gc

desc gc

- Golang 采用了标记清除的GC算法
- Golang的标记清除是一个三色标记法的实现，对于三色标记法，"三色"的概念可以简单的理解为:
  - 白色：还没有搜索过的对象（白色对象会被当成垃圾对象）
  - 灰色：正在搜索的对象
  - 黑色：搜索完成的对象（不会当成垃圾对象，不会被GC）

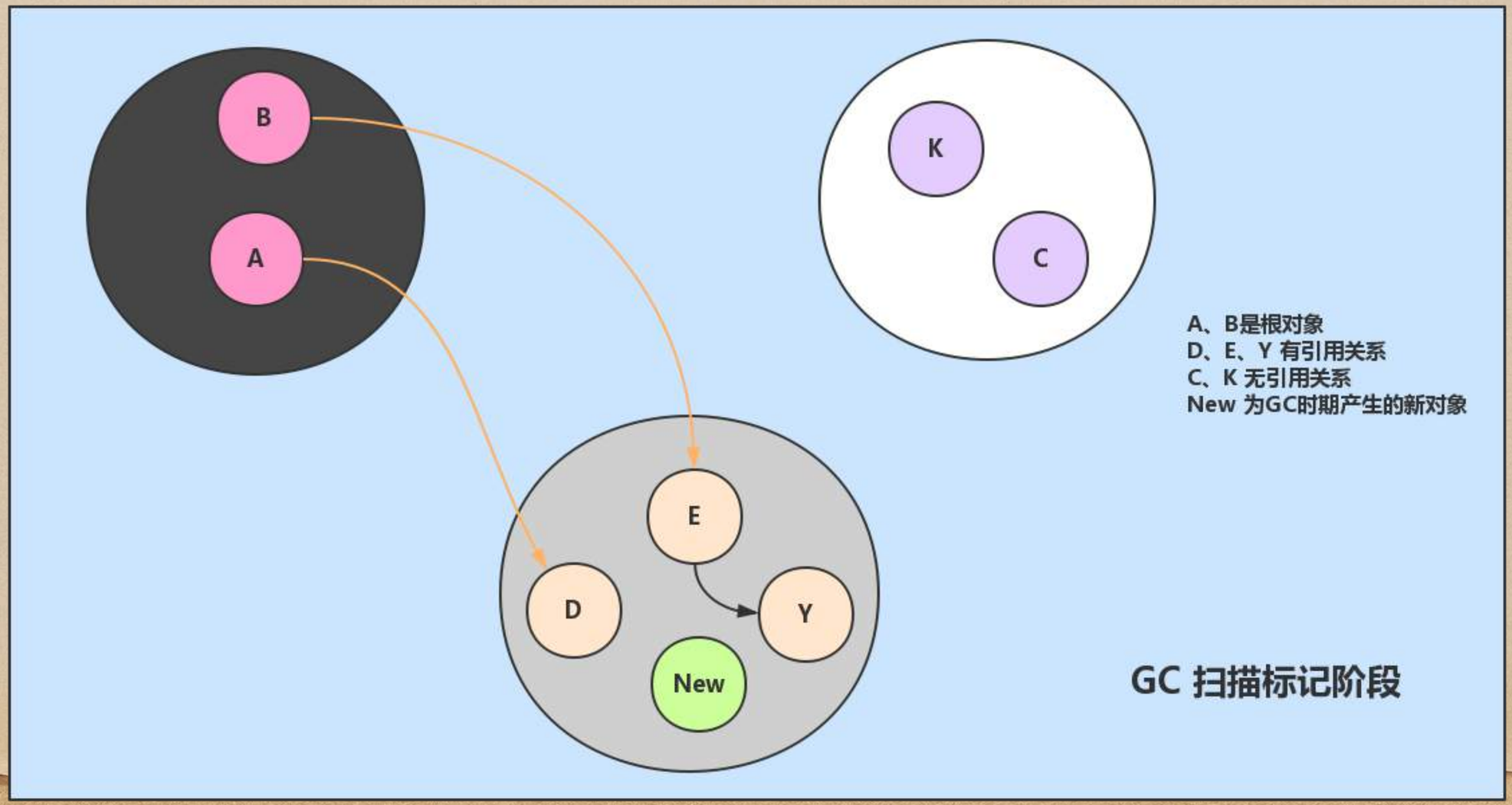
when trigger gc

- gcTriggerHeap
  - 当前分配的内存达到一定值就触发GC
    - default GCGO=100
- gcTriggerTime
  - 当一定时间没有执行过GC就触发GC
  - two minutes
- gcTriggerCycle
  - 要求启动新一轮的GC, 已启动则跳过, 手动触发GC的runtime.GC()会使用这个条件
- gcTriggerAlways
  - 强制触发GC

