GMP模型

- 深入golang runtime的调度

相关术语

> runtime 也叫运行时，对golang程序很重要，runtime包含了调度、内存管理、垃圾回收、内部数据结构、定时器和各种系统调用的封装等。可以说golang的强大都归功于runtime的实现。

> scheduler 有调度器、日程安排的意识，这里是指调度器，它的工作是将准备好运行的goroutine分散到工作线程中执行。

> TLS(thread local storage): TLS全称是Thread Local Storage，代表每个线程中的本地数据。写入TLS中的数据不会干扰到其余线程中的值。Go的协程实现非常依赖于TLS机制，会用于获取系统线程中当前的G和G所属于的M实例。 Go操作TLS会使用系统原生的接口，以Linux X64为例，go在新建M时候会调用arch_prctl 这个syscall来设置FS寄存器的值为M.tls的地址，运行中每个M的FS寄存器都会指向它们对应的M实例的tls， linux内核调度线程时FS寄存器会跟着线程一起切换，这样go代码只需要访问FS寄存器就可以获取到线程本地的数据。

> spinning 表示自旋，字面的意思是自己围绕自己转。在程序里一般指一直重复某块代码。

> systemstack、mcall或asmcgocall: 每个M启动都有一个叫g0的系统堆栈，runtime通常使用systemstack、mcall或asmcgocall临时切换到系统堆栈，以执行必须不被抢占的任务、不得增加用户堆栈的任务或切换用户goroutines。在系统堆栈上运行的代码隐式不可抢占，垃圾收集器不扫描系统堆栈。在系统堆栈上运行时，不会使用当前用户堆栈执行。

主要源码文件

```
runtime/asm_amd64.s 进程启动时调用的一些汇编函数，可以理解为进程的入口。
runtime/runtime2.go 主要是g、m、p、schedt的数据结构和g、p的状态定义，还定义了一些全局变量。
runtime/proc.go 主要是调度器逻辑代码的实现
```

调度基本组件

G（goroutine）

- 调度系统的最基本单位goroutine，存储了goroutine的执行stack信息、goroutine状态以及goroutine的任务函数等。
- 在G的眼中只有P，P就是运行G的“CPU”。
- 相当于两级线程

g的任务函数

```
go userFunc()
```

go 关键词创建了一个 goroutine ， 此时这个 goroutine 的任务函数就是 userFunc 。

P（processor）

- P表示逻辑processor，代表线程M的执行的上下文。
- P的最大作用是其拥有的各种G对象队列、链表、cache和状态。
- P的数量也代表了golang的执行并发度，即有多少goroutine可以同时运行

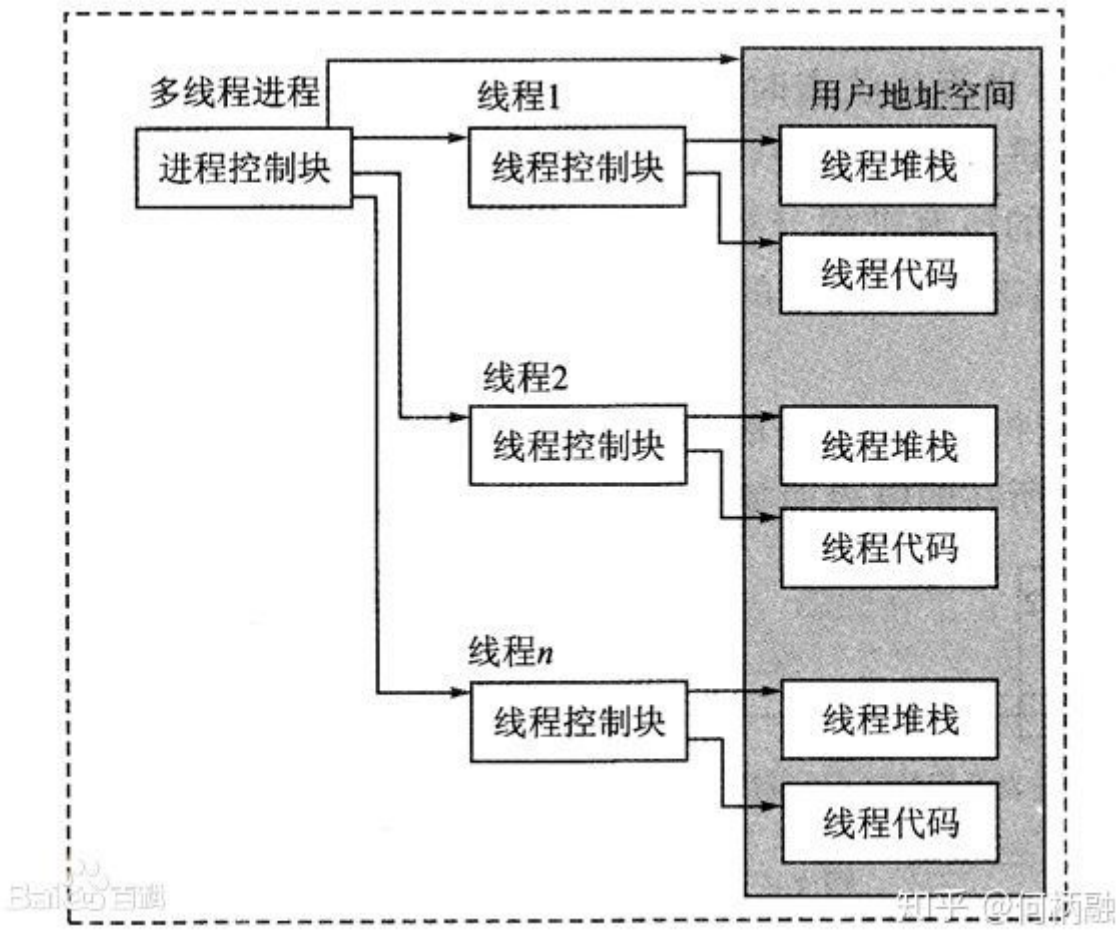
正确理解P

这里的p虽然表示逻辑处理器，但P并不执行任何代码，对G来说，P相当于CPU核，G只有绑定到P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

M（machine）

- M代表着真正的执行计算资源，可以认为它就是os thread（系统线程）。
- M是真正调度系统的执行者，每个M就像一个勤劳的工作者，总是从各种队列中找到可运行的G，而且这样M的可以同时存在多个。
- M在绑定有效的P后，进入调度循环，而且M并不保留G状态，这是G可以跨M调度的基础。

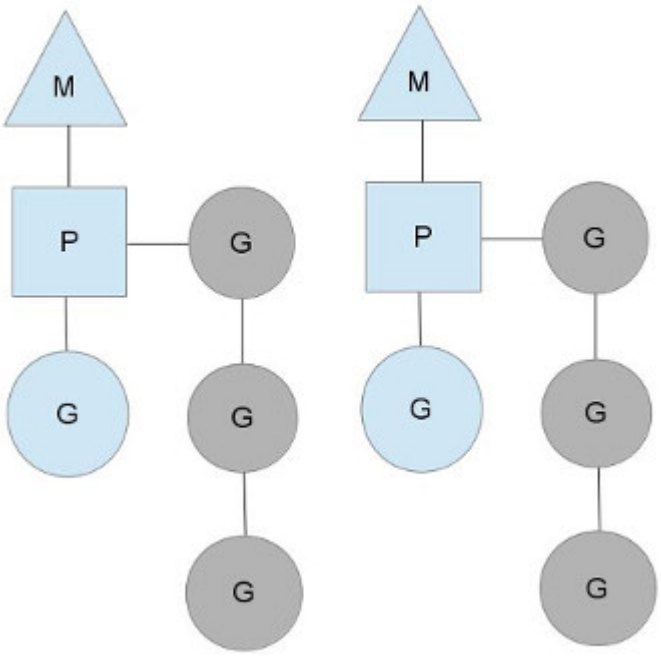
简单理解linux的线程概念



linux线程本质是轻量级进程，线程的特点是资源共享，意思就是线程之间共享进程的内存空间，所以一个线程可以访问另一个线程的局部变量， 线程也有自己独立的堆栈，用户的代码逻辑可以在这些堆栈上执行。当用户

