# 目　录

# 介绍

## 项目介绍

- 本项目地址：https://github.com/qcrao/Go-Questions

`Go 语言` 学习入门和进阶知识。以 `Go 语言` 为突破口，从问题切入，掌握 Go 语言、后端相关的各种硬核知识。希望本项目能在职场表现、项目实战上助你一臂之力！

## 学习交流

你可以加我的微信一起交流：raoquancheng1991。

也可以关注公众号，和更多的人一起学习：

# channel

# 从一个关闭的 channel 仍然能读出数据吗

从一个有缓冲的 channel 里读数据，当 channel 被关闭，依然能读出有效值。只有当返回的 ok 为 false 时，读出的数据才是无效的。

```
func main() {
    ch := make(chan int, 5)
    ch <- 18
    close(ch)
    x, ok := <-ch
    if ok {
        fmt.Println("received: ", x)
    }

    x, ok = <-ch
    if !ok {
        fmt.Println("channel closed, data invalid.")
    }
}
```

运行结果：

```
received:  18
channel closed, data invalid.
```

先创建了一个有缓冲的 channel，向其发送一个元素，然后关闭此 channel。之后两次尝试从 channel 中读取数据，第一次仍然能正常读出值。第二次返回的 ok 为 false，说明 channel 已关闭，且通道里没有数据。

具体过程可以参考“从 channel 接收数据的过程是怎样的”一节。

# channel 有哪些应用

Channel 和 goroutine 的结合是 Go 并发编程的大杀器。而 Channel 的实际应用也经常让人眼前一亮，通过与 select，cancel，timer 等结合，它能实现各种各样的功能。接下来，我们就要梳理一下 channel 的应用。

## 停止信号

“如何优雅地关闭 channel”那一节已经讲得很多了，这块就略过了。

channel 用于停止信号的场景还是挺多的，经常是关闭某个 channel 或者向 channel 发送一个元素，使得接收 channel 的那一方获知道此信息，进而做一些其他的操作。

## 任务定时

与 timer 结合，一般有两种玩法：实现超时控制，实现定期执行某个任务。

有时候，需要执行某项操作，但又不想它耗费太长时间，上一个定时器就可以搞定：

```
select {
    case <-time.After(100 * time.Millisecond):
    case <-s.stopc:
        return false
}
```

等待 100 ms 后，如果 s.stopc 还没有读出数据或者被关闭，就直接结束。这是来自 etcd 源码里的一个例子，这样的写法随处可见。

定时执行某个任务，也比较简单：

```
func worker() {
    ticker := time.Tick(1 * time.Second)
    for {
        select {
        case <- ticker:
            // 执行定时任务
            fmt.Println("执行 1s 定时任务")
        }
    }
}
```

每隔 1 秒种，执行一次定时任务。

## 解耦生产方和消费方

服务启动时，启动 n 个 worker，作为工作协程池，这些协程工作在一个 `for {}` 无限循环里，从某个 channel 消费工作任务并执行：

```
func main() {
    taskCh := make(chan int, 100)
    go worker(taskCh)
```

```
    // 塞任务
    for i := 0; i < 10; i++ {
        taskCh <- i
    }

    // 等待 1 小时
    select {
    case <-time.After(time.Hour):
    }
}

func worker(taskCh <-chan int) {
    const N = 5
    // 启动 5 个工作协程
    for i := 0; i < N; i++ {
        go func(id int) {
            for {
                task := <- taskCh
                fmt.Printf("finish task: %d by worker %d\n", task, id)
                time.Sleep(time.Second)
            }
        }(i)
    }
}
```

5 个工作协程在不断地从工作队列里取任务，生产方只管往 channel 发送任务即可，解耦生产方和消费方。

程序输出：

```
finish task: 1 by worker 4
finish task: 2 by worker 2
finish task: 4 by worker 3
finish task: 3 by worker 1
finish task: 0 by worker 0
finish task: 6 by worker 0
finish task: 8 by worker 3
finish task: 9 by worker 1
finish task: 7 by worker 4
finish task: 5 by worker 2
```

# 控制并发数

有时需要定时执行几百个任务，例如每天定时按城市来执行一些离线计算的任务。但是并发数又不能太高，因为任务执行过程依赖第三方的一些资源，对请求的速率有限制。这时就可以通过 channel 来控制并发数。

下面的例子来自《Go 语言高级编程》：

```
var limit = make(chan int, 3)

func main() {
    // ...........
    for _, w := range work {
        go func() {
            limit <- 1
            w()
            <-limit
        }()
```

```
    }
    // ...........
}
```

构建一个缓冲型的 channel，容量为 3。接着遍历任务列表，每个任务启动一个 goroutine 去完成。真正执行任务，访问第三方的动作在 w() 中完成，在执行 w() 之前，先要从 limit 中拿“许可证”，拿到许可证之后，才能执行 w()，并且在执行完任务，要将“许可证”归还。这样就可以控制同时运行的 goroutine 数。

这里，`limit <- 1` 放在 func 内部而不是外部，原因是：

> 如果在外层，就是控制系统 goroutine 的数量，可能会阻塞 for 循环，影响业务逻辑。

> limit 其实和逻辑无关，只是性能调优，放在内层和外层的语义不太一样。

还有一点要注意的是，如果 w() 发生 panic，那“许可证”可能就还不回去了，因此需要使用 defer 来保证。

# 参考资料

【channel 应用】https://www.s0nnet.com/archives/go-channels-practice

【应用举例】https://zhuyasen.com/post/go_queue.html

【应用】https://tonybai.com/2014/09/29/a-channel-compendium-for-golang/

【Go 语言高级并发编程】https://chai2010.cn/advanced-go-programming-book/

# 如何优雅地关闭 channel

关于 channel 的使用，有几点不方便的地方：

1. 在不改变 channel 自身状态的情况下，无法获知一个 channel 是否关闭。
2. 关闭一个 closed channel 会导致 panic。所以，如果关闭 channel 的一方在不知道 channel 是否处于关闭状态时就去贸然关闭 channel 是很危险的事情。
3. 向一个 closed channel 发送数据会导致 panic。所以，如果向 channel 发送数据的一方不知道 channel 是否处于关闭状态时就去贸然向 channel 发送数据是很危险的事情。

一个比较粗糙的检查 channel 是否关闭的函数：

```
func IsClosed(ch <-chan T) bool {
    select {
    case <-ch:
        return true
    default:
    }

    return false
}

func main() {
    c := make(chan T)
    fmt.Println(IsClosed(c)) // false
    close(c)
    fmt.Println(IsClosed(c)) // true
}
```

看一下代码，其实存在很多问题。首先，IsClosed 函数是一个有副作用的函数。每调用一次，都会读出 channel 里的一个元素，改变了 channel 的状态。这不是一个好的函数，干活就干活，还顺手牵羊！

其次，IsClosed 函数返回的结果仅代表调用那个瞬间，并不能保证调用之后会不会有其他 goroutine 对它进行了一些操作，改变了它的这种状态。例如，IsClosed 函数返回 true，但这时有另一个 goroutine 关闭了 channel，而你还拿着这个过时的“channel 未关闭”的信息，向其发送数据，就会导致 panic 的发生。当然，一个 channel 不会被重复关闭两次，如果 IsClosed 函数返回的结果是 true，说明 channel 是真的关闭了。

有一条广泛流传的关闭 channel 的原则：

> don’t close a channel from the receiver side and don’t close a channel if the channel has multiple concurrent senders.

不要从一个 receiver 侧关闭 channel，也不要在有多个 sender 时，关闭 channel。

比较好理解，向 channel 发送元素的就是 sender，因此 sender 可以决定何时不发送数据，并且关闭 channel。但是如果有多个 sender，某个 sender 同样没法确定其他 sender 的情况，这时也不能贸然关闭 channel。

但是上面所说的并不是最本质的，最本质的原则就只有一条：

> don’t close (or send values to) closed channels.

有两个不那么优雅地关闭 channel 的方法：

1. 使用 defer-recover 机制，放心大胆地关闭 channel 或者向 channel 发送数据。即使发生了 panic，有 defer-recover 在兜底。

2. 使用 sync.Once 来保证只关闭一次。

那到底应该如何优雅地关闭 channel？

根据 sender 和 receiver 的个数，分下面几种情况：

1. 一个 sender，一个 receiver
2. 一个 sender， M 个 receiver
3. N 个 sender，一个 reciver
4. N 个 sender， M 个 receiver

对于 1，2，只有一个 sender 的情况就不用说了，直接从 sender 端关闭就好了，没有问题。重点关注第 3，4 种情况。

第 3 种情形下，优雅关闭 channel 的方法是：the only receiver says “please stop sending more” by closing an additional signal channel。

解决方案就是增加一个传递关闭信号的 channel，receiver 通过信号 channel 下达关闭数据 channel 指令。senders 监听到关闭信号后，停止接收数据。代码如下：

```
func main() {
    rand.Seed(time.Now().UnixNano())

    const Max = 100000
    const NumSenders = 1000

    dataCh := make(chan int, 100)
    stopCh := make(chan struct{})

    // senders
    for i := 0; i < NumSenders; i++ {
        go func() {
            for {
                select {
                case <- stopCh:
                    return
                case dataCh <- rand.Intn(Max):
                }
            }
        }()
    }

    // the receiver
    go func() {
        for value := range dataCh {
            if value == Max-1 {
                fmt.Println("send stop signal to senders.")
                close(stopCh)
                return
            }

            fmt.Println(value)
        }
    }()

    select {
    case <- time.After(time.Hour):
    }
}
```

这里的 stopCh 就是信号 channel，它本身只有一个 sender，因此可以直接关闭它。senders 收到了关闭信号后，select 分支 “case <- stopCh” 被选中，退出函数，不再发送数据。

需要说明的是，上面的代码并没有明确关闭 dataCh。在 Go 语言中，对于一个 channel，如果最终没有任何 goroutine 引用它，不管 channel 有没有被关闭，最终都会被 gc 回收。所以，在这种情形下，所谓的优雅地关闭 channel 就是不关闭 channel，让 gc 代劳。

最后一种情况，优雅关闭 channel 的方法是：any one of them says “let’s end the game” by notifying a moderator to close an additional signal channel。

和第 3 种情况不同，这里有 M 个 receiver，如果直接还是采取第 3 种解决方案，由 receiver 直接关闭 stopCh 的话，就会重复关闭一个 channel，导致 panic。因此需要增加一个中间人，M 个 receiver 都向它发送关闭 dataCh 的“请求”，中间人收到第一个请求后，就会直接下达关闭 dataCh 的指令（通过关闭 stopCh，这时就不会发生重复关闭的情况，因为 stopCh 的发送方只有中间人一个）。另外，这里的 N 个 sender 也可以向中间人发送关闭 dataCh 的请求。

```
func main() {
    rand.Seed(time.Now().UnixNano())

    const Max = 100000
    const NumReceivers = 10
    const NumSenders = 1000

    dataCh := make(chan int, 100)
    stopCh := make(chan struct{})

    // It must be a buffered channel.
    toStop := make(chan string, 1)

    var stoppedBy string

    // moderator
    go func() {
        stoppedBy = <-toStop
        close(stopCh)
    }()

    // senders
    for i := 0; i < NumSenders; i++ {
        go func(id string) {
            for {
                value := rand.Intn(Max)
                if value == 0 {
                    select {
                    case toStop <- "sender#" + id:
                    default:
                    }
                    return
                }

                select {
                case <- stopCh:
                    return
                case dataCh <- value:
                }
            }
        }(strconv.Itoa(i))
    }

    // receivers
    for i := 0; i < NumReceivers; i++ {
        go func(id string) {
```

```
            for {
                select {
                case <- stopCh:
                    return
                case value := <-dataCh:
                    if value == Max-1 {
                        select {
                        case toStop <- "receiver#" + id:
                        default:
                        }
                        return
                    }

                    fmt.Println(value)
                }
            }
        }(strconv.Itoa(i))
    }

    select {
    case <- time.After(time.Hour):
    }

}
```

代码里 toStop 就是中间人的角色，使用它来接收 senders 和 receivers 发送过来的关闭 dataCh 请求。

这里将 toStop 声明成了一个 缓冲型的 channel。假设 toStop 声明的是一个非缓冲型的 channel，那么第一个发送的关闭 dataCh 请求可能会丢失。因为无论是 sender 还是 receiver 都是通过 select 语句来发送请求，如果中间人所在的 goroutine 没有准备好，那 select 语句就不会选中，直接走 default 选项，什么也不做。这样，第一个关闭 dataCh 的请求就会丢失。

如果，我们把 toStop 的容量声明成 Num(senders) + Num(receivers)，那发送 dataCh 请求的部分可以改成更简洁的形式：

```
...
toStop := make(chan string, NumReceivers + NumSenders)
...
            value := rand.Intn(Max)
            if value == 0 {
                toStop <- "sender#" + id
                return
            }
...
                if value == Max-1 {
                    toStop <- "receiver#" + id
                    return
                }
...
```

直接向 toStop 发送请求，因为 toStop 容量足够大，所以不用担心阻塞，自然也就不用 select 语句再加一个 default case 来避免阻塞。

可以看到，这里同样没有真正关闭 dataCh，原样同第 3 种情况。

以上，就是最基本的一些情形，但已经能覆盖几乎所有的情况及其变种了。只要记住：

> don’t close a channel from the receiver side and don’t close a channel if the channel has multiple concurrent senders.

以及更本质的原则：

> don’t close (or send values to) closed channels.

# 参考资料

https://go101.org/article/channel-closing.html

# channel 在什么情况下会引起资源泄漏

Channel 可能会引发 goroutine 泄漏。

泄漏的原因是 goroutine 操作 channel 后，处于发送或接收阻塞状态，而 channel 处于满或空的状态，一直得不到改变。同时，垃圾回收器也不会回收此类资源，进而导致 gouroutine 会一直处于等待队列中，不见天日。

另外，程序运行过程中，对于一个 channel，如果没有任何 goroutine 引用了，gc 会对其进行回收操作，不会引起内存泄漏。

# 什么是 CSP

> Do not communicate by sharing memory; instead, share memory by communicating.

不要通过共享内存来通信，而要通过通信来实现内存共享。

这就是 Go 的并发哲学，它依赖 CSP 模型，基于 channel 实现。

CSP 经常被认为是 Go 在并发编程上成功的关键因素。CSP 全称是 “Communicating Sequential Processes”，这也是 Tony Hoare 在 1978 年发表在 ACM 的一篇论文。论文里指出一门编程语言应该重视 input 和 output 的原语，尤其是并发编程的代码。

在那篇文章发表的时代，人们正在研究模块化编程的思想，该不该用 goto 语句在当时是最激烈的议题。彼时，面向对象编程的思想正在崛起，几乎没什么人关心并发编程。

在文章中，CSP 也是一门自定义的编程语言，作者定义了输入输出语句，用于 processes 间的通信（communicatiton）。processes 被认为是需要输入驱动，并且产生输出，供其他 processes 消费，processes 可以是进程、线程、甚至是代码块。输入命令是：!，用来向 processes 写入；输出是：?，用来从 processes 读出。这篇文章要讲的 channel 正是借鉴了这一设计。

Hoare 还提出了一个 -> 命令，如果 -> 左边的语句返回 false，那它右边的语句就不会执行。

通过这些输入输出命令，Hoare 证明了如果一门编程语言中把 processes 间的通信看得第一等重要，那么并发编程的问题就会变得简单。

Go 是第一个将 CSP 的这些思想引入，并且发扬光大的语言。仅管内存同步访问控制（原文是 memory access synchronization）在某些情况下大有用处，Go 里也有相应的 sync 包支持，但是这在大型程序很容易出错。

Go 一开始就把 CSP 的思想融入到语言的核心里，所以并发编程成为 Go 的一个独特的优势，而且很容易理解。

大多数的编程语言的并发编程模型是基于线程和内存同步访问控制，Go 的并发编程的模型则用 goroutine 和 channel 来替代。Goroutine 和线程类似，channel 和 mutex (用于内存同步访问控制)类似。

Goroutine 解放了程序员，让我们更能贴近业务去思考问题。而不用考虑各种像线程库、线程开销、线程调度等等这些繁琐的底层问题，goroutine 天生替你解决好了。

Channel 则天生就可以和其他 channel 组合。我们可以把收集各种子系统结果的 channel 输入到同一个 channel。Channel 还可以和 select, cancel, timeout 结合起来。而 mutex 就没有这些功能。

Go 的并发原则非常优秀，目标就是简单：尽量使用 channel；把 goroutine 当作免费的资源，随便用。

# channel 底层的数据结构是什么

## 数据结构

底层数据结构需要看源码，版本为 go 1.9.2：

```
type hchan struct {
    // chan 里元素数量
    qcount   uint
    // chan 底层循环数组的长度
    dataqsiz uint
    // 指向底层循环数组的指针
    // 只针对有缓冲的 channel
    buf      unsafe.Pointer
    // chan 中元素大小
    elemsize uint16
    // chan 是否被关闭的标志
    closed   uint32
    // chan 中元素类型
    elemtype *_type // element type
    // 已发送元素在循环数组中的索引
    sendx    uint   // send index
    // 已接收元素在循环数组中的索引
    recvx    uint   // receive index
    // 等待接收的 goroutine 队列
    recvq    waitq  // list of recv waiters
    // 等待发送的 goroutine 队列
    sendq    waitq  // list of send waiters

    // 保护 hchan 中所有字段
    lock mutex
}
```

关于字段的含义都写在注释里了，再来重点说几个字段：

`buf` 指向底层循环数组，只有缓冲型的 channel 才有。

`sendx`，`recvx` 均指向底层循环数组，表示当前可以发送和接收的元素位置索引值（相对于底层数组）。

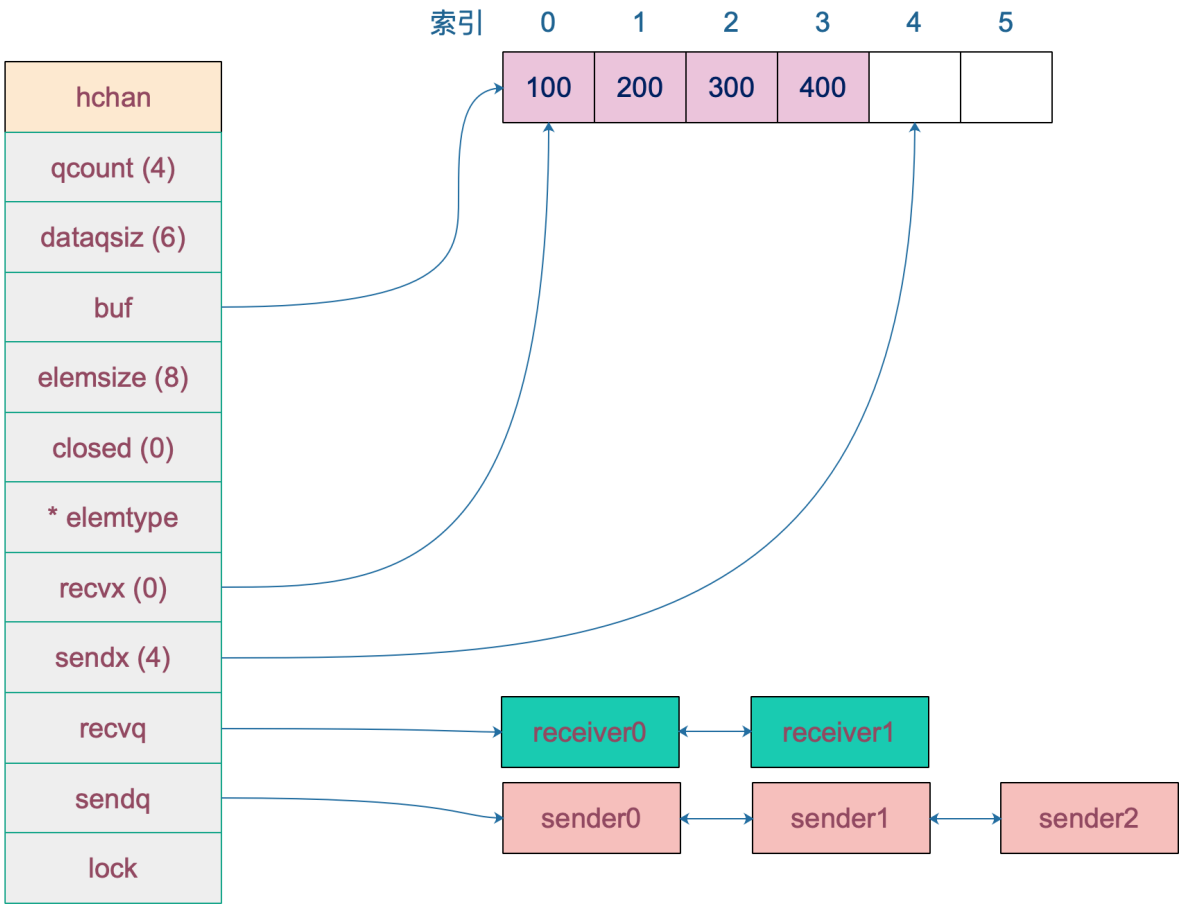
`sendq`，`recvq` 分别表示被阻塞的 goroutine，这些 goroutine 由于尝试读取 channel 或向 channel 发送数据而被阻塞。

`waitq` 是 `sudog` 的一个双向链表，而 `sudog` 实际上是对 goroutine 的一个封装：

```
type waitq struct {
    first *sudog
    last  *sudog
}
```

`lock` 用来保证每个读 channel 或写 channel 的操作都是原子的。

例如，创建一个容量为 6 的，元素为 int 型的 channel 数据结构如下 ：




# 创建

我们知道，通道有两个方向，发送和接收。理论上来说，我们可以创建一个只发送或只接收的通道，但是这种通道创建出来后，怎么使用呢？一个只能发的通道，怎么接收呢？同样，一个只能收的通道，如何向其发送数据呢？

一般而言，使用 `make` 创建一个能收能发的通道：

```
// 无缓冲通道
ch1 := make(chan int)
// 有缓冲通道
ch2 := make(chan int, 10)
```

通过汇编分析，我们知道，最终创建 chan 的函数是 `makechan` ：

```
func makechan(t *chantype, size int64) *hchan
```

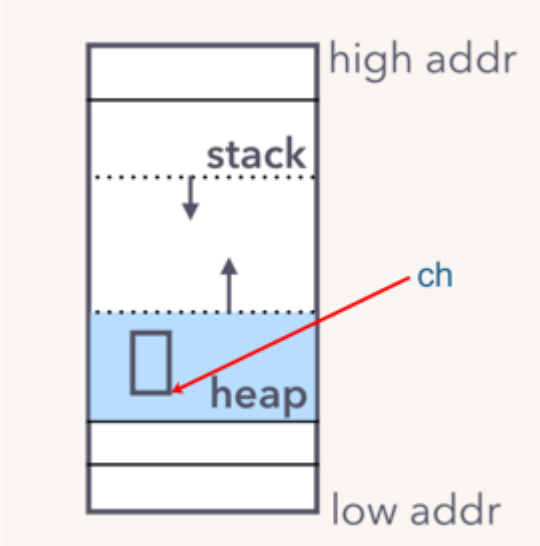
从函数原型来看，创建的 chan 是一个指针。所以我们能在函数间直接传递 channel，而不用传递 channel 的指针。

具体来看下代码：

```
const hchanSize = unsafe.Sizeof(hchan{}) + uintptr(-int(unsafe.Sizeof(hchan{}))&(maxAlign-1))

func makechan(t *chantype, size int64) *hchan {
    elem := t.elem
```

```
    // 省略了检查 channel size，align 的代码
    // ……

    var c *hchan
    // 如果元素类型不含指针 或者 size 大小为 0（无缓冲类型）
    // 只进行一次内存分配
    if elem.kind&kindNoPointers != 0 || size == 0 {
        // 如果 hchan 结构体中不含指针，GC 就不会扫描 chan 中的元素
        // 只分配 "hchan 结构体大小 + 元素大小*个数" 的内存
        c = (*hchan)(mallocgc(hchanSize+uintptr(size)*elem.size, nil, true))
        // 如果是缓冲型 channel 且元素大小不等于 0（大小等于 0的元素类型：struct{}）
        if size > 0 && elem.size != 0 {
            c.buf = add(unsafe.Pointer(c), hchanSize)
        } else {
            // race detector uses this location for synchronization
            // Also prevents us from pointing beyond the allocation (see issue 9401).
            // 1. 非缓冲型的，buf 没用，直接指向 chan 起始地址处
            // 2. 缓冲型的，能进入到这里，说明元素无指针且元素类型为 struct{}，也无影响
            // 因为只会用到接收和发送游标，不会真正拷贝东西到 c.buf 处（这会覆盖 chan的内容）
            c.buf = unsafe.Pointer(c)
        }
    } else {
        // 进行两次内存分配操作
        c = new(hchan)
        c.buf = newarray(elem, int(size))
    }
    c.elemsize = uint16(elem.size)
    c.elemtype = elem
    // 循环数组长度
    c.dataqsiz = uint(size)

    // 返回 hchan 指针
    return c
}
```

新建一个 chan 后，内存在堆上分配，大概长这样：




# 参考资料

【Kavya在Gopher Con 上关于 channel 的设计，非常好】
https://speakerd.s3.amazonaws.com/presentations/10ac0b1d76a6463aa98ad6a9dec917a7/GopherCon_v10.0.pdf

# channel 发送和接收元素的本质是什么

> All transfer of value on the go channels happens with the copy of value.

就是说 channel 的发送和接收操作本质上都是“值的拷贝”，无论是从 sender goroutine 的栈到 chan buf，还是从 chan buf 到 receiver goroutine，或者是直接从 sender goroutine 到 receiver goroutine。

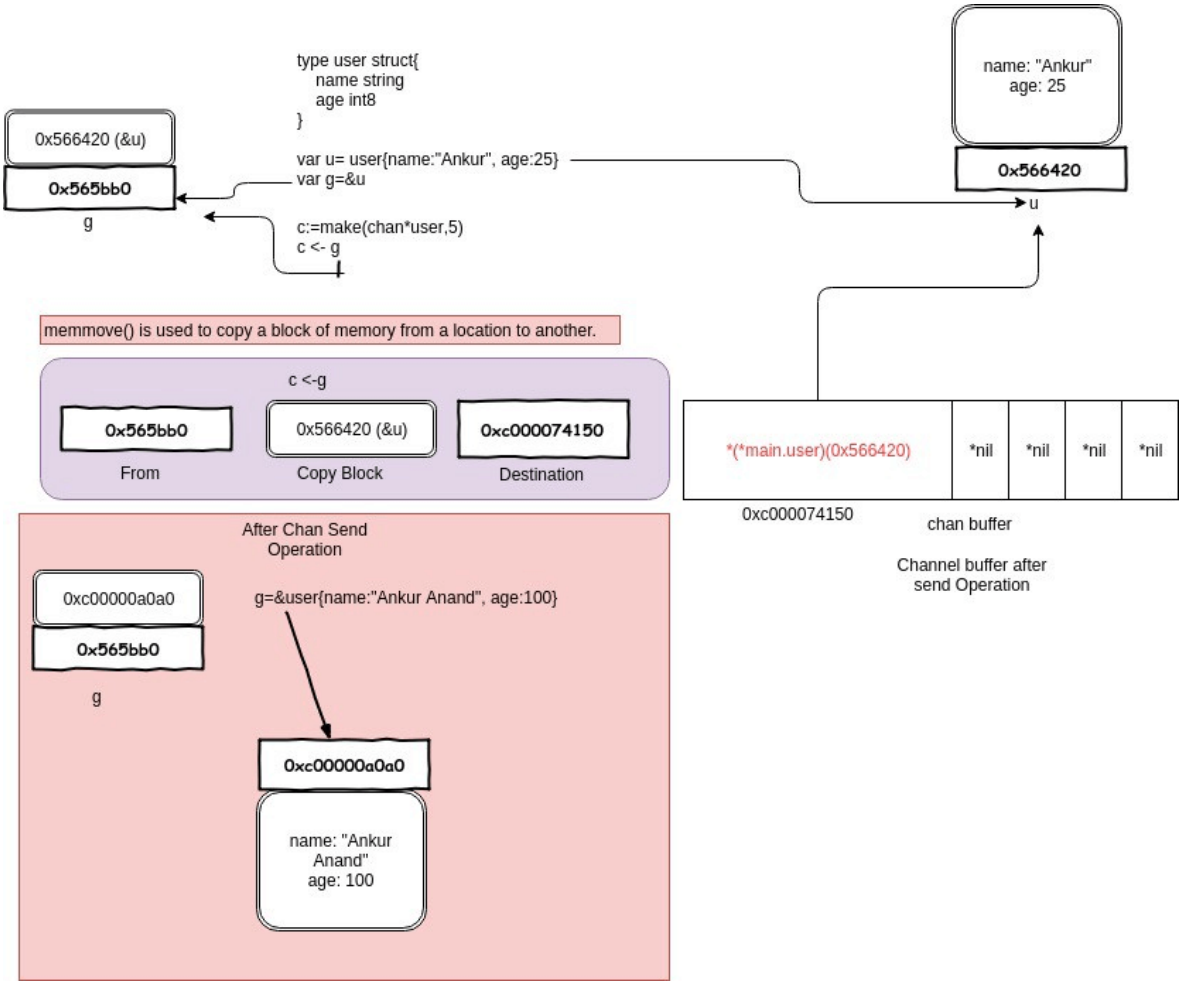
举一个例子：

```
type user struct {
    name string
    age int8
}

var u = user{name: "Ankur", age: 25}
var g = &u

func modifyUser(pu *user) {
    fmt.Println("modifyUser Received Vaule", pu)
    pu.name = "Anand"
}

func printUser(u <-chan *user) {
    time.Sleep(2 * time.Second)
    fmt.Println("printUser goRoutine called", <-u)
}

func main() {
    c := make(chan *user, 5)
    c <- g
    fmt.Println(g)
    // modify g
    g = &user{name: "Ankur Anand", age: 100}
    go printUser(c)
    go modifyUser(g)
    time.Sleep(5 * time.Second)
    fmt.Println(g)
}
```

运行结果：

```
&{Ankur 25}
modifyUser Received Vaule &{Ankur Anand 100}
printUser goRoutine called &{Ankur 25}
&{Anand 100}
```

这里就是一个很好的 `share memory by communicating` 的例子。




一开始构造一个结构体 u，地址是 0x56420，图中地址上方就是它的内容。接着把 `&u` 赋值给指针 `g`，g 的地址是 0x565bb0，它的内容就是一个地址，指向 u。

main 程序里，先把 g 发送到 c，根据 `copy value` 的本质，进入到 chan buf 里的就是 `0x56420`，它是指针 g 的值（不是它指向的内容），所以打印从 channel 接收到的元素时，它就是 `&{Ankur 25}`。因此，这里并不是将指针 g “发送” 到了 channel 里，只是拷贝它的值而已。

再强调一次：

> Remember all transfer of value on the go channels happens with the copy of value.

# 参考资料

【深入 channel 底层】https://codeburst.io/diving-deep-into-the-golang-channels-549fd4ed21a8

# 关于 channel 的 happened-before 有哪些

维基百科上给的定义：

> In computer science, the happened-before relation (denoted: ->) is a relation between the result of two events, such that if one event should happen before another event, the result must reflect that, even if those events are in reality executed out of order (usually to optimize program flow).

简单来说就是如果事件 a 和事件 b 存在 happened-before 关系，即 a -> b，那么 a，b 完成后的结果一定要体现这种关系。由于现代编译器、CPU 会做各种优化，包括编译器重排、内存重排等等，在并发代码里，happened-before 限制就非常重要了。

根据晃岳攀老师在 Gopher China 2019 上的并发编程分享，关于 channel 的发送（send）、发送完成（send finished）、接收（receive）、接收完成（receive finished）的 happened-before 关系如下：

1. 第 n 个 `send` 一定 `happened before` 第 n 个 `receive finished`，无论是缓冲型还是非缓冲型的 channel。
2. 对于容量为 m 的缓冲型 channel，第 n 个 `receive` 一定 `happened before` 第 n+m 个 `send finished`。
3. 对于非缓冲型的 channel，第 n 个 `receive` 一定 `happened before` 第 n 个 `send finished`。
4. channel close 一定 `happened before` receiver 得到通知。

我们来逐条解释一下。

第一条，我们从源码的角度看也是对的，send 不一定是 `happened before` receive，因为有时候是先 receive，然后 goroutine 被挂起，之后被 sender 唤醒，send happened after receive。但不管怎样，要想完成接收，一定是要先有发送。

第二条，缓冲型的 channel，当第 n+m 个 send 发生后，有下面两种情况：

若第 n 个 receive 没发生。这时，channel 被填满了，send 就会被阻塞。那当第 n 个 receive 发生时，sender goroutine 会被唤醒，之后再继续发送过程。这样，第 n 个 `receive` 一定 `happened before` 第 n+m 个 `send finished`。

若第 n 个 receive 已经发生过了，这直接就符合了要求。

第三条，也是比较好理解的。第 n 个 send 如果被阻塞，sender goroutine 挂起，第 n 个 receive 这时到来，先于第 n 个 send finished。如果第 n 个 send 未被阻塞，说明第 n 个 receive 早就在那等着了，它不仅 happened before send finished，它还 happened before send。

第四条，回忆一下源码，先设置完 closed = 1，再唤醒等待的 receiver，并将零值拷贝给 receiver。

参考资料【鸟窝 并发编程分享】这篇博文的评论区有 PPT 的下载链接，这是晁老师在 Gopher 2019 大会上的演讲。

关于 happened before，这里再介绍一个柴大和曹大的新书《Go 语言高级编程》里面提到的一个例子。

书中 1.5 节先讲了顺序一致性的内存模型，这是并发编程的基础。

我们直接来看例子：

```
var done = make(chan bool)
var msg string

func aGoroutine() {
    msg = "hello, world"
    done <- true
}
```

```
func main() {
    go aGoroutine()
    <-done
    println(msg)
}
```

先定义了一个 done channel 和一个待打印的字符串。在 main 函数里，启动一个 goroutine，等待从 done 里接收到一个值后，执行打印 msg 的操作。如果 main 函数中没有 `<-done` 这行代码，打印出来的 msg 为空，因为 aGoroutine 来不及被调度，还来不及给 msg 赋值，主程序就会退出。而在 Go 语言里，主协程退出时不会等待其他协程。

加了 `<-done` 这行代码后，就会阻塞在此。等 aGoroutine 里向 done 发送了一个值之后，才会被唤醒，继续执行打印 msg 的操作。而这在之前，msg 已经被赋值过了，所以会打印出 `hello, world` 。

这里依赖的 happened before 就是前面讲的第一条。第一个 send 一定 happened before 第一个 receive finished，即 `done <- true` 先于 `<-done` 发生，这意味着 main 函数里执行完 `<-done` 后接着执行 `println(msg)` 这一行代码时，msg 已经被赋过值了，所以会打印出想要的结果。

进一步利用前面提到的第 3 条 happened before 规则，修改一下代码：

```
var done = make(chan bool)
var msg string

func aGoroutine() {
    msg = "hello, world"
    <-done
}

func main() {
    go aGoroutine()
    done <- true
    println(msg)
}
```

同样可以得到相同的结果，为什么？根据第三条规则，对于非缓冲型的 channel，第一个 receive 一定 happened before 第一个 send finished。也就是说，
在 `done <- true` 完成之前， `<-done` 就已经发生了，也就意味着 msg 已经被赋上值了，最终也会打印出 `hello, world` 。

# 向 channel 发送数据的过程是怎样的

## 源码分析

发送操作最终转化为 `chansend` 函数，直接上源码，同样大部分都注释了，可以看懂主流程：

```
// 位于 src/runtime/chan.go

func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
    // 如果 channel 是 nil
    if c == nil {
        // 不能阻塞，直接返回 false，表示未发送成功
        if !block {
            return false
        }
        // 当前 goroutine 被挂起
        gopark(nil, nil, "chan send (nil chan)", traceEvGoStop, 2)
        throw("unreachable")
    }

    // 省略 debug 相关……

    // 对于不阻塞的 send，快速检测失败场景
    //
    // 如果 channel 未关闭且 channel 没有多余的缓冲空间。这可能是：
    // 1. channel 是非缓冲型的，且等待接收队列里没有 goroutine
    // 2. channel 是缓冲型的，但循环数组已经装满了元素
    if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == nil) ||
        (c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {
        return false
    }

    var t0 int64
    if blockprofilerate > 0 {
        t0 = cputicks()
    }

    // 锁住 channel，并发安全
    lock(&c.lock)

    // 如果 channel 关闭了
    if c.closed != 0 {
        // 解锁
        unlock(&c.lock)
        // 直接 panic
        panic(plainError("send on closed channel"))
    }

    // 如果接收队列里有 goroutine，直接将要发送的数据拷贝到接收 goroutine
    if sg := c.recvq.dequeue(); sg != nil {
        send(c, sg, ep, func() { unlock(&c.lock) }, 3)
        return true
    }

    // 对于缓冲型的 channel，如果还有缓冲空间
    if c.qcount < c.dataqsiz {
```

```
        // qp 指向 buf 的 sendx 位置
        qp := chanbuf(c, c.sendx)

        // ……

        // 将数据从 ep 处拷贝到 qp
        typedmemmove(c.elemtype, qp, ep)
        // 发送游标值加 1
        c.sendx++
        // 如果发送游标值等于容量值，游标值归 0
        if c.sendx == c.dataqsiz {
            c.sendx = 0
        }
        // 缓冲区的元素数量加一
        c.qcount++

        // 解锁
        unlock(&c.lock)
        return true
    }

    // 如果不需要阻塞，则直接返回错误
    if !block {
        unlock(&c.lock)
        return false
    }

    // channel 满了，发送方会被阻塞。接下来会构造一个 sudog

    // 获取当前 goroutine 的指针
    gp := getg()
    mysg := acquireSudog()
    mysg.releasetime = 0
    if t0 != 0 {
        mysg.releasetime = -1
    }

    mysg.elem = ep
    mysg.waitlink = nil
    mysg.g = gp
    mysg.selectdone = nil
    mysg.c = c
    gp.waiting = mysg
    gp.param = nil

    // 当前 goroutine 进入发送等待队列
    c.sendq.enqueue(mysg)

    // 当前 goroutine 被挂起
    goparkunlock(&c.lock, "chan send", traceEvGoBlockSend, 3)

    // 从这里开始被唤醒了（channel 有机会可以发送了）
    if mysg != gp.waiting {
        throw("G waiting list is corrupted")
    }
    gp.waiting = nil
    if gp.param == nil {
        if c.closed == 0 {
            throw("chansend: spurious wakeup")
        }
        // 被唤醒后，channel 关闭了。坑爹啊，panic
        panic(plainError("send on closed channel"))
```

```
    }
    gp.param = nil
    if mysg.releasetime > 0 {
        blockevent(mysg.releasetime-t0, 2)
    }
    // 去掉 mysg 上绑定的 channel
    mysg.c = nil
    releaseSudog(mysg)
    return true
}
```

上面的代码注释地比较详细了，我们来详细看看。

- 如果检测到 channel 是空的，当前 goroutine 会被挂起。
- 对于不阻塞的发送操作，如果 channel 未关闭并且没有多余的缓冲空间（说明：a. channel 是非缓冲型的，且等待接收队列里没有 goroutine；b. channel 是缓冲型的，但循环数组已经装满了元素）

对于这一点，runtime 源码里注释了很多。这一条判断语句是为了在不阻塞发送的场景下快速检测到发送失败，好快速返回。

```
if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == nil) || (c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {
    return false
}
```

注释里主要讲为什么这一块可以不加锁，我详细解释一下。 `if` 条件里先读了两个变量：block 和 c.closed。block 是函数的参数，不会变；c.closed 可能被其他 goroutine 改变，因为没加锁嘛，这是“与”条件前面两个表达式。

最后一项，涉及到三个变量：c.dataqsiz，c.recvq.first，c.qcount。 `c.dataqsiz == 0 && c.recvq.first == nil` 指的是非缓冲型的 channel，并且 recvq 里没有等待接收的 goroutine； `c.dataqsiz > 0 && c.qcount == c.dataqsiz` 指的是缓冲型的 channel，但循环数组已经满了。这里 `c.dataqsiz` 实际上也是不会被修改的，在创建的时候就已经确定了。不加锁真正影响地是 `c.qcount` 和 `c.recvq.first` 。

这一部分的条件就是两个 `word-sized read` ，就是读两个 word 操作： `c.closed` 和 `c.recvq.first` （非缓冲型） 或者 `c.qcount` （缓冲型）。

当我们发现 `c.closed == 0` 为真，也就是 channel 未被关闭，再去检测第三部分的条件时，观测到 `c.recvq.first == nil` 或者 `c.qcount == c.dataqsiz` 时（这里忽略 `c.dataqsiz` ），就断定要将这次发送操作作失败处理，快速返回 false。

这里涉及到两个观测项：channel 未关闭、channel not ready for sending。这两项都会因为没加锁而出现观测前后不一致的情况。例如我先观测到 channel 未被关闭，再观察到 channel not ready for sending，这时我以为能满足这个 if 条件了，但是如果这时 c.closed 变成 1，这时其实就不满足条件了，谁让你不加锁呢！

但是，因为一个 closed channel 不能将 channel 状态从 ‘ready for sending’ 变成 ‘not ready for sending’，所以当我观测到 ‘not ready for sending’ 时，channel 不是 closed。即使 `c.closed == 1` ，即 channel 是在这两个观测中间被关闭的，那也说明在这两个观测中间，channel 满足两个条件： `not closed` 和 `not ready for sending` ，这时，我直接返回 false 也是没有问题的。

这部分解释地比较绕，其实这样做的目的就是少获取一次锁，提升性能。

- 如果检测到 channel 已经关闭，直接 panic。
- 如果能从等待接收队列 recvq 里出队一个 sudog（代表一个 goroutine），说明此时 channel 是空的，没有元素，所以才会有等待接收者。这时会调用 send 函数将元素直接从发送者的栈拷贝到接收者的栈，关键操作由 `sendDirect` 函数完成。

```
// send 函数处理向一个空的 channel 发送操作

// ep 指向被发送的元素，会被直接拷贝到接收的 goroutine
// 之后，接收的 goroutine 会被唤醒
// c 必须是空的（因为等待队列里有 goroutine，肯定是空的）
// c 必须被上锁，发送操作执行完后，会使用 unlockf 函数解锁
// sg 必须已经从等待队列里取出来了
// ep 必须是非空，并且它指向堆或调用者的栈

func send(c *hchan, sg *sudog, ep unsafe.Pointer, unlockf func(), skip int) {
    // 省略一些用不到的
    // ......

    // sg.elem 指向接收到的值存放的位置，如 val <- ch，指的就是 &val
    if sg.elem != nil {
        // 直接拷贝内存（从发送者到接收者）
        sendDirect(c.elemtype, sg, ep)
        sg.elem = nil
    }
    // sudog 上绑定的 goroutine
    gp := sg.g
    // 解锁
    unlockf()
    gp.param = unsafe.Pointer(sg)
    if sg.releasetime != 0 {
        sg.releasetime = cputicks()
    }
    // 唤醒接收的 goroutine. skip 和打印栈相关，暂时不理会
    goready(gp, skip+1)
}
```

继续看 `sendDirect` 函数：

```
// 向一个非缓冲型的 channel 发送数据、从一个无元素的（非缓冲型或缓冲型但空）的 channel
// 接收数据，都会导致一个 goroutine 直接操作另一个 goroutine 的栈
// 由于 GC 假设对栈的写操作只能发生在 goroutine 正在运行中并且由当前 goroutine 来写
// 所以这里实际上违反了这个假设。可能会造成一些问题，所以需要用到写屏障来规避
func sendDirect(t *_type, sg *sudog, src unsafe.Pointer) {
    // src 在当前 goroutine 的栈上，dst 是另一个 goroutine 的栈

    // 直接进行内存"搬迁"
    // 如果目标地址的栈发生了栈收缩，当我们读出了 sg.elem 后
    // 就不能修改真正的 dst 位置的值了
    // 因此需要在读和写之前加上一个屏障
    dst := sg.elem
    typeBitsBulkBarrier(t, uintptr(dst), uintptr(src), t.size)
    memmove(dst, src, t.size)
}
```

这里涉及到一个 goroutine 直接写另一个 goroutine 栈的操作，一般而言，不同 goroutine 的栈是各自独有的。而这也违反了 GC 的一些假设。为了不出问题，写的过程中增加了写屏障，保证正确地完成写操作。这样做的好处是减少了一次内存 copy：不用先拷贝到 channel 的 buf，直接由发送者到接收者，没有中间商赚差价，效率得以提高，完美。

然后，解锁、唤醒接收者，等待调度器的光临，接收者也得以重见天日，可以继续执行接收操作之后的代码了。

- 如果 `c.qcount < c.dataqsiz`，说明缓冲区可用（肯定是缓冲型的 channel）。先通过函数取出待发送元素应该去到的位置：

```
qp := chanbuf(c, c.sendx)

// 返回循环队列里第 i 个元素的地址处
func chanbuf(c *hchan, i uint) unsafe.Pointer {
    return add(c.buf, uintptr(i)*uintptr(c.elemsize))
}
```

`c.sendx` 指向下一个待发送元素在循环数组中的位置，然后调用 `typedmemmove` 函数将其拷贝到循环数组中。之后 `c.sendx` 加 1，元素总量加 1： `c.qcount++` ，最后，解锁并返回。

- 如果没有命中以上条件的，说明 channel 已经满了。不管这个 channel 是缓冲型的还是非缓冲型的，都要将这个 sender “关起来”（goroutine 被阻塞）。如果 block 为 false，直接解锁，返回 false。

- 最后就是真的需要被阻塞的情况。先构造一个 sudog，将其入队（channel 的 sendq 字段）。然后调用 `goparkunlock` 将当前 goroutine 挂起，并解锁，等待合适的时机再唤醒。

唤醒之后，从 `goparkunlock` 下一行代码开始继续往下执行。

这里有一些绑定操作，sudog 通过 g 字段绑定 goroutine，而 goroutine 通过 waiting 绑定 sudog，sudog 还通过 `elem` 字段绑定待发送元素的地址，以及 `c` 字段绑定被“坑”在此处的 channel。

所以，待发送的元素地址其实是存储在 sudog 结构体里，也就是当前 goroutine 里。

# 案例分析

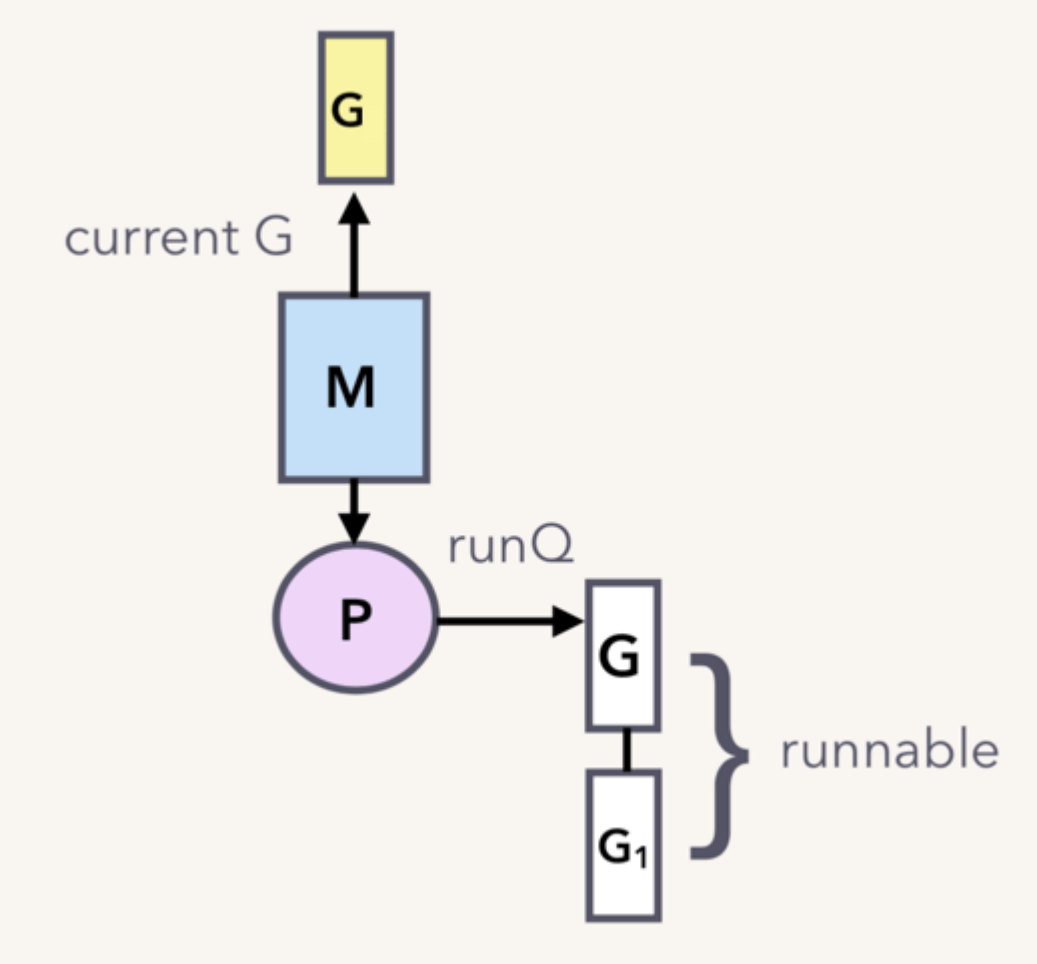
好了，看完源码。我们接着来分析例子，代码如下：

```
func goroutineA(a <-chan int) {
    val := <- a
    fmt.Println("goroutine A received data: ", val)
    return
}

func goroutineB(b <-chan int) {
    val := <- b
    fmt.Println("goroutine B received data: ", val)
    return
}

func main() {
    ch := make(chan int)
    go goroutineA(ch)
    go goroutineB(ch)
    ch <- 3
    time.Sleep(time.Second)

    ch1 := make(chan struct{})
}
```

在发送小节里我们说到 G1 和 G2 现在被挂起来了，等待 sender 的解救。在第 17 行，主协程向 ch 发送了一个元素 3，来看下接下来会发生什么。

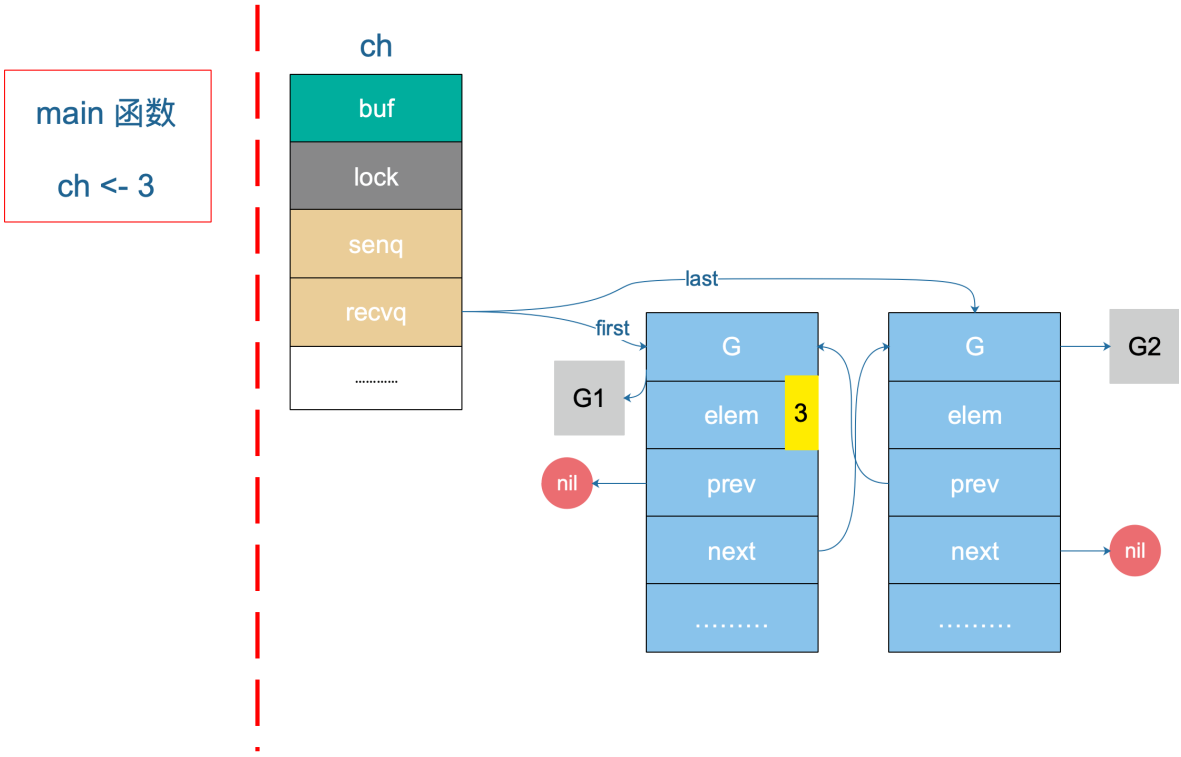
根据前面源码分析的结果，我们知道，sender 发现 ch 的 recvq 里有 receiver 在等待着接收，就会出队一个 sudog，把 recvq 里 first 指针的 sudo “推举”出来了，并将其加入到 P 的可运行 goroutine 队列中。

然后，sender 把发送元素拷贝到 sudog 的 elem 地址处，最后会调用 goready 将 G1 唤醒，状态变为 runnable。




当调度器光顾 G1 时，将 G1 变成 running 状态，执行 goroutineA 接下来的代码。G 表示其他可能有的 goroutine。

这里其实涉及到一个协程写另一个协程栈的操作。有两个 receiver 在 channel 的一边虎视眈眈地等着，这时 channel 另一边来了一个 sender 准备向 channel 发送数据，为了高效，用不着通过 channel 的 buf “中转”一次，直接从源地址把数据 copy 到目的地址就可以了，效率高啊！




上图是一个示意图，`3` 会被拷贝到 G1 栈上的某个位置，也就是 val 的地址处，保存在 elem 字段。

# 参考资料

【深入 channel 底层】https://codeburst.io/diving-deep-into-the-golang-channels-549fd4ed21a8

【Kavya在Gopher Con 上关于 channel 的设计，非常好】
https://speakerd.s3.amazonaws.com/presentations/10ac0b1d76a6463aa98ad6a9dec917a7/GopherCon_v10.0.pdf

# 从 channel 接收数据的过程是怎样的

## 源码分析

我们先来看一下接收相关的源码。在清楚了接收的具体过程之后，再根据一个实际的例子来具体研究。

接收操作有两种写法，一种带 “ok”，反应 channel 是否关闭；一种不带 “ok”，这种写法，当接收到相应类型的零值时无法知道是真实的发送者发送过来的值，还是 channel 被关闭后，返回给接收者的默认类型的零值。两种写法，都有各自的应用场景。

经过编译器的处理后，这两种写法最后对应源码里的这两个函数：

```
// entry points for <- c from compiled code
func chanrecv1(c *hchan, elem unsafe.Pointer) {
    chanrecv(c, elem, true)
}

func chanrecv2(c *hchan, elem unsafe.Pointer) (received bool) {
    _, received = chanrecv(c, elem, true)
    return
}
```

`chanrecv1` 函数处理不带 “ok” 的情形，`chanrecv2` 则通过返回 “received” 这个字段来反应 channel 是否被关闭。接收值则比较特殊，会“放到”参数 `elem` 所指向的地址了，这很像 C/C++ 里的写法。如果代码里忽略了接收值，这里的 elem 为 nil。

无论如何，最终转向了 `chanrecv` 函数：

```
// 位于 src/runtime/chan.go

// chanrecv 函数接收 channel c 的元素并将其写入 ep 所指向的内存地址。
// 如果 ep 是 nil，说明忽略了接收值。
// 如果 block == false，即非阻塞型接收，在没有数据可接收的情况下，返回 (false, false)
// 否则，如果 c 处于关闭状态，将 ep 指向的地址清零，返回 (true, false)
// 否则，用返回值填充 ep 指向的内存地址。返回 (true, true)
// 如果 ep 非空，则应该指向堆或者函数调用者的栈

func chanrecv(c *hchan, ep unsafe.Pointer, block bool) (selected, received bool) {
    // 省略 debug 内容 ……

    // 如果是一个 nil 的 channel
    if c == nil {
        // 如果不阻塞，直接返回 (false, false)
        if !block {
            return
        }
        // 否则，接收一个 nil 的 channel，goroutine 挂起
        gopark(nil, nil, "chan receive (nil chan)", traceEvGoStop, 2)
        // 不会执行到这里
        throw("unreachable")
    }

    // 在非阻塞模式下，快速检测到失败，不用获取锁，快速返回
    // 当我们观察到 channel 没准备好接收：
    // 1. 非缓冲型，等待发送列队 sendq 里没有 goroutine 在等待
```

```
    // 2. 缓冲型，但 buf 里没有元素
    // 之后，又观察到 closed == 0，即 channel 未关闭。
    // 因为 channel 不可能被重复打开，所以前一个观测的时候 channel 也是未关闭的，
    // 因此在这种情况下可以直接宣布接收失败，返回 (false, false)
    if !block && (c.dataqsiz == 0 && c.sendq.first == nil ||
        c.dataqsiz > 0 && atomic.Loaduint(&c.qcount) == 0) &&
        atomic.Load(&c.closed) == 0 {
        return
    }

    var t0 int64
    if blockprofilerate > 0 {
        t0 = cputicks()
    }

    // 加锁
    lock(&c.lock)

    // channel 已关闭，并且循环数组 buf 里没有元素
    // 这里可以处理非缓冲型关闭 和 缓冲型关闭但 buf 无元素的情况
    // 也就是说即使是关闭状态，但在缓冲型的 channel，
    // buf 里有元素的情况下还能接收到元素
    if c.closed != 0 && c.qcount == 0 {
        if raceenabled {
            raceacquire(unsafe.Pointer(c))
        }
        // 解锁
        unlock(&c.lock)
        if ep != nil {
            // 从一个已关闭的 channel 执行接收操作，且未忽略返回值
            // 那么接收的值将是一个该类型的零值
            // typedmemclr 根据类型清理相应地址的内存
            typedmemclr(c.elemtype, ep)
        }
        // 从一个已关闭的 channel 接收，selected 会返回true
        return true, false
    }

    // 等待发送队列里有 goroutine 存在，说明 buf 是满的
    // 这有可能是：
    // 1. 非缓冲型的 channel
    // 2. 缓冲型的 channel，但 buf 满了
    // 针对 1，直接进行内存拷贝（从 sender goroutine -> receiver goroutine）
    // 针对 2，接收到循环数组头部的元素，并将发送者的元素放到循环数组尾部
    if sg := c.sendq.dequeue(); sg != nil {
        // Found a waiting sender. If buffer is size 0, receive value
        // directly from sender. Otherwise, receive from head of queue
        // and add sender's value to the tail of the queue (both map to
        // the same buffer slot because the queue is full).
        recv(c, sg, ep, func() { unlock(&c.lock) }, 3)
        return true, true
    }

    // 缓冲型，buf 里有元素，可以正常接收
    if c.qcount > 0 {
        // 直接从循环数组里找到要接收的元素
        qp := chanbuf(c, c.recvx)

        // ...........

        // 代码里，没有忽略要接收的值，不是 "<- ch"，而是 "val <- ch"，ep 指向 val
        if ep != nil {
```

```
            typedmemmove(c.elemtype, ep, qp)
        }
        // 清理掉循环数组里相应位置的值
        typedmemclr(c.elemtype, qp)
        // 接收游标向前移动
        c.recvx++
        // 接收游标归零
        if c.recvx == c.dataqsiz {
            c.recvx = 0
        }
        // buf 数组里的元素个数减 1
        c.qcount--
        // 解锁
        unlock(&c.lock)
        return true, true
    }

    if !block {
        // 非阻塞接收，解锁。selected 返回 false，因为没有接收到值
        unlock(&c.lock)
        return false, false
    }

    // 接下来就是要被阻塞的情况了
    // 构造一个 sudog
    gp := getg()
    mysg := acquireSudog()
    mysg.releasetime = 0
    if t0 != 0 {
        mysg.releasetime = -1
    }

    // 待接收数据的地址保存下来
    mysg.elem = ep
    mysg.waitlink = nil
    gp.waiting = mysg
    mysg.g = gp
    mysg.selectdone = nil
    mysg.c = c
    gp.param = nil
    // 进入channel 的等待接收队列
    c.recvq.enqueue(mysg)
    // 将当前 goroutine 挂起
    goparkunlock(&c.lock, "chan receive", traceEvGoBlockRecv, 3)

    // 被唤醒了，接着从这里继续执行一些扫尾工作
    if mysg != gp.waiting {
        throw("G waiting list is corrupted")
    }
    gp.waiting = nil
    if mysg.releasetime > 0 {
        blockevent(mysg.releasetime-t0, 2)
    }
    closed := gp.param == nil
    gp.param = nil
    mysg.c = nil
    releaseSudog(mysg)
    return true, !closed
}
```

上面的代码注释地比较详细了，你可以对着源码一行行地去看，我们再来详细看一下。

- 如果 channel 是一个空值（nil），在非阻塞模式下，会直接返回。在阻塞模式下，会调用 gopark 函数挂起 goroutine，这个会一直阻塞下去。因为在 channel 是 nil 的情况下，要想不阻塞，只有关闭它，但关闭一个 nil 的 channel 又会发生 panic，所以没有机会被唤醒了。更详细地可以在 closechan 函数的时候再看。

- 和发送函数一样，接下来搞了一个在非阻塞模式下，不用获取锁，快速检测到失败并且返回的操作。顺带插一句，我们平时在写代码的时候，找到一些边界条件，快速返回，能让代码逻辑更清晰，因为接下来的正常情况就比较少，更聚焦了，看代码的人也更能专注地看核心代码逻辑了。

```
    // 在非阻塞模式下，快速检测到失败，不用获取锁，快速返回 (false, false)
    if !block && (c.dataqsiz == 0 && c.sendq.first == nil ||
        c.dataqsiz > 0 && atomic.Loaduint(&c.qcount) == 0) &&
        atomic.Load(&c.closed) == 0 {
        return
    }
```

当我们观察到 channel 没准备好接收：

1. 非缓冲型，等待发送列队里没有 goroutine 在等待
2. 缓冲型，但 buf 里没有元素

之后，又观察到 closed == 0，即 channel 未关闭。

因为 channel 不可能被重复打开，所以前一个观测的时候， channel 也是未关闭的，因此在这种情况下可以直接宣布接收失败，快速返回。因为没被选中，也没接收到数据，所以返回值为 (false, false)。

- 接下来的操作，首先会上一把锁，粒度比较大。如果 channel 已关闭，并且循环数组 buf 里没有元素。对应非缓冲型关闭和缓冲型关闭但 buf 无元素的情况，返回对应类型的零值，但 received 标识是 false，告诉调用者此 channel 已关闭，你取出来的值并不是正常由发送者发送过来的数据。但是如果处于 select 语境下，这种情况是被选中了的。很多将 channel 用作通知信号的场景就是命中了这里。

- 接下来，如果有等待发送的队列，说明 channel 已经满了，要么是非缓冲型的 channel，要么是缓冲型的 channel，但 buf 满了。这两种情况下都可以正常接收数据。

于是，调用 recv 函数：

```
func recv(c *hchan, sg *sudog, ep unsafe.Pointer, unlockf func(), skip int) {
    // 如果是非缓冲型的 channel
    if c.dataqsiz == 0 {
        if raceenabled {
            racesync(c, sg)
        }
        // 未忽略接收的数据
        if ep != nil {
            // 直接拷贝数据，从 sender goroutine -> receiver goroutine
            recvDirect(c.elemtype, sg, ep)
        }
    } else {
        // 缓冲型的 channel，但 buf 已满。
        // 将循环数组 buf 队首的元素拷贝到接收数据的地址
        // 将发送者的数据入队。实际上这时 revx 和 sendx 值相等
        // 找到接收游标
        qp := chanbuf(c, c.recvx)
        // ...........
        // 将接收游标处的数据拷贝给接收者
```

```
        if ep != nil {
            typedmemmove(c.elemtype, ep, qp)
        }

        // 将发送者数据拷贝到 buf
        typedmemmove(c.elemtype, qp, sg.elem)
        // 更新游标值
        c.recvx++
        if c.recvx == c.dataqsiz {
            c.recvx = 0
        }
        c.sendx = c.recvx
    }
    sg.elem = nil
    gp := sg.g

    // 解锁
    unlockf()
    gp.param = unsafe.Pointer(sg)
    if sg.releasetime != 0 {
        sg.releasetime = cputicks()
    }

    // 唤醒发送的 goroutine。需要等到调度器的光临
    goready(gp, skip+1)
}
```

如果是非缓冲型的，就直接从发送者的栈拷贝到接收者的栈。

```
func recvDirect(t *_type, sg *sudog, dst unsafe.Pointer) {
    // dst is on our stack or the heap, src is on another stack.
    src := sg.elem
    typeBitsBulkBarrier(t, uintptr(dst), uintptr(src), t.size)
    memmove(dst, src, t.size)
}
```

否则，就是缓冲型 channel，而 buf 又满了的情形。说明发送游标和接收游标重合了，因此需要先找到接收游标：

```
// chanbuf(c, i) is pointer to the i'th slot in the buffer.
func chanbuf(c *hchan, i uint) unsafe.Pointer {
    return add(c.buf, uintptr(i)*uintptr(c.elemsize))
}
```

将该处的元素拷贝到接收地址。然后将发送者待发送的数据拷贝到接收游标处。这样就完成了接收数据和发送数据的操作。接着，分别将发送游标和接收游标向前进一，如果发生“环绕”，再从 0 开始。

最后，取出 sudog 里的 goroutine，调用 goready 将其状态改成 “runnable”，待发送者被唤醒，等待调度器的调度。

- 然后，如果 channel 的 buf 里还有数据，说明可以比较正常地接收。注意，这里，即使是在 channel 已经关闭的情况下，也是可以走到这里的。这一步比较简单，正常地将 buf 里接收游标处的数据拷贝到接收数据的地址。

- 到了最后一步，走到这里来的情形是要阻塞的。当然，如果 block 传进来的值是 false，那就不阻塞，直接返回就好了。

先构造一个 sudog，接着就是保存各种值了。注意，这里会将接收数据的地址存储到了 `elem` 字段，当被唤醒时，接收到的数据就会保存到这个字段指向的地址。然后将 sudog 添加到 channel 的 recvq 队列里。调用 goparkunlock 函数将 goroutine 挂起。

接下来的代码就是 goroutine 被唤醒后的各种收尾工作了。

# 案例分析

从 channel 接收和向 channel 发送数据的过程我们均会使用下面这个例子来进行说明：

```
func goroutineA(a <-chan int) {
    val := <- a
    fmt.Println("G1 received data: ", val)
    return
}

func goroutineB(b <-chan int) {
    val := <- b
    fmt.Println("G2 received data: ", val)
    return
}

func main() {
    ch := make(chan int)
    go goroutineA(ch)
    go goroutineB(ch)
    ch <- 3
    time.Sleep(time.Second)
}
```

首先创建了一个无缓冲的 channel，接着启动两个 goroutine，并将前面创建的 channel 传递进去。然后，向这个 channel 中发送数据 3，最后 sleep 1 秒后程序退出。

程序第 14 行创建了一个非缓冲型的 channel，我们只看 chan 结构体中的一些重要字段，来从整体层面看一下 chan 的状态，一开始什么都没有：


接着，第 15、16 行分别创建了一个 goroutine，各自执行了一个接收操作。通过前面的源码分析，我们知道，这两个 goroutine （后面称为 G1 和 G2 好了）都会被阻塞在接收操作。G1 和 G2 会挂在 channel 的 recq 队列中，形成一个双向循环链表。

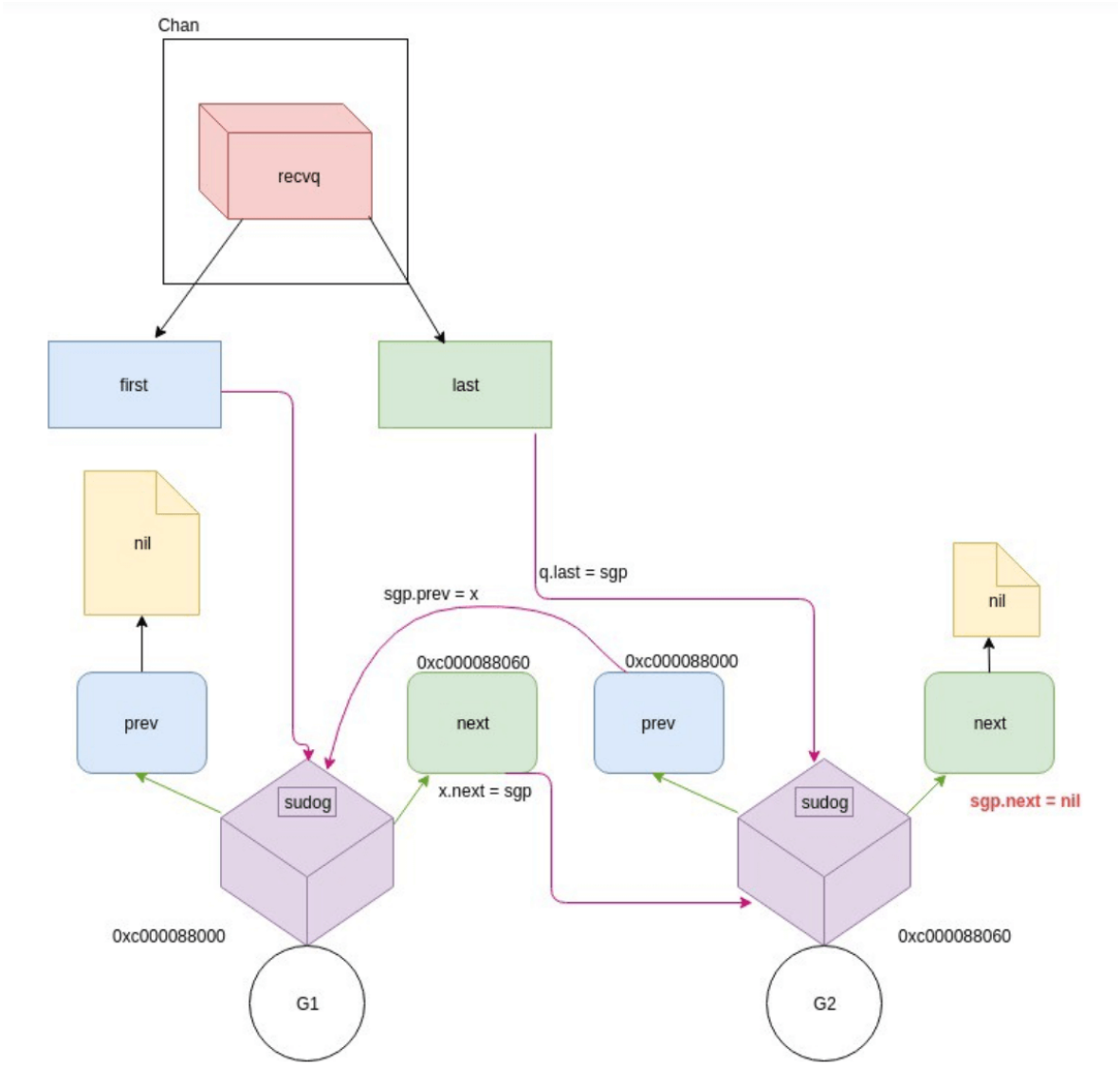
在程序的 17 行之前，chan 的整体数据结构如下：

```
chan int {
    qcount: 0,
    dataqsiz: 0,
    buf: *[0]int [],
    elemsize: 8,
    closed: 0,
    elemtype: *runtime._type {
        size: 8,
        ptrdata: 0,
        hash: 4149441018,
        tflag: tflagUncommon|tflagExtraStar|tflagNamed,
        align: 8,
        fieldalign: 8,
        kind: 130,
        alg: *(*runtime.typeAlg)(0x569eb0),
        gcdata: *1,
        str: 1015,
        ptrToThis: 45408,},
    sendx: 0,
    recvx: 0,
    recvq: waitq<int> {
        first: *(*sudog<int>)(0xc000088000),
        last: *(*sudog<int>)(0xc000088060),},
    sendq: waitq<int> {
        first: *sudog<int> nil,
        last: *sudog<int> nil,},
    lock: runtime.mutex {key: 0},}

```

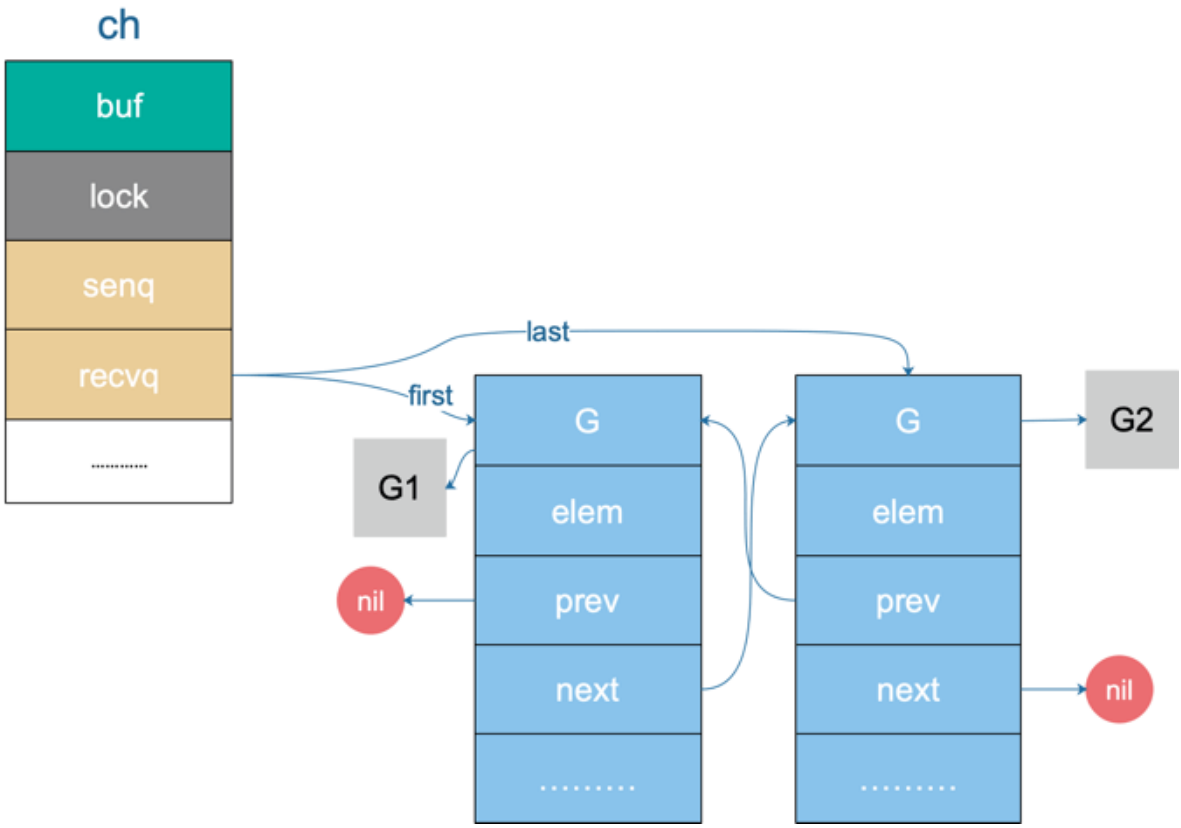
`buf` 指向一个长度为 0 的数组，qcount 为 0，表示 channel 中没有元素。重点关注 `recvq` 和 `sendq`，它们是 waitq 结构体，而 waitq 实际上就是一个双向链表，链表的元素是 sudog，里面包含 `g` 字段，`g` 表示一个 goroutine，所以 sudog 可以看成一个 goroutine。recvq 存储那些尝试读取 channel 但被阻塞的 goroutine，sendq 则存储那些尝试写入 channel，但被阻塞的 goroutine。

此时，我们可以看到，recvq 里挂了两个 goroutine，也就是前面启动的 G1 和 G2。因为没有 goroutine 接收，而 channel 又是无缓冲类型，所以 G1 和 G2 被阻塞。sendq 没有被阻塞的 goroutine。

`recvq` 的数据结构如下：

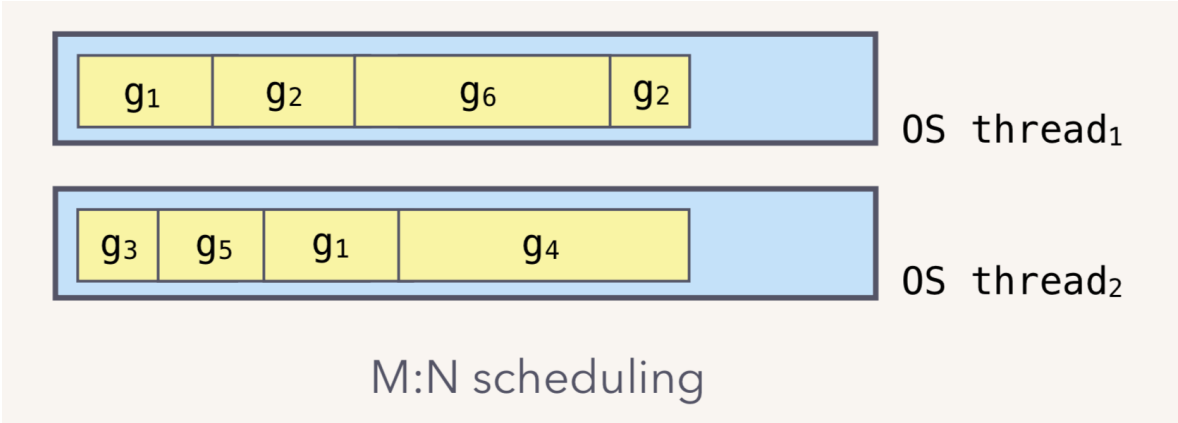



再从整体上来看一下 chan 此时的状态：

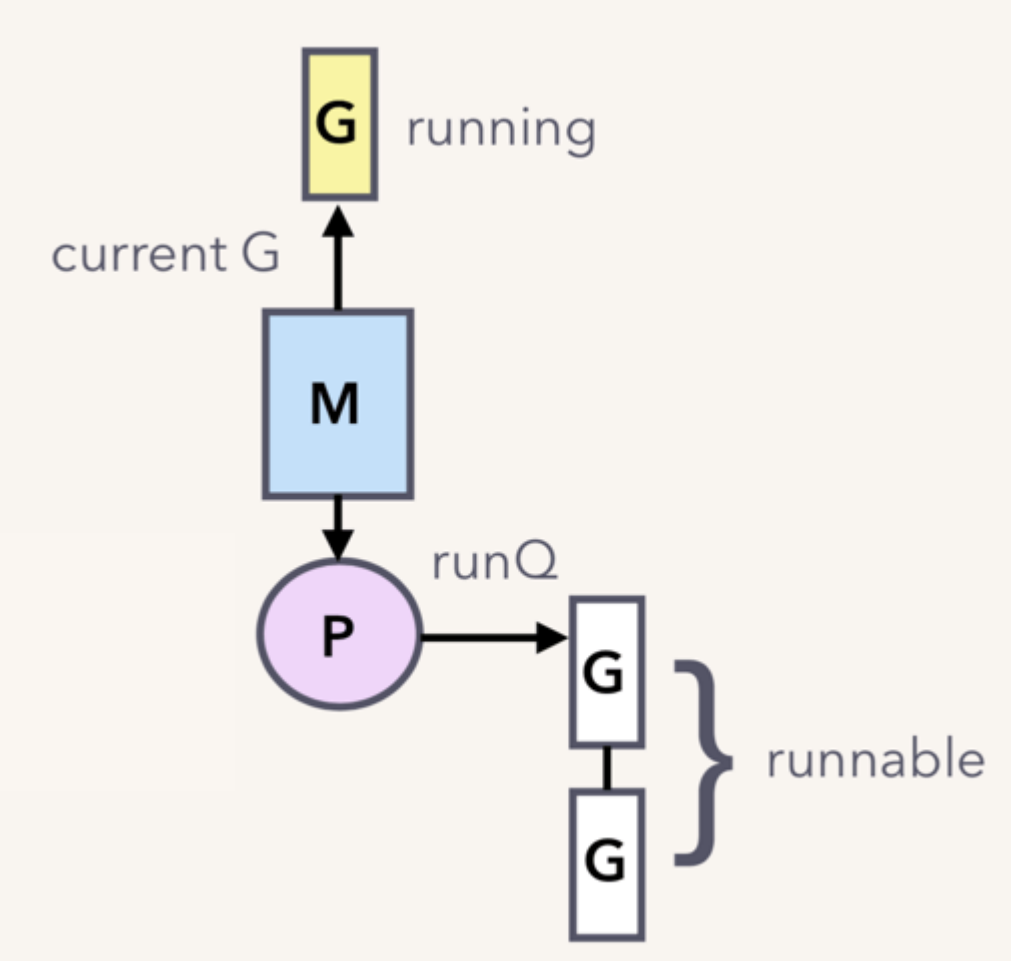



G1 和 G2 被挂起了，状态是 `WAITING` 。关于 goroutine 调度器这块不是今天的重点，当然后面肯定会写相关的文章。这里先简单说下，goroutine 是用户态的协程，由 Go runtime 进行管理，作为对比，内核线程由 OS 进行管理。Goroutine 更轻量，因此我们可以轻松创建数万 goroutine。

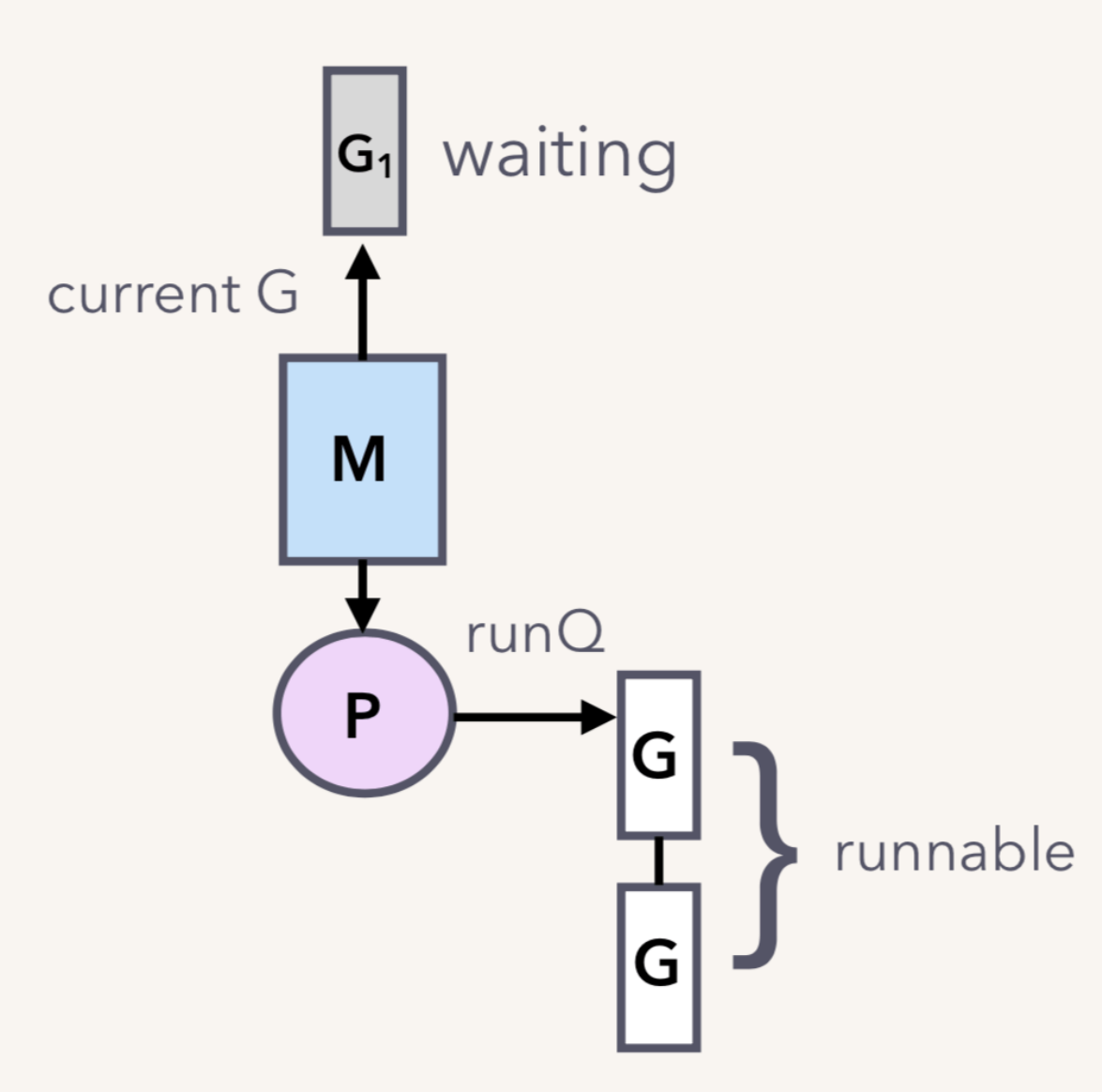
一个内核线程可以管理多个 goroutine，当其中一个 goroutine 阻塞时，内核线程可以调度其他的 goroutine 来运行，内核线程本身不会阻塞。这就是通常我们说的 `M:N` 模型：



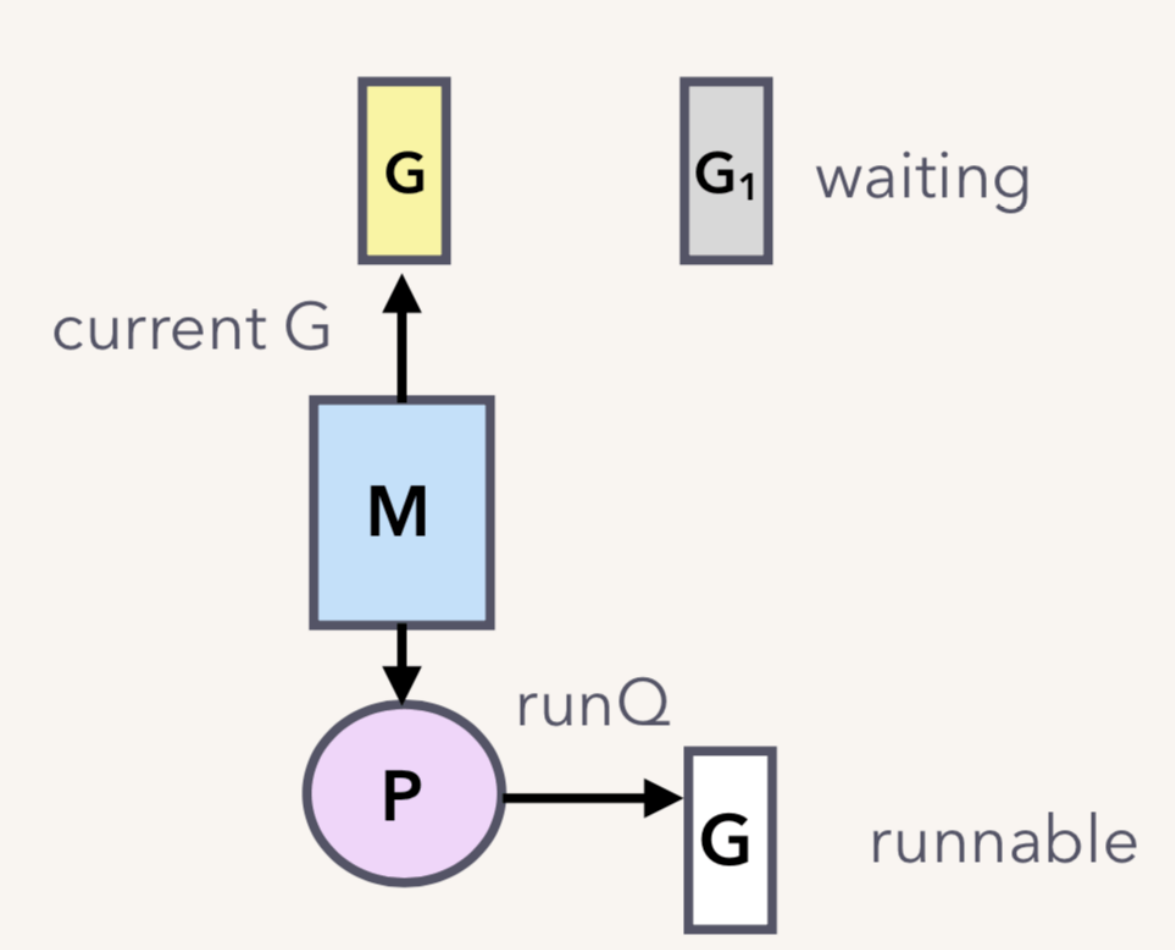

`M:N` 模型通常由三部分构成：M、P、G。M 是内核线程，负责运行 goroutine；P 是 context，保存 goroutine 运行所需要的上下文，它还维护了可运行（runnable）的 goroutine 列表；G 则是待运行的 goroutine。M 和 P 是 G 运行的基础。




继续回到例子。假设我们只有一个 M，当 G1（ `go goroutineA(ch)` ）运行到 `val := <- a` 时，它由本来的 running 状态变成了 waiting 状态（调用了 gopark 之后的结果）：




G1 脱离与 M 的关系，但调度器可不会让 M 闲着，所以会接着调度另一个 goroutine 来运行：




G2 也是同样的遭遇。现在 G1 和 G2 都被挂起了，等待着一个 sender 往 channel 里发送数据，才能得到解救。

## 参考资料

【深入 channel 底层】https://codeburst.io/diving-deep-into-the-golang-channels-549fd4ed21a8

【Kavya在Gopher Con 上关于 channel 的设计，非常好】
https://speakerd.s3.amazonaws.com/presentations/10ac0b1d76a6463aa98ad6a9dec917a7/GopherCon_v10.0.pdf

# 操作 channel 的情况总结

总结一下操作 channel 的结果：

| 操作 | nil channel | closed channel | not nil, not closed channel |
| --- | --- | --- | --- |
| close | panic | panic | 正常关闭 |
| 读 <- ch | 阻塞 | 读到对应类型的零值 | 阻塞或正常读取数据。缓冲型 channel 为空或非缓冲型 channel 没有等待发送者时会阻塞 |
| 写 ch <- | 阻塞 | panic | 阻塞或正常写入数据。非缓冲型 channel 没有等待接收者或缓冲型 channel buf 满时会被阻塞 |

总结一下，发生 panic 的情况有三种：向一个关闭的 channel 进行写操作；关闭一个 nil 的 channel；重复关闭一个 channel。

读、写一个 nil channel 都会被阻塞。

# 关闭一个 channel 的过程是怎样的

关闭某个 channel，会执行函数 `closechan` ：

```
func closechan(c *hchan) {
    // 关闭一个 nil channel，panic
    if c == nil {
        panic(plainError("close of nil channel"))
    }

    // 上锁
    lock(&c.lock)
    // 如果 channel 已经关闭
    if c.closed != 0 {
        unlock(&c.lock)
        // panic
        panic(plainError("close of closed channel"))
    }

    // ............

    // 修改关闭状态
    c.closed = 1

    var glist *g

    // 将 channel 所有等待接收队列的里 sudog 释放
    for {
        // 从接收队列里出队一个 sudog
        sg := c.recvq.dequeue()
        // 出队完毕，跳出循环
        if sg == nil {
            break
        }

        // 如果 elem 不为空，说明此 receiver 未忽略接收数据
        // 给它赋一个相应类型的零值
        if sg.elem != nil {
            typedmemclr(c.elemtype, sg.elem)
            sg.elem = nil
        }
        if sg.releasetime != 0 {
            sg.releasetime = cputicks()
        }
        // 取出 goroutine
        gp := sg.g
        gp.param = nil
        if raceenabled {
            raceacquireg(gp, unsafe.Pointer(c))
        }
        // 相连，形成链表
        gp.schedlink.set(glist)
        glist = gp
    }

    // 将 channel 等待发送队列里的 sudog 释放
    // 如果存在，这些 goroutine 将会 panic
    for {
        // 从发送队列里出队一个 sudog
```

```
        sg := c.sendq.dequeue()
        if sg == nil {
            break
        }

        // 发送者会 panic
        sg.elem = nil
        if sg.releasetime != 0 {
            sg.releasetime = cputicks()
        }
        gp := sg.g
        gp.param = nil
        if raceenabled {
            raceacquireg(gp, unsafe.Pointer(c))
        }
        // 形成链表
        gp.schedlink.set(glist)
        glist = gp
    }
    // 解锁
    unlock(&c.lock)

    // Ready all Gs now that we've dropped the channel lock.
    // 遍历链表
    for glist != nil {
        // 取最后一个
        gp := glist
        // 向前走一步，下一个唤醒的 g
        glist = glist.schedlink.ptr()
        gp.schedlink = 0
        // 唤醒相应 goroutine
        goready(gp, 3)
    }
}
```

close 逻辑比较简单，对于一个 channel，recvq 和 sendq 中分别保存了阻塞的发送者和接收者。关闭 channel 后，对于等待接收者而言，会收到一个相应类型的零值。对于等待发送者，会直接 panic。所以，在不了解 channel 还有没有接收者的情况下，不能贸然关闭 channel。

close 函数先上一把大锁，接着把所有挂在这个 channel 上的 sender 和 receiver 全都连成一个 sudog 链表，再解锁。最后，再将所有的 sudog 全都唤醒。

唤醒之后，该干嘛干嘛。sender 会继续执行 chansend 函数里 goparkunlock 函数之后的代码，很不幸，检测到 channel 已经关闭了，panic。receiver 则比较幸运，进行一些扫尾工作后，返回。这里，selected 返回 true，而返回值 received 则要根据 channel 是否关闭，返回不同的值。如果 channel 关闭，received 为 false，否则为 true。这我们分析的这种情况下，received 返回 false。

# map

# map 的底层实现原理是什么

## 什么是 map

维基百科里这样定义 map：

> In computer science, an associative array, map, symbol table, or dictionary is an abstract data type composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection.

