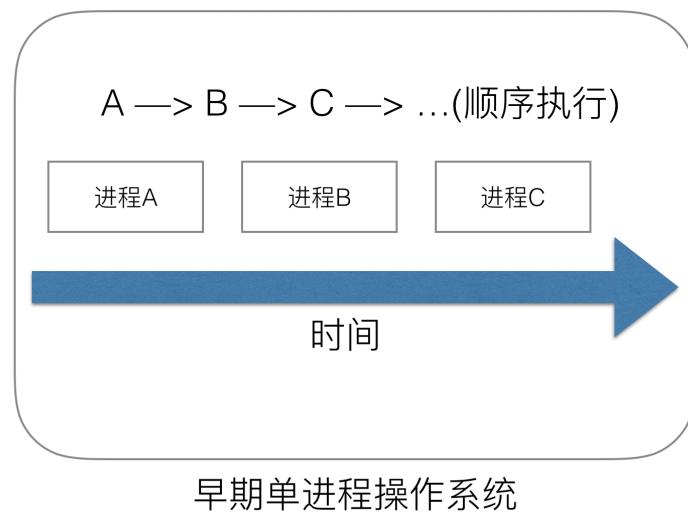


GMP原理

- 1.Golang协程调度器的由来
- 2.GMP设计思想
- 3.Go调度场景解析

一、协程的发展

(1) 单进程时代



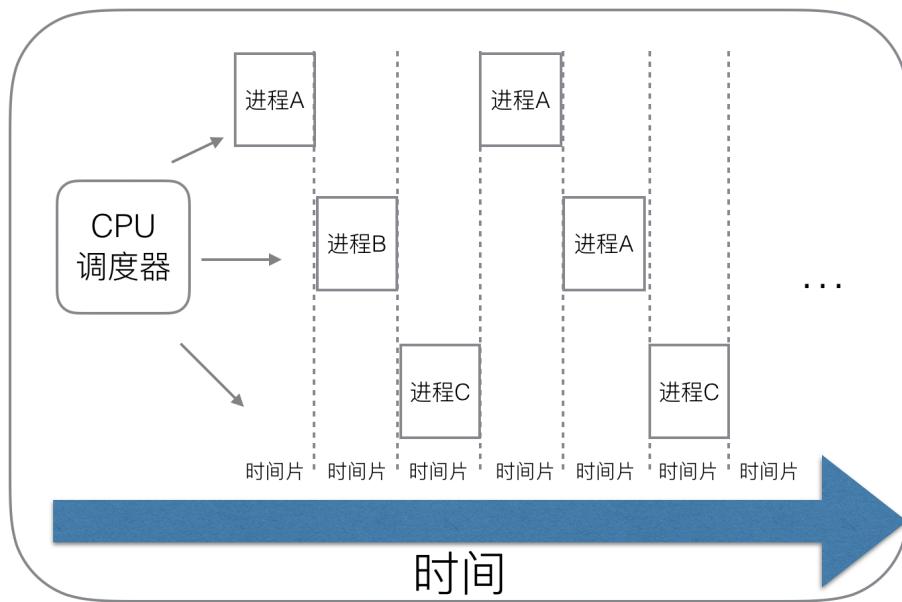
特点:一切程序只能串行发生

问题:

1. 单一的流程，任务只能一个一个执行
2. 任务阻塞所带来的的cpu资源的浪费

(2) 多进程/多线程

A ->
B ->
C ->
(并发执行)



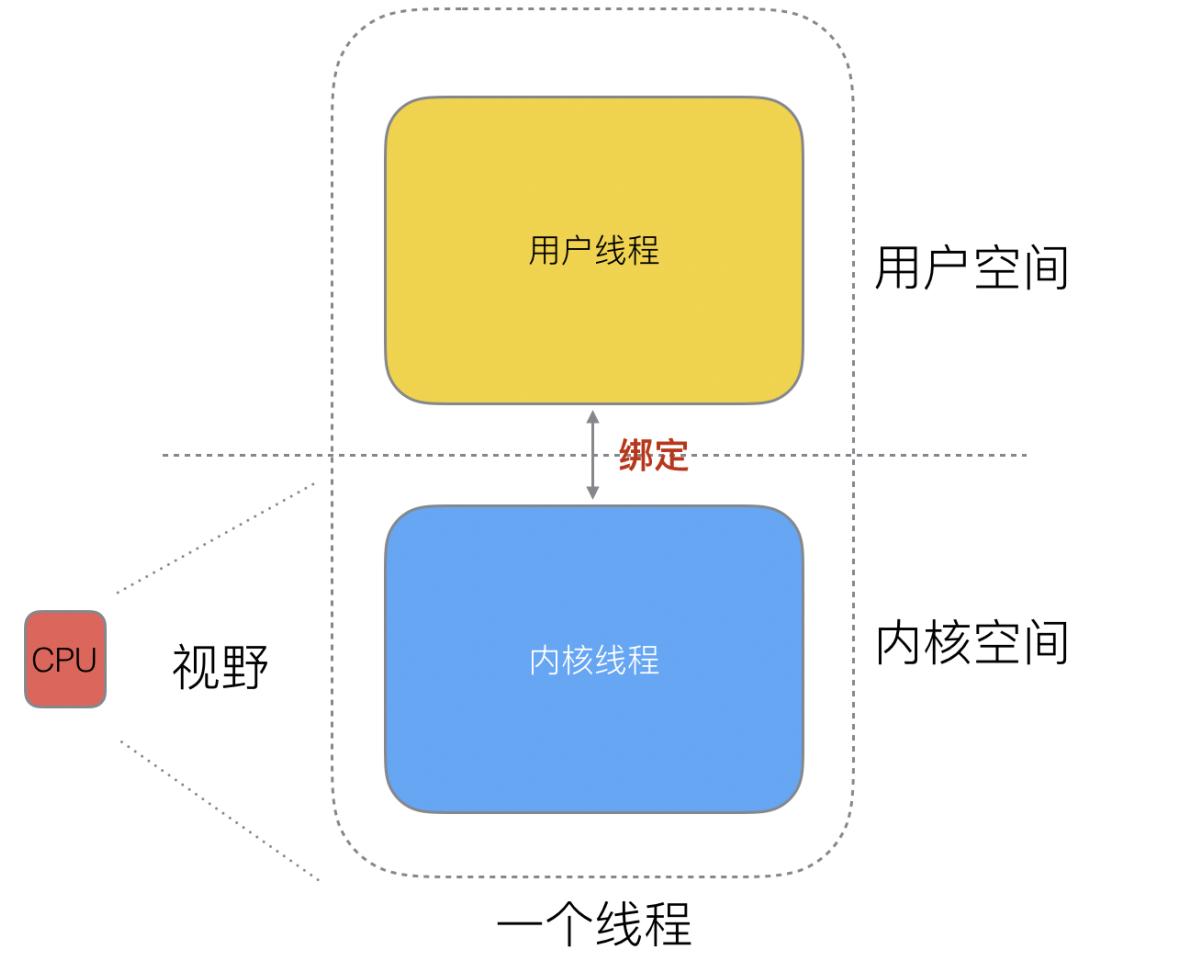
多线程/多进程操作系统

解决:阻塞问题，一个进程被阻塞，cpu可以切换到其他进程执行，调度cpu的算法可以保证进程能够获得cpu的时间片,宏观上来看，似乎是多个进程同时运行

新问题: 在当今互联网高并发的场景下，为每个任务创建一个进程/线程是不现实的，会消耗大量内存，进程虚拟内存会占用4G(32位)，线程也需要大约1M(至少MB级别)

