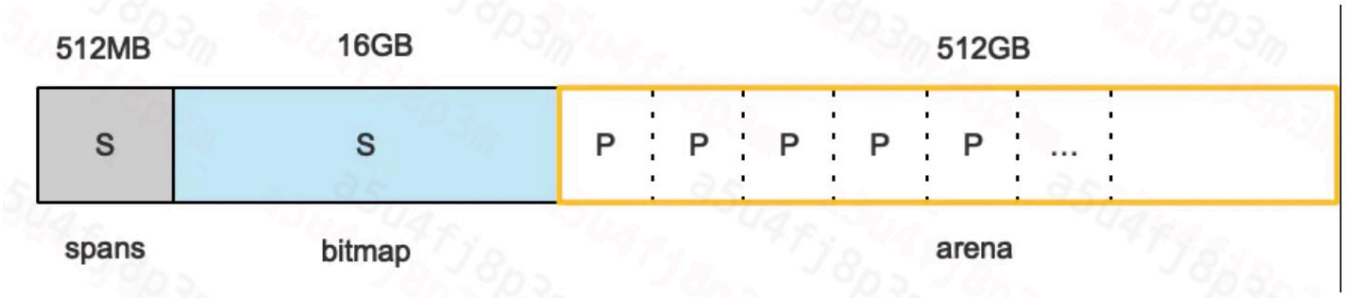
go基础学习——内存分配

go的内存分配算法源于Google为C语言开发的TCMalloc算法，全称Thread Cache Malloc，主要思想是采用了内存的多级管理。

内存管理

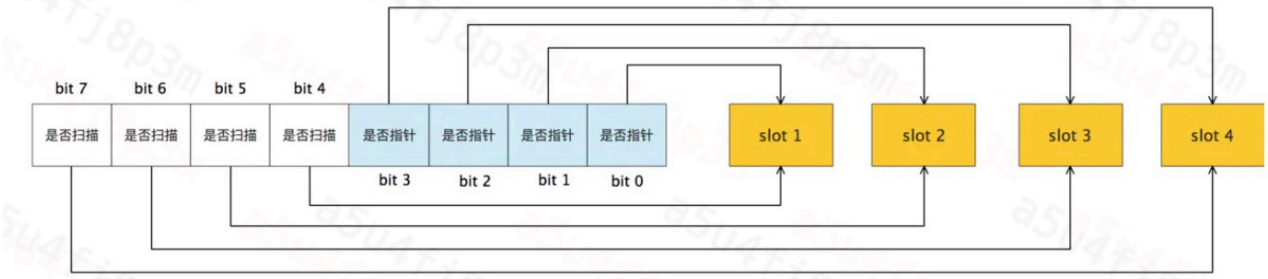
go的程序开始时，会先向操作系统申请一大块虚拟内存，然后切成小块自己进行管理。

申请到的内存主要分成三个部分：**arena、bitmap、spans**，分别是512G，16G和512MB。



arena就是我们常说的堆区，在这个区域当中，会将内存分成8KB大小的页，然后按页进行管理。

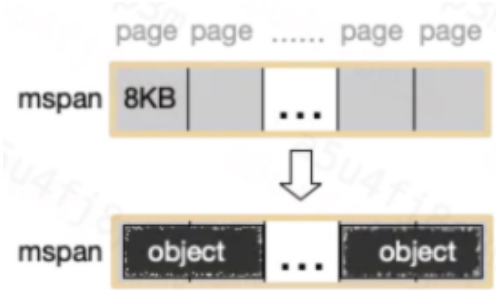
然后，bitmap当中保存了arena当中的一些信息，如下图所示



在**bitmap**中，以8bit为一个单位进行信息存储，高四位保存了slot的GC信息，低四位保存了slot是否包含数据，然后bitmap当中的高地址指向arena当中的低地址。

