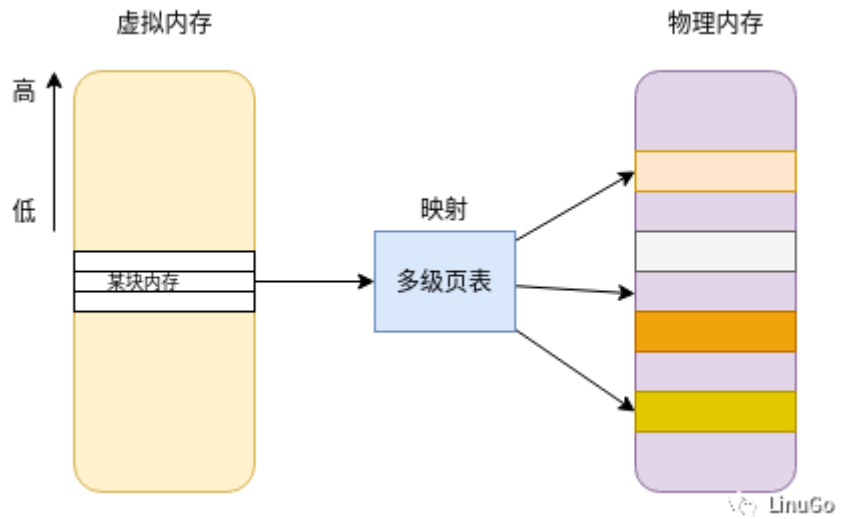
从进程谈起

进程与线程的区别是什么？这是一个老生长谈的一道面试题。处于不同层面对该问题的理解也大不相同。对于用户层面来说，进程就是一块运行起来的程序，线程就是程序里的一些并发的功能。对于操作系统层面来说，标准回答是“进程是资源分配的最小单位，线程是cpu调度的最小单位”。接下来先从操作系统层面介绍一下进程与线程。

进程

在程序启动时，操作系统会给该程序分配一块内存空间，对于程序但看到的是一整块连续的内存空间，称为虚拟内存空间，落实到操作系统内核则是一块一块的内存碎片的东西。为的是节省内核空间，方便对内存管理。



就这片内存空间，又划分为用户空间与内核空间，用户空间只用于用户程序的执行，若要执行各种IO操作，就会通过系统调用等进入内核空间进行操作。每个进程都有自己的PID，可以通过ps命令查看某个进程的pid，进入/proc/可以查看该进程的详细信息，如cgroup，进程资源大小等信息。

