《源码探秘 CPython》93. Python 是如何管理内存的？（下）

原创 古明地觉 古明地觉的编程教室 2022-05-18 08:30 发表于北京

在 Python 中，多个 pool 聚合的结果就是一个 arena。上一篇文章提到，pool 的大小默认是 4kb，同样每个 arena 的大小也有一个默认值。

```
#define ARENA_SIZE (256 << 10)
#define SYSTEM_PAGE_SIZE        (4 * 1024)
#define POOL_SIZE               SYSTEM_PAGE_SIZE
```

显然这个值默认是 256kb，也就是 64 个 pool 的大小。我们来看看arena的底层结构体定义，同样藏身于 Objects/obmalloc.c 中。

```
struct arena_object {
    //arena的地址
    uintptr_t address;

    //池对齐指针，指向下一个被划分的pool
    block* pool_address;

    //该arena中可用pool的数量
    uint nfreepools;

    //该arena中所有pool的数量
    uint ntotalpools;

    //我们在介绍pool的时候说过，pool之间也会形成一个链表
    //而这里freepools指的就是第一个可用的 pool
    struct pool_header* freepools;

    //从名字上也能看出这两个字段是干啥的
    //nextarena指向下一个arena，prevarena指向上一个arena
    //是不是说明arena之间也会组成链表呢？答案不是的
    //多个arena之间组成的是一个数组，至于为什么我们下面说
    struct arena_object* nextarena;
    struct arena_object* prevarena;
};
```

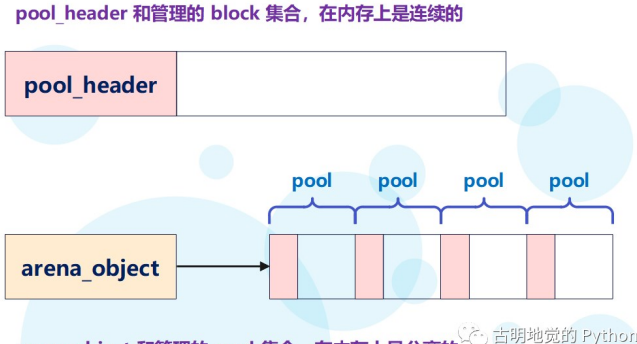
一个概念上的 arena 在源码中对应一个 arena_object 结构体实例，确切的说，arena_object 仅仅是 arena 的一部分，就像 pool_header 仅仅是 pool 的一部分一样。

一个完整的 pool 包括一个 pool_header 和透过这个 pool_header 管理的 block 集合；同理，一个完整的 arena 也包括一个 arena_object 和透过这个 arena_object 管理的 pool 集合。

在 arena_object 结构体的定义中，我们看到了 nextarena 和 prevarena 这两个东西，这似乎意味着会有一个或多个 arena 构成的链表。呃，这种猜测实际上只对了一半，实际上确实会存在多个 arena_object 构成的集合，但是这个集合不构成链表，而是一个数组。

数组的首地址由 arenas 来维护，这个数组就是 Python 中通用小块内存的**内存池**，所以现在我们算是知道这个内存池究竟是啥了。另一方面，nextarea 和 prevarena 也确实是用来连接 arena_object 组成链表的，咦，不是已经构成数组了吗？为啥又要来一个链表。

首先arena是用来管理一组 pool的集合的，arena_object 的作用看上去和 pool_header的作用是一样的。但是实际上，pool_header 管理的内存(block使用)和arena_object管理的内存(pool使用)有一点细微的差别，pool_header管理的内存和pool_header自身是一块连续的内存，但是arena_object与其管理的内存则是分离的。

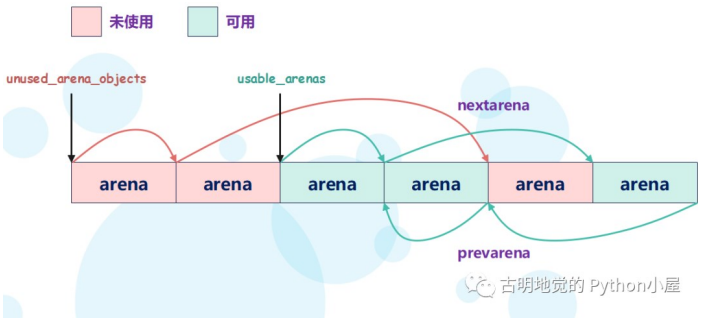


arena_object 和管理的 pool 集合，在内存上是分离的

乍一看，貌似没啥区别，不过一个是连着的，一个是分开的。但是这后面隐藏了这样一个事实：当pool_header被申请时，它所管理的block的内存也一定被申请了；但是当arena_object被申请时，它所管理的pool集合的内存则没有被申请。换句话说，arena_object和pool集合需要在某一时刻建立联系。

当一个arena的arena_object没有与pool集合建立联系的时候，这时的arena就处于**"未使用"**状态；一旦建立了联系，这时arena就转换到了**"可用"**状态。对于每一种状态，都有一个arena链表。

未使用的arena链表的表头是unused_arena_objects，多个arena_object之间通过nextarena连接，并且是一个单向的链表；而可用的arena链表的表头是usable_arenas，多个arena_object之间通过nextarena、prevarena连接，是一个双向链表。



当然啦，arena 指的是 arena_object 和管理的一组 pool 集合，所以图中应该是 arena_object，不应该是 arena。不过为了避免图像太长，就用 arena 代替了，我们理解就好。

总结一下就是：arena 的 arena_object 是通过数组进行组织的，这个数组（arenas）就是所谓的内存池。然后 arena 又分为未使用和可用，未使用的 arena，里面的 arena_object 会用单向链表组织起来；可用的 arena，里面的 arena_object 会用双向链表组织起来；

