

## 问题收集

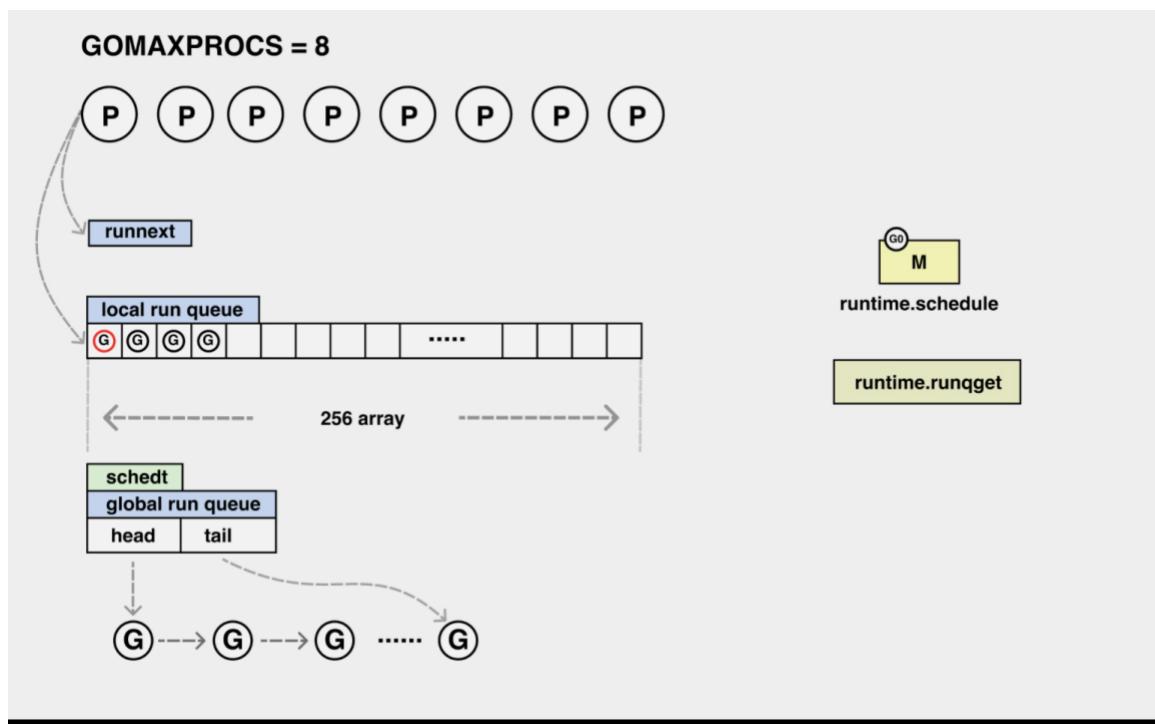
<https://shimo.im/sheets/6RPxvcTyrxTYVwWT/9IlkS>

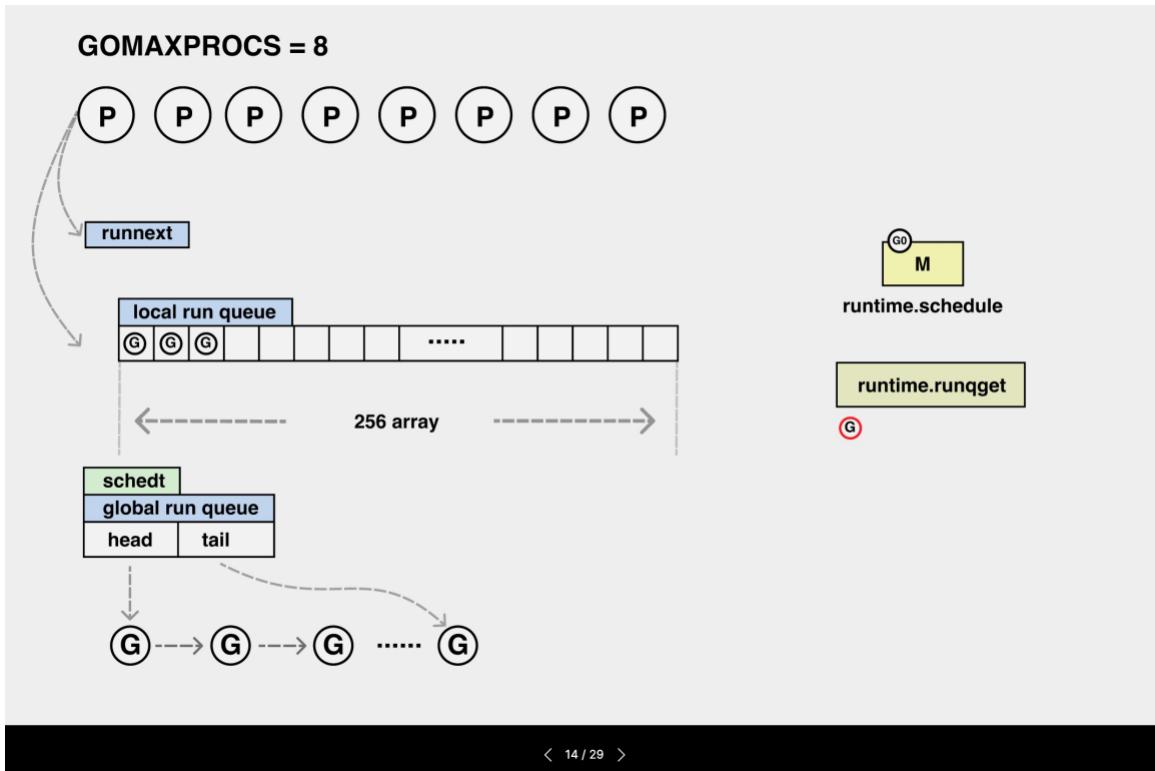
## Go 程序是怎么跑起来的

<https://class.imooc.com/lesson/62#mid=46353>

1. 本地队列是一个数组，具体是普通的数组还是一个环？(动画演示像环)

## 问题





## 回答

这里挺有意思的，结构上是一个 256 的数组，但其实用 head 和 tail 让他变成了一个环形数组，所以是个环形数组，我看看之后在什么地方把这个解释加上去。

local run queue 就是一个无锁队列，生产和消费过程都不需要加锁。  
而是用类似：

```
Go
for {
    atomic.Cas
}
```

这样的操作来对队列做操作，而且他就是个环~

## 2. 关于 getg()如何查找对应的实现代码

### 问题

# 一手资源+V 307570512

```
// getg returns the pointer to the current g.
// The compiler rewrites calls to this function into instructions
// that fetch the g directly (from TLS or from the dedicated register).
func getg() *g

// mcall switches from the g to the g0 stack and invokes fn(g),
// where g is the goroutine that made the call.
// mcall saves g's current PC/SP in g->sched so that it can be restored later
```

看到这函数声明，全局搜索没有找到对应的实现。看注释应该是编译器重写了，这种情况如何找到对应的实现代码呢？

```
> runtime.newproc() /usr/local/go/src/runtime/proc.go:3974 (hits goroutine(1):1 total:2) (PC: 0x1043c60)
Warning: debugging optimized function
3969: // This must be nosplit because this stack layout means there are
3970: // untyped arguments in newproc's argument frame. Stack copies won't
3971: // be able to adjust them and stack splits won't be able to copy them.
3972: //
3973: //go:nosplit
=>3974: func newproc(siz int32, fn *funcval) {
    3975:     argp := add(unsafe.Pointer(&fn), sys.PtrSize)
    3976:     gp := getg()
    3977:     pc := getcallerpc()
    3978:     systemstack(func() {
    3979:         newg := newproc1(fn, argp, siz, gp, pc)
(dlv) goroutines
* Goroutine 1 - User: /usr/local/go/src/runtime/proc.go:3974 runtime.newproc (0x1043c60) (thread 8570303)
[1 goroutines]
(dlv) break runtime.getg()
Command failed: Location "runtime.getg()" not found
(dlv) ■
```

```
Warning: debugging optimized function
3971: // be able to adjust them and stack splits won't be able to copy them.
3972: //
3973: //go:nosplit
3974: func newproc(siz int32, fn *funcval) {
    3975:     argp := add(unsafe.Pointer(&fn), sys.PtrSize)
=>3976:     gp := getg()
    3977:     pc := getcallerpc()
    3978:     systemstack(func() {
    3979:         newg := newproc1(fn, argp, siz, gp, pc)
    3980:
    3981:         _p_ := getg().m.p.ptr()
(dlv) break /usr/local/go/src/runtime/stubs.go:18
Command failed: could not find statement at /usr/local/go/src/runtime/stubs.go:18, please use a line with a statement
(dlv) ■
```

试了下 dlv，打断点，但是好像不能这么这样。

## 回答

这个实现比较 hack

代码在

# 一手资源+V 307570512

Plain Text

```
// Copyright 2014 The Go Authors. All rights reserved.  
// Use of this source code is governed by a BSD-style  
// license that can be found in the LICENSE file.  
  
#ifdef GOARCH_arm  
#define LR R14  
#endif  
  
#ifdef GOARCH_amd64  
#define get_tls(r) MOVQ TLS, r  
#define g(r) 0(r)(TLS*1)  
#endif  
  
#ifdef GOARCH_386  
#define get_tls(r) MOVL TLS, r  
#define g(r) 0(r)(TLS*1)  
#endif
```

实践上编译的时候还会再根据你的平台(amd64, arm)替换一次:

Plain Text

```
case ssa.OpAMD64LoweredGetG:  
    r := v.Reg()  
    // See the comments in cmd/internal/obj/x86/obj6.go  
    // near CanUse1InsnTLS for a detailed explanation of these instructions.  
    if x86.CanUse1InsnTLS(gc.Ctxt) {  
        // MOVQ (TLS), r  
        p := s.Prog(x86.AMOVQ)  
        p.From.Type = obj.TYPE_MEM  
        p.From.Reg = x86.REG_TLS  
        p.To.Type = obj.TYPE_REG  
        p.To.Reg = r  
    } else {  
        // MOVQ TLS, r  
        // MOVQ (r)(TLS*1), r  
        p := s.Prog(x86.AMOVQ)  
        p.From.Type = obj.TYPE_REG  
        p.From.Reg = x86.REG_TLS  
        p.To.Type = obj.TYPE_REG  
        p.To.Reg = r  
        q := s.Prog(x86.AMOVQ)  
        q.From.Type = obj.TYPE_MEM  
        q.From.Reg = r  
        q.From.Index = x86.REG_TLS  
        q.From.Scale = 1  
        q.To.Type = obj.TYPE_REG  
        q.To.Reg = r  
    }
```

最终的指令要通过反汇编来看(在 dlv 中 disass, 或者 go tool objdump)

# 一手资源+V 307570512

我个人理解这部分知道是怎么回事就行了。。。和编译器本身耦合的太重了

## 3. 处理阻塞那块有点疑问？

### 问题

调度那块 涉及到 syscall 的 goroutine 被 sysmon 线程接管

我现在两个问题

- 1. 非 syscall 的打包成 sudog 或 G 挂到了对应的结构的等待队列?是不是意味着队列中 goroutine 一旦被取走, 就不可能在放回去了. 还是说时间片到了, 会放回去等下一轮调度的?
- 2. 还是没太理解为什么有些被打包为了 sudog 有些打包为了 g.

### 回答

- 1. 非 syscall 的打包成 sudog 或 G 挂到了对应的结构的等待队列?是不是意味着队列中 goroutine 一旦被取走, 就不可能在放回去了. 还是说时间片到了, 会放回去等下一轮调度的?  
每一种进等待结构的操作都有对策的逆操作,  
lock -> unlock, chan send -> ch recv  
gopark 的 goroutine 找他们在哪里被 goready 就可以了
- 2. 还是没太理解为什么有些被打包为了 sudog 有些打包为了 g.  
这个第一节课讲过, 有些 g 会被同时挂在多个结构上等待, 比如 select 的时候, 一个 goroutine 会挂在多个 channel 的等待队列中, 这种时候需要 sudog, 其它情况下 g 就够了

## 4. 1.14 之前的主动让出的调度为什么就使得 STW 的时间变长

### 问题

1.14 之前的基于主动让出的调度和 1.14 的抢占式调度的一个疑问: 1.14 之前的主动让出的调度为什么就使得 STW 的时间变长、为什么会延长栈扫描时间呢

( <https://go.googlesource.com/proposal/+/master/design/24543-non-cooperative-preemption.md> )

### 回答

老版本 scanstack 需要把所有 goroutine 停下来, 循环次数比较多的 goroutine 内部没有函数调用, 那 scanstack 必须等待它主动停下来, 这段时间是在 stw 里的  
如果是个死循环, 那 stw 就会无限持续下去

# 一手资源+V 307570512

## 5. newproc1 三个 g 分别代表什么意思？有何作用？

### 问题

```
3448 func newproc1(fn *funcval, argp unsafe.Pointer, narg int32, callergp *g, callerpc uintptr) {
3449     g := getg() // 1
3450     if fn == nil { ... }
3451     acquirem() // disable preemption because it can be holding p in a local var
3452     siz := narg
3453     siz = (siz + 7) &^ 7
3454     ...
3455     if siz >= _StackMin-4*sys.RegSize-sys.RegSize { ... }
3456     ...
3457     _p := _B_.m.p.ptr() // 2
3458     newg := gfget(_p)
3459     if newg == nil { ... }
3460     if newg.stack.hi == 0 { ... }
3461     ...
3462     if readgstatus(newg) != _Gdead { ... }
3463     ...
3464     totalSize := 4*sys.RegSize + uintptr(siz) + sys.MinFrameSize // extra space in case of reads slightly beyond frame
3465     totalSize += -totalSize & (sys.SpAlign - 1) // align to spAlign
3466     sp := newg.stack.hi - totalSize
3467     spArg := sp
3468     if usesLR { ... }
3469     if narg > 0 { ... }
3470     ...
3471     memclrNoHeapPointers(unsafe.Pointer(&newg.sched), unsafe.Sizeof(newg.sched))
3472     newg.sched.sp = sp
3473     newg.stktopsp = sp
3474     newg.sched.pc = funcPC(goexit) + sys.PCQuantum // +PCQuantum so that previous instruction is in same function
3475     newg.sched.g = uintptr(unsafe.Pointer(newg))
3476     gostartcallfn(&newg.sched, fn)
3477     newg.gopc = callerpc
3478     newg.ancestors = saveAncestors(callergp)
3479     newg.startpc = fn.fn
3480     if _g_.m.curg != nil { ... }
3481     if isSystemRoutine(newg, fixed: false) { ... }
3482     casgstatus(newg, _Gdead, _Grunnable)
3483     ...
3484     if _p_.goidcache == _p_.goidcacheend { ... }
3485     newg.goid = int64(_p_.goidcache)
3486     _p_.goidcache++
3487     if raceenabled { ... }
3488     if trace.enabled { ... }
3489     runqput(_p_, newg, next: true)
3490     ...
3491     if atomic.Load(&sched.npidle) != 0 && atomic.Load(&sched.nmspinning) == 0 && mainStarted {
3492         wakcp()
3493     }
3494     ...
3495     releasem(&g.m)
```

Plain Text  
go func(){  
}()

这个调的是 newproc， newproc 调 newproc1 会传进去一个 g。但是 newproc1 里面有 3 个 g，分别是图里面的 1, 2, 3，最后调度的是 2，这是为什么呢？为什么不是就是在对 3 进行

# 一手资源+V 307570512

初始化操作？

4是不是全都是在初始化 g？

## 回答

newproc1 是从 newproc 进来的

```
Plain Text
func newproc(siz int32, fn *funcval) {
    argp := add(unsafe.Pointer(&fn), sys.PtrSize)
    gp := getg()
    pc := getcallerpc()
    systemstack(func() {
        newproc1(fn, argp, siz, gp, pc)
    })
}
```

要注意到这里有个 systemstack 的动作，这个是从用户 g 切换到 g0 栈来执行函数的，所以在 newproc1 中 getg 得到的 \_g\_ 是 g0，这个可以在 delve 里验证：

```
Plain Text
(dlv) 1
> runtime.newproc1() /usr/local/go/src/runtime/proc.go:3451 (PC: 0x10334da)
Warning: debugging optimized function
3446: // at argp. callerpc is the address of the go statement that created
3447: // this. The new g is put on the queue of g's waiting to run.
3448: func newproc1(fn *funcval, argp unsafe.Pointer, narg int32, callergp *g, callerpc
uintptr) {
3449:     _g_ := getg()
3450:
=>3451:     if fn == nil {
3452:         _g_.m.throwing = -1 // do not dump full stacks
3453:         throw("go of nil func value")
3454:     }
3455:     acquirem() // disable preemption because it can be holding p in a local var
3456:     siz := narg
(dlv) p _g_ == _g_.m.g0
true
```

callergp 指的是创建这个 goroutine 的原始的那个用户 g，

你图上的 2 才是 newproc1 真正创建出来的新 g，  
只不过 g 结构体本身有复用，要用 gfget 去找缓存

中间的代码确实是初始化代码。

## 6. 一些关于 noteXX 函数的疑问

## 问题

1. notewakeup、notesleep、noteclear 的内部原理是什么呢？
2. 以 notewakeup 为例，是怎么通过参数 note 中的 key 找到 m 的，继而唤醒他的？是指针偏移找的么？
3. notewakeup 底层的 semawakeup 是通过信号量 semaphore 来唤醒 m 的么？

## 回答

notewakeup 底层的 semawakeup 是通过信号量 semaphore 来唤醒 m 的么？  
这个 semawakeup 在 linux 上是没有的。

The screenshot shows a code editor's search results window. The search term 'func semawakeup' is entered in the search bar. The results list several definitions of the function across different Go files:

File	Line Number
os_darwin.go	69
lock_sema.go	24
os3_solaris.go	352
os_aix.go	88
os_netbsd.go	165
os_openbsd.go	175
os_plan9.go	444
os_windows.go	743

同时研究多个平台太累了，还是针对 linux 来看比较好~

## 7. 关于 M 的自旋和休眠

## 问题

①M 有几种状态：

- 自旋中(spinning): M 正在从运行队列获取 G, 这时候 M 会拥有一个 P;
- 执行 go 代码中: M 正在执行 go 代码, 这时候 M 会拥有一个 P;
- 执行原生代码中: M 正在执行原生代码或者阻塞的 syscall, 这时 M 并不拥有 P;
- 休眠中: M 发现无待运行的 G 时会进入休眠, 并添加到空闲 M 链表中, 这时 M 并不拥有 P。

# 一手资源+V 307570512

②第一点，工作线程 M 的自旋状态(spinning)。工作线程在从其它工作线程的本地运行队列中盗取 goroutine 时的状态称为自旋状态。从上面代码可以看到，当前 M 在去其它 p 的运行队列盗取 goroutine 之前把 spinning 标志设置成了 true，同时增加处于自旋状态的 M 的数量，而盗取结束之后则把 spinning 标志还原为 false，同时减少处于自旋状态的 M 的数量，从后面的分析我们可以看到，当有空闲 P 又有 goroutine 需要运行的时候，这个处于自旋状态的 M 的数量决定了是否需要唤醒或者创建新的工作线程。

曹大，①中对自旋状态描述是不是不对，②中好像更细一些，我看源码好像不止在 stealwork 的时候有置为自旋状态。

1.这个自旋状态应该怎么理解呢？

2.M 的 G0 执行 schedule，如果找到可用的 G，就切到 G 去继续走 execute，然后 exit 结束再到 G0 走下一轮 schedule。如果一直到 findrunnable 结束都没找到是不是就进入休眠，不会再走 schedule。这段我描述的对嘛？

## 回答

我觉得也是 ② 的精确一些

1.这个自旋状态应该怎么理解呢？

我觉得这个自旋就是暂时还没找到 Goroutine 来执行，但是正在找的一个状态。。有这个状态和 nmsping 的计数，能够帮 runtime 判断是不是需要再启动额外的线程来执行 goroutine

2.M 的 G0 执行 schedule，如果找到可用的 G，就切到 G 去继续走 execute，然后 exit 结束再到 G0 走下一轮 schedule。如果一直到 findrunnable 结束都没找到是不是就进入休眠，不会再走 schedule。这段我描述的对嘛？

就切到 G 去继续走 execute，这句不对，从 g0 切换到用户 g 是在汇编函数 gogo 里做的  
其它几句没啥问题

## 8. 为什么返回结构体指针就不会报错？

### 问题

```
Go
type Employee struct {
    ID      int
    Name    string
    Address string
    DoB     time.Time
    Position string
    Salary   int
    ManagerID int
}
```

```
func EmployeeByID(id int) Employee { //为什么返回指针 最后一行就不会报错?
```

# 一手资源+V 307570512

```
    return Employee{ }

}

func main() {
    var dilbert Employee

    id := dilbert.ID

    EmployeeByID(id).Salary = 0 // fired for... no real reason
}
```

## 回答

赋值语句左边的值需要是 addressable，

Each left-hand side operand must be **addressable**, a map index expression, or (for = assignments only) the **blank identifier**. Operands may be parenthesized.

具体 addressable 的定义：

[https://golang.org/ref/spec#Address\\_operators](https://golang.org/ref/spec#Address_operators)

这种就是个语法规定

9. 在 `runtime.mstart` 函数中，`_g_.m.mstartfn` 是在哪里赋值得？

## 问题

在 `runtime.mstart` 函数中，`g.m.mstartfn` 是在哪里赋值得？此时得 `m` 为 `m0`，在代码中未找到 `m.mstartfn` 得赋值、

```
Plain Text
if fn := _g_.m.mstartfn; fn != nil {
    fn()
}
```

## 回答

你是问 `m0` 的这个 `startfn` 是啥对吧，  
我调试了一下，本身也是没有人给它赋值的：

```
Plain Text
(dlv) p _g_.m.mstartfn
nil
```

# 一手资源+V 307570512

```
(dlv) bt
0 0x0000000000102df01 in runtime.mstart1
  at /usr/local/go/src/runtime/proc.go:1108
1 0x0000000000102de16 in runtime.mstart
  at /usr/local/go/src/runtime/proc.go:1087
2 0x00000000001050ebd in runtime.rt0_go
  at /usr/local/go/src/runtime/asm_amd64.s:225
(dlv) p _g_.m.mstartfn
nil
(dlv) l
> runtime.mstart1() /usr/local/go/src/runtime/proc.go:1108 (PC: 0x102df01)
Warning: debugging optimized function
 1103:     minit()
 1104:
 1105:     // Install signal handlers; after minit so that minit can
 1106:     // prepare the thread to be able to handle the signals.
 1107:     if _g_.m == &m0 {
=>1108:         mstartm0()
 1109:     }
 1110:
 1111:     if fn := _g_.m.mstartfn; fn != nil {
 1112:         fn()
 1113:     }
(dlv)
```

所以它就是 nil

## 再次追问

fn 其实并不是在执行 runtime.main 函数对吧？ runtime.main 需要在被 schedule 调度之后才能被执行？那怎么确保他一定会被第一个执行（因为目前为止应该信号 g 和 sysmon 都已经生成了）？

```
Plain Text
if fn := _g_.m.mstartfn; fn != nil {
    fn()
}

if _g_.m != &m0 {
    acquirep(_g_.m.nextp.ptr())
    _g_.m.nextp = 0
}
schedule()
```

## 回答

runtime.main 是生成了一个 goroutine 的，

# 一手资源+V 307570512

m0 最后进 schedule 的时候，只有那一个 goroutine

mstartfn 这个东西我看也只是一些 mspinning 之类的函数，和调度啥的没太大关系~

这个 fn 和 runtime.main 没啥关系

## 10. 一些关于调度的疑问

### 问题

1. 在程序启动 mstart 函数中，runtime.main 执行 main.main，如果 main.main 直接返回了，那么是不是不会进入到调度循环了？在 runtime.mstart1 中，main.main 执行完了之后才会进入到 schedule 循环中。
2. ppt 中说的 main.main 中有调度循环，是因为上面多了一个 defer 么？所以他根本走不到 main 结束，直接就去 schedule 了？
3. 如果 main.main 中有协程产生并且阻塞，我们会开一个新的 m 去运行产生的协程，那么这个新的 m 的调度就跟 m0 的调度不是一样的了吧？
4. 在 mstart1 中，如果 `_g_.m != &m`，会将 `_g_.m.nextp` 和 m 关联起来，为什么？
5. 为什么 mstart 中不是用 g0 和 m0 去执行 main.main，而是新建的一个 g 呢？

### 回答

1. 在程序启动 mstart 函数中，runtime.main 执行 main.main，如果 main.main 直接返回了，那么是不是不会进入到调度循环了？在 runtime.mstart1 中，main.main 执行完了之后才会进入到 schedule 循环中。
  - a. 你的意思是 main.main 执行很短一段时间就退出了吧，这种情况下，runtime.main 里的 f() 会返回，然后一路走到最后，整个进程就退出了
  - b. main.main 得碰到能接管的阻塞的时候，就是第一课 ppt 里那几种情况，m0 就会把这个 main goroutine 给 park 起来，然后重新进入调度循环。
  - c. main.main 如果执行完了，那后面直接调用 exit，整个进程就退出了，和调度循环啥的也没啥关系了
2. ppt 中说的 main.main 中有调度循环，是因为上面多了一个 defer 么？所以他根本走不到 main 结束，直接就去 schedule 了？
  - a. ppt 那里的意思是 m0 会负责执行 main.main，然后如果 main.main 有可接管的阻塞，后面 m0 是可以进调度循环的，进了 schedule 以后其实就和普通的线程没啥区别了~
  - b. 第一课直播讲的时候的图稍微有点问题~
3. 如果 main.main 中有协程产生并且阻塞，我们会开一个新的 m 去运行产生的协程，那么

# 一手资源+V 307570512

这个新的 m 的调度就跟 m0 的调度不是一样的了吧？

- a. main.main 本身就是在 main goroutine 里执行的，他阻塞的话，和其它 goroutine 阻塞没啥区别~
  - b. 如果是我们之前说的可接管的阻塞，那 m0 可以直接进调度循环去执行在 main 函数里创建的 goroutine~
  - c. 如果 main.main 的阻塞是 syscall 那种不可接管的，那 m0 就暂时被占用了，确实需要启动其它的 m 去执行调度
  - d. 进调度循环的线程都是一视同仁的，只不过 m0 只是初始线程，这个我印象中是操作系统帮我们创建的(就是其它语言里的主线程)，除 m0 以外的其它的线程都是在 Go 里面主动调用 syscall.Clone 创建的
4. 在 mstart1 中，如果 `_g_.m != &m`，会将 `_g_.m.nextp` 和 m 关联起来，为什么？
- a. 这里我看了看代码，就是启动新线程的时候，`newm -> startm` 是已经把这个 m 应该绑的 p 设置在 `m.nextp` 里了。。。到了 `mstart` 过程只是绑定一下，好像也没看出什么特殊的
5. 为什么 `mstart` 中不是用 `g0` 和 `m0` 去执行 `main.main`，而是新建的一个 `g` 呢？
- a. `g0` 在所有 `m` 里都是一个特殊的 `g`，只能用来执行 `runtime` 里的调度函数，在执行过程中不能影响到用户的运行栈，一般是 `system_stack` 或者 `mcall` 的时候会进去
  - b. 对于 `runtime` 来说，`main.main` 也是用户代码(`runtime.main` 也是)，所有用户代码都应该在专门的 `user g` 里执行

## 再次追问

for q4：为什么不直接将其绑定呢？而是要经过 `nextp` 中转一次呢？

for q5：在 `mstart` 中，进去得 `_g_` 就是 `g0`，何时将其切换为普通 `g` 得呢？在相应的代码里面没看到显示调用 `system_stack` 或者 `mcall` 呢。

q6：在 `runtime.main` 里面，最后

```
Plain Text
for {
    var x *int32
    *x = 0
}
```

这段代码有什么作用呢（我记得有人好像在微信问过。。尴尬）？

## 回答

for q4：为什么不直接将其绑定呢？而是要经过 `nextp` 中转一次呢？

这里我估计是老线程和新线程之间可能不太方便传这个 `p` 参数，所以只能通过 `m` 上的成员来转一下；

# 一手资源+V 307570512

for q5: 在 mstart 中，进去得\_g\_就是 g0，何时将其切换为普通 g 得呢？在相应的代码里面没看到显示调用 system\_stack 或者 mcall 呢。

在调度循环里 gogo，会切到用户 g 去执行

q6: 在 runtime.main 里面...

这种好像是老 unix 的传统，相当于 unreachable 的代码写个死循环能比较快的发现代码的 bug。

我记得很早的 linux 内核的 panic 就是这样的(个人意见)

fix : 这种好像是老 unix 的传统，相当于 unreachable 的代码 panic 能快速发现 bug

看代码的时候需要知道啥时候会切栈，  
要看 system\_stack 和 mcall，以及 gogo 之类的代码

这几个好像都是汇编实现的，不过注释写清楚了

## 再次追问

main 中不是只有

`runningPanicDefers != 0`

时，才能进入到调度循环么？我看这个

`runningPanicDefers`

的注释是说 当要退出时，另外一个协程发生了 panic，才会不为 0。那么这里 main 里面还有其他地方进去到调度循环么？

## 回答

执行调度循环的是线程，不是 goroutine..

## 再次追问

第五个问题是因为调度需要切换 g0 去进行调度循环么？

## 回答

对的，g0 主要就是用来执行调度循环的~

## [11. goland 调试有两个问题？](#)

## 问题

1. 为什么一下启了这么多个 g?

The screenshot shows the GoLand IDE interface during a debugging session. The top part displays the code for `test.go`:

```
1 package main
2
3 import (
4     "fmt"
5     "runtime"
6 )
7
8 func main() {
9
10    runtime.GOMAXPROCS(1)
11    arr := make([]int, 0)
12    fmt.Println(arr)
13 }
```

The line `arr := make([]int, 0)` is highlighted with a red breakpoint icon. The bottom part shows the debugger tool window:

- Frames:** A list of goroutines:
  - Goroutine 1 main.main (selected)
  - Goroutine 1 main.main
  - Goroutine 2 runtime.gopark
  - Goroutine 3 runtime.gopark
  - Goroutine 4 runtime.gopark
  - Goroutine 5 runtime.gopark
- Variables:** Shows the value `1373` for the variable `arr`.

A red arrow points to the value `1373` in the Variables list.

2. 如果我给 `runtime/proc.go` 打个断点, goland debug 后 hang 住不动, 不知道为什么?

# 一手资源+V 307570512

```
Project: test.go × proc.go ×
1 package main
2
3 import (
4     "fmt"
5     "runtime"
6 )
7
8 func main() {
9
10    runtime.GOMAXPROCS( n: 1 )
11    arr:=make([]int,0)
12    fmt.Println(arr)
13 }
```

main()

Debug: go build go-tutor1412/test/g04 ×

Debugger Console

```
GOROOT=/Users/mmaotai/go/go1.14.12 #gosetup
GOPATH=/Users/mmaotai/go #gosetup
/Users/mmaotai/go/go1.14.12/bin/go build -o /private/var/folders/6m/16z9zry920x1jcp0wzw4k8_w0000gn/T/_go_build_go_tutor1412_test_g04 -gcfla
/Applications/GoLand.app/Contents/plugins/go/lib/dlv/mac/dlv --listen=0.0.0.54905 --api-version=2 --check-go-version=false
s/6m/16z9zry920x1jcp0wzw4k8_w0000gn/T/_go_build_go_tutor1412_test_g04 --
API server listening at: [::]:54905
debugserver-@(#)PROGRAM:lldb PROJECT:lldb-1200.0.44
for x86_64.
Got a connection, launched process /private/var/folders/6m/16z9zry920x1jcp0wzw4k8_w0000gn/T/_go_build_go_tutor1412_test_g04 (pid = 50063).
```

## 回答

可以切换 goroutine 来看栈，我观察了一下除了 main 以外，其它几个分别是：

runtime.forcegchelper，每两分钟检查是不是已经进行过 gc，如果未进行过，主动触发  
runtime.bgsweep，后台清扫 goroutine  
runtime.bgscavenge，后台归还内存的 goroutine  
runtime.runfinq，这个应该是用来运行那些 finalizer 的 goroutine

这个 hang 住我也不太清楚。。。断点是打在什么地方了。。

## 再次追问

打在了这里。

# 一手资源+V 307570512

```
I > Users > mmaotai > go > go1.14.12 > src > runtime > proc.go
Project test.go proc.go stubs.go rt0_linux_amd64.s asm_amd64.s
os_windows_arm.go
panic.go
panic32.go
plugin.go
preempt.go
preempt_386.s
preempt_amd64.s
preempt_arm.s
preempt_arm64.s
preempt_mips64x.s
preempt_mipsx.s
preempt_nonwindows
preempt_ppc64x.s
preempt_riscv64.s
preempt_s390x.s
preempt_wasm.s
print.go
proc.go
proc_runtime_test.go
proc_test.go
procfunc.s

3424     *(*uintptr)(unsafe.Pointer(newg.stack.lo)) = 0
3425 }
3426     return newg
3427 }
3428
3429 // Create a new g running fn with siz bytes of arguments.
3430 // Put it on the queue of g's waiting to run.
3431 // The compiler turns a go statement into a call to this.
3432 // Cannot split the stack because it assumes that the arguments
3433 // are available sequentially after &fn; they would not be
3434 // copied if a stack split occurred.
3435 //go:nosplit
3436 func newproc(siz int32, fn *funcval) {
3437     argp := add(unsafe.Pointer(&fn), sys.PtrSize)
3438     gp := getg()
3439     pc := getcallerpc()
3440     systemstack(func() {
3441         newproc1(fn, argp, siz, gp, pc)
3442     })
3443 }
```

我发现 runtime 下有些函数不让打断点? 只能用 dlv 来 b. 不知道为何

## 回答

按道理来说 goland 用的就是 dlv 的。。

## 12. M 会在什么时候被销毁

### 问题

想问一下曹大, M 会在什么情况被销毁?

### 回答

默认是不销毁的, 有些场景还会碰到 M 创建数量过多的问题,

解决办法是:

<https://xargin.com/shrink-go-threads/>

## 13. dlv 调试问题

### 问题

# 一手资源+V 307570512

dlv 打断点调试时，能给指定函数传入参数吗？

## 回答

可以修改传进来的参数：

```
Plain Text
> main.main() ./add.go:3 (hits goroutine(1):1 total:1) (PC: 0x105edbf)
 1:     package main
 2:
=> 3:     func main() {
 4:         add(1, 2)
 5:     }
 6:
 7:     //go:noinline
 8:     func add(x, y int) (int, bool) {
(dlv) c
> main.add() ./add.go:8 (hits goroutine(1):1 total:1) (PC: 0x105ee0f)
 3:     func main() {
 4:         add(1, 2)
 5:     }
 6:
 7:     //go:noinline
=> 8:     func add(x, y int) (int, bool) {
 9:         var z = x + y
10:         println(z)
11:         return x + y, true
12:     }
(dlv) args
x = 1
y = 2
~r2 = 824633852016
~r3 = true
(dlv) set x = 3
(dlv) set y = 10
(dlv) args
x = 3
y = 10
~r2 = 824633852016
~r3 = true
(dlv)
```

[14. 调度循环切换 goroutine 的条件](#)

## 问题

# 一手资源+V 307570512

在一个调度循环中，是怎么判断要执行 goexit，结束本次 goroutine 的调用，我能够想到的就是协程阻塞的情况，请问老师还有什么情况下会结束本次调度？

## 回答

第一课应该讲过，如果碰到可接管的阻塞，就进 gopark 挂起正在执行的 g，然后进下一次调度

具体的场景翻一下 ppt~

## 再次追问

如果这个 goroutine 不会遇到阻塞，比如一个耗时超级长的计算任务，那是会一直做这个计算任务结束才会进入到下一个调度循环吗？有没有时间限制呢。。。唉，说着说着咋感觉这好像就是 sysmon 的 retake 操作？

## 回答

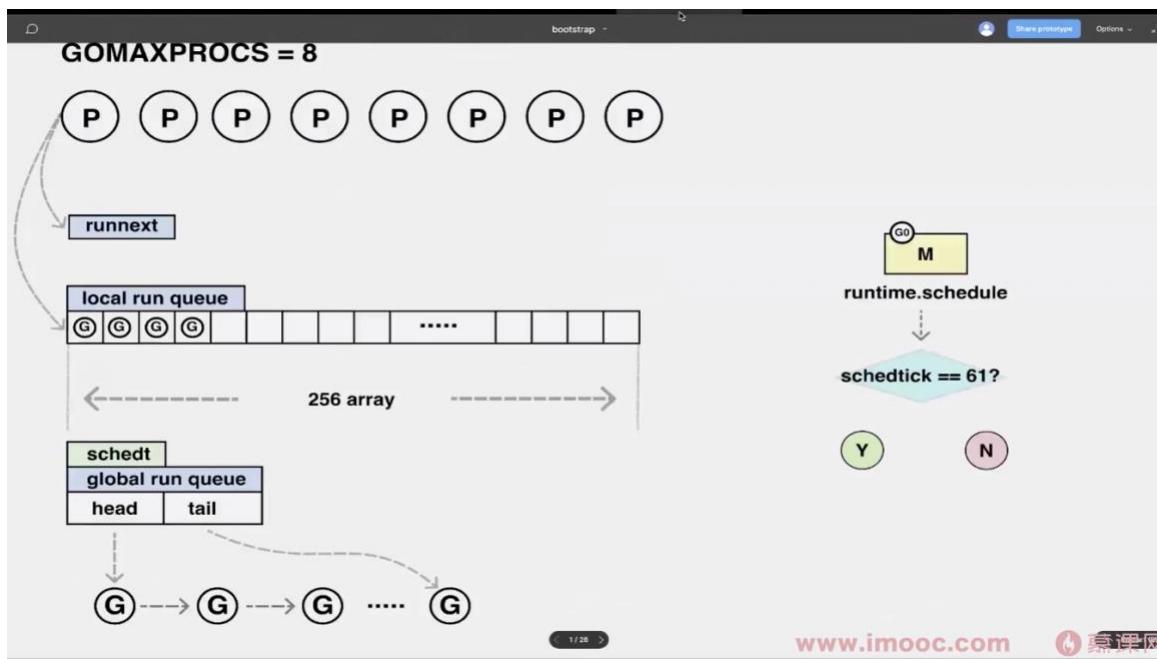
对的，你说的这个是 1.14 有了抢占式调度以后加的逻辑，有个 10ms 的抢占~

翻了一下 1.13 的代码，10ms 的抢占应该一直是有的，所以上面这个说的有问题~

[15. 本节课的 GMP 模型的动画演示链接可以发一下嘛](#)

## 问题

# 一手资源+V 307570512



## 回答

<https://golearn.coding.net/p/gonggongbanji/files/all/default/preview/23800957>

### 16. 调度器是通过什么方式运行的？

## 问题

请问，所有的代码都需要通过 M 和系统线程关联才能在 CPU 上运行，那调度器要执行调度也是要将自己封装成对应的 G 么？

## 回答

调度器这个东西没有实体，Go 里的调度器只能说是 GPM 和一堆相关结构的总称

Go 的调度过程主要是通过 M 绑定 P 之后，不断地执行调度循环，

调度循环的过程就是消费和执行 G。

### 17. 关于可以被接管的阻塞 goroutine 问题

## 问题

# 一手资源+V 307570512

## 问题描述:

可以被 runtime 接管的阻塞 goroutine，放在等待结构体的时候，这个 goroutines 是在 local queue 上么。当阻塞恢复时，这个 goroutine 是优先进行调度还是按照正常顺序进行调度？

请老师解答。

## 回答

放在等待结构体的时候，不在 run queue，可以从字面上理解一下 run queue 的含义，run = 可执行

阻塞恢复的时候，要进 run queue，(这里说的是可以接管的那些阻塞， syscall 的有另一套流程)，一般情况下是高优先级调度(可以看看 runtime.ready 这个函数， next 参数= true 就是高优先级)。

## 再次追问

```
Plain Text
next {
retryNext:
    oldnext := _p_.runnnext
    !_p_.runnnext.(oldnext.(gp))) {
        retryNext
    }
    oldnext == {
    }
    gp = oldnext.()
}
```

这个高优先级，就是有限使用 runnext 指针（把老 g 的踢到队列里）

看到除了 gcsweep，其他的 ready 调度都是 next 都是 true

## 回答

是的，字数补丁

阻塞的时候不在 **run queue** 队列里（这里可以理解一下 run queue 的 run 的意思，在 run queue 里的 g 应该都是 runnable 状态，这节课没有讲 goroutine 的状态切换，因为内容太多了(我们在最后会补)

而是在专门的等待结构里，对应的就是 PPT 图里的 channel 对应的 sendq、recvq 锁的 treap 里的节点的链表，等等

# 一手资源+V 307570512

## 18. go build 过程中的疑问

### 问题

```
root@Zhang-PC:/mnt/c/Users/zxm66/Documents/gowork/src/mock_test/hello# go build -x hello.go
$WORK/b001/exe/a.out -importcfg $WORK/b001/importcfg.link -buildmode=exe -buildid=cLNgGV0xYF6nDHCm57m/lDHzb7nIN_ml38AXPB9/iChQT9Nb-ic_xLMheGh/cLN3
GGDVxYF6nDHCm57m-extdl=cc $WORK/b001/pkg.a
$WORK/b001/buildid -w $WORK/b001/exe/a.out # internal
cp $WORK/b001/exe/a.out hello
rm -r $WORK/b001/
```

待解

→链接.

