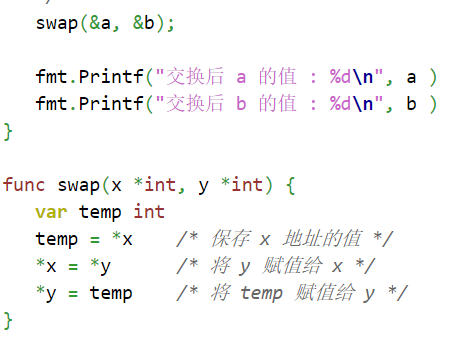
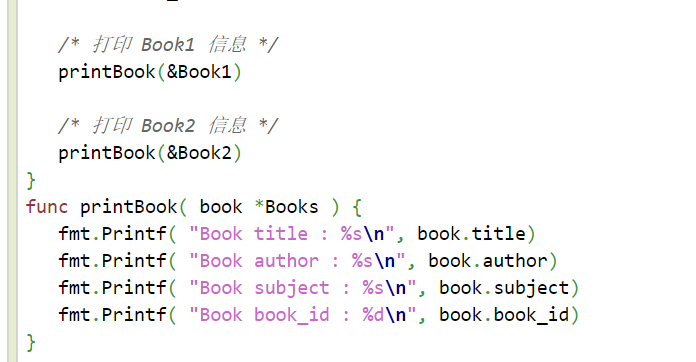


Println Printf

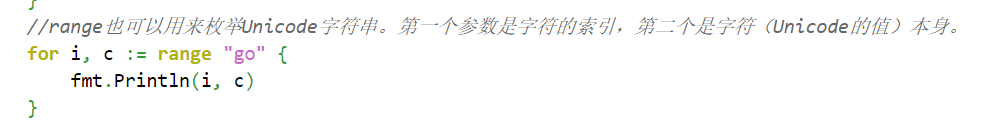


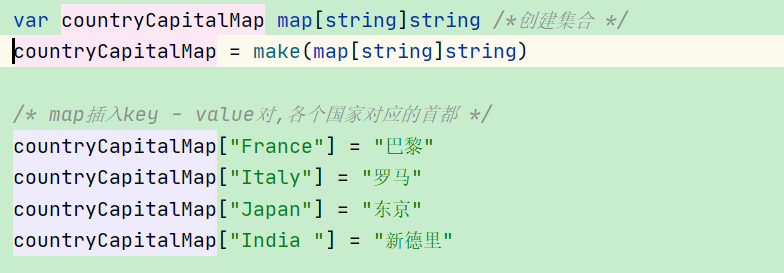
Const max int =3

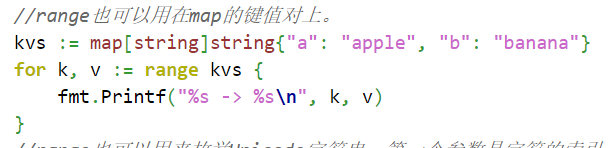




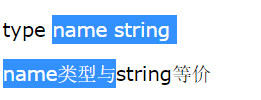
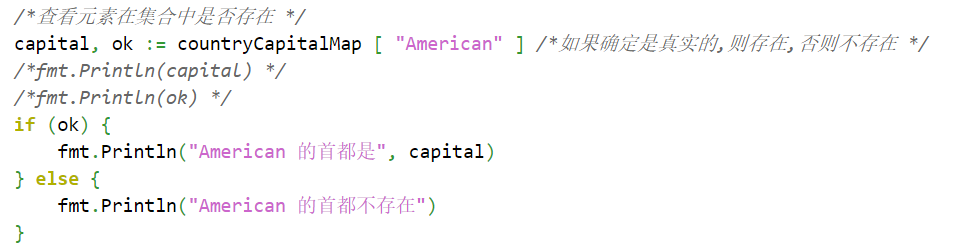








返回false



package main

import (

"fmt"

)

type Man interface {

name() string;

age() int;

}

type Woman struct {

}

//下面类似于struct的方法

func (woman Woman) name() string {

return "Jin Yawei"

}

func (woman Woman) age() int {

return 23;

}

type Men struct {

}

func ( men Men) name() string {

return "liweibin";

