 如果映射包含一个使用 x 键的条目，a[x] 是键为 x 的映射元素，并且 a[x] 的类型为 M 的

其余情况 a[x] 是非法的。

类型为 map[K]V 的映射 a 的索引表达式在赋值或初始化时会使用下面的特殊形式：

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

提供额外的无类型布尔值。如果键 x 出现在映射中，则 ok 的值为 true，否则为 false。

向 nil 映射中的一个元素赋值会导致运行时错误。

† 11.10　切片表达式

切片表达式可以从字符串、数组、数组指针或切片中构造出一个子字符串或切片。有两个变种：一个指定高低边界的简单形式和一个同样指出边界容量的完全形式。

• 11.10.1　简单切片表达式

对于字符串、数组、数组指针或切片 a，主表达式 a[low:high] 构造一个子字符串或切片。索引 low 和 high 用来决定操作数 a 的哪些元素可以被选进结果。结果索引起始于 0，且长度等于 high-low。在对数组 a 切片之后

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

切片 s 具有类型 []int，长度 3，容量 4，和元素

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

为方便，仍和索引指标都可以被省略。缺失的 low 索引默认为 0；缺失的 high 则默认为被切操作数的长度。

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

如果 a 是一个数组指针，(*a)[low : high] 可以简写为 a[low : high]。对于数组或字符串，索引指标需要在 $0 \leq low \leq high \leq len(a)$ 范围之内，否则被认为超出范围。对于切片，上索引边界时切片的容量 cap(a) 而非长度。常量索引必须是非负数并且可以被类型 int 表示；对于数组或常量字符串，常量索引指标必须同样在范围内。如果两个索引指标都是常量，则必须满足条件 $low \leq high$。如果索引指标在运行时超出范围，则会发生运行时错误。

除了无类型字符串，如果被切操作数是一个字符串或切片，则切片运算的结果是一个和操作数相同类型的非常量值。对于无类型字符串操作数，结果是一个字符串类型的非常量值。如果被切操作数是一个数组，则该数组必须可取地址，并且切片操作的结果是一个和数组具有相同元素类型的切片。

如果一个有效的切片表达式的被切操作数是一个 nil 切片，则结果仍是一个 nil 切片。否则，如果结果是一个切片则将于操作数共享底层数组。

• 11.10.2　完全切片表达式

对于数组，数组指针，或者切片 a （但非字符串），主表达式

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

构造了一个和简单表达式 a[low : high] 具有相同类型和长度以及元素的切片。除此之外，还将结果切片的容量设置为 max - low。值有第一个索引可以被省略；其默认值为 0。在对一个数组 a 进行切片后

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

切片 t 具有 []int 类型，长度 2，容量 4，并且元素

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

和简单切片表达式一样，如果 a 是一个数组指针，(*a)[low : high : max] 可以简写为 a[low : high : max] 。如果被切操作数是一个数组，则该数组必须可取地址。

索引指标必须满足条件 $0 \le low \le high \le max \le cap(a)$，否则将超出范围。一个常量索引必须是非负切能被 int 类型表示。对于数组，常量索引下标必须在范围内。如果多个索引下标是常量，则表达出来的常量必须在彼此相对范围内。如果索引下标运行时超出范围，则会导致运行时错误。

† 11.11　类型断言

对于接口类型的表达式 x 和类型 T，主表达式

```
x.(T)
```

将断言 x 不是 nil，切存储在 x 中的值为类型 T。记号 x.(T) 被称为一个类型断言。

更精确的讲，如果类型 T 不是一个接口类型，x.(T)将断言 x 的动态类型 x 等于类型 T。这种情况下 T 必须实现类型 x 的接口；否则类型断言将无效，因为 x 无法存储类型 T 的值。如果 T 是一个接口类型，x.(T) 可以断言 x 的动态类型实现了接口 T

如果类型断言正确，表达式的值为存储在 x 中的值，且类型为 T。如果类型断言错误，则会发送运行时错误。换句话说，即便 x 的动态类型只能在运行时被知晓，但在正确的程序中 x.(T) 的类型仍可以被视为 T。

```
var x interface{} = 7          // x has dynamic type int and value 7
i := x.( int)                   // i has type int and value 7

type I interface { m() }

func f(y I) {
// illegal: string does not implement I (missing method m)
    s := y.( string)

// r has type io.Reader and the dynamic type of y must implement both I and io.
    Reader
    r := y.(io.Reader)
    …
}
```

类型断言可以被用在特殊形式的赋值或初始化中

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

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

提供了额外的无类型布尔值。如果断言正确，ok 的值为 true，否则为 false。且 v 的值为 0，类型为 T。这种情况下不会发送运行时错误。

† 11.12　调用

给定函数类型 F 的表达式 f，

```
f(a1, a2, … an)
```

使用参数 a1, a2, ...an 调用 f。除了一个特殊情况，参数必须是可赋值给 F 的形参类型的单值表达，并在被调用之前被计算。表达式的类型是 F 的结果类型。方法调用于此相似，但是方法本身作为立于接收器之上的选择器。

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

在函数调用中，函数值和参数会按照通常顺序进行计算。当这些值被计算后，调用参数会在被调函数开始执行时按值传入。函数的返回参数会在函数返回时按值传递回调用函数。

调用一个 nil 函数会导致运行时错误。

作为特例，如果函数或方法 g 的返回值在数量上相等并各自赋值给另一个函数或方法 f 的参数，然后调用 f(g(parameters_of_g)) 会在 g 的返回值按序绑定到 f 的参数后调用 f。f 的调用不能包含除 g 的调用之外的参数，且 g 至少得有一个返回值。如果 f 最后有一个 …参数，则会被赋予 g 的多余返回值。

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

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

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

如果 x 的方法集包含 m 并且参数可以被赋予 m 的参数列表，则 x.m() 方法调用有效。如果 x 是可寻址的且 &x 的方法集包含 m，(&x).m() 可以简写为 x.m()：

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

† 11.13　传递参数给 ...形参

如果 f 的最后一个形参 p 为具有类型 ...T 的可变形参，则 f 中 p 的类型等于类型 []T。如果调用 f 的时候没有为 p 提供实参，则传递给 p 的值为 nil。不然，所传入的值为类型 []T 的新底层数组的一个新切片，其连续元素为实参，这些实参必须能够赋值给 T。切片的长度和容量为 p 所绑定的参数数量，可能会不同于每一次调用。

给出函数和调用：

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

在 Greeting 的第一次调用中，who 的值为 nil，且在第二次调用中值为 []string{"Joe", "Anna", "Eileen"} 。

如果最后一个参数可以被赋值给切片类型 []T 且形参跟在 ...后面，则可以无修改的作为 ...T 的值传入。这种情况下不会创建新的切片。

给定切片 s 和调用

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

在 Greeting 中，who 和 s 具有相同的底层数组。

† 11.14　运算符

运算符会将操作数绑定到一个表达式中。

```
1 Expression = UnaryExpr | Expression binary_op Expression .
2 UnaryExpr  = PrimaryExpr | unary_op UnaryExpr .
3
4 binary_op  = "||" | "&&" | rel_op | add_op | mul_op .
5 rel_op     = "==" | "!=" | "<" | "<=" | ">" | ">=" .
6 add_op     = "+" | "-" | "|" | "^" .
7 mul_op     = "*" | "/" | "%" | "<<" | ">>" | "&" | "&^" .
8
9 unary_op   = "+" | "-" | "!" | "^" | "*" | "&" | "<-" .
```