在试 go build -x hello.go 的时候，红线部分是.a 的一个链接文件吗。发现这个是因为，无论我是删掉可执行文件还是整个 hello.go 删掉，只要没有删掉这个目录，再次执行 go build 的时候都会去从 cache 里面找有没有链接文件。哪里有对应的代码吗？

### 回答

<https://github.com/golang/go/blob/24cff0f0444793be81062684c478a3f7ca955499/src/cmd/go/internal/work/exec.go#L444>

## 19. work steal 如何实现随机偷取一个 P 的 goroutine

### 问题

# 一手资源+V 307570512

```
var stealOrder randomOrder

// randomOrder/randomEnum are helper types for randomized work stealing.
// They allow to enumerate all Ps in different pseudo-random orders without repetitions.
// The algorithm is based on the fact that if we have X such that X and GOMAXPROCS
// are coprime, then a sequences of (i + X) % GOMAXPROCS gives the required enumeration.

type randomOrder struct {
    count  uint32
    coprimes []uint32
}

type randomEnum struct {
    i      uint32
    count uint32
    pos   uint32
    inc   uint32
}
```

我找了一下源码，感觉应该是这里，但是看了很多遍没能理解这段注释的意思

## 回答

这个是个数学性质，假设元素总共有 Y 个，  
我们选择一个数字和 Y 互质，比如 X，  
然后选一个起始位置： i  
然后用下面这样的代码：

```
Plain Text
for j := i;j+= X {
    pos := j % Y
}
```

可以把 Y 个元素全部遍历一遍，  
且不重复

比如你有 4 个元素，我选 X = 3，3 和 4 互质，选起始位置 0

那么我的遍历位置分别为：

```
(0+3)%4 = 3
(3+3) % 4 = 2
(6+3) % 4 = 1
(9+3) % 4 = 0
```

这样看起来是个伪随机序列，能把 4 个元素都遍历完

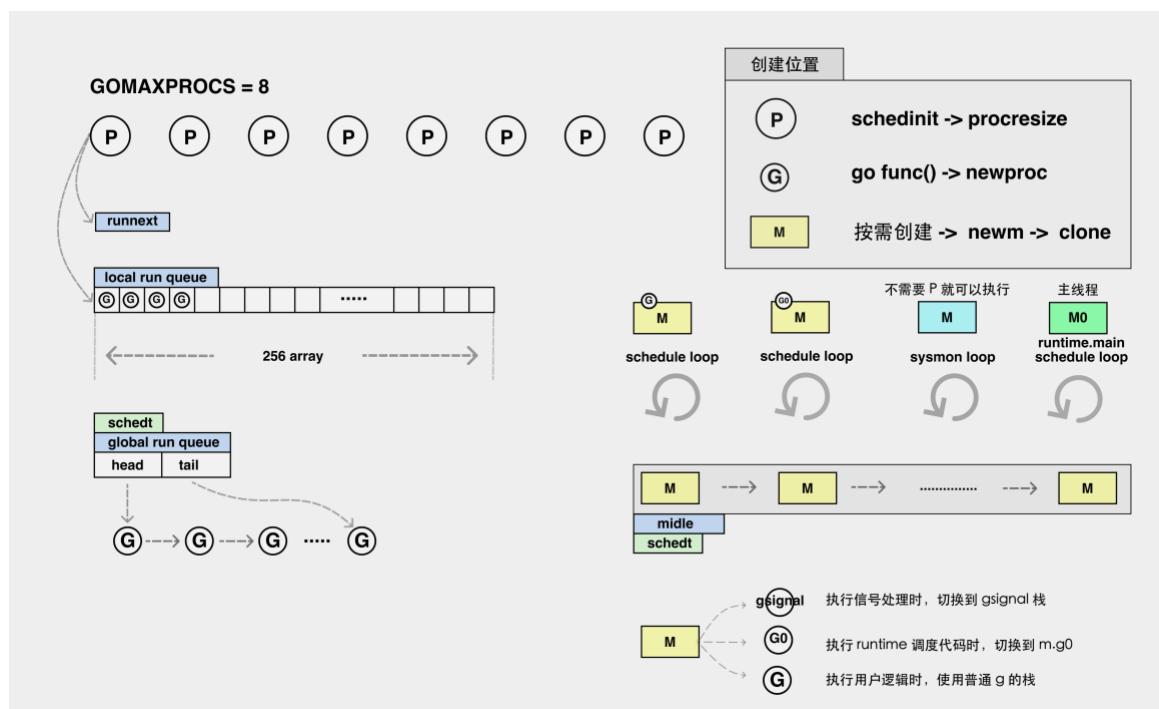
你可以自己实验一下~其它的 x 和 y~

## 20. M0 线程

### 问题

既然 m0 是 GO 程序启动后的第一个线程，那么关于“hello world”程序运行时只有一个主线程在运行的说法是不是错误的说法，这个主线程与 m0 是一个线程吗？

### 回答



我们第一课的图应该有这些。。。

sysmon 那个必须得有的呀，不太可能只有一个线程

**update 2021.06.20**

m0 就是主线程，操作系统自动帮我们创建的，在 Go 里也会执行调度循环

## 21. runtime 及其初始化（一）

### 问题

为了正确对后续问题进行提问的铺垫提问(老师概括性地回答即可，必要的地方可以指出代码位置和探索方式)，主要是对 rt0\_go 过程（包括 rt0\_go 结束）中涉及 runtime.main 与

# 一手资源+V 307570512

main.main 的 GMP、schedule 产生的疑问  
:from runtime/proc.go line 592

```
Go
The bootstrap sequence is:/
```

```
call osinit
```

```
call schedinit
```

```
make & queue new G
```

```
call runtime·mstart
```

```
The new G calls runtime·main.
```

上述是 package runtime 下 runtime bootstrap sequence 注释，围绕这几个过程提出几个概念性问题

- **问题抛出 1：**为了探究 ch1 课件中 runtime.main 在赣神魔的问题，先向上寻找 runtime 之前是谁在做什么并引起了一系列问题。我在找到了这段注释之后对 schedinit 进行了资料查找，发现 schedinit 是作为调度器（schedule）的初始化函数并且：

资料://From <https://draveness.me/golang/docs/part3-runtime/ch06-concurrency>

我们从环境变量 GOMAXPROCS 获取了程序能够同时运行的最大处理器数之后就会调用 runtime.procresize 更新程序中处理器的数量，在这时整个程序不会执行任何用户 Goroutine，调度器也会进入锁定状态，runtime.procresize 的执行过程如下：1. 如果全局变量 allp 切片中的处理器数量少于期望数量，会对切片进行扩容；2. 使用 new 创建新的处理器结构体并调用 runtime.p.init 初始化刚刚扩容的处理器；4. 通过指针将线程 m0 和处理器 allp[0] 绑定到一起；5. 调用 runtime.p.destroy 释放不再使用的处理器结构；6. 通过截断改变全局变量 allp 的长度保证与期望处理器数量相等；7. 将除 allp[0] 之外的处理器 P 全部设置成 \_Pidle 并加入到全局的空闲队列中；调用 runtime.procresize 是调度器启动的最后一步，在这一步过后调度器会完成相应数量处理器的启动，等待用户创建运行新的 Goroutine 并为 Goroutine 调度处理器资源。

D1: 经过 dlv 调试，对 schedinit 进行断点调试，发现其由 rt0\_go 栈跳转，

执行了 CALL runtime·args(SB)、CALL runtime·osinit(SB)、CALL runtime·schedinit(SB)、

下一步将 create a new goroutine to start program 入口为 MOVQ \$runtime·mainPC(SB), AX

此时输出全局 runtime.g0 runtime.m0 发现：

```
Plain Text
```

```
m0 与 g0 绑定
```

```
m0 中 P 与 nextp 均为 0, curg 为 nil
```

Q1: 在 schedinit 中调用的 procresize 对 allp 切片进行调整并初始化 p，将 allp[0] 与 m0 绑定在一起

# 一手资源+V 307570512

Q1.1: 为什么是 allp[0]与 m0 绑定

Q1.2: 在此之前的 rt0\_go 调用中是否载体都是 m0?若是, 那这个点之前的 m0 都没有绑定 p, 为什么仍能执行? (curg 也没有好奇心)

Q1.3: 资料中描述调用 runtime.procesize 是调度器启动的最后一步,但我并未在 procesize 代码中找到显式代码, 能指明一下么。

```
(dlv) print runtime.m0
runtime.m {
    g0: *runtime.g {
        stack: (*runtime.stack)(0x5541c0),
        stackguard0: 140724536964072,
        stackguard1: 140724536964072,
        _panic: *runtime._panic nil,
        _defer: *runtime._defer nil,
        m: *(*runtime.m)(0x554340),
        sched: (*runtime.gobuf)(0x5541f8),
        syscallsp: 0,
        syscalppc: 0,
        stktopsp: 0,
        param: unsafe.Pointer(0x0),
        atomicstatus: 0,
        stackLock: 0,
        goid: 0,
        schedlink: 0,
        waitsince: 0,
        waitreason: waitReasonZero (0),
        preempt: false,
        preemptStop: false,
        preemptShrink: false,
        asyncSafePoint: false,
        panicOnFault: false,
        gcscandone: false,
        throwsplit: false,
        activeStackChans: false,
        parkingOnChan: 0,
        raceignore: 0,
        sysblocktraced: false,
        sysexitticks: 0,
        traceseq: 0,
        tracelastp: 0,
        lockedm: 0,
        sig: 0,
        writebuf: []uint8 len: 0, cap: 0, nil,
        sigcode0: 0,
        sigcode1: 0,
        sigpc: 0,
        gopc: 0,
        ancestors: *[]runtime.ancestorInfo nil,
        startpc: 0,
        racectx: 0,
        waiting: *runtime.sudog nil,
        cgoCtxt: []uintptr len: 0, cap: 0, nil,
        labels: unsafe.Pointer(0x0),
        timer: *runtime.timer nil,
        selectDone: 0,
        gcAssistBytes: 0, },
    morebuf: runtime.gobuf {sp: 0, pc: 0, g: 0, ctxt: unsafe.Pointer(0x0), ret: 0, lr: 0, bp: 0},
    divmod: 0,
    procid: 0,
    gsignal: *runtime.g nil,
    goSigStack: runtime.gsignalStack {
```

# 一手资源+V 307570512

```
morebut: runtime.gobut {sp: 0, pc: 0, g: 0, ctxt: unsafe.Pointer(0x0), ret: 0, lr: 0, bp: 0},
divmod: 0,
procid: 0,
gsignal: *runtime.g nil,
goSigStack: runtime.gsignalStack {
    stack: (*runtime.stack)(0x554398),
    stackguard0: 0,
    stackguard1: 0,
    stktopsp: 0,},
sigmask: runtime.sigset [0,0],
tls: [6]uintptr [5587392,0,0,0,0,0],
mstartfn: nil,
curg: *runtime.g nil,
caughtsig: 0,
p: 0,
nextp: 0,
oldp: 0,
id: 0,
mallocing: 0,
throwing: 0,
preemptoff: "",
locks: 0,
dying: 0,
profilehz: 0,
spinning: false,
blocked: false,
newSigstack: false,
printlock: 0,
incgo: false,
freeWait: 0,
fastrand: [2]uint32 [0,0],
needextram: false,
```

D2: The bootstrap sequence 中 make & queue new G 在 rt0\_go 找到了相应过程

```
SQL
// create a new goroutine to start program
MOVQ $runtime.mainPC(SB), AX      // entry
PUSHQ AX
PUSHQ $0      // arg size
CALL runtime.newproc(SB)
POPQ AX
POPQ AX
```

Q2: 这里 runtime.mainPC(SB) 执行完 SB 中存储的是哪个代码段的入口 (SB 存的是地址还是偏移量我也不清楚) ?

# 一手资源+V 307570512

D3: mstart 我实在是看得不太懂，只能看到 mstart1 中显式执行 schedule，只能问些粒度大的问题，希望曹大别嫌烦==。

```
Plain Text
// start this M
CALL    runtime·mstart(SB)

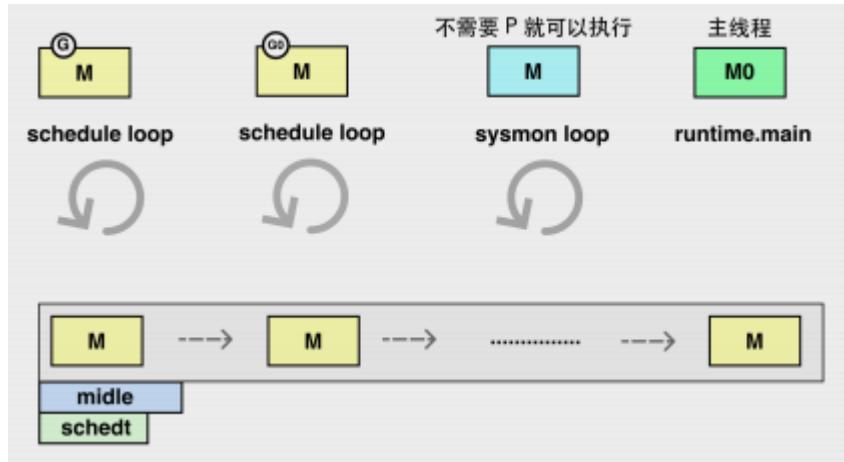
CALL    runtime·abort(SB)    // mstart should never return
```

Q3.1: mstart 的 M 是之前的 m0 么？

Q3.1: 在 mstart 中是如何跳转到之前生成的 G 的

# 一手资源+V 307570512

下次就问 runtime.main 了



## 回答

你需要先理解 m、线程这个概念还是一种多任务的抽象，操作系统执行代码的时候，在用户这边不定义线程也是可以执行的，比如：<https://github.com/cch123/lkp-trans/blob/d6b7f46c72e83ac9145d5534c6bc4e690da8d815/part1/assembly-language/example-calculating-string-length.md>。类似这个链接里的汇编程序，只是做一些运算，不会创建任何线程(用户态看来)，但仍然可以用 nasm 编译成为可执行文件来执行。

Q1:.....

A: 这不是个疑问句？

Q1.1: .....

A: 这种是个约定，p 理论上还能扩/缩容，如果你刚才是 4，现在是 2，那 m0 和 P 的绑定关系就会被调来调去。非要和 p[n-1] 绑定理论上也可以实现的~

Q1.2: ,...

A: 理解了开头的前提条件以后，你就知道代码执行的时候是可以没有线程的概念的，所以 rt0\_go 开头的这些汇编代码，在用户态是没线程概念的。到创建好各种 M 以后，后续的代码才都是在线程 M 上执行的。

Q1.3: .....

A: 像这种博客文章里的描述，你就不必要咬文嚼字，这意思只是想说 progresize 是 schedinit 里调用的最后一个函数而已。。非要咬文嚼字的话，后面还有三个 if 呢。

# 一手资源+V 307570512

```
566      }
567      if procsresize(procs) != nil {
568          throw(s: "unknown runnable goroutine during bootstrap")
569      }
570
571      // For cgocheck > 1, we turn on the write barrier at all times
572      // and check all pointer writes. We can't do this until after
573      // procsresize because the write barrier needs a P.
574      if debug.cgocheck > 1 {...}
581
582      if buildVersion == "" {...}
587      if len(modinfo) == 1 {...}
592  }
```

Q2: ....

A: 这种 \$开头的都是一些常量, (SB) 这种在汇编里一般表示的是全局变量, 这里 runtime.mainPC 说白了就是跟编译器约好的一个名字, 表示的是 runtime.main 函数的地址。 SB 这种是 Go 里的伪寄存器, 实际上是一个不存在的东西:

The SB pseudo-register can be thought of as the origin of memory, so the symbol foo(SB) is the name foo as an address in memory. This form is used to name global functions and data. Adding <> to the name, as in foo<>(SB), makes the name visible only in the current source file, like a top-level static declaration in a C file. Adding an offset to the name refers to that offset from the symbol's address, so foo+4(SB) is four bytes past the start of foo.

只是约定用来定义全局变量的。

Q3.1: mstart 的 M 是之前的 m0 么?

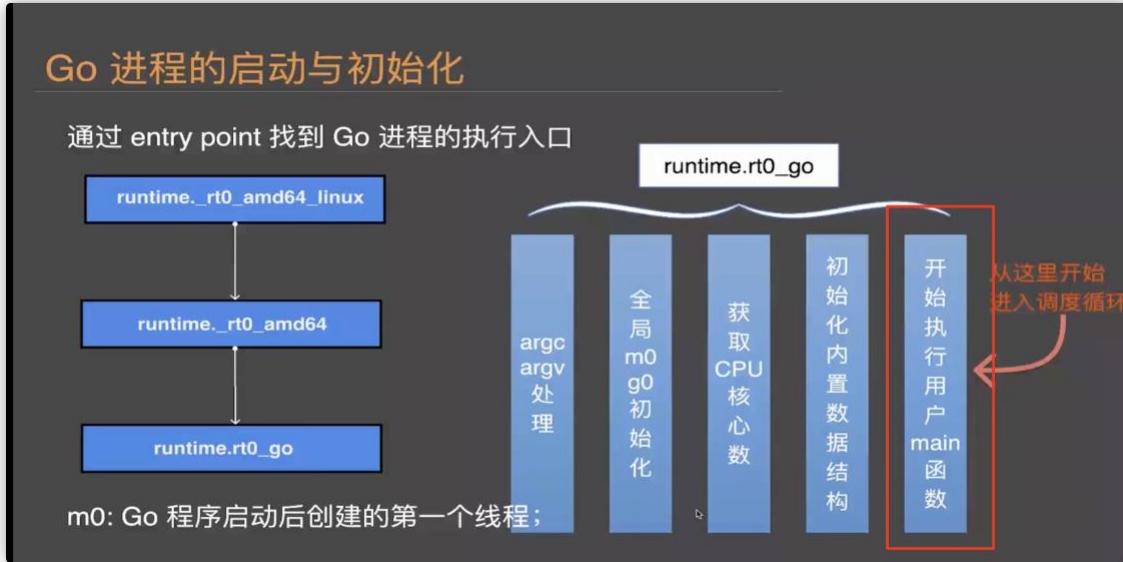
A. 启动阶段的 mstart 应该确实是启动的 m0, 本质是用 clone 系统调用创建了一个线程, 并把 runtime.mstart 的地址传进去。线程创建完成后, 会从 mstart 函数开始执行用户的代码。

Q3.2: 在 mstart 中是如何跳转到之前生成的 G 的

A. mstart 在把线程启动完毕之后, 一定是去调用 schedule。在 m0 启动之前, 已经通过 newproc 创建了一个 runtime.main 入口的 goroutine。m0 通过 mstart 启动之后, 最后去执行 schedule, 这时候应该只有一个 goroutine, 那总是能找到他的。。

## [22. rt0\\_go 启动过程](#)

## 问题



这里为什么是执行 main 函数，不是应该会先执行 init 方法之类的吗？

## 回答

有 init，我在 ppt 里省略掉了，你看这里：

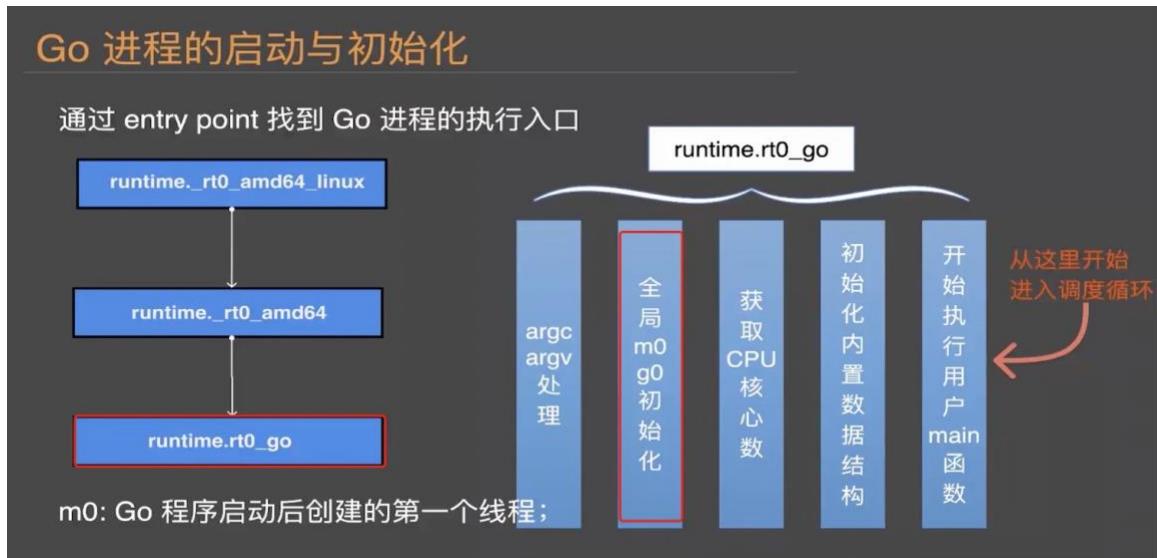
```
187     cgocall(_cgo_notify_runtime_init_done)
188 }
189
190 doInit(&main_inittask)
191
192 close(main_init_done)
193
194 needUnlock = false
195 unlockOSThread()
```

细节都画出来会比较复杂~

## [23. 关于 runtime.rt0\\_go 的执行](#)

## 问题

1. 第一课里面这里讲到特殊的线程，是指 runtime.rt0\_go 这个函数执行是一个特殊线程，还是全局 m0，g0 的初始化是一个特殊的线程呀？



2. 另外 runtime.rt0\_go 是有执行顺序是从左到右执行的，还是从右到左执行的呀

## 回答

我们作业讲解那节会讲咋调这些步骤~

rt0\_go 执行的时候还在初始化阶段，其实还没线程的概念呢~

执行顺序是从左向右的，这个图里是一些概览，其实代码里还有好多细节

24. 在有很多个 M 的情况下，新的 goroutine 会放到哪个 M 下

## 问题

在有很多个 M 的情况下，新的 goroutine 会放到哪个 M 下吗  
是当前执行 goroutine 的 M 吗 这样的话 什么时候会放在别的 M 里面呢  
请老师解答。

## 回答

# 一手资源+V 307570512

当前执行 goroutine 的那个 M，它绑定了一个 p，会放在这个 p 的 runnext、local runq，然后才是 global runq

这个应该第一节课讲过了，看一下当时的动画吧~

## 再次追问

gosdk:1.16.14

比如设置 GOMAXPROCS 为 4 个

```
Plain Text
runtime.GOMAXPROCS(4)
```

```
Plain Text
for i := 0; i < 11; i++ {
    go func(i int) {
        fmt.Println(i)
    }(i)
}
time.Sleep(time.Hour)
```

这种情况是不是跑到其他 M 绑定的 P 下面去执行了呢 如果是的话他是怎么挑选其他 M 绑定的 P 执行的呢

对于这点比较疑惑 动画里面似乎没有提到这一点

## 回答

g 默认是在本地队列的，

但是和其它 P 绑定的 M 在执行调度循环的时候，发现本地是空的话，会从另外的 P 里偷走他们的 g，这个在动画里也有

最终的效果其实就是达到了大致的平衡状态，所有 P 的本地队列里都有 g

像你这个例子里的代码，执行时间比较短的话，在一个 P 里面执行完理论上也是有可能的，被别人偷走也有可能

你甚至可以直接在 schedule 函数里加一些 print 日志来看这个结果

或者搜一下 schedtrace 这个 flag(执行太快的话，可能看不出效果来)

## [25. runnext 是什么数据结构？](#)

# 一手资源+V 307570512

## 问题

请问老师 runnext 只保存最新的一个 G，是什么数据结构！

## 回答

存 goroutine 的地址的，其实可以理解成一个指针

## [26. G 是怎么挑选 M 放入 P 下的](#)

## 问题

G 是怎么挑选 M 放入 P 下的

这个有点没明白

是通过一种随机算法吗 还是 会判断 M 的状态等相关

## 回答

go func 本质就是调用了 runtime.newproc 函数

调用这个函数之前有一个正在运行的 g，这个正在运行的 g 对应的线程，一定绑定了一个 P，

go func 创建出来的新的 g 就是往这里走的

## [27. 主 goroutine 包含哪些信息？](#)

## 问题

主 goroutine 包含哪些信息？

## 回答

主 goroutine 是执行主线程的，是 runtime.g0 的一个全局变量，在主 goroutine 退出的时候有一些特殊判断逻辑

其它的也没啥特殊的

## 28. map 的 bucketShift

### 问题

Plain Text

```
// bucketShift returns 1<<b, optimized for code generation.

func bucketShift(b uint8) uintptr {
    // Masking the shift amount allows overflow checks to be elided.

    return uintptr(1) << (b & (sys.PtrSize*8 - 1))
}
```

为什么和 b 进行与运算是 63 为 那这样如果 b 是 64 的话 这块计算的值就是了

### 回答

我想了想，应该是这么回事：

1. b 不可能是 64，你算算  $2^64$  个 bucket 的话，会占多少内存，所以不太可能有这么多
2. 按注释写的，这里保证移位  $\leq 63$ ，所以一定不会溢出，少掉一次溢出检查

估计这种函数是从老的 c 代码之类的地方继承来的，感觉这点性能优化可能在 map 那堆复杂操作里算不上个事。。。

## 29. 关于 P 的本地队列和全局队列

### 问题

老师说 runtext 和本地队列空了，本地队列会去全局队列截取一半的数据放到本地队列，但是本地队列的长度只有 256，那么如果全局队列的一半数据大于 256 的话怎么处理呢？

### 回答

是不是当时直播的时候说错了 orz，我后来在动画里修正了，

# 一手资源+V 307570512

是从全局队列里拿 ( $g$  总数 /  $gomaxprocs$ ) 个任务到本地，  
同时在拿的时候，有不能超过 128 的判断

## 再次追问

老师，应该是拿  $g$  总数 /  $gomaxprocs + 1$  个吧，我看您的动画里面是这样画的

## 回答

Plain Text

```
n := sched.runqsize/gomaxprocs + 1
```

这个 runqsize 是全局队列的  $g$  总数，我看看动画哦，不对的话我修一下

## 30. checkdead 中判断 $g$ 死锁逻辑

## 问题

Plain Text

```
func checkdead() {  
  
    // ...  
    grunning := 0  
    for i := 0; i < len(allgs); i++ {  
        gp := allgs[i]  
        if isSystemGoroutine(gp, false) {  
            continue  
        }  
        s := readgstatus(gp)  
        switch s && _Gscan {  
            case _Gwaiting,  
                _Gpreempted:  
                grunning++  
            case _Grunnable,  
                _Grunning,  
                _Gsyscall:  
                unlock(&allglock)  
                print("runtime: checkdead: find g ", gp.goid, " in status ", s, "\n")  
                throw("checkdead: runnable g")  
        }  
    }  
    //...
```

```
}
```

```
lock(&allglock)
for i := 0; i < len(allgs); i++ {
    gp := allgs[i]
    if isSystemGoroutine(gp, fixed: false) {
        continue
    }
    s := readgstatus(gp)
    switch s && _Gscan {
    case _Gwaiting,
        _Gpreempted:
        grunning++
    case _Grunnable,
        _Grunning,
        _Gsyscall:
        unlock(&allglock)
        print(args... "runtime: checkdead: find g ", gp.goid, " in status ", s, "\n")
        throw(s: "checkdead: runnable g")
    }
}
unlock(&allglock)
```

checkdead 中为啥检查全局的 g 在运行、可运行、或者 syscall 的状态就认为是死锁了，一个 g 处在这三个状态不正常么

## 回答

你看看前面，运行到这段逻辑的时候  $\text{run} = 0$  了，有 g 状态是 running，然而活动的 M 却是 0，逻辑上就有问题，所以要崩溃。

这个 checkdead 基本上在服务类应用里就触发不了。。。我感觉都没啥必要看

## 31. 生产环境中遇到的偶发问题如何定位？

### 问题

现象：服务正常接口的执行时间是 5ms 内，偶尔出现一些执行超 3 秒的请求（重试是 5ms 内）。

下游 timeout 是 3 秒，直接 504 了，也看不出实际用了几秒。

出现概率：0.008% 超时：5100/64800000

# 一手资源+V 307570512

1. mysql 连接池配置：

```
CSS
sqlDB.SetMaxIdleConns(100)
sqlDB.SetMaxOpenConns(100)
sqlDB.SetConnMaxLifetime(time.Hour)
```

2. cpu/内存/负载 正常
3. 数据库慢无查询， 负载正常  
问题一般如何定位？用什么工具定位
4. mysql 连接池问题？
5. goroutine 的调度有关？ qps250 左右， 应该还不到 go 的单机瓶颈
6. 网络波动？

## 回答

这种偶发问题感觉像是网络问题，生产环境有没有 tcp retrans 的监控呢？

之前 grab 的一个兄弟是用持续地抓包来排查的，他们碰到的丢包导致 tcp 队头阻塞，进一步导致的偶发延迟问题。

如果是偶发的 CPU/内存/goroutine 飙升，那是另外一套定位方法

## 33. Go 程序启动的疑问？

### 问题

曹大老师，按照本课的讲解和理解。

1. 我们写的代码，通过编译，输出为二进制可执行程序。
2. 在 linux 我们可以通过 readelf 下面查编译后的文件。
3. dlv 我们可以查看程序启动后的执行情况。

#### 问题描述：

编译的二进制文件，包含一系列指令和汇编代码，所以当我们执行程序的时候，操作系统会将文件读入内存，执行对应的指令。操作系统读取二进制可执行文件，其中就包括划分内存区域，分配数据在内存中（可能包含静态的，动态的等等）。这个应该是程序的初始化，1. 这个理解对不对？

2. 在初始化完成后，操作系统会读到 entry point address，因为初始化已经完成，这样读入 entry point 后实际就是开始执行程序的指令和数据了，对吗？
3. 有没有什么工具或者方法，能够查看或者分析操作系统如何初始化的过程？（readelf 能

查看吗？）

## 回答

1. 没什么问题
2. 是的

3. 在 linux 下执行 strace ./your\_binary 能看到一些加载相关的系统调用  
再细节的得去看 loaders 之类的知识了，这方面我个人不是特别感兴趣，你可以找找相关的书~比如国人写的那本《程序员的自我修养：链接、装载与库》或者国外的《linders and loaders》，这本在第二课的参考资料里我给出来了，参见 coding。

34. 为什么在不同机器上编译不同的程序 entry point 都是一样的？

## 问题

曹大老师，在使用`readelf`的时候有一个疑惑，就是无论编译上面程序，他们的`readelf`查看的 entry point address 为什么都一样？而且居然在不同的机器上，居然也都是`0x455780`。为什么 entry point 都是一样的呢？这是 virtual memory management 的原因吗？但是我想不同的机器上内存使用情况都不一样吧。`0x455780`是不是有什么特定含义或者用途？

## 回答

地址都一样大概有下面一些方面的原因：

1. reproducible builds 是编译里的一个概念，就是说只要我们的源代码相同，平台一致(如，都是 linux amd64)，Go 版本一致，那么编译出来的二进制文件也应该是完全一致的，这个概念你可以看看 [https://en.wikipedia.org/wiki/Reproducible\\_builds](https://en.wikipedia.org/wiki/Reproducible_builds)
2. 从源代码到二进制，中间要经过 编译-> 链接两个过程，链接过程给文件内的符号分配的都是虚拟地址，二进制文件完全一致的情况下，虚拟地址也应该是一致的。

35. 关于 go 进程的启动与初始化

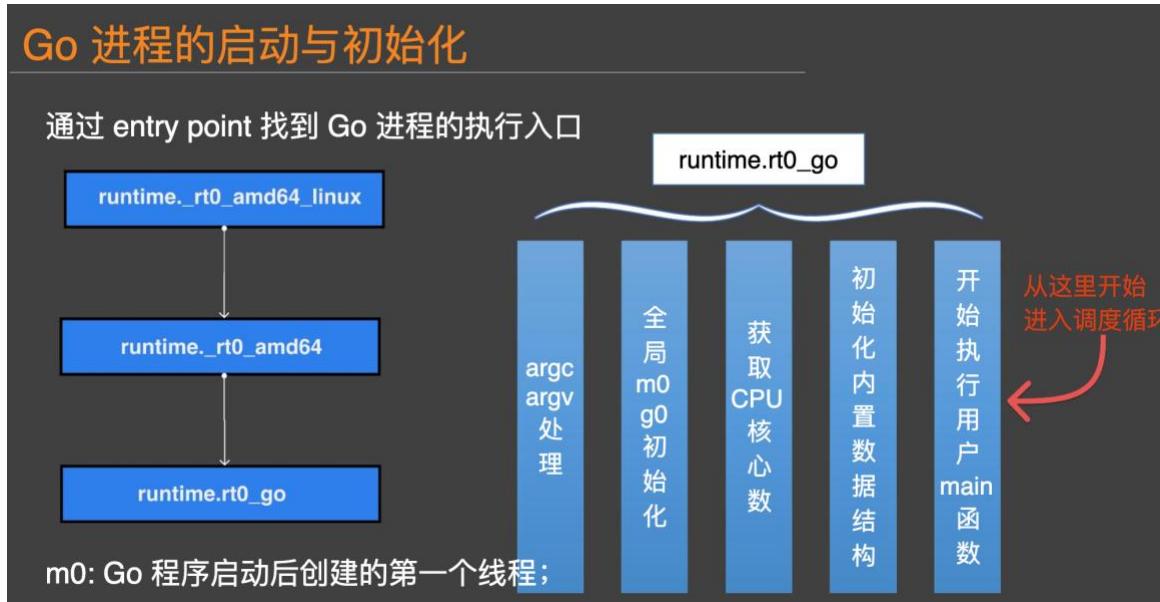
## 问题

问题描述：

怎样才能只能直观的感受 go 进程的启动与初始化呢，可以通过一些工具或命令进行发现吗

# 一手资源+V 307570512

相关截图：



## 回答

看一下第三节答疑课的开头的作业演示部分

## 36. 关于查看 PC 寄存器中的指令

## 问题

问题描述：

要执行的二进制机器码指令是存放在 pc 寄存器中的吗，怎么查看一段文本对应的机器码呢，不知道怎么才能得到右侧的内容

相关截图：

```
TEXT hello.main(SB) /Users/xurgia/test/hello.go
hello.go:1 0x1051578 05448b625300000000 MOVO GS:0x38, CX
hello.go:2 0x1051579 483b6110 CMPQ 0x18(CX), SP
hello.go:3 0x105157a 70000000 SUBQ 0x10(SP)
hello.go:3 0x105157f 4883ec18 SUBQ 0x10, SP
hello.go:3 0x1051583 48896c2410 MOVO BP, 0x10(SP)
hello.go:3 0x1051587 48896c2410 LEAO 0x10(SP), SP
hello.go:3 0x105158b 4883c410 JLEA 0x10(SP), SP
hello.go:4 0x1051592 4880d54ffcc0100 LEAO go.string.+$20(SB), AX
hello.go:4 0x1051593 48896c2410 MOVO AX, 0x10(SP)
hello.go:4 0x1051594 48c7f42440000000000 MOVO 0x10(SP)
hello.go:4 0x1051596 005543ffff CALL runtime.printString(SB)
hello.go:4 0x1051597 009838ffff CALL runtime.printUnloc(SB)
hello.go:4 0x1051598 009838ffff MOVO 0x10(SP)
hello.go:4 0x1051599 4083c410 ADDQ 0x10, SP
hello.go:4 0x105159b C3 RET
hello.go:4 0x105159c 48411fffff CALL runtime.morestack_noctxt(SB)
hello.go:4 0x105159d ehaf JMP main.main(SB)
```

请曹大指导一下

## 回答

ppt 上右边的内容是用：

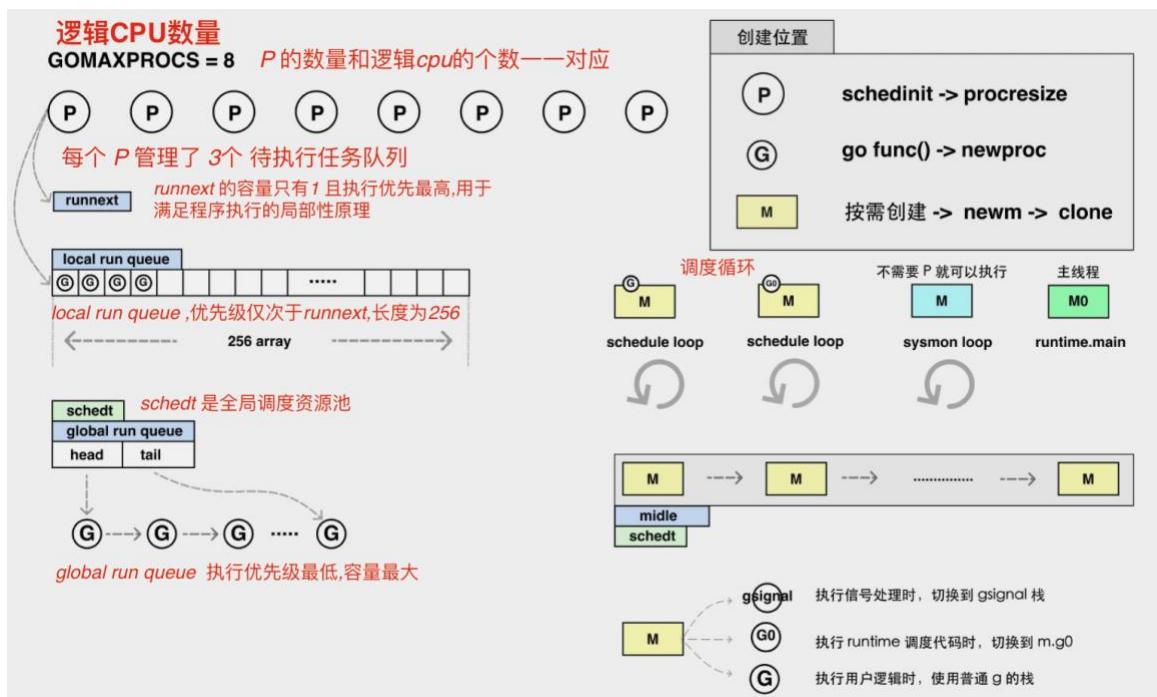
go tool objdump ./二进制文件 | grep -A 20 "main.main"  
得到的

在 dlv 调试时 disass 也可以得到

### 37. 关于 schedt global run queue ,

## 问题

global run queue 这是个链表队列 是所有 P 公用的么? 应为是公用的才需要锁对么?  
入下图这样理解有错误么?



全局队列是双向还是单向的?

## 回答

对的， global run queue 是所有 P 共享的，大家访问要加锁

# 一手资源+V 307570512

global run queue 不归属任何一个 P，是个链表

你图上那个“每个 P 管理三个待执行任务队列”有点问题吧，和 P 相关的应该就只有 runnext 和 local run queue

然后就是 GOMAXPROCS 默认和 CPU 核数一致(但是可以通过环境变量修改)

其它的没啥问题

## 38. 关于 newproc 问题

### 问题

Q1: newproc 使用 systemstack 切换的栈空间是谁的栈空间（在调度模型中有对应的对象么）？

Q2: 若在 systemstack 环境下使用 getg 获取的是 g0，为什么要用 g0 获取 P 而不是 g? (不是都关联着同一个 M 么？※)

```
Plain Text
type m struct { type m struct {
...
...
g0 *g           m *m
curg *g         ...
...
}
}
```

# 一手资源+V 307570512

```
func newproc(siz int32, fn *funcval) {
    argp := add(unsafe.Pointer(&fn), sys.PtrSize)
    gp := getg()
    pc := getcallerpc()
    systemstack(func() {
        newg := newproc1(fn, argp, siz, gp, pc)

        _p_ := getg().m.p.ptr()
        runqput(_p_, newg, true)

        if mainStarted {
            wakelp()
        }
    })
}
```

相关截图：

老师在群里给出的图

```
package main

func main() {
    println("main goroutine")
    go func() {
        newproc -> getg
        println("hello")
    }()
}
```

## 回答

Q1: newproc 使用 systemstack 切换的栈空间是谁的栈空间（在调度模型中有对应的对象么）？

systemstack 函数调用，会切到 m 的 g0 栈上，systemstack 返回的时候，会切回原来的 g 栈

Q2: 若在 systemstack 环境下使用 getg 获取的是 g0，为什么要用 g0 获取 P 而不是 g? (不

是都关联着同一个 M 么？※)

我看这两个值是一样的，可以调试打断点输出出来看看

# 一手资源+V 307570512

Plain Text

```
> runtime.newproc1() /usr/local/go/src/runtime/proc.go:3266 (PC: 0x1031dea)
Warning: debugging optimized function
3261: // at argp. callerpc is the address of the go statement that created
3262: // this. The new g is put on the queue of g's waiting to run.
3263: func newproc1(fn funcval, argp uint8, narg int32, callergp *g, callerpc uintptr) {
3264:     _g_ := getg()
3265:
=>3266: if fn == nil {
3267:     _g_.m.throwing = -1 // do not dump full stacks
3268:     throw("go of nil func value")
3269: }
3270: acquirem() // disable preemption because it can be holding p in a local var
3271: siz := narg
(dlv) p callergp.m.p
824633877760
(dlv) p _g_.m.p
824633877760
(dlv)
```

不过我猜测，可能在创建 newproc 的时候，可能会因为抢占调度之类的原因，导致创建出来的 g 和原来的不在一个 P 上了？但是不太好验证

[39.](#) 本地队列队列为空会去全局队列截取一半 goroutine 放入本地队列

## 问题

老师您说本地队列的长度是固定 256，那如果全局队列的一半超过了 256 是怎么处理的啊？

## 回答

最多只拿本地队列的一半，有代码为证：

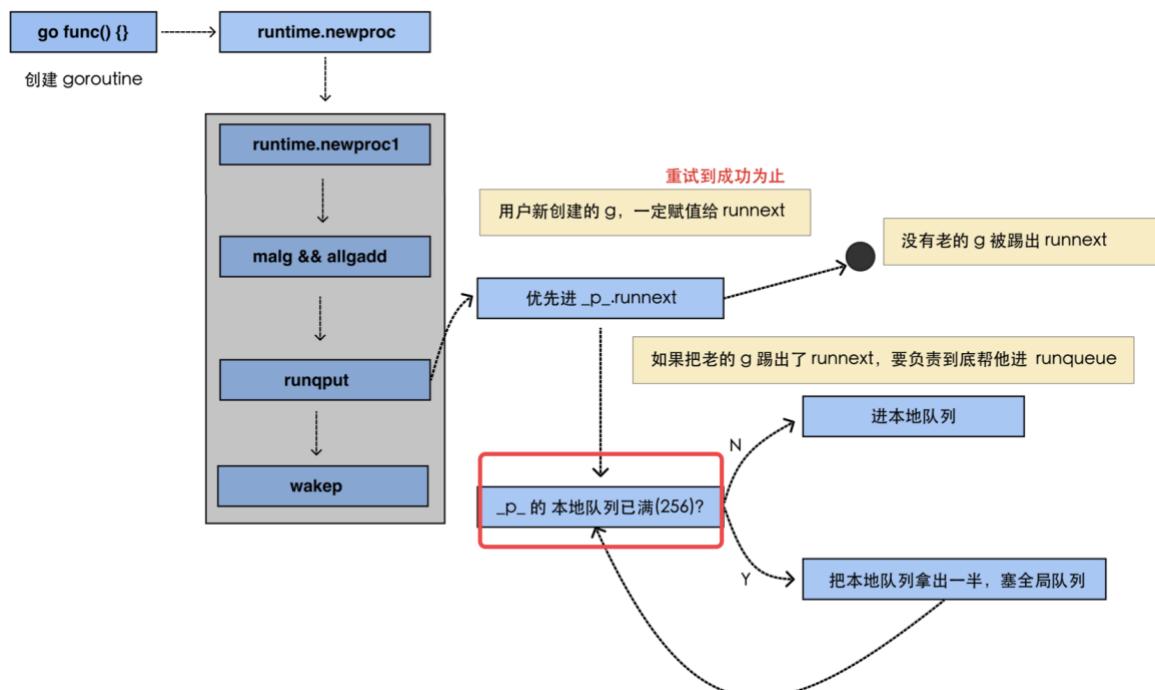
```
Plain Text
if n > int32(len(_p_.runq))/2 {
    n = int32(len(_p_.runq)) / 2
}
```

其中 \_p\_.runq 的长度是 256。

具体看看代码：src/runtime/proc.go:5571

## 40. goroutine 生产端和消费端的协作的疑问

### 问题



在运行 runnext 的时候，如果是 true，就将任务 kick out

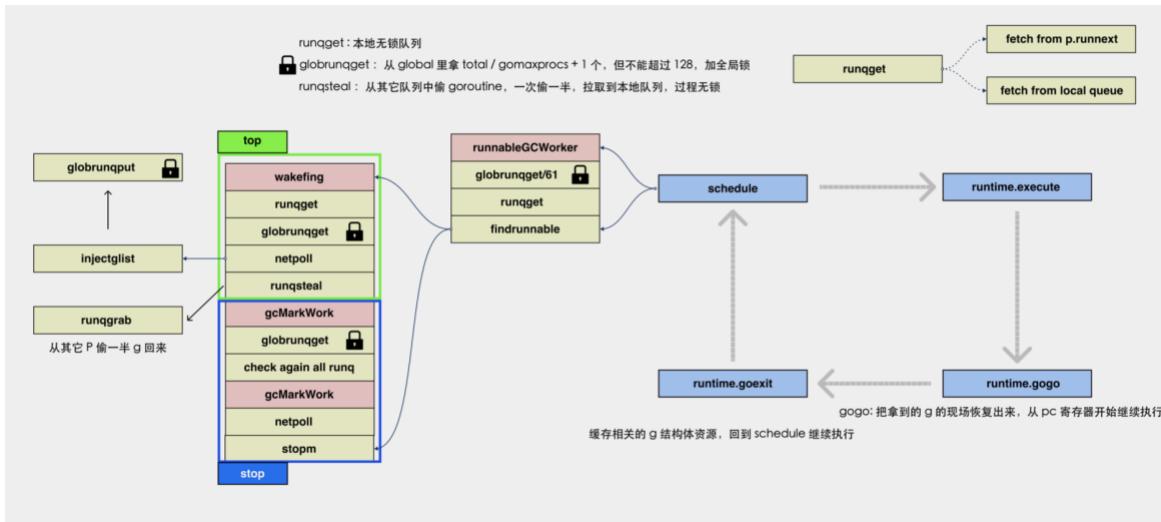
如果 next 是 false，就将 g 加到 runnable queue

`_p_.runq[t%uint32(len(_p_.runq))].set(gp)`

如果 runnable queue 已经满了，就执行 runqputslow 这个 slow 就是加到全局队列吧？

在 runqputslow 函数中，执行 globrunqputbatch，将本地队列的 g 一起加到全局队列上，并且此时要将调度器锁上，队列操作完了再解锁。

# 一手资源+V 307570512



但是消费端的“top”是在 park\_m 函数中，调用来源是在 proc 的 init 函数中（也就是说 proc 被 import 的时候，消费端就启动了），会一直循环的调用 goparkunlock，在这里会调用 gopark 函数来执行消费任务的调度。

所以我们第一次课学的其实主要就是 proc.go 中的一系列函数的调用过程。

问题：在 proc 的 init 函数中 goparkunlock 函数，但是在 dlv 的调用中发现调用来源是来自于

golang/src/runtime/asm\_amd64.s:1373

```
252:         atomic.Store(&forcegc.idle, 1)
=> 253:         goparkunlock(&forcegc.lock, waitReasonForceGGIdle, traceEvGoBlock, 1)
254:         // this goroutine is explicitly resumed by sysmon
255:         if debug.gctrace > 0 {
256:             println("GC forced")
257:         }
258:         // Time-triggered, fully concurrent.

(dlv) up
> runtime.gopark() /usr/lib/golang/src/runtime/proc.go:287 (hits goroutine(2):1 total:1) (PC: 0x430ea3)
Warning: debugging optimized function
Frame 3: /usr/lib/golang/src/runtime/asm_amd64.s:1373 (PC: 45b7c1)
1368:     RET
1369:
1370: // The top-most function running on a goroutine
1371: // returns to goexit+PCQuantum.
1372: TEXT runtime.goexit(SB),NOSPLIT,$0-0
=>1373:     BYTE    $0x90 // NOP
1374:     CALL    runtime.goexit1(SB)    // does not return
1375:     // traceback from goexit1 must hit code range of goexit
1376:     BYTE    $0x90 // NOP
1377:
1378: // This is called from .init_array and follows the platform, not Go, ABI.
```

这里为什么是来自于 goexit1 的调用的呢？但是实际看到的是调用到了 proc.main 函数，  
gcenable 调用的消费端

# 一手资源+V 307570512

```
1372: TEXT runtime·goexit(SB),NOSPLIT,$0-0
=>1373:     BYTE    $0x90 // NOP
1374:     CALL    runtime·goexit1(SB) // does not return
1375:     // traceback from goexit1 must hit code range of goexit
1376:     BYTE    $0x90 // NOP
1377:
1378: // This is called from .init_array and follows the platform, not Go, ABI.
(dlv) down
> runtime.gopark() /usr/lib/golang/src/runtime/proc.go:287 (hits goroutine(1):1 total
Warning: debugging optimized function
Frame 4: /usr/lib/golang/src/runtime/proc.go:166 (PC: 430ac3)
161:         }()
162:
163:         // Record when the world started.
164:         runtimeInitTime = nanotime()
165:
=> 166:         gcenable()
167:
168:         main_init_done = make(chan bool)
169:         if iscgo {
170:             if _cgo_thread_start == nil {
171:                 throw("_cgo_thread_start missing")
(dlv) █
```

## 回答

如果 runnable queue 已经满了，就执行 runqputslow 这个 slow 就是加到全局队列吧？  
runqputslow 是先做一些准备工作(准备那个 batch 结构)，然后再调用加到全局队列的函数

你后面这些问的我看着有点晕。。主要是想说调用方和自己想的不一样么？

## 再次追问

消费用 dlv 定位的调用是从 goexit1 这里开始的。所以 goroutine 的消费端是在函数退出前开始执行吗？

# 一手资源+V 307570512

```
1372: TEXT runtime·goexit(SB),NOSPLIT,$0-0
=>1373:     BYTE    $0x90 // NOP
1374:     CALL    runtime·goexit1(SB) // does not return
1375:     // traceback from goexit1 must hit code range of goexit
1376:     BYTE    $0x90 // NOP
1377:
1378: // This is called from .init_array and follows the platform, not Go, ABI.
(dlv) down
> runtime.gopark() /usr/lib/golang/src/runtime/proc.go:287 (hits goroutine(1):1 total
Warning: debugging optimized function
Frame 4: /usr/lib/golang/src/runtime/proc.go:166 (PC: 430ac3)
161:         }()
162:
163:         // Record when the world started.
164:         runtimeInitTime = nanotime()
165:
=> 166:         gcenable()
167:
168:         main_init_done = make(chan bool)
169:         if iscgo {
170:             if _cgo_thread_start == nil {
171:                 throw("_cgo_thread_start missing")
(dlv)
```

你是想说主线程的调度循环是怎么启动的么？

gopark 和 goparkunlock 是用来挂起那些应该阻塞的 goroutine 的，相当于正在执行的 goroutine 让他们进等待结构(例如 channel 的 recvq)，然后线程继续执行 schedule 函数做消费

调度循环指的是：

Plain Text  
schedule --> execute --> gogo --> goexit --> goexit1 --> goexit0 --> schedule  
挂起(gopark/goparkunlock)再到 schedule 算是调度循环里的一段逻辑

## 41. 阅读 runqput 源码的几个疑问

### 问题

Plain Text  
func runqput(\_p\_ \*p, gp \*g, next bool) {
 if randomizeScheduler && next && fastrand()%2 == 0 {
 next = false
 }
}

# 一手资源+V 307570512

```
if next {  
    retryNext:  
        oldnext := _p_.runnnext  
        if !_p_.runnnext.cas(oldnext, guintptr(unsafe.Pointer(gp))) {  
            goto retryNext  
        }  
        if oldnext == 0 {  
            return  
        }  
        // Kick the old runnnext out to the regular run queue.  
        gp = oldnext.ptr()  
    }  
}
```

问题一

上述代码中 `if randomizeScheduler && next && fastrand()%2 == 0` 这个判断是不是只是为了关闭调度中的 `runnnext` 机制？具体什么情况下会用到这种机制呢？

问题二

`if !_p_.runnnext.cas(oldnext, guintptr(unsafe.Pointer(gp)))`, 我的理解应该是用新加入的 `gp` 替换 `oldnext`, 如果失败了就重试, 那么什么情况会失败呢? 还是说这只是一个代码规范性问题?

