



下载APP



32 | 并发：聊聊Goroutine调度器的原理

2022-01-10 Tony Bai

《Tony Bai · Go语言第一课》

课程介绍 >

**讲述：Tony Bai**

时长 17:11 大小 15.75M



你好，我是 Tony Bai。

上一讲我们学习了并发的基本概念和 Go 的并发方案，也就是 Goroutine 的一些基本使用和注意事项。对于大多数 Gopher 来说，这些内容作为 Go 并发入门已经是足够了。

但毕竟 Go 没有采用基于线程的并发模型，可能很多 Gopher 都好奇 Go 运行时究竟如何将一个个 Goroutine 调度到 CPU 上执行的。当然，Goroutine 的调度本来是 Go 语言核心开发团队才应该关注的事情，大多数 Gopher 们无需关心。但就我个人的学习和实践经验而言，我觉得了解 Goroutine 的调度模型和原理，能够帮助我们编写出更高质量的 Go 代码。

领资料



因此，在这一讲中，我想和你一起简单探究一下 Goroutine 调度器的原理和演化历史。

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。

前面我们也提到，传统的编程语言，比如 C、C++ 等的并发实现，多是基于线程模型的，也就是应用程序负责创建线程（一般通过 libpthread 等库函数调用实现），操作系统负责调度线程。当然，我们也说过，这种传统支持并发的方式有很多不足。为了解决这些问题，Go 语言中的并发实现，使用了 Goroutine，代替了操作系统的线程，也不再依靠操作系统调度。

Goroutine 占用的资源非常小，上节课我们也说过，每个 Goroutine 栈的默认大小默认是 2KB。而且，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。而将这些 Goroutine 按照一定算法放到“CPU”上执行的程序，就被称为 **Goroutine 调度器（Goroutine Scheduler）**，注意，这里说的“CPU”打了引号。

不过，一个 Go 程序对于操作系统来说只是一个**用户层程序**，操作系统眼中只有线程，它甚至不知道有一种叫 **Goroutine** 的事物存在。所以，Goroutine 的调度全要靠 Go 自己完成。那么，实现 Go 程序内 Goroutine 之间“公平”竞争“CPU”资源的任务，就落到了 Go 运行时（runtime）头上了。要知道在一个 Go 程序中，除了用户层代码，剩下的就是 Go 运行时了。

于是，Goroutine 的调度问题就演变为，Go 运行时如何将程序内的众多 Goroutine，按照一定算法调度到“CPU”资源上运行的问题了。

可是，在操作系统层面，线程竞争的“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”资源又是什么呢？

Go 程序是用户层程序，它本身就是整体运行在一个或多个操作系统线程上的。所以这个答案就出来了：Goroutine 们要竞争的“CPU”资源就是操作系统线程。这样，Goroutine 调度器的任务也就明确了：**将 Goroutine 按照一定算法放到不同的操作系统线程中去执行。**

那么，Goroutine 调度器究竟是以怎样的算法模型，将 Goroutine 调度到不同的操作系统线程上去的呢？我们继续向下看。

Goroutine 调度器模型与演化过程

Goroutine 调度器的实现不是一蹴而就的，它的调度模型与算法也是几经演化，从最初的 G-M 模型、到 G-P-M 模型，从不支持抢占，到支持协作式抢占，再到支持基于信号的异步抢占，Goroutine 调度器经历了不断地优化与打磨。

首先我们来看最初的 **G-M 模型**。

2012 年 3 月 28 日，Go 1.0 正式发布。在这个版本中，Go 开发团队实现了一个简单的 Goroutine 调度器。在这个调度器中，每个 Goroutine 对应于运行时中的一个抽象结构：`G(``G``oroutine)`，

而被视作“物理 CPU”的操作系统线程，则被抽象为另外一个结构：`M(machine)`。

调度器的工作就是将 G 调度到 M 上去运行。为了更好地控制程序中活跃的 M 的数量，调度器引入了 `GOMAXPROCS` 变量来表示 Go 调度器可见的“处理器”的最大数量。

