cookbookgo

整理知识，传播智慧

好好学习，天天向上

gopher：石鸟路遇

Github：https://github.com/stonebirdjx

1、高文典册

函数体外每个语句应该以关键关键字开始（var、func、type、const、import、package）等

Go 是静态类型语言，不能在运行期改变变量类型。

不支持运算符重载。尤其需要注意，"++"、"--" 是语句而非表达式。只有i++，i--，没有++i，--i

没有 "~"，取反运算也用 "^"。

range会保存副本。对array 、slice的修改不会影响range。对的map修改会影响range。

break 可用于 for、switch、select，而 continue 仅能用于 for 循环。

函数返回值分为匿名放回、具名返回、nil值

命名(具名)返回参数允许 defer 延迟调用通过闭包读取和修改。

滥用 defer 可能会导致性能问题，尤其是在一个 "大循环" 里。

捕获函数 recover 只有在延迟调用内直接调用(defer func里面直接调用)才会终止错误，否则总是返回 nil。任何未捕获的错误都会沿调用堆栈向外传递。

标准库 errors.New 和 fmt.Errorf 函数用于创建实现 error 接口的错误对象。

使用len()计算切片长度时间复杂度为O(1)，不需要遍历切片

使用cap()计算切片容量时间复杂度为O(1)，不需要遍历切片

数组b := [...][2]int{{1, 1}, {2, 2}, {3, 3}} // 第 2 纬度不能用 "..."。第二维（多维）不能用语法糖

如果原Slice容量小于1024，则新Slice容量将扩大为原来的2倍；
如果原Slice容量大于等于1024，则新Slice容量将扩大为原来的1.25倍；

应及时将所需数据 copy 到较小的 slice，以便释放超大号底层数组内存。

copy过程中不会发生扩容,取两个切片容量的最小值

map中key不存在，取值会取零值。

map可以bucket可以存储8个键值对，当平均每个bucket超过6.5个会发送扩容。先等量扩容（内部挪动），存不下就增量扩容2倍

删除map的key，不会降低map容量。

空struct{}不占空间，可以用于实现set集合，或者占位。

return不是原子操作，执行过程是: 保存返回值(若有)-->执行defer（若有）-->执行ret跳转。

常见的tag用法，主要是JSON、xml数据解析、ORM映射等。

空接口 interface{} 没有任何方法签名，也就意味着任何类型都实现了空接口。其作用类似面向对象语言中的根对象 object。

没有*interface{}

当对interface变量进行判断是否为nil时，只有当动态类型和动态值都是nil，这个变量才是nil

go程：用以实现 "以通讯来共享内存" 的 CSP 模式。

向 closed channel 发送数据引发 panic 错误，接收立即返回零值。而 nil channel，无论收发都会被阻塞。

一个init函数只能有一个goroutine执行一次。runtime.init ->main.init->main.main

init初始化函数应该只负责当前包，或者当前逻辑。

内存分配

> arena：(512G）分成一个一个page（8KB）。
>
> spans：(512M) ,用于存放span指针，每个指针对应一个page
>
> 每个*span*存放小对象（*class*），以*8*的倍数为单位，能存放*66*种小对象，超过*32K*，由特殊的*class*对待
>
> bitmap：(16G):主要用于GC

编译器，将对象尽可能分配到栈上。减少垃圾回收的压力，有助于提升性能。

三色标记法：

- 白色：对象未被标记，gcmarkBits对应的位为0（该对象将会在本次GC中被清理）
- 灰色：对象还在标记队列中等待
- 黑色：对象已被标记，gcmarkBits对应的位为1（该对象不会在本次GC中被清理）

初始状态下所有对象都是白色的。接下来就开始分析灰色对象，分析A时，A没有引用其他对象很快就转入黑色。最终，黑色的对象会被保留下来，白色对象会被回收掉。

Golang中的STW（Stop The World）就是停掉所有的goroutine，专心做垃圾回收，待垃圾回收结束后再恢复goroutine。

优化算法：写屏障、辅助GC

回收方式：超过阈值回收、定时回收（2分钟）、手动回收runtime.GC()

所谓逃逸分析（Escape analysis）是指由编译器决定内存分配的位置，不需要程序员指定。

逃逸场景：指针逃逸、栈空间内存不足逃逸、动态类型逃逸（传入interface{}）、闭包逃逸（定义了外部局部变量）。

2、语言

2.1、变量

使用关键字 var 定义变量，自动初始化为零值。如果提供初始化值，可省略变量类型，由编译器自动推断。

> ":=" 精简方式定义变量 只可以写在函数内部。

```
var x, y, z int
var s, n = "abc", 123
var (
    a int
    b float32
)
func main() {
    n, s := 0x1234, "Hello, World!"
    println(x, s, n)
}
```

特殊只写变量 "_"，用于忽略值占位。

> 编译器会将未使用的局部变量当做错误。

```
var s string // 全局变量没问题。
func main() {
    i := 0 // Error: i declared and not used。(可使用 "_ = i" 规避)
}
```

2.2、常量

> 常量值必须是编译期可确定的数字、字符串、布尔值。

```
const x, y int = 1, 2 // 多常量初始化
const s = "Hello, World!" // 类型推断
```

在常量组中，如不提供类型和初始化值，那么视作与上一常量相同。

```
const (
    s = "abc"
    x           // x = "abc"
)
```

常量值还可以是 len、cap、unsafe.Sizeof 等编译期可确定结果的函数返回值。

> 由于常量需要编译器可确定，最好不用函数，除非函数对常量计算，返回可确定结果

```
const (
    a = "abc"
    b = len(a)
    c = unsafe.Sizeof(b)
)
```

枚举

关键字 iota 定义常量组中从 0 开始按行计数的自增枚举值。

iota是const的索引，在n行，值为n-1

```
const (
    Sunday = iota // 0
    Monday // 1，通常省略后续行表达式。
    Tuesday // 2
    Wednesday // 3
```

```
    Thursday // 4
    Friday // 5
    Saturday // 6
)

const (
    _ = iota // iota = 0
    KB int64 = 1 << (10 * iota) // iota = 1
    MB                          // 与 KB 表达式相同，但 iota = 2
    GB
    TB
)
```

在同一常量组中，可以提供多个 iota，它们各自增长

```
const (
    A, B = iota, iota << 10     // 0, 0 << 10
    C, D                        // 1, 1 << 10
)
```

如果 iota 自增被打断，须显式恢复。

```
const (
    A = iota    // 0
    B           // 1
    C = "c"     // c
    D           // c，与上一行相同。
    E = iota    // 4，显式恢复。注意计数包含了 C、D 两行。
    F           // 5
)
```

2.3、类型说明

类型	长度	默认值（零值）	说明
bool	1	false	
byte	1	0	uint8
rune	4	0	int32
int，uint	4或8	0	32位、64位系统
int8、uint8	1	0	-128-127, 0~255
int16、uint16	2	0	-32768 ~ 32767, 0 ~ 65535
int32、uint32	4	0	-（2~31）-（2~31）-1，0 ~（2~32）-1
int64、uint64	8	0	-（2~63）-（2~63）-1，0 ~（2~64）-1
float32	4	0	
float64	8	0	
complex64	8	(0+0i)	
complex128	16	(0+0i)	
uintptr	4或8	0	足以存储指针的 uint32 或 uint64 整数
array		值类型的默认值	值类型
struct		值类型的默认值	值类型
string		""	utf-8字符串
slice		nil	引用类型
map		nil	引用类型
channel		nil	引用类型
interface		nil	接口
function		nil	函数

