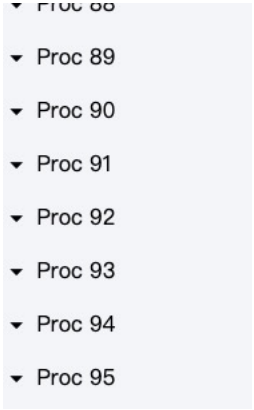
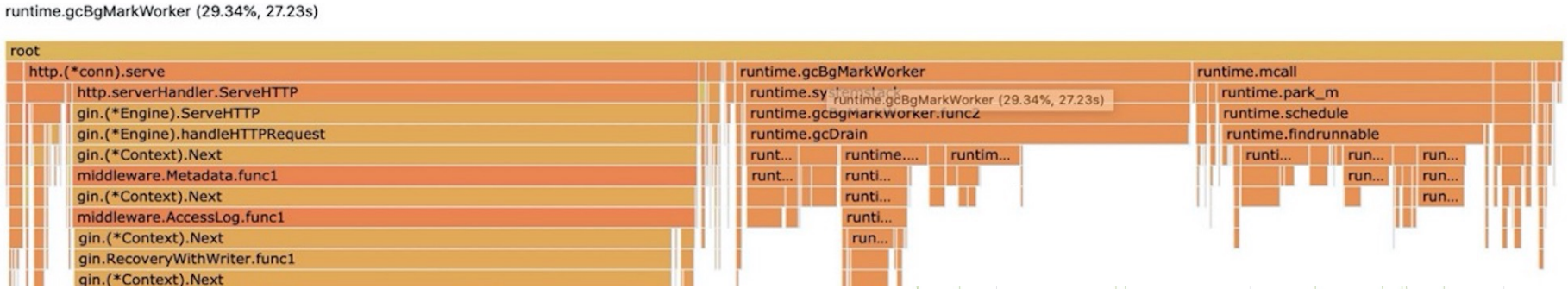
Go scheduler

