Go调度器中的三种结构G、P、M

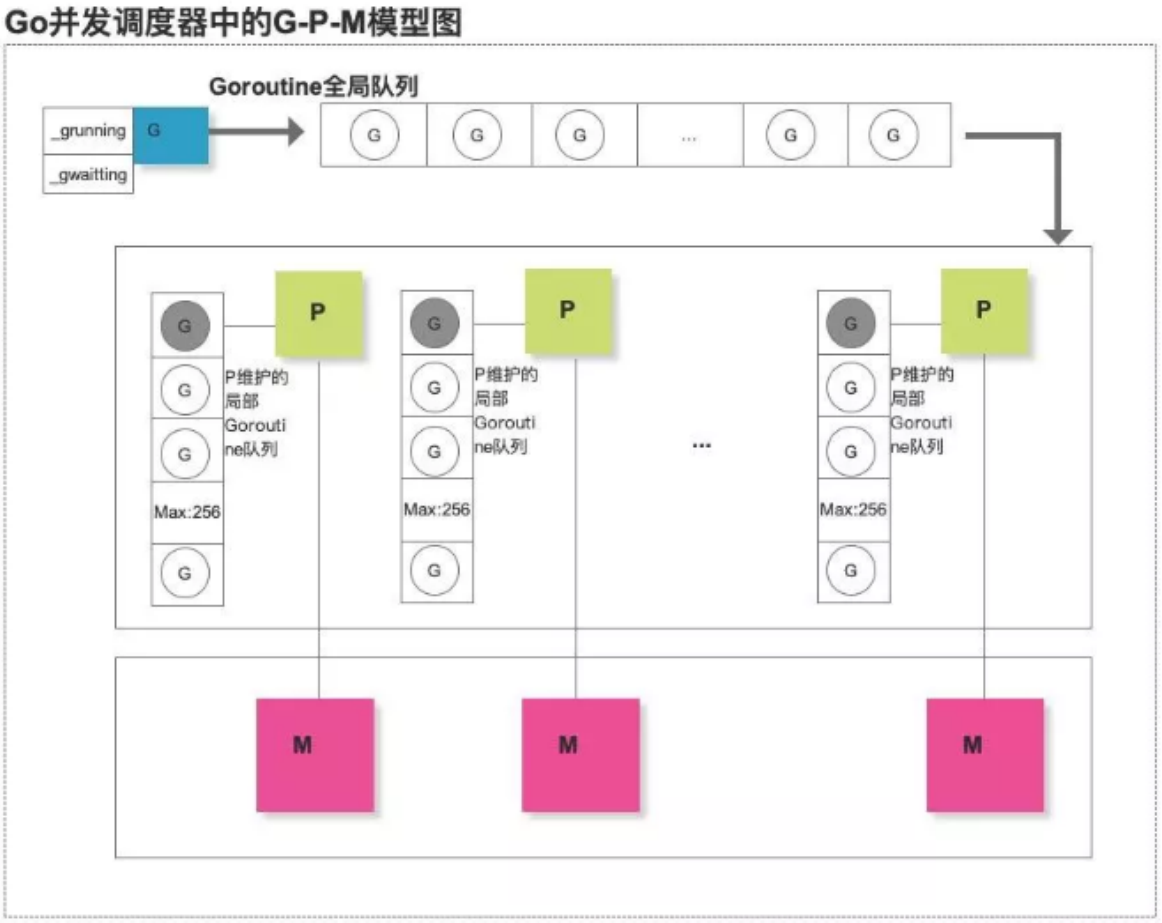
系统线程固定2M，且维护一堆上下文，对需求多变的并发应用并不友好，有可能造成内存浪费或内存不够用。Go将并发的单位下降到线程以下，由其设计的goroutine初始空间非常小，仅2kb，但支持动态扩容到最大1G，这就是go自己的并发单元——goroutine协程。

实际上系统最小的执行单元仍然是线程，go运行时执行的协程也是挂载到某一系统线程之上的，这种协程与系统线程的调度分配由Go的并发调度器承担，Go的并发调度器是属于混合的二级调度并发模型，其内部设计有G、P、M三种抽象结构，我们来看一下它们分别是什么：

