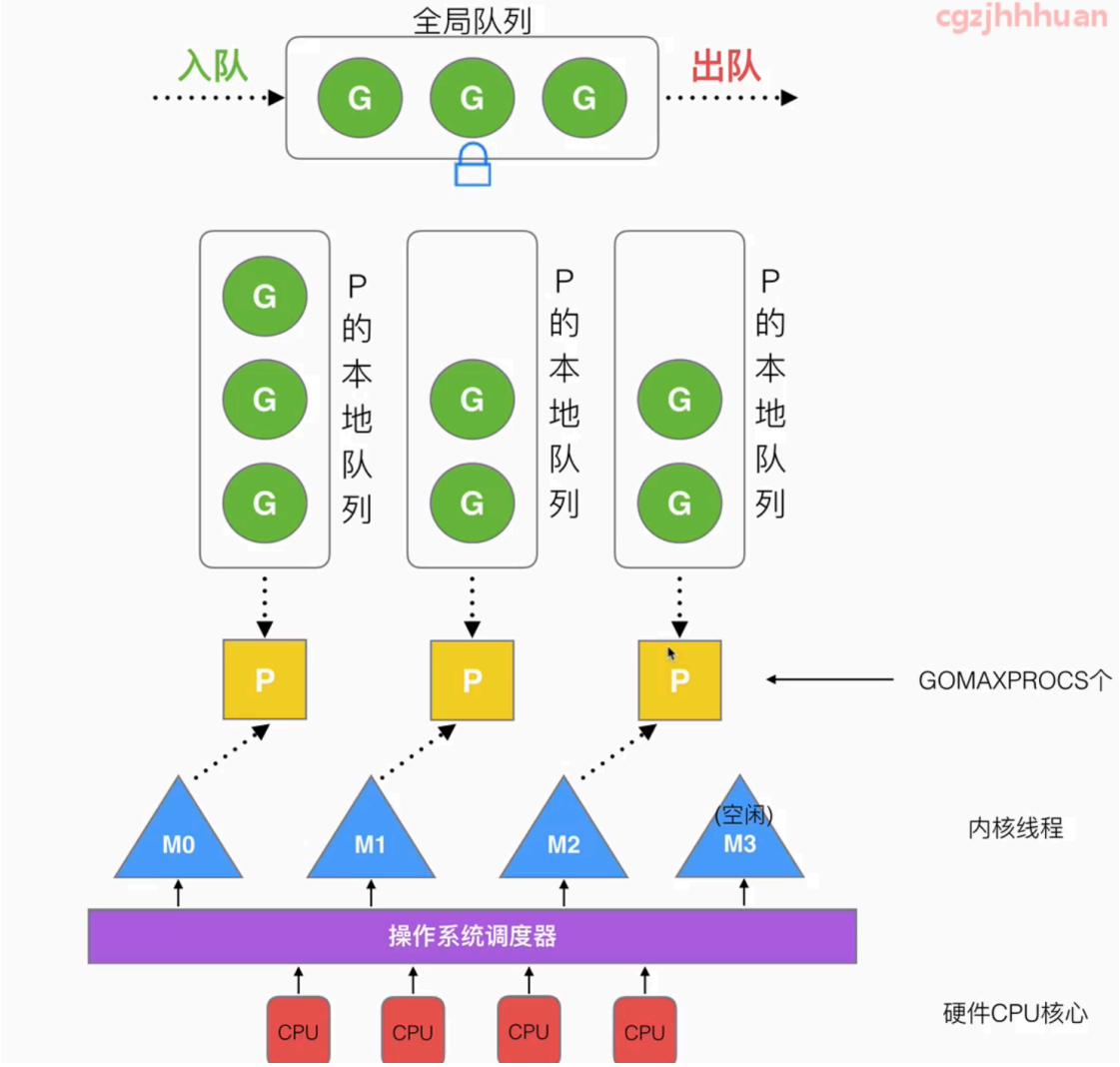
一、goroutine调度器的GMP模型的设计思想

1.GMP模型的简介

- G:goroutine协程
- P:processer（虚拟处理器，包含了可运行的goroutine队列）
- M:内核线程（需要绑定P才可以运行G）

如下图所示，有一个全局队列存放等待运行的G,每个P都有自己的本地队列存放等待运行的G，数量不超过256。每个G创建过后会先放到某个P的本地队列，如果满了就会把G放到全局队列。P的数量是固定的，程序启动时创建，有GOMAXPROCS个（环境变量配置或runtime提供了函数设置，一般默认和cpu的逻辑线程数量一致）。M的数量是不固定的，语言本身限制M的数量最大为10000，可通过runtime/debug包中SetMaxThreads函数来设置。M的数量是动态变化的，如果有M处于阻塞状态，调度器会创建一个新的M或者调度处于休眠状态的M来运行G。如果有M空闲，调度器会回收或者使其休眠。



2.调度器得设计策略

2.1 复用线程