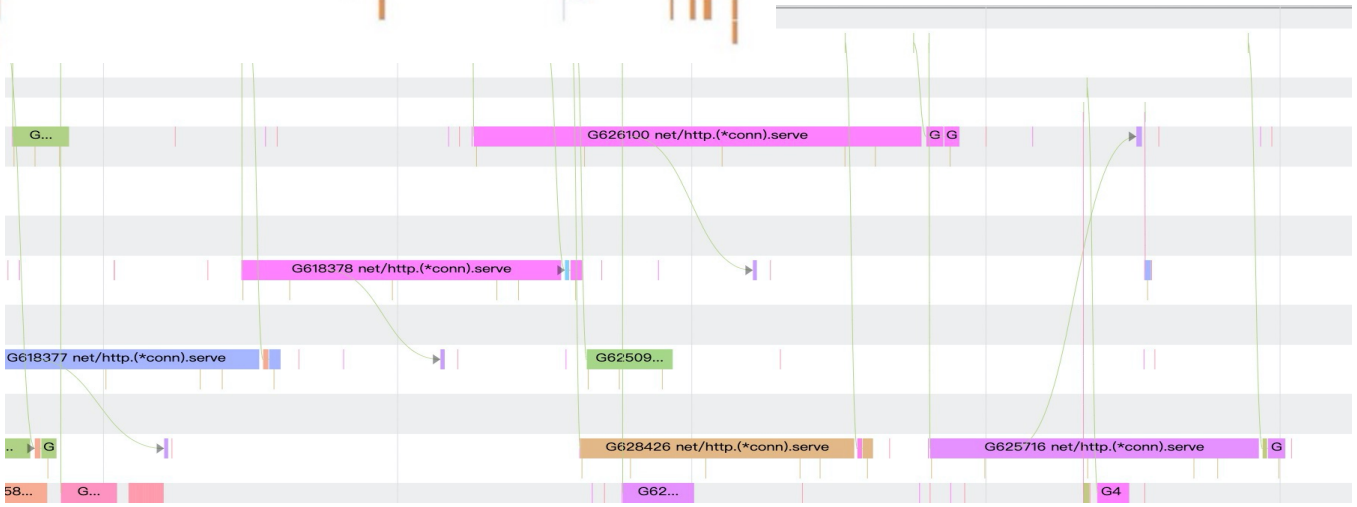
从问题出发

背景

- 在项目上通过go tool trace发现P启的太多了，就设置了MAXPROCS
- 但是除了压测时候改善，日常表现表现的并不明显

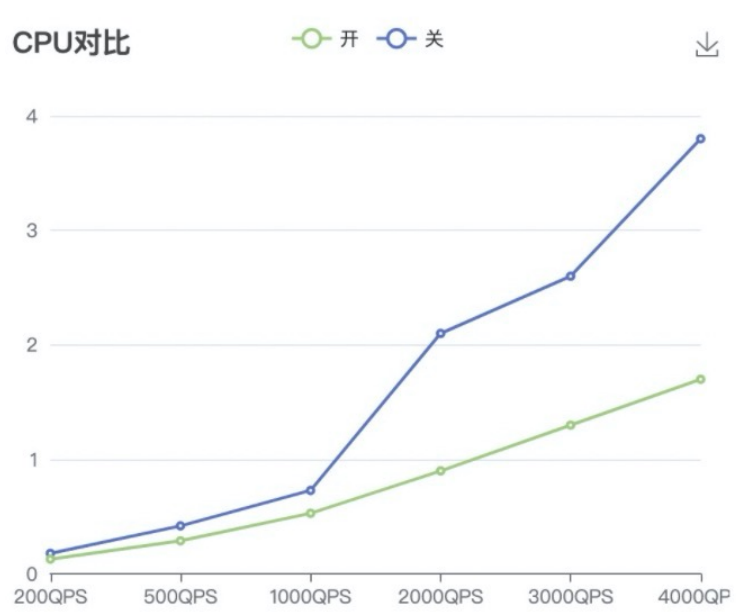


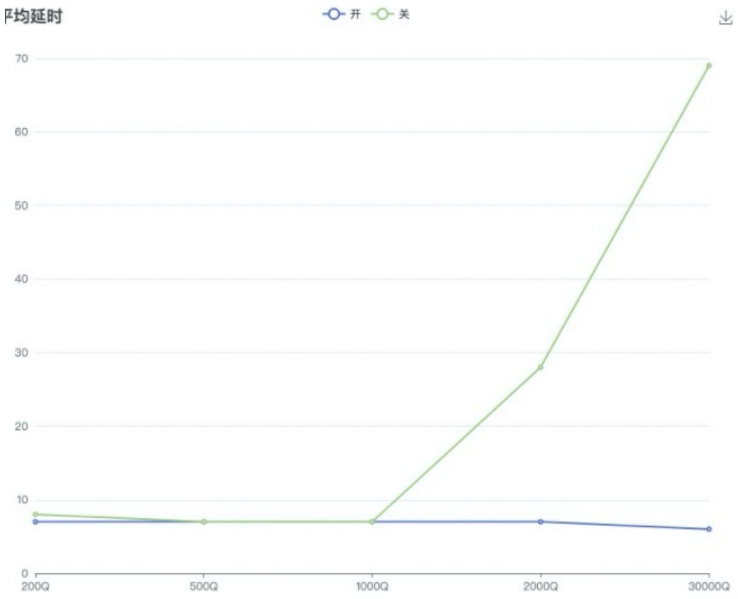


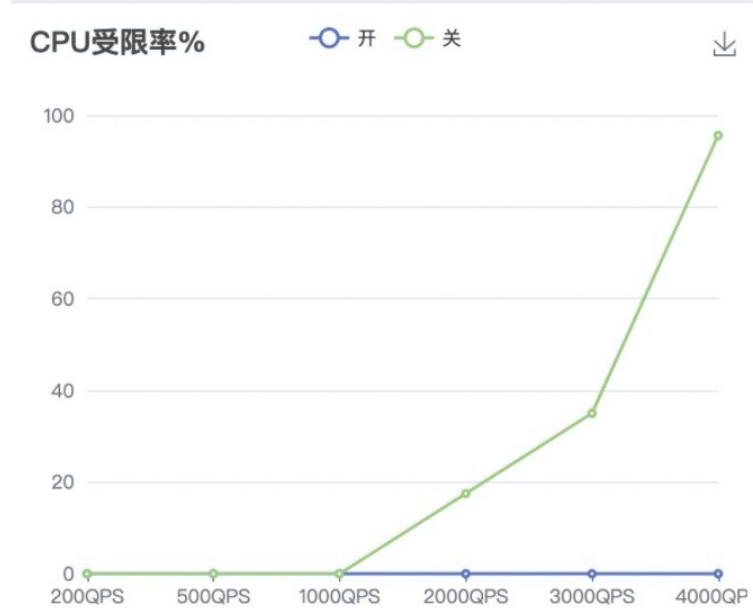


现象

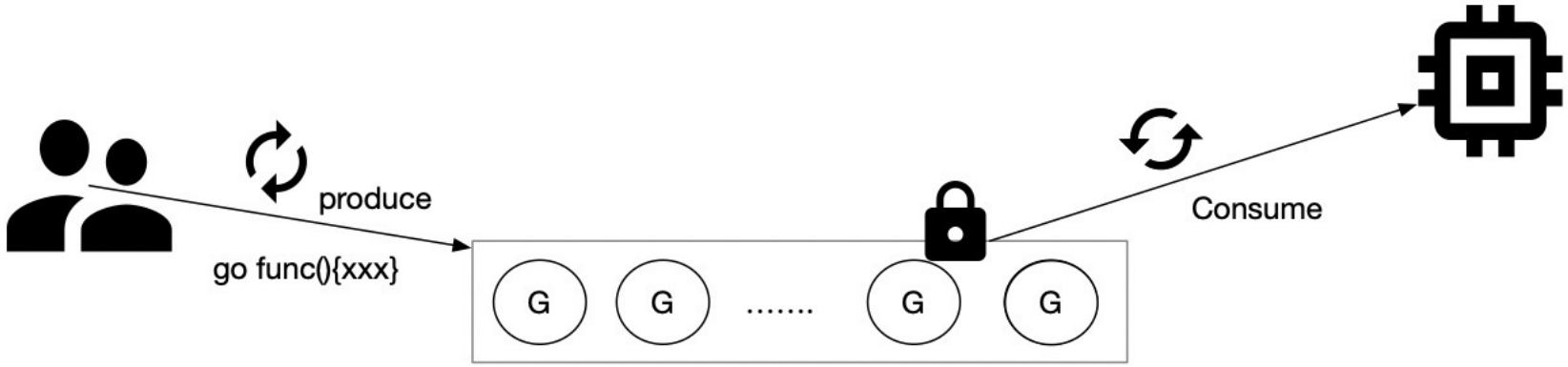
1、 为什么CPU会高
消费者多了自然要
2、 为什么受限率会特别高



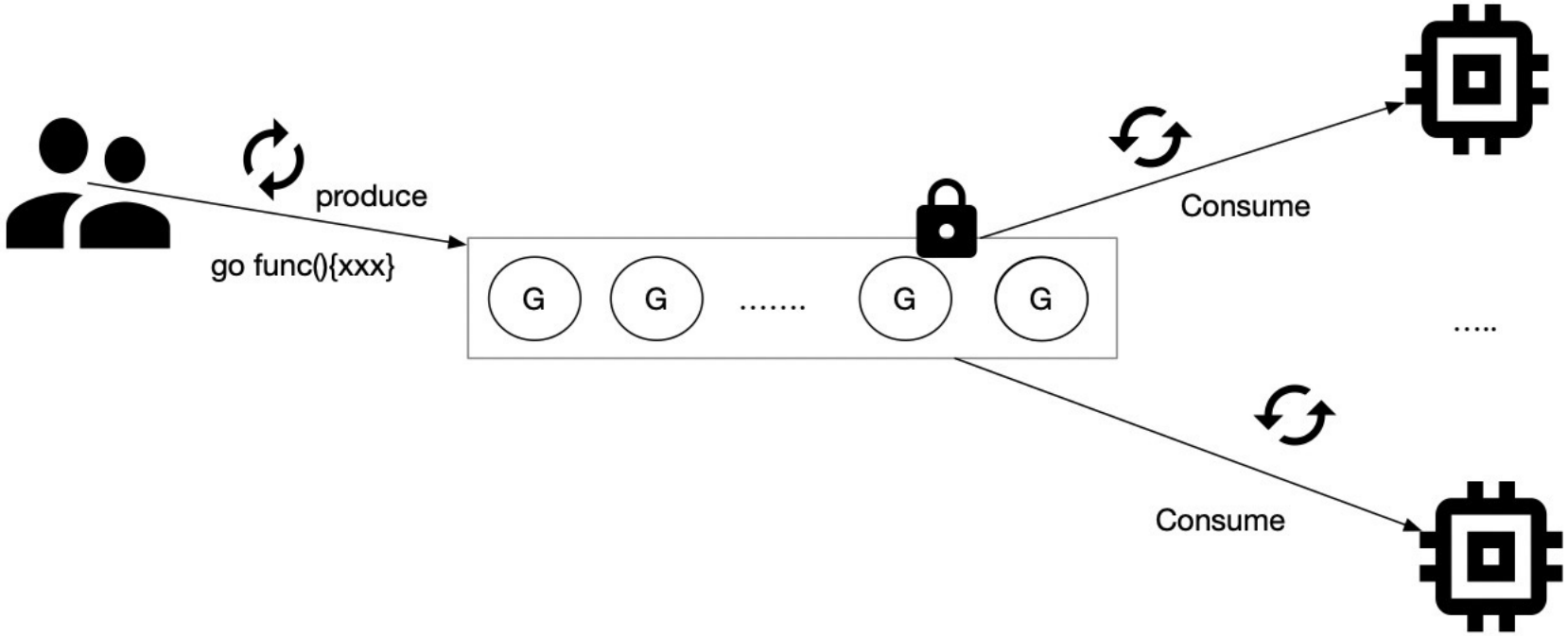




分析



分析



https://wiki.zuoyebang.cc/pages/viewpage.action?pageId=283585159#v2.5.x%E5%8D%87%E7%BA%A7-GOMAXProcs

