Golang学习日记III - 并发机制

-- by Levy, 2017-03-11 08:53

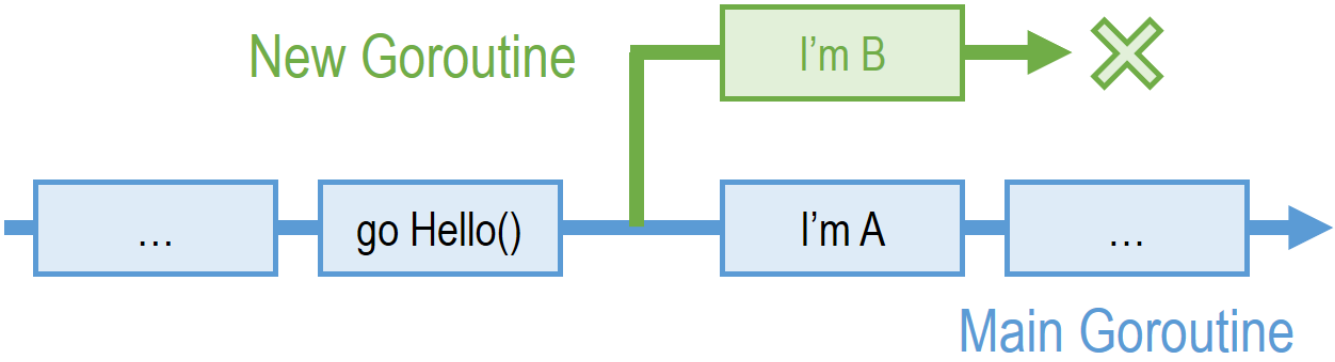
深坑「Golang学习日记」总算在有生之年被我决定续更了。由于对普通语法的介绍网上资源极多，Go官方的上手指南A Tour of Go(请自备梯子)就是极好的例子，我不再打算就语法细节进行详述。这次，让我们直切肯綮，从Go最大的卖点入手——并发(Concurrency)。

```
func Hello() {
    fmt.Println("I'm B")        // Output A
}
go Hello()
fmt.Println("I'm A")            // Output B
```

如果在双核（及以上）的机器编译运行上述Go代码，我们能观测到A/B输出的顺序随着运行次数的不同而不同，也就是说，仅依靠5行代码，我们就创建了两线并发的程序。相较于C/C++/Java/Python等语言为了创建一个并发执行环境所需要的调用POSIX-API/定义继承类等繁琐步骤，Golang简单一句 `go func()`的确给人眼前一亮的感觉。当然了，仅凭语法上的简洁显然不足以成为一个编程语言拿来吹嘘的资本，下文我们将对在这几行语句下Golang的并发机制和实现进行详细探索。

一等公民-Goroutine

Goroutine是Go的并发机制中绝对的主角。它代表了指令流及其执行环境，也是被调度的基本单位。宏观来看，goroutine类似操作系统中线程的概念（注意这里的类比并不严格，下文将会对两者做出详细比较）：不同线程间共享同一个内存空间，但不共享栈且各自并发执行；同样地，goroutine也同内存不同栈，并发运行。



如上图所示，上文代码片段第四行的`go Hello()`会创建一个新的goroutine（绿色线条），并开始执行`Hello()`函数。需要注意的是，由于主goroutine（蓝色线条）和新创建的goroutine拥有并发性，且主goroutine在执行`go Hello()`时并不会等待被调用函数执行结束，故“I'm A”（主goroutine输出）和“I'm B”（新goroutine输出）可能以任何顺序交错展现。

为何不用线程(pThread)?

直到现在，我们并不能从goroutine中看到任何有别于thread、从而促成Golang编写者抛弃传统的线程模型自己造轮子的地方。那么操作系统层面的线程(pThread)有什么问题呢?

