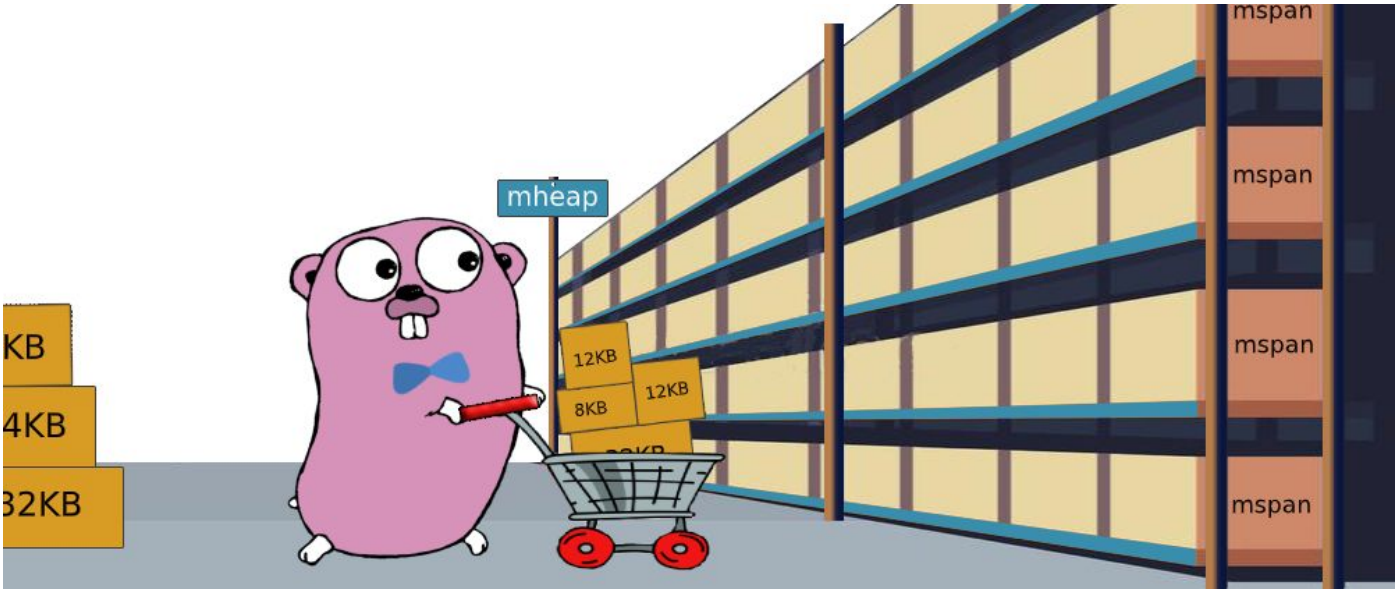
Tony Bai

一个程序员的心路历程

可视化Go内存管理

- 三月 10, 2020
- 8 条评论

本文翻译自《Visualizing memory management in Golang》。



“内存管理”系列的一部分

在这个由多部分组成的系列文章中，我旨在揭示内存管理背后的概念，并对某些现代编程语言的内存管理机制做更深入的探究。我希望该系列文章可以使您对这些语言在内存管理方面正在发生的事情能有所了解。

在本章中，我们将研究Go编程语言（Golang）的内存管理。和C/C++、Rust等一样，Go是一种静态类型的编译型语言。因此，Go不需要VM，Go应用程序二进制文件中嵌入了一个小型运行时(Go runtime)，可以处理诸如垃圾收集(GC)，调度和并发之类的语言功能。

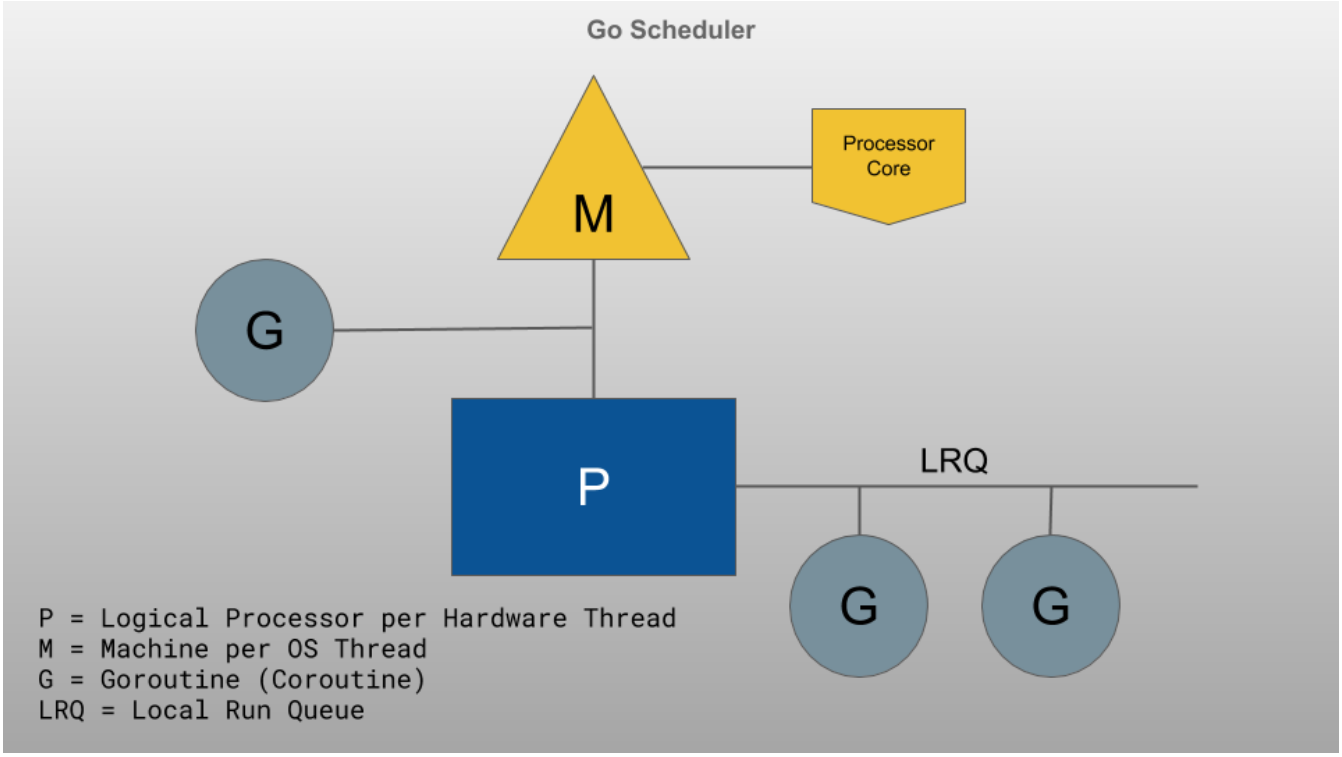
如果您还没有阅读本系列的第一部分，请先阅读它，因为在那篇文章中我解释了栈(stack)和堆(heap)内存之间的区别，这对于理解本文很有用。

