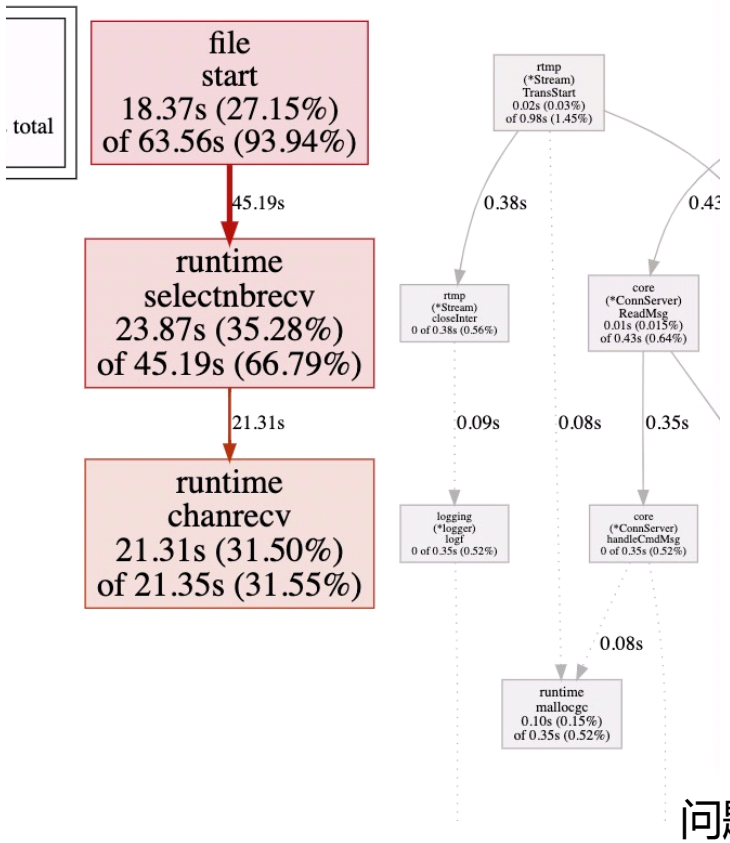
# 深入浅出Golang Runtime

腾讯NOW直播 郝以奋
深圳Gopher Meetup
2019.08.17

yifhao, 郝以奋, 毕业于华中科技大学
腾讯NOW直播后台开发
负责NOW直播 CPP+JAVA双栈 -> Golang转型:
框架协同建设, 业务功能定制, Go Mod引入, 服务模板, RPC协议管理, Golang培训, 文档等
参与内部多部门使用的Golang框架开发, 并在部门成功推广
在内部做过150人的Golang相关分享

# 为什么去了解Runtime?

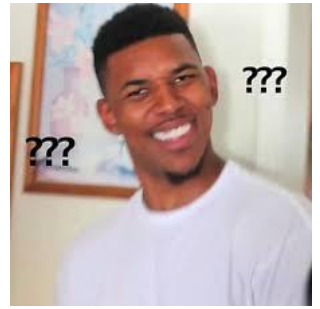
1. 解决疑难杂症&&优化
2. 好奇心
3. 技术深度的一种




```
for {
    select {
    case <-ch:
        {
            //....
        }
    default:
        {
            //....
        }
    }
}
```

问题分析

运行时到底是个什么东西?




Go的调度为什么说是轻量的?
Go调度都发生了啥?
Go的网络和锁会不会阻塞线程?
什么时候会阻塞线程?

Go的对象在内存中是怎样的?
Go的内存分配是怎样的?
栈的内存是怎么分配的?

GC是怎样的?
GC怎么帮我们回收对象?
Go的GC会不会漏掉对象或者回收还在用的对象?
Go GC什么时候开始?
Go GC啥时候结束?
Go GC会不会太慢, 跟不上内存分配的速度?
Go GC会不会暂停我们的应用? 暂停多久? 影不影响我的请求?
....

Runtime简介及历程

调度

内存与GC

实践

问答交流

1. 本次讨论基于最新2019.2发布的Go 1.12 Linux amd64版本
2. PPT中一些参数专门为 Linux amd64版本特化
3. 不同版本有差异, 流程基本类似
4. PPT中的图和文字只限于表达主要流程, 忽略一些分支和异常处理, 实际上源码复杂很多
5. 水平和精力有限, PPT中有错误, 欢迎指出. 代表不了公司其他程序员水平.
6. 有两份PPT, 一份是分享的, 较为精简, 一份是完整的, 均会放出 https://github.com/Frank-Hust/share
7. 有Go相关的问题分享后也可与我线下或微信讨论

希望本次分享和交流, 能够对Golang Runtime发展, 调度原理, 内存分配机制, GC流程有个大概的认识, 能够解决对Go底层的一些疑惑.
相互讨论, 相互学习!

**简介及发展**

调度

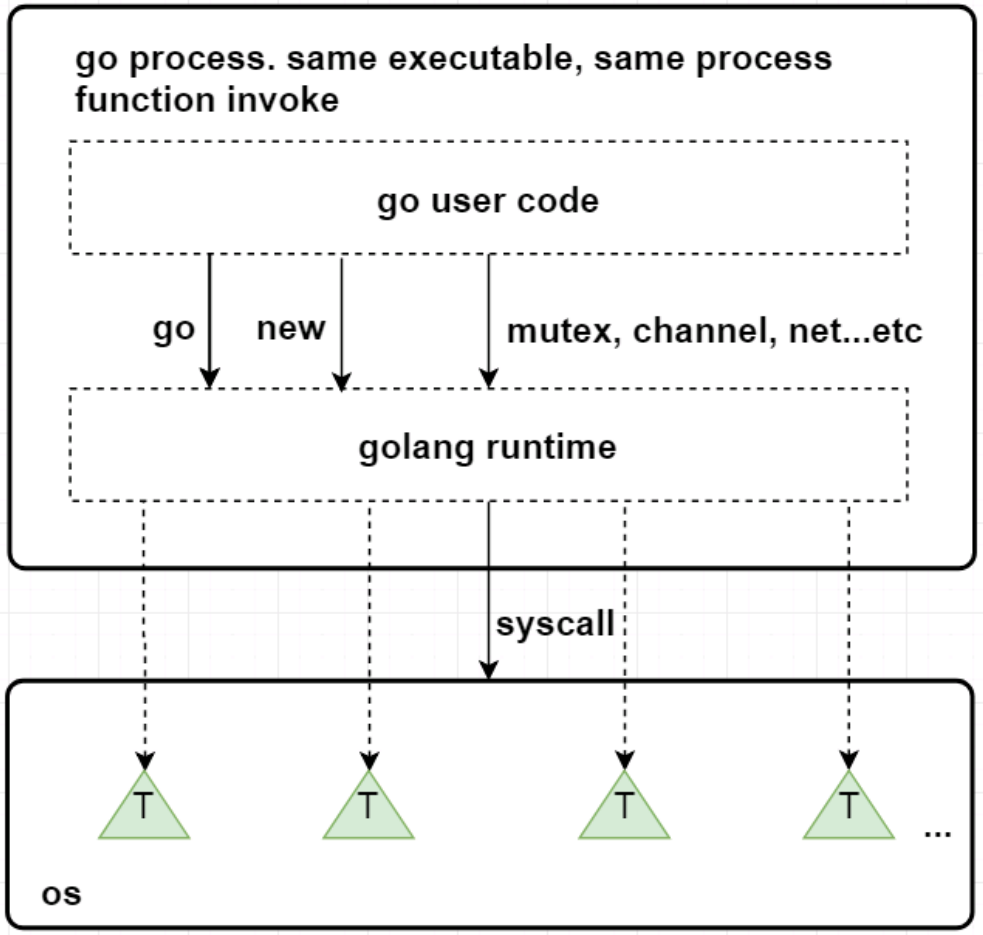
内存与GC

实践

# Golang Runtime简介

Golang Runtime是Go语言运行所需要的基础设施.

1. 协程调度, 内存分配, GC;
2. 操作系统及CPU相关的操作的封装(信号处理, 系统调用, 寄存器操作, 原子操作等), CGO;
3. pprof, trace, race检测的支持;
4. map, channel, string等内置类型及反射的实现.




1. 与Java, Python不同, Go并没有虚拟机的概念, Runtime也直接被编译成native code.
2. Go的Runtime与用户代码一起打包在一个可执行文件中
3. 用户代码与Runtime代码在执行的时候并没有明显的界限, 都是函数调用
4. Go对系统调用的指令进行了封装, 可不依赖于glibc
5. 一些Go的关键字被编译器编译成runtime包下的函数.

| 关键字 | 函数 |
| --- | --- |
| go | newproc |
| new | newobject |
| make | makeslice, makechan, makemap, makemap_small... |
| | gcStart |
| <- -> | chansend1, chanrecv1 |
| 等等 | |

# Runtime发展历程

| 版本 | 发布时间 | 改进特征 | GC STW时间 |
|---|---|---|---|
| **v1.0** | 2012/3 | 调度GM模型，GC STW | 百ms级别-秒级 |
| **v1.1** | 2013/5 | **调度G-P-M模型** | 同上 |
| v1.2 | 2013/12 | 实现合作式的抢占 | 同上 |
| v1.3 | 2014/6 | GC实现Mark STW, Sweep 并行.栈扩容由split stack改为复制方式的continus stack. 添加sync.Pool | 百m-几百ms级别 |
| v1.4 | 2014/12 | Runtime移除大部分C代码; 实现准确式GC. 引入写屏障, 为1.5的并发GC做准备. | 同上 |
| **v1.5** | 2015/8 | Runtime完全移除C代码, 实现了Go的自举.<br>**GC 并发标记清除, 三色标记法**; GOMAXPROCS默认为CPU核数, go tool trace引入 | 10ms级别 |
| v1.6 | 2016/2 | 1.5中一些与并发GC不协调的地方更改. 集中式的GC协调协程, 改为状态机实现 | 5ms级别 |
| v1.7 | 2016/8 | GC时由mark栈收缩改为并发, 引入dense bitmap, SSA引入 | ms级 |
| **v1.8** | 2017/2 | **hybrid write barrier**, 消除re-scanning stack, **GC进入sub ms.** defer和cgo调用开销减少一半 | sub ms(18GB堆) |
| V1.9 | 2017/8 | 保留用于debug的rescan stack代码移除，runtime.GC, debug.SetGCPercent, and debug.FreeOSMemory等触发STW GC改为并发GC | 基本同上 |
| V1.10 | 2018/2 | 不再限制最大GOMAXPROCS(Go 1.9为1024), LockOSThread的线程在运行的G结束后可以释放. | 基本同上 |
| V1.11 | 2018/8 | 连续的arena改为稀疏索引的方式 | 基本同上 |
| V1.12 | 2019/2 | Mark Termination流程优化 | Sub ms, 但几乎减少一半 |

注: GC STW时间与堆大小, 机器性能, 应用分配偏好, 对象数量均有关. 较早的版本来自网络上的数据. 1.4-1.9数据来源于twitter工程师. 这里是以较大的堆测试, 数据仅供参考. 普通应用的情况好于上述的数值.

