《源码探秘 CPython》92. Python 是如何管理内存的？（上）

原创 古明地觉 古明地觉的编程教室 2022-05-17 08:30 发表于北京

楔子

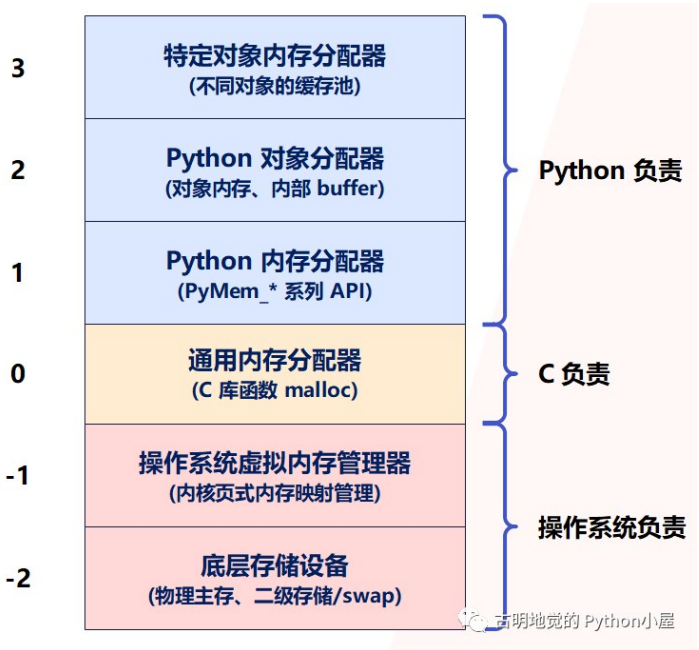
内存管理，对于Python这样的动态语言来说是非常重要的一部分，它在很大程度上决定了Python的执行效率。因为Python在运行的过程中会创建和销毁大量的对象，这些都涉及内存的管理，因此精湛的内存管理技术是确保内存使用效率的关键。

内存管理架构

首先 Python 的内存管理机制是分层次的，我们可以看成是有 6 层：-2, -1, 0, 1, 2, 3。

- 最底层，也就是 -2 和 -1 层，这是由操作系统提供的内存管理接口，因为所有和硬件相关的资源肯定是由操作系统负责管理的，应用程序通过系统调用向操作系统申请内存。注意：这一层 Python 是无权干预的；
- 第 0 层，C 的库函数会将系统调用封装成通用的内存分配器，也就是我们所熟悉的 malloc 系列函数。注意：这一层 Python 同样无法干预；
- 第1、2、3层，由Python解释器实现并负责维护；

所以 Python 的内存管理实际上是封装了 C 的 malloc，C 的 malloc 则是封装了系统调用。



我们自下而上来简单说一下，首先操作系统内部是一个基于页表的虚拟内存管理器(第 -1 层)，以页(page)为单位管理内存，而CPU内存管理单元(MMU)在这个过程中发挥重要作用。虚拟内存管理器下方则是底层存储设备(第 -2 层)，直接管理物理内存以及磁盘等二级存储设备。

> 所以最后的两层是操作系统的领域，过于底层，不在我们的涉及范围内，简单了解就好。有兴趣的话，可以网上查阅相关资料，看看操作系统是如何管理内存的。

C 库函数实现的通用内存分配器位于内存管理层次中的第 0 层，它封装了操作系统内存分配相关的系统调用。此层之上是应用程序自己的内存管理，此层之下则是操作系统的内存管理。

因此第 1、2、3 层是 Python 自己的内存管理，总共分为 3 层，我们来解释一下。

第 1 层：基于第 0 层的"通用内存分配器"包装而成。

这一层并没有在第 0 层上加入太多的动作，其目的仅仅是为 Python 提供一层统一的 raw memory 管理接口。这么做的原因就是虽然不同的操作系统都提供了 ANSI C 标准所定义的内存管理接口，但是对于某些特殊情况不同的操作系统有着不同的行为。

比如调用 malloc(0)，有的操作系统会返回 NULL，表示申请失败；但是有的操作系统则会返回一个貌似正常的指针，但是这个指针指向的内存并不是有效的。为了最广泛的可移植性，Python 必须保证相同的语义一定代表着相同的运行时行为，为了处理这些与平台相关的内存分配行为，Python 必须要在 C 的内存分配接口之上再提供一层包装。

