go路线

# 为什么要go&go概览&go安装和相关工具

## 简史

从贝尔实验室到google

## 语言特点

自动垃圾回收

更丰富的内置类型

函数多返回值

错误处理

匿名函数和闭包

类型和接口

并发编程

反射

语言交互性

工厂管理

问题追溯和调试

## 设计初衷

针对其他语言的痛点进行设计

并加入并发编程

为大数据，为服务，并发而生的通用编程语言

## 互联网时代的C考虑的关键问题

### 并行与分布式支持

A. **并发执行的"执行体"**

**操作系统掌管进程**

**进程内的线程**

**进程内的协程（轻量级线程）**

1.传统的通过库方式支持的协程功能,仅支持到协程的创建、销毁、切换

如果要实现同步IO操作（网络通信、本地文件读写），都会阻塞其他的并发执行协程

2.go语言级别支持协程，叫goroutine

go语言标准库提供的所有系统调用操作都会让CPU给其他goroutine，例如所有的同步IO操作

1. **执行体间的通信**
2. **执行体之间的互斥与同步**
3. **执行体之间的消息传递**

1.共享资源的相关执行体进行互斥

执行体间存在时序依赖时，执行体之间进行同步

2.多数语言提供了线程间的互斥和同步

执行体之间的消息传递，并发编程模型的两种选择：共享内存、消息传递

go执行体之间相互传递消息是基于一个消息队列（通道channel

### 软件工程支持

**代码风格规范**

**错误处理规范**

**包管理**

**契约规范（接口）**

**单元测试规范**

**功能开发的流程规范**

1.public的变量必须以大写字母开头

2.private变量则以小写字母开头

3.对{ }应该怎么写进行了强制

4.错误处理规范defer、error

### 编程哲学的重塑

A.

反对函数和重载

重载虽然解决了OOP的问题，但是带来了负担

B.

没有“对象”

没有继承多态

没有泛型

没有try/catch

没有构造函数

没有析构函数

D.有

非侵入式接口

函数式编程

csp并发模型（goroutine + channel)

• Go语言中的接口与其他语言最大的一点区别是它的非侵入性。在C++、 Java和C#中，为了实现一个接口，你需要从该接口继承

• Go语言的非侵入式接口：

其一， Go语言的标准库再也不需要绘制类库的继承树图。你只需要知道这个类实现了哪些

方法，每个方法是啥含义就足够了；

其二，不用再纠结接口需要拆得多细才合理；

其三，不用为了实现一个接口而专门导入一个包，而目的仅仅是引用其中的某个接口的定义

• 支持匿名函数与闭包

• 接受了以Erlang语言为代表的面向消息编程思想

## 概览

A.基本函数

变量、选择、循环、指针、数组、容器

B.面向接口

结构体、duck typing的概念、组合的思想

C.函数式编程

闭包、多样

D.工程化

资源管理、错误处理、测试和文档、性能调优

E.并发编程

goroutine、channel、调度器

## 安装和相关工具

IDE:golang、liteIDE

## GOPATH环境遍历

• 默认在~/go(unix,linux),%USERPROFILE%\go(windows)

官方推荐：所有项目和第三方库都在同一个GOPATH下

• 也可以不同的GOPATH

• go\bin加到path

获取整理import的工具

go -->on save --> go import

go get golang.org/x/tools/cmd/goimports

• 使用 gopm拉去镜像

go get -v github.com/gpmgo/gopm

gopm -get -g -v get golang.org/x/tools/cmd/goimports

• 使用 gopm拉去镜像

go get -v github.com/gpmgo/gopm

gopm -get -g -v -u golang.org/x/tools/cmd/goimports

• gopm只是把代码包拉下来了

go build src/golang.org/x/tools/cmd/goimports

• go install golang.org/x/tools/cmd/goimports

会做两件事build好后放到bin下面

• 目录下只能有一个main文件

• go instal ./... ...代表所有文件

# 基本规范

## 注释

A.块注释/\* \*/

/\*

块注释

可以注释多行

\*/

B.行注释

//行注释

## 包声明

一个包下的文件包名要一致

## 导入包

import "fmt" //导入包

## 主程序入口函数main

主函数 入口函数

func 函数名 （参数列表）返回值 {

函数体

}

func main() {

fmt.Printf("Hello")

}

1.主函数vs普通函数 主函数没有返回值

2.要求花括号和函数名在一行内

3.在同一个目录下 不能有重名的函数 除了主函数

## 命名规则

1.允许使用数字、字母、下划线、 不能使用%@

2.不允许使用关键字 25和关键字 30多个预定于的

3.不允许数字开头 （文件名除外）

4.区分大小写的

5.见名知意

## 驼峰命名法

1.小驼：userName

2.大驼峰：UserName

3.下划线连接：user\_name

## 输出

比较重要的几个格式化输出

%T 值的数据类型

%c 字符

%d 整型

%f 浮点数 默认会有六位小数 不够0补全 %.2f只保留两位小数

%t true或false输出的布尔值

%s 字符串

%v 自动匹配格式输出

%q会把string的quote打印出

## 输入

fmt.Scanf("%d", &a)

数据类型 &a是取地址运算符 把用户输入的数据给到a变量

更简单的输入方式

fmt.Scan(&a) //把用户输入的给到变量

fmt.Scan(&a)

# 变量

## 变量的声明

变量的初始化和赋值

只是声明没有初始化 会有默认值

在同一个{} 声明的变量要唯一

### var声明

声明 var 变量名 类型

var e float64 = 3.14 //初始化 一步到位

var s string = "xxxx"

### 自动推导类型

:= 左边变量名没用过 //只能用于第一次初始化

### 多重赋值

m, n := 10, 20

var cc, dd = 1, "ddddd" //这种定义不是一种数据类型也可以写在一行自动推导

x, y, z, ll := 10, 20, 3.33, "aaa"

var (

x3 = 1

x4 = 2.0

)

## 包外变量

A.

函数外部定义变量,需要关键字var

var (

aa2 = 3

bb = 4

)

var a, b, c bool

var d = 10

var (

dd = 10

aa = true

)

## 匿名变量

\_匿名变量 丢弃数据不处理

temp, \_ := 7, 8

println(temp)

不需要第三个值

var l, p, \_ = test()

println(l, p)

## 数据交换

n, m = m, n

cpu分三层 计算+执行+寄存

寄存器存放数据 go直接在寄存器交换数据

# 常量和枚举

## 常量

程序运行期间 不可以改变的量 声明用关键字 const

A.基本用法：

const i int = 10

const j float64 = 3.14

fmt.Println(i, j)

const (

i2 int = 10

j2 float64 = 3.14

)

const u, v float32 = 0, 3

const a, b, c = 3, 4, "foo"

B.自动推倒类型

const (

i3 = 10

j3 = 3.14

)

## 枚举

A.

iota给常量赋值使用

const (

x = iota

y = iota

z = iota

)

B.

