1.10

Go 学习笔记

第六版 · 下卷

前言

预览版。后续章节将发布到公众号，敬请关注。

雨　痕

二〇一八，初夏

更新

2012-01-11　开始学习。
2012-01-15　第一版，基于 R60。
2012-03-29　升级至 1.0。
2012-06-15　升级至 1.0.2。
2013-03-26　升级至 1.1。
2013-12-12　第二版，基于 1.2。
2014-05-22　第三版，基于 1.3。
2014-12-20　第四版，基于 1.4。
2015-06-15　第五版，基于 1.5。
2015-11-01　重写 第五版。
2015-12-09　第五版 下卷，基于 1.5.1。
2016-04-01　第五版 上卷，基于 1.6。
2018-05-07　第六版 下卷，基于 1.10。

下卷　源码剖析

Go 1.10

一. 准备

本书重点剖析运行时（runtime）相关内部机制，以便了解程序运行状态。这有助于深入理解语言规则，写出更高效的代码。无论是规避垃圾回收潜在问题，还是为了减少内存开销，提升算法性能。

> 为便于排版和阅读，相关代码均做了大幅删减，如有疑问请对照原始文件。
> 如版本和安装位置不同，示例代码和输出结果会有所差异，请以实际为准。

相关环境：

```
$ go version
go version 1.10.2 darwin/amd64

$ uname -a
Linux 4.9.87 x86_64

$ gdb --version
GNU GDB 8.0.1
```

二. 引导

语言简单不意味着编译出来的程序也会简单。无论如何，系统和硬件要完成指定工作，就需要对应的指令，这些不会因为高层语言而简化。如此，从语言层面剥离出去的复杂部分就需要额外的机制去完成。这个在幕后工作的机制，通常称作运行时，或其他什么。

以 Go 为例，编译出来的程序包含两部分：运行时和用户逻辑。用户逻辑以 main.main 为入口函数，那么运行时如何启动？如何初始化？有哪些具体组成部分？

研究这些问题，首先要找到正确起点。通常我们会采用“逆向”方式，而非一头栽到源码目录里，那样太低效。

既然是程序执行起点，那么随便编译一个可执行文件，使用 GDB 动态查看即可。

test.go

```
package main

func main() {
}
```

```
$ go build -o test
$ gdb test

(gdb) info files
Entry point: 0x4477c0

(gdb) b *0x4477c0
Breakpoint 1 at 0x4477c0: file /usr/local/go/src/runtime/rt0_linux_amd64.s, line 8.

(gdb) info symbol 0x4477c0
_rt0_amd64_linux in section .text
```

> 当然，也可使用 readelf、nm 等 GNU binutils 工具静态获取相关信息。
> 如在 docker 中运行，可能需要添加 --privileged 参数创建特权容器，以便使用 GDB。

这才是编译后的可执行程序真正起点。据此，从运行时源码目录中打开对应文件，定位到指定行号。鉴于该目录下有跨平台文件，注意选择和当前系统对应的文件名。

> 默认编译方式为可执行文件（-buildmode=exe）。
> 本书忽略包模式代码（-buildmode=c-archive or c-shared），相关内容请自行查看。

rt0_linux_amd64.s

```
TEXT _rt0_amd64_linux(SB),NOSPLIT,$-8
    JMP     _rt0_amd64(SB)
```

```
(gdb) b _rt0_amd64
Breakpoint 2 at 0x444100: file /usr/local/go/src/runtime/asm_amd64.s, line 15.
```

操作系统按惯例将命令行参数解析后存储到默认线程栈，然后由程序自行处理。至于汇编相关知识，请阅读本书上卷。

asm_amd64.s

```
// _rt0_amd64 is common startup code for most amd64 systems when using
// internal linking. This is the entry point for the program from the
// kernel for an ordinary -buildmode=exe program. The stack holds the
// number of arguments and the C-style argv.

TEXT _rt0_amd64(SB),NOSPLIT,$-8
    MOVQ    0(SP), DI                   // argc
    LEAQ    8(SP), SI                   // argv
    JMP     runtime·rt0_go(SB)
```

```
(gdb) b runtime.rt0_go
Breakpoint 3 at 0x444110: file /usr/local/go/src/runtime/asm_amd64.s, line 89.
```

> 注意：源码中使用中间点（runtime · rt0_go），但编译后的符号不同。

剔除无关代码后，真正的引导过程就显现在我们眼前。

首先，完成命令行参数、操作系统，以及调度器等一系列必须的初始化过程。接下来，创建 main goroutine 运行 runtime.main 函数。从这点上说， 整个进程从一开始就以并发模式运行。

asm_amd64.s

```
TEXT runtime·rt0_go(SB),NOSPLIT,$0
```

```
    ...

    CALL    runtime·args(SB)
    CALL    runtime·osinit(SB)
    CALL    runtime·schedinit(SB)

    // create a new goroutine to start program
    MOVQ    $runtime·mainPC(SB), AX                // entry
    PUSHQ   AX
    PUSHQ   $0                                     // arg size
    CALL    runtime·newproc(SB)
    POPQ    AX
    POPQ    AX

    // start this M
    CALL    runtime·mstart(SB)

    MOVL    $0xf1, 0xf1                            // crash（确保不会执行到此，调试用代码）
    RET
```

```
DATA runtime·mainPC+0(SB)/8, $runtime·main(SB)
```

函数 newproc 用于创建 goroutine 任务，将其放入待运行队列。而 mstart 则让线程进入任务调度模式，从而自队列中提取 main goroutine 并执行。

```
(gdb) b runtime.main
Breakpoint 5 at 0x4228b0: file /usr/local/go/src/runtime/proc.go, line 109.

(gdb) b runtime.newproc
Breakpoint 4 at 0x42a2f0: file /usr/local/go/src/runtime/proc.go, line 3240.
```

proc.go

```
func newproc(siz int32, fn *funcval) {
    ...
}
```

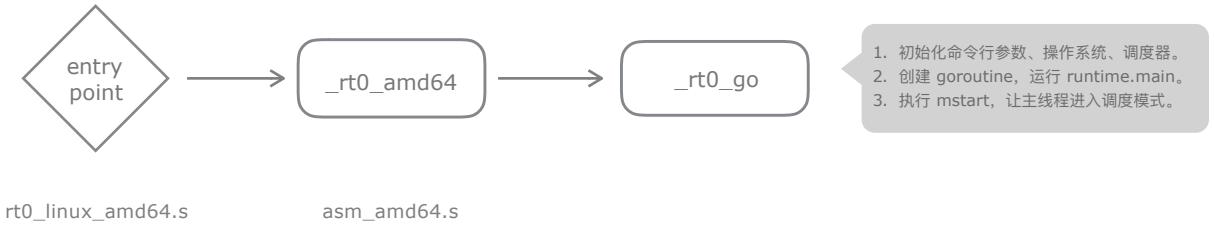
至此，初步获取了 Go 可执行程序启动引导过程，从而为后续深度挖掘提供线索。

在分析复杂体系架构时，建议先提取路线图。以此为路径，从宏观层面将流程和细节分开处理。好处在于，按图索骥跳转到需要关注的内容，而非浏览整个文件。多数时候，真正要了解的仅是执行逻辑，以及特定算法实现。所谓“遗其貌，得其骨，取其神”。

阅读源码时，可将目标代码片段拷贝到代码编辑器。然后快速浏览，以阅读自然文字的方式，标记出可能需要关注的行。待浏览完毕，有初步整体印象后，着手删减与标记内容无关的代码。如此，可提高阅读效率，将精力放在所需内容上，远比在大段代码间滚屏好得多。要知道，阅读和编码是两码事。我们只需知道实现过程和意图，类似文本摘要，根本无需关心删减后的代码能否执行，那是原作者的事情，与我们无关。

该方式也可用来提高编码质量，先以伪码（包括注释）实现骨架，然后再补充并完善细节。

快速导览：



三. 初始化

与命令行参数有关的内容不值得花心思，略过。而与操作系统相关的，仅仅是确定 CPU 处理器数量。

在并发编程时，CPU 处理器数量是重要的决定性参数。它决定了我们应该采取什么样的并行策略，甚至会影响架构设计。也因为如此，我们要知道相关函数（runtime.NumCPU）返回的是物理核数量，还是包含超线程（Hyper-Threading）的结果。

超线程技术是利用特殊指令，在单个物理核内虚拟多个逻辑处理器。这有点像多线程，将等待时间挖掘出来执行其他任务，以提升整体性能。可问题在于，多个逻辑处理器毕竟共享某些资源，某些时候可能适得其反拖累执行效率，比如缓存刷新等等。

程序员应该对硬件体系，以及操作系统有些基本认识。

---

runtime2.go

```
var ncpu int32
```

os_linux.go

```
func osinit() {
    ncpu = getproccount()       // 返回逻辑处理器数量。
}
```

debug.go

```
// returns the number of logical CPUs usable by the current process.

func NumCPU() int {
    return int(ncpu)
}
```

相比之下，调度器初始化内容就丰富得多。与之有关的内容包括内存管理、垃圾回收，以及并发任务数量。这些内容在后续章节详叙，此处只了解初始化都做了些什么即可。

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() {

    // M 最大数量限制。
    sched.maxmcount = 10000

    // 内存相关初始化。
    stackinit()
    mallocinit()

    // M 相关初始化。
    mcommoninit(_g_.m)

    // 存储命令行参数和环境变量。
    goargs()
    goenvs()

    // 解析 GODEBUG 的调试参数。
    parsedebugvars()

    // 初始化垃圾回收器。
    gcinit()

    // 初始化 poll 时间。
    sched.lastpoll = uint64(nanotime())

    // 设置 GOMAXPROCS。
    procs := ncpu
    if n, ok := atoi32(gogetenv("GOMAXPROCS")); ok && n > 0 {
        procs = n
    }
    if procresize(procs) != nil {
        throw("unknown runnable goroutine during bootstrap")
    }
}
```

> 与前一版本最大的区别是取消了 GOMAXPROCS 256 上限。

在完成核心初始化操作后，创建并执行 main goroutine，也就是 runtime.main 函数。

不过呢，有个概念要分清楚，前文所说初始化属于运行时内核层面。还有一种初始化属于逻辑层面，包括 runtime 包里的 init 函数，以及标准库、用户、第三方包初始化函数。不要看轻逻辑初始化执行方式，这可能关系到代码依赖，同步处理等等问题。

不要惊讶于栈（stack）可以有 1 GB 大小，这与 goroutine 实现方式有关。
说实话，我个人觉着这里面的一些内容应该放到 schedinit 里。

proc.go

```
// The main goroutine.
func main() {

    // 栈最大值。（64 位系统下 1GB）
    if sys.PtrSize == 8 {
        maxstacksize = 1000000000
    } else {
        maxstacksize = 250000000
    }

    // 启动后台监控。
    systemstack(func() {
        newm(sysmon, nil)
    })

    // 执行 runtime 包内的初始化函数。
    runtime_init()

    // 启动时间。
    runtimeInitTime = nanotime()

    // 启动垃圾回收器。
    gcenable()

    // 执行用户、标准库、第三方库初始化函数。
    fn := main_init
    fn()

    // 如果是库方式，则不执行用户入口函数。
    if isarchive || islibrary {
        // A program compiled with -buildmode=c-archive or c-shared
        // has a main, but it is not executed.
        return
    }

    // 执行用户入口函数。
    fn = main_main
    fn()

    // 退出进程。
    exit(0)
}
```

很显然，目标是 runtime.init、main.init，以及 main.main 这个用户入口函数。

```
//go:linkname runtime_init runtime.init
func runtime_init()

//go:linkname main_init main.init
func main_init()

//go:linkname main_main main.main
func main_main()
```

这两个初始化函数由编译器动态生成，所以需要写个示例来看看具体情形。

test.go

```
package main

import (
    "fmt"
    "runtime"
    _ "net/http"                                    // 标准库
    _ "github.com/shirou/gopsutil/cpu"              // 第三方库
)

func init() {                                       // 自定义
    println("user init1.")
}

func init() {
    println("user init2.")
}

func main() {
    fmt.Println(runtime.NumCPU())
}
```

编译调试版本，应禁用内联（inlining）和优化（optimizations）。

```
$ go build -gcflags "-N -l" -o test
```

用自带工具反汇编，输出编译器如何处理 init 初始化函数。这些函数都被重新命名，这也是同一包，甚至同一文件中允许有多个同名初始化函数的缘故。

```
$ go tool objdump test | grep "TEXT runtime\.init"
```

```
TEXT runtime.init.0(SB)  /usr/local/go/src/runtime/cpuflags_amd64.go
TEXT runtime.init.1(SB)  /usr/local/go/src/runtime/mgcwork.go
TEXT runtime.init.2(SB)  /usr/local/go/src/runtime/mstats.go
TEXT runtime.init.3(SB)  /usr/local/go/src/runtime/panic.go
TEXT runtime.init.4(SB)  /usr/local/go/src/runtime/proc.go
TEXT runtime.init.5(SB)  /usr/local/go/src/runtime/signal_unix.go
TEXT runtime.init(SB)    <autogenerated>
```

