1.6

Go 语言详解

《学习笔记》第五版 上册

二. 类型

1. 变量

在数学概念中，变量（variable）表示没有固定值，可改变的数。但从计算机系统实现角度来看，变量是一段或多段用来存储数据的内存。

作为静态类型语言，Go 变量总是有固定的数据类型，类型决定了变量内存的长度和存储格式。我们只能修改变量值，无法改变类型。

> 通过类型转换或指针操作，我们可用不同方式修改变量值，但这并不意味着改变了变量类型。

因为内存分配发生在运行期，所以在编码阶段我们用一个易于阅读的名字来表示这段内存。实际上，编译后的机器码从不使用变量名，直接透过内存地址来访问目标数据。保存在符号表中的变量名等信息可被删除，或用于输出更详细的错误信息。

定义

关键字 var 用于定义变量，和 C 不同，类型被放在变量名后。另外，运行时内存分配操作会确保变量自动初始化为二进制零值（zero value），避免出现不可预测行为。如显式提供初始化值，可省略变量类型，由编译器推断。

```
var x int           // 自动初始化为 0。
var y = false       // 自动推断为 bool 类型。
```

可一次定义多个变量，包括用不同初始值定义不同类型。

```
var x, y int                // 相同类型的多个变量。
var a, s = 100, "abc"       // 不同类型初始化值。
```

依照惯例，建议以组方式整理多行变量定义。

```
var (
    x, y int
    a, s = 100, "abc"
)
```

简短模式

除 var 关键字外，还可使用更加简短的变量定义和初始化语法。

```
func main() {
    x := 100
    a, s := 1, "abc"
}
```

只是要注意，简短模式（short variable declaration）有些限制：

- 定义变量，同时显式初始化。
- 不能提供数据类型。
- 仅能用在函数内部。

对于粗心的新手，这可能会造成意外错误。比如原本打算修改全局变量，结果变成重新定义同名局部变量。

```
var x = 100

func main() {
    println(&x, x)          // 全局变量。

    x := "abc"              // 重新定义和初始化同名局部变量。
    println(&x, x)
}
```

输出：

```
0xae020             100     // 对比内存地址，可以看出是两个不同的变量。
0xc820041f38        abc
```

简短定义在函数多返回值，以及 if/for/switch 等表达式中定义局部变量非常方便。

简短模式并不总是重新定义变量，也可能是部分退化的赋值操作。

```
func main() {
    x := 100
    println(&x)

    x, y := 200, "abc"     // 注意，x 退化为赋值操作，仅有 y 是变量定义。

    println(&x, x)
    println(y)
}
```

输出：

```
0xc820041f28
0xc820041f28 200           // 对比变量内存地址，可以确认 x 属于同一变量。
abc
```

退化赋值的前提条件是：最少有一个新变量被定义，且必须是同一作用域。

```
func main() {
    x := 100
    println(&x)

    x := 200               // 错误: no new variables on left side of :=
    println(&x, x)
}
```

```
func main() {
    x := 100
    println(&x, x)

    {
        x, y := 200, 300           // 不同作用域，全部是新变量定义。
        println(&x, x, y)
    }
}
```

输出：

```
0xc820041f30       100
0xc820041f38       200  300
```

在处理函数错误返回值时，退化赋值允许我们重复使用 err 变量，这是相当有益的。

```
package main

import (
    "log"
    "os"
)

func main() {
    f, err := os.Open("/dev/random")
    ...

    buf := make([]byte, 1024)
    n, err := f.Read(buf)          // err 退化赋值，n 新定义。
    ...
}
```

多变量赋值

在进行多变量赋值操作时，首先计算出所有右值，然后再依次完成赋值操作。

```
func main() {
    x, y := 1, 2
    x, y = y+3, x+2        // 先计算出右值 y+3、x+2，然后再对 x、y 变量赋值。

    println(x, y)
}
```

输出：

```
$ go build && ./test

5  3

$ go tool objdump -s "main\.main" test

TEXT main.main(SB) test.go
    MOVQ $0x1, AX                  // 先使用 AX、CX 寄存器完成表达式 x+2、y+3 操作。
    MOVQ $0x2, CX
    ADDQ $0x3, CX
    ADDQ $0x2, AX
    MOVQ CX, 0x10(SP)              // 然后将计算结果分别写入 x、y 变量。
    MOVQ AX, 0x8(SP)

    CALL runtime.printlock(SB)     // 依次 printint(x), printint(y)。
    MOVQ 0x10(SP), BX
```

```
    MOVQ BX, 0(SP)
    CALL runtime.printint(SB)
    CALL runtime.printsp(SB)

    MOVQ 0x8(SP), BX
    MOVQ BX, 0(SP)
    CALL runtime.printint(SB)
    CALL runtime.printnl(SB)
    CALL runtime.printunlock(SB)
```

赋值操作，必须确保左右值类型相同。

未使用错误

编译器将未使用局部变量当做错误。不要觉得麻烦，这有助于培养良好的编码习惯。

```
var x int           // 全局变量没问题。

func main() {
    y := 10
}
```

输出：

```
$ go build

./test.go: y declared and not used
```

2. 命名

通常我们优先选用有实际含义，易于阅读和理解的字母、单词或组合对变量、常量、函数、自定义类型进行命名。

命名建议

- 以字母或下划线开始，由多个字母、数字和下划线组合而成。
- 区分大小写。

- 使用驼峰（camel case）拼写格式。
- 局部变量优先使用短命名。
- 不要使用保留关键字。
- 不建议使用与预定义常量、类型、内置函数相同命名。
- 专有名词通常会全部大写，例如 escapeHTML。

尽管 Go 支持用汉字等 Unicode 字符命名，但从编程习惯上来说，这并不是好选择。

```
func main() {
    var c int                           // c 代替 count。
    for i := 0; i < 10; i++ {           // i 代替 index。
        c++
    }

    println(c)
}
```

符号名字首字母大小写决定了其作用域。首字母大写为导出成员，可被包外部引用，而小些则仅能在包内使用。相关细节，可参考后续章节。

空标识符

和 Python 类似，Go 也有个名为“_”的特殊成员（blank identifier）。通常作为忽略占位符使用，可作表达式左值，无法读取其内容。

```
import "strconv"

func main() {
    x, _ := strconv.Atoi("12")    // 忽略 Atoi 的 err 返回值。
    println(x)
}
```

空标识符可用来临时规避编译器对未使用变量和导入包的错误检查。但请注意，它是预置成员，不能重新定义。

3. 常量

常量表示运行时恒定不可改变的值，通常是一些字面量。使用常量可用一个易于阅读理解的标识符号来代替“魔法数字”，也使得我们在调整常量值时，无需修改所有引用代码。

常量值必须是编译期可确定的字符、字符串、数字或布尔值。可指定常量类型，或由编译器通过初始化值推断，不支持 C/C++ 数字类型后缀。

```
const x, y int = 123, 0x22
const s = "hello, world!"
const c = '我'                      // rune (unicode code point)

const (
    i, f = 1, 0.123               // int, float64 (默认)
    b    = false
)
```

可在函数代码块中定义常量，不曾使用的常量不会引发编译错误。

```
func main() {
    const x = 123
    println(x)

    const y = 1.23          // 未使用，不会引发编译错误。

    {
        const x = "abc"     // 在不同作用域定义同名常量。
        println(x)
    }
}
```

输出：

```
123
abc
```

如果显式指定类型，必须确保常量左右值类型一致，需要时可做显式转换。右值不能超出常量类型取值范围，否则会引发溢出错误。

```
const (
    x, y int = 99, -999
    b        = byte(x)      // 如果 x 没有显式类型定义，那么无需转换，编译器将右值直展开为 99。
```

```
    n        = uint8(y)    // 错误: constant -999 overflows uint8
)
```

常量值也可以是某些编译器能直接计算结果的表达式，如 unsafe.Sizeof、len、cap 等。

```
import "unsafe"

const (
    ptrSize = unsafe.Sizeof(uintptr(0))
    strSize = len("hello, world!")
)
```

在常量组中如不指定类型和初始化值，则与上一行非空常量右值（或表达式文本）相同。

```
import "fmt"

func main() {
    const (
        x   uint16 = 120
        y                                   // 与上一行 x 类型、右值相同。
        s   = "abc"
        z                                   // 与 s 类型、右值相同。
    )

    fmt.Printf("%T, %v\n", y, y)          // 输出类型和值。
    fmt.Printf("%T, %v\n", z, z)
}
```

输出：

```
uint16, 120
string, abc
```

枚举

Go 并没有明确意义上的 enum 定义，不过可借助 iota 标识符实现一组自增常量值来实现枚举类型。

```
const (
    x   = iota      // 0
    y               // 1
    z               // 2
```

```
)

const (
    _   = iota                    // 0
    KB  = 1 << (10 * iota)        // 1 << (10 * 1)
    MB                            // 1 << (10 * 2)
    GB                            // 1 << (10 * 3)
)
```

自增作用范围为常量组。可在多常量定义中使用多个 iota，它们各自单独计数，只须确保组中每行常量列数量相同即可。

```
const (
    _, _ = iota, iota * 10        // 0, 0 * 10
    a, b                          // 1, 1 * 10
    c, d                          // 2, 2 * 10
)
```

如中断 iota 自增，则必须显式恢复。且后续自增值按行序递增，而非 enum 那般按上一取值递增。

```
const (
    a   = iota      // 0
    b               // 1
    c   = 100       // 100
    d               // 100（与上一行常量右值表达式相同）
    e   = iota      // 4（恢复 iota 自增，计数包括 c、d）
    f               // 5
)
```

自增默认数据类型为 int，不过可显式指定常量类型。

```
const (
    a           = iota     // int
    b   float32 = iota     // float32
    c           = iota     // int（如不显式指定 iota，则与 b 数据类型相同）
)
```

在实际编码中，建议通过自定义类型来实现用途明确的枚举类型。但须注意，这并不能将取值限定在预定义的枚举常量值范围内。

```
type color byte            // 自定义类型

const (
    black color = iota     // 指定常量类型
    red
    blue
)

func test(c color) {
    println(c)
}

func main() {
    test(red)
    test(100)       // 100 并未超出 color/byte 类型取值范围。

    x := 2
    test(x)         // 错误: cannot use x (type int) as type color in argument to test
}
```

展开

常量除“只读”外，和变量究竟有什么不同？

```
var   x = 0x100
const y = 0x200

func main() {
    println(&x, x)
    println(&y, y)         // 错误: cannot take the address of y
}
```

不同于变量在运行期分配存储内存（非优化状态），常量通常会被编译器在预处理阶段直接展开，作为指令数据直接使用。

```
const y = 0x200

func main() {
    println(y)
}
```

输出：

```
$ go build && go tool objdump -s "main\.main" test
```

```
TEXT main.main(SB) test.go
    MOVQ $0x200, 0(SP)                    // 将常量值作为指令数据展开。
    CALL runtime.printint(SB)
```

数字常量不会分配存储空间，无需像变量那样通过内存寻址来取值，因此无法获取地址。

> TODO: 鉴于 Go 当前对动态库的支持尚不完善，是否存在“常量陷阱”问题，还有待观察。

提到常量展开，我们还需回头看看常量的两种状态对编译器的影响。

```
const x = 100              // 无类型声明的常量。
const y byte = x           // 直接展开 x，相当于 const y byte = 100。

const a int = 100          // 显式指定常量类型，编译器会做强类型检查。
const b byte = a           // 错误: cannot use a (type int) as type byte in const initializer
```

4. 基本类型

清晰完备的预定义基础类型，使得开发跨平台应用时无需过多考虑数字符号和长度差异。

类型	长度	默认值	说明
bool	1	fasle	
byte	1	0	uint8
int, uint	4, 8	0	默认整数类型，依据目标平台，32 或 64 位。
int8, uint8	1	0	-128 ~ 127, 0 ~ 255。
int16, uint16	2	0	-32768 ~ 32767, 0 ~ 65535。
int32, uint32	4	0	-21 亿 ~ 21 亿, 0 ~ 42 亿。
int64, uint64	8	0	
float32	4	0.0	
float64	8	0.0	默认浮点数类型。
complex64	8		
complex128	16		

类型	长度	默认值	说明
rune	4	0	Unicode Code Point, int32
uintptr	4, 8	0	足以存储指针的 uint。
string		""	字符串，默认值为空字符串，而非 NULL。
array			数组
struct			结构体
function		nil	函数
interface		nil	接口
map		nil	字典，引用类型。
slice		nil	切片，引用类型。
channel		nil	通道，引用类型。

支持八进制、十六进制以及科学记数法。标准库 math 定义了各数字类型的取值范围。

```
import (
    "fmt"
    "math"
)

func main() {
    a, b, c := 100, 0144, 0x64
    println(a, b, c)

    fmt.Printf("0b%b, %#o, %#x\n", a, a, a)

    println(math.MinInt8, math.MaxInt8)
}
```

输出：

```
100 100 100
0b1100100, 0144, 0x64

-128 127
```

标准库 strconv 可在不同进制（字符串）间转换。

```
import "strconv"

func main() {
    a, _ := strconv.ParseInt("1100100", 2, 32)
    b, _ := strconv.ParseInt("0144", 8, 32)
```

```
    c, _ := strconv.ParseInt("64", 16, 32)

    println(a, b, c)

    println("0b" + strconv.FormatInt(a, 2))
    println("0" + strconv.FormatInt(a, 8))
    println("0x" + strconv.FormatInt(a, 16))
}
```

输出：

```
100 100 100
0b1100100
0144
0x64
```

使用浮点数时，需注意小数位有效精度，相关细节可参考 IEEE-754 标准。

```
func main() {
    var a float32 = 1.1234567899          // 注意：默认浮点类型是 float64。
    var b float32 = 1.12345678
    var c float32 = 1.123456781

    println(a, b, c)
    println(a == b, a == c)
    fmt.Printf("%v %v, %v\n", a, b, c)
}
```

输出：

```
+1.123457e+000 +1.123457e+000 +1.123457e+000
true true
1.1234568 1.1234568, 1.1234568
```

别名

在官方的语言规范中，专门提到两个别名。

```
byte        alias for uint8
rune        alias for int32
```

别名类型无需转换，可直接使用。

```
func test(x byte) {
    println(x)
}

func main() {
    var a byte = 0x11
    var b uint8 = a
    var c uint8 = a + b

    test(c)
}
```

但这并不表示，拥有相同底层结构的就属类型别名。就算在 64 位平台 int 和 int64 结构完全一致，但亦分属不同类型，须显式转换。

```
func add(x, y int) int {
    return x + y
}

func main() {
    var x int = 100
    var y int64 = x        // 错误: cannot use x (type int) as type int64 in assignment

    add(x, y)              // 错误: cannot use y (type int64) as type int in argument to add
}
```

5. 引用类型

所谓引用类型（reference type）特指 slice、map、channel 这三种预定义类型。

相比数字、数组等类型，引用类型拥有更复杂的存储结构。除分配内存外，它们还需初始化一系列属性，诸如指针、长度，甚至包括哈希分布、数据队列等。

内置函数 new 依指定类型数据长度分配内存，返回指针。而引用类型则必须使用 make 函数创建，编译器将 make 转换为目标类型专用创建函数（或指令），以完成内存分配和属性初始化。

```
func mkslice() []int {
    s := make([]int, 0, 10)
    s = append(s, 100)
```

```
    return s
}

func mkmap() map[string]int {
    m := make(map[string]int)
    m["a"] = 1
    return m
}

func main() {
    m := mkmap()
    println(m["a"])

    s := mkslice()
    println(s[0])
}
```

输出：

```
$ go build -gcflags "-l"          // 禁用函数内联。

$ go tool objdump -s "main\.mk" test

TEXT main.mkslice(SB) test.go
    CALL runtime.makeslice(SB)

TEXT main.mkmap(SB) test.go
    CALL runtime.makemap(SB)
```

除 new/make 函数外，也可使用初始化表达式，编译器生成的创建指令基本相同。

当然，new 函数也可为引用类型分配内存，但这是不完整创建。除基本内存外，关联内存和内部属性都未初始化，无法确保目标对象能正确工作。

```
import "fmt"

func main() {
    m := *new(map[string]int)     // 函数 new 返回指针。
    m["a"] = 1                    // panic: assignment to entry in nil map  (运行期错误)

    fmt.Println(m)
}
```

6. 类型转换

隐式转换造成的问题远大于它带来的好处。

除常量、别名类型以及未命名类型外，Go 强制要求使用显式类型转换。加上不支持操作符重载，所以我们总是能确定语句及表达式的明确含义。

```
a := 10
b := byte(a)
c := a + int(b)     // 混合类型表达式必须确保类型一致。
```

同样不能将非 bool 类型结果当做 true/false 使用。

```
func main() {
    x := 100

    var b bool = x         // 错误: cannot use x (type int) as type bool in assignment

    if x {                 // 错误: non-bool x (type int) used as if condition
    }
}
```

TODO: 转换规则。

语法歧义

如果转换的目标类型是指针、单向通道或没有返回值的函数类型，那么必须使用括号，以避免语法优先级分解错误。

```
func main() {
    x := 100
    p := *int(&x)          // 错误: cannot convert &x (type *int) to type int
                           //        invalid indirect of int(&x) (type int)
    println(p)
}
```

正确的做法是用括号，让编译器将 *int 解析为指针类型。

```
(*int)(p)                   --> 如果没有括号 -->     *(int(p))
(<-chan int)(c)                                     <-(chan int(c))
(func())(x)                                         func() x

func()int(x)                --> 有返回值的函数类型可省略括号，但依然建议使用。
(func()int)(x)
```

7. 自定义类型

使用关键字 type 定义用户自定义类型，包括基于现有类型创建，或结构体、函数类型等。

```
type flags byte

const (
    read flags = 1 << iota
    write
    exec
)

func main() {
    f := read | exec
    fmt.Printf("%b\n", f)          // 输出二进制标记位。
}
```

输出：

```
101
```

和 var、const 类似，多个 type 定义可合并成组，可在函数或代码块内定义局部类型。

```
func main() {
    type (
        user struct {              // 结构体
            name string
            age  uint8
        }

        event func(string) bool    // 函数类型
    )

    u := user{"Tom", 20}
```

```
    fmt.Println(u)

    var f event = func(s string) bool {
        println(s)
        return s != ""
    }

    f("abc")
}
```

但须注意，即便指定了基础类型，也只是表明它们拥有相同的数据结构，两者间不存在任何关系，属完全不同的两种类型。除操作符外，自定义类型不会继承基础类型的任何信息（包括方法），不能视作别名，不能隐式转换或直接用于比较表达式。

```
func main() {
    type data int
    var d data = 10

    var x int = d          // 错误: cannot use d (type data) as type int in assignment
    println(x)

    println(d == x)        // 错误: invalid operation: d == x (mismatched types data and int)
}
```

未命名类型

与有明确标识符的 bool、int、string 等类型相比，array、slice、map、channel 等类型与具体的元素类型或长度等属性有关，故称作未命名类型（unnamed type）。当然，可用 type 为其提供具体名称以变为命名类型（named type）。

具有相同声明的未命名类型被视作同一类型。

- 具有相同基类型的指针。
- 具有相同元素类型和长度的 array。
- 具有相同元素类型的 slice。
- 具有相同键值类型的 map。
- 具有相同数据类型及操作方向的 channel。
- 具有相同字段序列（字段名、字段类型、标签以及字段顺序）的 struct。
- 具有相同签名（参数和返回值列表，不包括参数名）的 function。
- 具有相同方法集（方法名、方法签名，不包括顺序）的 interface。

容易忽视的就是 struct tag，它是类型的组成部分，而非仅仅元数据注释那么简单。

```
func main() {
    var a struct {                // 匿名结构类型。
        x   int    `x`
        s   string `s`
    }

    var b struct {
        x   int
        s   string
    }

    b = a           // 错误: cannot use a type
                    //        struct { x int "x"; s string "s" } as type
                    //        struct { x int; s string } in assignment

    fmt.Println(b)
}
```

同样，函数的参数顺序也属签名组成部分。

```
func main() {
    var a func(int, string)
    var b func(string, int)

    b = a           // 错误: cannot use a (type func(int, string)) as type
                    //        func(string, int) in assignment
    b("s", 1)
}
```

未命名类型转换规则：

- 类型相同。
- 基础类型相同，且其中一个是未命名类型。
- 数据类型相同，将双向 channel 赋值给单向类型，且其中一个为未命名类型。
- 将 nil 赋值给 slice、map、channel、pointer、function 以及 interface。
- 实现了目标 interface。

```
func main() {
    type data [2]int
```

```
    var d data = [2]int{1, 2}      // 基础类型相同，右值为未命名类型。

    fmt.Println(d)

    a := make(chan int, 2)
    var b chan<- int = a           // chan 转换为 chan<-，其中 b 为未命名类型。

    b <- 2
}
```

三. 表达式

1. 保留字

仅 25 个保留关键字（keywords），虽不是主流语言中最少的，但也体现了 Go 语法规则的简洁性。保留关键字不能用作常量、变量、函数名以及结构字段等标识符。

```
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
```

> 相比在更新版本中不停添加新语言功能，我更喜欢简单的语言设计。某些功能可通过类库扩展，或其他非侵入方式实现，完全没必要为了“方便”让语言变得臃肿。过于丰富的功能特征会随着时间的推移抬升门槛，还会让代码变得日趋“魔幻”，降低一致性和可维护性。

2. 运算符

很久以前，流传“程序 = 算法 + 数据”这样的说法。

算法是什么？通俗点说就是“解决问题的过程”。小到加法指令，大到成千上万台服务器组成的分布式计算集群。抛去抽象概念和宏观架构，最终都由最基础的机器指令过程去处理不同层次存储设备里的数据。

学习语言和设计架构不同，我们所关心的就是微观层次，诸如语法规则所映射的机器指令，以及数据存储位置和格式等等。其中，运算符和表达式用来串联数据和指令，算是最基础的算法。

> 另有一句话：“硬件的方向是物理，软件的结局是数学”。

全部运算符及分隔符列表：

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

没有乘幂和绝对值运算符，对应的是标准库 math 里的 Pow、Abs 函数实现。

优先级

一元运算符优先级最高，二元则分成五个级别，从高往低分别是：

```
highest     *    /    %    <<   >>   &   &^
            +    -    |    ^
            ==   !=   <    <=   >    >=
            &&
lowest      ||
```

相同优先级的二元运算符，从左往右依次计算。

二元运算符

操作数类型必须相同，除非是无显式类型声明的常量，且该常量操作数会自动转换为另一操作数类型。当然，位移操作会有些不同。

```
func main() {
    const v = 20                            // 无显式类型声明的常量。

    var a byte = 10
    b := v + a                              // v 自动转换为 byte/uint8 类型。
    fmt.Printf("%T, %v\n", b, b)

    const c float32 = 1.2
    d := c + v                              // v 自动转换为 float32 类型。
    fmt.Printf("%T, %v\n", d, d)
}
```

输出：

```
uint8, 30
float32, 21.2
```

位移右操作数必须是无符号整数，或可以转换的无显式类型常量。

```
func main() {
    b := 23         // b 是有符号 int 类型。
    x := 1 << b     // 无效操作: 1 << b (shift count type int, must be unsigned integer)
    println(x)
}
```