启动一个线程时，需要指定一个让该线程执行的起始函数和该函数的参数， 那么线程要做的事情就是去执行用户指定的函数。而且在线程里可以创建另一个线程。

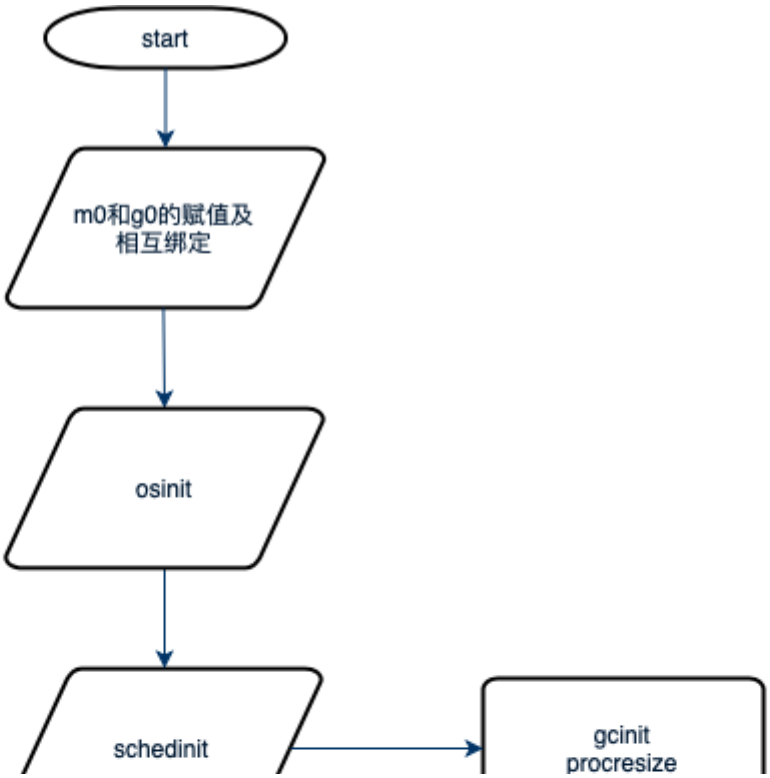
GPM的关系示意图

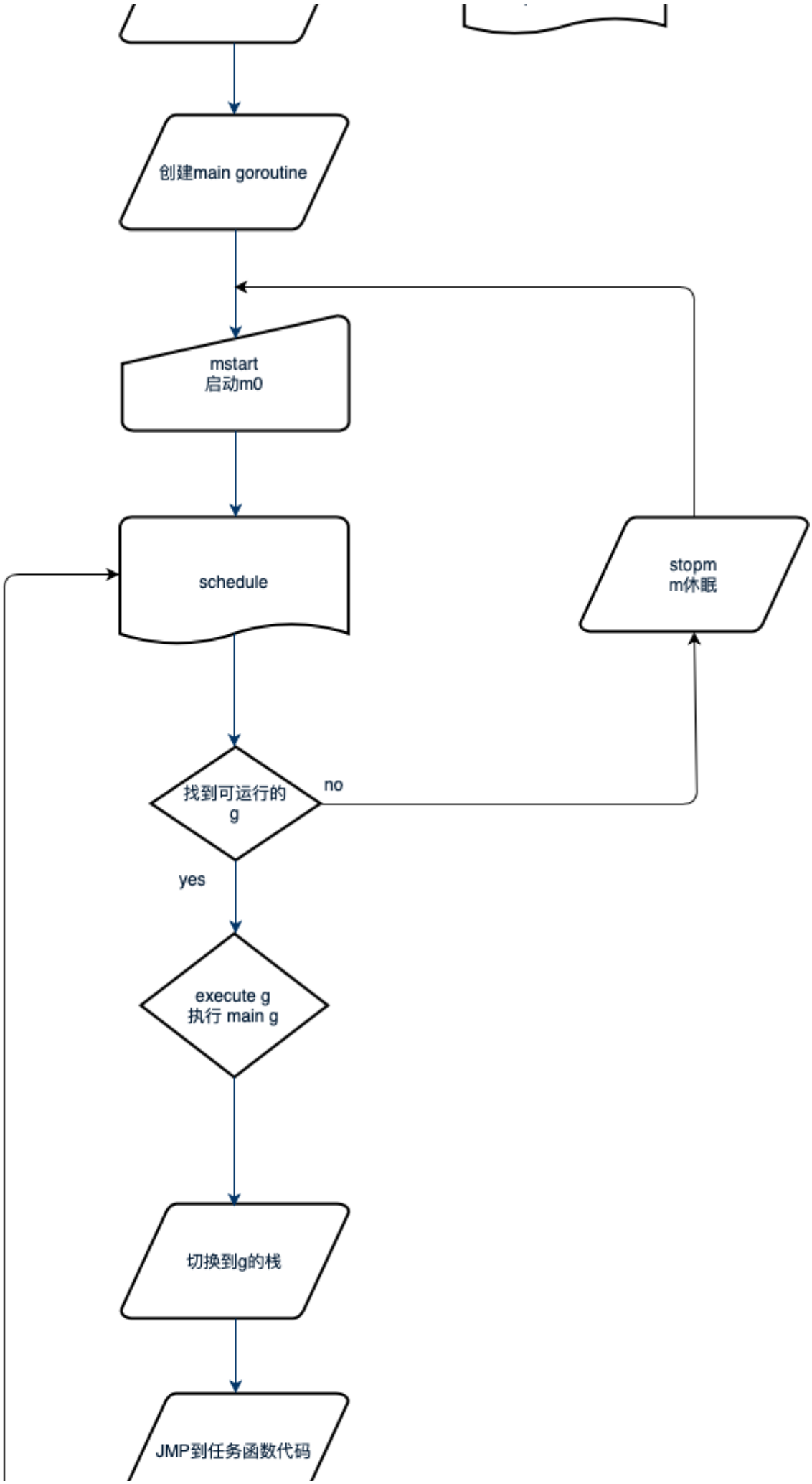


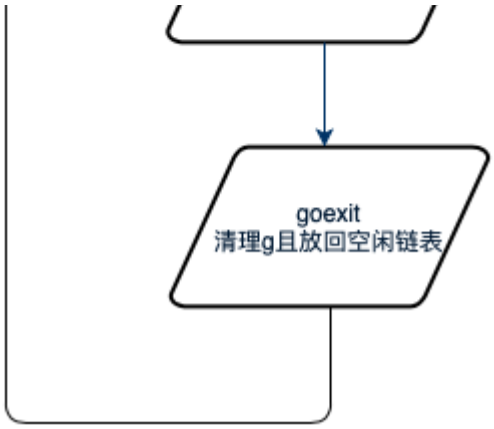
通过连线可以看出来，M和P有关联，G和P有关联，但M和G没有关联，这是为何G可以在不同的M上执行的本质原因。

整体流程概览

这里先整理一下整个流程，如下图：




创建main goroutine
mstart
启动m0
schedule
stopm
m休眠
找到可运行的
g
no
yes
execute g
执行 main g
切换到g的栈
JMP到任务函数代码



go进程的启动

进程的本质是代码区的指令不断执行，驱使动态数据区和静态数据区产生数据变化。- 来自图解编译原理

先看一下golang进程怎么启动的。熟悉c的同学应该知道，c语言的main函数是程序的入口函数，在golang中main包中的main函数并不是入口函数， 入口函数是在asm_amd64.s中定义的，而main包中的main函数是由runtime main函数启动的。但是这里我们不详细探讨系统是怎么到程序入口函数的， 我们只需要明白，go程序启动后，会调用 runtime·rt0_go 来执行程序的初始化和启动调度系统。runtime·rt0_go 很重要，如果要是自己看runtime的源码， 可以从这个函数看起。

runtime/asm_amd64.s

```
// runtime·rt0_go

// 程序刚启动的时候必定有一个线程启动（主线程）
// 将当前的栈和资源保存在g0
// 将该线程保存在m0
// tls: Thread Local Storage
// set the per-goroutine and per-mach "registers"
get_tls(BX)
LEAQ    runtime·g0(SB), CX
MOVQ    CX, g(BX)
LEAQ    runtime·m0(SB), AX

// m0和g0互相绑定
// save m->g0 = g0
MOVQ    CX, m_g0(AX)
// save m0 to g0->m
MOVQ    AX, g_m(CX)
// 处理args
CALL    runtime·args(SB)
// os初始化， os_linux.go
CALL    runtime·osinit(SB)
// 调度系统初始化，proc.go
CALL    runtime·schedinit(SB)

// 创建一个goroutine，然后开启执行程序
// create a new goroutine to start program
MOVQ    $runtime·mainPC(SB), AX     // entry
```

```
PUSHQ   AX
PUSHQ   $0          // arg size
CALL    runtime·newproc(SB)
POPQ    AX
POPQ    AX

// start this M
// 启动线程，并且启动调度系统
CALL    runtime·mstart(SB)
```

runtime/os_linux.go

```
// 获取cpu的数量
func osinit() {
    ncpu = getproccount()
}
```

runtime/proc.go

```
// 调度系统的初始化
func schedinit() {
    ...

    // 确认P的个数
    // 默认等于cpu个数，可以通过GOMAXPROCS环境变量更改
    procs := ncpu
    if n, ok := atoi32(gogetenv("GOMAXPROCS")); ok && n > 0 {
        procs = n
    }
    // 调整P的个数，这里是新分配procs个P
    // 这个函数很重要，所有的P都是从这里分配的，以后也不用担心没有P了
    if procresize(procs) != nil {
        throw("unknown runnable goroutine during bootstrap")
    }
    ...
}
```

go进程的启动在 proc.go 的注释中有写到，

```
// The bootstrap sequence is:
//
//  call osinit
//  call schedinit
//  make & queue new G
//  call runtime·mstart
```

主要分为四个步骤，

1. 调用 runtime·osinit 来获取系统的cpu个数。
2. 调用 runtime·schedinit 来初始化调度系统，会进行p的初始化，也会把m0和某个p绑定。 3. 调用 runtime·newproc 新建一个goroutine，也叫main goroutine，它的任务函数是 runtime.main 函数，建好后插入m0绑定的p的本地队列。
3. 调用 runtime·mstart 来启动m，进入启动调度系统。

我们已经看到了调用 runtime·mstart 来启动m，那刚新建的 main goroutine 是如何被调度来执行的呢？回答这个问题需要，继续探究 mstart 的实现。

runtime调度器的启动

上面我们看到入口函数是调用 `runtime·mstart(SB) 来启动的，那么就分析一下 mstart 做了什么?

runtime/proc.go

```
func mstart() {
    // 这里获取的g是g0，在系统堆栈
    _g_ := getg()

    ...

    mstart1(0)

    ...
}

// dummy一直为0，给getcallersp当参数
func mstart1(dummy int32) {
    _g_ := getg()

    // 确保g是系统栈上的g0
    // 调度器只在g0上执行
    if _g_ != _g_.m.g0 {
        throw("bad runtime·mstart")
    }

    ...

    // 初始化m，主要是设置线程的备用信号堆栈和信号掩码
    minit()

    // Install signal handlers; after minit so that minit can
    // prepare the thread to be able to handle the signals.
    // 如果当前g的m是初始m0，执行mstartm0()
    if _g_.m == &m0 {
        // 对于初始m，需要一些特殊处理，主要是设置系统信号量的处理函数
        mstartm0()
    }

    // 如果有m的起始任务函数，则执行，比如 sysmon 函数
```

```
    // 对于m0来说，是没有 mstartfn 的
    if fn := _g_.m.mstartfn; fn != nil {
        fn()
    }

    if _g_.m.helpgc != 0 {
        _g_.m.helpgc = 0
        stopm()
    } else if _g_.m != &m0 { // 如果不是m0，需要绑定p
        // 绑定p
        acquirep(_g_.m.nextp.ptr())
        _g_.m.nextp = 0
    }

    // 进入调度，而且不会在返回
    schedule()
}
```

mstart 只是简单的对 mstart1 的封装，接着看 mstart1 。

1. 调用 *g* := getg() 获取 g ，然后检查该 g 是不是 g0 ，如果不是 g0 ，直接抛出异常。
2. 初始化m，主要是设置线程的备用信号堆栈和信号掩码
3. 判断 g 绑定的 m 是不是 m0，如果是，需要一些特殊处理，主要是设置系统信号量的处理函数
4. 检查 m 是否有起始任务函数？若有，执行它。
5. 获取一个p，并和m绑定
6. 进入调度程序

总结来看，mstart1 也未进行真正的调度，而是调度前的一些处理，但是看到这里有多了几个疑问，

1. getg() 做了什么？
2. 检查 g 是否正确，不是 g0 直接退出，那这个 g0 是什么呢？ m0 又是什么？
3. mstartfn 对应的实际函数是啥？

真是越看问题越来越多啊，但没关系，这里我先解答上面的问题，如果不感兴趣的同学可以直接跳到 schedule 的分析。

getg() 函数

如果你看过runtime的源码，当你去查看 getg() 函数的定义时，你有可能会疑惑，为何这个函数没有body？

runtime/stubs.go

```
// getg returns the pointer to the current g.
// The compiler rewrites calls to this function into instructions
// that fetch the g directly (from TLS or from the dedicated register).
func getg() *g
```

这里只有函数的声明，却没有函数的实体，是因为这个函数的实体是编译器重写的，也就是编译器中才定义了该函数的实体。笔者也没有深究 getg 的实体具体是什么， 所以根据注释和自己的理解来解释这个函数做了什

么。

getg 返回指向当前g的指针。编译器将对此函数的调用重写为直接获取g的指令（来自TLS或来自专用寄存器）。 要获取当前用户堆栈的g，可以使用getg().m.curg。getg()返回当前g，但是当在系统或信号堆栈上执行时，这将分别返回当前m的 g0 或 gsignal 。 要确定g是在用户堆栈还是系统堆栈上运行，可以使用getg() == getg().m.curg，相等表示在用户态堆栈，不相等表示在系统堆栈。

g0和m0

g0 和 m0 在runtime中是比较重要的概念这里讲一下，它们到底什么?

