深入Golang调度器之GMP模型

前言

随着服务器硬件迭代升级，配置也越来越高。为充分利用服务器资源，并发编程也变的越来越重要。在开始之前，需要了解一下并发(concurrency)和并行(parallesim)的区别。

并发: 逻辑上具有处理多个同时性任务的能力。

并行: 物理上同一时刻执行多个并发任务。

通常所说的并发编程，也就是说它允许多个任务同时执行，但实际上并不一定在同一时刻被执 行。在单核处理器上，通过多线程共享CPU时间片串行执行(并发非并行)。而并行则依赖于多核处理器等物理资源，让多个任务可以实现并行执行(并发且并行)。

多线程或多进程是并行的基本条件，但单线程也可以用协程(coroutine)做到并发。简单将Goroutine归纳为协程并不合适，因为它运行时会创建多个线程来执行并发任务，且任务单元可被调度到其它线程执行。这更像是多线程和协程的结合体，能最大限度提升执行效率，发挥多核处理器能力。

Go编写一个并发编程程序很简单，只需要在函数之前使用一个Go关键字就可以实现并发编程。

```
func main() {       go func(){              fmt.Println("Hello,World!")       }() }
```

Go调度器组成

Go语言虽然使用一个Go关键字即可实现并发编程，但Goroutine被调度到后端之后，具体的实现比较复杂。先看看调度器有哪几部分组成。

1、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。其中包括执行的函数指令及参数；G保存的任务对象；线程上下文切换，现场保护和现场恢复需要的寄存器(SP、IP)等信息。

Go不同版本Goroutine默认栈大小不同。

```
// Go1.11版本默认stack大小为2KB

_StackMin = 2048
// 创建一个g对象,然后放到g队列
// 等待被执行
func newproc1(fn *funcval, argp *uint8, narg int32, callergp *g, callerpc uintptr) {
  _g_ := getg()       _g_.m.locks++       siz := narg       siz = (siz + 7) &^ 7       _p_ :=
_g_.m.p.ptr()       newg := gfget(_p_)
     if newg == nil {
```

```
            // 初始化g stack大小                    newg = malg(_StackMin)
  casgstatus(newg, _Gidle, _Gdead)            allgadd(newg)      }
    // 以下省略}
```

2、M

M是一个线程或称为Machine，所有M是有线程栈的。如果不对该线程栈提供内存的话，系统会给该线程栈提供内存(不同操作系统提供的线程栈大小不同)。当指定了线程栈，则M.stack→G.stack，M的PC寄存器指向G提供的函数，然后去执行。

```
type m struct {
    /*            1.    所有调用栈的Goroutine,这是一个比较特殊的Goroutine。                2.
  普通的Goroutine栈是在Heap分配的可增长的stack,而g0的stack是M对应的线程栈。                3.
  所有调度相关代码,会先切换到该Goroutine的栈再执行。        */       g0          *g
  curg       *g              // M当前绑定的结构体G       // SP、PC寄存器用于现场保护和现场
恢复       vdsoSP uintptr       vdsoPC uintptr       // 省略…}
```

3、P

P(Processor)是一个抽象的概念，并不是真正的物理CPU。所以当P有任务时需要创建或者唤醒一个系统线程来执行它队列里的任务。所以P/M需要进行绑定，构成一个执行单元。

P决定了同时可以并发任务的数量，可通过GOMAXPROCS限制同时执行用户级任务的操作系统线程。可以通过runtime.GOMAXPROCS进行指定。在Go1.5之后GOMAXPROCS被默认设置可用的核数，而之前则默认为1。

```
// 自定义设置GOMAXPROCS数量
func GOMAXPROCS(n int) int {
    /*            1.    GOMAXPROCS设置可执行的CPU的最大数量,同时返回之前的设置。
  2.    如果n < 1,则不更改当前的值。        */       ret := int(gomaxprocs)
  stopTheWorld("GOMAXPROCS")
    // startTheWorld启动时,使用newprocs。       newprocs = int32(n)       startTheWorld()

    return ret }
// 默认P被绑定到所有CPU核上
// P == cpu.cores

func getproccount() int32 {
    const maxCPUs = 64 * 1024       var buf [maxCPUs / 8]byte       // 获取CPU Core       r
:= sched_getaffinity(0, unsafe.Sizeof(buf), &buf[0])       n := int32(0)
    for _, v := range buf[:r] {
        for v != 0 {                    n += int32(v & 1)                    v >>= 1
    }       }
    if n == 0 {          n = 1       }
    return n }
// 一个进程默认被绑定在所有CPU核上,返回所有CPU core。
```

```
// 获取进程的CPU亲和性掩码系统调用
// rax 204                                          ; 系统调用码
// system_call sys_sched_getaffinity; 系统调用名称
// rid   pid                                        ; 进程号
// rsi unsigned int len
// rdx unsigned long *user_mask_ptr
sys_linux_amd64.s:   TEXT   runtime·sched_getaffinity(SB),NOSPLIT,$0              MOVQ
  pid+0(FP), DI       MOVQ       len+8(FP), SI       MOVQ       buf+16(FP), DX       MOVL
  $SYS_sched_getaffinity, AX       SYSCALL       MOVL       AX, ret+24(FP)       RET
```