生命周期开销太高

线程的创建、销毁和切换都需要一系列系统调用，而每一个系统调用意味着触发软中断、进入内核态、将寄存器的值全部存入内存、维护相关数据结构、恢复寄存器、返回用户态等一系列组合拳。这一轮操作不仅十分耗时、还可能让内存缓存的加速效果大幅度下滑。所以，避免频繁创建、销毁线程作为高性能并发的必要条件这一点已成为程序员的共识。

以线程为并发模型的C/C++/Java采用线程池的方法来降低线程昂贵的生命周期开销。既然线程创建/死亡代价高昂，我们何不让创建的线程永不死亡呢？具体来说，对于每个已经创建但已经完成工作的线程，我们令其休眠，并放进一个资源池中，在下次需要新的线程的时候，我们直接将线程池中休眠的线程拿出来唤醒使用而非新建线程。这样一来，绝大部分的线程创建/销毁需求都成功地被线程池吸收了。进一步，通过规定线程池的最大容量，我们可以将花费在线程创建和销毁上的开销控制在固定值，例如，常见的J

ava Web应用会设立一个30~50大小的线程池来处理HTTP请求，并取得非常好的并发效果。

不必要的线程切换

即使线程池很好地砍掉了线程生命周期开销，操作系统层面的线程依然存在不足：线程的语义在于并行，当线程数超出CPU核心数时，操作系统会定时给每个CPU核心切换不同的线程，让他们“看上去”是同时在进行的。当然，这样的切换同样需要付出若干中断、系统调用，以及当前线程的工作集从缓存中被新线程完全抹去的代价。

乍一听上去这样的代价是必不可少的，实则不然。由于在绝大部分时候我们的应用都是I/O和计算混合的，即，一段时间与硬盘/网络交互（I/O）、一段时间进行相对密集的内存访问和计算，而等待I/O完成期间该线程处于休眠状态，CPU已经会切换到其他线程，即使操作系统不强行打断并切换处于计算密集期的线程，应用在宏观上依然显示出一定并发性。而通过去掉计算密集期的线程切换，整体CPU效率得到了有效提升——NodeJS就是在这样的哲学下诞生的：单一线程、全异步的I/O、事件驱动、非抢占式调度（当某一个函数单纯进行计算和内存访问时不会被打断），在进行I/O密集型工作（如网站后台）时通过将单一CPU利用率逼到100%的方式在效率上力挫几乎其他所有能利用多线程多核脚本语言。这简直是本来就特立独行的Javascript对整个编程语言界的同僚竖起的又一根中指。当然了，仅仅能利用单核处理能力的NodeJS在处理对计算要求更高的工作上显然会力不从心，但其给我们的启示值得注意。

较高的切换开销

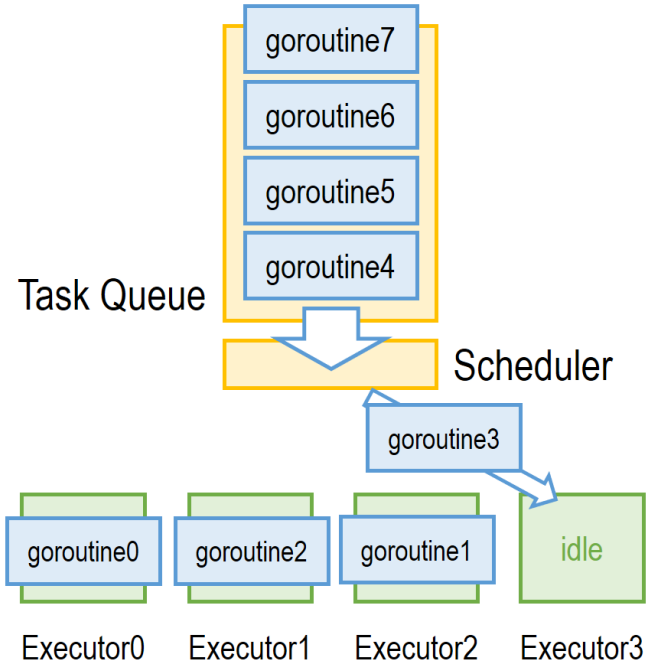
在锁竞争、协程同步等情况下，频繁进入内核态的线程模型会放大自身在切换开销上的劣势。而用户态的调度器（如goroutine调度器）则可以在用户态处理这一切，省时省力。另外，由于编程语言能够更好地对自己语言中的同步原语进行分析，编程语言自己的调度器能够更好地根据语义对调度进行优化。