如果是非常量位移表达式，那么会优先将无显式类型的常量左操作数转型。

```
func main() {
    var s uint = 3

    a := 1.0 << 3                       // 常量表达式（包括常量展开）。
    fmt.Printf("%T, %v\n", a, a)        // int, 8

    b := 1.0 << s                       // 无效操作: 1 << s (shift of type float64)
    fmt.Printf("%T, %v\n", b, b)        //   因为 b 没有提供类型，那么编译器通过 1.0 推断，
                                        //   显然无法对浮点数做位移操作。

    var c int32 = 1.0 << s              // 自动将 1.0 转换为 int32 类型。
    fmt.Printf("%T, %v\n", c, c)        // int32, 8
}
```

位运算符

二进制位运算符比较特别的就是“bit clear”，在其他语言里很少见到。

```
AND             按位与：都为 1                  a & b       0101 & 0011 = 0001
OR              按位或：至少一个 1              a | b       0101 | 0011 = 0111
XOR             按位亦或：只有一个 1            a ^ b       0101 ^ 0011 = 0110
NOT             按位取反（一元）                ^a          ^0111 = 1000
AND NOT         按位清除（bit clear）           a &^ b      0110 &^ 1011 = 0100
LEFT SHIFT      位左移                          a << 2      0001 << 3 = 1000
RIGHT SHIFT     位右移                          a >> 2      1010 >> 2 = 0010
```

位清除（AND NOT）和位亦或（XOR）是不同的。它的右操作数作用类似位图，将有标记的左操作数对应二进制位重置为 0，以达到一次清除多个标记位的目的。

```
const (
    read byte = 1 << iota
    write
    exec
    freeze
)

func main() {
    a := read | write | freeze
    b := read | freeze | exec
    c := a &^ b                    // 相当于 a ^ read ^ freeze，但不能包括 exec。

    fmt.Printf("%04b &^ %04b = %04b\n", a, b, c)
}
```

输出：

```
1011 &^ 1101 = 0010
```

自增运算符

自增、自减运算符只能作为独立语句，不能用于表达式。

```
func main() {
    a := 1
    ++a             // 语法错误：unexpected ++   （不能前置）

    if (a++)>1 {    // 语法错误：unexpected ++, expecting ) （语句不能作为表达式使用）
    }

    p := &a
    *p++            // 相当于 (*p)++
    println(a)
}
```

表达式通常是求值代码，可作为右值或参数使用。而语句完成一个行为，比如 if、for 代码块。表达式可作为语句用，但语句却不能当做表达式。

指针

不能将内存地址与指针混为一谈。

内存地址是内存中每个字节单元的唯一编号，而指针则是一个实体。它会分配内存空间，相当于一个专门用来保存地址的数字变量。

```
                  p := &x                  x := 100
-----------------+--------+------\\-------+------+---------
 memory  ...     | 0x1200 |     ....      | 100  |   ...
-----------------+--------+------\\-------+------+---------
 address         0x800                    0x1200
```

- 取址运算符 “&” 用于获取对象地址。
- 指针运算符 “*” 用于间接引用目标对象。
- 二级指针 **T，或包含包名 *package.T。

并非所有对象都能进行取地址操作，但变量总是能正确返回（addressable）。指针运算符为左值时，我们可更新目标对象状态，而为右值时则是为了获取目标状态。

```
func main() {
    x := 10
    var p *int = &x        // 获取地址，保存到指针变量。

    *p += 20               // 用指针间接引用，并更新对象。
    println(p, *p)         // 输出指针所存储地址，以及目标对象。
}
```

输出：

```
0xc82003df30 30
```

```
func main() {
    m := map[string]int{ "a": 1 }

    println(&m["a"])       // 错误: cannot take the address of m["a"]
}
```

指针类型支持相等运算符，但不能做加减法运算和直接类型转换。如果两个指针指向同一地址，或都为 nil，那么它们相等。

```
func main() {
    x := 10
    p := &x

    p++                     // 无效操作: p++ (non-numeric type *int)
    var p2 *int = p + 1     // 无效操作: p + 1 (mismatched types *int and int)

    p2 = &x
    println(p == p2)
}
```

可通过 unsafe.Pointer 将指针转换为 uintptr 后进行加减法运算，但这可能会造成非安全内存访问。实际上，unsafe.Pointer 类似 C void* 万能指针，可用来转换指针类型。详情参考后续章节。

没有专门指向成员的“->”运算符，统一使用“.”选择表达式。

```
func main() {
    a := struct {
        x int
    }{}

    a.x = 100

    p := &a
    p.x += 100      // 相当于 p->x += 100
    println(p.x)
}
```

零长度（zero-size）对象的地址是否相等和具体的实现版本有关，不过肯定不等于 nil。

```
func main() {
    var a, b struct{}

    println(&a, &b)
    println(&a == &b, &a == nil)
}
```

输出：

```
0xc820041f2f  0xc820041f2f
true  false
```

即便长度为 0，可该对象依然是“合法存在”的，拥有内存地址，这与 nil 语义完全不同。

在 runtime/malloc.go 里有个 zerobase 全局变量，所有通过 mallocgc 分配的零长度对象都使用该地址。不过上例中，对象 a、b 在栈上分配，并未调用 mallocgc 函数。

3. 初始化

对复合类型（array、slice、struct、map）变量初始化时，有一些语法限制。

- 初始化表达式必须有类型标签。
- 左大括号必须在类型尾部，不能另起一行。
- 多个成员初始值以逗号分隔。
- 允许多行，但每行必须以逗号或右大括号结束。

正确示例：

```
type data struct {
    x   int
    s   string
}

var a data = data{1, "abc"}

b := data{
    1,
    "abc",
}

c := []int{
    1,
    2 }

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

错误示例：

```
var d data = {1, "abc"}    // 语法错误: unexpected {  (缺类型标签)
```

```
d := data
    {               // 语法错误: unexpected semicolon or newline （左大括号不能另起一行）
        1,
        "abc"
    }

d := data{
    1,
    "abc"           // 语法错误: need trailing comma before newline （须以逗号或右大括号结束）
}
```

4. 流控制

精简（合并）了流控制语句，虽然某些时候不够便捷，但已足用。

if...else...

条件表达式值必须是 bool 类型，可省略括号，且左大括号不能另起一行。

```
func main() {
    x := 3

    if x > 5 {
        println("a")
    } else if x < 5 && x > 0 {
        println("b")
    } else {
        println("z")
    }
}
```

比较特别的是对初始化语句的支持，可定义块局部变量或执行初始化函数。

```
func main() {
    x := 10

    if pinit(); x == 0 {           // 优先执行 pinit 函数。
        println("a")
    }
```

```
    if a, b := x+1, x+10; a < b {          // 定义一个或多个局部变量（或函数返回值）。
        println(a)
    } else {
        println(b)
    }
}
```

局部变量有效范围包括全部 if/else 块。

对于编程初学者，可能会因条件匹配顺序写出死代码（dead code）。

```
func main() {
    x := 8

    if x > 5 {              // 优先判断，条件表达式结果为 true。
        println("a")
    } else if x > 7 {       // dead code
        println("b")
    }
}
```

输出：

```
a
```

死亡代码是指永远不会被执行的代码，通常用代码覆盖率（code coverage）测试进行检查。当然，一些比较智能的编译器也可主动清除死代码（dead code elimination, DCE）。吐槽一下：Go 官方编译器生成的汇编代码质量离 gcc 尚有很大差距。

尽可能减少代码嵌套，让正常逻辑处于相同层次。以常见的函数错误返回值处理为例。

```
import (
    "errors"
    "log"
)

func check(x int) error {
    if x <= 0 {
        return errors.New("x <= 0")
    }

    return nil
}
```

```
func main() {
    x := 10

    if err := check(x); err == nil {
        x++
        println(x)
    } else {
        log.Fatalln(err)
    }
}
```

很显然，if 块承担了两种逻辑：错误处理和后续正常操作。基于重构原则，我们应该保持代码块功能的单一性。

```
func main() {
    x := 10

    if err := check(x); err != nil {
        log.Fatalln(err)
    }

    x++
    println(x)
}
```

如此，if 块仅完成条件检查和错误处理，相关正常逻辑保持在同一层次。当有人试图通过阅读这段代码来获知逻辑流程时，完全可忽略 if 块细节。同时，单一功能可提升代码可维护性，更利于拆分重构。

当然，如需在多个条件块中使用局部变量，那么只能保留原层次，或直接使用外部变量。

```
import (
    "log"
    "strconv"
)

func main() {
    s := "9"

    n, err := strconv.ParseInt(s, 10, 64)        // 使用外部变量。

    if err != nil {
        log.Fatalln(err)
    } else if n < 0 || n > 10 {                  // 也可考虑拆分成另一个独立 if 块。
        log.Fatalln("invalid number")
    }
```

```
    println(n)                                          // 避免 if 局部变量将该逻辑放到 else 块。
}
```

对于某些过于复杂的组合条件，建议将其重构为函数。

```
import (
    "log"
    "strconv"
)

func main() {
    s := "9"

    if n, err := strconv.ParseInt(s, 10, 64); err != nil || n < 0 || n > 10 || n%2 != 0 {
        log.Fatalln("invalid number")
    }

    println("ok")
}
```

函数调用虽然有一些性能损失，可却让主流程变得更加清爽。况且，条件语句独立之后，更易于测试，同样会改善代码可维护性。

```
import (
    "errors"
    "log"
    "strconv"
)

func check(s string) error {
    n, err := strconv.ParseInt(s, 10, 64)
    if err != nil || n < 0 || n > 10 || n%2 != 0 {
        return errors.New("invalid number")
    }

    return nil
}

func main() {
    s := "9"

    if err := check(s); err != nil {
        log.Fatalln(err)
    }

    println("ok")
}
```

将流程和局部细节分离是很常见的做法，不同的变化因素被分隔在各自独立单元（函数或模块）内，可避免修改时造成关联错误，减少患肥胖症的函数数量。当然，代码单元测试也是主要原因之一。另一方面，该示例中的函数 check 仅被 if 块调用，也可将其作为局部函数，以避免扩大作用域，只是对测试的友好度会差一些。

当前编译器只能说够用，待优化的地方太多，其中内联处理做得也差强人意，所以代码维护性和性能平衡需要投入更多心力。

语言方面，最遗憾的是没有条件运算符 “a > b ? a : b”。有没有 lambda 无所谓，但没有这个却少了份优雅。加上一大堆 err != nil 判断语句，对于有完美主义的洁癖患者是个折磨。

---

switch

与 if 类似，switch 语句也用于选择执行，但具体使用场景会有所不同。

```
func main() {
    a, b, c, x := 1, 2, 3, 2

    switch x {                  // 将 x 与 case 条件匹配。
    case a, b:                  // 多个匹配条件命中其一即可（OR），变量。
        println("a | b")
    case c:                     // 单个匹配条件。
        println("c")
    case 4:                     // 常量
        println("d")
    default:
        println("z")
    }
}
```

输出：

```
a | b
```

条件表达式支持非常量值，这要比 C 更加灵活。相比 if 比较表达式，case 值列表要更加简洁。

抛开语法格式不说，编译器对 if、switch 生成的机器指令有可能完全相同，所谓谁性能更好需要看具体情况，不能作为主观条件。

同样支持初始化语句，按从左到右、从上到下顺序匹配执行。只有当全部 case 值匹配失败时，才会执行 default 块。

```
func main() {
    switch x := 5; x {
    default:               // 编译器确保不会先执行 default 块。
        x += 100
        println(x)
    case 5:
        x += 50
        println(x)
    }
}
```

输出：

```
55
```

考虑到 default 作用类似 else{}，建议将其放置在 switch 末尾。

相邻的空 case 不构成多条件匹配。

```
switch x {
case a:                    // 单条件，内容为空。隐式 “case a: break;”。
case b:
     println("b")
}
```

不能出现重复的 case 常量值。

```
func main() {
    switch x := 5; x {
    case 5:
        println("a")
    case 6, 5:             // 错误: duplicate case 5 in switch
        println("b")
    }
}
```

默认无须执行 break 语句，case 执行完毕后自动中断。如需要贯通后续 case（源码顺序），须使用 fallthrough，但不再匹配后续条件表达式。

```
func main() {
    switch x := 5; x {
    default:
        println(x)
    case 5:
        x += 10
        println(x)

        fallthrough        // 继续执行下一 case，但不再匹配条件表达式。
    case 6:
        x += 20
        println(x)

        //fallthrough      // 如果在此继续 fallthrough，不会执行 default，完全按源码顺序。
    }                      //   导致 “cannot fallthrough final case in switch” 错误。
}
```

输出：

```
15
35
```

注意，fallthrough 必须放在 case 块结尾。如要阻止，可使用 break 语句。

```
func main() {
    switch x := 5; x {
    case 5:
        x += 10
        println(x)

        if x >= 15 {
            break          // 终止，不再执行后续语句。
        }

        fallthrough        // 必须是 case 块的最后一条语句。
    case 6:
        x += 20
        println(x)
    }
}
```

输出：

```
15
```

某些时候，switch 还被用来替换 if 语句。被省略的 switch 条件表达式默认值为 true，继而与 case 比较表达式结果匹配。

```
func main() {
    switch x := 5; {              // 相当于 “switch x := 5; true { ... }”。
    case x > 5:
        println("a")
    case x > 0 && x <= 5:         // 多条件是 OR 关系，不能写成 “case x > 0, x <= 5”。
        println("b")
    default:
        println("z")
    }
}
```

输出：

```
b
```

---

switch 语句也可用于 interface 类型匹配，详见后续章节。

---

for

仅有 for 一种循环语句，常用方式都能支持。

```
for i := 0; i < 3; i++ {   // 初始化表达式支持函数调用或定义局部变量。
}

x := 3
for x < 10 {               // 相当于 “while x > 0 {}” 或 “for ; x < 10 ; {}”。
    x++
}

for {                      // 相当于 “while true {}” 或 “for true {}”。
    break
}
```

初始化语句仅被执行一次。条件表达式中如有函数调用，需确认是否会重复执行。可能会被编译器优化掉，也可能是动态结果需要每次执行确认。

```
func count() int {
    print("count.")
```

```
    return 3
}

func main() {
    for i, c := 0, count(); i < c; i++ {  // 初始化语句的 count 函数仅执行一次。
        println("a", i)
    }

    c := 0
    for c < count() {                     // 条件表达式中的 count 重复执行。
        println("b", c)
        c++
    }
}
```

输出：

```
count.a 0
a 1
a 2

count.b 0
count.b 1
count.b 2
```

规避方式就是在初始化表达式中定义局部变量保存 count 结果。

还可用 for...range 完成数据迭代，支持 string、array、*array、slice、map、channel 类型，返回索引、键值数据。

```
     data type        1st value         2nd value
-----------------+-----------------+-----------------+-----------------------------------
     string           index             s[index]           unicode, rune
     array/slice      index             v[index]
     map              key               value
     channel          element
```

```
func main() {
    data := [3]string{"a", "b", "c"}

    for i, s := range data {
        println(i, s)
    }
}
```

输出：

```
0 a
1 b
2 c
```

没有相关接口让我们实现自定义类型迭代，除非基础类型是上述类型之一。

允许返回单值，或用“_”忽略。

```
func main() {
    data := [3]string{"a", "b", "c"}

    for i := range data {          // 只返回 1st value。
        println(i, data[i])
    }

    for _, s := range data {       // 忽略 1st value。
        println(s)
    }

    for range data {               // 仅迭代，不返回。可用来执行清空 channel 等操作。
    }
}
```

无论普通 for 循环，还是 range 迭代，其定义的局部变量都会重复使用。

```
func main() {
    data := [3]string{"a", "b", "c"}

    for i, s := range data {
        println(&i, &s)
    }
}
```

输出：

```
0xc82003fe98 0xc82003fec8
0xc82003fe98 0xc82003fec8
0xc82003fe98 0xc82003fec8
```

这对闭包的使用有一些影响，相关详情，请参考后续章节。

注意，range 会复制目标数据。受直接影响的是 array，可改用 *array 或 slice 类型。

```
func main() {
    data := [3]int{10, 20, 30}

    for i, x := range data {              // 从 data 复制品中取值。
        if i == 0 {
            data[0] += 100
            data[1] += 200
            data[2] += 300
        }

        fmt.Printf("x: %d, data: %d\n", x, data[i])
    }

    for i, x := range data[:] {           // 仅复制 slice，不包括底层 array。
        if i == 0 {
            data[0] += 100
            data[1] += 200
            data[2] += 300
        }

        fmt.Printf("x: %d, data: %d\n", x, data[i])
    }
}
```

输出：

```
x: 10, data: 110
x: 20, data: 220           // range 返回的依旧是复制值。
x: 30, data: 330

x: 110, data: 210          // 当 i == 0 修改 data 时，x 已经取值，所以是 110。
x: 420, data: 420          // 复制的仅是 slice 自身，底层 array 依旧是原对象。
x: 630, data: 630
```

相关数据类型中，string、slice 基本结构是个很小的 struct，而 map、channel 是指针封装，复制成本都很小，不需专门优化。

如果 range 目标表达式是函数调用，也仅被执行一次。

```
func data() []int {
    println("origin data.")
    return []int{10, 20, 30}
}

func main() {
```

```
    for i, x := range data() {
        println(i, x)
    }
}
```

输出：

```
origin data.
0 10
1 20
2 30
```

嵌套循环建议不要超过 2 层，那会让代码变得很难维护，可重构为函数。

goto, continue, break

对于 goto 的讨伐由来已久，似乎是“笨蛋”标签一般。可事实上，你依旧能在许多场合见到它的身影，比如 go 源码里就有很多。

```
$ cd go/src
$ grep -r -n "goto" *
```

> 单就 1.6 的源码统计结果，就超出 1000 有余。很惊讶不是吗?

虽然某些设计模式可用来消除 goto 语句，但在一些性能优先场合，它依然能发挥积极作用。凡事都有两面，人云亦云并不是一个合格程序员的素质。

使用前，须先得定义标签。未使用的标签会引发编译错误，且区分大小写。

```
func main() {
start:                          // 错误: label start defined and not used
    for i := 0; i < 3; i++ {
        println(i)
        if i > 1 {
            goto exit
        }
    }

exit:
    println("exit.")
}
```

不能跳转到其他函数，或内层代码块。

```
func test() {
test:
    println("test")
    println("test exit.")
}

func main() {
    for i := 0; i < 3; i++ {
    loop:
        println(i)
    }

    goto test       // 错误: label test not defined
    goto loop       // 错误: goto loop jumps into block
}
```

和 goto 定点跳转不同，break、continue 则用于中断代码块执行。

- **break**：用于 switch、for、select 语句，终止整个语句块执行。
- **continue**：仅用于 for 循环，终止后续逻辑，立即进入下一轮循环。

```
func main() {
    for i := 0; i < 10; i++ {
        if i%2 == 0 {
            continue              // 立即进入下一轮循环。
        }

        if i > 5 {
            break                 // 立即终止整个 for 循环。
        }

        println(i)
    }
}
```

输出：

```
1
3
5
```

配合标签，break 和 continue 可指定多层嵌套循环的目标层级。

```
func main() {
outer:
    for x := 0; x < 5; x++ {
        for y := 0; y < 10; y++ {
            if y > 2 {
                println()
                continue outer
            }

            if x > 2 {
                break outer
            }

            print(x, ":", y, " ")
        }
    }
}
```

输出：

```
0:0 0:1 0:2
1:0 1:1 1:2
2:0 2:1 2:2
```

四. 函数

1. 定义

函数是结构化编程的最小模块单元。它将复杂的算法过程分解为若干较小任务，隐藏相关细节，使得程序结构更加清晰，易于维护。函数被设计成相对独立，通过接收输入参数完成一段算法指令，输出或存储相关结果。因此，函数还是代码复用和测试的基本单元。

关键字 func 用于定义函数。有些不太方便的限制，但也借鉴了动态语言的某些优点。

- 无需前置声明。
- 不支持命名嵌套定义（nested）。
- 不支持同名函数重载（overload）。
- 不支持默认参数。
- 支持不定长变参。
- 支持多返回值。
- 支持命名返回值。
- 支持匿名函数和闭包。

和前面曾说过的一样，左大括号依旧不能另起一行。

```
func test()
{                           // 错误: syntax error: unexpected semicolon or newline before {
}

func test(x int) {          // 错误: test redeclared in this block
}

func main() {
    func add(x, y int) int {      // 错误: syntax error: unexpected add, expecting (
        return x + y
    }
}
```

函数属第一类对象，具备相同签名（参数及返回值列表）的视作同一类型。

```
func hello() {
    println("hello, world!")
```

```
}

func exec(f func()) {
    f()
}

func main() {
    f := hello
    exec(f)
}
```

第一类对象（first-class object）指可在运行期创建，可用作函数参数或返回值，可存入变量的实体。最常见的用法就是匿名函数。

基于阅读和代码维护的角度，使用命名类型更加方便。

```
type FormatFunc func(string, ...interface{}) (string, error)

// 如果不使用命名类型，这个参数签名长到没法看。
func format(f FormatFunc, s string, a ...interface{}) (string, error) {
    return f(s, a...)
}
```

函数只能判断是否为 nil，不支持其他比较操作。

```
func a() {}
func b() {}

func main() {
    println(a == nil)
    println(a == b)        // 无效操作: a == b (func can only be compared to nil)
}
```

从函数返回局部变量指针是安全的，编译器会通过逃逸分析（escape analysis）来决定是否在堆上分配内存。

```
func test() *int {
    a := 0x100
    return &a
}

func main() {
    var a *int = test()
```

```
    println(a, *a)
}
```

输出：

```
$ go build -gcflags "-l -m"                         // 禁用函数内联，输出优化策略。

moved to heap: a
&a escapes to heap

$ go tool objdump -s "main\.main" test              // 反汇编确认。

TEXT main.main(SB) test.go
    CALL main.test(SB)

$ ./test

0xc820074000 256
```

函数内联对内存分配有一定的影响。如上例中不禁用内联，那么就直接在栈上分配内存。

```
$ go build -gcflags "-m"                            // 默认优化方式，允许内联。

inlining call to test
main &a does not escape

$ go tool objdump -s "main\.main" test

TEXT main.main(SB) test.go
    MOVQ $0x100, 0x10(SP)
    LEAQ 0x10(SP), BX
    MOVQ BX, 0x18(SP)
    MOVQ 0x18(SP), BX
    MOVQ BX, 0(SP)
    CALL runtime.printpointer(SB)
```

有关逃逸分析相关内容，请参考后续章节。

当前编译器并未实现尾递归优化（tail call optimization）。尽管 Go 执行栈上限是 GB 规模，轻易不会出现堆栈溢出（stack overflow）错误，但依然需要注意拷贝栈复制成本。

内存管理相关内容，请阅读本书下册《源码剖析》。

建议命名规则

在避免冲突的情况下，函数命名要本着精简短小、望文知意的原则。

- 通常是动词和介词加上名词，例如 scanWords。
- 避免不必要的缩写，printError 要比 printErr 更好一些。
- 避免使用类型关键字，比如 buildUserStruct 看上去很别扭。
- 避免歧义，不能有多种用途解释造成误解。
- 避免仅通过大小写区分的同名函数。
- 避免与内置函数同名，以导致误用。
- 避免使用数字，除非特定专有名词，例如 UTF8。
- 避免添加作用域提示前缀。
- 统一使用 camel/pascal case 拼写风格。
- 使用相同术语，保持一致性。
- 使用习惯用语，比如 init 表示初始化，is/has 返回布尔值结果。
- 使用反义词组命名行为相反的函数，比如 get/set、min/max 等。

函数和方法的命名规则稍有些不同。方法通过选择符调用，且具备状态上下文，可使用更简短的动词命名。

---

2. 参数

对参数的处理偏向保守和传统，不支持有默认值的可选参数，不支持命名实参。调用时，必须按签名顺序传递指定类型和数量的实参，就算以“_”命名也不能忽略。

参数列表中，相邻的同类型参数可合并。