这个模型实现起来比较简单，也能正常工作，但是却存在着诸多问题。前英特尔黑带级工程师、现谷歌工程师 [德米特里 - 维尤科夫 \(Dmitry Vyukov\)](#) 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。

这个问题主要体现在这几个方面：

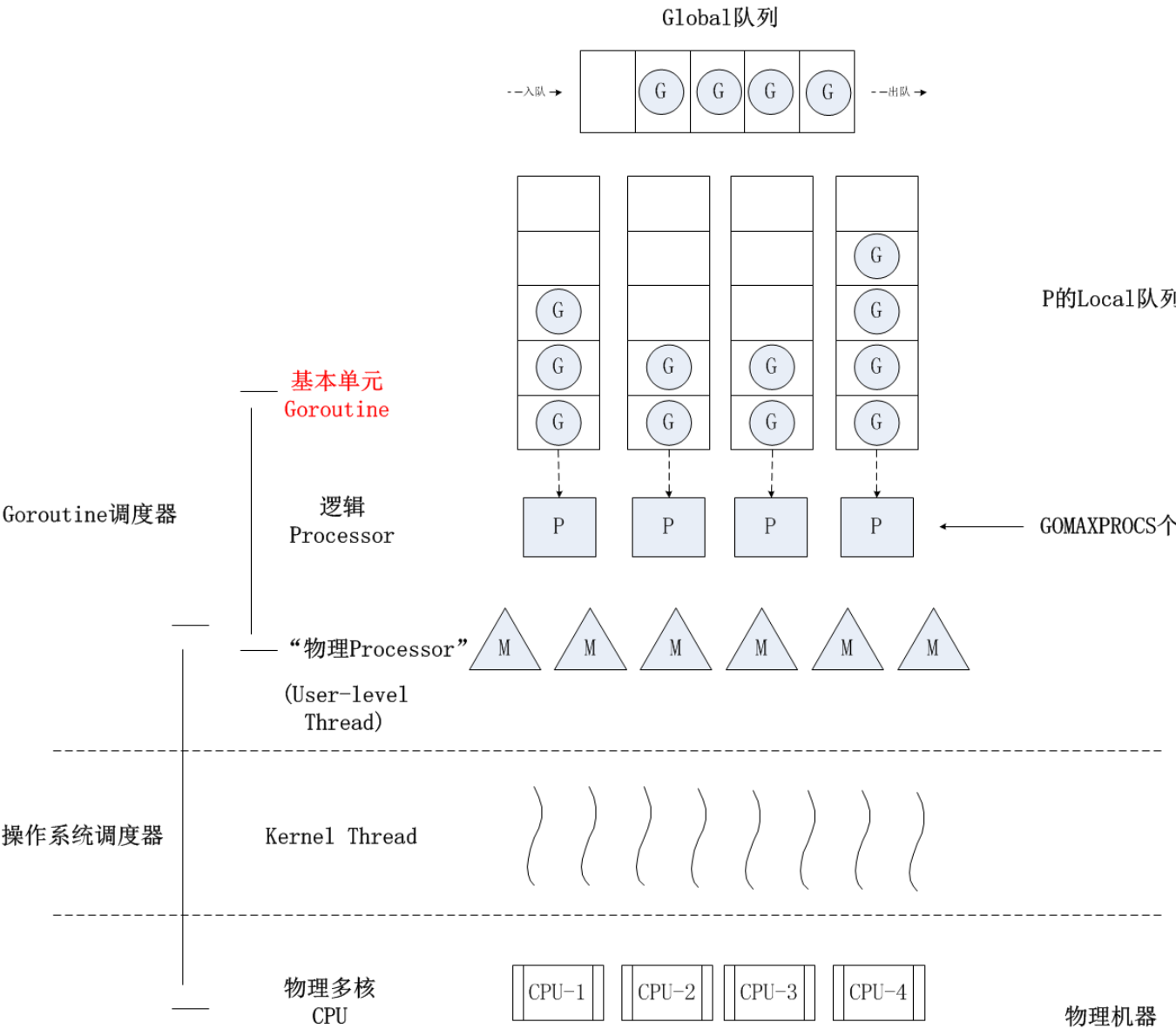
单一全局互斥锁(`Sched.Lock`)和集中状态存储的存在，导致所有 Goroutine 相关操作，比如创建、重新调度等，都要上锁；

