跟大家汇报一下，这段时间我在看 proc.go
的源码，其实就是调度器的源码。代码有几千行之多，不像以往的 map，channel 等等。想把这些代码都看明白，是一个庞大的工程。到今天为止，我也不敢说我都看明白了。

要深挖下去的话，会无穷无尽，所以阶段性的探索就到这里。接下来就把这段时间的探索分享出来。

其实，今天这篇文章仅仅算是一个引子，接下来会连续发布十篇系列文章。目录如下：

开始我们今天的正题。

一个月前，《Go 语言高级编程》作者柴树杉老师在 CSDN 上发表了一篇《Go 语言十年而立，Go2 蓄势待发》，视角十分宏大。我们既要低头看路，有时也要抬头看天，这篇文章就属于“抬头”看天类的，推荐阅读。

文章中提到了第一本写 Go 的小说《胡文 Go》。我找来看了下，嬉笑怒骂，还挺有意思的。书中有这样一句话：

> 在 Go 语言里，go func 是并发的单元，chan 是协调并发单元的机制，panic 和 recover 是出错处理的机制，而 defer

是神来之笔，大大简化了出错的管理。

Goroutines 在同一个用户空间里同时独立执行 functions，channels 则用于 goroutines 间的通信和同步访问控制。

上一篇文章里我们讲了 channel，并且提到，goroutine 和 channel 是 Go 并发编程的两大基石，那这篇文章就聚焦到 goroutine，以及调度 goroutine 的 go scheduler。

按照惯例，手动贴上本文的目录：

状态	解释
Waiting	等待状态。线程在等待某件事的发生。例如等待网络数据、硬盘；调用操作系统 API；等待内存同步访问条件 ready，如 atomic, mutexes
Runnable	就绪状态。只要给 CPU 资源我就能运行
Executing	运行状态。线程在执行指令，这是我们想要的

头条 @Go语言中文网

线程能做的事一般分为两种：计算型、IO 型。

计算型主要是占用 CPU 资源，一直在做计算任务，例如对一个大数做质数分解。这种类型的任务不会让线程跳到 Waiting 状态。

IO 型则是要获取外界资源，例如通过网络、系统调用等方式。内存同步访问控制原语：mutexes 也可以看作这种类型。共同特点是需要等待外界资源就绪。IO 型的任务会让线程跳到 Waiting 状态。

线程切换就是操作系统用一个处于 Runnable 的线程将 CPU 上正在运行的处于 Executing 状态的线程换下来的过程。新上场的线程会变成 Executing 状态，而下场的线程则可能变成 Waiting 或 Runnable 状态。正在做计算型任务的线程，会变成 Runnable 状态；正在做 IO 型任务的线程，则会变成 Waiting 状态。

因此，计算密集型任务和 IO 密集型任务对线程切换的“态度”是不一样的。由于

计算型密集型任务一直都有任务要做，或者说它一直有指令要执行，线程切换的过程会让它停掉当前的任务，损失非常大。

相反，专注于 IO 密集型的任务的线程，如果它因为某个操作而跳到 Waiting 状态，那么把它从 CPU
上换下，对它而言是没有影响的。而且，新换上来的线程可以继续利用 CPU 完成任务。从整个操作系统来看，“工作进度”是往前的。

