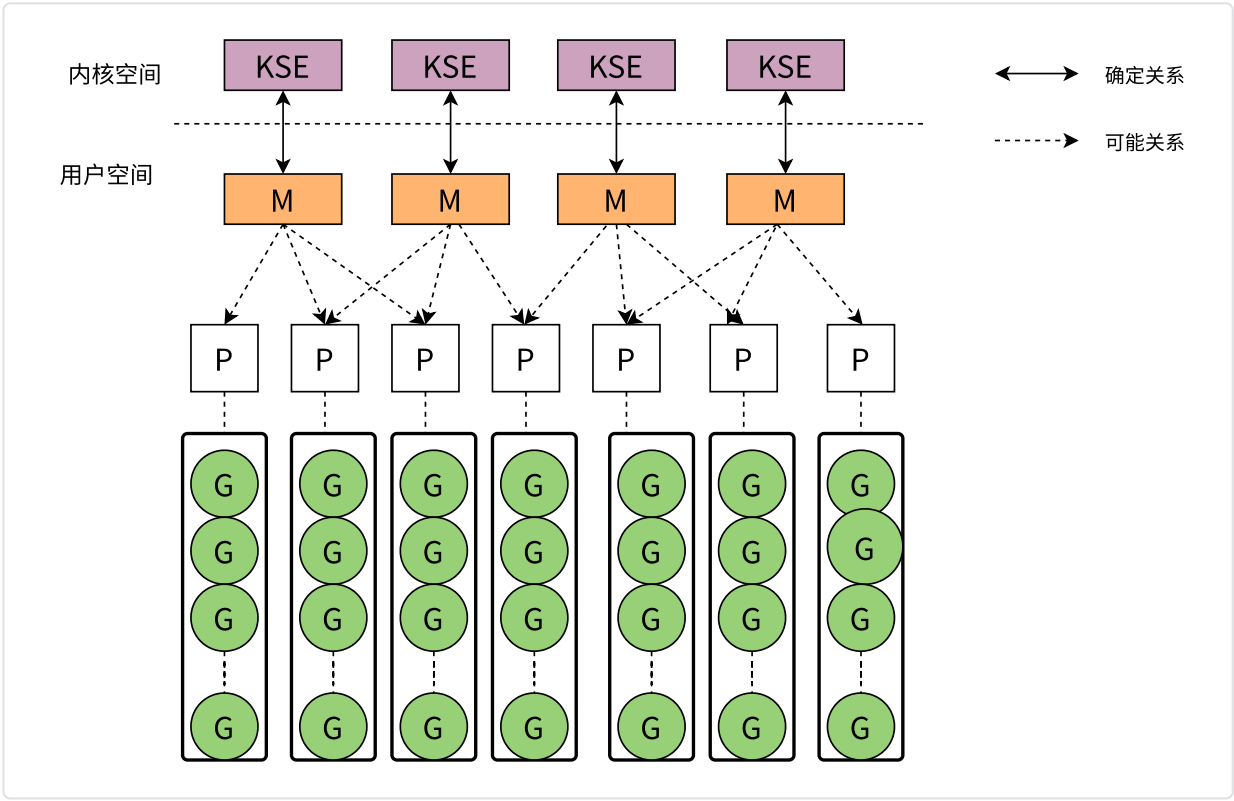
runtime-调度

1、GMP模型



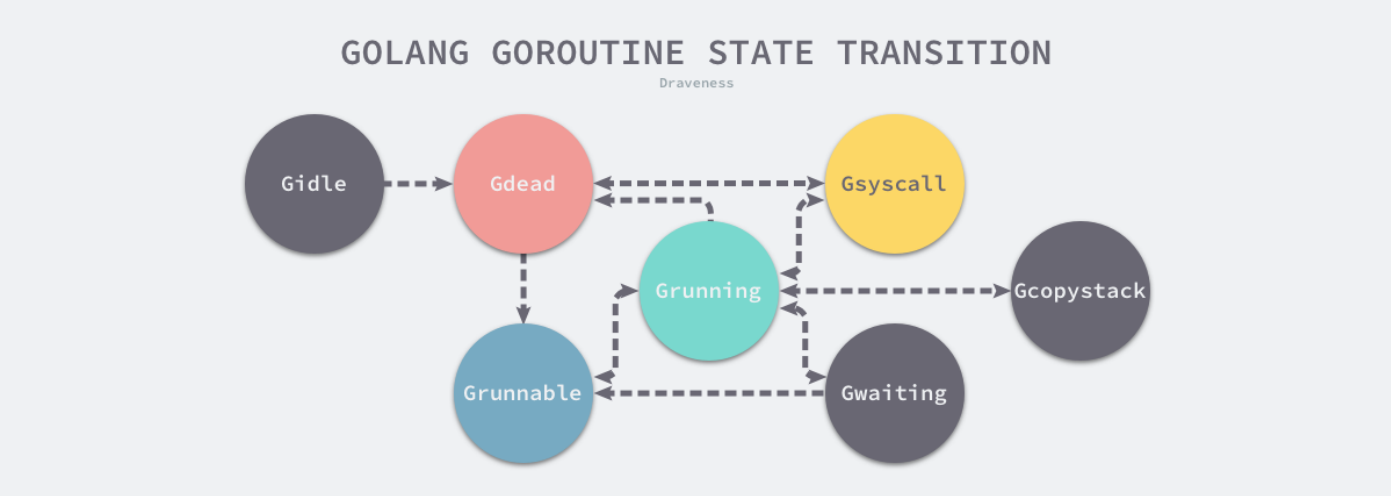
1.1 数据结构

- G — 表示 Goroutine，它是一个待执行的任务；
- M — 表示操作系统的线程，它由操作系统的调度器调度和管理；
- P — 表示处理器，它可以被看做运行在线程上的本地调度器；

