# Goroutine调度

# **1 调度器相关数据结构**

Go的调度的实现，涉及到几个重要的数据结构。运行时库用这几个数据结构来实现goroutine的调度，管理goroutine和物理线程的运行。这些数据结构分别是结构体G，结构体M，结构体P，以及Sched结构体。前三个的定义在文件runtime/runtime.h中，而Sched的定义在runtime/proc.c中。Go语言的调度相关实现也是在文件proc.c中。

## **1.1. 结构体G**

G是goroutine的缩写，相当于操作系统中的进程控制块，在这里就是goroutine的控制结构，是对goroutine的抽象。其中包括goid是这个goroutine的ID，status是这个goroutine的状态，如Gidle,Grunnable,Grunning,Gsyscall,Gwaiting,Gdead等。

struct G

{

uintptr stackguard; // 分段栈的可用空间下界

uintptr stackbase; // 分段栈的栈基址

Gobuf sched; //进程切换时，利用sched域来保存上下文

uintptr stack0;

FuncVal\* fnstart; // goroutine运行的函数

void\* param; // 用于传递参数，睡眠时其它goroutine设置param，唤醒时此goroutine可以获取

int16 status; // 状态Gidle,Grunnable,Grunning,Gsyscall,Gwaiting,Gdead

int64 goid; // goroutine的id号

G\* schedlink;

M\* m; // for debuggers, but offset not hard-coded

M\* lockedm; // G被锁定只能在这个m上运行

uintptr gopc; // 创建这个goroutine的go表达式的pc

...

};

结构体G中的部分域如上所示。可以看到，其中包含了栈信息stackbase和stackguard，有运行的函数信息fnstart。这些就足够成为一个可执行的单元了，只要得到CPU就可以运行。

goroutine切换时，上下文信息保存在结构体的sched域中。goroutine是轻量级的线程或者称为协程，切换时并不必陷入到操作系统内核中，所以保存过程很轻量。看一下结构体G中的Gobuf，其实只保存了当前栈指针，程序计数器，以及goroutine自身。

struct Gobuf

{

// The offsets of these fields are known to (hard-coded in) libmach.

uintptr sp;

byte\* pc;

G\* g;

...

};

记录g是为了恢复当前goroutine的结构体G指针，运行时库中使用了一个常驻的寄存器extern register G\* g，这个是当前goroutine的结构体G的指针。这样做是为了快速地访问goroutine中的信息，比如，Go的栈的实现并没有使用%ebp寄存器，不过这可以通过g->stackbase快速得到。"extern register"是由6c，8c等实现的一个特殊的存储。在ARM上它是实际的寄存器；其它平台是由段寄存器进行索引的线程本地存储的一个槽位。在linux系统中，对g和m使用的分别是0(GS)和4(GS)。需要注意的是，链接器还会根据特定操作系统改变编译器的输出，例如，6l/linux下会将0(GS)重写为-16(FS)。每个链接到Go程序的C文件都必须包含runtime.h头文件，这样C编译器知道避免使用专用的寄存器。

## **1.2. 结构体M**

M是machine的缩写，是对机器的抽象，每个m都是对应到一条操作系统的物理线程。M必须关联了P才可以执行Go代码，但是当它处理阻塞或者系统调用中时，可以不需要关联P。

struct M

{

G\* g0; // 带有调度栈的goroutine

G\* gsignal; // signal-handling G 处理信号的goroutine

void (\*mstartfn)(void);

G\* curg; // M中当前运行的goroutine

P\* p; // 关联P以执行Go代码 (如果没有执行Go代码则P为nil)

P\* nextp;

int32 id;

int32 mallocing; //状态

int32 throwing;

int32 gcing;

int32 locks;

int32 helpgc; //不为0表示此m在做帮忙gc。helpgc等于n只是一个编号

bool blockingsyscall;

bool spinning;

Note park;

M\* alllink; // 这个域用于链接allm

M\* schedlink;

MCache \*mcache;

G\* lockedg;

M\* nextwaitm; // next M waiting for lock

GCStats gcstats;

...

