Table of Contents

Implemention of golang 1.1

协程栈 1.1.1

概述 1.1.1.1

栈初始化 1.1.1.2

栈扩容 1.1.1.3

栈缩容 1.1.1.4

附录 1.1.1.5

参考资料 1.1.1.6

协程调度 1.1.2

协程状态 1.1.2.1

协程切换 1.1.2.2

调度时机 1.1.2.3

系统调用 1.1.2.3.1

管道读写 1.1.2.3.2

抢占式调度 1.1.2.3.3

数据结构 1.1.2.4

网络 1.1.3

数据结构 1.1.3.1

编程示例 1.1.3.2

内部实现 1.1.3.3

内部实现(二) 1.1.3.4

内部实现(三) 1.1.3.5

Socket超时 1.1.3.6

附录 1.1.3.7

附录二 1.1.3.8

附录三 1.1.3.9

内存管理 1.1.4

Stack内存管理	1.1.4.1
Heap管理	1.1.4.2
核心数据结构	1.1.4.3
内存分配算法	1.1.4.4
核心API实现	1.1.4.5

Implemention of golang

本书主要描述golang的内部实现，主要包括“协程调度”、“网络实现”、“内存管理”、“垃圾回收”等章节。

作者 丁凯：tracymacding@gmail.com

概述

说明

计算机中的栈一个很大的应用场合使用在函数调用中。我们这里简单说说golang的协程栈布局，学过计算机的应该都不会陌生。

程序事例

```
package main

func f(a, b int) int {
    sum := 0
    sum = a + b
    for i := 0; i < 1000; i++ {
        println("sum is:", sum)
    }
    return sum
}

func main() {
    f(1, 2)
}
```

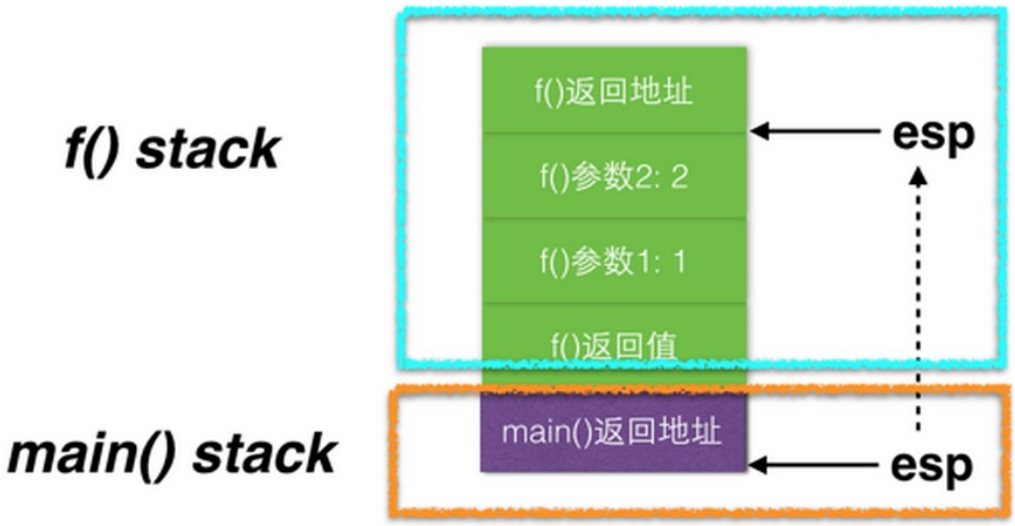
汇编代码

```
(gdb) disas
Dump of assembler code for function main.main:
0x00000000004010b0 <main.main+0>:       mov    %fs:0xfffffffffff
ffff8,%rcx
0x00000000004010b9 <main.main+9>:       cmp    0x10(%rcx),%rsp
0x00000000004010bd <main.main+13>:      jbe    0x4010de <main.ma
in+46>
0x00000000004010bf <main.main+15>:      sub    $0x18,%rsp
```

```
0x00000000004010c3 <main.main+19>:      movq   $0x1,(%rsp)
0x00000000004010cb <main.main+27>:      movq   $0x2,0x8(%rsp)
0x00000000004010d4 <main.main+36>:      callq  0x401000 <main.f>
0x00000000004010d9 <main.main+41>:      add    $0x18,%rsp
0x00000000004010dd <main.main+45>:      retq

0x00000000004010de <main.main+46>:      callq  0x44abd0 <runtime
.morestack_noctxt>
0x00000000004010e3 <main.main+51>:      jmp    0x4010b0 <main.ma
in>
0x00000000004010e5 <main.main+53>:      add    %al,(%rax)
0x00000000004010e7 <main.main+55>:      add    %al,(%rax)
0x00000000004010e9 <main.main+57>:      add    %al,(%rax)
0x00000000004010eb <main.main+59>:      add    %al,(%rax)
0x00000000004010ed <main.main+61>:      add    %al,(%rax)
0x00000000004010ef <main.main+63>:      add    %ah,-0x75(%rax,%r
cx,2)
End of assembler dump.
(gdb) disas
Dump of assembler code for function main.f:
0x0000000000401000 <main.f+0>:  mov    %fs:0xfffffffffffffff8,%r
cx
0x0000000000401009 <main.f+9>:  cmp    0x10(%rcx),%rsp
0x000000000040100d <main.f+13>: jbe    0x401097 <main.f+151>
0x0000000000401013 <main.f+19>: sub    $0x20,%rsp
0x0000000000401017 <main.f+23>: mov    0x28(%rsp),%rbx
0x000000000040101c <main.f+28>: mov    0x30(%rsp),%rbp
0x0000000000401021 <main.f+33>: add    %rbp,%rbx
0x0000000000401024 <main.f+36>: mov    %rbx,0x10(%rsp)
0x0000000000401029 <main.f+41>: xor    %eax,%eax
0x000000000040102b <main.f+43>: mov    %rax,0x18(%rsp)
0x0000000000401030 <main.f+48>: cmp    $0x3e8,%rax
0x0000000000401036 <main.f+54>: jge    0x401088 <main.f+136>
......
0x0000000000401088 <main.f+136>:        mov    0x10(%rsp),%rbx
0x000000000040108d <main.f+141>:        mov    %rbx,0x38(%rsp)
0x0000000000401092 <main.f+146>:        add    $0x20,%rsp
```

执行过程中**stack**变化情况

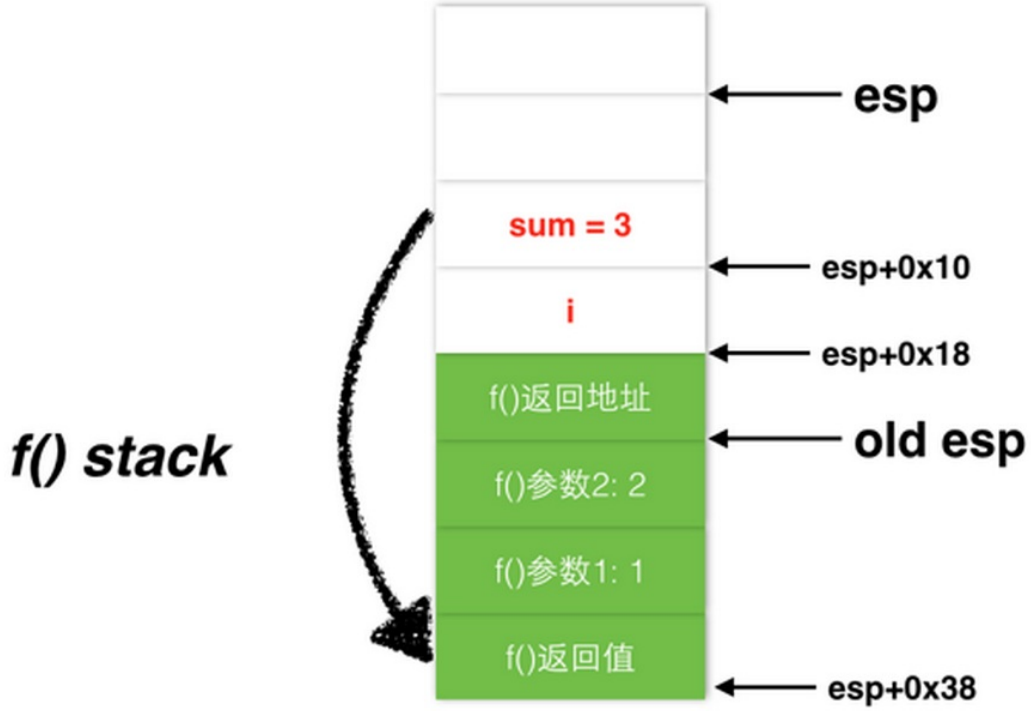
在main调用f()时，协程堆栈情况：



注意：这里的返回地址是由call指令自动push至esp所指向内存，而参数内容则是由调用者main函数设置的，如下代码：

```
// we have 2 argument and 1 return value
// so must reserve 24 bytes in amd64(0x18)
0x00000000004010bf <main.main+15>:      sub    $0x18,%rsp
0x00000000004010c3 <main.main+19>:      movq   $0x1,(%rsp)
0x00000000004010cb <main.main+27>:      movq   $0x2,0x8(%rsp)
0x00000000004010d4 <main.main+36>:      callq  0x401000 <main.f>
```

在f函数内部执行时，会扩充当前stack，为了临时存储一些本地变量，如sum等，则f执行时堆栈情况如下：



可以看到为本地变量sum和i自动在栈上分配了存储空间，计算sum，然后将sum的值存储到f()返回值该去的地方((esp) + 0x38)

可以简单看看main.f()的主要汇编代码

```
// sub esp to allocate space for local variable
0x0000000000401013 <main.f+19>: sub    $0x20,%rsp
// get parameters, compute and store sum
0x0000000000401017 <main.f+23>: mov    0x28(%rsp),%rbx
0x000000000040101c <main.f+28>: mov    0x30(%rsp),%rbp
0x0000000000401021 <main.f+33>: add    %rbp,%rbx
// store sum in (esp) + 0x10
0x0000000000401024 <main.f+36>: mov    %rbx,0x10(%rsp)
// for loop assemble code
0x0000000000401029 <main.f+41>: xor    %eax,%eax
0x000000000040102b <main.f+43>: mov    %rax,0x18(%rsp)
0x0000000000401030 <main.f+48>: cmp    $0x3e8,%rax
0x0000000000401036 <main.f+54>: jge    0x401088 <main.f+136>
......
// store sum into return value address(esp + 0x38)
// and shrink stack((%esp) + 0x20) and return to main
0x0000000000401088 <main.f+136>:        mov    0x10(%rsp),%rbx
0x000000000040108d <main.f+141>:        mov    %rbx,0x38(%rsp)
0x0000000000401092 <main.f+146>:        add    $0x20,%rsp
```

参考资料

- http://www.cs.nyu.edu/courses/fall04/V22.0201-003/ia32_chap_03.pdf

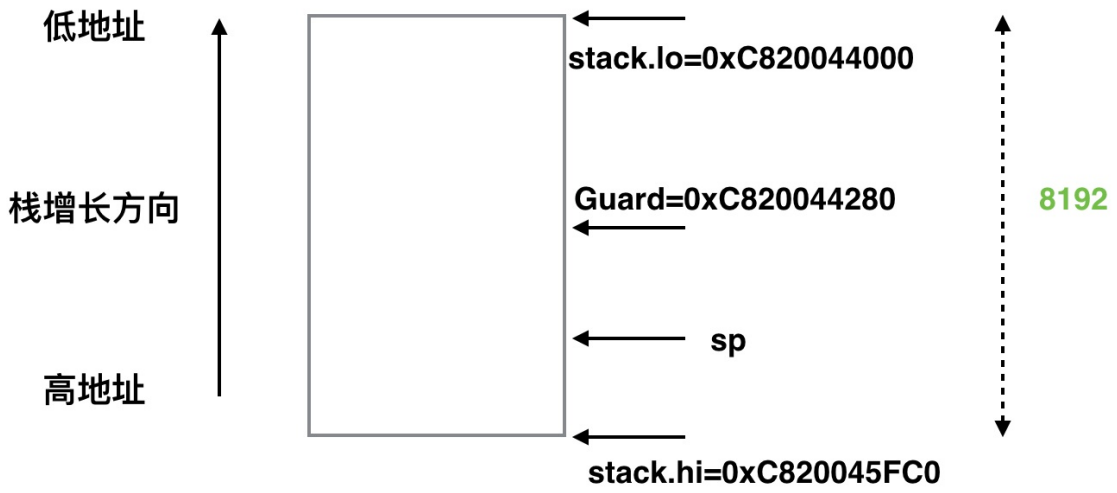
栈初始化

主协程初始化

Golang的主协程指的是运行main函数的协程，而子协程指的是在程序运行过程中由主协程创建的协程。每个线程(m)只会有一个主协程，而子协程可能会有很多很多。

子协程和主协程在概念和内部实现上几乎没有任何区别，唯一的不同在于它们的初始栈大小不同。

我们先看看测试过程中生成的主协程堆栈示例。我测试代码中就生成了一个主协程，通过反汇编代码看到他的样子大概如下：



主协程启动

分析连接器（libinit()）发现go程序的入口函数是_rt0_amd64_linux（linux amd64机器）

子协程初始化

Golang子协程堆栈在协程被创建时也一并创建，代码如下：

```
unc newproc1(fn *funcval, argp *uint8, narg int32, nret int32, c
allerpc uintptr) *g {
    _g_ := getg()

    ......

    _p_ := _g_.m.p.ptr()
    newg := gfget(_p_)
    if newg == nil {
        // 创建协程栈
        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.
    }
    ......

    totalSize := 4*regSize + uintptr(siz) // extra space in case
 of reads slightly beyond frame
    if hasLinkRegister {
        totalSize += ptrSize
    }
    totalSize += -totalSize & (spAlign - 1) // align to spAlign
    // 新协程的栈顶计算，将栈的基地址减去参数占用的空间
    sp := newg.stack.hi - totalSize
    spArg := sp
    if hasLinkRegister {
        // caller's LR
        *(*unsafe.Pointer)(unsafe.Pointer(sp)) = nil
        spArg += ptrSize
    }
    ...
    // 设置新建协程的栈顶sp
    newg.sched.sp = sp
}

// Allocate a new g, with a stack big enough for stacksize bytes
.
func malg(stacksize int32) *g {
```

```
    newg := new(g)
    if stacksize >= 0 {
        stacksize = round2(_StackSystem + stacksize)
        systemstack(func() {
            newg.stack, newg.stkbar = stackalloc(uint32(stacksiz
e))
        })
        // 设置stackguard，在协程栈不够用时再重新申请新的栈
        newg.stackguard0 = newg.stack.lo + _StackGuard
        newg.stackguard1 = ^uintptr(0)
        newg.stackAlloc = uintptr(stacksize)
    }
    return newg
}
```

在go1.5.1版本中，_StackMin大小被定义为2048(而在1.3.2版本中该值还是8192)，也即每个协程的初始堆栈大小为2KB，相当小了。缩小该值的好处是即使创建了很多的协程也不会导致内存使用的急剧增长。

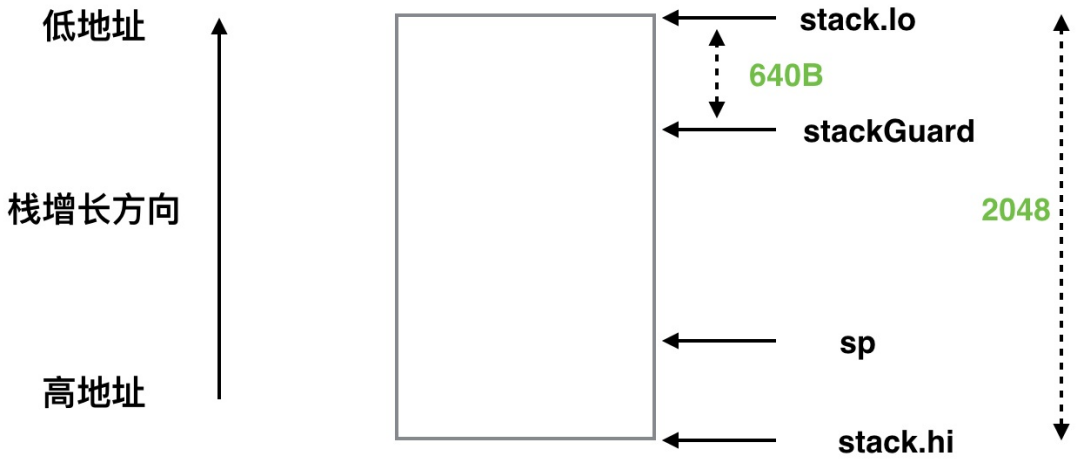
另外，在协程栈空间被分配出来后，还需要作一些其他的初始化，主要是协程栈顶的设置以及堆栈保护的设置。

- 栈顶的设置方法比较简单，将当前栈的起始地址减去参数占用的空间即可(注意栈是从高地址向低地址延伸的)。
- 栈保护的设置指的是设置一个临界点，当sp到达该临界点时认为栈空间可能会不足，需要进行栈扩容。当前版本的协程栈保护大小事640B。

以下是我对源码加入一些调试信息后打印的协程创建堆栈的详细信息：

```
create goroutine, stack:{stackLow: 859530534912, stackHi:8595305
36928, sp:859530536880, stackguard:859530535552}
```

根据上面的分析以及打印信息可以大致勾勒出协程初始化状态的堆栈：

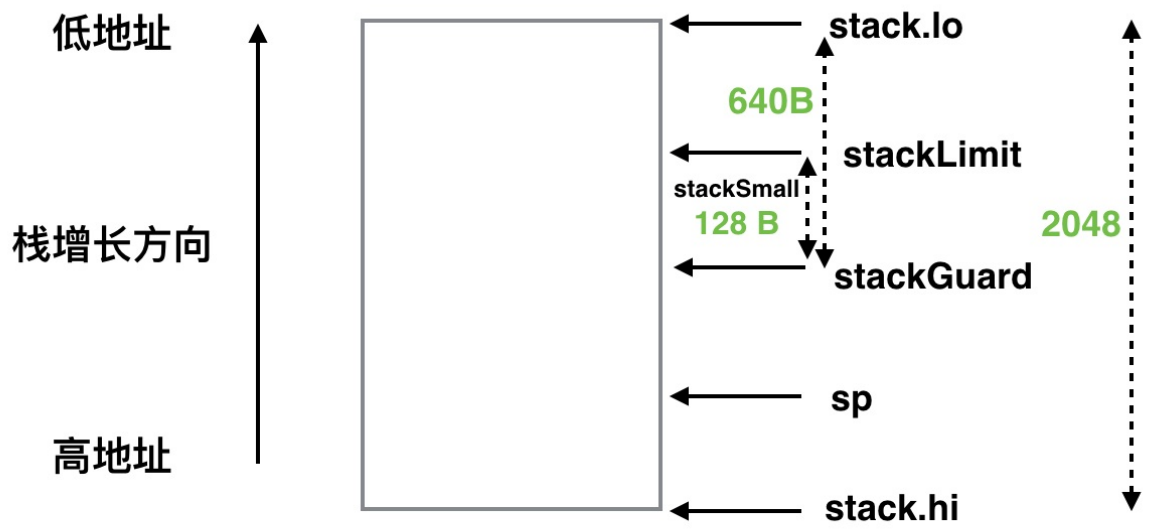

低地址
栈增长方向
高地址
stack.lo
640B
stackGuard
2048
sp
stack.hi

栈扩容

协程堆栈扩容

协程栈详细布局

我们前面说到，在创建一个协程时就为其创建了一个初始的栈，用来执行函数调用。协程栈的大概布局情况如下：



这里不仅弄出了stackGuard,还弄了一个stackLimit，至于它们有什么用途，我们会在下面仔细描述。

扩容

我们前面了解到，链接器在每个函数调用的开始部分会增加一段代码，用于检测是否需要进行栈扩容。为此，我们有以下几个问题需要解决：

- 当前函数需要占用的栈空间
- 栈空间不足的判断条件
- 如何进行扩容

我们接下来逐个击破。

计算函数栈空间

一个函数占用的栈空间主要由以下几个部分组成：

- 本地变量
- 函数调用其他函数的参数
- 函数调用其他参数的返回值

这里注意的是函数调用时，参数和返回值使用的空间也计算在调用者的使用空间上。

写了一个简单测试程序，反汇编该程序可以清晰看出每个函数需要的栈大小：

```
package main

func f(a, b int) (int, int) {
    sum := 0
    elements := make([]int, 100)
    for _, i := range elements {
        sum += i
    }
    sum += a
    sum += b
    return sum, a + b
}

func main() {
    f(1, 2)
}
```

```
go tool compile -S test.go

"".f t=1 size=176 value=0 args=0x20 locals=0x320
    0x0000 00000 (test.go:9)    TEXT    "".f(SB), $800-32
    0x0000 00000 (test.go:9)    MOVQ    (TLS), CX
    0x0009 00009 (test.go:9)    LEAQ    -672(SP), AX
    0x0011 00017 (test.go:9)    CMPQ    AX, 16(CX)

"".main t=1 size=64 value=0 args=0x0 locals=0x20
    0x0000 00000 (test.go:24)   TEXT    "".main(SB), $32-0
    0x0000 00000 (test.go:24)   MOVQ    (TLS), CX
    0x0009 00009 (test.go:24)   CMPQ    SP, 16(CX)
```

可以看到，f函数的栈大小是800字节，因为在f内部申请了大小为100的数组。而main函数的栈大小是32字节，因为它只需要向f传递两个int，以及接收两个int的返回值。

判断栈空间不足

上一节我们阐述了函数的栈空间计算方法。接下来我们就要看看golang如何判断栈空间不足了，这也是比较精彩的部分。

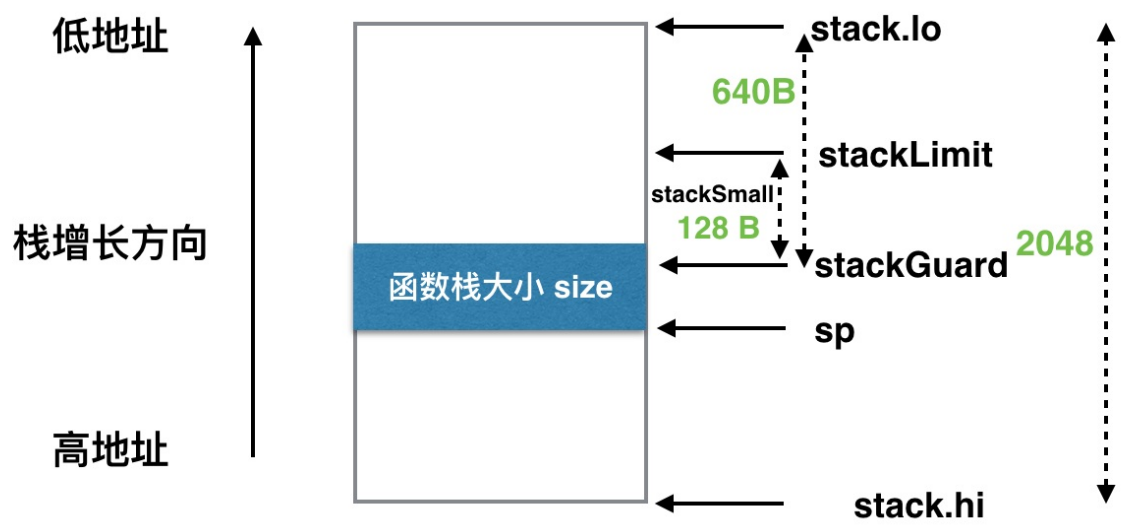
golang检测栈扩容的主要条件是SP是否达到了栈保护的边界，也就是我们前面途中的StackGuard。基本思想是如果SP小于StackGuard，那么需要进行栈扩容，否则无需栈扩容。

