Golang语言基础

--针对无基础或初学者

Huayulei_2003@hotmail.com
2018/01/15

目录

Golang简介：起源和历史

起源简介：

- Go语言起源2007年，并于 2009 年正式对外发布。
- 项目三位领导者：Robert Griesemer(参与开发 Java HotSpot 虚拟机)；Rob Pike(Go 语言项目总负责人，贝尔实验室 Unix 团队成员，参与的项目包括 Plan 9，Inferno 操作系统和 Limbo 编程语言)；Ken Thompson(贝尔实验室 Unix 团队成员，C 语言、Unix 和 Plan 9 的创始人之一，与 Rob Pike 共同开发了 UTF-8 字符集规范)
- Ian Lance Taylor 于 2008 年 5 月创建了一个 gcc 前端
- Russ Cox 加入开发团队后着手语言和类库方面的开发，

Fig 1.1: The designers of Go: Griesemer, Thompson and Pike

Release版本发布历史

- Go 1.9 (August 2017)
- Go 1.8 (February 2017)
- Go 1.7 (August 2016)
- Go 1.6 (February 2016)
- Go 1.5 (August 2015)
- Go 1.4 (December 2014)
- Go 1.3 (June 2014)
- Go 1.2 (December 2013)
- Go 1.1 (May 2013)
- Go 1 (March 2012)

- 2007年9月21日：雏形设计
- 2009年11月10日：首次公开发布
- 2010年1月8日：当选 2009 年年度语言
- 2010年5月：谷歌投入使用
- 2011年5月5日：Google App Engine 支持 Go 语言

Golang简介： 优缺点

优点：

- 性能高：支持goroutines ，channel，高并发，效率高
- 学习成本：上手容易，时间成本低，方便其他语言工程师转入
- 开发效率：自带package丰富，开源package比较多，让工程师更多的关注业务，开发效率高
- 编译：静态编译，编译过程耗时短，生成可执行程序小
- 部署：运行基本不依赖其他库，部署容易
- 测试：语言自带单元测试框架，编写单元测试简单；测试框架丰富（GoMock、GoConvey等）
- 性能调试：原生支持(pprof库包)
- 跨平台：原生支持Windows，Linux，FreeBSD，Darwin等

缺点：

- 依赖包管理：对package管理比较弱，需要借助开源工具或者其他方案
- 不支持类，继承，多态等特性,面向对象属性和方法支持不够强大（对比C++,Java等）
- 其他：语言本身非常灵活，写法多样，很难形成规范，这给代码review和审计等带来一定的难度

Golang简介：学习资源

- 安装包：https://golang.org/dl/
- 文档：https://golang.org/doc/
- 语法：https://tip.golang.org/ref/spec
- 入门资源：

https://tour.golang.org/welcome/1

https://golang.org/doc/effective_go.html

https://gobyexample.com/

https://github.com/golang/go/wiki

https://github.com/golang

https://godoc.org/

- Books：

https://github.com/golang/go/wiki/Books

- 视频：

https://www.youtube.com/

- 一些知名开源项目：

https://github.com/docker/docker-ce

https://github.com/kubernetes/kubernetes

https://github.com/coreos/etcd

https://github.com/pingcap/tidb

https://github.com/hashicorp/consul

https://github.com/CodisLabs/codis

https://github.com/astaxie/beego

https://github.com/prometheus/prometheus

https://github.com/gin-gonic/gin

…… ……

基本语法：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

基本语法：运算符

优先级	运算符
高	* / % << >> & &^
	+ - | ^
	== != < <= >= >
	<-
	&&
低	||
	可使用使用括号来临时提升某个表达式的整体运算优先级

注：Binary operators of the same precedence associate from left to right. For instance, x / y * z is the same as (x / y) * z.

基本语法：数据类型（1/2）

类型	长	零值	说明
bool	1	false	true or false
byte	1	0	等同于uint8
rune	4	0	等同于int32。存储Unicode Code Point
int/uint		0	平台有关，32 or 64。在 AMD64/X86-64 平台是 64 位整数
int8/uint8	1	0	范围：-128~127；0~255
int16/uint16	2	0	范围：-32768~32767；0~65535
int32/uint32	4	0	范围：-2147483648~2147483647；0~4294967295
int64/uint64	8	0	范围：-9223372036854775808~9223372036854775807；0~18446744073709551615
float32	4	0.0	精确到 7 个小小数位
float64	8	0.0	精确到 15 个小小数位

基本语法：数据类型（2/2）

类型	长	零值	说明
complex64	8	0.0	
complex128	16	0.0	
uintptr		nil	足够保存指针的 32 位戒 64 位整数
array		nil	值类型，如:[2]int
struct		nil	结构体，值类型。无零值，自动实例化
string		””	值类型。多行时可用“`”声明
slice		nil	引用类型，如:[]int
map		nil	引用类型
channel		nil	引用类型
interface		nil	接口类型
function		nil	函数类型
Pointer		nil	可以在一个类型前面使用*号，来表示对应的指针类型，也可以在可寻址的变量前面使用&，来获取其地址。