方案

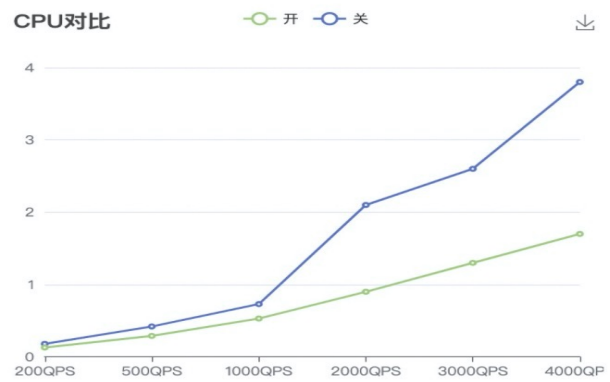
1、 runtime.GOMAXPROCS

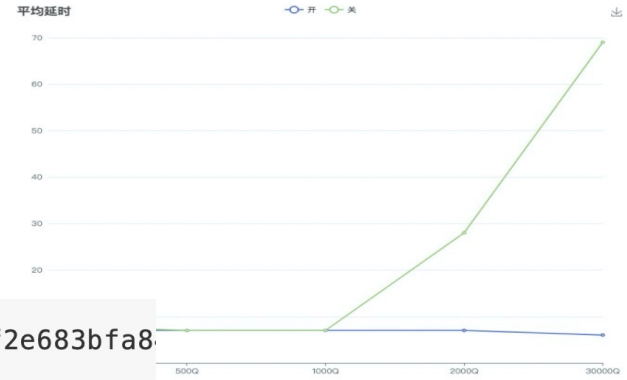
2、 env GOMAXPROCS

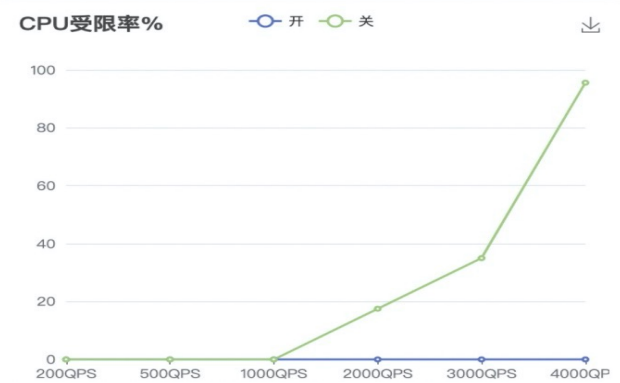
3、

```
import _ "go.uber.org/automaxprocs"
```

```
# ll /sys/fs/cgroup/cpu,cpuacct/docker/bbdadec016e2667a1de24f2e683bfa8
-rw-r--r-- 1 root root 0 11月 11 09:53 cpu.cfs_quota_us
-rw-r--r-- 1 root root 0 11月 11 09:53 cpu.rt_period_us
```