Go调度器调度过程

首先创建一个G对象，G对象保存到P本地队列或者是全局队列。P此时去唤醒一个M。P继续执行它的执行序。M寻找是否有空闲的P，如果有则将该G对象移动到它本身。接下来M执行一个调度循环(调用G对象->执行->清理线程→继续找新的Goroutine执行)。
M执行过程中，随时会发生上下文切换。当发生上线文切换时，需要对执行现场进行保护，以便下次被调度执行时进行现场恢复。Go调度器M的栈保存在G对象上，只需要将M所需要的寄存器(SP、PC等)保存到G对象上就可以实现现场保护。当这些寄存器数据被保护起来，就随时可以做上下文切换了，在中断之前把现场保存起来。如果此时G任务还没有执行完，M可以将任务重新丢到P的任务队列，等待下一次被调度执行。当再次被调度执行时，M通过访问G的vdsoSP、vdsoPC寄存器进行现场恢复(从上次中断位置继续执行)。

1、P 队列

通过上图可以发现，P有两种队列：本地队列和全局队列。

- **本地队列：** 当前P的队列，本地队列是Lock-Free，没有数据竞争问题，无需加锁处理，可以提升处理速度。
- **全局队列：**全局队列为了保证多个P之间任务的平衡。所有M共享P全局队列，为保证数据竞争问题，需要加锁处理。相比本地队列处理速度要低于全局队列。

2、**上线文切换**

简单理解为当时的环境即可，环境可以包括当时程序状态以及变量状态。例如线程切换的时候在内核会发生上下文切换，这里的上下文就包括了当时寄存器的值，把寄存器的值保存起来，等下次该线程又得到cpu时间的时候再恢复寄存器的值，这样线程才能正确运行。
对于代码中某个值说，上下文是指这个值所在的局部(全局)作用域对象。相对于进程而言，上下文就是进程执行时的环境，具体来说就是各个变量和数据，包括所有的寄存器变量、进程打开的文件、内存(堆栈)信息等。

3、线程清理

Goroutine被调度执行必须保证P/M进行绑定，所以线程清理只需要将P释放就可以实现线程的清理。什么时候P会释放，保证其它G可以被执行。P被释放主要有两种情况。

- **主动释放**：最典型的例子是，当执行G任务时有系统调用，当发生系统调用时M会处于Block状态。调度器会设置一个超时时间，当超时时会将P释放。
- **被动释放**：如果发生系统调用，有一个专门监控程序，进行扫描当前处于阻塞的P/M组合。当超过系统程序设置的超时时间，会自动将P资源抢走。去执行队列的其它G任务。

终于要来说说Golang中最吸引人的goroutine了，这也是Golang能够横空出世的主要原因。不同于Python基于进程的并发模型，以及C++、Java等基于线程的并发模型。Golang采用轻量级的goroutine来实现并发，可以大大减少CPU的切换。现在已经有太多的文章来介绍goroutine的用法，在这里，我们从源码的角度来看看其内部实现。

重申一下重点：goroutine中的三个实体

goroutine中最主要的是三个实体为GMP，其中：

`G:` 代表一个goroutine对象，每次go调用的时候，都会创建一个G对象，它包括栈、指令指针以及对于调用goroutines很重要的其它信息，比如阻塞它的任何channel，其主要数据结构：

```
type g struct { stack stack // 描述了真实的栈内存，包括上下界 m *m // 当前的m sched gobuf // goroutine切换时，用于保存g的上下文 param unsafe.Pointer // 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 atomicstatus uint32 stackLock uint32 goid int64 // goroutine的ID waitsince int64 // g被阻塞的大体时间 lockedm *m // G被锁定只在这个m上运行 }
```

其中最主要的当然是sched了，保存了goroutine的上下文。goroutine切换的时候不同于线程有OS来负责这部分数据，而是由一个gobuf对象来保存，这样能够更加轻量级，再来看看gobuf的结构：

```
type gobuf struct { sp uintptr pc uintptr g guintptr ctxt unsafe.Pointer ret sys.Uintreg lr uintptr bp uintptr // for GOEXPERIMENT=framepointer }
```