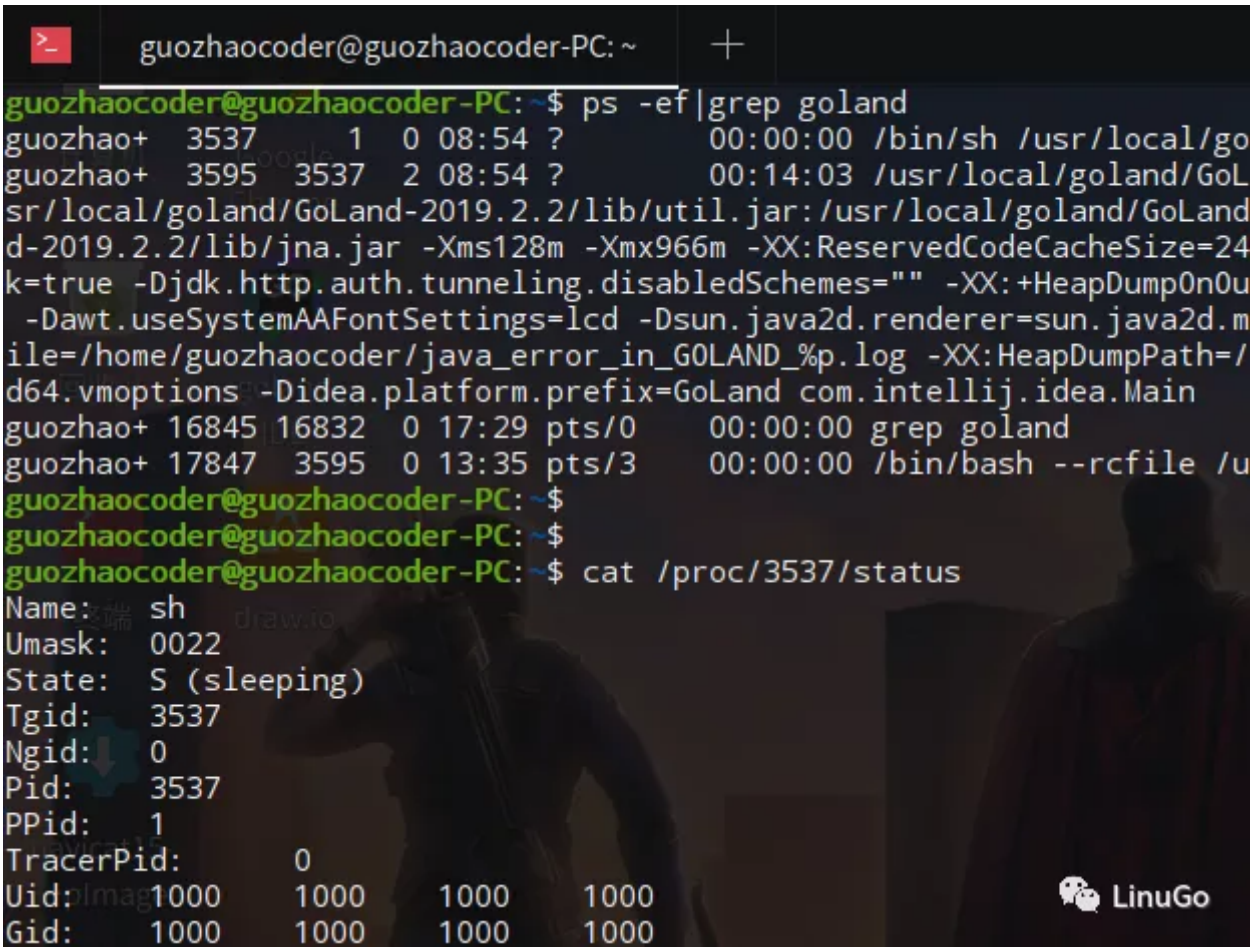


image.png

线程

线程是进程的一个执行单元，一个进程可以包含多个线程，只有拥有了线程的进程才会被CPU执行，所以一个进程最少拥有一个主线程。

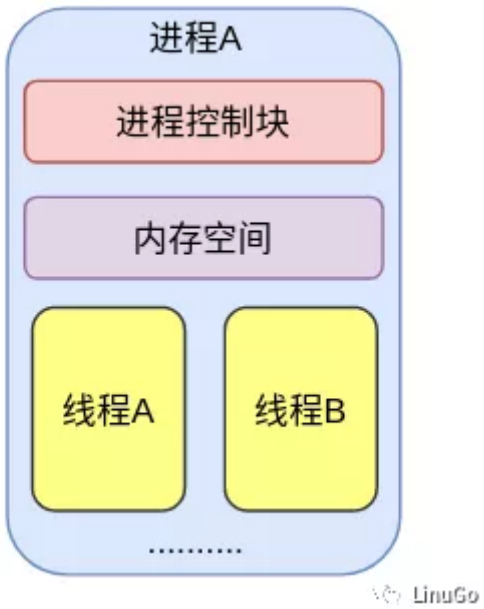


image.png

由于多个线程可以共享同一个进程的内存空间，线程的创建不需要额外的虚拟内存空间，线程之间的切换也就少了如进程切换的切换页表，切换虚拟地址空间此类的巨大开销。至于进程切换为什么较

大，简单理解是因为进程切换要保存的现场太多如寄存器，栈，代码段，执行位置等，而线程切换只需要上下文切换，保存线程执行的上下文即可。线程的的切换只需要保存线程的执行现场(程序计数器等状态)保存在该线程的栈里，CPU把栈指针，指令寄存器的值指向下一个线程。相比之下线程更加轻量级。
可以说进程面向的主要内容是内存分配管理，而线程主要面向的CPU调度。

协程

虽然线程比进程要轻量级，但是每个线程依然占有1M左右的空间，在高并发场景下非常吃机器内存，比如构建一个http服务器，如果一个每来一次请求分配一个线程，请求数暴增容易OOM，而且线程切换的开销也是不可忽视的。同时，线程的创建与销毁同样是比较大的系统开销，因为是由内核来做的，解决方法也有，可以通过线程池或协程来解决。
协程是用户态的线程，比线程更加的轻量级，操作系统对其没有感知，之所以没有感知是由于协程处于线程的用户栈能感知的范围，是由用户创建的而非操作系统。