在运行期间，虚拟机使用new_arena来创建一个arena_object，我们来看看它是如何被创建的。

```
//arenas，指向多个arena_object组成的数组
static struct arena_object* arenas = NULL;

//arena数组中所有arena_object的个数
static uint maxarenas = 0;

//未使用的arena链表的表头
static struct arena_object* unused_arena_objects = NULL;

//可用的arena链表的表头
static struct arena_object* usable_arenas = NULL;

//初始化需要申请的arena_object的个数
#define INITIAL_ARENA_OBJECTS 16

static struct arena_object*
new_arena(void)
{
    //arena_object结构体指针
    struct arena_object* arenaobj;
    uint excess;        /* number of bytes above pool alignment */

    //[1]:判断是否需要扩充"未使用"的arena链表
    if (unused_arena_objects == NULL) {
        uint i;
        uint numarenas;
        size_t nbytes;

        //[2]:确定本次需要申请的arena_object的个数，并申请内存
        numarenas = maxarenas ? maxarenas << 1 : INITIAL_ARENA_OBJECTS;
        nbytes = numarenas * sizeof(*arenas);
        arenaobj = (struct arena_object *)PyMem_RawRealloc(arenas, nbyte
s);
        if (arenaobj == NULL)
            return NULL;
        arenas = arenaobj;

        //[3]:初始化新申请的arena_object，并将其放入"未使用"arena链表中
        for (i = maxarenas; i < numarenas; ++i) {
            arenas[i].address = 0;              /* mark as unassociated
*/
            arenas[i].nextarena = i < numarenas - 1 ?
                                  &arenas[i+1] : NULL;
```

```
        }

        /* Update globals. */
        unused_arena_objects = &arenas[maxarenas];
        maxarenas = numarenas;
    }

    /* Take the next available arena object off the head of the list. */
    //[4]:从"未使用"arena链表中取出一个"未使用"的arena
    assert(unused_arena_objects != NULL);
    arenaobj = unused_arena_objects;
    unused_arena_objects = arenaobj->nextarena;
    assert(arenaobj->address == 0);
    //[5]:申请arena管理的内存，256 kb
    address = _PyObject_Arena.alloc(_PyObject_Arena.ctx, ARENA_SIZE);
    if (address == NULL) {
        arenaobj->nextarena = unused_arena_objects;
        unused_arena_objects = arenaobj;
        return NULL;
    }
    arenaobj->address = (uintptr_t)address;
    //调整个数
    ++narenas_currently_allocated;
    ++ntimes_arena_allocated;
    if (narenas_currently_allocated > narenas_highwater)
        narenas_highwater = narenas_currently_allocated;
    //[6]:设置pool集合的相关信息，初始的时候为NULL
    arenaobj->freepools = NULL;
    arenaobj->pool_address = (block*)arenaobj->address;
    arenaobj->nfreepools = MAX_POOLS_IN_ARENA;
    //将pool的起始地址调整为系统页的边界
    excess = (uint)(arenaobj->address & POOL_SIZE_MASK);
    if (excess != 0) {
        --arenaobj->nfreepools;
        arenaobj->pool_address += POOL_SIZE - excess;
    }
    arenaobj->ntotalpools = arenaobj->nfreepools;

    return arenaobj;
}
```

因此我们可以看到，Python 首先会检查当前"未使用"链表中是否还有"未使用"的arena，检查的结果将决定后续的动作。

如果在"未使用"的area链表中还存在未使用的arena，那么Python会从中抽取一个arena，准确的说是 arena_object，接着调整"未使用"链表，让它和抽取的 arena_object 断绝一切联系。

然后 Python 申请了一块 256kb 大小的内存，将申请的内存地址赋给抽取出来的 arena_object 的 address。我们已经知道，arena 中维护的是 pool 集合，这块 256kb 的内存就是 pool 的容身之处，这时候 arena_object 就已经和 pool 集合建立联系了。

既然建立了联系，那么这个 arena 已经具备了成为可用的条件，该 arena 和未使用arena链表便脱离了关系，就等着被可用arena链表接收了，不过什么时候接收呢？先别急。

随后，Python 在代码的[6]处设置了一些 arena 用于维护 pool 集合的信息。需要注意的是，Python 将申请到的 256kb 内存进行了处理，主要是放弃了一些内存，并将可使用的内存边界(pool_address)调整到了与系统页对齐。

然后通过 arenaobj->freepools = NULL 将 freepools 设置为NULL，这不奇怪，基于对 freeblock 的了解，我们知道要等到释放一个 pool 时，这个 freepools 才会有用。最后我们看到，pool 集合占用的 256kb 内存在进行边界对齐后，实际是交给pool_address来维护了。

回到new_arena中的[1]处，如果unused_arena_objects为NULL，则表明目前系统中已经没有未使用的arena了，那么首先会扩大系统的 arenas 数组(内存池)。Python在内部通过一个 maxarenas 的变量维护了存储 arena_object 的数组的元素个数，然后在[2]处将待申请的 arena_object 的个数设置为 maxarenas 的 2 倍。当然首次初始化的时候 maxarenas 为 0，此时为 16。

在获得了新的 maxarenas 后，Python 会检查这个新得到的值是否溢出了。如果检查顺利通过，Python就会在[3]处通过 realloc 扩大 arenas 指向的数组，并对新申请的arena_object进行设置，特别是那个不起眼的 address，要将新申请的 address 一律设置为0。

实际上，这是一个标识 arena是处于"未使用"状态还是"可用"状态的重要标记。而一旦arena（内部的arena_object）和pool集合建立了联系，这个address就变成了非 0，看代码的[6]处。当然别忘记我们为什么会走到[3]这里，是因为 unused_arena_objects == NULL 了，而且最后还设置了unused_arena_objects，这样系统中又有了"未使用"的arena了，接下来Python就在[4]处对一个arena进行初始化了。

看一个宏：

```
1 #define SMALL_REQUEST_THRESHOLD 512
```

显然它是Python内部默认的小块内存与大块内存的分界点，也就是说，当申请的内存小于512个字节，pymalloc_alloc 会在内存池中申请内存；而当申请的内存超过了512字节，那么pymalloc_alloc 将退化为 malloc，通过操作系统来申请内存。当然，通过修改 Python 源代码我们可以改变这个值，从而改变 Python 的默认内存管理行为。

