

- [关于我](#)
- [文章列表](#)

也谈goroutine调度器

- 六月 23, 2017
- [8 条评论](#)

[Go语言](#)在2016年再次拿下[TIBOE](#)年度编程语言称号，这充分证明了Go语言这几年在全世界范围内的受欢迎程度。如果要对世界范围内的gopher发起一次“你究竟喜欢Go的哪一点”的调查，我相信很多Gopher会提到：**goroutine**。

[Goroutine](#)是[Go语言](#)原生支持并发的具体实现，你的Go代码都无一例外地跑在goroutine中。你可以启动许多甚至成千上万的goroutine，Go的runtime负责对goroutine进行管理。所谓的管理就是“调度”，粗糙地说调度就是决定何时哪个goroutine将获得资源开始执行、哪个goroutine应该停止执行让出资源、哪个goroutine应该被唤醒恢复执行等。goroutine的调度是Go team care的事情，大多数gopher们无需关心。但个人觉得适当了解一下Goroutine的调度模型和原理，对于编写出更好的go代码是大有裨益的。因此，在这篇文章中，我将和大家一起来探究一下goroutine调度器的演化以及模型/原理。

注意：这里要写的并不是对goroutine调度器的源码分析，国内的[雨痕老师](#)在其《[Go语言学习笔记](#)》一书的下卷“源码剖析”中已经对Go 1.5.1的scheduler实现做了细致且高质量的源码分析了，对Go scheduler的实现特别感兴趣的gopher可以移步到这本书中去^0^。这里关于goroutine scheduler的介绍主要是参考了Go team有关scheduler的各种design doc、国外Gopher发表的有关scheduler的资料，当然雨痕老师的书也给了我很多的启示。

一、Goroutine调度器

提到“调度”，我们首先想到的就是操作系统对进程、线程的调度。操作系统调度器会将系统中的多个线程按照一定算法调度到物理CPU上去运行。传统的编程语言比如[C](#)、[C++](#)等的并发实现实际上就是基于操作系统调度的，即程序负责创建线程(一般通过pthread等lib调用实现)，操作系统负责调度。这种传统支持并发的方式有诸多不足：

- 复杂
 - 创建容易，退出难：做过[C/C++ Programming](#)的童鞋都知道，创建一个thread(比如利用pthread)虽然参数也不少，但好歹可以接受。但一旦涉及到thread的退出，就要考虑thread是detached，还是需要parent thread去join？是否需要在thread中设置cancel point，以保证join时能顺利退出？
 - 并发单元间通信困难，易错：多个thread之间的通信虽然有多种机制可选，但用起来是相当复杂；并且一旦涉及到shared memory，就会用到各种lock，死锁便成为家常便饭；
 - thread stack size的设定：是使用默认的，还是设置的大一些，或者小一些呢？
- 难于scaling
 - 一个thread的代价已经比进程小了很多了，但我们依然不能大量创建thread，因为除了每个thread占用的资源不小之外，操作系统调度切换thread的代价也不小；
 - 对于很多网络服务程序，由于不能大量创建thread，就要在少量thread里做网络多路复用，即：使用epoll/kqueue/IOCompletionPort这套机制，即便有[libevent/libev](#)这样的第三方库帮忙，写起这样的程序也是很不易的，存在大量callback，给程序员带来不小的心智负担。

为此，Go采用了用户层轻量级**thread**或者说是类**coroutine**的概念来解决这些问题，Go将之称为”**goroutine**“。goroutine占用的资源非常小([Go 1.4](#)将每个goroutine stack的size默认设置为2k)，goroutine调度的切换也不用陷入(trap)操作系统内核层完成，代价很低。因此，一个Go程序中可以创建成千上万个并发的goroutine。所有的Go代码都在goroutine中执行，哪怕是go的runtime也不例外。将这些goroutines按照一定算法放到“*CPU*”上执行的程序就称为**goroutine**调度器或**goroutine scheduler**。

不过，一个Go程序对于操作系统来说只是一个用户层程序，对于操作系统而言，它的眼中只有thread，它甚至不知道有什么叫Goroutine的东西的存在。goroutine的调度全要靠Go自己完成，实现Go程序内goroutine之间“公平”的竞争“CPU”资源，这个任务就落到了Go runtime头上，要知道在一个Go程序中，除了用户代码，剩下的就是go runtime了。

于是Goroutine的调度问题就演变为go runtime如何将程序内的众多goroutine按照一定算法调度到“CPU”资源上运行了。在操作系统层面，Thread竞争的“CPU”资源是真实的物理CPU，但在Go程序层面，各个Goroutine要竞争的”CPU”资源是什么呢？Go程序是用户层程序，它本身整体是运行在一个或多个操作系统线程上的，因此goroutine们要竞争的所谓“CPU”资源就是操作系统线程。这样Go scheduler的任务就明确了：将goroutines按照一定算法放到不同的操作系统线程中去执行。这种在语言层面自带调度器的，我们称之为原生支持[并发](#)。