比较运算符会在其他地方讨论。对于其他二元运算符，除了运算涉及位移或无类型常量，则操作数类型必须相等。对于只涉及常量的运算，请查看常量表达式章节。

除了移位运算，如果一个操作数时无类型常量，而另一操作数不是，该常量则会被转换为另一个操作数类型。

移位表达式中的右操作数必须具有无符号整数类型或者是一个可以被 uint 表示的无类型常量。如果非常量移位表达式的左操作数是一个无类型常量，且只有移位表达式被左操作数替换，则会被先转换为其假定的类型。

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

• 11.14.1　运算符优先级

一元运算符具有最搞优先级。`++` 和 `|–|` 运算符作为语句而不是表达式，故不在运算符层级之内。相应的，语句 `*p++` 和 `(*p)++` 一样。

二元运算符有五个优先级。乘法运算符绑定更紧密，后面时加法运算符，比较运算符，`&&`（逻辑与），和最后的 `||`（逻辑或）：

优先级	运算符
5	`* / % << >> & &^`
4	`+ - | ^`
3	`== != < <= > >=`
2	`&&`
1	`||`

相同优先级的二元运算符从左向右关联。比如，`x/y*z` 等同于 `(x/y)*z`。

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

† 11.15　算术运算符

算术运算符适用于数值并能提供和第一个操作数相同类型的结果。四个标准算术运算符（`+,-,*,/`）适用于整型，浮点，和复数类型；`+` 同样适用于字符串。按位逻辑和移位运算符只适用于整型。

+	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

• 11.15.1　整型运算符

对于两个整型值 `x` 和 `y`，整型商 `q = x / y` 和余数 `r = x % y` 满足下列关系：

$$x = q \cdot y + r \text{and} |r| < |y|$$

其中 `x / y` 会趋零截断（“被截断的除法”）。

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

该规则的一个特例是，如果被除数 `x` 是最小负 `int` 型的值，商 `q = x / -1` 由于二的补码整型溢出关系等于 `x` 且 `r = 0`：

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

如果除数是一个常量，则必须非零。如果除数在运行时为 0，那么会发生运行时错误。如果被除数非负且除数是 2 的常量幂，则除法可能会被替换为右移，并且余数运算可能会被按位与

操作替换。移位运算符会将左操作数位移右操作数指定的位数。如果左操作数是一个有符号整

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

型且移位运算符则会被实现为算术移位；如果是一个无符号整型，则会被时限为逻辑位移。位移的位数没有上限。位移 n 位表现为每一次位移 1 位，且重复 n 次。结果上，`x << 1` 和 `x*2` 相同，而 `x >> 1` 和趋负无穷截断的 `x/2` 相同。

对于整型操作数，一元运算符 `+`, `-` 和 `^` 的定义如下：

x	相当于 `0 +x`
-x	取负号，相当于 `0 - x`
^x	按位取补，相当于 `m ^ x`，其中 `m` 为无符号 `x` 的所有 bit 位置 1；如果 `x` 为有符号，则 `m = -1`

• 11.15.2　整型溢出

对于符号整型值，`+`、`-`、`*` 和 `<<` 运算会进行模 2^n 处理，其中 n 是无符号整数类型的 bit 宽度。简单的来讲，这些无符号整型操作会舍弃溢出的高 bit 位，且程序可以依赖于这种 “环绕” 特性。

对于有符号整型，`+`、`-`、`*` 和 `<<` 运算可以进行合法的溢出，并且结果值取决于有符号整型的表示、操作和其操作数。当结果溢出时，不会发生异常。编译器不会假定溢出不会发生，从而做出优化。比如：编译器不会假定 `x < x + 1` 永远为 `true`。

• 11.15.3　浮点运算符

对于浮点数和复数，`+x` 等同于 `x`，而 `-x` 为 `x` 的负。浮点数或复数被 0 除的结果超出 IEEE-754 标准，是否发生运行时错误受限于具体实现。

一个实现可能会将多个浮点运算组合为单个伪运算，并且提供一个跨语句，且不同于各自独立进行四舍五入计算所获取的值。一个浮点类型转换会显式的四舍五入到目标类型的精度，防止伪操作舍弃四舍五入。

比如，一些架构会提供 “伪乘和加法”（FMA）指令，该指令在计算 `x*y + z` 时不会在立即对 `x*y` 的结果取约。下列示例中展示了 Go 实现何时会使用这些指令：

```
1 // FMA allowed for computing r, because x*y is not explicitly rounded:
2 r  = x*y + z
3 r  = z;   r += x*y
4 t  = x*y; r = t + z
5 *p = x*y; r = *p + z
6 r  = x*y +  float64(z)
7
8 // FMA disallowed for computing r, because it would omit rounding of x*y:
9 r  =  float64(x*y) + z
```

```
r  = z; r +=  float64(x*y)
t  =  float64(x*y); r = t + z
```

• 11.15.4　字符串连接

字符串可以使用 + 运算符或 += 赋值运算符连接：

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

字符串加法会通过连接操作数来创建一个新的字符串。

† 11.16　比较运算符

比较运算符会比较两个操作数，并提供一个无类型布尔值。

==　等于
!=　不等于
<　小于
<=　小于或等于
>　大于
>=　大于或等于

在任何比较中，第一操作数必须可赋值给第一个操作数的类型，反之亦然。

等性操作符 == 和 != 适用于可比较的操作数。排序运算符 <、<=、和 >= 适用于有序操作数。这些术语以及比较结果定义如下：

- 布尔值是可比较的。仅当两个布尔值均为 true 或 false 时相等。
- 整型值通常意义上，是可比较且有序的。
- 浮点值按照 IEEE-754 标准是可比较且有序的。
- 复数值是可比较的。两个复数值 u 和 v 仅在 real(u) == real(v) 且 imag(u) == imag(v) 时相等。
- 字符串值在词法上按字节时可比较且有序的。
- 指针值时可比较的。两个指针当其指向相同的变量或者值均为 nil 时相等。指向不同的 0 大小变量的指针可能相等也可能不相等。
- 通道值时可比较的。make 调用而创建的两个通道值相等，或者当其值均为 nil 时相等。
- 接口值是可比较的。当两个接口具有等价的动态类型和相等的动态值或值均为 nil 时，两个接口值相等。
- 一个非接口类型 X 的值 x 和接口类型 T 的值 t，当类型 X 的值可以比较是且 X 实现了 T 时可比较。如果 t 的动态类型等价于 X 且 t 的动态值等于 x，x 和 t 相等。
- 当结构体的所有字段都是可比较时，结构体的值可比较。两个结构体相应的非空白字段相等时相等。

- 当数组元素类型的值可比较时，数组值可比较。当两个数组值相应的元素相等时，两个数组值相等。

如果接口值的类型不可比较，则比较两个具有等价动态类型的接口值时会导致运行时错误。这种行为不仅适用于接口值直接比较，也适用于接口值数组或具有接口值字段的结构体比较。

切片、映射和函数值是不可比较的。但是，这些值可以和预声明标识符 `nil` 比较。同样允许将指针、通道和接口值和 `nil` 比较，并遵循上述的通用规则。

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

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

† 11.17　逻辑运算符

逻辑运算符适用于布尔值，并提供和操作数相同类型的结果。右运算符会根据条件进行计算。

`&&`	条件与	`p && q` 为 “if p then q else false”
`||`	条件或	`p || q` 为 “if p then true else q”
`!`	非	`!p` 为 “not p”