复用线程采取两种机制：

- work stealing机制：当某个P的本地队列为空，与之绑定的M处于空闲状态时，这个P会从其他P中偷取一半的G来运行。
- hand off机制：某个M执行G时处于阻塞状态（这个阻塞是同步阻塞），就会把当前阻塞的M和G与P解绑，创建或唤醒一个新的M绑定P继续执行G（最大限度的压榨cpu）。

2.2 利用并行

可以通过设置GOMAXPROCS参数来设置P的数量，达到控制cpu利用率的目的，可以空出部分cpu执行其他进程的任务。

2.3 抢占机制

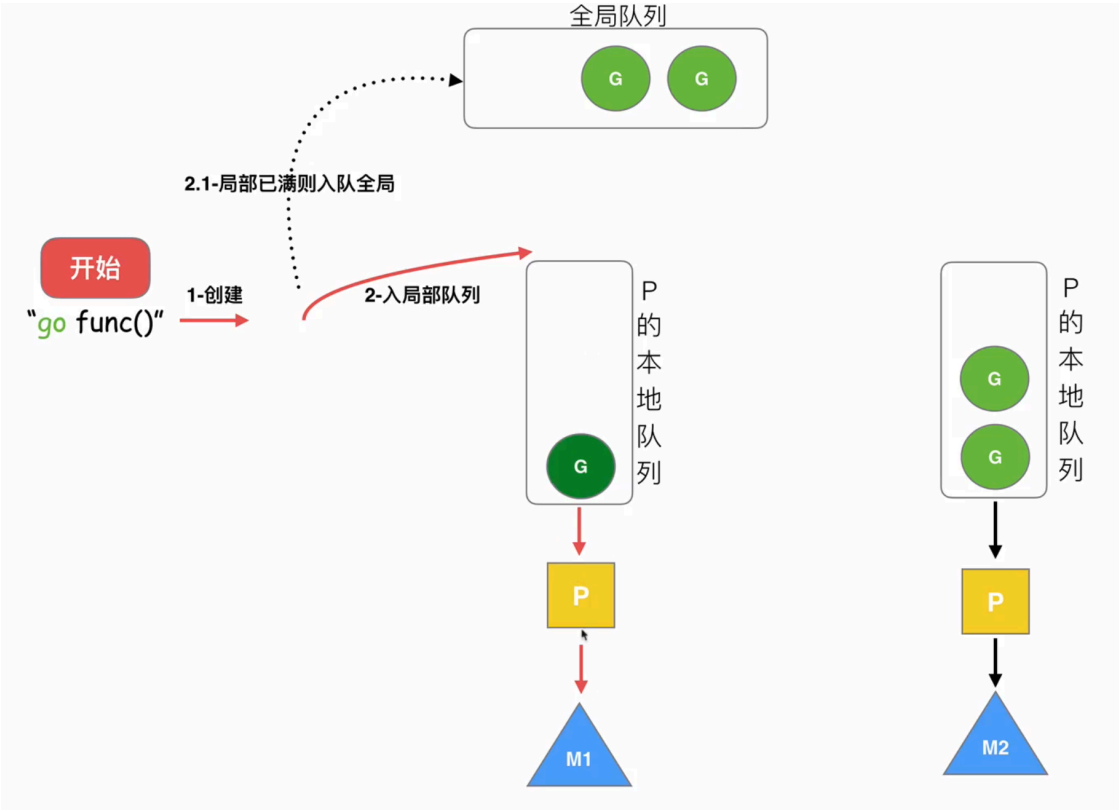
别的语言协程co-routine与cpu绑定会一直执行当前co-routine的任务，除非此co-routine主动释放cpu资源，其余的co-routine才有机会得到cpu资源。go中设计思想是M在运行goroutine时，如果有其他等待的goroutine，当前正在执行的goroutine最多10ms的执行时间，时间一到，等待的goroutine一定会抢占cpu资源，以达到最大化的并发度。