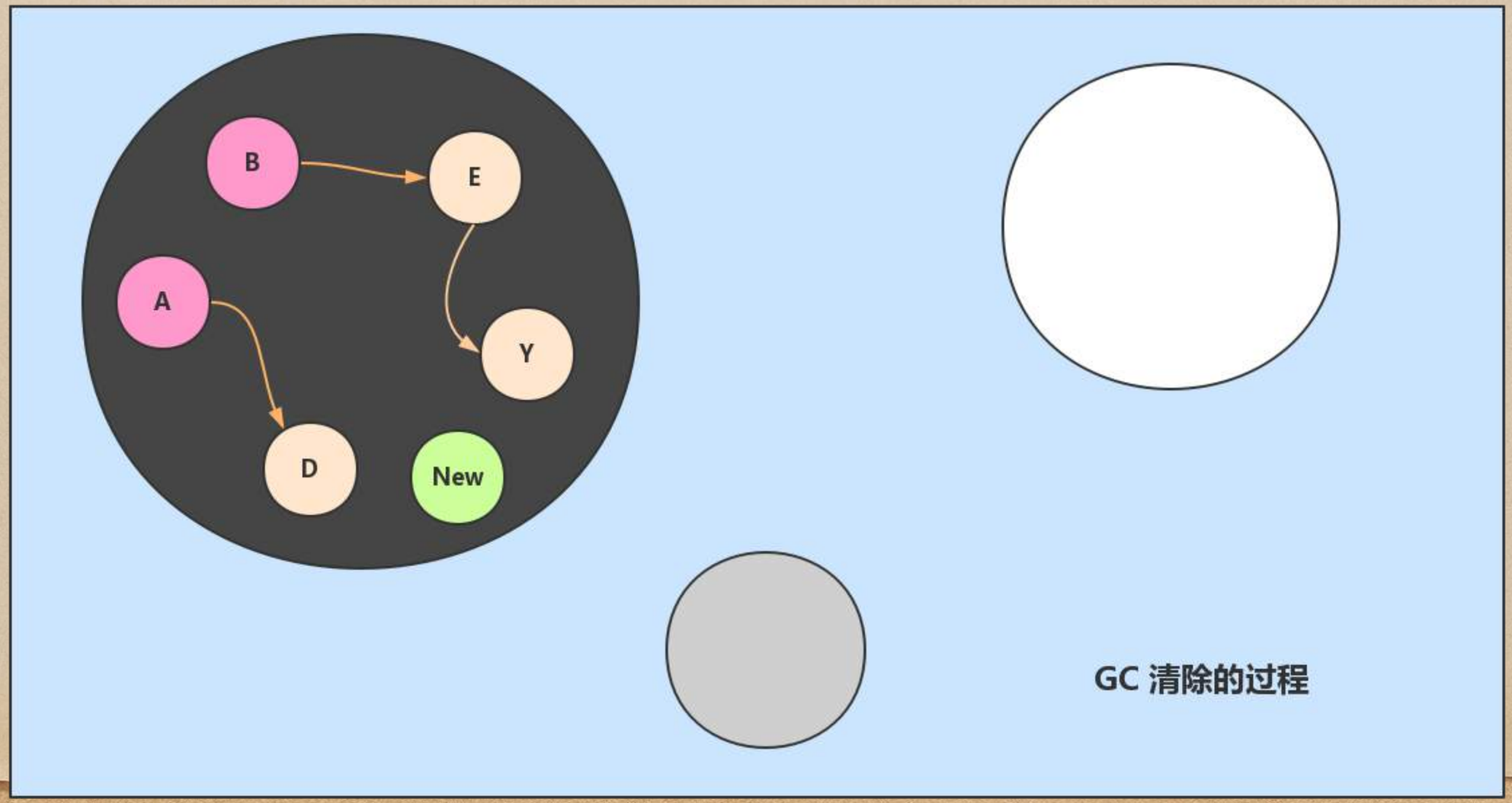
gc process

1. 首先创建三个集合：白、灰、黑。
2. 将所有对象放入白色集合中。
3. 然后从根节点开始遍历所有对象，把可追溯的对象从白色集合放入灰色集合。
4. 之后遍历灰色集合，将灰色对象引用的对象从白色集合放入灰色集合，之后将此灰色对象放入黑色集合。
5. 重复4直到灰色中无任何对象。
6. 通过写屏障检测对象有变化，重复以上操作。
7. 回收所有白色对象