枚举 iota 常量自动生成器，每个一行 自动加1

如果在同一行 ，值是一样的

const (

i = iota //0

j1, j2, j3 = iota, iota, iota //1，1，1，

k = iota //2

)

C.

iota遇到const,重置为0

const d = iota

D.

const (

b = 1 << (10 \* iota) //1 iota作为子增值的种子

kb //1024

mb //

gb

tb

pb

)

E.

可以只写一个iota 在一个括号里

const (

a1 = iota

\_

b1

c1

)

# 基础类型

## bool

占一个字节 没有初始化 零值为false

bool类型不接受其他类型的赋值，不支持强制类型转换

## string

var str1 string

str1 = "abc"

## int

(u)int

(u)int8

(u)int16

(u)int32

(u)int64

uintptr

加u无符号整数 不加u有符号整数

不规定长度的根据系统来32 64

uintptr指针

零值为0 占4或8字节（根据操作系统决定）

8位1字节

var x5 int32 = 10 //不指定就是int64

str = "Hello world" // 字符串赋值

ch := str[0] // 取字符串的第一个字符

fmt.Printf("The length of \"%s\" is %d \n", str, len(str))

fmt.Printf("The first character of \"%s\" is %c.\n", str, ch)

## byte(int8) 字符类型

单引号 代表UTF-8字符

var m byte = 'a'

1.fmt.Println(m) //输出97，计算机只能识别数字，用asci对应

2.大写字符（A）+32得到小写字符(a)

A== 65(asci) a==97 0==48

asci码表0~255

uint8 : byte字符类型是uint8的别名 值范围0~255

3.%c 表示占位符号 表示打印输出一个字符

fmt.Printf("%c", m) //a

fmt.Printf("%c", 97) //a

fmt.Println("%T\n", m) //得到的是unit8而不是byte

4. 输出\n \t

fmt.Println('\n') //输出10 \t 输出是9

5.\0的最用是区分字符串和字符（字符串末尾的结束标志） 字符空--asci码0

## rune(int32)字符类型

代表单个Unicode字符

Unicode/utf8包提供了两者之间的转换

Go语言的多数API都假设字符串为UTF-8编码。尽管Unicode字符在标

准库中有支持，但实际上较少使用

## float

float32 float64

单精度浮点型float32 和 双精度浮点型float64 默认为0.0

float32 小数点后7位，float64 15位

自动推到 创建的变量 默认为float64

## 复数

complex64 占8字节 两个float32组成

complex128 占16字节 两个float64组成

定义

var v1 complex64

v1 = 2.1 + 3.14i

v2 := 2.2 + 3.3i

通过内建函数取实部和虚部

fmt.Println(real(v2), imag(v2))

使用计算

v3 := 3 + 4i

fmt.Println(cmplx.Abs(v3))

一个牛逼哄哄的欧拉公式

fmt.Println(cmplx.Pow(math.E, 1i\*math.Pi) + 1) //这种方式的是通用

fmt.Println(cmplx.Exp(1i\*math.Pi) + 1) //Exp底数就是1，只用写指数 输出的不是1，而是1.2246467991473532e-16j

fmt.Printf("%.3f", cmplx.Exp(1i\*math.Pi)+1) //小数点后三位（0.000+0.000i)

## byte和string比较

ch := 'a'

str3 := "a"

'a''\0'组成的 \0字符串结束标志，打印的时候是打印\0前面的字符

len函数 计算字符串中字符的个数

fmt.Println(len(str1)) //输出3 不包含\0

fmt.Println(len(str2)) //输出12 go语言一个汉字占3个字符，为了和linux统一处理

## string包中常用的方法

strings.xxx

Fields,Split,Join

Contains,Index

ToLower,ToUpper

Trim,TrimRigth,TrimLeft

## rune和string比较

支持国际化多语言，rune相当于go的char

s:="yes我爱吃面条!"

len（s) = 19

printf %s打印出来内容

printf %X打印出来字节的十六进制数字

一个中文三个字节的UTF码，

一个英文一字节的UTF码，UTF是可变长编码

unicode是四字节十六进制

for i,ch := range s{

fmt.Printf( "%d %X", i,ch)

}

把string 做了一个utf-8的解码

解出来的字符转unicode,放到rune4个字节类型里面

rune是int32 一个别名

## rune库函数

### 统计长度

utf8.RuneCountInString(s)

### 遍历字符串输出

方法一：比较底层 每三个字节做一个解码

bytes := []byte(s)

for len(bytes) >0{

ch,size := utf8.DecodeRune(bytes)//utf8转uncode

bytes = bytes[size:]//每次获取到一个元素就把这个元素从切片中移除

fmt.printf("%c ",ch)

}

方法二：

或着

bytes := []rune(s) //这个类型转换是转换已经decode出来的结果，每个字符占了四个字节，重新开了一个rune数组把他们存起来

for i,ch := rnage []rune(s){

fmt.printf("%d %c ",i,ch)

}

## 案例：寻找最长不含有重复字符串的子串

国际化版

func lengthOfNonRepeatingSubStr(s string) int {

lastOccurred := make(map[rune]int)

start := 0

maxLength := 0

for i, ch := range []rune(s) {

if lastI, ok := lastOccurred[ch]; ok && lastI >= start {

start = lastI + 1

}

if i-start+1 > maxLength {

maxLength = i - start + 1

}

lastOccurred[ch] = i

}

return maxLength

}

# 运算符

## 算术运算符

+

-

\*

/

%取模 取余

++ 后自增，没有前自增

-- 后自减，没有前自减

## 关系运算符

==

!= <

>

<=

>=

## 逻辑运算符

! 非

&& 与

或

## 位运算符

& 按位与

| 按位或

^ 异或

<< 左移

>> 右移

## 赋值运算符

= 普通赋值

+= 相加后赋值

-= 相减后赋值

\*= 相乘后赋值

/= 相除后赋值

%= 求余后赋值

<<= 左移后赋值

>>= 右移后赋值

&= 按位与后赋值

^= 按位异或后赋值

!= 按位或后赋值

## 地址操作

& 取址运算符 &a查看变量a的地址

\* 取值运算符 变量指向内存中的值

## 地址表示

%p 已16进制表示地址

fmt.Println("%p\n",&a)

fmt.Println(a)

fmt.Printf("%d\n",\*&a) //这个内存地址中的值

## 运算符优先级

括号提升整体优先级

一元运算优先级高

二元运算符的运算方向从左至右

# 流程控制

## 条件语句if

A.

if b1 >= a1 {

fmt.Println("恭喜你2")

}

B.

支持一个初始化语句，初始化语句和判断条件用 ； 分割

if c1 := 710; c1 > a1 {

fmt.Println("恭喜你3")

}

C.

if else

if c2 := 650; c2 > a1 {

fmt.Println("恭喜你4")

} else { //没有条件

fmt.Println("需要再努力")

}

D.

if elseif else

if c2 := 650; c2 >= a1 {

fmt.Println("恭喜你一本")

} else if c2 >= a2 {

fmt.Println("恭喜你二本")

} else {

fmt.Println("需要再努力")

}

E.

if和if\_else区别

if 都会去执行

if\_else效率高,只要一个匹配到之后的不再执行

F.