- G
  - 执行单元，相比线程更轻量，上下文切换开销小
  - 存在运行时，用户态，由runtime.g表示，只有40多个字段
- M
  - 对应并发模型中操作系统的线程，调度器最多可以创建 10000 个线程
  - 大多数的线程都不会执行用户代码（可能陷入系统调用），最多只会有 GOMAXPROCS 个活跃线程能够正常运行
- P

- 处理器P是线程与goroutine的中间层
- 负责线程需要的上下文环境, 也负责调度线程上等待的队列
- 能在goroutine执行io操作时，及时让出cpu的，提供线程的利用率

1.2 goroutine的状态



	A	B
1	**状态**	**描述**
2	_Gidle	刚刚被分配并且还没有被初始化
3	_Grunnable	没有执行代码，没有栈的所有权，存储在运行队列中
4	_Grunning	可以执行代码，拥有栈的所有权，被赋予了内核线程 M 和处理器 P
5	_Gsyscall	正在执行系统调用，拥有栈的所有权，没有执行用户代码，被赋予了内核线程 M 但不在运行队列上
6	_Gwaiting	由于运行时而被阻塞，没有执行用户代码并且不在运行队列上，但是可能存在于 Channel 的等待队列上
7	_Gdead	没有被使用，没有执行代码，可能有分配的栈
8	_Gcopystack	栈正在被拷贝，没有执行代码，不在运行队列上
9	_Gpreempted	由于抢占而被阻塞，没有执行用户代码并且不在运行队列上，等待唤醒
10	_Gscan	GC 正在扫描栈空间，没有执行代码，可以与其他状态同时存在

1.3 p的状态

	A	B
1	**状态**	**描述**
2	_Pidle	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，这 队列为空
3	_Prunning	被线程 M 持有，并且正在执行用户代码或者调度器
4	_Psyscall	没有执行用户代码，当前线程陷入系统调用
5	_Pgcstop	被线程 M 持有，当前处理器由于垃圾回收被停止
6	_Pdead	当前处理器已经不被使用

1.4 深度剖析g0 & m0

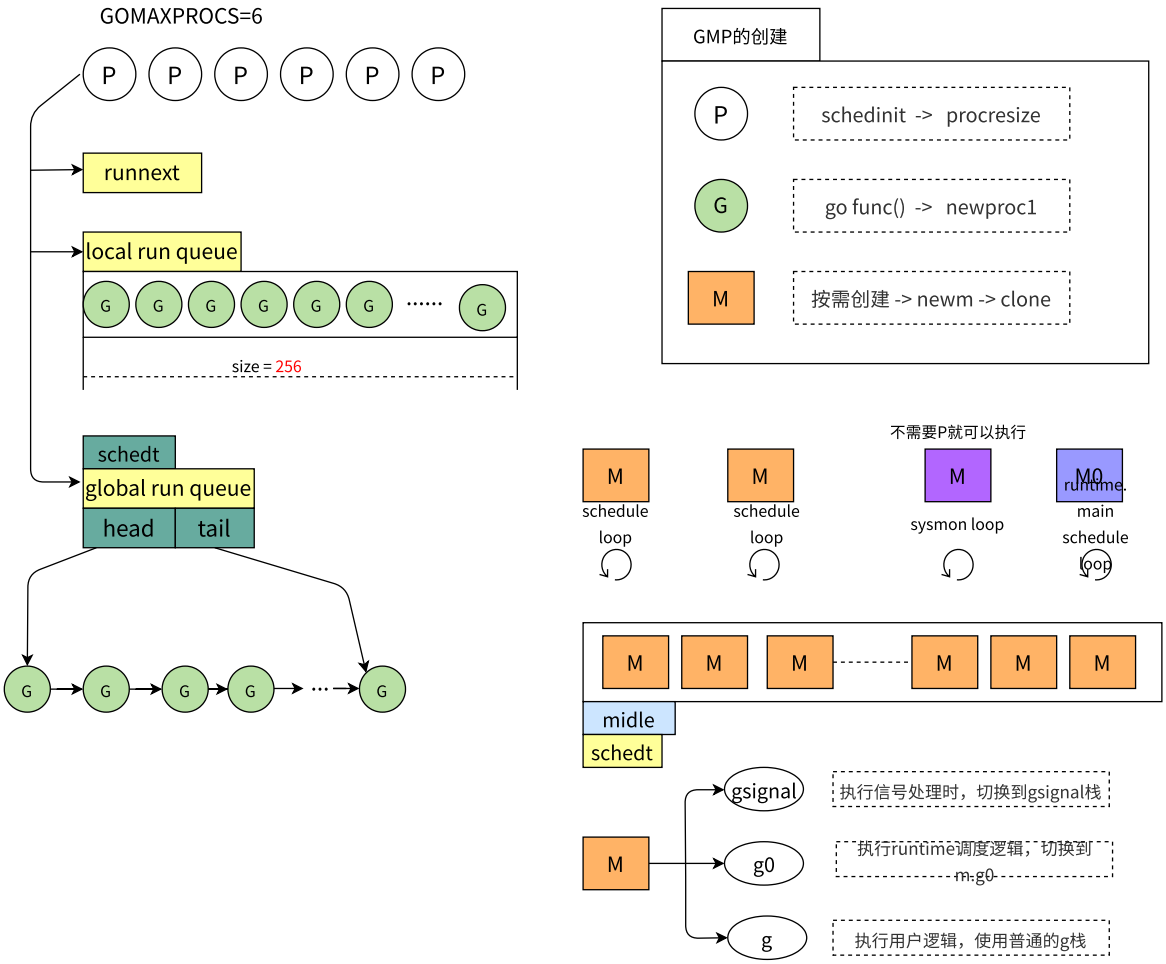
1.4.1 m0是什么？作用是什么？

- m0是runtime所创建的第一个系统线程，一个go进程只有一个m0, 也叫主线程
  - 数据结构： m0和其他创建的m没有任何区别
  - 创建过程： m0是进程在启动时由汇编直接复制给m0, 其他后续的m则都是由runtime自行创建的
  - 变量声明：m0和常规的m一样，m0的定义就是var m0 m, 与其他的没啥区别

