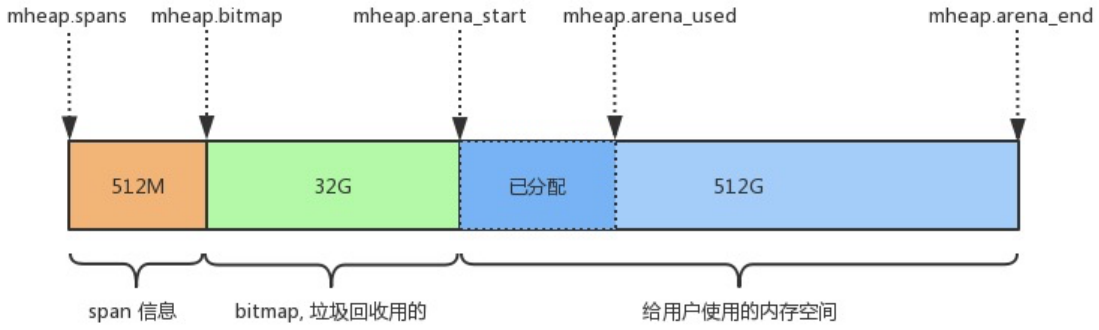
1、golang内存管理基础结构概述

1、概述

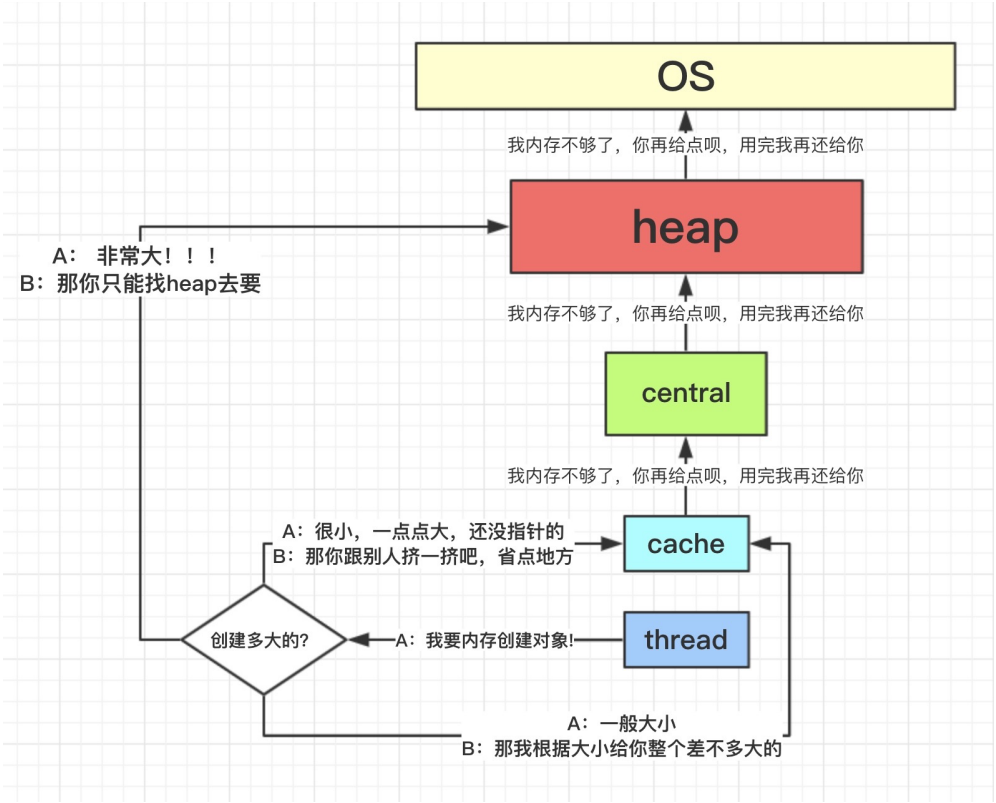
Golang的内存分配器是基于TCMalloc实现的。Golang 的程序在启动之初，会一次性从操作系统那里申请一大块内存(初始堆内存应该是 64M 左右)作为内存池。这块内存空间会放在一个叫 mheap 的 struct 中管理，mheap 负责将这一整块内存切割成不同的区域(spans, bitmap ,areana)，并将其中一部分的内存切割成合适的大小，分配给用户使用。

