



# Go语言编程

Andy Li



- Robert Griesemer, Rob Pike 和 Ken Thompson。Robert在开发Go之前是Google V8、Chubby和HotSpot JVM的主要贡献者；Rob主要是Unix、UTF-8、plan9的作者；Ken主要是B语言、C语言的作者、Unix作者

- 概括：开源、简单、高效
- 优点：强类型、自带gc、弱面向对象、自带并发、编译速度快、开发效率高、维护成本低、统一的编码风格、丰富的标准库
- 缺点：web开发生态不完善、error、泛型、零值
- Trends:[Google Trends](#) , [Github Trending](#)
- [知识图谱](#)

- Go程序是通过package来组织的
- package main 编译后生成可执行文件，其他package生成依赖包
- fmt是go自带的I/O包，类似C的printf和scanf
- import关键字用于引入包
- func关键字用于定义函数
- main()是程序入口
- Println是包fmt中的一个函数，输出字符串到控制台

```
package main  
  
import "fmt"  
  
func main() {  
    fmt.Println("Hello, 世界")  
}
```

Hello, 世界

Program exited.

# 基础语法

- 行分隔符

```
package main

import "fmt"

func main() {
    fmt.Println(a...: "Hello, 世界");
}
```

分号可有可无，编译器判断结束

```
package main

import "fmt"

func main() {
    fmt.Println(a...: "Hello, 世界");fmt.Println(a...: "Hello, 世界")
}
```

多个语句需要用分号隔开，一般不采用这种方式

- 25个关键字

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

- 36个预定义标识符

append	bool	byte	cap	close	complex
complex64	complex128	uint16	copy	false	float32
float64	imag	int	int8	int16	uint32
int32	int64	iota	len	make	new
nil	panic	uint64	print	println	real
recover	string	true	uint	uint8	uintptr



- 常用的数据类型

类型	描述
string	UTF-8-encoded text
bool	true false
int32	$-2^{31}$ - $2^{31}$ 21亿
int	Int32
uint32	$0$ - $2^{32}$ 42亿
int64	$-2^{63}$ - $2^{63}$
float64	<u>IEEE_754</u>
byte	字节 uint8

\*float64要格外注意Inf和NaN

- 变量定义关键字 **var**
- 变量定义的几种方式

```
//先定义再复制
var name string
name = "andy"
fmt.Println(a...: "name:", name)

//定义时直接初始化
var age uint8 = 29
fmt.Println(a...: "age:", age)

//定义时直接初始化, 省略类型, 由编译器判断
var married = true
fmt.Println(a...: "married:", married)

//省略var
address := "上海市"
fmt.Println(a...: "address:", address)

//同时定义多个变量
var x, y int = 1, 2
//var x, y = 1, 2
//x, y := 1, 2
fmt.Println(a...: "x+y=", x+y)
```

- 常量关键字 **const**
- 定义常量的几种方式

```
//先定义再复制,不能用const  
var name string  
name = "andy"  
fmt.Println(a...: "name:", name)
```

```
//定义时直接初始化  
const age uint8 = 29  
fmt.Println(a...: "age:", age)
```

```
//定义时直接初始化,省略类型,由编译器判断  
const married = true  
fmt.Println(a...: "married", married)
```

```
//省略var  
address := "上海市"  
fmt.Println(a...: "address:", address)
```

```
//同时定义多个变量  
const x, y int = 1, 2  
//var x, y = 1, 2  
//x, y := 1, 2  
fmt.Println(a...: "x+y=", x+y)
```

- 一个很好的例子

```
const (
    _      = iota           // ignore first value by assigning to blank identifier
    KB ByteSize = 1 << (10 * iota) // 1 << (10*1)
    MB           // 1 << (10*2)
    GB           // 1 << (10*3)
    TB           // 1 << (10*4)
    PB           // 1 << (10*5)
    EB           // 1 << (10*6)
    ZB           // 1 << (10*7)
    YB           // 1 << (10*8)
)
```

运算符	描述	实例
+	相加	A + B 输出结果 30
-	相减	A - B 输出结果 -10
*	相乘	A * B 输出结果 200
/	相除	B / A 输出结果 2
%	求余	B % A 输出结果 0
++	自增	A++ 输出结果 11
--	自减	A-- 输出结果 9

运算符	描述	实例
==	检查两个值是否相等	(A == B) 为 False
!=	检查两个值是否不相等	(A != B) 为 True
>	检查左边值是否大于右边值	(A > B) 为 False
<	检查左边值是否小于右边值	(A < B) 为 True
>=	检查左边值是否大于等于右边值	(A >= B) 为 False
<=	检查左边值是否小于等于右边值	(A <= B) 为 True

运算符	描述	实例
&&	逻辑 AND	(A && B) 为 False
	逻辑 OR	(A    B) 为 True
!	逻辑 NOT	!(A && B) 为 True