在Python中，第一层的实现就是一组以 PyMem_* 为前缀的函数簇，下面来看一下。

```
//Include/pymem.h
PyAPI_FUNC(void *) PyMem_Malloc(size_t size);
PyAPI_FUNC(void *) PyMem_Realloc(void *ptr, size_t new_size);
PyAPI_FUNC(void) PyMem_Free(void *ptr);


//Objects/obmalloc.c
void *
PyMem_Malloc(size_t size)
{
    /* see PyMem_RawMalloc() */
    if (size > (size_t)PY_SSIZE_T_MAX)
        return NULL;
    return _PyMem.malloc(_PyMem.ctx, size);
}

void *
PyMem_Realloc(void *ptr, size_t new_size)
{
    /* see PyMem_RawMalloc() */
    if (new_size > (size_t)PY_SSIZE_T_MAX)
        return NULL;
    return _PyMem.realloc(_PyMem.ctx, ptr, new_size);
}

void
PyMem_Free(void *ptr)
{
    _PyMem.free(_PyMem.ctx, ptr);
}
```

我们看到在第一层，Python 提供了类似于 C 中 malloc, realloc, free 的语义。比如 PyMem_Malloc 负责分配内存，首先会检测申请的内存大小，如果超过了 PY_SSIZE_T_MAX 则直接返回NULL，否则调用 _PyMem.malloc 申请内存。这个 _PyMem.malloc 和 C 的库函数 malloc 几乎没啥区别，但是会对特殊值进行一些处理。

到目前为止，仅仅是分配了 raw memory 而已。当然在第一层，Python 还提供了面向对象中类型的内存分配器。

```
//Include/pymem.h
#define PyMem_New(type, n) \
  ( ((size_t)(n) > PY_SSIZE_T_MAX / sizeof(type)) ? NULL :      \
        ( (type *) PyMem_Malloc((n) * sizeof(type)) ) )
#define PyMem_NEW(type, n) \
  ( ((size_t)(n) > PY_SSIZE_T_MAX / sizeof(type)) ? NULL :      \
        ( (type *) PyMem_MALLOC((n) * sizeof(type)) ) )
#define PyMem_Resize(p, type, n) \
  ( (p) = ((size_t)(n) > PY_SSIZE_T_MAX / sizeof(type)) ? NULL :        \
        (type *) PyMem_Realloc((p), (n) * sizeof(type)) )
#define PyMem_RESIZE(p, type, n) \
  ( (p) = ((size_t)(n) > PY_SSIZE_T_MAX / sizeof(type)) ? NULL :        \
        (type *) PyMem_REALLOC((p), (n) * sizeof(type)) )
#define PyMem_Del               PyMem_Free
#define PyMem_DEL               PyMem_FREE
```

很明显，在 PyMem_Malloc 中需要程序员自行提供所申请的空间大小。然而在 PyMem_New 中，只需要提供类型和数量，Python 会自动侦测其所需的内存空间大小。

第 2 层：在第 1 层提供的通用 PyMem_* 接口的基础上，实现统一的对象内存分配(object.tp_alloc)。

第 1 层所提供的内存管理接口的功能是非常有限的，如果创建一个 PyLongObject 对象，还需要做很多额外的工作，比如设置对象的类型参数、初始化对象的引用计数值等等。

因此为了简化 Python 自身的开发，Python 在比第 1 层更高的抽象层次上提供了第 2 层内存管理接口。在这一层，是一组以 PyObject_* 为前缀的函数簇，主要提供了创建 Python 对象的接口。这一套函数簇又被称为 Pymalloc 机制，因此在第 2 层的内存管理机制上，Python 对于一些内置对象构建了更高抽象层次的内存管理策略。

第 3 层：为特定对象服务。

这一层主要是用于对象的缓存机制，比如：小整数对象池，浮点数缓存池等等。

所以 Python 的 GC 是隐藏在哪一层呢？不用想，肯定是第二层，也是在 Python 的内存管理中发挥巨大作用的一层，我们后面也会基于第二层进行剖析。

为什么要有内存池

在 Python 中，很多时候申请的内存都是小块的内存，这些小块的内存在申请后很快又被释放，并且这些内存的申请并不是为了创建对象，所以并没有对象一级的缓存机制。这就意味着 Python 在运行期间需要大量地执行底层的 malloc 和 free 操作，导致操作系统在用户态和内核态之间进行切换，这将严重影响Python的效率。

所以为了提高执行效率，Python 引入了内存池机制，用于管理对小块内存的申请和释放，这就是之前说的Pymalloc机制，并且提供了pymalloc_alloc, pymalloc_realloc, pymalloc_free 三个接口。

可以认为 Python 会向操作系统预申请一部分内存，专门用于那些占用内存小的对象，这就是所谓的内存池机制。

需要注意这里的内存池和前面介绍对象时提到的缓存池不同，缓存池可以理解为是数组或者链表，目的是在对象不用的时候缓存起来，需要的时候再拿来用。也就是说，对象自始至终对象都存在于内存当中。