简单说明一下：在计算机科学里，被称为相关数组、map、符号表或者字典，是由一组 `<key, value>` 对组成的抽象数据结构，，并且同一个 key 只会出现一次。

有两个关键点：map 是由 `key-value` 对组成的； `key` 只会出现一次。

和 map 相关的操作主要是：

1. 增加一个 k-v 对 —— Add or insert；
2. 删除一个 k-v 对 —— Remove or delete；
3. 修改某个 k 对应的 v —— Reassign；
4. 查询某个 k 对应的 v —— Lookup；

简单说就是最基本的 `增删查改` 。

map 的设计也被称为 “The dictionary problem”，它的任务是设计一种数据结构用来维护一个集合的数据，并且可以同时对集合进行增删查改的操作。最主要的数据结构有两种： `哈希查找表（Hash table）` 、 `搜索树（Search tree）` 。

哈希查找表用一个哈希函数将 key 分配到不同的桶（bucket，也就是数组的不同 index）。这样，开销主要在哈希函数的计算以及数组的常数访问时间。在很多场景下，哈希查找表的性能很高。

哈希查找表一般会存在“碰撞”的问题，就是说不同的 key 被哈希到了同一个 bucket。一般有两种应对方法： `链表法` 和 `开放地址法` 。 `链表法` 将一个 bucket 实现成一个链表，落在同一个 bucket 中的 key 都会插入这个链表。 `开放地址法` 则是碰撞发生后，通过一定的规律，在数组的后面挑选“空位”，用来放置新的 key。

搜索树法一般采用自平衡搜索树，包括：AVL 树，红黑树。面试时经常会被问到，甚至被要求手写红黑树代码，很多时候，面试官自己都写不上来，非常过分。

自平衡搜索树法的最差搜索效率是 O(logN)，而哈希查找表最差是 O(N)。当然，哈希查找表的平均查找效率是 O(1)，如果哈希函数设计的很好，最坏的情况基本不会出现。还有一点，遍历自平衡搜索树，返回的 key 序列，一般会按照从小到大的顺序；而哈希查找表则是乱序的。

## map 的底层如何实现

首先声明我用的 Go 版本：

```
go version go1.9.2 darwin/amd64
```

前面说了 map 实现的几种方案，Go 语言采用的是哈希查找表，并且使用链表解决哈希冲突。

接下来我们要探索 map 的核心原理，一窥它的内部结构。

### map 内存模型

在源码中，表示 map 的结构体是 hmap，它是 hashmap 的“缩写”：

```
// A header for a Go map.
type hmap struct {
    // 元素个数，调用 len(map) 时，直接返回此值
    count     int
    flags     uint8
    // buckets 的对数 log_2
    B         uint8
    // overflow 的 bucket 近似数
    noverflow uint16
    // 计算 key 的哈希的时候会传入哈希函数
    hash0     uint32
    // 指向 buckets 数组，大小为 2^B
    // 如果元素个数为0，就为 nil
    buckets    unsafe.Pointer
    // 扩容的时候，buckets 长度会是 oldbuckets 的两倍
    oldbuckets unsafe.Pointer
    // 指示扩容进度，小于此地址的 buckets 迁移完成
    nevacuate  uintptr
    extra *mapextra // optional fields
}
```

说明一下，`B` 是 buckets 数组的长度的对数，也就是说 buckets 数组的长度就是 2^B。bucket 里面存储了 key 和 value，后面会再讲。

buckets 是一个指针，最终它指向的是一个结构体：

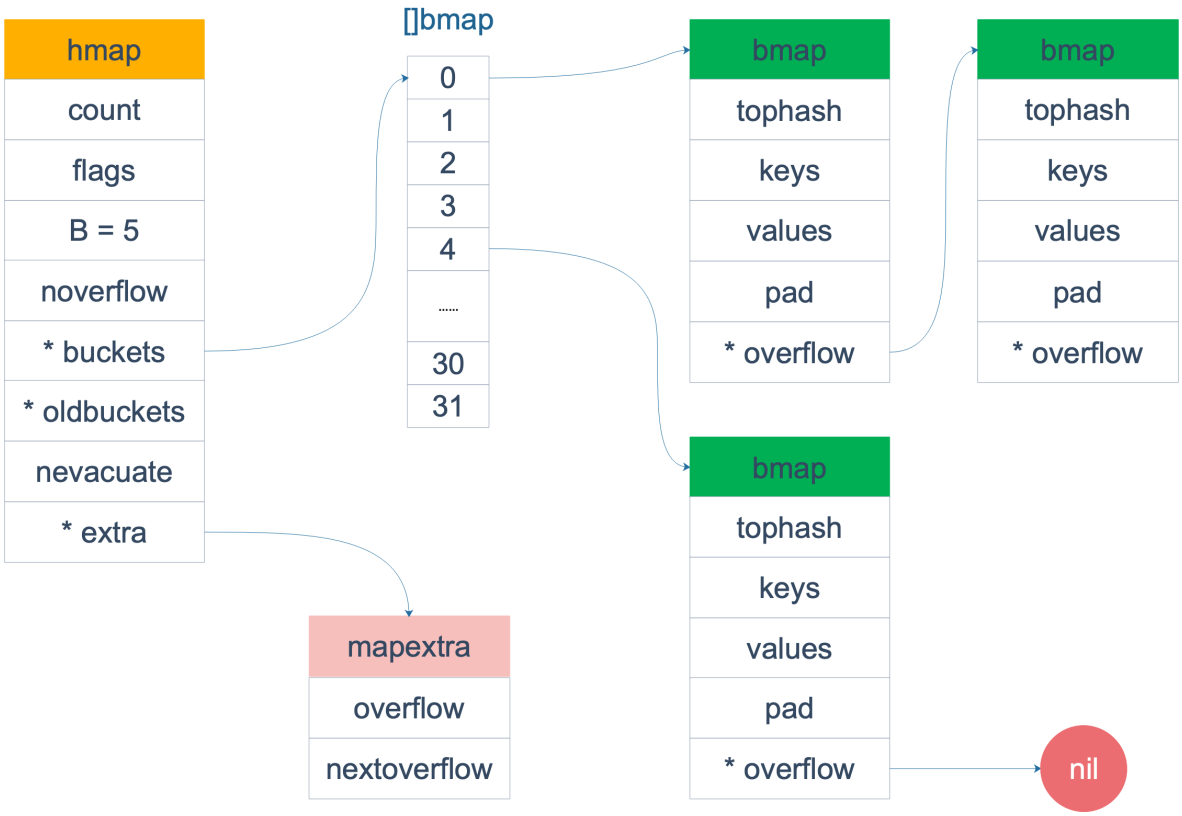
```
type bmap struct {
    tophash [bucketCnt]uint8
}
```

但这只是表面(src/runtime/hashmap.go)的结构，编译期间会给它加料，动态地创建一个新的结构：

```
type bmap struct {
    topbits  [8]uint8
    keys     [8]keytype
    values   [8]valuetype
    pad      uintptr
    overflow uintptr
}
```

`bmap` 就是我们常说的“桶”，桶里面会最多装 8 个 key，这些 key 之所以会落入同一个桶，是因为它们经过哈希计算后，哈希结果是“一类”的。在桶内，又会根据 key 计算出来的 hash 值的高 8 位来决定 key 到底落入桶内的哪个位置（一个桶内最多有8个位置）。

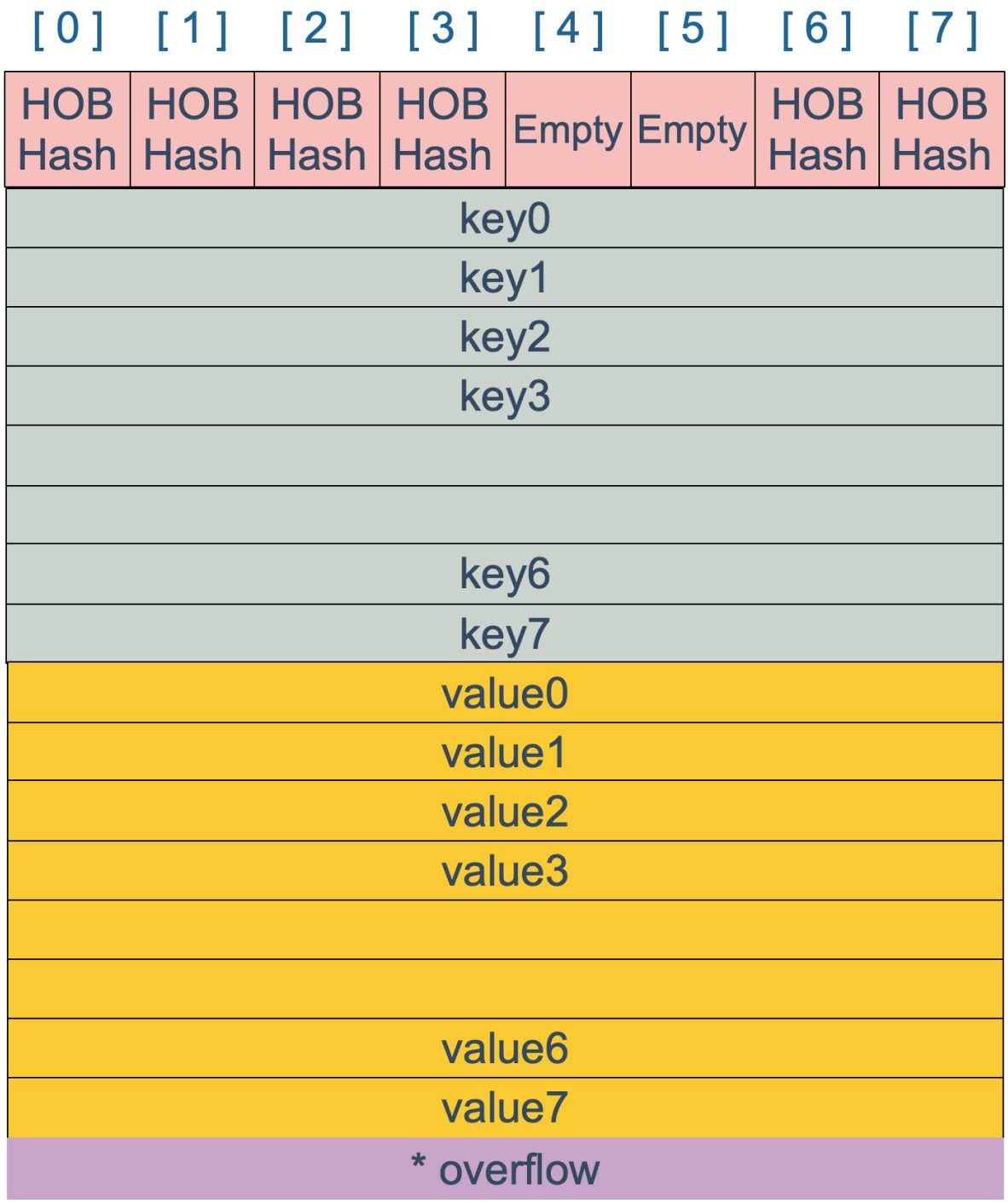
来一个整体的图：




当 map 的 key 和 value 都不是指针，并且 size 都小于 128 字节的情况下，会把 bmap 标记为不含指针，这样可以避免 gc 时扫描整个 hmap。但是，我们看 bmap 其实有一个 overflow 的字段，是指针类型的，破坏了 bmap 不含指针的设想，这时会把 overflow 移动到 extra 字段来。

```
type mapextra struct {
    // overflow[0] contains overflow buckets for hmap.buckets.
    // overflow[1] contains overflow buckets for hmap.oldbuckets.
    overflow [2]*[]*bmap

    // nextOverflow 包含空闲的 overflow bucket，这是预分配的 bucket
    nextOverflow *bmap
}
```

bmap 是存放 k-v 的地方，我们把视角拉近，仔细看 bmap 的内部组成。




上图就是 bucket 的内存模型， `HOB Hash` 指的就是 top hash。注意到 key 和 value 是各自放在一起的，并不是 `key/value/key/value/...` 这样的形式。源码里说明这样的好处是在某些情况下可以省略掉 padding 字段，节省内存空间。

例如，有这样一个类型的 map：

```
map[int64]int8
```

如果按照 `key/value/key/value/...` 这样的模式存储，那在每一个 key/value 对之后都要额外 padding 7 个字节；而将所有的 key，value 分别绑定到一起，这种形式 `key/key/.../value/value/...` ，则只需要在最后添加 padding。

每个 bucket 设计成最多只能放 8 个 key-value 对，如果有第 9 个 key-value 落入当前的 bucket，那就需要再构建一个 bucket，通过 `overflow` 指针连接起来。

## 创建 map

从语法层面上来说，创建 map 很简单：

```
ageMp := make(map[string]int)
// 指定 map 长度
ageMp := make(map[string]int, 8)

// ageMp 为 nil，不能向其添加元素，会直接panic
var ageMp map[string]int
```

通过汇编语言可以看到，实际上底层调用的是 `makemap` 函数，主要做的工作就是初始化 `hmap` 结构体的各种字段，例如计算 B 的大小，设置哈希种子 hash0 等等。

```
func makemap(t *maptype, hint int64, h *hmap, bucket unsafe.Pointer) *hmap {
    // 省略各种条件检查...

    // 找到一个 B，使得 map 的装载因子在正常范围内
    B := uint8(0)
    for ; overLoadFactor(hint, B); B++ {
    }

    // 初始化 hash table
    // 如果 B 等于 0，那么 buckets 就会在赋值的时候再分配
    // 如果长度比较大，分配内存会花费长一点
    buckets := bucket
    var extra *mapextra
    if B != 0 {
        var nextOverflow *bmap
        buckets, nextOverflow = makeBucketArray(t, B)
        if nextOverflow != nil {
            extra = new(mapextra)
            extra.nextOverflow = nextOverflow
        }
    }

    // 初始化 hamp
    if h == nil {
        h = (*hmap)(newobject(t.hmap))
    }
    h.count = 0
    h.B = B
    h.extra = extra
    h.flags = 0
    h.hash0 = fastrand()
    h.buckets = buckets
    h.oldbuckets = nil
    h.nevacuate = 0
    h.noverflow = 0

    return h
}
```

【引申1】slice 和 map 分别作为函数参数时有什么区别？

注意，这个函数返回的结果：`*hmap`，它是一个指针，而我们之前讲过的 `makeslice` 函数返回的是 `Slice` 结构体：

```
func makeslice(et *_type, len, cap int) slice
```

回顾一下 slice 的结构体定义：

```
// runtime/slice.go
type slice struct {
    array unsafe.Pointer // 元素指针
    len   int // 长度
    cap   int // 容量
}
```

结构体内部包含底层的数据指针。

makemap 和 makeslice 的区别，带来一个不同点：当 map 和 slice 作为函数参数时，在函数参数内部对 map 的操作会影响 map 自身；而对 slice 却不会（之前讲 slice 的文章里有讲过）。

主要原因：一个是指针（`*hmap`），一个是结构体（`slice`）。Go 语言中的函数传参都是值传递，在函数内部，参数会被 copy 到本地。`*hmap` 指针 copy 完之后，仍然指向同一个 map，因此函数内部对 map 的操作会影响实参。而 slice 被 copy 后，会成为一个新的 slice，对它进行的操作不会影响到实参。

## 哈希函数

map 的一个关键点在于，哈希函数的选择。在程序启动时，会检测 cpu 是否支持 aes，如果支持，则使用 aes hash，否则使用 memhash。这是在函数 `alginit()` 中完成，位于路径：`src/runtime/alg.go` 下。

> hash 函数，有加密型和非加密型。
> 加密型的一般用于加密数据、数字摘要等，典型代表就是 md5、sha1、sha256、aes256 这种；
> 非加密型的一般就是查找。在 map 的应用场景中，用的是查找。
> 选择 hash 函数主要考察的是两点：性能、碰撞概率。

之前我们讲过，表示类型的结构体：

```
type _type struct {
    size       uintptr
    ptrdata    uintptr // size of memory prefix holding all pointers
    hash       uint32
    tflag      tflag
    align      uint8
    fieldalign uint8
    kind       uint8
    alg        *typeAlg
    gcdata    *byte
    str       nameOff
    ptrToThis typeOff
}
```

其中 `alg` 字段就和哈希相关，它是指向如下结构体的指针：

```
// src/runtime/alg.go
type typeAlg struct {
    // (ptr to object, seed) -> hash
    hash func(unsafe.Pointer, uintptr) uintptr
    // (ptr to object A, ptr to object B) -> ==?
```

```
    equal func(unsafe.Pointer, unsafe.Pointer) bool
}
```

typeAlg 包含两个函数，hash 函数计算类型的哈希值，而 equal 函数则计算两个类型是否“哈希相等”。

对于 string 类型，它的 hash、equal 函数如下：

```
func strhash(a unsafe.Pointer, h uintptr) uintptr {
    x := (*stringStruct)(a)
    return memhash(x.str, h, uintptr(x.len))
}

func strequal(p, q unsafe.Pointer) bool {
    return *(*string)(p) == *(*string)(q)
}
```

根据 key 的类型，_type 结构体的 alg 字段会被设置对应类型的 hash 和 equal 函数。

## key 定位过程

key 经过哈希计算后得到哈希值，共 64 个 bit 位（64位机，32位机就不讨论了，现在主流都是64位机），计算它到底要落在哪个桶时，只会用到最后 B 个 bit 位。还记得前面提到过的 B 吗？如果 B = 5，那么桶的数量，也就是 buckets 数组的长度是 2^5 = 32。

例如，现在有一个 key 经过哈希函数计算后，得到的哈希结果是：

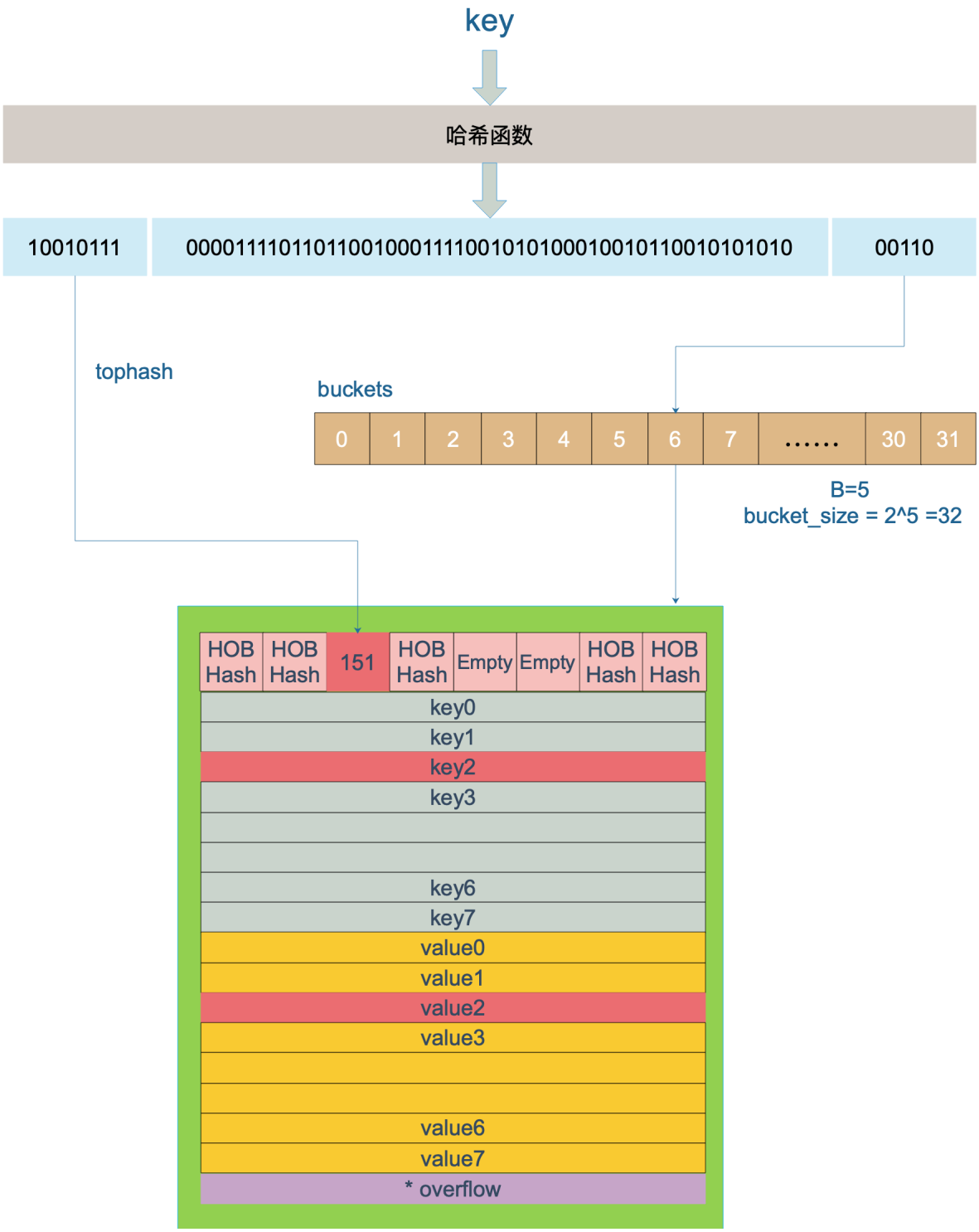
```
10010111 | 0000111101101100100011111001010100010010110010101010 | 01010
```

用最后的 5 个 bit 位，也就是 `01010`，值为 10，也就是 10 号桶。这个操作实际上就是取余操作，但是取余开销太大，所以代码实现上用的位操作代替。

再用哈希值的高 8 位，找到此 key 在 bucket 中的位置，这是在寻找已有的 key。最开始桶内还没有 key，新加入的 key 会找到第一个空位，放入。

buckets 编号就是桶编号，当两个不同的 key 落在同一个桶中，也就是发生了哈希冲突。冲突的解决手段是用链表法：在 bucket 中，从前往后找到第一个空位。这样，在查找某个 key 时，先找到对应的桶，再去遍历 bucket 中的 key。

这里参考曹大 github 博客里的一张图，原图是 ascii 图，geek 味十足，可以从参考资料找到曹大的博客，推荐大家去看看。




上图中，假定 B = 5，所以 bucket 总数就是 2^5 = 32。首先计算出待查找 key 的哈希，使用低 5 位 `00110` ，找到对应的 6 号 bucket，使用高 8 位 `10010111` ，对应十进制 151，在 6 号 bucket 中寻找 tophash 值（HOB hash）为 151 的 key，找到了 2 号槽位，这样整个查找过程就结束了。

如果在 bucket 中没找到，并且 overflow 不为空，还要继续去 overflow bucket 中寻找，直到找到或是所有的 key 槽位都找遍了，包括所有的 overflow bucket。

我们来看下源码吧，哈哈！通过汇编语言可以看到，查找某个 key 的底层函数是 `mapacess` 系列函数，函数的作用类似，区别在下一节会讲到。这里我们直接看 `mapacess1` 函数：

```
func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer {
    // ……

    // 如果 h 什么都没有，返回零值
    if h == nil || h.count == 0 {
        return unsafe.Pointer(&zeroVal[0])
    }

    // 写和读冲突
    if h.flags&hashWriting != 0 {
        throw("concurrent map read and map write")
    }

    // 不同类型 key 使用的 hash 算法在编译期确定
    alg := t.key.alg

    // 计算哈希值，并且加入 hash0 引入随机性
    hash := alg.hash(key, uintptr(h.hash0))

    // 比如 B=5，那 m 就是31，二进制是全 1
    // 求 bucket num 时，将 hash 与 m 相与，
    // 达到 bucket num 由 hash 的低 8 位决定的效果
    m := uintptr(1)<<h.B - 1

    // b 就是 bucket 的地址
    b := (*bmap)(add(h.buckets, (hash&m)*uintptr(t.bucketsize)))

    // oldbuckets 不为 nil，说明发生了扩容
    if c := h.oldbuckets; c != nil {
        // 如果不是同 size 扩容（看后面扩容的内容）
        // 对应条件 1 的解决方案
        if !h.sameSizeGrow() {
            // 新 bucket 数量是老的 2 倍
            m >>= 1
        }

        // 求出 key 在老的 map 中的 bucket 位置
        oldb := (*bmap)(add(c, (hash&m)*uintptr(t.bucketsize)))

        // 如果 oldb 没有搬迁到新的 bucket
        // 那就在老的 bucket 中寻找
        if !evacuated(oldb) {
            b = oldb
        }
    }

    // 计算出高 8 位的 hash
    // 相当于右移 56 位，只取高8位
    top := uint8(hash >> (sys.PtrSize*8 - 8))

    // 增加一个 minTopHash
    if top < minTopHash {
        top += minTopHash
    }
    for {
        // 遍历 8 个 bucket
        for i := uintptr(0); i < bucketCnt; i++ {
            // tophash 不匹配，继续
            if b.tophash[i] != top {
                continue
            }
```

```
            // tophash 匹配，定位到 key 的位置
            k := add(unsafe.Pointer(b), dataOffset+i*uintptr(t.keysize))
            // key 是指针
            if t.indirectkey {
                // 解引用
                k = *((*unsafe.Pointer)(k))
            }
            // 如果 key 相等
            if alg.equal(key, k) {
                // 定位到 value 的位置
                v := add(unsafe.Pointer(b), dataOffset+bucketCnt*uintptr(t.keysize)+i*uintptr(t.valuesize))
                // value 解引用
                if t.indirectvalue {
                    v = *((*unsafe.Pointer)(v))
                }
                return v
            }
        }

        // bucket 找完（还没找到），继续到 overflow bucket 里找
        b = b.overflow(t)
        // overflow bucket 也找完了，说明没有目标 key
        // 返回零值
        if b == nil {
            return unsafe.Pointer(&zeroVal[0])
        }
    }
}
```

函数返回 h[key] 的指针，如果 h 中没有此 key，那就会返回一个 key 相应类型的零值，不会返回 nil。

代码整体比较直接，没什么难懂的地方。跟着上面的注释一步步理解就好了。

这里，说一下定位 key 和 value 的方法以及整个循环的写法。

```
// key 定位公式
k := add(unsafe.Pointer(b), dataOffset+i*uintptr(t.keysize))

// value 定位公式
v := add(unsafe.Pointer(b), dataOffset+bucketCnt*uintptr(t.keysize)+i*uintptr(t.valuesize))
```

b 是 bmap 的地址，这里 bmap 还是源码里定义的结构体，只包含一个 tophash 数组，经编译器扩充之后的结构体才包含 key，value，overflow 这些字段。dataOffset 是 key 相对于 bmap 起始地址的偏移：

```
dataOffset = unsafe.Offsetof(struct {
        b bmap
        v int64
    }{}.v)
```

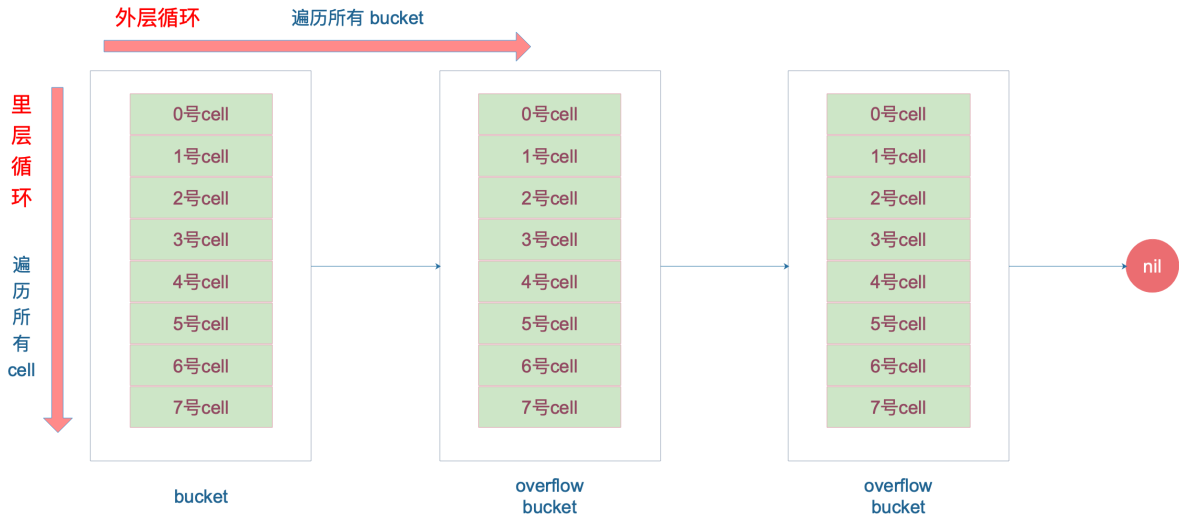
因此 bucket 里 key 的起始地址就是 unsafe.Pointer(b)+dataOffset。第 i 个 key 的地址就要在此基础上跨过 i 个 key 的大小；而我们又知道，value 的地址是在所有 key 之后，因此第 i 个 value 的地址还需要加上所有 key 的偏移。理解了这些，上面 key 和 value 的定位公式就很好理解了。

再说整个大循环的写法，最外层是一个无限循环，通过

```
b = b.overflow(t)
```

遍历所有的 bucket，这相当于是一个 bucket 链表。

当定位到一个具体的 bucket 时，里层循环就是遍历这个 bucket 里所有的 cell，或者说所有的槽位，也就是 bucketCnt=8 个槽位。整个循环过程：




再说一下 minTopHash，当一个 cell 的 tophash 值小于 minTopHash 时，标志这个 cell 的迁移状态。因为这个状态值是放在 tophash 数组里，为了和正常的哈希值区分开，会给 key 计算出来的哈希值一个增量：minTopHash。这样就能区分正常的 top hash 值和表示状态的哈希值。

下面的这几种状态就表征了 bucket 的情况：

```
// 空的 cell，也是初始时 bucket 的状态
empty          = 0
// 空的 cell，表示 cell 已经被迁移到新的 bucket
evacuatedEmpty = 1
// key,value 已经搬迁完毕，但是 key 都在新 bucket 前半部分，
// 后面扩容部分会再讲到。
evacuatedX     = 2
// 同上，key 在后半部分
evacuatedY     = 3
// tophash 的最小正常值
minTopHash     = 4
```

源码里判断这个 bucket 是否已经搬迁完毕，用到的函数：

```
func evacuated(b *bmap) bool {
    h := b.tophash[0]
    return h > empty && h < minTopHash
}
```

只取了 tophash 数组的第一个值，判断它是否在 0-4 之间。对比上面的常量，当 top hash 是 `evacuatedEmpty` 、 `evacuatedX` 、 `evacuatedY` 这三个值之一，说明此 bucket 中的 key 全部被搬迁到了新 bucket。

# 可以边遍历边删除吗

map 并不是一个线程安全的数据结构。同时读写一个 map 是未定义的行为，如果被检测到，会直接 panic。

上面说的是发生在多个协程同时读写同一个 map 的情况下。 如果在同一个协程内边遍历边删除，并不会检测到同时读写，理论上是可以这样做的。但是，遍历的结果就可能不会是相同的了，有可能结果遍历结果集中包含了删除的 key，也有可能不包含，这取决于删除 key 的时间：是在遍历到 key 所在的 bucket 时刻前或者后。

一般而言，这可以通过读写锁来解决： `sync.RWMutex` 。

读之前调用 `RLock()` 函数，读完之后调用 `RUnlock()` 函数解锁；写之前调用 `Lock()` 函数，写完之后，调用 `Unlock()` 解锁。

另外， `sync.Map` 是线程安全的 map，也可以使用。

# map 的删除过程是怎样的

写操作底层的执行函数是 `mapdelete` ：

```
func mapdelete(t *maptype, h *hmap, key unsafe.Pointer)
```

根据 key 类型的不同，删除操作会被优化成更具体的函数：

| key 类型 | 删除 |
| --- | --- |
| uint32 | mapdelete_fast32(t *maptype, h *hmap, key uint32) |
| uint64 | mapdelete_fast64(t *maptype, h *hmap, key uint64) |
| string | mapdelete_faststr(t *maptype, h *hmap, ky string) |

当然，我们只关心 `mapdelete` 函数。它首先会检查 h.flags 标志，如果发现写标位是 1，直接 panic，因为这表明有其他协程同时在进行写操作。

计算 key 的哈希，找到落入的 bucket。检查此 map 如果正在扩容的过程中，直接触发一次搬迁操作。

删除操作同样是两层循环，核心还是找到 key 的具体位置。寻找过程都是类似的，在 bucket 中挨个 cell 寻找。

找到对应位置后，对 key 或者 value 进行“清零”操作：

```
// 对 key 清零
if t.indirectkey {
    *(*unsafe.Pointer)(k) = nil
} else {
    typedmemclr(t.key, k)
}

// 对 value 清零
if t.indirectvalue {
    *(*unsafe.Pointer)(v) = nil
} else {
    typedmemclr(t.elem, v)
}
```

最后，将 count 值减 1，将对应位置的 tophash 值置成 `Empty` 。

这块源码同样比较简单，感兴起直接去看代码。

# 可以对 map 的元素取地址吗

无法对 map 的 key 或 value 进行取址。以下代码不能通过编译：

```
package main

import "fmt"

func main() {
    m := make(map[string]int)

    fmt.Println(&m["qcrao"])
}
```

编译报错：

```
./main.go:8:14: cannot take the address of m["qcrao"]
```

如果通过其他 hack 的方式，例如 unsafe.Pointer 等获取到了 key 或 value 的地址，也不能长期持有，因为一旦发生扩容，key 和 value 的位置就会改变，之前保存的地址也就失效了。

# 如何比较两个 map 相等

map 深度相等的条件：

```
1、都为 nil
2、非空、长度相等，指向同一个 map 实体对象
3、相应的 key 指向的 value “深度”相等
```

直接将使用 map1 == map2 是错误的。这种写法只能比较 map 是否为 nil。

```
package main

import "fmt"

func main() {
    var m map[string]int
    var n map[string]int

    fmt.Println(m == nil)
    fmt.Println(n == nil)

    // 不能通过编译
    //fmt.Println(m == n)
}
```

输出结果：

```
true
true
```

因此只能是遍历map 的每个元素，比较元素是否都是深度相等。

# 如何实现两种 get 操作

Go 语言中读取 map 有两种语法：带 comma 和 不带 comma。当要查询的 key 不在 map 里，带 comma 的用法会返回一个 bool 型变量提示 key 是否在 map 中；而不带 comma 的语句则会返回一个 key 类型的零值。如果 key 是 int 型就会返回 0，如果 key 是 string 类型，就会返回空字符串。

```
package main

import "fmt"

func main() {
    ageMap := make(map[string]int)
    ageMap["qcrao"] = 18

    // 不带 comma 用法
    age1 := ageMap["stefno"]
    fmt.Println(age1)

    // 带 comma 用法
    age2, ok := ageMap["stefno"]
    fmt.Println(age2, ok)
}
```

运行结果：

```
0
0 false
```

以前一直觉得好神奇，怎么实现的？这其实是编译器在背后做的工作：分析代码后，将两种语法对应到底层两个不同的函数。

```
// src/runtime/hashmap.go
func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer
func mapaccess2(t *maptype, h *hmap, key unsafe.Pointer) (unsafe.Pointer, bool)
```

源码里，函数命名不拘小节，直接带上后缀 1，2，完全不理会《代码大全》里的那一套命名的做法。从上面两个函数的声明也可以看出差别了，`mapaccess2` 函数返回值多了一个 bool 型变量，两者的代码也是完全一样的，只是在返回值后面多加了一个 false 或者 true。

另外，根据 key 的不同类型，编译器还会将查找、插入、删除的函数用更具体的函数替换，以优化效率：

| key 类型 | 查找 |
| --- | --- |
| uint32 | mapaccess1_fast32(t *maptype, h *hmap, key uint32) unsafe.Pointer |
| uint32 | mapaccess2_fast32(t *maptype, h *hmap, key uint32) (unsafe.Pointer, bool) |
| uint64 | mapaccess1_fast64(t *maptype, h *hmap, key uint64) unsafe.Pointer |

| key 类型 | 查找 |
| --- | --- |
| uint64 | mapaccess2_fast64(t *maptype, h *hmap, key uint64) (unsafe.Pointer, bool) |
| string | mapaccess1_faststr(t *maptype, h *hmap, ky string) unsafe.Pointer |
| string | mapaccess2_faststr(t *maptype, h *hmap, ky string) (unsafe.Pointer, bool) |

这些函数的参数类型直接是具体的 uint32、unt64、string，在函数内部由于提前知晓了 key 的类型，所以内存布局是很清楚的，因此能节省很多操作，提高效率。

上面这些函数都是在文件 `src/runtime/hashmap_fast.go` 里。

# map 是线程安全的吗

map 不是线程安全的。

在查找、赋值、遍历、删除的过程中都会检测写标志，一旦发现写标志置位（等于1），则直接 panic。赋值和删除函数在检测完写标志是复位之后，先将写标志位置位，才会进行之后的操作。

检测写标志：

```
if h.flags&hashWriting == 0 {
        throw("concurrent map writes")
    }
```

设置写标志：

```
h.flags |= hashWriting
```

# map 的遍历过程是怎样的

本来 map 的遍历过程比较简单：遍历所有的 bucket 以及它后面挂的 overflow bucket，然后挨个遍历 bucket 中的所有 cell。每个 bucket 中包含 8 个 cell，从有 key 的 cell 中取出 key 和 value，这个过程就完成了。

但是，现实并没有这么简单。还记得前面讲过的扩容过程吗？扩容过程不是一个原子的操作，它每次最多只搬运 2 个 bucket，所以如果触发了扩容操作，那么在很长时间里，map 的状态都是处于一个中间态：有些 bucket 已经搬迁到新家，而有些 bucket 还待在老地方。

因此，遍历如果发生在扩容的过程中，就会涉及到遍历新老 bucket 的过程，这是难点所在。

我先写一个简单的代码样例，假装不知道遍历过程具体调用的是什么函数：

```
package main

import "fmt"

func main() {
    ageMp := make(map[string]int)
    ageMp["qcrao"] = 18

    for name, age := range ageMp {
        fmt.Println(name, age)
    }
}
```

执行命令：

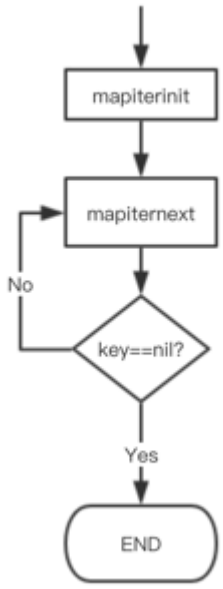
```
go tool compile -S main.go
```

得到汇编命令。这里就不逐行讲解了，可以去看之前的几篇文章，说得很详细。

关键的几行汇编代码如下：

```
// ......
0x0124 00292 (test16.go:9)      CALL    runtime.mapiterinit(SB)

// ......
0x01fb 00507 (test16.go:9)      CALL    runtime.mapiternext(SB)
0x0200 00512 (test16.go:9)      MOVQ    ""..autotmp_4+160(SP), AX
0x0208 00520 (test16.go:9)      TESTQ   AX, AX
0x020b 00523 (test16.go:9)      JNE     302

// ......
```

这样，关于 map 迭代，底层的函数调用关系一目了然。先是调用 `mapiterinit` 函数初始化迭代器，然后循环调用 `mapiternext` 函数进行 map 迭代。




迭代器的结构体定义：

```
type hiter struct {
    // key 指针
    key         unsafe.Pointer
    // value 指针
    value       unsafe.Pointer
    // map 类型，包含如 key size 大小等
    t           *maptype
    // map header
    h           *hmap
    // 初始化时指向的 bucket
    buckets     unsafe.Pointer
    // 当前遍历到的 bmap
    bptr        *bmap
    overflow    [2]*[]*bmap
    // 起始遍历的 bucet 编号
    startBucket uintptr
    // 遍历开始时 cell 的编号（每个 bucket 中有 8 个 cell）
    offset      uint8
    // 是否从头遍历了
    wrapped     bool
    // B 的大小
    B           uint8
    // 指示当前 cell 序号
    i           uint8
    // 指向当前的 bucket
    bucket      uintptr
    // 因为扩容，需要检查的 bucket
    checkBucket uintptr
}
```

`mapiterinit` 就是对 hiter 结构体里的字段进行初始化赋值操作。

前面已经提到过，即使是对一个写死的 map 进行遍历，每次出来的结果也是无序的。下面我们就可以近距离地观察他们的实现了。

```
// 生成随机数 r
r := uintptr(fastrand())
```

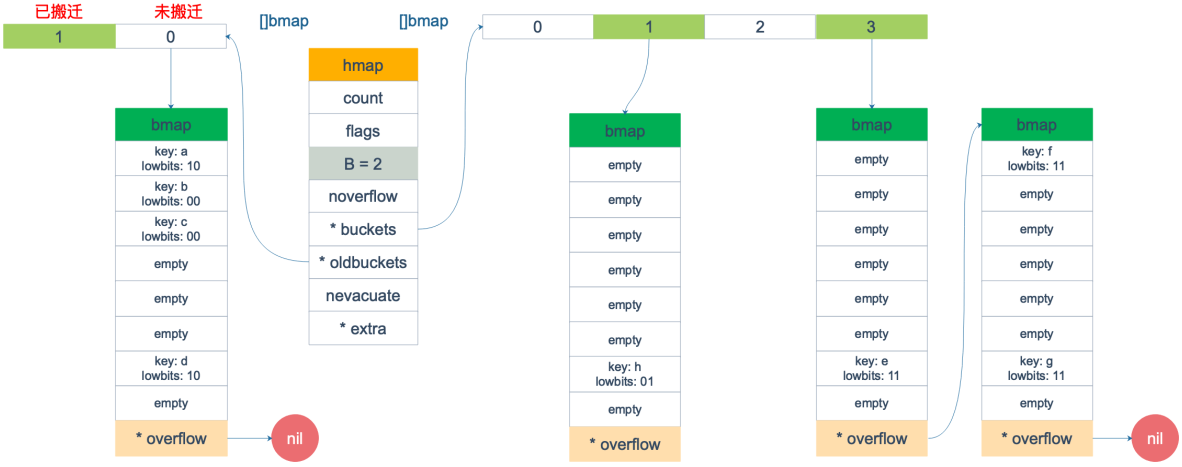
```
if h.B > 31-bucketCntBits {
    r += uintptr(fastrand()) << 31
}

// 从哪个 bucket 开始遍历
it.startBucket = r & (uintptr(1)<<h.B - 1)
// 从 bucket 的哪个 cell 开始遍历
it.offset = uint8(r >> h.B & (bucketCnt - 1))
```

例如，B = 2，那 `uintptr(1)<<h.B - 1` 结果就是 3，低 8 位为 `0000 0011` ，将 r 与之相与，就可以得到一个 `0~3` 的 bucket 序号；bucketCnt - 1 等于 7，低 8 位为 `0000 0111` ，将 r 右移 2 位后，与 7 相与，就可以得到一个 `0~7` 号的 cell。

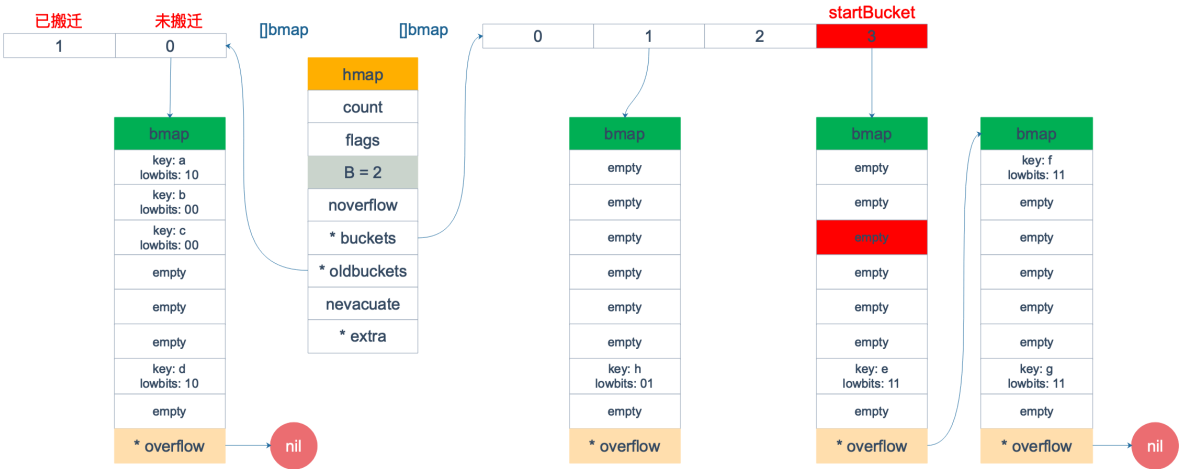
于是，在 `mapiternext` 函数中就会从 it.startBucket 的 it.offset 号的 cell 开始遍历，取出其中的 key 和 value，直到又回到起点 bucket，完成遍历过程。

源码部分比较好看懂，尤其是理解了前面注释的几段代码后，再看这部分代码就没什么压力了。所以，接下来，我将通过图形化的方式讲解整个遍历过程，希望能够清晰易懂。

假设我们有下图所示的一个 map，起始时 B = 1，有两个 bucket，后来触发了扩容（这里不要深究扩容条件，只是一个设定），B 变成 2。并且， `1 号` bucket 中的内容搬迁到了新的 bucket， `1 号` 裂变成 `1 号` 和 `3 号` ； `0 号` bucket 暂未搬迁。老的 bucket 挂在在 `*oldbuckets` 指针上面，新的 bucket 则挂在 `*buckets` 指针上面。




这时，我们对此 map 进行遍历。假设经过初始化后，startBucket = 3，offset = 2。于是，遍历的起点将是 3 号 bucket 的 2 号 cell，下面这张图就是开始遍历时的状态：



标红的表示起始位置，bucket 遍历顺序为：3 -> 0 -> 1 -> 2。

因为 3 号 bucket 对应老的 1 号 bucket，因此先检查老 1 号 bucket 是否已经被搬迁过。判断方法就是：

```
func evacuated(b *bmap) bool {
    h := b.tophash[0]
    return h > empty && h < minTopHash
}
```

如果 b.tophash[0] 的值在标志值范围内，即在 (0,4) 区间里，说明已经被搬迁过了。

```
empty = 0
evacuatedEmpty = 1
evacuatedX = 2
evacuatedY = 3
minTopHash = 4
```

在本例中，老 1 号 bucket 已经被搬迁过了。所以它的 tophash[0] 值在 (0,4) 范围内，因此只用遍历新的 3 号 bucket。

依次遍历 3 号 bucket 的 cell，这时候会找到第一个非空的 key：元素 e。到这里，mapiternext 函数返回，这时我们的遍历结果仅有一个元素：


由于返回的 key 不为空，所以会继续调用 mapiternext 函数。

继续从上次遍历到的地方往后遍历，从新 3 号 overflow bucket 中找到了元素 f 和 元素 g。

遍历结果集也因此壮大：


新 3 号 bucket 遍历完之后，回到了新 0 号 bucket。0 号 bucket 对应老的 0 号 bucket，经检查，老 0 号 bucket 并未搬迁，因此对新 0 号 bucket 的遍历就改为遍历老 0 号 bucket。那是不是把老 0 号 bucket 中的所有 key 都取出来呢？

并没有这么简单，回忆一下，老 0 号 bucket 在搬迁后将裂变成 2 个 bucket：新 0 号、新 2 号。而我们此时正在遍历的只是新 0 号 bucket（注意，遍历都是遍历的 `*bucket` 指针，也就是所谓的新 buckets）。所以，我们只会取出老 0 号 bucket 中那些在裂变之后，分配到新 0 号 bucket 中的那些 key。

因此，`lowbits == 00` 的将进入遍历结果集：


和之前的流程一样，继续遍历新 1 号 bucket，发现老 1 号 bucket 已经搬迁，只用遍历新 1 号 bucket 中现有的元素就可以了。结果集变成：


继续遍历新 2 号 bucket，它来自老 0 号 bucket，因此需要在老 0 号 bucket 中那些会裂变到新 2 号 bucket 中的 key，也就是 `lowbit == 10` 的那些 key。

这样，遍历结果集变成：


最后，继续遍历到新 3 号 bucket 时，发现所有的 bucket 都已经遍历完毕，整个迭代过程执行完毕。

顺便说一下，如果碰到 key 是 `math.NaN()` 这种的，处理方式类似。核心还是要看它被分裂后具体落入哪个 bucket。只不过只用看它 top hash 的最低位。如果 top hash 的最低位是 0 ，分配到 X part；如果是 1 ，则分配到 Y part。据此决定是否取出 key，放到遍历结果集里。

map 遍历的核心在于理解 2 倍扩容时，老 bucket 会分裂到 2 个新 bucket 中去。而遍历操作，会按照新 bucket 的序号顺序进行，碰到老 bucket 未搬迁的情况时，要在老 bucket 中找到将来要搬迁到新 bucket 来的 key。

# map 中的 key 为什么是无序的

map 在扩容后，会发生 key 的搬迁，原来落在同一个 bucket 中的 key，搬迁后，有些 key 就要远走高飞了（bucket 序号加上了 2^B）。而遍历的过程，就是按顺序遍历 bucket，同时按顺序遍历 bucket 中的 key。搬迁后，key 的位置发生了重大的变化，有些 key 飞上高枝，有些 key 则原地不动。这样，遍历 map 的结果就不可能按原来的顺序了。

当然，如果我就一个 hard code 的 map，我也不会向 map 进行插入删除的操作，按理说每次遍历这样的 map 都会返回一个固定顺序的 key/value 序列吧。的确是这样，但是 Go 杜绝了这种做法，因为这样会给新手程序员带来误解，以为这是一定会发生的事情，在某些情况下，可能会酿成大错。

当然，Go 做得更绝，当我们在遍历 map 时，并不是固定地从 0 号 bucket 开始遍历，每次都是从一个随机值序号的 bucket 开始遍历，并且是从这个 bucket 的一个随机序号的 cell 开始遍历。这样，即使你是一个写死的 map，仅仅只是遍历它，也不太可能会返回一个固定序列的 key/value 对了。

多说一句，“迭代 map 的结果是无序的”这个特性是从 go 1.0 开始加入的。

# float 类型可以作为 map 的 key 吗

从语法上看，是可以的。Go 语言中只要是可比较的类型都可以作为 key。除开 slice，map，functions 这几种类型，其他类型都是 OK 的。具体包括：布尔值、数字、字符串、指针、通道、接口类型、结构体、只包含上述类型的数组。这些类型的共同特征是支持 `==` 和 `!=` 操作符，`k1 == k2` 时，可认为 k1 和 k2 是同一个 key。如果是结构体，只有 hash 后的值相等以及字面值相等，才被认为是相同的 key。很多字面值相等的，hash出来的值不一定相等，比如引用。

顺便说一句，任何类型都可以作为 value，包括 map 类型。

来看个例子：

```
func main() {
    m := make(map[float64]int)
    m[1.4] = 1
    m[2.4] = 2
    m[math.NaN()] = 3
    m[math.NaN()] = 3

    for k, v := range m {
        fmt.Printf("[%v, %d] ", k, v)
    }

    fmt.Printf("\nk: %v, v: %d\n", math.NaN(), m[math.NaN()])
    fmt.Printf("k: %v, v: %d\n", 2.400000000001, m[2.400000000001])
    fmt.Printf("k: %v, v: %d\n", 2.400000000000000000000001, m[2.400000000000000000000001])

    fmt.Println(math.NaN() == math.NaN())
}
```

程序的输出：

```
[2.4, 2] [NaN, 3] [NaN, 3] [1.4, 1]
k: NaN, v: 0
k: 2.400000000001, v: 0
k: 2.4, v: 2
false
```

例子中定义了一个 key 类型是 float 型的 map，并向其中插入了 4 个 key：1.4， 2.4， NAN，NAN。

打印的时候也打印出了 4 个 key，如果你知道 NAN != NAN，也就不奇怪了。因为他们比较的结果不相等，自然，在 map 看来就是两个不同的 key 了。

接着，我们查询了几个 key，发现 NAN 不存在，2.400000000001 也不存在，而 2.400000000000000000000001 却存在。

有点诡异，不是吗？

接着，我通过汇编发现了如下的事实：

当用 float64 作为 key 的时候，先要将其转成 unit64 类型，再插入 key 中。

具体是通过 `Float64frombits` 函数完成：

```
// Float64frombits returns the floating point number corresponding
// the IEEE 754 binary representation b.
func Float64frombits(b uint64) float64 { return *(*float64)(unsafe.Pointer(&b)) }
```

也就是将浮点数表示成 IEEE 754 规定的格式。如赋值语句：

```
0x00bd 00189 (test18.go:9)      LEAQ    "".statictmp_0(SB), DX
0x00c4 00196 (test18.go:9)      MOVQ    DX, 16(SP)
0x00c9 00201 (test18.go:9)      PCDATA  $0, $2
0x00c9 00201 (test18.go:9)      CALL    runtime.mapassign(SB)
```

`"".statictmp_0(SB)` 变量是这样的：

```
"".statictmp_0 SRODATA size=8
        0x0000 33 33 33 33 33 33 03 40
"".statictmp_1 SRODATA size=8
        0x0000 ff 3b 33 33 33 33 03 40
"".statictmp_2 SRODATA size=8
        0x0000 33 33 33 33 33 33 03 40
```

我们再来输出点东西：

```
package main

import (
    "fmt"
    "math"
)

func main() {
    m := make(map[float64]int)
    m[2.4] = 2

    fmt.Println(math.Float64bits(2.4))
    fmt.Println(math.Float64bits(2.400000000001))
    fmt.Println(math.Float64bits(2.400000000000000000000001))
}
```

```
4612586738352862003
4612586738352864255
4612586738352862003
```

转成十六进制为：

```
0x4003333333333333
0x4003333333333BFF
0x4003333333333333
```

和前面的 `"".statictmp_0` 比较一下，很清晰了吧。 `2.4` 和 `2.400000000000000000000001` 经过 `math.Float64bits()` 函数转换后的结果是一样的。自然，二者在 map 看来，就是同一个 key 了。

再来看一下 NAN（not a number）：

```
// NaN returns an IEEE 754 ``not-a-number'' value.
func NaN() float64 { return Float64frombits(uvnan) }
```

uvan 的定义为：

```
uvnan    = 0x7FF8000000000001
```

NAN() 直接调用 `Float64frombits` ，传入写死的 const 型变量 `0x7FF8000000000001` ，得到 NAN 型值。既然，NAN 是从一个常量解析得来的，为什么插入 map 时，会被认为是不同的 key?

这是由类型的哈希函数决定的，例如，对于 64 位的浮点数，它的哈希函数如下：

```
func f64hash(p unsafe.Pointer, h uintptr) uintptr {
    f := *(*float64)(p)
    switch {
    case f == 0:
        return c1 * (c0 ^ h) // +0, -0
    case f != f:
        return c1 * (c0 ^ h ^ uintptr(fastrand())) // any kind of NaN
    default:
        return memhash(p, h, 8)
    }
}
```

第二个 case， `f != f` 就是针对 `NAN` ，这里会再加一个随机数。

这样，所有的谜题都解开了。

由于 NAN 的特性：

```
NAN != NAN
hash(NAN) != hash(NAN)
```

因此向 map 中查找的 key 为 NAN 时，什么也查不到；如果向其中增加了 4 次 NAN，遍历会得到 4 个 NAN。

最后说结论：float 型可以作为 key，但是由于精度的问题，会导致一些诡异的问题，慎用之。

---

关于当 key 是引用类型时，判断两个 key 是否相等，需要 hash 后的值相等并且 key 的字面量相等。由 @WuMingyu 补充的例子：

```
func TestT(t *testing.T) {
    type S struct {
        ID    int
    }
    s1 := S{ID: 1}
    s2 := S{ID: 1}

    var h = map[*S]int {}
    h[&s1] = 1
    t.Log(h[&s1])
    t.Log(h[&s2])
    t.Log(s1 == s2)
}
```

test output:

```
=== RUN   TestT
--- PASS: TestT (0.00s)
    endpoint_test.go:74: 1
    endpoint_test.go:75: 0
```

```
    endpoint_test.go:76: true
PASS

Process finished with exit code 0
```

# map 的赋值过程是怎样的

通过汇编语言可以看到，向 map 中插入或者修改 key，最终调用的是 `mapassign` 函数。

实际上插入或修改 key 的语法是一样的，只不过前者操作的 key 在 map 中不存在，而后者操作的 key 存在 map 中。

mapassign 有一个系列的函数，根据 key 类型的不同，编译器会将其优化为相应的“快速函数”。

| key 类型 | 插入 |
| --- | --- |
| uint32 | mapassign_fast32(t *maptype, h *hmap, key uint32) unsafe.Pointer |
| uint64 | mapassign_fast64(t *maptype, h *hmap, key uint64) unsafe.Pointer |
| string | mapassign_faststr(t *maptype, h *hmap, ky string) unsafe.Pointer |

我们只用研究最一般的赋值函数 `mapassign` 。

整体来看，流程非常得简单：对 key 计算 hash 值，根据 hash 值按照之前的流程，找到要赋值的位置（可能是插入新 key，也可能是更新老 key），对相应位置进行赋值。

源码大体和之前讲的类似，核心还是一个双层循环，外层遍历 bucket 和它的 overflow bucket，内层遍历整个 bucket 的各个 cell。限于篇幅，这部分代码的注释我也不展示了，有兴趣的可以去看，保证理解了这篇文章内容后，能够看懂。

我这里会针对这个过程提几点重要的。

函数首先会检查 map 的标志位 flags。如果 flags 的写标志位此时被置 1 了，说明有其他协程在执行“写”操作，进而导致程序 panic。这也说明了 map 对协程是不安全的。

通过前文我们知道扩容是渐进式的，如果 map 处在扩容的过程中，那么当 key 定位到了某个 bucket 后，需要确保这个 bucket 对应的老 bucket 完成了迁移过程。即老 bucket 里的 key 都要迁移到新的 bucket 中来（分裂到 2 个新 bucket），才能在新的 bucket 中进行插入或者更新的操作。

上面说的操作是在函数靠前的位置进行的，只有进行完了这个搬迁操作后，我们才能放心地在新 bucket 里定位 key 要安置的地址，再进行之后的操作。

现在到了定位 key 应该放置的位置了，所谓找准自己的位置很重要。准备两个指针，一个（ `inserti` ）指向 key 的 hash 值在 tophash 数组所处的位置，另一个( `insertk` )指向 cell 的位置（也就是 key 最终放置的地址），当然，对应 value 的位置就很容易定位出来了。这三者实际上都是关联的，在 tophash 数组中的索引位置决定了 key 在整个 bucket 中的位置（共 8 个 key），而 value 的位置需要“跨过” 8 个 key 的长度。

在循环的过程中，inserti 和 insertk 分别指向第一个找到的空闲的 cell。如果之后在 map 没有找到 key 的存在，也就是说原来 map 中没有此 key，这意味着插入新 key。那最终 key 的安置地址就是第一次发现的“空位”（tophash 是 empty）。

如果这个 bucket 的 8 个 key 都已经放置满了，那在跳出循环后，发现 inserti 和 insertk 都是空，这时候需要在 bucket 后面挂上 overflow bucket。当然，也有可能是在 overflow bucket 后面再挂上一个 overflow bucket。这就说明，太多 key hash 到了此 bucket。

在正式安置 key 之前，还要检查 map 的状态，看它是否需要进行扩容。如果满足扩容的条件，就主动触发一次扩容操作。

这之后，整个之前的查找定位 key 的过程，还得再重新走一次。因为扩容之后，key 的分布都发生了变化。

最后，会更新 map 相关的值，如果是插入新 key，map 的元素数量字段 count 值会加 1；在函数之初设置的 `hashWriting` 写标志出会清零。

另外，有一个重要的点要说一下。前面说的找到 key 的位置，进行赋值操作，实际上并不准确。我们看 `mapassign` 函数的原型就知道，函数并没有传入 value 值，所以赋值操作是什么时候执行的呢？

```
func mapassign(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer
```

答案还得从汇编语言中寻找。我直接揭晓答案，有兴趣可以私下去研究一下。 `mapassign` 函数返回的指针就是指向的 key 所对应的 value 值位置，有了地址，就很好操作赋值了。

# map 的扩容过程是怎样的

使用哈希表的目的就是要快速查找到目标 key，然而，随着向 map 中添加的 key 越来越多，key 发生碰撞的概率也越来越大。bucket 中的 8 个 cell 会被逐渐塞满，查找、插入、删除 key 的效率也会越来越低。最理想的情况是一个 bucket 只装一个 key，这样，就能达到 `O(1)` 的效率，但这样空间消耗太大，用空间换时间的代价太高。

Go 语言采用一个 bucket 里装载 8 个 key，定位到某个 bucket 后，还需要再定位到具体的 key，这实际上又用了时间换空间。

当然，这样做，要有一个度，不然所有的 key 都落在了同一个 bucket 里，直接退化成了链表，各种操作的效率直接降为 O(n)，是不行的。

因此，需要有一个指标来衡量前面描述的情况，这就是 `装载因子` 。Go 源码里这样定义 `装载因子` ：

```
loadFactor := count / (2^B)
```

count 就是 map 的元素个数，2^B 表示 bucket 数量。

再来说触发 map 扩容的时机：在向 map 插入新 key 的时候，会进行条件检测，符合下面这 2 个条件，就会触发扩容：

1. 装载因子超过阈值，源码里定义的阈值是 6.5。
2. overflow 的 bucket 数量过多：当 B 小于 15，也就是 bucket 总数 2^B 小于 2^15 时，如果 overflow 的 bucket 数量超过 2^B；当 B >= 15，也就是 bucket 总数 2^B 大于等于 2^15，如果 overflow 的 bucket 数量超过 2^15。

通过汇编语言可以找到赋值操作对应源码中的函数是 `mapassign` ，对应扩容条件的源码如下：

```
// src/runtime/hashmap.go/mapassign

// 触发扩容时机
if !h.growing() && (overLoadFactor(int64(h.count), h.B) || tooManyOverflowBuckets(h.noverflow, h.B)) {
        hashGrow(t, h)
    }

// 装载因子超过 6.5
func overLoadFactor(count int64, B uint8) bool {
    return count >= bucketCnt && float32(count) >= loadFactor*float32((uint64(1)<<B))
}

// overflow buckets 太多
func tooManyOverflowBuckets(noverflow uint16, B uint8) bool {
    if B < 16 {
        return noverflow >= uint16(1)<<B
    }
    return noverflow >= 1<<15
}
```

解释一下：

第 1 点：我们知道，每个 bucket 有 8 个空位，在没有溢出，且所有的桶都装满了的情况下，装载因子算出来的结果是 8。因此当装载因子超过 6.5 时，表明很多 bucket 都快要装满了，查找效率和插入效率都变低了。在这个时候进行扩容是有必要的。

第 2 点：是对第 1 点的补充。就是说在装载因子比较小的情况下，这时候 map 的查找和插入效率也很低，而第 1 点识别不出来这种情况。表面现象就是计算装载因子的分子比较小，即 map 里元素总数少，但是 bucket 数量多（真实分配的 bucket 数量多，包括大量的 overflow bucket）。

不难想像造成这种情况的原因：不停地插入、删除元素。先插入很多元素，导致创建了很多 bucket，但是装载因子达不到第 1 点的临界值，未触发扩容来缓解这种情况。之后，删除元素降低元素总数量，再插入很多元素，导致创建很多的 overflow bucket，但就是不会触犯第 1 点的规定，你能拿我怎么办？overflow bucket 数量太多，导致 key 会很分散，查找插入效率低得吓人，因此出台第 2 点规定。这就像是一座空城，房子很多，但是住户很少，都分散了，找起人来很困难。

对于命中条件 1，2 的限制，都会发生扩容。但是扩容的策略并不相同，毕竟两种条件应对的场景不同。

对于条件 1，元素太多，而 bucket 数量太少，很简单：将 B 加 1，bucket 最大数量（2^B）直接变成原来 bucket 数量的 2 倍。于是，就有新老 bucket 了。注意，这时候元素都在老 bucket 里，还没迁移到新的 bucket 来。而且，新 bucket 只是最大数量变为原来最大数量（2^B）的 2 倍（2^B * 2）。

对于条件 2，其实元素没那么多，但是 overflow bucket 数特别多，说明很多 bucket 都没装满。解决办法就是开辟一个新 bucket 空间，将老 bucket 中的元素移动到新 bucket，使得同一个 bucket 中的 key 排列地更紧密。这样，原来，在 overflow bucket 中的 key 可以移动到 bucket 中来。结果是节省空间，提高 bucket 利用率，map 的查找和插入效率自然就会提升。

对于条件 2 的解决方案，曹大的博客里还提出了一个极端的情况：如果插入 map 的 key 哈希都一样，就会落到同一个 bucket 里，超过 8 个就会产生 overflow bucket，结果也会造成 overflow bucket 数过多。移动元素其实解决不了问题，因为这时整个哈希表已经退化成了一个链表，操作效率变成了 `O(n)` 。

再来看一下扩容具体是怎么做的。由于 map 扩容需要将原有的 key/value 重新搬迁到新的内存地址，如果有大量的 key/value 需要搬迁，会非常影响性能。因此 Go map 的扩容采取了一种称为“渐进式”地方式，原有的 key 并不会一次性搬迁完毕，每次最多只会搬迁 2 个 bucket。

上面说的 `hashGrow()` 函数实际上并没有真正地“搬迁”，它只是分配好了新的 buckets，并将老的 buckets 挂到了 oldbuckets 字段上。真正搬迁 buckets 的动作在 `growWork()` 函数中，而调用 `growWork()` 函数的动作是在 mapassign 和 mapdelete 函数中。也就是插入或修改、删除 key 的时候，都会尝试进行搬迁 buckets 的工作。先检查 oldbuckets 是否搬迁完毕，具体来说就是检查 oldbuckets 是否为 nil。

我们先看 `hashGrow()` 函数所做的工作，再来看具体的搬迁 buckets 是如何进行的。

```
func hashGrow(t *maptype, h *hmap) {
    // B+1 相当于是原来 2 倍的空间
    bigger := uint8(1)

    // 对应条件 2
    if !overLoadFactor(int64(h.count), h.B) {
        // 进行等量的内存扩容，所以 B 不变
        bigger = 0
        h.flags |= sameSizeGrow
    }
    // 将老 buckets 挂到 buckets 上
    oldbuckets := h.buckets
    // 申请新的 buckets 空间
    newbuckets, nextOverflow := makeBucketArray(t, h.B+bigger)

    flags := h.flags &^ (iterator | oldIterator)
    if h.flags&iterator != 0 {
        flags |= oldIterator
    }
    // 提交 grow 的动作
    h.B += bigger
    h.flags = flags
    h.oldbuckets = oldbuckets
    h.buckets = newbuckets
    // 搬迁进度为 0
    h.nevacuate = 0
    // overflow buckets 数为 0
    h.noverflow = 0
```

```
    // ......
}
```

主要是申请到了新的 buckets 空间，把相关的标志位都进行了处理：例如标志 nevacuate 被置为 0， 表示当前搬迁进度为 0。

值得一说的是对 `h.flags` 的处理：

```
flags := h.flags &^ (iterator | oldIterator)
if h.flags&iterator != 0 {
    flags |= oldIterator
}
```

这里得先说下运算符：&^。这叫 `按位置 0` 运算符。例如：

```
x = 01010011
y = 01010100
z = x &^ y = 00000011
```

如果 y bit 位为 1，那么结果 z 对应 bit 位就为 0，否则 z 对应 bit 位就和 x 对应 bit 位的值相同。

所以上面那段对 flags 一顿操作的代码的意思是：先把 h.flags 中 iterator 和 oldIterator 对应位清 0，然后如果发现 iterator 位为 1，那就把它转接到 oldIterator 位，使得 oldIterator 标志位变成 1。潜台词就是：buckets 现在挂到了 oldBuckets 名下了，对应的标志位也转接过去吧。

几个标志位如下：

```
// 可能有迭代器使用 buckets
iterator     = 1
// 可能有迭代器使用 oldbuckets
oldIterator  = 2
// 有协程正在向 map 中写入 key
hashWriting  = 4
// 等量扩容（对应条件 2）
sameSizeGrow = 8
```

再来看看真正执行搬迁工作的 growWork() 函数。

```
func growWork(t *maptype, h *hmap, bucket uintptr) {
    // 确认搬迁老的 bucket 对应正在使用的 bucket
    evacuate(t, h, bucket&h.oldbucketmask())

    // 再搬迁一个 bucket，以加快搬迁进程
    if h.growing() {
        evacuate(t, h, h.nevacuate)
    }
}
```

h.growing() 函数非常简单：

```
func (h *hmap) growing() bool {
    return h.oldbuckets != nil
}
```

如果 `oldbuckets` 不为空，说明还没有搬迁完毕，还得继续搬。

`bucket&h.oldbucketmask()` 这行代码，如源码注释里说的，是为了确认搬迁的 bucket 是我们正在使用的 bucket。`oldbucketmask()` 函数返回扩容前的 map 的 bucketmask。

所谓的 bucketmask，作用就是将 key 计算出来的哈希值与 bucketmask 相与，得到的结果就是 key 应该落入的桶。比如 B = 5，那么 bucketmask 的低 5 位是 `11111`，其余位是 `0`，hash 值与其相与的意思是，只有 hash 值的低 5 位决策 key 到底落入哪个 bucket。

接下来，我们集中所有的精力在搬迁的关键函数 evacuate。源码贴在下面，不要紧张，我会加上大面积的注释，通过注释绝对是能看懂的。之后，我会再对搬迁过程作详细说明。

源码如下：

```
func evacuate(t *maptype, h *hmap, oldbucket uintptr) {
    // 定位老的 bucket 地址
    b := (*bmap)(add(h.oldbuckets, oldbucket*uintptr(t.bucketsize)))
    // 结果是 2^B，如 B = 5，结果为32
    newbit := h.noldbuckets()
    // key 的哈希函数
    alg := t.key.alg
    // 如果 b 没有被搬迁过
    if !evacuated(b) {
        var (
            // 表示bucket 移动的目标地址
            x, y   *bmap
            // 指向 x,y 中的 key/val
            xi, yi int
            // 指向 x，y 中的 key
            xk, yk unsafe.Pointer
            // 指向 x，y 中的 value
            xv, yv unsafe.Pointer
        )
        // 默认是等 size 扩容，前后 bucket 序号不变
        // 使用 x 来进行搬迁
        x = (*bmap)(add(h.buckets, oldbucket*uintptr(t.bucketsize)))
        xi = 0
        xk = add(unsafe.Pointer(x), dataOffset)
        xv = add(xk, bucketCnt*uintptr(t.keysize))、

        // 如果不是等 size 扩容，前后 bucket 序号有变
        // 使用 y 来进行搬迁
        if !h.sameSizeGrow() {
            // y 代表的 bucket 序号增加了 2^B
            y = (*bmap)(add(h.buckets, (oldbucket+newbit)*uintptr(t.bucketsize)))
            yi = 0
            yk = add(unsafe.Pointer(y), dataOffset)
            yv = add(yk, bucketCnt*uintptr(t.keysize))
        }

        // 遍历所有的 bucket，包括 overflow buckets
        // b 是老的 bucket 地址
        for ; b != nil; b = b.overflow(t) {
            k := add(unsafe.Pointer(b), dataOffset)
            v := add(k, bucketCnt*uintptr(t.keysize))

            // 遍历 bucket 中的所有 cell
            for i := 0; i < bucketCnt; i, k, v = i+1, add(k, uintptr(t.keysize)), add(v, uintptr(t.valuesiz
e)) {
                // 当前 cell 的 top hash 值
                top := b.tophash[i]
                // 如果 cell 为空，即没有 key
                if top == empty {
```

```
                    // 那就标志它被"搬迁"过
                    b.tophash[i] = evacuatedEmpty
                    // 继续下个 cell
                    continue
                }
                // 正常不会出现这种情况
                // 未被搬迁的 cell 只可能是 empty 或是
                // 正常的 top hash（大于 minTopHash）
                if top < minTopHash {
                    throw("bad map state")
                }

                k2 := k
                // 如果 key 是指针，则解引用
                if t.indirectkey {
                    k2 = *((*unsafe.Pointer)(k2))
                }

                // 默认使用 X，等量扩容
                useX := true
                // 如果不是等量扩容
                if !h.sameSizeGrow() {
                    // 计算 hash 值，和 key 第一次写入时一样
                    hash := alg.hash(k2, uintptr(h.hash0))

                    // 如果有协程正在遍历 map
                    if h.flags&iterator != 0 {
                        // 如果出现 相同的 key 值，算出来的 hash 值不同
                        if !t.reflexivekey && !alg.equal(k2, k2) {
                            // 只有在 float 变量的 NaN() 情况下会出现
                            if top&1 != 0 {
                                // 第 B 位置 1
                                hash |= newbit
                            } else {
                                // 第 B 位置 0
                                hash &^= newbit
                            }
                            // 取高 8 位作为 top hash 值
                            top = uint8(hash >> (sys.PtrSize*8 - 8))
                            if top < minTopHash {
                                top += minTopHash
                            }
                        }
                    }

                    // 取决于新哈希值的 oldB+1 位是 0 还是 1
                    // 详细看后面的文章
                    useX = hash&newbit == 0
                }

                // 如果 key 搬到 X 部分
                if useX {
                    // 标志老的 cell 的 top hash 值，表示搬移到 X 部分
                    b.tophash[i] = evacuatedX
                    // 如果 xi 等于 8，说明要溢出了
                    if xi == bucketCnt {
                        // 新建一个 bucket
                        newx := h.newoverflow(t, x)
                        x = newx
                        // xi 从 0 开始计数
                        xi = 0
                        // xk 表示 key 要移动到的位置
```

```
                        xk = add(unsafe.Pointer(x), dataOffset)
                        // xv 表示 value 要移动到的位置
                        xv = add(xk, bucketCnt*uintptr(t.keysize))
                    }
                    // 设置 top hash 值
                    x.tophash[xi] = top
                    // key 是指针
                    if t.indirectkey {
                        // 将原 key（是指针）复制到新位置
                        *(*unsafe.Pointer)(xk) = k2 // copy pointer
                    } else {
                        // 将原 key（是值）复制到新位置
                        typedmemmove(t.key, xk, k) // copy value
                    }
                    // value 是指针，操作同 key
                    if t.indirectvalue {
                        *(*unsafe.Pointer)(xv) = *(*unsafe.Pointer)(v)
                    } else {
                        typedmemmove(t.elem, xv, v)
                    }

                    // 定位到下一个 cell
                    xi++
                    xk = add(xk, uintptr(t.keysize))
                    xv = add(xv, uintptr(t.valuesize))
                } else { // key 搬到 Y 部分，操作同 X 部分
                    // ......
                    // 省略了这部分，操作和 X 部分相同
                }
            }
        }
        // 如果没有协程在使用老的 buckets，就把老 buckets 清除掉，帮助gc
        if h.flags&oldIterator == 0 {
            b = (*bmap)(add(h.oldbuckets, oldbucket*uintptr(t.bucketsize)))
            // 只清除bucket 的 key,value 部分，保留 top hash 部分，指示搬迁状态
            if t.bucket.kind&kindNoPointers == 0 {
                memclrHasPointers(add(unsafe.Pointer(b), dataOffset), uintptr(t.bucketsize)-dataOffset)
            } else {
                memclrNoHeapPointers(add(unsafe.Pointer(b), dataOffset), uintptr(t.bucketsize)-dataOffset)
            }
        }
    }

    // 更新搬迁进度
    // 如果此次搬迁的 bucket 等于当前进度
    if oldbucket == h.nevacuate {
        // 进度加 1
        h.nevacuate = oldbucket + 1
        // Experiments suggest that 1024 is overkill by at least an order of magnitude.
        // Put it in there as a safeguard anyway, to ensure O(1) behavior.
        // 尝试往后看 1024 个 bucket
        stop := h.nevacuate + 1024
        if stop > newbit {
            stop = newbit
        }
        // 寻找没有搬迁的 bucket
        for h.nevacuate != stop && bucketEvacuated(t, h, h.nevacuate) {
            h.nevacuate++
        }

        // 现在 h.nevacuate 之前的 bucket 都被搬迁完毕
```

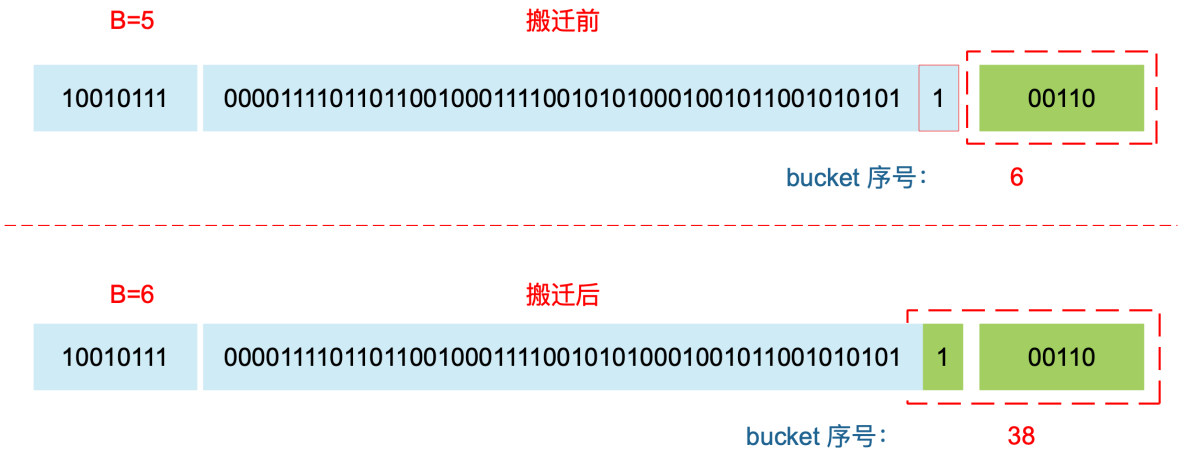
```
        // 所有的 buckets 搬迁完毕
        if h.nevacuate == newbit {
            // 清除老的 buckets
            h.oldbuckets = nil
            // 清除老的 overflow bucket
            // 回忆一下：[0] 表示当前 overflow bucket
            // [1] 表示 old overflow bucket
            if h.extra != nil {
                h.extra.overflow[1] = nil
            }
            // 清除正在扩容的标志位
            h.flags &^= sameSizeGrow
        }
    }
}
```

evacuate 函数的代码注释非常清晰，对着代码和注释是很容易看懂整个的搬迁过程的，耐心点。

搬迁的目的就是将老的 buckets 搬迁到新的 buckets。而通过前面的说明我们知道，应对条件 1，新的 buckets 数量是之前的一倍，应对条件 2，新的 buckets 数量和之前相等。

对于条件 1，从老的 buckets 搬迁到新的 buckets，由于 bucktes 数量不变，因此可以按序号来搬，比如原来在 0 号 bucktes，到新的地方后，仍然放在 0 号 buckets。

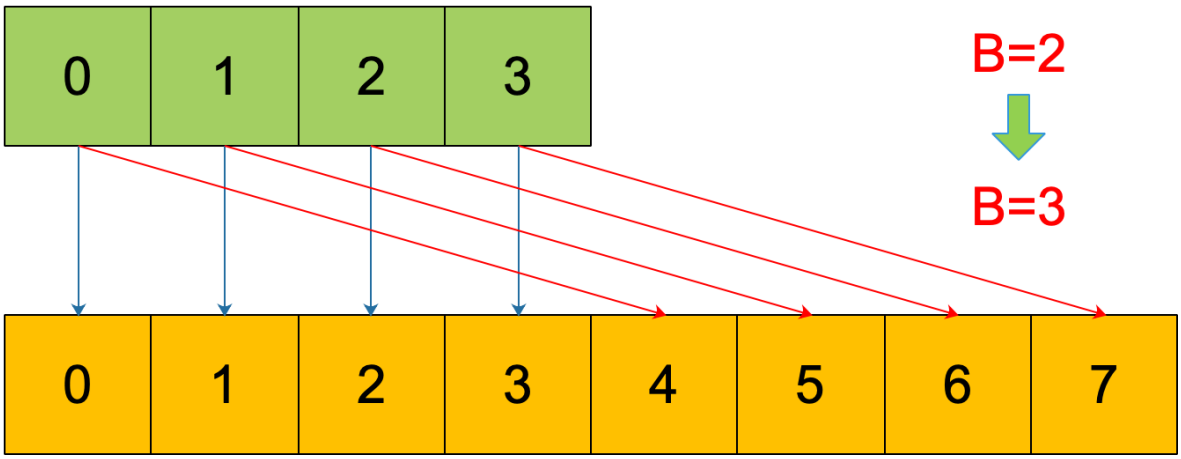
对于条件 2，就没这么简单了。要重新计算 key 的哈希，才能决定它到底落在哪个 bucket。例如，原来 B = 5，计算出 key 的哈希后，只用看它的低 5 位，就能决定它落在哪个 bucket。扩容后，B 变成了 6，因此需要多看一位，它的低 6 位决定 key 落在哪个 bucket。这称为 `rehash` 。




因此，某个 key 在搬迁前后 bucket 序号可能和原来相等，也可能是相比原来加上 2^B（原来的 B 值），取决于 hash 值 第 6 bit 位是 0 还是 1。

再明确一个问题：如果扩容后，B 增加了 1，意味着 buckets 总数是原来的 2 倍，原来 1 号的桶“裂变”到两个桶。

例如，原始 B = 2，1号 bucket 中有 2 个 key 的哈希值低 3 位分别为：010，110。由于原来 B = 2，所以低 2 位 `10` 决定它们落在 2 号桶，现在 B 变成 3，所以 `010` 、 `110` 分别落入 2、6 号桶。




理解了这个，后面讲 map 迭代的时候会用到。

再来讲搬迁函数中的几个关键点：

evacuate 函数每次只完成一个 bucket 的搬迁工作，因此要遍历完此 bucket 的所有的 cell，将有值的 cell copy 到新的地方。bucket 还会链接 overflow bucket，它们同样需要搬迁。因此会有 2 层循环，外层遍历 bucket 和 overflow bucket，内层遍历 bucket 的所有 cell。这样的循环在 map 的源码里到处都是，要理解透了。

源码里提到 X, Y part，其实就是我们说的如果是扩容到原来的 2 倍，桶的数量是原来的 2 倍，前一半桶被称为 X part，后一半桶被称为 Y part。一个 bucket 中的 key 可能会分裂落到 2 个桶，一个位于 X part，一个位于 Y part。所以在搬迁一个 cell 之前，需要知道这个 cell 中的 key 是落到哪个 Part。很简单，重新计算 cell 中 key 的 hash，并向前“多看”一位，决定落入哪个 Part，这个前面也说得很详细了。

有一个特殊情况是：有一种 key，每次对它计算 hash，得到的结果都不一样。这个 key 就是 `math.NaN()` 的结果，它的含义是 `not a number`，类型是 float64。当它作为 map 的 key，在搬迁的时候，会遇到一个问题：再次计算它的哈希值和它当初插入 map 时的计算出来的哈希值不一样！

你可能想到了，这样带来的一个后果是，这个 key 是永远不会被 Get 操作获取的！当我使用 `m[math.NaN()]` 语句的时候，是查不出来结果的。这个 key 只有在遍历整个 map 的时候，才有机会现身。所以，可以向一个 map 插入任意数量的 `math.NaN()` 作为 key。

当搬迁碰到 `math.NaN()` 的 key 时，只通过 tophash 的最低位决定分配到 X part 还是 Y part（如果扩容后是原来 buckets 数量的 2 倍）。如果 tophash 的最低位是 0 ，分配到 X part；如果是 1 ，则分配到 Y part。

这是通过 tophash 值与新算出来的哈希值进行运算得到的：

```
if top&1 != 0 {
    // top hash 最低位为 1
    // 新算出来的 hash 值的 B 位置 1
    hash |= newbit
} else {
    // 新算出来的 hash 值的 B 位置 0
    hash &^= newbit
}

// hash 值的 B 位为 0，则搬迁到 x part
// 当 B = 5时，newbit = 32，二进制低 6 位为 10 0000
useX = hash&newbit == 0
```

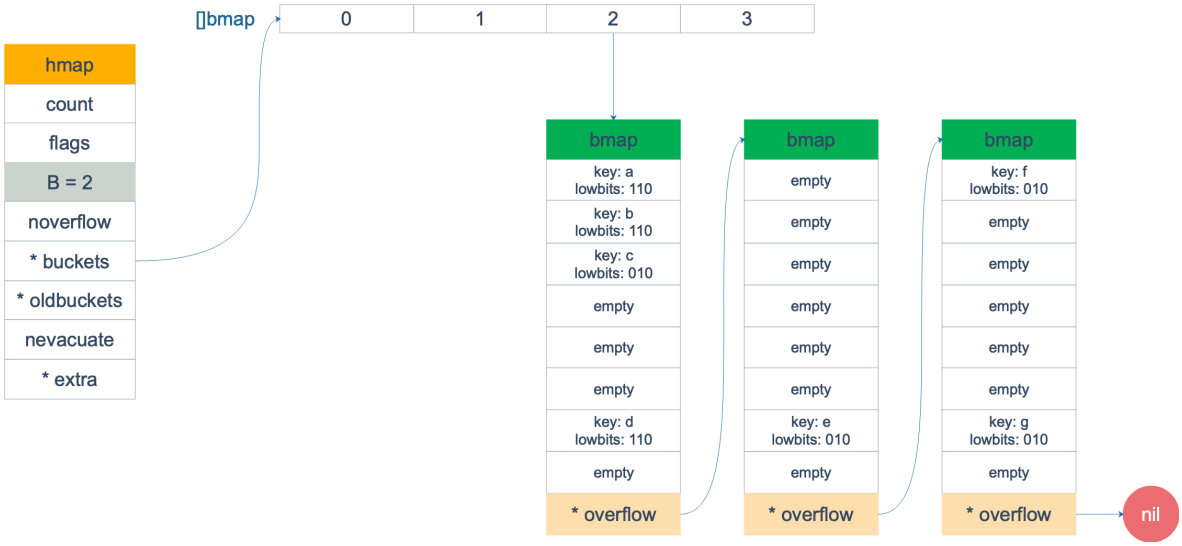
其实这样的 key 我随便搬迁到哪个 bucket 都行，当然，还是要搬迁到上面裂变那张图中的两个 bucket 中去。但这样做是有好处的，在后面讲 map 迭代的时候会再详细解释，暂时知道是这样分配的就行。

确定了要搬迁到的目标 bucket 后，搬迁操作就比较好进行了。将源 key/value 值 copy 到目的地相应的位置。

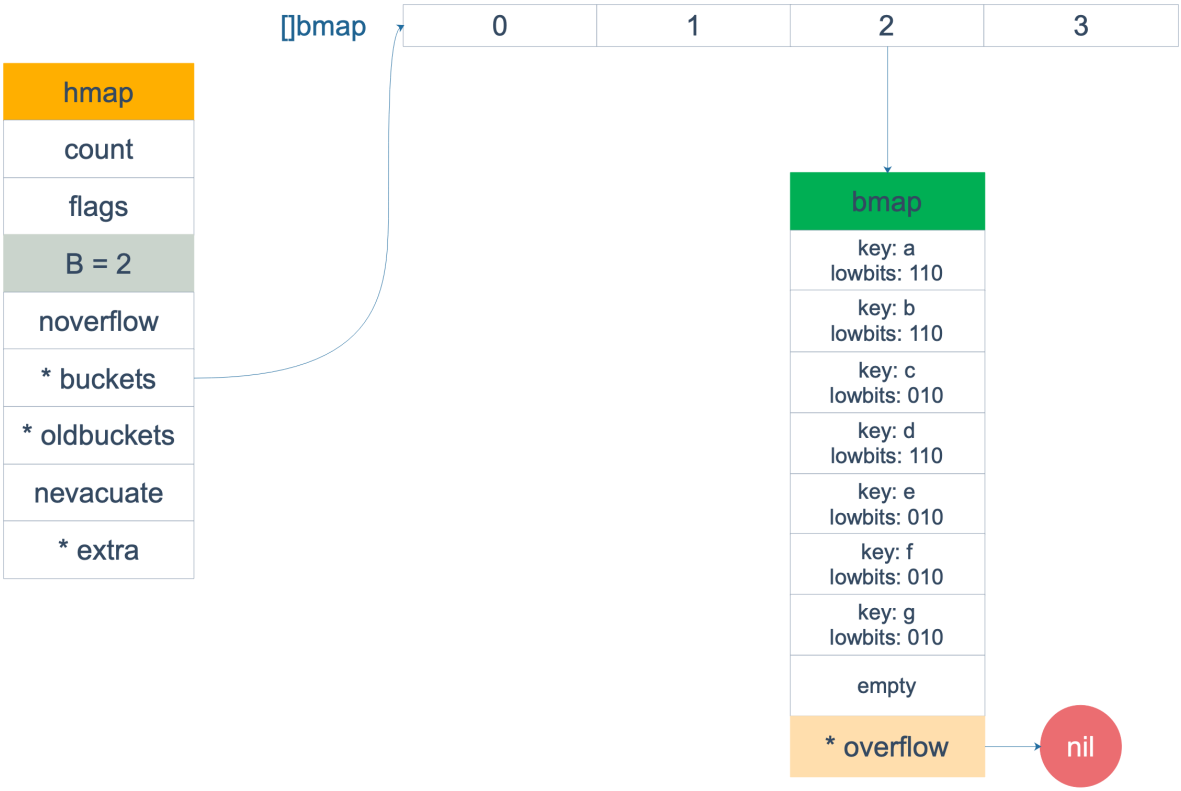
设置 key 在原始 buckets 的 tophash 为 `evacuatedX` 或是 `evacuatedY`，表示已经搬迁到了新 map 的 x part 或是 y part。新 map 的 tophash 则正常取 key 哈希值的高 8 位。

下面通过图来宏观地看一下扩容前后的变化。

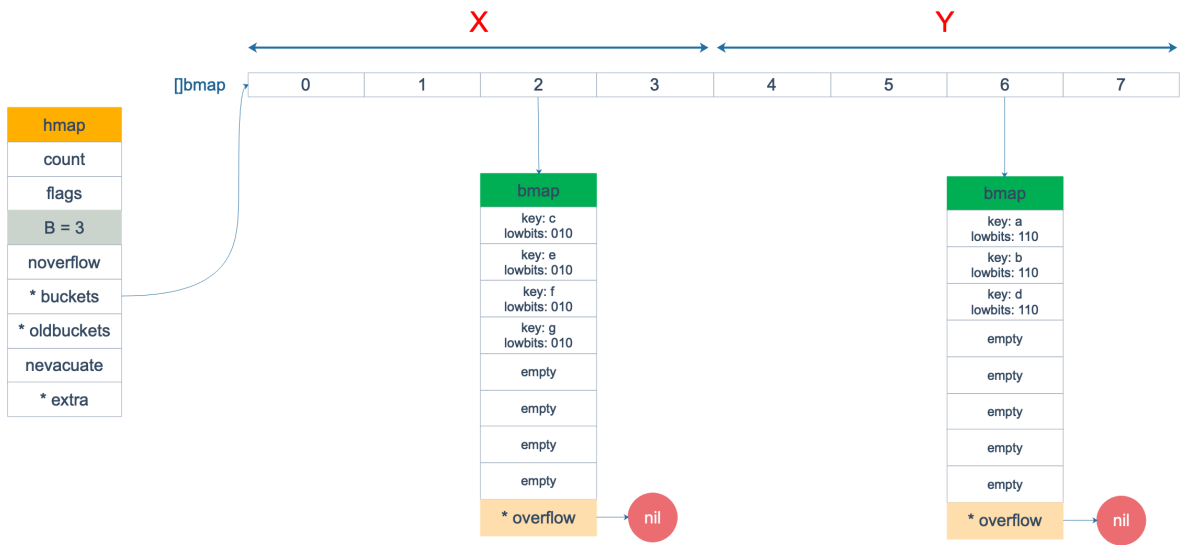
扩容前，B = 2，共有 4 个 buckets，lowbits 表示 hash 值的低位。假设我们不关注其他 buckets 情况，专注在 2 号 bucket。并且假设 overflow 太多，触发了等量扩容（对应于前面的条件 2）。




扩容完成后，overflow bucket 消失了，key 都集中到了一个 bucket，更为紧凑了，提高了查找的效率。




假设触发了 2 倍的扩容，那么扩容完成后，老 buckets 中的 key 分裂到了 2 个 新的 bucket。一个在 x part，一个在 y 的 part。依据是 hash 的 lowbits。新 map 中 `0-3` 称为 x part，`4-7` 称为 y part。




注意，上面的两张图忽略了其他 buckets 的搬迁情况，表示所有的 bucket 都搬迁完毕后的情形。实际上，我们知道，搬迁是一个“渐进”的过程，并不会一下子就全部搬迁完毕。所以在搬迁过程中，oldbuckets 指针还会指向原来老的 []bmap，并且已经搬迁完毕的 key 的 tophash 值会是一个状态值，表示 key 的搬迁去向。

# interface

# iface 和 eface 的区别是什么

`iface` 和 `eface` 都是 Go 中描述接口的底层结构体，区别在于 `iface` 描述的接口包含方法，而 `eface` 则是不包含任何方法的空接口：`interface{}`。

从源码层面看一下：

```
type iface struct {
    tab  *itab
    data unsafe.Pointer
}

type itab struct {
    inter  *interfacetype
    _type  *_type
    link   *itab
    hash   uint32 // copy of _type.hash. Used for type switches.
    bad    bool   // type does not implement interface
    inhash bool   // has this itab been added to hash?
    unused [2]byte
    fun    [1]uintptr // variable sized
}
```

`iface` 内部维护两个指针，`tab` 指向一个 `itab` 实体， 它表示接口的类型以及赋给这个接口的实体类型。`data` 则指向接口具体的值，一般而言是一个指向堆内存的指针。

再来仔细看一下 `itab` 结构体：`_type` 字段描述了实体的类型，包括内存对齐方式，大小等；`inter` 字段则描述了接口的类型。`fun` 字段放置和接口方法对应的具体数据类型的方法地址，实现接口调用方法的动态分派，一般在每次给接口赋值发生转换时会更新此表，或者直接拿缓存的 itab。

这里只会列出实体类型和接口相关的方法，实体类型的其他方法并不会出现在这里。如果你学过 C++ 的话，这里可以类比虚函数的概念。

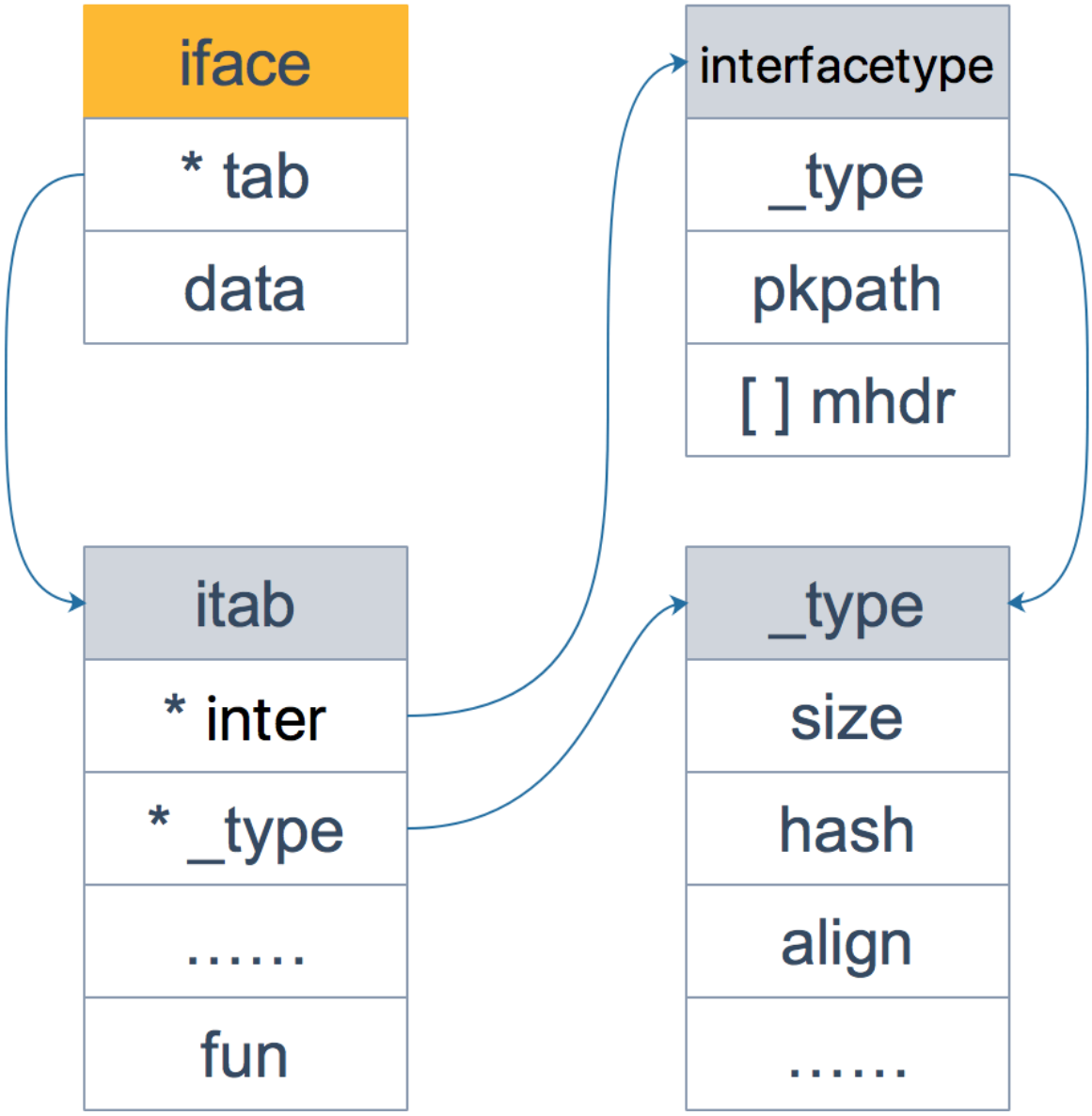
另外，你可能会觉得奇怪，为什么 `fun` 数组的大小为 1，要是接口定义了多个方法可怎么办？实际上，这里存储的是第一个方法的函数指针，如果有更多的方法，在它之后的内存空间里继续存储。从汇编角度来看，通过增加地址就能获取到这些函数指针，没什么影响。顺便提一句，这些方法是按照函数名称的字典序进行排列的。