运算符	描述	实例
&	按位与	(A & B) 结果为 12, 二进制为 0000 1100
	按位或	(A   B) 结果为 61, 二进制为 0011 1101
^	按位异或	(A ^ B) 结果为 49, 二进制为 0011 0001
<<	左移运算符	A << 2 结果为 240 , 二进制为 1111 0000
>>	右移运算符	A >> 2 结果为 15 , 二进制为 0000 1111



运算	描述	实例
=	简单的赋值运算符	$C = A + B$ 将 $A + B$ 表达式结果赋值给 $C$
+=	相加后再赋值	$C += A$ 等于 $C = C + A$
-=	相减后再赋值	$C -= A$ 等于 $C = C - A$
*=	相乘后再赋值	$C *= A$ 等于 $C = C * A$
/=	相除后再赋值	$C /= A$ 等于 $C = C / A$
%=	求余后再赋值	$C \% = A$ 等于 $C = C \% A$
<<=	左移后赋值	$C << = 2$ 等于 $C = C << 2$
>>=	右移后赋值	$C >> = 2$ 等于 $C = C >> 2$
&=	按位与后赋值	$C \& = 2$ 等于 $C = C \& 2$
^=	按位异或后赋值	$C \wedge = 2$ 等于 $C = C \wedge 2$
=	按位或后赋值	$C   = 2$ 等于 $C = C   2$

## if else

```
var x, y = 1,2
//if
if x < y {
    fmt.Println(a...: "x < y")
}
//if else
if x > y {
    fmt.Println(a...: "x < y")
} else {
    fmt.Println(a...: "x >= y")
}
//if elseif
if x > y {
    fmt.Println(a...: "x < y")
} else if x == y {
    fmt.Println(a...: "x == y")
} else {
    fmt.Println(a...: "x > y")
}
```

## 包含变量初始化的if

```
//包含初始化语句的if
if x,y:= 1,2; x<y {
    fmt.Println(a...: "x < y")
}
```

## 基础用法

```
switch {  
case grade == "A" :  
    fmt.Printf( format: "优秀!\n" )  
case grade == "B", grade == "C" :  
    fmt.Printf( format: "良好\n" )  
case grade == "D" :  
    fmt.Printf( format: "及格\n" )  
case grade == "F":  
    fmt.Printf( format: "不及格\n" )  
default:  
    fmt.Printf( format: "差\n" );  
}  
fmt.Printf( format: "你的等级是 %s\n", grade );
```

## Type Switch

```
var obj interface{}  
  
switch i := obj.(type) {  
case nil:  
    fmt.Printf( format: " x 的类型 :%T", i)  
case int:  
    fmt.Printf( format: "x 是 int 型")  
case float64:  
    fmt.Printf( format: "x 是 float64 型")  
case func(int) float64:  
    fmt.Printf( format: "x 是 func(int) 型")  
case bool, string:  
    fmt.Printf( format: "x 是 bool 或 string 型" )  
default:  
    fmt.Printf( format: "未知型")  
}
```

判断 interface 变量中实际存储的  
变量类型

fallthrough在switch中使用，在case的最尾端，用于告诉程序继续执行下一个case并忽略条件判断。

```
v := 42
switch v {
case 100:
    fmt.Println(a...: 100)
    fallthrough
case 42:
    fmt.Println(a...: 42)
    fallthrough
case 1:
    fmt.Println(a...: 1)
    fallthrough
default:
    fmt.Println(a...: "default")
}
// Output:
// 42
// 1
// default
```

## 基本用法

```
//类似C的for
sum := 0
for i := 0; i < 10; i++ {
    sum += i
}
```

```
//类似C的While
cnt := 0
for cnt < 10 {
    cnt++
}
```

```
//类似C的for(;;)
for {
    time.Sleep(time.Hour)
}
```

## 遍历数组

```
//index是数组下表
for index, value := range courseList {
    log.Print(index, value.ID)
}
```

## 遍历map

```
//遍历map
for key, value := range dataMap {
    fmt.Println(key, value)
}
```

控制条件： break, continue, goto

## 多返回值

```
func swap(x, y string) (string, string) {  
    return y, x  
}  
  
func main() {  
    a, b := swap(x: "Mahesh", y: "Kumar")  
    fmt.Println(a, b)  
}
```

## 匿名函数

```
type OutFunc func(output string)  
  
func Output(output string, f OutFunc) {  
    f(output)  
}  
  
func main() {  
    func() {  
        Output(output: "hello world", func(output string) {  
            fmt.Println(output)  
        })  
    }()  
}
```

## 一等公民

```
func main() {  
    f := func() {  
        fmt.Println(a...: "hello world")  
    }  
    f()  
    fmt.Printf(format: "%T", f) //func()  
}
```

//定义一个整型的数组, 可容纳10个元素

```
var list [10]int
```

```
fmt.Println(list) //[0 0 0 0 0 0 0 0 0 0]
```

//定义数组并初始化

```
var list2 = [10]int {1,2,3,4,5}
```

```
fmt.Println(list2) //[1 2 3 4 5 0 0 0 0 0]
```

//访问数组

```
fmt.Println(list2[1]) //2
```

//遍历数组

```
for index, value := range list2 {
```

```
    fmt.Println(index, value)
```

```
    //0 1
```

```
    //1 2
```

```
    //2 3
```

```
    //...
```

```
}
```