二、Go调度器模型与演化过程

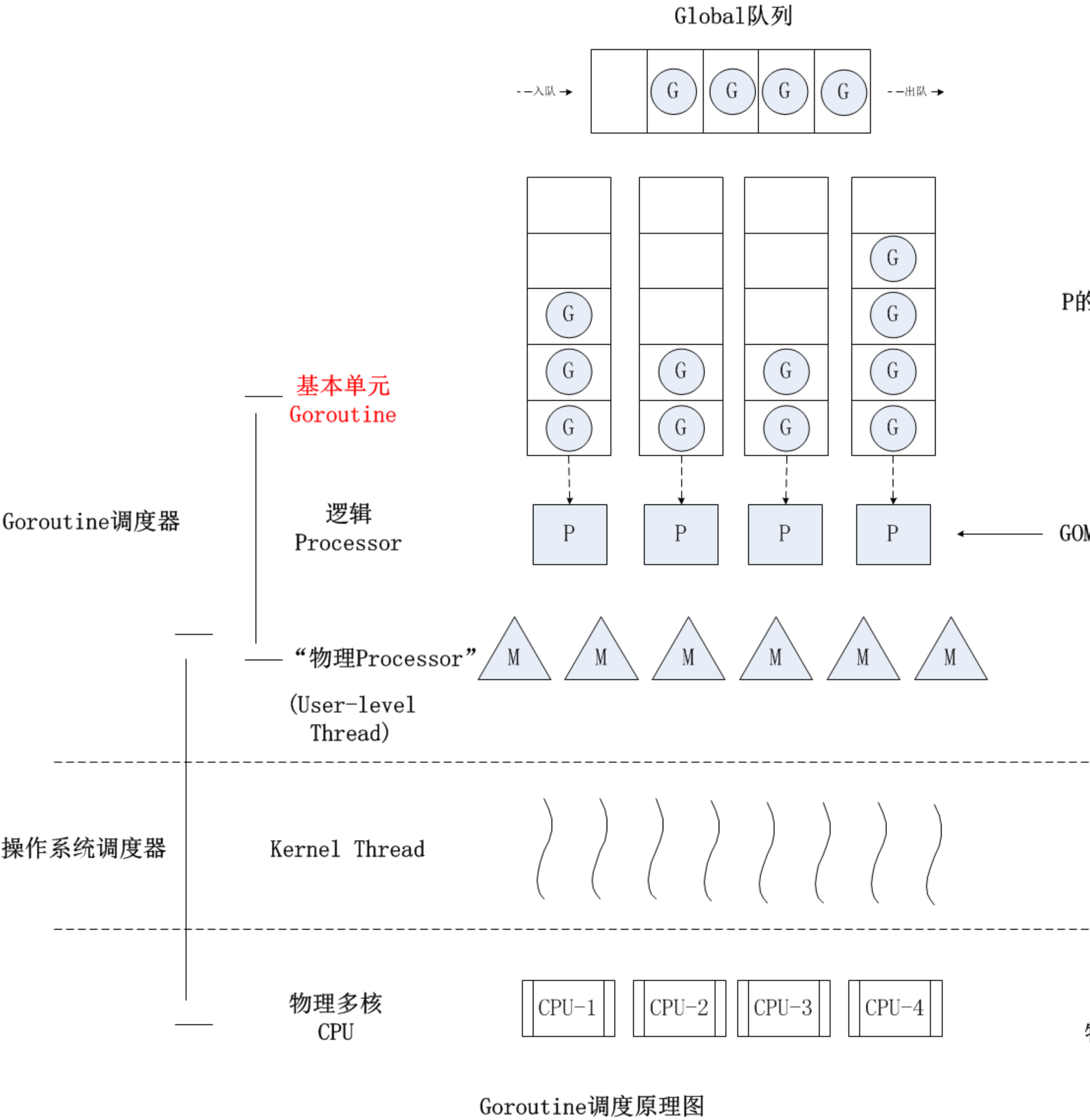
1、G-M模型

2012年3月28日，[Go 1.0正式发布](#)。在这个版本中，Go team实现了一个简单的调度器。在这个调度器中，每个goroutine对应于runtime中的一个抽象结构：G，而os thread作为“物理CPU”的存在而被抽象为一个结构：M(machine)。这个结构虽然简单，但是却存在着许多问题。前Intel blackbelt工程师、现Google工程师[Dmitry Vyukov](#)在其《[Scalable Go Scheduler Design](#)》一文中指出了**G-M**模型的一个重要不足：限制了Go并发程序的伸缩性，尤其是对那些有高吞吐或并行计算需求的服务程序。主要体现在如下几个方面：