};

这里也是截取结构体M中的部分域。和G类似，M中也有alllink域将所有的M放在allm链表中。lockedg是某些情况下，G锁定在这个M中运行而不会切换到其它M中去。M中还有一个MCache，是当前M的内存的缓存。M也和G一样有一个常驻寄存器变量，代表当前的M。同时存在多个M，表示同时存在多个物理线程。

结构体M中有两个G是需要关注一下的，一个是curg，代表结构体M当前绑定的结构体G。另一个是g0，是带有调度栈的goroutine，这是一个比较特殊的goroutine。普通的goroutine的栈是在堆上分配的可增长的栈，而g0的栈是M对应的线程的栈。所有调度相关的代码，会先切换到该goroutine的栈中再执行。

## **1.3. 结构体P**

Go1.1中新加入的一个数据结构，它是Processor的缩写。结构体P的加入是为了提高Go程序的并发度，实现更好的调度。M代表OS线程。P代表Go代码执行时需要的资源。当M执行Go代码时，它需要关联一个P，当M为idle或者在系统调用中时，它也需要P。有刚好GOMAXPROCS个P。所有的P被组织为一个数组，在P上实现了工作流窃取的调度器。

struct P

{

Lock;

uint32 status; // Pidle或Prunning等

P\* link;

uint32 schedtick; // 每次调度时将它加一

M\* m; // 链接到它关联的M (nil if idle)

MCache\* mcache;

G\* runq[256];

int32 runqhead;

int32 runqtail;

// Available G's (status == Gdead)

G\* gfree;

int32 gfreecnt;

byte pad[64];

};

结构体P中也有相应的状态：

Pidle,

Prunning,

Psyscall,

Pgcstop,

Pdead,

注意，跟G不同的是，P不存在waiting状态。MCache被移到了P中，但是在结构体M中也还保留着。在P中有一个Grunnable的goroutine队列，这是一个P的局部队列。当P执行Go代码时，它会优先从自己的这个局部队列中取，这时可以不用加锁，提高了并发度。如果发现这个队列空了，则去其它P的队列中拿一半过来，这样实现工作流窃取的调度。这种情况下是需要给调用器加锁的。

## **1.4. Sched**

Sched是调度实现中使用的数据结构，该结构体的定义在文件proc.c中。

struct Sched {

Lock;

uint64 goidgen;

M\* midle; // idle m's waiting for work

int32 nmidle; // number of idle m's waiting for work

int32 nmidlelocked; // number of locked m's waiting for work

int3 mcount; // number of m's that have been created

int32 maxmcount; // maximum number of m's allowed (or die)

P\* pidle; // idle P's

uint32 npidle; //idle P的数量

uint32 nmspinning;

// Global runnable queue.

G\* runqhead;

G\* runqtail;

int32 runqsize;

// Global cache of dead G's.

Lock gflock;

G\* gfree;

int32 stopwait;

Note stopnote;

uint32 sysmonwait;

Note sysmonnote;

uint64 lastpoll;

int32 profilehz; // cpu profiling rate

}

大多数需要的信息都已放在了结构体M、G和P中，Sched结构体只是一个壳。可以看到，其中有M的idle队列，P的idle队列，以及一个全局的就绪的G队列。Sched结构体中的Lock是非常必须的，如果M或P等做一些非局部的操作，它们一般需要先锁住调度器。

# **2 goroutine的生老病死**

本小节将通过goroutine的创建，消亡，阻塞和恢复等过程，来观察Go语言的调度策略，这里就称之为生老病死吧。整个Go语言的调度系统是比较复杂的，为了避免结构体M和结构体P引入的其它干扰，这里主要将注意力集中到结构体G中，以goroutine为主线。

## **2.1. goroutine的创建**

前面讲函数调用协议时说过go关键字最终被弄成了runtime.newproc。这就是一个goroutine的出生，所有新的goroutine都是通过这个函数创建的。

runtime.newproc(size, f, args)功能就是创建一个新的g，这个函数不能用分段栈，因为它假设参数的放置顺序是紧接着函数f的（见前面函数调用协议一章，有关go关键字调用时的内存布局）。分段栈会破坏这个布局，所以在代码中加入了标记#pragma textflag 7表示不使用分段栈。它会调用函数newproc1，在newproc1中可以使用分段栈。真正的工作是调用newproc1完成的。newproc1进行下面这些动作。