4、 lxcfs

- 通用方案，适合任何语言
- 重新mount部署proc目录mock机器信息
- 守护进程失败会导致当前运行的容器无法获取正确proc信息，需要重新注入

```
/proc/cpuinfo
/proc/diskstats
/proc/meminfo
/proc/stat
/proc/swaps
/proc/uptime
/proc/slabinfo
/sys/devices/system/cpu
/sys/devices/system/cpu/online
```

Go调度

概念介绍

G(goroutine): go关键字生成的用户态任务，由执行的代码和上下文（栈、代码位置）组成

M(machine): kernel thread， 和通过clone创建的thread没有区别

P(processer): 虚拟概念，M需要获得P才能执行任务，G需要被绑定到P上才能被执行，以及放G运行的资源

https://wiki.zuoyebang.cc/pages/viewpage.action?pageId=297064481

Go调度

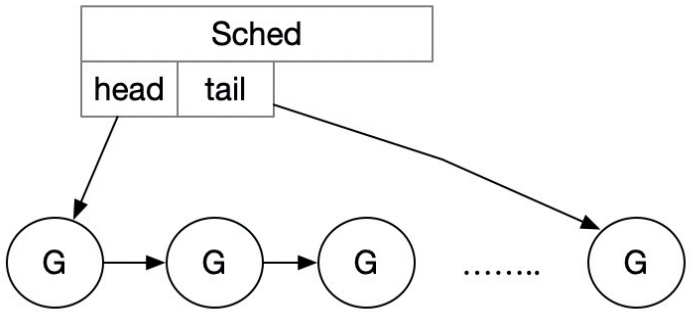
历史小科普

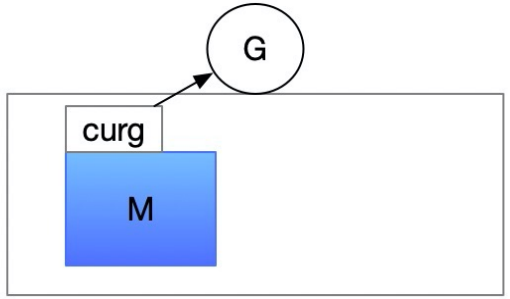
Go 1.1之前没有P的：

每次任务必须执行完！

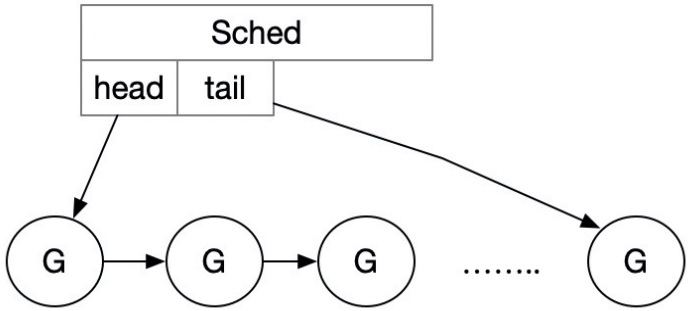
简单GM模型如左：

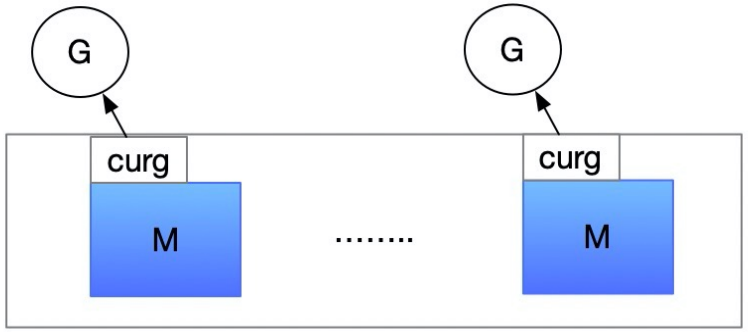
0.x





1.0

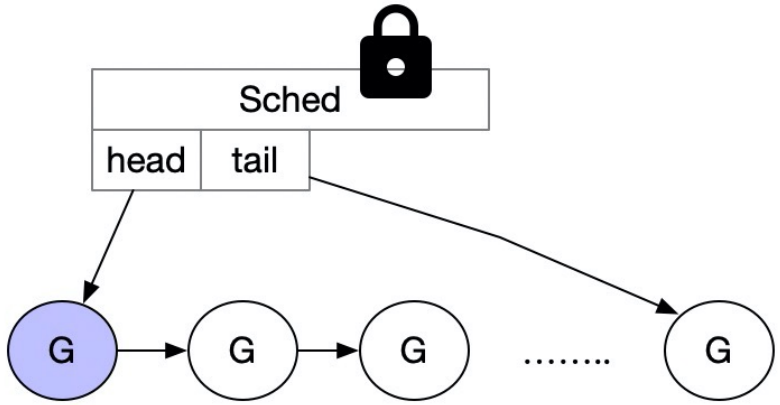


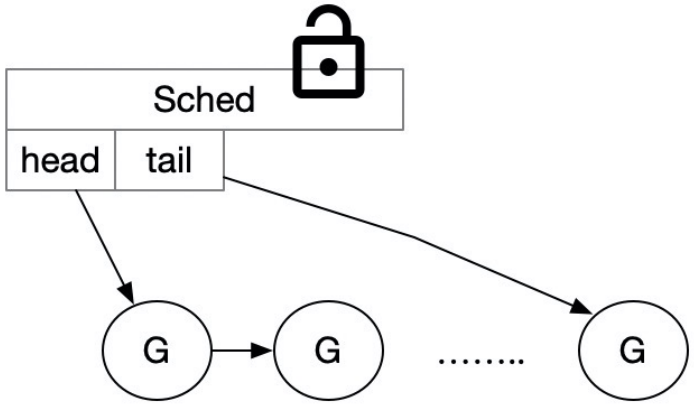


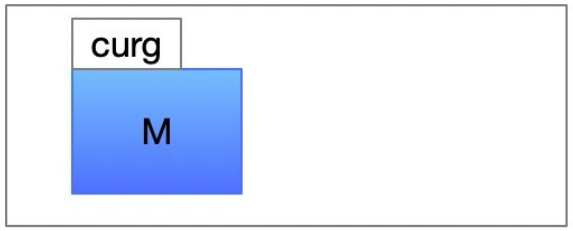
Go调度

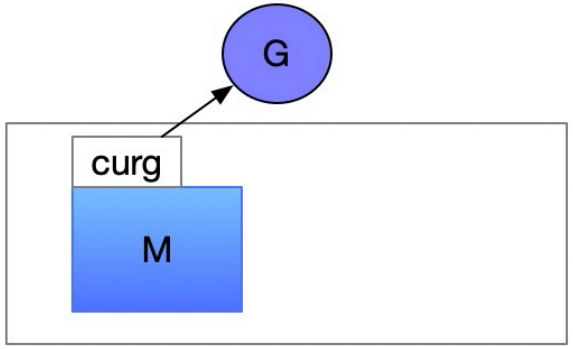
历史小科普

执行完了找下一个









Go调度

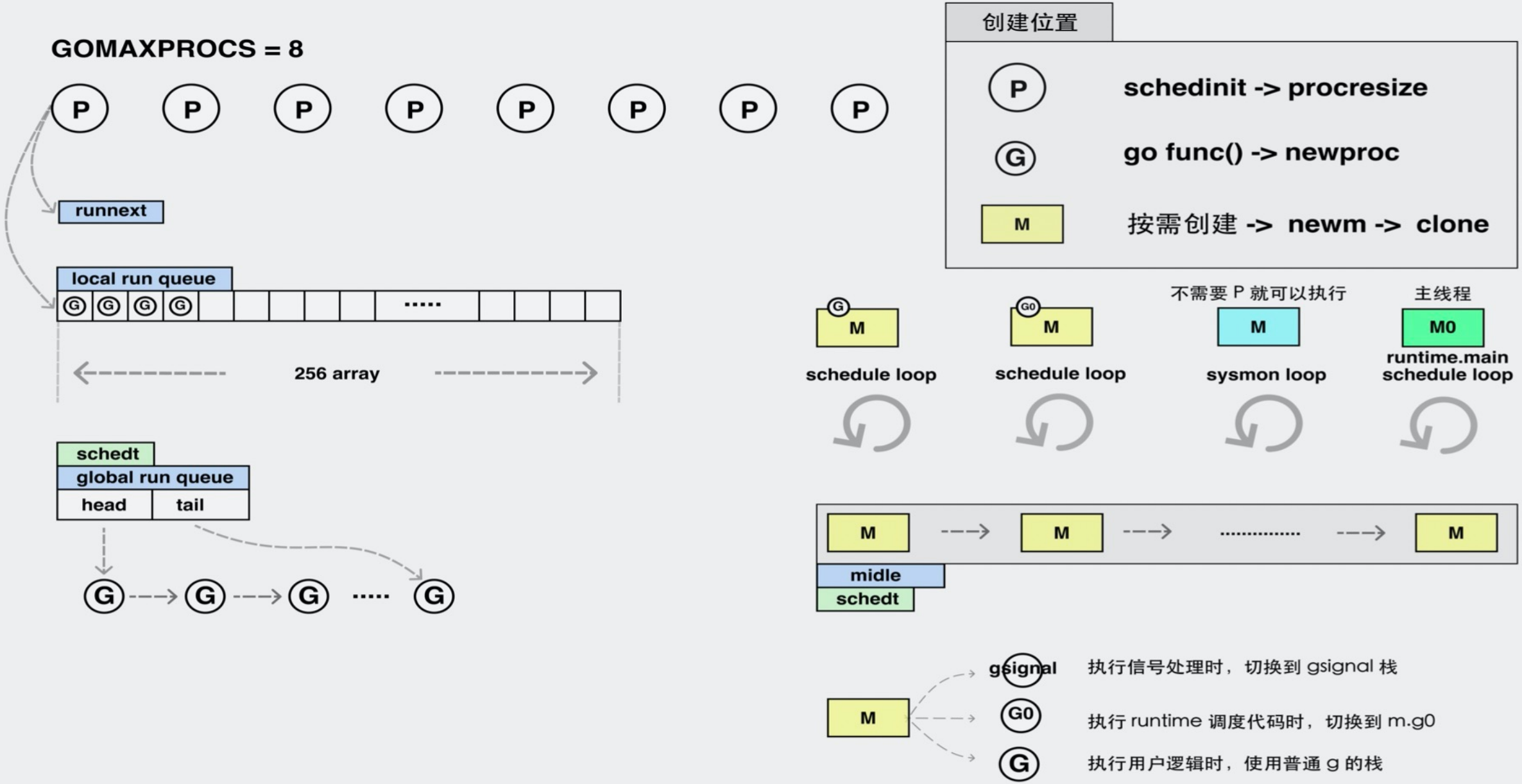
G-M模型的问题

- 全局大锁 + 中心化状态：这个大锁保护goroutine相关的所有操作，比如创建、完成以及调度等
- 频繁的换入换出： goroutine频繁的换入换出会导致延时增加以及额外的开销

为什么要有P

- 作用：P是执行Go代码所需的资源。
- 变化：一些全局变量放到了P上，一些放到M的变量也放到了P上
- 用法：当G要被执行的时候，必须获得P

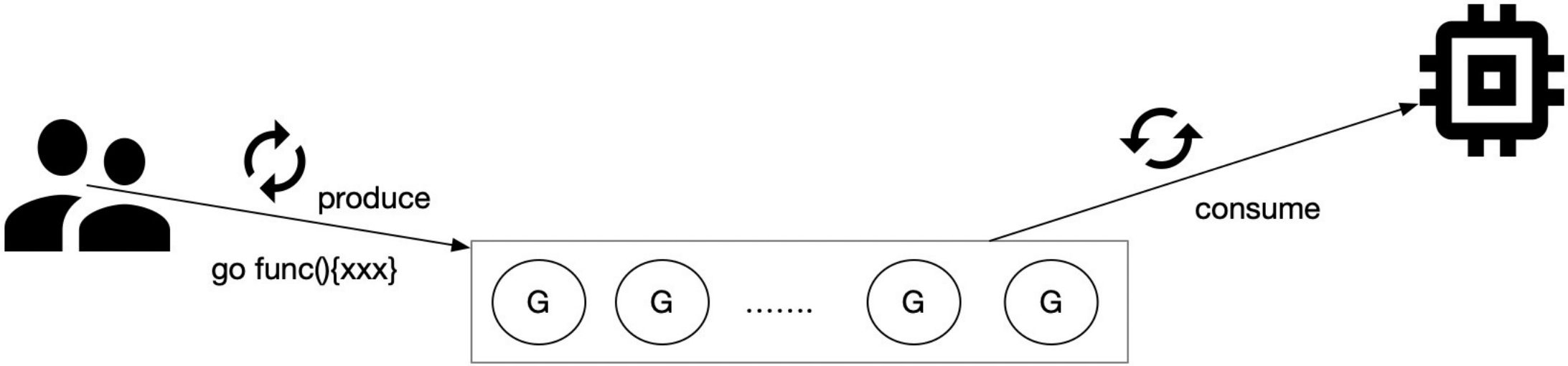
https://docs.google.com/document/d/1TTj4T2JO42uD5ID9e89oa0sLKhJYD0Y_kqxDv3I3XMw



