

GoLang 随笔杂谈

本想着写完，但是琐事太多，累了！换个活法，开个小店，希望大家来捧场。

我的微店：https://weidian.com/?userid=423856015&wfr=wechatpo_keywords_shop

我的公众号：lingzhuge8866

最近我迷上了Rust语言和人工神经网络，作为C++入行的我还是受不了GC类语言，C++程序员对于性能效率有着本能地渴望和追求！但是C++却又过于复杂，令程序员的心智负担过重！而Rust语言的优雅和效率令我折服！我先学习一下，以后再来分享！

0. 作者

良子、老子，别号尘了，初学电脑软件专业于98年，默默无闻，单机，网络，分布式，互联网，云计算，职场，创业，一路走来青丝已染白发，得失计较终是空，无所得，终耽误于忙碌！廉颇老矣，尚能饭否？！代码行空，灿若流星，终究追不可得，时反省，真真想让自己的心净下来！看云舒云卷！如能将点滴愚见赋予笔端，也算对得起敲过的键盘！

我的通信方式：QQ：285779289，微信号：justice_forever_123
Email：htyu_0203_39@sina.com 请多指正，交流学习。

1. 缘起

真真地厌倦了互联网，赋闲在家，本想躬耕田野，无奈寸土寸金，唯阳台还可养几盆花草！想着只要不搞电脑，出去练摊也是好的，终也是眼高手低，思虑敏捷，手脚迟钝，手指却是灵活，真是百无一用是码农！何况一老码农呼！当今之社会皆于一个“快”字穷追不舍！美名曰“互联网思维”！码农老矣，不敢苟同！

闲来写个小文，不求深邃，只求工程实效，经验之谈！谬误难免，只做抛砖引玉。

提示：golang 1.8.3 centos7/win10 我也只是小学生，讲得不透彻，深入的研究请看前人专著，原理讲得不多，多求工程实效，也多是经验之谈，难免谬误，请多包含，如果有现成的例子，则尽力不重复造轮子，拿来就用，只求文字和例子能说明问题就好，力求通俗。

2. 指针类型

- (1) 定义形如：var p *int；或 p := &变量名
- (2) 指针空值为：nil
- (3) 不允许直接算术运算，如：p++, p += x，此写法在C/C++中很普遍，但允许间接算数运算

(a) 正确例子

```
var x struct {  
    a bool  
    b int16  
    c []int
```

}

//以下语句意思等价于：pb := &x.b

```
pb := (*int16)(unsafe.Pointer(uintptr(unsafe.Pointer(&x))  
+ unsafe.Offsetof(x.b)))  
*pb = 42  
fmt.Println(x.b) //output: "42"
```

//注解：间接流程：具体类型的指针必须先转化为`unsafe.Pointer`类型的指针，其相当于C/C++的`void`指针，然后再将`unsafe.Pointer`转化为`uintptr`，这才允许做算术运算，运算完毕后，再按相返顺序转化回具体类型的指针，最后读取之；在Go语言中只允许数值类型参加算术运算，为什么Go语言不能像C/C++语言那样，允许指针随意做算数运算来调用指针的指向，这是语言哲学问题，首先可以随意自由对指针做算术运算调整其指向，这是C/C++工程开发时出问题最多的地方；再者Go语言拥有GC和栈自动伸缩，所以变量在内存中会被移动，变量的所有指针指向也会被自动调整，所以指针算术运算后的地址值很可能已无效！所以这样的运算是徒劳的，而且Go只允许具体类型的指针只有转化为`uintptr`类型时方可参加算术运算，Go只帮你自动调整好指针的新地址，而`uintptr`是数值类型，Go不可能帮你自动调用的！

(b) 错误例子

```
//警告：此为微妙错误  
tmp := uintptr(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b)  
pb := (*int16)(unsafe.Pointer(tmp))  
*pb = 42
```

//错误理由在上例中已经说过了，`tmp`的类型为`uintptr`，这不是一指针类型，只是一个数值类型，所以当变量`x`被Go Runtime 和GC移动时，此变量的所有指针都会被自动调整为新的地址值，但是`tmp`不是指针类型，所以不会被自动调整为正确的地址值，若其后再将`tmp`转化为`*int16`的指针类型时，`pb`的指向很可能是已失效的地址！后果自然可知！

(c) 错误例子

```
pT := uintptr(unsafe.Pointer(new(T)))  
//没有指针指向由new(T)新分配的变量，所以Go GC发现这个变量是孤立的，没人指向它，或者说引用它，所以被当做垃圾回收了！
```

(4) `&amap[k]` 是错误的，无法获取此元素的地址

(5) `&slice[i]` 可以获取`slice`中某个元素的地址，其实这个地址就是`slice`指向的`array`

对应元素的地址

(a) 例子

```
//a slice, 左闭右开区间
var myArray [10] int = [10] int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
var mySlice [] int = myArray[5:]
fmt.Println(mySlice)
fmt.Println(myArray[5] == mySlice[0]) //true
fmt.Println(&myArray[5] == &mySlice[0]) //true
//特别在并发编程时，更要注意（5）点，因为很可能同一个array的不同
slice的区间 //有可能重叠，那么在并发编程时
一定要注意是否需要加锁保护，以免数据被破坏！
```

3. 包

(1) 包名小写，不要有下划线

(2) 包名最好与所在目录同名

(3) 包内成员首字母大写的，则包外可见可访问，若是小写，则仅包内可见可访问

(4) 包搜索顺序和优先级

(a) 例子--目录结构

```
[root@localhost go_path]# tree
```

```
.
├── bin
├── pkg
└── src
```

```
    ├── main
    │   └── main.go
```

└── runtime // 需删除，只为证明： vendor 的优先级高于 go 标准库；

报错信息证明引用的是本地工程目录 vendor 中的 runtime 包；

| └── runtime.go // 标准库的优先级高于工程中非 vendor 目录，可知此时 import 的包为标准库中的 runtime 包；若将 runtime 包移动到 go_path/src/vendor 中，则报错信息表明此时引用的为 go_path/src/vendor/ 中的 runtime 包。

```
    ├── test
    |   └── p
    |       └── x.go
    └── unit
```

```
|   └── unit.go
|   └── vendor
|       └── test
|           └── p
|               └── x.go
|
└── unit2
    ├── unit1
    |   ├── my
    |   |   └── vendor
    |   |       └── test
    |   |           └── p
    |   |               └── x.go
    |   ├── unit1.go
    |   └── vendor
    |       └── test
    |           └── p
    |               └── x.go
    └── vendor
        └── test
            └── p
                └── x.go
```

26 directories, 10 files

(b) 测试代码

源文件 x.go:

```
package p
```

```
import (
```

```
        "fmt"
        "runtime"
    )


```

```
func P() {
    _, file, _, _ := runtime.Caller(0) //输出当前源文件完整路径
    fmt.Println(file)
}
```

源文件 go_path/src/unit/unit.go

```
package unit
```

```
import (
```

```
    "fmt"

```

```
    "runtime"

```

```
    p1 "test/p" //首先引用go_path/src/unit/vendor/test/p; 若无则
引用go_path/src/vendor/test/p; 若无则去/usr/local/go/src/ (from $GOROOT) 和
go_path/src/ (from $GOPATH) 中按照包路径比配搜索此包, 若无则报错停止.
)
```

```
func P() {
```

```
    _, file, _, _ := runtime.Caller(0)

```

```
    fmt.Println("here: " + file)

```

```
    p1.P()

```

```
}
```

源文件 go_path/src/unit2/unit1/unit1.go

```
package unit
```

```
import (
```

```
    "fmt"

```

```
    "runtime"

```

```
    "test/p" //首先引用go_path/src/unit2/unit1/vendor/test/p; 若无
则依次向自己的父目录中找vendor, 若无则向父目录的父目录中找vendor; 以此类推, 直
```

到go_path/src为止；若无则直接去/usr/local/go/src/ (from \$GOROOT)和go_path/src中搜索你引用的"包路径"；若无则报错停止。

)

```
func P() {
    _, file, _, _ := runtime.Caller(0)
    fmt.Println("here: " + file)
    p.P()
}
```

源文件 go_path/src/main/main.go

```
package main
import (
    "fmt"
    "runtime"
    p1 "test/p" //p1 为引用包别名
    //p2 "unit/test/p" //找不到此包路径，因为它在unit/vendor中
    //p21 "unit/vendor/test/p" //不允许包引用路径中包含vendor.
    p3 "unit"
    p4 "unit2/unit1"
)
```

```
func main() {
    _, file, _, _ := runtime.Caller(0)
    fmt.Println("here: " + file)
    p1.P()
    //p2.P()
    p3.P()
    p4.P()
}
```

源文件 go_path/src/runtime/runtime.go

```
package runtime
```

```
func Caller(index int) (string, string, string) {  
    return "test", "test", "test"  
}
```

(c) 总结

(c1) vendor只是个依赖包管理概念，不允许import包路径中出现它

(c2) 从引用此包的源文件所在当前目录开始寻找vendor，依次搜索其父目录，父目录的父目录，直到go_path/src为止；若无则放弃查找vendor，或是虽有vendor，但“包路径”不匹配，此时vendor失效，则去/usr/local/go/src/ (from \$GOROOT) 中搜索你引用的“包路径”；最后依次遍历GOPATH，搜索匹配每一个gopath/src中你引用的“包路径”

(c3) 一旦找到vendor，则以vendor为起点目录，在其中匹配搜索“包路径”；好比之前的src为包的搜索起点目录

(c4) main.go也就是main包不可以直接放在go_path/src下，否则引用不了vendor中的包路径，go报错找不到你的包，好比vendor机制失效