> 这篇文章基于Go 1.13的默认官方实现，有些概念细节可能会在Go的未来版本中发生变化

Go内部内存结构

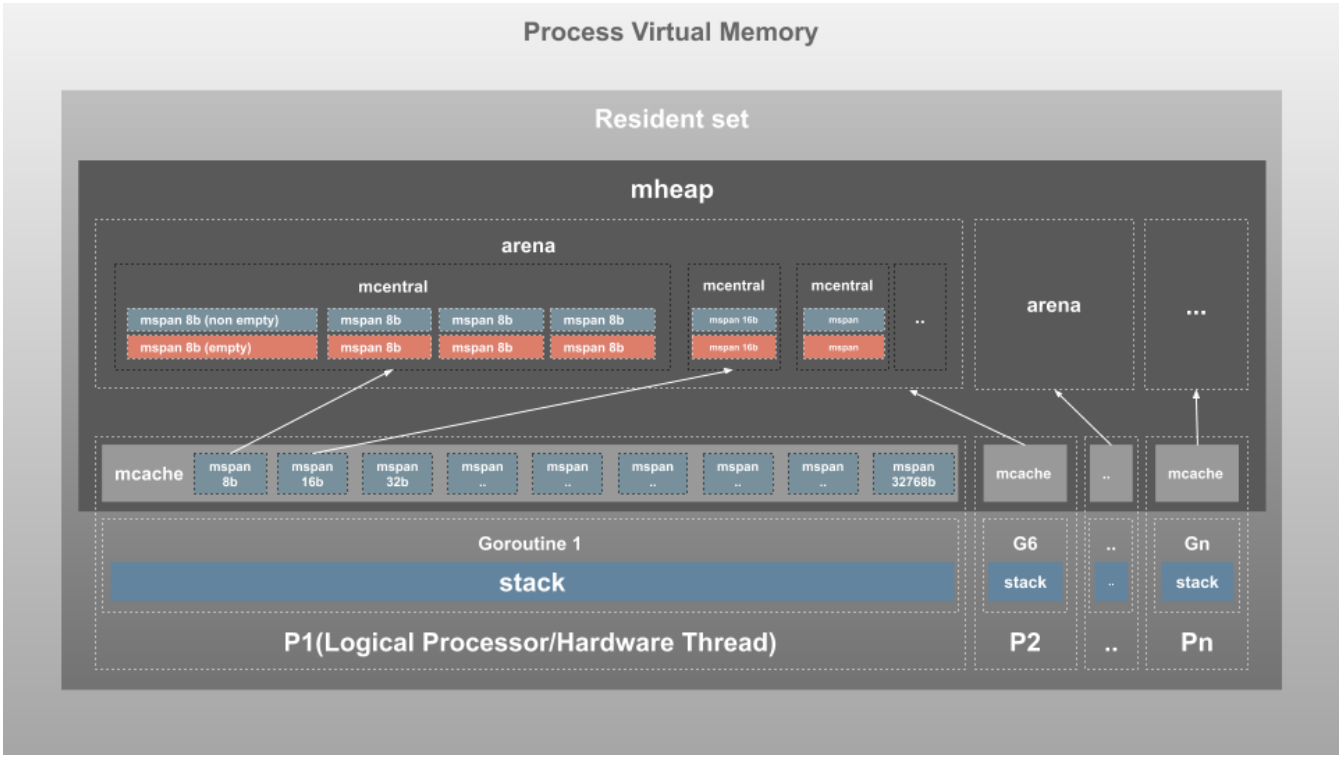
首先，让我们看看Go内部的内存结构是什么样子的。

> Go运行时将Goroutines（G）调度到逻辑处理器（P）上执行。每个P都有一台逻辑机器（M）。在这篇文章中，我们将使用P、M和G。如果您不熟悉Go调度程序，请先阅读《Go调度程序：Ms，Ps和Gs》。



Goroutine调度原理

每个Go程序进程都由操作系统（OS）分配了一些虚拟内存，这是该进程可以访问的全部内存。在这个虚拟内存中实际正在使用的内存称为Resident Set（驻留内存)。该空间由内部内存结构管理，如下所示：



