链滴

Go 并发原理

作者：xhaoxiong

原文链接：https://ld246.com/article/1524288601994

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

Go语言是为并发而生的语言，Go语言是为数不多的在语言层面实现并发的语言；也正是Go语言的并特性，吸引了全球无数的开发者。

并发(concurrency)和并行(parallellism)

并发(concurrency)：两个或两个以上的任务在一段时间内被执行。我们不必care这些任务在某一个间点是否是同时执行，可能同时执行，也可能不是，我们只关心在一段时间内，哪怕是很短的时间（秒或者两秒）是否执行解决了两个或两个以上任务。

并行(parallellism)：两个或两个以上的任务在同一时刻被同时执行。

并发说的是逻辑上的概念，而并行，强调的是物理运行状态。并发“包含”并行。

（详情请见：Rob Pike 的<a href="https://talks.golang.org/2012/concurrency.slide#1">PPT</>）

Go的CSP并发模型

Go实现了两种并发形式。第一种是大家普遍认知的：多线程共享内存。其实就是Java或者C++等语中的多线程开发。另外一种是Go语言特有的，也是Go语言推荐的：CSP（communicating sequentia processes）并发模型。

CSP并发模型是在1970年左右提出的概念，属于比较新的概念，不同于传统的多线程通过共享内存来信，CSP讲究的是“以通信的方式来共享内存”。

请记住下面这句话：

Do not communicate by sharing memory; instead, share memory by communicating.

“不要以共享内存的方式来通信，相反，要通过通信来共享内存。”

普通的线程并发模型，就是像Java、C++、或者Python，他们线程间通信都是通过共享内存的方式进行的。非常典型的方式就是，在访问共享数据（例如数组、Map、或者某个结构体或对象）的时候通过锁来访问，因此，在很多时候，衍生出一种方便操作的数据结构，叫做“线程安全的数据结构”例如Java提供的包”java.util.concurrent”中的数据结构。Go中也实现了传统的线程并发模型。

Go的CSP并发模型，是通过**goroutine**和**channel**来实现的。

- goroutine 是Go语言中并发的执行单位。有点抽象，其实就是和传统概念上的”线程“类似，可以解为”线程“。
- channel是Go语言中各个并发结构体(goroutine)之前的通信机制。 通俗的讲，就是各个goroutine间通信的”管道“，有点类似于Linux中的管道。