† 11.18　地址运算符

对于类型为 `T` 的操作数 `x`，取地址运算 `&x` 会产生一个指向 `x` 的 `*T` 类型的指针。操作数必须时可寻址的，也就是说可以是：一个变量、指针反引用，或切片索引运算、一个可寻址结构体操作数的字段选择器；或者可寻址数组的数组索引运算。作为可寻址的特例，`x` 同时可以是一个复合字面量（可以用圆括弧包含）。如果 `x` 的计算会导致一个运行时错误，那么 `&x` 同样也会导致运行时错误。

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

```

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

† 11.19　接收运算符

对于通道类型的操作数 ch，接收运算 <-ch 的值为从通道 ch 中接收到的值。通道方向必须允许接收运算，且接收运算的类型为通道的元素类型。表达式会一直阻塞到有可获得的值。从一个 nil 通道接收值会一直阻塞。闭合通道上的接收运算永远会立即关闭，并在任何前一个发送的值已被接收后，提供一个元素类型的 0 值。

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

在赋值或初始化中的接收表达式的特殊形式

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

会提供一个报告通信是否成功的无类型布尔结果。如果接收的值被一个成功的发送通道操作传达ok 的值为 true；如果通道是关闭的或为空，则会产生一个 false。

† 11.20　转换

转换是 T(x) 形式的表达式，其中 T 是一个类型且 x 是一个可以被转换为类型 T 的表达式。

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

如果类型起始于运算符 * 或 <-，或者起始于关键词 func 同时没有结果列表，那么必要时必须使用圆括弧包含来避免歧义；

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

如果常量值 x 可以被类型 T 的值表示，则 x 可以转换为类型 T。作为特殊情况，整型常量 x 可以使用和非常量 x 相同的规则转换为一个字符串类型。

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

下列情况下，非常量值 x 可以被转换为类型 T：

- x 可以被赋值给 T。
- 忽视掉结构体标签（见下文）后，x 的类型和 T 具有等价的底层类型。
- 忽略掉结构体标签（见下文）后，x 的类型和 T 是非定义的类型的指针类型，且其指针基类型拥有等价的底层类型。
- x 的类型和 T 均为整型或浮点类型。
- x 的类型和 T 都是复数类型。
- x 是一个整数或字节切片或符文，且 T 是一个字符串类型。

当为了转换而比较结构体类型特征时，结构体标签会被忽视：

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

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

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

特定规则适用于数值类型之间或数值和字符串类型之间的（非常量）转换。这些转换可能会更改 x 的表示，并造成运行时开销。所有其他转换只会更改类型，但不会更改 x 的表示。

没有语言层级的机制可以支持指针和整数之间转换。包 unsafe 在受限的环境下实现这一功能。

• 11.20.1　数值类型之间的转换

对于非常量数值间的转换，适用以下规则：

- 当在整型之间转换时，如果值是一个有符号整数，会进行符扩展至隐式无穷精度，否则使用 0 扩展。结果会被截断为结果类型的大小。比如：`v := uint16(0x10F0)`，此时 `uint32(int8(v)) == 0xFFFFFFF0`。这种转换永远会提供一个有效值，不会出现溢出。
- 将一个浮点数转换整数时，小数部分会被舍弃（趋零截断）。
- 将整数或浮点数转换为浮点类型，或将复数转换为另一个复数类型，结果值会被归约目标类型精度。比如，`float32` 类型的变量 `x` 的值可以使用超过 IEEE-754 32-bit 数的附加精度来存储，但是 `floa32(x)` 表示将 `x` 的值归约到 32-bit 精度。相似的，`x + 0.1` 可以使用超过 32 bits 精度，但是 `float32(x + 0.1)` 不能。

所有涉及浮点或复数值的非常量转换中，如果结果类型不能表示结果值，转换虽然可以成功，但是结果值依赖于具体实现。

• 11.20.2　与字符串类型的转换

1. 将一个有符或无符整数值转换为一个字符串类型，会提供一个包含 UTF-8 表示的整数的字符串。Unicode 编码点有效范围外的值会被转换为 “`\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. 将一个字节切片转换为一个字符串类型会提供一个字符串，其连续字节为切片元素。

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

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

3. 将符文切片转换为一个字符串类型将提供一个独立符文值转换为字符串后相连接的字符串。

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

type MyRunes [] rune
```

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

4. 将一个字符串类型值转换为一个切片类型将提供一个其连续元素为字符串字节的切片。

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

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

5. 将一个字符串类型转换为一个符文类型切片将提供一个包含字符串独立 Unicode 编码点的切片。

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

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

† 11.21　常量表达式

常量表达式只能包含在在编译时计算的常量操作数。

无类型布尔、数值、和字符串常量只要是合法的相应类型就可以被用作操作数。除了移位操作，如果二元运算的操作数是不同种类的无类型常量，结果会使用后出现在列表：整数、符文、浮点、复数种类的无类型常量。比如：一个无类型整数常量被一个无类型复数常量除，结果为无类型复数常量

一个常量比较永远会提供一个无类型布尔常量。如果常量移位表达式的左操作数是一个无类型常量，结果将为整数常量；否则结果为与左操作数相同类型的常量，其必须为整数类型。其他应用于无类型常量结果的所有操作符，将导致相同种类的无类型常量（也就是，一个布尔、整数、浮点、复数、或字符串常量）

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

```
const Δ = Σ + 2.0e-4       //            (untyped complex constant)
const Φ =  iota*1i - 1/1i   //            (untyped complex constant)
```

对无类型整数、符文或浮点常量使用内置函数 `complex` 会提供一个无类型复数常量。

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

常量表达式永远被真实的计算；中间值和常量可能会要求比语言中预声明的类型更高的有效精度。下面时合法的声明：

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

一个常量除法或余数运算的除数不能为 0：

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

有类型常量的值必须能被其类型精确的可表示。下列常量表达式时非法的：

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

非常量值使用一元按位补码运算符 ^ 中使用的掩码需满足规则：无符号常量的掩码全为 1，有符号且无类型常量则为 -1。

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

受制于实现：比那一起在计算无类型浮点或复数常量表达式时可能会进行四舍五入；详情可以查看常量章节中的实现限制。这种四舍五入会导致浮点常量表达式在整数上下文环境中无效，即便可以使用无穷精度进行计算时有效，反之亦然。

† 11.22　计算顺序

在包层级上，初始化之间的依赖决定了变量声明中各初始化表达式的计算顺序。否则，在计算表达式、赋值、返回语句、所有的函数调用、方法调用以及通讯运算符的操作数会从词法上的从左向右的顺序计算。

比如，在以下（function-local）赋值中

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

函数调用和通讯的发生顺序为：f(), h(), i(), j(), <-c, g(), k()。但是这些事件的顺序相对于 x 的所有和计算以及 y 的计算是未指明的。

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

在包层级上，初始化的依赖会重写单个初始化表达式的从左向右计算规则，而不是每一个表达式中的操作数。

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

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

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

函数调用的顺序为 u(), sqr(), v(), f(), v(), g()。

单个表达式中的浮点运算符会根据运算符的结合性。显式的圆括弧会通过重写默认结合性来影响计算。表达式 x + (y + z) 中的 y + z 会在加上 x 之前计算。