Goroutine调度模型

Go使用用户态的调度器对goroutine的执行进行控制，从而避免了大部分内核开销。具体而言，Golang的调度模型由三部分组成：**执行环境(Executor)**、**调度器(Scheduler)**和**goroutine**。

执行环境，顾名思义，用来执行代码。尽管其在抽象概念上应该对应一个CPU核心，但由于在用户态不能接触硬件资源，故Go将其具体实现为线程。当线程数等于CPU核心数时，既最大化了CPU核心利用率，又最小化了线程切换的开销，是最理想的情况（当然，实际情况下操作系统还会运行、切换来自其他进程的线程，但这已经超出一个普通程序的控制范畴）。故默认情况下，用于指定执行环境个数的运行时变量 `GOMAXPROCS`等于CPU核心数目。当然，开发者可以根据自己的需求更改该值，当 `GOMAXPROCS=1`时，Go的执行模型几乎等同于NodeJS。

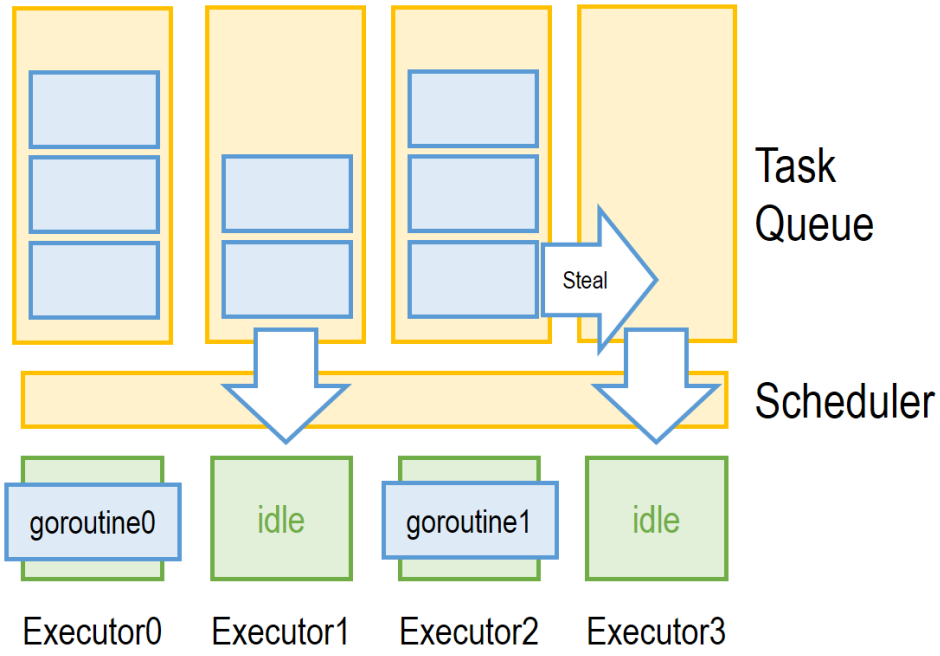
调度器则是调度模型的核心，它决定了每个执行环境（核）在什么时候执行什么样的goroutine。Go采用任务队列的方式对goroutine进行调度：



如上图所示，所有goroutine作为任务排在任务队列中，而scheduler所做的则是在executor空闲时从队首拿出下一个goroutine给其执行。每个任务(goroutine)会被executor执行到完成或阻塞（如发起I/O请求、系统调用、请求一个正在被其他人使用的锁或自行yield计算资源等），在第二种情况下，该goroutine既不在executor也不在队列中，而是处于阻塞态被Scheduler监视直到阻塞结束重新入队。值得注意的是，这里与上文提到的“去掉计算密集期的线程切换”的联系：由于调度器对任务采用非抢占式调度，即在正常计算和内存访问的情况下executor不会放弃当前goroutine，故多余的goroutine切换代价得以被去除。