```
$ go tool objdump test | grep "TEXT main\.init"

TEXT main.init.0(SB)  /yuhen/go/src/test/test.go
TEXT main.init.1(SB)  /yuhen/go/src/test/test.go
TEXT main.init(SB)    <autogenerated>
```

接下来，反汇编 runtime.init 和 main.init。可以看出所有初始化函数都由它们完成调用。

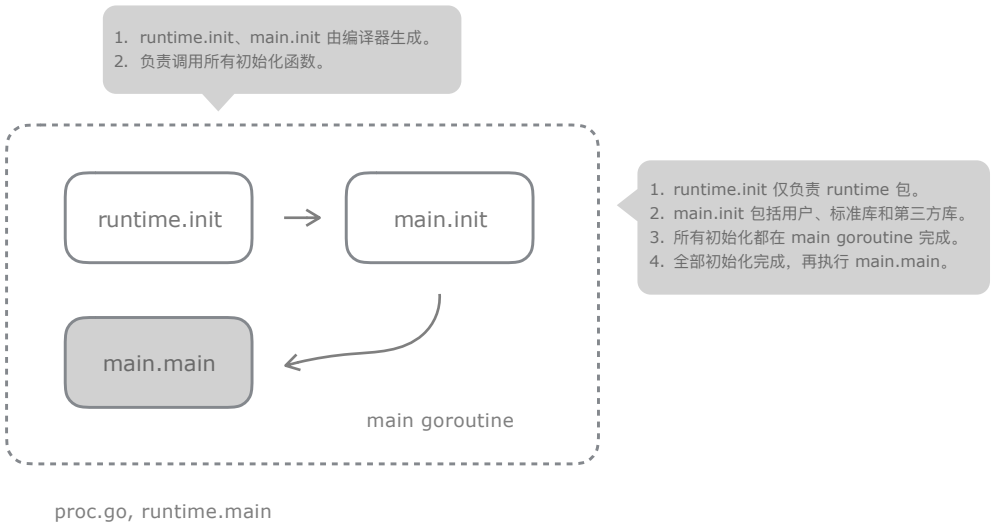
```
$ go tool objdump -s "runtime\.init" test | grep "CALL.*init"

  <autogenerated>      CALL runtime.init.0(SB)
  <autogenerated>      CALL runtime.init.1(SB)
  <autogenerated>      CALL runtime.init.2(SB)
  <autogenerated>      CALL runtime.init.3(SB)
  <autogenerated>      CALL runtime.init.4(SB)
  <autogenerated>      CALL runtime.init.5(SB)
```

```
$ go tool objdump -s "main\.init" test | grep "CALL.*init"

  <autogenerated>      CALL fmt.init(SB)
  <autogenerated>      CALL net/http.init(SB)
  <autogenerated>      CALL github.com/shirou/gopsutil/cpu.init(SB)
  <autogenerated>      CALL main.init.0(SB)
  <autogenerated>      CALL main.init.1(SB)
```

基于输出结果看，runtime.init 仅负责 runtime 包，main.init 则包含标准库、第三方库，以及用户自定义函数。另外，所有初始化逻辑都在 main goroutine 内执行。待这些初始化全部完成后，再调用 main.main 函数，转入用户逻辑。

不管编译器实现算法是否对 init 函数进行排序，我们都不应该让多个初始化逻辑依赖某种特定次序。这极易造成混乱，引发潜在错误。正确做法是，初始化函数仅负责当前包，甚至仅仅是当前文件的逻辑。

快速导览：


1. runtime.init、main.init 由编译器生成。
2. 负责调用所有初始化函数。
runtime.init
main.init
main.main
main goroutine
1. runtime.init 仅负责 runtime 包。
2. main.init 包括用户、标准库和第三方库。
3. 所有初始化都在 main goroutine 完成。
4. 全部初始化完成，再执行 main.main。
proc.go, runtime.main

四. 内存分配

1. 概述

分配内存简单，但管理内存却极为复杂。尤其是 Go 这种完全基于并发体系的现代语言，需要在时间和空间当中寻找平衡。原始架构和代码源自同门 tcmalloc，一种专门为并发而设计的高性能内存分配器，这是极高起点。只是延续至今，为配合垃圾回收和并发调度，除基础架构外，早已改得面目全非。

malloc.go

```
// Memory allocator.
//
// This was originally based on tcmalloc, but has diverged quite a bit.
// http://goog-perftools.sourceforge.net/doc/tcmalloc.html
```

如果只是研究内存分配器设计，直接阅读 tcmalloc 要更容易些。因为 Go 内存分配器代码中包含了太多其他部件的痕迹，反而显得有些凌乱。

与 tcmalloc 类似的还有 jemalloc，也是一款非常优秀的设计。

设想一下，如果手头有块空间用于零散出租，可能会面临什么状况？

虽然顾客未知，但首当其冲的必然是千奇百怪的大小需求。如果按单切分，那么这块空间该如何复用呢？为一块5字节大小的空间寻找合适的租客并不容易，几率有多大？更何况，这种需求简直没有规律可循。

Go 语言支持地址和指针，所以内存回收不能进行收缩处理，内存复用必须在原位置。

简单做法是将顾客按范围分类，如同盖房子划出 A、B、C 等户型。举例来说，8 字节空间可适应 1 ~ 8 字节需求，如只用来存储 5 字节对象，虽有浪费，但复用几率显然要高很多。另外，对于大空间需求，我们专门处理，直接卖别墅好了。

现在的基本思路是，将空间需求以某个阈值分为大小两类。大对象专门分配，这没什么好说的。反正那么大的空间根本算不上碎片，切分复用也很容易。

sizeclasses.go

```
_MaxSmallSize = 32768       // 32 KB
```

至于小对象，则以 8 倍数为单位，分成 66 种规格。

> 在相关代码中，class 0 用于表达大对象。

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%
//    ...
//    64      27264       81920        3         128     10.00%
//    65      28672       57344        2           0      4.91%
//    66      32768       32768        1           0     12.50%

_NumSizeClasses = 67
```

现在好了，只需要面对 67 种需求。初步在空间和管理之间找到一个平衡，清爽多了。

> size class 和 size 的转换，采用静态表，由 makesizeclass.go 生成。

sizeclasses.go

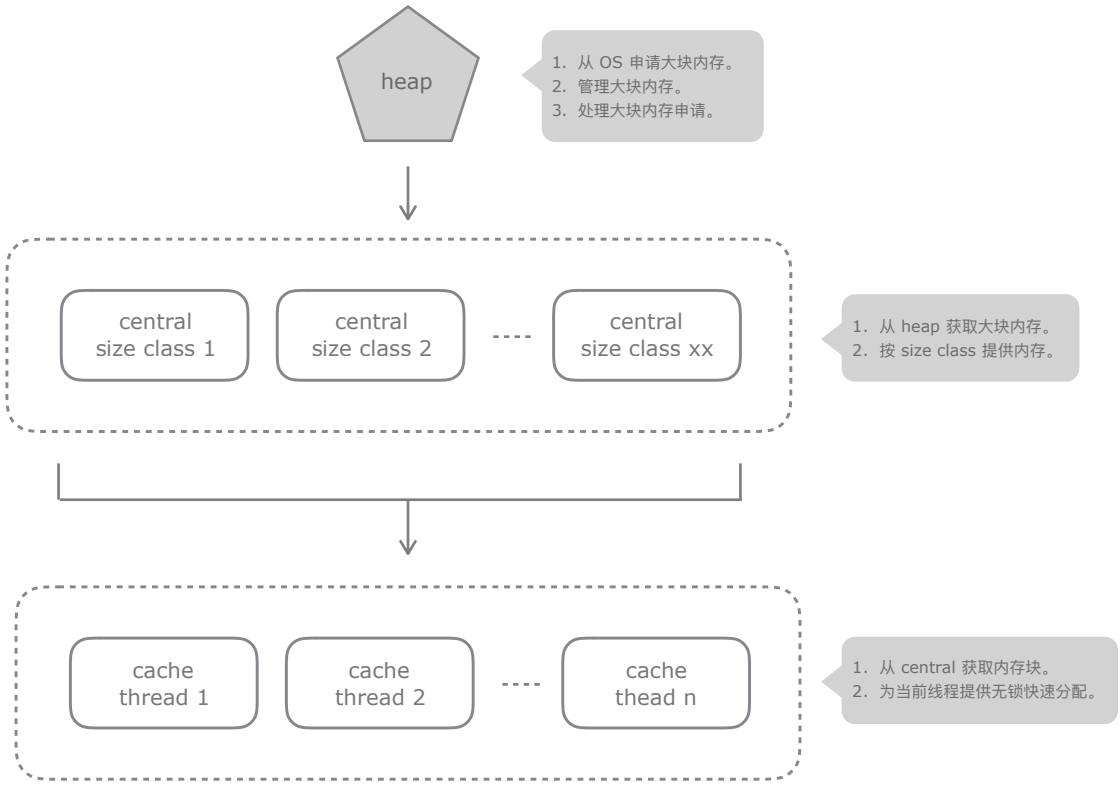
```
// Code generated by mksizeclasses.go; DO NOT EDIT.
// go:generate go run mksizeclasses.go

var class_to_size = [_NumSizeClasses]uint16{0, 8, 16, 32, 48, ..., 28672, 32768}
var class_to_allocnpages = [_NumSizeClasses]uint8{0, 1, 1, 1, 1, ..., 7, 4}
```

接下来，生意越做越大，顾客越来越多，该如何提高接待能力？让顾客排队？业务繁忙时，可能排出千百个去。这显然会影响业绩，性急的顾客可能会因此流失。

解决方案也算不上多新奇。首先，预先准备好不同大小（size class）的复用空间块；其次，增加接待窗口。基于平衡考虑，这里设置了三级机构，分别处理不同大小的空间块。

> 大内存块称作 span。而在 span 里按特定 size class 切分，存储单个小对象的块称作 object。

管理部件结构图：



顶层堆管理部件（heap）每次向操作系统申请一大块内存（最少 1 MB），以减少系统调用次数，提升性能。它还负责管理未使用大块内存（span），为大对象直接分配空间。

中间部件（central）从堆提取大块内存，为缓存部件提供后备内存池。每个中间部件仅负责一种规格，以简化管理。还有，不同规格请求会被分散到不同中间部件，如此可减小锁粒度，进一步提升性能。当然，从中间部件获取小块内存时采取批量操作，毕竟每个中间部件要为多个请求服务，存在竞争。

最后一级是与工作线程绑定的缓存部件（cache）。它从中间部件提取内存块，按规格大小计算地址偏移，返回给分配请求。因单个线程内不存在竞争，所以这里分配是无锁操作，性能会很高。直到内存消耗一空时，再与其他线程竞争。

> 虽然协程（coroutine）在单个线程上实现并发逻辑，但其执行依然是串行的。

至此，事情并未结束。因为除了出租，我们还需要回收。

从空间和时间局部性原则看，临近分配（函数或循环）的对象生命周期类似。所以，从中间部件批量转移到缓存部件的小块内存最好是连续地址，以便回收时有较大几率获得一个没有碎片的大块内存。

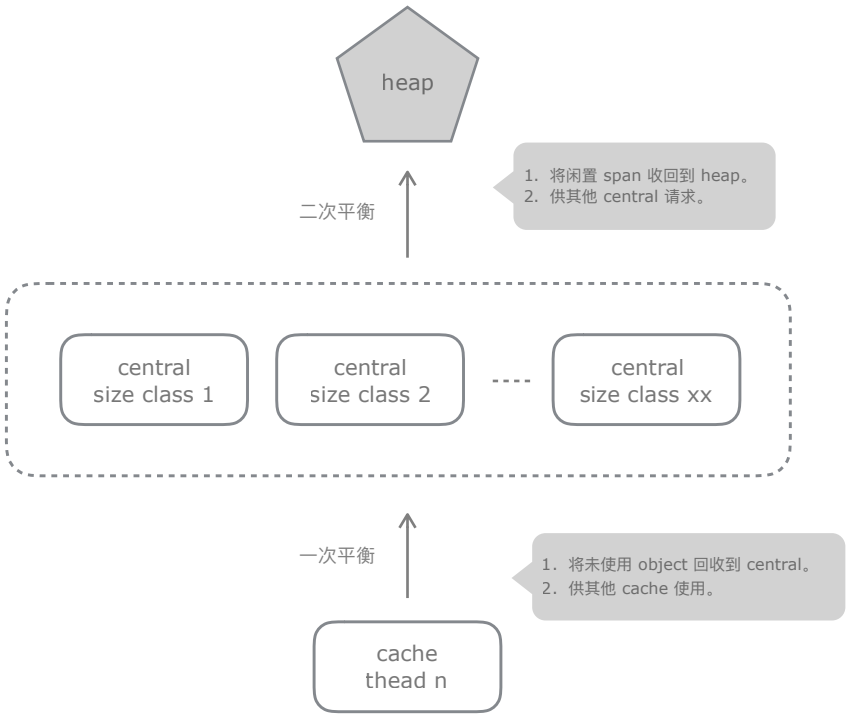
---

cache 从 central 批量提取操作，实际获取一个完整 span。那么，span 大小是固定的吗?