再看一下 `interfacetype` 类型，它描述的是接口的类型：

```
type interfacetype struct {
    typ     _type
    pkgpath name
    mhdr    []imethod
}
```

可以看到，它包装了 `_type` 类型，`_type` 实际上是描述 Go 语言中各种数据类型的结构体。我们注意到，这里还包含一个 `mhdr` 字段，表示接口所定义的函数列表，`pkgpath` 记录定义了接口的包名。

这里通过一张图来看下 `iface` 结构体的全貌：




接着来看一下 `eface` 的源码：

```
type eface struct {
    _type *_type
    data  unsafe.Pointer
}
```

相比 `iface` ， `eface` 就比较简单了。只维护了一个 `_type` 字段，表示空接口所承载的具体的实体类型。 `data` 描述了具体的值。


我们来看个例子：

```
package main

import "fmt"

func main() {
    x := 200
    var any interface{} = x
    fmt.Println(any)

    g := Gopher{"Go"}
    var c coder = g
    fmt.Println(c)
}

type coder interface {
    code()
    debug()
}

type Gopher struct {
    language string
}

func (p Gopher) code() {
    fmt.Printf("I am coding %s language\n", p.language)
}

func (p Gopher) debug() {
    fmt.Printf("I am debuging %s language\n", p.language)
}
```

执行命令，打印出汇编语言：

```
go tool compile -S ./src/main.go
```

可以看到，main 函数里调用了两个函数：

```
func convT2E64(t *_type, elem unsafe.Pointer) (e eface)
func convT2I(tab *itab, elem unsafe.Pointer) (i iface)
```

上面两个函数的参数和 `iface` 及 `eface` 结构体的字段是可以联系起来的：两个函数都是将参数 `组装` 一下，形成最终的接口。

作为补充，我们最后再来看下 `_type` 结构体：

```
type _type struct {
    // 类型大小
    size       uintptr
    ptrdata    uintptr
    // 类型的 hash 值
    hash       uint32
    // 类型的 flag，和反射相关
    tflag      tflag
    // 内存对齐相关
    align      uint8
    fieldalign uint8
    // 类型的编号，有bool, slice, struct 等等等等
    kind       uint8
    alg        *typeAlg
    // gc 相关
    gcdata    *byte
    str       nameOff
    ptrToThis typeOff
}
```

Go 语言各种数据类型都是在 `_type` 字段的基础上，增加一些额外的字段来进行管理的：

```
type arraytype struct {
    typ   _type
    elem  *_type
    slice *_type
    len   uintptr
}

type chantype struct {
    typ  _type
    elem *_type
    dir  uintptr
}

type slicetype struct {
    typ  _type
    elem *_type
}

type structtype struct {
    typ     _type
    pkgPath name
    fields  []structfield
}
```

这些数据类型的结构体定义，是反射实现的基础。

# 参考资料

【有汇编分析，不错】http://legendtkl.com/2017/07/01/golang-interface-implement/

【interface 源码解读 很不错 包含反射】http://wudaijun.com/2018/01/go-interface-implement/

# Go 接口与 C++ 接口有何异同

接口定义了一种规范，描述了类的行为和功能，而不做具体实现。

C++ 的接口是使用抽象类来实现的，如果类中至少有一个函数被声明为纯虚函数，则这个类就是抽象类。纯虚函数是通过在声明中使用 “= 0” 来指定的。例如：

```
class Shape
{
   public:
      // 纯虚函数
      virtual double getArea() = 0;
   private:
      string name;      // 名称
};
```

设计抽象类的目的，是为了给其他类提供一个可以继承的适当的基类。抽象类不能被用于实例化对象，它只能作为接口使用。

派生类需要明确地声明它继承自基类，并且需要实现基类中所有的纯虚函数。

C++ 定义接口的方式称为“侵入式”，而 Go 采用的是 “非侵入式”，不需要显式声明，只需要实现接口定义的函数，编译器自动会识别。

C++ 和 Go 在定义接口方式上的不同，也导致了底层实现上的不同。C++ 通过虚函数表来实现基类调用派生类的函数；而 Go 通过 `itab` 中的 `fun` 字段来实现接口变量调用实体类型的函数。C++ 中的虚函数表是在编译期生成的；而 Go 的 `itab` 中的 `fun` 字段是在运行期间动态生成的。原因在于，Go 中实体类型可能会无意中实现 N 多接口，很多接口并不是本来需要的，所以不能为类型实现的所有接口都生成一个 `itab` ， 这也是“非侵入式”带来的影响；这在 C++ 中是不存在的，因为派生需要显示声明它继承自哪个基类。

## 参考资料

【和 C++ 的对比】https://www.jianshu.com/p/b38b1719636e

# 接口转换的原理

通过前面提到的 `iface` 的源码可以看到，实际上它包含接口的类型 `interfacetype` 和 实体类型的类型 `_type` ，这两者都是 `iface` 的字段 `itab` 的成员。也就是说生成一个 `itab` 同时需要接口的类型和实体的类型。

> <interface 类型， 实体类型> ->itable

当判定一种类型是否满足某个接口时，Go 使用类型的方法集和接口所需要的方法集进行匹配，如果类型的方法集完全包含接口的方法集，则可认为该类型实现了该接口。

例如某类型有 `m` 个方法，某接口有 `n` 个方法，则很容易知道这种判定的时间复杂度为 `O(mn)` ，Go 会对方法集的函数按照函数名的字典序进行排序，所以实际的时间复杂度为 `O(m+n)` 。

这里我们来探索将一个接口转换给另外一个接口背后的原理，当然，能转换的原因必然是类型兼容。

直接来看一个例子：

```
package main

import "fmt"

type coder interface {
    code()
    run()
}

type runner interface {
    run()
}

type Gopher struct {
    language string
}

func (g Gopher) code() {
    return
}

func (g Gopher) run() {
    return
}

func main() {
    var c coder = Gopher{}

    var r runner
    r = c
    fmt.Println(c, r)
}
```

简单解释下上述代码：定义了两个 `interface` ： `coder` 和 `runner` 。定义了一个实体类型 `Gopher` ，类型 `Gopher` 实现了两个方法，分别是 `run()` 和 `code()` 。main 函数里定义了一个接口变量 `c` ，绑定了一个 `Gopher` 对象，之后将 `c` 赋值给另外一个接口变量 `r` 。赋值成功的原因是 `c` 中包含 `run()` 方法。这样，两个接口变量完成了转换。

执行命令：

```
go tool compile -S ./src/main.go
```

得到 main 函数的汇编命令，可以看到： `r = c` 这一行语句实际上是调用了 `runtime.convI2I(SB)` ，也就是 `convI2I` 函数，从函数名来看，就是将一个 `interface` 转换成另外一个 `interface` ，看下它的源代码：

```
func convI2I(inter *interfacetype, i iface) (r iface) {
    tab := i.tab
    if tab == nil {
        return
    }
    if tab.inter == inter {
        r.tab = tab
        r.data = i.data
        return
    }
    r.tab = getitab(inter, tab._type, false)
    r.data = i.data
    return
}
```

代码比较简单，函数参数 `inter` 表示接口类型， `i` 表示绑定了实体类型的接口， `r` 则表示接口转换了之后的新的 `iface` 。通过前面的分析，我们又知道， `iface` 是由 `tab` 和 `data` 两个字段组成。所以，实际上 `convI2I` 函数真正要做的事，找到新 `interface` 的 `tab` 和 `data` ，就大功告成了。

我们还知道， `tab` 是由接口类型 `interfacetype` 和 实体类型 `_type` 。所以最关键的语句是 `r.tab = getitab(inter, tab._type, false)` 。

因此，重点来看下 `getitab` 函数的源码，只看关键的地方：

```
func getitab(inter *interfacetype, typ *_type, canfail bool) *itab {
    // ......

    // 根据 inter, typ 计算出 hash 值
    h := itabhash(inter, typ)

    // look twice - once without lock, once with.
    // common case will be no lock contention.
    var m *itab
    var locked int
    for locked = 0; locked < 2; locked++ {
        if locked != 0 {
            lock(&ifaceLock)
        }

        // 遍历哈希表的一个 slot
        for m = (*itab)(atomic.Loadp(unsafe.Pointer(&hash[h]))); m != nil; m = m.link {

            // 如果在 hash 表中已经找到了 itab（inter 和 typ 指针都相同）
            if m.inter == inter && m._type == typ {
                // ......

                if locked != 0 {
                    unlock(&ifaceLock)
                }
                return m
            }
        }
    }
```

```
    // 在 hash 表中没有找到 itab，那么新生成一个 itab
    m = (*itab)(persistentalloc(unsafe.Sizeof(itab{})+uintptr(len(inter.mhdr)-1)*sys.PtrSize, 0, &memstats.ot
her_sys))
    m.inter = inter
    m._type = typ

    // 添加到全局的 hash 表中
    additab(m, true, canfail)
    unlock(&ifaceLock)
    if m.bad {
        return nil
    }
    return m
}
```

简单总结一下：getitab 函数会根据 `interfacetype` 和 `_type` 去全局的 itab 哈希表中查找，如果能找到，则直接返回；否则，会根据给定的 `interfacetype` 和 `_type` 新生成一个 `itab`，并插入到 itab 哈希表，这样下一次就可以直接拿到 `itab`。

这里查找了两次，并且第二次上锁了，这是因为如果第一次没找到，在第二次仍然没有找到相应的 `itab` 的情况下，需要新生成一个，并且写入哈希表，因此需要加锁。这样，其他协程在查找相同的 `itab` 并且也没有找到时，第二次查找时，会被挂住，之后，就会查到第一个协程写入哈希表的 `itab`。

再来看一下 `additab` 函数的代码：

```
// 检查 _type 是否符合 interface_type 并且创建对应的 itab 结构体 将其放到 hash 表中
func additab(m *itab, locked, canfail bool) {
    inter := m.inter
    typ := m._type
    x := typ.uncommon()

    // both inter and typ have method sorted by name,
    // and interface names are unique,
    // so can iterate over both in lock step;
    // the loop is O(ni+nt) not O(ni*nt).
    //
    // inter 和 typ 的方法都按方法名称进行了排序
    // 并且方法名都是唯一的。所以循环的次数是固定的
    // 只用循环 O(ni+nt)，而非 O(ni*nt)
    ni := len(inter.mhdr)
    nt := int(x.mcount)
    xmhdr := (*[1 << 16]method)(add(unsafe.Pointer(x), uintptr(x.moff)))[:nt:nt]
    j := 0
    for k := 0; k < ni; k++ {
        i := &inter.mhdr[k]
        itype := inter.typ.typeOff(i.ityp)
        name := inter.typ.nameOff(i.name)
        iname := name.name()
        ipkg := name.pkgPath()
        if ipkg == "" {
            ipkg = inter.pkgpath.name()
        }
        for ; j < nt; j++ {
            t := &xmhdr[j]
            tname := typ.nameOff(t.name)
            // 检查方法名字是否一致
            if typ.typeOff(t.mtyp) == itype && tname.name() == iname {
                pkgPath := tname.pkgPath()
                if pkgPath == "" {
```

```
                    pkgPath = typ.nameOff(x.pkgpath).name()
                }
                if tname.isExported() || pkgPath == ipkg {
                    if m != nil {
                        // 获取函数地址，并加入到itab.fun数组中
                        ifn := typ.textOff(t.ifn)
                        *(*unsafe.Pointer)(add(unsafe.Pointer(&m.fun[0]), uintptr(k)*sys.PtrSize)) = ifn
                    }
                    goto nextimethod
                }
            }
        }
        // ......

        m.bad = true
        break
    nextimethod:
    }
    if !locked {
        throw("invalid itab locking")
    }

    // 计算 hash 值
    h := itabhash(inter, typ)
    // 加到Hash Slot链表中
    m.link = hash[h]
    m.inhash = true
    atomicstorep(unsafe.Pointer(&hash[h]), unsafe.Pointer(m))
}
```

`additab` 会检查 `itab` 持有的 `interfacetype` 和 `_type` 是否符合，就是看 `_type` 是否完全实现了 `interfacetype` 的方法，也就是看两者的方法列表重叠的部分就是 `interfacetype` 所持有的方法列表。注意到其中有一个双层循环，乍一看，循环次数是 `ni * nt`，但由于两者的函数列表都按照函数名称进行了排序，因此最终只执行了 `ni + nt` 次，代码里通过一个小技巧来实现：第二层循环并没有从 0 开始计数，而是从上一次遍历到的位置开始。

求 hash 值的函数比较简单：

```
func itabhash(inter *interfacetype, typ *_type) uint32 {
    h := inter.typ.hash
    h += 17 * typ.hash
    return h % hashSize
}
```

`hashSize` 的值是 1009。

更一般的，当把实体类型赋值给接口的时候，会调用 `conv` 系列函数，例如空接口调用 `convT2E` 系列、非空接口调用 `convT2I` 系列。这些函数比较相似：

1. 具体类型转空接口时，_type 字段直接复制源类型的 _type；调用 mallocgc 获得一块新内存，把值复制进去，data 再指向这块新内存。
2. 具体类型转非空接口时，入参 tab 是编译器在编译阶段预先生成好的，新接口 tab 字段直接指向入参 tab 指向的 itab；调用 mallocgc 获得一块新内存，把值复制进去，data 再指向这块新内存。
3. 而对于接口转接口，itab 调用 getitab 函数获取。只用生成一次，之后直接从 hash 表中获取。

# 参考资料

【接口赋值、反射】http://wudaijun.com/2018/01/go-interface-implement/

【itab】http://legendtkl.com/2017/07/01/golang-interface-implement/

【和 C++ 的对比】https://www.jianshu.com/p/b38b1719636e

【itab 原理】https://ninokop.github.io/2017/10/29/Go-%E6%96%B9%E6%B3%95%E8%B0%83%E7%94%A8%E4%B8%8E%E6%8E%A5%E5%8F%A3/

【getitab源码说明】https://www.twblogs.net/a/5c245d59bd9eee16b3db561d

# 如何用 interface 实现多态

`Go` 语言并没有设计诸如虚函数、纯虚函数、继承、多重继承等概念，但它通过接口却非常优雅地支持了面向对象的特性。

多态是一种运行期的行为，它有以下几个特点：

> 1. 一种类型具有多种类型的能力
> 2. 允许不同的对象对同一消息做出灵活的反应
> 3. 以一种通用的方式对待个使用的对象
> 4. 非动态语言必须通过继承和接口的方式来实现

看一个实现了多态的代码例子：

```
package main

import "fmt"

func main() {
    qcrao := Student{age: 18}
    whatJob(&qcrao)

    growUp(&qcrao)
    fmt.Println(qcrao)

    stefno := Programmer{age: 100}
    whatJob(stefno)

    growUp(stefno)
    fmt.Println(stefno)
}

func whatJob(p Person) {
    p.job()
}

func growUp(p Person) {
    p.growUp()
}

type Person interface {
    job()
    growUp()
}

type Student struct {
    age int
}

func (p Student) job() {
    fmt.Println("I am a student.")
    return
}

func (p *Student) growUp() {
    p.age += 1
    return
}
```

```
type Programmer struct {
    age int
}

func (p Programmer) job() {
    fmt.Println("I am a programmer.")
    return
}

func (p Programmer) growUp() {
    // 程序员老得太快 ^_^
    p.age += 10
    return
}
```

代码里先定义了 1 个 `Person` 接口，包含两个函数：

```
job()
growUp()
```

然后，又定义了 2 个结构体，`Student` 和 `Programmer`，同时，类型 `*Student`、`Programmer` 实现了 `Person` 接口定义的两个函数。注意，`*Student` 类型实现了接口，`Student` 类型却没有。

之后，我又定义了函数参数是 `Person` 接口的两个函数：

```
func whatJob(p Person)
func growUp(p Person)
```

`main` 函数里先生成 `Student` 和 `Programmer` 的对象，再将它们分别传入到函数 `whatJob` 和 `growUp`。函数中，直接调用接口函数，实际执行的时候是看最终传入的实体类型是什么，调用的是实体类型实现的函数。于是，不同对象针对同一消息就有多种表现，`多态` 就实现了。

更深入一点来说的话，在函数 `whatJob()` 或者 `growUp()` 内部，接口 `person` 绑定了实体类型 `*Student` 或者 `Programmer`。根据前面分析的 `iface` 源码，这里会直接调用 `fun` 里保存的函数，类似于：`s.tab->fun[0]`，而因为 `fun` 数组里保存的是实体类型实现的函数，所以当函数传入不同的实体类型时，调用的实际上是不同的函数实现，从而实现多态。

运行一下代码：

```
I am a student.
{19}
I am a programmer.
{100}
```

# 参考资料

【各种面向对象的名词】https://cyent.github.io/golang/other/oo/

【多态与鸭子类型】https://www.jb51.net/article/116025.htm

# Go 语言与鸭子类型的关系

先直接来看维基百科里的定义：

> If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.

翻译过来就是：如果某个东西长得像鸭子，像鸭子一样游泳，像鸭子一样嘎嘎叫，那它就可以被看成是一只鸭子。

`Duck Typing`，鸭子类型，是动态编程语言的一种对象推断策略，它更关注对象能如何被使用，而不是对象的类型本身。Go 语言作为一门静态语言，它通过通过接口的方式完美支持鸭子类型。

例如，在动态语言 python 中，定义一个这样的函数：

```
def hello_world(coder):
    coder.say_hello()
```

当调用此函数的时候，可以传入任意类型，只要它实现了 `say_hello()` 函数就可以。如果没有实现，运行过程中会出现错误。

而在静态语言如 Java, C++ 中，必须要显示地声明实现了某个接口，之后，才能用在任何需要这个接口的地方。如果你在程序中调用 `hello_world` 函数，却传入了一个根本就没有实现 `say_hello()` 的类型，那在编译阶段就不会通过。这也是静态语言比动态语言更安全的原因。

动态语言和静态语言的差别在此就有所体现。静态语言在编译期间就能发现类型不匹配的错误，不像动态语言，必须要运行到那一行代码才会报错。插一句，这也是我不喜欢用 `python` 的一个原因。当然，静态语言要求程序员在编码阶段就要按照规定来编写程序，为每个变量规定数据类型，这在某种程度上，加大了工作量，也加长了代码量。动态语言则没有这些要求，可以让人更专注在业务上，代码也更短，写起来更快，这一点，写 python 的同学比较清楚。

Go 语言作为一门现代静态语言，是有后发优势的。它引入了动态语言的便利，同时又会进行静态语言的类型检查，写起来是非常 Happy 的。Go 采用了折中的做法：不要求类型显示地声明实现了某个接口，只要实现了相关的方法即可，编译器就能检测到。

来看个例子：

先定义一个接口，和使用此接口作为参数的函数：

```
type IGreeting interface {
    sayHello()
}

func sayHello(i IGreeting) {
    i.sayHello()
}
```

再来定义两个结构体：

```
type Go struct {}
func (g Go) sayHello() {
    fmt.Println("Hi, I am GO!")
}

type PHP struct {}
func (p PHP) sayHello() {
    fmt.Println("Hi, I am PHP!")
}
```

最后，在 main 函数里调用 sayHello() 函数：

```
func main() {
    golang := Go{}
    php := PHP{}

    sayHello(golang)
    sayHello(php)
}
```

程序输出：

```
Hi, I am GO!
Hi, I am PHP!
```

在 main 函数中，调用调用 sayHello() 函数时，传入了 `golang, php` 对象，它们并没有显式地声明实现了 IGreeting 类型，只是实现了接口所规定的 sayHello() 函数。实际上，编译器在调用 sayHello() 函数时，会隐式地将 `golang, php` 对象转换成 IGreeting 类型，这也是静态语言的类型检查功能。

顺带再提一下动态语言的特点：

> 变量绑定的类型是不确定的，在运行期间才能确定
> 函数和方法可以接收任何类型的参数，且调用时不检查参数类型
> 不需要实现接口

总结一下，鸭子类型是一种动态语言的风格，在这种风格中，一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由它"当前方法和属性的集合"决定。Go 作为一种静态语言，通过接口实现了 `鸭子类型` ，实际上是 Go 的编译器在其中作了隐匿的转换工作。

# 参考资料

【wikipedia】https://en.wikipedia.org/wiki/Duck_test

【Golang 与鸭子类型，讲得比较好】https://blog.csdn.net/cszhouwei/article/details/33741731

【各种面向对象的名词】https://cyent.github.io/golang/other/oo/

【多态、鸭子类型特性】https://www.jb51.net/article/116025.htm

【鸭子类型、动态静态语言】https://www.jianshu.com/p/650485b78d11

# 值接收者和指针接收者的区别

## 方法

方法能给用户自定义的类型添加新的行为。它和函数的区别在于方法有一个接收者，给一个函数添加一个接收者，那么它就变成了方法。接收者可以是 `值接收者` ，也可以是 `指针接收者` 。

在调用方法的时候，值类型既可以调用 `值接收者` 的方法，也可以调用 `指针接收者` 的方法；指针类型既可以调用 `指针接收者` 的方法，也可以调用 `值接收者` 的方法。

也就是说，不管方法的接收者是什么类型，该类型的值和指针都可以调用，不必严格符合接收者的类型。

来看个例子：

```
package main

import "fmt"

type Person struct {
    age int
}

func (p Person) howOld() int {
    return p.age
}

func (p *Person) growUp() {
    p.age += 1
}

func main() {
    // qcrao 是值类型
    qcrao := Person{age: 18}

    // 值类型 调用接收者也是值类型的方法
    fmt.Println(qcrao.howOld())

    // 值类型 调用接收者是指针类型的方法
    qcrao.growUp()
    fmt.Println(qcrao.howOld())

    // ----------------------

    // stefno 是指针类型
    stefno := &Person{age: 100}

    // 指针类型 调用接收者是值类型的方法
    fmt.Println(stefno.howOld())

    // 指针类型 调用接收者也是指针类型的方法
    stefno.growUp()
    fmt.Println(stefno.howOld())
}
```

上例子的输出结果是：

```
18
19
100
101
```

调用了 `growUp` 函数后，不管调用者是值类型还是指针类型，它的 `Age` 值都改变了。

实际上，当类型和方法的接收者类型不同时，其实是编译器在背后做了一些工作，用一个表格来呈现：

| - | 值接收者 | 指针接收者 |
| --- | --- | --- |
| 值类型调用者 | 方法会使用调用者的一个副本，类似于“传值” | 使用值的引用来调用方法，上例中，`qcrao.growUp()` 实际上是 `(&qcrao).growUp()` |
| 指针类型调用者 | 指针被解引用为值，上例中，`stefno.how0ld()` 实际上是 `(*stefno).how0ld()` | 实际上也是“传值”，方法里的操作会影响到调用者，类似于指针传参，拷贝了一份指针 |

# 值接收者和指针接收者

前面说过，不管接收者类型是值类型还是指针类型，都可以通过值类型或指针类型调用，这里面实际上通过语法糖起作用的。

先说结论：实现了接收者是值类型的方法，相当于自动实现了接收者是指针类型的方法；而实现了接收者是指针类型的方法，不会自动生成对应接收者是值类型的方法。

来看一个例子，就会完全明白：

```
package main

import "fmt"

type coder interface {
    code()
    debug()
}

type Gopher struct {
    language string
}

func (p Gopher) code() {
    fmt.Printf("I am coding %s language\n", p.language)
}

func (p *Gopher) debug() {
    fmt.Printf("I am debuging %s language\n", p.language)
}

func main() {
    var c coder = &Gopher{"Go"}
    c.code()
```

```
    c.debug()
}
```

上述代码里定义了一个接口 `coder` ，接口定义了两个函数：

```
code()
debug()
```

接着定义了一个结构体 `Gopher` ，它实现了两个方法，一个值接收者，一个指针接收者。

最后，我们在 `main` 函数里通过接口类型的变量调用了定义的两个函数。

运行一下，结果：

```
I am coding Go language
I am debuging Go language
```

但是如果我们把 `main` 函数的第一条语句换一下：

```
func main() {
    var c coder = Gopher{"Go"}
    c.code()
    c.debug()
}
```

运行一下，报错：

```
src/main.go:23:6: cannot use Gopher literal (type Gopher) as type coder in assignment:
    Gopher does not implement coder (debug method has pointer receiver)
```

看出这两处代码的差别了吗？第一次是将 `&Gopher` 赋给了 `coder` ；第二次则是将 `Gopher` 赋给了 `coder` 。

第二次报错是说， `Gopher` 没有实现 `coder` 。很明显了吧，因为 `Gopher` 类型并没有实现 `debug` 方法；表面上看， `*Gopher` 类型也没有实现 `code` 方法，但是因为 `Gopher` 类型实现了 `code` 方法，所以让 `*Gopher` 类型自动拥有了 `code` 方法。

当然，上面的说法有一个简单的解释：接收者是指针类型的方法，很可能在方法中会对接收者的属性进行更改操作，从而影响接收者；而对于接收者是值类型的方法，在方法中不会对接收者本身产生影响。

所以，当实现了一个接收者是值类型的方法，就可以自动生成一个接收者是对应指针类型的方法，因为两者都不会影响接收者。但是，当实现了一个接收者是指针类型的方法，如果此时自动生成一个接收者是值类型的方法，原本期望对接收者的改变（通过指针实现），现在无法实现，因为值类型会产生一个拷贝，不会真正影响调用者。

最后，只要记住下面这点就可以了：

> 如果实现了接收者是值类型的方法，会隐含地也实现了接收者是指针类型的方法。

# 两者分别在何时使用

如果方法的接收者是值类型，无论调用者是对象还是对象指针，修改的都是对象的副本，不影响调用者；如果方法的接收者是指针类型，则调用者修改的是指针指向的对象本身。

使用指针作为方法的接收者的理由：

- 方法能够修改接收者指向的值。
- 避免在每次调用方法时复制该值，在值的类型为大型结构体时，这样做会更加高效。

是使用值接收者还是指针接收者，不是由该方法是否修改了调用者（也就是接收者）来决定，而是应该基于该类型的 `本质` 。

如果类型具备“原始的本质”，也就是说它的成员都是由 Go 语言里内置的原始类型，如字符串，整型值等，那就定义值接收者类型的方法。像内置的引用类型，如 slice，map，interface，channel，这些类型比较特殊，声明他们的时候，实际上是创建了一个 `header` ， 对于他们也是直接定义值接收者类型的方法。这样，调用函数时，是直接 copy 了这些类型的 `header` ，而 `header` 本身就是为复制设计的。

如果类型具备非原始的本质，不能被安全地复制，这种类型总是应该被共享，那就定义指针接收者的方法。比如 go 源码里的文件结构体（struct File）就不应该被复制，应该只有一份 `实体` 。

这一段说的比较绕，大家可以去看《Go 语言实战》5.3 那一节。

# 参考资料

【飞雪无情 Go实战笔记】https://www.flysnow.org/2017/04/03/go-in-action-go-interface.html

【何时使用指针接收者】http://ironxu.com/711

【理解Go Interface】http://lanlingzi.cn/post/technical/2016/0803_go_interface/

【Go语言实战 类型的本置】 图书《Go In Action》

# 接口的构造过程是怎样的

我们已经看过了 `iface` 和 `eface` 的源码，知道 `iface` 最重要的是 `itab` 和 `_type` 。

为了研究清楚接口是如何构造的，接下来我会拿起汇编的武器，还原背后的真相。

来看一个示例代码：

```
package main

import "fmt"

type Person interface {
    growUp()
}

type Student struct {
    age int
}

func (p Student) growUp() {
    p.age += 1
    return
}

func main() {
    var qcrao = Person(Student{age: 18})

    fmt.Println(qcrao)
}
```

执行命令：

```
go tool compile -S main.go
```

得到 main 函数的汇编代码如下：

```
0x0000 00000 (./src/main.go:30) TEXT    "".main(SB), $80-0
0x0000 00000 (./src/main.go:30) MOVQ    (TLS), CX
0x0009 00009 (./src/main.go:30) CMPQ    SP, 16(CX)
0x000d 00013 (./src/main.go:30) JLS     157
0x0013 00019 (./src/main.go:30) SUBQ    $80, SP
0x0017 00023 (./src/main.go:30) MOVQ    BP, 72(SP)
0x001c 00028 (./src/main.go:30) LEAQ    72(SP), BP
0x0021 00033 (./src/main.go:30) FUNCDATA$0, gclocals·69c1753bd5f81501d95132d08af04464(SB)
0x0021 00033 (./src/main.go:30) FUNCDATA$1, gclocals·e226d4ae4a7cad8835311c6a4683c14f(SB)
0x0021 00033 (./src/main.go:31) MOVQ    $18, "".autotmp_1+48(SP)
0x002a 00042 (./src/main.go:31) LEAQ    go.itab."".Student,"".Person(SB), AX
0x0031 00049 (./src/main.go:31) MOVQ    AX, (SP)
0x0035 00053 (./src/main.go:31) LEAQ    "".autotmp_1+48(SP), AX
0x003a 00058 (./src/main.go:31) MOVQ    AX, 8(SP)
0x003f 00063 (./src/main.go:31) PCDATA  $0, $0
0x003f 00063 (./src/main.go:31) CALL    runtime.convT2I64(SB)
0x0044 00068 (./src/main.go:31) MOVQ    24(SP), AX
0x0049 00073 (./src/main.go:31) MOVQ    16(SP), CX
0x004e 00078 (./src/main.go:33) TESTQ   CX, CX
```

```
0x0051 00081 (./src/main.go:33) JEQ     87
0x0053 00083 (./src/main.go:33) MOVQ    8(CX), CX
0x0057 00087 (./src/main.go:33) MOVQ    $0, ""..autotmp_2+56(SP)
0x0060 00096 (./src/main.go:33) MOVQ    $0, ""..autotmp_2+64(SP)
0x0069 00105 (./src/main.go:33) MOVQ    CX, ""..autotmp_2+56(SP)
0x006e 00110 (./src/main.go:33) MOVQ    AX, ""..autotmp_2+64(SP)
0x0073 00115 (./src/main.go:33) LEAQ    ""..autotmp_2+56(SP), AX
0x0078 00120 (./src/main.go:33) MOVQ    AX, (SP)
0x007c 00124 (./src/main.go:33) MOVQ    $1, 8(SP)
0x0085 00133 (./src/main.go:33) MOVQ    $1, 16(SP)
0x008e 00142 (./src/main.go:33) PCDATA  $0, $1
0x008e 00142 (./src/main.go:33) CALL    fmt.Println(SB)
0x0093 00147 (./src/main.go:34) MOVQ    72(SP), BP
0x0098 00152 (./src/main.go:34) ADDQ    $80, SP
0x009c 00156 (./src/main.go:34) RET
0x009d 00157 (./src/main.go:34) NOP
0x009d 00157 (./src/main.go:30) PCDATA  $0, $-1
0x009d 00157 (./src/main.go:30) CALL    runtime.morestack_noctxt(SB)
0x00a2 00162 (./src/main.go:30) JMP     0
```

我们从第 10 行开始看，如果不理解前面几行汇编代码的话，可以回去看看公众号前面两篇文章，这里我就省略了。

| 汇编行数 | 操作 |
| --- | --- |
| 10-14 | 构造调用 `runtime.convT2I64(SB)` 的参数 |

我们来看下这个函数的参数形式：

```
func convT2I64(tab *itab, elem unsafe.Pointer) (i iface) {
    // ......
}
```

`convT2I64` 会构造出一个 `inteface` ，也就是我们的 `Person` 接口。

第一个参数的位置是 `(SP)` ，这里被赋上了 `go.itab."".Student,"".Person(SB)` 的地址。

我们从生成的汇编找到：

```
go.itab."".Student,"".Person SNOPTRDATA dupok size=40
        0x0000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
        0x0010 00 00 00 00 00 00 00 00 da 9f 20 d4              ..........
        rel 0+8 t=1 type.""".Person+0
        rel 8+8 t=1 type.""".Student+0
```

`size=40` 大小为40字节，回顾一下：

```
type itab struct {
    inter  *interfacetype // 8字节
    _type  *_type // 8字节
    link   *itab // 8字节
    hash   uint32 // 4字节
    bad    bool   // 1字节
    inhash bool   // 1字节
    unused [2]byte // 2字节
```

```
    fun     [1]uintptr // variable sized // 8字节
}
```

把每个字段的大小相加，`itab` 结构体的大小就是 40 字节。上面那一串数字实际上是 `itab` 序列化后的内容，注意到大部分数字是 0，从 24 字节开始的 4 个字节 `da 9f 20 d4` 实际上是 `itab` 的 `hash` 值，这在判断两个类型是否相同的时候会用到。

下面两行是链接指令，简单说就是将所有源文件综合起来，给每个符号赋予一个全局的位置值。这里的意思也比较明确：前8个字节最终存储的是 `type.*".".Person` 的地址，对应 `itab` 里的 `inter` 字段，表示接口类型；8-16 字节最终存储的是 `type.*".".Student` 的地址，对应 `itab` 里 `_type` 字段，表示具体类型。

第二个参数就比较简单了，它就是数字 `18` 的地址，这也是初始化 `Student` 结构体的时候会用到。

| 汇编行数 | 操作 |
| --- | --- |
| 15 | 调用 `runtime.convT2I64(SB)` |

具体看下代码：

```
func convT2I64(tab *itab, elem unsafe.Pointer) (i iface) {
    t := tab._type

    //...

    var x unsafe.Pointer
    if *(*uint64)(elem) == 0 {
        x = unsafe.Pointer(&zeroVal[0])
    } else {
        x = mallocgc(8, t, false)
        *(*uint64)(x) = *(*uint64)(elem)
    }
    i.tab = tab
    i.data = x
    return
}
```

这块代码比较简单，把 `tab` 赋给了 `iface` 的 `tab` 字段；`data` 部分则是在堆上申请了一块内存，然后将 `elem` 指向的 `18` 拷贝过去。这样 `iface` 就组装好了。

| 汇编行数 | 操作 |
| --- | --- |
| 17 | 把 `i.tab` 赋给 `CX` |
| 18 | 把 `i.data` 赋给 `AX` |
| 19-21 | 检测 `i.tab` 是否是 nil，如果不是的话，把 CX 移动 8 个字节，<br>也就是把 `itab` 的 `_type` 字段赋给了 CX，这也是接口的实体类型，<br>最终要作为 `fmt.Println` 函数的参数 |

后面，就是调用 `fmt.Println` 函数及之前的参数准备工作了，不再赘述。

这样，我们就把一个 `interface` 的构造过程说完了。

【引申1】
如何打印出接口类型的 `Hash` 值？

这里参考曹大神翻译的一篇文章，参考资料里会写上。具体做法如下：

```
type iface struct {
    tab  *itab
    data unsafe.Pointer
}
type itab struct {
    inter uintptr
    _type uintptr
    link uintptr
    hash  uint32
    _     [4]byte
    fun   [1]uintptr
}

func main() {
    var qcrao = Person(Student{age: 18})

    iface := (*iface)(unsafe.Pointer(&qcrao))
    fmt.Printf("iface.tab.hash = %#x\n", iface.tab.hash)
}
```

定义了一个 `山寨版` 的 `iface` 和 `itab`，说它 `山寨` 是因为 `itab` 里的一些关键数据结构都不具体展开了，比如 `_type`，对比一下正宗的定义就可以发现，但是 `山寨版` 依然能工作，因为 `_type` 就是一个指针而已嘛。

在 `main` 函数里，先构造出一个接口对象 `qcrao`，然后强制类型转换，最后读取出 `hash` 值，非常妙！你也可以自己动手试一下。

运行结果：

```
iface.tab.hash = 0xd4209fda
```

值得一提的是，构造接口 `qcrao` 的时候，即使我把 `age` 写成其他值，得到的 `hash` 值依然不变的，这应该是可以预料的，`hash` 值只和他的字段、方法相关。

# 参考资料

【曹大神翻译的文章，非常硬核】http://xargin.com/go-and-interface/#reconstructing-an-itab-from-an-executable

# 编译器自动检测类型是否实现接口

经常看到一些开源库里会有一些类似下面这种奇怪的用法：

```
var _ io.Writer = (*myWriter)(nil)
```

这时候会有点懵，不知道作者想要干什么，实际上这就是此问题的答案。编译器会由此检查 `*myWriter` 类型是否实现了 `io.Writer` 接口。

来看一个例子：

```
package main

import "io"

type myWriter struct {

}

/*func (w myWriter) Write(p []byte) (n int, err error) {
    return
}*/

func main() {
    // 检查 *myWriter 类型是否实现了 io.Writer 接口
    var _ io.Writer = (*myWriter)(nil)

    // 检查 myWriter 类型是否实现了 io.Writer 接口
    var _ io.Writer = myWriter{}
}
```

注释掉为 myWriter 定义的 Write 函数后，运行程序：

```
src/main.go:14:6: cannot use (*myWriter)(nil) (type *myWriter) as type io.Writer in assignment:
    *myWriter does not implement io.Writer (missing Write method)
src/main.go:15:6: cannot use myWriter literal (type myWriter) as type io.Writer in assignment:
    myWriter does not implement io.Writer (missing Write method)
```

报错信息：*myWriter/myWriter 未实现 io.Writer 接口，也就是未实现 Write 方法。

解除注释后，运行程序不报错。

实际上，上述赋值语句会发生隐式地类型转换，在转换的过程中，编译器会检测等号右边的类型是否实现了等号左边接口所规定的函数。

总结一下，可通过在代码中添加类似如下的代码，用来检测类型是否实现了接口：

```
var _ io.Writer = (*myWriter)(nil)
var _ io.Writer = myWriter{}
```

# 类型转换和断言的区别

我们知道，Go 语言中不允许隐式类型转换，也就是说 `=` 两边，不允许出现类型不相同的变量。

`类型转换` 、 `类型断言` 本质都是把一个类型转换成另外一个类型。不同之处在于，类型断言是对接口变量进行的操作。

## 类型转换

对于 `类型转换` 而言，转换前后的两个类型要相互兼容才行。类型转换的语法为：

> <结果类型> := <目标类型> ( <表达式> )

```
package main

import "fmt"

func main() {
    var i int = 9

    var f float64
    f = float64(i)
    fmt.Printf("%T, %v\n", f, f)

    f = 10.8
    a := int(f)
    fmt.Printf("%T, %v\n", a, a)

    // s := []int(i)
}
```

上面的代码里，我定义了一个 `int` 型和 `float64` 型的变量，尝试在它们之前相互转换，结果是成功的： `int` 型和 `float64` 是相互兼容的。

如果我把最后一行代码的注释去掉，编译器会报告类型不兼容的错误：

```
cannot convert i (type int) to type []int
```

## 断言

前面说过，因为空接口 `interface{}` 没有定义任何函数，因此 Go 中所有类型都实现了空接口。当一个函数的形参是 `interface{}` ，那么在函数中，需要对形参进行断言，从而得到它的真实类型。

断言的语法为：

> <目标类型的值>，<布尔参数> := <表达式>.( 目标类型 ) // 安全类型断言
> <目标类型的值> := <表达式>.( 目标类型 )　　//非安全类型断言

类型转换和类型断言有些相似，不同之处，在于类型断言是对接口进行的操作。

还是来看一个简短的例子：

```
package main

import "fmt"

type Student struct {
    Name string
    Age int
}

func main() {
    var i interface{} = new(Student)
    s := i.(Student)

    fmt.Println(s)
}
```

运行一下：

```
panic: interface conversion: interface {} is *main.Student, not main.Student
```

直接 `panic` 了，这是因为 `i` 是 `*Student` 类型，并非 `Student` 类型，断言失败。这里直接发生了 `panic`，线上代码可能并不适合这样做，可以采用“安全断言”的语法：

```
func main() {
    var i interface{} = new(Student)
    s, ok := i.(Student)
    if ok {
        fmt.Println(s)
    }
}
```

这样，即使断言失败也不会 `panic`。

断言其实还有另一种形式，就是用在利用 `switch` 语句判断接口的类型。每一个 `case` 会被顺序地考虑。当命中一个 `case` 时，就会执行 `case` 中的语句，因此 `case` 语句的顺序是很重要的，因为很有可能会有多个 `case` 匹配的情况。

代码示例如下：

```
func main() {
    //var i interface{} = new(Student)
    //var i interface{} = (*Student)(nil)
    var i interface{}

    fmt.Printf("%p %v\n", &i, i)

    judge(i)
}

func judge(v interface{}) {
    fmt.Printf("%p %v\n", &v, v)

    switch v := v.(type) {
    case nil:
        fmt.Printf("%p %v\n", &v, v)
        fmt.Printf("nil type[%T] %v\n", v, v)
```

```
    case Student:
        fmt.Printf("%p %v\n", &v, v)
        fmt.Printf("Student type[%T] %v\n", v, v)

    case *Student:
        fmt.Printf("%p %v\n", &v, v)
        fmt.Printf("*Student type[%T] %v\n", v, v)

    default:
        fmt.Printf("%p %v\n", &v, v)
        fmt.Printf("unknow\n")
    }
}

type Student struct {
    Name string
    Age int
}
```

`main` 函数里有三行不同的声明，每次运行一行，注释另外两行，得到三组运行结果：

```
// --- var i interface{} = new(Student)
0xc4200701b0 [Name: ], [Age: 0]
0xc4200701d0 [Name: ], [Age: 0]
0xc420080020 [Name: ], [Age: 0]
*Student type[*main.Student] [Name: ], [Age: 0]

// --- var i interface{} = (*Student)(nil)
0xc42000e1d0 <nil>
0xc42000e1f0 <nil>
0xc42000c030 <nil>
*Student type[*main.Student] <nil>

// --- var i interface{}
0xc42000e1d0 <nil>
0xc42000e1e0 <nil>
0xc42000e1f0 <nil>
nil type[<nil>] <nil>
```

对于第一行语句：

```
var i interface{} = new(Student)
```

`i` 是一个 `*Student` 类型，匹配上第三个 case，从打印的三个地址来看，这三处的变量实际上都是不一样的。在 `main` 函数里有一个局部变量 `i` ；调用函数时，实际上是复制了一份参数，因此函数里又有一个变量 `v` ，它是 `i` 的拷贝；断言之后，又生成了一份新的拷贝。所以最终打印的三个变量的地址都不一样。

对于第二行语句：

```
var i interface{} = (*Student)(nil)
```

这里想说明的其实是 `i` 在这里动态类型是 `(*Student)` ，数据为 `nil` ，它的类型并不是 `nil` ，它与 `nil` 作比较的时候，得到的结果也是 `false` 。

最后一行语句：

```
var i interface{}
```

这回 `i` 才是 `nil` 类型。

【引申1】
`fmt.Println` 函数的参数是 `interface` 。对于内置类型，函数内部会用穷举法，得出它的真实类型，然后转换为字符串打印。而对于自定义类型，首先确定该类型是否实现了 `String()` 方法，如果实现了，则直接打印输出 `String()` 方法的结果；否则，会通过反射来遍历对象的成员进行打印。

再来看一个简短的例子，比较简单，不要紧张：

```
package main

import "fmt"

type Student struct {
    Name string
    Age int
}

func main() {
    var s = Student{
        Name: "qcrao",
        Age: 18,
    }

    fmt.Println(s)
}
```

因为 `Student` 结构体没有实现 `String()` 方法，所以 `fmt.Println` 会利用反射挨个打印成员变量：

```
{qcrao 18}
```

增加一个 `String()` 方法的实现：

```
func (s Student) String() string {
    return fmt.Sprintf("[Name: %s], [Age: %d]", s.Name, s.Age)
}
```

打印结果：

```
[Name: qcrao], [Age: 18]
```

按照我们自定义的方法来打印了。

【引申2】
针对上面的例子，如果改一下：

```
func (s *Student) String() string {
    return fmt.Sprintf("[Name: %s], [Age: %d]", s.Name, s.Age)
}
```

注意看两个函数的接受者类型不同，现在 `Student` 结构体只有一个接受者类型为 `指针类型` 的 `String()` 函数，打印结果：

```
{qcrao 18}
```

为什么？

> 类型 `T` 只有接受者是 `T` 的方法；而类型 `*T` 拥有接受者是 `T` 和 `*T` 的方法。语法上 `T` 能直接调 `*T` 的方法仅仅是 `Go` 的语法糖。

所以， `Student` 结构体定义了接受者类型是值类型的 `String()` 方法时，通过

```
fmt.Println(s)
fmt.Println(&s)
```

均可以按照自定义的格式来打印。

如果 `Student` 结构体定义了接受者类型是指针类型的 `String()` 方法时，只有通过

```
fmt.Println(&s)
```

才能按照自定义的格式打印。

# 参考资料

【类型转换和断言】https://www.cnblogs.com/zrtqsk/p/4157350.html

【断言】https://studygolang.com/articles/11419

# 接口的动态类型和动态值

从源码里可以看到： `iface` 包含两个字段： `tab` 是接口表指针，指向类型信息； `data` 是数据指针，则指向具体的数据。它们分别被称为 `动态类型` 和 `动态值` 。而接口值包括 `动态类型` 和 `动态值` 。

【引申1】接口类型和 `nil` 作比较

接口值的零值是指 `动态类型` 和 `动态值` 都为 `nil` 。当仅且当这两部分的值都为 `nil` 的情况下，这个接口值就才会被认为 `接口值 == nil` 。

来看个例子：

```
package main

import "fmt"

type Coder interface {
    code()
}

type Gopher struct {
    name string
}

func (g Gopher) code() {
    fmt.Printf("%s is coding\n", g.name)
}

func main() {
    var c Coder
    fmt.Println(c == nil)
    fmt.Printf("c: %T, %v\n", c, c)

    var g *Gopher
    fmt.Println(g == nil)

    c = g
    fmt.Println(c == nil)
    fmt.Printf("c: %T, %v\n", c, c)
}
```

输出：

```
true
c: <nil>, <nil>
true
false
c: *main.Gopher, <nil>
```

一开始， `c` 的 动态类型和动态值都为 `nil` ， `g` 也为 `nil` ，当把 `g` 赋值给 `c` 后， `c` 的动态类型变成了 `*main.Gopher` ，仅管 `c` 的动态值仍为 `nil` ，但是当 `c` 和 `nil` 作比较的时候，结果就是 `false` 了。

【引申2】
来看一个例子，看一下它的输出：

```
package main

import "fmt"

type MyError struct {}

func (i MyError) Error() string {
    return "MyError"
}

func main() {
    err := Process()
    fmt.Println(err)

    fmt.Println(err == nil)
}

func Process() error {
    var err *MyError = nil
    return err
}
```

函数运行结果：

```
<nil>
false
```

这里先定义了一个 `MyError` 结构体，实现了 `Error` 函数，也就实现了 `error` 接口。 `Process` 函数返回了一个 `error` 接口，这块隐含了类型转换。所以，虽然它的值是 `nil` ，其实它的类型是 `*MyError` ，最后和 `nil` 比较的时候，结果为 `false` 。

【引申3】如何打印出接口的动态类型和值？

直接看代码：

```
package main

import (
    "unsafe"
    "fmt"
)

type iface struct {
    itab, data uintptr
}

func main() {
    var a interface{} = nil

    var b interface{} = (*int)(nil)

    x := 5
    var c interface{} = (*int)(&x)

    ia := *(*iface)(unsafe.Pointer(&a))
    ib := *(*iface)(unsafe.Pointer(&b))
    ic := *(*iface)(unsafe.Pointer(&c))
```

```
    fmt.Println(ia, ib, ic)

    fmt.Println(*(*int)(unsafe.Pointer(ic.data)))
}
```

代码里直接定义了一个 `iface` 结构体，用两个指针来描述 `itab` 和 `data` ，之后将 a, b, c 在内存中的内容强制解释成我们自定义的 `iface` 。最后就可以打印出动态类型和动态值的地址。

运行结果如下：

```
{0 0} {17426912 0} {17426912 842350714568}
5
```

a 的动态类型和动态值的地址均为 0，也就是 nil；b 的动态类型和 c 的动态类型一致，都是 `*int` ；最后，c 的动态值为 5。

# 参考资料

【一个包含NIL指针的接口不是NIL接口】https://i6448038.github.io/2018/07/18/golang-mistakes/

# 标准库

**context**

**unsafe**

# context

# context 如何被取消

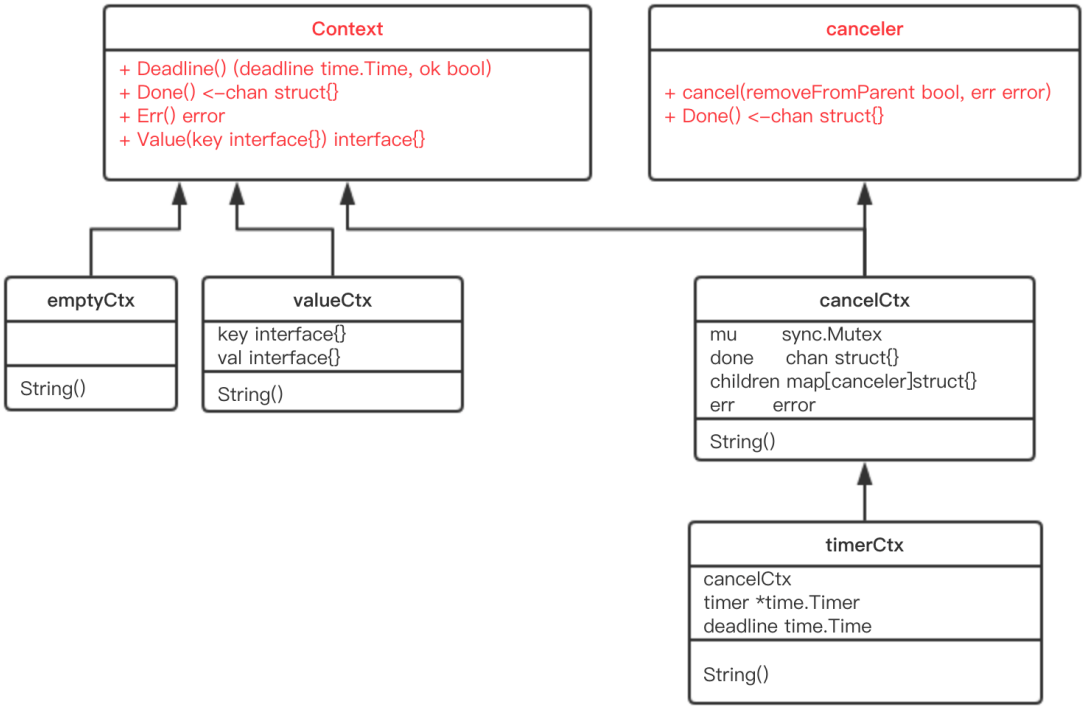
context 包的代码并不长， `context.go` 文件总共不到 500 行，其中还有很多大段的注释，代码可能也就 200 行左右的样子，是一个非常值得研究的代码库。

先看一张整体的图：

| 类型 | 名称 | 作用 |
| --- | --- | --- |
| Context | 接口 | 定义了 Context 接口的四个方法 |
| emptyCtx | 结构体 | 实现了 Context 接口，它其实是个空的 context |
| CancelFunc | 函数 | 取消函数 |
| canceler | 接口 | context 取消接口，定义了两个方法 |
| cancelCtx | 结构体 | 可以被取消 |
| timerCtx | 结构体 | 超时会被取消 |
| valueCtx | 结构体 | 可以存储 k-v 对 |
| Background | 函数 | 返回一个空的 context，常作为根 context |
| TODO | 函数 | 返回一个空的 context，常用于重构时期，没有合适的 context 可用 |
| WithCancel | 函数 | 基于父 context，生成一个可以取消的 context |
| newCancelCtx | 函数 | 创建一个可取消的 context |
| propagateCancel | 函数 | 向下传递 context 节点间的取消关系 |
| parentCancelCtx | 函数 | 找到第一个可取消的父节点 |
| removeChild | 函数 | 去掉父节点的孩子节点 |
| init | 函数 | 包初始化 |
| WithDeadline | 函数 | 创建一个有 deadline 的 context |
| WithTimeout | 函数 | 创建一个有 timeout 的 context |
| WithValue | 函数 | 创建一个存储 k-v 对的 context |

上面这张表展示了 context 的所有函数、接口、结构体，可以纵览全局，可以在读完文章后，再回头细看。

整体类图如下：




# 接口

## Context

现在可以直接看源码：

```
type Context interface {
    // 当 context 被取消或者到了 deadline，返回一个被关闭的 channel
    Done() <-chan struct{}

    // 在 channel Done 关闭后，返回 context 取消原因
    Err() error

    // 返回 context 是否会被取消以及自动取消时间（即 deadline）
    Deadline() (deadline time.Time, ok bool)

    // 获取 key 对应的 value
    Value(key interface{}) interface{}
}
```

`Context` 是一个接口，定义了 4 个方法，它们都是 `幂等` 的。也就是说连续多次调用同一个方法，得到的结果都是相同的。

`Done()` 返回一个 channel，可以表示 context 被取消的信号：当这个 channel 被关闭时，说明 context 被取消了。注意，这是一个只读的channel。 我们又知道，读一个关闭的 channel 会读出相应类型的零值。并且源码里没有地方会向这个 channel 里面塞入值。换句话说，这是一个 `receive-only` 的 channel。因此在子协程里读这个 channel，除非被关闭，否则读不出来任何东西。也正是利用了这一点，子协程从 channel 里读出了值（零值）后，就可以做一些收尾工作，尽快退出。

`Err()` 返回一个错误，表示 channel 被关闭的原因。例如是被取消，还是超时。

`Deadline()` 返回 context 的截止时间，通过此时间，函数就可以决定是否进行接下来的操作，如果时间太短，就可以不往下做了，否则浪费系统资源。当然，也可以用这个 deadline 来设置一个 I/O 操作的超时时间。

`Value()` 获取之前设置的 key 对应的 value。

## canceler

再来看另外一个接口：

```
type canceler interface {
    cancel(removeFromParent bool, err error)
    Done() <-chan struct{}
}
```

实现了上面定义的两个方法的 Context，就表明该 Context 是可取消的。源码中有两个类型实现了 canceler 接口：`*cancelCtx` 和 `*timerCtx`。注意是加了 `*` 号的，是这两个结构体的指针实现了 canceler 接口。

Context 接口设计成这个样子的原因：

- “取消”操作应该是建议性，而非强制性

caller 不应该去关心、干涉 callee 的情况，决定如何以及何时 return 是 callee 的责任。caller 只需发送“取消”信息，callee 根据收到的信息来做进一步的决策，因此接口并没有定义 cancel 方法。

- “取消”操作应该可传递

“取消”某个函数时，和它相关联的其他函数也应该“取消”。因此，`Done()` 方法返回一个只读的 channel，所有相关函数监听此 channel。一旦 channel 关闭，通过 channel 的“广播机制”，所有监听者都能收到。

# 结构体

## emptyCtx

源码中定义了 `Context` 接口后，并且给出了一个实现：

```
type emptyCtx int

func (*emptyCtx) Deadline() (deadline time.Time, ok bool) {
    return
}

func (*emptyCtx) Done() <-chan struct{} {
    return nil
}

func (*emptyCtx) Err() error {
    return nil
}

func (*emptyCtx) Value(key interface{}) interface{} {
    return nil
}
```

看这段源码，非常 happy。因为每个函数都实现的异常简单，要么是直接返回，要么是返回 nil。

所以，这实际上是一个空的 context，永远不会被 cancel，没有存储值，也没有 deadline。

它被包装成：

```
var (
    background = new(emptyCtx)
    todo       = new(emptyCtx)
)
```

通过下面两个导出的函数（首字母大写）对外公开：

```
func Background() Context {
    return background
}

func TODO() Context {
    return todo
}
```

background 通常用在 main 函数中，作为所有 context 的根节点。

todo 通常用在并不知道传递什么 context的情形。例如，调用一个需要传递 context 参数的函数，你手头并没有其他 context 可以传递，这时就可以传递 todo。这常常发生在重构进行中，给一些函数添加了一个 Context 参数，但不知道要传什么，就用 todo “占个位子”，最终要换成其他 context。

## cancelCtx

再来看一个重要的 context：

```
type cancelCtx struct {
    Context

    // 保护之后的字段
    mu       sync.Mutex
    done     chan struct{}
    children map[canceler]struct{}
    err      error
}
```

这是一个可以取消的 Context，实现了 canceler 接口。它直接将接口 Context 作为它的一个匿名字段，这样，它就可以被看成一个 Context。

先来看 `Done()` 方法的实现：

```
func (c *cancelCtx) Done() <-chan struct{} {
    c.mu.Lock()
    if c.done == nil {
        c.done = make(chan struct{})
    }
    d := c.done
    c.mu.Unlock()
    return d
}
```

c.done 是“懒汉式”创建，只有调用了 Done() 方法的时候才会被创建。再次说明，函数返回的是一个只读的 channel，而且没有地方向这个 channel 里面写数据。所以，直接调用读这个 channel，协程会被 block 住。一般通过搭配 select 来使用。

一旦关闭，就会立即读出零值。

`Err()` 和 `String()` 方法比较简单，不多说。推荐看源码，非常简单。

接下来，我们重点关注 `cancel()` 方法的实现：

```
func (c *cancelCtx) cancel(removeFromParent bool, err error) {
    // 必须要传 err
    if err == nil {
        panic("context: internal error: missing cancel error")
    }
    c.mu.Lock()
    if c.err != nil {
        c.mu.Unlock()
        return // 已经被其他协程取消
    }
    // 给 err 字段赋值
    c.err = err
    // 关闭 channel，通知其他协程
    if c.done == nil {
        c.done = closedchan
    } else {
        close(c.done)
    }

    // 遍历它的所有子节点
    for child := range c.children {
        // 递归地取消所有子节点
        child.cancel(false, err)
    }
    // 将子节点置空
    c.children = nil
    c.mu.Unlock()

    if removeFromParent {
        // 从父节点中移除自己
        removeChild(c.Context, c)
    }
}
```

总体来看，`cancel()` 方法的功能就是关闭 channel：c.done；递归地取消它的所有子节点；从父节点从删除自己。达到的效果是通过关闭 channel，将取消信号传递给了它的所有子节点。goroutine 接收到取消信号的方式就是 select 语句中的 `读 c.done` 被选中。

我们再来看创建一个可取消的 Context 的方法：

```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc) {
    c := newCancelCtx(parent)
    propagateCancel(parent, &c)
    return &c, func() { c.cancel(true, Canceled) }
}

func newCancelCtx(parent Context) cancelCtx {
    return cancelCtx{Context: parent}
}
```

这是一个暴露给用户的方法，传入一个父 Context（这通常是一个 `background`，作为根节点），返回新建的 context，新 context 的 done channel 是新建的（前文讲过）。

当 WithCancel 函数返回的 CancelFunc 被调用或者是父节点的 done channel 被关闭（父节点的 CancelFunc 被调用），此 context（子节点） 的 done channel 也会被关闭。

注意传给 WithCancel 方法的参数，前者是 true，也就是说取消的时候，需要将自己从父节点里删除。第二个参数则是一个固定的取消错误类型：

```
var Canceled = errors.New("context canceled")
```

还注意到一点，调用子节点 cancel 方法的时候，传入的第一个参数 `removeFromParent` 是 false。

两个问题需要回答：1. 什么时候会传 true？2. 为什么有时传 true，有时传 false？

当 `removeFromParent` 为 true 时，会将当前节点的 context 从父节点 context 中删除：

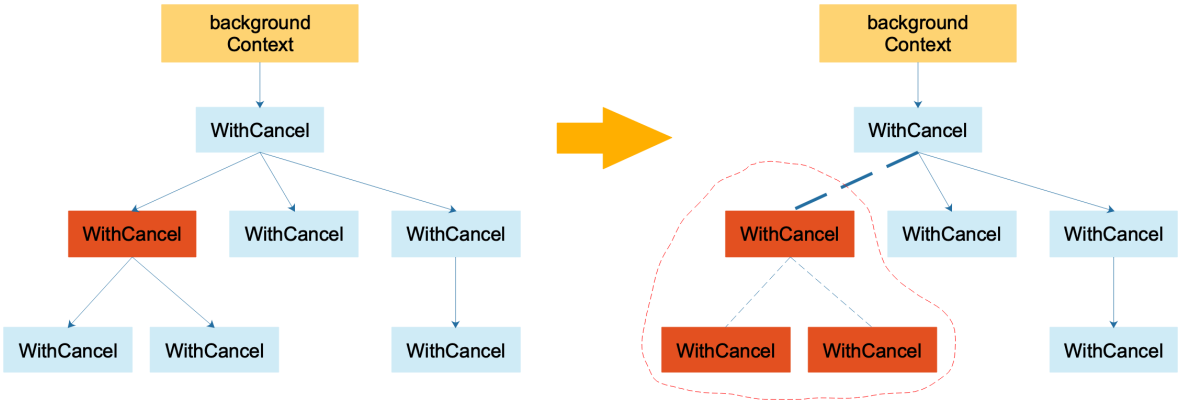
```
func removeChild(parent Context, child canceler) {
    p, ok := parentCancelCtx(parent)
    if !ok {
        return
    }
    p.mu.Lock()
    if p.children != nil {
        delete(p.children, child)
    }
    p.mu.Unlock()
}
```

最关键的一行：

```
delete(p.children, child)
```

什么时候会传 true 呢？答案是调用 `WithCancel()` 方法的时候，也就是新创建一个可取消的 context 节点时，返回的 cancelFunc 函数会传入 true。这样做的结果是：当调用返回的 cancelFunc 时，会将这个 context 从它的父节点里“除名”，因为父节点可能有很多子节点，你自己取消了，所以我要和你断绝关系，对其他人没影响。

在取消函数内部，我知道，我所有的子节点都会因为我的一：`c.children = nil` 而化为灰烬。我自然就没有必要再多做这一步，最后我所有的子节点都会和我断绝关系，没必要一个个做。另外，如果遍历子节点的时候，调用 child.cancel 函数传了 true，还会造成同时遍历和删除一个 map 的境地，会有问题的。




如上左图，代表一棵 context 树。当调用左图中标红 context 的 cancel 方法后，该 context 从它的父 context 中去除掉了：实线箭头变成了虚线。且虚线圈框出来的 context 都被取消了，圈内的 context 间的父子关系都荡然无存了。

重点看 `propagateCancel()`：

```
func propagateCancel(parent Context, child canceler) {
    // 父节点是个空节点
    if parent.Done() == nil {
        return // parent is never canceled
    }
    // 找到可以取消的父 context
    if p, ok := parentCancelCtx(parent); ok {
        p.mu.Lock()
        if p.err != nil {
            // 父节点已经被取消了，本节点（子节点）也要取消
            child.cancel(false, p.err)
        } else {
            // 父节点未取消
            if p.children == nil {
                p.children = make(map[canceler]struct{})
            }
            // "挂到"父节点上
            p.children[child] = struct{}{}
        }
        p.mu.Unlock()
    } else {
        // 如果没有找到可取消的父 context。新启动一个协程监控父节点或子节点取消信号
        go func() {
            select {
            case <-parent.Done():
                child.cancel(false, parent.Err())
            case <-child.Done():
            }
        }()
    }
}
```

这个方法的作用就是向上寻找可以“挂靠”的“可取消”的 context，并且“挂靠”上去。这样，调用上层 cancel 方法的时候，就可以层层传递，将那些挂靠的子 context 同时“取消”。

这里着重解释下为什么会有 else 描述的情况发生。 `else` 是指当前节点 context 没有向上找到可以取消的父节点，那么就要再启动一个协程监控父节点或者子节点的取消动作。

这里就有疑问了，既然没找到可以取消的父节点，那 `case <-parent.Done()` 这个 case 就永远不会发生，所以可以忽略这个 case；而 `case <-child.Done()` 这个 case 又啥事不干。那这个 `else` 不就多余了吗？

其实不然。我们来看 `parentCancelCtx` 的代码：

```
func parentCancelCtx(parent Context) (*cancelCtx, bool) {
    for {
        switch c := parent.(type) {
        case *cancelCtx:
            return c, true
        case *timerCtx:
            return &c.cancelCtx, true
        case *valueCtx:
            parent = c.Context
        default:
            return nil, false
        }
    }
}
```

这里只会识别三种 Context 类型：*cancelCtx*，timerCtx，*valueCtx。若是把 Context 内嵌到一个类型里，就识别不出来了。

由于 context 包的代码并不多，所以我直接把它 copy 出来了，然后在 else 语句里加上了几条打印语句，来验证上面的说法：

```
type MyContext struct {
    // 这里的 Context 是我 copy 出来的，所以前面不用加 context.
    Context
}

func main() {
    childCancel := true

    parentCtx, parentFunc := WithCancel(Background())
    mctx := MyContext{parentCtx}

    childCtx, childFun := WithCancel(mctx)

    if childCancel {
        childFun()
    } else {
        parentFunc()
    }

    fmt.Println(parentCtx)
    fmt.Println(mctx)
    fmt.Println(childCtx)

    // 防止主协程退出太快，子协程来不及打印
    time.Sleep(10 * time.Second)
}
```

我自己在 else 里添加的打印语句我就不贴出来了，感兴趣的可以自己动手实验下。我们看下三个 context 的打印结果：

```
context.Background.WithCancel
{context.Background.WithCancel}
{context.Background.WithCancel}.WithCancel
```

果然，mctx，childCtx 和正常的 parentCtx 不一样，因为它是一个自定义的结构体类型。

`else` 这段代码说明，如果把 ctx 强行塞进一个结构体，并用它作为父节点，调用 WithCancel 函数构建子节点 context 的时候，Go 会新启动一个协程来监控取消信号，明显有点浪费嘛。

再来说一下，select 语句里的两个 case 其实都不能删。

```
select {
    case <-parent.Done():
        child.cancel(false, parent.Err())
    case <-child.Done():
}
```

第一个 case 说明当父节点取消，则取消子节点。如果去掉这个 case，那么父节点取消的信号就不能传递到子节点。

第二个 case 是说如果子节点自己取消了，那就退出这个 select，父节点的取消信号就不用管了。如果去掉这个 case，那么很可能父节点一直不取消，这个 goroutine 就泄漏了。当然，如果父节点取消了，就会重复让子节点取消，不过，这也没什么影响嘛。

## timerCtx

timerCtx 基于 cancelCtx，只是多了一个 time.Timer 和一个 deadline。Timer 会在 deadline 到来时，自动取消 context。

```
type timerCtx struct {
    cancelCtx
    timer *time.Timer // Under cancelCtx.mu.

    deadline time.Time
}
```

timerCtx 首先是一个 cancelCtx，所以它能取消。看下 cancel() 方法：

```
func (c *timerCtx) cancel(removeFromParent bool, err error) {
    // 直接调用 cancelCtx 的取消方法
    c.cancelCtx.cancel(false, err)
    if removeFromParent {
        // 从父节点中删除子节点
        removeChild(c.cancelCtx.Context, c)
    }
    c.mu.Lock()
    if c.timer != nil {
        // 关掉定时器，这样，在deadline 到来时，不会再次取消
        c.timer.Stop()
        c.timer = nil
    }
    c.mu.Unlock()
}
```

创建 timerCtx 的方法：

```
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc) {
    return WithDeadline(parent, time.Now().Add(timeout))
}
```

`WithTimeout` 函数直接调用了 `WithDeadline`，传入的 deadline 是当前时间加上 timeout 的时间，也就是从现在开始再经过 timeout 时间就算超时。也就是说，`WithDeadline` 需要用的是绝对时间。重点来看它：

```
func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc) {
    if cur, ok := parent.Deadline(); ok && cur.Before(deadline) {
        // 如果父节点 context 的 deadline 早于指定时间。直接构建一个可取消的 context。
        // 原因是一旦父节点超时，自动调用 cancel 函数，子节点也会随之取消。
        // 所以不用单独处理子节点的计时器时间到了之后，自动调用 cancel 函数
        return WithCancel(parent)
    }

    // 构建 timerCtx
    c := &timerCtx{
        cancelCtx: newCancelCtx(parent),
        deadline:  deadline,
    }
    // 挂靠到父节点上
    propagateCancel(parent, c)

    // 计算当前距离 deadline 的时间
```

```
    d := time.Until(deadline)
    if d <= 0 {
        // 直接取消
        c.cancel(true, DeadlineExceeded) // deadline has already passed
        return c, func() { c.cancel(true, Canceled) }
    }
    c.mu.Lock()
    defer c.mu.Unlock()
    if c.err == nil {
        // d 时间后，timer 会自动调用 cancel 函数。自动取消
        c.timer = time.AfterFunc(d, func() {
            c.cancel(true, DeadlineExceeded)
        })
    }
    return c, func() { c.cancel(true, Canceled) }
}
```

也就是说仍然要把子节点挂靠到父节点，一旦父节点取消了，会把取消信号向下传递到子节点，子节点随之取消。

有一个特殊情况是，如果要创建的这个子节点的 deadline 比父节点要晚，也就是说如果父节点是时间到自动取消，那么一定会取消这个子节点，导致子节点的 deadline 根本不起作用，因为子节点在 deadline 到来之前就已经被父节点取消了。

这个函数的最核心的一句是：

```
c.timer = time.AfterFunc(d, func() {
    c.cancel(true, DeadlineExceeded)
})
```

c.timer 会在 d 时间间隔后，自动调用 cancel 函数，并且传入的错误就是 `DeadlineExceeded` ：

```
var DeadlineExceeded error = deadlineExceededError{}

type deadlineExceededError struct{}

func (deadlineExceededError) Error() string   { return "context deadline exceeded" }
```

也就是超时错误。

# context 是什么

Go 1.7 标准库引入 context，中文译作“上下文”，准确说它是 goroutine 的上下文，包含 goroutine 的运行状态、环境、现场等信息。

context 主要用来在 goroutine 之间传递上下文信息，包括：取消信号、超时时间、截止时间、k-v 等。

随着 context 包的引入，标准库中很多接口因此加上了 context 参数，例如 database/sql 包。context 几乎成为了并发控制和超时控制的标准做法。

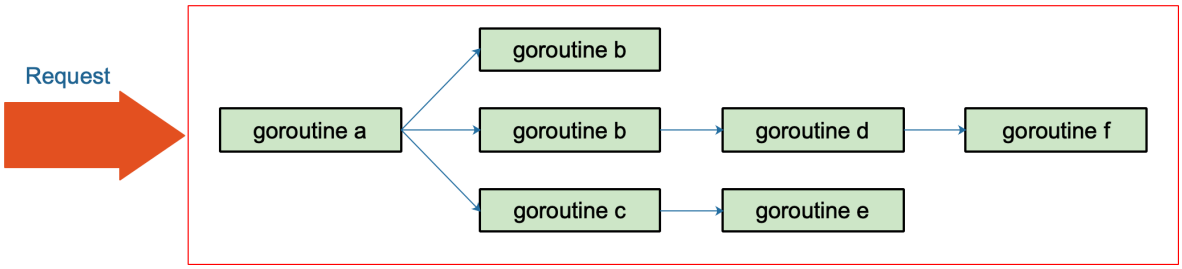
> context.Context 类型的值可以协调多个 groutine 中的代码执行“取消”操作，并且可以存储键值对。最重要的是它是并发安全的。

> 与它协作的 API 都可以由外部控制执行“取消”操作，例如：取消一个 HTTP 请求的执行。

# context 有什么作用

Go 常用来写后台服务，通常只需要几行代码，就可以搭建一个 http server。

在 Go 的 server 里，通常每来一个请求都会启动若干个 goroutine 同时工作：有些去数据库拿数据，有些调用下游接口获取相关数据.....

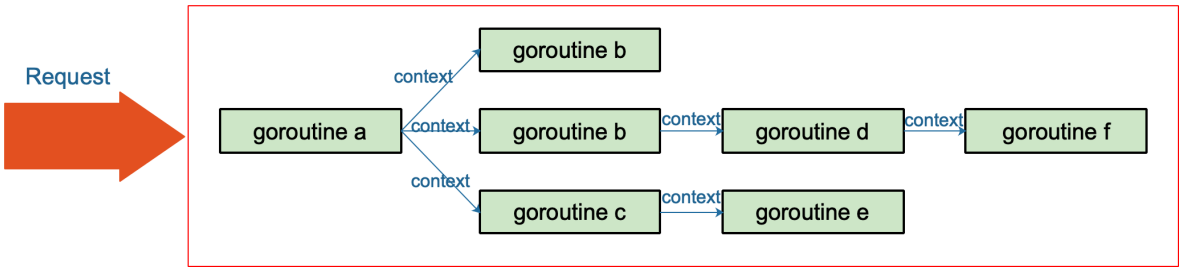



这些 goroutine 需要共享这个请求的基本数据，例如登陆的 token，处理请求的最大超时时间（如果超过此值再返回数据，请求方因为超时接收不到）等等。当请求被取消或是处理时间太长，这有可能是使用者关闭了浏览器或是已经超过了请求方规定的超时时间，请求方直接放弃了这次请求结果。这时，所有正在为这个请求工作的 goroutine 需要快速退出，因为它们的“工作成果”不再被需要了。在相关联的 goroutine 都退出后，系统就可以回收相关的资源。

再多说一点，Go 语言中的 server 实际上是一个“协程模型”，也就是说一个协程处理一个请求。例如在业务的高峰期，某个下游服务的响应变慢，而当前系统的请求又没有超时控制，或者超时时间设置地过大，那么等待下游服务返回数据的协程就会越来越多。而我们知道，协程是要消耗系统资源的，后果就是协程数激增，内存占用飙涨，甚至导致服务不可用。更严重的会导致雪崩效应，整个服务对外表现为不可用，这肯定是 P0 级别的事故。这时，肯定有人要背锅了。

其实前面描述的 P0 级别事故，通过设置“允许下游最长处理时间”就可以避免。例如，给下游设置的 timeout 是 50 ms，如果超过这个值还没有接收到返回数据，就直接向客户端返回一个默认值或者错误。例如，返回商品的一个默认库存数量。注意，这里设置的超时时间和创建一个 http client 设置的读写超时时间不一样，这里不详细展开。可以去看看参考资料 `【Go 在今日头条的实践】` 一文，有很精彩的论述。

context 包就是为了解决上面所说的这些问题而开发的：在 一组 goroutine 之间传递共享的值、取消信号、deadline.....




用简练一些的话来说，在Go 里，我们不能直接杀死协程，协程的关闭一般会用 `channel+select` 方式来控制。但是在某些场景下，例如处理一个请求衍生了很多协程，这些协程之间是相互关联的：需要共享一些全局变量、有共同的 deadline 等，而且可以同时被关闭。再用 `channel+select` 就会比较麻烦，这时就可以通过 context 来实现。

一句话：context 用来解决 goroutine 之间 `退出通知` 、 `元数据传递` 的功能。

【引申1】举例说明 context 在实际项目中如何使用。

context 使用起来非常方便。源码里对外提供了一个创建根节点 context 的函数：

```
func Background() Context
```

background 是一个空的 context， 它不能被取消，没有值，也没有超时时间。

有了根节点 context，又提供了四个函数创建子节点 context：

```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
func WithValue(parent Context, key, val interface{}) Context
```

context 会在函数传递间传递。只需要在适当的时间调用 cancel 函数向 goroutines 发出取消信号或者调用 Value 函数取出 context 中的值。

在官方博客里，对于使用 context 提出了几点建议：

1. Do not store Contexts inside a struct type; instead, pass a Context explicitly to each function that needs it. The Context should be the first parameter, typically named ctx.
2. Do not pass a nil Context, even if a function permits it. Pass context.TODO if you are unsure about which Context to use.
3. Use context Values only for request-scoped data that transits processes and APIs, not for passing optional parameters to functions.
4. The same Context may be passed to functions running in different goroutines; Contexts are safe for simultaneous use by multiple goroutines.

我翻译一下：

1. 不要将 Context 塞到结构体里。直接将 Context 类型作为函数的第一参数，而且一般都命名为 ctx。
2. 不要向函数传入一个 nil 的 context，如果你实在不知道传什么，标准库给你准备好了一个 context：todo。
3. 不要把本应该作为函数参数的类型塞到 context 中，context 存储的应该是一些共同的数据。例如：登陆的 session、cookie 等。
4. 同一个 context 可能会被传递到多个 goroutine，别担心，context 是并发安全的。

# 传递共享的数据

对于 Web 服务端开发，往往希望将一个请求处理的整个过程串起来，这就非常依赖于 Thread Local（对于 Go 可理解为单个协程所独有） 的变量，而在 Go 语言中并没有这个概念，因此需要在函数调用的时候传递 context。

```
package main

import (
    "context"
    "fmt"
)

func main() {
    ctx := context.Background()
    process(ctx)

    ctx = context.WithValue(ctx, "traceId", "qcrao-2019")
    process(ctx)
}

func process(ctx context.Context) {
    traceId, ok := ctx.Value("traceId").(string)
    if ok {
        fmt.Printf("process over. trace_id=%s\n", traceId)
    } else {
        fmt.Printf("process over. no trace_id\n")
    }
}
```

运行结果：

```
process over. no trace_id
process over. trace_id=qcrao-2019
```

第一次调用 process 函数时，ctx 是一个空的 context，自然取不出来 traceId。第二次，通过 `WithValue` 函数创建了一个 context，并赋上了 `traceId` 这个 key，自然就能取出来传入的 value 值。

当然，现实场景中可能是从一个 HTTP 请求中获取到的 Request-ID。所以，下面这个样例可能更适合：

```
const requestIDKey int = 0

func WithRequestID(next http.Handler) http.Handler {
    return http.HandlerFunc(
        func(rw http.ResponseWriter, req *http.Request) {
            // 从 header 中提取 request-id
            reqID := req.Header.Get("X-Request-ID")
            // 创建 valueCtx。使用自定义的类型，不容易冲突
            ctx := context.WithValue(
                req.Context(), requestIDKey, reqID)

            // 创建新的请求
            req = req.WithContext(ctx)

            // 调用 HTTP 处理函数
            next.ServeHTTP(rw, req)
        }
    )
}

// 获取 request-id
func GetRequestID(ctx context.Context) string {
    ctx.Value(requestIDKey).(string)
}

func Handle(rw http.ResponseWriter, req *http.Request) {
    // 拿到 reqId，后面可以记录日志等等
    reqID := GetRequestID(req.Context())
    ...
}

func main() {
    handler := WithRequestID(http.HandlerFunc(Handle))
    http.ListenAndServe("/", handler)
}
```

# 取消 goroutine

我们先来设想一个场景：打开外卖的订单页，地图上显示外卖小哥的位置，而且是每秒更新 1 次。app 端向后台发起 websocket 连接（现实中可能是轮询）请求后，后台启动一个协程，每隔 1 秒计算 1 次小哥的位置，并发送给端。如果用户退出此页面，则后台需要“取消”此过程，退出 goroutine，系统回收资源。

后端可能的实现如下：

```
func Perform() {
    for {
        calculatePos()
```

```
        sendResult()
        time.Sleep(time.Second)
    }
}
```

如果需要实现“取消”功能，并且在不了解 context 功能的前提下，可能会这样做：给函数增加一个指针型的 bool 变量，在 for 语句的开始处判断 bool 变量是发由 true 变为 false，如果改变，则退出循环。

上面给出的简单做法，可以实现想要的效果，没有问题，但是并不优雅，并且一旦协程数量多了之后，并且各种嵌套，就会很麻烦。优雅的做法，自然就要用到 context。

```
func Perform(ctx context.Context) {
    for {
        calculatePos()
        sendResult()

        select {
        case <-ctx.Done():
            // 被取消，直接返回
            return
        case <-time.After(time.Second):
            // block 1 秒钟
        }
    }
}
```

主流程可能是这样的：

```
ctx, cancel := context.WithTimeout(context.Background(), time.Hour)
go Perform(ctx)

// ......
// app 端返回页面，调用cancel 函数
cancel()
```

注意一个细节，WithTimeOut 函数返回的 context 和 cancelFun 是分开的。context 本身并没有取消函数，这样做的原因是取消函数只能由外层函数调用，防止子节点 context 调用取消函数，从而严格控制信息的流向：由父节点 context 流向子节点 context。

# 防止 goroutine 泄漏

前面那个例子里，goroutine 还是会自己执行完，最后返回，只不过会多浪费一些系统资源。这里改编一个“如果不用 context 取消，goroutine 就会泄漏的例子”，来自参考资料：【避免协程泄漏】。

```
func gen() <-chan int {
    ch := make(chan int)
    go func() {
        var n int
        for {
            ch <- n
            n++
            time.Sleep(time.Second)
        }
    }()
    return ch
}
```

这是一个可以生成无限整数的协程，但如果我只需要它产生的前 5 个数，那么就会发生 goroutine 泄漏：

```
func main() {
    for n := range gen() {
        fmt.Println(n)
        if n == 5 {
            break
        }
    }
    // ......
}
```

当 n == 5 的时候，直接 break 掉。那么 gen 函数的协程就会执行无限循环，永远不会停下来。发生了 goroutine 泄漏。

用 context 改进这个例子：

```
func gen(ctx context.Context) <-chan int {
    ch := make(chan int)
    go func() {
        var n int
        for {
            select {
            case <-ctx.Done():
                return
            case ch <- n:
                n++
                time.Sleep(time.Second)
            }
        }
    }()
    return ch
}

func main() {
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel() // 避免其他地方忘记 cancel，且重复调用不影响

    for n := range gen(ctx) {
        fmt.Println(n)
        if n == 5 {
            cancel()
            break
        }
    }
    // ......
}
```

增加一个 context，在 break 前调用 cancel 函数，取消 goroutine。gen 函数在接收到取消信号后，直接退出，系统回收资源。

# context.Value 的查找过程是怎样的

```
type valueCtx struct {
    Context
    key, val interface{}
}
```

它实现了两个方法：

```
func (c *valueCtx) String() string {
    return fmt.Sprintf("%v.WithValue(%#v, %#v)", c.Context, c.key, c.val)
}

func (c *valueCtx) Value(key interface{}) interface{} {
    if c.key == key {
        return c.val
    }
    return c.Context.Value(key)
}
```

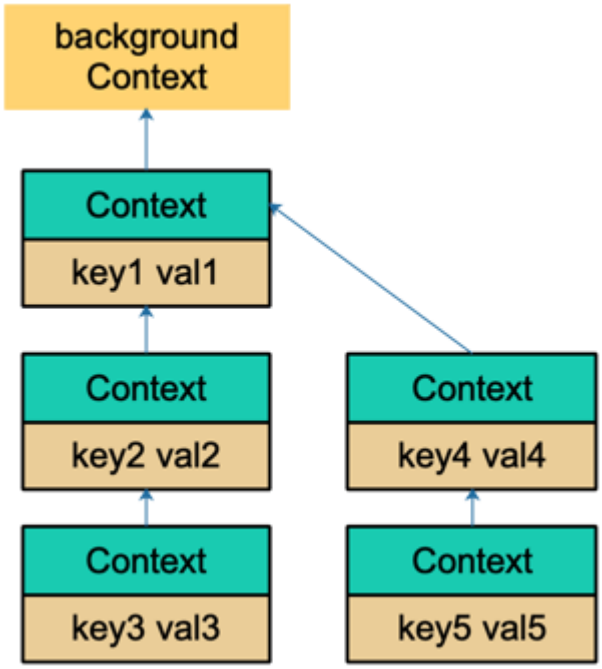
由于它直接将 Context 作为匿名字段，因此仅管它只实现了 2 个方法，其他方法继承自父 context。但它仍然是一个 Context，这是 Go 语言的一个特点。

创建 valueCtx 的函数：

```
func WithValue(parent Context, key, val interface{}) Context {
    if key == nil {
        panic("nil key")
    }
    if !reflect.TypeOf(key).Comparable() {
        panic("key is not comparable")
    }
    return &valueCtx{parent, key, val}
}
```

对 key 的要求是可比较，因为之后需要通过 key 取出 context 中的值，可比较是必须的。

通过层层传递 context，最终形成这样一棵树：




和链表有点像，只是它的方向相反：Context 指向它的父节点，链表则指向下一个节点。通过 WithValue 函数，可以创建层层的 valueCtx，存储 goroutine 间可以共享的变量。

取值的过程，实际上是一个递归查找的过程：

```
func (c *valueCtx) Value(key interface{}) interface{} {
    if c.key == key {
        return c.val
    }
    return c.Context.Value(key)
}
```

它会顺着链路一直往上找，比较当前节点的 key
是否是要找的 key，如果是，则直接返回 value。否则，一直顺着 context 往前，最终找到根节点（一般是 emptyCtx），直接返回一个 nil。所以用 Value 方法的时候要判断结果是否为 nil。

因为查找方向是往上走的，所以，父节点没法获取子节点存储的值，子节点却可以获取父节点的值。

`WithValue` 创建 context 节点的过程实际上就是创建链表节点的过程。两个节点的 key 值是可以相等的，但它们是两个不同的 context 节点。查找的时候，会向上查找到最后一个挂载的 context 节点，也就是离得比较近的一个父节点 context。所以，整体上而言，用 `WithValue` 构造的其实是一个低效率的链表。

如果你接手过项目，肯定经历过这样的窘境：在一个处理过程中，有若干子函数、子协程。各种不同的地方会向 context 里塞入各种不同的 k-v 对，最后在某个地方使用。

你根本就不知道什么时候什么地方传了什么值？这些值会不会被“覆盖”（底层是两个不同的 context 节点，查找的时候，只会返回一个结果）？你肯定会崩溃的。

而这也是 `context.Value` 最受争议的地方。很多人建议尽量不要通过 context 传值。

# unsafe

# Go指针和unsafe.Pointer有什么区别

Go 语言的作者之一 Ken Thompson 也是 C 语言的作者。所以，Go 可以看作 C 系语言，它的很多特性都和 C 类似，指针就是其中之一。

然而，Go 语言的指针相比 C 的指针有很多限制。这当然是为了安全考虑，要知道像 Java/Python 这些现代语言，生怕程序员出错，哪有什么指针（这里指的是显式的指针）？更别说像 C/C++ 还需要程序员自己清理“垃圾”。所以对于 Go 来说，有指针已经很不错了，仅管它有很多限制。

相比于 C 语言中指针的灵活，Go 的指针多了一些限制。但这也算是 Go 的成功之处：既可以享受指针带来的便利，又避免了指针的危险性。

限制一：`Go 的指针不能进行数学运算`。

来看一个简单的例子：

```
a := 5
p := &a

p++
p = &a + 3
```

上面的代码将不能通过编译，会报编译错误：`invalid operation`，也就是说不能对指针做数学运算。

限制二：`不同类型的指针不能相互转换`。

例如下面这个简短的例子：

```
func main() {
    a := int(100)
    var f *float64

    f = &a
}
```

也会报编译错误：

```
cannot use &a (type *int) as type *float64 in assignment
```

限制三：`不同类型的指针不能使用 == 或 != 比较`。

只有在两个指针类型相同或者可以相互转换的情况下，才可以对两者进行比较。另外，指针可以通过 `==` 和 `!=` 直接和 `nil` 作比较。

限制四：`不同类型的指针变量不能相互赋值`。

这一点同限制三。

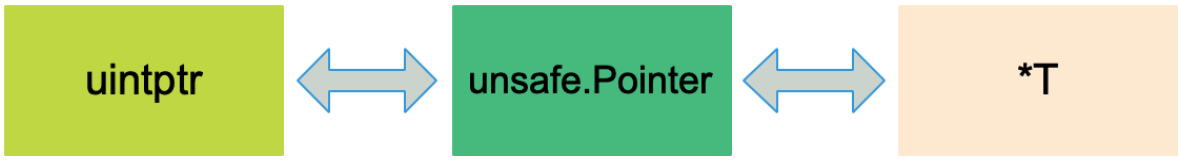
unsafe.Pointer 在 unsafe 包：

```
type ArbitraryType int

type Pointer *ArbitraryType
```

从命名来看，`Arbitrary`是任意的意思，也就是说 Pointer 可以指向任意类型，实际上它类似于 C 语言里的`void*`。

unsafe 包提供了 2 点重要的能力：

> 1. 任何类型的指针和 unsafe.Pointer 可以相互转换。
> 2. uintptr 类型和 unsafe.Pointer 可以相互转换。




pointer 不能直接进行数学运算，但可以把它转换成 uintptr，对 uintptr 类型进行数学运算，再转换成 pointer 类型。

```
// uintptr 是一个整数类型，它足够大，可以存储
type uintptr uintptr
```

还有一点要注意的是，uintptr 并没有指针的语义，意思就是 uintptr 所指向的对象会被 gc 无情地回收。而 unsafe.Pointer 有指针语义，可以保护它所指向的对象在“有用”的时候不会被垃圾回收。

unsafe 包中的几个函数都是在编译期间执行完毕，毕竟，编译器对内存分配这些操作“了然于胸”。在`/usr/local/go/src/cmd/compile/internal/gc/unsafe.go`路径下，可以看到编译期间 Go 对 unsafe 包中函数的处理。

# 如何利用unsafe包修改私有成员

对于一个结构体，通过 offset 函数可以获取结构体成员的偏移量，进而获取成员的地址，读写该地址的内存，就可以达到改变成员值的目的。

这里有一个内存分配相关的事实：结构体会被分配一块连续的内存，结构体的地址也代表了第一个成员的地址。

我们来看一个例子：

```
package main

import (
    "fmt"
    "unsafe"
)

type Programmer struct {
    name string
    language string
}

func main() {
    p := Programmer{"stefno", "go"}
    fmt.Println(p)

    name := (*string)(unsafe.Pointer(&p))
    *name = "qcrao"

    lang := (*string)(unsafe.Pointer(uintptr(unsafe.Pointer(&p)) + unsafe.Offsetof(p.language)))
    *lang = "Golang"

    fmt.Println(p)
}
```

运行代码，输出：

```
{stefno go}
{qcrao Golang}
```

name 是结构体的第一个成员，因此可以直接将 &p 解析成 *string。这一点，在前面获取 map 的 count 成员时，用的是同样的原理。

对于结构体的私有成员，现在有办法可以通过 unsafe.Pointer 改变它的值了。

我把 Programmer 结构体升级，多加一个字段：

```
type Programmer struct {
    name string
    age int
    language string
}
```

并且放在其他包，这样在 main 函数中，它的三个字段都是私有成员变量，不能直接修改。但我通过 unsafe.Sizeof() 函数可以获取成员大小，进而计算出成员的地址，直接修改内存。

```
func main() {
    p := Programmer{"stefno", 18, "go"}
    fmt.Println(p)

    lang := (*string)(unsafe.Pointer(uintptr(unsafe.Pointer(&p)) + unsafe.Sizeof(int(0)) + unsafe.Sizeof(string(""))))
    *lang = "Golang"

    fmt.Println(p)
}
```

输出：

```
{stefno 18 go}
{stefno 18 Golang}
```

# 如何利用unsafe获取slice&map的长度

## 获取 slice 长度

通过前面关于 slice 的文章，我们知道了 slice header 的结构体定义：

```
// runtime/slice.go
type slice struct {
    array unsafe.Pointer // 元素指针
    len   int // 长度
    cap   int // 容量
}
```

调用 make 函数新建一个 slice，底层调用的是 makeslice 函数，返回的是 slice 结构体：

```
func makeslice(et *_type, len, cap int) slice
```

因此我们可以通过 unsafe.Pointer 和 uintptr 进行转换，得到 slice 的字段值。

```
func main() {
    s := make([]int, 9, 20)
    var Len = *(*int)(unsafe.Pointer(uintptr(unsafe.Pointer(&s)) + uintptr(8)))
    fmt.Println(Len, len(s)) // 9 9

    var Cap = *(*int)(unsafe.Pointer(uintptr(unsafe.Pointer(&s)) + uintptr(16)))
    fmt.Println(Cap, cap(s)) // 20 20
}
```

Len，cap 的转换流程如下：

```
Len: &s => pointer => uintptr => pointer => *int => int
Cap: &s => pointer => uintptr => pointer => *int => int
```

## 获取 map 长度

再来看一下上篇文章我们讲到的 map：

```
type hmap struct {
    count     int
    flags     uint8
    B         uint8
    noverflow uint16
    hash0     uint32

    buckets    unsafe.Pointer
    oldbuckets unsafe.Pointer
    nevacuate  uintptr

    extra *mapextra
}
```

和 slice 不同的是，makemap 函数返回的是 hmap 的指针，注意是指针：

```
func makemap(t *maptype, hint int64, h *hmap, bucket unsafe.Pointer) *hmap
```

我们依然能通过 unsafe.Pointer 和 uintptr 进行转换，得到 hamp 字段的值，只不过，现在 count 变成二级指针了：

```
func main() {
    mp := make(map[string]int)
    mp["qcrao"] = 100
    mp["stefno"] = 18

    count := **(**int)(unsafe.Pointer(&mp))
    fmt.Println(count, len(mp)) // 2 2
}
```

count 的转换过程：

```
&mp => pointer => **int => int
```

# 如何实现字符串和byte切片的零拷贝转换

这是一个非常精典的例子。实现字符串和 bytes 切片之间的转换，要求是 `zero-copy` 。想一下，一般的做法，都需要遍历字符串或 bytes 切片，再挨个赋值。

完成这个任务，我们需要了解 slice 和 string 的底层数据结构：

```
type StringHeader struct {
    Data uintptr
    Len  int
}

type SliceHeader struct {
    Data uintptr
    Len  int
    Cap  int
}
```

上面是反射包下的结构体，路径：src/reflect/value.go。只需要共享底层 Data 和 Len 就可以实现 `zero-copy` 。

```
func string2bytes(s string) []byte {
    return *(*[]byte)(unsafe.Pointer(&s))
}
func bytes2string(b []byte) string{
    return *(*string)(unsafe.Pointer(&b))
}
```

原理上是利用指针的强转，代码比较简单，不作详细解释。

# goroutine 调度器

# g0 栈何用户栈如何切换

上一讲讲完了 main goroutine 的诞生，它不是第一个，算上 g0，它要算第二个了。不过，我们要考虑的就是这个 goroutine，它会真正执行用户代码。