2.4 全局G队列

不仅每个P有自己的本地队列，go中还设计了一个全局队列。全局队列有锁的保护，每个P从全局队列中获取待运行的G的过程会比较慢，当P从其他P中“偷”不到G队列的时候，就会从全局队列中获取可运行的G。

3.“go func()”经历了什么过程

在执行“go func()”时，首先会创建一个G，这个G会优先加入到与当前线程M绑定的P的本地队列中，如果当前P的本地队列已满，则会放到全局队列中。如下图所示



执行过程中，P执行的G如果需要阻塞等待，根据阻塞等待的不同，会发生不同的情况：

- 同步阻塞：这个M1运行的G发生的syscall/阻塞是需要同步阻塞的时候，调度器会切换M1，创建/唤醒一个空闲的M3绑定当前P，运行剩下的G队列，当M1的同步阻塞执行完后，如果没有空闲的P，则会休眠或者销毁M1。
- 异步阻塞：这时不会发生M的切换，调度器会把阻塞的G绑定到network poller上，M1则继续执行当前P剩余的G队列，当阻塞的G执行完后会重新回到P队列，等待M运行它完成剩下的任务。

如果P的可执行队列为空，当前P就会从其他P中“偷”一半的G加入到自己的本地队列，如果其他P也没有可运行的本地队列，则会从全局队列中获取G。 这是一个循环机制，GPM在调度器的指挥下合作无间，最大程度的压榨cpu资源。

4.调度器的生命周期

go程序启动后会先进行一系列初始化操作，生成M0和G0。

- M0: 启动go程序后的编号为0的主线程。保存在全局变量runtime.m0中，不需要在heap上分配，负责执行初始化操作和启动第一个G，启动第一个G之后，M0就和其他M一样了。
- G0:每次启动一个M，都会立刻创建的goroutine，就是G0.G0仅用于负责调度的G，G0不指向任何可执行的函数。每个M都会有一个自己的G0，在调度或系统调用时M会先切换到G0，来调度可运行的G。M0的G0会放在全局空间。

生成G0后会进行一系列的初始化操作，完成P和全局队列的初始化。之后M0开始执行main goroutine，把G0给切换，然后找到空闲的P绑定，开始执行main goroutine。

5.可视化的GMP编程

```
package main

import (
    "fmt"
    "os"
    "runtime/trace"
)

func main() {
    // 1. 创建一个trace文件
    f, err := os.Create("trace.out")
    if err != nil {
        panic(err)
    }
    defer f.Close()
    // 2. 启动trace
    err = trace.Start(f)
    if err != nil {
        panic(err)
    }
    fmt.Println("Hello trace")
    // 3. 停止trace
    trace.Stop()
}
```