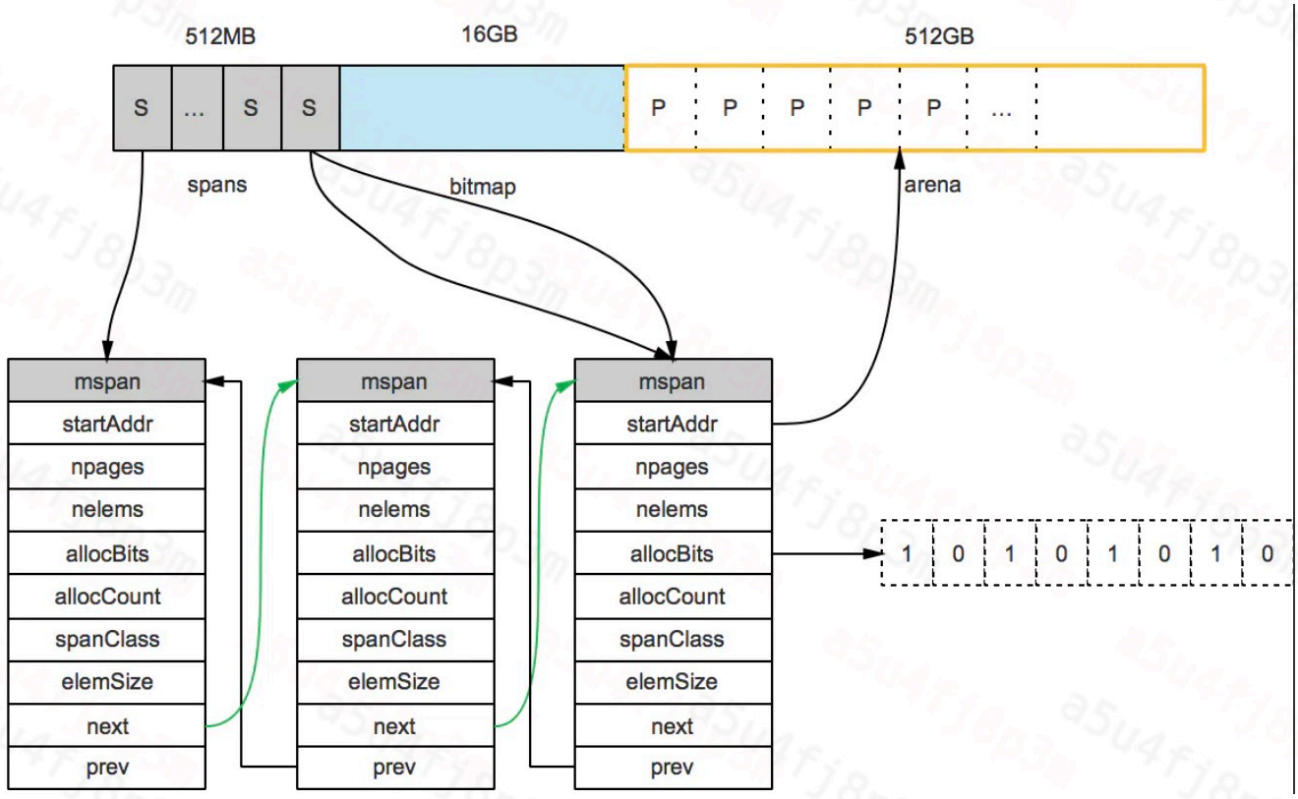
spans：存的是mspan指针，用于表示arena区域中的下标对应的页属于哪个mspan

mspan：是内存管理的基本单元，由一连串8KB页组成的大内存，其实质是包含起始地址、mspan规格和页数量等内容的双端链表。然后，mspan根据size class划分若干object，在内存分配时就将对象分配给相近大小的object，共有67种size class，可以划分出8*2^n大小的object，



object最大是32KB，再大就变成大对象，直接由堆进行分配。

spans与arena关系如下图所示：



内存分配

内存分配由三种组件完成，mcache，mcentral和mheap

mcache：每个工作线程都会绑定一个mcache，本地缓存可用的mspan资源，这样就可以直接给Goroutine分配，因为不存在多个Goroutine竞争的情况，所以不会消耗锁资源。mcache包含所有种类大小的mspan，并且double，因为可以将mspan分成两组，一组包含指针，一组不包含。

mcentral：为所有mcache提供切分好的mspan资源。每个central保存一种特定大小的全局mspan列表，包括已分配出去的和未分配出去的。每个mcentral对应一种mspan，而mspan的种类导致它分割的object大小不同。当工作线程的mcache中没有合适（也就是特定大小的）的mspan时就会从mcentral获取。

mheap：代表Go程序持有的所有堆空间，Go程序使用一个mheap的全局对象_mheap来管理堆内存。当mcentral没有空闲的mspan时，会向mheap申请。而mheap没有资源时，会向操作系统申请新内存。mheap主要用于大对象的内存分配，以及管理未切割的mspan，用于给mcentral切割成小对象。同时我们也看到，mheap中含有所有规格的mcentral，所以，当一个mcache从mcentral申请mspan时，只需要在独立的mcentral中使用锁，并不会影响申请其他规格的mspan。

内存分配流程

go将对象分为三类，分别是微对象(<16B),小对象(<=32KB),大对象(>32KB)

微对象直接使用mcache中的tiny分配器进行分配

小对象使用mcache当中近似大小的mspan进行分配，如果mcache中没有，向mcentral中申请，如果mcentral中没有，向mheap中申请，如果mheap中没有，向操作系统申请。

源码分析

1.mspan源码

```
type mspan struct {
//mspannextprev
next *mspan
prev *mspan
// debug
list *mSpanList
// mspanheapheapstartAddrmspan
startAddr uintptr
// mspanheap
npages uintptr
//
manualFreeList gclinkptr
//
freeindex uintptr
//msapnobjectnpageselementSize
nelems uintptr
//
allocCache uint64
//mspanelem
allocBits *gcBits
//maspango
gcmarkBits *gcBits
//GC
sweepgen uint32
// for divide by elemsize - divMagic.mul
divMul uint16
// if non-0, elemsize is a power of 2, & this will get object allocation
base
baseMask uint16
//object
allocCount uint16
//mspan
spanclass spanClass
//msapn
state mSpanState
//
needzero uint8
// for divide by elemsize - divMagic.shift
divShift uint8
// for divide by elemsize - divMagic.shift2
divShift2 uint8
//msapn
scavenged bool
//object
elemsize uintptr
//msapnobject
limit uintptr
// guards specials list
speciallock mutex
// linked list of special records sorted by offset.
specials *special
}
```

综上所述，mspan是go当中内存管理的基本单元，是由一片连续的 8KB的页组成的大块内存

1.它包含有startAddr以及npages可以确定其管理范围；
2.包含spanclass和elemsize可以确定当中每一个object的大小；
3.包含freeindex可以确定空闲对象的下表；
4.包含allocBits和gcmarkBits可以用来确定mspan中的分配情况以及垃圾回收过程中的标记情况。
5.包含的scavenged和sweegen可以表示垃圾回收的阶段如：是否需要回收、正在回收、正准备使用等。