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，也增加了额外的性能损耗；

每个 M 都做内存缓存，导致内存占用过高，数据局部性较差；

由于系统调用(`syscall`)而形成的频繁的工作线程阻塞和解除阻塞，导致额外的性能损耗。

为了解决这些问题，德米特里 - 维尤科夫又亲自操刀改进了 Go 调度器，在 Go 1.1 版本中实现了 **G-P-M 调度模型**和 **work stealing 算法**，这个模型一直沿用至今。模型如下图所示：



Goroutine调度原理图

有人说过：“**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**”，德米特里 - 维尤科夫的 **G-P-M** 模型恰是这一理论的践行者。你可以看到，德米特里 - 维尤科夫通过向 G-M 模型中增加了一个 P，让 Go 调度器具有很好的伸缩性。

P 是一个“逻辑 Proccessor”，每个 G (Goroutine) 要想真正运行起来，首先需要被分配一个 P，也就是进入到 P 的本地运行队列 (local runq) 中。对于 G 来说，P 就是运行它的“CPU”，可以说：**在 G 的眼里只有 P**。但从 Go 调度器的视角来看，真正的“CPU”是 M，只有将 P 和 M 绑定，才能让 P 的 runq 中的 G 真正运行起来。

G-P-M 模型的实现算是Go调度器的一大进步，但调度器仍然有一个令人头疼的问题，那就是**不支持抢占式调度**，这导致一旦某个 G 中出现死循环的代码逻辑，那么 G 将永久占用分配给它的 P 和 M，而位于同一个 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 (GOMAXPROCS=1) 时，整个 Go 程序中的其他 G 都将“饿死”。于是德米特里 - 维尤科夫又提出了《[Go Preemptive Scheduler Design](#)》并在 **Go 1.2** 中实现了基于协作的“**抢占式**”调度。

这个抢占式调度的原理就是，Go 编译器在每个函数或方法的入口处加上了一段额外的代码 (runtime.morestack_noctxt)，让运行时有机会在这段代码中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，只有在有函数调用的地方才能插入“抢占”代码（埋点），对于没有函数调用而是纯算法循环计算的 G，Go 调度器依然无法抢占。

比如，死循环等并没有给编译器插入抢占代码的机会，这就会导致 GC 在等待所有 Goroutine 停止时的等待时间过长，从而[导致 GC 延迟](#)，内存占用瞬间冲高；甚至在一些特殊情况下，导致在 STW (stop the world) 时死锁。

为了解决这些问题，Go 在 1.14 版本中接受了奥斯汀 - 克莱门茨 (Austin Clements) 的[提案](#)，增加了**对非协作的抢占式调度的支持**，这种抢占式调度是基于系统信号的，也就是通过向线程发送信号的方式来抢占正在运行的 Goroutine。

除了这些大的迭代外，Goroutine 的调度器还有一些小的优化改动，比如**通过文件 I/O poller 减少 M 的阻塞等**。

Go 运行时已经实现了 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），也就不会导致大量线程（M）被创建出来。

但是对于文件 I/O 操作来说，一旦阻塞，那么线程（M）将进入挂起状态，等待 I/O 返回后被唤醒。这种情况下 P 将与挂起的 M 分离，再选择一个处于空闲状态（idle）的 M。如果此时没有空闲的 M，就会新创建一个 M（线程），所以，这种情况下，大量 I/O 操作仍然会导致大量线程被创建。

