第 3 章

Go 语言内存管理洗髓经

Go 语言的内存管理及设计也是开发者需要了解的领域之一，要理解 Go 语言的内存管理，就必须先理解操作系统及机器硬件是如何管理内存的。因为 Go 语言的内部机制是建立在这个基础之上的，它的设计本质上就是尽可能地发挥操作系统层面的优势，而避开导致低效的情况。

本章会围绕以下六个话题逐步展开。

(1) 何为内存。

(2) 内存为什么需要管理。

(3) 操作系统是如何管理内存的。

(4) 如何用 Go 语言自己实现一个内存管理模型。

(5) Go 语言内存管理之魂：TCMalloc。

(6) Go 语言中是如何管理内存的。

3.1 何为内存

说到内存，即使没有任何软件基础知识，第一印象也应该想到的是如图 3.1 所示的实物。

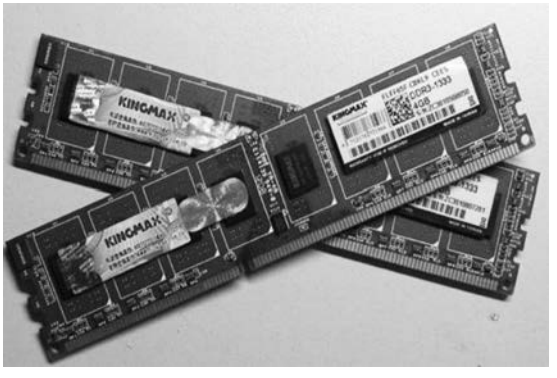

图 3.1　物理内存条

图 3.1 中的实物常被称为内存条，是计算机硬件组成的一部分，也是真正给软件提供内存的物理空间。如果计算机没有内存条，则根本谈不上有内存之说。

那么内存的作用是什么呢？如果将计算机的存储媒介中的处理性能与容量做一个对比，则会出现如图 3.2 所示的金字塔模型。

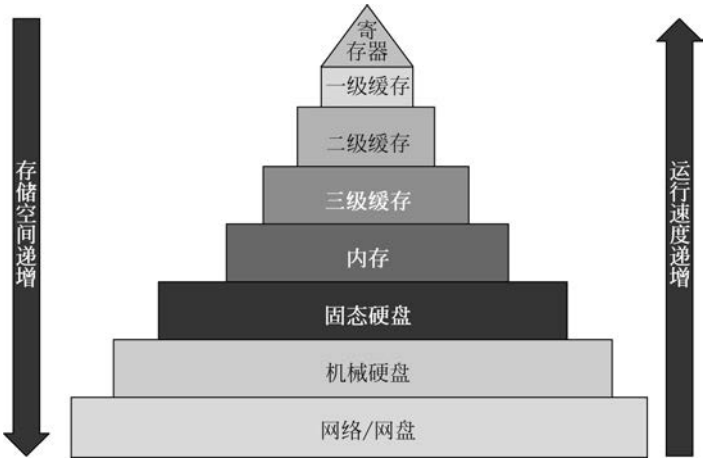

图 3.2 计算机存储媒介金字塔模型

从图 3.2 可以得出处理速度与存储容量是成反比的。也就是说，性能越强的计算机，其硬件资源越是稀缺，所以合理地利用和分配就越重要。

例如内存与硬盘的对比，因为硬盘的容量相对来讲非常廉价，虽然内存目前也可以用到 10GB 级别，但是从处理速度来看，两者的差距还是相差甚大的，具体如表 3.1 所示。

表 3.1 硬盘与内存的对比

DDR3 与硬盘速度对比	DDR4 与硬盘速度对比
DDR3 内存的读写速度大概 10GB/s	DDR4 内存的读写速度大概 50GB/s
固态硬盘的读写速度大概 300MB/s，大概是内存的三十分之一	固态硬盘的读写速度大概 300MB/s，大概是内存的二百分之一
机械硬盘的读写速度是 100MB/s，大概是内存的百分之一	机械硬盘的读写速度是 100MB/s，大概是内存的五百分之一

由于读写速度相差甚大，所以将大部分程序逻辑临时用的数据，全部存在内存之中，例如，变量、全局变量、函数跳转地址、静态库、执行代码、临时开辟的内存结构体（对象）等。

3.2 内存为什么需要管理

当存储的东西越来越多，也就发现物理内存的容量依然不够用，提高对物理内存的利用率和合理地分配内存，管理就变得非常重要了。

（1）操作系统会对内存进行非常详细的管理。

（2）基于操作系统的基础上，不同语言的内存管理机制也应运而生，有一些语言并没有提供自动的内存管理模式，有的语言却提供了自身程序的内存管理模式，如表 3.2 所示。

表 3.2　自动与非自动内存管理的语言

内存自动管理的语言（部分）	内存非自动管理的语言（部分）
Go	C
Java	C++
Python	Rust

为了降低内存管理的难度，像 C、C++这样的编程语言会完全将分配和回收内存的权限交给开发者，而 Rust 则通过生命周期限定开发者对非法权限内存的访问并以此来自动回收，因而并没有提供自动管理的一套机制，但是像 Go、Java、Python 这类为了完全让开发者关注代码逻辑本身，语言层提供了一套管理模式。因为 Go 编程语言给开发者提供了一套内存管理模式，所以开发者有必要了解一下 Go 语言提供了哪些内存管理方式。

在理解 Go 语言层内存管理之前，应先了解操作系统针对物理内存提供了哪些管理方式，当插上内存条之后，通过操作系统是如何将软件存放在这个绿色的物理内存条中去的。

3.3　操作系统是如何管理内存的

对计算机来讲内存真正的载体是物理内存条，这个是实打实的物理硬件容量，所以在操作系统中定义的这部分容量叫物理内存。

物理内存的布局实际上就是一个内存大数组，如图 3.3 所示。

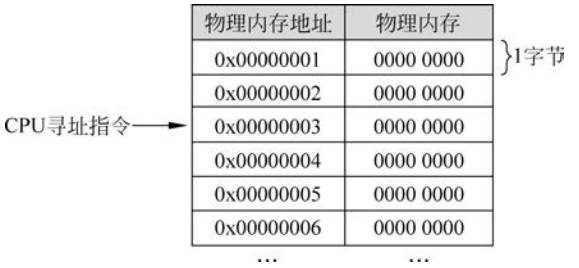

图 3.3　物理内存布局

每个元素都会对应一个地址，称为物理内存地址。CPU 在运算的过程中，如果需要从内存中取 1 字节的数据，就需要基于这个数据的物理内存地址去运算，而且物理内存的地址是连续的，可以根据一个基准地址进行偏移来取得相应的连续内存数据。

一个操作系统是不可能只运行一个程序的，这个大数组物理内存势必要被多个程序分成多份，供每个程序使用，但是程序是活的，一个程序可能一会需要 1MB 的内存，一会又需要 1GB 的内存。操作系统只能取这个程序允许的最大内存极限来将内存分配给这个进程，

但这样会导致每个进程都会多要去一部分内存，而这些多要的内存却大概率不会被使用，如图 3.4 所示。

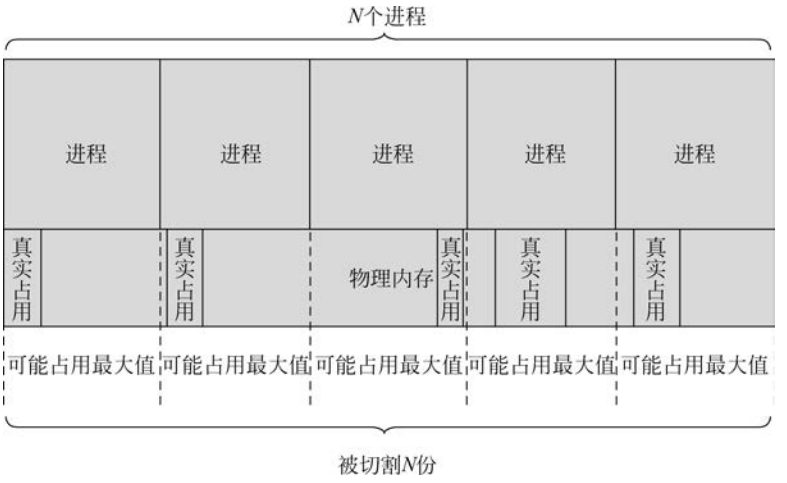

图 3.4　物理内存分配的困局

当 N 个程序同时使用同一块内存时，产生读写的冲突也在所难免。这样就会导致这些昂贵的物理内存条，几乎运行不了几个程序，内存的利用率也就提高不上来。

这就引出了操作系统的内存管理方式，操作系统提供了虚拟内存来解决这件事。

3.3.1　虚拟内存

所谓虚拟，类似假、凭空而造的意思。对比图 3.3 所示的物理内存布局，虚拟内存的大致表现方式如图 3.5 所示。

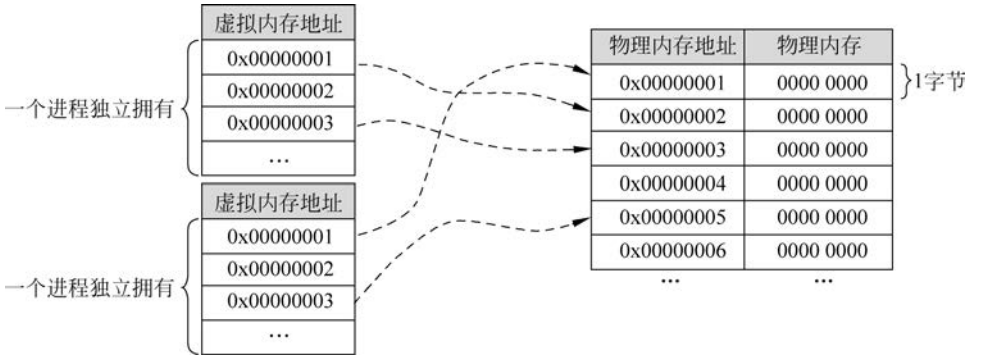

图 3.5　虚拟内存布局

虚拟内存地址是基于物理内存地址之上凭空而造的一个新的逻辑地址，而操作系统暴露给用户进程的只是虚拟内存地址，操作系统内部会对虚拟内存地址和真实的物理内存地

址建立映射关系，来管理地址的分配，从而使物理内存的利用率提高。

这样用户程序（进程）只能使用虚拟的内存地址获取数据，系统会将这个虚拟地址翻译成实际的物理地址。这里每个程序统一使用一套连续虚拟地址，例如 0x 0000 0000～0x ffff ffff。从程序的角度来看，它觉得自己独享了一整块内存，并且不用考虑访问冲突的问题。系统会将虚拟地址翻译成物理地址，从内存上加载数据。

但如果仅仅把虚拟内存直接理解为地址的映射关系，那就低估虚拟内存的作用了。

虚拟内存的目的是解决以下几件事：

（1）物理内存无法被最大化利用。

（2）程序逻辑内存空间使用独立。

（3）内存不够，继续虚拟磁盘空间。

对于（1）和（2）两点，上述已经有一定的描述了，其中针对（1）的最大化，虚拟内存还实现了“读时共享，写时复制”的机制，可以在物理层同一字节的内存地址被多个虚拟内存空间映射，表现方式如图 3.6 所示。

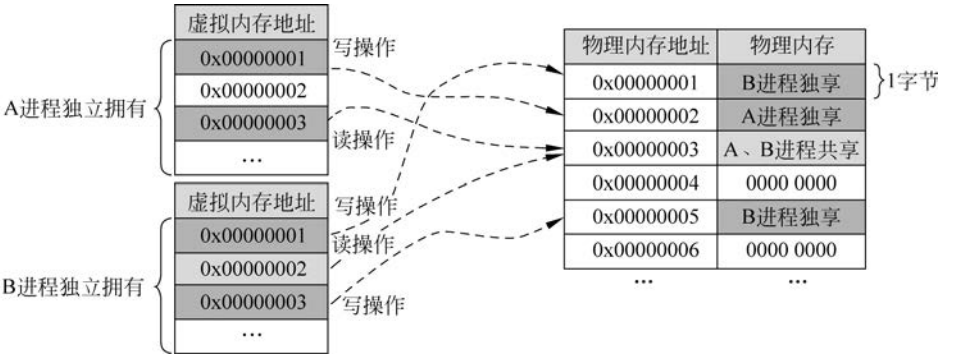

图 3.6　读时共享，写时复制

如图 3.6 所示，如果一个进程需要进行写操作，则这个内存将会被复制一份，成为当前进程的独享内存。如果是读操作，则可能多个进程访问的物理空间是相同的空间。

如果一个内存几乎是被读取的，则可能多个进程共享同一块物理内存，但是它们的各自虚拟内存是不同的。当然这个共享并不是永久的，当其中有一个进程对这个内存发生写操作时，就会复制一份，执行写操作的进程就会将虚拟内存地址映射到新的物理内存地址上。

对于第（3）点，是虚拟内存为了最大化利用物理内存，如果进程使用的内存足够大，则会导致物理内存短暂的供不应求，此时虚拟内存也会“开疆拓土”从磁盘（硬盘）上虚拟出一定量的空间，挂在虚拟地址上，而且这个动作对于进程来讲是不知道的，因为进程只能够“看见”自己的虚拟内存空间，如图 3.7 所示。

综上可见虚拟内存的重要性，不仅提高了利用率，而且整条内存调度的链路完全是对用户物理内存透明，用户可以安心地使用自身进程独立的虚拟内存空间进行开发。

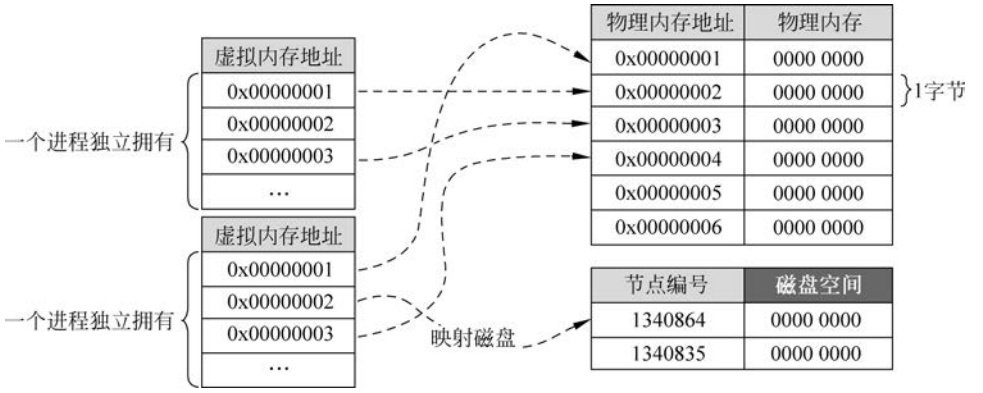

图 3.7 虚拟内存从磁盘映射空间

3.3.2 MMU 内存管理单元

对于虚拟内存地址是如何映射到物理内存地址上的呢？会不会是一个固定匹配地址逻辑处理的？假设使用固定匹配地址逻辑做映射，可能会出现很多虚拟内存映射到同一个物理内存上，如果发现被占用，则会再重新映射。这样对映射地址寻址的代价极大，所以操作系统又加了一层专门用来管理虚拟内存和物理内存映射关系的东西，即 MMU(Memory Management Unit，内存管理单元)，如图 3.8 所示。

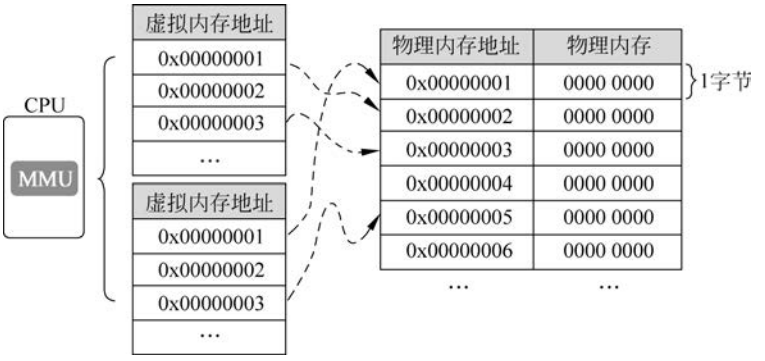

图 3.8 MMU 内存管理单元

MMU 是在 CPU 里的，或者说是 CPU 具有一个 MMU，下面来介绍一下 MMU 具体的管理逻辑。

3.3.3 虚拟内存本身怎么存放

虚拟内存本身是通过一个叫页表(Page Table)的东西实现的，接下来介绍页和页表这两个概念。

1. 页

页是操作系统中用来描述内存大小的一个单位名称。一个页的含义是大小为 4KB(1024×4=4096 字节)的内存空间。操作系统对虚拟内存空间是按照这个单位来管理的。

2. 页表

页表实际上就是页的集合,即基于页的一个数组。页只表示内存的大小,而页表条目(PTE[①])才是页表数组中的一个元素。

为了方便读者理解,下面用一个抽象的图来表示页、页表、页表元素 PTE 的概念和关系,如图 3.9 所示。

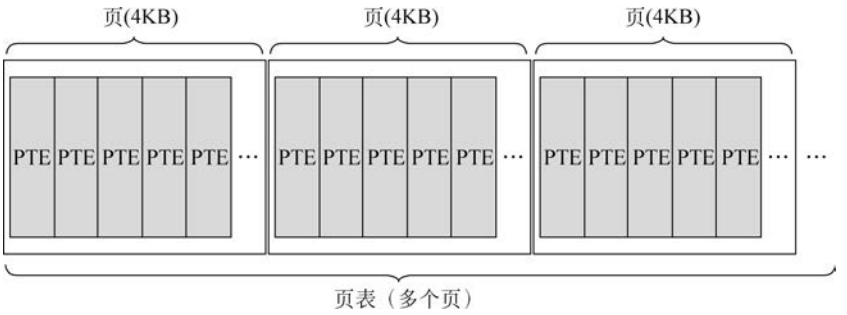

图 3.9　页、页表、PTE 之间的关系

虚拟内存的实现方式,大多数是通过页表实现的。操作系统虚拟内存空间被分成一页一页来管理,每页的大小为 4KB(当然这是可以配置的,不同操作系统不一样)。磁盘和主内存之间的置换也是以页为单位来操作的。4KB 算是通过实践折中出来的通用值,太小了会出现频繁置换,太大了又会浪费内存。

虚拟内存到物理内存的映射关系的存储结构类似上述图 3.9 中的页表记录,实则是一个数组。这里需要注意的是,页是一次读取的内存单元,但是真正到虚拟内存寻址的是 PTE,也就是页表中的一个元素。PTE 的大致内部结构如图 3.10 所示。

	有效位	物理内存页号或磁盘地址	
1个PTE	0	Null	PTE0
	1	0xff112233	PTE1
	0	Null	PTE2
	0	0x33ac93f1	PTE3
	1	0xbcff1463	PTE4

图 3.10　PTE 内部构造

可以看出每个 PTE 是由一个有效位和一个物理页号或者磁盘地址组成,有效位表示当前虚拟页是否已经被缓存在主内存中(或者 CPU 的高速缓存 Cache 中)。

虚拟页为何会有是否已经被缓存在主内存中一说?虚拟页表(简称页表)虽然作为虚拟

① PTE 是 Page Table Entry 的缩写,表示页表条目。PTE 由一个有效位和 N 位地址字段构成,能够有效标识这个虚拟内存地址是否分配了物理内存。

内存与物理内存的映射关系，但是本身也需要存放在某个位置上，所以自身也占用一定内存，所以页表本身也被操作系统放在物理内存的指定位置。CPU 把虚拟地址给 MMU，MMU 去物理内存中查询页表，得到实际的物理地址。当然 MMU 不会每次都去查询，它自己也有一份缓存，叫作 Translation Lookaside Buffer（TLB）[①]，是为了加速地址翻译。CPU、MMU 与 TLB 的相互关系如图 3.11 所示。

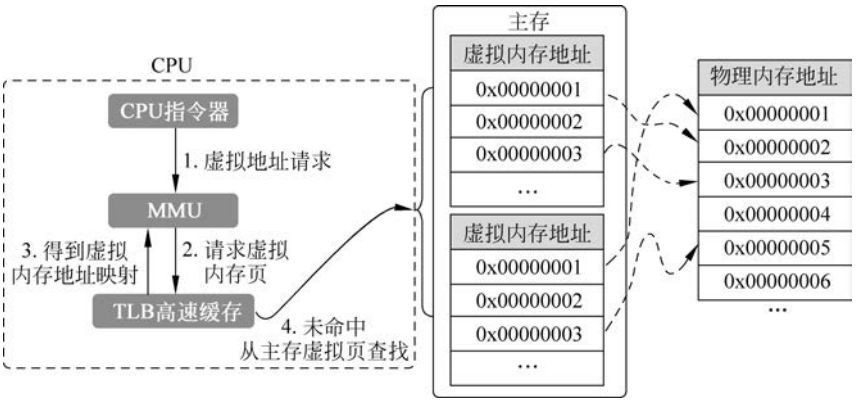

图 3.11　CPU、MMU 与 TLB 的交互关系

从图 3.11 可以看出，TLB 是虚拟内存页，即虚拟地址和物理地址映射关系的缓存层。MMU 当收到地址查询指令，第一时间请求 TLB，如果没有才会进行从内存中的虚拟页进行查找，这样可能会触发多次内存读取，而读取 TLB 则不需要内存读取，所进程读取的步骤如下：