当申请的内存小于512字节时，Python会使用arena所维护的内存空间，那么Python内部对于arena的个数是否有限制呢？换句话说，Python对于这个小块空间内存池的大小是否有限制？其实这个决策取决于用户，Python提供了一个编译符号，用于控制是否限制内存池的大小，不过这里不是重点，只需要知道就行。

虽然我们上面花了不少笔墨介绍 arena，同时也看到 arena 是 Python 小块内存池的最上层结构，其实所有 arena_object 的集合就是小块内存池（arenas）。然而在实际的使用中，Python 并不直接与 arena 打交道，当申请内存时，最基本的操作单元并不是arena，而是 pool。估计有人到这里懵了，别急，慢慢来。

举个例子，当我们申请一个28字节的内存时，Python内部会在内存池寻找一块能够满足需求的 pool，从中取出一个 block返回，而不会去寻找 arena。这实际上是由 pool和 arena的属性决定的，在 Python中，pool 是一个有 size 概念的内存管理抽象体，一个 pool中的 block总是有确定的大小，这个 pool总是和某个 Size class index 对应。

而 arena是没有 size概念的内存管理抽象体，它内部会管理一个 pool 集合，也就是一组 pool。每一个 pool 里面所有 block的规模都是相同的，但是不同 pool，里面的 block的规模可以不同。

这就意味着，一个arena在某个时刻，其内部的所有 pool 的 szidx 可能都是相同的，也就是这些 pool 管理的都是同一种规格的 block，比如 32字节；而到了另一个时刻，由于系统需要，这个 arena被重新划分，其内部的所有 pool管理的 block可能又都变成64字节了。甚至一半的 pool管理的是 32字节block，另一半的 pool管理 64字节block，当然还有更多可能。

因为没有 size 的概念，这就决定了在进行内存分配和销毁时，所有的动作都是在 pool上完成的。

> 所以一个 arena，并不要求 pool集合中所有 pool管理的 block都必须一样；可以有管理 16字节block的pool，也可以有管理32字节block的 pool 等等。但是同一个 pool，里面的 block一定都是一样的。

对了， 一个 arena 可以管理一组 pool，那么对于 pool 而言，它怎么知道自己是隶属于哪一个 arena 呢？很简单，还记得 pool_header 里面的 arenaindex 字段吗？该字段负责维护 pool 隶属的 arena 的 arena_object 在数组中的索引。

此外内存池中的 pool不仅仅是一个有 size概念的内存管理抽象体，更进一步的，它还是一个有状态的内存管理抽象体。正如我们之前说的，一个 pool在 Python运行的任何一个时刻，总是处于以下三种状态中的一种：

empty状态

pool中所有的block都未被使用，处于这个状态的pool的集合通过其pool_header中的nextpool构成一个链表，这个链表的表头就是 arena_object 里的 freepools 字段。

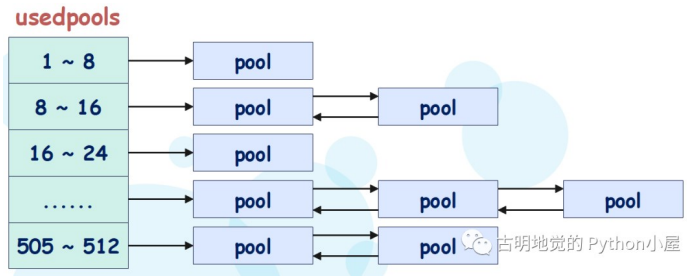
另外，虽然不同 szidx 的 pool 会管理不同的规格的 block，但不管什么样的 pool，只要处于 empty 状态，都会被加入到这个链表中。

used状态

pool中至少有一个block已经被使用，并且至少有一个block未被使用，这种类型的 pool 会通过内部的 prevpool 字段和 nextpool 字段组成一个双向链表。但是注意，和 empty 状态的 pool 不同，used 状态的 pool 组成的双向链表有 64 条。

也就是说，对于 used 状态的 pool 而言，每条链表上面的 pool 的种类是相同的（szidx 一样），管理的都是同一种规格的 block。而不同 szidx 的 pool，则位于不同的双向链表中。

而这 64 条双向链表会由一个名为 usedpools 的数组进行管理。

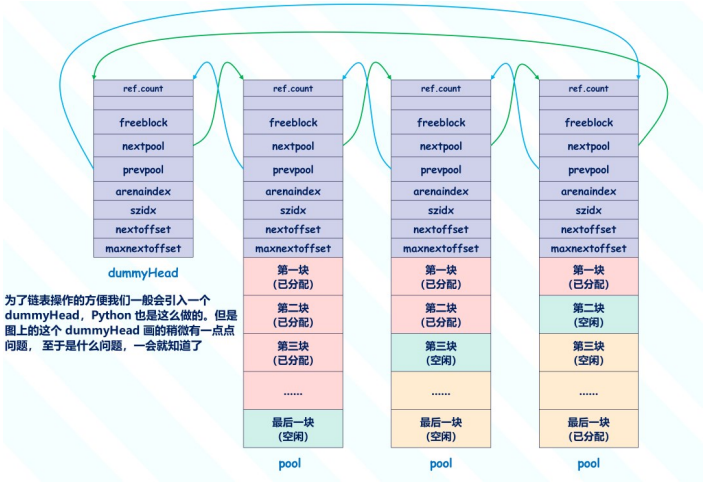


full状态

pool中所有的block都已经被使用，此时 Python 会将它闲置在一旁。因为full状态的pool是各自独立的，没有像其他状态的pool一样，连接成一个链表。

used状态的 pool 组成的双向链表

对于 used 状态的 pool 会组成一个双向链表，而这样的链表有 64 条，并且这 64 条双向链表会放在一个名为 usedpools 的数组中。关于这个数组一会再说，我们先以其中的一条链表为例，看看它的结构。