为了解决这个问题，Go 开发团队的伊恩 - 兰斯 - 泰勒 (Ian Lance Taylor) 在 Go 1.9 中增加了一个 [针对文件 I/O 的 Poller](#) 的功能，这个功能可以像 netpoller 那样，在 G 操作那些支持监听 (pollable) 的文件描述符时，仅会阻塞 G，而不会阻塞 M。不过这个功能依然不能对常规文件有效，常规文件是不支持监听的 (pollable)。但对于 Go 调度器而言，这也算是一个不小的进步了。

从 Go 1.2 以后，Go 调度器就一直稳定在 G-P-M 调度模型上，尽管有各种优化和改进，但也都是基于这个模型之上的。那未来的 Go 调度器会往哪方面发展呢？德米特里 - 维尤科夫在 2014 年 9 月提出了一个新的设计草案文档：《[NUMA-aware scheduler for Go](#)》，作为对未来 Goroutine 调度器演进方向的一个提议，不过至今似乎这个提议也没有列入开发计划。

通过前面对 Goroutine 调度器演化的分析，你可以看到，目前 G-M 模型已经废弃，NUMA 调度模型尚未实现，那么现在我们要理解如今的 Goroutine 调度，只需要学习 G-P-M 模型就可以了，接下来我们就来看看 G-P-M 模型下 Goroutine 的调度原理。

深入 G-P-M 模型

Go 语言中 Goroutine 的调度、GC、内存管理等是 Go 语言原理最复杂、最难懂的地方，随便拿出一个都可以讲上几节课，并且这三方面的内容随着 Go 版本的演进也在不断更新。因为我们是入门课，所以这里我就只基于 Go 1.12.7 版本（支持基于协作的抢占式调度）给你粗略介绍一下基于 G-P-M 模型的调度原理，如果你还对这方面感兴趣，可以基于这些介绍深入到相关的 Go 源码中去，深入挖掘细节。

G、P 和 M

关于 G、P、M 的定义，我们可以参见 `$GOROOT/src/runtime/runtime2.go` 这个源文件。你可以看到，G、P、M 这三个结构体定义都是大块头，每个结构体定义都包含十几个甚至二三十个字段。更不用说，像调度器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 的各自的大致用途，我们这里简要说明一下：

G: 代表 Goroutine，存储了 Goroutine 的执行栈信息、Goroutine 状态以及 Goroutine 的任务函数等，而且 G 对象是可以重用的；

P: 代表逻辑 processor , P 的数量决定了系统内最大可并行的 G 的数量 , P 的最大作用还是其拥有的各种 G 对象队列、链表、一些缓存和状态 ;

M: M 代表着真正的执行计算资源。在绑定有效的 P 后 , 进入一个调度循环 , 而调度循环的机制大致是从 P 的本地运行队列以及全局队列中获取 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。

我这里也给出了 G、P、M 定义的代码片段 (注意 : 我们这里使用的是 Go 1.12.7 版本 , 随着 Go 演化 , 结构体中的字段定义可能会有不同) , 你也可以看一看 :

[复制代码](#)

```
1 //src/runtime/runtime2.go
2 type g struct {
3     stack    stack    // offset known to runtime/cgo
4     sched    gobuf
5     goid      int64
6     gopc      uintptr // pc of go statement that created this goroutine
7     startpc   uintptr // pc of goroutine function
8     ... ..
9 }
10
11 type p struct {
12     lock mutex
13
14     id        int32
15     status    uint32 // one of pidle/prunning/...
16
17     mcache    *mcache
18     racectx   uintptr
19
20     // Queue of runnable goroutines. Accessed without lock.
21     runqhead  uint32
22     runqtail  uint32
23     runq      [256]uintptr
24
25     runnext   uintptr
26
27     // Available G's (status == Gdead)
28     gfree     *g
29     gfreecnt  int32
30
31     ... ..
32 }
33
34
35
```

```
36 type m struct {
37     g0          *g    // goroutine with scheduling stack
38     mstartfn    func()
39     curg        *g    // current running goroutine
40     ... ..
41 }
```