第十二章　语句

语句控制着执行。

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

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

† 12.1　终止语句

一个终止语句可以阻止词法上同一块中出现在该语句之后的所有语句执行。下列语句是终止语句：

1. 一个 “return” 或 “goto” 语句。
2. 一个内置函数 `panic` 的调用。
3. 一个一终止语句结束的语句列表的块。
4. 一个 “if” 语句，其中：
   - 具有 “else” 分支，且
   - 两个分支都是终止语句。
5. 一个 “for” 语句，其中：
   - 没有涉及 “for” 语句的 “break” 语句，且
   - 缺省循环条件。
6. 一个 “switch” 语句，其中：
   - 没有涉及该 “switch” 语句的 “break” 语句。
   - 有 default case，且
   - 每一个 case（包括默认 case）中的语句列表都已终止语句结束，或者具有 “fallthrough” 标签。
7. 一个 “select” 语句，其中

- 没有涉及该 “select” 语句的 “break” 语句，且
- 每一个 case 中的语句列表，包括 default 在内，一终止语句结束。

8. 一个标注了一个终止语句的标签语句。

所有其他语句不为终止。

如果语句列表非空，且最后一个非空语句是终止语句，则该语句列表以终止语句结束。

† 12.2　空语句

空语句不做任何事。

```
EmptyStmt = .
```

† 12.3　标签语句

一个标签语句可以是 goto 、break 或 continue 语句的目标。

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

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

† 12.4　表达式语句

除了特定的内置函数外，函数和方法调用以及接收操作可以出现在语句上下文中。这种语句可以使用圆括弧包含。

```
ExpressionStmt = Expression .
```

下列内置函数不允许出现在语句上下文中。

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

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

† 12.5　发送语句

一个发送语句会在通道上发送一个值。通道表达式必须是一个通道类型，通道方向必须允许发送操作，被发送的值的类型必须可赋值给通道的元素类型。

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

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

† 12.6　自增自减语句

“++” 和 “–” 语句会对其操作数自增或自减一个无类型常量 1。和赋值一样，操作数必须可寻址或是一个映射索引表达式。

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

下列赋值语句在语义上相等：

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

† 12.7　赋值

```
Assignment = ExpressionList assign_op ExpressionList .

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

每一个左侧操作数都必须可寻址、映射索引表达式、或者空白标识符（只用于 `=` 赋值）。操作数可以使用圆括弧包含。

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

一个赋值运算 `x op= y`，其中 op 为一个二元算术运算符，等价于 `x = x op (y)`，但是 `x` 只计算一次。`op=` 构造是一个单一记号。在赋值运算中，左右侧表达式列表必须只含有单值表达式，且左侧表达式不能是空白标识符。

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

一个元组赋值可以多值运算的结果独立的赋值给变量列表。有两种赋值形式。第一种，右手侧是单个多值表达式，如函数调用，通道或映射运算，或类型断言。左侧操作数的数量必须匹配值的数量。比如，如果 `f` 是一返回两个值的函数，

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

将第一个值赋值给 `x` 并将第二个值赋值给 `y`。在第二中形式中，操作数的数量必须等于右侧表达式的数量，每一个表达式必须是一个单值表达式，并且右侧第 n 个表达式会赋值给左侧的第 n 个操作数：

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

空白标识符提供一种忽视赋值语句中右侧值的方式：

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

赋值过程有两个阶段。首先，左侧表达式中的索引表达式和指针反引用（包括选择器中隐式的指针反引用）的操作数与右侧表达式会按照正常顺序计算。其次，赋值会按照从左向右的顺序展开。

```
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}
```

赋值中，每一个值必须能够被赋值给其要被赋予操作数的类型，有如下特列：

1. 任何有类型的值可以被赋值给空白标识符。

2. 如果一个无类型常量被赋值给一个接口类型变量或者空白标识符，常量会先转换为他的默认类型。
3. 如果无类型布尔值被赋值给一个接口类型变量或空白标识符符，会先转换为 `bool` 类型。

† 12.8　If 语句

“If” 语句通过一个布尔表达式的值来指明两个分支的条件执行。如果表达式计算为 true，执行 “if” 分支，否则，执行 “else” 分支（如果有的话）。

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

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

表达式前可以有一个简单语句，会在表达式执行前计算。

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

† 12.9　Switch 语句

“switch” 语句提供多路执行。一个表达式或类型说明符会和 “switch” 中的 “cases” 进行比较，从而决定执行哪一个分支。

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

有两种形式的 switch 语句：表达式 switch 和类型 switch。在表达式 switch 中，case 包含和 switch 表达式的值进行比较的表达式。在类型 switch 中，case 包含和 swtich 表达式中特别标注的类型进行比较的类型。swithc 语句中的 swtich 表达式只会计算一次。

• 12.9.1　表达式 switch

在表达式 switch 中，switch 表达式会被计算，且 case 表达式不必为常量，按照从左向右，自上而下的顺序计算；第一个等 switch 表达式的 case 表达式触发与 case 表达式相关联的执行；其余 case 会被跳过。如果没有匹配的 case，并且有 default case，则会执行 defalut case。至多有一个 default case，并可以出现在 “switch” 语句中的任何位置。一个缺失的 switch 表达式等价于布尔值 `true`。

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

如果 switch 表达式计算为一个无类型常量，则会先转换为其默认类型；如果是一个无类型布尔值，则会先转换为 `bool` 类型。预声明无类型值 `nil` 无法在 switch 表达式中使用。

如果一个 case 表达式是无类型的，会先转换为 switch 表达式的类型。对于每一个 case 表达式 `x` （可以是转换后的）和 switch 表达式的值 `t`，`x == t` 必须是一个有效比较。

换一句话说，switch 表达式被当作无显式类型的声明并初始化变量 `t`；`t` 的值会和每一个 case 表达式进行等价性测试。

在一个 case 或 default 子部中，最后一个非空语句可以是一个 “fallthrough” 语句，来表示控制流会进入下一个子部的第一个语句。否则，控制流会进入 “swtich” 语句的结尾处。一个 “fallthrough” 语句可以会出现在除表达式 swtich 最后一个子部的所有子部的最后一个语句。

switch 表达式前面可以有一个简单语句，会在表达式计算之前执行。

```
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()
}
```

受限于实现：一个编译器可能不允许多个表达式计算为相同的常量。比如，当前编译器不允许重复的整数、浮点、或字符串常量出现在 case 表达式中。

• 12.9.2　类型 switch

一个类型 switch 除了比较类型而非值外，和表达式 switch 相似。类型 switch 由一个具有类型断言的特殊 switch 表达式标记，并使用保留字 `type` 而非实际类型：

```
switch x.(type) {
// cases
}
```

case 根据表达式 x 的动态类型来匹配实际的类型 T。和类型断言一样，x 必须是一个接口类型，并且列在 case 中的每一个接口类型 T 必须实现了 x 的类型。列在类型 switch 中的 case 里的类型必须全部不同。

```
TypeSwitchStmt  = "switch" [ SimpleStmt ";" ] TypeSwitchGuard "{" { TypeCaseClause
    } "}" .
TypeSwitchGuard = [ identifier ":=" ] PrimaryExpr "." "(" "type" ")" .
TypeCaseClause  = TypeSwitchCase ":" StatementList .
TypeSwitchCase  = "case" TypeList | "default" .
TypeList        = Type { "," Type } .
```

TypeSwitchGuard 可以包含一个短变量声明。当使用这种形式的时候，变量声明在每一个子部的隐式块的 TypeSwitchCase 末尾。在只有一个类型的 case 子部，变量具有该类型；否则，变量具有 TypeSwitchGuard 中的表达式类型。

除了类型，一个 case 可以使用预声明标识符 nil；当 TypeSwitchGuard 是一个 nil 接口值时会选择该 case。至多有一个 nil case。

给定一个类型为 interface{} 的表达式 x，下列类型 switch：

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

可以重写为：