- m0: m0 表示进程启动的第一个线程，也叫主线程。它和其他的m没有什么区别，要说区别的话，它是进程启动通过汇编直接复制给m0的，m0是个全局变量，而其他的m都是runtime内自己创建的。 m0 的赋值过程，可以看前面 runtime/asm_amd64.s 的代码。一个go进程只有一个m0。
- g0: 首先要明确的是每个m都有一个g0，因为每个线程有一个系统堆栈，g0 虽然也是g的结构，但和普通的g还是有差别的，最重要的差别就是栈的差别。g0 上的栈是系统分配的栈，在linux上栈大小默认固定8MB，不能扩展，也不能缩小。 而普通g一开始只有2KB大小，可扩展。在 g0 上也没有任何任务函数，也没有任何状态，并且它不能被调度程序抢占。因为调度就是在g0上跑的。
- proc.go 中的全局变量 m0和g0

```
var (
    m0           m
    g0           g
    raceprocctx0 uintptr
)
```

在 runtime/proc.go 的文件中声明了两个全局变量，m0表示主线程，这里的g0表示和m0绑定的g0，也可以理解为m0线程的堆栈，这两个变量的赋值是汇编实现的。

到这里我们应该知道了g0和m0是什么了？ m0代表主线程、g0代表了线程的堆栈。调度都是在系统堆栈上跑的，也就是一定要跑在 g0 上，所以 mstart1 函数才检查是不是在g0上， 因为接下来就要执行调度程序了。

mstartfn

mstartfn 其实很简单，就是一个函数，有值就执行就好了，但是 mstartfn 实际对应的函数是什么？在runtime搜了一下，对应的函数还真不少，

```
newm(sysmon, nil)
newm(mhelpgc, nil)
newm(templateThread, nil)
mspinning
```

这里先不谈每个函数的意义了，知道有些情况，会执行 mstartfn 就够了。而且可以知道在go程序刚起的时候，mstartfn 是为 nil 的。

到目前为止，我们还没发现有任何调度程序去运行main goroutine。心中的疑惑依然没有解决，到底什么时候 main goroutine 才被调度运行啊？ 带着疑问继续看 schedule 函数。

真正的调度函数 schedule

那么 schedule 做了什么呢？注意，下面的源码忽略了一些gc和调度相关的操作。

runtime/proc.go

```
func schedule() {
    _g_ := getg()

    ...

top:
    // 如果当前GC需要停止整个世界（STW），则调用gcstopm休眠当前的M
    if sched.gcwaiting != 0 {
        // 为了STW，停止当前的M
        gcstopm()
        // STW结束后回到 top
        goto top
    }

    ...

    var gp *g
    var inheritTime bool

    ...

    if gp == nil {
        // Check the global runnable queue once in a while to ensure fairness.
        // Otherwise two goroutines can completely occupy the local runqueue
        // by constantly respawning each other.
        // 每隔61次调度，尝试从全局队列种获取G
        if _g_.m.p.ptr().schedtick%61 == 0 && sched.runqsize > 0 {
            lock(&sched.lock)
            gp = globrunqget(_g_.m.p.ptr(), 1)
            unlock(&sched.lock)
        }
    }
    if gp == nil {
        // 从p的本地队列中获取
        gp, inheritTime = runqget(_g_.m.p.ptr())
        if gp != nil && _g_.m.spinning {
            throw("schedule: spinning with local work")
        }
    }
    if gp == nil {
        // 想尽办法找到可运行的G，找不到就不用返回了
        gp, inheritTime = findrunnable() // blocks until work is available
    }
```

```
    ...

    // println("execute goroutine", gp.goid)
    // 找到了g，那就执行g上的任务函数
    execute(gp, inheritTime)
}
```

可以看到调度程序本质就是尽力找到可运行的g，然后去运行g上面的任务函数。查找g的流程如下，

1. 如果当前GC需要停止整个世界（STW), 则调用gcstopm休眠当前的M。
2. 每隔61次调度轮回从全局队列找，避免全局队列中的g被饿死。
3. 从p.runnext获取g，从p的本地队列中获取。
4. 调用 findrunnable 找g，找不到的话就将m休眠，等待唤醒。

当找到一个g后，就会调用 execute 去执行g，我们暂时不管 execute 是如何切到g的栈上和执行g的任务函数。那么到这里我们可以回答我们心中的疑惑了。前面说了 main goroutine 在和m绑定的p的本地队列中，那么调用 runqget 的时候 就可以从p的本地队列中获取到 main goroutine ，然后传给 execute 执行，这样 main goroutine就跑起来了。 那么 main goroutine 又做了什么呢?

runtime.main 的执行

由于 main goroutine 的任务函数是 runtime 中定义的 main 函数，所以我们看看 main 函数。

```
// The main goroutine.
func main() {
    // 获取 main goroutine
    g := getg()

    ...

    // 在系统栈上运行 sysmon
    systemstack(func() {
        // 分配一个新的m，运行sysmon系统后台监控
        //  （定期垃圾回收和调度抢占）
        newm(sysmon, nil)
    })

    ...

    // 确保是主线程
    if g.m != &m0 {
        throw("runtime.main not on m0")
    }

    // runtime 内部 init 函数的执行，编译器动态生成的。
    runtime_init() // must be before defer

    ...

    // gc 启动一个goroutine进行gc清扫
    gcenable()
```

```
    ...

    // 执行init函数，编译器动态生成的，
    // 包括用户定义的所有的init函数。
    // make an indirect call,
    // as the linker doesn't know the address of
    // the main package when laying down the runtime
    fn := main_init
    fn()

    ...

    // 真正的执行main func in package main
    // make an indirect call,
    // as the linker doesn't know the address of
    // the main package when laying down the runtime
    fn = main_main
    fn()

    ...

    // 退出程序
    exit(0)

    // 为何这里还需要for循环?
    // 下面的for循环一定会导致程序崩掉，这样就确保了程序一定会退出
    for {
        var x *int32
        *x = 0
    }
}
```

main 在执行main包中的main函数之前，还是做了一些其他工作包括：

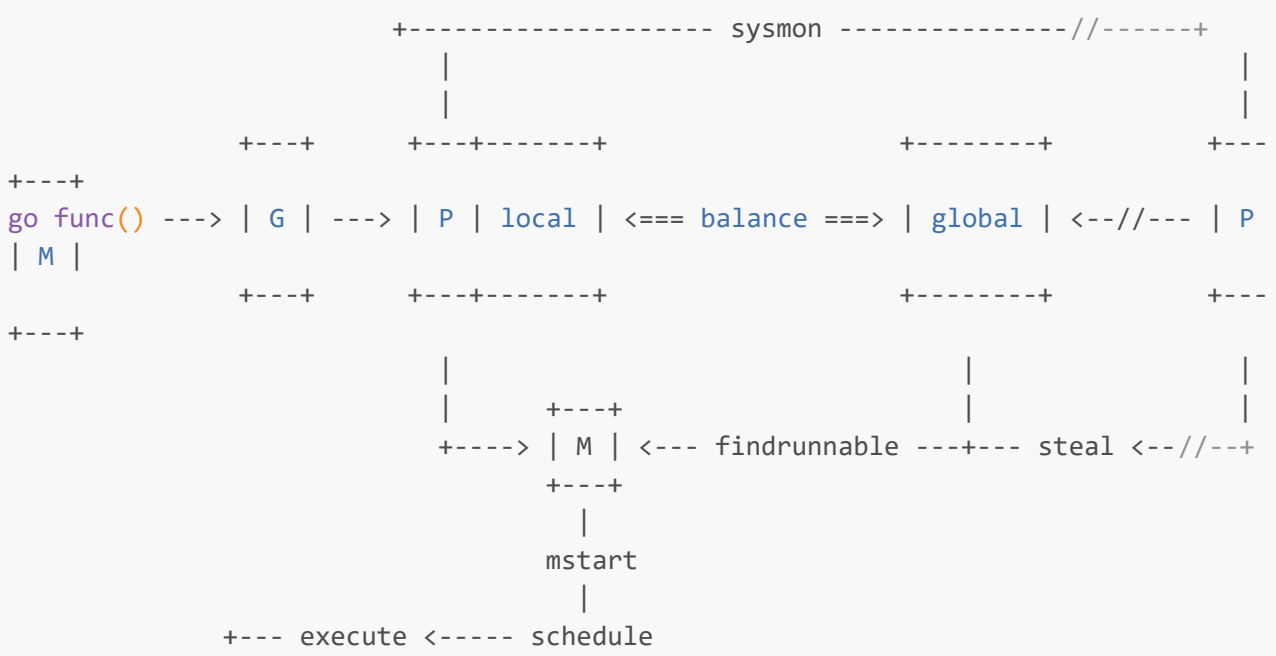
1. 新建一个线程来执行 sysmon ， sysmon的工作是系统后台监控（定期垃圾回收和调度抢占），下节再详细介绍。
2. 确保是在主线程上运行
3. runtime内存init函数的执行，runtime_init 是由编译器动态生成的，里面包含了 runtime 包中所有的 init 函数，感兴趣的同学可以在runtime包中搜 func init() ，会发现有挺多init函数。
4. 启动gc清扫的goroutine
5. 执行 main_init 函数，编译器动态生成的，包括用户定义的所有的init函数。

接下来就是真的执行 main 包中的main函数了，这里 main 函数由用户定义其具体内容，如果该 main 函数结束了，就会执行 exit(0) 来退出进程。 若是 exit(0) 后，进程没退出，由接下来的代码确保进程一会退出，就是for循环一直访问非法地址，正常情况下，一但出现非法地址访问，系统就会把该进程杀死，用这样的方法来确保进程退出。

调度机制

上面讲述了runtime启动用户代码main函数的过程，但是如果用户在代码中新建了很多goroutine，runtime又是如何管理的呢?

调度架构概览图



1. go func() 语气创建G。
2. 将G放入P的本地队列（或者平衡到全局队列）。
3. 唤醒或新建M来执行任务。
4. 进入调度循环
5. 尽力获取可执行的G，并执行
6. 清理现场并且重新进入调度循环

上图基本上概括了整个调度流程，go 关键字创建了G，并插入到P的本地队列或者全局队列，线程M从各个队列中或者从别的P中得到G， 切换到G的执行栈上并执行G上的任务函数，调用goexit做清理工作并回到调度程序，调度程序重新找个可执行的G，并执行，如此反复。 其中 sysmon 会监控整个调度系统，如果某个G长时间占用cpu，会被标记为可抢占。

接下来详细看一下golang的调度器在进程中是怎么起来的。

基本思想

> 复用线程

goroutines本身就是运行在一组线程之上，不需要频繁的创建、销毁线程，而是对线程的复用。在调度器中复用线程还有2个体现：

1. work stealing，当本线程无可运行的G时，尝试从其他线程绑定的P偷取G，而不是销毁线程。
2. hand off，当本线程因为G进行系统调用阻塞时，线程释放绑定的P，把P转移给其他空闲的线程执行。

> 利用并行 Go的一个强项就是天生支持并发，并发包含了并行的实现，并行度由GOMAXPROCS来控制，默认等于cpu数，当GOMAXPROCS大于1时，就最多有GOMAXPROCS个线程同时处于运行状态，这些线程可能分布在多个CPU核上同时运行，这样就可以充分利用多核。

> 调度器的两小策略

- 全局G队列 在调度器中有全局G队列，用来均衡每个P上面G的数量。
- 一定的抢占 当一个goroutine占用cpu超过10ms，会被抢占，防止其他goroutine饿死。

类比模型

为了更方便的理解调度器，我拿一个现实的例子来比喻，假设有一个物流公司，里面有司机、车、货物，类比M、P、G，当有用户需要运输货物（G）时，司机能得到货物，但是此时司机（M）是无法运输货物的，因为他还没有车（P），所以司机先从车库中找到一辆空闲的车，然后再将货物运到目的地，此时一次货物的运输完成，就像完成用户的模块逻辑代码。然后公司越开越大，现在有多个司机，也有多辆车，当然要运输的货物也很多。这里的每个司机都很勤劳，他们日夜的在运输货物，除非实在没有货物来运了，他们才会去休息(M休眠)。为了避免有些司机很轻松没事干，而有些司机忙不过来，当司机发现自己没活干了，他就会去向别的车里面拿一些货物过来，帮忙运输（M偷其他P里面的G）。