Go内部内存结构原理图

这是一个简化的视图，基于Go使用的内部对象。实际上，Go将内存划分和分组为页(page)，就像这篇文章描述的那样。

这与我们在前几章中看到的JVM和V8的内存结构完全不同。如您所见，这里没有分代内存。这样做的主要原因是TCMalloc（线程缓存Malloc），Go自己的内存分配器正是基于该模型实现的。

让我们看看Go独特的内存构造是什么样子的：

页堆page heap（mheap）

这里是Go存储动态数据（在编译时无法计算大小的任何数据）的地方。它是最大的内存块，也是进行垃圾收集（GC）的地方。

驻留内存(resident set)被划分为每个大小为8KB的页，并由一个全局**mheap对象**管理。

> 大对象（大小> 32kb的对象）直接从mheap分配。这些大对象申请请求是以获取中央锁(central lock)为代价的，因此在任何给定时间点只能满足一个P的请求。

mheap通过将页归类为不同结构进行管理的：

- mspan：mspan是mheap中管理的内存页的最基本结构。这是一个双向链接列表，其中包含起始页面的地址，span size class和span中的页面数量。像TCMalloc一样，Go将内存页按大小分为67个不同类别，大小从8字节到32KB，如下图所示

mspan结构

每个span存在两个，一个span用于带指针的对象（scan class），一个用于无指针的对象（noscan class）。这在GC期间有帮助，因为noscan类查找活动对象时无需遍历span。

- mcentral：mcentral将相同大小级别的span归类在一起。每个mcentral包含两个mspanList：
  - empty：双向span链表，包括没有空闲对象的span或缓存mcache中的span。当此处的span被释放时，它将被移至non-empty span链表。
  - non-empty：有空闲对象的span双向链表。当从mcentral请求新的span，mcentral将从该链表中获取span并将其移入empty span链表。

如果mcentral没有可用的span，它将向mheap请求新页。

- arena：堆在已分配的虚拟内存中根据需要增长和缩小。当需要更多内存时，mheap从虚拟内存中以每块64MB（对于64位体系结构）为单位获取新内存， 这块内存被称为**arena**。这块内存也会被划分页并映射到span。

- mcache：这是一个非常有趣的构造。mcache是提供给P（逻辑处理器）的高速缓存，用于存储小对象（对象大小<= 32Kb）。尽管这类似于线程堆栈，但它是堆的一部分，用于动态数据。所有类大小的mcache包含scan和noscan类型mspan。Goroutine可以从mcache没有任何锁的情况下获取内存，因为一次P只能有一个锁G。因此，这更有效。mcache从mcentral需要时请求新的span。

栈

这是栈存储区，每个Goroutine（G）有一个栈。在这里存储了静态数据，包括函数栈帧，静态结构，原生类型值和指向动态结构的指针。这与分配给每个P的mcache不是一回事。

Go内存使用（栈与堆）

现在我们已经清楚了内存的组织方式，现在让我们看看程序执行时Go是如何使用Stack和Heap的。

我们使用下面的这个Go程序，代码没有针对正确性进行优化，因此可以忽略诸如不必要的中间变量之类的问题，因此，重点是可视化栈和堆内存的使用情况。

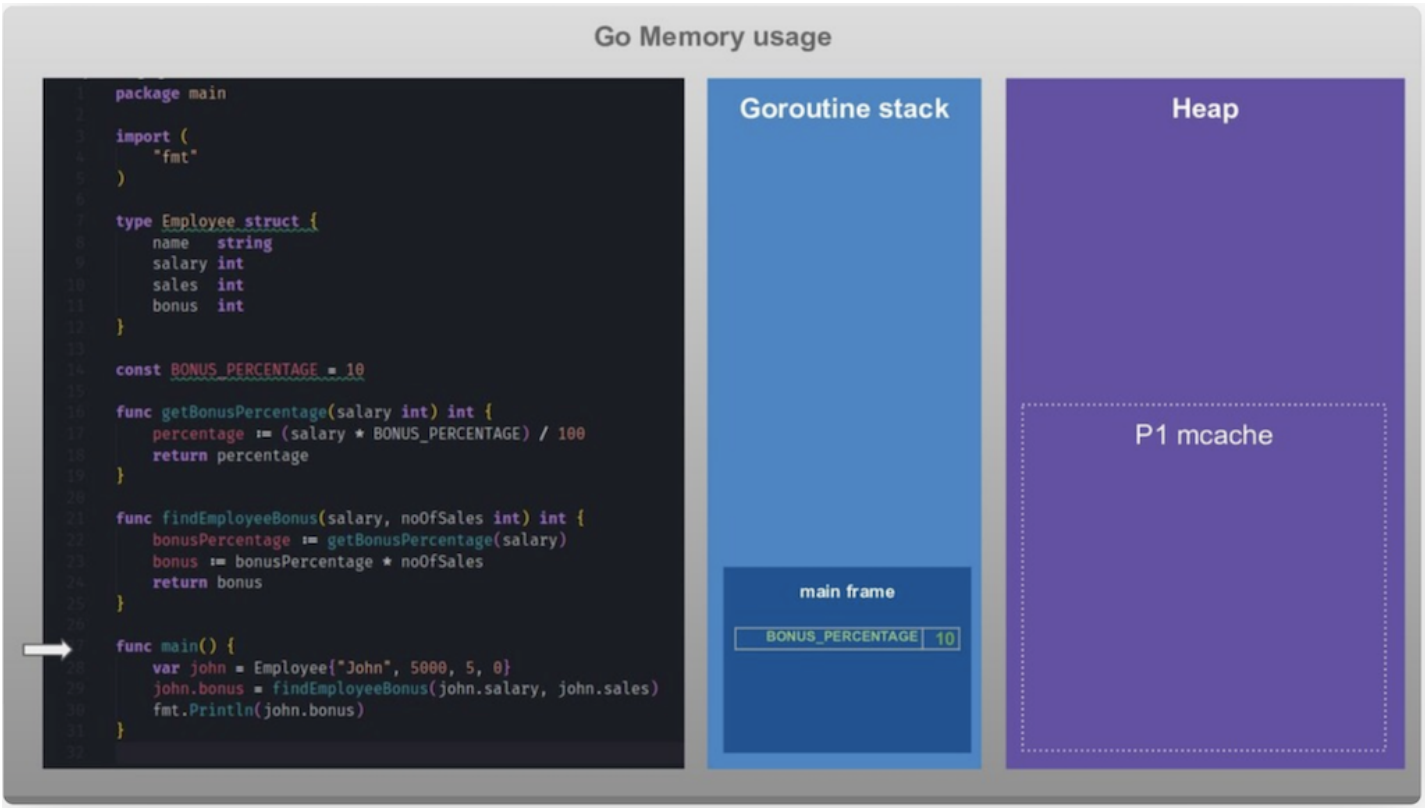
```
package main

import "fmt"
```

```
type Employee struct {
    name   string
    salary int
    sales  int
    bonus  int
}

const BONUS_PERCENTAGE = 10

func getBonusPercentage(salary int) int {
    percentage := (salary * BONUS_PERCENTAGE) / 100
    return percentage
}

func findEmployeeBonus(salary, noOfSales int) int {
    bonusPercentage := getBonusPercentage(salary)
    bonus := bonusPercentage * noOfSales
    return bonus
}

func main() {
    var john = Employee{"John", 5000, 5, 0}
    john.bonus = findEmployeeBonus(john.salary, john.sales)
    fmt.Println(john.bonus)
}
```