- 高内存占用
- 调度的高消耗CPU

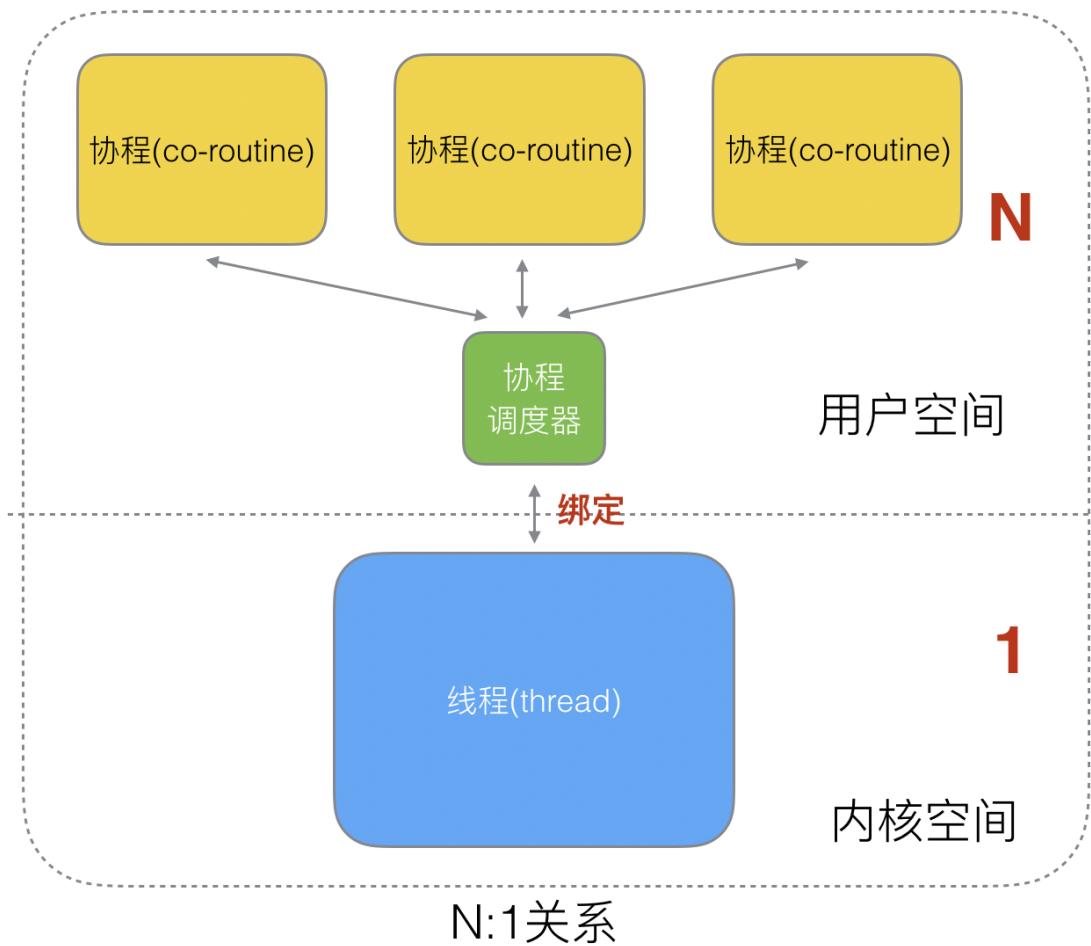
(3) 协程



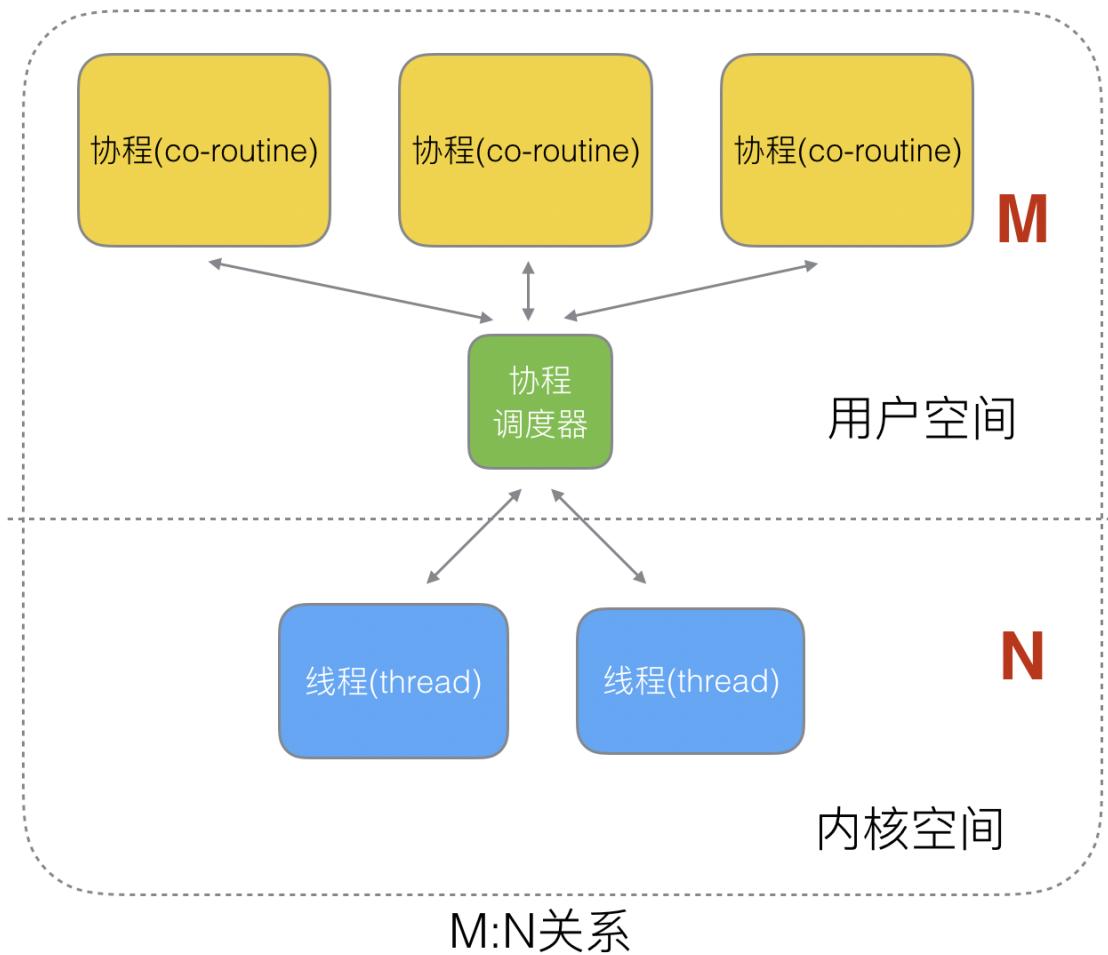
线程分为两部分:

- 内核态线程
- 用户态线程(协程)

N:1



M:N关系



特点:

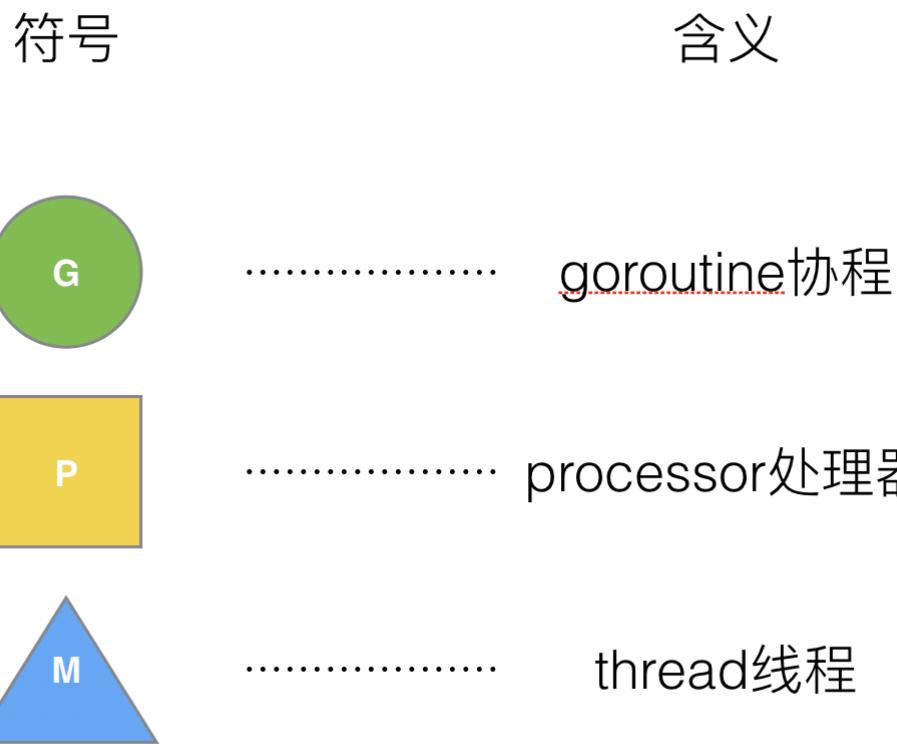
- 占用内存更小(几kb)
- 调度更灵活(runtime调度)

Go语言调度器的版本迭代 (目前go最新版本 1.15.6)

- 单线程调度器 ·0.x
 - 只包含 40 多行代码;
 - 程序中只能存在一个活跃线程, 由 G-M 模型组成;
- 多线程调度器 ·1.0
 - 允许运行多线程的程序;
 - 全局锁导致竞争严重;
- 任务窃取调度器 ·1.1
 - 引入了处理器 P, 构成了目前的 **G-M-P** 模型;
 - 在处理器 P 的基础上实现了基于**工作窃取**的调度器;
 - 在某些情况下, Goroutine 不会让出线程, 进而造成饥饿问题;
 - 时间过长的垃圾回收 (Stop-the-world, STW) 会导致程序长时间无法工作;
- 抢占式调度器 ·1.2~ 至今
 - 基于协作的抢占式调度器 - 1.2 ~ 1.13
 - 通过编译器在函数调用时插入**抢占检查**指令, 在函数调用时检查当前 Goroutine 是否发起了抢占请求, 实现基于协作的抢占式调度;
 - Goroutine 可能会因为垃圾回收和循环长时间占用资源导致程序暂停;
 - 基于信号的抢占式调度器 - 1.14 ~ 至今
 - 实现**基于信号的真抢占式调度**;
 - 垃圾回收在扫描栈时会触发抢占调度;
 - 抢占的时间点不够多, 还不能覆盖全部的边缘情况;
- 非均匀存储访问调度器 · 提案
 - 对运行时的各种资源进行分区;
 - 实现非常复杂, 到今天还没有提上日程;

二、GMP模型设计思想

Go最新版的调度器包含G、M、P



数据结构: runtime.g(40多个字段)

```
type g struct {
    stack          stack
    stackguard0 uintptr
    preempt       bool // 抢占信号
    preemptStop   bool // 抢占时将状态修改成 `_Gpreempted`
    preemptShrink bool // 在同步安全点收缩栈
    _panic        *_panic // 最内侧的 panic 结构体
    _defer         *_defer // 最内侧的延迟函数结构体
    m             *m
    sched         gobuf
    atomicstatus  uint32 // 存储了当前 Goroutine 的状态
    goid         int64
}
```

G常见状态 `Grunnable`、`Grunning`、`Gsyscall`、`Gwaiting`、`_Gpreempted`

GOROUTINE STATUS

Draveness

waiting

runnable

running

- 等待中: Goroutine 正在等待某些条件满足, 例如: 系统调用结束等, 包括 `_Gwaiting`、`_Gsyscall` 和 `_Gpreempted` 几个状态;

- 可运行: Goroutine 已经准备就绪, 可以在线程运行, 如果当前程序中有非常多的 Goroutine, 每个 Goroutine 就可能会等待更多的时间, 即 `_Grunnable`;
- 运行中: Goroutine 正在某个线程上运行, 即 `_Grunning`;

| 数据结构: runtime.m (包含了62个字段)

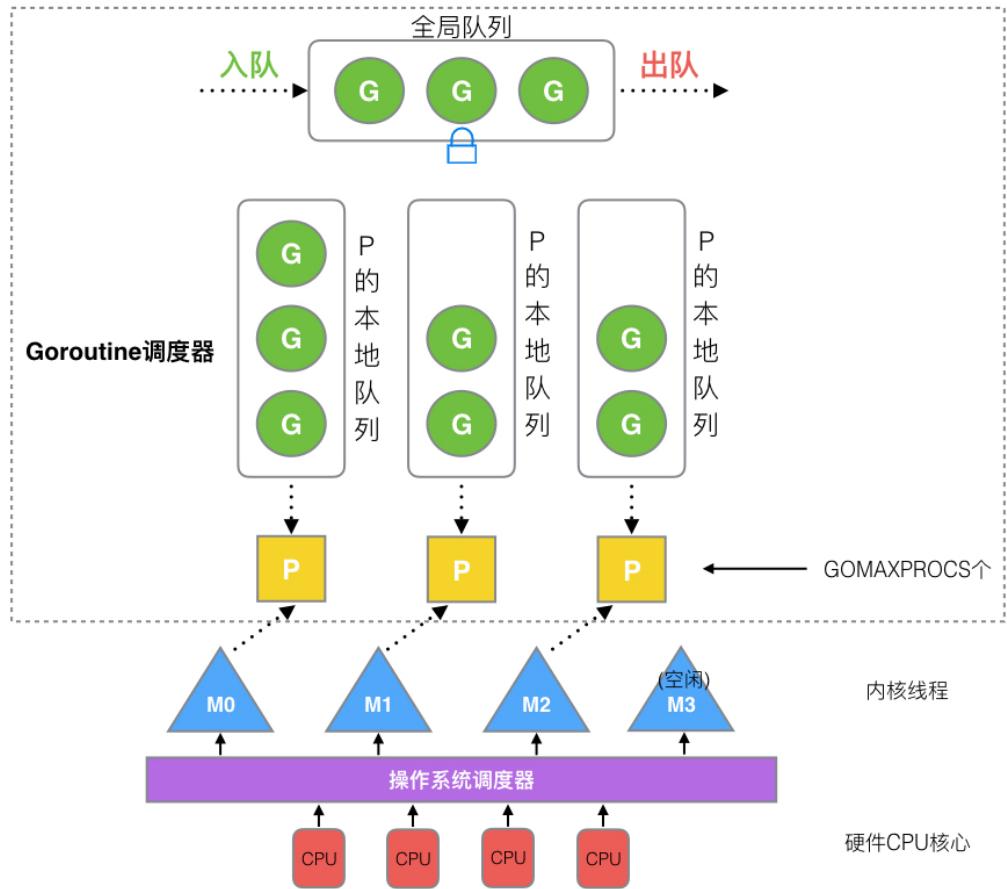
```
type m struct {
    g0    *g   //持有调度栈的goroutine,每启动一个m就是第一个创建的goroutine
    curg *g   //当前线程上运行的用户goroutine
    p     *uintptr //正在运行代码的处理器 p
    nextp      uintptr //暂存的处理器
    oldp      uintptr //行系统调用之前的使用线程的处理器
}
```