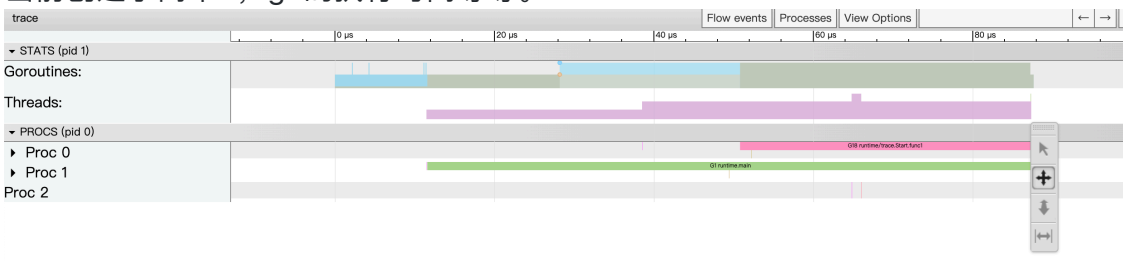
执行以上代码，会生成trace.out文件，通过以下命令：

```
jauhwan@jauhwan study % go tool trace trace.out
2020/06/17 22:15:07 Parsing trace...
2020/06/17 22:15:07 Splitting trace...
2020/06/17 22:15:07 Opening browser. Trace viewer is listening on
http://127.0.0.1:63980
```

可以解析生成的trace文件，通过浏览器访问可以得到详细报告。

View trace
Goroutine analysis
Network blocking profile (⬇)
Synchronization blocking profile (⬇)
Syscall blocking profile (⬇)
Scheduler latency profile (⬇)
User-defined tasks
User-defined regions
Minimum mutator utilization

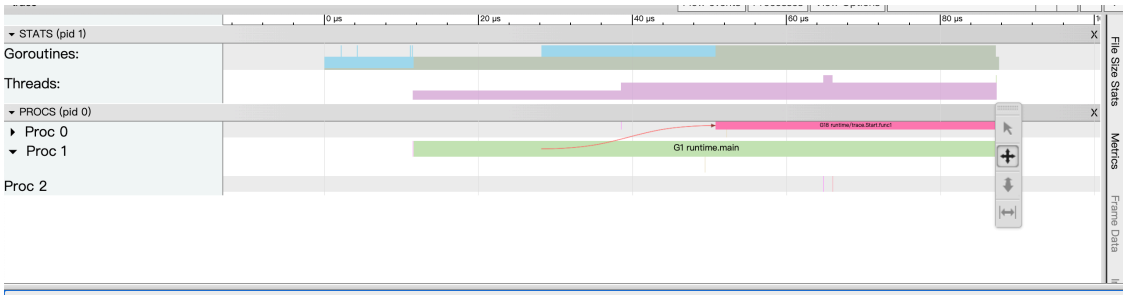
可以看到一系列的各种指标报告，用来进行代码分析。 点击view trace，可以看到当前创建了两个P，g0的执行时间等等。



3 items selected. Counter Samples (3)

Counter	Series	Time	Value
Goroutines	GCWaiting	0.028190000000000003	0
Goroutines	Runnable	0.028190000000000003	1
Goroutines	Running	0.028190000000000003	1

可以看到具体的p执行的G的耗时等等。



6 items selected. Slices (4) Flow Events (2)

Name	Wall Duration	Self time	Average Wall Duration	Occurrences
G1 runtime.main	76,043 ns	76,043 ns	76,043 ns	1
syscall	0 ns	0 ns	0 ns	2
G18 runtime/trace.Start.func1	36,411 ns	36,411 ns	36,411 ns	1
Totals	**112,454 ns**	**112,454 ns**	**28,114 ns**	**4**