而内存池是用来申请和释放内存的，一申请，对象就横空出世了；一释放，对象就会归于湮灭。

而整个小块内存的内存池可以视为一个层次结构，从下至上分别是：block、pool、arena。当然内存池只是一个概念上的东西，表示 Python 对整个小块内存分配和释放行为的内存管理机制。

block

在最底层，block 是一个确定大小的内存块。并且 block 有很多种，不同种类的 block 拥有不同的内存大小。为了在当前主流的 32 位平台和 64 位平台都能获得最佳性能，所有 block 的长度都是 8 字节对齐的。

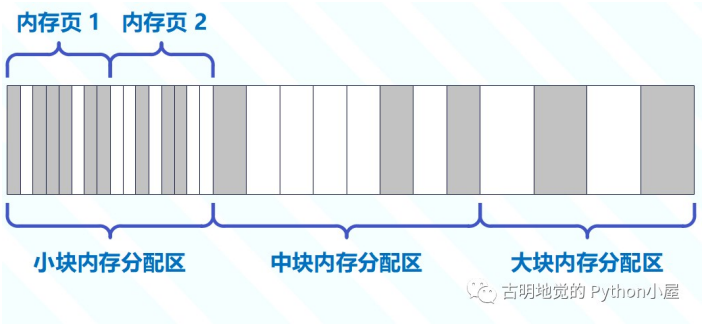
```
1 //Objects/obmalloc.c
2 #define ALIGNMENT               8     /* must be 2^N */
```

但是问题来了，为什么要有这么多种类的 block 呢？为了更好地理解这一点，我们需要了解内存碎片化这个概念。内存碎片化是困扰经典内存分配器的一大难题，碎片化导致的结果也是惨重的。看一个典型的内存碎片化的例子：

虽然还有 1350K 的可用内存，但由于分散在一系列不连续的碎片上，因此连 675K、总可用内存的一半都分配不出来。那么如何避免内存碎片化呢？想要解决问题，就必须先分析导致问题的根源。

我们知道，应用程序请求的内存尺寸是不确定的，有大有小；释放内存的时机也是不确定的，有先有后。经典内存分配器将不同尺寸的内存混合管理，按照先来后到的顺序分配：

由此可见，将不同尺寸的内存块混合管理，将大块内存切分后再次分配的做法是罪魁祸首。找到了问题的原因，那么解决方案也就自然而然浮出水面了，那就是将内存空间划分成不同区域，独立管理，比如：



如图，内存被划分成小、中、大三个不同尺寸的区域，区域可由若干内存页组成，每个页都划分为统一规格的内存块。这样一来，小块内存的分配，不会影响大块内存区域，使其碎片化。

不过每个区域的碎片仍无法完全避免，但这些碎片都是可以被重新分配出去的，影响不大。此外，通过优化分配策略，碎片还可被进一步合并。以小块内存为例，新内存优先从内存页 1 分配，内存页 2 将慢慢变空，最终将被整体回收。

多提一句，内存划分为不同尺寸这一策略的背后体现的是一种分而治之的思想，这种思想应用的非常广泛，比如微服务里面的多集群。假设你是一家视频网站的开发工程师，用户上传的视频必须要经过转码，但如果某个恶意用户上传了一个时间非常长、文件非常大的视频怎么办？这个时候其它正常用户上传的视频都会卡住。那么此时就可以考虑采用多集群，不同时长的视频进入不同的集群进行转码，问题不就解决了吗？

扯的有点远了，回到正题。在虚拟机内部，每时每刻都有对象创建、销毁，这引发频繁的内存申请、释放动作。这类内存尺寸一般不大，但分配、释放频率非常高，因此 Python 专门设计内存池对此进行优化。

那么，尺寸多大的内存才会使用内存池呢？Python 以 512 字节为上限，小于等于 512 的内存分配才会被内存池接管。所以当申请的内存大小不超过这个上限时， Python 可以使用不同种类的 block 满足对内存的需求。

当申请的内存大小超过了上限， Python 就会将对内存的请求转交给第一层的内存管理机制，即 PyMem 函数簇来处理（进而调用 malloc）。所以这个内存池可申请的大小是有上限的，只有大小不超过 512 字节的对象才会使用内存池，如果超过了这个值还是要经过操作系统临时申请的。

- 1 ~ 512: 由专门的内存池负责分配，内存池以内存尺寸进行划分；
- 0 或 512以上: 直接调动 malloc 函数；

```
1 //Objects/obmalloc.c
2 #define SMALL_REQUEST_THRESHOLD 512
3 #define NB_SMALL_SIZE_CLASSES   (SMALL_REQUEST_THRESHOLD / ALIGNMENT)
```

那么问题来了，Python 是否为每个尺寸的内存都准备一个独立的 block 呢？答案是否定的，原因有几个：

- 内存规格有 512 种之多，如果 block 也分 512 种，徒增复杂性；
- block 种类越多，额外开销越大；
- 如果某个尺寸的内存只申请一次，将浪费内存页内的其他空闲内存；