（1）CPU 进行虚拟地址请求 MMU。

（2）MMU 优先从 TLB 中得到虚拟页。

（3）如果得到，则返给上层。

（4）如果没有，则从主存的虚拟页表中查询关系。

下面继续分析 PTE 的内部构造，根据有效位的特征可以得到不同的含义：

（1）有效位为 1，表示虚拟页已经被缓存在内存（或者 CPU 高速缓存 TLB-Cache）中。

（2）有效位为 0，表示虚拟页未被创建且没有占用内存（或者 CPU 高速缓存 TLB-Cache），或者表示已经创建虚拟页，但是并没有存储到内存（或者 CPU 高速缓存 TLB-Cache）中。

通过上述的标识位，可以将虚拟页集合分成三个子集，如表 3.3 所示。

---

① CPU 每次访问虚拟内存，虚拟地址都必须转换为对应的物理地址。从概念上讲，这个转换需要遍历页表，页表是三级页表，需要 3 次内存访问。也就是说，每次虚拟内存访问都会导致 4 次物理内存访问。简单点说，如果一次虚拟内存访问对应了 4 次物理内存访问，肯定比 1 次物理访问慢，这样虚拟内存肯定不会发展起来。幸运的是，有一个聪明的做法解决了大部分问题：现代 CPU 使用一小块关联内存，用来缓存最近访问的虚拟页的 PTE。这块内存称为 Translation Lookaside Buffer（TLB），参考 IA-64 Linux Kernel：Design and Implementation。

表 3.3　虚拟页被分成的三个子集

有　效　位	集 合 特 征
1	虚拟内存已创建和分配页，已缓存在物理内存(或 TLB-Cache)中
0	虚拟内存还未分配或创建
0	虚拟内存已创建和分配页，但未缓存在物理内存(或 TLB-Cache)中

对于 Go 语言开发者，对虚拟内存的存储结构了解到此步即可，如果想更深入地了解 MMU 存储结果，则可以翻阅其他操作系统或硬件相关书籍或资料。下面来分析一下访问一次内存的整体流程。

3.3.4　CPU 内存访问过程

一次 CPU 内存访问的详细流程如图 3.12 所示。

当某个进程进行一次内存访问指令请求时，将触发如图 3.12 所示的内存访问，具体的访问流程如下：

(1) 进程将内存相关的寄存器指令请求运算发送给 CPU，CPU 得到具体的指令请求。

(2) 计算指令被 CPU 加载到寄存器中，准备执行相关指令逻辑。

(3) CPU 对相关可能请求的内存生成虚拟内存地址。一个虚拟内存地址包括虚拟页号 VPN(Virtual Page Number)和虚拟页偏移量 VPO(Virtual Page Offset)[①]。

(4) 从虚拟地址中得到虚拟页号 VPN。

(5) 通过虚拟页号 VPN 请求 MMU 内存管理单元。

(6) MMU 通过虚拟页号查找对应的 PTE 条目(优先层 TLB 缓存查询)。

(7) 通过得到对应的 PTE 上的有效位来判断当前虚拟页是否在主存中。

(8) 如果索引到的 PTE 条目的有效位为 1，则表示命中，将对应 PTE 上的物理页号 PPN(Physical Page Number)和虚拟地址中的虚拟页偏移量 VPO 进行串联从而构造出主存中的物理地址 PA(Physical Address)[②]，进入步骤(9)。

(9) 通过物理内存地址访问物理内存，当前的寻址流程结束。

(10) 如果有效位为 0，则表示未命中，一般称这种情况为缺页。此时 MMU 将产生一个缺页异常，抛给操作系统。

(11) 操作系统捕获到缺页异常，开始执行异常处理程序。

(12) 此时将选择一个牺牲页并将对应的所缺虚拟页调入并更正新页表上的 PTE，如果当前牺牲页有数据，则写入磁盘，得到物理内存页号 PPN(Physical Page Number)。

(13) 缺页处理程序更新之前索引到的 PTE，并且写入物理内存页号 PPN，有效位设置为 1。

---

① 一个虚拟地址 VA(Virtual Address)= 虚拟页号 VPN+虚拟页偏移量 VPO。

② 一个物理地址 PA(Physical Address)=物理页号 PPN×页长度 PageSize+物理页号偏移 PPO(Physical Page Offset)。

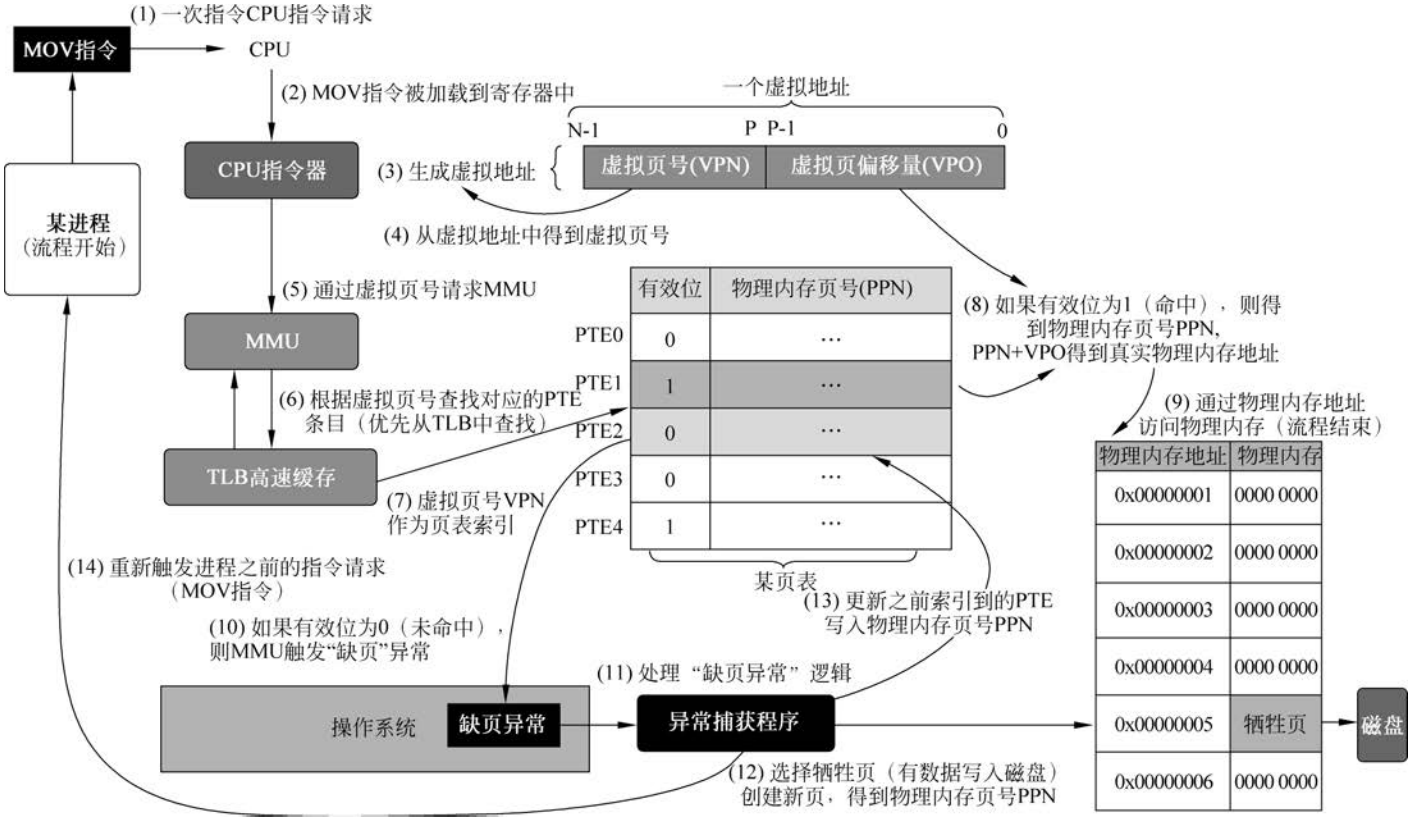

图 3.12　一次 CPU 内存访问的详细流程

(14) 缺页处理程序再次返回原来的进程,并且再次执行缺页指令,CPU 重新将虚拟地址发给 MMU,此时虚拟页已经存在物理内存中,本次一定会命中,通过步骤(1)～(9)流程,最终将请求的物理内存返给处理器。

以上就是一次 CPU 访问内存的详细流程。可以看出来在上述流程中,从第(10)步之后的流程就稍微有一些烦琐。类似产生异常信号、捕获异常,再处理缺页流程,如选择牺牲页,还要将牺牲页的数据存储到磁盘上等,所以如果频繁地执行步骤(10)～(14)会对性能影响很大。因为牺牲页有可能涉及磁盘的访问,而磁盘的访问速度非常慢,这样就会引发程序性能的急剧下降。

一般步骤(1)～(9)流程结束则表示页命中,反之为未命中,所以就会出现一个新的性能级指标,即命中率。命中率是访问次数与页命中次数之比。一般命中率低说明物理内存不足,数据在内存和磁盘之间交换频繁,但如果物理内存充分,则不会出现频繁的内存颠簸现象。

3.3.5　内存的局部性

上述了解到的内存命中率实际上是衡量每次内存访问均能被页直接寻址到而不是产生缺页的指标,所以如果经常在一定范围内,则出现缺页的情况就会降低,这就是程序的一种局部性特性的体现。

局部性就是在多次内存引用的时候,会出现有的内存被引用多次,而且在该位置附近的其他位置,也有可能接下来被引用。大多数程序会具备局部性的特点。

实际上操作系统在设计过程中经常会用到缓存来提升性能,或者在设计解决方案等架构的时候也会考虑到缓存或者缓冲层的概念,实则就是利用程序或业务的局部性特征。因为如果没有局部性的特性,则缓存级别将起不到太大的作用,所以在设计程序或者业务的时候应该多考虑增强程序局部性的特征,这样的程序执行起来会更快。

下面通过一个非常典型的案例来验证程序局部性,具体的代码如下:

```
//第一篇/chapter3/MyGolang/loop.go
package MyGolang

func Loop(nums []int, step int) {
   l := len(nums)
   for i := 0; i < step; i++{
      for j := i; j < l; j += step {
        nums[j] = 4 //访问内存,并写入值
      }
   }
}
```

Loop()函数的功能是遍历数组 nums,并且将 nums 中的每个元素均设置为 4,但是这里用了一个 step 来规定每次遍历的跨度。可以跟读上述代码,如果 step 等于 1,则外层 for

循环只会执行 1 次，内层 for 循环则正常遍历 nums，与此相当的代码如下：

```
func Loop(nums []int, step int) {
   l := len(nums)
   for j := 0; j < l; j += 1 {
       nums[j] = 4 //访问内存,并写入值
   }
}
```

如果 Step 等于 3，则表示外层 for 循环要一共完成 3 次，内层 for 循环每次遍历的数组下标值都相差 3。第一次遍历会被遍历的 nums 下标为 0、3、6、9、12……，第二次遍历会遍历的 nums 下标为 1、4、7、10、13……，第三次遍历会被遍历的 nums 下标为 2、5、8、11、14……，这样 3 次外循环就会完整遍历 nums 数组。

上述的程序表示访问数组的局部性，step 跨度越小，则表示访问 nums 相邻内存的局部性越好，step 越大则相反。

接下来用 Go 语言的 Benchmark 性能测试来分别对 step 取不同的值进行压测，来看一看通过 Benchmark 执行 Loop()函数而统计出来的几种情况，最终消耗的时间差距为多少。首先创建 loop_test.go 文件，实现一个制作数组并且赋值初始化内存值的函数 CreateSource()，代码如下：

```
//第一篇/chapter3/MyGolang/loop_test.go
package MyGolang

import "testing"

func CreateSource(len int) []int {
   nums := make([]int, 0, len)

   for i := 0 ; i < len; i++{
      nums = append(nums, i)
   }

   return nums
}
```

其次实现一个 Benchmark，制作一个长度为 10000 的数组，这里需要注意的是，创建完数组后要执行 b.ResetTimer()重置计时，去掉 CreateSource()消耗的时间，step 跨度为 1 的代码如下：

```
//第一篇/chapter3/MyGolang/loop_test.go

func BenchmarkLoopStep1(b *testing.B) {
```

```
    //制作源数据,长度为 10000
    src := CreateSource(10000)

    b.ResetTimer()
    for i:= 0; i < b.N; i++{
        Loop(src, 1)
    }
}
```

Go 语言中的 b.N 表示 Go 一次压测最终循环的次数。BenchmarkLoopStep1()会将 N 次的总耗时时间除以 N,得到平均一次执行 Loop()函数的耗时。因为要对比多个 step 的耗时差距,按照上述代码再依次实现 step 为 2、3、4、5、6、12、16 等 Benchmark 性能测试,代码如下:

```
//第一篇/chapter3/MyGolang/loop_test.go
func BenchmarkLoopStep2(b *testing.B) {
    //制作源数据,长度为 10000
    src := CreateSource(10000)

    b.ResetTimer()
    for i:= 0; i < b.N; i++{
        Loop(src, 2)
    }
}

func BenchmarkLoopStep3(b *testing.B) {
    //制作源数据,长度为 10000
    src := CreateSource(10000)

    b.ResetTimer()
    for i:= 0; i < b.N; i++{
        Loop(src, 3)
    }
}

func BenchmarkLoopStep4(b *testing.B) {
    //制作源数据,长度为 10000
    src := CreateSource(10000)

    b.ResetTimer()
    for i:= 0; i < b.N; i++{
        Loop(src, 4)
    }
```

```
}

func BenchmarkLoopStep5(b * testing.B) {
   //制作源数据,长度为 10000
   src : = CreateSource(10000)

   b.ResetTimer()
   for i: = 0; i < b.N; i++{
      Loop(src, 5)
   }
}

func BenchmarkLoopStep6(b * testing.B) {
   //制作源数据,长度为 10000
   src : = CreateSource(10000)

   b.ResetTimer()
   for i: = 0; i < b.N; i++{
      Loop(src, 6)
   }
}

func BenchmarkLoopStep12(b * testing.B) {
   //制作源数据,长度为 10000
   src : = CreateSource(10000)

   b.ResetTimer()
   for i: = 0; i < b.N; i++{
      Loop(src, 12)
   }
}

func BenchmarkLoopStep16(b * testing.B) {
   //制作源数据,长度为 10000
   src : = CreateSource(10000)

   b.ResetTimer()
   for i: = 0; i < b.N; i++{
      Loop(src, 16)
   }
}
```

上述每个 Benchmark 都是相似的代码,只有 step 传参不同,接下来通过执行下述指令进行压测,指令如下:

```
$ go test -bench=. -count=3
```

其中“count=3”表示每个 Benchmark 要执行 3 次，这样可以更好地验证上述的结果，具体的运行结果如下：

```
goos: darwin
goarch: amd64
pkg: MyGolang
BenchmarkLoopStep1-12            366787        2792 ns/op
BenchmarkLoopStep1-12            432235        2787 ns/op
BenchmarkLoopStep1-12            428527        2849 ns/op
BenchmarkLoopStep2-12            374282        3282 ns/op
BenchmarkLoopStep2-12            363969        3263 ns/op
BenchmarkLoopStep2-12            361790        3315 ns/op
BenchmarkLoopStep3-12            308587        3760 ns/op
BenchmarkLoopStep3-12            311551        4369 ns/op
BenchmarkLoopStep3-12            289584        4622 ns/op
BenchmarkLoopStep4-12            275166        4921 ns/op
BenchmarkLoopStep4-12            264282        4504 ns/op
BenchmarkLoopStep4-12            286933        4869 ns/op
BenchmarkLoopStep5-12            223366        5609 ns/op
BenchmarkLoopStep5-12            202597        5655 ns/op
BenchmarkLoopStep5-12            214666        5623 ns/op
BenchmarkLoopStep6-12            187147        6344 ns/op
BenchmarkLoopStep6-12            177363        6397 ns/op
BenchmarkLoopStep6-12            185377        6333 ns/op
BenchmarkLoopStep12-12           126860        9660 ns/op
BenchmarkLoopStep12-12           127557        9741 ns/op
BenchmarkLoopStep12-12           126658        9492 ns/op
BenchmarkLoopStep16-12            95116       12754 ns/op
BenchmarkLoopStep16-12            95175       12591 ns/op
BenchmarkLoopStep16-12            92106       12533 ns/op
PASS
ok   MyGolang 31.712s
```

对上述结果以第一行为例进行简单解读：

(1) BenchmarkLoopStep1－12 中的 12 表示 GOMAXPROCS(线程数)为 12，这个在此处不需要过度关心。

(2) 366787 表示一共执行了 366787 次，即代码中 b.N 的值，这个值不是固定不变的。实际上是通过循环调用 366787 次 Loop()函数得到的最后性能结果。

(3) 2792 ns/op 表示平均每次执行 Loop()函数所消耗的时间是 2792ns。

通过上述结果可以看出，随着 Step 参数的增加，内存访问的局部性就越差，执行 Loop()的

性能也就越差，从 Step 为 16 和 Step 为 1 的结果来看，性能相差近 4～5 倍。

通过结果可以得出结论，如果要设计出一个更加高效的程序，则提高代码的局部性访问是非常有效的程序性能优化手段之一。

思考 在 Go 语言的 GPM 调度器模型中，为什么一个 G 开辟的子 G 优先放在当前的本地 G 队列中，而不是放在其他 M 上的本地 P 队列中？GPM 为何要满足局部性的调度设计？

3.4 如何用 Go 语言实现内存管理和内存池设计

本节介绍自主实现一个内存管理模块大致需要哪些基础的开发和组件建设。接下来的一些代码不需要读者去掌握，因为 Go 语言已经给开发者提供了内存管理模式，开发者不需要关心 Go 语言的内存分配情况，但是为了更好地理解 Go 语言的内存管理模型，需要了解如果自己实现一套简单的内存管理模块应该需要关注哪些点和需要实现哪些必要的模块和机制。

本节接下来的内容即是通过 Go 语言自我实现一个内存管理模块和内存池的建设，该模块非企业级开发而是帮助理解内存管理模型的教程型代码。

3.4.1 基于 Cgo 的内存 C 接口封装

因为 Go 语言已经内置了内存管理机制，所以如果用 Go 语言原生的语法结构（如 Slice、String、Map 等）都会自动触发 Go 语言的内存管理机制。本案例为了介绍如何实现一个自我管理的内存模型，所以直接使用 C 语言的 malloc()、free()系统调用来开辟和释放内存空间，使用 C 语言的 memcpy()、memmove()等进行内存的复制和移动。至于如何封装 Go 语法的 Malloc()、Free()、Memcpy()、Memmove()等函数，即利用了 Go 语言中的 Cgo 机制。

注意 Cgo 提供了 Go 语言和 C 语言相互调用的机制。可以通过 Cgo 用 Go 调用 C 的接口，对于 C++的接口可以用 C 包装一下提供给 Go 调用。被调用的 C 代码可以直接以源代码形式提供或者打包静态库或动态库在编译时的链接。

Cgo 的具体使用教程本章将不继续详细介绍，本章主要介绍在内存管理设计中所涉及的部分 Cgo 语法。

创建一个 zmem/目录，作为当前内存实现案例的项目名称。在 zmem/目录下再创建 c/文件夹，这里用来实现通过 Cgo 来封装 C 语言的内存管理接口。

在 c/目录下创建 memory.go 文件，分别封装 C 语言的内存接口，代码如下：

```
//zmem/c/memory.go

package c
```

```
/*
#include <string.h>
#include <stdlib.h>
 */
import "C"
import "unsafe"

func Malloc(size int) unsafe.Pointer {
   return C.malloc(C.size_t(size))
}

func Free(data unsafe.Pointer) {
   C.free(data)
}

func Memmove(dest, src unsafe.Pointer, length int) {
   C.memmove(dest, src, C.size_t(length))
}

func Memcpy(dest unsafe.Pointer, src []Byte, length int) {
   srcData := C.CBytes(src)
   C.memcpy(dest, srcData, C.size_t(length))
}
```

接下来分别介绍上述代码几个需要注意的地方。

1. import "C"

代表 Cgo 模块的启动，其中 import "C"上面的全部注释代码（中间不允许有空白行）均为 C 语言原生代码。因为在下述接口封装中使用了 C 语言的 malloc()、free()、memmove()、memcpy()等函数，这些函数的声明需要包含头文件 string.h 和 stdlib.h，所以在注释部分添加了导入这两个头文件的代码，并且通过 import "C"导入。