简介及发展

**调度**

内存与GC

实践

# Goroutine

Process -> Thread(LWP, lightweight process) -> Goroutine (一种lightweight userspace thread)

不断共享, 不断减少切换成本

**Go实现有栈协程**

代表协程这种执行流的结构体

保护和恢复上下文的函数

运行队列

编译器将go关键字编译为生成一个协程结构体, 并放入运行队列

解决网络IO阻塞问题

协程级别的同步结构

调度: findrunnable

...

**注:Goroutine的形式只是协程的一种形式**

# 协程结构体和切换函数

```
type g struct {
    goid   int64  //协程的id
    status uint32 //协程状态
    stack  struct {
        lo uintptr //该协程拥有的栈低位
        hi uintptr //该协程拥有的栈高位
    }
    sched     gobuf; //切换时保存的上下文信息
    startfunc uintptr //程序地址
}

type gobuf struct {
    sp uintptr //栈指针位置
    pc uintptr //运行到的程序位置
}
```

代表执行流(协程)的结构体

```
TEXT runtime·mcall(SB), NOSPLIT, $0-8
    MOVQ    fn+0(FP), DI

    get_tls(CX)
    MOVQ    g(CX), AX   //从tls中获取g
    MOVQ    0(SP), BX   //PC
    MOVQ    BX, (g_sched+gobuf_pc)(AX) //保存PC到sched pc
    LEAQ    fn+0(FP), BX
    MOVQ    BX, (g_sched+gobuf_sp)(AX) //保存SP到sched sp
    MOVQ    AX, (g_sched+gobuf_g)(AX) //保存g到sched g
    MOVQ    BP, (g_sched+gobuf_bp)(AX)//保存BP到sched bp
```

切换时保存上下文

```
TEXT runtime·gogo(SB), NOSPLIT, $16-8
    MOVQ    buf+0(FP), BX
    MOVQ    gobuf_g(BX), DX     //DX保存了gobuf结构
    MOVQ    0(DX), CX       // make sure g != nil
    get_tls(CX)
    MOVQ    DX, g(CX)
    MOVQ    gobuf_sp(BX), SP    //恢复SP
    MOVQ    gobuf_ret(BX), AX
    MOVQ    gobuf_ctxt(BX), DX
    MOVQ    gobuf_bp(BX), BP
    MOVQ    gobuf_pc(BX), BX //恢复PC
    JMP BX //跳到对应地址
```

重新调度时恢复上下文

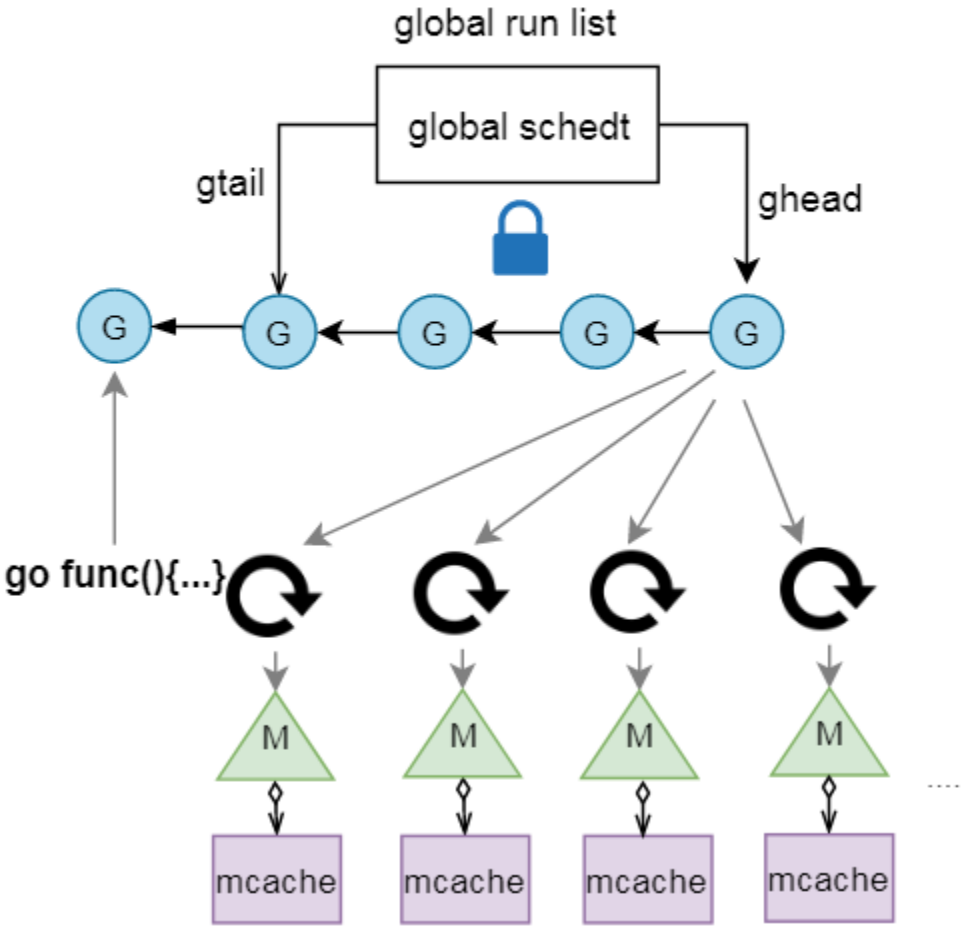
# GM模型

一开始, 实现一个简单一点的, 一个全局队列放待运行的g.

新生成G, 阻塞的G变为待运行, M寻找可运行的G等操作都在全局队列中操作, 需要加线程级别的锁.

- 调度锁问题. 单一的全局调度锁(Sched.Lock)和集中的状态, 导致伸缩性下降.
- G传递问题. 在工作线程M之间需要经常传递runnable的G, 会加大调度延迟, 并带来额外的性能损耗.
- Per-M的内存问题. 类似TCMalloc结构的内存结构, 每个M都需要memory cache和其他类型的cache(比如stack alloc), 然而实际上只有M在运行Go代码时才需要这些Per-M Cache, 阻塞在系统调用的M并不需要这些cache. 正在运行Go代码的M与进行系统调用的M的比例可能高达1:100, 这造成了很大的内存消耗.

等

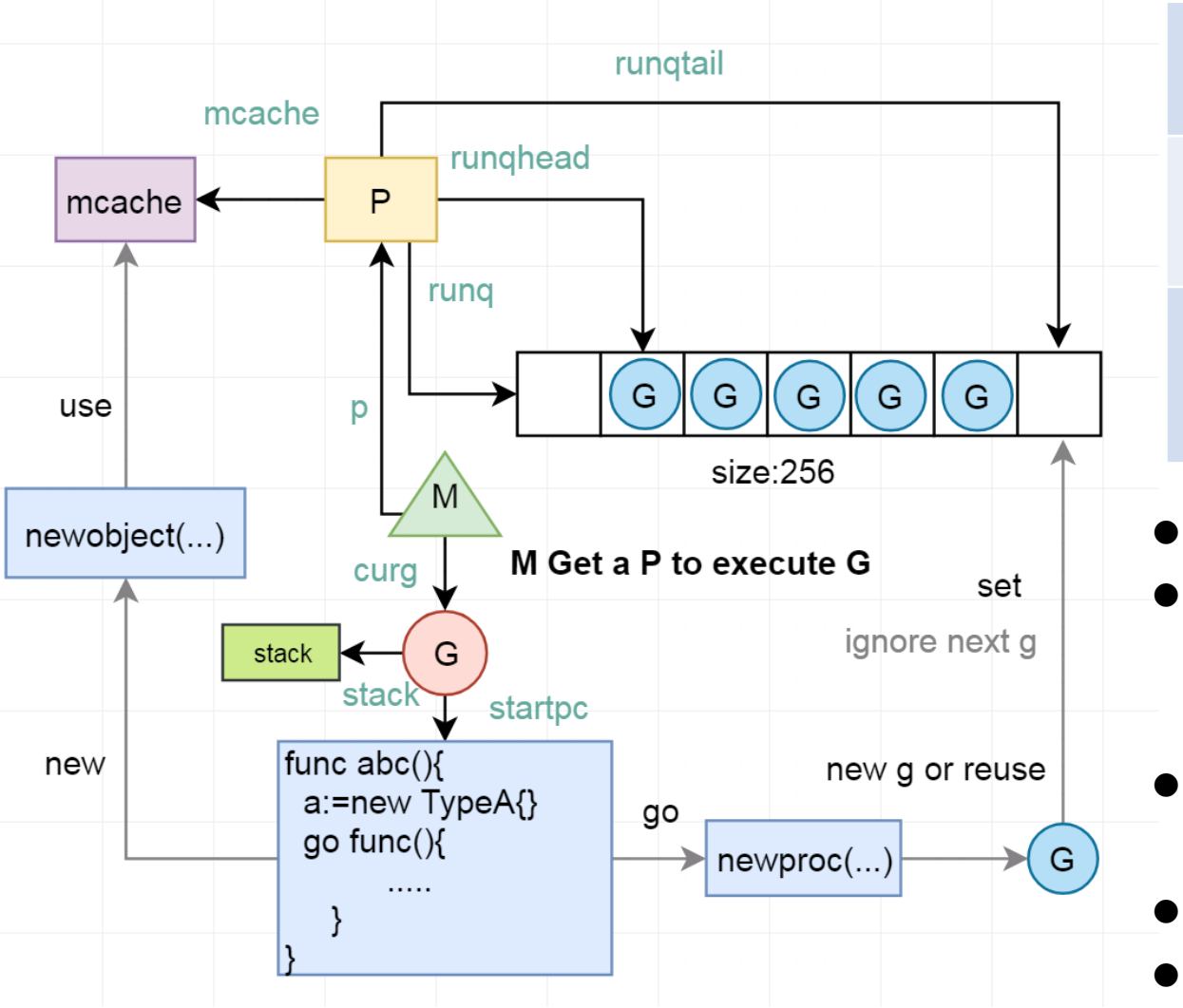



**是不是可以给运行的M加个本地队列?**

**是不是可以剥夺阻塞的M的mcache给其他M使用?**

# GPM模型

Golang 1.1中调度为GPM模型. 通过引入逻辑Processer P来解决GM模型的几个问题.




| | 数据结构 | 数量 | 意义 |
|---|---|---|---|
| G<br>Goroutine | runtime.g<br>运行的函数指针, stack, 上下文等 | 每次都go func都代表一个G, 无限制 | 代表一个用户代码执行流 |
| P<br>Processor | runtime.p<br>per-P的cache, runq和free g等 | 默认为机器核数.<br>可通过GOMAXPROCS环境变量调整. | 表示执行所需的资源 |
| M<br>Machine | runtime.m<br>对应一个由clone创建的线程 | 比P多, 一般不会多太多.<br>最大1万个 | 代表执行者, 底层线程 |