生成一个goroutine的方式非常的简单：Go一下，就生成了。

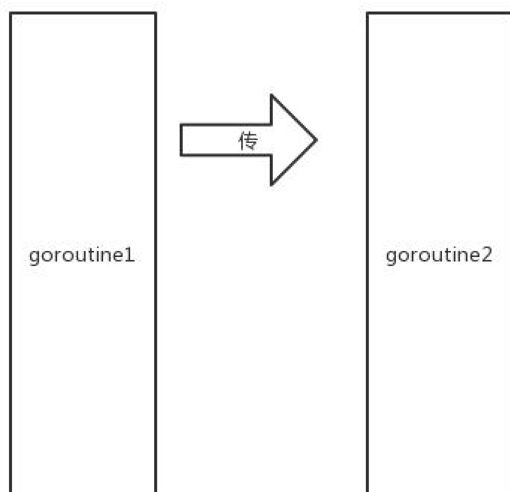
```
go f();
```

通信机制channel也很方便，传数据用channel <- data，取数据用<-channel。

在通信过程中，传数据channel <- data和取数据<-channel必然会成对出现，因为这边传，那边取两个goroutine之间才会实现通信。

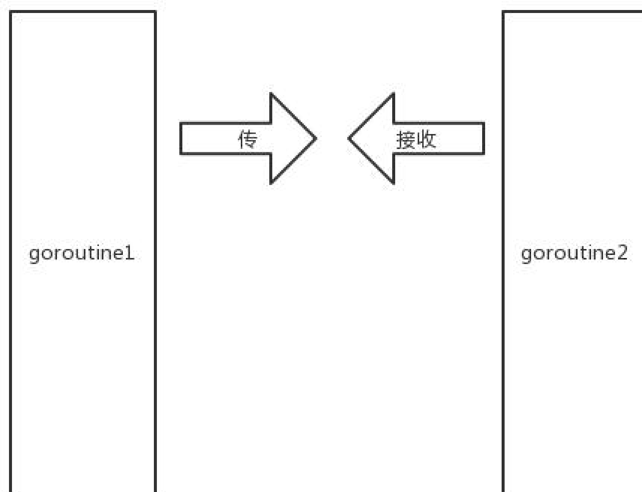
而且不管传还是取，必阻塞，直到另外的goroutine传或者取为止。

有两个goroutine，其中一个发起了向channel中发起了传值操作。（goroutine为矩形，channel为头）



左边的goroutine开始阻塞，等待有人接收。

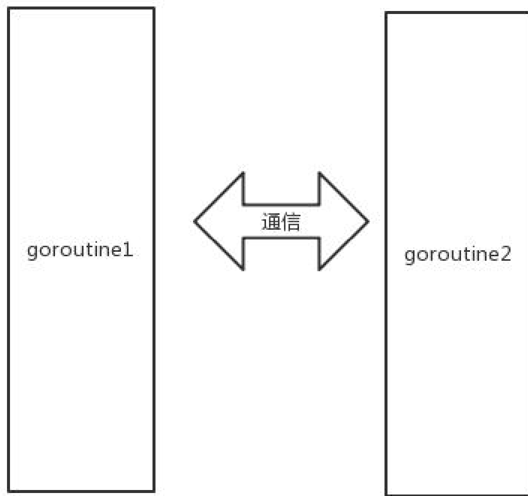
这时候，右边的goroutine发起了接收操作。



右边的goroutine也开始阻塞，等待别人传送。

这时候，两边goroutine都发现了对方，于是两个goroutine开始一传，一收。

原文链接：Go 并发原理



这便是Golang CSP并发模型最基本的形式。

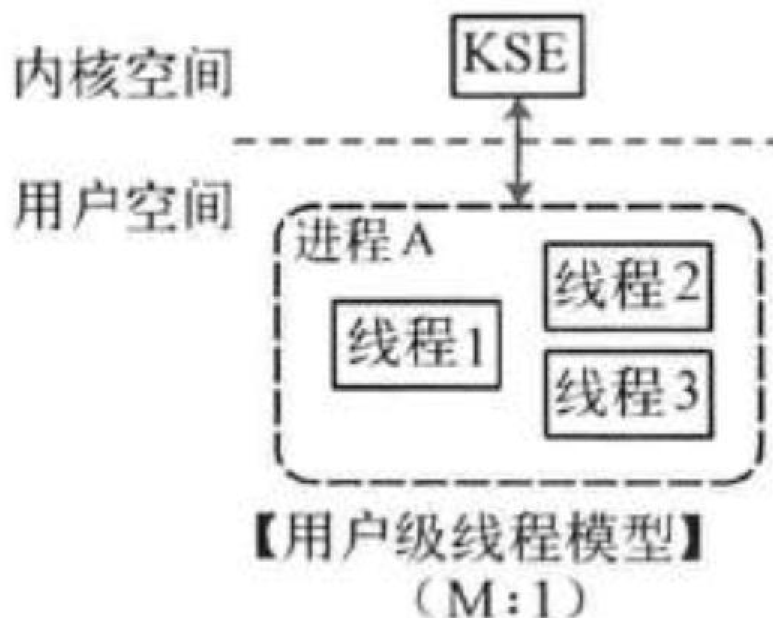
Go并发模型的实现原理

我们先从线程讲起，无论语言层面何种并发模型，到了操作系统层面，一定是以线程的形态存在的。操作系统根据资源访问权限的不同，体系架构可分为用户空间和内核空间；内核空间主要操作访问CP资源、I/O资源、内存资源等硬件资源，为上层应用程序提供最基本的基础资源，用户空间呢就是上应用程序的固定活动空间，用户空间不可以直接访问资源，必须通过“系统调用”、“库函数”或“Sell脚本”来调用内核空间提供的资源。