其实就是保存了当前的栈指针，计数器，当然还有g自身，这里记录自身g的指针是为了能快速的访问到goroutine中的信息。

`M:` 代表一个线程，每次创建一个M的时候，都会有一个底层线程创建；所有的G任务，最终还是在M上执行，其主要数据结构：

```
type m struct { g0 *g // 带有调度栈的goroutine gsignal *g // 处理信号的goroutine tls [6]uintptr // thread-local storage mstartfn func() curg *g // 当前运行的goroutine caughtsig guintptr p puintptr // 关联p和执行的go代码 nextp puintptr id int32 mallocing int32 // 状态
```

spinning bool // m是否out of work blocked bool // m是否被阻塞 inwb bool // m是否在执行写屏蔽 printlock int8 incgo bool // m在执行cgo吗 fastrand uint32 ncgocall uint64 // cgo调用的总数 ncgo int32 // 当前cgo调用的数目 park note alllink *m // 用于链接allm schedlink muintptr mcache *mcache // 当前m的内存缓存 lockedg *g // 锁定g在当前m上执行，而不会切换到其他m createstack [32]uintptr // thread创建的栈 }

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。也就是说线程的栈也是用的g实现，而不是使用的OS的。

P：代表一个处理器，每一个运行的M都必须绑定一个P，就像线程必须在么一个CPU核上执行一样，由P来调度G在M上的运行，P的个数就是GOMAXPROCS（最大256），启动时固定的，一般不修改；M的个数和P的个数不一定一样多（会有休眠的M或者不需要太多的M）（最大10000）；每一个P保存着本地G任务队列，也有一个全局G任务队列。P的数据结构：

type p struct { lock mutex id int32 status uint32 // 状态，可以为pidle/prunning/... link puintptr schedtick uint32 // 每调度一次加1 syscalltick uint32 // 每一次系统调用加1 sysmontick sysmontick m muintptr // 回链到关联的m mcache *mcache racectx uintptr goidcache uint64 // goroutine的ID的缓存 goidcacheend uint64 // 可运行的goroutine的队列 runqhead uint32 runqtail uint32 runq [256]guintptr runnext guintptr // 下一个运行的g sudogcache []*sudog sudogbuf [128]*sudog palloc persistentAlloc // per-P to avoid mutex pad [sys.CacheLineSize]byte

其中P的状态有Pidle, Prunning, Psyscall, Pgcstop, Pdead；在其内部队列runqhead里面有可运行的goroutine，P优先从内部获取执行的g，这样能够提高效率。

除此之外，还有一个数据结构需要在这里提及，就是schedt，可以看做是一个全局的调度者：

type schedt struct { goidgen uint64 lastpoll uint64 lock mutex midle muintptr // idle状态的m nmidle int32 // idle状态的m个数 nmidlelocked int32 // lockde状态的m个数 mcount int32 // 创建的m的总数 maxmcount int32 // m允许的最大个数 ngsys uint32 // 系统中goroutine的数目，会自动更新 pidle puintptr // idle的p npidle uint32 nmspinning uint32 // 全局的可运行的g队列 runqhead guintptr runqtail guintptr runqsize int32 // dead的G的全局缓存 gflock mutex gfreeStack *g

```
gfreeNoStack *g ngfree int32 // sudog的缓存中心 sudoglock mutex sudogcache *sudog }
```

大多数需要的信息都已放在了结构体M、G和P中，schedt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。schedt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。

goroutine的运行过程

所有的goroutine都是由函数`newproc`来创建的，但是由于该函数不能调用分段栈，最后真正调用的是`newproc1`。在`newproc1`中主要进行如下动作：

```
func newproc1(fn *funcval, argp *uint8, narg int32, nret int32, callerpc uintptr) *g { newg = malg(_StackMin) casgstatus(newg, _Gidle, _Gdead) allgadd(newg) newg.sched.sp = sp newg.stktopsp = sp newg.sched.pc = funcPC(goexit) + sys.PCQuantum newg.sched.g = guintptr(unsafe.Pointer(newg)) gostartcallfn(&newg.sched, fn) newg.gopc = callerpc newg.startpc = fn.fn ..... }
```

分配一个g的结构体

初始化这个结构体的一些域

将g挂在就绪队列

绑定g到一个m上

这个绑定只要m没有突破上限GOMAXPROCS,就拿一个m绑定一个g。如果m的waiting队列中有就从队列中拿,否则就要新建一个m,调用`newm`。

```
func newm(fn func(), _p_ *p) { mp := allocm(_p_, fn) mp.nextp.set(_p_) mp.sigmask = initSigmask execLock.rlock() newosproc(mp, unsafe.Pointer(mp.g0.stack.hi)) execLock.runlock() }
```