首先，它会检查当前结构体M中的P中，是否有可用的结构体G。如果有，则直接从中取一个，否则，需要分配一个新的结构体G。如果分配了新的G，需要将它挂到runtime的相关队列中。

获取了结构体G之后，将调用参数保存到g的栈，将sp，pc等上下文环境保存在g的sched域，这样整个goroutine就准备好了，整个状态和一个运行中的goroutine被中断时一样，只要等分配到CPU，它就可以继续运行。

newg->sched.sp = (uintptr)sp;

newg->sched.pc = (byte\*)runtime·goexit;

newg->sched.g = newg;

runtime·gostartcallfn(&newg->sched, fn);

newg->gopc = (uintptr)callerpc;

newg->status = Grunnable;

newg->goid = runtime·xadd64(&runtime·sched.goidgen, 1);

然后将这个“准备好”的结构体G挂到当前M的P的队列中。这里会给予新的goroutine一次运行的机会，即：如果当前的P的数目没有到上限，也没有正在自旋抢CPU的M，则调用wakep将P立即投入运行。

wakep函数唤醒P时，调度器会试着寻找一个可用的M来绑定P，必要的时候会新建M。让我们看看新建M的函数newm：

// 新建一个m，它将以调用fn开始，或者是从调度器开始

static void

newm(void(\*fn)(void), P \*p)

{

M \*mp;

mp = runtime·allocm(p);

mp->nextp = p;

mp->mstartfn = fn;

runtime·newosproc(mp, (byte\*)mp->g0->stackbase);

}

runtime.newm功能跟newproc相似,前者分配一个goroutine,而后者分配一个M。其实一个M就是一个操作系统线程的抽象，可以看到它会调用runtime.newosproc。

总算看到了从Go的运行时库到操作系统的接口，runtime.newosproc(平台相关的)会调用系统的runtime.clone(平台相关的)来新建一个线程，新的线程将以runtime.mstart为入口函数。runtime.newosproc是个很有意思的函数，还有一些信号处理方面的细节，但是对鉴于我们是专注于调度方面，就不对它进行更细致的分析了，感兴趣的读者可以自行去runtime/os\_linux.c看看源代码。runtime.clone是用汇编实现的,代码在sys\_linux\_amd64.s。

既然线程是以runtime.mstart为入口的，那么接下来看mstart函数。

mstart是runtime.newosproc新建的系统线程的入口地址，新线程执行时会从这里开始运行。新线程的执行和goroutine的执行是两个概念，由于有m这一层对机器的抽象，是m在执行g而不是线程在执行g。所以线程的入口是mstart，g的执行要到schedule才算入口。函数mstart最后调用了schedule。

终于到了schedule了！

如果是从mstart进入到schedule的，那么schedule中逻辑非常简单，大概就这几步：

找到一个等待运行的g

如果g是锁定到某个M的，则让那个M运行

否则，调用execute函数让g在当前的M中运行

execute会恢复newproc1中设置的上下文，这样就跳转到新的goroutine去执行了。从newproc出生一直到运行的过程分析，到此结束!

虽然按这样a调用b，b调用c，c调用d，d调用e的方式去分析源代码谁看都会晕掉，但还是要重复一遍这里的读代码过程，希望感兴趣的读者可以拿着注释过的源码按顺序走一遍：

newproc -> newproc1 -> (如果P数目没到上限)wakep -> startm -> (可能引发)newm -> newosproc -> (线程入口)mstart -> schedule -> execute -> goroutine运行