从今年gopher china上一个PPT上抄的图

Go调度

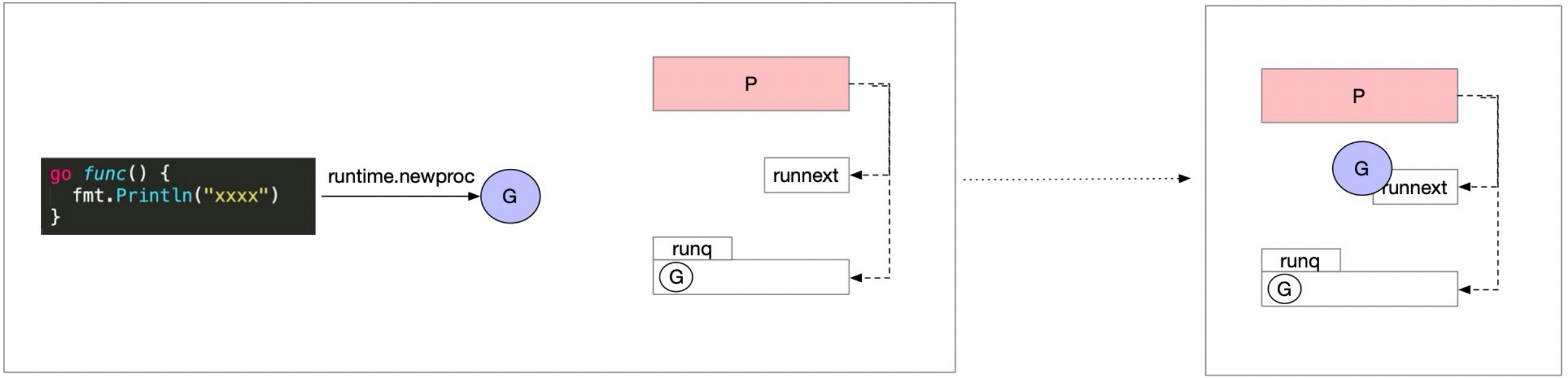
生产和消费



Go调度

生产端逻辑

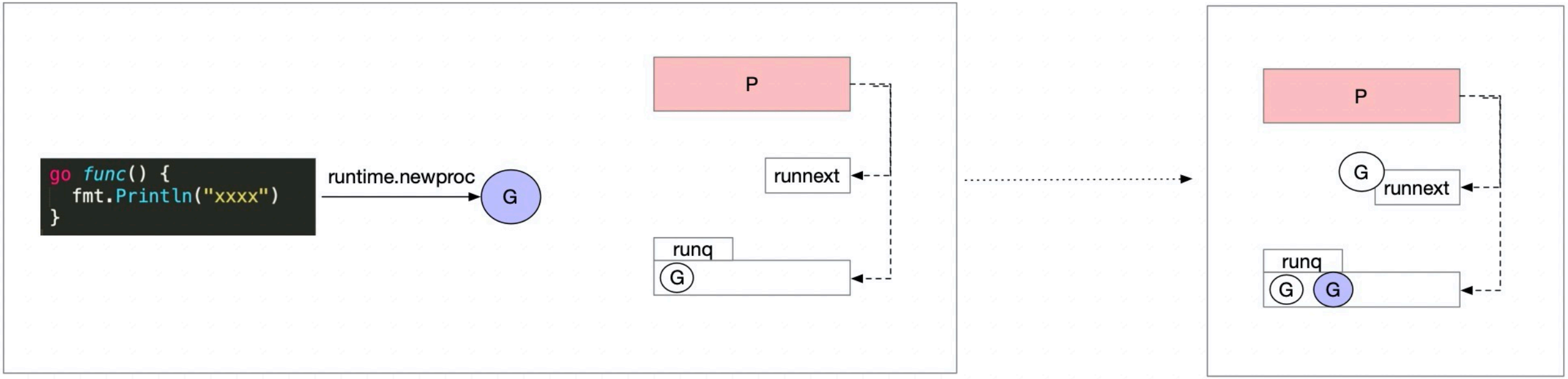
1、p.runnext = nil



Go调度

生产端逻辑

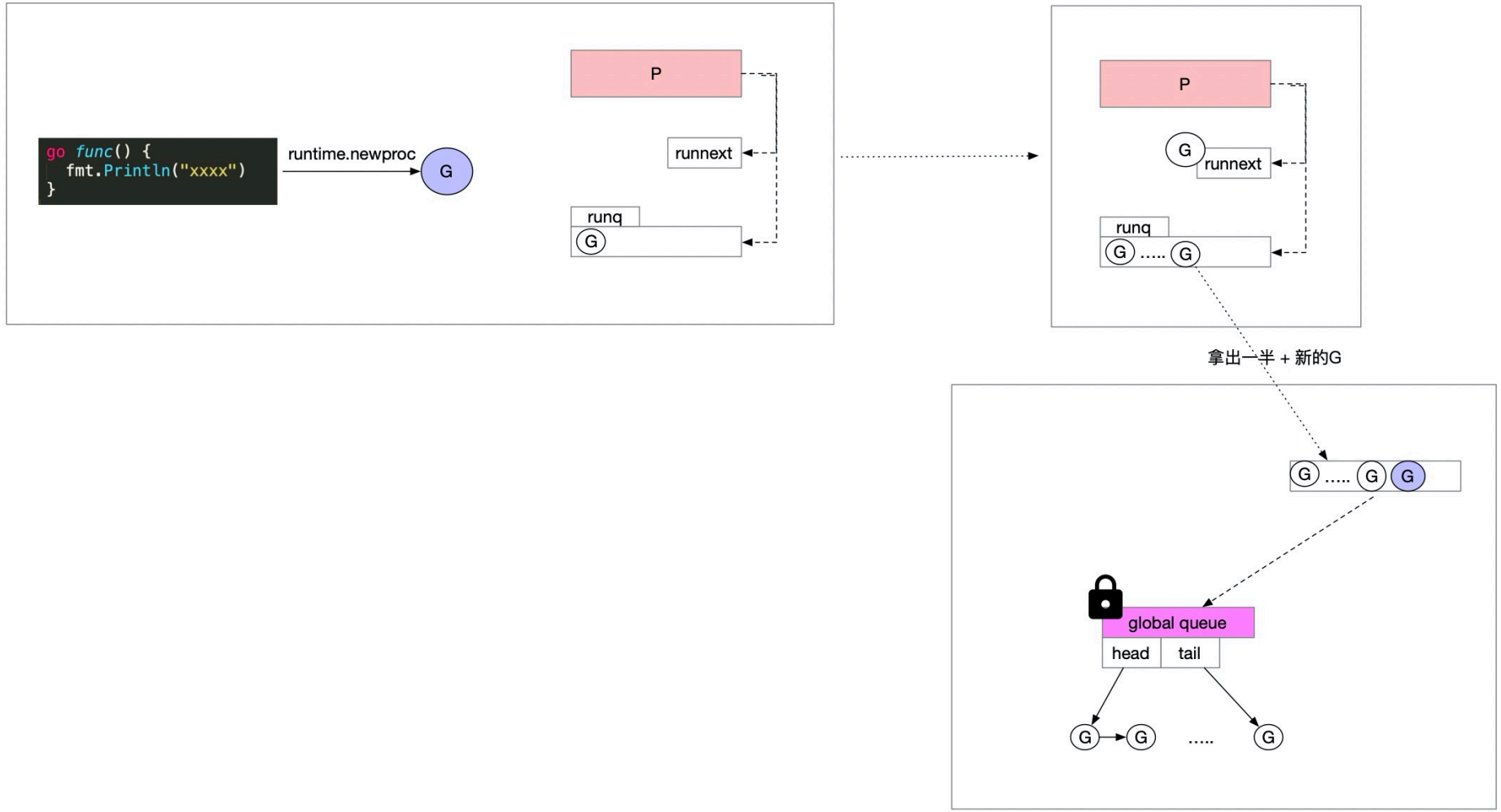
2、p.runnext != nil && p.runq 没满



Go调度

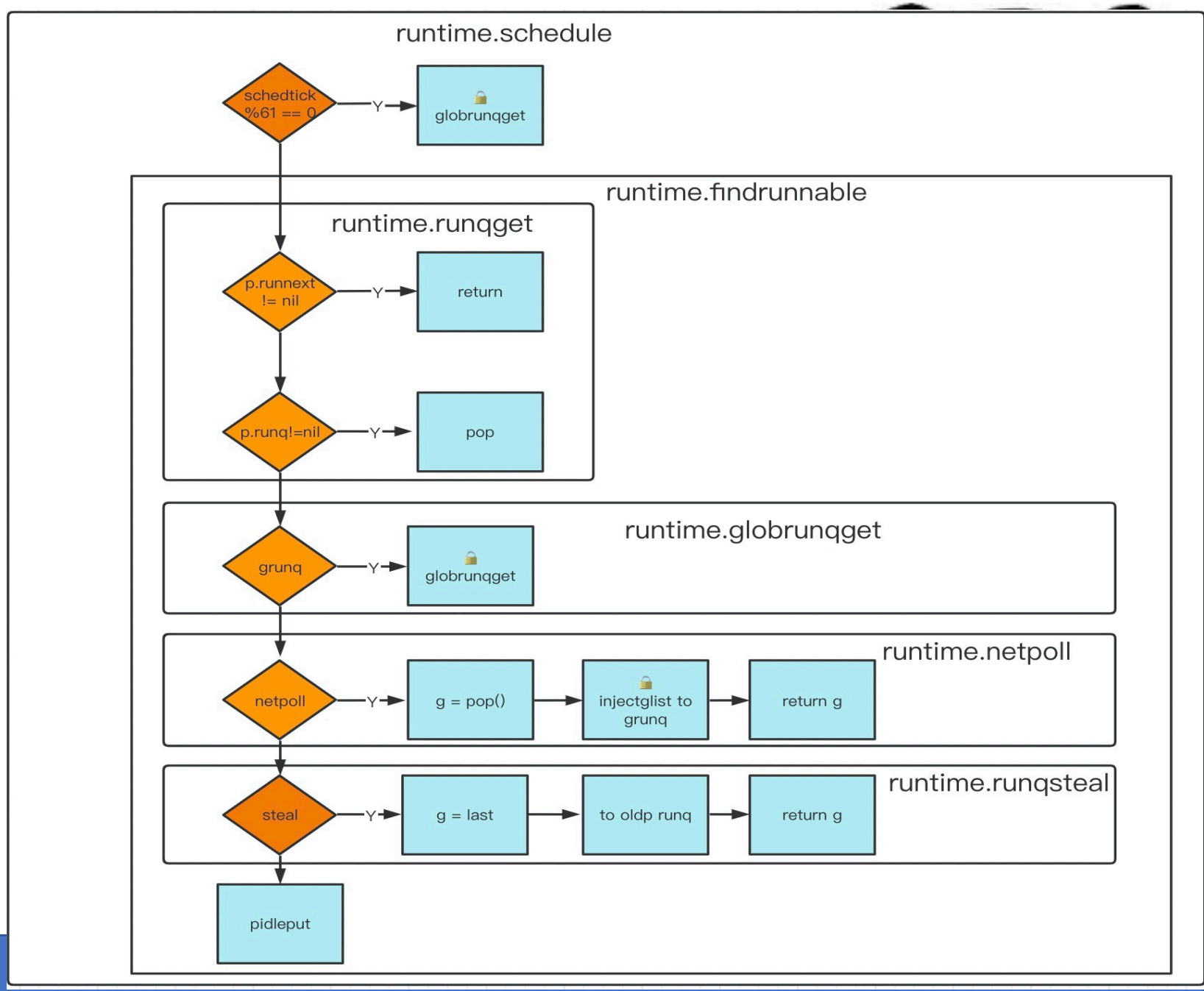
生产端逻辑

3、p.runnext != nil && p.runq 满