# 基础语法-切片slices



//定义切片

```
var list []int
fmt.Println(list == nil)//true
fmt.Println(list)//[]
```

//定义切片并初始化

```
var list2 = []int {1,2,3,4,5}
fmt.Println(list2 == nil)//false
fmt.Println(list2)//[1 2 3 4 5]
```

//使用make定义切片,定义一个长度是5, 容量是10的切片

```
var list3 = make([]int,5,10)
fmt.Println(list3 == nil)//false
fmt.Println(len(list3))//5
fmt.Println(cap(list3))//10
```

//根据数组定义切片

```
var arr = [5]int{1,2,3,4,5}
var slice = arr[:]
fmt.Println(slice) //[1 2 3 4 5]
fmt.Printf(format: "%T, %T\n", arr, slice)//[5]int, []int
```

//引用

```
slice[0] = 0
fmt.Println(arr)//[0 2 3 4 5]
```

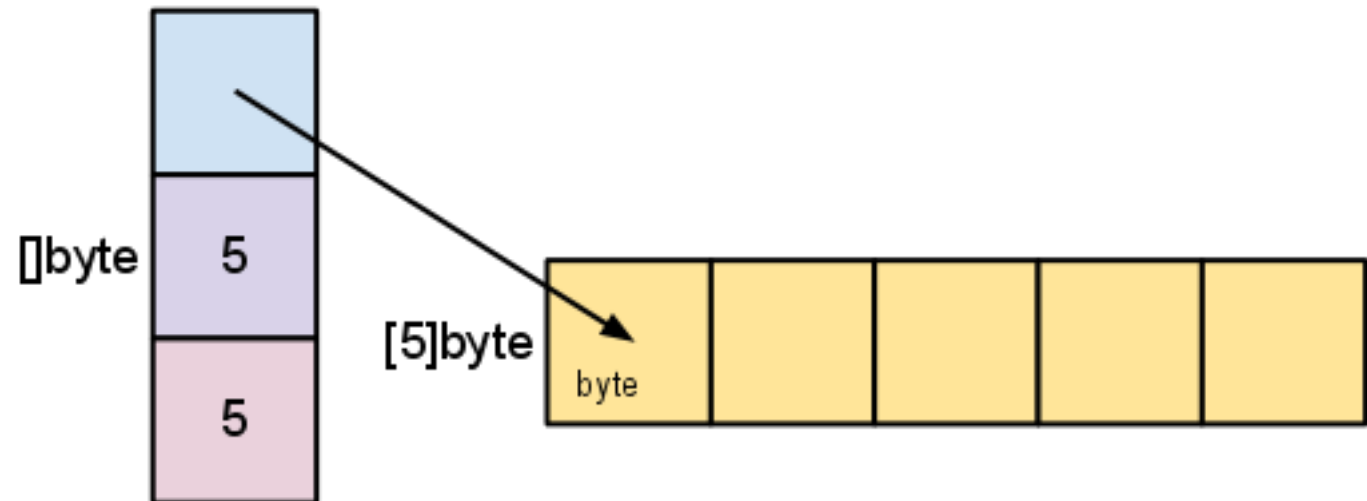
---



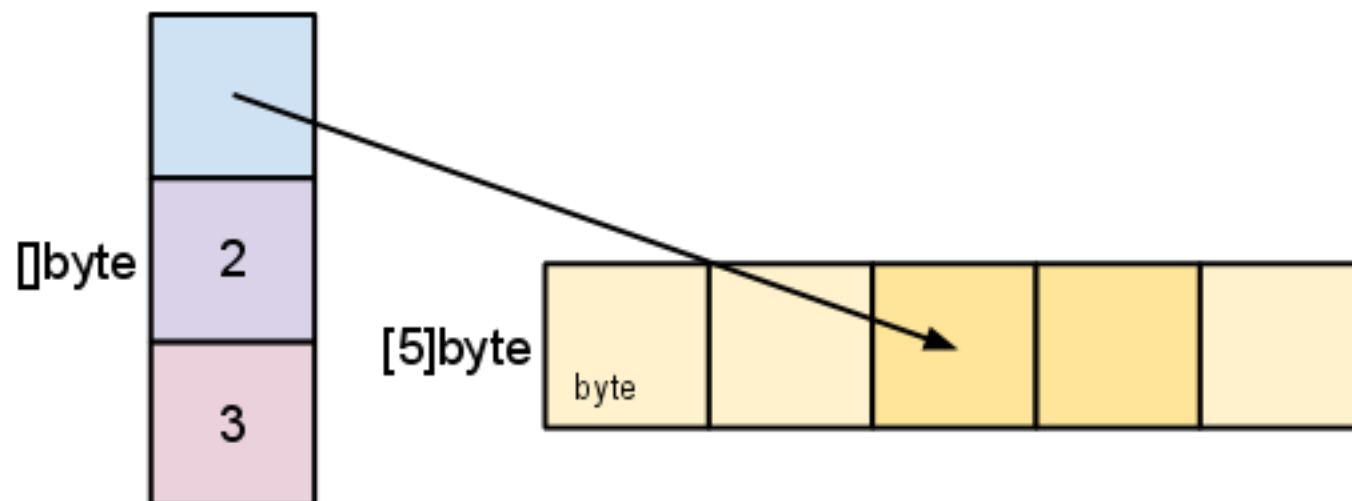
## 1.切片结构



## 2. `s:=make([]byte, 5)`



## 3. `s=s[2:4]`



- 关键字 **new**，用于各种类型的内存分配，返回一个指针

```
p := new(int)
fmt.Println(p)//0xc000078008
```