- 单一全局互斥锁(Sched.Lock)和集中状态存储的存在导致所有goroutine相关操作，比如：创建、重新调度等都要上锁；
- goroutine传递问题：M经常在M之间传递”可运行”的goroutine，这导致调度延迟增大以及额外的性能损耗；
- 每个M做内存缓存，导致内存占用过高，数据局部性较差；
- 由于syscall调用而形成的剧烈的worker thread阻塞和解除阻塞，导致额外的性能损耗。

2、G-P-M模型

于是Dmitry Vyukov亲自操刀改进Go scheduler，在[Go 1.1](#)中实现了**G-P-M**调度模型和[work stealing算法](#)，这个模型一直沿用至今：



有名人曾说过：“计算机科学领域的任何问题都可以通过增加一个间接的中间层来解决”，我觉得Dmitry Vyukov的**G-P-M**模型恰是这一理论的践行者。Dmitry Vyukov通过向G-M模型中增加了一个P，实现了Go scheduler的scalable。

P是一个“逻辑Processor”，每个G要想真正运行起来，首先需要被分配一个P（进入到P的local runq中，这里暂忽略global runq那个环节）。对于G来说，P就是运行它的“CPU”，可以说：**G的眼里只有P**。但从Go scheduler视角来看，真正的“CPU”是M，只有将P和M绑定才能让P的runq中G得以真实运行起来。这样的P与M的关系，就好比Linux操作系统调度层面用户线程(user thread)与核心线程(kernel thread)的对应关系那样(N x M)。

3、抢占式调度

G-P-M模型的实现算是Go scheduler的一大进步，但Scheduler仍然有一个头疼的问题，那就是不支持抢占式调度，导致一旦某个G中出现死循环或永久循环的代码逻辑，那么G将永久占用分配给它的P和M，位于同一个P中的其他G将得不到调度，出现“饿死”的情况。更为严重的是，当只有一个P时 (GOMAXPROCS=1)时，整个Go程序中的其他G都将“饿死”。于是Dmitry Vyukov又提出了《[Go Preemptive Scheduler Design](#)》并在[Go 1.2](#)中实现了“抢占式”调度。

这个抢占式调度的原理则是在每个函数或方法的入口，加上一段额外的代码，让runtime有机会检查是否需要执行抢占调度。这种解决方案只能说局部解决

了“饿死”问题，对于没有函数调用，纯算法循环计算的G，scheduler依然无法抢占。

4、NUMA调度模型

从Go 1.2以后，Go似乎将重点放在了对GC的低延迟的优化上了，对scheduler的优化和改进似乎不那么热心了，只是伴随着GC的改进而作了些小的改动。Dmitry Vyukov在2014年9月提出了一个新的proposal design doc：《[NUMA-aware scheduler for Go](#)》，作为未来Go scheduler演进方向的一个提议，不过至今似乎这个proposal也没有列入开发计划。

5、其他优化

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

[Ian Lance Taylor](#)在[Go 1.9](#) dev周期中增加了一个[Poller for os package](#)的功能，这个功能可以像netpoller那样，在G操作支持pollable的fd时，仅阻塞G，而不阻塞M。不过该功能依然不能对regular file有效，regular file不是pollable的。不过，对于scheduler而言，这也算是一个进步了。

三、Go调度器原理的进一步理解

1、G、P、M

关于G、P、M的定义，大家可以参见\$GOROOT/src/runtime/runtime2.go这个源文件。这三个struct都是大块儿头，每个struct定义都包含十几个甚至二、三十个字段。像scheduler这样的核心代码向来很复杂，考虑的因素也非常多，代码“耦合”成一坨。不过从复杂的代码中，我们依然可以看出来G、P、M的各自大致用途（当然雨痕老师的源码分析功不可没），这里简要说明一下：

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

下面是G、P、M定义的代码片段：

```
//src/runtime/runtime2.go
type g struct {
    stack      stack    // offset known to runtime/cgo
    sched      gobuf
    goid       int64
    gopc       uintptr // pc of go statement that created this goroutine
    startpc    uintptr // pc of goroutine function
    ... ..
}

type p struct {
    lock mutex

    id         int32
    status     uint32 // one of pidle/prunning/...

    mcache     *mcache
    racectx    uintptr

    // Queue of runnable goroutines. Accessed without lock.
    runqhead uint32
    runqtail uint32
    runq      [256]uintptr

    runnext uintptr

    // Available G's (status == Gdead)
    gfree     *g
    gfreecnt int32

    ... ..
}

type m struct {
    g0      *g    // goroutine with scheduling stack
    mstartfn func()
    curg    *g    // current running goroutine
    .... ..
}
```