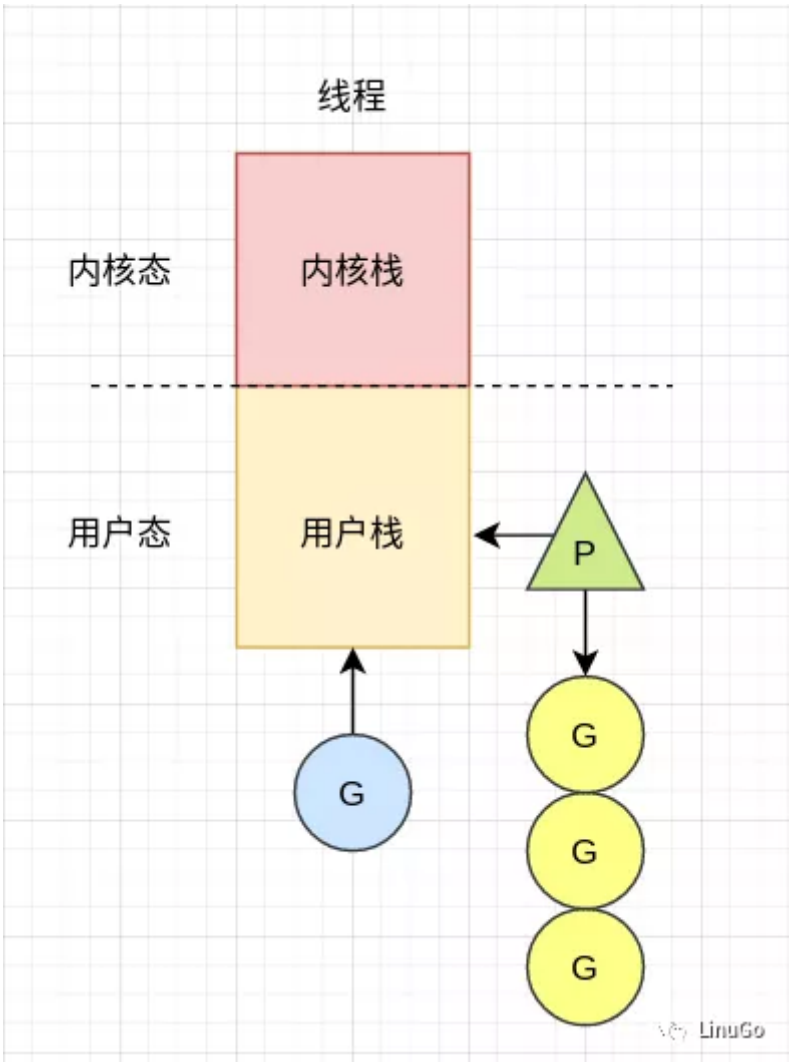


image.png

如一个进程可拥有以有多个线程一样，一个线程也可以拥有多个协程。协程之于线程如同线程之于cpu，拥有自己的协程队列，每个协程拥有自己的栈空间，在协程切换时候只需要保存协程的上下文，开销要比内核态的线程切换要小很多。