而 Goroutine 调度器的目标，就是公平合理地将各个 G 调度到 P 上“运行”，下面我们重点看看 G 是如何被调度的。

G 被抢占调度

我们先来说常规情况，也就是如果某个 G 没有进行系统调用（syscall）、没有进行 I/O 操作、没有阻塞在一个 channel 操作上，**调度器是如何让 G 停下来并调度下一个可运行的 G 的呢？**

答案就是：**G 是被抢占调度的。**

前面说过，除非极端的无限循环，否则只要 G 调用函数，Go 运行时就有了抢占 G 的机会。Go 程序启动时，运行时会去启动一个名为 sysmon 的 M（一般称为监控线程），这个 M 的特殊之处在于它不需要绑定 P 就可以运行（以 g0 这个 G 的形式），这个 M 在整个 Go 程序的运行过程中至关重要，你可以看下我对 sysmon 被创建的部分代码以及 sysmon 的执行逻辑摘录：

```
1  //$GOROOT/src/runtime/proc.go
2
3  // The main goroutine.
4  func main() {
5      ... ..
6      systemstack(func() {
7          newm(sysmon, nil)
8      })
9      .... ..
10 }
11
12 // Always runs without a P, so write barriers are not allowed.
13 //
14 //go:nowritebarrierrec
15 func sysmon() {
16     // If a heap span goes unused for 5 minutes after a garbage collection,
17     // we hand it back to the operating system.
```

[复制代码](#)


```


18     scavengelimit := int64(5 * 60 * 1e9)
19     ... ..
20
21     if .... {
22         ... ..
23         // retake P's blocked in syscalls
24         // and preempt long running G's
25         if retake(now) != 0 {
26             idle = 0
27         } else {
28             idle++
29         }
30         ... ..
31     }
32 }

```

我们看到，sysmon 每 20us~10ms 启动一次，sysmon 主要完成了这些工作：

- 释放闲置超过 5 分钟的 span 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”，这个事情由函数retake 实施：

 复制代码

```

1 // $GOROOT/src/runtime/proc.go
2
3 // forcePreemptNS is the time slice given to a G before it is
4 // preempted.
5 const forcePreemptNS = 10 * 1000 * 1000 // 10ms
6
7 func retake(now int64) uint32 {
8     ... ..
9     // Preempt G if it's running for too long.
10    t := int64(_p_.schedtick)
11    if int64(pd.schedtick) != t {
12        pd.schedtick = uint32(t)
13        pd.schedwhen = now
14        continue

```

```
15         }
16         if pd.schedwhen+forcePreemptNS > now {
17             continue
18         }
19         preemptone(_p_)
20         ... ..
21     }
22
23     func preemptone(_p_ *p) bool {
24         mp := _p_.m.ptr()
25         if mp == nil || mp == getg().m {
26             return false
27         }
28         gp := mp.curg
29         if gp == nil || gp == mp.g0 {
30             return false
31         }
32
33         gp.preempt = true //设置被抢占标志
34
35         // Every call in a go routine checks for stack overflow by
36         // comparing the current stack pointer to gp->stackguard0.
37         // Setting gp->stackguard0 to StackPreempt folds
38         // preemption into the normal stack overflow check.
39         gp.stackguard0 = stackPreempt
40         return true
41     }
```

从上面的代码中，我们可以看出，**如果一个 G 任务运行 10ms，sysmon 就会认为它的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，那么等到这个 G 下一次调用函数或方法时，运行时就可以将 G 抢占并移出运行状态，放入队列中，等待下一次被调度。

不过，除了这个常规调度之外，还有两个特殊情况下 G 的调度方法。

第一种：channel 阻塞或网络 I/O 情况下的调度。

如果 G 被阻塞在某个 channel 操作或网络 I/O 操作上时，G 会被放置到某个等待（wait）队列中，而 M 会尝试运行 P 的下一个可运行的 G。如果这个时候 P 没有可运行的 G 供 M 运行，那么 M 将解绑 P，并进入挂起状态。当 I/O 操作完成或 channel 操作完成，在等待队列中的 G 会被唤醒，标记为可运行（runnable），并被放入到某 P 的队列中，绑定一个 M 后继续执行。