1.4.2 g0是什么？作用是什么？

- g的分类
  - 执行用户任务的叫做g
  - 执行runtime.main的 main goroutine
  - 指定调度任务的叫g0
- g0比较特殊，每一个m都只有一个g0(有且只有一个)，每个m都只会绑定一个g0
  - 数据结构：g0与其他的g数据结构一样，但存在栈的差别，g0使用的是系统栈，默认8mb, 不能扩容，而普通的g使用的是协程栈，只有2kb，可扩容
  - 运行状态：g0与普通的g不一样，没有那么多状态，也不会发生抢占，只会进行调度
  - 创建过程：g0是通过汇编赋值，其他的都是通过runtime创建，且变量声明没有什么不同

2、调度组件



2.1 三级队列

2.1.1 runnext

Go

```go
type p struct {

    // 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.
    //
    // Note that while other P's may atomically CAS this to zero,
    // only the owner P can CAS it to a valid G.
    runnext guintptr

}
```

- 优先级最高，主要为了提升cpu的亲和度

2.1.2 local queue

Go

```go
type p struct {
    // Queue of runnable goroutines. Accessed without lock.
    runqhead uint32
    runqtail uint32
    runq     [256]guintptr
}
```

- 本地队列，使用环形数组，大小256，访问了时候不需要加锁

2.1.3 global queue

Go

```go
type schedt struct {
    lock mutex

    // Global runnable queue.
    runq     gQueue
    runqsize int32
}

// A gQueue is a dequeue of Gs linked through g.schedlink. A G can only
// be on one gQueue or gList at a time.
type gQueue struct {
   head guintptr
   tail guintptr
}
```

- 使用一个链表实现，访问的时候需要加锁

2.2 调度器的初始化

2.2.1 mstart0

Go

```go
// mstart0 is the Go entry-point for new Ms.
// This must not split the stack because we may not even have stack
// bounds set up yet.
//
// May run during STW (because it doesn't have a P yet), so write
// barriers are not allowed.
//
//go:nosplit
//go:nowritebarrierrec
func mstart0() {
    _g_ := getg()
    osStack := _g_.stack.lo == 0
    if osStack {
        size := _g_.stack.hi
        if size == 0 {
            size = 8192 * sys.StackGuardMultiplier
        }
        _g_.stack.hi = uintptr(noescape(unsafe.Pointer(&size)))
        _g_.stack.lo = _g_.stack.hi - size + 1024
    }
    _g_.stackguard0 = _g_.stack.lo + _StackGuard
    _g_.stackguard1 = _g_.stackguard0
    // ...
}
```