}

func ( men Men) age() int {

return 27;

}

func main(){

var man Man;

man = new(Woman);

fmt.Println( man.name());

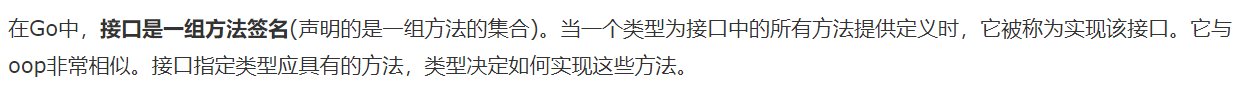
fmt.Println( man.age());

man = new(Men);

fmt.Println( man.name());

fmt.Println( man.age());

}



len 代表当前切片的长度，cap 是当前切片的容量

Go if变量定义在if语句内



Go语言的并发是基于 goroutine 的，goroutine 类似于线程，但并非线程。可以将 goroutine 理解为一种虚拟线程。Go语言运行时会参与调度 goroutine，并将 goroutine 合理地分配到每个 CPU 中，最大限度地使用 CPU 性能。

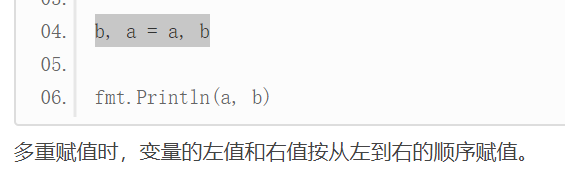
多个 goroutine 中，Go语言使用通道（channel）进行通信，通道是一种内置的数据结构，可以让用户在不同的 goroutine 之间同步发送具有类型的消息。这让编程模型更倾向于在 goroutine 之间发送消息，而不是让多个 goroutine 争夺同一个数据的使用权。

go func() {

.....

}()

以并发的方式调用匿名函数func



复数

Complex(x,y)

real()实部

Imag 虚部

String 多行定义 反引号

Unicode == chinese

ASCII 字符串长度使用 len() 函数。

Unicode 字符串长度使用 utf8.RuneCountInString() 函数。

ASCII 字符串遍历直接使用下标。

Unicode 字符串遍历用 for range。

，使用 fmt.Printf 的动词%p打印 cat 和 str 变量的内存地址，指针的值是带有0x十六进制前缀的一组数据。

// 打印ptr的类型

fmt.Printf("ptr type: %T\n", ptr)

// 打印ptr的指针地址

fmt.Printf("address: %p\n", ptr)

字符串与其它类型的转换，**strconv**包可以帮我们实现此功能

str := new(string)

\*str = "Go语言教程"

fmt.Println(\*str)

str := **strconv**.Itoa(num)

整形转字符

num2, err := strconv.Atoi(str2)

ParseBool() 函数用于将字符串转换为 bool 类型的值，它只能接受 1、0、t、f、T、F、true、false、True、False、TRUE、FALSE，其它的值均返回错误，函数签名如下。

boo1, err := strconv.ParseBool(str1)

func ParseInt(s string, base int, bitSize int)(i int64, err error)base 指定进制，取值范围是 2 到 36。**如果 base 为 0，则会从字符串前置判断，“0x”是 16 进制，“0”是 8 进制，否则是 10 进制。**bitSize 指定结果必须能无溢出赋值的整数类型，0、8、16、32、64 分别代表 int、int8、int16、int32、int64。

num, err := strconv.ParseInt(str, 10, 0)

num, err := strconv.ParseFloat(str, 64)

var a [3]int // 定义三个整数的数组

var r [3]int = [3]int{1, 2}

多维数组

array = [4][2]int{1: {20, 21}, 3: {40, 41}}



多维数组可以直接复制

Copy 覆盖复制

slice1 := []int{1, 2, 3, 4, 5}

slice2 := []int{5, 4, 3}

copy(slice2, slice1) // 只会复制slice1的前3个元素到slice2中

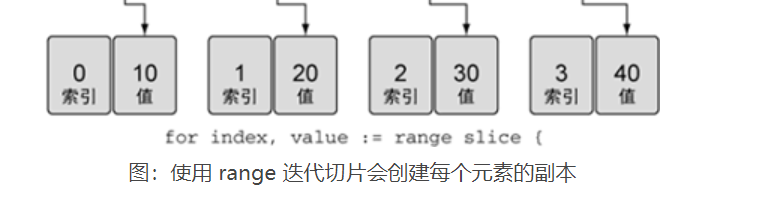
copy(slice1, slice2) // 只会复制slice2的3个元素到slice1的前3个位置