(c5) 强调一下c2中描述的优先级顺序：本工程目录的vendor > go标准库/usr/local/go/src > \$GOPATH/src

(5) 包初始化

(c1) 定义形如：

```
func init() {  
    //包被引用时，初始化此包的代码。  
}
```

(c2) go允许同一个包内每一个源文件都可以定义init包初始化函数，且每一个源文件中可以同时重复定义多个init，每一个init，go保证且仅执行一次。经验之谈，一个包如果需要就定义一个init包初始化函数就好，咱们工程开发简单最好，别玩花活，作茧自缚！此主题深入研究请看：雨痕著作《Go语言学习笔记》

4. array

(1) 定义形如：

```
var a [5]int  
var twoD [2][3]int
```

```
b := [...] int{1, 2, 3, 4, 5}  
var c = [5]int{0, 1, 2, 3, 4}
```

(2) 彻底的值语义

```
var a = [5]int{0, 1, 2, 3, 4}  
b := a //数组b完整地复制了a  
//---  
func f( c [5]int ) { //... }  
f(a) //数组c完整地复制了a；切不可以为如C/C++那样只是传了个指针，没有  
复制数组！  
//验证也容易，分别取两数组同一index元素的地址比对一下，就知道是否复  
制了！  
//切记别做无用功，明确你想要的是“引用”还是“复制”，以免  
浪费效率和内存空间。
```

(3) 数组*index* 范围

```
index := [0, size -1] 或 [0, size) //size为数组大小哟
```

(4) 赋值和传参的类型匹配

```
q := [3]int{1, 2, 3}  
q = [4]int{1, 2, 3, 4} // compiler error: cannot assign [4]int to  
[3]int
```

//切记定义数组时指定的size也是数组类型的一部分，而go又是强类型语
言，不同类型不可以直接赋值！！对于函数形参也是如此，所以通常将函数形参定义为
slice类型，从而规避数组类型的死板规定。

5. slice

(1) 不同slice引用同一个数组的不同，相同，重叠的区间

```
var a = [10]int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
s1 := a[:5]
s2 := a[5:]
intersect := a[3:7]
```

//注意：切片s1, s2, intersect此时都指向同一个数组a；所以对于重叠区间元素的修改大家都会看到，切忌哟！切片的这个特性在我们写并发程序时，尤其需要注意！以免数据被破坏，当然善用此特性可以提高并发度！

(2) slice读写越界

```
s1[5] = 100 //从上例可知切片s1的size为5，且切片与数组一样index皆从0开始！
```

```
//报错信息： panic: runtime error: index out of range
//goroutine 1 [running]:
//main.main()
//      E:/go_path/src/main.go:27 +0x475
//exit status 2
```

(3) slice“悄悄”指向新的array

```
fmt.Println(len(s1)) //输出:5
fmt.Println(cap(s1)) //输出: 10
s1 = append(s1, 100, 101, 102, 103, 104, 105)
/*切片s1中已经存在5个元素，所以append还可以再追加5个元素，此时没有超出切片s1的cap，切片s1仍然指向原数组a！但是当超出5个，多了个元素105时，append函数发现底层数组存储空间不够，所以就会新建一个数组用于存储更多的元素，从而实现切片的自动伸展！注意此时切片s1不再指向原数组a，对于想通过切片的引用特性来修改函数外的数组数据时，尤其要注意，别自己不小心切断了“引用关系”，还不自知！做了功，外面的数据却没有改变！少加班吧！对自己好一点！*/
```

(4) 切记slice为引用类型哟

切片是go语言的引用类型，其结构如：`type slice struct { array unsafe.Pointer, len int, cap int}`，值语义，对于它的复制和传递所占用空间很小，就上面3个成员！其实它就是数组的封装，令数组可以被灵活使用和扩展！所以通常定义函数形参时使用切片类型而非数组类型！

6. map

(1) map 为引用类型

map 为引用类型，所以赋值=和函数传参时，不用担心数据集合被整体复制；无序集合，存入map 中的 value 集合是无序的，所以不要按你存入元素的顺序凭空想象有序，这是不对的；空值为 nil。

(2) 对 key 的要求

对 key 要求比较简单，key 必须是可比较的，使用 ==。

(3) &amap[k] 是不可以的

amap[k] 并不是一个 go 语言变量，所以取不到地址。

(4) 如何确定 key 是否存在

形如： value, ok := amap[key] if ok { // 找到了 } else { // 没找到 }

(5) 创建并初始化 map 时，逗号要记牢

```
amap := map[string]int {  
    "hi": 7788, // 注意：最后一行的逗号必须有，否则编译报错哟！  
}
```

(6) 并发读写保护

其实提这个有些多余，因为但凡编写并发程序，对于“全局资源”和“临界区”都要做保护措施，比如加锁串行化；不只是对 map 需要这样！

7. struct

(1) 形如：

```
type person struct {  
    name string  
    age int  
}
```

// 在 go 中，一个 struct 就简洁高效地实现了“基于对象”与“面向对象”的语义功能，没有了 C++ 中 struct 与 class 的混淆；它是值语义的，所以在赋值和传参时，它会复制的哟；可以为它定义方法，形如：

```

type rect struct {
    width, height int
}

func (r *rect) area() int {
    return r.width * r.height
}

func (r rect) perim() int {
    return 2*r.width + 2*r.height
}

```

//注意：关于go语言的“值语义”和“引用语义”的讲解，我推荐：《Go语言编程》许式伟 吕桂华等编著，人民邮电出版社；我就不再班门弄斧地复述了；一个严谨地程序员在学习和使用任何一门编程语言时，都要弄清“值语义”和“引用语义”的差异，以免做无用功，出现不必要的复制和不小心的状态修改，切记哟，别挖坑埋自己。

(2) 组合实现继承，既匿名组合

```

type Base struct {
    name string
}

func (b *Base) Foo() { //... }

type Foo struct {
    Base
}

//或者

type Foo struct {
    *Base //此定义方式只是灵活一些，可以随后再指定Base或其子类的实例
指针
}

```

//子类Foo相当于继承了Base，故可以通过Foo调用到Base的方法，同时可以访问父类Base的成员变量，如果同名，需加前缀：Base.，当然如果子类Foo也定义了相同的方法，则通过子类调用此方法时，则调用子类的同名方法，如果想调用父类的，需要加前

缀：Base.；父类Base的方法访问不到子类Foo的成员变量和方法；详细清晰的讲解，我推荐：《Go语言编程》 许式伟 吕桂华等编著， 人民邮电出版社。

(3) 为field 加上tag

```
type Response2 struct {
    Page    int      `json:"page"`
    Fruits []string `json:"fruits"`
}
```

//tag具有很好的说明作用， json及xml解析器会使用它们； 注意哟红色的标点可不是单引号哟， 一般键盘最左侧有个波浪线按键， 那个键上就有上面红色标点。详细讲解，我推荐《Go程序设计语言》 艾伦 A. A. 多诺万 布莱恩 W. 柯尼汉著。

8. interface

还是直接来个例子代码吧， 取自：

<https://gobyexample.com/interfaces>

```
package main
```

```
import (
    "fmt"
    "math"
)
```

//geometry 为接口名， 接口为go语言中的抽象数据类型， 其声明了一个方法集， 具体类型只要实现了这个接口的方法集， 就被认为是此接口的一个实例； 查看go语言接口的实现代码， 可以其为结构体， 其内定义两个指针成员， 所以除:slice， map， channel为引用类型外， 接口也算是个引用类型。

```
type geometry interface {
    area() float64
    perim() float64
}
```

```
type rect struct {
    width, height float64
}
```

```
type circle struct {
    radius float64
}
```

```

//具体类型实现了上面接口的全部方法集
func (r rect) area() float64 {
    return r.width * r.height
}
func (r *rect) perim() float64 {
    return 2*r.width + 2*r.height
}

func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}
func (c *circle) perim() float64 {
    return 2 * math.Pi * c.radius
}

//以接口类型为此函数形参类型，则可以在函数调用时，接受所有实现了接口方法集的具体类型值或指针作为实参传递。
func measure(g geometry) {
    //由此可知接口不光可以调用具体类型实现的方法，还可以存储具体类型的值或指针
    fmt.Println(g)
    //通过此接口可以调用具体类型实现的接口方法
    fmt.Println(g.area())
    fmt.Println(g.perim())
}

func main() {
    r := rect{width: 3, height: 4}
    c := circle{radius: 5}
    //以值为实参调用
    //编译报错信息
    //interface.go:42: cannot use r (type rect) as type geometry in argument
    //to measure:
    // rect does not implement geometry (perim method has pointer receiver)
    //interface.go:43: cannot use c (type circle) as type geometry in
    argument to measure:
    // circle does not implement geometry (perim method has pointer receiver)
    //measure(r)
    //measure(c)

    //以指针为实参调用
    measure(&r)
    measure(&c)
}

```

8.1. 分别通过interface和struct调用方法的不同

我们用*struct*实现某个接口的方法集时，可以将*receiver*定义为值类型，也可以定义为指针类型，当我们通过*struct*实例调用方法时，可以用其值也可以用其指针，不管此方法的*receiver*是值类型，还是指针类型，*go*语言会帮我们自动转化！当然程序员要注意产生复制或修改共享数据的问题！但是如果你是通过接口来调用具体类型实现的方法时，*go*语言不会再帮你把“值”和“指针”做自动转化，以适应相应的*receiver*类型，所以上例中红色部分代码会报错！！！所以我们在实现接口的方法集时，*receiver*一律定义为指针类型，或是值类型，抑或两者兼而有之，此需要我们设计时考虑清楚！当然我们也可以走另一条路，就是调用时一律以“具体类型实例的指针”来赋值给接口！

9. 零值

在*Go*语言中，未进行显式初始化的变量都会被自动初始化为该类型的零值，如：

bool : *false*, *int* : 0; *string* : 空字符串，指针: *nil*，(引用类型: *slice*, *map*, *channel*, *interface*) : *nil*，*struct* : 每一个成员皆为其类型对应零值, *function values*: *nil* 等等。切记不要小看零值问题，有的语言不会帮你将未初始化的变量自动设为其类型零值的，在实际工程开发时，这是最爱出问题的点，状态有时对有时不对，或刚开始不对，后来又对等等！粗心的程序员好辛苦哟！