记住，对于 OS scheduler 来说，最重要的是不要让一个 CPU
核心闲着，尽量让每个 CPU 核心都有任务可做。

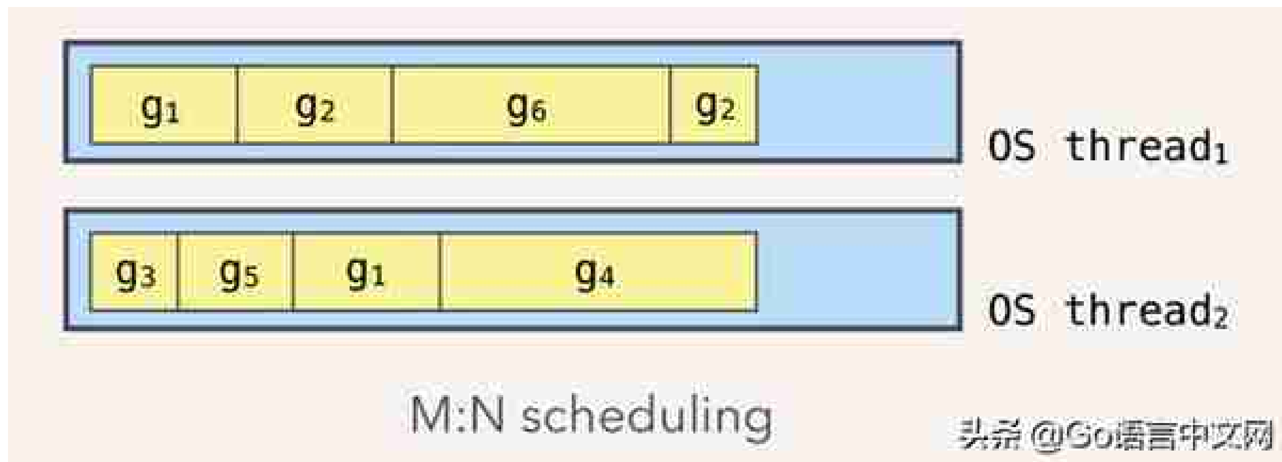
> If you have a program that is focused on IO-Bound work, then context switches are going to be an advantage. Once a Thread moves into a Waiting state, another Thread in a Runnable state is there to take its place. This allows the core to always be doing work. This is one of the most important aspects of scheduling. Don’t allow a core to go idle if there is work (Threads in a Runnable state) to be done.

函数调用过程分析

要想理解 Go scheduler 的底层原理，对于函数调用过程的理解是必不可少的。它涉及到函数参数的传递，CPU 的指令跳转，函数返回值的传递等等。这需要对汇编语言有一定的了解，因为只有汇编语言才能进行像寄存器赋值这样的底层操作。之前的一些文章里也有说明，这里再来复习一遍。

> 函数栈帧的空间主要由函数参数和返回值、局部变量和被调用其它函数的参数和返回值空间组成。

宏观看一下，Go 语言中函数调用的规范，引用曹大博客里的一张图：



在同一时刻，一个线程上只能跑一个 goroutine。当 goroutine
发生阻塞（例如上篇文章提到的向一个 channel 发送数据，被阻塞）时，runtime
会把当前 goroutine 调度走，让其他 goroutine
来执行。目的就是不让一个线程闲着，榨干 CPU 的每一滴油水。

什么是 scheduler

Go 程序的执行由两层组成：Go Program，Runtime，即用户程序和运行时。它们之间通过函数调用来实现内存管理、channel 通信、goroutines
创建等功能。用户程序进行的系统调用都会被 Runtime
拦截，以此来帮助它进行调度以及垃圾回收相关的工作。

一个展现了全景式的关系如下图：

#schedgoals

for scheduling **goroutines** onto **kernel threads**.

☑ use a **small number of kernel threads.**
ideas: reuse threads & limit the number of goroutine-running threads.

☑ support high **concurrency.**
ideas: threads use independent runqueues & keep them balanced.

☑ leverage **parallelism i.e. scale to N cores.**
ideas: use a runqueue per core & employ thread sp头条@Go语言中文网

Go scheduler 的核心思想是：

1. reuse threads；
2. 限制同时运行（不包含阻塞）的线程数为 N，N 等于 CPU 的核心数目；
3. 线程私有的 runqueues，并且可以从其他线程 stealing goroutine 来运行，线程阻塞后，可以将 runqueues 传递给其他线程。

为什么需要 P 这个组件，直接把 runqueues 放到 M 不行吗？

> You might wonder now, why have contexts at all? Can't we just put the runqueues on the threads and get rid of contexts? Not really. The reason we have contexts is so that we can hand them off to other threads if the running thread needs to block for some reason.
>
> An example of when we need to block, is when we call into a syscall. Since a thread cannot both be executing code and be blocked on a syscall, we need to hand off the context so it can keep scheduling.