关于G

G是调度的对象。G存储了goroutine的执行stack信息、goroutine状态以及goroutine的任务函数等，但它并不执行用户代码。

G的结构

```
type g struct {
    // Stack parameters.
    // stack describes the actual stack memory: [stack.lo, stack.hi).
    // stackguard0 is the stack pointer compared in the Go stack growth prologue.
    // It is stack.lo+StackGuard normally, but can be StackPreempt to trigger a
preemption.
    // stackguard1 is the stack pointer compared in the C stack growth prologue.
    // It is stack.lo+StackGuard on g0 and gsignal stacks.
    // It is ~0 on other goroutine stacks, to trigger a call to morestackc (and
crash).
    // 简单数据结构，lo 和 hi 成员描述了栈的下界和上界内存地址
    stack       stack   // offset known to runtime/cgo
    stackguard0 uintptr // offset known to liblink
    stackguard1 uintptr // offset known to liblink

    _panic *_panic // innermost panic - offset known to liblink
    _defer *_defer // innermost defer
    // 当前的m
    m *m // current m; offset known to arm liblink
    // goroutine切换时，用于保存g的上下文
    sched     gobuf
    syscallsp uintptr // if status==Gsyscall, syscallsp = sched.sp to use during
gc
    syscallpc uintptr // if status==Gsyscall, syscallpc = sched.pc to use during
gc
```

```
    stktopsp  uintptr // expected sp at top of stack, to check in traceback
    // 用于传递参数，睡眠时其他goroutine可以设置param，唤醒时该goroutine可以获取
    param        unsafe.Pointer // passed parameter on wakeup
    atomicstatus uint32
    stackLock    uint32 // sigprof/scang lock; TODO: fold in to atomicstatus
    // 唯一的goroutine的ID
    goid int64
    // g被阻塞的大体时间
    waitsince  int64  // approx time when the g become blocked
    waitreason string // if status==Gwaiting
    schedlink  guintptr
    // 标记是否可抢占
    preempt        bool     // preemption signal, duplicates stackguard0 =
stackpreempt
    paniconfault   bool     // panic (instead of crash) on unexpected fault
address
    preemptscan    bool     // preempted g does scan for gc
    gcscandone     bool     // g has scanned stack; protected by _Gscan bit in
status
    gcscanvalid    bool     // false at start of gc cycle, true if G has not run
since last scan; TODO: remove?
    throwsplit     bool     // must not split stack
    raceignore     int8     // ignore race detection events
    sysblocktraced bool     // StartTrace has emitted EvGoInSyscall about this
goroutine
    sysexitticks   int64    // cputicks when syscall has returned (for tracing)
    traceseq       uint64   // trace event sequencer
    tracelastp     puintptr // last P emitted an event for this goroutine
    // G被锁定只在这个m上运行
    lockedm  muintptr
    sig      uint32
    writebuf []byte
    sigcode0 uintptr
    sigcode1 uintptr
    sigpc    uintptr
    // 调用者的 PC/IP
    gopc uintptr // pc of go statement that created this goroutine
    // 任务函数
    startpc    uintptr // pc of goroutine function
    racectx    uintptr
    waiting    *sudog         // sudog structures this g is waiting on (that have
a valid elem ptr); in lock order
    cgoCtxt    []uintptr      // cgo traceback context
    labels     unsafe.Pointer // profiler labels
    timer      *timer         // cached timer for time.Sleep
    selectDone uint32         // are we participating in a select and did someone
win the race?

    // Per-G GC state

    // gcAssistBytes is this G's GC assist credit in terms of
    // bytes allocated. If this is positive, then the G has credit
    // to allocate gcAssistBytes bytes without assisting. If this
    // is negative, then the G must correct this by performing
```

```
    // scan work. We track this in bytes to make it fast to update
    // and check for debt in the malloc hot path. The assist ratio
    // determines how this corresponds to scan work debt.
    gcAssistBytes int64
}
```

可以说G的一生都由runtime管理，包括G的新建, 休眠, 恢复, 停止。 这里需要注意的是，一个G一但被创建，那就不会消失，因为runtime有个allgs保存着所有的g指针，但不要担心，g对象引用的其他对象是会释放的，所以也占不了啥内存。

G的新建

```
go func() { ... }
```

当使用go关键字新建一个goroutine时，runtime 会调用 newproc 来生成新的g，详细流程如下：

1. 用 systemstack 切换到系统堆栈，调用 newproc1 ， newproc1 实现g的获取。
2. 尝试从p的本地g空闲链表和全局g空闲链表找到一个g的实例。
3. 如果上面未找到，则调用 malg 生成新的g的实例，且分配好g的栈和设置好栈的边界，接着添加到 allgs 数组里面，allgs保存了所有的g。
4. 保存g切换的上下文，这里很关键，g的切换依赖 sched 字段。
5. 生成唯一的goid，赋值给该g。
6. 调用 runqput 将g插入队列中，如果本地队列还有剩余的位置，将G插入本地队列的尾部，若本地队列已满，插入全局队列。
7. 如果有空闲的p 且 m没有处于自旋状态 且 main goroutine已经启动，那么唤醒或新建某个m来执行任务。

```
// Create a new g running fn with siz bytes of arguments.
// Put it on the queue of g's waiting to run.
// The compiler turns a go statement into a call to this.
// Cannot split the stack because it assumes that the arguments
// are available sequentially after &fn; they would not be
// copied if a stack split occurred.
//go:nosplit
// 新建一个goroutine,
// 用fn + PtrSize 获取第一个参数的地址，也就是argp
// 用siz - 8 获取pc地址
func newproc(siz int32, fn *funcval) {
    argp := add(unsafe.Pointer(&fn), sys.PtrSize)
    pc := getcallerpc()
    // 用g0的栈创建G对象
    systemstack(func() {
        newproc1(fn, (*uint8)(argp), siz, pc)
    })
}

// Create a new g running fn with narg bytes of arguments starting
// at argp. callerpc is the address of the go statement that created
```

```
// this. The new g is put on the queue of g's waiting to run.
// 根据函数参数和函数地址，创建一个新的G，然后将这个G加入队列等待运行
// callerpc是newproc函数的pc
func newproc1(fn *funcval, argp *uint8, narg int32, callerpc uintptr) {
    // print("fn=", fn.fn, " argp=", argp, " narg=", narg, " callerpc=", callerpc,
"\n")
    _g_ := getg() // g0

    // 任务函数如果为nil，则抛出异常
    if fn == nil {
        _g_.m.throwing = -1 // do not dump full stacks
        throw("go of nil func value")
    }
    // 禁用抢占，因为它可以在本地var中保存p
    _g_.m.locks++ // disable preemption because it can be holding p in a local var
    siz := narg
    siz = (siz + 7) &^ 7

    // We could allocate a larger initial stack if necessary.
    // Not worth it: this is almost always an error.
    // 4*sizeof(uintreg): extra space added below
    // sizeof(uintreg): caller's LR (arm) or return address (x86, in gostartcall).
    // 如果函数的参数大小比2048大的话，直接panic
    if siz >= _StackMin-4*sys.RegSize-sys.RegSize {
        throw("newproc: function arguments too large for new goroutine")
    }

    // 从m中获取p
    _p_ := _g_.m.p.ptr()
    // 尝试从gfree list获取g，包括本地和全局list
    newg := gfget(_p_)
    // 如果没获取到g，则新建一个
    if newg == nil {
        // 分配栈为 2k 大小的G对象
        newg = malg(_StackMin)
        casgstatus(newg, _Gidle, _Gdead) //将g的状态改为_Gdead
        // 添加到allg数组，防止gc扫描清除掉
        allgadd(newg) // publishes with a g->status of Gdead so GC scanner doesn't
look at uninitialized stack.
    }
    if newg.stack.hi == 0 {
        throw("newproc1: newg missing stack")
    }

    // 此时获取的g一定得是 _Gdead
    if readgstatus(newg) != _Gdead {
        throw("newproc1: new g is not Gdead")
    }

    // 参数大小+稍微一点空间
    totalSize := 4*sys.RegSize + uintptr(siz) + sys.MinFrameSize // extra space in
case of reads slightly beyond frame
    totalSize += -totalSize & (sys.SpAlign - 1)                  // align to
spAlign
```

```
    // 新协程的栈顶计算，将栈顶减去参数占用的空间
    sp := newg.stack.hi - totalSize
    spArg := sp
    if usesLR { // amd64平台下 usesLR 为 false
        // caller's LR
        *(*uintptr)(unsafe.Pointer(sp)) = 0
        prepGoExitFrame(sp)
        spArg += sys.MinFrameSize
    }

    // 如果有参数
    if narg > 0 {
        // copy参数到栈上
        memmove(unsafe.Pointer(spArg), unsafe.Pointer(argp), uintptr(narg))
        // 这是一个堆栈到堆栈的副本。如果启用了写入屏障并且源堆栈为灰色（目标始终为黑色），
        // 则执行屏障复制。我们在* memmove之后执行此操作，因为目标堆栈上可能有垃圾。
        if writeBarrier.needed && !_g_.m.curg.gcscandone {
            f := findfunc(fn.fn)
            stkmap := (*stackmap)(funcdata(f, _FUNCDATA_ArgsPointerMaps))
            // We're in the prologue, so it's always stack map index 0.
            bv := stackmapdata(stkmap, 0)
            bulkBarrierBitmap(spArg, spArg, uintptr(narg), 0, bv.bytedata)
        }
    }

    // 初始化G的gobuf，保存sp，pc，任务函数等
    memclrNoHeapPointers(unsafe.Pointer(&newg.sched), unsafe.Sizeof(newg.sched))
    newg.sched.sp = sp
    newg.stktopsp = sp
    // 保存goexit的地址到sched.pc，后面会调节 goexit 作为任务函数返回后执行的地址，所以goroutine结束后会调用goexit
    newg.sched.pc = funcPC(goexit) + sys.PCQuantum // +PCQuantum so that previous instruction is in same function
    // sched.g保存当前新的G
    newg.sched.g = guintptr(unsafe.Pointer(newg))
    // 将当前的pc压入栈，保存g的任务函数为pc
    gostartcallfn(&newg.sched, fn)
    // gopc保存newproc的pc
    newg.gopc = callerpc
    // 任务函数的地址
    newg.startpc = fn.fn
    if _g_.m.curg != nil {
        newg.labels = _g_.m.curg.labels
    }
    // 判断g的任务函数是不是runtime系统的任务函数，是则sched.ngsys加1
    if isSystemGoroutine(newg) {
        atomic.Xadd(&sched.ngsys, +1)
    }
    newg.gcscanvalid = false
    // 更改当前g的状态为_Grunnable
    casgstatus(newg, _Gdead, _Grunnable)

    if _p_.goidcache == _p_.goidcacheend {
```

```
        // Sched.goidgen is the last allocated id,
        // this batch must be [sched.goidgen+1, sched.goidgen+GoidCacheBatch].
        // At startup sched.goidgen=0, so main goroutine receives goid=1.
        _p_.goidcache = atomic.Xadd64(&sched.goidgen, _GoidCacheBatch)
        _p_.goidcache -= _GoidCacheBatch - 1
        _p_.goidcacheend = _p_.goidcache + _GoidCacheBatch
    }
    // 生成唯一的goid
    newg.goid = int64(_p_.goidcache)
    _p_.goidcache++
    if raceenabled {
        newg.racectx = racegostart(callerpc)
    }
    if trace.enabled {
        // 如果启动了go trace，记录go create事件
        traceGoCreate(newg, newg.startpc)
    }

    // 将当前新生成的g，放入队列
    runqput(_p_, newg, true)

    // 如果有空闲的p 且 m没有处于自旋状态 且 main goroutine已经启动，那么唤醒某个m来执
行任务
    if atomic.Load(&sched.npidle) != 0 && atomic.Load(&sched.nmspinning) == 0 &&
mainStarted {
        // 如果还有空闲P，那么新建M来运行G
        wakep()
    }
    _g_.m.locks--
    if _g_.m.locks == 0 && _g_.preempt { // restore the preemption request in case
we've cleared it in newstack
        _g_.stackguard0 = stackPreempt
    }
}
```