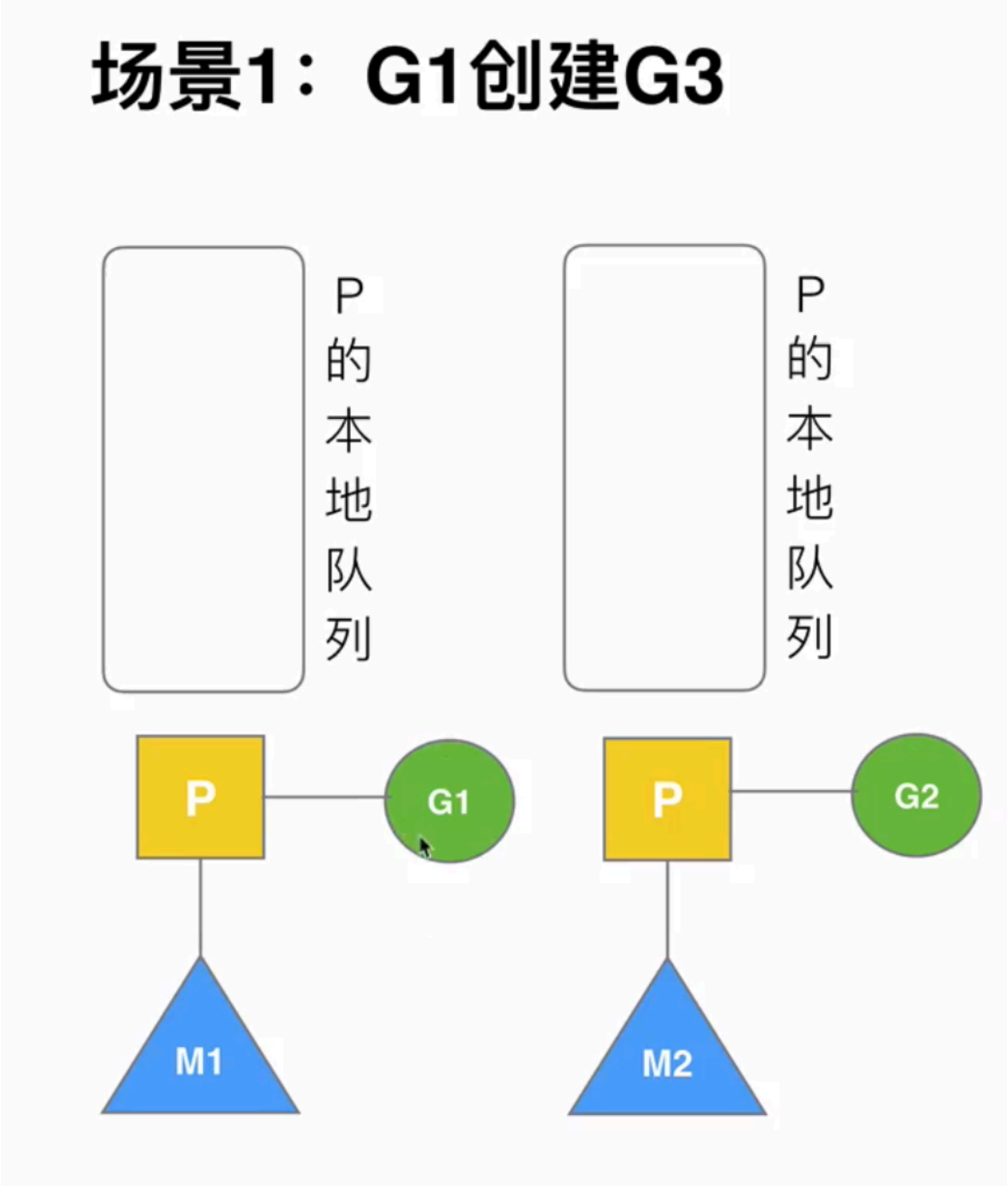
Selection start	11,586 ns
Selection extent	76,043 ns

Event(s)	Link
Outgoing flow	go
Internal flow	go
Preceding events	5 events of various types
Following events	6 events of various types
All connected events	6 events of various types