2、G被抢占调度

和操作系统按时间片调度线程不同，Go并没有时间片的概念。如果某个G没有进行system call调用、没有进行I/O操作、没有阻塞在一个channel操作上，那么m是如何让G停下来并调度下一个**runnable G**的呢？答案是：G是被抢占调度的。

前面说过，除非极端的无限循环或死循环，否则只要G调用函数，Go runtime就有抢占G的机会。Go程序启动时，runtime会去启动一个名为sysmon的m(一般称为监控线程)，该m无需绑定p即可运行，该m在整个Go程序的运行过程中至关重要：

```
//$GOROOT/src/runtime/proc.go

// The main goroutine.
func main() {
    ... ..
}
```

```
    systemstack(func() {
        newm(sysmon, nil)
    })
    .... ...
}

// Always runs without a P, so write barriers are not allowed.
//
//go:nowritebarrierrec
func sysmon() {
    // If a heap span goes unused for 5 minutes after a garbage collection,
    // we hand it back to the operating system.
    scavengelimit := int64(5 * 60 * 1e9)
    ... ...

    if .... {
        ... ...
        // retake P's blocked in syscalls
        // and preempt long running G's
        if retake(now) != 0 {
            idle = 0
        } else {
            idle++
        }
        ... ...
    }
}
```

sysmon每20us~10ms启动一次，按照《Go语言学习笔记》中的总结，sysmon主要完成如下工作：

- 释放闲置超过5分钟的span物理内存；
- 如果超过2分钟没有垃圾回收，强制执行；
- 将长时间未处理的netpoll结果添加到任务队列；
- 向长时间运行的G任务发出抢占调度；
- 收回因syscall长时间阻塞的P；

我们看到sysmon将“向长时间运行的G任务发出抢占调度”，这个事情由retake实施：

```
// forcePreemptNS is the time slice given to a G before it is
// preempted.
const forcePreemptNS = 10 * 1000 * 1000 // 10ms

func retake(now int64) uint32 {
    ... ...
    // 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
        continue
    }
    if pd.schedwhen+forcePreemptNS > now {
        continue
    }
    preemptone(_p_)
    ... ...
}
```

可以看出，如果一个G任务运行10ms，sysmon就会认为其运行时间太久而发出抢占式调度的请求。一旦G的抢占标志位被设为true，那么待这个G下一次调用函数或方法时，runtime便可以将G抢占，并移出运行状态，放入P的local runq中，等待下一次被调度。

3、channel阻塞或network I/O情况下的调度

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

4、system call阻塞情况下的调度

如果G被阻塞在某个system call操作上，那么不光G会阻塞，执行该G的M也会解绑P(实质是被sysmon抢走了)，与G一起进入sleep状态。如果此时有idle的M，则P与其绑定继续执行其他G；如果没有idle M，但仍然有其他G要去执行，那么就会创建一个新M。

当阻塞在syscall上的G完成syscall调用后，G会去尝试获取一个可用的P，如果没有可用的P，那么G会被标记为runnable，之前的那个sleep的M将再次进入sleep。

四、调度器状态的查看方法

Go提供了调度器当前状态的查看方法：使用Go运行时环境变量GODEBUG。

```
$GODEBUG=schedtrace=1000 godoc -http=:6060
SCHED 0ms: gomaxprocs=4 idleprocs=3 threads=3 spinningthreads=0 idlethreads=0 runqueue=0 [0 0 0 0]
SCHED 1001ms: gomaxprocs=4 idleprocs=0 threads=9 spinningthreads=0 idlethreads=3 runqueue=2 [8 14 5 2]
SCHED 2006ms: gomaxprocs=4 idleprocs=0 threads=25 spinningthreads=0 idlethreads=19 runqueue=12 [0 0 4 0]
SCHED 3006ms: gomaxprocs=4 idleprocs=0 threads=26 spinningthreads=0 idlethreads=8 runqueue=2 [0 1 1 0]
SCHED 4010ms: gomaxprocs=4 idleprocs=0 threads=26 spinningthreads=0 idlethreads=20 runqueue=12 [6 3 1 0]
SCHED 5010ms: gomaxprocs=4 idleprocs=0 threads=26 spinningthreads=1 idlethreads=20 runqueue=17 [0 0 0 0]
SCHED 6016ms: gomaxprocs=4 idleprocs=0 threads=26 spinningthreads=0 idlethreads=20 runqueue=1 [3 4 0 10]
... ..
```

GODEBUG这个Go运行时环境变量很是强大，通过给其传入不同的key1=value1,key2=value2... 组合，Go的runtime会输出不同的调试信息，比如在这里我们给GODEBUG传入了”schedtrace=1000”，其含义就是每1000ms，打印输出一次goroutine scheduler的状态，每次一行。每一行各字段含义如下：

