Notes on Go Runtime

计算机科学与技术学院　PB13011038
阴钰
yxonic@gmail.com

目　录

1　Go 的特色

Go，又称 golang，是 Google 开发的一种**静态强类型**，**编译型**，**并发型**，并具有**垃圾回收**功能的**系统级**编程语言 [2]。

Go 区别于其他语言的主要特色，集中体现在 Go 的运行时系统上，尤其是运行时的高效的调度、栈空间管理、同步通信。本文主要就这些内容，对 Go 的源代码进行分块解析。

2　准备工作

采用的 Go 源码为 1.5.3 版。对于本文讨论的内容，具观察，从 1.4 到 1.5，实现思路都变化不大，大部分算法、甚至函数名，都几乎没有变化；1.5 版本

后运行时也全部采用 Go 和汇编，去掉了 C 代码，使得代码更为一致，因此本文主要对 1.5 的代码进行分析。

要了解运行时各函数的作用、何时被调用，我们还需要对编译好的 Go 程序的结构有所了解。使用如下命令将 Go 程序编译到 Plan9 的汇编代码（这是一种通用汇编码，可以被 Go 的汇编器转为常见架构的机器码）：

```
$ go tool 6g -S hello.go
```

得到的汇编码，以及标准库中的汇编码，都遵循相同的格式。这种汇编代码中，有几个符号需要注意：SB (static base) 用来表示静态数据区的基址，FP (frame pointer) 用来表示函数参数基址，SP (stack pointer) 表示局部数据的基址。用 foo(SB) 表示一个叫 foo 的全局名字，foo<>(SB) 表示一个 static 的全局名字，0(FP), 8(FP) 分别表示第一个、第二个参数（64 位系统上），x-8(SP) 表示位于-8 位置的叫 x 的局部名字。

函数、全局变量分别为下面的结构：

```
TEXT runtime·profileloop(SB),NOSPLIT,$8
        MOVQ    $runtime·profileloop1(SB), CX
        MOVQ    CX, 0(SP)
        CALL    runtime·externalthreadhandler(SB)
        RET


DATA symbol+offset(SB)/width, value
```

另外，为了更方便地了解程序流程，我也采用了 GDB 调试的方法。使用：

```
$ go build hello.go
```

得到的程序即带有调试信息，可以直接进行 GDB。

Go 的 runtime 代码全部集中于 src/runtime 下。下文提到的文件如不加说明都是这个目录下的文件。

本文将只关心 linux+amd64 环境。其他环境道理是相同的。

3 Go 程序的启动流程

程序员编写的 Go 程序的入口为 `main.main`，但是在执行这个函数前，Go 运行时有一些工作要做。在 gdb 中，把断点设为入口点地址（通过 info files

获取)，可以知道，程序开始于 rt0_linux_amd64.s 中的 _rt0_amd64_linux 函数:

```
TEXT _rt0_amd64_linux(SB),NOSPLIT,$-8
    LEAQ    8(SP), SI // argv
    MOVQ    0(SP), DI // argc
    MOVQ    $main(SB), AX
    JMPAX             // call main in this file

TEXT main(SB),NOSPLIT,$-8
    MOVQ    $runtime·rt0_go(SB), AX
    JMPAX             // call runtime.rt0_go
```

之后调用了 runtime.rt0_go，它在 asm_amd64.s 中。针对 CPU 做一些检测后，分别进行了如下工作：为每个 CPU 初始化两个寄存器变量 g0 和 m0，它们的作用之后在调度器部分会提到，现在只需要知道 g0 和 m0 为寄存器变量，不同处理器上这些变量并不相同即可；然后处理参数、初始化系统、初始化调度器，都是对一些全局参数赋值、为一些全局数据结构初始化等操作；然后以 runtime.mainPC 为参数，调用 runtime.newproc，这个函数的机制也在调度器部分再考虑，此处只要知道它将 runtime.mainPC 函数加入调度队列；然后紧接着的 runtime.mstart 开启调度器，runtime.mainPC 会被调度执行，它将负责调用 runtime.main。

```
TEXT runtime·rt0_go(SB),NOSPLIT,$0
    // ...

    // 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

    // save m->g0 = g0
    MOVQ    CX, m_g0(AX)
```

```
// save m0 to g0->m
MOVQ    AX, g_m(CX)

CLD     // convention is D is always left cleared
CALL    runtime·check(SB)

MOVL    16(SP), AX  // copy argc
MOVL    AX, 0(SP)
MOVQ    24(SP), AX  // copy argv
MOVQ    AX, 8(SP)
CALL    runtime·args(SB)
CALL    runtime·osinit(SB)
CALL    runtime·schedinit(SB)

// 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
CALL    runtime·mstart(SB)

MOVL    $0xf1, 0xf1  // crash
RET
```

