24 | GC实例：Python和Go的内存管理机制是怎样的？

2021-12-22 海纳

《编程高手必学的内存知识》 课程介绍 >

讲述：海纳

时长 24:40 大小 22.59M

你好，我是海纳。

我们前面几节课主要是以 Java 为例，介绍了 JVM 中垃圾回收算法的演进过程。实际上，除了 JVM 之外，用于运行 JavaScript 的 V8 虚拟机、Lua 虚拟机、Python 虚拟机和 Go 的虚拟机都采用了自动内存管理技术。这节课，我们就一起来分析一下它们的实现。

通过这节课，你将会看到垃圾回收算法的设计是十分灵活而且多种多样的，这会为你以后改进应用中自动或半自动的内存管理，提供很好的参考。你要注意的是，学习自动内存管理，一定要抓住核心原理，不要陷入到细节里去。另外，你可以通过查看虚拟机源代码，验证自己的猜想，但不要把源代码教条化。

接下来，我先解释一下为什么选择 Python 和 GO 这两种语言做为例子。

静态语言和动态语言

我先介绍两个基本概念：**动态语言和静态语言**。动态语言的特征是可以在运行时，为对象甚至是类添加新的属性和方法，而静态语言不能在运行期间做这样的修改。

动态语言的代表是 Python 和 JavaScript，静态语言的代表是 C++。Java 本质上是一门静态语言，但它又提供了反射（reflection）的能力为动态性开了一个小口子。

从实现的层面讲，静态语言往往在编译时就能确定各个属性的偏移值，所以编译器能确定某一种类型的对象，它的大小是多少。这就方便了在分配和运行时快速定位对象属性。关于静态语言的对象内存布局，我们在导学（三）和第 19 节课都做了介绍，你可以去看看。

而动态语言往往会将对象组织成一个字典结构，例如下面这个 Python 的例子：

复制代码

```
1  >>> class A(object):
2  ...  pass
3  ...
4  >>> a = A()
5  >>> a.b = 1
6  >>> b = A()
7  >>> b.c = 2
8  >>> b.__dict__
9  {'c': 2}
10 >>> a.__dict__
11 {'b': 1}
```

你可以看到，类 A 的两个对象 a 和 b，它们的属性是不相同的。这在 Java 语言中是很难想象的，但是在 Python 或者 JavaScript 中，却是司空见惯的操作。

如果把上述代码中的 a 对象和 b 对象想象成一个字典的话，这段代码就不难理解了。第 5 行和第 6 行的操作不过是相当于在字典中添加了新的一个键值对而已。

在了解了动态语言和静态语言的区别以后，我们就从动态语言中选择 Python 语言，从静态语言中选择 Go 语言，来对两种语言的实现加以解释。其中，Python 语言的分配过程与第 9 节课所讲的 malloc 的实现非常相似，所以我们重点看它的垃圾回收过程。Go 语言

的垃圾回收过程就是简单的 CMS，所以我们重点分析它的分配过程。下面，我们先从 Python 语言开始吧。

Python 的内存管理机制

我们先从 Python 的对象布局讲起，以最简单的浮点数为例，在 Include/floatobject.h 中，python 中的浮点数是这么定义的：

复制代码

```
typedef struct {
    PyObject_HEAD
    double ob_fval;
} PyFloatObject;
```

我们继续查看 PyObject_HEAD 的定义：

复制代码

```
/* PyObject_HEAD defines the initial segment of every PyObject. */
#define PyObject_HEAD                   \
    _PyObject_HEAD_EXTRA                \
    Py_ssize_t ob_refcnt;               \
    struct _typeobject *ob_type;
```

这是一个宏定义，而其中的 EXTRA 在正常编译时是空的。所以，我们直接展开所有宏，那么 PyFloatObject 的定义就是这样子的：

复制代码

```
typedef struct {
    Py_ssize_t ob_refcnt;
    struct _typeobject *ob_type;
    double ob_fval;
} PyFloatObject;
```

这样就很清楚了，ob_refcnt 就是引用计数，而 ob_fval 是真正的值。例如我写一段这样的代码：

复制代码

```
a = 1000.0
a = 2000.0
```

在执行第 1 句时，Python 虚拟机真正执行的逻辑是创建一个 PyFloatObject 对象，然后使它的 ob_fval 为 1000.0，同时，它的引用计数为 1；当执行到第 2 句时，创建一个值为 2000.0 的 PyFloatObject 对象，并且使这个对象的引用计数为 1，而前一个对象的引用计数就要减 1，从而变成 0。那么前一个对象就会被回收。

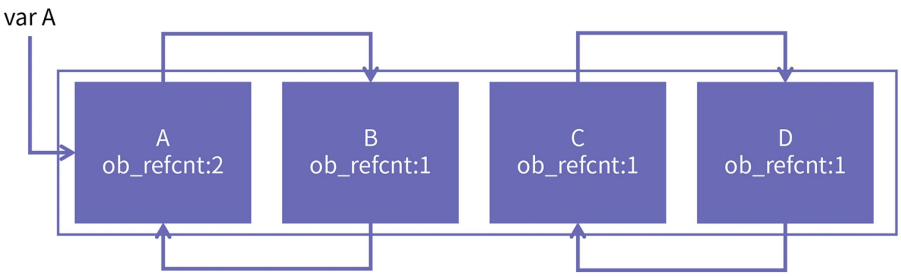
在 Python 中，引用计数的维护是通过这两个宏来实现的：

复制代码

```
#define Py_INCREF(op) (                         \
    _Py_INC_REFTOTAL  _Py_REF_DEBUG_COMMA       \
    ((PyObject*)(op))->ob_refcnt++)
#define Py_DECREF(op)                                   \
    do {                                                \
        if (_Py_DEC_REFTOTAL  _Py_REF_DEBUG_COMMA       \
        --((PyObject*)(op))->ob_refcnt != 0)            \
            _Py_CHECK_REFCNT(op)                        \
        else                                            \
        _Py_Dealloc((PyObject *)(op));                  \
    } while (0)
```