读取文件并打印

const filename = "abc.txt"

if contents,err := ioutil.ReadFile(filename);err != nil{

fmt.Println(err)

}else {

fmt.Printf("%s\n",contents)

}

## 条件语句switch

### 支持多个条件的匹配

a.不同的case之间不需要使用break

b. fallthrough 强制往下执行一个case

score := 90

switch score {

case 90:

fmt.Println('C')

case 80:

fmt.Println("B")

case 10, 20, 30, 40:

fmt.Println("D")

default:

fmt.Println("A")

}

### 支持一个初始化语句

初始化语句和判断条件用 ； 分割

switch s1:=90;s1 { //初始化语句 ; 条件

case 90:

fmt.Println('C')

case 80:

fmt.Println("B")

case 10, 20, 30, 40:

fmt.Println("D")

default:

fmt.Println("A")

}

### 只有初始化语句

switch s3:=90; { //只有初始化没有条件

case s3>90:

fmt.Println('A')

}

### 没有条件

var s2 int = 90

switch { //没有条件

case s2>90://写判断语句

fmt.Println('A')

}

## 循环语句for

### 语法

1.for 初始化条件;判断条件 ;条件变化 {...}

2.迭代打印每个元素 默认返回两个值 第一个元素位置 一个元素本身

for i,data :=range str {

fmt.Println(i,data)

for i,\_ := range str{

fmt.Println(i)

}

for \_,data := range str{

fmt.Println(data)

}

for i := range str{

fmt.Println(i)

}

### 嵌套

count := 0

for i :=0;i<5 ;i++ {

for j :=0;j<5 ;j++ {

fmt.Println(i,j)

count +=1

}

}

fmt.Println(count)

### 无限循环

for{...}

## 跳出循环

break 跳出循环 如果多层for循环 跳出最近的内循环

continue 跳出本次循环 进入下一次循环

break 可以在for和switch使用，continue只能是在for

## 跳转

定义：

goto 跳转 goto关键字 End是标签 用户起的关键字

goto使用场景是测试的时候可以跳过一些不用注释代码

使用：

fmt.Println("aaa")

goto End //如果End在goto在goto之前了 就会无限制的循环

fmt.Println("bbb")

End:

fmt.Println("ccc")

## switch vs if

if适合区间判断，嵌套使用

switch执行效率高，将多个满足条件的值放一起，适合做固定值的判断

if相对switch执行效率低，分支少用if

switch不建议使用嵌套

## 案例

### 百鸡百钱算术

5cock 3hen 1/3chicken

//方法一：

count1 := 0

for cock:=0;cock<=20;cock++{

for hen:=0;hen<=33 ;hen++ {

for chicken:=0;chicken<=100 ;chicken+=3 {

count1+=1

if 5\*cock+3\*hen+chicken/3 == 100 &&cock+hen+chicken==100 {

fmt.Println(cock,hen,chicken)

}

}

}

}

//方法二：

count2 := 0

for cock:=0;cock<=20;cock++{

for hen:=0;hen<=33 ;hen++ {

count2 +=1

chicken := 100-cock-hen

if chicken % 3 ==0 && 5\*cock+3\*hen+chicken/3 == 100{

fmt.Println(cock,hen,chicken)

}

}

}

fmt.Println(count1,count2)

方法一 vs 方法二

空间复杂度少一个数量级，循环次数减少

不要过多的使用for嵌套循环

### 十进制转二进制

func convertToBin(n int) string {

result := ""

for ; n > 0; n /= 2 {

lsb := n % 2

result = strconv.Itoa(lsb) + result

}

return result

}

### 输出读入的数据循环打印

//输出读入的数据

func printFileContents(reader io.Reader) {

scanner := bufio.NewScanner(reader)

for scanner.Scan() {

fmt.Println(scanner.Text())

}

}

//把字符串转化为io.Reader类型

s := `abc"d"

kkkk

123

p`

printFileContents(strings.NewReader(s))

//把文件读取为io.Reader

printFile("basic/branch/abc.txt")

func printFile(filename string) {

file, err := os.Open(filename)

if err != nil {

panic(err)

}

printFileContents(file)

}

# 复合类型

## 数组

str6 := "hello go"

fmt.Println(str6[0], str6[1]) //输出的结果 104 101

fmt.Printf("%c,$c", str6[0], str6[1]) //输出h,e

### 定义 var 数组名 [长度] 类型

[10]int和[20]是不同类型

### 初始化，默认值是0

var e [5]int = [5]int{1,2,3,4,5}

### 给数组赋值 按照下标赋值a[0] = 1

### 自动推导

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

### 部分初始化 没有初始化的赋值为0

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

### 指定某个元素初始化

h:= [5]int{2:10,4:20}

### ...通过初始化确认数组长度

i:=[...]int{1,2,3}

### 如果两个数组元素个数相同 类型相同 是可以赋值的

j:=i

### 数组地址

数组名表示整个数组 数组名对应的地址就是数组第一个元素的地址

fmt.Printf("%p\n",&j)

fmt.Printf("%p\n",&j[0])

fmt.Printf("%p\n",&j[1])

fmt.Printf("%p\n",&j[2])

### 数组冒泡排序

数组冒泡排序 从小到大

var ars[10]int = [10]int{3,1,2,7,5,6,4,10,9,8}

{1,2,3,5,6,4,7，9，8，10} 没执行一趟确定一个数据的位置 执行次数9次 i=len(ars)-1次

10个元素走9趟就能确定了

i表示总共需要执行多少周 i= len(ars)-1

j 交换了多少次 第一趟9次 第二趟8次 。。。 第九趟1次

外层控制行

for i :=0 ;i<len(ars)-1;i++{//比较了多少次

内层控制列

for j:=0;j<len(ars)-i-1;j++{ //每趟对比的次数

满足条件进行交换 大于升序 小于降序

if ars[j]>ars[j+1]{

ars[j],ars[j+1] = ars[j+1],ars[j] //大的往后移

}

}

}

fmt.Println(ars)

### 二维数组

var grid [4][5]int

外层控制行 内层控制列

for i:=0;i<len(a2) ;i++ {

for j:=0;j<len(a2[0]) ;j++ {

fmt.Println(a2[i][j])

}

}

二维数组初始化

//全部初始化 部分初始化

a3 := [3][4]int{{1,2,3,4},{5,6,7,8},{9,10,11,12}}

fmt.Println(a3)

### 数组的拷贝

数组是值类型

调用func(arr [10]int)会拷贝数组

在go语言中一般不直接使用数组

## 切片

### 基于数组定义

slice本身没有数据，是对底层array的视图

arr:=[...]int{0,1,2,3,4,5,6,7}

s := arr[2:6]

s2 := arr[:]

#### 使用

s[0]对应的是arr[2]

修改s[0]会对原本arr[2]修改

#### slice 的实现

ptr指向slice开头的元素

len是只能取到lice的长度

cap是从ptr到array末尾的长度

只要不超过cap就可以向后扩展,不可以向前扩展

s[i]不可以超越len(s)

#### reslice slice的扩展

arr :=[...]int{0,1,2,3,4,5,6,7}

s:= arr[2:6] //2,3,4,5

s2:=s[3:5] //5,6 没有报错

6不在s中也能出来，但是s2[4] 会报错

#### slice添加元素

len(arr) = 8

s := arr[2:6] //

s2 := s[3:5]

s3 := append(s2,10)//把7编程10

s4:=append(s3,11)//s4不再是对arr的view

s5:=append(s4,12)//s5和s4不再时对arr的view,

A.

