Python 源码剖析

——Python 的内存管理机制

本文作者: Robert Chen(search.pythoner@gmail.com)

内存管理，对于 Python 这样的动态语言，是至关重要的一部分，它在很大程度上甚至决定了 Python 的执行效率，因为在 Python 的运行中，会创建和销毁大量的对象，这些都涉及到内存的管理。另一方面，和 Java、C#这些编程语言一样，Python 提供了对内存的垃圾收集（GC）机制，将开发者从繁琐的手动维护内存的工作中解放出来，Python 中的 GC 机制又是如何实现的呢？在这一章中，我们将来细致地剖析 Python 内部所采用的内存管理机制。

1 内存管理架构

在剖析 Python 的内存管理架构之前，有一点需要说明。Python 中所有的内存管理机制都有两套实现，这两套实现由编译符号 PYMALLOC_DEBUG 控制，当该符号被定义时，使用的是 debug 模式下的内存管理机制，这套机制在正常的内存管理动作之外，还会记录许多关于内存的信息，以方便 Python 在开发时进行调试；而该符号未被定义时，Python 的内存管理机制只进行正常的内存管理动作。在本章中，我们将关注的焦点只放在非 debug 模式下的内存管理机制上。

在 Python 中，内存管理机制被抽象成一种层次似的结果，如图 1 所示。

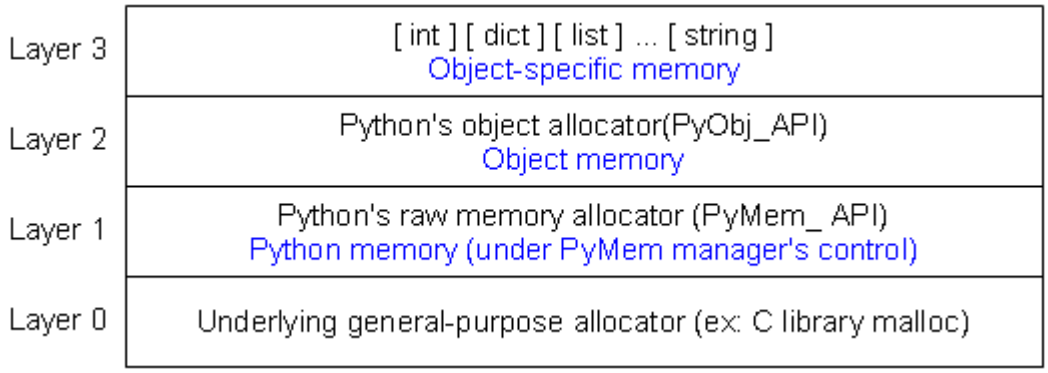

图 1 Python 内存管理机制的层次结构

在最底层（第 0 层），是操作系统提供的内存管理接口，比如 C 运行时所提供的 malloc 和 free 接口。这一层是由操作系统实现并管理的，Python 不能干涉这一层的行为。从这一层往上，剩余的三层都是由 Python 实现并维护。

第一层是 Python 基于第 0 层操作系统的内存管理接口包装而成的，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 的管理接口。Python 是用 C 实现的，为什么还需要在 C 所提供的内存管理接口之上再提供一层并不太多实际意义的包装层呢？这是因为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，对于某些特殊的情况不同操作系统有不同的行为。比如调用 malloc(0)，有的操作系统会返回 NULL，表示内存申请失败；然而有的操作系统会返回一个貌似正常的指针，但是这个指针所指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_为前缀的函数族，下面我们来看一看这第一层内存管理机制：

```
[pymem.h]
PyAPI_FUNC(void *) PyMem_Malloc(size_t);
PyAPI_FUNC(void *) PyMem_Realloc(void *, size_t);
```

```
PyAPI_FUNC(void) PyMem_Free(void *);

[object.c]
void* PyMem_Malloc(size_t nbytes)
{
    return PyMem_MALLOC(nbytes);
}

void* PyMem_Realloc(void *p, size_t nbytes)
{
    return PyMem_REALLOC(p, nbytes);
}

void PyMem_Free(void *p)
{
    PyMem_FREE(p);
}
```

我们看到，在第一层，Python 提供了类似于 C 中 malloc，realloc，free 的语义。有趣的是，PyMem_Malloc 是通过一个宏 PyMem_MALLOC 来实现的，对于 PyMem_Realloc 和 PyMem_Free，情况也是如此：

```
[pymem.h]
#define PyMem_MALLOC(n)          malloc((n) ? (n) : 1)
#define PyMem_REALLOC(p, n)      realloc((p), (n) ? (n) : 1)
#define PyMem_FREE               free
```

显然，PyMem_Malloc 其实就是 C 中的 malloc，只是对申请大小为 0 的内存这种特殊情况进行了处理。Python 不允许申请大小为 0 的内存空间，它会强制将其装换为申请大小为 1 个字节的内存空间，从而避免了不同操作系统上的不同运行时行为。

这里有一个有趣的问题，为什么 Python 会同时提供函数和宏这两套接口呢？其实，这正是 Python 为了执行效率殚精竭虑的表现。使用宏可以避免一次函数调用的开销，提高运行效率。但是对于用户使用 C 来编写 Python 的扩展模块时，使用宏是危险的，因为随着 Python 的不断演进，其内存管理机制的具体实现很可能会发生变化，所以虽然 Python 中宏的名字是不会变的，但是其所代表的实现代码是会变的，因此，使用旧的宏编写的 C 扩展模块就可能与新版

本的 Python 产生二进制不兼容，这是一个非常危险的陷阱。因此 Python 对于这些内存管理接口，总是同时提供函数和宏这两套接口，这一点我们在后面会经常看到。如果使用 C 来编写 Python 的扩展模块，使用函数接口是一个良好的编程习惯。

到现在为止，我们所介绍的内存管理接口都与 malloc 等有相同的语义，仅仅是分配 raw memory 而已。其实在第一层中，Python 还提供了面向 Python 中类型的内存分配器：

```
[pymem.h]
#define PyMem_New(type, n) \
    ( (type *) PyMem_Malloc((n) * sizeof(type)) )
#define PyMem_NEW(type, n) \
    ( (type *) PyMem_MALLOC((n) * sizeof(type)) )

#define PyMem_Resize(p, type, n) \
    ( (p) = (type *) PyMem_Realloc((p), (n) * sizeof(type)) )
#define PyMem_RESIZE(p, type, n) \
    ( (p) = (type *) PyMem_REALLOC((p), (n) * sizeof(type)) )

#define PyMem_Del       PyMem_Free
#define PyMem_DEL       PyMem_FREE
```

在 PyMem_Malloc 中，和 malloc 一样，程序员需要自行提供所申请空间的大小。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。

第一层所提供的内存管理接口其功能是有限的，想象一下，假如我要创建一个 PyIntObject 对象，还需要进行许多额外的工作，比如设置对象的类型对象参数，初始化对象的引用计数值等等。为了简化 Python 自身的开发，Python 在比第一层更高的抽象层次上提供了第二层内存管理接口。在这一层，是一组以 PyObje_为前缀的函数族，主要提供了创建 Python 对象的接口。这一套函数族又被唤作 Pymalloc 机制，从 Python2.1 开始，它才慢慢登上历史舞台，在 Python2.1 和 2.2 中，这个机制是作为实验性质的机制，所以默认情况下是不打开的，只有自己通过—with-pymalloc 编译符号重新编译 Python，才能激活 Pymalloc 机制。在 Python2.3 发布时，Pymalloc 机制已经经过了长期的优化和稳定，终于登上了正宫的位置，在默认情况下被

打开了。

在第二层内存管理机制之上，对于 Python 中的一些常用对象，比如整数对象，字符串对象等等，Python 又构建了更高抽象层次的内存管理策略。对于第三层的内存管理策略，主要就是对象缓冲池机制，这一点在本书第一部分我们已经剖析了。而第一层的内存管理机制仅仅是对 malloc 的简单包装，真正在 Python 中发挥巨大作用的内存管理机制，同时也是 GC 的藏身之处，就在第二层内存管理机制中。所以，从下面开始，我们将进入对这套机制的剖析。

2 小块空间的内存池

在 Python 中，许多时候申请的内存都是小块的内存，这些小块内存在申请后，很快又会被释放，由于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内存池机制。这就意味着 Python 在运行期间会大量地执行 malloc 和 free 的操作，频繁地在用户态和核心态之间进行切换，这将严重影响 Python 的执行效率。为了加速 Python 的执行效率，Python 引入了一个内存池机制，用于管理对小块内存的申请和释放。这也就是之前提到的 Pymalloc 机制，前面我们已经看到，这套机制在 Python2.5 中默认是启动了的，也就是说，在 Python2.5 中，用于管理小块内存的内存池就被激活了，并通过 PyObject_Malloc，PyObject_Realloc 和 PyObject_Free 三个接口暴露给 Python。