## **2.2. 进出系统调用**

假设goroutine"生病"了，它要进入系统调用了，暂时无法继续执行。进入系统调用时，如果系统调用是阻塞的，goroutine会被剥夺CPU，将状态设置成Gsyscall后放到就绪队列。Go的syscall库中提供了对系统调用的封装，它会在真正执行系统调用之前先调用函数.entersyscall，并在系统调用函数返回后调用.exitsyscall函数。这两个函数就是通知Go的运行时库这个goroutine进入了系统调用或者完成了系统调用，调度器会做相应的调度。

比如syscall包中的Open函数，它会调用Syscall(SYS\_OPEN, uintptr(unsafe.Pointer(\_p0)), uintptr(mode), uintptr(perm))实现。这个函数是用汇编写的，在syscall/asm\_linux\_amd64.s中可以看到它的定义：

TEXT ·Syscall(SB),7,$0

CALL runtime·entersyscall(SB)

MOVQ 16(SP), DI

MOVQ 24(SP), SI

MOVQ 32(SP), DX

MOVQ $0, R10

MOVQ $0, R8

MOVQ $0, R9

MOVQ 8(SP), AX // syscall entry

SYSCALL

CMPQ AX, $0xfffffffffffff001

JLS ok

MOVQ $-1, 40(SP) // r1

MOVQ $0, 48(SP) // r2

NEGQ AX

MOVQ AX, 56(SP) // errno

CALL runtime·exitsyscall(SB)

RET

ok:

MOVQ AX, 40(SP) // r1

MOVQ DX, 48(SP) // r2

MOVQ $0, 56(SP) // errno

CALL runtime·exitsyscall(SB)

RET

可以看到它进系统调用和出系统调用时分别调用了runtime.entersyscall和runtime.exitsyscall函数。那么，这两个函数做什么特殊的处理呢？

首先，将函数的调用者的SP,PC等保存到结构体G的sched域中。同时，也保存到g->gcsp和g->gcpc等，这个是跟垃圾回收相关的。

然后检查结构体Sched中的sysmonwait域，如果不为0，则将它置为0，并调用runtime·notewakeup(&runtime·sched.sysmonnote)。做这这一步的原因是，目前这个goroutine要进入Gsyscall状态了，它将要让出CPU。如果有人在等待CPU的话，会通知并唤醒等待者，马上就有CPU可用了。

接下来，将m的MCache置为空，并将m->p->m置为空，表示进入系统调用后结构体M是不需要MCache的，并且P也被剥离了，将P的状态设置为PSyscall。

有一个与entersyscall函数稍微不同的函数叫entersyscallblock，它会告诉提示这个系统调用是会阻塞的，因此会有一点点区别。它调用的releasep和handoffp。

releasep将P和M完全分离，使p->m为空，m->p也为空，剥离m->mcache，并将P的状态设置为Pidle。注意这里的区别，在非阻塞的系统调用entersyscall中只是设置成Psyscall，并且也没有将m->p置为空。

handoffp切换P。将P从处于syscall或者locked的M中，切换出来交给其它M。每个P中是挂了一个可执行的G的队列的，如果这个队列不为空，即如果P中还有G需要执行，则调用startm让P与某个M绑定后立刻去执行，否则将P挂到idlep队列中。

出系统调用时会调用到runtime·exitsyscall，这个函数跟进系统调用做相反的操作。它会先检查当前m的P和它状态，如果P不空且状态为Psyscall，则说明是从一个非阻塞的系统调用中返回的，这时是仍然有CPU可用的。因此将p->m设置为当前m，将p的mcache放回到m，恢复g的状态为Grunning。否则，它是从一个阻塞的系统调用中返回的，因此之前m的P已经完全被剥离了。这时会查看调用中是否还有idle的P，如果有，则将它与当前的M绑定。

如果从一个阻塞的系统调用中出来，并且出来的这一时刻又没有idle的P了，要怎么办呢？这种情况代码当前的goroutine无法继续运行了，调度器会将它的状态设置为Grunnable，将它挂到全局的就绪G队列中，然后停止当前m并调用schedule函数。