10. *switch*

此处想说的不多，只两点：每一个*case*无需*break*来明确退出此*case*；其二：在*case*中需明确添加*fallthrough*关键字，才会继续执行紧跟的下一个*case*！这和以前的*C/C++*完全相反。*go*语言的*switch*可比*C/C++*中的灵活的多呀，如*case*中条件表达类型就放宽了好多，详情请看我参考列表中的书籍。

11. *for range* 坑

(1) *v*为值

```
package main

import (
    "fmt"
    "time"
)
```

```

type field struct {
    name string
}

func (p *field) print() {
    fmt.Printf("print: p: %p, v: %s\n", p, p.name)
}

func main() {
    data := []field{{"one"}, {"two"}, {"three"}}

    for _, v := range data {
        // 注意:for语句中的迭代变量(如: v)在每次迭代时被重新使用,一直复用
        go v.print()
        // 注意: 此处可理解为: go (&v).print(), 也就用的是v的指针去调用, 而且v
        // 会在每次迭代时复用, 所以每一个调用的receiver都是共同指向v的指针, 而且v在最后
        // 一次迭代后, 被复制为:"three", 所以才有了打印出3个"three"的结果.

    }
    //程序执行结果: three, three, three
    time.Sleep(6 * time.Second) //偷懒才用sleep方式同步, 工程中不要用哟!

    //解决方法: (a) aCopy := v 复制v, 加到go v.print()上一句, 紧挨着
    (2) 把field::print的receiver由指针类型改为值类型。
}

```

(2) v 为指针型

```

package main

import (
    "fmt"
    "time"
)

type field struct {
    name string
}

func (p *field) print() {
    fmt.Println(p.name)
}

func main() {
}

```

```

data := []*field{{"one"}, {"two"}, {"three}} //注意：此处变为指针类型啦
for _, v := range data {
    go v.print()
}

//v本身就是指针，迭代时会改变指向，分别指向one, two, three 所以作为
//receiver直接调用不复制，不存在如上例(&v).print()的问题，因为v就是指针！！！
//所以相当于：("one"结构体地址).print(), ("two"结构体地址).print()
//, ("three"结构体地址).print(); 所以就不存在上例(1)中的问题啦！！！

}

//程序执行结果：one, two, three

time.Sleep(6 * time.Second) //再次警告，只是为了例子偷懒这么写，实际工程
中，不要把sleep用作同步工具哟!
}

```

总结：

- (1) Pointer Receiver不复制，所以当以值方式调用时，直接 &Value取地址指针；而不是先复制后，再取副本变量的地址。
- (2) 若 for _, v := range data中的v本身就是指针，则直接作为Pointer Receiver来调用。
- (3) for range会在每次迭代中复用v变量。
- (4) go语言本身是值语义的，也就是说传参，调用，迭代都会复制，只不过像：指针，引用类型只是复制了其本身，而非其指向的data，这样复制代价很低很低。
- (5) stackoverflow中的解释：

这在Go中是个很常见的技巧。for语句中的迭代变量在每次迭代时被重新使用。这意味着你在for循环中创建的闭包将会引用同一个迭代变量v。

12. 协程闭包

协程就是闭包，闭包就会劫持其所处代码上下文中自己用到的变量，对于“值类型变量”复制一份，对于引用和指针也会复制其值，同时会增加指向关系，相当于智能指针的增加引用计数！这样GC就会维护一个新的指向关系，不会提前清理回收资源了！来个例子：

```
package main

import "fmt"

func intSeq() func() int {
    // 红色代码中的匿名函数就是个闭包

    // 黄色的变量 i 被闭包使用了， 所以被劫持，此变量被快照，复制了一份，

    快照值：0，生成副本 i

    i := 0

    return func() int {
        // 副本 i 的初始值为：0

        i += 1

        return i
    }
}

func main() {

    nextInt := intSeq()
    fmt.Println(nextInt())
    fmt.Println(nextInt())
    fmt.Println(nextInt())
    newInts := intSeq()
    fmt.Println(newInts())
}

/* 执行结果

1
2
3
1
*/
```

//为什么会是这个结果，有些同学转不过弯来！关键点：闭包劫持了**黄色变量**i
后，只要闭包生命周期没有完结，则它会一直持有这个**“绿色副本i变量”**
所以闭包执行结果才是累加的！有些同学以为一直为1吧！那就错了！！！

12.1. 协程与线程

我感觉“协程”与早年的“用户态线程”好像！都是为了减轻操作系统的负担，减少跨进程和线程的执行上下文切换负担，对于协程而言，其在用户空间，且又在自己的进程线程上下文之内，切换轻量且容易，好比自家有好多间房子，我随意搬到那间直接就去住，只需搬很少的东西！但是如果你是搬出你家，比如你家拆迁了，那你要搬好多好多东西，非常沉重的！所以协程的好处不言自明。

12.2. 协程泄漏(如何从外部杀死协程)

对于线程，我们可以从其外部杀死或取消它，如：`pthread_kill()`,
`pthread_cancel()`等等，我认为这非常有必要，比如在某线程中Call一个阻塞型API，且此API没有超时功能，故其一旦阻塞就不会再返回，则此线程一直僵死！此线程所持有的资源也不可回收，相当于资源泄漏了，此时提供从外部杀死线程的机制就非常必要了。其实协程也存在同样情况，但是直到golang1.8.3版本，仍然没有提供从外杀死或取消协程的机制！golang的协程除了持有资源，还有defer的执行问题；有些棘手！所以golang的协程倾向自己select退出信号，但这是死循环了！除非保证golang调用的所有API都是具有超时能力的！不会一直阻塞僵死！哪个更完美呢，也许以后会出现完美的机制。但是我们在做项目时一定要考虑到，避免协程不断僵死，吃光所有资源！对于高并发服务器程序尤其要关注！别让绳子勒到自己的脖子！

14. 函数一等公民

现在了解一下Function Values(直译为函数值)，在go中函数和方法为一等公民，可以作为值任意传递保存，或从函数中返回。

(1) 普通函数

如：

```
func square(n int) int { return n * n }
func negative(n int) int { return -n }
```

```

func product(m, n int) int {return m * n}

//...
f := square
fmt.Println(f(3)) // "9"

f = negative
fmt.Println(f(3)) // "-3"
fmt.Printf("%T\n", f) // "func(int) int"

f = product //compile error: can't assign f(int, int) int to f(int) int
//由此看来：函数类型包括： 形参类型， 形参数量， 形参顺序， 函数返回参数
（类型， 数量， 顺序）； 不包括： 函数名， 形参名， 返回参数名。
f = nil //Function Values 零值: nil ;
if f != nil {
    f(3) //注意Function Values 之间不可比较， 只允许与零值nil做比较。
}
//因为Function Values不支持比较， 所以不可以做为map的key， 只可以做其value.

```

(2) 类型实例之方法

形如：

```

p := Point{1, 2}
q := Point{4, 6}

```

distanceFromP := p.Distance //Distance为类型Point的成员方法， p为此类型的
具体实例

fmt.Println(distanceFromP(q)) // 称distanceFromP为 method values， 其
已自动绑定function和receiver.

(3) 类型之方法

形如：

```

p := Point{1, 2}
q := Point{4, 6}

```

```
distance := Point.Distance // method expression, 直译为：方法表达式；  
Point为类型名
```

```
fmt.Println( distance(p, q) ) //Point.Distance的方法签名: "func(Point)  
float64"
```

//distance的函数签名为: func(Point, Point) float64, 可以看出形参多了一个Point, 第一个形参Point是编译器自动加的语法糖作为receiver, 这是固定模式, 我们在调用时, 记得第一个实参传p 作为receiver就好, 其它参数照旧。这个方法表达式的好处就是可以任选同一类型的不同实例作为receiver来调用方法. 详细讲解, 我推荐《Go程序设计语言》 艾伦 A. A. 多诺万 布莱恩 W. 柯尼汉著。

(4) C/C++传统思维作怪

形如：

```
package main

import (
    "fmt"
)

func test(t int) int {
    fmt.Println(t)
    return t
}

var pT func(int) int = test //Go范, pT相当于C/C++函数指针, 正确go写法。
//var ppT *func(int) int = &test //compile error: cannot take the address
of test
//意思说: 不能取函数地址, 这是C/C++作怪, 定义函数指针不加个星号*, 就别
扭, 这在go中是错误的!!!
//var ppT *func(int) int = test //compile error: cannot use test (type
func(int) int) as type *func(int) int in assignment
//终于不去执着取函数地址了, 但还想保留星号*, 哈哈报错类型不匹配, 不许赋
值, 别较劲啦!

func main() {
    pT(1)
    //ppT(2)
}
```

15. 方法之*Receiver* (值, 指针)

以下的测试例子很简单，主要为了验证是否产生了复制，还是原对象！我在运行结果中有简单分析。见下例：

```
package main

import (
    "fmt"
)

type XMan string

func (x XMan) ValueReceiver() {
    fmt.Printf("value receiver: %p\n", &x)
}

func (p *XMan) PointerReceiver() {
    fmt.Printf("pointer receiver: %p\n", p)
}

func main() {

    var x1 XMan
    x1 = "x1"

    fmt.Printf("x1: %p\n", &x1)

    //value call.
    fmt.Println("value call")
    x1.ValueReceiver()
    x1.PointerReceiver()

    //pointer call
    fmt.Println("pointer call")
    p := &x1
    p.ValueReceiver()
    p.PointerReceiver()

}

//运行结果:
$ go run receiver.go
x1: 0xc042008250 //原对象的内存地址
value call
value receiver: 0xc0420082a0 //产生复制， 原对象的副本
pointer receiver: 0xc042008250 //原对象
```

```
pointer call  
value receiver: 0xc0420082c0 //产生复制， 原对象的副本  
pointer receiver: 0xc042008250 //原对象
```