2. unsafe.Pointer

这里以 malloc()系统调用为例，通过 man[①] 手册查看 malloc()函数的原型如下：

```
#include <stdlib.h>

void *malloc(size_t size);
```

函数 malloc()形参是 C 语言中的 size_t 数据类型，在 Go 语言中使用对应的 C 类型是 C.size_t，一般的 C 基本类型只需通过 C 包直接访问，但是对于 malloc()的返回值 void* 来

① Man 手册页（Manua pages，man page）是 Linux 操作系统在线软件文档的一种普遍形式。内容包括计算机程序库和系统调用等命令的帮助手册。

讲，这是一个万能指针，其用法类似 Go 语言中的 interface{}，但是在语法上并不能将二者直接画等号，而 Go 语言给开发者提供了一个可以直接对等 C 中 void* 的数据类型，即 unsafe.Pointer。unsafe.Pointer 是 Go 语言封装好的可以比较自由访问的指针类型，其含义和 void* 万能指针相似。在语法上，也可以直接将 void* 类型数据赋值给 unsafe.Pointer 类型数据。

3. Go 与 C 的字符串等类型转换

在 Cgo 中 Go 的字符串与 Byte 数组都会转换为 C 的 char 数组，其中 Go 的 Cgo 模块提供了几种方法供开发者使用：

```
//Go 字符串转换为 C 字符串.C 字符串使用 malloc 分配,因此需要使用 C.free 以避免内存泄漏
func C.CString(string) *C.char

//Go Byte 数组转换为 C 的数组.使用 malloc 分配的空间,因此需要使用 C.free 避免内存泄漏
func C.CBytes([]Byte) unsafe.Pointer

//C 字符串转换为 Go 字符串
func C.GoString(*C.char) string

//C 字符串转换为 Go 字符串,指定转换长度
func C.GoStringN(*C.char, C.int) string

//C 数据转换为 Byte 数组,指定转换的长度
func C.GoBytes(unsafe.Pointer, C.int) []Byte
```

其中 C.CBytes()方法可以将 Go 的[]Byte 切片转换成 unsafe.Pointer 类型。利用这个转换功能，来分析一下是如何封装 memcpy()函数的：

```
func Memcpy(dest unsafe.Pointer, src []Byte, length int) {
   srcData := C.CBytes(src)
   C.memcpy(dest, srcData, C.size_t(length))
}
```

新封装的 Memcpy()的第 1 个形参是任意指针类型，表示复制的目标地址；第 2 个形参是[]Byte 类型，表示被复制的源数据；第 3 个参数表示本次复制数据的长度。因为 C 语言中的 memcpy()函数的原型如下：

```
    #include <string.h>

void *memcpy(void *dst, const void *src, size_t n);
```

对于 src 数据源形参需要[]Byte 转换为 unsafe.Pointer，因此在调用 C 的接口时应通

过 C.CBytes()转换一下。

Free()和 Memmove()方法的封装和上述一样。Free()与 Malloc()对应，Memmove()为移动一块连续内存。

接下来将上述封装做一个简单的单元测试，在 c/目录下创建 memory_test.go 文件，实现代码如下：

```
package c_test

import (
   "zmem/c"
   "Bytes"
   "encoding/binary"
   "fmt"
   "testing"
   "unsafe"
)

func IsLittleEndian() bool {
   var n int32 = 0x01020304

   //下面是为了将 int32 类型的指针转换成 Byte 类型的指针
   u := unsafe.Pointer(&n)
   pb := (*Byte)(u)

   //取得 pb 位置对应的值
   b := *pb

   //由于 b 是 Byte 类型，最多保存 8 位，所以只能取得开始的 8 位
   //小端: 04 (03 02 01)
   //大端: 01 (02 03 04)
   return (b == 0x04)
}

func IntToBytes(n uint32) []Byte {
   x := int32(n)
   BytesBuffer := Bytes.NewBuffer([]Byte{})

   var order binary.ByteOrder
   if IsLittleEndian() {
     order = binary.LittleEndian
   } else {
     order = binary.BigEndian
   }
```

```
    binary.Write(BytesBuffer, order, x)

    return BytesBuffer.Bytes()
}

func TestMemoryC(t *testing.T) {
    data := c.Malloc(4)
    fmt.Printf("data %+v, %T\n", data, data)
    myData := (*uint32)(data)
    *myData = 5
    fmt.Printf("data %+v, %T\n", *myData, *myData)

    var a uint32 = 100
    c.Memcpy(data, IntToBytes(a), 4)
    fmt.Printf("data %+v, %T\n", *myData, *myData)

    c.Free(data)
}
```

单元测试接口是 TestMemoryC()，首先通过 Malloc()开辟 4 字节内存，然后将这 4 字节赋值为 5，打印结果看 data 的值是否为 5。最后将 100 通过 Memcpy()复制给这 4 字节，看最后的结果是否为 100，运行结果如下：

```
=== RUN  TestMemoryC
data 0x9d040a0, unsafe.Pointer
data 5, uint32
data 100, uint32
--- PASS: TestMemoryC (0.00s)
PASS
```

通过单元测试结果来看，目前的内存开辟和复制的相关接口可以正常使用，接下来基于这些接口实现内存管理的模块。

3.4.2 基础内存缓冲 Buf 实现

在 zmem 目录下再创建 mem 文件夹，包 mem 模块作为内存管理相关代码的包名，然后在 mem 目下面创建 buf.go 文件，作为 Buf 的代码实现，文件路径结构如下：

```
zmem/
├── README.md
├── c/
│   ├── memory.go
│   └── memory_test.go
```

```
├── go.mod
└── mem/
    └── buf.go
```

接下来定义一个 Buf 数据结构，具体的定义如下：

```
//zmem/mem/buf.go

package mem

import "unsafe"

type Buf struct {
   //如果存在多个 buffer,则采用链表的形式连接起来
   Next  * Buf
   //当前 buffer 的缓存容量大小
   Capacity int
   //当前 buffer 的有效数据长度
   length int
   //未处理数据的头部位置索引
   head int
   //当前 buf 所保存的数据地址
   data unsafe.Pointer
}
```

一个 Buf 内存缓冲包含以下几个成员属性：

(1) Capacity，表示当前缓冲的容量大小，实则是底层内存分配的最大内存空间上限。

(2) length，当前缓冲区的有效数据长度，有效数据长度为用户存入但又未访问的剩余数据长度。

(3) head，缓冲中未处理的头部位置索引。

(4) data，当前 buf 所保存内存的首地址指针，这里用的是 unsafe.Pointer 类型，表示 data 所存放的为基础的虚拟内存地址。

(5) Next，Buf 类型的指针，指向下一个 Buf 地址。Buf 与 Buf 之间的关系是一个链表结构。

一个 Buf 的数据内存结构布局如图 3.13 所示。

Buf 采用链表的集合方式，每个 Buf 通过 Next 进行关联，其中 Data 为指向底层开辟出来供用户使用的内存。一个内存中有几个刻度索引，内存首地址索引位置定义为 0，Head 为当前用户应用有效数据的首地址索引，Length 为有效数据尾地址索引，有效数据的长度为“Length-Head”。Capacity 是开辟内存的尾地址索引，表示当前 Buf 的可使用内存容量。

接下来提供一个 Buf 的构造方法，具体的代码如下：

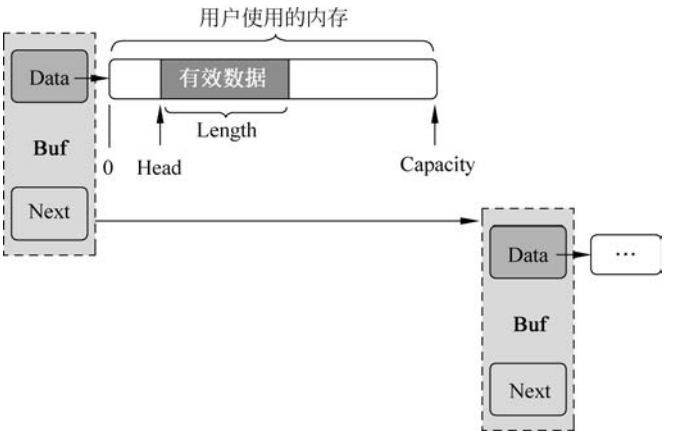

图 3.13 Buf 的数据内存结构布局

```
//zmem/mem/buf.go

//构造,创建一个 Buf 对象
func NewBuf(size int) *Buf {
   return &Buf{
      Capacity: size,
      length: 0,
      head: 0,
      Next: nil,
      data : c.Malloc(size),
   }
}
```

NewBuf()接收一个 size 形参,用来表示开辟的内存空间长度。这里调用封装的 c.Malloc()方法来申请 size 长度的内存空间,并且赋值给 data。

Buf 被初始化之后,需要给 Buf 赋予让调用方传入数据的接口,这里允许一个 Buf 的内存可以赋予[]Byte 类型的源数据,方法名称是 SetBytes(),定义如下:

```
//zmem/mem/buf.go

//给一个 Buf 填充[]Byte 数据
func (b *Buf) SetBytes(src []Byte) {
   c.Memcpy(unsafe.Pointer(uintptr(b.data) + uintptr(b.head)), src, len(src))
   b.length += len(src)
}
```

操作一共由两个过程组成:

(1) 将[]Byte 源数据 src 通过 C 接口的内存复制，给 Buf 的 data 赋值。这里需要注意的是被复制的 data 的起始地址是 b.head。

(2) 复制之后 Buf 的有效数据长度要相应地累加偏移，具体的过程如图 3.14 所示。

这里需要注意的是，复制的起始地址会基于 data 的基地址向右偏移 head 的长度，因为定义从 Head 到 Length 是有效的合法数据。对于 unsafe.Pointer 的地址偏移需要转换为 uintptr 类型进行地址计算。

与 SetBytes() 对应的是 GetBytes()，是从 Buf 的 data 中获取数据，具体实现代码如下：

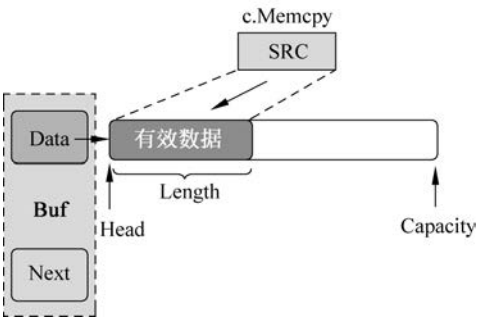

图 3.14　SetBytes 内存操作

```
//zmem/mem/buf.go

//获取一个 Buf 的数据，以[]Byte 形式展现
func (b *Buf) GetBytes() []Byte {
    data := C.GoBytes(unsafe.Pointer(uintptr(b.data) + uintptr(b.head)), C.int(b.length))
    return data
}
```

其中 C.GoBytes()是 Cgo 模块提供的将 C 数据转换为 Byte 数组的一个函数，并且可指定转换的长度。

取数据的起始地址依然是基于 data 进行 head 长度的偏移。

Buf 还需要提供一个 Copy()方法，用来将其他 Buf 缓冲对象直接复制到自身当中，并且 head、length 等与对方完全一样，具体实现的代码如下：

```
//zmem/mem/buf.go

//将其他 Buf 对象数据复制到自己中
func (b *Buf) Copy(other *Buf) {
    c.Memcpy(b.data, other.GetBytes(), other.length)
    b.head = 0
    b.length = other.length
}
```

接下来需要提供可以移动 head 的方法，其作用是缩小有效数据长度，当调用方已经使用了一部分数据之后，这部分数据可能会变成非法的非有效数据，所以就需要将 head 向后偏移，以便缩小有效数据的长度，Buf 将提供一个名字叫 Pop()的方法，具体定义如下：

```
//zmem/mem/buf.go

//处理长度为 len 的数据,移动 head 和修正 length
func (b *Buf) Pop(len int) {
   if b.data == nil {
     fmt.Printf("pop data is nil")
     return
   }
   if len > b.length {
     fmt.Printf("pop len > length")
     return
   }
   b.length -= len
   b.head += len
}
```

一次 Pop()操作,首先会判断弹出合法有效数据的长度是否越界,然后对应的 head 向右偏移,length 的有效长度相应地缩减,具体的流程如图 3.15 所示。

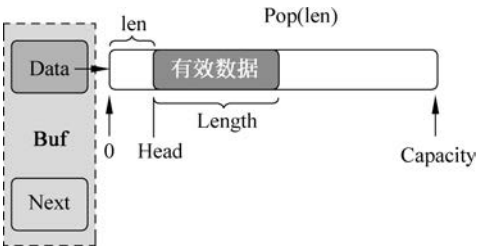

图 3.15 Pop 内存操作的 head 与 length 偏移

因为调用方经常地获取数据,然后调用 Pop()缩减有效长度,这样不出几次,可能就会导致 head 越来越接近 Capacity,也会导致有效数据之前的已经过期的非法数据越来越多,所以 Buf 需要提供一个 Adjust()方法,来将有效数据的内存迁移至 data 基地址位置,覆盖之前的已使用过的过期数据,将后续的空白可使用空间扩大。Adjust()的实现方法如下:

```
//zmem/mem/buf.go

//将已经处理过的数据清空,将未处理的数据提至数据首地址
func (b *Buf) Adjust() {
   if b.head != 0 {
     if (b.length != 0) {
         c.Memmove(b.data, unsafe.Pointer(uintptr(b.data) + uintptr(b.head)), b.length)
     }
     b.head = 0
   }
}
```

Adjust()调用之前封装好的 c.Memmove()方法,将有效数据内存平移至 Buf 的 data 基地址,同时将 head 重置到 0 位置,具体的流程如图 3.16 所示。

Buf 也要提供一个清空缓冲内存的方法 Clear()，Clear()实现起来很简单，只需将几个索引值清零，Clear()并不会以操作系统层面回收内存，因为 Buf 的是否回收及是否被重置等需要依赖 BufPool 内存池来管理，将在 3.4.3 节介绍内存池管理 Buf 的情况。为了降低系统内存的开辟和回收，Buf 可能长期在内存池中存在。调用方只需改变几个地址索引值便可以达到内存的使用和回收。Clear()方法的实现如下：

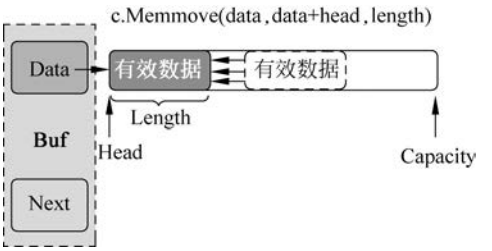

图 3.16　Adjust 操作的内存平移

```
//zmem/mem/buf.go

//清空数据
func (b *Buf) Clear() {
   b.length = 0
   b.head = 0
}
```

其他的提供的访问 head 和 length 的方法如下：

```
func (b *Buf) Head() int {
   return b.head
}

func (b *Buf) Length() int {
   return b.length
}
```

现在 Buf 的基本功能已经实现了，接下来实现对 Buf 的管理内存池模块。

3.4.3　内存池设计与实现

一个 Buf 只是一次内存使用所需要存放数据的缓冲空间，为了方便多个 Buf 直接申请与管理，则需要设计一个内存池来统一进行 Buf 的调配。

内存池的设计是预开辟内存，就是在首次申请创建内存池的时候，就将池子里全部可以被使用的 Buf 内存空间集合一并申请开辟出来。调用方在申请内存的时候，是通过内存池来申请的，内存池从 Buf 集合中选择未被使用或未被占用的 Buf 返给调用方。调用方在使用完 Buf 之后将 Buf 退还给内存池。这样调用方即使频繁地申请和回收小空间的内存也不会出现系统频繁地调用申请物理内存空间，降低了内存动态开辟的开销成本，业务方的内存访问速度也会有很大的提升。

下面实现内存池 BufPool，首先在 zmem/mem/目录下创建 buf_pool.go 文件，在当前文件实现 BufPool 内存池的功能，以及 BufPool 的数据结构，代码如下：

```
//zmem/mem/buf_pool.go
package mem

import (
   "sync"
)

//内存管理池类型
type Pool map[int] *Buf

//Buf 内存池
type BufPool struct {
   //所有 buffer 的一个 map 集合句柄
   Pool Pool
   PoolLock sync.RWMutex

   //总 buffer 池的内存大小 单位为 KB
   TotalMem uint64
}
```

首先定义 Pool 数据类型，该类型表示管理全部 Buf 的 Map 集合，其中 Key 表示当前一组 Buf 的 Capacity 容量，Value 则是一个 Buf 链表。每个 Key 下面挂载着相同 Capacity 的 Buf 集合链表，其实是 BufPool 的成员属性，定义如下：

（1）Pool，当前内存池全部的 Buf 缓冲对象集合，是一个 Map 数据结构。

（2）PoolLock，对 Map 读写并发安全的读写锁。

（3）TotalMem，当前 BufPool 所开辟内存池申请虚拟内存的总容量。

接下来提供 BufPool 的初始化构造函数方法，BufPool 作为内存池，全局应该设计成唯一，所以采用单例模式设计，下面定义公共方法 MemPool()，用来初始化并且获取 BufPool 单例对象，具体的实现方式如下：

```
//zmem/mem/buf_pool.go

//单例对象
var bufPoolInstance *BufPool
var once sync.Once

//获取 BufPool 对象(单例模式)
func MemPool() *BufPool{
   once.Do(func() {
```

```
        bufPoolInstance = new(BufPool)
        bufPoolInstance.Pool = make(map[int] * Buf)
        bufPoolInstance.TotalMem = 0
        bufPoolInstance.prev = nil
        bufPoolInstance.initPool()
    })

    return bufPoolInstance
}
```

全局遍历指针 bufPoolInstance 作为指向 BufPool 单例实例的唯一指针，通过 Go 语言标准库提供 sync.Once 来只执行依次的 Do()方法，以此来初始化 BufPool。在将 BufPool 成员均赋值完之后，最后通过 initPool()方法来初始化内存池的内存申请布局。

内存申请 initPool()会将内存分配结构，如图 3.17 所示。BufPool 会预先将所有要管理的 Buf 按照内存刻度大小进行分组，如 4KB 一组或 16KB 一组等。容量越小的 Buf，所管理的 Buf 链表的数量越多，容量越大的 Buf 数量则越少。全部的 Buf 关系通过 Map 数据结构来管理，由于 Buf 本身是链表数据结构，所以每个 Key 所对应的 Value 只需保存头节点 Buf 信息，之后的 Buf 可以通过 Buf 的 Next 指针找到。

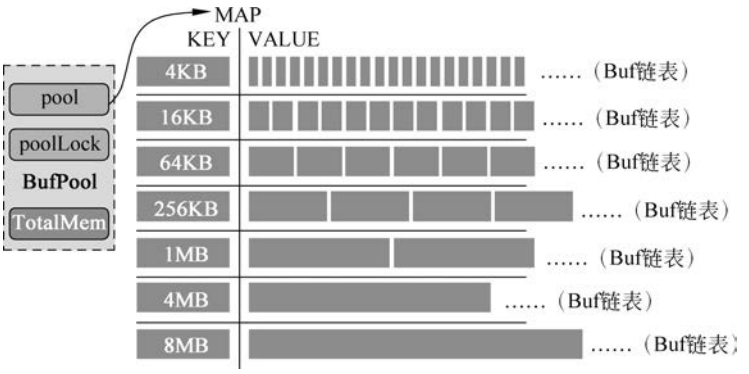

图 3.17　BufPool 内存池的内存管理布局

BufPool 的 initPool()初始化内存方法的具体实现如下：

```
//zmem/mem/buf_pool.go

const (
    m4K int = 4096
    m16K int = 16384
    m64K int = 655535
    m256K int = 262144
    m1M int = 1048576
```

```
    m4M int = 4194304
    m8M int = 8388608
)

/*
    初始化内存池 主要是预先开辟一定量的空间
  这里 BufPool 是一个 hash,每个 key 都是不同空间容量
  对应的 value 是一个 Buf 集合的链表

BufPool --> [m4K] -- Buf - Buf - Buf - Buf...(BufList)
            [m16K] -- Buf - Buf - Buf - Buf...(BufList)
            [m64K] -- Buf - Buf - Buf - Buf...(BufList)
            [m256K] -- Buf - Buf - Buf - Buf...(BufList)
            [m1M] -- Buf - Buf - Buf - Buf...(BufList)
            [m4M] -- Buf - Buf - Buf - Buf...(BufList)
            [m8M] -- Buf - Buf - Buf - Buf...(BufList)
 */
func (bp *BufPool) initPool() {
    //----> 开辟 4KB buf 内存池
    //4KB 的 Buf 预先开辟 5000 个,约 20MB 供开发者使用
    bp.makeBufList(m4K, 5000)

    //----> 开辟 16KB buf 内存池
    //16KB 的 Buf 预先开辟 1000 个,约 16MB 供开发者使用
    bp.makeBufList(m16K, 1000)

    //----> 开辟 64KB buf 内存池
    //64KB 的 Buf 预先开辟 500 个,约 32MB 供开发者使用
    bp.makeBufList(m64K, 500)

    //----> 开辟 256KB buf 内存池
    //256KB 的 Buf 预先开辟 200 个,约 50MB 供开发者使用
    bp.makeBufList(m256K, 200)

    //----> 开辟 1MB buf 内存池
    //1MB 的 Buf 预先开辟 50 个,约 50MB 供开发者使用
    bp.makeBufList(m1M, 50)

    //----> 开辟 4MB buf 内存池
    //4MB 的 Buf 预先开辟 20 个,约 80MB 供开发者使用
    bp.makeBufList(m4M, 20)

    //----> 开辟 8MB buf 内存池
    //8MB 的 io_buf 预先开辟 10 个,约 80MB 供开发者使用
    bp.makeBufList(m8M, 10)
}
```