这两个宏位于 Include/object.h 中。这段代码里最重要的地方在于 ob_refcnt 增一和减一的操作。这段代码与第 19 节课所讲的引用计数法的伪代码十分相似，我就不重复分析了。

使用了引用计数的地方，就会存在循环引用。例如下图中的四个对象，A 是根对象，它与 B 之间有循环引用，那么它们都不是垃圾对象。C 和 D 之间也有循环引用，但因为没有外界的引用指向它们了，所以它们就是垃圾对象，但是循环引用导致他们都不能释放。

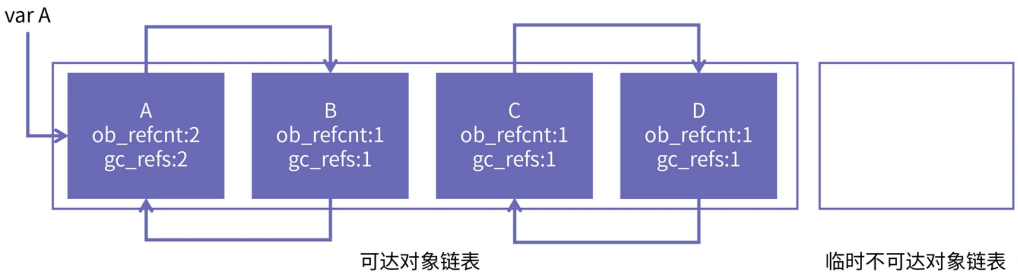


Python 为了解决这个问题，在虚拟机中引入了一个双向链表，把所有对象都放到这个链表里。Python 的每个对象头上都有一个名为 PyGC_Head 的结构：


```
/* GC information is stored BEFORE the object structure. */
typedef union _gc_head {
    struct {
        union _gc_head *gc_next;
        union _gc_head *gc_prev;
        Py_ssize_t gc_refs;
    } gc;
    long double dummy;  /* force worst-case alignment */
} PyGC_Head;
```

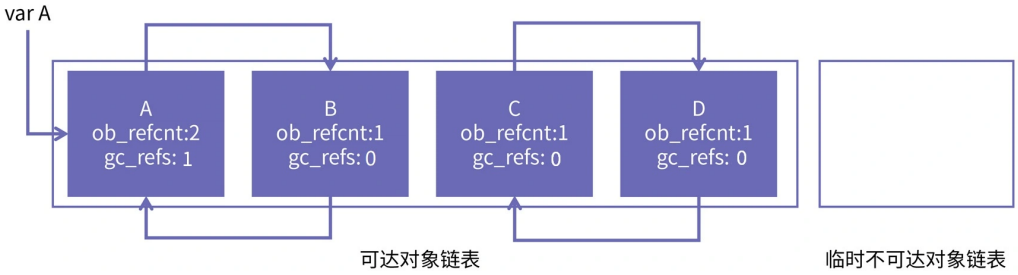
在这个结构里，gc_next 和 gc_prev 的作用就是把对象关联到链表里。而 gc_refs 则是用于消除循环引用的。当链表中的对象达到一定数目时，Python 的 GC 模块就会执行一次标记清除。具体来讲，一共有四步。

第一步，**将 ob_refcnt 的值复制到 gc_refs 中**。对于上面的例子，它们的 gc_refs 的值就如下图所示：



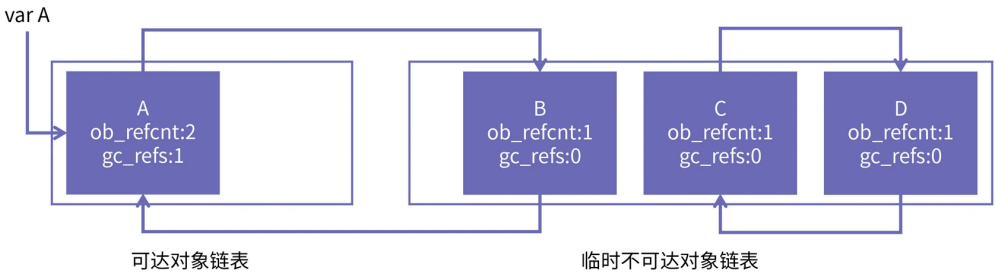
极客时间

第二步是**遍历整个链表，对每个对象，将它直接引用的对象的 gc_refs 的值减一**。比如遍历到 A 对象时，只把 B 对象的 gc_refs 值减一；遍历到 B 对象时，再把它直接引用的 A 对象的 gc_refs 值减一。经过这一步骤后，四个对象的 gc_refs 的值如下图所示：



极客时间

第三步，**将 gc_refs 值为 0 的对象，从对象链表中摘下来，放入一个名为“临时不可达”的链表中**。之所以使用“临时”，是因为有循环引用的垃圾对象的 gc_refs 在此时一定为 0，比如 C 和 D。但 gc_refs 值为 0 的对象不一定是垃圾对象，比如 B 对象。此时，B、C 和 D 对象就被放入临时不可达链表中了，示意图如下所示：