- 引用传递

```
func swap(x *int, y *int) {
    var temp int
    temp = *x
    *x = *y
    *y = temp
}
```

```
func main() {
    var a int = 100
    var b int= 200

    fmt.Println(&a,&b,a,b)//0xc0000160b0 0xc0000160b8 100 200

    swap(&a, &b);

    fmt.Println(&a,&b,a,b)//0xc0000160b0 0xc0000160b8 200 100
}
```

# 基础语法-Struct



```
// 声明一个新的类型
type person struct {
    name string
    age int
}

// 比较两个人的年龄，返回年龄大的那个人，并且返回年龄差
// struct也是传值的
Older := func(p1, p2 person) (person, int) {
    if p1.age > p2.age { // 比较p1和p2这两个人的年龄
        return p1, p1.age - p2.age
    }
    return p2, p2.age - p1.age
}

var tom person

// 赋值初始化
tom.name, tom.age = "Tom", 18

// 两个字段都写清楚的初始化
bob := person{age: 25, name: "Bob"}

// 按照struct定义顺序初始化值
paul := person{ name: "Paul", age: 43}

tb_Older, tb_diff := Older(tom, bob)

fmt.Printf("format: %s and %s, %s is older by %d years\n",
    tom.name, bob.name, tb_Older.name, tb_diff)
//output: Tom and Bob, Bob is older by 7 years
```

- 方法，语法糖

```
type person struct {  
    name string  
    age int  
}  
  
func (p *person) GetAge() int {  
    return p.age  
}  
  
func GetAge(p *person) int {  
    return p.age  
}  
  
func main() {  
  
    var tom person  
    tom.name, tom.age = "Tom", 18  
    fmt.Println(tom.GetAge()) //18  
    fmt.Println(GetAge(&tom)) //18  
}
```

```
type person struct {  
    name string  
    age int  
}  
  
func (p *person) SetAge(age int) {  
    p.age = age  
}  
  
func (p *person) GetAge() int {  
    return p.age  
}  
  
type teacher struct {  
    person  
    school string  
}  
  
func main() {  
    var tom teacher  
    tom.name = "tom"  
    tom.SetAge( age: 22)  
    fmt.Println(tom.GetAge())//22  
}
```

```
var countryCapitalMap map[string]string
fmt.Println(countryCapitalMap == nil)//true
countryCapitalMap = make(map[string]string)
```

```
countryCapitalMap [ "中国" ] = "北京"
countryCapitalMap [ "意大利" ] = "罗马"
```

//遍历

```
for country, capital := range countryCapitalMap {
    fmt.Println(country, "首都是", capital)
    //中国 首都是 北京
    //意大利 首都是 罗马
}
```

//key是否存在

```
capital, ok := countryCapitalMap [ "美国" ]
if ok {
    fmt.Println( a...: "美国的首都是", capital)
} else {
    fmt.Println( a...: "美国的首都不存在")
}
```

//删除元素

```
delete(countryCapitalMap, key: "意大利")
```

//遍历

```
for country, capital := range countryCapitalMap {
    fmt.Println(country, "首都是", capital)
    //中国 首都是 北京
}
```

\*非并发安全

- interface 是一种具有一组方法的类型，这些方法定义了 interface 的行为
- 如果一个类型实现了一个 interface 中所有方法，我们说类型实现了该 interface

```
type I interface {  
    Get() int  
    Set(int)  
}  
  
type R struct{ i int }  
func (p *R) Get() int { return p.i }  
func (p *R) Set(v int) { p.i = v }  
  
func f(i I) {  
    i.Set(10)  
}  
  
func main() {  
    var r R  
    f(&r)  
    fmt.Println(r.Get())//10
```

- interface{} 是一个空的 interface 类型，类似C中的void \*

```
// 定义a为空接口
var a interface{}
var i int = 5
s := "Hello world"
// a可以存储任意类型的数值
a = i
fmt.Println(a)//5
a = s
fmt.Println(a)//Hello world
```