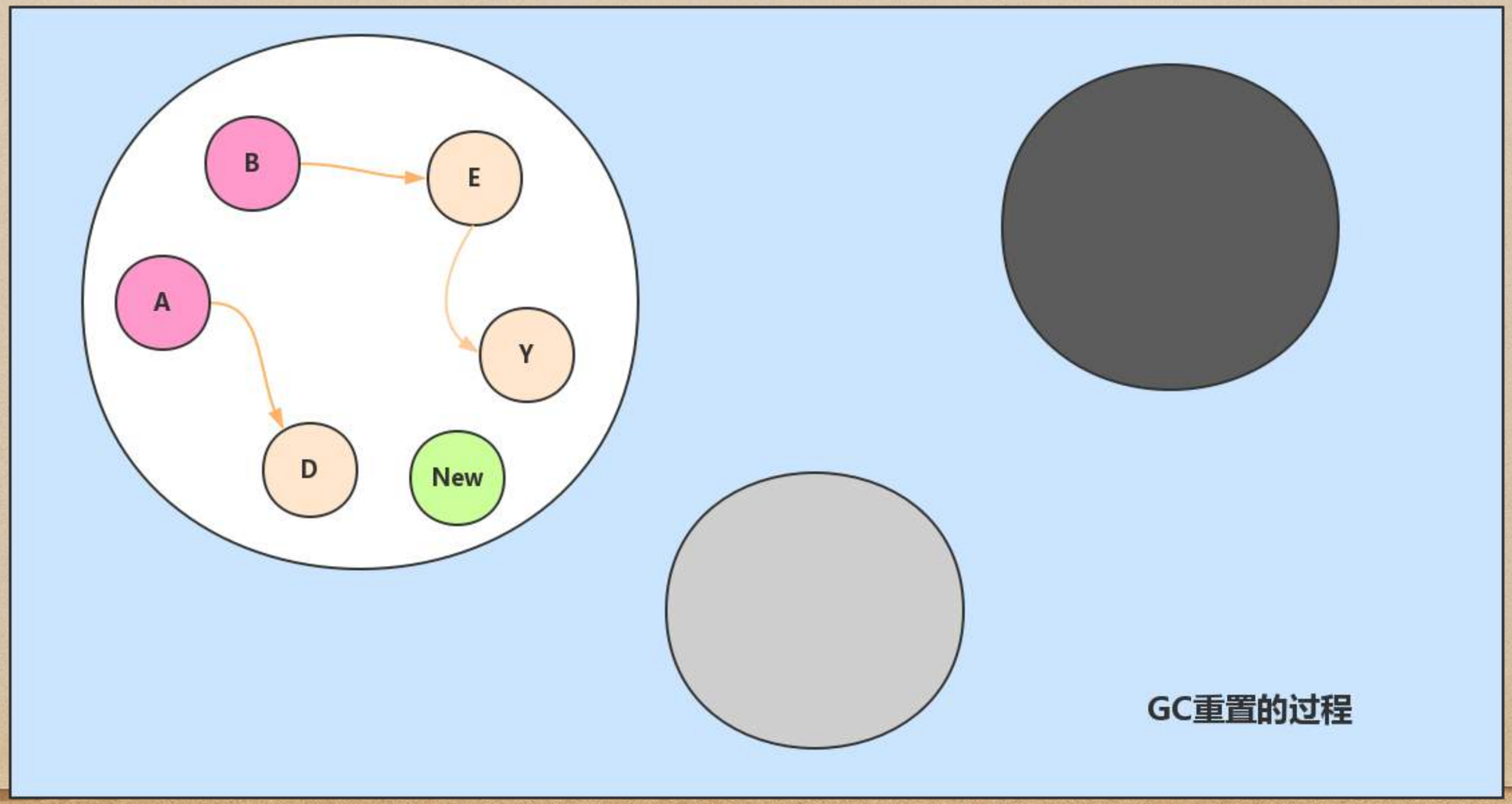
GC 扫描标记


B
A
K
C
E
D
Y
New
A、B是根对象
D、E、Y 有引用关系
C、K 无引用关系
New 为GC时期产生的新对象
GC 扫描标记阶段

GC 清除



GC 重置

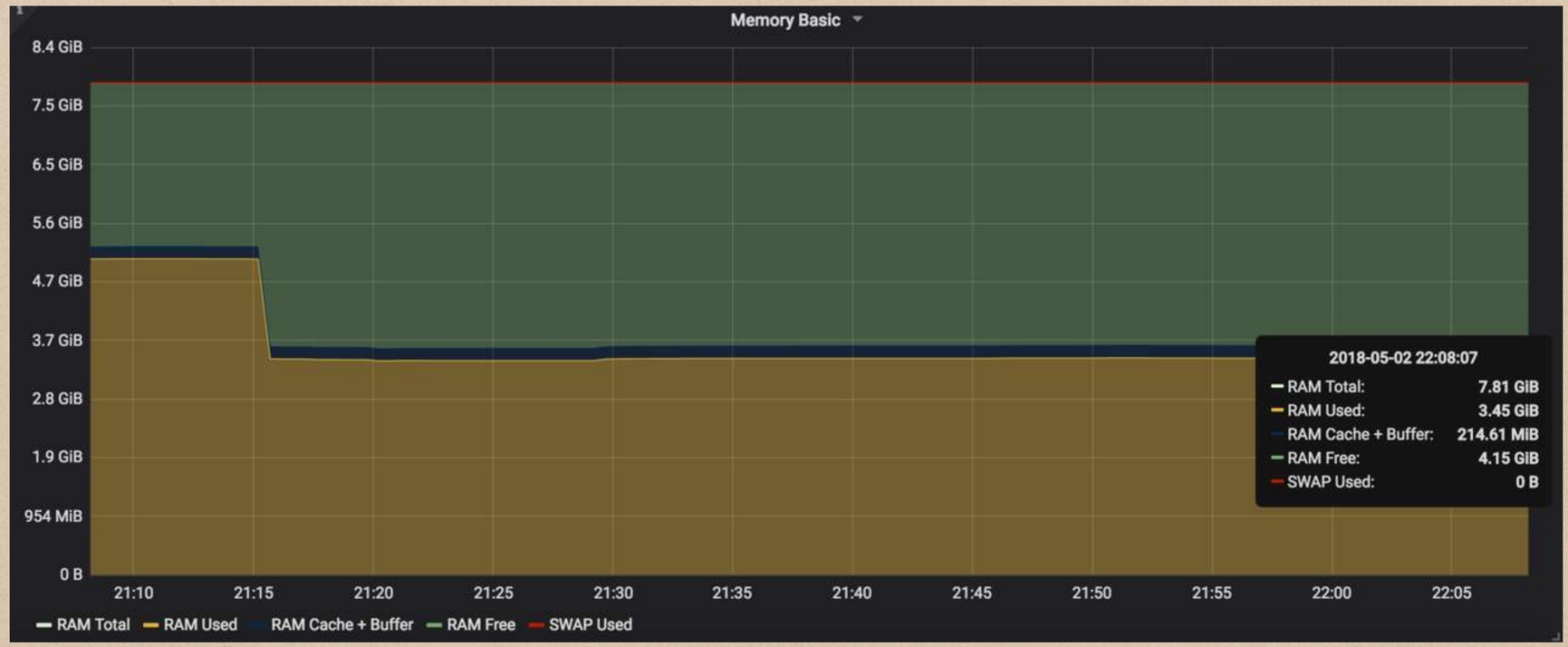


forcegc & return os

```
4219 // This is a variable for testing purposes. It normally doesn't change.
4220 var forcegcperiod int64 = 2 * 60 * 1e9
4221
4222 // Always runs without a P, so write barriers are not allowed.