| 数据结构 runtime.p

```
type p struct {
    m        muintptr
    status   uint32 // _Pidle|_Prunning|_Psyscall|_Pgcstop|_Pdead
    runqhead uint32
    runqtail uint32
    runq    [256]guintptr
    runnext guintptr
    ...
}
```

状态	描述
<code>_Pidle</code>	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
<code>_Prunning</code>	被线程 M 持有, 并且正在执行用户代码或者调度器
<code>_Psyscall</code>	没有执行用户代码, 当前线程陷入系统调用
<code>_Pgcstop</code>	被线程 M 持有, 当前处理器由于垃圾回收被停止
<code>_Pdead</code>	当前处理器已经不被使用

(1) GMP模型



(2) 调度器的设计策略

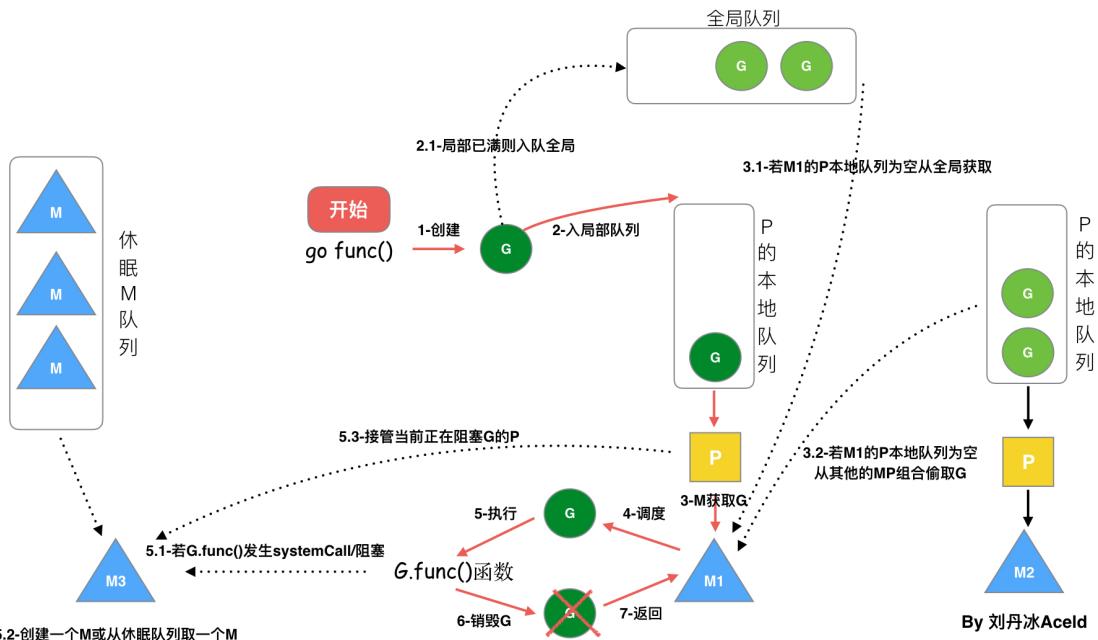
1. work stealing机制

当本线程无可运行的G时，会从其他线程绑定的P的本地队列中偷取G

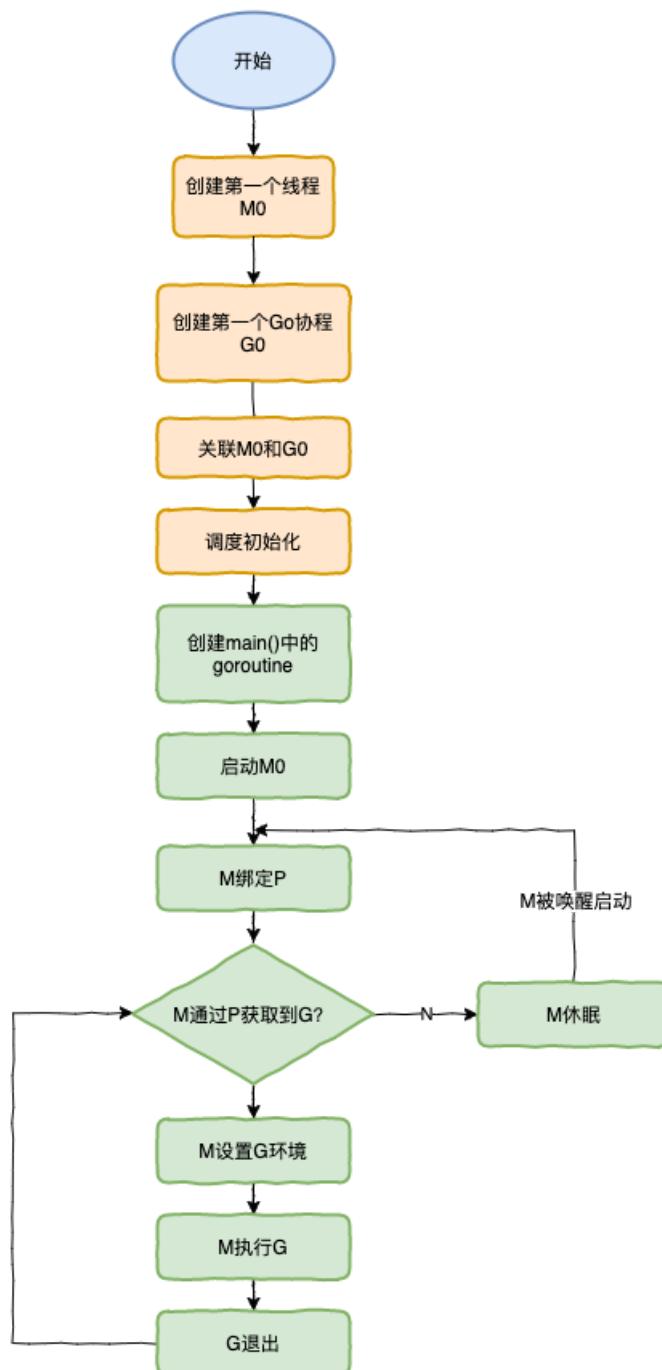
2. hand off 机制

当本线程因为系统调用出现阻塞时，线程释放绑定的P，将P转移给其他空闲的线程执行

(3) go func的调度流程



(4) 调度器的生命周期



1. M0

M0是启动程序后编号为0的主线程,M对应的实例会在全局变量runtime.m0中,不需要在heap上分配,M0负责执行初始化的操作和启动第一个G,之后M0和其他M就一样了