在 Python2.4 中，整个小块内存的内存池可以视为一个层次结构，在这个层次结构中，一共分为 4 层，从下至上分别是：block，pool，arena 和内存池。需要说明的是，block，pool，和 arena 都是 Python 代码中可以找到的实体，而最顶层的“内存池”只是一个概念上的东西，表示 Python 对于整个小块内存分配和释放行为的内存管理机制。

2.1 Block

在最底层，block 是一个确定大小的内存块，在 Python 中，有很多种 block，不同种类的 block 都有不同的内存大小，这个内存大小的值被称为 size class。为了在当前主流的 32 位平台和 64 位平台上都能获得最佳的性能，所有的 block 的长度都是 8 字节对齐的。

```
[obmalloc.c]
#define ALIGNMENT        8        /* must be 2^N */
#define ALIGNMENT_SHIFT     3
#define ALIGNMENT_MASK       (ALIGNMENT - 1)
```

同时，Python 为 block 的大小设定了一个上限，当申请的内存大小小于这个上限时，Python 可以使用不同种类的 block 来满足对内存的需求；当申请的内存大小超过了这个上限，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 函数族，来处理。这个上限值在 Python2.5 中被设置为 256。

```
[obmalloc.c]
#define SMALL_REQUEST_THRESHOLD  256
#define NB_SMALL_SIZE_CLASSES (SMALL_REQUEST_THRESHOLD / ALIGNMENT)
```

根据 SMALL_REQUEST_THRESHOLD 和 ALIGNMENT 的限定，实际上，我们可以由此得到不同种类的 block 的 size class 分别为：8，16，32，……，256。每个 size class 对应一个 size class index，这个 index 从 0 开始。所以对于小于 256 字节的小块内存的分配，我们可以得到如下的表：

```
* 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
*    ...                    ...                     ...
*  241-248                  248                     30
*  249-256                  256                     31
*
*  0, 257 and up: routed to the underlying allocator.
```

也就是说，当我们申请一块大小为 28 字节的内存时，实际上 PyObject_Malloc 从内存池中划给我们的内存是 32 字节的一个 block，从 size class index 为 3 的 arena 中划出。下面的两个式子给出了在 size class 和 size class index 之间的转换：

```
//从 size class index 转换到 size class
#define INDEX2SIZE(I) (((uint)(I) + 1) << ALIGNMENT_SHIFT)

//size class 转换到 size class index
size = (uint )(nbytes - 1) >> ALIGNMENT_SHIFT;
```

现在，需要指出一个相当关键的点，虽然我们这里谈论了很多 block，但是在 Python 中，block 只是一个概念，但是在 Python 源码中没有与之对应的实体存在。之前我们说对象，对象在 Python 源码中有对应的 PyObject；我们说列表，列表在 Python 源码中对应 PyListObject、PyType_List。这里的 block 就很奇怪了，它仅仅是概念上的东西，我们知道它是具有一定大小的内存，但它不与 Python 源码里的某个东西对应。然而，Python 却提供了一个管理 block 的东西，这就是下面要剖析的 pool。

2.2 Pool

一组 block 的集合称为一个 pool，换句话说，一个 pool 管理着一堆有固定大小的内存块。事实上，pool 管理着一大块内存，它有一定的策略，将这块大的内存划分为多个小的内存块。在 Python 中，一个 pool 的大小通常为一个系统内存页，由于当前大多数 Python 支持的系统的

内存页都是 4K，所以 Python 内部也将一个 pool 的大小定义为 4K。

```
[obmalloc.c]
#define SYSTEM_PAGE_SIZE    (4 * 1024)
#define SYSTEM_PAGE_SIZE_MASK   (SYSTEM_PAGE_SIZE - 1)

#define POOL_SIZE       SYSTEM_PAGE_SIZE    /* must be 2^N */
#define POOL_SIZE_MASK      SYSTEM_PAGE_SIZE_MASK
```

虽然 Python 没有为 block 提供对应的结构，但它对于 pool 却是相当照顾的。Python 源码中的 pool_header 就是为 pool 这个概念提供的实现。

```
[obmalloc.c]
typedef uchar block;

/* Pool for small blocks. */
struct pool_header {
    union { block *_padding;
        uint count; } ref;  /* number of allocated blocks    */
    block *freeblock;       /* pool's free list head         */
    struct pool_header *nextpool;   /* next pool of this size class  */
    struct pool_header *prevpool;   /* previous pool       ""        */
    uint arenaindex;        /* index into arenas of base adr */
    uint szidx;         /* block size class index    */
    uint nextoffset;        /* bytes to virgin block     */
    uint maxnextoffset;     /* largest valid nextoffset  */
};
```

我们刚刚才说了一个 pool 的大小在 Python2.5 中是 4k，但是看看这个 pool_header 呢？就算是用大腿看也能看出 pool_header 吃不完 4k 的内存。关键就在于“header”这个词，原来这个 pool_header 仅仅是一个 pool 的头部，4k 的内存，除去 pool_header，还有很大一块。还记得我们说过 pool 管理着一堆 block 吗？对了，这剩下的很大一块的内存就是 pool 中维护的 block 的集合占用的内存。

前面提到 block 是有固定大小的内存块，因此，pool 也携带了大小这样的信息。一个 pool 管理的所有 block，它们的大小都是一样的。也就是说，一个 pool 可能管理了 100 个 32 个字节

的 block，也可能管理了 100 个 64 个字节的 block，但是绝不会有一个管理了 50 个 32 字节的 block 和 50 个 64 字节的 block 的 pool 存在。每一个 pool 都和一个 size 联系在一起，更确切地说，都和一个 size class index 联系在一起。这就是 pool_header 中的 szindex 的意义。

假设我们手上现在有一块 4k 的内存，来看看 Python 是如何将这块内存改造为一个管理 32 字节 block 的 pool，并从 pool 中取出第一块 block 的。

```
[obmalloc.c]-[convert 4k raw memory to pool]
#define ROUNDUP(x)              (((x) + ALIGNMENT_MASK) & ~ALIGNMENT_MASK)
#define POOL_OVERHEAD           ROUNDUP(sizeof(struct pool_header))
#define struct pool_header* poolp
#define uchar block

poolp pool;
block* bp;

…… // pool 指向了一块 4k 的内存

pool->ref.count = 1;

//设置 pool 的 size class index

pool->szidx = size;

//将 size class index 转换为 size，比如 3 转换为 32 字节

size = INDEX2SIZE(size);

//跳过用于 pool_header 的内存，并进行对齐

bp = (block *)pool + POOL_OVERHEAD;

//实际就是 pool->nextoffset = POOL_OVERHEAD+size+size

pool->nextoffset = POOL_OVERHEAD + (size << 1);
pool->maxnextoffset = POOL_SIZE - size;
pool->freeblock = bp + size;
*(block **)(pool->freeblock) = NULL;
return (void *)bp;
```

最后返回的 bp 就是指向从 pool 中取出的第一块 block 的指针，也就是说，pool 中第一块 block 已经被分配了，所以在 ref.count 中记录了当前已经被分配的 block 数量，这时为 1。特别需要注意的是，bp 返回的实际是一个地址，这个地址之后有将近 4k 的内存实际上都是可用的，

但是可以肯定申请内存的函数只会使用[bp, bp+size]这个区间的内存，这是由 size class index 可以保证的。好了，来看一看图 2 所示的一块经过改造后的 4k 内存。

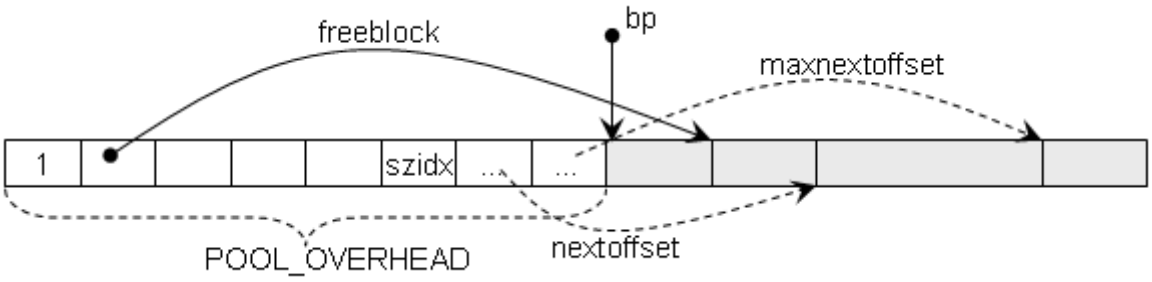

图 2 改造成 pool 后的 4k 内存

注意其中的实现箭头是指针，但是虚线箭头不是代表指针，是偏移位置的形象表示。在 nextoffset 和 maxnextoffset 中存储的是相对于 poo 头部的偏移位置。