4 Go 的调度器

编译一个含有 goroutine 的 Go 程序，可以看到 go 语句被简单地转换为一个 `newproc` 函数。这和之前初始化第一个 goroutine 时是同一个函数。它负责将一个函数加入调度队列。在了解它的详细行为之前，我们先整体了解一下

Go 的调度器。

Go 调度器主要在 proc.go 中实现，有关机器的细节分散在 `asm_amd64.s`, `sys_linux_amd64.s` 之中。

Go 调度器最重要的三个数据结构分别为 G、M 和 P。

G 对应于 goroutine，是抽象的任务。M 对应系统线程（在 linux 下即进程），它负责执行 G。G 只保存任务本身的信息，执行时必须绑定到 M 上。而底层的处理器对应 P，M 执行时绑定于 P 之上。这个设计的细节后面还会提到。

调度器的调度算法基于任务窃取。于是，调度器本身的工作比较容易，即找一个等待中的任务，执行它。整个工作由 proc.go 中的 `schedule` 实现。这个函数选择一个任务 g（通过 `runtime.findrunnable`，这个函数如果在当前队列找不到任务会从别的队列窃取，有关调度队列的细节在后面有讲述），然后调用 `execute`，其中调用 `runtime.gogo` 实际执行，这个汇编函数会直接修改栈为 g 的栈（在这之前将返回地址设为 `goexit`，从而在 G 结束时做一些清理工作），从而（继续）执行 g。于是，我们开始时需要建立一个定时调用 `schedule` 的守护进程。当然也可以主动调用 `schedule` 释放控制。

这一切都开始于 `runtime.mstart` 函数。这个函数在做了一些基本的初始化之后，调用了 `mstart1`:

```
func mstart1() {
    _g_ := getg()

    if _g_ != _g_.m.g0 {
        throw("bad runtime·mstart")
    }

    // Record top of stack for use by mcall.
    gosave(&_g_.m.g0.sched)
    _g_.m.g0.sched.pc = ^uintptr(0)
    asminit()
    minit()

    // Install signal handlers.
```

```
    if _g_.m == &m0 {
        if iscgo && !cgoHasExtraM {
            cgoHasExtraM = true
            newextram()
        }
        initsig()
    }

    // Call startup callbacks.
    if fn := _g_.m.mstartfn; fn != nil {
        fn()
    }

    if _g_.m.helpgc != 0 {
        _g_.m.helpgc = 0
        stopm()
    } else if _g_.m != &m0 {
        acquirep(_g_.m.nextp.ptr())
        _g_.m.nextp = 0
    }
    schedule()
}
```

注意到，如果这个 M 没有和某个 P 绑定，就会调用 `acquirep` 来绑定。在准备工作做好后，便执行了核心的 `schedule` 函数，于是进入调度循环。

下面再详细讨论调度队列的结构。

G 的创建（通过 `newproc`），默认会复用之前的 G 对象，这个空闲的对象来自 P 的 cache（这是为什么有了 M 仍然需要 P 结构的一个理由：M 对应的系统进程不一定在同一个 CPU 上执行，于是寄存器、cache 信息不宜保存于 M 中）。当然，如果没有之前的 cache，就新建一个 G（通过 `malg` 分派空间）。特别注明，新 G 的栈空间是非常小的（2KB），只有需要时才会扩大，这允许一个程序运行相当大量的 G。

新建好的 G 被放置到当前 P 的待运行队列中（`runqput`）。而 P 的待运行

队列是分级的，分别是下一个执行的任务、P 的本地队列、全局队列。前两个由于限定于本 CPU，是不需要加锁访问的。如果 P 的本地队列满，那么说明 P 比较忙，这时执行 `runqputslow`，专门负责把一半任务分到全局队列，没有任务执行的 P 这时就可以去执行这些任务了。因此，这一切结束前，应当唤醒空闲的 P。

唤醒 P 调用了 `wakep` 函数。它最终在某个 P 上调用了 `startm`，基本逻辑为找到一个 M，然后执行它。没有空闲 M 则新建一个 (通过 `newm`，正是这里建立了系统进程，在 linux 下，此处调用了 `clone` 函数)。从此，正常情况下，这个新的 M 便会一直跑在这个空闲的 P 上，进行完整的调度逻辑。然而没有任务、阻塞过久等情况时，M 的 P 会被夺走，于是这个 M 便空闲出来。它之后可以被再次 `mstart`。

对于机制有所了解后，再来看具体的代码就很清楚了。下面是之前提到的重要函数的有所删减和注释的源码:

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

    // ...