相反，Python 以 8 字节为梯度，将内存块分为：8 字节、16 字节、24 字节，以此类推，总共 64 种 block：

```
* For small requests we have the following table:
*
* Request in bytes     Size of allocated block      Size class idx
* ----------------------------------------------------------------
*        1-8                     8                       0
*        9-16                   16                       1
*       17-24                   24                       2
*       25-32                   32                       3
*       33-40                   40                       4
*       41-48                   48                       5
*       49-56                   56                       6
*       57-64                   64                       7
*       65-72                   72                       8
*         ...                   ...                    ...
*      497-504                  504                      62
*      505-512                  512                      63
*
*      0, SMALL_REQUEST_THRESHOLD + 1 and up: routed to the underlying
*      allocator.
*/
```


- 里面的 Size class idx 指的就是 block 的种类，总共 64 种，分别用 0 ~ 63 表示；
- 里面的 Size of allocated block 指定的就是 block 对应的大小；

当然 Python 也提供了一个宏，来描述这两者的关系，事实上我们也能看出来。

```
//Objects/obmalloc.c
#define ALIGNMENT_SHIFT         3
#define INDEX2SIZE(I) (((uint)(I) + 1) << ALIGNMENT_SHIFT)
```

索引为 0 的话，就是 1 << 3，显然结果为 8；索引为 1 的话，就是 2 << 3，显然结果为16，以此类推。因此当我们申请一个 44 字节的内存时， PyObject_Malloc 会从内存池中划分一个 48 字节的 block 给我们。

所以这样就解决了内存碎片的问题，但是又暴露了一个新的问题，首先内存池是由多个内存页组成，每个内存页又划分为多个具有相同规格的内存块(block)，这些后面会说。假设我们申请 7 字节的内存，那么毫无疑问会给我们一个 8 字节的块；但是当我们申请 1 字节的时候，分配给我们的还是 8 字节的块。因为每次分配的时候，给的一定是一个新的块，比如第一次申请 1 字节，会分配一个 8 字节的块；第二次申请 1 字节，还是会分配一个 8 字节的块。

这种做法好处显而易见，前面提到的碎片化问题得到解决。此外这种方式是字对齐的，内存以字对齐的方式可以提高读写速度。字大小从早期硬件的 2 字节、4 字节，慢慢发展到现在的 8 字节，甚至 16 字节。

当然了，有得必有失，内存利用率成了被牺牲的因素，以 8 字节内存块为例，平均利用率为 (1+8)/2/8*100% ，大约只有 56.25% 。当然啦，只有那些规模较小的内存块的利用率会不高，而对于那些规模大的内存块，利用率还是很高的。如果将所有规模的内存块都考虑在一起，那么整体的内存利用率能达到大概 98%。虽然规模较小的内存块会存在浪费，但是对于现在的机器而言，完全是可以容忍的。

另外在 Python 底层，block 其实只是一个概念，源码中没有与之对应的实体存在。之前我们说的对象，对象在源码中有对应的 PyObject，但是这里的 block 仅仅是概念上的东西，我们知道它是具有一定大小的内存，但是它并不与 Python 源码里面的某个结构相对应。不过 Python 提供了一个管理 block 的东西，也就是要分析的 pool，或者理解为上面说的内存页。

一组 block 的集合称为一个 pool，换句话说，一个 pool 管理着一堆具有固定大小的内存块(block)。而一个 pool 的大小通常为一个系统内存页，也就是 4kb。

```
//Objects/obmalloc.c
#define SYSTEM_PAGE_SIZE        (4 * 1024)
#define SYSTEM_PAGE_SIZE_MASK   (SYSTEM_PAGE_SIZE - 1)
#define POOL_SIZE               SYSTEM_PAGE_SIZE
#define POOL_SIZE_MASK          SYSTEM_PAGE_SIZE_MASK
```

SYSTEM_PAGE_SIZE 指的就是内存页大小，至于它下面的 MASK 不用想肯定是负责将取模运算优化成按位与运算。然后 POOL_SIZE 和内存页大小相等，因此可以把 pool 理解成内存页。

虽然 Python 没有为 block 提供对应的结构，但是提供了和 pool 相关的结构，因为 Python 是将内存页看成由一个个内存块(block)组成的池子(pool)，我们来看看 pool 的结构：

```
//Objects/obmalloc.c
struct pool_header {
    //当前pool里面已经分配出去的block的数量
    union { block *_padding;
            uint count; } ref;
    //指向第一块可用的 block
    block *freeblock;
    //底层会有多个pool
    //多个pool之间也会形成一个链表
    //所以nextpool指向下一个pool
    struct pool_header *nextpool;
    //prevpool指向上一个pool
    struct pool_header *prevpool;
    //在arena里面的索引(arena后面会说)
    uint arenaindex;
    //每个 pool 都维护了一组相同规格的 block
    //而 szindex 指的就是 block 的 Size class idx
    //如果是2，那么管理的每个 block 的大小就是 24
    uint szidx;
    //下一个可用 block 的内存偏移量
    uint nextoffset;
    //最后一个 block 的内存偏移量
    uint maxnextoffset;
};

typedef struct pool_header *poolp;
```