我们现在的计算机语言，可以狭义的认为是一种“软件”，它们中所谓的“线程”，往往是用户态的程，和操作系统本身内核态的线程（简称KSE），还是有区别的。

线程模型的实现，可以分为以下几种方式：

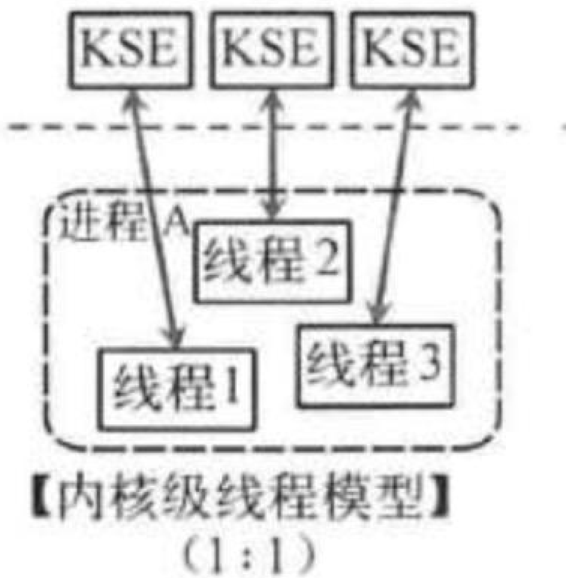
用户级线程模型



原文链接：Go 并发原理

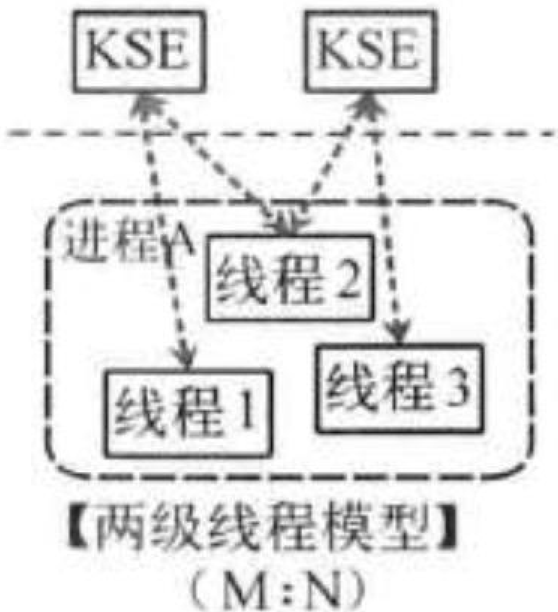
如图所示，多个用户态的线程对应着一个内核线程，程序线程的创建、终止、切换或者同步等线程工必须自身来完成。

内核级线程模型



这种模型直接调用操作系统的内核线程，所有线程的创建、终止、切换、同步等操作，都由内核来完。C++就是这种。

两级线程模型



这种模型是介于用户级线程模型和内核级线程模型之间的一种线程模型。这种模型的实现非常复杂，内核级线程模型类似，一个进程中可以对应多个内核级线程，但是进程中的线程不和内核线程一一对；这种线程模型会先创建多个内核级线程，然后用自身的用户级线程去对应创建的多个内核级线程，