// 引用切片数据

refData := srcData

slice1[1:]... 表示从1位置开始到最后,以elem的形式，append的第二个参数只能是elem

Range迭代会创建副本



var mapLit map[string]int 字典

切片作为map的值

mp1 := make(map[int][]int)

mp2 := make(map[int]\*[]int)

delete(map, 键)

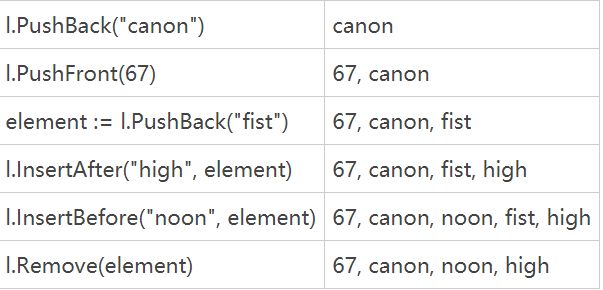
l := list.New()

l.PushBack("fist")

l.PushFront(67)

InsertAfter(v interface {}, mark \* Element) \* Element 在 mark 点之后插入元素，mark 点由其他插入函数提供

InsertBefore(v interface {}, mark \* Element) \*Element 在 mark 点之前插入元素，mark 点由其他插入函数提供



new 只分配内存，而 make 只能用于 slice、map 和 channel 的初始化

var sum \*int

sum = new(int) //分配空间

\*sum = 98

type Student struct {

name string

age int

}

var s \*Student

s = new(Student) //分配空间

s.name ="dequan"

fmt.Println(s)

make 分配空间后，会进行初始化

If语句

if condition1 {

// do something

} else if condition2 {

// do something else

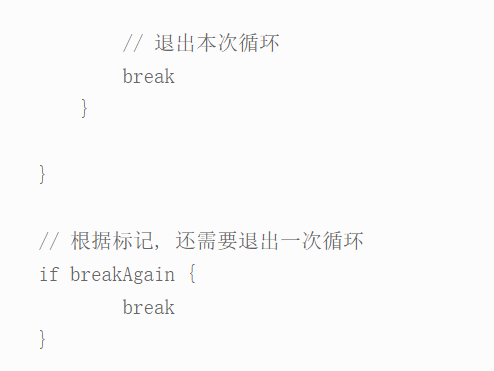
}else {

// catch-all or default

}

左花括号{必须与 for 处于同一行

break



inputReader := bufio.NewReader(os.Stdin) //string standard input

input, err := inputReader.ReadString('\n') //读入字符串，直到遇到换行符(包含)

input = strings.ToLower(input) 变小写

接口：<http://c.biancheng.net/view/58.html>

用 interface{} 传递任意类型数据是Go语言的惯例用法，使用 interface{} 仍然是类型安全的

Go语言的 defer 语句会将其后面跟随的语句进行延迟处理，在 defer 归属的函数即将返回时，将延迟处理的语句按 defer 的逆序进行执行，也就是说，先被 defer 的语句最后被执行，最后被 defer 的语句，最先被执行。（栈）

defer的函数在压栈的时候也会保存参数的值（但是保存的是参数的引用），并非在执行时取值。

func c() (i int) {

defer func() { i++ }()

return 1

}

i被return 变成1，执行defer，i+=1，i==2

Recover是一个从panic恢复的内建函数。Recover只有在defer的函数里面才能发挥真正的作用。如果当前的goroutine panic了，recover的调用将会捕获到panic的值，并且恢复正常执行。