极客时间

最后一步，**以可达对象链表中的对象为根开始深度优先搜索，将所有访问到 gc_refs 为 0 的对象，再从临时不可达链表中移回可达链表中**。最后留在临时不可达链表中的对象，就是真正的垃圾对象了。

接下来就可以使用 _Py_Dealloc 逐个释放链表中的对象了，对于上面的例子，就是把 B 对象重新加回到可达对象链表中，然后将 C 和 D 分别释放。

到这里，Python 内存管理的核心知识我们就介绍完了，接下来，我们来看 Go 语言的例子。它的特点是分配算法复杂，但清除算法简单。

Go 语言的内存管理机制

Go 语言采用的垃圾回收算法是**并发标记清理**（Concurrent Mark Sweep，CMS）算法。CMS 算法是 Tracing GC 算法中非常经典且朴素的例子，我们在前边的课程中讲了基于复制的 GC 算法和基于分区的 GC 算法，它们都在不同方面比 CMS GC 有优势，那为什么 Go 语言还是选择了 CMS 作为其 GC 算法呢？这里原因主要有两点：

第一点是，Go 语言通过内存分配策略缓解了 CMS 容易产生内存碎片的缺陷。

相对于 CMS GC，基于压缩的 GC 算法和基于复制的 GC 算法最大的优势是，能够降低内存的碎片率。这一点我们在前边的课程里已经充分讨论了。Go 的 GC 算法采用 TCMalloc 分配器的内存分配思路，虽然不能像基于 copy 的 GC 算法那样消除掉内存碎片化的问题，但也极大地降低了碎片率。

另外，基于 Thread Cache 的分配机制可以使得 Go 在分配的大部分场景下避免加锁，这使得 Go 在高并发场景下能够发挥巨大的性能优势。

第二点是，Go 语言是有值类型数据的，即 struct 类型。有了值类型的介入，编译器只需要关注函数内部的逃逸分析（intraprocedural escape analysis），而不用关注函数间的逃逸分析（interprocedural analysis），由此可以将生命周期很短的对象安排在栈上分配。

在前面的课程里，我们介绍了分代 GC 可以区分长生命周期和短生命周期对象，它的优势是能够快速回收生命周期短的对象。但由于逃逸分析的优势，Go 语言中的短生命周期对象并没有那么多，所以分代 GC 在 Go 语言中收益较低。另外，由于分代 GC 需要额外的 write barrier 来维护老年代对年轻代的引用关系，也加重了 GC 引入的开销。

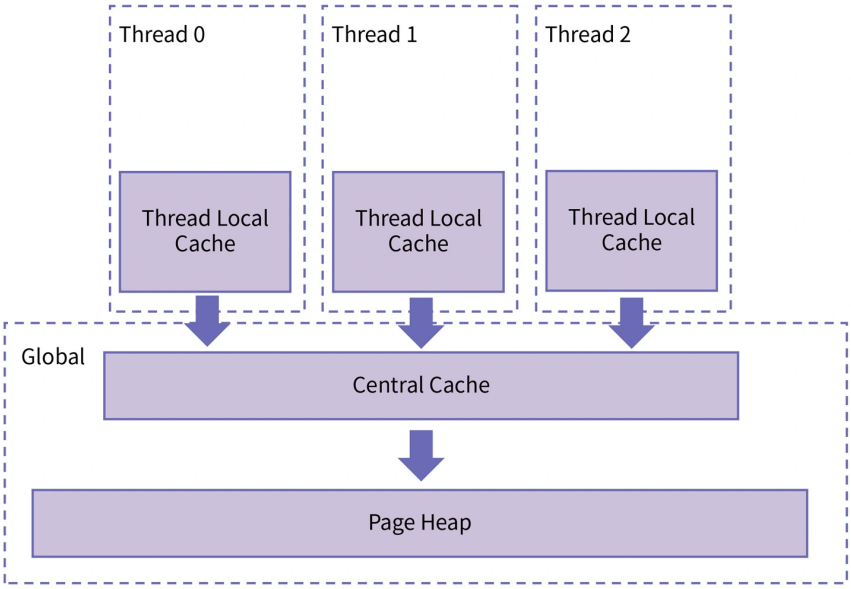
基于这两个主要原因，Go 语言目前采用的 GC 算法是 CMS 算法。当然，Go 语言在演进的过程中也曾采用过 ROC 和分代 GC 的算法，但后来也都放弃了。如果你对 Go GC 的演进过程感兴趣，可以看下《Getting to Go: The Journey of Go’s Garbage Collector》这篇文章。

在上面讲的第一点原因中，我们提到 Go 在 CMS 算法中，为了提高分配效率并且保障堆空间较低的碎片率，采用了 TCMalloc 的分配思想。所以，我们先来看一下 TCMalloc 的分配思想是怎样的，把握住 TCMalloc 的思想，你就能容易理解 Go 的分配机制了。

TCMalloc 的分配思想

在 TCMalloc 中，“TC”是 Thread Cache 的意思，其核心思想是：**TCMalloc 会给每个线程分配一个 Thread-Local Cache，对于每个线程的分配请求，就可以从自己的 Thread-Local Cache 区间来进行分配。此时因为不会涉及多线程操作，所以并不需要进行加锁，从而减少了因为锁竞争而引起的性能损耗。**