身的用户级线程需要本身程序去调度，内核级的线程交给操作系统内核去调度。

Go语言的线程模型就是一种特殊的两级线程模型。暂且叫它“MPG”模型吧。

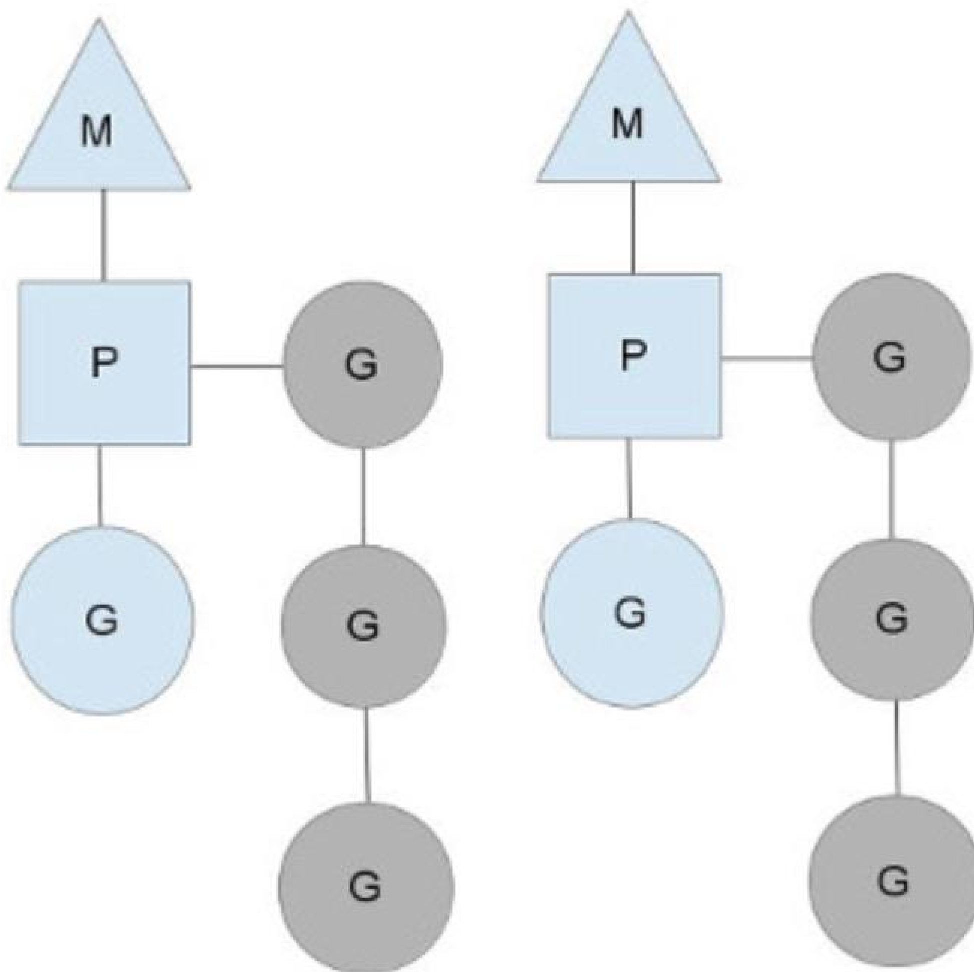
Go线程实现模型MPG

M指的是 Machine，一个M直接关联了一个内核线程。

P指的是 processor，代表了M所需的上下文环境，也是处理用户级代码逻辑的处理器。

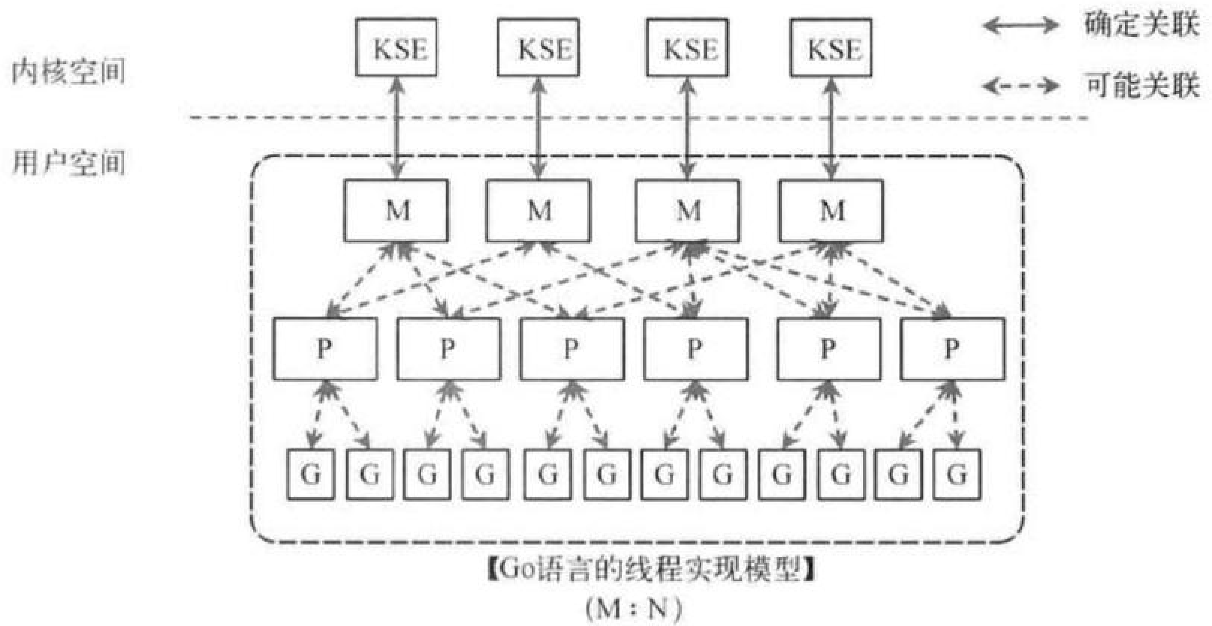
G 指的是Goroutine，其实本质上也是一种轻量级的线程。

三者关系如下图所示：



以上这个图讲的是两个线程(内核线程)的情况。一个M会对应一个内核线程，一个M也会连接一个上文P，一个上下文P相当于一个“处理器”，一个上下文连接一个或者多个Goroutine。P(Processor)数量是在启动时被设置为环境变量GOMAXPROCS的值，或者通过运行时调用函数runtime.GOMAXPROCS()进行设置。Processor数量固定意味着任意时刻只有固定数量的线程在运行go代码。Goroutine就是我们要执行并发的代码。图中P正在执行的Goroutine为蓝色的；处于待执行状态的Goroutine为色的，灰色的Goroutine形成了一个队列runqueues