翻译一下，当一个线程阻塞的时候，将和它绑定的 P 上的 goroutines 转移到其他线程。

Go scheduler 会启动一个后台线程 sysmon，用来检测长时间（超过 10

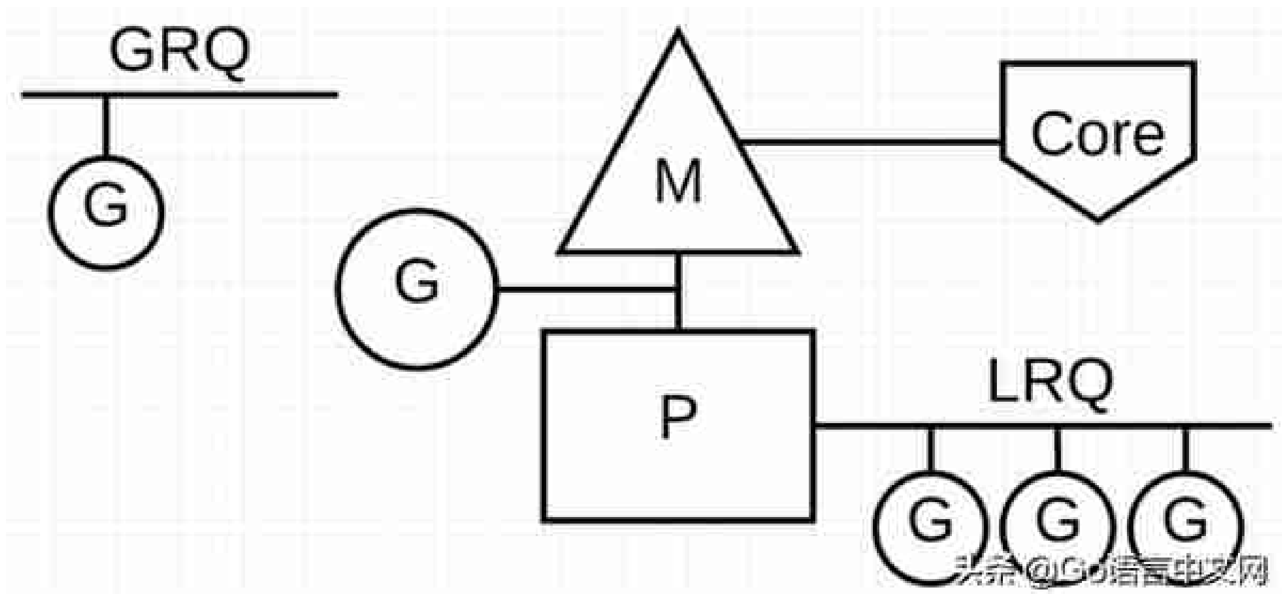
ms）运行的 goroutine，将其调度到 global runqueues。这是一个全局的 runqueue，优先级比较低，以示惩罚。

MacBook Pro

硬件概览：

型号名称：	MacBook Pro
型号标识符：	MacBookPro12,1
处理器名称：	Intel Core i5
处理器速度：	2.7 GHz
处理器数目：	1
核总数：	2
L2 缓存（每个核）：	256 KB
L3 缓存：	3 MB
内存：	8 GB
Boot ROM 版本：	MBP121.0177.B00
SMC 版本（系统）：	2.28f7
序列号（系统）：	C02R4Q1JFVH3
硬件 UUID：	E093D02F-4857-56FC-A1B[illegible]

但是配上 CPU 的超线程，1 个核可以变成 2 个，所以当我在 mac 上运行下面的程序时，会打印出 4。



Go scheduler 是 Go runtime 的一部分，它内嵌在 Go 程序里，和 Go 程序一起运行。因此它运行在用户空间，在 kernel 的上一层。和 Os scheduler 抢占式调度（preemptive）不一样，Go scheduler 采用协作式调度（cooperating）。