- g0的初始化，使用的系统栈，主要用于调度

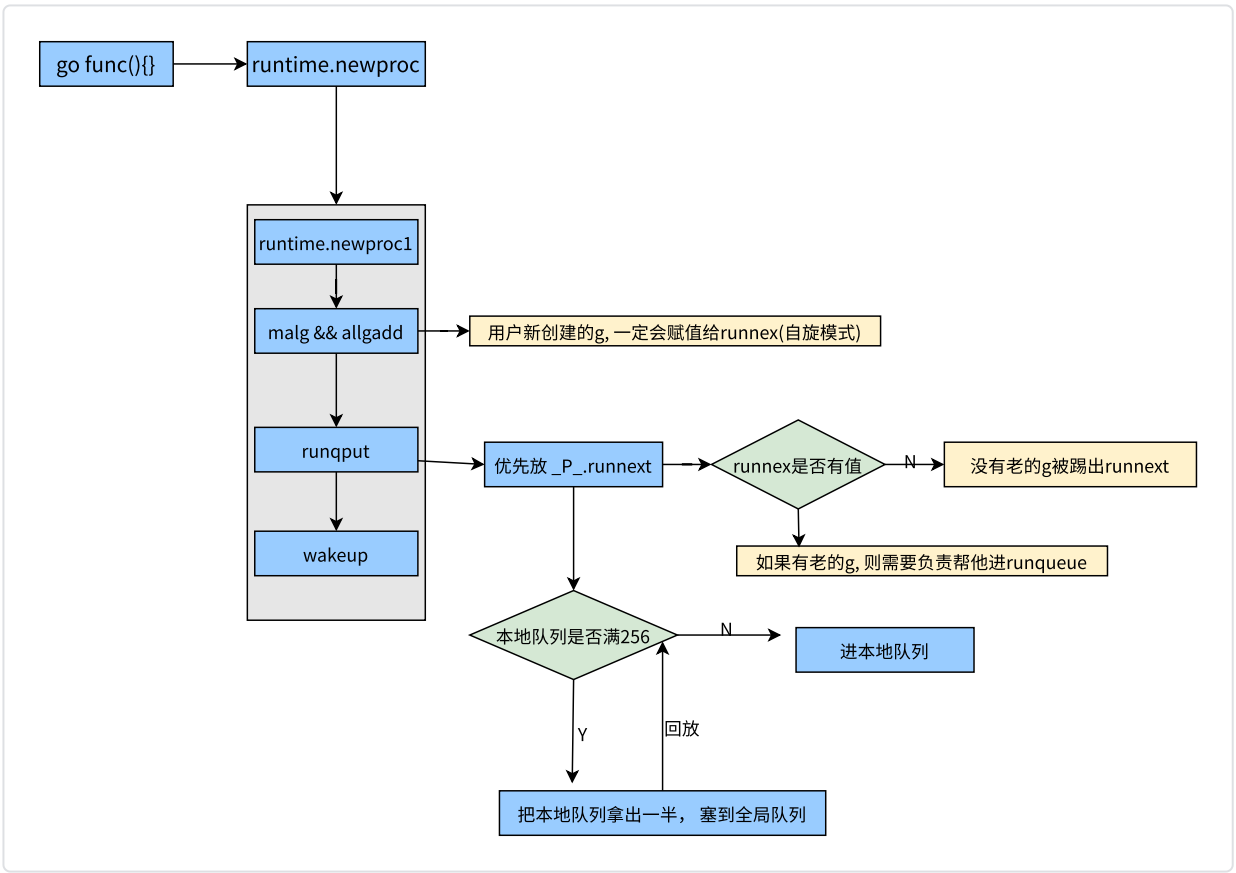
2.2.2 mstart1的执行，会指定schedule

Go

```go
// One round of scheduler: find a runnable goroutine and execute it.
// Never returns.
func schedule() {
//...
top:
    //...
   if gp == nil && gcBlackenEnabled != 0 {
      gp = gcController.findRunnableGCWorker(_g_.m.p.ptr())
      if gp != nil {
         tryWakeP = true
      }
   }
   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.
      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 {
      gp, inheritTime = runqget(_g_.m.p.ptr())
      // We can see gp != nil here even if the M is spinning,
      // if checkTimers added a local goroutine via goready.
   }
   if gp == nil {
      gp, inheritTime = findrunnable() // blocks until work is available
   }

   execute(gp, inheritTime)
}
```

- schedule作用是找到一个g, 然后执行
  - 为了保证公平, 通过 schedtick 在每执行61次之后就会从全局队列拿可以执行的g
  - runqget是从本地队列中拿g
  - 如果前两种方法都没有找到 g，会通过 `runtime.findrunnable` 进行阻塞地查找 g
- findrunnable获取g的过程
  - 从本地运行队列、全局运行队列中查找
  - 从网络轮询器中查找是否有 g 等待运行
  - 从其他的p中窃取可运行的g，通过 `runtime.runqsteal` , 也叫工作窃取

3、 goroutine生产者



3.1 普通任务g的创建

Go

```go
// Create a new g in state _Grunnable, starting at fn, with narg bytes
// of arguments starting at argp. callerpc is the address of the go
// statement that created this. The caller is responsible for adding
// the new g to the scheduler.
//
// This must run on the system stack because it's the continuation of
// newproc, which cannot split the stack.
//
//go:systemstack
func newproc1(fn *funcval, argp unsafe.Pointer, narg int32, callergp *g,
callerpc uintptr) *g {
    // 1. 得到一个_Gdead状态的g，如果没有就会创建一个g
    newg := gfget(_p_)
    if newg == nil {
        newg = malg(_StackMin)
        casgstatus(newg, _Gidle, _Gdead)
        allgadd(newg) // publishes with a g->status of Gdead so GC scanner
doesn't look at uninitialized stack.
    }

    // 2. 将参数拷贝到协程栈上
    if narg > 0 {
        memmove(unsafe.Pointer(spArg), argp, uintptr(narg))
    }

    // 3. 更新g相关相关属性
    newg.sched.sp = sp
    newg.stktopsp = sp
    newg.sched.pc = abi.FuncPCABI0(goexit) + sys.PCQuantum // +PCQuantum so that
previous instruction is in same function
    newg.sched.g = guintptr(unsafe.Pointer(newg))
    gostartcallfn(&newg.sched, fn)
    newg.gopc = callerpc
    newg.ancestors = saveAncestors(callergp)
    newg.startpc = fn.fn
    casgstatus(newg, _Gdead, _Grunnable)
    newg.goid = int64(_p_.goidcache)
}

```

3.1.1 newproc1函数具体作用

- 获取或者创建新的 g 结构体
- 调用runtime.memmove将fn函数的所有参数拷贝到栈上

        - argp 和 narg 分别是参数的内存空间和大小
    - 更新 g 调度相关的属性

3.1.2 在调度循环中goexit是什么时候加入的?

- 在执行abi.FuncPCABI0(goexit)

3.2 将g放到队列中

Go

```go
// runqput tries to put g on the local runnable queue.
// If next is false, runqput adds g to the tail of the runnable queue.
// If next is true, runqput puts g in the _p_.runnext slot.
// If the run queue is full, runnext puts g on the global queue.
// Executed only by the owner P.
func runqput(_p_ *p, gp *g, next bool) {
    if randomizeScheduler && next && fastrand()%2 == 0 {
        next = false
    }
    //1. 先将g放在runnext中
    if next {
    retryNext:
        oldnext := _p_.runnext
        if !_p_.runnext.cas(oldnext, guintptr(unsafe.Pointer(gp))) {
            goto retryNext
        }
        if oldnext == 0 {
            return
        }
        // Kick the old runnext out to the regular run queue.
        gp = oldnext.ptr()
    }

retry:
    //2. 将g放在本地队列中
    h := atomic.LoadAcq(&_p_.runqhead) // load-acquire, synchronize with consumers
    t := _p_.runqtail
    if t-h < uint32(len(_p_.runq)) {
        _p_.runq[t%uint32(len(_p_.runq))].set(gp)
        atomic.StoreRel(&_p_.runqtail, t+1) // store-release, makes the item available for consumption
        return
    }

    // 3. 当本地的队列没有空间，则将本地队列中一半的g放在全局队列中
    if runqputslow(_p_, gp, h, t) {
        return
    }
    // the queue is not full, now the put above must succeed
    goto retry
}
```

3.2.1 runqput的作用

- 当 next 为 true 时，将 g 放在runnext上
- 当 next 为 false 并且本地运行队列还有剩余空间时，将 g 本地运行队列
- 当本地运行队列已经没有剩余空间时, 就会把本地队列中一半的g 和待加入的 g 通过 `runtime.runqputslow` 添加到全局运行队列上
  - 全局队列的操作需要加锁完成