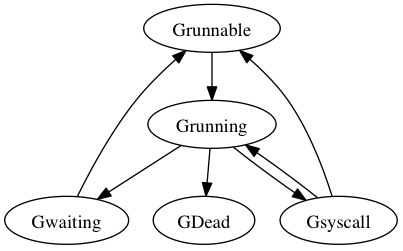
## **2.3. goroutine的消亡以及状态变化**

goroutine的消亡比较简单，注意在函数newproc1，设置了fnstart为goroutine执行的函数，而将新建的goroutine的sched域的pc设置为了函数runtime.exit。当fnstart函数执行完返回时，它会返回到runtime.exit中。这时Go就知道这个goroutine要结束了，runtime.exit中会做一些回收工作，会将g的状态设置为Gdead等，并将g挂到P的free队列中。

从以上的分析中，其实已经基本上经历了goroutine的各种状态变化。在newproc1中新建的goroutine被设置为Grunnable状态，投入运行时设置成Grunning。在entersyscall的时候goroutine的状态被设置为Gsyscall，到出系统调用时根据它是从阻塞系统调用中出来还是非阻塞系统调用中出来，又会被设置成Grunning或者Grunnable的状态。在goroutine最终退出的runtime.exit函数中，goroutine被设置为Gdead状态。

等等，好像缺了什么？是的，Gidle始终没有出现过。这个状态好像实际上没有被用到。只有一个runtime.park函数会使goroutine进入到Gwaiting状态，但是park这个有什么作用我暂时还没看懂...

goroutine的状态变迁图：



# **3 设计与演化**

其实讲一个东西，讲它是什么样是不足够的。如果能讲清楚它为什么会是这样子，则会举一反三。为了理解goroutine的本质，这里将从最基本的线程池讲起，谈谈Go调度设计背后的故事，讲清楚它为什么是这样子。

## **3.1. 线程池**

先看一些简单点的吧。一个常规的 线程池+任务队列 的模型如图所示：



把每个工作线程叫worker的话，每条线程运行一个worker，每个worker做的事情就是不停地从队列中取出任务并执行：

while(!empty(queue)) {

q = get(queue); //从任务队列中取一个(涉及加锁等)

q->callback(); //执行该任务

}

当然，这是最简单的情形，但是一个很明显的问题就是一个进入callback之后，就失去了控制权。因为没有一个调度器层的东西，一个任务可以执行很长很长时间一直占用的worker线程，或者阻塞于io之类的。

也许用Go语言表述会更地道一些。好吧，那么让我们用Go语言来描述。假设我们有一些“任务”，任务是一个可运行的东西，也就是只要满足Run函数，它就是一个任务。所以我们就把这个任务叫作接口G吧。

type G interface {

Run()

}

我们有一个全局的任务队列，里面包含很多可运行的任务。线程池的各个线程从全局的任务队列中取任务时，显然是需要并发保护的，所以有下面这个结构体：

type Sched struct {

allg []G

lock \*sync.Mutex

}

以及它的变量

var sched Sched

每条线程是一个worker，这里我们给worker换个名字，就把它叫M吧。前面已经说过了，worker做的事情就是不停的去任务队列中取一个任务出来执行。于是用Go语言大概可以写成这样子：

func M() {

for {

sched.lock.Lock() //互斥地从就绪G队列中取一个g出来运行

if sched.allg > 0 {

g := sched.allg[0]

sched.allg = sched.allg[1:]

sched.lock.Unlock()

g.Run() //运行它

} else {

sched.lock.Unlock()

}

}

}

接下来，将整个系统启动：

for i:=0; i<GOMAXPROCS; i++ {

go M()

}

假定我们有一个满足G接口的main，然后它在自己的Run中不断地将新的任务挂到sched.allg中，这个线程池+任务队列的系统模型就会一直运行下去。

可以看到，这里在代码取中故意地用Go语言中的G，M，甚至包括GOMAXPROCS等取名字。其实本质上，Go语言的调度层无非就是这样一个工作模式的：几条物理线程，不停地取goroutine运行。