在了解了初始化后的 pool 的样子之后，可以来看看 Python 在申请 block 时，pool_header 中的各个域是怎么变动的。假设我们现在开始连续申请 5 块 28 字节内存，由于 28 个字节对应的 size class index 为 3，所以实际上会申请 5 块 32 字节的内存。

```
[obmalloc.c]-[allocate block]
if (pool != pool->nextpool) {
        ++pool->ref.count;
        bp = pool->freeblock;
        ……
        if (pool->nextoffset <= pool->maxnextoffset) {
            //有足够的block空间
            pool->freeblock = (block *)pool + pool->nextoffset;
            pool->nextoffset += INDEX2SIZE(size);
            *(block **)(pool->freeblock) = NULL;
            return (void *)bp;
        }
    }
```

原来 freeblock 指向的是下一个可用的 block 的起始地址，这一点在图 2 中也可以看得出。当再次申请 32 字节的 block 时，只需返回 freeblock 指向的地址就可以了，很显然，这时

freeblock 需要向前进，指向下一个可用的 block。这时，nextoffset 现身了。

在 pool header 中，nextoffset 和 maxoffset 是两个用于对 pool 中的 block 集合进行迭代的变量：从初始化 pool 的结果以及图 2 中可以看到，它所指示的偏移位置正好指向了 freeblock 之后的下一个可用的 block 的地址。从这里分配 block 的动作也可以看到，在分配了 block 之后，freeblock 和 nextoffset 都会向前移动一个 block 的距离，如此反复，就可对所有的 block 进行一次遍历。而 maxnextoffset 指名了该 pool 中最后一个可用的 block 距 pool 开始位置的便移，它界定了 pool 的边界，当 nextoffset >= maxnextoffset 时，也就意味着已经遍历完了 pool 中所有的 block 了。

嗯，申请，前进，申请，前进，这个过程非常自然，也容易理解。但是且慢，这好像意味着一个 pool 中只能满足 POOL_SIZE/size 次对 block 的申请，这很难让人接受。如果这样不容易理解，我们来考虑一个形象的例子。现在我们已经进行了 5 次连续的 32 字节的内存分配，可以想见，pool 中 5 个连续的 block 都被分配出去了。过了一段时间，程序释放了其中第 2 和第 4 块 block，那么下一次再分配 32 字节的内存时，pool 提交的应该是第 2 块还是第 6 块 block 呢？很显然，为了 pool 的使用效率，最好再次分配自由的第 2 块 block。可以想见，一旦 Python 运转起来，内存的释放动作将会导致 pool 中出现大量的离散的自由 block，Python 必须建立一种机制，将这些离散的自由 block 组织起来，再次使用。这个机制就是所谓的自由 block 链表。这个链表的关键就着落在 pool_header 中的那个 freeblock 身上。

刚才我们就说了，当 pool 初始化完成之后，freeblock 指向了一个有效的地址，为下一个可以分配出去的 block 的地址，然而奇特的是，Python 在设置了 freeblock 之后，还设置了 *freeblock。这一个动作似乎非常诡异，然而我们马上就会看到，设置*freeblock 的动作正是建立离散自由 block 链表的关键所在。目前我们看到的 freeblock 只是在机械地前进前进，这是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 4k 内存上开始灵活地舞动。这个特殊的时刻就是一个 block 被释放的时刻。

```
[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)((uptr)(P) & ~(uptr)POOL_SIZE_MASK))

void PyObject_Free(void *p)
{
    poolp pool;
    block *lastfree;
    poolp next, prev;
    uint size;

    pool = POOL_ADDR(p);

    //判断 P 指向的 block 是否属于 pool
    if (Py_ADDRESS_IN_RANGE(p, pool)) {
        *(block **)p = lastfree = pool->freeblock; //[1]
        pool->freeblock = (block *)p;                  //[2]

        ……

    }
}
```

在释放 block 时， 神秘的 freeblock 惊鸿一现，覆盖在 freeblock 身上那层神秘的面纱就要揭开了。我们知道，这时 freeblock 虽然指向了一个有效的 pool 内地址，但是*freeblock 是为 NULL 的。假设这时 Python 释放的是 block A，在代码[1]之后，A 中第一个字节的值被设置为了当前 freeblock 的值，而在代码[2]之后 freeblock 的值被更新了，指向了 block A 的首地址。就是这短短的两步，一个 block 被插入到了离散自由 block 链表中。所以当第 2 块和第 4 块 block 都被释放之后，我们可以看到一个初具规模的离散自由 block 链表了，如图 3 所示。

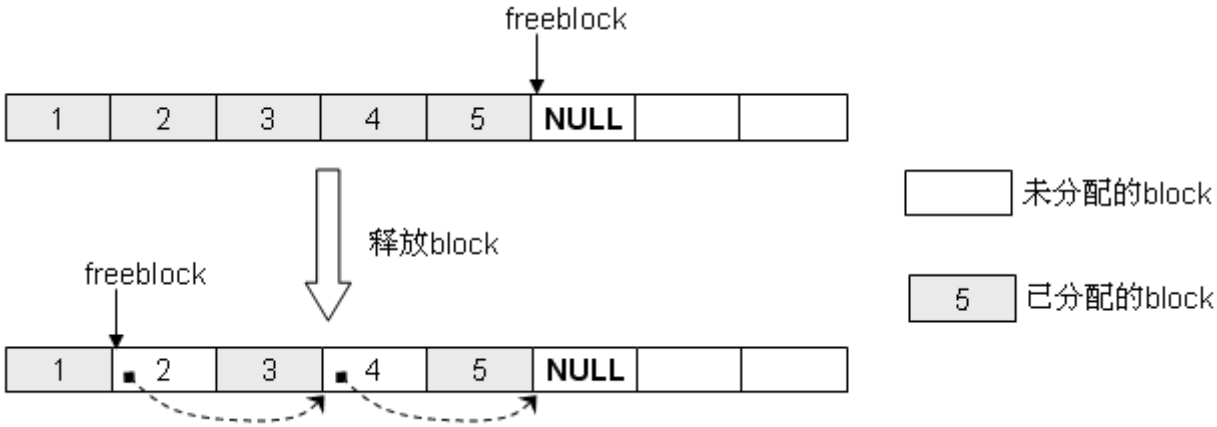

图 3 释放了 block 之后产生的自由 block 链表

到了这里，这条实现方式非常奇特的离散自由 block 链表被我们挖掘出来了，从 freeblock 开始，我们可以很容易地以 freeblock = *freeblock 的方式遍历这条链表，而当发现了*freeblock 为 NULL 时，则表明到达了该链表的尾部了。这条链表将影响到在本小节开始时我们剖析过的一般的 block 分配行为。

```
[obmalloc.c]-[allocate block]
      ……
      if (pool != pool->nextpool) {
          /*
           * There is a used pool for this size class.
           * Pick up the head block of its free list.
           */
          ++pool->ref.count;
          bp = pool->freeblock; //[1]
          if ((pool->freeblock = *(block **)bp) != NULL) { //[2]
              return (void *)bp;
          }
          if (pool->nextoffset <= pool->maxnextoffset) {
                ……
            }
            ……
        }
```

这里的代码[1]和代码[2]正是 Python 中 freeblock = *freeblock 的实现方式。当代码[2]处的判

断为真时，表明已经不存在离散自由 block 链表了，如果可能，则会继续分配 pool 的 nextoffset 指定的下一块 block。但是，如果连 nextoffset <= maxnextoffset 都不成立了呢？嗨，老兄，别忘了，我们现在谈论的仅仅是一个 pool，如果这个 pool 中的 block 被用光了，最简单的解决方案就是：我给你另一个 pool。这就意味着，在 Python 中，存在一个 pool 的集合。

2.3 arena

在 Python 中，多个 pool 聚合的结果就是一个 arena。上一节提到，pool 的大小的默认值为 4K，同样，每个 arena 的大小都有一个默认的值，在 Python2.5 中，这个值由名为 ARENA_SIZE 的符号控制，为 256K。那么很显然，一个 arena 中容纳的 pool 的个数就是 ARENA_SIZE / POOL_SIZE = 64 个。

```
[obmalloc.c]
#define ARENA_SIZE      (256 << 10) /* 256KB */
```

