

25 | 运筹帷幄：协程的运行机制与调度器原理

2022-12-06 郑建勋 来自北京



天下无鱼

<https://shikey.com/>

《Go进阶·分布式爬虫实战》

[课程介绍 >](#)



讲述：郑建勋

时长 16:56 大小 15.46M



你好，我是郑建勋。

Go 语言以容易编写高并发的程序而闻名。之前我们介绍 Go 语言的网路模型时，就提到了 Go 运行时借助对 I/O 多路复用的封装还有协程的灵巧调度，实现了高并发的网路处理。不过当时我们还没有深入地审视协程这一最重要的 Go 特性，所以在搭建高并发的爬虫模型之前，让我们先来深入看看协程的运行机制，以及调度器是如何实现灵巧调度的。

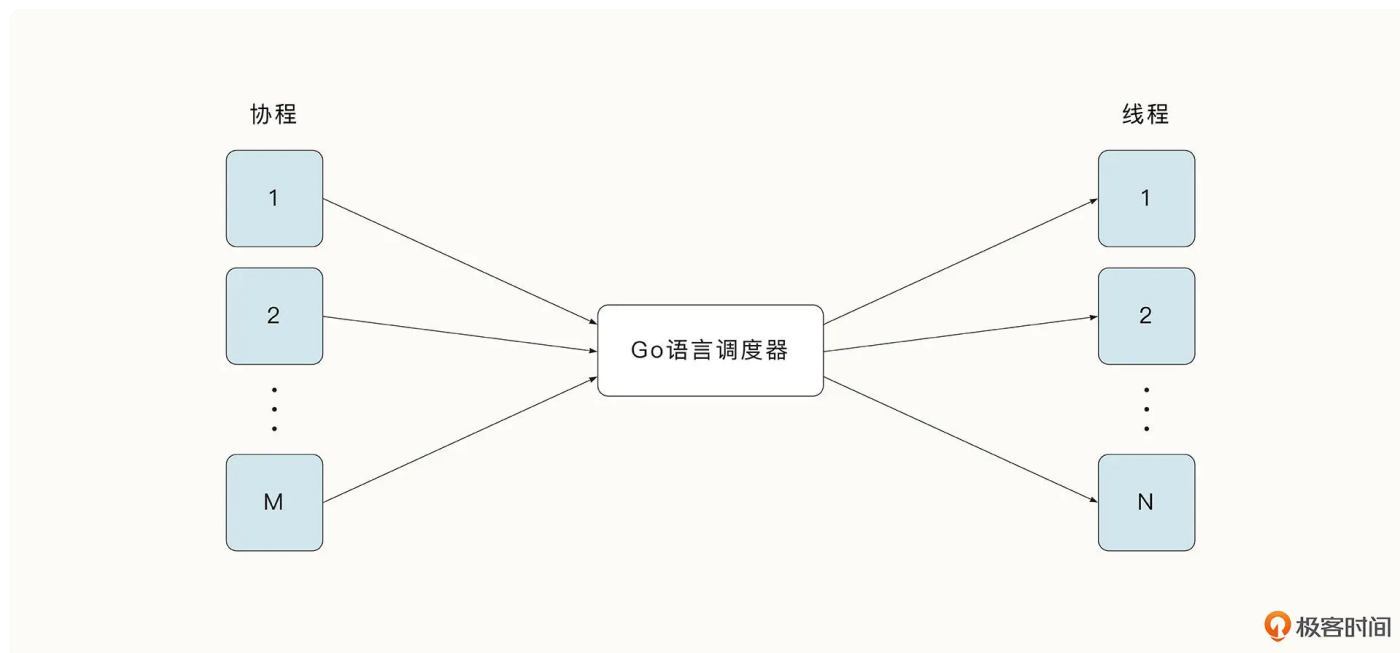
线程 VS 协程

协程一般被认为是轻量级的线程。线程是操作系统资源调度的基本单位，但操作系统却感知不到协程的存在，协程的管理依赖 Go 语言运行时自身提供的调度器。因此准确地说，Go 语言中的协程是从属于某一个线程的，只有协程和实际线程绑定，才有执行的机会。

为什么 Go 语言需要在线程的基础上抽象出协程的概念，而不是直接操作线程呢？要回答这个问题，就需要深入理解线程和协程的区别。下面我就简单从调度方式、上下文切换的速度、调度策略、栈的大小这四个方面分析一下线程和协程的不同之处。