但是这里有个问题是：因为每个函数调用都会作这样的检查，对于函数调用的开销会增加，而且这种增加是无条件的。

为了避免该问题，Golang作了优化：对于极小的，明显不用扩容就不做检查了。我们前面看到的stackLimit就开始发挥作用了。

函数栈极小，无需扩容

这种情况下，该函数需要的栈空间极小，这时候压根不需要作检查，如下图：



通过上图看到，如果函数f需要的栈大小小于stackSmall=128B, 且此时sp还是小于stackguard，那么这时候认为它还是安全的，无需进行栈扩容。
这点也很好理解：如果当前sp还位于安全区域，而且此时调用的函数需要的栈很小，不会触及stack.lo的话，确实没有必要再去给它分配新的栈。

为此，写了个简单的测试程序并对其进行反汇编：

```
package main

func f(a, b int) (int, int) {
    sum := 0
    elements := make([]int, 10)
    for _, i := range elements {
        sum += i
    }
    sum += a
    sum += b
    return sum, a + b
}

func main() {
    f(1, 2)
}
```

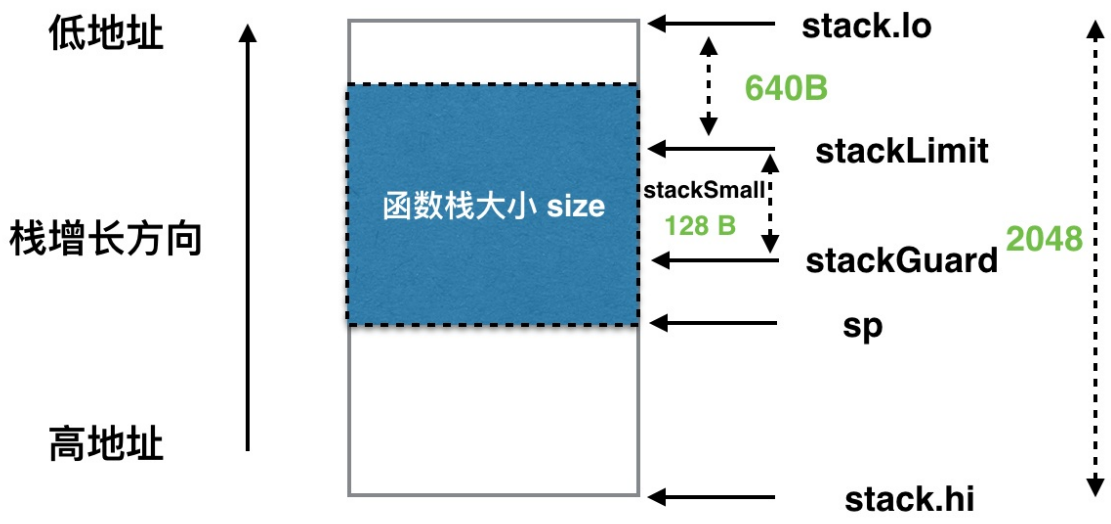
对函数f的反汇编结果如下：

```
"".f t=1 size=128 value=0 args=0x20 locals=0x50
    0x0000 00000 (test.go:9)    TEXT    "".f(SB), $80-32
    0x0000 00000 (test.go:9)    SUBQ    $80, SP
    0x0004 00004 (test.go:9)    MOVQ    "".a+88(FP), R9
    0x0009 00009 (test.go:9)    MOVQ    "".b+96(FP), R8
    0x000e 00014 (test.go:9)    FUNCDATA    $0, gclocals·a8eabfc
4a4514ed6b3b0c61e9680e440(SB)
    0x000e 00014 (test.go:9)    FUNCDATA    $1, gclocals·33cdecc
ccebe80329f1fdbee7f5874cb(SB)
    0x000e 00014 (test.go:12)   MOVQ    $0, DX
    0x0010 00016 (test.go:13)   LEAQ    "".autotmp_0004(SP), DI
   ......
```

可以看到，由于f使用的的堆栈很小(80B)，在程序的开始部分并没有出现栈扩容的判断。

函数栈适中，需要判断

这种情况就真的需要插入额外的判断指令。



这时候判断需要栈扩容的条件是函数延伸的栈不应该超过stackLimit的限制,转化为数学表达：

$$sp-frameSize < stackLimit$$
=>

$$sp-frameSize < stackGuard-stackSmall$$
=>

$$sp-frameSize + stackSmall < stackGuard$$
=>

$$sp-frameSize + 128 < stackGuard$$

类似上面，写了另外一个测试程序并进行反汇编：

```
package main

func f(a, b int) (int, int) {
    sum := 0
    elements := make([]int, 100)
    for _, i := range elements {
        sum += i
    }
    sum += a
    sum += b
    return sum, a + b
}

func main() {
    f(1, 2)
}
```

```
"".f t=1 size=176 value=0 args=0x20 locals=0x320
    0x0000 00000 (test.go:9)    TEXT    "".f(SB), $800-32
    0x0000 00000 (test.go:9)    MOVQ    (TLS), CX
    // 就是sp-800+128与stackGuard对比
    0x0009 00009 (test.go:9)    LEAQ    -672(SP), AX
    0x0011 00017 (test.go:9)    CMPQ    AX, 16(CX)
    0x0015 00021 (test.go:9)    JLS 158
```

可以看到，这里面的判断逻辑与我们前面提到的是相符合的。

这里其实遗留了两个疑问：

1. %fs:0xfffffffffffffff8,%rcx这里存储的到底是什么？猜测应该是线程相关变量,可能指的是正运行m的g；而0x10(%rcx)代表的是g的stackguard0，这样就能讲通这个比较的意义。关于%fs寄存器的相关说明可参考 http://www.airs.com/blog/archives/44，写的很不错；
2. 这个栈容量检测的汇编代码是谁插入的？根据一些介绍说是linker，有待仔细思考。

栈空间扩容

对协程的栈进行扩容必然是原有堆栈空间不足，因此，我们首先需要切换到该线程的堆栈上来调用扩容函数。否则就变成了鸡生蛋和蛋生鸡的问题了：要调用新函数来进行栈扩容，而调用新函数又需要新的栈。

其次，在新的栈空间申请成功后，我们还需要将现有栈的内容全部拷贝过去。拷贝完成后还得继续现有的函数流程走下去(我们需要能够从线程堆栈切换回协程堆栈)。因此，在调用扩容函数时需要将一些当前的运行环境保存下来。

让我们接下来看看具体实现：

```
TEXT runtime·morestack(SB),NOSPLIT,$0-0
    // Cannot grow scheduler stack (m->g0).
    get_tls(CX)
    MOVQ    g(CX), BX
    MOVQ    g_m(BX), BX
    MOVQ    m_g0(BX), SI
    CMPQ    g(CX), SI
    JNE     2(PC)
    INT     $3

    // Cannot grow signal stack (m->gsignal).
    MOVQ    m_gsignal(BX), SI
    CMPQ    g(CX), SI
    JNE     2(PC)
    INT     $3

    // Called from f.
    // Set m->morebuf to f's caller.
```

```
// ??? what does this do?
MOVQ    8(SP), AX       // f's caller's PC
MOVQ    AX, (m_morebuf+gobuf_pc)(BX)
LEAQ    16(SP), AX      // f's caller's SP
MOVQ    AX, (m_morebuf+gobuf_sp)(BX)
get_tls(CX)
// SI is current go-routine
// 将当前申请扩容stack的协程记录在m_morebuf中
// 这样后面切换到m.g0协程分配堆栈成功后知道返回到
// 哪个协程继续执行。
MOVQ    g(CX), SI
MOVQ    SI, (m_morebuf+gobuf_g)(BX)

// 将申请分配新堆栈的协程执行环境记录下来
// 这样下次返回该协程时知道从哪继续执行
// why 0(SP) is f's PC?
MOVQ    0(SP), AX // f's PC
MOVQ    AX, (g_sched+gobuf_pc)(SI)
MOVQ    SI, (g_sched+gobuf_g)(SI)
LEAQ    8(SP), AX // f's SP
MOVQ    AX, (g_sched+gobuf_sp)(SI)
// what does DX store?
MOVQ    DX, (g_sched+gobuf_ctxt)(SI)
MOVQ    BP, (g_sched+gobuf_bp)(SI)

// Call newstack on m->g0's stack.
// 切换到线程的调度协程和线程堆栈
MOVQ    m_g0(BX), BX
// switch to m.g0
MOVQ    BX, g(CX)
// what does this mean?
// 切换到线程堆栈
MOVQ    (g_sched+gobuf_sp)(BX), SP
// runtime.newstack() never return
CALL    runtime·newstack(SB)
MOVQ    $0, 0x1003  // crash if newstack returns
RET
```

最终调用了newstack来进行实际的栈扩容，让我们继续深入看看栈扩容到底如何实现：

```
func newstack() {
    // gp是申请堆栈扩容的协程
    gp := thisg.m.curg

    ......

    // Allocate a bigger segment and move the stack.
    oldsize := int(gp.stackAlloc)
    // 新的栈大小是原来的两倍
    newsize := oldsize * 2
    if uintptr(newsize) > maxstacksize {
        print("runtime: goroutine stack exceeds ", maxstacksize,
 "-byte limit\n")
        throw("stack overflow")
    }

    casgstatus(gp, _Gwaiting, _Gcopystack)

    // The concurrent GC will not scan the stack while we are do
ing the copy since
    // the gp is in a Gcopystack status.
    // 执行堆栈扩容并将原有堆栈数据拷贝至新栈
    copystack(gp, uintptr(newsize))
    if stackDebug >= 1 {
        print("stack grow done\n")
    }
    casgstatus(gp, _Gcopystack, _Grunning)
    gogo(&gp.sched)
}

// 申请新的栈空间并将原有栈数据拷贝至这里
func copystack(gp *g, newsize uintptr) {
    if gp.syscallsp != 0 {
        throw("stack growth not allowed in system call")
    }
    old := gp.stack
    if old.lo == 0 {
```

```
        throw("nil stackbase")
    }
    // 原有堆栈使用的空间
    used := old.hi - gp.sched.sp

    // allocate new stack
    // newstkbar是什么？
    // 0xfc是什么？
    new, newstkbar := stackalloc(uint32(newsize))
    if stackPoisonCopy != 0 {
        fillstack(new, 0xfd)
    }

    ......

    // adjust pointers in the to-be-copied frames
    // 这里主要调整g的一些调度相关参数
    // 如果它们存储在老的栈上面，需要将它们拷贝到新栈上
    var adjinfo adjustinfo
    adjinfo.old = old
    adjinfo.delta = new.hi - old.hi
    gentraceback(^uintptr(0), ^uintptr(0), 0, gp, 0, nil, 0x7fff
ffff, adjustframe, noescape(unsafe.Pointer(&adjinfo)), 0)

    // adjust other miscellaneous things that have pointers into
 stacks.
    adjustctxt(gp, &adjinfo)
    adjustdefers(gp, &adjinfo)
    adjustpanics(gp, &adjinfo)
    adjustsudogs(gp, &adjinfo)
    adjuststkbar(gp, &adjinfo)

    // copy the stack to the new location
    // 0xfb又是什么？
    if stackPoisonCopy != 0 {
        fillstack(new, 0xfb)
    }
    // 数据拷贝，老的堆栈数据拷贝到新堆栈
    memmove(unsafe.Pointer(new.hi-used), unsafe.Pointer(old.hi-u
sed), used)
```

```
    // copy old stack barriers to new stack barrier array
    newstkbar = newstkbar[:len(gp.stkbar)]
    copy(newstkbar, gp.stkbar)

    // Swap out old stack for new one
    // 切换到新堆栈上工作
    gp.stack = new
    gp.stackguard0 = new.lo + _StackGuard
    gp.sched.sp = new.hi - used
    oldsize := gp.stackAlloc
    gp.stackAlloc = newsize
    gp.stkbar = newstkbar

    // 释放老的堆栈
    if stackPoisonCopy != 0 {
        fillstack(old, 0xfc)
    }
    stackfree(old, oldsize)
}

func fillstack(stk stack, b byte) {
    for p := stk.lo; p < stk.hi; p++ {
        *(*byte)(unsafe.Pointer(p)) = b
    }
}
```

参考资料

- https://docs.google.com/document/d/1wAaf1rYoM4S4gtnPh0zOlGzWtrZFQ5suE8qr2sD8uWQ/pub

栈缩容

前面我们说过协程堆栈扩容，堆栈扩容以后，协程的堆栈空间可能变得比较大，但是这部分空间后续不一定能用得上，

附录

gdb调试汇编指令

关于GDB调试C程序的常用命令与手段就不多说了，这里主要介绍一下如何对C程序做到汇编指令级别的调试。 首先是获取汇编代码，这可以通过disassemble命令或x命令或类似的命令：

```
// 下断点
(gdb) b test.go:28
The current source language is "auto; currently minimal".
// 查看断点所在函数的反汇编代码
(gdb) disassemble
Dump of assembler code for function main.main:
0x00000000004010b0 <main.main+0>:    mov    %fs:0xfffffffffffffff8,%rcx
0x00000000004010b9 <main.main+9>:    cmp    0x10(%rcx),%rsp
0x00000000004010bd <main.main+13>:    jbe    0x40112b <main.main+123>

// 查看下一条要执行的汇编代码,以下两种方法均可
// 如果要调试汇编代码好像必须执行这个才可以
(gdb) x/i $pc
0x4010c3 <main.main+19>:    movq   $0x1,0x30(%rsp)
(gdb) x/i $rip
0x4010c3 <main.main+19>:    movq   $0x1,0x30(%rsp)

// 单步执行si or ni
(gdb) x/i $pc
0x401000 <main.f>:    mov    %fs:0xfffffffffffffff8,%rcx
(gdb) si
0x0000000000401009    9    func f(a, b int) (int, int) {
(gdb) x/i $pc
0x401009 <main.f+9>:    mov    0x10(%rcx),%rsi

// 查看寄存器和内存地址的值
(gdb) p/x $rcx
$1 = 0xc820000f00
(gdb) x/1ug 0xc820000f00
0xc820000f00:    859530522624
(gdb) x/1ug 0xc820000f08
0xc820000f08:    859530524640
(gdb) x/1ug 0xc820000f10
0xc820000f10:    859530523264
(gdb)
```

参考资料

- http://www.lenky.info/archives/2012/05/1694

参考资料

- https://golang.org/doc/asm

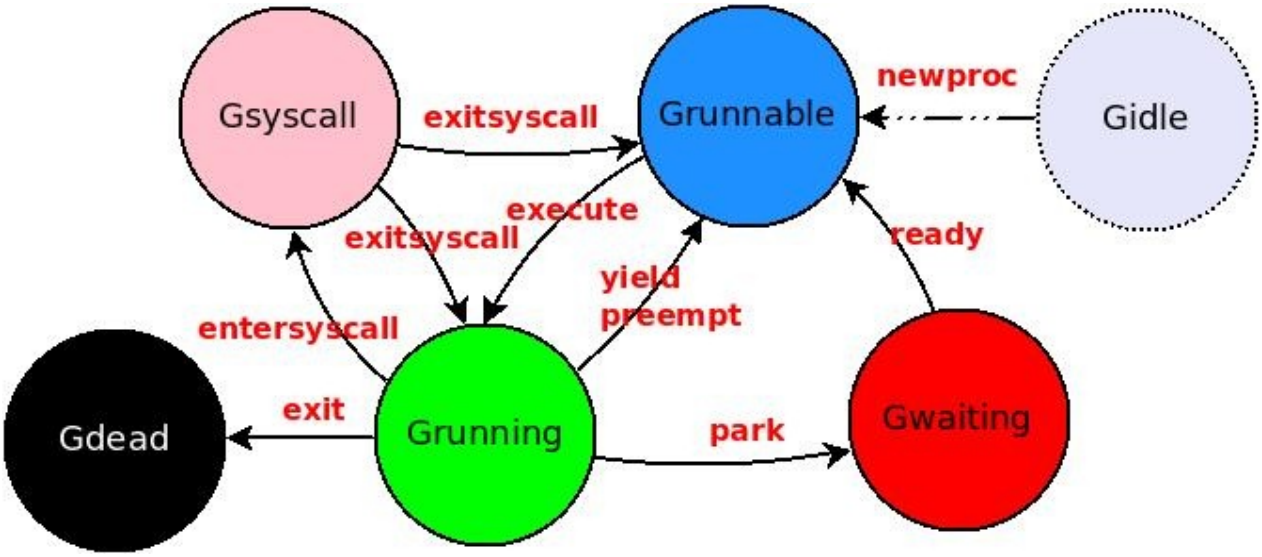
协程状态

状态总览

在讲解操作系统进程调度的部分时，几乎所有的书籍都会先列出一张进程的状态迁移图，通过状态图，能很清晰的把进程调度的每个环节串联起来，方便理解。

Go运行时的调度器其实可以看成OS调度器的某种简化版 本，一个goroutine在其生命周期之中，同样包含了各种状态的变换。弄清了这些状态及状态间切换的原理，对搞清整个Go调度器会非常有帮助。

以下是我总结的一张goroutine的状态迁移图，圆形框表示状态，箭头及文字信息表示切换的方向和条件：



状态详述

下面来简单分析一下， 其中状态 Gidle 在Go调度器代码中并没有被眞正被使用到，所以直接忽略。 事实上，一旦runtime新建了一个goroutine结构，就会将其状态置为Grunnable并加入到任务队列中，因此我们以该状态作为起点进行介绍。

Grunnable

Golang中，一个协程在以下几种情况下会被设置为 Grunnable状态：

创建

Go 语言中，包括用户入口函数main·main的执行goroutine在内的所有任务，都是通过runtime·newproc -> runtime·newproc1 这两个函数创建的，前者其实就是对后者的一层封装，提供可变参数支持，Go语言的go关键字最终会被编译器映射为对runtime·newproc的调用。当runtime·newproc1完成了资源的分配及初始化后，新任务的状态会被置为Grunnable，然后被添加到当前 P 的私有任务队列中，等待调度执行。相关初始化代码如下:

```
G* runtime·newproc1(FuncVal *fn, byte *argp, int32 narg, int32 n
ret, void *callerpc)
{
    G *newg;
    P *p;
    int32 siz;
    ......
    // 获取当前g所在的p，从p中创建一个新g(newg)
    p = g->m->p;
    if((newg = gfget(p)) == nil) {
        ......
    }
    ......
    // 设置Goroutine状态为Grunnable
    runtime·casgstatus(newg, Gdead,
        Grunnable);
    .....
    // 新创建的g添加到run队列中
    runqput(p, newg);
    ......
}
```

阻塞任务唤醒

当某个阻塞任务（状态为Gwaiting）的等待条件满足而被唤醒时—如一个任务G#1向某个channel写入数据将唤醒之前等待读取该channel数据的任务G#2——G#1通过调用runtime·ready将G#2状态重新置为Grunnable并添加到任务队列中。关于任务阻塞，稍后还很详细介绍。

```
// Mark gp ready to run.
void
runtime·ready(G *gp)
{
    uint32 status;
    status = runtime·readgstatus(gp);
    // Mark runnable.
    g->m->locks++;
    if((status&~Gscan) != Gwaiting){
        dumpgstatus(gp);
        runtime·throw("bad g->status in ready");
    }
    // 设置被唤醒的g状态从Gwaiting转变至Grunnable
    runtime·casgstatus(gp, Gwaiting, Grunnable);
    // 添加到运行队列中
    runqput(g->m->p, gp);
    if(runtime·atomicload(&runtime·sched.npidle) != 0 && runtime
·atomicload(&runtime·sched.nmspinning) == 0)
    // 看起来这是个比较重要的函数，但还不是很理解
    wakep();
    g->m->locks--;
    if(g->m->locks == 0 && g->preempt)
        g->stackguard0 = StackPreempt;
}
```

其他

另外的路径是从Grunning和Gsyscall状态变换到Grunnable，我们也都合并到后面介绍。 总之，处于Grunnable的任务一定在某个任务队列中，随时等待被调度执行。

Grunning

所有状态为Grunnable的任务都可能通过findrunnable函数被调度器（P&M）获取，进而通过execute函数将其状态切换到Grunning, 最后调用runtime·gogo加载其上下文并执行。

```
// One round of scheduler: find a runnable goroutine and execute
```

```
 it. Never returns.
static void
schedule(void)
{
    G *gp;
    uint32 tick;

    if(g->m->locks)
        runtime·throw("schedule: holding locks");

    if(g->m->lockedg) {
        stoplockedm();
        execute(g->m->lockedg);  // Never returns.
    }

top:
    if(runtime·sched.gcwaiting) {
        gcstopm();
        goto top;
    }

    gp = nil;
    // 挑一个可运行的g，并执行
    ......
    if(gp == nil) {
        gp = findrunnable();  // blocks until work is available
        resetspinning();
    }
    ......
    execute(gp);
}

// Schedules gp to run on the current M.
// Never returns.
static void
execute(G *gp)
{
    int32 hz;
    // 状态从Grunnable转变为Grunning
    runtime·casgstatus(gp, Grunnable, Grunning);
```

```
    gp->waitsince = 0;
    gp->preempt = false;
    gp->stackguard0 = gp->stack.lo + StackGuard;
    g->m->p->schedtick++;
    g->m->curg = gp;
    gp->m = g->m;

    // Check whether the profiler needs to be turned on or off.
    hz = runtime·sched.profilehz;
    if(g->m->profilehz != hz)
        runtime·resetcpuprofiler(hz);
    // 真正执行g
    runtime·gogo(&gp->sched);
}
```

前面讲过Go本质采用一种协作式调度方案，一个正在运行的任务，需要通过调用yield的方式显式让出处理器；在Go1.2之后，运行时也开始支持一定程度的任务抢占——当系统线程sysmon发现某个任务执行时间过长或者runtime判断需要进行垃圾收集时，会将任务置为"可被抢占"的，当该任务下一次函数调用时， 就会让出处理器并重新切会到Grunnable状态。关于Go1.2中抢占机制的实现细节，后面又机会再做介绍。

Gsyscall

Go运行时为了保证高的并发性能，当会在任务执行OS系统调用前，先调用runtime·entersyscall函数将自己的状态置为Gsyscall——如果系统调用是阻塞式的或者执行过久，则将当前M与P分离——当系统调用返回后，执行线程调用runtime·exitsyscall尝试重新获取P，如果成功且当前任务没有被抢占，则将状态切回Grunning并继续执行；否则将状态置为Grunnable，等待再次被调度执行。

```
// Puts the current goroutine into a waiting state and calls unl
ockf.
// If unlockf returns false, the goroutine is resumed.
void
runtime·park(bool(*unlockf)(G*, void*), void *lock, String reaso
n)
{
    void (*fn)(G*);

    g->m->waitlock = lock;
    g->m->waitunlockf = unlockf;
    g->waitreason = reason;
    fn = runtime·park_m;
    runtime·mcall(&fn);
}
// runtime·park continuation on g0.
void
runtime·park_m(G *gp)
{
    bool ok;
    // 设置当前状态从Grunning-->Gwaiting
    runtime·casgstatus(gp, Grunning, Gwaiting);
    // 当前g放弃m
    dropg();

    if(g->m->waitunlockf) {
        ok = g->m->waitunlockf(gp, g->m->waitlock);
        g->m->waitunlockf = nil;
        g->m->waitlock = nil;
        if(!ok) {
            runtime·casgstatus(gp, Gwaiting, Grunnable);
            execute(gp);  // Schedule it back, never returns.
        }
    }

    schedule();
}
```