零值说明

```
int uint float32 float64//数值类型为 0，
bool //布尔类型为 false，
string //字符串为 ""（空字符串）。
array struct // 默认值为值类型
//而其他引用类型（指针、切片、映射、通道、函数和接口）的零值则是 nil
```

2.4、引用类型

引用类型包括 slice、map 和 channel。它们有复杂的内部结构，除了申请内存外，还需要初始化相关属性。

内置函数 new 计算类型大小，为其分配零值内存，返回指针。而make 会被编译器翻译成具体的创建函数，由其分配内存和初始化成员结构，返回对象而非指针。

```
    a := new(int)
    b := new(float64)
    d := new([10]int)
    e := new(struct{})
    c := make(map[string]string)
    f := make([]int,10,20)
    g := make(chan int)
```

> *new* 适用值类型 *int*、*float32*，*string*、数组、结构体等
>
> *make*适用于引用类型*slice*、*map*、*channel*

2.5、类型转换

不支持隐式类型转换，即便是从窄向宽转换也不行。T(v)

```
var b byte = 100
// var n int = b // Error: cannot use b (type byte) as type int in assignment
var n int = int(b) // 显式转换
```

使用括号避免优先级错误。

```
*Point(p) // 相当于 *(Point(p))
(*Point)(p)
<-chan int(c) // 相当于 <-(chan int(c))
(<-chan int)(c)
```

2.6、字符串

字符串是不可变值类型，内部用指针指向 UTF-8 字节数组。

用索引号访问某字节，如 s[i]。返回byte uint8

要修改字符串，可先将其转换成 []rune 或 []byte，完成后再转换为 string。无论哪种转换，都会重新分配内存，并复制字节数组。

用 for 循环遍历字符串时，也有 byte 和 rune 两种方式。

```
func main() {
    s := "abcef"
    for i := 0; i < len(s); i++ { // byte
        fmt.Printf("%c,", s[i])
    }
    fmt.Println()
    for _, r := range s { // rune
        fmt.Printf("%c,", r)
    }
}
```

> 字符串的存储在只读段上，并不是堆上或者栈上

2.7、指针

支持指针类型 *T，指针的指针 **T，以及包含包名前缀的 *.T。

操作符 "&" 取变量地址，"*" 透过指针访问目标对象。

不支持指针运算。直接用 "." 访问目标成员。

```
x := 1234
p := &x
p++ // Error: invalid operation: p += 1 (mismatched types *int and int)
```

可以在 unsafe.Pointer 和任意类型指针间进行转换。

```
func main() {
    x := 0x12345678
    p := unsafe.Pointer(&x) // *int -> Pointer
    n := (*[4]byte)(p) // Pointer -> *[4]byte
    for i := 0; i < len(n); i++ {
        fmt.Printf("%X ", n[i])
    }
}
```

返回局部变量指针是安全的，编译器会根据需要将其分配在 GC Heap 上。

```
func test() *int {
    x := 100
    return &x // 在堆上分配 x 内存。但在内联时，也可能直接分配在目标栈。
}
```

将 Pointer 转换成 uintptr，可变相实现指针运算。

```
func main() {
    d := struct {
        s string
        x int
    }{"abc", 100}
    p := uintptr(unsafe.Pointer(&d)) // *struct -> Pointer -> uintptr
    p += unsafe.Offsetof(d.x) // uintptr + offset

    p2 := unsafe.Pointer(p) // uintptr -> Pointer
    px := (*int)(p2) // Pointer -> *int
    *px = 200 // d.x = 200
    fmt.Printf("%#v\n", d)
}
```

> interface{}能接收任何类型， 只有interfave{} ,没有* *interface{}*

2.8、自定义类型

可用 type 在全局或函数内定义新类型。

```
x := 1234
var b bigint = bigint(x) // 必须显式转换，除非是常量。
var b2 int64 = int64(b)
```

可将类型分为命名（值类型）和未命名（引用类型）两大类。

命名类型（值类型）包括 bool、int、string、float32 等，

而 array、slice、map 等和具体元素类型、长度等有关，属于未命名类型（引用类型）。

具有相同声明的未命名类型被视为同一类型。

- 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。
- 具有相同元素类型和传送方向的 channel。
- 具有相同字段序列 (字段名、类型、标签、顺序) 的匿名 struct。
- 签名相同 (参数和返回值，不包括参数名称) 的 function。
- 方法集相同 (方法名、方法签名相同，和次序无关) 的 interface。

3、表达式

3.1、保留字(关键字)

关键字只有25个，nil 不属于关键字。

```
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
```

3.2、运算符

简单位运算演示。

```
0110 & 1011 = 0010 AND 都为 1。
0110 | 1011 = 1111 OR 至少一个为 1。
0110 ^ 1011 = 1101 XOR 只能一个为 1。
0110 &^ 1011 = 0100 AND NOT 清除标志位。
```

标志位操作。

```
a := 0
a |= 1 << 2 // 0000100: 在 bit2 设置标志位。
a |= 1 << 6 // 1000100: 在 bit6 设置标志位
a = a &^ (1 << 6) // 0000100: 清除 bit6 标志位。
```

没有 "~"，取反运算也用 "^"。

```
x := 1
x, ^x // 0001, -0010
```

3.3、初始化

初始化复合对象，必须使用类型标签，且左大括号必须在类型尾部。

初始化值以 "," 分隔。可以分多行，但最后一行必须以 "," 或 "}" 结尾。

```
var b = []int{1, 2, 3}
a := []int{
        1,
        2 // Error: need trailing comma before newline in composite literal
    }
a := []int{
        1,
        2, // ok
    }
```

3.4、控制流

3.4.1、if

• 可省略条件表达式括号。

• 支持初始化语句，可定义代码块局部变量。

• 代码块左大括号必须在条件表达式尾部。

```
x := 0
if x > 10 {
}
if n := "abc"; x > 0 { // 初始化语句未必就是定义变量，比如 println("init") 也是可以的。
    println(n[2])
} else if x < 0 { // 注意 else if 和 else 左大括号位置。
    println(n[1])
} else {
    println(n[0])
}
```

不支持三元操作符 "a > b ? a : b"。

3.4.2、for

支持三种循环方式，包括类 while 语法。

golang 只有for循环，基本的 for 循环由三部分组成，它们用分号隔开：for $1;$2;$3{} {} 必须有，()只能用在$2上，不能for($1;$2;$3){}

```
    s := "abc"
    for i, n := 0, len(s); i < n; i++ { // 常见的 for 循环，支持初始化语句。
        println(s[i])
    }
    n := len(s)
    for n > 0 { // 替代 while (n > 0) {}
        println(s[n]) // 替代 for (; n > 0;) {}
        n--
    }
    for { // 替代 while (true) {}
        println(s) // 替代 for (;;) {}
    }

    for i, n := 0, length(s); i < n; i++ { // 避免多次调用 length 函数。
```

```
        println(i, s[i])
    }
```