二、go调度器GMP调度场景分析

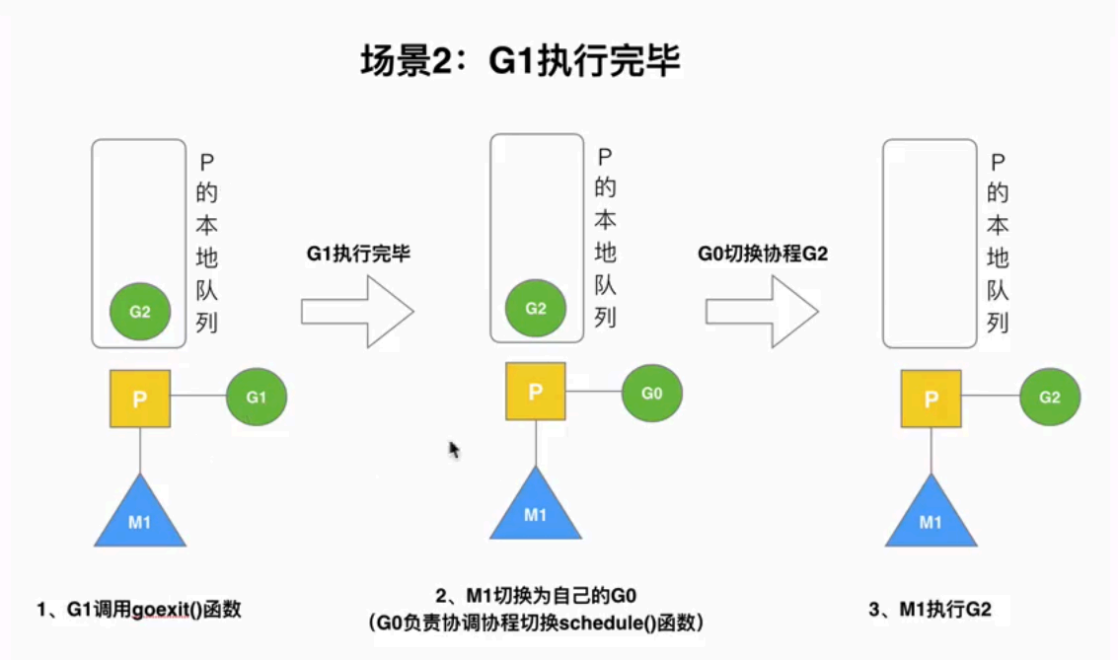
2.1 G1创建G3

当G1要创建一个G3时，为了保证局部性G3优先加入G1所在的P的本地队列。 如下图所示。



2.2 G1执行完毕

如下图所示，当G1执行完毕（函数：goexit()），M上运行的goroutine切换为G0，G0负责调度时协程的切换（函数schedule）。从P的本地队列取G2，从G0切换到G2，并开始运行G2（函数：execute）。