Gwaiting

当一个任务需要的资源或运行条件不能被满足时，需要调用runtime·park函数进入该状态，之后除非等待条件满足，否则任务将一直处于等待状态不能执行。除了之前举过的channel的例子外，Go语言的定时器、网络IO操作都可能引起任务的阻塞。

```
// runtime·park continuation on g0.
void
runtime·park_m(G *gp)
{
    bool ok;

    runtime·casgstatus(gp, Grunning, Gwaiting);
    dropg();

    if(g->m->waitunlockf) {
        ok = g->m->waitunlockf(gp, g->m->waitlock);
        g->m->waitunlockf = nil;
        g->m->waitlock = nil;
        if(!ok) {
            runtime·casgstatus(gp, Gwaiting, Grunnable);
            execute(gp);  // Schedule it back, never returns.
        }
    }

    schedule();
}
```

runtime·park函数包含3个参数，第一个是解锁函数指针，第二个是一个Lock指针，最后是一个字符串用以描述阻塞的原因。 很明显，前两个参数是配对的结构——由于任务阻塞前可能获得了某些Lock，这些Lock必须在任务状态保存完成后才能释放，以避免数据竞争。我们知道channel必须通过Lock确保互斥访问，一个阻塞的任务G#1需要将自己放到channel的等待队列中，如果在完成上下文保存前就释放了Lock，则可能导致G#2将未知状态的G#1置为Grunnable，因此释放Lock必须在runtime·park内完成。 由于阻塞时任务持有的Lock类型不尽相同——如Select操作的锁实际上是一组Lock的集合——因此需要特别指出Unlock的具体方式。 最后一

个参数主要是在gdb调试的时候方便发现任务阻塞的原因。顺便说一下，当所有的任务都处于Gwaiting状态时，也就表示当前程序进入了死锁态，不可能继续执行了，那么runtime会检测到这种情况，并输出所有Gwaiting任务的backtrace信息。

Gdead

最后，当一个任务执行结束后，会调用runtime·goexit结束自己的生命——将状态置为Gdead，并将结构体链到一个属于当前P的空闲G链表中，以备后续使用。

Go语言的并发模型基本上遵照了CSP模型，goroutine间完全靠channel通信，没有像Unix进程的wait或waitpid的等待机制，也没有类似“POSIX Thread”中的pthread_join的汇合机制，更没有像kill或signal这类的中断机制。每个goroutine结束后就自行退出销毁，不留一丝痕迹。

1.2 协程切换

概述

协程是Golang中的轻量级线程，麻雀虽小五脏俱全，Golang管理协程时也必然会涉及到协程之间的切换：阻塞的协程被切换出去，可运行的协程被切换进来。我们在本章节就来仔细分析下协程如何切换。

TLS

thread local storage:

getg()

goget()用来获取当前线程正在执行的协程g。该协程g被存储在TLS中。

mcall()

mcall在golang需要进行协程切换时被调用，用来保存被切换出去协程的信息，并在当前线程的g0协程堆栈上执行新的函数。一般情况下，会在新函数中执行一次schedule()来挑选新的协程来运行。接下来我们就看看mcall的实现。

调用时机

系统调用返回

当执行系统调用的线程从系统调用中返回后，有可能需要执行一次新的schedule，此时可能会调用mcall来完成该工作，如下：

```
func exitsyscall(dummy int32) {
    ......
    // Call the scheduler.
    mcall(exitsyscall0)
    ......
}
```

在exitsyscall0中如果可能会放弃当前协程并执行一次schedule，挑选新的协程来占有m。

由于阻塞放弃执行

由于某些原因，当前执行的协程可能会被阻塞，如管道读写时条件无法满足，则当前协程会被阻塞直到条件满足。

在gopark()函数中，便会调用该mcall放弃当前协程并执行一次协程调度。

```
func gopark(unlockf func(*g, unsafe.Pointer) bool, lock unsafe.P
ointer, reason string, traceEv byte, traceskip int) {
    mp := acquirem()
    gp := mp.curg
    status := readgstatus(gp)
    if status != _Grunning && status != _Gscanrunning {
        throw("gopark: bad g status")
    }
    mp.waitlock = lock
    mp.waitunlockf = *(*unsafe.Pointer)(unsafe.Pointer(&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)
}
```

而park_m函数我们在后面会分析，它放弃之前执行的协程并调用一次schedule()挑选新的协程来执行。

执行原理

前面我们主要描述了mcall被调用的时机，现在我们要来看看mcall的实现原理。

mcall的函数原型是：

```
func mcall(fn func(*g))
```

这里fn的参数指的是在调用mcall之前正在运行的协程。

我们前面说到，mcall的主要作用是协程切换，它将当前正在执行的协程状态保存起来，然后在m->g0的堆栈上调用新的函数。 在新的函数内会将之前运行的协程放弃，然后调用一次schedule()来挑选新的协程运行。

```
// func mcall(fn func(*g))
// Switch to m->g0's stack, call fn(g).
// Fn must never return.  It should gogo(&g->sched)
// to keep running g.
TEXT runtime·mcall(SB), NOSPLIT, $0-8

    // DI中存储参数fn
    MOVQ    fn+0(FP), DI

    get_tls(CX)
    // 获取当前正在运行的协程g信息
    // 将其状态保存在g.sched变量
    MOVQ    g(CX), AX   // save state in g->sched
    MOVQ    0(SP), BX   // caller's PC
    MOVQ    BX, (g_sched+gobuf_pc)(AX)
    LEAQ    fn+0(FP), BX    // caller's SP
    MOVQ    BX, (g_sched+gobuf_sp)(AX)
    MOVQ    AX, (g_sched+gobuf_g)(AX)
    MOVQ    BP, (g_sched+gobuf_bp)(AX)

    // switch to m->g0 & its stack, call fn
    MOVQ    g(CX), BX
    MOVQ    g_m(BX), BX
    MOVQ    m_g0(BX), SI
    CMPQ    SI, AX  // if g == m->g0 call badmcall
```

```
    JNE 3(PC)
    MOVQ    $runtime·badmcall(SB), AX
    JMP AX
    MOVQ    SI, g(CX)   // g = m->g0

    // 切换到m->g0堆栈
    MOVQ    (g_sched+gobuf_sp)(SI), SP  // sp = m->g0->sched.sp
    // 参数AX为之前运行的协程g
    PUSHQ   AX
    MOVQ    DI, DX
    MOVQ    0(DI), DI
    // 在m->g0堆栈上执行函数fn
    CALL    DI
    POPQ    AX
    MOVQ    $runtime·badmcall2(SB), AX
    JMP AX
    RET
```

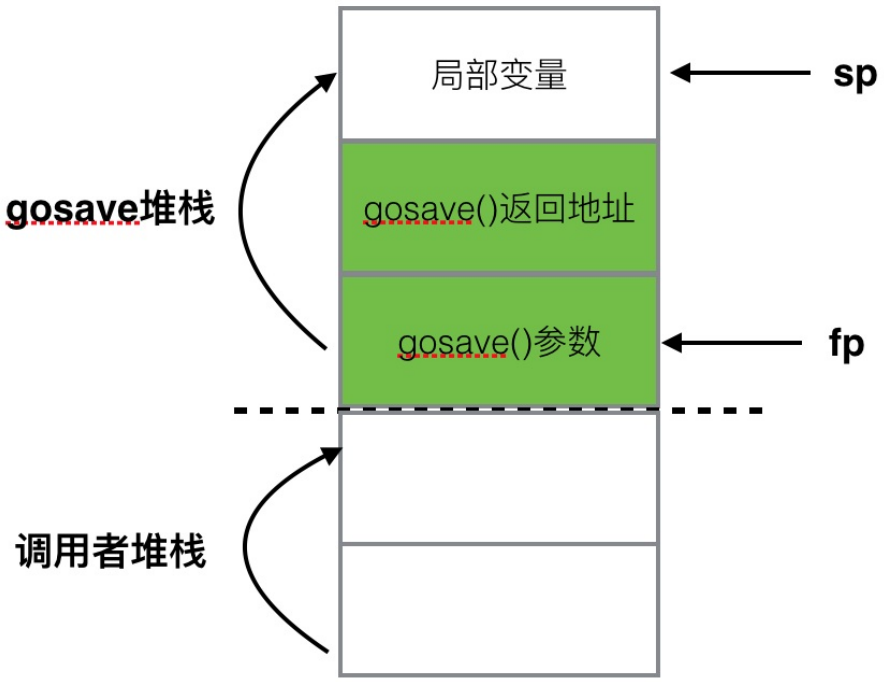
如何获取当前协程执行信息

前两句理解起来可能比较晦涩：

- buf+0(FP) 其实就是获取gosave的第一个参数(gobuf地址)，参考 ***A Quick Guide to Go's Assembler***

  > The FP pseudo-register is a virtual frame pointer used to refer to function arguments. The compilers maintain a virtual frame pointer and refer to the arguments on the stack as offsets from that pseudo-register. ***Thus 0(FP) is the first argument to the function, 8(FP) is the second (on a 64-bit machine), and so on***. However, when referring to a function argument this way, it is necessary to place a name at the beginning, as in first_arg+0(FP) and second_arg+8(FP).

- LEAQ buf+0(FP), BX则是获取到第一个参数的存储地址，而根据golang的堆栈布局，这个地址其实是调用者的sp，如下：



接下来的几句比较容易理解，在第一句获取了gobuf的地址后，接下来将一些相关成员设置成合适的value。 最关键的是以下几句

```
get_tls(CX)
MOVQ    g(CX), BX
MOVQ    BX, gobuf_g(AX)
```

这几句的作用是从TLS中获取当前线程运行的g，然后将其存储在gobuf的成员g。

gosave()

gosave在golang协程切换时被调用，用来保存被切换出去协程的信息，以便在下次该协程被重新调度执行时可以快速恢复出协程的执行上下文。

与协程调度相关的数据结构如下：

```
type g struct {
        stack       stack
        stackguard0 uintptr
        stackguard1 uintptr
        ......
        sched       gobuf
        ......
}
// gobuf记录与协程切换相关信息
type gobuf struct {
        sp   uintptr
        pc   uintptr
        g    guintptr
        ctxt unsafe.Pointer
        ret  uintreg
        lr   uintptr
        bp   uintptr
}
```

gosave是用汇编语言写的，性能比较高，但理解起来就没那么容易。

TODO: gosave()的调用路径是什么呢？

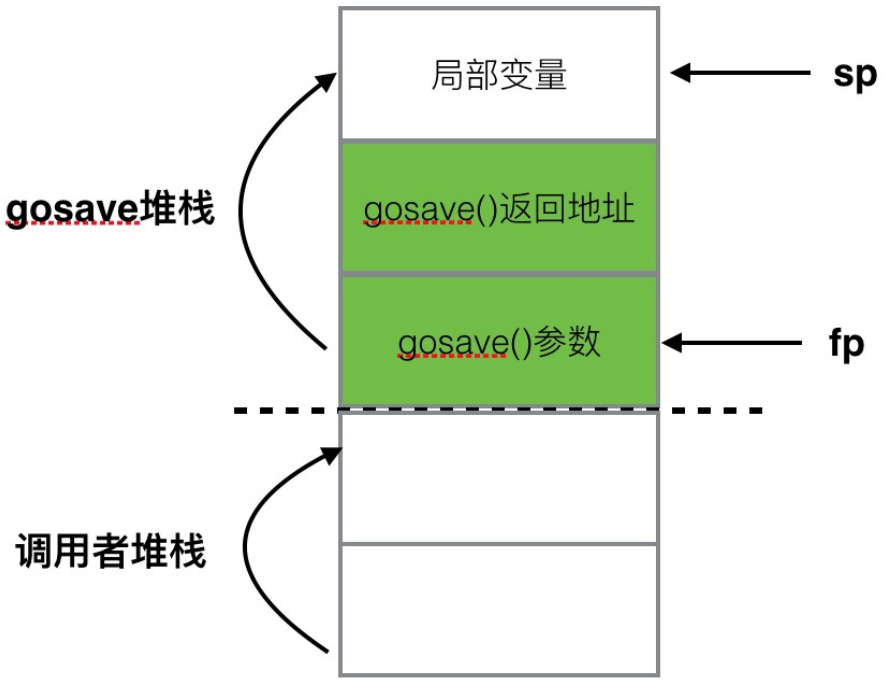
```
// void gosave(Gobuf*)
// save state in Gobuf; setjmp
TEXT runtime·gosave(SB), NOSPLIT, $0-8
        MOVQ    buf+0(FP), AX           // gobuf
        LEAQ    buf+0(FP), BX           // caller's SP
        MOVQ    BX, gobuf_sp(AX)
        MOVQ    0(SP), BX               // caller's PC
        MOVQ    BX, gobuf_pc(AX)
        MOVQ    $0, gobuf_ret(AX)
        MOVQ    $0, gobuf_ctxt(AX)
        MOVQ    BP, gobuf_bp(AX)
        get_tls(CX)
        MOVQ    g(CX), BX
        MOVQ    BX, gobuf_g(AX)
        RET
```

前两句理解起来可能比较晦涩：

- buf+0(FP) 其实就是获取gosave的第一个参数(gobuf地址)，参考 ***A Quick Guide to Go's Assembler***

  > The FP pseudo-register is a virtual frame pointer used to refer to function arguments. The compilers maintain a virtual frame pointer and refer to the arguments on the stack as offsets from that pseudo-register. ***Thus 0(FP) is the first argument to the function, 8(FP) is the second (on a 64-bit machine), and so on***. However, when referring to a function argument this way, it is necessary to place a name at the beginning, as in first_arg+0(FP) and second_arg+8(FP).

- LEAQ buf+0(FP), BX则是获取到第一个参数的存储地址，而根据golang的堆栈布局，这个地址其实是调用者的sp，如下：



接下来的几句比较容易理解，在第一句获取了gobuf的地址后，接下来将一些相关成员设置成合适的value。 最关键的是以下几句

```
get_tls(CX)
MOVQ    g(CX), BX
MOVQ    BX, gobuf_g(AX)
```

这几句的作用是从TLS中获取当前线程运行的g，然后将其存储在gobuf的成员g。

gogo()

gogo的作用正好相反，用来从gobuf中恢复出协程执行状态并跳转到上一次指令处继续执行。因此，其代码也相对比较容易理解，我们就不过多赘述，如下：

gogo()主要的调用路径：schedule()-->execute()-->googo()

```
// void gogo(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
        get_tls(CX)
        MOVQ    DX, g(CX)
        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)
        MOVQ    $0, gobuf_ret(BX)
        MOVQ    $0, gobuf_ctxt(BX)
        MOVQ    $0, gobuf_bp(BX)
        // 恢复出上一次执行指令，并跳转至该指令处
        MOVQ    gobuf_pc(BX), BX
        JMP     BX
```

这里最后一句跳转至该协程被调度出的那条语句继续执行，需要注意的是该函数不再返回调用者。

系统调用

前言

在讲述系统调用发生的协程调度之前，让我们看看go是如何进入系统调用的，理解了这个让我们不会对后面所说的一些东西感到很陌生。

golang对操作系统的系统调用作了封装，提供了syscall这样的库让我们执行系统调用。例如，Read系统调用实现如下：

```
func Read(fd int, p []byte) (n int, err error) {
    n, err = read(fd, p)
    if raceenabled {
        if n > 0 {
            ......
        }
        ......
    }
    return
}

// 最终封装了Syscall
func read(fd int, p []byte) (n int, err error) {
    var _p0 unsafe.Pointer
    if len(p) > 0 {
        _p0 = unsafe.Pointer(&p[0])
    } else {
        _p0 = unsafe.Pointer(&_zero)
    }
    r0, _, e1 := Syscall(SYS_READ, uintptr(fd), uintptr(_p0), ui
ntptr(len(p)))
    n = int(r0)
    if e1 != 0 {
        err = e1
    }
    return
}
```

```
// 我们只关心进入系统调用时调用的runtime·entersyscall
// 和退出时调用的runtime·exitsyscall
TEXT    ·Syscall(SB),NOSPLIT,$0-56
    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
```

我们并不关心系统调用到底怎么实现。我们只关心系统调用过程与调度器相关内容，因为Golang自己接管系统调用，调度器便可以在进出系统调用时做一些你所不明白的优化，这里我要带你弄清楚调度器怎么做优化的。

进入系统调用前

我们前面说过，系统调用是一个相对耗时的过程。一旦P中的某个G进入系统调用状态而阻塞了该P内的其他协程。此时调度器必须得做点什么吧，这就是调度器在进入系统调用前call runtime·entersyscall目的所在。

```
void
·entersyscall(int32 dummy)
{
    runtime·reentersyscall((uintptr)runtime·getcallerpc(&dummy),
 runtime·getcallersp(&dummy));
}
```

```
void
runtime·reentersyscall(uintptr pc, uintptr sp)
{
    void (*fn)(void);

    // 为什么g->m->locks++?
    g->m->locks++;

    g->stackguard0 = StackPreempt;
    g->throwsplit = 1;

    // Leave SP around for GC and traceback.
    // save()到底在save什么？
    save(pc, sp);
    g->syscallsp = sp;
    g->syscallpc = pc;
    runtime·casgstatus(g, Grunning, Gsyscall);
    // 这些堆栈之间到底是什么关系？
    if(g->syscallsp < g->stack.lo || g->stack.hi < g->syscallsp)

    {
        fn = entersyscall_bad;
        runtime·onM(&fn);
    }

    // 这个还不知道是啥意思
    if(runtime·atomicload(&runtime·sched.sysmonwait)) {
        fn = entersyscall_sysmon;
        runtime·onM(&fn);
        save(pc, sp);
    }
    // 这里很关键：P的M已经陷入系统调用，于是P忍痛放弃该M
    // 但是请注意：此时M还指向P，在M从系统调用返回后还能找到P
    g->m->mcache = nil;
    g->m->p->m = nil;
    // P的状态变为Psyscall
    runtime·atomicstore(&g->m->p->status, Psyscall);
    if(runtime·sched.gcwaiting) {
        fn = entersyscall_gcwait;
```

```
        runtime·onM(&fn);
        save(pc, sp);
    }
    g->stackguard0 = StackPreempt;
    g->m->locks--;
}
```

上面与调度器相关的内容其实就是将M从P剥离出去，告诉调度器，我已经放弃M了，我不能饿着我的孩子们（G）。但是M内心还是记着P的，在系统调用返回后，M还尽量找回原来的P，至于P是不是另结新欢就得看情况了。

注意这时候P放弃了前妻M，但是还没有给孩子们找后妈（M），只是将P的状态标记为PSyscall，那么什么时候以及怎么样给孩子们找后妈呢？我们在后面详细阐述。

从系统调用返回后

从系统调用返回后，也要告诉调度器，因为需要调度器做一些事情，根据前面系统调用的实现，具体实现是：

```
void
·exitsyscall(int32 dummy)
{
    void (*fn)(G*);

    // 这个g到底是什么？
    g->m->locks++;  // see comment in entersyscall

    if(runtime·getcallersp(&dummy) > g->syscallsp)
        runtime·throw("exitsyscall: syscall frame is no longer v
alid");

    g->waitsince = 0;
    // 判断能否快速找到归属
    if(exitsyscallfast()) {
        g->m->p->syscalltick++;
        // g的状态从syscall变成running，继续欢快地跑着
        runtime·casgstatus(g, Gsyscall, Grunning);
```

```
        g->syscallsp = (uintptr)nil;
        g->m->locks--;
        if(g->preempt) {
            g->stackguard0 = StackPreempt;
        } else {
            g->stackguard0 = g->stack.lo + StackGuard;
        }
        g->throwsplit = 0;
        return;
    }
    g->m->locks--;

    // Call the scheduler.
    // 如果M回来发现P已经有别人服务了，那只能将自己挂起
    // 等着服务别人。
    fn = exitsyscall0;
    runtime·mcall(&fn);
    ......
}

static bool
exitsyscallfast(void)
{
    void (*fn)(void);
    if(runtime·sched.stopwait) {
        g->m->p = nil;
        return false;
    }

    // 如果之前附属的P尚未被其他M,尝试绑定该P
    if(g->m->p && g->m->p->status == Psyscall && runtime·cas(&g-
>m->p->status, Psyscall, Prunning)) {
        g->m->mcache = g->m->p->mcache;
        g->m->p->m = g->m;
        return true;
    }
    // Try to get any other idle P.
    // 否则从空闲P列表中随便捞一个出来
    g->m->p = nil;
    if(runtime·sched.pidle) {
```

```
        fn = exitsyscallfast_pidle;
        runtime·onM(&fn);
        if(g->m->scalararg[0]) {
            g->m->scalararg[0] = 0;
            return true;
        }
    }
    return false;
}
```

G从系统调用返回的过程，其实就是失足妇女找男人的逻辑：

- 首先看看能否回到当初爱人(P)的怀抱：找到当初被我抛弃的男人，我这里还存着它的名片(m->p)，家庭住址什么的我都还知道；
- 如果爱人受不了寂寞和抚养孩子的压力已经变节（P的状态不再是Psyscall），那我就随便找个单身待解救男人从了也行；
- 如果上面的1、2都找不到，那也没办法，男人都死绝了，老娘只好另想他法。

以上过程1&2其实就是exitsyscallfast()的主要流程，用怀孕了的失足妇女找男人再合适不过。 一个女人由于年轻不懂事失足，抛家弃子（家是P，子是P的G）。当浪子回头后，意欲寻回从前的夫君，只能有两种可能：

- 等了很久已然心灰意冷的夫君在家人的安排下另娶他人；
- 痴情的夫君已然和嗷嗷待哺的孩子们依然在等待她的归回。

当然第二种的结局比较圆满，这个女人从此死心塌地守着这个家，于是p->m又回来了，孩子们(g)又可以继续活下去了。 第一种就比较难办了，女人（m）心灰意冷，将产下的儿子（陷入系统调用的g）交于他人（全局g的运行队列）抚养，远走他乡，从此接收命运的安排（参与调度，以后可能服务于别的p）。 对于第二种可能性，只能说女人的命运比较悲惨了：

```
static void
exitsyscall0(G *gp)
{
    P *p;
    runtime·casgstatus(gp, Gsyscall, Grunnable);
    dropg();
    runtime·lock(&runtime·sched.lock);
    // 这里M再次尝试为自己找个归宿P
    p = pidleget();
    // 如果没找到P，M讲自己放入全局的运行队列中
    // 同时将它的g放置到全局的P queue中进去，自己不管了
    if(p == nil)
        globrunqput(gp);
    else if(runtime·atomicload(&runtime·sched.sysmonwait)) {
        runtime·atomicstore(&runtime·sched.sysmonwait, 0);
        runtime·notewakeup(&runtime·sched.sysmonnote);
    }
    runtime·unlock(&runtime·sched.lock);
    // 如果找到了P，占有P并且开始执行P内的g，永不回头
    if(p) {
        acquirep(p);
        execute(gp);  // Never returns.
    }
    if(g->m->lockedg) {
        // Wait until another thread schedules gp and so m again
.
        stoplockedm();
        execute(gp);  // Never returns.
    }
    // 找了一圈还是没找到，释放掉M当前执行环境，M不再做事
    // stopm会暂停当前M直到其找到了可运行的P为止
    // 找到以后进入schedule，执行P内的g
    stopm();
    // m从stopm()中返回以后，说明该m被绑定至某个P,可以开始
    // 继续欢快地跑了,此时就需要调度找到一个g去执行
    // 这就是调用schedule的目的所在
    schedule();  // Never returns.
}
```