3.4.3、range

类似迭代器操作，返回 (索引, 值) 或 (键, 值)。

string 、array/slice、channel、map

可忽略不想要的返回值，或用 "_" 这个特殊变量。

```
s := "abc"
for i := range s { // 忽略 2nd value，支持 string/array/slice/map。
    println(s[i])
}
for _, c := range s { // 忽略 index。
    println(c)
}
for range s { // 忽略全部返回值，仅迭代。
    ...
}
m := map[string]int{"a": 1, "b": 2}
for k, v := range m { // 返回 (key, value)。
    println(k, v)
}
```

range会保存副本。对array 、slice的修改不会影响range

```
s := []int{1, 2, 3, 4, 5}
for i, v := range s { // 复制 struct slice { pointer, len, cap }。
    if i == 0 {
        s = s[:3] // 对 slice 的修改，不会影响 range。
        s[2] = 100 // 对底层数据的修改。
    }
    println(i, v)
}
```

对map的修改会影响range

3.4.4、switch

分支表达式可以是任意类型，不限于常量。可省略 break，默认自动终止。

```
switch i {
 case x[1]:
    println("a")
 case 1, 3:
    println("b")
 default:
    println("c")
}
```

如需要继续下一分支，可使用 fallthrough，但不再判断条件。

```
switch x {
case 10:
    println("a")
    fallthrough
case 0:
    println("b")
}
```

省略条件表达式，可当 if...else if...else 使用。

```
switch {
     case x[1] > 0:
     println("a")
     case x[1] < 0:
     println("b")
     default:
     println("c")
}

switch i := x[2]; { // 带初始化语句
     case i > 0:
     println("a")
     case i < 0:
     println("b")
     default:
     println("c")
}
```

3.4.5、goto、break、continue

支持在函数内 goto 跳转。标签名区分大小写，未使用标签引发错误。

```
    var i int
    for {
        println(i)
        i++
        if i > 2 {
            goto BREAK
        }
    }
BREAK:
    println("break")
EXIT: // Error: label EXIT defined and not used
```

配合标签，break 和 continue 可在多级嵌套循环中跳出。

```
func main() {
L1:
    for x := 0; x < 3; x++ {
L2:
        for y := 0; y < 5; y++ {
            if y > 2 { continue L2 }
            if x > 1 { break L1 }
            print(x, ":", y, " ")
        }
        println()
    }
}
```

break 可用于 for、switch、select，而 continue 仅能用于 for 循环。

4、函数

4.1、函数定义

不支持 嵌套 (nested)、重载 (overload) 和 默认参数 (default parameter)。

• 无需声明原型。

• 支持不定长变参。

• 支持多返回值。

• 支持命名返回参数。

• 支持匿名函数和闭包。

```
func format(fn FormatFunc, s string, x, y int) string {
    return fn(s, x, y)
}
```

有返回值的函数，必须有明确的终止语句，否则会引发编译错误。

4.2、可变参数

变参本质上就是 slice。只能有一个，且必须是最后一个。

```
func test(s string, n ...int) string {
    var x int
    for _, i := range n {
        x += i
    }
    return fmt.Sprintf(s, x)
}
```

4.3、返回值

命名返回参数可看做与形参类似的局部变量，最后由 return 隐式返回。

```go
func split(sum int) (int,int) {
    return 10,20
}
func split(sum int) (x, y int) {
    x = sum * 4 / 9
    y = sum - x
    return //具名返回，要么不带参数，要么全部带上
    //如果函数没有返回值，默认是nil，return也是nil
    // return 10 , y 会自动把10赋值给x并返回
}
```

命名返回参数允许 defer 延迟调用通过闭包读取和修改。

```go
func add(x, y int) (z int) {
    defer func() {
        z += 100
    }()
    z = x + y
    return
}
func main() {
    println(add(1, 2)) // 输出: 103
}
```

4.4、匿名函数

匿名函数可赋值给变量，做为结构字段，或者在 channel 里传送。

```go
func main() {
    // --- function variable ---
    fn := func() { println("Hello, World!") }
    fn()
    // --- function collection ---
    fns := [](func(x int) int){
        func(x int) int { return x + 1 },
        func(x int) int { return x + 2 },
    }
    println(fns[0](100))
    // --- function as field ---
    d := struct {
        fn func() string
    }{
        fn: func() string { return "Hello, World!" },
    }
    println(d.fn())
    // --- channel of function ---
    fc := make(chan func() string, 2)
    fc <- func() string { return "Hello, World!" }
    println((<-fc)())
}
```

闭包复制的是原对象指针，这就很容易解释延迟引用现象。闭包可以造成内存逃逸

```go
func test() func() {
    x := 100
```

```
    fmt.Printf("x (%p) = %d\n", &x, x)
    return func() {
        fmt.Printf("x (%p) = %d\n", &x, x)
    }
}
func main() {
    f := test()
    f()
}

// x (0x2101ef018) = 100
// x (0x2101ef018) = 100
```

4.5、延迟调用

关键字 defer 用于注册延迟调用。这些调用直到 ret 前才被执行，通常用于释放资源或错误处理。

多个 defer 注册，按 FILO 次序执行。哪怕函数或某个延迟调用发生错误，这些调用依旧会被执行。

延迟调用参数在注册时求值或复制，可用指针或闭包 "延迟" 读取。

```
    var x,y int
    defer func(i int) {
        println("defer:", i, y) // y 闭包引用
    }(x)
```

滥用 defer 可能会导致性能问题，尤其是在一个 "大循环" 里。

申请资源后立即使用defer关闭资源是好习惯

4.6、错误处理

没有结构化异常，使用 panic 抛出错误，recover 捕获错误。recover 必须在defer里面使用.

由于 panic、recover 参数类型为 interface{}，因此可抛出任何类型对象。

```
func test() {
    defer func() {
        if err := recover(); err != nil {
            println(err.(string)) // 将 interface{} 转型为具体类型。
        }
    }()
    panic("panic error!")
}
```

延迟调用中引发的错误，可被后续延迟调用捕获，但仅最后一个错误可被捕获。

```
func test() {
    defer func() {
        fmt.Println(recover())
    }()
    defer func() {
        panic("defer panic")
    }()
    panic("test panic")
}

// defer pannic
```

捕获函数 recover 只有在延迟调用内直接调用(defer func里面直接调用)才会终止错误，否则总是返回 nil。任何未捕获的错误都会沿调用堆栈向外传递。

```
func test() {
    defer recover() // 无效!
    defer fmt.Println(recover()) // 无效!
    defer func() {
        func() {
            println("defer inner")
            recover() // 无效!
        }()
    }()
    panic("test panic")
}
// ----------------------------------------
func except() {
    recover()
}
func test() {
    defer except() // 有效
    panic("test panic")
}
```

除用 panic 引发中断性错误外，还可返回 error 类型错误对象来表示函数调用状态。标准库 errors.New 和 fmt.Errorf 函数用于创建实现 error 接口的错误对象。通过判断错误对象实例来确定具体错误类型。