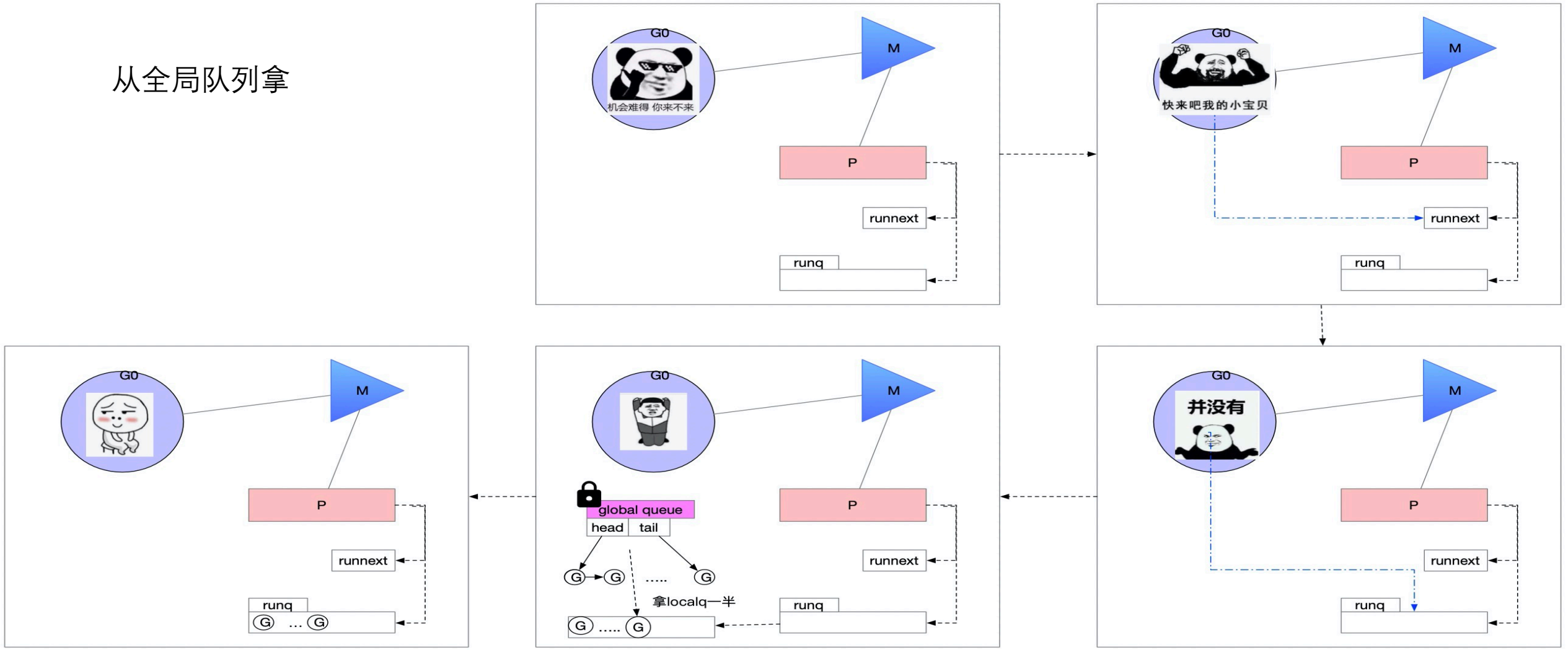
Go调度

消费端逻辑


runtime.schedule
schedtick %61 == 0
Y
globrunqget
runtime.findrunnable
runtime.runqget
p.runnext != nil
Y
return
p.runq!=nil
Y
pop
runtime.globrunqget
grunq
Y
globrunqget
runtime.netpoll
netpoll
Y
g = pop()
injectglist to grunq
return g
runtime.runqsteal
steal
Y
g = last
to oldp runq
return g
pidleput