```
func test(x, y int, s string, _ bool) *int {
    return nil
}

func main() {
    test(1, 2, "abc")      // 错误: not enough arguments in call to test
}
```

参数可视作函数局部变量，因此不能在相同层次定义同名变量。

```
func add(x, y int) int {
    x := 100                    // 错误: no new variables on left side of :=
    var y int                   // 错误: y redeclared in this block

    return x + y
}
```

形参是指函数定义中的参数，实参则是函数调用时所传递的参数。形参类同函数局部变量，而实参则函数外部对象，可以是常量、变量、表达式或函数等。

不管是指针、引用类型，还是其他类型参数，都是值拷贝传递（pass by value）。无非是拷贝目标对象，还是拷贝指针自身而已。在函数调用时，会为形参和返回值分配内存空间，并将实参数据拷贝到形参内存。

```
func test(x *int) {
    println(&x, x)          // 输出形参 x 的地址。
}

func main() {
    a := 0x100
    p := &a
    println(&p, p)          // 输出实参 p 的地址。

    test(p)
}
```

输出：

```
0xc82003ff38  0xc82003ff28
0xc82003ff30  0xc82003ff28
```

从输出结果可以看出，尽管实参和形参都指向同一目标，但指针自身依然被复制。

表面上看，指针参数的性能要更好一些，但实情得具体分析。被复制的指针会延长目标对象生命周期，这可能会导致它被分配到堆上，那么性能消耗就得加上堆内存分配和垃圾回收的成本。其实在栈上复制小对象只需很少的指令就可完成，远比运行时完成堆内存分配要快得多。另外，并发编程也提倡尽可能使用不可变对象（复制品），这可消除数据同步等麻烦。当然，如果复制成本很高，或需要修改原对象状态，自然是用指针为佳。

下面是一个指针参数导致实参变量被分配到堆上的简单示例。可对比传值参数的汇编代码，从中可看出具体而差别。

```
func test(p *int) {
    go func() {                          // 延长 p 生命周期。
        println(p)
    }()
}

func main() {
    x := 100
    p := &x

    test(p)
}
```

输出：

```
$ go build -gcflags "-m"                 // 输出编译器优化策略。

moved to heap: x
&x escapes to heap

$ go tool objdump -s "main\.main" test

TEXT main.main(SB) test.go
    CALL runtime.newobject(SB)           // 在堆上为 x 分配内存。
    CALL main.test(SB)
```

要实现传出参数，通常建议使用返回值。当然，二级指针也是常用的手法。

```
func test(p **int) {
    x := 100
    *p = &x
}

func main() {
    var p *int

    test(&p)
    println(*p)
}
```

输出：

```
100
```

如果函数参数过多，建议将其重构为一个复合类型参数，也算是变向实现可选参数和命名实参功能。

```
type serverOption struct {
    address string
    port    int
    path    string
    timeout time.Duration
    log     *log.Logger
}

func newOption() *serverOption {
    return &serverOption{
        address: "0.0.0.0",
        port:    8080,
        path:    "/var/test",
        timeout: time.Second * 5,
        log:     nil,
    }
}

func server(option *serverOption) {}

func main() {
    opt := newOption()
    opt.port = 8085

    server(opt)
}
```

将过多的参数独立成 option struct，既便于扩展参数集，也方便了通过 newOption 函数设置默认配置。这也是代码复用的一种方式，避免多处调用时设置繁琐的配置参数。

变参

变参本质上就是一个 slice，它只能接收相同类型的参数值，且必须放在参数列表的最后。

```
func test(s string, a ...int) {
    fmt.Printf("%T, %v\n", a, a)          // 显示类型和值。
}

func main() {
    test("abc", 1, 2, 3, 4)
}
```

输出：

```
[]int, [1 2 3 4]
```

使用数组对象作为变参时，须转换为 slice 后才能展开。

```
func test(a ...int) {
    fmt.Println(a)
}

func main() {
    a := [3]int{10, 20, 30}
    test(a[:]...)                 // 转换为 slice 后展开。
}
```

既然变参是 slice，那么参数复制的仅是 slice struct 自身，不包括底层数组，因此我们可以修改原数据。如果需要，可使用内置函数 copy 复制 slice 数据。

```
func test(a ...int) {
    for i := range a {
        a[i] += 100
    }
}

func main() {
    a := []int{10, 20, 30}
    test(a...)

    fmt.Println(a)
}
```

输出：

```
[110 120 130]
```

3. 返回值

有返回值的函数，必须有明确的 return 终止语句。

```
func test(x int) int {
```

```
    if x > 0 {
        return 1
    } else if x < 0 {
        return -1
    }
}                           // 错误: missing return at end of function
```

特例情况是 panic，或者无 break 的死循环。

```
func test(x int) int {
    for {
        break
    }
}                           // 错误: missing return at end of function
```

借鉴自动态语言的多返回值模式，让函数得以返回更多状态，尤其是 error 模式。

```
import "errors"

func div(x, y int) (int, error) {         // 多返回值列表必须使用括号。
    if y == 0 {
        return 0, errors.New("division by zero")
    }

    return x / y, nil
}
```

稍微不方便的是没有 tuple 类型，也无法用 array、slice 接收多个返回结果，但可用“_”忽略。相同类型的多返回值可用作调用实参，或直接返回。

```
func div(x, y int) (int, error) {
    if y == 0 {
        return 0, errors.New("division by zero")
    }

    return x / y, nil
}

func log(x int, err error) {
    fmt.Println(x, err)
}

func test() (int, error) {
    return div(5, 0)              // 多返回值直接用作 return 结果。
}
```

```
func main() {
    log(test())                    // 多返回值直接用作实参。
}
```

命名返回值

对返回值命名和简短变量定义一样，优缺点共存。

```
func paging(sql string, index int) (count int, pages int, err error) {
}
```

从这个简单的示例就可看出，命名返回值让函数声明更加清晰，同时也会改善帮助文档和代码编辑器提示。

命名返回值和参数一样，可当做函数局部变量使用，最后由 return 隐式返回。

```
func div(x, y int) (z int, err error) {
    if y == 0 {
        err = errors.New("division by zero")
        return
    }

    z = x / y
    return                 // 相当于 “return z, err”。
}
```

只是这些特殊的“局部变量”会被不同层级的同名变量遮蔽，造成潜在错误。好在编译器会自动检查此类状况，只需改为显式 return 返回即可。

```
func add(x, y int) (z int) {
    {
        z := x + y         // 新定义的同名局部变量，同名遮蔽。
        return             // 错误: z is shadowed during return （改成 “return z” 即可）
    }

    return
}
```

除遮蔽外，我们还必须对全部返回值命名，否则编译器会稀里糊涂，不知所以。

```
func test() (int, s string, e error) {
    return 0, "", nil      // 错误: cannot use 0 (type int) as type string in return argument
}
```

显然编译器在处理 return 语句的时候，会跳过未命名返回值，无法准确匹配。

如果返回值类型能明确表明其涵义，就尽量不要对其命名。

```
func NewUser() (*User, error)
```

4. 匿名函数

匿名函数是指没有定义名字符号的函数。

匿名函数除没有名字外，其他和普通函数完全相同。最大的区别是，我们可在函数内部定义匿名函数，形成类似嵌套函数的效果。匿名函数可直接调用，保存到变量，作为参数或返回值。

直接执行：

```
func main() {
    func(s string) {
        println(s)
    }("hello, world!")
}
```

变量赋值：

```
func main() {
    add := func(x, y int) int {
        return x + y
    }
```

```
    println(add(1, 2))
}
```

参数传递：

```
func test(f func()) {
    f()
}

func main() {
    test(func() {
        println("hello, world!")
    })
}
```

返回值：

```
func test() func(int, int) int {
    return func(x, y int) int {
        return x + y
    }
}

func main() {
    add := test()
    println(add(1, 2))
}
```

将匿名函数赋值给变量，与为普通函数提供名字标识符有着根本的区别。当然，编译器会为匿名函数生成一个“随机”符号名。函数符号名最终和 text 段内的一个地址相对应。

作为第一类对象，不管是普通函数，还是匿名函数都可作为 struct 字段，或通过 channel 传递。

```
func testStruct() {
    type calc struct {
        mul func(x, y int) int
    }

    x := calc{
        mul: func(x, y int) int {
            return x * y
        },
```

```
    }

    println(x.mul(2, 3))
}

func testChannel() {
    c := make(chan func(int, int) int, 2)

    c <- func(x, y int) int {
        return x + y
    }

    println((<-c)(1, 2))
}
```

不曾使用的匿名函数会被编译器当作错误。

```
func main() {
    func(s string) {       // 错误: func literal evaluated but not used
        println(s)
    }
}
```

除闭包外，匿名函数也是一种重要的重构手段。我们可以将一个庞大的函数分解成多个相对独立的匿名函数块，然后用相对简洁的调用完成逻辑流程，以实现骨干和细节分离。

匿名函数作用域受限，不会引发外部污染，但比普通代码块更加灵活。可与外部环境完全隔离（不使用闭包），没有代码顺序要求，便于实现优美清晰的代码层次。

闭包

闭包（closure）是在其词法上下文中引用了自由变量的函数，或者说是函数和其引用环境的组合体。这太学术范的说明很难理解，我们先看一个例子。

```
func test(x int) func() {
    return func() {
        println(x)
    }
}

func main() {
    f := test(123)
    f()
```

```
}
```

输出：

```
123
```

就这段代码而言，test 返回的匿名函数会引用上下文环境变量 x。当该函数在 main 中执行时，它依然可以正确读取 x 的值，这种现象就称作闭包。

闭包是如何实现的？匿名函数被返回后，如何继续读取环境变量？修改一下代码再看。

```
package main

func test(x int) func() {
    println(&x)

    return func() {
        println(&x, x)
    }
}

func main() {
    f := test(0x100)
    f()
}
```

输出：

```
0xc82000a100
0xc82000a100 256
```

通过输出指针，我们注意到闭包直接引用了原环境变量。分析汇编代码，你会看到返回的不仅仅是匿名函数，还包括所引用的环境变量指针。所以说，闭包是函数和引用环境组合体更加确切。

本质上返回的是一个 funcval 结构，可在 runtime/runtime2.go 中找到相关定义。

```
$ go build -gcflags "-N -l"        # 禁用内联和代码优化。

$ gdb test

(gdb) b 6
(gdb) b 13
(gdb) r
```

```
(gdb) info locals          # 进入 main.test，获取环境变量 x 地址。
&x = 0xc82000a130

(gdb) c                    # 继续执行，进入 main.main。

(gdb) disas
Dump of assembler code for function main.main:
   0x000000000000213f <+15>:    sub    rsp,0x18
   0x0000000000002143 <+19>:    mov    QWORD PTR [rsp],0x100
   0x000000000000214b <+27>:    call   0x2040 <main.test>
   0x0000000000002150 <+32>:    mov    rbx,QWORD PTR [rsp+0x8]      # test 返回值。
   0x0000000000002155 <+37>:    mov    QWORD PTR [rsp+0x10],rbx
=> 0x000000000000215a <+42>:    mov    rbx,QWORD PTR [rsp+0x10]
   0x000000000000215f <+47>:    mov    rdx,rbx                      # 保存到 rdx 寄存器。
   0x0000000000002162 <+50>:    mov    rbx,QWORD PTR [rdx]
   0x0000000000002165 <+53>:    call   rbx
   0x0000000000002167 <+55>:    add    rsp,0x18
   0x000000000000216b <+59>:    ret

(gdb) x/2xg $rbx           # 显然，funcval 包含匿名函数和而环境变量的地址。
0xc82000a140:   0x0000000000002180  0x000000c82000a130

(gdb) info symbol 0x0000000000002180
main.test.func1 in section .text

(gdb) s                    # 继续，进入匿名函数。
Breakpoint 1, main.test.func1

(gdb) disas
Dump of assembler code for function main.test.func1:
   0x000000000000218f <+15>:    sub    rsp,0x18
   0x0000000000002193 <+19>:    mov    rbx,QWORD PTR [rdx+0x8]      # 经 rdx 读取环境变量地址。
   0x0000000000002197 <+23>:    mov    QWORD PTR [rsp+0x10],rbx
   0x000000000000219c <+28>:    mov    rbx,QWORD PTR [rsp+0x10]
   0x00000000000021a1 <+33>:    mov    QWORD PTR [rsp+0x8],rbx
   0x00000000000021a6 <+38>:    call   0x23ac0 <runtime.printlock>
   0x00000000000021ab <+43>:    mov    rbx,QWORD PTR [rsp+0x8]
   0x00000000000021b0 <+48>:    mov    QWORD PTR [rsp],rbx
   0x00000000000021b4 <+52>:    call   0x244a0 <runtime.printpointer>

(gdb) x/1xg $rdx+0x8
0xc82000a148:   0x000000c82000a130
```

正因为闭包通过指针引用环境变量，那么可能会导致其生命周期延长，被分配到堆内存。另一方面，也有所谓“延迟求值”的特性。

```
func test() []func() {
    var s []func()

    for i := 0; i < 2; i++ {
```

```
        s = append(s, func() {            // 将多个匿名函数添加到列表。
            println(&i, i)
        })
    }

    return s                              // 返回匿名函数列表。
}

func main() {
    for _, f := range test() {            // 迭代执行所有匿名函数。
        f()
    }
}
```

输出：

```
0xc82000a078 2
0xc82000a078 2
```

对这个输出结果不必惊讶。道理很简单，for 循环复用局部变量 i，那么每次添加的匿名函数引用的自然是同一变量。而且添加操作也仅仅是将匿名函数放入列表，并未执行。因此，当 main 执行这些函数时，它们读取的是环境变量 i 循环结束后的最终值，不是 2，还能是什么？

解决方法就是每次用不同的环境变量，让各自闭包环境俱不相同。

```
func test() []func() {
    var s []func()

    for i := 0; i < 2; i++ {
        x := i                            // x 每次循环都重新定义分配。
        s = append(s, func() {
            println(&x, x)
        })
    }

    return s
}
```

输出：

```
0xc82006e000 0
0xc82006e008 1
```

多个匿名函数引用同一环境变量，也会让事情变得更加复杂。任何的修改行为都会影响其他函数取值，在并发模式下可能需要做同步处理。

```
func test(x int) (func(), func()) {       // 返回两个匿名函数。
    return func() {
            println(x)
            x += 10                       // 修改环境变量。
        }, func() {
            println(x)                    // 显示环境变量。
        }
}

func main() {
    a, b := test(100)
    a()
    b()
}
```

输出：

```
100
110
```

闭包让我们不用传递参数就可读取或修改环境状态，当然也需要为此付出额外代价。对于性能要求较高的场合，须慎重使用。

5. 延迟调用

语句 defer 用于向当前函数注册稍后执行的函数调用。这些调用被称作延迟调用，它们直到当前函数执行结束前才被执行，常用于资源释放、错误处理等操作。

```
func main() {
    f, err := os.Open("./main.go")
    if err != nil {
        log.Fatalln(err)
    }

    defer f.Close()        // 仅注册，直到 main 退出前才执行。

    ... do something ...
}
```

需要注意，延迟调用注册的是调用，而非函数，须提供执行所需参数。参数值在注册时被复制并缓存起来，如对状态敏感，可改用指针或闭包（可修改命名返回值）。

```
func main() {
    x, y := 1, 2

    defer func(a int) {
        println("defer x, y = ", a, y)          // y 为闭包引用。
    }(x)                                        // 注册时复制调用参数。

    x += 100                                    // 对 x 的修改，不会影响延迟调用。
    y += 200
    println(x, y)
}
```

输出：

```
101 202
defer x, y = 1 202
```

延迟函数返回值会被抛弃。

多个延迟注册按 FILO 次序执行。

```
func main() {
    defer println("a")
    defer println("b")
}
```

输出：

```
b
a
```

编译器通过在 ret 指令前插入指令来实现延迟调用执行，而 return 和 panic 语句都会终止当前函数流程，引发延迟调用。return 不是 ret，它会更新返回值。

```
func test() (z int) {
    defer func() {
        println("defer:", z)
        z += 100                    // 修改命名返回值。
    }()

    return 100                      // 实际执行次序：z = 100, call defer, ret
}
```

```
func main() {
    println("test:", test())
}
```

输出：

```
defer: 100
test: 200
```

有关 defer 更详细的分析，请阅读下册《源码剖析》。

误用

千万记住，延迟调用是在函数结束时才被执行。不合理的使用方式会浪费更多的资源，甚至造成逻辑错误。

案例：循环处理多个日志文件，defer 导致文件关闭时机延长。

```
func main() {
    for i := 0; i < 10000; i++ {
        path := fmt.Sprintf("./log/%d.txt", i)

        f, err := os.Open(path)
        if err != nil {
            log.Println(err)
            continue
        }

        // 这个关闭操作在 main 函数结束时才会执行，而不是当前循环。
        // 这无端延长了逻辑结束时间和 f 生命周期，平白多消耗了内存等资源。
        defer f.Close()

        ... do something ...
    }
}
```

应该直接调用，或重构为函数，让循环和处理算法分离。

```
func main() {
    // 日志处理算法。
    do := func(n int) {
        path := fmt.Sprintf("./log/%d.txt", n)
```

```
        f, err := os.Open(path)
        if err != nil {
            log.Println(err)
            continue
        }

        // 该延迟调用在此匿名函数结束时执行，而非 main。
        defer f.Close()

        ... do something ...
    }

    for i := 0; i < 10000; i++ {
        do(i)
    }
}
```

性能

相比直接用 CALL 指令调用函数，延迟调用则需花费更大代价。这其中包括注册、调用等操作，还有额外的缓存开销。

以最常用的 mutex 为例，我们简单测试一下两者的性能差异。

```
var m sync.Mutex

func call() {
    m.Lock()
    m.Unlock()
}

func deferCall() {
    m.Lock()
    defer m.Unlock()
}

func BenchmarkCall(b *testing.B) {
    for i := 0; i < b.N; i++ {
        call()
    }
}

func BenchmarkDefer(b *testing.B) {
    for i := 0; i < b.N; i++ {
        deferCall()
    }
}
```

输出：

```
BenchmarkCall-4     100000000           22.4 ns/op
BenchmarkDefer-4    20000000            93.8 ns/op
```

相差几倍的结果足以引起重视。尤其是那些性能要求高且压力大的算法，应避免使用。

6. 错误处理

返古的错误处理方式，是 Go 被谈及最多的内容之一。有人戏称做“Stuck in 70's”，可见它与流行趋势背道而驰。

error

官方推荐的标准做法是返回 error 状态。

```
func Scanln(a ...interface{}) (n int, err error)
```

标准库将 error 定义为接口类型，以便于实现自定义错误类型。

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

按惯例，error 总是最后一个返回参数。标准库提供了相关创建函数，可非常方便地创建包含简单错误文本的 error 对象。

```
var errDivByZero = errors.New("division by zero")

func div(x, y int) (int, error) {
    if y == 0 {
        return 0, errDivByZero
    }

    return x / y, nil
}
```

```
func main() {
    z, err := div(5, 0)
    if err == errDivByZero {
        log.Fatalln(err)
    }

    println(z)
}
```

应通过变量实例，而非文本内容来判定错误类别。

错误变量通常以 err 作为前缀，且字符串内容全部小写，没有结束标点，以便于嵌入到其他格式化字符串中输出。

与 errors.New 类似的还有 fmt.Errorf，它返回一个格式化内容的错误对象。

某些时候，我们需要自定义错误类型，以便容纳更多上下文状态信息。如此，还可基于类型做出判断。

```
type DivError struct                    // 自定义错误类型。
    x, y int
}

func (DivError) Error() string {        // 实现 error 接口方法。
    return "division by zero"
}

func div(x, y int) (int, error) {
    if y == 0 {
        return 0, DivError{x, y}        // 返回自定义错误类型。
    }

    return x / y, nil
}

func main() {
    z, err := div(5, 0)

    if err != nil {
        switch e := err.(type) {        // 根据类型匹配。
        case DivError:
            fmt.Println(e, e.x, e.y)
        default:
            fmt.Println(e)
        }
```

```
        log.Fatalln(err)
    }

    println(z)
}
```

> 自定义错误类型通常以 Error 为后缀。在用 switch 按类型匹配时，注意 case 顺序。应将自定义类型放在前面，优先匹配具体错误类型。

在正式代码中，我们不能忽略 error 返回值，应做严格检查，否则可能会导致错误的逻辑状态。调用多返回值函数时，除 error 外，其他返回值同样需要关注。

> 以 os.File.Read 方法为例，它会同时返回剩余内容和 EOF。
> 可用 Linter 检查代码里的 unchecked error。

大量函数和方法返回 error，这使得调用代码变得很难看，一堆堆的检查语句充斥在代码行间。解决思路有：

- 使用专门的 checkError 函数处理错误逻辑（比如记录日志），以此简化检查代码。
- 在不影响逻辑的情况下，使用 defer 延后处理错误状态（err 退化赋值）。
- 在不中断逻辑的情况下，将错误作为内部状态保存，最终“提交”时再处理。

panic, recover

与 error 相比，panic/recover 在使用方法上更接近 try/catch 结构化异常。

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

比较有趣的是，它们都是内置函数。panic 会立即中断当前函数流程，触发执行延迟调用。而在延迟调用函数中，recover 可捕获并返回 panic 提交的错误对象。

```
func main() {
    defer func() {
        if err := recover(); err != nil {         // 捕获错误。
            log.Fatalln(err)
        }
```

```
    }()

    panic("i am dead")                          // 引发错误。
    println("exit.")                            // 永不会执行。
}
```

因为 panic 参数是 interface{} 类型，因此可使用任何类型对象作为错误状态。而 recover 返回结果同样要做转型才能获取具体信息。

无论是否执行 recover，所有延迟调用都会被执行。但中断性错误会沿调用堆栈向外传递，要么被外层捕获，要么导致进程崩溃。

```
func test() {
    defer println("test.1")
    defer println("test.2")

    panic("i am dead")
}

func main() {
    defer func() {
        log.Println(recover())
    }()

    test()
}
```

输出：

```
test.2
test.1
i am dead
```

连续调用 panic，仅最后一个会被 recover 捕获。

```
func main() {
    defer func() {
        for {
            if err := recover(); err != nil {
                log.Println(err)
            } else {
                log.Fatalln("fatal")
            }
        }
    }()

    defer func() {
```

```
        panic("you are dead")      // 类似重新抛出异常的意思。
    }()

    panic("i am dead")
}
```

输出：

```
you are dead
fatal
```

在延迟函数中 panic，不会影响后续延迟调用执行。而 recover 之后 panic，可被再次捕获。另外，recover 函数必须在延迟调用函数中执行才能正常工作。