基本语法：内置函数

- close：关闭channel。
- len：获取string、array、slice的长度，map key的数量，以及缓冲channel的可用数据数量。
- cap：获取array的长度，slice的容量，以及缓冲channel的最大缓冲容量。
- new：通常用于值类型，为指定类型分配初始化过的内存空间，返回指针。
- make：仁用于slice、map、channel这些引用类型，除了初始化内存， 还负责设置相关属性。
- append：向slice追加（在其尾部添加）一个或多个元素。
- copy：在不同slice间复制数据。
- delete:从map中删除key对应的element。
- print/println：打印standard error信息，不支持format，要格式化输出，要使用fmt包。
- complex/real/imag：复数处理。
- panic/recover：错误处理。

基本语法：常量之Const

例子1：

```
const (
size int64 = 1024
Pi float64 = 3.14159265358979323846
eof        = -1  // untyped integer constant
)
const a, b, c = 3, 4, "foo"  // a = 3, b = 4, c = "foo", untyped integer and string
const u, v float32 = 0, 3    // u = 0.0, v = 3.0
```

例子2： type TimeDay int

```
const (
Sunday TimeDay  = iota        //  0
Monday                        //  1
Tuesday                       //  2
Wednesday //   3
Thursday                      //   4
Friday                        //   5
Partyday                      //   6
numberOfDays  // this constant is not exported
)
```

基本语法：变量

声明：

```
var i int
var U, V, W float64
```

声明并赋值：

```
var k = 0    // int
var x, y float32 = -1, -2
var (
i       int
u, v, s = 2.0, 3.0, “bar” // float64, float64, string
)
```

其他：

```
var d = math.Sin(0.5) // d is float64

var _, found = entries[name]  // map lookup;
                      // only interested in "found”

var ErrNilKey   =  errors.New("key is null")

var m map[string]int

var c chan
```

声明并赋值(仅在函数内部有效)：

```
i, j := 0, 10

f := func() int {

 return 7

}

ch := make(chan int)

r, w := os.Pipe(fd)

i2, s2 := 123, “hello”//

array1 := [...]{1,2,3}
```

基本语法：string

```go
package main

import "fmt"

func main() {
    s1 := "abcdefg中国"
    fmt.Printf("s1: %d,%s\n", len(s1), s1[2:3])
    // => s1: 13,c
    ba1 := []byte(s1)
    ba1[2] = 'C'
    s2 := string(ba1)
    fmt.Printf("s2: %s\n", s2)
    // => s2: abCdefg中国
    r2 := []rune(s2)
    fmt.Printf("r2 (rune array): %d,%v\n", len(r2), r2)
    // => r2 (rune array): 9,[97 98 67 100 101 102 103 20013 22269]
    fmt.Printf("Raw string:\n%s\n", `a\t \r\nb`)
    // => Raw string: a\t \r\nb
}

```

注：golang使用utf-8编码，判断带有中文字符串的长度时一定要转成rune。

基本语法：array

- 类型 [n]T 是一个有 n 个类型为 T 的值的数组。
- 数组的长度是其类型的一部分，因此数组不能改变大小

例子1：

```
var a [10]int
```

定义变量 a 是一个有十个整数的数组。

例子2：

```
var a [2]string
a[0] = "Hello"
a[1] = "World"
for idx, val := range a {
fmt.Printf("%v:%v\n", idx, val)
}
```

基本语法：slice

- 一个 slice 会指向一个序列的值，并且包含了长度信息。
- 一个 nil 的 slice 的长度和容量是 0
- len(s) 返回 slice s 的长度。
- slice 可以包含任意的类型，包括另一个 slice
- slice 由函数 make 创建，make(T, n) slice slice of type T with length n and capacity n ；make(T, n, m) slice slice of type T with length n and capacity m
- slice 可以重新切片，创建一个新的 slice 值指向相同的数组。如s[lo:hi]表示
从 lo 到 hi-1 的 slice 元素，含前端，不包含后端。
- 当使用 for 循环遍历时，每次 range 将返回两个值。 当前下标（序号）和对应元素拷贝。

例子1：

```
s := []int{2, 3, 5, 7, 11, 13}

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

fmt.Printf("s[%d] == %d\n", i, s[i])

}
```

例子2：

```
var arr1 []int

arr1 = append(arr1, 1)

arr1 = append(arr1, []int{2, 3, 4}...)

// => arr1 (2):[1 2 3 4]

arr2 := make([]int, 5)

n:=copy(arr2, arr1[1:4])

// => 3 copied, arr2 (2):[2 3 4 0 0]
```

基本语法：map

- map 映射键到值。
- map 在使用之前必须用 make 来创建；值为 nil 的 map 是空的，并且不能对其赋值
- map 的文法跟结构体文法相似，不过必须有键名。
- 在 map m 中插入或修改一个元素：m[key] = elem；获得元素：elem = m[key]；删除元素：delete(m, key)；通过双赋值检测某个键存在：elem, ok = m[key]；如果key在m中，ok为true。否则，ok为false，并且elem是map的元素类型的零值。
- 当从 map 中读取某个不存在的键时，结果是 map 的元素类型的零值

例子1：

```
m := make(map[string]int)
m["A1"] = 42
m["A2"] = 48
delete(m, "A1")
v, ok := m["A5"]
fmt.Println("The value:", v, "Present?", ok)
```

