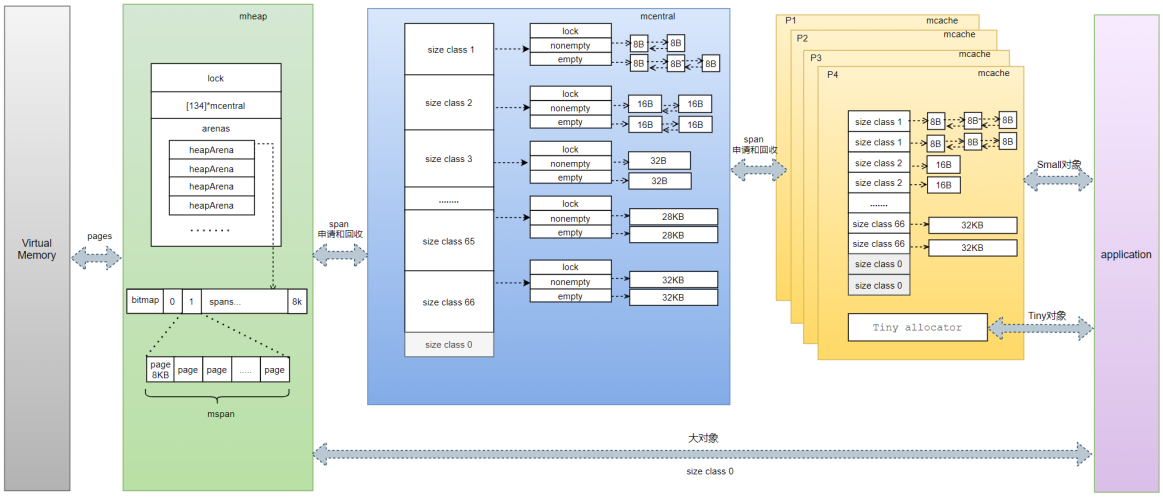
Golang的内存管理的核心思想就是完成类似预分配、内存池等操作，以避开系统调用带来的性能问题，防止每次分配内存都需要系统调用。

下图是Golang内存管理流程图 PS：如果有什么错请各位大佬指正



对图中几个名词进行解释

page

mheap向虚拟内存申请的最小单位。一般为8KB

span

go内存分配的基本单位，有n个page组成

class size

为了减少内存碎片，将span的大小分级。目前分为0-66级共67级。可以看到class=0是没有使用的(图中也标为灰色)

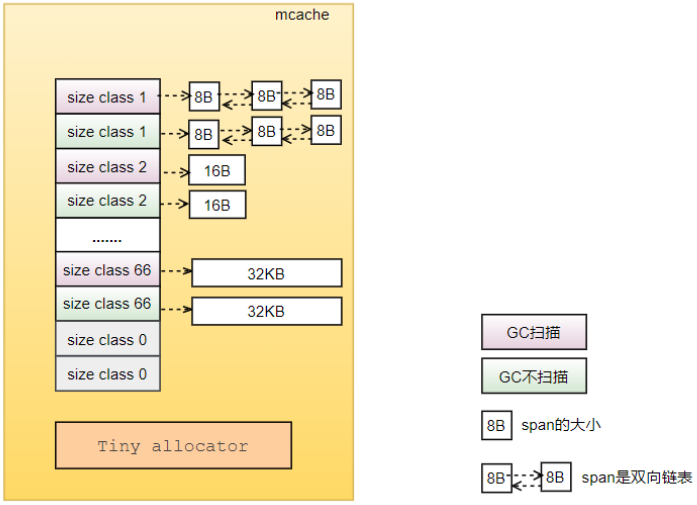
66种span如下： (v.14.13)

```
// class  bytes/obj  bytes/span  objects  tail waste  max waste
//     1          8        8192     1024           0     87.50%
//     2         16        8192      512           0     43.75%
//     3         32        8192      256           0     46.88%
//     4         48        8192      170          32     31.52%
//     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%
```

bytes/obj 指的是span的大小，可以看到范围是8B~32KB
bytes/span 指的是占用堆的字节数，也就是页数*页大小 (eg:8192=1x8k)
objects 值得是该span可以分配对象的个数（eg:1024=8192/8）
tail waste 产生的内存碎片 (eg:32=8192%48)

mcache



mcache是分配给M运行中的goroutine，是协程级所以无需加锁。为什么不用加锁呢，是因为在M上运行的goroutine只有一个，不会存在抢占资源的情况，所以是无需加锁的。