话说到这里，其实这个M当前没有运行的价值了（无法找到p运行它），那么我们就将她挂起，直到被其他人唤醒。 m被挂起调用的函数是stopm()

```
// Stops execution of the current m until new work is available.
// Returns with acquired P.
static void stopm(void)
{
    if(g->m->locks)
        runtime·throw("stopm holding locks");
    if(g->m->p)
        runtime·throw("stopm holding p");
    if(g->m->spinning) {
        g->m->spinning = false;
        runtime·xadd(&runtime·sched.nmspinning, -1);
    }
retry:
    runtime·lock(&runtime·sched.lock);
    // 将m插入到空闲m队列中，统一管理
    mput(g->m);
    runtime·unlock(&runtime·sched.lock);
    // 在这里被挂起，阻塞在m->park上，位于lock_futex.go
    runtime·notesleep(&g->m->park);
    // 从挂起被唤醒后开始执行
    runtime·noteclear(&g->m->park);
    if(g->m->helpgc) {
        runtime·gchelper();
        g->m->helpgc = 0;
        g->m->mcache = nil;
        goto retry;
    }
    // m->nextp是什么？
    acquirep(g->m->nextp);
    g->m->nextp = nil;
}
```

那么说到这里，其实很多事情都一目了然，当一个M从系统调用返回后，通过各种方式想找到可以托付的P(找前夫—>找闲汉)，求之不得最终只能将自己挂起，等待下次系统中有空闲的P的时候被唤醒。

陷入系统调用的P的管理

前面我们重点讲了一个m是如何陷入系统调用和如何返回的心酸之路。我们忽略了p的感情，因为他才是真正的受害者，它被剥夺了m，从此无人理会它嗷嗷待哺的孩子们(g)，并且状态还被变成了Psyscall，相当于贴上了屌丝标签，别无他法，只能等待陷入系统调用的m返回，再续前缘。 当然，这样做是不合理的，因为如果m进入系统调用后乐不思蜀，那P的孩子们都得饿死，这在现实社会中可以发生，但在数字世界里是决不允许的。 OK，组织绝对不会忽略这种情况的，于是，保姆（管家）出现了，它就是sysmon线程，这是一个特殊的m，专门监控系统状态。
sysmon周期性醒来，并且遍历所有的p，如果发现有Psyscall状态的p并且已经处于该状态超过一定时间了，那就不管那个负心的前妻，再次p安排一个m，这样p内的任务又可以得到处理了。

```
func sysmon() {
    ......
    retake(now);
    ......
}
```

我们只摘取了sysmon中与P处理相关的代码分析：

```
static uint32
retake(int64 now)
{
    uint32 i, s, n;
    int64 t;
    P *p;
    Pdesc *pd;

    n = 0;
    // 遍历所有的P，根据其状态作相应处理，我们只关注Psyscall
    for(i = 0; i < runtime·gomaxprocs; i++) {
        p = runtime·allp[i];
        if(p==nil)
            continue;
        pd = &pdesc[i];
        s = p->status;
        if(s == Psyscall) {
            t = p->syscalltick;
            if(pd->syscalltick != t) {
                pd->syscalltick = t;
                pd->syscallwhen = now;
                continue;
            }

            if(p->runqhead == p->runqtail &&
             runtime·atomicload(&runtime·sched.nmspinning) + run
time·atomicload(&runtime·sched.npidle) > 0 &&
                pd->syscallwhen + 10*1000*1000 > now)
                continue;
            incidlelocked(-1);
            // 因为需要将P重新安排m，所以状态转化为Pidle
            if(runtime·cas(&p->status, s, Pidle)) {
                n++;
                handoffp(p);
            }
            incidlelocked(1);
    ......
}
```

找到了处于Psyscall状态的P后，继续判断它等待的时间是否已经太长，如果是这样，就准备抛弃原来的还陷入syscall的m，调用handoff(p)，开始为p准备新生活。

我们接下来仔细分析下p是怎么过上新生活的，handoffp无非就是找一个新的m，将m与该p绑定，接下来将由m继续执行该p内的g。

handoffp()找到的新的m可能是别人以前的m(私生活好混乱)。由于这里获得的m是处于idle状态，处于wait状态（在stopm()中被sleep的），在这里会通过startm()来唤醒它。被唤醒的m继续执行它被阻塞的下一条语句：

```
stopm()
{
    ......
    // 从挂起被唤醒后开始执行
    runtime·noteclear(&g->m->park);
    if(g->m->helpgc) {
        runtime·gchelper();
        g->m->helpgc = 0;
        g->m->mcache = nil;
        goto retry;
    }
    // 将M和P绑定
    acquirep(g->m->nextp);
    g->m->nextp = nil;
}
```

由于m在sleep前的调用路径是exitsyscall0() --> stopm()，从stopm()中返回至exitsyscall0后，执行接下来的语句

```
unc exitsyscall0(gp *g) {
        _g_ := getg()

        ......
        stopm()
        // m继续run起来后，执行一次schedule
        // 找到m->p里面可运行的g并执行
        schedule() // Never returns.
}

// One round of scheduler: find a runnable goroutine and execute
 it.
// Never returns.
func schedule() {
        _g_ := getg()
        ......
        if gp == nil {
                gp, inheritTime = runqget(_g_.m.p.ptr())
                if gp != nil && _g_.m.spinning {
                        throw("schedule: spinning with local wor
k")
                }
        }
        if gp == nil {
                gp, inheritTime = findrunnable() // blocks until
 work is available
                resetspinning()
        }

        if gp.lockedm != nil {
                // Hands off own p to the locked m,
                // then blocks waiting for a new p.
                startlockedm(gp)
                goto top
        }
        // 执行该gp
        execute(gp, inheritTime)
}
```

管道读写

Golang中channle是协程间信息交互的主要手段。Golang的channel分为有缓冲和无缓冲两种，关于他们之间的用法区别可以自行google，这里不再赘述。Golang中的channel读写均是同步语义，写满的、读空的channel都会触发协程调度。

向**channel**写数据

无论是无缓冲还是有缓冲channel，当向channel写数据发现channel已满时，都需要将当前写的协程挂起，并进行一次调度，当前M转而执行P内的其他协程。直到有人再读该channel时发现有阻塞等待写的协程时将其唤醒。

```
func chansend(t *chantype, c *hchan, ep unsafe.Pointer, block bo
ol, callerpc uintptr) bool {
    ......
    ......
    // 对应缓冲区大小为0的channel
    if c.dataqsiz == 0 {
        // 如果有接收者在等着
        // 这时候可以写入成功，将数据拷贝至
        // 接收者的缓冲区，唤醒接收者即可
        sg := c.recvq.dequeue()
        if sg != nil {
            ......
        }
        // 如果尚未有接收者，需要将其block
        // no receiver available: block on this channel.
        gp := getg()
        mysg := acquireSudog()
        mysg.releasetime = 0
        if t0 != 0 {
            mysg.releasetime = -1
        }
        mysg.elem = ep
        mysg.waitlink = nil
        gp.waiting = mysg
        mysg.g = gp
        mysg.selectdone = nil
        gp.param = nil
        c.sendq.enqueue(mysg)
        // 在这里将发送者协程block
        goparkunlock(&c.lock, "chan send")
    }
    // 对于有缓冲区的channel，流程相似
    ......
}
```

如果尚未有接收者，那就将发送者阻塞，调用了函数goparkunlock

```
func goparkunlock(lock *mutex, reason string) {
    gopark(unsafe.Pointer(&parkunlock_c), unsafe.Pointer(lock),
```

```
reason)
}

func gopark(unlockf unsafe.Pointer, lock unsafe.Pointer, reason
string) {
    mp := acquirem()
    gp := mp.curg
    status := readgstatus(gp)
    if status != _Grunning && status != _Gscanrunning {
        gothrow("gopark: bad g status")
    }
    mp.waitlock = lock
    mp.waitunlockf = unlockf
    gp.waitreason = reason
    releasem(mp)
    // 最终调用函数park_m
    mcall(park_m)
}

void runtime·park_m(G *gp)
{
    bool ok;

    // 首先将g的status变为Gwaiting
    runtime·casgstatus(gp, Grunning, Gwaiting);
    dropg();

    if(g->m->waitunlockf) {
        ok = g->m->waitunlockf(gp, g->m->waitlock);
        g->m->waitunlockf = nil;
        g->m->waitlock = nil;
        if(!ok) {
            runtime·casgstatus(gp, Gwaiting, Grunnable);
            execute(gp);  // Schedule it back, never returns.
        }
    }
    // 发起一次调度，该m不再执行当前的g，而是选择一个新的g重新执行
    // schedule的具体实现不在这里描述
    schedule();
}
```

从channel读数据

从channel读数据和向channel写数据的实现几乎一样，只是方向反了而已，在这里不再赘述，各位自己钻研代码吧。

抢占式调度

虽然我们一直强调golang调度器是非抢占式。非抢占式的一个最大坏处是无法保证公平性，如果一个g处于死循环状态，那么其他协程可能就会被饿死。 所幸的是，Golang在1.4版本中加入了抢占式调度的逻辑，抢占式调度必然可能在g执行的某个时刻被剥夺cpu，让给其他协程。

还记得我们之前说过Golang的sysmon协程么，该协程会定期唤醒作系统状态检查，我们前面说过了它如何检查处于Psyscall状态的p，以便让处于系统调用状态的P可以被继续执行，不至于饿死。 除了检查这个意外，sysmon还检查处于Prunning状态的P，检查它的目的就是避免这里的某个g占用了过多的cpu时间，并在某个时刻剥夺其cpu运行时间。

```
static uint32
retake(int64 now)
{
    uint32 i, s, n;
    int64 t;
    P *p;
    Pdesc *pd;

    n = 0;
    for(i = 0; i < runtime·gomaxprocs; i++) {
        p = runtime·allp[i];
        if(p==nil)
            continue;
        pd = &pdesc[i];
        s = p->status;
        if(s == Psyscall) {
            ......
        } else if(s == Prunning) {
            // Preempt G if it's running for more than 10ms.
            t = p->schedtick;
            if(pd->schedtick != t) {
                pd->schedtick = t;
                pd->schedwhen = now;
                continue;
```

```
            }
            if(pd->schedwhen + 10*1000*1000 > now)
                continue;
            // 如果自从上次发生调度时间已经超过了10ms
            preemptone(p);
        }
    }
    return n;
}

// 这里的抢占只是将g的preempt设置为true
// 只有在g进行函数调用时才会检查该标志位
// 并进而可能发生调度，非常弱
static bool
preemptone(P *p)
{
    M *mp;
    G *gp;

    mp = p->m;
    if(mp == nil || mp == g->m)
        return false;
    gp = mp->curg;
    if(gp == nil || gp == mp->g0)
        return false;
    gp->preempt = true;
    // Every call in a go routine checks for stack overflow by
    // comparing the current stack pointer to gp->stackguard0.
    // Setting gp->stackguard0 to StackPreempt folds
    // preemption into the normal stack overflow check.
    gp->stackguard0 = StackPreempt;
    return true;
}
```

之前我们说过在函数调用时会进行堆栈检测，现在将gp->stackGuard0设置为StackPreempt(-1314，非常小的值),肯定会调用一次runtime.morestack,逻辑如下：

```
TEXT runtime·morestack(SB),NOSPLIT,$0-0
    // Cannot grow scheduler stack (m->g0).
```

```
    get_tls(CX)
    MOVQ    g(CX), BX
    MOVQ    g_m(BX), BX
    MOVQ    m_g0(BX), SI
    CMPQ    g(CX), SI
    JNE 2(PC)
    INT $3

    // Cannot grow signal stack (m->gsignal).
    MOVQ    m_gsignal(BX), SI
    CMPQ    g(CX), SI
    JNE 2(PC)
    INT $3

    // Called from f.
    // Set m->morebuf to f's caller.
    MOVQ    8(SP), AX   // f's caller's PC
    MOVQ    AX, (m_morebuf+gobuf_pc)(BX)
    LEAQ    16(SP), AX  // f's caller's SP
    MOVQ    AX, (m_morebuf+gobuf_sp)(BX)
    get_tls(CX)
    MOVQ    g(CX), SI
    MOVQ    SI, (m_morebuf+gobuf_g)(BX)

    // Set g->sched to context in f.
    MOVQ    0(SP), AX // f's PC
    MOVQ    AX, (g_sched+gobuf_pc)(SI)
    MOVQ    SI, (g_sched+gobuf_g)(SI)
    LEAQ    8(SP), AX // f's SP
    MOVQ    AX, (g_sched+gobuf_sp)(SI)
    MOVQ    DX, (g_sched+gobuf_ctxt)(SI)
    MOVQ    BP, (g_sched+gobuf_bp)(SI)

    // Call newstack on m->g0's stack.
    MOVQ    m_g0(BX), BX
    MOVQ    BX, g(CX)
    MOVQ    (g_sched+gobuf_sp)(BX), SP
    CALL    runtime·newstack(SB)
    MOVQ    $0, 0x1003  // crash if newstack returns
    RET
```

最终调用newstack来进行堆栈扩容：

```
func newstack() {
    thisg := getg()
    // TODO: double check all gp. shouldn't be getg().
    if thisg.m.morebuf.g.ptr().stackguard0 == stackFork {
        throw("stack growth after fork")
    }
    if thisg.m.morebuf.g.ptr() != thisg.m.curg {
        print("runtime: newstack called from g=", thisg.m.morebu
f.g, "\n"+"\tm=", thisg.m, " m->curg=", thisg.m.curg, " m->g0=",
 thisg.m.g0, " m->gsignal=", thisg.m.gsignal, "\n")
        morebuf := thisg.m.morebuf
        traceback(morebuf.pc, morebuf.sp, morebuf.lr, morebuf.g.
ptr())
        throw("runtime: wrong goroutine in newstack")
    }

    gp := thisg.m.curg
    morebuf := thisg.m.morebuf
    thisg.m.morebuf.pc = 0
    thisg.m.morebuf.lr = 0
    thisg.m.morebuf.sp = 0
    thisg.m.morebuf.g = 0
    rewindmorestack(&gp.sched)

    // NOTE: stackguard0 may change underfoot, if another thread
    // is about to try to preempt gp. Read it just once and use
that same
    // value now and below.
    preempt := atomicloaduintptr(&gp.stackguard0) == stackPreemp
t
    if preempt {
        if thisg.m.locks != 0 || thisg.m.mallocing != 0 || thisg
.m.preemptoff != "" || thisg.m.p.ptr().status != _Prunning {
            // Let the goroutine keep running for now.
            // gp->preempt is set, so it will be preempted next
time.
            gp.stackguard0 = gp.stack.lo + _StackGuard
```

```
            gogo(&gp.sched) // never return
        }
    }

    ......
    // 进行重新调度
    if preempt {
        if gp == thisg.m.g0 {
            throw("runtime: preempt g0")
        }
        if thisg.m.p == 0 && thisg.m.locks == 0 {
            throw("runtime: g is running but p is not")
        }
        if gp.preemptscan {
            for !castogscanstatus(gp, _Gwaiting, _Gscanwaiting)
{
                // Likely to be racing with the GC as
                // it sees a _Gwaiting and does the
                // stack scan. If so, gcworkdone will
                // be set and gcphasework will simply
                // return.
            }
            if !gp.gcscandone {
                scanstack(gp)
                gp.gcscandone = true
            }
            gp.preemptscan = false
            gp.preempt = false
            casfrom_Gscanstatus(gp, _Gscanwaiting, _Gwaiting)
            casgstatus(gp, _Gwaiting, _Grunning)
            gp.stackguard0 = gp.stack.lo + _StackGuard
            gogo(&gp.sched) // never return
        }

        // Act like goroutine called runtime.Gosched.
        casgstatus(gp, _Gwaiting, _Grunning)
        // 放弃当前协程，调度新协程执行
        gopreempt_m(gp) // never return
    }
}
```

这里需要注意两个东西：

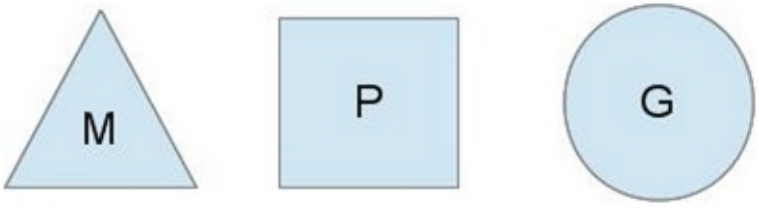
thisg := getg()：这个代表当前执行newstack()函数的堆栈，也是当前线程的g0的stack；

gp := thisg.m.curg：这个代表的是申请栈扩容的协程，与上面的thisg不是一个东西。

因为虽然调用了newstack，但是对于stackguard0==stackPreempt的协程来说，它的目的压根不是堆栈扩容，而是发起一次调度，所以直接进入了gopreempt_m，这里将当前协程挂起，并发起一次schedule().

数据结构

Golang中实现协程调度算法的主要有以下三个数据结构，正是这三个结构加上一些算法构成了Golang的协程调度算法，当然，这些数据结构也是在不断进化的，保不准未来又会加入其他结构来提升调度器性能。



其中：

- M: 代表操作系统线程，也就是我们经常理解的线程，是真正参与OS调度的单元，每个goroutine只有依附于某个M方可真正地执行；
- G: 代表协程，也就是我们经常使用的Goroutine；
- P: 协程调度器的基本调度单元，G的容身之所，连接G和M的桥梁。

理解调度器的关键是理解P的作用：它是实现M:N调度的关键结构。在Golang1.1版本中被引入，解决了1.0中的全局调度队列带来的可扩展性问题。

三个数据结构之间的关系如下：

1). 每个G都必须依附于某个P；
2). 每个M想运行的时候，必须先获取一个P，再执行P中的G；
3). 每个P维护可运行G队列，避免了全局G队列带来的锁竞争问题，提高了调度器的可扩展性；
4). 如果一个M在执行G的时候进入了syscall，那么该M就被占据了，与其关联的P此时就游离出来，可被同样处于idle状态的M获取到，进而P内的G得到继续执行的机会；
5). go中可通过GOMAXPROCS()来设置P的数量，一般建议设置成系统核数即可，P的数量也代表了当前活跃的M数(因进入syscall而block的M不计算在内)。

```
struct G {
    ......
    m *m
}

struct  M
{
    ......
    G*  curg;   // current running goroutine
    P*  p;      // attached P for executing Go code (nil if not
executing Go code)
    ......
}

struct P
{
    ......
    M*  m;  // back-link to associated M (nil if idle)
    ......
    // Queue of runnable goroutines.
    uint32  runqhead;
    uint32  runqtail;
    G*  runq[256];

    // Available G's (status == Gdead)
    G*  gfree;
    ......
};
```

阅读上面的数据结构可发现他们三者之间的关系是这样的： g：有一个指针指向执行它的m，也即g隶属于m； m：保存了当前正在执行的g，同时还指向p，也即m隶属于p； p: 保存了当前正执行p内g的m，同时还有属于它的g的链表

数据结构

编程接口

```
func Listen(net, laddr string) (Listener, error)
func (*TCPListener) Accept (c Conn, err error)
func (c *conn) Read(b []byte) (int, error)
func (c *conn) Write(b []byte) (int, error)
```

内部数据结构

Listener 和 TCPListener

简单来说，一个是接口，一个是具体实现。因为golang支持tcp、udp等各种协议，天然使用golang的interface。

Listener定义了针对socket操作的各种接口：

```
type Listener interface {
    Accept() (c Conn, err error)
    Close() error
    Addr() Addr
}
```

TCPListener则定义了tcp协议需要使用的数据结构

```
type TCPListener struct {
    fd *netFD
}
```

netFD是golang网络库最核心数据结构，贯穿了golang网络库的所有API，它对底层的socket进行封装，屏蔽了不同os的网络实现。我们接下来会单独分析该数据结构。

TCPListener中的fd是与监听套接字关联的socket。

Conn和TCPConn

Conn与TCPConn的关系类似Listener与TCPListener。也是抽象与具象的关系。

```
type TCPConn struct {
    conn
}
type conn struct {
    fd *netFD
}
```

netFD

这个结构太关键了，说它是golang网络库最核心数据结构一点也不为过。所有golang网络接口最终都会转化为对该结构的方法。

```
// Network file descriptor.
type netFD struct {
    // 不太明白这个mutex的作用
    fdmu fdMutex
    // 该socket相关的fd
    sysfd       int
    family      int
    sotype      int
    isConnected bool
    net         string
    laddr       Addr
    raddr       Addr
    // wait server
    // 与epoll相关结构
    pd pollDesc
}
```

而这个结构中最重要的又是最后一个成员pd，golang网络库实现高并发全靠它了。

```
type pollDesc struct {
    runtimeCtx uintptr
}
```

大致梳理了golang网络库中的几个基本数据结构，接下来我们就可以深入到内部实现流程了。

内部实现

这一章节我们将详细描述网络关键API的实现，主要包括Listen、Accept、Read、Write等。另外，为了突出关键流程，我们选择忽略所有的错误。这样可以使得代码看起来更为简单。而且我们只关注tcp协议实现，udp和unix socket不是我们关心的。

Listen()

```
func Listen(net, laddr string) (Listener, error) {
    la, err := resolveAddr("listen", net, laddr, noDeadline)
    ......
    switch la := la.toAddr().(type) {
    case *TCPAddr:
        l, err = ListenTCP(net, la)
    case *UnixAddr:
        ......
    }
   ......
}

// 对于tcp协议，返回的的是TCPListener
func ListenTCP(net string, laddr *TCPAddr) (*TCPListener, error)
 {
    ......
    fd, err := internetSocket(net, laddr, nil, noDeadline, sysca
ll.SOCK_STREAM, 0, "listen")
    ......
    return &TCPListener{fd}, nil
}

func internetSocket(net string, laddr, raddr sockaddr, deadline
time.Time, sotype, proto int, mode string) (fd *netFD, err error
) {
    ......
    return socket(net, family, sotype, proto, ipv6only, laddr, r
```

```
addr, deadline)
}

func socket(net string, family, sotype, proto int, ipv6only bool
, laddr, raddr sockaddr, deadline time.Time) (fd *netFD, err err
or) {
    // 创建底层socket，设置属性为O_NONBLOCK
    s, err := sysSocket(family, sotype, proto)
    ......
    setDefaultSockopts(s, family, sotype, ipv6only)
    // 创建新netFD结构
    fd, err = newFD(s, family, sotype, net)
    ......
    if laddr != nil && raddr == nil {
        switch sotype {
        case syscall.SOCK_STREAM, syscall.SOCK_SEQPACKET:
            // 调用底层listen监听创建的套接字
            fd.listenStream(laddr, listenerBacklog)
            return fd, nil
        case syscall.SOCK_DGRAM:
            ......
        }
    }
}

// 最终调用该函数来创建一个socket
// 并且将socket属性设置为O_NONBLOCK
func sysSocket(family, sotype, proto int) (int, error) {
    syscall.ForkLock.RLock()
    s, err := syscall.Socket(family, sotype, proto)
    if err == nil {
        syscall.CloseOnExec(s)
    }
    syscall.ForkLock.RUnlock()
    if err != nil {
        return -1, err
    }
    if err = syscall.SetNonblock(s, true); err != nil {
        syscall.Close(s)
        return -1, err
```

```
    }
    return s, nil
}

func (fd *netFD) listenStream(laddr sockaddr, backlog int) error
 {
    if err := setDefaultListenerSockopts(fd.sysfd)
    if lsa, err := laddr.sockaddr(fd.family); err != nil {
        return err
    } else if lsa != nil {
        // Bind绑定至该socket
        if err := syscall.Bind(fd.sysfd, lsa); err != nil {
            return os.NewSyscallError("bind", err)
        }
    }
    // 监听该socket
    if err := syscall.Listen(fd.sysfd, backlog);
    // 这里非常关键：初始化socket与异步IO相关的内容
    if err := fd.init(); err != nil {
        return err
    }
    lsa, _ := syscall.Getsockname(fd.sysfd)
    fd.setAddr(fd.addrFunc()(lsa), nil)
    return nil
}
```