## **3.2. 系统调用**

上面的情形太简单了，就是工作线程不停地取goroutine运行，这个还不能称之为调度。调度之所以为调度，是因为有一些复杂的控制机制，比如哪个goroutine应该被运行，它应该运行多久，什么时候将它换出来。用前面的代码来说明Go的调度会有一些小问题。Run函数会一直执行，在它结束之前不会返回到调用器层面。那么假设上面的任务中Run进入到一个阻塞的系统调用了，那么M也就跟着一起阻塞了，实际工作的线程就少了一个，无法充分利用CPU。

一个简单的解决办法是在进入系统调用之前再制造一个M出来干活，这样就填补了这个进入系统调用的M的空缺，始终保证有GOMAXPROCS个工作线程在干活了。

func entersyscall() {

go M()

}

那么出系统调用时怎么办呢？如果让M接着干活，岂不超过了GOMAXPROCS个线程了？所以这个M不能再干活了，要限制干活的M个数为GOMAXPROCS个，多了则让它们闲置(物理线程比CPU多很多就没意义了，让它们相互抢CPU反而会降低利用率)。

func exitsyscall() {

if len(allm) >= GOMAXPROCS {

sched.lock.Lock()

sched.allg = append(sched.allg, g) //把g放回到队列中

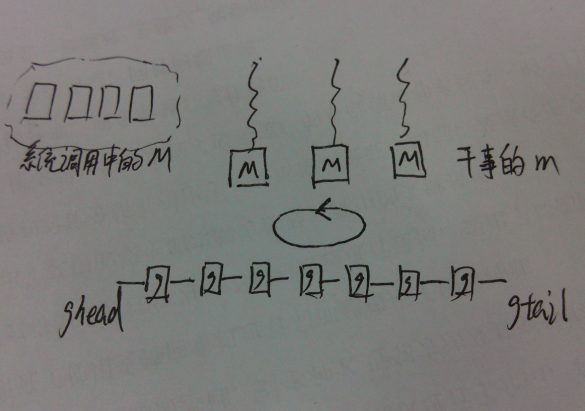
sched.lock.Unlock()

time.Sleep() //这个M不再干活

}

}

于是就变成了这样子:



其实这个也很好理解，就像线程池做负载调节一样，当任务队列很长后，忙不过来了，则再开几条线程出来。而如果任务队列为空了，则可以释放一些线程。

## **3.3. 协程与保存上下文**

大家都知道阻塞于系统调用，会白白浪费CPU。而使用异步事件或回调的思维方式又十分反人类。上面的模型既然这么简单明了，为什么不这么用呢？其实上面的东西看上去简单，但实现起来确不那么容易。

将一个正在执行的任务yield出去，再在某个时刻再弄回来继续运行，这就涉及到一个麻烦的问题，即保存和恢复运行时的上下文环境。

在此先引入协程的概念。协程是轻量级的线程，它相对线程的优势就在于协程非常轻量级，进行切换以及保存上下文环境代价非常的小。协程的具体的实现方式有多种，上面就是其中一种基于线程池的实现方式。每个协程是一个任务，可以保存和恢复任务运行时的上下文环境。

协程一类的东西一般会提供类似yield的函数。协程运行到一定时候就主动调用yield放弃自己的执行，把自己再次放回到任务队列中等待下一次调用时机等等。

其实Go语言中的goroutine就是协程。每个结构体G中有一个sched域就是用于保存自己上下文的。这样，这种goroutine就可以被换出去，再换进来。这种上下文保存在用户态完成，不必陷入到内核，非常的轻量，速度很快。保存的信息很少，只有当前的PC,SP等少量信息。只是由于要优化，所以代码看上去更复杂一些，比如要重用内存空间所以会有gfree和mhead之类的东西。

## **3.4. Go1.0**

在前面的代码中，线程与M是直接对应的关系，这个解耦还是不够。Go1.0中将M抽出来成为了一个结构体，startm函数是线程的入口地址，而goroutine的入口地址是go表达式中的那个函数。总体上跟上面的结构差不多，进出系统调用的时候goroutine会跟M一起进入到系统调用中，schedule中会匹配g和m，让空闲的m来运行g。如果检测到干活的数量少于GOMAXPROCS并且没有空闲着的m，则会创建新的m来运行g。出系统调用的时候，如果已经有GOMAXPROCS个m在干活了，则这个出系统调用的m会被挂起，它的g也会被挂到待运行的goroutine队列中。