    // if there is a locked g, resume it
    if _g_.m.lockedg != nil {
        stoplockedm()
        execute(_g_.m.lockedg, false) // Never returns.
    }

    // ...

    var gp *g
    var inheritTime bool

    // ...

    // try getting from global queue
```

```
    if gp == nil {
     if _g_.m.p.ptr().schedtick%61 == 0 && sched.runqsize > 0 {
            lock(&sched.lock)
            gp = globrunqget(_g_.m.p.ptr(), 1)
            unlock(&sched.lock)
        }
    }

    // try getting from local queue
    if gp == nil {
        gp, inheritTime = runqget(_g_.m.p.ptr())
        if gp != nil && _g_.m.spinning {
            throw(``schedule: spinning with local work'')
        }
    }

    // nothing to run, call findrunnable
    if gp == nil {
        gp, inheritTime = findrunnable()
    }

    execute(gp, inheritTime)
}

func findrunnable() (gp *g, inheritTime bool) {
    _g_ := getg()

    // ...
    // local runq
    if gp, inheritTime := runqget(_g_.m.p.ptr()); gp != nil {
        return gp, inheritTime
    }
```

```
// global runq
if sched.runqsize != 0 {
    lock(&sched.lock)
    gp := globrunqget(_g_.m.p.ptr(), 0)
    unlock(&sched.lock)
    if gp != nil {
        return gp, false
    }
}

// Poll network.
if netpollinited() && sched.lastpoll != 0 {
    if gp := netpoll(false); gp != nil {
        injectglist(gp.schedlink.ptr())
        casgstatus(gp, _Gwaiting, _Grunnable)
        return gp, false
    }
}

// ...

// random steal from other P's
for i := 0; i < int(4*gomaxprocs); i++ {
    _p_ := allp[fastrand1()%uint32(gomaxprocs)]
    var gp *g
    if _p_ == _g_.m.p.ptr() {
        gp, _ = runqget(_p_)
    } else {
        stealRunNextG := i > 2*int(gomaxprocs)
        gp = runqsteal(_g_.m.p.ptr(), _p_, stealRunNextG)
    }
    if gp != nil {
        return gp, false
```

```
        }
    }

    // ...
}

func newproc(siz int32, fn *funcval) {
    argp := add(unsafe.Pointer(&fn), sys.PtrSize)
    pc := getcallerpc(unsafe.Pointer(&siz))
    systemstack(func() {
        newproc1(fn, (*uint8)(argp), siz, 0, pc)
    })
}

func newproc1(fn *funcval, argp *uint8, narg int32,
              nret int32, callerpc uintptr) *g {
    _g_ := getg()

    // ...

    _p_ := _g_.m.p.ptr()
    newg := gfget(_p_)
    if newg == nil {
        newg = malg(_StackMin)
        casgstatus(newg, _Gidle, _Gdead)
        allgadd(newg)
    }

    // configure newg
    // ...

    runqput(_p_, newg, true)
```

```
    // ...

    return newg
}
```

5 Go channel

Go 中有两种 chan：分别是单个的 channel 和有 buffer 的 channel；有两种操作：`x <- c`, `c <- x`，分别是向其中添加内容和取出内容。构建 chan 的操作在代码生成时被翻译为 `runtime.makechan` 函数，添加和取出则分别翻译为 `runtime.chansend`、`runtime.chanrecv`。这几个函数在 chan.go 中实现。

Go 语言的 channel 实现并非针对高性能并行设计，而是为了实现高度并发环境下的通信设计的 [3]，它只是掩盖了底层的线程、锁等机制，提供一个易于使用的通信模式。它类似 Unix 的管道，在进程间提供了最基础的通信服务，但是比管道拥有更多的特性，比如支持类型。但是它不直接支持大部分并行模式，如基于共享变量的并行模式及广播等模式（不过，由于 Go 是系统级语言，可以使用 Go 实现这样的库或工具）。实现高性能并行主要需要靠用户的正确使用。

底层的执行基于进程，因此全部的通信基于内存复制而非内存共享。当然，由于可以在 channel 中传递指针，亦即传递一块内存，事实上内存共享模式也是支持的。我们后面再分析效率上的问题。

现在，我们具体看一下 Go 中 chan 的实现机制 [4]。

Channel 实现中信息由记录类型 Hchan 保存，其中记录如下一些信息：channel 容量、类型、元素 buffer、接收队列、发送队列。

Go 中的 channel 细分可以分为三种类型。下面是三种类型分别的构建方式：