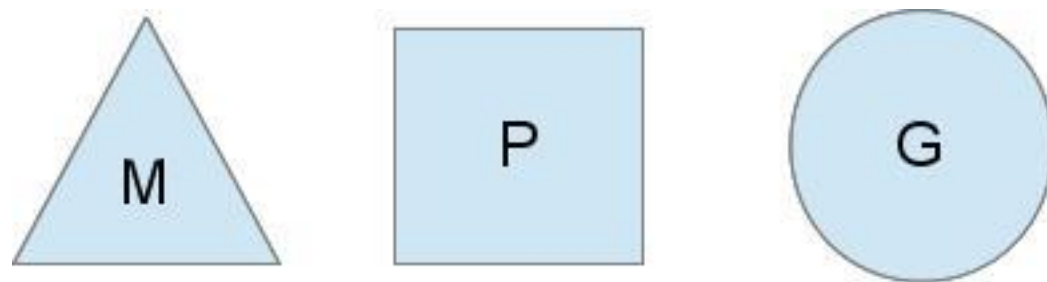


- 协程，使用关键字go

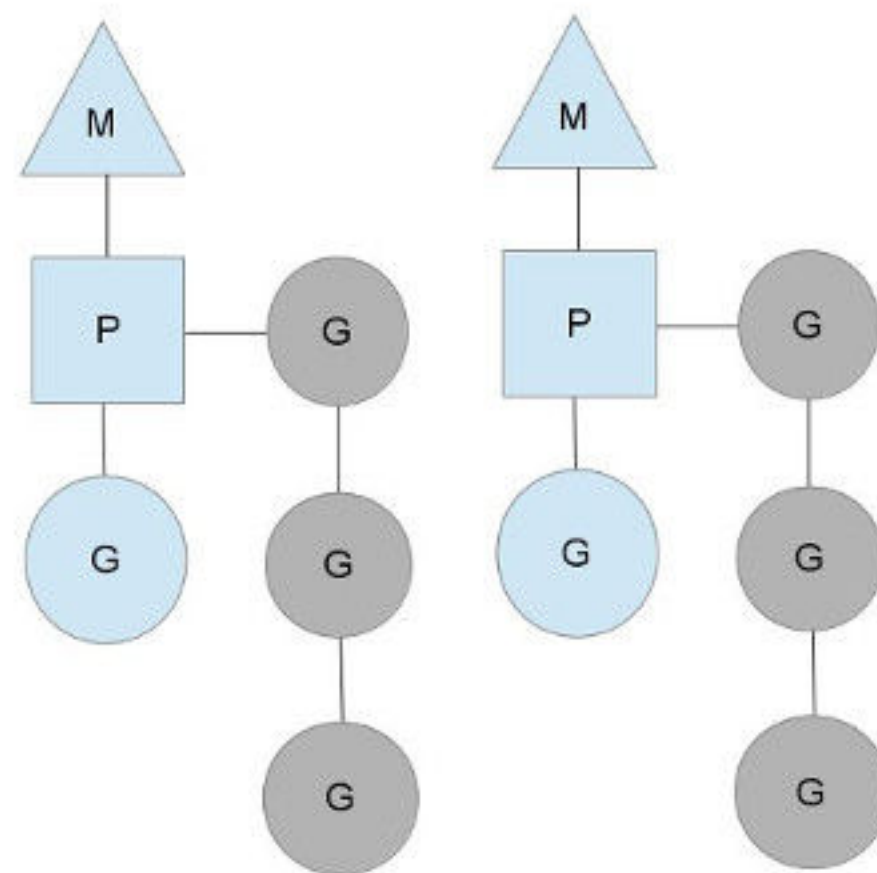
```
func say(s string) {  
    for i := 0; i < 5; i++ {  
        time.Sleep(100 * time.Millisecond)  
        fmt.Println(s)  
    }  
}
```