`g0` 栈用于执行调度器的代码，执行完之后，要跳转到执行用户代码的地方，如何跳转？这中间涉及到栈和寄存器的切换。要知道，函数调用和返回主要靠的也是 CPU 寄存器的切换。`goroutine` 的切换和此类似。

继续看 `proc1` 函数的代码。中间有一段调整运行空间的代码，计算出的结果一般为 0，也就是一般不会调整 SP 的位置，忽略好了。

```
// 确定参数入栈位置
spArg := sp
```

参数的入参位置也是从 SP 处开始，通过：

```
// 将参数从执行 newproc 函数的栈拷贝到新 g 的栈
memmove(unsafe.Pointer(spArg), unsafe.Pointer(argp), uintptr(narg))
```

将 fn 的参数从 g0 栈上拷贝到 newg 的栈上，memmove 函数需要传入源地址、目的地址、参数大小。由于 main 函数在这里没有参数需要拷贝，因此这里相当于没做什么。

接着，初始化 newg 的各种字段，而且涉及到最重要的 pc，sp 等字段：

```
// 把 newg.sched 结构体成员的所有成员设置为 0
memclrNoHeapPointers(unsafe.Pointer(&newg.sched), unsafe.Sizeof(newg.sched))
// 设置 newg 的 sched 成员，调度器需要依靠这些字段才能把 goroutine 调度到 CPU 上运行
newg.sched.sp = sp
newg.stktopsp = sp
// newg.sched.pc 表示当 newg 被调度起来运行时从这个地址开始执行指令
newg.sched.pc = funcPC(goexit) + sys.PCQuantum // +PCQuantum so that previous instruction is in same function
newg.sched.g = guintptr(unsafe.Pointer(newg))
gostartcallfn(&newg.sched, fn)
newg.gopc = callerpc
// 设置 newg 的 startpc 为 fn.fn，该成员主要用于函数调用栈的 traceback 和栈收缩
// newg 真正从哪里开始执行并不依赖于这个成员，而是 sched.pc
newg.startpc = fn.fn
if _g_.m.curg != nil {
    newg.labels = _g_.m.curg.labels
}
```

首先，`memclrNoHeapPointers` 将 newg.sched 的内存全部清零。接着，设置 sched 的 sp 字段，当 goroutine 被调度到 m 上运行时，需要通过 sp 字段来指示栈顶的位置，这里设置的就是新栈的栈顶位置。

最关键的一行来了：

```
// newg.sched.pc 表示当 newg 被调度起来运行时从这个地址开始执行指令
newg.sched.pc = funcPC(goexit) + sys.PCQuantum // +PCQuantum so that previous instruction is in same function
```

设置 `pc` 字段为函数 `goexit` 的地址加 1，也说是 `goexit` 函数的第二条指令，`goexit` 函数是 `goroutine` 退出后的一些清理工作。有点奇怪，这是要干嘛？接着往后看。

```
newg.sched.g = guintptr(unsafe.Pointer(newg))
```

设置 `g` 字段为 newg 的地址。插一句，sched 是 g 结构体的一个字段，它本身也是一个结构体，保存调度信息。复习一下：

```
type gobuf struct {
    // 存储 rsp 寄存器的值
    sp   uintptr
    // 存储 rip 寄存器的值
    pc   uintptr
    // 指向 goroutine
    g    guintptr
    ctxt unsafe.Pointer // this has to be a pointer so that gc scans it
    // 保存系统调用的返回值
    ret  sys.Uintreg
    lr   uintptr
    bp   uintptr // for GOEXPERIMENT=framepointer
}
```

接下来的这个函数非常重要，可以解释之前为什么要那样设置 `pc` 字段的值。调用 `gostartcallfn` ：

```
gostartcallfn(&newg.sched, fn) //调整sched成员和newg的栈
```

传入 newg.sched 和 fn。

```
func gostartcallfn(gobuf *gobuf, fv *funcval) {
    var fn unsafe.Pointer
    if fv != nil {
        // fn: gorotine 的入口地址，初始化时对应的是 runtime.main
        fn = unsafe.Pointer(fv.fn)
    } else {
        fn = unsafe.Pointer(funcPC(nilfunc))
    }
    gostartcall(gobuf, fn, unsafe.Pointer(fv))
}

func gostartcall(buf *gobuf, fn, ctxt unsafe.Pointer) {
    // newg 的栈顶，目前 newg 栈上只有 fn 函数的参数，sp 指向的是 fn 的第一参数
    sp := buf.sp

    // ..........................

    // 为返回地址预留空间
    sp -= sys.PtrSize
    // 这里填的是 newproc1 函数里设置的 goexit 函数的第二条指令
    // 伪装 fn 是被 goexit 函数调用的，使得 fn 执行完后返回到 goexit 继续执行，从而完成清理工作
    *(*uintptr)(unsafe.Pointer(sp)) = buf.pc
    // 重新设置 buf.sp
    buf.sp = sp
    // 当 goroutine 被调度起来执行时，会从这里的 pc 值开始执行，初始化时就是 runtime.main
    buf.pc = uintptr(fn)
    buf.ctxt = ctxt
}
```

函数 `gostartcallfn` 只是拆解出了包含在 funcval 结构体里的函数指针，转过头就调用 `gostartcall` 。将 sp 减小了一个指针的位置，这是给返回地址留空间。果然接着就把 buf.pc 填入了栈顶的位置：

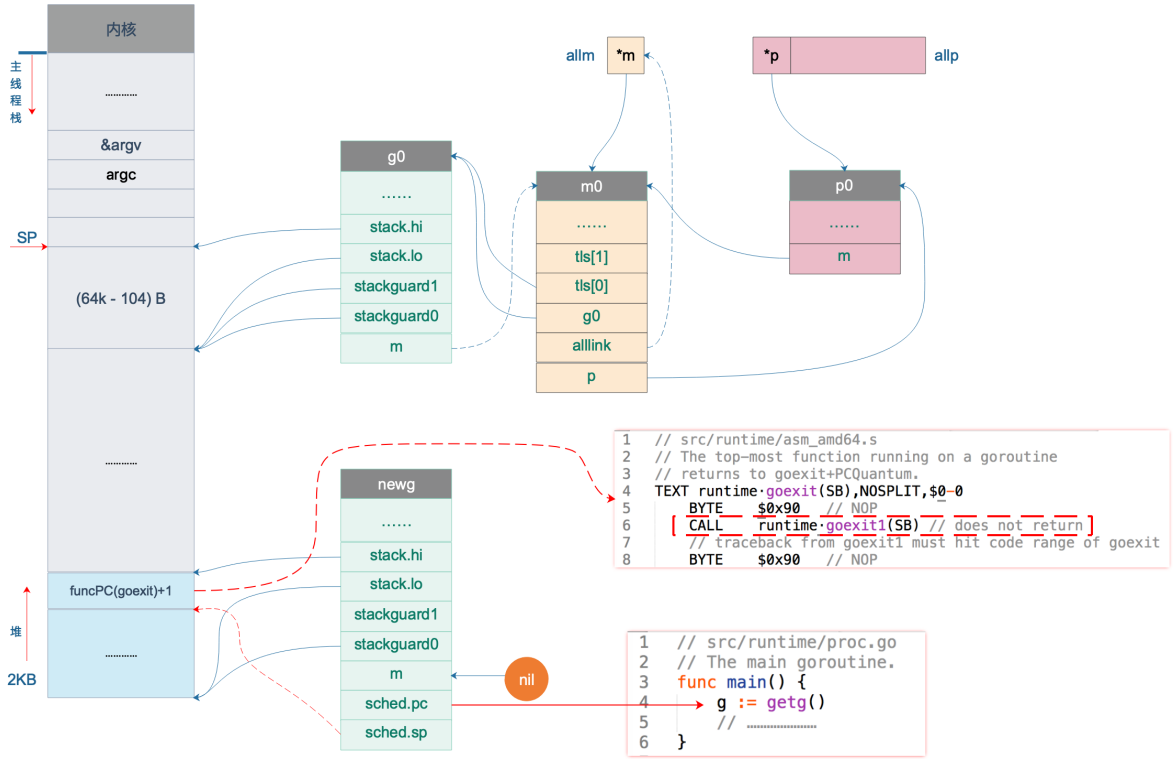
```
*(*uintptr)(unsafe.Pointer(sp)) = buf.pc
```

原来 buf.pc 只是做了一个搬运工，搞什么啊。重新设置 buf.sp 为送减掉一个指针位置之后的值，设置 buf.pc 为 fn，指向要执行的函数，这里就是指的 runtime.main 函数。

对嘛，这才是应有的操作。之后，当调度器“光顾”此 goroutine 时，取出 buf.sp 和 buf.pc，恢复 CPU 相应的寄存器，就可以构造出 goroutine 的运行环境。

而 goexit 函数也通过“偷天换日”将自己的地址“强行”放到 newg 的栈顶，达到自己不可告人的目的：每个 goroutine 执行完之后，都要经过我的一些清理工作，才能“放行”。这样一说，goexit 函数还真是无私，默默地做一些“扫尾”的工作。

设置完 newg.sched 这后，我们的图又可以前进一步：




上图中，newg 新增了 sched.pc 指向 `runtime.main` 函数，当它被调度起来执行时，就从这里开始；新增了 sched.sp 指向了 newg 栈顶位置，同时，newg 栈顶位置的内容是一个跳转地址，指向 `runtime.goexit` 的第二条指令，当 goroutine 退出时，这条地址会载入 CPU 的 PC 寄存器，跳转到这里执行“扫尾”工作。

之后，将 newg 的状态改为 runnable，设置 goroutine 的 id：

```
// 设置 g 的状态为 _Grunnable，可以运行了
casgstatus(newg, _Gdead, _Grunnable)
newg.goid = int64(_p_.goidcache)
```

每个 P 每次会批量（16个）申请 id，每次调用 newproc 函数，新创建一个 goroutine，id 加 1。因此 g0 的 id 是 0，而 main goroutine 的 id 就是 1。

`newg` 的状态变成可执行后（Runnable），就可以将它加入到 P 的本地运行队列里，等待调度。所以，goroutine 何时被执行，用户代码决定不了。来看源码：

```
// 将 G 放入 _p_ 的本地待运行队列
runqput(_p_, newg, true)

// runqput 尝试将 g 放到本地可执行队列里。
// 如果 next 为假，runqput 将 g 添加到可运行队列的尾部
// 如果 next 为真，runqput 将 g 添加到 p.runnext 字段
```

```
// 如果 run queue 满了，runnext 将 g 放到全局队列里
//
// runnext 成员中的 goroutine 会被优先调度起来运行
func runqput(_p_ *p, gp *g, next bool) {
    // ......................

    if next {
    retryNext:
        oldnext := _p_.runnext
        if !_p_.runnext.cas(oldnext, guintptr(unsafe.Pointer(gp))) {
            // 有其它线程在操作 runnext 成员，需要重试
            goto retryNext
        }
        // 老的 runnext 为 nil，不用管了
        if oldnext == 0 {
            return
        }
        // 把之前的 runnext 踢到正常的 runq 中
        // 原本存放在 runnext 的 gp 放入 runq 的尾部
        gp = oldnext.ptr()
    }

retry:
    h := atomic.Load(&_p_.runqhead) // load-acquire, synchronize with consumers
    t := _p_.runqtail
    // 如果 P 的本地队列没有满，入队
    if t-h < uint32(len(_p_.runq)) {
        _p_.runq[t%uint32(len(_p_.runq))].set(gp)
        // 原子写入
        atomic.Store(&_p_.runqtail, t+1) // store-release, makes the item available for consumption
        return
    }
    // 可运行队列已经满了，放入全局队列了
    if runqputslow(_p_, gp, h, t) {
        return
    }
    // the queue is not full, now the put above must succeed
    // 没有成功放入全局队列，说明本地队列没满，重试一下
    goto retry
}
```

`runqput` 函数的主要作用就是将新创建的 goroutine 加入到 P 的可运行队列，如果本地队列满了，则加入到全局可运行队列。前两个参数都好理解，最后一个参数 `next` 的作用是，当它为 true 时，会将 newg 加入到 P 的 runnext 字段，具有最高优先级，将先于普通队列中的 goroutine 得到执行。

先将 P 老的 runnext 成员取出，接着用一个原子操作 cas 来试图将 runnext 成员设置成 newg，目的是防止其他线程在同时修改 runnext 字段。

设置成功之后，相当于 newg “挤掉” 了原来老的处于 runnext 的 goroutine，还得给人遣散费，安顿好人家嘛，不然和强盗有何区别？

“安顿”的动作在 retry 代码段中执行。先通过 `head` ， `tail` ， `len(_p_.runq)` 来判断队列是否已满，如果没满，则直接写到队列尾部，同时修改队列尾部的指针。

```
// store-release, makes it available for consumption
atomic.Store(&_p_.runqtail, t+1)
```

这里使用原子操作写入 runtail，防止编译器和 CPU 指令重排，保证上一行代码对 runq 的修改发生在修改 runqtail 之前，并且保证当前线程对队列的修改对其它线程立即可见。

如果本地队列满了，那就只能试图将 newg 添加到全局可运行队列中了。调用 `runqputslow(_p_, gp, h, t)` 完成。

```
// 将 g 和 _p_ 本地队列的一半 goroutine 放入全局队列。
// 因为要获取锁，所以会慢
func runqputslow(_p_ *p, gp *g, h, t uint32) bool {
    var batch [len(_p_.runq)/2 + 1]*g

    // First, grab a batch from local queue.
    n := t - h
    n = n / 2
    if n != uint32(len(_p_.runq)/2) {
        throw("runqputslow: queue is not full")
    }
    for i := uint32(0); i < n; i++ {
        batch[i] = _p_.runq[(h+i)%uint32(len(_p_.runq))].ptr()
    }
    // 如果 cas 操作失败，说明本地队列不满了，直接返回
    if !atomic.Cas(&_p_.runqhead, h, h+n) { // cas-release, commits consume
        return false
    }
    batch[n] = gp

    // ..............................

    // Link the goroutines.
    // 全局运行队列是一个链表，这里首先把所有需要放入全局运行队列的 g 链接起来，
    // 减小锁粒度，从而降低锁冲突，提升性能
    for i := uint32(0); i < n; i++ {
        batch[i].schedlink.set(batch[i+1])
    }

    // Now put the batch on global queue.
    lock(&sched.lock)
    globrunqputbatch(batch[0], batch[n], int32(n+1))
    unlock(&sched.lock)
    return true
}
```

先将 P 本地队列里所有的 goroutine 加入到一个数组中，数组长度为 `len(_p_.runq)/2 + 1`，也就是 runq 的一半加上 newg。

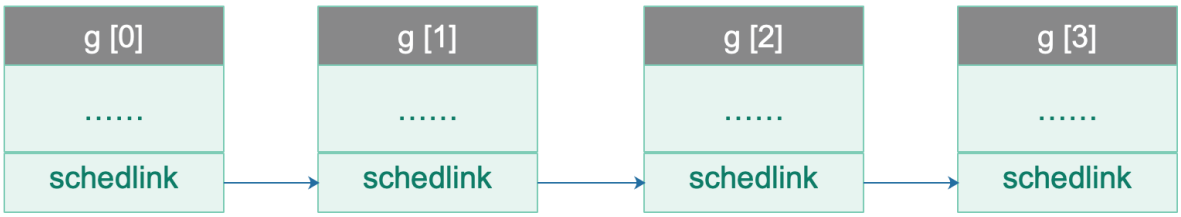
接着，将从 runq 的头部开始的前一半 goroutine 存入 bacth 数组。然后，使用原子操作尝试修改 P 的队列头，因为出队了一半 goroutine，所以 head 要向后移动 1/2 的长度。如果修改失败，说明 runq 的本地队列被其他线程修改了，因此后面的操作就不进行了，直接返回 false，表示 newg 没被添加进来。

```
batch[n] = gp
```

将 newg 本身添加到数组。

通过循环将 batch 数组里的所有 g 串成链表：

```
for i := uint32(0); i < n; i++ {
    batch[i].schedlink.set(batch[i+1])
}
```



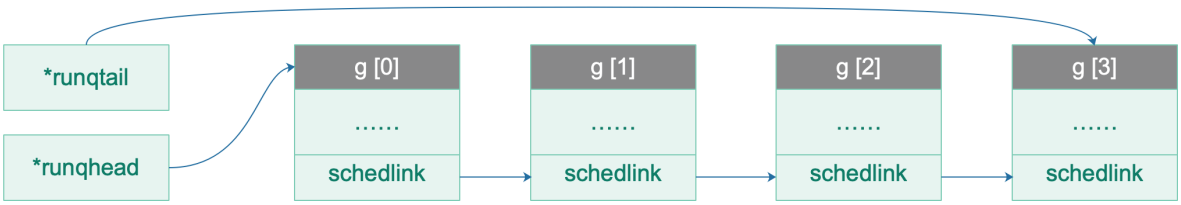

最后，将链表添加到全局队列中。由于操作的是全局队列，因此需要获取锁，因为存在竞争，所以代价较高。这也是本地可运行队列存在的原因。调用 `globrunqputbatch(batch[0], batch[n], int32(n+1))` ：

```
// Put a batch of runnable goroutines on the global runnable queue.
// Sched must be locked.
func globrunqputbatch(ghead *g, gtail *g, n int32) {
    gtail.schedlink = 0
    if sched.runqtail != 0 {
        sched.runqtail.ptr().schedlink.set(ghead)
    } else {
        sched.runqhead.set(ghead)
    }
    sched.runqtail.set(gtail)
    sched.runqsize += n
}
```

如果全局的队列尾 `sched.runqtail` 不为空，则直接将其和前面生成的链表头相接，否则说明全局的可运行列队为空，那就直接将前面生成的链表头设置到 sched.runqhead。

最后，再设置好队列尾，增加 runqsize。

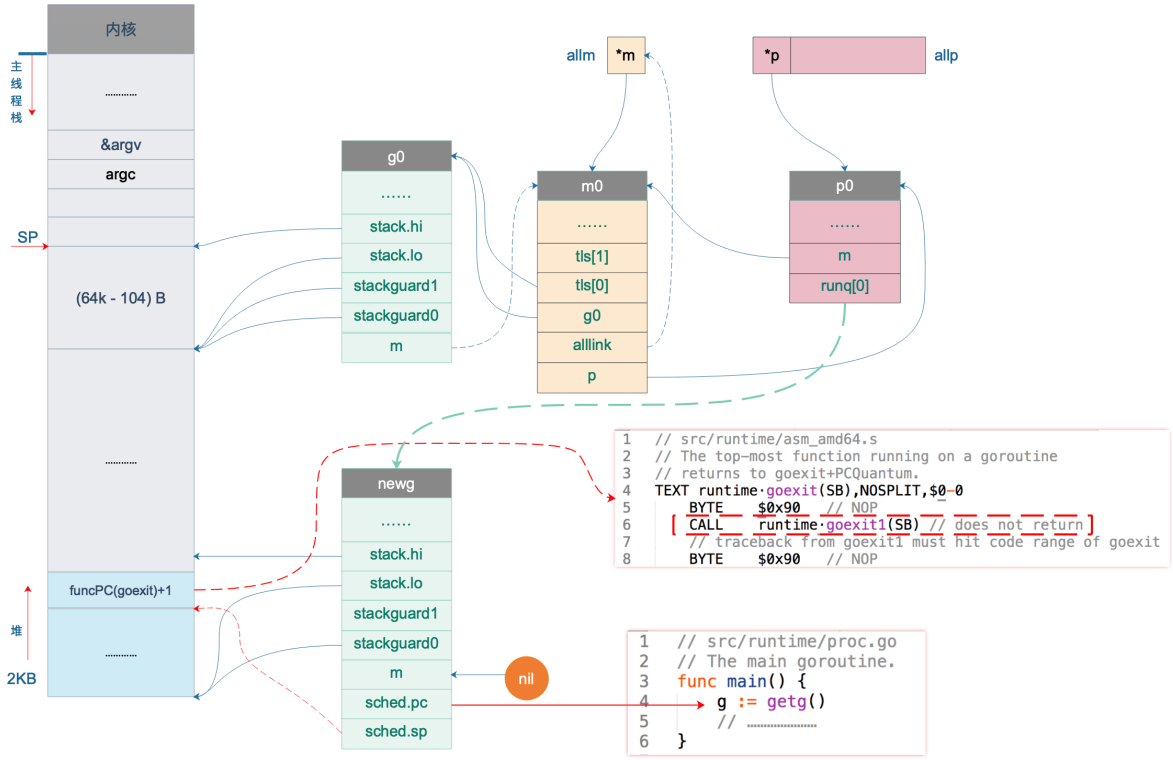
设置完成之后：




再回到 `runqput` 函数，如果将 newg 添加到全局队列失败了，说明本地队列在此过程中发生了变化，又有了位置可以添加 newg，因此重试 retry 代码段。我们也可以发现，P 的本地可运行队列的长度为 256，它是一个循环队列，因此最多只能放下 256 个 goroutine。

因为本文还是处于初始化的场景，所以 newg 被成功放入 p0 的本地可运行队列，等待被调度。

将我们的图再完善一下：




# 参考资料

【阿波张 Go语言调度器之调度 main 】https://mp.weixin.qq.com/s/8eJm5hjwKXya85VnT4y8Cw

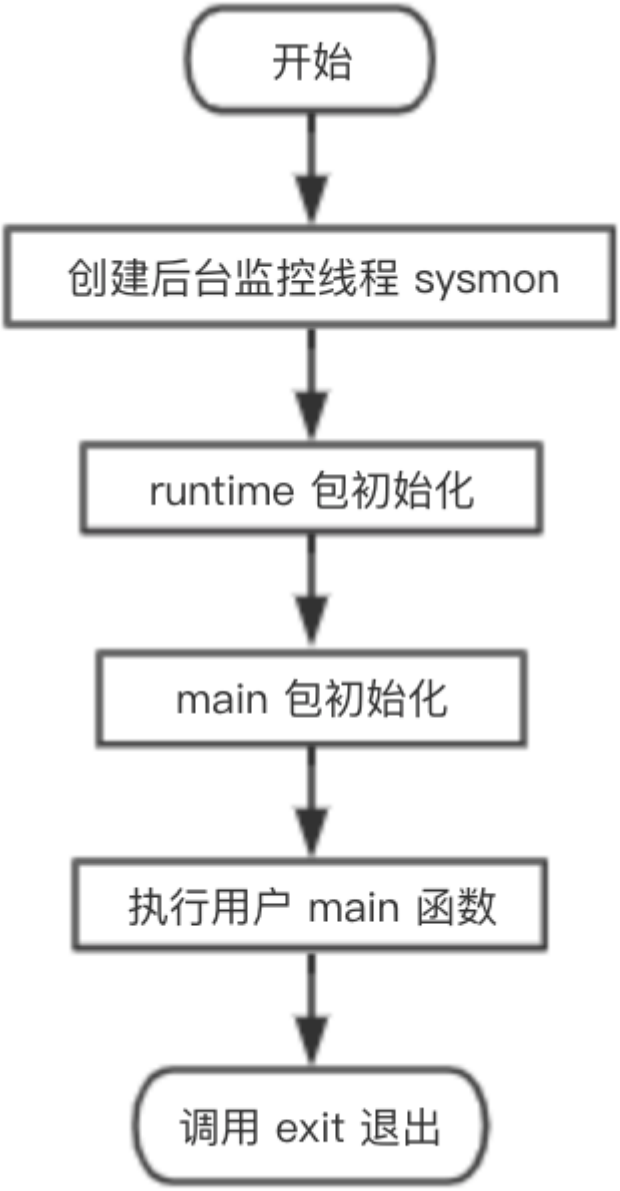
# goroutine 如何退出

上一讲说到调度器将 main goroutine 推上舞台，为它铺好了道路，开始执行 `runtime.main` 函数。这一讲，我们探索 main goroutine 以及普通 goroutine 从执行到退出的整个过程。

```
// The main goroutine.
func main() {
    // g = main goroutine，不再是 g0 了
    g := getg()

    // ......................

    if sys.PtrSize == 8 {
        maxstacksize = 1000000000
    } else {
        maxstacksize = 250000000
    }

    // Allow newproc to start new Ms.
    mainStarted = true

    systemstack(func() {
        // 创建监控线程，该线程独立于调度器，不需要跟 p 关联即可运行
        newm(sysmon, nil)
    })

    lockOSThread()

    if g.m != &m0 {
        throw("runtime.main not on m0")
    }

    // 调用 runtime 包的初始化函数，由编译器实现
    runtime_init() // must be before defer
    if nanotime() == 0 {
        throw("nanotime returning zero")
    }

    // Defer unlock so that runtime.Goexit during init does the unlock too.
    needUnlock := true
    defer func() {
        if needUnlock {
            unlockOSThread()
        }
    }()

    // Record when the world started. Must be after runtime_init
    // because nanotime on some platforms depends on startNano.
    runtimeInitTime = nanotime()

    // 开启垃圾回收器
    gcenable()

    main_init_done = make(chan bool)

    // ......................

    // main 包的初始化，递归的调用我们 import 进来的包的初始化函数
    fn := main_init
```

```
    fn()
    close(main_init_done)

    needUnlock = false
    unlockOSThread()

    // ......................

    // 调用 main.main 函数
    fn = main_main
    fn()
    if raceenabled {
        racefini()
    }

    // ......................

    // 进入系统调用，退出进程，可以看出 main goroutine 并未返回，而是直接进入系统调用退出进程了
    exit(0)
    // 保护性代码，如果 exit 意外返回，下面的代码会让该进程 crash 死掉
    for {
        var x *int32
        *x = 0
    }
}
```

`main` 函数执行流程如下图：




从流程图可知，main goroutine 执行完之后就直接调用 `exit(0)` 退出了，这会导致整个进程退出，太粗暴了。

不过，main goroutine 实际上就是代表用户的 main 函数，它都执行完了，肯定是用户的任务都执行完了，直接退出就可以了，就算有其他的 goroutine 没执行完，同样会直接退出。

```
package main

import "fmt"

func main() {
    go func() {fmt.Println("hello qcrao.com")}()
}
```

在这个例子中，main gorutine 退出时，还来不及执行 `go 出去` 的函数，整个进程就直接退出了，打印语句不会执行。因此，main goroutine 不会等待其他 goroutine 执行完再退出，知道这个有时能解释一些现象，比如上面那个例子。

这时，心中可能会跳出疑问，我们在新创建 goroutine 的时候，不是整出了个“偷天换日”，风风火火地设置了 goroutine 退出时应该跳到 `runtime.goexit` 函数吗，怎么这会不用了，闲得慌？

回顾一下上一讲的内容，跳转到 main 函数的两行代码：

```
// 把 sched.pc 值放入 BX 寄存器
MOVQ    gobuf_pc(BX), BX
// JMP 把 BX 寄存器的包含的地址值放入 CPU 的 IP 寄存器，于是，CPU 跳转到该地址继续执行指令
JMP    BX
```

直接使用了一个跳转，并没有使用 `CALL` 指令，而 runtime.main 函数中确实也没有 `RET` 返回的指令。所以，main goroutine 执行完后，直接调用 exit(0) 退出整个进程。

那之前整地“偷天换日”还有用吗？有的！这是针对非 main goroutine 起作用。

参考资料【阿波张 非 goroutine 的退出】中用调试工具验证了非 main goroutine 的退出，感兴趣的可以去跟着实践一遍。

我们继续探索非 main goroutine （后文我们就称 gp 好了）的退出流程。

`gp` 执行完后，RET 指令弹出 `goexit` 函数地址（实际上是 funcPC(goexit)+1），CPU 跳转到 `goexit` 的第二条指令继续执行：

```
// src/runtime/asm_amd64.s

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

直接调用 `runtime·goexit1` ：

```
// src/runtime/proc.go
// Finishes execution of the current goroutine.
func goexit1() {
    // ......................
    mcall(goexit0)
}
```

调用 `mcall` 函数：

```
// 切换到 g0 栈，执行 fn(g)
// Fn 不能返回
TEXT runtime·mcall(SB), NOSPLIT, $0-8
    // 取出参数的值放入 DI 寄存器，它是 funcval 对象的指针，此场景中 fn.fn 是 goexit0 的地址
    MOVQ    fn+0(FP), DI

    get_tls(CX)
    // AX = g
    MOVQ    g(CX), AX   // save state in g->sched
    // mcall 返回地址放入 BX
    MOVQ    0(SP), BX   // caller's PC
    // g.sched.pc = BX，保存 g 的 PC
    MOVQ    BX, (g_sched+gobuf_pc)(AX)
    LEAQ    fn+0(FP), BX    // caller's SP
```

```
    // 保存 g 的 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
    // SI = g0
    MOVQ    m_g0(BX), SI
    CMPQ    SI, AX  // if g == m->g0 call badmcall
    JNE 3(PC)
    MOVQ    $runtime·badmcall(SB), AX
    JMP AX
    // 把 g0 的地址设置到线程本地存储中
    MOVQ    SI, g(CX)    // g = m->g0
    // 从 g 的栈切换到了 g0 的栈D
    MOVQ    (g_sched+gobuf_sp)(SI), SP  // sp = m->g0->sched.sp
    // AX = g，参数入栈
    PUSHQ   AX
    MOVQ    DI, DX
    // DI 是结构体 funcval 实例对象的指针，它的第一个成员才是 goexit0 的地址
    // 读取第一个成员到 DI 寄存器
    MOVQ    0(DI), DI
    // 调用 goexit0(g)
    CALL    DI
    POPQ    AX
    MOVQ    $runtime·badmcall2(SB), AX
    JMP AX
    RET
```

函数参数是：

```
type funcval struct {
    fn uintptr
    // variable-size, fn-specific data here
}
```

字段 fn 就表示 goexit0 函数的地址。

L5 将函数参数保存到 DI 寄存器，这里 fn.fn 就是 goexit0 的地址。

L7 将 tls 保存到 CX 寄存器，L9 将 当前线程指向的 goroutine （非 main goroutine，称为 gp）保存到 AX 寄存器，L11 将调用者（调用 mcall 函数）的栈顶，这里就是 mcall 完成后的返回地址，存入 BX 寄存器。

L13 将 mcall 的返回地址保存到 gp 的 g.sched.pc 字段，L14 将 gp 的栈顶，也就是 SP 保存到 BX 寄存器，L16 将 SP 保存到 gp 的 g.sched.sp 字段，L17 将 g 保存到 gp 的 g.sched.g 字段，L18 将 BP 保存 到 gp 的 g.sched.bp 字段。这一段主要是保存 gp 的调度信息。

L21 将当前指向的 g 保存到 BX 寄存器，L22 将 g.m 字段保存到 BX 寄存器，L23 将 g.m.g0 字段保存到 SI，g.m.g0 就是当前工作线程的 g0。

现在，SI = g0， AX = gp，L25 判断 gp 是否是 g0，如果 gp == g0 说明有问题，执行 runtime·badmcall。正常情况下，PC 值加 3，跳过下面的两条指令，直接到达 L30。

L30 将 g0 的地址设置到线程本地存储中，L32 将 g0.SP 设置到 CPU 的 SP 寄存器，这也就意味着我们从 gp 栈切换到了 g0 的栈，要变天了！

L34 将参数 gp 入栈，为调用 goexit0 构造参数。L35 将 DI 寄存器的内容设置到 DX 寄存器，DI 是结构体 funcval 实例对象的指针，它的第一个成员才是 goexit0 的地址。L36 读取 DI 第一成员，也就是 goexit0 函数的地址。

L40 调用 goexit0 函数，这已经是在 g0 栈上执行了，函数参数就是 gp。

到这里，就会去执行 goexit0 函数，注意，这里永远都不会返回。所以，在 CALL 指令后面，如果返回了，又会去调用 `runtime.badmcall2` 函数去处理意外情况。

来继续看 goexit0：

```
// goexit continuation on g0.
// 在 g0 上执行
func goexit0(gp *g) {
    // g0
    _g_ := getg()

    casgstatus(gp, _Grunning, _Gdead)
    if isSystemGoroutine(gp) {
        atomic.Xadd(&sched.ngsys, -1)
    }

    // 清空 gp 的一些字段
    gp.m = nil
    gp.lockedm = nil
    _g_.m.lockedg = nil
    gp.paniconfault = false
    gp._defer = nil // should be true already but just in case.
    gp._panic = nil // non-nil for Goexit during panic. points at stack-allocated data.
    gp.writebuf = nil
    gp.waitreason = ""
    gp.param = nil
    gp.labels = nil
    gp.timer = nil

    // Note that gp's stack scan is now "valid" because it has no
    // stack.
    gp.gcscanvalid = true
    // 解除 g 与 m 的关系
    dropg()

    if _g_.m.locked&^_LockExternal != 0 {
        print("invalid m->locked = ", _g_.m.locked, "\n")
        throw("internal lockOSThread error")
    }
    _g_.m.locked = 0
    // 将 g 放入 free 队列缓存起来
    gfput(_g_.m.p.ptr(), gp)
    schedule()
}
```

它主要完成最后的清理工作：

1. 把 g 的状态从 `_Grunning` 更新为 `_Gdead`；

2. 清空 g 的一些字段；

3. 调用 dropg 函数解除 g 和 m 之间的关系，其实就是设置 g->m = nil, m->currg = nil；

> 4. 把 g 放入 p 的 freeg 队列缓存起来供下次创建 g 时快速获取而不用从内存分配。freeg 就是 g 的一个对象池；

> 5. 调用 schedule 函数再次进行调度。

到这里，gp 就完成了它的历史使命，功成身退，进入了 goroutine 缓存池，待下次有任务再重新启用。

而工作线程，又继续调用 schedule 函数进行新一轮的调度，整个过程形成了一个循环。

总结一下，main goroutine 和普通 goroutine 的退出过程：

对于 main goroutine，在执行完用户定义的 main 函数的所有代码后，直接调用 exit(0) 退出整个进程，非常霸道。

对于普通 goroutine 则没这么“舒服”，需要经历一系列的过程。先是跳转到提前设置好的 goexit 函数的第二条指令，然后调用 runtime.goexit1，接着调用 `mcall(goexit0)` ，而 mcall 函数会切换到 g0 栈，运行 goexit0 函数，清理 goroutine 的一些字段，并将其添加到 goroutine 缓存池里，然后进入 schedule 调度循环。到这里，普通 goroutine 才算完成使命。

# 参考资料

【阿波张 非 main goroutine 的退出及调度循环】https://mp.weixin.qq.com/s/XttP9q7-PO7VXhskaBzGqA

# goroutine 调度时机有哪些

在四种情形下，goroutine 可能会发生调度，但也并不一定会发生，只是说 Go scheduler 有机会进行调度。

| 情形 | 说明 |
| --- | --- |
| 使用关键字 `go` | go 创建一个新的 goroutine，Go scheduler 会考虑调度 |
| GC | 由于进行 GC 的 goroutine 也需要在 M 上运行，<br>因此肯定会发生调度。当然，Go scheduler 还会做很多其他的调度，<br>例如调度不涉及堆访问的 goroutine 来运行。GC 不管栈上的内存，<br>只会回收堆上的内存 |
| 系统调用 | 当 goroutine 进行系统调用时，会阻塞 M，所以它会被调度走，<br>同时一个新的 goroutine 会被调度上来 |
| 内存同步访问 | atomic，mutex，channel 操作等会使 goroutine 阻塞，<br>因此会被调度走。等条件满足后（例如其他 goroutine 解锁了）<br>还会被调度上来继续运行 |

# goroutine和线程的区别

谈到 goroutine，绕不开的一个话题是：它和 thread 有什么区别？

参考资料【How Goroutines Work】告诉我们可以从三个角度区别：内存消耗、创建与销毁、切换。

- 内存占用

创建一个 goroutine 的栈内存消耗为 2 KB，实际运行过程中，如果栈空间不够用，会自动进行扩容。创建一个 thread 则需要消耗 1 MB 栈内存，而且还需要一个被称为 “a guard page” 的区域用于和其他 thread 的栈空间进行隔离。

对于一个用 Go 构建的 HTTP Server 而言，对到来的每个请求，创建一个 goroutine 用来处理是非常轻松的一件事。而如果用一个使用线程作为并发原语的语言构建的服务，例如 Java 来说，每个请求对应一个线程则太浪费资源了，很快就会出 OOM 错误（OutOfMermoryError）。

- 创建和销毁

Thread 创建和销毁都会有巨大的消耗，因为要和操作系统打交道，是内核级的，通常解决的办法就是线程池。而 goroutine 因为是由 Go runtime 负责管理的，创建和销毁的消耗非常小，是用户级。

- 切换

当 threads 切换时，需要保存各种寄存器，以便将来恢复：

> 16 general purpose registers, PC (Program Counter), SP (Stack Pointer), segment registers, 16 XMM registers, FP coprocessor state, 16 AVX registers, all MSRs etc.

而 goroutines 切换只需保存三个寄存器：Program Counter, Stack Pointer and BP。

一般而言，线程切换会消耗 1000-1500 纳秒，一个纳秒平均可以执行 12-18 条指令。所以由于线程切换，执行指令的条数会减少 12000-18000。

Goroutine 的切换约为 200 ns，相当于 2400-3600 条指令。

因此，goroutines 切换成本比 threads 要小得多。

# GPM 是什么

G、P、M 是 Go 调度器的三个核心组件，各司其职。在它们精密地配合下，Go 调度器得以高效运转，这也是 Go 天然支持高并发的内在动力。今天这篇文章我们来深入理解 GPM 模型。

先看 G，取 goroutine 的首字母，主要保存 goroutine 的一些状态信息以及 CPU 的一些寄存器的值，例如 IP 寄存器，以便在轮到本 goroutine 执行时，CPU 知道要从哪一条指令处开始执行。

> 当 goroutine 被调离 CPU 时，调度器负责把 CPU 寄存器的值保存在 g 对象的成员变量之中。

> 当 goroutine 被调度起来运行时，调度器又负责把 g 对象的成员变量所保存的寄存器值恢复到 CPU 的寄存器。

本系列使用的代码版本是 1.9.2，来看一下 g 的源码：

```
type g struct {

    // goroutine 使用的栈
    stack       stack   // offset known to runtime/cgo
    // 用于栈的扩张和收缩检查，抢占标志
    stackguard0 uintptr // offset known to liblink
    stackguard1 uintptr // offset known to liblink

    _panic         *_panic // innermost panic - offset known to liblink
    _defer         *_defer // innermost defer
    // 当前与 g 绑定的 m
    m              *m      // current m; offset known to arm liblink
    // goroutine 的运行现场
    sched          gobuf
    syscallsp      uintptr        // if status==Gsyscall, syscallsp = sched.sp to use during gc
    syscallpc      uintptr        // if status==Gsyscall, syscallpc = sched.pc to use during gc
    stktopsp       uintptr        // expected sp at top of stack, to check in traceback
    // wakeup 时传入的参数
    param          unsafe.Pointer // passed parameter on wakeup
    atomicstatus   uint32
    stackLock      uint32 // sigprof/scang lock; TODO: fold in to atomicstatus
    goid           int64
    // g 被阻塞之后的近似时间
    waitsince      int64  // approx time when the g become blocked
    // g 被阻塞的原因
    waitreason     string // if status==Gwaiting
    // 指向全局队列里下一个 g
    schedlink      guintptr
    // 抢占调度标志。这个为 true 时，stackguard0 等于 stackpreempt
    preempt        bool     // preemption signal, duplicates stackguard0 = stackpreempt
    paniconfault   bool     // panic (instead of crash) on unexpected fault address
    preemptscan    bool     // preempted g does scan for gc
    gcscandone     bool     // g has scanned stack; protected by _Gscan bit in status
    gcscanvalid    bool     // false at start of gc cycle, true if G has not run since last scan; TODO: remov
e?
    throwsplit     bool     // must not split stack
    raceignore     int8     // ignore race detection events
    sysblocktraced bool     // StartTrace has emitted EvGoInSyscall about this goroutine
    // syscall 返回之后的 cputicks，用来做 tracing
    sysexitticks   int64    // cputicks when syscall has returned (for tracing)
    traceseq       uint64   // trace event sequencer
    tracelastp     puintptr // last P emitted an event for this goroutine
    // 如果调用了 LockOsThread，那么这个 g 会绑定到某个 m 上
    lockedm        *m
```

```
    sig             uint32
    writebuf        []byte
    sigcode0        uintptr
    sigcode1        uintptr
    sigpc           uintptr
    // 创建该 goroutine 的语句的指令地址
    gopc            uintptr // pc of go statement that created this goroutine
    // goroutine 函数的指令地址
    startpc         uintptr // pc of goroutine function
    racectx         uintptr
    waiting         *sudog         // sudog structures this g is waiting on (that have a valid elem ptr); in l
ock order
    cgoCtxt         []uintptr      // cgo traceback context
    labels          unsafe.Pointer // profiler labels
    // time.Sleep 缓存的定时器
    timer           *timer         // cached timer for time.Sleep

    gcAssistBytes int64
}
```

源码中，比较重要的字段我已经作了注释，其他未作注释的与调度关系不大或者我暂时也没有理解的。

`g` 结构体关联了两个比较简单的结构体，stack 表示 goroutine 运行时的栈：

```
// 描述栈的数据结构，栈的范围：[lo, hi)
type stack struct {
    // 栈顶，低地址
    lo uintptr
    // 栈低，高地址
    hi uintptr
}
```

Goroutine 运行时，光有栈还不行，至少还得包括 PC，SP 等寄存器，gobuf 就保存了这些值：

```
type gobuf struct {
    // 存储 rsp 寄存器的值
    sp   uintptr
    // 存储 rip 寄存器的值
    pc   uintptr
    // 指向 goroutine
    g    guintptr
    ctxt unsafe.Pointer // this has to be a pointer so that gc scans it
    // 保存系统调用的返回值
    ret  sys.Uintreg
    lr   uintptr
    bp   uintptr // for GOEXPERIMENT=framepointer
}
```

再来看 M，取 machine 的首字母，它代表一个工作线程，或者说系统线程。G 需要调度到 M 上才能运行，M 是真正工作的人。结构体 m 就是我们常说的 M，它保存了 M 自身使用的栈信息、当前正在 M 上执行的 G 信息、与之绑定的 P 信息……

当 M 没有工作可做的时候，在它休眠前，会“自旋”地来找工作：检查全局队列，查看 network poller，试图执行 gc 任务，或者“偷”工作。

结构体 m 的源码如下：

```
// m 代表工作线程，保存了自身使用的栈信息
type m struct {
```

```
    // 记录工作线程（也就是内核线程）使用的栈信息。在执行调度代码时需要使用
    // 执行用户 goroutine 代码时，使用用户 goroutine 自己的栈，因此调度时会发生栈的切换
    g0      *g     // goroutine with scheduling stack/
    morebuf gobuf  // gobuf arg to morestack
    divmod  uint32 // div/mod denominator for arm - known to liblink

    // Fields not known to debuggers.
    procid        uint64     // for debuggers, but offset not hard-coded
    gsignal       *g         // signal-handling g
    sigmask       sigset     // storage for saved signal mask
    // 通过 tls 结构体实现 m 与工作线程的绑定
    // 这里是线程本地存储
    tls           [6]uintptr // thread-local storage (for x86 extern register)
    mstartfn      func()
    // 指向正在运行的 gorutine 对象
    curg          *g       // current running goroutine
    caughtsig     guintptr // goroutine running during fatal signal
    // 当前工作线程绑定的 p
    p             puintptr // attached p for executing go code (nil if not executing go code)
    nextp         puintptr
    id            int32
    mallocing     int32
    throwing      int32
    // 该字段不等于空字符串的话，要保持 curg 始终在这个 m 上运行
    preemptoff    string // if != "", keep curg running on this m
    locks         int32
    softfloat     int32
    dying         int32
    profilehz     int32
    helpgc        int32
    // 为 true 时表示当前 m 处于自旋状态，正在从其他线程偷工作
    spinning      bool // m is out of work and is actively looking for work
    // m 正阻塞在 note 上
    blocked       bool // m is blocked on a note
    // m 正在执行 write barrier
    inwb          bool // m is executing a write barrier
    newSigstack   bool // minit on C thread called sigaltstack
    printlock     int8
    // 正在执行 cgo 调用
    incgo         bool // m is executing a cgo call
    fastrand      uint32
    // cgo 调用总计数
    ncgocall      uint64      // number of cgo calls in total
    ncgo          int32       // number of cgo calls currently in progress
    cgoCallersUse uint32      // if non-zero, cgoCallers in use temporarily
    cgoCallers    *cgoCallers // cgo traceback if crashing in cgo call
    // 没有 goroutine 需要运行时，工作线程睡眠在这个 park 成员上，
    // 其它线程通过这个 park 唤醒该工作线程
    park          note
    // 记录所有工作线程的链表
    alllink       *m // on allm
    schedlink     muintptr
    mcache        *mcache
    lockedg       *g
    createstack   [32]uintptr // stack that created this thread.
    freglo        [16]uint32  // d[i] lsb and f[i]
    freghi        [16]uint32  // d[i] msb and f[i+16]
    fflag         uint32      // floating point compare flags
    locked        uint32      // tracking for lockosthread
    // 正在等待锁的下一个 m
    nextwaitm     uintptr     // next m waiting for lock
    needextram    bool
```

```
    traceback     uint8
    waitunlockf   unsafe.Pointer // todo go func(*g, unsafe.pointer) bool
    waitlock      unsafe.Pointer
    waittraceev   byte
    waittraceskip int
    startingtrace bool
    syscalltick   uint32
    // 工作线程 id
    thread        uintptr // thread handle

    // these are here because they are too large to be on the stack
    // of low-level NOSPLIT functions.
    libcall   libcall
    libcallpc uintptr // for cpu profiler
    libcallsp uintptr
    libcallg  guintptr
    syscall   libcall // stores syscall parameters on windows

    mOS
}
```

再来看 P，取 processor 的首字母，为 M 的执行提供“上下文”，保存 M 执行 G 时的一些资源，例如本地可运行 G 队列，memeory cache 等。

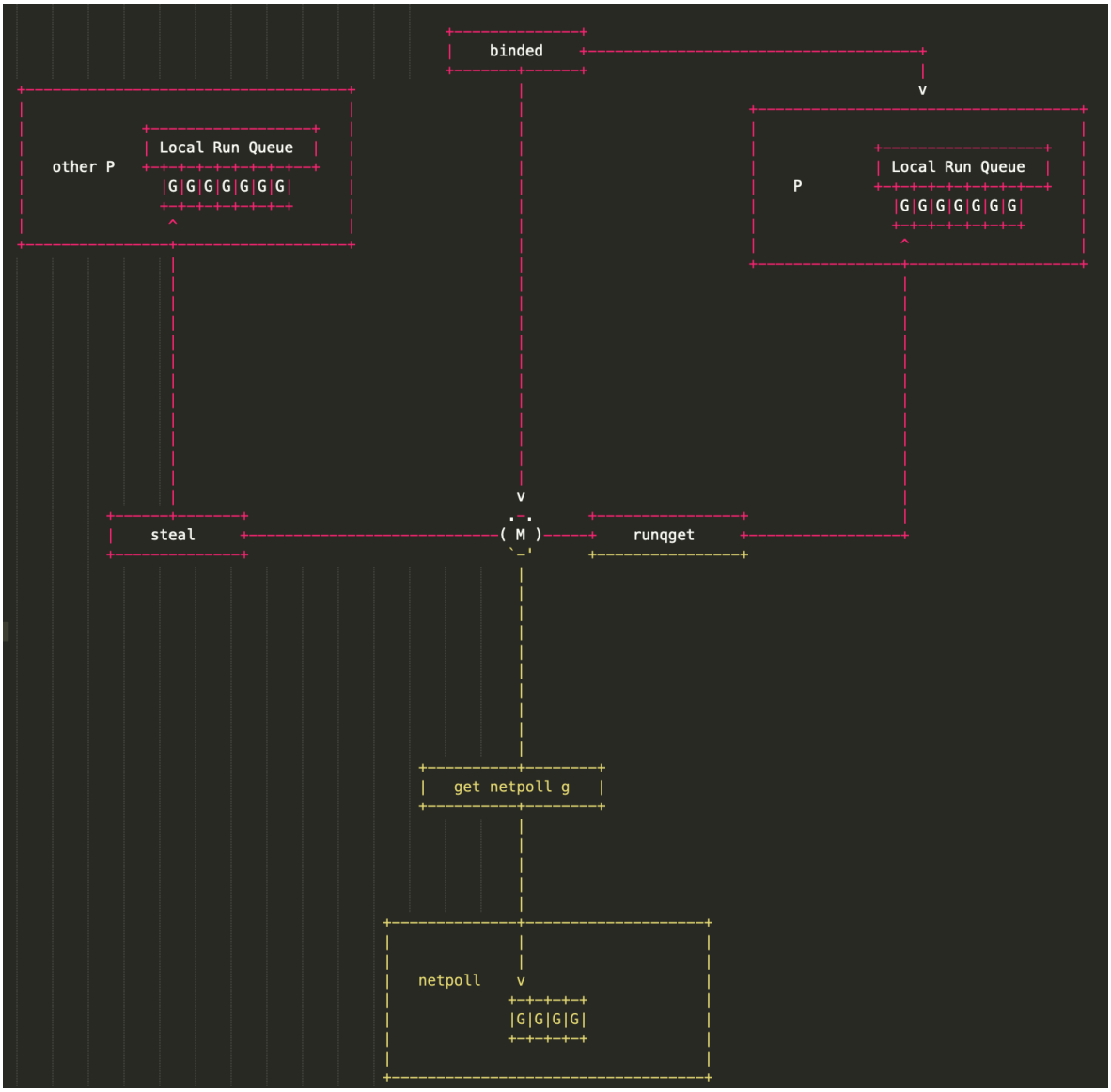
一个 M 只有绑定 P 才能执行 goroutine，当 M 被阻塞时，整个 P 会被传递给其他 M ，或者说整个 P 被接管。

```
// p 保存 go 运行时所必须的资源
type p struct {
    lock mutex

    // 在 allp 中的索引
    id          int32
    status      uint32 // one of pidle/prunning/...
    link        puintptr
    // 每次调用 schedule 时会加一
    schedtick   uint32
    // 每次系统调用时加一
    syscalltick uint32
    // 用于 sysmon 线程记录被监控 p 的系统调用时间和运行时间
    sysmontick  sysmontick // last tick observed by sysmon
    // 指向绑定的 m，如果 p 是 idle 的话，那这个指针是 nil
    m           muintptr   // back-link to associated m (nil if idle)
    mcache      *mcache
    racectx     uintptr

    deferpool    [5][]*_defer // pool of available defer structs of different sizes (see panic.go)
    deferpoolbuf [5][32]*_defer

    // Cache of goroutine ids, amortizes accesses to runtime·sched.goidgen.
    goidcache    uint64
    goidcacheend uint64

    // Queue of runnable goroutines. Accessed without lock.
    // 本地可运行的队列，不用通过锁即可访问
    runqhead uint32 // 队列头
    runqtail uint32 // 队列尾
    // 使用数组实现的循环队列
    runq     [256]guintptr

    // runnext 非空时，代表的是一个 runnable 状态的 G，
```

```
    // 这个 G 被 当前 G 修改为 ready 状态，相比 runq 中的 G 有更高的优先级。
    // 如果当前 G 还有剩余的可用时间，那么就应该运行这个 G
    // 运行之后，该 G 会继承当前 G 的剩余时间
    runnext guintptr

    // Available G's (status == Gdead)
    // 空闲的 g
    gfree    *g
    gfreecnt int32

    sudogcache []*sudog
    sudogbuf   [128]*sudog

    tracebuf traceBufPtr
    traceSwept, traceReclaimed uintptr

    palloc persistentAlloc // per-P to avoid mutex

    // Per-P GC state
    gcAssistTime     int64 // Nanoseconds in assistAlloc
    gcBgMarkWorker   guintptr
    gcMarkWorkerMode gcMarkWorkerMode
    runSafePointFn uint32 // if 1, run sched.safePointFn at next safe point

    pad [sys.CacheLineSize]byte
}
```

GPM 三足鼎力，共同成就 Go scheduler。G 需要在 M 上才能运行，M 依赖 P 提供的资源，P 则持有待运行的 G。你中有我，我中有你。

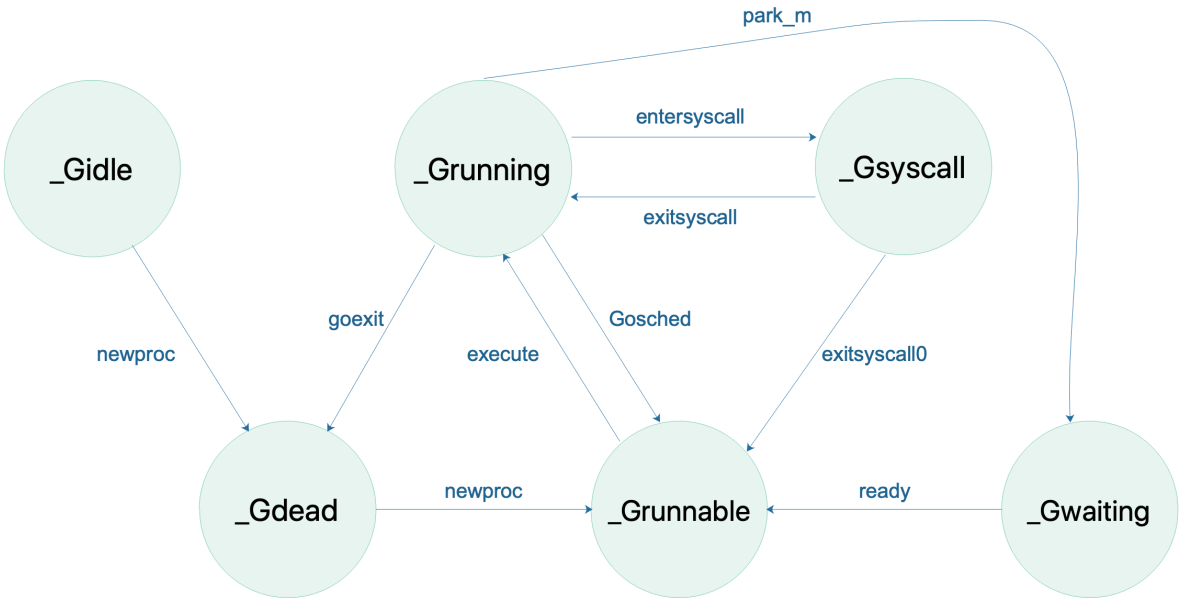
描述三者的关系：




M 会从与它绑定的 P 的本地队列获取可运行的 G，也会从 network poller 里获取可运行的 G，还会从其他 P 偷 G。

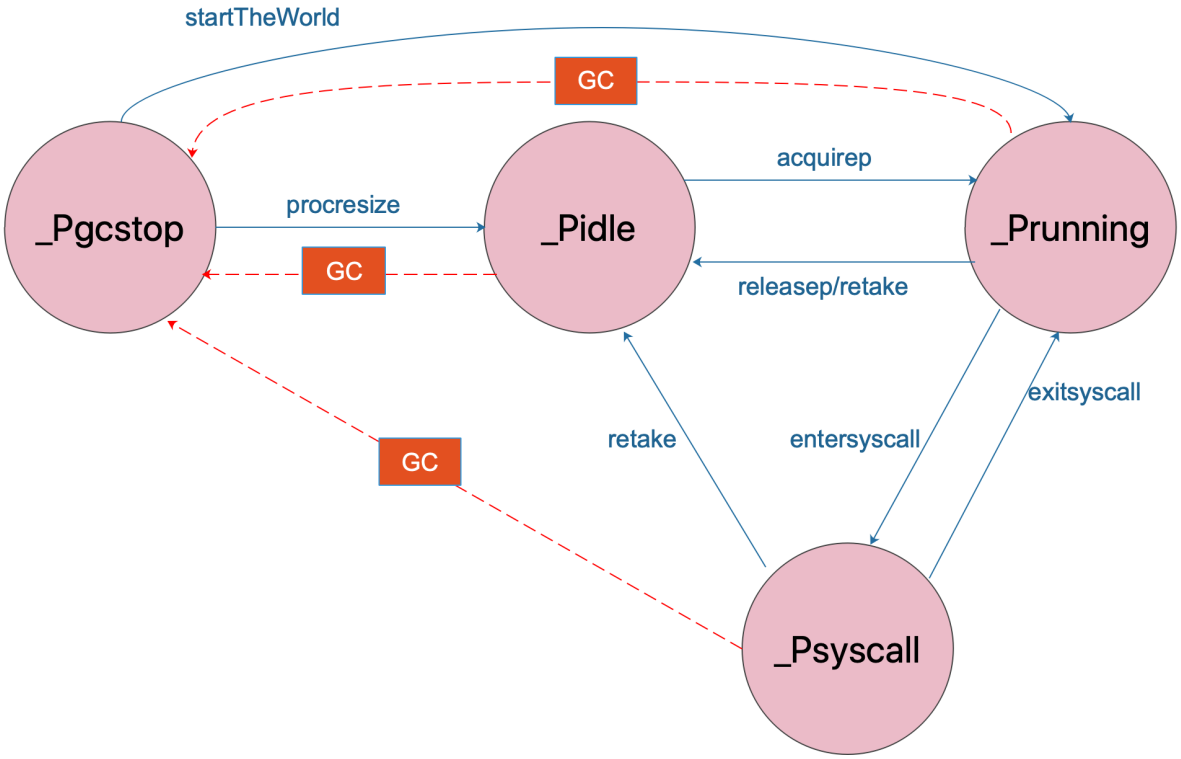
最后我们从宏观上总结一下 GPM，这篇文章尝试从它们的状态流转角度总结。

首先是 G 的状态流转：




说明一下，上图省略了一些垃圾回收的状态。

接着是 P 的状态流转：




> 通常情况下（在程序运行时不调整 P 的个数），P 只会在上图中的四种状态下进行切换。 当程序刚开始运行进行初始化时，所有的 P 都处于 `_Pgcstop` 状态， 随着 P 的初始化（ `runtime.procresize` ），会被置于 `_Pidle` 。

> 当 M 需要运行时，会 `runtime.acquirep` 来使 P 变成 `Prunning` 状态，并通过 `runtime.releasep` 来释放。

> 当 G 执行时需要进入系统调用，P 会被设置为 `_Psyscall` ， 如果这个时候被系统监控抢夺（ `runtime.retake` ），则 P 会被重新修改为 `_Pidle` 。

> 如果在程序运行中发生 `GC` ，则 P 会被设置为 `_Pgcstop` ， 并在 `runtime.startTheWorld` 时重新调整为 `_Prunning` 。

最后，我们来看 M 的状态变化：

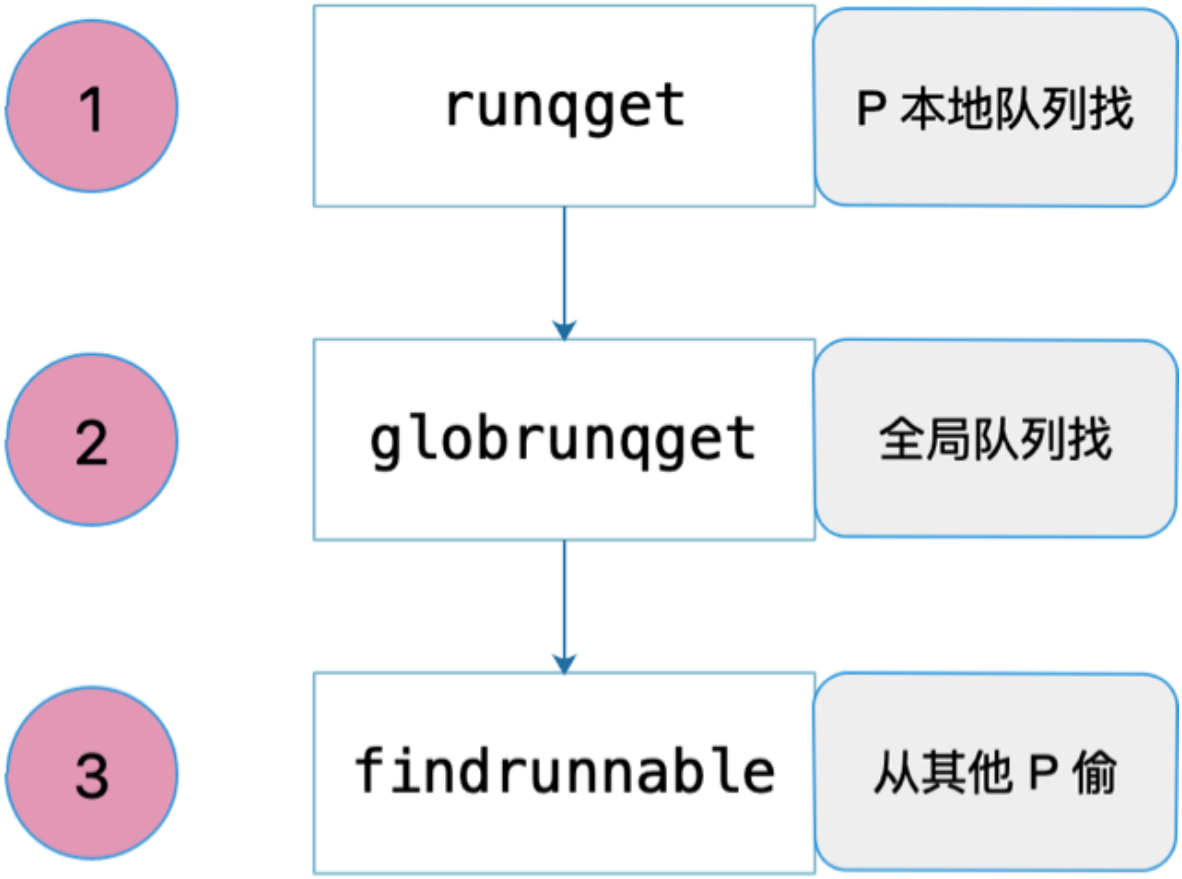

M 只有自旋和非自旋两种状态。自旋的时候，会努力找工作；找不到的时候会进入非自旋状态，之后会休眠，直到有工作需要处理时，被其他工作线程唤醒，又进入自旋状态。

# M 如何找工作

在 schedule 函数中，我们简单提过找一个 runnable goroutine 的过程，这一讲我们来详细分析源码。

工作线程 M 费尽心机也要找到一个可运行的 goroutine，这是它的工作和职责，不达目的，绝不罢休，这种锲而不舍的精神值得每个人学习。

共经历三个过程：先从本地队列找，定期会从全局队列找，最后实在没办法，就去别的 P 偷。如下图所示：




先看第一个：从 P 本地队列找。源码如下：

```
// 从本地可运行队列里找到一个 g
// 如果 inheritTime 为真，gp 应该继承这个时间片，否则，新开启一个时间片
func runqget(_p_ *p) (gp *g, inheritTime bool) {
    // If there's a runnext, it's the next G to run.
    // 如果 runnext 不为空，则 runnext 是下一个待运行的 G
    for {
        next := _p_.runnext
        if next == 0 {
            // 为空，则直接跳出循环
            break
        }
        // 再次比较 next 是否没有变化
        if _p_.runnext.cas(next, 0) {
            // 如果没有变化，则返回 next 所指向的 g。且需要继承时间片
            return next.ptr(), true
        }
    }
```

```
    for {
        // 获取队列头
        h := atomic.Load(&_p_.runqhead) // load-acquire, synchronize with other consumers
        // 获取队列尾
        t := _p_.runqtail
        if t == h {
            // 头和尾相等，说明本地队列为空，找不到 g
            return nil, false
        }
        // 获取队列头的 g
        gp := _p_.runq[h%uint32(len(_p_.runq))].ptr()
        // 原子操作，防止这中间被其他线程因为偷工作而修改
        if atomic.Cas(&_p_.runqhead, h, h+1) { // cas-release, commits consume
            return gp, false
        }
    }
}
```

整个源码结构比较简单，主要是两个 for 循环。

第一个 for 循环尝试返回 P 的 runnext 成员，因为 runnext 具有最高的运行优先级，因此要首先尝试获取 runnext。当发现 runnext 为空时，直接跳出循环，进入第二个。否则，用原子操作获取 runnext，并将其值修改为 0，也就是空。这里用到原子操作的原因是防止在这个过程中，有其他线程过来“偷工作”，导致并发修改 runnext 成员。

第二个 for 循环则是在尝试获取 runnext 成员失败后，尝试从本地队列中返回队列头的 goroutine。同样，先用原子操作获取队列头，使用原子操作的原因同样是防止其他线程“偷工作”时并发对队列头的并发写操作。之后，直接获取队列尾，因为不担心其他线程同时更改，所以直接获取。注意，“偷工作”时只会修改队列头。

比较队列头和队列尾，如果两者相等，说明 P 本地队列没有可运行的 goroutine，直接返回空。否则，算出队列头指向的 goroutine，再用一个 CAS 原子操作来尝试修改队列头，使用原子操作的原因同上。

从本地队列获取可运行 goroutine 的过程比较简单，我们再来看从全局队列获取 goroutine 的过程。在 schedule 函数中调用 `globrunqget` 的代码：

```
// 为了公平，每调用 schedule 函数 61 次就要从全局可运行 goroutine 队列中获取
if _g_.m.p.ptr().schedtick%61 == 0 && sched.runqsize > 0 {
    lock(&sched.lock)
    // 从全局队列最大获取 1 个 gorutine
    gp = globrunqget(_g_.m.p.ptr(), 1)
    unlock(&sched.lock)
}
```

这说明，并不是每次调度都会从全局队列获取可运行的 goroutine。实际情况是调度器每调度 61 次并且全局队列有可运行 goroutine 的情况下才会调用 `globrunqget` 函数尝试从全局获取可运行 goroutine。毕竟，从全局获取需要上锁，这个开销可就大了，能不做就不做。

我们来详细看下 `globrunqget` 的源码：

```
// 尝试从全局队列里获取可运行的 goroutine 队列
func globrunqget(_p_ *p, max int32) *g {
    // 如果队列大小为 0
    if sched.runqsize == 0 {
        return nil
    }

    // 根据 p 的数量平分全局运行队列中的 goroutines
    n := sched.runqsize/gomaxprocs + 1
    if n > sched.runqsize {
```

```
        n = sched.runqsize // 如果 gomaxprocs 为 1
    }

    // 修正"偷"的数量
    if max > 0 && n > max {
        n = max
    }
    // 最多只能"偷"本地工作队列一半的数量
    if n > int32(len(_p_.runq))/2 {
        n = int32(len(_p_.runq)) / 2
    }

    // 更新全局可运行队列长度
    sched.runqsize -= n
    // 如果都要被"偷"走，修改队列尾
    if sched.runqsize == 0 {
        sched.runqtail = 0
    }

    // 获取队列头指向的 goroutine
    gp := sched.runqhead.ptr()
    // 移动队列头
    sched.runqhead = gp.schedlink
    n--
    for ; n > 0; n-- {
        // 获取当前队列头
        gp1 := sched.runqhead.ptr()
        // 移动队列头
        sched.runqhead = gp1.schedlink
        // 尝试将 gp1 放入 P 本地，使全局队列得到更多的执行机会
        runqput(_p_, gp1, false)
    }
    // 返回最开始获取到的队列头所指向的 goroutine
    return gp
}
```

代码比较简单。首先根据全局队列的可运行 goroutine 长度和 P 的总数，来计算一个数值，表示每个 P 可平均分到的 goroutine 数量。

然后根据函数参数中的 max 以及 P 本地队列的长度来决定把多少全局队列中的 goroutine 转移到 P 本地。

最后，for 循环挨个把全局队列中 n-1 个 goroutine 转移到本地，并且返回最开始获取到的队列头所指向的 goroutine，毕竟它最需要得到运行的机会。

把全局队列中的可运行 goroutine 转移到本地队列，给了全局队列中可运行 goroutine 运行的机会，不然全局队列中的 goroutine 一直得不到运行。

最后，我们继续看第三个过程，从其他 P “偷工作”：

```
// 从本地运行队列和全局运行队列都没有找到需要运行的 goroutine，
// 调用 findrunnable 函数从其它工作线程的运行队列中偷取，如果偷不到，则当前工作线程进入睡眠
// 直到获取到 runnable goroutine 之后 findrunnable 函数才会返回。
if gp == nil {
    gp, inheritTime = findrunnable() // blocks until work is available
}
```

这是整个找工作过程最复杂的部分：

```
// 从其他地方找 goroutine 来执行
func findrunnable() (gp *g, inheritTime bool) {
    _g_ := getg()

top:
    _p_ := _g_.m.p.ptr()

    // ......................

    // local runq
    // 从本地队列获取
    if gp, inheritTime := runqget(_p_); gp != nil {
        return gp, inheritTime
    }

    // global runq
    // 从全局队列获取
    if sched.runqsize != 0 {
        lock(&sched.lock)
        gp := globrunqget(_p_, 0)
        unlock(&sched.lock)
        if gp != nil {
            return gp, false
        }
    }

    // ......................

    // Steal work from other P's.

    // 如果其他的 P 都处于空闲状态，那肯定没有其他工作要做
    procs := uint32(gomaxprocs)
    if atomic.Load(&sched.npidle) == procs-1 {
        goto stop
    }

    // 如果有很多工作线程在找工作，那我就停下休息。避免消耗太多 CPU
    if !_g_.m.spinning && 2*atomic.Load(&sched.nmspinning) >= procs-atomic.Load(&sched.npidle) {
        goto stop
    }

    if !_g_.m.spinning {
        // 设置自旋状态为 true
        _g_.m.spinning = true
        // 自旋状态数加 1
        atomic.Xadd(&sched.nmspinning, 1)
    }
    // 从其它 p 的本地运行队列盗取 goroutine
    for i := 0; i < 4; i++ {
        for enum := stealOrder.start(fastrand()); !enum.done(); enum.next() {
            // ......................
            stealRunNextG := i > 2 // first look for ready queues with more than 1 g
            if gp := runqsteal(_p_, allp[enum.position()], stealRunNextG); gp != nil {
                return gp, false
            }
        }
    }

stop:

    // ......................
```

```
    // return P and block
    lock(&sched.lock)
    if sched.gcwaiting != 0 || _p_.runSafePointFn != 0 {
        unlock(&sched.lock)
        goto top
    }
    if sched.runqsize != 0 {
        gp := globrunqget(_p_, 0)
        unlock(&sched.lock)
        return gp, false
    }
    // 当前工作线程解除与 p 之间的绑定，准备去休眠
    if releasep() != _p_ {
        throw("findrunnable: wrong p")
    }
    // 把 p 放入空闲队列
    pidleput(_p_)
    unlock(&sched.lock)

    wasSpinning := _g_.m.spinning
    if _g_.m.spinning {
        // m 即将睡眠，不再处于自旋
        _g_.m.spinning = false
        if int32(atomic.Xadd(&sched.nmspinning, -1)) < 0 {
            throw("findrunnable: negative nmspinning")
        }
    }

    // check all runqueues once again
    // 休眠之前再检查一下所有的 p，看一下是否有工作要做
    for i := 0; i < int(gomaxprocs); i++ {
        _p_ := allp[i]
        if _p_ != nil && !runqempty(_p_) {
            lock(&sched.lock)
            _p_ = pidleget()
            unlock(&sched.lock)
            if _p_ != nil {
                acquirep(_p_)
                if wasSpinning {
                    _g_.m.spinning = true
                    atomic.Xadd(&sched.nmspinning, 1)
                }
                goto top
            }
            break
        }
    }

    // ......................

    // 休眠
    stopm()
    goto top
}
```

这部分也是最能说明 M 找工作的锲而不舍精神：尽力去各个运行队列中寻找 goroutine，如果实在找不到则进入睡眠状态，等待有工作时，被其他 M 唤醒。

先获取当前指向的 g，也就是 g0，然后拿到其绑定的 p，即 `_p_` 。

首先再次尝试从 `_p_` 本地队列获取 goroutine，如果没有获取到，则尝试从全局队列获取。如果还没有获取到就会尝试去“偷”了，这也是没有办法的事。

不过，在偷之前，先看大的局势。如果其他所有的 P 都处于空闲状态，就说明其他 P 肯定没有工作可做，就没必要再去偷了，毕竟“地主家也没有余粮了”，跳到 stop 部分。接着再看下当前正在“偷工作”的线程数量“太多了”，就没必要扎堆了，这么多人，竞争肯定大，工作肯定不好找，也不好偷。

在真正的“偷”工作之前，把自己的自旋状态设置为 true，全局自旋数量加 1。

终于到了“偷工作”的部分了，好紧张！整个过程由两层 for 循环组成，外层控制尝试偷的次数，内层控制“偷”的顺序，并真正的去“偷”。实际上，内层会遍历所有的 P，因此，整体看来，会尝试 4 次扫遍所有的 P，并去“偷工作”，是不是非常有毅力！

第二层的循环并不是每次都按一个固定的顺序去遍历所有的 P，这样不太科学，而是使用了一些方法，“随机”地遍历。具体是使用了下面这个变量：

```
var stealOrder randomOrder

type randomOrder struct {
    count    uint32
    coprimes []uint32
}
```

初始化的时候会给 count 赋一个值，例如 8，根据 count 计算出 coprimes，里面的元素是小于 count 的值，且和 8 互质，算出来是：[1, 3, 5, 7]。

第二层循环，开始随机给一个值，例如 2，则第一个访问的 P 就是 P2；从 coprimes 里取出索引为 2 的值为 5，那么，第二个访问的 P 索引就是 2+5=7；依此类推，第三个就是 7+5=12，和 count 做一个取余操作，即 12%8=4.....

在最后一次遍历所有的 P 的过程中，连人家的 runnext 也要尝试偷过来，毕竟前三次的失败经验证明，工作太不好“偷”了，民不聊生啊，只能做得绝一点了，`stealRunNextG` 控制是否要打 runnext 的主意：

```
stealRunNextG := i > 2
```

确定好准备偷的对象 `allp[enum.position()]` 之后，调用 `runqsteal(_p_, allp[enum.position()], stealRunNextG)` 函数执行。

```
// 从 p2 偷走一半的工作放到 _p_ 的本地
func runqsteal(_p_, p2 *p, stealRunNextG bool) *g {
    // 队尾
    t := _p_.runqtail
    // 从 p2 偷取工作，放到 _p_.runq 的队尾
    n := runqgrab(p2, &_p_.runq, t, stealRunNextG)
    if n == 0 {
        return nil
    }
    n--
    // 找到最后一个 g，准备返回
    gp := _p_.runq[(t+n)%uint32(len(_p_.runq))].ptr()
    if n == 0 {
        // 说明只偷了一个 g
        return gp
    }
    // 队列头
    h := atomic.Load(&_p_.runqhead) // load-acquire, synchronize with consumers
    // 判断是否偷太多了
    if t-h+n >= uint32(len(_p_.runq)) {
        throw("runqsteal: runq overflow")
    }
```

```
    // 更新队尾，将偷来的工作加入队列
    atomic.Store(&_p_.runqtail, t+n) // store-release, makes the item available for consumption
    return gp
}
```

调用 `runqgrab` 从 p2 偷走它一半的工作放到 `_p_` 本地：

```
n := runqgrab(p2, &_p_.runq, t, stealRunNextG)
```

`runqgrab` 函数将从 p2 偷来的工作放到以 t 为地址的数组里，数组就是 `_p_.runq` 。我们知道，`t` 是 `_p_.runq` 的队尾，因此这行代码表达的真正意思是将从 p2 偷来的工作，神不知，鬼不觉地放到 `_p_.runq` 的队尾，之后，再悄悄改一下 `p.runqtail` 就把这些偷来的工作据为己有了。

接着往下看，返回的 n 表示偷到的工作数量。先将 n 自减 1，目的是把第 n 个工作（也就是 g）直接返回，如果这时候 n 变成 0 了，说明就只偷到了一个 g，那就直接返回。否则，将队尾往后移动 n，把偷来的工作合法化，简直完美！

我们接着往下看 `runqgrab` 函数的实现：

```
// 从 _p_ 批量获取可运行 goroutine，放到 batch 数组里
// batch 是一个环，起始于 batchHead
// 返回偷的数量，返回的 goroutine 可被任何 P 执行
func runqgrab(_p_ *p, batch *[256]guintptr, batchHead uint32, stealRunNextG bool) uint32 {
    for {
        // 队列头
        h := atomic.Load(&_p_.runqhead) // load-acquire, synchronize with other consumers
        // 队列尾
        t := atomic.Load(&_p_.runqtail) // load-acquire, synchronize with the producer
        // g 的数量
        n := t - h
        // 取一半
        n = n - n/2
        if n == 0 {
            if stealRunNextG {
                // 连 runnext 都要偷，没有人性
                // Try to steal from _p_.runnext.
                if next := _p_.runnext; next != 0 {
                    // 这里是为了防止 _p_ 执行当前 g，并且马上就要阻塞，所以会马上执行 runnext，
                    // 这个时候偷就没必要了，因为让 g 在 P 之间"游走"不太划算，
                    // 就不偷了，给他们一个机会。
                    // channel 一次同步的的接收发送需要 50ns 左右，因此 3us 差不多给了他们 50 次机会了，做得
还是不错的
                    if GOOS != "windows" {
                        usleep(3)
                    } else {
                        osyield()
                    }
                    if !_p_.runnext.cas(next, 0) {
                        continue
                    }
                    // 真的偷走了 next
                    batch[batchHead%uint32(len(batch))] = next
                    // 返回偷的数量，只有 1 个
                    return 1
                }
            }
            // 没偷到
            return 0
        }
        // 如果 n 这时变得太大了，重新来一遍了，不能偷的太多，做得太过分了
```

```
        if n > uint32(len(_p_.runq)/2) { // read inconsistent h and t
            continue
        }
        // 将 g 放置到 bacth 中
        for i := uint32(0); i < n; i++ {
            g := _p_.runq[(h+i)%uint32(len(_p_.runq))]
            batch[(batchHead+i)%uint32(len(batch))] = g
        }
        // 工作被偷走了，更新一下队列头指针
        if atomic.Cas(&_p_.runqhead, h, h+n) { // cas-release, commits consume
            return n
        }
    }
}
```

外层直接就是一个无限循环，先用原子操作取出 p 的队列头和队列尾，算出一半的 g 的数量，如果 n == 0，说明地主家也没有余粮，这时看 `stealRunNextG` 的值。如果为假，说明不偷 runnext，那就直接返回 0，啥也没偷到；如果为真，则要尝试偷一下 runnext。

先判断 runnext 不为空，那就真的准备偷了。不过在这之前，要先休眠 3 us。这是为了防止 p 正在执行当前的 g，马上就要阻塞（可能是向一个非缓冲的 channel 发送数据，没有接收者），之后会马上执行 runnext。这个时候偷就没必要了，因为 runnext 马上就要执行了，偷走它还不是要去执行，那何必要偷呢？大家的愿望就是提高效率，这样让 g 在 P 之间"游走"不太划算，索性先不偷了，给他们一个机会。 `channel` 一次同步的的接收或发送需要 50ns 左右，因此休眠 3us 差不多给了他们 50 次机会了，做得还是挺厚道的。

继续看，再次判断 n 是否小于等于 p.runq 长度的一半，因为这个时候很可能 p 也被其他线程偷了，它的 p.runq 就没那么多工作了，这个时候就不能偷这么多了，要重新再走一次循环。

最后一个 for 循环，将 p.runq 里的 g 放到 batch 数组里。使用原子操作更新 p 的队列头指针，往后移动 n 个位置，这些都是被偷走的，伤心！

回到 `findrunnable` 函数，经过上述三个层面的"偷窃"过程，我们仍然没有找到工作，真惨！于是就走到了 `stop` 这个代码块。

先上锁，因为要将 P 放到全局空闲 P 链表里去。在这之前还不死心，再瞧一下全局队列里是否有工作，如果有，再去尝试偷全局。

如果没有，就先解除当前工作线程和当前 P 的绑定关系：

```
// 解除 p 与 m 的关联
func releasep() *p {
    _g_ := getg()

    // ......................

    _p_ := _g_.m.p.ptr()

    // ......................

    // 清空一些字段
    _g_.m.p = 0
    _g_.m.mcache = nil
    _p_.m = 0
    _p_.status = _Pidle
    return _p_
}
```

主要的工作就是将 p 的 m 字段清空，并将 p 的状态修改为 `_Pidle` 。

这之后，将其放入全局空闲 P 列表：

```
// 将 p 放到 _Pidle 列表里
//go:nowritebarrierrec
func pidleput(_p_ *p) {
    if !runqempty(_p_) {
        throw("pidleput: P has non-empty run queue")
    }
    _p_.link = sched.pidle
    sched.pidle.set(_p_)
    // 增加全局空闲 P 的数量
    atomic.Xadd(&sched.npidle, 1) // TODO: fast atomic
}
```

构造链表的过程其实比较简单，先将 p.link 指向原来的 sched.pidle 所指向的 p，也就是原空闲链表的最后一个 P，最后，再更新 sched.pidle，使其指向当前 p，这样，新的链表就构造完成。

接下来就要真正地准备休眠了，但是仍然不死心！还要再查看一次所有的 P 是否有工作，如果发现任何一个 P 有工作的话（判断 P 的本地队列不空），就先从全局空闲 P 链表里先拿到一个 P：

```
// 试图从 _Pidle 列表里获取 p
//go:nowritebarrierrec
func pidleget() *p {
    _p_ := sched.pidle.ptr()
    if _p_ != nil {
        sched.pidle = _p_.link
        atomic.Xadd(&sched.npidle, -1) // TODO: fast atomic
    }
    return _p_
}
```

比较简单，获取链表最后一个，再更新 sched.pidle，使其指向前一个 P。调用 `acquirep(_p_)` 绑定获取到的 p 和 m，主要的动作就是设置 p 的 m 字段，更改 p 的工作状态为 `_Prunning` ，并且设置 m 的 p 字段。做完这些之后，再次进入 top 代码段，再走一遍之前找工作的过程。

```
// 休眠，停止执行工作，直到有新的工作需要做为止
func stopm() {
    // 当前 goroutine，g0
    _g_ := getg()

    // ....................
retry:
    lock(&sched.lock)
    // 将 m 放到全局空闲链表里去
    mput(_g_.m)
    unlock(&sched.lock)
    // 进入睡眠状态
    notesleep(&_g_.m.park)
    // 这里被其他工作线程唤醒
    noteclear(&_g_.m.park)

    // ....................

    acquirep(_g_.m.nextp.ptr())
    _g_.m.nextp = 0
}
```

先将 m 放入全局空闲链表里，注意涉及到全局变量的修改，要上锁。接着，调用 `notesleep(&_g_.m.park)` 使得当前工作线程进入休眠状态。其他工作线程在检测到“当前有很多工作要做”，会调用 `noteclear(&_g_.m.park)` 将其唤醒。注意，这两个函数传入的参数都是一样的： `&_g_.m.park` ，它的类型是：

```
type note struct {
    key uintptr
}
```

很简单，只有一个 key 字段。

> note 的底层实现机制跟操作系统相关，不同系统使用不同的机制，比如 linux 下使用的 futex 系统调用，而 mac 下则是使用的 pthread_cond_t 条件变量，note 对这些底层机制做了一个抽象和封装。

> 这种封装给扩展性带来了很大的好处，比如当睡眠和唤醒功能需要支持新平台时，只需要在 note 层增加对特定平台的支持即可，不需要修改上层的任何代码。

上面这一段来自阿波张的系列教程。我们接着来看下 notesleep 的实现：

```
// runtime/lock_futex.go
func notesleep(n *note) {
    // g0
    gp := getg()
    if gp != gp.m.g0 {
        throw("notesleep not on g0")
    }
    // -1 表示无限期休眠
    ns := int64(-1)

    // ......................

    // 这里之所以需要用一个循环，是因为 futexsleep 有可能意外从睡眠中返回，
    // 所以 futexsleep 函数返回后还需要检查 note.key 是否还是 0，
    // 如果是 0 则表示并不是其它工作线程唤醒了我们，
    // 只是 futexsleep 意外返回了，需要再次调用 futexsleep 进入睡眠
    for atomic.Load(key32(&n.key)) == 0 {
        // 表示 m 被阻塞
        gp.m.blocked = true
        futexsleep(key32(&n.key), 0, ns)

        // ......................

        // 被唤醒，更新标志
        gp.m.blocked = false
    }
}
```

继续往下追：

```
// runtime/os_linux.go
func futexsleep(addr *uint32, val uint32, ns int64) {
    var ts timespec

    if ns < 0 {
        futex(unsafe.Pointer(addr), _FUTEX_WAIT, val, nil, nil, 0)
        return
    }
```

```
    // ......................
}
```

当 *addr 和 val 相等的时候，休眠。 `futex` 由汇编语言实现：

```
TEXT runtime·futex(SB),NOSPLIT,$0
    // 为系统调用准备参数
    MOVQ    addr+0(FP), DI
    MOVL    op+8(FP), SI
    MOVL    val+12(FP), DX
    MOVQ    ts+16(FP), R10
    MOVQ    addr2+24(FP), R8
    MOVL    val3+32(FP), R9
    // 系统调用编号
    MOVL    $202, AX
    // 执行 futex 系统调用进入休眠，被唤醒后接着执行下一条 MOVL 指令
    SYSCALL
    // 保存系统调用的返回值
    MOVL    AX, ret+40(FP)
    RET
```

这样，找不到工作的 m 就休眠了。当其他线程发现有工作要做时，就会先找到空闲的 m，再通过 m.park 字段来唤醒本线程。唤醒之后，回到 `findrunnable` 函数，继续寻找 goroutine，找到后返回 schedule 函数，然后就会去运行找到的 goroutine。

这就是 m 找工作的整个过程，历尽千辛万苦，终于修成正果。

# 参考资料

【阿波张 Goroutine 调度策略】https://mp.weixin.qq.com/s/2objs5JrlnKnwFbF4a2z2g

# mian gorutine 如何创建

上一讲我们讲完了 Go scheduler 的初始化，现在调度器一切就绪，就差被调度的实体了。本文就来讲述 main goroutine 是如何诞生，并且被调度的。

继续看代码，前面我们完成了 `schedinit` 函数，这是 runtime·rt0_go 函数里的一步，接着往后看：

```
// 创建一个新的 goroutine 来启动程序
MOVQ    $runtime·mainPC(SB), AX         // entry
// newproc 的第二个参数入栈，也就是新的 goroutine 需要执行的函数
// AX = &funcval{runtime·main},
PUSHQ   AX
// newproc 的第一个参数入栈，该参数表示 runtime.main 函数需要的参数大小，
// 因为 runtime.main 没有参数，所以这里是 0
PUSHQ   $0              // arg size
// 创建 main goroutine
CALL    runtime·newproc(SB)
POPQ    AX
POPQ    AX

// start this M
// 主线程进入调度循环，运行刚刚创建的 goroutine
CALL    runtime·mstart(SB)

// 永远不会返回，万一返回了，crash 掉
MOVL    $0xf1, 0xf1  // crash
RET
```

代码前面几行是在为调用 newproc 函数构“造栈”，执行完 `runtime·newproc(SB)` 后，就会以一个新的 goroutine 来执行 mainPC 也就是 `runtime.main()` 函数。`runtime.main()` 函数最终会执行到我们写的 main 函数，舞台交给我们。

重点来看 `newproc` 函数：

```
// src/runtime/proc.go
// 创建一个新的 g，运行 fn 函数，需要 siz byte 的参数
// 将其放至 G 队列等待运行
// 编译器会将 go 关键字的语句转化成此函数

//go:nosplit
func newproc(siz int32, fn *funcval)
```

从这里开始要进入 hard 模式了，打起精神！当我们随手一句：

```
go func() {
    // 要做的事
}()
```

就启动了一个 goroutine 的时候，一定要知道，在 Go 编译器的作用下，这条语句最终会转化成 newproc 函数。

因此，`newproc` 函数需要两个参数：一个是新创建的 goroutine 需要执行的任务，也就是 fn，它代表一个函数 func；还有一个是 fn 的参数大小。

再回过头看，构造 newproc 函数调用栈的时候，第一个参数是 0，因为 runtime.main 函数没有参数：

```
// src/runtime/proc.go

func main()
```

第二个参数则是 runtime.main 函数的地址。

可能会感到奇怪，为什么要给 `newproc` 传一个表示 fn 的参数大小的参数呢？

我们知道，goroutine 和线程一样，都有自己的栈，不同的是 goroutine 的初始栈比较小，只有 2K，而且是可伸缩的，这也是创建 goroutine 的代价比创建线程代价小的原因。

换句话说，每个 goroutine 都有自己的栈空间，newproc 函数会新创建一个新的 goroutine 来执行 fn 函数，在新 goroutine 上执行指令，就要用新 goroutine 的栈。而执行函数需要参数，这个参数又是在老的 goroutine 上，所以需要将其拷贝到新 goroutine 的栈上。拷贝的起始位置就是栈顶，这好办，那拷贝多少数据呢？由 siz 来确定。

继续看代码，newproc 函数的第二个参数：

```
type funcval struct {
    fn uintptr
    // variable-size, fn-specific data here
}
```

它是一个变长结构，第一个字段是一个指针 fn，内存中，紧挨着 fn 的是函数的参数。

参考资料【欧神 关键字 go】有一个例子：

```
package main

func hello(msg string) {
    println(msg)
}

func main() {
    go hello("hello world")
}
```

栈布局是这样的：


栈顶是 siz，再往上是函数的地址，再往上就是传给 hello 函数的参数，string 在这里是一个地址。因此前面代码里先 push 参数的地址，再 push 参数大小。

```
// src/runtime/proc.go

//go:nosplit
func newproc(siz int32, fn *funcval) {
    // 获取第一个参数地址
    argp := add(unsafe.Pointer(&fn), sys.PtrSize)
    // 获取调用者的指令地址，也就是调用 newproc 时由 call 指令压栈的函数返回地址
    pc := getcallerpc(unsafe.Pointer(&siz))
    // systemstack 的作用是切换到 g0 栈执行作为参数的函数
    // 用 g0 系统栈创建 goroutine 对象
    // 传递的参数包括 fn 函数入口地址，argp 参数起始地址，siz 参数长度，调用方 pc（goroutine）

    systemstack(func() {
        newproc1(fn, (*uint8)(argp), siz, 0, pc)
    })
}
```

因此，argp 跳过 fn，向上跳一个指针的长度，拿到 fn 参数的地址。

接着通过 getcallerpc 获取调用者的指令地址，也就是调用 newproc 时由 call 指令压栈的函数返回地址，也就是 `runtime·rt0_go` 函数里 `CALL runtime·newproc(SB)` 指令后面的 `POPQ AX` 这条指令的地址。

最后，调用 systemstack 函数在 g0 栈执行 fn 函数。由于本文讲述的是初始化过程中，由 `runtime·rt0_go` 函数调用，本身是在 g0 栈执行，因此会直接执行 fn 函数。而如果是我们在程序中写的 `go xxx` 代码，在执行时，就会先切换到 g0 栈执行，然后再切回来。

一鼓作气，继续看 `newproc1` 函数，为了连贯性，我先将整个函数的代码贴出来，并且加上了注释。当然，这篇文章不会涉及到所有的代码，只会讲部分内容。放在这里，方便阅读后面的文章时对照：

```
// 创建一个新的 g 来跑 fn
func newproc1(fn *funcval, argp *uint8, narg int32, nret int32, callerpc uintptr) *g {
    // 当前 goroutine 的指针
```

```
    // 因为已经切换到 g0 栈，所以无论什么场景都是 _g_ = g0
    // g0 是指当前工作线程的 g0
    _g_ := getg()

    if fn == nil {
        _g_.m.throwing = -1 // do not dump full stacks
        throw("go of nil func value")
    }
    _g_.m.locks++ // disable preemption because it can be holding p in a local var

    // 参数加返回值所需要的空间（经过内存对齐）
    siz := narg + nret
    siz = (siz + 7) &^ 7

    // ..........................

    // 当前工作线程所绑定的 p
    // 初始化时 _p_ = g0.m.p，也就是 _p_ = allp[0]
    _p_ := _g_.m.p.ptr()
    // 从 p 的本地缓冲里获取一个没有使用的 g，初始化时为空，返回 nil
    newg := gfget(_p_)
    if newg == nil {
        // new 一个 g 结构体对象，然后从堆上为其分配栈，并设置 g 的 stack 成员和两个 stackgard 成员
        newg = malg(_StackMin)
        // 初始化 g 的状态为 _Gdead
        casgstatus(newg, _Gidle, _Gdead)
        // 放入全局变量 allgs 切片中
        allgadd(newg) // publishes with a g->status of Gdead so GC scanner doesn't look at uninitialized stac
k.
    }
    if newg.stack.hi == 0 {
        throw("newproc1: newg missing stack")
    }

    if readgstatus(newg) != _Gdead {
        throw("newproc1: new g is not Gdead")
    }

    // 计算运行空间大小，对齐
    totalSize := 4*sys.RegSize + uintptr(siz) + sys.MinFrameSize // extra space in case of reads slightly bey
ond frame
    totalSize += -totalSize & (sys.SpAlign - 1)                  // align to spAlign
    // 确定 sp 位置
    sp := newg.stack.hi - totalSize
    // 确定参数入栈位置
    spArg := sp

    // ..........................

    if narg > 0 {
        // 将参数从执行 newproc 函数的栈拷贝到新 g 的栈
        memmove(unsafe.Pointer(spArg), unsafe.Pointer(argp), uintptr(narg))

        // ..........................
    }

    // 把 newg.sched 结构体成员的所有成员设置为 0
    memclrNoHeapPointers(unsafe.Pointer(&newg.sched), unsafe.Sizeof(newg.sched))
    // 设置 newg 的 sched 成员，调度器需要依靠这些字段才能把 goroutine 调度到 CPU 上运行
    newg.sched.sp = sp
    newg.stktopsp = sp
    // newg.sched.pc 表示当 newg 被调度起来运行时从这个地址开始执行指令
```

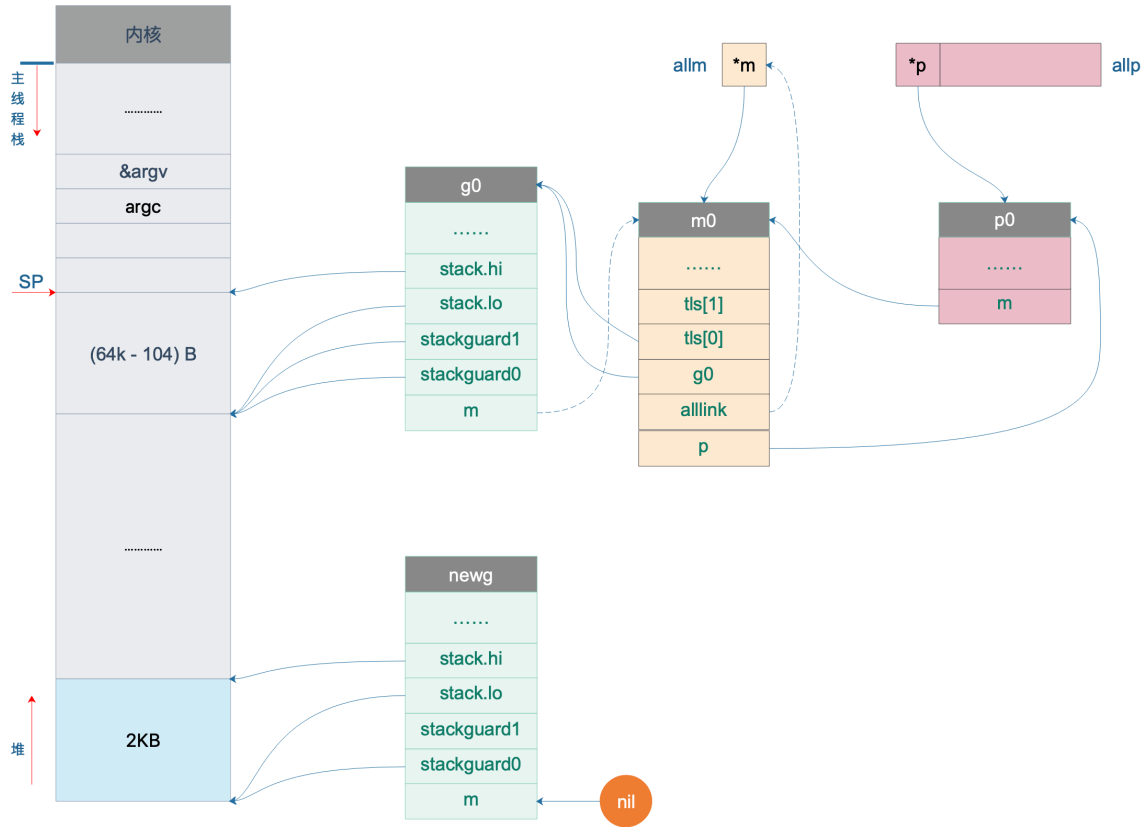
```
    newg.sched.pc = funcPC(goexit) + sys.PCQuantum // +PCQuantum so that previous instruction is in same func
tion
    newg.sched.g = guintptr(unsafe.Pointer(newg))
    gostartcallfn(&newg.sched, fn)
    newg.gopc = callerpc
    // 设置 newg 的 startpc 为 fn.fn，该成员主要用于函数调用栈的 traceback 和栈收缩
    // newg 真正从哪里开始执行并不依赖于这个成员，而是 sched.pc
    newg.startpc = fn.fn
    if _g_.m.curg != nil {
        newg.labels = _g_.m.curg.labels
    }
    if isSystemGoroutine(newg) {
        atomic.Xadd(&sched.ngsys, +1)
    }
    newg.gcscanvalid = false
    // 设置 g 的状态为 _Grunnable，可以运行了
    casgstatus(newg, _Gdead, _Grunnable)

    if _p_.goidcache == _p_.goidcacheend {
        _p_.goidcache = atomic.Xadd64(&sched.goidgen, _GoidCacheBatch)
        _p_.goidcache -= _GoidCacheBatch - 1
        _p_.goidcacheend = _p_.goidcache + _GoidCacheBatch
    }
    // 设置 goid
    newg.goid = int64(_p_.goidcache)
    _p_.goidcache++

    // ......................