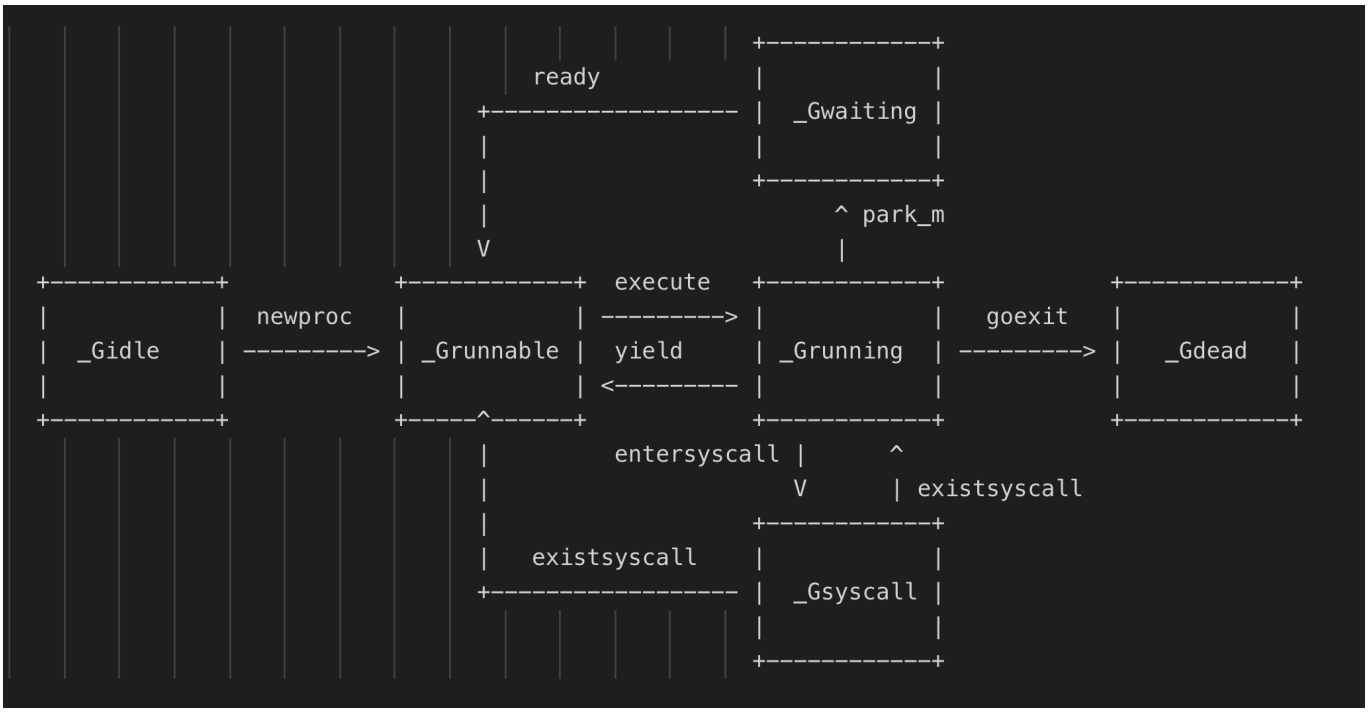
G的栈

新建g的时候都会申请好 stacksize 大小的栈，赋值给g上的 stack 字段， 并设置两个很重要的栈边界，stackguard0 和 stackguard1 。 而且g的栈是动态扩展和收缩的，但这里并不深入，知道上面两个边界即可。

```
// 新分配一个g对象，并申请好 stacksize 大小的栈，此时g的状态为_Gidle
func malg(stacksize int32) *g {
    newg := new(g)
    if stacksize >= 0 {
        stacksize = round2(_StackSystem + stacksize)
        systemstack(func() {
            // 调用 stackalloc 分配栈
            newg.stack = stackalloc(uint32(stacksize))
        })
        // 设置 stackguard
        newg.stackguard0 = newg.stack.lo + _StackGuard
```

```
        newg.stackguard1 = ^uintptr(0)
    }
    return newg
}
```

G的状态



可以看到G的状态很多，但没关系，这里先不深入去了解每个状态的切换(打算另一篇文章来介绍)，只要知道只有 _Grunnable 的G才能被M执行。 G退出的时候会做清理工作，将引用的对象都置为nil，这样对象就能被gc。这里特别强调一下 goexit ，因为当调度器执行完一个G时，并不会主动去循环调度，而是在 goexit 再次调用 schedule 来达到目的。


```
// goexit continuation on g0.
func goexit0(gp *g) {
    _g_ := getg()

    // gp的状态置为_Gdead
    casgstatus(gp, _Grunning, _Gdead)

    ...

    // 状态重置
```

```
    gp.m = nil
    // G和M是否锁定
    locked := gp.lockedm != 0
    // G和M解除锁定
    gp.lockedm = 0
    _g_.m.lockedg = 0

    gp.paniconfault = false
    gp._defer = nil // should be true already but just in case.
    gp._panic = nil // non-nil for Goexit during panic. points at stack-allocated
data.
    gp.writebuf = nil
    gp.waitreason = ""
    gp.param = nil
    gp.labels = nil
    gp.timer = nil

    ...

    // Note that gp's stack scan is now "valid" because it has no
    // stack.
    gp.gcscanvalid = true
    // 处理G和M的清除工作
    dropg()

    ...

    // 将G放入P的G空闲链表
    gfput(_g_.m.p.ptr(), gp)

    ...

    // 再次计入调度
    schedule()
}
```

关于P

P是逻辑CPU，它最主要的是本身有G的队列，保存着可运行的G。

P的结构

```
type p struct {
    lock mutex

    // id也是allp的数组下标
    id     int32
    status uint32 // one of pidle/prunning/...
    // 单向链表，指向下一个P的地址
    link puintptr
    // 每调度一次加1
```

```
    schedtick uint32 // incremented on every scheduler call
    // 每一次系统调用加1
    syscalltick uint32     // incremented on every system call
    sysmontick  sysmontick // last tick observed by sysmon
    // 回链到关联的m
    m       muintptr // back-link to associated m (nil if idle)
    mcache  *mcache
    racectx uintptr

    deferpool    [5][]*_defer // pool of available defer structs of different
sizes (see panic.go)
    deferpoolbuf [5][32]*_defer

    // Cache of goroutine ids, amortizes accesses to runtime·sched.goidgen.
    // goroutine的ID的缓存
    goidcache    uint64
    goidcacheend uint64

    // Queue of runnable goroutines. Accessed without lock.
    // 可运行的goroutine的队列
    runqhead uint32
    runqtail uint32
    runq     [256]guintptr
    // runnext, if non-nil, is a runnable G that was ready'd by
    // the current G and should be run next instead of what's in
    // runq if there's time remaining in the running G's time
    // slice. It will inherit the time left in the current time
    // slice. If a set of goroutines is locked in a
    // communicate-and-wait pattern, this schedules that set as a
    // unit and eliminates the (potentially large) scheduling
    // latency that otherwise arises from adding the ready'd
    // goroutines to the end of the run queue.
    // 下一个运行的g，优先级最高
    runnext guintptr

    // Available G's (status == Gdead)
    gfree    *g
    gfreecnt int32

    sudogcache []*sudog
    sudogbuf   [128]*sudog

    ...
}
```

P的分配

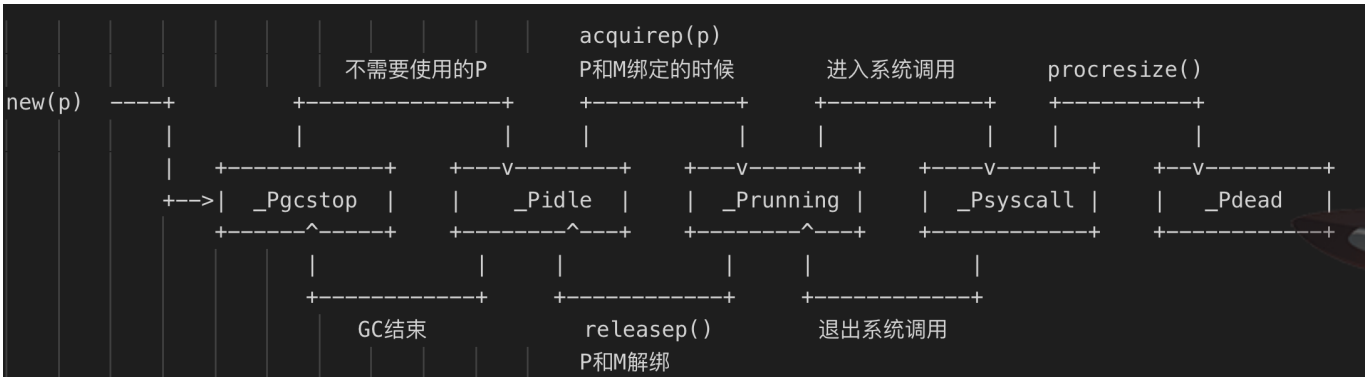
p的初始化是在 schedinit 函数中调用的，最终调用 procresize 来实现，p的分配。 p的个数默认等于系统的cpu数，如果设置了 GOMAXPROCS 环境变量，会采用环境变量设置的个数。 很多人认为runtime.GOMAXPROCS可以限制系统线程的数量，但这是错误的，M是按需创建的，和runtime.GOMAXPROCS没有直接关系。

```
func schedinit() {
    ...

    // 让procs等于cpu个数
    procs := ncpu
    // 如果环境变量设置了 GOMAXPROCS 且大于0，那么将 procs 更改
    if n, ok := atoi32(gogetenv("GOMAXPROCS")); ok && n > 0 {
        procs = n
    }
    // 调整P的个数，这里是新分配procs个P
    // 这个函数很重要，所有的P都是从这里分配的，以后也不用担心没有P了
    if procresize(procs) != nil {
        throw("unknown runnable goroutine during bootstrap")
    }

    ...
}

// 所有的P都在这个函数分配，不管是最开始的初始化分配，还是后期调整
func procresize(nprocs int32) *p {
    old := gomaxprocs
    // 如果 gomaxprocs <=0 抛出异常
    if old < 0 || nprocs <= 0 {
        throw("procresize: invalid arg")
    }
    // 如果开启trace，记录事件
    if trace.enabled {
        traceGomaxprocs(nprocs)
    }

    // update statistics
    // 更新全局状态统计
    now := nanotime()
    if sched.procresizetime != 0 {
        sched.totaltime += int64(old) * (now - sched.procresizetime)
    }
    sched.procresizetime = now

    // Grow allp if necessary.
    if nprocs > int32(len(allp)) {
        // Synchronize with retake, which could be running
        // concurrently since it doesn't run on a P.
        lock(&allpLock)
        if nprocs <= int32(cap(allp)) {
            allp = allp[:nprocs]
        } else {
            // 分配nprocs个*p
            nallp := make([]*p, nprocs)
            // Copy everything up to allp's cap so we
            // never lose old allocated Ps.
            copy(nallp, allp[:cap(allp)])
            allp = nallp
        }
```

```
            unlock(&allpLock)
        }

        ...
    }
```

所有的P在程序启动的时候就设置好了，并用一个 allp slice维护，可以调用 runtime.GOMAXPROCS 调整P的个数，但是代价很大，会 STW。

P的状态



P的状态换这里也不详细讲述，对于理解调度来说，知道M需要绑定P才能执行G即可，这个规则有一个例外，就是 sysmon ， 它不需要P，它是直接运行在M上，相当于GM模型。sysmon 更像C语言的线程上的任务函数。