以上面例子中最后一行为例：

SCHED 6016ms: gomaxprocs=4 idleprocs=0 threads=26 spinningthreads=0 idlethreads=20 runqueue=1 [3 4 0 10]

SCHED：调试信息输出标志字符串，代表本行是goroutine scheduler的输出；
6016ms：即从程序启动到输出这行日志的时间；
gomaxprocs：P的数量；
idleprocs：处于idle状态的P的数量；通过gomaxprocs和idleprocs的差值，我们就可知道执行go代码的P的数量；
threads：os threads的数量，包含scheduler使用的m数量，加上runtime自用的类似sysmon这样的thread的数量；
spinningthreads：处于自旋状态的os thread数量；
idlethread：处于idle状态的os thread的数量；
runqueue=1： go scheduler全局队列中G的数量；
[3 4 0 10]：分别为4个P的local queue中的G的数量。

我们还可以输出每个goroutine、m和p的详细调度信息，但对于Go user来说，绝大多数时间这是不必要的：

\$ GODEBUG=schedtrace=1000,scheddetail=1 godoc -http=:6060

SCHED 0ms: gomaxprocs=4 idleprocs=3 threads=3 spinningthreads=0 idlethreads=0 runqueue=0 gcwaiting=0 nmidlelocked=0 stopwait=0 sysmonwait=0
P0: status=1 schedtick=0 syscalltick=0 m=0 runqsize=0 gfreecnt=0
P1: status=0 schedtick=0 syscalltick=0 m=-1 runqsize=0 gfreecnt=0
P2: status=0 schedtick=0 syscalltick=0 m=-1 runqsize=0 gfreecnt=0
P3: status=0 schedtick=0 syscalltick=0 m=-1 runqsize=0 gfreecnt=0
M2: p=-1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=1 dying=0 helpgc=0 spinning=false blocked=false lockedg=-1
M1: p=-1 curg=17 mallocing=0 throwing=0 preemptoff= locks=0 dying=0 helpgc=0 spinning=false blocked=false lockedg=17
M0: p=0 curg=1 mallocing=0 throwing=0 preemptoff= locks=1 dying=0 helpgc=0 spinning=false blocked=false lockedg=1
G1: status=8() m=0 lockedm=0
G17: status=3() m=1 lockedm=1

SCHED 1002ms: gomaxprocs=4 idleprocs=0 threads=13 spinningthreads=0 idlethreads=7 runqueue=6 gcwaiting=0 nmidlelocked=0 stopwait=0 sysmonwait=0

P0: status=2 schedtick=2293 syscalltick=18928 m=-1 runqsize=12 gfreecnt=2
P1: status=1 schedtick=2356 syscalltick=19060 m=11 runqsize=11 gfreecnt=0
P2: status=2 schedtick=2482 syscalltick=18316 m=-1 runqsize=37 gfreecnt=1
P3: status=2 schedtick=2816 syscalltick=18907 m=-1 runqsize=2 gfreecnt=4
M12: p=-1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=0 dying=0 helpgc=0 spinning=false blocked=true lockedg=-1
M11: p=1 curg=6160 mallocing=0 throwing=0 preemptoff= locks=2 dying=0 helpgc=0 spinning=false blocked=false lockedg=-1
M10: p=-1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=0 dying=0 helpgc=0 spinning=false blocked=true lockedg=-1
... ..

SCHED 2002ms: gomaxprocs=4 idleprocs=0 threads=23 spinningthreads=0 idlethreads=5 runqueue=4 gcwaiting=0 nmidlelocked=0 stopwait=0 sysmonwait=0
P0: status=0 schedtick=2972 syscalltick=29458 m=-1 runqsize=0 gfreecnt=6
P1: status=2 schedtick=2964 syscalltick=33464 m=-1 runqsize=0 gfreecnt=39
P2: status=1 schedtick=3415 syscalltick=33283 m=18 runqsize=0 gfreecnt=12
P3: status=2 schedtick=3736 syscalltick=33701 m=-1 runqsize=1 gfreecnt=6
M22: p=-1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=0 dying=0 helpgc=0 spinning=false blocked=true lockedg=-1
M21: p=-1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=0 dying=0 helpgc=0 spinning=false blocked=true lockedg=-1
... ..

关于go scheduler调试信息输出的详细信息，可以参考Dmitry Vyukov的大作：《[Debugging performance issues in Go programs](#)》。这也应该是每个gopher必读的经典文章。当然更详尽的代码可参考\$GOROOT/src/runtime/proc.go中的schedtrace函数。

微博：[@tonybai_cn](#)
微信公众号：iamtonybai
github.com: <https://github.com/bigwhite>

© 2017, [bigwhite](#). 版权所有.