在Go语言中m是machine的缩写，也就是机器的抽象。它被设计成了可以运行所有的G。比如说一个g开始在某个m上运行，经过几次进出系统调用之后，可能运行它的m挂起了，其它的m会将它从队列中取出并继续运行。

每次调度都会涉及对g和m等队列的操作，这些全局的数据在多线程情况下使用就会涉及到大量的锁操作。在频繁的系统调用中这将是一个很大的开销。为了减少系统调用开销，Go1.0在这里做了一些优化的。1.0版中，在它的Sched结构体中有一个atomic字段，类型是一个volatile的无符32位整型。

// sched中的原子字段是一个原子的uint32，存放下列域

// 15位 mcpu --正在占用cpu运行的m数量 (进入syscall的m是不占用cpu的)

// 15位 mcpumax --最大允许这么多个m同时使用cpu

// 1位 waitstop --有g等待结束

// 1位 gwaiting --等待队列不为空，有g处于waiting状态

// [15 bits] mcpu number of m's executing on cpu

// [15 bits] mcpumax max number of m's allowed on cpu

// [1 bit] waitstop some g is waiting on stopped

// [1 bit] gwaiting gwait != 0

这些信息是进行系统调用和出系统调用时需要用到的，它会决定是否需要进入到调度器层面。直接用CAS操作Sched的atomic字段判断，将它们打包成一个字节使得可以通过一次原子读写获取它们而不用加锁。这将极大的减少那些大量使用系统调用或者cgo的多线程程序的contention。

除了进出系统调用以外，操作这些域只会发生于持有调度器锁的时候，因此goroutines不用担心其它goroutine会对这些字段进行操作。特别是，进出系统调用只会读mcpumax，waitstop和gwaiting。决不会写他们。因此，(持有调度器锁)写这些域时完全不用担心会发生写冲突。

总体上看，Go1.0调度设计结构比较简单，代码也比较清晰。但是也存在一些问题。这样的调度器设计限制了Go程序的并发度。测试发现有14%是的时间浪费在了runtime.futex()中。

具体地看：

1. 单个全局锁(Sched.Lock)用来保护所有的goroutine相关的操作(创建，完成，调度等)。
2. Goroutine切换。工作线程在各自之前切换goroutine，这导致延迟和额外的负担。每个M都必须可以执行任何的G.
3. 内存缓存MCache是每个M的。而当M阻塞后，相应的内存资源也被一起拿走了。
4. 过多的线程阻塞、恢复。系统调用时的工作线程会频繁地阻塞，恢复，造成过多的负担。

第一点很明显，所有的goroutine都用一个锁保护的，这个锁粒度是比较大的，只要goroutine的相关操作都会锁住调度。然后是goroutine切换，前面说了，每个M都是可以执行所有的goroutine的。举个很简单的类比，多核CPU中每个核都去执行不同线程的代码，这显然是不利于缓存的局部性的，切换开销也会变大。内存缓存和其它缓存是关联到所有的M的，而事实上它本只需要关联到运行Go代码的M(阻塞于系统调用的M是不需要mcache的)。运行着Go代码的M和所有M的比例可能高达1:100。这导致过度的资源消耗。

## **3.5. Go1.1**

Go1.1相对于1.0一个重要的改动就是重新调用了调度器。前面已经看到，老版本中的调度器实现是存在一些问题的。解决方式是引入Processor的概念，并在Processors之上实现工作流窃取的调度器。

M代表OS线程。P代表Go代码执行时需要的资源。当M执行Go代码时，它需要关联一个P，当M为idle或者在系统调用中时，它也需要P。有刚好GOMAXPROCS个P。所有的P被组织为一个数组，工作流窃取需要这个条件。GOMAXPROCS的改变涉及到stop/start the world来resize数组P的大小。

gfree和grunnable从sched中移到P中。这样就解决了前面的单个全局锁保护用有goroutine的问题，由于goroutine现在被分到每个P中，它们是P局部的goroutine，因此P只管去操作自己的goroutine就行了，不会与其它P上的goroutine冲突。全局的grunnable队列也仍然是存在的，只有在P去访问全局grunnable队列时才涉及到加锁操作。mcache从M中移到P中。不过当前还不彻底，在M中还是保留着mcache域的。