2.mcache源码

Go 像 TCMalloc 一样为每一个逻辑处理器P 提供一个本地span缓存称作 mcache。如果 协程 需要内存可以直接从 mcache 中获取，由于在同一时间只有一个 协程运行在 逻辑处理器P上，所以中间不需要任何锁的参与。mcache 包含所有大小规格的 mspan 作为缓存，但是每种规格大小只包含一个。

```
type mcache struct {
// The following members are accessed on every malloc,
// so they are grouped here for better caching.
// trigger heap sample after allocating this many bytes
next_sample uintptr
//
local_scan uintptr
//tiny
tiny uintptr
//tiny
tinyoffset uintptr
//tiny
local_tinyallocs uintptr // number of tiny allocs not counted in other
stats
// The rest is not accessed on every malloc.
//mcache67*2msapn67msapnmcachegc
alloc [numSpanClasses]*mspan
//
stackcache [_NumStackOrders]stackfreelist
//large object
local_largefree uintptr
//large object
local_nlargefree uintptr
//mspan
local_nsmallfree [_NumSizeClasses]uintptr
//gc
flushGen uint32
}
```

综上所述

1.mcache中包含了tiny和tinyoffset对象用来分配微对象(<16B)
2.mcache中包含了alloc对象，该对象是67种msapn的集合，可以用来分配小对象(<32KB)。
3.mcache无法分配大对象，由堆直接分配(>32KB)

3.mcentral源码

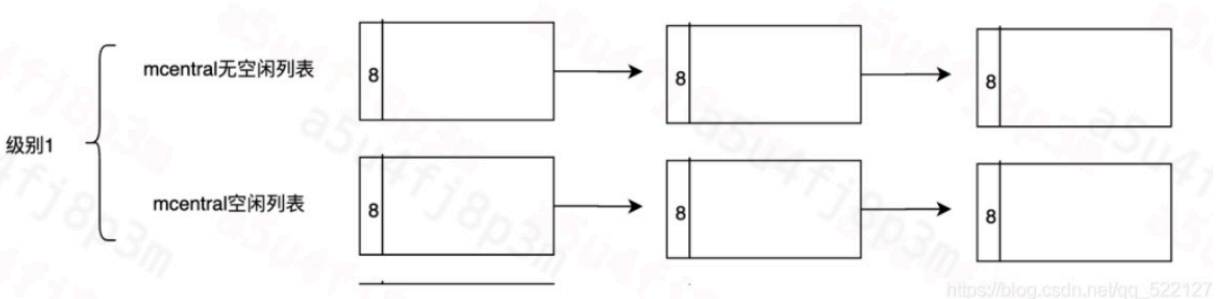
mcentral 对象收集所有给定规格大小的 span。每一个 mcentral 都包含两个 mspan 的链表：

做这种区主要是为了更快的分配span到mcache中。

除了级别0，每一个级别都会有一个mcentral，管理span列表。

nonempty mspanList - 有空闲对象的 span链表，这里的nonempty指的是可用内存空间不为空

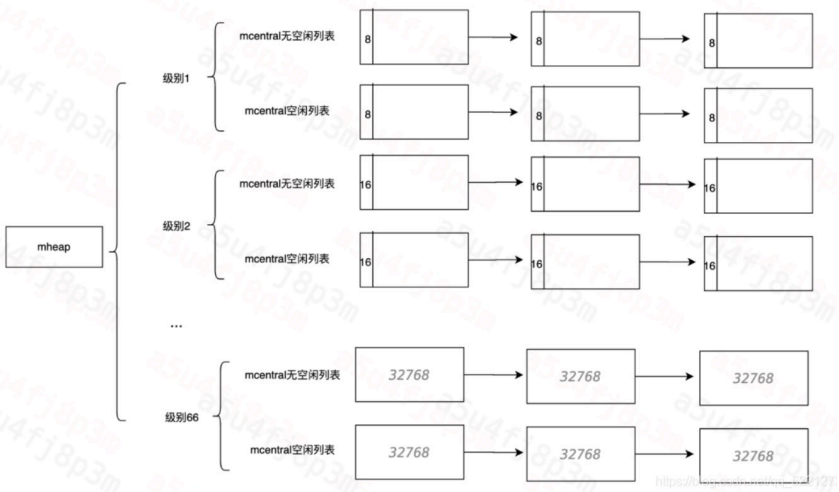
empty mspanList - 没有空闲对象或 span 已经被 mcache 缓存的 span 链表



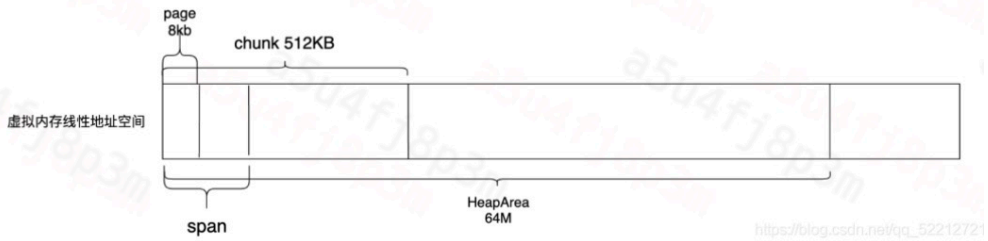

```
type mcentral struct {
//mcentralmcentral
lock mutex
//2*sizeclassscannonscan
spanclass spanClass
//type mSpanList struct {
//first *mspan // first span in list, or nil if none
//last *mspan // last span in list, or nil if none
//}
//mspanListmsapn
//nonempty
nonempty mSpanList
//mspan
empty mSpanList
//
nmalloc uint64
}
```