当函数F调用了panic，F的执行会被停止，在F中panic前面定义的defer操作都会被执行，然后F函数返回。

使用 fmt.Errorf 加上 %w 格式符来生成一个嵌套的 error

err1 := errors.New("new error")

err2 := fmt.Errorf("err2: [%w]", err1)

err3 := fmt.Errorf("err3: [%w]", err2)

fmt.Println(err3)

func Factorial(n uint64) (result uint64) {

if n > 0 {

result = n \* Factorial(n-1)

return result

}

return 1

}

type Command struct {

Name string // 指令名称

Var \*int // 指令绑定的变量

Comment string // 指令的注释

}

struct实例

Cmd := &Command{} // cmd :=new(Command)

cmd.Name = “version”

Var version int = 1

Cmd.Var = &version

Func newCommand(name string,varref \*int,comment string) \*Command{

Return &Command{

Name: name,

Var:varref,

Comment:comment,

}

}

定义struct方法

type Property struct {

value int

}

func (p \*Property) SetValue(v int){

p.value = v

}

结构体内嵌

type A struct {

ax, ay int

}

type B struct {

A

bx, by float32

}

func main() {

b := B{A{1, 2}, 3.0, 4.0}

fmt.Println(b.ax, b.ay, b.bx, b.by)

fmt.Println(b.A)

}

结构内嵌初始化

type Wheel struct {

Size int

}

// 车

type Car struct {

Wheel

// 引擎

Engine struct {

Power int // 功率

Type string // 类型

}

}

c := Car{

// 初始化轮子

Wheel: Wheel{

Size: 18,

},

// 初始化引擎

Engine: struct {

Power int

Type string

}{

Type: "1.4T",

Power: 143,

},

}

Read() 方法的功能是读取数据，并存放到字节切片 p 中。Read() 执行结束会返回已读取的字节数

Readstring

r := strings.NewReader("hello world !")

reader := bufio.NewReader(r)

for {

str, err := reader.ReadString(byte(' ')) //以空格作为分界

fmt.Println(str)

if err != nil {

return

}

}

在Go语言中类型断言的语法格式如下：

value, ok := x.(T)

其中，x 表示一个接口的类型，T 表示一个具体的类型（也可为接口类型）。

该断言表达式会返回 x 的值（也就是 value）和一个布尔值（也就是 ok）

var x interface{}

x = 10

value, ok := x.(int)

fmt.Print(value, ",", ok)

%v the value in a default format

when printing structs, the plus flag (%+v) adds field names

%#v a Go-syntax representation of the value

%T a Go-syntax representation of the type of the value

判断类型

t,ok := i.(T)

var obj interface = new(bird)

f, isFlyer := obj.(Flyer)

因为结构内部不能包含方法

多种类型的比较

map 宕机错误，不可比较

切片（[]T） 宕机错误，不可比较 未定义容量为切片

通道（channel） 可比较，必须由同一个 make 生成，也就是同一个通道才会是 true，否则为 false

数组（[容量]T） 可比较，编译期知道两个数组是否一致

结构体 可比较，可以逐个比较结构体的值

函数 可比较

func Sqrt(f float64) (float64, error) {

if f < 0 {

return -1, errors.New("math: square root of negative number")

}

return math.Sqrt(f), nil

}

Go语言初始定位为高并发的服务器端程序

包的绝对路径就是GOROOT/src/或GOPATH/src/后面包的存放路径

如果它们的字段或方法首字母是大写，外部可以访问这些字段和方法

type MyStruct struct {

// 包外可以访问的字段

ExportedField int

// 仅限包内访问的字段

privateField int

}

并发主要由**切换时间片**来实现“同时”运行，并行则是直接利用多核实现多线程的运行，Go程序可以设置使用核心数，以发挥多核计算机的能力。

scan函数会识别空格左右的内容，哪怕换行符号存在也不会影响scan对内容的获取。

scanln函数会识别空格左右的内容，但是一旦遇到换行符就会立即结束，不论后续还是否存在需要带输入的内容。

go func( 参数列表 ){

函数体

}( 调用参数列表 )

runtime.Gosched()用于让出CPU时间片。这就像跑接力赛，A跑了一会碰到代码runtime.Gosched()就把接力棒交给B了，A歇着了，B继续跑。