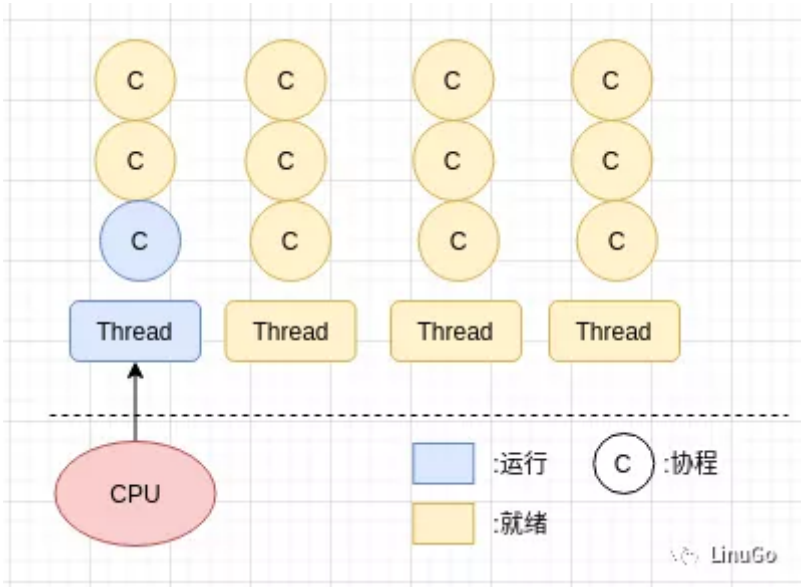


image.png

GMP模型

含义

Goroutine的并发编程模型基于GMP模型，简要解释一下GMP的含义：

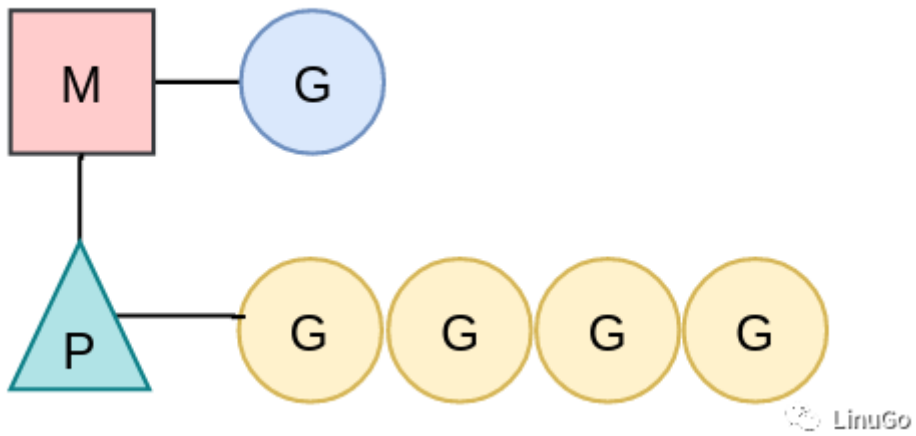
G:表示goroutine，每个goroutine都有自己的栈空间，定时器，初始化的栈空间在2k左右，空间会随着需求增长。

M:抽象化代表内核线程，记录内核线程栈信息，当goroutine调度到线程时，使用该goroutine自己的栈信息。

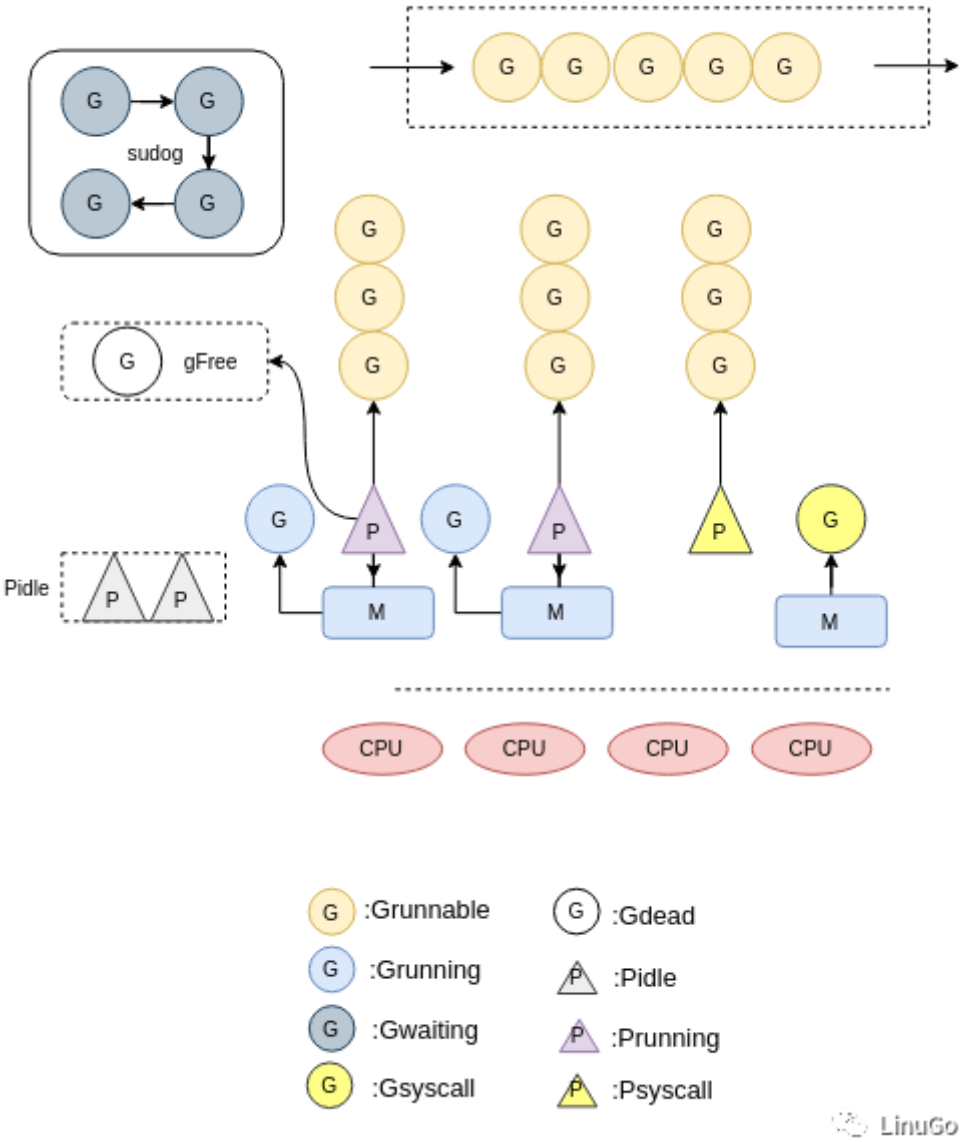
P:代表调度器，负责调度goroutine，维护一个本地goroutine队列，M从P上获得goroutine并执行，同时还负责部分内存的管理。