而当 Thread-Local Cache 空间不足的时候，才向下一级的内存管理器请求新的空间。TCMalloc 引入了 Thread cache、Central cache 以及 Page heap 三个级别的管理器来管理内存，可以充分利用不同级别下的性能优势。这个时候你会发现，TCMalloc 的多级管理机制非常类似计算机系统结构的内存多级缓存机制。



TCMalloc三层缓存机制

极客时间

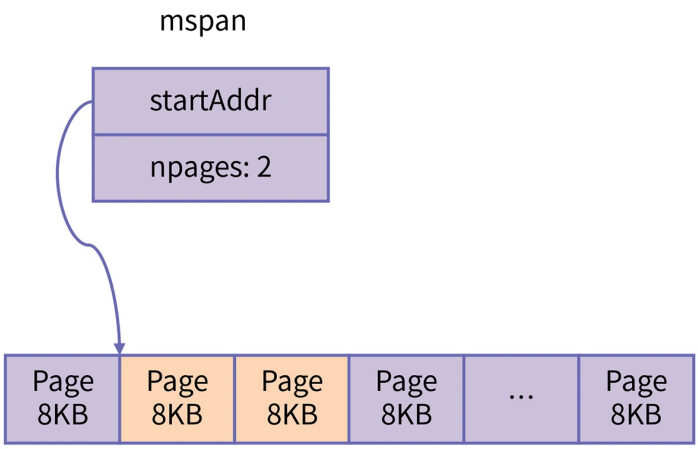
围绕着这个核心思想，我们具体看下 Go 的分配器是怎么实现的。在 Go 的内存管理机制中，有几个重要的数据结构需要关注，分别是 **mspan、heapArena、mcache、mcentral 以及 mheap**。其中，mspan 和 heapArena 维护了 Go 的虚拟内存布局，而 mcache、mcentral 以及 mheap 则构成了 Go 的三层内存管理器。

虚拟内存布局

Go 的内存管理基本单元是 mspan，每个 mspan 中会维护着一块连续的虚拟内存空间，内存的起始地址由 startAddr 来记录。每个 mspan 存储的内存空间大小都是内存页的整数倍，由 npages 来保存。不过你需要注意的是，这里内存页并非是操作系统的物理页大小，Go 的内存页大小设置的是 8KB。mspan 结构的部分定义如下：

复制代码

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

    startAddr uintptr // address of first byte of span aka s.base()
    npages    uintptr // number of pages in span
    ...
    spanclass   spanClass     // size class and noscan (uint8)
    ...
}
```



mspan内存管理单元

极客时间

heapArena 的结构相当于 Go 的一个内存块，在 x86-64 架构下的 Linux 系统上，一个 heapArena 维护的内存空间大小是 64MB。该结构中存放了 ArenaSize/PageSize 长度的 mspan 数组，heapArena 结构的 spans 变量，用来精确管理每一个内存页。而整个 arena 内存空间的基址则存放在 zeroedBase 中。heapArena 结构的部分定义如下：

复制代码

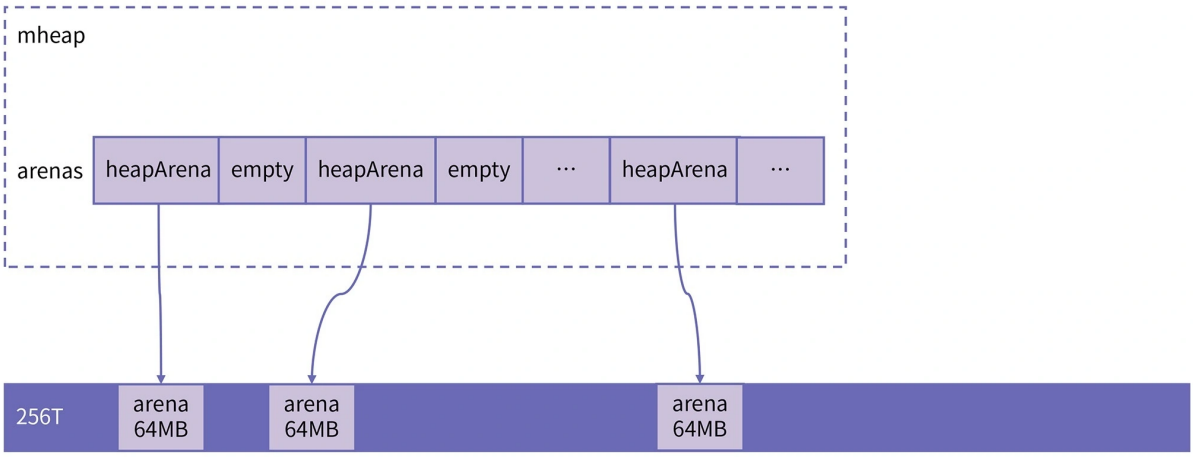
```
type heapArena struct {
    ...
    spans [pagesPerArena]*mspan
    zeroedBase uintptr
}
```

有了这两个结构，我们就可以整体看下 Go 的虚拟内存布局了。Go 整体的虚拟内存布局是存放在 mheap 中的一个 heapArena 的二维数组。定义如下：

复制代码