从上图中可以看到，mache供2种类型的对象分配内存。一个是微对象[1B,16B)，一个是小对象[16B,32KB]。

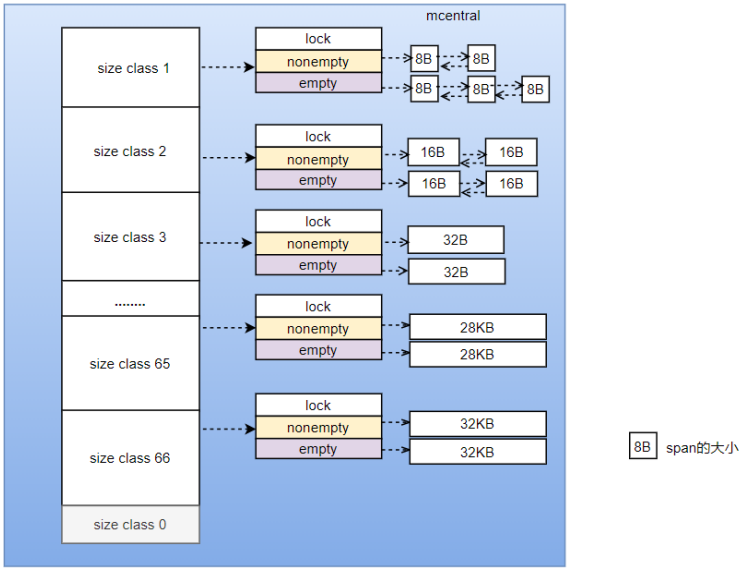
在图中可以看到，对于微对象的内存分配是由mcache提供专门的Tiny allocator专门进行分配，具体的分配流程后续会介绍。

小对象是选择最适应自己大小的span进行分配，从图中可以可以看到同一级别的span是分成2类的，一类是可以被GC扫描的span,里面是包含指针的对象；另一类似不可以被GC扫描的span,里面不包含指针的对象。可以看到分配内存的时候会按照是否有指针对象对应不同的span，为了后续GC垃圾回收使用。

每一个级别的span链表是一个双向链表，每一个span都会指向前一个span和后一个span。每一级size class可以有1个或者多个span

当小对象申请内存在mache不够时，会继续向mcentral进行申请

mcentral



mcentral是为mcache提供切分好的span。mcentral是全局的，也就是多个M共享mcentral，会出现并发问题，所以此时申请都是需要加锁的。

mcentral存储67级别大小span，其中size=0是不使用的(图中标灰色）。每一级别的span分为2种，一种empty表示这条链的mspan已经被分配了对象，或者已经被mcache使用，被对应线程占用；nonempty表示有空闲对象的 mspan列表

值得注意的是mcentral链表都在mheap进行维护

若分配内存是没有空闲的span的列表，此时需要像mheap申请。

mheap

mheap是go程序持有的整个堆空间，是go的全局变量，所以在使用的时候需要全局锁。

大对象(大于32KB)直接通过mheap进行分配。除此之外，mcentrals保存在mheap中，mheap对mcentral了如指掌。

若mheap没有足够的内存，则会向虚拟内存申请page，然后将page组装成span再供程序使用。

mheap还存储多个heapArena ，heapArena 存储连续的span，主要是为了mheap管理span和GC垃圾回收

微对象 [1B,16B)

微对象的内存分配是由mcache提供专门的Tiny allocator专门进行分配，分配的对象是不包含指针的，例如一些小的字符串和不包含指针的独立逃逸变量等。

小对象 [16B,32KB]

小对象是在mache申请适合自己大小的span，若mache没有可用的span，mache会向mcentral申请，加锁，找一个可用的span，从nonempty删除该span，然后放到empty链表中，将span返回给工作线程，解锁；若没有足够的内存，mcentral还会继续向mheap继续申请。

当归还时，加锁，将empty链表删除对应的span，然后将其加到nonempty链表中，解锁。

大对象(32KB,+∞)

大对象，使用mheap直接分配，若mheap没有足够的内存，则mheap向虚拟内存申请若干个pages。可以看到，约到后面申请内存的代价就越来越大

回头看这样的内存管理的优点是什么

1、可以看到申请内存的时候是以span为单位的，span又分为不同大小，从大小的规律我们可以看到不是简单的按照2次幂进行递增的，是根据计算造成碎片最少的情况下对span的分类，在申请的时候会减少内存碎片。比如在申请47B大小的时候，如果按照2次幂会提供64B大小的内存供应用使用，但是如果按照span会提供48B大小的span，很明显看出，后者造成的碎片会更少。

2、每次从操作系统申请一大块内存，由Go来做分配，减少了系统调用

3、go的内存算法是使用google的TCMalloc内存管理算法，把内存分的非常细，分为多级管理，减少锁的粒度。在回收对象内存时，并没有将其真正释放掉，只是放回预先分配的大块内存中，以便复用。只有内存闲置过多的时候，才会尝试归还部分内存给操作系统，降低整体开销