与许多垃圾回收语言相比，Go的一个主要区别是许多对象直接在程序栈上分配。Go编译器使用一种称为“逃逸分析”的过程来查找其生命周期在编译时已知的对象，并将它们分配在栈上，而不是在垃圾回收的堆内存中。在编译过程中，Go进行了逃逸分析，以确定哪些可以放入栈（静态数据），哪些需要放入堆（动态数据）。我们可以通过运行带有`-gcflags '-m'`标志的go build命令来查看分析的细节。对于上面的代码，它将输出如下内容：

```
❯ go build -gcflags '-m' gc.go
# command-line-arguments
temp/gc.go:14:6: can inline getBonusPercentage
temp/gc.go:19:6: can inline findEmployeeBonus
temp/gc.go:20:39: inlining call to getBonusPercentage
temp/gc.go:27:32: inlining call to findEmployeeBonus
temp/gc.go:27:32: inlining call to getBonusPercentage
temp/gc.go:28:13: inlining call to fmt.Println
temp/gc.go:28:18: john.bonus escapes to heap
temp/gc.go:28:13: io.Writer(os.Stdout) escapes to heap
temp/gc.go:28:13: main []interface {} literal does not escape
<autogenerated>:1: os.(*File).close .this does not escape
```

让我们将其可视化。单击下方图片下载幻灯片，然后翻阅幻灯片，以查看上述程序是如何执行的以及如何使用栈和堆存储器的：



可视化程序执行过程中栈和堆的使用

正如你看到的：

- main函数被保存栈中的“main栈帧”中
- 每个函数调用都作为一个栈帧块被添加到栈中
- 包括参数和返回值在内的所有静态变量都保存在函数的栈帧块内
- 无论类型如何，所有静态值都直接存储在栈中。这也适用于全局范畴
- 所有动态类型都在堆上创建，并且被栈上的指针所引用。小于32Kb的对象由P的mcache分配。这同样适用于全局范畴
- 具有静态数据的结构体保留在栈上，直到在该位置将任何动态值添加到该结构中为止。该结构被移到堆上。
- 从当前函数调用的函数被推入堆顶部
- 当函数返回时，其栈帧将从栈中删除
- 一旦主过程(main)完成，堆上的对象将不再具有来自Stack的指针的引用，并成为孤立对象

您可以看到，栈是由操作系统自动管理的，而不是Go本身。因此，我们不必担心栈。另一方面，堆并不是由操作系统自动管理的，并且由于其具有最大的内存空间并保存动态数据，因此它可能会成倍增长，从而导致我们的程序随着时间耗尽内存。随着时间的流逝，它也变得支离破碎，使应用程序变慢。解决这些问题是垃圾收集的初衷。

Go内存管理

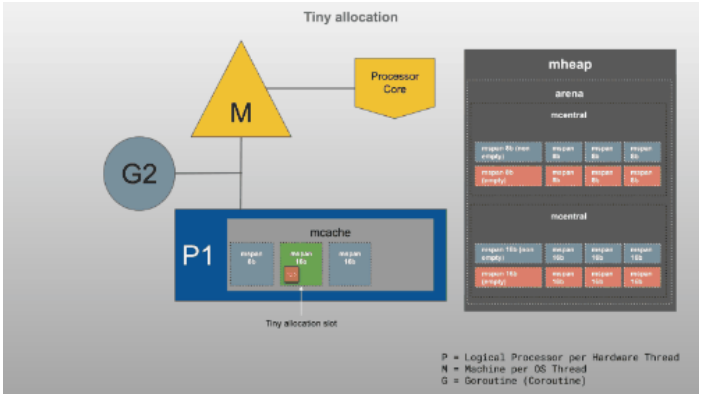
Go的内存管理包括在需要内存时自动分配内存，在不再需要内存时进行垃圾回收。这是由标准库完成的(译注：应该是运行时完成的)。与C/C++不同，开发人员不必处理它，并且Go进行的基础管理得到了高效的优化。