调度方式

Go 语言中的协程是从属于某一个线程的，协程与线程的关系为多对多的对应关系。Go 语言调度器可以将多个协程调度到同一个线程中去执行，一个协程也可能切换到多个线程中去执行。



上下文切换的速度

协程上下文切换的速度要快于线程，因为切换协程不必同时切换用户态与操作系统内核态，而且在 Go 语言中切换协程只需要保留极少的状态和寄存器值（SP/BP/PC），而切换线程则会保留额外的寄存器值（例如浮点寄存器）。

总的来说，线程切换的速度大约为 1~2 微秒，Go 语言中协程切换的速度则比它快数倍，为 0.2 微秒左右。不过上下文切换的速度受到诸多因素的影响，会根据实际情况有所波动。

调度策略

线程的调度在多数时间里是抢占式的，操作系统调度器为了均衡每个线程的执行周期，会定时发出中断信号强制切换线程上下文。而 Go 语言中的协程在一般情况下是协作式调度的，当一个协程处理完自己的任务后，可以主动将执行权限让渡给其他协程。这意味着协程可以更好地

在规定时间内完成自己的工作，而不会轻易被抢占。只有当一个协程运行了太长时间时，Go 语言调度器才会强制抢占其任务的执行。



栈的大小

线程的栈的大小一般是在创建时指定的。为了避免出现栈溢出（Stack Overflow）的情况，默认的栈会相对较大（例如 2MB），这意味着每创建 1000 个线程就需要消耗 2GB 的虚拟内存，大大限制了可以创建的线程的数量（64 位的虚拟内存地址空间已经让这种限制变得不太严重）。而 Go 语言中的协程栈默认为 2KB，所以在实践中，我们经常会看到成千上万的协程存在。

同时，线程的栈在运行时也不能更改。但是 Go 语言中的协程栈在 Go 运行时的帮助下会动态检测栈的大小，并动态地进行扩容。因此在实践中，我们可以将协程看作轻量的资源。

从 GM 到 GMP

协程的调度依赖于线程，下面就让我们看看 Go 运行时是如何将协程与线程绑定在一起的。

在 Go 源码中，结构体 m 代表了操作系统线程。结构体 m 中包含了特殊的调度协程 g0，绑定的逻辑处理器 P，绑定的用户协程 g 等重要结构。

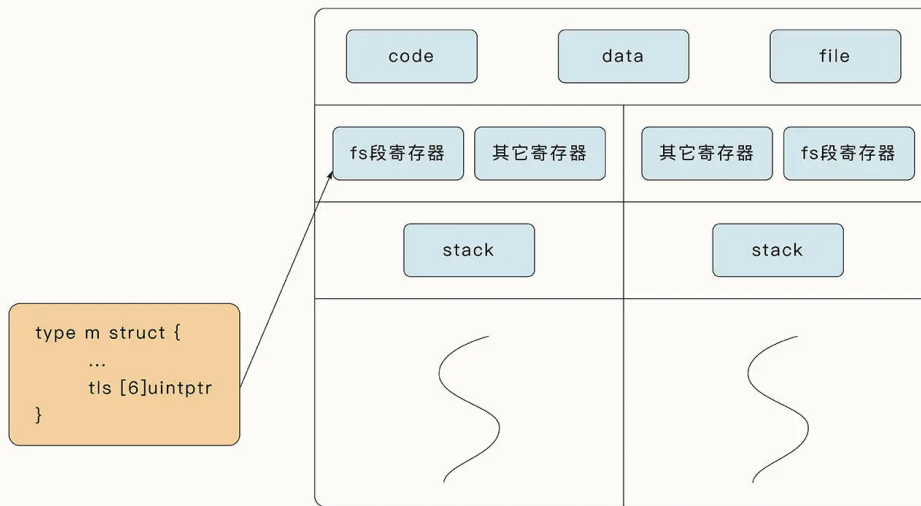
 复制代码

```
1 type m struct {
2     g0      *g      // 特殊的调度协程g0
3     p       uintptr // m当前对应的逻辑处理器P
4     curg    *g      // 当前m绑定的用户协程g
5     tls     [6]uintptr // 线程局部存储
6     ...
7 }
```