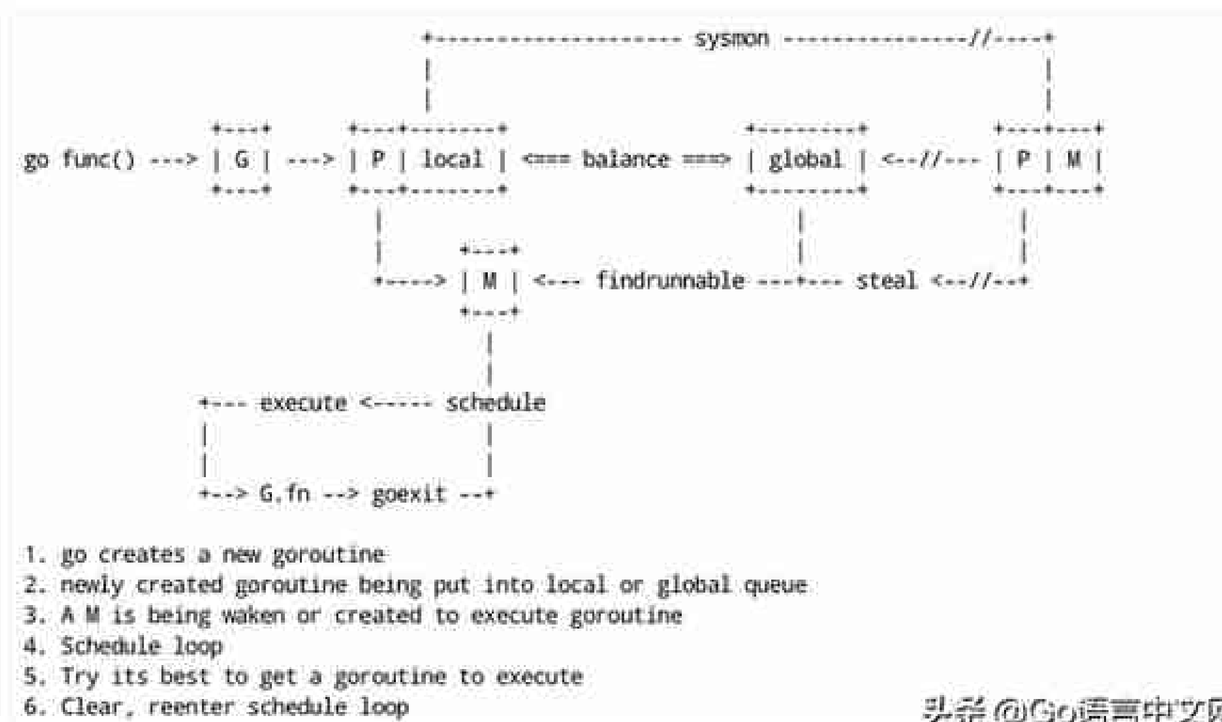
> Being a cooperating scheduler means the scheduler needs well-defined user space events that happen at safe points in the code to

make scheduling decisions.

协作式调度一般会由用户设置调度点，例如 python 中的 yield 会告诉 Os scheduler 可以将我调度出去了。

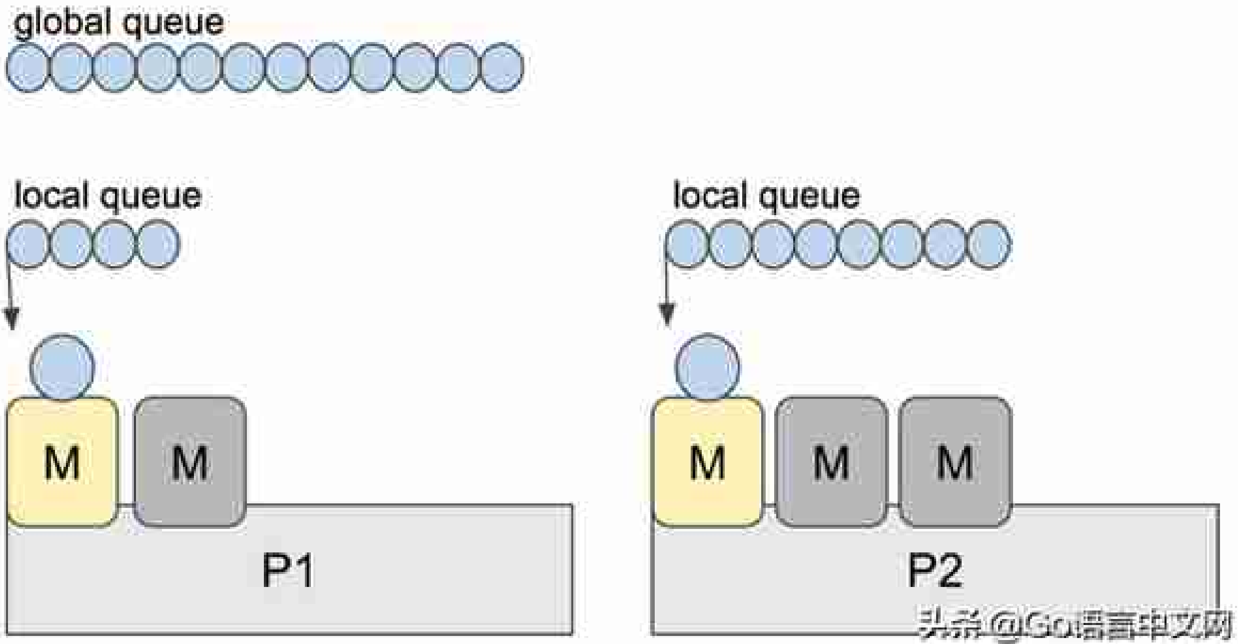
但是由于在 Go 语言里，goroutine 调度的事情是由 Go runtime 来做，并非由用户控制，所以我们依然可以将 Go scheduler 看成是抢占式调度，因为用户无法预测调度器下一步的动作是什么。