关于M

M是系统线程的抽象，是一个勤劳的工作者，它拥有真正的计算资源，执行代码。

M的结构

```
type m struct {
    // 用来执行调度指令的 goroutine
    g0      *g     // goroutine with scheduling stack
    morebuf gobuf  // gobuf arg to morestack
    divmod  uint32 // div/mod denominator for arm - known to liblink

    // Fields not known to debuggers.
    procid uint64 // for debuggers, but offset not hard-coded
    // 处理信号的goroutine
    gsignal    *g           // signal-handling g
    goSigStack gsignalStack // Go-allocated signal handling stack
    sigmask    sigset       // storage for saved signal mask
    // thread-local storage
    tls      [6]uintptr // thread-local storage (for x86 extern register)
    mstartfn func()
    // 当前运行的goroutine
    curg      *g       // current running goroutine
    caughtsig guintptr // goroutine running during fatal signal
    // 关联p和执行的go代码
    p     puintptr // attached p for executing go code (nil if not executing go
```

```
code)
    nextp puintptr
    id    int64
    // 状态
    mallocing  int32
    throwing   int32
    preemptoff string // if != "", keep curg running on this m
    // locks表示该M是否被锁的状态，M被锁的状态下该M无法执行gc
    locks     int32
    softfloat int32
    dying     int32
    profilehz int32
    helpgc    int32
    // 是否自旋，自旋就表示M正在找G来运行
    spinning bool // m is out of work and is actively looking for work
    // m是否被阻塞
    blocked bool // m is blocked on a note
    // m是否在执行写屏蔽
    inwb        bool // m is executing a write barrier
    newSigstack bool // minit on C thread called sigaltstack
    printlock   int8
    // m在执行cgo吗
    incgo      bool   // m is executing a cgo call
    freeWait   uint32 // if == 0, safe to free g0 and delete m (atomic)
    fastrand   [2]uint32
    needextram bool
    traceback  uint8
    // 当前cgo调用的数目
    ncgocall uint64 // number of cgo calls in total
    // cgo调用的总数
    ncgo          int32       // number of cgo calls currently in progress
    cgoCallersUse uint32      // if non-zero, cgoCallers in use temporarily
    cgoCallers    *cgoCallers // cgo traceback if crashing in cgo call
    park          note
    // 用于链接allm
    alllink   *m // on allm
    schedlink muintptr
    // 当前m的内存缓存
    mcache *mcache
    // 锁定g在当前m上执行，而不会切换到其他m，一般cgo调用或者手动调用LockOSThread()才会
有值
    lockedg guintptr
    // thread创建的栈
    createstack [32]uintptr // stack that created this thread.
    freglo      [16]uint32  // d[i] lsb and f[i]
    freghi      [16]uint32  // d[i] msb and f[i+16]
    fflag       uint32      // floating point compare flags
    // 用户锁定M的标记
    lockedExt uint32 // tracking for external LockOSThread
    // runtime 内部锁定M的标记
    lockedInt     uint32        // tracking for internal lockOSThread
    nextwaitm     muintptr      // next m waiting for lock
    waitunlockf   unsafe.Pointer // todo go func(*g, unsafe.pointer) bool
    waitlock      unsafe.Pointer
```

```
    waittraceev   byte
    waittraceskip int
    startingtrace bool
    syscalltick   uint32
    thread        uintptr // thread handle
    freelink      *m      // on sched.freem

    // these are here because they are too large to be on the stack
    // of low-level NOSPLIT functions.
    libcall   libcall
    libcallpc uintptr // for cpu profiler
    libcallsp uintptr
    libcallg  guintptr
    syscall   libcall // stores syscall parameters on windows

    mOS
}
```

M的新建

新建M是通过 newm 来创建的，最终是通过 newosproc 函数来实现的新建线程，不通平台的 newosproc 有不同的具体实现， 比如linux平台下，是调用 clone 系统调用来实现创建线程。新建线程的任务函数为 mstart，所以当线程启动的时候，是执行 mstart 函数的代码。 mstart 上面已经讲了，这里不在赘述。

```
// Create a new m. It will start off with a call to fn, or else the scheduler.
// fn needs to be static and not a heap allocated closure.
// May run with m.p==nil, so write barriers are not allowed.
//go:nowritebarrierrec
// 创建一个新的m，它将从fn或者调度程序开始
// fn需要是静态的，而不是堆分配的闭包。
func newm(fn func(), _p_ *p) {
    // 根据fn和p和绑定一个m对象
    mp := allocm(_p_, fn)
    // 设置当前m的下一个p为_p_
    mp.nextp.set(_p_)
    mp.sigmask = initSigmask
    if gp := getg(); gp != nil && gp.m != nil && (gp.m.lockedExt != 0 ||
gp.m.incgo) && GOOS != "plan9" {
        // 我们处于锁定的M或可能由C启动的线程。此线程的内核状态可能很奇怪（用户可能已将
其锁定为此目的）。
        // 我们不想将其克隆到另一个线程中。相反，请求一个已知良好的线程为我们创建线程。
        lock(&newmHandoff.lock)
        if newmHandoff.haveTemplateThread == 0 {
            throw("on a locked thread with no template thread")
        }
        mp.schedlink = newmHandoff.newm
        newmHandoff.newm.set(mp)
        if newmHandoff.waiting {
            newmHandoff.waiting = false
            notewakeup(&newmHandoff.wake)
        }
```

```
            unlock(&newmHandoff.lock)
            return
        }
        // 真正的分配os thread
        newm1(mp)
    }

    func newm1(mp *m) {
        // 对cgo的处理
        ...

        execLock.rlock() // Prevent process clone.
        // 创建一个系统线程，并且传入该 mp 绑定的 g0 的栈顶指针
        // 让系统线程执行 mstart 函数，后面的逻辑都在 mstart 函数中
        newosproc(mp, unsafe.Pointer(mp.g0.stack.hi))
        execLock.runlock()
    }
```

linux 平台 newosproc 实现。

```
// May run with m.p==nil, so write barriers are not allowed.
//go:nowritebarrier
// 分配一个系统线程，且完成 g0 和 g0上的栈分配
// 传入 mstart 函数，让线程执行 mstart
func newosproc(mp *m, stk unsafe.Pointer) {
    ...
    // Disable signals during clone, so that the new thread starts
    // with signals disabled. It will enable them in minit.
    var oset sigset
    sigprocmask(_SIG_SETMASK, &sigset_all, &oset)
    // stk 是 g0.stack.hi，也就是说 g0 的堆栈是当前这个系统线程的堆栈，也被称为系统堆栈
    ret := clone(cloneFlags, stk, unsafe.Pointer(mp), unsafe.Pointer(mp.g0),
unsafe.Pointer(funcPC(mstart)))
    sigprocmask(_SIG_SETMASK, &oset, nil)
    ...
}
```

辛勤工作的M

M是真正的执行者，它负责整个调度的运作，上面也说了，调度的本质就是查找可以运行的G，然后去运行G上面的任务函数。 这些过程都是在 schedule 函数里实现的，这样看来可以简单理解为M就是循环的执行着 schedule 函数而已。 这里再贴一下 schedule 函数的代码。可以看到查找可运行的G是通过 findrunnable 函数来实现的。

```
func schedule() {
    _g_ := getg()

    ...
```

```
top:
    // 如果当前GC需要停止整个世界（STW），则调用gcstopm休眠当前的M
    if sched.gcwaiting != 0 {
        // 为了STW，停止当前的M
        gcstopm()
        // STW结束后回到 top
        goto top
    }

    ...

    var gp *g
    var inheritTime bool

    ...

    if gp == nil {
        // Check the global runnable queue once in a while to ensure fairness.
        // Otherwise two goroutines can completely occupy the local runqueue
        // by constantly respawning each other.
        // 每隔61次调度，尝试从全局队列种获取G
        if _g_.m.p.ptr().schedtick%61 == 0 && sched.runqsize > 0 {
            lock(&sched.lock)
            gp = globrunqget(_g_.m.p.ptr(), 1)
            unlock(&sched.lock)
        }
    }
    if gp == nil {
        // 从p的本地队列中获取
        gp, inheritTime = runqget(_g_.m.p.ptr())
        if gp != nil && _g_.m.spinning {
            throw("schedule: spinning with local work")
        }
    }
    if gp == nil {
        // 想尽办法找到可运行的G，找不到就不用返回了
        gp, inheritTime = findrunnable() // blocks until work is available
    }

    ...

    // println("execute goroutine", gp.goid)
    // 找到了g，那就执行g上的任务函数
    execute(gp, inheritTime)
}
```

findrunnable 查找G的过程：

1. 调用 runqget ， 尝试从P本地队列中获取G，获取到返回
2. 调用 globrunqget ， 尝试从全局队列中获取G，获取到返回
3. 从网络IO轮询器中找到就绪的G，把这个G变为可运行的G
4. 如果不是所有的P都是空闲的，最多四次，随机选一个P，尝试从这P中偷取一些G，获取到返回