内存分配

许多采用垃圾收集的编程语言都使用分代内存结构来使收集高效，同时进行压缩以减少碎片。正如我们前面所看到的，Go在这里采用了不同的方法，Go在构造内存方面有很大的不同。Go使用线程本地缓存(thread local cache)来加速小对象分配，并维护着scan/noscan的span来加速GC。这种结构以及整个过程避免了碎片，从而在GC期间无需做紧缩处理。让我们看看这种分配是如何发生的。

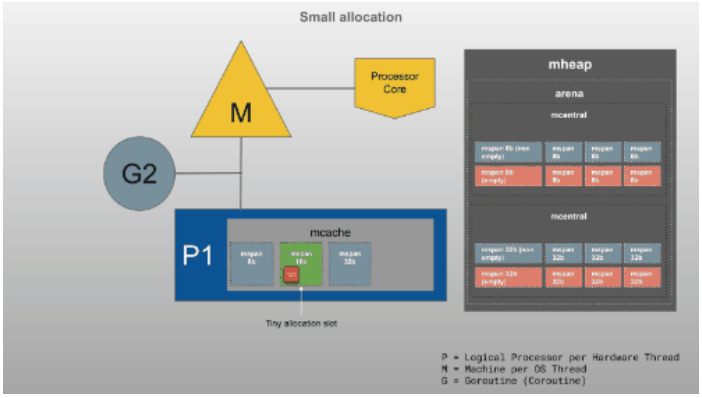
Go根据对象的大小决定对象的分配过程，分为三类：

微小对象(Tiny)（size <16B）：使用mcache的微小分配器分配大小小于16个字节的对象。这是高效的，并且在单个16字节块上可完成多个微小分配。



微小分配

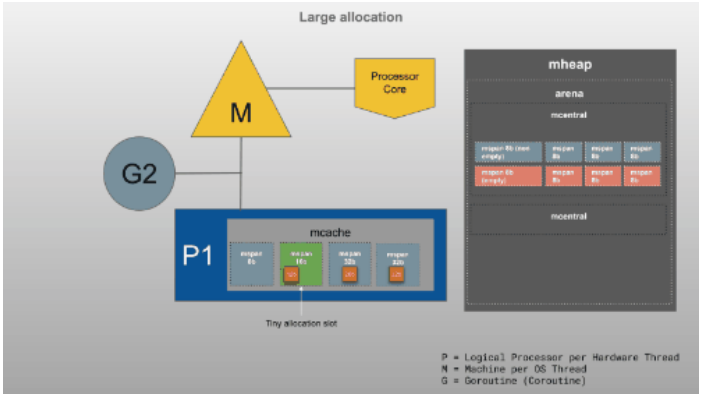
小对象（尺寸16B～32KB）：大小在16个字节和32k字节之间的对象被分配在G运行所在的P的mcache的对应的mspan size class上。



小对象分配

在微小型和小型对象分配中，如果mspan的列表为空，分配器将从mheap获取大量的页面用于mspan。如果mheap为空或没有足够大的页面满足分配请求，那么它将从操作系统中分配一组新的页（至少1MB）。

大对象（大小> 32KB）：大于32 KB的对象直接分配在mheap的相应大小类上(size class)。如果mheap为空或没有足够大的页面满足分配请求，则它将从操作系统中分配一组新的页（至少1MB）。



大对象分配

注意：您可以在此处找到以幻灯片形式记录的GIF图像

垃圾收集(GC)

现在我们知道Go如何分配内存了，让我们再看看它是如何自动回收堆内存的，这对于应用程序的性能非常重要。当程序尝试在堆上分配的内存大于可用内存时，我们会遇到内存不足的错误(out of memory)。不当的堆内存管理也可能导致内存泄漏。