通过对比内存地址， 我们可以明确判断， 那些调用产生了复制， 哪些为原对象！
归总一下： `value receiver` 一定产生复制， 以原对象的副本调用方法； `pointer receiver` 不产生复制， 以原对象调用方法。对于， 赋值， 传参， 调用 等弄明白其是否生产复制， 可以提高效率， 节省空间， 避免状态错乱！

16. 函数多返回值

我只说， 这个语言特性太方便了， 不再需要剥洋葱了， 给个小例子看看就好！ 不要小看特性， 这可是生成力哟！ golang的目标之一就是解放程序员不必要的心智负担， 极大释放生产力哟！

```
package main  
  
import "fmt"  
  
func vals() (int, int) {  
    return 3, 7  
}  
  
func main() {  
  
    a, b := vals()  
    fmt.Println(a)  
    fmt.Println(b)  
    _, c := vals()  
    fmt.Println(c)  
}
```

17. `interface{}` 相当于C/C++的Void

原理我不讲了， 可以看我的参考资料， 只是强调`interface{}` 一般用于函数形参， 用于接收不同类型的实参！， 类似于`void`, `variant`等等。

18. 并发

在这个并发主题下有多很小点，提醒大家规避一些坑，让并发程序稳定可靠高效，在工程开发时，我们不要急于写程序，要先在纸上写写画画，画个UML，写个ppt，组内开发人员交流讨论，多做头脑风暴，将各个关注点考虑到，少走弯路；其实很多公司，特别是互联网公司，一个项目做完了，就只有代码，文档几乎没有，遇到问题只有读代码来还原设计思路！互联网公司开发人员流动性大，最后这个项目就没人能说明白了，只能扔掉，再重新开发，这在大公司特别常见！有钱有人，就要求快！出了问题人力顶，程序员请保重身体！

18.1. 同步互斥

这块我只是简单说一下，一是`golang`语言哲学推荐我们用“协程+Channel”的方式搞定一切！即以通信来实现共享内存，而不是以共享内存来实现通信！这一语言设计大大简化了并发程序的编写！极大地提高了并发效率！但是我们习惯了传统，即是以共享内存方式来实现通讯，所以本能地需要：锁，信号量等传统同步互斥手段，幸好`golang`照顾了我们的习惯，在其`sync`包中提供了这些传统工具。具体的例子代码网上好多，`golang`自带的`test`中也有好多例子，我就不再啰嗦了！

18.1.1. `sleep`不是同步原语

学过操作系统进程线程调度的同学应该知道，进程线程如何执行推进是不可预测的，虽然我们都画过“推进图”，但那只能说明在某一种情况下执行推进的可能性，所以你给`sleep`设定为多长时间合适呢！答案是多少都不合适，比如：时间设短了，人家还没有被执行或是执行完，所以得不到结果；时间设长了，人家早就执行完了，你就是不来取结果，真耽误时间！可是时间设多少合适呢！都不合适，因为你无法准确预测人家什么时候被调试执行，执行多少时间，是否被抢占了，是执行着，还是阻塞着；这些都是操作系统动态的行为！所以千万别用`sleep`之类的时间等待函数来做同步之用！这是不对的！不可以这么用！其实这么用最可怕的情况就是，它有时候执行正常，有时候就不正常了！这是自己挖坑埋自己，千万别做傻事，再说了，有专门的锁，信号量之类的专门工具你不用，你用旁门左道干什么呢？！

18.2. `select`

来个例子，摘自<https://gobyexample.com/>:

```
package main
```

```

import "time"
import "fmt"

func main() {

    c1 := make(chan string)
    c2 := make(chan string)

    go func() {
        time.Sleep(time.Second * 1)
        c1 <- "one"
    }()
    go func() {
        time.Sleep(time.Second * 2)
        c2 <- "two"
    }()
}

for i := 0; i < 2; i++ {
    select {
    case msg1 := <-c1: //是否可读
        fmt.Println("received", msg1)
    case msg2 := <-c2: //是否可读
        fmt.Println("received", msg2)
    }
}
}
}

```

上面例子中的select用于监视 c1 和 c2 两个channel是否发生io事件，如：可读可写，如果有，则随机选择一个case执行，若无io事件，且无default:，则阻塞在那等等io事件到来。io多路复用大家并不陌生，linux/unix都提供类似机制和api，如select，epoll，kqueue等，用于监控文件描述符是否有io事件到来，golang则在语言层面提供支持！只不过golang select与channel 是天生一对，golang select专门监护channel。

18.2.1 select 与for 是好兄弟

select监控每一个case是否有io事件发生，如果有，则随即选择一个case执行，如果无有io事件到来，但是有default: 则执行之；否则阻塞等待；一旦其个case或default被执行了，则select就退出了，如果想持续不断地监控每一个case中的channel是否发生io事件，则需要借助for，一直循环在那里，如上例。

注意：如果某个case的channel为nil，则select忽略此case，不再监控它，白话说是：当这个case不存在。

18.2.2. *select {}*会永远block

我们可以试着在代码中， 加一行：*select {}*语句， 然后编译报错， 错误信息如下：

```
fatal error: all goroutines are asleep - deadlock!  
goroutine 1 [select (no cases)]
```

错误信息的意思为： 一个没有*case*的空*select*会导致所有*go routine*睡去， 死锁！看来*go*的编译器越来越聪明了， 可以分辨出许多错误用法！ 早期曾经见过这种错误用法的代码， 所以提出来警告一下大家， 千万别这么用！

18.2.3. *select*关注点

来个例子简洁说明问题：

```
package main  
  
import (  
    "fmt"  
    "time"  
)  
  
func main() {  
  
    c1 := make(chan string)  
    c2 := make(chan string)  
    c3 := make(chan string)  
    var c4 chan string  
    c5 := make(chan string)  
    close(c5) //关闭c5 channel  
    c6 := make(chan string, 3)  
    c6 <- "buffered 1"  
    c6 <- "buffered 2"  
    c6 <- "buffered 3"  
    close(c6)  
  
    go func() {  
  
        time.Sleep(time.Second * 1)  
        c1 <- "one"  
        close(c1) //关闭c1 channel  
    }()  
}
```

```

go func() {
    time.Sleep(time.Second * 2)
    c2 <- "two"
}()

go func() {
    fmt.Println(<-c3)
}()

for {
    select {

        case msg1, ok := <-c1:
            if ok {
                fmt.Println("received1 ", msg1)
            } else { //ok为false则channel已被关闭
                c1 = nil //判断c1 channel已经关闭后，将其赋值为nil，则此case将被忽略，故此select继续评估其它case，避免了死循环
                fmt.Println("c1 channel has closed.")
            }
        case msg2 := <-c2: //从c2 channel中读
            fmt.Println("received2 ", msg2)

        case c3 <- "write string": //向c3 channel中写
            fmt.Println("c3 channel wrote.")

        case <-c4: //因为c4 is nil，所以此case将被忽略，故select继续评估其它case。
            fmt.Println("never reach here.")

        case c4 <- "never": //因为c4 is nil，所以此case将被忽略，故select继续评估其它case。
            fmt.Println("never reach here.")

        case str := <-c5: //死循环，不断地读取出零值，即空字符串。
            fmt.Printf("empty: %s", str)
        //case c5 <- "panic": //向已经关闭的channel写，发生panic异常。
        //    fmt.Println("panic")

        case str := <-c6: //死循环，在读取完c6中已写入的值之后，则继续不断地读取出零值，即空字符串。
            fmt.Printf("buffered channel value: %s", str)
    }
}

```

```

//case c6 <- "buffered channel closed, write to panic":
//向已经关闭的channel写，发生panic异常。

//fmt.Println("buffered channel closed, write to panic")
case <-time.After(time.Second * 5): //定时器，防止操作无限期阻塞
    fmt.Println("timeout\n")
//default: //无default则select为阻塞等待的；有default则为非阻塞等待
{
    //      fmt.Println("default\n")
}
}
}

```

总结：

`select`中的每一个`case`都是针对`channel`的I/O操作，读或写，当有多个`case`同时就绪（可读可写）时，`select`会随机选择一个执行。

因为`select`的特点是只要其中一个`case`触发，执行完成，则`select`执行完成，程序就会继续从`select`的右大括号后开始执行，而不再会评估其它`case`。

`select {}`将会永远阻塞；通常`select`与`for`并用，这样`select`就可以循环不断地评估每一个`case`，随机选取一个`case`执行。

其实`select`这个概念在linux中早就出现了，用于监视多个文件描述符是否有I/O操作，当有文件描述符发生I/O操作时，`select`调用立即返回，返回值中包括发生I/O操作的文件描述符集合。

18.3. channel

来个例子简洁说明问题：

```

package main

import (
    "fmt"
)

func main() {

    //case 1: 对一个nil channel 进行写将发生阻塞。
    //var c chan int //go语言自动对未初始化的变量赋其类型对应的零值,如: nil,
0, ""等。
    //c <- 1

    //case 2: 对一个nil channel 进行读将发生阻塞。
    //c

    //case 3: 对一个已经关闭的unbuffered channel进行写操作,将发生panic异常。
    //c1 := make(chan int)
    //close(c1)
}

```

```

//c1 <- 1

//case 4: 对一个已经关闭的unbuffered channel进行读操作, 返回channel元素类型的零值, 如:false, 0, nil等
//x := <-c1
//fmt.Println(x)

//case 5: 对一个已经关闭的buffered channel进行读操作, 见以下代码注释.
c2 := make(chan int, 3)
c2 <- 1
c2 <- 2
c2 <- 3
close(c2)
//尽管channel 已经关闭了, 但是我们依然可以从中读出关闭前写入的3个值, 自第4次开始读取时, 则返回该channel元素类型对应的零值: 0,
//即使仍继续不断读取, 仍然是零值.
fmt.Printf("%d\n", <-c2)
fmt.Printf("%d\n", <-c2)
fmt.Printf("%d\n", <-c2)
fmt.Printf("%d\n", <-c2)
fmt.Printf("%d\n", <-c2)
fmt.Printf("%d\n", <-c2)

//case 6: 对一个已经关闭的buffered channel进行写操作, 将发生panic异常.
c2 <- 4

//大原则: 1. 最好由生产者或称写入者负责关闭channel, 这样可以有效避免发生panic异常.
//2. 不带缓冲的channel本身具有通信同步两个特点, channel是线程安全的, 协程安全的, 可多个goroutine共享使用不必加锁保护它.

}