```
func catch() {
    log.Println("catch:", recover())
}

func main() {
    defer catch()                          // 捕获。
    defer log.Println(recover())           // 失败!
    defer recover()                        // 失败!

    panic("i am dead")
}
```

输出：

```
<nil>
catch: i am dead
```

考虑到 recover 特性，如果要保护代码片段，那么只能将其重构为函数调用。

```
func test(x, y int) {
    z := 0

    func() {                               // 利用匿名函数保护 “z = x / y”。
        defer func() {
            if recover() != nil {
                z = 0
            }
        }()

        z = x / y
    }()

    println("x / y =", z)
```

```
}

func main() {
    test(5, 0)
}
```

调试阶段，可使用 runtime/debug.PrintStack 函数输出完整调用堆栈信息。

```
import (
    "runtime/debug"
)

func test() {
    panic("i am dead")
}

func main() {
    defer func() {
        if err := recover(); err != nil {
            debug.PrintStack()
        }
    }()

    test()
}
```

输出：

```
goroutine 1 [running]:
main.main.func1()
    test.go:15 +0x6c
panic(0x7e3a0, 0xc82000a260)
    runtime/panic.go:426 +0x4e9
main.test()
    test.go:8 +0x65
main.main()
    test.go:20 +0x35
```

建议：除非是不可恢复性，导致系统无法正常工作的错误，否则不建议使用 panic。

> 例如：文件系统没有操作权限，服务端口被占用，数据库未启动等情况。

五. 数据

1. 字符串

字符串是不可变字节序列，其本身是一个复合结构。

```
type stringStruct struct {
    str unsafe.Pointer
    len int
}
```

头部指针指向字节数组，但没有 NULL 结尾。默认以 UTF-8 编码存储 Unicode 字符，字面量里允许使用十六进制、八进制和 UTF 编码格式。

```
func main() {
    s := "雨痕\x61\142\u0041"

    fmt.Printf("%s\n", s)
    fmt.Printf("% x, len: %d\n", s, len(s))
}
```

输出：

```
雨痕abA
e9 9b a8 e7 97 95 61 62 41, len: 9
```

内置函数 len 返回字节数组长度，cap 不接受字符串类型参数。

需要注意，默认空字符串不是 nil，而是 ""。

```
func main() {
    var s string

    println(s == "")       // true
    println(s == nil)      // 无效操作: s == nil (mismatched types string and nil)
}
```

使用“`”定义不做转义的原始字符串（raw string），支持跨行。

```
func main() {
    s := `line\r\n,
    line 2`

    println(s)
}
```

输出：

```
line\r\n,
    line 2
```

编译器不会解析原始字符串内的注释语句，且前置空格也属字符串内容。

支持“!=、==、<、>、+、+=”操作符。

```
func main() {
    s := "ab" +                    // 跨行时，加法操作符必须在上行结尾。
        "cd"

    println(s == "abcd")
    println(s > "abc")
}
```

允许以索引号访问字节数组（并非字符），但不能获取数组元素地址。

```
func main() {
    s := "abc"

    println(s[1])
    println(&s[1])         // 错误: cannot take the address of s[1]
}
```

以切片语法（起始和结束索引号）返回子串时，其内部依旧指向原字节数组。

```
import (
    "fmt"
    "reflect"
    "unsafe"
```

```
)

func main() {
    s := "abcdefg"

    s1 := s[:3]            // 从头开始，仅指定结束索引位置。
    s2 := s[1:4]           // 指定开始和结束位置，返回 [start, end)。
    s3 := s[2:]            // 指定开始位置，返回后面全部内容。

    println(s1, s2, s3)

    // 提示：
    //   reflect.StringHeader 和 string 头结构相同。
    //   unsafe.Pointer 用于指针类型转换。

    fmt.Printf("%#v\n", (*reflect.StringHeader)(unsafe.Pointer(&s)))
    fmt.Printf("%#v\n", (*reflect.StringHeader)(unsafe.Pointer(&s1)))
}
```

输出：

```
abc  bcd  cdefg

&StringHeader{ Data:0xfb838, Len:7 }
&StringHeader{ Data:0xfb838, Len:3 }
```

使用 for 遍历字符串时，分 byte 和 rune 两种方式。

```
func main() {
    s := "雨痕"

    for i := 0; i < len(s); i++ {                // byte
        fmt.Printf("%d: [%c]\n", i, s[i])
    }

    for i, c := range s {                        // rune，返回数组索引号，以及 Unicode 字符。
        fmt.Printf("%d: [%c]\n", i, c)
    }
}
```

输出：

```
0: [é]
1: []
2: [¨]
3: [ç]
4: []
5: []

0: [雨]
```

```
3: [痕]
```

转换

要修改字符串，须先将其转换为可变类型（[]rune 或 []byte）。但无论是朝哪个方向转换，都需重新分配内存，并复制数据。

```
// 转换指针类型，读取头部指向数组的指针。
// 使用了接口类型推断，详情参考后续章节。
func printDataPointer(format string, ptr interface{}) {
    var p uintptr

    switch v := ptr.(type) {
    case *string:
        p = (*reflect.StringHeader)(unsafe.Pointer(v)).Data
    case *[]byte:
        p = (*reflect.SliceHeader)(unsafe.Pointer(v)).Data
    case *[]rune:
        p = (*reflect.SliceHeader)(unsafe.Pointer(v)).Data
    }

    fmt.Printf(format, p)
}

func main() {
    s := "hello, world!"

    bs := []byte(s)
    s2 := string(bs)

    rs := []rune(s)
    s3 := string(rs)

    printDataPointer("s: %x\n", &s)

    printDataPointer("string to []byte, bs: %x\n", &bs)
    printDataPointer("[]byte to string, s2: %x\n", &s2)

    printDataPointer("string to []rune, rs: %x\n", &rs)
    printDataPointer("[]rune to string, s3: %x\n", &s3)
}
```

输出：

```
s: ffdf0

string to []byte, bs: c82003fe38
[]byte to string, s2: c82003fe18
```

```
string to []rune, rs: c82003fe58
[]rune to string, s3: c82003fdf8
```

可直接使用 &bs[0] 获取 slice 底层数组的起始地址。

某些时候，转换操作会严重影响算法性能，可改用“非安全”方法。

```
func toString(bs []byte) string {
    return *(*string)(unsafe.Pointer(&bs))
}

func main() {
    bs := []byte("hello, world!")
    s := toString(bs)

    printDataPointer("bs: %x\n", &bs)
    printDataPointer("s : %x\n", &s)
}
```

输出：

```
bs: c82003fec8
s : c82003fec8
```

该方法利用了 []byte 和 string 头结构“部分相同”，以非安全的指针类型转换来实现类型“变更”，从而避免了底层数组复制。在很多 Web Framework 中都能看到此类做法，在高并发高压力下，此种做法可有效改善执行性能。只是，unsafe 的使用存在一定的风险性，须小心谨慎！

也可以用 append 函数，将 string 直接追加到 []byte 内。

```
func main() {
    var bs []byte
    bs = append(bs, "abc"...)

    fmt.Println(bs)
}
```

输出：

```
[97 98 99]
```

考虑到字符串只读特征，转换时复制数据到新分配内存是可以理解的。当然，性能同样重要，编译器会为某些场合做出专门的优化策略，避免额外的分配和复制操作。

- 将 []byte 转换为 string key，去 map[string] 查询时。
- 将 string 转换为 []byte 进行 for range 迭代时，直接取字节赋值给局部变量。

用 GDB 验证一下这说法是否准确。

```
package main

func main() {
    m := map[string]int{
        "abc": 123,
    }

    key := []byte("abc")
    x, ok := m[string(key)]

    println(x, ok)
}
```

输出：

```
$ go build -gcflags "-N -l"                                          // 阻止优化。

$ gdb test

(gdb) b 9                                                            // 设置断点。
(gdb) r

Breakpoint 1, main.main () at test.go:9
9       x, ok := m[string(key)]

(gdb) info locals                                                    // 显示局部变量信息。
key = {array = 0xc820031ef0 "abc \310", len = 3, cap = 3}            // 注意 key 底层数组地址。

(gdb) b runtime.mapaccess2_faststr                                   // 在 map 访问函数上打断点。
Breakpoint 2 at 0x4090d0: runtime/hashmap_fast.go, line 298.

(gdb) c

Breakpoint 2, runtime.mapaccess2_faststr at runtime/hashmap_fast.go:298
298 func mapaccess2_faststr(t *maptype, h *hmap, ky string) (unsafe.Pointer, bool) {

(gdb) info args                     // 显示函数参数信息。
ky = 0xc820031ef0 "abc"             // 和 key []byte 底层数组地址相同，证明没有分配和复制。
```

性能

除类型转换外，动态构建字符串也容易造成性能问题。

用加法操作符拼接字符串时，每次都需重新分配内存。如此，在构建“超大”字符串时，性能就显得极差。

```
func test() string {
    var s string
    for i := 0; i < 1000; i++ {
        s += "a"
    }

    return s
}

func BenchmarkTest(b *testing.B) {
    for i := 0; i < b.N; i++ {
        test()
    }
}
```

输出：

```
BenchmarkTest-4    10000        226285 ns/op      530348 B/op        999 allocs/op
```

改进方法就是预先分配足够的内存空间。常用方法是用 strings.Join 函数，它会统计所有参数长度，并一次性完成内存分配操作。

src/strings/strings.go

```
func Join(a []string, sep string) string {
    ...

    // 统计分隔符长度。
    n := len(sep) * (len(a) - 1)

    // 统计所有待拼接字符串长度。
    for i := 0; i < len(a); i++ {
        n += len(a[i])
    }

    // 一次分配所需长度的数组空间。
    b := make([]byte, n)

    // 拷贝数据。
    bp := copy(b, a[0])
```

```
    for _, s := range a[1:] {
        bp += copy(b[bp:], sep)
        bp += copy(b[bp:], s)
    }
    return string(b)
}
```

我们以此改进测试用例，看看性能是否有所改善。

```
func test() string {
    s := make([]string, 1000)              // 分配足够的内存，避免中途扩张底层数组。
    for i := 0; i < 1000; i++ {
        s[i] = "a"
    }

    return strings.Join(s, "")
}
```

输出：

```
BenchmarkTest-4   100000         14868 ns/op        2048 B/op          2 allocs/op
```

编译器对“s1 + s2 + s3”这类表达式的处理方式和 strings.Join 类似。

显然，改进后的算法有巨大提升。还有 bytes.Buffer 也可完成相同操作，且性能相当。

```
func test() string {
    var b bytes.Buffer
    b.Grow(1000)                           // 事先准备足够的内存，避免中途扩张。

    for i := 0; i < 1000; i++ {
        b.WriteString("a")
    }

    return b.String()
}
```

输出：

```
BenchmarkTest-4   100000         15063 ns/op        2160 B/op          3 allocs/op
```

对于数量较少的字符串格式化拼接，可使用 fmt.Sprintf、text/template 等方法。

> 字符串操作通常在堆上分配内存，这会对 Web 等高并发应用会造成较大影响，会有大量字符串对象要垃圾回收。建议使用 []byte 缓存池，或在栈上自行构建等方式实现来 zero-garbage。

Unicode

类型 rune 专门用来存储 Unicode Code Point，它是 int32 的别名，相当于 UCS-4/UTF-32 编码格式。使用单引号的字面量，其默认类型就是 rune。

```
func main() {
    r := '我'
    fmt.Printf("%T\n", r)
}
```

输出：

```
int32
```

除 []rune 外，还可直接在 rune、byte、string 间进行转换。

```
func main() {
    r := '我'

    s := string(r)         // rune to string
    b := byte(r)           // rune to byte

    s2 := string(b)        // byte to string
    r2 := rune(b)          // byte to rune

    fmt.Println(s, b, s2, r2)
}
```

要知道字符串存储的字节数组，不一定就是合法的 UTF-8 文本。

```
import (
    "fmt"
    "unicode/utf8"
)

func main() {
    s := "雨痕"
    s = string(s[0:1] + s[3:4])           // 截取并拼接一个“不合法”的字符串。
```

```
    fmt.Println(s, utf8.ValidString(s))
}
```

输出：

```
?? false
```

标准库 unicode 里提供了丰富的操作函数。除验证函数外，还可用 RuneCountInString 代替 len 返回准确的 Unicode 字符数量。

```
func main() {
    s := "雨.痕"
    println(len(s), utf8.RuneCountInString(s))
}
```

输出：

```
7  3
```

官方扩展库 golang.org/x/text/encoding/unicode 提供了对 BOM 的支持。

---

2. 数组

定义数组类型时，数组长度必须使用非负整型常量表达式。数组长度是类型的一部分，也就是说，元素类型相同，但数组长度不同的不属同一类型。

```
func main() {
    var d1 [3]int
    var d2 [2]int
    d1 = d2         // 错误: cannot use d2 (type [2]int) as type [3]int in assignment
}
```

可使用灵活的初始化方式。

```
func main() {
    var a [4]int                    // 元素自动初始化为零。
```

```
    b := [4]int{2, 5}               // 未提供初始值的元素自动初始化为 0。
    c := [4]int{5, 3: 10}           // 可指定索引位置初始化。

    d := [...]int{1, 2, 3}          // 编译器按初始化值数量确定数组长度。
    e := [...]int{10, 3: 100}       // 支持索引初始化，但注意数组长度与此有关。

    fmt.Println(a, b, c, d, e)
}
```

输出：

```
[0 0 0 0]

[2 5 0 0]
[5 0 0 10]

[1 2 3]
[10 0 0 100]
```

对于结构等复合类型，可省略元素初始化表达式中的类型标签。

```
func main() {
   type user struct {
        name string
        age  byte
    }

    d := [...]user{
        {"Tom", 20},                // 省略了类型标签。
        {"Mary", 18},
   }

    fmt.Printf("%#v\n", d)
}
```

输出：

```
[2]main.user{
    main.user{name:"Tom", age:0x14},
    main.user{name:"Mary", age:0x12}
}
```

在多维数组定义中，仅第一纬度允许使用“...”。

```
func main() {
    a := [2][2]int{
```

```
        {1, 2},
        {3, 4},
    }

    b := [...][2]int{
        {10, 20},
        {30, 40},
    }

    c := [...][2][2]int{          // 三维数组
        {
            {1, 2},
            {3, 4},
        },
        {
            {10, 20},
            {30, 40},
        },
    }

    fmt.Println(a)
    fmt.Println(b)
    fmt.Println(c)
}
```

输出：

```
[ [1 2], [3 4] ]                  // 为便于阅读，输出结果经过整理。

[ [10 20], [30 40] ]

[
    [ [1 2], [3 4] ]
    [ [10 20], [30 40] ]
]
```

内置函数 len 和 cap 都返回第一纬度长度。

```
func main() {
    a := [2]int{}
    b := [...][2]int{
        {10, 20},
        {30, 40},
        {50, 60},
    }

    println(len(a), cap(a))
    println(len(b), cap(b))
    println(len(b[1]), cap(b[1]))
}
```

输出：

```
2 2
3 3
2 2
```

如果数组元素类型支持“==、!=”操作符，那么数组也支持比较操作。

```
func main() {
    var a, b [2]int
    println(a == b)

    c := [2]int{1, 2}
    d := [2]int{0, 1}
    println(c != d)

    var e, f [2]map[string]int
    println(e == f)               // 无效操作: e == f ([2]map[string]int cannot be compared)
}
```

指针

要分清指针数组和数组指针的区别。指针数组是指元素为指针类型的数组，数组指针是获取数组变量的地址。

```
func main() {
    x, y := 10, 20
    a := [...]*int{&x, &y}                // 元素为指针的指针数组。
    p := &a                               // 存储数组地址的指针。

    fmt.Printf("%T, %v\n", a, a)
    fmt.Printf("%T, %v\n", p, p)
}
```

输出：

```
[2]*int, [0xc82000a298  0xc82000a2c0]
*[2]*int, &[0xc82000a298  0xc82000a2c0]
```

可获取任意元素地址。

```
func main() {
```

```
    a := [...]int{1, 2}
    println(&a, &a[0], &a[1])
}
```

输出：

```
0xc82003ff20  0xc82003ff20  0xc82003ff28
```

数组指针可直接用来操作元素。

```
func main() {
    a := [...]int{1, 2}
    p := &a

    p[1] += 10
    println(p[1])
}
```

输出：

```
12
```

可通过 unsafe.Pointer 转换不同长度的数组指针来实现越界访问，或使用参数 gcflags "-B" 阻止编译器插入检查指令。

复制

与 C 数组变量隐式作为指针使用不同，Go 数组默认是值类型，赋值和传参操作都会复制整个数组内容。

```
func test(x [2]int) {
    fmt.Printf("x: %p, %v\n", &x, x)
}

func main() {
    a := [2]int{10, 20}

    var b [2]int
    b = a

    fmt.Printf("a: %p, %v\n", &a, a)
    fmt.Printf("b: %p, %v\n", &b, b)
```

```
    test(a)
}
```

输出：

```
a: 0xc820076050, [10 20]
b: 0xc820076060, [10 20]
x: 0xc8200760a0, [10 20]
```

如果需要，可改用指针或 slice，以避免数据复制。

```
func test(x *[2]int) {
    fmt.Printf("x: %p, %v\n", x, *x)
    x[1] += 100
}

func main() {
    a := [2]int{10, 20}
    test(&a)

    fmt.Printf("a: %p, %v\n", &a, a)
}
```

输出：

```
x: 0xc8200741c0, [10 20]
a: 0xc8200741c0, [10 120]
```

3. 切片

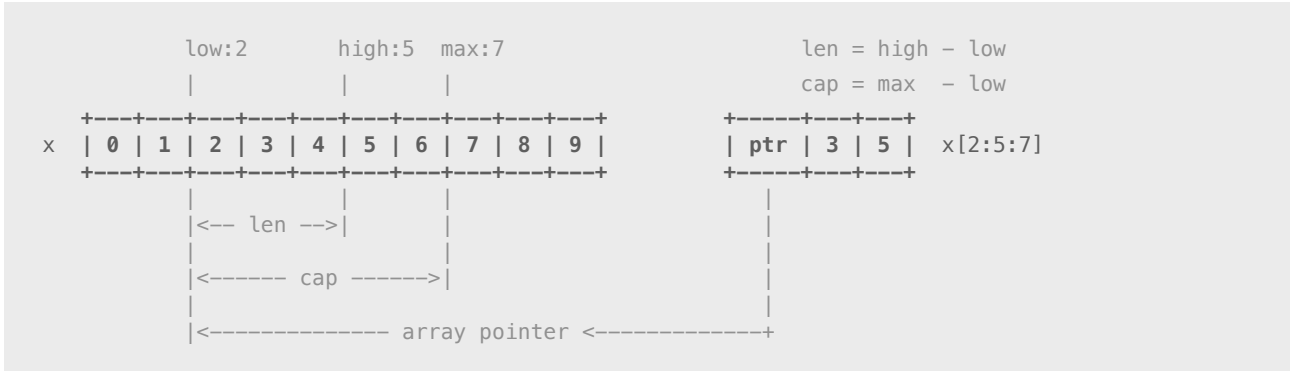
切片类型（slice）本身并不是动态数组或数组指针。它内部通过指针引用底层数组，设定相关属性将操作限定在指定范围内。当需要时，会申请更大的内存，将当前数据复制过去，以实现类似动态数组的功能。

```
type slice struct {
    array unsafe.Pointer
    len   int
    cap   int
}
```

可基于数组或数组指针创建切片，以开始和结束索引位置确定所引用片段，但不支持反向索引。实际内容是一个右半开区间，不要被语法误导。

```
  x := [...]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

  expression   slice                     len    cap
-------------+-------------------------+------+------+-------------------------------
  x[:]         [0 1 2 3 4 5 6 7 8 9]     10     10     x[0:len(x)]
  x[2:5]       [2 3 4]                   3      8
  x[2:5:7]     [2 3 4]                   3      5
  x[4:]        [4 5 6 7 8 9]             6      6      x[4:len(x)]
  x[:4]        [0 1 2 3]                 4      10     x[0:4]
  x[:4:6]      [0 1 2 3]                 4      6      x[0:4:6]
```



属性 cap 表示切片所引用数组片段真实长度，len 用于限定可读写元素数量。另外，数组必须 addressable，否则会引发错误。

```
func main() {
    m := map[string][2]int{
        "a": {1, 2},
    }

    s := m["a"][:]         // 无效操作 m["a"][:] (slice of unaddressable value)
    fmt.Println(s)
}
```

和数组一样，切片同样使用索引号访问元素内容。起始索引为 0，而非对应的底层数组真实索引位置。

```
func main() {
    x := [...]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
    s := x[2:5]
```

```
    for i := 0; i < len(s); i++ {
        println(s[i])
    }
}
```

输出：

```
2
3
4
```

可直接创建切片对象，无需预先准备数组。因为是引用类型，须使用 make 函数或显式初始化语句，它会自动完成底层数组内存分配。

```
func main() {
    s1 := make([]int, 3, 5)              // 指定 len、cap，底层数组初始化为零值。
    s2 := make([]int, 3)                 // 省略 cap，和 len 相等。
    s3 := []int{10, 20, 5: 30}           // 按初始化元素分配底层数组，并设置 len、cap。

    fmt.Println(s2, len(s2), cap(s2))
    fmt.Println(s1, len(s1), cap(s1))
    fmt.Println(s3, len(s3), cap(s3))
}
```

输出：

```
[0 0 0]           3  5
[0 0 0]           3  3
[10 20 0 0 0 30]  6  6
```

注意下面两种定义方式的区别。前者仅定义了一个 []int 类型变量，并未执行初始化操作，而后者则用初始化表达式完成了全部创建过程。

```
func main() {
    var a []int
    b := []int{}

    println(a == nil, b == nil)
}
```

输出：

```
true  false
```

通过输出更详细的信息，我们可以看到两者的差异。

```
func main() {
    var a []int
    b := []int{}

    fmt.Printf("a: %#v\n", (*reflect.SliceHeader)(unsafe.Pointer(&a)))
    fmt.Printf("b: %#v\n", (*reflect.SliceHeader)(unsafe.Pointer(&b)))

    fmt.Printf("a size: %d\n", unsafe.Sizeof(a))
}
```

输出：

```
a: &reflect.SliceHeader{Data:0x0, Len:0, Cap:0}
b: &reflect.SliceHeader{Data:0x19c730, Len:0, Cap:0}

a size: 24
```

变量 b 的数组指针被赋值，尽管它指向 runtime.zerobase，但它依然完成了初始化操作。
另外，a == nil，不表示 a 是一个 nil 指针，它本身是一个 struct，依然会分配所需内存。
甚至可以直接对 nil 切片执行 slice[:] 操作，同样返回 nil。

不支持比较操作（就算元素类型支持也不行），仅能判断是否为 nil。

```
func main() {
    a := make([]int, 1)
    b := make([]int, 1)

    println(a == b)        // 无效操作: a == b (slice can only be compared to nil)
}
```

可获取元素地址，但不能向数组那样直接通过指针（*slice）访问元素内容。