4.mheap源码

而所有级别的这些mcentral，其实是一个数组，由 mheap进行管理。



mheap的作用不只是管理central,另外大对象也会直接通过mheap进行分配。mheap实现了对于虚拟内存线性地址空间的精准管理，建立了span与具体线性地址空间的联系，保存了分配的位图信息，是管理内存的最核心单元。后面我们还会看到，堆区的内存还被分成了HeapArea大小进行管理。对heap进行的操作必须得全局加锁，而不管是mcache、mcentral都只能看做是某种形式的缓存。




```
type mheap struct {
// lock must only be acquired on the system stack, otherwise a g
// could self-deadlock if its stack grows with the lock held.
lock mutex
//mtreap go
free mTreap
//mspansweegen
sweepgen uint32
// all spans are swept
sweepdone uint32
// number of active sweepone calls
sweepers uint32
//mspan
allspans []*mspan
//
sweepSpans [2]gcSweepBuf
// align uint64 fields on 32-bit for atomics
_ uint32
//pages
pagesInUse uint64
//gcpage
pagesSwept uint64
//pagesSwept to use as the origin of the sweep ratio; updated atomically
pagesSweptBasis uint64
// value of heap_live to use as the origin of sweep ratio; written with
lock, read without
sweepHeapLiveBasis uint64
//proportional sweep ratio; written with lock, read without
sweepPagesPerByte float64
//
scavengeTimeBasis int64
scavengeRetainedBasis uint64
scavengeBytesPerNS float64
scavengeRetainedGoal uint64
scavengeGen uint64 // incremented on each pacing update
reclaimIndex uint64
reclaimCredit uintptr
// Malloc stats.
//
largealloc uint64
//
nlargealloc uint64
//
```

```
largefree uint64
//
nlargefree uint64
//
nsmallfree [_NumSizeClasses]uint64
// arenas is the heap arena map. It points to the metadata for
// the heap for every arena frame of the entire usable virtual
// address space.
arenas [1 << arenaL1Bits]*[1 << arenaL2Bits]*heapArena
//
heapArenaAlloc linearAlloc
// arenaHints is a list of addresses at which to attempt to
// add more heap arenas. This is initially populated with a
// set of general hint addresses, and grown with the bounds of
// actual heap arena ranges.
arenaHints *arenaHint
// arena is a pre-reserved space for allocating heap arenas
// (the actual arenas). This is only used on 32-bit.
arena linearAlloc
// allArenas is the arenaIndex of every mapped arena. This can
// be used to iterate through the address space.
//
// Access is protected by mheap_.lock. However, since this is
// append-only and old backing arrays are never freed, it is
// safe to acquire mheap_.lock, copy the slice header, and
// then release mheap_.lock.
allArenas []arenaIdx
// sweepArenas is a snapshot of allArenas taken at the
// beginning of the sweep cycle. This can be read safely by
// simply blocking GC (by disabling preemption).
sweepArenas []arenaIdx
// curArena is the arena that the heap is currently growing
// into. This should always be physPageSize-aligned.
curArena struct {
base, end uintptr
}
_ uint32 // ensure 64-bit alignment of central
// central free lists for small size classes.
// the padding makes sure that the mcentrals are
// spaced CacheLinePadSize bytes apart, so that each mcentral.lock
// gets its own cache line.
// central is indexed by spanClass.
central [numSpanClasses]struct {
mcentral mcentral
pad [cpu.CacheLinePadSize - unsafe.Sizeof(mcentral{})%cpu.CacheLinePadSize]
byte
}

//span
spanalloc fixalloc
//mcache
cachealloc fixalloc
//treapNode
```

```
treapalloc fixalloc
//specialfinalizer
specialfinalizeralloc fixalloc
//specialprofile
specialprofilealloc fixalloc
//special record
speciallock mutex
//arenaHints
arenaHintAlloc fixalloc
unused *specialfinalizer // never set, just here to force the
specialfinalizer type into DWARF
}
```

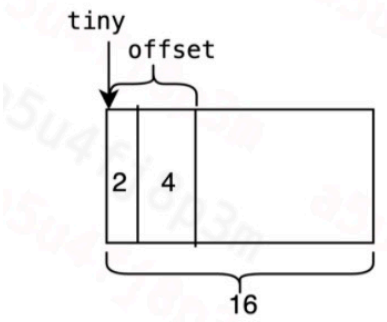
内存分配具体流程——以tiny对象举例

对于小于16字节的对象，Go语言将其划分为了tiny对象。划分tiny对象的主要目的是为了处理极小的字符串和独立的转义变量。对json的基准测试表明，使用tiny对象减少了12%的分配次数和20%的堆大小[1]

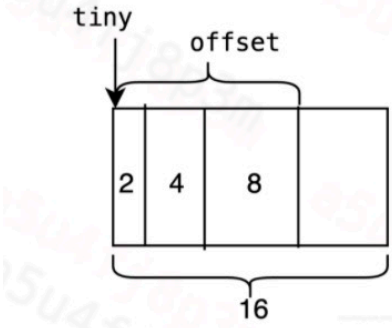
tiny对象会被放入class为2的span中，由上例中知道，class为2的span元素大小为16字节。首先对tiny对象按照2、4、8进行字节对齐。例如字节为1的元素会分配2个字节，字节为7的元素会分配8个字节。

```
if size&7 == 0 {
off = alignUp(off, 8)
} else if size&3 == 0 {
off = alignUp(off, 4)
} else if size&1 == 0 {
off = alignUp(off, 2)
}
```