```
arenas [1 << arenaL1Bits]*[1 << arenaL2Bits]*heapArena
```

这里二维数组的大小在不同架构跟操作系统上有所不同，对于 x86-64 架构下的 Linux 系统，第一维数组长度是 1，而第二维数组长度是 4194304。这样每个 heapArena 管理的内存大小是 64MB，由此可以算出 Go 的整个堆空间最多可以管理 256TB 的大小。



Go的虚拟内存布局

极客时间

这里我们又会发现，Go 通过 heapArena 来对虚拟内存进行管理的方式其实跟操作系统通过页表来管理物理内存是一样的。了解了 Go 的虚拟内存布局之后，我们再来看下 Go 的三级内存管理器。

三级内存管理

在 Go 的三级内存管理器中，维护的对象都是小于 32KB 的小对象。对于这些小对象，Go 又将其按照大小分成了 67 个类别，称为 spanClass。每一个 spanClass 都用来存储固定大小的对象。这 67 个 spanClass 的信息在 runtime.sizeclasses.go 中可以看到详细的说明。我选取了一部分注释放在了下面，你可以看看。

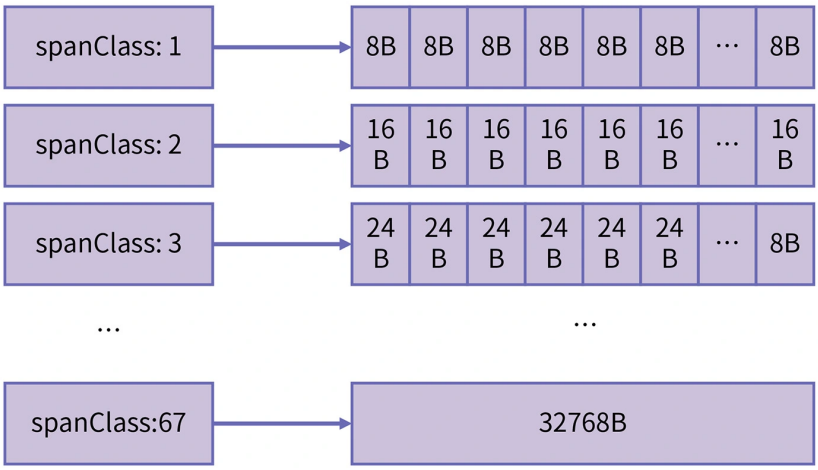
复制代码

```
// class  bytes/obj  bytes/span  objects  tail waste  max waste  min align
//     1          8        8192     1024           0     87.50%          8
//     2         16        8192      512           0     43.75%         16
//     3         24        8192      341           8     29.24%          8
//     4         32        8192      256           0     21.88%         32
//    ...
//    67      32768       32768        1           0     12.50%       8192
```

对于上面的注释，我以 class 3 为例做个介绍。class 3 是说在 spanClass 为 3 的 span 结构中，存储的对象的大小是 24 字节，整个 span 的大小是 8192 字节，也就是一个内存页的大小，可以存放的对象数目最多是 341。

tail waste 这里是 8 字节，这个 8 是通过 8192 mod 24 计算得到，意思是，当这个 span 填满了对象后，会有 8 字节大小的外部碎片。而 max waste 的计算方式则是 $[(24-17)\times 341+8]\div 8192$ 得到，意思是极端场景下，该 span 上分配的所有对象大小都是 17 字节，此时的内存浪费率为 29.24%。

以上 67 个存储小对象的 spanClass 级别，再加上 class 为 0 时用来管理大于 32KB 对象的 spanClass，共总是 68 个 spanClass。这些数据都是通过在 runtime.mksizeclasses.go 中计算得到的。我们从上边的注释可以看出，Go 在分配的时候，是通过控制每个 spanClass 场景下的最大浪费率，来保障堆内存在 GC 时的碎片率的。



67级别spanClass

另外，spanClass 的 ID 中还会通过最后一位来存放 noscan 的属性。这个标志位是用来告诉 Collector 该 span 中是否需要扫描。如果当前 span 中并不存放任何堆上的指针，就意味着 **Collector 不需要扫描这段 span 区间**。

复制代码

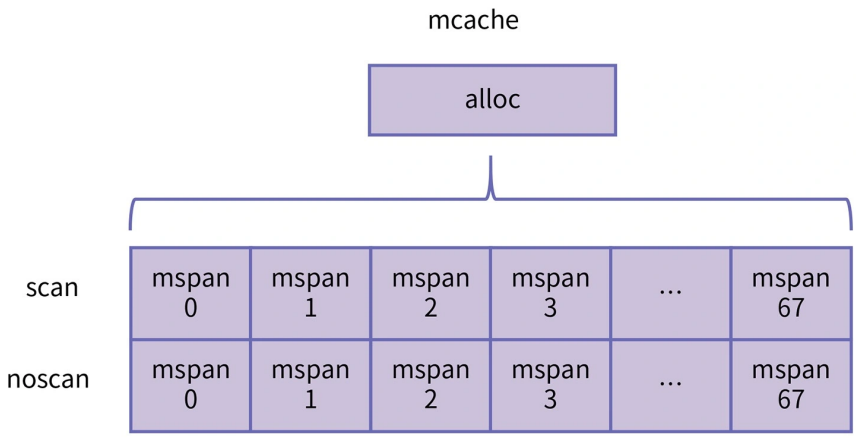
```
func makeSpanClass(sizeclass uint8, noscan bool) spanClass {
    return spanClass(sizeclass<<1) | spanClass(bool2int(noscan))
}

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

```
}

func (sc spanClass) noscan() bool {
    return sc&1 != 0
}
```