基本语法：for循环

Go只有一种循环结构： for 循环。

基本的 for 循环包含三个由分号分开的组成部分：

- 初始化语句：在第一次循环执行前被执行
- 循环条件表达式：每轮迭代开始前被求值
- 后置语句：每轮迭代后被执行

初始化语句一般是一个短变量声明，这里声明的变量仅在整个 for 循环语句可见；循环初始化语句和后置语句都是可选的。

如果条件表达式的值变为 false，那么迭代将终止

例子1：

```
sum := 0
for i := 0; i < 10; i++ {
sum += i
}
```

例子2：

```
for sum < 1000 {
sum += sum
}
```

例子3：

```
stringMap := map[int]string{1: "A", 2: "B", 3: "C"}
for k, v := range stringMap {
fmt.Printf("%d: %s\n", k, v)
}
```

基本语法：if语句

- if 语句不要求用（）将条件括起来， 必须有{ }
- 跟 for 一样， if 语句可以在条件之前执行一个简单语句。由这个语句定义的变量的作用域仅在 if 范围之内。
- 在 if 语句定义的变量同样可以在任何对应的 else 块中使用

例子1：

```
if x < 0 {
return x
}
```

例子2：

```
if v := math.Pow(x, n); v < lim {
return v
}
```

例子3：

```
if v := math.Pow(x, n); v < lim {
return v
} else {
fmt.Printf("%g >= %g\n", v, lim)
}
```

基本语法：switch语句

- switch语句除非以 fallthrough 语句结束，否则分支会自动终止
- switch 的条件从上到下的执行，当匹配成功的时候停止。
- 没有条件的 switch 同 switch true 一样。

例子1：

```
switch os := runtime.GOOS; os {
case "darwin":
fmt.Println("OS X.")
case "linux":
fmt.Println("Linux.")
default:
fmt.Printf("%s.", os)
}
```

例子2：

```
t := time.Now()
switch {
case t.Hour() < 12:
fmt.Println("Good morning!")
case t.Hour() < 17:
fmt.Println("Good afternoon.")
default:
fmt.Println("Good evening.")
}
```

基本语法：函数和方法

函数：

```
func min(x int, y int) int {
if x < y {
return x
}
return y
}
```

函数内部特殊用法：

```
go func() {
// code
}()
```

方法：

方法可与命名类型或命名类型的指针关联。

如：

```
func (p *Point) Length() float64 {
return math.Sqrt(p.x * p.x + p.y * p.y)
}
func (p *Point) Scale(factor float64) {
p.x *= factor
p.y *= factor
}
```

问题：红色部分如果是func (p Point) Scale(factor float64) 会有什么区别

基本语法：再谈函数

- 函数可作为参数
- 函数可作为返回值
- 函数允讲有多返回值。可用 占位符“_”扔掉
- 函数可以是一个闭包。闭包是一个函数值，它引用了函数体之外的变量。 这个函数可以对这个引用的变量进行访问和赋值；换句话说这个函数被“绑定”在这个变量上。如：

```
func adder() func(int) int {
sum := 0
return func(x int) int {
sum += x
return sum
}
}
```

例子：

```
func GenMyFunc(hash func(string) int64, content string) {
 return func() string {
return fmt.Sprintf("Content Hash: %v", hash(content))
}
}
```

调用上述函数例子：

```
myFunc := GenMyFunc(func(s string) int64 {
result, _ := strconv.ParseInt(s, 0, 64)
return result
}, "0x10")
fmt.Printf("%s\n", myFunc())
```

基本语法：接口

- 接口是由一组方法定义的集合。
- 接口类型的值可以存放实现这些方法的任何值。
- 隐式接口解藕了实现接口的包和定义接口的包：互不依赖;如：

```
type Writer interface {
Write(b []byte) (n int, err error)
}
var w Writer
// os.Stdout 实现了 Writer
w = os.Stdout
if sw, ok := w.(Writer); ok { //code
}
```

例子：

```
type Abser interface {
Abs() float64
}
type Vertex struct {
X, Y float64
}
func (v *Vertex) Abs() float64 {
return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

调用上述函数例子：

```
var a Abser
a = &Vertex{3, 4}
fmt.Println(a.Abs())
```

基本语法：接口的继承

- 当一个类型包含（内嵌）另一个类型（实现了一个或多个接口）的指针时，这个类型就可以使用（另一个类型）所有的接口方法。如：

1）type Task struct {

Command string

*log.Logger

}

2）这个类型的工厂方法像这样：

```
func NewTask(command string, logger *log.Logger) *Task {
return &Task{command, logger}
}
```

3）当 log.Logger 实现了 Log() 方法后，Task 的实例 task 就可以调用该方法：

```
task.Log()
```

- 类型可以通过继承多个接口来提供像 多重继承 一样的特性：

```
type ReaderWriter struct {
*io.Reader
*io.Writer
 }
```

注：上面概述的原理可被应用于整个包，多态用得越多，代码就相对越少

基本语法：reflect

- 反射可以在运行时检查类型和变量，例如它的大小、方法和 动态 的调用这些方法。
- 变量的最基本信息就是类型和值：reflect.TypeOf 和 reflect.ValueOf，返回被检查对象的类型和值
- 可通过反射修改(设置)值，可使用 CanSet() 方法测试是否可设置,然后设置值
- 有些时候需要反射一个结构类型。使用 NumField() 方法返回结构内的字段数量；通过 for 循环用索引取得每个字段的值 Field(i)。

注：reflect是一个强大的工具，除非真有必要，否则应当避免使用或小心使用

例子1：

```
var x float64 = 3.4
v := reflect.ValueOf(x)
fmt.Println("type:", v.Type())
fmt.Println("kind:", v.Kind())
```

例子2：