Related posts:

- [Go语言TCP Socket编程](#)
- [Goroutine是如何工作的](#)
- [论golang Timer Reset方法使用的正确姿势](#)
- [Go coding in go way](#)
- [Go程序调试、分析与优化](#)

已有 8 条评论

-  [开发者头条](#)
2017/06/26

感谢分享！已推荐到《开发者头条》：<https://toutiao.io/posts/9s0j2w> 欢迎点赞支持！
欢迎订阅《一码平川的独家号》<https://toutiao.io/subjects/164069>

[回复](#)

-  [Pencilcap](#)
2017/10/13

好文！虽然我是操作系统小白，仍然能够对goroutine调度的原理有了大致的认识。果然收藏

[回复](#)



3. *leavesdrift*
2017/11/22

非常，非常感谢你的这篇文章！终极好文，希望能看到更多这样的文章，初学Golang，能看到这样的分析文章我非常幸运！

[回复](#)



◦ *bigwhite*
2017/11/22

过誉了。终极的解释一定是在代码里，在代码里，在代码里^0^。

[回复](#)



4. *blueblue*
2017/12/25

如果G被阻塞在某个system call操作上，那么不光G会阻塞，执行该G的M也会解绑P(实质是被sysmon抢走了)，与G一起进入sleep状态。如果此时有idle的M，则P与其绑定继续执行其他G；如果没有idle M，但仍然有其他G要去执行，那么就会创建一个新M。

当阻塞在syscall上的G完成syscall调用后，G会去尝试获取一个可用的P，如果没有可用的P，那么G会被标记为runnable，之前的那个sleep的M将再次进入sleep。

有个问题，执行系统调用的那个M与P解除关联的时候是进入sleep状态了吗？

[回复](#)



◦ *bigwhite*
2017/12/25

嗯，是block on syscall的状态。这里本想表达的也是这个意思。只是用了一个广义的“sleep”，可能不是那么恰当:(。

[回复](#)



5. *blueblue*
2017/12/25

因为我在一个地方看到，说是：
The os thread processing the goroutine idles waiting for the sys call to return.

When a G completes a system call, it will have to re-acquire the P. If there are no P’s available, it will be marked runnable and the M will go to sleep.

[回复](#)



6. *blueblue*
2017/12/25

The syscalling thread will hold on to the goroutine that made the syscall since it’s technically still executing, albeit blocked in the OS.

[回复](#)

添加新评论

称呼

邮箱

网站

http://example.com

提交评论

输入关键字搜索

搜索

欢迎使用邮件订阅我的博客

输入邮箱订阅本站，只要有新文章发布，就会第一时间发送邮件通知你哦！

名字:

邮箱:

[马上订阅](#)



这里是[Tony Bai](#)的个人Blog，欢迎访问、订阅和留言！[订阅Feed请点击上面图片。](#)

如果您觉得这里的文章对您有帮助，请扫描上方二维码进行捐赠，加油后的Tony Bai将会为您呈现更多精彩的文章，谢谢！

如果您希望通过微信捐赠，请用微信客户端扫描下方赞赏码：



如果您希望通过比特币或以太币捐赠，可以扫描下方二维码：

比特币：



以太坊：



如果您喜欢通过微信App浏览本站内容，可以扫描下方二维码，订阅本站官方微信订阅号“iamtonybai”；点击二维码，可直达本人官方微博主页^_^：



本站Powered by Digital Ocean VPS。

[选择Digital Ocean VPS主机](#)，即可获得10美元现金充值，可免费使用两个月哟！

著名主机提供商Linode 10\$优惠码：linode10，在[这里注册](#)即可免费获得。

阿里云推荐码：**1WFZ0V**，**立享9折**！

bigwhite.cn@Gmail.com



文章


- [TB一周萃选\[第4期\]](#)
- [使用istio治理微服务入门](#)
- [TB一周萃选\[第3期\]](#)
- [TB一周萃选\[第2期\]](#)
- [追求极简：Docker镜像构建演化史](#)
- [TB一周萃选\[第1期\]](#)
- [在Kubernetes集群上部署高可用Harbor镜像仓库](#)
- [Goroutine调度实例简要分析](#)
- [理解Docker的多阶段镜像构建](#)
- [Hello, Termux](#)

评论

- 
tyan 在 [Hello, Termux](#)
先将QQ输入法切换到中文，然后连接蓝牙键盘，可输入中文。
- 
bigwhite 在 [Hello, Termux](#)
termux的初衷就是无需root权限的linux环境模拟。关于root的问题还是参考官方FAQ吧： ...
- 
不见也散 在 [Hello, Termux](#)
如何使用root，输入SU之后，termux能用的命令全都失效了，变成了普通的终端
- 
cyl 在 [理解Docker的多阶段镜像构建](#)
嗯，感谢感谢，我后来也看大家说这种缓存机制会有不少帮助，所以中间none镜像不应该被默认删除。
- 
bigwhite 在 [关于我](#)
欢迎关注。在没找到更好的防止恶意评论的插件之前，我只能用这个了:(。因为我的wordpress版本较...
- 
辛必果 在 [使用Golang开发微信公众平台-接入验证](#)
我不得不吐槽一波微信的加密，折腾死我了
- 
辛必果 在 [智慧城市到底满足的是谁的诉求](#)
这个问题属于世界难题，普通人比如我，属于那种政府设计的不满意，自己又设计不出来的，积水点的问题本来就...
- 
辛必果 在 [关于我](#)
从TensorFlow那篇文章过来的，期待后续DL方面的文章，同时也吐槽一波这个滚动条，如果忘了滑动...
- 