添加元素时如果超过cap,系统会重新分配更大的底层数组，把原来的元素拷过去，原来的数组如果没人用就会被垃圾回收掉

B.

由于是值传递的机制，必须接收append的返回，因为prt len cap都有可能会变掉

### 直接创建

#### var s []int

s类型是slice

s的值Zero value for slice is null

s1:=[]int{2,4,6}

s2:=make([]int,16) //指定长度

s3:=make([]int,16,32) //capacity是32 提前开空间,容量需要大于长度

#### slice长度的扩增

不断的往slice添加数字

长度为添加完数字的长度

cap 每次放不下了长度就会在原来的基础上扩大二倍

当容量增加到1024以后 扩容上次cap的1/4

### slice操作

#### 拷贝：

系统内建copy函数 copy(disc,src)

从src拷贝到disc

#### 删除元素reslice功能，没有内建函数

s4:=append(s3[:3],s3[4:...]) //s被跳掉

上面s3[4:]...代表的是4以后的所有元素

删一个元素len减少，cap不会变

删除头

s2:=s2[1:]

删除尾

s2:=s2[:len(s2)-1]

#### 排序

sort.Ints(slice）

### slice作为参数传递

slice存放的是数组的地址空间

所以形参的改变会影响实参的值

## 指针

指针不能运算

go语言只有值传递一种方式

（值传递和指针配合）

参数传递：拷贝值和拷贝地址传递

## map类型

### 定义

基本

map[k]v //map[string]string

复合map

map[k1]map[k2]v

m2:=make(map[string]int)//m2 == empty map

var m3 map[string]int //m3 == nil

nil是可以参与运算的

nil 和empty map可以混用

### 操作

#### 遍历

for k,v := range map3 {...}

乱序的

#### 获取指定key的value

v := m3["k"]

k不存在会打印空串 //zero value

判断key是否存在

v,ok := m3["k"]

if v,ok:=m3["k"];ok{...}

#### 删除元素

delete(m3,"k")

如果key不存在，不会报错

### map的key

#### map使用哈希表,必须可以比较相等

不需要用户指定hashmap还是treemap

#### 除了slice 、map 、function的内建类型都可以作为key

#### struct不包含上述类型 也可以作为key

编译的时候检查

### 案例寻找最长不含有重复字符串的子串

https://leetcode.com/problems/longsubstring-without-repeating-characters/description/

比如遍历到x

1.start 当前不含有重复子串的开始位置

2.查看x前面是否有x 和x的位置 lastOccured[x]

查询map会有两个三种情况：没出现 、在start前 、出现在中间或start位置，前两种不用管 第三种就需要把start的位置改为lastOccured[x]+1 开始找新的子串

## 通道（ chan）

## 结构体（ struct）

## 接口（ interface）

# 类型转换 (只有强制类型转换）

m1, m2, m3 := 54, 30, 20

sum := m1 + m2 + m3

fmt.Println(sum)

avg := (sum) / 3

fmt.Println(avg) //输出34 因为三个原始都是整型 推倒结果也是整型

1.数据类型（变量）int(a)

2.数据类型（表达式）

avg2 := (float64(sum)) / 3

fmt.Println(avg2)

3.布尔不能转其他 其他也不能转bool

类型转换是在不同的，但是相互兼容的方式之间转换

var d32 float32 = 3.1

var f64 float64 = 3.5

在类型转换时 建议从低精度的转成高精度，反则会损失精度 会数据溢出 符号可能会发生改变

int8-->int16-->int32-->int64

float32-->float64

int64 -- float64 建议整型转成浮点型

num := float64(d32) + f64

println(num)

var i64 int64 = 12345

i8 := int8(i64)

fmt.Println(i8) //输出57

## 类型别名

给int64起一个别名叫bigint

type bigint int64

var ab bigint //其实是var ab int64

fmt.Println(ab)

type (

long int64

char byte

)

var bl long = 11

var cc char = 'a'

fmt.Println(bl, cc)

## 字符串的拼接 +

# 随机数

创建三步骤：

1.导入头文件 math/rand

2.随机数种子

rand.Seed(1)

3.创建随机数

随机数是根据随机数种子计算出来的一个数（计算方法一定 随机数就不会变化 不管执行多少次值和顺序都不变）

fmt.Println(rand.Int()) //使用int生成较大的随机数，

fmt.Println(rand.Int())

4.利用事件生成随机数种子

rand.Seed(time.Now().UnixNano()) //随机数种子 纳秒

fmt.Println(rand.Int())//创建随机数

5.生成一百以内的数字

fmt.Println(rand.Intn(100)) //取模100 得到0~99

6.模拟双色球

模拟双色球

红球1-33 选6

蓝球1-16 选1

蓝球红球数字可重复

rand.Seed(time.Now().UnixNano())

var red [6] int

for i:=0;i<len(red) ;i++ {

v := rand.Intn(33)+1 //因为值是0-32 ,所以加1

for j:=0;j<i ;j++ { //把这个新得到的v和前面的数字比较，查看是否有重复值

if red[j] == v {

v =rand.Intn(33)+1

j -- //把j回退到当前位置 重新比较

}

}

red[i] = v

}

fmt.Println("红球",red,"蓝球",rand.Intn(16)+1)

# 内存

A.

内存 --- 堆栈

从主函数运行

加载函数信息和函数变量名到栈区内存

栈区内存的存储模型

内存相对一个可执行程序来说 可以分为：代码区 数据区 堆区 栈区

栈区：用来存储程序执行过程中函数内部定义的信息 和 局部变量

形式参数 告知调用函数时的格式

实参 在调用函数的时候将实参传递给形参

函数执行调用完毕就会从内存中释放 栈区存储

原理：先进后出，函数先销毁是后加载进来的

B.

内存 函数返回值

函数调用执行完后会把结果赋值到一个新的变量 ，函数自身销毁了

内存 函数的嵌套调用

C.