```
v = reflect.ValueOf(&x) // address
v = v.Elem()
fmt.Println("settability of v:",
v.CanSet())
v.SetFloat(3.1415) // setval
```

例子3：

```
s := reflect.ValueOf(&t).Elem()
for i := 0; i < s.NumField(); i++ {
 f := s.Field(i)
fmt.Printf("%d: %s %s = %v\n", i,
typeOfT.Field(i).Name, f.Type(),
f.Interface())}
```

基本语法：error

- Go 程序使用 error 值来表示错误状态：

  error 类型是一个内建接口：type error interface { Error() string }
- 通常函数会返回一个 error 值，调用的它的代码应当判断这个错误是否等于 nil， 来进行错误处理：error 为 nil 时表示成功；非 nil 的 error 表示错误。

例子：

```
i, err := strconv.Atoi(input)
if err != nil {
fmt.Printf("couldn't convert number: %v\n", err)
return err
}
```

基本语法：defer语句

- defer 语句会延迟函数的执行直到上层函数返回。延迟调用的参数会立刻生成，但是在上层函数返回前函数都不会被调用。
- 延迟的函数调用被压入一个栈中。当函数返回时， 会按照后进先出的顺序调用被延迟的函数调用。

例子1：

```
func ReadFile(filePath string) error {
file, err := os.Open(filePath)
if err != nil {
return err
}
defer func() {
file.Close()
}()
return nil
}
```

例子2：

```
for i := 0; i < 10; i++ {
defer fmt.Println(i)
}
```

基本语法：panic 和recover

- 在多层嵌套的函数调用中调用 panic，可以马上中止当前函数的执行，所有的 defer 语句都会保证执行并把控制权交还给接收到 panic 的函数调用者。这样向上冒泡直到最顶层，并执行（每层的） defer，在栈顶处程序崩溃，并在命令行中用传给 panic 的值报告错误情况：这个终止过程就是 panicking。
- recover内建函数被用于从 panic 或 错误场景中恢复：让程序可以从 panicking 重新获得控制权，停止终止过程进而恢复正常执行。recover 只能在 defer 修饰的函数中使用：用于取得 panic 调用中传递过来的错误值，如果是正常执行，调用 recover 会返回 nil，且没有其它效果。panic 会导致栈被展开直到 defer 修饰的 recover() 被调用或者程序中止

例子：

panic("A severe error occurred: stopping the program!")

例子：

```go
package main

import (
    "fmt"
)

func badCall() {
    panic("bad end")
}

func test() {
    defer func() {
        if e := recover(); e != nil {
            fmt.Printf("Panicing %s\r\n", e)
        }
    }()
    badCall()
    fmt.Printf("After bad call\r\n") // no running
}

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

并发编程：概述

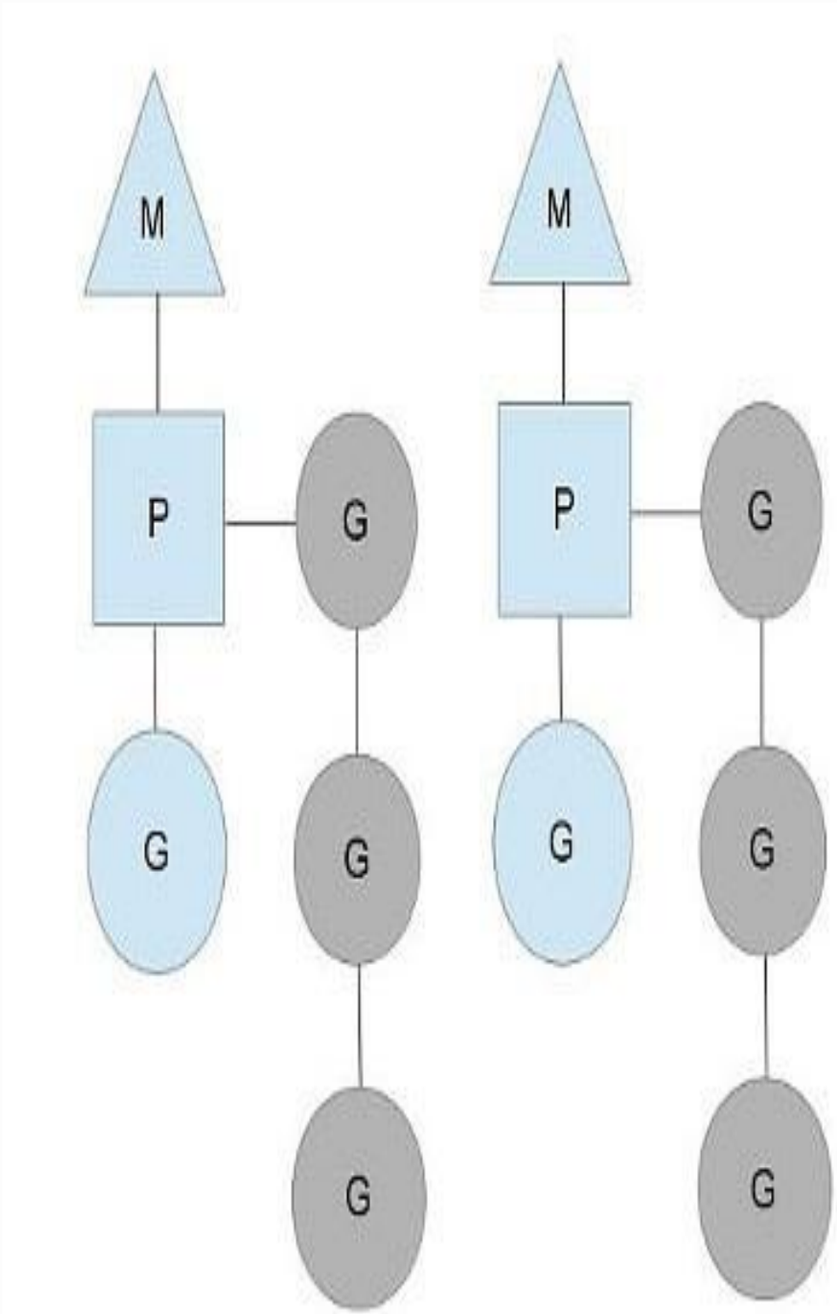
- goroutines 一种比线程更轻量，比协程更灵活的语言级开发机制。
- channel 通道，可以在普通和Goruntine场景下作为数据甚至代码的载体和通讯 手段。
- sync 提供了一些经典的并发同步机制，比如锁和原子操作。