结构体 m 要与真实的操作系统线程绑定在一起，这就需要借助线程本地存储技术了。和普通的全局变量对程序中的所有线程可见不同，线程本地存储中的变量只对当前线程可见。因此，这种类型的变量可以看作是线程“私有”的。一般情况下，操作系统会使用 FS/GS 段寄存器存储线程本地变量。

在 Go 语言中，并没有直接暴露线程本地存储的编程方式，但是 Go 语言运行时使用线程本地存储，将具体操作系统的线程与运行时代表线程的 m 结构体绑定在了一起。线程本地存储的

数据实际是结构体 **m** 中 **m.tls** 的地址，同时，**m.tls[0]** 会存储当前线程正在运行的协程 **g** 的地址，因此在任意一个线程内部，通过线程本地存储，都可以在任意时刻获取绑定在当前线程上的协程 **g**、结构体 **m**、逻辑处理器 **P**、特殊协程 **g0** 等的信息。



极客时间

线程局部存储帮助我们实现了结构体 **m** 与实际线程的绑定，不过此外，我们还需要实现结构体 **m** 与某一个协程的绑定，这就要用到调度器了。在 **Go1.1** 之前的源码实现中，调度器还是用 **C** 语言实现的，无论是线程启动还是协程切换时，都会执行调度函数 **schedule**，**schedule** 再从全局队列中获取可运行的协程并予以执行。

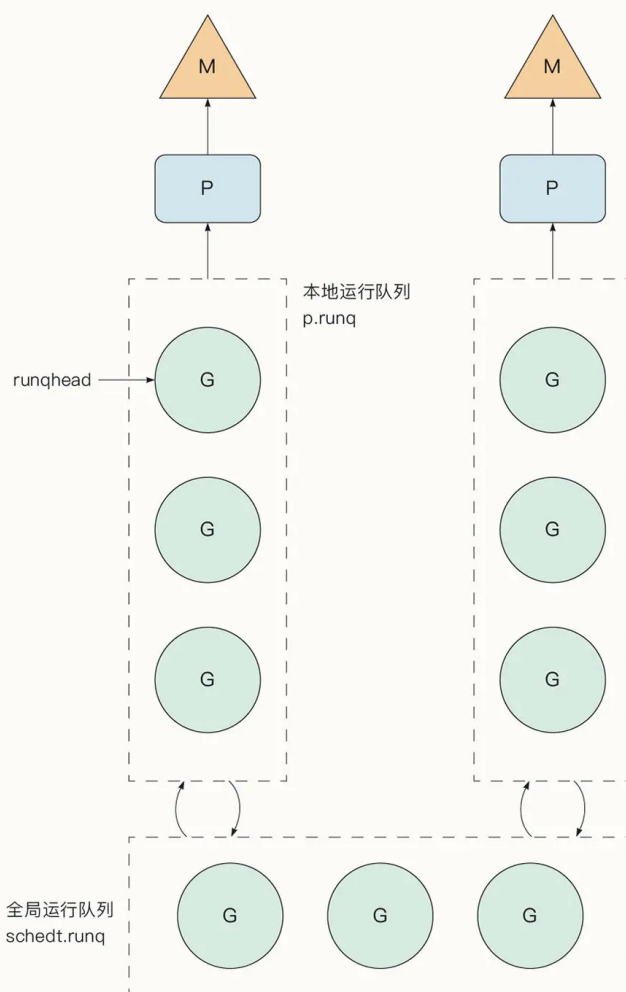
复制代码

```
1 static void
2 schedule(G *gp)
3 {
4     ...
5     schedlock();
6     if(gp != nil) {
7         ...
8         switch(gp->status){
9             case Grunning:
10                 gp->status = Grunnable;
11                 gput(gp);
12                 break;
13         }
14
15         gp = nextgandunlock();
16         gp->readyonstop = 0;
17         gp->status = Grunning;
18         m->curg = gp;
19         gp->m = m;
```

```
20 ...
21 runtime·gogo(&gp->sched, 0);
22 }
```

这种方式有许多不足，其中最核心的问题就是，调度器每次获取可以运行的协程都需要加锁，随着 CPU 核心数量的增多，这种方式缺少扩展性的问题会越来越明显。此外，当协程执行系统调用时，线程还会整个被堵塞住。