好了，我们现在来看一看 Python 中的 arena 到底是个什么东西。

```
[obmalloc.c]
typedef uchar block;

struct arena_object {
   uptr address;
   block* pool_address;
   uint nfreepools;
   uint ntotalpools;
   struct pool_header* freepools;

   struct arena_object* nextarena;
   struct arena_object* prevarena;
};
```

一个概念上的 arena 在 Python 源码中就对应 arena_object 结构体，确切地说，arena_object 仅仅是一个 arena 的一部分，就像 pool_header 只是 pool 的一部分一样。一个完整的 arena 包括一个 arena_object 和透过这个 arena_object 管理着的 pool 集合，同样，pool 的情况类似，一个

完整的 pool 包括一个 pool_header 和透过这个 pool_header 管理着的 block 集合。

2.3.1 “未使用”的 arena 和“可用”的 arena

在 arena_object 结构体的定义中，我们看到了 nextarea 和 prevarea 这两个东西，这似乎意味着在 Python 中会有一个多个 arena 构成的链表，这个链表的表头就是 arenas。呃，这种猜测只对了一半，实际上，在 Python 中，确实会存在多个 arena_object 构成的集合，但是这个集合并不构成链表，而是构成了一个 arena 的数组，数的组的首地址由 arenas 维护，这个数组就是 Python 中的通用小块内存的内存池；另一方面，nextarea 和 prevarea 也确实是用来连接 arena_object 组成链表的，既然多个 arena_object 已经通过数组组织起来了，为什么又要搞出一个链表来。乍一看，真的有点稀奇古怪。

我们曾说 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 管理的内存和 areana_object 管理的内存有一点细微的差别，pool_header 管理的内存与 pool_header 自身是一块连续的内存，而 areana_object 与其管理的内存则是分离的。这种差别如图 4 所示。

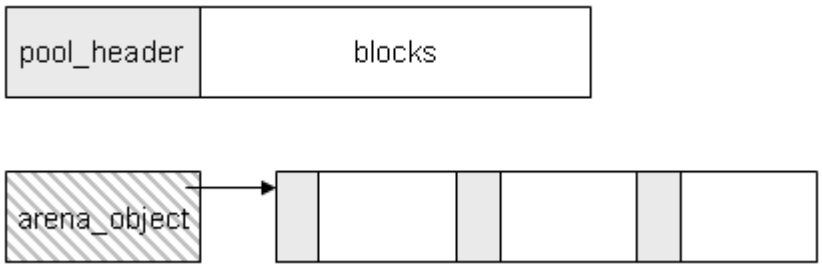

图 4 pool 和 arena 的内存布局区别

初看上去，这似乎没什么大不了的，不就一个是连着的，一个是分开的吗？但是这后面隐藏着这样一个事实：当 pool_header 被申请时，它所管理的 block 集合的内存一定也被申请了；但是当 aerna_object 被申请时，它所管理的 pool 集合的内存则没有被申请。换句话所，

arena_object 和 pool 集合在某一时刻需要建立联系。注意，这个建立联系的时刻是一个关键的时刻，Python 从这个时刻一刀切下，将一个 arena_object 切分为两种状态。

当一个 arena 的 area_object 没有与 pool 集合建立联系时，这时的 arena 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 的链表。“未使用”的 arena 的链表表头是 unused_arena_objects，arena 与 arena 之间通过 nextarena 连接，是一个单向链表；而“可用”的 arena 的链表表头是 usable_arena_objects，arena 与 arena 之间通过 nextarena 和 prevarena 连接，是一个双向链表。图 5 展示了某一时刻多个 arena 的一个可能状态。

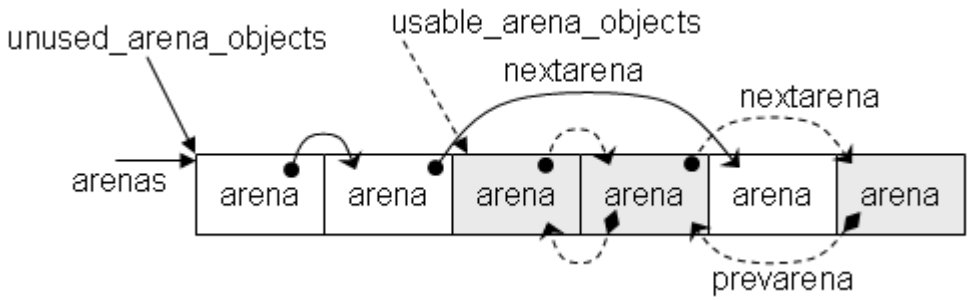

图 5 arena 集合在某时刻的可能状态

2.3.2 申请 arena

在运行期间，Python 使用 new_arena 来创建一个 arena，我们来看看 arena 创建时的情形。

```
[obmalloc.c]
//arenas管理着arena_object的集合
static struct arena_object* arenas = NULL;
//当前arenas中管理的arena_object的个数
static uint maxarenas = 0;
//"未使用的" arena_object链表
static struct arena_object* unused_arena_objects = NULL;
```

```
//"可用的" arena_object链表
static struct arena_object* usable_arenas = NULL;

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

static struct arena_object* new_arena(void)
{
   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_object列表
   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;
      if (numarenas <= maxarenas)
         return NULL;  //overflow (溢出)
      nbytes = numarenas * sizeof(*arenas);
      if (nbytes / sizeof(*arenas) != numarenas)
         return NULL;  //overflow
      arenaobj = (struct arena_object *)realloc(arenas, nbytes);
      if (arenaobj == NULL)
         return NULL;
      arenas = arenaobj;

      //[3]: 初始化新申请的arena_object，并将其放入unused_arena_objects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;
   }
```

```
    //[4]: 从unused_arena_objects链表中取出一个“未使用的”arena_object
    arenaobj = unused_arena_objects;
    unused_arena_objects = arenaobj->nextarena;
    assert(arenaobj->address == 0);

    //[5]: 申请arena_object管理的内存
    arenaobj->address = (uptr)malloc(ARENA_SIZE);
    ++narenas_currently_allocated;

    //[6]: 设置的pool相关信息
    arenaobj->freepools = NULL;
    arenaobj->pool_address = (block*)arenaobj->address;
    arenaobj->nfreepools = ARENA_SIZE / POOL_SIZE;
    //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;
}
```

在代码[1]处，Python 首先会检查当前 unused_arena_objects 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 unused_arena_objects 中还存在“未使用”的 arena，那么 Python 将直接开始代码[4]处的动作，从 unused_arena_objects 中抽取出一个 arena，接着调整 unused_arena_objects，与抽取出的 arena 彻底断绝一切联系。然后，在代码[5]处，Python 申请了一块大小为 ARENA_SIZE（256K）的内存，将申请的内存的地址赋给 arena 的 address。我们已经知道，arena 中维护的是 pool 的集合，这块 256K 的内存就是 pool 的容身之处，这时 arena_object 就和

pool 集合建立联系了，这个 arena 已经具备了成为“可用”内存的条件。到了这里，arena 和 unused_arena_objects 已经脱离了关系，就等着 usable_arena_object 这个组织的接收了，到底什么才能接收呢，别急，谜底一会儿揭晓。

随后，在代码[6]处，Python 设置了一些 arena 用于维护 pool 集合的信息，特别注意的是，在代码[6]的动作中，Python 将申请到的 256K 内存进行了处理，放弃了一些内存，而将可使用的内存边界(pool_address)调整到了与系统页对齐。代码[6]处将 freepools 设置为 NULL，基于前面我们对 pool 中 freeblock 的了解，这没什么大惊小怪的，看来要等到释放一个 pool 时，这个 freepools 才有用了。最后我们看到，pool 集合所占用的 256K 的内存在进行边界对齐后，实际上是交给 pool_address 来维护了。

回到 new_arena 中代码[1]处的判断，如果 unused_arena_objects 为 NULL 了，则表明目前系统中已经没有“未使用”的 arena 了，Python 将首先扩大系统的 arena 集合（小块内存内存池）。Python 在内部通过一个唤作 maxarenas 的变量维护了在 arenas 指向的数组中存储的 arena_object 的个数。在代码[2]处，Python 将待申请的 arena_object 的个数设置为当前 arena_object 个数（maxarenas）的 2 倍。当然，在首次初始化时，maxarenas 为 0，这时，Python 将新的 maxarenas 初始化为 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 在代码[3]处通过 realloc 扩大 arenas 指向的内存，并对新申请的 arena_object 进行设置，特别要提到的是那个貌似毫不起眼的 address，代码[3]处将新申请的 arena 的 address 一律设置为 0，实际上，这是一个标识一个 arena 是出于“未使用”状态还是“可用”状态的重要标记。看看代码[6]处，一旦 arena 的 arena_object 与 pool 集合建立了联系，这个 address 就变成了非 0。当然，别忘了我们为什么会来到代码[3]这里的，咱们可不是饭后随便溜达到这里的，是因为那个重要的 unused_arena_objects 变为 NULL 了，对喽，所以最后还设置了

unused_arena_objects。这样一来，系统中又有了“未使用”的 arena 了，接下来，Python 就将进入代码[4]处对一个 arena 的初始化了。