首先查看之前分配的元素中是否有空余的空间。如下所示，如果当前对象要分配8个字节，并且前一个分配的元素可以容纳大小为8，则返回tiny+ooffset的地址，意味着当前的地址往后8个字节都是可以被分配的。

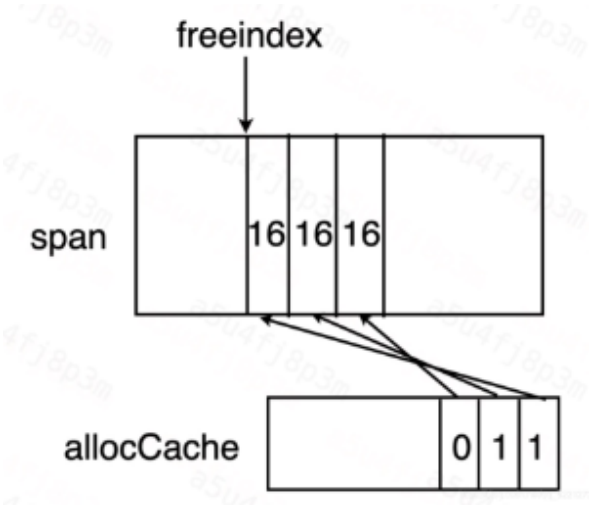


分配后offset进行移动



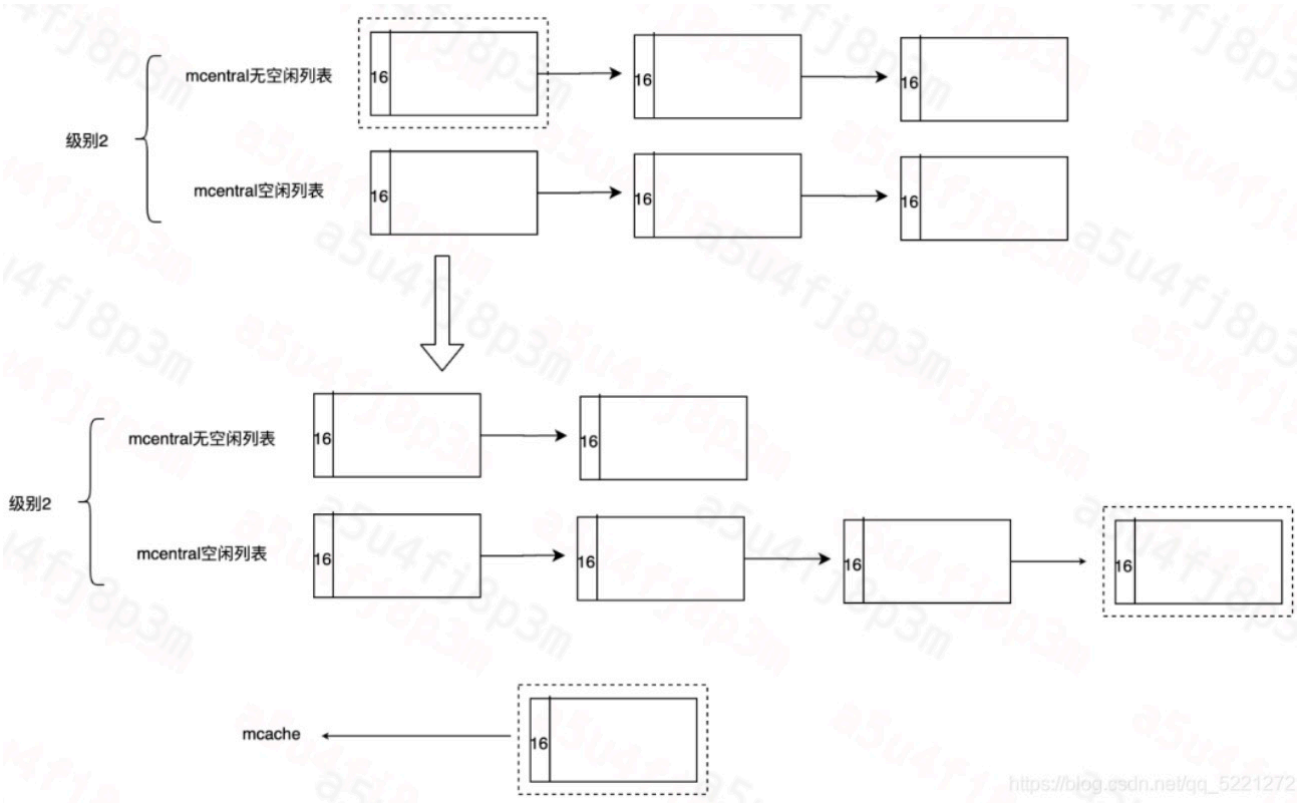
```
off := c.tinyoffset
if off+size <= maxTinySize && c.tiny != 0 {
x = unsafe.Pointer(c.tiny + off)
c.tinyoffset = off + size
return x
}
```

所以tiny对象分配的第一步是进行一个对齐操作，之后查看上一个分配对象是否有足够的内存容纳该tiny对象，如果内存不够用，则向mcache中申请。
向mcache中申请的步骤：
1.找到对应级别的mspan
2.查看当前的alloccache，alloccache是一个位图，指示mspan中freeindex一定范围内是否有空闲对象，如下图所示