三者关系的宏观的图为：

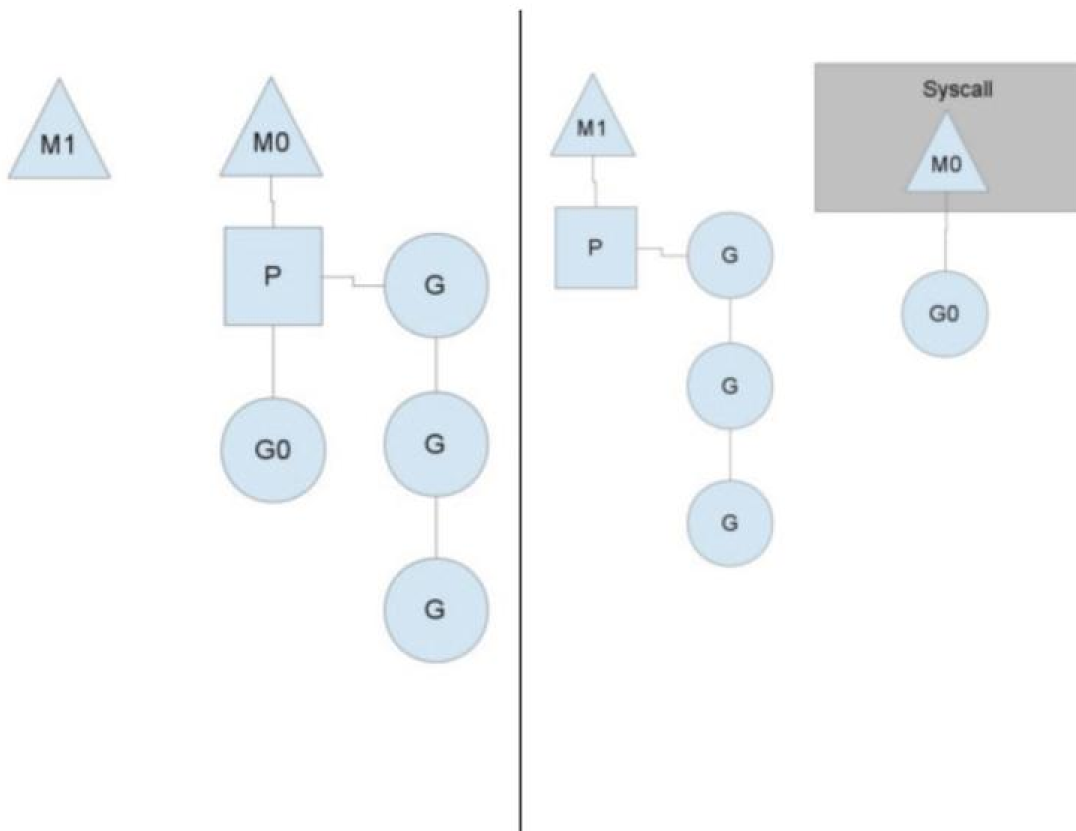


【Go语言的线程实现模型】
（M：N）

抛弃P(Processor)

你可能会想，为什么一定需要一个上下文，我们能不能直接除去上下文，让Goroutine的runqueues到M上呢？答案是不行，需要上下文的目的，是让我们可以直接放开其他线程，当遇到内核线程阻塞时候。

一个很简单的例子就是系统调用sysall，一个线程肯定不能同时执行代码和系统调用被阻塞，这个时，此线程M需要放弃当前的上下文环境P，以便可以让其他的Goroutine被调度执行。



原文链接：Go 并发原理

如上图左图所示，M0中的G0执行了syscall，然后就创建了一个M1(也有可能本身就存在，没创建)，转向右图）然后M0丢弃了P，等待syscall的返回值，M1接受了P，将·继续执行Goroutine队列中的他Goroutine。

当系统调用syscall结束后，M0会“偷”一个上下文，如果不成功，M0就把它的Gouroutine G0放到个全局的runqueue中，然后自己放到线程池或者转入休眠状态。全局runqueue是各个P在运行完自的本地的Goroutine runqueue后用来拉取新goroutine的地方。P也会周期性的检查这个全局runqueue上的goroutine，否则，全局runqueue上的goroutines可能得不到执行而饿死。

均衡的分配工作

按照以上的说法，上下文P会定期的检查全局的goroutine 队列中的goroutine，以便自己在消费掉自身oroutine队列的时候有事可做。假如全局goroutine队列中的goroutine也没了呢？就从其他运行的的P的runqueue里偷。

每个P中的Goroutine不同导致他们运行的效率和时间也不同，在一个有很多P和M的环境中，不能让个P跑完自身的Goroutine就没事可做了，因为或许其他的P有很长的goroutine队列要跑，得需要均。

该如何解决呢?

Go的做法倒也直接，从其他P中偷一半!

参考文献

RyuGou的客

The Go scheduler