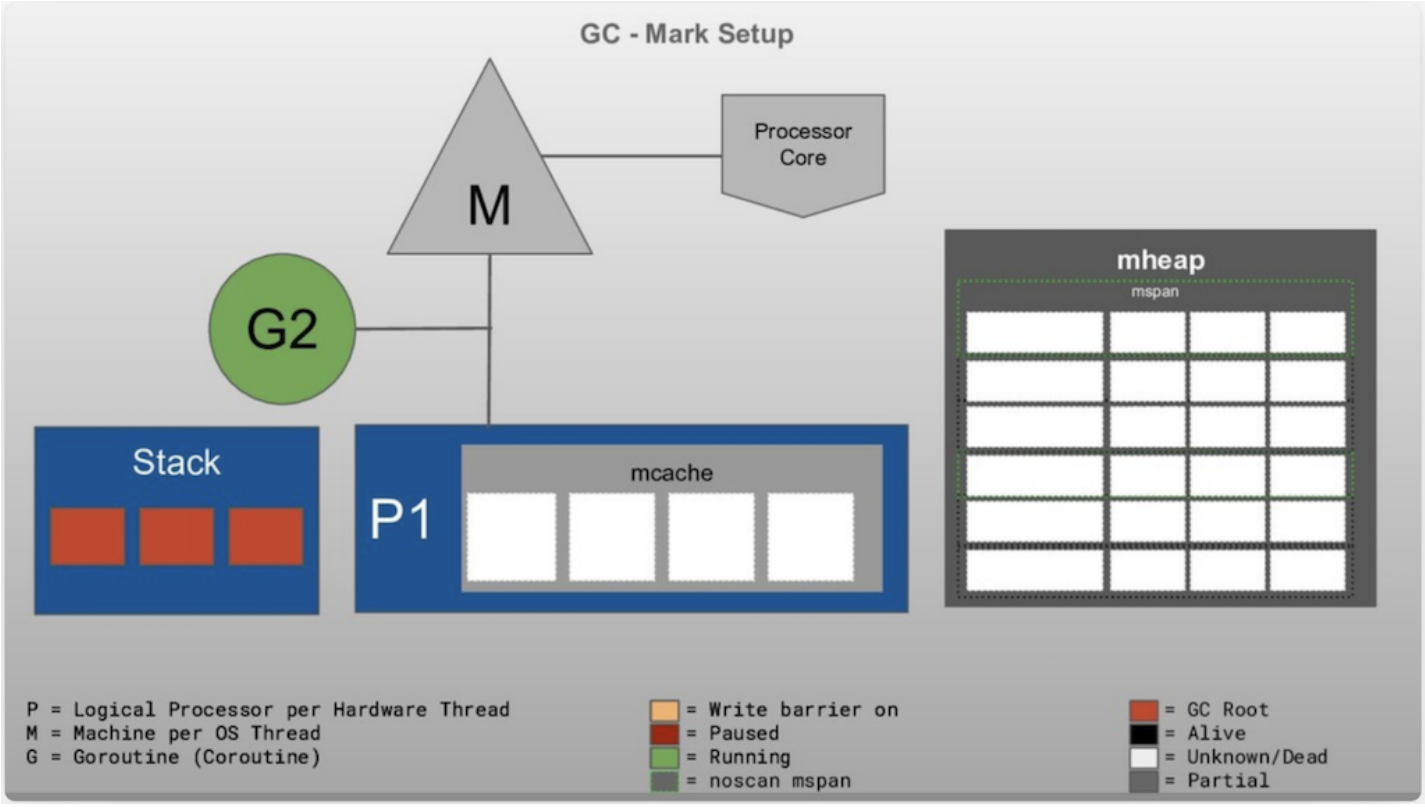
Go通过垃圾回收机制管理堆内存。简单来说，它释放了孤儿对象(orphan object)使用的内存，所谓孤儿对象是指那些不再被栈直接或间接（通过另一个对象中的引用）引用的对象，从而为创建新对象的分配腾出了空间。

从Go 1.12版本开始，Go使用了非分代的、并发的、基于三色标记和清除的垃圾回收器。收集过程大致如下所示，由于版本之间的差异，我不想做细节的描述。但是，如果您对此感兴趣，那么我推荐这个很棒的系列文章。

当完成一定百分比（GC百分比）的堆分配，GC过程就开始了。收集器将在不同工作阶段执行不同的工作：

- 标记设置（mark setup, stw）：GC启动时，收集器将打开写屏障(write barrier)，以便可以在下一个并发阶段维护数据完整性。此步骤需要非常小的暂停(stw)，因此每个正在运行的Goroutine都会暂停以启用此功能，然后继续。
- 标记（并发执行的）：打开写屏障后，实际的标记过程将并行启动，这个过程将使用可用CPU能力的25%。对应的P将保留，直到该标记过程完成。这个过程是使用专用的Goroutines完成的。在这个过程中，GC标记了堆中的活动对象(被任何活动的Goroutine的栈中引用的）。当采集花费更长的时间时，该过程可以从应用程序中征用活动的Goroutine来辅助标记过程。这称为**Mark Assist**。
- 标记终止（stw）：标记一旦完成，每个活动的Goroutine都会暂停，写入屏障将关闭，清理任务将开始执行。GC还会在此处计算下一个GC目标。完成此操作后，保留的P的会释放回应用程序。
- 清除（并发）：当完成收集并尝试分配后，清除过程开始将未标记为活动的对象回收。清除的内存量与分配的内存量是同步的(即回收后的内存马上可以被再分配了)。

让我们在一个Goroutine中看看这个过程。为了简洁起见，将对象的数量保持较小。单击下面图片，可下载幻灯片，然后翻阅幻灯片查看该过程：



xx

- 我们以一个Goroutine为例，实际过程是对所有活动Goroutine都进行的。首先打开写屏障。
- 标记过程选择GC root并将其着色为黑色，并以深度优先的树状方式遍历该该根节点里面的指针，将遇到的每个对象都标记为灰色
- 当它到达noscan span中的某个对象或某个对象不再有指针时，它完成了这个根节点的标记操作并选取下一个GC root对象
- 当扫描完所有GC root节点之后，它将选取灰色对象，并以类似方式继续遍历其指针
- 如果在打开写屏障时，指向对象的指针发生任何变化，则该对象将变为灰色，以便GC对其进行重新扫描
- 当不再有灰色对象留下时，标记过程完成，并且写屏障被关闭
- 当分配开始时(因为写屏障关闭了)，清除过程也会同步进行

我们看到这里有一些停止世界(stop)的过程，但是通常这个过程非常快，在大多数情况下可以忽略不计。对象的着色在span的gcmarkBits属性中进行。

结论

这篇文章为您提供了Go内存结构和内存管理的概述。这里不是全面详尽的说明，有许多更高级的概念，实现细节在各个版本之间都在不断变化。但是对于大多数Go开发人员来说，这些信息就已经足够了，我希望它能帮助您编写出更好的、性能更高的应用程序，牢记这些，将有助于您避免下一个内存泄漏问题。