不是！首先，heap 向操作系统申请内存时，只是设定最小 1 MB，最大值可没有指定；其次，在 central 向 heap 请求时，会按 size class 静态表设置大小进行分割，如此就变成大小不等的两块。在回收时，还尝试与相邻地址未使用的 span 合并，以形成更大可切分空间。

试想一下，某缓存部件批量提取了 50 个特定规格的小块内存。但仅使用了 3 个，或者说相当长时间内只使用了 3 个。这会造成严重浪费，毕竟内存是种极宝贵的资源。回收操作有必要将剩余未分配小块内存重新放回中间部件，以便给其他缓存部件使用，这是基于空间利用率的第一次平衡。

还有，某中间部件在特定时期（比如执行某个算法）可能分配很多指定大小内存块。但随后，这些内存块被长时间闲置。如此，回收操作就尝试将其交还给堆。擦拭掉切分痕迹后，应对其他中间部件请求，切分成另一种规格的小内存块。此为第二次空间平衡。

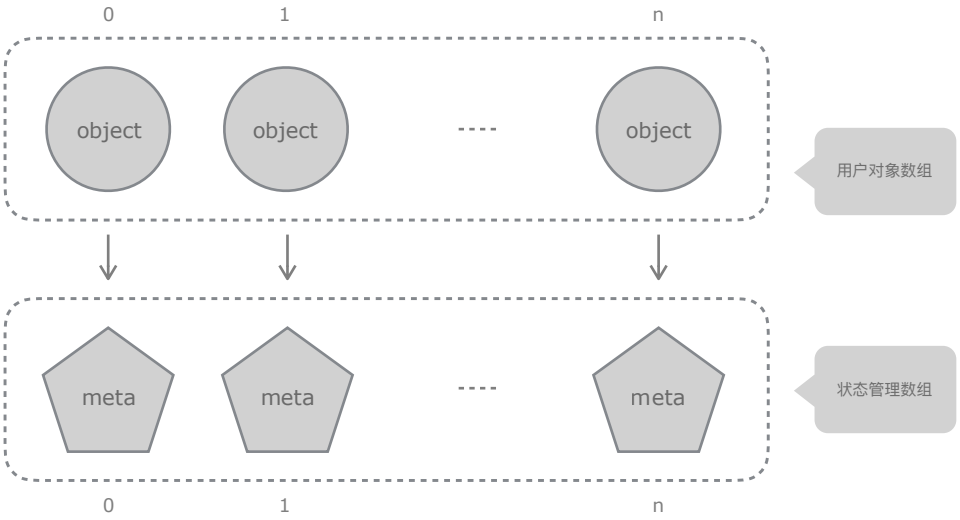


从以上描述中，可以看出三级部件基本工作机制和设计原理。至于详细内容，我们在后面章节进一步分析。

2. 初始化

内存分配算法设计思路很巧妙。

首先面临的问题是：对象没有额外附加字段，可内存分配和垃圾回收却要设置管理状态，如何存储并高效检索？鉴于每个对象在生存期内都有唯一内存地址，可以此计算出一个唯一索引号。将管理数据存储在专门数组内，以相同索引号访问岂非即简单又高效。



看上去似乎很美好，可实际会引发另一个问题。在分配某对象时，如何确保其对应元数据能同时分配？在这个算法里，用户对象内存地址和状态管理对象地址之间存在一一对应关系。如何保证状态对象所需地址不被占用？总不能将管理数组提前预分配吧？那得多大？

我们知道每个进程都有自己独立虚拟地址空间。如不考虑物理内存，64 位系统巨大的地址空间可允许我们随便使用，甚至奢侈地只管分配，不考虑释放。

当然，实际情况下，我们无法绕开物理内存限制。任何时候，超出物理内存大小都会引发严重性能问题，这不是想要的结果。可如果，只释放物理内存而不释放虚拟内存呢？

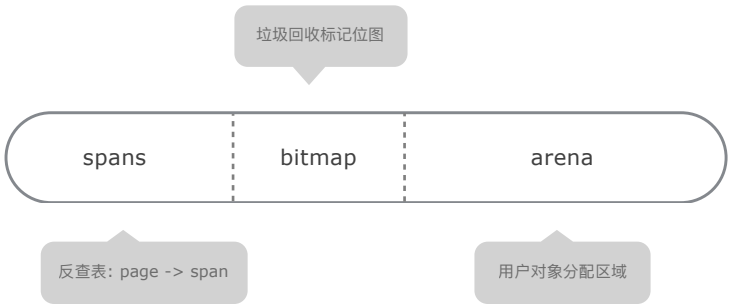
如某段虚拟内存长时间不使用，可通知操作系统解除其物理内存映射，如此其物理内存被回收。只是站在算法视角，该内存依旧存在，因为代码操作的都是虚拟内存。当被解除物

理映射的虚拟内存再次使用时，会引发缺页异常，操作系统自动补上所需物理内存。如此一来，事情就简单了。

> 回收后复用只是覆盖数据，与内存释放是两回事。
> 有关虚拟内存，以及换入换出（swap in/out）、缺页异常（page fault）等概念是必须掌握的。

不管是用户对象，还是状态管理数组，只要提前圈定所需地址空间，就不怕地址冲突了。更进一步，采取线性分配策略，每次分配一个较大范围，以提升性能。

内存布局示意图：



右边区域（arena）是专门为用户对象预留地址空间。当堆（heap）向操作系统申请大块内存时，会将此区域地址作为参数传入系统调用。完成后调整并保存下一次分配位置。

> Linux mmap、Windows VirtualAlloc 等都可传入起始地址参数。
> 另外，运行时管理对象因生命周期策略不同，不在 arena 范围内分配。

```
index = (object.address - arena.start) / step
```

中间区域（bitmap）是位图数据结构，为每个对象提供几个二进制位，用来标记垃圾回收状态。左边（spans）则是内存页（page）和其管理对象（span）反查表。毕竟内存回收时，需要从哪来回哪去。同时，利用该表还可访问地址相邻大块内存使用状态，以便将多个内存块合并成更大内存块，减少碎片，更好适应分配需求。

了解基本思路后，我们开始研究具体代码。

mheap.go

```
type mheap struct {
    spans []*mspan

    bitmap        uintptr       // Points to one byte past the end of the bitmap
    bitmap_mapped uintptr

    arena_start uintptr
    arena_used  uintptr         // Set with setArenaUsed.

    arena_alloc uintptr
    arena_end   uintptr
}
```

内存布局和线性分配位置保存在 mheap 结构，在初始化函数中进行设置。

首先需要计算出所有区域大小，通知操作系统保留内存地址范围，避免被其他对象分配操作占用。当然，此操作仅保留地址空间，不会立即分配内存。

> 这就是某些系统监视工具里，任何一个 Go 程序，哪怕仅输出 “hello, world!” 也会占用很大内存的缘故。我们要了解其显示数据具体含义，或者明确查看物理内存占用。

malloc.go

```
func mallocinit() {

    // 计算 spans、bitmap 区域大小。
    var spansSize uintptr = (_MaxMem + 1) / _PageSize * sys.PtrSize
    var bitmapSize uintptr = (_MaxMem + 1) / (sys.PtrSize * 8 / 2)

    // 64 位系统。
    if sys.PtrSize == 8 {

            // 累计所有区域大小。
            arenaSize := round(_MaxMem, _PageSize)
            pSize = bitmapSize + spansSize + arenaSize + _PageSize

            // 尝试在固定起始位置（0x0000XXc000000000，XX = 00 ... 7f）保留地址空间。
            for i := 0; i <= 0x7f; i++ {
                    switch {
                    default:
                            p = uintptr(i)<<40 | uintptrMask&(0x00c0<<32)
                    }

                    // 仅保留虚拟内存地址，并非分配内存。
                    p = uintptr(sysReserve(unsafe.Pointer(p), pSize, &reserved))
                    if p != 0 {
```

```
                            break
                    }
                }
        }

        // 地址对齐，并调整。
        p1 := round(p, _PageSize)
        pSize -= p1 - p

        // 计算 bitmap 区域范围。(mheap.bitmap 指向尾部)
        spansStart := p1
        p1 += spansSize
        mheap_.bitmap = p1 + bitmapSize
        p1 += bitmapSize

        // 计算 arena 区域范围。
        mheap_.arena_start = p1
        mheap_.arena_end = p + pSize
        mheap_.arena_used = p1
        mheap_.arena_alloc = p1

        // 初始化堆状态，包括 mheap.spans。
        mheap_.init(spansStart, spansSize)
        _g_ := getg()
        _g_.m.mcache = allocmcache()
}
```

初始化函数并不复杂，arena 大小为静态设定，上限 512 GB，超出会引发 OOM 错误。而 bitmap 和 spans 以此分别计算。

按照 size class 规则，最小 object 是 8 字节。

状态位图 bitmap 为每个对象提供 2 个二进制位，那么其大小计算公式：

bitmap.size = arena.size / min_object.size * 2 / 8　　16 GB（返回字节数，所以除以 8）

反查表 spans 是基于 page，对应计算公式：

spans.size = arena.size / page.size * pointer.size　　512 MB（表存储指针）

---

test.go

```
func main() {
    a := [512 << 30]byte{}       // 分配 512 GB，引发 OOM。
    a[512<<30-1] = 1
}
```

```
$ go build -gcflags "-N -l" -o test && ./test
```

```
runtime: out of memory: cannot allocate 549755813888-byte block (753664 in use)
fatal error: out of memory
```

至于保留操作不引发内存分配行为，这与系统调用参数有关。

PROT_NONE: Pages may not be accessed. Accessing PROT_NONE memory will result in a segfault, but the memory region will be reserved and not used for any other purpose.

MEM_RESERVE: Reserves a range of the process's virtual address space without allocating any actual physical storage in memory or in the paging file on disk.

参数 reserved 返回在指定位置保留操作是否成功。

---

mem_linux.go

```
func sysReserve(v unsafe.Pointer, n uintptr, reserved *bool) unsafe.Pointer {
    p, err := mmap(v, n, _PROT_NONE, _MAP_ANON|_MAP_PRIVATE, -1, 0)
    if err != 0 {
        return nil
    }
    *reserved = true
    return p
}
```

mem_windows.go

```
func sysReserve(v unsafe.Pointer, n uintptr, reserved *bool) unsafe.Pointer {
    *reserved = true
    v = unsafe.Pointer(stdcall4(_VirtualAlloc, uintptr(v), n, _MEM_RESERVE, _PAGE_READWRITE))
    if v != nil {
        return v
    }
}
```

至于 heap.spans 设置，被放在 mheap 初始化函数里。

以 unsafe 方式，将 heap.spans 内部字段指向对应保留区域。

---

mheap.go

```
var mheap_ mheap

func (h *mheap) init(spansStart, spansBytes uintptr) {
     sp := (*slice)(unsafe.Pointer(&h.spans))
```

```
    sp.array = unsafe.Pointer(spansStart)          // 指向 spans 指针数组。
    sp.len = 0                                     // 初始化 len 和 cap 字段。
    sp.cap = int(spansBytes / sys.PtrSize)
}
```

保留地址空间完成后，实际用户内存还是按需分配。哪怕虚拟内存已经分配，操作系统也未必会立即映射物理内存，这与其内存管理方式有关。

> 建议阅读《深入理解计算机系统》虚拟存储器 等相关章节。

3. 分配

我们知道，编译器有责任将对象尽可能分配在栈上，这有助于减少垃圾回收压力，以提升性能。如此，是否意味着栈上对象不在内存分配器管理范围?