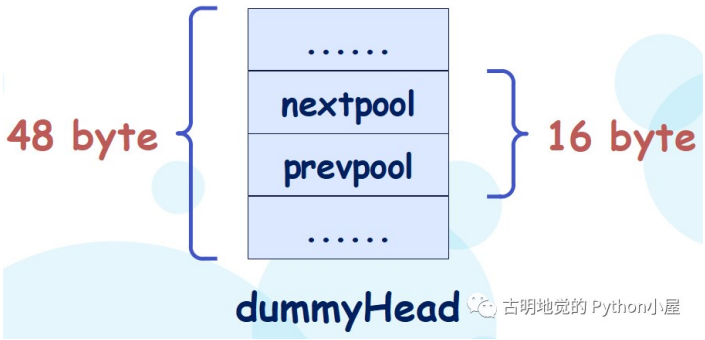


同一个双向链表中所有 pool 管理的内存块大小规格都是一样的，另外为了简化链表处理逻辑，Python 引入了一个 dummyHead，也就是虚拟头结点。如果这个链表是空，那么 dummyHead 的 nextpool 和 prevpool 都指向它自己。

另外 dummyHead 只是为了方便链表的维护，并不管理内存块，所以它只有一个 pool_header，不会为它分配 4kb 的内存。但即便如此，仍然存在着内存浪费，因为 pool_header 内部有很多字段，可对于维护链表而言，只需要用 nextpool 和 prevpool 两个字段就够了。

这两个字段总共 16 字节，而一个完整的 pool_header 是 48 字节，所以给 dummyHead 分配一个完整的 pool_header 会浪费 32 字节的内存。一条链表会浪费 32 字节，而这样的链表有 64 条，所以总共会浪费 2048 字节、也就是 2kb 的内存。

因此 Python 的作者为了不让这样的事情发生，在分配的时候真的就只分配了 16 字节，并且把它当成是一个完整的 pool_header 来用。但问题是一个 pool_header 实例是 48 字节，只分配 16 字节也不够啊，Python 是怎么把 16 字节当成 48 字节来用的呢?

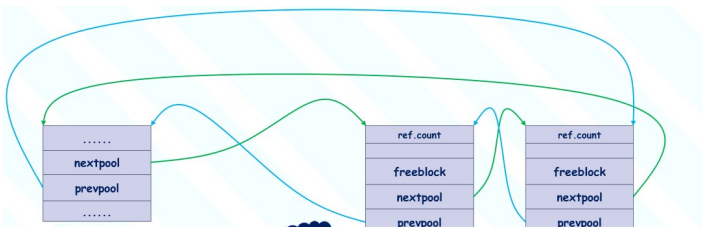


非常简单，分配的 16 字节不够的话，再借 32 字节不就行了吗。因此将 nextpool 和 prevpool 字段前后的空间都**想象成**是 pool_header 的一部分，这样就能够得到一个完整的 pool_header 结构体实例。

虽然这个实例是非法的，因为实际上这两个字段前后的空间不属于pool_header，但是这并不妨碍在精神层面上 YY 一下，就假装是自己的（相当于借用）。因为除了 nextpool 和 prevpool 之外，其它部分的内存根本不会访问到，因此即便 pool_header 实例非法，也无伤大雅。

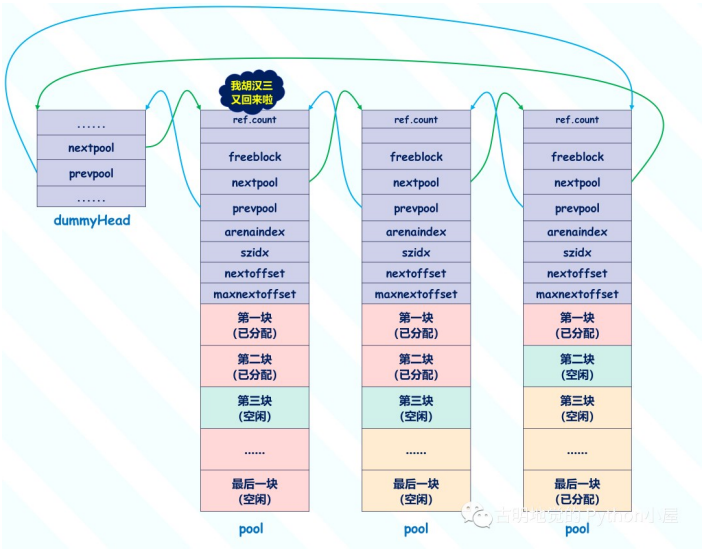
所以我们上面画的那张双向链表的图，里面的 dummyHead 是不对的，真正存在的只有 nextpool 和 prevpool 两个字段，至于这两个字段前后的空间到底存了什么内容，在介绍 usedpools 时再聊。

然后此时来了一个内存分配请求，那么会优先从链表的第一个 pool 里面分配内存块，如果可用的内存块都分配完了，那么就将它从链表中剔除。



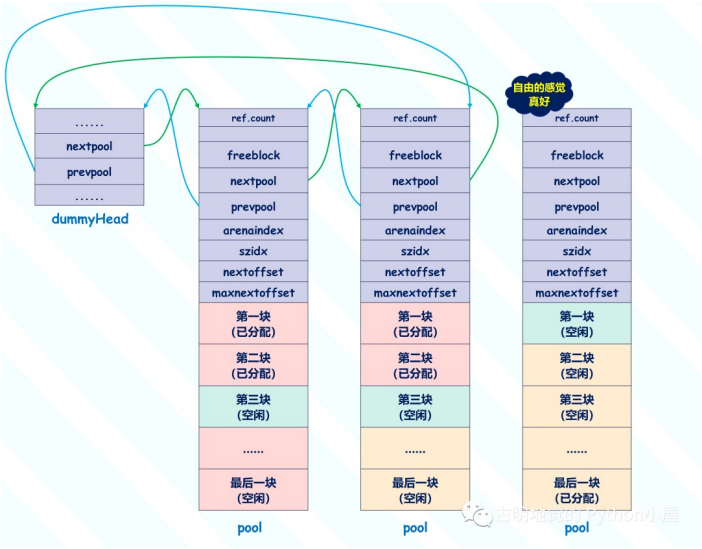
当一个内存块(block)被回收，Python根据块地址计算得到距离该块最近的pool边界地址，计算方式就是我们上面说的那个宏：POOL_ADDR，将块(block)地址对齐为内存页(pool)尺寸的整数倍，便得到pool地址，然后对内存块进行处理。