```
c1 := make(chan int)
c2 := make(chan int, 10)
c3 := make(chan struct{})
```

具体分析三种 channel：

注意到，第一种 channel 中只有一个空位，即表示内容传输始终是一对一的，只要运行时确定一对发送者和接收者（各自取队首的 g），然后直接转交，

进行一次内存复制即可。由于没有竞争，这里使用最简单的加锁实现，不会影响性能。如果没有匹配，则将一个 sudog 加入发送者/接收者队列。sudog 是可以被复用的，它负责等待配对者出现而被唤醒。

第二种 channel 有一个元素 buffer，这是个传统的生产者消费者模式，为了性能，给整个 buffer 加锁是不理想的。于是这里采用了 CAS 操作来实现无锁访问。以 send 为例，在 buffer 未满的情况下，各个 sender 竞争写 sendx，竞争到的进行一次（不需要加锁的）写操作。读也是一样。如果 buffer 满了，则休眠等待唤醒。

第三种和第二种是类似的，只是不需要存储数据。它就作为信号量来使用。

自然，我们还需要实现一个高效的 select。首先对所有 channel 进行 shuffle。之后先对每一个 channel 尝试执行非阻塞的通信，如果失败（即等待的 channel 全部阻塞），则在每个 channel 的等待列表中加入自己，并阻塞，等待之后被唤醒。整个操作不需要对 channel 全部加锁，在很多情况下都不需要访问每个 channel。

有关 Go 的内存模型，更为详细的内存访问时机的讨论，还可以阅读 [1]。

知道了机制，我们可以再来看一下具体的代码。

用来保存 channel 的数据结构叫做`hchan`:

```
type hchan struct {
    qcount   uint           // total data in the queue
    dataqsiz uint           // size of the circular queue
 buf    unsafe.Pointer // points to an array of dataqsiz elements
    elemsize uint16
    closed   uint32
    elemtype *_type // element type
    sendx    uint   // send index
    recvx    uint   // receive index
    recvq    waitq  // list of recv waiters
    sendq    waitq  // list of send waiters
    lock     mutex
}
```

构建 channel 的函数`makechan` 即返回一个新的 hchan 对象。send 和 recv

的代码分别如下:

```
func chansend(t *chantype, c *hchan, ep unsafe.Pointer,
              block bool, callerpc uintptr) bool {
    // ...

    // check for failed operation without acquiring the lock
    if !block && c.closed == 0 &&
       ((c.dataqsiz == 0 && c.recvq.first == nil) ||
        (c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {
        return false
    }

    // ...

    lock(&c.lock)

    // if there is a sudog waiting in recvq
    if sg := c.recvq.dequeue(); sg != nil {
        send(c, sg, ep, func() { unlock(&c.lock) })
        return true
    }

    // if channel buffer not full
    if c.qcount < c.dataqsiz {
        qp := chanbuf(c, c.sendx)
        typedmemmove(c.elemtype, qp, ep)
        c.sendx++
        if c.sendx == c.dataqsiz {
            c.sendx = 0
        }
        c.qcount++
        unlock(&c.lock)
```

```
        return true
    }

    // ...

    // buffer is full, block
    gp := getg()
    mysg := acquireSudog()
    mysg.releasetime = 0
    if t0 != 0 {
        mysg.releasetime = -1
    }
    mysg.elem = ep
    mysg.waitlink = nil
    mysg.g = gp
    mysg.selectdone = nil
    gp.waiting = mysg
    gp.param = nil
    c.sendq.enqueue(mysg)
   goparkunlock(&c.lock, "chan send", traceEvGoBlockSend, 3)

    // someone woke us up.
    gp.waiting = nil
    gp.param = nil
    if mysg.releasetime > 0 {
        blockevent(int64(mysg.releasetime)-t0, 2)
    }
    releaseSudog(mysg)
    return true
}

func chanrecv(t *chantype, c *hchan, ep unsafe.Pointer,
              block bool) (selected, received bool) {
```

```
        // fairly similar to chansend
    }
```

参考文献

[1] The Go Memory Model.

[2] Rob Pike. The go programming language (flv), 2009.

[3] Rob Pike. Concurrency is not Parallelism, 2012.

[4] Dmitry Vyukov. Go channels on steroids, 2014.