因此，只要从acclocCache开始找到哪一位为0即可。假如找到了X位为0，那么X + freeindex 为当前span中可用的元素序号。当allocCache中全部都已经标记为1后，就需要移动freeindex ，并更新allocCache。一直到达span元素末尾为止。
如果当前的span中并没有可以使用的元素，这时就需要从mcentral中加锁查找。
之前介绍过，在mcentral中有两种类型的span链表，分别是有空闲元素的nonempty，以及没有空闲元素的empty链表。会分别遍历这两个列表,查找是否有可用的span。有些读者可能会有疑问，既然是没有空闲元素的empty列表，怎么还需要去遍历呢？这是由于有些span可能已经被垃圾回收器标记为空闲了，只是还没有来得及清理。这些Span在清扫后仍然是可以使用的，因此需要遍历。

向central中申请内存步骤：
1.如果在mcentral元素中查找到有空闲元素的span，则将其赋值到mcache中，并更新allocCache，同时还需要将span添加到mcentral的empty链表中去。

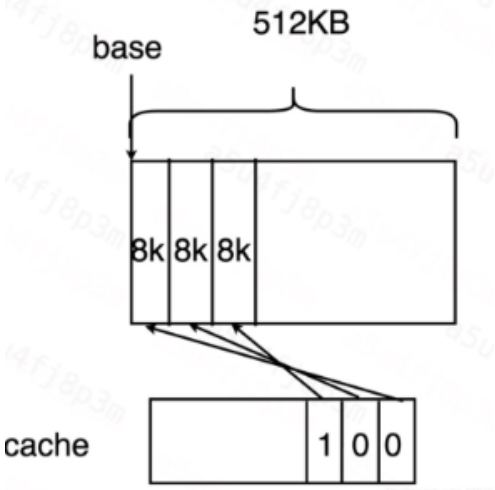


当在mcentral中找不到可以使用的span后，就需要从mheap中查找。

在Go1.12的时候,Go语言采用了Treap 进行内存的管理，Treap 是一种引入了随机数的二叉树搜索树，其实现简单，并且引入的随机数以及必要时的旋转保证了比较好的平衡特性。Michael Knyszek 提出[2]这种方式具有扩展性的问题，由于这棵树是mheap管理，当操作此二叉树的时候都需要维持一个lock。这在密集的对象分配以及逻辑处理器P过多的时候，会导致更长的等待时间。Michael Knyszek 提出用bitmap来管理内存页,并在每个P中维护一份page cache。这就是现在Go语言实现的方式。因此在go1.14之后，我们会看到在每个逻辑处理器P内部都有一个cache。

```
type pageCache struct {
base uintptr
cache uint64
scav uint64
}
```

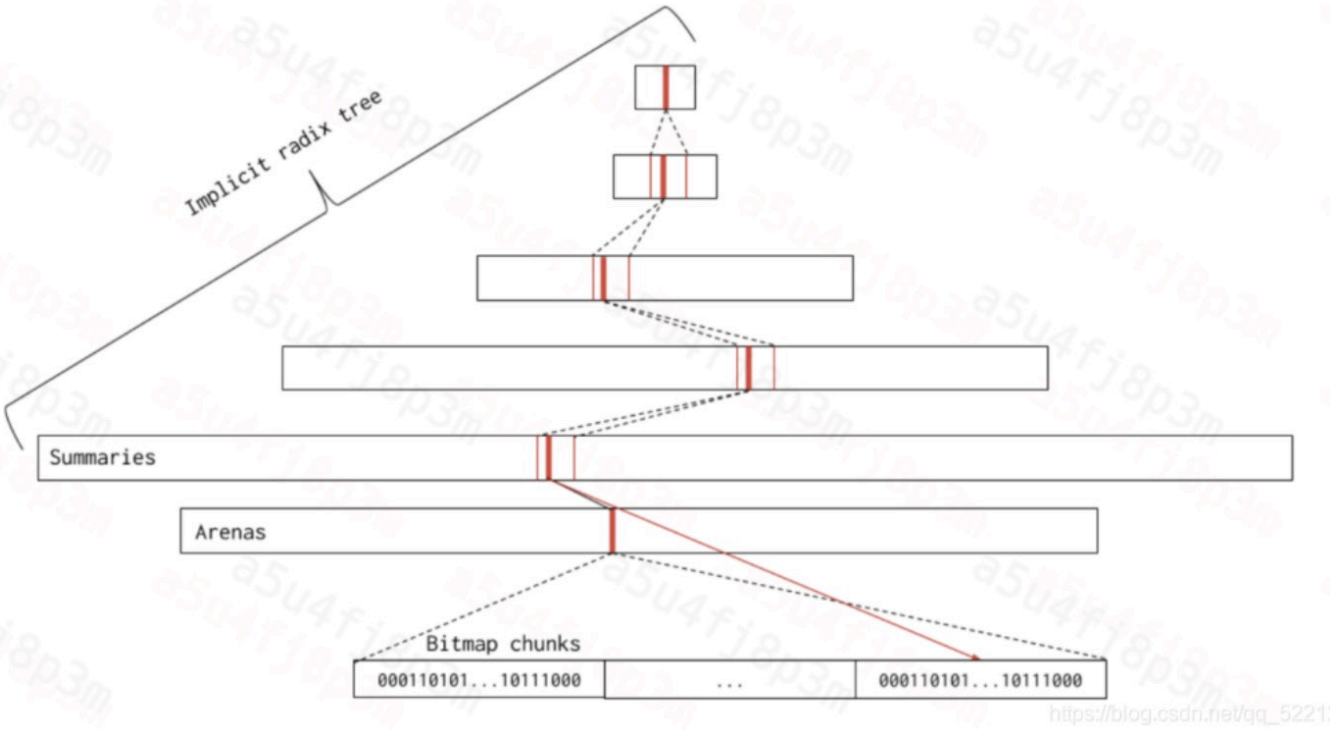
mheap会首先查找在每个逻辑处理器P中pageCache字段的cache。cache也是一个位图，其每一位都代表了一个page(8 KB) 因此，由于cache为uint64类型，其一共可以存储64*8=512KB的缓存。这512KB是连续的虚拟内存。在cache中，1代表未分配的内存，而0代表已分配的内存。base代表该虚拟内存的基地址。当需要分配的页数小于 512/4=128KB时，需要首先从cache中分配。