2.4 内存池

2.4.1 可用 pool 缓冲池——usedpools

在 Python2.5 中，Python 内部默认的小块内存与大块内存的分界点定在 256 个字节，这个分界点由前面我们看到的名为 SMALL_REQUEST_THRESHOLD 的符号控制。也就是说，当申请的内存小于 256 字节时，PyObject_Malloc 会在内存池中申请内存；当申请的内存大于 256 字节时，PyObject_Malloc 的行为将蜕化为 malloc 的行为。当然，通过修改 Python 源代码，我们可以改变这个默认值，从而改变 Python 的默认内存管理行为。

当 Python 申请小于 256 字节的内存时，Python 会使用 arenas 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？这个决策取决于用户，Python 提供了一个编译符号，用于控制是否限制这个内存池的大小。

当 Python 在 WITH_MEMORY_LIMITS 编译符号打开的背景下进行编译时，Python 内部另一个符号会被激活，这个名为 SMALL_MEMORY_LIMIT 的符号限制了整个内存池的大小，同时，也就限制了可以创建的 arena 的个数。在默认情况下，不论是 win32 平台，还是 unix 平台，这个编译符号都是没有打开的，所以通常 Python 都没有对小块内存的内存池的大小做任何的限制。

```
[obmalloc.c]
#ifdef WITH_MEMORY_LIMITS
#ifndef SMALL_MEMORY_LIMIT
```

```
#define SMALL_MEMORY_LIMIT  (64 * 1024 * 1024)  /* 64 MB -- more? */
#endif
#endif

#ifdef WITH_MEMORY_LIMITS
#define MAX_ARENAS      (SMALL_MEMORY_LIMIT / ARENA_SIZE)
#endif
```

尽管我们在前面花费了大量篇幅介绍 arena，同时也看到 arena 是 Python 小块内存池的最上层结构，所有 arena 的集合实际就是小块内存池。然而在实际的使用中，Python 并不直接与 arenas 和 arena 打交道。当 Python 申请内存时，最基本的操作单元并不是 arena，而是 pool。唉，绕来绕去的，我都觉得快晕了，没办法，兄弟，挺住，后面还有很多呢若

举个例子，当我们申请一个 28 字节的内存时，Python 内部会在内存池中寻找一块能满足需求的 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 自身的属性决定的。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，还记得 pool_head 中的那个 szidx 么？而 arena 是没有 size 概念的内存管理抽象体，这就意味着，同一个 arena，在某个时刻，其内的 pool 集合可能都是管理的 32 字节的 block；而到了另一时刻，由于系统需要，这个 arena 可能被重新划分，其中的 pool 集合可能改为管理 64 字节的 block 了，甚至 pool 集合中一半管理 32 字节，一半管理 64 字节。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。

内存池中的 pool，不仅是一个有 size 概念的内存管理抽象体，而且，更进一步的，它还是一个有状态的内存管理抽象体。一个 pool 在 Python 运行的任何一个时刻，总是处于一下三种状态的一种：

1、used 状态：pool 中至少有一个 block 已经被使用，并且至少有一个 block 还未被使用。这种状态的 pool 受控于 Python 内部维护的 usedpools 数组。

2、full 状态：pool 中所有的 block 都已经被使用，这种状态的 pool 在 arena 中，但不再 arena 的 freepools 链表中。

3、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 中的 freepools。

图 6 给出了一个 arena 中包含三种状态的 pool 的集合的一个可能状态。

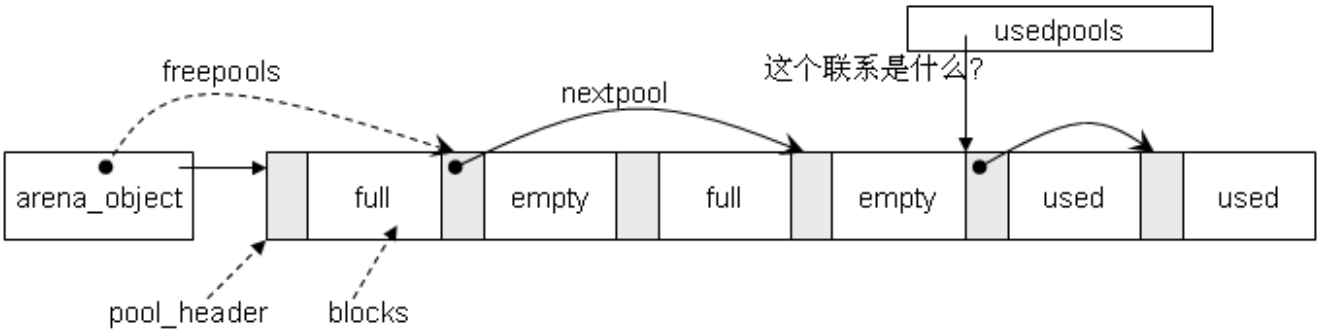

图 6 某个时刻 aerna 中 pool 集合的可能状态

请注意，arena 中处于 full 状态的 pool 是各自独立的，并没有像其他的 pool 一样会链接成链表。

在图 6 中，我们看到所有处于 used 状态的 pool 都被置于 usedpools 的控制之下。Python 内部维护的 usedpools 数组是一个非常巧妙的实现，维护着所有的处于 used 状态的 pool。当申请内存时，Python 就会通过 usedpools 寻找到一块可用的(处于 used 状态的)pool，从中分配一个 block。从这个简要的叙述中，我们已经可以看到，一定有一个与 usedpools 相关联的机制，完成从申请的内存的大小到 size class index 之间的转换，否则 Python 也就无法寻找到最合适的 pool 了。这种机制与 usedpools 的结构有密切的关系，我们来看一看 usedpools 的结构。

```
[obmalloc.c]
typedef uchar block;

#define PTA(x)   ((poolp )((uchar *)&(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)

    ......

#endif
}
```

其中的 NB_SMALL_SIZE_CLASSES 指明了在当前的配置之下，一共有多少个 size class。

```
[obmalloc.c]
#define NB_SMALL_SIZE_CLASSES    (SMALL_REQUEST_THRESHOLD / ALIGNMENT)
```

这个数组的定义有些怪异，别急，待我们用一幅图来展示这个怪异的 usedpools 数组，如图 7 所示。

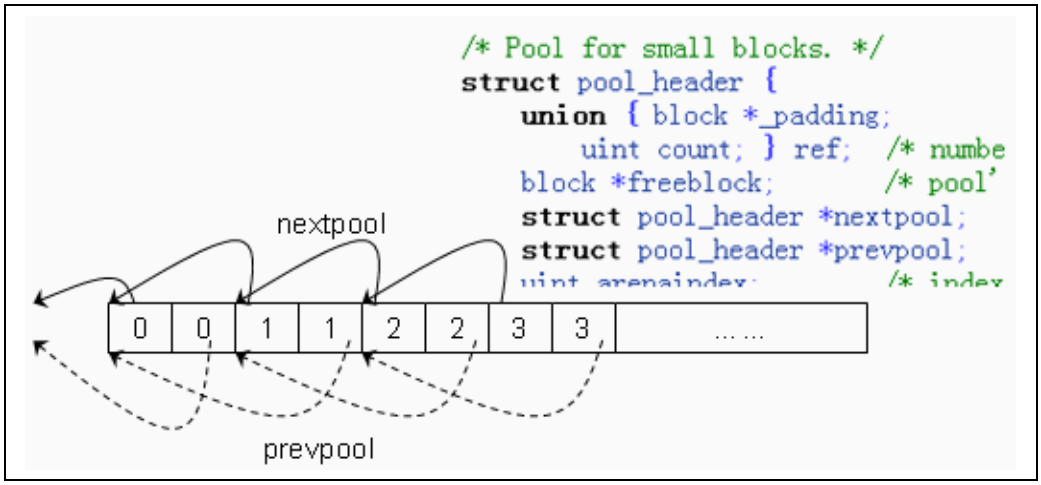

图 7 usedpools 数组

这样看上去似乎仍然摸不着头脑，别急，我们来考虑一下当申请 28 个字节时的情形。前面我们提到，Python 会首先获得 size class index，通过 size = (uint)(nbytes - 1) >> ALIGNMENT_SHIFT;得到 size class index 为 3。在 usedpools 中，寻找第 3+3=6 个元素，发现 usedpools[6]的值是指向 usedpools[4]的地址。有些迷惑了，对吧？好了，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呢？是从 usedpools[6]（即 usedpools+4）开始向后偏移 8 个字节（一个 ref 的大小加上一个 freeblock 的大小）后的内存，