好了，接下来我们继续看下 mcache 的结构。mcache 是 Go 的线程缓存，对应于 TCMalloc 中的 Thread cache 结构。mcache 会与线程绑定，每个 goroutine 在向 mcache 申请内存时，都不会与其他 goroutine 发生竞争。mcache 中会维护上述 68×2 种 spanClass 的 mspan 数组，存放在 mcache 的 alloc 中，包括 scan 以及 noscan 两个队列。mcache 的主要结构如下，其中 tiny 相关的三个 field 涉及到 tiny 对象的分配，我们稍后在对象分配机制中再进行介绍。

复制代码

```
type mcache struct {
    ...
    tiny       uintptr
    tinyoffset uintptr
    tinyAllocs uintptr

    alloc [numSpanClasses]*mspan // spans to allocate from, indexed by spanCla
    ...
}
```



mcache结构

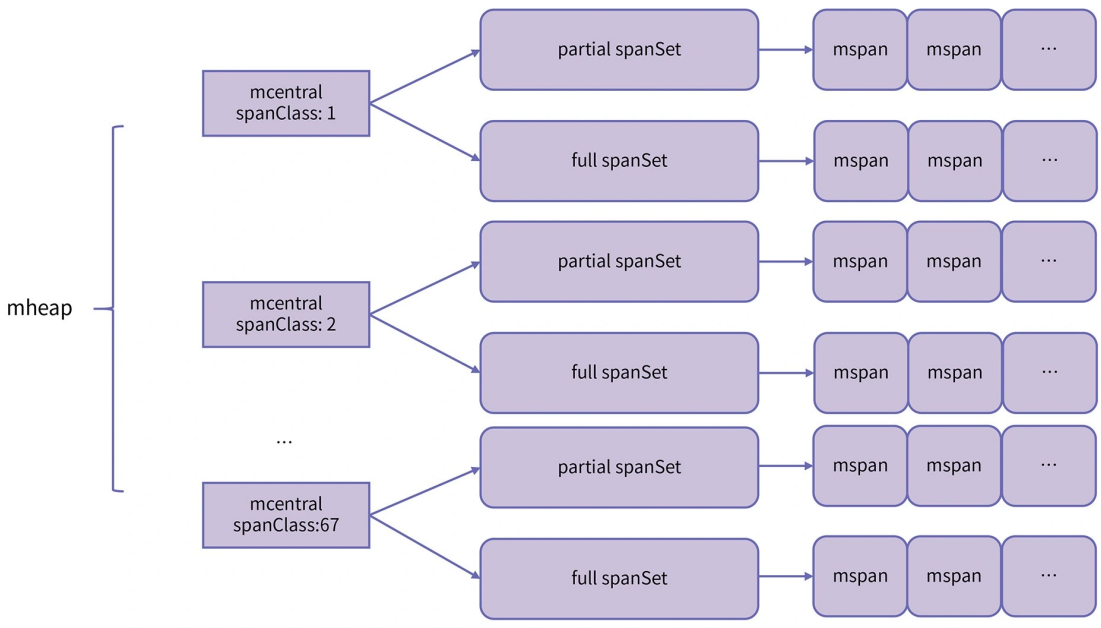
当 mcache 中的内存不够需要扩容时，需要向 mcentral 请求，mcentral 对应于 TCMalloc 中的 Central cache 结构。mcentral 的主要结构如下：

复制代码

```
type mcentral struct {
    spanclass spanClass
    partial [2]spanSet // list of spans with a free object
    full    [2]spanSet // list of spans with no free objects
}
```

可以看到，mcentral 中也存有 spanClass 的 ID 标识符，这表示说每个 mcentral 维护着固定一种 spanClass 的 mspan。spanClass 下面是两个 spanSet，它们是 mcentral 维护的 mspan 集合。partial 里存放的是包含着空闲空间的 mspan 集合，full 里存放的是不包含空闲空间的 span 集合。这里每种集合都存放两个元素，用来**区分集合中 mspan 是否被清理过**。

mcentral 不同于 mcache，每次请求 mcentral 中的 mspan 时，都可能发生不同线程直接的竞争。因此，在使用 mcentral 时需要进行加锁访问，具体来讲，就是 spanSet 的结构中会有一个 mutex 的锁的字段。



mcentral与mheap

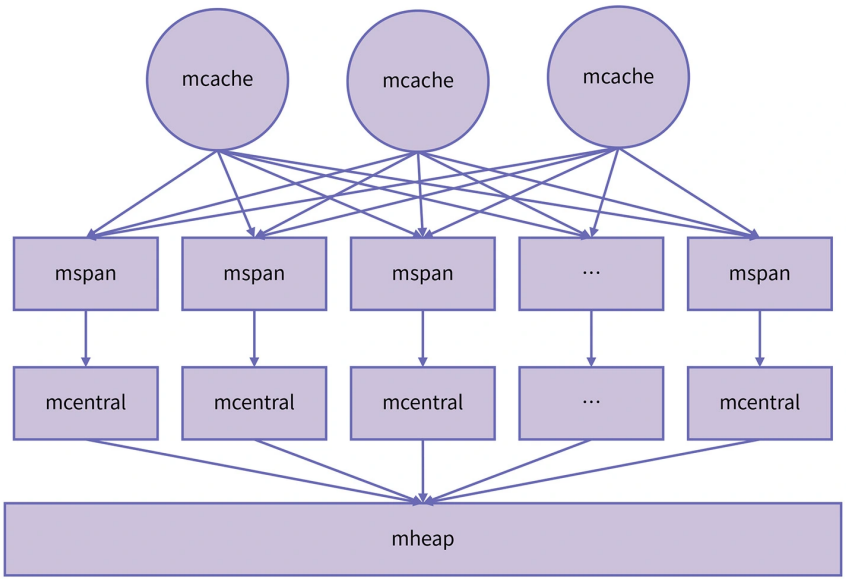
我们最后看下 mheap 的结构。mheap 在 Go 的运行时里边是只有一个实例的全局变量。上面我们讲到，维护 Go 的整个虚拟内存布局的 heapArena 的二维数组，就存放在 mheap 中。mheap 结构对应于 TCMalloc 中的 Page heap 结构。mheap 的主要结构如下：