5.数据

5.1、数组array

数组是值类型，赋值和传参会复制整个数组，而不是指针。

数组长度必须是常量，且是类型的组成部分。[2]int 和 [3]int 是不同类型。

支持 "=="、"!=" 操作符，因为内存总是被初始化过的。

指针数组 [n]*T，数组指针 *[n]T。

内置函数 len 和 cap 都返回数组长度 (容量)。

支持多维数组。

```
var a [10]int //下面情况是等价的
a[0:10]、a[:10]、a[0:]、a[:]

a := [2][3]int{{1, 2, 3}, {4, 5, 6}}
b := [...][2]int{{1, 1}, {2, 2}, {3, 3}} // 第 2 纬度不能用 "..."。第二维（多维）不能用语法糖
```

值拷贝行为会造成性能问题，通常会建议使用 slice，或数组指针。

5.2、slice

slice 并不是数组或数组指针。它通过内部指针和相关属性引用数组片段，以实现变长方案。

如果 slice == nil，那么 len、cap 结果都等于 0。

可以向字典一样通过索引初始化

```
//数组
var a = [4]string{
        1:"hjx",
        2:"nan",
        "nxlg"  //不指定索引。默认是前面的索引 +1
        0:"18",
}

//切片
s1 := []int{0, 1, 2, 3, 8: 100} //未指定的索引按零值初始化
```

预先给slice申请一个容量，有利于提高性能，避免重复扩容

```
s := make([]int,0,1024)
```

5.2.3、reslice

所谓 reslice，是基于已有 slice 创建新 slice 对象，以便在 cap 允许范围内调整属性。

```
s := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
s1 := s[2:5] // [2 3 4]
s2 := s1[2:6:7] // [4 5 6 7]
s3 := s2[3:6] // Error
```

5.2.4、append

向 slice 尾部添加数据，返回新的 slice 对象。一旦超出原 slice.cap 限制，就会重新分配底层数组。

```
//append 函数追加切片
s = append(s, 1)
s = append(s, 2, 3, 4) // 可以一次性添加多个元素
s = append(s, []int{1,2,3}...) //追加一个新切片，go语法糖
```

使用append向Slice追加元素时，如果Slice空间不足，将会触发Slice扩容，扩容实际上重新一配一块更大的内存，将原Slice数据拷贝进新Slice，然后返回新Slice，扩容后再将数据追加进去。

> 如果原Slice容量小于1024，则新Slice容量将扩大为原来的2倍；
> 如果原Slice容量大于等于1024，则新Slice容量将扩大为原来的1.25倍；

5.2.5、copy

函数 copy 在两个 slice 间复制数据，复制长度以 len 小的为准。

```
copy(s2, s)
```

应及时将所需数据 copy 到较小的 slice，以便释放超大号底层数组内存。

创建切片时可`跟据实际需要预分配容量`，尽量避免追加过程中扩容操作，有利于提升性能；尽量使用 s := make([]int,l,c)方式定义切片。

5.2.6、slice原理

```
// src/runtime/slice.go:slice
type slice struct {
    array unsafe.Pointer
    len   int
    cap   int
}
```

特殊切片

```
sliceA := make([]int, 5, 10)  //length = 5; capacity = 10
sliceB := sliceA[0:5]         //length = 5; capacity = 10
sliceC := sliceA[0:5:5]       //length = 5; capacity = 5
```

5.3 map

引用类型，哈希表。键必须是支持相等运算符 (==、!=) 类型，比如 number、string、pointer、array、struct，以及对应的 interface。值可以是任意类型，没有限制。

> *key*必须为值类型，不能为引用类型

预先给 make 函数一个合理元素容量参数，有助于提升性能。

```
m := make(map[string]int, 1000)
```

常见操作

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

for k, v := range m { // 迭代，可仅返回 key。随机顺序返回，每次都不相同。
    println(k, v)
}

var mp map[string]string // nil map 不能添加键
var mp = map[string]string{} // 不是nil，申请了空间，可以添加键 len(mp) == 0
var mp = make(map[string]string)
```

```
m[key] = elem //在映射 m 中插入或修改元素：
elem := m[key] //获取元素，最好用if ele，ok := m[key];ok{} 来取字典的值，
// 空key 返回其类型的默认值
delete(m, key) // 删除元素
```

不能保证迭代返回次序，通常是随机结果，具体和版本实现有关。

5.3.1、map原理

map使用哈希表作为底层实现，一个哈希表里可以有多个哈希表节点，也即`bucket(哈希桶)`，而每个bucket就保存了map中的一个或一组键值对。

```
// runtime/map.go/hmap
type bmap struct {
    tophash [8]uint8 //存储哈希值的高8位
    data    byte[1]  //key value数据:key/key/key/.../value/value/value...
    overflow *bmap   //溢出bucket的地址
}
```

每个桶可以存储8个键值对。

> *hash*冲突：当有两个或以上数量的键被哈希到了同一个*bucket*时，我们称这些键发生了冲突。*Go*使用链地址法来解决键冲突。

负载因子=键数量/bucket数量

> 扩容：当平均每个*bucket*存储超过*6.5*个*key*时、或者*overflow*数量 > *2^15*时，也即*overflow*数量超过*32768*时，才会扩容。

增量扩容：原来的2倍

等量扩容：所谓等量扩容，实际上并不是扩大容量，buckets数量不变，重新做一遍类似增量扩容的搬迁动作，把松散的键值对重新排列一次，以使bucket的使用率更高，进而保证更快的存取。

5.4、struct

结构体是值类型，赋值和传参会复制全部内容。可用 "_" 定义补位字段，支持指向自身类型的指针成员

```
type Node struct {
    _ int
    id int
    data *byte
    next *Node
}
```

顺序初始化必须包含全部字段，否则会出错。

支持匿名结构，可用作结构成员或定义变量。

```
type File struct {
    name string
    size int
    attr struct {
        perm int
        owner int
    }
}
```

```
f := File{
    name: "test.txt",
    size: 1025,
    // attr: {0755, 1}, // Error: missing type in composite literal
}
f.attr.owner = 1
f.attr.perm = 0755
var attr = struct {
    perm int
    owner int
}{2, 0755}
f.attr = attr
```

支持 "=="、"!=" 相等操作符，可用作 map 键类型。

可定义字段标签，用反射读取。标签是类型的组成部分。

空结构 "节省" 内存，比如用来实现 set 数据结构，或者实现没有 "状态" 只有方法的 "静态类"。

5.4.1 tag

`Tag` 本身是一个字符串，但字符串中却是：`以空格分隔的 key:value 对`。

> 注意：冒号前后不能有空格

```
type Server struct {
    ServerName string `key1:"value1" key11:"value11"`
    ServerIP   string `key2:"value2"`
}
```

使用reflect反射包获取tag

```
type Server struct {
    ServerName string `key1:"value1" key11:"value11"`
    ServerIP   string `key2:"value2"`
}

func main() {
    s := Server{}
    st := reflect.TypeOf(s)

    field1 := st.Field(0)
    fmt.Printf("key1:%v\n", field1.Tag.Get("key1"))
    fmt.Printf("key11:%v\n", field1.Tag.Get("key11"))

    filed2 := st.Field(1)
    fmt.Printf("key2:%v\n", filed2.Tag.Get("key2"))
}
```

常见的tag用法，主要是JSON、xml数据解析、ORM映射等。

6、方法

方式是一种特殊的函数(带接收者参数的函数)。方法总是绑定对象实例，并隐式将实例作为第一实参(receiver)。

只能为当前包内命名类型定义方法。

不支持方法重载，receiver 只是参数签名的组成部分。

通过匿名字段（结构体组合的方式），可获得和继承类似的复用能力。

方法集(指针可以调用非指针的方法)