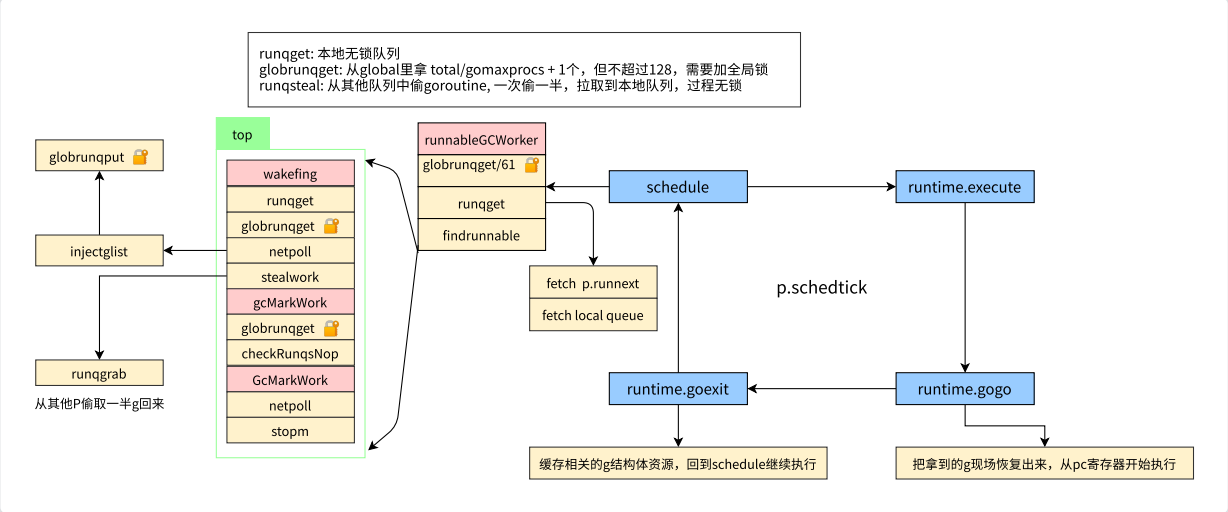
3.3 唤醒空闲的p

Go

```go
// Tries to add one more P to execute G's.
// Called when a G is made runnable (newproc, ready).
func wakep() {
    if atomic.Load(&sched.npidle) == 0 {
        return
    }
    // be conservative about spinning threads
    if atomic.Load(&sched.nmspinning) != 0 || !atomic.Cas(&sched.nmspinning, 0, 1) {
        return
    }
    startm(nil, true)
}
```

- wakep的作用
  - 可能会调用 runtime.wakep 唤醒闲置的处理器
  - runtime.startm 启动新的线程执行该 Goroutine

4、goroutine消费者



4.1 runtime.schedule

- 在前面的调度组件中有讲到，主要是从各种队列中拿出可执行的g

4.2 runtime.execute

Go

```go
// Schedules gp to run on the current M.
// If inheritTime is true, gp inherits the remaining time in the
// current time slice. Otherwise, it starts a new time slice.
// Never returns.
//
// Write barriers are allowed because this is called immediately after
// acquiring a P in several places.
//
//go:yeswritebarrierrec
func execute(gp *g, inheritTime bool) {
    _g_ := getg()

    // Assign gp.m before entering _Grunning so running Gs have an
    // M.
    _g_.m.curg = gp
    gp.m = _g_.m
    casgstatus(gp, _Grunnable, _Grunning)
    gp.waitsince = 0
    gp.preempt = false
    gp.stackguard0 = gp.stack.lo + _StackGuard
    if !inheritTime {
        _g_.m.p.ptr().schedtick++
    }

    // Check whether the profiler needs to be turned on or off.
    hz := sched.profilehz
    if _g_.m.profilehz != hz {
        setThreadCPUProfiler(hz)
    }

    if trace.enabled {
        // GoSysExit has to happen when we have a P, but before GoStart.
        // So we emit it here.
        if gp.syscallsp != 0 && gp.sysblocktraced {
            traceGoSysExit(gp.sysexitticks)
        }
        traceGoStart()
    }

    gogo(&gp.sched)
}
```

- 主要是做一些准备工作，然后再去调用gogo汇编函数

4.3 gogo函数

Assembly language

```
// func gogo(buf *gobuf)
// restore state from Gobuf; longjmp
TEXT runtime·gogo(SB), NOSPLIT, $0-8
    MOVQ    buf+0(FP), BX      // gobuf
    MOVQ    gobuf_g(BX), DX
    MOVQ    0(DX), CX      // make sure g != nil
    JMP     gogo<>(SB)

TEXT gogo<>(SB), NOSPLIT, $0
    get_tls(CX)
    MOVQ    DX, g(CX)
    MOVQ    DX, R14        // set the g register
    MOVQ    gobuf_sp(BX), SP   // restore SP
    MOVQ    gobuf_ret(BX), AX
    MOVQ    gobuf_ctxt(BX), DX
    MOVQ    gobuf_bp(BX), BP
    MOVQ    $0, gobuf_sp(BX)   // 将 runtime.goexit 函数的 PC 恢复到 SP 中 clear to help garbage collector
    MOVQ    $0, gobuf_ret(BX)
    MOVQ    $0, gobuf_ctxt(BX)
    MOVQ    $0, gobuf_bp(BX)
    MOVQ    gobuf_pc(BX), BX  // 获取待执行函数的程序计数器
    JMP     BX    // 开始执行，也就是调用 runtime·goexit(SB)
```

4.4 goexit函数

Assembly language

```
// The top-most function running on a goroutine
// returns to goexit+PCQuantum.
TEXT runtime·goexit(SB),NOSPLIT|TOPFRAME,$0-0
    BYTE    $0x90  // NOP
    CALL    runtime·goexit1(SB)    // does not return
    // traceback from goexit1 must hit code range of goexit
    BYTE    $0x90  // NOP