参考文献

- blog.learngoprogramming.com https://blog.learngoprogramming.com/a-visual-guide-to-golang-memory-allocator-from-ground-up-e132258453ed
- www.ardanlabs.com https://www.ardanlabs.com/blog/2018/12/garbage-collection-in-go-part1-semantics.html
- povilasv.me https://povilasv.me/go-memory-management/
- medium.com/a-journey-with-go https://medium.com/a-journey-with-go/go-memory-management-and-allocation-a7396d430f44
- medium.com/a-journey-with-go https://medium.com/a-journey-with-go/go-how-does-the-garbage-collector-mark-the-memory-72cfc12c6976
- hub.packtpub.com https://hub.packtpub.com/implementing-memory-management-with-golang-garbage-collector/
- making.pusher.com https://making.pusher.com/golangs-real-time-gc-in-theory-and-practice/
- segment.com/blog https://segment.com/blog/allocation-efficiency-in-high-performance-go-services/
- go101.org https://go101.org/article/memory-block.html

我的联系方式：

微博：https://weibo.com/bigwhite20xx
微信公众号：iamtonybai
博客：tonybai.com
github: https://github.com/bigwhite

微信赞赏：

商务合作方式：撰稿、出书、培训、在线课程、合伙创业、咨询、广告合作。

Related posts:

1. 图解Go内存分配器
2. 也谈goroutine调度器
3. Go语言回顾：从Go 1.0到Go 1.13
4. Go语言是如何处理栈的
5. Go 1.14中值得关注的几个变化

已有 8 条评论

1. *峰云就她了*
   2020/03/10

   动画的图有些不清晰，压缩太猛了吧

   回复
2. *峰云就她了*
   2020/03/10

   哦，原文就这样。

   回复
3. *aka*
   2020/03/20

   > 每个函数调用都作为一个栈帧块被添加到*堆*中

   怎么看图感觉是添加到*栈*中呢?

   回复
   - *bigwhite*
     2020/03/21

     你看的很细。笔误，已改。3ks

回复

4. *banbo*
2020/08/26

“无论类型如何，所有静态值都直接存储在栈中。这也适用于全局范畴”

https://studygolang.com/articles/25632
这篇文章的结论是：
初始化的全局变量分配在.data段(Section)内
未初始化的全局变量分配在.bss段(Section)内

我本地用gdb调试也是这样的，博主怎么看

回复

5. *wmpeng*
2020/08/28

“从当前函数调用的函数被推入堆顶部” 这里应该也是栈吧?

回复

6. *andy*
2021/03/18

声明的局部结构体变量应该是直接存储在栈中的吧，不可能存储在堆中，大小确定，不会逃逸。

回复

- *bigwhite*
2021/03/19

```
$cat -n test6.go
1 package main
2
3 import “fmt”
4
5 type foo struct {
6 a int
7 }
8
9 func demo1() {
10 var f foo
11 f.a = 7
12 var p *foo = &f
13 fmt.Printf(“%p\n”, p)
14 }
15
16 func main() {
17 demo1()
18 }

$go build -gcflags=’-N -l -m’ test6.go
# command-line-arguments
./test6.go:10:6: moved to heap: f
./test6.go:13:12: … argument does not escape
```

go 1.16下测试。

回复

添加新评论

称呼

邮箱

网站 http://example.com

提交评论

本站Powered by Digital Ocean VPS。
选择Digital Ocean VPS主机，即可获得10美元现金充值，可 免费使用两个月哟！ 著名主机提供商Linode 10$优惠码：linode10，在 这里注册即可免费获 得。阿里云推荐码： 1WFZ0V， 立享9折！

我的业余项目

- smspush短信发送平台

文章

- 一文搞懂Go语言的plugin
- 一文告诉你如何用好uber开源的zap日志库
- 使用section.key的形式读取ini配置项
- 使用go-metrics在Go应用中增加度量
- 通过实例理解Go Execution Tracer
- 使用functrace辅助进行Go项目源码分析
- 通过实例理解Go逃逸分析
- minikube v1.20.0版本的一个bug
- Go标准库http与fasthttp服务端性能比较
- 使用reflect包在反射世界里读写各类型变量

评论

- Leo 在 Go语言的“黑暗角落”：盘点学习Go语言时遇到的那些陷阱[译]（第一部分）
  感谢分享！
- Leo 在 Go标准库http与fasthttp服务端性能比较
  搜索点进来，看了几篇文章，发现质量好高！厉害！
- stemon 在 论golang Timer Reset方法使用的正确姿势
  为什么不这样呢：timeout := time.NewTimer(T)for { timeou...
- sbilly 在 使用functrace辅助进行Go项目源码分析
  感觉这样更方便```bashgo install github.com/bigwhite/funct...
- bigwhite 在 Go语言的“黑暗角落”：盘点学习Go语言时遇到的那些陷阱[译]（第二部分）
  感谢，我检查一下。
- delphier 在 Go语言的“黑暗角落”：盘点学习Go语言时遇到的那些陷阱[译]（第二部分）
  GET https://secure.gravatar.com/avatar/157bf082b86...
- 雄哼哼 在 通过实例深入理解sync.Map的工作原理
  而且写的时候，明显是对整个map加锁的为什么会适合，多个goroutine读/写/修改的key集合没...
- 雄哼哼 在 通过实例深入理解sync.Map的工作原理
  “另外一种多个goroutine读/写/修改的key集合没有交集。”这个点没有解释把？
- Muyinliu 在 Hello，Termux
  实际测试并不能用 QQ 拼音输入法在 Termux 里面使用蓝牙键盘直接输入中文，只能在 Termu...
- Muyinliu 在 Hello，Termux
  SwiftKey 可能是目前唯一一个在 Termux 里面使用外接键盘能输入中文的输入法，但是 Sw...
- 下一页 »