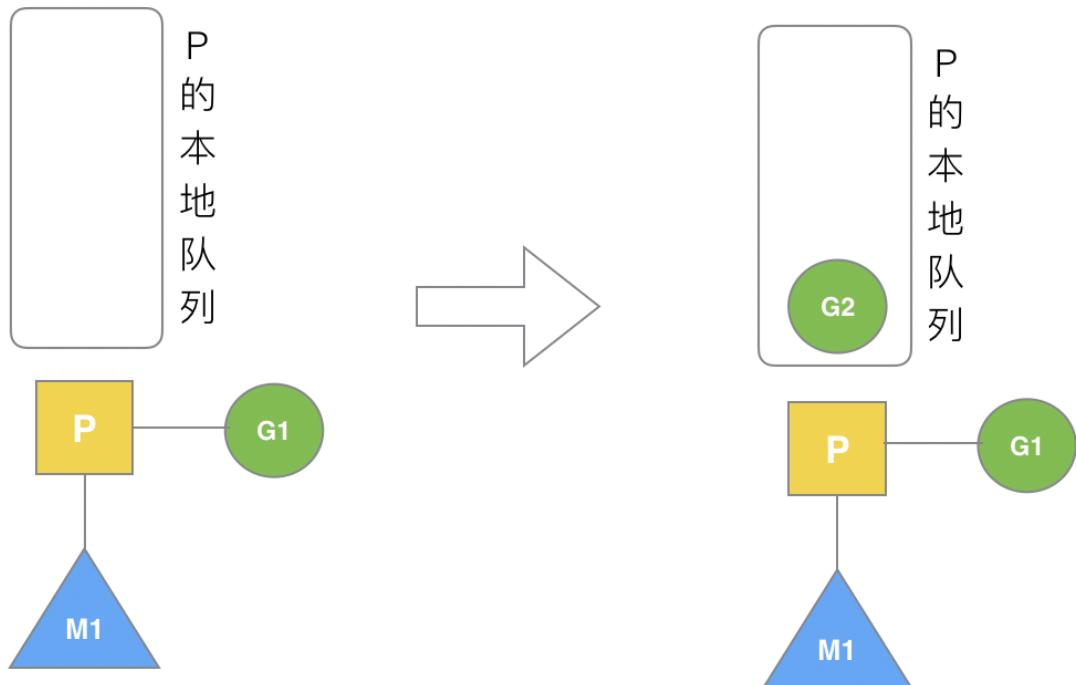
2. G0

G0是每次启动一个M都会第一个创建的goroutine, G0仅用于负责调度G,G0不指向任何可执行的函数,每个M都会有一个属于自己的G0,全局变量的G0是M0的G0

三、Go调度器调度场景过程解析

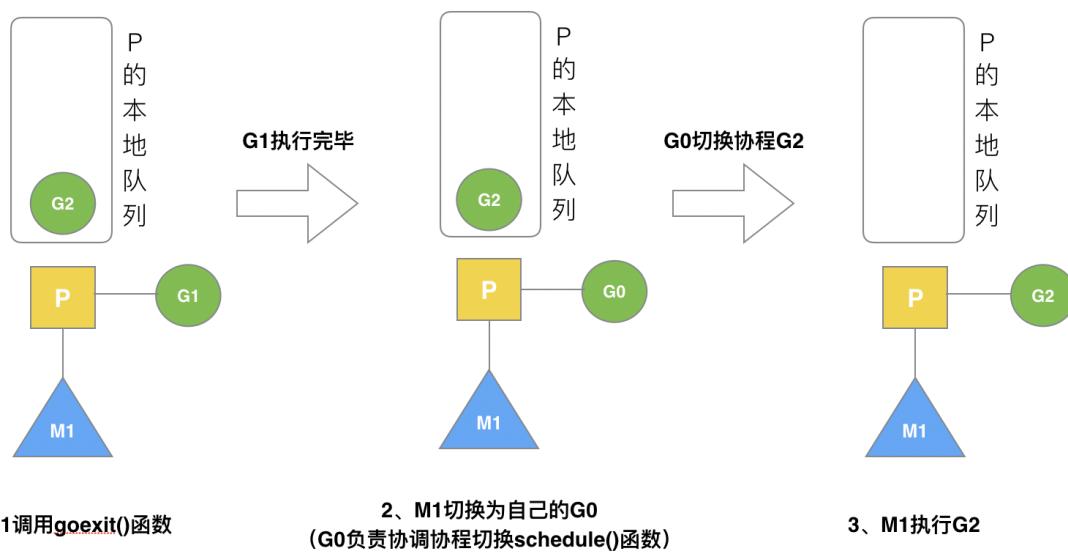
(1)

场景1：G1创建G2



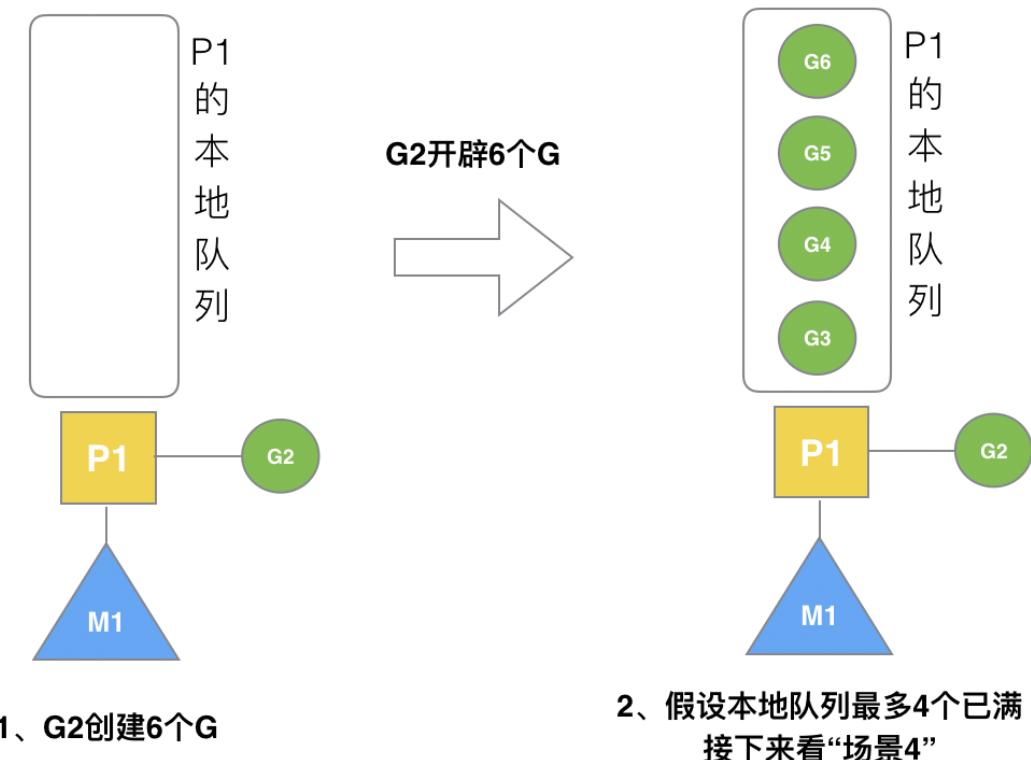
(2)