goroutines之所以被叫做goroutines，是因为现存的一些术语——线程、协程和进程等等——都传达了错误的涵义。

——摘自官方文档《Effective Go》

golang的口号：Do not communicate by sharing memory. Instead, share memory by communicating. （不要通过共享内存来通信，而通过通信来共享内存。）

并发编程：goroutine高级协程



- G: 表示goroutine，存储了goroutine的执行stack信息、goroutine状态以及goroutine的任务函数等；另外G对象是可以重用的。
- P: 表示逻辑processor，P的数量决定了系统内最大可并行的G的数量（前提：系统的物理cpu核数>=P的数量）；P的最大作用还是其拥有的各种G对象队列、链表、一些cache和状态。
- M: M代表着真正的执行计算资源。在绑定有效的p后，进入schedule循环；而schedule循环的机制大致是从各种队列、p的本地队列中获取G，切换到G的执行栈上并执行G的函数，调用goexit做清理工作并回到m，如此反复。M并不保留G状态，这是G可以跨M调度的基础。

并发编程：调度器(1/2)

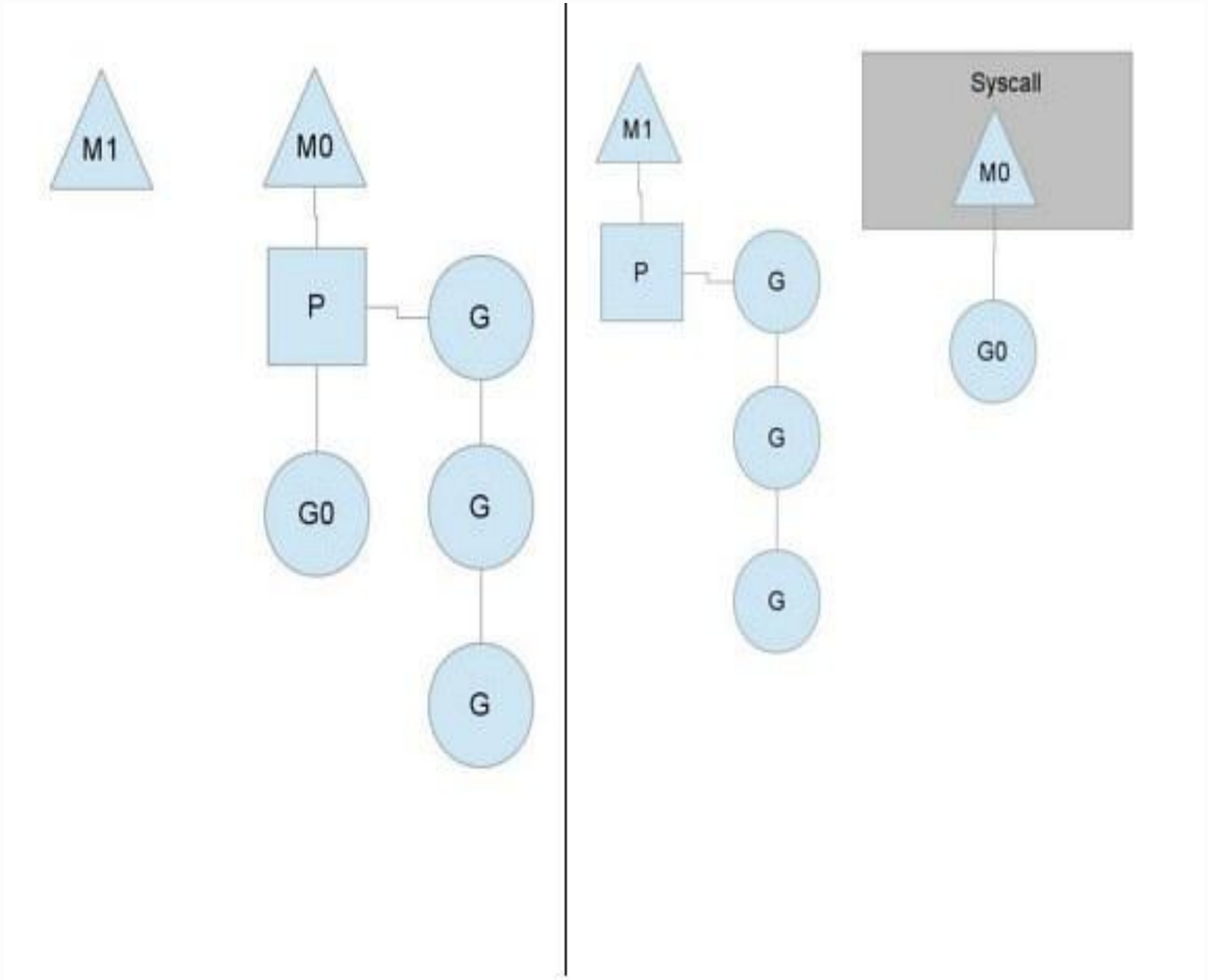
1.抢占式调度的原理是在每个函数或方法的入口，加上一段额外的代码，让runtime有机会检查是否需要执行抢占调度。实现：如果一个G任务运行10ms，sysmon就会认为其运行时间太久而发出抢占式调度的请求。一旦G的抢占标志位被设为true，那么当这个G下一次调用函数或方法时，runtime便可以将G抢占，并移出运行状态，放入P的local runq中，等待下一次被调度