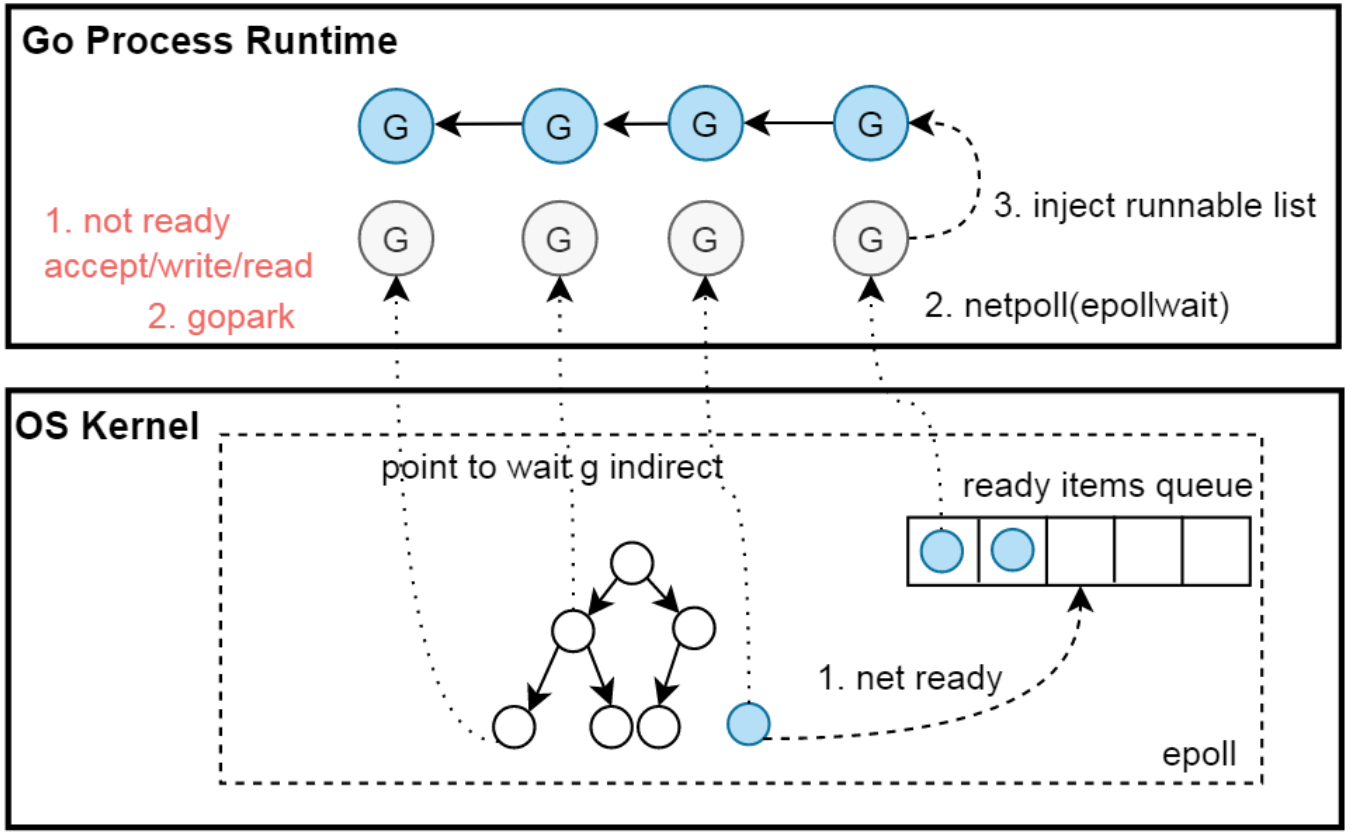
- mcache从M中移到P中.
- 不再是单独的全局runq. 每个P拥有自己的runq. 新的G放入自己的runq. 满了后再批量放入全局runq中. 优先从自己的runq获取G执行
- 实现work stealing, 当某个P的runq中没有可运行G时, 可以从全局获取, 从其他P获取
- 当G因为网络或者锁切换, 那么G和M分离, M通过调度执行新的G
- 当M因为系统调用阻塞或cgo运行一段时间后, sysmon协程会将P与M分离. 由其他的M来结合P进行调度.

# 网络

JavaScript网络操作是异步非阻塞的, 通过事件循环, 回调对应的函数.
一些状态机模式的框架, 每次网络操作都有一个新的状态.
代码执行流被打散.

用户态的协程: 结合 epoll, nonblock模式的fd操作;
网络操作未ready时的切换协程和ready后把相关协程添加到待运行队列. 网络操作达到既不阻塞线程, 又是同步执行流的效果.

1. 封装epoll, 有网络操作时会epollcreate一个epfd.
2. 所有网络fd均通过fcntl设置为NONBLOCK模式, 以边缘触发模式放入epoll节点中.
3. 对网络fd执行Accept(syscall.accept4), Read(syscall.read), Write(syscall.write)操作时, 相关操作未ready, 则系统调用会立即返回EAGAIN; 使用gopark切换该协程
4. 在不同的时机, 通过epollwait来获取ready的epollevents, 通过其中data指针可获取对应的g, 将其置为待运行状态, 添加到runq



# 未涉及的点

- G状态流转
- 具体调度流程
- 栈扩容
- 合作式抢占
- sysmon
- P状态流转
- M的spin与unspin
- LockOSThread

等

简介及发展

调度

**内存与GC**

实践

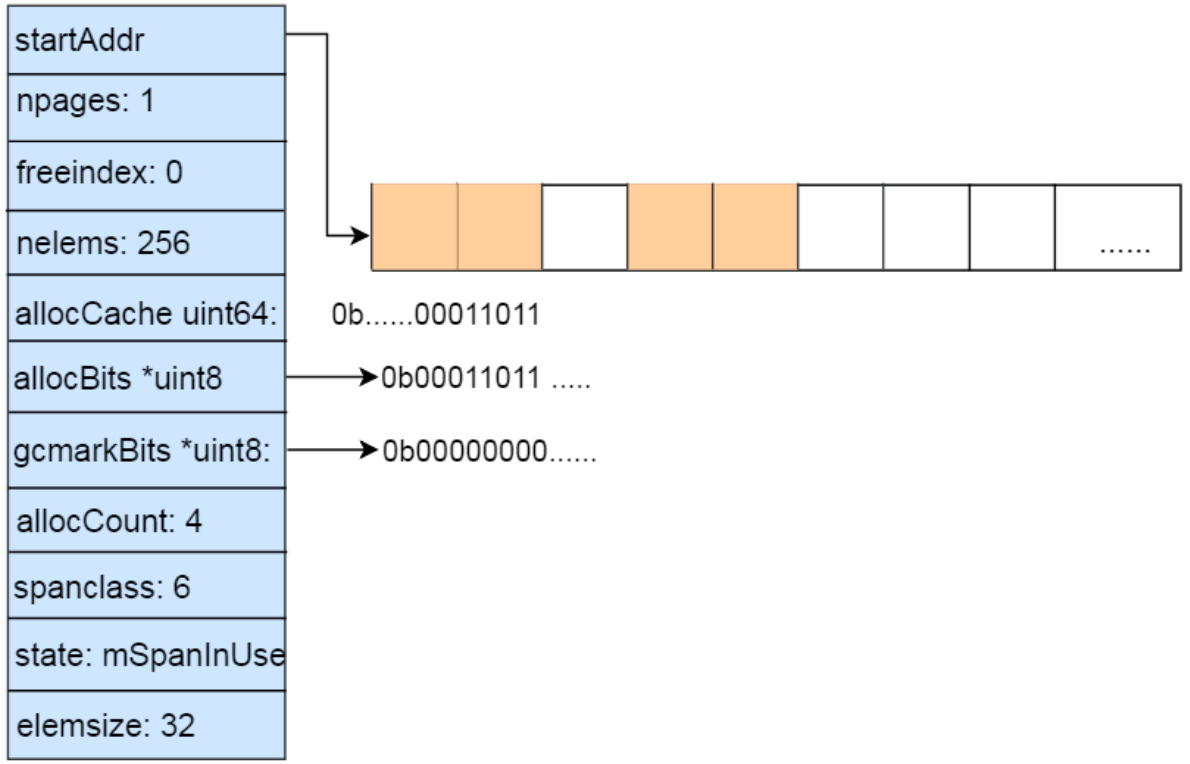
# mspan

使用span机制来减少碎片. 每个span至少分配1个page(8KB), 划分成固定大小的slot, 用于分配一定大小范围的内存需求.

runtime/sizeclasses.go

| class | bytes/obj | bytes/span | objects | tail waste | max waste |
|---|---|---|---|---|---|
| 1 | 8 | 8192 | 1024 | 0 | 87.50% |
| 2 | 16 | 8192 | 512 | 0 | 43.75% |
| 3 | 32 | 8192 | 256 | 0 | 46.88% |
| 4 | 48 | 8192 | 170 | 32 | 31.52% |
| 5 | 64 | 8192 | 128 | 0 | 23.44% |
| 6 | 80 | 8192 | 102 | 32 | 19.07% |
| 7 | 96 | 8192 | 85 | 32 | 15.95% |
| 8 | 112 | 8192 | 73 | 16 | 13.56% |
| 9 | 128 | 8192 | 64 | 0 | 11.72% |
| 10 | 144 | 8192 | 56 | 128 | 11.82% |
| 11 | 160 | 8192 | 51 | 32 | 9.73% |
| 12 | 176 | 8192 | 46 | 96 | 9.59% |
| 13 | 192 | 8192 | 42 | 128 | 9.25% |
| 14 | 208 | 8192 | 39 | 80 | 8.12% |
| 15 | 224 | 8192 | 36 | 128 | 8.15% |
| .... | | | | | |
| .... | | | | | |
| 58 | 16384 | 16384 | 1 | 0 | 12.49% |
| 59 | 18432 | 73728 | 4 | 0 | 11.11% |
| 60 | 19072 | 57344 | 3 | 128 | 3.57% |
| 61 | 20480 | 40960 | 2 | 0 | 6.87% |
| 62 | 21760 | 65536 | 3 | 256 | 6.25% |
| 63 | 24576 | 24576 | 1 | 0 | 11.45% |
| 64 | 27264 | 81920 | 3 | 128 | 10.00% |
| 65 | 28672 | 57344 | 2 | 0 | 4.91% |
| 66 | 32768 | 32768 | 1 | 0 | 12.50% |