函数名表示的是一个地址 函数在代码区的地址 （代码区 数据区 堆区 栈区)

内存四区模型 （从低地址到高地址 系统内部不做考虑--》（代码区 数据区 堆区 栈区)==》。。。 ）

代码区： 计算机指令 ：只读

数据区： 又分为：初始数据区、未初始化数据区、常量区

堆区：一个很大的空间 在使用时开辟空间，使用结束时释放

栈区: 函数内部定义的变量

# 函数基本使用

## 定义

将代码重用的机制

func 函数名（参数列表） 返回值 {...}

函数不能重名

函数的使用 run()

## 函数

### 普通函数

func test(x string){...}

test("aa") //实参

### 不定长参数

test(t ...int)定义

test4(1,2,3,4)使用

test5(x1 string,x2 ...int)定义

test5("a",1,2,3,4)使用,不定参要放到最后，不能在其他定参前面

固定参数必须传值，不定参数根据需要决定是否传值

能将不定参的名称传给另一个不定参，如果要传值需要获取指定个数的数据

func func1(args ...int){

//func2(args) 错误写法

func2(args[0],args[1],args[2])

func2(args[0:len(args)]...)//[:3] [2:3]

func2(args[:]...)

func2(args...)

}

## 函数的嵌套使用：

函数的嵌套使用 在一个函数中调用另一个函数

## 函数嵌套的执行过程

## 函数返回值

返回多个值

接收返回值

匿名变量，丢弃接收的数据

func func3() int { //返回整型数据 通过return返回数据

var a int = 5

var b int =1

var sum = a+ b

return sum

}

func func4(a,b int) (sum int) { //返回整型数据 通过return返回数据

sum = a+ b

return //sum

}

func func5(a,b int) (a1,b1,sum int) { //返回多个值

sum = a+ b

a1=a

b1=b

return //sum

}

func func55(a,b int) (int, int) { //返回值的一种写法

sum2 := a+ b

return sum2,a

}

推荐：

func div(a, b int) (q, r int) {

return a / b, a % b

}

接收返回值：

函数返回多个值

a,b,c := func5(1,5)

fmt.Println(a,b,c)

匿名变量 丢弃接收的数据

\_,\_,sum := func5(1,5)

## 函数作为参数

### 定义

一个函数接收参数第一个是函数，后面的是基本数据类型

函数作为参数

func apply(op func(int, int) int, a, b int) int {

p := reflect.ValueOf(op).Pointer() //获得函数的指针

opName := runtime.FuncForPC(p).Name() //获得函数名字

fmt.Printf("Calling function %s with args "+

"(%d, %d)\n", opName, a, b)

return op(a, b)

}

### 调用函数

fmt.Println("pow(3, 4) is:", apply(

func(a int, b int) int {

return int(math.Pow(

float64(a), float64(b)))

}, 3, 4))//计算3的4次方

## 函数类型变量

### 用变量接收一个地址 通过f调用了函数

f:=demo1

f(1,2) //告诉我们这是代表一个函数

fmt.Println(f) //地址和demo1函数的内存地址一样的

### 声明一个函数类型的变量

var f1 func(int,int)

### 定义函数类型 给已存在的函数起别名 （之前学过类型别名）

## 定义函数类型 给已存在的函数起别名

type FUNCDEMO1 func(int, int)

var f2 FUNCDEMO1

要求demo1 的定义 和FUNCDEMO1要一致 包括：参数类型 个数 返回值

f2 = demo1

## 作用域

# 匿名函数&闭包&递归

## 匿名函数

### 定义没有名字的函数 在函数中定义一个函数

### 使用：

匿名函数需要一个变量去接收

在匿名函数里面是可以直接使用主函数中的局部变量 匿名函数对主函数的修改 影响到主函数的值

f := func() {

num ++

fmt.Println(num)

}

### 函数的调用

方法一：

f()

方法二：

fmt.Println(num)

func() {

num ++

fmt.Println(num)

}()

方法三：

自己定义一个函数类型 创建一个变量 调用函数

type FType func()

var func1 FType

func1 = f

func1()

4.有参数和返回值的匿名函数

func(a,b int) {

sum :=a+b

fmt.Println(sum)

}(3,6)

x,y,z := func(a,b int)(i,j,sum int){

i= a

j =b

sum = a+b

return

}(1,3)

## 闭包

匿名函数 实现了闭包 ： 在一个函数中有权访问另外一个函数作用域中变量的函数 匿名函数访问main中定义的num

匿名函数就是闭包

闭关不关心捕获的变量 或者常量是否超出了作用域 只要闭包还在使用它 这个变量就还存在 所以就不会重复的初始化x的操作

正常没使用闭包：

fmt.Println(test1())

fmt.Println(test1()) //返回值始终是1 因为上次调用完后就销毁了

-------------------------------------

定义一个函数 返回值是匿名函数：

func test2() func() int{ //返回匿名函数类型

var x int

return func() int{ //返回匿名函数

x++

return x

}

}

f11:= test2() //使用变量接收函数

fmt.Println(f11()) //f11是地址 加括号是函数的调用

fmt.Println(f11()) //输出的是 2， 数字变成累加

## 递归（回调）

递归函数 调用函数本身

1+2+。。。+100

func add100(num int) int {

if num == 1{

return 1

}

return num+add100(num - 1)

//100+（99+（98+。。。+（3+（2+（1

}

### 值接收者 vs 指针接收者

要改变内容必须使用指针接收者

结构过大也考虑使用指针接收者

一致性：如果有指针接收者，最好都是指针接收者

值接收者 是go语言特有的

指针接收者 java的this引用、 python的self 、c/++的this指针

值/指针接收者 均可接收 值/指针

# 面向对象

## 定义结构

定义 在函数外部

type 结构体名 struct{

结构体成员列表

成员名 数据类型

name string

age int

}

## go语言仅支持封装，不支持继承和多态

面向接口编程 实现继承和多态

## go没有class 只有struct

## 使用结构

创建结构体对象

结构体的工厂函数

### 创建treeNode

内容是value 和左右指针指向左右treeNode

不论地址还是结构体本身，一律使用“.”访问成员

type treeNode struct {

value int

left, right \*treeNode

}

var root treeNode

root = treeNode{value:3}

root.left=&treeNode{}

root.right=&treeNode{5,nil,nil}

root.right.left=new(treeNode)

root.right是指针，go语言里指针也可以"."