当然不是。事实上，并发单元（goroutine）所需执行栈内存同样从保留区域获取。当对象在栈分配时，因栈内存已经存在，只需以寄存器偏移即可访问，无需从缓存和中间等部件获取，避免了额外的分配函数调用，其性能自然要高很多。且对垃圾回收器而言，整个栈内存被视作单一对象。总体数量少了，压力自然也就小了。

至于对象具体分配在栈还是堆，由编译器决定，而非代码。

test.go

```
package main

func test() *int {
    p := new(int)
    *p = 100
    return p
}

func main() {
    p := test()
    println(p, *p)
}
```

就上面示例而言，影响分配位置的决定因素是函数内联与否。

```
$ go build -o test -gcflags "-N -l"                ; 禁止优化和内联，输出优化决策。
```

```
new(int) escapes to heap

$ go tool objdump -s "main\.test" test
TEXT main.test(SB)
    CALL runtime.newobject(SB)                    ; 在堆上分配内存。
```

> 可使用 go tool compile -h 和 go tool link -h 查看编译和链接参数。

有关栈内容，后文另述，本章仅分析堆内存分配细节。

相比初始化过程，分配算法要复杂得多。为便于理解和阅读，先从流程入手，然后再深入具体算法细节。

malloc.go

```
func newobject(typ *_type) unsafe.Pointer {
    return mallocgc(typ.size, typ, true)
}
```

对于内核代码而言，性能是优先考虑因素。所以会有很多大函数，或者不同部件相混合的风格。目的是为了减少额外调用开销，这与应用编码规范不同。建议做精简处理，剔除无关内容和细节，以便于理清思路。

malloc.go

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

func mallocgc(size uintptr, typ *_type, needzero bool) unsafe.Pointer {

    if size == 0 {
        // 零长度对象 ...
    }

    if size <= maxSmallSize {
        if noscan && size < maxTinySize {
            // 微小 (tiny) 对象分配 ...
        } else {
            // 小对象分配 ...
        }
    } else {
        // 大对象分配 ...
    }
}
```

从基本流程看，以长度为分配策略条件，有零长度、小对象和大对象三类。其中小对象可进一步分出微小（tiny）对象，其目的是为了减少内存浪费。

3.1 零长度对象

首先要理解什么是零长度对象。

> 注意，size 0 和 nil 不是一回事。
>
> 前者表达一个合法对象，无非没有实际可读写内容。而后者，则表达一个未经分配的目标。因为零长度对象内容无法访问，所以不同类型可指向同一位置。但是，因内存地址可能相同，所以作为主键时需要特别注意。

test.go

```
package main

func test() (*[0]int, *struct{}) {
    var a [0]int
    var b struct{}

    return &a, &b
}

func main() {
    pa, pb := test()
    println(pa, pb)
}
```

```
$ go build -o test -gcflags "-N -l -m"    ; 禁用内联
&a escapes to heap
&b escapes to heap

$ ./test
0x4c2a00 0x4c2a00
```

如同前文所说，对象优先分配在栈上。这句话，同样适用于零长度对象。

```
$ go build -o test -gcflags "-N -m"       ; 允许内联
can inline test
main &a does not escape
main &b does not escape
```

```
$ ./test
0xc42003ff40 0xc42003ff40                ; 栈内地址

(gdb) p pa
$1 = ([0]int *) 0xc42003ff40

(gdb) info frame
Locals at 0xc42003ff30
```

如明确在堆上分配，则指向预设的同一全局变量。

malloc.go

```
// base address for all 0-byte allocations
var zerobase uintptr

func mallocgc(size uintptr, typ *_type, needzero bool) unsafe.Pointer {
    if size == 0 {
        return unsafe.Pointer(&zerobase)
    }
}
```

3.2 大对象

从尺寸上说，大对象所占内存算不上碎片。且在通常情况下，程序里的大对象数量相对较少，生命周期较长，故其内存复用的可能性也较低。如此，分开处理自然就合情合理。

> 与某些语言将大对象直接在堆上分配不同。Go 自定义栈大小上限为 1GB，依旧栈上分配优先。

malloc.go

```
func mallocgc(size uintptr, typ *_type, needzero bool) unsafe.Pointer {

    // 大于 32KB 的是大对象。
    if size <= maxSmallSize {
    } else {
        systemstack(func() {
            s = largeAlloc(size, needzero, noscan)
        })
    }
}
```

在堆上分配时，直接从 heap 获取大小合适的内存块。

malloc.go

```
func largeAlloc(size uintptr, needzero bool, noscan bool) *mspan {

    // 大块内存以页为单位。
    npages := size >> _PageShift
    if size & _PageMask != 0 {
        npages++
    }

    // 直接从 heap 获取内存块。
    s := mheap_.alloc(npages, makeSpanClass(0, noscan), true, needzero)
    return s
}
```

3.2.1 管理

研究提取算法前，须先了解 heap 如何管理内存块。

相关字段中，以 free 和 busy 表达“自由可用”和“已被使用”两种状态。对于常用，也就是少于 128 页的内存块，以数组存储。其他更大内存块，全部放入树结构有序存储。

mheap.go

```
type mheap struct {
    free      [_MaxMHeapList]mSpanList   // free lists of given length up to _MaxMHeapList
    freelarge mTreap                    // free treap of length >= _MaxMHeapList

    busy      [_MaxMHeapList]mSpanList   // busy lists of large spans of given length
    busylarge mSpanList                 // busy lists of large spans length >= _MaxMHeapList
}
```

malloc.go

```
_MaxMHeapList   = 1 << (20 - _PageShift)   // 128
```

数组 free 以页数为索引，元素是由多个相同页数的内存块所构成链表。

mheap.go

```
type mSpanList struct {
    first *mspan              // first span in list, or nil if none
    last  *mspan              // last span in list, or nil if none
}
```

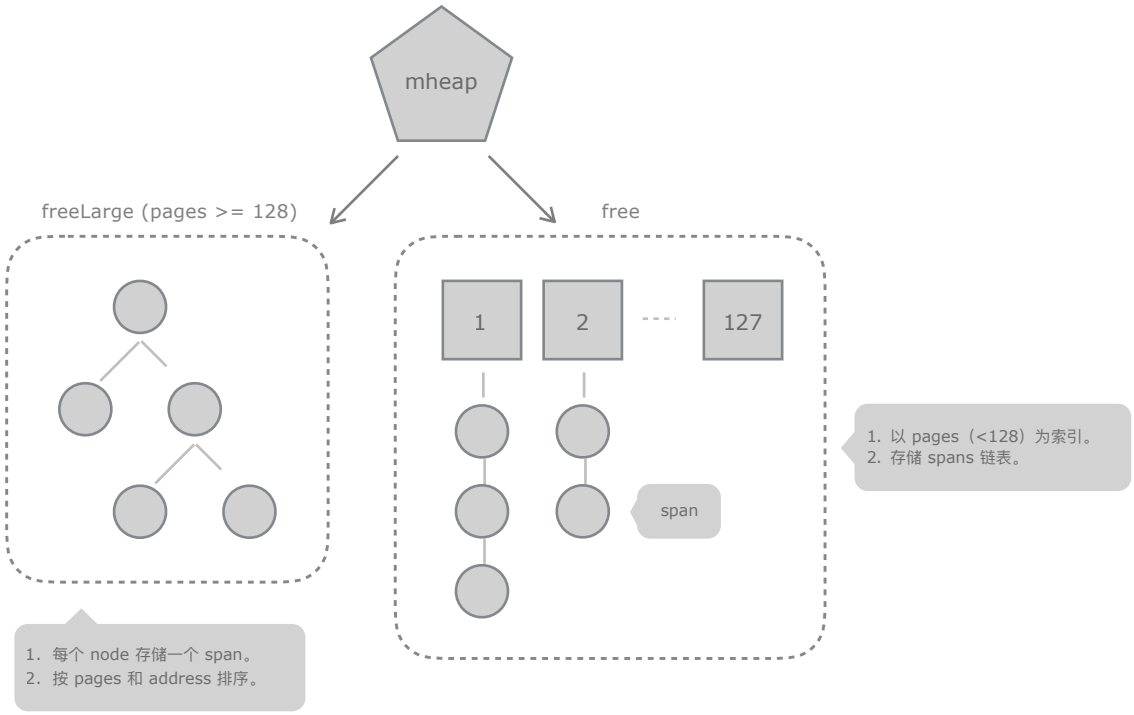
```
type mspan struct {
    next *mspan             // next span in list, or nil if none
    prev *mspan             // previous span in list, or nil if none
}
```

树堆 freelarge 是二叉搜索树，以内存块页数和起始地址为排序条件。可返回大小适中，且地址靠前的内存块，用来替代早期 best fit 简易算法实现。

mgclarge.go

```
// Each treapNode holds a single span. The treap is sorted by page size
// and for spans of the same size a secondary sort based on start address
// is done.

// Spans are returned based on a best fit algorithm and for spans of the same
// size the one at the lowest address is selected.
```

结构示意图：



相比以前版本，调整后的结构设计目的更加明确。以页数为索引的链表数组，拥有最佳访问效率，且提取和添加都极为方便。至于更大的块，因其数量不多，树堆结构足以保障，同时实现了原定算法意图。

3.2.2 获取

了解管理背景后，我们继续算法细节研究。

mheap.go

```
func (h *mheap) alloc(npage uintptr, spanclass spanClass, large bool, needzero bool) *mspan {
    systemstack(func() {
        s = h.alloc_m(npage, spanclass, large)
    })

    return s
}
```

```
func (h *mheap) alloc_m(npage uintptr, spanclass spanClass, large bool) *mspan {

    // 从 heap 提取内存块。
    s := h.allocSpanLocked(npage, &memstats.heap_inuse)

    if s != nil {
       if large {
            // 放入 busy 链表数组或树堆。
            if s.npages < uintptr(len(h.busy)) {
                h.busy[s.npages].insertBack(s)
            } else {
                h.busylarge.insertBack(s)
            }
        }
    }

    return s
}
```

内存块被提取后，放入已使用列表，这没什么好说的。真正核心内容是下面这个方法。

首先，从指定页数起遍历 free 数组。为何不用页数为索引直接访问呢？要知道与条件相符的数组元素可能为空，不管是在初始化时，还是运行过程中，其对应链表都可能没有可用内存块。此时，最佳做法是继续尝试页数更多的链表，而非向操作系统申请新内存。

> 15 页的没有，那么就去 16、17 页的链表里找。

如果 free 里没有返回，那么继续查找 freelarge。再不济，就向操作系统申请新内存。总之，尽量在已有内存块里返回容量最接近的那个。

mheap.go

```
func (h *mheap) allocSpanLocked(npage uintptr, stat *uint64) *mspan {

    // 从指定页数起，遍历 free 链表数组。
    for i := int(npage); i < len(h.free); i++ {
        // 从链表头部提取内存块。
        list = &h.free[i]
        if !list.isEmpty() {
            s = list.first
            list.remove(s)
            goto HaveSpan
        }
    }

    // 在 freelarge 里查找。
    s = h.allocLarge(npage)
    if s == nil {
        // 扩张，向操作系统申请新内存（最少 1 MB, 128 pages）。
        if !h.grow(npage) {
            return nil
        }
        // 新申请内存放在 freelarge，提取。
        s = h.allocLarge(npage)
   }

HaveSpan:

   // ... 分割多余内存，参考下节 ...

   return s
}
```

3.2.3 分割

从提取过程看，所返回内存块大小可能超出预期。此时，需对其作分割处理，避免内存浪费，毕竟此处是以页为单位。也正因为分割的缘故，所以要优先返回大小合适的内存块，尽量避免碎片化。

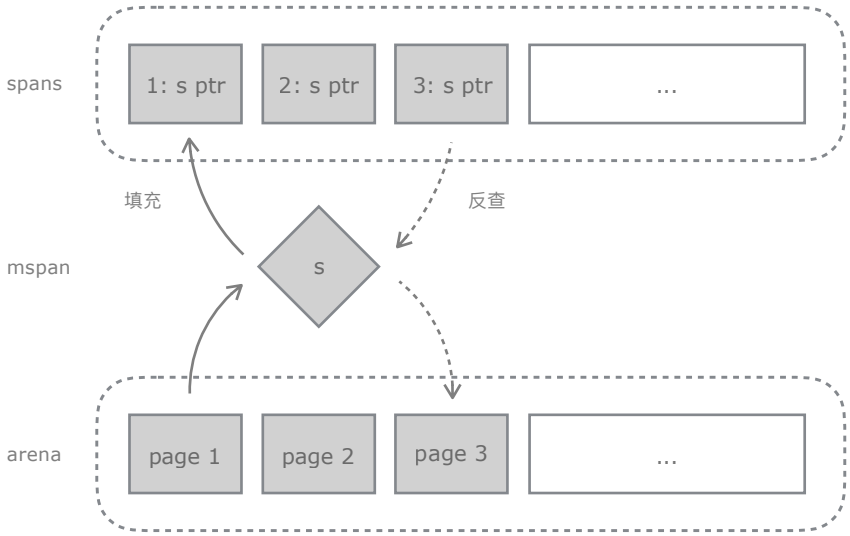
mheap.go

```
func (h *mheap) allocSpanLocked(npage uintptr, stat *uint64) *mspan {

    // 提取内存块，详情见上节 ...