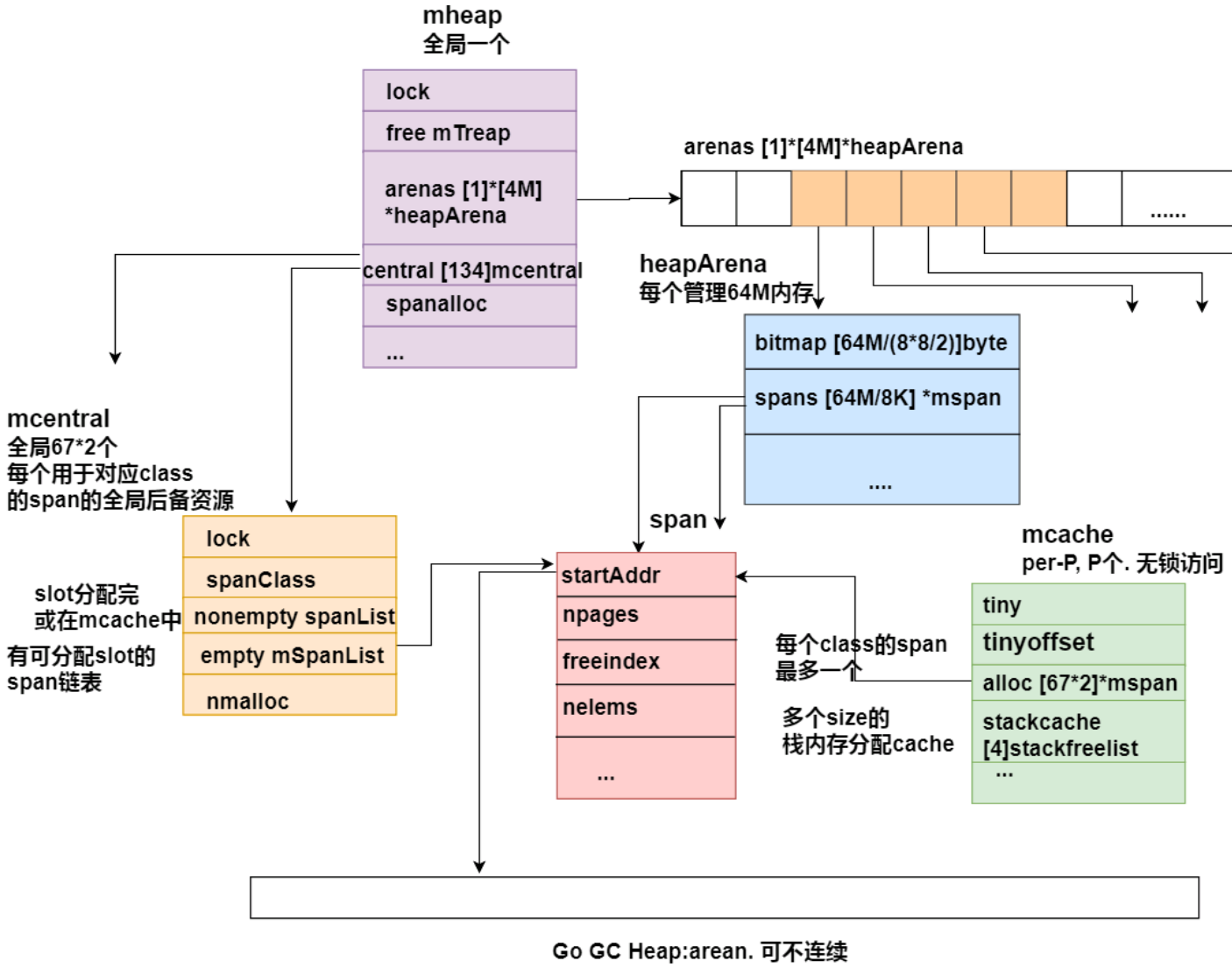
class0表示单独分配一个>32KB对象的span, 有67个size, 每个size两种, 用于分配有指针和无指针对象, 所以有67*2=134个class.




span结构体主要字段如上. 使用bitmap的方式表示每个slot是否在使用.

上图具体表示每个slot为32byte的span, 在上一次gc之后, 前8个slot使用情况. 结合freeindex和bitmap, ctz(Count Trailing Zeros指令)来找到第一个非0位进行分配.

# 全景及分配策略




多层次Cache来减少分配的冲突, 加快分配.
从无锁到粒度较低的锁, 再到全局一个锁, 或系统调用.

**分配策略**

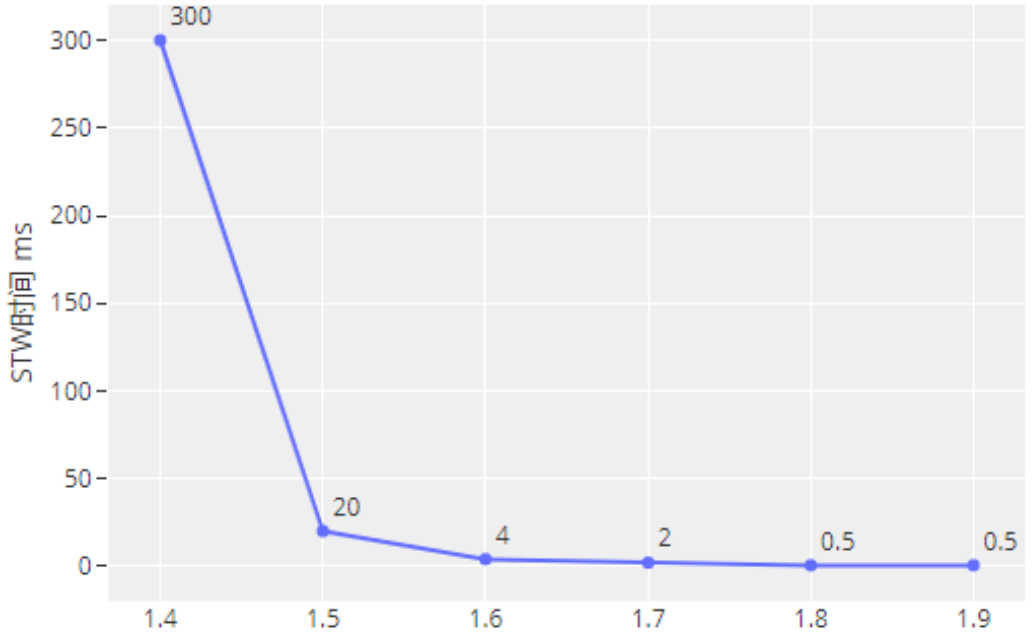
1. new, make最终调用mallocgc
2. >32KB对象, 直接从mheap中分配, 构成一个span
3. <16byte且无指针(noscan), 使用tiny分配器, 合并分配.
4. <16byte有指针或16byte-32KB, 如果mcache中有对应class的空闲mspan, 则直接从该mspan中分配一个slot.
5. (mcentral.cacheSpan) mcache没有对应的空余span, 则从对应mcentral中申请一个有空余slot的span到mcache中. 再进行分配
6. ( mcentral.grow)对应mcentral没有空余span, 则向mheap( mheap_.alloc)中申请一个span, 能sweep出span则返回. 否则看mheap的free mTreap能否分配最大于该size的连续页, 能则分配, 多的页放回 .
7. mheap的free mTreap无可用, 则调用sysAlloc(mmap)向系统申请.
8. 6, 7步中获得的内存构建成span, 返回给mcache, 分配对象.

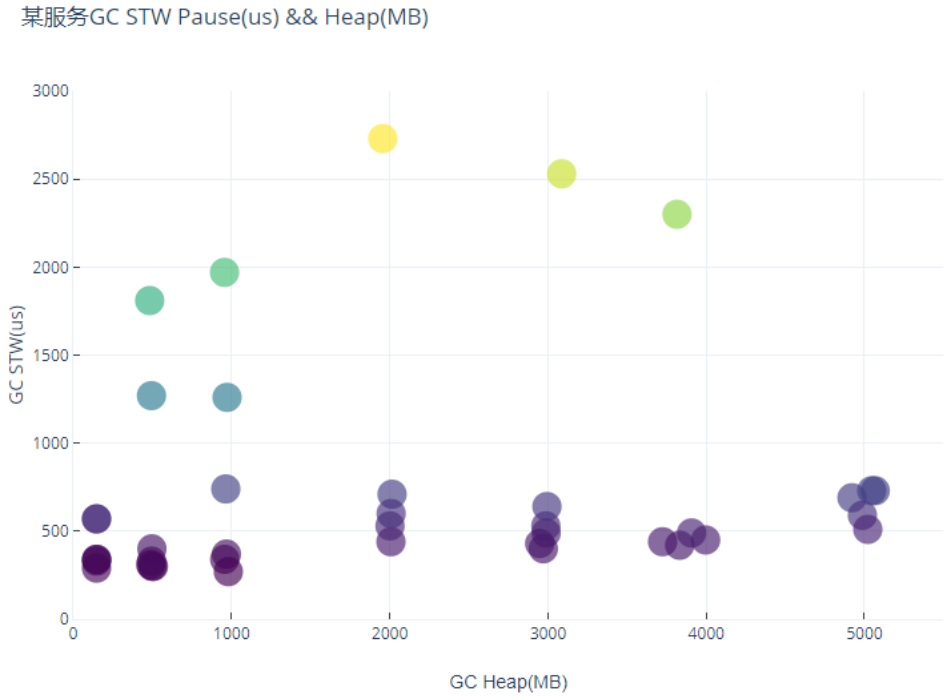
# Golang GC发展

Golang早期版本GC可能问题比较多，但每一个版本的发布都伴随着 GC 的改进.

1.5版本之后, go的GC已经能满足大部分生产环境使用要求. 1.8通过hybrid write barrier, 使得STW降到了sub ms. 下面列出一些GC方面比较重大的改动:

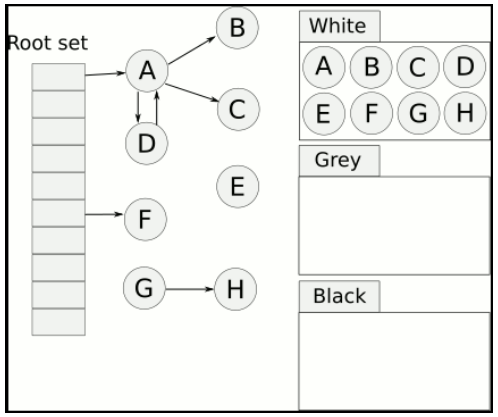
| 版本 | 发布时间 | GC | STW时间(见备注twitter数据) |
| --- | --- | --- | --- |
| v1.1 | 2013/5 | STW | 百ms-几百ms级别 |
| v1.3 | 2014/6 | Mark STW, Sweep 并行 | 百ms级别 |
| v1.5 | 2015/8 | 三色标记法, 并发标记清除 | 10ms级别 |
| v1.8 | 2017/2 | hybrid write barrier | sub ms |



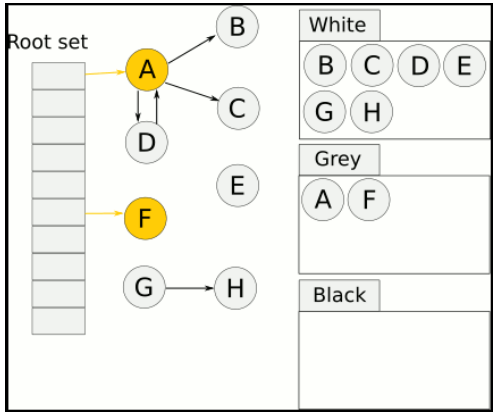



# 三色标记

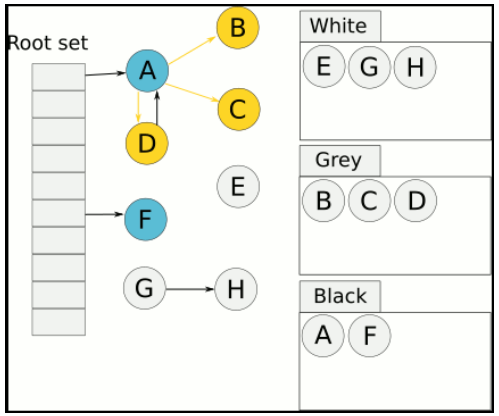
初始状态




Root标记(栈, 全局对象等)