WaitGroup 对象内部有一个计数器，最初从0开始，它有三个方法：Add(), Done(), Wait() 用来控制计数器的数量。Add(n) 把计数器设置为n ，Done() 每次把计数器-1 ，wait() 会阻塞代码的运行，直到计数器地值减为0。

func main() {

wg := sync.WaitGroup{}

wg.Add(100)

for i := 0; i < 100; i++ {

go f(i, &wg)

}

wg.Wait()

}

// 一定要通过指针传值，不然进程会进入死锁状态

func f(i int, wg \*sync.WaitGroup) {

fmt.Println(i)

wg.Done()

}

在go build命令中多加了一个-race 标志，这样生成的可执行程序就自带了检测资源竞争的功能，运行生成的可执行文件

channel 就是一种队列一样的结构,FIFO

ch1 := make(chan int) // 创建一个整型类型的通道

ch2 := make(chan interface{}) // 创建一个空接口类型的通道, 可以存放任意格式

type Equip struct{ /\* 一些字段 \*/ }

ch2 := make(chan \*Equip) // 创建Equip指针类型的通道, 可以存放\*Equip

阻塞模式接收数据时，将接收变量作为<-操作符的左值，格式如下：

data := <-ch

执行该语句时将会阻塞，直到接收到数据并赋值给 data 变量。

非阻塞接收数据

使用非阻塞方式从通道接收数据时，语句不会发生阻塞，格式如下：

data, ok := <-ch

data：表示接收到的数据。未接收到数据时，data 为通道类型的零值。

ok：表示是否接收到数据。

ch := make(chan int)

// 声明一个只能写入数据的通道类型, 并赋值为ch

var chSendOnly chan<- int = ch

//声明一个只能读取数据的通道类型, 并赋值为ch

var chRecvOnly <-chan int = ch

与 switch 语句相比，select 有比较多的限制，其中最大的一条限制就是每个 case 语句里必须是一个 IO 操作，大致的结构如下：

调用 time.AfterFunc() 函数，传入等待的时间和一个回调。回调使用一个匿名函数，在时间到达后，匿名函数会在另外一个 goroutine 中被调用。time.AfterFunc() 函数是在 time.After 基础上增加了到时的回调，方便使用。

time.AfterFunc(time.Second, func() {

// 1秒后, 打印结果

fmt.Println("one second after")

// 通知main()的goroutine已经结束

exit <- 0

})

缓冲通道在关闭后依然可以访问内部的数据。

运行结果第三行的“0 false”表示通道在关闭状态下取出的值。0 表示这个通道的默认值，false 表示没有获取成功

defer // 在函数退出时解除锁定

这个成员类型后面带有一个以 ` 开始和结尾的字符串。这个字符串在Go语言中被称为 Tag（标签）。一般用于给字段添加自定义信息

golang os.OpenFile

os.O\_WRONLY | os.O\_CREATE | O\_EXCL 【如果已经存在，则失败】

os.O\_WRONLY | os.O\_CREATE 【如果已经存在，从头直接覆盖写】

os.O\_WRONLY | os.O\_CREATE | os.O\_APPEND 【如果已经存在，则在尾部添加写】

文本写入

writer := bufio.NewWriter(file)

for i := 0; i < 3; i++ {

writer.WriteString(str)

}

//因为 writer 是带缓存的，因此在调用 WriterString 方法时，内容是先写入缓存的

//所以要调用 flush方法，将缓存的数据真正写入到文件中。

writer.Flush()

文本读入

reader := bufio.NewReader(file)