HaveSpan:

    // 如果页数超出预期，则进行分割。
    if s.npages > npage {
```

```
        // 创建新 mspan 对象，用于管理分割下来的内存块。
        t := (*mspan)(h.spanalloc.alloc())
        t.init(s.base()+npage<<_PageShift, s.npages-npage)
        s.npages = npage

        // 计算分割下来的内存块索引，并修改 heap.spans 反查表内容。
        p := (t.base() - h.arena_start) >> _PageShift
        if p > 0 {
            h.spans[p-1] = s                                // s_spans 尾部边界
        }
        h.spans[p] = t                                      // t_spans 头部边界
        h.spans[p+t.npages-1] = t                           // t_spans 尾部边界

        // 将分割下来的内存块放回 heap，会引发内存块合并操作。
        h.freeSpanLocked(t, false, false, s.unusedsince)
    }

    // 计算索引号，并使用 span 指针填充 heap.spans 反查表。
    p := (s.base() - h.arena_start) >> _PageShift
    for n := uintptr(0); n < npage; n++ {
        h.spans[p+n] = s
    }

    return s
}
```

在这里，我们看到了 heap.spans 所存储内容，就是内存块管理对象 mspan 指针。

heap.spans 内容示意图：



3.2.4 合并

不管是分割所余，还是垃圾回收，只要将内存块放回 heap，都将引发合并操作。

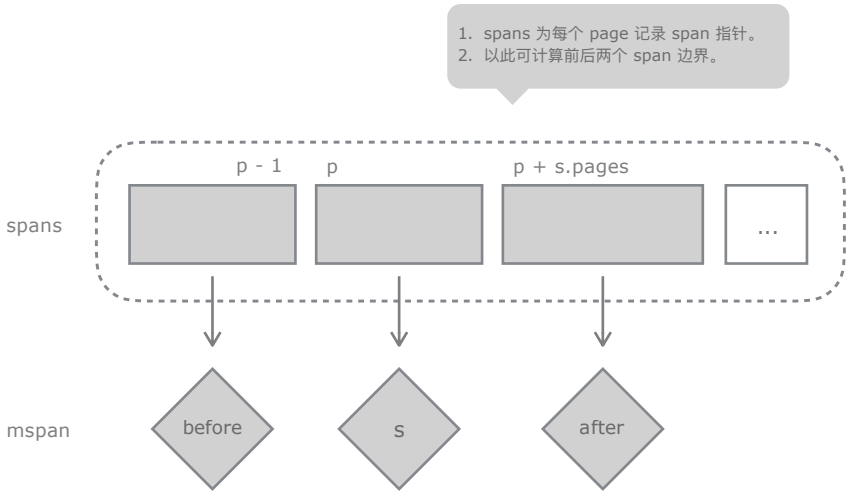
通过 spans 反查表，可访问左右地址相邻内存块。如它们也处于自由状态，就进行合并操作。如此，可获得更大自由空间，能适应更多请求，并减少碎片。

mheap.go

```
func (h *mheap) freeSpanLocked(s *mspan, acctinuse, acctidle bool, unusedsince int64) {

    // 先将当前 span 从 busy 里移除。
    s.state = _MSpanFree
    if s.inList() {
        h.busyList(s.npages).remove(s)
    }

    // 计算 span 索引号。
    p := (s.base() - h.arena_start) >> _PageShift
    if p > 0 {
        // 通过 heap.spans 查看左侧相邻 span。
        before := h.spans[p-1]

        // 如果左侧 span 存在，且处于 free 状态，则合并到当前 span。
        if before != nil && before.state == _MSpanFree {
            // 调整当前 span 属性。
            s.startAddr = before.startAddr
            s.npages += before.npages

            // 调整索引号，修改 heap.spans 内容。
            p -= before.npages
            h.spans[p] = s

            // 将左侧 span 从相关列表移除。
            if h.isLargeSpan(before.npages) {
                h.freelarge.removeSpan(before)
            } else {
                h.freeList(before.npages).remove(before)
            }
        }
    }

    // 检查右侧 span。
    if (p + s.npages) < uintptr(len(h.spans)) {
        // 右侧 span。
        after := h.spans[p+s.npages]

        // 如果存在，且处于 free 状态，合并。
        if after != nil && after.state == _MSpanFree {
            // 调整当前 span 属性。
            s.npages += after.npages
```

```
            // 修改 heap.spans 内容。
            h.spans[p+s.npages-1] = s

            // 将右侧 span 从相关列表移除。
            if h.isLargeSpan(after.npages) {
                h.freelarge.removeSpan(after)
            } else {
                h.freeList(after.npages).remove(after)
            }
        }
    }

    // 将 span（或者合并后的 span）放入合适的列表。
    if h.isLargeSpan(s.npages) {
        h.freelarge.insert(s)
    } else {
        h.freeList(s.npages).insert(s)
    }
}
```

合并状态示意图：



3.2.5 申请

仅在现有内存块都不足以满足请求时，才向操作系统发出申请。

基于性能考虑，每次都申请足够大的内存空间。无需担心此举会造成浪费，因为所申请内存此刻尚未使用，操作系统不会立即为其分配物理内存。

mheap.go

```
func (h *mheap) grow(npage uintptr) bool {
```

```
    // Ask for a big chunk, to reduce the number of mappings
    // the operating system needs to track; also amortizes
    // the overhead of an operating system mapping.

    // 总是 64 KB 倍数，且不能少于 1 MB。
    npage = round(npage, (64<<10)/_PageSize)
    ask := npage << _PageShift
    if ask < _HeapAllocChunk {
        ask = _HeapAllocChunk
    }

    // 向操作系统申请内存。
    v := h.sysAlloc(ask)

    // 创建 mspan 进行管理。
    s := (*mspan)(h.spanalloc.alloc())
    s.init(uintptr(v), ask>>_PageShift)

    // 向 heap.spans 填充 mspan 指针。
    p := (s.base() - h.arena_start) >> _PageShift
    for i := p; i < p+s.npages; i++ {
        h.spans[i] = s
    }

    // 放回 heap，尝试合并操作。
    h.freeSpanLocked(s, false, true, 0)
    return true
}
```

malloc.go

```
_HeapAllocChunk = 1 << 20               // Chunk size for heap growth
```

完整可工作内存不仅仅是 arena，还包括 bitmap 和 spans。如果没有关联元数据，运行时相关机制根本无法工作。所以，得同步扩张这两部分内存。

malloc.go

```
func (h *mheap) sysAlloc(n uintptr) unsafe.Pointer {

    // 范围检查，确保 arena 还有足够空间。
    if n <= h.arena_end-h.arena_alloc {

        // 注意使用 arena_alloc 作为分配起始地址。
        p := h.arena_alloc
        sysMap(unsafe.Pointer(p), n, h.arena_reserved, &memstats.heap_sys)
        h.arena_alloc += n

        // 为 heap.spans 和 heap.bitmap 同步分配内存。
        if h.arena_alloc > h.arena_used {
```

```
            h.setArenaUsed(h.arena_alloc, true)
        }

        return unsafe.Pointer(p)
    }
}
```

mheap.go

```
func (h *mheap) setArenaUsed(arena_used uintptr, racemap bool) {

    // 同步分配 heap.bitmap、spans 内存。
    h.mapBits(arena_used)
    h.mapSpans(arena_used)

    h.arena_used = arena_used
}
```

相关字段：

mheap.arena_alloc：下次分配地址。

mheap.arena_used：合法地址边界，关联 spans、bitmap 内存已分配。

至于 sysMap，无非是封装过的系统调用，并无多少秘密可言。

mem_linux.go

```
func sysMap(v unsafe.Pointer, n uintptr, reserved bool, sysStat *uint64) {
    p, err := mmap(v, n, _PROT_READ|_PROT_WRITE, _MAP_ANON|_MAP_FIXED|_MAP_PRIVATE, -1, 0)
}
```

mem_windows.go

```
func sysMap(v unsafe.Pointer, n uintptr, reserved bool, sysStat *uint64) {
    p := stdcall4(_VirtualAlloc, uintptr(v), n, _MEM_COMMIT, _PAGE_READWRITE)
}
```

3.3 小对象

数量众多的小对象才是内存分配器重心所在。这其中涉及多级部件如何协作，如何为不同大小规格对象提供无锁分配，如何优化性能等等内容。

3.3.1 缓存

前文提及，三级部件中，缓存（cache）直接为当前线程提供小对象内存分配。如此，我们从数据结构开始。

缓存部件使用数组（alloc）存储多个内存块，每个代表一种大小规格（size class）。

mcache.go

```
type mcache struct {
    alloc [numSpanClasses]*mspan     // spans to allocate from, indexed by spanClass
}
```

但是，不能以大小规格为索引直接访问数组。每个请求内存的对象，根据其是否包含指针可分作 scan 和 noscan 两类。指针决定了垃圾回收时是否需要深度扫描，据此分开处理有诸多好处。

数组容量是预期的两倍，就是因为每种规格有 noscan 和 scan 两个内存块。

mheap.go

```
numSpanClasses = _NumSizeClasses << 1

type mspan struct {
    spanclass   spanClass           // size class and noscan (uint8)
    elemsize    uintptr             // computed from sizeclass or from npages
}
```

大小规格和数组索引之间通过 spanclass 转换。算法非常简单：将规格左移一个二进制位，用来存储扫描标记。

```
spanclass = sizeclass << 1 | noscan
```

比如说，size class 3 就有如下两种索引：

```
0b11 << 1 | 0    =   0b110    =   6      ; scan
0b11 << 1 | 1    =   0b111    =   7      ; noscan
```

---

mheap.go

```
type spanClass uint8
```

```
func makeSpanClass(sizeclass uint8, noscan bool) spanClass {
    return spanClass(sizeclass<<1) | spanClass(bool2int(noscan))
}

func (sc spanClass) sizeclass() int8 {
    return int8(sc >> 1)
}
```

接下来，无非是根据对象大小获知规格，进而加入扫描标记，计算出数组实际索引。

malloc.go

```
func mallocgc(size uintptr, typ *_type, needzero bool) unsafe.Pointer {

    c := gomcache()
    noscan := typ == nil || typ.kind & kindNoPointers != 0

    if size <= maxSmallSize {

        // 查表，返回大小规格。
        if size <= smallSizeMax - 8 {
            sizeclass = size_to_class8[size...]
        } else {
            sizeclass = size_to_class128[size...]
        }

        // 计算数组索引，返回规格对应的 span。
        spc := makeSpanClass(sizeclass, noscan)
        span := c.alloc[spc]

        // 从 span 内提取 object。
        v := nextFreeFast(span)
        if v == 0 {
            v, span, shouldhelpgc = c.nextFree(spc)
        }

        x = unsafe.Pointer(v)
    }

    return x
}
```

3.3.2 提取

所谓切分，只是按规格大小和偏移位置计算出分段内存地址，并非真的切成碎块。

管理对象用位图（allocBits）标记内存使用状态。同时，为避免扫描整个位图，还采用专门字段（freeindex）记录上次分配位置，以此为顺序分配起点。

mheap.go

```
type mspan struct {
    nelems     uintptr               // number of object in the span.

    allocBits  *gcBits
    freeindex  uintptr

    allocCache uint64
}

type gcBits uint8
```

即便没有细节，我们也能理解下面的算法过程。

因为每个 span 仅为一种规格服务，那么可将其视作 object 数组。只要通过位图找到可用索引号，就能计算出实际内存地址。

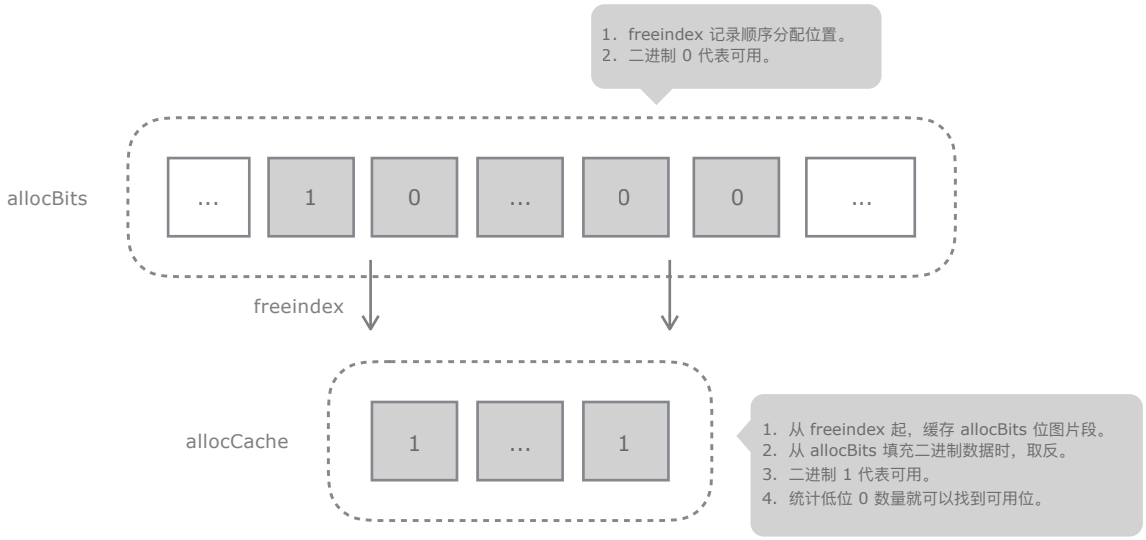
malloc.go

```
func (c *mcache) nextFree(spc spanClass) (v gclinkptr, s *mspan, shouldhelpgc bool) {

    // 获取下一个可用 object 索引。
    s = c.alloc[spc]
    freeIndex := s.nextFreeIndex()

    // 如果没有剩余空间,
    if freeIndex == s.nelems {

        // 从 central 扩容.
        systemstack(func() {
            c.refill(spc)
        })

        // 扩容后，重新获取。
        s = c.alloc[spc]
        freeIndex = s.nextFreeIndex()
    }