2. Go runtime实现了netpoller，这使得G发起网络I/O操作也不会导致M被阻塞（仅阻塞G），从而不会导致大量M被创建出来。但是对于regular file的I/O操作一旦阻塞，那么M将进入sleep状态，等待I/O返回后被唤醒；这种情况下P将与sleep的M分离，再选择一个idle的M。如果此时没有idle的M，则会新创建一个M，这就是大量I/O操作导致大量Thread被创建的原因。

3. channel阻塞或network I/O情况下的调度：如果G被阻塞在某个channel操作或network I/O操作上时，G会被放置到某个wait队列中，而M会尝试运行下一个runnable的G；如果此时没有runnable的G供m运行，那么m将解绑P，并进入sleep状态。当I/O available或channel操作完成，在wait队列中的G会被唤醒，标记为runnable，放入到某P的队列中，绑定一个M继续执行。

4. system call阻塞情况下的调度:如果G被阻塞在某个system call操作上，那么不光G会阻塞，执行该G的M也会解绑P，与G一起进入阻塞状态。如果此时有idle的M，则P与其绑定继续执行其他G；如果没有idle M，但仍然有其他G要去执行，那么就会创建一个新M。当阻塞在syscall上的G完成syscall调用后，G会去尝试获取一个可用的P，如果没有可用的P，那么G会被标记为runnable，之前的那个sleep的M将再次进入sleep。

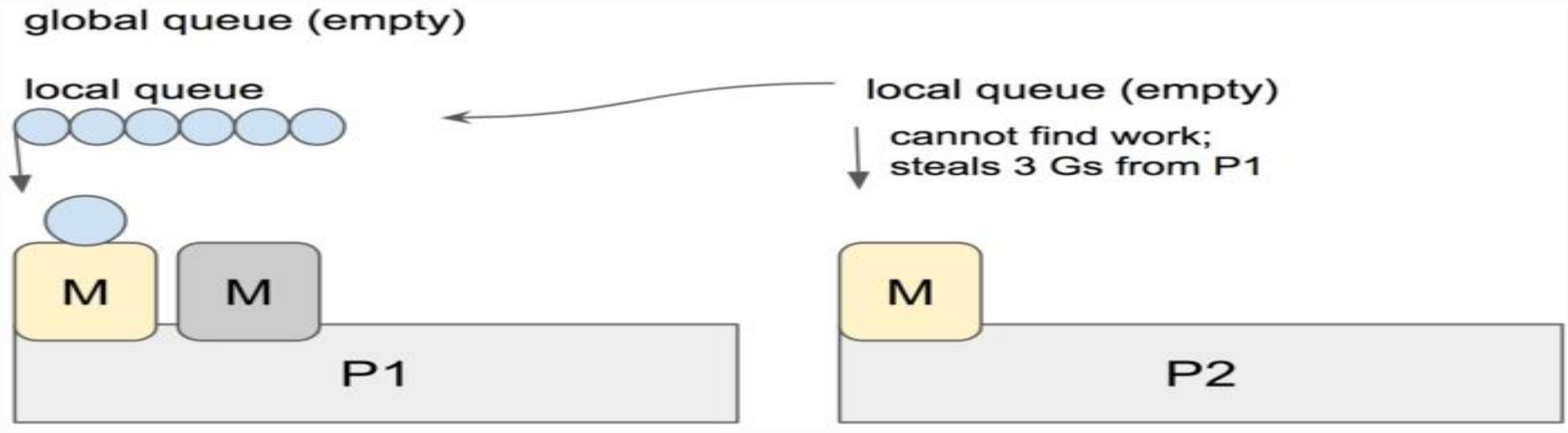
并发编程：调度器(2/2)



1.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。图中的M1可能是被创建，也可能从线程缓存中取出。

2.当M0返回时，它必须尝试取得一个context P来运行goroutine，一般情况下，它会从其他的OS线程那里steal偷一个过来，如果没有偷到，它就把goroutine放在global runqueue里。然后把自己放到线程缓存中或者转入睡眠状态。