```

Go

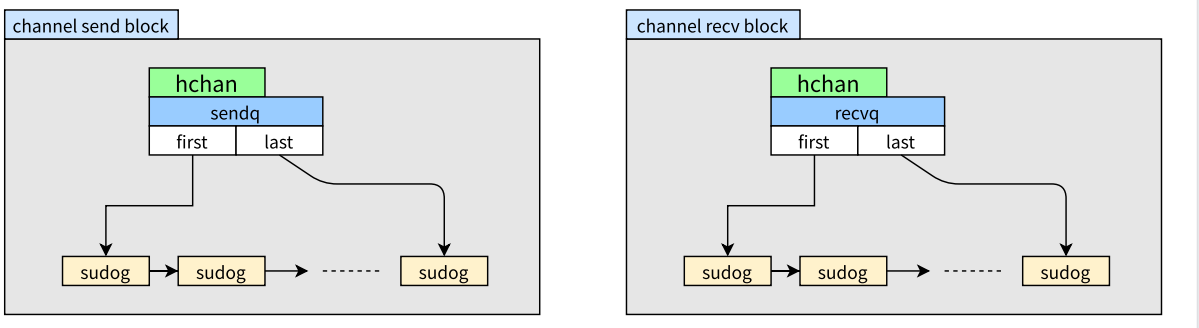
```go
// Finishes execution of the current goroutine.
func goexit1() {
    if raceenabled {
        racegoend()
    }
    if trace.enabled {
        traceGoEnd()
    }
    mcall(goexit0)
}

// goexit continuation on g0.
func goexit0(gp *g) {
    //...
    schedule()
}
```

- runtime.goexit0函数的作用
  - 该函数会将 g 转换会 _Gdead 状态、
  - 清理其中的字段、
  - 移除 g 和线程的关联
  - 并调用 `runtime.gfput` 重新加入处理器的 g 空闲列表 `gFree`：