for {

str, err := reader.ReadString('\n') //读到一个换行就结束

if err == io.EOF { //io.EOF 表示文件的末尾

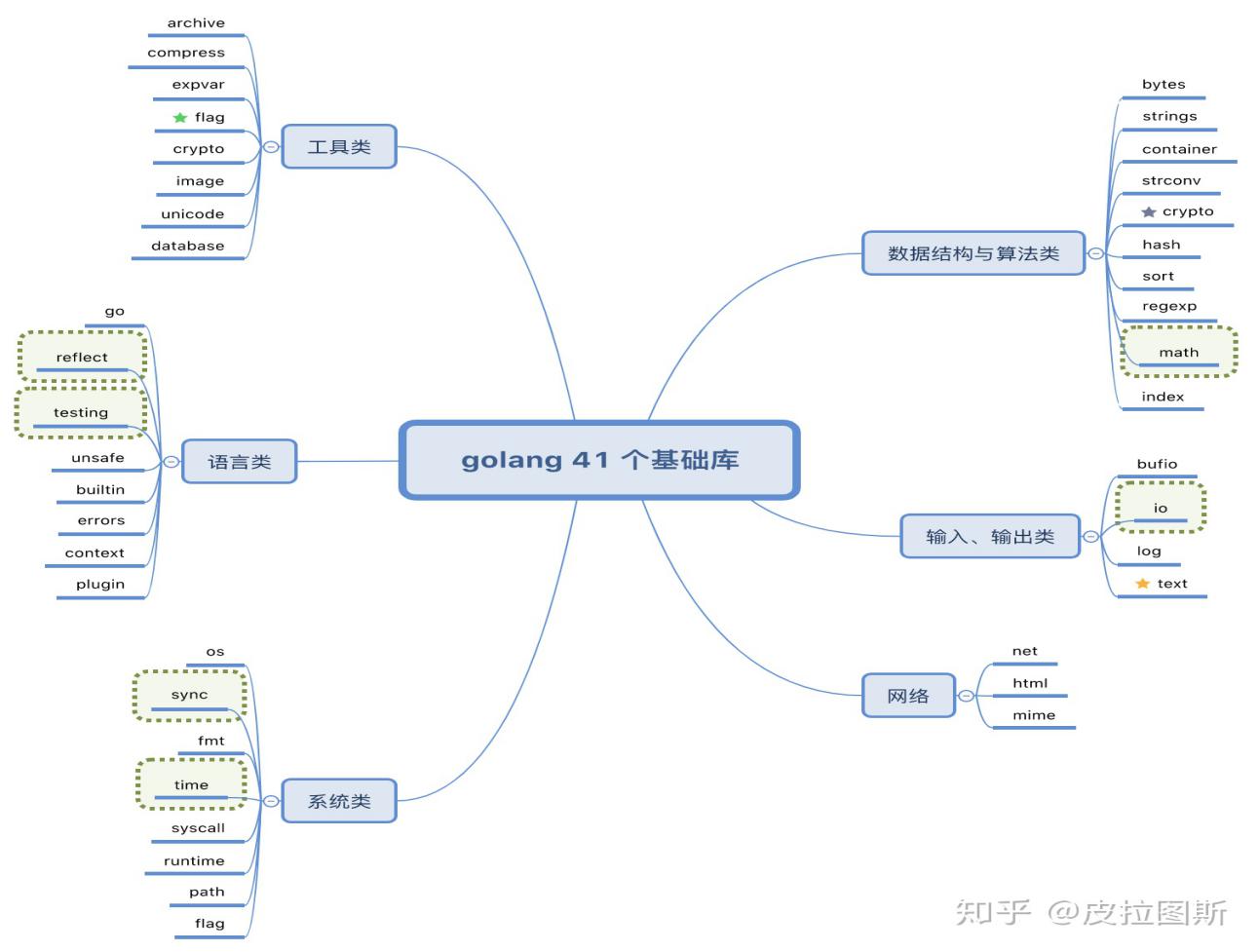
break

}

fmt.Print(str)

}

go build+文件列表”方式编译时，可执行文件默认选择文件列表中第一个源码文件作为可执行文件名输出。go build -o myexec main.go lib.go build 和文件列表之间插入了-o myexec参数，表示指定输出文件名为 myexec。



1. Go中的数组是值类型，换句话说，如果你将一个数组赋值给另外一个数组，那么，实际上就是将整个数组拷贝一份

2. 如果Go中的数组作为函数的参数，那么实际传递的参数是一份数组的拷贝，而不是数组的指针。这个和C要区分开。因此，在Go中如果将数组作为函数的参数传递的话，那效率就肯定没有传递指针高了