4223 //
4224 //go:nowritebarrierrec
4225 func sysmon() {
4226     lock(&sched.lock)
4227     sched.nmsys++
4228     checkdead()
4229     unlock(&sched.lock)
4230
4231     // If a heap span goes unused for 5 minutes after a garbage collection,
4232     // we hand it back to the operating system.
4233     scavengelimit := int64(5 * 60 * 1e9)
4234
4235     if debug.scavenge > 0 {
4236         // Scavenge-a-lot for testing.
4237         forcegcperiod = 10 * 1e6
4238         scavengelimit = 20 * 1e6
4239     }
```

- Sysmon 会强制两分钟进行一次 GC
- 每5分钟会释放一些span, 归还操作系统

map的释放情况

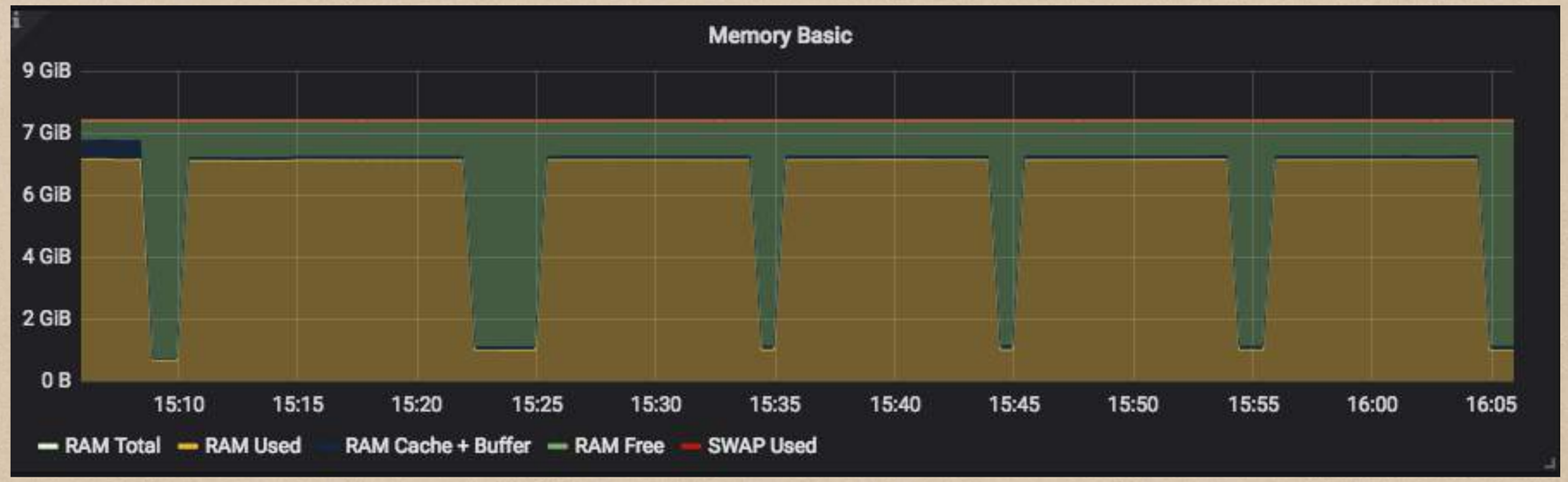


大量的使用big map下会出现对象被清楚，但释放不干净！

map = nil

debug.FreeOSMemory()

channel的释放情况



Channel被清空后，大约10分钟释放内存 .

debug.FreeOSMemory()