- 辛必果 在 [给女儿搭建一个博客站点](#)

给女儿点赞，好可爱+1


- bigwhite 在 [理解Docker的多阶段镜像构建](#)

这回我终于看清了。原来你说的是 这种中间image啊，我的build也存在，当然不光是多阶段构建，普...

• [下一页](#) »

分类

- [光影汇](#) (7)
- [影音坊](#) (36)
- [思考控](#) (66)
- [技术志](#) (542)
- [教育记](#) (1)
- [杂货铺](#) (75)
- [生活簿](#) (154)
- [职场录](#) (14)
- [读书吧](#) (14)
- [运动迷](#) (107)
- [驴友秀](#) (40)

标签

[Blog](#) [Blogger](#) [C](#) [Cpp](#) [docker](#) [English](#) [GCC](#) [github](#) [GNU](#) [Go](#) [Golang](#) [Google](#) [Java](#) [k8s](#) [Kernel](#) [Kubernetes](#) [Linux](#) [M10](#) [Opensource](#) [Programmer](#) [Python](#) [Solaris](#) [Subversion](#)

[Ubuntu](#) [Unix](#) [Windows](#) [世界杯](#) [博客](#) [学习](#) [容器](#) [工作](#) [巴萨](#) [开源](#) [思考](#) [感悟](#) [摄影](#) [旅游](#) [梅西](#) [球王](#) [生活](#) [程序员](#) [编译器](#) [西里](#) [足球](#) [驴友](#)

归档

- [2018 年一月](#) (2)
- [2017 年十二月](#) (5)
- [2017 年十一月](#) (4)
- [2017 年十月](#) (3)
- [2017 年九月](#) (2)
- [2017 年八月](#) (3)
- [2017 年七月](#) (4)
- [2017 年六月](#) (8)
- [2017 年五月](#) (5)
- [2017 年四月](#) (3)
- [2017 年三月](#) (2)
- [2017 年二月](#) (5)
- [2017 年一月](#) (7)
- [2016 年十二月](#) (7)
- [2016 年十一月](#) (7)
- [2016 年十月](#) (3)
- [2016 年九月](#) (2)
- [2016 年八月](#) (1)
- [2016 年六月](#) (2)
- [2016 年五月](#) (2)
- [2016 年四月](#) (2)
- [2016 年三月](#) (2)
- [2016 年二月](#) (3)
- [2016 年一月](#) (2)
- [2015 年十二月](#) (1)
- [2015 年十一月](#) (1)
- [2015 年十月](#) (1)
- [2015 年九月](#) (3)
- [2015 年八月](#) (5)
- [2015 年七月](#) (6)
- [2015 年六月](#) (4)
- [2015 年五月](#) (1)
- [2015 年四月](#) (2)
- [2015 年三月](#) (2)
- [2015 年一月](#) (2)
- [2014 年十二月](#) (5)
- [2014 年十一月](#) (8)
- [2014 年十月](#) (9)
- [2014 年九月](#) (2)
- [2014 年八月](#) (1)
- [2014 年七月](#) (1)