加入了P后，sched.atomic也从Sched结构体中去掉了。

当一个新的G创建或者现有的G变成runnable，它将一个runnable的goroutine推到当前的P。当P完成执行G，它将G从自己的runnable goroutine中pop出去。如果链为空，P会随机从其它P中窃取一半的可运行的goroutine。

当M创建一个新G的时候，必须保证有另一个M来执行这个G。类似的，当一个M进入到系统调用时，必须保证有另一个M来执行G的代码。

2层自旋：关联了P的处于idle状态的的M自旋寻找新的G；没有关联P的M自旋等待可用的P。最多有GOMAXPROCS个自旋的M。只要有第二类M时第一类M就不会阻塞。

# **4 抢占式调度**

goroutine本来是设计为协程形式，但是随着调度器的实现越来越成熟，Go在1.2版中开始引入比较初级的抢占式调度。

## **4.1. 从一个bug说起**

Go在设计之初并没考虑将goroutine设计成抢占式的。用户负责让各个goroutine交互合作完成任务。一个goroutine只有在涉及到加锁，读写通道或者主动让出CPU等操作时才会触发切换。

垃圾回收器是需要stop the world的。如果垃圾回收器想要运行了，那么它必须先通知其它的goroutine合作停下来，这会造成较长时间的等待时间。考虑一种很极端的情况，所有的goroutine都停下来了，只有其中一个没有停，那么垃圾回收就会一直等待着没有停的那一个。

抢占式调度可以解决这种问题，在抢占式情况下，如果一个goroutine运行时间过长，它就会被剥夺运行权。

## **4.2. 总体思路**

引入抢占式调度，会对最初的设计产生比较大的影响，Go还只是引入了一些很初级的抢占，并没有像操作系统调度那么复杂，没有对goroutine分时间片，设置优先级等。

只有长时间阻塞于系统调用，或者运行了较长时间才会被抢占。runtime会在后台有一个检测线程，它会检测这些情况，并通知goroutine执行调度。

目前并没有直接在后台的检测线程中做处理调度器相关逻辑，只是相当于给goroutine加了一个“标记”，然后在它进入函数时才会触发调度。这么做应该是出于对现有代码的修改最小的考虑。

## **4.3. sysmon**

前面讲Go程序的初始化过程中有提到过，runtime开了一条后台线程，运行一个sysmon函数。这个函数会周期性地做epoll操作，同时它还会检测每个P是否运行了较长时间。

如果检测到某个P状态处于Psyscall超过了一个sysmon的时间周期(20us)，并且还有其它可运行的任务，则切换P。

如果检测到某个P的状态为Prunning，并且它已经运行了超过10ms，则会将P的当前的G的stackguard设置为StackPreempt。这个操作其实是相当于加上一个标记，通知这个G在合适时机进行调度。

目前这里只是尽最大努力送达，但并不保证收到消息的goroutine一定会执行调度让出运行权。

## **4.4. morestack的修改**

前面说的，将stackguard设置为StackPreempt实际上是一个比较trick的代码。我们知道Go会在每个函数入口处比较当前的栈寄存器值和stackguard值来决定是否触发morestack函数。

将stackguard设置为StackPreempt作用是进入函数时必定触发morestack，然后在morestack中再引发调度。

看一下StackPreempt的定义，它是大于任何实际的栈寄存器的值的：

// 0xfffffade in hex.

#define StackPreempt ((uint64)-1314)

然后在morestack中加了一小段代码，如果发现stackguard为StackPreempt，则相当于调用runtime.Gosched。

所以，到目前为止Go的抢占式调度还是很初级的，比如一个goroutine运行了很久，但是它并没有调用另一个函数，则它不会被抢占。当然，一个运行很久却不调用函数的代码并不是多数情况。