    // 根据序号计算内存地址。
    v = gclinkptr(freeIndex * s.elemsize + s.base())
}
```

关键在于 freeindex 如何工作。

位图虽便于记录状态，但其访问效率不算太好。为此，采用 64 位整数（allocCache）缓存位图片段来提升工作效率，就像 CPU cache 机制。要知道，整数无论是放入寄存器，还是多字节二进制位运算都要比位图方便、高效。

工作示意图：



mbitmap.go

```
// nextFreeIndex returns the index of the next free object in s at
// or after s.freeindex.

func (s *mspan) nextFreeIndex() uintptr {
    sfreeindex := s.freeindex
    snelems := s.nelems

    // freeindex 指向尾部，没有可用空间。
    if sfreeindex == snelems {
        return sfreeindex
    }

    // 获取缓存中第一个可用位。
    aCache := s.allocCache
    bitIndex := sys.Ctz64(aCache)

    // 无有效位，刷新缓存。（后面填充的数据未必有可用位，循环直到有可用位为止）
    for bitIndex == 64 {

        // 从位图填充后 64 位数据。
        sfreeindex = (sfreeindex + 64) &^ (64 - 1)
        whichByte := sfreeindex / 8
        s.refillAllocCache(whichByte)

        // 重新计算。
        aCache = s.allocCache
        bitIndex = sys.Ctz64(aCache)
```

```
    }

    // 计算基于位图的有效位索引。
    result := sfreeindex + uintptr(bitIndex)

    // 调整缓存数据，同时修正 freeindex 值。
    s.allocCache >>= uint(bitIndex + 1)
    sfreeindex = result + 1
    s.freeindex = sfreeindex

    return result
}
```

从位图填充数据到缓存，会进行取反操作。也就是说，位图 0 代表可用位，而缓存则是 1 代表可用位。举例来说：

allocCache: 11...11000100
H L

1. 统计（Ctz64）低位 0 数量，确定第一个可用位序号（bitIndex = 2）。
2. 计算基于位图的实际索引（result = freeindex + bitIndex）。
3. 缓存右移，剔除已检查片段（bitIndex + 1，含本次）。
4. 同步修正 freeindex 值（result + 1）。

allocCache: 00011...11000 // 右移 3 位
H L

通过位移剔除已检查片段后，每次只需低位统计即可，简单高效。也正因为如此，才需要在填充时取反。比如，0000 不管如何位移，都不会有结果，但 1111 右移一位就是 0111。

另外，垃圾回收会重置 allocCache 和 freeindex 值。

优先使用快速版本。没有无关调用，不填充，不扩容，一切为性能让路。

malloc.go

```
func nextFreeFast(s *mspan) gclinkptr {

    // 基于缓存查找可用位。
    theBit := sys.Ctz64(s.allocCache)     // Is there a free object in the allocCache?

    // 如果有，
    if theBit < 64 {
        // 计算基于位图的索引。
        result := s.freeindex + uintptr(theBit)
```

```
        if result < s.nelems {
            // 调整相关数据。
            freeidx := result + 1
            s.allocCache >>= uint(theBit + 1)
            s.freeindex = freeidx

            // 返回可用位内存地址。
            return gclinkptr(result * s.elemsize + s.base())
        }
    }
}
```

3.3.3 扩容

扩容前，将当前没有剩余空间的内存块相关状态解除，以便垃圾回收器进行扫描和回收。随后，从中间部件提取新内存块放回数组。

mcache.go

```
func (c *mcache) refill(spc spanClass) {

    // 解除当前 span 相关状态。
    s := c.alloc[spc]
    if s != &emptymspan {
        s.incache = false
    }

    // 从 central 获取新 span 放回数组。
    s = mheap_.central[spc].mcentral.cacheSpan()
    c.alloc[spc] = s
}
```

与缓存部件类似，中间部件同样有 scan 和 noscan 之分，并由堆进行统一管理。

mheap.go

```
type mheap struct {
    central [numSpanClasses]struct {
        mcentral mcentral
    }
}
```

中间部件用两个链表管理所有经手的内存块。

其中，empty 表达“无法使用”状态。存储没有剩余空间，或被移交给缓存的内存块。对应的 noempty 则存储有剩余空间，可提供服务的内存块。

> 垃圾回收后，并不表示 span 内存空间全部收回。但这不妨碍将它交给其他线程使用。

mcentral.go

```
type mcentral struct {
    spanclass spanClass
    nonempty  mSpanList     // list of spans with a free object, ie a nonempty free list
    empty     mSpanList     // list of spans with no free objects (or cached in an mcache)
}
```

在这里，需要和垃圾回收器进行一些交互。每个内存块都有与垃圾清理相关的状态，这关系到我们该如何对其复用。

> 垃圾回收器使用累增计数器（heap.sweepgen）来表达代龄。每次清理操作，该计数器 +2。
>
> span.sweepgen =
>
> sweepgen - 2 : 垃圾已标记，需要清理。
>
> sweepgen - 1 : 正在清理。
>
> sweepgen : 清理完成，可以使用。

mheap.go

```
type mspan struct {

    // sweep generation:
    // if sweepgen == h->sweepgen - 2, the span needs sweeping
    // if sweepgen == h->sweepgen - 1, the span is currently being swept
    // if sweepgen == h->sweepgen, the span is swept and ready to use
    // h->sweepgen is incremented by 2 after every GC

    sweepgen    uint32
}
```

首先检查 nomepty 列表中有剩余空间的内存块。鉴于垃圾回收器并发扫描和清理，所以在移交给缓存部件前，应先完成清理操作，以获取更多可用空间。

如 noempty 没有返回，那么接着检查 empty 列表。要知道 empty 里的内存块或许已被垃圾标记，只需清理就有空间可供使用。只有上述列表都没有结果时，才向堆申请内存。

mcentral.go

```
func (c *mcentral) cacheSpan() *mspan {

    lock(&c.lock)
    sg := mheap_.sweepgen

retry:
    // 遍历有剩余空间的内存块链表。
    for s = c.nonempty.first; s != nil; s = s.next {

        // 需要清理。
        // 修改代龄状态，先转移到 empty 列表。
        if s.sweepgen == sg-2 && atomic.Cas(&s.sweepgen, sg-2, sg-1) {
            c.nonempty.remove(s)
            c.empty.insertBack(s)
            s.sweep(true)
            goto havespan
        }

        // 如果正在清理，跳过。（可能正被 bgSweep 或其他 cacheSpan 处理）
        if s.sweepgen == sg-1 {
            continue
        }

        // 已经清理过的，直接使用。
        c.nonempty.remove(s)
        c.empty.insertBack(s)
        goto havespan
    }

    // 遍历没有剩余空间的列表，尝试清理出可用空间。
    for s = c.empty.first; s != nil; s = s.next {

        // 需要清理。
        if s.sweepgen == sg-2 && atomic.Cas(&s.sweepgen, sg-2, sg-1) {
            // 转移到 empty 列表尾部。
            c.empty.remove(s)
            c.empty.insertBack(s)
            s.sweep(true)

            // 检查相关状态，查看是否有可用空间。
            freeIndex := s.nextFreeIndex()
            if freeIndex != s.nelems {
                s.freeindex = freeIndex
                goto havespan
            }

            // 清理后依然没有剩余空间。
            // 重试 noempty，没必要在这折腾。
            goto retry
        }

        // 正在被清理，跳过。
```

```
        if s.sweepgen == sg-1 {
            continue
        }

        // 被移交给缓存，或清理后没有剩余空间的，都追加到 empty 尾部，它们 sweepgen == sg。
        // 所以遇到这样的内存块时，表示后面无需再检查，直接跳出循环。
        break
    }

    // 最后，从 heap 扩容，并直接返回。
    s = c.grow()
    c.empty.insertBack(s)

havespan:

    // 设置 span 状态。
    s.incache = true
    return s
}
```

> 显然，对 empty 的检查采取了谨慎态度。一旦势头不对，立即回过头去重试 noempty 列表。早期版本里，central 会进行切分操作，构建 object freelist，现在取消了。

至于中间部件自身扩容操作，与大对象内存分配如出一辙，无庸赘述。

mcentral.go

```
func (c *mcentral) grow() *mspan {
    s := mheap_.alloc(npages, c.spanclass, false, true)
    return s
}
```

3.3.4 微小对象

微小对象（tiny）长度小于 16 字节，最常见的就是小字符串。如果将多个微小对象组合起来，用单块内存（object）存储，可有效减少内存浪费。

malloc.go

```
maxTinySize    = _TinySize

_TinySize      = 16
_TinySizeClass = int8(2)
```

缓存部件按指定规格（tiny size class）取出一块内存，专用于微小对象分配。如果容量不足，则重新提取一块。

mcache.go

```
type mcache struct {
    tiny             uintptr
    tinyoffset       uintptr
}
```

当然，因为垃圾回收的缘故，用来组合的微小对象不能包含指针。直到存储单元里所有微小对象都不可达时，该内存才能回收。

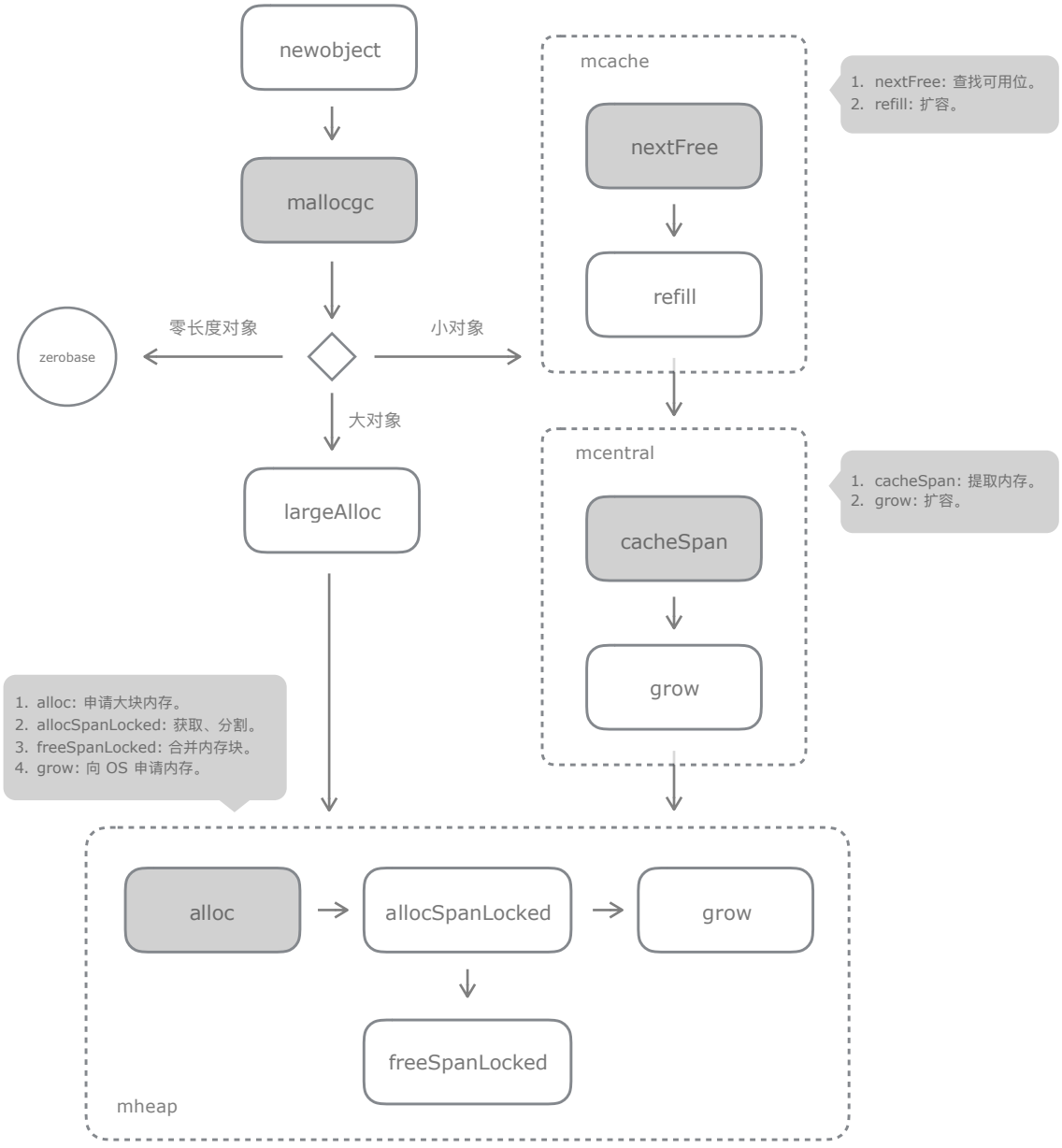
通过偏移位置（tinyoffset）可判断剩余空间是否满足需求。如果可以，以此计算并返回内存地址。不足，则提取新内存块，返回起始地址便可。最后，对比新旧两块内存，留下剩余空间更大的那块。

malloc.go