如果pool状态是由full变成used，那么Python还会将它插回到可用pool链表的头部。



插入到可用pool链表的头部是为了保证内存块使用较多的 pool在链表的前面，以便优先使用。位于尾部的 pool 被使用的概率很低，随着时间的推移，更多的内存块被释放出来，慢慢变空。因此可用 pool 链表会变得头重脚轻，靠前的 pool 比较满，靠后的 pool 比较空。

当一个pool中所有的内存块(block)都被释放，状态就变成了empty，那么 Python 就会将它移除可用 pool 链表，将其加入到 freepools 链表中，或者直接归还给操作系统。



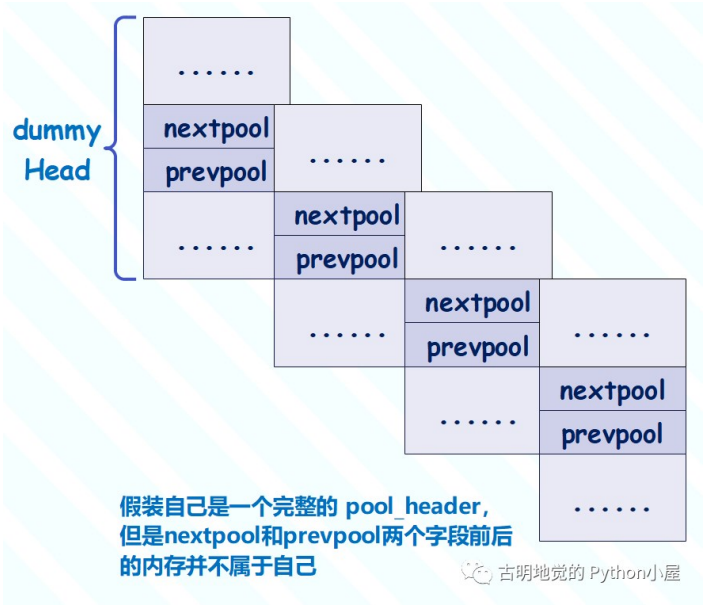
当然啦，pool 链表里的任一节点均有机会完全空闲下来，这由概率决定，尾部节点概率最高。

64 个可用 pool 链表组成的数组

Python 内存块有 64 种，而每种规格的多个内存块都需要用一个可用 pool 链表来维护，那么显然会有 64 种 pool 链表。

那么如何组织这么多 pool 链表呢？最直接的办法就是分配一个长度为 64 的数组，这个数组就是

上面提到的 usedpools，它里面存储了 64 个 pool 双向链表的 dummyHead。但我们知道这个 dummyHead 并不是一个正常的 pool_header，而是 nextpool 和 prevpool 两个字段、再加上前后各 16 字节的内存假装出来的 pool_header。



所以现在我们知道了，每个 dummyHead 只能访问 nextpool 和 prevpool 两个字段，至于它前后的内存只是假装是自己的，实际上并不属于自己。那属于谁呢？图上画的很明显了，显然是属于其它的 dummyHead。

因为每个 dummyHead 只分配了 16 字节的内存，用于 nextpool 和 prevpool，那要想把自己当成 48 字节的 pool_header，那么只能借助它的上一个 dummyHead 和下一个 dummyHead。也正因为如此，Python 才能从这个 usedpools 数组中扣掉 2k 的内存。所以高级程序猿对内存的精打细算，完全堪比、甚至凌驾于菜市场买菜的大妈。

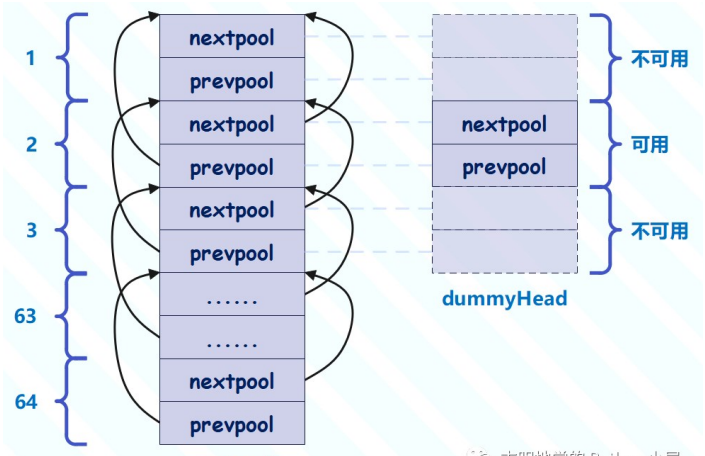
而 usedpools 数组则维护着所有处于 used 状态的pool 链表（的 dummyHead），当申请内存时，Python 就会通过 usedpools 寻找到一个可用的 pool(处于used状态)，从中分配一个 block。