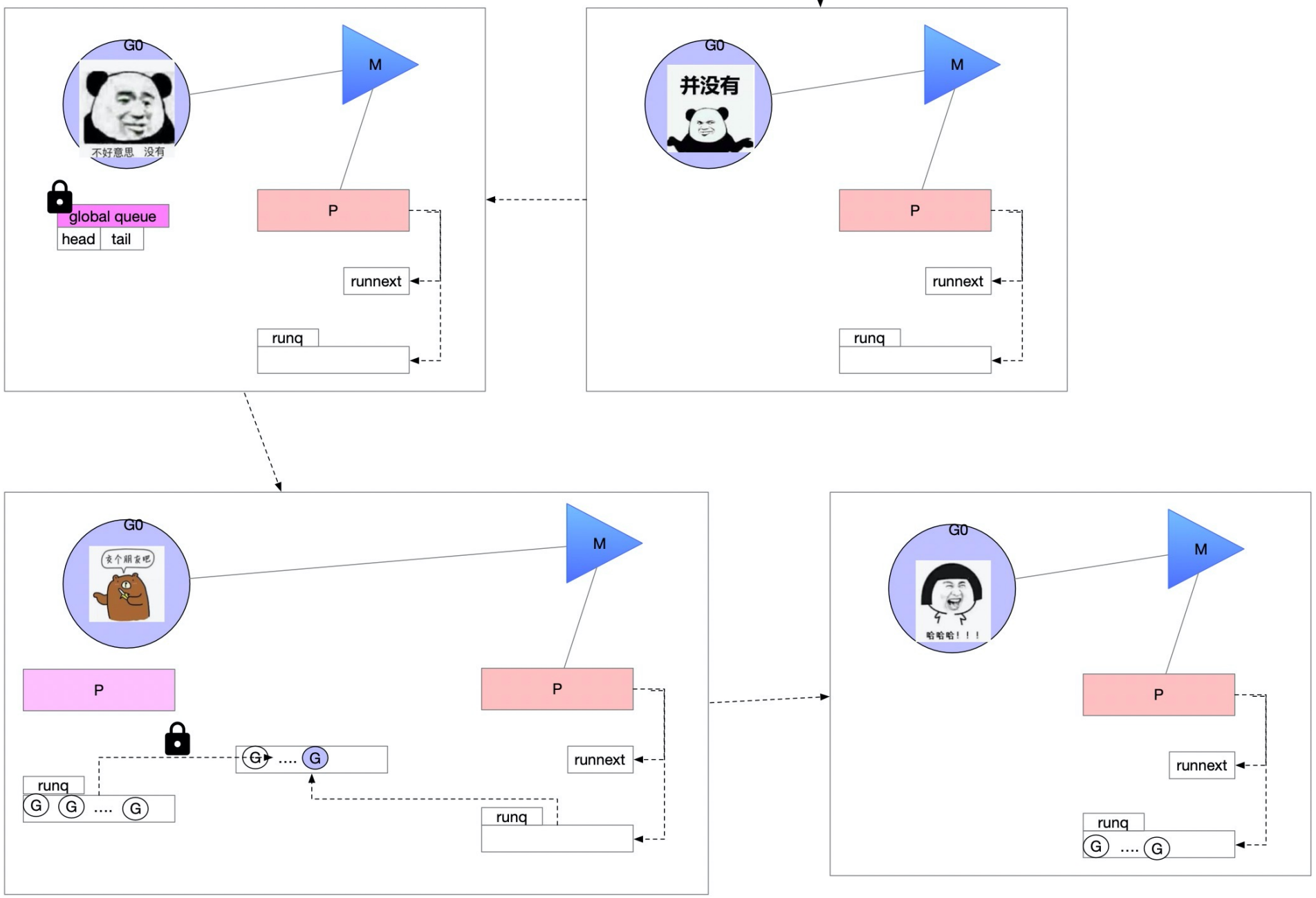
Go调度

从全局队列拿


G0
M
P
runnext
runq
机会难得 你来不来
快来吧我的小宝贝
并没有
global queue
head
tail
拿localq一半

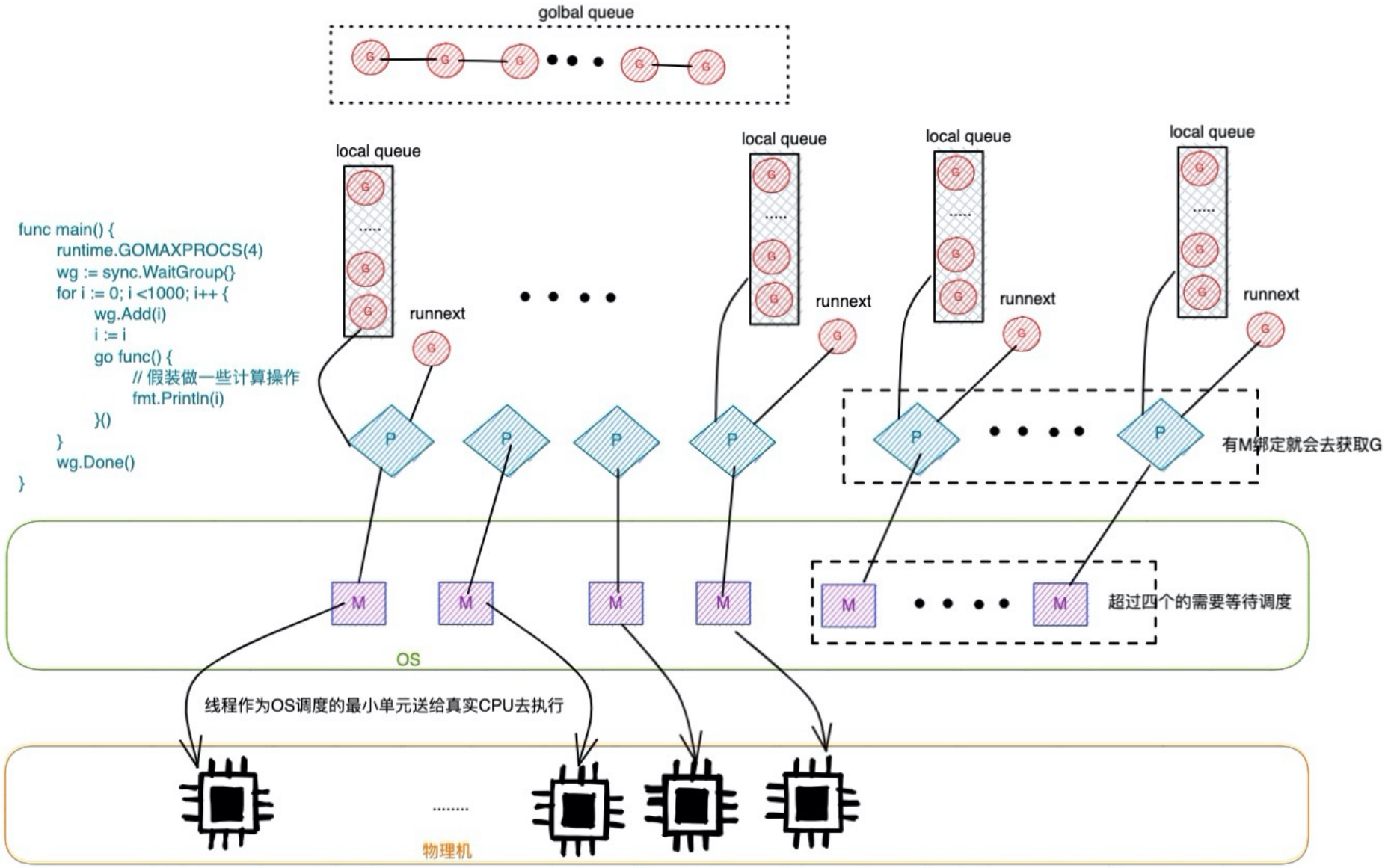
Go调度

消费端逻辑 – steal working



Go调度

最后

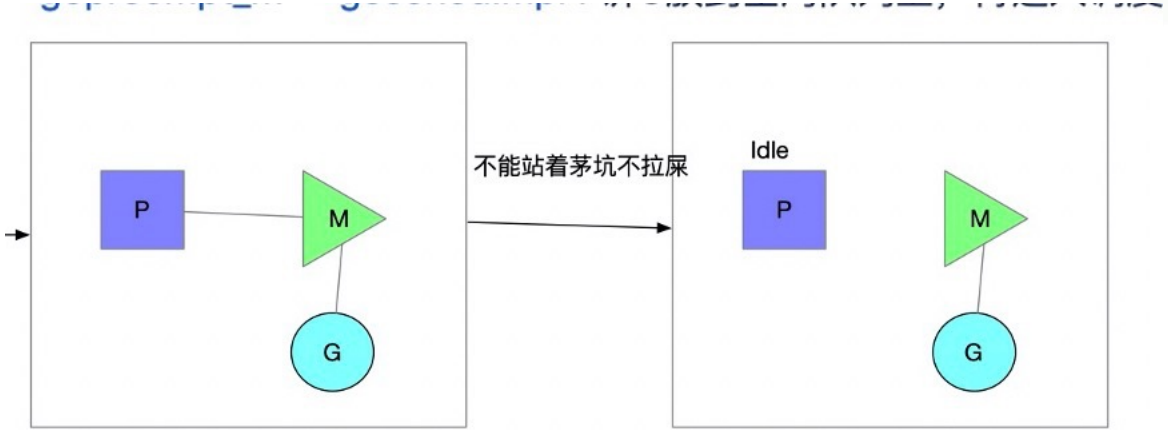

golbal queue
local queue
local queue
local queue
local queue
runnext
runnext
runnext
runnext
func main() {
runtime.GOMAXPROCS(4)
wg := sync.WaitGroup{}
for i := 0; i <1000; i++ {
wg.Add(i)
i := i
go func() {
// 假装做一些计算操作
fmt.Println(i)
}()
}
wg.Done()
}
有M绑定就会去获取G
超过四个的需要等待调度
OS
线程作为OS调度的最小单元送给真实CPU去执行
物理机

Go调度

如何处理阻塞

会阻塞的场景

- System call
- Channel
- Cgo
- preempt



gopark + goready

Go调度

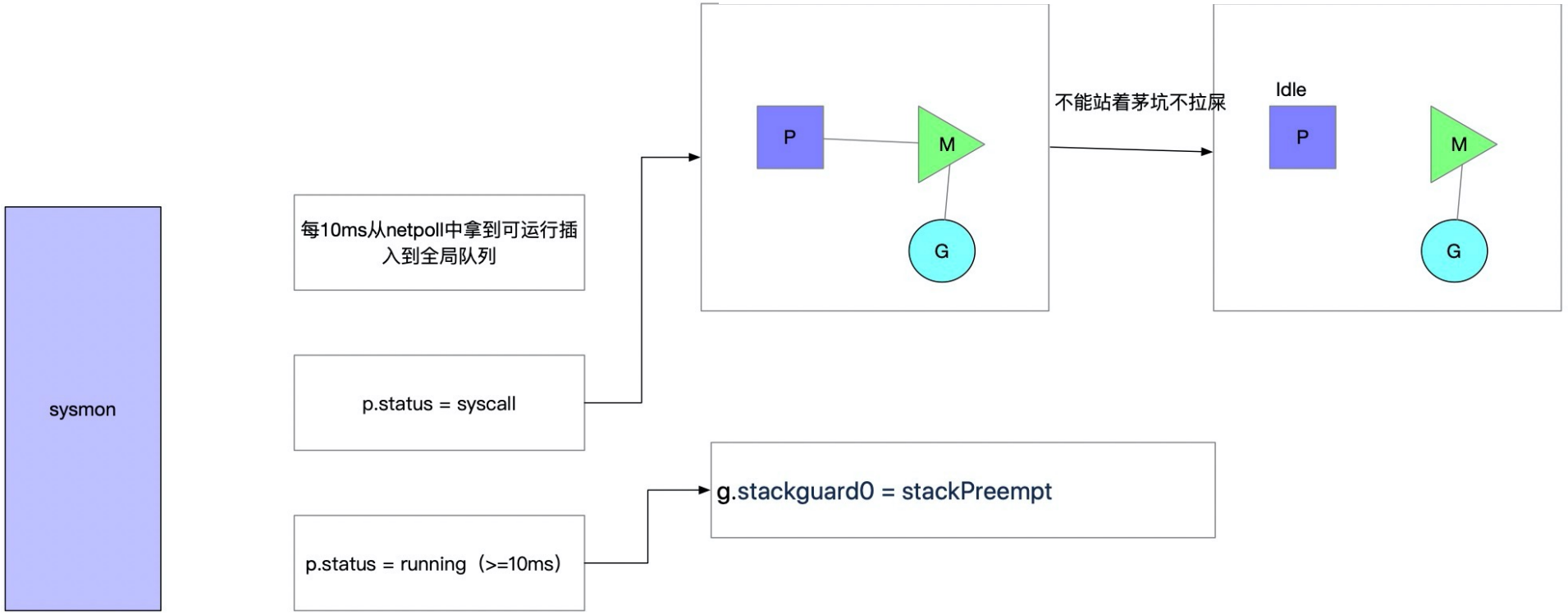
抢占是如何发生的

触发抢占的地方（将g.stackguard0 = stackPreempt）：

- sysmon中如果发现一个G运行了超过了10ms就调用preemptone触发抢占
- 当GC的时候调用preemptall把所有的G都标记为抢占

执行抢占流程：

- 编译器会在函数调用前插入morestack，
- morestack → newstack： 如果g.stackguard0等于stackPreempt走下面流程进行抢占
- gopreempt_m → goschedImpl：讲G放到全局队列上，再进入调度循环