不正是 usedpools[6]的地址（即 usedpools+6）吗？这是 Python 内部使用的一个 trick。

想象一下，当我们手中有一个 size class 为 32 字节的 pool，想要将其放入这个 usedpools 中时，需要怎么做呢？从上面的描述，可以看到，只需要进行 usedpools[i+i]->nextpool = pool 即可，其中 i 为 size class index，对应于 32 字节，这个 i 为 3。当下次需要访问 size class 为 32 字节（size class index 为 3）的 pool 时，只需要简单地访问 usedpool[3+3]就可以得到了。Python 正是使用这个 usedpools 快速地从众多的 pool 中快速地寻找到一个最适合当前内存需求的 pool，从中分配一块 block。

在我们即将看到的 PyObject_Malloc 代码中，Python 利用了 usedpools 的巧妙结构，通过简单的判断来发现与某个 class size index 对应的 pool 是否在 usedpools 中存在。下面是 PyObject_Malloc 中进行这个判断的代码。

```
[obmalloc.c]
void* PyObject_Malloc(size_t nbytes)
{
    block *bp;
    poolp pool;
    poolp next;
    uint size;
    if ((nbytes - 1) < SMALL_REQUEST_THRESHOLD) {
        LOCK();

        //获得size class index

        size = (uint )(nbytes - 1) >> ALIGNMENT_SHIFT;
        pool = usedpools[size + size];

        //usedpools中有可用的pool

        if (pool != pool->nextpool) {

            ……//usedpools中有可用的pool

        }


        …… //usedpools中无可用pool，尝试获取empty状态pool
}
```

在图 8 中，还是以申请大小为 28 字节的内存块为例，展示了 pool != pool->nextpool 为什么能够工作的原因。我们在 Python 的源代码中添加了代码，使得 Python 在第一次申请 size class index 为 3 的内存块时发生中断，以便形象地观察这时 usedpools 的内存布局。图 7 左侧粗体显示的地址即是 pool 和 pool->nextpool 的值。

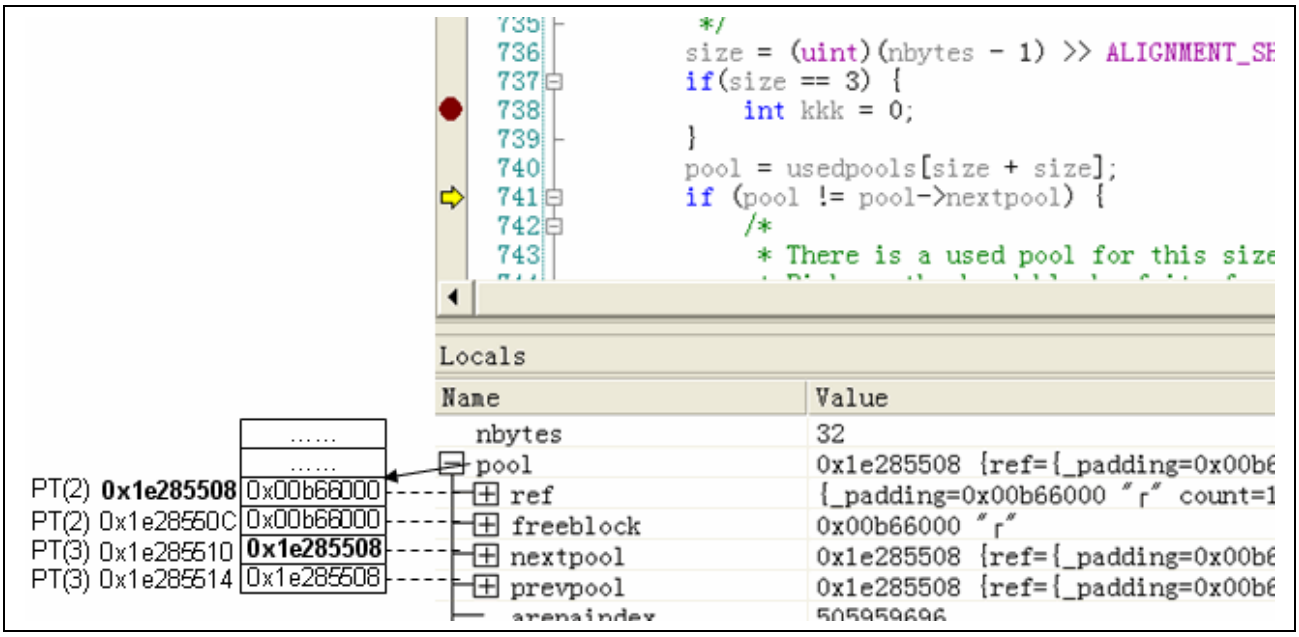

图 8 第一次申请 28 字节的内存块时 usedpools 中相关的内存布局

2.4.2 Pool 的初始化

当 Python 启动之后，在 usedpools 这个小块空间内存池中，并不存在任何可用的内存，准确地说，不存在任何可用的 pool。在这里，Python 采用了延迟分配的策略，即当我们确实开始申请小块内存时，Python 才开始建立这个内存池。正如前面所提到的，当申请 28 个字节的内存时，Python 将实际上申请 32 字节的内存。Python 会首先会根据 32 字节对应的 class size index（3）在 usedpools 中对应的位置查找，如果发现在对应的位置后并没有链接任何可用的 pool，首先，Python 会从 usable_arenas 链表中的第一个“可用的”arena 中获得一个 pool。需要

特别注意的是，当前获得的 arena 中包含的这些 pools 可能并不属于同一个 class size index。

考虑一下这样的情况，当申请 32 字节内存时，从“可用的”arena 中取出其中一个 pool 用作 32 字节的 pool。当下一次内存分配请求分配 64 字节的内存时，Python 可以直接使用当前“可用的”arena 的另一个 pool 即可，这正如我们前面所说，arena 没有 size class 的属性，而 pool 才有。

```
[malloc.c]
void * PyObject_Malloc(size_t nbytes)
{
   block *bp;
   poolp pool;
   poolp next;
   uint size;

   if ((nbytes - 1) < SMALL_REQUEST_THRESHOLD) {
      LOCK();
      size = (uint)(nbytes - 1) >> ALIGNMENT_SHIFT;
      pool = usedpools[size + size];
      if (pool != pool->nextpool) {

         …… //usedpools中有可用的pool

      }

      //usedpools中无可用pool，尝试获取empty状态pool

      //[1]: 如果usable_arenas链表为空，则创建链表

      if (usable_arenas == NULL) {

         //申请新的arena_object，并放入usable_arenas链表

         usable_arenas = new_arena();
         usable_arenas->nextarena = usable_arenas->prevarena = NULL;
      }


      //[2]: 从usable_arenas链表中第一个arena的freepools中抽取一个可用的pool

      pool = usable_arenas->freepools;
      if (pool != NULL) {
         usable_arenas->freepools = pool->nextpool;
```

```
        //[3]: 调整usable_arenas链表中第一个arena中的可用pool数量

         //如果调整后数量为0，则将该arena从usable_arenas链表中摘除
        --usable_arenas->nfreepools;
        if (usable_arenas->nfreepools == 0) {
            usable_arenas = usable_arenas->nextarena;
            if (usable_arenas != NULL) {
                usable_arenas->prevarena = NULL;
            }
        }
    init pool:

         ......
}
```

可以看到，如果开始时 usable_arenas 为空，那么 Python 会在代码[1]处通过 new_arena 申请一个 arena，开始构建 usable_arenas 链表。在代码[2]处，Python 会尝试从 usable_arenas 链表中的第一个 arena 所维护的 pool 集合中取出一个可用的 pool。如果成功地取出了这个 pool，那么在代码[3]处，Python 会进行一些维护信息的更新工作，甚至在当前 arena 中可用的 pool 已经用完之后，会将该 arena 从 usable_arenas 链表中摘除。那位客官说了，你不能摘下来就一了百了啊，摘下来之后这块内存不就失去控制了？别急，别忘了还有个 arenas 这个数组啊，孙猴子再厉害，也逃不出如来佛的魔爪的敢