**[5] int { 2:1,3:2,4:3}**

**长度为5的数组，key:value,其元素值依次为：0，0，1，2，3**

数组的长度不可改变，在特定场景中这样的集合就不太适用，Go中提供了一种灵活，功能强悍的内置类型Slices切片(“动态数组"),与数组相比切片的长度是不固定的，可以追加元素，在追加时可能使切片的容量增大。切片中有两个概念：一是len长度，二是cap容量，长度是指已经被赋过值的最大下标+1，可通过内置函数len()获得。容量是指切片目前可容纳的最多元素个数，可通过内置函数cap()获得。切片是引用类型，因此在当传递切片时将引用同一指针，修改值将会影响其他的对象。

深度拷贝: copy(sliceA, sliceB)

浅拷贝: sliceA = sliceB

但是，当涉及到 append 触发扩容时，原来的指针指向的地址会发生变化，之后再对数组值进行更改，原切片将不受影响.当使用 append(slice,data) 时候，Golang 会检查底层的数组的长度是否已经不够，如果长度不够，Golang 则会新建一个数组，把原数组的数据拷贝过去，再将 slice 中的指向数组的指针指向新的数组。

‘…’ 其实是go的一种语法糖。

它的第一个用法主要是用于函数有多个不定参数的情况，可以接受多个不确定数量的参数。

第二个用法是slice可以被打散进行传递。

go func(msg string) {

fmt.Println(msg)

}("going")

匿名函数

命令源码文件如何接收参数？

Go 语言标准库中有一个代码包专门用于接收和解析命令参数，即flag包

flag.StringVar

inputReader := bufio.NewReader(os.stdin)声明并初始化带缓冲的读取器

&^ 常用于按位对齐，a&^b 清除a中ab都为1的1

测试

go test -cover （在当前目录，测所有test文件）

go test --cover mul\_test.go mul.go 指定当前目录的某个文件#再加-v，详细显示

进行测试，并输出html

go test -coverprofile test.cover

go tool cover -html=test.cover -o coverage.html

go test -coverpkg=./... -coverprofile=coverage.data -timeout=5s ./...

go tool cover -html=coverage.data -o coverage.html

go test --cover mul\_test.go mul.go

// 输出测试日志

t.Logf()

t.Logf()

// 标记错误，但仍然执行后面的语句

t.Fail()

// 获取是否当前用例是执行错误的

t.Failed()

// 错误输出，等于 t.Logf 再执行 t.Fail()

t.Errorf("%s", "run ErrorF")

// 标记函数错误，并中断后面的执行

t.FailNow()

// 致命错误输出，等同于调用了 t.Logf 然后调用 t.FailNow()

t.Fatalf("%s", "run Fatelf")

log.Fatal函数完成：

打印输出内容

退出应用程序

defer函数不会执行

os.Exit()

应用程序马上退出。

defer函数不会执行。

log.Fatal函数完成

打印输出内容

退出应用程序

defer函数不会执行

Project取名 test.cn

不同文件夹调用是调用文件夹名

Import projec名/文件夹名

如果包外不需要访问请用小写开头的函数

如果需要暴露出去给包外访问需要使用大写开头的函数名称

函数命名使用camelcase

%x 十六进制，小写字母，每字节两个字符 Printf("%x", "golang") 676f6c616e67

%X 十六进制，大写字母，每字节两个字符 Printf("%X", "golang") 676F6C616E67

**当return后面为空是，函数声明时的 (a int) 会把 a 作为返回值，当 return 不为空时，会把 return 后面的值作为返回值**

//go官方推荐写法

func myfunc02() (a int, b int, c int) {

    a, b, c = 111, 222, 333

    return

}

[]byte 转 string

String(source)

String 转byte

[]byte(source)

uint8类型，或者叫 byte 型，代表了ASCII码的一个字符。

rune类型，代表一个 UTF-8字符。

ASCII编码是1个字节，而UTF-8是可变长的编码，当要表示中文等非ASCll编码的字符时，需要使用UTF-8编码来保证不会乱码。

pprof 是一个强大的性能分析工具，可以捕捉到多维度的运行状态的数据

profit[0][0][0], profit[0][0][1] = 0, -prices[0]

runtime.stack

func Stack(buf []byte, all bool) int

Stack将调用其的go程的调用栈踪迹格式化后写入到buf中并返回写入的字节数。若all为true，函数会在写入当前go程的踪迹信息后，将其它所有go程的调用栈踪迹都格式化写入到buf中。