一个 pool 的大小是 4KB，但是从当前的这个 pool 的结构体来看，用鼻子想也知道吃不完 4KB 的

内存，事实上这个结构体只占48字节。

所以呀，这个结构体叫做 struct pool_header，它仅仅是一个 pool 的头部，除去这个 pool_header，剩下的就是维护的所有 block 所占的内存。

我们注意到，pool_header 里面有一个 szidx，这就意味着 pool 里面管理的内存块大小都是一样的。也就是说，一个 pool 管理的 block 可以是 32 字节、也可以是 64 字节，但是不会出现既有 32 字节的 block、又有 64 字节的 block。

每一个 pool 都和一个 size 联系在一起，更确切的说是都和一个 Size class index 联系在一起，表示 pool 里面存储的 block 都是多少字节的。这就是里面的 szidx 字段存在的意义。

我们以 16 字节(szidx=1) 的 block 为例，看看 Python 是如何将一块 4KB 的内存改造成管理 16 字节 block 的 pool：

```
// /Objects/obmalloc.c
typedef uint8_t block;


static void*
pymalloc_alloc(void *ctx, size_t nbytes)
{
    block *bp;
    poolp pool;
    poolp next;
    uint size;
    //......
    //......
    init_pool: //pool指向了一块4KB的内存
        next = usedpools[size + size]; /* == prev */
        pool->nextpool = next;
        pool->prevpool = next;
        next->nextpool = pool;
        next->prevpool = pool;
        pool->ref.count = 1;
        //......
        //设置pool的size class index
        pool->szidx = size;
        //一个宏，将szidx转成内存块的大小
        //比如：0->8，1->16，63->512
        size = INDEX2SIZE(size);
        //跳过用于pool_header的内存，并进行对齐
        //POOL_OVERHEAD就是pool结构体的大小，48字节
        //此时的bp指向第一块 block
        bp = (block *)pool + POOL_OVERHEAD;
        //等价于pool->nextoffset = POOL_OVERHEAD+size*2
        pool->nextoffset = POOL_OVERHEAD + (size << 1);
        pool->maxnextoffset = POOL_SIZE - size;
        pool->freeblock = bp + size;
        *(block **)(pool->freeblock) = NULL;
        goto success;
    }
   //.....
success:
    assert(bp != NULL);
    return (void *)bp;

failed:
    return NULL;
}
```

我们注意一下里面的 freeblock，它等于 bp+size，首先 bp 指向的是 pool 中的第一块block，而 freeblock 指向的是第一块可用的 block。由于新内存页总是由内存请求触发，所以第一个 block 一定会被分配出去（已不可用），第二个 block 才是可用的。

从 ref.count 也可以看出端倪，我们说 ref.count 记录了当前已经被分配的 block 的数量，但初始化的时候不是 0，而是 1，也证明了这一点。而 freeblock 指向第一个可用的 block，所以它在源码中被设置成了 bp + size，也就是第二个 block。

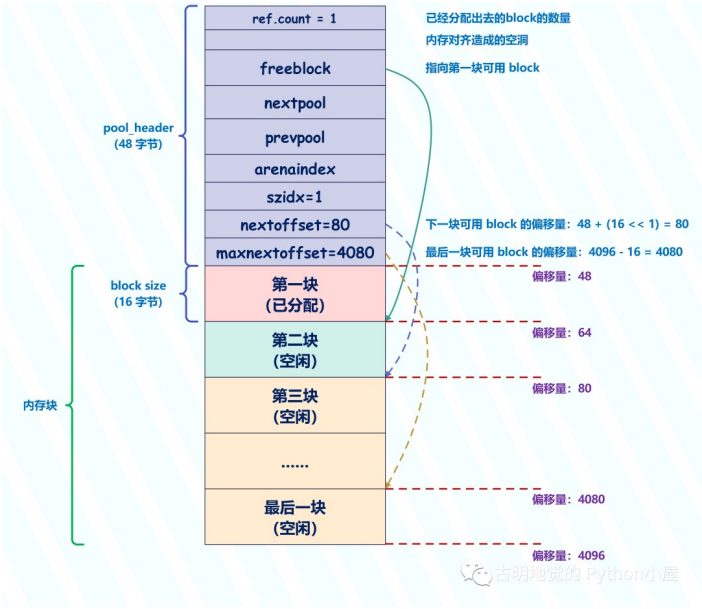
然后是 nextoffset 字段，它表示下一个可用 block 的内存偏移量，需要注意这里的下一个可用。对于新创建的内存页来说，第一个可用的 block 就是第二个 block，那么下一个可用的 block 指的就是第三个 block。所以它应该等于 POOL_OVERHEAD 加上两个 block 的大小，因此在源码中被设置成了 48 + (size << 1)。