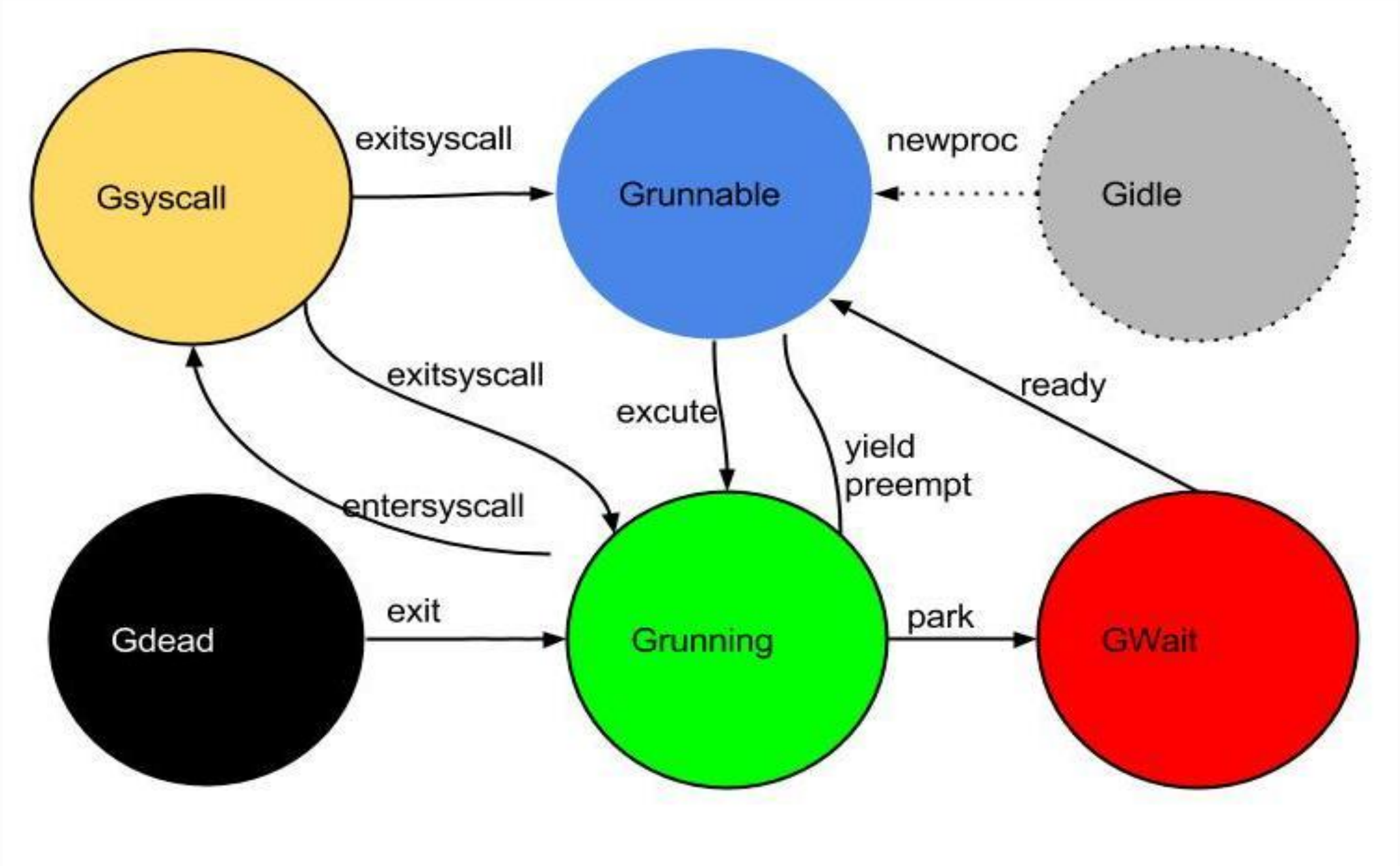
并发编程：G的偷取




```
runtime.schedule() {
// only 1/61 of the time, check the global runnable queue for a G.
 // if not found, check the local queue. // if not found,
// try to steal from other Ps.
// if not, check the global runnable queue.
// if not found, poll network.
}
```

在上图中，P2 这个处理器无法找到任何可执行的 goroutines。因此，它随机选择另一个处理器 P1，并将 3 个 goroutines 偷到自己的局部队列中。P2 将执行这些 goroutines，而调度器也将在多个处理器之间更均衡的调度。

并发编程：goroutine状态迁移



并发编程：channel基础

Go推荐使用channel来处理goroutine之间的同步问题。

- channel 是有类型的管道，可以用 channel 操作符 <- 对其发送或者接收值
- channel 使用前用make创建，无缓冲channel必须要一个接收者准备好接收通道的数据然后发送者可以直接把数据发送给接收者。在另一端准备好之前，发送和接收都会阻塞。
- channel 可以是带缓冲的，向带缓冲的 channel 发送数据的时候，只有在缓冲区满的时候才会阻塞。 而当缓冲区为空的时候接收操作会阻塞。
- 发送者可以 close channel 表示没有值再被发送

例子1：

```
var ch1 chan = make(chan int) // 非缓冲
ch2 := make(chan int, 5) // 有缓冲
```

例子2：

```
package main
import "fmt"
func main() {
ch1 := make(chan int)
go pump(ch1) // pump hangs
fmt.Println(<-ch1) // print  0
}
func pump(ch chan int) {
ch <- 0
 }
```

并发编程：goroutine和channel(1/2)