```

18.4. 线程安全与可重入

现在写程序基本上都是并发, 不是线程, 就是协程, 如何保护全局资源被按序有效使用, 全局状态正确一致, 就成了关键点之一! 说白了, 我们都知道, 对于全局资源和变量等需要加锁保护起来! 我们称竞争抢夺全局资源的这部分代码块为临界区, 需要在此代码块的开头加锁, 在其末尾解锁! 这就提出一个新问题, 锁粒度大了, 效率并发展降低, 小了可能保护不到所有全局资源, 所以需要程序员花点时间 考虑清楚, 另一个问题就是“死锁问题”, 如果只有一个全局资源, 一直被某协程霸占, 一直不释放, 则其它协程只有阻塞死等, 这是一种死锁, 通常全局资源数 \geq

2，协程A抢到了资源Q，接着去抢资源P，而协程B抢到了资源P，接着去抢资源Q，两者谁也无法再走下去了，只能阻塞僵持在那，谁也无法抢到对象手中的资源，故此死锁，关于这些方面的详细资料很多，我推荐《C++编程思想》第2卷，《Linux多线程服务器端编程 使用muduo C++ 网络库》，在“C++编程思想”中有一段讲哲学家进餐的问题，其中讲到，死锁发生需同时满足4个条件，摘抄如下，略有删改：

- 1) 相互排斥。说白了，对于资源的使用是互斥的，我用你就不能用，比如打印机。
- 2) 至少有一个进程或协程必须持有某一个资源，并且同时等待获得正在被加外一个进程或协程所持有的资源。
- 3) 不能以抢占的方式剥夺一个进程或协程持有的资源。
- 4) 出现一个循环等待，一个进程等待另外的进程所持有的资源，而这个被等待的进程又等待另一个进程所持有的资源，以此类推直到某个进程去等待被第1个进程所持有的资源。

总结：以上4个条件全部满足才死锁，只要打破一个就不会死锁！言简意赅，对我很是启发，早年读这本书，受益非常大，每次复读都受益非常大！这就是经典的力量吧，经得住时间的考验。

所以，当一个函数中，一个或多个临界区被正确加锁保护时，我们称这个函数是“线程安全”的函数；当然，我希望理想的情况就是程序里没有全局资源，函数代码中使用的都是“局部资源”，别人看不到，当然也就不可能来争抢，对于这样的函数称其为“可重入的”，显然“可重入”要比“线程安全”严苛多了，太难了，我们当然希望程序中每一个函数都是可重入的，但现实是，我们可能总是需要依赖一些全局的资源才可以走下去！

啰嗦这么多，只是提醒大家，在做架构和编码时，了解这些点，有助于我们设计出高效，正确的解决方案，否则的话，高并发就会是个恶梦，低效并难以追踪分析！别等上了线，入了坑再不断加班读log，程序员们爱惜一下自己的身体吧，都是爹妈给的，出门在外，保重自己的身体就是孝顺孩子哟！

18.5. signal

golang对信号处理这块做了很好的简化和封装，非常清晰易用，再加上channel与select的配合，所以在golang中处理信号相当容易，来个例子吧，此例取自：

<https://gobyexample.com/signals>

```
package main

import "fmt"
import "os"
import "os/signal"
import "syscall"

func main() { //主go routine, 相当于我们多线程中的主线程

    sigs := make(chan os.Signal, 1) //信号传递的channel
    done := make(chan bool, 1) //用于通知应用退出的channel

    //通常在主go routine中注册.
    signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM) //注册需要处理的信号
集, 以及信号传递的channel.

    go func() { //在另外一个专门的go routine中监视信号的到来, 当然也可以在业务
处理代码中监视信号, 比如在业务处理代码的select中多加一个监视信号传递channel的
case.

        sig := <-sigs //阻塞在这等待信号到来
        fmt.Println()
        fmt.Println(sig)
        done <- true //关闭信号来了, 则发送退出应用消息
    }()

    fmt.Println("awaiting signal")
    <-done //一旦有值可读, 代表应用被通知退出
    fmt.Println("exiting")
}
```

也许大家认为，这信号处理多么简单呀，有必要拿出来说事吗？如果您在 unix/linux下编写过C/C++程序，您就会有些切身的体会，比如：在单线程时代，编写信号处理函数就非常麻烦，首先函数中只允许调用“可重入”函数，其次如果在信号处理函数中需要修改全局数据，那么被修改的变量必须是sig_atomic_t类型的，否则被信号打断的程序在恢复执行后，很可能不能立刻看到信号处理函数对全局变量所做的修改。到了多线程时代情况就更复杂了，信号分为两类：发送给某一特定线程的 (SIGSEGV)，发送给任意线程的信号，不同的线程可能还要设定不同的信号掩码，屏蔽不想处理的信号，留下想要处理的信号；并且在信号处理函数中不可以调用 Pthreads 中的函数，不能通过Condition variable来通知其他线程。麻烦吧，我只是转述一下，详细的讲解，我推荐《Linux 多线程服务器端编程 使用muduo C++ 网络库》电子工业出版社 陈硕著，作者功力深厚，绝佳的好书，希望大家多看看，国人写的好书。现在大家有了解了吧，是不是真的很麻烦，好在golang为我们屏蔽了这些复杂，让我们可以专心于业务领域的开发了。

以下是一些不错的参考资料：

<https://golang.org/pkg/os/signal/#Stop>

<http://colobu.com/2015/10/09/Linux-Signals/>

<http://tonybai.com/2012/09/21/signal-handling-in-go/>

<http://blog.didibird.com/2016/03/16/Graceful-Restart-in-Golang/>

<https://github.com/rcrowley/goagain/blob/master/goagain.go>

最后再总结一下golang信号处理的关键点：新建一个专门用于传递信号的channel，其它go routine可以读取这个channel，或是通过select监控它，从而得知信号的到来并处理之！

18.6. fork会怎样

以下是我找的一些资料，以供参看，`fork()`出现在单进程单线程时代，所以直接调用它没有问题，但是到了多线程时代，调用`fork()`会产生一个问题，首先说明一下，`fork()`复制了父进程，包括内存，变量，资源句柄，基本上原封不动copy了一份，并应用写时复制机制保证效率，即变量逻辑地址相同，其实此时物理地址也相同，当发生写操作时，操作系统处理冲突，才会为子进程分配新的物理内存，这没问题，但是调用`fork()`后，除了调用`fork()`的那个线程被保留外，其它线程一律消失了，也就是说新的子进程中就剩下一个线程，这对于某些程序也许没有问题，但是对于golang程序问题就太大了，因为golang runtime底层严重依赖多线程来支撑，如：协程调度运行，GC等许多运行时支撑任务都是在多线程中运行的，所以`fork()`后子进程成了光杆司令，则golang runtime无法再运行，则golang程序无法再运行！故最终只可以用`exec`来执行新的程序。

所以golang没有提供独立的`fork()`接口，只有`syscall.ForkExec`或`Exec`；原因很明白，`fork()`之后没法玩下去了，只能`exec`一个新程序来此进程中运行了。

`exec`包中提供了在新进程中运行`command`的接口，`os`中提供了一些底层的支持接口，至于`syscall`则是更底层的操作系统接口封装。下面<https://gobyexample..com>中的两个例子和说明很是清晰，可以参考引用。

之所以列出这个点，就是想提醒同学们，别想当然的`fork`，特别是早期unix/linux编程时的用法，此时未必可用了，别让传统思维习惯误导了，到时掉入坑中还不自

知！

早期之所以fork()比较有用，因为其主要应用模式为一主多从，主进程负责监听事件，并向从进程派发任务，如果从进程不存在，或是数量不够，则fork()之，但是此模式很重，创建，切换，调度代价都比较大；详细深入学习，我推荐《Linux 多线程服务器端编程 使用muduo C++ 网络库》电子工业出版社 陈硕著

参考资料：

You supposedly want `syscall.ForkExec()` from the [syscall](#) package.

Note that `fork()` has been invented at the time when no threads were used at all, and a process had always had just a single thread of execution in it, and hence forking it was safe. With Go, the situation is radically different as it heavily uses OS-level threads to power its goroutine scheduling.

Now, unadorned [`fork\(2\)`](#) on Linux will make the child process have just the single thread—the one which called `fork(2)` in the parent process—among all those which were active, including some crucial threads used by the Go runtime. Basically this means that you simply *cannot expect the child process to be able to continue executing Go code*, and the only thing you can sensibly do is to somehow immediately perform `exec(2)`. Notice that that's what `syscall.ForkExec()` is supposed to be used for.

And now think about the problem further. I'd say these days the only thing a direct call to `fork(2)` is useful for is "best-effort asynchronous process state snapshotting"—the kind, say, [Redis](#) uses. This technique relies on the fact the child process inherits all the memory data pages from its parent, but the OS uses copy-on-write technique to not really copy all that data, so the child can just sit there and save all the data structures to disk while its parent is chugging away modifying them in its own address space. Every other conceivable use for `fork()` implies immediate `exec()`, and that's what `exec.Command()` et al is for, so why just not use it?

<https://stackoverflow.com/questions/28370646/how-do-i-fork-a-go-process>

//-----

Package exec runs external commands. It wraps os.StartProcess to make it easier to remap stdin and stdout, connect I/O with pipes, and do other adjustments.