那么这个数组长什么样子呢？

```
#define PTA(x)  ((poolp )((uint8_t *)&(usedpools[2*(x)]) - 2*sizeof(block *)))
#define PT(x)   PTA(x), PTA(x)

static poolp usedpools[2 * ((NB_SMALL_SIZE_CLASSES + 7) / 8) * 8] = {
    PT(0), PT(1), PT(2), PT(3), PT(4), PT(5), PT(6), PT(7)
#if NB_SMALL_SIZE_CLASSES > 8
    , PT(8), PT(9), PT(10), PT(11), PT(12), PT(13), PT(14), PT(15)
#if NB_SMALL_SIZE_CLASSES > 16
    , PT(16), PT(17), PT(18), PT(19), PT(20), PT(21), PT(22), PT(23)
#if NB_SMALL_SIZE_CLASSES > 24
    , PT(24), PT(25), PT(26), PT(27), PT(28), PT(29), PT(30), PT(31)
#if NB_SMALL_SIZE_CLASSES > 32
    , PT(32), PT(33), PT(34), PT(35), PT(36), PT(37), PT(38), PT(39)
#if NB_SMALL_SIZE_CLASSES > 40
    , PT(40), PT(41), PT(42), PT(43), PT(44), PT(45), PT(46), PT(47)
#if NB_SMALL_SIZE_CLASSES > 48
    , PT(48), PT(49), PT(50), PT(51), PT(52), PT(53), PT(54), PT(55)
#if NB_SMALL_SIZE_CLASSES > 56
    , PT(56), PT(57), PT(58), PT(59), PT(60), PT(61), PT(62), PT(63)
#if NB_SMALL_SIZE_CLASSES > 64
#error "NB_SMALL_SIZE_CLASSES should be less than 64"
#endif /* NB_SMALL_SIZE_CLASSES > 64 */
#endif /* NB_SMALL_SIZE_CLASSES > 56 */
#endif /* NB_SMALL_SIZE_CLASSES > 48 */
#endif /* NB_SMALL_SIZE_CLASSES > 40 */
#endif /* NB_SMALL_SIZE_CLASSES > 32 */
#endif /* NB_SMALL_SIZE_CLASSES > 24 */
#endif /* NB_SMALL_SIZE_CLASSES > 16 */
#endif /* NB_SMALL_SIZE_CLASSES >  8 */
};
```


可能有人感觉这个数组看起来有点怪异，别急，我们画一张图就清晰了。



因此更准确的说，数组里面存储的其实是一组组的 nextpool 和 prevpool，当然一组 nextpool 和 prevpool 也可以简单理解为一个 dummyHead，只不过少了 32 字节。

当我们想获取第二个 dummyHead，那么就获取第二组的 nextpool 和 prevpool 即可。但别忘了，此时只有 16 字节，而 dummyHead 要想伪装成 pool_header 还差 32 字节。而差的这 32 字节，就由第一组和第三组的 nextpool 和 prevpool 来承载，只不过它们不能访问。

> 所以 usedpools 里面实际上有 128 个元素，如果把 nextpool 和 prevpool 整体看成是一个 dummyHead（少了 32 字节）的话，那么逻辑上还是维护了 64 条双向链表。

所以当程序请求 5 字节时，Python 将分配 8 字节内存块，通过数组中索引为 0 的 dummyHead 即可找到 8 字节 pool 链表；如果程序请求 56 字节，Python 将分配 64 字节内存块，通过数组中索引为 7 的 dummyHead 即可找到。

但是在查找的时候有一个小 trick，再看一下上图，里面的每一个 nextpool 和 prevpool 都指向了上一个 nextpool。考虑一下当申请 28 字节的情形，前面我们说到，Python首先会获取 Size class index，显然这里是 3。那么在 usedpools中，就会寻找索引为 3+3=6 的元素，发现 usedpools[6] 的值是指向 usedpools[4] 的地址。

好晕啊，好吧，现在对照 pool_header 的定义来看一看 **usedpools[6] -> nextpool** 这个指针指向哪里了？

```
1
2  //Objects/obmalloc.c
3  struct pool_header {
4      union { block *_padding;
5              uint count; } ref;
6      block *freeblock;
7      struct pool_header *nextpool;
8      struct pool_header *prevpool;
9      uint arenaindex;
10     uint szidx;
11     uint nextoffset;
12     uint maxnextoffset;
13 };
14
15 typedef struct pool_header *poolp;
```

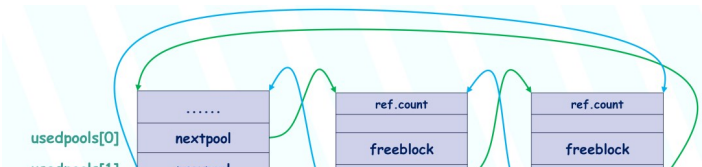
显然是从usedpools[4]开始向后偏移 16 个字节（一个ref的大小加上一个freeblock的大小）后的内存，正好是 usedpools[6] 的地址。因为 usedpools 里面存储的不是完整的 pool_header，而是 nextpool 和 prevpool，总共 16 字节。而通过 pool_header 结构体可以发现，pool_header 中 nextpool 成员的偏移量也是 16 字节。