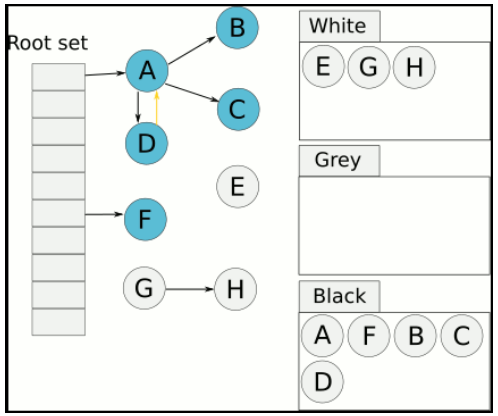



标记灰色对象引用的对象,
灰色对象变成黑色

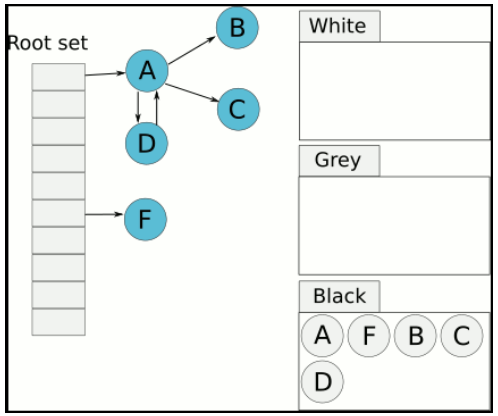



1. 有黑白灰三个集合. 初始时所有对象都是白色
2. 从Root对象开始标记, 将所有可达对象标记为灰色
3. 从灰色对象集合取出对象, 将其引用的对象标记为灰色, 放入灰色集合, 并将自己标记为黑色
4. 重复第三步, 直到灰色集合为空, 即所有可达对象都被标记
5. 标记结束后, 不可达的白色对象即为垃圾. 对内存进行迭代清扫, 回收白色对象.
6. 重置GC状态

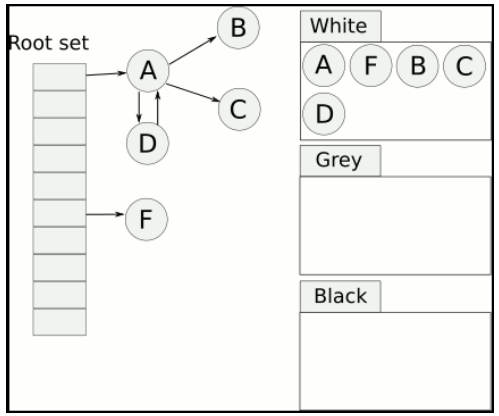
继续标记灰色对象




清扫白色对象




重置状态




**go和java不同, go的对象在内存中并没有header.**

1. 标记和程序并发, 会漏标记对象吗? 如何解决的?
2. 哪里记录了对象的三色标记状态?
3. 标记时, 拿到一个指针, 怎么知道它是哪个对象? 也许是某个对象的内部指针? 这个对象的内存哪些地方代表了它引用的对象呢?

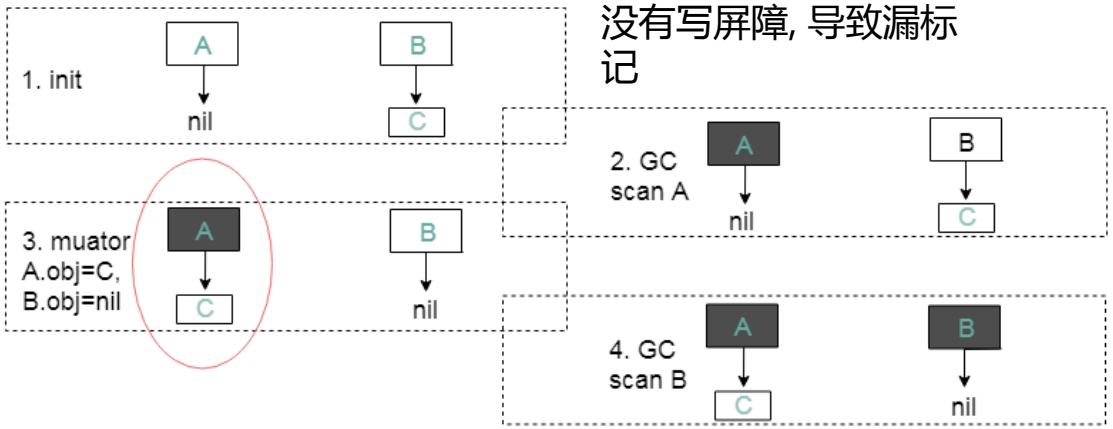
图来自https://en.wikipedia.org/wiki/Tracing_garbage_collection

# Go三色标记实现的一点细节: 写屏障

```go
var A Wb
var B Wb

type Wb struct {
    Obj *int
}

func simpleSet(c *int) {
    A.Obj = nil
    B.Obj = c
    //if GC Begin
    //scan A
    A.Obj = c
    B.Obj = nil
    //scan B
}
```




没有写屏障, 导致漏标记

三色标记需要维护(弱)不变性条件: 黑色对象不能引用无法被灰色对象可达的白色对象.
并发标记时, 如果没有做正确性保障措施, 可能会导致漏标记对象, 导致实际上可达的对象被清扫掉.

为了解决这个问题, go使用了写屏障(和内存写屏障不是同一个概念).
写屏障是在写入指针前执行的一小段代码, 用以防止并发标记时指针丢失, 这一小段代码Go是在编译时加入的.
Golang写屏障在mark和marktermination阶段处于开启状态.

左图, 第3步,A.obj=C时, 会将C进行标记, 加入写屏障buf, 最终会flush到待扫描队列, 这样就不会丢失C及C引用的对象.

# Go三色标记实现的一点细节: 写屏障

```go
var A Wb
var B Wb

type Wb struct {
    Obj *int
}

func simpleSet(c *int) {
    A.Obj = nil
    B.Obj = c
    //if GC Begin
    //scan A
    A.Obj = c
    B.Obj = nil
    //scan B
}
```

**go build -gcflags "-N -l"**
**go tool objdump -s 'main\.simpleSet' -S ./main.exe**

```
    B.Obj = c
0x44eb9e    488b442410          MOVQ 0x10(SP), AX          参数c
0x44eba3    833d565e080000      CMPL $0x0, runtime.writeBarrier(SB)  判断写屏障是否开启
0x44ebaa    7402                JE 0x44ebae                没开启则直接赋值
0x44ebac    eb52                JMP 0x44ec00               开启了则跳转到这里执行一个汇编函数, 写屏障逻辑
0x44ebae    488905f3c60600      MOVQ AX, main.B(SB)        真正执行B.Obj = c
0x44ebb5    eb00                JMP 0x44ebb7               只是执行下一条
  A.Obj = c
0x44ebb7    488b442410          MOVQ 0x10(SP), AX
    ......

0x44ec00    488d3da1c60600      LEAQ main.B(SB), DI
0x44ec07    e834a0ffff          CALL runtime.gcWriteBarrier(SB)   写屏障逻辑,在asm_amd64p32.s
0x44ec0c    eba9                JMP 0x44ebb7               执行完写屏障, 跳回去
```

栈中指针slot的操作没有写屏障.

Dijkstra写屏障是对被写入的指针进行grey操作, 不能防止指针从heap被隐藏到黑色的栈中, 需要STW重扫描栈.
Yuasa写屏障是对将被覆盖的指针进行grey操作, 不能防止指针从栈被隐藏到黑色的heap对象中, 需要在GC开始时保存栈的快照.

go 1.8写屏障混合了两者, 既不需要GC开始时保存栈快照, 也不需要STW重扫描栈, 原型如下:

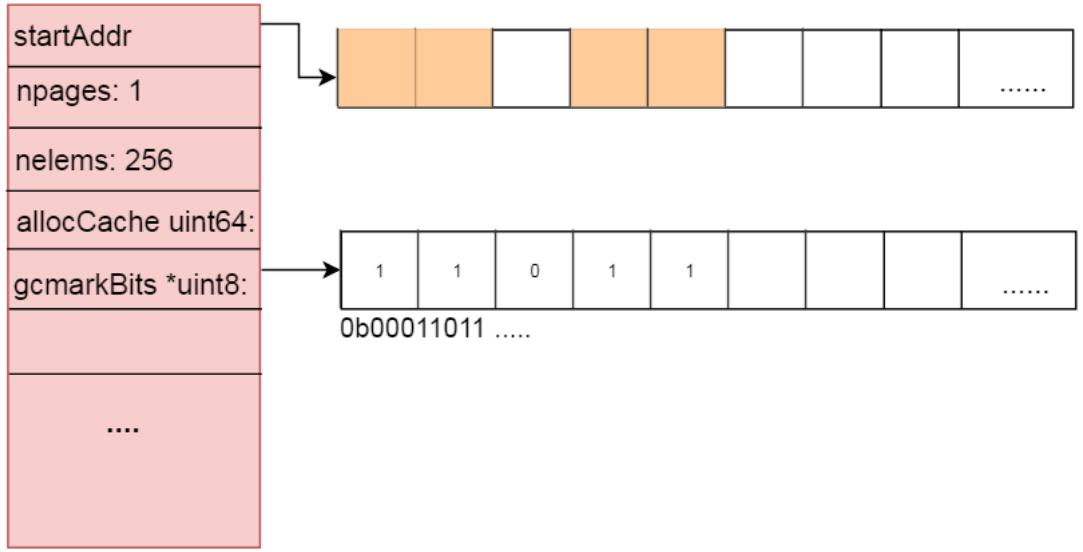
```
writePointer(slot, ptr):
    shade(*slot)
    if current stack is grey:
        shade(ptr)
    *slot = ptr
```

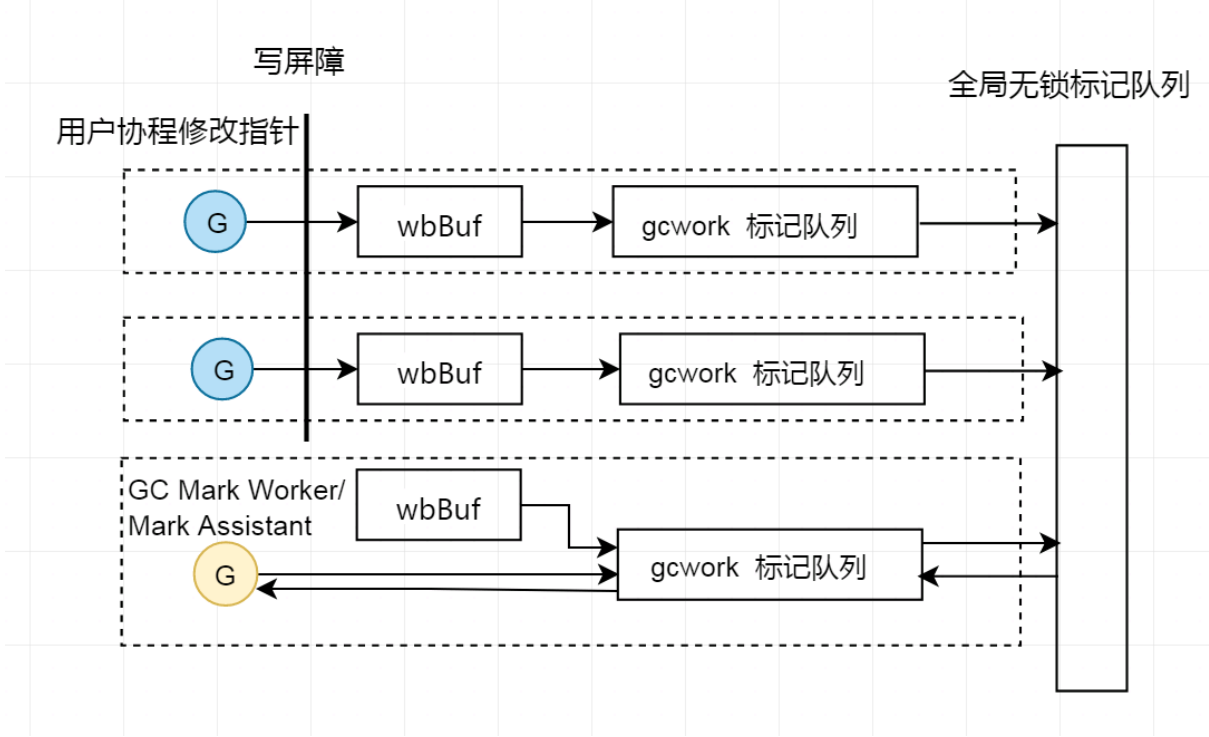
Proposal: Eliminate STW stack re-scanning
https://github.com/golang/proposal/blob/master/design/17503-eliminate-rescan.md