    // 将 G 放入 _p_ 的本地待运行队列
    runqput(_p_, newg, true)

    if atomic.Load(&sched.npidle) != 0 && atomic.Load(&sched.nmspinning) == 0 && mainStarted {
        wakep()
    }
    _g_.m.locks--
    if _g_.m.locks == 0 && _g_.preempt {
        _g_.stackguard0 = stackPreempt
    }
    return newg
}
```

当前代码在 g0 栈上执行，因此执行完 `_g_ := getg()` 之后，无论是在什么情况下都可以得到 `_g_ = g0` 。之后通过 g0 找到其绑定的 P，也就是 p0。

接着，尝试从 p0 上找一个空闲的 G：

```
// 从 p 的本地缓冲里获取一个没有使用的 g，初始化时为空，返回 nil
newg := gfget(_p_)
```

如果拿不到，则会在堆上创建一个新的 G，为其分配 2KB 大小的栈，并设置好新 goroutine 的 stack 成员，设置其状态为 _Gdead，并将其添加到全局变量 allgs 中。创建完成之后，我们就在堆上有了一个 2K 大小的栈。于是，我们的图再次丰富：




这样，main goroutine 就诞生了。

# 参考资料

【欧神 关键字 go】https://github.com/changkun/go-under-the-hood/blob/master/book/zh-cn/part3compile/ch11keyword/go.md

【欧神 Go scheduler】https://github.com/changkun/go-under-the-hood/blob/master/book/zh-cn/part2runtime/ch06sched/init.md

# schedule 循环如何启动

上一讲新创建了一个 goroutine，设置好了 sched 成员的 sp 和 pc 字段，并且将其添加到了 p0 的本地可运行队列，坐等调度器的调度。

我们继续看代码。搞了半天，我们其实还在 `runtime·rt0_go` 函数里，执行完 `runtime·newproc(SB)` 后，两条 POP 指令将之前为调用它构建的参数弹出栈。好消息是，最后就只剩下一个函数了：

```
// start this M
// 主线程进入调度循环，运行刚刚创建的 goroutine
CALL    runtime·mstart(SB)
```

这到达了本系列的核心区，前面铺垫了半天，调度器终于要开始运转了。

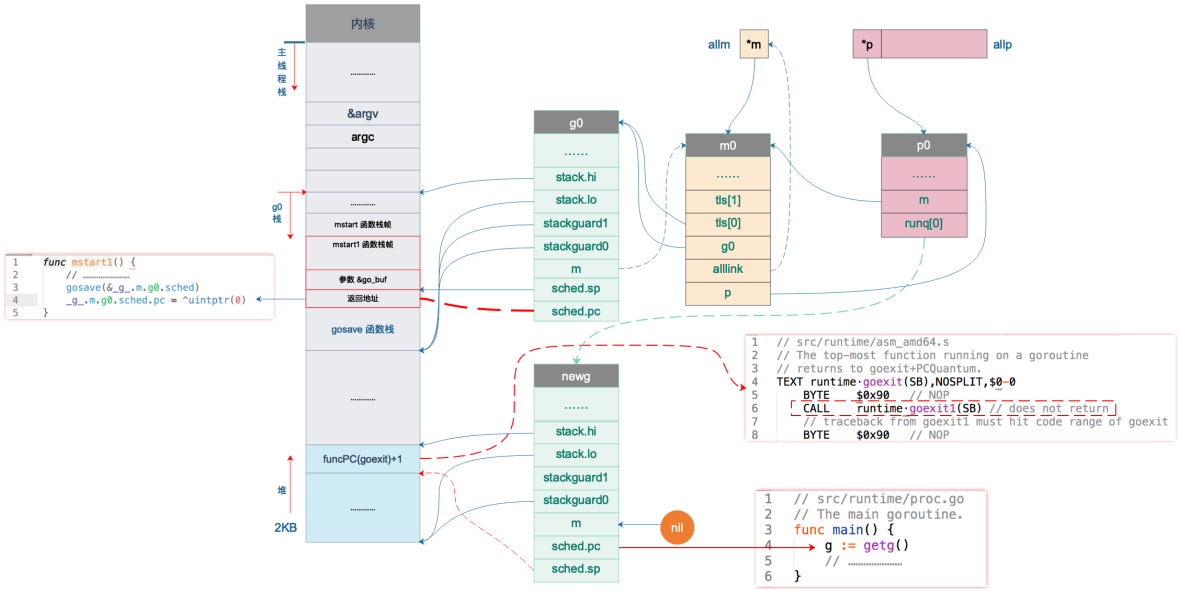
`mstart` 函数设置了 stackguard0 和 stackguard1 字段后，就直接调用 mstart1() 函数：

```
func mstart1() {
    // 启动过程时 _g_ = m0.g0
    _g_ := getg()

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

    // Record top of stack for use by mcall.
    // Once we call schedule we're never coming back,
    // so other calls can reuse this stack space.
    //
    // 一旦调用 schedule() 函数，永不返回
    // 所以栈帧可以被复用
    gosave(&_g_.m.g0.sched)
    _g_.m.g0.sched.pc = ^uintptr(0) // make sure it is never used
    asminit()
    minit()

    // ......................

    // 执行启动函数。初始化过程中，fn == nil
    if fn := _g_.m.mstartfn; fn != nil {
        fn()
    }

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

    // 进入调度循环。永不返回
    schedule()
}
```

调用 `gosave` 函数来保存调度信息到 `g0.sched` 结构体，来看源码：

```
// void gosave(Gobuf*)
// save state in Gobuf; setjmp
TEXT runtime·gosave(SB), NOSPLIT, $0-8
    // 将 gobuf 赋值给 AX
    MOVQ    buf+0(FP), AX       // gobuf
    // 取参数地址，也就是 caller 的 SP
    LEAQ    buf+0(FP), BX       // caller's SP
    // 保存 caller's SP，再次运行时的栈顶
    MOVQ    BX, gobuf_sp(AX)
    MOVQ    0(SP), BX       // caller's PC
    // 保存 caller's PC，再次运行时的指令地址
    MOVQ    BX, gobuf_pc(AX)
    MOVQ    $0, gobuf_ret(AX)
    MOVQ    BP, gobuf_bp(AX)
    // Assert ctxt is zero. See func save.
    MOVQ    gobuf_ctxt(AX), BX
    TESTQ   BX, BX
    JZ  2(PC)
    CALL    runtime·badctxt(SB)
    // 获取 tls
    get_tls(CX)
    // 将 g 的地址存入 BX
    MOVQ    g(CX), BX
    // 保存 g 的地址
    MOVQ    BX, gobuf_g(AX)
    RET
```

主要是设置了 g0.sched.sp 和 g0.sched.pc，前者指向 mstart1 函数栈上参数的位置，后者则指向 gosave 函数返回后的下一条指令。如下图：




图中 sched.pc 并不直接指向返回地址，所以图中的虚线并没有箭头。

接下来，进入 schedule 函数，永不返回。

```
// 执行一轮调度器的工作：找到一个 runnable 的 goroutine，并且执行它
// 永不返回
func schedule() {
    // _g_ = 每个工作线程 m 对应的 g0，初始化时是 m0 的 g0
    _g_ := getg()
```

```
    // ......................

top:
    // ......................

    var gp *g
    var inheritTime bool

    // ......................

    if gp == nil {
        // Check the global runnable queue once in a while to ensure fairness.
        // Otherwise two goroutines can completely occupy the local runqueue
        // by constantly respawning each other.
        // 为了公平，每调用 schedule 函数 61 次就要从全局可运行 goroutine 队列中获取
        if _g_.m.p.ptr().schedtick%61 == 0 && sched.runqsize > 0 {
            lock(&sched.lock)
            // 从全局队列最大获取 1 个 gorutine
            gp = globrunqget(_g_.m.p.ptr(), 1)
            unlock(&sched.lock)
        }
    }

    // 从 P 本地获取 G 任务
    if gp == nil {
        gp, inheritTime = runqget(_g_.m.p.ptr())
        if gp != nil && _g_.m.spinning {
            throw("schedule: spinning with local work")
        }
    }

    if gp == nil {
        // 从本地运行队列和全局运行队列都没有找到需要运行的 goroutine，
        // 调用 findrunnable 函数从其它工作线程的运行队列中偷取，如果偷不到，则当前工作线程进入睡眠
        // 直到获取到 runnable goroutine 之后 findrunnable 函数才会返回。
        gp, inheritTime = findrunnable() // blocks until work is available
    }

    // This thread is going to run a goroutine and is not spinning anymore,
    // so if it was marked as spinning we need to reset it now and potentially
    // start a new spinning M.
    if _g_.m.spinning {
        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 goroutine 任务函数
    // 当前运行的是 runtime 的代码，函数调用栈使用的是 g0 的栈空间
    // 调用 execute 切换到 gp 的代码和栈空间去运行
    execute(gp, inheritTime)
}
```

调用 `runqget` ，从 P 本地可运行队列先选出一个可运行的 goroutine；为了公平，调度器每调度 61 次的时候，都会尝试从全局队列里取出待运行的 goroutine 来运行，调用 `globrunqget` ；如果还没找到，就要去其他 P 里面去偷一些

goroutine 来执行，调用 `findrunnable` 函数。

经过千辛万苦，终于找到了可以运行的 goroutine，调用 `execute(gp, inheritTime)` 切换到选出的 goroutine 栈执行，调度器的调度次数会在这里更新，源码如下：

```
// 调度 gp 在当前 M 上运行
// 如果 inheritTime 为真，gp 执行当前的时间片
// 否则，开启一个新的时间片
//
//go:yeswritebarrierrec
func execute(gp *g, inheritTime bool) {
    // g0
    _g_ := getg()

    // 将 gp 的状态改为 running
    casgstatus(gp, _Grunnable, _Grunning)
    gp.waitsince = 0
    gp.preempt = false
    gp.stackguard0 = gp.stack.lo + _StackGuard
    if !inheritTime {
        // 调度器调度次数增加 1
        _g_.m.p.ptr().schedtick++
    }

    // 将 gp 和 m 关联起来
    _g_.m.curg = gp
    gp.m = _g_.m

    // ..............................

    // gogo 完成从 g0 到 gp 真正的切换
    // CPU 执行权的转让以及栈的切换
    // 执行流的切换从本质上来说就是 CPU 寄存器以及函数调用栈的切换，
    // 然而不管是 go 还是 c 这种高级语言都无法精确控制 CPU 寄存器的修改，
    // 因而高级语言在这里也就无能为力了，只能依靠汇编指令来达成目的
    gogo(&gp.sched)
}
```

将 gp 的状态改为 `_Grunning`，将 m 和 gp 相互关联起来。最后，调用 `gogo` 完成从 g0 到 gp 的切换，CPU 的执行权将从 g0 转让到 gp。`gogo` 函数用汇编语言写成，原因如下：

> `gogo` 函数也是通过汇编语言编写的，这里之所以需要使用汇编，是因为 goroutine 的调度涉及不同执行流之间的切换。

> 前面我们在讨论操作系统切换线程时已经看到过，执行流的切换从本质上来说就是 CPU 寄存器以及函数调用栈的切换，然而不管是 go 还是 c 这种高级语言都无法精确控制 CPU 寄存器，因而高级语言在这里也就无能为力了，只能依靠汇编指令来达成目的。

继续看 `gogo` 函数的实现，传入 `&gp.sched` 参数，源码如下：

```
TEXT runtime·gogo(SB), NOSPLIT, $16-8
    // 0(FP) 表示第一个参数，即 buf = &gp.sched
    MOVQ    buf+0(FP), BX       // gobuf

    // ......................

    MOVQ    buf+0(FP), BX
```

```
nilctxt:
    // DX = gp.sched.g
    MOVQ    gobuf_g(BX), DX
    MOVQ    0(DX), CX       // make sure g != nil
    get_tls(CX)
    // 将 g 放入到 tls[0]
    // 把要运行的 g 的指针放入线程本地存储，这样后面的代码就可以通过线程本地存储
    // 获取到当前正在执行的 goroutine 的 g 结构体对象，从而找到与之关联的 m 和 p
    // 运行这条指令之前，线程本地存储存放的是 g0 的地址
    MOVQ    DX, g(CX)
    // 把 CPU 的 SP 寄存器设置为 sched.sp，完成了栈的切换
    MOVQ    gobuf_sp(BX), SP    // restore SP
    // 恢复调度上下文到CPU相关寄存器
    MOVQ    gobuf_ret(BX), AX
    MOVQ    gobuf_ctxt(BX), DX
    MOVQ    gobuf_bp(BX), BP
    // 清空 sched 的值，因为我们已把相关值放入 CPU 对应的寄存器了，不再需要，这样做可以少 GC 的工作量
    MOVQ    $0, gobuf_sp(BX)    // clear to help garbage collector
    MOVQ    $0, gobuf_ret(BX)
    MOVQ    $0, gobuf_ctxt(BX)
    MOVQ    $0, gobuf_bp(BX)
    // 把 sched.pc 值放入 BX 寄存器
    MOVQ    gobuf_pc(BX), BX
    // JMP 把 BX 寄存器的包含的地址值放入 CPU 的 IP 寄存器，于是，CPU 跳转到该地址继续执行指令
    JMP BX
```

注释地比较详细了。核心的地方是：

```
MOVQ    gobuf_g(BX), DX
// ......
get_tls(CX)
MOVQ    DX, g(CX)
```

第一行，将 gp.sched.g 保存到 DX 寄存器；第二行，我们见得已经比较多了，`get_tls` 将 tls 保存到 CX 寄存器，再将 gp.sched.g 放到 tls[0] 处。这样，当下次再调用 `get_tls` 时，取出的就是 gp，而不再是 g0，这一行完成从 g0 栈切换到 gp。

可能需要提一下的是，Go plan9 汇编中的一些奇怪的符号：

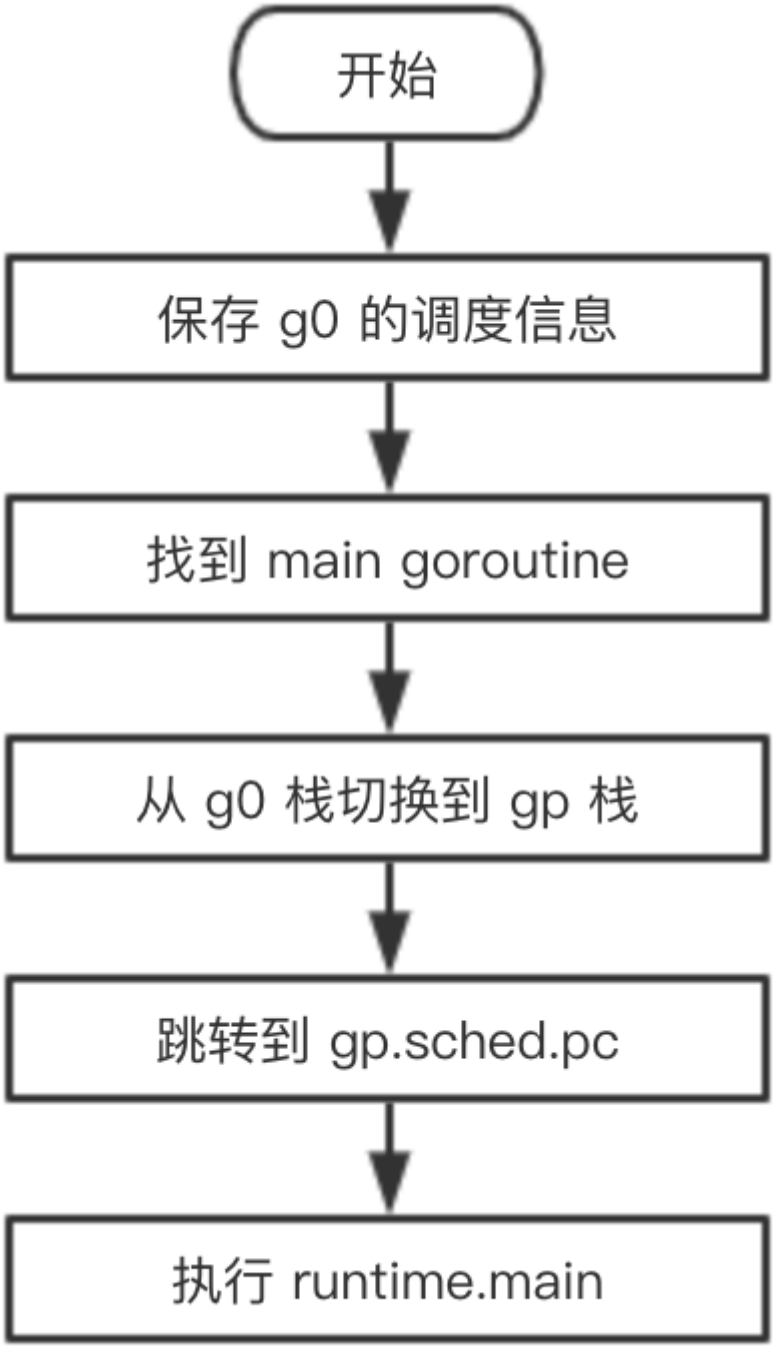
```
MOVQ    buf+0(FP), BX  # &gp.sched --> BX
```

`FP` 是个伪奇存器，前面加 0 表示是第一个寄存器，表示参数的位置，最前面的 buf 表示一个符号。关于 Go 汇编语言的一些知识，可以参考曹大在夜读上的分享和《Go 语言高级编程》的相关章节，地址见参考资料。

接下来，将 gp.sched 的相关成员恢复到 CPU 对应的寄存器。最重要的是 sched.sp 和 sched.pc，前者被恢复到了 SP 寄存器，后者被保存到 BX 寄存器，最后一条跳转指令跳转到新的地址开始执行。通过之前的文章，我们知道，这里保存的就是 `runtime.main` 函数的地址。

最终，调度器完成了这个值得铭记的时刻，从 g0 转到 gp，开始执行 `runtime.main` 函数。

用一张流程图总结一下从 g0 切换到 main goroutine 的过程：




## 参考资料

【欧神 调度循环】https://github.com/changkun/go-under-the-hood/blob/master/book/zh-cn/part2runtime/ch06sched/exec.md

【go 语言核心编程技术 调度器系列】https://mp.weixin.qq.com/s/8eJm5hjwKXya85VnT4y8Cw

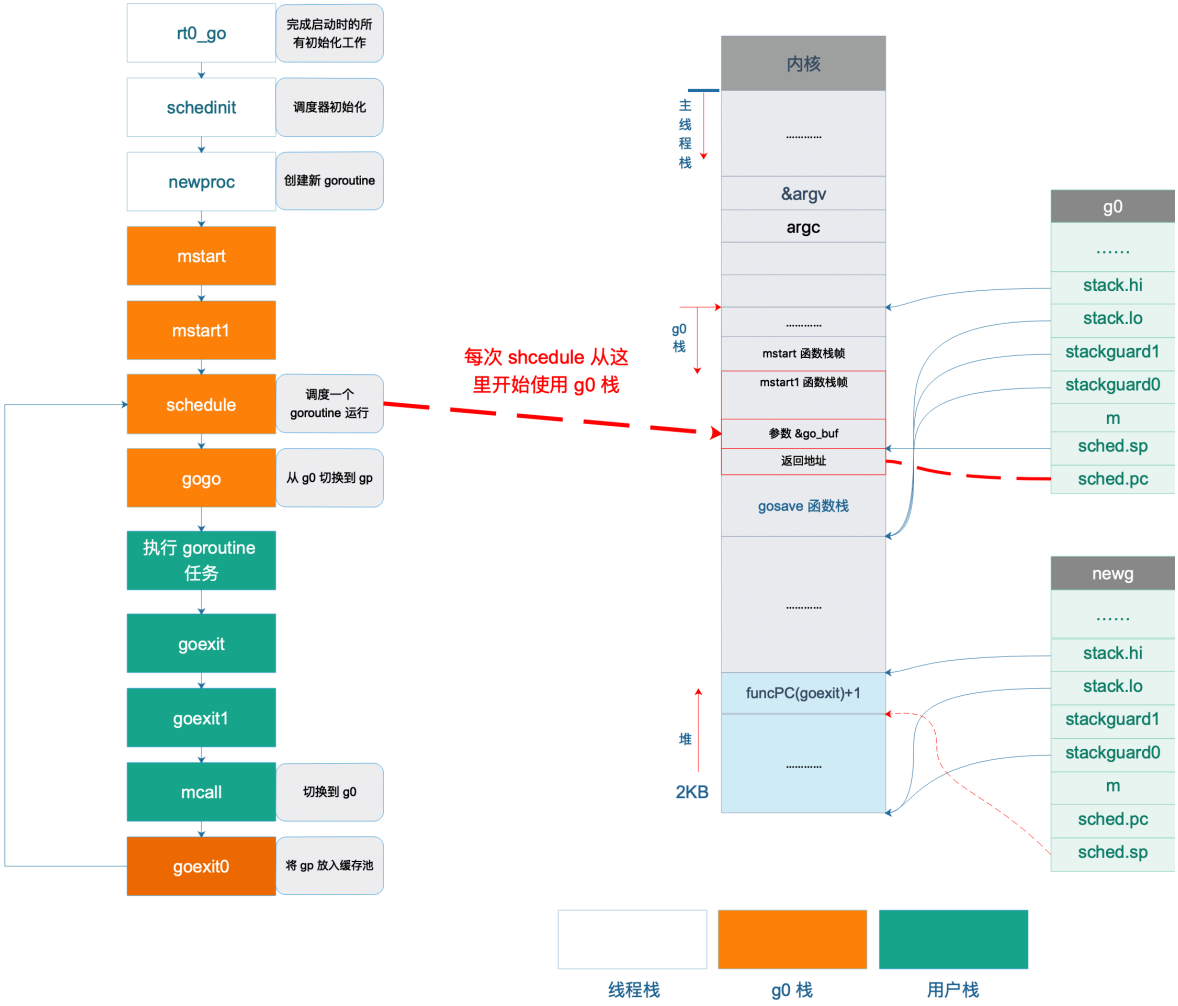
【曹大 Go plan9 汇编】https://github.com/cch123/asmshare/blob/master/layout.md

【Go 语言高级编程】https://chai2010.cn/advanced-go-programming-book/ch3-asm/readme.html

# schedule 循环如何运转

上一节，我们讲完 main goroutine 以及普通 goroutine 的退出过程。main goroutine 退出后直接调用 exit(0) 使得整个进程退出，而普通 goroutine 退出后，则进行了一系列的调用，最终又切到 g0 栈，执行 schedule 函数。

从前面的文章我们知道，普通 goroutine（gp）就是在 schedule 函数中被选中，然后才有机会执行。而现在，gp 执行完之后，再次进入 schedule 函数，形成一个循环。这个循环太长了，我们有必要再重新梳理一下。




如图所示，rt0_go 负责 Go 程序启动的所有初始化，中间进行了很多初始化工作，调用 mstart 之前，已经切换到了 g0 栈，图中不同色块表示使用不同的栈空间。

接着调用 gogo 函数，完成从 g0 栈到用户 goroutine 栈的切换，包括 main goroutine 和普通 goroutine。

之后，执行 main 函数或者用户自定义的 goroutine 任务。

执行完成后，main goroutine 直接调用 eixt(0) 退出，普通 goroutine 则调用 goexit -> goexit1 -> mcall，完成普通 goroutine 退出后的清理工作，然后切换到 g0 栈，调用 goexit0 函数，将普通 goroutine 添加到缓存池中，再调用 schedule 函数进行新一轮的调度。

```
schedule() -> execute() -> gogo() -> goroutine 任务 -> goexit() -> goexit1() -> mcall() -> goexit0() -> schedule()
```

> 可以看出，一轮调度从调用 schedule 函数开始，经过一系列过程再次调用 schedule 函数来进行新一轮的调度，从一轮调度到新一轮调度的过程称之为一个调度循环。

> 这里说的调度循环是指某一个工作线程的调度循环，而同一个Go 程序中存在多个工作线程，每个工作线程都在进行着自己的调度循环。

> 从前面的代码分析可以得知，上面调度循环中的每一个函数调用都没有返回，虽然 `goroutine 任务-> goexit() -> goexit1() -> mcall()` 是在 g2 的栈空间执行的，但剩下的函数都是在 g0 的栈空间执行的。

> 那么问题就来了，在一个复杂的程序中，调度可能会进行无数次循环，也就是说会进行无数次没有返回的函数调用，大家都知道，每调用一次函数都会消耗一定的栈空间，而如果一直这样无返回的调用下去无论 g0 有多少栈空间终究是会耗尽的，那么这里是不是有问题？其实没有问题！关键点就在于，每次执行 mcall 切换到 g0 栈时都是切换到 g0.sched.sp 所指的固定位置，这之所以行得通，正是因为从 schedule 函数开始之后的一系列函数永远都不会返回，所以重用这些函数上一轮调度时所使用过的栈内存是没有问题的。

我再解释一下：栈空间在调用函数时会自动“增大”，而函数返回时，会自动“减小”，这里的增大和减小是指栈顶指针 SP 的变化。上述这些函数都没有返回，说明调用者不需要用到被调用者的返回值，有点像“尾递归”。

因为 g0 一直没有动过，所有它之前保存的 sp 还能继续使用。每一次调度循环都会覆盖上一次调度循环的栈数据，完美！

# 参考资料

【阿波张 非 main goroutine 的退出及调度循环】https://mp.weixin.qq.com/s/XttP9q7-PO7VXhskaBzGqA

# sysmon 后台监控线程做了什么

在 `runtime.main()` 函数中，执行 `runtime_init()` 前，会启动一个 sysmon 的监控线程，执行后台监控任务：

```
systemstack(func() {
    // 创建监控线程，该线程独立于调度器，不需要跟 p 关联即可运行
    newm(sysmon, nil)
})
```

`sysmon` 函数不依赖 P 直接执行，通过 newm 函数创建一个工作线程：

```
func newm(fn func(), _p_ *p) {
    // 创建 m 对象
    mp := allocm(_p_, fn)
    // 暂存 m
    mp.nextp.set(_p_)
    mp.sigmask = initSigmask

    // ......................

    execLock.rlock() // Prevent process clone.
    // 创建系统线程
    newosproc(mp, unsafe.Pointer(mp.g0.stack.hi))
    execLock.runlock()
}
```

先调用 `allocm` 在堆上创建一个 m，接着调用 `newosproc` 函数启动一个工作线程：

```
// src/runtime/os_linux.go
//go:nowritebarrier
func newosproc(mp *m, stk unsafe.Pointer) {
    // ......................

    ret := clone(cloneFlags, stk, unsafe.Pointer(mp), unsafe.Pointer(mp.g0), unsafe.Pointer(funcPC(mstart)))

    // ......................
}
```

核心就是调用 clone 函数创建系统线程，新线程从 mstart 函数开始执行。 `clone` 函数由汇编语言实现：

```
// int32 clone(int32 flags, void *stk, M *mp, G *gp, void (*fn)(void));
TEXT runtime·clone(SB),NOSPLIT,$0
    // 准备系统调用的参数
    MOVL    flags+0(FP), DI
    MOVQ    stk+8(FP), SI
    MOVQ    $0, DX
    MOVQ    $0, R10

    // 将 mp, gp, fn 拷贝到寄存器，对子线程可见
    MOVQ    mp+16(FP), R8
    MOVQ    gp+24(FP), R9
    MOVQ    fn+32(FP), R12

    // 系统调用 clone
    MOVL    $56, AX
```

```
    SYSCALL

    // In parent, return.
    CMPQ    AX, $0
    JEQ    3(PC)
    // 父线程，返回
    MOVL    AX, ret+40(FP)
    RET

    // In child, on new stack.
    // 在子线程中。设置 CPU 栈顶寄存器指向子线程的栈顶
    MOVQ    SI, SP

    // If g or m are nil, skip Go-related setup.
    CMPQ    R8, $0    // m
    JEQ    nog
    CMPQ    R9, $0    // g
    JEQ    nog

    // Initialize m->procid to Linux tid
    // 通过 gettid 系统调用获取线程 ID（tid）
    MOVL    $186, AX    // gettid
    SYSCALL
    // 设置 m.procid = tid
    MOVQ    AX, m_procid(R8)

    // Set FS to point at m->tls.
    // 新线程刚刚创建出来，还未设置线程本地存储，即 m 结构体对象还未与工作线程关联起来，
    // 下面的指令负责设置新线程的 TLS，把 m 对象和工作线程关联起来
    LEAQ    m_tls(R8), DI
    CALL    runtime·settls(SB)

    // In child, set up new stack
    get_tls(CX)
    MOVQ    R8, g_m(R9) // g.m = m
    MOVQ    R9, g(CX) // tls.g = &m.g0
    CALL    runtime·stackcheck(SB)

nog:
    // Call fn
    // 调用 mstart 函数。永不返回
    CALL    R12

    // It shouldn't return. If it does, exit that thread.
    MOVL    $111, DI
    MOVL    $60, AX
    SYSCALL
    JMP    -3(PC)    // keep exiting
```

先是为 clone 系统调用准备参数，参数通过寄存器传递。第一个参数指定内核创建线程时的选项，第二个参数指定新线程应该使用的栈，这两个参数都是通过 newosproc 函数传递进来的。

接着将 m, g0, fn 分别保存到寄存器中，待子线程创建好后再拿出来使用。因为这些参数此时是在父线程的栈上，若不保存到寄存器中，子线程就取不出来了。

> 这个几个参数保存在父线程的寄存器中，创建子线程时，操作系统内核会把父线程所有的寄存器帮我们复制一份给子线程，所以当子线程开始运行时就能拿到父线程保存在寄存器中的值，从而拿到这几个参数。

之后，调用 clone 系统调用，内核帮我们创建出了一个子线程。相当于原来的一个执行分支现在变成了两个执行分支，于是会有两个返回。这和著名的 fork 系统调用类似，根据返回值来判断现在是处于父线程还是子线程。

如果是父线程，就直接返回了。如果是子线程，接着还要执行一堆操作，例如设置 tls，设置 m.procid 等等。

最后执行 mstart 函数，这是在 newosproc 函数传递进来的。 `mstart` 函数再调用 `mstart1` ，在 `mstart1` 里会执行这一行：

```
// 执行启动函数。初始化过程中，fn == nil
if fn := _g_.m.mstartfn; fn != nil {
    fn()
}
```

之前我们在讲初始化的时候，这里的 fn 是空，会跳过的。但在这里，fn 就是最开始在 `runtime.main` 里设置的 `sysmon` 函数，因此这里会执行 `sysmon` ，而它又是一个无限循环，永不返回。

所以，这里不会执行到 mstart1 函数后面的 schedule 函数，也就不会进入 schedule 循环。因此这是一个不用和 p 结合的 m，它直接在后台执行，默默地执行监控任务。

接下来，我们就来看 `sysmon` 函数到底做了什么？

> `sysmon` 执行一个无限循环，一开始每次循环休眠 20us，之后（1 ms 后）每次休眠时间倍增，最终每一轮都会休眠 10ms。

> `sysmon` 中会进行 netpool（获取 fd 事件）、retake（抢占）、forcegc（按时间强制执行 gc），scavenge heap（释放自由列表中多余的项减少内存占用）等处理。

和调度相关的，我们只关心 retake 函数：

```
func retake(now int64) uint32 {
    n := 0
    // 遍历所有的 p
    for i := int32(0); i < gomaxprocs; i++ {
        _p_ := allp[i]
        if _p_ == nil {
            continue
        }
        // 用于 sysmon 线程记录被监控 p 的系统调用时间和运行时间
        pd := &_p_.sysmontick
        // p 的状态
        s := _p_.status
        if s == _Psyscall {
            // P 处于系统调用之中，需要检查是否需要抢占
            // Retake P from syscall if it's there for more than 1 sysmon tick (at least 20us).
            // _p_.syscalltick 用于记录系统调用的次数，在完成系统调用之后加 1
            t := int64(_p_.syscalltick)
            if int64(pd.syscalltick) != t {
                // pd.syscalltick != _p_.syscalltick，说明已经不是上次观察到的系统调用了，
                // 而是另外一次系统调用，所以需要重新记录 tick 和 when 值
                pd.syscalltick = uint32(t)
                pd.syscallwhen = now
                continue
            }

            // 只要满足下面三个条件中的任意一个，则抢占该 p，否则不抢占
            // 1. p 的运行队列里面有等待运行的 goroutine
            // 2. 没有无所事事的 p
            // 3. 从上一次监控线程观察到 p 对应的 m 处于系统调用之中到现在已经超过 10 毫秒
            if runqempty(_p_) && atomic.Load(&sched.nmspinning)+atomic.Load(&sched.npidle) > 0 && pd.syscallw
hen+10*1000*1000 > now {
                continue
```

```
            }

            incidlelocked(-1)
            if atomic.Cas(&_p_.status, s, _Pidle) {
                // ·····················
                n++
                _p_.syscalltick++
                // 寻找一新的 m 接管 p
                handoffp(_p_)
            }
            incidlelocked(1)
        } else if s == _Prunning {
            // P 处于运行状态，检查是否运行得太久了
            // Preempt G if it's running for too long.
            // 每发生一次调度，调度器 ++ 该值
            t := int64(_p_.schedtick)
            if int64(pd.schedtick) != t {
                pd.schedtick = uint32(t)
                pd.schedwhen = now
                continue
            }
            //pd.schedtick == t 说明(pd.schedwhen ~ now)这段时间未发生过调度
            // 这段时间是同一个goroutine一直在运行，检查是否连续运行超过了 10 毫秒
            if pd.schedwhen+forcePreemptNS > now {
                continue
            }
            // 连续运行超过 10 毫秒了，发起抢占请求
            preemptone(_p_)
        }
    }
    return uint32(n)
}
```

从代码来看，主要会对处于 `_Psyscall` 和 `_Prunning` 状态的 p 进行抢占。

# 抢占进行系统调用的 P

当 P 处于 `_Psyscall` 状态时，表明对应的 goroutine 正在进行系统调用。如果抢占 p，需要满足几个条件：

1. p 的本地运行队列里面有等待运行的 goroutine。这时 p 绑定的 g 正在进行系统调用，无法去执行其他的 g，因此需要接管 p 来执行其他的 g。

2. 没有“无所事事”的 p。 `sched.nmspinning` 和 `sched.npidle` 都为 0，这就意味着没有“找工作”的 m，也没有空闲的 p，大家都在“忙”，可能有很多工作要做。因此要抢占当前的 p，让它来承担一部分工作。

3. 从上一次监控线程观察到 p 对应的 m 处于系统调用之中到现在已经超过 10 毫秒。这说明系统调用所花费的时间较长，需要对其进行抢占，以此来使得 `retake` 函数返回值不为 0，这样，会保持 sysmon 线程 20 us 的检查周期，提高 sysmon 监控的实时性。

注意，原代码是用的三个与条件，三者都要满足才会执行下面的 continue，也就是不进行抢占。因此要想进行抢占的话，只需要三个条件有一个不满足就行了。于是就有了上述三种情况。

确定要抢占当前 p 后，先使用原子操作将 p 的状态修改为 `_Pidle` ，最后调用 `handoffp` 进行抢占。

```
func handoffp(_p_ *p) {
    // 如果 p 本地有工作或者全局有工作，需要绑定一个 m
    if !runqempty(_p_) || sched.runqsize != 0 {
        startm(_p_, false)
```

```
        return
    }

    // ......................

    // 所有其它 p 都在运行 goroutine，说明系统比较忙，需要启动 m
    if atomic.Load(&sched.nmspinning)+atomic.Load(&sched.npidle) == 0 && atomic.Cas(&sched.nmspinning, 0, 1)
{ // TODO: fast atomic
        // p 没有本地工作，启动一个自旋 m 来找工作
        startm(_p_, true)
        return
    }
    lock(&sched.lock)

    // ......................

    // 全局队列有工作
    if sched.runqsize != 0 {
        unlock(&sched.lock)
        startm(_p_, false)
        return
    }

    // ......................

    // 没有工作要处理，把 p 放入全局空闲队列
    pidleput(_p_)
    unlock(&sched.lock)
}
```

`handoffp` 再次进行场景判断，以调用 `startm` 启动一个工作线程来绑定 p，使得整体工作继续推进。

当 p 的本地运行队列或全局运行队列里面有待运行的 goroutine，说明还有很多工作要做，调用 `startm(_p_, false)` 启动一个 m 来结合 p，继续工作。

当除了当前的 p 外，其他所有的 p 都在运行 goroutine，说明天下太平，每个人都有自己的事做，唯独自己没有。为了全局更快地完成工作，需要启动一个 m，且要使得 m 处于自旋状态，和 p 结合之后，尽快找到工作。

最后，如果实在没有工作要处理，就将 p 放入全局空闲队列里。

我们接着来看 `startm` 函数都做了些什么：

```
// runtime/proc.go
//
// 调用 m 来绑定 p，如果没有 m，那就新建一个
// 如果 p 为空，那就尝试获取一个处于空闲状态的 p，如果找到 p，那就什么都不做
func startm(_p_ *p, spinning bool) {
    lock(&sched.lock)
    if _p_ == nil {
        // 没有指定 p 则需要从全局空闲队列中获取一个 p
        _p_ = pidleget()
        if _p_ == nil {
            unlock(&sched.lock)
            if spinning {
                // 如果找到 p，放弃。还原全局处于自旋状态的 m 的数量
                if int32(atomic.Xadd(&sched.nmspinning, -1)) < 0 {
                    throw("startm: negative nmspinning")
                }
            }
            // 没有空闲的 p，直接返回
```

```
            return
        }
    }

    // 从 m 空闲队列中获取正处于睡眠之中的工作线程，
    // 所有处于睡眠状态的 m 都在此队列中
    mp := mget()
    unlock(&sched.lock)
    if mp == nil {
        // 如果没有找到 m
        var fn func()
        if spinning {
            // The caller incremented nmspinning, so set m.spinning in the new M.
            fn = mspinning
        }
        // 创建新的工作线程
        newm(fn, _p_)
        return
    }
    if mp.spinning {
        throw("startm: m is spinning")
    }
    if mp.nextp != 0 {
        throw("startm: m has p")
    }
    if spinning && !runqempty(_p_) {
        throw("startm: p has runnable gs")
    }
    // The caller incremented nmspinning, so set m.spinning in the new M.
    mp.spinning = spinning
    // 设置 m 马上要结合的 p
    mp.nextp.set(_p_)
    // 唤醒 m
    notewakeup(&mp.park)
}
```

首先处理 p 为空的情况，直接从全局空闲 p 队列里找，如果没找到，则直接返回。如果设置了 spinning 为 true 的话，还需要还原全局的处于自旋状态的 m 的数值： `&sched.nmspinning` 。

搞定了 p，接下来看 m。先调用 `mget` 函数从全局空闲的 m 队列里获取一个 m，如果没找到 m，则要调用 newm 新创建一个 m，并且如果设置了 spinning 为 true 的话，先要设置好 mstartfn：

```
func mspinning() {
    // startm's caller incremented nmspinning. Set the new M's spinning.
    getg().m.spinning = true
}
```

这样，启动 m 后，在 mstart1 函数里，进入 schedule 循环前，执行 mstartfn 函数，使得 m 处于自旋状态。

接下来是正常情况下（找到了 p 和 m）的处理：

```
mp.spinning = spinning
// 设置 m 马上要结合的 p
mp.nextp.set(_p_)
// 唤醒 m
notewakeup(&mp.park)
```

设置 nextp 为找到的 p，调用 `notewakeup` 唤醒 m。之前我们讲 findrunnable 函数的时候，对于最后没有找到工作的 m，我们调用 `notesleep(&_g_.m.park)`，使得 m 进入睡眠状态。现在终于有工作了，需要老将出山，将其唤醒：

```
// src/runtime/lock_futex.go
func notewakeup(n *note) {
    // 设置 n.key = 1, 被唤醒的线程通过查看该值是否等于 1
    // 来确定是被其它线程唤醒还是意外从睡眠中苏醒
    old := atomic.Xchg(key32(&n.key), 1)
    if old != 0 {
        print("notewakeup - double wakeup (", old, ")\n")
        throw("notewakeup - double wakeup")
    }
    futexwakeup(key32(&n.key), 1)
}
```

> `notewakeup` 函数首先使用 `atomic.Xchg` 设置 `note.key` 值为 1，这是为了使被唤醒的线程可以通过查看该值是否等于 1 来确定是被其它线程唤醒还是意外从睡眠中苏醒了过来。

> 如果该值为 1 则表示是被唤醒的，可以继续工作，但如果该值为 0 则表示是意外苏醒，需要再次进入睡眠。

调用 `futexwakeup` 来唤醒工作线程，它和 `futexsleep` 是相对的。

```
func futexwakeup(addr *uint32, cnt uint32) {
    // 调用 futex 函数唤醒工作线程
    ret := futex(unsafe.Pointer(addr), _FUTEX_WAKE, cnt, nil, nil, 0)
    if ret >= 0 {
        return
    }

    // ......................

}
```

`futex` 由汇编语言实现，前面已经分析过，这里就不重复了。主要内容就是先准备好参数，然后进行系统调用，由内核唤醒线程。

> 内核在完成唤醒工作之后当前工作线程从内核返回到 futex 函数继续执行 SYSCALL 指令之后的代码并按函数调用链原路返回，继续执行其它代码。

> 而被唤醒的工作线程则由内核负责在适当的时候调度到 CPU 上运行。

# 抢占长时间运行的 P

我们知道，Go scheduler 采用的是一种称为协作式的抢占式调度，就是说并不强制调度，大家保持协作关系，互相信任。对于长时间运行的 P，或者说绑定在 P 上的长时间运行的 goroutine，sysmon 会检测到这种情况，然后设置一些标志，表示 goroutine 自己让出 CPU 的执行权，给其他 goroutine 一些机会。

接下来我们就来分析当 P 处于 `_Prunning` 状态的情况。`sysmon` 扫描每个 p 时，都会记录下当前调度器调度的次数和当前时间，数据记录在结构体：

```
type sysmontick struct {
    schedtick   uint32
    schedwhen   int64
    syscalltick uint32
```

```
    syscallwhen int64
}
```

前面两个字段记录调度器调度的次数和时间，后面两个字段记录系统调用的次数和时间。

在下一次扫描时，对比 sysmon 记录下的 p 的调度次数和时间，与当前 p 自己记录下的调度次数和时间对比，如果一致。说明 P 在这一段时间内一直在运行同一个 goroutine。那就来计算一下运行时间是否太长了。

如果发现运行时间超过了 10 ms，则要调用 `preemptone(_p_)` 发起抢占的请求：

```
func preemptone(_p_ *p) bool {
    mp := _p_.m.ptr()
    if mp == nil || mp == getg().m {
        return false
    }
    // 被抢占的 goroutine
    gp := mp.curg
    if gp == nil || gp == mp.g0 {
        return false
    }

    // 设置抢占标志
    gp.preempt = true

    // 在 goroutine 内部的每次调用都会比较栈顶指针和 g.stackguard0，
    // 来判断是否发生了栈溢出。stackPreempt 非常大的一个数，比任何栈都大
    // stackPreempt = 0xfffffade
    gp.stackguard0 = stackPreempt
    return true
}
```

基本上只是将 stackguard0 设置了一个很大的值，而检查 stackguard0 的地方在函数调用前的一段汇编代码里进行。

举一个简单的例子：

```
package main

import "fmt"

func main() {
    fmt.Println("hello qcrao.com!")
}
```

执行命令：

```
go tool compile -S main.go
```

得到汇编代码：

```
"".main STEXT size=120 args=0x0 locals=0x48
    0x0000 00000 (test26.go:5)    TEXT    "".main(SB), $72-0
    0x0000 00000 (test26.go:5)    MOVQ    (TLS), CX
    0x0009 00009 (test26.go:5)    CMPQ    SP, 16(CX)
    0x000d 00013 (test26.go:5)    JLS    113
    0x000f 00015 (test26.go:5)    SUBQ    $72, SP
    0x0013 00019 (test26.go:5)    MOVQ    BP, 64(SP)
    0x0018 00024 (test26.go:5)    LEAQ    64(SP), BP
```

```
    0x001d 00029 (test26.go:5)    FUNCDATA    $0, gclocals·69c1753bd5f81501d95132d08af04464(SB)
    0x001d 00029 (test26.go:5)    FUNCDATA    $1, gclocals·e226d4ae4a7cad8835311c6a4683c14f(SB)
    0x001d 00029 (test26.go:6)    MOVQ    $0, ""..autotmp_0+48(SP)
    0x0026 00038 (test26.go:6)    MOVQ    $0, ""..autotmp_0+56(SP)
    0x002f 00047 (test26.go:6)    LEAQ    type.string(SB), AX
    0x0036 00054 (test26.go:6)    MOVQ    AX, ""..autotmp_0+48(SP)
    0x003b 00059 (test26.go:6)    LEAQ    ""..statictmp_0(SB), AX
    0x0042 00066 (test26.go:6)    MOVQ    AX, ""..autotmp_0+56(SP)
    0x0047 00071 (test26.go:6)    LEAQ    ""..autotmp_0+48(SP), AX
    0x004c 00076 (test26.go:6)    MOVQ    AX, (SP)
    0x0050 00080 (test26.go:6)    MOVQ    $1, 8(SP)
    0x0059 00089 (test26.go:6)    MOVQ    $1, 16(SP)
    0x0062 00098 (test26.go:6)    PCDATA    $0, $1
    0x0062 00098 (test26.go:6)    CALL    fmt.Println(SB)
    0x0067 00103 (test26.go:7)    MOVQ    64(SP), BP
    0x006c 00108 (test26.go:7)    ADDQ    $72, SP
    0x0070 00112 (test26.go:7)    RET
    0x0071 00113 (test26.go:7)    NOP
    0x0071 00113 (test26.go:5)    PCDATA    $0, $-1
    0x0071 00113 (test26.go:5)    CALL    runtime.morestack_noctxt(SB)
    0x0076 00118 (test26.go:5)    JMP    0
```

以前看这段代码的时候会直接跳过前面的几行代码，看不懂。这次能看懂了！所以，那些暂时看不懂的，先放一放，没关系，让子弹飞一会儿，很多东西回过头再来看就会豁然开朗，这就是一个很好的例子。

```
0x0000 00000 (test26.go:5)    MOVQ    (TLS), CX
```

将本地存储 tls 保存到 CX 寄存器中，（TLS）表示它所关联的 g，这里就是前面所讲到的 main gouroutine。

```
0x0009 00009 (test26.go:5)    CMPQ    SP, 16(CX)
```

比较 SP 寄存器（代表当前 main goroutine 的栈顶寄存器）和 16(CX)，我们看下 g 结构体：

```
type g struct {
    // goroutine 使用的栈
    stack       stack   // offset known to runtime/cgo
    // 用于栈的扩张和收缩检查
    stackguard0 uintptr // offset known to liblink
    // ......................
}
```

对象 g 的第一个字段是 stack 结构体：

```
type stack struct {
    lo uintptr
    hi uintptr
}
```

共 16 字节。而 `16(CX)` 表示 g 对象的第 16 个字节，跳过了 g 的第一个字段，也就是 `g.stackguard0` 字段。

如果 SP 小于 g.stackguard0，这是必然的，因为前面已经把 g.stackguard0 设置成了一个非常大的值，因此跳转到了 113 行。

```
0x0071 00113 (test26.go:7)    NOP
0x0071 00113 (test26.go:5)    PCDATA    $0, $-1
```

```
0x0071 00113 (test26.go:5)    CALL    runtime.morestack_noctxt(SB)
0x0076 00118 (test26.go:5)    JMP    0
```

调用 `runtime.morestack_noctxt` 函数：

```
// src/runtime/asm_amd64.s
TEXT runtime·morestack_noctxt(SB),NOSPLIT,$0
    MOVL    $0, DX
    JMP    runtime·morestack(SB)
```

直接跳转到 `morestack` 函数：

```
TEXT runtime·morestack(SB),NOSPLIT,$0-0
    // Cannot grow scheduler stack (m->g0).
    get_tls(CX)
    // BX = g, g 表示 main goroutine
    MOVQ    g(CX), BX
    // BX = g.m
    MOVQ    g_m(BX), BX
    // SI = g.m.g0
    MOVQ    m_g0(BX), SI
    CMPQ    g(CX), SI
    JNE    3(PC)
    CALL    runtime·badmorestackg0(SB)
    INT    $3

    // ....................

    // Set g->sched to context in f.
    // 将函数的返回地址保存到 AX 寄存器
    MOVQ    0(SP), AX // f's PC
    // 将函数的返回地址保存到 g.sched.pc
    MOVQ    AX, (g_sched+gobuf_pc)(SI)
    // g.sched.g = g
    MOVQ    SI, (g_sched+gobuf_g)(SI)
    // 取地址操作符，调用 morestack_noctxt 之前的 rsp
    LEAQ    8(SP), AX // f's SP
    // 将 main 函数的栈顶地址保存到 g.sched.sp
    MOVQ    AX, (g_sched+gobuf_sp)(SI)
    // 将 BP 寄存器保存到 g.sched.bp
    MOVQ    BP, (g_sched+gobuf_bp)(SI)
    // newstack will fill gobuf.ctxt.

    // Call newstack on m->g0's stack.
    // BX = g.m.g0
    MOVQ    m_g0(BX), BX
    // 将 g0 保存到本地存储 tls
    MOVQ    BX, g(CX)
    // 把 g0 栈的栈顶寄存器的值恢复到 CPU 的寄存器 SP，达到切换栈的目的，下面这一条指令执行之前，
    // CPU 还是使用的调用此函数的 g 的栈，执行之后 CPU 就开始使用 g0 的栈了
    MOVQ    (g_sched+gobuf_sp)(BX), SP
    // 准备参数
    PUSHQ    DX    // ctxt argument
    // 不返回
    CALL    runtime·newstack(SB)
    MOVQ    $0, 0x1003    // crash if newstack returns
    POPQ    DX    // keep balance check happy
    RET
```

主要做的工作就是将当前 goroutine，也就是 main goroutine 的和调度相关的信息保存到 g.sched 中，以便在调度到它执行时，可以恢复。

最后，将 g0 的地址保存到 tls 本地存储，并且切到 g0 栈执行之后的代码。继续调用 newstack 函数：

```
func newstack(ctxt unsafe.Pointer) {
    // thisg = g0
    thisg := getg()

    // ......................

    // gp = main goroutine
    gp := thisg.m.curg
    // Write ctxt to gp.sched. We do this here instead of in
    // morestack so it has the necessary write barrier.
    gp.sched.ctxt = ctxt

    // ......................

    morebuf := thisg.m.morebuf
    thisg.m.morebuf.pc = 0
    thisg.m.morebuf.lr = 0
    thisg.m.morebuf.sp = 0
    thisg.m.morebuf.g = 0

    // 检查 g.stackguard0 是否被设置成抢占标志
    preempt := atomic.Loaduintptr(&gp.stackguard0) == stackPreempt

    if preempt {
        if thisg.m.locks != 0 || thisg.m.mallocing != 0 || thisg.m.preemptoff != "" || thisg.m.p.ptr().status
!= _Prunning {
            // 还原 stackguard0 为正常值，表示我们已经处理过抢占请求了
            gp.stackguard0 = gp.stack.lo + _StackGuard
            // 不抢占，调用 gogo 继续运行当前这个 g，不需要调用 schedule 函数去挑选另一个 goroutine
            gogo(&gp.sched) // never return
        }
    }

    // ......................

    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")
        }
        // Synchronize with scang.
        casgstatus(gp, _Grunning, _Gwaiting)

        // ......................

        // Act like goroutine called runtime.Gosched.
        // 修改为 running，调度起来运行
        casgstatus(gp, _Gwaiting, _Grunning)
        // 调用 gopreempt_m 把 gp 切换出去
        gopreempt_m(gp) // never return
    }

    // ......................
}
```

去掉了很多暂时还看不懂的地方，留到后面再研究。只关注有关抢占相关的。第一次判断 preempt 标志是 true 时，检查了 g 的状态，发现不能抢占，例如它所绑定的 P 的状态不是 `_Prunning` ，那就恢复它的 stackguard0 字段，下次就不会走这一套流程了。然后，调用 `gogo(&gp.sched)` 继续执行当前的 goroutine。

中间又处理了很多判断流程，再次判断 preempt 标志是 true 时，调用 `gopreempt_m(gp)` 将 gp 切换出去。

```
func gopreempt_m(gp *g) {
    if trace.enabled {
        traceGoPreempt()
    }
    goschedImpl(gp)
}
```

最终调用 `goschedImpl` 函数：

```
func goschedImpl(gp *g) {
    status := readgstatus(gp)
    if status&^_Gscan != _Grunning {
        dumpgstatus(gp)
        throw("bad g status")
    }
    // 更改 gp 的状态
    casgstatus(gp, _Grunning, _Grunnable)
    // 解除 m 和 g 的关系
    dropg()
    lock(&sched.lock)
    // 将 gp 放入全局可运行队列
    globrunqput(gp)
    unlock(&sched.lock)

    // 进入新一轮的调度循环
    schedule()
}
```

将 gp 的状态改为 `_Grunnable` ，放入全局可运行队列，等待下次有 m 来全局队列找工作时才能继续运行，毕竟你已经运行这么长时间了，给别人一点机会嘛。

最后，调用 `schedule()` 函数进入新一轮的调度循环，会找出一个 goroutine 来运行，永不返回。

这样，关于 sysmon 线程在关于调度这块到底做了啥，我们已经回答完了。总结一下：

1. 抢占处于系统调用的 P，让其他 m 接管它，以运行其他的 goroutine。
2. 将运行时间过长的 goroutine 调度出去，给其他 goroutine 运行的机会。

# 一个调度相关的陷阱

由于 Go 语言是协作式的调度，不会像线程那样，在时间片用完后，由 CPU 中断任务强行将其调度走。对于 Go 语言中运行时间过长的 goroutine，Go scheduler 有一个后台线程在持续监控，一旦发现 goroutine 运行超过 10 ms，会设置 goroutine 的“抢占标志位”，之后调度器会处理。但是设置标志位的时机只有在函数“序言”部分，对于没有函数调用的就没有办法了。

> Golang implements a co-operative partially preemptive scheduler.

所以在某些极端情况下，会掉进一些陷阱。下面这个例子来自参考资料【scheduler 的陷阱】。

```
func main() {
    var x int
    threads := runtime.GOMAXPROCS(0)
    for i := 0; i < threads; i++ {
        go func() {
            for { x++ }
        }()
    }
    time.Sleep(time.Second)
    fmt.Println("x =", x)
}
```

运行结果是：在死循环里出不来，不会输出最后的那条打印语句。

为什么？上面的例子会启动和机器的 CPU 核心数相等的 goroutine，每个 goroutine 都会执行一个无限循环。

创建完这些 goroutines 后，main 函数里执行一条 `time.Sleep(time.Second)` 语句。Go scheduler 看到这条语句后，简直高兴坏了，要来活了。这是调度的好时机啊，于是主 goroutine 被调度走。先前创建的 `threads` 个 goroutines，刚好“一个萝卜一个坑”，把 M 和 P 都占满了。

在这些 goroutine 内部，又没有调用一些诸如 `channel` ， `time.sleep` 这些会引发调度器工作的事情。麻烦了，只能任由这些无限循环执行下去了。

解决的办法也有，把 threads 减小 1：

```
func main() {
    var x int
    threads := runtime.GOMAXPROCS(0) - 1
    for i := 0; i < threads; i++ {
        go func() {
            for { x++ }
        }()
    }
    time.Sleep(time.Second)
    fmt.Println("x =", x)
}
```

运行结果：

```
x = 0
```

不难理解了吧，主 goroutine 休眠一秒后，被 go schduler 重新唤醒，调度到 M 上继续执行，打印一行语句后，退出。主 goroutine 退出后，其他所有的 goroutine 都必须跟着退出。所谓“覆巢之下 焉有完卵”，一损俱损。

至于为什么最后打印出的 x 为 0，之前的文章《曹大谈内存重排》里有讲到过，这里不再深究了。

还有一种解决办法是在 for 循环里加一句：

```
go func() {
    time.Sleep(time.Second)
    for { x++ }
}()
```

同样可以让 main goroutine 有机会调度执行。

# 什么是 go shceduler

## 什么是 sheduler

Go 程序的执行由两层组成：Go Program，Runtime，即用户程序和运行时。它们之间通过函数调用来实现内存管理、channel 通信、goroutines 创建等功能。用户程序进行的系统调用都会被 Runtime 拦截，以此来帮助它进行调度以及垃圾回收相关的工作。

一个展现了全景式的关系如下图：

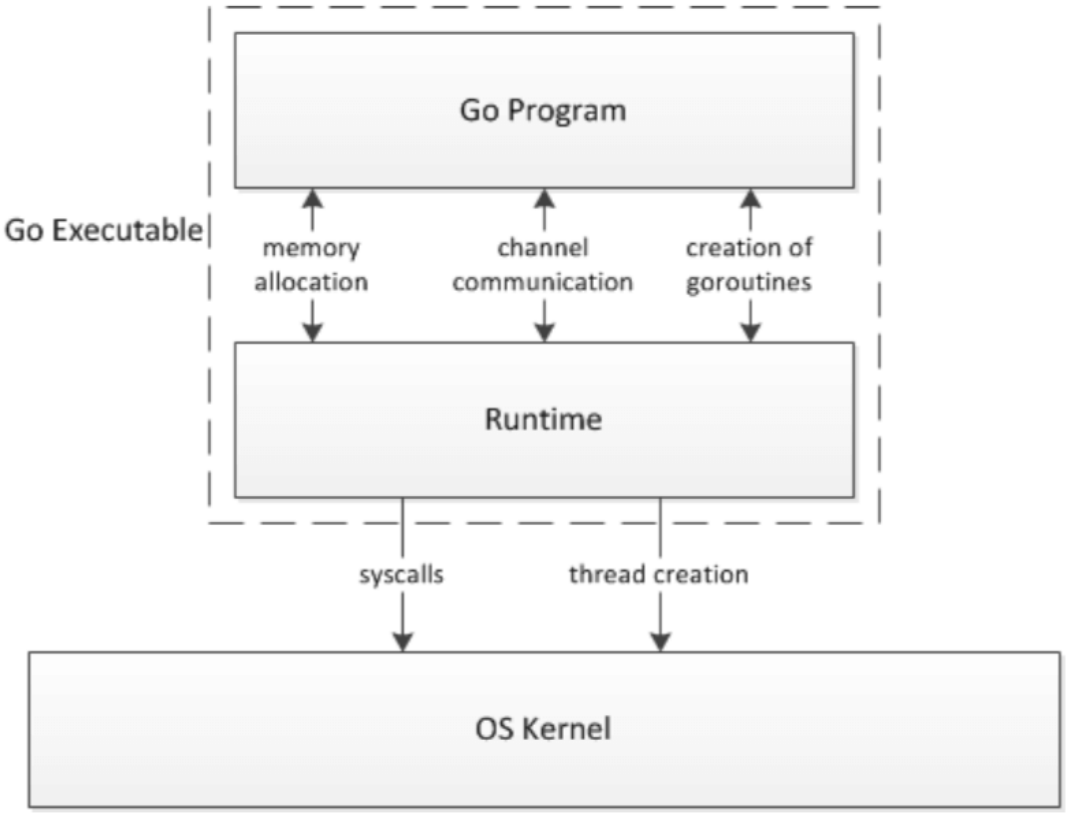



**Figure 1: Diagram of the relationships between the runtime, OS, and programmer defined code**

## 为什么要 scheduler

Go scheduler 可以说是 Go 运行时的一个最重要的部分了。Runtime 维护所有的 goroutines，并通过 scheduler 来进行调度。Goroutines 和 threads 是独立的，但是 goroutines 要依赖 threads 才能执行。

Go 程序执行的高效和 scheduler 的调度是分不开的。

## scheduler 底层原理

实际上在操作系统看来，所有的程序都是在执行多线程。将 goroutines 调度到线程上执行，仅仅是 runtime 层面的一个概念，在操作系统之上的层面。

有三个基础的结构体来实现 goroutines 的调度。g，m，p。

`g` 代表一个 goroutine，它包含：表示 goroutine 栈的一些字段，指示当前 goroutine 的状态，指示当前运行到的指令地址，也就是 PC 值。

`m` 表示内核线程，包含正在运行的 goroutine 等字段。

`p` 代表一个虚拟的 Processor，它维护一个处于 Runnable 状态的 g 队列，`m` 需要获得 `p` 才能运行 `g`。

当然还有一个核心的结构体：`sched`，它总览全局。

Runtime 起始时会启动一些 G：垃圾回收的 G，执行调度的 G，运行用户代码的 G；并且会创建一个 M 用来开始 G 的运行。随着时间的推移，更多的 G 会被创建出来，更多的 M 也会被创建出来。

当然，在 Go 的早期版本，并没有 p 这个结构体，`m` 必须从一个全局的队列里获取要运行的 `g`，因此需要获取一个全局的锁，当并发量大的时候，锁就成了瓶颈。后来在大神 Dmitry Vyokov 的实现里，加上了 `p` 结构体。每个 `p` 自己维护一个处于 Runnable 状态的 `g` 的队列，解决了原来的全局锁问题。

Go scheduler 的目标：

> For scheduling goroutines onto kernel threads.


Go scheduler 的核心思想是：

1. reuse threads；
2. 限制同时运行（不包含阻塞）的线程数为 N，N 等于 CPU 的核心数目；
3. 线程私有的 runqueues，并且可以从其他线程 stealing goroutine 来运行，线程阻塞后，可以将 runqueues 传递给其他线程。

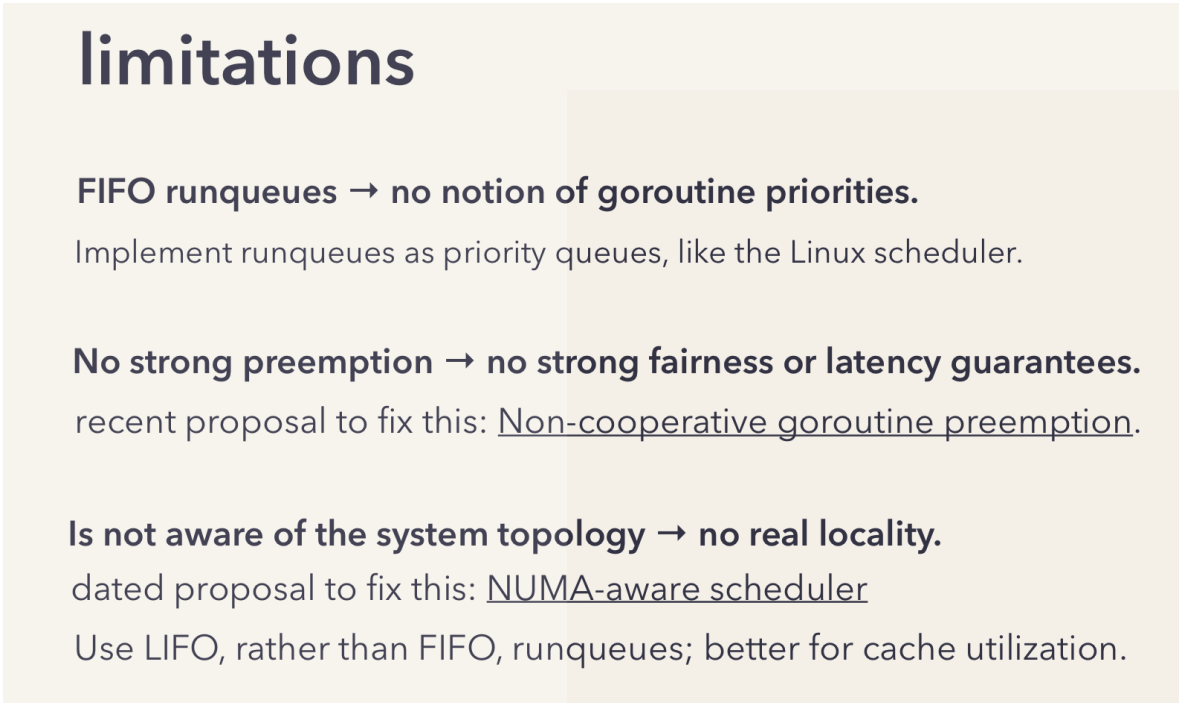
为什么需要 P 这个组件，直接把 runqueues 放到 M 不行吗？

> You might wonder now, why have contexts at all? Can’t we just put the runqueues on the threads and get rid of contexts? Not really. The reason we have contexts is so that we can hand them off to other threads if the running thread needs to block for some reason.

> An example of when we need to block, is when we call into a syscall. Since a thread cannot both be executing code and be blocked on a syscall, we need to hand off the context so it can keep scheduling.

翻译一下，当一个线程阻塞的时候，将和它绑定的 P 上的 goroutines 转移到其他线程。

Go scheduler 会启动一个后台线程 sysmon，用来检测长时间（超过 10 ms）运行的 goroutine，将其调度到 global runqueues。这是一个全局的 runqueue，优先级比较低，以示惩罚。




## 总览

通常讲到 Go scheduler 都会提到 GPM 模型，我们来一个个地看。

下图是我使用的 mac 的硬件信息，只有 2 个核。


但是配上 CPU 的超线程，1 个核可以变成 2 个，所以当我在 mac 上运行下面的程序时，会打印出 4。

```
func main() {
    // NumCPU 返回当前进程可以用到的逻辑核心数
    fmt.Println(runtime.NumCPU())
}
```

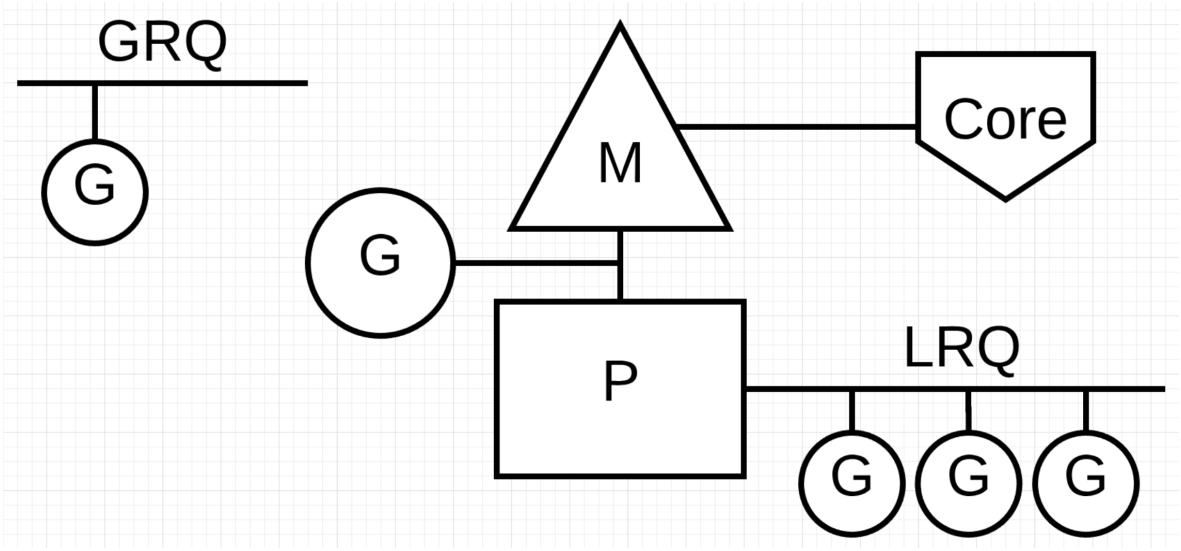
因为 NumCPU 返回的是逻辑核心数，而非物理核心数，所以最终结果是 4。

Go 程序启动后，会给每个逻辑核心分配一个 P（Logical Processor）；同时，会给每个 P 分配一个 M（Machine，表示内核线程），这些内核线程仍然由 OS scheduler 来调度。

总结一下，当我在本地启动一个 Go 程序时，会得到 4 个系统线程去执行任务，每个线程会搭配一个 P。

在初始化时，Go 程序会有一个 G（initial Goroutine），执行指令的单位。G 会在 M 上得到执行，内核线程是在 CPU 核心上调度，而 G 则是在 M 上进行调度。

G、P、M 都说完了，还有两个比较重要的组件没有提到： 全局可运行队列（GRQ）和本地可运行队列（LRQ）。 LRQ 存储本地（也就是具体的 P）的可运行 goroutine，GRQ 存储全局的可运行 goroutine，这些 goroutine 还没有分配到具体的 P。




Go scheduler 是 Go runtime 的一部分，它内嵌在 Go 程序里，和 Go 程序一起运行。因此它运行在用户空间，在 kernel 的上一层。和 Os scheduler 抢占式调度（preemptive）不一样，Go scheduler 采用协作式调度（cooperating）。

> Being a cooperating scheduler means the scheduler needs well-defined user space events that happen at safe points in the code to make scheduling decisions.

协作式调度一般会由用户设置调度点，例如 python 中的 yield 会告诉 Os scheduler 可以将我调度出去了。

但是由于在 Go 语言里，goroutine 调度的事情是由 Go runtime 来做，并非由用户控制，所以我们依然可以将 Go scheduler 看成是抢占式调度，因为用户无法预测调度器下一步的动作是什么。

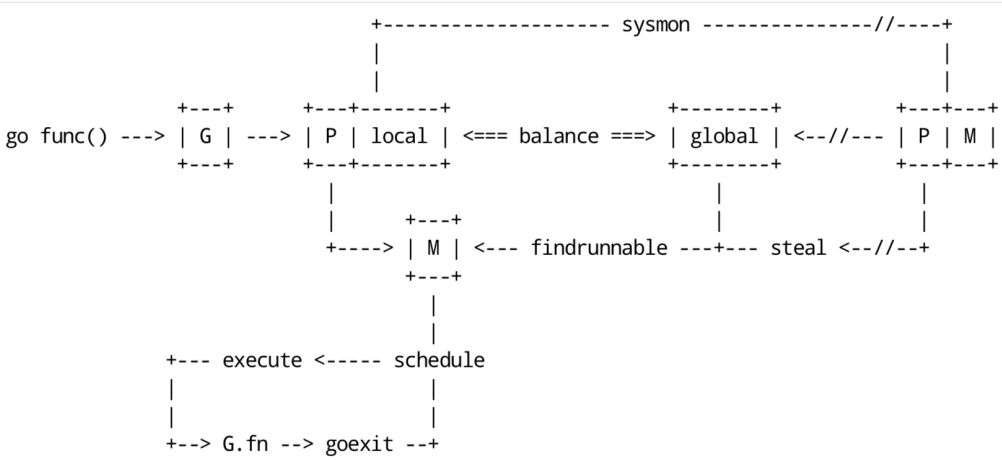
和线程类似，goroutine 的状态也是三种（简化版的）：

| 状态 | 解释 |
| --- | --- |
| Waiting | 等待状态，goroutine 在等待某件事的发生。例如等待网络数据、硬盘；调用操作系统 API；等待内存同步访问条件 ready，如 atomic, mutexes |
| Runnable | 就绪状态，只要给 M 我就可以运行 |

| 状态 | 解释 |
| --- | --- |
| Executing | 运行状态。goroutine 在 M 上执行指令，这是我们想要的 |

下面这张 GPM 全局的运行示意图见得比较多，可以留着，看完后面的系列文章之后再回头来看，还是很有感触的：

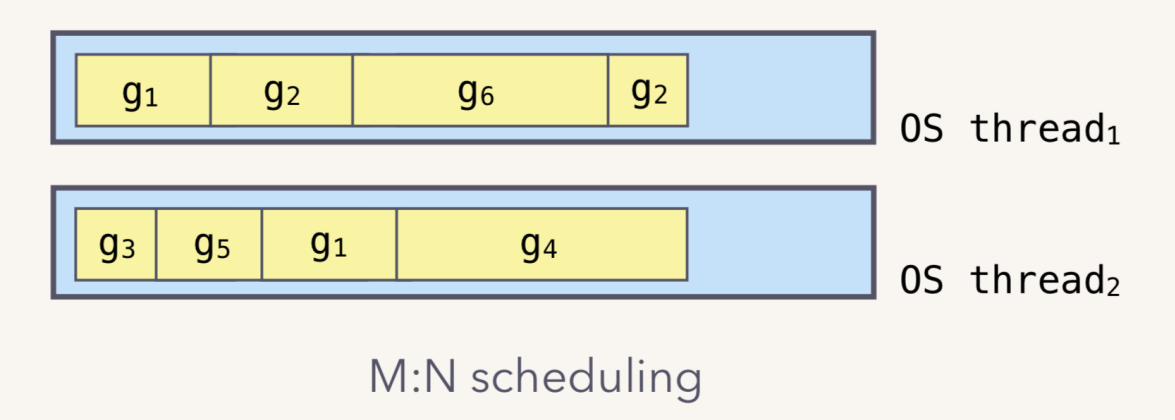
## Workflow




1. go creates a new goroutine
2. newly created goroutine being put into local or global queue
3. A M is being waken or created to execute goroutine
4. Schedule loop
5. Try its best to get a goroutine to execute
6. Clear, reenter schedule loop

# 什么是M:N模型

我们都知道，Go runtime 会负责 goroutine 的生老病死，从创建到销毁，都一手包办。Runtime 会在程序启动的时候，创建 M 个线程（CPU 执行调度的单位），之后创建的 N 个 goroutine 都会依附在这 M 个线程上执行。这就是 M:N 模型：



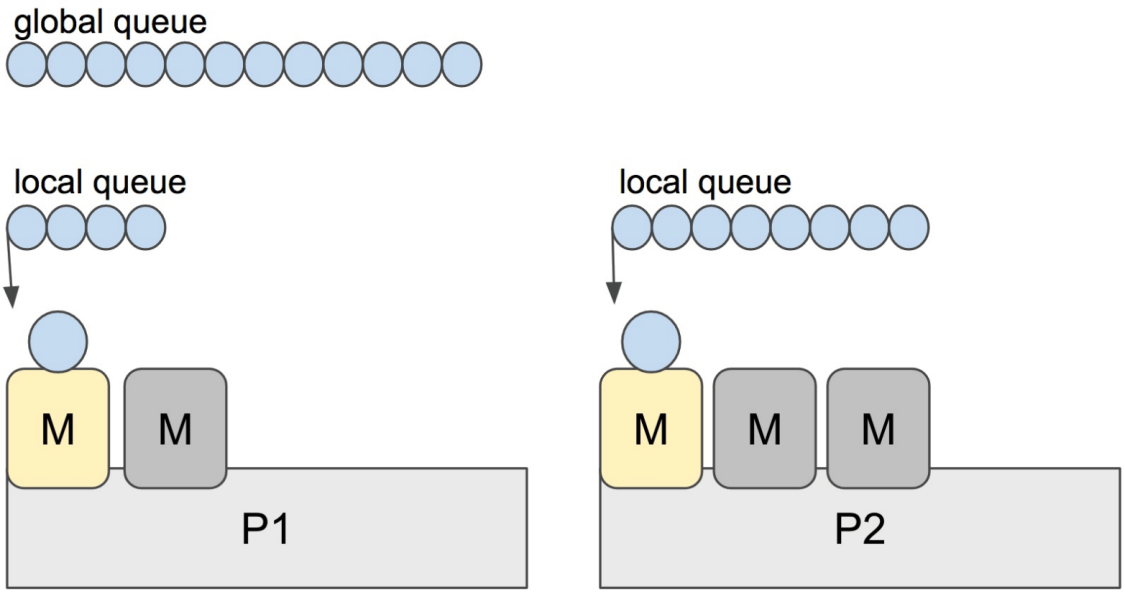

在同一时刻，一个线程上只能跑一个 goroutine。当 goroutine 发生阻塞（例如上篇文章提到的向一个 channel 发送数据，被阻塞）时，runtime 会把当前 goroutine 调度走，让其他 goroutine 来执行。目的就是不让一个线程闲着，榨干 CPU 的每一滴油水。

# 什么是workstealing

Go scheduler 的职责就是将所有处于 runnable 的 goroutines 均匀分布到在 P 上运行的 M。

当一个 P 发现自己的 LRQ 已经没有 G 时，会从其他 P “偷” 一些 G 来运行。看看这是什么精神！自己的工作做完了，为了全局的利益，主动为别人分担。这被称为 `Work-stealing` ，Go 从 1.1 开始实现。

Go scheduler 使用 M:N 模型，在任一时刻，M 个 goroutines（G） 要分配到 N 个内核线程（M），这些 M 跑在个数最多为 GOMAXPROCS 的逻辑处理器（P）上。每个 M 必须依附于一个 P，每个 P 在同一时刻只能运行一个 M。如果 P 上的 M 阻塞了，那它就需要其他的 M 来运行 P 的 LRQ 里的 goroutines。



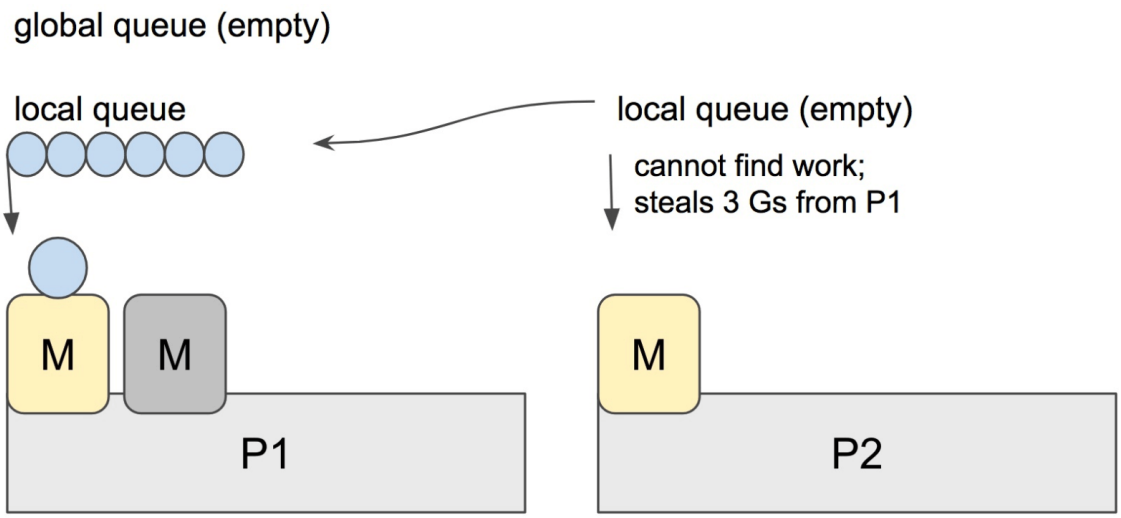

个人感觉，上面这张图比常见的那些用三角形表示 M，圆形表示 G，矩形表示 P 的那些图更生动形象。

实际上，Go scheduler 每一轮调度要做的工作就是找到处于 runnable 的 goroutines，并执行它。找的顺序如下：

```
runtime.schedule() {
    // only 1/61 of the time, check the global runnable queue for a G.
    // if not found, check the local queue.
    // if not found,
    //     try to steal from other Ps.
    //     if not, check the global runnable queue.
    //     if not found, poll network.
}
```

找到一个可执行的 goroutine 后，就会一直执行下去，直到被阻塞。

当 P2 上的一个 G 执行结束，它就会去 LRQ 获取下一个 G 来执行。如果 LRQ 已经空了，就是说本地可运行队列已经没有 G 需要执行，并且这时 GRQ 也没有 G 了。这时，P2 会随机选择一个 P（称为 P1），P2 会从 P1 的 LRQ “偷”过来一半的 G。




这样做的好处是，有更多的 P 可以一起工作，加速执行完所有的 G。

# 描述 scheduler 的初始化过程

上一节我们说完了 GPM 结构体，这一讲，我们来研究 Go sheduler 结构体，以及整个调度器的初始化过程。

Go scheduler 在源码中的结构体为 `schedt` ，保存调度器的状态信息、全局的可运行 G 队列等。源码如下：

```
// 保存调度器的信息
type schedt struct {
    // accessed atomically. keep at top to ensure alignment on 32-bit systems.
    // 需以原子访问访问。
    // 保持在 struct 顶部，以使其在 32 位系统上可以对齐
    goidgen  uint64
    lastpoll uint64

    lock mutex

    // 由空闲的工作线程组成的链表
    midle        muintptr // idle m's waiting for work
    // 空闲的工作线程数量
    nmidle       int32    // number of idle m's waiting for work
    // 空闲的且被 lock 的 m 计数
    nmidlelocked int32    // number of locked m's waiting for work
    // 已经创建的工作线程数量
    mcount       int32    // number of m's that have been created
    // 表示最多所能创建的工作线程数量
    maxmcount    int32    // maximum number of m's allowed (or die)

    // goroutine 的数量，自动更新
    ngsys uint32 // number of system goroutines; updated atomically

    // 由空闲的 p 结构体对象组成的链表
    pidle      puintptr // idle p's
    // 空闲的 p 结构体对象的数量
    npidle     uint32
    nmspinning uint32 // See "Worker thread parking/unparking" comment in proc.go.

    // Global runnable queue.
    // 全局可运行的 G队列
    runqhead guintptr // 队列头
    runqtail guintptr // 队列尾
    runqsize int32 // 元素数量

    // Global cache of dead G's.
    // dead G 的全局缓存
    // 已退出的 goroutine 对象，缓存下来
    // 避免每次创建 goroutine 时都重新分配内存
    gflock       mutex
    gfreeStack   *g
    gfreeNoStack *g
    // 空闲 g 的数量
    ngfree       int32

    // Central cache of sudog structs.
    // sudog 结构的集中缓存
    sudoglock  mutex
    sudogcache *sudog

    // Central pool of available defer structs of different sizes.
    // 不同大小的可用的 defer struct 的集中缓存池
```

```
    deferlock mutex
    deferpool [5]*_defer

    gcwaiting  uint32 // gc is waiting to run
    stopwait   int32
    stopnote   note
    sysmonwait uint32
    sysmonnote note

    // safepointFn should be called on each P at the next GC
    // safepoint if p.runSafePointFn is set.
    safePointFn   func(*p)
    safePointWait int32
    safePointNote note

    profilehz int32 // cpu profiling rate

    // 上次修改 gomaxprocs 的纳秒时间
    procresizetime int64 // nanotime() of last change to gomaxprocs
    totaltime      int64 // ∫gomaxprocs dt up to procresizetime
}
```

在程序运行过程中， `schedt` 对象只有一份实体，它维护了调度器的所有信息。

在 proc.go 和 runtime2.go 文件中，有一些很重要全局的变量，我们先列出来：

```
// 所有 g 的长度
allglen    uintptr

// 保存所有的 g
allgs    []*g

// 保存所有的 m
allm       *m

// 保存所有的 p，_MaxGomaxprocs = 1024
allp       [_MaxGomaxprocs + 1]*p

// p 的最大值，默认等于 ncpu
gomaxprocs  int32

// 程序启动时，会调用 osinit 函数获得此值
ncpu       int32

// 调度器结构体对象，记录了调度器的工作状态
sched      schedt

// 代表进程的主线程
m0          m

// m0 的 g0，即 m0.g0 = &g0
g0          g
```

> 在程序初始化时，这些全局变量都会被初始化为零值：指针被初始化为 nil 指针，切片被初始化为 nil 切片，int 被初始化为 0，结构体的所有成员变量按其类型被初始化为对应的零值。

> 因此程序刚启动时 allgs，allm 和allp 都不包含任何 g，m 和 p。

不仅是 Go 程序，系统加载可执行文件大概都会经过这几个阶段：

> 1. 从磁盘上读取可执行文件，加载到内存
> 2. 创建进程和主线程
> 3. 为主线程分配栈空间
> 4. 把由用户在命令行输入的参数拷贝到主线程的栈
> 5. 把主线程放入操作系统的运行队列等待被调度

上面这段描述，来自公众号“ go语言核心编程技术”的调度系列教程。

我们从一个 `Hello World` 的例子来回顾一下 Go 程序初始化的过程：

```
package main

import "fmt"

func main() {
    fmt.Println("hello world")
}
```

在项目根目录下执行：

```
go build -gcflags "-N -l" -o hello src/main.go
```

`-gcflags "-N -l"` 是为了关闭编译器优化和函数内联，防止后面在设置断点的时候找不到相对应的代码位置。

得到了可执行文件 `hello` ，执行：

```
[qcrao@qcrao hello-world]$ gdb hello
```

进入 gdb 调试模式，执行 `info files` ，得到可执行文件的文件头，列出了各种段：

```
(gdb) info files
Symbols from "/home/qcrao/hello-world/hello".
Local exec file:
        `/home/qcrao/hello-world/hello', file type elf64-x86-64.
        Entry point: 0x450e20
        0x0000000000401000 - 0x000000000049cded is .text
        0x000000000049ce00 - 0x000000000049cfe0 is .plt
        0x000000000049d000 - 0x00000000004e8c6e is .rodata
        0x00000000004e9440 - 0x00000000004e9770 is .dynsym
        0x00000000004e8c70 - 0x00000000004e8c88 is .rela
        0x00000000004e8c88 - 0x00000000004e8f40 is .rela.plt
        0x00000000004e8f40 - 0x00000000004e8f84 is .gnu.version
        0x00000000004e8fa0 - 0x00000000004e8ff0 is .gnu.version_r
        0x00000000004e9000 - 0x00000000004e90ac is .hash
        0x00000000004e9240 - 0x00000000004e942d is .dynstr
        0x00000000004e9780 - 0x00000000004ea4ac is .typelink
        0x00000000004ea4b0 - 0x00000000004ea5b0 is .itablink
        0x00000000004ea5b0 - 0x00000000004ea5b0 is .gosymtab
        0x00000000004ea5c0 - 0x0000000000544fa5 is .gopclntab
        0x0000000000545000 - 0x0000000000545100 is .got.plt
        0x0000000000545100 - 0x0000000000545230 is .dynamic
        0x0000000000545230 - 0x0000000000545238 is .got
        0x0000000000545240 - 0x00000000005531b8 is .noptrdata
        0x00000000005531c0 - 0x000000000055a070 is .data
        0x000000000055a080 - 0x0000000000576968 is .bss
        0x0000000000576980 - 0x0000000000579118 is .noptrbss
        0x0000000000000000 - 0x0000000000000008 is .tbss
        0x0000000000400fe4 - 0x0000000000401000 is .interp
        0x0000000000400f80 - 0x0000000000400fe4 is .note.go.buildid
```

同时，我们也得到了入口地址：0x450e20。

```
(gdb) b *0x450e20
Breakpoint 1 at 0x450e20: file /usr/local/go/src/runtime/rt0_linux_amd64.s, line 8.
```

这就是 Go 程序的入口地址，我是在 linux 上运行的，所以入口文件为 `src/runtime/rt0_linux_amd64.s` ，runtime 目录下有各种不同名称的程序入口文件，支持各种操作系统和架构，代码为：

```
TEXT _rt0_amd64_linux(SB),NOSPLIT,$-8
    LEAQ    8(SP), SI // argv
    MOVQ    0(SP), DI // argc
    MOVQ    $main(SB), AX
    JMP     AX
```

主要是把 argc，argv 从内存拉到了寄存器。这里 LEAQ 是计算内存地址，然后把内存地址本身放进寄存器里，也就是把 argv 的地址放到了 SI 寄存器中。最后跳转到：

```
TEXT main(SB),NOSPLIT,$-8
    MOVQ    $runtime·rt0_go(SB), AX
    JMP     AX
```

继续跳转到 `runtime·rt0_go(SB)` ，完成 go 启动时所有的初始化工作。位于 `/usr/local/go/src/runtime/asm_amd64.s` ，代码：

```
TEXT runtime·rt0_go(SB),NOSPLIT,$0
    // copy arguments forward on an even stack
    MOVQ    DI, AX      // argc
    MOVQ    SI, BX      // argv
    SUBQ    $(4*8+7), SP        // 2args 2auto
    // 调整栈顶寄存器使其按 16 字节对齐
    ANDQ    $~15, SP
    // argc 放在 SP+16 字节处
    MOVQ    AX, 16(SP)
    // argv 放在 SP+24 字节处
    MOVQ    BX, 24(SP)

    // create istack out of the given (operating system) stack.
    // _cgo_init may update stackguard.
    // 给 g0 分配栈空间

    // 把 g0 的地址存入 DI
    MOVQ    $runtime·g0(SB), DI
    // BX = SP - 64*1024 + 104
    LEAQ    (-64*1024+104)(SP), BX
    // g0.stackguard0 = SP - 64*1024 + 104
    MOVQ    BX, g_stackguard0(DI)
    // g0.stackguard1 = SP - 64*1024 + 104
    MOVQ    BX, g_stackguard1(DI)
    // g0.stack.lo = SP - 64*1024 + 104
    MOVQ    BX, (g_stack+stack_lo)(DI)
    // g0.stack.hi = SP
    MOVQ    SP, (g_stack+stack_hi)(DI)

    // ……................
    // 省略了很多检测 CPU 信息的代码
    // ……................