复制代码

```
type mheap struct {
    lock  mutex

    arenas [1 << arenaL1Bits]*[1 << arenaL2Bits]*heapArena
    central [numSpanClasses]struct {
        mcentral mcentral
        pad      [cpu.CacheLinePadSize - unsafe.Sizeof(mcentral{})%cpu.CacheLi
    }
}

var mheap_ mheap
```

mheap 中存放了 68×2 个不同 spanClass 的 mcentral 数组，分别区分了 scan 队列以及 noscan 队列。

到目前为止，我们就对 Go 的三级内存管理器有了一个整体的认识，下面这张图展示了这几个结构的关系。



Go三级内存管理器

极客时间

学习了 Go 的三级内存管理机制，那么 Go 的对象分配逻辑也就很清晰了。

对象分配机制

我们在这里需要注意的一点是：Go 根据对象大小分成了三个级别，分别是微小对象、小对象和大对象。**微小对象**，也就是指大小在 0 ~ 16 字节的非指针类型对象；**小对象**，指的是大小在 16 ~ 32KB 的对象以及小于 16 字节的指针对象；**大对象**，也就是上文提到的 spanClass 为 0 的类型。

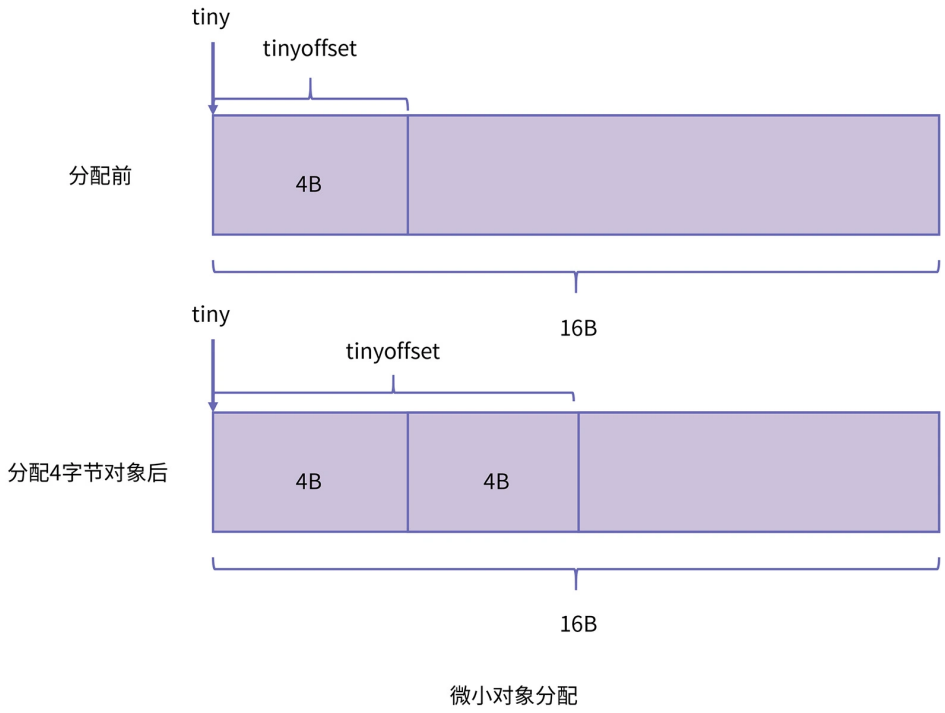
对于这三种类型对象，Go 的分配策略也不相同，对象分配的主要逻辑如下：

复制代码

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

微小对象会被放到 spanClass 为 2 的 mspan 中，由于每个对象最大是 16 字节，在分配时会尽量将多个小对象放到同一个内存块内（16 字节），这样可以更进一步的降低这种微小对象带来的碎片化。

我们在 mcache 中提到的三个字段：tiny、tinyoffset 和 tinyAllocs 就是用来维护微小对象分配器的。tiny 是指向当前在使用的 16 字节内存块的地址，tinyoffset 则是指新分配微小对象需要的起始偏移，tinyAllocs 则存放了目前这个 mcache 中共存放了多少微小对象。



微小对象分配

对于小对象的分配，整体逻辑跟 TCMalloc 是保持一致的，就是依次向三级内存管理器请求内存，一旦内存请求成功则返回，这里我就不详细展开了。

对于大对象的分配，Go 并不会走上述的三次内存管理器，而是直接通过调用 mcache.allocLarge 来分配大内存。allocLarge 会以内存页的倍数大小来通过 mheap_.alloc 申请对应大小内存，并构建起 spanClass 为 0 的 mspan 对象返回。