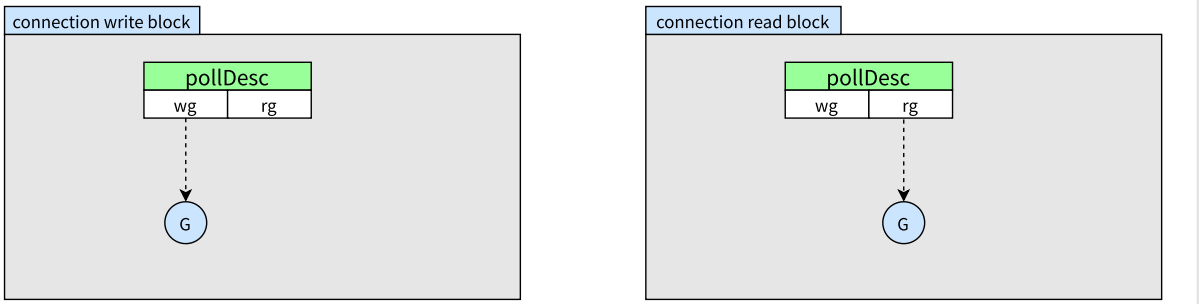
5、处理阻塞

5.1 runtime可以处理的阻塞

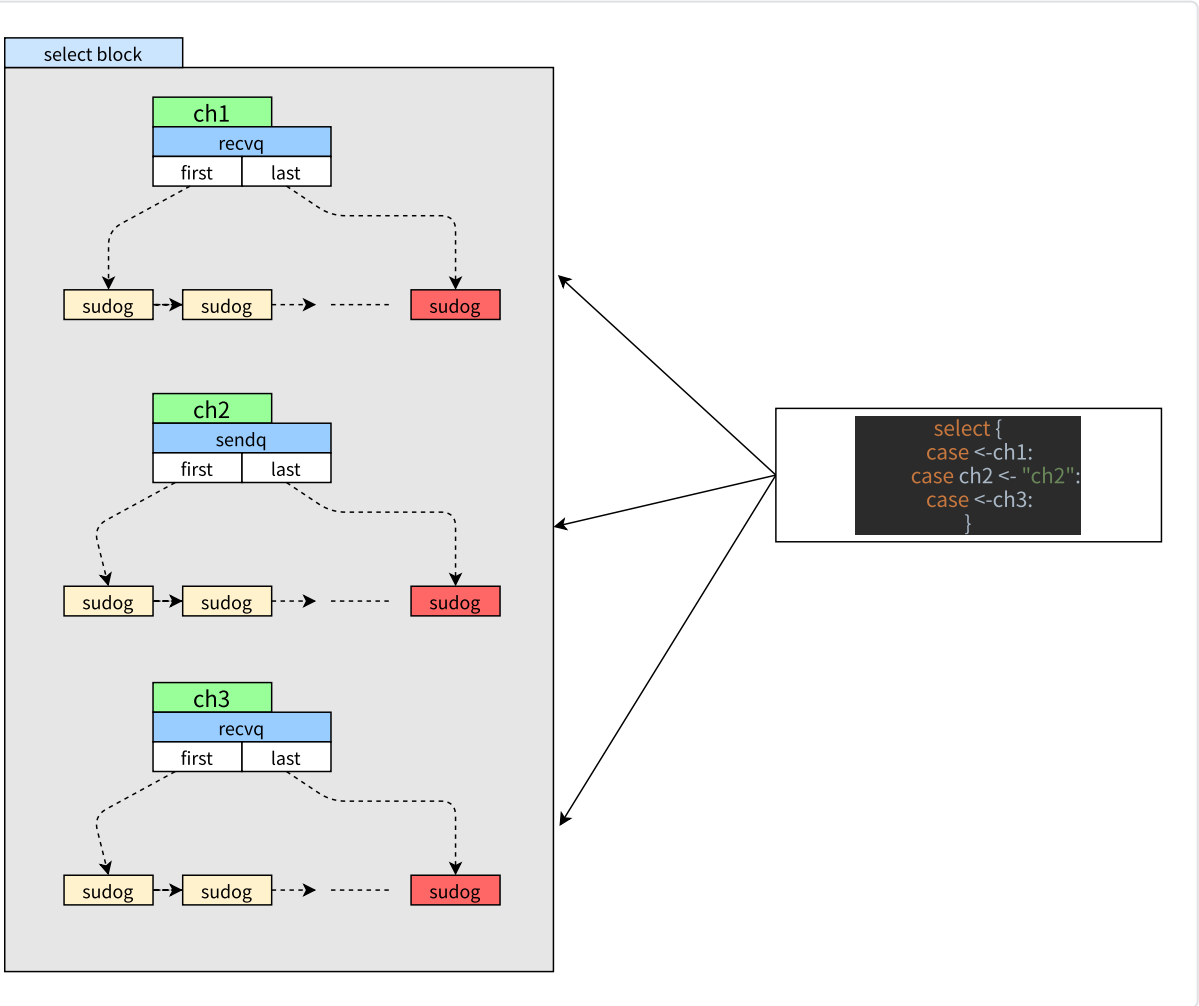
5.1.1 channel发送与接收阻塞 Send-recv block



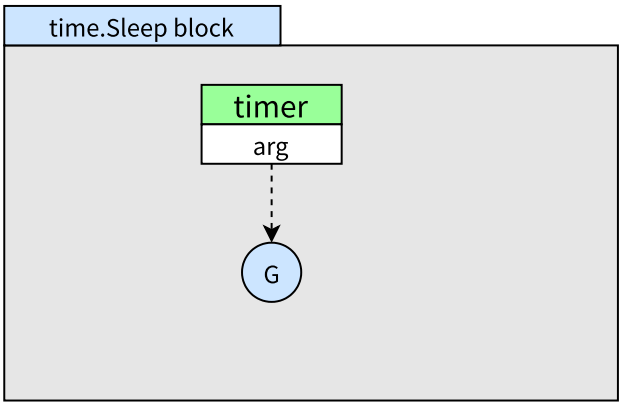
5.1.2 网络IO读写阻塞 Net read-write block



5.1.3 Select block



5.1.4 Sleep block



5.1.5 lock

5.2 runtime如何处理阻塞

5.2.1 主动挂起 runtime.gopark （挂起）

Go

```go
// Puts the current goroutine into a waiting state and calls unlockf on the
// system stack.
//
// If unlockf returns false, the goroutine is resumed.
//
// unlockf must not access this G's stack, as it may be moved between
// the call to gopark and the call to unlockf.
//
// Note that because unlockf is called after putting the G into a waiting
// state, the G may have already been readied by the time unlockf is called
// unless there is external synchronization preventing the G from being
// readied. If unlockf returns false, it must guarantee that the G cannot be
// externally readied.
//
// Reason explains why the goroutine has been parked. It is displayed in stack
// traces and heap dumps. Reasons should be unique and descriptive. Do not
// re-use reasons, add new ones.
func gopark(unlockf func(*g, unsafe.Pointer) bool, lock unsafe.Pointer, reason waitReason, traceEv byte, traceskip int) {
    if reason != waitReasonSleep {
        checkTimeouts() // timeouts may expire while two goroutines keep the scheduler busy
    }
    mp := acquirem()
    gp := mp.curg
    status := readgstatus(gp)
    if status != _Grunning && status != _Gscanrunning {
        throw("gopark: bad g status")
    }
    mp.waitlock = lock
    mp.waitunlockf = unlockf
    gp.waitreason = reason
    mp.waittraceev = traceEv
    mp.waittraceskip = traceskip
    releasem(mp)
    // can't do anything that might move the G between Ms here.
    mcall(park_m)
}
```

- 挂起逻辑
  - gopark会将g变为等待状态，但不会放到队列中
  - 会调用park_m, 切换到 g0 的栈上， 将g 的状态从 `_Grunning` 切换至 `_Gwaiting`
  - 调用dropg, 将当前移除线程和 g之间的关联，然后再调用schedule触发调度

5.2.2 主动恢复 runtime.goready （唤醒）

Go

```go
func goready(gp *g, traceskip int) {
    systemstack(func() {
        ready(gp, traceskip, true)
    })
}

// Mark gp ready to run.
func ready(gp *g, traceskip int, next bool) {
    if trace.enabled {
        traceGoUnpark(gp, traceskip)
    }

    status := readgstatus(gp)

    // Mark runnable.
    _g_ := getg()
    mp := acquirem() // 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)
    wakep()
    releasem(mp)
}
```

- 注意
  - 程序的调度gopark与goready必须是成对出现，不然会发生死锁
- goready的作用
  - 将准备就绪的 g 的状态切换至 `_Grunnable`
  - 并将其加入处理器的运行队列中，等待调度器的调度。