需要注意的是，在代码[2]处的判断表明获得 pool 有可能失败，那么在什么情况下，一个 arena 中的 freepools 会是 NULL 呢，呃，回忆一下前面对 new_arena 的剖析，没错，在那里，new_arena 准备返回的 arena 的 freepools 就是为 NULL 的。那么在代码[2]处发现 pool 为 NULL 时 Python 会怎么处理呢，我们把这个话题放到后面。

2.4.2.1初始化之一

好了，现在我们手里有了一块用于 32 字节内存分配的 pool，为了以后内存分配的效率，

我们需要将这个 pool 放入到 usedpools 中。这一步，叫做 init pool。

```
[obmalloc.c]
#define ROUNDUP(x)              (((x) + ALIGNMENT_MASK) & ~ALIGNMENT_MASK)
#define POOL_OVERHEAD           ROUNDUP(sizeof(struct pool_header))
void * PyObject_Malloc(size_t nbytes) {

……

init_pool:

          //[1]: 将pool放入usedpools中

          next = usedpools[size + size]; /* == prev */
          pool->nextpool = next;
          pool->prevpool = next;
          next->nextpool = pool;
          next->prevpool = pool;
          pool->ref.count = 1;

          //[2]：pool在之前就具有正确的size结构，直接返回pool中的一个block

          if (pool->szidx == size) {
              bp = pool->freeblock;
              pool->freeblock = *(block **)bp;
              UNLOCK();
              return (void *)bp;
          }


           //[3]： 初始化pool header，将freeblock指向第二个block，返回第一个block

          pool->szidx = size;
          size = INDEX2SIZE(size);
          bp = (block *)pool + POOL_OVERHEAD;
          pool->nextoffset = POOL_OVERHEAD + (size << 1);
          pool->maxnextoffset = POOL_SIZE - size;
          pool->freeblock = bp + size;
          *(block **)(pool->freeblock) = NULL;
          UNLOCK();
          return (void *)bp;

……

}
```

在代码[1]处，Python 将得到的 pool 放入了 usedpools 中。当一个从未被使用的 pool（也就是

由 new_arena 返回的 aerna 中的 pool）被链入到 usedpools 中时，从后面的分析可以看到，其 szidx 是被设为了 0xFFFF 的，所以这时 init pool 的动作会执行代码[2]，而不会执行代码[1]。只有当一个 pool 从 empty 状态重新转为 used 状态之后，由于这时 szidx 还是其转为 empty 状态之前的 szidx，所以才有可能执行代码[1]。

在什么样的情况下才会发生一个 pool 从 empty 状态转换为 used 状态呢？假设申请的内存的 size class index 为 i，且 usedpools[i+i]处没有处于 used 状态的 pool，同时在 Python 维护的全局变量 freepools 中还有处于 empty 的 pool，那么位于 freepools 所维护的 pool 链表头部的 pool 将被取出来，放入 usedpools 中，并从其内部分配一块 block，同时，这个 pool 也就从 empty 状态转换到了 used 状态。下面我们看一看这个行为在代码中是如何体现的。

```
[obmalloc.c]
      ……
       pool = usable_arenas->freepools;
       if (pool != NULL) {
           usable_arenas->freepools = pool->nextpool;
           …… //调整usable_arenas->nfreepools和usable_arenas自身
           [init_pool]
       }
```

其中[init_pool]处引用的是前面剖析的关于 init pool 的代码。需要注意的是，虽然一个 pool 从 empty 状态转为 used 状态时，携带了有效的 szidx 信息，但是这只是上一次 pool 被使用时的信息，只有当当前内存分配动作对应的 size class index 与这个 szidx 完全一致时，才会执行代码[1]，否则，Python 还是会照常进行代码[2]，以重新对 pool 进行初始化。

2.4.2.2初始化之二

我们现在可以来看看，当 PyObject_Malloc 从 new_arena 中得到一个新的 arena 后，怎么样来初始化其中的 pool 集合，并最终完成 PyObject_Malloc 函数的分配一个 block 这个终极任务

的。

```
[obmalloc.c]
#define DUMMY_SIZE_IDX    0xffff /* size class of newly cached pools */
void * PyObject_Malloc(size_t nbytes)
{
   block *bp;
   poolp pool;
   poolp next;
   uint size;

    ......

   //从arena中取出一个新的pool

   pool = (poolp)usable_arenas->pool_address;
   pool->arenaindex = usable_arenas - arenas;
   pool->szidx = DUMMY_SIZE_IDX;
   usable_arenas->pool_address += POOL_SIZE;
   --usable_arenas->nfreepools;

   if (usable_arenas->nfreepools == 0) {
       /* Unlink the arena:  it is completely allocated. */
       usable_arenas = usable_arenas->nextarena;
       if (usable_arenas != NULL) {
           usable_arenas->prevarena = NULL;
       }
   }
   goto init_pool;

     ......
}
```

Python 首先在代码[1]处从新申请的 arena 中取出一个崭新的 pool，然后设置 pool 中的 arenaindex，这个 index 实际上就是 pool 所在的 arena 位于 arenas 所指的数组中的序号，这个东西有什么用呢，用处大着呢。它虽然不能医治跌打损伤，却能用来判断一个 block 是否在某个 pool 中，还记得我们在考察 pool 管理 block 集合时看到的那个引起 freeblock 舞动的 PyObject_Free 吗，里面，就有用到这个 arenaindex。

```
 [obmalloc.c]
```

```
//P 为指向一个 block 的指针，pool 为指向一个 pool 的指针
int Py_ADDRESS_IN_RANGE(void *P, poolp pool)
{
   return pool->arenaindex < maxarenas &&
          (uptr)P - arenas[pool->arenaindex].address < (uptr)ARENA_SIZE &&
          arenas[pool->arenaindex].address != 0;
}
```

在实际发布的 Python2.5 中， PyObject_Free 里实际上使用的是宏版本的 Py_ADDRESS_IN_RANGE，而并非这里给出的函数版本的 Py_ADDRESS_IN_RANGE，但是它们的代码都是一样的。

随后 Python 将新得到的 pool 的 szidx 设置为 0xffff，以表示这家伙以前从来没管理过 block 集合。接着，Python 还调整了刚获得的 arena 中的 pools 集合，甚至可能调整 usable_arenas。

在做完这些之后，Python 会通过 goto 直接跳到 init pool 的地方，完成将 pool 放入 usedpools 中。

无论什么开源的项目，内存管理都是最繁琐，最能体现“细节是魔鬼”的地方，在申请内存的过程中，还有一些细节这里就不再一一涉及了，不过最后我们还是要给出 PyObject_Malloc 的总体的结构，同时，强烈建议您打开代码阅读的工具，一头扎进 PyObject_Malloc 的细节中。

```
[obmalloc.c]
void * PyObject_Malloc(size_t nbytes)
{
   block *bp;
   poolp pool;
   poolp next;
   uint size;

   //如果申请的内存小于 SMALL_REQUEST_THRESHOLD，使用 Python 的小块内存的内存池

   //否则，转向 malloc
   if ((nbytes - 1) < SMALL_REQUEST_THRESHOLD) {
```

```
    //根据申请内存的大小获得对应的 size class index
    size = (uint)(nbytes - 1) >> ALIGNMENT_SHIFT;
    pool = usedpools[size + size];

    //如果 usedpools 中可用的 pool，使用这个 pool 来分配 block
    if (pool != pool->nextpool) {

        ......//在 pool 中分配 block

        //分配结束后，如果 pool 中的 block 都被分配了，将 pool 从 usedpools 中摘除
        next = pool->nextpool;
        pool = pool->prevpool;
        next->prevpool = pool;
        pool->nextpool = next;
        return (void *)bp;
    }


    //usedpools 中没有可用的 pool，从 usable_arenas 中获取 pool
    if (usable_arenas == NULL) {

        //usable_arenas 中没有就“可用”的 arena，开始申请 arena
        usable_arenas = new_arena();
        usable_arenas->nextarena = usable_arenas->prevarena = NULL;
    }


    //从 usable_arenas 的第一个 arena 中获取一个 pool
    pool = usable_arenas->freepools;
    if (pool != NULL) {
init pool:

         //获取 pool 成功，进行 init pool 的动作，将 pool 放入 used_pools 中，

         //并返回分配得到的 block

        ......
    }


  //获取 pool 失败，对 arena 中的 pool 集合进行初始化，

  //然后转入 goto 到 init pool 的动作处，初始化一个特定的 pool
```

```
    ......
       goto init_pool;
    }

redirect:

    //如果申请的内存不小于 SMALL_REQUEST_THRESHOLD，使用 malloc
    if (nbytes == 0)
        nbytes = 1;
    return (void *)malloc(nbytes);
}
```