最后是 maxnextoffset，它表示最后一个 block 的内存偏移量，显然它等于 POOL_SIZE 减去一个内存块的大小。

因此通过 freeblock 能拿到第一个可用 block，至于剩余的可用 block 由于还没有人用，暂时先不管，只是通过 nextoffset 和 maxnextoffset 记录了第二个可用 block 和最后一个可用 block 的偏移位置。

> 所以 nextoffset 和 maxnextoffset 里面存储的是相对于 pool 头部的偏移位置

最终改造成 pool 之后的 4kb 内存如图所示：

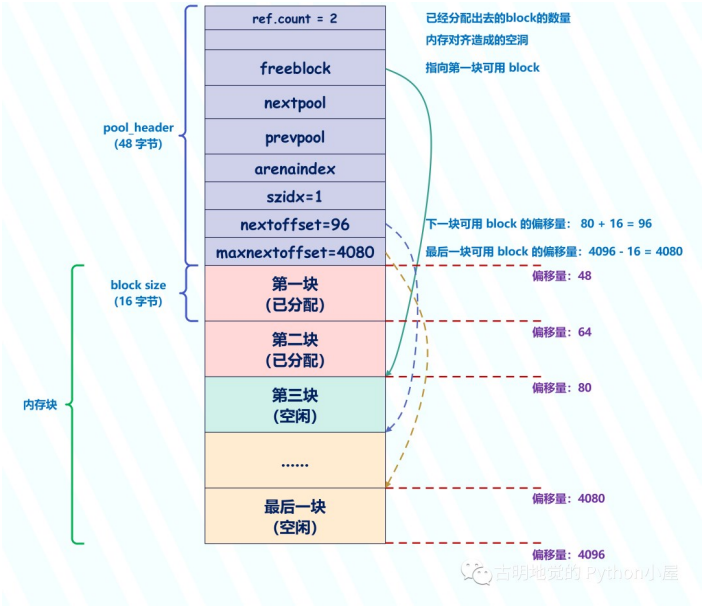


里面的实线箭头是指针，虚线箭头则是偏移位置的形象表示。

在了解初始化之后的 pool 的样子之后，可以来看看 Python 在申请 block 时，pool_header 中的各个字段是怎么变动的。假设我们再申请 1 块 16 字节的内存块：

```
//Objects/obmalloc.c
static void*
pymalloc_alloc(void *ctx, size_t nbytes)
{
    //......
    if (pool != pool->nextpool) {
        //首先pool中已分配的block数自增1
        ++pool->ref.count;
        //freeblock指向的是第一块可用的 block
        bp = pool->freeblock;
        assert(bp != NULL);
        if ((pool->freeblock = *(block **)bp) != NULL) {
            goto success;
        }

        //由于freeblock本身就指向可用的block
        //因此当再次申请16字节的block时，返回freeblock即可
        //那么很显然，freeblock还需要前进，指向下一块可用的block
        //分配之前，第一块可用block是第二块block，被freeblock指向
        //分配之后，第二块block就不可用了
        //所以freeblock要指向第三块block（作为新的第一块可用block）
        //那么怎么才能获取下一块可用的block呢？
        if (pool->nextoffset <= pool->maxnextoffset) {
          //显然要依赖于nextoffset
          //它保存的正是下一块可用block的偏移量（分配前的第三块）
          //由于这个偏移量是相对pool而言的
          //所以pool+nextoffset就是下一块可用block的偏移量了
          //将它赋值给freeblock即可
            pool->freeblock = (block*)pool +
                              pool->nextoffset;
          //同理，nextoffset也要向前移动一个block的距离
          //因为分配之后，第三块block成为了第一块可用block
          //那么下一个可用block就应该是第四块block
            pool->nextoffset += INDEX2SIZE(size);
          //依次反复，即可对所有的block进行遍历
          //而maxnextoffset指明了该pool中最后一个可用block的偏移量
          //当pool->nextoffset > pool->maxnextoffset
          //也就是上面的if条件不满足时，就说明遍历完pool中所有的block了
          //再次获取显然就是NULL了
            *(block **)(pool->freeblock) = NULL;
            goto success;
        }

        /* Pool is full, unlink from used pools. */
        next = pool->nextpool;
        pool = pool->prevpool;
        next->prevpool = pool;
        pool->nextpool = next;
        goto success;
    }
    //......
}
```