<https://golang.org/pkg/os/exec/#Cmd.Run> //在新进程中执行命令
<https://golang.org/pkg/os/#StartProcess> // 创建新进程，底层接口，高级接口在exec包中

```
//-----  
Exec  
ForkExec  
https://golang.org/pkg/syscall/ //封装操作系统的api，最底层接口  
//-----  
https://gobyexample.com/spawning-processes //在新进程中执行  
https://gobyexample.com/execing-processes //在当前调用进程中执行
```

18.7. 优雅退出和重启

退出也好，重启也罢，怎么做到从容优雅呢？通俗点说：你要我退出或是重启，请事先通知我一下，我好停下手里干的活，记一下log，入一下库，关闭一些东西，清理一些资源，最后没有烂尾地，清清白白地离开；比如：你按下Ctrl-C，则应用程序必须监控并获得这个信号，执行一系列善后的工作后，退出，或是重启；对于golang程序，我们只须注册并监控那个“信号传递channel”就好，简单吧！如果你是通过socket，或是pipe之类，也没关系，反正golang内部我们都是用channel通信的，我们只要创建专门的channel传递相应类别的消息就好！简洁吧！具体的例子代码不用写了吧，网上有好多，再说，原理弄明白了，完全就自己发挥吧！最后再啰嗦一下：网络程序的重启在原理与本文所讲没有不同，只是为了降低对客户端的影响，需要考虑一些比较琐碎细节的点，我会在后文golang网络编程中讲一讲的。

19. error处理

错误处理对于任何一门语言都是最重要的主题之一，C语言errno，C++的exception与try-catch机制。对于golang支持函数返回多值，故error与函数执行结果一起返回；在golang中不存在什么错误基类东西，因为它不支持继承，虽然可以通过组合实现之！但是类继承太死板了！，所以golang只是把错误定义为一个接口，如：

```
type error interface {
```

```
Error() string  
}
```

这真灵活了，只要你这个具体类型实现了这个接口，那golang编译器就认你为一个error类型，就可以返回，传递，判断之；所以只要你的类型可以并实现了这个接口，那你就是合法的error类型；真是方便高效简洁灵活呀！可以具体类型可以是int(只保存错误码)，也可以是string(保存一段错误描述)，通常为一个struct，在其中定义许多成员，用于分门别类记录更多的错误信息，如：

```
type PathError struct {  
    Op string  
    Path string  
    Err error  
}  
  
//具体类型PathError实现error接口  
func (e *PathError) Error() string {  
    return e.Op + " " + e.Path + ": " + e.Err.Error()  
}
```

//下面为golang标准库中的一个函数

```
func Stat(name string) (fi FileInfo, err error) { //go范：error参数放在返回参数列表最右边
```

```
var stat syscall.Stat_t  
  
err = syscall.Stat(name, &stat)  
  
if err != nil { //go范：通常我们只判断错误是否为空  
    return nil, &PathError{"stat", name, err}  
}  
  
return fileInfoFromStat(&stat, name), nil //go范：无错误时，我们就返回nil  
}  
  
//调用Stat并处理错误  
fi, err := os.Stat("a.txt")
```

```
if err != nil {
    if e, ok := err. (*os.PathError); ok && e.Err != nil {
        //使用golang的“类型查询”， 可以判断接口e实际指向的具体类型是否为
        *PathError， 这是golang的用法。
        //我在学校期间就喜欢C++, 下了点功夫， 毕业一入行也是C++， 用了多年， 所以还算了解， 但是觉得C++的错误处理为了与C兼容， 所以做得不彻底， 有点乱， 比如你用了一些开源库， 有的明确以exception与try-catch方式处理错误；有的明确以errno为错误处理方式；有的则乱一点兼而有之！我个人比较喜欢errno方式， 一个整数， 节省空间， 传递与判断效率都高！所以， 我可以在定义PathError时， 再增加一个成员ErrNo int就可以了！也不错呀！只不过需要再定义一个全局， 或包级的错误码常量表呀！
    }
}
```

//再来看来golang标准库中的错误处理方法
例如： golang/src/os/error.go 定义了一些具体error类型及常量

```
package os

import (
    "errors"
)

//这就是包级别的错误常量表，只不过不是错误码，而是一个个errorString通用错误类型对象指针，在golang/src/errors/errors.go中定义了一个通用的具体错误类型errorString。
//在代码中可以return os.ErrInvalid返回错误；
//在代码中可以 if err == os.ErrInvalid {}来判断具体的错误类型。
// Portable analogs of some common system call errors.
var (
    ErrInvalid      = errors.New("invalid argument") // methods on File will
    return this error when the receiver is nil
    ErrPermission   = errors.New("permission denied")
    ErrExist        = errors.New("file already exists")
    ErrNotExist     = errors.New("file does not exist")
    ErrClosed       = errors.New("file already closed")
)

//为此包定义一些更具体的错误类型
// PathError records an error and the operation and file path that caused it.
type PathError struct {
```

```

Op  string
Path string
Err  error
}

func (e *PathError) Error() string { return e.Op + " " + e.Path + ": " +
e.Err.Error() }

// SyscallError records an error from a specific system call.
type SyscallError struct {
    Syscall string
    Err     error
}

func (e *SyscallError) Error() string { return e.Syscall + ": " +
e.Err.Error() }
//... ...

```

//更强的开源error增强库

只要现实error接口就好， 这真的很灵活， 所以github上出现了一些不错的增强库， 我之前做工具用过一两个， 其大的原则：自定义更丰富的具体错误类型，用于保存更加丰富的错误信息； 提供方法和机制用于抓取收集出错处的有效的丰富的错误，执行环境， 上下文信息等等； 提供方法和机制提取分析错误信息，有助于更快更准确地判断错误，进而处理之； 错误信息与日志的对接等等。之前的项目做完就节了， 互联网吗？做得快扔得快， 库名也没记， 想不起来了， 您可以在github上找找， 好找的！最后说点经验之谈， 对于错误处理这块是大问题， 需要项目开始之初就定下来， 具体到用什么方法， 什么机制， 什么错误处理库， 以及日志等， 边写边弄就乱了， 因为是多人合作哟， 不定死， 就是各自为政， 最后代码如何对接呢！

20. panic defer recover

用过C++的同行应该比较熟悉“资源守卫”的东西， 其实就是一个普通的C++类，在其构造函数中申请或持有资源， 在其析构函数中释放清理资源； 再者C++栈对象皆遵循“作用域”的限制， 离开“作用域”则死， 自动被析构！ 利用这两个C++特性， 尽力保证资源被及时释放， 为什么不是100%呢， 因为析构不是100%会被执行到， 比如代码中调用了特别的函数， 如：`abort`、`exit`； 深入学习， 我推荐《Linux多线程服务器端编程》 陈硕著 电子工业出版社； 当然网上关于此主题的文章很多， 一搜便知！

来个golang的例子直接说明吧！

```
func CopyFile(dst, src string) (w int64, err error) {
```

```

srcFile, err := os.Open(src)
if err != nil {
    return
}
defer srcFile.Close()

dstFile, err := os.Create(dst)
if err != nil {
    return
}
defer dstFile.Close()

//defer为golang关键字， 以上两行红色defer代码， 向golang系统注册需要在
//当前函数执行完时， 自动执行注册的清理函数， 方法， 或匿名函数； 注册清理函数被执行
//的次序为：“先进后出”， 意思是说： 最先注册的最后执行！ 就是数据结构中的栈！

//以下为匿名函数， 对于defer而言毫无问题！
defer func() {
    //做一些清理工作
}

return io.Copy(dstFile, srcFile)
}

```

//注意哟： 在defer的延迟函数中， 可以修改其外围函数的“命名返回参数”哟，
下面的例子可以看看哟！

20.1. defer一定被执行吗？(return exit abort panic)

golang保证， 即使发生panic， 所有的defer延迟调用函数也一定会按次序执行的！
如果在当前函数退出时， 执行了： os.Exit(1) 则程序立即终止， 所有defer 延迟调用
函数不再调用！！！

详情请看：<https://golang.org/pkg/os/#Exit>

如果您需要向操作系统返回status code， 必须调用os.Exit(status code)时， 请
将您自己的业务代码都统一封装到一个函数中， 并加入defer+panic保护， 如：

```

func main() {
    ret := DoWork() //这样在DoWork执行完毕退出时， 其所有的defer一定会被执行完， panic会被拦截处理完！最后返回一个代表程序执行结果状态的code就好了！
    os.Exit(ret) //除了在这，其它地方不允许调用os.Exit了！最好程序单入口，单出口！
}

```

20.2. panic defer recover的正确用法

(a) panic 内置函数

`func panic(interface{})`，有了这个内置函数，除了golang runtime抛出panic外，我们也可以自己抛出panic，如：`panic(404)`、`panic("network broken")`、`panic(Error("file not exists"))`等等；因为`interface{}`在golang中相当于c/c++的void，所以可以存储任意类型的值！故此在抛出panic时，可以一并带走丰富的信息！如果panic没有recover将其捕获，则会沿调用栈向上传递，直到被recover捕获，否则进程崩溃。

(b) recover 内置函数

- (1) recover可以直接在代码中调用，不论是否发生panic，没有副作用
- (2) recover只有在defer 的延迟调用函数中调用，才可以正常工作，捕获panic，切记哟

如：

```

defer func() {
    if r := recover(); r != nil {
        fmt.Println(r)
    }
}()

```

(3) recover一定要在可以发生panic的代码之前定义

(3) 通常用法1

```

go func() { //匿名函数， 函数， 方法皆可
    //加入panic捕获保护代码， 防止此go routine的panic外传。
    defer func() {
        if r := recover(); r != nil {
            fmt.Println(r)
        }
    }()
}

```

```
    }()
```

```
//可能发生panic的代码  
//.....
```

```
    }()
```

(4) 通常用法2

此例摘自我之前写的CSDN博客

```
package main
```

```
import (  
    "fmt"  
    "errors"  
)
```

```
func testPanic2Error() (err error) {
```

```
    //捕获 and 恢复 panic.
```