问题三

`if oldnext == 0` 是因为 `gp` 加入进来之前, `runnnext` 里是空的, 所以把 `gp` 塞入 `runnnext` 就不管了, 直接 `return`。不知道我的理解对不对

## 回答

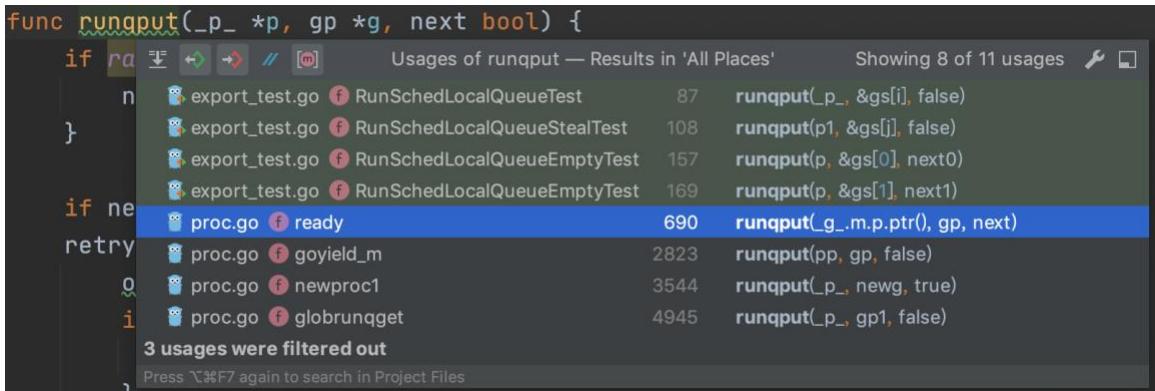
问题一:

这个是和 race 相关的, 一般我们看源码的时候, 碰到 race 直接跳过, 不用深究。

问题二:

这个 cas 的意思是将 `_p_.runnnext` 与 `oldnext` 比较, 如果相等, 再替换, 是一个原子操作。主要是其他地方也有 goroutine 相往这个 P 上塞。

# 一手资源+V 307570512



```
func runqput(_p_ *p, gp *g, next bool) {
    if ra
        n
    }
    if ne
        re
    o
    i
}

n
o
i
    n
    o
    i
    3 usages were filtered out
    Press ⌘⌘F7 again to search in Project Files
```

File	Function	Line Number	Usage Description
export_test.go	RunSchedLocalQueueTest	87	runqput(_p_, &gs[i], false)
export_test.go	RunSchedLocalQueueStealTest	108	runqput(p1, &gs[j], false)
export_test.go	RunSchedLocalQueueEmptyTest	157	runqput(p, &gs[0], next0)
export_test.go	RunSchedLocalQueueEmptyTest	169	runqput(p, &gs[1], next1)
proc.go	ready	690	runqput(g_.m.p.ptr(), gp, next)
proc.go	goyield_m	2823	runqput(pp, gp, false)
proc.go	newproc1	3544	runqput(_p_, newg, true)
proc.go	globrunqget	4945	runqput(_p_, gp1, false)

问题三：

对的。

## 42. 关于 debug 问题

### 问题

问题描述：

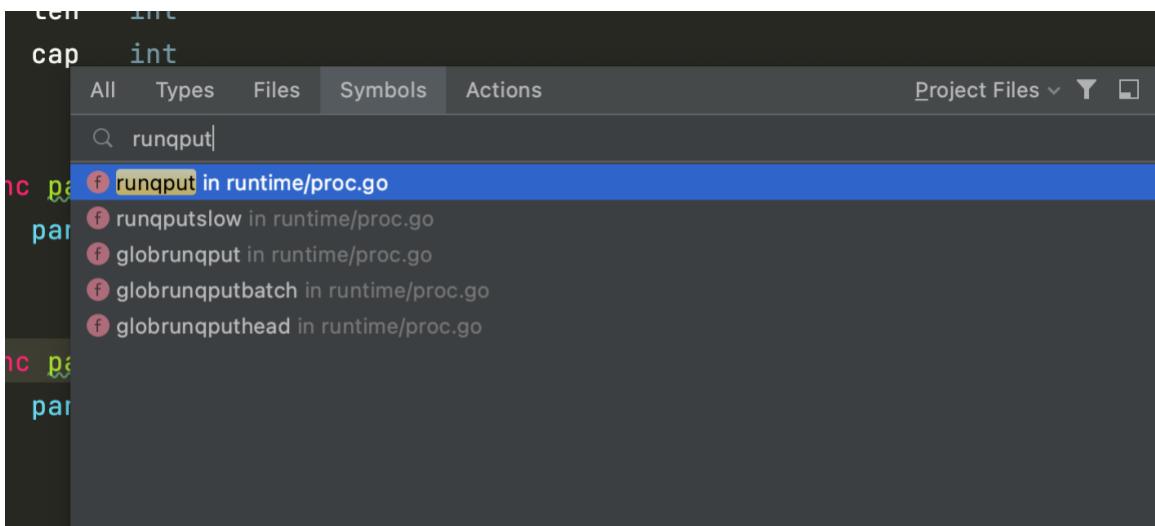
- 必做：runqput, runqget, globrunqput, globrunqget
- 选做：schedule, findRunnable, sysmon

如何通过 goland 找到上述方法调用的函数？runtime.gopark 是 runtime 可接管的阻塞，那么如何能够通过调试获得所有的调用方？

### 回答

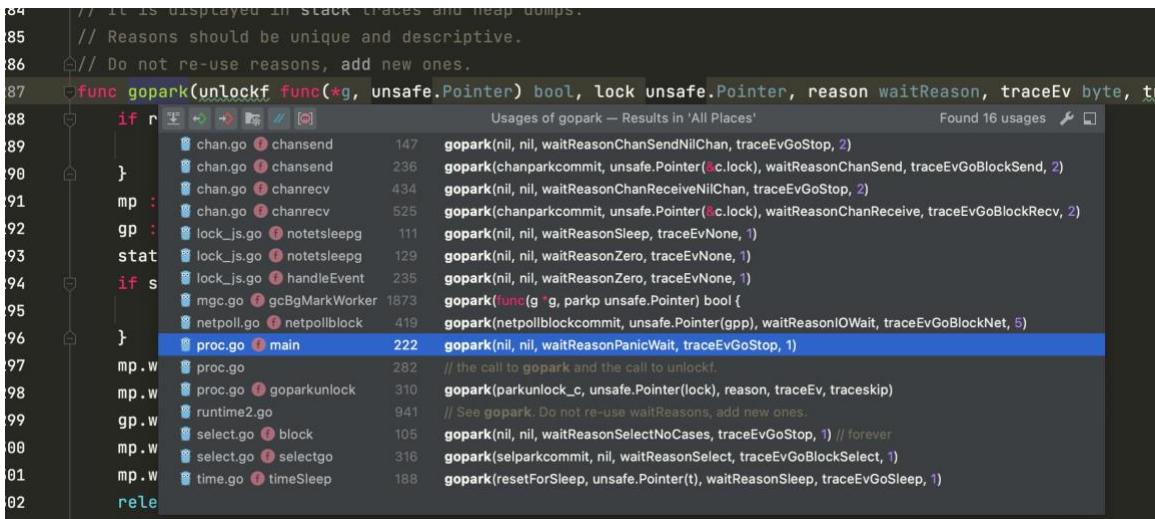
go env, 找到 GOROOT,  
用 goland 打开 GOROOT 目录，  
在 mac 下 command + option + o, 输入任意函数名, 如: runqput

# 一手资源+V 307570512



那么如何能够通过调试获得所有的调用方？

这里通过 IDE 的查找使用功能就可以了，按住 command 键，点击函数名



## 43. 关于不可接管阻塞问题补充

### 问题

- 1、对一些磁盘读写这种较长时间等待的阻塞属于不可接管阻塞吗
- 2、go 中对于处理 io 阻塞问题有用到 io 多路复用吗，如果有能列举哪些用到了吗。
- 3、对于大量不可接管阻塞 go 如何优雅的处理的
- 4、syscall 阻塞时 M 和 G 会锁定吗

# 一手资源+V 307570512

## 回答

1. 磁盘读写确实是不可接管的，在 runtime 里的 FD 区分 pollable 和非 pollable，网络的那种才是 pollable，可以接管，文件的不行
2. 你这个 io 多路复用我没理解。。。是说的网络协议里的多路复用么，一条连接上可以并发收发请求？类型 http2 那样的？
3. 没有太好的办法，我们线上的服务一般和文件系统打交道比较少，还好，如果碰上了的话，一般可能是固定的 goroutine 数消费 channel(内存里的 buffer) 去写文件，控制被阻塞的数量，不让线程数涨太高。但实际场景还是会有线程数爆炸的情况的，我们在线上碰到过，特别一些用 cgo 的。
4. syscall 阻塞的时候 M 和 G 是在一起的，不会分开。只会把 P 拿走

## 再次追问

曹大第二点我不知道说的对不对。在 linux 系统上对于 io 阻塞处理提供了五种 io 模型提供用户调用是吧。其中 io 多路复用这种模型应用在网络 io 中比较强大。我想知道除了用在网络 io 中还能用在其他 io 阻塞中吗。如果一个创建的一个系统线程只能处理一个 io 事件的话这样效率就会很低。想并发只能再开线程。这种结果很容易线程撑爆。那对于多个 io 阻塞问题复用一个线程这样子是不是就会更高效呢。

## 回答

网络 io，linux 系统给我们提供的是同步非阻塞接口，如果 os 提供的是阻塞接口(其实就是 syscall)，那在 Go 里面也处理不了，只能开线程。

开线程就得注意线程数

## [44. 关于不可接管阻塞问题](#)

## 问题

go 对于不可接管阻塞：

比如 syscall 这样的阻塞发生时，为什么是将 M 和 P 剥离，为什么不能做到 M 和 G 剥离。是因为

go 调度器没办法做到当前 g 发生 syscall 的不知道什么时候唤醒吗。还有就是 M 和 P 剥离后这个 M 后续怎么处理的是一直卡在那等待 syscall 返回码

## 回答

因为这时候阻塞位置的指令是 SYSCALL，其返回是由操作系统控制的。。Go 本身啥也做不了

剥掉 P 以后，只能等待 SYSCALL 指令返回。返回之后，M 会判断身上是否有 P，有的话继续执行，没有就把 G 塞进队列去。没返回之前只能等待。

## 45. 何时重新执行阻塞挂起的 G?

### 问题

M 执行调度循环的时候，何时会去执行之前阻塞挂起但现在已经 ready 的 G?

### 回答

M 不管这个，是由对称的操作方的 goroutine 去管理的，举个例子：

如果你的 g 阻塞在 channel send，那么他就进了 sendq，当前这个 M 就不管这个 g 了，之后 channel 有 recv 的 goroutine 执行 channel recv 操作的时候，会检查 channel 的 sendq 上是不是有被阻塞住的 g，由这个 recv 方把原来阻塞的 g 唤醒。

其它结构也是同理，如果是 Lock 阻塞的，那就由执行 Unlock 的 goroutine 把原来阻塞在锁上的 goroutine 唤醒。

## 46. M 执行调度循环时，何时重新执行挂起的 G

### 问题

课上说消费端 M 执行调度循环时会从绑定 P 的 runq 或 global runq 中获取 G，或者执行 work stealing。那么问题来了：

- 1、何时可以重新执行之前挂起但现在已经 ready 的 G，重新作为生产者提交吗？
- 2、还是会创建一个新的 M 来执行？如果其他 P 和 M 都正常绑定执行，何时能再次抢占到 P？
- 3、挂起的 G 重新 ready 后会如何分配给 P？

### 回答

1. 只要它 ready 了，就会进入三级队列中的一个，另一端就会被 m “消费”到，从而得以执

行。

2. 这个问题有点乱，可能没太想清楚。 $M$  是没  $P$  找  $P$ ，有  $P$  找  $G$  来执行。
3. 挂起的  $G$  ready 后会先进入三级队列，然后会分配给  $M$  来执行。就看哪个  $M$  来取了，没有特定的规律。 $M$  来取的时候肯定是绑定了一个  $P$  的。

## 47. 理解可执行文件的 ppt 页有段代码没解释

### 问题



上面这段是什么意思？

### 回答

这个是我用的命令行。。。运行了一下 ls 命令

## 48. 关于 GMP 调度模型的几个问题

### 问题

在内网上找到几篇关于 GMP 的文章，关于里面的一些描述我整理了一些疑问。（D 作为一大项问题）

虽然有些问题之前解答过，视频中也有，但我还是想再问问。。希望老师不要厌烦。。

**D1:** 当  $M$  执行某一个 goroutine 时如果发生了 syscall 或者其余阻塞操作， $M$  会阻塞，如果

# 一手资源+V 307570512

当前有一些 G 在执行, runtime 会把这个线程 M 从 P 中摘除, 然后寻找空闲线程 (创建新的线程) 来服务这个 P。

Q1: 这种说法 (模型) 是否正确?

Q2: 其余阻塞操作都有哪些?

Q3: 与因执行 syscall 而阻塞并从 P 上被摘除的 M 相关联的 G 在 M 发生阻塞时和阻塞结束后状态及所处位置 (逻辑位置) 的变化是怎样的。

**D2:** G1 运行时向已满 channel 中写入数据时进入 waiting, 此时从 M 上摘除 G1, 转而从相关联的 P 中的 runQ 队列内选择 runnable 的 G2 进行执行。

Q1: 与 **D1** 中的 syscall 相比较, 同样都是阻塞操作, 有什么不同?

Q2: 与 **D1** 相比较, 为什么 **D2** 不需要更换 M

在整理问题的过程中有些问题就已经得到了解决, 但还是打算问出来

老师回答地宽泛一点、偏向概念就可以

## 回答

D1: 当 M 执行某一个 goroutine 时如果发生了 syscall 或者其余阻塞操作, M 会阻塞, 如果当前有一些 G 在执行, runtime 会把这个线程 M 从 P 中摘除, 然后寻找空闲线程 (创建新的线程) 来服务这个 P。

Q1: 这种说法 (模型) 是否正确?

他这里说的比较简单, 我 ppt 里说的详细, 阻塞分为可接管可不可接管的, syscall 和 cgo 是不可接管的阻塞, 这种会占用住这个线程。需要把 P 和 M 剥离开, 然后找其它线程。那些可接管的阻塞就直接进各种等待队列了, 有人会说这叫 G 和 M 解绑。

Q2: 其余阻塞操作都有哪些?

ppt 上比较详细。。六种可接管阻塞和两种不可接管的。不过 cgo 一般用的少, 可以不用管。

Q3: 与因执行 syscall 而阻塞并从 P 上被摘除的 M 相关联的 G 在 M 发生阻塞时和阻塞结束后状态及所处位置 (逻辑位置) 的变化是怎样的。

阻塞结束后, 相当于 syscall 返回, 这时候 M 要检查自己是不是还和 P 绑定, 如果还绑定, 那就继续执行 syscall 返回的逻辑, 否则就要让这个 G 进全局队列(因为这个 M 已经没有 P 了, 所以没有 local run queue, 没有 runnext), 这部分代码你感兴趣可以看看 entersyscall 和 exitsyscall

## 再次追问

曹大, 这里所说的“那些可接管的阻塞就直接进各种等待队列”, 这里的等待队列是指全局队列还是本地队列又或是一个单独的队列呢, 在源码中没有找到这部分。。。

## 回答

这个不是调度队列, 比如 channel, 就是 sendq 和 recvq

# 一手资源+V 307570512

这个等待队列是和具体的数据结构相关的

D2: G1 运行时向已满 channel 中写入数据时进入 waiting, 此时从 M 上摘除 G1, 转而从相关联的 P 中的 runQ 队列内选择 runnable 的 G2 进行执行。

Q1: 与 D1 中的 syscall 相比较, 同样都是阻塞操作, 有什么不同?

ppt 里已经说了。。这种是可以接管的, 因为原来在用户视角看来被阻塞的 g, 在 runtime 里是塞进某种等待队列里了。syscall 操作要和操作系统交互, 必须阻塞住线程等操作系统返回才能继续, 所以必须占用线程。

Q2: 与 D1 相比较, 为什么 D2 不需要更换 M

因为 M 这时候不需要被占用, 自然不需要更换了

## 49. runnext 里面的 G 为什么不是放在本地队列

### 问题

根据局部原理, 为什么把 runnext 里面的 G 一起放在全局队列里面? 理论上来说 runnext 里面的 G 放在本地队列不是更符合局部性原理么

### 回答

我个人的猜测:

1. 局部队列满的时候, 说明系统整体的压力已经比较大了, 至少是当前的 p 压力很大
2. 这时候把踢出的 g 放进全局队列, 可能有更多的机会被其它 p 执行, 如果待在本地队列, 可能被前面的慢任务堵着  
不过真实的原因只能去问问官方了

### 再次追问

第 2 种解释还不是很明白, 如果是放在全局队列中, 为什么有更多机会被其它 P 执行了?  
其他 P 如果空闲, 执行 work stealing 也是从其他 P 的 local queue 队头偷, 放在 local 更容易被偷走执行; 放在全局中唯一被执行的可能性是其他 P 执行了 6 1 循环到全局中去拿一次而已

### 回答

work stealing 的优先级在从全局队列里拿之后呢,

这个课程的动画应该画的挺清楚的, 你需要手动点点玩玩

## 50. 调度相关：协作式调度和抢占式调度的实现和发生的时机是怎样的？

### 问题

协作式调度和抢占式调度的实现和发生的时机是怎样的？还请老师帮忙解答一下。

### 回答

先从概念上理解：

- 协作式调度依靠被调度方主动弃权；
- 抢占式调度则依靠调度器强制将被调度方中断。

协作式调度发生的时机是在函数序言部分的扩栈检测指令，当检测到 stackGuard0 是一个特定值 stackPreempt 的时候，就会主动放弃 CPU 执行权。用编译命令 go tool compile -S 可以看到比较指令：

#### 主动调度弃权

The diagram shows a vertical stack from low address at the bottom to high address at the top. It is divided into several regions: stack.hi (light green), main (purple), stackguard0 (light green), StackSmall (dotted line), stack.lo (light green), and StackGuard (dotted line). An arrow labeled "栈增长方向" points downwards through the stack. Labels <- stack.hi, main, <- stackguard0, StackSmall, <- stack.lo, and StackGuard are placed near their respective regions.

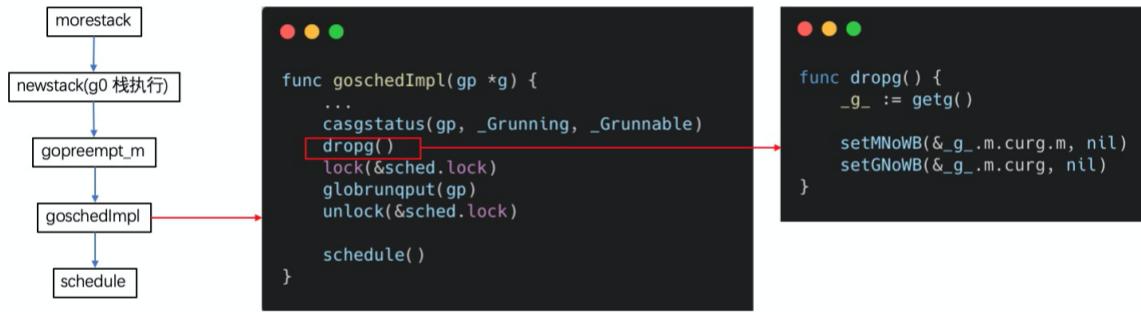
The terminal shows the command `go tool compile -S main.go`. The assembly output includes instructions like `XORPS X0, X0`, `MOVUPS X0, """,~r0+16(SP)`, and `JMP os.(*file).close(SB)`. A red box highlights the `MOVQ (TLS), CX` instruction. The source code on the right defines the `g` and `stack` structures with `stackguard0` and `stackguard1` fields.

```
0x000d 00013 (<autogenerated>:1)    XORPS  X0, X0
0x0010 00016 (<autogenerated>:1)    MOVUPS  X0, """",~r0+16(SP)
0x0015 00021 (<autogenerated>:1)    JMP    os.(*file).close(SB)
0x0009 48 8b 44 24 08 48 8b 00 48 89 44 24 08 0f 57 c0 H.D$H..H.D$..W.
0x0010 0f 11 44 24 10 e9 00 00 00 00 ..0$.....
rel 22+4 t=8 os.(*file).close+0
...main STEXT size=138 args=0x0 locals=0x58
0x0000 00000 (main.go:5)      TEXT   ""_.main(SR)  ABIInternal, $88-0
0x0001 00000 (main.go:5)      MOVQ   (TLS), CX
0x0009 00009 (main.go:5)      CMPQ   SP, 16(CX)
0x0009 00013 (main.go:5)      PCDATA $0, $-2
0x0009 00013 (main.go:5)      JLS    128
0x000f 00015 (main.go:5)      PCDATA $1, $-1
0x000f 00015 (main.go:5)      SUBQ   $0, SP
0x0013 00019 (main.go:5)      MOVO   BP, 00(SP)
0x0018 00024 (main.go:5)      LEAD   $0, BP
0x001d 00029 (main.go:5)      FUNCDATA $0, glocals:33cdcccccebe80329f1fdbee7f5
874cb(SB)
0x001d 00029 (main.go:5)      FUNCDATA $1, glocals:f20267fb96a0178e8758c6e3e
0ce28(SB)
```

0x0080 00128 (main.go:5) CALL runtime.morestack\_noctxt(SB)|

接着就会跳到 morestack 执行。最终会执行到 goschedImpl，将 g 和 m 解绑，将 g 放到全局可运行队列里去。而 m 这时又会去找其他的 g 来执行。

## 主动调度弃权



- 必须要有函数调用且有扩栈检测指令才有机会执行抢占
- 如果执行的是 `i++` 这种就会一直无限循环下去，无法抢占
- Go 1.14 实现了基于信号的抢占

主动调度弃权依赖在函数序言部分插入抢占检测指令，需要有函数调用。有些场景下，我们写出一个 `for {i++}` 这样的无限循环，就没法抢占了。这在 Go 1.14 之前会出现死机的事故。Go 1.14 实现了信号抢占，就可以解决这种场景。

预告一下，曹大在今年的 gopherchina 上会讲信号抢占这个 topic。

信号抢占的时机是：

1. 当 goroutine 执行时间过长，超过 10ms，sysmon 会检测到，然后向这个 goroutine 所在的 m 发 SIGURG 信号。
2. GC 时会停止所有正在运行的 goroutine，这时也会向 m 发 SIGURG 信号。

m 收到信号后，会转去执行 sighandler 函数，sighandler 函数会做一些手脚（插入了一个抢占函数：`asyncPreempt`）就马上返回。返回之后，m 不会接着执行 goroutine 的指令了，而是去执行 `asyncPreempt` 去了，抢占就在这里发生：把当前的 goroutine 扔到全局可运行队列里，m 则去找其他的 goroutine 执行。

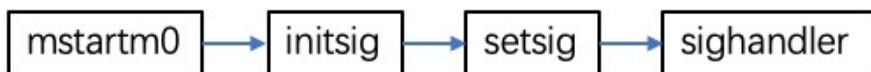
函数调用流程如下：

# 一手资源+V 307570512

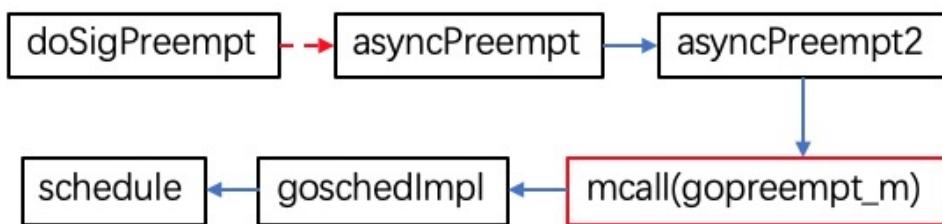
发出抢占信号：



注册抢占信号：



处理抢占信号：



关于这个话题，推荐一篇文章，写得非常好，值得看 10 遍：

<https://golang.design/under-the-hood/zh-cn/part2runtime/ch06sched/preemption/>

51. 源码中找不到函数体~~~

问题

# 一手资源+V 307570512

```
i64.s
 25 MOVD    $runtime·tls_g($B), R2
 26 MOVD    R2, 8(RSP)           // arg1: &tls_g
 27 BL    .tlsinit($B)
 28 ADD    $32, RSP
 29 #endif
 30
 31 // create istack out of the given (operating system) stack.
 32 // _cgo_init may update stackguard.
 33 MOVD    $runtime·g0($B), g
 34 MOVD    RSP, R7
 35 MOVD    $(-64*1024)(R7), R0
 36 MOVD    R0, g_stackguard0(g)
 37 MOVD    R0, g_stackguard1(g)
 38 MOVD    R0, (g_stack+stack_lo)(g)
 39 MOVD    R7, (g_stack+stack_hi)(g)
 40
 41 // if there is a _cgo_init, call it using the gcc ABI.
 42 MOVD    _cgo_init($B), R12
 43 CBZ   R12, nocgo
 44
 45 #ifdef GOOS_android
 46     MRS_TPIDR_R0          // Load TLS base pointer
 47     MOVD    R0, R3           // arg 3: TLS base pointer
 48     MOVD    $runtime·tls_g($B), R2 // arg 2: &tls_g
 49 #else
 50     MOVD    $0, R2           // arg 2: not used when using platform's T
 51 #endif
 52     MOVD    $setg_gcc$($B), R1 // arg 1: setg
 53     MOVD    g, R0             // arg 0: G
```

老师你好，像这种 g0() 我找不到函数内容，还有比如，proc.go 内

Plain Text

g := getg()

这种我也找不带函数体，，该怎么理解？

## 回答

\$runtime.g0(\$B)，这个是汇编代码，指的是 runtime 代码里那个 g0 的全局变量，不是一个函数

getg 是个比较特殊的东西，看这里：<https://class.imooc.com/course/qadetail/289245>

## 52. 一个很简单但是又不太好理解的问题

### 问题

以下是在 Linux 里运行一个 web 服务器的过程：

- 1、go build main.go 生成 main 二进制文件
- 2、发布系统将这个 main 二机制文件 push 到物理机器并执行类似于./main
- 3、至此，一个简易的 web 服务已经启动，g0 也创建完成，等待用户请求

问题：

- 1、所有的用户请求都是落到这个机器，整个系统里只有一个 g0 吗
- 2、假设现在只有两个请求进来，用户 A 和用户 B，假设他们运行期间只产生一个 G，Ga1 和 Ab1，这 2 个 G 直接有联系吗
- 3、Ga1 和 Gb1 他们会去竞争某些公共资源吗
- 4、我的理解是 Ga1 和 Gb1 分属于 2 个不同的 goroutine，之间应该没啥联系，但是看很多源码或者开源项目里，sync.Mutex 经常用来锁住某些操作，如果没有区别，为啥需要锁住，这里就开始矛盾了
- 5、锁的意义，是针对一个用户产生的多个 goroutine，还是针对不同用户产生的 goroutine，其实这里是被 redis 这种第三方锁干扰了，redis 锁是针对不同的用户来讲的

### 回答

1. 每个线程 m 都有自己的 g0。runtime.g0 是和 runtime.m0 绑定的 g0，这个 g0 是 runtime 的一个全局变量，整个系统只有一个；其它普通线程，也有自己的 g0，**g0 主要负责执行调度相关的代码**(用户代码在普通 g 上执行)。
2. 这个你要看看 Go 的 tcp server 模型，简单来说：

Plain Text

```
for {
    c, err := listener.Accept()
    // handle connection
    go func() {
        // 用户的连接处理逻辑在各自的 goroutine 里，两者没有任何关系
    }()
}
```

3. 这还是有可能的，举个例子，假如这是个在线游戏服务，Ga 和 Gb 在同一个房间都对 boss 进行了攻击，也就是需要修改 boss 的血量，那在修改的时候就必须加锁(或者用 atomic 操作)，这就是一种资源竞争，因为他们在同一个房间内打同一个 boss
4. 锁是用来保护公共资源的，大家如果都只是读，那就不用加锁，一旦多个用户对同一个资源有并发的读/写，那就需要使用同步库里的操作进行同步

## 再次追问

1.其实我含糊的就是曹大回答的第四点里的几个关键字，“多个用户”和“同一资源”，这里的多个用户是指同一个用户 a 在代码中通过 go 关键字创建的多个 goroutine，比如 a1-a2-a3，还是就是用户 a 和用户 b 2.其次，同一资源，怪物的血量可以理解成全局共享变量或者资源，操作理应加锁，我看很多来源项目，很多 struct 里也有一个 sync.Mutex 的字段，一个 struct 定义的对象，也能是公共资源吗 3.如果能理解“谁跟谁”去操作“同一资源”，那么这个问题就 ok 了，还麻烦老师帮忙解答下

## 回答

一个 struct 也可能被共享呀， net.conn，这一条连接可能被多个 goroutine 并发写呢

## 53. 生产模型和消费模型的问题

### 问题

local run queue 中的 256 和 GMP 中的 P(数量最大为 256)是指同一个吗？ schedtick 执行到 61 先去找 runnext 还是 global queue?

### 回答

P 数量最大 256 这个说法你是在哪里看到的？自己本地想实验的话你可以这样  
GOMAXPROCS=512 go run hello.go

local run queue 是一个 256 大小的数组，但是有 head 和 tail 指针，使用上是个环形数组

schedtick 执行到 61 的时候，先去 global，这里的逻辑我动画中应该体现出来了

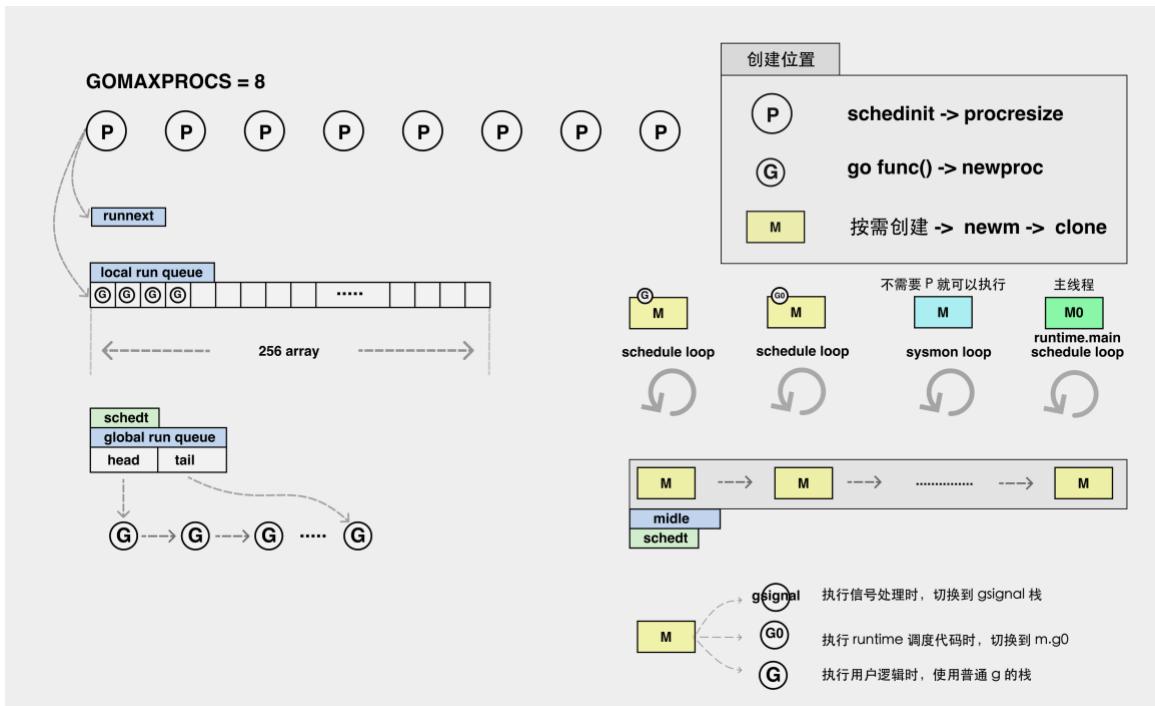
## 54. 怎么理解 g0 和 m0，和生产消费的 g、m 有什么不同

### 问题

怎么理解 g0 和 m0，和曹大在课程上说的生产和消费的 g、m 有什么不同

### 回答

再补充一句，m0 因为执行的是主函数，所以它退出的时候有一些特殊逻辑  
比如你的 main 函数退出了，那它相关的资源什么的也需要退出。



g0 是 M 在执行调度指令时使用的 g，栈使用了线程创建的时候默认的线程栈  
m0 在图上有了解释

#### update 2021.06.20:

m0 是程序启动时操作系统自动给我们创建好的第一个线程，一般也被称为主线程。  
在 Go 里，这个线程也是参与调度的，之前画的图有点问题~

## 55. Work stealing 相关

### 问题

Work stealing 是只做一次，还是如果没偷到，会再去另外的 P 上偷？如果一直偷不到是不是就到 ppt 里的那个 stop 部分了？

目前水平不够，看视频梳理还算凑合，但没有自己追踪源码的能力

### 回答

对的，要检查所有的 P，一直偷不到就进 stop 部分

## 56. init()在哪个启动环节执行？

# 一手资源+V 307570512

## 问题

init()在哪个环节执行？

Go 启动过程从代码结果上大概是 全局变量->init()->main(),  
全局变量和 init()是不是都在 runtime.schedinit 这个过程执行?  
不熟悉看源码，暂时没有自己去确认。

## 回答

全局变量由编译器分配内存，不需要执行 runtime 代码。

init 是在 runtime 包中的 proc.go 文件中的 main 函数中执行的。执行完 init 函数后，再执行用户写的 main 函数。

这里有 2 篇参考文章，与最新版的 Go 源码有所区别，不过过程是类似的。

<https://blog.csdn.net/whh995496580/article/details/53423215>

这篇是雨痕的《Go 语言学习笔记》，可以直接看原书。

<https://golang.design/under-the-hood/zh-cn/part1basic/ch05life/main/>

这是欧神的一篇文章，可以参考~

## 57. Go1.14 带来了哪 3 个涉及本堂课程的主要变化？

## 问题

Go1.14 带来了哪 3 个涉及本堂课程的主要变化？

1. time.Sleep()不再另起 Goroutine，而是在当前 P 下安全调度，无需发生 M/P 的切换
2. 支持基于信号的异步抢占式调度
3. 增加 runnext，最后启动的 Goroutine 最先执行 (?)

问题：

1. 是不是以上三项？查阅了一些资料，前两项有说明，第三项存疑。
2. 课件中“一个无聊的输出顺序问题”，1.14 及以后，是不是应该这样解释？

左图，Channel 阻塞，抢占 P，开始执行 runnext，优先输出 9，后 0-8，所有线程 hang 住，程序异常退出。

右图，time.Sleep()不再另起 Goroutine，那运行时应该：抢占 P，开始执行 runnext，优先输出 9，后 0-8，后超时，平滑退出。

## 回答

1-3 里，前两项没问题，第三项不对，runnext 一直都有，不是 1.14 加的

# 一手资源+V 307570512

你说的结果也没问题~

## 再次追问

谢谢曹大，那请问您在课程和解答中提到的“3个变化”的另外一个是什么呢？

## 回答

我印象中 1.14 涉及调度的变化就是信号式抢占和 timer 的变更  
没第三点了吧~

## 58. 找不到 Dockerfile

## 问题

### 理解可执行文件

本节课涉及的工具都准备在 Dockerfile 里了，大家可以自行实验

```
FROM centos
RUN yum install golang -y \
    && yum install dlv -y \
    && yum install binutils -y \
    && yum install vim -y \
    && yum install gdb -y
```

```
docker build -t test .
```

```
docker run -it --rm test bash
```

图片中的 Dockerfile 在哪里可以找到？

## 回答

Dockerfile 文件的内容就是

```
Plain Text
FROM centos
RUN yum install golang -y \
```

# 一手资源+V 307570512

```
&& yum install dlv -y \
&& yum install binutils -y \
&& yum install vim -y \
&& yum install gdb -y
```

建立这个文件后，运行

```
Plain Text
docker build . -t test
```

这个命令会，会找当前目录下的 Dockerfile 的文件，你也可以通过-f 指定类似

```
Plain Text
docker build . -f Dockerfile -t test
```

经过 docker build 后就会创建 docker 镜像，docker image 就能看到，

ps: 这属于 docker 的基础使用，网上搜索下，就有一堆的。可以看看网上的，

```
total 8
-rw-r--r-- 1 Lee  staff  155B  5 16 11:58 Dockerfile
+ devtool cat Dockerfile
FROM centos
RUN yum install golang -y \
    && yum install dlv -y \
    && yum install binutils -y \
    && yum install vim -y \
    && yum install gdb -y
+ devtool
```

```
Plain Text
FROM centos
RUN yum install golang -y \
    && yum install dlv -y \
    && yum install binutils -y \
    && yum install vim -y \
    && yum install gdb -y
```

自己新建文件就可以了

59. runqget 中返回的 inheritTime 具体有什么作用呢？

## 问题

runqget 会返回一个 inheritTime 值，看源码知道这个值跟 p.schedtick 有关，但是还是不理

# 一手资源+V 307570512

解为什么 runqget 需要返回这个值。跟时间片又有什么关系呢？

Plain Text

```
// If inheritTime is true, gp inherits the remaining time in the
// current time slice. Otherwise, it starts a new time slice.
// Never returns.
func runqget(_p_ *p) (gp *g, inheritTime bool) {
    // If there's a runnext, it's the next G to run.
    for {
        next := _p_.runnext
        if next == 0 {
            break
        }
        if _p_.runnext.cas(next, 0) {
            return next.ptr(), true
        }
    }

    for {
        h := atomic.LoadAcq(&_p_.runqhead) // load-acquire, synchronize with other
consumers
        t := _p_.runqtail
        if t == h {
            return nil, false
        }
        gp := _p_.runq[h%uint32(len(_p_.runq))].ptr()
        if atomic.CasRel(&_p_.runqhead, h, h+1) { // cas-release, commits consume
            return gp, false
        }
    }
}
```

## 回答

结论: runnext 与主协程共享时间片, 防止两者相互阻塞-唤醒导致其他协程饥饿.

我看了下源码, 如果调度的时候, 获取了 runnext, 而不是其他队列的 g, 那就继承时间片, 其他情况基本都是不继承.

# 一手资源+V 307570512

```
// Executed only by the owner P.
func runqget(_p_ *p) (gp *g, inheritTime bool) {
    // If there's a runnext, it's the next G to run.
    for {
        next := _p_.runnext
        if next == 0 {
            break
        }
        if _p_.runnext.cas(next, new: 0) {
            return next.ptr(), inheritTime: true
        }
    }

    for {
        h := atomic.LoadAcq(&_p_.runqhead) // load-acquire, synchronize with other consumers
        t := _p_.runqtail
        if t == h : nil, false ↴
        gp := _p_.runq[h%uint32(len(_p_.runq))].ptr()
        if atomic.CasRel(&_p_.runqhead, h, h+1) { // cas-release, commits consume
            return gp, inheritTime: false
        }
    }
}
```

在接下来的 execute 方法中, 可以看到. 如果不继承, 就 schedtick++, 继承的话就不变.

```
//go:yeswritebarrierrec
func execute(gp *g, inheritTime bool) {
    _g_ := getg()

    // Assign gp.m before entering _Grunning so running Gs have an
    // M.
    _g_.m.curg = gp
    gp.m = _g_.m
    casgstatus(gp, _Grunnable, _Grunning)
    gp.waitsince = 0
    gp.preempt = false
    gp.stackguard0 = gp.stack.lo + _StackGuard
    if !inheritTime {
        _g_.m.p.ptr().schedtick++
    }
}
```

这样的话, sysmon 里根据 schedtick 来计算协程有没有运行太久. 如果 schedtick 变了, 那表示开启新一轮计算. 没变的话, 表示复用一个时间片.

# 一手资源+V 307570512

```
func retake(now int64) uint32 {
    n := 0
    // Prevent allp slice changes. This lock will be completely
    // uncontended unless we're already stopping the world.
    lock(&allpLock)
    // We can't use a range loop over allp because we may
    // temporarily drop the allpLock. Hence, we need to re-fetch
    // allp each time around the loop.
    for i := 0; i < len(allp); i++ {
        _p_ := allp[i]
        if _p_ == nil {
            // This can happen if procslice has grown
            // allp but not yet created new Ps.
            continue
        }
        pd := &_p_.sysmontick
        s := _p_.status
        sysretake := false
        if s == _Prunning || s == _Pssyscall {
            // Preempt G if it's running for too long.
            t := int64(_p_.schedtick)
            if int64(pd.schedtick) != t {
                pd.schedtick = uint32(t)
                pd.schedwhen = now
            } else if pd.schedwhen+forcePreemptNS <= now {
                preemptone(_p_)
                // In case of syscall, preemptone() doesn't
                // work, because there is no M wired to P.
                sysretake = true
            }
        }
    }
}
```

这样看起来像可以防止一个协程不停的启动/等待子协程, 然后子协程运行, 又复活父协程。(没有父子关系, 大概这个意思).

所以我去翻了下这个 commit, 还比较难翻到. 基本我就是我说的这个意思.

<https://go-review.googlesource.com/c/go/+/9289/>

runtime: yield time slice to most recently readied G

这个是和 runnext 一起优化.