第二种：系统调用阻塞情况下的调度。

如果 G 被阻塞在某个系统调用（system call）上，那么不光 G 会阻塞，执行这个 G 的 M 也会解绑 P，与 G 一起进入挂起状态。如果此时有空闲的 M，那么 P 就会和它绑定，并继续执行其他 G；如果没有空闲的 M，但仍然有其他 G 要去执行，那么 Go 运行时就会创建一个新 M（线程）。

当系统调用返回后，阻塞在这个系统调用上的 G 会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，之前的那个挂起的 M 将再次进入挂起状态。

小结

好了，今天的课讲到这里就结束了，现在我们一起回顾一下吧。

基于 Goroutine 的并发设计离不开一个高效的生产级调度器。Goroutine 调度器演进了 10 余年，先后经历了 G-M 模型、G-P-M 模型和 work stealing 算法、协作式的抢占调度以及基于信号的异步抢占等改进与优化，目前 Goroutine 调度器相对稳定和成熟，可以适合绝大部分生产场合。

现在的 G-P-M 模型和最初的 G-M 模型相比，通过向 G-M 模型中增加了一个代表逻辑处理器的 P，使得 Goroutine 调度器具有了更好的伸缩性。

M 是 Go 代码运行的真实载体，包括 Goroutine 调度器自身的逻辑也是在 M 中运行的。

P 在 G-P-M 模型中占据核心地位，它拥有待调度的 G 的队列，同时 M 要想运行 G 必须绑定一个 P。一个 G 被调度执行的时间不能过长，超过特定长的时间后，G 会被设置为**可抢占**，并在下一次执行函数或方法时被 Go 运行时移出运行状态。

如果 G 被阻塞在某个 channel 操作或网络 I/O 操作上时，M 可以不被阻塞，这避免了大量创建 M 导致的开销。但如果 G 因慢系统调用而阻塞，那么 M 也会一起阻塞，但在阻塞前会与 P 解绑，P 会尝试与其他 M 绑定继续运行其他 G。但若没有现成的 M，Go 运行时建立新的 M，这也是系统调用可能导致系统线程数量增加的原因，你一定要注意这一点。

思考题

为了让你更好理解 Goroutine 调度原理，我这里留个思考题。请看下面代码：

[复制代码](#)

```
1 func deadloop() {  
2     for {  
3     }  
4 }  
5  
6 func main() {  
7     go deadloop()  
8     for {  
9         time.Sleep(time.Second * 1)  
10        fmt.Println("I got scheduled!")  
11    }  
12 }
```

我的问题是：

1. 在一个拥有多核处理器的主机上，使用 Go 1.13.x 版本运行这个示例代码，你在命令行终端上是否能看到 “I got scheduled!” 输出呢？也就是 main goroutine 在创建 deadloop goroutine 之后是否能继续得到调度呢？
2. 我们通过什么方法可以让上面示例中的 main goroutine，在创建 deadloop goroutine 之后无法继续得到调度？

欢迎你把这节课分享给更多对 Goroutine 调度原理感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta订阅后你可得 **20** 元现金奖励

 生成海报并分享

 赞 3  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

[上一篇](#) 31 | 并发：Go的并发方案实现方案是怎样的？

更多课程推荐

陈天 · Rust 编程第一课

实战驱动，快速上手 Rust

陈天

Tubi TV 研发副总裁



涨价倒计时 

今日订阅 **¥89**，1月12日涨价至**¥199**

精选留言 (2)

 写留言



一步 

2022-01-10

思考题：

是不是不管怎么处理，main goroutine 都会被调度？



bearlu

2022-01-10

老师，想学习goroutine调度器，演进的关键版本，依次是go的什么发行版？还有什么相关资料书籍？谢谢老师