场景2：G1执行完毕



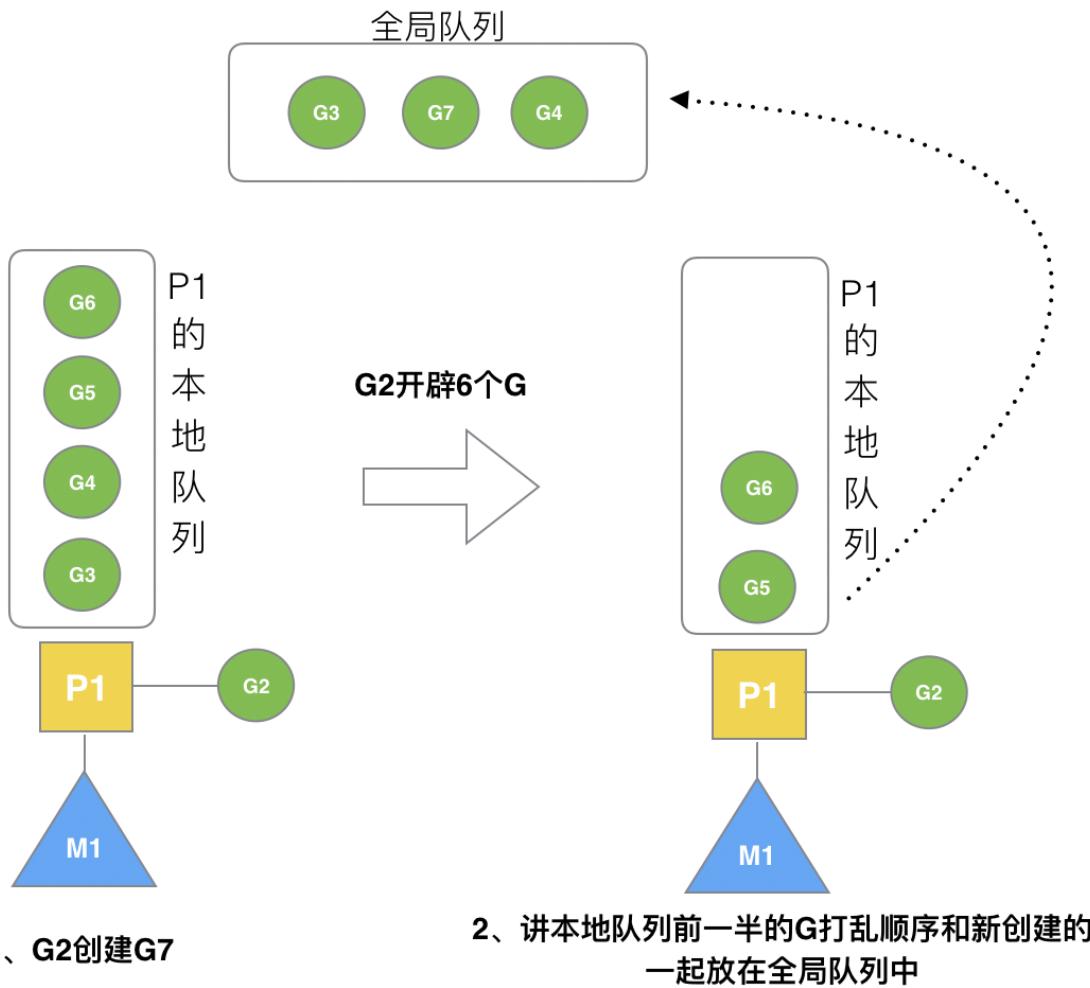
(3)

场景3：G2开辟过多的G



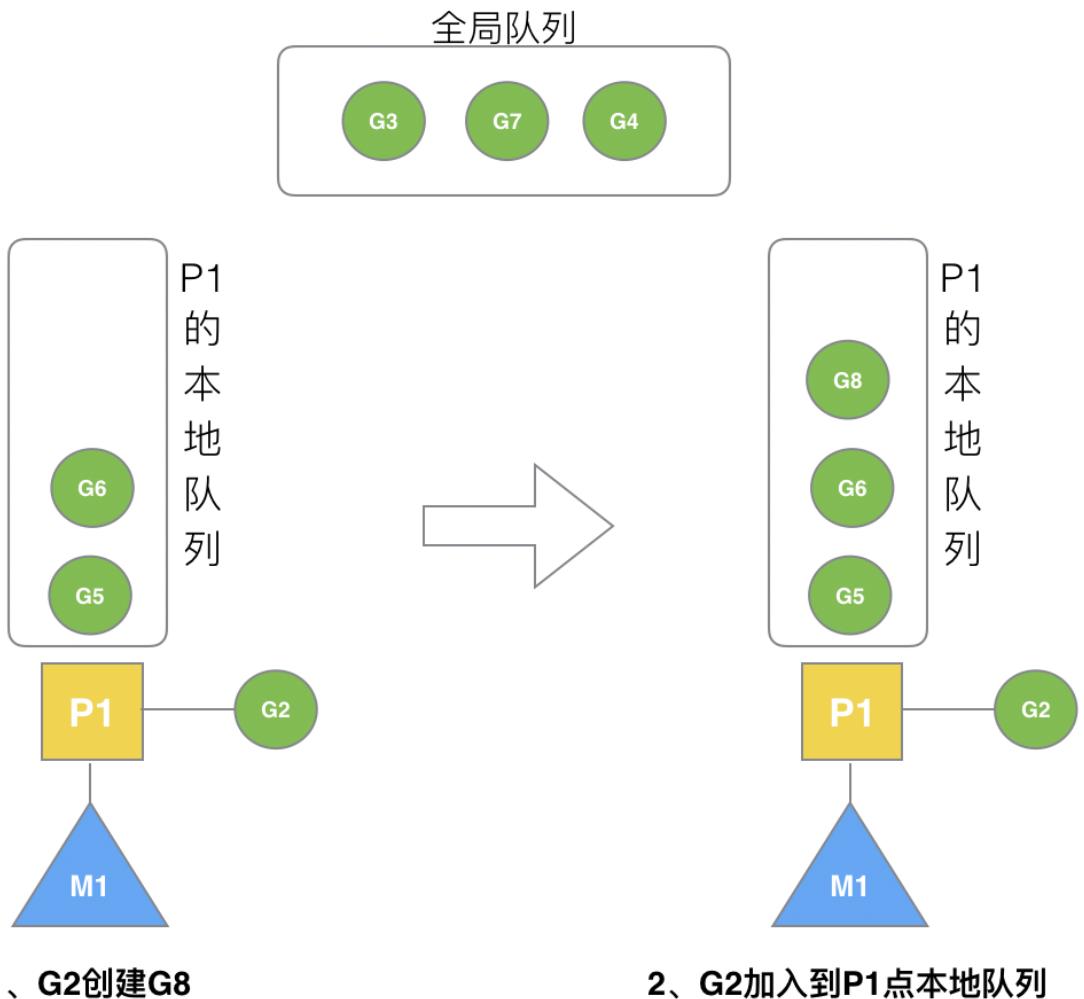
(4)