/*注意defer延迟执行的函数可以修改外围函数“testPanic2Error”的命名返回值。

*通过调用recover捕获panic并转化为error。也许有人打算在main函数中放一个下面这个defer语句，用于捕获程序中的一切panic异常，

*建立最后一道防火墙，从而使程序避免崩溃运行下去，但很不幸，当main函数的defer延迟函数被执行时，也就意味着main函数要退出了，

*此时再捕获panic恢复程序，意义还有多大呢；不过我们有办法克服，建立一个像“testPanic2Error”这样的一个外围封装函数，在这个函数中，

*建立最后一道防火墙，就像此处例子代码中所做的一样，将panic封闭在自己的包内，不允许蔓延传染给其它包，包与包之间只通过error传递

*结果状态。

```
*/
```

//第一个注册引defer + recover延迟调用函数，这样它就不仅仅可以捕获当前函数中发生的panic，就连在其后注册的defer延迟调用函数中发生的panic也一并可以捕获处理！道理很简单，第一个注册则最后一个执行！所以在其执行之前发生的panic就都可以捕获！

```
    defer func() {
```

```
        if r := recover(); r != nil {
```

```
            fmt.Println("Recovered in testPanic2Error", r)
```

```
            //check exactly what the panic was and create error.  
            switch x := r.(type) {
```

```

        case string:
            err = errors.New(x)
        case error:
            err = x
        default:
            err = errors.New("Unknown panic")
    }
}

//logic code , panic here.
//panic("i am string")
//panic(errors.New("i am error"))
panic(-1)

//从此开始一直到右花括号中的代码不会再被执行了！因为panic会立刻终断当前函数执行流程， 执行defer延迟调用， 最后返回到调用者处，切记！

fmt.Println("never executed")
return nil
}
func main() {
    fmt.Println(testPanic2Error())
    fmt.Println("panic restore now, continue.")
}

//注意： 在某一个defer延迟函数中发生panic， 不会影响其它defer延迟函数的执行！

```

20. 3. panic 不要蔓延到包外（对外只报error）

*golang*用*error*应对可以预见可能出现的错误与异常， 用*panic*应对预见不到， 或预见到不太可能发生， 或不应该发生的错误与异常；出现*panic*大家就别玩了， 重启洗牌重来！对于*panic*其不可以不允许扩散到包外！这是*golang*的铁律， 我们自己定义包时必须遵守！通常是在包内拦截下*panic*， 处理后， 转化为*error*具体错误类型对象返回给包外调用者！调用者只需查询*error*就好， 不可以因为你包内的*panic*发生， 导致包外的调用者， 使用者也*panic*了！这样大家就都别玩了！这很好理解吧！你可以上报*error*说：这事我做错误了， 我做不了； 但你不可以让调用者， 使用者和你一起完蛋哟！

21. GC

内存管理分类： 手动管理， 引用计数， *GC*； *golang* *GC*现在虽然是并发的了， *STW*时间也非常短了！但是*STW*还是有的！希望未来可以没有*STW*问题， 因为高并发与实时系统最怕*STW*了！其实引用计数也不错， 苹果的

*swift*就采用的引用计数技术呀， 简洁且没有STW的问题；引用计数把时间消耗分散到各个语句中了， 而GC则集中消耗时间在自身！到底哪个更有效率，不太好分！但是*golang* GC自身降低消耗时间，去除STW，保持最低的空间消耗！这三点还需不断努力突破哟！

我们在做项目时，要尽可能降低GC的负担，(1) 控制对象数量，越少越好；(2) 避免频繁申请和释放对象；(3) 简化对象引用关系，不是说所有情况下都用指针，大家指来指去，就像蜘蛛网，有时也需要复制来避免过度引用。所以对象复用是个好办法，如：对象池；github上有好多，自己也可以写一个！

Go 1.8: Argument Liveness

详细的解释请看：<http://www.jianshu.com/p/c792f3eb53eb>

之所以强调一下这个GC新特性，因为它有关系到了对象的生死，这非常重要，程序员必需明确了解对象的生死，不然引用死的对象则panic，或是太长的对象生存期，则浪费内存。

21.1. 并发泄漏

现在服务器端基本都是高并发，或是异步，或是`epoll`，但是有一点不变，那就是资源有限！来一个网络请求建一个线程，或是协程，若是这些协程执行很慢，或是干脆阻塞不动！面对源源不断的请求进来，则主机资源将很快耗尽！这就是并发泄漏！当然内存泄漏也可能隐藏其中！怎么办？抛开计算机，从现实世界找解决方法！这是一个架构师的根本立足点！交通限速，分流；大河大江避免洪灾，建闸门，建分支，建水库，目标就是限速分流；北京的地铁也是如此！所以这是很自然的道理！当然还需要排队，对于不重要的请求还可以丢弃！如果量还是很大，则放弃次要任务，全力保证主要任务！实在不成了，用户可以慢一点等一下，但我的系统必须保持服务。所以在做项目时，还是要有一些ppt，uml来关注一下这些点！上线后堵漏达不到太好的效果！辛苦的程序员别急得编码哟！ppt，uml更重要，再加上几次团队一起的头脑风暴！方案成熟一点，程序员少加班！

22. 调试追踪

- (a) 读代码，在可能出错的点加Print.
- (b) 通常我们用log在可能出错的点记一条日志，如：传入的值，结果值，逻辑判断值等。

- (c) 如果程序在本机运行，则用调试器也可以，如：gdb, delve等。
- (d) 可以给go build、go run、go test加上-race标志，用于检查并发问题。
- (e) 在linux shell下，按下组合键：Ctrl + \ 终止程序并产生core dump，core dump记录了stack。
- (f) linux 工具和命令：netstat, ps, nmap, strace, lsof, tcpdump, ping, ip, traceroute, nslookup, iftop, iperf, ldd 等等，有助于分析问题。
- (g) 在linux 的一个虚拟目录 /proc 中，全是操作系统以及当前程序运行时的动态信息，如：内存等。
- (h) go tool pprof 是全面追踪分析golang程序的一把瑞士军刀。
- (i) go vet, golint 可以对代码做很好的静态分析检查，有效避免编码错误。

其实说真的，现实项目中就是(b) 用得最多，因为都是分布式程序，出错通常都是读日志，线上留给你的都是事故后现场，可能程序已然挂掉了，只剩下日志了！即使是在预发布环境也是如此！所以实际项目中，日志必须加好，所用的日志库保证不丢日志，而且还要兼顾效率，不可以拖累业务程序！说多了，哈哈，关于调试器，我推荐delve 和 gdb，它们的用法网上有很多资料，我就不再凑字了！至于写完的程序，在解决完编译错误后，还要用单元测试检测每一个函数的正解性；系统测试和压力测试主要由测试部同学来做，需要编写大量测试case验证正确性，对于高并发网络程序压力测试是必须要做的！这些测试任务，可以人力做，也有一些自动化测试工具可用。

上面主要说了一下调试手段和工具，关于调试思路，我的理解是：输入，计算，输出；对于并发，还要考虑：加锁，同步，数据一致等，这些点是我们入手的关键点！

我写这篇文章并不想谈什么高深的理论，很多只是提醒和忠告。

23. 1. *internal package*

23. 2. *go get* 研究 (*trunk* , *branch with the tag==go1*)

24. 测试

golang在语言和工具方面对测试都有很好的设计，支持；很难想像没有测试，代码将会怎样？！测试这一块是个非常大的主题，最近鄙人琐事太多，精力严重不够，不打算写了，关于测试的例子，网上有很多代码和文章，参学一下就好。

25. 日志

对于布布式程序来说，日志是我们最根本，有时甚至是唯一的救命稻草，所以在ppt阶段，或是架构设计阶段就要关注日志的设计，写日志，收集日志，分析日志等，也许是我太笨吧，不管用了听起来多牛的技术，一旦出现线上问题，程序员还是要加班读日志，所以日志极为重要！！！

最好细化到分析众多日志库的优缺点，不要小看哟，有些点是架构关注点，比如：不可以丢日志，速度要快等！对于日志的收集和分析的技术和开源系统也很多！也要分析其优缺点，是否满足项目的架构关注点！这些工作前期必需做好，不要到了编码时才随意发挥，我经常看到一些项目的代码中，光日志库就用了好多个！有些只是程序员出于个人的好恶就用了！十分的混乱！另一个值得注意的问题就是日志文件的问题，有的项目程序就只输出日志到一个日志文件中，写满就rotate；有的项目分门别类创建多个日志文件名，不同的日志输出到各自的日志文件中；哪一个好呢？我个人觉得，如果我们在读出日志分析问题时，调用顺序，或是执行顺序等上下文信息对我们分析日志非常必要时，则最好这些日志就输出到同一个日志文件就好！别给程序员挖坑，他们够累了！

具体的技术细节我不想谈了，太细，再说我个人也不是这方面的技术能手。

26. cgo & c

27. go语言哲学浅析

以共享内存通讯，以通讯来共享内存

轻量协程

屏蔽堆栈

去除类继承，以组合可实现之

非侵入式接口设计，灵活，简洁

变量默认初始化为其零值

提供受限的指针

彻底的值语义，比如C的数组在结构体中与函数形参之不同

函数多返回值，方便

函数与方法为一等公民，可以自由灵活传递保存，调用

简洁清晰的error和panic机制

高效简洁的工程代码组织，引用，管理

简约而不简单，少即是多！

28. 网络与操作系统异常情况处理（网络断开，对端失效，信号中断发生时，**操作系统是自动恢复操作，还是放弃并报错？如何时检测对端失效，淘汰无效连接及长时间无消息的连接等）**