# Go三色标记实现的一点细节: 三色状态

**哪里记录了对象的三色标记状态?**

并没有真正的三个集合来分别装三色对象.
前面分析内存的时候, 介绍了go的对象是分配在span中, span里还有一个字段是gcmarkBits, mark阶段里面每个bit代表一个slot已被标记.
白色对象该bit为0, 灰色或黑色为1. (runtime.markBits)
每个p中都有wbBuf和gcw gcWork, 以及全局的workbuf标记队列, 实现生产者-消费者模型, 在这些队列中的指针为灰色对象, 表示已标记, 待扫描.
从队列中出来并把其引用对象入队的为黑色对象, 表示已标记, 已扫描. (runtime.scanobject).

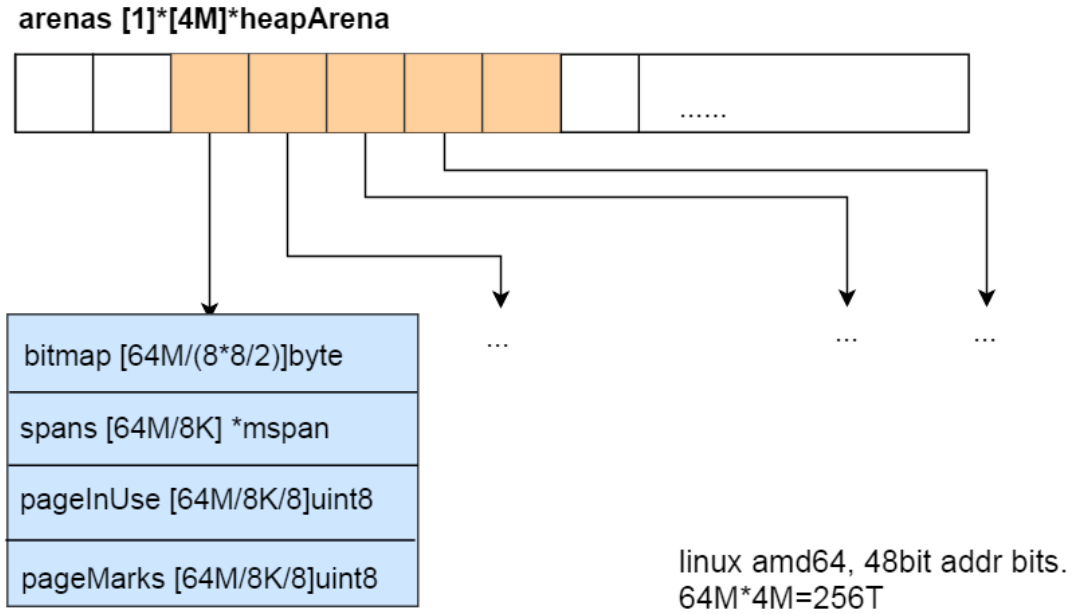


# Go三色标记实现的一点细节: 扫描与元信息

**2. 标记时拿到一个指针p1, 如何知道哪里是其引用的对象?**

回到前面所提到的内存结构图. go的gc heap通过右图的arenas进行划分, 每个heapArena管理了64M内存. heapArena存储着pointer, span, bitmap的索引关系.



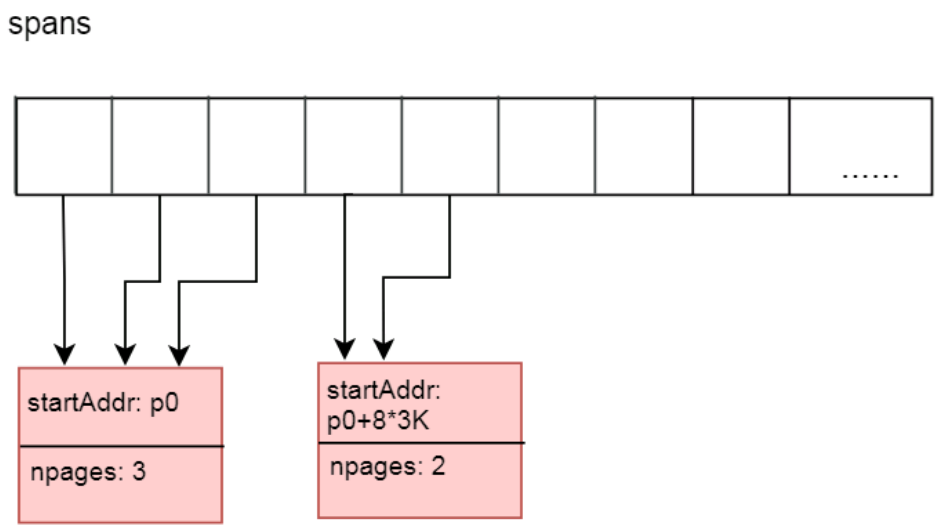

**p1指向的对象所属heapArena**: arenas[0][p+constbase/64M]

**找到对象所属span:** p1%64M/8K就知道了该对象在该heapArena中的页index, 通过spans[index]即可找到其所属的span(runtime.spanOf)

**对象首地址:** 找到对象所属的span, 根据span的elemsize和span的startAddr, 即可知道其为该span中第几个对象以及其地址(runtime.findObject)

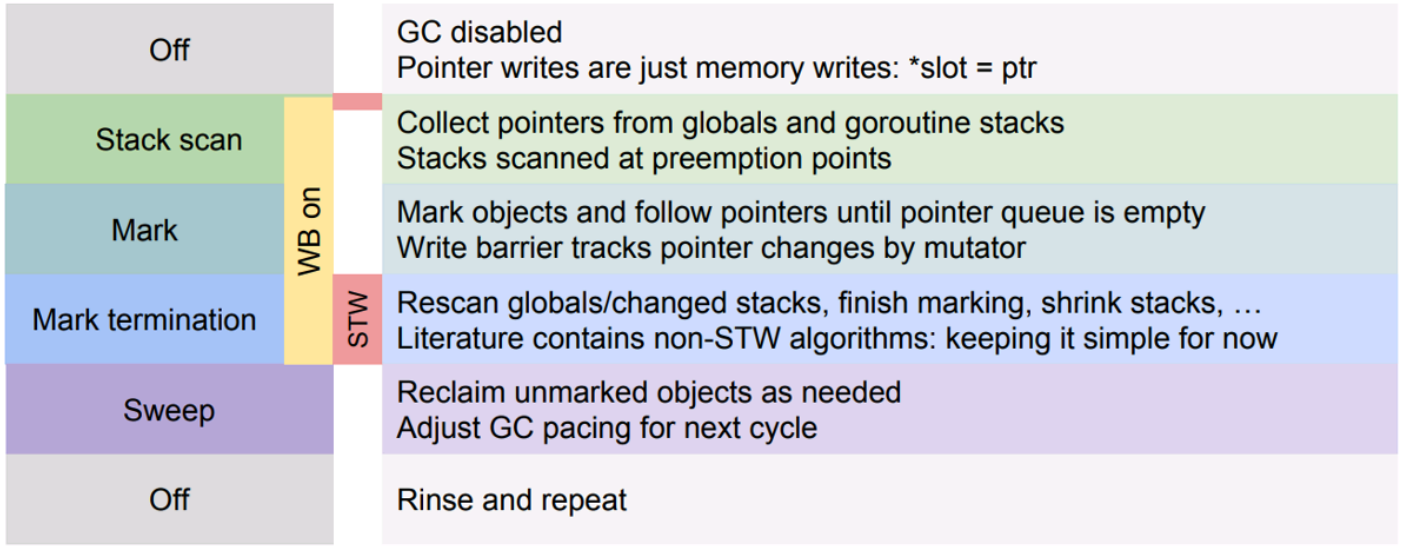
**对象的gcmarkBits:** 知道了obj在span中属于第几个对象, 即可知道如何设置其gcmarkBits.

**对象引用的对象:** bitmap每两个bit分别表示某8个字节是否是指针, 以及该字节所属对象后续字节是否还包含指针, 以此知道其引用的对象和减少扫描工作量. 这些bit是分配对象时, 根据对象type信息设置的.

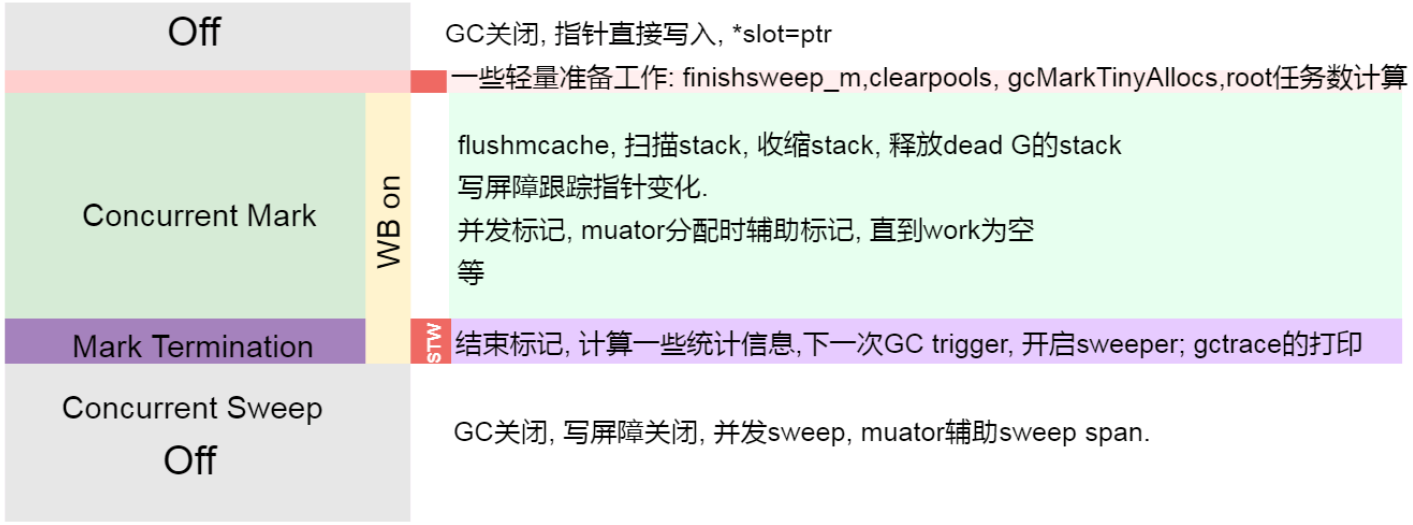


# Golang GC流程

Go 1.5 Release : 2015/8 图来自 https://talks.golang.org/2015/go-gc.pdf




Go 1.12 Release: 2019/2 根据源码绘制




**GC Trigger**

gcTriggerHeap: 分配内存时，当前已分配内存与上一次GC结束时存活对象的内存达到某个比例时就触发GC.
gcTriggerTime: sysmon检测2min内是否运行过GC, 没运行过则执行GC.
gcTriggerAlways: runtime.GC()强制触发GC.