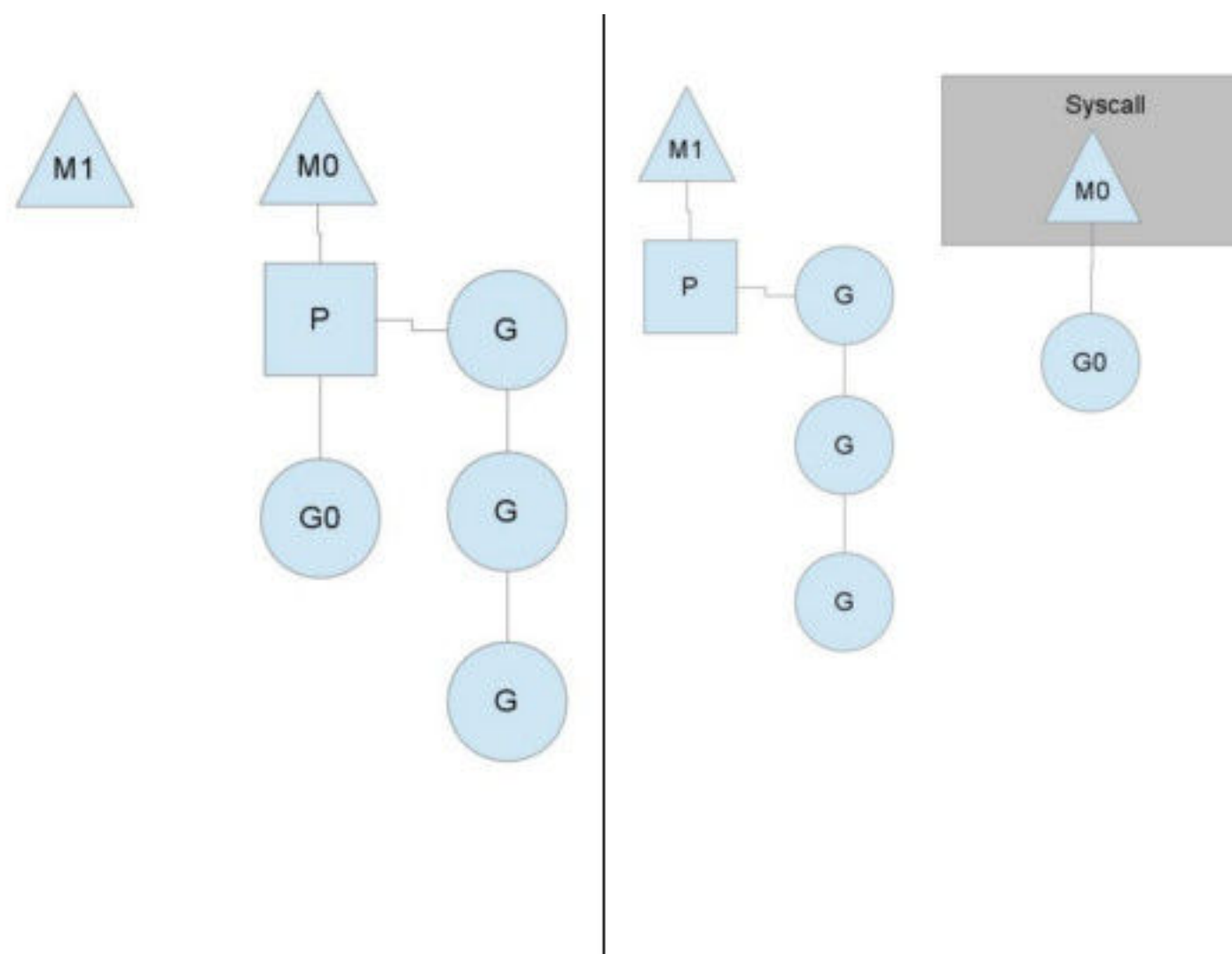
```
func main() {  
    go say(s: "world")  
    say(s: "hello")  
    //hello  
    //world  
    //world  
    //hello  
    //world  
    //hello  
    //hello  
    //world  
    //hello  
    //world  
}
```

- Go的调度器内部有三个重要的结构：M，P，G
- M:代表真正的内核OS线程，真正干活的人
- G:代表一个goroutine，它有自己的栈，用于调度。
- P:代表调度的上下文，可以把它看做一个局部的调度器，使go代码在一个线程上跑。



- 有2个物理线程M，每一个M都拥有一个context（P），每一个也都有一个正在运行的goroutine
- 图中灰色的那些goroutine并没有运行，而是处于ready的就绪态，正在等待被调度。P维护着这个队列
- 每有一个go语句被执行，队列就





- 当一个OS线程M0陷入阻塞时，P转而在OS线程M1上运行。调度器保证有足够的线程来运行所有的context P
- 当M0返回时，它必须尝试取得一个context P来运行goroutine

- channel是一种数据结构，用来在协程之间传递数据

```
func sum(s []int, c chan int) {
    sum := 0
    for _, v := range s {
        sum += v
    }
    c <- sum // 把 sum 发送到 c
}

func main() {
    s := []int{7, 2, 8, -9, 4, 0}

    //创建一个channel
    c := make(chan int)

    go sum(s[:len(s)/2], c)
    go sum(s[len(s)/2:], c)
    x, y := <-c, <-c // 从通道 c 中接收

    fmt.Println(x, y, x+y) //-5 17 12
}
```

发送方会阻塞直到接收方从通道中接收了值

- 带缓冲区的channel

```
func main() {  
    // 这里我们定义了一个可以存储整数类型的带缓冲通道  
    // 缓冲区大小为2  
    ch := make(chan int, 2)  
  
    // 因为 ch 是带缓冲的通道，我们可以同时发送两个数据  
    // 而不用立刻需要去同步读取数据  
    ch <- 1  
    ch <- 2  
  
    // 获取这两个数据  
    fmt.Println(<-ch)  
    fmt.Println(<-ch)  
}
```

发送方则会阻塞直到发送的值被拷贝到缓冲区内；如果缓冲区已满，则意味着需要等待直到某个接收方获取到一个值。接收方在有值可以接收之前会一直阻塞

- defer用于延迟执行，在函数return之前执行，一般用于资源的释放

```
func writeFile(filename string, content []byte) {  
    f, err := os.Open(filename)  
    if err != nil {  
        panic(err)  
    }  
    defer f.Close()  
  
    _, err = f.Write(content)  
    if err != nil {  
        fmt.Println(err.Error())  
    }  
  
    return  
}
```