```
func main() {
    s := []int{0, 1, 2, 3, 4}

    p := &s                // 取 header 地址。
    p0 := &s[0]            // 取 array[0] 地址。
    p1 := &s[1]

    println(p, p0, p1)
```

```
    (*p)[1] += 100          // *[]int 不支持 indexing 操作，须先用指针操作符获取 []int 对象。
    fmt.Println(s)
}
```

输出：

```
0xc82003ff00  0xc8200141e0  0xc8200141e8
[0 101 2 3 4]
```

如果元素类型也是切片，那么就能实现类似交错数组的功能。

```
func main() {
    x := [][]int{
        {1, 2},
        {10, 20, 30},
        {100},
    }

    fmt.Println(x[1])

    x[2] = append(x[2], 200, 300)
    fmt.Println(x[2])
}
```

输出：

```
[10 20 30]
[100 200 300]
```

很显然，切片只是很小的结构体，用来代替数组传参可避免复制开销。还有，make 函数允许在运行期动态指定底层数组长度，避免了数组类型必须使用编译期常量的限制。

并非所有时候都适合用切片代替数组，因为切片底层数组可能会在堆上分配内存。而且小数组在栈上拷贝的消耗也未必就比 make 代价大。

```
func array() [1024]int {
    var x [1024]int
    for i := 0; i < len(x); i++ {
        x[i] = i
    }

    return x
}

func slice() []int {
    x := make([]int, 1024)
```

```
    for i := 0; i < len(x); i++ {
        x[i] = i
    }

    return x
}

func BenchmarkArray(b *testing.B) {
    for i := 0; i < b.N; i++ {
        array()
    }
}

func BenchmarkSlice(b *testing.B) {
    for i := 0; i < b.N; i++ {
        slice()
    }
}
```

输出：

```
BenchmarkArray-4     1000000          1303 ns/op           0 B/op          0 allocs/op
BenchmarkSlice-4      500000          2690 ns/op        8192 B/op          1 allocs/op
```

reslice

将切片视作 [cap]slice 数据源，创建新切片对象。不能超出 cap，但不受 len 限制。

```
+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |   |   |   |  s              len:6, cap:10
+---+---+---+---+---+---+---+---+---+---+
0           3                   8       10
            +---+---+---+---+---+---+---+
            | 3 | 4 | 5 | 6 | 0 |   |   |  s1 = s[3:8]     len:5, cap:7
            +---+---+---+---+---+---+---+
            0       2       4       6
                    +---+---+---+---+
                    | 5 | 6 |   |   |      s2 = s1[2:4:6]  len:2, cap:4
                    +---+---+---+---+
                    0   1
                    +---+---+---+---+---+
                    | 5 |   |   |   |   |  s3 = s2[:1:5]   error: slice bounds out of range
                    +---+---+---+---+---+
```

新建切片对象依旧指向原底层数组，也就是说修改对所有关联切片可见。

```
func main() {
    d := [...]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

    s1 := d[3:7]
```

```
    s2 := s1[1:3]

    for i := range s2 {
        s2[i] += 100
    }

    fmt.Println(d)
    fmt.Println(s1)
    fmt.Println(s2)
}
```

输出：

```
[0 1 2 3 104 105 6 7 8 9]
[3 104 105 6]
[104 105]
```

利用 reslice 操作，很容易就能实现一个栈式数据结构。

```
func main() {
    // 栈最大容量 5。
    stack := make([]int, 0, 5)

    // 入栈。
    push := func(x int) error {
        n := len(stack)
        if n == cap(stack) {
            return errors.New("stack is full")
        }

        stack = stack[:n+1]
        stack[n] = x

        return nil
    }

    // 出栈。
    pop := func() (int, error) {
        n := len(stack)
        if n == 0 {
            return 0, errors.New("stack is empty")
        }

        x := stack[n-1]
        stack = stack[:n-1]

        return x, nil
    }

    // 入栈测试。
    for i := 0; i < 7; i++ {
```

```
        fmt.Printf("push %d: %v, %v\n", i, push(i), stack)
    }

    // 出栈测试。
    for i := 0; i < 7; i++ {
        x, err := pop()
        fmt.Printf("pop: %d, %v, %v\n", x, err, stack)
    }
}
```

输出：

```
push 0: <nil>, [0]
push 1: <nil>, [0 1]
push 2: <nil>, [0 1 2]
push 3: <nil>, [0 1 2 3]
push 4: <nil>, [0 1 2 3 4]
push 5: stack is full, [0 1 2 3 4]
push 6: stack is full, [0 1 2 3 4]

pop: 4, <nil>, [0 1 2 3]
pop: 3, <nil>, [0 1 2]
pop: 2, <nil>, [0 1]
pop: 1, <nil>, [0]
pop: 0, <nil>, []
pop: 0, stack is empty, []
pop: 0, stack is empty, []
```

append

向切片尾部（slice[len]）添加数据，并返回新的切片对象。

```
func main() {
    s := make([]int, 0, 5)

    s1 := append(s, 10)
    s2 := append(s1, 20, 30)             // 追加多个数据。

    fmt.Println(s, len(s), cap(s))       // 不会修改原 slice 属性。
    fmt.Println(s1, len(s1), cap(s1))
    fmt.Println(s2, len(s2), cap(s2))
}
```

输出：

```
[]          0  5
[10]        1  5
[10 20 30]  3  5
```

数据被追加到原底层数组。如超出 cap 限制，则为新切片对象重新分配数组。

```
func main() {
    s := make([]int, 0, 100)
    s1 := s[:2:4]
    s2 := append(s1, 1, 2, 3, 4, 5, 6)    // 超出 s1 cap 限制，分配新底层数组。

    fmt.Printf("s1: %p: %v\n", &s1[0], s1)
    fmt.Printf("s2: %p: %v\n", &s2[0], s2)

    fmt.Printf("s data: %v\n", s[:10])
    fmt.Printf("s1 cap: %d, s2 cap: %d\n", cap(s1), cap(s2))
}
```

输出：

```
s1: 0xc82007e380: [0 0]
s2: 0xc820070100: [0 0 1 2 3 4 5 6]       // 数组地址不同，确认新分配。
s data: [0 0 0 0 0 0 0 0 0 0]             // append 并未向原数组写入部分数据。
s1 cap: 4, s2 cap: 8                      // 新数组是原 cap 的 2 倍。
```

注意：

- 是超出切片 cap 限制，而非底层数组长度限制，因为 cap 可小于底层数组长度。
- 新分配数组长度是原 cap 的 2 倍，而非原数组的 2 倍。

并非总是 2 倍，对于较大的切片，会尝试扩容 1/4，以节约内存。

向 nil 切片追加数据时，直接分配底层数组内存。

```
func main() {
    var s []int
    s = append(s, 1, 2, 3)
    fmt.Println(s)
}
```

输出：

```
[1 2 3]
```

正因为存在重新分配底层数组的缘故，在某些场合建议预留足够多的空间，避免中途内存分配和数据复制开销。

copy

在两个切片对象间复制数据，允许指向同一底层数组，允许目标区间重叠。最终所复制长度以较短的切片 len 为准。

```
func main() {
    s := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

    s1 := s[5:8]
    n := copy(s[4:], s1)          // 在同一底层数组的不同区间复制。
    fmt.Println(n, s)

    s2 := make([]int, 6)          // 在不同数组间复制。
    n = copy(s2, s)
    fmt.Println(n, s2)
}
```

输出：

```
3  [0 1 2 3 5 6 7 7 8 9]          // copy([4 5 6 7 8 9], [5 6 7])
6  [0 1 2 3 5 6]
```

还可直接从字符串中复制数据到 []byte。

```
func main() {
    b := make([]byte, 3)
    n := copy(b, "abcde")
    fmt.Println(n, b)
}
```

输出：

```
3  [97 98 99]
```

如果切片仅引用大数组很小的片段时，那么建议新建独立切片，复制所需数据，以便原数组内存被及时回收。

4. 字典

字典（哈希表）是一种使用频率很高的数据结构。将其作为语言内置类型，且从运行时层面提供优化，可获得更高效的性能。

作为无序键值对集合，字典要求 key 必须是支持相等运算符（==、!=）的数据类型。比如，数字、字符串、指针、数组、结构体，及其对应接口类型。

字典是引用类型，使用 make 函数或初始化表达语句创建。

```
func main() {
    m := make(map[string]int)
    m["a"] = 1
    m["b"] = 2

    m2 := map[int]struct {         // 值为匿名结构类型。
        x int
    }{
        1:  {x: 100},              // 可省略 key、value 类型标签。
        2:  {x: 200},
    }

    fmt.Println(m, m2)
}
```

基本操作演示：

```
func main() {
    m := map[string]int{
        "a": 1,
        "b": 2,
    }

    m["a"] = 10                    // 修改。
    m["c"] = 30                    // 新增。

    if v, ok := m["d"]; ok {       // 判断 key 是否存在，返回值。
        println(v)
    }

    delete(m, "d")                 // 删除键值对。不存在时，不会出错。
}
```

访问不存在的键值，默认返回零值，不会引发错误。但推荐做法是 ok-idiom 模式，毕竟通过零值并不能判断键值是否存在，亦或许存储的 value 就是零。

对字典进行迭代，每次返回的键值次序都不相同。

```
func main() {
    m := make(map[string]int)

    for i := 0; i < 8; i++ {
        m[string('a'+i)] = i
    }

    for i := 0; i < 4; i++ {
        for k, v := range m {
            print(k, ":", v, "  ")
        }

        println()
    }
}
```

输出：

```
h:7  a:0  b:1  c:2  d:3  e:4  f:5  g:6
g:6  h:7  a:0  b:1  c:2  d:3  e:4  f:5
d:3  e:4  f:5  g:6  h:7  a:0  b:1  c:2
b:1  c:2  d:3  e:4  f:5  g:6  h:7  a:0
```

函数 len 返回当前键值对数量，cap 不接受字典类型。另外，因内存访问安全和哈希算法等缘故，字典被设计成“not addressable”，因此不能直接修改值成员（结构或数组等）。

```
func main() {
    type user struct {
        name string
        age  byte
    }

    m := map[int]user{
        1: {"Tom", 19},
    }

    m[1].age += 1          // 错误: cannot assign to m[1].age
}
```

正确做法修改后是对 value 赋值，或直接使用指针类型。

```
func main() {
    type user struct {
        name string
        age  byte
    }

    m := map[int]user{
        1: {"Tom", 19},
    }

    u := m[1]
    u.age += 1
    m[1] = u                        // 替换 value。

    m2 := map[int]*user{
        1: &user{"Jack", 20},
    }

    m2[1].age++                     // m2[1] 返回的是指针，透过指针修改目标对象。
}
```

同理，m[key]++ 是合法操作。

我们不能对 nil 字典进行写操作，但却能读。

```
func main() {
    var m map[string]int
    println(m["a"])        // 返回零值。
    m["a"] = 1             // panic: assignment to entry in nil map
}
```

注意内容为空的字典，与 nil 是不同的。

```
func main() {
    var m1 map[string]int
    m2 := map[string]int{}
    println(m1 == nil, m2 == nil)
}
```

输出：

```
true  false
```

安全

在迭代期间删除或新增键值是安全的。

```
func main() {
    m := make(map[int]int)

    for i := 0; i < 10; i++ {
        m[i] = i + 10
    }

    for k := range m {
        if k == 5 {
            m[100] = 1000
        }

        delete(m, k)
        fmt.Println(k, m)
    }
}
```

输出：

```
2 map[6:16 7:17 9:19 0:10 1:11 5:15 8:18 3:13 4:14]
3 map[4:14 8:18 0:10 1:11 5:15 6:16 7:17 9:19]
4 map[1:11 5:15 6:16 7:17 9:19 0:10 8:18]
8 map[7:17 9:19 0:10 1:11 5:15 6:16]
0 map[1:11 5:15 6:16 7:17 9:19]
1 map[5:15 6:16 7:17 9:19]
5 map[6:16 7:17 9:19 100:1000]
6 map[100:1000 7:17 9:19]
7 map[100:1000 9:19]
9 map[100:1000]
```

就此例而言，不能保证迭代操作会删除新增的键值。

运行时会对字典并发操作做出检测。如某个 goroutine 正在对字典进行写操作，那么其他 goroutine 就不能对该字典执行并发操作（读、写、删除），否则会导致进行崩溃。

```
import "time"

func main() {
    m := make(map[string]int)

    go func() {
```

```
        for {
            m["a"] += 1                         // 写操作。
            time.Sleep(time.Microsecond)
        }
    }()

    go func() {
        for {
            _ = m["b"]                          // 读操作。
            time.Sleep(time.Microsecond)
        }
    }()

    select {}                                   // 阻止进程退出。
}
```

输出：

```
fatal error: concurrent map read and map write
```

可启用数据竞争（data race）检查此类问题，它会输出极为详细的提示信息。

```
$ go run -race test.go

==================
WARNING: DATA RACE

Write by goroutine 5:
  runtime.mapassign1()
      hashmap.go:429 +0x0
  main.main.func1()
      test.go:13 +0xbe

Previous read by goroutine 6:
  runtime.mapaccess1_faststr()
      hashmap_fast.go:193 +0x0
  main.main.func2()
      test.go:20 +0x60

Goroutine 5 (running) created at:
  main.main()
      test.go:16 +0x76

Goroutine 6 (running) created at:
  main.main()
      test.go:23 +0x98
==================
fatal error: concurrent map read and map write
```

可使用 sync.RWMutex 实现并发同步，避免读写同时进行。

```
import (
    "sync"
    "time"
)

func main() {
    var lock sync.RWMutex                // 使用读写锁，以获得最佳性能。
    m := make(map[string]int)

    go func() {
        for {
            lock.Lock()                  // 注意锁的粒度。
            m["a"] += 1
            lock.Unlock()                // 不能使用 defer。

            time.Sleep(time.Microsecond)
        }
    }()

    go func() {
        for {
            lock.RLock()
            _ = m["b"]
            lock.RUnlock()

            time.Sleep(time.Microsecond)
        }
    }()

    select {}
}
```

性能

字典对象本身就是指针类型，传参时无需再次取地址。

```
func test(x map[string]int) {
    fmt.Printf("x: %p\n", x)
}

func main() {
    m := make(map[string]int)
    test(m)
    fmt.Printf("m: %p, %d\n", m, unsafe.Sizeof(m))

    m2 := map[string]int{}
    test(m2)
```

```
    fmt.Printf("m2: %p, %d\n", m2, unsafe.Sizeof(m2))
}
```

输出：

```
x : 0xc8200780c0
m : 0xc8200780c0, 8

x : 0xc8200780f0
m2: 0xc8200780f0, 8
```

在创建时预先准备足够空间有助于提升性能，减少扩张时内存分配和重新哈希操作。

```
func test() map[int]int {
    m := make(map[int]int)
    for i := 0; i < 1000; i++ {
        m[i] = i
    }

    return m
}

func testCap() map[int]int {
    m := make(map[int]int, 1000)          // 预先准备足够的空间。
    for i := 0; i < 1000; i++ {
        m[i] = i
    }

    return m
}

func BenchmarkTest(b *testing.B) {
    for i := 0; i < b.N; i++ {
        test()
    }
}

func BenchmarkTestCap(b *testing.B) {
    for i := 0; i < b.N; i++ {
        testCap()
    }
}
```

输出：

```
BenchmarkTest-4        10000        154601 ns/op       89557 B/op         98 allocs/op
BenchmarkTestCap-4     20000         63804 ns/op       41828 B/op         12 allocs/op
```

对于海量小对象，应直接用字典存储键值数据拷贝，而非指针。这有助于减少需要扫描的对象数量，大幅缩短垃圾回收时间。另外，字典不会收缩内存，所以适当替换成新对象是很有必要的。

5. 结构

结构体（struct）将多个不同类型命名字段（field）序列打包成一个复合类型。

字段名必须唯一，可用“_”补位，支持使用自身类型的指针成员。字段名、排列顺序属类型组成部分。除对齐处理外，编译器不会优化、调整内存布局。

```
type node struct {
    _    int
    id   int
    next *node
}

func main() {
    n1 := node{
        id: 1,
    }

    n2 := node{
        id:   2,
        next: &n1,
    }

    fmt.Println(n1, n2)
}
```

可按顺序初始化全部字段，或使用命名方式初始化指定字段。

```
func main() {
    type user struct {
        name string
        age  byte
    }

    u1 := user{"Tom", 12}
    u2 := user{"Tom"}             // 错误: too few values in struct initializer
    fmt.Println(u1, u2)
}
```

> 推荐使用命名初始化方式。这样在扩充结构字段或调整字段顺序时，不会导致初始化语句出错。

可直接定义匿名结构类型变量，或用作字段类型。但因缺少类型标识，在作为字段类型时无法直接初始化，稍显麻烦。

```
func main() {
    u := struct {                   // 直接定义匿名结构变量。
        name string
        age  byte
    }{
        name: "Tom",
        age:  12,
    }

    type file struct {
        name string
        attr struct {               // 定义匿名结构类型字段。
            owner int
            perm  int
        }
    }

    f := file{
        name: "test.dat",

        // attr: {                  // 错误: missing type in composite literal
        //     owner: 1,
        //     perm:  0755,
        // },
    }

    f.attr.owner = 1                // 正确方式。
    f.attr.perm = 0755

    fmt.Println(u, f)
}
```

> 当然，也可在初始化语句中使用未命名类型，但那就失去了匿名结构的便利性。不知道以后版本，是否会如同 1.5 修正 map 初始化语法一样改进这个规则。

仅在字段类型全部支持时，才可做相等操作。

```
func main() {
    type data struct {
```

```
        x   int
        y   map[string]int
    }

    d1 := data{
        x: 100,
    }

    d2 := data{
        x: 100,
    }

    println(d1 == d2)      // 无效操作: struct containing map[string]int cannot be compared
}
```

可使用指针直接操作结构字段，但不能是多级指针。

```
func main() {
    type user struct {
        name string
        age  int
    }

    p := &user{
        name: "Tom",
        age:  20,
    }

    p.age++

    p2 := &p
    *p2.name = "Jack"   // 错误: p2.name undefined (type **user has no field or method name)
}
```

空结构

空结构（struct{}）是指没有字段的结构类型。它比较特殊，是因为无论是其自身，还是作为数组元素类型，其长度都为零。

```
func main() {
    var a struct{}
    var b [100]struct{}

    println(unsafe.Sizeof(a), unsafe.Sizeof(b))
}
```

输出：

```
0 0
```

尽管没有分配存储内存，但依然可以操作数组元素，对应切片 len、cap 属性也正常。

```
func main() {
    var d [100]struct{}
    s := d[:]

    d[1] = struct{}{}
    s[2] = struct{}{}

    fmt.Println(s[3], len(s), cap(s))
}
```

输出：

```
{} 100 100
```

实际上，这类“长度”为零的对象通常都指向 runtime.zerobase 变量。

```
func main() {
    a := [10]struct{}{}
    b := a[:]                       // 指底层数组，而非 header。
    c := [0]int{}

    fmt.Printf("%p, %p, %p\n", &a[0], &b[0], &c)
}
```

输出：

```
0x19c730, 0x19c730, 0x19c730
```

空结构对象可用于 channel，作为事件通知。

```
func main() {
    exit := make(chan struct{})

    go func() {
        println("hello, world!")
        exit <- struct{}{}
    }()
```

```
    <-exit
    println("end.")
}
```

匿名字段

所谓匿名字段（anonymous field），是指没有名字，仅有类型的字段。也被称作嵌入字段或嵌入类型。

```
type attr struct {
    perm int
}

type file struct {
    name string
    attr                    // 仅有类型名。
}
```

从编译器角度看，这只是隐式以类型标识作名字的字段。相当于语法糖，可实现面向对象语言中引用基类成员的使用方法。

```
func main() {
    type attr struct {
        perm int
    }

    type file struct {
        name string
        attr
    }

    f := file{"test.dat", attr{0755}}

    f = file{
        name: "test.dat",
        attr: attr{
            perm: 0755,
        },
    }

    f.perm = 0644           // 设置匿名字段成员。
    println(f.perm)         // 读取匿名字段成员。
}
```

对于其他包里的类型，隐式字段名字不包括包名称。

```
func main() {
    type data struct {
        os.File
    }

    d := data{
        File: os.File{},           // 注意区分字段名字和初始化成员对象的差别。
    }

    fmt.Printf("%#v\n", d)
}
```

不仅仅是结构体，除接口指针和多级指针以外的任何命名类型都可作为匿名字段。

```
func main() {
    type data struct {
        *int                       // 嵌入指针类型。
        string
    }

    x := 100

    d := data{
        int:    &x,                // 使用基础类型作为字段名。
        string: "abc",
    }

    fmt.Printf("%#v\n", d)
}
```

输出：

```
main.data{
    int:(*int)(0xc8200741c0),
    string:"abc"
}
```

未命名类型没有名字标识，自然无法提供隐式字段名。

```
type a *int
type b **int
type c interface{}
```

```
type d struct {
    *a              // 错误: embedded type cannot be a pointer
    b               // 错误: embedded type cannot be a pointer
    *c              // 错误: embedded type cannot be a pointer to interface
}
```

不能将基础类型和其指针类型同时嵌入，因为两者字段名字相同。

```
type data struct {
    *int
    int                    // 错误: duplicate field int
}
```

虽然可以像普通字段那样访问匿名字段成员，但会存在重名问题。默认情况下，编译器从当前显式命名字段开始，逐步向内查找匿名字段成员。如匿名字段成员被外层同名字段遮蔽，那么必须使用显式字段名。

```
func main() {
    type file struct {
        name string
    }

    type data struct {
        file
        name string               // 与匿名字段 file.name 重名。
    }

    d := data{
        name: "data",
        file: file{"file"},
    }

    d.name = "data2"                      // 访问 data.name。
    d.file.name = "file2"                 // 使用显式字段名访问 data.file.name。

    fmt.Println(d.name, d.file.name)
    fmt.Printf("%#v\n", d)
}
```

如果多个相同层次的匿名字段成员重名，就只能使用显式字段名访问，因为编译器无法确认正确目标。

```
func main() {
```

```
    type file struct {
        name string
    }

    type log struct {
        name string
    }

    type data struct {
        file                        // file 和 log 层次相同。
        log                         // file.name 和 log.name 重名。
    }

    d := data{}
    d.name = "name"                 // 错误: ambiguous selector d.name

    d.file.name = "file"
    d.log.name = "log"
}
```

严格来说，Go 并不是传统意义上的面向对象编程语言。匿名字段不是继承机制，也无法做多态处理。虽然配合方法集，可以用接口来实现一些类似的调用操作，但其本质是完全不同的。

字段标签

字段标签（tag）并不是注释，而是用来对字段进行描述的元数据。尽管它不属于数据成员，但却是类型的组成部分。

在运行期，可用反射获取标签信息。常被用作格式校验，数据库关系映射等。

```
func main() {
    type user struct {
        name string `昵称`
        sex  byte   `性别`
    }

    u := user{"Tom", 1}
    v := reflect.ValueOf(u)
    t := v.Type()

    for i, n := 0, t.NumField(); i < n; i++ {
        fmt.Printf("%s: %v\n", t.Field(i).Tag, v.Field(i))
    }
}
```

输出：