- [2014 年五月](#) (2)
- [2014 年四月](#) (5)
- [2014 年三月](#) (4)
- [2014 年二月](#) (1)
- [2014 年一月](#) (1)
- [2013 年十二月](#) (3)
- [2013 年十一月](#) (5)
- [2013 年十月](#) (6)
- [2013 年九月](#) (4)
- [2013 年八月](#) (5)
- [2013 年七月](#) (6)
- [2013 年六月](#) (2)
- [2013 年五月](#) (6)
- [2013 年四月](#) (3)
- [2013 年三月](#) (7)
- [2013 年二月](#) (4)
- [2013 年一月](#) (6)
- [2012 年十二月](#) (8)
- [2012 年十一月](#) (10)
- [2012 年十月](#) (5)
- [2012 年九月](#) (3)
- [2012 年八月](#) (10)
- [2012 年七月](#) (4)
- [2012 年六月](#) (2)
- [2012 年五月](#) (4)
- [2012 年四月](#) (10)
- [2012 年三月](#) (8)
- [2012 年二月](#) (6)
- [2012 年一月](#) (6)
- [2011 年十二月](#) (4)
- [2011 年十一月](#) (4)
- [2011 年十月](#) (5)
- [2011 年九月](#) (8)
- [2011 年八月](#) (7)
- [2011 年七月](#) (6)
- [2011 年六月](#) (7)
- [2011 年五月](#) (8)
- [2011 年四月](#) (6)
- [2011 年三月](#) (10)
- [2011 年二月](#) (7)
- [2011 年一月](#) (10)
- [2010 年十二月](#) (7)
- [2010 年十一月](#) (6)
- [2010 年十月](#) (7)
- [2010 年九月](#) (12)
- [2010 年八月](#) (8)
- [2010 年七月](#) (3)
- [2010 年六月](#) (5)
- [2010 年五月](#) (4)
- [2010 年四月](#) (2)
- [2010 年三月](#) (6)
- [2010 年二月](#) (4)
- [2010 年一月](#) (6)
- [2009 年十二月](#) (6)
- [2009 年十一月](#) (6)
- [2009 年十月](#) (5)
- [2009 年九月](#) (8)
- [2009 年八月](#) (8)
- [2009 年七月](#) (8)
- [2009 年六月](#) (2)
- [2009 年五月](#) (5)
- [2009 年四月](#) (7)
- [2009 年三月](#) (12)
- [2009 年二月](#) (9)
- [2009 年一月](#) (15)
- [2008 年十二月](#) (9)
- [2008 年十一月](#) (5)
- [2008 年十月](#) (10)
- [2008 年九月](#) (13)
- [2008 年八月](#) (13)
- [2008 年七月](#) (3)

- [2008 年六月](#) (1)
- [2008 年五月](#) (7)
- [2008 年四月](#) (4)
- [2008 年三月](#) (9)
- [2008 年二月](#) (11)
- [2008 年一月](#) (15)
- [2007 年十二月](#) (11)
- [2007 年十一月](#) (14)
- [2007 年十月](#) (4)
- [2007 年九月](#) (5)
- [2007 年八月](#) (1)
- [2007 年七月](#) (10)
- [2007 年六月](#) (10)
- [2007 年五月](#) (10)
- [2007 年四月](#) (8)
- [2007 年三月](#) (15)
- [2007 年二月](#) (4)
- [2007 年一月](#) (17)
- [2006 年十二月](#) (18)
- [2006 年十一月](#) (9)
- [2006 年十月](#) (11)
- [2006 年九月](#) (6)
- [2006 年八月](#) (5)
- [2006 年七月](#) (22)
- [2006 年六月](#) (35)
- [2006 年五月](#) (24)
- [2006 年四月](#) (26)
- [2006 年三月](#) (25)
- [2006 年二月](#) (18)
- [2006 年一月](#) (15)
- [2005 年十二月](#) (10)
- [2005 年十一月](#) (10)
- [2005 年九月](#) (13)
- [2005 年八月](#) (11)
- [2005 年七月](#) (6)
- [2005 年六月](#) (2)
- [2005 年五月](#) (3)
- [2005 年四月](#) (6)
- [2005 年三月](#) (1)
- [2005 年一月](#) (15)
- [2004 年十二月](#) (9)
- [2004 年十一月](#) (14)
- [2004 年十月](#) (2)
- [2004 年九月](#) (2)

私人

- [我的女儿](#)

链接

- [@douban](#)
- [@flickr](#)
- [@github](#)
- [@googlecode](#)
- [@picasa](#)
- [@slideshare](#)
- [@twitter](#)
- [@weibo](#)
- [Hoterran](#)
- [Lionel Messi](#)
- [Puras He](#)
- [梦想风暴](#)
- [过眼云烟](#)

开源项目


- [buildc](#)
- [cbehave](#)
- [lcut](#)

翻译项目

- [C语言编码风格和标准](#)



微博



tonybai_cn

辽宁 沈阳

+ 加关注

求雪成功

沈阳晚报

V

#沈阳下雪了#

这次终于等到你了，😏沈阳迎来了2018年第一场“认真”的雪❄️



今天 06:53

转发 | 评论

今天 07:00

转发 | 评论

转发微博

LaLiga西甲联赛 **V**：527👁️ Messi 232👁️ César 194👁️ Kubala 184👁️ Samitier 167👁️ Escolà 143👁️ Alcántara 132👁️ SUÁREZ 131👁️ Eto'o 苏亚雷斯已经超过埃托奥，成为@巴塞罗那足球俱乐部 队史第七射手！#激情灵魂，大爱西甲#

[我的豆瓣主页](#)



[我的豆瓣主页](#)



01465323 [View My Stats](#)



更多

0

© 2018 [Tony Bai](#). 由 [Wordpress](#) 强力驱动. 模板由[cho](#)制作.