预申请的内存划分为spans、bitmap、arena三部分。其中arena即为所谓的堆区，应用中需要的内存从这里分配。其中spans和bitmap是为了管理arena区而存在的。
arena的大小为512G，为了方便管理把arena区域划分成一个个的page，每个page为8KB,一共有512GB/8KB个页；
spans区域存放span的指针，每个指针对应一个page，所以span区域的大小为(512GB/8KB)乘以指针大小8byte = 512M
bitmap区域大小也是通过arena计算出来，不过主要用于GC。



2、基础概念

概念	描述
page	内存页，一块8K大小的内存空间。Go与操作系统之间的内存申请和释放，都是以page为单位的。
mheap	堆分配器，以8192byte页进行管理
mspan	由mheap管理的页面
mcentral	所有给定大小类的mspan集合，Central组件其实也是一个缓存，但它缓存的不是小对象内存块，而是一组一组的内存page
mcache	运行时分配池，每个线程都有自己的局部内存缓存mCache，实现goroutine高并发的重要因素(分配小对象可直接从mCache中分配，不用加锁)
arena	区域就是heap，是供分配维护的内存池，对应区域大小是512G；
bitmap	区域是标识arena中那些地址保存了对象，及对象中是否包含了指针，其中1个byte（8bit）对应arena中4个指针大小的内存（即：2bit对应1个指针大小），对应大小16G；
span	是页管理单元，是内存分配的基本单位，其中一个指针对应arena中1个虚拟地址页大小（8kb），对应大小512M
sizeclass	空间规格，每个span都带有一个sizeclass，标记着该span中的page应该如何使用。使用上面的比喻，就是sizeclass标志着span是一个什么样的队伍。
object	对象，用来存储一个变量数据内存空间，一个span在初始化时，会被切割成一堆等大的object。假设object的大小是16B，span大小是8K，那么就会把span中的page就会被初始化8K/16B=512个object。所谓内存分配，就是分配一个object出去。



分配流程

1. 大对象： >32KB 的对象，直接从heap上分配
2. 小对象： 16B < obj <= 32KB 计算规格在mcache中找到合适大小的mspan进行分配（你有多大就住多大的房子竟可能的不要浪费房子的空间）
3. 微小对象： <=16B 的对象使用mcache的tiny分配器分配；（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。）