为了解决上面的问题，Go 团队对调度器进行了很大的优化，其中一个最重要的优化就是引入了逻辑处理器 P。逻辑处理器 P 和唯一的线程 M 绑定，逻辑处理器 P 可以在本地存储协程的运行队列，同时也保留了全局的运行队列，稍后我们会看到它们之间的交互。

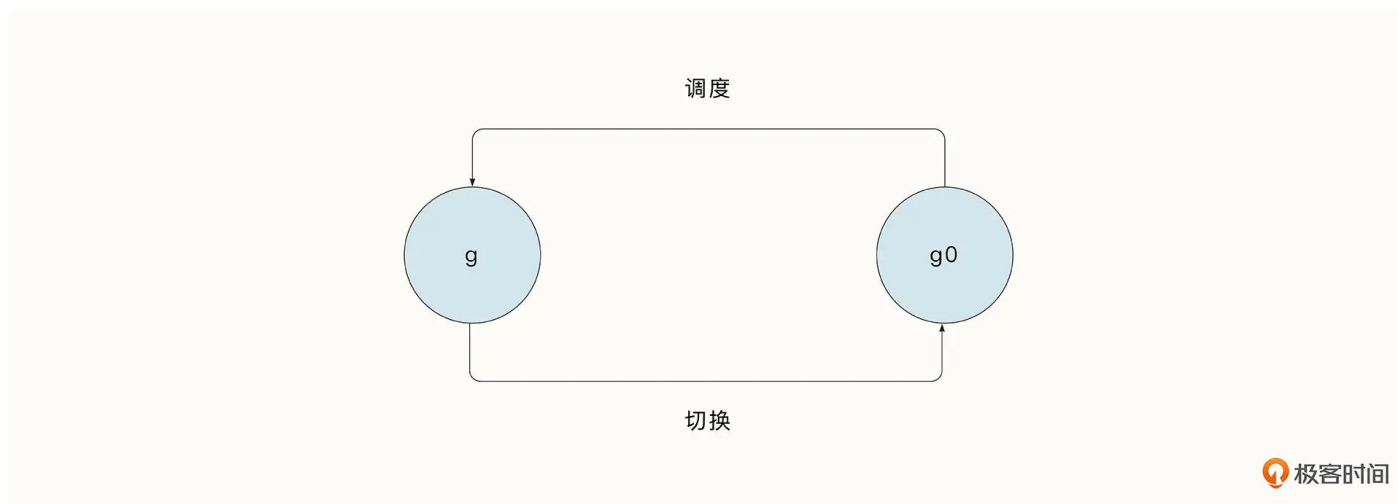


逻辑处理器 P 与 M 绑定的特性决定了，正常情况下有多少个 P 就会有对应数量的线程存在。

假设现在有 4 个 P，我们就知道有 4 个线程。我们通常会认为，这代表着能够并行执行的协程数量。默认情况下，Go 运行时会读取 CPU 核心的数量，并让创建的逻辑处理器 P 的数量和机器 CPU 核心的数量相同。当然，我们也可以通过配置环境变量中的 GOMAXPROCS 来指定 P 的数量。我们后面会看到，为什么一些特殊的场景需要调整 P 的数量。

另外，每一个 M 结构中都存储了一个特殊的协程 g0，协程 g0 运行在操作系统的线程栈上，它的主要作用是执行协程调度的一系列运行时代码，一般的协程则负责无差别地执行用户代码。

很显然，执行用户代码的任何协程都不适合进行全局调度。当用户协程退出或者被抢占时，意味着需要重新执行协程调度，这时，我们需要从用户协程 g 切换到协程 g0，这样才能完成协程的调度。



协程经历从 $g \rightarrow g0 \rightarrow g$ 的过程之后，就完成了一次调度循环。和线程类似，协程切换的过程叫作协程的上下文切换。

当某一个协程 g 执行上下文切换时，需要保存当前协程的执行现场，才能够在后续切换回 g 协程时正常执行。协程的执行现场存储在 g.gobuf 结构体中，g.gobuf 结构体主要保存 CPU 中几个重要的寄存器值，分别是 rsp、rip、rbp。