    // 初始化 m 的 tls
    // DI = &m0.tls，取 m0 的 tls 成员的地址到 DI 寄存器
    LEAQ    runtime·m0+m_tls(SB), DI
    // 调用 settls 设置线程本地存储，settls 函数的参数在 DI 寄存器中
    // 之后，可通过 fs 段寄存器找到 m.tls
    CALL    runtime·settls(SB)

    // store through it, to make sure it works
    // 获取 fs 段基址并放入 BX 寄存器，其实就是 m0.tls[1] 的地址，get_tls 的代码由编译器生成
    get_tls(BX)
    MOVQ    $0x123, g(BX)
    MOVQ    runtime·m0+m_tls(SB), AX
    CMPQ    AX, $0x123
    JEQ 2(PC)
    MOVL    AX, 0    // abort
ok:
    // set the per-goroutine and per-mach "registers"
    // 获取 fs 段基址到 BX 寄存器
    get_tls(BX)
    // 将 g0 的地址存储到 CX，CX = &g0
    LEAQ    runtime·g0(SB), CX
    // 把 g0 的地址保存在线程本地存储里面，也就是 m0.tls[0]=&g0
    MOVQ    CX, g(BX)
    // 将 m0 的地址存储到 AX，AX = &m0
```

```
    LEAQ    runtime·m0(SB), AX

    // save m->g0 = g0
    // m0.g0 = &g0
    MOVQ    CX, m_g0(AX)
    // save m0 to g0->m
    // g0.m = &m0
    MOVQ    AX, g_m(CX)

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

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

    // 初始化系统核心数
    CALL    runtime·osinit(SB)
    // 调度器初始化
    CALL    runtime·schedinit(SB)

    // create a new goroutine to start program
    MOVQ    $runtime·mainPC(SB), AX     // entry
    // newproc 的第二个参数入栈，也就是新的 goroutine 需要执行的函数
    // AX = &funcval{runtime·main},
    PUSHQ   AX
    // newproc 的第一个参数入栈，该参数表示 runtime.main 函数需要的参数大小，
    // 因为 runtime.main 没有参数，所以这里是 0
    PUSHQ   $0          // arg size
    // 创建 main goroutine
    CALL    runtime·newproc(SB)
    POPQ    AX
    POPQ    AX

    // start this M
    // 主线程进入调度循环，运行刚刚创建的 goroutine
    CALL    runtime·mstart(SB)

    // 永远不会返回，万一返回了，crash 掉
    MOVL    $0xf1, 0xf1  // crash
    RET
```

这段代码完成之后，整个 Go 程序就可以跑起来了，是非常核心的代码。这一讲其实只讲到了第 80 行，也就是调度器初始化函数：

```
CALL    runtime·schedinit(SB)
```

`schedinit` 函数返回后，调度器的相关参数都已经初始化好了，犹如盘古开天辟地，万事万物各就其位。接下来详细解释上面的汇编代码。

# 调整 SP

第一段代码，将 SP 调整到了一个地址是 16 的倍数的位置：

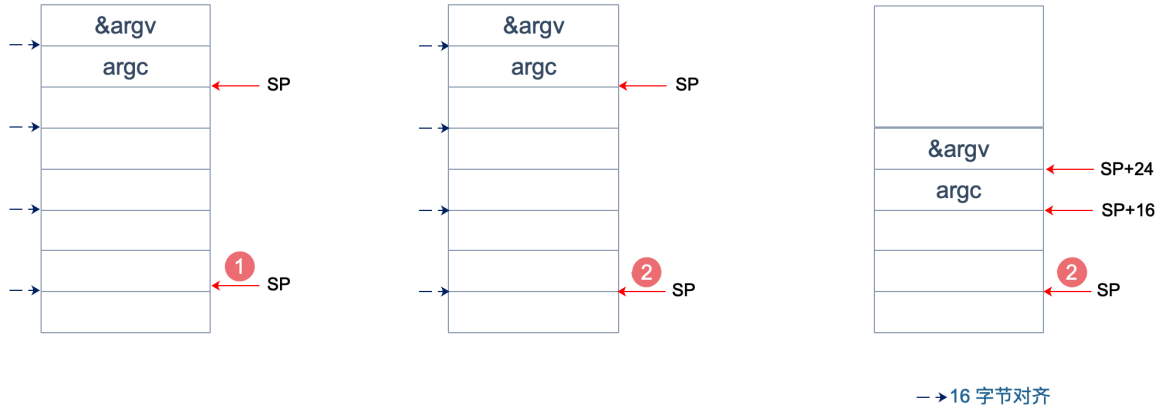
```
SUBQ    $(4*8+7), SP        // 2args 2auto
// 调整栈顶寄存器使其按 16 个字节对齐
```

```
ANDQ    $~15, SP
```

先是将 SP 减掉 39，也就是向下移动了 39 个 Byte，再进行与运算。

`15` 的二进制低四位是全 1：`1111`，其他位都是 0；取反后，变成了 `0000`，高位则是全 1。这样，与 SP 进行了与运算后，低 4 位变成了全 0，高位则不变。因此 SP 继续向下移动，并且这回是在一个地址值为 16 的倍数的地方，16 字节对齐的地方。

为什么要这么做？画一张图就明白了。不过先得说明一点，前面 `_rt0_amd64_linux` 函数里讲过，DI 里存的是 argc 的值，8 个字节，而 SI 里则存的是 argv 的地址，8 个字节。




上面两张图中，左侧用箭头标注了 16 字节对齐的位置。第一步表示向下移动 39 B，第二步表示与 `~15` 相与。

存在两种情况，这也是第一步将 SP 下移的时候，多移了 7 个 Byte 的原因。第一张图里，与 `~15` 相与的时候，SP 值减少了 1，第二张图则减少了 9。最后都是移位到了 16 字节对齐的位置。

两张图的共同点是 SP 与 argc 中间多出了 16 个字节的空位。这个后面应该会用到，我们接着探索。

至于为什么进行 16 个字节对齐，就比较好理解了：因为 CPU 有一组 SSE 指令，这些指令中出现的内存地址必须是 16 的倍数。

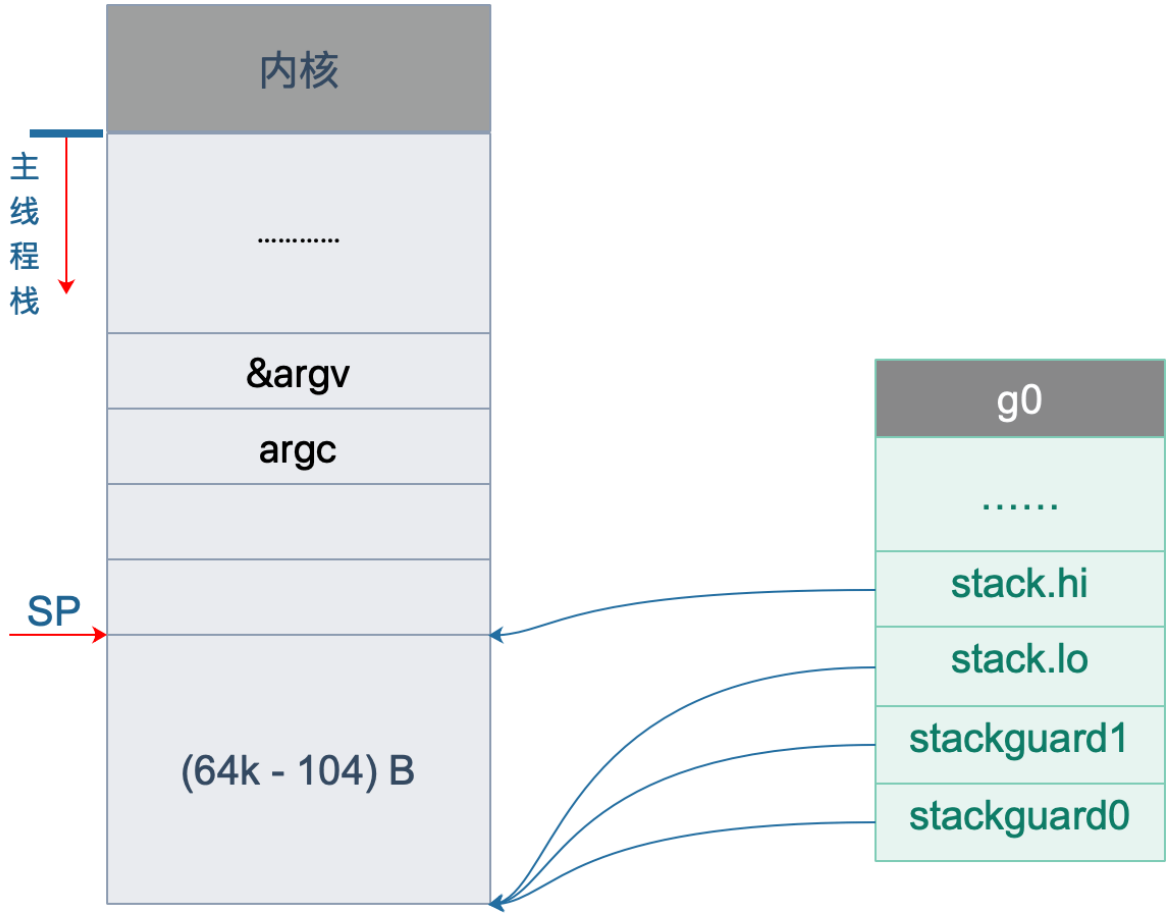
# 初始化 g0 栈

接着往后看，开始初始化 g0 的栈了。g0 栈的作用就是为运行 runtime 代码提供一个“环境”。

```
// 把 g0 的地址存入 DI
MOVQ    $runtime·g0(SB), DI
// BX = SP - 64*1024 + 104
LEAQ    (-64*1024+104)(SP), BX
// g0.stackguard0 = SP - 64*1024 + 104
MOVQ    BX, g_stackguard0(DI)
// g0.stackguard1 = SP - 64*1024 + 104
MOVQ    BX, g_stackguard1(DI)
// g0.stack.lo = SP - 64*1024 + 104
MOVQ    BX, (g_stack+stack_lo)(DI)
// g0.stack.hi = SP
MOVQ    SP, (g_stack+stack_hi)(DI)
```

代码 L2 把 g0 的地址存入 DI 寄存器；L4 将 SP 下移 (64K-104)B，并将地址存入 BX 寄存器；L6 将 BX 里存储的地址赋给 g0.stackguard0；L8，L10，L12 分别 将 BX 里存储的地址赋给 g0.stackguard1， g0.stack.lo， g0.stack.hi。

这部分完成之后，g0 栈空间如下图：




## 主线程绑定 m0

接着往下看，中间我们省略了很多检查 CPU 相关的代码，直接看主线程绑定 m0 的部分：

```
// 初始化 m 的 tls
// DI = &m0.tls，取 m0 的 tls 成员的地址到 DI 寄存器
LEAQ    runtime·m0+m_tls(SB), DI
// 调用 settls 设置线程本地存储，settls 函数的参数在 DI 寄存器中
// 之后，可通过 fs 段寄存器找到 m.tls
CALL    runtime·settls(SB)

// store through it, to make sure it works
// 获取 fs 段基地址并放入 BX 寄存器，其实就是 m0.tls[1] 的地址，get_tls 的代码由编译器生成
get_tls(BX)
MOVQ    $0x123, g(BX)
MOVQ    runtime·m0+m_tls(SB), AX
CMPQ    AX, $0x123
JEQ 2(PC)
MOVL    AX, 0    // abort
```

因为 m0 是全局变量，而 m0 又要绑定到工作线程才能执行。我们又知道，runtime 会启动多个工作线程，每个线程都会绑定一个 m0。而且，代码里还得保持一致，都是用 m0 来表示。这就要用到线程本地存储的知识了，也就是常说的 TLS（Thread Local Storage）。简单来说，TLS 就是线程本地的私有的全局变量。

一般而言，全局变量对进程中的多个线程同时可见。进程中的全局变量与函数内定义的静态（static）变量，是各个线程都可以访问的共享变量。一个线程修改了，其他线程就会“看见”。要想搞出一个线程私有的变量，就需要用到 TLS 技术。

如果需要在一个线程内部的各个函数调用都能访问、但其它线程不能访问的变量（被称为 static memory local to a thread，线程局部静态变量），就需要新的机制来实现。这就是 TLS。

继续来看源码，L3 将 m0.tls 地址存储到 DI 寄存器，再调用 settls 完成 tls 的设置，tls 是 m 结构体中的一个数组。

```
// thread-local storage (for x86 extern register)
tls [6]uintptr
```

设置 tls 的函数 runtime·settls(SB) 位于源码 `src/runtime/sys_linux_amd64.s` 处，主要内容就是通过一个系统调用将 fs 段基址设置成 m.tls[1] 的地址，而 fs 段基址又可以通过 CPU 里的寄存器 fs 来获取。

> 而每个线程都有自己的一组 CPU 寄存器值，操作系统在把线程调离 CPU 时会帮我们把所有寄存器中的值保存在内存中，调度线程来运行时又会从内存中把这些寄存器的值恢复到 CPU。

这样，工作线程代码就可以通过 fs 寄存器来找到 m.tls。

关于 settls 这个函数的解析可以去看阿波张的教程第 12 篇，写得很详细。

设置完 tls 之后，又来了一段验证上面 settls 是否能正常工作。如果不能，会直接 crash。

```
get_tls(BX)
MOVQ    $0x123, g(BX)
MOVQ    runtime·m0+m_tls(SB), AX
CMPQ    AX, $0x123
JEQ 2(PC)
MOVL    AX, 0    // abort
```

第一行代码，获取 tls，`get_tls(BX)` 的代码由编译器生成，源码中并没有看到，可以理解为将 `m.tls` 的地址存入 BX 寄存器。

L2 将一个数 `0x123` 放入 `m.tls[0]` 处，L3 则将 `m.tls[0]` 处的数据取出来放到 AX 寄存器，L4 则比较两者是否相等。如果相等，则跳过 L6 行的代码，否则执行 L6，程序 crash。

继续看代码：

```
// set the per-goroutine and per-mach "registers"
// 获取 fs 段基址到 BX 寄存器
get_tls(BX)
// 将 g0 的地址存储到 CX，CX = &g0
LEAQ    runtime·g0(SB), CX
// 把 g0 的地址保存在线程本地存储里面，也就是 m0.tls[0]=&g0
MOVQ    CX, g(BX)
// 将 m0 的地址存储到 AX，AX = &m0
LEAQ    runtime·m0(SB), AX

// save m->g0 = g0
// m0.g0 = &g0
MOVQ    CX, m_g0(AX)
// save m0 to g0->m
// g0.m = &m0
MOVQ    AX, g_m(CX)
```

L3 将 m.tls 地址存入 BX；L5 将 g0 的地址存入 CX；L7 将 CX，也就是 g0 的地址存入 m.tls[0]；L9 将 m0 的地址存入 AX；L13 将 g0 的地址存入 m0.g0；L16 将 m0 存入 g0.m。也就是：

```
tls[0] = g0
m0.g0 = &g0
g0.m = &m0
```

代码中寄存器前面的符号看着比较奇怪，其实它们最后会被链接器转化为偏移量。

看曹大 golang_notes 用 gobuf_sp(BX) 这个例子讲的：

> 这种写法在标准 plan9 汇编中只是个 symbol，没有任何偏移量的意思，但这里却用名字来代替了其偏移量，这是怎么回事呢？

> 实际上这是 runtime 的特权，是需要链接器配合完成的，再来看看 gobuf 在 runtime 中的 struct 定义开头部分的注释：

> // The offsets of sp, pc, and g are known to (hard-coded in) libmach.

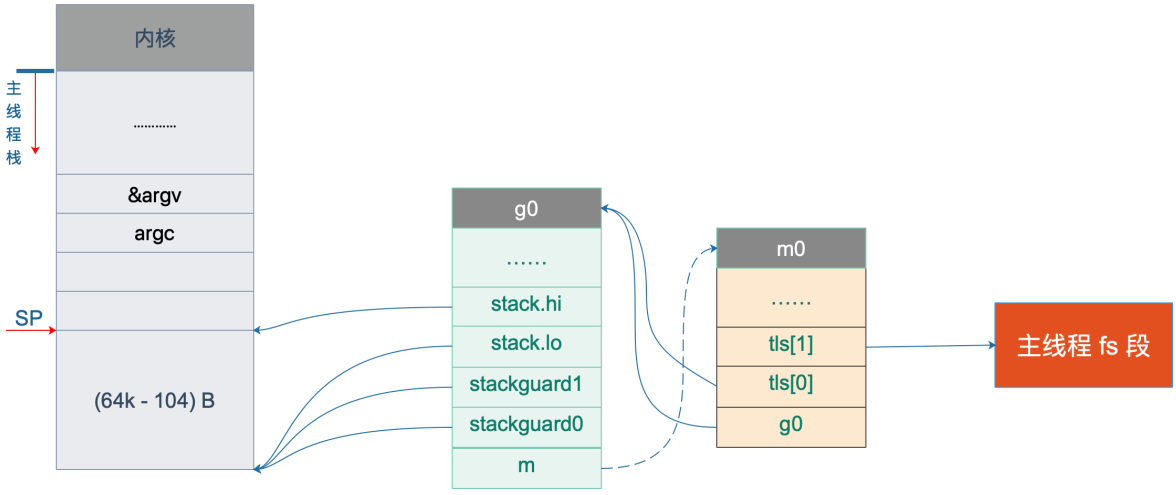
对于我们而言，这种写法读起来比较容易。

这一段执行完之后，就把 m0，g0，m.tls[0] 串联起来了。通过 m.tls[0] 可以找到 g0，通过 g0 可以找到 m0（通过 g 结构体的 m 字段）。并且，通过 m 的字段 g0，m0 也可以找到 g0。于是，主线程和 m0，g0 就关联起来了。

> 从这里还可以看到，保存在主线程本地存储中的值是 g0 的地址，也就是说工作线程的私有全局变量其实是一个指向 g 的指针而不是指向 m 的指针。

> 目前这个指针指向g0，表示代码正运行在 g0 栈。

于是，前面的图又增加了新的玩伴 m0：




# 初始化 m0

```
MOVL    16(SP), AX      // copy argc
MOVL    AX, 0(SP)
MOVQ    24(SP), AX      // copy argv
MOVQ    AX, 8(SP)
CALL    runtime·args(SB)
// 初始化系统核心数
CALL    runtime·osinit(SB)
```

```
// 调度器初始化
CALL    runtime·schedinit(SB)
```

L1-L2 将 16(SP) 处的内容移动到 0(SP)，也就是栈顶，通过前面的图，16(SP) 处的内容为 argc；L3-L4 将 argv 存入 8(SP)，接下来调用 `runtime·args` 函数，处理命令行参数。

接着，连续调用了两个 runtime 函数。osinit 函数初始化系统核心数，将全局变量 ncpu 初始化的核心数，schedinit 则是本文的核心：调度器的初始化。

下面，我们来重点看 schedinit 函数：

```
// src/runtime/proc.go

// The bootstrap sequence is:
//
//    call osinit
//    call schedinit
//    make & queue new G
//    call runtime·mstart
//
// The new G calls runtime·main.
func schedinit() {
    // getg 由编译器实现
    // get_tls(CX)
    // MOVQ g(CX), BX; BX存器里面现在放的是当前g结构体对象的地址
    _g_ := getg()
    if raceenabled {
        _g_.racectx, raceprocctx0 = raceinit()
    }

    // 最多启动 10000 个工作线程
    sched.maxmcount = 10000

    tracebackinit()
    moduledataverify()

    // 初始化栈空间复用管理链表
    stackinit()
    mallocinit()

    // 初始化 m0
    mcommoninit(_g_.m)
    alginit()       // maps must not be used before this call
    modulesinit()   // provides activeModules
    typelinksinit() // uses maps, activeModules
    itabsinit()     // uses activeModules

    msigsave(_g_.m)
    initSigmask = _g_.m.sigmask

    goargs()
    goenvs()
    parsedebugvars()
    gcinit()

    sched.lastpoll = uint64(nanotime())

    // 初始化 P 的个数
    // 系统中有多少核，就创建和初始化多少个 p 结构体对象
    procs := ncpu
```

```
    if n, ok := atoi32(gogetenv("GOMAXPROCS")); ok && n > 0 {
        procs = n
    }
    if procs > _MaxGomaxprocs {
        procs = _MaxGomaxprocs
    }

    // 初始化所有的 P，正常情况下不会返回有本地任务的 P
    if procresize(procs) != nil {
        throw("unknown runnable goroutine during bootstrap")
    }

    // ......................
}
```

这个函数开头的注释很贴心地把 Go 程序初始化的过程又说了一遍：

1. call osinit。初始化系统核心数。
2. call schedinit。初始化调度器。
3. make & queue new G。创建新的 goroutine。
4. call runtime·mstart。调用 mstart，启动调度。
5. The new G calls runtime·main。在新的 goroutine 上运行 runtime.main 函数。

函数首先调用 `getg()` 函数获取当前正在运行的 `g`，`getg()` 在 `src/runtime/stubs.go` 中声明，真正的代码由编译器生成。

```
// getg returns the pointer to the current g.
// The compiler rewrites calls to this function into instructions
// that fetch the g directly (from TLS or from the dedicated register).
func getg() *g
```

注释里也说了，getg 返回当前正在运行的 goroutine 的指针，它会从 tls 里取出 tls[0]，也就是当前运行的 goroutine 的地址。编译器插入类似下面的代码：

```
get_tls(CX)
MOVQ g(CX), BX; // BX存器里面现在放的是当前g结构体对象的地址
```

继续往下看：

```
sched.maxmcount = 10000
```

设置最多只能创建 10000 个工作线程。

然后，调用了一堆 init 函数，初始化各种配置，现在不去深究。只关心本小节的重点，m0 的初始化：

```
// 初始化 m
func mcommoninit(mp *m) {
    // 初始化过程中_g_ = g0
    _g_ := getg()

    // g0 stack won't make sense for user (and is not necessary unwindable).
    if _g_ != _g_.m.g0 {
        callers(1, mp.createstack[:])
    }

    // random 初始化
    mp.fastrand = 0x49f6428a + uint32(mp.id) + uint32(cputicks())
```

```
    if mp.fastrand == 0 {
        mp.fastrand = 0x49f6428a
    }

    lock(&sched.lock)
    // 设置 m 的 id
    mp.id = sched.mcount
    sched.mcount++
    // 检查已创建系统线程是否超过了数量限制（10000）
    checkmcount()

    // ·····省略了初始化 gsignal

    // Add to allm so garbage collector doesn't free g->m
    // when it is just in a register or thread-local storage.
    mp.alllink = allm

    atomicstorep(unsafe.Pointer(&allm), unsafe.Pointer(mp))
    unlock(&sched.lock)

    // ·····
}
```

因为 sched 是一个全局变量，多个线程同时操作 sched 会有并发问题，因此先要加锁，操作结束之后再解锁。

```
mp.id = sched.mcount
sched.mcount++
checkmcount()
```

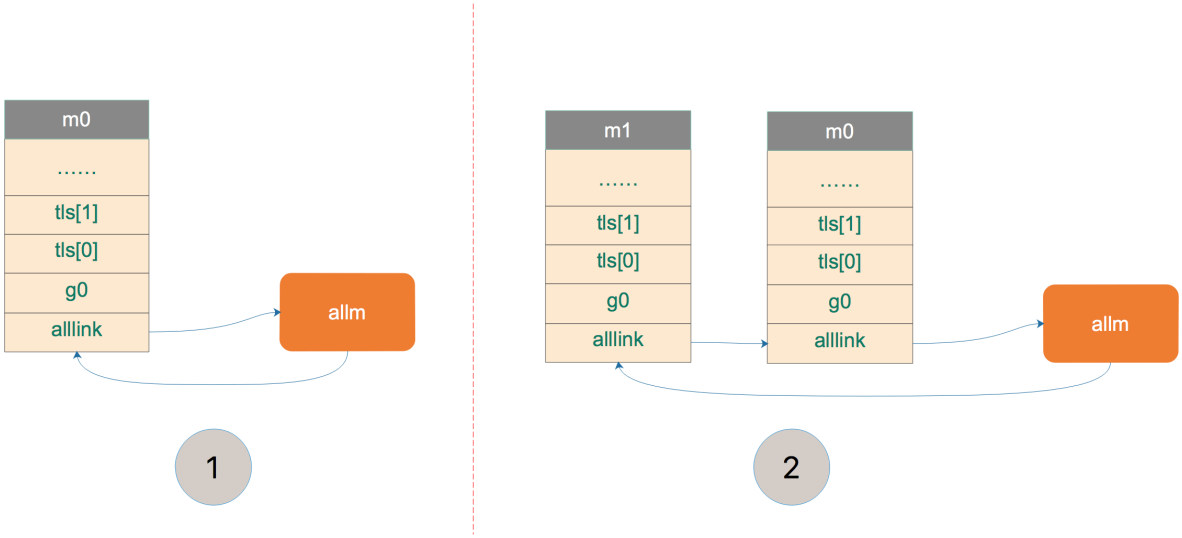
可以看到，m0 的 id 是 0，并且之后创建的 m 的 id 是递增的。 `checkmcount()` 函数检查已创建系统线程是否超过了数量限制（10000）。

```
mp.alllink = allm
```

将 m 挂到全局变量 allm 上，allm 是一个指向 m 的的指针。

```
atomicstorep(unsafe.Pointer(&allm), unsafe.Pointer(mp))
```

这一行将 allm 变成 m 的地址，这样变成了一个循环链表。之后再新建 m 的时候，新 m 的 alllink 就会指向本次的 m，最后 allm 又会指向新创建的 m。




上图中，1 将 m0 挂在 allm 上。之后，若新创建 m，则 m1 会和 m0 相连。

完成这些操作后，大功告成！解锁。

# 初始化 allp

跳过一些其他的初始化代码，继续往后看：

```
procs := ncpu
if n, ok := atoi32(gogetenv("GOMAXPROCS")); ok && n > 0 {
    procs = n
}
if procs > _MaxGomaxprocs {
    procs = _MaxGomaxprocs
}
```

这里就是设置 procs，它决定创建 P 的数量。ncpu 这里已经被赋上了系统的核心数，因此代码里不设置 GOMAXPROCS 也是没问题的。这里还限制了 procs 的最大值，为 1024。

来看最后一个核心的函数：

```
// src/runtime/proc.go

func procresize(nprocs int32) *p {
    old := gomaxprocs
    if old < 0 || old > _MaxGomaxprocs || nprocs <= 0 || nprocs > _MaxGomaxprocs {
        throw("procresize: invalid arg")
    }

    // .....................

    // update statistics
    // 更新数据
    now := nanotime()
    if sched.procresizetime != 0 {
        sched.totaltime += int64(old) * (now - sched.procresizetime)
    }
    sched.procresizetime = now
```

```
    // 初始化所有的 P
    for i := int32(0); i < nprocs; i++ {
        pp := allp[i]
        if pp == nil {
            // 申请新对象
            pp = new(p)
            pp.id = i
            // pp 的初始状态为 stop
            pp.status = _Pgcstop
            pp.sudogcache = pp.sudogbuf[:0]
            for i := range pp.deferpool {
                pp.deferpool[i] = pp.deferpoolbuf[i][:0]
            }
            // 将 pp 存放到 allp 处
            atomicstorep(unsafe.Pointer(&allp[i]), unsafe.Pointer(pp))
        }

        // ......................

    }

    // 释放多余的 P。由于减少了旧的 procs 的数量，因此需要释放
    // ......................

    // 获取当前正在运行的 g 指针，初始化时 _g_ = g0
    _g_ := getg()
    if _g_.m.p != 0 && _g_.m.p.ptr().id < nprocs {
        // continue to use the current P
        // 继续使用当前 P
        _g_.m.p.ptr().status = _Prunning
    } else {
        // 初始化时执行这个分支

        // ......................

        _g_.m.p = 0
        _g_.m.mcache = nil
        // 取出第 0 号 p
        p := allp[0]
        p.m = 0
        p.status = _Pidle
        // 将 p0 和 m0 关联起来
        acquirep(p)
        if trace.enabled {
            traceGoStart()
        }
    }
    var runnablePs *p
    // 下面这个 for 循环把所有空闲的 p 放入空闲链表
    for i := nprocs - 1; i >= 0; i-- {
        p := allp[i]
        // allp[0] 跟 m0 关联了，不会进行之后的“放入空闲链表”
        if _g_.m.p.ptr() == p {
            continue
        }

        // 状态转为 idle
        p.status = _Pidle
        // p 的 LRQ 里没有 G
        if runqempty(p) {
            // 放入全局空闲链表
            pidleput(p)
```

```
        } else {
            p.m.set(mget())
            p.link.set(runnablePs)
            runnablePs = p
        }
    }
    stealOrder.reset(uint32(nprocs))
    var int32p *int32 = &gomaxprocs // make compiler check that gomaxprocs is an int32
    atomic.Store((*uint32)(unsafe.Pointer(int32p)), uint32(nprocs))
    // 返回有本地任务的 P 链表
    return runnablePs
}
```

代码比较长，这个函数不仅是初始化的时候会执行到，在中途改变 procs 的值的时候，仍然会调用它。所有存在很多一般不用关心的代码，因为一般不会在中途重新设置 procs 的值。我把初始化无关的代码删掉了，这样会更清晰一些。

函数先是从堆上创建了 nproc 个 P，并且把 P 的状态设置为 `_Pgcstop` ，现在全局变量 allp 里就维护了所有的 P。

接着，调用函数 `acquirep` 将 p0 和 m0 关联起来。我们来详细看一下：

```
func acquirep(_p_ *p) {
    // Do the part that isn't allowed to have write barriers.
    acquirep1(_p_)

    // have p; write barriers now allowed
    _g_ := getg()
    _g_.m.mcache = _p_.mcache

    // ·····················
}
```

先调用 `acquirep1` 函数真正地进行关联，之后，将 p0 的 mcache 资源赋给 m0。再来看 `acquirep1` ：

```
func acquirep1(_p_ *p) {
    _g_ := getg()

    // ·····················

    _g_.m.p.set(_p_)
    _p_.m.set(_g_.m)
    _p_.status = _Prunning
}
```

可以看到就是一些字段相互设置，执行完成后：

```
g0.m.p = p0
p0.m = m0
```

并且，p0 的状态变成了 `_Prunning` 。

接下来是一个循环，它将除了 p0 的所有非空闲的 P，放入 P 链表 runnablePs，并返回给 procresize 函数的调用者，并由调用者来“调度”这些 P。

函数 `runqempty` 用来判断一个 P 是否是空闲，依据是 P 的本地 run queue 队列里有没有 runnable 的 G，如果没有，那 P 就是空闲的。

```
// src/runtime/proc.go

// 如果 _p_ 的本地队列里没有待运行的 G，则返回 true
func runqempty(_p_ *p) bool {
    // 这里涉及到一些数据竞争，并不是简单地判断 runqhead == runqtail 并且 runqnext == nil 就可以
    //
    for {
        head := atomic.Load(&_p_.runqhead)
        tail := atomic.Load(&_p_.runqtail)
        runnext := atomic.Loaduintptr((*uintptr)(unsafe.Pointer(&_p_.runnext)))
        if tail == atomic.Load(&_p_.runqtail) {
            return head == tail && runnext == 0
        }
    }
}
```

并不是简单地判断 head == tail 并且 runnext == nil 为真，就可以说明 runq 是空的。因为涉及到一些数据竞争，例如在比较 head == tail 时为真，但此时 runnext 上其实有一个 G，之后再去比较 runnext == nil 的时候，这个 G 又通过 runqput跑到了 runq 里去了或者通过 runqget 拿走了，runnext 也为真，于是函数就判断这个 P 是空闲的，这就会形成误判。

因此 runqempty 函数先是通过原子操作取出了 head，tail，runnext，然后再次确认 tail 没有发生变化，最后再比较 head == tail 以及 runnext == nil，保证了在观察三者都是在“同时”观察到的，因此，返回的结果就是正确的。

说明一下，runnext 上有时会绑定一个 G，这个 G 是被当前 G 唤醒的，相比其他 G 有更高的执行优先级，因此把它单独拿出来。

函数的最后，初始化了一个“随机分配器”：

```
stealOrder.reset(uint32(nprocs))
```

将来有些 m 去偷工作的时候，会遍历所有的 P，这时为了偷地随机一些，就会用到 stealOrder 来返回一个随机选择的 P，后面的文章会再讲。

这样，整个 procresize 函数就讲完了，这也意味着，调度器的初始化工作已经完成了。

还是引用阿波张公号文章里的总结，写得太好了，很简洁，很难再优化了：

> 1. 使用 make([]*p, nprocs) 初始化全局变量 allp，即 allp = make([]p, nprocs)

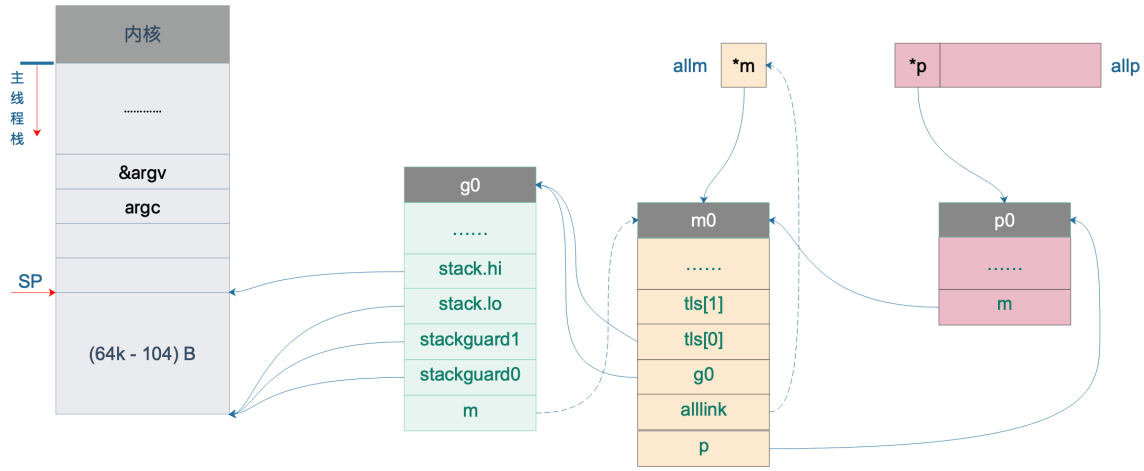
> 2. 循环创建并初始化 nprocs 个 p 结构体对象并依次保存在 allp 切片之中

> 3. 把 m0 和 allp[0] 绑定在一起，即 m0.p = allp[0]，allp[0].m = m0

> 4. 把除了 allp[0] 之外的所有 p 放入到全局变量 sched 的 pidle 空闲队列之中

说明一下，最后一步，代码里是将所有空闲的 P 放入到调度器的全局空闲队列；对于非空闲的 P（本地队列里有 G 待执行），则是生成一个 P 链表，返回给 procresize 函数的调用者。

最后我们将 allp 和 allm 都添加到图上：




# 参考资料

【阿波张 goroutine 调度器初始化】https://mp.weixin.qq.com/s/W9D4Sl-6jYfcpczzdPfByQ

# 编译和链接

# Go 程序启动过程是怎样的

我们从一个 `Hello World` 的例子开始：

```
package main

import "fmt"

func main() {
    fmt.Println("hello world")
}
```

在项目根目录下执行：

```
go build -gcflags "-N -l" -o hello src/main.go
```

`-gcflags "-N -l"` 是为了关闭编译器优化和函数内联，防止后面在设置断点的时候找不到相对应的代码位置。

得到了可执行文件 hello，执行：

```
[qcrao@qcrao hello-world]$ gdb hello
```

进入 gdb 调试模式，执行 `info files`，得到可执行文件的文件头，列出了各种段：

```
The commands are:

        build       compile packages and dependencies
        clean       remove object files
        doc         show documentation for package or symbol
        env         print Go environment information
        bug         start a bug report
        fix         run go tool fix on packages
        fmt         run gofmt on package sources
        generate    generate Go files by processing source
        get         download and install packages and dependencies
        install     compile and install packages and dependencies
        list        list packages
        run         compile and run Go program
        test        test packages
        tool        run specified go tool
        version     print Go version
        vet         run go tool vet on packages
```

同时，我们也得到了入口地址：0x450e20。

```
(gdb) b *0x450e20
Breakpoint 1 at 0x450e20: file /usr/local/go/src/runtime/rt0_linux_amd64.s, line 8.
```

这就是 Go 程序的入口地址，我是在 linux 上运行的，所以入口文件为 `src/runtime/rt0_linux_amd64.s`，runtime 目录下有各种不同名称的程序入口文件，支持各种操作系统和架构，代码为：

```
TEXT _rt0_amd64_linux(SB),NOSPLIT,$-8
    LEAQ    8(SP), SI // argv
    MOVQ    0(SP), DI // argc
    MOVQ    $main(SB), AX
    JMP     AX
```

主要是把 argc，argv 从内存拉到了寄存器。这里 LEAQ 是计算内存地址，然后把内存地址本身放进寄存器里，也就是把 argv 的地址放到了 SI 寄存器中。最后跳转到：

```
TEXT main(SB),NOSPLIT,$-8
    MOVQ    $runtime·rt0_go(SB), AX
    JMP     AX
```

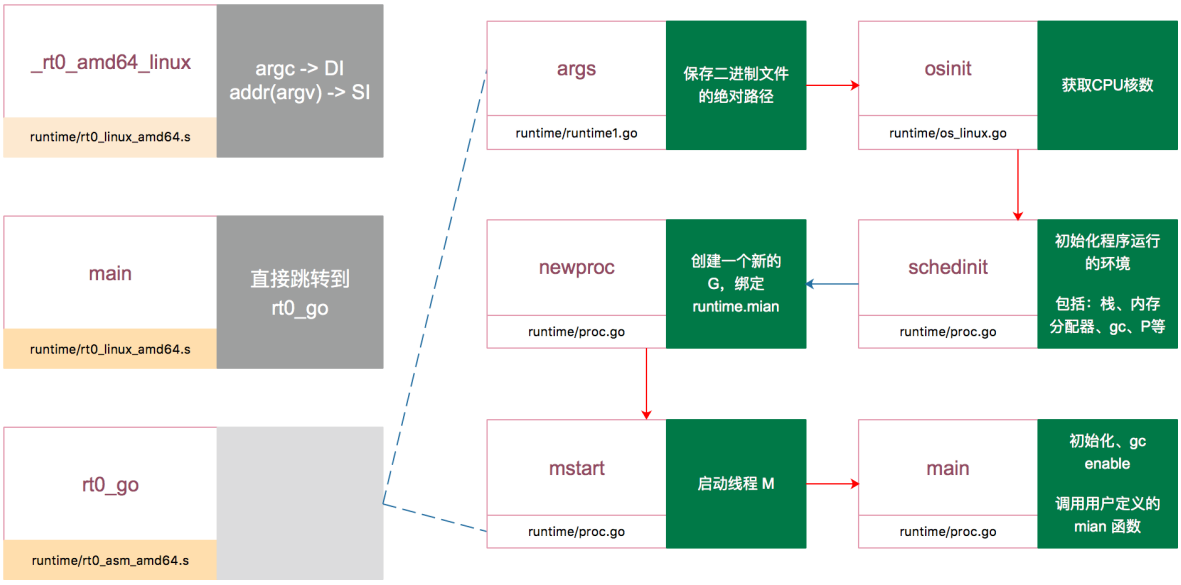
继续跳转到 `runtime·rt0_go(SB)` ，位置： `/usr/local/go/src/runtime/asm_amd64.s` ，代码：

```
TEXT runtime·rt0_go(SB),NOSPLIT,$0
    // 省略很多 CPU 相关的特性标志位检查的代码
    // 主要是看不懂，^_^

    // ..................................

    // 下面是最后调用的一些函数，比较重要
    // 初始化执行文件的绝对路径
    CALL    runtime·args(SB)
    // 初始化 CPU 个数和内存页大小
    CALL    runtime·osinit(SB)
    // 初始化命令行参数、环境变量、gc、栈空间、内存管理、所有 P 实例、HASH算法等
    CALL    runtime·schedinit(SB)

    // 要在 main goroutine 上运行的函数
    MOVQ    $runtime·mainPC(SB), AX         // entry
    PUSHQ    AX
    PUSHQ    $0             // arg size

    // 新建一个 goroutine，该 goroutine 绑定 runtime.main，放在 P 的本地队列，等待调度
    CALL    runtime·newproc(SB)
    POPQ    AX
    POPQ    AX

    // 启动M，开始调度goroutine
    CALL    runtime·mstart(SB)

    MOVL    $0xf1, 0xf1  // crash
    RET


DATA    runtime·mainPC+0(SB)/8,$runtime·main(SB)
GLOBL    runtime·mainPC(SB),RODATA,$8
```

参考文献里的一篇文章【探索 golang 程序启动过程】研究得比较深入，总结下：

> 1. 检查运行平台的CPU，设置好程序运行需要相关标志。
> 2. TLS的初始化。
> 3. runtime.args、runtime.osinit、runtime.schedinit 三个方法做好程序运行需要的各种变量与调度器。
> 4. runtime.newproc创建新的goroutine用于绑定用户写的main方法。
> 5. runtime.mstart开始goroutine的调度。

最后用一张图来总结 go bootstrap 过程吧：




main 函数里执行的一些重要的操作包括：新建一个线程执行 sysmon 函数，定期垃圾回收和调度抢占；启动 gc；执行所有的 init 函数等等。

上面是启动过程，看一下退出过程：

> 当 main 函数执行结束之后，会执行 exit(0) 来退出进程。若执行 exit(0) 后，进程没有退出，main 函数最后的代码会一直访问非法地址：

```
exit(0)
for {
    var x *int32
    *x = 0
}
```

> 正常情况下，一旦出现非法地址访问，系统会把进程杀死，用这样的方法确保进程退出。

关于程序退出这一段的阐述来自群聊《golang runtime 阅读》，又是一个高阶的读源码的组织，github 主页见参考资料。

当然 Go 程序启动这一部分其实还会涉及到 fork 一个新进程、装载可执行文件，控制权转移等问题。还是推荐看前面的两本书，我觉得我不会写得更好，就不叙述了。

# Go 编译相关的命令详解

直接在终端执行：

```
go
```

就能得到和 go 相关的命令简介：

```
The commands are:

        build       compile packages and dependencies
        clean       remove object files
        doc         show documentation for package or symbol
        env         print Go environment information
        bug         start a bug report
        fix         run go tool fix on packages
        fmt         run gofmt on package sources
        generate    generate Go files by processing source
        get         download and install packages and dependencies
        install     compile and install packages and dependencies
        list        list packages
        run         compile and run Go program
        test        test packages
        tool        run specified go tool
        version     print Go version
        vet         run go tool vet on packages
```

和编译相关的命令主要是：

```
go build
go install
go run
```

## go build

`go build` 用来编译指定 packages 里的源码文件以及它们的依赖包，编译的时候会到 `$GoPath/src/package` 路径下寻找源码文件。 `go build` 还可以直接编译指定的源码文件，并且可以同时指定多个。

通过执行 `go help build` 命令得到 `go build` 的使用方法：

```
usage: go build [-o output] [-i] [build flags] [packages]
```

`-o` 只能在编译单个包的时候出现，它指定输出的可执行文件的名字。

`-i` 会安装编译目标所依赖的包，安装是指生成与代码包相对应的 `.a` 文件，即静态库文件（后面要参与链接），并且放置到当前工作区的 pkg 目录下，且库文件的目录层级和源码层级一致。

至于 build flags 参数， `build, clean, get, install, list, run, test` 这些命令会共用一套：

| 参数 | 作用 |
| --- | --- |
| -a | 强制重新编译所有涉及到的包，包括标准库中的代码包，这会重写 /usr/local/go 目录下的 `.a` 文件 |
| -n | 打印命令执行过程，不真正执行 |
| -p n | 指定编译过程中命令执行的并行数，n 默认为 CPU 核数 |
| -race | 检测并报告程序中的数据竞争问题 |
| -v | 打印命令执行过程中所涉及到的代码包名称 |
| -x | 打印命令执行过程中所涉及到的命令，并执行 |
| -work | 打印编译过程中的临时文件夹。通常情况下，编译完成后会被删除 |

我们知道，Go 语言的源码文件分为三类：命令源码、库源码、测试源码。

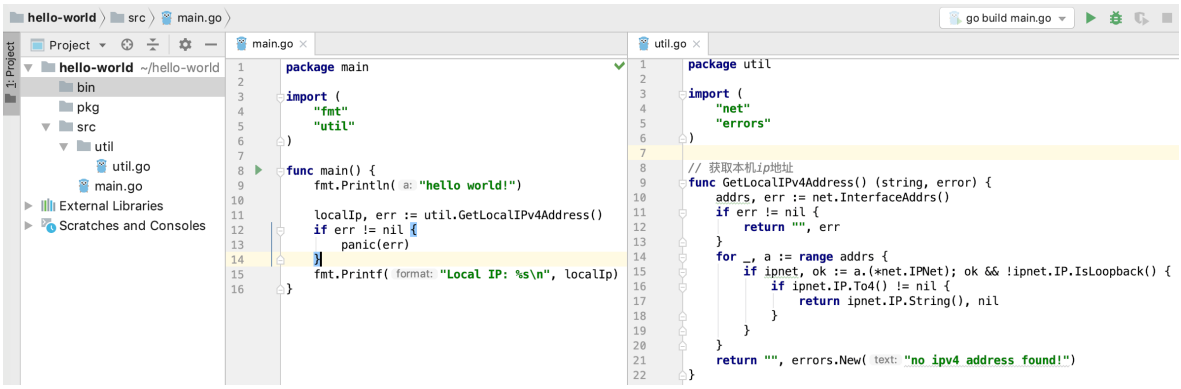
> 命令源码文件：是 Go 程序的入口，包含 `func main()` 函数，且第一行用 `package main` 声明属于 main 包。

> 库源码文件：主要是各种函数、接口等，例如工具类的函数。

> 测试源码文件：以 `_test.go` 为后缀的文件，用于测试程序的功能和性能。

注意， `go build` 会忽略 `*_test.go` 文件。

我们通过一个很简单的例子来演示 `go build` 命令。我用 Goland 新建了一个 `hello-world` 项目（为了展示引用自定义的包，和之前的 hello-world 程序不同），项目的结构如下：




最左边可以看到项目的结构，包含三个文件夹：bin，pkg，src。其中 src 目录下有一个 main.go，里面定义了 main 函数，是整个项目的入口，也就是前面提过的所谓的命令源码文件；src 目录下还有一个 util 目录，里面有 util.go 文件，定义了一个可以获取本机 IP 地址的函数，也就是所谓的库源码文件。

中间是 main.go 的源码，引用了两个包，一个是标准库的 fmt；一个是 util 包，util 的导入路径是 `util` 。所谓的导入路径是指相对于 Go 的源码目录 `$GoRoot/src` 或者 `$GoPath/src` 的下的子路径。例如 main 包里引用的 fmt 的

源码路径是 `/usr/local/go/src/fmt`，而 util 的源码路径是 `/Users/qcrao/hello-world/src/util`，正好我们设置的 GoPath = /Users/qcrao/hello-world。

最右边是库函数的源码，实现了获取本机 IP 的函数。

在 src 目录下，直接执行 `go build` 命令，在同级目录生成了一个可执行文件，文件名为 `src`，使用 `./src` 命令直接执行，输出：

```
hello world!
Local IP: 192.168.1.3
```

我们也可以指定生成的可执行文件的名称：

```
go build -o bin/hello
```

这样，在 bin 目录下会生成一个可执行文件，运行结果和上面的 `src` 一样。

其实，util 包可以单独被编译。我们可以在项目根目录下执行：

```
go build util
```

编译程序会去 $GoPath/src 路径找 util 包（其实是找文件夹）。还可以在 `./src/util` 目录下直接执行 `go build` 编译。

当然，直接编译库源码文件不会生成 .a 文件，因为：

> go build 命令在编译只包含库源码文件的代码包（或者同时编译多个代码包）时，只会做检查性的编译，而不会输出任何结果文件。

为了展示整个编译链接的运行过程，我们在项目根目录执行如下的命令：

```
go build -v -x -work -o bin/hello src/main.go
```

`-v` 会打印所编译过的包名字，`-x` 打印编译期间所执行的命令，`-work` 打印编译期间生成的临时文件路径，并且编译完成之后不会被删除。

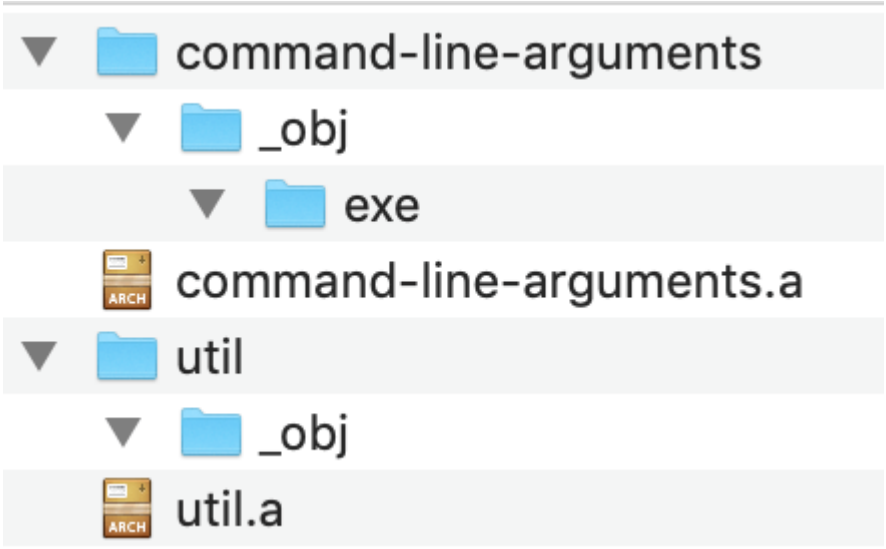
执行结果：

```
qcrao@qcrao ~/hello-world go build -v -x -work -o bin/hello src/main.go
WORK=/var/folders/df/d1llp90s68b739_7mg4ck3s40000gn/T/go-build122206432
util
mkdir -p $WORK/util/_obj/
mkdir -p $WORK/
cd /Users/qcrao/hello-world/src/util
/usr/local/go/pkg/tool/darwin_amd64/compile -o $WORK/util.a -trimpath $WORK -goversion go1.9.2 -p util -complete -b
uildid 7599f6bb75c982820e56f44973775ab1a3785da1 -D _/Users/qcrao/hello-world/src/util -I $WORK -pack ./util.go
command-line-arguments
mkdir -p $WORK/command-line-arguments/_obj/
mkdir -p $WORK/command-line-arguments/_obj/exe/
cd /Users/qcrao/hello-world/src
/usr/local/go/pkg/tool/darwin_amd64/compile -o $WORK/command-line-arguments.a -trimpath $WORK -goversion go1.9.2 -p
 main -complete -buildid b4d63f8a097eab7a04bc2a7ffd343f16c5f124a9 -D _/Users/qcrao/hello-world/src -I $WORK -I /Use
rs/qcrao/hello-world/pkg/darwin_amd64 -pack ./main.go
cd .
/usr/local/go/pkg/tool/darwin_amd64/link -o $WORK/command-line-arguments/_obj/exe/a.out -L $WORK -L /Users/qcrao/he
llo-world/pkg/darwin_amd64 -extld=clang -buildmode=exe -buildid=b4d63f8a097eab7a04bc2a7ffd343f16c5f124a9 $WORK/comm
and-line-arguments.a
mkdir -p bin/
mv $WORK/command-line-arguments/_obj/exe/a.out bin/hello
```

从结果来看，图中用箭头标注了本次编译过程涉及 2 个包：util，command-line-arguments。第二个包比较诡异，源码里根本就没有这个名字好吗？其实这是 `go build` 命令检测到 [packages] 处填的是一个 `.go` 文件，因此创建了一个虚拟的包：command-line-arguments。

同时，用红框圈出了 compile, link，也就是先编译了 util 包和 `main.go` 文件，分别得到 `.a` 文件，之后将两者进行链接，最终生成可执行文件，并且移动到 bin 目录下，改名为 hello。

另外，第一行显示了编译过程中的工作目录，此目录的文件结构是：




可以看到，和 hello-world 目录的层级基本一致。command-line-arguments 就是虚拟的 main.go 文件所处的包。exe 目录下的可执行文件在最后一步被移动到了 bin 目录下，所以这里是空的。

整体来看，`go build` 在执行时，会先递归寻找 main.go 所依赖的包，以及依赖的依赖，直至最底层的包。这里可以是深度优先遍历也可以是宽度优先遍历。如果发现有循环依赖，就会直接退出，这也是经常会发生的循环引用编译错误。

正常情况下，这些依赖关系会形成一棵倒着生长的树，树根在最上面，就是 main.go 文件，最下面是没有任何其他依赖的包。编译器会从最左的节点所代表的包开始挨个编译，完成之后，再去编译上一层的包。

这里，引用郝林老师几年前在 github 上发表的 go 命令教程，可以从参考资料找到原文地址。

> 从代码包编译的角度来说，如果代码包 A 依赖代码包 B，则称代码包 B 是代码包 A 的依赖代码包（以下简称依赖包），代码包 A 是代码包 B 的触发代码包（以下简称触发包）。

> 执行 `go build` 命令的计算机如果拥有多个逻辑 CPU 核心，那么编译代码包的顺序可能会存在一些不确定性。但是，它一定会满足这样的约束条件：依赖代码包 -> 当前代码包 -> 触发代码包。

顺便推荐一个浏览器插件 Octotree，在看 github 项目的时候，此插件可以在浏览器里直接展示整个项目的文件结构，非常方便：


到这里，你一定会发现，对于 hello-wrold 文件夹下的 pkg 目录好像一直没有涉及到。

其实，pkg 目录下面应该存放的是涉及到的库文件编译后的包，也就是一些 `.a` 文件。但是 go build 执行过程中，这些 `.a` 文件放在临时文件夹中，编译完成后会被直接删掉，因此一般不会用到。

前面我们提到过，在 go build 命令里加上 `-i` 参数会安装这些库文件编译的包，也就是这些 `.a` 文件会放到 pkg 目录下。

在项目根目录执行 `go build -i src/main.go` 后，pkg 目录里增加了 util.a 文件：


`darwin_amd64` 表示的是：

> GOOS 和 GOARCH。这两个环境变量不用我们设置，系统默认的。

> GOOS 是 Go 所在的操作系统类型，GOARCH 是 Go 所在的计算架构。

> Mac 平台上这个目录名就是 darwin_amd64。

生成了 util.a 文件后，再次编译的时候，就不会再重新编译 util.go 文件，加快了编译速度。

同时，在根目录下生成了名称为 main 的可执行文件，这是以 main.go 的文件名命令的。

hello-world 这个项目的代码已经上传到了 github 项目 `Go-Questions` ，这个项目由问题导入，企图串连 Go 的所有知识点，正在完善，期待你的 star。 地址见参考资料【Go-Questions hello-world项目】。

# go install

`go install` 用于编译并安装指定的代码包及它们的依赖包。相比 `go build` ，它只是多了一个“安装编译后的结果文件到指定目录”的步骤。

还是使用之前 hello-world 项目的例子，我们先将 pkg 目录删掉，在项目根目录执行：

```
go install src/main.go

或者

go install util
```

两者都会在根目录下新建一个 `pkg` 目录，并且生成一个 `util.a` 文件。

并且，在执行前者的时候，会在 GOBIN 目录下生成名为 main 的可执行文件。

所以，运行 `go install` 命令，库源码包对应的 `.a` 文件会被放置到 `pkg` 目录下，命令源码包生成的可执行文件会被放到 GOBIN 目录。

`go install` 在 GoPath 有多个目录的时候，会产生一些问题，具体可以去看郝林老师的 `Go 命令教程` ，这里不展开了。

# go run

`go run` 用于编译并运行命令源码文件。

在 hello-world 项目的根目录，执行 go run 命令：

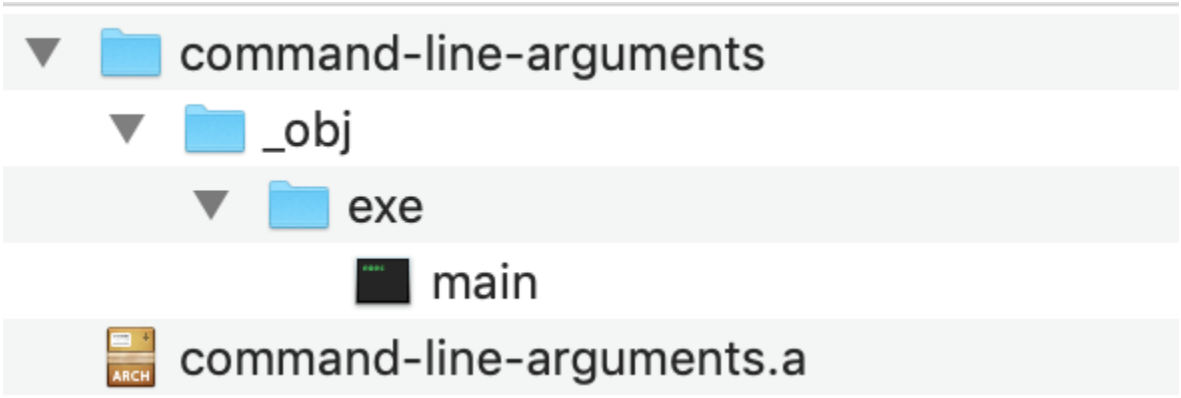
```
go run -x -work src/main.go
```

-x 可以打印整个过程涉及到的命令，-work 可以看到临时的工作目录：

```
qcrao@qcrao ~/hello-world go run -x -work src/main.go
WORK=/var/folders/df/d1llp90s68b739_7mg4ck3s40000gn/T/go-build441629785
mkdir -p $WORK/command-line-arguments/_obj/
mkdir -p $WORK/command-line-arguments/_obj/exe/
cd /Users/qcrao/hello-world/src
/usr/local/go/pkg/tool/darwin_amd64/compile -o $WORK/command-line-arguments.a -trimpath $WORK -goversion go1.9.2 -p
 main -complete -buildid b4d63f8a097eab7a04bc2a7ffd343f16c5f124a9 -dwarf=false -D _/Users/qcrao/hello-world/src -I
$WORK -I /Users/qcrao/hello-world/pkg/darwin_amd64 -pack ./main.go
cd .
/usr/local/go/pkg/tool/darwin_amd64/link -o $WORK/command-line-arguments/_obj/exe/main -L $WORK -L /Users/qcrao/hel
lo-world/pkg/darwin_amd64 -s -w -extld=clang -buildmode=exe -buildid=b4d63f8a097eab7a04bc2a7ffd343f16c5f124a9 $WORK
/command-line-arguments.a
$WORK/command-line-arguments/_obj/exe/main
hello world!
Local IP: 192.168.1.3
```

从上图中可以看到，仍然是先编译，再连接，最后直接执行，并打印出了执行结果。

第一行打印的就是工作目录，最终生成的可执行文件就是放置于此：




main 就是最终生成的可执行文件。

# Go 编译链接过程概述

我们从一个 `Hello World` 的例子开始：

```
package main

import "fmt"

func main() {
    fmt.Println("hello world")
}
```

当我们用键盘敲完上面的 hello world 代码时，保存在硬盘上的 `hello.go` 文件就是一个字节序列了，每个字节代表一个字符。

用 vim 打开 hello.go 文件，在命令行模式下，输入命令：

```
:%!xxd
```

就能在 vim 里以十六进制查看文件内容：

```
00000000: 7061 636b 6167 6520 6d61 696e 0a0a 696d  package main..im
00000010: 706f 7274 2022 666d 7422 0a0a 6675 6e63  port "fmt"..func
00000020: 206d 6169 6e28 2920 7b0a 0966 6d74 2e50   main() {..fmt.P
00000030: 7269 6e74 6c6e 2822 6865 6c6c 6f20 776f  rintln("hello wo
00000040: 726c 6422 290a 7d                        rld").}
```

最左边的一列代表地址值，中间一列代表文本对应的 ASCII 字符，最右边的列就是我们的代码。再在终端里执行 `man ascii` ：

```
00 nul   01 soh   02 stx   03 etx   04 eot   05 enq   06 ack   07 bel
08 bs    09 ht    0a nl    0b vt    0c np    0d cr    0e so    0f si
10 dle   11 dc1   12 dc2   13 dc3   14 dc4   15 nak   16 syn   17 etb
18 can   19 em    1a sub   1b esc   1c fs    1d gs    1e rs    1f us
20 sp    21  !    22  "    23  #    24  $    25  %    26  &    27  '
28  (    29  )    2a  *    2b  +    2c  ,    2d  -    2e  .    2f  /
30  0    31  1    32  2    33  3    34  4    35  5    36  6    37  7
38  8    39  9    3a  :    3b  ;    3c  <    3d  =    3e  >    3f  ?
40  @    41  A    42  B    43  C    44  D    45  E    46  F    47  G
48  H    49  I    4a  J    4b  K    4c  L    4d  M    4e  N    4f  O
50  P    51  Q    52  R    53  S    54  T    55  U    56  V    57  W
58  X    59  Y    5a  Z    5b  [    5c  \    5d  ]    5e  ^    5f  _
60  `    61  a    62  b    63  c    64  d    65  e    66  f    67  g
68  h    69  i    6a  j    6b  k    6c  l    6d  m    6e  n    6f  o
70  p    71  q    72  r    73  s    74  t    75  u    76  v    77  w
78  x    79  y    7a  z    7b  {    7c  |    7d  }    7e  ~    7f del
```

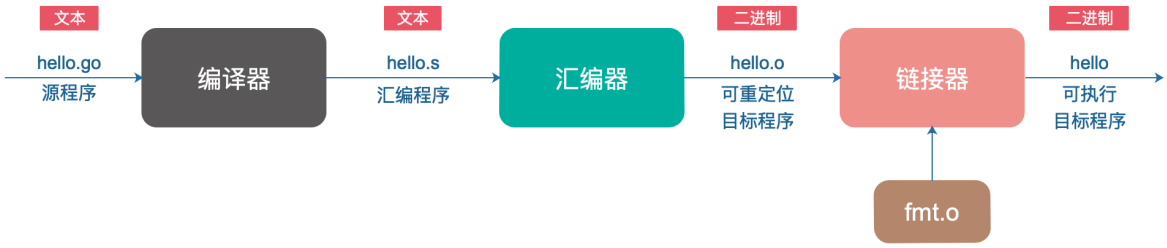
和 ASCII 字符表一对比，就能发现，中间的列和最右边的列是一一对应的。也就是说，刚刚写完的 hello.go 文件都是由 ASCII 字符表示的，它被称为 `文本文件` ，其他文件被称为 `二进制文件` 。

当然，更深入地看，计算机中的所有数据，像磁盘文件、网络中的数据其实都是一串比特位组成，取决于如何看待它。在不同的情景下，一个相同的字节序列可能表示成一个整数、浮点数、字符串或者是机器指令。

而像 hello.go 这个文件，8 个 bit，也就是一个字节看成一个单位（假定源程序的字符都是 ASCII 码），最终解释成人类能读懂的 Go 源码。

Go 程序并不能直接运行，每条 Go 语句必须转化为一系列的低级机器语言指令，将这些指令打包到一起，并以二进制磁盘文件的形式存储起来，也就是可执行目标文件。

从源文件到可执行目标文件的转化过程：




完成以上各个阶段的就是 Go 编译系统。你肯定知道大名鼎鼎的 GCC（GNU Compile Collection），中文名为 GNU 编译器套装，它支持像 C，C++，Java，Python，Objective-C，Ada，Fortran，Pascal，能够为很多不同的机器生成机器码。

可执行目标文件可以直接在机器上执行。一般而言，先执行一些初始化的工作；找到 main 函数的入口，执行用户写的代码；执行完成后，main 函数退出；再执行一些收尾的工作，整个过程完毕。

在接下来的文章里，我们将探索 `编译` 和 `运行` 的过程。

Go 源码里的编译器源码位于 `src/cmd/compile` 路径下，链接器源码位于 `src/cmd/link` 路径下。

# 编译过程

我比较喜欢用 IDE（集成开发环境）来写代码， Go 源码用的 Goland，有时候直接点击 IDE 菜单栏里的“运行”按钮，程序就跑起来了。这实际上隐含了编译和链接的过程，我们通常将编译和链接合并到一起的过程称为构建（Build）。

编译过程就是对源文件进行词法分析、语法分析、语义分析、优化，最后生成汇编代码文件，以 `.s` 作为文件后缀。

之后，汇编器会将汇编代码转变成机器可以执行的指令。由于每一条汇编语句几乎都与一条机器指令相对应，所以只是一个简单的一一对应，比较简单，没有语法、语义分析，也没有优化这些步骤。

编译器是将高级语言翻译成机器语言的一个工具，编译过程一般分为 6 步：扫描、语法分析、语义分析、源代码优化、代码生成、目标代码优化。下图来自《程序员的自我修养》：

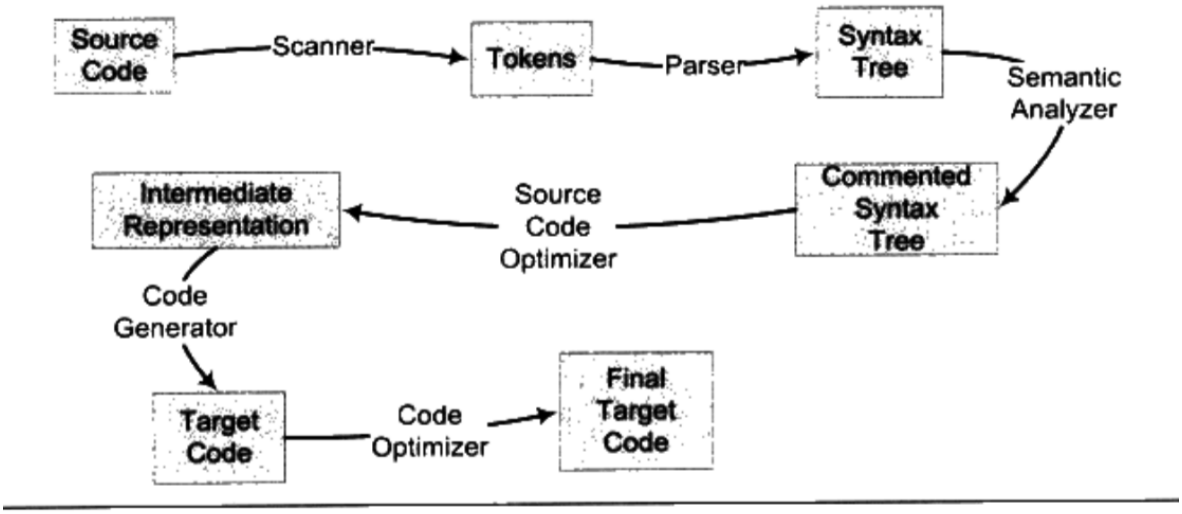



图 2-2　编译过程

# 词法分析

通过前面的例子，我们知道，Go 程序文件在机器看来不过是一堆二进制位。我们能读懂，是因为 Goland 按照 ASCII 码（实际上是 UTF-8）把这堆二进制位进行了编码。例如，把 8个 bit 位分成一组，对应一个字符，通过对照 ASCII 码表就可以查出来。

当把所有的二进制位都对应成了 ASCII 码字符后，我们就能看到有意义的字符串。它可能是关键字，例如：package；可能是字符串，例如：“Hello World”。

词法分析其实干的就是这个。输入是原始的 Go 程序文件，在词法分析器看来，就是一堆二进制位，根本不知道是什么东西，经过它的分析后，变成有意义的记号。简单来说，词法分析是计算机科学中将字符序列转换为标记（token）序列的过程。

我们来看一下维基百科上给出的定义：

> 词法分析（lexical analysis）是计算机科学中将字符序列转换为标记（token）序列的过程。进行词法分析的程序或者函数叫作词法分析器（lexical analyzer，简称lexer），也叫扫描器（scanner）。词法分析器一般以函数的形式存在，供语法分析器调用。

`.go` 文件被输入到扫描器（Scanner），它使用一种类似于 `有限状态机` 的算法，将源代码的字符系列分割成一系列的记号（Token）。

记号一般分为这几类：关键字、标识符、字面量（包含数字、字符串）、特殊符号（如加号、等号）。

例如，对于如下的代码：

```
slice[i] = i * (2 + 6)
```

总共包含 16 个非空字符，经过扫描后，

| 记号 | 类型 |
| --- | --- |
| slice | 标识符 |
| [ | 左方括号 |

| 记号 | 类型 |
| --- | --- |
| i | 标识符 |
| ] | 右方括号 |
| = | 赋值 |
| i | 标识符 |
| * | 乘号 |
| ( | 左圆括号 |
| 2 | 数字 |
| + | 加号 |
| 6 | 数字 |
| ) | 右圆括号 |

上面的例子源自《程序员的自我修养》，主要讲解编译、链接相关的内容，很精彩，推荐研读。

Go 语言（本文的 Go 版本是 1.9.2）扫描器支持的 Token 在源码中的路径：

```
src/cmd/compile/internal/syntax/token.go
```

感受一下：

```
var tokstrings = [...]string{
    // source control
    _EOF: "EOF",

    // names and literals
    _Name:    "name",
    _Literal: "literal",

    // operators and operations
    _Operator: "op",
    _AssignOp: "op=",
    _IncOp:    "opop",
    _Assign:   "=",
    _Define:   ":=",
    _Arrow:    "<-",
    _Star:     "*",

    // delimitors
    _Lparen:    "(",
    _Lbrack:    "[",
    _Lbrace:    "{",
    _Rparen:    ")",
```

```
    _Rbrack:    "]",
    _Rbrace:    "}",
    _Comma:     ",",
    _Semi:      ";",
    _Colon:     ":",
    _Dot:       ".",
    _DotDotDot: "...",

    // keywords
    _Break:       "break",
    _Case:        "case",
    _Chan:        "chan",
    _Const:       "const",
    _Continue:    "continue",
    _Default:     "default",
    _Defer:       "defer",
    _Else:        "else",
    _Fallthrough: "fallthrough",
    _For:         "for",
    _Func:        "func",
    _Go:          "go",
    _Goto:        "goto",
    _If:          "if",
    _Import:      "import",
    _Interface:   "interface",
    _Map:         "map",
    _Package:     "package",
    _Range:       "range",
    _Return:      "return",
    _Select:      "select",
    _Struct:      "struct",
    _Switch:      "switch",
    _Type:        "type",
    _Var:         "var",
}
```

还是比较熟悉的，包括名称和字面量、操作符、分隔符和关键字。

而扫描器的路径是：

```
src/cmd/compile/internal/syntax/scanner.go
```

其中最关键的函数就是 next 函数，它不断地读取下一个字符（不是下一个字节，因为 Go 语言支持 Unicode 编码，并不是像我们前面举得 ASCII 码的例子，一个字符只有一个字节），直到这些字符可以构成一个 Token。

```
func (s *scanner) next() {
// ......

redo:
    // skip white space
    c := s.getr()
    for c == ' ' || c == '\t' || c == '\n' && !nlsemi || c == '\r' {
        c = s.getr()
    }

    // token start
    s.line, s.col = s.source.line0, s.source.col0

    if isLetter(c) || c >= utf8.RuneSelf && s.isIdentRune(c, true) {
```

```
        s.ident()
        return
    }

    switch c {
    // ......

    case '\n':
        s.lit = "newline"
        s.tok = _Semi

    case '0', '1', '2', '3', '4', '5', '6', '7', '8', '9':
        s.number(c)

    // ......

  default:
        s.tok = 0
        s.error(fmt.Sprintf("invalid character %#U", c))
        goto redo
    return

assignop:
    if c == '=' {
        s.tok = _AssignOp
        return
    }
    s.ungetr()
    s.tok = _Operator
}
```

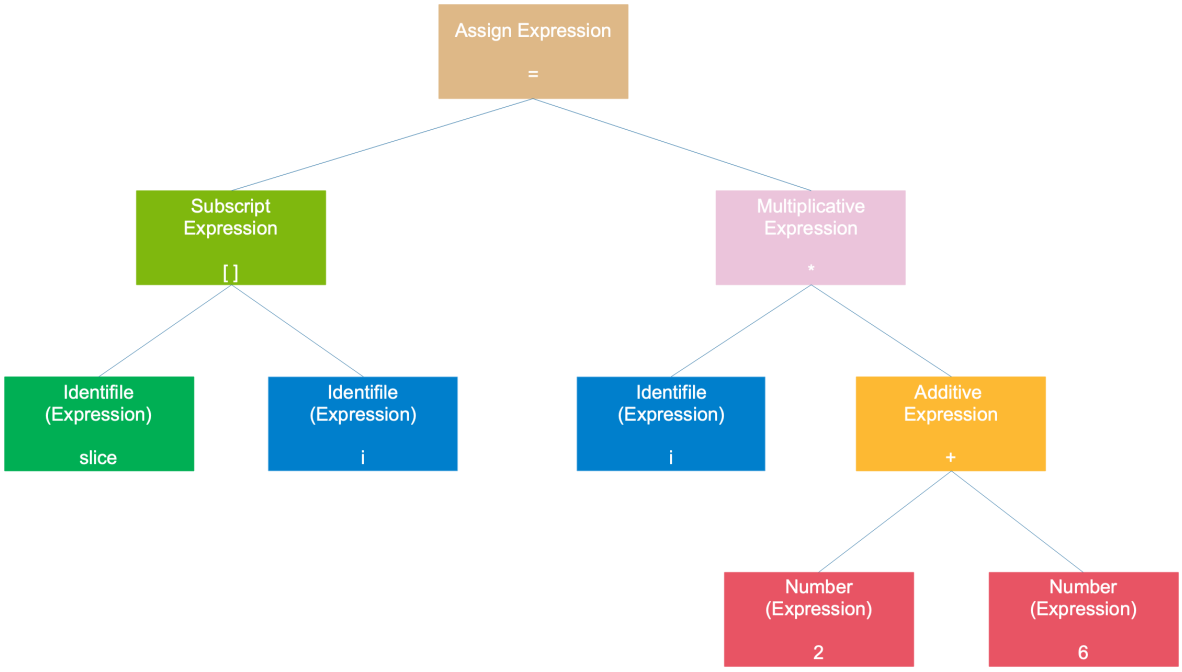
代码的主要逻辑就是通过 `c := s.getr()` 获取下一个未被解析的字符，并且会跳过之后的空格、回车、换行、tab 字符，然后进入一个大的 `switch-case` 语句，匹配各种不同的情形，最终可以解析出一个 Token，并且把相关的行、列数字记录下来，这样就完成一次解析过程。

> 当前包中的词法分析器 scanner 也只是为上层提供了 next 方法，词法解析的过程都是惰性的，只有在上层的解析器需要时才会调用 next 获取最新的 Token。

# 语法分析

上一步生成的 Token 序列，需要经过进一步处理，生成一棵以 `表达式` 为结点的 `语法树` 。

比如最开始的那个例子， `slice[i] = i * (2 + 6)` ，得到的一棵语法树如下：




整个语句被看作是一个赋值表达式，左子树是一个数组表达式，右子树是一个乘法表达式；数组表达式由 2 个符号表达式组成；乘号表达式则是由一个符号表达式和一个加号表达式组成；加号表达式则是由两个数字组成。符号和数字是最小的表达式，它们不能再被分解，通常作为树的叶子节点。

语法分析的过程可以检测一些形式上的错误，例如：括号是否缺少一半，`+` 号表达式缺少一个操作数等。

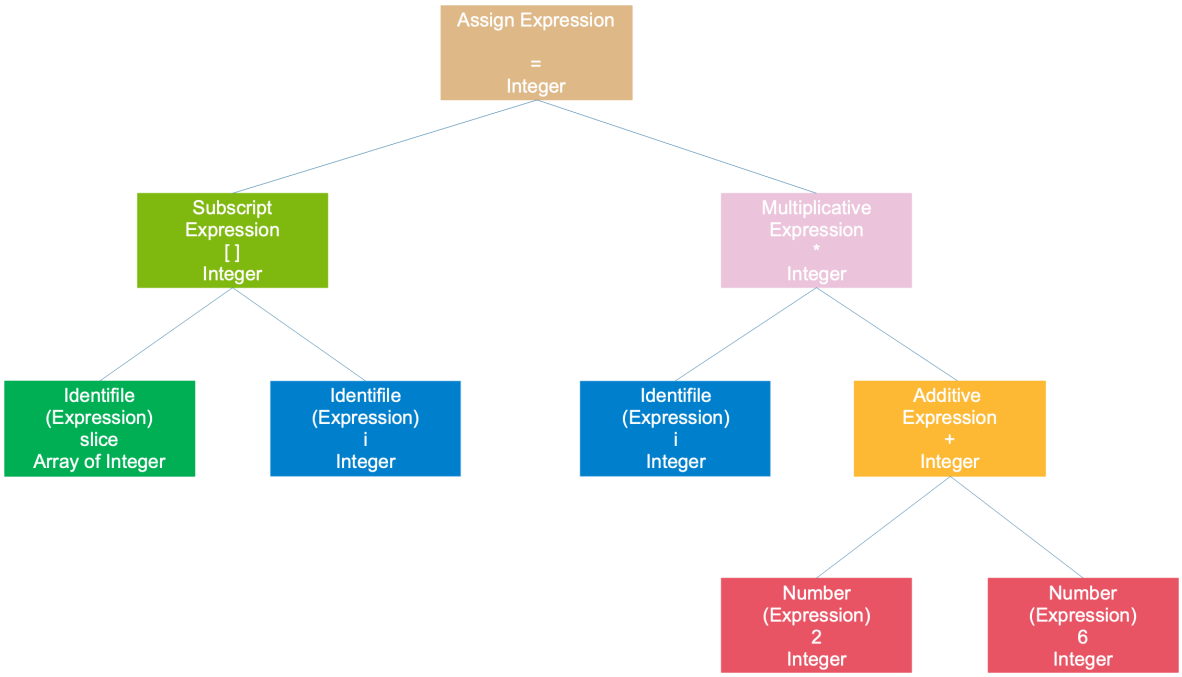
> 语法分析是根据某种特定的形式文法（Grammar）对 Token 序列构成的输入文本进行分析并确定其语法结构的一种过程。

# 语义分析

语法分析完成后，我们并不知道语句的具体意义是什么。像上面的 `*` 号的两棵子树如果是两个指针，这是不合法的，但语法分析检测不出来，语义分析就是干这个事。

编译期所能检查的是静态语义，可以认为这是在“代码”阶段，包括变量类型的匹配、转换等。例如，将一个浮点值赋给一个指针变量的时候，明显的类型不匹配，就会报编译错误。而对于运行期间才会出现的错误：不小心除了一个 0 ，语义分析是没办法检测的。

语义分析阶段完成之后，会在每个节点上标注上类型：




Go 语言编译器在这一阶段检查常量、类型、函数声明以及变量赋值语句的类型，然后检查哈希中键的类型。实现类型检查的函数通常都是几千行的巨型 switch/case 语句。

> 类型检查是 Go 语言编译的第二个阶段，在词法和语法分析之后我们得到了每个文件对应的抽象语法树，随后的类型检查会遍历抽象语法树中的节点，对每个节点的类型进行检验，找出其中存在的语法错误。

> 在这个过程中也可能会对抽象语法树进行改写，这不仅能够去除一些不会被执行的代码对编译进行优化提高执行效率，而且也会修改 make、new 等关键字对应节点的操作类型。

例如比较常用的 make 关键字，用它可以创建各种类型，如 slice，map，channel 等等。到这一步的时候，对于 make 关键字，也就是 OMAKE 节点，会先检查它的参数类型，根据类型的不同，进入相应的分支。如果参数类型是 slice，就会进入 TSLICE case 分支，检查 len 和 cap 是否满足要求，如 len <= cap。最后节点类型会从 OMAKE 改成 OMAKESLICE。

# 中间代码生成

我们知道，编译过程一般可以分为前端和后端，前端生成和平台无关的中间代码，后端会针对不同的平台，生成不同的机器码。

前面词法分析、语法分析、语义分析等都属于编译器前端，之后的阶段属于编译器后端。

编译过程有很多优化的环节，在这个环节是指源代码级别的优化。它将语法树转换成中间代码，它是语法树的顺序表示。

中间代码一般和目标机器以及运行时环境无关，它有几种常见的形式：三地址码、P-代码。例如，最基本的 `三地址码` 是这样的：

```
x = y op z
```

表示变量 y 和 变量 z 进行 op 操作后，赋值给 x。op 可以是数学运算，例如加减乘除。

前面我们举的例子可以写成如下的形式：

```
t1 = 2 + 6
t2 = i * t1
```

```
slice[i] = t2
```

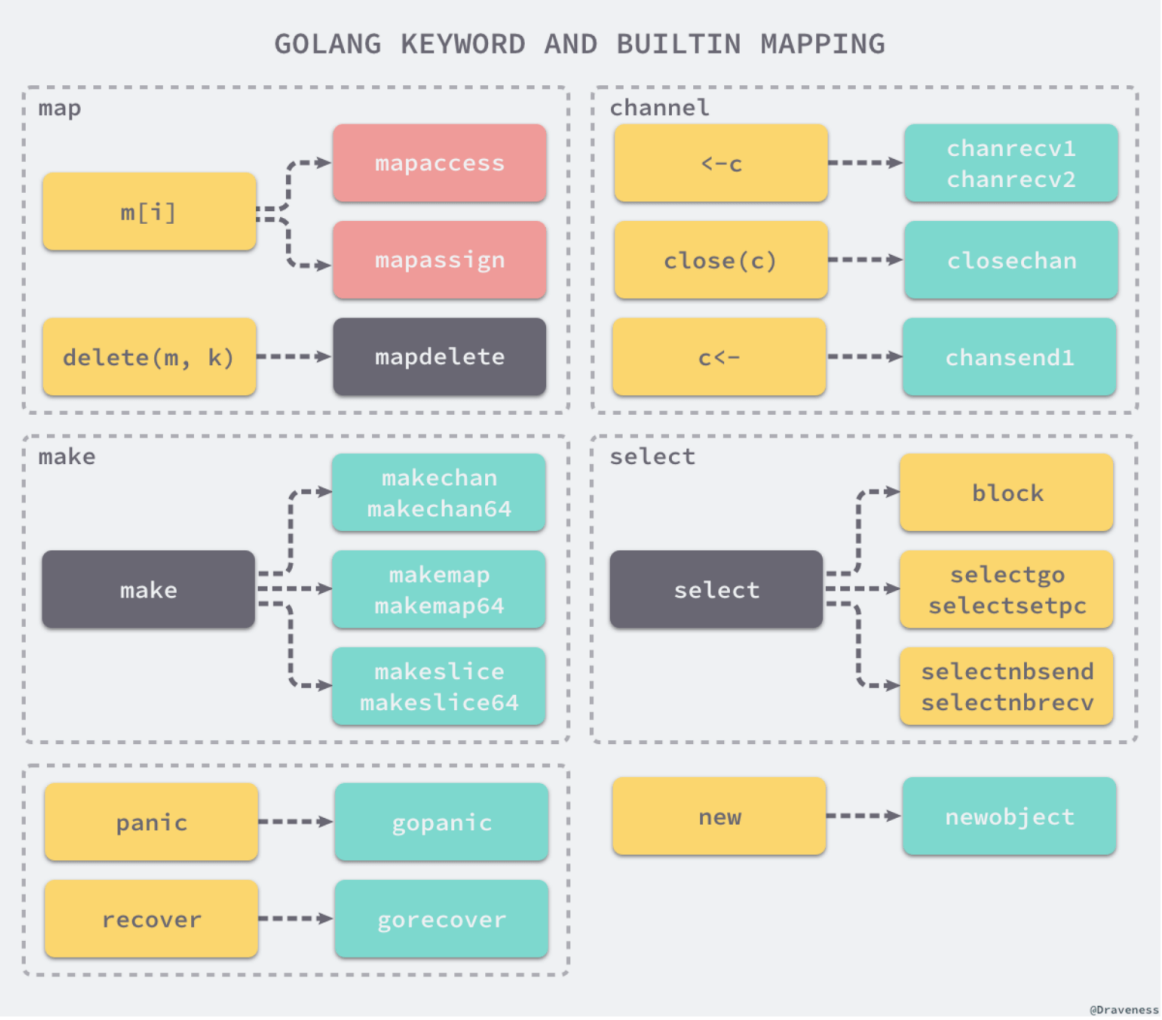
这里 2 + 6 是可以直接计算出来的，这样就把 t1 这个临时变量“优化”掉了，而且 t1 变量可以重复利用，因此 t2 也可以“优化”掉。优化之后：

```
t1 = i * 8
slice[i] = t1
```

Go 语言的中间代码表示形式为 SSA（Static Single-Assignment，静态单赋值），之所以称之为单赋值，是因为每个名字在 SSA 中仅被赋值一次。。

这一阶段会根据 CPU 的架构设置相应的用于生成中间代码的变量，例如编译器使用的指针和寄存器的大小、可用寄存器列表等。中间代码生成和机器码生成这两部分会共享相同的设置。

在生成中间代码之前，会对抽象语法树中节点的一些元素进行替换。这里引用《面向信仰编程》编译原理相关博客里的一张图：




例如对于 map 的操作 m[i]，在这里会被转换成 mapacess 或 mapassign。

> Go 语言的主程序在执行时会调用 runtime 中的函数，也就是说关键字和内置函数的功能其实是由语言的编译器和运行时共同完成的。

> 中间代码的生成过程其实就是从 AST 抽象语法树到 SSA 中间代码的转换过程，在这期间会对语法树中的关键字在进行一次更新，更新后的语法树会经过多轮处理转变最后的 SSA 中间代码。

# 目标代码生成与优化

不同机器的机器字长、寄存器等等都不一样，意味着在不同机器上跑的机器码是不一样的。最后一步的目的就是要生成能在不同 CPU 架构上运行的代码。

为了榨干机器的每一滴油水，目标代码优化器会对一些指令进行优化，例如使用移位指令代替乘法指令等。

这块实在没能力深入，幸好也不需要深入。对于应用层的软件开发工程师来说，了解一下就可以了。

# 链接过程

编译过程是针对单个文件进行的，文件与文件之间不可避免地要引用定义在其他模块的全局变量或者函数，这些变量或函数的地址只有在此阶段才能确定。

链接过程就是要把编译器生成的一个个目标文件链接成可执行文件。最终得到的文件是分成各种段的，比如数据段、代码段、BSS段等等，运行时会被装载到内存中。各个段具有不同的读写、执行属性，保护了程序的安全运行。

这部分内容，推荐看《程序员的自我修养》和《深入理解计算机系统》。

# GoRoot 和 GoPath 有什么用

GoRoot 是 Go 的安装路径。mac 或 unix 是在 `/usr/local/go` 路径上，来看下这里都装了些什么：

```
[qcrao@qcrao go]$ ls
api      bin   CONTRIBUTING.md  doc          lib      misc     pkg        robots.txt  test
AUTHORS  blog  CONTRIBUTORS     favicon.ico  LICENSE  PATENTS  README.md  src         VERSION
```

bin 目录下面：

```
[qcrao@qcrao go]$ ls pkg/
include      linux_amd64_dynlink  linux_amd64_shared              tool
linux_amd64  linux_amd64_race     linux_amd64_testcshared_shared
```

pkg 目录下面：

```
[qcrao@qcrao go]$ ls pkg/
include      linux_amd64_dynlink  linux_amd64_shared              tool
linux_amd64  linux_amd64_race     linux_amd64_testcshared_shared
[qcrao@qcrao go]$ ls pkg/linux_amd64
archive    context.a  encoding.a  hash     index     math    os         regexp     strings.a  text
bufio.a    crypto     errors.a    hash.a   internal  math.a  os.a       regexp.a   sync       time.a
bytes.a    crypto.a   expvar.a    html     io        mime    path       runtime    sync.a     unicode
cmd        database   flag.a      html.a   io.a      mime.a  path.a     runtime.a  syscall.a  unicode.a
compress   debug      fmt.a       image    log       net     plugin.a   sort.a     testing    vendor
container  encoding   go          image.a  log.a     net.a   reflect.a  strconv.a  testing.a
```

Go 工具目录如下，其中比较重要的有编译器 `compile`，链接器 `link`：

```
/usr/local/go/pkg/tool/linux_amd64
[qcrao@qcrao linux_amd64]$ ls
addr2line  buildid  compile  dist  fix   nm       pack   test2json  trace
asm        cgo      cover    doc   link  objdump  pprof  tour       vet
```

GoPath 的作用在于提供一个可以寻找 `.go` 源码的路径，它是一个工作空间的概念，可以设置多个目录。Go 官方要求，GoPath 下面需要包含三个文件夹：

```
src
pkg
bin
```

src 存放源文件，pkg 存放源文件编译后的库文件，后缀为 `.a`；bin 则存放可执行文件。

# 逃逸分析是怎么进行的

在编译原理中，分析指针动态范围的方法称之为逃逸分析。通俗来讲，当一个对象的指针被多个方法或线程引用时，我们称这个指针发生了逃逸。

Go语言的逃逸分析是编译器执行静态代码分析后，对内存管理进行的优化和简化，它可以决定一个变量是分配到堆还栈上。

写过C/C++的同学都知道，调用著名的malloc和new函数可以在堆上分配一块内存，这块内存的使用和销毁的责任都在程序员。一不小心，就会发生内存泄露。

Go语言里，基本不用担心内存泄露了。虽然也有new函数，但是使用new函数得到的内存不一定就在堆上。堆和栈的区别对程序员“模糊化”了，当然这一切都是Go编译器在背后帮我们完成的。

Go语言逃逸分析最基本的原则是：如果一个函数返回对一个变量的引用，那么它就会发生逃逸。

简单来说，编译器会分析代码的特征和代码生命周期，Go中的变量只有在编译器可以证明在函数返回后不会再被引用的，才分配到栈上，其他情况下都是分配到堆上。

Go语言里没有一个关键字或者函数可以直接让变量被编译器分配到堆上，相反，编译器通过分析代码来决定将变量分配到何处。

对一个变量取地址，可能会被分配到堆上。但是编译器进行逃逸分析后，如果考察到在函数返回后，此变量不会被引用，那么还是会被分配到栈上。

编译器会根据变量是否被外部引用来决定是否逃逸：

> 1. 如果函数外部没有引用，则优先放到栈中；
> 2. 如果函数外部存在引用，则必定放到堆中；

写C/C++代码时，为了提高效率，常常将pass-by-value（传值）“升级”成pass-by-reference，企图避免构造函数的运行，并且直接返回一个指针。

你一定还记得，这里隐藏了一个很大的坑：在函数内部定义了一个局部变量，然后返回这个局部变量的地址（指针）。这些局部变量是在栈上分配的（静态内存分配），一旦函数执行完毕，变量占据的内存会被销毁，任何对这个返回值作的动作（如解引用），都将扰乱程序的运行，甚至导致程序直接崩溃。比如下面的这段代码：

```
int *foo ( void )
{
    int t = 3;
    return &t;
}
```

有些同学可能知道上面这个坑，用了个更聪明的做法：在函数内部使用new函数构造一个变量（动态内存分配），然后返回此变量的地址。因为变量是在堆上创建的，所以函数退出时不会被销毁。但是，这样就行了吗？new出来的对象该在何时何地delete呢？调用者可能会忘记delete或者直接拿返回值传给其他函数，之后就再也不能delete它了，也就是发生了内存泄露。关于这个坑，大家可以去看看《Effective C++》条款21，讲得非常好！

C++是公认的语法最复杂的语言，据说没有人可以完全掌握C++的语法。而这一切在Go语言中就大不相同了。像上面示例的C++代码放到Go里，没有任何问题。

前面讲的C/C++中出现的问题，在Go中作为一个语言特性被大力推崇。真是C/C++之砒霜Go之蜜糖！

C/C++中动态分配的内存需要我们手动释放，导致我们平时在写程序时，如履薄冰。这样做有他的好处：程序员可以完全掌控内存。但是缺点也是很多的：经常出现忘记释放内存，导致内存泄露。所以，很多现代语言都加上了垃圾回收机制。

Go的垃圾回收，让堆和栈对程序员保持透明。真正解放了程序员的双手，让他们可以专注于业务，“高效”地完成代码编写。把那些内存管理的复杂机制交给编译器，而程序员可以去享受生活。

逃逸分析这种“骚操作”把变量合理地分配到它该去的地方。即使你是用new申请到的内存，如果我发现你竟然在退出函数后没有用了，那么就把你丢到栈上，毕竟栈上的内存分配比堆上快很多；反之，即使你表面上只是一个普通的变量，但是经过逃逸分析后发现在退出函数之后还有其他地方在引用，那我就把你分配到堆上。

如果变量都分配到堆上，堆不像栈可以自动清理。它会引起Go频繁地进行垃圾回收，而垃圾回收会占用比较大的系统开销（占用CPU容量的25%）。

堆和栈相比，堆适合不可预知大小的内存分配。但是为此付出的代价是分配速度较慢，而且会形成内存碎片。栈内存分配则会非常快。栈分配内存只需要两个CPU指令：“PUSH”和“RELEASE”，分配和释放；而堆分配内存首先需要去找到一块大小合适的内存块，之后要通过垃圾回收才能释放。

通过逃逸分析，可以尽量把那些不需要分配到堆上的变量直接分配到栈上，堆上的变量少了，会减轻分配堆内存的开销，同时也会减少gc的压力，提高程序的运行速度。

引申1：如何查看某个变量是否发生了逃逸？
两种方法：使用go命令，查看逃逸分析结果；反汇编源码；

比如用这个例子：

```
package main

import "fmt"

func foo() *int {
    t := 3
    return &t;
}

func main() {
    x := foo()
    fmt.Println(*x)
}
```

使用go命令：

```
go build -gcflags '-m -l' main.go
```

加 `-l` 是为了不让foo函数被内联。得到如下输出：

```
# command-line-arguments
src/main.go:7:9: &t escapes to heap
src/main.go:6:7: moved to heap: t
src/main.go:12:14: *x escapes to heap
src/main.go:12:13: main ... argument does not escape
```

foo函数里的变量 `t` 逃逸了，和我们预想的一致。让我们不解的是为什么main函数里的 `x` 也逃逸了？这是因为有些函数参数为interface类型，比如fmt.Println(a ...interface{})，编译期间很难确定其参数的具体类型，也会发生逃逸。

反汇编源码：

```
go tool compile -S main.go
```

截取部分结果，图中标记出来的说明 `t` 是在堆上分配内存，发生了逃逸。

```
"".foo STEXT size=79 args=0x8 locals=0x18
        0x0000 00000 (./src/main.go:5)  TEXT    "".foo(SB), $24-8
        0x0000 00000 (./src/main.go:5)  MOVQ    (TLS), CX
        0x0009 00009 (./src/main.go:5)  CMPQ    SP, 16(CX)
        0x000d 00013 (./src/main.go:5)  JLS     72
        0x000f 00015 (./src/main.go:5)  SUBQ    $24, SP
        0x0013 00019 (./src/main.go:5)  MOVQ    BP, 16(SP)
        0x0018 00024 (./src/main.go:5)  LEAQ    16(SP), BP
        0x001d 00029 (./src/main.go:5)  FUNCDATA        $0, gclocals·2a5305abe05176240e61b8620e19a815(SB)
        0x001d 00029 (./src/main.go:5)  FUNCDATA        $1, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
        0x001d 00029 (./src/main.go:5)  LEAQ    type.int(SB), AX
        0x0024 00036 (./src/main.go:6)  MOVQ    AX, (SP)
        0x0028 00040 (./src/main.go:6)  PCDATA  $0, $0
        0x0028 00040 (./src/main.go:6)  CALL    runtime.newobject(SB)
        0x002d 00045 (./src/main.go:6)  MOVQ    8(SP), AX
        0x0032 00050 (./src/main.go:6)  MOVQ    $3, (AX)
        0x0039 00057 (./src/main.go:7)  MOVQ    AX, "".~r0+32(SP)
        0x003e 00062 (./src/main.go:7)  MOVQ    16(SP), BP
        0x0043 00067 (./src/main.go:7)  ADDQ    $24, SP
        0x0047 00071 (./src/main.go:7)  RET
        0x0048 00072 (./src/main.go:7)  NOP
        0x0048 00072 (./src/main.go:5)  PCDATA  $0, $-1
        0x0048 00072 (./src/main.go:5)  CALL    runtime.morestack_noctxt(SB)
        0x004d 00077 (./src/main.go:5)  JMP     0
```

引申2：下面代码中的变量发生逃逸了吗？
示例1：代码出处

```
package main
type S struct {}

func main() {
  var x S
  _ = identity(x)
}

func identity(x S) S {
  return x
}
```

分析：Go语言函数传递都是通过值的，调用函数的时候，直接在栈上copy出一份参数，不存在逃逸。

示例2：

```
package main

type S struct {}

func main() {
  var x S
  y := &x
  _ = *identity(y)
}

func identity(z *S) *S {
  return z
}
```

分析：identity函数的输入直接当成返回值了，因为没有对z作引用，所以z没有逃逸。对x的引用也没有逃出main函数的作用域，因此x也没有发生逃逸。

示例3：

```
package main

type S struct {}

func main() {
  var x S
  _ = *ref(x)
}

func ref(z S) *S {
  return &z
}
```

分析：z是对x的拷贝，ref函数中对z取了引用，所以z不能放在栈上，否则在ref函数之外，通过引用如何找到z，所以z必须要逃逸到堆上。仅管在main函数中，直接丢弃了ref的结果，但是Go的编译器还没有那么智能，分析不出来这种情况。而对x从来就没有取引用，所以x不会发生逃逸。

示例4：如果对一个结构体成员赋引用如何？

```
package main

type S struct {
  M *int
}

func main() {
  var i int
  refStruct(i)
}

func refStruct(y int) (z S) {
  z.M = &y
  return z
}
```

分析：refStruct函数对y取了引用，所以y发生了逃逸。

示例5：

```
package main

type S struct {
  M *int
}

func main() {
  var i int
  refStruct(&i)
}

func refStruct(y *int) (z S) {
  z.M = y
```

```
  return z
}
```

分析：在main函数里对i取了引用，并且把它传给了refStruct函数，i的引用一直在main函数的作用域用，因此i没有发生逃逸。和上一个例子相比，有一点小差别，但是导致的程序效果是不同的：例子4中，i先在main的栈帧中分配，之后又在refStruct栈帧中分配，然后又逃逸到堆上，到堆上分配了一次，共3次分配。本例中，i只分配了一次，然后通过引用传递。

示例6：