模型

从大体看一下GMP模型。



M代表一个工作线程，在M上有一个P和G，P是绑定到M上的，G是通过P的调度获取的，在某一时刻，一个M上只有一个G（g0除外）。在P上拥有一个G队列，里面是已经就绪的G，是可以被调度到线程栈上执行的协程，称为运行队列。
接下来看一下程序中GMP的分布。



每个进程都有一个全局的G队列，也拥有P的本地执行队列，同时也有不在运行队列中的G。如正处于channel的阻塞状态的G，还有脱离P绑定在M的(系统调用)G，还有执行结束后进入P的gFree列表中的G等等，接下来列举一下常见的几种状态。

状态汇总

G状态

G的主要几种状态：

本文基于Go1.13，具体代码见（<GOROOT>/src/runtime/runtime2.go）

_Gidle：刚刚被分配并且还没有被初始化，值为0，为创建goroutine后的默认值

_Grunnable： 没有执行代码，没有栈的所有权，存储在运行队列中，可能在某个P的本地队列或全局队列中(如上图)。

_Grunning： 正在执行代码的goroutine，拥有栈的所有权(如上图)。

_Gsyscall：正在执行系统调用，拥有栈的所有权，与P脱离，但是与某个M绑定，会在调用结束后被分配到运行队列(如上图)。

_Gwaiting：被阻塞的goroutine，阻塞在某个channel的发送或者接收队列(如上图)。

_Gdead： 当前goroutine未被使用，没有执行代码，可能有分配的栈，分布在空闲列表gFree，可能是一个刚刚初始化的goroutine，也可能是执行了goexit退出的goroutine(如上图)。

_Gcopystac：栈正在被拷贝，没有执行代码，不在运行队列上，执行权在

_Gscan ： GC 正在扫描栈空间，没有执行代码，可以与其他状态同时存在

P的状态

_Pidle ： 处理器没有运行用户代码或者调度器，被空闲队列或者改变其状态的结构持有，运行队列为空

_Prunning ： 被线程 M 持有，并且正在执行用户代码或者调度器(如上图)

_Psyscall：没有执行用户代码，当前线程陷入系统调用(如上图)

_Pgcstop ： 被线程 M 持有，当前处理器由于垃圾回收被停止

_Pdead ： 当前处理器已经不被使用

M的状态

自旋线程：处于运行状态但是没有可执行goroutine的线程(如下图)，数量最多为GOMAXPROC，若是数量大于GOMAXPROC就会进入休眠。

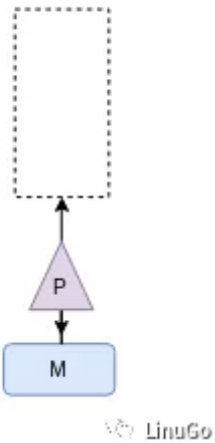


image.png

非自旋线程：处于运行状态有可执行goroutine的线程。

调度场景

Channel阻塞：当goroutine读写channel发生阻塞时候，会调用gopark函数，该G会脱离当前的M与P，调度器会执行schedule函数调度新的G到当前M。可参考上一篇文章channel探秘。

系统调用：当某个G由于系统调用陷入内核态时，该P就会脱离当前的M，此时P会更新自己的状态为Psyscall，M与G互相绑定，进行系统调用。结束以后若该P状态还是Psyscall，则直接关联该M和G，

否则使用闲置的处理器处理该G。

系统监控：当某个G在P上运行的时间超过10ms时候，或者P处于Psyscall状态过长等情况就会调用retake函数，触发新的调度。

主动让出：由于是协作式调度，该G会主动让出当前的P，更新状态为Grunnable，该P会调度队列中的G运行。

总结

Go语言中通过GMP模型实现了对CPU和内存的合理利用，使得用户在不用担心内存的情况下体验到线程的好处。虽说协程的空间很小，但是也需要关注一下协程的生命周期，防止过多的协程滞留造成OOM。