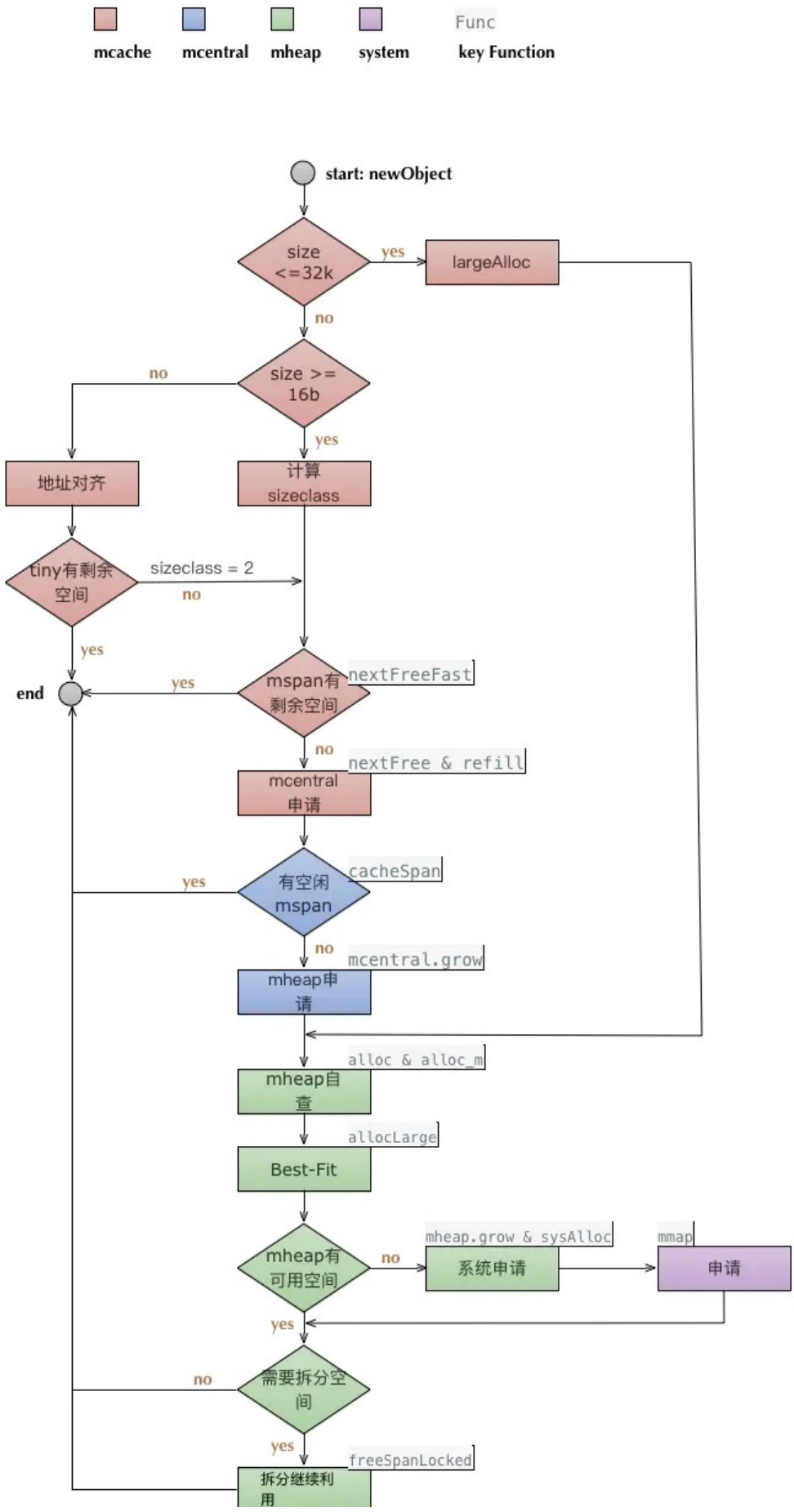
秉持原则：给到最合适的大小，然后能凑一起的凑一起挤一挤

内存分配算法

针对待分配对象的大小不同有不同的分配逻辑：

- 大于 32K 的大对象直接从 **mheap** 分配。
- 小于 16B 的使用 **mcache** 的微型分配器分配
- 对象大小在 16B ~ 32K 之间的的，首先通过计算使用的大小规格，然后使用 **mcache** 中对应大小规格的块分配
- 如果对应的大小规格在 **mcache** 中没有可用的块，则向 **mcentral** 申请
- 如果 **mcentral** 中没有可用的块，则向 **mheap** 申请，并**根据 BestFit 算法找到最合适的 mspan**。如果申请到的 **mspan** 超出申请大小，将会根据需求进行切分，以返回用户所需的页数。剩余的页构成一个新的 **mspan** 放回 **mheap** 的空闲列表。
- 如果 **mheap** 中没有可用 span，则向操作系统申请一系列新的页（最小 1MB）。