G-P-M模型抽象结构：

- G: 表示Goroutine，每个Goroutine对应一个G结构体，G存储Goroutine的运行堆栈、状态以及任务函数，可重用。G运行队列是一个栈结构，分全局队列和P绑定的局部队列，每个G不能独立运行，它需要绑定到P才能被调度执行。
- P: Processor，表示逻辑处理器， 对G来说，P相当于CPU核，G只有绑定到P(在P的local runq中)才能被调度。对M来说，P提供了相关的执行环境(Context)，如内存分配状态(mcache)，任务队列(G)等，P的数量决定了系统内最大可并行的G的数量（前提：物理CPU核数 >= P的数量），P的数量由用户设置的GOMAXPROCS决定，但是不论GOMAXPROCS设置为多大，P的数量最大为256。
- M: Machine，系统物理线程，代表着真正执行计算的资源，在绑定有效的P后，进入schedule循环；而schedule循环的机制大致是从Global队列、P的Local队列以及wait队列中获取G，切换到G的执行栈上并执行G的函数，调用goexit做清理工作并回到M，如此反复。M并不保留G状态，这是G可以跨M调度的基础，M的数量是不定的，由Go Runtime调整，为了防止创建过多OS线程导致系统调度不过来，目前默认最大限制为10000个。

关于P这个设计，是在Go1.0之后才实现的，起初的Go并发性能并不十分亮眼，协程和系统线程的调度比较粗暴，导致很多性能问题，如全局资源锁、M的内存过高等造成许多性能损耗，加入P的设计后实现了一个叫做 work-stealing 的调度算法：由P来维护Goroutine队列并选择一个适当的M绑定



Go并发调度器的GPM模型.jpg

G-P-M模型调度

我们来看看go关键字创建一个协程后其调度器是怎么工作的：

- go关键字创建goroutine(G)，优先加入某个P维护的局部队列（当局部队列已满时才加入全局队列）；
- P需要持有或者绑定一个M，而M会启动一个系统线程，不断的从P的本地队列取出G并执行；
- M执行完P维护的局部队列后，它会尝试从全局队列寻找G，如果全局队列为空，则从其他的P维护的队列里窃取一般的G到自己的队列；
- 重复以上知道所有的G执行完毕。

当然也有一些情况会造成Goroutine阻塞，如：

- 系统GC；
- 系统IO资源的调用，如文件读写；
- 网络IO的延迟；
- 管道阻塞；
- 同步操作。

当遇到上述阻塞时，Go调度器也有相应的处理方式：

- 1.系统调度引起阻塞：

如系统GC，M会解绑P，出让控制权给其他M，让该P维护的G运行队列不至于阻塞。

- 2.用户态的阻塞：

当goroutine因为管道操作或者系统IO、网络IO而阻塞时，对应的G会被放置到某个等待队列，该G的状态由运行时变为等待状态，而M会跳过该G尝试获取并执行下一个G，如果此时没有可运行的G供M运行，那么M将解绑P，并进入休眠状态；当阻塞的G被另一端的G2唤醒时，如管道通知，G又被标记为可运行状态，尝试加入G2所在P局部队列的队头，然后再是G全局队列。

- 3.当存在空闲的P时，窃取其他队列的G：

当P维护的局部队列全部运行完毕，它会尝试在全局队列获取G，直到全局队列为空，再向其他局部队列窃取一般的G。

至此Go的调度器模型解析完毕。基于Go调度器的优越设计，它号称能实现百万级并发，即使日常很难达到这种并发量，我们也应该对并发的使用要心存敬畏，真正的并发依赖于物理核心，启动并发是需要系统开销的，虽然在Go的运行时它看起来很小，但量变引起质变，当业务启动的并发到十万级、百万级甚至千万级时，其性能开销还是非常巨大的。可以通过一定的手段控制并发数量以防止系统奔溃，如实现一个协程池，通过worker机制控制并发数。

Ok，希望学完这一专题你会对Go的并发有更深刻的了解。