```
类型 T 方法集包含全部 receiver T 方法。
类型 *T 方法集包含全部 receiver T + *T 方法。
如类型 S 包含匿名字段 T，则 S 方法集包含 T 方法。
如类型 S 包含匿名字段 *T，则 S 方法集包含 T + *T 方法。
不管嵌入 T 或 *T，*S 方法集总是包含 T + *T 方法。
```

7、接口

7.1、结构定义

接口是一个或多个方法签名的集合，任何类型的方法集中只要拥有与之对应的全部方法，就表示它 "实现" 了该接口

所谓对应方法，是指有相同名称、参数列表 (不包括参数名) 以及返回值。当然，该类型还可以有其他方法

• 接口命名习惯以 er 结尾，结构体。

• 接口只有方法签名，没有实现。

• 接口没有数据字段。

• 可在接口中嵌入其他接口。

• 类型可实现多个接口

```
type Stringer interface {
    String() string
}

type Printer interface {
    Stringer // 接口嵌入。
    Print()
}
```

空接口 interface{} 没有任何方法签名，也就意味着任何类型都实现了空接口。其作用类似面向对象语言中的根对象 object。

匿名接口可用作变量类型，或结构成员。

```
type Tester struct {
     s interface {
        String() string
     }
}
```

当对interface变量进行判断是否为nil时，只有当动态类型和动态值都是nil，这个变量才是nil

7.2、接口转换

利用类型推断，可判断接口对象是否某个具体的接口或类型。

interface.(type)

```
if i, ok := o.(fmt.Stringer); ok { // ok-idiom
    fmt.Println(i)
}

// 不支持fallthrough
switch v := o.(type) {
case nil: // o == nil
    fmt.Println("nil")
case fmt.Stringer: // interface
    fmt.Println(v)
default:
    fmt.Println("unknown")
}
```

> *switch* 转换时不支持*fallthrough*

7.3、接口技巧

某些时候，让函数直接 "实现" 接口能省不少事。

```
type Tester interface {
    Do()
}
type FuncDo func()
func (self FuncDo) Do() { self() }
func main() {
    var t Tester = FuncDo(func() { println("Hello, World!") })
    t.Do()
}
```

8、并发

8.1、goroutine

Go 在语言层面对并发编程提供支持。添加 go 关键字，就可创建并发执行单元。

用以实现 "以通讯来共享内存" 的 CSP 模式。

可使用环境变量或标准库函数 runtime.GOMAXPROCS 修改使用cpu核数，默认全部使用。

调用 runtime.Goexit 将立即终止当前 goroutine 执行，调度器确保所有已注册 defer 延迟调用被执行。

和协程 yield 作用类似，Gosched 让出底层线程，将当前 goroutine 暂停，放回队列等待下次被调度执行。

8.2、channel

用于多个 goroutine 通讯。其内部实现了同步，确保并发安全。

> 一个*channel*同时仅允许被一个*goroutine*读写

默认为同步模式，需要发送和接收配对。

异步 channel 可减少排队阻塞，具备更高的效率。

向 closed channel 发送数据引发 panic 错误，接收立即返回零值。而 nil channel，无论收发都会被阻塞。

内置函数 len 返回未被读取的缓冲元素数量，cap 返回缓冲区大小。

8.2.1 channel原理

```
// src/runtime/chan.go:hchan
func makechan(t *chantype, size int) *hchan {
    var c *hchan
    c = new(hchan)
    c.buf = malloc(元素类型大小*size)
    c.elemsize = 元素类型大小
    c.elemtype = 元素类型
    c.dataqsiz = size

    return c
}
```

单向channel

> - *func readChan(chanName <-chan int)*：通过形参限定函数内部只能从*channel*中读取数据
> - *func writeChan(chanName chan<- int)*：通过形参限定函数内部只能向*channel*中写入数据

select可以监控多channel

```
select {
    case e := <- chan1 :
    fmt.Printf("Get element from chan1: %d\n", e)
    case e := <- chan2 :
    fmt.Printf("Get element from chan2: %d\n", e)
    default:
    fmt.Printf("No element in chan1 and chan2.\n")
    time.Sleep(1 * time.Second)
}
```

range 读取channel

```
for e := range chanName {
    fmt.Printf("Get element from chan: %d\n", e)
}
```

通过range可以持续从channel中读出数据，好像在遍历一个数组一样，当channel中没有数据时会阻塞当前goroutine，与读channel时阻塞处理机制一样。

9、内存管理

9.1、内存分配

高性能内存分配器有Google的tcmalloc

预申请的内存划分为spans、bitmap、arena三部分

arena的大小为512G，为了方便管理把arena区域划分成一个个的page，每个page为8KB,一共有512GB/8KB个页；

spans区域存放span的指针，每个指针对应一个page，所以span区域的大小为(512GB/8KB)*指针大小8byte = 512M

bitmap区域大小也是通过arena计算出来，不过主要用于GC。

```
// class  bytes/obj  bytes/span  objects  waste bytes
//     1          8        8192     1024            0
//     2         16        8192      512            0
//     3         32        8192      256            0
//     4         48        8192      170           32
...
//    65      28672       57344        2            0
//    66      32768       32768        1            0
```

> 每个*span*存放小对象（*class*），以*8*的倍数为单位，能存放*66*种小对象，超过*32K*，由特殊的*class*对待

9.2、垃圾回收

所谓垃圾就是不再需要的内存块，这些垃圾如果不清理就没办法再次被分配使用，标准与清除

三色标记法：

- 白色：对象未被标记，gcmarkBits对应的位为0（该对象将会在本次GC中被清理）
- 灰色：对象还在标记队列中等待
- 黑色：对象已被标记，gcmarkBits对应的位为1（该对象不会在本次GC中被清理）

初始状态下所有对象都是白色的。接下来就开始分析灰色对象，分析A时，A没有引用其他对象很快就转入黑色。最终，黑色的对象会被保留下来，白色对象会被回收掉。

垃圾回收优化（减少STW时间（Stop The World））

最开始的版本，Golang中的STW（Stop The World）就是停掉所有的goroutine，专心做垃圾回收，待垃圾回收结束后再恢复goroutine。

写屏障(Write Barrier)

而写屏障就是让goroutine与GC同时运行的手段。虽然写屏障不能完全消除STW，但是可以大大减少STW的时间。

> 写屏障类似一种开关，在GC的特定时机开启，开启后指针传递时会把指针标记，即本轮不回收，下次GC时再确定。

GC过程中新分配的内存会被立即标记，用的并不是写屏障技术，也即GC过程中分配的内存不会在本轮GC中回收。

辅助GC(Mutator Assist)

> 为了防止内存分配过快，在GC执行过程中，如果goroutine需要分配内存，那么这个goroutine会参与一部分GC的工作，即帮助GC做一部分工作，这个机制叫作Mutator Assist。

垃圾回收触发时机

内存分配量达到阀值触发GC

每次内存分配时都会检查当前内存分配量是否已达到阀值，如果达到阀值则立即启动GC。