和线程类似，goroutine 的状态也是三种（简化版的）：



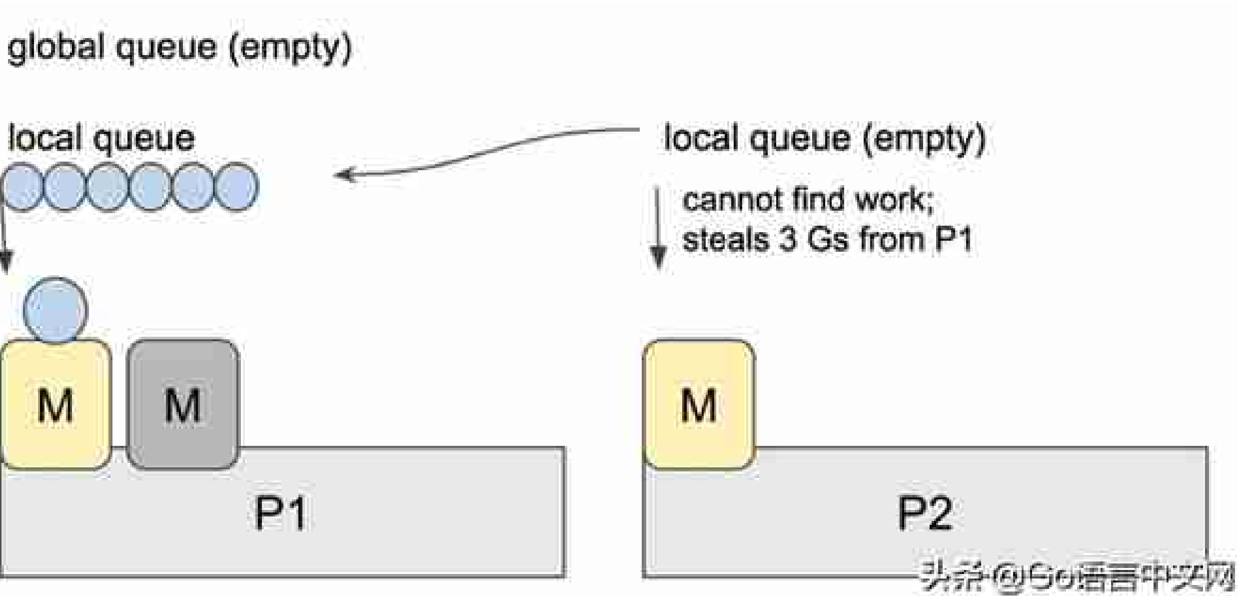
goroutine 调度时机

在四种情形下，goroutine 可能会发生调度，但也并不一定会发生，只是说 Go scheduler 有机会进行调度。



个人感觉，上面这张图比常见的那些用三角形表示 M，圆形表示 G，矩形表示 P 的那些图更生动形象。

实际上，Go scheduler 每一轮调度要做的工作就是找到处于 runnable 的 goroutines，并执行它。找的顺序如下：

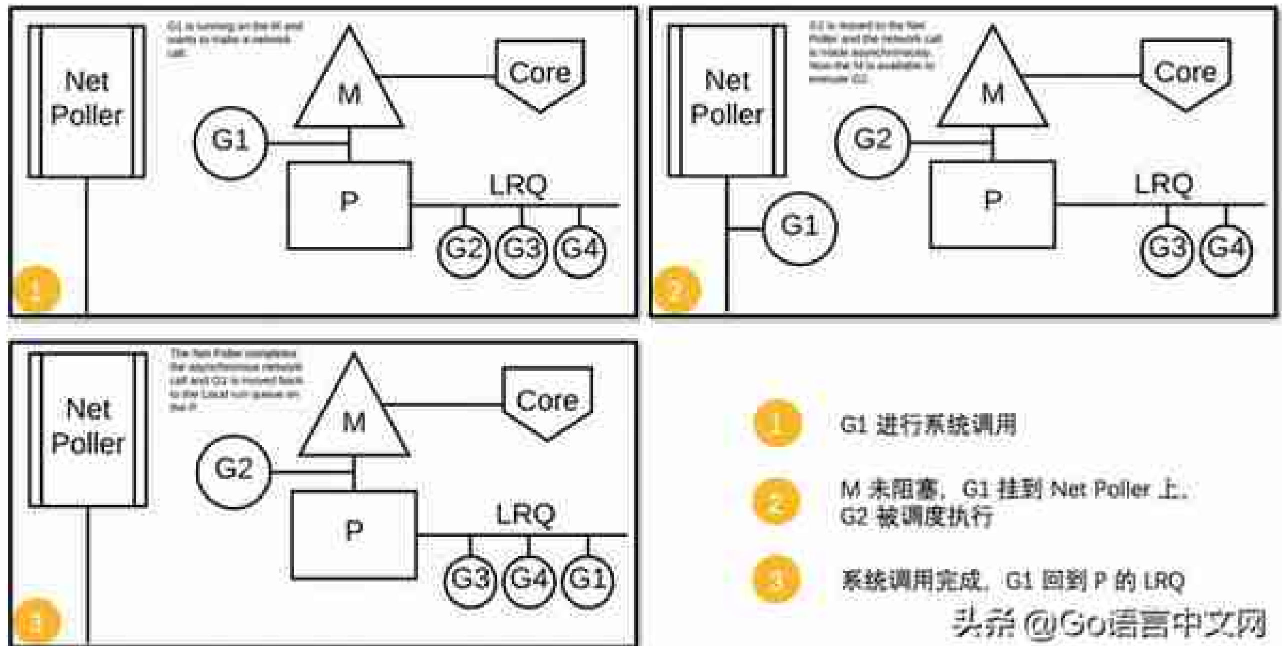


这样做的好处是，有更多的 P 可以一起工作，加速执行完所有的 G。

同步/异步系统调用

当 G 需要进行系统调用时，根据调用的类型，它所依附的 M 有两种情况：同步和异步。

对于同步的情况，M 会被阻塞，进而从 P 上调度下来，P 可不养闲人，G 仍然依附于 M。之后，一个新的 M 会被调用到 P 上，接着执行 P 的 LRQ 里嗷嗷待哺的 G 们。一旦系统调用完成，G 还会加入到 P 的 LRQ 里，M 则会被“雪藏”，待到需要时再“放”出来。



可以看到，异步情况下，通过调度，Go scheduler 成功地将 I/O 的任务转变成了 CPU 任务，或者说将内核级别的线程切换转变成了用户级别的 goroutine 切换，大大提高了效率。

> The ability to turn IO/Blocking work into CPU-bound work at the OS level is where we get a big win in leveraging more CPU capacity over time.

Go scheduler 像一个非常苛刻的监工一样，不会让一个 M 闲着，总是会通过各种办法让你干更多的事。