```
v := x  // x is evaluated exactly once
if v ==  nil {
    i := v                                 // type of i is type of x (interface{})
    printString("x is nil")
} else if i, isInt := v.( int); isInt {
    printInt(i)                            // type of i is int
} else if i, isFloat64 := v.( float64); isFloat64 {
    printFloat64(i)                        // type of i is float64
} else if i, isFunc := v.(func( int)  float64); isFunc {
    printFunction(i)                       // type of i is func(int) float64
} else {
    _, isBool := v.( bool)
```

```
    _, isString := v.( string)
    if isBool || isString {
        i := v                          // type of i is type of x (interface{})
        printString("type is bool or string")
    } else {
        i := v                          // type of i is type of x (interface{})
        printString("don't know the type")
    }
}
```

类型 switch guard 前面可以放上一个简单语句，该语句会在 guard 计算之前执行。

“fallthrough” 语句不允许出现在类型 switch 中。

† 12.10　For 语句

一个 “for” 语句表示一个块的重复执行。有三种形式：迭代可以被一个单一条件、一个 “for” 子部、或一个 “range” 子部控制。

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

• 12.10.1　具有但条件的 for 语句

在这种最简单的形式中，一个 “for” 语句表示只要布尔条件计算为真，就会重复块执行。条件会在每次迭代之前执行。如果条件缺省，等价于布尔值 `true`。

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

• 12.10.2　带有 for 子部的 For 语句

带有 ForClause 的 “for” 语句也被其条件控制，但会有额外的初始和后置语句，比如一个赋值，一个自增或自减语句。初始语句可以是一个短变量声明，但是后置语句不可以。初始语句中声明的变量可以在每次迭代中重复使用。

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

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

如果非空，初始语句会在第一迭代计算条件之前执行一次；后置语句则会在每一次块执行（且只要块被执行）后执行。ForClause 的任何元素都可以为空，但是除非只有一个条件时，分号是必须的。如果条件缺省，则等价于布尔值 `true`

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

• 12.10.3　带有 range 子部的语句

一个带有 “range” 子部的 “for” 语句会迭代数组、切片、字符串、或映射的所有条目或在通道中接收的所有值。每一个条目会将迭代值被赋值给迭代变量（如果有的话）。

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

“range” 子部右侧表达式被称为 range 表达式，可以是一个数组、数组指针、切片、字符串、映射、或者允许接收操作的通道。如果作为操作数出现在赋值左侧，则必须是可寻址的，或为映射索引表达式；他们表示迭代变量。如果 range 表达式是要给通道，则至多允许一个迭代变量，否则可以多达两个。如果最后一个迭代变量是一个空白标识符，range 子部等价于无该标识符的相同子部。

range 表达式 `x` 只会在循环开始前计算一次，有一个特例：如果至多有一个迭代变量并且 `len(x)` 是一个常量，则不会计算 range 表达式。

左侧的函数调用在每次迭代中都会计算一次。在每一次迭代中，如果提供了相应的迭代变量，迭代值会按照如下方式提供：

```
Range expression                          1st value          2nd value

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

1. 对于数组、数组指针、切片值 `a`，索引迭代值会按照递增顺序从零提供。如果最多出现了一个迭代变量，range 循环会提供从 0 到 `len(a) - 1` 的迭代值，并且不会索引到数组或切片自身。对于 `nil` 切片，迭代次数为 0。
2. 对于字符串值，“range” 子部会从字节索引 0 处迭代字符串中的 Unicode 编码点。在连续的迭代中，索引值必须是字符串中连续 UTF-8 编码点的第一个字节的索引，而第二个 `rune` 类型的值将会是相应的编码点。如果迭代中遇到一个无效的 UTF-8 序列，第二个值将为 `0xFFFD`（Unicode 替换字符），并且下一个迭代会提前字符串一个字节。
3. 映射的迭代顺序是未指定的，并且不能保证下一次迭代时顺序相同。如果一个映射条目尚未获取，则将会在迭代中移除并且不提供相应的迭代值。如果在迭代中创建了一个映射条目，该条目可能会在迭代中提供也可能会被跳过。这种选择会在不同的迭代中有所不同。如果映射是 `nil`，迭代次数为 0。

4. 对于通道，提供的迭代值为被发送到通道中的连续值，直到通道被关闭。如果通道是 `nil`，range 表达式会永远的阻塞。

和赋值语句一样会将迭代值赋值给相应的迭代变量。

由 “range” 子部声明的迭代变量可以使用短变量声明形式（`:=`）。在这种情况下，他们的类型会被设置为相应迭代值的类型，并且他们的作用域为 “for” 语句块；并可以在每次迭代中重复使用。如果迭代变量声明在 “for” 语句外面，执行完成后，这些变量的值将变成最后一次迭代的值。

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

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

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

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

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

† 12.11　Go 语句

一个 “go” 语句会作为独立的并行线程控制（或 goroutine）在同一个地址空间中执行一个函数调用。

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

其中表达式必须是一个函数或方法调用；不能被圆括弧包含。调用内置函数的限制和表达式语句一样。

函数值和参数会和 goroutine 调用一样计算，但是不想常规调用，程序执行不会等待被调函数完成。而是，使用新的 goroutine 独立执行。当函数终止时，其 goroutine 也会终止。如果函数具有任何返回值，则会在函数完成时舍弃。

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

† 12.12　选择语句

一个 “select” 语句会选择一组可能会处理的 send 或 receive 操作。类似与 “switch” 语句，但是这一次所有的 case 都是通讯操作。

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

一个带有 RecvStmt 的 case 会将 RecvExpr 的结果赋值给一个或两个变量，这些变量可以使用短变量声明来声明。RecvExpr 必须使用接收操作（可以使用圆括弧包含）。至多有一个默认 case，并且可以出现在 case 列表中的任何位置。

执行一个 “select” 语句会有以下几个步骤：

1. 对于语句中的所有 case，一进入 “select” 语句，接收操作的通道操作数和通道以及发送语句的右手侧表达式就会按照源顺序计算一次。结果为接收或发送的通道的集合，以及相应的发送值。不管通讯操作是否被选择处理，计算时的任何副作用都会发送。RecvStmt 左手侧的表达式具有短变量声明或者尚未计算的赋值。
2. 如果一个或多个通讯可以被处理，则会通过伪随机算法选择其中的一个。否则，如果有 default case，则会选则该 case。如果没有 default case，“select” 语句会一直阻塞到至少有一个通讯可以被处理。
3. 除非被选择的 case 是 default case，则会执行各自的通讯操作。
4. 如果被选择的 case 是一个带有短变量声明或赋值的 RecvStmt，左手侧表达式会被计算，或者接收值会被赋值。
5. 执行被选择的 case 的语句列表。