元素被gc清除后，什么时候释放内存归还 OS？

编码说 5分钟, 但事实不是这样…

手动触发 FreeOSMemory() 会更好的释放！

debug

GODEBUG=gctrace=1

```
gc 307 @1288.241s 1%: 0.28+73+0.082 ms clock, 0.57+83/42/0+0.16 ms cpu, 212->215->42 MB, 221 MB goal, 2 P
gc 308 @1288.673s 1%: 0.44+83+0.066 ms clock, 0.88+96/45/0+0.13 ms cpu, 202->204->40 MB, 210 MB goal, 2 P
gc 309 @1289.082s 1%: 0.20+70+0.046 ms clock, 0.40+83/40/0+0.093 ms cpu, 194->197->39 MB, 202 MB goal, 2 P
gc 310 @1289.495s 1%: 0.49+68+0.023 ms clock, 0.98+80/37/0+0.047 ms cpu, 190->192->37 MB, 198 MB goal, 2 P
gc 311 @1289.878s 1%: 0.39+61+0.091 ms clock, 0.79+81/30/0+0.18 ms cpu, 180->182->36 MB, 188 MB goal, 2 P
gc 312 @1290.254s 1%: 0.20+71+0.031 ms clock, 0.41+84/41/0+0.062 ms cpu, 175->177->36 MB, 182 MB goal, 2 P
gc 313 @1290.612s 1%: 0.18+69+0.055 ms clock, 0.37+83/39/0+0.11 ms cpu, 173->175->35 MB, 180 MB goal, 2 P
gc 314 @1290.960s 1%: 0.49+51+0.052 ms clock, 0.99+61/30/0+0.10 ms cpu, 172->174->31 MB, 179 MB goal, 2 P
GC forced
gc 12 @1324.050s 0%: 0.012+1.8+0.019 ms clock, 0.025+0/1.8/1.7+0.038 ms cpu, 2->2->2 MB, 16 MB goal, 2 P
```

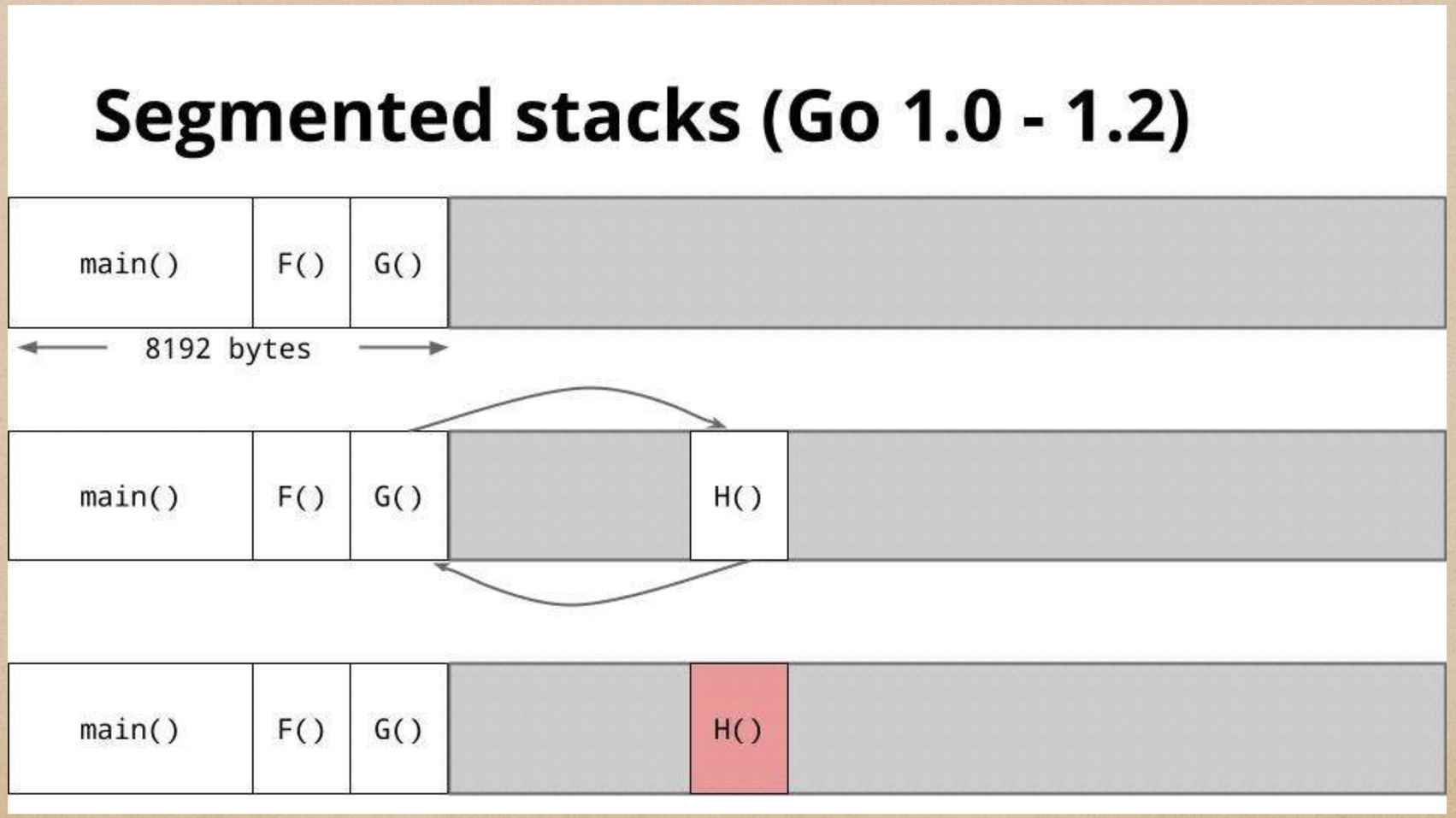
```
func printMem() {
    runtime.ReadMemStats(&mem)
    log.Println("mem.Sys: ", mem.Sys/1024/1024)
    log.Println("mem.Alloc: ", mem.Alloc/1024/1024)
    log.Println("mem.TotalAlloc: ", mem.TotalAlloc/1024/
    log.Println("mem.HeapAlloc: ", mem.HeapAlloc/1024
    log.Println("mem.HeapSys: ", mem.HeapSys/1024/1
    log.Println("mem.HeapObjects: ", mem.HeapObjects
}
```

```
2018/05/02 15:55:36 mem.Sys:  6121
2018/05/02 15:55:36 mem.Alloc:  15
2018/05/02 15:55:36 mem.HeapIdle:  5863
2018/05/02 15:55:36 mem.TotalAlloc:  5882
2018/05/02 15:55:36 mem.HeapAlloc:  15
2018/05/02 15:55:36 mem.HeapSys:  5879
2018/05/02 15:55:36 mem.HeapObjects:  144
```