1. runnext 可以加快上面说的运行模型

2. 共享时间窗口, 又能避免因为不停的 ping-pong 双向唤醒导致其他协程饥饿

一手资源+V 307570512

60. go\_tls.h 里的 g(r)是干什么的

问题

一手资源+V 307570512

amd64

.s(r)

s(r)

# 一手资源+V 307570512

在 runtime.gosave, runtime.gogo 里都碰到了，不太懂它的作用。

## 回答

这是个特殊的命令。。从 get\_g 开始一路替换，参考一下这个帖子：  
<https://class.imooc.com/course/qadetail/289245>

## 61. 无缓存 Channel 在只有写入没有读取的情况下为什么会死锁

### 问题

出于一些原因还没有听课，如果问的问题重复了请见谅。

相关代码：

```
Plain Text
package main

import "fmt"

func main() {
ch := make(chan int)
ch <- 1

fmt.Println("hello word")
}
//fatal error: all goroutines are asleep - deadlock!
```

翻看源码的时候有一部分代码因为各种原因看不懂，包括且不限于

```
getg
acquireSudog
gopark
acquirem
KeepAlive
```

担心自己不成熟的想法带偏老师，干脆就直接问个大问题“无缓存 Channel 在只有写入没有读取的情况下为什么会死锁，请从源码角度给予解答”

相关代码：

```
Plain Text
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
.....
//至少我想弄明白这个锁在无缓存的情况下是怎么释放的
lock(&c.lock)
.....
gopark(chanparkcommit, unsafe.Pointer(&c.lock), waitReasonChanSend,
traceEvGoBlockSend, 2)
```

# 一手资源+V 307570512

.....  
}

提前谢谢老师

PS：老师给文章的话最好是内网的==

## 回答

checkdead 是检查是不是所有的线程都被阻塞住了，你现在只有一个主线程，被阻塞住整个程序就没法继续执行了，所以就跪了，如果你启动一个额外的 g 就不会挂

getg，这个命令非常特殊，参考一下这个 <https://class.imooc.com/course/qadetail/289245>  
acquireSudog，这个 acquireSudog 和 releaseSudog，分别对应获取一个 sudog 和释放一个  
sudog 的缓存，因为 sudog 是频繁分配/释放的结构，runtime 里对这种类型的结构专门做了  
缓存优化

gopark，课上其实讲到了，如果可以被 runtime 接管的阻塞，最终都是走到这个函数，把  
要阻塞的 g 挂起在某个等待结构上，也就是 ppt 里的六个例子

acquirem，这个是获取一个线程，应该逻辑挺简单的

KeepAlive，这个初学者可以不用关心，是和 GC 有关的一个函数

62. 同样的程序，在 linux 上面的结果符合消费端结论，但  
**windows 好像不符合？**

## 问题

同样的程序，在 linux 上面和 windows 上面表现不一致？linux 结果符合消费端得结论，  
windows 好像不符合。

代码如下：

```
Plain Text
func main() {
    runtime.GOMAXPROCS(1)
    for i := 1; i < 259; i++ {
        i := i
        go func() {
            fmt.Printf("%v;", i)
        }()
    }
    //time.Sleep(time.Hour)
    var ch = make(chan int)
    <-ch
}
```

# 一手资源+V 307570512

}

linux 输出结果:

Plain Text

```
258;129;130;131;132;133;134;135;136;137;138;139;140;141;142;143;144;145;146;147;148;149;  
150;151;152;153;154;155;156;157;158;159;160;161;162;163;164;165;166;167;168;169;170;171;  
172;173;174;175;176;177;178;179;180;181;182;183;184;185;186;187;188;1;189;190;191;192;19  
3;194;195;196;197;198;199;200;201;202;203;204;205;206;207;208;209;210;211;212;213;214;21  
5;216;217;218;219;220;221;222;223;224;225;226;227;228;229;230;231;232;233;234;235;236;23  
7;238;239;240;241;242;243;244;245;246;247;248;2;249;250;251;252;253;254;255;256;3;4;5;6;7;  
8;9;10;11;12;13;14;15;16;17;18;19;20;21;22;23;24;25;26;27;28;29;30;31;32;33;34;35;36;37;38;3  
9;40;41;42;43;44;45;46;47;48;49;50;51;52;53;54;55;56;57;58;59;60;61;62;63;64;65;66;67;68;69;  
70;71;72;73;74;75;76;77;78;79;80;81;82;83;84;85;86;87;88;89;90;91;92;93;94;95;96;97;98;99;1  
00;101;102;103;104;105;106;107;108;109;110;111;112;113;114;115;116;117;118;119;120;121;12  
2;123;124;125;126;127;128;257;
```

windows 输出结果:

Plain Text

```
258;129;130;131;132;133;134;135;136;137;138;139;140;141;142;143;144;145;146;147;148;149;  
150;151;152;153;154;155;156;157;158;159;160;161;162;163;164;165;166;167;168;169;170;171;  
172;173;174;175;176;177;178;179;180;181;182;183;184;185;186;187;188;189;190;191;192;193;  
194;195;196;197;198;199;200;201;202;203;204;205;206;207;208;209;210;211;212;213;214;215;  
216;217;218;219;220;221;222;223;224;225;226;227;228;229;230;231;232;233;234;235;236;237;  
238;239;240;241;242;243;244;245;246;247;248;249;250;251;252;253;254;255;256;1;2;3;4;5;6;7;  
8;9;10;11;12;13;14;15;16;116;17;18;19;20;21;22;23;24;25;26;27;28;29;30;31;32;33;34;35;36;37;  
38;39;40;41;42;43;44;45;46;47;48;49;50;51;52;53;54;55;56;57;58;59;60;61;62;63;64;65;66;67;6  
8;69;70;71;72;73;74;75;76;117;77;78;79;80;81;82;83;84;85;86;87;88;89;90;91;92;93;94;95;96;9  
7;98;99;100;101;102;103;104;105;106;107;108;109;110;111;112;113;114;115;116;117;118;119;120;121;1  
22;123;124;125;126;127;128;257;
```

## 回答

那个问题主要是为了说明 runnext 和 time.Sleep 的特征，知道这里有这个优先级的概念就可以了~

你这里创建的 g 已经超过本地队列的长度了，课上我们看到 生产和消费流程都会有调整队列的情况，而且这 200 多个有点考眼神儿。。。

平常我们线上也不太可能只跑一个 P，要想靠逻辑推断出顺序意义不是很大的。。。

## 63. gmp 与 os

## 问题

请教下：

# 一手资源+V 307570512

1. m 从 P 中找 g 的时候，是已经拿到时间片了吗？
2. m 和 linux 中的 task\_struct 是什么关系？

## 回答

1. 对的。m 在找 g，说明线程在执行（你可以把 m 看做线程）
2. task\_struct 表示线程，m 其实是线程的一个 tls 变量。

## 64. runnext 的抢占是如何发生的

## 问题

```
func runqput(_p_ *p, gp *g, next bool) {
    if randomizeScheduler && next && fastrand()%2 == 0 {
        next = false
    }

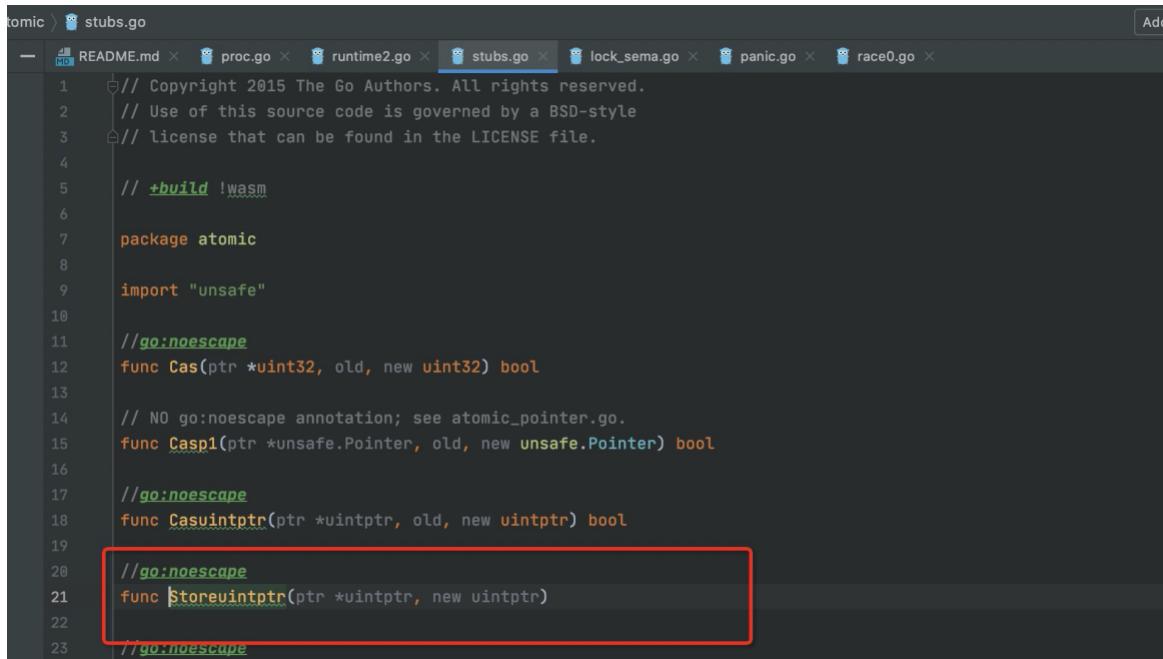
    if next {
        retryNext:
        oldnext := _p_.runnext
        if !_p_.runnext.cas(oldnext, guintptr(unsafe.Pointer(gp))) {
            goto retryNext
        }
        if oldnext == 0 {
            return
        }
        // Kick the old runnext out to the regular run queue.
        gp = oldnext.ptr()
    }
}
```

这个 cas 指令是怎么做到：

- 1、返回 true 代表新的 g 抢占成功
  - 2、返回 false 代表等待正在执行的 g 的呢
- 是什么条件判断的抢占成功呢？

还就就是想看这个 cas 的详细内容点进去是个没有实现的函数，如下图

# 一手资源+V 307570512



```
tomic > stubs.go
  └─ README.md × proc.go × runtime2.go × stubs.go × lock_sema.go × panic.go × race0.go ×
1 // Copyright 2015 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build !wasm
6
7 package atomic
8
9 import "unsafe"
10
11 //go:nosescape
12 func Cas(ptr *uint32, old, new uint32) bool
13
14 // NO go:nosescape annotation; see atomic_pointer.go.
15 func Cas1(ptr *unsafe.Pointer, old, new unsafe.Pointer) bool
16
17 //go:nosescape
18 func Casuintptr(ptr *uintptr, old, new uintptr) bool
19
20 //go:nosescape
21 func Storeuintptr(ptr *uintptr, new uintptr)
22
23 //go:nosescape
```

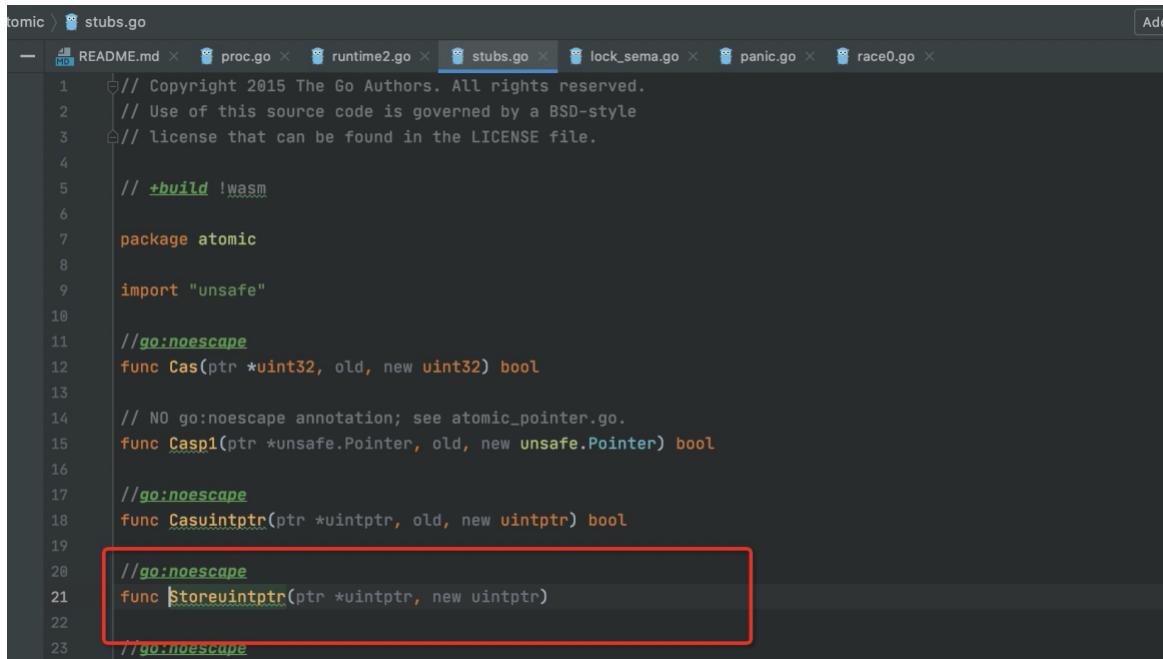
## 回答

runnext 只是 P 结构体的一个字段，替换就是就是把 old 值保存下来，new 值塞进去就行了。。这里不叫抢占，抢占是要把正在执行的那些 goroutine 停下来

## 再次追问

不是还有个等待老的 g 执行完再抢占的逻辑么。要是单纯 CAS 没有什么判断逻辑的话，理论上每次都能成功吧  
咋看这种没有实现的代码呢

# 一手资源+V 307570512



```
tomic > 🐿 stubs.go
- README.md × proc.go × runtime2.go × stubs.go × lock_sema.go × panic.go × race0.go ×
1 // Copyright 2015 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build !wasm
6
7 package atomic
8
9 import "unsafe"
10
11 //go:nosescape
12 func Cas(ptr *uint32, old, new uint32) bool
13
14 // NO go:nosescape annotation; see atomic_pointer.go.
15 func Cas1(ptr *unsafe.Pointer, old, new unsafe.Pointer) bool
16
17 //go:nosescape
18 func Casuintptr(ptr *uintptr, old, new uintptr) bool
19
20 //go:nosescape
21 func Storeuintptr(ptr *uintptr, new uintptr)
22
23 //go:nosescape
```

## 回答

等待老的执行完，这时候是线程进 schedule 调度循环，会检查自己绑定的那个 P 有没有 runnnext，有的话就执行了，这个不是抢占逻辑，就是调度循环里的正常 runqget 的逻辑。

## 65. dlv 怎么跟踪源码

## 问题

曹大，dlv 怎么跟踪 go 源码？比如说我怎么才走到 main.main，怎么才走到 runqget 函数这样子，我现在不知道为什么就这么给这些函数打断点，是我先看源码，，看到这些函数然后去打断点吗？还是我一步步跟着源码，，看到这些函数再打断点

```
Plain Text
(dlv) b runtime.runqget
(dlv) b runtime.globrunqget
(dlv) b runtime.runqput
(dlv) b runtime.globrunqput
(dlv) b main.main
```

## 回答

可以用 continue 命令(简写是 c)

快速跳到下一个断点

## 66. 关于线程和协程

### 问题

曹大，怎么通俗易懂理解线程和协程，网上搜一堆说的不明不白

### 回答

如果只是想理解的话，`g` 就是一个计算任务

用户代码里写：

```
```  
go func() {  
    xxxx  
}()  
```
```

的时候，是希望 runtime 把这个 go func 里面的代码执行完。任务在执行的过程中需要一些资源，资源可以分两种：

1. 栈，我的 goroutine 里发生函数调用，需要给被调用的函数分配栈空间(stack frame)，这部分分配工作是由 runtime 管理的
2. 堆，在任务执行的过程中，有一些变量需要被分配到堆上，这部分工作也是 runtime 管理的

`m` 是实际的线程，和其它语言的线程完全一样，他主要的职责不断执行调度循环，这个循环的任务就是从队列中取出用户提交过来的 `g`，并且执行

概念上这样就可以理解了，实际上很多人还会比较线程和 goroutine 的资源开销以及切换成本，我个人觉得这没啥意义。。不过还是列在 ppt 上了

## 67. coding 的 demo 代码库 go install 在不同版本的用法

### 问题

在 coding 的代码库中有一个 demo 的程序，其中用了 embed 的方式启动一个 web 项目。可以通过 `go instasll` 命令 build 和 install。

但是我发现 go 1.16 后可以直接执行，甚至不需要 download 代码。

Plain Text

```
go install github.com/gotomicro/embedctl@latest
```

而在 go 1.16 以前，需要这样才能 install，对吧？

```
git clone https://github.com/gotomicro/embedctl
```

```
cd embedctl
```

# 一手资源+V 307570512

go install github.com/gotomicro/embedctl

这是在 go install 有什么新的改动吗？

## 回答

我对工具链更新关注的不多哈哈，1.16 只关注到了 go mod 的一些变化，倒是填了不少坑，等助教能来答题了我拉一个对 embed 比较熟的问问

## 68. 关于直播时答疑和视频回放

### 问题

感谢曹大今天的精彩课程，感觉干货还是挺多的。但是真心建议曹大调整下授课方式：

1. 建议每讲一块知识点专门留一页答疑 PPT，而不是一直去答疑；今天听课时好多知识点一直不停地被打断，感觉思路十分跳跃；
2. 建议回答评论区问题时可以先简单的将问题复述一下；每节课课程的容量还是有一些多的，而且每个人也不能保证每次上课都能赶上直播，不可避免的要去看回放，这样直接回答评论区问题会让大家看回放时不知所云；
3. 希望曹大能有意识地控制下课程时间，优质问题沉淀到问答区，课堂上专注于本节课内容；

曹大人真的超好，但希望课堂能够再高效一点点，大家共同高效地学习！

以上都是个人的一点点小建议，抛砖引玉，希望大家能够一起变好

## 回答

是的，我回看了一下视频，中间被打断的话，录下来的视频回看体验确实不好，后面我把讲课和答疑分开，保证讲课的时候内容是连贯的

## 69. 被挤出的 G 加在链表的 tail 处是不是降低了处理优化级？

### 问题

当 local run queue 满的时候取一半+挤出的 runnext，加入到 gobal run queue 的 tail 位置，可是消费的时候是从 gobal 的 head 开始，这样是不是降低该 G 本应该有的优化级？

## 回答

对，把它们往全局队列推，确实相当于降低了本地队列后半部分的优先级

# 一手资源+V 307570512

不过这时候系统整体的压力应该挺高的了

## 70. Goroutine 启动与初始化的顺序

### 问题

昨天曹大布置了一个代码问题，要求自行下去运行观察结果

The image shows two side-by-side code snippets in a terminal window. The left snippet is titled "一个无聊的输出顺序的问题" and contains the following Go code:package main  
import "fmt"  
import "runtime"  
  
func main() {  
 runtime.GOMAXPROCS(1)  
 for i := 0; i < 10; i++ {  
 i:=i  
 go func() {  
 fmt.Println("A: ", i)  
 }()  
 }  
  
 var ch = make(chan int)  
 ←ch|  
}The right snippet contains the same code but with a different execution flow:package main  
import "time"  
import "fmt"  
import "runtime"  
  
func main() {  
 runtime.GOMAXPROCS(1)  
 for i := 0; i < 10; i++ {  
 i := i  
 go func() {  
 fmt.Println("A: ", i)  
 }()  
 }  
  
 time.Sleep(time.Hour)  
}A question above the right code asks, "这两段代码分别输出什么？"

使用 channel 的版本会死锁。因为 channel 必须有一对接收。

第二个版本会打印，结果。但是都会最先最后一个 goroutine 的代码。这是因为曹大昨天说的局部性原理吗？最后一个被加到 runnext 中，最先执行？

我的理解是启动瞬间，创建了很多 goroutine，然后前面的还没有来得及抢占 runnext，就被挤到 P 的队列去了，最后一个就最后抢占到 runnext，所以最后一个就最先执行。

### 回答

这里的关键其实就是 `GOMAXPROCS = 1`，和 `runnext`，新创建的 g 优先级最高

这个结论不需要背，因为它在不同的 Go 版本中表现是不一样的，如果有人问这种问题，你能说清楚 `runnext`, `local queue` 和 `global queue` 就可以了

## 71. 线程创建的策略是什么，创建数量是否可控，会不会导致线程数量过多？

# 一手资源+V 307570512

## 问题

线程是如何创建的，数量是否可控，会不会导致线程数量过多，该如何处理

## 回答

一般情况下，M 约等于 P 的数量，当有 M 被阻塞的时候，会创建额外的 M 去处理任务

课程中讲了可以被接管的阻塞和不可被接管的阻塞，

不可接管的阻塞(cgo, syscall)状态的线程如果太多，那就有可能创建很多线程

M 的数量在 runtime 中默认限制 10000，超过 10000 进程会崩溃。已经创建出来的 M 是没有办法销毁的，可以看看我课上讲 cgo 阻塞的那页下面的链接给出来的方案。

## 72. M 的创建和 P 的分配疑问

## 问题

GPM 是怎样一个策略创建的 M？runtime 打包的计算任务(go func)是怎样的策略绑定给哪个 P

## 回答

M 创建其实就是几个原则：

1. 创建了多少个 P，那就至少需要多少个 M
2. 如果 M 被 syscall/cgo 这类必须独占线程的操作阻塞住了，要保证 CPU 所有核心仍然能够得到充分利用，就需要创建更多的 M(这里说的是没有空闲 M 的时候)，这时候 sysmon 剥离掉那些阻塞住的 M 的 P 以后，P> 正在执行调度循环的 M，那就需要把 M 补齐  
创建 M 的函数是 runtime.startm，有兴趣你可以跟一下这个函数在哪里被调用

go func 是个语法糖，本质上就是调用了 runtime.newproc，创建出来的 g 和当前创建这个 g 的那个 g 在同一个线程上，也就对应的是这个线程绑定的那个 P，进的队列也是优先进这个 P 的 runnext 和 local queue

## 再次追问

创建出来的 g 和当前创建这个 g 的那个 g 在同一个线程上

这里我有个疑问，就是在 main 函数里面 go func 创建的 g 是不是都在 main goroutine 绑定线程所在的 P 的 runq。假如我的应用都是在 main 函数创建的 goroutine，是不是都在同一

# 一手资源+V 307570512

个 runq 了，然后其它 m 绑定的 p 没有可执行的 g，再去 main goroutin 对应的 runq 去 stealing。不知道我的理解对不对。

创建出来的 g 和当前创建这个 g 的那个 g 在同一个线程上 ———》 那么在 main 函数调用的后续函数中，如果创建 go func，那么都会绑定到 main goroutine 绑定线程所在的 P 的 runq。

那么剩余的其他 p 不是在空闲了吗？

## 回答

创建出来的 g 和当前创建这个 g 的那个 g 在同一个线程上  
这里我有个疑问，就是在 main 函数里面 go func 创建的 g 是不是都在 main goroutine 绑定线程所在的 P 的 runq。假如我的应用都是在 main 函数创建的 goroutine，是不是都在同一个 runq 了，然后其它 m 绑定的 p 没有可执行的 g，再去 main goroutin 对应的 runq 去 stealing。不知道我的理解对不对。

是的，main goroutine 创建出来的一开始和 main goroutine 肯定在同一个线程，都在一个 runq 里

其它 m 没可执行的 g 会偷走

你理解的没问题~

慕课这种问题后面的连环问好像没法看到通知，好尴尬

## 73. 如何从一个知识点获得源码层面的理解

### 问题

经验已经告诉我执行下面的代码会阻塞主进程，然后会报错 all goroutines are asleep - deadlock!

```
Plain Text
package main
func main() {
    ch1 := make(chan string)
    ch1 <- "hello"
}
```

但是就是好奇，

# 一手资源+V 307570512

- 为什么非缓存队列只接受不发送就会阻塞？
- 为什么主进程阻塞了，会报错 all goroutines are asleep - deadlock!

然后去源码搜索发现了个函数

```
Plain Text
// sched.lock must be held.
func checkdead() {
    // For -buildmode=c-shared or -buildmode=c-archive it's OK if
    // there are no running goroutines. The calling program is
    // assumed to be running.
    if islibrary || isarchive {
        return
    }
    ...
    ...
    ...
    // There are no goroutines running, so we can look at the P's.
    for _, _p_ := range allp {
        if len(_p_.timers) > 0 {
            return
        }
    }
    getg().m.throwing = -1 // do not dump full stacks
    unlock(&sched.lock) // unlock so that GODEBUG=scheddetail=1 doesn't hang
    throw("all goroutines are asleep - deadlock!")
}
```

但看到这些，毫无头绪从哪里看起，所以向问问曹大，当年读源码的经验？，指点一二。

## 回答

首先要有一个宏观的理解，比如就是第一课讲的，知道整个 Go 底层就是一堆线程在不停地执行调度循环

然后在日常碰到问题的时候，再去看详细的实现。

比如你这里的 checkdead：

- 搜索代码以后发现是在 sysmon 里，通过课程中讲的内容，知道 sysmon 是后台高优先级的一个监控线程，负责干什么事情
- 然后看 checkdead 的注释，写的是检查是不是目前所有的线程都被阻塞住，那就大概了解这个流程了

有宏观的认识，再去填充细节，我个人认为这样是比较好的

不过 checkdead 这个流程在我们写服务的时候基本上没啥用

# 一手资源+V 307570512

第一节课讲了调度循环，了解了调度循环的流程，你对整体的运行框架应该就有了基础的理解

碰到问题的时候，像你这里说的 checkdead，你只要看看 checkdead 是在哪个阶段被调用的，它的功能是什么，再简单画一下图就理解了。

我看 checkdead 的时候，感觉他的注释写的还挺清楚的

中间我看看要不要讲讲怎么读 Go 的底层代码，里面其实有很多代码是可以忽略不读的，特别是一些 if raceenable, traceenable 或者你这里前几行 c-shared, library 之类的，可以先跳过的

[74. 曹大，在基础、进阶、深入三个阶段分别有什么 Golang 书单推荐一下吗？](#)

## 问题

下列 3 本是准备开始啃的书。

基础：

- [《Go by Example 中文版》](#)
- [《The Go Programming Language》中文版](#)

进阶：

- [《Go 语言高级编程》](#)

上课中提到了一本好书叫： 《Designing Data-Intensive Application》，其中文版叫《数据密集型应用设计》

## 回答

基础： go by example, the go programming language, go in action

进阶： 语言方向，Go 语言高级编程或者一些讲源码的书可以翻翻；领域方向，要看自己业务相关的书，比如 k8 之类的

深入： 自己看代码，找实现，写总结，主要是实践。深入下去很多东西就比较理论了，要看 Go 以外的书，比如计算机组成原理，操作系统，这些基础也需要复习才能懂语言

# 一手资源+V 307570512

的底层为什么这么设计

## 75. 调度流程没看明白

### 问题

触发调度中， preemptPark -> schedule 和 goyield\_m -> schedule 没看明白，希望曹大解答。

从看源码发现了有 7 个入口触发 schedule：

1. mstart1 -> schedule // 线程启动时触发
2. park\_m -> schedule // gopark
3. goschedImpl -> schedule // 协作式调度
4. goexit0 -> schedule // Goroutine 执行结束后触发
5. exitsyscall0 -> schedule // 系统调用
6. preemptPark -> schedule // ???
7. goyield\_m -> schedule // ???

### 回答

8. mstart1 -> schedule // 线程启动时触发
9. park\_m -> schedule // gopark
10. goschedImpl -> schedule // 协作式调度
11. goexit0 -> schedule // Goroutine 执行结束后触发
12. exitsyscall0 -> schedule // 系统调用
13. preemptPark -> schedule // 这个我看代码是 GC 流程抢占正在运行的 g, m 进 schedule
14. goyield\_m -> schedule // 这个是饥饿模式，当前的执行 unlock 那个 g 需要主动让出一下，进 local runq，让被唤醒的那个 goroutine 去执行

## 语法背后的秘密

<https://class.imooc.com/lesson/62#mid=46368>

### 1. 第二课 Go 语法背后的秘密课堂实时问题文档

### 问题

课堂上问答的石墨记录

<https://shimo.im/docs/ghpktdY8VyhVKgw>

### 回答

# 一手资源+V 307570512

直播过程中的问题汇总在上述文档中了

## 2. 关于未初始化 channel 读写问题

### 问题

在读写 nil channel 的时候会永久阻塞，阅读源码发现 chansend chanrecv 开头都判断了 ch 为 nil 就直接 gopark 了

```
Plain Text
if c == nil {
    if !block {
        return
    }
    gopark(nil, nil, waitReasonChanReceiveNilChan, traceEvGoStop, 2)
    throw("unreachable")
}
```

想问一下为什么要这么做呢？有什么应用场景吗

### 回答

<https://medium.com/justforfunc/why-are-there-nil-channels-in-go-9877cc0b2308>

看看这篇文章里提的例子，

TLDR; merge 多个 channel 的时候，如果 select 里的某个 case 的 channel 已经 close 了，可以把它置为 nil，避免 busy loop 浪费 CPU

## 3. 函数调用时基于栈和基于寄存器有什么区别？

### 问题

在第一课后，我们有看文章关于函数调用规约的。现在的 go 的函数调用就会在栈上面操作，进栈/出栈等等。但是未来会改到寄存器（有 [github](#) 讨论，也有 [blog](#) 的陈述）。

栈和寄存器都是“虚拟”出来的，既然寄存器更快，为什么编程语言实现函数调用不都基于寄存器呢？另外我也有看到一些文章说，其实寄存器也是被虚拟出来的，部分语言也用内存虚拟寄存器。

函数调用就是需要进栈/出栈的效果，这样一来，其实是内存上，还是寄存器上，感觉是不是无所谓了？只要能存一个临时的数据即可。那基于栈的和基于寄存器的实现到底有什么区别呢？

# 一手资源+V 307570512

么区别呢？如果虚拟寄存器也有使用内存来虚拟，那基于栈和基于寄存器，本质上速度不是一样吗？

## 回答

程序用的是虚拟内存，但你在访问内存的时候，虚拟内存肯定还是要访问物理内存的。。  
这个是抽象

寄存器一般不是虚拟出来的，Go 语言比较特殊，是为了编程方便，所以在汇编代码里有几个虚拟寄存器，这些东西在最终的汇编代码一定是被翻译成物理寄存器的~

从硬件来说，内存和寄存器实现的物理元件都不一样，内存是 DRAM，寄存器一般是 SRAM，SRAM 要比 DRAM 快很多。

这种速度快是有代价的，主要代价就是贵，所以本身寄存器的数量都是有限的，每个寄存器有固定的名字，寄存器的总数量也是有限的。

用寄存器来传参数可以加速函数调用过程，但你也可以看到，因为寄存器数量有限，但函数参数是可以无限的，总会有参数数量超过寄存器数量的时候，所以就算是用寄存器来做函数调用优化，在超出寄存器数量之后，还是要用到栈来传参的~

## 4. go 如何部署新代码

## 问题

go 线上部署的时候 都是通过 k8s 和 docker 配合部署吗 有那个单台机器类似 nginx 的 reload 的部署方式吗 从而保证请求无缝切换

## 回答

有 graceful 的方案  
实际用的比较少

## 5. 如何搭建线上 pprof 的采集器

### 问题

在线上能够安装使用 pprof fgprof 这些采集器吗 对业务有什么影响吗 并且如何限制安全性  
如何

如何配置让 pprof 自动在 CPU 等资源飙升的时候记录信息

如果线上突然一个 go 的接口出现超时 如何排查呀

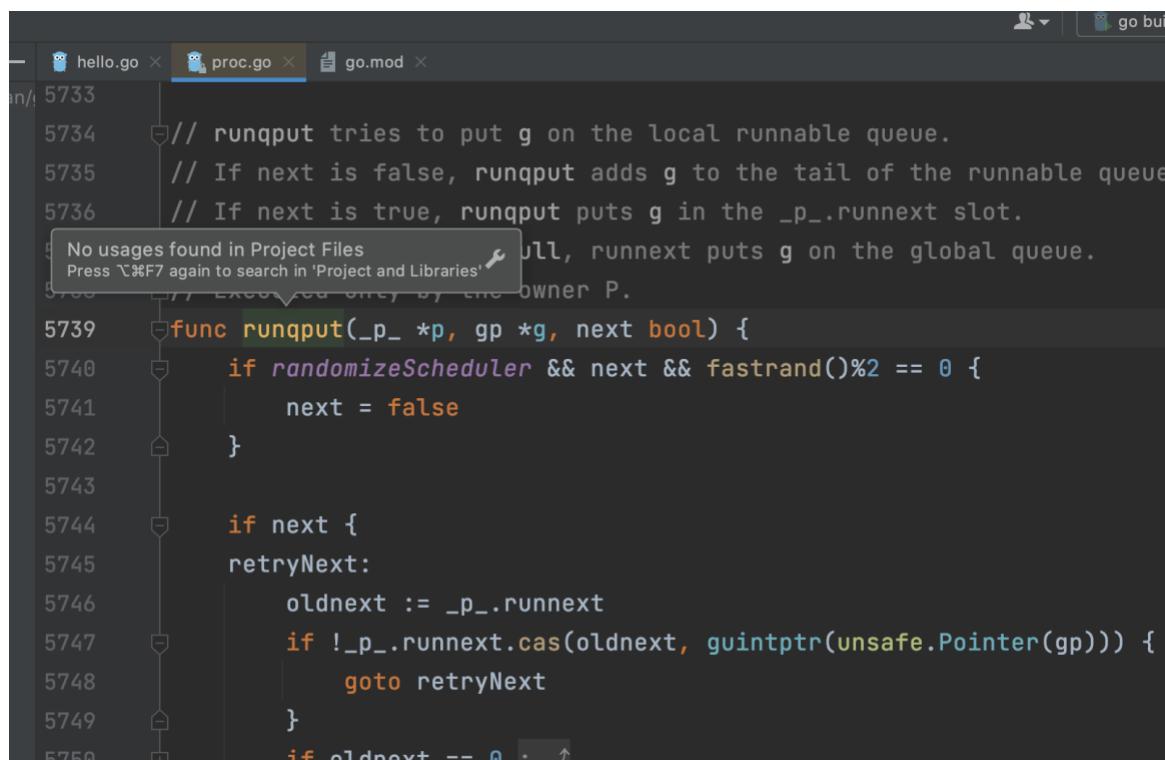
## 回答

1. pprof 可以开，但是有些公司可能开端口比较敏感，可以设成动态开启、关闭  
fgprof 成本很高，不建议开，线下压测可以用用
2. 默认不可以，需要借助工具，可以参考我在蚂蚁工作的时候写的工具：  
[github.com/mosn/holmes](https://github.com/mosn/holmes)
3. 要分情况讨论，如果是因为资源使用飙升导致延迟升高，需要用 2 中的工具；如果是因为丢包之类的导致的，那需要你们有 tcp\_retrans 之类的监控。

## 6. mac 的 GoLand 无法查看调用方

## 问题

按老师您的操作，按住 command 然后点击 runqput 函数名，没有出现和视频里的效果，是因为你的是 IDE，我的是 GoLand？那 GoLand 的快捷键是什么呢？



The screenshot shows the GoLand IDE interface with three tabs open: 'hello.go', 'proc.go' (which is the active tab), and 'go.mod'. The code in 'proc.go' is as follows:

```
5733
5734     // runqput tries to put g on the local runnable queue.
5735     // If next is false, runqput adds g to the tail of the runnable queue
5736     // If next is true, runqput puts g in the _p_.runnext slot.
5737     // Local queue only, b, the owner P.
5738
5739 func runqput(_p_ *p, gp *g, next bool) {
5740     if randomizeScheduler && next && fastrand()%2 == 0 {
5741         next = false
5742     }
5743
5744     if next {
5745         retryNext:
5746         oldnext := _p_.runnext
5747         if !_p_.runnext.cas(oldnext, guintptr(unsafe.Pointer(gp))) {
5748             goto retryNext
5749         }
5750     }
5751 }
```

A tooltip is visible over the 'runqput' function definition, containing the text: 'No usages found in Project Files' and 'Press ⌘F7 again to search in "Project and Libraries"'. The tooltip also includes a small icon of a person.

## 回答

你这个看起来是从应用代码点进去的，

# 一手资源+V 307570512

得用 goland 打开你的 go 安装目录，

默认是在 /usr/local/go 下，也就是用 goland 直接打开 /usr/local/go

## 7. 看 ast 结果， Decl: \*(obj @ 7)什么意思

### 问题

Plain Text

```
0 *ast.File {
    1 . Package: 2:1 #go 解析出的 package 是在源文件(src)的第二行第一个位置
    2 . Name: *ast.Ident { #File.Name 指的是包名
    3 . . NamePos: 2:9 #包名的位置是在第二航的第 9 个位置开始
    4 . . Name: "main" #包名叫 main
    5 . }
    6 . Decl: []ast.Decl {#顶级申明， 是一个 Decl 类型的 slice， 长度为 1， 表示
        有一个函数
        7 . . 0: *ast.FuncDecl {#第 0 个函数的定义
        8 . . . Name: *ast.Ident {#是一个标识符类型
        9 . . . . NamePos: 4:6 #方法名出现在 src 文件第 4 行的第 6 个位置
        10 . . . . Name: "main" #名称就叫 main
        11 . . . . Obj: *ast.Object {#这个标识符的扩展信息
        12 . . . . . Kind: func #类型是 func
        13 . . . . . Name: "main" #名字为 main
        14 . . . . . Decl: *(obj @ 7)
        15 . . . . }
        16 . . . }
    17 . . . Type: *ast.FuncType {#类型的申明， 是一个 FuncType 类型
    18 . . . . Func: 4:1 #func 关键字传销在第 4 行的第一个位置
    19 . . . . Params: *ast.FieldList {#函数的入参
    20 . . . . . Opening: 4:10 #左半边圆括号的位置
    21 . . . . . Closing: 4:11 #右半边圆括号的位置
    22 . . . }
                                #Results, 如果函数有返回值， 将会有
```

Results 这个属性

```
23 . . .
24 . . . Body: *ast.BlockStmt { #函数体
25 . . . . Lbrace: 4:12 #左半个大括号{出现的位置
26 . . . . List: []ast.Stmt { #每一行称述语句的定义， 是一个 Stmt 的切片
27 . . . . . 0: *ast.ExprStmt { #第 0 行， ExprStmt 的结构体定义只有一个 X， X 的类型
        是一个 Expr
```

# 一手资源+V 307570512

```
#interface, 需要实现的是一个 exprNode()方法, CallExpr 实现了
```

```
#这个方法, 所以此处的 X 是一个 CallExpr 调用表达式类型
```

```
28 . . . . . X: *ast.CallExpr {  
29 . . . . .   Fun: *ast.Ident {  
30 . . . . .     NamePos: 5:2 #方法名的位置是第 5 行第 2 个  
31 . . . . .     Name: "println" #方法名是 println  
32 . . . . .   }  
33 . . . . .   Lparen: 5:9 #左括号(  
34 . . . . .     Args: []ast.Expr {len = 1} { #参数的长度是 1  
35 . . . . .       0: *ast.BinaryExpr { #二进制  
36 . . . . .         X: *ast.BasicLit { #操作符左边  
37 . . . . .           ValuePos: 5:10 #出现位置  
38 . . . . .           Kind: INT #int 类型  
39 . . . . .           Value: "1" #值为 1  
40 . . . . .         }  
41 . . . . .       OpPos: 5:12 #操作符位置  
42 . . . . .       Op: + #操作符  
43 . . . . .       Y: *ast.BasicLit {  
44 . . . . .         ValuePos: 5:14  
45 . . . . .         Kind: INT  
46 . . . . .         Value: "2"  
47 . . . . .       }  
48 . . . . .     }  
49 . . . . .   }  
50 . . . . .   Ellipsis: -  
51 . . . . .   Rparen: 5:15 #右括号  
52 . . . . . }  
53 . . . . }  
54 . . . . }  
55 . . . Rbrace: 6:1 #右半个大括号 } 的位置  
56 . . . }  
57 . . . }  
58 . . . }  
59 . Scope: *ast.Scope {  
60 . . Objects: map[string]*ast.Object {len = 1}  
61 . . . "main": *(obj @ 11)  
62 . . . }  
63 . . . }  
64 . Unresolved: []*ast.Ident {len = 1}  
65 . . . 0: *(obj @ 29)  
66 . . . }  
67 }
```

上面是我根据源码对 ast 的一点解释, 代码中的 14、61、65 行里的 obj@数字什么意思, 没理解, 麻烦曹大解释一下

## 回答

```
Plain Text
case reflect.Ptr:
    p.printf("*")
    // type-checked ASTs may contain cycles - use ptrmap
    // to keep track of objects that have been printed
    // already and print the respective line number instead
    ptr := x.Interface()
if line, exists := p.ptrmap[ptr]; exists {
    p.printf("(obj @ %d)", line)
} else {
    p.ptrmap[ptr] = p.line
    p.print(x.Elem())
}
```

## 8. ast 结构里解释，Decl: \*(obj @ 7)什么意思

### 问题

```
Plain Text
0 *ast.File {
1 . Package: 2:1 #go 解析出的 package 是在源文件(src)的第二行第一个位置
2 . Name: *ast.Ident { #File.Name 指的是包名
3 .. NamePos: 2:9 #包名的位置是在第二航的第 9 个位置开始
4 .. Name: "main" #包名叫 main
5 .
6 . Decl: []ast.Decl {#顶级申明，是一个 Decl 类型的 slice，长度为 1，表示有一个函数
7 .. 0: *ast.FuncDecl {#第 0 个函数的定义
8 ... Name: *ast.Ident {#是一个标识符类型
9 .... NamePos: 4:6 #方法名出现在 src 文件第 4 行的第 6 个位置
10 .... Name: "main" #名称就叫 main
11 .... Obj: *ast.Object {#这个标识符的扩展信息
12 ..... Kind: func #类型是 func
13 ..... Name: "main" #名字为 main
14 ..... Decl: *(obj @ 7)
15 .... }
16 ...
17 ... Type: *ast.FuncType {#类型的申明，是一个 FuncType 类型
18 .... Func: 4:1 #func 关键字传销在第 4 行的第一个位置
19 .... Params: *ast.FieldList {#函数的入参
20 ..... Opening: 4:10 #左半边圆括号的位置
21 ..... Closing: 4:11 #右半边圆括号的位置
22 .... }
#Results, 如果函数有返回值，将会有 Results 这个属性
```

# 一手资源+V 307570512

```
23... }
24... Body: *ast.BlockStmt { #函数体
25.... Lbrace: 4:12 #左半个大括号{出现的位置
26.... List: []ast.Stmt (len = 1) { #每一行称述语句的定义，是一个 Stmt 的切片
27.... 0: *ast.ExprStmt { #第 0 行， ExprStmt 的结构体定义只有一个 X， X 的类型是一个 Expr
#interface，需要实现的是一个 exprNode()方法， CallExpr 实现了
#这个方法，所以此处的 X 是一个 CallExpr 调用表达式类型
28..... X: *ast.CallExpr {
29..... Fun: *ast.Ident {
30..... NamePos: 5:2 #方法名的位置是第 5 行第 2 个
31..... Name: "println" #方法名是 println
32..... }
33..... Lparen: 5:9 #左括号(
34..... Args: []ast.Expr (len = 1) { #参数的长度是 1
35..... 0: *ast.BinaryExpr { #二进制
36..... X: *ast.BasicLit { #操作符左边
37..... ValuePos: 5:10 #出现位置
38..... Kind: INT #int 类型
39..... Value: "1" #值为 1
40..... }
41..... OpPos: 5:12 #操作符位置
42..... Op: + #操作符
43..... Y: *ast.BasicLit {
44..... ValuePos: 5:14
45..... Kind: INT
46..... Value: "2"
47..... }
48..... }
49..... }
50..... Ellipsis: -
51..... Rparen: 5:15 #右括号
52..... }
53..... }
54..... }
55.... Rbrace: 6:1 #右半个大括号}的位置
56... }
57.. }
58. }
59. Scope: *ast.Scope {
60.. Objects: map[string]*ast.Object (len = 1) {
61.. "main": *(obj @ 11)
62.. }
63.. }
64. Unresolved: []*ast.Ident (len = 1) {
65.. 0: *(obj @ 29)
66.. }
67 }
```

# 一手资源+V 307570512

这是一段我自己看完 ast.go 相关源码后的解释，其中

Plain Text

Decl: \*(obj @ 7)

Plain Text

"main": \*(obj @ 11)

Plain Text

0: \*(obj @ 29)

是什么意思，没太看懂，尤其是@后面跟一个数字

## 回答

这个序号其实就是输出的这一块的行号。例如

Plain Text

Decl: \*(obj @ 7)

其实是指的：

Plain Text

7 .. 0: \*ast.FuncDecl {#第 0 个函数的定义}

其他和类推。

谜底就在谜面上。^\_^

如果你还想再了解多一点，我再给多一点提示：

用下面这个命令，你可以找到 go 源码中输出 'obj @' 是在哪个文件，顺着它就能找到为何输出这个了：

Go

```
find ./ -name "*.go" | xargs grep "obj @"
```

不过，我问了下曹大，他也不知道这个，可以到此就行了，不用深究，万一哪天用了再来研究就行了。

[9. 请问 closechan\(\)为什么没有回收 chan 的内存？](#)

## 问题

# 一手资源+V 307570512

```
Plain Text
func main() {
ch := make(chan int, 1)
fmt.Printf("%p\n", ch)
close(ch)
fmt.Printf("%p\n", ch)
}
0xc000052070
0xc000052070
```

通过示例和查看源代码，看到 closechan 之后，chan 的内存并没有被回收。

想象中，closechan()后，这个 chan 就没用了，应该回收掉减少 gc 压力  
golang 为什么不这么做？

## 回答

close 的 channel，  
buffer 不一定清空了啊，还可能被使用的

你 main 函数后半段也用到了这个 chan  
chan 这时候从语义上还是可达的，GC 也不应该回收它

## 再次追问

```
Plain Text
func Consume(ch chan int) {
    go func() {
        select {
        case x, ok := <-ch:
            fmt.Println(x, ok)
        }
    }()
}

func main() {
    ch := make(chan int, 1)
    Consume(ch)
    ch <-99
    fmt.Printf("%p\n", ch)
    close(ch)
    fmt.Printf("%p\n", ch)
    ch = make(chan int, 1)
    fmt.Printf("%p\n", ch)
    ch <-100
}
```

# 一手资源+V 307570512

```
    time.Sleep(time.Hour)  
}
```

```
0xc000052070  
0xc000052070  
0xc0000520e0  
99 true
```

请看下这个例子

我重新赋值之后，chan 的地址已经改变了，之前的那个 chan 地址也没利用起来，Consume 收不到了。

还有，原来的那个地址，在重新赋值时候被回收了吗？

## 回答

按你的代码，也不会被回收的，因为 consume 那边还在访问这个 ch 的，  
垃圾回收的前提是从代码上就访问不到这个对象了，像 ch 这种，就是你生产和消费端都  
退出了，才会被 GC

## 10. 多态情况下的函数调用，汇编地址及对象模型

### 问题

golang 在多态的情况下，

1. 汇编代码的地址是怎样的？

a. 静态函数的地址是使用了一个 call，比如 close(ch) -> CALL runtime.closechan(SB)

2. 对象模型是怎样的？类似 C++ 中的虚表吗？

比如下面 A 中的函数 Do() 在变量 i 中是怎样存在的？运行时，是怎样找到 A 中的 Do() 的？

相关代码：

```
Plain Text  
type ITest interface {  
Do()  
}  
  
type A struct {  
}  
  
func (r *A)Do() {  
}
```

# 一手资源+V 307570512

```
func main() {
var i ITest = &A{}
i.Do()
}
```

## 回答

你先看看我发的那个链接，里面讲的还是比较全的  
有问题再跟贴就行了

[https://github.com/go-internals-cn/go-internals/blob/master/chapter2\\_interfaces/README.md](https://github.com/go-internals-cn/go-internals/blob/master/chapter2_interfaces/README.md)

## [11. 关于从当前 m 运行的 g 栈上切换到 g0 栈上的问题](#)

### 问题

- 1、为什么源码中很多关键的地方（执行调度，创建 g）都是在 m 的 g0 的栈上运行。
- 2、怎么理解 tls 他是干嘛用的。在 m 中他的作用是什么
- 3、g 栈在切换过程中，原来的栈是只是保存了必要信息然后将整个栈空间销毁吗

## 回答

1. 因为执行函数会改变 goroutine 的栈，调度指令执行的时候不能影响用户的栈，所以要切
2. tls 是 thread local storage，在 Go 语言里，tls 用来保存当前线程正在执行的 g 的地址。具体实现得看 arch\_prctl 这个 linux 的系统调用(用到了 FS 寄存器)，在 linux 以外的平台，可能实现不一样。
3. g 栈在切换的过程中，原来的 g 会保存在 g.sched 字段里，是个 gobuf 结构，你得看看 gobuf 的字段，原来的栈没销毁。只是现在执行栈跑去 g0 了而已。

如果你想看切换的具体逻辑，需要有汇编基础，看 runtime·systemstack

## [12. 我在调试 panic: send on closed channel 的情况使用 dlv 一直 s 只会进入 fatalpanic\(\)](#)

### 问题

```
Plain Text
package main
```

# 一手资源+V 307570512

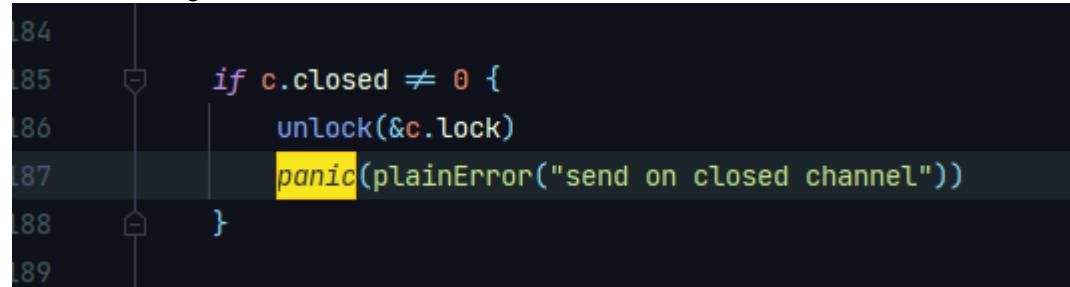
```
func main() {
    ch := make(chan int)
    close(ch)
    ch <- 1
}
```

上面的代码用 dlv 工具进行 debug，在 `ch <- 1` 打断点，一直 s 下去只会进入 `fatalpanic()` 函数然后直接 exit 出去了。

Plain Text

```
systemstack(func() {
    exit(2)
})
```

但是后面换 goland 直接全局搜索到 `chan.go` 中的 `send on closed channel`，在此处打断点，然后进行 debug 却可以走到这一步



```
l84
l85      if c.closed != 0 {
l86          unlock(&c.lock)
l87          panic(plainError("send on closed channel"))
l88      }
l89
```

请问老师这是为啥？

## 回答

我试了一下

这里要用 si 才能进去~

s 不行，

si 要多按几次

之后看到函数调用就可以直接断点进去了

## 再次追问

为啥 `ch <- 1` 的时候只有 `

si` 单步单核调试能到 `chansend1()` 里？难道往 channel 里塞数据也会起 goroutine？

## 回答

这个应该只是 dlv 自己不认识这种语法糖翻译的问题，不用想太多。。晚上讲作业的时候

# 一手资源+V 307570512

会说说这个

## [13. 关于向关闭的 channel 中发送数据的问题](#)

### 问题

```
Plain Text
ch := make(chan int)
close(ch)
ch<-1
```

#### 问题描述:

曹大代码如上，再用 dlv 调试的时候我发现上述代码好像一共调用了 2 次 chansend 方法，请问这个是什么逻辑呢。

#### 尝试过的解决方式:

自己在顺着源码分析得时候，没能看懂相应得原因

### 回答

程序启动的时候 gc 相关的也有一些 channel 操作，你可以用 dlv 的 bt 命令，看看当前的 chansend 是从哪里进去的，

```
Plain Text
(dlv) c
> runtime.chansend() /usr/local/go/src/runtime/chan.go:142 (hits goroutine(4):1 total:1) (PC:
0x1004053)
Warning: debugging optimized function
 137: * sleep can wake up with g.param == nil
 138: * when a channel involved in the sleep has
 139: * been closed. it is easiest to loop and re-run
 140: * the operation; we'll see that it's now closed.
 141: */
=> 142: func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
 143:     if c == nil {
 144:         if !block {
 145:             return false
 146:         }
 147:         gopark(nil, nil, waitReasonChanSendNilChan, traceEvGoStop, 2)
(dlv) bt
0 0x0000000001004053 in runtime.chansend
    at /usr/local/go/src/runtime/chan.go:142
1 0x0000000001004035 in runtime.chansend1
    at /usr/local/go/src/runtime/chan.go:127
2 0x000000000101c0c3 in runtime.bgscavenge
    at /usr/local/go/src/runtime/mgcscavenge.go:236
3 0x000000000105a941 in runtime.goexit
```

# 一手资源+V 307570512

```
at /usr/local/go/src/runtime/asm_amd64.s:1373
```

你要调试的是 main 里的这一句，所以可以先进 main.main 函数，然后再在 chansend 上打断点，然后按 c(continue)

下面这种有 main.main 的才是你应该注意的操作位置：

```
Plain Text
(dlv) c
> runtime.chansend() /usr/local/go/src/runtime/chan.go:142 (hits goroutine(1):1 total:3) (PC:
0x1004053)
Warning: debugging optimized function
137:     * sleep can wake up with g.param == nil
138:     * when a channel involved in the sleep has
139:     * been closed. it is easiest to loop and re-run
140:     * the operation; we'll see that it's now closed.
141: */
=> 142: func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
143:     if c == nil {
144:         if !block {
145:             return false
146:         }
147:         gopark(nil, nil, waitReasonChanSendNilChan, traceEvGoStop, 2)
(dlv) bt
0 0x00000000001004053 in runtime.chansend
    at /usr/local/go/src/runtime/chan.go:142
1 0x00000000001004035 in runtime.chansend1
    at /usr/local/go/src/runtime/chan.go:127
2 0x0000000000105ee13 in main.main
    at ./cs.go:6
3 0x0000000000102fdb8 in runtime.main
    at /usr/local/go/src/runtime/proc.go:203
4 0x0000000000105a941 in runtime.goexit
    at /usr/local/go/src/runtime/asm_amd64.s:1373
```

[14. 1.go 和 2.go 汇编后与 ppt 不太一致怎么评估哪个快](#)

## 问题

谢大，曹大，助教好，麻烦看下，下面是汇编后的情况

# 一手资源+V 307570512

The screenshot shows two instances of the GoLand IDE. The top instance is for project 'golearn' and file '1.go'. The bottom instance is for project 'golearn' and file '2.go'. Both files contain a main function that prints a struct to the console.