- 程序运行异常时会触发panic，也可以手动触发

```
func badCall() {
    panic(v: "bad end")
}

func test() {
    defer func() {
        if e := recover(); e != nil {
            fmt.Printf(format: "Panicing %s\r\n", e)
        }
    }()
    badCall()
    fmt.Printf(format: "After bad call\r\n")
}

func main() {
    fmt.Printf(format: "Calling test\r\n")
    test()
    fmt.Printf(format: "Test completed\r\n")
}

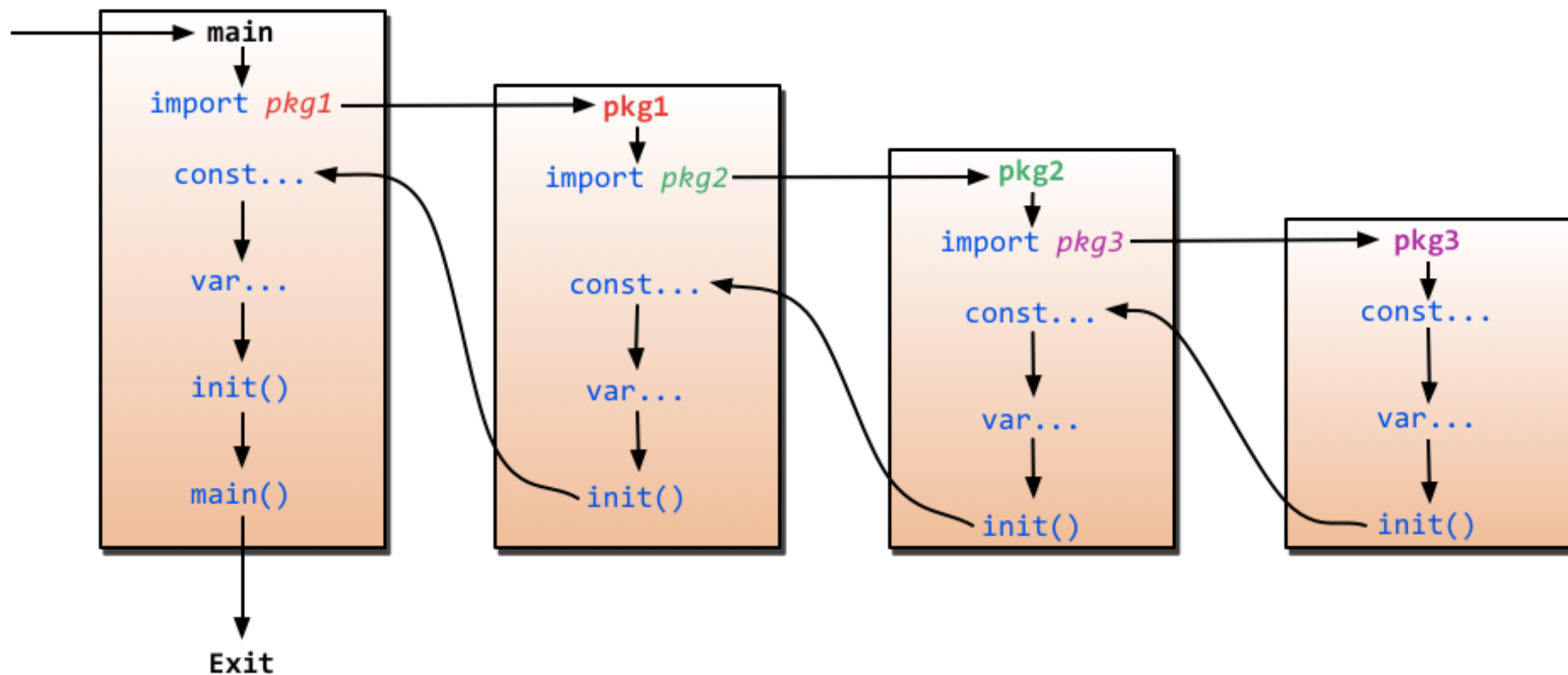
//Calling test
//Panicing bad end
//Test completed
```