### 切片类型的struct

eg:

[]treeNode{

{value:3},

{}

{6,nil,&root}

}

### go没有提供构造函数

程序已经提供了构造方法，不需要构造函数

如果想自己控制构造，可以加工厂函数，工厂函数返回指针类型，内部是构造方法

func CreateNode(value int) \*Node {

return &Node{Value: value}

}

1.方法返回的是一个局部变量的地址给主方法

在c/c++中是会报错，但是go可以 局部变量返回给别人用

2.root.Left.Right = creatTreeNode(2)

3.go创建的局部变量是在堆上还是栈上？

不需要知道

编译器和运行环境给决定的：

如果局部变量没有取地址返回出去就在栈上分配；

如果取了一个地址有返回给别人用 就在堆上创建 参与垃圾回收 当外面不再使用了就由垃圾回收机制销毁掉

## 为结构定义方法遍历树

### 方法定义位置

不是struct里面，而是结构外面一个function

### 方法名前需要加一个接收者：

这种写法只是调法不一样，其实和普通函数 参数写在方法名（参数）没区别，只是调用的时候形式有区别

值接收者

func (node Node) Print() {

fmt.Print(node.Value, " ")

}

指针接收者

只有指针才可以改变结构内容

nil指针也可以调用方法（和其他语言null不能参与运算有很大的区别）

func (node \*Node) SetValue(value int) {

if node == nil {

fmt.Println("Setting Value to nil " +

"node. Ignored.")

return

}

node.Value = value

}

### 中序遍历（如果左节点有数据就会递归调用）

func (node \*treeNode) traverseFunc () {

if node == nil {

return

}

node.left.traverse ()

node.print()

node.right.traverse()

}

# 封装&包

## 封装

方法的可见不可见

1.名字使用CamelCase

2.首字母大写：public

3.首字母小写：private

4.以上规则不止是方法，常量 函数 结构体 都是

## 包

每个目录只能一个包

目录名和包名可以不同，但是要唯一

main包包含可执行入口

为结构定义的方法必须放在同一个包内

结构体和方法可以在一个包下的不同文件

命名技巧：不需要重复包名

idea shift+f6 对结构体重构命名

### main包间不能互相调用

main 包中的不同的文件的代码不能相互调用，其他包可以。所以其实Link.go没有被一起编译执行

## 扩展已有类型

如何扩充系统类型或者别人的类型

A.

方法一：定义别名

1.type Queue []int

2.定义方法

func (q \*Queue) Push(v int){

\*q = append(\*q,v)//不仅改了q指向内容 也改了q本身

}

B.

方法二：使用组合

1.定义一个自己的类型

元素类型为要扩展的类型指针

如果直接传数据需要把数据拷贝，所以直接传地址

2.定义自己的结构体方法

对象 != nil

对象.属性 != null

在使用自己扩展的结构体时需要先定义遍历

# 接口

go是面向接口的语言，只支持封装

完成其他语言的继承和多态

duck typing

描述事物的外部行为而非内部结构

go是编译时绑定，而非动态绑定

go属于结构化类型系统，类似duck typing

传统面向接口是实现者告诉所有人自己实现了哪些接口，别人可以使用

python 运行时判断接口中是否有方法

需要注释来说明接口

c++ 引入模板 编译时判断接口中是否有方法

需要注释来说明接口

java中是extend 接口类，强制要求参入的参数必须实现接口.缺点是实现多个接口实现不方便

go具有c++、python的灵活性，又具有java的类型检查

面向对象会告诉使用者 我实现了哪些接口，别人可以用我

go语言 接口由使用者定义？？？

接口的定义

type Retriver interface{

Get(source string) string

}

接口的使用

func download(retriver Retriver) string{

return retriver.Get("http://xxx")

}

接口方法的实现：

实现是隐式的

只要实现接口里的方法

实现的方法可以接收的值类型：指针、值

接口的调用：

调用download需要传递的是一个实现了Get方法的接口变量

接口变量

实现者的类型

实现者的值 或者 实现者的指针--》实现者

1.接口的实现中 接收者类型不要用接口变量的地址，因为接口本身就包含一个指针

2.接口变量有什么：

接口变量自带指针

接口变量同样采用值传递，几乎不需要使用接口的指针

指针接收者只能以指针方式使用；值接收者都可

查看接口变量：

type assertion

type switch

表示任何类型interface{}

改造原本queue支持的是int类型

改为支持任何类型

type Queue []int

返回值类型int

改为

type Queue []interface{}

返回值 interface{}

如果要限制输入和输出的是int

输入参数是int

输出返回值类型改为int，并且return v.(int)

这样把输出的接口对象肚子里的值强转为int

这样编译时不会报错 如果运行时转换失败会报运行时错误

组合接口

定义组合接口：（使用者）

接口一 接口二 组合

type RetriverPoster interface {

Retriver

Poster

}

一个结构体，实现两个接口方法get post

接口实现方法中 要修改结构体中变量值，需要接收指针变量

常用系统接口

Stringer

实现String()

Reader

接口方法Read(p []byte)

从文件读内容到byte

Fscanf

Writer

接口方法Write(p []byte)

把数据写道文件

Fprintf

File实现了Read Write接口方法

所以参数要用Reader Writer接口对象

文件、string、byte、slice、网络 都可以Read Write

Strings.NewReader()

bytes.NewReader()

# 函数式编程

函数式编程vs函数指针

函数是一等公民：参数、变量、返回值都可以是函数

高阶函数

函数的参数仍然是函数

函数->闭包

闭包： 函数体 局部变量 自由变量

自由体变量是返回函数方法以外的变量，变量可以是多层结构嵌套（树结构）

匿名函数返回的不止是return 后的代码 是一个闭包

正统 函数式编程

不可变性：不能有状态、只有常量和函数

函数只能有一个参数

其他语言

python 原生支持

对自由变量的引用要加nonlocal

使用\_\_closure\_\_查看闭包的内容

c++

过去stl或者boost带有类似库

c++11及以后：支持闭包

[=] 代表传值传进来的

mutable 代表可以改变值

java

1.8以后：使用Function接口和lamdba表达式来创建对象

匿名类或lamdba表达式均支持闭包

案例：

斐波那契数列

斐波那契数列

把斐波那契数当作文件Reader

# 资源管理与出错处理

程序执行时因为异常导致资源无法正常关闭

2.defer延迟调用

确保调用在函数结束时发生

参数在defer语句时计算

defer列表为先进后出

何时使用defer调用

Open/Close

Lock/Unlock

PrintHeader/PrintFooter

3.err的捕获与出错处理

自定义error类型

自定义error实现Error interface

error异常捕获 把已知类型的错误处理掉

4.错误处理规范

统一错误处理逻辑

# 测试与性能调优

子主题 6

1.传统测试 vs 表格驱动测试

go语言使用表格驱动测试

debug 断点调试

传统测试：

1.使用方式：

@Test

传统 assertEquals(其他值，add(1,2))

2.优缺点：

缺点：

测试数据和测试逻辑混在一起

出错信息不明确（希望的是3 得到的是4 但是不会说哪里出来的是4）

一旦一个数据出错测试全部结束

表格驱动测试

1.明确的测试数据和测试逻辑

2.明确的出错信息

3.可以部分失败

4.go语言的语法使得我们更易实践表格驱动测试

程序方法名开头大写

可以在命令行 当前文件目录下执行

go test .

func TestTriangle(t \*testing.T) {

tests := []struct{ a, b, c int }{

{3, 4, 5},

{5, 12, 13},

{8, 15, 17},

{12, 35, 37},

{30000, 40000, 50000},

{math.MaxInt32,1,math.MinInt32},

}

for \_, tt := range tests {

if actual := calcTriangle(tt.a, tt.b); actual != tt.c {

t.Errorf("calcTriangle(%d, %d); "+

"got %d; expected %d",

tt.a, tt.b, actual, tt.c)

}

}

}

案例 最长子串测试

2.代码覆盖率和性能测试

converage

benchmark

通过IDEA查看代码覆盖

通过命令的方式查看：

go test -converprofile=c.out

go tool conver -html=c.out

性能计算

Benchmark

执行多少遍b.N决定

go test -bench .