**File 1.go:**

```
package main
type Person1 struct {
    age int
}
func main() {
    var a = &Person1{age: 111}
    println(a)
}
```

**File 2.go:**

```
package main
type Person2 struct {
    age int
}
func main() {
    var b = Person2{age: 111}
    var a = &b
    println(a)
}
```

**Assembly Output (highlighted in red boxes):**

**File 1.go:**

Line	Address	OpCode	Description
1	0x440	483b6110	CMPQ \$0x10(CX), SP
2	0x444	7658	JBE 0x49e
3	0x446	4883ec28	SUBQ \$0x28, SP
4	0x448	48896c2420	MOVQ BP, 0x20(SP)
5	0x44f	488d6c2420	LEAQ 0x20(SP), BP
6	0x454	48c744240800000000	MOVQ \$0x0, 0x8(SP)
7	0x45d	488d442408	LEAQ 0x8(SP), AX
8	0x462	4889442418	MOVQ AX, 0x18(SP)
9	0x467	8400	TESTB AL, 0(AX)
10	0x469	48c74424086f000000	MOVQ \$0x6f, 0x8(SP)
11	0x472	4889442410	MOVQ AX, 0x10(SP)
12	0x477	e800000000	CALL 0x47c [1:5]R_CALL:runtime.printlock
13	0x47c	488b442410	MOVQ 0x10(SP), AX
14	0x481	48890424	MOVQ AX, 0(SP)
15	0x485	e800000000	CALL 0x48a [1:5]R_CALL:runtime.printpointer
16	0x48a	e800000000	CALL 0x48f [1:5]R_CALL:runtime.println
17	0x48f	e800000000	CALL 0x494 [1:5]R_CALL:runtime.printunlock
18	0x494	488b6c2420	MOVQ 0x20(SP), BP
19	0x499	4883c428	ADDQ \$0x28, SP
20	0x49d	c3	RET

**File 2.go:**

Line	Address	OpCode	Description
1	0x41f	65488b0c2500000000	MOVQ GS:0, CX [5:9]R_TLS_LE
2	0x428	483b6110	CMPQ \$0x10(CX), SP
3	0x42c	7651	JBE 0x47f
4	0x42e	4883ec20	SUBQ \$0x20, SP
5	0x432	48896c2418	MOVQ BP, 0x18(SP)
6	0x437	488d6c2418	LEAQ 0x18(SP), BP
7	0x45c	48c744240800000000	MOVQ \$0x0, 0x8(SP)
8	0x445	48c74424086f000000	MOVQ \$0x6f, 0x8(SP)
9	0x44e	488d442408	LEAQ 0x8(SP), AX
10	0x453	4889442410	MOVQ AX, 0x10(SP)
11	0x458	e800000000	CALL 0x45d [1:5]R_CALL:runtime.printlock
12	0x45d	488b442410	MOVQ 0x10(SP), AX
13	0x462	48890424	MOVQ AX, 0(SP)
14	0x466	e800000000	CALL 0x46b [1:5]R_CALL:runtime.printpointer
15	0x46b	e800000000	CALL 0x470 [1:5]R_CALL:runtime.println
16	0x470	e800000000	CALL 0x475 [1:5]R_CALL:runtime.printunlock
17	0x475	488b6c2418	MOVQ 0x18(SP), BP
18	0x47a	4883c420	ADDQ \$0x20, SP
19	0x47e	c3	RET

哪个代码效率比较高我认为是不是汇编代码行数比较少的效率高，为什么要这样，ppt说的是一样的，我的环境是在 mac 下的 go 版本是

# 一手资源+V 307570512

go1.14.12 darwin/amd64

## 回答

我在 mac 上得到的结果也是一样，不确定你那里是不是环境还是什么有问题：

```
Plain Text
git:master >>> cat -n 1.go
1      package main
2
3      type person struct {
4          age int
5      }
6
7      func main() {
8          var a = &person{111}
9          println(a)
10     }

git:master >>> cat -n 2.go
1      package main
2
3      type person struct {
4          age int
5      }
6
7      func main() {
8          var b = person{111}
9          var a = &b
10         println(a)
11     }

git:master >>> go tool objdump 1 | grep -A 20 "main.main"
TEXT main.main(SB) /Users/xargin/test/fuc/1.go
1.go:7    0x1051570    65488b0c2530000000    MOVQ GS:0x30, CX
1.go:7    0x1051579    483b6110    CMPQ 0x10(CX), SP
1.go:7    0x105157d    7647    JBE 0x10515c6
1.go:7    0x105157f    4883ec18    SUBQ $0x18, SP
1.go:7    0x1051583    48896c2410    MOVQ BP, 0x10(SP)
1.go:7    0x1051588    488d6c2410    LEAQ 0x10(SP), BP
1.go:8    0x105158d    48c744240800000000    MOVQ $0x0, 0x8(SP)
1.go:8    0x1051596    48c74424086f000000    MOVQ $0x6f, 0x8(SP)
1.go:9    0x105159f    e81c3afdf    CALL runtime.printlock(SB)
1.go:9    0x10515a4    488d442408    LEAQ 0x8(SP), AX
1.go:9    0x10515a9    48890424    MOVQ AX, 0(SP)
1.go:9    0x10515ad    e80e43fdff    CALL runtime.printpointer(SB)
1.go:9    0x10515b2    e8993cfdf    CALL runtime.println(SB)
1.go:9    0x10515b7    e8843afdf    CALL runtime.printunlock(SB)
1.go:10   0x10515bc    488b6c2410    MOVQ 0x10(SP), BP
```

# 一手资源+V 307570512

```
1.go:10      0x10515c1    4883c418    ADDQ $0x18, SP
1.go:10      0x10515c5    c3          RET
1.go:7       0x10515c6    e83581ffff  CALL
runtime.morestack_noctxt(SB)
1.go:7       0x10515cb    eba3        JMP main.main(SB)
>>> go tool objdump 2 | grep -A 20 "main.main"
TEXT main.main(SB) /Users/xargin/test/fuc/2.go
2.go:7       0x1057390    65488b0c2530000000  MOVQ GS:0x30, CX
2.go:7       0x1057399    483b6110    CMPQ 0x10(CX), SP
2.go:7       0x105739d    7647        JBE 0x10573e6
2.go:7       0x105739f    4883ec18    SUBQ $0x18, SP
2.go:7       0x10573a3    48896c2410  MOVQ BP, 0x10(SP)
2.go:7       0x10573a8    488d6c2410  LEAQ 0x10(SP), BP
2.go:8       0x10573ad    48c744240800000000  MOVQ $0x0, 0x8(SP)
2.go:8       0x10573b6    48c74424086f000000  MOVQ $0x6f, 0x8(SP)
2.go:10      0x10573bf    e86c33fdff  CALL runtime.printlock(SB)
2.go:10      0x10573c4    488d442408  LEAQ 0x8(SP), AX
2.go:10      0x10573c9    48890424    MOVQ AX, 0(SP)
2.go:10      0x10573cd    e85e3cfdff  CALL runtime.printpointer(SB)
2.go:10      0x10573d2    e8e935fdff  CALL runtime.println(SB)
2.go:10      0x10573d7    e8d433fdff  CALL runtime.printunlock(SB)
2.go:11      0x10573dc    488b6c2410  MOVQ 0x10(SP), BP
2.go:11      0x10573e1    4883c418    ADDQ $0x18, SP
2.go:11      0x10573e5    c3          RET
2.go:7       0x10573e6    e8259dfffff CALL
runtime.morestack_noctxt(SB)
2.go:7       0x10573eb    eba3        JMP main.main(SB)

git:master >>> go version
go version go1.14.12 darwin/amd64
```

## 15. 关于 compile 指令的问题

### 问题

执行 go tool -S compile ./hello.go 指令后会生成一个 hello.o 的文件，这个文件是编译后的文件吗。如果是如何将这个 hello.o 的文件变成可执行文件呢，有什么命令可以执行吗。这两个过程就是编译和链接吗

### 回答

```
go tool compile hello.go ----> hello.o : 编译
go tool link hello.o ----> a.out: 链接
```

## 16. 曹大，关于函数调用时寄存器的位置，伪 SP, 硬件 SP, FP

# 一手资源+V 307570512

## 问题

函数调用时，栈是从高地址向低地址增长？

虚拟的 SP 寄存器，指向的对应的是栈的底部，其实高地址？

真的硬件 SP 寄存器对应的是栈顶，其实对应的是低地址？

调用方在调用函数时，已经将被方的入参和返回地址分配自己当前的栈上，所以被调方通过 FP+offset 的方式访问入参和返回值？

被调函数通过虚拟的 SP 减去偏移量访问局部变量，最先分配的局部变量其实离虚拟的 SP 远，离硬件 SP 近？

例如：

```
Plain Text
func p() {
    var a int
    var b int
    var c int
}
```

访问 a 其实是 a-24(SP) 或者 0(SP)？

## 回答

你这个就比较卷了，要学细节看看我以前写的 plan9 汇编总结吧

<https://github.com/cch123/golang-notes/blob/master/assembly.md>

里面应该能解答大部分 plan9 汇编的疑问

[17. 为什么说 rpc 的性能不高？](#)

## 问题

如题？

## 回答

课上的意思是说用 RPC 来提供模块的功能扩展性能不高

当时没说清楚 orz

## 18. Go 版本选择

### 问题

老师， golang 的版本确定了是 1.14.2 吗？我用最新版本是否有影响？

### 回答

1.14.12

你读可以读新的，但是交作业的时候注意要用这个

## 19. 1.14 之后的信号抢占调度

### 问题

谢大，曹大，助教好，我看到 1.14 之后的信号抢占有点不明白的地方：在 sysmon 检测到 g 执行时间过长的时候发起中断信号，这方面涉及到信号量的设置，信号的处理是涉及到系统底层的知识吗？有这方面的资料吗？我想研究下，还有就是后面会说信号抢占调度吗？

### 回答

信号处理要看看操作系统知识，软中断可以从任意位置(汇编指令粒度)中断用户正在执行的代码，并且进入信号处理函数。可以看看《The Linux Programming Interface》这本书的 signal 的两个章节。

这部分考虑到比较难，没在课上讲，我在 Gopher China 今年会讲讲这个

## 20. 关于原子操作的几个函数

### 问题

# 一手资源+V 307570512

```
// tbuild !wasm

package atomic

import "unsafe"

//go:nosescape
func Cas(ptr *uint32, old, new uint32) bool

// NO go:nosescape annotation; see atomic_pointer.go.
func Casp1(ptr *unsafe.Pointer, old, new unsafe.Pointer) bool

//go:nosescape
func Casuintptr(ptr *uintptr, old, new uintptr) bool

//go:nosescape
func Storeuintptr(ptr *uintptr, new uintptr)

//go:nosescape
func Loaduintptr(ptr *uintptr) uintptr

//go:nosescape
func Loaduint(ptr *uint) uint

// TODO(matloob): Should these functions have the go:nosescape annotation?

//go:nosescape
func Loadint64(ptr *int64) int64

//go:nosescape
func Xaddint64(ptr *int64, delta int64) int64
```

请问这几个函数的具体作用，每个函数中 old 或 new 或 ptr 这些形参传入的目的是什么。  
底层是如何实现的

## 回答

这个感觉应该先写一些简单的 atomic demo。。。然后再看

## [21. 关于 runtime 包中的 getg\(\) 函数](#)

## 问题

# 一手资源+V 307570512

```
// getg returns the pointer to the current g.  
// The compiler rewrites calls to this function into instructions  
// that fetch the g directly (from TLS or from the dedicated register).  
// getg返回指向当前g的指针。编译器将对此函数的调用重写为直接(从TLS或专用寄存器)获取g的指令。  
func getg() *g
```

曹大这个 getg() 返回的当前的 g 指的值什么。是全局的 g0 吗还是 M 结构体中的字段 g0，还是普通的 g

## 回答

指的是当前 m 上正在运行的 goroutine。可能是普通的用户 goroutine，如果 m 在执行 schedule 函数的话，就指向 m 结构体上的 g0。

全局 g0 指的是主线程（其实就是 m0）对应的 g0。

## [22. 关于 schedt 类型中的几个字段含义](#)

## 问题

```
gcwaiting uint32 // gc is waiting to run  
stopwait int32  
stopnote note  
sysmonwait uint32  
sysmonnote note

// safepointFn should be called on each P at the next GC
// safepoint if p.runSafePointFn is set.
safePointFn func(*p)
safePointWait int32
safePointNote note

profilehz int32 // cpu profiling rate
```

曹大这个是 runtime 包中的 schedt 结构体中定义的字段，红框标记的几个字段含义是什么啊

# 一手资源+V 307570512

## 回答

gwaiting 表示正在等待执行 GC，这时候可能还在抢占那些正在执行的 goroutine  
stopwait 和 stopnote 也是和 GC 流程相关计数器和类似锁的东西(需要阻塞等待)

sysmonwait 表示后台的 sysmon 线程是不是在休眠

sysmonnote 是 sysmon 休眠的时候需要用到的一个类似锁的东西(需要阻塞等待)

runtime 里的 xxxnote 一般都是需要休眠的时候会用到的结构，具体的解释在 type note struct 的结构体注释里

看代码的时候其实可以不用看这么细。。。刚开始了解脉络就行了，所有字段都想搞清楚很容易迷失

## 23. \$runtime.mainPC

## 问题

```
[dlv] n
> runtime.rt0_go() /usr/local/go/src/runtime/asm_amd64.s:223 (PC: 0x462405)
Warning: debugging optimized function
 218:     CALL    runtime.args(SB)
 219:     CALL    runtime.osinit(SB)
 220:     CALL    runtime.schedinit(SB)
 221:
 222:     // create a new goroutine to start program
=> 223:     MOVQ    $runtime.mainPC(SB), AX          // entry
 224:     PUSHQ   AX
 225:     PUSHQ   $0                      // arg size
 226:     CALL    runtime.newproc(SB)
 227:     POPQ    AX
 228:     POPQ    AX
```

[dlv] b mainPC
这个 \$ 代表啥意思？\$runtime.mainPC 是不是就是 runtime.main 函数

## 回答

\$ 开头，后面加一些字符串的，

在 Go 的汇编里表示的是一种常量，这里说的应该就是 runtime.main 函数的入口地址

## 24. goroutine 阻塞和解除阻塞这两个过程是什么样的

# 一手资源+V 307570512

## 问题

曹大当一个 goroutine 被阻塞是这个 G 被放在哪里这个过程是什么样的，当解除阻塞过又是什么样的。还有这两个过程都是当前 M 对应的 G0 调度的吗

## 回答

gopark  
goready

这两部分不是在 g0 里， runtime 切换到 g0 一般有会调用单独的函数，比如 mcall, systemstack 之类的

阻塞分两种：

**可接管阻塞：**

这部分流程

阻塞走的是 runtime.gopark/runtime.goparkunlock

恢复走的是 runtime.goready/runtime.ready， ready 的调用位置就是各种结构的唤醒流程

- 比如 channel 的， send 调用的时候，会看看 recvq 上有没有等待的 goroutine，如果有，就调用 goready 唤醒它
- 比如 mutex 的， unlock 的时候，会把 sema tree 上对应的等待队列的 goroutine 唤醒
- ppt 上列的几种情况你都可以按这个思路找一下阻塞进等待结构(一般都是个队列)，和唤醒流程

被阻塞的 g 要放在哪里，这个看 ppt 可接管阻塞的那页，有很多种情况，都列出来了。

**不可接管阻塞：**

这种只能让线程卡在那里，等恢复以后要看这个线程 M 绑定的 P 是不是被 sysmon 拿走了，如果已拿走，需要把这个 G 主动放到队列里

## 神奇的内置数据结构

### [1. mapextra 的作用？](#)

## 问题

曹大你好，不是很理解 map 中的 extra 的作用，我们不是可以通过 map 中的桶里面的 overflow 访问到溢出的数据，为什么还要链接到 extra？

## 回答

# 一手资源+V 307570512

回复套回复没法编辑，我另开一个能编辑的回复吧：

这个涉及到 Go 的 GC，因为 map 的 key 和 value 不含指针的时候，GC 是不扫描 bucket 里的 entry 的，但是 overflow 本质上是个指针，这些 overflow 的桶（虽然不多）需要被 GC 扫描得到，要不然会被当成不可达的内存给回收掉

所以在 map 的 kv 不含指针，且 key size < 128 && v size < 128 的时候，需要专门把这些溢出桶找一个结构记录一下，保证这些 bucket 对象在内存管理看来是 "alive" 的。要不然 GC 会将这些 overflow 的 bucket 当成白色对象提前回收掉。

<https://class.imooc.com/course/qadetail/291010>，看看这个的话呢，里面有俩图

## 再次追问

嗯嗯，这个我看过了，我看遍历也是直接通过 overflow 指针遍历的，感觉没有 extra 这样也可以完成 map 数据的增删改查；为什么要设计这个 extra 呢？

## 回答

哦，我知道你的意思了

这个涉及到 Go 的 GC，因为 map 的 key 和 value 不含指针的时候，GC 是不扫描 bucket 的，但是 overflow 本质上是个指针，这些 overflow 的桶（虽然不多）还是需要被 GC 扫描得到，要不然会被当成不可达的内存给回收掉

所以在 map 的 kv 不含指针，且 size < 128 的时候，需要专门把这些溢出桶找一个结构记录一下，来保证这些 bucket 对象在内存管理看来是 "alive" 的。

## 2. 关于 map 中的 extra 问题

## 问题

曹大，课上我们讲到 mapextra 表示预留的 bucket 组，

- ①那么比如 bucket1 的元素存满的时候是不是优先将新的元素存到预留的 bucket 中呢？
- ②是不是当 nextoverflow 满的时候，才会往 bucket1 的 overflow 链一个新的 bucket 呢？
- ③如果有数据在 mapextra 的 buckets 中，那么当我查询 key 的时候，我的 low bits 根据 B 的大小做&操作，那也就是说肯定无法命中到 mapextra 的 buckets 中，那不是废了吗？
- ④是不是 mapextra 的 bucket 也有指向原来的被 hash 冲突或者说是被 low bits 命中的那个 bucket 呢？

这里有点没理的很清楚。

## 回答

我们课上是 16 个桶，两个预留的

mapextra 里面那俩预留的就相当于提前分配了两个 overflow bucket，也就是说，前两个 overflow bucket 生成的时候，直接用这两个就行了，不够用了(nextoverflow == nil 说明不够用了)再 new。

这两个预留的 overflow bucket 不会通过索引来访问的，都是从原来的 bucket 的 overflow 指针链过去的

## 3. channel 源码细节问题

### 问题

问题 1：raceaddr 方法是做什么的？

在 makechan 中 当需要初始化一个没有缓冲的 channel 的时候有如下的代码.

#### Plain Text

```
// Hchan does not contain pointers interesting for GC when elements stored in buf do not
contain pointers.

    // buf points into the same allocation, elemtype is persistent.
    // SudoG's are referenced from their owning thread so they can't be collected.
    // TODO(dvyukov,rlh): Rethink when collector can move allocated objects.
    var c *hchan
    switch {
        case mem == 0:
            // Queue or element size is zero.
            c = (*hchan)(mallocgc(hchanSize, nil, true))
            // Race detector uses this location for synchronization.
            c.buf = c.raceaddr()
        case elem.ptrdata == 0:
            // Elements do not contain pointers.
            // Allocate hchan and buf in one call.
            c = (*hchan)(mallocgc(hchanSize+mem, nil, true))
            c.buf = add(unsafe.Pointer(c), hchanSize)
        default:
            // Elements contain pointers.
            c = new(hchan)
            c.buf = mallocgc(mem, elem, true)
    }
```

其中 c.buf = c.raceaddr()

# 一手资源+V 307570512

Plain Text

```
func (c *hchan) raceaddr() unsafe.Pointer {
    // Treat read-like and write-like operations on the channel to
    // happen at this address. Avoid using the address of qcount
    // or dataqsiz, because the len() and cap() builtins read
    // those addresses, and we don't want them racing with
    // operations like close().
    return unsafe.Pointer(&c.buf)
}
```

raceaddr 是做什么用的，不理解为什么需要这个方法

问题 2：当出现 channel 中元素类型为指针的时候为什么需要分开 hchan 和 buf

我的理解是：如何存放在一起 gc 扫描的时候不好扫

问题 3：当出现 channel 元素类型不为指针的时候，是为了避免 gc 吗？那么 gc 是如何知道说这个 channel 分配的内存该不该被扫描，对应的标识是在哪里？

我看都是调用的 mallocgc 方法，也没有什么区别，不知道这个标识在哪？

## 回答

1. 这个是在 build 的时候带 -race flag 才会触发的 race detect，在 channel 里 if raceenabled = true 的时候，会执行诸如 raceread 和 racewrite 之类的操作来检查是不是有 race，这部分需要有一个地址来做同步，race detector 的具体实现可以参考这里：<https://medium.com/a-journey-with-go/go-race-detector-with-threadsanitizer-8e497f9e42db>
2. 没有指针的时候，chan 里的对象在 GC 阶段应该是不需要扫描的；无指针放在一起，有指针分开存，这在 runtime 里面应该算是个通用设计，map 也是这样的
3. mallocgc 的时候，你看有指针和没指针，传进去的 typ 这个参数是不一样的，在 mallocgc 的时候，有一个 noscan 的判断：noscan := typ == nil || typ.ptrdata == 0，noscan 的话，后面 GC 就不对元素内的 field 做扫描了
4. 为什么打印 map 的时候会特意做排序。

## 问题

曹大，有没有了解过为啥 fmt 打印的时候会对 map 做字母顺序排序。看了代码，只有打印 map 的时候会自动排序，其他 array、slice 啥的都没有，不知道是作者的强迫症还是有啥特殊的想法hhh。。。

# 一手资源+V 307570512

```
case reflect.Map:
    if p fmt sharpV {
        p buf.writeString(f.Type().String())
        if f.NotNil() {
            p buf.writeString(nilParenString)
            return
        }
        p buf.writeByte('{')
    } else {
        p buf.writeString(mapString)
    }
    sorted := fmtsort.Sort(f)
    for i, key := range sorted.Key {
        if i > 0 {
            if p fmt sharpV {
                p buf.writeString(commaSpaceString)
            } else {
                p buf.writeByte(' ')
            }
        }
        p.printValue(key, verb, depth+1)
        p buf.writeByte(':')
        p.printValue(sorted.Value[i], verb, depth+1)
    }
    if p fmt sharpV {
        p buf.writeByte('}')
    } else {
        p buf.writeByte(']')
    }
case reflect.Struct:
```

## 回答

这个应该是最近的几个版本才改成按字母排序的，以前是随机的

我试了一下 1.9.2 是随机的

## 再次追问

发现如果单纯地 for range 的话，map 里面顺序是随机的，只是直接 print map 的话是单独做了个排序处理。感觉就是纯粹的强迫症

# 一手资源+V 307570512

## 回答

print 这个好像是网友吐槽太多， debug 不便特意改的

## 5. 超时的逻辑判断，或者是在哪里触发 ctx.Done 的，源码在哪里？**contextWithDeadline**

## 问题

context 源码里面没找到这个 timeout 的逻辑，是无形生成一个 timer，挂在 P 下面了吗

## 回答

```
Plain Text
func WithDeadline(parent Context, d time.Time) (Context, CancelFunc) {
    .....
    c.mu.Lock()
    defer c.mu.Unlock()
    if c.err == nil {
        ////////////// 这里，创建了一个 timer，超时后会回调，执行 c.cancel
        c.timer = time.AfterFunc(dur, func() {
            c.cancel(true, DeadlineExceeded)
        })
    }
    return c, func() { c.cancel(true, Canceled) }
}
```

## 再次追问

那是不是在 schedule 里面看到相对应超时的代码，应该就是 checkTimer 里面的逻辑，会调一个回调

## 回答

是的，checkTimers 里有

## 6. sync.Map 的使用问题

## 问题

课程中，曹大介绍了内置的 map 在并发的情况下会有问题。所以建议大家使用 sync.Map。今天在一个博客上看到说，sync.Map 也是有一些问题的

1. 它适合一次写入，多次共享读取的情况。我理解如果有共享，但是多个 goroutine 同时写，也有潜在风险，sync.Map 也是不合适的，对吧？
2. sync.Map 用的是 interface，所以还需要额外的对数据进行检查。

所以有些情况下，还是用 sync.RWMutex 来设置临界区的方式使用内置的 map。

我们在第三课的作业中也实现了一个自己的并发安全的 map 版本，我就是用的 mutex 的方式实现的，而且我在源码 1.14.4 中看到

```
Plain Text
```

```
```
type Map struct{
    mu Mutex
}
```

```

其实也是有 mutex 互斥量的，我理解只要在写的时候 mutex lock 了。就是安全的，无论多写还是一次写多次读的情况

想听听曹大的教导

## 回答

1. 多个 writer 没有风险，内部会加锁，只是如果写并发高的情况下，sync.Map 的效率可能还不如普通的 map + RWMutex，这个可以通过自己构造代码压测来验证，也可以跑 parallel benchmark
2. 并发安全问题可以通过 build -race 来构建出一个特殊版本，通过给服务发压，看看有没有 race 输出来检查是否有并发问题

之前课上我说建议用 sync.Map，主要是因为自己用 map 加锁，很容易写出没控制好临界区，导致死锁的代码

现在因为没有泛型，所以 sync.Map 的 api 全是用 interface，1.18 以后，这部分肯定会大改

## 7. 关于 struct 实现 interface 的问题

## 问题

目前看了源码和很多资料得出了以下结论，但感觉好像有点怪怪的。麻烦曹大纠错一下~

判断一个 struct 是否实现了 interface，在编译阶段会 Go 语言内部会隐式的进行转换工作。而这个转换工作是跟显式的接口转换类似，调用 runtime.convI2I() 方法。把这个 struct 的 \_type 赋值给了 iface.itab 内的 \_type。然后比较 iface.interfacetype.imethod 和 iface.func。

# 一手资源+V 307570512

## 回答

你上面给的函数应该是在运行阶段做的，不是编译阶段。编译阶段的话 grep 一下：

```
find ./ -name "*.go" | xargs grep "does not implement" | grep "(missing"
```

就会知道，判断逻辑是在 `cmd/compile/internal/gc/subr.go` 这个里面。

给一个例子：

```
Plain Text
```go
package main

import "io"

type myWriter struct {

}

/*func (w myWriter) Write(p []byte) (n int, err error) {
    return
}*/ 

func main() {
// 检查 *myWriter 类型是否实现了 io.Writer 接口
var _ io.Writer = (*myWriter)(nil)

// 检查 myWriter 类型是否实现了 io.Writer 接口
var _ io.Writer = myWriter{ }
}
```

```

编译它，会报错，根据报错信息，就可以找到具体的源码，再去分析下。

```
```sh
# command-line-arguments
./a.go:15:9: cannot use (*myWriter)(nil) (type *myWriter) as type io.Writer in assignment:
    *myWriter does not implement io.Writer (missing Write method)
./a.go:18:9: cannot use myWriter literal (type myWriter) as type io.Writer in assignment:
    myWriter does not implement io.Writer (missing Write method)
```

```

## 8. chansend 中 gp.param 相关疑问

### 问题

在 chansend 函数中，如果 chan 被阻塞，生成 sudog 得时候会将 gp.param 置为 nil，然后挂起当前 gp，当被唤醒之后，继续执行，这个时候会检查 gp.param 是否为 nil，如果为 nil 会 panic；否则将 gp.param 置为 nil。那么是在何处给 gp.param 赋值得呢（因为挂起之前将 gp.param 置为了 nil 了）？代码如下：

```
Swift
gp := getg()
mysg := acquireSudog()
mysg.releasetime = 0
if t0 != 0 {
    mysg.releasetime = -1
}
// No stack splits between assigning elem and enqueueing mysg
// on gp.waiting where copystack can find it.
mysg.elem = ep
mysg.waitlink = nil
mysg.g = gp
mysg.isSelect = false
mysg.c = c
gp.waiting = mysg
gp.param = nil
c.sendq.enqueue(mysg)
// Signal to anyone trying to shrink our stack that we're about
// to park on a channel. The window between when this G's status
// changes and when we set gp.activeStackChans is not safe for
// stack shrinking.
atomic.Store8(&gp.parkingOnChan, 1)
gopark(chanparkcommit, unsafe.Pointer(&c.lock), waitReasonChanSend,
traceEvGoBlockSend, 2)
// Ensure the value being sent is kept alive until the
// receiver copies it out. The sudog has a pointer to the
// stack object, but sudogs aren't considered as roots of the
// stack tracer.
KeepAlive(ep)

// someone woke us up.
if mysg != gp.waiting {
    throw("G waiting list is corrupted")
}
gp.waiting = nil
gp.activeStackChans = false
if gp.param == nil {
    if c.closed == 0 {
```

# 一手资源+V 307570512

```
    throw("chansend: spurious wakeup")
}
panic(plainError("send on closed channel"))
}
gp.param = nil
```

## 回答

在 chanrecv 里面：

```
Plain Text
if sg := c.sendq.dequeue(); sg != nil {
// Found a waiting sender. If buffer is size 0, receive value
// directly from sender. Otherwise, receive from head of queue
// and add sender's value to the tail of the queue (both map to
// the same buffer slot because the queue is full).
recv(c, sg, ep, func() { unlock(&c.lock) }, 3)
return true, true
}
```

然后 recv 里面

```
Plain Text
gp.param = unsafe.Pointer(sg)
```

[9. 对比 map/chan 想不通为何 makeslice 返回一个可转换的指针？](#)

## 问题

- makechan 返回 hchan 结构体实例的指针

# 一手资源+V 307570512

```
71 func makechan(t *chantype, size int) *hchan {
72     elem := t.elem
73
74     // compiler checks this but be safe.
75     if elem.size >= 1<<16 {
76         throw( s: "makechan: invalid channel element type")
77     }
78     if hchanSize%maxAlign != 0 || elem.align > maxAlign {
79         throw( s: "makechan: bad alignment")
80     }
81
82     mem, overflow := math.MulUintptr(elem.size, uintptr(size)
83     if overflow || mem > maxAlloc-hchanSize || size < 0 : pl
- makemap 返回 hmap 结构体实例的指针
```

```
303 func makemap(t *maptype, hint int, h *hmap) *hmap {
304     mem, overflow := math.MulUintptr(uintptr(hint), t.bucket.size)
305     if overflow || mem > maxAlloc {
306         hint = 0
307     }
308
309     // initialize Hmap
310     if h == nil {
311         h = new(hmap)
312     }
313     h.hash0 = fastrand()
314
```

- makeslice 为何返回这个

## Plain Text

unsafe.Pointer 不是 type slice 的一个实例的指针? 不知道有什么特殊的用意

```
83 func makeslice(et *_type, len, cap int) unsafe.Pointer {
84     mem, overflow := math.MulUintptr(et.size, uintptr(cap))
85     if overflow || mem > maxAlloc || len < 0 || len > cap {
86         // NOTE: Produce a 'len out of range' error instead of a
87         // 'cap out of range' error when someone does make([]T, bignum).
88         // 'cap out of range' is true too, but since the cap is only being
89         // supplied implicitly, saying len is clearer.
90         // See golang.org/issue/4085.
91         mem, overflow := math.MulUintptr(et.size, uintptr(len))
92         if overflow || mem > maxAlloc || len < 0 {
93             panicmakeslicelen()
94         }
95         panicmakeslicecap()
96     }
97
98     return mallocgc(mem, et, needzero: true)
99 }
```

# 一手资源+V 307570512

## 回答

slice 和其它结构这种算是语义不太一致的情况。。。

比如你给一个结构体传内置数据结构，大多数相当于是传个指针进去，

slice 则相当于传了 addr, len, cap 这三个字段进去

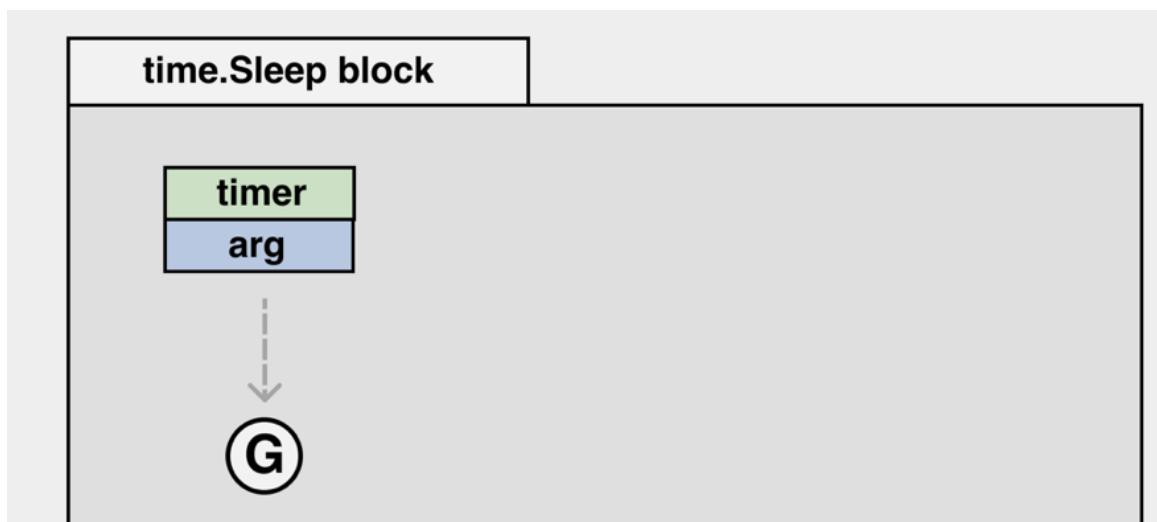
个人觉得单纯是没设计好。。

## [10. time.Sleep 也会把 G 放到 4 叉堆上嘛](#)

## 问题

老师,想问一下 time.sleep 也会把 G 放到 4 叉堆上嘛

## 回答



timer 这个结构会被放在四叉堆，

goroutine 是 g 结构，放在 timer.arg 这个字段里，然后执行 gopark 挂起这个 g，等待唤醒

等 timer 时间到了用 arg 能找到 g  
执行 goready 唤醒 g 继续执行

这个流程在 `runtime/time.go` 下的 `timeSleep` 函数里

## 11. `mapdelete` 为什么还需要搬运？

### 问题

相关截图：

**Map-元素操作**

---

扩容中

- `mapassign`: 将命中的 bucket 从 `oldbuckets` 顺手搬运到 `buckets` 中，顺便再多搬运一个 bucket
- `mapdelete`: 将命中的 bucket 从 `oldbuckets` 顺手搬运到 `buckets` 中，顺便再多搬运一个 bucket
- `mapaccess`: 优先在 `oldbuckets` 中找，如果命中，则说明这个 bucket 没有被搬运

搬运 bucket x 时，会被该桶的 overflow 桶也一并搬完

`mapdelete` 操作直接从 `oldbuckets` 删除元素就可以了啊？为什么还要从 `oldbucket` 搬运到新的 bucket？不是多此一举吗？

### 回答

数据结构：

读：读

写：写入新值，更新旧值、删除旧值

可以认为更新和删除也是写入操作~

## 12. 用 `slice` 作为栈使用，久而久之是否空间会占很多？

### 问题

曹大，有个疑问确认下：slice 栈的扩展问题利用 slice 底层自动扩容，但是有个问题，是不是 slice 越来越大是不是没法缩容，所占空间越来越多，比如我们的出栈操作：`v := sl[len(sl)-1]; sl = sl[0:len(sl)-1]`。但是实际底层上面还是存在一个很大数组。解决办法是在

# 一手资源+V 307570512

一定的时候重新创建 slice 吗？

## 回答

确实没法缩容，  
所以对 slice 进行复用的时候(比如 sync.Pool)，很多时候也要判断要放的这个 slice 是不是  
太大了  
这个在 runtime 里就有一些例子~

[13. 可以提供下 string array slice 相关的学习资料吗？](#)

## 问题

1. 没有讲 string, array 和 slice? 有相关的动画和资料吗?
2. makeslice 结构返回一个 unsafe.ptr, 而其他 chan map 等都返回对应结构实例的指针, 不清楚有啥用意?

## 回答

这几个结构都比较简单，string 相关的都在 runtime/string.go 里(以及 strconv, strings, bytes 库)，slice 都在 runtime/slice.go 里

slice 的 api 设计的不好，之前有人说过 Go 这里的拼凑痕迹很重

比如这种 bug:

```
Plain Text
func addElem(sl []int) {
    sl = append(sl, 1)
}
```

还是出现的比较多的

[14. same size grow 等大扩容的含义？](#)

## 问题

老师，我理解的等大扩容是通过一些处理方法将 overflow bucket 的 key 移动到 bucket 中来，

# 一手资源+V 307570512

节省空间提高 bucket 的利用率，使 key 的排布更加紧密提高查询效率，实际上并没有对 map 进行扩容操作，这样理解对吗？

## 回答

你的理解没错，map 的 bucket 数(除 overflow 外)没有变化，不过他还是得走扩容流程做个元素的整理

## 15. channel 有锁和无锁的区别

### 问题

老师您好，我看您在问答区回答问题提到，无锁设计比有锁耗费 CPU，我没明白有没有锁和 CPU 有什么关系？

### 回答

无锁算法一般都有自旋，比如调度里的一些逻辑，本质都是 for 循环套 cas 一直尝试在多核竞争激烈的情况下会消耗很多 CPU，你们有压测系统的话可以观察一下不同 goroutine 数量下几个调度的关键函数的 CPU 消耗

有锁的设计基本都是阻塞+等待。

## 16. 在 Map 中根据 key 取值的时候，其中一个步骤使用了 bucketMask，那么它是什么？

### 问题

go 的 map 中，通过 key 去 map 里面取值的时候，步骤为：

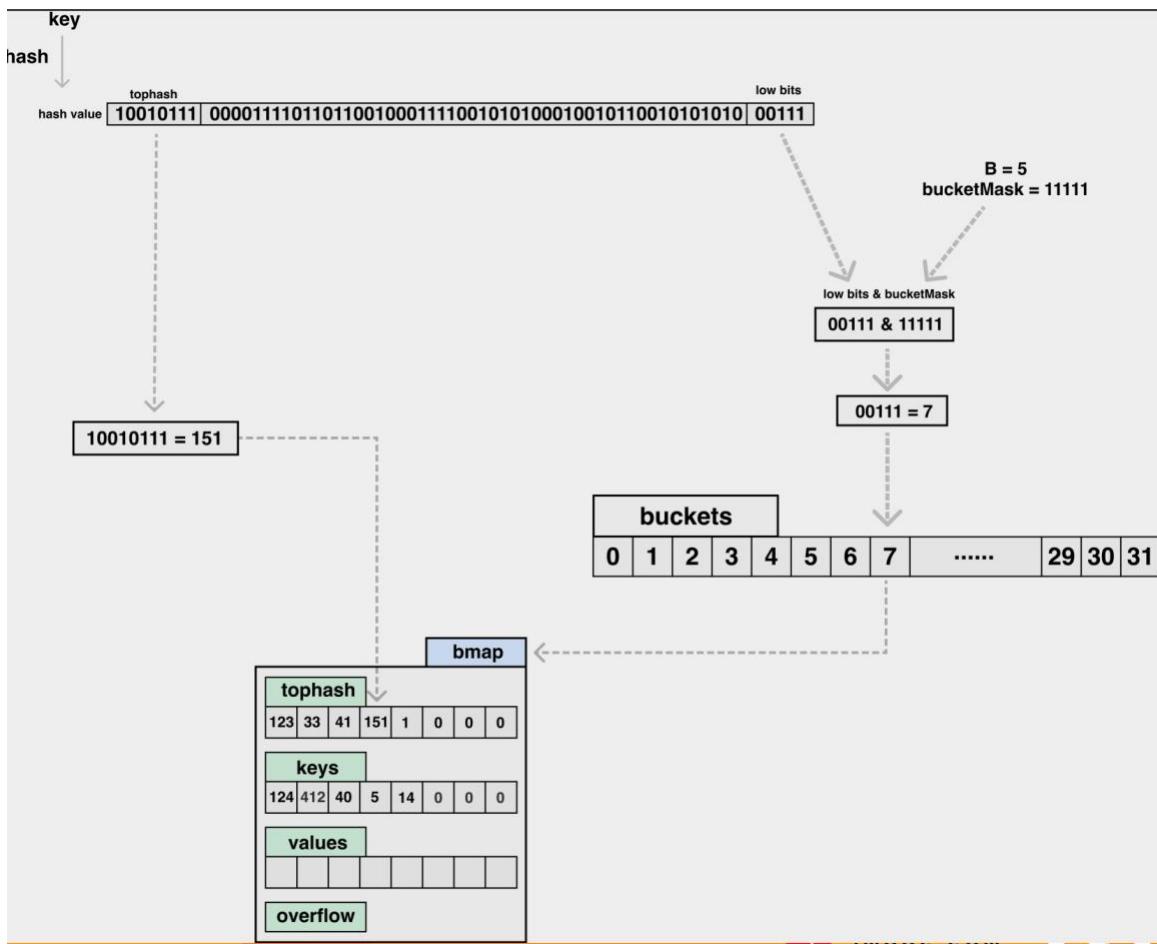
1、将 key 做 hash 运算，得到二进制串

2、取 low bits (不知道是啥，只知道是后 5 位)，和 bucketMask 做位运算，即：

low bits & bucketMask (00111&11111 得到 00111 即 7，位置为 7 的桶)

3、确定了桶的位置是 7 后，取 hmap 的 buckets 数组的 7 位置 (即第 8 个桶) 那里，再将 hash 值的 tophash (前 8 位，计算其 10 进制的值)，对比 bmap 里头的 tophash，得到第 4 个位置，然后对比 keys 里面第四个位置是否和 key 相等，相等则取 values 第四个位置的值出来即可。不想等，则代表这个 key 在这个 map 里面没有值。

# 一手资源+V 307570512



问题：

1、课件中给的 bucketMask 是 11111，那么这个 bucketMask 是什么呢？图中画了  $B=5$ ， $B$  数值在这一步计算中起什么作用呢？

2、上面是我理解的取值步骤，是否正确呢？

谢谢老师

## 回答

$11111 = 2^5 - 1$

后面选桶的时候就用到这个 bucketMask 了

## [17. Map 如何查看容量大小](#)

## 问题

上课提到了 Map 无法扩容，并给出了这个例子让我们测试：

# 一手资源+V 307570512

Plain Text

```
package main

var m = make(map[int]int)

func main() {
    for i := 0; i < 10000000; i++ {
        m[i] = i
    }

    for i := 0; i < 990000; i++ {
        delete(m, i)
    }
}
```

那么如何查看 Map 的 buckets 的长度，或者拿到 Map 的容量呢？试了下反射好像也拿不到，直接 unsafe.Sizeof() 好像也只拿到了指针的大小...