```
func mallocgc(size uintptr, typ *_type, needzero bool) unsafe.Pointer {

    if size <= maxSmallSize {

        // 检查是否使用微小对象分配器。
        if noscan && size < maxTinySize {

            // 当前内存块（cache.tiny）分配位置。
            off := c.tinyoffset

            // 如果剩余空间足以分配，
            if off + size <= maxTinySize && c.tiny != 0 {
                // 计算内存地址，调整下次分配位置。
                x = unsafe.Pointer(c.tiny + off)
                c.tinyoffset = off + size
                return x
            }

            // 如果剩余空间不足，则从 cache 里新取一块 object。
            span := c.alloc[tinySpanClass]
            v := nextFreeFast(span)
            if v == 0 {
                v, _, shouldhelpgc = c.nextFree(tinySpanClass)
            }

            // 初始化新 object，返回起始地址即可。
            x = unsafe.Pointer(v)
            (*[2]uint64)(x)[0] = 0
            (*[2]uint64)(x)[1] = 0
```

```
            // 对比新旧两块 tiny 内存，留下剩余空间更大的那个。
            if size < c.tinyoffset || c.tiny == 0 {
                c.tiny = uintptr(x)
                c.tinyoffset = size
            }
        }
    }

    return x
}
```

快速导览：



4. 回收

内存回收由垃圾回收器引发，内存分配器执行。

4.1 缓存

缓存部件是垃圾回收作根（root）对象。在扫描和标记之前，它们所持有的内存块被重新放回中间部件。如此，闲置内存就可调配给其他缓存部件使用，避免浪费。

mgcmark.go

```
func markroot(gcw *gcWork, i uint32) {
    flushmcache(...)
}
```

mstats.go

```
func flushmcache(i int) {
    p := allp[i]
    c := p.mcache
    c.releaseAll()
}
```

扩容时，cache 每次从 central 提取一个 span。但不保证会将该 span 内的 object 全部分配出去。如此，剩余内存会被闲置。将包含剩余空间的 span 重新放回 central，就可转交给其他 cache，从而平衡资源使用，避免内存浪费。

不用担心一个 span 被多个 cache 使用会有什么影响。好比一板感冒药，扣出几粒胶囊给张三，余下的大可给了李四、王五，没有另买两盒的道理。分配操作只是从内存块中“切”出一个片段交给某个对象，剩余空间后续如何，与之无关。

---

mcache.go

```
func (c *mcache) releaseAll() {

    // 遍历内存块数组。
    for i := range c.alloc {
        s := c.alloc[i]

        // 如果是合法内存块，上交给 central。
        if s != &emptymspan {
            mheap_.central[i].mcentral.uncacheSpan(s)
            c.alloc[i] = &emptymspan
        }
```

```
    }

    // 释放微小对象分配内存。
    c.tiny = 0
    c.tinyoffset = 0
}
```

---

emptymspan 用于占位，并不持有内存。

---

mcache.go

```
// dummy MSpan that contains no free objects.
var emptymspan mspan
```

收归中间部件时，根据是否有剩余空间决定存储列表。上述操作是将所有内存块上交，并不仅仅是有剩余空间的。兴许某个内存块正好内存耗尽，缓存部件尚未及扩容。

mcentral.go

```
func (c *mcentral) uncacheSpan(s *mspan) {

    // 解除相关属性。
    s.incache = false

    // 根据剩余空间决定存储列表。
    n := cap - int32(s.allocCount)
    if n > 0 {
        c.empty.remove(s)
        c.nonempty.insert(s)
    }
}
```

4.2 清理

垃圾标记完成后，以内存块（span）为单位进行清理操作。

所谓清理，并非挨个检查所有内存单元。实际上，内存块除分配位图外，还有个一摸一样的垃圾标记位图（gcmarkBits）。垃圾回收器以此标记出可回收，也就是可重新使用的内存位置。

如此，只需将标记位图数据“复制”给分配位图就可实现清理目的。至于内存单元里的遗留数据清除与否，则是分配操作要考虑的。

mheap.go

```
type mspan struct {
    gcmarkBits *gcBits         // 标记位图
    allocBits  *gcBits         // 分配位图
}
```

mgcsweep.go

```
func (s *mspan) sweep(preserve bool) bool {

    spc := s.spanclass

    // 基于标记位图统计已分配数量（不含可回收部分）。
    nalloc := uint16(s.countAlloc())

    // 如果是大对象（size class 0），那么稍后将其归还给 heap。
    if spc.sizeclass() == 0 && nalloc == 0 {
        freeToHeap = true
    }

    // 计算本次回收数量（标记前分配数量 - 标记后分配数量）。
    nfreed := s.allocCount - nalloc

    // 调整内存块属性。
    s.allocCount = nalloc
    s.freeindex = 0

    // 将标记位图直接当作分配位图使用，便可实现“复制”。
    s.allocBits = s.gcmarkBits
    s.gcmarkBits = newMarkBits(s.nelems)

    // 放回 central 或 heap。
    if nfreed > 0 && spc.sizeclass() != 0 {
        res = mheap_.central[spc].mcentral.freeSpan(s, preserve, wasempty)
    } else if freeToHeap {
        mheap_.freeSpan(s, 1)
    }
}
```

接下来，由对应部件完成后续回收操作。

mcentral.go

```
func (c *mcentral) freeSpan(s *mspan, preserve bool, wasempty bool) bool {

    // 该参数决定是否调整存储列表。
    if preserve {
        return false
    }
```

```
    // 放入合适的存储列表。
    if wasempty {
        c.empty.remove(s)
        c.nonempty.insert(s)
    }

    // 更新垃圾回收代龄。
    atomic.Store(&s.sweepgen, mheap_.sweepgen)

    // 如果还有分配，那么直接返回。
    if s.allocCount != 0 {
        return false
    }

    // 如果内存全部收回，上交给 heap。
    c.nonempty.remove(s)
    mheap_.freeSpan(s, 0)

    return true
}
```

如果中间部件持有的内存块收回其全部空间，则将其上交给堆，以便调剂给其他中间部件使用。这就是第二级资源平衡，使得不同大小规格的请求能充分利用已有内存。

假设 central 5 持有一堆闲置内存块，而 6、7 等因内存耗尽发出扩容请求，那么 heap 就可能因此向操作系统发出新的申请。如此下去，就会有更多内存被闲置和浪费。让 central 5 尽快上交闲置内存，以便转给 6、7 使用才是合理做法。

为什么上交的是完整收回空间的 span ？因为每个 span 仅服务于一种 size class，如果将剩余空间转给其他 central，就必须对其进行切分。这么做将导致内存碎片化，最终得不偿失。

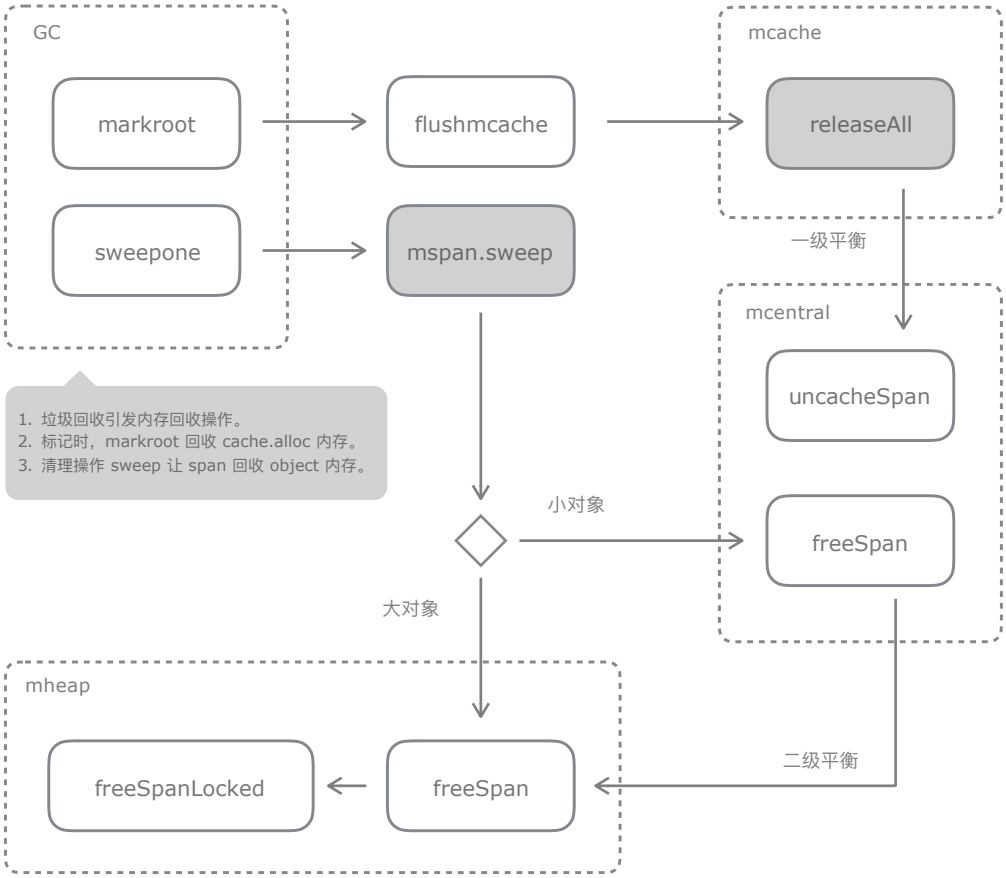
---

mheap.go

```
func (h *mheap) freeSpan(s *mspan, acct int32) {
    systemstack(func() {
        // 收回，尝试合并。
        h.freeSpanLocked(s, true, true, 0)
    })
}
```

回收操作到此为止，相关内存经平衡后放归不同部件继续提供复用服务。至于物理内存释放，则另有玄机。

快速导览：



5. 释放

运行时初始化时，创建专门线程用于后台监控，执行定期任务。其中包括强制垃圾回收、物理内存释放等。

proc.go

```
func main() {
    systemstack(func() {
        newm(sysmon, nil)
    })
}
```

默认设置下，闲置超过 5 分钟的内存块将被释放物理内存。如打开命令行开关，则时间阈值可缩短为 20 毫秒，更利于调试和观察。

proc.go

```
func sysmon() {

    // If a heap span goes unused for 5 minutes after a garbage collection,
    // we hand it back to the operating system.

    scavengelimit := int64(5 * 60 * 1e9)   // 5 分钟。
    lastscavenge := nanotime()             // 最后执行时间。
    nscavenge := 0                         // 执行次数统计。

    for {
        usleep(delay)
        now := nanotime()

        // 检查并释放物理内存。
        if lastscavenge+scavengelimit/2 < now {
            mheap_.scavenge(int32(nscavenge), uint64(now), uint64(scavengelimit))
            lastscavenge = now
            nscavenge++
        }
    }
}
```

5.1 闲置

所谓闲置内存是指由堆管理，尚未被中间部件或大对象使用的内存块。或是分割余留，或是回收上交。如长时间不使用，那么应该释放其物理内存，以节约系统资源。

内存管理对象有两个相关计数器，分别存储闲置起始时间和释放页数。这就是判断闲置和释放的基本条件。

mheap.go

```
type mspan struct {
    unusedsince int64        // first time spotted by gc in mspanfree state
    nprereleased  uintptr      // number of pages released to the os
}
```

计数器在内存块获取和放回操作中被重置。

mheap.go

```
func (h *mheap) allocSpanLocked(npage uintptr, stat *uint64) *mspan {

    // 如物理内存被释放，则重新分配。
    if s.nprereleased > 0 {
```

```
        sysUsed(unsafe.Pointer(s.base()), s.npages<<_PageShift)
        s.npreleased = 0
    }

    s.unusedsince = 0
}
```

```
func (h *mheap) freeSpanLocked(s *mspan, acctinuse, acctidle bool, unusedsince int64) {

    s.unusedsince = unusedsince
    if unusedsince == 0 {
        s.unusedsince = nanotime()
    }

    s.npreleased = 0
}
```

注意，存在局部释放的情况。

假设内存块已释放物理内存，此时 npreleased == npages。稍后与其他内存块合并，那么合并后 npreleased < npages。

5.2 释放

在大对象内存分配一节里，我们分析了堆如何管理内存块。少于 128 页的可用内存块放置于 free 链表数组，更大的则以树堆 freelarge 存储。释放操作所针对目标就是这两个列表。

参数 now 作为超时判断基准时间，limit 则是超时阈值（默认 5 分钟）。

---

mheap.go

```
func (h *mheap) scavenge(k int32, now, limit uint64) {

    for i := 0; i < len(h.free); i++ {
        sumreleased += scavengelist(&h.free[i], now, limit)
    }

    sumreleased += scavengetreap(h.freelarge.treap, now, limit)

    // 输出统计结果。
    if debug.gctrace > 0 {
        if sumreleased > 0 {
            print("scvg", k, ": ", sumreleased>>20, " MB released\n")
```