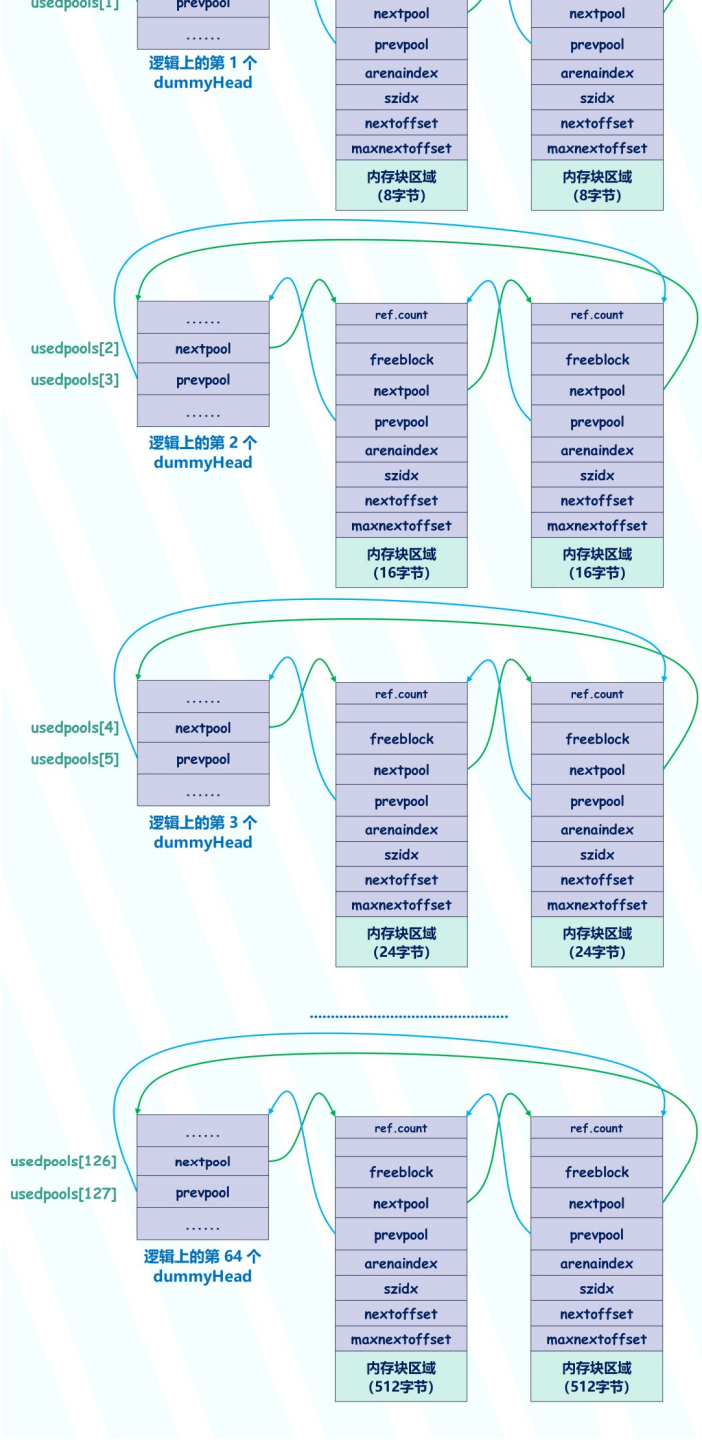
这个设计就非常巧妙，以后只需要通过 **usedpools[i+i] -> nextpool** 便可快速地从众多的 pool 中寻找到一个最适合当前内存需求的 pool，并从中分配一块block。当然，也可以很方便地将一个 pool 加入到 usedpools 中。

```
1  static int
2  pymalloc_alloc(void *ctx, void **ptr_p, size_t nbytes)
3  {
4      block *bp;
5      poolp pool;
6      poolp next;
7      uint size;
8      //...
9      LOCK();
10     //获得size class index
11     size = (uint)(nbytes - 1) >> ALIGNMENT_SHIFT;
12     //直接通过usedpools[size+size]
13     //这里的size就是我们上面说的 i
14     pool = usedpools[size + size];
15     //如果usedpools中有可用的pool
16     if (pool != pool->nextpool) {
17         //... //有可用pool
18     }
19     //... //无可用pool，尝试获取empty状态的pool
20 }
21
```

关于具体的查找过程可能不是很好理解，当然也不必深究，直接当 usedpools 里面存储了 64 个 dummyHead 即可。分配 8 字节内存块，就去找索引为 0 的 dummyHead；分配 64 字节内存块，就去找索引为 7 的 dummyHead。

整个 usedpools 全貌如下：





当然啦，每个链表上的 pool 的个数是不同的，这里为了画图方便，就假设每个链表上面都挂了两个 pool。

block 的释放

最后看看 block 的释放，释放 block 实际上就是将它归还给pool。我们知道 pool 存在 3 种状态，这 3 种状态的区别也说的很清楚了，总之状态不同，所在的位置也不同。

而当我们释放一个block 时，可能会引起 pool 状态的转变，这种转变分为两种情况：

- used状态转变为empty状态；
- full状态转变为used状态；

释放 block 由 pymalloc_free 函数负责。

```
//obmalloc.c
static int
pymalloc_free(void *ctx, void *p)
{
    poolp pool;
    block *lastfree;
    poolp next, prev;
    uint size;
    pool = POOL_ADDR(p);
    if (!address_in_range(p, pool)) {
        return 0;
    }

```

```
    LOCK();
    assert(pool->ref.count > 0);
    //设置离散可用的block链表
    *(block **)p = lastfree = pool->freeblock;
    pool->freeblock = (block *)p;
    //如果!lastfree成立，那么意味着不存在lastfree
    //说明这个pool在释放block之前是满的
    if (!lastfree) {
        /* Pool was full, so doesn't currently live in any list:
         * link it to the front of the appropriate usedpools[] list.
         * This mimics LRU pool usage for new allocations and
         * targets optimal filling when several pools contain
         * blocks of the same size class.
         */
        //当前pool处于full状态，在释放一块block之后
        //需要将其转换为used状态，并重新链入到usedpools的头部
        --pool->ref.count;
        assert(pool->ref.count > 0);
        size = pool->szidx;
        next = usedpools[size + size];
        prev = next->prevpool;
        pool->nextpool = next;
        pool->prevpool = prev;
        next->prevpool = pool;
        prev->nextpool = pool;
        goto success;
    }

    struct arena_object* ao;
    uint nf;  /* ao->nfreepools */

    //否则到这一步表示lastfree有效，也就是pool处于used状态
    //并且pool回收了一个block之后，ref.count 不等于 0
    //因此仍处于 used 状态，那么直接跳转到 success 标签
    if (--pool->ref.count != 0) {
        /* pool isn't empty:  leave it in usedpools */
        goto success;
    }
    /* Pool is now empty:  unlink from usedpools, and
     * link to the front of freepools.  This ensures that
     * previously freed pools will be allocated later
     * (being not referenced, they are perhaps paged out).
     */
    //否则说明pool为空
    next = pool->nextpool;
    prev = pool->prevpool;
    next->prevpool = prev;
    prev->nextpool = next;

    //将pool放入freepools维护的链表中
    ao = &arenas[pool->arenaindex];
    pool->nextpool = ao->freepools;
    ao->freepools = pool;
    nf = ++ao->nfreepools;

    if (nf == ao->ntotalpools) {
        //调整usable_arenas链表
        if (ao->prevarena == NULL) {
            usable_arenas = ao->nextarena;
            assert(usable_arenas == NULL ||
                   usable_arenas->address != 0);
        }
        else {
            assert(ao->prevarena->nextarena == ao);
            ao->prevarena->nextarena =
                ao->nextarena;
        }
        /* Fix the pointer in the nextarena. */
        if (ao->nextarena != NULL) {
            assert(ao->nextarena->prevarena == ao);
            ao->nextarena->prevarena =
                ao->prevarena;
        }
        //调整"未使用"arena链表
        ao->nextarena = unused_arena_objects;
        unused_arena_objects = ao;

        //将内存归还给操作系统
        _PyObject_Arena.free(_PyObject_Arena.ctx,
                             (void *)ao->address, ARENA_SIZE);
        //设置address，将arena的状态转为"未使用"
        ao->address = 0;
        --narenas_currently_allocated;
```

```
97
98          goto success;
99      }
100 }
```