其中 makeBufList()为每次初始化一种刻度容量的 Buf 链表,代码如下:

```
//zmem/mem/buf_pool.go

func (bp * BufPool) makeBufList(cap int, num int) {
   bp.Pool[cap] = NewBuf(cap)

   var prev * Buf
   prev = bp.Pool[cap]
   for i := 1; i < num; i ++{
      prev.Next = NewBuf(cap)
      prev = prev.Next
   }
   bp.TotalMem += (uint64(cap)/1024) * uint64(num)
}
```

每次创建一行 BufList 之后,BufPool 内存池的 TotalMem 就对应增加相应申请内存的容量,这个属性就作为当前内存池已经从操作系统获取的内存总容量为多少。

现在 BufPool 已经具备了申请首次初始化内存池的能力,还应该提供从 BufPool 获取一个 Buf 内存的接口,同时需要当调用方使用完后,再将内存退还给 BufPool 的接口。

1. 获取 Buf

下面定义 Alloc()方法来标识从 BufPool 中申请一个可用的 Buf 对象,代码如下:

```
//zmem/mem/buf_pool.go

package mem

import (
   "errors"
   "fmt"
   "sync"
)
const (
   //总内存池最大限制 单位是 KB,所以目前的限制是 5GB
   EXTRA_MEM_LIMIT int = 5 * 1024 * 1024
)

/*
   开辟一个 Buf
 */
func (bp * BufPool) Alloc(N int) ( * Buf, error) {
   //1 找到 N 最接近哪个 hash 组
   var index int
```

```
    if N <= m4K {
      index = m4K
    } else if (N <= m16K) {
      index = m16K
    } else if (N <= m64K) {
      index = m64K
    } else if (N <= m256K) {
      index = m256K
    } else if (N <= m1M) {
      index = m1M
    } else if (N <= m4M) {
      index = m4M
    } else if (N <= m8M) {
      index = m8M
    } else {
      return nil, errors.New("Alloc size Too Large!");
    }

    //2 如果该组已经没有,则需要额外申请,所以需要加锁保护
    bp.PoolLock.Lock()
    if bp.Pool[index] == nil {
      if (bp.TotalMem + uint64(index/1024)) >= uint64(EXTRA_MEM_LIMIT) {
        errStr := fmt.Sprintf("already use too many memory!\n")
        return nil, errors.New(errStr)
      }

      newBuf := NewBuf(index)
      bp.TotalMem += uint64(index/1024)
      bp.PoolLock.Unlock()
      fmt.Printf("Alloc Mem Size: %d KB\n", newBuf.Capacity/1024)
      return newBuf, nil
    }

    //3 如果该组有 Buf 内存存在,则得到一个 Buf 并返回,并且从 pool 中移除该内存块
    targetBuf := bp.Pool[index]
    bp.Pool[index] = targetBuf.Next
    bp.TotalMem -= uint64(index/1024)
    bp.PoolLock.Unlock()
    targetBuf.Next = nil
    fmt.Printf("Alloc Mem Size: %d KB\n", targetBuf.Capacity/1024)
    return targetBuf, nil
}
```

Alloc()函数有 3 个关键步骤：

(1) 如果上层需要 N 字节大小的空间,找到与 N 最接近的 Buf 链表集合,从当前 Buf

集合取出。

（2）如果该组已经没有节点可供使用，则可以额外申请总申请长度不能够超过最大的限制大小 EXTRA_MEM_LIMIT。

（3）如果有该节点需要的内存块，则直接取出，并且将该内存块从 BufPool 移除。

2. 退还 Buf

定义 Revert()方法，为将使用后的 Buf 退还给 BufPool 内存池，代码如下：

```
//当 Alloc 之后，当前 Buf 被使用完，需要重置这个 Buf，并且需要将该 buf 放回 pool 中
func (bp * BufPool) Revert(buf * Buf) error {,
   //每个 buf 的容量都是固定的，在 hash 的 key 中取值
   index : = buf.Capacity
   //重置 buf 中的内置位置指针
   buf.Clear()

   bp.PoolLock.Lock()
   //找到对应的 hash 组 buf 首节点地址
   if _, ok : = bp.Pool[index]; !ok {
     errStr : = fmt.Sprintf("Index % d not in BufPoll!\n", index)
     return errors.New(errStr)
   }

   //将 buffer 插回链表头部
   buf.Next = bp.Pool[index]
   bp.Pool[index] = buf
   bp.TotalMem += uint64(index/1024)
   bp.PoolLock.Unlock()
   fmt.Printf("Revert Mem Size: % d KB\n",index/1024)

   return nil
}
```

Revert()会根据当前 Buf 的 Capacity 找到对应的 Hash 刻度，然后将 Buf 插入链表的头部，在插入之前通过 Buf 的 Clear()将 Buf 的全部有效数据清空。

3.4.4　内存池的功能单元测试

接下来对上述接口做一些单元测试，在 zmem/mem/目录下创建 buf_test.go 文件。

1. TestBufPoolSetGet

首先测试基本的 SetBytes()和 GetBytes()方法，单元测试代码如下：

```
//zmem/mem/buf_test.go

package mem_test
```

```
import (
   "zmem/mem"
   "fmt"
   "testing"
)

func TestBufPoolSetGet(t * testing.T) {
   pool : = mem.MemPool()

   buffer, err : = pool.Alloc(1)
   if err != nil {
     fmt.Println("pool Alloc Error ", err)
     return
   }

   buffer.SetBytes([]Byte("Aceld12345"))
   fmt.Printf("GetBytes = % + v, ToString = % s\n", buffer.GetBytes(), string(buffer.
GetBytes()))
   buffer.Pop(4)
   fmt.Printf("GetBytes = % + v, ToString = % s\n", buffer.GetBytes(), string(buffer.
GetBytes()))
}
```

单元测试用例首先申请一个内存 buffer，接着设置"Aceld12345"内容，然后输出日志，接下来弹出有效数据 4 字节，再打印 buffer 可以访问的合法数据，最后执行单元测试代码，指令如下：

```
$ go test - run TestBufPoolSetGet
Alloc Mem Size: 4 KB
GetBytes = [65 99 101 108 100 49 50 51 52 53], ToString = Aceld12345
GetBytes = [100 49 50 51 52 53], ToString = d12345
PASS
ok     zmem/mem       0.010s
```

通过上述结果可得出通过 Pop(4)之后，已经弹出了 Acel 前 4 字节数据。

2. TestBufPoolCopy

接下来测试 Buf 的 Copy()赋值方法，具体的代码如下：

```
//zmem/mem/buf_test.go

package mem_test

import (
```

```
    "zmem/mem"
    "fmt"
    "testing"
)

func TestBufPoolCopy(t * testing.T) {
    pool := mem.MemPool()

    buffer, err := pool.Alloc(1)
    if err != nil {
        fmt.Println("pool Alloc Error ", err)
        return
    }

    buffer.SetBytes([]Byte("Aceld12345"))
    fmt.Printf("Buffer GetBytes = %+v\n", string(buffer.GetBytes()))

    buffer2, err := pool.Alloc(1)
    if err != nil {
        fmt.Println("pool Alloc Error ", err)
        return
    }
    buffer2.Copy(buffer)
    fmt.Printf("Buffer2 GetBytes = %+v\n", string(buffer2.GetBytes()))
}
```

将 buffer 复制的 buffer2 中，查看 buffer 存放的数据内容，执行单元测试指令和所得到的结果如下：

```
$ go test -run TestBufPoolCopy
Alloc Mem Size: 4KB
Buffer GetBytes = Aceld12345
Alloc Mem Size: 4KB
Buffer2 GetBytes = Aceld12345
PASS
ok     zmem/mem       0.008s
```

3. TestBufPoolAdjust

之后针对 Buf 的 Adjust()方法进行单元测试，相关代码如下：

```
//zmem/mem/buf_test.go

package mem_test
```

```
import (
   "zmem/mem"
   "fmt"
   "testing"
)

func TestBufPoolAdjust(t * testing.T) {
   pool := mem.MemPool()

   buffer, err := pool.Alloc(4096)
   if err != nil {
      fmt.Println("pool Alloc Error ", err)
      return
   }

   buffer.SetBytes([]Byte("Aceld12345"))
   fmt.Printf("GetBytes = %+v, Head = %d, Length = %d\n", buffer.GetBytes(), buffer.
Head(), buffer.Length())
   buffer.Pop(4)
   fmt.Printf("GetBytes = %+v, Head = %d, Length = %d\n", buffer.GetBytes(), buffer.
Head(), buffer.Length())
   buffer.Adjust()
   fmt.Printf("GetBytes = %+v, Head = %d, Length = %d\n", buffer.GetBytes(), buffer.
Head(), buffer.Length())
}
```

首先 buffer 被填充为"Aceld12345",然后打印 Head 索引和 Length 长度,接着通过 Pop 弹出有效数据 4 字节,继续打印日志,再通过 Adjust()重置 Head,最后输出 buffer 信息,通过指令执行单元测试和得到的结果如下:

```
$ go test -run TestBufPoolAdjust
Alloc Mem Size: 4 KB
GetBytes = [65 99 101 108 100 49 50 51 52 53], Head = 0, Length = 10
GetBytes = [100 49 50 51 52 53], Head = 4, Length = 6
GetBytes = [100 49 50 51 52 53], Head = 0, Length = 6
PASS
ok     zmem/mem       0.009s
```

可以看出第三次输出的日志 Head 已经被重置为 0,并且 GetBytes()得到的有效数据没有改变。

3.4.5 内存管理应用接口

3.4.1 节～3.4.4 节已经基本实现了一个简单的内存池管理,但如果希望更方便地使

用，则需要对 Buf 和 BufPool 再做一层封装，这里首先定义新数据结构 Zbuf，然后对 Buf 的基本操作进行封装，使内存管理的接口更加友好，在 zmem/mem/目录下创建 zbuf.go 文件，定义数据类型 Zbuf，具体的代码如下：

```
//zmem/mem/zbuf.go

package mem

//应用层的 buffer 数据
type ZBuf struct {
   b *Buf
}
```

接下来定义 Zbuf 对外提供的一些使用方法。

1. Clear()方法

Zbuf 的 Clear()方法实则是将 ZBuf 中的 Buf 退还给 BufPool，具体的代码如下：

```
//zmem/mem/zbuf.go

//清空当前的 ZBuf
func (zb *ZBuf) Clear() {
   if zb.b != nil {
     //将 Buf 重新放回 buf_pool 中
     MemPool().Revert(zb.b)
     zb.b = nil
   }
}
```

在 Buf 的 Clear()中调用了 MemPool()的 Revert()方法，回收了当前 Zbuf 中的 Buf 对象。

2. Pop()方法

Zbuf 的 Pop()方法对之前的 Pop 进行了一些安全性越界校验，具体的代码如下：

```
//zmem/mem/zbuf.go

//弹出已使用的有效长度
func (zb *ZBuf) Pop(len int) {
   if zb.b == nil || len > zb.b.Length() {
     return
   }

   zb.b.Pop(len)
```

```
    //当此时 Buf 的可用长度已经为 0 时,将 Buf 重新放回 BufPool 中
    if zb.b.Length() == 0 {
      MemPool().Revert(zb.b)
      zb.b = nil
    }
  }
```

如果 Buf 在 Pop()之后的有效数据长度为 0,就将当前 Buf 退还给 BufPool。

3. Data()方法

Zbuf 的 Data()方法用于返回 Buf 的数据,代码如下:

```
//zmem/mem/zbuf.go

//获取 Buf 中的数据
func (zb *ZBuf) Data() []Byte {
  if zb.b == nil {
    return nil
  }
  return zb.b.GetBytes()
}
```

4. Adjust()方法

Zbuf 的 Adjust()方法的封装没有任何改变,代码如下:

```
//zmem/mem/zbuf.go

//重置缓冲区
func (zb *ZBuf) Adjust() {
  if zb.b != nil {
    zb.b.Adjust()
  }
}
```

5. Read()方法

Zbuf 的 Read()方法是将数据填充到 Zbuf 的 Buf 中。Read()方法是将被填充的数据作为形参[]Byte 传递进来,代码如下:

```
//zmem/mem/zbuf.go

//将数据读取到 Buf 中
func (zb *ZBuf) Read(src []Byte) (err error){
  if zb.b == nil {
```

```
        zb.b, err = MemPool().Alloc(len(src))
        if err != nil {
          fmt.Println("pool Alloc Error ", err)
        }
      } else {
        if zb.b.Head() != 0 {
          return nil
        }
        if zb.b.Capacity - zb.b.Length() < len(src) {
          //不够存放,重新从内存池申请
          newBuf, err := MemPool().Alloc(len(src) + zb.b.Length())
          if err != nil {
            return nil
          }
          //将之前的 Buf 复制到新申请的 Buf 中
          newBuf.Copy(zb.b)
          //将之前的 Buf 回收到内存池中
          MemPool().Revert(zb.b)
          //新申请的 Buf 成为当前的 ZBuf
          zb.b = newBuf
        }
      }

      //将内容写进 ZBuf 缓冲中
      zb.b.SetBytes(src)

      return nil
    }
```

如果当前 Zbuf 的 Buf 为空,则会向 BufPool 申请内存。如果传递的源数据超过了当前 Buf 所能承载的容量,则 Zbuf 会申请一个更大的 Buf,将之前的已有的数据通过 Copy()方法复制到新申请的 Buf 中,之后将之前的 Buf 退还给 BufPool。

6. 其他可拓展方法

上述的 Read()方法代表 Zbuf 从参数获取源数据,如果为了更方便地填充 Zbuf,可以封装类似接口,如在 Fd 文件描述符中将数据读取到 Zbuf 中、从文件将数据读取到 Zbuf 中、从网络套接字将数据读取到 Zbuf 中等,相关函数原型的代码如下:

```
//zmem/mem/zbuf.go

//从 Fd 文件描述符中读取数据
func (zb *ZBuf) ReadFromFd(fd int) error {
  //...
```

```
    return nil
}

//将数据写入 Fd 文件描述符中
func (zb * ZBuf) WriteToFd(fd int) error {
    //...
    return nil
}

//从文件中读取数据
func (zb * ZBuf) ReadFromFile(path string) error {
    //...
    return nil
}

func (zb * ZBuf) WriteToFile(path string) error {
    //...
    return nil
}

//从网络连接中读取数据
func (zb * ZBuf) ReadFromConn(conn net.Conn) error {
    //...
    return nil
}

func (zb * ZBuf) WriteToConn(conn net.Conn) error {
    //...
    return nil
}
```

这里就不一一展开了，具体实现方式和 Read()方法类似。这样 Zbuf 就可以通过不同的媒介来填充 Buf 并且使用，业务层只需面向 Zbuf 就可以获取数据，无须关心具体的 I/O 层逻辑。

3.5 Go 语言内存管理之魂 TCMalloc

在了解 Go 语言的内存管理之前，一定要了解基本的申请内存模式，即 TCMalloc (Thread Cache Malloc)。Go 语言的内存管理就是基于 TCMalloc 的核心思想来构建的。本节将介绍 TCMalloc 的基础理念和结构。

3.5.1　TCMalloc

TCMalloc 最大的优势就是每个线程都会独立维护自己的内存池。在之前章节介绍的自定义实现的 Go 语言内存池版 BufPool 实则是所有 Goroutine 或者所有线程共享的内存池，其关系如图 3.18 所示。

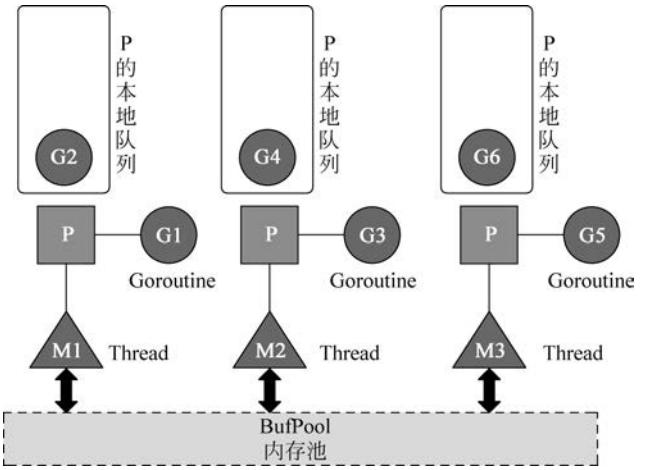

图 3.18　BufPool 内存池与线程 Thread 的关系

这种内存池的设计缺点显而易见，应用方全部的内存申请均需要和全局的 BufPool 交互，为了线程的并发安全，BufPool 频繁地申请内存和退还内存需要加互斥和同步机制，这样会影响内存的使用性能。

TCMalloc 则是为每个 Thread 预分配一块缓存，每个 Thread 在申请内存时首先会从这个缓存区 ThreadCache 申请，并且所有 ThreadCache 缓存区还共享一个叫作 CentralCache 的中心缓存。这里假设目前 Go 语言的内存管理用的是原生 TCMalloc 模式，此种情况线程与内存的关系如图 3.19 所示。

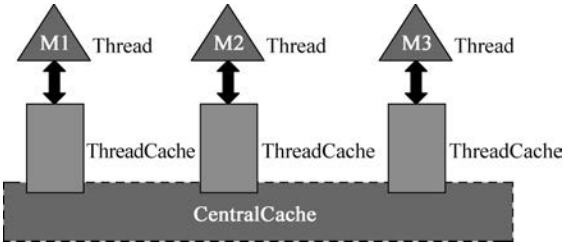

图 3.19　TCMalloc 内存池与线程 Thread 的关系

这样做的好处其一是 ThreadCache 作为每个线程独立的缓存，能够明显地提高 Thread 获取高命中的数据，其二是 ThreadCache 从堆空间一次性申请，即只触发一次系统调用。

每个 ThreadCache 还会共同访问 CentralCache，这个与 BufPool 类似，但是设计得更为精细一些。CentralCache 是所有线程共享的缓存，当 ThreadCache 的缓存不足时，就会从 CentralCache 获取，当 ThreadCache 的缓存充足或者过多时，则会将内存退还给 CentralCache，但是 CentralCache 由于共享，所以访问一定需要加锁。ThreadCache 作为线程独立的第一交互内存，访问无须加锁，CentralCache 则作为 ThreadCache 的临时补充缓存。

TCMalloc 的构造不仅于此，提供的 ThreadCache 和 CentralCache 可以解决小对象内存块的申请，但是对于大块内存 Cache 显然是不适合的。TCMalloc 将内存分为三类，如表 3.4 所示。

表 3.4　TCMalloc 的内存分类

对　　象	容　　量
小对象	(0,256KB]
中对象	(256KB,1MB]
大对象	(1MB,+∞)

所以为了解决中对象和大对象的内存申请，TCMalloc 依然有一个全局共享内存堆 PageHeap，如图 3.20 所示。

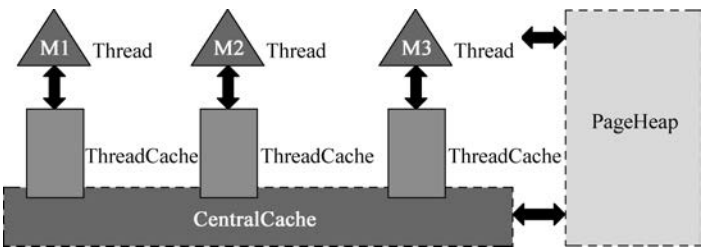

图 3.20　TCMalloc 中的 PageHeap

PageHeap 也是通过一次系统调用从虚拟内存中申请的，PageHeap 很明显是全局的，所以访问时一定要加锁。其作用是当 CentralCache 没有足够内存时会从 PageHeap 获取，当 CentralCache 内存过多或者充足时，则将低命中内存块退还 PageHeap。如果 Thread 需要大对象申请超过 Cache 容纳的内存块单元，则会直接从 PageHeap 获取。

3.5.2　TCMalloc 模型相关基础结构

在讲解 TCMalloc 的一些内部设计结构时，首先要了解的是 TCMalloc 定义的一些基本名词，如 Page、Span 和 Size Class。

1. Page

TCMalloc 中的 Page 与之前章节介绍操作系统对虚拟内存管理的 MMU 定义的物理页有相似的定义，TCMalloc 将虚拟内存空间划分为多份同等大小的 Page，每个 Page 默认为 8KB。