例如，假如要分配n pages，就需要查找cache中是否有连续n个1位。如果存在，则说明在缓存中查找到了合适的内存，用于初始化span。

当要分配的page过大或者在逻辑处理器P的cache中没有找到可用的页数时，就需要对mheap加锁，并在整个mheap管理的虚拟地址空间的位图中查找是否有可用的pages。而且其在本质上涉及到Go语言是如何对线性的地址空间进行位图管理的。

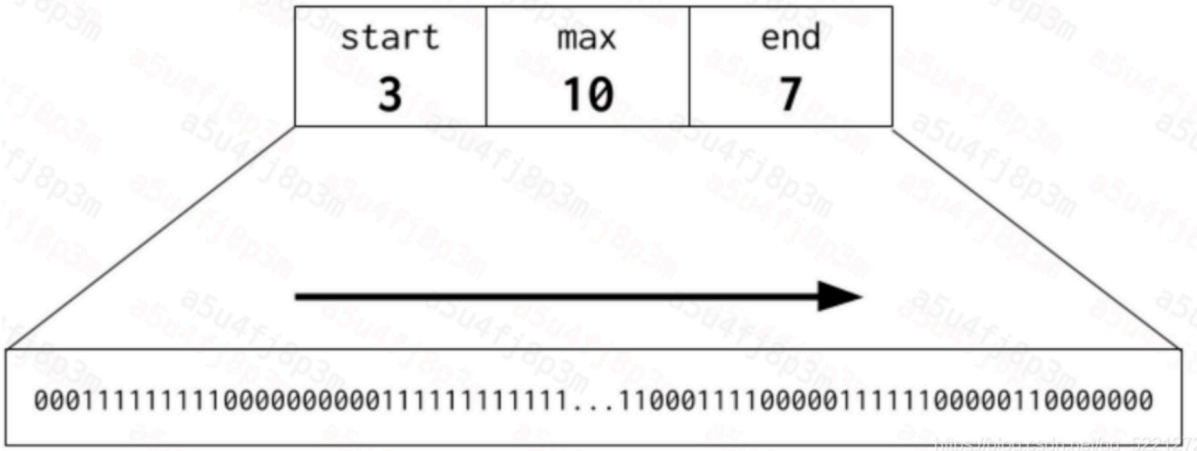
管理线性的地址空间的位图结构叫做基数树(radix tree)， 他和一般的基数树结构有点不太一样，这个名字很大一部分是由于父节点包含了子节点的若干信息。



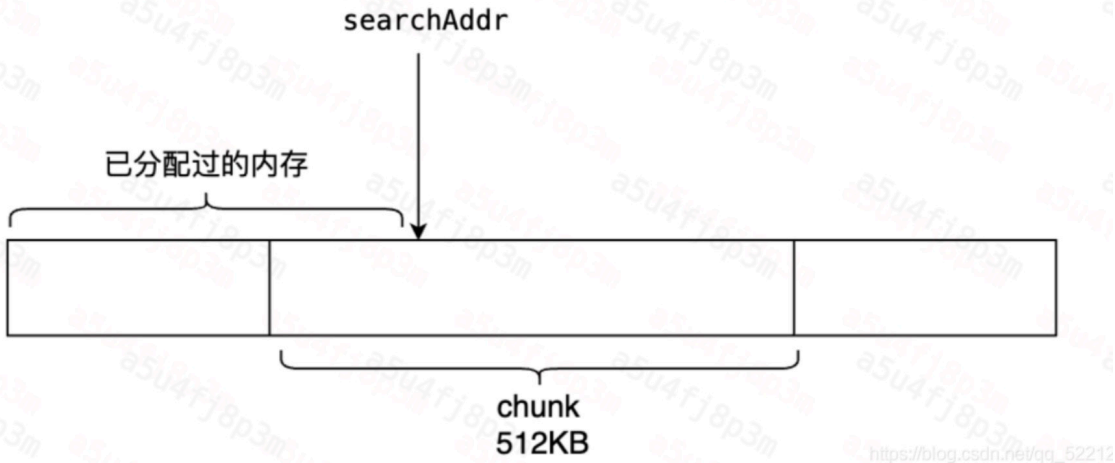
在该树中的每一个节点对应一个pallocSum结构。其中最底层的叶子节点对应的一个pallocSum结构包含了一个chunk的信息(512 * 8 KB),而除了叶子节点外的节点都包含了连续8个子节点的内存信息。例如,倒数第二层的节点包含了8个叶子节点(即8*chunk)的连续内存信息。因此，越上层的节点，其对应的内存就越多。

```
type pallocSum uint64

func (p pallocSum) start() uint {
return uint(uint64(p) & (maxPackedValue - 1))
}

// max extracts the max value from a packed sum.
func (p pallocSum) max() uint {
return uint((uint64(p) >> logMaxPackedValue) & (maxPackedValue - 1))
}

// end extracts the end value from a packed sum.
func (p pallocSum) end() uint {
return uint((uint64(p) >> (2 * logMaxPackedValue)) & (maxPackedValue - 1))
}
```

pallocSum虽然是一个简单的uint64类型，但是分成了开头、中间、末尾3个部分，开头语末尾部分占据了21bit,中间部分占据了22bit。它们分别包含了在这个区域中连续空闲内存页的信息。包括了在开头有多少连续内存页，最大有多少连续内存页，在末尾有多少连续内存页。对于最顶层的节点，由于其中间的max位为22bit，因此一颗完整的基数树最多代表2^21 pages=16G内存。