我们这里看到了如何实现Listen。流程基本都很简单，但是因为我们使用了异步编程，因此，我们在Listen完该s ocket后，还必须将其添加到监听队列中，以后该socket有事件到来时能够及时通知到。

对linux有所了解的应该都知道epoll，没错golang使用的就是epoll机制来实现socket事件通知。那我们看对一个 监听socket，是如何将其添加到epoll的监听队列中呢？

```
func (fd *netFD) init() error {
    if err := fd.pd.Init(fd); err != nil {
        return err
    }
    return nil
}

func (pd *pollDesc) Init(fd *netFD) error {
    // 利用了Once机制，保证一个进程只会执行一次
    // runtime_pollServerInit:
    // TEXT net·runtime_pollServerInit(SB),NOSPLIT,$0-0
    // JMP runtime·netpollServerInit(SB)
    serverInit.Do(runtime_pollServerInit)
    // runtime_pollOpen:
    // TEXT net·runtime_pollOpen(SB),NOSPLIT,$0-0
    // JMP runtime·netpollOpen(SB)
    ctx, errno := runtime_pollOpen(uintptr(fd.sysfd))
    if errno != 0 {
        return syscall.Errno(errno)
    }
    pd.runtimeCtx = ctx
    return nil
}
```

这里就是socket异步编程的关键：

- netpollServerInit()初始化异步编程结构，对于epoll，该函数是netpollinit，且使用Once机制保证一个进程 只会初始化一次；

```
func netpollinit() {
  epfd = epollcreate1(_EPOLL_CLOEXEC)
  if epfd >= 0 {
      return
  }
  epfd = epollcreate(1024)
  if epfd >= 0 {
      closeonexec(epfd)
      return
  }
  ......
}
```

- netpollOpen则在socket被创建出来后将其添加到epoll队列中，对于epoll，该函数被实例化为netpollopen。

```
func netpollopen(fd uintptr, pd *pollDesc) int32 {
    var ev epollevent
    ev.events = _EPOLLIN | _EPOLLOUT | _EPOLLRDHUP | _EPOLLET
    *(**pollDesc)(unsafe.Pointer(&ev.data)) = pd
    return -epollctl(epfd, _EPOLL_CTL_ADD, int32(fd), &ev)
}
```

OK，看到这里，我们也就明白了，监听一个套接字的时候无非就是传统的socket异步编程，然后将该socket添加到 epoll的事件监听队列中。

Accept()

既然我们描述的重点的tcp协议，因此，我们看看TCPListener的Accept方法是怎么实现的：

```
func (l *TCPListener) Accept() (Conn, error) {
    c, err := l.AcceptTCP()
    ......
}

func (l *TCPListener) AcceptTCP() (*TCPConn, error) {
```

```
    ......
    fd, err := l.fd.accept()
    ......
    // 返回给调用者一个新的TCPConn
    return newTCPConn(fd), nil
}

func (fd *netFD) accept() (netfd *netFD, err error) {
    // 为什么对该函数加读锁？
    if err := fd.readLock(); err != nil {
        return nil, err
    }
    defer fd.readUnlock()
    ......
    for {
        // 这个accept是golang包装的系统调用
        // 用来处理跨平台
        s, rsa, err = accept(fd.sysfd)
        if err != nil {
            if err == syscall.EAGAIN {
                // 如果没有可用连接，WaitRead()阻塞该协程
                // 后面会详细分析WaitRead.
                if err = fd.pd.WaitRead(); err == nil {
                    continue
                }
            } else if err == syscall.ECONNABORTED {
                // 如果连接在Listen queue时就已经被对端关闭
                continue
            }
        }
        break
    }

    netfd, err = newFD(s, fd.family, fd.sotype, fd.net)
    ......
    // 这个前面已经分析，将该fd添加到epoll队列中
    err = netfd.init()
    ......
    lsa, _ := syscall.Getsockname(netfd.sysfd)
    netfd.setAddr(netfd.addrFunc()(lsa), netfd.addrFunc()(rsa))
```

```
    return netfd, nil
}
```

OK，从前面的编程事例中我们知道，一般在主协程中会accept新的connection，使用异步编程我们知道，如果没有 新连接到来，该协程会一直被阻塞，直到新连接到来有人唤醒了该协程。

一般在主协程中调用accept，如果返回值为EAGAIN，则调用WaitRead来阻塞当前协程，后续在该socket有事件到来 时被唤醒，WaitRead以及唤醒过程我们会在后面仔细分析。

Read

```
func (c *conn) Read(b []byte) (int, error) {
    if !c.ok() {
        return 0, syscall.EINVAL
    }
    return c.fd.Read(b)
}

func (fd *netFD) Read(p []byte) (n int, err error) {
    // 为什么对函数调用加读锁
    if err := fd.readLock(); err != nil {
        return 0, err
    }
    defer fd.readUnlock()
    // 这个又是干嘛？
    if err := fd.pd.PrepareRead(); err != nil {
        return 0, &OpError{"read", fd.net, fd.raddr, err}
    }
    for {
        n, err = syscall.Read(int(fd.sysfd), p)
        if err != nil {
            n = 0
            // 如果返回EAGIN，阻塞当前协程直到有数据可读被唤醒
            if err == syscall.EAGAIN {
                if err = fd.pd.WaitRead(); err == nil {
                    continue
```

```
                }
            }
        }
        // 检查错误，封装io.EOF
        err = chkReadErr(n, err, fd)
        break
    }
    if err != nil && err != io.EOF {
        err = &OpError{"read", fd.net, fd.raddr, err}
    }
    return
}

func chkReadErr(n int, err error, fd *netFD) error {
    if n == 0 && err == nil && fd.sotype != syscall.SOCK_DGRAM &
& fd.sotype != syscall.SOCK_RAW {
        return io.EOF
    }
    return err
}
```

Read的流程与Accept流程极其一致，阅读起来也很简单。相信不用作过多解释，自己看吧。 需要注意的是每次Read不能保证可以读到想读的那么多内容，比如缓冲区大小是10，而实际可能只读到5，应用程 序需要能够处理这种情况。

Write

```
func (fd *netFD) Write(p []byte) (nn int, err error) {
    // 为什么这里加写锁
    if err := fd.writeLock(); err != nil {
        return 0, err
    }
    defer fd.writeUnlock()
    // 这个是干什么？
    if err := fd.pd.PrepareWrite(); err != nil {
        return 0, &OpError{"write", fd.net, fd.raddr, err}
    }
    // nn记录总共写入的数据量，每次Write可能只能写入部分数据
```

```
    for {
        var n int
        n, err = syscall.Write(int(fd.sysfd), p[nn:])
        if n > 0 {
            nn += n
        }
        // 如果数组数据已经全部写完，函数返回
        if nn == len(p) {
            break
        }
        // 如果写入数据时被block了，阻塞当前协程
        if err == syscall.EAGAIN {
            if err = fd.pd.WaitWrite(); err == nil {
                continue
            }
        }
        if err != nil {
            n = 0
            break
        }
        // 如果返回值为0，代表了什么？
        if n == 0 {
            err = io.ErrUnexpectedEOF
            break
        }
    }
    if err != nil {
        err = &OpError{"write", fd.net, fd.raddr, err}
    }
    return nn, err
}
```

注意Write语义与Read不一样的地方：

> Write尽量将用户缓冲区的内容全部写入至底层socket，如果遇到socket暂时不可写入，会阻塞当前协程； Read在某次读取成功时立即返回，可能会导致读取的数据量少于用户缓冲区的大小； 为什么会在实现上有此不同，我想可能read的优先级比较高吧，应用程序可能一直在等着，我们不能等到数据一直 读完才返回，会阻塞用户。 而写不一样，优先级相对较低，而且用户一般也不着急写立即返回，所以可以将所有的数据全部写入，而且这样 也能简化应用程序的写法。

总结

上面我们基本说完了golang网络编程内的关键API流程，我们遗留了一个关键内容：当系统调用返回EAGAIN时，会 调用WaitRead/WaitWrite来阻塞当前协程，我会在接下来的章节中继续分析。

内部实现(二)

前面的章节我们基本聊完了golang网络编程的关键API流程，但遗留了一个关键内容：当系统调用返回EAGAIN时，会调用WaitRead/WaitWrite来阻塞当前协程，现在我们接着聊。

WaitRead/WaitWrite

```
func (pd *pollDesc) Wait(mode int) error {
    res := runtime_pollWait(pd.runtimeCtx, mode)
    return convertErr(res)
}

func (pd *pollDesc) WaitRead() error {
    return pd.Wait('r')
}

func (pd *pollDesc) WaitWrite() error {
    return pd.Wait('w')
}
```

最终runtime_pollWait走到下面去了：

```
TEXT net·runtime_pollWait(SB),NOSPLIT,$0-0
    JMP runtime·netpollWait(SB)
```

我们仔细考虑应该明白：netpollWait的主要作用是：等待关心的socket是否有事件（其实后面我们知道只是等待一个标记位是否发生改变），如果没有事件，那么就将当前的协程挂起，直到有通知事件发生，我们接下来看看到底如何实现：

```
func netpollWait(pd *pollDesc, mode int) int {
    // 先检查该socket是否有error发生（如关闭、超时等）
    err := netpollcheckerr(pd, int32(mode))
    if err != 0 {
        return err
```

```
    }
    // As for now only Solaris uses level-triggered IO.
    if GOOS == "solaris" {
        onM(func() {
            netpollarm(pd, mode)
        })
    }
    // 循环等待netpollblock返回值为true
    // 如果返回值为false且该socket未出现任何错误
    // 那该协程可能被意外唤醒，需要重新被挂起
    // 还有一种可能：该socket由于超时而被唤醒
    // 此时netpollcheckerr就是用来检测超时错误的
    for !netpollblock(pd, int32(mode), false) {
        err = netpollcheckerr(pd, int32(mode))
        if err != 0 {
            return err
        }
    }
    return 0
}

func netpollblock(pd *pollDesc, mode int32, waitio bool) bool {
    gpp := &pd.rg
    if mode == 'w' {
        gpp = &pd.wg
    }

    // set the gpp semaphore to WAIT
    // 首先将轮询状态设置为pdWait
    // 为什么要使用for呢？因为casuintptr使用了自旋锁
    // 为什么使用自旋锁就要加for循环呢？
    for {
        old := *gpp
        if old == pdReady {
            *gpp = 0
            return true
        }
        if old != 0 {
            gothrow("netpollblock: double wait")
        }
```

```
        // 将socket轮询相关的状态设置为pdWait
        if casuintptr(gpp, 0, pdWait) {
            break
        }
    }
    // 如果未出错将该协程挂起，解锁函数是netpollblockcommit
    if waitio || netpollcheckerr(pd, mode) == 0 {
        f := netpollblockcommit
        gopark(**(**unsafe.Pointer)(unsafe.Pointer(&f)), unsafe.
Pointer(gpp), "IO wait")
    }
    // 可能是被挂起的协程被唤醒
    // 或者由于某些原因该协程压根未被挂起
    // 获取其当前状态记录在old中
    old := xchguintptr(gpp, 0)
    if old > pdWait {
        gothrow("netpollblock: corrupted state")
    }
    return old == pdReady
}
```

从上面的分析我们看到，如果无法读写，golang会将当前协程挂起，在协程被唤醒的时候，该标记位应该会被置位。 我们接下来看看这些挂起的协程何时会被唤醒。

事件通知

golang运行库在系统运行过程中存在socket事件检查点，目前，该检查点主要位于以下几个地方：

- runtime·startTheWorldWithSema(void)：在完成gc后；
- findrunnable()：这个暂时不知道何时会触发？
- sysmon：golang中的监控协程，会周期性检查就绪socket

TODO: 为什么是在这些地方检查socket就绪事件呢？

接下来我们看看如何检查socket就绪事件，在socket就绪后又是如何唤醒被挂起的协程？主要调用函数runtime-netpoll()

我们只关注epoll的实现，对于epoll，上面的方法具体实现是netpoll_epoll.go中的netpoll

```
func netpoll(block bool) (gp *g) {
    if epfd == -1 {
        return
    }
    waitms := int32(-1)
    if !block {
        // 如果调用者不希望block
        // 设置waitsm为0
        waitms = 0
    }
    var events [128]epollevent
retry:
    // 调用epoll_wait获取就绪事件
    n := epollwait(epfd, &events[0], int32(len(events)), waitms)
    if n < 0 {
        ...
        }
        goto retry
    }
    for i := int32(0); i < n; i++ {
        ev := &events[i]
        if ev.events == 0 {
            continue
        }
        var mode int32
        if ev.events&(_EPOLLIN|_EPOLLRDHUP|_EPOLLHUP|_EPOLLERR)
!= 0 {
            mode += 'r'
        }
        if ev.events&(_EPOLLOUT|_EPOLLHUP|_EPOLLERR) != 0 {
            mode += 'w'
        }
        // 对每个事件，调用了netpollready
        // pd主要记录了与该socket关联的等待协程
        if mode != 0 {
            pd := *(**pollDesc)(unsafe.Pointer(&ev.data))
            netpollready((**g)(noescape(unsafe.Pointer(&gp))), p
```

```
d, mode)
        }
    }
    // 如果调用者同步等待且本次未获取到就绪socket
    // 继续重试
    if block && gp == nil {
        goto retry
    }
    return gp
}
```

这个函数主要调用epoll_wait（当然，golang封装了系统调用）来获取就绪socket fd，对每个就绪的fd，调用netpollready()作进一步处理。这个函数的最终返回值就是一个已经就绪的协程(g)链表。

netpollready主要是将该socket fd标记为IOReady，并唤醒等待在该fd上的协程g，将其添加到传入的g链表中。

```
// make pd ready, newly runnable goroutines (if any) are returne
d in rg/wg
func netpollready(gpp **g, pd *pollDesc, mode int32) {
    var rg, wg *g
    if mode == 'r' || mode == 'r'+'w' {
        rg = netpollunblock(pd, 'r', true)
    }
    if mode == 'w' || mode == 'r'+'w' {
        wg = netpollunblock(pd, 'w', true)
    }
    // 将就绪协程添加至链表中
    if rg != nil {
        rg.schedlink = *gpp
        *gpp = rg
    }
    if wg != nil {
        wg.schedlink = *gpp
        *gpp = wg
    }
}
// 将pollDesc的状态置为pdReady并返回就绪协程
```

```
func netpollunblock(pd *pollDesc, mode int32, ioready bool) *g {
    gpp := &pd.rg
    if mode == 'w' {
        gpp = &pd.wg
    }

    for {
        old := *gpp
        if old == pdReady {
            return nil
        }
        if old == 0 && !ioready {
            return nil
        }
        var new uintptr
        if ioready {
            new = pdReady
        }
        if casuintptr(gpp, old, new) {
            if old == pdReady || old == pdWait {
                old = 0
            }
            return (*g)(unsafe.Pointer(old))
        }
    }
}
```

疑问：一个fd会被多个协程同时进行IO么？比如一个协程读，另外一个协程写？或者多个协程同时读？此时返回的是哪个协程就绪呢？

一个socket fd可支持并发读写，因为对于tcp协议来说，是全双工。读写操作的是不同缓冲区，但是不支持并发读和并发写，因为这样会错乱的。所以上面的netFD.RWLock()就是干这个作用的。

内部实现(三)

上面的章节中我们大概提到了在哪些情景下会检查哪些套接字上有可读写事件发生。接下来我们就仔细描述下这些场景并看看有事件发生后到底golang内部如何处理这些事件。

sysmon

golang的后台监控协程，会定期运行，检测系统状态。其中一个重要的任务就是检测是否有就绪的socket事件。如下：

```
func sysmon() {
    ......
    // poll network if not polled for more than 10ms
    lastpoll := int64(atomicload64(&sched.lastpoll))
    now := nanotime()
    unixnow := unixnanotime()
    if lastpoll != 0 && lastpoll+10*1000*1000 < now {
        cas64(&sched.lastpoll, uint64(lastpoll), uint64(now))
        // non-blocking - returns list of goroutines
        gp := netpoll(false)
        if gp != nil {
            // Need to decrement number of idle locked M's
            // (pretending that one more is running) before inje
ctglist.
            // 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)
        }
    }
    ......
}
```

在sysmon中调用netpoll()函数，获得当前就绪的socket事件，返回等待在这些事件上的协程g链表。 注意，这里调用netpoll()时传入的参数为false，表示告诉netpoll()内部无需wait。如果没有就绪的socket事件就立即返回。

findrunnable

该函数在调用schedule()时触发，用于找到当前可运行的协程。这里也有可能会触发等待网络事件。

```
// Finds a runnable goroutine to execute.
// Tries to steal from other P's, get g from global queue, poll
network.
func findrunnable() (gp *g, inheritTime bool) {
    _g_ := getg()
    ......

    // Poll network.
    // This netpoll is only an optimization before we resort to
stealing.
    // We can safely skip it if there a thread blocked in netpol
l already.
    // If there is any kind of logical race with that blocked th
read
    // (e.g. it has already returned from netpoll, but does not
set lastpoll yet),
    // this thread will do blocking netpoll below anyway.
    if netpollinited() && sched.lastpoll != 0 {
        if gp := netpoll(false); gp != nil { // non-blocking
        // netpoll returns list of goroutines linked by schedlin
k.
        injectglist(gp.schedlink.ptr())
        casgstatus(gp, _Gwaiting, _Grunnable)
        if trace.enabled {
            traceGoUnpark(gp, 0)
        }
        return gp, false
    }
    ...
}
```

如果线程m.p没有可运行g且全局g队列也没有可运行g，此时就需要看看有没有网络事件就绪，如果有，那最好，可以用当前等待在这些事件上的g来运行。

这里注意：调用netpoll()时传入的参数也是false，调用者并不想阻塞。因为它在后面还可以通过其他手段来拿到可运行的g。

startTheWorld

在做完gc后，golang也会查看下是否有网络事件在等待。但这里想不出有什么特别的需求要这么做。

对唤醒网络事件协程的处理

上面我们仔细描述了golang中何时会去检测网络事件。接下来我们要描述的是检测到就绪网络事件后，并获得了等待在这些网络事件上的协程后，该如何处理这些协程。

前面章节我们仔细描述了一个协程如何阻塞在网络IO上。等到这些网络IO就绪后，在上面描述的三种场景中可以获得那些阻塞的协程。接下来调用injectglist()处理这些协程。

```
// Injects the list of runnable G's into the scheduler.
// Can run concurrently with GC.
func injectglist(glist *g) {
    if glist == nil {
        return
    }
    if trace.enabled {
        for gp := glist; gp != nil; gp = gp.schedlink.ptr() {
            traceGoUnpark(gp, 0)
        }
    }
    lock(&sched.lock)
    var n int
    for n = 0; glist != nil; n++ {
        gp := glist
        glist = gp.schedlink.ptr()
        casgstatus(gp, _Gwaiting, _Grunnable)
        globrunqput(gp)
    }
    unlock(&sched.lock)
    for ; n != 0 && sched.npidle != 0; n-- {
        startm(nil, false)
    }
}
```

顾名思义，这个函数是要将获得的协程列表插入到某个地方，看代码知道时插入到了全局的可运行g队列。为什么是插入到这里呢？ 因为这些被阻塞的协程在阻塞的时候都调用了dropg()来放弃自己的m。这样他们也就没有了所属的p。

因此，我们只能将这些g添加到全局的可运行g队列中。这样，在下次调度时，这些g就有可能得到执行的机会。

Socket超时

说明

我们前面仔细梳理了Golang中的网络实现，还有一个很重要的点没有说到的是超时，因为超时在网络编程中非常重要。我们今天来研究下golang超时的实现机制。

API

一个Connection可以通过下面的接口来实现该连接的读写超时：

```
SetDeadline(t time.Time) error
SetReadDeadline(t time.Time) error
SetWriteDeadline(t time.Time) error
```

实现

我们以SetReadDeadline()为例研究下超时实现

```
func (c *conn) SetReadDeadline(t time.Time) error {
    ...
    if err := c.fd.setReadDeadline(t); err != nil {
        return &OpError{Op: "set", Net: c.fd.net, Source: nil, A
ddr: c.fd.laddr, Err: err}
    }
    return nil
}

func (fd *netFD) setReadDeadline(t time.Time) error {
    return setDeadlineImpl(fd, t, 'r')
}

func setDeadlineImpl(fd *netFD, t time.Time, mode int) error {
    d := runtimeNano() + int64(t.Sub(time.Now()))
```

```
    if t.IsZero() {
        d = 0
    }
    if err := fd.incref(); err != nil {
        return err
    }
    runtime_pollSetDeadline(fd.pd.runtimeCtx, d, mode)
    fd.decref()
    return nil
}
/go:linkname net_runtime_pollSetDeadline net.runtime_pollSetDead
line
func net_runtime_pollSetDeadline(pd *pollDesc, d int64, mode int
) {
    lock(&pd.lock)
    if pd.closing {
        unlock(&pd.lock)
        return
    }
    // 可以让老定时器无效
    pd.seq++
    // 定时器已经存在，删除已经存在的定时器.
    // 重设新的定时器
    if pd.rt.f != nil {
        deltimer(&pd.rt)
        pd.rt.f = nil
    }
    if pd.wt.f != nil {
        deltimer(&pd.wt)
        pd.wt.f = nil
    }
    // Setup new timers.
    if d != 0 && d <= nanotime() {
        d = -1
    }
    if mode == 'r' || mode == 'r'+'w' {
        pd.rd = d
    }
    if mode == 'w' || mode == 'r'+'w' {
        pd.wd = d
```

```
    }
    if pd.rd > 0 && pd.rd == pd.wd {
        pd.rt.f = netpollDeadline
        pd.rt.when = pd.rd
        pd.rt.arg = pd
        pd.rt.seq = pd.seq
        addtimer(&pd.rt)
    } else {
        if pd.rd > 0 {
            pd.rt.f = netpollReadDeadline
            pd.rt.when = pd.rd
            pd.rt.arg = pd
            pd.rt.seq = pd.seq
            addtimer(&pd.rt)
        }
        if pd.wd > 0 {
            pd.wt.f = netpollWriteDeadline
            pd.wt.when = pd.wd
            pd.wt.arg = pd
            pd.wt.seq = pd.seq
            addtimer(&pd.wt)
        }
    }
    ...
}
```

golang对socket的超时实现从上面来看，无非就是将socket绑定读/写定时器，超时时间由用户设置。关键是超时处理函数，我们看到设置的读超时函数为netpollReadDeadline。

```
func netpollReadDeadline(arg interface{}, seq uintptr) {
    netpolldeadlineimpl(arg.(*pollDesc), seq, true, false)
}

func netpolldeadlineimpl(pd *pollDesc, seq uintptr, read, write
bool) {
    lock(&pd.lock)
    // 判断是否是过期定时器
    if seq != pd.seq {
```

```
        unlock(&pd.lock)
        return
    }
    var rg *g
    if read {
        if pd.rd <= 0 || pd.rt.f == nil {
            throw("netpolldeadlineimpl: inconsistent read deadli
ne")
        }
        // pd.rd=-1表示读的deadline已经过了
        // 任何的reader在读之前应该判断deadline
        // 如果deadline过期了，应该返回timeout错误
        // 而且这里会唤醒已经阻塞在读的协程
        // 协程被唤醒后第一件事就是检查是否出现超时错误
        // (即pd.rd == -1)
        pd.rd = -1
        atomicstorep(unsafe.Pointer(&pd.rt.f), nil)
        rg = netpollunblock(pd, 'r', false)
    }
    var wg *g
    if write {
        if pd.wd <= 0 || pd.wt.f == nil && !read {
            throw("netpolldeadlineimpl: inconsistent write deadl
ine")
        }
        // pd.wd=-1表示写的deadline已经过了
        // 任何的writer在写之前应该判断deadline
        // 如果deadline过期了，应该返回timeout错误
        // 而且这里会唤醒已经阻塞在写的协程
        // 协程被唤醒后第一件事就是检查是否出现超时错误
        // (即pd.wd == -1)
        pd.wd = -1
        atomicstorep(unsafe.Pointer(&pd.wt.f), nil)
        wg = netpollunblock(pd, 'w', false)
    }
    unlock(&pd.lock)
    // 唤醒等待读的协程
    if rg != nil {
        goready(rg, 0)
    }
```

```
    // 唤醒等待写的协程
    if wg != nil {
        goready(wg, 0)
    }
}
```