> In Go, it’s possible to get more work done, over time, because the Go scheduler attempts to use less Threads and do more on each Thread, which helps to reduce load on the OS and the hardware.

scheduler 的陷阱

由于 Go 语言是协作式的调度，不会像线程那样，在时间片用完后，由 CPU 中断任务强行将其调度走。对于 Go 语言中运行时间过长的 goroutine，Go scheduler 有一个后台线程在持续监控，一旦发现 goroutine 运行超过 10 ms，会设置 goroutine 的“抢占标志位”，之后调度器会处理。但是设置标志位的时机只有在函数“序言”部分，对于没有函数调用的就没有办法了。

Golang implements a co-operative partially preemptive scheduler.

所以在某些极端情况下，会掉进一些陷阱。下面这个例子来自参考资料【scheduler 的陷阱】。

```
func main() {
    var x int
    threads := runtime.GOMAXPROCS(0) - 1
    for i := 0; i < threads; i++ {
        go func() {
            for { x++ }
        }()
    }
    time.Sleep(time.Second)
    fmt.Println("x =", x)
}
```

头条 @Go语言中文网

运行结果：

x = 0

不难理解了吧，主 goroutine 休眠一秒后，被 go schduler 重新唤醒，调度到 M 上继续执行，打印一行语句后，退出。主 goroutine 退出后，其他所有的 goroutine 都必须跟着退出。所谓“覆巢之下 焉有完卵”，一损俱损。

至于为什么最后打印出的 x 为 0，之前的文章《曹大谈内存重排》里有讲到过，这里不再深究了。

还有一种解决办法是在 for 循环里加一句：

码农桃花源

我不只在乎输赢，

我还很认真！