当然1.5的Mark和Sweep都是Concurrent的, 只不过下图我特地注明了下.

**改进点:**

Go 1.5是Go转为并发三色标记清除法的版本. 大部分情况下能够将STW控制在10ms以下.

1.5中GC流程及状态的转换由一个协程集中式控制, 容易导致调度延迟.
1.6中采用了分布式检测, 协程也可以进行检测并状态转换.

1.5中栈收缩在Mark Termination(STW)阶段, 如果协程很多, 会导致STW时间暴增.
1.7将栈收缩移入并发Mark阶段.

1.5只采用了Dijkstra写屏障, 在Mark Termination(STW)阶段需要重新扫描栈, 这个是STW最大的来源.
1.8采用了混合写屏障, 消除了重新扫描栈, STW进入sub ms.
.....

1.12对Mark Terminiation阶段进行了优化

# 未涉及的点

- 栈分配
- fixalloc, tinyalloc
- 逃逸分析
- 内存归还
- 清扫与辅助清扫
- go gc mark任务分配
- 辅助标记
- revise
- mark termination流程
- GC Pacer, trigger计算, goal计算

....

简介及发展

调度

内存与GC

**实践**

# 观察一下调度

通过godoc来开启一个go进程, 用ab来调用, 加上debug trace观察其状态

GOMAXPROCS=8 GODEBUG=schedtrace=500  godoc -http=:6060 (加上scheddetail=1观察更详细信息)

ab -c 1000 -n 100000 'http://localhost:6060/'

```
SCHED 6978ms: gomaxprocs=8 idleprocs=0 threads=15 spinningthreads=0 idlethreads=3 runqueue=30 [4 25 3 59 0 3 2 17]
SCHED 7280ms: gomaxprocs=8 idleprocs=0 threads=15 spinningthreads=0 idlethreads=3 runqueue=0 [0 0 15 0 24 12 45 0]
SCHED 7584ms: gomaxprocs=8 idleprocs=0 threads=15 spinningthreads=1 idlethreads=3 runqueue=1 [0 0 0 0 5 56 14 0]
SCHED 7887ms: gomaxprocs=8 idleprocs=0 threads=16 spinningthreads=0 idlethreads=3 runqueue=35 [13 6 12 10 7 8 6 6]
SCHED 8189ms: gomaxprocs=8 idleprocs=0 threads=16 spinningthreads=0 idlethreads=3 runqueue=0 [65 0 12 1 3 11 0 2]
SCHED 8492ms: gomaxprocs=8 idleprocs=0 threads=16 spinningthreads=0 idlethreads=3 runqueue=3 [0 0 1 2 43 55 6 7]
SCHED 8800ms: gomaxprocs=8 idleprocs=5 threads=16 spinningthreads=1 idlethreads=8 runqueue=105 [3 6 0 0 0 0 0 0]
SCHED 9102ms: gomaxprocs=8 idleprocs=0 threads=16 spinningthreads=0 idlethreads=4 runqueue=44 [14 9 6 6 1 9 6 6]
SCHED 9406ms: gomaxprocs=8 idleprocs=1 threads=16 spinningthreads=1 idlethreads=5 runqueue=17 [3 4 6 2 4 3 0 0]
SCHED 9708ms: gomaxprocs=8 idleprocs=4 threads=16 spinningthreads=1 idlethreads=7 runqueue=69 [13 11 0 0 12 0 0 0]
```

# 观察一下GC

这是一个服务通过调用debug.SetGCPercent设置GOGC, 分别是100, 550, 1650时的表现.

GOGC越大, GC频次越低, 但是触发GC的堆内存也越大. 其中具体含义可线下讨论.

```
GOGC=100
gc 8913 @2163.341s 1%: 0.13+14+0.20 ms clock, 1.1+21/22/0+1.6 ms cpu, 147->149->75 MB, 151 MB goal, 8 P
gc 8914 @2163.488s 1%: 0.39+12+0.18 ms clock, 3.1+6.3/20/0+1.5 ms cpu, 147->150->75 MB, 151 MB goal, 8 P
gc 8915 @2163.631s 1%: 0.12+7.2+0.17 ms clock, 0.97+14/14/0+1.4 ms cpu, 147->149->76 MB, 151 MB goal, 8 P
gc 8916 @2163.805s 1%: 0.074+9.9+0.20 ms clock, 0.59+5.7/18/0+1.6 ms cpu, 148->150->75 MB, 152 MB goal, 8 P


GOGC=550
gc 9651 @2366.838s 1%: 0.10+13+0.20 ms clock, 0.82+8.7/23/0+1.6 ms cpu, 507->510->79 MB, 529 MB goal, 8 P
gc 9652 @2367.977s 1%: 1.1+9.0+0.17 ms clock, 8.8+5.4/16/0.65+1.3 ms cpu, 494->496->77 MB, 516 MB goal, 8 P
gc 9653 @2369.093s 1%: 0.21+15+1.6 ms clock, 1.7+12/25/0+13 ms cpu, 483->488->80 MB, 505 MB goal, 8 P
gc 9654 @2370.181s 1%: 0.12+10+0.20 ms clock, 1.0+11/18/0+1.6 ms cpu, 498->501->79 MB, 520 MB goal, 8 P

GOGC=1650
gc 11581 @4300.415s 1%: 0.17+11+0.54 ms clock, 1.3+11/20/0+4.3 ms cpu, 2015->2018->118 MB, 2115 MB goal, 8 P
gc 11582 @4306.141s 1%: 0.29+15+0.32 ms clock, 2.3+12/29/0+2.6 ms cpu, 1983->1987->120 MB, 2081 MB goal, 8 P
gc 11583 @4311.562s 1%: 0.26+18+0.27 ms clock, 2.1+13/30/0+2.1 ms cpu, 2002->2007->120 MB, 2101 MB goal, 8 P
gc 11584 @4316.900s 1%: 0.25+11+0.35 ms clock, 2.0+20/20/0+2.8 ms cpu, 2009->2012->118 MB, 2109 MB goal, 8 P
```



**premature optimization is the root of all evil.**

# 一些优化(写业务时用不到就算了)

**GODEBUG, GOMAXPROCS, GOGC**

**涉及文件, CGO较多的程序, 可以将P增大几个.**
runtime.GOMAXPROCS(GOMAXPROCS(0)+1)

**协程池的重要性远没有Java, CPP中线程池那么重要.**
协程的生成不涉及系统调用, 需要的栈资源也很少. 同时P和全局都做了dead G的缓存. 协程池实现的不好, 反而因为协程池里的一把锁影响了扩展性. 至于并发控制, 保护其他资源, 可以选用其他方式.
**什么时候需要协程池?**
主要还是隔离减少栈扩容和缩容. 有些场景下栈扩容和缩容消耗CPU(可结合pprof查看morestack)的确比较多. 比如长连接, 大量维持连接的协程可以不用扩容栈, 复杂任务交给任务协程处理, 此类协程的数量比较少.

**GOGC=200或更多,:** GC Pacer会根据GC情况和GOGC参数来计算gc trigger, 增大GOGC, 可降低GC频率, 注意, 会增加触发GC的堆大小.

**sync.Pool:** 对于频繁分配的对象, 可以使用sync.Pool, 减少分配频次, 进而降低GC频率 (1.13对sync.Pool进行了优化)

**全局缓存对象有大量的key的情况, value少用指针**
GC并发Mark需要mark存活的对象, 如果value里指针多, 导致mark消耗的CPU很大, 使用一个struct内嵌数据消除指针.

**一点点拷贝胜过传指针:** 对象在栈上分配, 减少GC频率.
**[]byte和string的magic: 慎用, 仅用在不会修改的地方**
**slice和map的容量初始化: 减少不断加元素时的扩容**
**json-iterator替换encoding/json等等**

# 框架或模板集成gops及默认开启pprof

往往有问题才想起没引入pprof，无法查看stack，又需要保留现场.

**服务模板代码默认引入一个库开启pprof，集成到服务列表页.**
有问题，点一点，通过一个agent，直接获取idc机器上服务的pprof图.

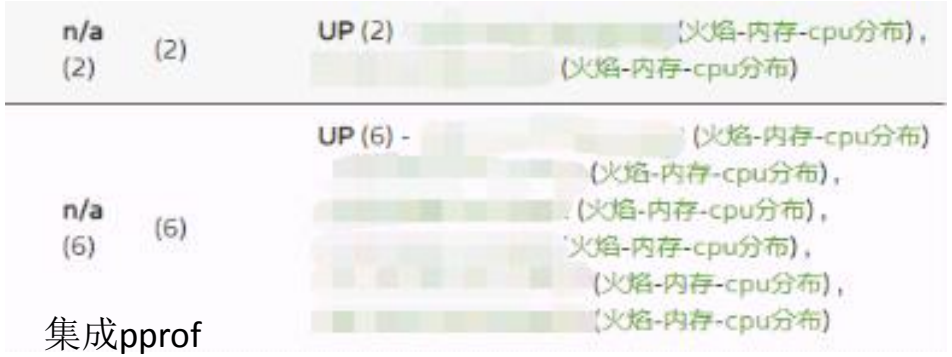
**服务模板默认引入一个库封装gops**
gops:=golang版( jps + jstack + jstat + jinfo )

https://github.com/google/gops

gops和pprof占用的端口根据服务端口+1，+2.

```
$ gops stack (<pid>|<addr>)
gops stack 85709
goroutine 8 [running]:
runtime/pprof.writeGoroutineStacks(0x13c7bc0, 0xc42000e008, 0xc420ec8520, 0xc420ec8520)
        /Users/jbd/go/src/runtime/pprof/pprof.go:603 +0x79
runtime/pprof.writeGoroutine(0x13c7bc0, 0xc42000e008, 0x2, 0xc428f1c048, 0xc420ec8608)
        /Users/jbd/go/src/runtime/pprof/pprof.go:592 +0x44
runtime/pprof.(*Profile).WriteTo(0x13eeda0, 0x13c7bc0, 0xc42000e008, 0x2, 0xc42000e008, 0x0)
        /Users/jbd/go/src/runtime/pprof/pprof.go:302 +0x3b5
github.com/google/gops/agent.handle(0x13cd560, 0xc42000e008, 0xc420186000, 0x1, 0x1, 0x0, 0x0)
        /Users/jbd/src/github.com/google/gops/agent/agent.go:150 +0x1b3
github.com/google/gops/agent.listen()
        /Users/jbd/src/github.com/google/gops/agent/agent.go:113 +0x2b2
created by github.com/google/gops/agent.Listen
        /Users/jbd/src/github.com/google/gops/agent/agent.go:94 +0x480
```