复制代码

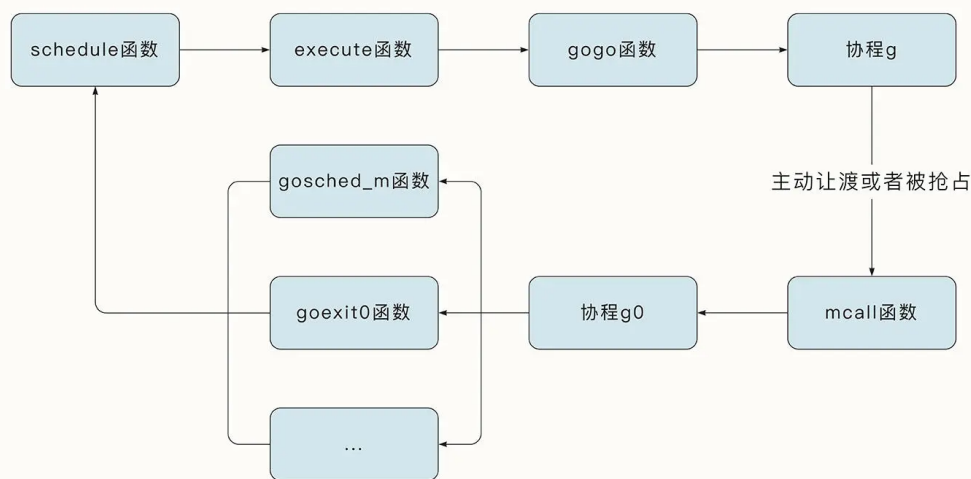
```
1 type gobuf struct {
2     // 保存CPU 的rsp 寄存器的值
3     sp uintptr
4     // 保存CPU 的rip 寄存器的值
5     pc uintptr
6     // 记录当前这个gobuf 对象属于哪个Goroutine
7     g guintptr
```

```
8 // 保存系统调用的返回值
9 ret sys.Uintreg
10 // 保存CPU 的rbp 寄存器的值
11 bp uintptr
12 ...
13 }
```

调度循环

从协程 **g0** 调度到协程 **g**，经历了从 **schedule** 函数到 **execute** 函数再到 **gogo** 函数的过程。其中，**schedule** 函数处理的是具体的调度策略，也就是选择下一个要执行的协程；**execute** 函数执行的是一些具体的状态转移、协程 **g** 与结构体 **m** 之间的绑定等操作；**gogo** 函数是与操作系统有关的函数，用于完成栈的切换以及恢复 **CPU** 寄存器。执行完这一步之后，协程就会切换到协程 **g** 去执行，当协程 **g** 主动让渡、被抢占或退出后，又会切换到协程 **g0** 开始新一轮调度。

在从协程 **g** 切换回协程 **g0** 时，**mcall** 函数会保存当前协程的执行现场，**mcall** 函数是和平台有关的汇编指令。协程切换到 **g0** 后，根据切换原因的不同，会执行不同的函数。例如，如果是用户调用 **Gosched** 函数主动让渡执行权，就会执行 **gosched_m** 函数；如果协程已经退出，则执行 **goexit0** 函数，将协程 **g** 放入 **p** 的 **freeg** 队列，方便下次重用。执行完毕后，运行时再次调用 **schedule** 函数开始新一轮的调度循环，从而形成一个完整的闭环，循环往复。



调度器原理

前面我们看到了 GMP 是如何绑定在一起的，了解了调度循环的过程。接下来我们具体看一看调度器是如何对协程进行调度的。



调度算法

调度的核心策略位于 `schedule` 函数中。

 复制代码

```
1 // runtime/proc.gofunc
2 schedule() {
3     ...
4 }
```

由于程序中不可能同时执行成千上万个协程，因此，那些等待被调度的协程就存储在了运行队列中。

Go 语言调度器将运行队列分为局部运行队列与全局运行队列。局部运行队列是每个 P 特有的长度为 256 的数组。这个数组模拟了一个循环队列，其中，`runqhead` 标识了循环队列的开头，`runqtail` 标识了循环队列的末尾。每次将 G 放入本地队列时，都是从循环队列的末尾插入，而获取 G 时则是从循环队列的头部获取。

除此之外，在每个 P 内部还有一个特殊的 `runnext` 字段，它标识了下一个要执行的协程。如果 `runnext` 不为空，则会直接执行当前 `runnext` 指向的协程，不会再去 `runq` 数组中寻找。