```
昵称: Tom
性别: 1
```

标准库 reflect.StructTag 还提供了键值数据解析功能。有关反射更多信息，请参考后续章节。

内存布局

不管结构体包含多少字段，其内存总是一次性分配的，各字段在相邻的地址空间按定义顺序排列。当然，对于引用类型、字符串和指针，结构内存中只包含其基本（头部）数据。还有，所有匿名字段成员也被包含在内。

借助 unsafe 包相关函数，我们可以输出所有字段的偏移量和长度。

```
func main() {
    type point struct {
        x, y int
    }

    type value struct {
        id   int           // 基本类型。
        name string        // 字符串。
        data []byte        // 引用类型。
        next *value        // 指针类型。
        point              // 匿名字段
    }

    v := value{
        id:   1,
        name: "test",
        data: []byte{1, 2, 3, 4},
        point: point{
            x:  100,
            y:  200,
        },
    }

    s := `
        v: %p ~ %x, size: %d, align: %d

        field   address        offset   size
        ------+--------------+--------+-----------
        id      %p             %d       %d
        name    %p             %d       %d
```

```
        data    %p              %d      %d
        next    %p              %d      %d
        x       %p              %d      %d
        y       %p              %d      %d
    `

    fmt.Printf(s,
        &v, uintptr(unsafe.Pointer(&v))+unsafe.Sizeof(v), unsafe.Sizeof(v), unsafe.Alignof(v),

        &v.id,   unsafe.Offsetof(v.id),   unsafe.Sizeof(v.id),
        &v.name, unsafe.Offsetof(v.name), unsafe.Sizeof(v.name),
        &v.data, unsafe.Offsetof(v.data), unsafe.Sizeof(v.data),
        &v.next, unsafe.Offsetof(v.next), unsafe.Sizeof(v.next),
        &v.x,    unsafe.Offsetof(v.x),    unsafe.Sizeof(v.x),
        &v.y,    unsafe.Offsetof(v.y),    unsafe.Sizeof(v.y))
}
```

输出：

```
v: 0xc820012140 ~ c820012188, size: 72, align: 8

 field   address         offset    size
-------+---------------+---------+-----------
 id      0xc820012140      0         8
 name    0xc820012148      8        16
 data    0xc820012158     24        24
 next    0xc820012170     48         8
 x       0xc820012178     56         8
 y       0xc820012180     64         8
```

```
     |------- name --------|------------- data -------------|      |------ point ------|
+----+----------+----------+----------+----------+----------+------+---------+---------+
| id | name.ptr | name.len | data.ptr | data.len | data.cap | next | point.x | point.y |
+----+----------+----------+----------+----------+----------+------+---------+---------+
0    8          16         24         32         40         48     56        64        72
```

在分配内存时，字段须做对齐处理，通常以所有字段中最长的基础类型宽度为标准。

```
func main() {
    v1 := struct {
        a   byte
        b   byte
        c   int32          // 对齐宽度 4
    }{}

    v2 := struct {
        a   byte
        b   byte           // 对齐宽度 1
    }{}
```

```
    v3 := struct {
        a   byte
        b   []int          // 基础类型 int，对齐宽度 8。
        c   byte
    }{}

    fmt.Printf("v1: %d, %d\n", unsafe.Alignof(v1), unsafe.Sizeof(v1))
    fmt.Printf("v2: %d, %d\n", unsafe.Alignof(v2), unsafe.Sizeof(v2))
    fmt.Printf("v3: %d, %d\n", unsafe.Alignof(v3), unsafe.Sizeof(v3))
}
```

输出：

```
v1: 4, 8
v2: 1, 2
v3: 8, 40
```

比较特殊的是空结构字段。如果它是最后一个字段，那么编译器将其当做长度为 1 的类型做对齐处理，以便其地址不会越界，避免引发垃圾回收错误。

```
func main() {
    v := struct {
        a   struct{}
        b   int
        c   struct{}
    }{}

    s := `
        v: %p ~ %x, size: %d, align: %d

        field   address     offset    size
        ------+-----------+---------+---------
        a       %p          %d        %d
        b       %p          %d        %d
        c       %p          %d        %d
    `

    fmt.Printf(s,
        &v, uintptr(unsafe.Pointer(&v))+unsafe.Sizeof(v), unsafe.Sizeof(v), unsafe.Alignof(v),
        &v.a, unsafe.Offsetof(v.a), unsafe.Sizeof(v.a),
        &v.b, unsafe.Offsetof(v.b), unsafe.Sizeof(v.b),
        &v.c, unsafe.Offsetof(v.c), unsafe.Sizeof(v.c))
}
```

输出：

```
v: 0xc8200741c0 ~ c8200741d0, size: 16, align: 8

 field   address         offset    size
-------+---------------+---------+---------
 a       0xc8200741c0    0         0
```

```
b       0xc8200741c0    0         8
c       0xc8200741c8    8         0
```

如果仅有一个空结构字段，那么同样按 1 对齐，只不过长度为 0，且指向 runtime.zerobase 变量。

```
func main() {
    v := struct {
        a struct{}
    }{}

    fmt.Printf("%p, %d, %d\n", &v, unsafe.Sizeof(v), unsafe.Alignof(v))
}
```

输出：

```
0x19c730, 0, 1
```

对齐的原因与硬件平台，以及访问效率有关。某些平台只能访问特定地址，比如只能是偶数地址。而另一方面，CPU 访问自然对齐的数据可以用最少的读周期，还可避免拼接数据。

六. 方法

1. 定义

方法是与对象实例相绑定的特殊函数。

方法是面向对象编程的基本概念，用于维护和展示对象的自身状态。对象是内敛的，每个实例都有各自不同的独立特征，以属性和方法来暴露对外通讯接口。普通函数则专注于算法流程，通过接收参数来完成特定逻辑运算，并返回最终结果。换句话说，方法是有状态的，而函数通常没有。

方法和函数定义语法区别在于有个前置的实例接收参数（receiver），类似 Python self 的做法。在某些语言里，尽管没有显式定义，但也会隐式传递 this 参数。

可为当前包里，除接口和指针以外的任何类型定义方法。

```
type N int

func (n N) toString() string {
    return fmt.Sprintf("%#x", n)
}

func main() {
    var a N = 25
    println(a.toString())
}
```

输出：

```
0x19
```

方法同样不支持重载（overload），receiver 参数名没有限制，按惯例会选用一些简短有意义的名称（不推荐使用 this、self）。如果方法内部并不引用实例，甚至可以省略参数名，仅保留其类型。

```
type N int

func (N) test() {
    println("hi!")
```

```
}
```

既然方法是特殊的函数，那么 receiver 的类型自然可以是基础类型或指针类型。这就涉及到实例对象是否被复制。

```
type N int

func (n N) value() {                            // func value(n N)
    n++
    fmt.Printf("v: %p, %v\n", &n, n)
}

func (n *N) pointer() {                         // func pointer(n *N)
    (*n)++
    fmt.Printf("p: %p, %v\n", n, *n)
}

func main() {
    var a N = 25

    a.value()
    a.pointer()

    fmt.Printf("a: %p, %v\n", &a, a)
}
```

输出：

```
v: 0xc8200741c8, 26               // receiver 被复制。
p: 0xc8200741c0, 26
a: 0xc8200741c0, 26
```

可使用实例值或指针调用方法，编译器会自动根据方法 receiver 类型自动在基础类型和指针类型间转换。

```
func main() {
    var a N = 25
    p := &a

    a.value()
    a.pointer()

    p.value()
    p.pointer()
}
```

输出：

```
v: 0xc82000a2c0, 26
p: 0xc82000a298, 26

v: 0xc82000a2f0, 27
p: 0xc82000a298, 27
```

不能用多级指针调用方法。

```
func main() {
    var a N = 25

    p := &a
    p2 := &p

    p2.value()      // 错误: calling method value with receiver p2 (type **N)
                    //          requires explicit dereference

    p2.pointer()    // 错误: calling method pointer with receiver p2 (type **N)
                    //          requires explicit dereference
}
```

对于指针类型的 receiver 而言，必须是合法指针（包括 nil），或能获取实例地址。

```
type X struct{}

func (x *X) test() {
    println("hi!", x)
}

func main() {
    var a *X
    a.test()        // 相当于 test(nil)

    X{}.test()      // 错误: cannot take the address of X literal
}
```

将方法看做普通函数，就能很容易理解 receiver 的传参方式。

如何确定方法的 receiver 类型?

- 要修改实例状态，用 *T。
- 无需修改状态，小对象或固定值，建议用 T。

- 大对象建议用 *T，减少复制成本。
- 引用类型、字符串、函数等指针包装对象，直接用 T。
- 包含 Mutex 等同步字段，用 *T，避免复制造成锁操作无效。
- 实在搞不清，就都用 *T。

2. 匿名字段

可以像访问匿名字段成员那样调用其方法，由编译器负责查找。

```
type data struct {
    sync.Mutex
    buf [1024]byte
}

func main() {
    d := data{}
    d.Lock()               // 编译会处理为 sync.(*Mutex).Lock() 调用。
    defer d.Unlock()
}
```

方法也同样会有同名遮蔽的问题。但利用这种特性，可实现类似 override 的操作。

```
type user struct{}

type manager struct {
    user
}

func (user) toString() string {
    return "user"
}

func (m manager) toString() string {
    return m.user.toString() + "; manager"
}

func main() {
    var m manager

    println(m.toString())
    println(m.user.toString())
}
```

输出：

```
user; manager
user
```

尽管能直接访问匿名字段的成员和方法，但它们依然属于不同类型，并不存在继承关系。

3. 方法集

类型有一个与之相关的方法集合（method set），这决定了它是否实现某个接口。

- 类型 T 方法集包含所有 receiver T 方法。
- 类型 *T 方法集包含所有 receiver T + *T 方法。
- 匿名嵌入 S，T 方法集包含所有 receiver S 方法。
- 匿名嵌入 *S，T 方法集包含所有 receiver S + *S 方法。
- 匿名嵌入 S 或 *S，*T 方法集包含所有 receiver S + *S 方法。

可利用反射（reflect）测试这些规则。

```
type S struct{}

type T struct {
    S                                                   // 匿名嵌入字段。
}

func (S) sVal()  {}
func (*S) sPtr() {}
func (T) tVal()  {}
func (*T) tPtr() {}

func methodSet(a interface{}) {                         // 显示方法集里所有方法名字。
    t := reflect.TypeOf(a)
    for i, n := 0, t.NumMethod(); i < n; i++ {
        m := t.Method(i)
        fmt.Println(m.Name, m.Type)
    }
}

func main() {
    var t T

    methodSet(t)                                        // 显示 T 方法集。
    println("----------")
```

```
    methodSet(&t)                                       // 显示 *T 方法集。
}
```

输出：

```
sVal func(main.T)
tVal func(main.T)
----------------------
sPtr func(*main.T)
sVal func(*main.T)
tPtr func(*main.T)
tVal func(*main.T)
```

输出结果符合预期，但我们也注意到某些方法的 receiver 类型发生了改变。真实情况是，这些都是由编译器按方法集所需自动生成的额外包装方法。

```
$ nm test | grep "main\."

0000000000002040 t main.S.sVal
0000000000002050 t main.(*S).sPtr
00000000000023b0 t main.(*S).sVal

0000000000002570 t main.T.sVal
0000000000002060 t main.T.tVal

0000000000002490 t main.(*T).sPtr
0000000000002450 t main.(*T).sVal
0000000000002070 t main.(*T).tPtr
00000000000024d0 t main.(*T).tVal


$ go tool objdump -s "main\." test | grep "TEXT.*autogenerated"

TEXT main.(*S).sVal(SB) <autogenerated>

TEXT main.T.sVal(SB)    <autogenerated>
TEXT main.(*T).sVal(SB) <autogenerated>
TEXT main.(*T).sPtr(SB) <autogenerated>
TEXT main.(*T).tVal(SB) <autogenerated>
```

方法集仅影响接口实现和方法表达式转换，与通过实例或实例指针调用方法无关。实例并不使用方法集，而是直接调用（或通过隐式字段名）。

很显然，匿名字段就是为方法集准备的。否则，完全没必要为少写个字段名而大费周章。

面向对象的三大特征“封装”、“继承”和“多态”，Go 仅实现了部分特征，它更倾向于“组合优于继承”这种思想。将模块分解成相互独立的更小单元，分别处理不同方面的需求，

最后以匿名嵌入方式组合到一起，共同实现对外接口。且简短一致的调用方式，更是隐藏了内部实现细节。

组合没有父子依赖，不会破坏封装。且整体和局部松耦合，可任意增加来实现扩展。各单元持有单一职责，互无关联，实现和维护更加简单。

尽管接口也是多态的一种实现形式，但我认为应该和基于继承体系的多态分离开来。

---

4. 表达式

方法和函数一样，除直接调用外，还可赋值给变量，或作为参数传递。依照具体引用方式的不同，可分为 expression 和 value 两种状态。

Method Expression

通过类型引用的 method expression 会被还原普通函数样式，receiver 是第一参数，调用时须显式传递。至于类型，可以是 T 或 *T，只要目标方法存在于该类型方法集中即可。

```
type N int

func (n N) test() {
    fmt.Printf("test.n: %p, %d\n", &n, n)
}

func main() {
    var n N = 25
    fmt.Printf("main.n: %p, %d\n", &n, n)

    f1 := N.test                                    // func(n N)
    f1(n)

    f2 := (*N).test                                 // func(n *N)
    f2(&n)                                          // 按方法集中的签名传递正确类型的参数。
}
```

输出：

```
main.n: 0xc82000a140, 25
test.n: 0xc82000a158, 25
test.n: 0xc82000a168, 25
```

> 尽管 *N 方法集包装的 test 方法 receiver 类型不同，但编译器会保证按原定义类型拷贝传值。

当然，也可直接以表达式方式调用。

```
func main() {
    var n N = 25

    N.test(n)
    (*N).test(&n)          // 注意：*N 须使用括号，以免语法解析错误。
}
```

Method Value

基于实例或指针引用的 method value，参数签名不会改变，依旧按正常方式调用。

但当 method value 被赋值给变量或作为参数传递时，会立即计算并复制该方法执行所需的 receiver 对象。与其绑定，以便在稍后执行时，能隐式传入 receiver 参数。

```
type N int

func (n N) test() {
    fmt.Printf("test.n: %p, %v\n", &n, n)
}

func main() {
    var n N = 100
    p := &n

    n++
    f1 := n.test                                    // 因为 test 方法的 receiver 是 N 类型，
                                                    //   所以复制 n，等于 101。
    n++
    f2 := p.test                                    // 复制 *p，等于 102。

    n++
    fmt.Printf("main.n: %p, %v\n", p, n)

    f1()
    f2()
}
```

输出：

```
main.n: 0xc820076028, 103
test.n: 0xc820076060, 101
test.n: 0xc820076070, 102
```

编译器会为 method value 生成一个包装函数，实现间接调用。至于 receiver 复制，和闭包的实现方法基本相同，打包成 funcval，经由 rdx 寄存器传递。

作为函数实参，会进行同样的复制操作。

```
func call(m func()) {
    m()
}

func main() {
    var n N = 100
    p := &n

    fmt.Printf("main.n: %p, %v\n", p, n)

    n++
    call(n.test)

    n++
    call(p.test)
}
```

输出：

```
main.n: 0xc82000a288, 100
test.n: 0xc82000a2c0, 101
test.n: 0xc82000a2d0, 102
```

当然，如果目标方法的 receiver 是指针类型，那么被复制的仅是指针。

```
type N int

func (n *N) test() {
    fmt.Printf("test.n: %p, %v\n", n, *n)
}

func main() {
    var n N = 100
    p := &n

    n++
```

```
    f1 := n.test                                    // 因为 test 方法的 receiver 是 *N 类型，
                                                    //   所以复制 &n。
    n++
    f2 := p.test                                    // 复制 p 指针。

    n++
    fmt.Printf("main.n: %p, %v\n", p, n)

    f1()                                            // 延迟调用，n == 103。
    f2()
}
```

输出：

```
main.n: 0xc82000a298, 103
test.n: 0xc82000a298, 103
test.n: 0xc82000a298, 103
```

只要能正确处理好 receiver 参数，使用 nil 也无可厚非。

```
type N int

func (N) value()    {}
func (*N) pointer() {}

func main() {
    var p *N

    p.pointer()            // method value
    (*N)(nil).pointer()    // method value
    (*N).pointer(nil)      // method expression

    // p.value()           // 错误: invalid memory address or nil pointer dereference
}
```

七. 接口

1. 定义

接口代表一种调用契约，是多个方法声明的集合。

在某些动态语言里，接口（interface）也被称作协议（protocol）。准备交互的双方，共同遵守事先约定的规则，使得在无需知道对方身份的情况下，进行协作。接口要实现的是做什么，而不关心怎么做谁来做。

接口解除了类型依赖，有助于减少用户可视方法，屏蔽内部结构和实现细节。似乎好处很多，但这并不意味着可以滥用接口，毕竟接口实现机制会有运行期开销。对于相同包，或者不会频繁变化的内部模块之间，并不需要抽象出接口来强行分离。接口最常见使用场景，是对包外提供访问，或预留扩展空间。

Go 接口实现机制很简洁，只要目标类型方法集内包含接口声明的全部方法就被视为实现了该接口，无需做显示声明。当然，目标类型可实现多个接口。

> 换句话说，我们可以先实现类型，而后再抽象出所需接口。这种非侵入式设计有很多好处，举例来说：在项目前期就设计出最合理接口并不容易，而在代码重构，模块分拆时再分离出接口，用以解耦就很常见。另外，在使用第三方库时，抽象出所需接口，即可屏蔽太多不需要关注的内容，也便于日后替换。

从内部实现来看，接口自身也是一种结构类型，只是编译器会对其做出很多限制。

```
type iface struct {
    tab  *itab
    data unsafe.Pointer
}
```

- 不能有字段。
- 不能定义自己的方法。
- 只能声明方法，不能实现。
- 可嵌入其他接口类型。

接口习惯以 er 作为名称后缀，方法名是声明组成部分，但参数名可不同或省略。

```
type tester interface {
    test()
    string() string
}

type data struct{}

func (*data) test() {}
func (data) string() string {
    return ""
}

func main() {
    var d data

    // var t tester = d           // 错误: data does not implement tester
                                  //       (test method has pointer receiver)

    var t tester = &d
    t.test()
    println(t.string())
}
```

编译器根据方法集来判断是否实现了接口，显然在上例中只有 *data 才复合 tester 的要求。

如果接口没有任何方法声明，那么就是一个空接口（interface{}），它的用途有些像面向对象里的根类型 Object，可被赋值为任何类型的对象。

接口变量默认值是 nil。如果实现接口的类型支持，可做相等运算。

```
func main() {
    var t1, t2 interface{}
    println(t1 == nil, t1 == t2)

    t1, t2 = 100, 100
    println(t1 == t2)

    t1, t2 = map[string]int{}, map[string]int{}
    println(t1 == t2)
}
```

输出：

```
true true
true
```

```
panic: runtime error: comparing uncomparable type map[string]int
```

可像匿名字段那样，嵌入其他接口。目标类型方法集中必须拥有包含嵌入接口方法在内的全部方法才算实现了该接口。

嵌入其他接口类型，相当于将其声明的方法集导入。这就要求不能有同名方法，因为不支持重载。还有，不能嵌入自身或循环嵌入，那会导致递归错误。

```
type stringer interface {
    string() string
}

type tester interface {
    stringer                        // 嵌入其他接口。
    test()
}

type data struct{}

func (*data) test() {}
func (data) string() string {
    return ""
}

func main() {
    var d data

    var t tester = &d
    t.test()
    println(t.string())
}
```

超集接口变量可隐式转换为子集，反过来不行。

```
func pp(a stringer) {
    println(a.string())
}

func main() {
    var d data

    var t tester = &d
    pp(t)                           // 隐式转换为子集接口。

    var s stringer = t              // 超级转换为子集。
    println(s.string())
```

```
    // var t2 tester = s          // 错误: stringer does not implement tester
                                  //       (missing test method)
}
```

支持匿名接口类型，可直接用于变量定义，或作为结构字段类型。

```
type data struct{}

func (data) string() string {
    return ""
}

type node struct {
    data interface {              // 匿名接口类型。
        string() string
    }
}

func main() {
    var t interface {             // 定义匿名接口变量。
        string() string
    } = data{}

    n := node{
        data: t,
    }

    println(n.data.string())
}
```

2. 执行机制

接口使用一个名为 itab 的结构存储运行期所需的相关类型信息。

```
type iface struct {
    tab  *itab                    # 类型信息。
    data unsafe.Pointer           # 实际对象指针。
}

type itab struct {
    inter  *interfacetype         # 接口类型。
    _type  *_type                 # 实际对象类型。
    fun    [1]uintptr             # 实际对象方法地址。
```

```
}
```

利用调试器，我们可查看这些结构存储的具体内容。

```
type Ner interface {
    a()
    b(int)
    c(string) string
}

type N int
func (N) a() {}
func (*N) b(int) {}
func (*N) c(string) string { return "" }

func main() {
    var n N
    var t Ner = &n

    t.a()
}
```

输出：

```
$ go build -gcflags "-N -l"

$ gdb test

...

(gdb) info locals                          # 设置断点，运行，查看局部变量信息。
&n = 0xc82000a130
t = {
  tab = 0x12f028,
  data = 0xc82000a130
}

(gdb) p *t.tab.inter.typ._string           # 接口类型名称。
$17 = 0x737f0 "main.Ner"

(gdb) p *t.tab._type._string               # 实际对象类型。
$20 = 0x707a0 "*main.N"

(gdb) p t.tab.inter.mhdr                   # 接口类型方法集。
$27 = {
  array = 0x60158 <type.*+72888>,
  len = 3,
  cap = 3
}

(gdb) p *t.tab.inter.mhdr.array[0].name    # 接口方法名称。
```

```
$30 = 0x70a48 "a"

(gdb) p *t.tab.inter.mhdr.array[1].name
$31 = 0x70b08 "b"

(gdb) p *t.tab.inter.mhdr.array[2].name
$32 = 0x70ba0 "c"

(gdb) info symbol t.tab.fun[0]            # 实际对象方法地址。
main.(*N).a in section .text

(gdb) info symbol t.tab.fun[1]
main.(*N).b in section .text

(gdb) info symbol t.tab.fun[2]
main.(*N).c in section .text
```

很显然，相关类型信息里保存了接口和实际对象的元数据。同时，itab 还用 fun 数组（不定长结构）保存了实际对象方法地址，从而实现在运行期对目标方法的动态调用。

除此之外，接口还有一个重要特征：将对象赋值给接口变量时，会复制该对象。

```
type data struct {
    x int
}

func main() {
    d := data{100}
    var t interface{} = d

    println(t.(data).x)
}
```

输出：

```
$ go build -gcflags "-N -l"

$ gdb test

...