5. 上面都找不到G来运行，判断此时P是否处于 GC mark 阶段，如果是，那么此时可以安全的扫描和黑化对象和返回 gcBgMarkWorker 来运行， gcBgMarkWorker 是GC后代标记的goroutine。
6. 再次从全局队列中获取G，获取到返回
7. 再次检查所有的P，有没有可以运行的G
8. 再次检查网络IO轮询器
9. 实在找不到可运行的G了，那就调用 stopm 休眠吧

```
// Finds a runnable goroutine to execute.
// Tries to steal from other P's, get g from global queue, poll network.
// 找到一个可以运行的G，不找到就让M休眠，然后等待唤醒，直到找到一个G返回
func findrunnable() (gp *g, inheritTime bool) {
    _g_ := getg()

    // 此处和handoffp中的条件必须一致：如果findrunnable将返回G运行，则handoffp必须启动M.
top:
    // 当前m绑定的p
    _p_ := _g_.m.p.ptr()

    ...

    // local runq
    // 再尝试从本地队列中获取G
    if gp, inheritTime := runqget(_p_); gp != nil {
        return gp, inheritTime
    }

    // global runq
    // 尝试从全局队列中获取G
    if sched.runqsize != 0 {
        lock(&sched.lock)
        gp := globrunqget(_p_, 0)
        unlock(&sched.lock)
        if gp != nil {
            return gp, false
        }
    }

    // 从网络IO轮询器中找到就绪的G，把这个G变为可运行的G
    if netpollinited() && atomic.Load(&netpollWaiters) > 0 && atomic.Load64(&sched.lastpoll) != 0 {
        if gp := netpoll(false); gp != nil { // non-blocking
            // netpoll returns list of goroutines linked by schedlink.
            // 如果找到的可运行的网络IO的G列表，则把相关的G插入全局队列
            injectglist(gp.schedlink.ptr())
            // 更改G的状态为_Grunnable，以便下次M能找到这些G来执行
            casgstatus(gp, _Gwaiting, _Grunnable)
            // goroutine trace事件记录-unpark
            if trace.enabled {
                traceGoUnpark(gp, 0)
            }
            return gp, false
```

```
        }
    }

    // Steal work from other P's.
    procs := uint32(gomaxprocs)
    // 如果其他P都是空闲的，就不从其他P哪里偷取G了
    if atomic.Load(&sched.npidle) == procs-1 {
        // Either GOMAXPROCS=1 or everybody, except for us, is idle already.
        // New work can appear from returning syscall/cgocall, network or timers.
        // Neither of that submits to local run queues, so no point in stealing.
        goto stop
    }

    // 如果当前的M没在自旋 且 空闲P的数目小于正在自旋的M个数的2倍，那么让该M进入自旋状态
    if !_g_.m.spinning && 2*atomic.Load(&sched.nmspinning) >= procs-
atomic.Load(&sched.npidle) {
        goto stop
    }

    // 如果M为非自旋，那么设置为自旋状态
    if !_g_.m.spinning {
        _g_.m.spinning = true
        atomic.Xadd(&sched.nmspinning, 1)
    }
    // 随机选一个P，尝试从这P中偷取一些G
    for i := 0; i < 4; i++ { // 尝试四次
        for enum := stealOrder.start(fastrand()); !enum.done(); enum.next() {
            if sched.gcwaiting != 0 {
                goto top
            }
            stealRunNextG := i > 2 // first look for ready queues with more than 1
g
            // 从allp[enum.position()]偷去一半的G，并返回其中的一个
            if gp := runqsteal(_p_, allp[enum.position()], stealRunNextG); gp !=
nil {
                return gp, false
            }
        }
    }

stop:

    // 当前的M找不到G来运行。如果此时P处于 GC mark 阶段
    // 那么此时可以安全的扫描和黑化对象，和返回 gcBgMarkWorker 来运行
    if gcBlackenEnabled != 0 && _p_.gcBgMarkWorker != 0 &&
gcMarkWorkAvailable(_p_) {
        // 设置gcMarkWorkerMode 为 gcMarkWorkerIdleMode
        _p_.gcMarkWorkerMode = gcMarkWorkerIdleMode
        // 获取gcBgMarkWorker goroutine
        gp := _p_.gcBgMarkWorker.ptr()
        casgstatus(gp, _Gwaiting, _Grunnable)
        if trace.enabled {
            traceGoUnpark(gp, 0)
        }
```

```
        return gp, false
    }

    // Before we drop our P, make a snapshot of the allp slice,
    // which can change underfoot once we no longer block
    // safe-points. We don't need to snapshot the contents because
    // everything up to cap(allp) is immutable.
    allpSnapshot := allp

    // return P and block
    lock(&sched.lock)
    if sched.gcwaiting != 0 || _p_.runSafePointFn != 0 {
        unlock(&sched.lock)
        goto top
    }
    // 再次从全局队列中获取G
    if sched.runqsize != 0 {
        gp := globrunqget(_p_, 0)
        unlock(&sched.lock)
        return gp, false
    }

    // 将当前对M和P解绑
    if releasep() != _p_ {
        throw("findrunnable: wrong p")
    }
    // 将p放入p空闲链表
    pidleput(_p_)
    unlock(&sched.lock)

    wasSpinning := _g_.m.spinning
    // M取消自旋状态
    if _g_.m.spinning {
        _g_.m.spinning = false
        if int32(atomic.Xadd(&sched.nmspinning, -1)) < 0 {
            throw("findrunnable: negative nmspinning")
        }
    }

    // check all runqueues once again
    // 再次检查所有的P，有没有可以运行的G
    for _, _p_ := range allpSnapshot {
        // 如果p的本地队列有G
        if !runqempty(_p_) {
            lock(&sched.lock)
            // 获取另外一个空闲P
            _p_ = pidleget()
            unlock(&sched.lock)
            if _p_ != nil {
                // 如果P不是nil，将M绑定P
                acquirep(_p_)
                // 如果是自旋，设置M为自旋
                if wasSpinning {
                    _g_.m.spinning = true
```

```
                    atomic.Xadd(&sched.nmspinning, 1)
                }
                // 返回到函数开头，从本地p获取G
                goto top
            }
            break
        }
    }

    // Check for idle-priority GC work again.
    if gcBlackenEnabled != 0 && gcMarkWorkAvailable(nil) {
        lock(&sched.lock)
        _p_ = pidleget()
        if _p_ != nil && _p_.gcBgMarkWorker == 0 {
            pidleput(_p_)
            _p_ = nil
        }
        unlock(&sched.lock)
        if _p_ != nil {
            acquirep(_p_)
            if wasSpinning {
                _g_.m.spinning = true
                atomic.Xadd(&sched.nmspinning, 1)
            }
            // Go back to idle GC check.
            goto stop
        }
    }

    // poll network
    // 再次检查netpoll
    if netpollinited() && atomic.Load(&netpollWaiters) > 0 &&
atomic.Xchg64(&sched.lastpoll, 0) != 0 {
        if _g_.m.p != 0 {
            throw("findrunnable: netpoll with p")
        }
        if _g_.m.spinning {
            throw("findrunnable: netpoll with spinning")
        }
        gp := netpoll(true) // block until new work is available
        atomic.Store64(&sched.lastpoll, uint64(nanotime()))
        if gp != nil {
            lock(&sched.lock)
            _p_ = pidleget()
            unlock(&sched.lock)
            if _p_ != nil {
                acquirep(_p_)
                injectglist(gp.schedlink.ptr())
                casgstatus(gp, _Gwaiting, _Grunnable)
                if trace.enabled {
                    traceGoUnpark(gp, 0)
                }
                return gp, false
            }
```

```
            injectglist(gp)
        }
    }
    // 实在找不到G，那就休眠吧
    // 且此时的M一定不是自旋状态
    stopm()
    goto top
}
```

M的状态

```
mstart
    |
    v
+------+       找不到可执行任务           +-------+
|unspin| ----------------------------> |spining|
|      | <---------------------------- |       |
+------+       找到可执行任务            +-------+
    ^                                       | stopm
    |               wakep                   v
notewakeup <-------------------------  notesleep
```

详见下面的解释。

M的管理

调度器的思想是尽量复用线程，它需要在保持足够多的运行工作线程以利用可用的硬件并行性和运行过多的工作线程之间取得平衡。 所以调度器就需要管理什么时候启动M，什么时候停掉M（休眠），但是需要注意的是M一旦在runtime里创建，是不会销毁的，具体可以看issue_14592。

要理解runtime对M的管理，需要先理解什么是 自旋 状态。我们上面讲了 findrunnable 函数，该函数会让M进入自旋。 如果M在本地队列、全局运行队列、netpoller中找不到工作，则认为M正在自旋，也就是M进入了一种循环找可运行G的状态。 自旋状态用 m.spinning 和 sched.nmspinning 表示。其中 m.spinning 表示当前的M是否为自旋状态，sched.nmspinning 表示runtime中一共有多少个M在自旋状态。当一个新的goroutine创建时，或者有个goroutine准备好时， runtime会根据m的自旋状态来决定是否启动新的M。我们先看一下M进入自旋状态的代码，在 findrunnable 函数中。

```
    // If number of spinning M's >= number of busy P's, block.
    // This is necessary to prevent excessive CPU consumption
    // when GOMAXPROCS>>1 but the program parallelism is low.
    //
    // 如果当前的M没在自旋 且 正在自旋的M个数的2倍>=正在忙的p的个数时，不让该M进入自旋状
态
    if !_g_.m.spinning && 2*atomic.Load(&sched.nmspinning) >= procs-
atomic.Load(&sched.npidle) {
        goto stop
    }
```

```
    // 如果M为非自旋，那么设置为自旋状态
    if !_g_.m.spinning {
        _g_.m.spinning = true
        atomic.Xadd(&sched.nmspinning, 1)
    }

    ...

    wasSpinning := _g_.m.spinning
    // M取消自旋状态
    if _g_.m.spinning {
        _g_.m.spinning = false
        if int32(atomic.Xadd(&sched.nmspinning, -1)) < 0 {
            throw("findrunnable: negative nmspinning")
        }
    }

    // check all runqueues once again
    // 再次检查所有的P，有没有可以运行的G
    for _, _p_ := range allpSnapshot {
        // 如果p的本地队列有G
        if !runqempty(_p_) {
            lock(&sched.lock)
            // 获取另外一个空闲P
            _p_ = pidleget()
            unlock(&sched.lock)
            if _p_ != nil {
                // 如果P不是nil，将M绑定P
                acquirep(_p_)
                // 如果是自旋，设置M为自旋
                if wasSpinning {
                    _g_.m.spinning = true
                    atomic.Xadd(&sched.nmspinning, 1)
                }
                // 返回到函数开头，从本地p获取G
                goto top
            }
            break
        }
    }

    ...

    // 实在找不到G，那就休眠吧
    // 且此时的M一定不是自旋状态
    stopm()
    goto top
```

从上面可以看出，并不是找不到工作（G）就都进入自旋状态，需要判断一下m本身是否已经在自旋状态和sched.nmspinning*2 >= (procs -sched.npidle)。 m如果已经处于自旋，那当然是没必要在设置为自旋状态，主要关注另外一个条件，当正在自旋的M的个数的2倍大于等于正在忙的P的个数时，不要让该M自旋。 这是为啥？注释上解释说为了避免CPU的过多的消耗。毕竟自旋是需要消耗CPU的，如果已经有不少的P处于忙时，应

该尽量让P去占用CPU资源，而不是为了查找G而浪费CPU。当然一旦自旋的M找到G后，就会自动退出自旋状态。

如果一个M找不到工作，又没有进入自旋状态，那么会调用 stopm 来休眠该M，并且会将P和M解绑，那何时会重新唤醒该M呢？这就要看 wakep 函数了。一般来说新建一个goroutine或者有个goroutine准备好时，会调用 wakep 来唤醒M或者新建M。

wakep 代码：

```
// Tries to add one more P to execute G's.
// Called when a G is made runnable (newproc, ready).
// 尝试获取一个M来运行可运行的G
func wakep() {
    // be conservative about spinning threads
    // 如果有其他的M处于自旋状态，那么就不管了，直接返回
    // 因为自旋的M会拼命找G来运行的，就不新找一个M（劳动者）来运行了。
    if !atomic.Cas(&sched.nmspinning, 0, 1) {
        return
    }
    startm(nil, true)
}
```

新建goroutine的相关代码：

```
    // 如果有空闲的p 且 m没有处于自旋状态 且 main goroutine已经启动，那么唤醒某个m来执
行任务
    if atomic.Load(&sched.npidle) != 0 && atomic.Load(&sched.nmspinning) == 0 &&
mainStarted {
        // 唤醒M或者新建M
        wakep()
    }
```

goroutine准备好相关代码：

```
// Mark gp ready to run.
// 将gp的状态更改为_Grunnable，以便调度器调度执行
// 并且如果next==true，那么设置为优先级最高，并尝试wakep
func ready(gp *g, traceskip int, next bool) {
    if trace.enabled {
        traceGoUnpark(gp, traceskip)
    }

    status := readgstatus(gp)

    // Mark runnable.
    _g_ := getg()
    _g_.m.locks++ // disable preemption because it can be holding p in a local var
    if status&^_Gscan != _Gwaiting {
```

```
            dumpgstatus(gp)
            throw("bad g->status in ready")
        }

        // status is Gwaiting or Gscanwaiting, make Grunnable and put on runq
        casgstatus(gp, _Gwaiting, _Grunnable)
        runqput(_g_.m.p.ptr(), gp, next)
        // 如果有空闲P且没有自旋的M。
        if atomic.Load(&sched.npidle) != 0 && atomic.Load(&sched.nmspinning) == 0 {
            // 唤醒M或者新建M
            wakep()
        }
        _g_.m.locks--
        if _g_.m.locks == 0 && _g_.preempt { // restore the preemption request in Case
    we've cleared it in newstack
            _g_.stackguard0 = stackPreempt
        }
    }
```