以申请size为n的内存为例，分配步骤如下：


mcache
mcentral
mheap
system
Func
key Function
start: newObject
size <=32k
yes
largeAlloc
no
size >= 16b
no
yes
地址对齐
计算 sizeclass
tiny有剩余空间
sizeclass = 2
no
yes
end
yes
mspan有剩余空间
nextFreeFast
no
mcentral 申请
nextFree & refill
yes
有空闲 mspan
cacheSpan
no
mheap申请
mcentral.grow
alloc & alloc_m
mheap自查
allocLarge
Best-Fit
mheap有可用空间
no
mheap.grow & sysAlloc
系统申请
mmap
申请
yes
no
需要拆分空间
yes
freeSpanLocked
拆分继续利用

tiny object allocation

对于小于maxTinySize(16B)字节对象的内存分配请求。go采取了将小对象合并存储的解决方案。在线程的本地缓存中维护了专门的区域(mcache.tiny)来存储tiny object。

在请求tiny object内存分配的时候，首先查看mcache.tiny的剩余空间是否能够满足tiny object对象的分配。如果足够则直接返回；如果mcache.tiny的内存不够分配，则需要向上一届mcache, mcentral, mheap, system依次申请内存，然后再分配。

```
if noscan && size < maxTinySize {
        off := c.tinyoffset
        // Align tiny pointer for required (conservative) alignment.
        //
        //   1. 8B8B
        //   2. 4B8B4B
        //   3. 1B4B2B
        //   4. 1B TODO lmj
        if size&7 == 0 {
                off = round(off, 8)
        } else if size&3 == 0 {
                off = round(off, 4)
        } else if size&1 == 0 {
                off = round(off, 2)
        }
        /* c.tiny!=0mcachetiny object+size <= maxTinySize
        mcache
        */
        if off+size <= maxTinySize && c.tiny != 0 {
                // The object fits into existing tiny block.
                // x
                x = unsafe.Pointer(c.tiny + off)
                // c.tinyoffsetsize
                c.tinyoffset = off + size
                c.local_tinyallocs++
                mp.mallocing = 0
                releasem(mp)
                return x
        }
        // Allocate a new maxTinySize block.
        /*
        mcache+spanClass(span)
        spanspanspan
         */
        span := c.alloc[tinySpanClass]
        // mcachespan
        v := nextFreeFast(span)
        // v == 0 span
        if v == 0 {
                // mcentral
                v, _, shouldhelpgc = c.nextFree(tinySpanClass)
        }
        //
        x = unsafe.Pointer(v)
        //  64/8 = 8B8B=16BtinySpan16B
```

```
        (*[2]uint64)(x)[0] = 0
        (*[2]uint64)(x)[1] = 0
        // See if we need to replace the existing tiny block with the new
one
        // based on amount of remaining free space.
        // free spacetiny blocktiny block
        if size < c.tinyoffset || c.tiny == 0 {
                c.tiny = uintptr(x)
                c.tinyoffset = size
        }
        size = maxTinySize
}
```

big object allocation

对于32KB的对象，跳过mcache和mcentral，直接在mheap上进行分配。

```
var s *mspan
// mheapheavyshouldhelpgctrue
shouldhelpgc = true
systemstack(func() {
  //
  s = largeAlloc(size, needzero, noscan)
})
s.freeindex = 1
s.allocCount = 1
x = unsafe.Pointer(s.base())
size = s.elemsize
```

small object allocation

对于 >=16B && <= 32KB的对象

- 如果 mcache 对应的 size class 的 span 已经没有可用的块，则向 mcentral 请求。
- 如果 mcentral 也没有可用的块，则向 mheap 申请，并切分。
- 如果 mheap 也没有合适的 span，则向操作系统申请。

```
// sizeclass()
var sizeclass uint8
if size <= smallSizeMax-8 {
  sizeclass = size_to_class8[(size+smallSizeDiv-1)/smallSizeDiv]
} else {
  sizeclass = size_to_class128[(size-smallSizeMax+largeSizeDiv-1)
/largeSizeDiv]
}
size = uintptr(class_to_size[sizeclass])
spc := makeSpanClass(sizeclass, noscan)
span := c.alloc[spc]
// tiny objectmcache,mcentral,mheap,os
v := nextFreeFast(span)
if v == 0 {
  v, span, shouldhelpgc = c.nextFree(spc)
}
x = unsafe.Pointer(v)
if needzero && span.needzero != 0 {
  //
  memclrNoHeapPointers(unsafe.Pointer(v), size)
}
```