 复制代码

```
1 type p struct {
2     // 使用数组实现的循环队列
3     runq [256]guintptr
4     runnext uintptr
5 }
```

一般的思路是，先查找每个 P 局部的运行队列，当获取不到局部运行队列时，再从全局队列中获取。但是这种方法可能存在问题，如果只是循环往复地执行局部运行队列中的 G，那么全局队列中的 G 可能一直无法执行。为了避免出现这种情况，Go 语言调度器制定了一种策略：P 每执行 61 次调度，就需要从全局运行队列中查找一批协程，分配给本地运行队列。


```

1 if _g_.m.p.ptr().schedtick%61 == 0 && sched.runqsize > 0 {
2     lock(&sched.lock)
3     // 从全局运行队列中获取1 个G
4     gp = globrunqget(_g_.m.p.ptr(), 1)
5     unlock(&sched.lock)
6 }

```

 复制代码



天下无鱼

<https://shikey.com/>

这个时候你可能会问了，如果本地运行队列已经满了，无法处理全局运行队列中的协程怎么办？

如果本地运行队列满了，那么调度器会将本地运行队列的一半放入全局运行队列。这就确保了当程序中有很多协程时，每个协程都有执行的机会。

如果局部运行队列和全局运行队列中都找不到可用的协程，这时，调度器会寻找当前是否有已经准备好运行的网络协程。`runtime.netpoll` 函数会获取当前可运行的协程列表，返回第一个可运行的协程，并通过 `injectglist` 函数将其余协程放入全局运行队列等待被调度。

当局部运行队列、全局运行队列以及准备就绪的网络列表中都找不到可用协程时，调度器就需要从其他 P 的本地队列中窃取可用的协程来执行了。

怎么做呢？

由于所有的 P 都存储在全局的 `allp []*p` 中，一种可以想到的简单的方法就是循环遍历 `allp`，找到可用的协程，然后去窃取协程。但是这种方法很显然缺少公平性，在数组前面的 P 将会被窃取得更多。Go 语言采取了一种独特的方式，它的代码位于 `findrunnable` 函数中。

```

1 func findrunnable() (gp *g, inheritTime bool) {
2     for i := 0; i < 4; i++ {
3         for enum := stealOrder.start(fastrand()); !enum.done(); enum.next()
4         {
5             ...
6         }
7     }
8 }

```

 复制代码

findrunnable 函数尝试循环 4 次，并随机遍历 `allp` 数组，找到可窃取的 `P` 就立即窃取并返回。



我们用一个例子来说明一下随机调度算法的原理。假设一共有 8 个 `P`，第 1 步，`fastrand` 函数会借助随机算法选择一个数并对 8 取模，假设最后结果为 6。

第 2 步，找到一个比 8 小且与 8 互质的数。比 8 小且与 8 互质的数有 4 个：`coprimes=[1,3,5,7]`，代码中取 `coprimes[6%4] = 5`，这 4 个数中任取一个都有相同的数学特性。计算过程为：

复制代码

```
1 (6+5) %8 = 3
2 (3+5) %8 = 0 (0+5) %8 = 5 (5+5) %8 = 2 (2+5) %8 = 7 (7+5) %8 = 4 (4+5) %8 = 1
3 (1+5) %8 = 6
```

可以看到，这里将上一个计算的结果作为下一个计算的条件，这就保证了一定会遍历到 `allp` 数组中所有的 `P`。

找到要窃取的 `P` 之后就可以正式开始窃取了，这部分的核心代码位于 `runqgrab` 函数中。窃取的核心逻辑比较简单，将要窃取的 `P` 本地运行队列中 `Goroutine` 个数的一半放入自己的运行队列中。

我们总结一下查找协程的先后顺序：

1. 获取需要执行垃圾回收的后台标记协程；
2. 获取 `P.runnext` 中待运行的协程；
3. 获取 `P` 的本地运行队列中待运行的协程；
4. 获取全局运行队列中待运行的协程；
5. 获取已经准备好要运行的网络协程；
6. 窃取其他 `P` 中待运行的协程。

上面我们看到了调度器的调度策略，我们再来看看什么时候会触发对协程的调度。根据调度方式的不同，可以将调度时机分为主动调度、被动调度和抢占调度。

主动调度

协程可以选择主动让渡自己的执行权利，这主要是通过用户在代码中执行 `runtime.Gosched` 函数实现的。在大多数情况下，用户并不需要执行这个函数，因为 Go 语言编译器会在调用函数之前插入检查代码，判断这个协程是否需要被抢占。