Go调度

抢占是如何发生的

```
helios@heliosdeMacBook-Pro ~/Desktop/helios/test-go/interface_tt go1.13 tool compile -S main.go | grep main.go:13 | grep -v FUNCDATA | grep  -v  PCDATA
    0x0000 00000 (main.go:13)       TEXT    "".a(SB), ABIInternal, $8-0
    0x0000 00000 (main.go:13)       MOVQ    (TLS), CX
    0x0009 00009 (main.go:13)       CMPQ    SP, 16(CX)
    0x000d 00013 (main.go:13)       JLS     41
    0x000f 00015 (main.go:13)       SUBQ    $8, SP
    0x0013 00019 (main.go:13)       MOVQ    BP, (SP)
    0x0017 00023 (main.go:13)       LEAQ    (SP), BP
    0x0029 00041 (main.go:13)       CALL    runtime.morestack_noctxt(SB)
    0x002e 00046 (main.go:13)       JMP     0
```

```
// func morestack_noctxt()
TEXT runtime·morestack_noctxt(SB),NOSPLIT|NOFRAME,$0-0
        MOV     ZERO, CTXT
        JMP     runtime·morestack(SB)
```

```
TEXT runtime·morestack(SB),NOSPLIT|NOFRAME,$0-0
    // ...
    BL  runtime·newstack(SB)

    RET
```

```
987             // NOTE: stackguard0 may change underfoot, if another thread
988             // is about to try to preempt gp. Read it just once and use that same
989             // value now and below.
990             preempt := atomic.Loaduintptr(&gp.stackguard0) == stackPreempt
991
```

```go
package main

import (
  "fmt"
)

//go:noinline
func b() {
  // TODO
}

//go:noinline
func a() {
  // some op
  b()
}

func main() {
  go func() {
    a()
  }()

  fmt.Println( a...: "ending...")
}
```

https://github.com/golang/go/blob/0d0193409492b96881be6407ad50123e3557fdfb/src/runtime/stack.go

Go调度

基于主动让出的协作式调度有什么问题？

1.14之前的所谓协作式就是需要主动让出，但是如果一个goroutine因为代码死循环不能主动让出，那么这个任务就一直不会被让出（能保证自己不写死循环，不能保证依赖方也不写）。比如下面的代码在1.13和1.14运行处不同的结果：

```
func main() {
    n := runtime.GOMAXPROCS(0)
    for i := 0; i < n; i++ {
        go func() {
            for {}
        }()
    }

    time.Sleep(1 * time.Millisecond)
    fmt.Println("end")
}
```

Go1.14回输出end，但是1.13会一直卡着。

其他人踩过的坑：

1、踩坑记#2：Go服务锁死

2、“???”引发的线上事故

代码写出比较低级bug，往依赖服务上甩不甩，this is question。

https://github.com/golang/go/issues/10958

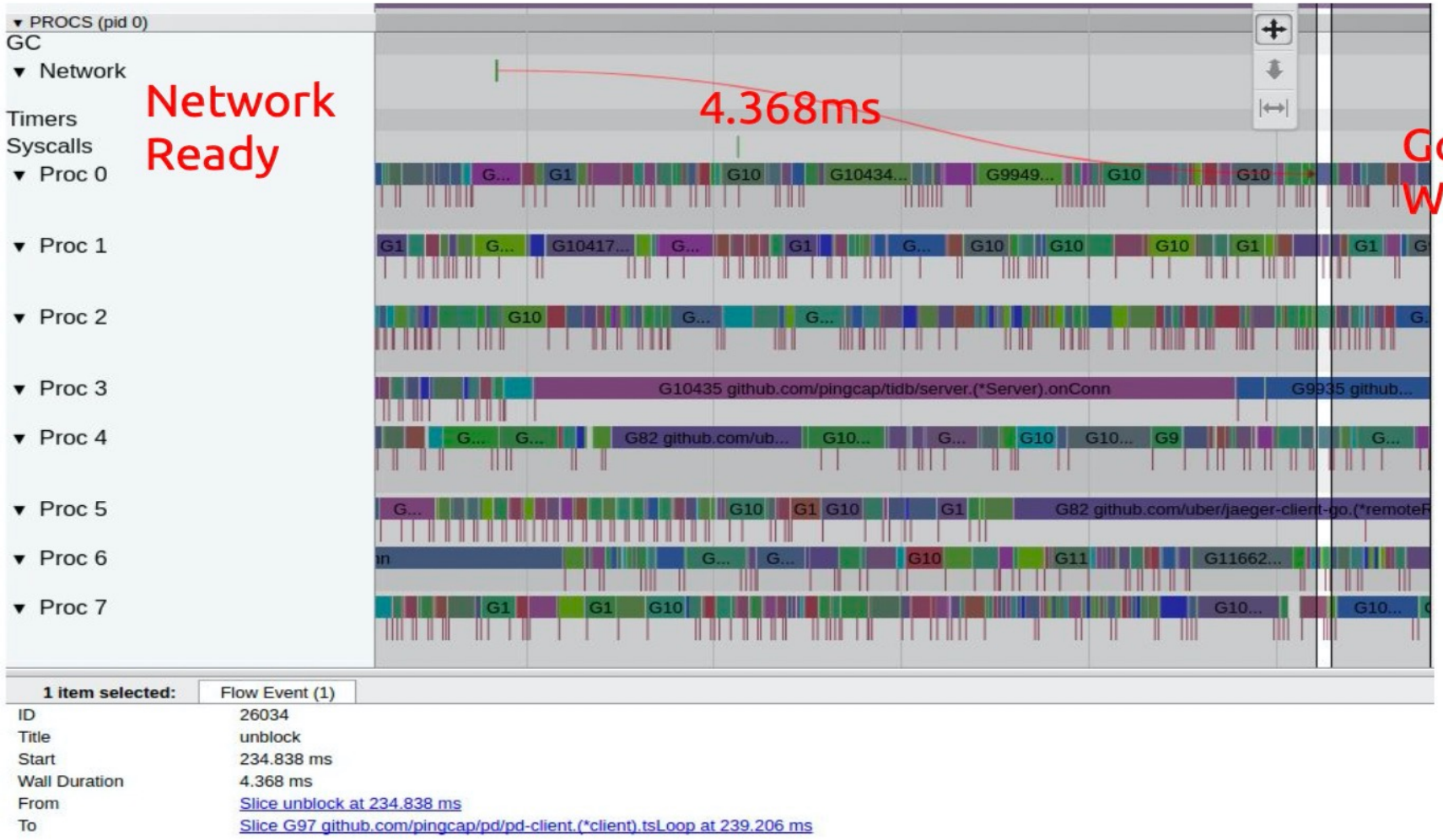
Go调度

无聊的比较

	Goroutine	Thread
内存占用	2KB -> 1GB	从 8k 开始，服务端程序上限很多是 8M(用 ulimit -a 可看)，调用多会 stack overflow
Context switch	几十 NS 级	1-2 us
由谁管理	Go runtime	操作系统
通信方式	CSP/传统共享内存	传统共享内存
ID	有，用户无法访问	有
抢占	1.13 以前需主动让出 1.14 开始可由信号中断	内核抢占

一些小问题

代码执行耗时高，
还是调度延时比较高



解决方式：
别让Go服务的CPU超过50%

https://www.zenlife.tk/go-scheduler-pitfall.md

一些小问题

创建出来的线程永远不会退出


```go
// Monitor M thread and reduce them if its number exceeds setting max limitation.
func monitorAndReduceMThread() {
    maxLimitation := int(atomic.LoadInt64((*int64)(unsafe.Pointer(&threadMaxLimitation))))
    mThreadNum, _ := runtime.ThreadCreateProfile(nil)
    reduce := (mThreadNum - maxLimitation) / 3
    if reduce > 0 {
        wg := sync.WaitGroup{}
        wg.Add(reduce)
        for i := 0; i < reduce; i++ {
            go func() {
                runtime.LockOSThread()
                wg.Done()
            }()
        }
        wg.Wait()
    }
}
```

https://github.com/golang/go/issues/14592#issuecomment-692450566