(gdb) info locals                         # 输出局部变量。
d = {
  x = 100
}
t = {
  _type = 0x5ec00 <type.*+67296>,
  data = 0xc820035f20                     # 接口变量存储的对象地址。
}
```

```
(gdb) p/x &d                                # 局部变量地址。显然和接口存储的不是同一对象。
$1 = 0xc820035f10
```

我们甚至无法修改接口存储的复制品。

```
func main() {
    d := data{100}
    var t interface{} = d

    p := &t.(data)         // 错误: cannot take the address of t.(data)
    t.(data).x = 200       // 错误: cannot assign to t.(data).x
}
```

即便将其复制出来，用本地变量修改后，依然无法对 iface.data 赋值。解决方法就是将真实对象的指针赋值给接口，那么接口内存储的就是指针的复制品。

```
func main() {
    d := data{100}
    var t interface{} = &d

    t.(*data).x = 200
    println(t.(*data).x)
}
```

输出：

```
$ go build -gcflags "-N -l" && ./test

200

$ gdb test

...

(gdb) info locals                           # 显示局部变量。
d = {
  x = 100
}
t = {
  _type = 0x50480 <type.*+8096>,
  data = 0xc820035f10
}

(gdb) p/x &d                                # 显然和接口内 data 存储的地址一致。
$1 = 0xc820035f10
```

只有当接口变量内部的两个指针（itab, data）都为 nil 时，接口才等于 nil。

```
func main() {
    var a interface{} = nil
    var b interface{} = (*int)(nil)

    println(a == nil, b == nil)
}
```

输出：

```
true false

(gdb) info locals

b = {
  _type = 0x500c0 <type.*+7616>,          # 显然 b 包含了类型信息。
  data = 0x0
}
a = {
  _type = 0x0,
  data = 0x0
}
```

由此造成的错误并不罕见，尤其是在函数返回 error 时。

```
type MyError struct {                           # 自定义错误类型。
    s string
}

func (e *MyError) Error() string {
    return e.s
}

func test(x int) (int, error) {
    var err *MyError

    if x < 0 {
        err = &MyError{s: "invalid number"}
        x = 0
    } else {
        x += 100
    }

    return x, err                               # 注意: 这个 err 是有类型的。
}

func main() {
    x, err := test(100)
    if err != nil {
```

```
        log.Fatalln(err)
    }

    println(x)
}
```

输出：

```
2020/01/01 19:48:27 <nil>
exit status 1

(gdb) info locals                  # 很显然 x 没问题，但 err 并不等于 nil。
x = 200
err = {
  tab = 0x2161e8,
  data = 0x0
}
```

正确做法是明确返回 nil。

```
func test(x int) (int, error) {
    if x < 0 {
        return 0, &MyError{s: "invalid number"}
    }

    return x + 100, nil
}

func main() {
    x, err := test(100)
    if err != nil {
        log.Fatalln(err)
    }

    println(x)
}
```

输出：

```
200
```

3. 类型转换

类型推断可将接口变量转换回原始类型，或用来判断是否实现了某个具体接口类型。

```
type data int

func (d data) String() string {
    return fmt.Sprintf("data:%d", d)
}

func main() {
    var d data = 15
    var x interface{} = d

    if n, ok := x.(fmt.Stringer); ok {    // 转换为更具体的接口类型。
        fmt.Println(n)
    }

    if d2, ok := x.(data); ok {           // 转换回原始类型。
        fmt.Println(d2)
    }

    e := x.(error)                        // 错误: main.data is not error
    fmt.Println(e)
}
```

输出：

```
data:15
data:15
panic: interface conversion: main.data is not error: missing method Error
```

使用 ok-idiom 模式，即便转换失败也不会引发 panic 错误。还支持用 switch 语句在多种类型间做出推断匹配，这样 interface{} 就有更多发挥空间。

```
func main() {
    var x interface{} = func(x int) string {
        return fmt.Sprintf("d:%d", x)
    }

    switch v := x.(type) {                // 局部变量 v 是类型转换后的结果。
    case nil:
        println("nil")
    case *int:
        println(*v)
    case func(int) string:
        println(v(100))
    case fmt.Stringer:
        fmt.Println(v)
    default:
        println("unknown")
    }
}
```

输出：

```
d:100
```

> 提示：type switch 不支持 fallthrought。

4. 技巧

让编译器检查，确保类型实现了指定接口。

```
type x int

func init() {                       // 包初始化函数。
    var _ fmt.Stringer = x(0)
}
```

输出：

```
cannot use x(0) (type x) as type fmt.Stringer in assignment:
    x does not implement fmt.Stringer (missing String method)
```

定义一个通用函数类型，让相同签名的函数自动实现某个接口。

```
type FuncString func() string

func (f FuncString) String() string {
    return f()
}

func main() {
    var t fmt.Stringer = FuncString(func() string {      // 转换类型，使其实现 Stringer 接口。
        return "hello, world!"
    })

    fmt.Println(t)
}
```

八. 并发

1. 并发

在开始之前，需要了解并发（concurrency）和并行（parallesim）的区别。

- 并发：逻辑上具备处理多个同时性任务的能力。
- 并行：物理上同一时刻执行多个并发任务。

我们通常会说程序是并发设计的，也就是说它允许多个任务同时执行，但实际上并不一定真在同一时刻发生。在单核处理器上，它们能以间隔方式切换执行。而并行则依赖多核处理器等物理设备，让多个任务真正在同一时刻执行，它代表了当前程序运行状态。简单点说，并行是并发设计的理想执行模式。

Concurrency is not parallelism : Different concurrent designs enable different ways to parallelize.

多线程或多进程是并行的基本条件，但单线程也可用协程（coroutine）做到并发。尽管协程在单个线程上通过主动切换来实现多任务并发执行，但它也有自己的优势。除将因阻塞而浪费的时间找回来外，还免除了线程切换开销，有着不错的执行效率。协程上运行的多个任务本质上是依旧串行的，加上可控自主调度，所以并不需要做同步处理。

即便采用多线程也未必就能并行。Python 就因 GIL 限制，默认只能并发而不能并行。大多时候采用“多进程 + 协程”架构模型。

很难说哪种方式更好一些，它们有各自适用的场景。通常情况下，用多进程来实现分布式和负载平衡，减轻单进程垃圾回收压力；用多线程（LWP）抢夺更多的处理器资源；用协程来提高处理器时间片利用率。

简单将 goroutine 归纳为协程并不合适。运行时会创建多个线程来执行并发任务，且任务单元可被调度到其他线程并行执行。这更像是多线程和协程的综合体，能最大限度提升执行效率，发挥多核处理能力。

更多实现细节，请阅读本书下册《源码剖析》。

只需在函数调用前添加 go 关键字即可创建并发任务。

```
go println("hello, world!")

go func(s string) {
    println(s)
}("hello, world!")
```

注意是函数调用，所以必须提供相应的参数。

关键字 go 并非执行并发操作，而是创建一个并发任务单元。新建任务被放置在系统队列中，等待调度器安排合适系统线程去获取执行。当前流程不会阻塞，不会等待该任务启动，且运行时也不保证并发任务执行次序。

每个任务单元除保存函数指针、调用参数外，还会分配执行所需的栈内存空间。相比系统默认 MB 级别的线程栈，goroutine 自定义栈初始仅需 2 KB，所以才能创建成千上万的并发任务。自定义栈采取按需分配策略，在需要时进行扩容，最大能到 GB 规模。

在不同版本中，自定义栈大小略有不同。如未做说明，本书特指 1.6 amd64。

与 defer 一样，goroutine 也会因“延迟执行”而立即计算并复制执行参数。

```
var c int

func counter() int {
    c++
    return c
}

func main() {
    a := 100

    go func(x, y int) {
        time.Sleep(time.Second)           // 让 goroutine 在 main 逻辑之后执行。
        println("go:", x, y)
    }(a, counter())                       // 立即计算并复制参数。

    a += 100
    println("main:", a, counter())

    time.Sleep(time.Second * 3)           // 等待 goroutine 结束。
```

```
}
```

输出：

```
main: 200 2
go: 100 1
```

Wait

进程退出时不会等待并发任务结束。可用通道（channel）阻塞，然后发出退出信号。

```
func main() {
    exit := make(chan struct{})           // 创建通道，因为仅是通知，数据并没有实际意义。

    go func() {
        time.Sleep(time.Second)
        println("goroutine done.")

        close(exit)                       // 关闭通道，发出信号。
    }()

    println("main ...")
    <-exit                                // 如通道关闭，立即解除阻塞。
    println("main exit.")
}
```

输出：

```
main ...
goroutine done.
main exit.
```

> 除关闭通道外，写入数据也可解除阻塞。channel 的更多信息，后节再做详述。

如要等待多个任务结束，推荐使用 sync.WaitGroup。通过设定计数器，让每个 goroutine 在退出前递减，直至归零时解除阻塞。

```
import (
    "sync"
    "time"
)

func main() {
```

```
    var wg sync.WaitGroup

    for i := 0; i < 10; i++ {
        wg.Add(1)                               // 累加计数。

        go func(id int) {
            defer wg.Done()                     // 递减计数。

            time.Sleep(time.Second)
            println("goroutine", id, "done.")
        }(i)
    }

    println("main ...")
    wg.Wait()                                   // 阻塞，直到计数归零。
    println("main exit.")
}
```

输出：

```
main ...
goroutine 9 done.
goroutine 4 done.
goroutine 2 done.
goroutine 6 done.
goroutine 8 done.
goroutine 3 done.
goroutine 5 done.
goroutine 1 done.
goroutine 0 done.
goroutine 7 done.
main exit.
```

可在多处使用 Wait 阻塞，它们都能接收到通知。

```
func main() {
    var wg sync.WaitGroup
    wg.Add(1)

    go func() {
        wg.Wait()                       // 等待归零，解除阻塞。
        println("wait exit.")
    }()

    go func() {
        time.Sleep(time.Second)
        println("done.")
        wg.Done()                       // 递减计数。
    }()

    wg.Wait()                           // 等待归零，解除阻塞。
```

```
    println("main exit.")
}
```

输出：

```
done.
wait exit.
main exit.
```

GOMAXPROCS

运行时可能会创建很多线程，但任何时候仅有限的几个参与并发任务执行。该数量默认与处理器核数相等，可用 runtime.GOMAXPROCS 函数（或环境变量）修改。

> 如调用参数小于 1，仅返回当前设置值，不做任何调整。

```
import (
    "math"
    "runtime"
    "sync"
)

// 测试目标函数。
func count() {
    x := 0
    for i := 0; i < math.MaxUint32; i++ {
        x += i
    }

    println(x)
}

// 循环执行。
func test(n int) {
    for i := 0; i < n; i++ {
        count()
    }
}

// 并发执行。
func test2(n int) {
    var wg sync.WaitGroup
    wg.Add(n)

    for i := 0; i < n; i++ {
        go func() {
            count()
```

```
            wg.Done()
        }()
    }

    wg.Wait()
}

func main() {
    n := runtime.GOMAXPROCS(0)
    test(n)
    // test2(n)
}
```

输出：

```
$ time ./test

9223372030412324865
9223372030412324865
9223372030412324865
9223372030412324865

real    0m8.395s
user    0m8.281s
sys     0m0.056s


$ time ./test2

9223372030412324865
9223372030412324865
9223372030412324865
9223372030412324865

real    0m3.907s            // 程序实际执行时间。
user    0m14.438s           // 多核执行时间累加。
sys     0m0.041s
```

该测试机器是 4 核，可用 runtime.NumCPU 函数返回。

Local Storage

与线程不同，goroutine 任务无法设置优先级，无法获取编号，没有局部存储（TLS），甚至连返回值都会被抛弃。但除优先级外，其他功能都很容易实现。

```
func main() {
    var wg sync.WaitGroup
```

```
    var gs [5]struct {                              // 用于实现类似 TLS 功能。
        id     int                                  // 编号。
        result int                                  // 返回值。
    }

    for i := 0; i < len(gs); i++ {
        wg.Add(1)

        go func(id int) {                           // 使用参数避免闭包延迟求值。
            defer wg.Done()

            gs[id].id = id
            gs[id].result = (id + 1) * 100
        }(i)
    }

    wg.Wait()
    fmt.Printf("%+v\n", gs)
}
```

输出：

```
{id:0 result:100} {id:1 result:200} {id:2 result:300} {id:3 result:400} {id:4 result:500}
```

如使用 map 作为局部存储容器，建议做同步处理，因为运行时会对其做并发读写检查。
虽然可用汇编代码获取 goroutine id，但这并没有实际意义，因为要考虑 G 缓存复用的问题。

Gosched

暂停，释放线程去执行其他任务。当前任务被放回队列，等待下次调度时恢复执行。

```
func main() {
    runtime.GOMAXPROCS(1)
    exit := make(chan struct{})

    go func() {
        go func() {                         // 确保 a 优先执行。
            println("b")
        }()

        for i := 0; i < 5; i++ {
            println("a:", i)

            if i == 2 {                     // 让出当前线程，调度执行 b。
                runtime.Gosched()
            }
        }
```

```
        close(exit)
    }()

    <-exit
}
```

输出：

```
a: 0
a: 1
a: 2
b
a: 3
a: 4
```

该函数很少被使用，因为运行时会主动向长时间运行（10 ms）的任务发出抢占调度。只是当前版本实现的算法稍显粗糙，不能保证调度总能成功，所以主动切换还有适用场合。

Goexit

立即终止当前任务，运行时确保所有已注册延迟调用被执行。该函数不会影响其他并发任务，不会引发 panic，自然也就无法捕获。

```
func main() {
    exit := make(chan struct{})

    go func() {
        defer close(exit)
        defer println("a")

        func() {
            defer func() {
                println("b", recover() == nil)
            }()

            func() {                                    // 多层调用。
                println("c")
                runtime.Goexit()                        // 立即终止整个调用堆栈。
                println("c done.")                      // 不会执行。
            }()

            println("b done.")                          // 不会执行。
        }()

        println("a done.")                              // 不会执行。
    }()
```

```
    <-exit
    println("main exit.")
}
```

输出：

```
c
b true
a
main exit.
```

如果在 main.main 里调用 Goexit，它会等待其他任务结束，然后让进程直接崩溃。

```
func main() {
    for i := 0; i < 3; i++ {                    // Goexit 会等待所有任务结束。
        go func(x int) {
            for n := 0; n < 2; n++ {
                fmt.Printf("%c: %d\n", 'a'+x, n)
                time.Sleep(time.Millisecond)
            }
        }(i)
    }

    runtime.Goexit()
    println("main exit.")
}
```

输出：

```
c: 0
a: 0
b: 0
b: 1
a: 1
c: 1
fatal error: no goroutines (main called runtime.Goexit) - deadlock!
```

无论身处哪一层，Goexit 都能立即终止整个调用堆栈，这与 return 仅退出当前函数不同。

---

2. 通道

相比 Erlang，Go 并未实现严格的并发安全。

允许全局变量、指针、引用类型这些非安全内存共享操作，就需要开发人员自行维护数据一致和完整性。Go 鼓励使用 CSP 通道，以通讯来代替内存共享，实现并发安全。

> Don't communicate by sharing memory, share memory by communicating.
> CSP: Communicating Sequential Process.

通过消息来避免竞态的模型除了 CSP，还有 Actor。但两者有较大区别。

作为 CSP 核心，channel 是显式的，要求操作双方必须知道数据类型和具体通道。但并不关心另一端操作者身份和数量。尽管通道实现了缓冲异步，但究其根本依然是同步操作的。通道两端都会关心消息处理情况，未能及时处理时，阻塞操作端。相比起来，Actor 里的 mailbox 是透明的，它不在乎数据类型及通道，只要知道接收者信箱即可。默认就是异步方式，发送方对消息是否被接收和处理并不关心。

从底层实现上来说，通道只是一个队列。同步模式下，发送和接收双方配对，然后直接复制数据给对方。如配对失败，则置入等待队列，直到另一方出现后才被唤醒。异步模式抢夺的则是数据缓冲槽。发送方要求有空槽可供写入，而接收方则要求有缓冲数据可读。需求不符时，同样加入等待队列，直到有另一方写入数据或腾出空槽后被唤醒。

除传递消息（数据）外，通道还常被用作事件通知。

```
func main() {
    done := make(chan struct{})
    c := make(chan string)

    go func() {
        println(<-c)                    // 接收消息。
        close(done)                     // 发送接收结束通知。
    }()

    c <- "hi!"                          // 发送消息。
    <-done                              // 阻塞，等待结束通知。
}
```

输出：

```
hi!
```

同步模式必须有配对操作的 goroutine 出现，否则会一直阻塞。而异步模式在缓冲区未满，或数据未读完前，不会阻塞。

```
func main() {
    c := make(chan int, 3)          // 创建带 3 个缓冲槽的通道。

    c <- 1                          // 缓冲区未满，不会阻塞。
    c <- 2

    println(<-c)                    // 缓冲区尚有数据，不会阻塞。
    println(<-c)
}
```

输出：

```
1
2
```

> 多数时候，异步通道有助于提升性能，减少排队阻塞。

缓冲区大小仅是内部属性，不属于类型组成部分。另外 channel 变量自身就是指针，可用相等操作符判断是否为同一对象或 nil。

```
func main() {
    var a, b chan int = make(chan int, 3), make(chan int)
    var c chan bool

    println(a == b)
    println(c == nil)

    fmt.Printf("%p, %d\n", a, unsafe.Sizeof(a))
}
```

输出：

```
false
true
0xc820076000, 8
```

> 虽然可传递指针来避免数据复制，但须额外注意数据并发安全。

内置函数 cap 和 len 返回缓冲区大小和当前已缓冲数量。而对于同步通道则都返回 0，据此可判断通道是同步还是异步。

```
func main() {
    a, b := make(chan int), make(chan int, 3)

    b <- 1
    b <- 2

    println("a:", len(a), cap(a))
    println("b:", len(b), cap(b))
}
```

输出：

```
a: 0 0
b: 2 3
```

Goroutine Leak 是指 goroutine 处于发送或接收阻塞状态，但一直无合作方响应。垃圾回收器不会收集此类资源，导致它们会在 channel 等待队列里持久休眠。

收发

除使用简单的发送和接收操作符外，还可用 ok-idom 或 range 模式处理数据。

```
func main() {
    done := make(chan struct{})
    c := make(chan int)

    go func() {
        for {
            x, ok := <-c

            if !ok {                        // 据此判断通道是否被关闭。
                close(done)
                return
            }

            println(x)
        }
    }()

    c <- 1
    c <- 2
    c <- 3
```

```
    close(c)
    <-done
}
```

输出：

```
1
2
3
```

对于循环读取数据，range 模式更简洁一些。但 ok-idiom 可用来判断通道是否已被关闭。

```
func main() {
    done := make(chan struct{})
    c := make(chan int)

    go func() {
        for x := range c {                // 循环，直到通道被关闭。
            println(x)
        }

        close(done)
    }()

    c <- 1
    c <- 2
    c <- 3

    close(c)
    <-done
}
```

及时用 close 函数关闭通道引发结束通知，否则可能会导致死锁。

```
fatal error: all goroutines are asleep - deadlock!
```

通知可以是群体性的。也未必就是结束，可以是任何需要表达的事件。

```
func main() {
    var wg sync.WaitGroup
    ready := make(chan struct{})

    for i := 0; i < 3; i++ {
        wg.Add(1)
```

```
        go func(id int) {
            println(id, ": i am ready.")          // 运动员准备就绪。
            <-ready                               // 等待发令。
            println(id, ": running...")

            wg.Done()
        }(i)
    }

    time.Sleep(time.Second)
    println("Ready? Go!")

    close(ready)                                  // 砰！小的们，都给爷跑起来！
    wg.Wait()
}
```

输出：

```
0 : i am ready.
2 : i am ready.
1 : i am ready.

Ready? Go!

1 : running...
0 : running...
2 : running...
```

一次性事件使用 close 效率更好，没有多余开销。至于连续或多样性事件，可传递不同数据标志来实现。

对于 closed 或 nil 通道，发送和接收操作都有相应规则：

- 向 closed channel 发送数据，引发 panic。
- 从 closed channel 接收数据，返回已缓冲数据或零值。
- 无论收发，nil channel 都会阻塞。

```
func main() {
    c := make(chan int, 3)

    c <- 10
    c <- 20
    close(c)

    for i := 0; i < cap(c)+1; i++ {
        x, ok := <-c
```

```
        println(i, ":", ok, x)
    }
}
```

输出：

```
0 : true 10
1 : true 20
2 : false 0
3 : false 0
```

重复关闭，或关闭 nil 通道都会引发 panic 错误。

```
panic: close of closed channel
panic: close of nil channel
```

单向

通道默认是双向的，并不区分发送和接收端。但某些时候，我们可限制收发操作方向来获得更严谨的操作逻辑。

尽管可用 make 创建单向通道，但那没有任何意义。通常使用类型转换来获取单向通道，并分别赋予操作双方。

```
func main() {
    var wg sync.WaitGroup
    wg.Add(2)

    c := make(chan int)

    var send chan<- int = c
    var recv <-chan int = c

    go func() {
        defer wg.Done()

        for x := range recv {
            println(x)
        }
    }()

    go func() {
        defer wg.Done()
        defer close(c)
```

```
        for i := 0; i < 3; i++ {
            send <- i
        }
    }()

    wg.Wait()
}
```

不能在单向通道上做逆向操作。

```
func main() {
    c := make(chan int, 2)

    var send chan<- int = c
    var recv <-chan int = c

    <-send                  // 无效操作: <-send (receive from send-only type chan<- int)
    recv <- 1               // 无效操作: recv <- 1 (send to receive-only type <-chan int)
}
```

同样，close 不能用于接收端。

```
func main() {
    c := make(chan int, 2)
    var recv <-chan int = c

    close(recv)             // 无效操作: close(recv) (cannot close receive-only channel)
}
```

无法将单向通道重新转换回去。

```
func main() {
    var a, b chan int

    a = make(chan int, 2)
    var recv <-chan int = a
    var send chan<- int = a

    b = (chan int)(recv)    // 错误: cannot convert recv (type <-chan int) to type chan int
    b = (chan int)(send)    // 错误: cannot convert send (type chan<- int) to type chan int
}
```

选择

如要同时处理多个通道，可选用 select 语句。它会随机选择一个可用通道做收发操作。

```
func main() {
    done := make(chan struct{})
    a, b := make(chan int), make(chan int)

    go func() {                          // 接收
        defer close(done)

        for {
            select {
            case v, ok := <-a:           // ok-idiom 模式。
                if !ok {                 // 如果该通道被关闭，将其置为 nil，永远阻塞，
                    a = nil              //   避免该 case 再次命中返回零值。
                    break
                }

                println("a:", v)
            case v, ok := <-b:
                if !ok {
                    b = nil
                    break
                }

                println("b:", v)
            }

            if a == nil && b == nil {    // 等待所有通道处理完毕后，退出。
                return
            }
        }
    }()

    go func() {                          // 发送
        defer close(a)
        defer close(b)

        for i := 0; i < 10; i++ {
            select {
            case a <- i:
            case b <- i + 10:
            }
        }
    }()

    <-done
}
```

输出：

```
b: 10
a: 1
a: 2
b: 13
a: 4
a: 5
b: 16
a: 7
b: 18
b: 19
```

即便是同一 channel，也会随机选择 case 执行。