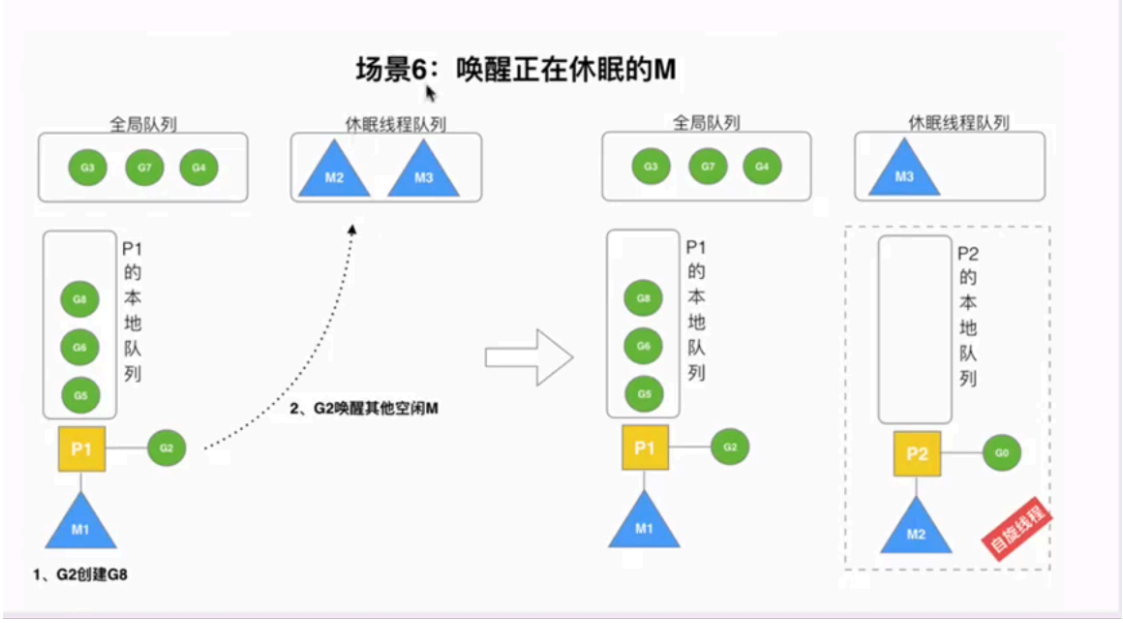


2.3 G2开辟过多的G

当某个G不断的连续创建新的G时，如果当前P的本地队列已满，调度器会把当前P队列头部的一半G顺序打乱放入全局队列中。

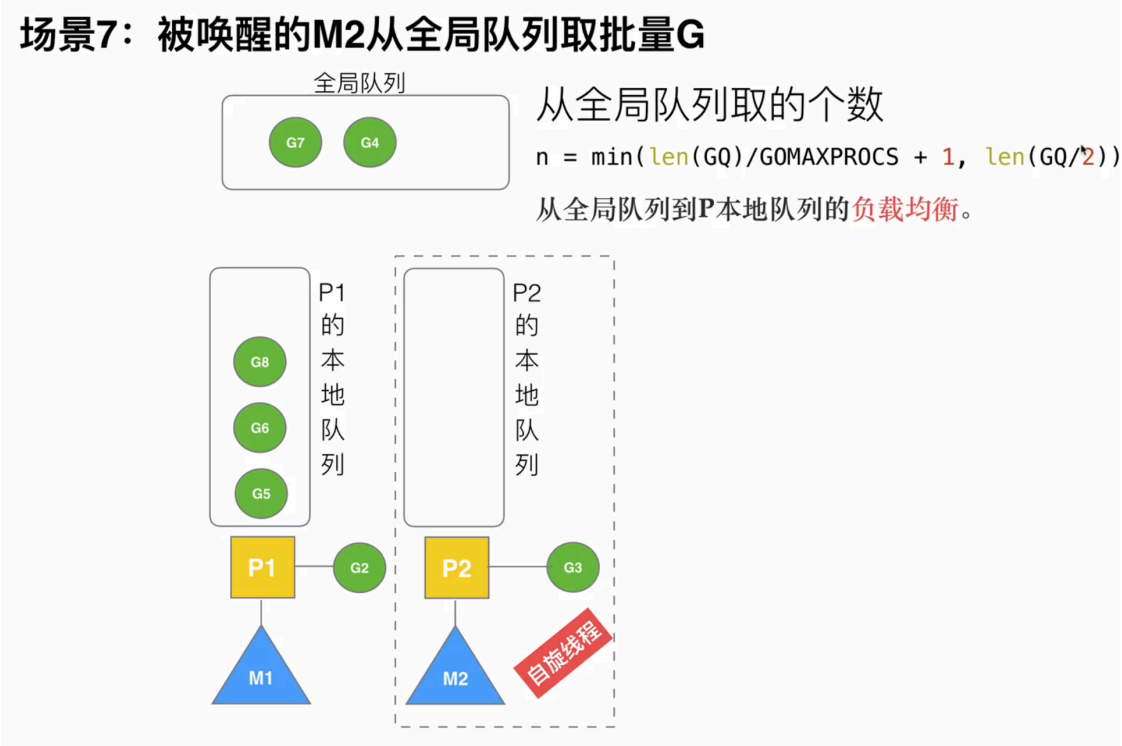
2.4 唤醒正在休眠的M

当G2创建G8的时，会尝试去唤醒一个正在休眠的线程，如果当前有空闲的P2，被唤醒的M2就会绑定到空闲的P2上，这个时候如果P2本地队列中没有G，这个线程就会不断的寻找G，尝试从其他P中“偷”或者从全局G队列中拿G，这个时候M2线程就称为自旋线程（没有G但为运行状态的状态，不断寻找G）。