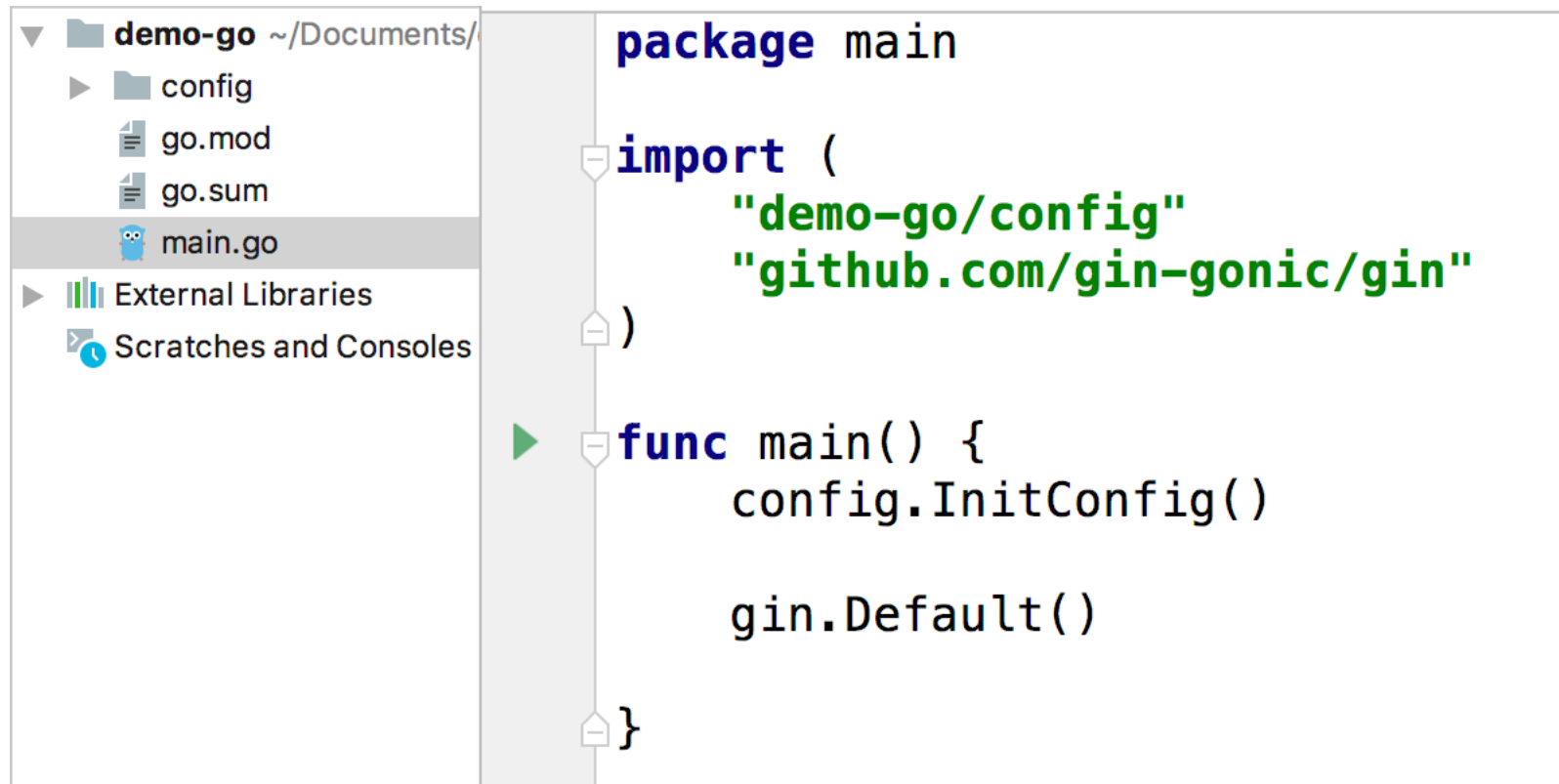
# 项目结构

- package是golang最基本的分发单位，是工程管理中依赖关系的体现
- 每个golang源代码文件开头都拥有一个package声明，表示该golang代码所属的package
- 同一个路径下只能存在一个package，一个package可以拆成多个源文件组成
- import关键字导入的是package路径，而在源文件中使用package时，才需要package名

# 项目结构-执行顺序



- 默认GOPATH \$home/go
- go get将第三方包下载到GOPATH中
- go.mod文件管理第三方依赖
- go.mod定义了模块，是一系列包的集合



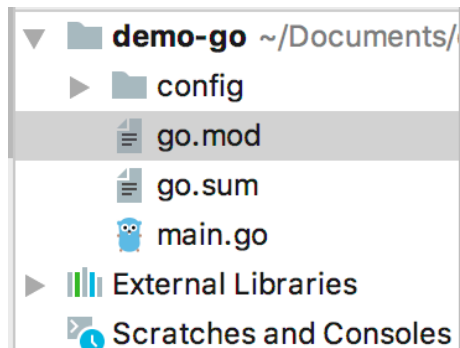
The screenshot shows an IDE interface. On the left, a file explorer displays the project structure: a folder named 'demo-go' at the path '~/Documents/' contains a subfolder 'config' and files 'go.mod', 'go.sum', and 'main.go'. The 'main.go' file is selected. On the right, the code editor shows the following Go code:

```
package main

import (
    "demo-go/config"
    "github.com/gin-gonic/gin"
)

func main() {
    config.InitConfig()

    gin.Default()
}
```



```
module demo-go
```

```
require (
```

```
    github.com/gin-contrib/sse v0.0.0-20170109093832-22d885f9ecc7 // indirect
```

```
    github.com/gin-gonic/gin v1.3.0
```

```
    github.com/golang/protobuf v1.2.0 // indirect
```

```
    github.com/matttn/go-isatty v0.0.4 // indirect
```

```
    github.com/ugorji/go/codec v0.0.0-20181209151446-772ced7fd4c2 // indirect
```

```
    gopkg.in/go-playground/validator.v8 v8.18.2 // indirect
```

```
    gopkg.in/yaml.v2 v2.2.2 // indirect
```

```
)
```

进阶

- 单元测试、性能测试
- GC
- CGO
- pprof
- GDB
- Select
- ...

结束