场景4：G2本地满再创建G7



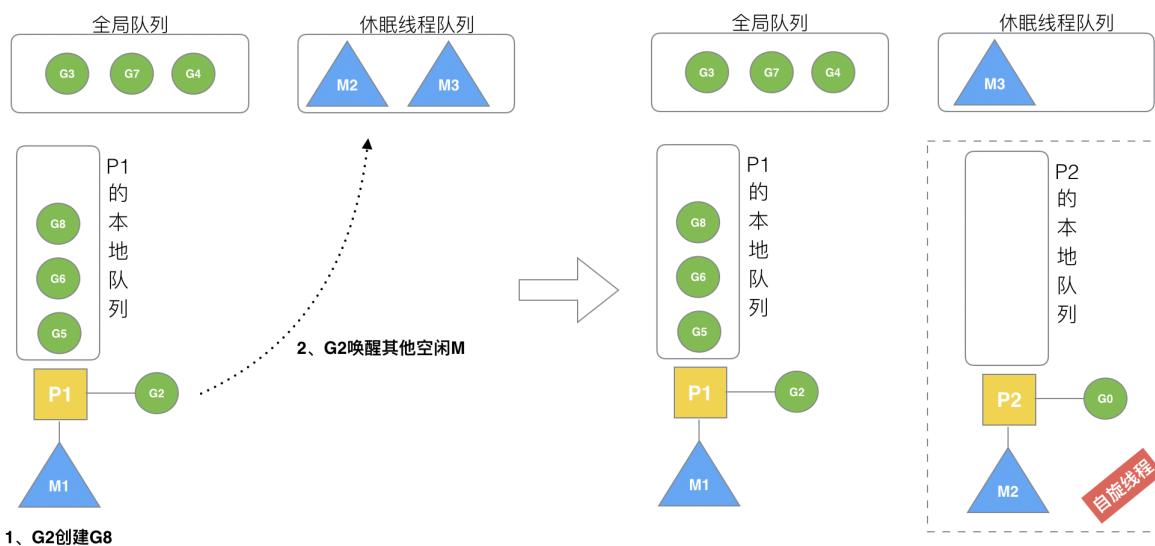
(5)

场景5：G2本地未满创建G8

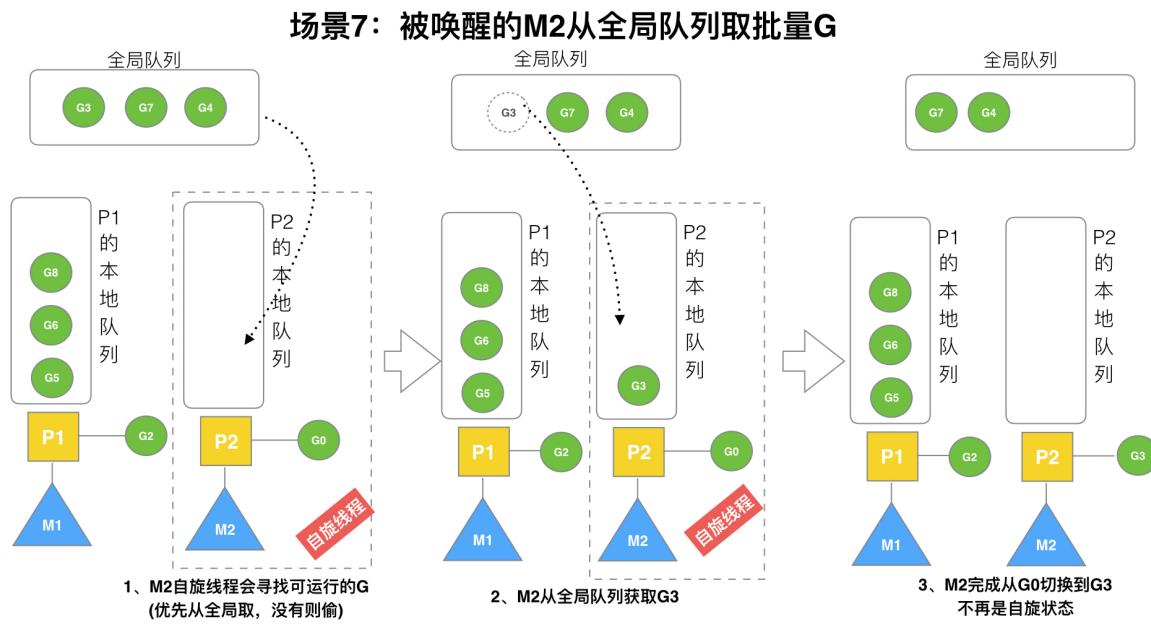


(6)

场景6：唤醒正在休眠的M

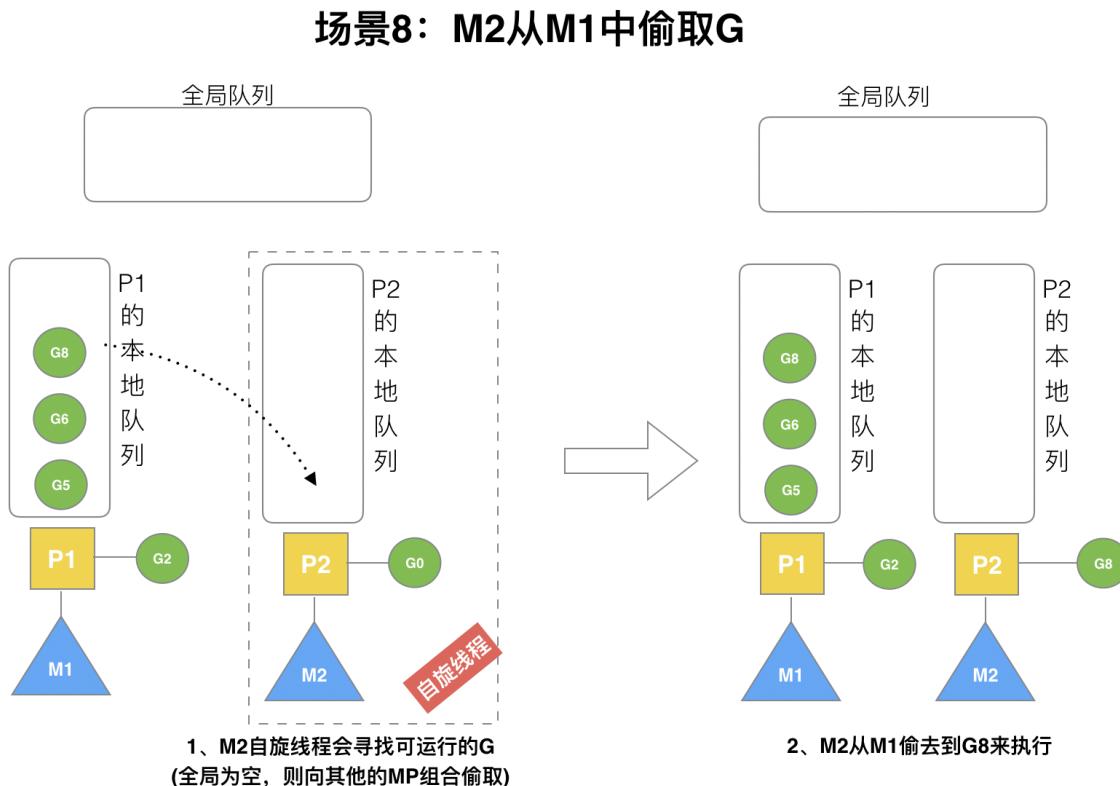


(7)