SpanClass

根据对象大小，划分了一系列class，每个class都代表一个固定大小的对象，以及每个span的大小。如下表所示：

```
// class  bytes/obj  bytes/span  objects  waste bytes
//     1          8        8192     1024            0
//     2         16        8192      512            0
//     3         32        8192      256            0
//     4         48        8192      170           32
//     5         64        8192      128            0
//     6         80        8192      102           32
//     7         96        8192       85           32
//     8        112        8192       73           16
//     9        128        8192       64            0
//    10        144        8192       56          128
//    11        160        8192       51           32
//    12        176        8192       46           96
//    13        192        8192       42          128
//    14        208        8192       39           80
//    15        224        8192       36          128
//    16        240        8192       34           32
//    17        256        8192       32            0
//    18        288        8192       28          128
//    19        320        8192       25          192
//    20        352        8192       23           96
//    21        384        8192       21          128
//    22        416        8192       19          288
```

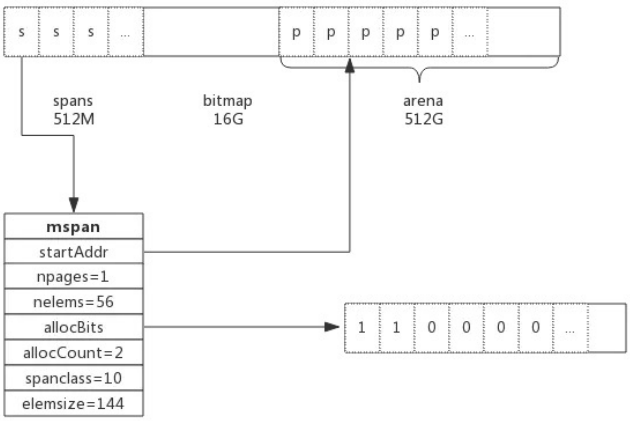
```
//    23        448        8192       18          128
//    24        480        8192       17           32
//    25        512        8192       16            0
//    26        576        8192       14          128
//    27        640        8192       12          512
//    28        704        8192       11          448
//    29        768        8192       10          512
//    30        896        8192        9          128
//    31       1024        8192        8            0
//    32       1152        8192        7          128
//    33       1280        8192        6          512
//    34       1408       16384       11          896
//    35       1536        8192        5          512
//    36       1792       16384        9          256
//    37       2048        8192        4            0
//    38       2304       16384        7          256
//    39       2688        8192        3          128
//    40       3072       24576        8            0
//    41       3200       16384        5          384
//    42       3456       24576        7          384
//    43       4096        8192        2            0
//    44       4864       24576        5          256
//    45       5376       16384        3          256
//    46       6144       24576        4            0
//    47       6528       32768        5          128
//    48       6784       40960        6          256
//    49       6912       49152        7          768
//    50       8192        8192        1            0
//    51       9472       57344        6          512
//    52       9728       49152        5          512
//    53      10240       40960        4            0
//    54      10880       32768        3          128
//    55      12288       24576        2            0
//    56      13568       40960        3          256
//    57      14336       57344        4            0
//    58      16384       16384        1            0
//    59      18432       73728        4            0
//    60      19072       57344        3          128
//    61      20480       40960        2            0
//    62      21760       65536        3          256
//    63      24576       24576        1            0
//    64      27264       81920        3          128
//    65      28672       57344        2            0
//    66      32768       32768        1            0



class class IDspanclass ID, span
bytes/objclass
bytes/spanspan
objects: spanbytes/spans/bytes/objwaste
bytes: spanbytes/spans%bytes/obj32K32Kclassclass ID0class
```