好，接下来我们具体看下 Go 的内存回收器是怎么实现的。

内存回收机制

前面我们讲到，Go 的 GC 算法采用的是经典的 CMS 算法，在并发标记的时候使用的是三色标记清除算法。而在 write barrier 上则采用了 **Dijkstra 的插入 barrier**，以及汤浅太一提出的删除 **barrier 混合的 barrier 算法**。

另外，为了降低 GC 在回收时 STW 的最长时间，Go 在内存回收时并不是一次性回收完全部的垃圾，而是采用增量回收的策略，将整个垃圾回收的过程切分成多个小的回收 cycle。具体来讲，Go 的内存回收主要分为这几个阶段：

1. **清除终止阶段；**

这个阶段会进行 STW，让所有的线程都进入安全点。如果此次 GC 是被强制触发的话，那么这个阶段还需要清除上次 GC 尚未清除的 mspan。

2. **标记阶段；**

在进行标记阶段之前，需要先做一些准备工作，也就是将 gcphase 状态从 _GCoff 切换到 _GCmark，并打开 write barrier 和 mutator assists，然后将根对象压入扫描队列中。此时，所有的 mutator 还处于 STW 状态。

准备就绪后重启 mutator 的运行，此时后台中的标记线程和 mutator assists 可以共同帮助 GC 进行标记。在标记过程中，write barrier 会把所有被覆盖的指针以及新指针都标记为灰色，而新分配的对象指针则直接标记为黑色。

标记线程开始进行根对象扫描，包括所有的栈对象、全局变量的对象，还有所有堆外的运行时数据结构。这里你要注意的是，当对栈进行扫描的时候需要暂停当前的线程。扫描完根对象后，标记线程会继续扫描灰色队列中的对象，将对象标记为黑色并依次将其引用的对象标灰入队。

由于 Go 中每个线程都有一个 Thread Local 的 cache，GC 采用的是分布式终止算法来检查各个 mcache 中是否标记完成。

3. **标记终止阶段；**

在标记终止阶段会进行 STW，暂停所有的线程。STW 之后将 gcphase 的状态切换到 _GCmarktermination，并关闭标记进程和 mutator assists。此外还会进行一些清理工作，刷新 mcache。

4. **清除阶段。**

在开始清除阶段之前，GC 会先将 gcphase 状态切换到 _GCoff，并关闭 write barrier。接着再恢复用户线程执行，此时新分配的对象只会是白色状态。并且清除工作是增量是进行的，所以在分配时，也会在必要的情况下先进行内存的清理。这个阶段的内存清理工作会在后台并发完成。

总结

这节课做为自动内存管理的最后一节，同样也是我们专栏的最后一篇文章，我们通过两个具体的例子来展示了，实际场景中语言虚拟机是如何进行内存管理的。

首先，我们介绍了动态语言和静态语言的区别，并在动态语言中选择 Python 做为实例，在静态语言中选择 Go 做为实例进行介绍。

它们的数据结构和算法设计其实并没有超出我们之前课程所介绍的理论。只是在代码实现上，在细节上做了一些调整。

我们讲解了动态类型语言的内存布局，接着又分析了 Python 中的内存回收的具体实现。Python 主要使用引用计数法进行内存管理。为了解决循环引用的问题，又引入了一种特殊的标记算法来释放垃圾对象。

在 Go 语言的内存管理机制中，我们详细学习了 Go 中内存分配的实现。Go 语言通过采用 TCMalloc 的分配器思路，以及对内存对象类别大小的精确控制，保障了程序在运行过程中能够有高速的分配效率，以及维持较低的碎片率，这样回收器就可以采用相对简单的 CMS 算法。

到这里，我们就把自动内存管理的核心内容全部介绍完了。

思考题

如果你对 Go 语言比较熟悉的话，可以思考一下，Go 语言中保留了值类型，有哪些优点和缺点呢？欢迎在留言区分享你的想法，我在留言区等你。

吊打面试官

- 请你谈一下对动态语言和静态语言的理解。

动态语言的特点是可以在运行时修改对象属性，甚至是对类进行修改，这就导致很难在编译期间就确定一个对象的大小。编译器也无法确定属性在对象中的位置，从而不能使用相对对象头寻址的办法快速访问对象属性。而静态语言刚好相反，它不允许在运行时修改对象属性，相同类型的对象，它们的大小是确定的。所以编译器可以推算属性在对象中的偏移值，从而可以使用简单的寻址指令访问对象属性。

动态语言的优点是表达力强，方便灵活，适合做为脚本语言进行开发，但是代码多了则不易维护。而静态语言则往往是强类型语言，IDE支持比较完善，性能好，适合开发较复杂的软件，维护更方便。

高频面试真题

极客时间

好啦，这节课到这就结束啦。欢迎你把这节课分享给更多对计算机内存感兴趣的朋友。我是海纳，我们下节课再见！

上一篇 23 | Pauseless GC：挑战无暂停的垃圾回收

下一篇 不定期福利第一期 | 海纳：我是如何学习计算机知识的？

精选留言 (2)

写留言

那时刻
2021-12-23

请问老师，有了值类型的介入，编译器只需要关注函数内部的逃逸分析（intraprocedural escape analysis），而不用关注函数间的逃逸分析。 为什么有值类型介入后，就不需要关注函数间逃逸分析了呢？

费城的二鹏
2021-12-22

最后一集课程啦，感谢老师的细致讲解！
思考题，我猜一下值类型的优点吧，值类型都在栈上分配，随用随释放，有利于减少gc压力。

作者回复: Good~希望有机会再见~