但是还是会有一些特殊的情况。例如一个密集计算，无限 `for` 循环的场景，这种场景由于没有抢占的时机，在 Go 1.14 版本之前是无法被抢占的，这在 CPU 密集型场景下会出现其他任务无法及时被执行的情况。后面我们会看到，Go 1.14 之后的版本对于长时间执行的协程，都使用了操作系统的信号机制进行强制抢占。

被动调度

被动调度指协程因为在休眠、Channel 通道堵塞、网络 I/O 堵塞、执行垃圾回收而暂停时，被动让渡自己执行权利的过程。被动调度具有重要的意义，它可以保证 CPU 的资源利用率最大化。根据被动调度原因的不同，调度器可能执行一些特殊的操作。

由于被动调度仍然是协程发起的操作，因此它的调度时机相对明确。和主动调度类似的是，被动调度需要先从当前协程切换到协程 `g0`，更新协程的状态，并与 `M` 解绑，重新调度。和主动调度不同的是，被动调度不会将 `G` 放入全局运行队列，因为当前的 `G` 还是不可以运行的，需要一个额外的唤醒机制。

我以通道的堵塞为例说明一下被动调度的过程。在这个例子里，通道 `C` 会一直等待通道中的消息。当通道中暂时没有数据时，协程会陷入阻塞状态。

```
1 func recieve(c chan int) {  
2     <-c  
3 }
```

复制代码

当其他协程向相同的通道发送消息后，堵塞的协程需要被唤醒，这时会先将协程的状态从 `_Gwaiting` 转换为 `_Grunnable`，并添加到当前 `P` 的局部运行队列中。

抢占调度

为了让每个协程都有执行的机会，并且最大化利用 CPU 资源，Go 语言在初始化时会启动一个特殊的线程来执行系统监控任务。



系统监控在一个独立的 M 上运行，不用绑定逻辑处理器 P，系统监控会每隔 10ms 检测一下有没有准备就绪的网络协程，如果有就放置到全局队列中。和抢占调度相关的是，系统监控服务会判断当前协程是否运行时间过长，或是否处于系统调用阶段，如果是，则会抢占当前 G 的执行。它的核心逻辑位于 runtime.retake 函数中。

执行时间过长的抢占调度

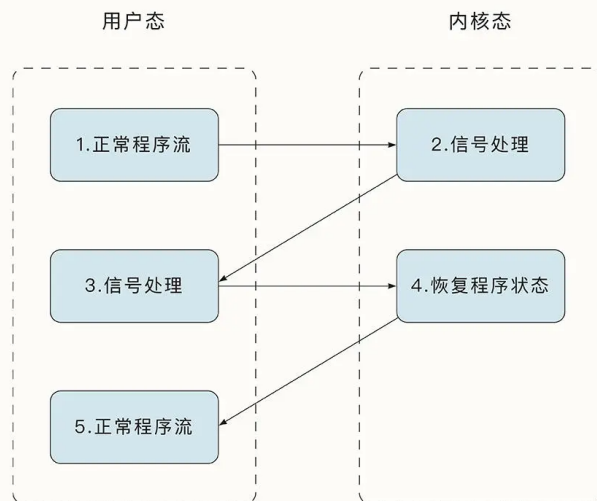
在 Go 1.14 之前，虽然仍然有系统监控抢占时间过长的 G，但是这种抢占的时机发生在函数调用阶段，因此没有办法解决对于死循环的抢占，就像下面这个例子一样：

 复制代码

```
1 for{
2     i++
3 }
```

为了解决这一问题，Go 1.14 之后引入了信号强制抢占的机制。这需要借助下图中的类 UNIX 操作系统信号处理机制。

信号是发送给进程的各种通知，它可以将各种重要的事件都通知给进程。运行时系统监控程序借助操作系统信号中断当前程序，保存程序的执行状态和寄存器值，并切换到内核态处理信号。在内核态处理完信号后，操作系统还会返回到用户态执行程序注册的信号处理函数，之后再回到内核恢复程序原始的栈和寄存器值，并切换到用户态继续执行程序。