对于 TCMalloc 来讲，虚拟内存空间的全部内存都按照 Page 的容量被分成均等份，并且给每份 Page 标记了 ID 编号，如图 3.21 所示。

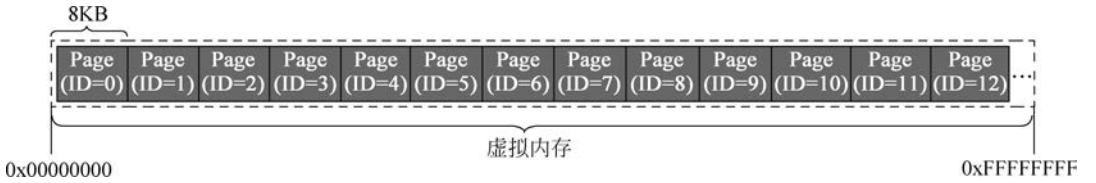

图 3.21　TCMalloc 将虚拟内存平均分成多份 Page

对 Page 进行编号的好处是，可以根据任意内存的地址指针，进行固定算法偏移计算，以此算出所在的 Page。

2. Span

多个连续的 Page 称为是一个 Span，其含义与操作系统管理的页表相似，Page 与 Span 的关系如图 3.22 所示。

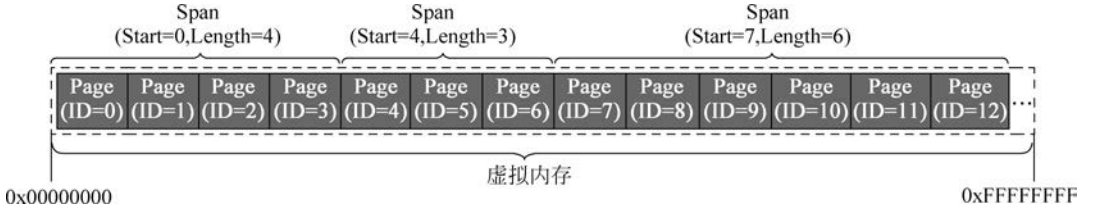

图 3.22　TCMalloc 中 Page 与 Span 的关系

TCMalloc 以 Span 为单位向操作系统申请内存。每个 Span 记录了第 1 个起始 Page 的编号 Start 和一共有多少个连续 Page 的数量 Length。

为了方便 Span 和 Span 之间的管理，Span 集合以双向链表的形式构建，如图 3.23 所示。

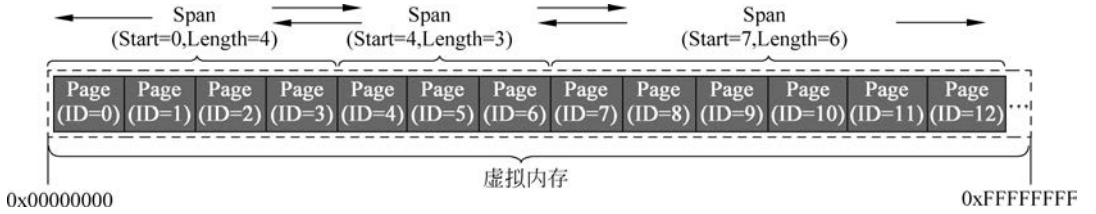

图 3.23　TCMalloc 中 Span 的存储形式

3. Size Class

参考表 3.4，对于 256KB 以内的小对象，TCMalloc 会将这些小对象集合划分成多个内存刻度[①]，同属于一个刻度类别下的内存集合称为一个 Size Class。这与之前章节自定义实

① TCMalloc 官方文档称一共划分 88 个 size-classes，Each small object size maps to one of approximately 88 allocatable size-classes，参考 TCMalloc ：Thread-Caching Malloc，https://gperftools.github.io/gperftools/tcmalloc.html。

现的内存池类似，即将 Buf 划分为多个刻度的 BufList。

每个 Size Class 都对应一个大小，例如 8 字节、16 字节、32 字节等。在申请小对象内存的时候，TCMalloc 会根据使用方申请的空间大小就近向上取最接近的一个 Size Class 的 Span（由多个等空间的 Page 组成）内存块返给使用方。

如果将 Size Class、Span、Page 用一张图来表示，则具体的抽象关系如图 3.24 所示。

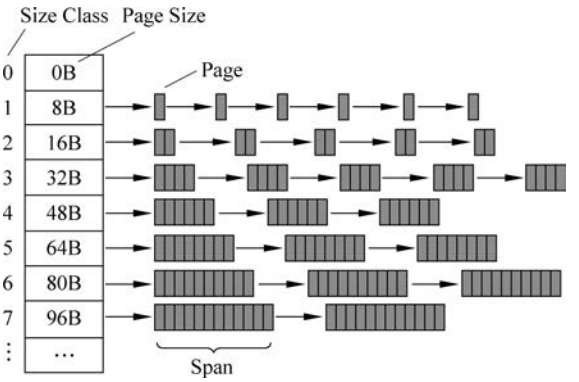

图 3.24　TCMalloc 中 Size Class、Page、Span 的结构关系

接下来剖析一下 ThreadCache、CentralCache、PageHeap 的内存管理结构。

3.5.3　ThreadCache

在 TCMalloc 中每个线程都会有一份单独的缓存，即 ThreadCache。ThreadCache 中对于每个 Size Class 都会有一个对应的 FreeList，FreeList 表示当前缓存中还有多少个空闲的内存可用，具体的结构布局如图 3.25 所示。

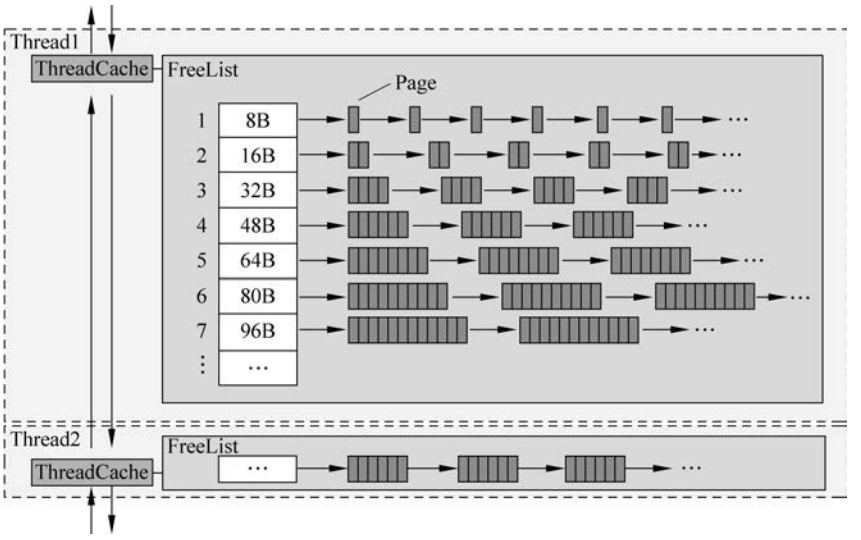

图 3.25　TCMalloc 中的 ThreadCache

使用方对于从 TCMalloc 申请的小对象，会直接从 ThreadCache 获取，实则是从 FreeList 中返回一个空闲的对象，如果对应的 Size Class 刻度下已经没有空闲的 Span 可以被获取，则 ThreadCache 会从 CentralCache 中获取。当使用方使用完内存之后，归还时也是直接归还给当前的 ThreadCache 中对应刻度下的 FreeList。

整条申请和归还的流程不需要加锁，因为 ThreadCache 为当前线程独享，但如果 ThreadCache 不够用，则需要从 CentralCache 申请内存，这个动作需要加锁。不同 Thread 之间的 ThreadCache 是以双向链表的结构进行关联，这是为了方便 TCMalloc 进行统计和管理。

3.5.4　CentralCache

CentralCache 由各个线程共用，所以与 CentralCache 获取内存交互时需要加锁。CentralCache 缓存的 Size Class 和 ThreadCache 的 Size Class 一样，这些缓存都被放在 CentralFreeList 中，当 ThreadCache 中的某个 Size Class 刻度下的缓存小对象不够用时，就会向 CentralCache 对应的 Size Class 刻度的 CentralFreeList 获取，同样地，如果 ThreadCache 有多余的缓存对象，则会退还给响应的 CentralFreeList，流程和关系如图 3.26 所示。

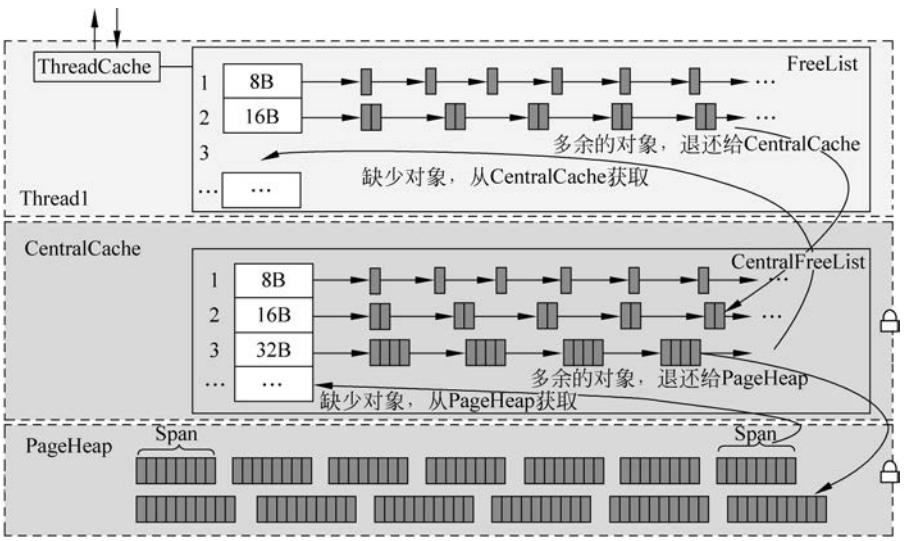

图 3.26　TCMalloc 中的 CentralCache

CentralCache 与 PageHeap 的角色关系与 ThreadCache 与 CentralCache 的角色关系相似，当 CentralCache 出现 Span 不足时，会从 PageHeap 申请 Span，以及将不再使用的 Span 退还给 PageHeap。

3.5.5　PageHeap

PageHeap 是提供 CentralCache 的内存来源。PageHeap 与 CentralCache 不同的是，

CentralCache 是与 ThreadCache 布局一模一样的缓存，主要针对 ThreadCache 的一层二级缓存起作用，并且只支持小对象内存分配，而 PageHeap 则是针对 CentralCache 的三级缓存。弥补对于中对象内存和大对象内存的分配，PageHeap 是直接和操作系统虚拟内存衔接的一层缓存，当 ThreadCache、CentralCache、PageHeap 都找不到合适的 Span 时，PageHeap 则会调用操作系统的内存申请系统的调用函数来从虚拟内存的堆区中取出内存填充到 PageHeap 中，具体的结构如图 3.27 所示。

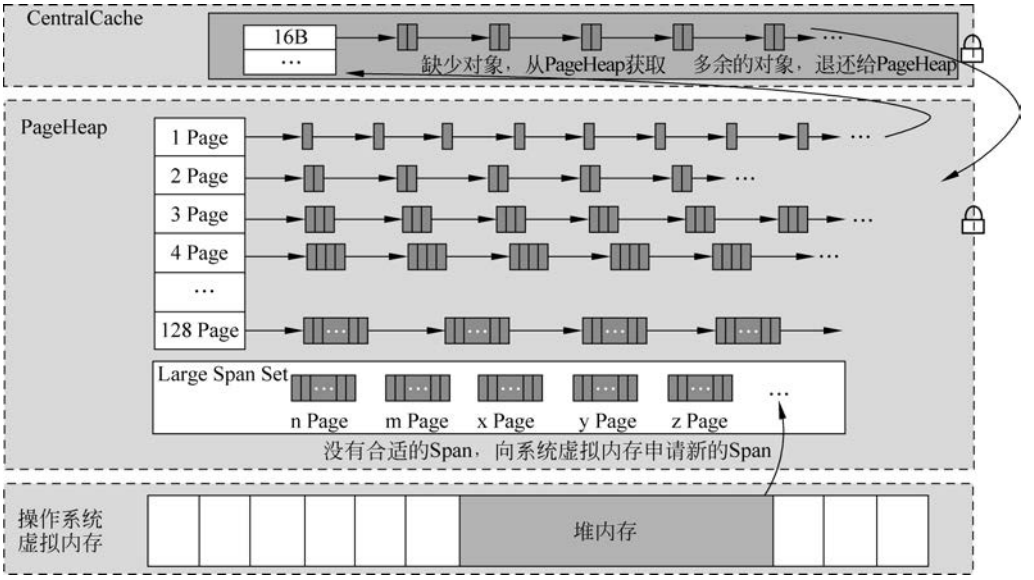

图 3.27 TCMalloc 中 PageHeap

PageHeap 内部的 Span 管理，采用两种不同的方式，对于 128 个 Page 以内的 Span 申请，每个 Page 刻度都会用一个链表形式的缓存来存储。对于 128 个 Page 以上的内存申请，PageHeap 以有序集合(C++标准库 STL 中的 Std::Set 容器)来存放。

3.5.6 TCMalloc 的小对象分配

至此，已经介绍了 TCMalloc 的几种基础结构，接下来总结一下 TCMalloc 针对小对象、中对象和大对象的分配流程，小对象分配流程如图 3.28 所示。

小对象为占用内存小于或等于 256KB 的内存，参考图 3.28 中的流程，下面将介绍详细的流程：

(1) Thread 用户线程应用逻辑申请内存，当前 Thread 访问对应的 ThreadCache 获取内存，此过程不需要加锁。

(2) ThreadCache 得到申请内存的 SizeClass(一般向上取整，大于或等于申请的内存大小)，通过 SizeClass 索引去请求自身对应的 FreeList。

(3) 判断得到的 FreeList 是否为非空。

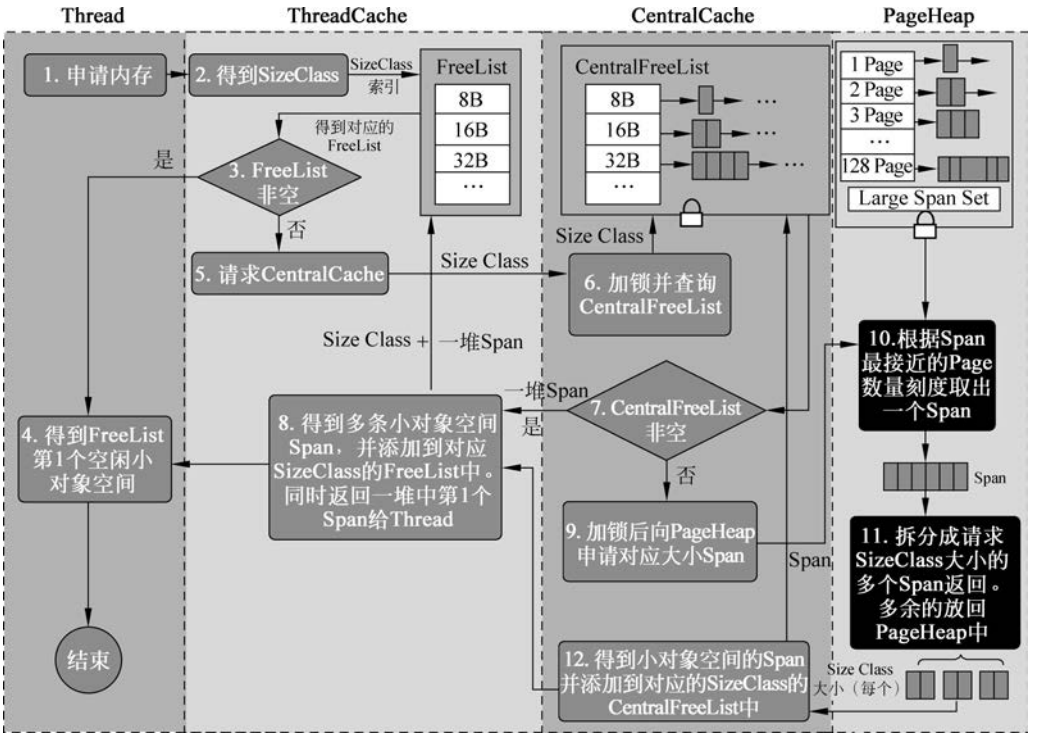

图 3.28　TCMalloc 小对象分配流程

（4）如果 FreeList 为非空，则表示目前有对应内存空间供 Thread 使用，得到 FreeList 第 1 个空闲 Span 后返给 Thread 用户逻辑，流程结束。

（5）如果 FreeList 为空，则表示目前没有对应 SizeClass 的空闲 Span 可使用，请求 CentralCache 并告知 CentralCache 具体的 SizeClass。

（6）CentralCache 收到请求后，加锁访问 CentralFreeList，根据 SizeClass 进行索引找到对应的 CentralFreeList。

（7）判断得到的 CentralFreeList 是否为非空。

（8）如果 CentralFreeList 为非空，则表示目前有空闲的 Span 可使用。返回多个 Span，将这些 Span（除了第 1 个 Span）放置于 ThreadCache 的 FreeList 中，并且将第 1 个 Span 返给 Thread 用户逻辑，流程结束。

（9）如果 CentralFreeList 为空，则表示目前没有可用的 Span，向 PageHeap 申请对应大小的 Span。

（10）PageHeap 得到 CentralCache 的申请，加锁请求对应的 Page 刻度的 Span 链表。

（11）PageHeap 将得到的 Span 根据本次流程请求的 SizeClass 大小为刻度进行拆分，分成 N 份 SizeClass 大小的 Span 返给 CentralCache，如果有多余的 Span，则放回 PageHeap 对应 Page 的 Span 链表中。

(12) CentralCache 得到对应的 N 个 Span，添加至 CentralFreeList 中，跳转至第(8)步。

综上是 TCMalloc 一次申请小对象的全部详细流程，接下来分析中对象的分配流程。

3.5.7 TCMalloc 的中对象分配

中对象为大于 256KB 且小于或等于 1MB 的内存。对于中对象申请分配的流程与处理小对象分配有一定的区别。对于中对象分配，Thread 不再按照小对象的流程路径向 ThreadCache 获取，而是直接从 PageHeap 获取，具体的流程如图 3.29 所示。

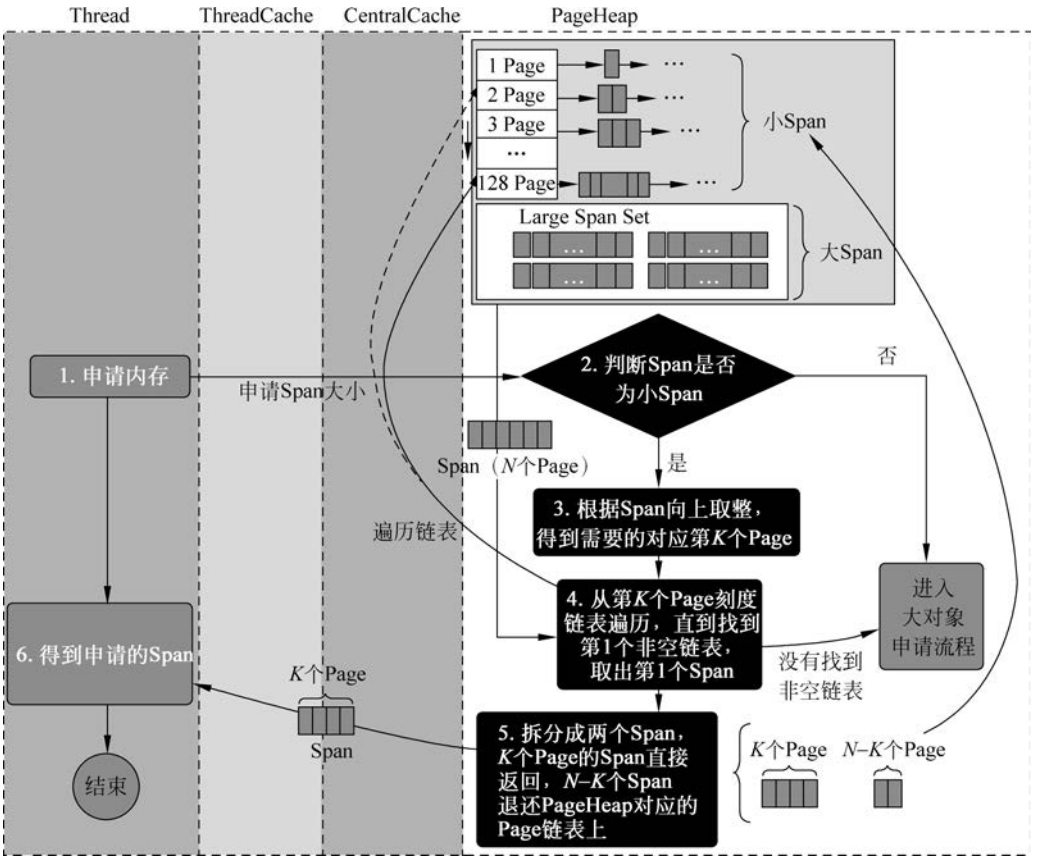

图 3.29 TCMalloc 中对象分配流程

PageHeap 将 128 个 Page 以内大小的 Span 定义为小 Span，将 128 个 Page 以上大小的 Span 定义为大 Span。由于一个 Page 为 8KB，所以 128 个 Page 为 1MB，对于中对象的申请，PageHeap 均按照小 Span 的申请流程，具体如下：