- for 循环的 range 语句可以用在通道 ch 上，便可以从通道中获取值；读取数据直到通道关闭，才会继续执行
- Channel是可以关闭的，使用逗号，ok 操作符：用来检测通道是否被关闭。如： v, ok := <-ch // ok is true if v received value。通常和 if 语句一起使用：if v, ok := <-ch; ok { process(v) }
- 从不同的并发执行的channel中获取值可以通过关键字select来完成。当任何被监听的channel中都没有数据的时候，select语句块会阻塞，此时当有default时，default子句将会被执行；当多个被监听channel中都有数据时，则select会随机选择一个case 执行。

例子1：

```
ch := make(chan int)
go func(){
// doSomething
ch <- val
}()
// doSomething
result <- ch
```

后台服务模式例子：

```
func backend() {
for {
 select {
case cmd := <-ch1: // Handle ...
case cmd := <-ch2: ...
case cmd := <-chStop: // stop server
}
}
}
```

并发编程：goroutine和channel(2/2)

time包里提供了一些功能可以与channel结合使用：

- After函数:起到定时器的作用，指定的纳秒后会向返回的channel中放入一个当前时间(time.Time)的实例。
- Tick函数:起到循环定时器的作用，每过指定的纳秒后都会向返回的channel中放入一个当前时间(time.Time)的实例。
- Ticker结构:循环定时器。Tick函数就是包装它来完成功能的。该定时器可以被中止。

例子1：// 检查超时

```
for {
select {
case <-ch: // do something
case <-time.After(timeoutNs): // call timed out
break
}
}
```

例子2：//定时器

```
ticker := time.NewTicker(TickerTime)
defer ticker.Stop()
for {
select {
case <-stop:  // stop handle
case <-ticker.C:  //do handle
}
}
```

并发编程： sync

- Mutex 是互斥锁：使用Lock和Unlock来加锁或者解锁。
- Once是执行一次操作的一个对象：使用Do(f func())来调用执行函数。
- RWMutex是一个读/写者互斥锁。锁可以由任意数量的读或单个写所有；使用Lock（）来写；使用RLock来读；使用Unlock来解锁写；使用RUnlock来解锁读。
- WaitGroup用来管理一组goroutines。The main goroutine calls Add() to set the number of goroutines to wait for. Then each of the goroutines runs and calls Done() when finished. At the same time, Wait() can be used to block until all goroutines have finished.

例子：

```
self.lock.Lock()
defer self.lock.Unlock()
```

例子：// WaitGroup

```
var wg sync.WaitGroup
var urls =
[]string{"http://www.golang.org/"}
for _, url := range urls {
          wg.Add(1)
          go func(url string) {
                    defer wg.Done()
                    fmt.Println(url)
          }(url)
}
fmt.Println("wait.")
wg.Wait()
fmt.Println("done.")
```

常见问题和错误：

1. 何时使用new()和make()？

   切片、映射和通道，使用make

   数组、结构体和所有的值类型，使用new

2. 不需要将一个指向切片的指针传递给函数

   切片实际是一个指向潜在数组的指针，

3. 误用短声明导致变量覆盖

```
var remember bool = false
if something {
  remember := true //错误
}
```

4. 使用低效的字符串操作

go中使用诸如a += b形式连接字符串效率低下，尤其在一个循环内部。这会导致大量的内存开销和拷贝。应该使用一个字符数组代替字符串，将字符串内容写入一个缓存中。

```
var b bytes.Buffer ...
for condition {
  b.WriteString(str) // 将字符串str写入缓存buffer
}
return b.String()
```

常见问题和错误：

5. 不要使用一个指针指向一个接口类型，因为它已经是一个指针。正确使用：

```
type nexter interface { next() byte }
func nextFew1(n nexter, num int) []byte {
 }
```

6. 正确理解context

https://golang.org/pkg/context/

https://blog.golang.org/context

7. 闭包和协程的正确使用:见右图

8. 创建slice时，在能确定长度和容量的情况下，参数一定要指定。因为增加切片的容量必须创建一个新的、更大容量的切片，然后将原有切片的内容复制到新的切片，这种操作非常耗费资源。尽量使用func make([]T, len, cap) []T。

9. 理解和使用init函数：对同一go文件的init()调用顺序是从上到下的；对于同一package中不同go文件是按文件名字符串比较“从小到大”顺序调用各文件中init()函数；对于不同的package，如果不相互依赖的话，按照main包中“先import后调用”的顺序调用其包中的init()；如果package存在依赖，则先调用最早被依赖的package中的init()。init()先于main()执行

```go
package main

import (
    "fmt"
    "time"
)

var values = [5]int{10, 11, 12, 13, 14}

func main() {
    // 版本A:
    for ix := range values { // ix是索引值
        func() {
            fmt.Print(ix, " ")
        }() // 调用闭包打印每个索引值
    }
    fmt.Println()
    // 版本B: 和A版本类似，但是通过调用闭包作为一个协程
    for ix := range values {
        go func() {
            fmt.Print(ix, " ")
        }()
    }
    fmt.Println()
    time.Sleep(5e9)
    // 版本C: 正确的处理方式
    for ix := range values {
        go func(ix interface{}) {
            fmt.Print(ix, " ")
        }(ix)
    }
    fmt.Println()
    time.Sleep(5e9)
    // 版本D: 输出值:
    for ix := range values {
        val := values[ix]
        go func() {
            fmt.Print(val, " ")
        }()
    }
    time.Sleep(1e9)
}
```

Q&A

Thank you very much !