5.3 不可处理的阻塞

5.3.1 系统调用

5.3.2 cgo

5.4 系统监控 sysmon

Go

```
/ Always runs without a P, so write barriers are not allowed.
//
//go:nowritebarrierrec
func sysmon() {
    lock(&sched.lock)
    sched.nmsys++
    checkdead()
    unlock(&sched.lock)

    // For syscall_runtime_doAllThreadsSyscall, sysmon is
    // sufficiently up to participate in fixups.
    atomic.Store(&sched.sysmonStarting, 0)

    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)
        mDoFixup()

        // sysmon should not enter deep sleep if schedtrace is enabled so that
        // it can print that information at the right time.
        //
        // It should also not enter deep sleep if there are any active P's so
        // that it can retake P's from syscalls, preempt long running G's, and
        // poll the network if all P's are busy for long stretches.
        //
        // It should wakeup from deep sleep if any P's become active either due
        // to exiting a syscall or waking up due to a timer expiring so that it
        // can resume performing those duties. If it wakes from a syscall it
        // resets idle and delay as a bet that since it had retaken a P from a
        // syscall before, it may need to do it again shortly after the
        // application starts work again. It does not reset idle when waking
```

```
43          // from a timer to avoid adding system load to applications that spend
44          // most of their time sleeping.
45          now := nanotime()
46          if debug.schedtrace <= 0 && (sched.gcwaiting != 0 ||
    atomic.Load(&sched.npidle) == uint32(gomaxprocs)) {
47             lock(&sched.lock)
48             if atomic.Load(&sched.gcwaiting) != 0 || atomic.Load(&sched.npidle) ==
    uint32(gomaxprocs) {
49                syscallWake := false
50                next, _ := timeSleepUntil()
51                if next > now {
52                   atomic.Store(&sched.sysmonwait, 1)
53                   unlock(&sched.lock)
54                   // Make wake-up period small enough
55                   // for the sampling to be correct.
56                   sleep := forcegcperiod / 2
57                   if next-now < sleep {
58                      sleep = next - now
59                   }
60                   shouldRelax := sleep >= osRelaxMinNS
61                   if shouldRelax {
62                      osRelax(true)
63                   }
64                   syscallWake = notetsleep(&sched.sysmonnote, sleep)
65                   mDoFixup()
66                   if shouldRelax {
67                      osRelax(false)
68                   }
69                   lock(&sched.lock)
70                   atomic.Store(&sched.sysmonwait, 0)
71                   noteclear(&sched.sysmonnote)
72                }
73                if syscallWake {
74                   idle = 0
75                   delay = 20
76                }
77             }
78             unlock(&sched.lock)
79          }
80
81          lock(&sched.sysmonlock)
82          // Update now in case we blocked on sysmonnote or spent a long time
83          // blocked on schedlock or sysmonlock above.
84          now = nanotime()
85
86          // trigger libc interceptors if needed
87          if *cgo_yield != nil {
88             asmcgocall(*cgo_yield, nil)
```

```
89          }
90          // poll network if not polled for more than 10ms
91          lastpoll := int64(atomic.Load64(&sched.lastpoll))
92          if netpollinited() && lastpoll != 0 && lastpoll+10*1000*1000 < now {
93             atomic.Cas64(&sched.lastpoll, uint64(lastpoll), uint64(now))
94             list := netpoll(0) // non-blocking - returns list of goroutines
95             if !list.empty() {
96                // Need to decrement number of idle locked M's
97                // (pretending that one more is running) before injectglist.
98                // Otherwise it can lead to the following situation:
99                // injectglist grabs all P's but before it starts M's to run the
    P's,
100               // another M returns from syscall, finishes running its G,
101               // observes that there is no work to do and no other running M's
102               // and reports deadlock.
103               incidlelocked(-1)
104               injectglist(&list)
105               incidlelocked(1)
106            }
107         }
108         mDoFixup()
109         if GOOS == "netbsd" {
110            // netpoll is responsible for waiting for timer
111            // expiration, so we typically don't have to worry
112            // about starting an M to service timers. (Note that
113            // sleep for timeSleepUntil above simply ensures sysmon
114            // starts running again when that timer expiration may
115            // cause Go code to run again).
116            //
117            // However, netbsd has a kernel bug that sometimes
118            // misses netpollBreak wake-ups, which can lead to
119            // unbounded delays servicing timers. If we detect this
120            // overrun, then startm to get something to handle the
121            // timer.
122            //
123            // See issue 42515 and
124            // https://gnats.netbsd.org/cgi-bin/query-pr-single.pl?number=50094.
125            if next, _ := timeSleepUntil(); next < now {
126               startm(nil, false)
127            }
128         }
129         if atomic.Load(&scavenge.sysmonWake) != 0 {
130            // Kick the scavenger awake if someone requested it.
131            wakeScavenger()
132         }
133         // retake P's blocked in syscalls
134         // and preempt long running G's
135         if retake(now) != 0 {
136            idle = 0
```

```
137         } else {
138             idle++
139         }
140         // check if we need to force a GC
141         if t := (gcTrigger{kind: gcTriggerTime, now: now}); t.test() &&
    atomic.Load(&forcegc.idle) != 0 {
142             lock(&forcegc.lock)
143             forcegc.idle = 0
144             var list gList
145             list.push(forcegc.g)
146             injectglist(&list)
147             unlock(&forcegc.lock)
148         }
149         if debug.schedtrace > 0 && lasttrace+int64(debug.schedtrace)*1000000 <=
    now {
150             lasttrace = now
151             schedtrace(debug.scheddetail > 0)
152         }
153         unlock(&sched.sysmonlock)
154     }
155 }
```

- 特点
  - 高优先级，不需要绑定p就可以执行
- sysmon休眠机制
  - 初始的休眠时间是 20μs；
  - 最长的休眠时间是 10ms；
  - 当系统监控在 50 个循环中都没有唤醒 Goroutine 时，休眠时间在每个循环都会倍增；
- 作用
  - Checkdead
    - 检查是否存在死锁，然后进入核心的监控循环
  - netpoll
    - 非阻塞地调用 `runtime.netpoll` 检查待执行的文件描述符并通过 `runtime.injectglist` 将所有处于就绪状态的 Goroutine 加入全局运行队列中
  - Retake
    - 如果是 syscall 卡了很久，那就把 p 剥离(handoffp)
    - 如果g执行了很长时间，就会发送信号抢占

6、参考文档

- Go scheduler变更历史
- go1.17 runtime源码

- Go 语言调度器与 Goroutine 实现原理