( Python 程序员老想着 Debug 的时候直接访问这个对象的私有变量... 可能这里不是直接在 debug 的时候去看？而是去写一个 benchmark 之类的？ )

## 回答

可以在 delete 后面加一个 map 访问，断点设置过去，然后再 p map.B  
但是后面还有 overflow 呢

## 再次追问

我又搞了一个这样的 demo

Plain Text

```
package main

import "fmt"

func main() {

    dict := make(map[int]int, 10)

    a := dict[0]
    fmt.Println(a)

    for i := 0; i < 10000; i++ {
        dict[i] = i
    }
}
```

# 一手资源+V 307570512

```
a = dict[0]
fmt.Println(a)

for i := 0; i < 10000; i++ {
    delete(dict, i)
}

a = dict[0]
fmt.Println(a)
}
```

并在每个 `a:= dict[0]` 打了断点。然后出现了这样的错误。。。

Plain Text

```
```bash
(dlv) b test_map_grow.go:9
Breakpoint 1 (enabled) set at 0x4ae873 for main.main() ./test_map_grow.go:9
(dlv) b test_map_grow.go:16
Breakpoint 2 (enabled) set at 0x4ae987 for main.main() ./test_map_grow.go:16
(dlv) b test_map_grow.go:23
Breakpoint 3 (enabled) set at 0x4aea92 for main.main() ./test_map_grow.go:23
(dlv) c
> main.main() ./test_map_grow.go:9 (hits goroutine(1):1 total:1) (PC: 0x4ae873)
 4:
 5: func main() {
 6:
 7:     dict := make(map[int]int, 10)
 8:
=> 9:     a := dict[0]
 10:    fmt.Println(a)
 11:
 12:    for i := 0; i < 10000; i++ {
 13:        dict[i] = i
 14:    }
(dlv) p dict.B
Command failed: dict (type map[int]int) is not a struct
```

```

## 回答

在外面不行，我给你撸个例子

Plain Text

```
Breakpoint 3 set at 0x1066803 for main.main() ./mmm.go:19
(dlv) c
```

# 一手资源+V 307570512

```
> main.main() ./mmm.go:19 (hits goroutine(1):1 total:1) (PC: 0x1066803)
14:
15:    for i := 0; i < 9999999; i++ {
16:        delete(m, i)
17:    }
18:
=> 19:    m[1] = 1
20:    time.Sleep(time.Hour)
21: }
(dlv) disass
TEXT main.main(SB) /Users/xargin/test/mmm.go
    mmm.go:9      0x1066730  65488b0c2530000000  mov rcx, qword ptr
gs:[0x30]
    mmm.go:9      0x1066739  483b6110      cmp rsp, qword ptr [rcx+0x10]
    mmm.go:9      0x106673d  0f8615010000  jbe 0x1066858
    mmm.go:9      0x1066743* 4883ec48      sub rsp, 0x48
    mmm.go:9      0x1066747  48896c2440    mov qword ptr [rsp+0x40], rbp
    mmm.go:9      0x106674c  488d6c2440    lea rbp, ptr [rsp+0x40]
    mmm.go:10     0x1066751  48c744242800000000  mov qword ptr
[rsp+0x28], 0x0
    mmm.go:10     0x106675a  eb00          jmp 0x106675c
    mmm.go:10     0x106675c  48817c242880969800  cmp qword ptr
[rsp+0x28], 0x989680
    mmm.go:10     0x1066765  7c02          jl 0x1066769
    mmm.go:10     0x1066767  eb4b          jmp 0x10667b4
    mmm.go:11     0x1066769  488d0510d50000  lea rax, ptr [rip+0xd510]
    mmm.go:11     0x1066770  48890424      mov qword ptr [rsp], rax
    mmm.go:11     0x1066774  488b0d4db30700  mov rcx, qword ptr
[main.m]
    mmm.go:11     0x106677b  48894c2408    mov qword ptr [rsp+0x8], rcx
    mmm.go:11     0x1066780  488b4c2428    mov rcx, qword ptr
[rsp+0x28]
    mmm.go:11     0x1066785  48894c2410    mov qword ptr [rsp+0x10],
rcx
    mmm.go:11     0x106678a  e8618afaff    call $runtime.mapassign_fast64
    mmm.go:11     0x106678f  488b442418    mov rax, qword ptr
[rsp+0x18]
    mmm.go:11     0x1066794  4889442438    mov qword ptr [rsp+0x38],
rax
    mmm.go:11     0x1066799  8400          test byte ptr [rax], al
    mmm.go:11     0x106679b  488b4c2428    mov rcx, qword ptr
[rsp+0x28]
    mmm.go:11     0x10667a0  488908          mov qword ptr [rax], rcx
    mmm.go:11     0x10667a3  eb00          jmp 0x10667a5
    mmm.go:10     0x10667a5  488b442428    mov rax, qword ptr
[rsp+0x28]
    mmm.go:10     0x10667aa  48ffc0          inc rax
    mmm.go:10     0x10667ad  4889442428    mov qword ptr [rsp+0x28],
rax
    mmm.go:10     0x10667b2  eba8          jmp 0x106675c
    mmm.go:15     0x10667b4* 48c744242800000000  mov qword ptr
```

# 一手资源+V 307570512

|  |            |                      |                                 |
|--|------------|----------------------|---------------------------------|
| [rsp+0x20], 0x0  |            |                      |                                 |
| mmm.go:15  | 0x10667bd  | eb00                 | jmp 0x10667bf                   |
| mmm.go:15  | 0x10667bf  | 48817c24207f969800   | cmp qword ptr                   |
| [rsp+0x20], 0x98967f   |            |                      |                                 |
| mmm.go:15  | 0x10667c8  | 7c02                 | jl 0x10667cc                    |
| mmm.go:15  | 0x10667ca  | eb37                 | jmp 0x1066803                   |
| mmm.go:16  | 0x10667cc  | 488d05add40000       | lea rax, ptr [rip+0xd4ad]       |
| mmm.go:16  | 0x10667d3  | 48890424             | mov qword ptr [rsp], rax        |
| mmm.go:16  | 0x10667d7  | 488b0deab20700       | mov rcx, qword ptr              |
| [main.m]   |            |                      |                                 |
| mmm.go:16  | 0x10667de  | 48894c2408           | mov qword ptr [rsp+0x8], rcx    |
| mmm.go:16  | 0x10667e3  | 488b4c2420           | mov rcx, qword ptr              |
| [rsp+0x20]   |            |                      |                                 |
| mmm.go:16  | 0x10667e8  | 48894c2410           | mov qword ptr [rsp+0x10],       |
| rcx  |            |                      |                                 |
| mmm.go:16  | 0x10667ed  | e87e8efaff           | call \$runtime.mapdelete_fast64 |
| mmm.go:16  | 0x10667f2  | eb00                 | jmp 0x10667f4                   |
| mmm.go:15  | 0x10667f4  | 488b442420           | mov rax, qword ptr              |
| [rsp+0x20]   |            |                      |                                 |
| mmm.go:15  | 0x10667f9  | 48ffc0               | inc rax                         |
| mmm.go:15  | 0x10667fc  | 4889442420           | mov qword ptr [rsp+0x20], rax   |
| mmm.go:15  | 0x1066801  | ebbc                 | jmp 0x10667bf                   |
| =>    mmm.go:19  | 0x1066803* | 488d0576d40000       | lea rax, ptr                    |
| [rip+0xd476]   |            |                      |                                 |
| mmm.go:19  | 0x106680a  | 48890424             | mov qword ptr [rsp], rax        |
| mmm.go:19  | 0x106680e  | 488b05b3b20700       | mov rax, qword ptr              |
| [main.m]   |            |                      |                                 |
| mmm.go:19  | 0x1066815  | 4889442408           | mov qword ptr [rsp+0x8], rax    |
| mmm.go:19  | 0x106681a  | 48c744241001000000   | mov qword ptr                   |
| [rsp+0x10], 0x1  |            |                      |                                 |
| mmm.go:19  | 0x1066823  | e8c889faff           | call \$runtime.mapassign_fast64 |
| mmm.go:19  | 0x1066828  | 488b442418           | mov rax, qword ptr              |
| [rsp+0x18]   |            |                      |                                 |
| mmm.go:19  | 0x106682d  | 4889442430           | mov qword ptr [rsp+0x30],       |
| rax  |            |                      |                                 |
| mmm.go:19  | 0x1066832  | 8400                 | test byte ptr [rax], al         |
| mmm.go:19  | 0x1066834  | 48c70001000000       | mov qword ptr [rax], 0x1        |
| mmm.go:20  | 0x106683b  | 48b800a0b83046030000 | mov rax,                        |
| 0x34630b8a000  |            |                      |                                 |
| mmm.go:20  | 0x1066845  | 48890424             | mov qword ptr [rsp], rax        |
| mmm.go:20  | 0x1066849  | e8a262feff           | call \$time.Sleep               |
| mmm.go:21  | 0x106684e  | 488b6c2440           | mov rbp, qword ptr              |
| [rsp+0x40]   |            |                      |                                 |
| mmm.go:21  | 0x1066853  | 4883c448             | add rsp, 0x48                   |
| mmm.go:21  | 0x1066857  | c3                   | ret                             |
| mmm.go:9   | 0x1066858  | e8e348ffff           | call \$runtime.morestack_noctxt |
| < autogenerated >:1  | 0x106685d  | e9cefefeff           | jmp \$main.main                 |
| (dlv) b runtime.mapassign_fast64                             |            |                      |                                 |
| Breakpoint 4 set at 0x100f203 for runtime.mapassign_fast64() |            |                      |                                 |
| /usr/local/go/src/runtime/map_fast64.go:92                   |            |                      |                                 |
| (dlv) c  |            |                      |                                 |

# 一手资源+V 307570512

```
> runtime.mapassign_fast64() /usr/local/go/src/runtime/map_fast64.go:92 (hits goroutine(1):1
total:1) (PC: 0x100f203)
Warning: debugging optimized function
 87:         }
 88:     }
 89:     return unsafe.Pointer(&zeroVal[0]), false
 90:   }
 91:
=> 92: func mapassign_fast64(t *maptpe, h *hmap, key uint64) unsafe.Pointer {
 93:     if h == nil {
 94:         panic(plainError("assignment to entry in nil map"))
 95:     }
 96:     if raceenabled {
 97:         callerpc := getcallerpc()
(dlv) n
> runtime.mapassign_fast64() /usr/local/go/src/runtime/map_fast64.go:93 (PC: 0x100f211)
Warning: debugging optimized function
 88:         }
 89:     return unsafe.Pointer(&zeroVal[0]), false
 90:   }
 91:
 92: func mapassign_fast64(t *maptpe, h *hmap, key uint64) unsafe.Pointer {
=> 93:     if h == nil {
 94:         panic(plainError("assignment to entry in nil map"))
 95:     }
 96:     if raceenabled {
 97:         callerpc := getcallerpc()
 98:         racewritepc(unsafe.Pointer(h), callerpc, funcPC(mapassign_fast64))
(dlv) p h.B
21
```

## 18. Mutex unlock 问题

### 问题

为什么饥饿状态下 unlockSlow 直接返回？

或者说，在下图 unlockSlow 直接返回的情况下，队列中的 goroutine 如何获取锁（激活）？

# 一手资源+V 307570512

```
func (m *Mutex) unlockSlow(new int32) {
    if (new&mutexLocked)&mutexLocked == 0 {
        throw("sync: unlock of unlocked mutex")
    }
    if new&mutexStarving == 0 {
        old := new
        for {
            // If there are no waiters or a goroutine has already
            // been woken or grabbed the lock, no need to wake anyone.
            // In starvation mode ownership is directly handed off from unlocking
            // goroutine to the next waiter. We are not part of this chain,
            // since we did not observe mutexStarving when we unlocked the mutex above.
            // So get off the way.
            if old>>mutexWaiterShift == 0 || old&(mutexLocked|mutexWoken|mutexStarving) != 0 {
                return
            }
            // Grab the right to wake someone.
            new = (old - 1<<mutexWaiterShift) | mutexWoken
            if atomic.CompareAndSwapInt32(&m.state, old, new) {
                runtime_Semrelease(&m.sema, false, 1)
                return
            }
            old = m.state
        }
    } else {
    }
}
```

## 回答

这是因为这个 unlock 的 goroutine 一开始认为 new & mutexStarving == 0，是非饥饿模式，但是进入 if 以后再检查发现已经有别人抢到了锁/或者锁进入了饥饿模式，这说明已经有其它的 goroutine 把队列中等待的 g 唤醒了，这种情况下，控制权应该在新的 goroutine 那边，自己就不需要再做一次唤醒了

## 19. 关于 Mutex 问题

### 问题

在网上看到一些文章与视频的描述有些不解

Q1：对于唤醒标识 mutexWoken，“当前 goroutine 持有唤醒标识”是指当前 goroutine 所在的栈中

`awoke`临时变量为 true 或是`new`临时变量的 mutexWoken 位为 1 么？

Q2：对于唤醒标识 mutexWoken，“当前 goroutine 都不再是被唤醒的 goroutine”，goroutine 是被谁唤醒的？如果当前执行 goroutine 不是从 mutex 的阻塞队列中来的而是新来的、直接进行自旋的 goroutine 的话，那是不是“当前 goroutine 都不再是醒着的 goroutine”更合语境？

PS：问的地方有不对的请指正下谢谢=。。=

# 一手资源+V 307570512

PPS: 疑问出处 <https://www.bilibili.com/video/BV15V411n7fM?p=3>  
03:21“当前 goroutine 都不再是被唤醒的 goroutine”， 03:04 “当前 goroutine 持有唤醒标识”，

## 回答

Q1: 对于唤醒标识 mutexWoken, “当前 goroutine 持有唤醒标识”是指当前 goroutine 所在的栈中

`awoke`临时变量为 true 或是`new`临时变量的 mutexWoken 位为 1 么？

我个人感觉他说的是指当前的 goroutine 把 mutexWoken 置一成功，就算是持有了。

Q2: 对于唤醒标识 mutexWoken, “当前 goroutine 都不再是被唤醒的 goroutine”， goroutine 是被谁唤醒的？如果当前执行 goroutine 不是从 mutex 的阻塞队列中来的而是新来的、直接进行自旋的 goroutine 的话，那是不是“当前 goroutine 都不再是醒着的 goroutine”更合语境？  
goroutine 唤醒主要是 unlock 流程~

“当前 goroutine 都不再是醒着的 goroutine”

这里释放标识是因为当前的 goroutine 已经上锁或者进队列了，这时候需要能让别人去抢这个 mutexWoken 标识位。她视频里说的因为所以我觉得比较牵强

PS: 问的地方有不对的请指正下谢谢=。。=

PPS: 疑问出处 <https://www.bilibili.com/video/BV15V411n7fM?p=3>

03:21“当前 goroutine 都不再是被唤醒的 goroutine”， 03:04 “当前 goroutine 持有唤醒标识”，

## 20. Map 的哈希过程遇到的问题

### 问题

不知道我对把 key hash 以后找具体 bucket 下的 tophash 理解有没有问题，如果有问题麻烦曹大也帮忙纠错一下：

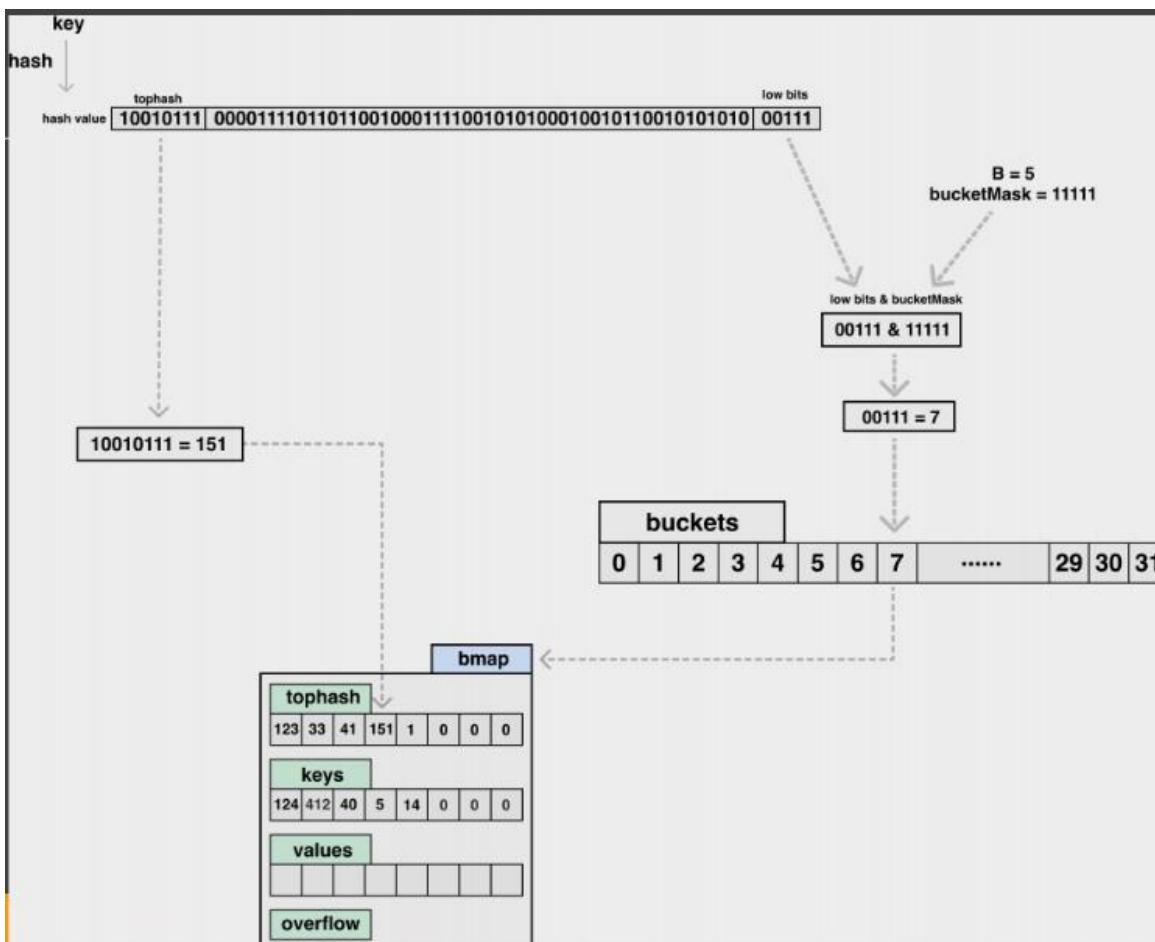
首先通过哈希函数计算 key，得到一个 hash value，然后取二进制位的后 5 位(low bits)，和 bucketMask 进行 & 运算，得到的就是 bucket 所在位置。然后取 hash value 二进制位的头八位转化为十进制，存入对应 bucket bmap 的 tophash 数组里。先判断当前 tophash 对应的 key 是否为空，如果为空该位置就是对应位置，如果不为空，则表明发生哈希冲突，需要使用拉链法在该 tophash 后链一个新的 tophash-key-value 结构

问题一：

找到了对应 bucket, bmap 结构中的 tophash 的位置是如何确定的？

问题二：

发生哈希冲突时，使用拉链法链的新结构是个什么结构？ bmap 里的 tophash、 keys、 values 不都是长度为 8 的数组嘛？难道要链一个新的 tophash-key-value 结构？



## 回答

问题一：

找到了对应 bucket, bmap 结构中的 tophash 的位置是如何确定的？

要遍历，一个 bucket 里面最多有 8 个 entry，要把这 8 个 entry 的 tophash[i] 遍历完，如果 tophash[i] = 输入 key 的 tophash，那么需要再比较实际的 key 是否等于输入的 key，如果相等，这时候覆盖掉第 i 个位置的 value

如果一直没找到相等的，则需要把 overflow 都遍历完毕，全完毕之后都没有，说明需要插入到第一次的 empty entry 的位置上(这个在遍历一开始就记录过了)

问题二：

发生哈希冲突时，使用拉链法链的新结构是个什么结构？bmap 里的 tophash、keys、values 不都是长度为 8 的数组嘛？难道要链一个新的 tophash-key-value 结构？  
就是我们课上那个图的样子，overflow 指针指向一个新的 bmap。bmap 里面还是有 tophash、

# 一手资源+V 307570512

keys、values

## 21. chan.go recv 函数的最后一个参数

### 问题

Plain Text

```
if sg := c.sendq.dequeue(); sg != nil {  
    // Found a waiting sender. If buffer is size 0, receive value  
    // directly from sender. Otherwise, receive from head of queue  
    // and add sender's value to the tail of the queue (both map to  
    // the same buffer slot because the queue is full).  
    recv(c, sg, ep, func() { unlock(&c.lock) }, 3)  
    return true, true  
}
```

上面这段代码摘自 chan.go，chanrecv 函数内对于 sendq 不为空的情况，请问 recv 的最后那个 3 是什么意思

### 回答

这个是给 trace 或者 pprof 的工具用的，

这些工具会记录一些调用栈，但没必要把 runtime 中所有的栈全打印出来(skip 一般指的就是省略几层栈之类的意思)，所以中间会跳掉一些

不了解 trace 和 pprof 的话，可以先认为这个参数没啥用

## 22. 关于 Map 数据结构的 bmap 结构

### 问题

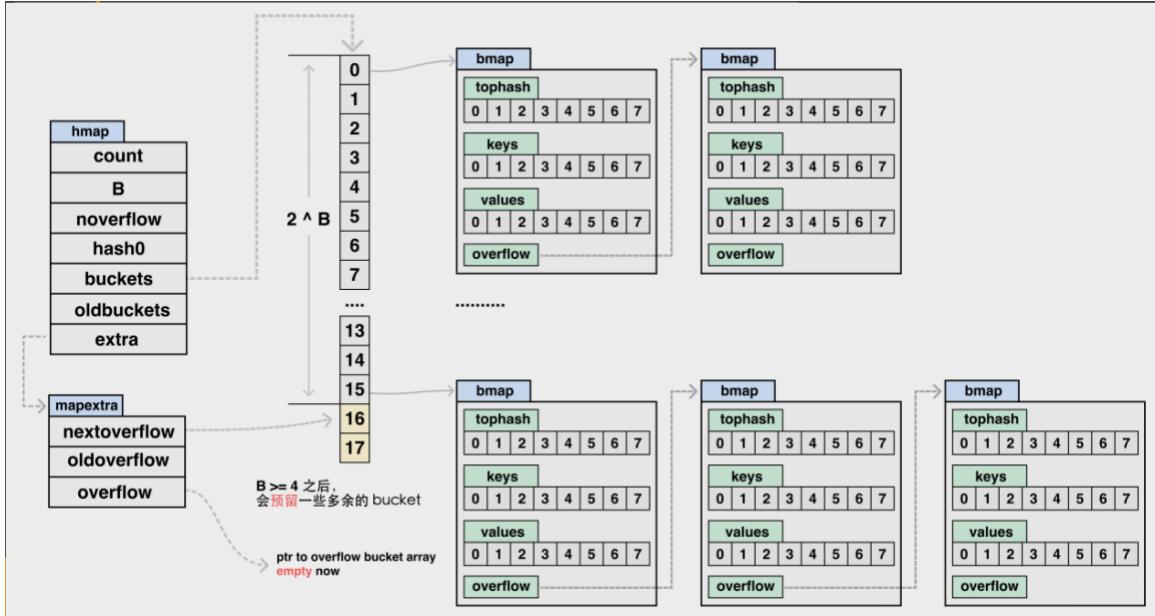
根据课件里图表明 bmap 应该是 hmap 里 buckets 的每一个 bucket 的数据结构，但是这个 bmap 里只有一个 tophash 字段，课件上的结构图里这些 keys、values 和 overflow 是怎么来的？

Plain Text

```
// A bucket for a Go map.  
type bmap struct {  
    // tophash generally contains the top byte of the hash value  
    // for each key in this bucket. If tophash[0] < minTopHash,  
    // tophash[0] is a bucket evacuation state instead.  
    tophash [bucketCnt]uint8  
    // Followed by bucketCnt keys and then bucketCnt elems.  
    // NOTE: packing all the keys together and then all the elems together makes the
```

# 一手资源+V 307570512

```
// code a bit more complicated than alternating key/elem/key/elem/... but it allows  
// us to eliminate padding which would be needed for, e.g., map[int64]int8.  
// Followed by an overflow pointer.  
}
```



## 回答

bmap 里面的 key、value 和 overflow 指针都是通过 unsafe 来访问的

我个人猜测可能是没有泛型，所以才这么恶心的

## 23. type switch 问题

### 问题

```
Plain Text  
package main  
  
import "fmt"  
  
type Person interface {  
    sayHello(name string) string  
    sayGoodbye(name string) string  
}  
  
type Student struct {
```

# 一手资源+V 307570512

```
name string
}

//go:noinline
func (s *Student) sayGoodbye(name string) string {
    fmt.Printf("%s goodbye to %s \n", s.name, name)
    return fmt.Sprintf("%v: Hi %v, see you next time.\n", s.name, name)
}

//go:noinline
func (s *Student) sayHello(name string) string {
    fmt.Printf("%s: to %s \n", s.name, name)
    return fmt.Sprintf("%v: Hello %v, nice to meet you.\n", s.name, name)
}

func main() {
    var s Person = &Student{name: "halfrost"}
    switch s.(type) {
    case Person:
        person := s.(Person)
        person.sayHello("everyone")
        person.sayGoodbye("everyone")
        fmt.Println("person")
    // error here
    case Student:
        student := s.(Student)
        student.sayHello("everyone")
        student.sayGoodbye("everyone")
        fmt.Println("student")
    //
    case *Student:
        student := s.(*Student)
        student.sayHello("everyone")
        student.sayGoodbye("everyone")
        fmt.Println("*student")

    }
}
"""

# command-line-arguments
.\interface_implement.go:27:6: cannot use Student{...} (type Student) as type Person in
assignment:
    Student does not implement Person (sayGoodbye method has pointer receiver)
.\interface_implement.go:34:2: impossible type switch case: s (type Person) cannot have
dynamic type Student (sayGoodbye method has pointer receiver)
.\interface_implement.go:35:15: impossible type assertion:
    Student does not implement Person (sayGoodbye method has pointer receiver)
"""

```

Q1: 这段报错的原因

# 一手资源+V 307570512

Q2: 为什么这么设计，类型也有“实现”的概念么？

## Type assertions

For an expression `x` of interface type and a type `T`, the primary expression

`x. (T)`

asserts that `x` is not `nil` and that the value stored in `x` is of type `T`. The notation `x. (T)` is called a *type assertion*.

More precisely, if `T` is not an interface type, `x. (T)` asserts that the dynamic type of `x` is identical to the type `T`. In this case, `T` must implement the (interface) type of `x`; otherwise the type assertion is invalid since it is not possible for `x` to store a value of type `T`. If `T` is an interface type, `x. (T)` asserts that the dynamic type of `x` implements the interface `T`.

If the type assertion holds, the value of the expression is the value stored in `x` and its type is `T`. If the type

| Methods Receivers   | Values                             |
|---------------------|------------------------------------|
| <code>(t T)</code>  | <code>T</code> and <code>*T</code> |
| <code>(t *T)</code> | <code>*T</code>                    |

## 回答

PS：群里讨论了，也得到了启发，想着分享一下放问答区

PPS： demo 来自 [https://halfrost.com/go\\_interface/#toc-4](https://halfrost.com/go_interface/#toc-4)

## 24. 四叉堆是什么，以及在 timer 中如何使用的？

## 问题

有没有介绍四叉堆的文章？

## 回答

和二叉堆类似，只不过有四个分叉

英文区搜 n-ary tree 之类的应该可以搜到

老版本四叉堆 timer 实现我之前写过一篇文章介绍，

<https://github.com/cch123/golang-notes/blob/master/timer.md>

## 25. 关于 bmap

### 问题

Key 经过哈希得到的高 8 位值是怎么确定放入 topbits 数组中对应的哪个下标位置的。这个你说的哈希冲突指的是低位还是这个的高 8 位

### 回答

lowbits 是用来选 bucket 的

topbits 排序应该没什么讲究，放不下就往里面塞，放不下就挂链表 bucket

哈希冲突，其实按 Go 这个 map 设计的话，同一个 bucket 里都是 hash 冲突的。。。

## 26. 复杂 struct 的字面值初始化，为什么 goland 老是报错啊？

### 问题

相关代码:

```
Plain Text
type foo1 struct{
    foo1 string
}

type foo2 struct{
    foo2 string
}

type mydata struct{
    md struct{
        foo1
        foo2
    }
}
```

为什么以字面值初始化的时候，会报错类型识别不了：

相关代码:

```
Plain Text
MD := mydata{
    md: struct{
        foo1
        foo2
    }
}
```

# 一手资源+V 307570512

```
}{  
foo1:foo1{"test"},  
foo2:foo2{"test"}  
}  
}
```

但是有些时候又不会报错

## 回答

你这太考眼力了，最好不要搞这么复杂的写法。

一般出错了，不要慌，好好读下报错信息，哪里有错改哪里。

---

具体到这个问题，有两处 typos：

The screenshot shows a Go code editor with syntax highlighting. Two specific errors are highlighted with red boxes:

- A red box highlights the word "struct" in the line "md struct{", with the annotation "你写成了 struct" (You wrote it as struct) in red.
- A red box highlights the letter "I" in the identifier "foo1", with the annotation "你写成了 I (L的小写)" (You wrote it as I (L's lowercase)) in red.

```
type foo2 struct{  
    fooo2 string  
}  
  
type mydata struct{  
    md struct{  
        foo1  
        foo2  
    }  
}  
  
func main() {  
    MD := mydata{  
        md: struct{  
            foo1  
            foo2  
        }{  
            foo1:foo1, fooo1: "test"},  
            foo2:foo2{ fooo2: "test"},  
        },  
    }  
    fmt.Println(MD)  
}
```

# 一手资源+V 307570512

## 再次追问

老师不好意思，我昨天在家里没有 ide，只能试着回想一下我的结构体是怎么构造的，现在贴上源码：

```
teststring1 := UpData{
    Sp: struct {
        ESBASE
        DocPlusID `json:"doc"`
    } struct {
        ESBASE
        DocPlusID
    }{ESBASE: ESBASE{Index: Index, Type: Type}, DocPlusID: DocPlusID{Id: "1"}}),
}
```

是这样的！

为什么这里我已经声明了字面值，却还是要做一次类型转换？

## 回答

因为嵌套结构初始化就是比较麻烦的

有不少语言都是不支持这样嵌套初始化的~

## [27. hmap 中的 B 是怎么初始化的？](#)

## 问题

hmap 中的 B 是怎么初始化的？

## 回答

# 一手资源+V 307570512

```
57
58     // makeBucketArray initializes a backing array for map buckets.
59     // 1<<b is the minimum number of buckets to allocate.
60     // dirtyalloc should either be nil or a bucket array previously
61     // allocated by makeBucketArray with the same t and b parameters
62     // If dirtyalloc is nil a new backing array will be allocoed and
63     // otherwise dirtyalloc will be cleared and reused as backing ar
64 func makeBucketArray(t *maptype, b uint8, dirtyalloc unsafe.Pointer)
65     base := bucketShift(b)
66     nbuckets := base
67     // For small b, overflow buckets are unlikely.
68     // Avoid the overhead of the calculation.
69     if b >= 4 {
70         // Add on the estimated number of overflow buckets
71         // required to insert the median number of elements
72         // used with this value of b.
73         nbuckets += bucketShift(b - 4)
```

看一下这个

28. 当 hint < 8 时， disass 没有看到调用 makemap\_small

## 问题

Plain Text  
package main

```
func main() {
var m = make(map[int]int, 7)
println(m)
}
```

# 一手资源+V 307570512

```
(dlv) disass
EXT main.main(SB) /home/golearn/testmap.go
testmap.go:3    0x460190      64488b0c25f8fffff
testmap.go:3    0x460199      488d4424a0
testmap.go:3    0x46019e      483b4110
testmap.go:3    0x4601a2      0f86d3000000
testmap.go:3    0x4601a8      4881ece0000000
testmap.go:3    0x4601af      4889ac24d8000000
testmap.go:3    0x4601b7      488dac24d8000000
=> testmap.go:4  0x4601bf*   0f57c9
testmap.go:4    0x4601c2      0f118c24a8000000
testmap.go:4    0x4601ca      0f57c9
testmap.go:4    0x4601cd      0f118c24b8000000
testmap.go:4    0x4601d5      0f57c9
testmap.go:4    0x4601d8      0f118c24c8000000
testmap.go:4    0x4601e0      488d7c2408
testmap.go:4    0x4601e5      0f57c0
testmap.go:4    0x4601e8      488d7fd0
testmap.go:4    0x4601ec      48896c24f0
testmap.go:4    0x4601f1      488d6c24f0
testmap.go:4    0x4601f6      e8e7c0ffff
testmap.go:4    0x4601fb      488b6d00
testmap.go:4    0x4601ff      488d8424a8000000
testmap.go:4    0x460207      8400
testmap.go:4    0x460209      488d442408
testmap.go:4    0x46020e      48898424b8000000
testmap.go:4    0x460216      488d8424a8000000
testmap.go:4    0x46021e      48898424a0000000
testmap.go:4    0x460226      e8e583feff
testmap.go:4    0x46022b      488b8424a0000000
testmap.go:4    0x460233      8400
testmap.go:4    0x460235      8b0c24
testmap.go:4    0x460238      89480c
testmap.go:4    0x46023b      488d8424a8000000
testmap.go:4    0x460243      4889842498000000
                                         mov rcx, qword ptr fs:[0xffffffff]
                                         lea rax, ptr [rsp-0x60]
                                         cmp rax, qword ptr [rcx+0x10]
                                         jbe 0x46027b
                                         sub rsp, 0xe0
                                         mov qword ptr [rsp+0xd8], rbp
                                         lea rbp, ptr [rsp+0xd8]
                                         xorps xmml, xmml
                                         movups xmmword ptr [rsp+0xa8], xmml
                                         xorps xmml, xmml
                                         movups xmmword ptr [rsp+0xb8], xmml
                                         xorps xmml, xmml
                                         movups xmmword ptr [rsp+0xc8], xmml
                                         lea rdi, ptr [rsp+0x8]
                                         xorps xmml, xmml
                                         lea rdi, ptr [rdi-0x30]
                                         mov qword ptr [rsp-0x10], rbp
                                         lea rbp, ptr [rsp-0x10]
                                         call 0x45c2e2
                                         mov rbp, qword ptr [rbp]
                                         lea rax, ptr [rsp+0xa8]
                                         test byte ptr [rax], al
                                         lea rax, ptr [rsp+0x8]
                                         mov qword ptr [rsp+0xb8], rax
                                         lea rax, ptr [rsp+0xa8]
                                         mov qword ptr [rsp+0xa0], rax
                                         call $runtime.fastrand
                                         mov rax, qword ptr [rsp+0xa0]
                                         test byte ptr [rax], al
                                         mov ecx, dword ptr [rsp]
                                         mov dword ptr [rax+0xc], ecx
                                         lea rax, ptr [rsp+0xa8]
                                         mov qword ptr [rsp+0x98], rax
```

## 回答

编译器会判断这个 map 是在栈还是堆上，你这个代码是在 main 函数栈上分配的

你把他从 main 函数移出去就能看到了

## 29. Timer 数据如何防丢失

### 问题

Timer heap 和 GMP 中的 P 绑定

1. 那程序重启或异常中断时，是否会丢失四叉堆上的数据
2. 业界热更新有什么方案解决这个

### 回答

既然会丢，说明就不适合只放内存里。

给你举个实际的业务场景例子，有些搞外卖、打车业务的公司有一种叫“慢必赔”的业务，

# 一手资源+V 307570512

比如打车，如果司机 10 分钟内没接到你，那就需要给你发个优惠券，这种就是需要通过 10min 的定时任务来做的。

但是因为每个用户的定时开始都不一样，相当于就是一大堆 timer，且不能丢。

所以他们会在订单开始的时候，往数据库里插一条数据，记录什么时间点需要发券，然后再通过简单的定时扫描逻辑，来处理这些过期的定时器。

## 再次追问

所以业务上使用的情况基本都是 db 落地，或者 timer + db 兜底来保证业务，这个跟老师讲的消息队列解耦的隐式坑又有点类似

## 回答

这个都是团队内搞定的，一般不会跨部门（比如主流程负责状态处理，同部门内的运营系统组负责相关的计算），

我写那篇文章有很多上下游会跨部门，跨部门的不好搞  
这种肯定要落地的~

## 30. 关于 send 有缓冲区且 recvq 不为空情况的问题

### 问题

Plain Text  
func main() {

```
    ch := make(chan int, 3)
    go func() {
        <- ch
    }()
    go func() {
        time.Sleep(time.Second)
        ch <- 1
    }()
    fmt.Println("Done!")
    time.Sleep(time.Hour)
}
```

这段代码应该就是向一个有 recvq 且 buf 为空数组的 channel 发一个数据。debug 走的是这个地方。

# 一手资源+V 307570512

```
Go
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
    ...
    if sg := c.recvq.dequeue(); sg != nil {
        // Found a waiting receiver. We pass the value we want to send
        // directly to the receiver, bypassing the channel buffer (if any).
        send(c, sg, ep, func() { unlock(&c.lock) }, 3)
        return true
    }
    ...
}
```

继续往后 debug，跳过了所有的 if 条件到了下面这个部分。

```
Plain Text
func send(c *hchan, sg *sudog, ep unsafe.Pointer, unlockf func(), skip int) {
    ...
    gp := sg.g
    unlockf()
    gp.param = unsafe.Pointer(sg)
    if sg.releasetime != 0 {
        sg.releasetime = cputicks()
    }
    goready(gp, skip+1)
}
```

那么问题来了，最后 send 函数的 ep 应该就是我们往 channel 里传递的数据，但是这里完全没有涉及到把 ep 传递给这个 gp 的操作。请问，这里 send 是怎么直接把数据传递给等待 recv 的 gp 呢？

## 回答

这里是个优化吧，  
你把

```
Plain Text
go func() {
    <- ch
}
```

改成

```
Plain Text
go func() {
    x := <- ch
    println(x)
}
```

再调试一下看看

# 一手资源+V 307570512

[31.](#) map 遍历，为什么 startBucket 和 offset 要加个随机数？

## 问题

曹大你好，map 遍历，为什么 startBucket 和 offset 要加个随机数？理论上从 0 来时便利不也是一样么？

## 回答

Go 的设计者认为 hash 表遍历本身是不保证顺序的（比如扩容的时候，遍历，顺序就会变），所以不希望用户依赖这个顺序

在遍历的时候特意做了随机化，就是为了避免你依赖默认的输出顺序

## 再次追问

这个还是不是很懂，正常从 hash 桶的计算来理解，从 0 开始或者从随机数开始，都是无法保障顺序输出的

## 回答

你理解的没问题，hash 桶就是没有办法保证逻辑，但是在非扩容期间，你不插新元素，他遍历顺序就是固定的，有的程序员看到这样的输出就认为遍历一直是固定顺序，可能就会对这个顺序产生逻辑依赖

[32.](#) context 最复杂的使用场景

## 问题

曹大见过比较复杂的 context 的使用场景是在哪里？有想过的开源项目或者代码段演示下么？

## 回答

context 一般不怎么复杂。。

你就找 etcd 之类的项目，直接去搜搜他们 ctx 的代码就行了~

[33.](#) 关于 chansend 函数

# 一手资源+V 307570512

## 问题

```
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool
```

chansend 这个函数中的 block 参数什么情况下为 false 和 true。源码中只看到了在 select 中会是 false。

## 回答

只有在

```
Plain Text
select {
    case <- ch:
    default:
}
```

才会是 false

## 34. map 简单的问题

### 问题

如果申请一个大小为 20 的 map，会分配几个 bucket？源码定义每个 bucket 大小为 8；是 buckets 数组会有 4 个 bucket？（2 的幂次 20->32）所以 B 是 2 吗？不太理解 谢谢老师  
相关截图：

```
const (
    // Maximum number of key/elel pairs a bucket can hold.
    bucketCntBits = 3
    bucketCnt     = 1 << bucketCntBits

    // A header for a Go map.
    type hmap struct {
        // Note: the format of the hmap is also encoded in cmd/compile/internal/gc/reflect.go.
        // Make sure this is sync with the compiler's definition.
        count      int // live cells == size of map. Must be first (used by len() builtin)
        flags      uint8
        B          uint8 // log_2 of # of buckets (can hold up to loadFactor * 2^B items)
        noverflow  uint16 // approximate number of overflow buckets; see incrnoflow for details
        hash0      uint32 // hash seed

        buckets     unsafe.Pointer // array of 2^B Buckets, may be nil if count==0.
        oldbuckets  unsafe.Pointer // previous bucket array of half the size, non-nil only when growing
        nevacuate   uintptr       // progress counter for evacuation (buckets less than this have been evacuated)

        extra *mapextra // optional fields
    }
)
```

# 一手资源+V 307570512

## 回答

你理解的没问题，所以接下来你需要的只是调试：

```
Plain Text
~/test git:master >>> cat map.go
*
package main

func main() {
    var m = make(map[int]int, 20)
    println(m)
}
```

```
Plain Text
~/test git:master >>> cat map.go
*
package main

func main() {
    var m = make(map[int]int, 20)
    println(m)
}
~/test git:master >>> dlv debug map.go
*
Type 'help' for list of commands.
(dlv) b main.main
Breakpoint 1 set at 0x105edb for main.main() ./map.go:3
(dlv) c
> main.main() ./map.go:3 (hits goroutine(1):1 total:1) (PC: 0x105edb)
 1:     package main
 2:
=> 3:     func main() {
 4:         var m = make(map[int]int, 20)
 5:         println(m)
 6:     }
(dlv) disass
TEXT main.main(SB) /Users/xargin/test/map.go
    map.go:3      0x105edb0    65488b0c2530000000    mov rcx, qword ptr
gs:[0x30]
    map.go:3      0x105edb9    483b6110      cmp rsp, qword ptr [rcx+0x10]
    map.go:3      0x105edb0    767a          jbe 0x105ee39
=>   map.go:3      0x105edb*    4883ec60      sub rsp, 0x60
    map.go:3      0x105edc3    48896c2458    mov qword ptr [rsp+0x58], rbp
    map.go:3      0x105edc8    488d6c2458    lea rbp, ptr [rsp+0x58]
    map.go:4      0x105edcd    0f57c0        xorps xmm0, xmm0
```