该函数其实就是创建一个m，跟`newproc`有些相似，之前也说了m在底层就是一个线程的创建，也即是`newosproc`函数，在往下挖可以看到会根据不同的OS来执行不同的`bsdthread_create`函数，而底层就是调用的`runtime.clone`：

```
clone(cloneFlags,stk,unsafe.Pointer(mp),unsafe.Pointer(mp.g0),unsafe.Pointer(funcPC(mstart)))
```

m创建好之后，线程的入口是mstart，最后调用的即是`mstart1`：

```
func mstart1() { _g_ := getg() gosave(&_g_.m.g0.sched) _g_.m.g0.sched.pc = ^uintptr(0) asminit() minit() if _g_.m == &m0 { initsig(false) } if fn := _g_.m.mstartfn; fn != nil { fn() } schedule() }
```

里面最重要的就是schedule了，在schedule中的动作大体就是找到一个等待运行的g，然后然后搬到m上，设置其状态为`Grunning`,直接切换到g的上下文环境,恢复g的执行。

```
func schedule() { _g_ := getg() if _g_.m.lockedg != nil { stoplockedm() execute(_g_.m.lockedg, false) // Never returns. } }
```

`schedule`的执行可以大体总结为：

schedule函数获取g => [必要时休眠] => [唤醒后继续获取] => execute函数执行g => 执行后返回到goexit => 重新执行schedule函数

简单来说g所经历的几个主要的过程就是：Gwaiting->Grunnable->Grunning。经历了创建,到挂在就绪队列,到从就绪队列拿出并运行整个过程。

```
casgstatus(gp, _Gwaiting, _Grunnable) casgstatus(gp, _Grunnable, _Grunning)
```

引入了struct M这层抽象。m就是这里的worker,但不是线程。处理系统调用中的m不会占用mcpu数量,只有干事的m才会对应到线程.当mcpu数量少于GOMAXPROCS时可以一直开新的线程干活.而goroutine的执行则是在m和g都满足之后通过schedule切换上下文进入的.

抢占式调度

当有很多goroutine需要执行的时候，是怎么调度的了，上面说的P还没有出场呢，在`runtime.main`中会创建一个额外m运行`sysmon`函数，抢占就是在sysmon中实现的。

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.

```
func sysmon() { lasttrace := int64(0) 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 } usleep(delay) ..... } }
```

里面的函数`retake`负责抢占：

```
func retake(now int64) uint32 { n := 0 for i := int32(0); i < gomaxprocs; i++ { _p_ := allp[i] if _p_ == nil { continue } pd := &_p_.sysmontick s := _p_.status if s == _Psyscall { // 如果p的syscall时间超过一个sysmon tick则抢占该p t := int64(_p_.syscalltick) if int64(pd.syscalltick) != t { pd.syscalltick = uint32(t) pd.syscallwhen = now continue } if runqempty(_p_) && atomic.Load(&sched.nmspinning)+atomic.Load(&sched.npidle) > 0 && pd.syscallwhen+10*1000*1000 > now { continue } incidlelocked(-1) if atomic.Cas(&_p_.status, s, _Pidle) { if trace.enabled { traceGoSysBlock(_p_) traceProcStop(_p_) } n++ _p_.syscalltick++ handoffp(_p_) } incidlelocked(1) } else if s == _Prunning { // 如果
```

```
G 运 行 时 间 过 长 ， 则 抢 占 该 G  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_) } } return uint32(n) }
```

枚举所有的P 如果P在系统调用中(_Psyscall), 且经过了一次sysmon循环(20us~10ms), 则抢占这个P， 调用handoffp解除M和P之间的关联， 如果P在运行中(_Prunning), 且经过了一次sysmon循环并且G运行时间超过forcePreemptNS(10ms), 则抢占这个P

并设置g.preempt = true，g.stackguard0 = stackPreempt。

为什么设置了stackguard就可以实现抢占?

因为这个值用于检查当前栈空间是否足够, go函数的开头会比对这个值判断是否需要扩张栈。

newstack函数判断g.stackguard0等于stackPreempt, 就知道这是抢占触发的, 这时会再检查一遍是否要抢占。

抢占机制保证了不会有一个G长时间的运行导致其他G无法运行的情况发生。

总结

相比大多数并行设计模型，Go比较优势的设计就是P上下文这个概念的出现，如果只有G和M的对应关系，那么当G阻塞在IO上的时候，M是没有实际在工作的，这样造成了资源的浪费，没有了P，那么所有G的列表都放在全局，这样导致临界区太大，对多核调度造成极大影响。

而goroutine在使用上面的特点，感觉既可以用来做密集的多核计算，又可以做高并发的IO应用，做IO应用的时候，写起来感觉和对程序员最友好的同步阻塞一样，而实际上由于runtime的调度，底层是以同步非阻塞的方式在运行（即IO多路复用）。

所以说保护现场的抢占式调度和G被阻塞后传递给其他m调用的核心思想，使得goroutine的产生。

本文从宏观角度介绍了一下Go调度器的调度过程。Go调度器也是Go语言最精华的部分，希望对大家有所帮助。