```
package main

type S struct {
  M *int
}

func main() {
  var x S
  var i int
  ref(&i, &x)
}

func ref(y *int, z *S) {
  z.M = y
}
```

分析：本例i发生了逃逸，按照前面例子5的分析，i不会逃逸。两个例子的区别是例子5中的S是在返回值里的，输入只能“流入”到输出，本例中的S是在输入参数中，所以逃逸分析失败，i要逃逸到堆上。

# 反射

# Go 语言中反射有哪些应用

Go 语言中反射的应用非常广：IDE 中的代码自动补全功能、对象序列化（encoding/json）、fmt 相关函数的实现、ORM（全称是：Object Relational Mapping，对象关系映射）.....

# Go 语言如何实现反射

`interface`，它是 Go 语言实现抽象的一个非常强大的工具。当向接口变量赋予一个实体类型的时候，接口会存储实体的类型信息，反射就是通过接口的类型信息实现的，反射建立在类型的基础上。

Go 语言在 reflect 包里定义了各种类型，实现了反射的各种函数，通过它们可以在运行时检测类型的信息、改变类型的值。

## types 和 interface

Go 语言中，每个变量都有一个静态类型，在编译阶段就确定了的，比如 `int, float64, []int` 等等。注意，这个类型是声明时候的类型，不是底层数据类型。

Go 官方博客里就举了一个例子：

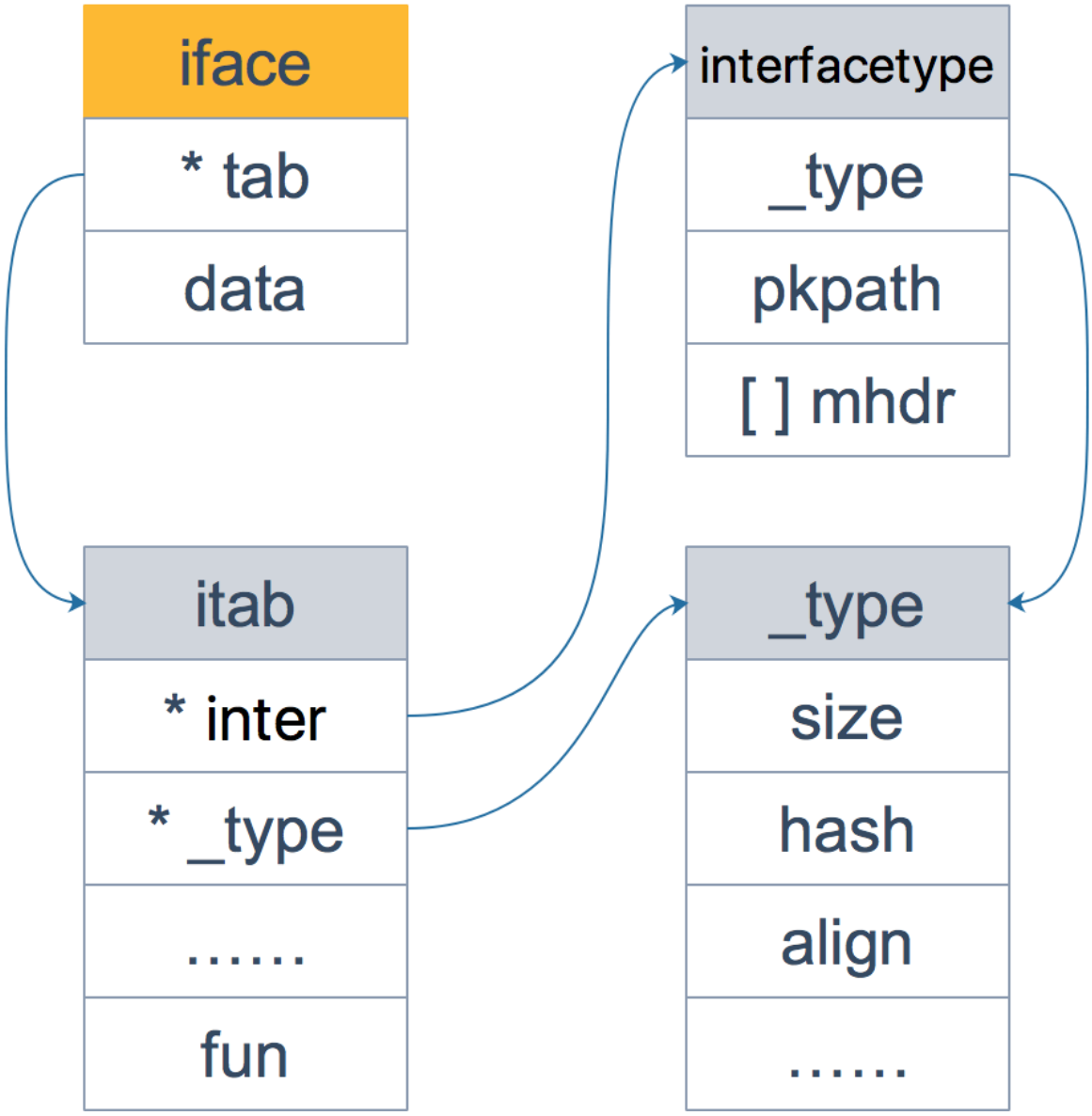
```
type MyInt int

var i int
var j MyInt
```

尽管 i，j 的底层类型都是 int，但我们知道，他们是不同的静态类型，除非进行类型转换，否则，i 和 j 不能同时出现在等号两侧。j 的静态类型就是 `MyInt`。

反射主要与 interface{} 类型相关。关于 interface 的底层结构，可以参考前面有关 interface 章节的内容，这里复习一下。

```
type iface struct {
    tab  *itab
    data unsafe.Pointer
}

type itab struct {
    inter  *interfacetype
    _type  *_type
    link   *itab
    hash   uint32
    bad    bool
    inhash bool
    unused [2]byte
    fun    [1]uintptr
}
```

其中 `itab` 由具体类型 `_type` 以及 `interfacetype` 组成。`_type` 表示具体类型，而 `interfacetype` 则表示具体类型实现的接口类型。




实际上，iface 描述的是非空接口，它包含方法；与之相对的是 `eface`，描述的是空接口，不包含任何方法，Go 语言里有的类型都 `“实现了”` 空接口。

```
type eface struct {
    _type *_type
    data  unsafe.Pointer
}
```

相比 `iface`，`eface` 就比较简单了。只维护了一个 `_type` 字段，表示空接口所承载的具体的实体类型。`data` 描述了具体的值。


还是用 Go 官方关于反射的博客里的例子，当然，我会用图形来详细解释，结合两者来看会更清楚。顺便提一下，搞技术的不要害怕英文资料，要想成为技术专家，读英文原始资料是技术提高的一条必经之路。

先明确一点：接口变量可以存储任何实现了接口定义的所有方法的变量。

Go 语言中最常见的就是 `Reader` 和 `Writer` 接口：

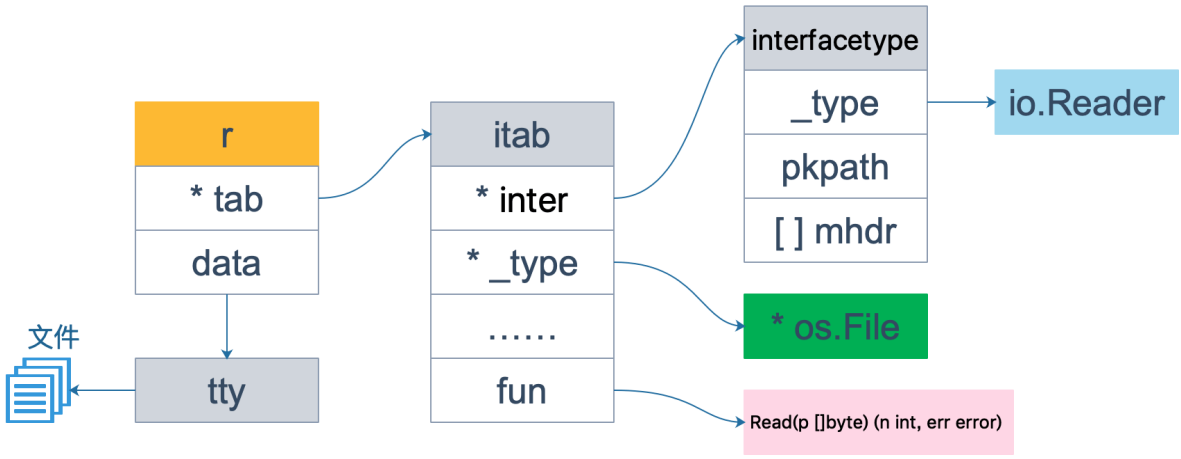
```
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
    Write(p []byte) (n int, err error)
}
```

接下来，就是接口之间的各种转换和赋值了：

```
var r io.Reader
tty, err := os.OpenFile("/Users/qcrao/Desktop/test", os.O_RDWR, 0)
if err != nil {
    return nil, err
}
r = tty
```

首先声明 `r` 的类型是 `io.Reader`，注意，这是 `r` 的静态类型，此时它的动态类型为 `nil`，并且它的动态值也是 `nil`。

之后，`r = tty` 这一语句，将 `r` 的动态类型变成 `*os.File`，动态值则变成非空，表示打开的文件对象。这时，r 可以用 `<value, type>` 对来表示为：`<tty, *os.File>`。

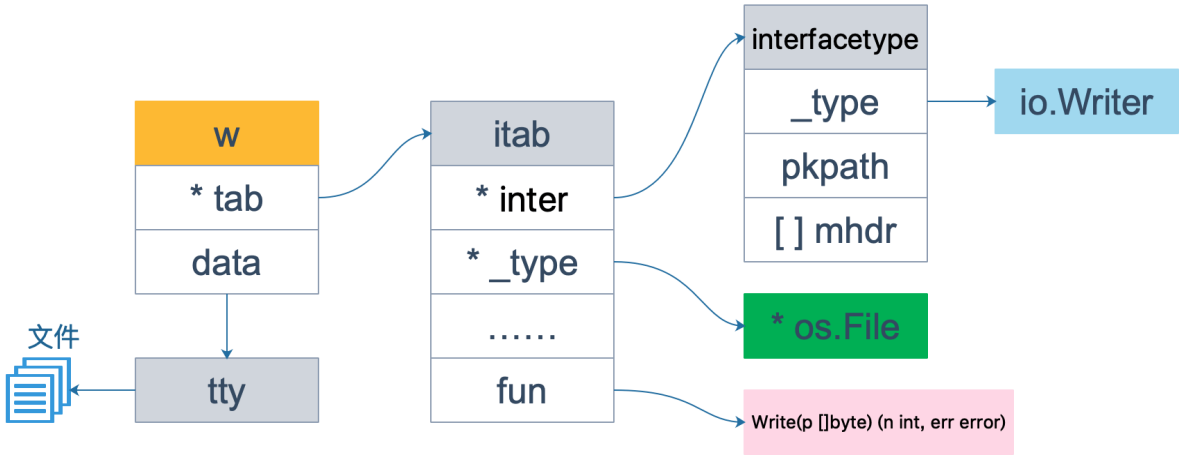



注意看上图，此时虽然 `fun` 所指向的函数只有一个 `Read` 函数，其实 `*os.File` 还包含 `Write` 函数，也就是说 `*os.File` 其实还实现了 `io.Writer` 接口。因此下面的断言语句可以执行：

```
var w io.Writer
w = r.(io.Writer)
```

之所以用断言，而不能直接赋值，是因为 `r` 的静态类型是 `io.Reader`，并没有实现 `io.Writer` 接口。断言能否成功，看 `r` 的动态类型是否符合要求。

这样，w 也可以表示成 `<tty, *os.File>`，仅管它和 `r` 一样，但是 w 可调用的函数取决于它的静态类型 `io.Writer`，也就是说它只能有这样的调用形式：`w.Write()`。`w` 的内存形式如下图：

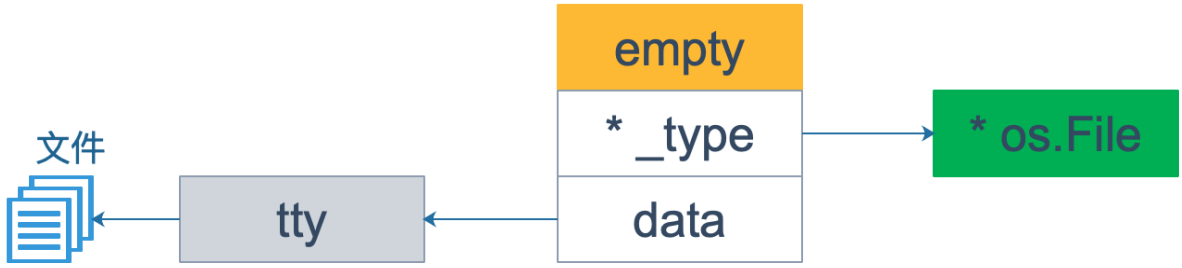



和 `r` 相比，仅仅是 `fun` 对应的函数变了：`Read -> Write`。

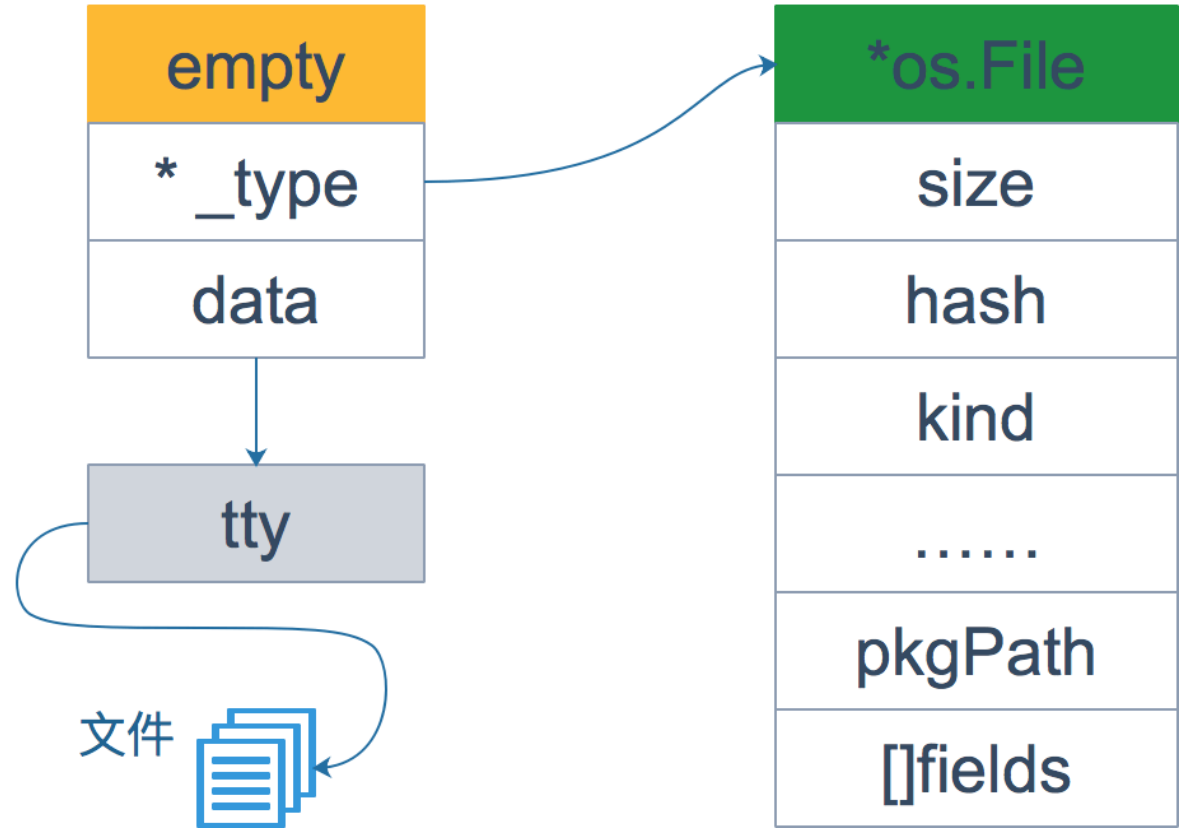
最后，再来一个赋值：

```
var empty interface{}
empty = w
```

由于 `empty` 是一个空接口，因此所有的类型都实现了它，w 可以直接赋给它，不需要执行断言操作。




从上面的三张图可以看到，interface 包含三部分信息：`_type` 是类型信息，`*data` 指向实际类型的实际值，`itab` 包含实际类型的信息，包括大小、包路径，还包含绑定在类型上的各种方法（图上没有画出方法），补充一下关于 os.File 结构体的图：




这一节的最后，展示一个技巧：

先参考源码，分别定义一个 “伪装” 的 iface 和 eface 结构体。

```
type iface struct {
    tab  *itab
    data unsafe.Pointer
}
type itab struct {
    inter uintptr
    _type uintptr
    link uintptr
    hash  uint32
    _     [4]byte
    fun   [1]uintptr
}

type eface struct {
    _type uintptr
```

```
    data unsafe.Pointer
}
```

接着，将接口变量占据的内存内容强制解释成上面定义的类型，再打印出来：

```
package main

import (
    "os"
    "fmt"
    "io"
    "unsafe"
)

func main() {
    var r io.Reader
    fmt.Printf("initial r: %T, %v\n", r, r)

    tty, _ := os.OpenFile("/Users/qcrao/Desktop/test", os.O_RDWR, 0)
    fmt.Printf("tty: %T, %v\n", tty, tty)

    // 给 r 赋值
    r = tty
    fmt.Printf("r: %T, %v\n", r, r)

    rIface := (*iface)(unsafe.Pointer(&r))
    fmt.Printf("r: iface.tab._type = %#x, iface.data = %#x\n", rIface.tab._type, rIface.data)

    // 给 w 赋值
    var w io.Writer
    w = r.(io.Writer)
    fmt.Printf("w: %T, %v\n", w, w)

    wIface := (*iface)(unsafe.Pointer(&w))
    fmt.Printf("w: iface.tab._type = %#x, iface.data = %#x\n", wIface.tab._type, wIface.data)

    // 给 empty 赋值
    var empty interface{}
    empty = w
    fmt.Printf("empty: %T, %v\n", empty, empty)

    emptyEface := (*eface)(unsafe.Pointer(&empty))
    fmt.Printf("empty: eface._type = %#x, eface.data = %#x\n", emptyEface._type, emptyEface.data)
}
```

运行结果：

```
initial r: <nil>, <nil>
tty: *os.File, &{0xc4200820f0}
r: *os.File, &{0xc4200820f0}
r: iface.tab._type = 0x10bfcc0, iface.data = 0xc420080020
w: *os.File, &{0xc4200820f0}
w: iface.tab._type = 0x10bfcc0, iface.data = 0xc420080020
empty: *os.File, &{0xc4200820f0}
empty: eface._type = 0x10bfcc0, eface.data = 0xc420080020
```

`r, w, empty` 的动态类型和动态值都一样。不再详细解释了，结合前面的图可以看得非常清晰。

# 反射的基本函数

reflect 包里定义了一个接口和一个结构体，即 `reflect.Type` 和 `reflect.Value` ，它们提供很多函数来获取存储在接口里的类型信息。

`reflect.Type` 主要提供关于类型相关的信息，所以它和 `_type` 关联比较紧密； `reflect.Value` 则结合 `_type` 和 `data` 两者，因此程序员可以获取甚至改变类型的值。

reflect 包中提供了两个基础的关于反射的函数来获取上述的接口和结构体：

```
func TypeOf(i interface{}) Type
func ValueOf(i interface{}) Value
```

`TypeOf` 函数用来提取一个接口中值的类型信息。由于它的输入参数是一个空的 `interface{}` ，调用此函数时，实参会先被转化为 `interface{}` 类型。这样，实参的类型信息、方法集、值信息都存储到 `interface{}` 变量里了。

看下源码：

```
func TypeOf(i interface{}) Type {
    eface := *(*emptyInterface)(unsafe.Pointer(&i))
    return toType(eface.typ)
}
```

这里的 `emptyInterface` 和上面提到的 `eface` 是一回事（字段名略有差异，字段是相同的），并且在不同的源码包：前者在 `reflect` 包，后者在 `runtime` 包。 `eface.typ` 就是动态类型。

```
type emptyInterface struct {
    typ  *rtype
    word unsafe.Pointer
}
```

至于 `toType` 函数，只是做了一个类型转换：

```
func toType(t *rtype) Type {
    if t == nil {
        return nil
    }
    return t
}
```

注意，返回值 `Type` 实际上是一个接口，定义了很多方法，用来获取类型相关的各种信息，而 `*rtype` 实现了 `Type` 接口。

```
type Type interface {
    // 所有的类型都可以调用下面这些函数

    // 此类型的变量对齐后所占用的字节数
    Align() int

    // 如果是 struct 的字段，对齐后占用的字节数
    FieldAlign() int

    // 返回类型方法集里的第 `i` (传入的参数)个方法
    Method(int) Method
```

```
    // 通过名称获取方法
    MethodByName(string) (Method, bool)

    // 获取类型方法集里导出的方法个数
    NumMethod() int

    // 类型名称
    Name() string

    // 返回类型所在的路径，如：encoding/base64
    PkgPath() string

    // 返回类型的大小，和 unsafe.Sizeof 功能类似
    Size() uintptr

    // 返回类型的字符串表示形式
    String() string

    // 返回类型的类型值
    Kind() Kind

    // 类型是否实现了接口 u
    Implements(u Type) bool

    // 是否可以赋值给 u
    AssignableTo(u Type) bool

    // 是否可以类型转换成 u
    ConvertibleTo(u Type) bool

    // 类型是否可以比较
    Comparable() bool

    // 下面这些函数只有特定类型可以调用
    // 如：Key, Elem 两个方法就只能是 Map 类型才能调用

    // 类型所占据的位数
    Bits() int

    // 返回通道的方向，只能是 chan 类型调用
    ChanDir() ChanDir

    // 返回类型是否是可变参数，只能是 func 类型调用
    // 比如 t 是类型 func(x int, y ... float64)
    // 那么 t.IsVariadic() == true
    IsVariadic() bool

    // 返回内部子元素类型，只能由类型 Array, Chan, Map, Ptr, or Slice 调用
    Elem() Type

    // 返回结构体类型的第 i 个字段，只能是结构体类型调用
    // 如果 i 超过了总字段数，就会 panic
    Field(i int) StructField

    // 返回嵌套的结构体的字段
    FieldByIndex(index []int) StructField

    // 通过字段名称获取字段
    FieldByName(name string) (StructField, bool)

    // FieldByNameFunc returns the struct field with a name
    // 返回名称符合 func 函数的字段
```

```
    FieldByNameFunc(match func(string) bool) (StructField, bool)

    // 获取函数类型的第 i 个参数的类型
    In(i int) Type

    // 返回 map 的 key 类型，只能由类型 map 调用
    Key() Type

    // 返回 Array 的长度，只能由类型 Array 调用
    Len() int

    // 返回类型字段的数量，只能由类型 Struct 调用
    NumField() int

    // 返回函数类型的输入参数个数
    NumIn() int

    // 返回函数类型的返回值个数
    NumOut() int

    // 返回函数类型的第 i 个值的类型
    Out(i int) Type

    // 返回类型结构体的相同部分
    common() *rtype

    // 返回类型结构体的不同部分
    uncommon() *uncommonType
}
```

可见 `Type` 定义了非常多的方法，通过它们可以获取类型的一切信息，大家一定要完整的过一遍上面所有的方法。

注意到 `Type` 方法集的倒数第二个方法 `common`，返回的 `rtype` 类型，它和上一篇文章讲到的 `_type` 是一回事，而且源代码里也注释了：两边要保持同步：

```
// rtype must be kept in sync with ../runtime/type.go:/^type._type.
```

```
type rtype struct {
    size       uintptr
    ptrdata    uintptr
    hash       uint32
    tflag      tflag
    align      uint8
    fieldAlign uint8
    kind       uint8
    alg        *typeAlg
    gcdata     *byte
    str        nameOff
    ptrToThis  typeOff
}
```

所有的类型都会包含 `rtype` 这个字段，表示各种类型的公共信息；另外，不同类型包含自己的一些独特的部分。

比如下面的 `arrayType` 和 `chanType` 都包含 `rytpe`，而前者还包含 slice，len 等和数组相关的信息；后者则包含 `dir` 表示通道方向的信息。

```
// arrayType represents a fixed array type.
type arrayType struct {
```

```
    rtype `reflect:"array"`
    elem  *rtype // array element type
    slice *rtype // slice type
    len   uintptr
}

// chanType represents a channel type.
type chanType struct {
    rtype `reflect:"chan"`
    elem  *rtype  // channel element type
    dir   uintptr // channel direction (ChanDir)
}
```

注意到，`Type` 接口实现了 `String()` 函数，满足 `fmt.Stringer` 接口，因此使用 `fmt.Println` 打印的时候，输出的是 `String()` 的结果。另外，`fmt.Printf()` 函数，如果使用 `%T` 来作为格式参数，输出的是 `reflect.TypeOf` 的结果，也就是动态类型。例如：

```
fmt.Printf("%T", 3) // int
```

---

讲完了 `TypeOf` 函数，再来看一下 `ValueOf` 函数。返回值 `reflect.Value` 表示 `interface{}` 里存储的实际变量，它能提供实际变量的各种信息。相关的方法常常是需要结合类型信息和值信息。例如，如果要提取一个结构体的字段信息，那就需要用到 _type (具体到这里是指 structType) 类型持有的关于结构体的字段信息、偏移信息，以及 `*data` 所指向的内容 —— 结构体的实际值。

源码如下：

```
func ValueOf(i interface{}) Value {
    if i == nil {
        return Value{}
    }

    // ......
    return unpackEface(i)
}

// 分解 eface
func unpackEface(i interface{}) Value {
    e := (*emptyInterface)(unsafe.Pointer(&i))

    t := e.typ
    if t == nil {
        return Value{}
    }

    f := flag(t.Kind())
    if ifaceIndir(t) {
        f |= flagIndir
    }
    return Value{t, e.word, f}
}
```

从源码看，比较简单：将先将 `i` 转换成 `*emptyInterface` 类型，再将它的 `typ` 字段和 `word` 字段以及一个标志位字段组装成一个 `Value` 结构体，而这就是 `ValueOf` 函数的返回值，它包含类型结构体指针、真实数据的地址、标志位。

Value 结构体定义了很多方法，通过这些方法可以直接操作 Value 字段 ptr 所指向的实际数据：

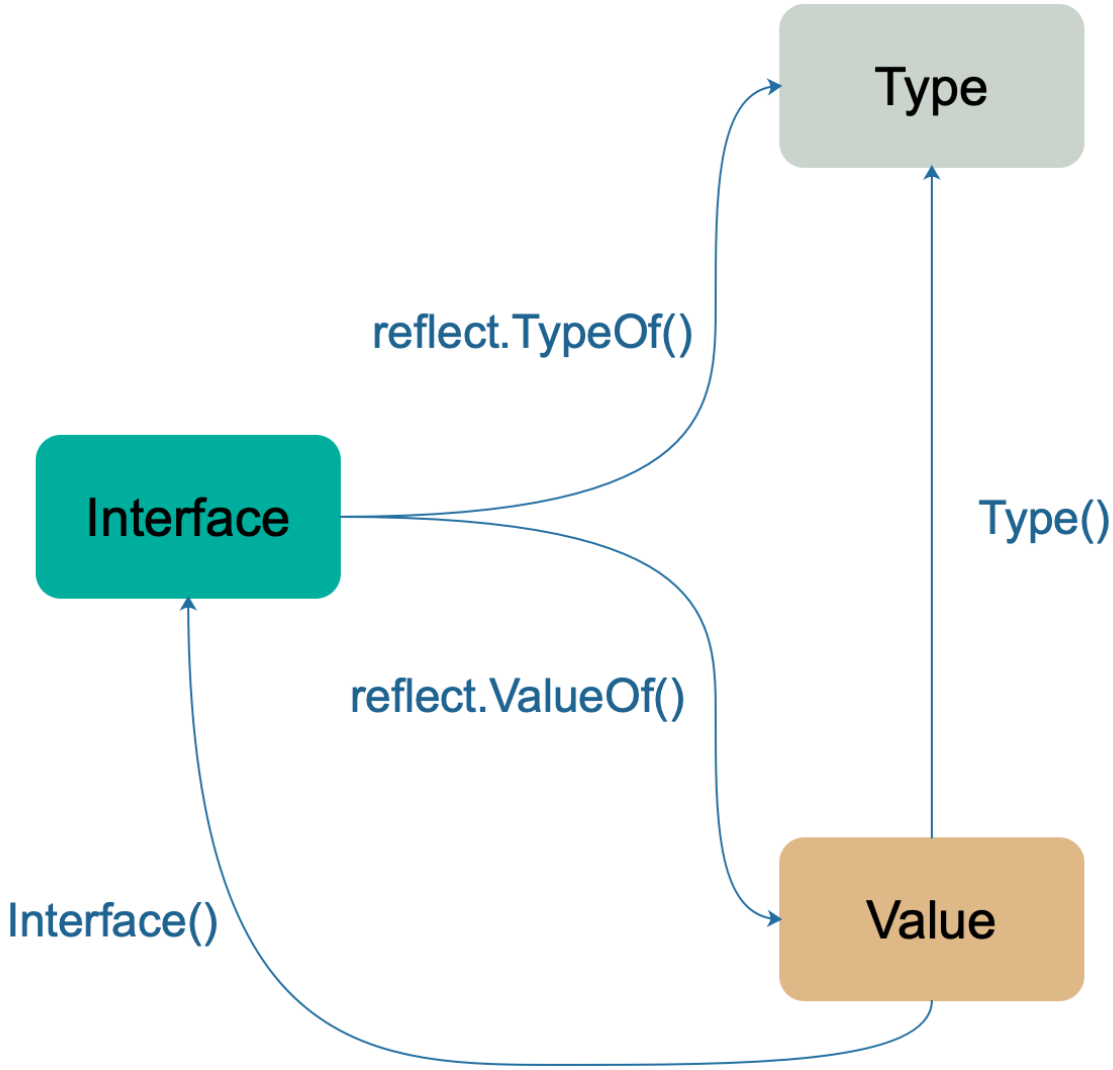
```
// 设置切片的 len 字段，如果类型不是切片，就会panic
func (v Value) SetLen(n int)

// 设置切片的 cap 字段
func (v Value) SetCap(n int)

// 设置字典的 kv
func (v Value) SetMapIndex(key, val Value)

// 返回切片、字符串、数组的索引 i 处的值
func (v Value) Index(i int) Value

// 根据名称获取结构体的内部字段值
func (v Value) FieldByName(name string) Value

// ......
```

`Value` 字段还有很多其他的方法。例如：

```
// 用来获取 int 类型的值
func (v Value) Int() int64

// 用来获取结构体字段（成员）数量
func (v Value) NumField() int

// 尝试向通道发送数据（不会阻塞）
func (v Value) TrySend(x reflect.Value) bool

// 通过参数列表 in 调用 v 值所代表的函数（或方法
func (v Value) Call(in []Value) (r []Value)

// 调用变参长度可变的函数
func (v Value) CallSlice(in []Value) []Value
```

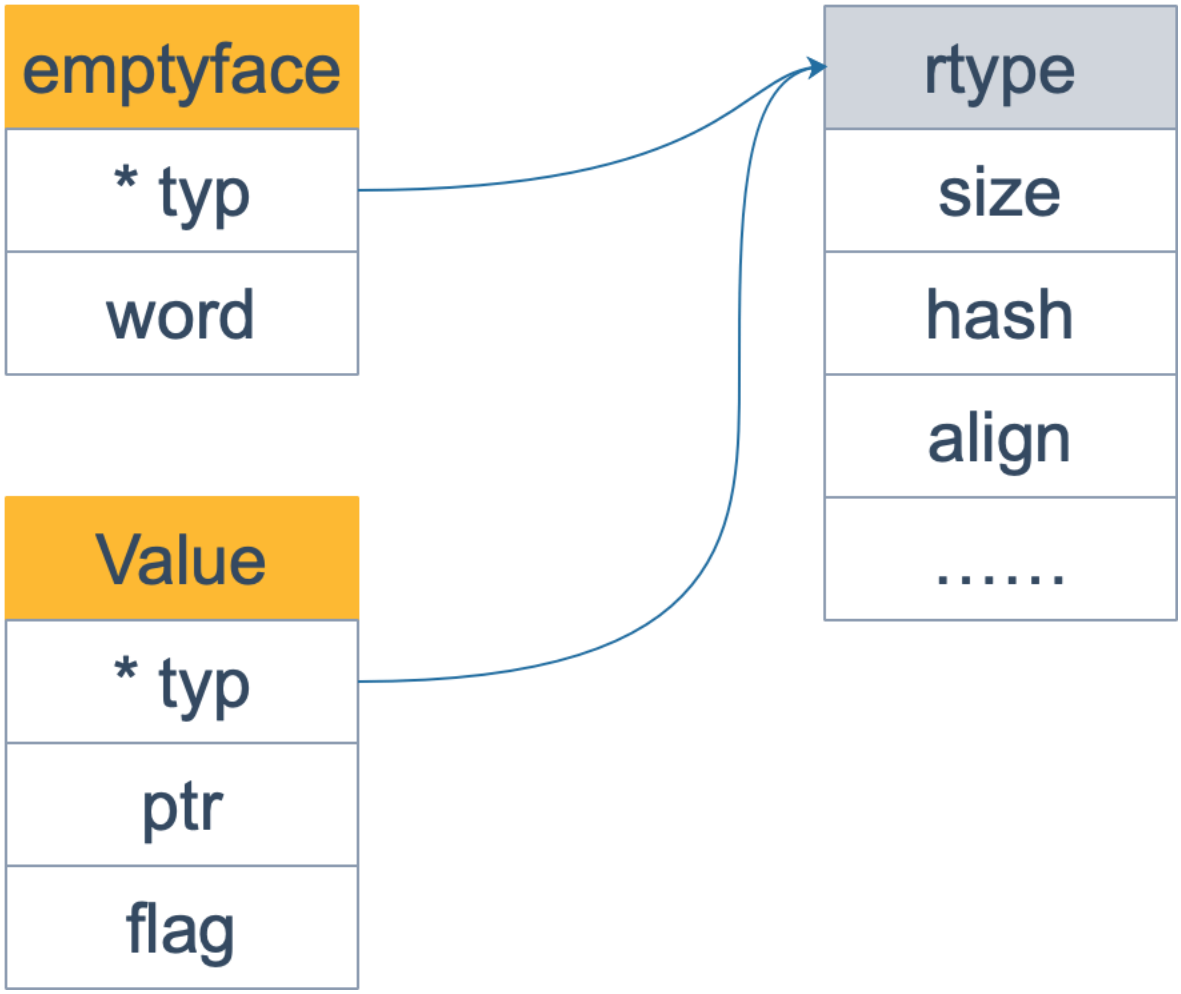
不一一列举了，反正是非常多。可以去 `src/reflect/value.go` 去看看源码，搜索 `func (v Value)` 就能看到。

另外，通过 `Type()` 方法和 `Interface()` 方法可以打通 `interface` 、 `Type` 、 `Value` 三者。Type() 方法也可以返回变量的类型信息，与 reflect.TypeOf() 函数等价。Interface() 方法可以将 Value 还原成原来的 interface。




总结一下：`TypeOf()` 函数返回一个接口，这个接口定义了一系列方法，利用这些方法可以获取关于类型的所有信息；`ValueOf()` 函数返回一个结构体变量，包含类型信息以及实际值。

用一张图来串一下：




上图中，`rtye` 实现了 `Type` 接口，是所有类型的公共部分。emptyface 结构体和 eface 其实是一个东西，而 rtype 其实和 _type 是一个东西，只是一些字段稍微有点差别，比如 emptyface 的 word 字段和 eface 的 data 字段名称不同，但是数据型是一样的。

## 反射的三大定律

根据 Go 官方关于反射的博客，反射有三大定律：

> 1. Reflection goes from interface value to reflection object.

> 2. Reflection goes from reflection object to interface value.

> 3. To modify a reflection object, the value must be settable.

第一条是最基本的：反射是一种检测存储在 `interface` 中的类型和值机制。这可以通过 `TypeOf` 函数和 `ValueOf` 函数得到。

第二条实际上和第一条是相反的机制，它将 `ValueOf` 的返回值通过 `Interface()` 函数反向转变成 `interface` 变量。

前两条就是说 `接口型变量` 和 `反射类型对象` 可以相互转化，反射类型对象实际上就是指的前面说的 `reflect.Type` 和 `reflect.Value` 。

第三条不太好懂：如果需要操作一个反射变量，那么它必须是可设置的。反射变量可设置的本质是它存储了原变量本身，这样对反射变量的操作，就会反映到原变量本身；反之，如果反射变量不能代表原变量，那么操作了反射变量，不会对原变量产生任何影响，这会给使用者带来疑惑。所以第二种情况在语言层面是不被允许的。

举一个经典例子：

```
var x float64 = 3.4
v := reflect.ValueOf(x)
v.SetFloat(7.1) // Error: will panic.
```

执行上面的代码会产生 panic，原因是反射变量 `v` 不能代表 `x` 本身，为什么？因为调用 `reflect.ValueOf(x)` 这一行代码的时候，传入的参数在函数内部只是一个拷贝，是值传递，所以 `v` 代表的只是 `x` 的一个拷贝，因此对 `v` 进行操作是被禁止的。

可设置是反射变量 `Value` 的一个性质，但不是所有的 `Value` 都是可被设置的。

就像在一般的函数里那样，当我们想改变传入的变量时，使用指针就可以解决了。

```
var x float64 = 3.4
p := reflect.ValueOf(&x)
fmt.Println("type of p:", p.Type())
fmt.Println("settability of p:", p.CanSet())
```

输出是这样的：

```
type of p: *float64
settability of p: false
```

`p` 还不是代表 `x`，`p.Elem()` 才真正代表 `x`，这样就可以真正操作 `x` 了：

```
v := p.Elem()
v.SetFloat(7.1)
fmt.Println(v.Interface()) // 7.1
fmt.Println(x) // 7.1
```

关于第三条，记住一句话：如果想要操作原变量，反射变量 `Value` 必须要 hold 住原变量的地址才行。

# 什么情况下需要使用反射

使用反射的常见场景有以下两种：

1. 不能明确接口调用哪个函数，需要根据传入的参数在运行时决定。
2. 不能明确传入函数的参数类型，需要在运行时处理任意对象。

【引申1】不推荐使用反射的理由有哪些？

1. 与反射相关的代码，经常是难以阅读的。在软件工程中，代码可读性也是一个非常重要的指标。
2. Go 语言作为一门静态语言，编码过程中，编译器能提前发现一些类型错误，但是对于反射代码是无能为力的。所以包含反射相关的代码，很可能会运行很久，才会出错，这时候经常是直接 panic，可能会造成严重的后果。
3. 反射对性能影响还是比较大的，比正常代码运行速度慢一到两个数量级。所以，对于一个项目中处于运行效率关键位置的代码，尽量避免使用反射特性。

# 什么是反射

维基百科上反射的定义：

> 在计算机科学中，反射是指计算机程序在运行时（Run time）可以访问、检测和修改它本身状态或行为的一种能力。用比喻来说，反射就是程序在运行的时候能够“观察”并且修改自己的行为。

难道不用反射就不能在运行时访问、检测和修改它本身的状态和行为吗？

问题的回答，其实要首先理解什么叫访问、检测和修改它本身状态或行为，它的本质是什么？

实际上，它的本质是程序在运行期探知对象的类型信息和内存结构。不用反射能行吗？可以的！使用汇编语言，直接和内层打交道，可以获取任何信息？但是，当编程迁移到高级语言上来之后，就不行了！只能通过 `反射` 来达到此项技能。

不同语言的反射模型不尽相同，有些语言还不支持反射。《Go 语言圣经》中是这样定义反射的：

> Go 语言提供了一种机制在运行时更新变量和检查它们的值、调用它们的方法，但是在编译时并不知道这些变量的具体类型，这称为反射机制。

# 如何比较两个对象完全相同

Go 语言中提供了一个函数可以完成此项功能：

```
func DeepEqual(x, y interface{}) bool
```

`DeepEqual` 函数的参数是两个 `interface`，实际上也就是可以输入任意类型，输出 true 或者 flase 表示输入的两个变量是否是“深度”相等。

先明白一点，如果是不同的类型，即使是底层类型相同，相应的值也相同，那么两者也不是“深度”相等。

```
type MyInt int
type YourInt int

func main() {
    m := MyInt(1)
    y := YourInt(1)

    fmt.Println(reflect.DeepEqual(m, y)) // false
}
```

上面的代码中，m, y 底层都是 int，而且值都是 1，但是两者静态类型不同，前者是 `MyInt`，后者是 `YourInt`，因此两者不是“深度”相等。

在源码里，有对 DeepEqual 函数的非常清楚地注释，列举了不同类型，DeepEqual 的比较情形，这里做一个总结：

| 类型 | 深度相等情形 |
| --- | --- |
| Array | 相同索引处的元素“深度”相等 |
| Struct | 相应字段，包含导出和不导出，“深度”相等 |
| Func | 只有两者都是 nil 时 |
| Interface | 两者存储的具体值“深度”相等 |
| Map | 1、都为 nil；2、非空、长度相等，指向同一个 map 实体对象，或者相应的 key 指向的 value “深度”相等 |
| Pointer | 1、使用 == 比较的结果相等；2、指向的实体“深度”相等 |
| Slice | 1、都为 nil；2、非空、长度相等，<br>首元素指向同一个底层数组的相同元素，即 &x[0] == &y[0]<br>或者 相同索引处的元素“深度”相等 |
| numbers, bools, strings, and channels | 使用 == 比较的结果为真 |

一般情况下，DeepEqual 的实现只需要递归地调用 == 就可以比较两个变量是否是真的“深度”相等。

但是，有一些异常情况：比如 func 类型是不可比较的类型，只有在两个 func 类型都是 nil 的情况下，才是“深度”相等；float 类型，由于精度的原因，也是不能使用 == 比较的；包含 func 类型或者 float 类型的 struct， interface， array 等。

对于指针而言，当两个值相等的指针就是“深度”相等，因为两者指向的内容是相等的，即使两者指向的是 func 类型或者 float 类型，这种情况下不关心指针所指向的内容。

同样，对于指向相同 slice， map 的两个变量也是“深度”相等的，不关心 slice， map 具体的内容。

对于“有环”的类型，比如循环链表，比较两者是否“深度”相等的过程中，需要对已比较的内容作一个标记，一旦发现两个指针之前比较过，立即停止比较，并判定二者是深度相等的。这样做的原因是，及时停止比较，避免陷入无限循环。

来看源码：

```
func DeepEqual(x, y interface{}) bool {
    if x == nil || y == nil {
        return x == y
    }
    v1 := ValueOf(x)
    v2 := ValueOf(y)
    if v1.Type() != v2.Type() {
        return false
    }
    return deepValueEqual(v1, v2, make(map[visit]bool), 0)
}
```

首先查看两者是否有一个是 nil 的情况，这种情况下，只有两者都是 nil，函数才会返回 true

接着，使用反射，获取x，y 的反射对象，并且立即比较两者的类型，根据前面的内容，这里实际上是动态类型，如果类型不同，直接返回 false。

最后，最核心的内容在子函数 `deepValueEqual` 中。

代码比较长，思路却比较简单清晰：核心是一个 switch 语句，识别输入参数的不同类型，分别递归调用 deepValueEqual 函数，一直递归到最基本的数据类型，比较 int，string 等可以直接得出 true 或者 false，再一层层地返回，最终得到“深度”相等的比较结果。

实际上，各种类型的比较套路比较相似，这里就直接节选一个稍微复杂一点的 `map` 类型的比较：

```
// deepValueEqual 函数
// ……

case Map:
    if v1.IsNil() != v2.IsNil() {
        return false
    }
    if v1.Len() != v2.Len() {
        return false
    }
    if v1.Pointer() == v2.Pointer() {
        return true
    }
    for _, k := range v1.MapKeys() {
        val1 := v1.MapIndex(k)
        val2 := v2.MapIndex(k)
        if !val1.IsValid() || !val2.IsValid() || !deepValueEqual(v1.MapIndex(k), v2.MapIndex(k), visited, dep
th+1) {
            return false
```

```
        }
    }
    return true

// ......
```

和前文总结的表格里，比较 map 是否相等的思路比较一致，也不需要多说什么。说明一点，`visited` 是一个 map，记录递归过程中，比较过的“对”：

```
type visit struct {
    a1  unsafe.Pointer
    a2  unsafe.Pointer
    typ Type
}

map[visit]bool
```

比较过程中，一旦发现比较的“对”，已经在 map 里出现过的话，直接判定“深度”比较结果的是 `true` 。

# 数组与切片

# 切片作为函数参数

前面我们说到，slice 其实是一个结构体，包含了三个成员：len, cap, array。分别表示切片长度，容量，底层数据的地址。

当 slice 作为函数参数时，就是一个普通的结构体。其实很好理解：若直接传 slice，在调用者看来，实参 slice 并不会被函数中的操作改变；若传的是 slice 的指针，在调用者看来，是会被改变原 slice 的。

值的注意的是，不管传的是 slice 还是 slice 指针，如果改变了 slice 底层数组的数据，会反应到实参 slice 的底层数据。为什么能改变底层数组的数据？很好理解：底层数据在 slice 结构体里是一个指针，仅管 slice 结构体自身不会被改变，也就是说底层数据地址不会被改变。 但是通过指向底层数据的指针，可以改变切片的底层数据，没有问题。

通过 slice 的 array 字段就可以拿到数组的地址。在代码里，是直接通过类似 `s[i]=10` 这种操作改变 slice 底层数组元素值。

另外，值得注意的是，Go 语言的函数参数传递，只有值传递，没有引用传递。

来看一个代码片段：

```
package main

func main() {
    s := []int{1, 1, 1}
    f(s)
    fmt.Println(s)
}

func f(s []int) {
    // i只是一个副本，不能改变s中元素的值
    /*for _, i := range s {
        i++
    }
    */

    for i := range s {
        s[i] += 1
    }
}
```

运行一下，程序输出：

```
[2 2 2]
```

果真改变了原始 slice 的底层数据。这里传递的是一个 slice 的副本，在 `f` 函数中，`s` 只是 `main` 函数中 `s` 的一个拷贝。在 `f` 函数内部，对 `s` 的作用并不会改变外层 `main` 函数的 `s` 。

要想真的改变外层 `slice` ，只有将返回的新的 slice 赋值到原始 slice，或者向函数传递一个指向 slice 的指针。我们再来看一个例子：

```
package main

import "fmt"

func myAppend(s []int) []int {
    // 这里 s 虽然改变了，但并不会影响外层函数的 s
    s = append(s, 100)
    return s
```

```
}

func myAppendPtr(s *[]int) {
    // 会改变外层 s 本身
    *s = append(*s, 100)
    return
}

func main() {
    s := []int{1, 1, 1}
    newS := myAppend(s)

    fmt.Println(s)
    fmt.Println(newS)

    s = newS

    myAppendPtr(&s)
    fmt.Println(s)
}
```

运行结果：

```
[1 1 1]
[1 1 1 100]
[1 1 1 100 100]
```

`myAppend` 函数里，虽然改变了 `s`，但它只是一个值传递，并不会影响外层的 `s`，因此第一行打印出来的结果仍然是 `[1 1 1]`。

而 `newS` 是一个新的 `slice`，它是基于 `s` 得到的。因此它打印的是追加了一个 `100` 之后的结果：`[1 1 1 100]`。

最后，将 `newS` 赋值给了 `s`，`s` 这时才真正变成了一个新的slice。之后，再给 `myAppendPtr` 函数传入一个 `s 指针`，这回它真的被改变了：`[1 1 1 100 100]`。

# 切片的容量是怎样增长的

一般都是在向 slice 追加了元素之后，才会引起扩容。追加元素调用的是 `append` 函数。

先来看看 `append` 函数的原型：

```
func append(slice []Type, elems ...Type) []Type
```

append 函数的参数长度可变，因此可以追加多个值到 slice 中，还可以用 `...` 传入 slice，直接追加一个切片。

```
slice = append(slice, elem1, elem2)
slice = append(slice, anotherSlice...)
```

`append` 函数返回值是一个新的slice，Go编译器不允许调用了 append 函数后不使用返回值。

```
append(slice, elem1, elem2)
append(slice, anotherSlice...)
```

所以上面的用法是错的，不能编译通过。

使用 append 可以向 slice 追加元素，实际上是往底层数组添加元素。但是底层数组的长度是固定的，如果索引 `len-1` 所指向的元素已经是底层数组的最后一个元素，就没法再添加了。

这时，slice 会迁移到新的内存位置，新底层数组的长度也会增加，这样就可以放置新增的元素。同时，为了应对未来可能再次发生的 append 操作，新的底层数组的长度，也就是新 `slice` 的容量是留了一定的 `buffer` 的。否则，每次添加元素的时候，都会发生迁移，成本太高。

新 slice 预留的 `buffer` 大小是有一定规律的。网上大多数的文章都是这样描述的：

> 当原 slice 容量小于 `1024` 的时候，新 slice 容量变成原来的 `2` 倍；原 slice 容量超过 `1024`，新 slice 容量变成原来的 `1.25` 倍。

我在这里先说结论：以上描述是错误的。

为了说明上面的规律是错误的，我写了一小段玩具代码：

```
package main

import "fmt"

func main() {
    s := make([]int, 0)

    oldCap := cap(s)

    for i := 0; i < 2048; i++ {
        s = append(s, i)

        newCap := cap(s)

        if newCap != oldCap {
            fmt.Printf("[%d -> %4d] cap = %-4d  |  after append %-4d  cap = %-4d\n", 0, i-1, oldCap, i, newCap)
            oldCap = newCap
```

```
        }
    }
}
```

我先创建了一个空的 `slice`，然后，在一个循环里不断往里面 `append` 新的元素。然后记录容量的变化，并且每当容量发生变化的时候，记录下老的容量，以及添加完元素之后的容量，同时记下此时 `slice` 里的元素。这样，我就可以观察，新老 `slice` 的容量变化情况，从而找出规律。

运行结果：

```
[0 ->   -1] cap = 0    |  after append 0    cap = 1
[0 ->    0] cap = 1    |  after append 1    cap = 2
[0 ->    1] cap = 2    |  after append 2    cap = 4
[0 ->    3] cap = 4    |  after append 4    cap = 8
[0 ->    7] cap = 8    |  after append 8    cap = 16
[0 ->   15] cap = 16   |  after append 16   cap = 32
[0 ->   31] cap = 32   |  after append 32   cap = 64
[0 ->   63] cap = 64   |  after append 64   cap = 128
[0 ->  127] cap = 128  |  after append 128  cap = 256
[0 ->  255] cap = 256  |  after append 256  cap = 512
[0 ->  511] cap = 512  |  after append 512  cap = 1024
[0 -> 1023] cap = 1024 |  after append 1024 cap = 1280
[0 -> 1279] cap = 1280 |  after append 1280 cap = 1696
[0 -> 1695] cap = 1696 |  after append 1696 cap = 2304
```

在老 slice 容量小于1024的时候，新 slice 的容量的确是老 slice 的2倍。目前还算正确。

但是，当老 slice 容量大于等于 `1024` 的时候，情况就有变化了。当向 slice 中添加元素 `1280` 的时候，老 slice 的容量为 `1280`，之后变成了 `1696`，两者并不是 `1.25` 倍的关系（1696/1280=1.325）。添加完 `1696` 后，新的容量 `2304` 当然也不是 `1696` 的 `1.25` 倍。

可见，现在网上各种文章中的扩容策略并不正确。我们直接搬出源码：源码面前，了无秘密。

从前面汇编代码我们也看到了，向 slice 追加元素的时候，若容量不够，会调用 `growslice` 函数，所以我们直接看它的代码。

```
// go 1.9.5 src/runtime/slice.go:82
func growslice(et *_type, old slice, cap int) slice {
    // ......
    newcap := old.cap
    doublecap := newcap + newcap
    if cap > doublecap {
        newcap = cap
    } else {
        if old.len < 1024 {
            newcap = doublecap
        } else {
            for newcap < cap {
                newcap += newcap / 4
            }
        }
    }
    // ......

    capmem = roundupsize(uintptr(newcap) * ptrSize)
    newcap = int(capmem / ptrSize)
}
```

看到了吗？如果只看前半部分，现在网上各种文章里说的 `newcap` 的规律是对的。现实是，后半部分还对 `newcap` 作了一个 `内存对齐` ，这个和内存分配策略相关。进行内存对齐之后，新 slice 的容量是要 `大于等于` 老 slice 容量的 `2倍` 或者 `1.25倍` 。

之后，向 Go 内存管理器申请内存，将老 slice 中的数据复制过去，并且将 append 的元素添加到新的底层数组中。

最后，向 `growslice` 函数调用者返回一个新的 slice，这个 slice 的长度并没有变化，而容量却增大了。

【引申1】

来看一个例子，来源于这里

```
package main

import "fmt"

func main() {
    s := []int{5}
    s = append(s, 7)
    s = append(s, 9)
    x := append(s, 11)
    y := append(s, 12)
    fmt.Println(s, x, y)
}
```

| 代码 | 切片对应状态 |
| --- | --- |
| s := []int{5} | s 只有一个元素，`[5]` |
| s = append(s, 7) | s 扩容，容量变为2，`[5, 7]` |
| s = append(s, 9) | s 扩容，容量变为4，`[5, 7, 9]` 。注意，这时 s 长度是3，只有3个元素 |
| x := append(s, 11) | 由于 s 的底层数组仍然有空间，因此并不会扩容。这样，底层数组就变成了 `[5, 7, 9, 11]` 。注意，此时 s = `[5, 7, 9]` ，容量为4；x = `[5, 7, 9, 11]` ，容量为4。这里 s 不变 |
| y := append(s, 12) | 这里还是在 s 元素的尾部追加元素，由于 s 的长度为3，容量为4，所以直接在底层数组索引为3的地方填上12。结果：s = `[5, 7, 9]` ，y = `[5, 7, 9, 12]` ，x = `[5, 7, 9, 12]` ，x，y 的长度均为4，容量也均为4 |

所以最后程序的执行结果是：

```
[5 7 9] [5 7 9 12] [5 7 9 12]
```

这里要注意的是，append函数执行完后，返回的是一个全新的 slice，并且对传入的 slice 并不影响。

【引申2】

关于 `append` ，我们最后来看一个例子，来源于 Golang Slice的扩容规则。

```
package main

import "fmt"

func main() {
    s := []int{1,2}
    s = append(s,4,5,6)
    fmt.Printf("len=%d, cap=%d",len(s),cap(s))
}
```

运行结果是：

```
len=5, cap=6
```

如果按网上各种文章中总结的那样：小于原 slice 长度小于 1024 的时候，容量每次增加 1 倍。添加元素 4 的时候，容量变为 4；添加元素 5 的时候不变；添加元素 6 的时候容量增加 1 倍，变成 8。

那上面代码的运行结果就是：

```
len=5, cap=8
```

这是错误的！我们来仔细看看，为什么会这样，再次搬出代码：

```
// go 1.9.5 src/runtime/slice.go:82
func growslice(et *_type, old slice, cap int) slice {
    // ......
    newcap := old.cap
    doublecap := newcap + newcap
    if cap > doublecap {
        newcap = cap
    } else {
        // ......
    }
    // ......

    capmem = roundupsize(uintptr(newcap) * ptrSize)
    newcap = int(capmem / ptrSize)
}
```

这个函数的参数依次是 `元素的类型，老的 slice，新 slice 最小求的容量` 。

例子中 `s` 原来只有 2 个元素， `len` 和 `cap` 都为 2， `append` 了三个元素后，长度变为 5，容量最小要变成 5，即调用 `growslice` 函数时，传入的第三个参数应该为 5。即 `cap=5` 。而一方面， `doublecap` 是原 `slice` 容量的 2 倍，等于 4。满足第一个 `if` 条件，所以 `newcap` 变成了 5。

接着调用了 `roundupsize` 函数，传入 40。（代码中ptrSize是指一个指针的大小，在64位机上是8）

我们再看内存对齐，搬出 `roundupsize` 函数的代码：

```
// src/runtime/msize.go:13
func roundupsize(size uintptr) uintptr {
    if size < _MaxSmallSize {
        if size <= smallSizeMax-8 {
            return uintptr(class_to_size[size_to_class8[(size+smallSizeDiv-1)/smallSizeDiv]])
        } else {
            //......
        }
    }
    //......
}

const _MaxSmallSize = 32768
const smallSizeMax = 1024
const smallSizeDiv = 8
```

很明显，我们最终将返回这个式子的结果：

```
class_to_size[size_to_class8[(size+smallSizeDiv-1)/smallSizeDiv]]
```

这是 `Go` 源码中有关内存分配的两个 `slice` 。 `class_to_size` 通过 `spanClass` 获取 `span` 划分的 `object` 大小。而 `size_to_class8` 表示通过 `size` 获取它的 `spanClass` 。

```
var size_to_class8 = [smallSizeMax/smallSizeDiv + 1]uint8{0, 1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 10, 10, 11, 11, 12, 12, 13, 13, 14, 14, 15, 15, 16, 16, 17, 17, 18, 18, 18, 18, 19, 19, 19, 19, 20, 20, 20, 20, 21, 21, 21, 21, 22, 22, 22, 22, 23, 23, 23, 23, 24, 24, 24, 24, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 27, 27, 27, 27, 27, 27, 27, 27, 28, 28, 28, 28, 28, 28, 28, 28, 29, 29, 29, 29, 29, 29, 29, 29, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31}

var class_to_size = [_NumSizeClasses]uint16{0, 8, 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192, 208, 224, 240, 256, 288, 320, 352, 384, 416, 448, 480, 512, 576, 640, 704, 768, 896, 1024, 1152, 1280, 1408, 1536, 1792, 2048, 2304, 2688, 3072, 3200, 3456, 4096, 4864, 5376, 6144, 6528, 6784, 6912, 8192, 9472, 9728, 10240, 10880, 12288, 13568, 14336, 16384, 18432, 19072, 20480, 21760, 24576, 27264, 28672, 32768}
```

我们传进去的 `size` 等于 40。所以 `(size+smallSizeDiv-1)/smallSizeDiv = 5` ；获取 `size_to_class8` 数组中索引为 `5` 的元素为 `4` ；获取 `class_to_size` 中索引为 `4` 的元素为 `48` 。

最终，新的 slice 的容量为 `6` ：

```
newcap = int(capmem / ptrSize) // 6
```

至于，上面的两个 `魔法数组` 的由来，就不展开了。

【引申2】
向一个nil的slice添加元素会发生什么？为什么？

其实 `nil slice` 或者 `empty slice` 都是可以通过调用 append 函数来获得底层数组的扩容。最终都是调用 `mallocgc` 来向 Go 的内存管理器申请到一块内存，然后再赋给原来的 `nil slice` 或 `empty slice` ，然后摇身一变，成为“真正”的 `slice` 了。

# 数组和切片有什么异同

slice 的底层数据是数组，slice 是对数组的封装，它描述一个数组的片段。两者都可以通过下标来访问单个元素。

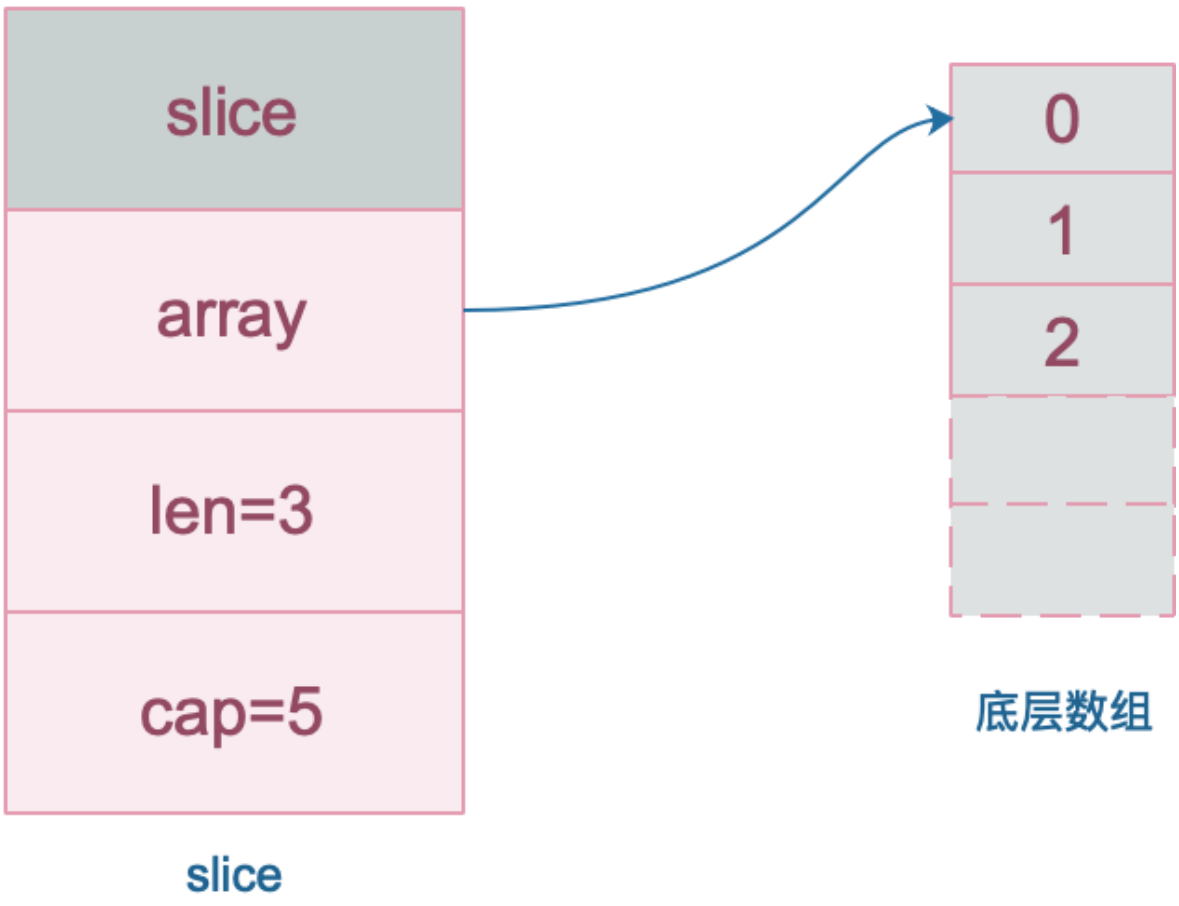
数组是定长的，长度定义好之后，不能再更改。在 Go 中，数组是不常见的，因为其长度是类型的一部分，限制了它的表达能力，比如 [3]int 和 [4]int 就是不同的类型。

而切片则非常灵活，它可以动态地扩容。切片的类型和长度无关。

数组就是一片连续的内存， slice 实际上是一个结构体，包含三个字段：长度、容量、底层数组。

```
// runtime/slice.go
type slice struct {
    array unsafe.Pointer // 元素指针
    len   int // 长度
    cap   int // 容量
}
```

slice 的数据结构如下：




注意，底层数组是可以被多个 slice 同时指向的，因此对一个 slice 的元素进行操作是有可能影响到其他 slice 的。

【引申1】
[3]int 和 [4]int 是同一个类型吗？

不是。因为数组的长度是类型的一部分，这是与 slice 不同的一点。

【引申2】
下面的代码输出是什么？

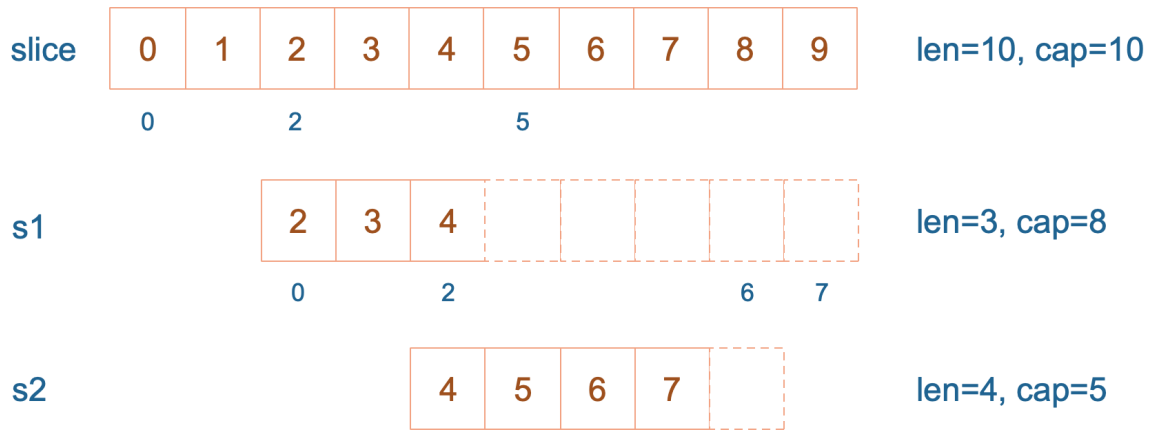
说明：例子来自雨痕大佬《Go学习笔记》第四版，P43页。这里我会进行扩展，并会作图详细分析。

```
package main

import "fmt"

func main() {
    slice := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
    s1 := slice[2:5]
    s2 := s1[2:6:7]

    s2 = append(s2, 100)
    s2 = append(s2, 200)

    s1[2] = 20

    fmt.Println(s1)
    fmt.Println(s2)
    fmt.Println(slice)
}
```

结果：

```
[2 3 20]
[4 5 6 7 100 200]
[0 1 2 3 20 5 6 7 100 9]
```

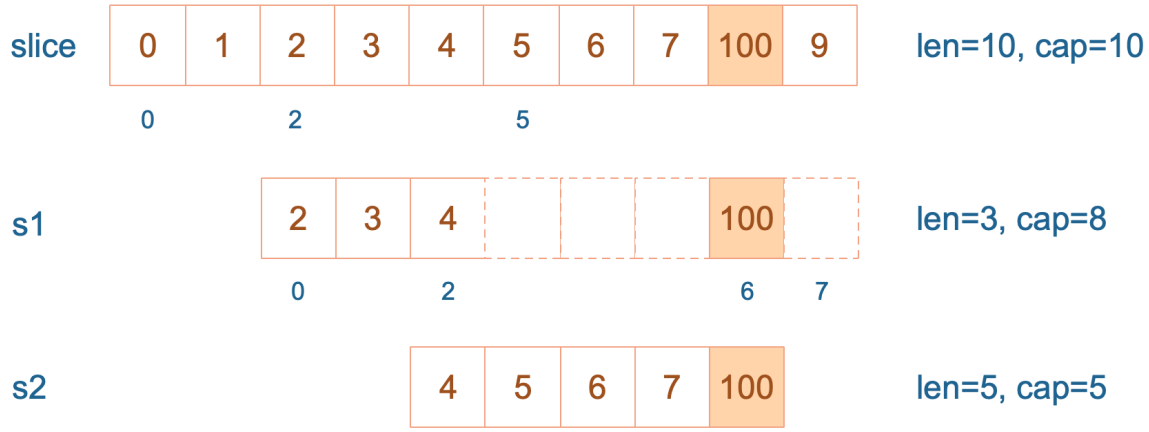
`s1` 从 `slice` 索引2（闭区间）到索引5（开区间，元素真正取到索引4），长度为3，容量默认到数组结尾，为8。
`s2` 从 `s1` 的索引2（闭区间）到索引6（开区间，元素真正取到索引5），容量到索引7（开区间，真正到索引6），为5。




接着，向 `s2` 尾部追加一个元素 100：

```
s2 = append(s2, 100)
```

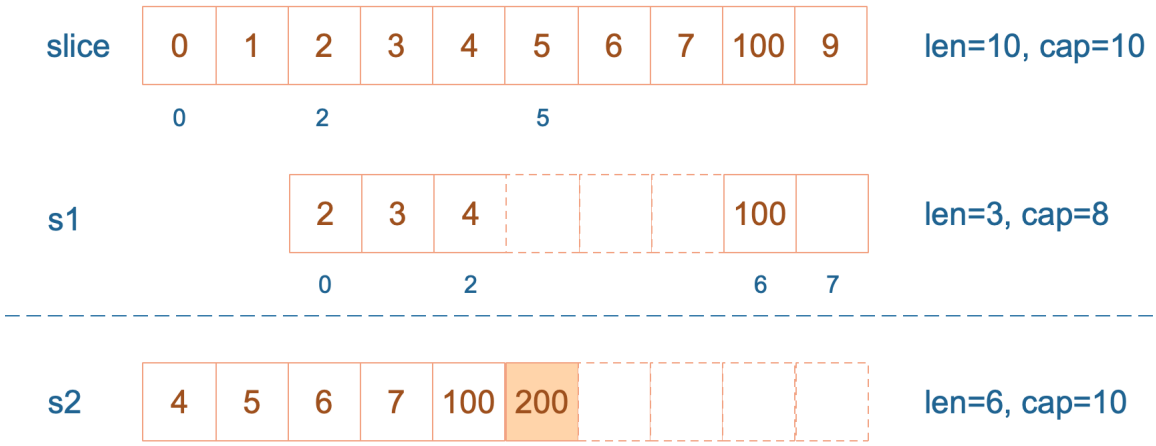
`s2` 容量刚好够，直接追加。不过，这会修改原始数组对应位置的元素。这一改动，数组和 `s1` 都可以看得到。




再次向 `s2` 追加元素200：

```
s2 = append(s2, 100)
```

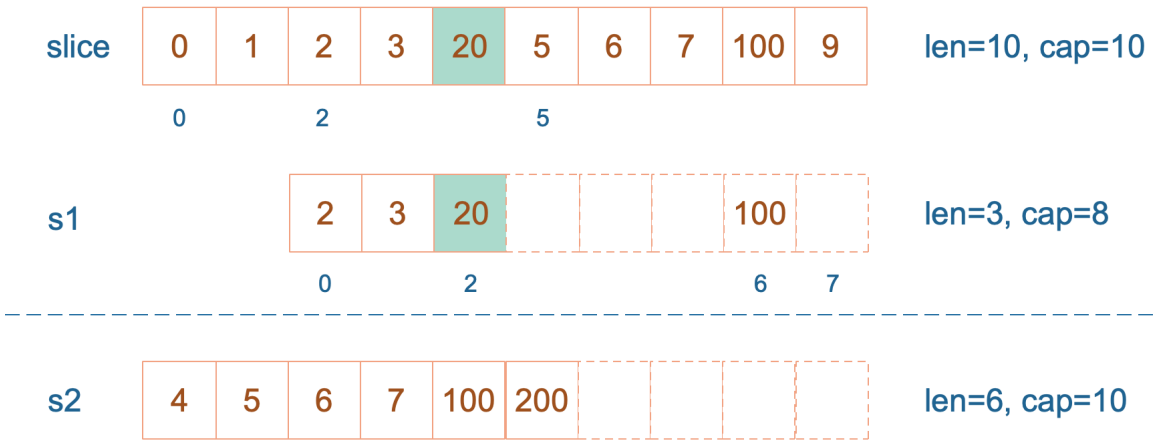
这时， `s2` 的容量不够用，该扩容了。于是， `s2` 另起炉灶，将原来的元素复制新的位置，扩大自己的容量。并且为了应对未来可能的 `append` 带来的再一次扩容， `s2` 会在此次扩容的时候多留一些 `buffer` ，将新的容量将扩大为原始容量的2倍，也就是10了。




最后，修改 `s1` 索引为2位置的元素：

```
s1[2] = 20
```

这次只会影响原始数组相应位置的元素。它影响不到 `s2` 了，人家已经远走高飞了。




再提一点，打印 `s1` 的时候，只会打印出 `s1` 长度以内的元素。所以，只会打印出3个元素，虽然它的底层数组不止3个元素。

# GC 的认识

# 什么是 GC，有什么作用？

## 1. 什么是 GC，有什么作用？

`GC`，全称 `Garbage Collection`，即垃圾回收，是一种自动内存管理的机制。

当程序向操作系统申请的内存不再需要时，垃圾回收主动将其回收并供其他代码进行内存申请时候复用，或者将其归还给操作系统，这种针对内存级别资源的自动回收过程，即为垃圾回收。而负责垃圾回收的程序组件，即为垃圾回收器。

垃圾回收其实一个完美的 “Simplicity is Complicated” 的例子。一方面，程序员受益于 GC，无需操心、也不再需要对内存进行手动的申请和释放操作，GC 在程序运行时自动释放残留的内存。另一方面，GC 对程序员几乎不可见，仅在程序需要进行特殊优化时，通过提供可调控的 API，对 GC 的运行时机、运行开销进行把控的时候才得以现身。

通常，垃圾回收器的执行过程被划分为两个半独立的组件：

- 赋值器（Mutator）：这一名称本质上是在指代用户态的代码。因为对垃圾回收器而言，用户态的代码仅仅只是在修改对象之间的引用关系，也就是在对象图（对象之间引用关系的一个有向图）上进行操作。
- 回收器（Collector）：负责执行垃圾回收的代码。

# 根对象到底是什么？

## 2. 根对象到底是什么？

根对象在垃圾回收的术语中又叫做根集合，它是垃圾回收器在标记过程时最先检查的对象，包括：

1. 全局变量：程序在编译期就能确定的那些存在于程序整个生命周期的变量。
2. 执行栈：每个 goroutine 都包含自己的执行栈，这些执行栈上包含栈上的变量及指向分配的堆内存区块的指针。
3. 寄存器：寄存器的值可能表示一个指针，参与计算的这些指针可能指向某些赋值器分配的堆内存区块。

# 常见的 GC 实现方式有哪些？Go 语言的 GC 使用的是什么？

## 3. 常见的 GC 实现方式有哪些？Go 语言的 GC 使用的是什么？

所有的 GC 算法其存在形式可以归结为追踪（Tracing）和引用计数（Reference Counting）这两种形式的混合运用。

- 追踪式 GC

  从根对象出发，根据对象之间的引用信息，一步步推进直到扫描完毕整个堆并确定需要保留的对象，从而回收所有可回收的对象。Go、 Java、V8 对 JavaScript 的实现等均为追踪式 GC。

- 引用计数式 GC

  每个对象自身包含一个被引用的计数器，当计数器归零时自动得到回收。因为此方法缺陷较多，在追求高性能时通常不被应用。Python、Objective-C 等均为引用计数式 GC。

目前比较常见的 GC 实现方式包括：

- 追踪式，分为多种不同类型，例如：
  - 标记清扫：从根对象出发，将确定存活的对象进行标记，并清扫可以回收的对象。
  - 标记整理：为了解决内存碎片问题而提出，在标记过程中，将对象尽可能整理到一块连续的内存上。
  - 增量式：将标记与清扫的过程分批执行，每次执行很小的部分，从而增量的推进垃圾回收，达到近似实时、几乎无停顿的目的。
  - 增量整理：在增量式的基础上，增加对对象的整理过程。
  - 分代式：将对象根据存活时间的长短进行分类，存活时间小于某个值的为年轻代，存活时间大于某个值的为老年代，永远不会参与回收的对象为永久代。并根据分代假设（如果一个对象存活时间不长则倾向于被回收，如果一个对象已经存活很长时间则倾向于存活更长时间）对对象进行回收。
- 引用计数：根据对象自身的引用计数来回收，当引用计数归零时立即回收。

关于各类方法的详细介绍及其实现不在本文中详细讨论。对于 Go 而言，Go 的 GC 目前使用的是无分代（对象没有代际之分）、不整理（回收过程中不对对象进行移动与整理）、并发（与用户代码并发执行）的三色标记清扫算法。原因[1]在于：

1. 对象整理的优势是解决内存碎片问题以及“允许”使用顺序内存分配器。但 Go 运行时的分配算法基于 tcmalloc，基本上没有碎片问题。 并且顺序内存分配器在多线程的场景下并不适用。Go 使用的是基于 tcmalloc 的现代内存分配算法，对对象进行整理不会带来实质性的性能提升。
2. 分代 GC 依赖分代假设，即 GC 将主要的回收目标放在新创建的对象上（存活时间短，更倾向于被回收），而非频繁检查所有对象。但 Go 的编译器会通过**逃逸分析**将大部分新生对象存储在栈上（栈直接被回收），只有那些需要长期存在的对象才会被分配到需要进行垃圾回收的堆中。也就是说，分代 GC 回收的那些存活时间短的对象在 Go 中是直接被分配到栈上，当 goroutine 死亡后栈也会被直接回收，不需要 GC 的参与，进而分代假设并没有带来直接优势。并且 Go 的垃圾回收器与用户代码并发执行，使得 STW 的时间与对象的代际、对象的 size 没有关系。Go 团队更关注于如何更好地让 GC 与用户代码并发执行（使用适当的 CPU 来执行垃圾回收），而非减少停顿时间这一单一目标上。

# 三色标记法是什么？

## 4. 三色标记法是什么？

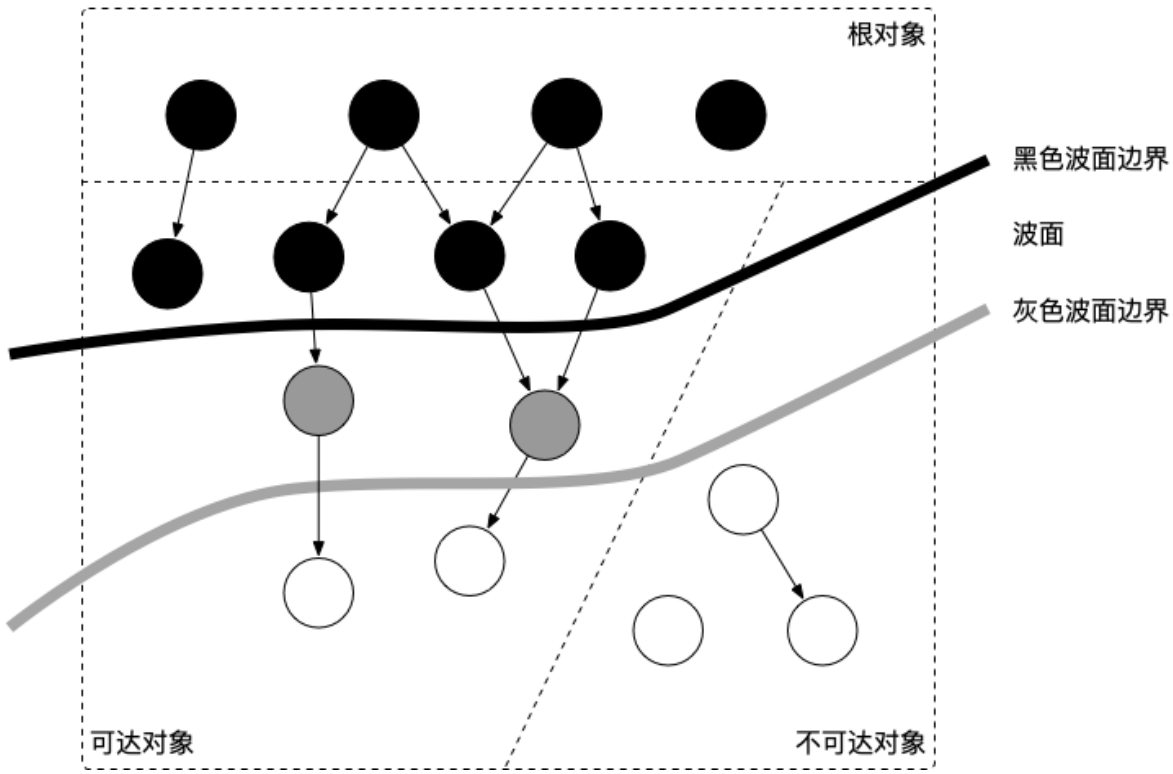
理解**三色标记法**的关键是理解对象的**三色抽象**以及**波面（wavefront）推进**这两个概念。三色抽象只是一种描述追踪式回收器的方法，在实践中并没有实际含义，它的重要作用在于从逻辑上严密推导标记清理这种垃圾回收方法的正确性。也就是说，当我们谈及三色标记法时，通常指标记清扫的垃圾回收。

从垃圾回收器的视角来看，三色抽象规定了三种不同类型的对象，并用不同的颜色相称：

- 白色对象（可能死亡）：未被回收器访问到的对象。在回收开始阶段，所有对象均为白色，当回收结束后，白色对象均不可达。
- 灰色对象（波面）：已被回收器访问到的对象，但回收器需要对其中的一个或多个指针进行扫描，因为他们可能还指向白色对象。
- 黑色对象（确定存活）：已被回收器访问到的对象，其中所有字段都已被扫描，黑色对象中任何一个指针都不可能直接指向白色对象。

这样三种不变性所定义的回收过程其实是一个**波面**不断前进的过程，这个波面同时也是黑色对象和白色对象的边界，灰色对象就是这个波面。

当垃圾回收开始时，只有白色对象。随着标记过程开始进行时，灰色对象开始出现（着色），这时候波面便开始扩大。当一个对象的所有子节点均完成扫描时，会被着色为黑色。当整个堆遍历完成时，只剩下黑色和白色对象，这时的黑色对象为可达对象，即存活；而白色对象为不可达对象，即死亡。这个过程可以视为以灰色对象为波面，将黑色对象和白色对象分离，使波面不断向前推进，直到所有可达的灰色对象都变为黑色对象为止的过程。如下图所示：




图中展示了根对象、可达对象、不可达对象，黑、灰、白对象以及波面之间的关系。

# STW 是什么意思？

## 5. STW 是什么意思？

`STW` 可以是 `Stop the World` 的缩写，也可以是 `Start the World` 的缩写。通常意义上指指代从 `Stop the World` 这一动作发生时到 `Start the World` 这一动作发生时这一段时间间隔，即万物静止。STW 在垃圾回收过程中为了保证实现的正确性、防止无止境的内存增长等问题而不可避免的需要停止赋值器进一步操作对象图的一段过程。

在这个过程中整个用户代码被停止或者放缓执行， `STW` 越长，对用户代码造成的影响（例如延迟）就越大，早期 Go 对垃圾回收器的实现中 `STW` 长达几百毫秒，对时间敏感的实时通信等应用程序会造成巨大的影响。我们来看一个例子：

```
package main

import (
    "runtime"
    "time"
)

func main() {
    go func() {
        for {
        }
    }()

    time.Sleep(time.Millisecond)
    runtime.GC()
    println("OK")
}
```

上面的这个程序在 Go 1.14 以前永远都不会输出 `OK` ，其罪魁祸首是进入 STW 这一操作的执行无限制的被延长。

尽管 STW 如今已经优化到了半毫秒级别以下，但这个程序被卡死原因是由于需要进入 STW 导致的。原因在于，GC 在需要进入 STW 时，需要通知并让所有的用户态代码停止，但是 `for {}` 所在的 goroutine 永远都不会被中断，从而始终无法进入 STW 阶段。实际实践中也是如此，当程序的某个 goroutine 长时间得不到停止，强行拖慢进入 STW 的时机，这种情况下造成的影响（卡死）是非常可怕的。好在自 Go 1.14 之后，这类 goroutine 能够被异步地抢占，从而使得进入 STW 的时间不会超过抢占信号触发的周期，程序也不会因为仅仅等待一个 goroutine 的停止而停顿在进入 STW 之前的操作上。

# 如何观察 Go GC？

## 6. 如何观察 Go GC？

我们以下面的程序为例，先使用四种不同的方式来介绍如何观察 GC，并在后面的问题中通过几个详细的例子再来讨论如何优化 GC。

```
package main

func allocate() {
    _ = make([]byte, 1<<20)
}

func main() {
    for n := 1; n < 100000; n++ {
        allocate()
    }
}
```

### 方式1：`GODEBUG=gctrace=1`

我们首先可以通过

```
$ go build -o main
$ GODEBUG=gctrace=1 ./main

gc 1 @0.000s 2%: 0.009+0.23+0.004 ms clock, 0.11+0.083/0.019/0.14+0.049 ms cpu, 4->6->2 MB, 5 MB goal, 12 P
scvg: 8 KB released
scvg: inuse: 3, idle: 60, sys: 63, released: 57, consumed: 6 (MB)
gc 2 @0.001s 2%: 0.018+1.1+0.029 ms clock, 0.22+0.047/0.074/0.048+0.34 ms cpu, 4->7->3 MB, 5 MB goal, 12 P
scvg: inuse: 3, idle: 60, sys: 63, released: 56, consumed: 7 (MB)
gc 3 @0.003s 2%: 0.018+0.59+0.011 ms clock, 0.22+0.073/0.008/0.042+0.13 ms cpu, 5->6->1 MB, 6 MB goal, 12 P
scvg: 8 KB released
scvg: inuse: 2, idle: 61, sys: 63, released: 56, consumed: 7 (MB)
gc 4 @0.003s 4%: 0.019+0.70+0.054 ms clock, 0.23+0.051/0.047/0.085+0.65 ms cpu, 4->6->2 MB, 5 MB goal, 12 P
scvg: 8 KB released
scvg: inuse: 3, idle: 60, sys: 63, released: 56, consumed: 7 (MB)
scvg: 8 KB released
scvg: inuse: 4, idle: 59, sys: 63, released: 56, consumed: 7 (MB)
gc 5 @0.004s 12%: 0.021+0.26+0.49 ms clock, 0.26+0.046/0.037/0.11+5.8 ms cpu, 4->7->3 MB, 5 MB goal, 12 P
scvg: inuse: 5, idle: 58, sys: 63, released: 56, consumed: 7 (MB)
gc 6 @0.005s 12%: 0.020+0.17+0.004 ms clock, 0.25+0.080/0.070/0.053+0.051 ms cpu, 5->6->1 MB, 6 MB goal, 12 P
scvg: 8 KB released
scvg: inuse: 1, idle: 62, sys: 63, released: 56, consumed: 7 (MB)
```

在这个日志中可以观察到两类不同的信息：

```
gc 1 @0.000s 2%: 0.009+0.23+0.004 ms clock, 0.11+0.083/0.019/0.14+0.049 ms cpu, 4->6->2 MB, 5 MB goal, 12 P
gc 2 @0.001s 2%: 0.018+1.1+0.029 ms clock, 0.22+0.047/0.074/0.048+0.34 ms cpu, 4->7->3 MB, 5 MB goal, 12 P
...
```

以及：

```
scvg: 8 KB released
scvg: inuse: 3, idle: 60, sys: 63, released: 57, consumed: 6 (MB)
scvg: inuse: 3, idle: 60, sys: 63, released: 56, consumed: 7 (MB)
...
```

对于用户代码向运行时申请内存产生的垃圾回收：

```
gc 2 @0.001s 2%: 0.018+1.1+0.029 ms clock, 0.22+0.047/0.074/0.048+0.34 ms cpu, 4->7->3 MB, 5 MB goal, 12 P
```

含义由下表所示：

| 字段 | 含义 |
| --- | --- |
| gc 2 | 第二个 GC 周期 |
| 0.001 | 程序开始后的 0.001 秒 |
| 2% | 该 GC 周期中 CPU 的使用率 |
| 0.018 | 标记开始时， STW 所花费的时间（wall clock） |
| 1.1 | 标记过程中，并发标记所花费的时间（wall clock） |
| 0.029 | 标记终止时， STW 所花费的时间（wall clock） |
| 0.22 | 标记开始时， STW 所花费的时间（cpu time） |
| 0.047 | 标记过程中，标记辅助所花费的时间（cpu time） |
| 0.074 | 标记过程中，并发标记所花费的时间（cpu time） |
| 0.048 | 标记过程中，GC 空闲的时间（cpu time） |
| 0.34 | 标记终止时， STW 所花费的时间（cpu time） |
| 4 | 标记开始时，堆的大小的实际值 |
| 7 | 标记结束时，堆的大小的实际值 |
| 3 | 标记结束时，标记为存活的对象大小 |
| 5 | 标记结束时，堆的大小的预测值 |
| 12 | P 的数量 |

> wall clock 是指开始执行到完成所经历的实际时间，包括其他程序和本程序所消耗的时间；
> cpu time 是指特定程序使用 CPU 的时间；
> 他们存在以下关系：

> - wall clock < cpu time: 充分利用多核
> - wall clock ≈ cpu time: 未并行执行
> - wall clock > cpu time: 多核优势不明显

对于运行时向操作系统申请内存产生的垃圾回收（向操作系统归还多余的内存）：

```
scvg: 8 KB released
scvg: inuse: 3, idle: 60, sys: 63, released: 57, consumed: 6 (MB)
```

含义由下表所示：

| 字段 | 含义 |
| --- | --- |
| 8 KB released | 向操作系统归还了 8 KB 内存 |
| 3 | 已经分配给用户代码、正在使用的总内存大小 (MB) |
| 60 | 空闲以及等待归还给操作系统的总内存大小（MB） |
| 63 | 通知操作系统中保留的内存大小（MB） |
| 57 | 已经归还给操作系统的（或者说还未正式申请）的内存大小（MB） |
| 6 | 已经从操作系统中申请的内存大小（MB） |

### 方式2：`go tool trace`

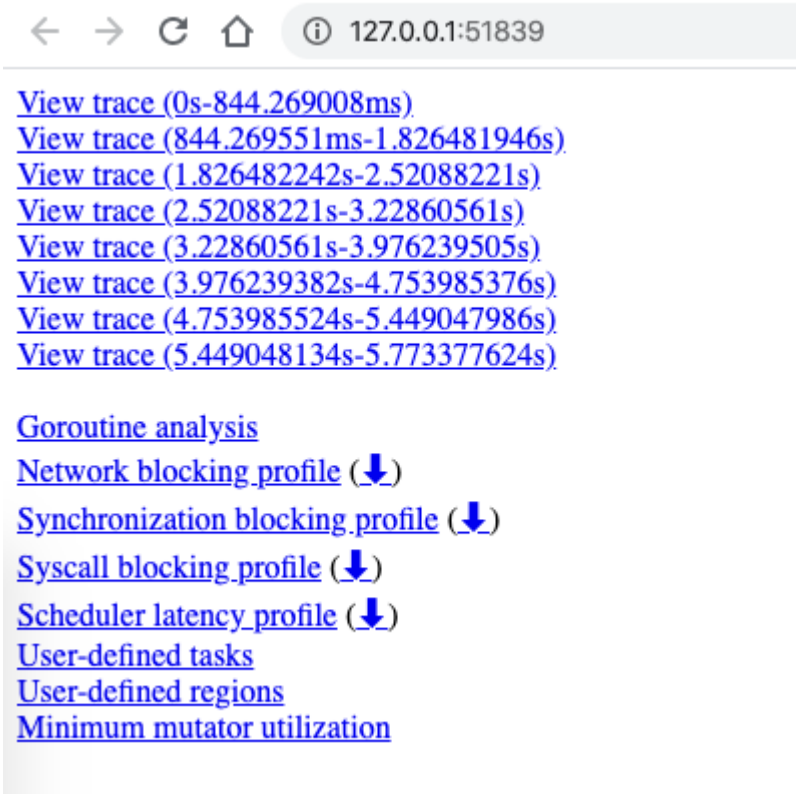
`go tool trace` 的主要功能是将统计而来的信息以一种可视化的方式展示给用户。要使用此工具，可以通过调用 trace API：

```
package main

func main() {
    f, _ := os.Create("trace.out")
    defer f.Close()
    trace.Start(f)
    defer trace.Stop()
    (...)
}
```

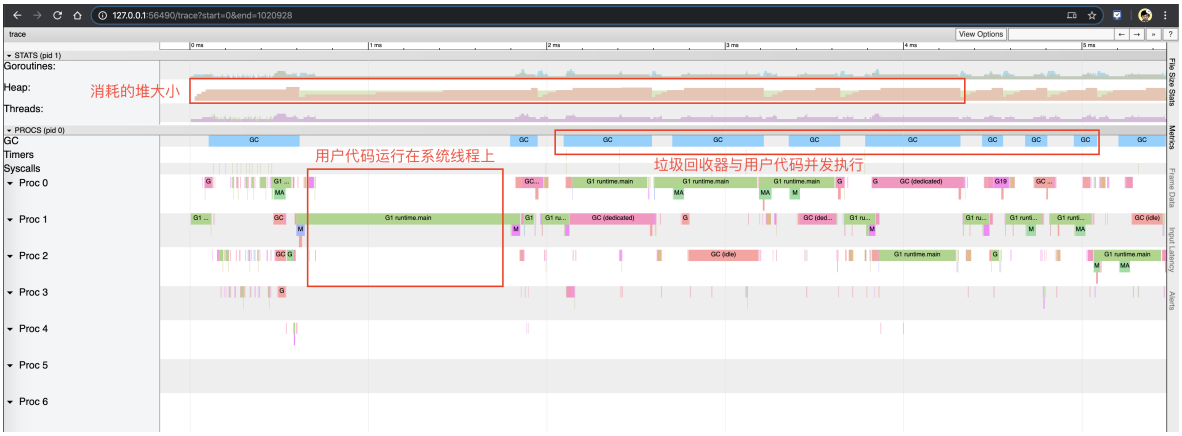
并通过

```
$ go tool trace trace.out
2019/12/30 15:50:33 Parsing trace...
2019/12/30 15:50:38 Splitting trace...
2019/12/30 15:50:45 Opening browser. Trace viewer is listening on http://127.0.0.1:51839
```

命令来启动可视化界面：




选择第一个链接可以获得如下图示：




右上角的问号可以打开帮助菜单，主要使用方式包括：

- w/s 键可以用于放大或者缩小视图
- a/d 键可以用于左右移动
- 按住 Shift 可以选取多个事件

### 方式3：`debug.ReadGCStats`

此方式可以通过代码的方式来直接实现对感兴趣指标的监控，例如我们希望每隔一秒钟监控一次 GC 的状态：

```
func printGCStats() {
    t := time.NewTicker(time.Second)
    s := debug.GCStats{}
    for {
```

```
        select {
        case <-t.C:
            debug.ReadGCStats(&s)
            fmt.Printf("gc %d last@%v, PauseTotal %v\n", s.NumGC, s.LastGC, s.PauseTotal)
        }
    }
}
func main() {
    go printGCStats()
    (...)
}
```

我们能够看到如下输出：

```
$ go run main.go

gc 4954 last@2019-12-30 15:19:37.505575 +0100 CET, PauseTotal 29.901171ms
gc 9195 last@2019-12-30 15:19:38.50565 +0100 CET, PauseTotal 77.579622ms
gc 13502 last@2019-12-30 15:19:39.505714 +0100 CET, PauseTotal 128.022307ms
gc 17555 last@2019-12-30 15:19:40.505579 +0100 CET, PauseTotal 182.816528ms
gc 21838 last@2019-12-30 15:19:41.505595 +0100 CET, PauseTotal 246.618502ms
```

## 方式4：`runtime.ReadMemStats`

除了使用 debug 包提供的方法外，还可以直接通过运行时的内存相关的 API 进行监控：

```
func printMemStats() {
    t := time.NewTicker(time.Second)
    s := runtime.MemStats{}

    for {
        select {
        case <-t.C:
            runtime.ReadMemStats(&s)
            fmt.Printf("gc %d last@%v, next_heap_size@%vMB\n", s.NumGC, time.Unix(int64(time.Duration(s.LastG
C).Seconds()), 0), s.NextGC/(1<<20))
        }
    }
}
func main() {
    go printMemStats()
    (...)
}
```

```
$ go run main.go

gc 4887 last@2019-12-30 15:44:56 +0100 CET, next_heap_size@4MB
gc 10049 last@2019-12-30 15:44:57 +0100 CET, next_heap_size@4MB
gc 15231 last@2019-12-30 15:44:58 +0100 CET, next_heap_size@4MB
gc 20378 last@2019-12-30 15:44:59 +0100 CET, next_heap_size@6MB
```

当然，后两种方式能够监控的指标很多，读者可以自行查看 `debug.GCStats` [2] 和 `runtime.MemStats` [3] 的字段，这里不再赘述。

# 有了 GC，为什么还会发生内存泄露？

## 7. 有了 GC，为什么还会发生内存泄露？

在一个具有 GC 的语言中，我们常说的内存泄漏，用严谨的话来说应该是：预期的能很快被释放的内存由于附着在了长期存活的内存上、或生命期意外地被延长，导致预计能够立即回收的内存而长时间得不到回收。

在 Go 中，由于 goroutine 的存在，所谓的内存泄漏除了附着在长期对象上之外，还存在多种不同的形式。

### 形式1：预期能被快速释放的内存因被根对象引用而没有得到迅速释放

当有一个全局对象时，可能不经意间将某个变量附着在其上，且忽略的将其进行释放，则该内存永远不会得到释放。例如：