所以当我们再申请 1 块 16 字节的内存块时，pool 的结构图就变成了这样。



首先freeblock指向了第三块block，仍然是第一块可用block；nextoffset 表示下一块可用block的偏移量，显然下一块的可用block在分配之后就变成了第四块，因此 48 + 16 * 3 = 96；至于maxnextoffset 仍然是4080，它是不变的。

随着内存分配的请求不断发起，空闲、或者说可用的内存块（block）也将不断地分配出去，freeblock 不断前进、指向下一个可用内存块作为新的第一个可用内存块；nextoffset 也在不断前进、偏移量每次增加内存块的大小，也就是保存新的下一个可用内存块的偏移量，直到所有的空闲内存块被消耗完。

所以，申请、前进、申请、前进，一直重复着相同的动作，整个过程非常自然，也很容易理解。但是我们知道一个 pool 里面的 block 都是相同大小的，这就使得一个 pool 只能满足 (POOL_SIZE - 48) / size 次对 block 的申请，这样存在一个问题。

我们知道内存块不可能一直被使用，肯定有释放的那一天。假设我们分配了两个内存块，理论上讲下一次应该申请第三个内存块，但是某一时刻第一个内存块被释放了，那么下一次申请的时候，Python 是申请第一个内存块、还是第三个内存块呢？

显然为了 pool 的使用效率，最好分配第一个 block。因此可以想象，一旦 Python 运转起来，内存的释放动作将导致 pool 中出现大量的离散可用 block。而 Python 为了知道哪些 block 是被使用之后再次被释放（离散可用）的，必须建立一种机制，将这些离散可用的 block 组合起来，再次使用。这个机制就是 block 链表，这个链表的关键就在 pool_header 里面的那个 freeblock 身上。

再来回顾一下pool_header的定义：

```
//Objects/obmalloc.c
struct pool_header {
    union { block *_padding;
            uint count; } ref;
    //指向第一块可用的 block
    //或者说指向可用block链表中的第一块block
    block *freeblock;
    struct pool_header *nextpool;
    struct pool_header *prevpool;
    uint arenaindex;
    uint szidx;
    uint nextoffset;
    uint maxnextoffset;
};
```

当 pool 初始化完后之后，freeblock 指向了一个有效的地址，也就是可以分配出去的block的地址。然而奇特的是，当 Python 设置了freeblock时，还设置了 *freeblock。这个动作看似诡异，然而我们马上就能看到设置 *freeblock 的动作正是建立离散可用block链表的关键所在。

目前我们看到的 freeblock 只是在机械地前进，因为它在等待一个特殊的时刻。在这个特殊的时刻，你会发现 freeblock 开始成为一个苏醒的精灵，在这 4kb 的内存上灵活地舞动，而这个特殊的时刻就是一个 block 被释放的时刻。

```
//Objects/obmalloc.c

//基于地址P获得离P最近的pool的边界地址
#define POOL_ADDR(P) ((poolp)_Py_ALIGN_DOWN((P), POOL_SIZE))

static int
pymalloc_free(void *ctx, void *p)
{
    poolp pool;
```

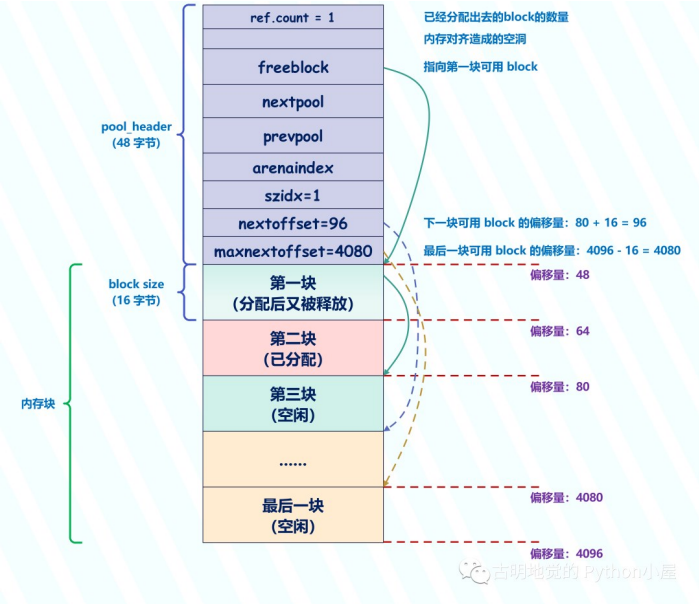
```
    block *lastfree;
    poolp next, prev;
    uint size;

    assert(p != NULL);

    pool = POOL_ADDR(p);
    //如果p不再pool里面，直接返回0
    if (!address_in_range(p, pool)) {
        return 0;
    }
    //释放，那么ref.count就势必大于0
    assert(pool->ref.count > 0);            /* else it was empty */
    *(block **)p = lastfree = pool->freeblock;
    pool->freeblock = (block *)p;
    //......
}
```