上面的超时处理逻辑看起来也非常简单：

将与该timer相关的socket标记为deadline已经到来，对socket的任何一次读写之前都要判断是否超时，我们在后面看到；

调用netpollunblock，这个设置socket当前状态为error

调用goready()唤醒任何等待读/写的协程，这里不再赘述。

我们前面说过了超时是通过定时器实现的，在定时器超时函数被触发时将会设置一个读写deadline已经超过的标记位(pd.rd = -1/pd.wd = -1)。 在定时器超时到达时候，上层应用可能有三种情况：

- 应用程序调用的Read/Write在超时之后；
- 超时发生在应用程序调用的Read/Write阻塞后；
- 定时器发生在Read/Write成功后。

我们接下来按照这三种情况一一分析。

先超时，再**Read**

每次Read之前都检查是否已经读超时：

```
func (fd *netFD) Read(p []byte) (n int, err error) {
    if err := fd.readLock(); err != nil {
        return 0, err
    }
    defer fd.readUnlock()
    if err := fd.pd.PrepareRead(); err != nil
    {
        return 0, err
    }
    .....
}
```

```
func (pd *pollDesc) PrepareRead() error {
    return pd.Prepare('r')
}

func (pd *pollDesc) Prepare(mode int) error {
    res := runtime_pollReset(pd.runtimeCtx, mode)
    return convertErr(res)
}

func net_runtime_pollReset(pd *pollDesc, mode int) int {
    // reset 之前先检查是否已经超时错误了
    err := netpollcheckerr(pd, int32(mode))
    if err != 0 {
        return err
    }
    if mode == 'r' {
        pd.rg = 0
    } else if mode == 'w' {
        pd.wg = 0
    }
    return 0
}

// 这里检查了是否有错，对于已经超时的socket
// 返回错误码2
func netpollcheckerr(pd *pollDesc, mode int32) int {
    if pd.closing {
        return 1 // errClosing
    }
    if (mode == 'r' && pd.rd < 0) || (mode == 'w' && pd.wd < 0)
{
        return 2 // errTimeout
    }
    return 0
}
```

通过上面的分析，我们知道，在每次调用Read之前都会重设定时器（其实不算是重设，而是将定时器归零）。但是在归零之前先检查是否已经超时了，如果是，返回错误码2（timeout）。 因此，发生这种情况时我们是能够顺利检测出socket超时

的。

Read阻塞后再超时

若满足此情况，我们前面分析了，超时定时器触发的时候会唤醒Read阻塞的协程，我们看看该协程被唤醒的时候做了什么:

```
func net_runtime_pollWait(pd *pollDesc, mode int) int {
    err := netpollcheckerr(pd, int32(mode))
    if err != 0 {
        return err
    }
    // As for now only Solaris uses level-triggered IO.
    if GOOS == "solaris" {
        netpollarm(pd, mode)
    }
    // netpollblock阻塞协程
    for !netpollblock(pd, int32(mode), false) {
        // 阻塞协程被唤醒时先检查是否出错
        err = netpollcheckerr(pd, int32(mode))
        if err != 0 {
            println("go routine ", getg(), " is woken up and got
 error:", err)
            return err
        }
        // Can happen if timeout has fired and unblocked us,
        // but before we had a chance to run, timeout has been r
eset.
        // Pretend it has not happened and retry.
    }
    return 0
}
```

因为该阻塞协程可能被以下两种事件唤醒：

- 有数据可读
- 超时函数中将其唤醒

所以被唤醒后的第一件事就是检测是否由于超时唤醒，如果是，则返回错误码2。

可见，这种情况在golang中也能正确被处理。

Read成功后再超时

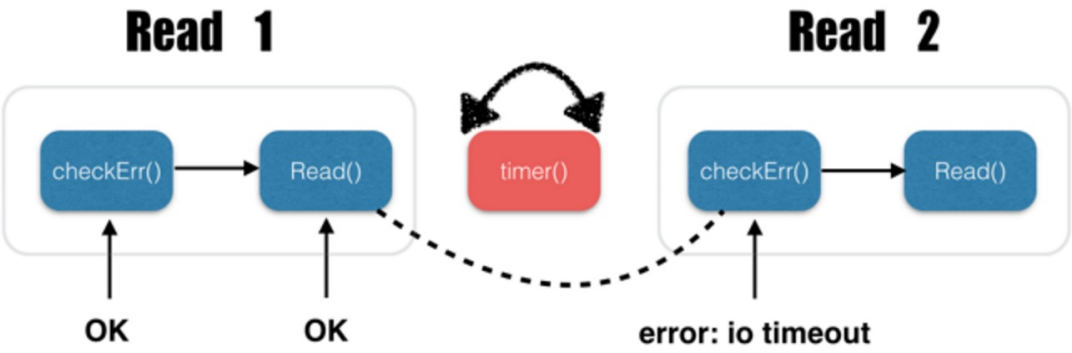
这种情况其实是最难分析的。我们前面说过，每次读之前都会调用PrepareRead()。

```
func (pd *pollDesc) PrepareRead() error {
    return pd.Prepare('r')
}

func (pd *pollDesc) Prepare(mode int) error {
    res := runtime_pollReset(pd.runtimeCtx, mode)
    return convertErr(res)
}

func net_runtime_pollReset(pd *pollDesc, mode int) int {
    err := netpollcheckerr(pd, int32(mode))
    if err != 0 {
        return err
    }
    // 如果未超时，那么将pd.rg设置为0
    // 这样下次定时器超时的时候就无法唤醒任何人
    if mode == 'r' {
        pd.rg = 0
    } else if mode == 'w' {
        pd.wg = 0
    }
    return 0
}
```

我们看到golang对这种情况的处理是：读之前将定时器的rg(记录了定时器该唤醒谁)清空。

如果socket超时发生在读成功后，因为在PrepareRead()中已经将pd.rg设置为0，因此无法唤醒任何人了。 但是下次再来读的时候先判断该连接上是否有错误发生，而发生超时的时候会将pd.rd = -1，因此下次再读的时候就会出错？这个好像不合理，

如下图：



Read 1 success, read some data
then, socket timer arrives, so socket got error
then checkErr() will get error before reading

如何解决上面这个问题呢？

目前分析代码发现，golang并没有解决这个问题，具体的测试请见我的另一篇关于golang超时的测试博客。

附录

说明

最近在钻研golang网络库的实现，顺便写了些测试代码，还真发现了一些有趣的东西。不知道该说是golang实现简洁呢还是该喷呢？

测试

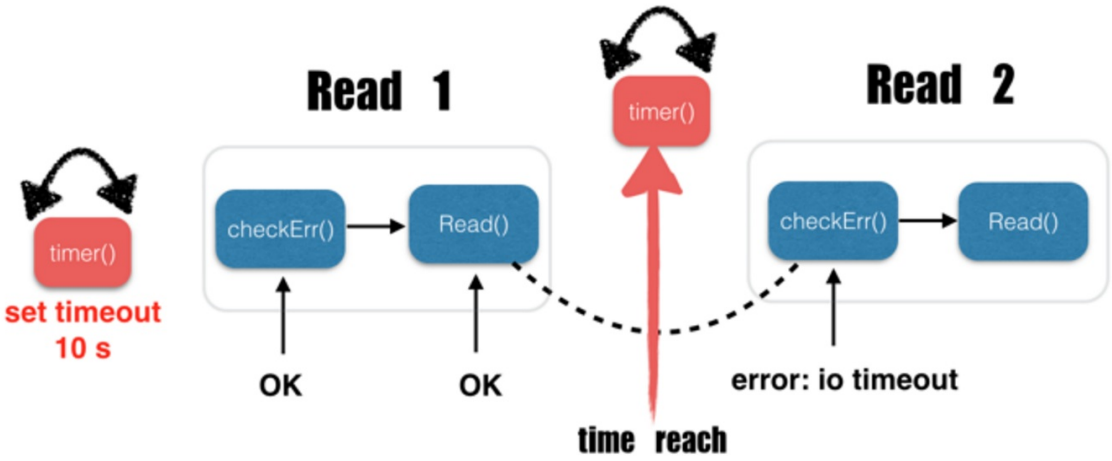
read成功然后超时测试

服务器程序和客户端程序互相配合，服务器端对新来的连接（创建了新协程来处理该连接）设置超时时间10s，在超时时间内，客户端发送了数据，然后客户端停止发送数据（但不断开连接），服务器端在某个时刻出现超时。

为了更直观地查看到服务器内部运行时信息，稍微hack了下go1.5的代码，加了一些打印语句，以下是服务器终端输出信息：

```
accept connection: &{conn:{fd:0xc8200580e0}}
sys_accept returns EAGAIN, sleep~

set deadline 1441023477949855606 for mode 233
// 服务器端新连接第一次收到数据
Now: 1441023470, hello
// 超时定时器到来，等待读的协程被唤醒，检查出现了
// error=2(timeout)错误并返回给应用程序，应用程序打印
// 错误输出为i/o timeout
go routine 0xc820001500 is woken up because of read timeout at t
ime 1441023477954550830
go routine 0xc820001500 is woken up because of read timeout
go routine 0xc820001500 is woken up and got error: 2
Now: 1441023477, error:read tcp 127.0.0.1:8080->127.0.0.1:58005:
 i/o timeout
```

这个过程用图表示如下：



上图中read1成功，在等待read2时之前设置的定时器在read1和read2之间被触发。结果会导致read2出现超时失败。

这与我之前期望的行为不太一致：本期望read1成功后与之相关的定时器会被删除，这样如果我们在read2之前不再次设置socket超时时间，read2就会一直等待。

read前已经超时测试

服务器程序和客户端程序互相配合，服务器端对新来的连接（创建了新协程来处理该连接）设置超时时间1s，在超时时间后，客户端发送了数据，服务器端在读之前就已经超时，而且接下来每次读都会超时，无法接收到客户端的数据。 为了更直观地查看到服务器内部运行时信息，稍微hack了下go1.5的代码，加了一些打印语句，以下是服务器终端输出信息：

```
accept connection: &{conn:{fd:0xc820078000}}
set deadline 1441062762123875542 for mode 233
go routine 0xc820064300 is woken up because of read timeout at t
ime 1441062762128411796
go routine 0xc820064300 is woken up and got error: 2
Now: 1441062762, error:read tcp 127.0.0.1:8080->127.0.0.1:58918:
 i/o timeout
Now: 1441062767, error:read tcp 127.0.0.1:8080->127.0.0.1:58918:
 i/o timeout
Now: 1441062772, error:read tcp 127.0.0.1:8080->127.0.0.1:58918:
 i/o timeout
```

可以看到，在客户端数据发送之前，服务器的连接就已经超时了，接下来的每次读都返回i/o timeout，即使客户端后来发送了数据，服务器端也无法响应。

总结

仔细思考了golang的网络超时实现，发现这好像与我们理解的socket超时不太一样。 golang实现的超时比较简单粗暴：在应用程序调用设置超时接口起开始启动一个定时器，一旦定时器到时，就将该socket设置为错误，接下来所有的读写都会失败。没有作更多精细化处理。

后记

看了个帖子，讨论了golang的网络超时的设计思想：不再设置socket层面的超时（因为该socket超时好像是给同步调用使用的）。而是设置一次请求的超时时间：所使用的SetReadDeadline()也就是这个意思。 引用一篇帖子内容：

> 具体的原因那个CL有说，简单的说，就是，SetTimeout(time.Second)的话，如果网络上的 数据一个字节一个字节的来的话，Read100字节的实际等待时间是100秒，这太容易导致误解。 为了让用户明确表明等待时间到底是指每次Read syscall的还是整个Read()调用的，所以改成了让用户提供一个绝对的时间。 如果任意一次Read syscall返回的时候超过了这个时间点，那么就算超时。 这样做的好处是，你可以在更高层次上设置Timeout，比如读取一个http请求的。deadline而不用管它的实现到底调用了多少次Read()。这样从上而下设置超时的方式都统一一点。

思考

使用操作系统的设置socket超时接口会有什么副作用？

- setsockopt()设置SO_RCVTIMEO是针对blocking socket的，而golang则对所有的socket都使用了non-blocking方式，可参考nginx实现，也是non-blocking，使用红黑树来管理所有的socket超时；
- 传统的超时涵义是设置操作系统的SO_RCVTIMEO选项，这样每次应用程序的一次Read假如触发了操作系统的多次read syscall，那这个超时就可能会被放大很多倍，可参考上面的引用；

附录二

说明

之前在阅读整理golang网络库的时候，遇到了最核心的数据结构pollDesc，一直对该结构中的某些成员的含义不是十分清楚

```
type pollDesc struct {
    link *pollDesc
    lock    mutex
    fd      uintptr
    closing bool
    seq     uintptr
    rg      uintptr
    rt      timer
    rd      int64
    wg      uintptr
    wt      timer
    wd      int64
    user    uint32
}
```

不清楚的主要部分集中在rg和wg这两个成员。这两天花了点时间好好地将它梳理了下，终于明白了实现原理，特记录下来，也让自己可以好好思考下。

引子

我们前面文章在讲述golang网络连接的读写的时候说到：当操作系统的read/write返回EAGAIN时就会阻塞当前协程，直到该socket有可读写事件发生时，操作系统通知golang的runtime，runtime进而唤醒阻塞在该socket的读写协程。而协程阻塞和唤醒逻辑中主要使用了上面描述的pollDesc结构。

阻塞

我们以读过程为例，描述阻塞过程。 在每次Read真正读之前都会调用PrepareRead()来初始化一些变量：

```
func (pd *pollDesc) Prepare(mode int) error {
    res := runtime_pollReset(pd.runtimeCtx, mode)
    return convertErr(res)
}

func (pd *pollDesc) PrepareRead() error {
    return pd.Prepare('r')
}

func net_runtime_pollReset(pd *pollDesc, mode int) int {
    err := netpollcheckerr(pd, int32(mode))
    if err != 0 {
        return err
    }
    if mode == 'r' {
        pd.rg = 0
    } else if mode == 'w' {
        pd.wg = 0
    }
    return 0
}
```

对于read来说，主要是将pd.rg设置为0。相当于初始状态。 接下来read如果返回了EAGAIN错误码，会进入协程阻塞函数：

```
func net_runtime_pollWait(pd *pollDesc, mode int) int {
    err := netpollcheckerr(pd, int32(mode))
    if err != 0 {
        return err
    }
    for !netpollblock(pd, int32(mode), false) {
        err = netpollcheckerr(pd, int32(mode))
        if err != 0 {
            return err
        }
    }
    return 0
}
```

这里主要调用netpollblock来阻塞当前协程g，但是协程g可能有多种原因被唤醒：

- 有读写事件到来；
- 出现超时错误
- 其他

判断是否是正常返回()的根据是netpollblock的返回值，如果返回值为true，表示是有读写事件发生；返回值为false代表可能出现超时或者其他错误

因此实现尚我们利用一个for循环来阻塞，一旦g从netpollblock()中返回且返回值为false，我们检查是否是出现了超时错误，如果是这样，就返回给应用程序该错误，如果是其他错误，我们继续进入netpollblock阻塞当前协程。 如果netpollblock返回值是true，代表了g被io事件唤醒，接下来调用者可以继续读了。

```
func netpollblock(pd *pollDesc, mode int32, waitio bool) bool {
    gpp := &pd.rg
    if mode == 'w' {
        gpp = &pd.wg
    }
    // 首先将pd.rg设为pdWait
    // 之所以需要使用for循环是因为
    // casuintptr可能会失败
    for {
        old := *gpp
        if old == pdReady {
```

```
            *gpp = 0
            return true
        }
        if old != 0 {
            throw("netpollblock: double wait")
        }
        // 如果成功，跳出循环
        if casuintptr(gpp, 0, pdWait) {
            break
        }
    }
    // gopark会阻塞当前协程g
    // gopark阻塞g之前，先调用了netpollblockcommit
    // 该函数将pd.rg从pdWait变成g的地址
    // casuintptr((*uintptr)(gpp), pdWait,
    // uintptr(unsafe.Pointer(gp)))
    if waitio || netpollcheckerr(pd, mode) == 0 {
        gopark(netpollblockcommit, unsafe.Pointer(gpp), "IO wait
", traceEvGoBlockNet, 5)
    }
    // 之所以判断old > pdWait
    // 是因为可能gopark压根就没有阻塞g成功
    // 此时会立即返回至现在逻辑
    // 被唤醒后查看pd.rg，此时应该是pdReady了
    // 如果是被IO事件唤醒的话
    old := xchguintptr(gpp, 0)
    if old > pdWait {
        throw("netpollblock: corrupted state")
    }
    // 如果阻塞的协程g返回且原因是pdReady，返回true
    return old == pdReady
}
```

这个函数比较关键：当前协程g在这里陷入阻塞，阻塞之前先将pd.rg状态设置为**pdWait**，表示等待IO输入，这很好理解。

接下来在调用gopark时传入了函数指针netpollblockcommit，这个函数会在真正block前被调用，而该函数又将**pd.rg**设置为**g**的地址（具体请参考该函数实现），这个应该也不难理解：如果pd不记录g的地址，以后该pd收到IO事件的时候就不知

道到底该唤醒谁了。

上面的实现一个困惑的地方在于：既然g在阻塞前会设置pd.rg为g的地址，那为什么在之前还要多此一举地设置pd.rg为pdWait呢？ 猜测：调用gopark是有条件的，参考if判断，如果条件不满足，此时pd.rg状态就是pdWait。那到底什么条件下无需阻塞当前协程呢？

好，至此，我们弄清楚了在协程g读的时候是如何被block的，接下来我们看看g是如何被唤醒的。

唤醒

被block的协程可能因为以下几种情况被唤醒：

- 超时
- IO事件到来

超时唤醒

```
func netpolldeadlineimpl(pd *pollDesc, seq uintptr, read, write
bool) {
    lock(&pd.lock)
    // 判断是否是过期定时器
    if seq != pd.seq {
        unlock(&pd.lock)
        return
    }
    var rg *g
    if read {
        if pd.rd <= 0 || pd.rt.f == nil {
            ......
        }
        pd.rd = -1
        atomicstorep(unsafe.Pointer(&pd.rt.f), nil)
        rg = netpollunblock(pd, 'r', false)
    }
    var wg *g
    if write {
       ......
    }
    unlock(&pd.lock)

    if rg != nil {
        goready(rg, 0)
    }
    ......
}
```

可以看到，在超时处理函数中确实调用了netpollunblock()来唤醒等待协程g，且其第三个参数设置为false，表示并非由于IO事件唤醒。

另外，注意，这里唤醒的过程需要在pd.lock保护下执行，因为可能会有其他协程并发唤醒情况发生。

IO事件唤醒

```
func netpollready(gpp *guintptr, pd *pollDesc, mode int32) {
    var rg, wg guintptr
    if mode == 'r' || mode == 'r'+'w' {
        rg.set(netpollunblock(pd, 'r', true))
    }
    if mode == 'w' || mode == 'r'+'w' {
        wg.set(netpollunblock(pd, 'w', true))
    }
    if rg != 0 {
        rg.ptr().schedlink = *gpp
        *gpp = rg
    }
    if wg != 0 {
        wg.ptr().schedlink = *gpp
        *gpp = wg
    }
}
```

这里同样调用netpollunblock唤醒等待协程g，只是这次调用netpollunblock的参数三变成了true。

唤醒过程

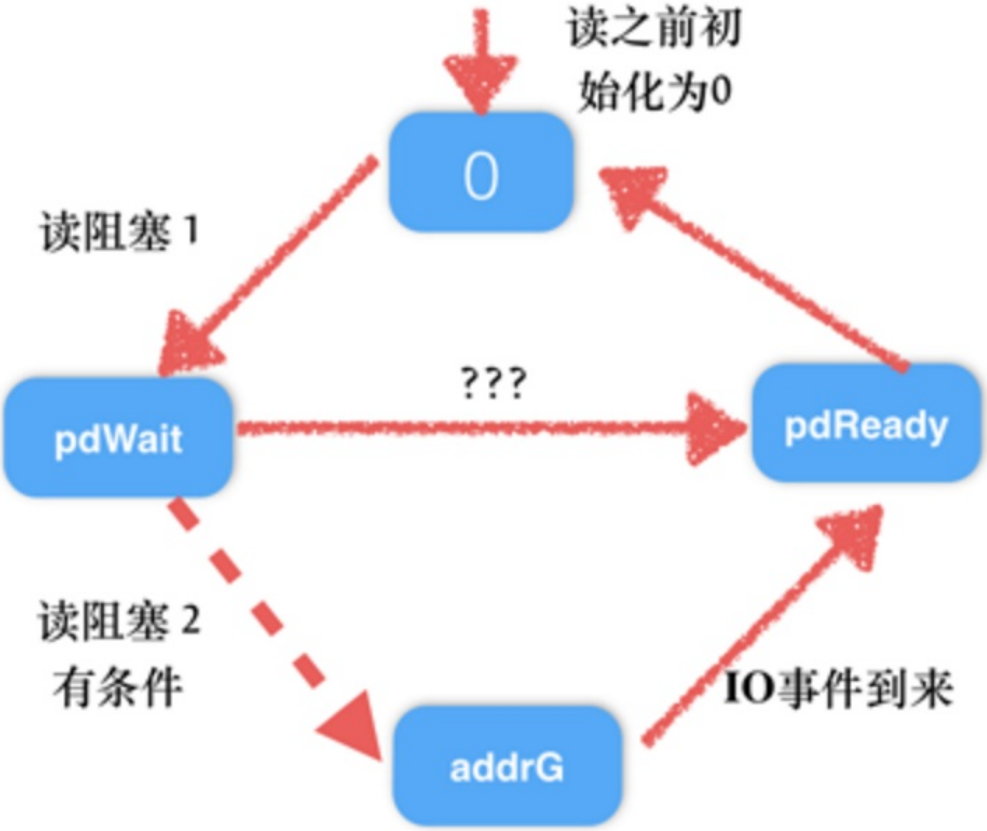
```
func netpollunblock(pd *pollDesc, mode int32, ioready bool) *g {
    gpp := &pd.rg
    if mode == 'w' {
        gpp = &pd.wg
    }
    for {
        old := *gpp
        // 如果已经调用过netpollunblock
        // 这次返回nil
        if old == pdReady {
            return nil
        }
        if old == 0 && !ioready {
            return nil
        }
        var new uintptr

        if ioready {
            new = pdReady
        }
        // 如果io事件到来了，将pd.rg状态设置为pdReady
        // 如果是由于超时被唤醒,pd.rg状态将是0
        // 如果casuintptr失败，继续循环
        if casuintptr(gpp, old, new) {
            if old == pdReady || old == pdWait {
                old = 0
            }
            // 将阻塞在pd.rg上的协程g地址返回
            return (*g)(unsafe.Pointer(old))
        }
    }
}
```

netpollunblock返回了阻塞在pd上的协程g地址（如果有的话），并且将pd.rg状态设置为pdReady，表示netpollunblock阻塞的协程g已经被唤醒了。再来调用的话是唤醒不到任何协程的。