```
阀值 = 上次GC内存分配量 * 内存增长率
```

内存增长率由环境变量`GOGC`控制，默认为100，即每当内存扩大一倍时启动GC。

定期触发GC

默认情况下，最长2分钟触发一次GC，这个间隔在`src/runtime/proc.go:forcegcperiod`变量中被声明

手动触发

程序代码中也可以使用`runtime.GC()`来手动触发GC。这主要用于GC性能测试和统计。

GC性能优化

GC性能与对象数量负相关，对象越多GC性能越差，对程序影响越大。

> 所以GC性能优化的思路之一就是减少对象分配个数，比如对象复用或使用大对象组合多个小对象等等。

9.3、内存分析

所谓逃逸分析（Escape analysis）是指由编译器决定内存分配的位置，不需要程序员指定。

> 1. 如果函数外部没有引用，则优先放到栈中；
> 2. 如果函数外部存在引用，则必定放到堆中；

指针逃逸

Go可以返回局部变量指针，这其实是一个典型的变量逃逸案例

```go
package main

type Student struct {
    Name string
    Age  int
}

func StudentRegister(name string, age int) *Student {
    s := new(Student) //局部变量s逃逸到堆

    s.Name = name
    s.Age = age

    return s
}

func main() {
    StudentRegister("Jim", 18)
}
```

函数StudentRegister()内部s为局部变量，其值通过函数返回值返回，`s本身为一指针，其指向的内存地址不会是栈而是堆`，这就是典型的逃逸案例。

通过编译参数-gcflags=-m可以查看编译过程中的逃逸分析

```
E:\SomeFile\gospace\prac\gcflag>go build  -gcflags=-m
# prac/gcflag
.\gcflag.go:16:6: can inline StudentRegister
.\gcflag.go:25:6: can inline main
.\gcflag.go:26:17: inlining call to StudentRegister
.\gcflag.go:16:22: leaking param: name
.\gcflag.go:17:10: new(Student) escapes to heap #逃逸到了堆
.\gcflag.go:26:17: new(Student) does not escape
```

栈空间不足逃逸

```
package main

func Slice() {
    s := make([]int, 1000, 1000)
    // s := make([]int, 10000, 10000) # 此时栈空间不足会逃逸到堆

    for index, _ := range s {
        s[index] = index
    }
}

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

```
**动态类型逃逸**
```

很多函数参数为interface类型，比如fmt.Println(a ...interface{})，编译期间很难确定其参数的具体类型，也人产生逃逸。

```
package main

import "fmt"

func main() {
    s := "Escape"
    fmt.Println(s) //interface 会逃逸到堆
}
```

闭包引用对象逃逸

```
func Fibonacci() func() int {
    a, b := 0, 1  // 闭包引用会将这两个值保存到堆上，内存逃逸
    return func() int {
        a, b = b, a+b
        return a
    }
}
```

> 闭包会保存外部函数局部变量值，局部变量的a和b由于闭包的引用，不得不将二者放到堆上，以致产生逃逸

total

- 栈上分配内存比在堆中分配内存有更高的效率
- 栈上分配的内存不需要GC处理
- 堆上分配的内存使用完毕会交给GC处理
- 逃逸分析目的是决定内分配地址是栈还是堆
- 逃逸分析在编译阶段完成

函数传递指针真的比传值效率高吗?

我们知道传递指针可以减少底层值的拷贝，可以提高效率，但是如果拷贝的数据量小，由于指针传递会产生逃逸，可能会使用堆，也可能会增加GC的负担，所以传递指针不一定是高效的。

> 结构体方法可用指针，函数参数尽可能用值传递

10、反射（自省）

1. 反射提供一种让程序检查自身结构的能力
2. 反射是困惑的源泉

反射包里有两个接口类型要先了解一下.

- `reflect.Type` 提供一组接口处理interface的类型，即（value, type）中的type
- `reflect.Value` 提供一组接口处理interface的值,即(value, type)中的value

反射三定律:

> 反射可以将*interface*类型变量转换成反射对象
>
> 反射可以将反射对象还原成*interface*对象
>
> 反射对象可修改，*value*值必须是可设置的

11、go test

11.1、单元测试

其中`unit.go`为源代码文件，`unit_test.go`为测试文件。要保证测试文件以"_test.go"结尾。

```
package gotest_test

import (
    "testing"
    "gotest"
)

func TestAdd(t *testing.T) {
    var a = 1
    var b = 2
    var expected = 3

    actual := gotest.Add(a, b)
    if actual != expected {
        t.Errorf("Add(%d, %d) = %d; expected: %d", a, b, actual, expected)
    }
}
```

测试函数名必须以"TestXxx"开始;

go test 启动测试即可

11.2、性能测试

```
package gotest_test

import (
    "testing"
    "gotest"
)

func BenchmarkMakeSliceWithoutAlloc(b *testing.B) {
    for i := 0; i < b.N; i++ {
        gotest.MakeSliceWithoutAlloc()
    }
}

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

- 文件名必须以“_test.go”结尾；
- 函数名必须以“BenchmarkXxx”开始；
- 使用命令“go test -bench=.”即可开始性能测试；

11.3、示例测试(godoc文档)

```
package gotest_test

import "gotest"

// 检测单行输出
func ExampleSayHello() {
    gotest.SayHello()
    // OutPut: Hello World
}

// 检测多行输出
func ExampleSayGoodbye() {
    gotest.SayGoodbye()
    // OutPut:
    // Hello,
    // goodbye
}
```

1. 例子测试函数名需要以"Example"开头；
2. 检测单行输出格式为“// Output: <期望字符串>”；
3. 检测多行输出格式为“// Output: \ <期望字符串> \ <期望字符串>”，每个期望字符串占一行；
4. 检测无序输出格式为"// Unordered output: \ <期望字符串> \ <期望字符串>"，每个期望字符串占一行；
5. 测试字符串时会自动忽略字符串前后的空白字符；
6. 如果测试函数中没有“Output”标识，则该测试函数不会被执行；
7. 执行测试可以使用`go test`，此时该目录下的其他测试文件也会一并执行；
8. 执行测试可以使用`go test <xxx_test.go>`，此时仅执行特定文件中的测试函数；

11.4、子测试

子测试提供一种在一个测试函数中执行多个测试的能力。子测试的一个方便之处在于可以让多个测试共享Setup和Tear-down。

```
package gotest_test

import (
    "testing"
    "gotest"
)

// sub1 为子测试，只做加法测试
func sub1(t *testing.T) {
    var a = 1
    var b = 2
    var expected = 3

    actual := gotest.Add(a, b)
    if actual != expected {
        t.Errorf("Add(%d, %d) = %d; expected: %d", a, b, actual, expected)
    }
}

// sub2 为子测试，只做加法测试
func sub2(t *testing.T) {
    var a = 1
    var b = 2
    var expected = 3

    actual := gotest.Add(a, b)
    if actual != expected {
        t.Errorf("Add(%d, %d) = %d; expected: %d", a, b, actual, expected)
    }
}

// sub3 为子测试，只做加法测试
func sub3(t *testing.T) {
    var a = 1
    var b = 2
    var expected = 3

    actual := gotest.Add(a, b)
    if actual != expected {
        t.Errorf("Add(%d, %d) = %d; expected: %d", a, b, actual, expected)
    }
}