(1) Thread 用户逻辑层提交内存申请，如果本次申请内存超过 256KB 但不超过 1MB，则属于中对象申请。TCMalloc 将直接向 PageHeap 发起申请 Span 请求。

(2) PageHeap 接收到申请后需要判断本次申请是否属于小 Span(128 个 Page 以内)，如果是，则申请小 Span，即中对象申请流程，如果不是，则进入大对象申请流程，3.5.8 节介绍。

(3) PageHeap 根据申请的 Span 在小 Span 的链表中向上取整，得到最适合的第 K 个 Page 刻度的 Span 链表。

(4) 得到第 K 个 Page 链表刻度后，将 K 作为起始点，向下遍历找到第 1 个非空链表，直至 128 个 Page 刻度位置，如果找到，则停止，将停止处的非空 Span 链表作为提供此次返回的内存 Span，将链表中的第 1 个 Span 取出。如果找不到非空链表，则将本次申请当作大 Span 申请，进入大对象申请流程。

(5) 假设本次获取的 Span 由 N 个 Page 组成。PageHeap 将 N 个 Page 的 Span 拆分成两个 Span，其中一个为 K 个 Page 组成的 Span，作为本次内存申请的返回，返给 Thread，另一个为 $N-K$ 个 Page 组成的 Span，重新插入 $N-K$ 个 Page 对应的 Span 链表中。

综上是 TCMalloc 对于中对象分配的详细流程。

3.5.8　TCMalloc 的大对象分配

对于超过 128 个 Page(1MB)的内存分配采用大对象分配流程。大对象分配与中对象分配情况类似，Thread 绕过 ThreadCache 和 CentralCache，直接向 PageHeap 获取。详细的分配流程如图 3.30 所示。

进入大对象分配流程除了申请的 Span 大于 128 个 Page 之外，对于中对象分配如果找不到非空链表也会进入大对象分配流程，大对象分配的具体流程如下：

(1) Thread 用户逻辑层提交内存申请，如果本次申请内存超过 1MB，则属于大对象申请。TCMalloc 将直接向 PageHeap 发起申请 Span。

(2) PageHeap 接收到申请后需要判断本次申请是否属于小 Span(128 个 Page 以内)，如果是，则进入小 Span 中对象申请流程(3.5.7 节已介绍)，如果不是，则进入大对象申请流程。

(3) PageHeap 根据 Span 的大小按照 Page 单元进行除法运算，向上取整，得到最接近 Span 的且大于 Span 的 Page 倍数 K，此时的 K 应该大于 128。如果是从中对象流程分过来的(中对象申请流程可能没有非空链表提供 Span)，则 K 值应该小于 128。

(4) 搜索 Large Span Set 集合，找到不小于 K 个 Page 的最小 Span(N 个 Page)。如果没有找到合适的 Span，则说明 PageHeap 已经无法满足需求，遇到此种情况时向操作系统虚拟内存的堆空间申请一堆内存，将申请到的内存安置在 PageHeap 的内存结构中，重新执行步骤(3)。

(5) 将从 Large Span Set 集合得到的 N 个 Page 组成的 Span 拆分成两个 Span，K 个 Page 的 Span 直接返给 Thread 用户逻辑，$N-K$ 个 Span 退还给 PageHeap。其中如果 $N-K$ 大于 128，则退还到 Large Span Set 集合中，如果 $N-K$ 小于 128，则退还到 Page 链表中。

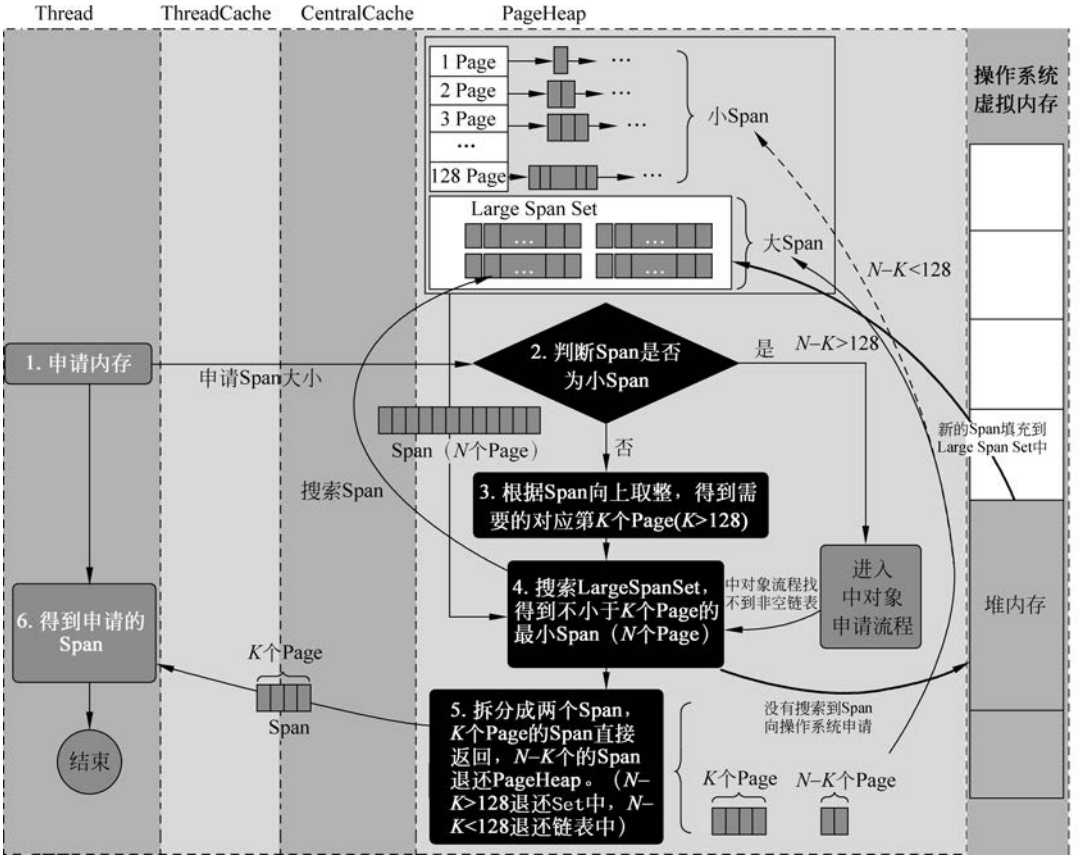

图 3.30 TCMalloc 大对象分配流程

综上是 TCMalloc 对于大对象分配的详细流程。

3.6 Go 语言堆内存管理

本节将介绍 Go 语言的内存管理模型，学习本节之前强烈建议读者将上述章节内容均理解透彻，更有助于理解 Go 语言的内存管理机制。

3.6.1 Go 语言内存模型层级结构

Go 语言内存管理模型的逻辑层次全景图，如图 3.31 所示。

Go 语言内存管理模型与 TCMalloc 的设计极其相似。基本轮廓和概念也几乎相同，只是一些规则和流程存在差异，接下来分析一下 Go 语言内存管理模型的基本层级模块的相关概念。

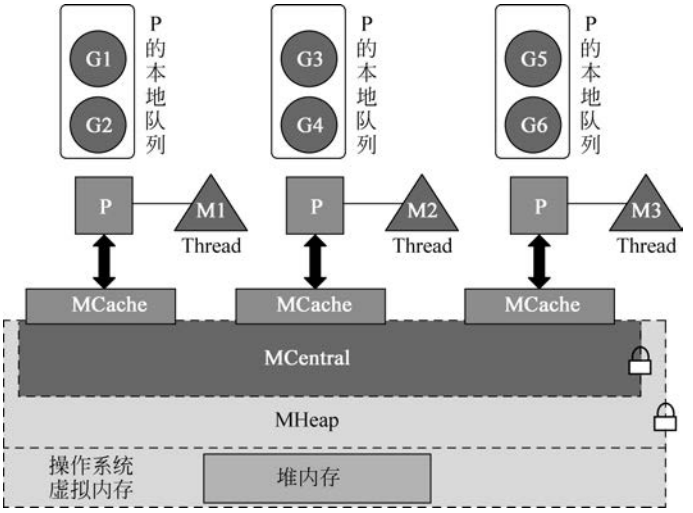

图 3.31　Go 语言内存管理模块关系

3.6.2　Go 语言内存管理单元相关概念

Go 语言内存管理中依然保留了 TCMalloc 中的 Page、Span、Size Class 等概念。

1. Page

与 TCMalloc 的 Page 一致。Go 语言内存管理模型延续了 TCMalloc 的概念，一个 Page 的大小依然是 8KB。Page 表示 Go 语言内存管理与虚拟内存交互时内存的最小单元。操作系统虚拟内存对于 Go 语言来讲，依然是划分成等份的 N 个 Page 组成的一块大内存公共池，如图 3.21 所示。

2. mSpan

与 TCMalloc 中的 Span 一致。mSpan 概念依然延续 TCMalloc 中的 Span 概念，在 Go 语言中将 Span 的名称改为 mSpan，依然表示一组连续的 Page。

3. Size Class 相关

Go 语言内存管理针对 Size Class 对衡量内存的概念又更加详细了很多，这里介绍一些基础的有关内存大小的名词及算法。

(1) Object Size，是指协程应用逻辑一次向 Go 语言内存申请的对象 Object 大小。Object 是 Go 语言内存管理模块针对内存管理更加细化的内存管理单元。一个 Span 在初始化时会被分成多个 Object。例如 Object Size 是 8B(8 字节)大小的 Object，所属的 Span 大小是 8KB(8192 字节)，那么这个 Span 就会被平均分割成 1024(8192/8＝1024)个 Object。逻辑层向 Go 语言内存模型取内存，实则是分配一个 Object 出去。为了更好地让读者理解，这里假设了几个数据来标识 Object Size 和 Span 的关系，如图 3.32 所示。

图 3.32 中的 Num of Object 表示当前 Span 中一共存在多少个 Object。

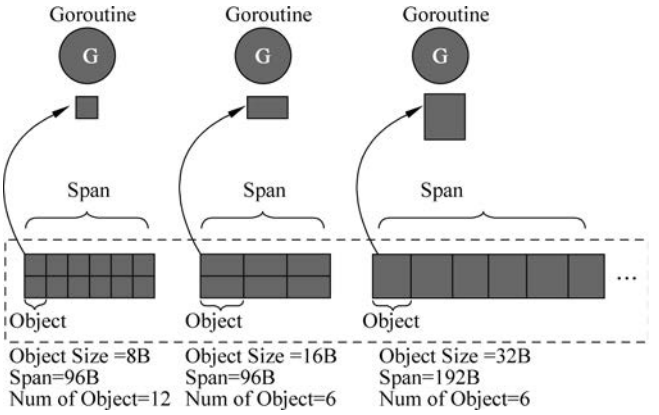

图 3.32 Object Size 与 Span 的关系

注意 Page 是 Go 语言内存管理与操作系统交互衡量内存容量的基本单元，Go 语言内存管理内部用来给对象存储内存的基本单元是 Object。

(2) Size Class，Go 语言内存管理中的 Size Class 与 TCMalloc 所表示的设计含义是一致的，都表示一块内存的所属规格或者刻度。Go 语言内存管理中的 Size Class 是针对 Object Size 来划分内存的，即划分 Object 大小的级别。例如 Object Size 在 1～8B 的 Object 属于 Size Class 1 级别，Object Size 在 8～16B 的属于 Size Class 2 级别。

(3) Span Class，这个是 Go 语言内存管理额外定义的规格属性，针对 Span 进行划分，是 Span 大小的级别。一个 Size Class 会对应两个 Span Class，其中一个 Span 为存放需要 GC 扫描的对象(包含指针的对象)，另一个 Span 为存放不需要 GC 扫描的对象(不包含指针的对象)，具体 Span Class 与 Size Class 的逻辑结构关系如图 3.33 所示。

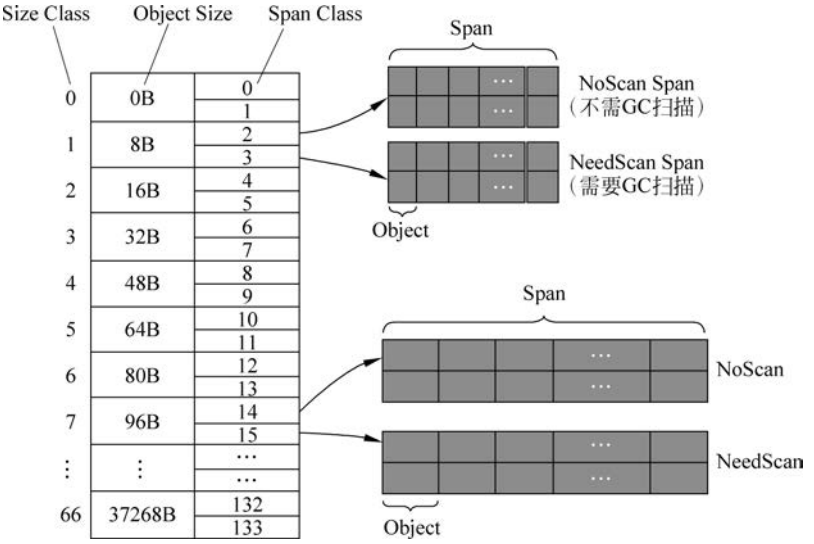

图 3.33 Span Class 与 Size Class 的逻辑结构关系

图 3.33 中 Size Class 和 Span Class 对应关系的计算方式可以参考 Go 语言源代码，如下：

```
//usr/local/go/src/runtime/mheap.go

type spanClass uint8

   …

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

   …
```

这里 makeSpanClass()函数为通过 Size Class 来得到对应的 Span Class，其中第 2 个形参 noscan 表示当前对象是否需要 GC 扫描，不难看出 Span Class 和 Size Class 的对应关系及计算公式如表 3.5 所示。

表 3.5　Span Class 和 Size Class 的对应关系及计算公式

对　　象	Size Class 与 Span Class 对应公式
需要 GC 扫描	Span Class＝Size Class× 2＋0
不需要 GC 扫描	Span Class＝Size Class× 2＋1

4. Size Class 明细

如果再具体一些，则通过 Go 语言的源码可以看出，Go 语言给内存池固定划分了 66[①] 个 Size Class，这里列举了详细的 Size Class 和 Object 大小、存放的 Object 数量，以及每个 Size Class 对应的 Span 内存大小关系，代码如下：

```
//usr/local/go/src/runtime/sizeclasses.go

package runtime

//标题 Title 解释
//[class]: Size Class
//[Bytes/obj]: Object Size,一次对外提供内存 Object 的大小
//[B/span]: 当前 Object 所对应 Span 的内存大小
//[objects]: 当前 Span 一共有多少个 Object
//[tail waste]: 为当前 Span 平均分成 N 份 Object,会有多少内存浪费
//[max waste]: 当前 Size Class 最大可能浪费的空间所占百分比
```

① 参考 Go 1.14 版本，其中还有扩展到 128 个 Size Class 的对应关系，本书不详细介绍，具体细节可参考 Go 源码/usr/local/go/src/runtime/sizeclasses.go 文件。

```
//class    Bytes/obj   B/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 %
//16         240         8192        34        32        6.62 %
//17         256         8192        32         0        5.86 %
//18         288         8192        28       128       12.16 %
//19         320         8192        25       192       11.80 %
//20         352         8192        23        96        9.88 %
//21         384         8192        21       128        9.51 %
//22         416         8192        19       288       10.71 %
//23         448         8192        18       128        8.37 %
//24         480         8192        17        32        6.82 %
//25         512         8192        16         0        6.05 %
//26         576         8192        14       128       12.33 %
//27         640         8192        12       512       15.48 %
//28         704         8192        11       448       13.93 %
//29         768         8192        10       512       13.94 %
//30         896         8192         9       128       15.52 %
//31        1024         8192         8         0       12.40 %
//32        1152         8192         7       128       12.41 %
//33        1280         8192         6       512       15.55 %
//34        1408        16384        11       896       14.00 %
//35        1536         8192         5       512       14.00 %
//36        1792        16384         9       256       15.57 %
//37        2048         8192         4         0       12.45 %
//38        2304        16384         7       256       12.46 %
//39        2688         8192         3       128       15.59 %
```

```
    //40        3072        24576        8          0        12.47%
    //41        3200        16384        5        384         6.22%
    //42        3456        24576        7        384         8.83%
    //43        4096         8192        2          0        15.60%
    //44        4864        24576        5        256        16.65%
    //45        5376        16384        3        256        10.92%
    //46        6144        24576        4          0        12.48%
    //47        6528        32768        5        128         6.23%
    //48        6784        40960        6        256         4.36%
    //49        6912        49152        7        768         3.37%
    //50        8192         8192        1          0        15.61%
    //51        9472        57344        6        512        14.28%
    //52        9728        49152        5        512         3.64%
    //53       10240        40960        4          0         4.99%
    //54       10880        32768        3        128         6.24%
    //55       12288        24576        2          0        11.45%
    //56       13568        40960        3        256         9.99%
    //57       14336        57344        4          0         5.35%
    //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%
```

下面分别解释一下每一列的含义：

(1) Class 列为 Size Class 规格级别。

(2) Bytes/obj 列为 Object Size，即一次对外提供内存 Object 的大小(单位为 B)，可能有一定的浪费，例如业务逻辑层需要 2B 的数据，实则会定位到 Size Class 为 1，返回一个 Object(8B)的内存空间。

(3) B/span 列为当前 Object 所对应 Span 的内存大小(单位为 B)。

(4) objects 列为当前 Span 一共有多少个 Object，该字段是通过 B/span 和 Bytes/obj 相除计算而来。

(5) tail waste 列为当前 Span 平均分成 N 份 Object 时会有多少内存浪费，这个值是通过 B/span 对 Bytes/obj 求余得出，即 span%obj。

(6) max waste 列为当前 Size Class 最大可能浪费的空间所占百分比。这里最大的情

况就是一个 Object 保存的实际数据刚好是上一级 Size Class 的 Object 大小加上 1B。当前 Size Class 的 Object 所保存的真实数据对象都是这种情况，这些全部空间的浪费再加上最后的 tail waste 就是 max waste 最大浪费的内存百分比，具体如图 3.34 所示。

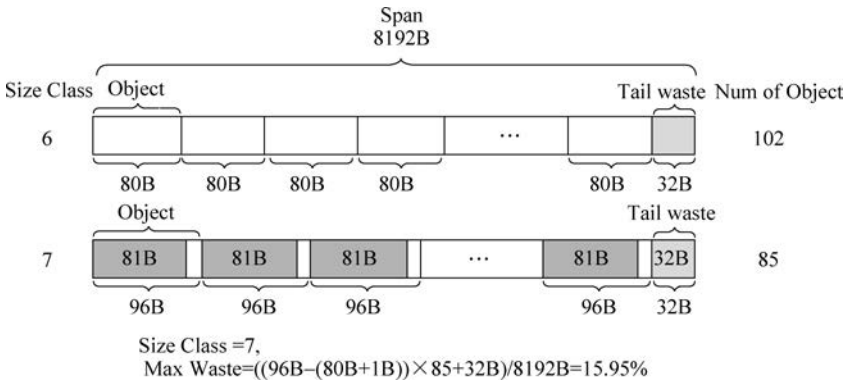

图 3.34 Max Waste 最大浪费空间计算公式

图 3.34 中以 Size Class 为 7 的 Span 为例，通过源代码 runtime/sizeclasses.go 的详细 Size Class 数据可以得知具体 Span 的细节如下：

```
//class  Bytes/obj  B/span  objects  tail waste  max waste

  …
//6         80       8192      102       32        19.07%
//7         96       8192       85       32        15.95%
  …
```

从图 3.34 可以看出，Size Class 为 7 的 Span 如果每个 Object 均超过 Size Class 为 7 中的 Object 的 1 字节，就会导致 Size Class 为 7 的 Span 出现最大空间浪费情况。综上可以得出计算最大浪费空间比例的算法如下：

```
(本级 Object Size - (上级 Object Size + 1) * 本级 Object 数量) / 本级 Span Size
```

3.6.3 MCache

从概念来讲 MCache 与 TCMalloc 的 ThreadCache 十分相似，访问 MCache 依然不需要加锁而是直接访问，并且 MCache 中依然保存着各种大小的 Span。