由于 nil 通道的通讯从不被处理，一个只有 nil 通道并且没有 default case 的选择会一直阻塞。

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

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

select {}  // block forever
```

† 12.13　返回语句

函数 F 的返回语句会终止 F 的执行，并且提供一个或多个可选的结果值。任何被 F 推迟的函数会在 F 返回到调用者之前执行。

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

在一个无结果类型的函数中，返回语句不能指定任何结果值。

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

有三种方式从一个带结果类型的函数中返回值：

1. 返回值可以显式的列在返回语句中。每一个表达式必须是单值表达式并且可以赋值给相应的函数结果类型中的元素。

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

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

2. 返回语句中的表达式列表可以是一个多值函数调用。其效果过相当于每一个从函数中返回的值被赋值给一个具有相应类型的临时变量，返回语句后面的这些变量适用于前一种情况的规则。

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

3. 如果函数的结果类型为其结果参数指定了名字，则表达式列表可以为空。结果参数和普通的本地变量一样，并且函数可以在必要的时候将值赋值给他们。最终的返回语句将会返回这些变量的值。

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

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

不管如何声明，所有的结果值都会在进入函数时初始化为相应类型的 0 值。一个返回语句会在任何延迟函数执行之前指定结果参数集。

实现限制：如果在返回语句范围内，作为结果参数，不同的条目（常量，类型，变量）具有相同的名字，编译器可能会不允许返回语句使用空表达式列表。

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

† 12.14　Break 语句

一个 “break” 语句会终止相同函数内 “for”，“switch”，“select” 语句的执行。

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

如果有标签，则必须被 “for”，“switch”，或 “select” 语句包含，并且将成为执行终止处。

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

† 12.15　Continue 语句

一个 “continue” 语句会从（for 语句的）后置语句开始最内层 “for” 循环的下一次迭代。“for” 循环必须在相同的函数内。

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

如果有标签，则必需是一个闭合的 “for” 语句，也就是说该标签为 “for” 语句的执行开始处。

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

† 12.16　Goto 语句

一个 “goto” 语句会将控制移交给相同函数内相应的标签语句。

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

```
goto Error
```

“goto” 语句的执行不能在该点范围内引入任何变量。比如，该示例中：

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

会因为标签 `L` 跳过了 `v` 的创建而发生错误。

块外层的 “goto” 语句无法跳入该块内的标签，比如，该示例中：

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

会因为 `L1` 位于 “for” 语句块中，而 “goto” 不是，从而发送错误。

† 12.17　Fallthrough 语句

一个 “fallthrough” 语句会将控制转移给 “switch” 语句中的下一个 case 子块。该语句只能用作这种子块的最后一个非空表达式。

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

† 12.18　Defer 语句

一个 “defer” 语句将一个函数推迟到外围函数返回时执行，这种执行不管外围函数是执行了返回语句、还是到达函数体的末尾、或者相应的 goroutine 崩坏。

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

表达式必须是一个函数或方法调用，无法被圆括弧包含。内置函数的调用具有表达式语句一样的限制。

每次执行 “defer” 语句，函数值和参数会和通常一样执行，并重新保持，但是并不调用实际函数。推迟的函数会在外层函数返回之前按照与推迟顺序相反的顺序立即调用。如果推迟的函数计算为 `nil`，在调用该函数时会造成崩坏，而不是在执行 “defer” 语句时。

比如，如果推迟的函数是一个函数字面量并且外层函数具有在该字面量范围内的命名的结果参数，推迟的函数可以在命名结果参数返回之前修该他们。如果推迟的函数具有任何返回值，则会在函数完成时舍弃他们。（See also 处理崩坏章节）。

```
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
}
```

第十三章　内置函数

内置函数是预声明的。可以和其他任何函数一样调用，但其中部分内置函数使用一个类型而不是表达式来作为其第一个参数。

内置函数没有标准 Go 类型，因此只能出现在调用表达式中；他们不能用作函数值。

† 13.1　Close

对于通道 `c`，内置函数 `close(c)` 表示不会有更多的值会被发送到通道里。如果 `c` 是一个只接收通道，则会发送错误。向一个已关闭的通道或关闭一个已关闭的通道会导致运行时崩坏。关闭 `nil` 通道会导致运行时崩坏。调用 `close` 之后，以及任何之前发送的值已经被接收之后，接收操作会无阻塞的返回一个 0 值。不论通道是否关闭，多值接收操作会返回带有指示的接收值。

† 13.2　长度和容量

内置函数 `len` 以及 `cap` 使用不同类型作为参数，并返回 `int` 类型的结果。实现要保证结果永远能够纳入 `int` 型。

调用	参数类型	结果
`len(s)`	`string`	字符串字节单位长度
	`[n]T, *[n]T`	数组长度，等于 n
	`[]T`	切片长度
	`map[K]T`	映射长度（已定义的 key 的数量）
	`chan T`	通道缓冲中排队的元素数量
`cap(s)`	`[n]T, *[n]T`	数组长度，等于 n
	`[]T`	切片容量
	`chan T`	通道缓冲容量

切片的容量为底层数组申请的元素数量。任何时候都具有关系：$0 \le len(s) \le cap(s)$。值为 `nil` 的切片、映射或通道的长度为 0。一个值为`nil` 的切片或通道的容量为 0。

如果 `s` 是一个字符串常量，表达式 `len(s)` 则为常量。如果类型 `s` 是一个数组或数组指针并且表达式 `s` 不包含通道接收或（非常量）函数调用，表达式 `len(s)` 和 `cap(s)` 则为常量。否则，`len` 和 `cap` 的调用是非常量的，并且会计算 `s`。

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

† 13.3　申请

内置函数 new 使用类型参数 T 来为该类型的变量申请运行时存储，并且返回一个指向该存储的 *T 类型的值。变量会和初始化值章节中一样初始化。