Go语言并不是一开始就直接从根节点往下查找的，而是首先做了一定的优化，类似于又一级别的缓存。在Go语言中，存储了一个特别的字段searchAddr，看名字就可以猜想到是用于搜索可用内存时使用的。searchAddr有一个重要的设定是在searchAddr地址之前一定是已经分配过的。因此在查找时，只需要往searchAddr地址的后方查找即可跳过查找的节点，减少查找的时间。



在第一次查找时，会首先从当前searchAddr的chunk块中查找是否有对应大小的连续空间。这种优化主要是针对比较小的内存分配(至少小于512KB)时使用的。Go语言对于内存有非常精细化的管理，chunk块的每一个page(8 KB)都有位图表明其是否已经被分配。

每一个chunk都有一个pallocData结构，其中pallocBits管理其分配的位图。pallocBits是一个uint64的大小为8的数组。由于每一位对应着一个page，因此pallocBits总共对应着64*8=512KB，恰好是一个chunk块的大小。位图的对应方式和之前是一样的。

```
type pallocData struct {
pallocBits
scavenged pageBits
}

type pallocBits [8]uint64
```

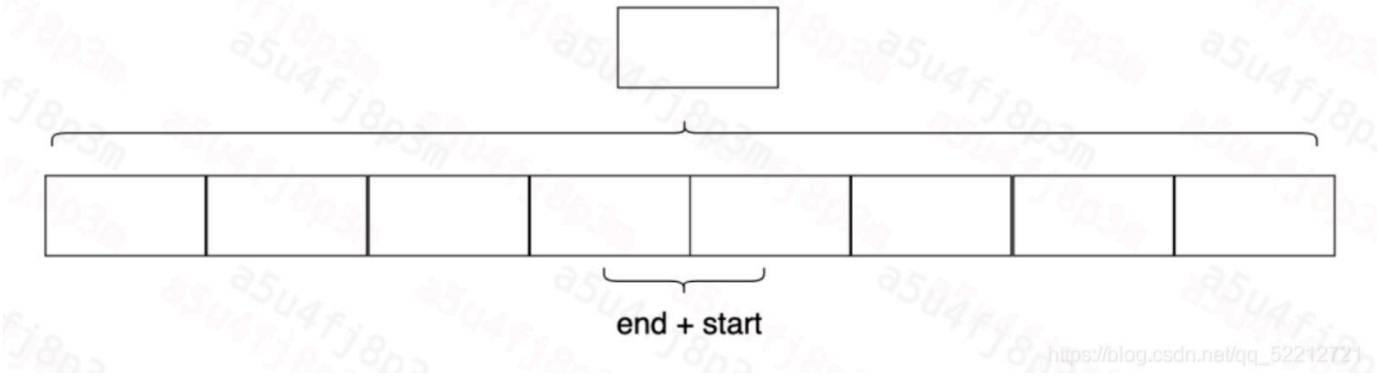
而所有的chunk pallocData都在pageAlloc结构中进行管理。

```
type pageAlloc struct {
chunks [1 << pallocChunksL1Bits]*[1 << pallocChunksL2Bits]pallocData
}
```

当所有的内存分配过大或者当前chunk块没有连续的npages空间时，就需要到基数树中从上到下进行查找。基数树有一个特性，即当要分配的内存越大时，它能够越快的查找到当前的基数树中是否有连续的空间能够满足。

在查找基数树的过程中，从上到下，从左到右的查找每一个节点是否符合要求。首先计算

pallocSum字段的开头start有多少连续的内存空间。如果start大于npages，说明我们已经查找到了可用的空间和地址。没有找到时，会计算pallocSum字段的max，即中间有多少连续的内存空间。如果max大于npages，我们需要继续往基数树当前节点对应的下一级继续查找。原因在于，max大于npages，表明当前一定有连续的空间满足npages，但是我们并不知道具体在哪一个位置，必须要继续往下一级查找时才能找到具体可用的地址。如果max也不满足，是不是就不满足了呢？不一定，因为有可能两个节点可以合并起来组成一个更大的连续空间。因此还需要将当前pallocSum计算的end与后一个节点的start加起来查看是否能够组合成大于npages的连续空间。



每一次从基数树中查找到内存，或者事后从操作系统分配内存的时候，都需要更新基数树中每一个节点的pallocSum。

当在基数树都查找不到可用的连续内存时，就需要从操作系统中索取内存。

参考：

图解Go语言内存分配

exfat 分配单元大小_理解 Go 内存管理之内存分配

Golang 内存管理

深入理解Go-垃圾回收机制

golang 垃圾回收GC的深层原理

Golang 内存组件之mspan、mcache、mcentral 和 mheap 数据结构

golang源码解析--内存mspan,mcache结构体