3.使用pprof进行性能调优

命令行执行：

go test -bench . cpuprofile cpu.out

以上命令输出二进制文件,用以下命令去查看

go tool pprof cpu.out

输入web 可以有svg图查看占比（需要提前安装graphviz）

quiet(退出)

4.测试http服务器

5.生成文档和示例代码

# goroutine

并发编程

## 29.1.原生支持

关键字go 加了go就会并发的执行

10个并发协程（Coroutine） 不断的打印

func main() {

for i := 0; i < 10; i++ {

go func(i int) {

for {

fmt.Printf("Hello from "+

"goroutine %d\n", i)

}

}(i)

}

//time.Millisecond一毫秒

//time.Minute

time.Sleep(time.Millisecond)

}

1.go func中的i直接使用for中的i不安全，所以需要传进去

2.main和go func是并发执行的，当main执行完后会把其他的协程杀掉了

3.使用sleep让main等待再退出

4.一般操作系统开一百个线程没问题，但是上千个线程并发执行 就需要异步io

go 语言中就只需要加个go就可以并发执行

## 29.2. go的调度

特性coroutine是轻量级“线程”

• 线程在任何时候都有可能被系统切换 是抢占式任务处理，系统处理完其他的还会回来继续执行完没完成的线程 需要存更多的上下文关系

非抢占式多任务处理

• 由协程主动交出控制权 不同于抢占式，要保存各线程的状态 只用保存资源切换的几个点

编译器/解释器/虚拟机层面的多任务

• 编译器级别的多任务 go有调度器调度协程 操作系统也有调度器调度协程

多个协程可能在一个或多个线程上运行

• 由调度器决定的

抢占式 vs 非抢占式

func main2() {

var a[10] int

for i := 0; i < 10; i++ {

go func(ii int) {

for {

a[i]++

runtime.Gosched()

}

}(i)

}

time.Sleep(time.Millisecond)

fmt.Println(a)

}

io操作有等待的过程，会切换 交出控制权

创建了一个10长度的数组，每个数组创建一个协程 分别对数据增加

//time.Millisecond一毫秒

//time.Minute

time.Sleep(time.Millisecond)

a[i]++ 是不会主动交出控制权，所以一直在for循环 main函数也无法获取控制权

也可以手动交出runtime.GoSched()

让别人有机会执行 每个人获得的机会是不一样的 一般不这么写

3.检测数据访问冲突

race condition

以上直接引用for中的i不安全 因为外层循环会改变i的值，和里面for循环读取冲突，所以需要把i传进去

go run 1.goroutine.go

go run -race goruntine.go 检测数据访问冲突

如果匿名函数使用自由变量i,go func读了i 但是main重写了i

fmt.println(a)读的时候 协程正在写

4.子程序是协程的一个特例

协程和普通函数区别

普通函数

• 在一个线程内 一个函数执行完把再把控制权交还main函数，再去调用下一个语句

协程

• 在一个或多个线程内 main和dowork之间数据和控制权可以互相通信，交换控制权

协程和协程之间双向通道

其他语言对协程的支持

c++ boost.coroutine

python 3.5 asnyc def对协程原生支持，之前是使用yield关键字实现协程

go语言的协程

一个程序开启后会有一个调度器，负责调度协程，有些协程放到一个线程，有些可能两个或多个放到一个线程里面

• 调度器控制是否放一个线程里

任何函数只需要加上go就能送给调度器运行

不需要在定义时区分是否时异步函数（相当于python asnyc def 来讲的）

调度器在合适的点进行切换【切换的点并不能完全的控制】

5.goroutine可能的切换点

[只是参考 不能保证切换 不能保证在其他地方不切换]

i/o ,select channel进行树的遍历

channel

等待锁

函数调用（调度器决定是否切换）

runtime.Gosched()

6.查看开启一千个协程 启动了多少线程

查看实际系统上只是最多4个线程 因为4个核

# channel

协程和协程之间双向的通道

## 作用

goroutine和gouroutine之间的双向通道

用作两个routine之间数据的交换

## 使用channel

### 定义channel

var c chan int

定义一个channel 传入的数据是int，但是这个channel没给做出来是c == nil

c:=make(chan int)

创建一个channel

### 接收者

定义一个goroutine接收数据

for{

n := <-c 从channel取出数据

}

### 数据发送

c<- 发数据到channel

不接收会datalock异常

chan指定方向

chan<-

<-chan

### 案例

channel作为参数

channel也是一等公民

可以作为函数 可以作为参数 可以作为返回

1.定义10个channel

2.定义对应到10个接收者的goroutine

3.发送数据到10个channel

接收的时候是按顺序接收的 但是print io调度是无序的

func chanDemo(){

var channels [10]chan int

for i:=0;i<10;i++{

channels[i]=make(chan int)

go worker(i,channels[i])

}

for i:=0;i<10;i++{

channels[i]<-'a'+i

}

}

func worker(id int) chan int{

for{

fmt.Printf("%d,%c\n",id,<-c)

}

}

将创建channel和数据接收goroutine抽取成函数

func createWork(id int) chan int{

c:=make(chan int)

go func(){

for{

fmt.Printf("%d,%c\n",id,<-c)

}

}()

return c

}

func chanDemo(){

var channels [10]chan int

for i:=0;i，10；i++{

channels[i]=createWorker(i)

}

for i:=0;i<10;i++{

channels[i]<-'a'+i

}

## bufferChannel 添加缓冲区

c:=make(chan int,3) //缓冲区是3

## 通知接收方数据发送结束

如果发的人不通知，直到发的人程序退掉 接收也就退掉了

close(c) //关闭chan 告诉接收方数据已经发完了

接收方还会收到chan的零值

判断法一：

n,ok := <-c

如果有值ok就是true

判断法二：

直接for ... range channel

for{

n,ok ；= <-c

if !ok{

break

}

fmt.p(<-c)

}

for n:=range c{

fmt.p(n)

}

## channel的理论基础CSP

communication sequential process

CSP：

不要通过共享内存来通信；通过通信共享内存

chanDemo发完数据后work通知打印完毕了

上面案例中的打印 是设置了等待时间才让goroutine有时间执行完，现需要每个n打印完了通知外面

1.定义结构体包含两个channel ：

一个用来接收数据

一个用通知

2.创建channel和goroutine

3.调用

以上代码打印出来的是顺序执行了，发一个 等他结束再下一个

1.

type worker struct {

in chan int

done func()