# 一手资源+V 307570512

|  |           |                    |                                 |
|--|-----------|--------------------|---------------------------------|
| map.go:4   | 0x105edd0 | 0f11442428         | movups xmmword ptr              |
| [rsp+0x28], xmm0   |           |                    |                                 |
| map.go:4   | 0x105edd5 | 0f57c0             | xorps xmm0, xmm0                |
| map.go:4   | 0x105edd8 | 0f11442438         | movups xmmword ptr              |
| [rsp+0x38], xmm0   |           |                    |                                 |
| map.go:4   | 0x105eddd | 0f57c0             | xorps xmm0, xmm0                |
| map.go:4   | 0x105ede0 | 0f11442448         | movups xmmword ptr              |
| [rsp+0x48], xmm0   |           |                    |                                 |
| map.go:4   | 0x105ede5 | 488d0534c60000     | lea rax, ptr [rip+0xc634]       |
| map.go:4   | 0x105edec | 48890424           | mov qword ptr [rsp], rax        |
| map.go:4   | 0x105edf0 | 48c744240814000000 | mov qword ptr [rsp+0x8],        |
| 0x14   |           |                    |                                 |
| map.go:4   | 0x105edf9 | 488d442428         | lea rax, ptr [rsp+0x28]         |
| map.go:4   | 0x105edfe | 4889442410         | mov qword ptr [rsp+0x10], rax   |
| map.go:4   | 0x105ee03 | e8a8d4faff         | call \$runtime.makemap          |
| map.go:4   | 0x105ee08 | 488b442418         | mov rax, qword ptr [rsp+0x18]   |
| map.go:4   | 0x105ee0d | 4889442420         | mov qword ptr [rsp+0x20], rax   |
| map.go:5   | 0x105ee12 | e83900fdff         | call \$runtime.printlock        |
| map.go:5   | 0x105ee17 | 488b442420         | mov rax, qword ptr [rsp+0x20]   |
| map.go:5   | 0x105ee1c | 48890424           | mov qword ptr [rsp], rax        |
| map.go:5   | 0x105ee20 | e82b09fdff         | call \$runtime.printpointer     |
| map.go:5   | 0x105ee25 | e8b602fdff         | call \$runtime.println          |
| map.go:5   | 0x105ee2a | e8a100fdff         | call \$runtime.printunlock      |
| map.go:6   | 0x105ee2f | 488b6c2458         | mov rbp, qword ptr [rsp+0x58]   |
| map.go:6   | 0x105ee34 | 4883c460           | add rsp, 0x60                   |
| map.go:6   | 0x105ee38 | c3                 | ret                             |
| map.go:3   | 0x105ee39 | e8029dffff         | call \$runtime.morestack_noctxt |
| <autogenerated>:1  | 0x105ee3e | e96dffff           | jmp \$main.main                 |
| (dlv) b makemap  |           |                    |                                 |
| Breakpoint 2 set at 0x100c2c3 for runtime.makemap() /usr/local/go/src/runtime/map.go:303               |           |                    |                                 |
| (dlv) c  |           |                    |                                 |
| > runtime.makemap() /usr/local/go/src/runtime/map.go:303 (hits goroutine(1):1 total:1) (PC: 0x100c2c3) |           |                    |                                 |
| Warning: debugging optimized function  |           |                    |                                 |
| 298: // makemap implements Go map creation for make(map[k]v, hint).                                    |           |                    |                                 |
| 299: // If the compiler has determined that the map or the first bucket                                |           |                    |                                 |
| 300: // can be created on the stack, h and/or bucket may be non-nil.                                   |           |                    |                                 |
| 301: // If h != nil, the map can be created directly in h.   |           |                    |                                 |
| 302: // If h.buckets != nil, bucket pointed to can be used as the first bucket.                        |           |                    |                                 |
| => 303: func makemap(t *maptpe, hint int, h *hmap) *hmap {   |           |                    |                                 |
| 304:     mem, overflow := math.MulUintptr(uintptr(hint), t.bucket.size)                                |           |                    |                                 |
| 305:     if overflow    mem > maxAlloc {   |           |                    |                                 |
| 306:         hint = 0  |           |                    |                                 |
| 307:     }   |           |                    |                                 |
| 308:   |           |                    |                                 |
| (dlv) n  |           |                    |                                 |
| > runtime.makemap() /usr/local/go/src/runtime/map.go:304 (PC: 0x100c2d1)                               |           |                    |                                 |
| Warning: debugging optimized function  |           |                    |                                 |
| 299: // If the compiler has determined that the map or the first bucket                                |           |                    |                                 |
| 300: // can be created on the stack, h and/or bucket may be non-nil.                                   |           |                    |                                 |
| 301: // If h != nil, the map can be created directly in h.   |           |                    |                                 |

# 一手资源+V 307570512

```
302: // If h.buckets != nil, bucket pointed to can be used as the first bucket.
303: func makemap(t *maptype, hint int, h *hmap) *hmap {
=> 304:     mem, overflow := math.MulUintptr(uintptr(hint), t.bucket.size)
305:     if overflow || mem > maxAlloc {
306:         hint = 0
307:     }
308:
309:     // initialize Hmap
(dlv) n
> runtime.makemap() /usr/local/go/src/runtime/map.go:305 (PC: 0x100c2e5)
Warning: debugging optimized function
300: // can be created on the stack, h and/or bucket may be non-nil.
301: // If h != nil, the map can be created directly in h.
302: // If h.buckets != nil, bucket pointed to can be used as the first bucket.
303: func makemap(t *maptype, hint int, h *hmap) *hmap {
304:     mem, overflow := math.MulUintptr(uintptr(hint), t.bucket.size)
=> 305:     if overflow || mem > maxAlloc {
306:         hint = 0
307:     }
308:
309:     // initialize Hmap
310:     if h == nil {
(dlv) n
> runtime.makemap() /usr/local/go/src/runtime/map.go:310 (PC: 0x100c2f2)
Warning: debugging optimized function
305:     if overflow || mem > maxAlloc {
306:         hint = 0
307:     }
308:
309:     // initialize Hmap
=> 310:     if h == nil {
311:         h = new(hmap)
312:     }
313:     h.hash0 = fastrand()
314:
315:     // Find the size parameter B which will hold the requested # of elements.
(dlv) n
> runtime.makemap() /usr/local/go/src/runtime/map.go:313 (PC: 0x100c300)
Warning: debugging optimized function
308:
309:     // initialize Hmap
310:     if h == nil {
311:         h = new(hmap)
312:     }
=> 313:     h.hash0 = fastrand()
314:
315:     // Find the size parameter B which will hold the requested # of elements.
316:     // For hint < 0 overLoadFactor returns false since hint < bucketCnt.
317:     B := uint8(0)
318:     for overLoadFactor(hint, B) {
(dlv) n
```

# 一手资源+V 307570512

```
> runtime.makemap() /usr/local/go/src/runtime/map.go:318 (PC: 0x100c317)
Warning: debugging optimized function
 313:     h.hash0 = fastrand()
 314:
 315:     // Find the size parameter B which will hold the requested # of elements.
 316:     // For hint < 0 overLoadFactor returns false since hint < bucketCnt.
 317:     B := uint8(0)
=> 318:     for overLoadFactor(hint, B) {
 319:         B++
 320:     }
 321:     h.B = B
 322:
 323:     // allocate initial hash table
(dlv) n
> runtime.makemap() /usr/local/go/src/runtime/map.go:319 (PC: 0x100c319)
Warning: debugging optimized function
 314:
 315:     // Find the size parameter B which will hold the requested # of elements.
 316:     // For hint < 0 overLoadFactor returns false since hint < bucketCnt.
 317:     B := uint8(0)
 318:     for overLoadFactor(hint, B) {
=> 319:         B++
 320:     }
 321:     h.B = B
 322:
 323:     // allocate initial hash table
 324:     // if B == 0, the buckets field is allocated lazily later (in mapassign)
(dlv) n
> runtime.makemap() /usr/local/go/src/runtime/map.go:318 (PC: 0x100c321)
Warning: debugging optimized function
 313:     h.hash0 = fastrand()
 314:
 315:     // Find the size parameter B which will hold the requested # of elements.
 316:     // For hint < 0 overLoadFactor returns false since hint < bucketCnt.
 317:     B := uint8(0)
=> 318:     for overLoadFactor(hint, B) {
 319:         B++
 320:     }
 321:     h.B = B
 322:
 323:     // allocate initial hash table
(dlv) n
> runtime.makemap() /usr/local/go/src/runtime/map.go:319 (PC: 0x100c319)
Warning: debugging optimized function
 314:
 315:     // Find the size parameter B which will hold the requested # of elements.
 316:     // For hint < 0 overLoadFactor returns false since hint < bucketCnt.
 317:     B := uint8(0)
 318:     for overLoadFactor(hint, B) {
=> 319:         B++
 320:     }
```

# 一手资源+V 307570512

```
321:     h.B = B
322:
323:     // allocate initial hash table
324:     // if B == 0, the buckets field is allocated lazily later (in mapassign)
(dlv) n
> runtime.makemap() /usr/local/go/src/runtime/map.go:318 (PC: 0x100c321)
Warning: debugging optimized function
313:     h.hash0 = fastrand()
314:
315:     // Find the size parameter B which will hold the requested # of elements.
316:     // For hint < 0 overLoadFactor returns false since hint < bucketCnt.
317:     B := uint8(0)
=> 318:     for overLoadFactor(hint, B) {
319:         B++
320:     }
321:     h.B = B
322:
323:     // allocate initial hash table

(dlv) p h
*runtime.hmap {
    count: 0,
    flags: 0,
    B: 2,
    nooverflow: 0,
    hash0: 1797362353,
    buckets: unsafe.Pointer(0xc00008e000),
    oldbuckets: unsafe.Pointer(0x0),
    nevacuate: 0,
    extra: *runtime.mapextra nil,
}
(dlv)
```

count 是在 map 里已经存了元素的时候才会有值的，hint 只是说大概可能会有多少个元素，提前分配足够的 bucket，减少一部分扩容成本(实际应用中很多时候 hint 也没啥用)

## 35. 关于 bucket 再多搬运一遍的问题

### 问题

#### 问题描述:

1. mapasssign: 将命中的 bucket 从 oldbuckets 顺便搬运到 buckets 中，顺便再多搬运一个 bucket
2. mapdelete: 将命中的 bucket 从 oldbuckets 顺便搬运到 buckets 中，顺便再多搬运一个 bucket

这里的顺便再多搬运一个，这个多搬运的一个 bucket，是如何选取的？

# 一手资源+V 307570512

还是上面想说的是，将命中的 **entry** 从 **oldbuckets** 顺收搬到新的 **bucket** 中，然后顺带，将该 **entry** 当前的这个 **oldbucket** 中所有 **entry** 都按照规则搬一遍？

## 回答

搬运过程中有一个 **nevacuate** 索引，记录搬运的位置，搬运完自己命中的那个以后，从 **nevacuate** 位置再搬一个

当然，这是一个简化的说法，因为命中的 **bucket** 是随机的，所以也可能 **nevacuate** 位置的 **bucket** 已经被搬运过了，这时候只要简单改一下 **nevacuate++** 就行了

## 36. 关于 Map 增长时的遍历问题

### 问题

问题描述：

如果在迁移过程中遍历，只返回那些将迁移到当前 **bucket** 的 **oldbucket** 的条目。

这句话是否理解成，在遍历的同时，**map** 还在持续增长并且伴随 **bucket** 的迁移。

比如一个原来 **bucket-1** 中的 8 个元素，增长后，变成 **bucket-1** 中有 3 个（原来 **bucket-1**）+ 1 个（来自 **bucket-M**），另外 5 个（来自 **bucket-1**）分配到 **bucket-N** 中了。

当遍历到 **bucket-1** 时，只返回最原始的 3 个（**oldbucket**），是这个意思吗？

如果是的话，那来自 **bucket-M** 的那 1 个元素，会如何被遍历到呢？

## 回答

两种情况：

1. 等大扩容(扩容前 **hmap.B** = 扩容后 **hmap.B**)的时候，**buckets[i]** 的元素一定是来自于原来的 **oldbuckets[i]** 的，这种情况下无脑把 **oldbuckets[i]** 和它后面挂的 **overflow** 无脑遍历完就行了。
2. 增大扩容(扩容前 **hmap.B+1** = 扩容后 **hmap.B**，即 **bucket** 总数 \*2)的时候，**oldbuckets[i]**，在 **buckets** 中一部分去了 **buckets[i]**(代码里的 X part)，一部分去了 **buckets[2^B+i]**(代码里的 Y part)，我们遍历 **buckets[i]** 的时候，就只要按那个扩容规则，就是 **lowbits** 来算一下，就知道 **oldbuckets[i]** 哪些元素会进 **buckets[i]** 了，只把这部分拿出来就行，剩下的那部分，会在遍历 **buckets[N+i]** 的时候被遍历到

这东西看起来拿文字描述确实不好懂，明天讲完课，我再补一下动画吧。

# 一手资源+V 307570512

## 37. 关于 go 中 type xxx struct 结构体中 tag 的问题

### 问题

Plain Text

```
type AlipayConfig struct {
    AppID      string `mapstructure:"app_id" json:"app_id"`
    PrivateKey string `mapstructure:"private_key" json:"private_key"`
    AliPublicKey string `mapstructure:"ali_public_key" json:"ali_public_key"`
    NotifyURL  string `mapstructure:"notify_url" json:"notify_url"`
    ReturnURL  string `mapstructure:"return_url" json:"return_url"`
}
```

像这样一个结构，里面可以有很多的 tag 配置，有些不是很了解，故想咨询一下老师。

1.mapstructure 是做什么用的？

2.json 这个 tag 的作用是不是将结构映射成 json 字符串时，json 字符串中的 key 值为结构体 tag 中 json 配置的值？

3.除了 mapstructure || json || binding 外还有没有其它比较常用的 tag？

### 回答

mapstructure 是社区的一个库吧，我记得是把 map 转成 struct 的，在转换的时候需要知道字段映射关系，比如

NotifyURL 在转换前在 map 中是 {"notify\_url" : "<https://baidu.com>"}

json 这种是 json encode、decode 的时候用的，需要知道 json 字符串里的东西和你解析后的结构字段怎么对应(要不反射的代码没法写)。

你这里举例的这些 tag，理论上都是可以随意替换的，纯粹看社区库作者的喜好

PS: 你这个是不是把你们公司代码贴出来了。。。注意安全

## 38. Timer 为什么用四叉堆

### 问题

请问曹大，为什么 Timer 要选择四叉堆，而不是二叉堆或者五叉堆，选择四叉堆是有什么好处么

### 回答

<https://www.geeksforgeeks.org/k-ary-heap/>

# 一手资源+V 307570512

你看看上面这个链接，里面提到了多叉树的优点，和我上课说的也差不多：

- K-ary heap when used in the implementation of [priority queue](#) allows faster decrease key operation as compared to binary heap ( $O(\log 2n)$ ) for binary heap vs  $O(\log kn)$  for K-ary heap). Nevertheless, it causes the complexity of extractMin() operation to increase to  $O(k \log kn)$  as compared to the complexity of  $O(\log 2n)$  when using binary heaps for priority queue. This allows K-ary heap to be more efficient in algorithms where decrease priority operations are more common than extractMin() operation. Example: [Dijkstra's algorithm](#) for single source shortest path and [Prim's algorithm](#) for minimum spanning tree
- K-ary heap has better memory cache behaviour than a binary heap which allows them to run more quickly in practice, although it has a larger worst case running time of both extractMin() and delete() operation (both being  $O(k \log kn)$ ).

1 的意思是虽然大家都是  $\log kn$ ，但是多叉树的 k 要小，所以实际做堆调整的实际上时间复杂度低一些

2 像 Go 里这个四叉，其实每个节点都保存了 4 个 timer，空间局部性比两叉好一些，对缓存友好(虽然实际上里面存的是指针，还是有二级寻址)

至于为什么是 4 叉，而不是 5 叉，6 叉。。我觉得可能没什么特殊考量，这种设计的时候能是 2 的整数倍，能让树层数变少就行了

## 39. Timer 为什么用四叉堆

### 问题

请问曹大，为什么 Timer 要选择四叉堆，而不是二叉堆或者五叉堆，选择四叉堆是什么好处么

### 回答

和这个重复了：<https://class.imooc.com/course/qadetail/290681>

## 40. atomic 和 mutex 锁问题

### 问题

曹大，昨天直播中提到了

1. atomic 和 mutex 分别在什么情况下用会比较好，channel 用的是 mutex 吗？

- 尽量 mutex
- 一些比较简单的场景，比如：
  - 配置加载双 buffer，可以用 atomic。
  - 并发安全计数器，可以用 atomic

# 一手资源+V 307570512

c. atomic 相对来说是比较底层的 API，应用层还是尽量少用~除非在锁有明显的性能瓶颈时，需要考虑优化。

私下我测试了下

```
Plain Text
// sync.Mutex
func test1() {
var wg sync.WaitGroup
    var mutex sync.Mutex
    count := int64(0)
t := time.Now()
for i := 0; i < 10000; i++ {
wg.Add(1)
go func(i int) {
mutex.Lock()
count++
wg.Done()
mutex.Unlock()
}(i)
}

wg.Wait()

fmt.Printf("test1 花费时间: %d, count 的值为: %d \n", time.Now().Sub(t), count)
}

// sync.atomic
func test2() {
var wg sync.WaitGroup
    count := int64(0)
t := time.Now()
for i := 0; i < 10000; i++ {
wg.Add(1)
go func(i int) {
atomic.AddInt64(&count, 1)
wg.Done()
}(i)
}

wg.Wait()

fmt.Printf("test2 花费时间: %d, count 的值为: %d \n", time.Now().Sub(t), count)
}
```

测试结果：

test1 花费时间: 4030930, count 的值为: 10000  
test2 花费时间: 3097699, count 的值为: 10000

测试结果说明 atomic 的效率是优于 mutex 的，为什么建议平常使用 mutex，是因为 atomic 是对内存块加锁，影响系统的整体性能么？

## 回答

1. atomic 只能保护一个值，而 mutex 可以保护临界区内所有的值
2. mutex 是用 atomic 指令结合其它的数据结构来实现的
3. 你这里的测试场景就属于简单的计数器场景，这种用 atomic 没啥问题，因为简单呀~

## 41. atomic 和 mutex 锁使用场景问题

### 问题

曹大，昨天在直播中提到了 atomic 和 mutex 的使用场景问题

1. atomic 和 mutex 分别在什么情况下用会比较好， channel 用的是 mutex 吗？
  - a. 尽量 mutex
  - b. 一些比较简单的场景，比如：
    - i. 配置加载双 buffer，可以用 atomic。
    - ii. 并发安全计数器，可以用 atomic
  - c. atomic 相对来说是比较底层的 API，应用层还是尽量少用~除非在锁有明显的性能瓶颈时，需要考虑优化。

下来我测试了下

```
Plain Text
// sync.Mutex
func test1() {
    var wg sync.WaitGroup
    var mutex sync.Mutex
    count := int64(0)
    t := time.Now()
    for i := 0; i < 10000; i++ {
        wg.Add(1)
        go func(i int) {
            mutex.Lock()
            count++
            wg.Done()
            mutex.Unlock()
        }(i)
    }
    wg.Wait()
}
fmt.Printf("test1 花费时间: %d, count 的值为: %d \n", time.Now().Sub(t), count)
```

# 一手资源+V 307570512

```
// sync.atomic
func test2() {
    var wg sync.WaitGroup
    count := int64(0)
    t := time.Now()
    for i := 0; i < 10000; i++ {
        wg.Add(1)
        go func(i int) {
            atomic.AddInt64(&count, 1)
            wg.Done()
        }(i)
    }
    wg.Wait()

    fmt.Printf("test2 花费时间: %d, count 的值为: %d \n", time.Now().Sub(t), count)
}
```

执行结果:

test1 花费时间: 4573742, count 的值为: 10000

test2 花费时间: 3195538, count 的值为: 10000

结果显示 atomic 的效率会高点。

为什么你建议业务层要尽量使用 mutex 呢？是因为 atomic 会对内存块直接加锁，影响系统整体的性能么？

## 回答

atomic 是较底层的 api

如果用 atomic 编程需要对并发有比较深的理解，要深入地理解我们课上提了几句的内存重排，memory barrier 等相关知识

用 atomic 编写高性能数据结构还要知道使用这种 api 独有的一些并发问题，比如 ABA 问题

在深入理解这些知识的前提下，才不容易写出 bug，

同步编程这节里提到的一些数据结构就是做无锁处理的，比如 sync.Pool，你可以看看，还是比较复杂的

对于上层应用来说，atomic 太底层了~

个别计数、配置替换的场景用一用问题不大，因为逻辑比较简单

你这里的代码是很简单的计数场景，用 atomic 也没啥问题

## 42. chan struct{} 相关的特殊优化问题

## 问题

# 一手资源+V 307570512

上课原问题：

make(chan struct{})记得有文档说有特殊优化，譬如分配内存少，不实现直接发送给 recv 方等，是否还有其它优化？

查代码找到到的只有分配的内存较少（mem=0）

相关代码：

<https://github.com/golang/go/blob/bc9c580409b61af6b29f0cbd9d45bec63dbe2ccb/src/runtime/chan.go#L93-L97>

```
Plain Text
case mem == 0:
// Queue or element size is zero.
c = (*hchan)(mallocgc(hchanSize, nil, true))
// Race detector uses this location for synchronization.
c.buf = c.raceaddr()switch {
```

和不执行 handoff

相关代码：

<https://github.com/golang/go/blob/bc9c580409b61af6b29f0cbd9d45bec63dbe2ccb/src/runtime/chan.go#L295-L298>

```
Plain Text
if sg.elem != nil {
sendDirect(c.elemtpe, sg, ep)
sg.elem = nil
}
```

但是所谓的不执行 handoff 依赖的是在 recv 的时候忽略收到的值 (<-ch)，而不是对 chan struct{} 的特殊优化？

## 回答

你这个文档是在哪里看到的呀

我实际调试了一下，chan 元素是 struct{} 的时候，sendDirect 还是会进去的，只是在 memmove 的时候，t.size == 0

## 再次追问

看来本人是被源码分析博客“毒害”的反面教材；因为之前有读到过 chan struct{} 是一种特殊的 chan，所以一直有这种印象；

这次直接动手做实验

```
Plain Text
schan1.go
```
package main
```

# 一手资源+V 307570512

```
func main() {  
  
    c1 := make(chan struct{})  
    c2 := make(chan bool)
```

```
    go func() {  
        <-c1  
        c2 <- false  
    }()
```

```
    c1 <- struct{}{}  
    <-c2  
}
```

```
```  
schan2.go  
```
```

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    c1 := make(chan struct{})  
    c2 := make(chan bool)
```

```
    go func() {  
        v := <-c1  
        fmt.Printf("%v\n", v)  
        c2 <- false  
    }()
```

```
    c1 <- struct{}{}  
    <-c2  
}
```

```
```
```

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    c1 := make(chan struct{})  
    c2 := make(chan bool)
```

```
    go func() {  
        v := <-c1
```

# 一手资源+V 307570512

```
fmt.Printf("%v\n", v)
c2 <- false
}()

c1 <- struct{}{}
r := <-c2
if r {
    fmt.Println(r)
}
}

```

```

分别编译得到 schan1~schan3, dlv 调试对 runtime.sendDirect 打断点后得到结果  
只有 schan3 会进入 sendDirect

但是按照逻辑 schan2 应该也会进入 sendDirect, 所以这里猜测, 在编译阶段, 如果 chan 的类型为 struct{}, 那么会吧 recv 端的 elem 变为 nil

继续实验, 对 schan2 dlv, b runtime.send

```
=> 295:     if sg.elem != nil {
296:         sendDirect(c.elemtpe, sg, ep)
297:         sg.elem = nil
298:     }
299:     gp := sg.g
300:     unlockf()
(dlvg) p sg.elem
unsafe.Pointer(0x0)
(dlvg)
```

## 回答

嘿嘿, 肯定自己调试是最准的

## [43. time.Now\(\)的性能问题](#)

### 问题

time.Now()的性能问题, 部分语言是需要获取操作系统本地时间, 存在性能问题, go 语言如何

### 回答

time.Now 底层有 vdso 的优化, 一般不需要陷入到内核,  
但在一些特殊情况下可能会导致进入内核态, 这时候会碰到性能问题,

具体可以参考我朋友的这篇文章：  
<https://mp.weixin.qq.com/s/06SDQLzDprJf2AEaDnX-QQ>。

## 6.2 号答疑

<https://class.imooc.com/lesson/62#mid=46372>

### 1. 指针实现接口与结构体实现指针的区别？

#### 问题

##### 问题描述：

为什么指针实现接口时，只有指针类型的变量才会实现该接口；而使用结构体实现接口时，指针类型和结构体类型都会实现该接口？

##### 相关代码：

```
Plain Text
type Animal interface {
Eat()
}

type Dog struct{ }

// 结构体指针实现
func (d *Dog) Eat() {
fmt.Println("dog eat")
}
// 结构体实现
func (d Dog) Eat() {
fmt.Println("dog eat")
}
```

##### 调用代码：

```
Plain Text
// 指针变量调用
d := &Dog{}
d.Eat()

// 结构体变量调用
d := Dog{}
d.Eat()
```

能理解 d 调用时，因为函数调用时都是值传递，所以会复制一个指针，而结构体变量时会赋值一个变量 d，但是这样为什么会“指针类型和结构体类型都会实现该接口”呢？

# 一手资源+V 307570512

## 回答

Go 里面有很多特性其实是编译器的功劳。例如对 map 取值既可以返回 ok，也可以不返回 ok，这是编译器分析后调用的不同的底层 map 的取值函数。

这里其实也是编译器的功劳，真要想探索到底是哪些代码做的事，可以用 ide 调试一番，参考一下其他班学员刚写的一篇探索文章：

<https://blog.csdn.net/zxxshaycormac/article/details/117606285>

个人觉得没太大必要啥都要怼到源码，有些就是简单的语法糖而已。

再贴一篇相关的文章也可以看看：<https://www.qcrao.com/2019/04/25/dive-into-go-interface/#2-%E5%80%BC%E6%8E%A5%E6%94%B6%E8%80%85%E5%92%8C%E6%8C%87%E9%92%88%E6%8E%A5%E6%94%B6%E8%80%85%E7%9A%84%E5%8C%BA%E5%88%AB>

## 再次追问

也就是说，只是语法层面做了一些语法糖，使用结构体实现接口时，代码自动生成了指针类型和结构体类型的接口实现，对吗？

## 回答

是的，你可以看看最后的汇编结果，其实是帮你生成了对应的 method 的

## Go 的内存管理与垃圾回收

<https://class.imooc.com/lesson/62#mid=46373>

### 1. GC 代码怎么看？

## 问题

最近从 gcStart 开始 trace 代码，结合课程里的 svg 那张图，但是图上只有调用流程，没有文字性的描述(或者哪里有文字性的描述)，想要看具体细节就得看源码。想说写一个小代码，然后创建一个不可达的变量，手动出发 GC，再通过 dlv 去 trace，这样的方式是可以的吗？还是曹大之前是怎么 trace gc 代码的？

## 回答

gc 这部分要看看理论，再看代码，要不然看不懂。理论部分对应的那几本书的章节在 ppt 里标出来了

我之前看也是把关键的数据结构(多级分配结构，wbBuf 之类的)都画出来，然后跟着流程

理解分配和回收过程的。

课上也给了一些图，阅读代码流程的时候能把这些结构的关系搞清楚应该就差不多了。感兴趣的部分细节可以 dlv 帮助理解，这个没问题。

## 2. 关于内存分配的多级缓存

### 问题

1.1.11 版本以前的线性内存和 1.11 以后的稀疏内存，是不是只是页堆的存储方式不同，多级缓存机制不受这个变化影响？

2.当我们有一个微对象 tiny 需要申请内存的时候，这个流程对不对：①mcache 中的 tiny 微分配器如果有合适的空间就直接分配到这里②如果没有就按照相应的 spanclass 从 mcache 的 alloc 中获取相应的 mspan③如果没有获取到，就同样按照 spanclass 找到相应的 mcentral，在这个 mcentral 的空闲集合中找到一个 mspan④如果还没有 mspan 可以用，就需要走扩容，去 mheap 里面拿过来。

3.按照上面流程我们每次拿到的 mspan 结构体管理的 pages 是真的被拿到了各级缓存中，还是说所有的 pages 一直都是在 mheap 的 arenas 管理的 arena 中， mspan 里面真的包含了实际的虚拟内存页吗？

4.我们各级缓存中的 mspan 被 GC 后，释放出来之后是还在对应的缓存中，还是统一回到 mheap 中？

### 回答

1.1.11 版本以前的线性内存和 1.11 以后的稀疏内存，是不是只是页堆的存储方式不同，多级缓存机制不受这个变化影响？

A: 对，就是全局的内存管理有差别，多级缓存变化不大。

2.当我们有一个微对象 tiny 需要申请内存的时候，这个流程对不对：①mcache 中的 tiny 微分配器如果有合适的空间就直接分配到这里②如果没有就按照相应的 spanclass 从 mcache 的 alloc 中获取相应的 mspan③如果没有获取到，就同样按照 spanclass 找到相应的 mcentral，在这个 mcentral 的空闲集合中找到一个 mspan④如果还没有 mspan 可以用，就需要走扩容，去 mheap 里面拿过来。

A: 你这个描述问题不大。

差不多是这么个流程：<https://www.figma.com/proto/tSl3CoSWKitJtvIhqLd8Ek/memory-management-%26%26-garbage-collection?page-id=165%3A81&node-id=165%3A197&viewport=-20163%2C1534%2C2.7550384998321533&scaling=scale-down>

3.按照上面流程我们每次拿到的 mspan 结构体管理的 pages 是真的被拿到了各级缓存中，还是说所有的 pages 一直都是在 mheap 的 arenas 管理的 arena 中， mspan 里面真的包含了实际的虚拟内存页吗？

A: mspan, mcache, mcentral 这些其实是对内存分配过程的抽象，只是记录位置用的~页

# 一手资源+V 307570512

管理肯定还是全局的 mheap 负责的

4. 我们各级缓存中的 mspan 被 GC 后，释放出来之后是还在对应的缓存中，还是统一回到 mheap 中？

sweep 过程分两种：

1) 后台的 bgsweep ---> 这个过程有可能会把内存放回到 mheap 的 mcentral

2) 分配内存时候的顺手 sweep ---> 这个过程中只是扫一些 slot 出来给分配过程使用，不会放回 mcentral

简单的说明可以看看这个 <https://medium.com/a-journey-with-go/go-memory-management-and-memory-sweep-cc71b484de05>,

代码的话需要看一下， bgsweep 和 alloc 过程中的 sweep 代码。

## 3. 关于 gc 中的 sweep 过程

### 问题

曹大我看这个环节是并发的，那么它是怎么保证我待清扫的对象不会被其他的协程使用呢。还是说所有待清扫的对象不可能会被引用。

### 回答

待清扫的语法上不可达的，一定不会被引用。

同时 mark 阶段过后， mcache 和 P 也是断开连接的，在分配过程中也会负责一部分清扫工作，但是清扫的也一定是不可达对象。

## 4. gc 标记 使用 yuasa 时 函数中初始化的堆上的数据是灰色么

### 问题

如果运行在栈上允许添加 barrier，并采用 yuasa 的写 barrier 设计 那么，在函数内的 b := make([]int,1000000) 因为这个 b 太大会逃逸到堆上，那么这个 b 初始化灰色的么？

### 回答

yuasa 是在指针发生删除行为的时候才标灰，你这里的应该是栈上指针直接指向堆(没有指针的删除操作)。所以应该是不涉及 yuasa 的 barrier(因为你说允许栈上加 barrier，那这个也应该算是 dijistra insertion barrier)

现在的 gc 设计， gc 开始之后，所有新创建的堆对象都是黑色的。

## 5. 申请大对象时 内存管理的 heapArena mspan 是如何分发的

## 问题

比如申请一个>32KB 的对象 当前的 arena 不足 会通过 mmap 申请 64MB 空间 会创建 heapArena 这时候 heapArena 这个结构体数据 是存储在申请的 64MB 上么？而且 分发这个对象的时候 创建的 mspan 会绑定到当前的 heapArena.spans 么 这个 mspan 存储的位置也会放到申请的 64MB 中么 这个 heapArena 会放到到 mheap\_.arenas 么

## 回答

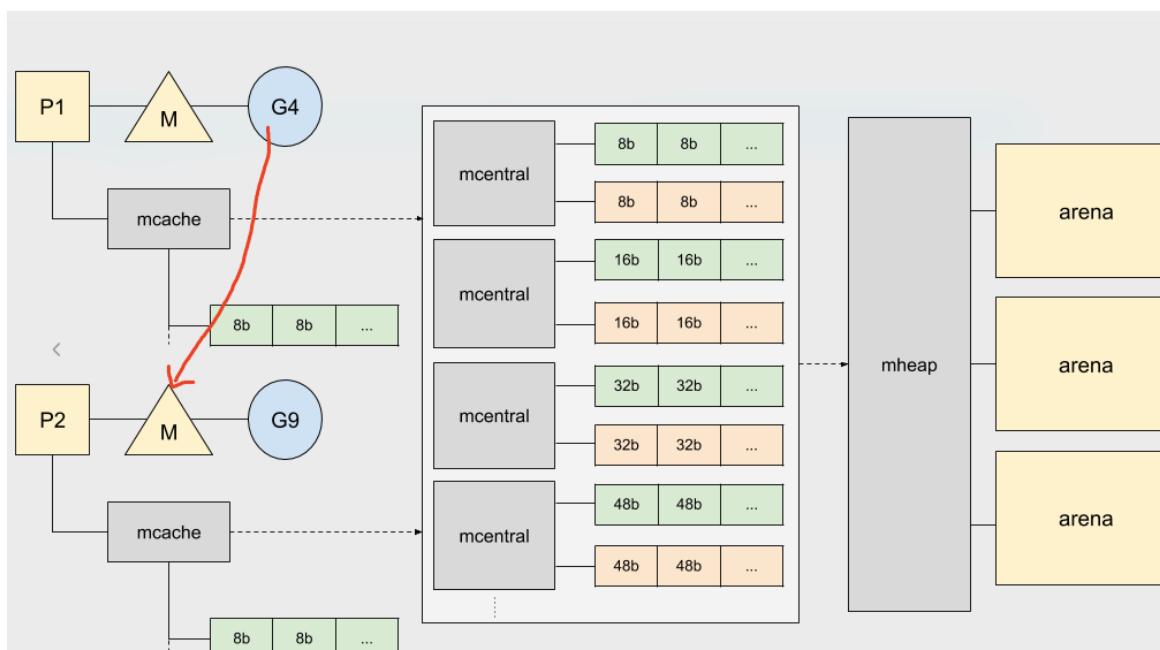
heapArena 和 mspan 结构体上面都有个 go:notinheap 的注释

这种比较特殊，走的是 persistentAlloc 路径，是另外一条内存分配路径

所以应该也不在刚分配出的 64MB 里面

## 6. tiny 对象分配后，对象所在 G 被抢占了会发生什么？

## 问题



假设 G4 分配 tiny 对象后，被抢占到了另一个 M，那么到了另一个 M，可以调用之前分配的地址吗？还是重新分配？

## 回答

mallocgc 期间会置：

# 一手资源+V 307570512

Plain Text

```
mp.mallocing = 1
```

在抢占信号处理的时候，会判断这个值。

Plain Text

```
func doSigPreempt(gp *g, ctxt *sigctxt) {
    // Check if this G wants to be preempted and is safe to
    // preempt.
    if wantAsyncPreempt(gp) && isAsyncSafePoint(gp, ctxt.sigpc(), ctxt.sigsp(), ctxt.siglr()) {
        // Inject a call to asyncPreempt.
        ctxt.pushCall(funcPC(asyncPreempt))
    }

    // Acknowledge the preemption.
    atomic.Xadd(&gp.m.preemptGen, 1)
    atomic.Store(&gp.m.signalPending, 0)
}

func isAsyncSafePoint(gp *g, pc, sp, lr uintptr) bool {
    mp := gp.m

    // Check M state.
    if mp.p == 0 || !canPreemptM(mp) {
        return false
    }
    ....
}

// 注意这里面的 mp.mallocing
func canPreemptM(mp *m) bool {
    return mp.locks == 0 && mp.mallocing == 0 && mp.preemptoff == "" && mp.p.ptr().status
== _Prunning
}
```

所以 malloc 完成之前，不可能被抢占。

malloc 完成之后，地址已经被赋予给对象了，这时候再去其它的 M 没啥关系。

[7.](#) 想知道 dlv 能不能根据函数的指针打印出函数对应的所在文件位置呀？

## 问题

dlv 能不能根据函数的指针 打印出函数对应的所在文件位置呀？

## 回答

# 一手资源+V 307570512