keyword

- 写屏障？
- 并发GC？
- heap 内存什么时候归还系统?
- golang1.9 gc的改进？

stack


Segmented stacks (Go 1.0 - 1.2)
main()
F()
G()
8192 bytes
main()
F()
G()
H()
main()
F()
G()
H()

netpoller

```
    //判断获取最后一次从网络I/O轮循查找G的时间
    if lastpoll != 0 && lastpoll+10*1000*1000 < now {
    //更新最后一次查询G时间，为了下一次做判断。
    atomic.Cas64(&sched.lastpoll, uint64(lastpoll), uint64(now))
    //从网络轮询器查找已经就绪的，这里是非阻塞读取。
    gp := netpoll(false)

    if gp != nil {
        incidlelocked(-1)
        //找到后注入到调度器下面的可获取的G队列
        injectglist(gp)
        incidlelocked(1)
    }
}
```

sysmon

```
func netpoll(block bool) *g {
    if epfd == -1 {
        return nil
    }
    // ...
    n := epollwait(epfd, &events[0], int32(len(events)), waitms)
    if n < 0 {
        if n != -_EINTR {
            println("runtime: epollwait on fd", epfd, "failed with"
            throw("epollwait failed")
        }
        goto retry
    }
    var gp guintptr
    for i := int32(0); i < n; i++ {
        ev := &events[i]
        if ev.events == 0 {
            continue
        }
        var mode int32
        if ev.events&(_EPOLLIN|_EPOLLRDHUP|_EPOLLHUP|_EPOLLERR) !=
            mode += 'r'
        }
        if ev.events&(_EPOLLOUT|_EPOLLHUP|_EPOLLERR) != 0 {
            mode += 'w'
```

唤醒中 ...