}

2.

func createWorker(id int) worker{

w := worker{

in:make(chan int),

done:make(chan bool),

}

go doWork(id,w.in,w.done)

return w

}

func doWork(id int,c chan int,done chan bool){

for n:=range c{

fmt.p(n)

done<-true

}

}

3.func chanDemo(){

var workers [10]worker

for i:=0;i<10;i++{

workers[i]=createWorker(i)

}

for i:=0;i<10;i++{

workers[i].in<-'a'+i

<-workers[i].done

}

for i:=0;i<10;i++{

workers[i].in<-'A'+i

<-workers[i].done

}

}

希望数据直接发出去，等结束以后再收二十个done

以上思路错误 第一个循环发完，返回来的done没人收，就会出现循环等待deadlock,第二个for发不出去(所有的channel发送都是blocked)

解决思路:

法一：

给goroutine里面再开一个goroutine 去返回done

法二：

两个for分开接收done

func doWork(id int,w worker) {

for n := range w.in {

fmt.Printf("Worker %d received %c\n",

id, n)

go func(){w.done <- true}()// 再开一个goroutine 并发发送done 就不会卡住

}

}

## 库函数 sync.WaitGroup 等待多人的事情

Add 是添加任务

Done 每个任务做完有个done

Wait 等待任务做完

共用一个waitGroup

wg.Add(1) 每个for添加一个任务

也可以一下写wg.Add(20)

1.定义结构体

type worker struct {

in chan int

done func()

}

2.创建channel+goroutine

接收者goroutine使用waitgroup返回done

func createWorker(

id int, wg \*sync.WaitGroup) worker {

w := worker{

in: make(chan int),

done: func() {

wg.Done()

},

}

go doWork(id, w)

return w

}

3.方法调用者创建等待组，传到接收者，wait方法接收waitgroup done返回的数据

func chanDemo() {

var wg sync.WaitGroup

var workers [10]worker

for i := 0; i < 10; i++ {

workers[i] = createWorker(i, &wg)

}

wg.Add(20)

for i, worker := range workers {

worker.in <- 'a' + i

}

for i, worker := range workers {

worker.in <- 'A' + i

}

wg.Wait()

}

## 案例

使用channel实现树的遍历

func (node \*Node) TraverseWithChannel() chan \*Node {

out := make(chan \*Node)

go func() {

node.TraverseFunc(func(node \*Node) {

out <- node

})

close(out)

}()

return out

}

main(){

c := root.TraverseWithChannel()

maxNodeValue := 0

for node := range c {

if node.Value > maxNodeValue {

maxNodeValue = node.Value

}

}

fmt.Println("Max node value:", maxNodeValue)

}

## select进行调度（select ...case ...default）

定义了多个channel，哪个数据来的快就接收哪个

# 标准库

## http库

### 服务端起服务

http.HandleFunc(“/hello”,HelloServer)

log.Fatal(http.ListenAndServe(“:12345”,nil”))

### 起客户端

开发步骤：

#### 发请求

http.Get(“url”)

处理错误

close

#### 解析返回数据

httputl.DumpResponse(resp,true) 其中true代表是否dump这个body

处理错误

使用这个返回数据

#### 丰富访问头数据

request,err := http.NewRequest(

http.MethodGet 请求的方法

“url” 请求的地址

nil 如果是post会有form表单

)

request.Header.Add(“key”,“value”)

eg : User-Agent 指定设备

#### 使用默认Client请求不能再使用A中的Get，而是

http.DefaultClient.Do(request)

#### 使用自定义Client，自己生成一个Client

http.Client{

Transport 设置一些带来服务器

CheckRedirect 重定向从这个函数过

Jar CookieJar 爬虫的时候模拟登陆可能会需要用

TimeOut

}

client := http.Client{

CheckRedirect:func(

req \*http.Request,

via []\*http.Request) error{

redirect可以有很多次，所有重定向路径都放via，每一次请求路径放req里面，error如果是nil 就是运行重定向，如果错误就会终止重定向

fmt.Println(“redirect”,req)

return nil

}

}

resp,err := client.Do(request)

### http服务的性能分析

import \_“net/http/pprof” 下划线表示：虽然这个没有用到，但是需要load其中的一些帮助程序进来

#### 引入了pprof 访问路径会默认有个ip:port/debug/pprof/

Stack trace:堆栈轨迹:如果你需要打印出某个时间的调用堆栈状态，你将产生一个堆栈轨迹

#### go tool pprof命令分析性能

源文件在go/net/http/pprof/pprof.go

eg :查看30sCPU的使用率

go tool pprof ip:port/debug/pprof/profile

查看cpu的使用，判断是否需要优化

eg:查看heap使用情况

go tool pprof ip:port/debug/pprof/heap

## 其他标准库

bufio 例如读取文件加缓存会快，一次读取很多在flush到磁盘

log

encoding/json 结构体按照json进行encode 或者decode

regexp正则表达

time 用法是100\*second 或 duration

strings/math/rand

### 查看标准库文档

#### 自己启动

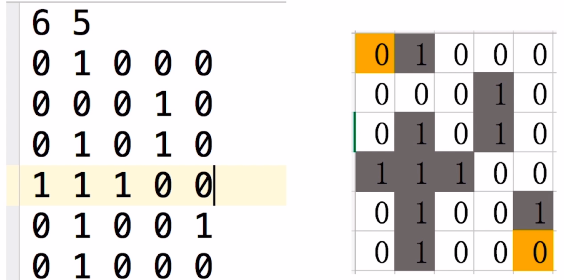
网上拉下来的和自己写的也会出现在库列表里

godoc -http :8888

#### 网上文档

<https://studygolang.com>

# 广度优先搜索



1代表墙

只能上下走不能左右左

左上角进，右下角出

求出最短的路

从里往外，按照一个上左下右的顺序 一层一层的探索递进 就能找到最短路径；

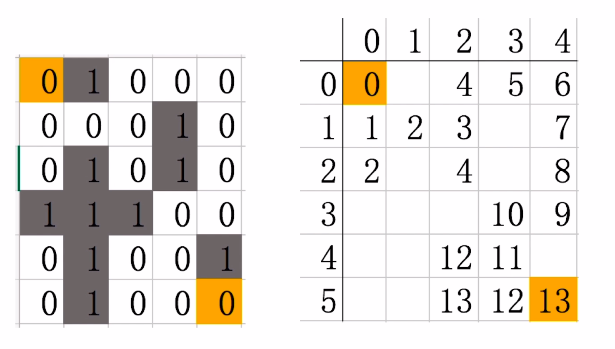
如果顺着一个方向探索就是深度优先，是找不到最短路径的，因为它有可能会绕路；

## 案例分析

案例中节点分为三种状态：

发现未探索、探索、未发现

发现未探索的放入队列



倒过来走，最终就能走到起点，因为13附件就一个12，12一个11，...

结束情况有两种：

1. 探索队列为空
2. 走到终点