虽然 MCache 与 ThreadCache 概念相似，但是二者存在一定的区别，MCache 与 Go 语言协程调度模型 GPM 中的 P 绑定，而不是和线程绑定。因为 Go 语言调度的 GPM 模型，真正可运行的线程 M 的数量与 P 的数量一致，即 GOMAXPROCS 个，所以 MCache 与 P 进

行绑定更能节省内存空间，可以保证每个 G 使用 MCache 时不需要加锁就可以获取内存，而 TCMalloc 中的 ThreadCache 随着 Thread 的增多，ThreadCache 的数量相对成正比增多，二者绑定关系的区别如图 3.35 所示。

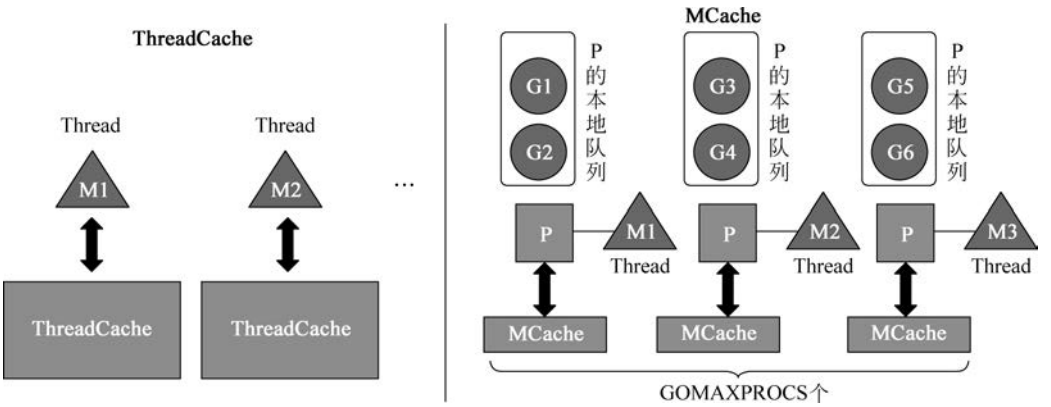

图 3.35　ThreadCache 与 MCache 的绑定关系区别

如果将图 3.35 中的 MCache 展开，来看 MCache 的内部构造，则具体的结构形式如图 3.36 所示。

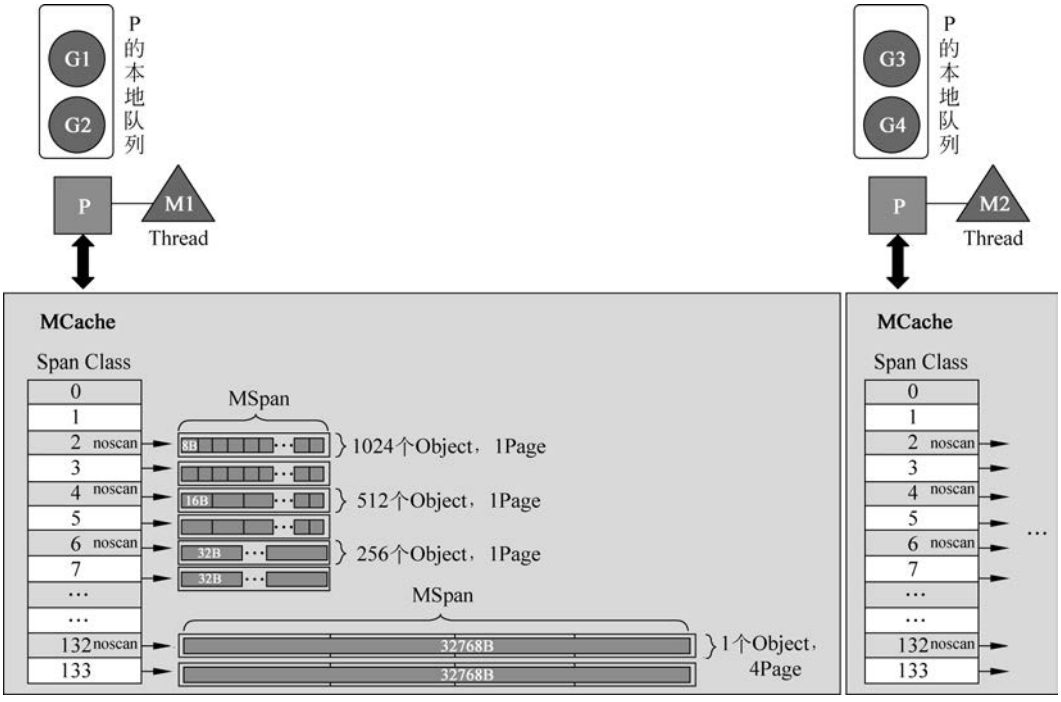

图 3.36　MCache 内部构造

协程逻辑层从 MCache 上获取内存时不需要加锁，因为一个 P 只有一个 M 在其上运行，不可能出现竞争，由于没有锁限制，所以 MCache 会加速内存分配。

MCache 中每个 Span Class 都会对应一个 MSpan，不同 Span Class 的 MSpan 的总体长度不同，参考 runtime/sizeclasses.go 文件中的标准规定划分。例如对于 Span Class 为 4 的 MSpan 来讲，存放内存大小为 1Page，即 8KB。每个对外提供的 Object 大小为 16B，共存放 512 个 Object。其他 Span Class 的存放方式与此类似。当其中某个 Span Class 的 MSpan 已经没有可提供的 Object 时，MCache 则会向 MCentral 申请一个对应的 MSpan。

在图 3.36 中应该会发现，对于 Span Class 为 0 和 1，也就是对应 Size Class 为 0 的规格刻度内存，MCache 实际上没有分配任何内存。因为 Go 语言内存管理对内存为 0 的数据申请做了特殊处理，如果申请的数据大小为 0，则将直接返回一个固定内存地址，不会进入 Go 语言内存管理的正常逻辑，相关 Go 语言源代码如下：

```
//usr/local/go/src/runtime/malloc.go

//Al Allocate an object of size Bytes.
//Sm Small objects are allocated from the per-P cache's free lists.
//La Large objects (> 32 kB) are allocated straight from the heap.
func mallocgc(size uintptr, typ *_type, needzero bool) unsafe.Pointer {
    …

    if size == 0 {
        return unsafe.Pointer(&zerobase)
}

…
}
```

从上述代码可以看到，如果申请的 size 为 0，则直接 return 一个固定地址 zerobase。下面来测试一下有关 0 空间申请的情况，在 Go 语言中，[0]int 和 struct{}所需要大小均是 0，这也是为什么很多开发者在通过 Channel 做同步时，会发送一个 struct{}数据，因为不会申请任何内存，能够适当节省一部分内存空间，测试代码如下：

```
//第一篇/chapter3/MyGolang/zeroBase.go
package main

import (
    "fmt"
)

func main() {
    var (
```

```
        //0 内存对象
        a struct{}
        b [0]int

        //100 个 0 内存 struct{}
        c [100]struct{}

        //100 个 0 内存 struct{},make 申请形式
        d = make([]struct{}, 100)
    )

    fmt.Printf("%p\n", &a)
    fmt.Printf("%p\n", &b)
    fmt.Printf("%p\n", &c[50])              //取任意元素
    fmt.Printf("%p\n", &(d[50]))            //取任意元素
}
```

运行结果如下：

```
$ go run zeroBase.go
0x11aac78
0x11aac78
0x11aac78
0x11aac78
```

从结果可以看出，全部的 0 内存对象分配，返回的都是一个固定的地址。

3.6.4　MCentral

MCentral 与 TCMalloc 中的 Central 概念依然相似。向 MCentral 申请 Span 时同样需要加锁。当 MCache 中某个 Size Class 对应的 Span 被一次次 Object 上层取走后，如果出现当前 Size Class 的 Span 空缺情况，MCache 则会向 MCentral 申请对应的 Span。Goroutine、MCache、MCentral、MHeap 互相交换的内存单位是不同的，具体如图 3.37 所示。

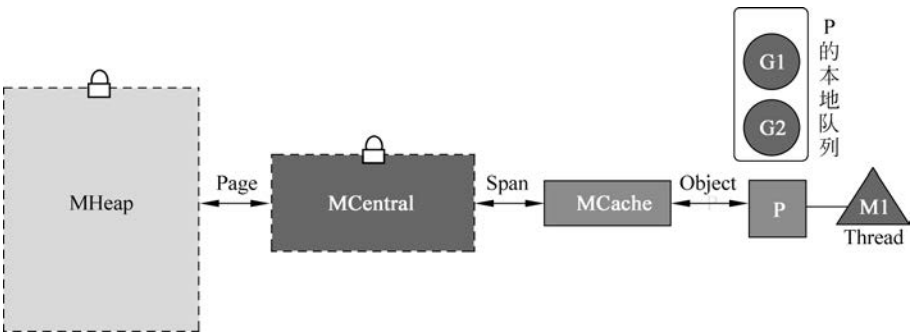

图 3.37　Go 语言内存管理各层级内存交换单位

其中协程逻辑层与 MCache 的内存交换单位是 Object，MCache 与 MCentral 的内存交换单位是 Span，而 MCentral 与 MHeap 的内存交换单位是 Page。

MCentral 与 TCMalloc 中的 Central 不同的是 MCentral 针对每个 Span Class 级别有两个 Span 链表，而 TCMalloc 中的 Central 只有一个。MCentral 的内部构造如图 3.38 所示。

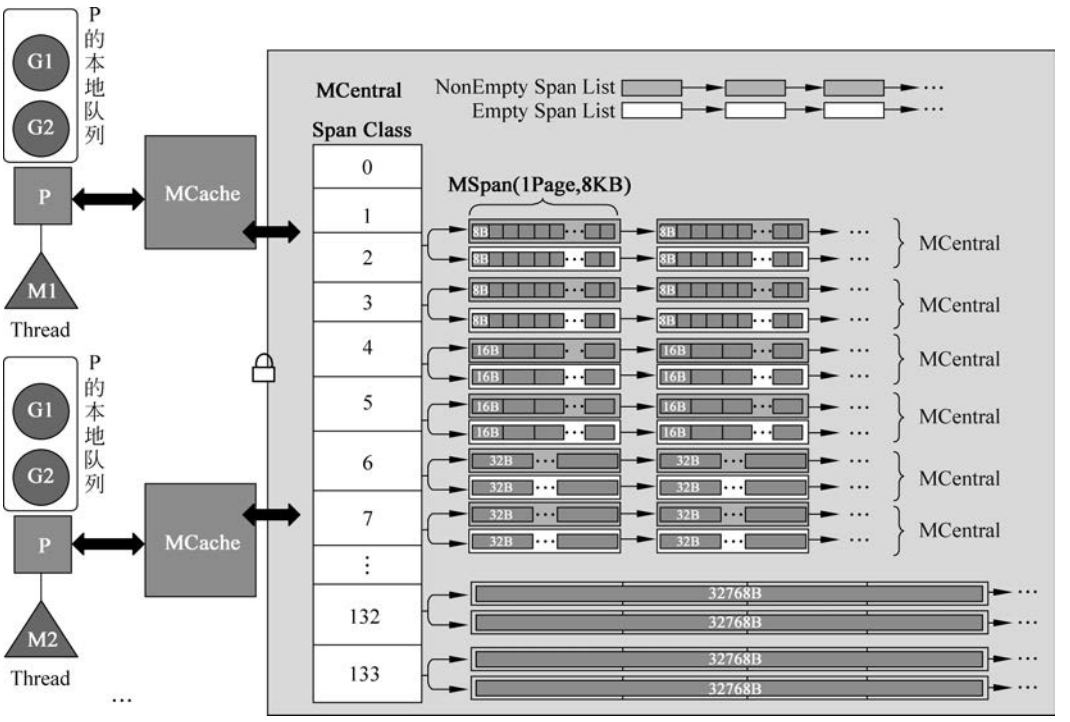

图 3.38 MCentral 的内部构造

MCentral 与 MCache 不同的是，每个级别保存的不是一个 Span，而是一个 Span List 链表。与 TCMalloc 中的 Central 不同的是，MCentral 每个级别都保存了两个 Span List。

注意 图 3.38 中 MCentral 表示一层抽象的概念，实际上每个 Span Class 对应的内存数据结构是一个 MCentral，即在 MCentral 这层数据管理中，实际上有 Span Class 个 MCentral 小内存管理单元。

1. NonEmpty Span List

NonEmpty Span List 表示还有可用空间的 Span 链表。链表中的所有 Span 都至少有 1 个空闲的 Object 空间。如果 MCentral 上游的 MCache 退还 Span，则会将退还的 Span 加入 NonEmpty Span List 链表中。

2. Empty Span List

Empty Span List 表示没有可用空间的 Span 链表。该链表上的 Span 都不确定是否还有空闲的 Object 空间。如果 MCentral 将一个 Span 提供给上游 MCache，则被提供的 Span

就会加入 Empty List 链表中。

注意　在 Go 1.16 版本之后，MCentral 中的 NonEmpty Span List 和 Empty Span List 均由链表管理改成集合管理，分别对应 Partial Span Set 和 Full Span Set。虽然存储的数据结构有变化，但是基本的作用和职责没有区别。

下面是 MCentral 层级中其中一个 Size Class 级别的 MCentral 的定义，Go 源代码（V1.14 版本）如下：

```
//usr/local/go/src/runtime/mcentral.go , Go V1.14

//Central list of free objects of a given size.
//go:notinheap
type mcentral struct {
lock       mutex                          //申请 MCentral 内存分配时需要加的锁

spanclass spanClass                       //当前属于哪个 Size Class 级别

//list of spans with a free object, ie a nonempty free list
//还有可用空间的 Span 链表
nonempty mSpanList

//list of spans with no free objects (or cached in an MCache)
//没有可用空间的 Span 链表，或者当前链表里的 Span 已经交给 MCache
    empty mSpanList

    //nmalloc is the cumulative count of objects allocated from
    //this mcentral, assuming all spans in mcaches are
//fully-allocated. Written atomically, read under STW.
//nmalloc 是从该 mcentral 分配的对象的累积计数
//假设 mcaches 中的所有跨度都已完全分配
//以原子方式书写，在 STW 下阅读
    nmalloc uint64
}
```

在 Go V1.16 及之后版本（截止本书编写时）的相关 MCentral 结构代码如下：

```
//usr/local/go/src/runtime/mcentral.go , Go V1.16+

…

type mcentral struct {
    //mcentral 对应的 spanClass
    spanclass spanClass
```

```
        partial [2]spanSet                    //维护全部空闲的 Span 集合
        full [2]spanSet                       //维护存在非空闲的 Span 集合
    }

    ...
```

新版本的改进是将 List 变成了两个 Set 集合，Partial 集合与 NonEmpty Span List 的责任类似，Full 集合与 Empty Span List 的责任类似。可以看到 Partial 和 Full 都是一个 [2]spanSet 类型，即每个 Partial 和 Full 都各有两个 spanSet 集合，这是为了给 GC 垃圾回收使用的，其中一个集合是已扫描的，另一个集合是未扫描的。

3.6.5 MHeap

Go 内存管理的 MHeap 依然继承了 TCMalloc 的 PageHeap 设计。MHeap 的上游是 MCentral，当 MCentral 中的 Span 不够时会向 MHeap 申请。MHeap 的下游是操作系统，当 MHeap 的内存不够时会向操作系统的虚拟内存空间申请。访问 MHeap 获取内存依然需要加锁。

MHeap 是内存块的管理对象，通过 Page 对内存单元进行管理。用来详细管理每一系列 Page 的结构称为一个 HeapArena，它们的逻辑层级关系如图 3.39 所示。

一个 HeapArena 占用内存 64MB[①]，其中里面的内存是一个一个的 mspan，当然最小单元依然是 Page，图中没有表示出 mspan，因为多个连续的 Page 就是一个 mspan。所有的 HeapArena 组成的集合是一个 Arenas，即 MHeap 针对堆内存的管理。MHeap 是 Go 语言进程全局唯一的，所以访问依然加锁。图 3.39 中又出现了 MCentral，因为 MCentral 本属于 MHeap 中的一部分。只不过会优先从 MCentral 获取内存，如果没有 MCentral，则会从 Arenas 中的某个 HeapArena 获取 Page。

如果再详细剖析 MHeap 里面相关的数据结构和指针依赖关系，则可以参考图 3.40，这里不做过多解释，如果想详细理解 MHeap，则建议研读源代码/usr/local/go/src/runtime/mheap.go 文件。

MHeap 中的 HeapArena 占用了绝大部分的空间，其中每个 HeapArean 包含一个 bitmap，其作用是标记当前这个 HeapArena 的内存使用情况。其主要服务于 GC 垃圾回收模块，bitmap 共有两种标记，一种是标记对应地址中是否存在对象，另一种是标记此对象是否被 GC 模块标记过，所以当前 HeapArena 中的所有 Page 均会被 bitmap 所标记。

ArenaHint 为寻址 HeapArena 的结构，其有三个成员：

(1) addr 为指向的对应 HeapArena 的首地址。

(2) down 为当前的 HeapArena 是否可以扩容。

---

① 在 Linux 64 位操作系统上。

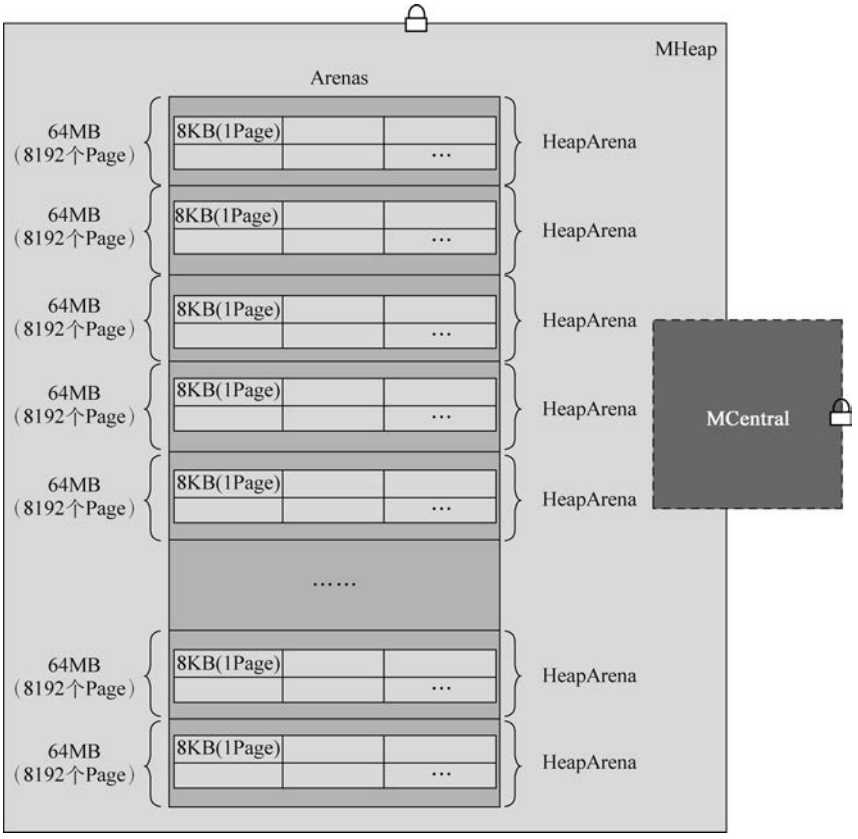

图 3.39　MHeap 内部逻辑层级构造

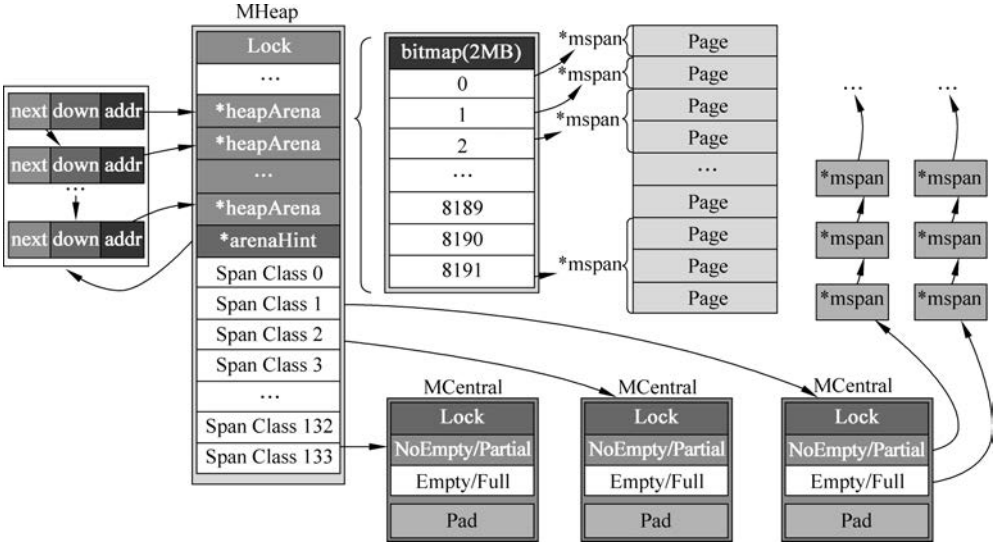

图 3.40　MHeap 数据结构引用依赖

(3) next 指向下一个 HeapArena 所对应的 ArenaHint 的首地址。

从图 3.40 可以看出,MCentral 实际上隶属于 MHeap 的一部分,从数据结构来看,每个 Span Class 对应一个 MCentral,而之前在分析 Go 语言内存管理的逻辑分层中,将这些 MCentral 集合统一归类为 MCentral 层。