```
        }
    }
}
```

遍历列表，用计数器与基准条件比对。

mheap.go

```
func scavengelist(list *mSpanList, now, limit uint64) uintptr {

    // 跳过空链表。
    if list.isEmpty() {
        return 0
    }

    // 遍历链表内的 span。
    for s := list.first; s != nil; s = s.next {

        // 如闲置时间小于 limit，或已被全部释放，跳过。
        if (now-uint64(s.unusedsince)) <= limit || s.npreleased == s.npages {
            continue
        }

        // 统计要释放的空间大小。
        start := s.base()
        end := start + s.npages<<_PageShift
        len := end - start

        released := len - (s.npreleased << _PageShift)
        sumreleased += released

        // 设置释放计数。
        s.npreleased = len >> _PageShift

        // 通知操作系统释放物理内存（整块内存）。
        sysUnused(unsafe.Pointer(start), len)
    }

    return sumreleased
}
```

递归树堆全部节点，相关操作基本一致。

mheap.go

```
func scavengeTreapNode(t *treapNode, now, limit uint64) uintptr {
    s := t.spanKey

    // 如闲置时间超出 limit，且有物理内存未曾释放。
    if (now-uint64(s.unusedsince)) > limit && s.npreleased != s.npages {
```

```
            start := s.base()
            end := start + s.npages<<_PageShift
            len := end - start

            released := len - (s.npreleased << _PageShift)
            sumreleased += released

            // 设置释放计数。
            s.npreleased = len >> _PageShift

            // 通知操作系统释放物理内存（整块内存）。
            sysUnused(unsafe.Pointer(start), len)
        }

        return sumreleased
    }
```

到此，我们获知哪些内存块可被释放。至于如何释放其物理内存，则因操作系统而异。

UNIX-like 系统以 madvise 建议内核解除物理内存映射。从而在保留虚拟内存的情况下，达到释放物理内存的目的。当这些内存被重新使用时，由内核自动补齐所需物理内存。

> 注意，madvise 仅是建议，没有规定必须执行或立即执行。当物理内存充足时，某些内核会忽略该建议，或延后执行。有鉴于此，相关统计数据未必准确，仅做参考。

mem_linux.go

```
func sysUnused(v unsafe.Pointer, n uintptr) {
    madvise(v, n, _MADV_DONTNEED)
}

func sysUsed(v unsafe.Pointer, n uintptr) {
}
```

mem_darwin.go

```
func sysUnused(v unsafe.Pointer, n uintptr) {
    madvise(v, n, _MADV_FREE)
}

func sysUsed(v unsafe.Pointer, n uintptr) {
}
```

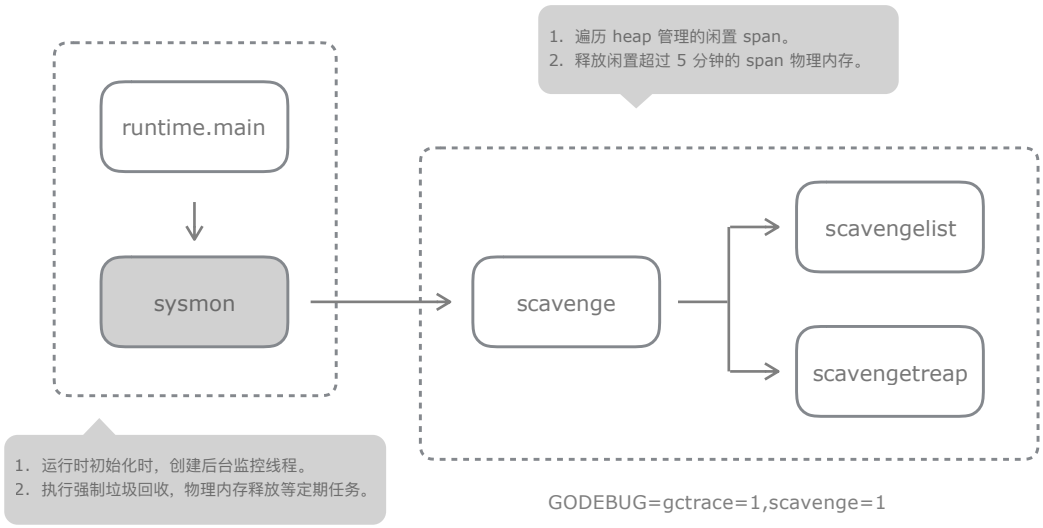
Windows 不支持类似机制，所以要调用 API 进行释放和重新分配。

获取内存块时，会调用 sysUsed 方法，从而补齐被释放的内存。

mem_windows.go

```
func sysUnused(v unsafe.Pointer, n uintptr) {
    r := stdcall3(_VirtualFree, uintptr(v), n, _MEM_DECOMMIT)
}

func sysUsed(v unsafe.Pointer, n uintptr) {
    r := stdcall4(_VirtualAlloc, uintptr(v), n, _MEM_COMMIT, _PAGE_READWRITE)
}
```

快速导览：



6. 其他

6.1 固定分配器

前文所提内存，多指用户对象存储。但编译后的程序，实际由用户和运行时两部分代码共同组成。运行时相关操作，同样需要分配内存。只是这些系统对象不在保留区域分配，且采取不同生命周期和复用策略。为此，专门设计了 FixAlloc 固定分配器。

从堆数据结构中，可以看到几个使用固定分配器的字段。

mheap.go

```
type mheap struct {
    spanalloc             fixalloc     // allocator for span*
    cachealloc            fixalloc     // allocator for mcache*
    treapalloc            fixalloc     // allocator for treapNodes* used by large objects
    specialfinalizeralloc fixalloc     // allocator for specialfinalizer*
    specialprofilealloc   fixalloc     // allocator for specialprofile*
}
```

这其中，我们最为熟悉的是管理内存块的 mspan。不同于 mheap 唯一实例，这几个字段各自持有独立固定分配器。在堆初始化方法里，分别被设定不同的分配属性。

> 前文中，span 指代内存块，而 mspan 则是管理对象自身。

mheap.go

```
func (h *mheap) init(spansStart, spansBytes uintptr) {
    h.treapalloc.init(unsafe.Sizeof(treapNode{}), nil, ...)
    h.spanalloc.init(unsafe.Sizeof(mspan{}), recordspan, ...)
    h.cachealloc.init(unsafe.Sizeof(mcache{}), nil, ...)
    h.specialfinalizeralloc.init(unsafe.Sizeof(specialfinalizer{}), nil, ...)
    h.specialprofilealloc.init(unsafe.Sizeof(specialprofile{}), nil, ...)
}
```

6.1.1 初始化

初始化时，除内存单元大小外，还可指定一关联函数和执行参数。

mfixalloc.go

```
// FixAlloc is a simple free-list allocator for fixed size objects.
// Malloc uses a FixAlloc wrapped around sysAlloc to manages its
// MCache and MSpan objects.

type fixalloc struct {
    size   uintptr
    first  func(arg, p unsafe.Pointer)
    arg    unsafe.Pointer
}
```

```
func (f *fixalloc) init(size uintptr, first func(arg, p unsafe.Pointer), ...) {
    f.size = size
```

```
    f.first = first
    f.arg = arg
    f.list = nil
    ...
}
```

6.1.2 分配

从结构上看，固定分配器采用早期内存分配器设计思路。以链表（list）管理回收可复用内存单元，另持一块待分割内存块（chunk）作为后备资源。

mfixalloc.go

```
type fixalloc struct {
    list   *mlink

    chunk  uintptr             // 后备内存。
    nchunk uint32              // 后备内存可分割数量。
}
```

分配时优先从复用链表提取，不足再从后备内存分割。如此看来，后备内存类似堆在三级结构中地位。另外，关联函数仅在分割新内存单元时执行，很适合做些记录性工作。

```
func (f *fixalloc) alloc() unsafe.Pointer {

    // 尝试从复用链表提取。
    if f.list != nil {
        v := unsafe.Pointer(f.list)
        f.list = f.list.next
        return v
    }

    // 如果后备资源不足，则重新获取（16 KB）。
    if uintptr(f.nchunk) < f.size {
        f.chunk = uintptr(persistentalloc(_FixAllocChunk, 0, f.stat))
        f.nchunk = _FixAllocChunk
    }

    // 从后备内存块分割。
    v := unsafe.Pointer(f.chunk)

    // 如果有关联函数，则执行。
    if f.first != nil {
        f.first(f.arg, v)
    }
```

```
    f.chunk = f.chunk + f.size
    f.nchunk -= uint32(f.size)

    return v
}
```

6.1.3 回收

回收操作仅将内存单元放回复用链表，并没有与释放相关的行为。

> 使用 FixAlloc 的几种管理对象，其数量不会太多，不释放物理内存影响不大。

mfixalloc.go

```
func (f *fixalloc) free(p unsafe.Pointer) {
    v := (*mlink)(p)
    v.next = f.list
    f.list = v
}
```

以堆为例，在提取和放回内存块时，会用固定分配器分配和回收 mspan 管理对象。

mheap.go

```
func (h *mheap) allocSpanLocked(npage uintptr, stat *uint64) *mspan {
    t := (*mspan)(h.spanalloc.alloc())
    t.init(s.base()+npage<<_PageShift, s.npages-npage)
}
```

```
func (h *mheap) freeSpanLocked(s *mspan, acctinuse, acctidle bool, unusedsince int64) {
    h.spanalloc.free(unsafe.Pointer(before))
}
```

6.1.4 后备

实际上，后备内存不仅仅为固定分配器服务，还频繁出现在运行时其他部件中。此处，我们只需了解其结构和工作机制便可。

数据结构颇为简单，仅存储内存地址和分配偏移。

malloc.go

```
type persistentAlloc struct {
    base *notInHeap
    off  uintptr
}
```

---

notInHeap 是空结构，包含一个计算偏移地址的方法。

```
type notInHeap struct{}
```

```
func (p *notInHeap) add(bytes uintptr) *notInHeap {
    return (*notInHeap)(unsafe.Pointer(uintptr(unsafe.Pointer(p)) + bytes))
}
```

同样基于性能考虑，线程（P）拥有独立后备内存。另以全局变量作为补充。

runtime2.go

```
type p struct {                     // 每个工作线程（M）都会绑定一个 P 对象。
    palloc persistentAlloc
}
```

malloc.go

```
var globalAlloc struct {
    mutex
    persistentAlloc
}
```

如此，优先从当前线程内无锁提取。如果不足，再向操作系统申请。

malloc.go

```
func persistentalloc1(size, align uintptr, sysStat *uint64) *notInHeap {
    const (
        chunk    = 256 << 10
        maxBlock = 64 << 10
    )

    // 如果大小超过 64 KB，直接向操作系统申请。
```

```
    if size >= maxBlock {
        return (*notInHeap)(sysAlloc(size, sysStat))
    }

    // 确定后备内存位置（本地或全局）。
    if mp != nil && mp.p != 0 {
        persistent = &mp.p.ptr().palloc
    } else {
        lock(&globalAlloc.mutex)
        persistent = &globalAlloc.persistentAlloc
    }

    // 如果内存不足，向操作系统申请（256 KB）。
    if persistent.off+size > chunk || persistent.base == nil {
        persistent.base = (*notInHeap)(sysAlloc(chunk, &memstats.other_sys))
        persistent.off = 0
    }

    // 从后备内存切分。
    p := persistent.base.add(persistent.off)
    persistent.off += size

    return p
}
```

向操作系统申请内存时，直接使用 sysAlloc，避开内存分配器的保留区域。

mem_linux.go

```
func sysAlloc(n uintptr, sysStat *uint64) unsafe.Pointer {
    p, err := mmap(nil, n, _PROT_READ|_PROT_WRITE, _MAP_ANON|_MAP_PRIVATE, -1, 0)
}
```

> 与 heap.sysAlloc 指定 arena_alloc 作为起始地址不同，sysAlloc 参数为 nil，由内核决定。

malloc.go

```
func (h *mheap) sysAlloc(n uintptr) unsafe.Pointer {
    p := h.arena_alloc
    sysMap(unsafe.Pointer(p), n, h.arena_reserved, &memstats.heap_sys)
}
```

6.2 内存块记录

在使用固定分配器的几个堆字段中，仅 mspan 指定了关联函数。

目的是在创建（非复用）新 mspan 实例时，将其保存到 heap.allspans 列表中。
该列表在 mgcmark 和 heapdump 中被用来遍历已分配内存。

mheap.go

```
type mheap struct {
    allspans []*mspan    // all spans out there
}
```

```
// recordspan adds a newly allocated span to h.allspans.
//
// This only happens the first time a span is allocated from
// mheap.spanalloc (it is not called when a span is reused).

func recordspan(vh unsafe.Pointer, p unsafe.Pointer) {

    h := (*mheap)(vh)
    s := (*mspan)(p)

    // 如 allspans 已满，扩容。
    if len(h.allspans) >= cap(h.allspans) {
        ...
    }

    // 将 mspan 添加到 allspans。
    h.allspans = h.allspans[:len(h.allspans)+1]
    h.allspans[len(h.allspans)-1] = s
}
```