span数据结构

span是内存管理的基本单位,每个span用于管理特定的class对象, 跟据对象大小，span将一个或多个页拆分成多个块进行管理。src/runtime/mheap.go:mspan定义了其数据结构：

```
type mspan struct {
    next *mspan            //span
    prev *mspan            //span
    startAddr uintptr //
    npages    uintptr //


        freeindex uintptr //
    nelems uintptr //

    allocBits  *gcBits //

    allocCount  uint16     //
    spanclass   spanClass  // classclass ID

    elemsize    uintptr    // class
}
```

以class 10为例，span和管理的内存如下图所示：



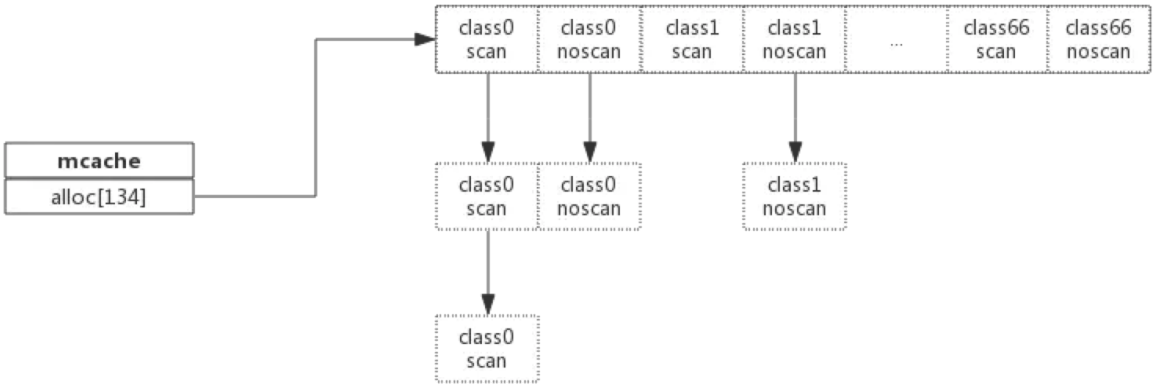
spanclass为10，参照class表可得出npages=1,nelems=56,elemsize为144。其中startAddr是在span初始化时就指定了某个页的地址。allocBits指向一个位图，每位代表一个块是否被分配，本例中有两个块已经被分配，其allocCount也为2。next和prev用于将多个span链接起来，这有利于管理多个span，接下来会进行说明。

cache

有了管理内存的基本单位span，还要有个数据结构来管理span，这个数据结构叫mcentral，各线程需要内存时从mcentral管理的span中申请内存，为了避免多线程申请内存时不断的加锁，Golang为每个线程分配了span的缓存，这个缓存即是cache。src/runtime/mcache.go:mcache定义了cache的数据结构

```
type mcache struct {
    alloc [67*2]*mspan // classmspan
}
```

alloc为mspan的指针数组，数组大小为class总数的2倍。数组中每个元素代表了一种class类型的span列表，每种class类型都有两组span列表，第一组列表中所表示的对象中包含了指针，第二组列表中所表示的对象不含有指针，这么做是为了提高GC扫描性能，对于不包含指针的span列表，没必要去扫描。根据对象是否包含指针，将对象分为noscan和scan两类，其中noscan代表没有指针，而scan则代表有指针，需要GC进行扫描。mcache和span的对应关系如下图所示：



mchache在初始化时是没有任何span的，在使用过程中会动态的从central中获取并缓存下来，跟据使用情况，每种class的span个数也不相同。上图所示，class 0的span数比class1的要多，说明本线程中分配的小对象要多一些。