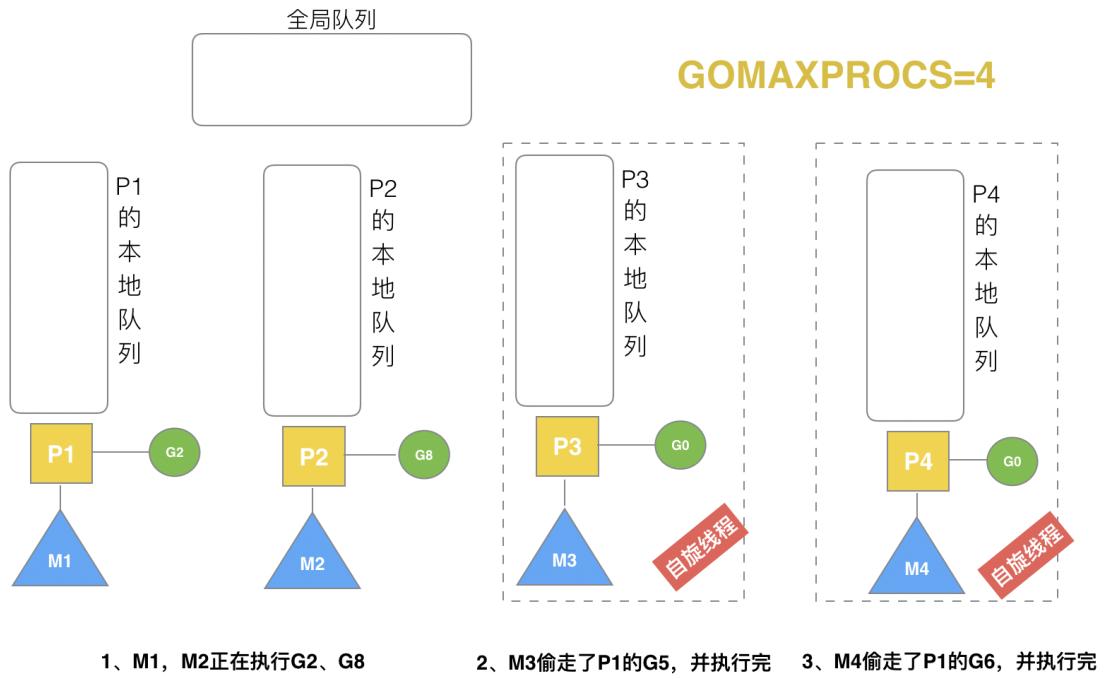
$$n = \min(\lceil \text{len}(GQ)/\text{GOMAXPROCS} + 1, \text{len}(GQ)/2 \rceil)$$

(8)



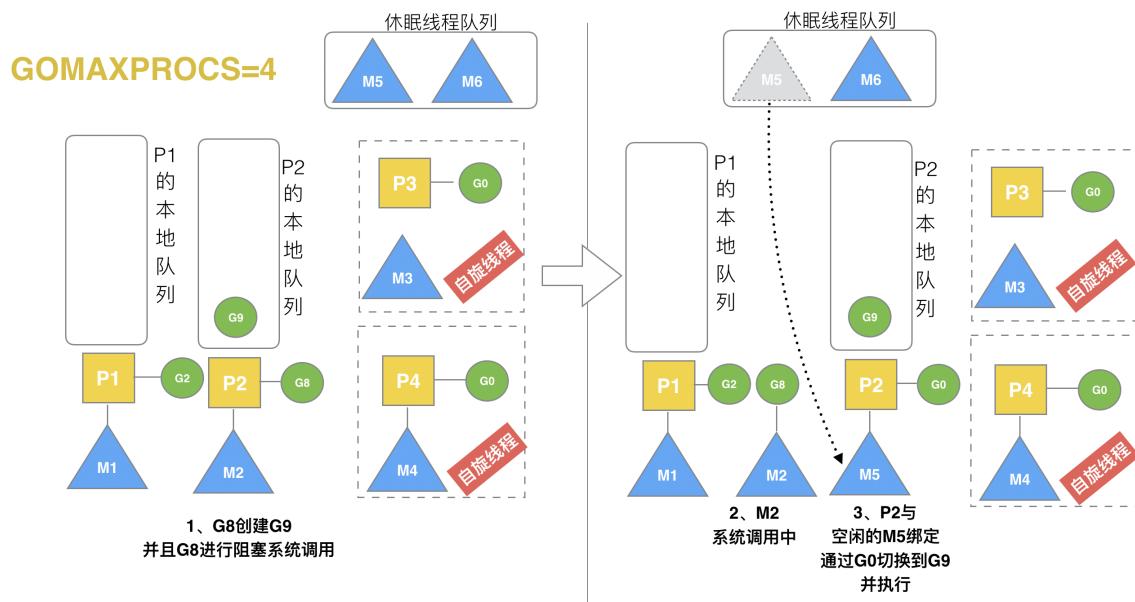
(9)

场景9：自旋线程的最大限制



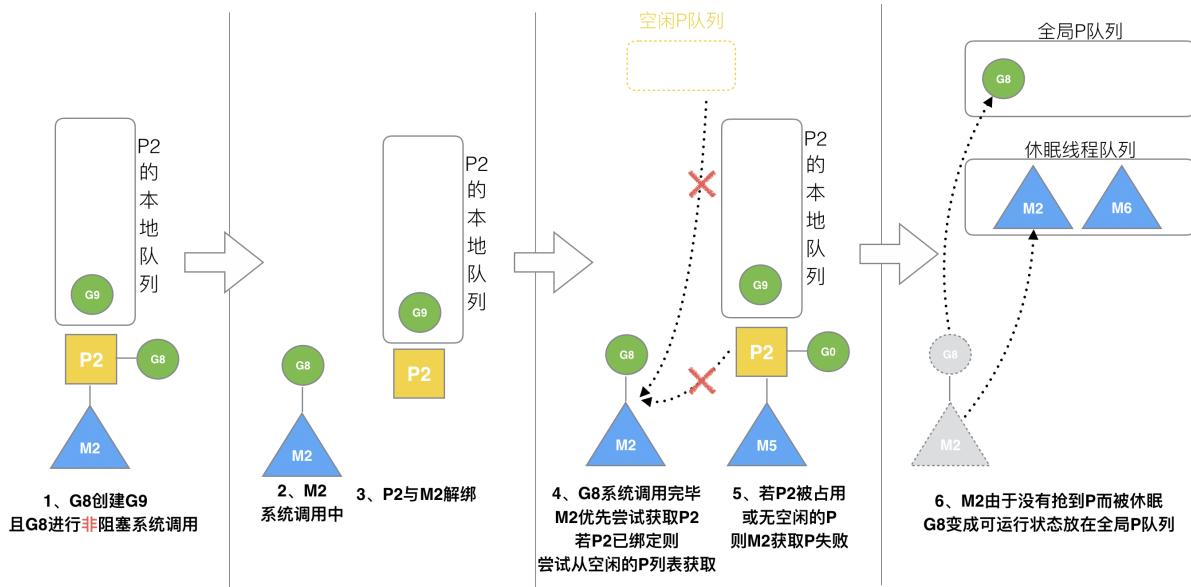
(10)

场景10：G发生系统调用/阻塞



(11)

场景11：G发生系统调用/非阻塞



本质：Go调度的本质是把大量的goroutine分配到少量的线程上去执行，并利用多核并行，实现更强大的并发。

参考资料：

Go修养之路

Go的设计与实现