2.5 被唤醒的M2从全局队列中取批量G

被唤醒的M2首先会从全局队列中批量获取G，获取G的数量如图中公示所示。n = min(len(GQ)/GOMAXPROCS+1,len(GQ/2))



2.6 M2从M1中偷取G

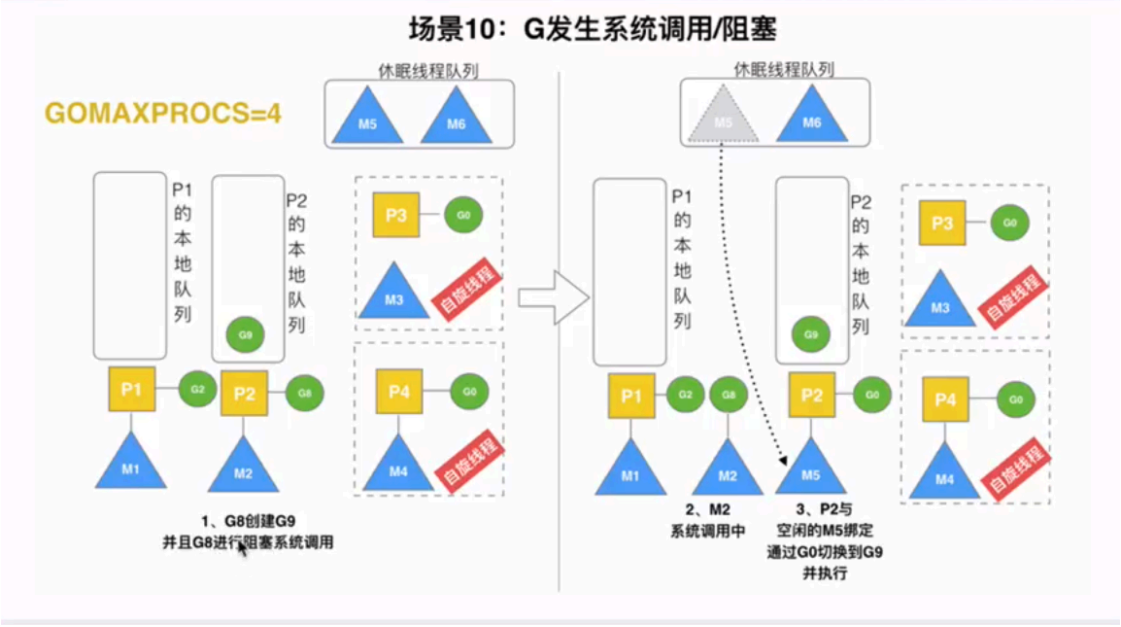
当全局队列为空时，M2就会从其他p的本地队列尾部中“偷”一半的G。

2.7 自旋线程的最大限制

自旋线程+执行线程<=GOMAXPROCS，因为P的个数为GOMAXPROCS，所以自旋线程过多也没有P与之绑定，多的线程会进入休眠状态，放在休眠线程队列中。

2.8 G发生系统调用/阻塞

G8如果发生系统调用/阻塞（同步阻塞）时，线程P2会和M2解绑，与休眠状态的M5进行绑定，不让cpu处于空闲状态。



2.9 G发生系统调用/非阻塞

当G8阻塞结束时，M2会尝试获取原来与它绑定的P2，如果无法获取，就会去获取空闲的P，如果依然获取不到，G8会被标记为可运行状态，加入到全局队列中。M2就会进入休眠状态。