central

cache作为线程的私有资源为单个线程服务，而central则是全局资源，为多个线程服务，当某个线程内存不足时会向central申请，当某个线程释放内存时又会回收进central。src/runtime/mcentral.go:mcentral定义了central数据结构：

```
type mcentral struct {
        lock      mutex    // P
        spanclass spanClass  // mspanspanclass
        nonempty  mSpanList // mcentralmspan
        empty     mSpanList // mcentralmspan
}
```

lock: 线程间互斥锁，防止多线程读写冲突
spanclass : 每个mcentral管理着一组有相同class的span列表
nonempty: 指还有内存可用的span列表
empty: 指没有内存可用的span列表
nmalloc: 指累计分配的对象个数线程从central获取span步骤如下：

1. 加锁
2. 从nonempty列表获取一个可用span，并将其从链表中删除
3. 将取出的span放入empty链表
4. 将span返回给线程
5. 解锁
6. 线程将该span缓存进cache线程

将span归还步骤如下：

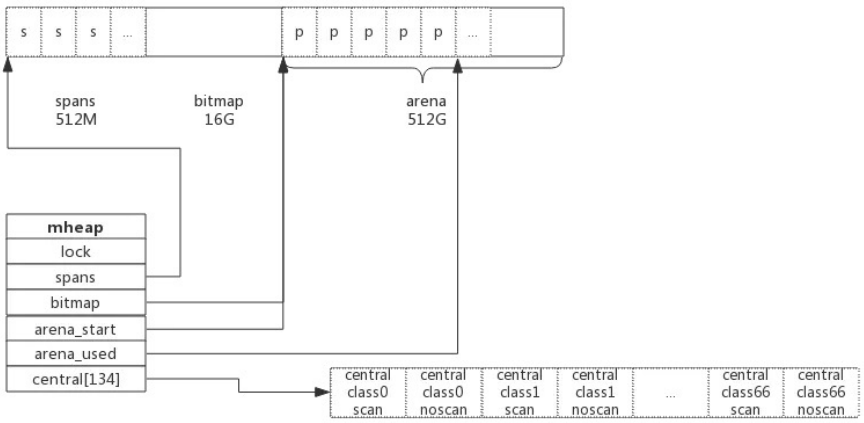
1. 加锁
2. 将span从empty列表删除
3. 将span加入noneempty列表
4. 解锁上述线程从central中获取span和归还span只是简单流程，为简单起见，并未对具体细节展开。

heap

从mcentral数据结构可见，每个mcentral对象只管理特定的class规格的span。事实上每种class都会对应一个mcentral,这个mcentral的集合存放于mheap数据结构中。src/runtime/mheap.go:mheap定义了heap的数据结构：

```
type mheap struct {
    lock      mutex

    free      [_MaxMHeapList]mSpanList // spanlistfree[3]3 page  mspan
        allspans []*mspan                   //  mspan  allspans

    allArenas []arenaIdx          // arena

    central [67*2]struct {
        mcentral mcentral
        pad      [sys.CacheLineSize - unsafe.Sizeof(mcentral{})%sys.
CacheLineSize]byte
    }
}
```

从数据结构可见，mheap管理着全部的内存，事实上Golang就是通过一个mheap类型的全局变量进行内存管理的。mheap内存管理示意图如下：



系统预分配的内存分为spans、bitmap、arean三个区域，通过mheap管理起来。接下来看内存分配过程。

总结

Golang内存分配是个相当复杂的过程，其中还掺杂了GC的处理，这里仅仅对其关键数据结构进行了说明，了解其原理而又不至于深陷实现细节。1、Golang程序启动时申请一大块内存并划分成spans、bitmap、arena区域
2、arena区域按页划分成一个个小块。

3、span管理一个或多个页。
4、mcentral管理多个span供线程申请使用
5、mcache作为线程私有资源，资源来源于mcentral。