另外，由于netpollunblock内部使用了循环，且netpollunblock可能会被多协程并发访问，这就导致了pd.rg状态可能有很多(0/pdWait/pdReady等)，因此内部做了复杂的状态判断。

总结

上面我们仔细梳理了pd.rg状态机模型，下图整理状态变化的一些过程：



附录三

说明

前面几篇文章我们通过测试、分析代码看到了golang网络超时的实现机制。

分析发现，golang内部实现的超时与我们理解的os的超时有着本质不同。golang的超时是针对一次读为其设置了read deadline，而且是一个绝对时间：一旦时间到期，继续读该网络连接便会出现io timeout的错误。

这里我们来简单总结下，如果我们要实现超时有哪几种方法。

方案一：使用go接口

本章中仔细描述了go的超时接口、实现方案以及可能的坑。那如果要使用go自带的超时功能，我们该如何用呢？

如果我们要使用**golang**的超时的话，必须要小心谨慎了：在每次读之前我们都要去设置下次超时时间，否则会出现一次读很快就出现超时错误。

即以下两个API的调用总是出现在一起的：

```
SetDeadline(t time.Time) error
Read()
```

这样可以避免我们前面说过的问题：每次重设超时能将老的超时定时器给删掉。如果老的超时定时器回调函数已经触发，也不会产生任何影响。因为此时内部的seq++了，老的超时定时器的回调函数触发时能判断自己已经过时了，于是就什么也不做了。

方案二:自己管理超时

该方案采用类似golang http库的实现机制，在http.Client的Get方法中，可以为其设置一个超时时间，用法如下：

```
// 设置Get的超时时间为5s
client = http.Client{Timeout: 5 * time.Second}
client.Get(url)
我们看看http.Client.Get()是如何实现超时的：

func (c *Client) Get(url string) (resp *Response, err error) {
    req, err := NewRequest("GET", url, nil)
    if err != nil {
        return nil, err
    }
    return c.doFollowingRedirects(req, shouldRedirectGet)
}

func (c *Client) doFollowingRedirects(ireq *Request, shouldRedir
ect func(int) bool) (resp *Response, err error) {
    ......
    if c.Timeout > 0 {
        wasCanceled = func() bool { return atomic.LoadInt32(&ato
micWasCanceled) != 0 }

      timer = time.AfterFunc(c.Timeout, func() {
           atomic.StoreInt32(&atomicWasCanceled, 1)
           reqmu.Lock()
           defer reqmu.Unlock()
           tr.CancelRequest(req)
       })
    }
}
```

可以看到http库压根就没利用golang socket的超时，而是在上层实现了自己的超时机制：使用time.AfterFunc()方法在超时时间后执行了一个取消request方法。

内存管理

从该节起，我们主要研究Go内存管理模块。Go runtime的内存管理设计思想主要继承自tcmalloc，这决定了其内存管理无论是各方面都有着比较优秀的性能。

任何一门语言对内存的管理都是语言极其核心的模块，尤其是对类似go此种自带垃圾回收的语言。因为内存申请释放是程序中最频繁的操作，实现时要考虑性能和空间利用率的高效。

所有的从os申请内存或者向os释放内存代价较大。可取的做法是在用户态维护内存池，一次性向os申请较大块的内存，程序对象的申请和释放都在内存池内进行。

用户程序申请释放对象一般以object为单位，而object大小千变万化。这要求内存池在管理时必须进行合理的妥协：我们无法做到在内存池管理时维护如此细粒度的映射关系。可以忍受一定程度的内存浪费来换取实现的简洁性。

本章节我们将从“核心数据结构”、“内存分配算法”、“核心**API**实现”等小节来一步步地由浅入深地描述Go的内存管理机制。

Stack内存管理

介绍

要启动go协程，必须要为其准备栈空间，用来存储函数内变量、执行函数调用等。在go协程退出后，其占用的stack内存也会随之一同释放。

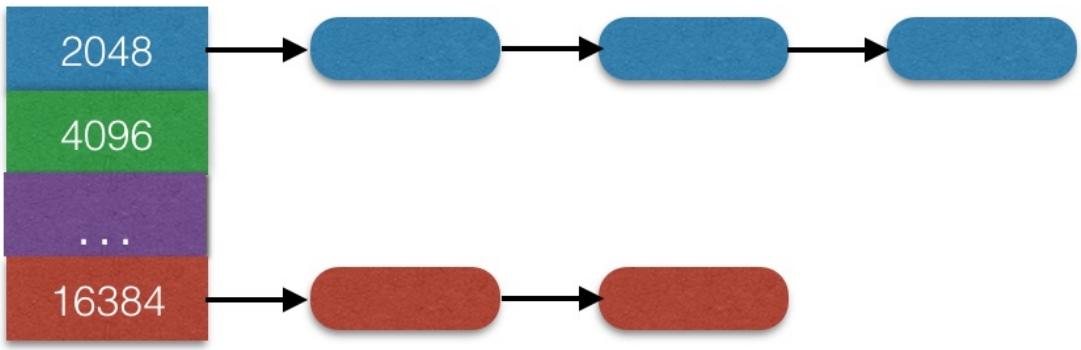
实际应用中协程的启动、退出可能会比较频繁，runtime必须要做点什么来保证启动、销毁协程的代价尽量小。而申请、释放stack空间所需内存则是一个比较大的开销，因此，go runtime采用了stack cache pool来缓存一定数量的stack memory。申请时从stack cache pool分配，释放时首先归还给stack cache pool。

主要思想

stack cache pool的主要思想是按照固定大小划分成多级：每级别其实是一个空闲stack队列：最小的stack size是_FixedStack=2KB。级别之间的大小按照2倍的关系递增。

同时为了效率考虑，除了全局的stack cache pool外，每个线程m还有自己的stack cache。这样，线程在分配的时候可以优先从自己的stack cache中查找，提高效率。

初始时，stack cache pool为空：当有stack分配需求时，首先向os申请一块较大的内存，将其按照该级别的stack大小切割后放在空闲stack列表上，然后再从该空闲stack列表上分配。主要结构如下：



数据结构

```
type mspan struct {
        next     *mspan    // in a span linked list
        prev     *mspan    // in a span linked list
        start    pageID    // starting page number
        npages   uintptr   // number of pages in span
        freelist gclinkptr // list of free objects
        // sweep generation:
        // if sweepgen == h->sweepgen - 2, the span needs sweepi
ng
        // if sweepgen == h->sweepgen - 1, the span is currently
 being swept
        // if sweepgen == h->sweepgen, the span is swept and rea
dy to use
        // h->sweepgen is incremented by 2 after every GC

        sweepgen    uint32
        divMul      uint32   // for divide by elemsize - divMagi
c.mul
        ref         uint16   // capacity - number of objects in
freelist
        sizeclass   uint8    // size class
        incache     bool     // being used by an mcache
        state       uint8    // mspaninuse etc
        needzero    uint8    // needs to be zeroed before alloca
tion
        divShift    uint8    // for divide by elemsize - divMagi
c.shift
        divShift2   uint8    // for divide by elemsize - divMagi
c.shift2
        elemsize    uintptr  // computed from sizeclass or from
npages
        unusedsince int64    // first time spotted by gc in mspa
nfree state
        nprereleased  uintptr  // number of pages released to the
os
        limit       uintptr  // end of data in span
        speciallock mutex    // guards specials list
```

```
        specials    *special // linked list of special records s
orted by offset.
        baseMask    uintptr  // if non-0, elemsize is a power of
 2, & this will get object allocation base
}
```

mspan是在go内存管理模块机器核心的一个数据结构，我们会在后面的章节中自习描述该结构的含义。

```
var stackpoolmu mutex
var stackpool [_NumStackOrders]mspan
```

主要流程

stackinit()

go程序在启动时首先会执行go runtime的初始化，此时会调用stackinit()来初始化stack cache pool。

```
func stackinit() {
        if _StackCacheSize&_PageMask != 0 {
                throw("cache size must be a multiple of page siz
e")
        }
        for i := range stackpool {
                mSpanList_Init(&stackpool[i])
        }
        mSpanList_Init(&stackFreeQueue)
}
```

初始化做的事情非常简单：初始化每个stack size级别的链表为空即可。

stackalloc(n uint32)

```
func stackalloc(n uint32) (stack, []stkbar) {
```

```
        ......

        if stackCache != 0 && n < _FixedStack<<_NumStackOrders &
& n < _StackCacheSize {
                order := uint8(0)
                n2 := n
                for n2 > _FixedStack {
                        order++
                        n2 >>= 1
                }
                var x gclinkptr
                c := thisg.m.mcache
                if c == nil || thisg.m.preemptoff != "" || thisg
.m.helpgc != 0 {
                        // 线程本地stack cache为空
                        // 直接从全局stack cache pool分配
                        lock(&stackpoolmu)
                        x = stackpoolalloc(order)
                        unlock(&stackpoolmu)
                } else {
                        // 直接从线程本地缓存分配
                        x = c.stackcache[order].list
                        if x.ptr() == nil {
                                // 如果线程本地分配完,则从全局stack
cache pool分配一批给线程
                                // 再从线程的local stack cache分配
                                stackcacherefill(c, order)
                                x = c.stackcache[order].list
                        }
                        c.stackcache[order].list = x.ptr().next
                        c.stackcache[order].size -= uintptr(n)
                }
                v = (unsafe.Pointer)(x)
        } else {
                s := mHeap_AllocStack(&mheap_, round(uintptr(n),
 _PageSize)>>_PageShift)
                if s == nil {
                        throw("out of memory")
                }
                v = (unsafe.Pointer)(s.start << _PageShift)
```

```
        }
        ......
}
```

这里的分配算法思想是：

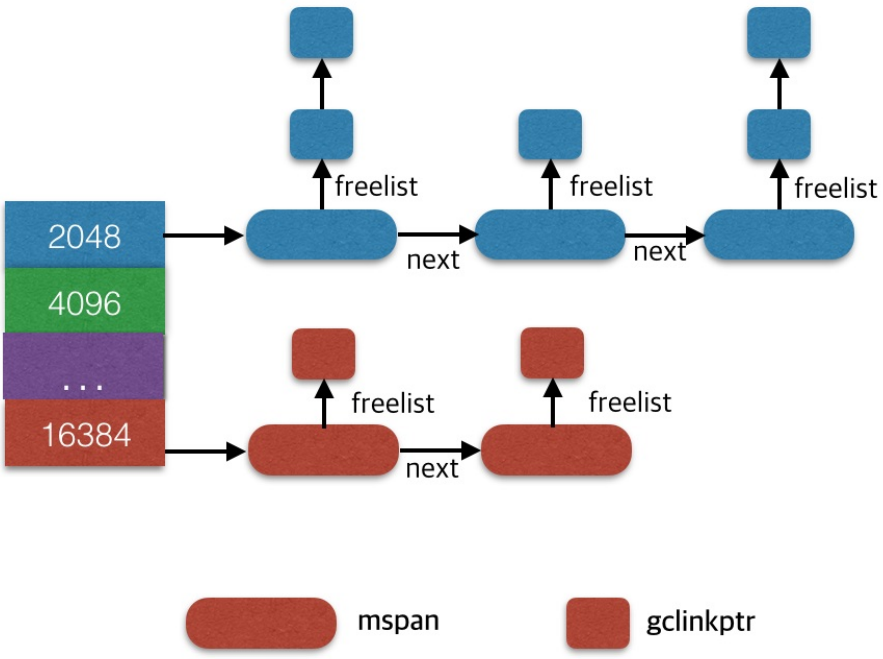
1. 如果线程m的local stack cache为空，则直接从全局的stack cache pool中分配；
2. 否则先从线程的local stack cache中分配，如果无法分配除，则需要先给线程分配出一批stack，赋给该线程，再从线程local stack cache中再分配。
3.

stackpoolalloc(order uint8)

```
// Allocates a stack from the free pool.  Must be called with
// stackpoolmu held.
func stackpoolalloc(order uint8) gclinkptr {
        list := &stackpool[order]
        s := list.next
        // 如果该队列为空了,需要向heap管理模块再申请
        // 分配的结果为一个mspan结构
        if s == list {
                s = mHeap_AllocStack(&mheap_, _StackCacheSize>>_
PageShift)
                if s == nil {
                        throw("out of memory")
                }
                if s.ref != 0 {
                        throw("bad ref")
                }
                if s.freelist.ptr() != nil {
                        throw("bad freelist")
                }
                // 将申请得到的空间mspan按照该级别的stack大小拆分成多项
(gclinkptr)并添加到mspan的空闲链表(mspan.freelist)中
                for i := uintptr(0); i < _StackCacheSize; i += _
FixedStack << order {
                        x := gclinkptr(uintptr(s.start)<<_PageSh
```

```
ift + i)
                        x.ptr().next = s.freelist
                        s.freelist = x
                }
                // 将分配的mspan插入到pool的空闲span链表
                mSpanList_Insert(list, s)
        }
        x := s.freelist
        if x.ptr() == nil {
                throw("span has no free stacks")
        }
        s.freelist = x.ptr().next
        s.ref++
        // 如果分配导致s(mspan)的空闲链表变空
        // 将其从pool的空闲span链表中摘除
        if s.freelist.ptr() == nil {
                // all stacks in s are allocated.
                mSpanList_Remove(s)
        }
        return x
}
```

分配后的系统stack cache pool结构如下：



stackfree()

```
func stackfree(stk stack, n uintptr) {
        ......

        if stackCache != 0 && n < _FixedStack<<_NumStackOrders &
& n < _StackCacheSize {
                order := uint8(0)
                n2 := n
                for n2 > _FixedStack {
                        order++
                        n2 >>= 1
                }
                x := gclinkptr(v)
                c := gp.m.mcache
                // 如果禁用了m local stack cache,直接归还至全局stack
 cache pool
                if c == nil || gp.m.preemptoff != "" || gp.m.hel
pgc != 0 {
                        lock(&stackpoolmu)
                        stackpoolfree(x, order)
                        unlock(&stackpoolmu)
                } else {
                // 如果m的local stack cache超标
                // 先归还一部分(一半)至全局stack cache pool
                if c.stackcache[order].size >= _StackCacheSize {
                                stackcacherelease(c, order)
                        }
                        x.ptr().next = c.stackcache[order].list
                        c.stackcache[order].list = x
                        c.stackcache[order].size += n
                }
        // 直接归还，暂时不考虑
        } else {
                s := mHeap_Lookup(&mheap_, v)
                if s.state != _MSpanStack {
                        println(hex(s.start<<_PageShift), v)
                        throw("bad span state")
                }
```

```
                if gcphase == _GCoff {
                        // Free the stack immediately if we're
                        // sweeping.
                        mHeap_FreeStack(&mheap_, s)
                } else {
                        // Otherwise, add it to a list of stack
spans
                        // to be freed at the end of GC.
                        //
                        // TODO(austin): Make it possible to re-
use
                        // these spans as stacks, like we do for
 small
                        // stack spans. (See issue #11466.)
                        lock(&stackpoolmu)
                        mSpanList_Insert(&stackFreeQueue, s)
                        unlock(&stackpoolmu)
                }
        }
}
```

stackpoolfree()

```
// Adds stack x to the free pool.  Must be called with stackpool
mu held.
func stackpoolfree(x gclinkptr, order uint8) {
        // 查找该x所属的mspan
        s := mHeap_Lookup(&mheap_, (unsafe.Pointer)(x))
        if s.state != _MSpanStack {
                throw("freeing stack not in a stack span")
        }
        // 如果s之前的freelist为空，已经从pool的空闲span链表中摘除
        // 现在归还了,说明有空闲空间了，将其重新添加到pool的空闲span链表
中
        if s.freelist.ptr() == nil {
                mSpanList_Insert(&stackpool[order], s)
        }
        x.ptr().next = s.freelist
```

```
        s.freelist = x
        s.ref--
        // s.ref==0说明span的freelist所有栈空间全部被释放
        // 此时就可以将该span的空间释放
        if gcphase == _GCoff && s.ref == 0 {
                // Span is completely free. Return it to the hea
p
                // immediately if we're sweeping.
                //
                // If GC is active, we delay the free until the
end of
                // GC to avoid the following type of situation:
                //
                // 1) GC starts, scans a SudoG but does not yet
mark the SudoG.elem pointer
                // 2) The stack that pointer points to is copied
                // 3) The old stack is freed
                // 4) The containing span is marked free
                // 5) GC attempts to mark the SudoG.elem pointer
. The
                //    marking fails because the pointer looks li
ke a
                //    pointer into a free span.
                //
                // By not freeing, we prevent step #4 until GC i
s done.
                mSpanList_Remove(s)
                s.freelist = 0
                mHeap_FreeStack(&mheap_, s)
        }
}
```

Heap管理

内置运行时的编程语言通常会抛弃传统的内存分配方式,改由自主管理内存。这样可以完成类 似预分配、内存池等操作,以避开频繁地申请释放内存引起的系统调而导致的性能问题。当然,还有一个重要原因是 为了更好地配合语言的垃圾回收机制。

Go内部实现也不例外，go runtime接管了所有的内存申请和释放动作。在os上层实现了内存池机制(源自tcmalloc设计)。

Go内存池管理的核心数据结构为mHeap。该结构管理从os申请的大块内存，将大块内存切分成多种不同大小的小块，每种小块由数据结构mspan表示。mheap通过数组+链表的方式来维护所有的空闲span。

应用程序在申请内存时一般都是以object为单位。在go runtime内部必须要计算object大小，然后找到合适的mspan大小。从这里面分配出想要的内存，返回给应用程序。

程序在向go runtime申请分配某种object所需内存时，会计算出object占用的内存空间，然后找到最接近的mspan(因为mspan管理的块大小是按照固定倍数增长的方案。如一个17字节的object需要的块大小应该是24字节，存在轻微的内存浪费)，将其分配出去。

对象分配

```
// implementation of new builtin
func newobject(typ *_type) unsafe.Pointer {
        flags := uint32(0)
        if typ.kind&kindNoPointers != 0 {
                flags |= flagNoScan
        }
        return mallocgc(uintptr(typ.size), typ, flags)
}

// Allocate an object of size bytes.
// Small objects are allocated from the per-P cache's free lists
.
// Large objects (> 32 kB) are allocated straight from the heap.
```

```
func mallocgc(size uintptr, typ *_type, flags uint32) unsafe.Poi
nter {
// Set mp.mallocing to keep from being preempted by GC.
        mp := acquirem()
        if mp.mallocing != 0 {
                throw("malloc deadlock")
        }
        if mp.gsignal == getg() {
                throw("malloc during signal")
        }
        mp.mallocing = 1

        shouldhelpgc := false
        dataSize := size
        c := gomcache()
        var s *mspan
        var x unsafe.Pointer
        if size <= maxSmallSize {
                // 对极小对象(<=16B)分配的优化
                if flags&flagNoScan != 0 && size < maxTinySize {
                        off := c.tinyoffset
                        // Align tiny pointer for required (cons
ervative) alignment.
                        if size&7 == 0 {
                                off = round(off, 8)
                        } else if size&3 == 0 {
                                off = round(off, 4)
                        } else if size&1 == 0 {
                                off = round(off, 2)
                        }
                        // 将众多小对象存储在同一个block内
                        if off+size <= maxTinySize && c.tiny !=
nil {
                                x = add(c.tiny, off)
                                c.tinyoffset = off + size
                                c.local_tinyallocs++
                                mp.mallocing = 0
                                releasem(mp)
                                return x
                        }
```

```
                        // 存储小对象的块为nil或者之前的小对象块已经容
纳不下了，申请一个新的小对象块并
                        // Allocate a new maxTinySize block.
                        s = c.alloc[tinySizeClass]
                        v := s.freelist
                        if v.ptr() == nil {
                                systemstack(func() {
                                        mCache_Refill(c, tinySiz
eClass)
                                })
                                shouldhelpgc = true
                                s = c.alloc[tinySizeClass]
                                v = s.freelist
                        }
                        s.freelist = v.ptr().next
                        s.ref++
                        prefetchnta(uintptr(v.ptr().next))
                        x = unsafe.Pointer(v)
                        (*[2]uint64)(x)[0] = 0
                        (*[2]uint64)(x)[1] = 0
                        // See if we need to replace the existin
g tiny block with the new one
                        // based on amount of remaining free spa
ce.
                        if size < c.tinyoffset {
                                c.tiny = x
                                c.tinyoffset = size
                        }
                }
```

Heap回收

Heap释放

核心数据结构

Go内存管理模块的核心数据结构比较少:

- **mheap**：管理全局的从os申请的虚拟内存空间；
- **mspan**：将mheap按照固定大小切分而成的细粒度的内存区块，每个区块映射了虚拟内存中的若干连续页面，页大小由Go内部定义；
- **mcache**：与线程相关缓存，该结构的存在是为了减少内存分配时的锁操作，优化内存分配性能。
- **mcentral**：集中内存池，线程在本地分配失败后，尝试向mcentral申请，如果mcentral也没有资源，则尝试向mheap分配。

mheap

```
type mheap struct {
        lock      mutex
        // 空闲mspan链表
        free      [_MaxMHeapList]mspan
        freelarge mspan
        busy      [_MaxMHeapList]mspan
        busylarge mspan
        allspans  **mspan
        gcspans   **mspan
        nspan     uint32
        sweepgen  uint32
        sweepdone uint32

        spans        **mspan
        spans_mapped uintptr

        // Proportional sweep
        spanBytesAlloc    uint64  // bytes of spans allocated th
is cycle; updated atomically
        pagesSwept        uint64  // pages swept this cycle; upd
ated atomically
        sweepPagesPerByte float64 // proportional sweep ratio; w
```

```
ritten with lock, read without

        // Malloc stats.
        largefree  uint64
        nlargefree uint64
        nsmallfree [_NumSizeClasses]uint64

        // range of addresses we might see in the heap
        bitmap         uintptr
        bitmap_mapped  uintptr
        arena_start    uintptr
        arena_used     uintptr // always mHeap_Map{Bits,Spans} b
efore updating
        arena_end      uintptr
        arena_reserved bool

        // central free lists for small size classes.
        // the padding makes sure that the MCentrals are
        // spaced CacheLineSize bytes apart, so that each MCentr
al.lock
        // gets its own cache line.
        central [_NumSizeClasses]struct {
                mcentral mcentral
                pad      [_CacheLineSize]byte
        }

        spanalloc             fixalloc // allocator for span*
        cachealloc            fixalloc // allocator for mcache*
        specialfinalizeralloc fixalloc // allocator for specialf
inalizer*
        specialprofilealloc   fixalloc // allocator for specialp
rofile*
        speciallock           mutex    // lock for special recor
d allocators.
}
```

mheap内部主要维护多级free以及busy mspan链表。每个级别的mspan链表上的mspan总大小相同(即映射的内存页面数一样)。

除此之外，mheap内部还记录了必须的管理信息，如记录位图位置等。这些我们在后面遇到的时候作一一分析。

mspan

```
type mspan struct {
        // 每个mspan会根据状态被link到特定的双向链表中(如free、busy链表
)
        next     *mspan
        prev     *mspan
        // 该mspan映射的起始内存page以及page数
        start    pageID
        npages   uintptr
        freelist gclinkptr
        // sweep generation:
        // if sweepgen == h->sweepgen - 2, the span needs sweepi
ng
        // if sweepgen == h->sweepgen - 1, the span is currently
 being swept
        // if sweepgen == h->sweepgen, the span is swept and rea
dy to use
        // h->sweepgen is incremented by 2 after every GC
        sweepgen    uint32
        divMul      uint32
        ref         uint16
        sizeclass   uint8    // size class
        incache     bool     // being used by an mcache
        state       uint8    // mspaninuse etc
        needzero    uint8    // needs to be zeroed before alloca
tion
        divShift    uint8
        divShift2   uint8
        // ???
        elemsize    uintptr
        unusedsince int64    // first time spotted by gc in mspa
nfree state
        npreleased  uintptr  // number of pages released to the
os
        limit       uintptr  // end of data in span
```

```
        speciallock mutex    // guards specials list
        specials    *special // linked list of special records s
orted by offset.
        baseMask    uintptr  // if non-0, elemsize is a power of
 2, & this will get object allocation base
}
```

mspan中记录的最关键信息是freelist：所有的对象分配最终都被委派到从mspan上以分配合适大小的内存空间。因此，mspan在经历分配、释放等过程之后，也会变得支离破碎，如下图所示：

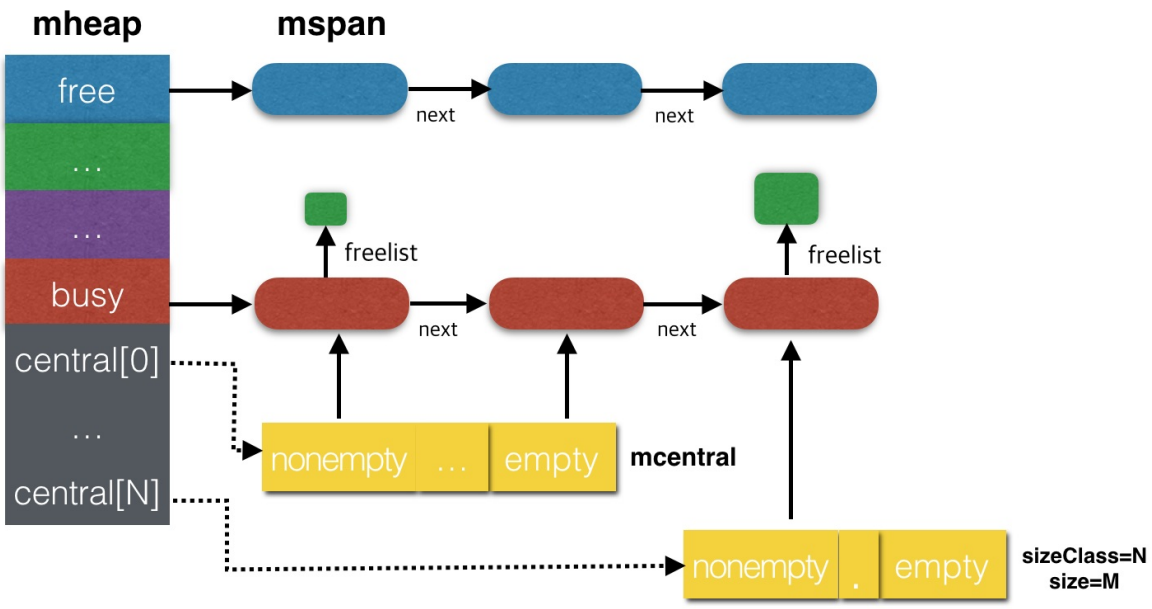
mcache

```
// Per-thread (in Go, per-P) cache for small objects.
// No locking needed because it is per-thread (per-P).
type mcache struct {
        // The following members are accessed on every malloc,
        // so they are grouped here for better caching.
        next_sample      int32
        local_cachealloc uintptr
        local_scan       uintptr
        // Allocator cache for tiny objects w/o pointers.
        // See "Tiny allocator" comment in malloc.go.
        tiny             unsafe.Pointer
        tinyoffset       uintptr
        local_tinyallocs uintptr // number of tiny allocs not co
unted in other stats

        // The rest is not accessed on every malloc.
        alloc [_NumSizeClasses]*mspan
        // Local allocator stats, flushed during GC.
        local_nlookup    uintptr                  // number of p
ointer lookups
        local_largefree  uintptr                  // bytes freed
 for large objects (>maxsmallsize)
        local_nlargefree uintptr                  // number of f
rees for large objects (>maxsmallsize)
        local_nsmallfree [_NumSizeClasses]uintptr // number of f
rees for small objects (<=maxsmallsize)
}
```

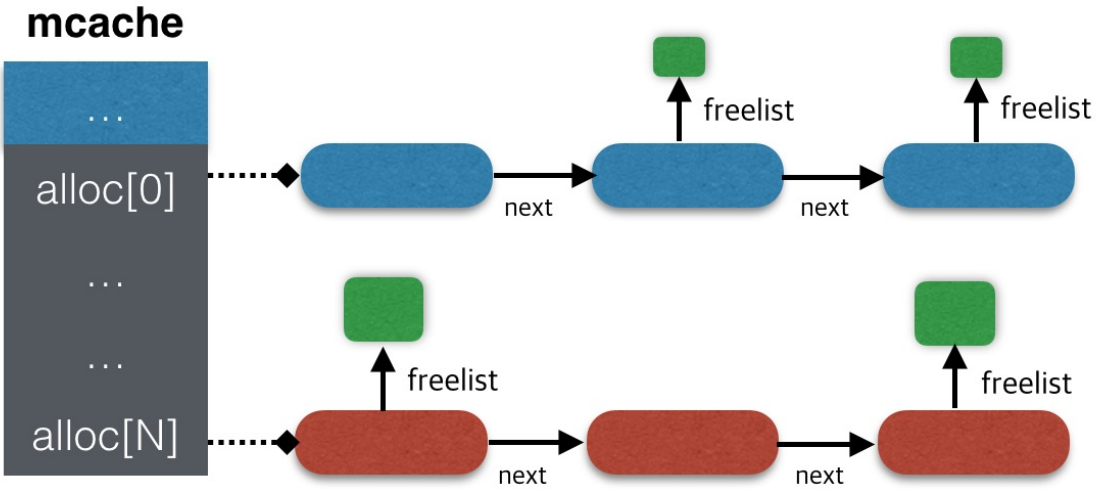
mcentral

```
// Central list of free objects of a given size.
type mcentral struct {
    lock      mutex
    // mcentral负责分配的内存块大小
    sizeclass int32
    // 拥有空闲object的mspan链表
    nonempty  mspan
    // 无空闲object的mspan链表
    empty     mspan
}
```

数据结构关系图



1. mheap管理向os申请、释放、组织span；
2. mcentral按照自己管理的块大小将span划分成多个小block，并分配给mcache；
3. mspan是数据的实际存储区域，按照central管理的块规格被切分成小block;
4. mcache管理不同规格(块大小)的mspan：规格相同的mspan被链接到同一个链表中。


mcache
...
alloc[0]
...
...
alloc[N]
freelist
freelist
next
next
freelist
freelist
next
next

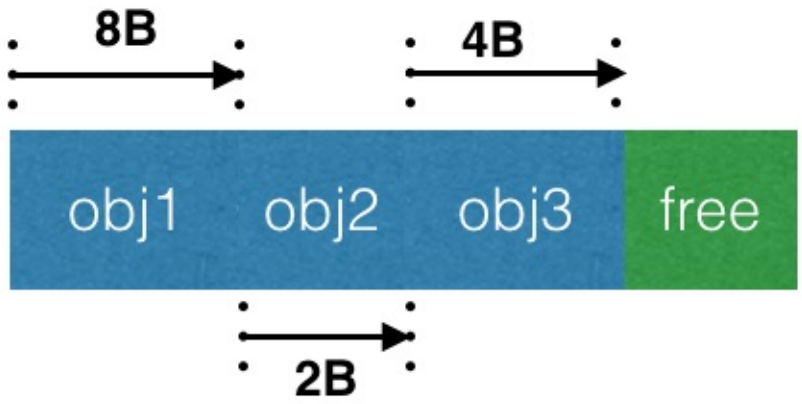
内存分配算法

go runtime对不同大小的对象采取的分配算法有一些不同：主要目的是提高时间和空间效率。

极小对象分配

对于小于maxTinySize(16B)字节对象的内存分配请求。go采取了将小对象合并存储的解决方案。

每个线程在本地维护了专门的memory block来存储tiny object。每次分配时计算该block内是否可容纳该tiny object，如果可以，直接返回block的起始内存地址加上block内当前的使用偏移。如下图所示：



note:对于极小对象，其实际占用的内存起始地址按照一定的规则进行对其。对其规则如下：

1. 对于大于等于8B的对象，其内存地址按照8B对齐；
2. 对于小于8B大于等于4B的对象，其内存地址按照4B对齐；
3. 对于小于4B大于1B的对象，其内存地址按照2B对齐；
4. 对于1B对象，无对齐要求。

这种对齐要求可能会造成一定的内存浪费，然后却能带来内存访问效率的提升。

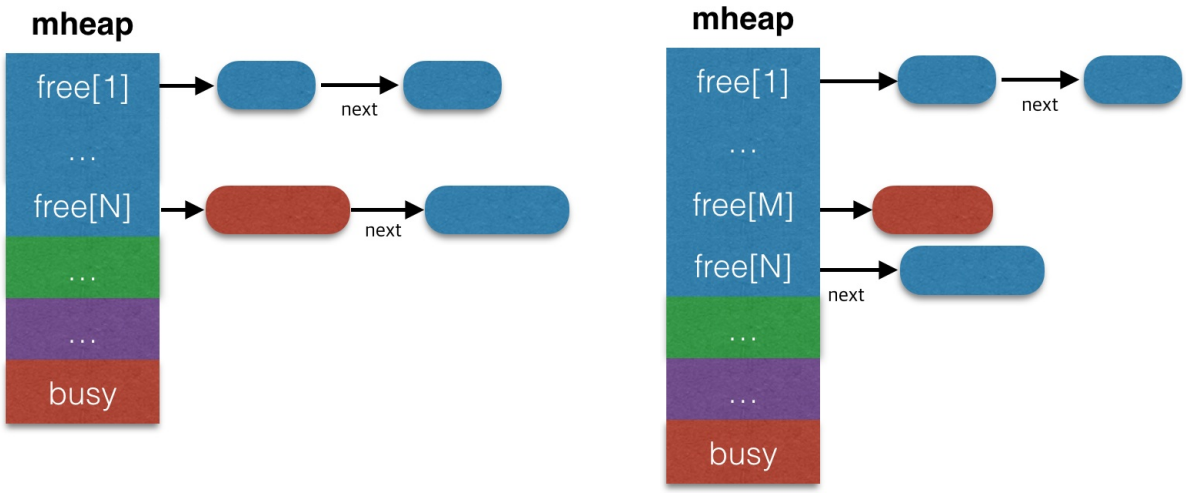
如果当前正在使用的memory block无法容纳当前申请的对象。则需要申请一个新的memory block。此时需要考虑的一个问题是原来的正在被使用的memory block是否需要被替换出去。这个留给读者自己思考吧。

大对象分配

对于超过_MaxSmallSize(32KB)的object，go并非使用上面这些复杂的内存分配机制，因为过于复杂，代价太大，得不偿失。

对于此类的大对象，go直接从heap上分配内存。分配算法：

1. 计算大对象占用的内存页面数；
2. 从heap的free mspan中查找具备合适页面数的mspan，如果找到进入步骤4；
3. 从heap的large mspan链表中采用best fit算法找到最合适的mspan；
4. 至此，找到合适的mspan，但是该mspan的空闲页面数可能超过申请的数量，此时需要将该mspan切割，给用户返回需求的页面数，剩余的页面数构成一个全新的mspan，继续放到heap的空闲mspan链表中。



上图中红色的mspan被分配出去部分memory page后，余下的空间作为一个新的mspan被插入到全新的链表free[M]。M为多少取决于该mspan剩余的memory page数量。

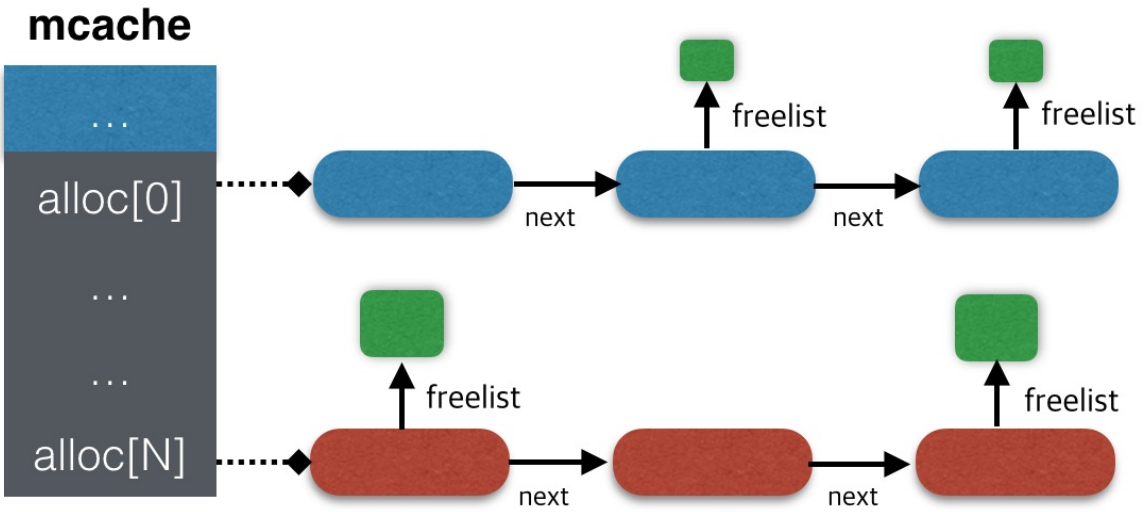
中等大小对象分配

中等大小的对象是go内部实现较为复杂的部分。主要问题存在：

1. Go需要根据对象的大小分配合适的内存块；
2. 为了提高分配性能，需要实现线程内存缓存，分配时优先从线程内缓存分配，不够时再从中央分配器分配来填充本地缓存。

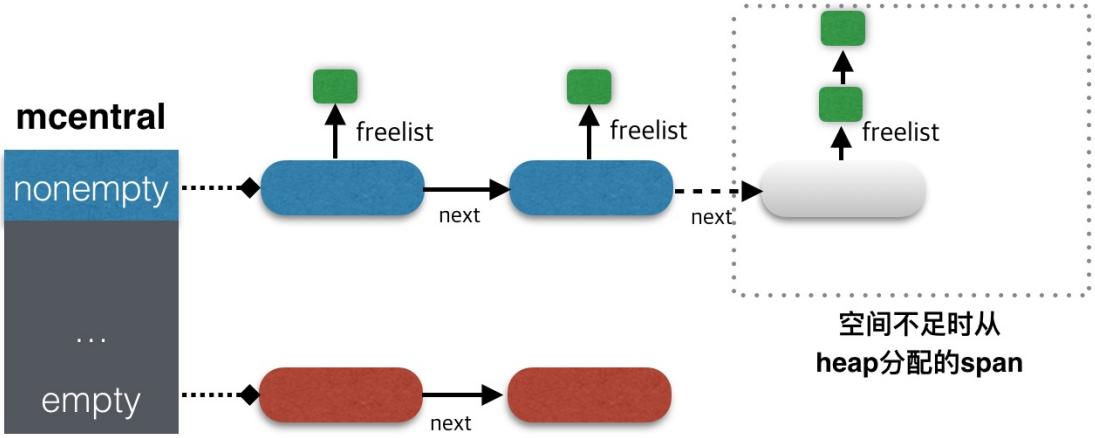
从m的内存块cache中分配空闲内存块算法如下：

1. 根据对象大小和既定规则来计算需要分配的块大小级别，具体计算规则后面仔细描述；
2. 查找本地缓存中该级别的空闲内存块，如果为空，从中央分配器中分配一定数量的该级别的空闲内存块(可能分配一个span并切割)；
3. 从m的缓存中分配空闲内存块。



从中央分配器中分配空闲内存块算法如下：

1. 从请求的内存块大小的级别的mcentral中查找nonempty链表，不为空则直接分配，返回；
2. 扫描empty链表，看看是否有已经垃圾回收完(不确定??)的mspan,如果有，也可以分配出来;
3. 到此，说明需要向heap中再申请空闲mspan，将其按照既定块大小切割成小内存块，插入到nonempty链表中，再从该mspan种分配内存块即可。


mcentral
nonempty
...
empty
freelist
freelist
freelist
next
next
next
空间不足时从
heap分配的span

核心API实现

mallocinit()

```
func mallocinit() {

    initSizes()

    ...

    // linux下最终调用mmap()来预留虚拟内存空间
    // 预留的大小为_MaxMem,在linux amd_64为512GB
    if ptrSize == 8 && (limit == 0 || limit > 1<<30) {
        // 计算各部分占用的空间,总空间为各项之和
        arenaSize := round(_MaxMem, _PageSize)
        bitmapSize = arenaSize / (ptrSize * 8 / 4)
        spansSize = arenaSize / _PageSize * ptrSize
        spansSize = round(spansSize, _PageSize)

        // 尝试从0xc000000000开始设置保留地址
        // 失败，则尝试0x1c000000000~0x7fc000000000
        for i := 0; i <= 0x7f; i++ {
            switch {
            case GOARCH == "arm64" && GOOS == "darwin":
                p = uintptr(i)<<40 | uintptrMask&(0x0013<<28)
            case GOARCH == "arm64":
                p = uintptr(i)<<40 | uintptrMask&(0x0040<<32)
            default:
                p = uintptr(i)<<40 | uintptrMask&(0x00c0<<32)
            }
            // 之所以多分配一个_PageSize是因为mmap()分配出的可能不是按照_PageSize对其的，如果这样我们需要进行再对其
            pSize = bitmapSize + spansSize + arenaSize + _PageSize
            p = uintptr(sysReserve(unsafe.Pointer(p), pSize, &reserved))
            if p != 0 {
```

```
                break
            }
        }
        if p == 0 {
                ......
        }
    }
    // 按_PageSize对其，也是前面多分配一个_PageSize的原因
    p1 := round(p, _PageSize)

    // 初始化heap管理的各字段
    mheap_.spans = (**mspan)(unsafe.Pointer(p1))
    mheap_.bitmap = p1 + spansSize
    mheap_.arena_start = p1 + (spansSize + bitmapSize)
    mheap_.arena_used = mheap_.arena_start
    mheap_.arena_end = p + pSize
    mheap_.arena_reserved = reserved

    if mheap_.arena_start&(_PageSize-1) != 0 {
        ......
    }
    // 初始化全局heap
    mHeap_Init(&mheap_, spansSize)
    _g_ := getg()

    // 为当前线程绑定cache对象
    _g_.m.mcache = allocmcache()
}
```

内存管理初始化在go进程被启动时调用，负责向os申请预留虚拟内存空间，并将该虚拟空间分为以下几个部分：

```
页所属span指针数组 || GC标记位图 || 用户内存分配区域
+------------------+-------------+-------------------------+
| spans 512MB .....| bitmap 32GB | arena 512GB .....       |
+------------------+-------------+-------------------------+
```

spans 512MB: ???
bitmap 32GB: ???
arena 512GB: ???

mallocgc()

```
// implementation of new builtin
func newobject(typ *_type) unsafe.Pointer {
    flags := uint32(0)
    if typ.kind&kindNoPointers != 0 {
        flags |= flagNoScan
    }
    return mallocgc(uintptr(typ.size), typ, flags)
}

// Allocate an object of size bytes.
// Small objects are allocated from the per-P cache's free lists
.
// Large objects (> 32 kB) are allocated straight from the heap.
func mallocgc(size uintptr, typ *_type, flags uint32) unsafe.Poi
nter {
    // Set mp.mallocing to keep from being preempted by GC.
    mp := acquirem()
    if mp.mallocing != 0 {
        throw("malloc deadlock")
    }
    if mp.gsignal == getg() {
        throw("malloc during signal")
    }
    mp.mallocing = 1

    shouldhelpgc := false
    dataSize := size
    c := gomcache()
    var s *mspan
    var x unsafe.Pointer

    // 小对象(<32KB)分配
```

```
    if size <= maxSmallSize {
        // 对极小对象(<=16B)分配的优化,见“性能优化”
        if flags&flagNoScan != 0 && size < maxTinySize {
            ......
        } else {
            var sizeclass int8
            if size <= 1024-8 {
                sizeclass = size_to_class8[(size+7)>>3]
            } else {
                sizeclass = size_to_class128[(size-1024+127)>>7]
            }
            size = uintptr(class_to_size[sizeclass])
            // 从线程本地cache的freelist分配
            s = c.alloc[sizeclass]
            v := s.freelist
            // 如果线程本地cache的freelist为空
            // 首先从全局heap中为本地线程分配一批
            if v.ptr() == nil {
                systemstack(func() {
                    mCache_Refill(c, int32(sizeclass))
                })
                shouldhelpgc = true
                s = c.alloc[sizeclass]
                v = s.freelist
            }
            s.freelist = v.ptr().next
            s.ref++
            // ?
            prefetchnta(uintptr(v.ptr().next))
            x = unsafe.Pointer(v)
            if flags&flagNoZero == 0 {
                v.ptr().next = 0
                if size > 2*ptrSize && ((*[2]uintptr)(x))[1] !=
0 {
                    memclr(unsafe.Pointer(v), size)
                }
            }
        }
        c.local_cachealloc += size
    // 对大对象(>32KB)的分配
```

```
    } else {
        var s *mspan
        shouldhelpgc = true
        systemstack(func() {
            s = largeAlloc(size, uint32(flags))
        })
        x = unsafe.Pointer(uintptr(s.start << pageShift))
        size = uintptr(s.elemsize)
    }
    ......
    return x
}
```