// TestSub 内部调用sub1、sub2和sub3三个子测试
func TestSub(t *testing.T) {
    // setup code

    t.Run("A=1", sub1)
    t.Run("A=2", sub2)
    t.Run("B=1", sub3)

    // tear-down code
```

```
}
//func (t *T) Run(name string, f func(t *T)) bool
```

`Run()`会启动新的协程来执行`f`，并阻塞等待`f`执行结束才返回，除非`f`中使用`t.Parallel()`设置子测试为并发。

命令行下，使用`-v`参数执行测试

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\got
est>go test subunit_test.go -v
=== RUN   TestSub
=== RUN   TestSub/A=1
=== RUN   TestSub/A=2
=== RUN   TestSub/B=1
--- PASS: TestSub (0.00s)
    --- PASS: TestSub/A=1 (0.00s)
    --- PASS: TestSub/A=2 (0.00s)
    --- PASS: TestSub/B=1 (0.00s)
PASS
ok      command-line-arguments  0.354s
```

子测试命名规则

"<父测试名字>/<传递给*Run*的名字>"。比如，传递给`Run()`的名字是“A=1”，那么子测试名字为“TestSub/A=1”。这个在上面的命令行输出中也可以看出。

过滤筛选

只执行上例中“A=*”的子测试，那么执行时使用`-run Sub/A=`参数即可，

子性能测试则使用`-bench`参数来筛选

子测试并发

前面提到的多个子测试共享setup和teardown有一个前提是子测试没有并发，如果子测试使用`t.Parallel()`指定并发，那么就没办法共享teardown了，因为执行顺序很可能是setup->子测试1->teardown->子测试2...。

```
package gotest_test

import (
    "testing"
    "time"
)

// 并发子测试，无实际测试工作，仅用于演示
func parallelTest1(t *testing.T) {
    t.Parallel()
    time.Sleep(3 * time.Second)
    // do some testing
}

// 并发子测试，无实际测试工作，仅用于演示
func parallelTest2(t *testing.T) {
    t.Parallel()
    time.Sleep(2 * time.Second)
    // do some testing
}
```

```go
// 并发子测试，无实际测试工作，仅用于演示
func parallelTest3(t *testing.T) {
    t.Parallel()
    time.Sleep(1 * time.Second)
    // do some testing
}

// TestSubParallel 通过把多个子测试放到一个组中并发执行，同时多个子测试可以共享setup和tear-down
func TestSubParallel(t *testing.T) {
    // setup
    t.Logf("Setup")

    t.Run("group", func(t *testing.T) {
        t.Run("Test1", parallelTest1)
        t.Run("Test2", parallelTest2)
        t.Run("Test3", parallelTest3)
    })

    // tear down
    t.Logf("teardown")
}
```

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\gotest>go test subparallel_test.go -v -run SubParallel
=== RUN   TestSubParallel
=== RUN   TestSubParallel/group
=== RUN   TestSubParallel/group/Test1
=== RUN   TestSubParallel/group/Test2
=== RUN   TestSubParallel/group/Test3
--- PASS: TestSubParallel (3.01s)
        subparallel_test.go:25: Setup
    --- PASS: TestSubParallel/group (0.00s)
        --- PASS: TestSubParallel/group/Test3 (1.00s)
        --- PASS: TestSubParallel/group/Test2 (2.01s)
        --- PASS: TestSubParallel/group/Test1 (3.01s)
        subparallel_test.go:34: teardown
PASS
ok      command-line-arguments  3.353s
```

- 子测试适用于单元测试和性能测试；
- 子测试可以控制并发；
- 子测试提供一种类似table-driven风格的测试；
- 子测试可以共享setup和tear-down；

11.5、main测试

子测试的一个方便之处在于可以让多个测试共享Setup和Tear-down。但这种程度的共享有时并不满足需求，有时希望在整个测试程序做一些全局的setup和Tear-down，这时就需要Main测试了。

```go
// TestMain 用于主动执行各种测试，可以测试前后做setup和tear-down操作
func TestMain(m *testing.M) {
    println("TestMain setup.")

    retCode := m.Run() // 执行测试，包括单元测试、性能测试和示例测试

    println("TestMain tear-down.")

    os.Exit(retCode)
}
```

11.6、gotest工作机制

go test运行时，跟据是否指定package分为两种模式，即本地目录模式和包列表模式。

本地目录模式

当执行测试并没有指定package时，即以本地目录模式运行，例如使用"go test"或者"go test -v"来启动测试。

本地目录模式下，go test编译当前目录的源码文件和测试文件，并生成一个二进制文件，最后执行并打印结果。

包列表模式

当执行测试并显式指定package时，即以包列表模式运行，例如使用"go test math"来启动测试。

缓存机制

当满足一定的条件，测试的缓存是自动启用的，也可以显式的关闭缓存。

可缓存参数集合如下：

- -cpu
- -list
- -parallel
- -run
- -short
- -v

需要注意的是，测试参数必须全部来自这个集合，其结果才会被缓存，没有参数或包含任一此集合之外的参数，结果都不会缓存。

使用缓存结果

使用缓存结果也需要满足一定的条件：

- 本次测试的二进制及测试参数与之前的一次完全一致；
- 本次测试的源文件及环境变量与之前的一次完全一致；
- 之前的一次测试结果是成功的；
- 本次测试运行模式是列表模式

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src>go
test gotest
ok      gotest  3.434s

E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src>go
test gotest
ok      gotest  (cached)
```

禁用缓存

测试时使用一个不在“可缓存参数”集合中的参数，就不会使用缓存，比较常用的方法是指定一个参数“-count=1”。

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src>go
test gotest
ok      gotest  3.434s

E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src>go
test gotest
ok      gotest  (cached)

E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src>go
test gotest -count=1
ok      gotest  3.354s
```

第三次执行使用了参数"-count=1"，所以执行时不会从缓存中获取结果。

11.7、gotest参数

-args

指示go test把-args后面的参数带到测试中去。具体的测试函数会跟据此参数来控制测试流程。

-args后面可以附带多个参数，所有参数都将以字符串形式传入，每个参数做为一个string，并存放到字符串切片中。

-json

-json 参数用于指示go test将结果输出转换成json格式，以方便自动化测试解析使用。

-o

-o 参数指定生成的二进制可执行程序，并执行测试，测试结束不会删除该程序。

-bench regexp

> go test默认不执行性能测试，使用-bench参数才可以运行，而且只运行性能测试函数。

其中正则表达式用于筛选所要执行的性能测试。如果要执行所有的性能测试，使用参数"-bench ."或"-bench=."。

-benchtime s

-benchtime指定每个性能测试的执行时间，如果不指定，则使用默认时间1s。

例如，执定每个性能测试执行2s，则参数为："go test -bench Sub/A=1 -benchtime 2s"。

-cpu 1,2,4

-cpu 参数提供一个CPU个数的列表，提供此列表后，那么测试将按照这个列表指定的CPU数设置GOMAXPROCS并分别测试。

比如“-cpu 1,2”，那么每个测试将执行两次，一次是用1个CPU执行，一次是用2个CPU执行。 例如，使用命令"go test -bench Sub/A=1 -cpu 1,2,3,4" 执行测试：