3.6.6 Tiny 对象分配流程

在之前章节的表 3.4 中可以得到 TCMalloc 将对象分为小对象、中对象和大对象,而 Go 语言内存管理则将对象的分类进行了更细的划分,具体的划分区别对比如表 3.6 所示。

表 3.6 Go 语言内存与 TCMalloc 对内存的分类对比

TCMalloc	Go
小对象 (0,256KB]	Tiny 对象 [1,16B)
中对象 (256KB,1MB]	小对象 [16B,32KB]
大对象 (1MB,+∞)	大对象 (32KB,+∞)

针对 Tiny 微小对象的分配,实际上 Go 语言做了比较特殊的处理,之前在介绍 MCache 的时候并没有提及有关 Tiny 的存储和分配问题,MCache 中不仅保存着各个 Span Class 级别的内存块空间,还有一个比较特殊的 Tiny 存储空间,如图 3.41 所示。

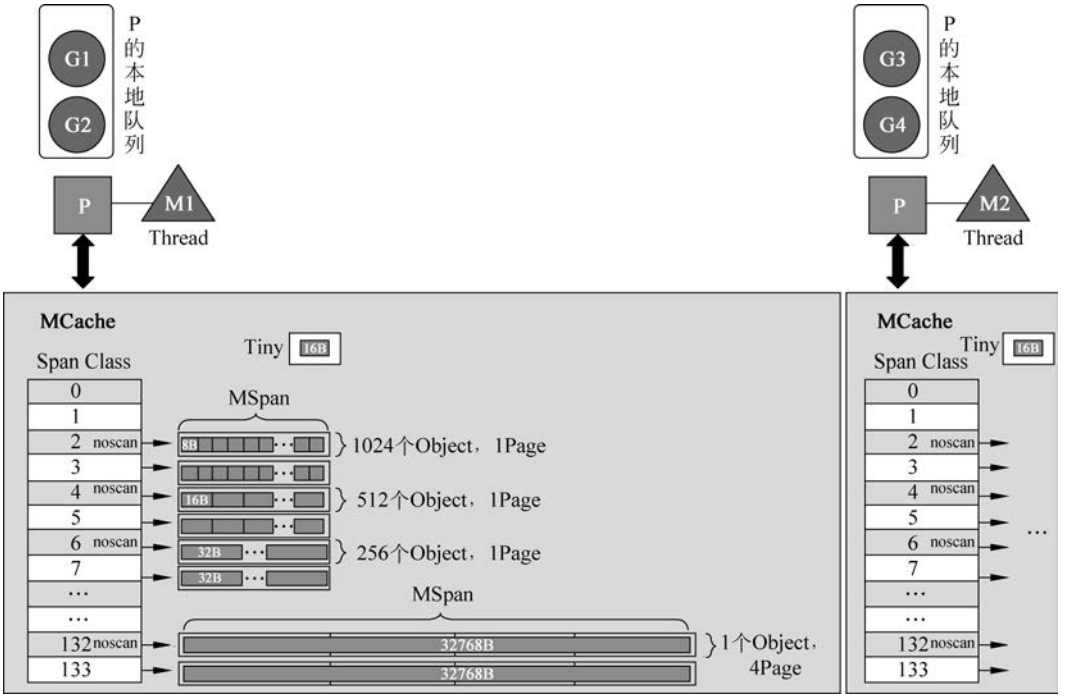

图 3.41 MCache 中的 Tiny 空间

Tiny 空间是从 Size Class＝2(对应 Span Class＝4 或 5)中获取一个 16B 的 Object,作为 Tiny 对象的分配空间。Go 语言内存管理为什么需要一个 Tiny 这样的 16B 空间? 原因是如果协程逻辑层申请的内存空间小于或等于 8B,则根据正常的 Size Class 匹配会匹配到 Size Class＝1(对应 Span Class＝2 或 3),所以像 int32、Byte、bool 及小字符串等经常使用的 Tiny 微小对象,也都会使用从 Size Class＝1 申请的这 8B 的空间,但是类似 bool 或者 1 字节的 Byte,也都会各自独享这 8B 的空间,进而导致一定的内存空间浪费,如图 3.42 所示。

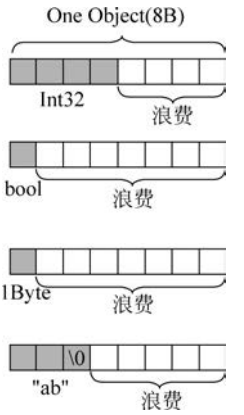

图 3.42　如果微小对象不存在 Tiny 空间

可以看出,当大量地使用微小对象时可能会对 Size Class＝1 的 Span 造成浪费,所以 Go 语言内存管理决定尽量不使用 Size Class＝1 的 Span,而是将申请的 Object 小于 16B 的申请统一归类为 Tiny 对象申请。具体的申请流程如图 3.43 所示。

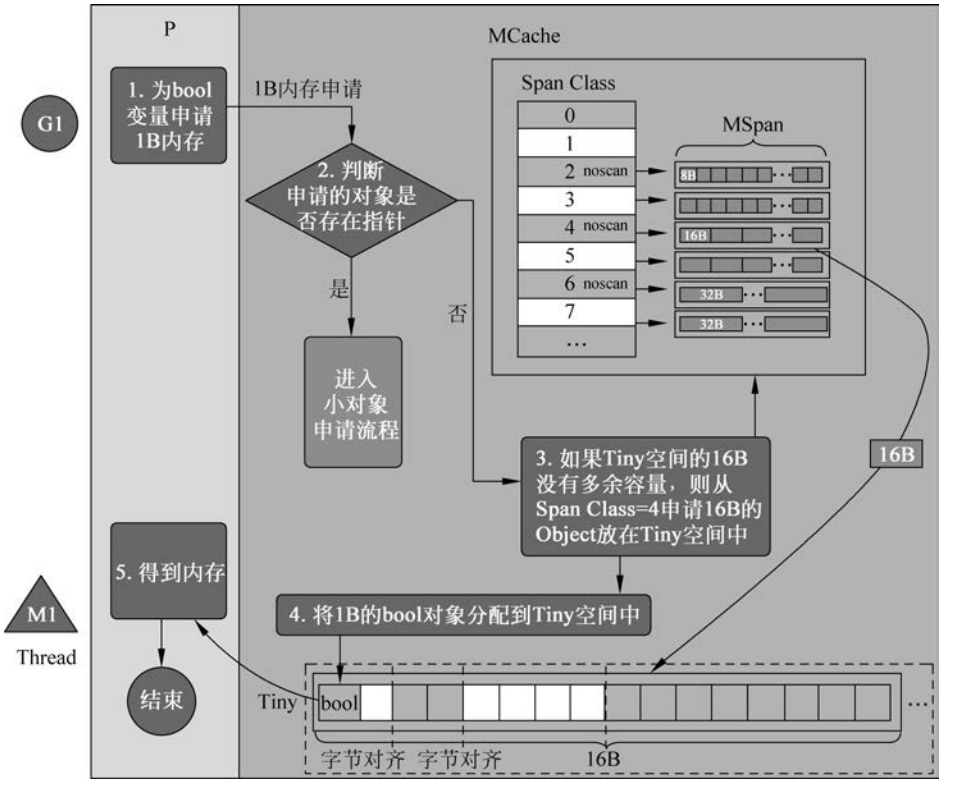

图 3.43　MCache 中 Tiny 微小对象分配流程

MCache 中对于 Tiny 微小对象的申请流程如下:

(1) P 向 MCache 申请微小对象,如一个 Bool 变量。如果申请的 Object 在 Tiny 对象

的大小范围，则进入 Tiny 对象申请流程，否则进入小对象或大对象申请流程。

(2) 判断申请的 Tiny 对象是否包含指针，如果包含指针，则进入小对象申请流程(不会放在 Tiny 缓冲区，因为需要 GC 进入扫描等流程)。

(3) 如果 Tiny 空间的 16B 没有多余的存储容量，则从 Size Class=2(Span Class=4 或 5)的 Span 中获取一个 16B 的 Object 放置于 Tiny 缓冲区。

(4) 将 1B 的 Bool 类型放置在 16B 的 Tiny 空间中，以字节对齐的方式放置。

Tiny 对象的申请也达不到内存利用率 100%，以图 3.43 为例，当前 Tiny 缓冲 16B 的内存利用率为$\frac{1+2+8}{16}\times 100\%=68.75\%$，而如果不用 Tiny 微小对象的方式来存储，则内存的布局将如图 3.44 所示。

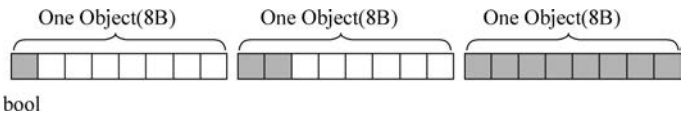

图 3.44 不用 Tiny 缓冲存储情况

可以算出利用率为$\frac{1+2+8}{8\times 3}\times 100\%=45.83\%$。Go 语言内存管理通过 Tiny 对象的处理，可以平均节省 20%左右的内存。

3.6.7 小对象分配流程

3.6.6 节已经介绍了分配在 1B～16B 的 Tiny 对象的分配流程，对于对象在 16B～32B 的内存分配，Go 语言会采用小对象的分配流程。

分配小对象的标准流程是按照 Span Class 规格匹配的。在之前介绍 MCache 的内部构造时已经介绍了，MCache 一共有 67 份 Size Class，其中对 Size Class 为 0 的情况做了特殊处理，即直接返回一个固定的地址。Span Class 为 Size Class 的两倍，也就是 0～133 共 134 个 Span Class。

当协程逻辑层 P 主动申请一个小对象的时候，Go 语言内存管理的内存申请流程如图 3.45 所示。

下面来分析一下具体的流程：

(1) 首先协程逻辑层 P 向 Go 语言内存管理申请一个对象所需的内存空间。

(2) MCache 在收到请求后，会根据对象所需的内存空间计算出具体的大小 Size。

(3) 判断 Size 是否小于 16B，如果小于 16B，则进入 Tiny 微对象申请流程，否则进入小对象申请流程。

(4) 根据 Size 匹配对应的 Size Class 内存规格，再根据 Size Class 和该对象是否包含指针，来定位是从 noscan Span Class 还是从 scan Span Class 获取空间，如果没有指针，则锁定 noscan。

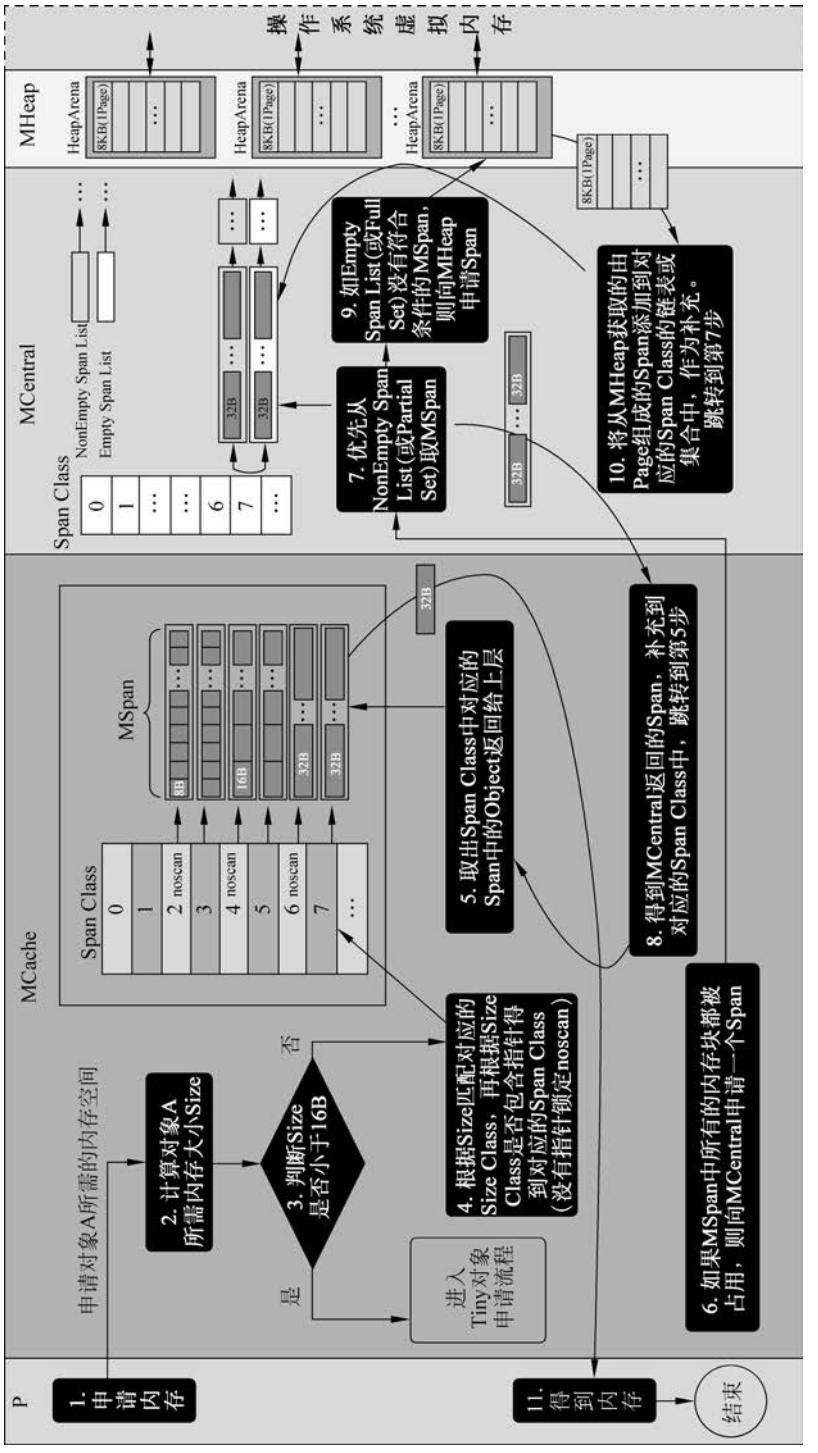

图 3.45　Go 语言小对象内存分配流程

(5) 在定位的 Span Class 中的 Span 取出一个 Object 返给协程逻辑层 P,P 得到内存空间,流程结束。

(6) 如果定位的 Span Class 中的 Span 所有的内存块 Object 都被占用,则 MCache 会向 MCentral 申请一个 Span。

(7) MCentral 收到内存申请后,优先从相对应的 Span Class 中的 NonEmpty Span List(或 Partial Set,Go V1.16+)里取出 Span(由多个 Object 组成),如果 NonEmpty Span List 没有,则从 Empty List(或 Full Set Go V1.16+)中取,返给 MCache。

(8) MCache 得到 MCentral 返回的 Span,补充到对应的 Span Class 中,之后再次执行第(5)步流程。

(9) 如果 Empty Span List(或 Full Set)中没有符合条件的 Span,则 MCentral 会向 MHeap 申请内存。

(10) MHeap 收到内存请求后从其中一个 HeapArena 从取出一部分 Pages 返给 MCentral,当 MHeap 没有足够的内存时,MHeap 会向操作系统申请内存,将申请的内存也保存到 HeapArena 中的 mspan 中。MCentral 将从 MHeap 获取的由 Pages 组成的 Span 添加到对应的 Span Class 链表或集合中,作为新的补充,之后再次执行第(7)步。

(11) 最后协程业务逻辑层得到该对象申请到的内存,流程结束。

3.6.8 大对象分配流程

小对象是在 MCache 中分配的,而大对象则直接从 MHeap 中分配。对于不满足 MCache 分配范围的对象,均按照大对象分配流程处理。

大对象分配流程是协程逻辑层直接向 MHeap 申请对象所需要的适当 Pages,从而绕过从 MCache 到 MCentral 的烦琐申请内存流程,大对象的内存分配流程相对比较简单,具体的流程如图 3.46 所示。

下面分析具体的大对象内存分配流程:

(1) 协程逻辑层申请大对象所需的内存空间,如果超过 32KB,则直接绕过 MCache 和 MCentral 向 MHeap 申请。

(2) MHeap 根据对象所需的空间计算得到需要多少个 Page。

(3) MHeap 向 Arenas 中的 HeapArena 申请相对应的 Pages。

(4) 如果 Arenas 中没有 HeapArena 可提供合适的 Pages 内存,则向操作系统的虚拟内存申请,并且填充至 Arenas 中。

(5) MHeap 返回大对象的内存空间。

(6) 协程逻辑层 P 得到内存,流程结束。

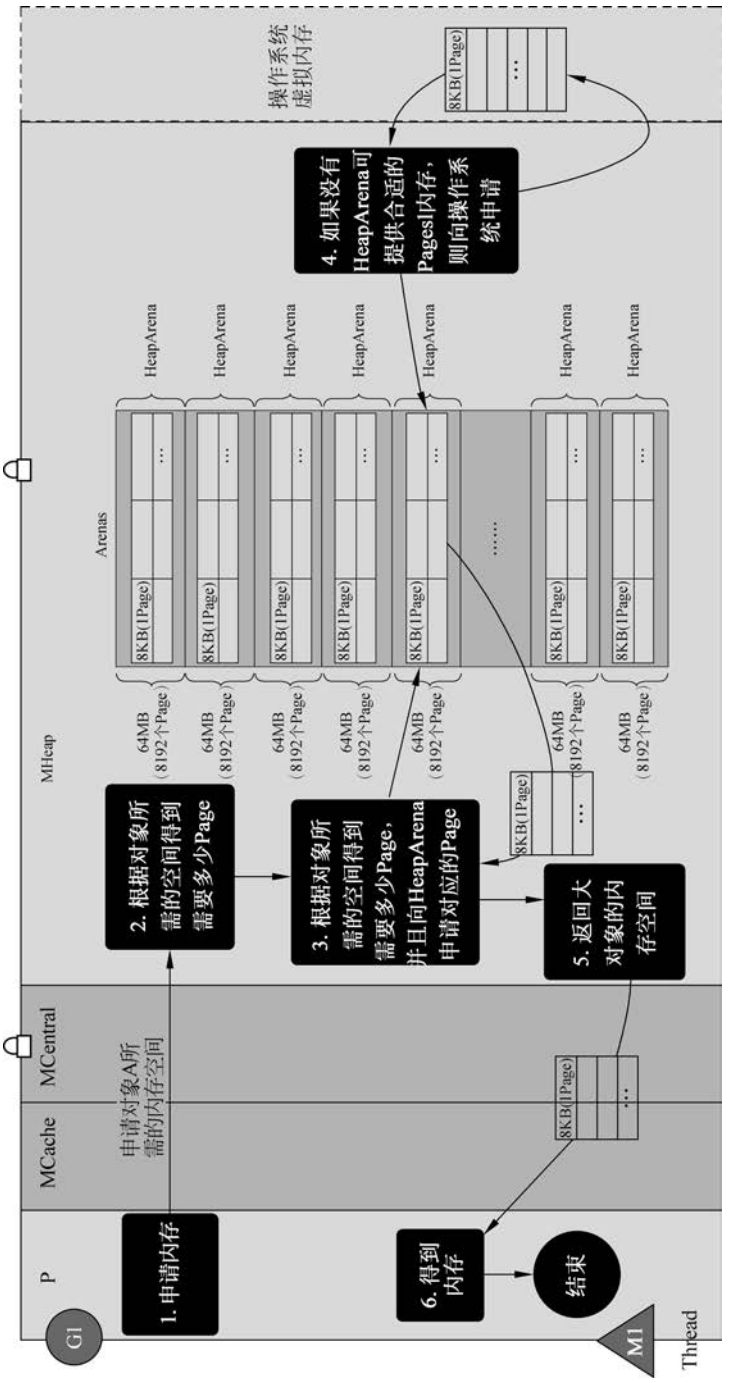

图 3.46　Go 语言大对象内存分配流程

3.7 小结

本章从操作系统的虚拟内存申请到Go语言内存模型进行理论的推进和逐层剖析。通过本章讲解的内存管理，可以了解无论是操作系统虚拟内存管理，还是C++的TCMalloc、Go语言内存模型，均有一个共同特点，即分层的缓存机制。针对不同的内存场景采用不同的独特解决方式，提高局部性逻辑和细微粒度内存的复用率，这也是程序设计的至高理念。