在信号处理时，Go 语言借助用户态注册的信号处理程序完成了协程的上下文切换，最终实现了抢占调度。

陷入到系统调用中的抢占调度

还有一类特殊的情况涉及到协程长时间堵塞在系统调用中的问题。这时，当前正在工作的线程会陷入等待状态，等待内核完成系统调用并返回。如果当前局部运行队列中有等待运行的 G，或者当前系统调用的时间过长，运行时系统监控程序会进行抢占调度。

系统调用时的抢占原理主要是将 P 与陷入系统调用的 M 解绑，并建立（或者从缓存池中获取）一个新的 M 与 P 绑定，然后开始新一轮的调度。而之前陷入到系统调用的 M 从内核返回后，会尝试与 P 进行绑定，如果没有空闲的 P，则当前 M 会陷入到休眠中。

正是因为 Go 运行时定时地对陷入到系统调用中的协程进行抢占调度，才确保了即便每一个协程都陷入到系统调用中，也不会阻塞任意一个 P。P 会与新的 M 绑定，确保始终有 GOMAXPROCS 数量的协程在执行用户任务，这大大增加了 CPU 资源的利用率。

不过，Go 的调度机制也存在一些局限，由于 Go 运行时需要 10ms 的时间才能做出反应实现抢占，因此对于会频繁陷入到操作系统堵塞的程序，特别是磁盘 I/O 密集型的程序，这 10ms 的反应时间也会导致 CPU 资源的浪费，因为这段时间里 CPU 本可以处理更多的资源。所以对于磁盘 I/O 密集型的程序，我们一般会调大 GOMAXPROCS 的数量，提升系统的性能。

不过，如果这个系统的线程数随着系统堵塞持续增长，线程数可能会远远大于 GOMAXPROCS 的数量，而且确实有过磁盘 I/O 堵塞，系统积压了上万个线程导致 panic 的案例。如果遇到类似情况，观测线程的数量指标是很有必要的。



总结

这节课就讲到这里。我们知道，Go 语言比较容易开发高并发的程序，这得益于 Go 语言在线程之上创建了更轻量级的协程 G。相比于线程，协程在时间和空间上都有明显的优势。同时，Go 运行时抽象出了逻辑处理器 P 和代表线程的 M，P 与 M 一一绑定。借助 M 中特殊的协程 g0，Go 运行时能够完成对于协程公平并且高效的调度。

协程一般是被动调度的，当它陷入堵塞后，会主动让渡自己的执行权利，这和操作系统通常强制执行线程的上下文有所不同。同时，Go 运行时也存在系统监控，它会每隔 10ms 强制切换长期运行或者陷入系统调用的协程。Go 对系统调用做了一些封装，导致当协程被堵塞时，不会真正阻塞某一个 P。P 会与新的 M 绑定，确保始终有 GOMAXPROCS 数量的协程在执行用户任务，这大大增加了 CPU 资源的利用能力。Go 运行时就是依靠着这些灵巧的调度实现了对于海量协程的管理。


课后题


学完这节课，请你思考下面两个问题。

1. Go 没有暴露协程的 ID，但其实在内部每一个协程都是有一个 ID 的，你知道 Go 为什么这样设计吗？
2. 协程是很轻量级的资源，你觉得在实践中 Go 有必要设计像线程池这样的协程池吗？

欢迎你在留言区与我交流讨论，我们下节课见。

分享给需要的人，Ta 购买本课程，你将得 20 元

 生成海报并分享

 赞 1  提建议

精选留言 (3)

 写留言



江楠大盗

2022-12-07 来自北京

这节课太干了，全是干货，赞赞赞



Realm

2022-12-06 来自浙江

1. 谷歌开发者不建议大家获取协程ID，避免开发者滥用协程Id实现Goroutine Local Storage，滥用协程ID会导致GC不能及时回收内存。

2 协程池在大并发的场景中很有必要,虽然goroutine开销很小,无休止开辟Goroutine，会高频率的调度Groutine，在上下文切换上浪费很多资源。

<https://github.com/panjf2000/ants> 这个是一个高性能协程池。



shuff1e

2022-12-06 来自上海

谷歌开发者不建议大家获取协程ID，主要是为了GC更好的工作，滥用协程ID会导致GC不能及时回收内存。如果一个第三方库使用了协程ID，那么使用该库的人将会莫名中招。