```
JavaScript
Type 'help' for list of commands.
(dlv) b *0x1054910
Breakpoint 1 set at 0x1054910 for _rt0_amd64_darwin()
```

8. 协程调度 那个 m 上的 g0 是什么时候初始化的 这个 g0 会和 m 一直绑定不会被其他 p 给窃取吗

## 问题

协程调度 那个 m 上的 g0 是什么时候初始化的 这个 g0 会和 m 一直绑定不会被其他 p 给窃取吗

当执行阻塞系统调度时 m 不是会跟 p 解绑吗 那么这个 m 上的 g0 跟其他 g 一样吗 还是一直绑定到当前这个 m 上

## 回答

每个 m 都有一个 g0，负责执行一些调度相关的指令，g0 是和 m 绑定死的

g0 用的栈空间是创建线程的时候系统给线程分配的栈空间。

9. 内存分配 协程的 p 切换的问题

## 问题

我看先在 p.mcache 进行堆内存分配 那如果这个协程因为系统调用和 p 解绑的话 或者其他 p 窃取了这个协程 那么这个协程的分配在原本 p.mcache 的内存内容如何访问

## 回答

mspan 只是个分配的记录对象(记一些 freeindex 之类的)，  
对象分配完毕以后你直接用对象就行了，不需要和 mspan 发生啥交互了。

所以分配完毕后，g 跑到别的 P 去执行也没关系

对象之后死掉的话，这个对象的内存回收是 GC 的事情

10. 栈上的黑色对象指向堆上的白色对象问题

## 问题

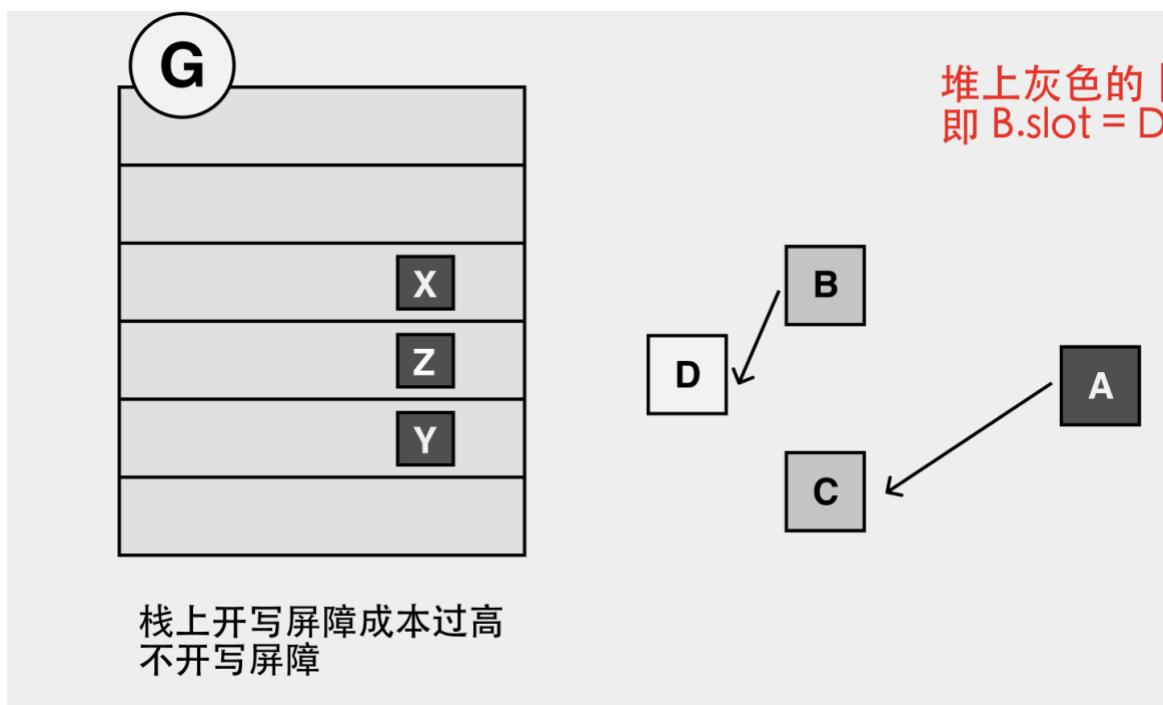
曹大：根据强三色不变性，栈上的黑色对象指向堆上的白色对象是不合法的，但是如果这个白色对象堆上没有被其他对象引用，栈上又不会有写屏障，那么这个白色对象岂不是永远不会被标记为灰色或者黑色么？

## 回答

课上的这个动画还有一种更复杂的情况，你可以看看：

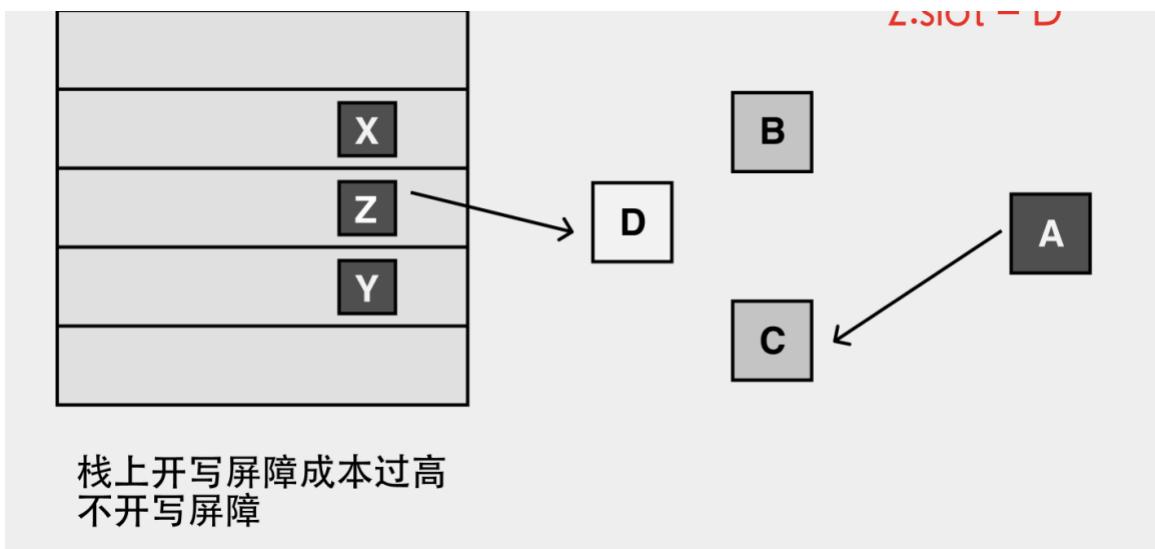
<https://www.figma.com/proto/tSl3CoSWKitJtvIhqLd8Ek/memory-management-and-and-garbage-collection?page-id=201%3A293&node-id=201%3A294&viewport=3552%2C382%2C0.5088089108467102&scaling=contain>

栈上的黑指向了堆上的白，说明这个堆上的白一定是从堆上某条链路断掉的对象：

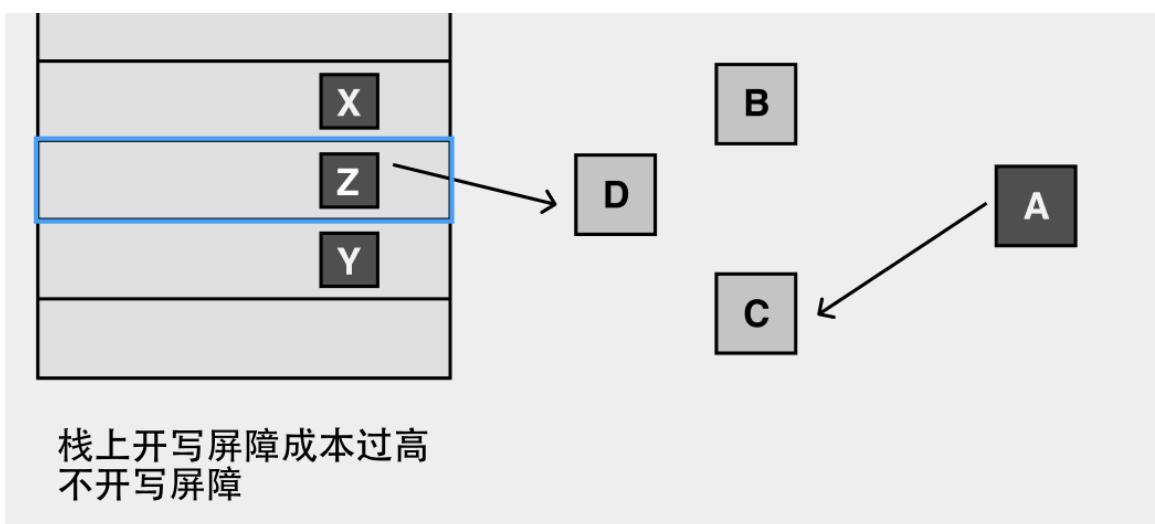


比如这个，我们后续让 z 指向 D

# 一手资源+V 307570512



但是要考虑到 Go 的屏障是混合屏障(包含了删除写屏障的逻辑),  
所以堆上的链路在标记过程中断开的话(被 mutator 修改), 被断开的那个对象一定会被标  
灰,  
所以第二张图这样的情况不会出现, B -> D 断开的时候, D 会被标灰(就是 D 对象的  
gcMarkbit = 1 && 进 wbBuf)  
正常的应该是这样:



## [11. 关于多线程堆栈分配](#)

### 问题

曹大每个线程拥有自己独立的虚拟地址空间吗(堆空间和栈空间都是吗还是只有栈空间或者其他)。如果每个线程都有自己独立的栈或堆空间的话他们的大小是固定吗还是动态收缩的。

## 回答

这个应该是操作系统方面的基础，线程的地址空间是共用的。  
每个线程有自己的栈(见图)。

堆是公用的(见下面的图)。在 C 语言里，栈是可以增长的，但是有个限制，可以通过 ulimit 来看到。

~ >>> ulimit -a	
-t: cpu time (seconds)	unlimited
-f: file size (blocks)	unlimited
-d: data seg size (kbytes)	unlimited
<u>-s: stack size (kbytes)</u>	8192
-c: core file size (blocks)	0
-v: address space (kbytes)	unlimited
-l: locked-in-memory size (kbytes)	unlimited
-u: processes	2784
-n: file descriptors	256

Go 的 goroutine 的栈是在堆区分配的，能增长，但是有限制(最大 1GB)，可以通过 SetMaxStack 来改，不过一般不改这个。

上面这些内容和这个图是可以对应的上的：

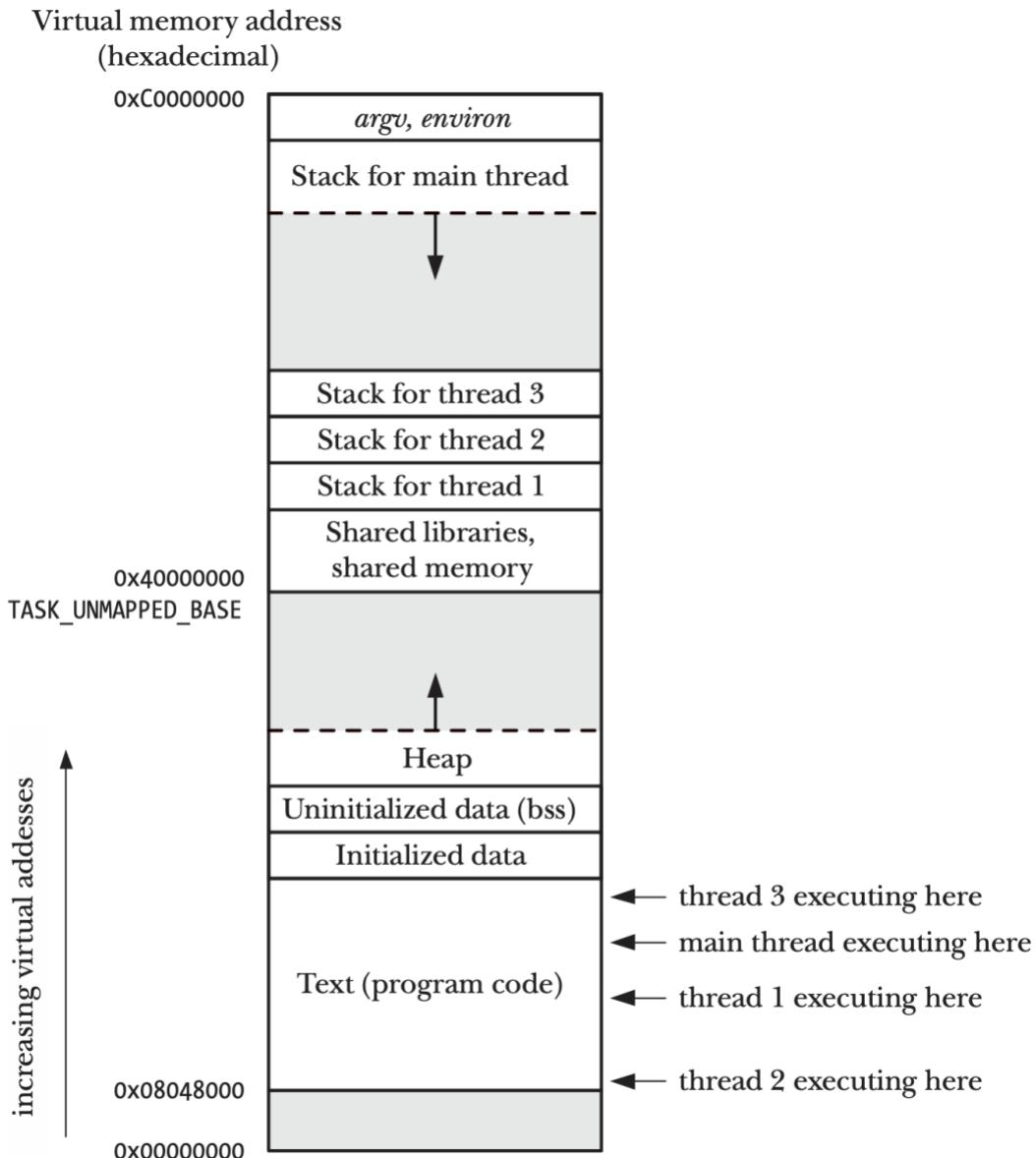


Figure 29-1: Four threads executing in a process (Linux/x86-32)

## [12. p 本地内存 mcache 分配为什么不需要加锁](#)

### 问题

曹大，如果 P 的本地内存 mcache 的某个 spanclass 用完了，回去 mcenter 中拿一个 span 过来，这个 span 里面的 slot 不一定全部都为空，那就是说可能有其他 P 使用了这个 span 里某些 slot，这样设计到 gc 或者分配的时候，为什么不需要加锁。

## 回答

课上没有太讲 sweep 的内容，  
可以看看这篇文章：

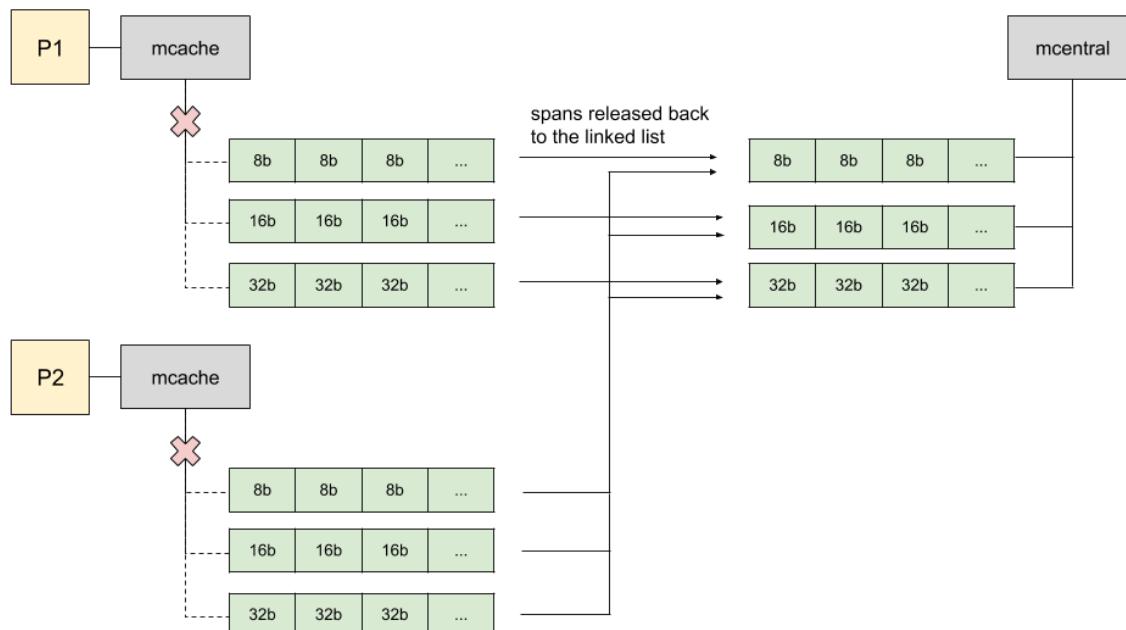
<https://medium.com/a-journey-with-go/go-memory-management-and-memory-sweep-cc71b484de05>

同一个 mspan，同一时刻应该只能归属到一个 P 里，它里面有一些 slot 确实可能之前被其它的 P 用了，但是其它 P 在用掉这块内存之后，之后就对这个 mspan 没有任何操作了。

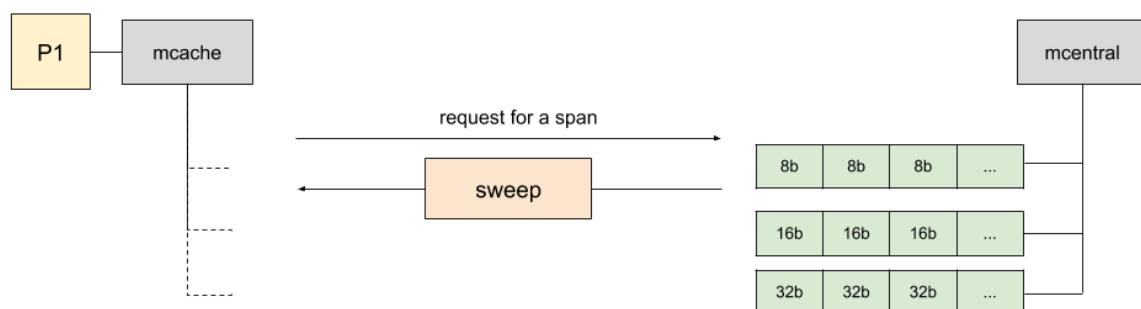
(如果内存不使用了，那么之后的操作也是在 gc mark 和 sweep 里)

gc mark 的时候，操作的是 gcmarkbits，这个是并发安全的(atomic.Or)。

sweep 的时候，为了让 sweep 能和 mutator 并发执行，会将本地的 mcache 先和 P 解除关系，把 mcache 里的 mspan 放回到 mcentral 里去



下次用户申请内存走 refill 逻辑的时候，会顺便执行一些 sweep 工作(也就是说，sweep 工作要么是后台的 bgsweep 协程在做，要么是 refill 的时候，由对应 P 上的唯一的协程来做)。refill 最终把 mspan 分配给唯一的一个 P。



# 一手资源+V 307570512

这样，拿到的 mcache 里的 mspan 的空槽就肯定没有其它 P 在操作了。

分配的时候能够保证一定没有其它的 P 或者后台的 sweep 协程操作，所以不加锁也是安全的。

## [13. Gc 在什么时候占用 cpu 非常高](#)

### 问题

是在标记阶段还是清扫阶段

### 回答

主要是标记

可以看看你们线上项目的火焰图的~

## [14. 对象分配到栈的相关源码](#)

### 问题

怎么看对象分配到栈的源码，是反编译后的那些汇编代码吗？

src/runtime/stack.go 这个文件是做什么，和上面问题相关吗？下面注释没看明白。

Visual Basic

Stack layout parameters.

Included both by runtime (compiled via 6c) and linkers (compiled via gcc).

The per-goroutine g->stackguard is set to point StackGuard bytes above the bottom of the stack. Each function compares its stack pointer against g->stackguard to check for overflow. To cut one instruction from the check sequence for functions with tiny frames, the stack is allowed to protrude StackSmall bytes below the stack guard. Functions with large frames don't bother with the check and always call morestack. The sequences are (for amd64, others are similar):

```
guard = g->stackguard
frame = function's stack frame size
argsize = size of function arguments (call + return)
```

```
stack frame size <= StackSmall:
CMPQ guard, SP
JHI 3(PC)
MOVQ m->morearg, $(argsize << 32)
```

# 一手资源+V 307570512

```
CALL morestack(SB)
```

```
stack frame size > StackSmall but < StackBig  
LEAQ (frame-StackSmall)(SP), R0  
CMPQ guard, R0  
JHI 3(PC)  
MOVQ m->morearg, $(argsize << 32)  
CALL morestack(SB)
```

```
stack frame size >= StackBig:  
MOVQ m->morearg, $((argsize << 32) | frame)  
CALL morestack(SB)
```

The bottom StackGuard - StackSmall bytes are important: there has to be enough room to execute functions that refuse to check for stack overflow, either because they need to be adjacent to the actual caller's frame (deferproc) or because they handle the imminent stack overflow (morestack).

For example, deferproc might call malloc, which does one of the above checks (without allocating a full frame), which might trigger a call to morestack. This sequence needs to fit in the bottom section of the stack. On amd64, morestack's frame is 40 bytes, and deferproc's frame is 56 bytes. That fits well within the StackGuard - StackSmall bytes at the bottom.  
The linkers explore all possible call traces involving non-splitting functions to make sure that this limit cannot be violated.

## 回答

要看和栈分配相关的，能看懂 stack 的操作汇编就行，主要就是 sub rsp，然后在 rsp 范围以内变量搬来搬去

runtime 里这个文件主要是栈空间的管理，不是栈上变量的管理，Go 的栈空间分配走 newstack，morestack 这些逻辑，也是一套比较复杂的流程。

## 再次追问

正好学习了下 <https://xargin.com/plan9-assembly/> ?

然后正好看到了

<https://draveness.me/golang/docs/part3-runtime/ch07-memory/golang-stack-management/#%E6%A0%88%E5%88%9D%E5%A7%8B%E5%8C%96>

里的栈空间管理，看了下源码， stackinit() 里 stackpool 和 stackLarge 确实都和 mspan 有关，文章里说到

“这两个用于分配空间的全局变量都与内存管理单元 runtime.mspan 有关，我们可以认为 Go 语言的栈内存都是分配在堆上的，运行时初始化会调用 runtime.stackinit 初始化这些全局变量”

我们可以认为 Go 语言的栈内存都是分配在堆上的  
上面这句话是说 栈空间的那块分配地址 其实是放在 堆上面管理的？有点不太理解，似懂非懂。。。

## 回答

在操作系统的语境下，栈内存主要指的是线程的栈，  
Go 语言里，goroutine 的栈是另外管理的，

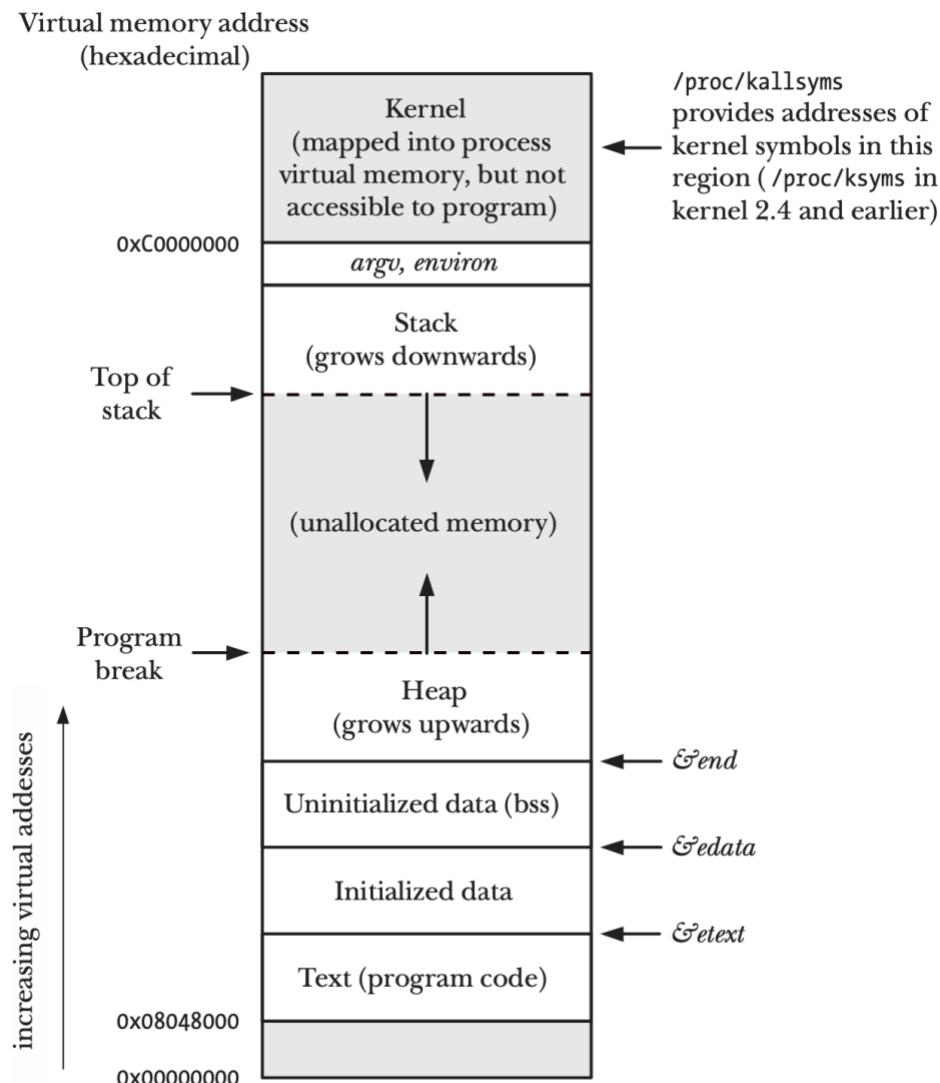


Figure 6-1: Typical memory layout of a process on Linux/x86-32

本质上相当于被分配到了这张图的 Heap 区域，当然，g0 的栈用的还是线程栈~  
上面这句话是说 栈空间的那块分配地址 其实是放在 堆上面管理的，你这个理解没错，普通 goroutine 的栈空间，就是在 Heap 区管理的。

## 15. 课程内容以外问题: 如何学习新的编程语言

### 问题

曹大您好，最近公司在调研使用 rust 来写一些新的服务  
我最近看了些 rust 相关内容，之前有 python 和 go 的经验，但是发现 rust 学习起来很是费劲  
想请教下曹大，学习新的语言比如 rust，一般有什么比较好的经验吗

### 回答

rust 还是挺难的哈哈。  
学新语言得多抄抄各种形式的代码(也可以花一小段时间，用 rust 写写 leetcode 上一些题，来熟悉语法)，  
学习过程搞个自己的代码库，后续可以去搜

像 rust 这种每次更新都有新特性加进来的语言。。如果跟得上，最好是每个版本的都实验一下  
有项目机会更好~

我一直没碰上 rust 的公司项目，所以目前也只是大概会用语法~

## 16. 源码中 class\_to\_\* 的作用

### 问题

在读 `mcentral.cacheSpan` 的时候，发现第一行：  
`spanBytes := uintptr(class_to_allocnpages[c.spanclass.sizeclass()]) * _PageSize`  
其实这一行很好理解，就是分配按照 class 分配多少字节。

我点到 `class_to_allocnpages` 实现，首先看到了 `class_to_size`，这个很好理解就是每个 class 占用多少字节，然后下面一行就是 `class_to_allocnpages` 了，这个最开始也很好理解就是每个 class 分配多少页呗，前面都很正常，但是我看到最后发现了异常（截取最后十个）：

```
var class_to_allocnpages = [_NumSizeClasses]uint8{0, 1, 1, 1, ... 5, 8, 3, 10, 7, 4}
```

从后向前看：

- 省略号前面都能理解，
- 最后一个 4 对应的 `class66(32768byte)` :  $32768 / 8192 = 4$  这个很正常
- 但是倒数第二个分配了 7 个 page 这就很尴尬了，先求出来吧： $28672 / 8192 = 3.5$ ，是 7 的 1/2 这就很尴尬了，不理解
- 在算一个（倒数第三个）:  $27264 / 8192 = 3.328125$ ，难道是我理解错了？？？

然后就顺便把 `sizeclasses.go` 中所有字段都看了，还发了 `divMagic` 和 `class_to_divmagic`，这两个又是啥 magic 都出来了，使用方是在 `mhead.allocSpan`(从 `mcentral.grow` 点进去)，估计

# 一手资源+V 307570512

也看不懂，希望曹大也能指点一样，这样就把 [sizeclasses.go](#) 中字段都能理解了？

## 回答

要看一下这个表格：

C++	// class	bytes/obj	bytes/span	objects	tail	waste	max	waste
// 1	8	8192	1024	0	87.50%			
// 2	16	8192	512	0	43.75%			
// 3	32	8192	256	0	46.88%			
// 4	48	8192	170	32	31.52%			
// 5	64	8192	128	0	23.44%			
// 6	80	8192	102	32	19.07%			
// 7	96	8192	85	32	15.95%			
// 8	112	8192	73	16	13.56%			
// 9	128	8192	64	0	11.72%			
// 10	144	8192	56	128	11.82%			
// 11	160	8192	51	32	9.73%			
// 12	176	8192	46	96	9.59%			
// 13	192	8192	42	128	9.25%			
// 14	208	8192	39	80	8.12%			
// 15	224	8192	36	128	8.15%			
// 16	240	8192	34	32	6.62%			
// 17	256	8192	32	0	5.86%			
// 18	288	8192	28	128	12.16%			
// 19	320	8192	25	192	11.80%			
// 20	352	8192	23	96	9.88%			
// 21	384	8192	21	128	9.51%			
// 22	416	8192	19	288	10.71%			
// 23	448	8192	18	128	8.37%			
// 24	480	8192	17	32	6.82%			
// 25	512	8192	16	0	6.05%			
// 26	576	8192	14	128	12.33%			
// 27	640	8192	12	512	15.48%			
// 28	704	8192	11	448	13.93%			
// 29	768	8192	10	512	13.94%			
// 30	896	8192	9	128	15.52%			
// 31	1024	8192	8	0	12.40%			
// 32	1152	8192	7	128	12.41%			
// 33	1280	8192	6	512	15.55%			
// 34	1408	16384	11	896	14.00%			
// 35	1536	8192	5	512	14.00%			
// 36	1792	16384	9	256	15.57%			
// 37	2048	8192	4	0	12.45%			
// 38	2304	16384	7	256	12.46%			
// 39	2688	8192	3	128	15.59%			
// 40	3072	24576	8	0	12.47%			
// 41	3200	16384	5	384	6.22%			
// 42	3456	24576	7	384	8.83%			

# 一手资源+V 307570512

// 43	4096	8192	2	0	15.60%
// 44	4864	24576	5	256	16.65%
// 45	5376	16384	3	256	10.92%
// 46	6144	24576	4	0	12.48%
// 47	6528	32768	5	128	6.23%
// 48	6784	40960	6	256	4.36%
// 49	6912	49152	7	768	3.37%
// 50	8192	8192	1	0	15.61%
// 51	9472	57344	6	512	14.28%
// 52	9728	49152	5	512	3.64%
// 53	10240	40960	4	0	4.99%
// 54	10880	32768	3	128	6.24%
// 55	12288	24576	2	0	11.45%
// 56	13568	40960	3	256	9.99%
// 57	14336	57344	4	0	5.35%
// 58	16384	16384	1	0	12.49%
// 59	18432	73728	4	0	11.11%
// 60	19072	57344	3	128	3.57%
// 61	20480	40960	2	0	6.87%
// 62	21760	65536	3	256	6.25%
// 63	24576	24576	1	0	11.45%
// 64	27264	81920	3	128	10.00%
// 65	28672	57344	2	0	4.91%
// 66	32768	32768	1	0	12.50%

bytes/span，这个应该是 8192 的整数倍，obj 是不是整数倍无所谓，不过你看 tail waste，最终有些 span 内部还是会有一点浪费掉的空间的

你算的时候拿的不是 bytes/span 这一列吧。  
divShift 这些看着是计算对象在 span 之类的数据结构内的偏移的~  
我也没太看计算细节~

## 17. 源码中 size\_to\_\* 的作用

### 问题

在 `sizeclasses.go` 中有这么几个常量：

```
Plain Text
const (
    _MaxSmallSize = 32768 // 8K
    smallSizeDiv  = 8     // ? ? ? smallsize 分配的对齐么
    smallSizeMax  = 1024
    largeSizeDiv  = 128   // ? ? ? largesize 分配的对齐么
    _NumSizeClasses = 67
    _PageShift     = 13    // 1 << 13 = 8192(8K) 为啥要有个 shift
)
```

# 一手资源+V 307570512

然后有个疑问?:

`smallSizeDiv` 和 `largeSizeDiv` 的作用是什么? 看了一下 `smallSizeDiv` 使用的地方是在 `mallocgc` 中:

```
Plain Text
if size <= smallSizeMax-8 {
    sizeclass = size_to_class8[(size+smallSizeDiv-1)/smallSizeDiv]
} else {
    sizeclass = size_to_class128[(size-smallSizeMax+largeSizeDiv-1)/largeSizeDiv]
}
```

看完就更加蒙圈了一些, `size_to_class8` 和 `size_to_class128` 又是什么? 8 和 128 代表什么? 看到了 `size_to_class8` 和 `size_to_class128` 在 `sizeclasses.go` 中的定义, 看到 `smallSizeDiv` 和 `largeSizeDiv` 作为被除数好像理解 Div 的含义, 但是还是不确定, 所以就引发了下面的几个问题:

1. `size_to_class8` 和 `size_to_class128` 中的 8 和 128 到底是个啥? 代表 class 8 代表 112byte 那个 class 么, 那 class128 是啥? size 的单位是什么?
2. `size_to_class8` 和 `size_to_class128` 的含义是什么?
3. `smallSizeDiv` 和 `largeSizeDiv` 的作用是什么, 可能知道的上面两个为回答, 这个问题自然就知道了

还有个小疑问就是为什么有 `_PageShift` 呢? 看了使用的代码用 `npages<<_PageShift` 表示 `npages` 占用多少字节, 是因为位运算 (不直接`*8192`) 快么

## 回答

`sizeClass` 就是那个表格里的 66 种

这里感觉是想了办法 (就是查表算法), 能用两个表格把 32 KB 以内的内存都映射到那 66 种里面去,

又尽量让查的表小一些, 感觉没啥神奇的。。。

移位肯定比乘除要快的

## 18. 多个 goroutine 情况下 gc 如何回收的呢?

## 问题

本节讲到的内存管理和 gc 回收好像不涉及到多个 goroutine 的情况, 比如 gc 在标记的时候 (甚至回收的时候), 如果是多个 goroutine 的情况, 变量的内存会不会导致被错误标记? go 如何处理的? 或者有那些资料可以了解下如何处理?

## 回答

mark 本来就是并发的, `markroot` 的时候, 通过一个偏移 `i` 来让不同的标记 goroutine 尽量从不同的根起始

标记过程中使用了并发安全的 `atomic.Or8`

之前有一个类似的问题~

<https://class.imooc.com/course/qadetail/292422>

## 19. 关于 tiny 的几个疑问

### 问题

当需要的内存小于 16 bytes && 没有包含指针的时候，内存分配的是 tiny，基于这个前提，有几个疑问：

- 1、图中 mcache 里 tiny 指向的列表下面的 64bytes 是一个 tiny elem，那么一个 tiny 对象最大 16byte，岂不是访问 alloc 很频繁？
- 2、mcache 中的 alloc 指向的列表，是不是和 central 一样甚至是同一个东西？
- 3、为什么说会 tiny 对象本地用完之后，都在第 5 个槽上找内存？如果说是没有指针，noscan 放在奇数槽，那么 1, 3, 5... 分别对应的 spanClass 应该是 8bytes、16bytes、32bytes？是不是我理解错了？

### 回答

课上的图画的有点问题，槽是第 5 个，不过这个槽里的元素大小是 16 bytes，不是 64 bytes，ppt 已修复，槽内元素大小要用 sizeClass 算，而不是 spanClass，sizeClass = spanClass(5) >> 1 = 2，对应代码表格里的 class 2，也就是 16 bytes

本质上都是指向一块内存，只不过不同区域看待内存的角度不一样，所以有不同的结构，mcache 里的 134 个槽，每个槽里只有一个 mspan，而 mcentral 里的那 134 个槽，每个槽是一个 mspan 的列表(当然他还分了 empty 和 non-empty)。列表里的元素是从 central 被分配到 mcache 里的，除了 large 的分配，其它都是一级一级向上申请，向下下发。同一个 spanClass 的 mspan 结构上是完全一样的，不管他是在 mcentral 还是在 mcache 里。

0 和 1 号槽是给 large 预留的，2-3 号槽是给 <=8 字节的做分配，4-5 号槽是给 <=16 字节的做分配。只不过 5 号还会被用来给 tiny 类型的做分配。

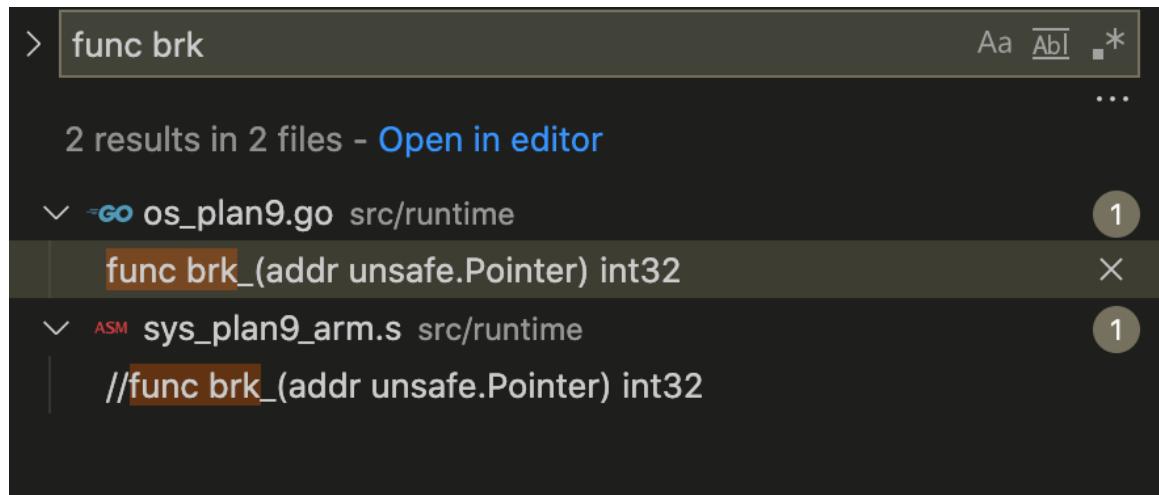
5 号槽是这么算出来的：

```
Go
type spanClass uint8
tinySpanClass = spanClass(tinySizeClass<<1 | 1)
tinySizeClass = _TinySizeClass
_TinySizeClass = int8(2)
2 << 1 | 1 = 5
```

## 20. 如何查找对应的虚拟内存操作的 golang 实现代码？

### 问题

在学习 stack 和 heap 分配的时候，在 heap 上分配，有一个 brk 操作。我看到操作系统的书介绍，其实有调用 brk(\* addr)方法，于是我对应查找 golang 的实现代码 brk 和 mmap。  
比如我找 brk



A screenshot of a code search interface. The search bar at the top contains the text "func brk". Below the search bar, it says "2 results in 2 files - Open in editor". The results list two items:

- os\_plan9.go src/runtime:  
func brk\_(addr unsafe.Pointer) int32
- sys\_plan9\_arm.s src/runtime:  
//func brk\_(addr unsafe.Pointer) int32

只有一个汇编的和一个 go 的方法定义，没有具体实现。这是表示此方法实际是调用操作系统的 brk 方法吗？这样的代码如何可以找到对应的真正调用源码？

### 回答

我们一般看 linux 下的就行，你看的这两个文件后缀是 plan9，现在基本没人用这系统的相关的系统调用直接查看 mem\_xxx.go 就行

比如 linux 就看 mem\_linux.go。

具体的实现，如 mmap，需要用“·mmap”搜索

## 21. 关于多线程 gc 问题

### 问题

多个 gc 扫描是出现扫描同一个块内存的问题是如何解决的

### 回答

# 一手资源+V 307570512

我看了一下，大概是下面三个方面有保障：

markroot 的时候，有一个 i 参数，来保证每个 gcMarkWorker 标记不同的根

mark 的时候，使用的是 atomic.Or8，这是一个并发安全的位操作函数

mark 之前有一个 mbits.isMarked 的判断，也就是说如果指针有交叉，那到这时候还有一次判断，能拦住一部分重复的标记工作

## 22. 单核上 p0 和 m0 m1 如何绑定处理

### 问题

单核上 默认只会有一个 p 但 m0 和 p0 已经绑定的话 而且 m0 上只执行 g0 那么启动的其他协程 如何运行 是会在加一个 m 然后 p0 和 m0 m1 不停的解绑 绑定吗

### 回答

是不是我之前哪个回答写的有点问题，m0 不只跑 g0。。。 初始化如果 main.main 的 goroutine 发生可接管阻塞的话，也会跑调度循环的。

## 23. m0 上只跑 g0 么 m0 会阻塞吗

### 问题

m0 上只跑 g0,那 g0 上运行的 main.main 里面如果执行了 time.sleep 那这个 m0 也只会阻塞在这么 而不是运行其他的 g 么

### 回答

m0 的 g0 和普通 m 的 g0，都是用来跑调度相关的代码的

m0 在跑用户的代码的时候，也会创建专门的 goroutine:

下面这个是启动过程中的 rt0\_go:

```
SQL
// create a new goroutine to start program
MOVQ $runtime·mainPC(SB), AX      // entry
PUSHQ AX
PUSHQ $0    // arg size
CALL runtime·newproc(SB)
POPQ AX
```

# 一手资源+V 307570512

```
POPQ AX
```

```
// start this M  
CALL runtime·mstart(SB)
```

这里新创建的 goroutine，传入的方法是 runtime.main

这里新启动的就是 m0，m0 通过 mstart 函数，最终进入 schedule，找到启动阶段创建的新 goroutine(不是 g0)，这个启动阶段创建的 goroutine 要执行的方法就是 runtime.main

我之前给的回答应该是有点问题，m0 在初始化完毕后，应该和其它线程没啥两样了，都是该阻塞阻塞，该执行调度就执行调度

## 24. 用 dlv 调试发现 runtime.g0 和 runtime.m0 在一开始就有值了 那这个是什么时候复值

### 问题

用 dlv 调试的时候发现 runtime.g0 和 runtime.m0 在一开始就有值了 那这个是什么时候复值的？刚开始的 g0 主要负责什么呀

### 回答

m0 默认是个全局变量，内部字段默认就是 0 值，在初始化过程中会逐渐填充这个 m0 的各种字段，比如 schedinit 里的 procesize 会给他绑定 P，initsig 会给他初始化 gsignal 等等，其实最终效果就是让 m0 和程序启动的主线程给对上，m0.g0 用程序的主线程栈

m0 的 g0 和其它 m 的 g0 一样，都是执行调度相关代码的时候会切过去，比如 runtime 里的 systemstack、mcall 之类的会有栈切换

最开始的 g0 应该也是执行 schedule 函数代码的时候会用到

## 25. 内存分配的问题？

### 问题

曹大，如果申请内存的时候，发现 mcache 中 tiny 中满了，要从 alloc 中的第 5 个槽位找，为什么是第 5 个槽位，那其他的槽位都是干什么用的？

### 回答

# 一手资源+V 307570512

<https://class.imooc.com/course/qadetail/292027>

看看上面这个

其它槽位是给其它大小的对象的内存分配准备的~

32KB 以内，每种大小的对象分配根据其有指针/无指针都会映射到某一种 spanClass，在 134 种槽中找到自己的位置。

## 26. 曹大,有点汇编问题搞不懂

### 问题

我在看你写的<<Go 语言高级编程>>.这本书的关于汇编章节的书.看到下面这段有点疑惑

# 一手资源+V 307570512

第一步改写依然是使用Go语言，只不过是用汇编的思维改写：

```
func main() {
    var a, b int

    a = 10
    runtime.Println(a)
    runtime.Println()

    b = a
    b += b
    b *= a
    runtime.Println(b)
    runtime.Println()
}
```

首选模仿C语言的处理方式在函数入口处声明全部的局部变量。然后根据MOV、ADD、MUL等指令的风格，将之前的变量表达式展开为用 = 、 += 和 \*= 几种运算表达的多个指令。最后用runtime包内部的Println和Printn函数代替之前的Println函数输出结果。

经过用汇编的思维改写过后，上述的Go函数虽然看着繁琐了一点，但是还是比较容易理解的。下面我们进一步尝试将改写后的函数继续转译为汇编函数：

```
TEXT .main(SB), $24-0
MOVQ $0, a-8*2(SP) // a = 0
MOVQ $0, b-8*1(SP) // b = 0

// 将新的值写入a对应内存
MOVQ $10, AX          // AX = 10
MOVQ AX, a-8*2(SP) // a = AX

// 以a为参数调用函数
MOVQ AX, 0(SP)
CALL runtime.Println(SB)
CALL runtime.Println(SB)

// 函数调用后，AX/BX 寄存器可能被污染，需要重新加载
MOVQ a-8*2(SP), AX // AX = a
MOVQ b-8*1(SP), BX // BX = b

// 计算b值，并写入内存
MOVQ AX, BX          // BX = AX // b = a
ADDQ BX, BX          // BX += BX // b += a
IMULQ AX, BX          // BX *= AX // b *= a
MOVQ BX, b-8*1(SP) // b = BX

// 以b为参数调用函数
MOVQ BX, 0(SP)
CALL runtime.Println(SB)
CALL runtime.Println(SB)

RET
```

汇编实现main函数的第一步是要计算函数栈帧的大小。因为函数内有a、b两个int类型变量，同时调用的runtime.Println函数参数是一个int类型并且没有返回值，因此main函数的栈帧是3个int类型组成的24个字节的栈内存空间。

在函数的开始处先将变量初始化为0值，其中 a-8\*2(SP) 对应a变量、 a-8\*1(SP) 对应b变量（因为a变量先定义，因此a变量的地址更小）。

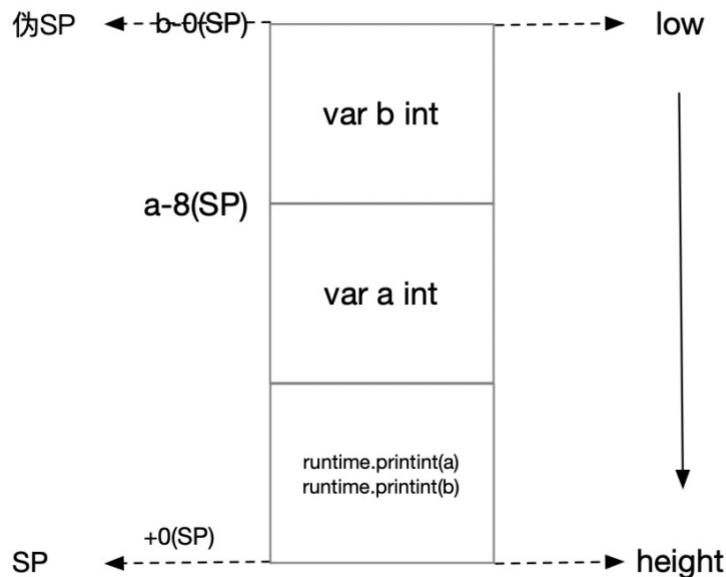
这里面的 a 的变量为啥不是 a-8(SP)呢? b 的话不应该是 b-0(SP)吗?还有 +0(SP)不是指向了栈顶了吗?

因为 a 变量先定义,因此 a 变量的地址更小.

栈内存地址不是从大往小吗?为啥 a 变量先定义地址更小呢?

我按照我的理解画了一张图.

栈桢24字节



## 回答

Go 的汇编有个比较难理解的地方。。。

我们课上之前讲的是通过编译、反编译工具看的汇编，这种情况下看到的 SP 表示的都是物理寄存器，就是那个栈顶

但是在手写汇编的时候，SP 有的时候是指伪寄存器，有的时候是指物理寄存器，这个有一个具体的区分规则，比较蛋疼

# 一手资源+V 307570512

当前的 callee 函数是 add，在 add 的代码中引用 FP，该 FP 指向的位置不在 callee 的 stack frame 之内，而是在 caller 的 stack frame 上。具体可参见之后的 栈结构 一章。

- PC: 实际上就是在体系结构的知识中常见的 pc 寄存器，在 x86 平台下对应 ip 寄存器，amd64 上则是 rip。除了个别跳转之外，手写 plan9 代码与 PC 寄存器打交道的情况较少。
- SB: 全局静态基指针，一般用来声明函数或全局变量，在之后的函数知识和示例部分会看到具体用法。
- SP: plan9 的这个 SP 寄存器指向当前栈帧的局部变量的开始位置，使用形如 symbol+offset(SP) 的方式，引用函数的局部变量。offset 的合法取值是 [-framesize, 0)，注意是个左闭右开的区间。假如局部变量都是 8 字节，那么第一个局部变量就可以用 localvar0-8(SP) 来表示。这也是一个词不表意的寄存器。与硬件寄存器 SP 是两个不同的东西，在栈帧 size 为 0 的情况下，伪寄存器 SP 和硬件寄存器 SP 指向同一位置。手写汇编代码时，如果是 symbol+offset(SP) 形式，则表示伪寄存器 SP。如果是 offset(SP) 则表示硬件寄存器 SP。务必注意。对于编译输出(go tool compile -S / go tool objdump)的代码来讲，目前所有的 SP 都是硬件寄存器 SP，无论是否带 symbol。

我们这里对容易混淆的几点简单进行说明：

1. 伪 SP 和硬件 SP 不是一回事，在手写代码时，伪 SP 和硬件 SP 的区分方法是看该 SP 前是否有 symbol。如果有 symbol，那么即为伪寄存器，如果没有，那么说明是硬件 SP 寄存器。
2. SP 和 FP 的相对位置是会变的，所以不应该尝试用伪 SP 寄存器去找那些用 FP + offset 来引用的值，例如函数的入参和返回值。
3. 官方文档中说的伪 SP 指向 stack 的 top，是有问题的。其指向的局部变量位置实际上是整个栈的栈底(除 caller BP 之外)，所以说 bottom 更合适一些。
4. 在 go tool objdump/go tool compile -S 输出的代码中，是没有伪 SP 和 FP 寄存器的，我们上面说的区分伪 SP 和硬件 SP 寄存器的方法，对于上述两个命令的输出结果是没法使用的。在编译和反汇编的结果中，只有真实的 SP 寄存器。
5. FP 和 Go 的官方源代码里的 framepointer 不是一回事，源代码里的 framepointer 指的是 caller BP 寄存器的值，在这里和 caller 的伪 SP 是值是相等的。

我之前写过怎么区分的文章

<https://github.com/cch123/golang-notes/blob/master/assembly.md>

写的时候差点把自己劝退。

## 27. 1 个 G 的 slice，实际内存占用相差甚远的原因

### 问题

```
Go
package main

import "time"

func main() {
    a := make([]int, 2<<30)
    _ = a
    time.Sleep(time.Hour)
}
```

上面应该是分配了 1 个 G 的 slice

	GoLand	2.73 GB	112	1,357	85984
com.docker.hyperkit		2.58 GB	15	39	21901
WindowServer		1.14 GB	15	5,128	143
main		530.6 MB	5	14	20020
印象笔记		465.0 MB	17	839	18996

# 一手资源+V 307570512

看到的是 500 多 M

请问下这种现象和操作系统（linux/window/macOs）是否有区别？以及少去的 400 多 M 大致的原因。

## 回答

操作系统是惰性分配内存的，没有访问到的地址可能要访问触发缺页中断才会真正分配

比如你在 go 里 make 一个 1gb 的 slice 但是从来不访问，就不会真的分配 1gb

和不同系统的内存分配策略也确实有关系。

## 28. 关于 Tiny 对象的几个问题

### 问题

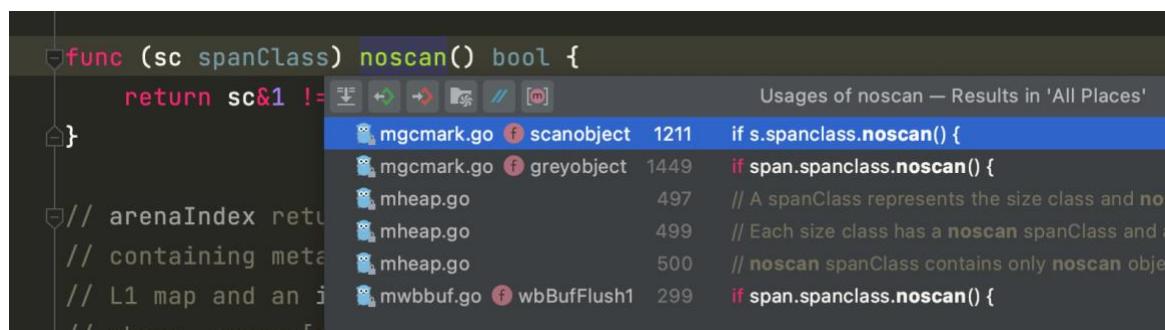
关于 Tiny 对象的几个问题：

1. 有指针的 scan，这里的指针，是指对象内部的成员包含指针，还是自己被其他人引用？
2. 分成奇数偶数 sapnslot，是为了在回收时方便么？
3. 如果 tiny 对象自身增长了，是不是会用 small/large 重新分配？

## 回答

1. 应该是内部含指针

2. 是的哦，比如一种类型是 noscan，那么扫描到堆上的它的时候，一定是叶子节点了：



The screenshot shows a code editor with a search results panel titled 'Usages of noscan — Results in 'All Places''.

File	Line	Usage
mgcmark.go	1211	if s.spanclass.noscan() {
mgcmark.go	1449	if span.spanclass.noscan() {
mheap.go	497	// A spanClass represents the size class and no
mheap.go	499	// Each size class has a noscan spanClass and a
mheap.go	500	// noscan spanClass contains only noscan obje
mwbbuf.go	299	if span.spanclass.noscan() {

3. 自身咋增长呢，如果是 slice 之类的，扩容到 cap 放不下都是重新分配新的地址了，要从堆上重新分配空间

【做完了】服务上线后——成为 Go 语言性能调优专家

# 一手资源+V 307570512

<https://class.imooc.com/lesson/62#mid=46384>

## 1. 会终止运行吗

### 问题

如果一个 api 发起请求 因为 go 处理时间较长 浏览器认为超时，主动关闭了这个 tcp 连接  
那个这个 api 对应的协程是不是不会终止 还会继续运行呀？ 直到这个协程流程执行完成

### 回答

c 端一般情况下没法主动请求 s 端停止操作的。

## 2. 需要什么参数运行这些 benchmark？

### 问题

在 Go 语言性能调优专家一课中，有一些动手实验的 benchmark 测试，需要什么参数运行  
这些 benchmark？都是 -benchmem 还是 -benchcpu？或者还需要其他？

### 回答

主要就是 **-bench=.** 和 **-benchmem**，你不知道用哪个就全都带上。

## 3. 中台本身会依赖外部服务么？

### 问题

中台自己也就是个微服务，别人都是去中台的服务去获取数据，那中台本身会依赖外部服  
务么？比如中台服务通过 grpc 去调用个账号服务获取用户信息之类的

### 回答

这个还真是有可能的。所以很多公司 **passport** 服务一挂，所有服务就全挂了。