```
func main() {
    done := make(chan struct{})
    c := make(chan int)

    go func() {                         // 接收。
        for {
            var v int
            var ok bool

            select {                    // 随机选择 case。
            case v, ok = <-c:
                println("a1:", v)
            case v, ok = <-c:
                println("a2:", v)
            }

            if !ok {
                close(done)
                return
            }
        }
    }()

    go func() {                         // 发送。
        defer close(c)

        for i := 0; i < 10; i++ {
            select {                    // 随机选择 case。
            case c <- 1:
            case c <- 0:
            }
        }
    }()

    <-done
}
```

输出：

```
a1: 1
a2: 0
a2: 0
a2: 1
a1: 1
a1: 1
a2: 1
a1: 0
a2: 1
a2: 1
a2: 0
```

当所有通道都不可用时，select 会执行 default 语句。如此可避开 select 阻塞，但须注意处理外层循环，以免陷入空耗。

```
func main() {
    over := make(chan struct{})

    go func() {
        for {
            select {
            case <-over:                          // 接收结束通知。
                over <- struct{}{}
                return
            default:                              // 避免 select 阻塞。
            }

            fmt.Println(time.Now())               // 执行正常逻辑。
            time.Sleep(time.Second)
        }
    }()

    time.Sleep(time.Second * 5)
    over <- struct{}{}                            // 发出结束通知。

    <-over
}
```

本例没有使用 close，而是采取双向通知的做法。

也可用来处理一些默认逻辑。

```
func main() {
    done := make(chan struct{})
```

```
    data := []chan int{                                    // 数据缓冲区。
        make(chan int, 3),
    }

    go func() {
        defer close(done)

        for i := 0; i < 10; i++ {
            select {
            case data[len(data)-1] <- i:                   // 生产数据。
            default:                                       // 当前通道已满，生成新的缓存通道。
                data = append(data, make(chan int, 3))
            }
        }
    }()

    <-done

    for i := 0; i < len(data); i++ {                       // 显示所有数据。
        c := data[i]
        close(c)

        for x := range c {
            println(x)
        }
    }
}
```

模式

通常使用工厂方法将 goroutine 和 channel 绑定。

```
type receiver struct {
    sync.WaitGroup
    data chan int
}

func newReceiver() *receiver {
    r := &receiver{
        data: make(chan int),
    }

    go func() {
        r.Add(1)
        defer r.Done()

        for x := range r.data {            // 处理消息，直到通道被关闭。
            println("recv:", x)
        }
```

```
    }()

    return r
}

func main() {
    r := newReceiver()
    r.data <- 1
    r.data <- 2

    close(r.data)                          // 关闭通道，发出结束通知。
    r.Wait()                               // 等待接收者处理结束。
}
```

输出：

```
recv: 1
recv: 2
```

用 channel 实现信号量（semaphore）。

```
func main() {
    runtime.GOMAXPROCS(4)
    var wg sync.WaitGroup

    sem := make(chan struct{}, 2)

    for i := 0; i < 5; i++ {
        wg.Add(1)

        go func(id int) {
            defer wg.Done()

            sem <- struct{}{}                  // acquire: 获取信号。

            time.Sleep(time.Second * 2)
            fmt.Println(id, time.Now())

            <-sem                              // release: 释放信号。
        }(i)
    }

    wg.Wait()
}
```

输出：

```
4 2016-02-19 18:24:09
0 2016-02-19 18:24:09
2 2016-02-19 18:24:11
1 2016-02-19 18:24:11
```

```
3 2016-02-19 18:24:13
```

标准库 time 提供了 timeout 和 tick channel 实现。

```
func main() {
    go func() {
        for {
            select {
            case <-time.After(time.Second * 5):
                println("timeout ...")
                os.Exit(0)
            }
        }
    }()

    go func() {
        tick := time.Tick(time.Second)

        for {
            select {
            case <-tick:
                fmt.Println(time.Now())
            }
        }
    }()

    var exit chan struct{}
    <-exit                                  // 直接用 nil channel 阻塞进程。
}
```

捕获 INT、TERM 信号，顺便实现一个简易的 atexit 函数。

```
import (
    "os"
    "os/signal"
    "sync"
    "syscall"
)

var exits = &struct {
    sync.RWMutex
    funcs   []func()
    signals chan os.Signal
}{}

func atexit(f func()) {
    exits.Lock()
    defer exits.Unlock()
    exits.funcs = append(exits.funcs, f)
```

```
}

func waitExit() {
    if exits.signals == nil {
        exits.signals = make(chan os.Signal)
        signal.Notify(exits.signals, syscall.SIGINT, syscall.SIGTERM)
    }

    exits.RLock()
    for _, f := range exits.funcs {
        defer f()                          // 即便某些函数 panic，也能确保后续函数执行。
    }                                      // 按 FILO 顺序执行。
    exits.RUnlock()

    <-exits.signals
}

func main() {
    atexit(func() { println("exit1 ...") })
    atexit(func() { println("exit2 ...") })

    waitExit()
}
```

将发往 channel 的数据打包，减少传输次数，可有效提升性能。从实现上来说，channel 队列依旧使用锁同步机制，单次获取更多数据，可改善因频繁加锁造成的性能问题。

```
const (
    max     = 50000000                 // 数据统计上限。
    block   = 500                      // 数据块大小。
    bufsize = 100                      // 缓冲区大小。
)

func test() {                          // 普通模式：每次传递一个整数。
    done := make(chan struct{})
    c := make(chan int, bufsize)

    go func() {
        count := 0
        for x := range c {
            count += x
        }

        close(done)
    }()

    for i := 0; i < max; i++ {
        c <- i
    }

    close(c)
```

```
    <-done

}

func testBlock() {                          // 块模式: 每次将 500 个数字打包成块传输。
    done := make(chan struct{})
    c := make(chan [block]int, bufsize)

    go func() {
        count := 0
        for a := range c {
            for _, x := range a {
                count += x
            }
        }

        close(done)
    }()

    for i := 0; i < max; i += block {
        var b [block]int                    // 数据打包。
        for n := 0; n < block; n++ {
            b[n] = i + n
            if i+n == max-1 {
                break
            }
        }

        c <- b
    }

    close(c)
    <-done
}
```

输出：

```
BenchmarkTest-4            1    4299047783 ns/op        3296 B/op          8 allocs/op
BenchmarkTestBlock-4      10     122825583 ns/op      401516 B/op          2 allocs/op
```

> 虽然单次消耗更多内存，但性能提升非常明显。如改用 []int 会造成更多的内存分配次数。

3. 同步

通道并非是用来取代锁，它们有各自不同的使用场景。通道倾向于解决逻辑层次的并发处理架构，而锁则用来保护局部范围内的数据安全。

标准库 sync 所提供的 Mutex、RWMutex 使用并不复杂，只几个地方需要注意。

将 Mutex 作为匿名字段时，相关方法必须实现为 pointer-receiver。否则会因复制的关系，导致锁机制失效。

```
type data struct {
    sync.Mutex
}

func (d data) test(s string) {
    d.Lock()
    defer d.Unlock()

    for i := 0; i < 5; i++ {
        println(s, i)
        time.Sleep(time.Second)
    }
}

func main() {
    var wg sync.WaitGroup
    wg.Add(2)

    var d data

    go func() {
        defer wg.Done()
        d.test("read")
    }()

    go func() {
        defer wg.Done()
        d.test("write")
    }()

    wg.Wait()
}
```

输出：

```
write 0
read  0
read  1
write 1
write 2
read  2
read  3
write 3
write 4
read  4
```

锁失效，改为 *data receiver 后正常。
也可用嵌入 *Mutex 来避免复制问题，但那需要额外初始化。

应将 Mutex 锁粒度控制在最小范围内，及早释放。

```
// 错误用法
func doSomething() {
    m.Lock()
    url := cache["key"]
    http.Get(url)                   // 该操作并不需要锁保护。
    m.Unlock()
}

// 正确用法
func doSomething() {
    m.Lock()
    url := cache["key"]
    m.Unlock()                      // 如使用 defer，则依旧将 Get 保护在内。
    http.Get(url)
}
```

Mutex 不支持递归锁，即便在同一 goroutine 下也会导致死锁。

```
func main() {
    var m sync.Mutex

    m.Lock()
    {
        m.Lock()
        m.Unlock()
    }
    m.Unlock()
}
```

输出：

```
fatal error: all goroutines are asleep - deadlock!
```

在设计并发安全类型时，千万注意此类问题。

```
type cache struct {
    sync.Mutex
```

```
    data []int
}

func (c *cache) count() int {
    c.Lock()
    n := len(c.data)
    c.Unlock()

    return n
}

func (c *cache) get() int {
    c.Lock()
    defer c.Unlock()

    var d int
    if n := c.count(); n > 0 {            // count 重复锁定，导致死锁。
        d = c.data[0]
        c.data = c.data[1:]
    }

    return d
}

func main() {
    c := cache{
        data: []int{1, 2, 3, 4},
    }

    println(c.get())
}
```

输出：

```
fatal error: all goroutines are asleep - deadlock!
```

---

1. 对性能要求较高时，应避免使用 defer Unlock。
2. 读写并发时，用 RWMutex 性能会更好一些。
3. 单个数据读写保护，可尝试用原子操作。
4. 执行严格测试，尽可能打开数据竞争检查。

十. 包结构

1. 工作空间

依照规范，工作空间（workspace）由 src、bin、pkg 三个目录组成。通常要将空间路径添加到 GOPATH 环境变量列表中，以便相关工具能正常工作。

```
workspace/
    |
    +-- src/                                源码。
    |    |
    |    +-- server/
    |    |     |
    |    |     +-- main.go
    |    |
    |    +-- service/
    |          |
    |          +-- user.go
    |
    +-- bin/                                可执行文件安装路径，不会创建额外子目录。
    |    |
    |    +-- server
    |
    +-- pkg/                                包安装路径，按操作系统和平台隔离。
         |
         +-- linux_amd64/
               |
               +-- service.a
```

包括子包在内的所有源码文件都必须保存在 src 目录下。至于 bin、pkg 两个目录，主要影响 go install/get 命令，它们会将编译结果（可执行文件或静态库）安装到这两个目录下，以实现增量编译。

环境变量

编译器等相关工具按 GOPATH 设置的路径搜索目标。也就是说在导入目标库时，排在列表前面的路径比当前工作空间优先级更高。另外，go get 默认将下载的第三方包保存到列表中第一个工作空间内。

正因为搜索优先级和默认下载位置等原因，社区对于是否为每个项目单独设置环境变量，还是将所有项目组织到同一个工作空间内存在争议。我个人做法是写一个脚本工具，作用类似 Python VirtualEnv，在激活某个项目时，自动设置相关环境变量。

> 请注意：不同操作系统下，GOPATH 列表的分隔符不同。UNIX-like 通常使用冒号，而 Windows 则使用分号。

环境变量 GOROOT 用于指示工具链和标准库的存放位置。在生成工具链时，相关路径就已经嵌入到可执行文件内，故无需额外设置。但如果出现类似下面这样的错误提示，请检查路径是否一致。

```
$ go build
go: cannot find GOROOT directory: /usr/local/go

$ strings `which go` | grep "/usr/local" | head -n1
/usr/local/go1.6
```

除通过设置 GOROOT 环境变量覆盖内部路径外，还可移动目录（改名），创建目录符号链接，或重新编译工具链来解决。

至于 GOBIN，则是强制替代工作空间的 bin 目录，作为 go install 目标保存路径。

> 在使用 Git 等版本控制工具时，建议忽略 pkg、bin 目录。直接在 src，或具体的子包下创建代码仓库（repository）。

2. 导入包

使用标准库或第三方包前，须用 import 导入，参数是工作空间中以 src 为起始的绝对路径。编译器从标准库开始搜索，然后依次是 GOPATH 列表中的各个工作空间。

```
import  "net/http"          // 实际路径: /usr/local/go/src/net/http
```

除使用默认包名外，还可使用别名，以避免多个包同名冲突。

```
import  osx  "github.com/apple/osx/lib"
import  nix  "github.com/linux/lib"
```

> 注意：import 导入参数是路径，而非包名。尽管习惯将包和目录名保持一致，但这不是强制规定。在代码中引用包成员时，须使用包名而非目录名。

归纳起来，有四种不同的导入方式。

```
import     "github.com/qyuhen/test"     默认方式：test.A
import  X  "github.com/qyuhen/test"     别名方式：X.A
import  .  "github.com/qyuhen/test"     简便方式：A
import  _  "github.com/qyuhen/test"     初始化方式：无法引用，仅用来初始化目标包。
```

> 便捷方式常用于单元测试代码中，不推荐在正式项目代码中使用。另外，初始化方式仅是为了让目标包的初始化函数得以执行，而非引用其成员。
>
> 不能直接或间接导入自己，不支持任何形式的循环导入。

未使用的导入（不包括“_”模式）会被编译器视为错误。

```
imported and not used: "fmt"
```

相对路径

除导入绝对路径外，还可用以“./”和“../”开头的相对路径。可在命令行使用，或用来快速构建一个非工作空间目录下的多包测试环境。

```
/demo/
  |
  +-- test/
  |    |
  |    +-- test.go
  |
  +-- lib/
       |
       +-- lib.go
```

test/test.go

```
package main

import (
```

```
    "../lib"                        # 相对路径导入。
)

func main() {
    lib.Hello()
}
```

lib/lib.go

```
package lib

func Hello() {
    println("Hello, World!")
}
```

不管是否在 test 目录下，只要命令行路径正确，就可用 go build/run/test 进行编译、运行或测试。但因缺少工作空间相关目录，go install 无法工作。

```
 /demo      $  go build -o hello ./test
 /demo/test $  go build
 /demo/test $  go run test.go
```

还有，相对路径无法在设置了 GOPATH 工作空间中编译。

```
$ go env GOPATH
/demo

$ go build
can't load package: /demo/src/test/test.go:4:5: local import "../lib" in non-local package
```

> go run 不受影响，依旧能正常执行。

自定义路径

相关工具对 GitHub 等代码托管环境提供了良好支持。但某些时候，我们可能希望使用自己的域名来定义下载和导入路径。实现方法很简单，在自家服务器对应路径下放置一个包含“go-import meta”跳转信息的页面即可。

myserver.go

```
package main

import (
    "fmt"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, `
        <meta name="go-import" content="qyuhen.com/test git https://github.com/qyuhen/test">
    `)
}

func main() {
    http.HandleFunc("/test", handler)
    http.ListenAndServe(":80", nil)
}
```

编译并启动服务器后，用新路径下载该包。

```
$ go get -v -insecure qyuhen.com/test

Fetching https://qyuhen.com/test?go-get=1
https fetch failed: Get https://qyuhen.com/test?go-get=1: connection refused

Fetching http://qyuhen.com/test?go-get=1
Parsing meta tags from http://qyuhen.com/test?go-get=1 (status code 200)

get "qyuhen.com/test": found meta tag main.metaImport{
    Prefix:"qyuhen.com/test",
    VCS:"git",
    RepoRoot:"https://github.com/qyuhen/test"
} at http://test.com/test?go-get=1

qyuhen.com/test (download)
```

从输出信息中，可以看到跳转过程。并且保存目录不再是 github.com，而是自有域名。

只是如此一来，该包就可用两个不同路径下载，本地也可能会存在两个副本。为避免版本不一致等情况发生，可添加“import comment”，让编译器检查导入路径与其是否一致。

github.com/qyuhen/test/hello.go

```
package lib  // import "qyuhen.com/test"
```

```
func Hello() {
    println("Hello!")
}
```

添加注释后，就要求该包必须以“qyuhen.com/test”导入，否则编译出错。因 go get 也会执行编译操作，所以用“github.com/qyuhen/test”下载安装同样失败。

```
$ go build
can't load package: package test:
  code in directory github.com/qyuhen/test expects import "qyuhen.com/test"
```

强制使用唯一导入路径，便于日后迁移。糟心的是，此方式对 Vendor 无效。

---

3. 组织结构

包（package）由一个或多个保存在同一目录下（不含子目录）的源码文件组成。包的用途类似名字空间（namespace），是成员作用域和访问权限边界。

包名与所在目录名无关。虽说习惯保持一致，但并非硬性规定。

```
package service

func Ping() {
    return "pong"
}
```

包名通常使用单数形式。
源码文件使用 UTF-8 格式，否则会导致编译出错。

同一目录下所有源码文件必须使用相同包名称。因导入时使用绝对路径，所以在搜索路径下，包必须有唯一路径，但无须是唯一名字。

```
$ go list net/...          # 显示包路径列表。("..." 表示其下所有包)

net
```

```
net/http
net/http/cgi
net/http/cookiejar
net/http/fcgi
net/http/httptest
net/http/httputil
net/http/internal
net/http/pprof
net/internal/socktest
net/mail
net/rpc
net/rpc/jsonrpc
net/smtp
net/textproto
net/url
```

另有几个被保留有特殊含义的包名称。

- **main**: 可执行入口（入口函数 main.main）。
- **all**: 标准库以及 GOPATH 中能找到的所有包。
- **std, cmd**: 标准库及工具链。
- **documentation**: 存储文档信息，无法导入（和目录名无关）。

相关工具忽略以“.”或“_”开头的目录或文件，但又允许导入保存在这些目录中的包。Orz！

权限

所有成员在包内均可访问，无论是否在同一源码文件中。但只有名称首字母大写的为可导出成员，在包外可视。

该规则适用于全局变量、全局常量、类型、结构字段、函数、方法等。

通过指针转换等方式绕开该限制。

lib/data.go

```
package lib

type data struct {
    x   int
    Y   int
}
```

```
func NewData() *data {
    return new(data)
}
```

test.go

```
package main

import (
    "fmt"
    "test/lib"
    "unsafe"
)

func main() {
    d := lib.NewData()
    d.Y = 200                                          // 直接访问导出字段。

    p := (*struct{ x int })(unsafe.Pointer(d))         // 利用指针转换访问私有字段。
    p.x = 100

    fmt.Printf("%+v\n", *d)
}
```

输出：

```
{x:100, Y:200}
```

初始化

包内每个源码文件都可定义一到多个初始化函数，但编译器不保证执行次序。

实际上，所有这些初始化函数（包括标准库和导入的第三方包）都由一个自动生成的包装函数调用。因此可保证在单一线程上执行，且仅执行一次。

> 从当前版本实现看，执行次序与依赖关系、文件名和定义次序有关。只是这种次序非常不便于维护，容易引起混乱。初始化函数之间不应有逻辑关联，最好仅处理当前文件的初始化操作。

全局变量被保存在 .noptrdata、.data 等区域，可在初始化函数中安全访问。只有等初始化函数全部执行结束后，运行时才会进入 main.main 入口函数。

```
var x = 100

func init() {
    println("init:", x)
    x++
}

func main() {
    println("main:", x)
}
```

输出：

```
init: 100
main: 101
```

可在初始化函数中创建 goroutine，可等它结束。

```
func init() {
    done := make(chan struct{})

    go func() {
        defer close(done)
        fmt.Println("init:", time.Now())
        time.Sleep(time.Second * 2)
    }()

    <-done
}

func main() {
    fmt.Println("main:", time.Now())
}
```

输出：

```
init: 2016-02-22 09:50:39
main: 2016-02-22 09:50:41
```

如在多个初始化函数中引用全局变量，那么最好直接在变量定义处赋值。因为无法保证执行次序，所以任何初始化函数中的赋值都有可能“延迟无效”。

b.go

```
func init() {
    println("init:", x)
}
```

main.go

```
var x int

func init() {
    x = 100
}

func main() {
    println("main:", x)
}
```

输出：

```
init: 0
main: 100
```

初始化函数无法调用。

```
func init() {
    println("init")
}

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

输出：

```
undefined: init
```

内部包

在进行代码重构时，我们会将一些内部模块陆续分离出来，以独立包形式维护。此时，基于首字母大小写的访问权限控制就显得过于粗旷。因为我们希望这些包导出成员仅在特定范围内访问，而不是向对最终用户公开。

内部包机制相当于增加了新的访问权限，所有保存在 internal 目录下的包（包括自身）仅能被其父目录下的包（含所有层次的子目录）访问。

结构示意：

```
src/
 |
 +-- main.go
 |
 +-- lib/                          # 内部包 internal、a、b 仅能被 lib、lib/x、lib/x/y 访问。
      |
      +-- internal/                # 内部包之间可相互访问。
      |      |
      |      +-- a/                # 可导入外部包，比如 lib/x/y。
      |      |
      |      +-- b/
      |
      +-- x/
          |
          +-- y/
```

在 lib 目录以外（比如 main.go）导入内部包会引发编译错误。

```
imports lib/internal/a: use of internal package not allowed
```

> 导入内部包必须使用完整路径，例如：import "lib/internal/a"。

4. 依赖管理

如何管理和保存第三方包，一直是个头疼的问题。

将所有项目的第三方依赖都放到一个独立工作空间中，可能会导致版本冲突。但如果放到项目工作空间，又会把工作目录搞得面目全非。为此，专门引入名为 vendor 的机制，专门存放第三方包，实现将项目源码和第三方依赖完整打包分发。

> 如果说 internal 针对内部，那么 vendor 显然就是 external。

```
src/
 |
 +-- server/
       |
       +-- vendor/
       |     |
       |     +-- github.com/
       |           |
       |           +-- qyuhen/
       |                 |
       |                 +-- test/
       +-- main.go
```

main.go

```
package main

import "github.com/qyuhen/test"

func main() {
    test.Hello()
}
```

> 在 main.go 中导入 github.com/qyuhen/test 时，优先使用 vendor/github.com/qyuhen/test。

导入 vendor 中第三方包，参数是以 vendor/ 为基准的绝对路径。这就避免了 vendor 机制带来的麻烦，让代码无论使用 vendor 还是 GOPATH 都能保持一致。注意，vendor 比标准库优先级更高。

当多个 vendor 目录嵌套时，如何查找正确目标？要知道引入的第三方包也可能有自己的 vendor 依赖目录。

```
src/
 |
 +-- server/
       |
       +-- vendor/
       |      |
       |      +-- p/                              # p1: src/vendor/p
       |      |
       |      +-- x/
       |          |
       |          +-- test.go
       |          |
       |          +-- vendor/
       |                |
       |                +-- p/                    # p2: src/vendor/x/vendor/p
       +-- main.go
```

显然，上面这个例子中有两个名为 p 的包，在 main.go 和 test.go 分别导入 p 时，它们各自对应谁？

规则算不上复杂：从当前源文件所在目录开始，逐级向上构造 vendor 全路径，直到发现路径匹配的目标为止。匹配失败，则依旧搜索 GOPATH。

对 main.go 而言，可构造出的路径是 src/server/vendor/p，也就是 p1。而 test.go 最先构造出的路径是 src/server/vendor/x/vendor/p，所以选择 p2。

查看 go/src/cmd/go/pkg.go vendoredImportPath 函数源码获知算法细节。

要使用 vendor 机制，需开启“GO15VENDOREXPERIMENT=1”环境变量开关，Go 1.6 默认启用，而 1.5 则需要额外设置。

使用 go get 下载第三方包时，依旧使用 GOPATH 第一个工作空间，而非 vendor。
当前的工具链中并没有真正意义上的包依赖管理，好在有不少第三方工具可选。