在释放 block 时，神秘的 freeblock 惊鸿一瞥，显然覆盖在 freeblock 身上的那层面纱就要被揭开了。我们知道，这时 freeblock 虽然指向了一个有效的 pool 里面的地址，但是 *freeblock 是为 NULL 的。

假设这时候 Python 释放的是 block 1，那么 block 1 中的第一个字节的值被设置成了当前 freeblock 的值，然后 freeblock 的值被更新了，指向了 block 1 的首地址。就是这两个步骤，一个 block 被插入到了离散可用的 block 链表中。

简单点，说人话就是：原来 freeblock 指向 block 3，而 block 1 在被释放之后就变成了 block 1 指向 block 3，而 freeblock 则指向了 block 1。block 链表里面的每一个块都有一个 next 指针，指向下一个可用的块。

所以在第一块 block 释放之后，pool 的结构图变化如下：

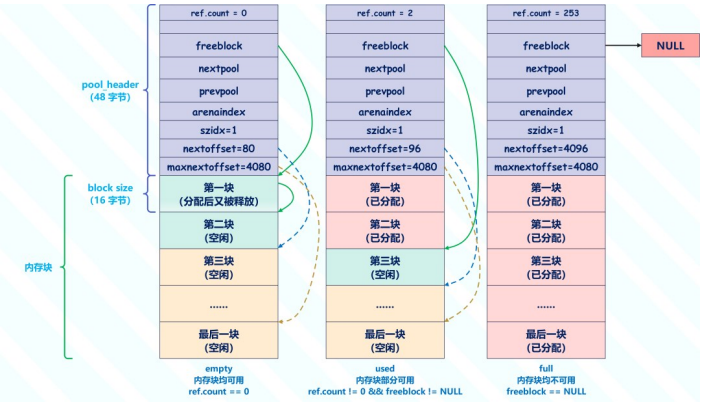


到了这里，这条实现方式非常奇特的 block 链表就被我们挖掘出来了，从 freeblock 开始，我们可以很容易地以 freeblock = *freeblock 的方式遍历这条链表。而当发现 *freeblock 为 NULL 时，则表明到达了该链表(可用的block链表)的尾部了，那么下次就需要申请新的 block 了。

```
//Objects/obmalloc.c
static void*
pymalloc_alloc(void *ctx, size_t nbytes)
{
    if (pool != pool->nextpool) {
        ++pool->ref.count;
        bp = pool->freeblock;
        assert(bp != NULL);
        //如果这里的条件不为真
        //表明离散可用block链表中已经不存在可用的block了
        if ((pool->freeblock = *(block **)bp) != NULL) {
            goto success;
        }
        //上面的代码之前省略了
        //可以看到，从pool里面申请新的block之前
        //会先检测离散可用block链表中是否存在可用的block
        if (pool->nextoffset <= pool->maxnextoffset) {
            pool->freeblock = (block*)pool +
                              pool->nextoffset;
            pool->nextoffset += INDEX2SIZE(size);
            *(block **)(pool->freeblock) = NULL;
            goto success;
        }

```

```
25          //......
26      }
27  }
```

逻辑还是很好理解的，因此我们可以得出，一个 pool 在其声明周期内，可以处于以下三种状态：



值得一提的是 empty 状态，此时内存块全部都是可用的，而可用有两种情况，第一种是从未被使用过，第二种是使用完之后又被释放了。而我们知道内存页一旦创建，第一个块肯定会分配出去。所以 empty 状态的内存页，至少有一个块是分配之后又被释放了的。当然不管怎么样，此时它们都是可用的。

然后再来考虑一个新的问题，我们在上面的代码中看到，如果离散可用的 block 链表中不存在可用的 block 时，会从 pool 中申请。而这是有条件的，必须满足：nextoffset <= maxnextoffset 才行。但如果连这个条件都不成立了呢？说明 pool 中已经没有可用的 block 了，因为 pool 是有大小限制的。

那么这个时候如果想再申请一个 block 要怎么做呢？答案很简单，再来一个 pool，然后从新的 pool 里面申请不就好了。

所以 block 组合起来可以成为一个 pool，那么同理多个 pool 也是可以组合起来的，而多个 pool 组合起来会得到什么呢。我们说内存池是分层次的，从下至上分别是：block、pool、arena，显然多个 pool 组合起来，就是我们下面下一篇文章要介绍的 arena。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》93. Python 是如何管理内存的？（下）

下一篇 ›
《源码探秘 CPython》91. 可执行文件的内存模型，变量的值是放在栈上还是放在堆上