- (1) 读写超时
- (2) 对端失效
- (3) 操作系统信号中断

28.2. 网络程序优雅重启

<https://grisha.org/blog/2014/06/03/graceful-restart-in-golang/>

<https://github.com/fvbock/endless>

<http://blog.didibird.com/2016/03/16/Graceful-Restart-in-Golang/>

29. C1000k

这块我不是能手，只懂皮毛，只是写网络程序，高并发，高吞吐时又不得不提它！因为非常重要，我大概说一下我的了解，不准确，说来解闷而已。这是一个逐步演化的过程：任务系统，批处理系统，多进程单线程，多进程多线程，协程，10多路复用和异步执行；至此这些优化演化都是在操作系统这个盒子内折腾！特别是对于网络程序而言，网络包都要由操作系统内核插手，帮我们的程序收包发包调度进程线程运行等诸多任务，一旦海量网络请求涌来，首先操作系统的负载就会非常繁重，甚至垮掉！所以现在操作系统成了瓶颈所在！所以聪明人就想，程序不再受操作系统调度控制，这不太可行！但是网络数据包可不可以不让操作系统插手，由应用程序自己搞定！这样操作系统就轻松了许多哟！汹涌的网络请求不再直接冲击操作系统，而是应用程序，如果这个应用挂了，也不会干扰到其它应用的运行！最重要的是速度和效率会被极大地提升！故此才可以让并发和吞吐提升到一个崭新的高度！我理解的还比较肤浅，甚至是谬误！只当做破砖引玉罢了！您切莫见笑！具体的技术细节资料，网上有好多，我就不再班门弄斧了！因为项目的架构关注点中很有可能有它这一点，所以才列在此。

31. 分布式系统愚见

cap base

分布式系统不可靠属性 c1000k高并发系统要点 数据一致性 系统可靠性， 可用性， 可扩展性

合久必分： 分而治之 应对规模问题

分久必合： 加抽象层 应对零碎问题

狡兔三窟， 不要把鸡蛋都放到一个篮子

人多力量大， 可人多也乱

可靠有了， 可数据乱了， 谁来保证数据一致， 这是个问题！ 强制写多份都成功， 还是什么版本加协商投票机制！

最需要的就在我手边， 缓存原理， 本地缓存， 分布式缓存， 以小博大， 常量时间获取！ 适合做缓存的数据结构（内存快速加载和获取和硬盘持久化）

索引力量大（以小博大， 大而化小）

职责专一

动静分离

要摸批量， 要摸实时， 各有侧重

无状态， 有状态大不相同

主动防御， 事后补偿对账， 失败， 重试， 放弃， 日志很重要！ api幂等吗？ 若非幂等， 如何防止重复请求？

少数服从多数吗？

快速失败或倔强不死

协商或专权

无序或有序（绝对时间有序或相对时间有序）

有序只是解决数据一致性问题的一半！

最优情况： P（网络分区） 不会出现！ 那么多个节点经过协商同步最终持有最新一致数据！

万事大吉了吗？ no， 数据仍然存在不一致窗口时间！ 比如各节点协商同步时间！

如果写操作在并发进行着！且无锁！那数据合适可以收敛为一致！不一致时间窗口会持续滑动！

看来又是民主需要集中，专权有时是必须的！

以上讨论只是在最理想情况下的讨论！如果出现网络分区，或节点不可靠！那么数据如何达成一致！所以对于银行业务，数据绝不可以乱，必须一致！如何权衡？有时专权和锁真是好东西！有时又是绊脚石！快速失败。倔强不到，本地记账事后对账纠错！事务的力量

分布式锁可靠吗？靠网络心跳和租约维持的分布式锁可靠吗？一个节点抢到了锁，然后处理此业务请求，如果网络异常，锁失效！及时监测锁失效，放弃处理此业务请求吗？还是继续默默处理，只是通过其他方式互斥？

消息队列适合什么场景

etcd适合什么场景的？

分流，限流，限速，降级

raft paxos算法

32. 分布式系统之别想当然

33. 分布式一致性协商算法 (*raft, paxos*)

有很多高手写了非常精确，有深度的讲解文章，我不是这方面的高手，只是了解一些，也很可能是谬误的，说来大家乐呵一下罢了！

分布式的初衷就是人多力量大，也可以叫分而治之！目标就是可以处理远远超过单机处理能力的海量请求！思想很朴素！效果也很明显！但是现实告诉我们，人多了思想就可能不统一，步调就可能不一致！系统虽然是分布式的，但是我们希望这个分布式系统对外来说，它就是一个整体，而不是一个个分体，因为请求者不关心这个，它只在乎请求可以被快速且正确地响应！矛盾出现了，因为分布式系统的不统一，不一致，不可靠是一定会存在的！这是必然！如何解决它，现实社会就有现成的方案，我们需要抽象出来：

- (1) 大家一起商量着来，这是民主。
- (2) 民主通常是低效的，有时甚至是无效的，因为大家商量，即达不成一致。
- (3) 所以我们需要唯一的统帅来集中专权领导我们，他是最高的权威，使大家达成最终一致。
- (4) 现实很残酷，这个大家可能分裂，最终导致选出各自的统帅。
- (5) 天呀，怎么办呢，是少数服从多数，还是老的统帅服从新的统帅。

(6) 分裂不会长久，和平统一才是常态，所以有人需要放弃，比如某一个统帅放弃身份，改为服从新统帅。

(7) 这个放弃与选择需要规则，大家一起遵循之。

(8) 每个人都有可能成为统帅，也可能又变为平民，这就是实现，但不管怎样，我们需要有自己的统帅并与统帅保持一致。

(9) 这唯一的目的是，就是对外始终保持一个一致的透时整体。

(10) 以悲观视角制定规则是比较复杂的，因为每个人都可能是个骗子；若以乐观视角看世界，相信绝大多数人是诚实守信的，则制定的规则就简单一些！但是两种规则不会是万无一失的万灵药，可能只适用绝大多数情况！极少数情况下还是会出错。

(11) 根本矛盾在于网络请求的全局无序性，每个节点都有精确计时，但是这个计时不是全局唯一有序有效的，各个节点计时有快，有慢，有误差，甚至错误！

(12) 如果可以给每一个网络请求都打上一个“全局唯一有序有效的时间戳”，那判定一致性的规则就简单了，因为总要有个先来后到吧！*google spanner*实现了类似方案，好像是基于，原子钟，GPS，加点盐（时间间隔随机值），目标就是生成一个全局唯一有序有效的时间戳。

理论是终极完备的吗？*paxos/raft*是终极完备的吗？这是终极真理，还是最大近似，我不是专家，我无法认证！我只是提醒，在做架构设计时，多给自己留条后路，比如由于某个极端苛刻的原因，大家就是无法达成一致，整体系统无法处理此请求，甚至系统瘫痪时，我们有没有设计可以外部直接干预使之恢复正常运行的方案！比如保证数据虽然不一致，但是新老数据都被保存下来，可以人工纠正使之一致！亦或是无法纠正，人工放弃等架构机制设计，当然我们希望有一个自动化系统伺服在那，比如仲裁，事后查账纠正等等措施！或是干脆串行排队，再加个单机强事务保护，比如银行业务。这也可以接受，因为有时是速度第一，有时需要安全第一。没有万无一失！程序员最大的问题之一，就是盲目崇拜某一个自己掌握的技术和经验。

34. 较真（同步，异步，阻塞，非阻塞，系统陷入，中断及恢复，惊群现象）

(1) 进程线程惊群问题???

举一个很简单的例子，当你往一群鸽子中间扔一块食物，虽然最终只有一个鸽子抢到食物，但所有鸽子都会被惊动来争夺，没有抢到食物的鸽子只好回去继续睡觉，等待下一块食物到来。这样，每扔一块食物，都会惊动所有的鸽子，即为惊群。对于操作系

统来说，多个进程/线程在等待同一资源时，也会产生类似的效果，其结果就是每当资源可用，所有的进程/线程都来竞争资源，造成的后果：

- 1) 系统对用户进程/线程频繁的做无效的调度、上下文切换，系统性能大大折扣。
- 2) 为了确保只有一个线程得到资源，用户必须对资源操作进行加锁保护，进一步加大了系统开销。

最常见的例子就是对于socket描述符的accept操作，当多个用户进程/线程监听在同一个端口上时，由于实际只可能accept一次，因此就会产生惊群现象，当然前面已经说过了，这个问题是一个古老的问题，新的操作系统内核已经解决了这一问题。

<http://www.cnblogs.com/lchb/articles/3612707.html>

(2) epoll惊群问题???

新连接过来时，多个子进程都会在epoll_wait后被唤醒！

参考资料：

http://blog.csdn.net/russell_tao/article/details/7204260

<http://www.cnblogs.com/Anker/p/7071849.html>

<https://jin-yang.github.io/post/linux-details-of-thundering-herd.html>

<http://blog.csdn.net/liujiyong7/article/details/43346829>

35. 参考资料

- (1) Linux 多线程服务器端编程 使用muduo C++ 网络库 电子工业出版社
陈硕著
- (2) Go语言学习笔记 电子工业出版社，中国工信出版集团 雨痕著
- (3) Go程序设计语言 机械工业出版社 艾伦 A. A. 多诺万 布莱恩 W. 柯尼汉著
- (4) Go语言编程 人民邮电出版社 许世伟 吕桂华等著
- (5) <http://tonybai.com/>
- (6) <https://github.com/golang>
- (7) <https://golang.org/doc/>
- (8) <https://gobyexample.com/>

- (9) <http://studygolang.com/>
- (10) http://blog.csdn.net/htyu_0203_39 , 我之前写了几篇blog.
- (11) <http://colobu.com/2015/09/07/gotchas-and-common-mistakes-in-go-golang/>
- (12) <http://www.cnblogs.com/lchb/articles/3612707.html>
- (13) <https://stackoverflow.com>
- (14) C++编程思想 (美)Bruce Eckel Chuck Allison著 机械工业出版社
- (15) 深入理解计算机系统(修订版) [美] Randal E. Bryant著 雷迎春译 中国电力出版社
- (16) *The Linux Programming Interface Michael Kerrisk*
- (17) <https://ttboj.wordpress.com/2016/02/15/debugging-golang-programs/>