这样的任务队列模型仍然存在不小的问题：由于任务队列只有一个，为了保证出入队的原子性，任务分配/加入时需要对整个队列加互斥锁，当goroutine执行时间短时，频繁

给大量executor分配新任务会让单一队列成为并行的性能瓶颈。为了解决该问题，Go采用了多任务队列的方式进行任务调度：



如上图所示，在多任务调度模型中，每个executor均有一个自己对应的任务队列。在正常情况下，每个executor从自己的队列中拿goroutine，并将生成的新goroutine放进自己队列队尾。分布式结构可能带来的问题是显而易见的：如果任务在队列的分布不均匀会导致计算资源的浪费，如上图中的executor3，如果缺乏其他措施，该核会因为对应队列没有任务而空闲。对于该问题，Go的解决方法是引入“偷任务”机制：当Scheduler发现某队列无任务可用时，会从其他队列里“偷”一部分任务过来。由于偷任务的代价较高（需要锁两个队列），Scheduler会争取一次性偷足够多的任务以降低未来偷任务的频率。

而对于处于阻塞状态的goroutine，Scheduler需要监视其脱离阻塞状态并重新入队。Goroutine被阻塞的原因大体分两种：

- 阻塞I/O或系统调用。由于底层实现限制，该类阻塞需要一个线程显式执行相应的syscall并等待调用返回。在这种情况下，Scheduler会新建一个线程执行该syscall，并在返回后通知Scheduler。同样地，为了节省开销，该线程被维护在线程池中。值得注意的是，该类线程由于整个生命周期都几乎在等待阻塞（阻塞结束后立即通知Scheduler而后结束），而阻塞的线程是不参与操作系统线程切换的，故其并不会带来太大的线程切换开销。当然，如果借鉴NodeJS、尽可能用异步版本api替换同步版，则可以省去线程池操作，进一步优化性能（Go是否采用该优化尚存疑）。

- 内部同步机制，Goroutine因为调用了Go内部同步机制（channel、互斥锁、wait group、conditional variable等）而阻塞。对于此类阻塞，由于同步机制的语义是Go定义从而对Scheduler透明的，Scheduler可以分析出阻塞依赖，从而将监视该阻塞状态的任务交给其依赖的goroutine。例如，goroutine A请求了一个正被goroutine B获取了的互斥锁，从而陷入阻塞，那么Scheduler可以在goroutine B释放该锁时由对应的executor将goroutine A唤醒并加入队列。在这整个过程中不需要引入新的线程。

以上便是Golang Scheduler的大致工作逻辑，在各个组件的相互配合下，一个高性能、支持调度成千上万goroutine的并发环境就此搭建起来。

总结和启发

从Golang的并发机制中我们可以得到如下几点启发：

- 系统调用和内核态是昂贵的，用户态的调度器拥有更好的性能。
- 由于频繁进行不必要的切换，线程并不是合适的并发执行基本单位；相反，将线程作为执行资源(CPU)的抽象、为一个CPU核心建立一个线程作为执行器则是一个很不错的主意。
- 单一任务队列在任务短而多时劣势明显，分布式队列+任务偷取能够较好的解决问题。

可以说，Golang的并发机制是NodeJS的普适版，拥有能够更好利用多核计算力的优势；和采用OS线程、阻塞I/O、GIL的Python并发模式相比则更是云泥之别。正是更为精巧的并发机制和简单的并发原语，使得Concurrency成为Go语言最大的卖点。

需要指出的是，Go所采用的一切技术都并非原创——`go func()`的同步原语与Cilk十分类似，分布式任务队列也多少有模仿Cilk/OpenMP的意味，如果非要说不同之处，大概在于Go是一个原生支持该功能的完整编程语言，而另外两者只是C/C++的语法扩展插件吧...