```
 new(T)
```

比如：

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

会为类型 S 的变量申请存储，初始化（a=0, b=0.0），并返回包含该位置地址，类型为 *S 的值。

† 13.4　创建切片，映射和通道

内置函数 make 使用类型 T 作为参数，该类型必须为切片、映射或通道类型，后面可以追加特定于类型的表达式列表。make 函数将返回类型 T （非 *T）的值。内存会按照初始值章节中描述的方式初始化。

调用	类型 T	结果
make(T, n)	切片	类型为 T，长度为 n 且容量为 n 的切片
make(T, n, m)	切片	类型为 T，长度为 n 且容量为 m 的切片
make(T)	映射	类型为 T 的映射
make(T, n)	映射	初始空间约为 n 个元素的类型 T 的映射
make(T)	通道	无缓冲类型为 T 的通道
make(T, n)	通道	类型为 T 的有缓冲通道，缓冲大小为 n

每一个尺寸参数 n 和 m 必须是一个整数类型或无类型常量。一个常量尺寸参数必须是非负的，并可以表示为一个 int 类型的值；如果是无类型常量，则给定类型为 int。如果 n 和 m 都

是常量，则 n 必须不大于 m。如果运行时 n 是负数或比 m 大，则会导致运行时崩溃。

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

使用用映射类型和大小 n 来调用 make 以创建一个带有 n 个映射元素初始化空间的映射。更精确的行为依赖于实现。

† 13.5　追加以及拷贝切片

内置函数 append 和 copy 可以辅助常规切片操作。两个函数，不管被参数引用的内存是否重叠，结果都是独立的。

可变参数函数 append 追加零个或多个值 x 到一个类型为 S 的变量 s 中，其中 S 必须是切片类型，并且返回的结果也必须为类型 S。传递给类型参数类型为 ...T 的值 x（T 为 S 的元素类型）会应用相应的参数传递规则。作为特例，append 的第一个参数也可以是可赋值给类型 []byte 的值，第二个参数为跟在 ... 后面的字符串类型。这种形式会将字节追加到字符串中。

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

如果 s 的容量不足以容纳额外的值，append 会申请一个新的、足够大的底层数组来同时容纳已存在的切片元素和额外的值。否则，append 会复用底层数组。

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

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

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

函数 copy 会将切片元素从源 src 拷贝到目的 dst 中，并返回被拷贝的元素数量。两个参数必须具有相等的元素类型 T 并必须可被赋值给类型为 []T 的切片。被拷贝的元素数量为 len(src) 和 len(dst) 中最小的一个。作为特例，copy 也可以接收可以被赋值给类型 []byte 的目标参数和一个字符串类型作为源参数。这种形式会将字符串中的字节拷贝到字节切片中。

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

示例：

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

† 13.6　映射元素的删除

内置函数 delete 会一个映射 m 中移除键 k。k 的类型必须可以赋值给 m 的键类型。

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

如果映射 m 是 nil 或者元素 m[k] 不存在，delete 不做任何操作。

† 13.7　操作复数

有三个函数可以用来组合或解析复数。内置函数 complex 会从一个浮点实部和虚部构造一个复数值，其中 real 和 imag 会将复数值的实部和虚部。

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

参数类型和返回值类型是相对应的。对于 complex，两个参数必须具有相同的浮点类型，并且返回值类型是由相应的浮点类型构造出来的复数类型：comlex64 对应 float32 类型参数，并且 complex128 对应 float64 类型参数。如果其中一个参数计算为无类型常量，则必须先转换为其他参数类型。如果两个参数都计算为无类型常量，他们必须是非复数或者虚部必须为零，然后函数返回值是一个无类型复数常量。

对于 real 和 imag，参数必须是一个复数类型，然后返回类型是对应的浮点类型：float32 对应 complex64 参数，float64 对应 complex128 参数。如果参数计算为无类型常量，则必须为一个数值，函数返回为一个无类型浮点常量。

real 和 imag 函数一起形成 complex 的逆，对于复数类型 Z 的值 z，有 z == Z(complex(real(z), imag(z)))。

如果这些函数的操作数均为常量，则返回值也是一个常量。

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

† 13.8　处理 panics

两个内置函数，panic 和 recover，用于辅助报告和处理运行时崩溃和程序定义的错误条件。

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

当执行函数 F 时，一个显式的 panic 调用或一个运行时崩溃会终止 F 的执行。任何被 F 延迟的函数可以被正常执行。然后，执行仍和被 F 的调用者延迟的函数，以此类推直到正在执行的 goroutine 中顶层函数延迟的任何函数。这时，程序被终止，并且将报告错误条件，其中包括 panic 的参数值。终止序列被称为 *panicking*

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

recover 函数允许一个程序管理一个正在崩坏的 goroutine 行为。假定一个函数 G 延迟一个调用 recover 的函数 D 且在相同的 goroutine 中执行 G 时发生了崩坏。当延迟函数执行到 D 时，D 所调用的 recover 的返回值为传入 panic 的参数值。如果 D 正常返回，而不开始一个新的崩坏，将会停止崩坏序列。这种情况下，关于 G 和 painc 调用之间的函数状态会被舍弃，并恢复正常执行。然后，执行任何在 D 之前被 G 延迟的函数并通过返回到调用者来终止 G 的执行。

如果遇到以下条件，recover 的返回值为 nil：

- panic 的参数为 nil；
- goroutine 未处于崩坏状态；
- recover 没有被延迟函数直接调用。

在下面的示例中，protect 函数调用函数参数 g 并且保护调用者免于g 引发的运行时崩溃。

```
func protect(g func()) {
    defer func() {
```

```
3           log.Println("done")  // Println executes normally even if there is a panic
4           if x :=  recover(); x !=  nil {
5               log.Printf("run time panic: %v", x)
6           }
7       }()
8       log.Println("start")
9       g()
10  }
```

† 13.9　自举

在自举过程中当前实现提供几个有用的内置函数。这些函数是为了完整性而出现在文章中，并不保证出现在语言里。他们不返回结果。

函数	行为
`print`	打印所有的参数；参数格式依赖于实现
`println`	类似于 `print` 但会打印传参数间的空格并在结尾处打印换行

受制于实现：`print` 和 `println` 不必接收任意参数类型，但是打印布尔、数值和字符串类型必须被支持。

第十四章　包

Go 程序由链接在一起的包组成。一个包则由一个或多个声明了属于包的常量、类型、变量和函数的源文件，这些声明物可以被相同包内的所有文件访问。这些元素可以被导出，并在其他包内使用。

† 14.1　源文件组织

每一个源文件由定义了其所属的包的包子部组成，后面跟上可能为空集的表示哪些包期望使用的导入声明，在跟上可能为空集的函数、类型、变量和常量的声明。

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

† 14.2　包子句

一个包子句作为每一个源文件的开始，定义了该文件所属的包。

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

其中 PackageName 不能为空白标识符。

```
package math
```

一组共享相同 PackageName 的文件形成了一个包的实现。具体实现可能会要求一个包的所有源文件位于相同的目录下。

† 14.3　导入声明

一个导入声明表面源文件中包含依赖于导入的包的功能的声明，并使之能够访问该包中导出的标识符。导入会命名一个用来访问的标识符（PackageName）以及表示被导入的包的 ImportPath。

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

PackageName 被用在限定标识符中，用来访问导入的源文件中导出的标识符。PackageName 需要声明在文件块中。如果忽略 PackageName，则默认为导入的包中的包子句中使用的标识符。如果出现的是显式的句点（.）而不是一个名字，则被导入的包的包块处声明导出的标识符将声明在导入源文件的文件块中，并且禁止使用限定符来访问。

ImportPath 的解释依赖于实现，但一般为被编译的包的文件全名的子字符串，并可能使用相对于已安装的包的仓库的相对路径。

受制于实现：一个编译器可能限制 ImportPath 为非空字符串并只能使用属于 Unicode 的 L、M、N、P 和 S 通用类别（无空格的图形字符）的字符，同时排除字符 `!"#$%&'()*,:;<=>?[\]^`{|}` 和 Unicode 替换字符 U+FFFD

假定我们有已编译的包，包含包子句 `package math`，其中导出函数 `Sin`，并将已编译的包安装在 “`lib/math`”。下表阐述了 `Sin` 在各种类型的导入声明后，如何在文件中进行访问。

导入声明	Sin 的本地名
`import "lib/math"`	`math.Sin`
`import m "lib/math"`	`m.Sin`
`import . "lib/math"`	`Sin`

一个导入声明声明了导入和被导入的包之间的依赖关系。直接或间接的导入包自身，或直接导入一个包而不使用其中任何导出的标识符都是非法的。如果仅是为了导入包的副作用（初始化），可以使用空白标识符作为显式的包名。

† 14.4　包示例

下面是一个实现了并发素数筛选的完整 Go 包。

```
1  package main
2
3  import "fmt"
4
5  // Send the sequence 2, 3, 4, … to channel 'ch'.
6  func generate(ch chan<-  int) {
7      for i := 2; ; i++ {
8          ch <- i  // Send 'i' to channel 'ch'.
9      }
10 }
11
12 // Copy the values from channel 'src' to channel 'dst',
13 // removing those divisible by 'prime'.
14 func filter(src <-chan  int, dst chan<-  int, prime  int) {
15     for i := range src {  // Loop over values received from 'src'.
16         if i%prime != 0 {
17             dst <- i  // Send 'i' to channel 'dst'.
18         }
```