此图来自官方git
查看go协程栈




集成pprof

```
$ gops memstats <pid>
alloc: 22.96MB (24078368 bytes)
total-alloc: 84.27GB (90483014360 bytes)
sys: 70.63MB (74055928 bytes)
mallocs: 1628424898
frees: 1628111117
heap-sys: 62.59MB (65634304 bytes)
heap-idle: 35.53MB (37257216 bytes)
heap-in-use: 27.06MB (28377088 bytes)
heap-released: 32.55MB (34127872 bytes)
heap-objects: 313781
stack-in-use: 1.41MB (1474560 bytes)
stack-sys: 1.41MB (1474560 bytes)
stack-mspan-inuse: 372.80KB (381744 bytes)
stack-mspan-sys: 448.00KB (458752 bytes)
stack-mcache-inuse: 6.78KB (6944 bytes)
stack-mcache-sys: 16.00KB (16384 bytes)
other-sys: 773.62KB (792188 bytes)
gc-sys: 2.50MB (2619392 bytes)
next-gc: when heap-alloc >= 27.80MB (29150848 bytes)
last-gc: 2019-08-10 15:01:05.05691364 +0800 CST
gc-pause-total: 639.073511ms
gc-pause: 17171
num-gc: 11396
enable-gc: true
```

查看runtime内存信息

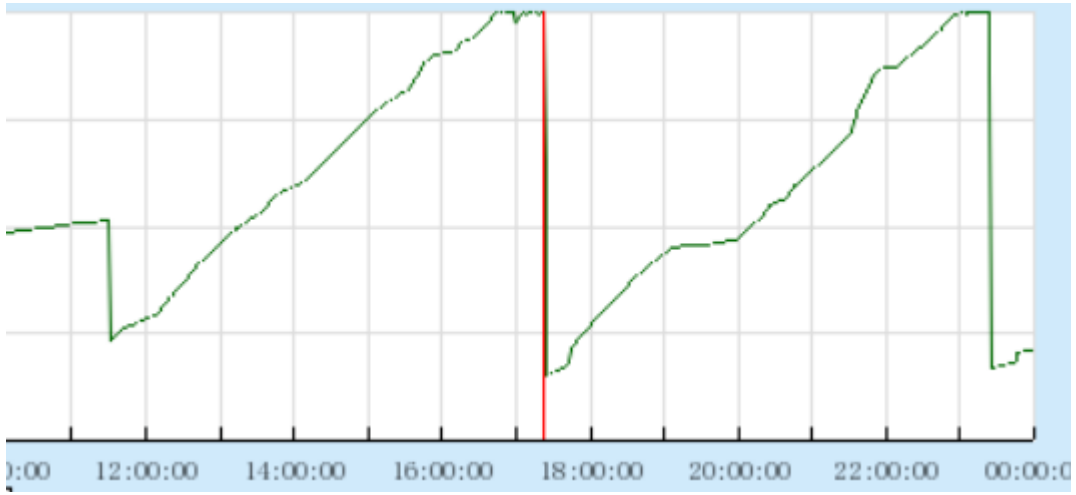
# 问题排查的一点思路

和Runtime相关的几类问题

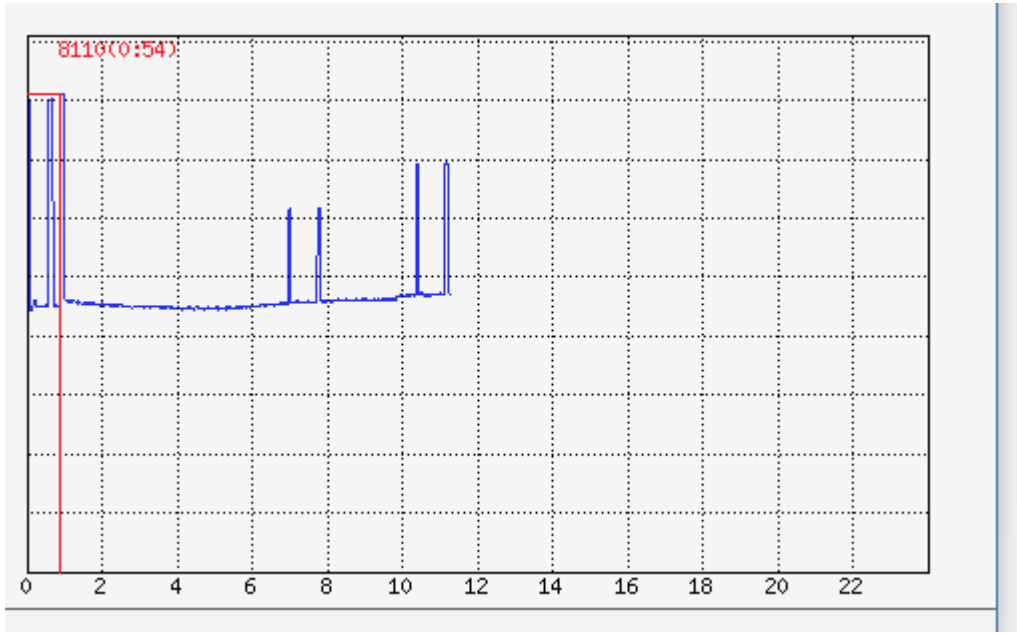
1. 内存慢慢增长OOM: 结合memory inuse_space的pprof和list, 加上源码流程即可定位出. 一直把新对象放到全局对象或者长生命周期对象中. 比如长连接, 连接池应用或者忘记close http resp body, sql Stmt等.

2. 内存突增OOM:  如果多次分配才OOM, 可使用方法1排查. 对于一次就OOM的, 比较难抓, 可结合go无法分配内存时throw输出的协程栈排查.
比如没有校验参数, 调用者填错或恶意, 使用传过来的length来进行make([]byte, length)用于编解码

3. 性能问题: 结合火焰图, 查看影响性能的热点部分, 进行优化: GC频繁, 编解码效率低等.




缓慢增长至OOM



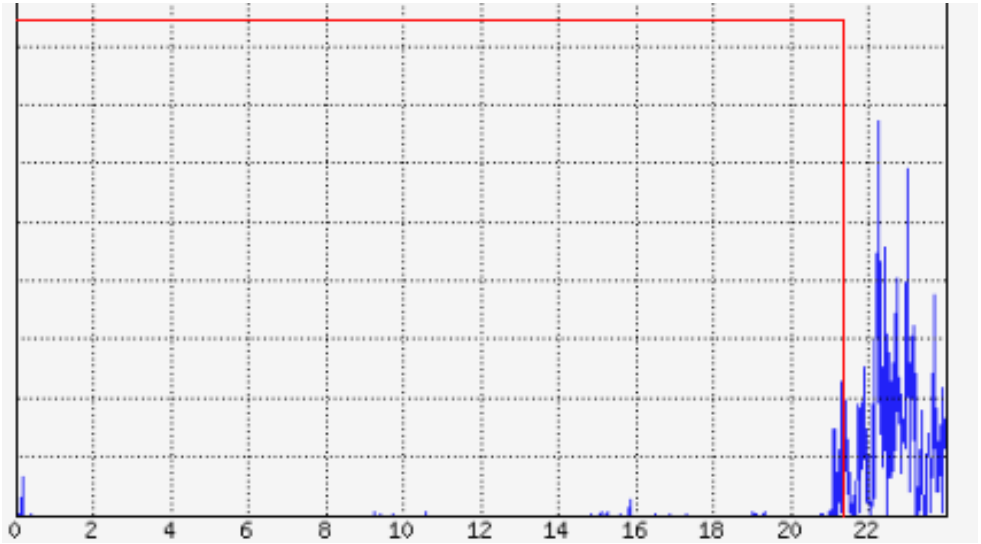

突然的OOM

# 一次有趣的排查

一同学找我说自己的服务突然超时很多, 频繁出现页面拉不出数据, 排查了很久, 实在找不出来问题.

1. 查看负载, 内存, 依赖的服务也均正常
2. 框架会捕捉panic, 目前也没有panic发生过
3. cpu的pprof也比较正常
4. 日志打的有点多, 但日志是由业务协程通过channel发给单独一个协程异步打印, 没有channel满的情况.
5. 服务开启了GODEBUG=gctrace=1, 标准输出重定向到一个文件, tail看GC的stw没什么问题, 监控上的GC数据也正常.

看监控发现个很神奇的地方: 内存cache获取>800ms很多. 但是缓存的数据也不多.




cache Get耗时>800ms

# 一次有趣的排查

关键词: 日志多, gctrace, 重定向

1. Golang gc最后一步gcMarkTermination会判断是否需要打印gctrace信息
此时还是处于STW中
2. 如果是输出console, 纯内存操作, 没影响
3. 早期的服务模板做了重定向操作到文件, 而这个同学的服务现网日志很多, 导致阻塞了gc完成.

GC的STW统计时间反映不出来这个问题

```
    // Print gctrace before dropping worldsema. As soon as we drop
    // worldsema another cycle could start and smash the stats
    // we're trying to print.
    if debug.gctrace > 0 {
        util := int(memstats.gc_cpu_fraction * 100)

        var sbuf [24]byte
        printlock()
        print( args...: "gc ", memstats.numgc,
            " @", string(itoaDiv(sbuf[:],
uint64(work.tSweepTerm-runtimeInitTime)/1e6, dec: 3)), "s ",
            util, "%: ")
        prev := work.tSweepTerm
        for i, ns := range []int64{work.tMark, work.tMarkTerm, work.tEnd} {
            if i != 0 {
                print( args...: "+")
            }
            print(string(fmtNSAsMS(sbuf[:], uint64(ns-prev))))
            prev = ns
        }
        print( args...: " ms clock, ")
        for i, ns := range []int64{sweepTermCpu, gcController.assistTime,
gcController.dedicatedMarkTime + gcController.fractionalMarkTime,
gcController.idleMarkTime, markTermCpu} {
            if i == 2 || i == 3 {
```

# Runtime的一点个人总结

| 思想 | 作用 | 示例 |
| --- | --- | --- |
| 并行 | 减少操作的wall time和阻塞 | stw mark -> concurrent mark,<br>stw stack scan, shrink-> 并发mark阶段的逐个g 等 |
| 纵向多层次 | 尽量减少锁竞争和冲突.<br>Per-P无锁 -> 粒度范围比较小的锁->最后才全局和系统调用 | 调度findrunnable，内存分配mallocgc, stack分配等 |
| 横向多个class | 找到最适配的, 减少内存浪费和碎片 | tinyalloc, 内存分配span机制, 多个class的stack分配等 |
| 缓存 | 减少重新申请 | sync.Pool, per-P的mcache, deadg的free list等, 延缓释放归还给mheap的pages |
| 缓冲 | 放入队列, 减少阻塞, 操作异步化 | 写屏障的wbBuf, GC标记队列 |
| 均衡 | 负载均衡, 不会因为work太多的而成为瓶颈 | 调度时从全局runq获取, 从其他P进行work stealing; GC标记工作的本地和全局之间的flush和get. |

# QUESTION?

分享小调查

NOW直播招人

yifhao@tencent.com

# THANKS