每隔30ms重绘界面:

for range time.Tick(30 \* time.Millisecond) {

repaint()}

每隔一秒 输出时间

tiker := time.Tick(time.Second)  
for t := range tiker {  
 fmt.Println(t)

make([]int, 2, 4) 第三种用法，第二参数指定的是切片的长度，第三个参数是用来指定预留的空间长度

Reflect 简单的来说反射就是程序在运行时，通过检查其定义的变量以及值，进而找到其对应的真实类型。

type order struct{

ordId int

customerId int}

func query(q interface{}) {

t := reflect.TypeOf(q)

v := reflect.ValueOf(q)

fmt.Println("Type ", t)

fmt.Println("Value ", v)}

func main(){

o := order{

ordId: 456,

customerId: 56,

}

query(o)}

Type main.order

Value {456 56}

Outloop:

Continue outloop

outloop: for \_, v1 := range words {  
 // 拿每一个单词。  
 for \_, v2 := range v1{  
 // 取字母  
 if strings.Count(v1, string(v2)) > strings.Count(chars, string(v2)) {  
 // 如果在单词中某个字母的数量大于词库中的数量，说明弄不了。  
 continue outloop  
 }

统计字符串v1中v2字符串出现的次数

strings.Count(v1,v2)

make(map[string]int) //表示创建一个key为string，value的值为int的数据类型

go除法为向下整除

20/3 = 6

sort.Ints(ints)

升序排序

FormatFloat()参数众多：

func FormatFloat(f float64, fmt byte, prec, bitSize int) string

bitSize表示f的来源类型（32：float32、64：float64），会据此进行舍入。

fmt表示格式：'f'（-ddd.dddd）、'b'（-ddddp±ddd，指数为二进制）、'e'（-d.dddde±dd，十进制指数）、'E'（-d.ddddE±dd，十进制指数）、'g'（指数很大时用'e'格式，否则'f'格式）、'G'（指数很大时用'E'格式，否则'f'格式）。

prec控制精度（排除指数部分）：对'f'、'e'、'E'，它表示小数点后的数字个数；对'g'、'G'，它控制总的数字个数。如果prec 为-1，则代表使用最少数量的、但又必需的数字来表示f。

Monkey的使用方法

调用monkey.Patch(<target function>, <replacement function>)以替换功能

monkey.Unpatch(<target function>)撤销

如果要修补实例方法，则需要使用monkey.PatchInstanceMethod(<type>, <name>, <replacement>)

用reflect.TypeOf获取类型

monkey.UnpatchInstanceMethod(<type>, <name>)撤销

monkey.UnpatchAll去掉所有

go test -gcflags=-l 禁用内联

func Replace(s, old, new string, n int) string

n<0 替换所有 n>0替换前n个，返回新字符串

[Go的json解析：Marshal与Unmarshal](https://www.cnblogs.com/haiguixiansheng/p/10718531.html)

type Class struct {

    Name  string

    Grade int

}

cla := new(Class)   //这个new方法，相当于cla:=&Class{}，是一个取地址的操作

Go json转化、struct中 ``表示tag，转化成json格式时，会用tag里的替代原有key

Eg：Name  string `json:"name"`

<https://www.cnblogs.com/haiguixiansheng/articles/10718531.html>

file,err:=os.Create("a.txt")

//创建目录和权限 os.Mkdir("./benben",0777)

//创建多级目录和设置权限 os.MkdirAll("./benben/test",0777)

//删除目录 err:=os.Remove("./benben")

//删除多级目录 os.RemoveAll("./benben")

sync.Once.Do(f func())是一个挺有趣的东西,能保证once只执行一次，无论你是否更换once.Do(xx)这里的方法,这个sync.Once块只会执行一次。

切片排序规则自定义

sort.Slice(people,func(i,j int)bool{  
 return (people[i][0]>people[j][0]) || (people[i][0]==people[j][0] && people[i][1]<people[j][1])  
})

Rest接口时

Put 是idempotent ,post不是。都可以用来创建or更新一个方法

假如对同一个接口发送两个请求，产生了两个不同的结果，而不是后一个覆盖前一个结果，则应该用post

Var (map)只声明了类型，需要再make一下给空间