```
    }
}

// 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()
}
```

第十五章　程序初始化和执行

† 15.1　0 值

当为一个变量申请存储空间是，不管是通过声明还是 `new` 调用，亦或是，当创建新值时，使用的时复合字面量或者 `make` 调用，如果没有提供显式的初始化，变量或值会给予一个默认值。每一个这种变量或值的元素会被设为其类型的零值：`false` 用于布尔型，`0` 用于数值类型，`""` 则为字符类型，最后 `nil` 用于指针、函数、接口、切片、通道和映射类型。初始化操作会递归进行，因此对于一个结构体数组中的每一个元素在不提供初始值的情况下都会有零填充的字段。

下面两个简单声明是等价的：

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

然后

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

具有下面的效果：

```
t.i == 0
t.f == 0.0
t.next ==  nil
```

下面的声明也如此

```
var t T
```

† 15.2　包的初始化

在一个包中，包级别的变量会按照声明的顺序初始化，但必须在他们所依赖的任何变量初始化完成后。

更准确的来讲，如果一个包级别的变量没有初始化变量或者其初始化变量没有依赖于未初始化的变量，可以被认为是可即将初始化的（ready for initialization）。通过不断重复的初始化下一个最早位于声明序列且已准备好初始化的包级变量来进行包初始化，直到再也没有准备好初始化的变量。

如果当该过程结束时仍有变量未初始化，这些变量可能是一个或多个初始化循环中的一部分，并且程序无效。

在多个文件中的变量声明的顺序由展示给编译的文件顺序决定：声明在第一个文件中的变量在声明于第二个文件中的变量之前，等等。

依赖分析不会依靠变量的实际值，只取决于在源中的词法引用，并传递性的分析。比如，如果变量 x 的初始表达式涉及到一个函数体中使用了变量 y 的函数，然后 x 依赖于 y。特别的是：

- 对一个变量或函数的引用表示该变量或函数。
- 对方法 `m` 的引用是一个形如 `t.m` 的方法值或方法表达式，其中 `t` 的静态类型不为接口类型，且方法 `m` 存在于 `t` 的方法集中。并不关心结果函数值 `t.m` 是否被调用。
- 如果一个变量、函数或方法 `x` 的初始表达式或函数体（针对函数和方法）包含对变量 `y` 的引用或者包含一个依赖于 `y` 的函数或方法的引用，则 `x` 依赖于 `y`。

依赖分析会在每一个包中进行；只考虑对当前包中声明的变量、函数和方法。

比如，给出下列声明

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

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

初始化顺序为 `d, b, c, a`。

变量可以使用声明在包块中名为 `init` 的函数初始化，无需参数和结果参数。

```
func init() { … }
```

每个包中，甚至单个源文件中都可以定义多个这种函数。在包块中，`init` 标识符只能用来声明 `init` 函数，然而标识符本身并没有被声明。因此 `init` 函数无法在程序的其他地方被使用。

一个无导入的包通过对包级别变量赋予初始值，以及后面跟着按源文件中出现的顺序（如果有多个文件，则按照展示给编译器的文件顺序）进行调用的 `init` 函数。如果包具有被导入的包，则会在初始化包自身前，先初始化被导入的包。如果多个包导入一个包，则被导入的包只会初始化一次。包的导入，通过这种方法来保证没有循环初始化依赖。

包初始化 – 变量初始化和 `init` 函数调用 – 一次一个包的连续的发生在单个 goroutine 中。一个 `init` 函数可能会发生于其他 goroutine 中，可以和初始化代码并发执行。但是，初始化永远会序列化 `init` 函数：直到前一个返回之前是不会调用下一个的。

为了确保初始化行为的可重复性，build 系统建议将归属于同一个包的多个文件按照文件名字典顺序展示给编译器。

† 15.3　程序执行

一个完整的程序通过连接一个单一的，未导入的名为 main 的包和所有它导入的包，并以此传递来创建。主包的名字必须为 main，并且必须声明一个无参且无返回值的 `main` 函数。

```
1 func main() { ··· }
```

程序通过初始化主包，然后调用 `main` 函数来开始执行。当函数 `main` 函数调用返回时，程序退出。不会等待其他非主 goroutines 的完成。

第十六章　错误

预声明类型 `error` 定义如下：

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

这是一个常规的用来表示一个错误条件的接口，使用 `nil` 值来表示没有错误。比如，一个从文件中读取数据的函数可以定义为：

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

第十七章　运行时崩溃

尝试使用超出边界的索引来访问数组会触发运行时崩溃，这种执行错误等价于使用实现定义的接口类型 `runtime.Error` 参数来对内置函数 `painc` 进行调用。这种类型满足预声明接口类型 `error`。用来表示不同运行时错误条件的实际错误值是未指明的。

```
1 package runtime
2
3 type Error interface {
4      error
5     // and perhaps other methods
6 }
```

第十八章　系统考量

† 18.1　包 unsafe

内置包 unsafe，为编译器所知，并通过导入路径 “unsafe” 来获取，提供了用于低级变成包括那些违反类型系统的操作。一个使用了 unsafe 的包必须手动的审查其类型安全性并且可能不可移植。该包提供了以下接口：

```
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
```

`Pointer` 是一个指针类型，但是 `Pointer` 值可能不能被反引用。任何指针或底层类型 `uintptr` 的值都可以被转换为底层类型 `Pointer` 的类型，反之亦然。`Pointer` 和 `uintptr` 之间的转换效果是由实现定义的。

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

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

var p ptr =  nil
```

函数 `Alignof` 和 `Sizeof` 使用任何类型的表达式 `x` 作为参数，并返回假定的变量 `v`（就好像 `var v =x` 来声明的一样）的相应的对齐或尺寸。

函数 `Offsetof` 的参数为一个选择器（可以使用圆括弧）`s.f`，表示由 `s` 或 `*s` 表示结构体的字段 `f`，返回该字段按字节相对于结构体地址的偏移。如果 `f` 是一个嵌入的字段，则必须是可以无需通过结构体字段的间接指针访问。对于具有字段 `f` 的结构体 `s`：

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

计算机架构可能会要求内存地址对齐；也就是说，一个变量的地址为一个因子的倍数，该因子为变量类型的对齐。函数 `Alignof` 需要一个参数，该参数为一个表示任何类型的变量的表达式,，并按字节返回该变量（类型）的对齐。对于一个变量 `x`：

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

调用 `Alignof`，`Offsetof`，和 `Sizeof` 是 `uintptr` 类型的编译时常量表达式。

† 18.2　大小和对齐保证

对于数值类型，可以保证为下列大小：

type	size in bytes
byte, uint8, int8	1
uint16, int16	2
uint32, int32, float32	4
uint64, int64, float64, complex64	8
complex128	16

可以保证下列最小对齐属性：

1. 对于任何类型的变量 `x`：`unsafe.Alignof(x)` 最少为 1。
2. 对于结构体类型变量 `x`：`unsafe.Alignof(x)` 为 `x` 中每一个字段 `f` 的 `unsafe.Alignof(x.f)` 中最大的值，但至少为 1。
3. 对于数组类型变量 `x`：`unsafe.Alignof(x)` 和数组元素类型变量的对齐一样。

一个结构体或数组类型如果没有尺寸大于 0 的字段（或元素），则大小为 0。两个不同的 0 值变量在内存中可能具有相同的地址。