所以综上所述，当一个新的goroutine创建时，或者有个goroutine准备好时，会判断是否有M处于自旋状态，如果有的话， 是不会进行新建M来运行的，因为自旋其实意味着M是没事干的。如果没有任何一个M处于自旋，那么就需要考虑调用 startm 来尝试启动一个M了。

```
// Schedules some M to run the p (creates an M if necessary).
// If p==nil, tries to get an idle P, if no idle P's does nothing.
// May run with m.p==nil, so write barriers are not allowed.
// If spinning is set, the caller has incremented nmspinning and startm will
// either decrement nmspinning or set m.spinning in the newly started M.
//go:nowritebarrierrec
// startm是启动一个M，先尝试获取一个空闲P，如果获取不到则返回
// 获取到P后，在尝试获取M，如果获取不到就新建一个M
func startm(_p_ *p, spinning bool) {
    lock(&sched.lock)
    // 如果P为nil，则尝试获取一个空闲P
    if _p_ == nil {
        _p_ = pidleget()
        if _p_ == nil {
            unlock(&sched.lock)
            if spinning {
                // The caller incremented nmspinning, but there are no idle Ps,
                // so it's okay to just undo the increment and give up.
                if int32(atomic.Xadd(&sched.nmspinning, -1)) < 0 {
                    throw("startm: negative nmspinning")
                }
            }
            return
        }
    }
    // 获取一个空闲的M
    mp := mget()
    unlock(&sched.lock)
```

```
    if mp == nil {
        var fn func()
        if spinning {
            // The caller incremented nmspinning, so set m.spinning in the new M.
            fn = mspinning
        }
        // 如果获取不到，则新建一个，新建完成后就立即返回
        newm(fn, _p_)
        return
    }

    // 到这里表示获取到了一个空闲M
    if mp.spinning { // 从midle中获取的mp，不应该是spinning状态，获取的都是经过stopm
的，stopm之前都会推出spinning
        throw("startm: m is spinning")
    }
    if mp.nextp != 0 { // 这个位置是要留给参数_p_的，stopm中如果被唤醒，则关联nextp和m
        throw("startm: m has p")
    }
    if spinning && !runqempty(_p_) { // spinning状态的M是在本地和全局都获取不到工作的
情况，不能与spinning语义矛盾
        throw("startm: p has runnable gs")
    }
    // The caller incremented nmspinning, so set m.spinning in the new M.
    mp.spinning = spinning //标记该M是否在自旋
    mp.nextp.set(_p_)      // 暂存P
    notewakeup(&mp.park)   // 唤醒M
}
```

startm 简单来说是判断是否有空闲的P，如果没有则返回，如果有空闲的P，再尝试看有没有空闲的M，有的话， 唤醒该M起来工作。这样看起来好像M的个数会被P的个数限制，但其实不一定，因为 *p* 参数不一定为 nil， 当 *p* 不为 nil 的时候，是可能新建一个M来服务的，比如cgo调用，阻塞的系统调用等。如果要继续深究，就看一下 handoffp 函数及相关调用，这里就不展开了。

抢占的实现

抢占是在 sysmon 中实现的，这个我们之前说了，现在看一下具体是怎么实现。

cgo和syscall时，p的状态会被设置为 _Psyscall， sysmon 周期性地检查并retake p， 如果发现p处于这个状态且超过10ms就会强制性收回p，m从cgo和syscall返回后会重新尝试拿p，进入调度循环。

后台监控会先检查程序是否死锁，很多初学者跑一个go程序会报 "all goroutines are asleep - deadlock!" ，就是在这里报的。

接着sysmon 会进入一个无限循环, 第一轮回休眠20us, 之后每次休眠时间倍增, 最终每一轮都会休眠10ms。sysmon 中有netpool(获取fd事件), retake(抢占), forcegc(按时间强制执行gc), scavenge heap(释放自由列表中多余的项减少内存占用)等处理。这里关注 retake 实现，因为它负责抢占的实现，其他的可以看注释。

```
// Always runs without a P, so write barriers are not allowed.
func sysmon() {
    lock(&sched.lock)
```

```
        sched.nmsys++
        // 判断程序是否死锁
        checkdead()
        unlock(&sched.lock)

        ...

        idle := 0 // how many cycles in succession we had not wokeup somebody
        delay := uint32(0)
        for {
            if idle == 0 { // start with 20us sleep...
                delay = 20
            } else if idle > 50 { // start doubling the sleep after 1ms...
                delay *= 2
            }
            if delay > 10*1000 { // up to 10ms
                delay = 10 * 1000
            }
            // 休眠delay us
            usleep(delay)
            ...
            // poll network if not polled for more than 10ms
            lastpoll := int64(atomic.Load64(&sched.lastpoll))
            now := nanotime()
            // 如果超过10ms都没进行 netpoll ，那么强制执行一次 netpoll，
            // 并且如果获取到了可运行的G，那么插入全局列表。
            if netpollinited() && lastpoll != 0 && lastpoll+10*1000*1000 < now {
                atomic.Cas64(&sched.lastpoll, uint64(lastpoll), uint64(now))
                gp := netpoll(false) // non-blocking - returns list of goroutines
                if gp != nil {
                    // Need to decrement number of idle locked M's
                    // (pretending that one more is running) before injectglist.
                    // Otherwise it can lead to the following situation:
                    // injectglist grabs all P's but before it starts M's to run the
P's,
                    // another M returns from syscall, finishes running its G,
                    // observes that there is no work to do and no other running M's
                    // and reports deadlock.
                    incidlelocked(-1)
                    injectglist(gp)
                    incidlelocked(1)
                }
            }
            // retake P's blocked in syscalls
            // and preempt long running G's
            // 抢夺阻塞时间长的syscall的G
            if retake(now) != 0 {
                idle = 0
            } else {
                idle++
            }

            ...
```

```
    }
}
```

retake 代码：

```
func retake(now int64) uint32 {
    n := 0
    // Prevent allp slice changes. This lock will be completely
    // uncontended unless we're already stopping the world.
    lock(&allpLock)
    // We can't use a range loop over allp because we may
    // temporarily drop the allpLock. Hence, we need to re-fetch
    // allp each time around the loop.
    for i := 0; i < len(allp); i++ {
        _p_ := allp[i]
        if _p_ == nil {
            // This can happen if procresize has grown
            // allp but not yet created new Ps.
            continue
        }
        pd := &_p_.sysmontick
        s := _p_.status
        if s == _Psyscall {
            // Retake P from syscall if it's there for more than 1 sysmon tick (at
least 20us).
            t := int64(_p_.syscalltick)
            if int64(pd.syscalltick) != t {
                pd.syscalltick = uint32(t)
                pd.syscallwhen = now
                continue
            }
            // On the one hand we don't want to retake Ps if there is no other
work to do,
            // but on the other hand we want to retake them eventually
            // because they can prevent the sysmon thread from deep sleep.
            if runqempty(_p_) &&
atomic.Load(&sched.nmspinning)+atomic.Load(&sched.npidle) > 0 &&
pd.syscallwhen+10*1000*1000 > now {
                continue
            }
            // Drop allpLock so we can take sched.lock.
            unlock(&allpLock)
            // Need to decrement number of idle locked M's
            // (pretending that one more is running) before the CAS.
            // Otherwise the M from which we retake can exit the syscall,
            // increment nmidle and report deadlock.
            incidlelocked(-1)
            if atomic.Cas(&_p_.status, s, _Pidle) {
                if trace.enabled {
                    traceGoSysBlock(_p_)
                    traceProcStop(_p_)
                }
```

```
                n++
                _p_.syscalltick++
                handoffp(_p_)
            }
            incidlelocked(1)
            lock(&allpLock)
        } else if s == _Prunning {
            // Preempt G if it's running for too long.
            t := int64(_p_.schedtick)
            if int64(pd.schedtick) != t {
                pd.schedtick = uint32(t)
                pd.schedwhen = now
                continue
            }
            if pd.schedwhen+forcePreemptNS > now {
                continue
            }
            preemptone(_p_)
        }
    }
    unlock(&allpLock)
    return uint32(n)
}
```

retake 函数会遍历所有的P，如果一个P处于执行状态，且已经连续执行了较长时间，就会被抢占。 抢占的情况分为两种：

1. P处于系统调用中 当P处于系统调用状态超过10ms，那么调用 handoffp 来检查该P下是否有其他可运行的G。如果有的话， 调用 startm 来常获取一个M或新建一个M来服务。这就是为啥cgo里阻塞很久或者系统调用阻塞很久， 会导致runtime创建很多线程的原因。这里还有一个要注意的地方就是在设置 p.status之前，P已经丢弃了M， 这个细节后再讲吧，这里就不介绍了。

2. P处于正在运行中 检查G运行时间超过forcePreemptNS(10ms)，则抢占这个P，当然这个抢占不是主动的，而是被动的，通过 preemptone 设置P的 stackguard0 设为 stackPreempt 和 gp.preempt = true ，导致该P中正在执行的G进行下一次函数调用时， 导致栈空间检查失败。进而触发 morestack（汇编代码，位于asm_XXX.s中），然后进行一连串的函数调用，最终会调用 goschedImpl 函数，进行解除P与当前M的关联，让该G进入 _Grunnable 状态，插入全局G列表，等待下次调度。触发的一系列函数如下：

```
morestack() -> newstack() -> gopreempt_m() -> goschedImpl() -> schedule()
```

自此抢占原理已明了，第一种情况是为了防止系统调用的G，导致P里面的其他G无法调度，也就是饿死的状态。 第二种情况是为何防止一个G运行太长时间，而导致P里面的其他G饿死的状态。

调度跟踪信息可视化

> 开启 schedtrace

GODEBUG=schedtrace=1000 ./example // 1000ms出发一次打印

```
SCHED 0ms: gomaxprocs=4 idleprocs=2 threads=4 spinningthreads=1 idlethreads=0
runqueue=0 [0 0 0 0]
SCHED 1001ms: gomaxprocs=4 idleprocs=0 threads=8 spinningthreads=0 idlethreads=2
runqueue=0 [189 197 231 142]
SCHED 2004ms: gomaxprocs=4 idleprocs=0 threads=9 spinningthreads=0 idlethreads=1
runqueue=0 [54 45 38 86]
SCHED 3011ms: gomaxprocs=4 idleprocs=0 threads=9 spinningthreads=0 idlethreads=2
runqueue=2 [85 0 67 111]
SCHED 4018ms: gomaxprocs=4 idleprocs=3 threads=9 spinningthreads=0 idlethreads=4
runqueue=0 [0 0 0 0]
```

GODEBUG=schedtrace=1000,scheddetail=1 ./example // scheddetail表示更详细的打印信息

```
SCHED 0ms: gomaxprocs=4 idleprocs=3 threads=3 spinningthreads=0 idlethreads=0
runqueue=0 gcwaiting=0 nmidlelocked=0 stopwait=0 sysmonwait=0
P0: status=1 schedtick=0 syscalltick=0 m=0 runqsize=0 gfreecnt=0
P1: status=0 schedtick=0 syscalltick=0 m=-1 runqsize=0 gfreecnt=0
P2: status=0 schedtick=0 syscalltick=0 m=-1 runqsize=0 gfreecnt=0
P3: status=0 schedtick=0 syscalltick=0 m=-1 runqsize=0 gfreecnt=0
M2: p=-1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=1 dying=0 helpgc=0
spinning=false blocked=false lockedg=-1
M1: p=-1 curg=17 mallocing=0 throwing=0 preemptoff= locks=0 dying=0 helpgc=0
spinning=false blocked=false lockedg=17
M0: p=0 curg=1 mallocing=0 throwing=0 preemptoff= locks=2 dying=0 helpgc=0
spinning=false blocked=false lockedg=1
G1: status=2(stack growth) m=0 lockedm=0
G17: status=3() m=1 lockedm=1
G2: status=1() m=-1 lockedm=-1
```