```
BenchmarkSub/A=1                    1000          1256835 ns/op
BenchmarkSub/A=1-2                  2000           912109 ns/op
BenchmarkSub/A=1-3                  2000           888671 ns/op
BenchmarkSub/A=1-4                  2000           894531 ns/op
```

测试结果中测试名后面的-2、-3、-4分别代表执行时GOMAXPROCS的数值。

-count n

-count指定每个测试执行的次数，默认执行一次。

-failfast

默认情况下，go test将会执行所有匹配到的测试，并最后打印测试结果，无论成功或失败。

-failfast指定如果有测试出现失败，则立即停止测试。

-list regexp

-list 只是列出匹配成功的测试函数，并不真正执行。

-parallel n

指定测试的最大并发数。

当测试使用t.Parallel()方法将测试转为并发时，将受到最大并发数的限制，默认情况下最多有GOMAXPROCS个测试并发，其他的测试只能阻塞等待。

-run regexp

跟据正则表达式执行单元测试和示例测试。正则匹配规则与-bench 类似。

-timeout d

默认情况下，测试执行超过10分钟就会超时而退出。

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\got
est>go test -timeout=1s
TestMain setup.
panic: test timed out after 1s
```

- 按秒设置：-timeout xs或-timeout=xs
- 按分设置：-timeout xm或-timeout=xm
- 按时设置：-timeout xh或-timeout=xh

-v

默认情况下，测试结果只打印简单的测试结果，-v 参数可以打印详细的日志。

-benchmem

默认情况下，性能测试结果只打印运行次数、每个操作耗时。使用-benchmem则可以打印每个操作分配的字节数、每个操作分配的对象数。

```
func BenchmarkMakeSliceWithoutAlloc(b *testing.B) {
    for i := 0; i < b.N; i++ {
        gotest.MakeSliceWithoutAlloc() // 一次操作
    }
}

// 没有使用-benchmem
BenchmarkMakeSliceWithoutAlloc-4            2000            971191 ns/op

// 使用-benchmem
BenchmarkMakeSliceWithoutAlloc-4            2000            914550 ns/op
4654335 B/op          30 allocs/op
```

12、pprof 性能分析

pprof 就是用来解决这个问题的。pprof 包含两部分：

- 编译到程序中的 `runtime/pprof` 包
- 性能剖析工具 `go tool pprof`

CPU 性能分析

内存性能分析

阻塞性能分析

锁性能分析

数据分析

> 生成 *profile*，`go tool pprof` 分析这份数据，需要安装*Graphviz*

13、定时器

Go提供了两种定时器，此处分为一次性定时器、周期性定时器。

- 一次性定时器：定时器只计时一次，结束便停止；
- 周期性定时器：定时器周期性进行计时，除非主动停止，否则将永久运行；

13.1、timer

一定性定时器的使用方法及其使用原理

Timer实际上是一种单一事件的定时器，即经过指定的时间后触发一个事件，这个事件通过其本身提供的channel进行通知。

通过timer.NewTimer(d Duration)可以创建一个timer。

源码包`src/time/sleep.go:Timer`定义了Timer数据结构：

```
type Timer struct { // Timer代表一次定时，时间到来后仅发生一个事件。
    C <-chan Time
    r runtimeTimer
}
```

Timer对外仅暴露一个channel，指定的时间到来时就往该channel中写入系统时间，也即一个事件。

使用场景

设定超时时间

```
func WaitChannel(conn <-chan string) bool {
    timer := time.NewTimer(1 * time.Second)

    select {
    case <- conn:
        timer.Stop()
        return true
    case <- timer.C: // 超时
        println("WaitChannel timeout!")
        return false
    }
}
```

延迟执行某个方法

```
func DelayFunction() {
    timer := time.NewTimer(5 * time.Second)

    select {
    case <- timer.C:
        log.Println("Delayed 5s, start to do something.")
    }
}
```

简单接口

After()

有时我们就是想等指定的时间，没有需求提前停止定时器，也没有需求复用该定时器，那么可以使用匿名的定时器。

`func After(d Duration) <-chan Time`方法创建一个定时器，并返回定时器的管道，如下代码所示：

```
func AfterDemo() {
    log.Println(time.Now())
    <- time.After(1 * time.Second)
    log.Println(time.Now())
}
```

AfterDemo()两条打印时间间隔为1s，实际还是一个定时器，但代码变得更简洁。

AfterFunc()

前面我们例子中讲到延迟一个方法的调用，实际上通过AfterFunc可以更简洁。AfterFunc的原型为：

```
func AfterFunc(d Duration, f func()) *Timer
```

该方法在指定时间到来后会执行函数f。例如：

```
func AfterFuncDemo() {
    log.Println("AfterFuncDemo start: ", time.Now())
    time.AfterFunc(1 * time.Second, func() {
        log.Println("AfterFuncDemo end: ", time.Now())
    })

    time.Sleep(2 * time.Second) // 等待协程退出
}
```

AfterFuncDemo()中先打印一个时间，然后使用AfterFunc启动一个定器，并指定定时器结束时执行一个方法打印结束时间。

total

- time.NewTimer(d)创建一个Timer;
- timer.Stop()停掉当前Timer;
- timer.Reset(d)重置当前Timer;

13.2 ticker

Ticker是周期性定时器

Ticker的数据结构与Timer完全一致：

```
type Ticker struct {
    C <-chan Time
    r runtimeTimer
}
```

使用场景

简单定时任务

有时，我们希望定时执行一个任务，这时就可以使用ticker来完成。

下面代码演示，每隔1s记录一次日志：

```
// TickerDemo 用于演示ticker基础用法
func TickerDemo() {
    ticker := time.NewTicker(1 * time.Second)
    defer ticker.Stop()

    for range ticker.C {
        log.Println("Ticker tick.")
    }
}
```

定时聚合任务

```go
func TickerLaunch() {
    ticker := time.NewTicker(5 * time.Minute)
    maxPassenger := 30                    // 每车最大装载人数
    passengers := make([]string, 0, maxPassenger)

    for {
        passenger := GetNewPassenger() // 获取一个新乘客
        if passenger != "" {
            passengers = append(passengers, passenger)
        } else {
            time.Sleep(1 * time.Second)
        }

        select {
        case <- ticker.C:                 // 时间到，发车
            Launch(passengers)
            passengers = []string{}
        default:
            if len(passengers) >= maxPassenger {  // 时间没到，车已座满，发车
                Launch(passengers)
                passengers = []string{}
            }
        }
    }
}
```

> Ticker在使用完后务必要释放，否则会产生资源泄露，进而会持续消耗CPU资源，最后会把CPU耗尽。

错误示例

Ticker用于for循环时，很容易出现意想不到的资源泄露问题，下面代码演示了一个泄露问题：

```go
func WrongTicker() {
    for {
        select {
        case <-time.Tick(1 * time.Second):
            log.Printf("Resource leak!")
        }
    }
}
```

上面代码，select每次检测case语句时都会创建一个定时器，for循环又会不断的执行select语句，所以系统里会有越来越多的定时器不断的消耗CPU资源，最终CPU会被耗尽。

total

Ticker相关内容总结如下：

- 使用time.NewTicker()来创建一个定时器；
- 使用Stop()来停止一个定时器；
- 定时器使用完毕要释放，否则会产生资源泄露；

14、语法糖

最常用的语法糖莫过于赋值符`:=`，其次，表示函数变参的`...`。

...只能用于一维数据，或者最后一个参数解包