实际上在 Python2.4 之前，arena 是不会释放pool的，这样的话就会引起内存泄漏。比如我们申请 **10 * 1024 * 1024** 个 16 字节的小内存，这就意味着必须使用 160MB 的内存。

由于 Python 默认会使用全部的 arena 来满足你的需求（这一点前面没有提到），那么当我们将所有 16 字节的内存全部释放了，这些内存也会回到 arena 的控制之中，这都没有问题。但关键的是，这些内存是被 arena 控制的，并没有交还给操作系统，所以这 160MB 的内存始终会被 Python 占用，即使后面程序不再需要这 160MB 的内存，那么这就不就相当于浪费了吗?

由于这种情况必须在大量持续申请小内存对象时才会出现，因为大的话会自动交给操作系统，小的才会由 arena 控制，而持续申请大量小内存的情况几乎不会碰到，所以这个问题在 2.4 之前一直没有人发现，因此也就留在了 Python 中。

直到某一天，国外一个工程师在跑任务的时候发现即使是在低峰期，内存占用率仍然居高不下，这才挖出了这个隐藏的问题并进行了反馈。于是在 Python2.5 的时候得到了解决。

而早期的 Python 之所以不将内存归还给操作系统，是因为当时 arena 还没有区分"未使用"和"可用"两种状态。但到了 Python2.5，arena 已经可以将自己维护的 pool 集合释放，交给操作系统了，从而将"可用"状态转化为"未使用"状态。而当 Python 处理完pool，就开始处理 arena 了。

而对 arena 的处理实际上分为了 4 种情况：

1）如果 arena 中所有的 pool 都是 empty状态，则释放 pool 集合所占用的内存。而一旦 pool 集合释放，那么 arena_object 就和管理的 pool 集合失去了联系，因此状态就由"可用"变成了"未使用"；

2）如果之前 arena 中没有 empty状态的 pool，那么在"可用"链表中就找不到该arena，但由于现在 arena 中有了一个pool，所以需要将这个arena链入到"可用"链表的表头；

3）如果arena中 empty状态的 pool 的个数为 n，那么会从"可用"arena链表中开始寻找arena可以插入的位置，将arena插入到"可用"链表。这样操作的原因就在于"可用"arena链表实际上是一个有序的链表，从表头开始往后，每一个arena中empty状态的pool的个数、即nfreepools，都不能大于前面的arena，也不能小于后面的arena。保持这样有序性的原则是因为分配block时，是从"可用"链表的表头开始寻找可用arena的，这样就能保证如果一个arena的empty pool数量越多，它被使用的机会就越少。因此它最终释放其维护的pool集合的内存的机会就越大，这样就能保证多余的内存会被归还给操作系统；

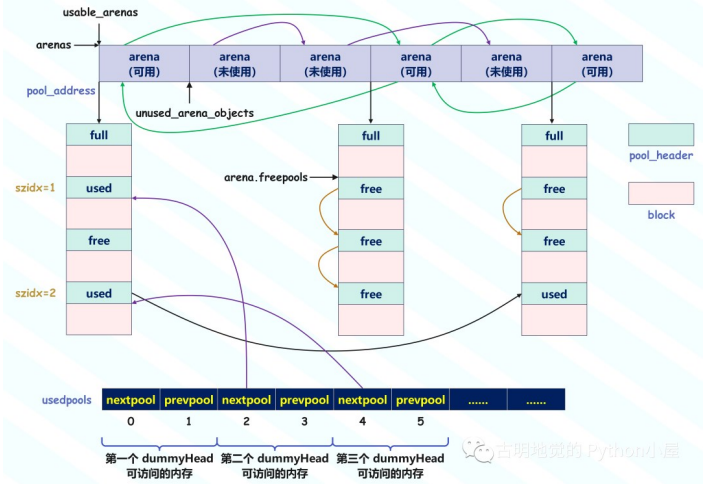
4）其他情况，则不对arena进行任何处理；

小结

对于一个用 C 开发的庞大的软件（虽然 Python是一门高级语言，但是执行对应代码的解释器则可以看成是 C 的一个软件），其中的内存管理可谓是最复杂、最繁琐的地方了。不同尺度的内存会有不同的抽象，这些抽象在各种情况下会组成各式各样的链表，非常复杂。

不过我们还是能从一个整体的尺度把握整个内存池，尽管不同的链表变幻无常，但只需记住，所有的内存都在 arenas（或者说那个存放多个 arena_object 的数组）的掌握之中。

整个内存池全景如下：



以上就是 Python 管理内存的全部秘密，包括内存池的实现细节等等。总之，凡是和内存管理相关的都不简单，更详细的内容可以自己进入 Objects/obmalloc.c 中查看对应的源码，主要看两个函数：

- pymalloc_alloc: 负责内存分配；
- pymalloc_free: 负责内存释放；

关于内存管理和内存池我们就说到这里，下一篇介绍 Python 的垃圾回收机制。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》94. Python 的引用计数与标记-清除

下一篇 ›
《源码探秘 CPython》92. Python 是如何管理内存的？（上）