分类

- 光影汇 (7)
- 影音坊 (36)
- 思考控 (66)

- 技术志 (670)
- 教育记 (2)
- 杂货铺 (75)
- 生活簿 (157)
- 职场录 (14)
- 读书吧 (14)
- 运动迷 (107)
- 驴友秀 (40)

标签

Blog Blogger C container Cpp docker GCC github GNU Go Golang Google Gopher goroutine Java k8s Kernel Kubernetes Linux M10 Opensource Programmer Python Solaris Subversion Ubuntu Unix Windows 博客 学习 容器 工作 巴萨 开源 思考 感悟 摄影 旅游 标准库 梅西 生活 程序员 编译器 西甲 足球

归档

- 2021 年七月 (4)
- 2021 年六月 (2)
- 2021 年五月 (2)
- 2021 年四月 (7)
- 2021 年三月 (7)
- 2021 年二月 (5)
- 2021 年一月 (4)
- 2020 年十二月 (11)
- 2020 年十一月 (9)
- 2020 年十月 (1)
- 2020 年九月 (1)
- 2020 年八月 (1)
- 2020 年七月 (1)
- 2020 年六月 (4)
- 2020 年五月 (3)
- 2020 年四月 (2)
- 2020 年三月 (6)
- 2020 年二月 (2)
- 2019 年十二月 (2)
- 2019 年十一月 (6)
- 2019 年十月 (5)
- 2019 年九月 (4)
- 2019 年八月 (5)
- 2019 年七月 (1)
- 2019 年六月 (2)
- 2019 年五月 (1)
- 2019 年四月 (4)
- 2019 年三月 (2)
- 2019 年二月 (1)
- 2019 年一月 (2)
- 2018 年十一月 (3)
- 2018 年十月 (1)
- 2018 年九月 (1)
- 2018 年七月 (1)
- 2018 年六月 (4)
- 2018 年五月 (2)
- 2018 年四月 (1)
- 2018 年三月 (3)
- 2018 年二月 (3)
- 2018 年一月 (7)
- 2017 年十二月 (5)
- 2017 年十一月 (4)
- 2017 年十月 (3)
- 2017 年九月 (2)
- 2017 年八月 (3)

- 2012 年二月 (6)
- 2012 年一月 (6)
- 2011 年十二月 (4)
- 2011 年十一月 (4)
- 2011 年十月 (5)
- 2011 年九月 (8)
- 2011 年八月 (7)
- 2011 年七月 (6)
- 2011 年六月 (7)
- 2011 年五月 (8)
- 2011 年四月 (6)
- 2011 年三月 (10)
- 2011 年二月 (7)
- 2011 年一月 (10)
- 2010 年十二月 (7)
- 2010 年十一月 (6)
- 2010 年十月 (7)
- 2010 年九月 (12)
- 2010 年八月 (8)
- 2010 年七月 (3)
- 2010 年六月 (5)
- 2010 年五月 (4)
- 2010 年四月 (2)
- 2010 年三月 (6)
- 2010 年二月 (4)
- 2010 年一月 (6)
- 2009 年十二月 (6)
- 2009 年十一月 (6)
- 2009 年十月 (5)
- 2009 年九月 (8)
- 2009 年八月 (8)
- 2009 年七月 (8)
- 2009 年六月 (2)
- 2009 年五月 (5)
- 2009 年四月 (7)
- 2009 年三月 (12)
- 2009 年二月 (9)
- 2009 年一月 (15)
- 2008 年十二月 (9)
- 2008 年十一月 (5)
- 2008 年十月 (10)
- 2008 年九月 (13)
- 2008 年八月 (13)
- 2008 年七月 (3)
- 2008 年六月 (1)
- 2008 年五月 (7)
- 2008 年四月 (4)
- 2008 年三月 (9)
- 2008 年二月 (11)
- 2008 年一月 (15)
- 2007 年十二月 (11)
- 2007 年十一月 (14)
- 2007 年十月 (4)
- 2007 年九月 (5)
- 2007 年八月 (1)
- 2007 年七月 (10)
- 2007 年六月 (10)
- 2007 年五月 (10)
- 2007 年四月 (8)
- 2007 年三月 (15)
- 2007 年二月 (4)
- 2007 年一月 (17)

- 2006 年十二月 (18)
- 2006 年十一月 (9)
- 2006 年十月 (11)
- 2006 年九月 (6)
- 2006 年八月 (5)
- 2006 年七月 (22)
- 2006 年六月 (35)
- 2006 年五月 (24)
- 2006 年四月 (26)
- 2006 年三月 (25)
- 2006 年二月 (18)
- 2006 年一月 (15)
- 2005 年十二月 (10)
- 2005 年十一月 (10)
- 2005 年九月 (13)
- 2005 年八月 (11)
- 2005 年七月 (6)
- 2005 年六月 (2)
- 2005 年五月 (3)
- 2005 年四月 (6)
- 2005 年三月 (1)
- 2005 年一月 (15)
- 2004 年十二月 (9)
- 2004 年十一月 (14)
- 2004 年十月 (2)
- 2004 年九月 (2)

私人

- 我的二女儿
- 我的大女儿

链接

- @douban
- @flickr
- @github
- @googlecode
- @picasa
- @slideshare
- @twitter
- @weibo
- Hoterran
- Lionel Messi
- Puras He
- 梦想风暴
- 磊磊落落的博客
- 过眼云烟

开源项目

- buildc
- cbehave
- lcut

翻译项目

- C语言编码风格和标准
- 《Programming in Haskell》中文翻译项目

View My Stats