```
var cache = map[interface{}]interface{}{}

func keepalloc() {
    for i := 0; i < 10000; i++ {
        m := make([]byte, 1<<10)
        cache[i] = m
    }
}
```

### 形式2：goroutine 泄漏

Goroutine 作为一种逻辑上理解的轻量级线程，需要维护执行用户代码的上下文信息。在运行过程中也需要消耗一定的内存来保存这类信息，而这些内存在目前版本的 Go 中是不会被释放的。因此，如果一个程序持续不断地产生新的 goroutine、且不结束已经创建的 goroutine 并复用这部分内存，就会造成内存泄漏的现象，例如：

```
func keepalloc2() {
    for i := 0; i < 100000; i++ {
        go func() {
            select {}
        }()
    }
}
```

### 验证

我们可以通过如下形式来调用上述两个函数：

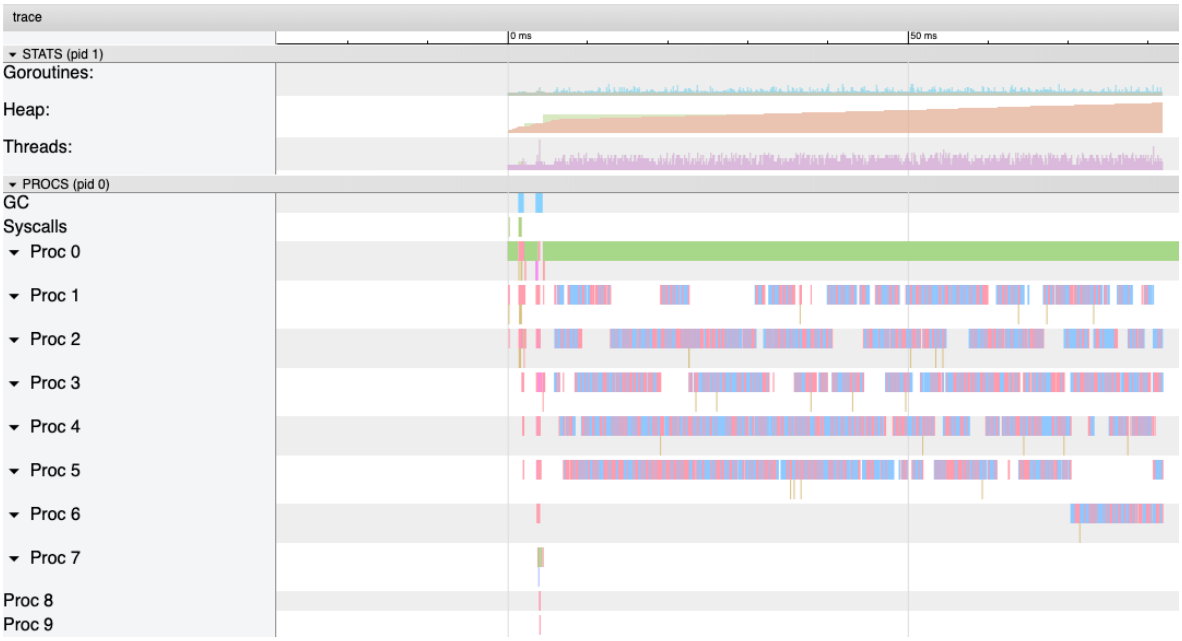
```
package main

import (
    "os"
    "runtime/trace"
)

func main() {
    f, _ := os.Create("trace.out")
    defer f.Close()
    trace.Start(f)
    defer trace.Stop()
    keepalloc()
```

```
    keepalloc2()
}
```

运行程序：

```
go run main.go
```

会看到程序中生成了 `trace.out` 文件，我们可以使用 `go tool trace trace.out` 命令得到下图：




可以看到，途中的 Heap 在持续增长，没有内存被回收，产生了内存泄漏的现象。

值得一提的是，这种形式的 goroutine 泄漏还可能由 channel 泄漏导致。而 channel 的泄漏本质上与 goroutine 泄漏存在直接联系。Channel 作为一种同步原语，会连接两个不同的 goroutine，如果一个 goroutine 尝试向一个没有接收方的无缓冲 channel 发送消息，则该 goroutine 会被永久的休眠，整个 goroutine 及其执行栈都得不到释放，例如：

```
var ch = make(chan struct{})

func keepalloc3() {
    for i := 0; i < 100000; i++ {
        // 没有接收方，goroutine 会一直阻塞
        go func() { ch <- struct{}{} }()
    }
}
```

# 并发标记清除法的难点是什么？

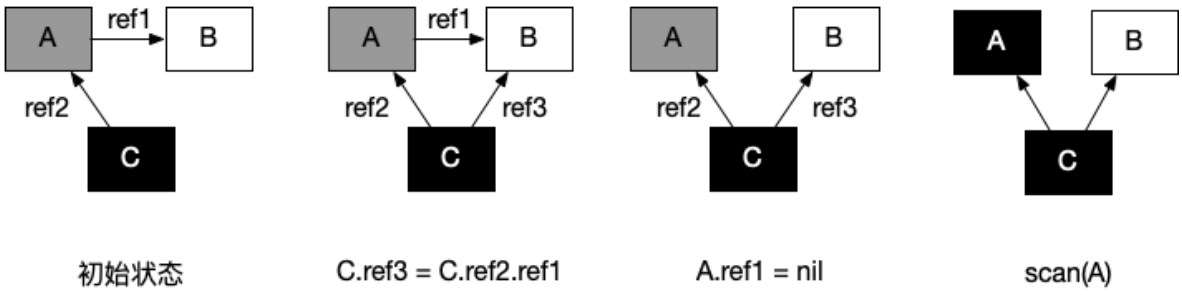
## 8. 并发标记清除法的难点是什么？

在没有用户态代码并发修改 `三色抽象` 的情况下，回收可以正常结束。但是并发回收的根本问题在于，用户态代码在回收过程中会并发地更新对象图，从而造成赋值器和回收器可能对对象图的结构产生不同的认知。这时以一个固定的三色波面作为回收过程前进的边界则不再合理。

我们不妨考虑赋值器写操作的例子：

| 时序 | 回收器 | 赋值器 | 说明 |
| --- | --- | --- | --- |
| 1 | shade(A, gray) | | 回收器：根对象的子节点着色为灰色对象 |
| 2 | shade(C, black) | | 回收器：当所有子节点着色为灰色后，<br>将节点着为黑色 |
| 3 | | C.ref3 = C.ref2.ref1 | 赋值器：并发的修改了 C 的子节点 |
| 4 | | A.ref1 = nil | 赋值器：并发的修改了 A 的子节点 |
| 5 | shade(A.ref1, gray) | | 回收器：<br>进一步灰色对象的子节点并着色为灰色对象，<br>这时由于 `A.ref1` 为 `nil`，<br>什么事情也没有发生 |
| 6 | shade(A, black) | | 回收器：由于所有子节点均已标记，<br>回收器也不会重新扫描已经被标记为黑色的对象，此时 A 被着色为黑色，`scan(A)`<br>什么也不会发生，进而 B<br>在此次回收过程中永远不会被标记为黑色，<br>进而错误地被回收。 |

- 初始状态：假设某个黑色对象 C 指向某个灰色对象 A ，而 A 指向白色对象 B；
- `C.ref3 = C.ref2.ref1`：赋值器并发地将黑色对象 C 指向（ref3）了白色对象 B；
- `A.ref1 = nil`：移除灰色对象 A 对白色对象 B 的引用（ref2）；
- 最终状态：在继续扫描的过程中，白色对象 B 永远不会被标记为黑色对象了（回收器不会重新扫描黑色对象），进而对象 B 被错误地回收。




总而言之，并发标记清除中面临的一个根本问题就是如何保证标记与清除过程的正确性。

# 什么是写屏障、混合写屏障，如何实现？

## 9. 什么是写屏障、混合写屏障，如何实现？

要讲清楚写屏障，就需要理解三色标记清除算法中的**强弱不变性**以及**赋值器的颜色**，理解他们需要一定的抽象思维。写屏障是一个在并发垃圾回收器中才会出现的概念，垃圾回收器的正确性体现在：**不应出现对象的丢失，也不应错误的回收还不需要回收的对象。**

可以证明，当以下两个条件同时满足时会破坏垃圾回收器的正确性：

- **条件 1**: 赋值器修改对象图，导致某一黑色对象引用白色对象；
- **条件 2**: 从灰色对象出发，到达白色对象的、未经访问过的路径被赋值器破坏。

只要能够避免其中任何一个条件，则不会出现对象丢失的情况，因为：

- 如果条件 1 被避免，则所有白色对象均被灰色对象引用，没有白色对象会被遗漏；
- 如果条件 2 被避免，即便白色对象的指针被写入到黑色对象中，但从灰色对象出发，总存在一条没有访问过的路径，从而找到到达白色对象的路径，白色对象最终不会被遗漏。

我们不妨将三色不变性所定义的波面根据这两个条件进行削弱：

- 当满足原有的三色不变性定义（或上面的两个条件都不满足时）的情况称为**强三色不变性（strong tricolor invariant）**
- 当赋值器令黑色对象引用白色对象时（满足条件 1 时）的情况称为**弱三色不变性（weak tricolor invariant）**

当赋值器进一步破坏灰色对象到达白色对象的路径时（进一步满足条件 2 时），即打破弱三色不变性，
也就破坏了回收器的正确性；或者说，在破坏强弱三色不变性时必须引入额外的辅助操作。
弱三色不变形的好处在于：**只要存在未访问的能够到达白色对象的路径，就可以将黑色对象指向白色对象。**

如果我们考虑并发的用户态代码，回收器不允许同时停止所有赋值器，就是涉及了存在的多个不同状态的赋值器。为了对概念加以明确，还需要换一个角度，把回收器视为对象，把赋值器视为影响回收器这一对象的实际行为（即影响 GC 周期的长短），从而引入赋值器的颜色：

- 黑色赋值器：已经由回收器扫描过，不会再次对其进行扫描。
- 灰色赋值器：尚未被回收器扫描过，或尽管已经扫描过但仍需要重新扫描。

赋值器的颜色对回收周期的结束产生影响：

- 如果某种并发回收器允许灰色赋值器的存在，则必须在回收结束之前重新扫描对象图。
- 如果重新扫描过程中发现了新的灰色或白色对象，回收器还需要对新发现的对象进行追踪，但是在新追踪的过程中，赋值器仍然可能在其根中插入新的非黑色的引用，如此往复，直到重新扫描过程中没有发现新的白色或灰色对象。

于是，在允许灰色赋值器存在的算法，最坏的情况下，回收器只能将所有赋值器线程停止才能完成其跟对象的完整扫描，也就是我们所说的 STW。

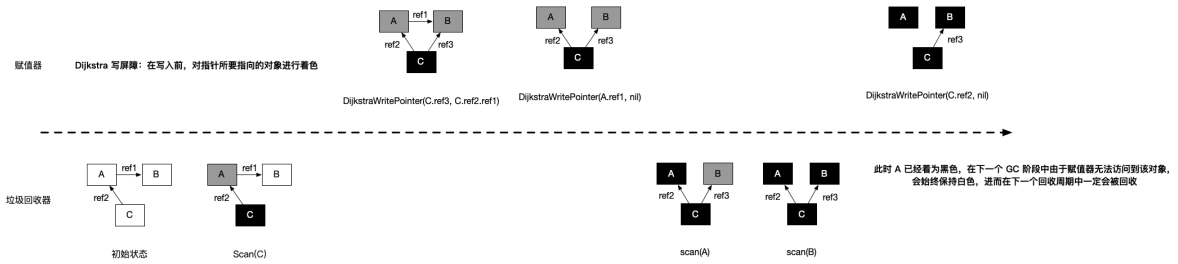
为了确保强弱三色不变性的并发指针更新操作，需要通过赋值器屏障技术来保证指针的读写操作一致。因此我们所说的 Go 中的写屏障、混合写屏障，其实是指赋值器的写屏障，赋值器的写屏障作为一种同步机制，使赋值器在进行指针写操作时，能够“通知”回收器，进而不会破坏弱三色不变性。

有两种非常经典的写屏障：Dijkstra 插入屏障和 Yuasa 删除屏障。

灰色赋值器的 Dijkstra 插入屏障的基本思想是避免满足条件 1：

```
// 灰色赋值器 Dijkstra 插入屏障
func DijkstraWritePointer(slot *unsafe.Pointer, ptr unsafe.Pointer) {
    shade(ptr)
    *slot = ptr
}
```

为了防止黑色对象指向白色对象，应该假设 `*slot` 可能会变为黑色，为了确保 `ptr` 不会在被赋值到 `*slot` 前变为白色， `shade(ptr)` 会先将指针 `ptr` 标记为灰色，进而避免了条件 1。如图所示：




Dijkstra 插入屏障的好处在于可以立刻开始并发标记。但存在两个缺点：

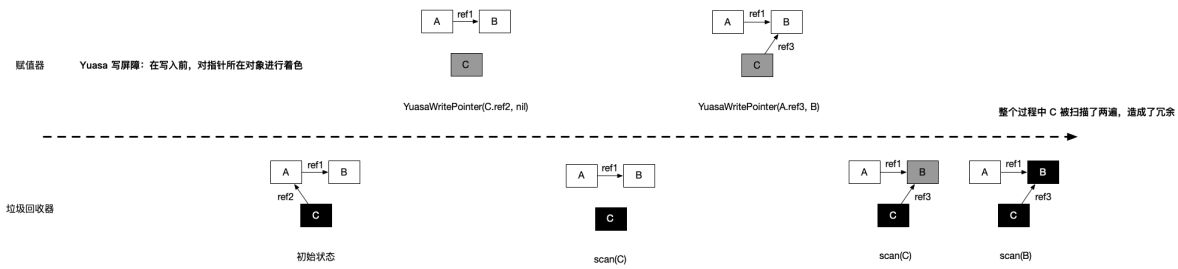
1. 由于 Dijkstra 插入屏障的“保守”，在一次回收过程中可能会残留一部分对象没有回收成功，只有在下一个回收过程中才会被回收；
2. 在标记阶段中，每次进行指针赋值操作时，都需要引入写屏障，这无疑会增加大量性能开销；为了避免造成性能问题，Go 团队在最终实现时，没有为所有栈上的指针写操作，启用写屏障，而是当发生栈上的写操作时，将栈标记为灰色，但此举产生了灰色赋值器，将会需要标记终止阶段 STW 时对这些栈进行重新扫描。

另一种比较经典的写屏障是黑色赋值器的 Yuasa 删除屏障。其基本思想是避免满足条件 2：

```
// 黑色赋值器 Yuasa 屏障
func YuasaWritePointer(slot *unsafe.Pointer, ptr unsafe.Pointer) {
    shade(*slot)
    *slot = ptr
}
```

为了防止丢失从灰色对象到白色对象的路径，应该假设 `*slot` 可能会变为黑色，为了确保 `ptr` 不会在被赋值到 `*slot` 前变为白色， `shade(*slot)` 会先将 `*slot` 标记为灰色，进而该写操作总是创造了一条灰色到灰色或者灰色到白色对象的路径，进而避免了条件 2。

Yuasa 删除屏障的优势则在于不需要标记结束阶段的重新扫描，结束时候能够准确的回收所有需要回收的白色对象。缺陷是 Yuasa 删除屏障会拦截写操作，进而导致波面的退后，产生“冗余”的扫描：




Go 在 1.8 的时候为了简化 GC 的流程，同时减少标记终止阶段的重扫成本，将 Dijkstra 插入屏障和 Yuasa 删除屏障进行混合，形成混合写屏障。该屏障提出时的基本思想是：**对正在被覆盖的对象进行着色，且如果当前栈未扫描完成，则同样对指针进行着色。**

但在最终实现时原提案[4]中对 `ptr` 的着色还额外包含对执行栈的着色检查，但由于时间有限，并未完整实现过，所以混合写屏障在目前的实现伪代码是：

```
// 混合写屏障
func HybridWritePointerSimple(slot *unsafe.Pointer, ptr unsafe.Pointer) {
    shade(*slot)
    shade(ptr)
    *slot = ptr
}
```

在这个实现中，如果无条件对引用双方进行着色，自然结合了 Dijkstra 和 Yuasa 写屏障的优势，但缺点也非常明显，因为着色成本是双倍的，而且编译器需要插入的代码也成倍增加，随之带来的结果就是编译后的二进制文件大小也进一步增加。为了针对写屏障的性能进行优化，Go 1.10 前后，Go 团队随后实现了批量写屏障机制。其基本想法是将需要着色的指针统一写入一个缓存，每当缓存满时统一对缓存中的所有 `ptr` 指针进行着色。

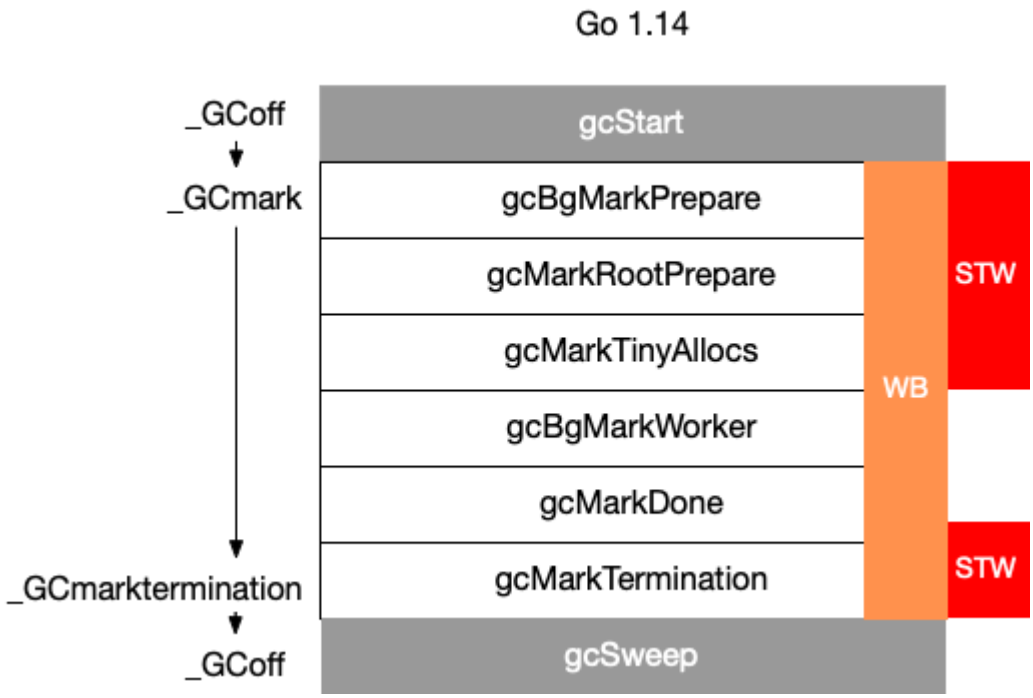
# Go 语言中 GC 的流程是什么？

## 10. Go 语言中 GC 的流程是什么？

当前版本的 Go 以 STW 为界限，可以将 GC 划分为五个阶段：

| 阶段 | 说明 | 赋值器状态 |
| --- | --- | --- |
| SweepTermination | 清扫终止阶段，<br>为下一个阶段的并发标记做准备工作，<br>启动写屏障 | STW |
| Mark | 扫描标记阶段，与赋值器并发执行，写屏障开启 | 并发 |
| MarkTermination | 标记终止阶段，保证一个周期内标记任务完成，<br>停止写屏障 | STW |
| GCoff | 内存清扫阶段，将需要回收的内存归还到堆中，<br>写屏障关闭 | 并发 |
| GCoff | 内存归还阶段，将过多的内存归还给操作系统，<br>写屏障关闭 | 并发 |

具体而言，各个阶段的触发函数分别为：



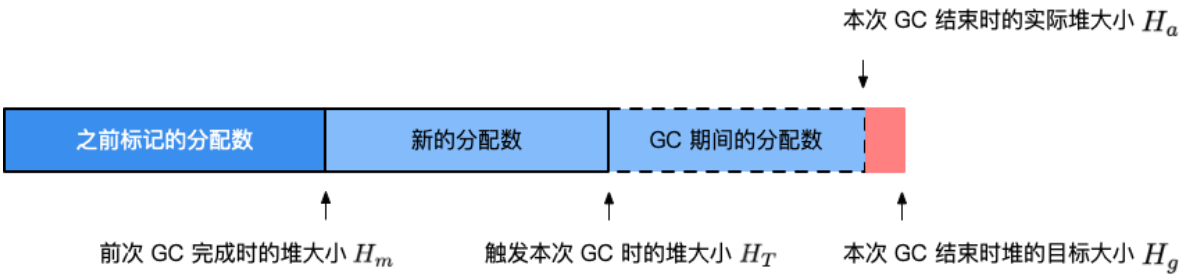
# 触发 GC 的时机是什么？

## 11. 触发 GC 的时机是什么？

Go 语言中对 GC 的触发时机存在两种形式：

1. **主动触发**，通过调用 runtime.GC 来触发 GC，此调用阻塞式地等待当前 GC 运行完毕。
2. **被动触发**，分为两种方式：
   - 使用系统监控，当超过两分钟没有产生任何 GC 时，强制触发 GC。
   - 使用步调（Pacing）算法，其核心思想是控制内存增长的比例。

通过 `GOGC` 或者 `debug.SetGCPercent` 进行控制（他们控制的是同一个变量，即堆的增长率 $\rho$）。整个算法的设计考虑的是优化问题：如果设上一次 GC 完成时，内存的数量为 $H_m$（heap marked），估计需要触发 GC 时的堆大小 $H_T$（heap trigger），使得完成 GC 时候的目标堆大小 $H_g$（heap goal） 与实际完成时候的堆大小 $H_a$（heap actual）最为接近，即： $\min |H_g - H_a| = \min|(1+\rho)H_m - H_a|$。




除此之外，步调算法还需要考虑 CPU 利用率的问题，显然我们不应该让垃圾回收器占用过多的 CPU，即不应该让每个负责执行用户 goroutine 的线程都在执行标记过程。理想情况下，在用户代码满载的时候，GC 的 CPU 使用率不应该超过 25%，即另一个优化问题：如果设 $u_g$为目标 CPU 使用率（goal utilization），而 $u_a$为实际 CPU 使用率（actual utilization），则 $\min|u_g - u_a|$。

> 求解这两个优化问题的具体数学建模过程我们不在此做深入讨论，有兴趣的读者可以参考两个设计文档：Go 1.5 concurrent garbage collector pacing[5] 和 Separate soft and hard heap size goal[6]。

计算 $H_T$ 的最终结论（从 Go 1.10 时开始 $h_t$ 增加了上界 $0.95 \rho$，从 Go 1.14 开始时 $h_t$ 增加了下界 0.6）是：

- 设第 n 次触发 GC 时 (n > 1)，估计得到的堆增长率为 $h_t^{(n)}$、运行过程中的实际堆增长率为 $h_a^{(n)}$，用户设置的增长率为 $\rho = \text{GOGC}/100$（ $\rho > 0$）则第 $n+1$ 次出触发 GC 时候，估计的堆增长率为：

$$h_t^{(n+1)} = h_t^{(n)} + 0.5\left[\frac{H_g^{(n)} - H_a^{(n)}}{H_a^{(n)}} - h_t^{(n)} - \frac{u_a^{(n)}}{u_g^{(n)}}\left(h_a^{(n)} - h_t^{(n)}\right)\right]$$

ht(n+1)=ht(n)
+0.5[Ha(n)Hg(n)−Ha(n)−ht(n)−ug(n)ua(n)(ha(n)−ht(n))]

- 特别的，$h_t^{(1)} = 7 / 8$，$u_a^{(1)} = 0.25$，$u_g^{(1)} = 0.3$。第一次触发 GC 时，如果当前的堆小于 $4\rho$ MB，则强制调整到 $4\rho$ MB 时触发 GC

- 特别的，当 $h_t^{(n)}<0.6$时，将其调整为 $0.6$，当 $h_t^{(n)} > 0.95 \rho$ 时，将其设置为 $0.95 \rho$
- 默认情况下，$\rho = 1$（即 GOGC = 100），第一次触发 GC 时强制设置触发第一次 GC 为 4MB，可以写如下程序进行验证：

```
package main

import (
    "os"
    "runtime"
    "runtime/trace"
    "sync/atomic"
)

var stop uint64

// 通过对象 P 的释放状态，来确定 GC 是否已经完成
func gcfinished() *int {
    p := 1
    runtime.SetFinalizer(&p, func(_ *int) {
        println("gc finished")
        atomic.StoreUint64(&stop, 1) // 通知停止分配
    })
    return &p
}

func allocate() {
    // 每次调用分配 0.25MB
    _ = make([]byte, int((1<<20)*0.25))
}

func main() {
    f, _ := os.Create("trace.out")
    defer f.Close()
    trace.Start(f)
    defer trace.Stop()

    gcfinished()

    // 当完成 GC 时停止分配
    for n := 1; atomic.LoadUint64(&stop) != 1; n++ {
        println("#allocate: ", n)
        allocate()
    }
    println("terminate")
}
```

我们先来验证最简单的一种情况，即第一次触发 GC 时的堆大小：

```
$ go build -o main
$ GODEBUG=gctrace=1 ./main
#allocate:  1
(...)
#allocate:  20
gc finished
gc 1 @0.001s 3%: 0.016+0.23+0.019 ms clock, 0.20+0.11/0.060/0.13+0.22 ms cpu, 4->5->1 MB, 5 MB goal, 12 P
scvg: 8 KB released
```

```
scvg: inuse: 1, idle: 62, sys: 63, released: 58, consumed: 5 (MB)
terminate
```

通过这一行数据我们可以看到：

```
gc 1 @0.001s 3%: 0.016+0.23+0.019 ms clock, 0.20+0.11/0.060/0.13+0.22 ms cpu, 4->5->1 MB, 5 MB goal, 12 P
```

1. 程序在完成第一次 GC 后便终止了程序，符合我们的设想
2. 第一次 GC 开始时的堆大小为 4MB，符合我们的设想
3. 当标记终止时，堆大小为 5MB，此后开始执行清扫，这时分配执行到第 20 次，即 20*0.25 = 5MB，符合我们的设想

我们将分配次数调整到 50 次

```
for n := 1; n < 50; n++ {
    println("#allocate: ", n)
    allocate()
}
```

来验证第二次 GC 触发时是否满足公式所计算得到的值（为 GODEBUG 进一步设置 `gcpacertrace=1` ）：

```
$ go build -o main
$ GODEBUG=gctrace=1,gcpacertrace=1 ./main
#allocate:  1
(...)

pacer: H_m_prev=2236962 h_t=+8.750000e-001 H_T=4194304 h_a=+2.387451e+000 H_a=7577600 h_g=+1.442627e+000 H_g=5464064 u_a=+2.652227e-001 u_g=+3.000000e-001 W_a=152832 goal Δ=+5.676271e-001 actual Δ=+1.512451e+000 u_a/u_g=+8.840755e-001
#allocate:  28
gc 1 @0.001s 5%: 0.032+0.32+0.055 ms clock, 0.38+0.068/0.053/0.11+0.67 ms cpu, 4->7->3 MB, 5 MB goal, 12 P

(...)
#allocate:  37
pacer: H_m_prev=3307736 h_t=+6.000000e-001 H_T=5292377 h_a=+7.949171e-001 H_a=5937112 h_g=+1.000000e+000 H_g=6615472 u_a=+2.658428e-001 u_g=+3.000000e-001 W_a=154240 goal Δ=+4.000000e-001 actual Δ=+1.949171e-001 u_a/u_g=+8.861428e-001
#allocate:  38
gc 2 @0.002s 9%: 0.017+0.26+0.16 ms clock, 0.20+0.079/0.058/0.12+1.9 ms cpu, 5->5->0 MB, 6 MB goal, 12 P
```

我们可以得到数据：

- 第一次估计得到的堆增长率为 $h_t^{(1)} = 0.875$
- 第一次的运行过程中的实际堆增长率为 $h_a^{(1)} = 0.2387451$
- 第一次实际的堆大小为 $H_a^{(1)}=7577600$
- 第一次目标的堆大小为 $H_g^{(1)}=5464064$
- 第一次的 CPU 实际使用率为 $u_a^{(1)} = 0.2652227$
- 第一次的 CPU 目标使用率为 $u_g^{(1)} = 0.3$

我们据此计算第二次估计的堆增长率：

因为 $0.52534543909 < 0.6\rho = 0.6$，因此下一次的触发率为 $h_t^{2} = 0.6$，与我们实际观察到的第二次 GC 的触发率 0.6 吻合。

# 如果内存分配速度超过了标记清除的速度怎么办？

## 12. 如果内存分配速度超过了标记清除的速度怎么办？

目前的 Go 实现中，当 GC 触发后，会首先进入并发标记的阶段。并发标记会设置一个标志，并在 mallocgc 调用时进行检查。当存在新的内存分配时，会暂停分配内存过快的那些 goroutine，并将其转去执行一些辅助标记（Mark Assist）的工作，从而达到放缓继续分配、辅助 GC 的标记工作的目的。

编译器会分析用户代码，并在需要分配内存的位置，将申请内存的操作翻译为 `mallocgc` 调用，而 `mallocgc` 的实现决定了标记辅助的实现，其伪代码思路如下：

```
func mallocgc(t typ.Type, size uint64) {
    if enableMarkAssist {
        // 进行标记辅助，此时用户代码没有得到执行
        (...)
    }
    // 执行内存分配
    (...)
}
```

# GC 关注的指标有哪些？

## 13. GC 关注的指标有哪些？

Go 的 GC 被设计为成比例触发、大部分工作与赋值器并发、不分代、无内存移动且会主动向操作系统归还申请的内存。因此最主要关注的、能够影响赋值器的性能指标有：

- CPU 利用率：回收算法会在多大程度上拖慢程序？有时候，这个是通过回收占用的 CPU 时间与其它 CPU 时间的百分比来描述的。
- GC 停顿时间：回收器会造成多长时间的停顿？目前的 GC 中需要考虑 STW 和 Mark Assist 两个部分可能造成的停顿。
- GC 停顿频率：回收器造成的停顿频率是怎样的？目前的 GC 中需要考虑 STW 和 Mark Assist 两个部分可能造成的停顿。
- GC 可扩展性：当堆内存变大时，垃圾回收器的性能如何？但大部分的程序可能并不一定关心这个问题。

# Go 的 GC 如何调优？

## 14. Go 的 GC 如何调优？

Go 的 GC 被设计为极致简洁，与较为成熟的 Java GC 的数十个可控参数相比，严格意义上来讲，Go 可供用户调整的参数只有 GOGC 环境变量。当我们谈论 GC 调优时，通常是指减少用户代码对 GC 产生的压力，这一方面包含了减少用户代码分配内存的数量（即对程序的代码行为进行调优），另一方面包含了最小化 Go 的 GC 对 CPU 的使用率（即调整 GOGC）。

GC 的调优是在特定场景下产生的，并非所有程序都需要针对 GC 进行调优。只有那些对执行延迟非常敏感、
当 GC 的开销成为程序性能瓶颈的程序，才需要针对 GC 进行性能调优，几乎不存在于实际开发中 99% 的情况。
除此之外，Go 的 GC 也仍然有一定的可改进的空间，也有部分 GC 造成的问题，目前仍属于 Open Problem。

总的来说，我们可以在现在的开发中处理的有以下几种情况：

1. 对停顿敏感：GC 过程中产生的长时间停顿、或由于需要执行 GC 而没有执行用户代码，导致需要立即执行的用户代码执行滞后。
2. 对资源消耗敏感：对于频繁分配内存的应用而言，频繁分配内存增加 GC 的工作量，原本可以充分利用 CPU 的应用不得不频繁地执行垃圾回收，影响用户代码对 CPU 的利用率，进而影响用户代码的执行效率。

从这两点来看，所谓 GC 调优的核心思想也就是充分的围绕上面的两点来展开：优化内存的申请速度，尽可能的少申请内存，复用已申请的内存。或者简单来说，不外乎这三个关键字：**控制、减少、复用**。

我们将通过三个实际例子介绍如何定位 GC 的存在的问题，并一步一步进行性能调优。当然，在实际情况中问题远比这些例子要复杂，这里也只是讨论调优的核心思想，更多的时候也只能具体问题具体分析。

### 例1：合理化内存分配的速度、提高赋值器的 CPU 利用率

我们来看这样一个例子。在这个例子中，`concat` 函数负责拼接一些长度不确定的字符串。并且为了快速完成任务，出于某种原因，在两个嵌套的 for 循环中一口气创建了 800 个 goroutine。在 main 函数中，启动了一个 goroutine 并在程序结束前不断的触发 GC，并尝试输出 GC 的平均执行时间：

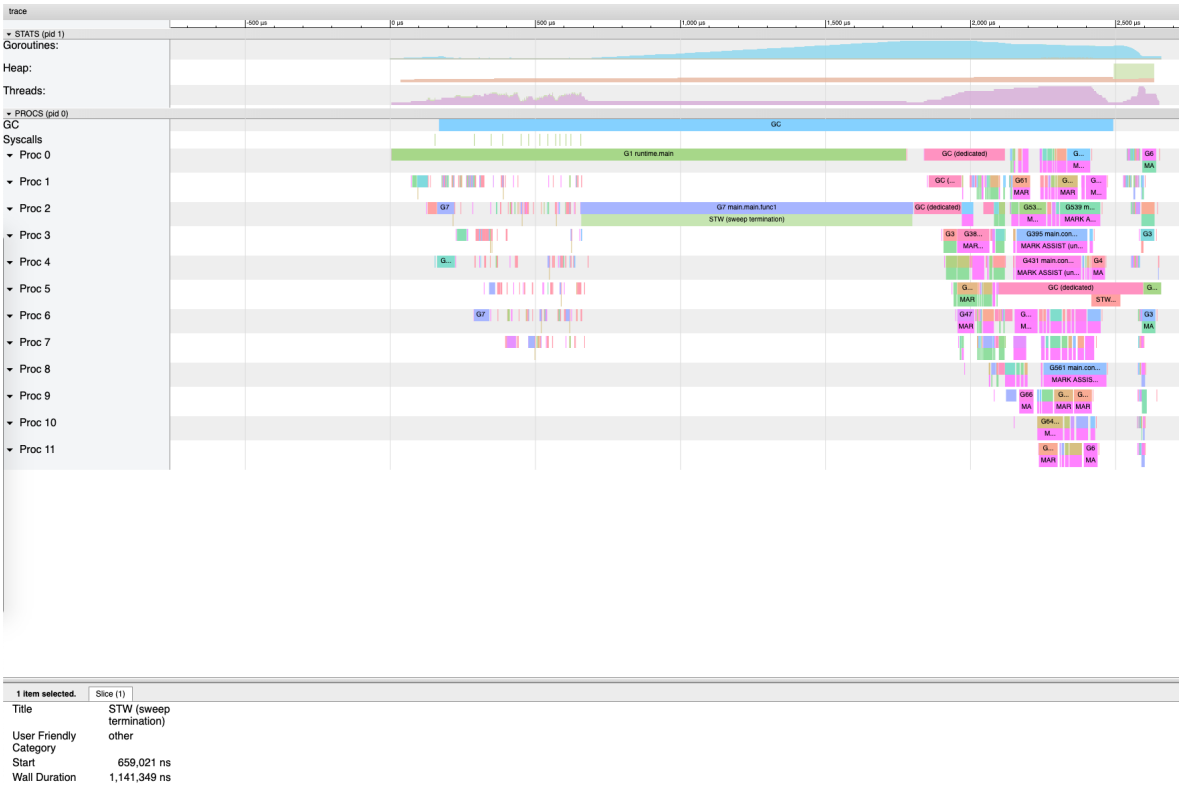
```
package main

import (
    "fmt"
    "os"
    "runtime"
    "runtime/trace"
    "sync/atomic"
    "time"
)

var (
    stop  int32
    count int64
    sum   time.Duration
)

func concat() {
    for n := 0; n < 100; n++ {
        for i := 0; i < 8; i++ {
            go func() {
                s := "Go GC"
                s += " " + "Hello"
                s += " " + "World"
                _ = s
```

```
            }()
        }
    }
}

func main() {
    f, _ := os.Create("trace.out")
    defer f.Close()
    trace.Start(f)
    defer trace.Stop()

    go func() {
        var t time.Time
        for atomic.LoadInt32(&stop) == 0 {
            t = time.Now()
            runtime.GC()
            sum += time.Since(t)
            count++
        }
        fmt.Printf("GC spend avg: %v\n", time.Duration(int64(sum)/count))
    }()

    concat()
    atomic.StoreInt32(&stop, 1)
}
```

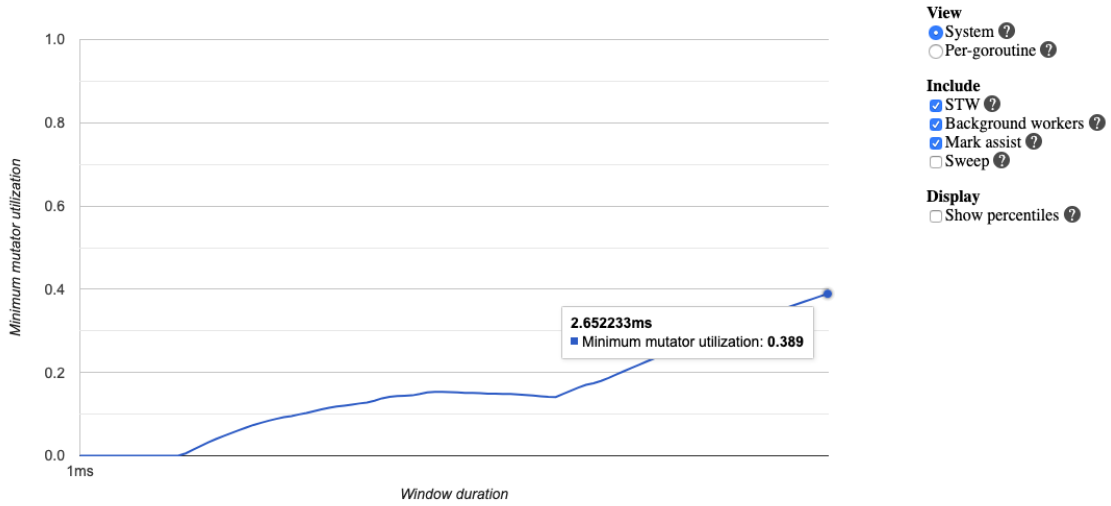
这个程序的执行结果是：

```
$ go build -o main
$ ./main
GC spend avg: 2.583421ms
```

GC 平均执行一次需要长达 2ms 的时间，我们再进一步观察 trace 的结果：




程序的整个执行过程中仅执行了一次 GC，而且仅 Sweep STW 就耗费了超过 1 ms，非常反常。甚至查看赋值器 mutator 的 CPU 利用率，在整个 trace 尺度下连 40% 都不到：




主要原因是什么呢？我们不妨查看 goroutine 的分析：

Goroutine Name: main.concat.func1
Number of Goroutines: 764
Execution Time: 48.94% of total program execution time
Network Wait Time: graph(download)
Sync Block Time: graph(download)
Blocking Syscall Time: graph(download)
Scheduler Wait Time: graph(download)

| Goroutine | Total | Execution | Network wait | Sync block | Blocking syscall | Scheduler wait | GC sweeping | GC pause |
|---|---|---|---|---|---|---|---|---|
| 394 | 1966µs | 8149ns | 0ns | 0ns | 0ns | 1632µs | 0ns (0.0%) | 1801µs (91.6%) |
| 395 | 1965µs | 290µs | 0ns | 0ns | 0ns | 1674µs | 0ns (0.0%) | 1800µs (91.6%) |
| 396 | 1964µs | 11µs | 0ns | 235µs | 0ns | 1717µs | 0ns (0.0%) | 1799µs (91.6%) |
| 397 | 1963µs | 47µs | 0ns | 0ns | 0ns | 1916µs | 7753ns (0.4%) | 1798µs (91.6%) |
| 426 | 1903µs | 4963ns | 0ns | 0ns | 0ns | 1888µs | 0ns (0.0%) | 1738µs (91.3%) |
| 390 | 1902µs | 9359ns | 0ns | 0ns | 0ns | 1893µs | 0ns (0.0%) | 1804µs (94.9%) |
| 432 | 1897µs | 6247ns | 0ns | 265µs | 0ns | 1625µs | 0ns (0.0%) | 1732µs (91.3%) |
| 433 | 1894µs | 3284ns | 0ns | 0ns | 0ns | 1809µs | 0ns (0.0%) | 1729µs (91.3%) |
| 434 | 1893µs | 7234ns | 0ns | 248µs | 0ns | 1637µs | 0ns (0.0%) | 1728µs (91.3%) |
| 435 | 1892µs | 3062ns | 0ns | 0ns | 0ns | 1822µs | 0ns (0.0%) | 1727µs (91.3%) |
| 430 | 1890µs | 5827ns | 0ns | 0ns | 0ns | 1554µs | 0ns (0.0%) | 1734µs (91.8%) |
| 389 | 1879µs | 112µs | 0ns | 0ns | 0ns | 1767µs | 0ns (0.0%) | 1805µs (96.1%) |
| 436 | 1878µs | 47µs | 0ns | 193µs | 0ns | 1637µs | 0ns (0.0%) | 1713µs (91.2%) |
| 431 | 1842µs | 237µs | 0ns | 0ns | 0ns | 1605µs | 0ns (0.0%) | 1733µs (94.1%) |
| 459 | 1833µs | 41µs | 0ns | 231µs | 0ns | 1557µs | 0ns (0.0%) | 1668µs (91.0%) |
| 425 | 1811µs | 41µs | 0ns | 0ns | 0ns | 1770µs | 0ns (0.0%) | 1738µs (96.0%) |
| 437 | 1776µs | 2766ns | 0ns | 0ns | 0ns | 1773µs | 0ns (0.0%) | 1712µs (96.4%) |
| 438 | 1776µs | 296ns | 0ns | 0ns | 0ns | 1776µs | 0ns (0.0%) | 1711µs (96.3%) |
| 439 | 1776µs | 296ns | 0ns | 0ns | 0ns | 1775µs | 0ns (0.0%) | 1710µs (96.3%) |
| 440 | 1775µs | 247ns | 0ns | 0ns | 0ns | 1775µs | 0ns (0.0%) | 1709µs (96.3%) |
| 441 | 1775µs | 272ns | 0ns | 0ns | 0ns | 1775µs | 0ns (0.0%) | 1708µs (96.2%) |
| 442 | 1775µs | 271ns | 0ns | 0ns | 0ns | 1774µs | 0ns (0.0%) | 1707µs (96.2%) |
| 443 | 1774µs | 271ns | 0ns | 0ns | 0ns | 1774µs | 0ns (0.0%) | 1706µs (96.2%) |
| 444 | 1774µs | 247ns | 0ns | 0ns | 0ns | 1773µs | 0ns (0.0%) | 1705µs (96.2%) |
| 445 | 1768µs | 296ns | 0ns | 0ns | 0ns | 1768µs | 0ns (0.0%) | 1700µs (96.1%) |
| 446 | 1768µs | 296ns | 0ns | 0ns | 0ns | 1768µs | 0ns (0.0%) | 1698µs (96.1%) |

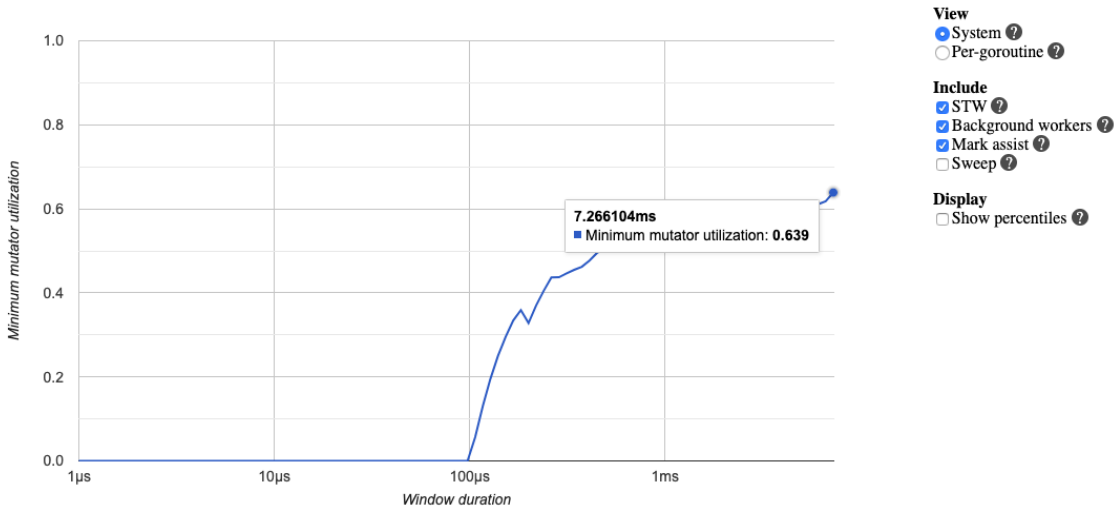
在这个榜单中我们不难发现，goroutine 的执行时间占其生命周期总时间非常短的一部分，但大部分时间都花费在调度器的等待上了（蓝色的部分），说明同时创建大量 goroutine 对调度器产生的压力确实不小，我们不妨将这一产生速率减慢，一批一批地创建 goroutine：

```
func concat() {
    wg := sync.WaitGroup{}
    for n := 0; n < 100; n++ {
        wg.Add(8)
        for i := 0; i < 8; i++ {
            go func() {
                s := "Go GC"
                s += " " + "Hello"
                s += " " + "World"
                _ = s
                wg.Done()
            }()
        }
        wg.Wait()
    }
}
```

这时候我们再来看：

```
$ go build -o main
$ ./main
GC spend avg: 328.54µs
```

GC 的平均时间就降到 300 微秒了。这时的赋值器 CPU 使用率也提高到了 60%，相对来说就很可观了：




当然，这个程序仍然有优化空间，例如我们其实没有必要等待很多 goroutine 同时执行完毕才去执行下一组 goroutine。而可以当一个 goroutine 执行完毕时，直接启动一个新的 goroutine，也就是 goroutine 池的使用。
有兴趣的读者可以沿着这个思路进一步优化这个程序中赋值器对 CPU 的使用率。

## 例2：降低并复用已经申请的内存

我们通过一个非常简单的 Web 程序来说明复用内存的重要性。在这个程序中，每当产生一个 `/example2` 的请求时，都会创建一段内存，并用于进行一些后续的工作。

```
package main

import (
    "fmt"
    "net/http"
    _ "net/http/pprof"
)

func newBuf() []byte {
    return make([]byte, 10<<20)
}

func main() {
    go func() {
        http.ListenAndServe("localhost:6060", nil)
    }()

    http.HandleFunc("/example2", func(w http.ResponseWriter, r *http.Request) {
        b := newBuf()

        // 模拟执行一些工作
        for idx := range b {
            b[idx] = 1
        }

        fmt.Fprintf(w, "done, %v", r.URL.Path[1:])
    })
    http.ListenAndServe(":8080", nil)
}
```

为了进行性能分析，我们还额外创建了一个监听 6060 端口的 goroutine，用于使用 pprof 进行分析。
我们先让服务器跑起来：

```
$ go build -o main
$ ./main
```

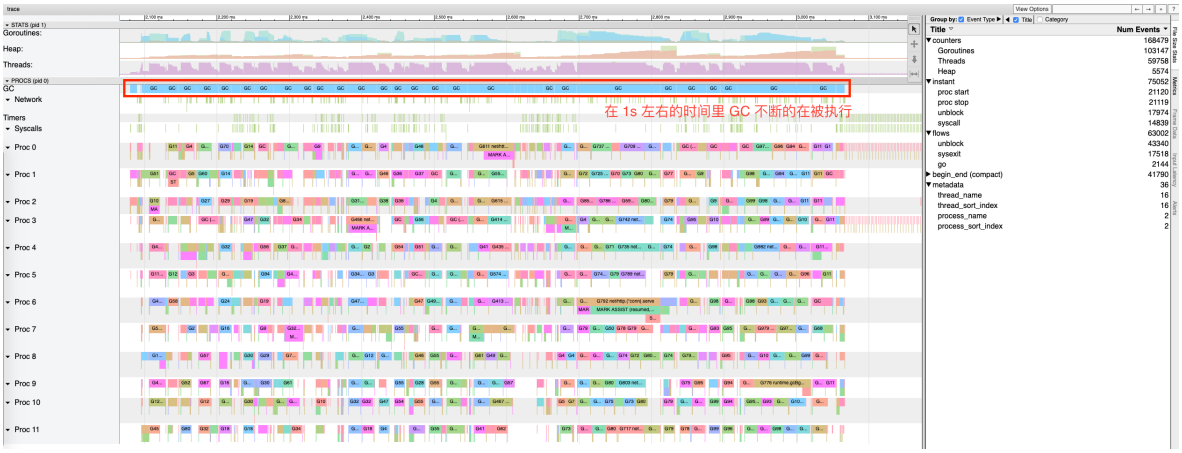
我们这次使用 pprof 的 trace 来查看 GC 在此服务器中面对大量请求时候的状态，要使用 trace 可以通过访问 `/debug/pprof/trace` 路由来进行，其中 `seconds` 参数设置为 20s，并将 trace 的结果保存为 `trace.out`：

```
$ wget http://127.0.0.1:6060/debug/pprof/trace\?seconds\=20 -O trace.out
--2020-01-01 22:13:34--  http://127.0.0.1:6060/debug/pprof/trace?seconds=20
Connecting to 127.0.0.1:6060... connected.
HTTP request sent, awaiting response...
```

这时候我们使用一个压测工具 `ab`，来同时产生 500 个请求（`-n` 一共 500 个请求，`-c` 一个时刻执行请求的数量，每次 100 个并发请求）：

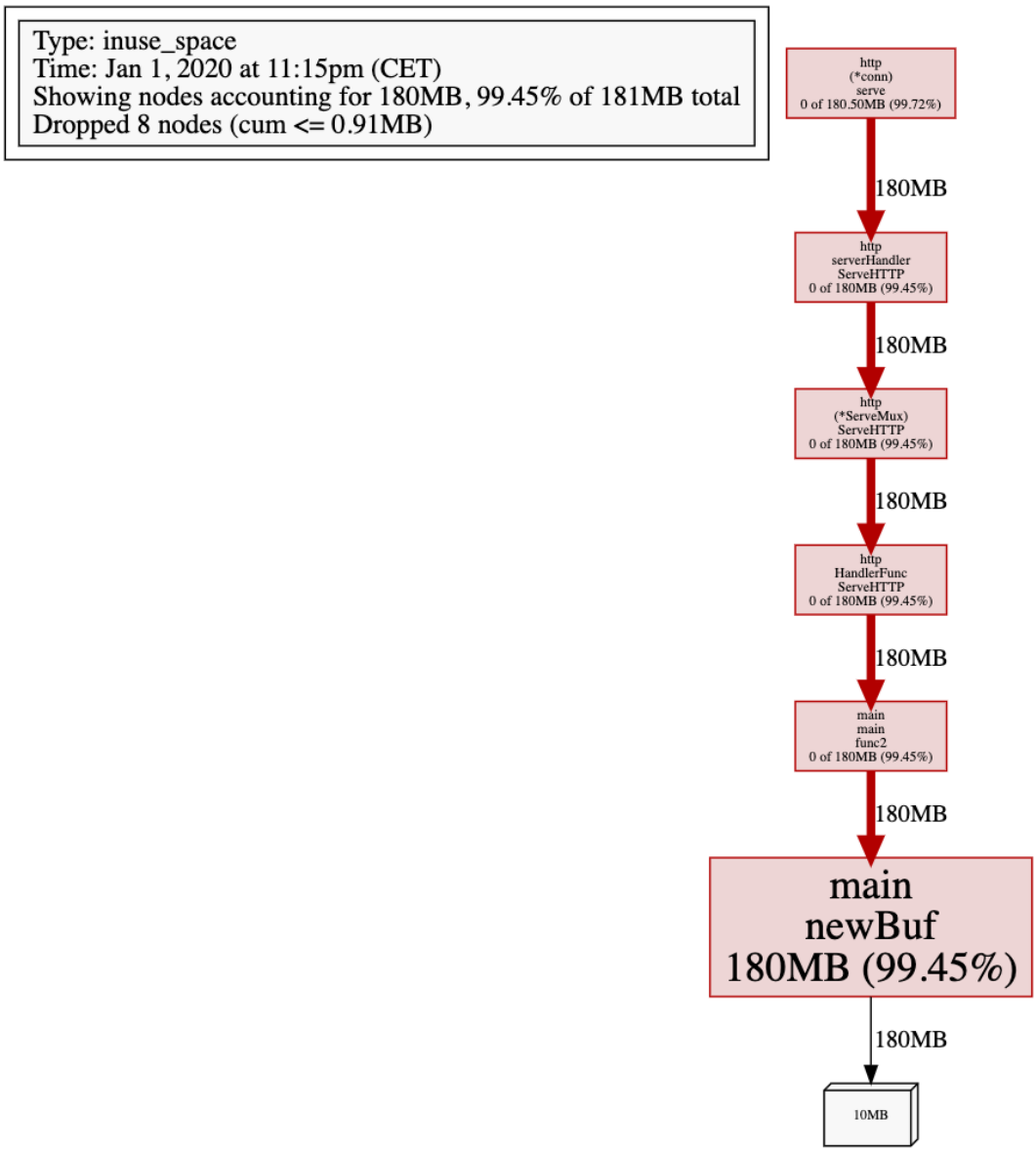
```
$ ab -n 500 -c 100 http://127.0.0.1:8080/example2
This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 127.0.0.1 (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Finished 500 requests


Server Software:
Server Hostname:        127.0.0.1
Server Port:            8080

Document Path:          /example2
Document Length:        14 bytes

Concurrency Level:      100
Time taken for tests:   0.987 seconds
Complete requests:      500
Failed requests:        0
Total transferred:      65500 bytes
HTML transferred:       7000 bytes
Requests per second:    506.63 [#/sec] (mean)
Time per request:       197.382 [ms] (mean)
Time per request:       1.974 [ms] (mean, across all concurrent requests)
Transfer rate:          64.81 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   1.1      0       7
Processing:    13  179  77.5    170     456
Waiting:       10  168  78.8    162     455
Total:         14  180  77.3    171     458

Percentage of the requests served within a certain time (ms)
  50%    171
  66%    203
  75%    222
  80%    239
  90%    281
```

```
 95%    335
 98%    365
 99%    400
100%    458 (longest request)
```




GC 反复被触发，一个显而易见的原因就是内存分配过多。我们可以通过 `go tool pprof` 来查看究竟是谁分配了大量内存（使用 web 指令来使用浏览器打开统计信息的可视化图形）：

```
$ go tool pprof http://127.0.0.1:6060/debug/pprof/heap
Fetching profile over HTTP from http://localhost:6060/debug/pprof/heap
Saved profile in /Users/changkun/pprof/pprof.alloc_objects.alloc_space.inuse_o
bjects.inuse_space.003.pb.gz
Type: inuse_space
Time: Jan 1, 2020 at 11:15pm (CET)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) web
(pprof)
```




可见 `newBuf` 产生的申请的内存过多，现在我们使用 sync.Pool 来复用 `newBuf` 所产生的对象：

```
package main

import (
    "fmt"
    "net/http"
    _ "net/http/pprof"
    "sync"
)

// 使用 sync.Pool 复用需要的 buf
var bufPool = sync.Pool{
    New: func() interface{} {
        return make([]byte, 10<<20)
    },
}
```

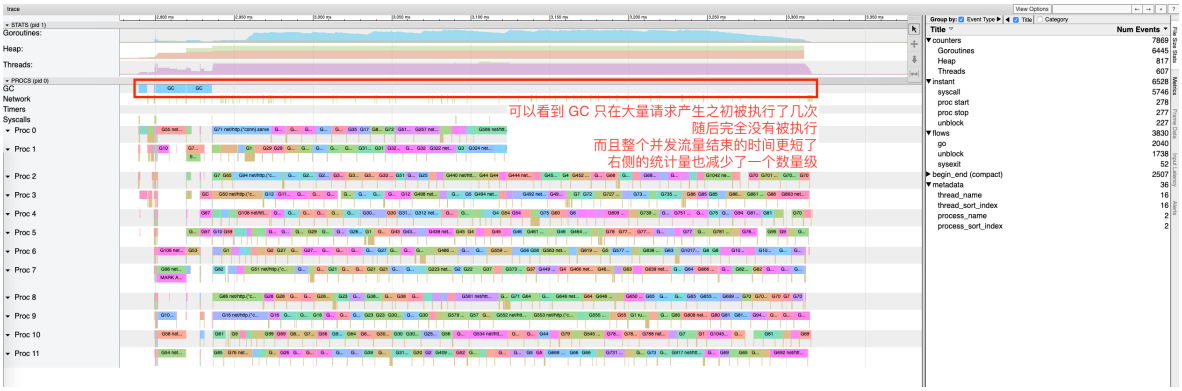
```
func main() {
    go func() {
        http.ListenAndServe("localhost:6060", nil)
    }()
    http.HandleFunc("/example2", func(w http.ResponseWriter, r *http.Request) {
        b := bufPool.Get().([]byte)
        for idx := range b {
            b[idx] = 0
        }
        fmt.Fprintf(w, "done, %v", r.URL.Path[1:])
        bufPool.Put(b)
    })
    http.ListenAndServe(":8080", nil)
}
```

其中 ab 输出的统计结果为：

```
$ ab -n 500 -c 100 http://127.0.0.1:8080/example2
This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 127.0.0.1 (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Finished 500 requests


Server Software:
Server Hostname:        127.0.0.1
Server Port:            8080

Document Path:          /example2
Document Length:        14 bytes

Concurrency Level:      100
Time taken for tests:   0.427 seconds
Complete requests:      500
Failed requests:        0
Total transferred:      65500 bytes
HTML transferred:       7000 bytes
Requests per second:    1171.32 [#/sec] (mean)
Time per request:       85.374 [ms] (mean)
Time per request:       0.854 [ms] (mean, across all concurrent requests)
Transfer rate:          149.85 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   1.4      1       9
Processing:     5   75  48.2     66     211
Waiting:        5   72  46.8     63     207
Total:          5   77  48.2     67     211

Percentage of the requests served within a certain time (ms)
  50%     67
  66%     89
  75%    107
```

```
 80%    122
 90%    148
 95%    167
 98%    196
 99%    204
100%    211 (longest request)
```

但从 `Requests per second` 每秒请求数来看，从原来的 506.63 变为 1171.32 得到了近乎一倍的提升。从 trace 的结果来看，GC 也没有频繁的被触发从而长期消耗 CPU 使用率：




sync.Pool 是内存复用的一个最为显著的例子，从语言层面上还有很多类似的例子，例如在例 1 中，`concat` 函数可以预先分配一定长度的缓存，而后再通过 append 的方式将字符串存储到缓存中：

```
func concat() {
    wg := sync.WaitGroup{}
    for n := 0; n < 100; n++ {
        wg.Add(8)
        for i := 0; i < 8; i++ {
            go func() {
                s := make([]byte, 0, 20)
                s = append(s, "Go GC"...)
                s = append(s, ' ')
                s = append(s, "Hello"...)
                s = append(s, ' ')
                s = append(s, "World"...)
                _ = string(s)
                wg.Done()
            }()
        }
        wg.Wait()
    }
}
```

原因在于 `+` 运算符会随着字符串长度的增加而申请更多的内存，并将内容从原来的内存位置拷贝到新的内存位置，造成大量不必要的内存分配，先提前分配好足够的内存，再慢慢地填充，也是一种减少内存分配、复用内存形式的一种表现。

## 例3：调整 GOGC

我们已经知道了 GC 的触发原则是由步调算法来控制的，其关键在于估计下一次需要触发 GC 时，堆的大小。可想而知，如果我们在遇到海量请求的时，为了避免 GC 频繁触发，是否可以通过将 GOGC 的值设置得更大，让 GC 触发的时间变得更晚，从而减少其触发频率，进而增加用户代码对机器的使用率呢？答案是肯定的。

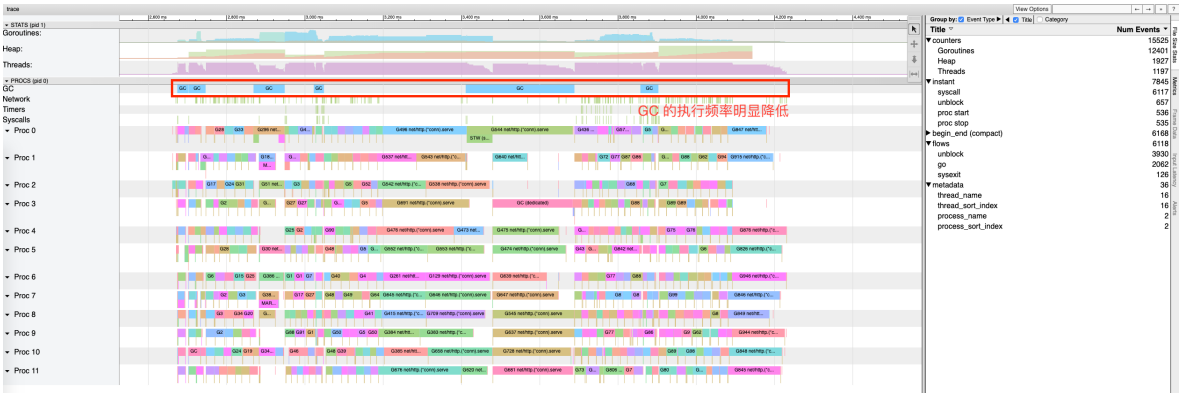
我们可以非常简单粗暴的将 GOGC 调整为 1000，来执行上一个例子中未复用对象之前的程序：

```
$ GOGC=1000 ./main
```

这时我们再重新执行压测：

```
$ ab -n 500 -c 100 http://127.0.0.1:8080/example2
This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 127.0.0.1 (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Finished 500 requests


Server Software:
Server Hostname:        127.0.0.1
Server Port:            8080

Document Path:          /example2
Document Length:        14 bytes

Concurrency Level:      100
Time taken for tests:   0.923 seconds
Complete requests:      500
Failed requests:        0
Total transferred:      65500 bytes
HTML transferred:       7000 bytes
Requests per second:    541.61 [#/sec] (mean)
Time per request:       184.636 [ms] (mean)
Time per request:       1.846 [ms] (mean, across all concurrent requests)
Transfer rate:          69.29 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   1.8      0      20
Processing:     9  171 210.4     66     859
Waiting:        5  158 199.6     62     813
Total:          9  173 210.6     68     860

Percentage of the requests served within a certain time (ms)
  50%     68
  66%    133
  75%    198
  80%    292
  90%    566
  95%    696
  98%    723
  99%    743
 100%    860 (longest request)
```

可以看到，压测的结果得到了一定幅度的改善（ `Requests per second` 从原来的 506.63 提高为了 541.61），并且 GC 的执行频率明显降低：




在实际实践中可表现为需要紧急处理一些由 GC 带来的瓶颈时，人为将 GOGC 调大，加钱加内存，扛过这一段峰值流量时期。

当然，这种做法其实是治标不治本，并没有从根本上解决内存分配过于频繁的问题，极端情况下，反而会由于 GOGC 太大而导致回收不及时而耗费更多的时间来清理产生的垃圾，这对时间不算敏感的应用还好，但对实时性要求较高的程序来说就是致命的打击了。

因此这时更妥当的做法仍然是，定位问题的所在，并从代码层面上进行优化。

## 小结

通过上面的三个例子我们可以看到在 GC 调优过程中 `go tool pprof` 和 `go tool trace` 的强大作用是帮助我们快速定位 GC 导致瓶颈的具体位置，但这些例子中仅仅覆盖了其功能的很小一部分，我们也没有必要完整覆盖所有的功能，因为总是可以通过http pprof 官方文档[7]、runtime pprof官方文档[8]以及 trace 官方文档[9]来举一反三。

现在我们来总结一下前面三个例子中的优化情况：

1. 控制内存分配的速度，限制 goroutine 的数量，从而提高赋值器对 CPU 的利用率。
2. 减少并复用内存，例如使用 sync.Pool 来复用需要频繁创建临时对象，例如提前分配足够的内存来降低多余的拷贝。
3. 需要时，增大 GOGC 的值，降低 GC 的运行频率。

这三种情况几乎涵盖了 GC 调优中的核心思路，虽然从语言上还有很多小技巧可说，但我们并不会在这里事无巨细的进行总结。实际情况也是千变万化，我们更应该着重于培养具体问题具体分析的能力。

当然，我们还应该谨记 **过早优化是万恶之源**这一警语，在没有遇到应用的真正瓶颈时，将宝贵的时间分配在开发中其他优先级更高的任务上。

# Go 的垃圾回收器有哪些相关的 API？其作用分别是什么？

## 15. Go 的垃圾回收器有哪些相关的 API？其作用分别是什么？

在 Go 中存在数量极少的与 GC 相关的 API，它们是

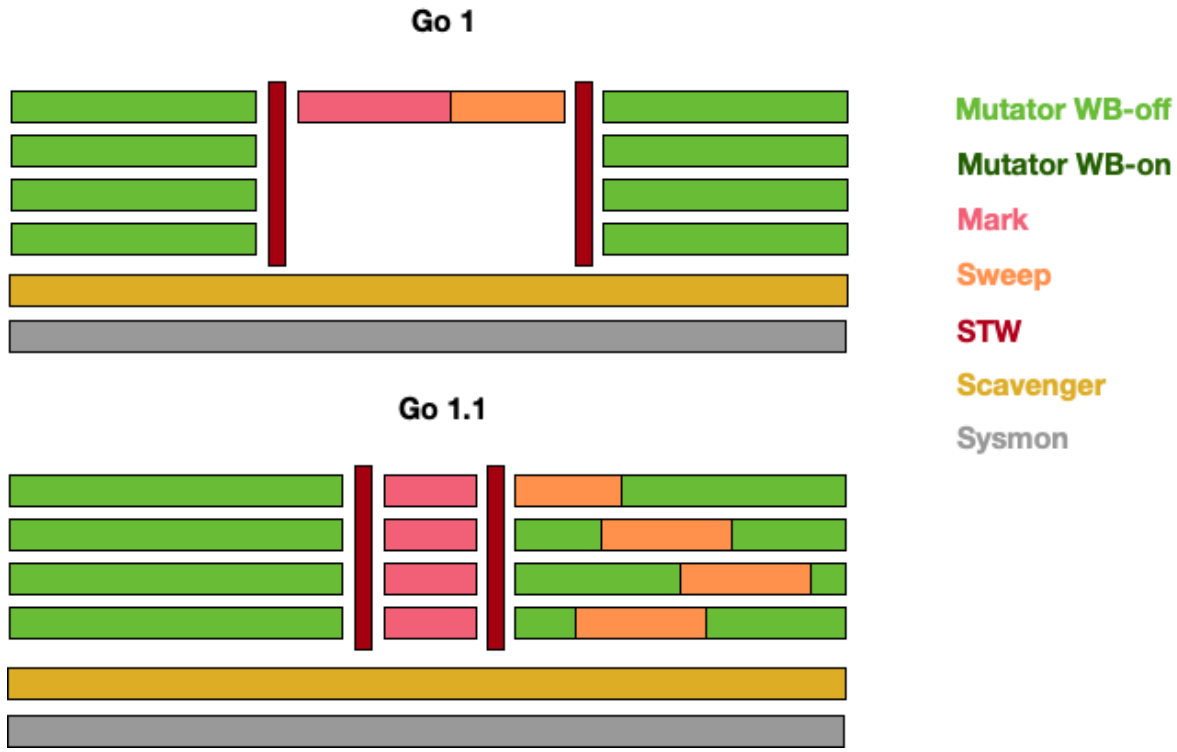
- runtime.GC：手动触发 GC
- runtime.ReadMemStats：读取内存相关的统计信息，其中包含部分 GC 相关的统计信息
- debug.FreeOSMemory：手动将内存归还给操作系统
- debug.ReadGCStats：读取关于 GC 的相关统计信息
- debug.SetGCPercent：设置 GOGC 调步变量
- debug.SetMaxHeap（尚未发布[10]）：设置 Go 程序堆的上限值

# Go 历史各个版本在 GC 方面的改进？

## 16. Go 历史各个版本在 GC 方面的改进？

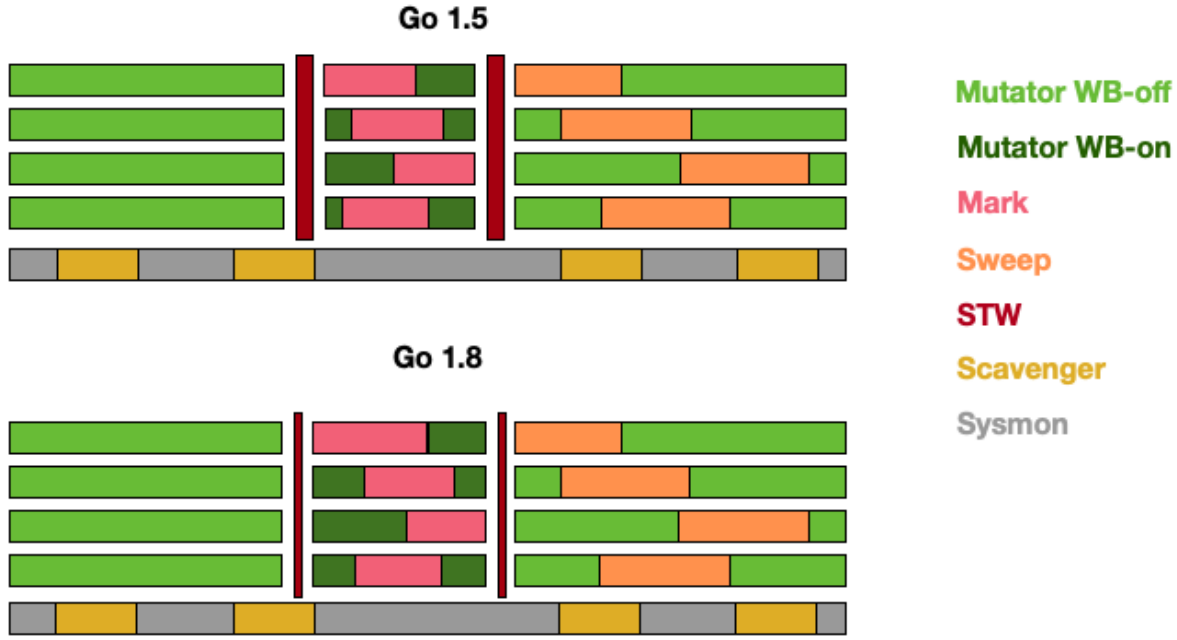
- Go 1：串行三色标记清扫
- Go 1.3：并行清扫，标记过程需要 STW，停顿时间在约几百毫秒
- Go 1.5：并发标记清扫，停顿时间在一百毫秒以内
- Go 1.6：使用 bitmap 来记录回收内存的位置，大幅优化垃圾回收器自身消耗的内存，停顿时间在十毫秒以内
- Go 1.7：停顿时间控制在两毫秒以内
- Go 1.8：混合写屏障，停顿时间在半个毫秒左右
- Go 1.9：彻底移除了栈的重扫描过程
- Go 1.12：整合了两个阶段的 Mark Termination，但引入了一个严重的 GC Bug 至今未修（见问题 20），尚无该 Bug 对 GC 性能影响的报告
- Go 1.13：着手解决向操作系统归还内存的，提出了新的 Scavenger
- Go 1.14：替代了仅存活了一个版本的 scavenger，全新的页分配器，优化分配内存过程的速率与现有的扩展性问题，并引入了异步抢占，解决了由于密集循环导致的 STW 时间过长的问题

可以用下图直观地说明 GC 的演进历史：



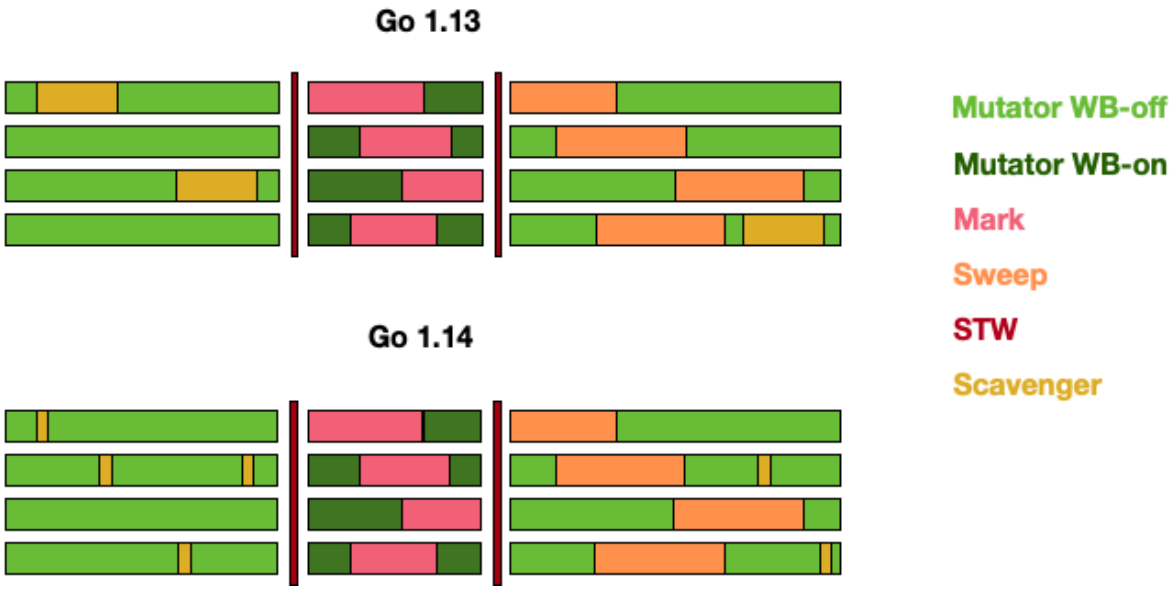

在 Go 1 刚发布时的版本中，甚至没有将 Mark-Sweep 的过程并行化，当需要进行垃圾回收时，所有的代码都必须进入 STW 的状态。而到了 Go 1.3 时，官方迅速地将清扫过程进行了并行化的处理，即仅在标记阶段进入 STW。

这一想法很自然，因为并行化导致算法结果不一致的情况仅仅发生在标记阶段，而当时的垃圾回收器没有针对并行结果的一致性进行任何优化，因此才需要在标记阶段进入 STW。对于 Scavenger 而言，早期的版本中会有一个单独的线程来定期将多余的内存归还给操作系统。




而到了 Go 1.5 后，Go 团队花费了相当大的力气，通过引入写屏障的机制来保证算法的一致性，才得以将整个 GC 控制在很小的 STW 内，而到了 1.8 时，由于新的混合屏障的出现，消除了对栈本身的重新扫描，STW 的时间进一步缩减。

从这个时候开始，Scavenger 已经从独立线程中移除，并合并至系统监控这个独立的线程中，并周期性地向操作系统归还内存，但仍然会有内存溢出这种比较极端的情况出现，因为程序可能在短时间内应对突发性的内存申请需求时，内存还没来得及归还操作系统，导致堆不断向操作系统申请内存，从而出现内存溢出。




到了 Go 1.13，定期归还操作系统的问题得以解决，Go 团队开始将周期性的 Scavenger 转化为可被调度的 goroutine，并将其与用户代码并发执行。而到了 Go 1.14，这一向操作系统归还内存的操作时间进一步得到缩减。

# Go GC 在演化过程中还存在哪些其他设计？为什么没有被采用？

## 17. Go GC 在演化过程中还存在哪些其他设计？为什么没有被采用？

### 并发栈重扫

正如我们前面所说，允许灰色赋值器存在的垃圾回收器需要引入重扫过程来保证算法的正确性，除了引入混合屏障来消除重扫这一过程外，有另一种做法可以提高重扫过程的性能，那就是将重扫的过程并发执行。然而这一方案[11]并没有得以实现，原因很简单：实现过程相比引入混合屏障而言十分复杂，而且引入混合屏障能够消除重扫这一过程，将简化垃圾回收的步骤。

### ROC

ROC 的全称是面向请求的回收器（Request Oriented Collector）[12]，它其实也是分代 GC 的一种重新叙述。它提出了一个请求假设（Request Hypothesis）：与一个完整请求、休眠 goroutine 所关联的对象比其他对象更容易死亡。这个假设听起来非常符合直觉，但在实现上，由于垃圾回收器必须确保是否有 goroutine 私有指针被写入公共对象，因此写屏障必须一直打开，这也就产生了该方法的致命缺点：昂贵的写屏障及其带来的缓存未命中，这也是这一设计最终没有被采用的主要原因。

### 传统分代 GC

在发现 ROC 性能不行之后，作为备选方案，Go 团队还尝试了实现传统的分代式 GC [13]。但最终同样发现分代假设并不适用于 Go 的运行栈机制，年轻代对象在栈上就已经死亡，扫描本就该回收的执行栈并没有为由于分代假设带来明显的性能提升。这也是这一设计最终没有被采用的主要原因。

# 目前提供 GC 的语言以及不提供 GC 的语言有哪些？GC 和 No GC 各自的优缺点是什么？

## 18. 目前提供 GC 的语言以及不提供 GC 的语言有哪些？GC 和 No GC 各自的优缺点是什么？

从原理上而言，所有的语言都能够自行实现 GC。从语言诞生之初就提供 GC 的语言，例如：

- Python
- JavaScript
- Java
- Objective-C
- Swift

而不以 GC 为目标，被直接设计为手动管理内存、但可以自行实现 GC 的语言有：

- C
- C++

也有一些语言可以在编译期，依靠编译器插入清理代码的方式，实现精准的清理，例如：

- Rust

垃圾回收使程序员无需手动处理内存释放，从而能够消除一些需要手动管理内存才会出现的运行时错误：

1. 在仍然有指向内存区块的指针的情况下释放这块内存时，会产生悬挂指针，从而后续可能错误的访问已经用于他用的内存区域。
2. 多重释放同一块申请的内存区域可能导致不可知的内存损坏。

当然，垃圾回收也会伴随一些缺陷，这也就造就了没有 GC 的一些优势：

1. 没有额外的性能开销
2. 精准的手动内存管理，极致的利用机器的性能

# Go 对比 Java、V8 中 JavaScript 的 GC 性能如何？

## 19. Go 对比 Java、V8 中 JavaScript 的 GC 性能如何？

无论是 Java 还是 JavaScript 中的 GC 均为分代式 GC。分代式 GC 的一个核心假设就是分代假说：将对象依据存活时间分配到不同的区域，每次回收只回收其中的一个区域。

### V8 的 GC

在 V8 中主要将内存分为新生代和老生代。新生代中的对象为存活时间较短的对象，老生代中的对象为存活时间较长、常驻内存、占用内存较大的对象：

1. 新生代中的对象主要通过副垃圾回收器进行回收。该回收过程是一种采用复制的方式实现的垃圾回收算法，它将堆内存一分为二，这两个空间中只有一个处于使用中，另一个则处于闲置状态。处于使用状态的空间称为 From 空间，处于闲置的空间称为 To 空间。分配对象时，先是在 From 空间中进行分配，当开始垃圾回收时，会检查 From 空间中的存活对象，并将这些存活对象复制到 To 空间中，而非存活对象占用的空间被释放。完成复制后，From 空间和 To 空间的角色互换。也就是通过将存活对象在两个空间中进行复制。
2. 老生代则由主垃圾回收器负责。它实现的是标记清扫过程，但略有不同之处在于它还会在清扫完成后对内存碎片进行整理，进而是一种标记整理的回收器。

### Java 的 GC

Java 的 GC 称之为 G1，并将整个堆分为年轻代、老年代和永久代。包括四种不同的收集操作，从上往下的这几个阶段会选择性地执行，触发条件是用户的配置和实际代码行为的预测。

1. 年轻代收集周期：只对年轻代对象进行收集与清理
2. 老年代收集周期：只对老年代对象进行收集与清理
3. 混合式收集周期：同时对年轻代和老年代进行收集与清理
4. 完整 GC 周期：完整的对整个堆进行收集与清理

在回收过程中，G1 会对停顿时间进行预测，竭尽所能地调整 GC 的策略从而达到用户代码通过系统参数（ `-XX:MaxGCPauseMillis` ）所配置的对停顿时间的要求。

这四个周期的执行成本逐渐上升，优化得当的程序可以完全避免完整 GC 周期。

### 性能比较

在 Go、Java 和 V8 JavaScript 之间比较 GC 的性能本质上是一个不切实际的问题。如前面所说，垃圾回收器的设计权衡了很多方面的因素，同时还受语言自身设计的影响，因为语言的设计也直接影响了程序员编写代码的形式，也就自然影响了产生垃圾的方式。

但总的来说，他们三者对垃圾回收的实现都需要 STW，并均已达到了用户代码几乎无法感知到的状态（据 Go GC 作者 Austin 宣称STW 小于 100 微秒 [14]）。当然，随着 STW 的减少，垃圾回收器会增加 CPU 的使用率，这也是程序员在编写代码时需要手动进行优化的部分，即充分考虑内存分配的必要性，减少过多申请内存带给垃圾回收器的压力。

# 目前 Go 语言的 GC 还存在哪些问题？

## 20. 目前 Go 语言的 GC 还存在哪些问题？

尽管 Go 团队宣称 STW 停顿时间得以优化到 100 微秒级别，但这本质上是一种取舍。原本的 STW 某种意义上来说其实转移到了可能导致用户代码停顿的几个位置；除此之外，由于运行时调度器的实现方式，同样对 GC 存在一定程度的影响。

目前 Go 中的 GC 仍然存在以下问题：

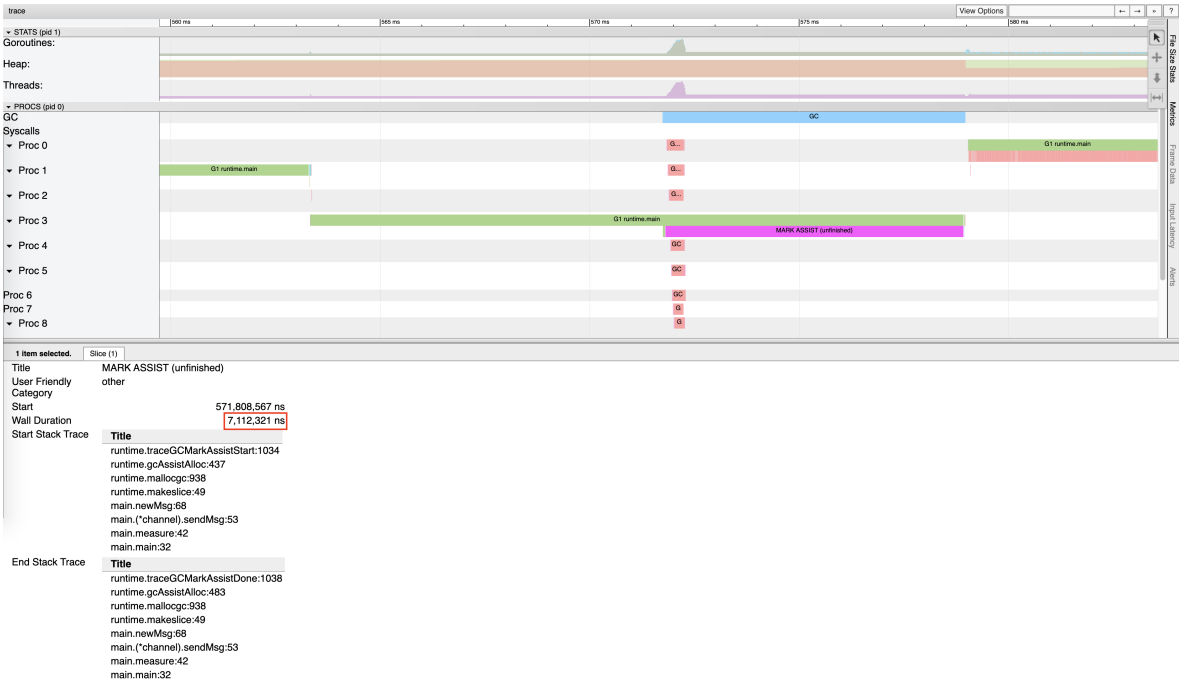
### 1. Mark Assist 停顿时间过长

```
package main

import (
    "fmt"
    "os"
    "runtime"
    "runtime/trace"
    "time"
)

const (
    windowSize = 200000
    msgCount   = 1000000
)

var (
    best    time.Duration = time.Second
    bestAt  time.Time
    worst   time.Duration
    worstAt time.Time

    start = time.Now()
)

func main() {
    f, _ := os.Create("trace.out")
    defer f.Close()
    trace.Start(f)
    defer trace.Stop()

    for i := 0; i < 5; i++ {
        measure()
        worst = 0
        best = time.Second
        runtime.GC()
    }
}

func measure() {
    var c channel
    for i := 0; i < msgCount; i++ {
        c.sendMsg(i)
    }
    fmt.Printf("Best send delay %v at %v, worst send delay: %v at %v. Wall clock: %v \n", best, bestAt.Sub(st
art), worst, worstAt.Sub(start), time.Since(start))
```

```
}

type channel [windowSize][]byte

func (c *channel) sendMsg(id int) {
    start := time.Now()

    // 模拟发送
    (*c)[id%windowSize] = newMsg(id)

    end := time.Now()
    elapsed := end.Sub(start)
    if elapsed > worst {
        worst = elapsed
        worstAt = end
    }
    if elapsed < best {
        best = elapsed
        bestAt = end
    }
}

func newMsg(n int) []byte {
    m := make([]byte, 1024)
    for i := range m {
        m[i] = byte(n)
    }
    return m
}
```

运行此程序我们可以得到类似下面的结果：

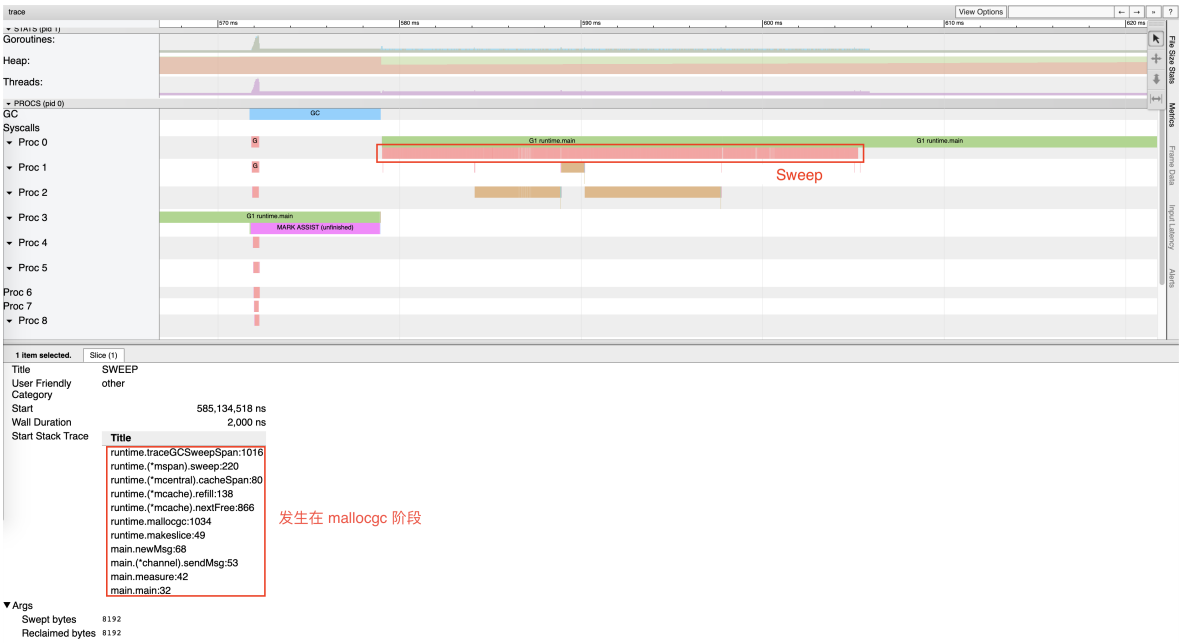
```
$ go run main.go

Best send delay 330ns at 773.037956ms, worst send delay: 7.127915ms at 579.835487ms. Wall clock: 831.066632ms
Best send delay 331ns at 873.672966ms, worst send delay: 6.731947ms at 1.023969626s. Wall clock: 1.515295559s
Best send delay 330ns at 1.812141567s, worst send delay: 5.34028ms at 2.193858359s. Wall clock: 2.199921749s
Best send delay 338ns at 2.722161771s, worst send delay: 7.479482ms at 2.665355216s. Wall clock: 2.920174197s
Best send delay 337ns at 3.173649445s, worst send delay: 6.989577ms at 3.361716121s. Wall clock: 3.615079348s
```




在这个结果中，第一次的最坏延迟时间高达 7.12 毫秒，发生在程序运行 578 毫秒左右。通过 `go tool trace` 可以发现，这个时间段中，Mark Assist 执行了 7112312ns，约为 7.127915ms；可见，此时最坏情况下，标记辅助拖慢了用户代码的执行，是造成 7 毫秒延迟的原因。

## 2. Sweep 停顿时间过长

同样还是刚才的例子，如果我们仔细观察 Mark Assist 后发生的 Sweep 阶段，竟然对用户代码的影响长达约 30ms，根据调用栈信息可以看到，该 Sweep 过程发生在内存分配阶段：




## 3. 由于 GC 算法的不正确性导致 GC 周期被迫重新执行

此问题很难复现，但是一个已知的问题，根据 Go 团队的描述，能够在 1334 次构建中发生一次 [15]，我们可以计算出其触发概率约为 0.0007496251874。虽然发生概率很低，但一旦发生，GC 需要被重新执行，非常不幸。

## 4. 创建大量 Goroutine 后导致 GC 消耗更多的 CPU

这个问题可以通过以下程序进行验证：

```
func BenchmarkGCLargeGs(b *testing.B) {
    wg := sync.WaitGroup{}

    for ng := 100; ng <= 1000000; ng *= 10 {
        b.Run(fmt.Sprintf("#g-%d", ng), func(b *testing.B) {
            // 创建大量 goroutine，由于每次创建的 goroutine 会休眠
            // 从而运行时不会复用正在休眠的 goroutine，进而不断创建新的 g
            wg.Add(ng)
            for i := 0; i < ng; i++ {
                go func() {
                    time.Sleep(100 * time.Millisecond)
                    wg.Done()
                }()
            }
            wg.Wait()

            // 现运行一次 GC 来提供一致的内存环境
            runtime.GC()

            // 记录运行 b.N 次 GC 需要的时间
            b.ResetTimer()
            for i := 0; i < b.N; i++ {
                runtime.GC()
            }
        })

    }
}
```

其结果可以通过如下指令来获得：

```
$ go test -bench=BenchmarkGCLargeGs -run=^$ -count=5 -v . | tee 4.txt
$ benchstat 4.txt
name                       time/op
GCLargeGs/#g-100-12         192µs ± 5%
GCLargeGs/#g-1000-12        331µs ± 1%
GCLargeGs/#g-10000-12      1.22ms ± 1%
GCLargeGs/#g-100000-12     10.9ms ± 3%
GCLargeGs/#g-1000000-12    32.5ms ± 4%
```

这种情况通常发生于峰值流量后，大量 goroutine 由于任务等待被休眠，从而运行时不断创建新的 goroutine，
旧的 goroutine 由于休眠未被销毁且得不到复用，导致 GC 需要扫描的执行栈越来越多，进而完成 GC 所需的时间越来越长。
一个解决办法是使用 goroutine 池来限制创建的 goroutine 数量。

# 总结

## 总结

GC 是一个复杂的系统工程，本文讨论的二十个问题尽管已经展现了一个相对全面的 Go GC。
但它们仍然只是 GC 这一宏观问题的一小部分较为重要的内容，还有非常多的细枝末节、研究进展无法在有限的篇幅内完整讨论。

从 Go 诞生之初，Go 团队就一直在对 GC 的表现进行实验与优化，但仍然有诸多未解决的公开问题，我们不妨对 GC 未来的改进拭目以待。

## 进一步阅读的主要参考文献


- [1] Ian Lance Taylor. Why golang garbage-collector not implement Generational and Compact gc? May 2017. https://groups.google.com/forum/#!msg/golang-nuts/KJiyv2mV2pU/wdBUH1mHCAAJ
- [2] Go Team. `debug.GCStats`. Last access: Jan, 2020. https://golang.org/pkg/runtime/debug/#GCStats
- [3] Go Team. `runtime.MemStats`. Last access: Jan, 2020. https://golang.org/pkg/runtime/#MemStats
- [4] Austin Clements, Rick Hudson. Proposal: Eliminate STW stack re-scanning. Oct, 2016. https://github.com/golang/proposal/blob/master/design/17503-eliminate-rescan.md
- [5] Austin Clements. Go 1.5 concurrent garbage collector pacing. Mar, 2015. https://docs.google.com/document/d/1wmjrocXIWTr1JxU-3EQBI6BK6KgtiFArkG47XK73xIQ/edit#
- [6] Austin Clements. Proposal: Separate soft and hard heap size goal. Oct, 2017. https://github.com/golang/proposal/blob/master/design/14951-soft-heap-limit.md
- [7] Go Team. HTTP pprof. Last access: Jan, 2020. https://golang.org/pkg/net/http/pprof/
- [8] Go Team. Runtime pprof. Last access: Jan, 2020. https://golang.org/pkg/runtime/pprof/
- [9] Go Team. Package trace. Last access: Jan, 2020. https://golang.org/pkg/runtime/trace/
- [10] Caleb Spare. proposal: runtime: add a mechanism for specifying a minimum target heap size. Last access: Jan, 2020. https://github.com/golang/go/issues/23044
- [11] Austin Clements, Rick Hudson. Proposal: Concurrent stack re-scanning. Oct, 2016. https://github.com/golang/proposal/blob/master/design/17505-concurrent-rescan.md
- [12] Rick Hudson, Austin Clements. Request Oriented Collector (ROC) Algorithm. Jun, 2016. https://docs.google.com/document/d/1gCsFxXamW8RRvOe5hECz98Ftk-tcRRJcDFANj2VwCB0/edit
- [13] Rick Hudson. runtime: constants and data structures for generational GC. Mar, 2019. https://go-review.googlesource.com/c/go/+/137476/12
- [14] Austin Clements. Sub-millisecond GC pauses. Oct, 2016. https://groups.google.com/d/msg/golang-dev/Ab1sFeoZg_8/_DaL0E8fAwAJ
- [15] Austin Clements. runtime: error message: P has cached GC work at end of mark termination. Nov, 2018. https://github.com/golang/go/issues/27993#issuecomment-441719687


## 其他参考文献


- [16] Dmitry Soshnikov. Writing a Memory Allocator. Feb. 2019. http://dmitrysoshnikov.com/compilers/writing-a-memory-allocator/#more-3590

- [17] William Kennedy. Garbage Collection In Go : Part II - GC Traces. May 2019. https://www.ardanlabs.com/blog/2019/05/garbage-collection-in-go-part2-gctraces.html
- [18] Rhys Hiltner. An Introduction to go tool trace. Last access: Jan, 2020. https://about.sourcegraph.com/go/an-introduction-to-go-tool-trace-rhys-hiltner
- [19] 煎鱼. 用 GODEBUG 看 GC. Sep, 2019. https://segmentfault.com/a/1190000020255157
- [20] 煎鱼. Go 大杀器之跟踪剖析 trace. Last access: Jan, 2020. https://eddycjy.gitbook.io/golang/di-9-ke-gong-ju/go-tool-trace