2.4.3 block 的释放

考察完了对 block 的分配，是时候来看看对 block 的释放了。对 block 的释放实际上就是将一块 block 归还给 pool，我们已经知道，pool 可能有 3 种状态，在分别处于 3 种状态，它们各自的位置是不同的。

当我们释放一个 block 后，可能会引起 pool 的状态的转变，这种转变可能分为两种情况：

1、used 状态转变为 empty 状态

2、full 状态转变为 used 状态

当然，更多的情况是 pool 中尽管收回了一个 block，但是它仍然处于 used 状态，这是最简单的情况，我们从这个最简单的情况说起。

```
[obmalloc.c]
void PyObject_Free(void *p)
{
    poolp pool;
    block *lastfree;
    poolp next, prev;
    uint size;

    pool = POOL_ADDR(p);
    if (Py_ADDRESS_IN_RANGE(p, pool)) {
```

```
        //设置离散自由block链表
        *(block **)p = lastfree = pool->freeblock;
        pool->freeblock = (block *)p;
        if (lastfree) { //lastfree有效，表明当前pool不是处于full状态
            if (--pool->ref.count != 0) { //pool不需要转换为empty状态
                return;
            }
            ......
        }
        ......
    }

    //待释放的内存在PyObject_Malloc中是通过malloc获得的
    //所以要归还给系统
    free(p);
}
```

在 pool 的状态保持 used 状态这种情况下，Python 仅仅将 block 重新放入到自由 block 链表中，并调整了 pool 中的 ref.count 这个引用计数，确实非常简单。

如果释放 block 之前，block 所属的 pool 处于 full 状态呢？这种情况也比较简单，仅仅是将 pool 重新链回到 usedpools 中即可，看下面的代码。

```
[obmalloc.c]
void PyObject_Free(void *p)
{
    poolp pool;
    block *lastfree;
    poolp next, prev;
    uint size;

    pool = POOL_ADDR(p);
    if (Py_ADDRESS_IN_RANGE(p, pool)) {
        ....
```

```
        //当前pool处于full状态，在释放一块block后，需将其转换为used状态，并重新//链
        入usedpools的头部
        --pool->ref.count;
        size = pool->szidx;
        next = usedpools[size + size];
        prev = next->prevpool;
        /* insert pool before next:   prev <-> pool <-> next */
        pool->nextpool = next;
        pool->prevpool = prev;
        next->prevpool = pool;
        prev->nextpool = pool;
        return;
    }

    ......
}
```

最复杂的情况发生在 pool 在收回 block 前后状态从 used 状态转为 empty 状态的情形下，我们来看看 Python 的处理。首先 Python 要做的肯定是将 empty 状态的 pool 链入到 freepools 中去。

```
[obmalloc.c]
void PyObject_Free(void *p)
{
    poolp pool;
    block *lastfree;
    poolp next, prev;
    uint size;

    pool = POOL_ADDR(p);
    if (Py_ADDRESS_IN_RANGE(p, pool)) {
        *(block **)p = lastfree = pool->freeblock;
        pool->freeblock = (block *)p;
        if (lastfree) {
            struct arena_object* ao;
            uint nf;  //ao->nfreepools
            if (--pool->ref.count != 0) {
                return;
            }

            //[1]: 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;

            ......

        }

        ......

    }

    ......

}
```

代码[1]完成了将 empty 状态的 pool 放入 freepools 维护的链表中的工作，似乎这样就可以了，一切都能够正常运转了，所有的内存始终都在掌握之中。

确实，在 Python2.5 之前，包括 2.4，Python 就是这么做的，但是这样做隐藏着一个类似于内存泄漏的问题。很多人也都意识到这个问题，但都没有太大的动力去修改，因为这种情况只在极少数情况下会发生。

这个问题就是：Python 的 arena 从来不释放 pool。这个问题为什么会引起类似于内存泄漏的现象呢。考虑这样一种情形，申请 10*1024*1024 个 16 字节的小内存，这就意味着必须使用 160M 的内存，由于 Python 没有默认将前面提到的限制内存池的 WITH_MEMORY_LIMITS 编译符号打开，所以 Python 会完全使用 arena 来满足你的需求，这都没有问题，关键的问题在于过了一段时间，你将所有这些 16 字节的内存都释放了，这些内存都回到 arena 的控制中，似乎也没有问题。但是问题恰恰就在这时出现了。因为 arena 始终不会释放它维护的 pool 集合，所以这 160M 的内存始终被 Python 占用，如果以后程序运行中再也不需要 160M 如此巨大的内存，这点内存岂不是就浪费了?

当然，这种情形必须在大量持续申请小内存对象时才会出现，平时大家几乎不会碰到这种情况，所以这个问题也就一直留在了 Python 中，但是在 2004 年的时候，一个叫 Evan Jones 的

老兄不能忍受下去了。他在对一些巨大的图做某种算法操作时必须持续申请大量小块内存，这导致 Python 占用的内存冲上 1G 后就再也掉不下来。想一想，确实相当痛苦，这位老兄一番探索，终于发现问题的根源在 arena 这里，于是一鼓作气，搞出了一套解决方案，并在 PyCon 2005 上做了个报告，引起了强烈的反响。但是 Python 的核心开发团队一直没有将这个 patch 并入到 Python 代码中，一直到 2006 年，才由 Tim Peters（这位老兄的名头在 Python 社区也是响当当的，在 Python 的交互环境下键入 import this，看到这位老兄的名号了吧，the zen of python 的创造者，当然，他在 Python 社区的地位可不是靠这几句广告语获得的）将这个 patch 整理，并入到了 Python 代码中，加入了这个 patch 的 Python 就是我们花了这么多精力剖析的 Python2.5。

在 Python2.4 中，实际上对 arena 是没有区分“未使用”和“可用”两种状态的，到了 Python2.5 中，arena 可以将自己维护的 pool 集合释放，返回给操作系统，从而必须从“可用”状态转为“未使用”状态，这也是必须要两种状态的原因。在前面那段代码的代码[1]之后，当 Python 处理完 pool 之后，就要开始处理 arena 了。

对 arena 的处理实际上分为了四种情况：

1、如果 arena 中所有的 pool 都是 empty 的，释放 pool 集合占用的内存。

```
[obmalloc.c]
void PyObject_Free(void *p)
{
    poolp pool;
    block *lastfree;
    poolp next, prev;
    uint size;

    pool = POOL_ADDR(p);
    struct arena_object* ao;
    uint nf;  //ao->nfreepools

    ......
```

```
    //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;
        }
        else {
            ao->prevarena->nextarena = ao->nextarena;
        }

        if (ao->nextarena != NULL) {
            ao->nextarena->prevarena = ao->prevarena;
        }

        //调整 unused_arena_objects 链表
        ao->nextarena = unused_arena_objects;
        unused_arena_objects = ao;

        //释放内存
        free((void *)ao->address);

        //设置 address，将 arena 的状态转为“未使用”
        ao->address = 0;
        --narenas_currently_allocated;
    }

    ......
}
```

可以看见，除了将 arena 维护的 pools 的内存归还给系统之外，Python 还调整了 usable_arenas 和 unused_arena_object 链表，将 arena 的状态转到了“未使用”状态，以及一些其他的维护工作。

2、如果之前 arena 中没有了 empty 的 pool，那么在 usable_arenas 链表中就找不到该 arena，

由于现在 arena 中有了一个 pool，所以需要将这个 aerna 链入到 usable_arenas 链表的表头

3、若 arena 中的 empty 的 pool 个数为 n，则从 usable_arenas 开始寻找 arena 可以插入的位置，将 arena 插入到 usable_arenas。这个操作的原因是由于 usable_arenas 实际上是一个有序的链表，从表头开始往后，每一个 arena 中的 empty 的 pool 的个数，即 nfreepools，都不能大于前面的 arena，也不能小于前面的 arena。保持这种有序性的原因是因为分配 block 时，是从 usable_arenas 的表头开始寻找可用的 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系统。

4、其他情况，不进行任何对 arena 的处理。

后面三种情况的代码这里就不一一列出了，建议读者自行到 Python 源码中去探索一番。

2.4.4 内存池全景

前面我们已经提到了，对于一个用 C 开发的庞大的软件，其中的内存管理可能是最复杂最繁琐的部分了，这里我们看到了对不同尺度内存的不同的抽象，看到了这些抽象在各种情况下组成的各式各样的链表，非常的复杂。但是，我们还是有可能从一个整体的尺度上把握整个内存池。这就是下面的图 9 希望完成的目标。尽管各种不同的链表变幻无常，我们只需记住，所有的内存都在 arenas 的掌控之中。

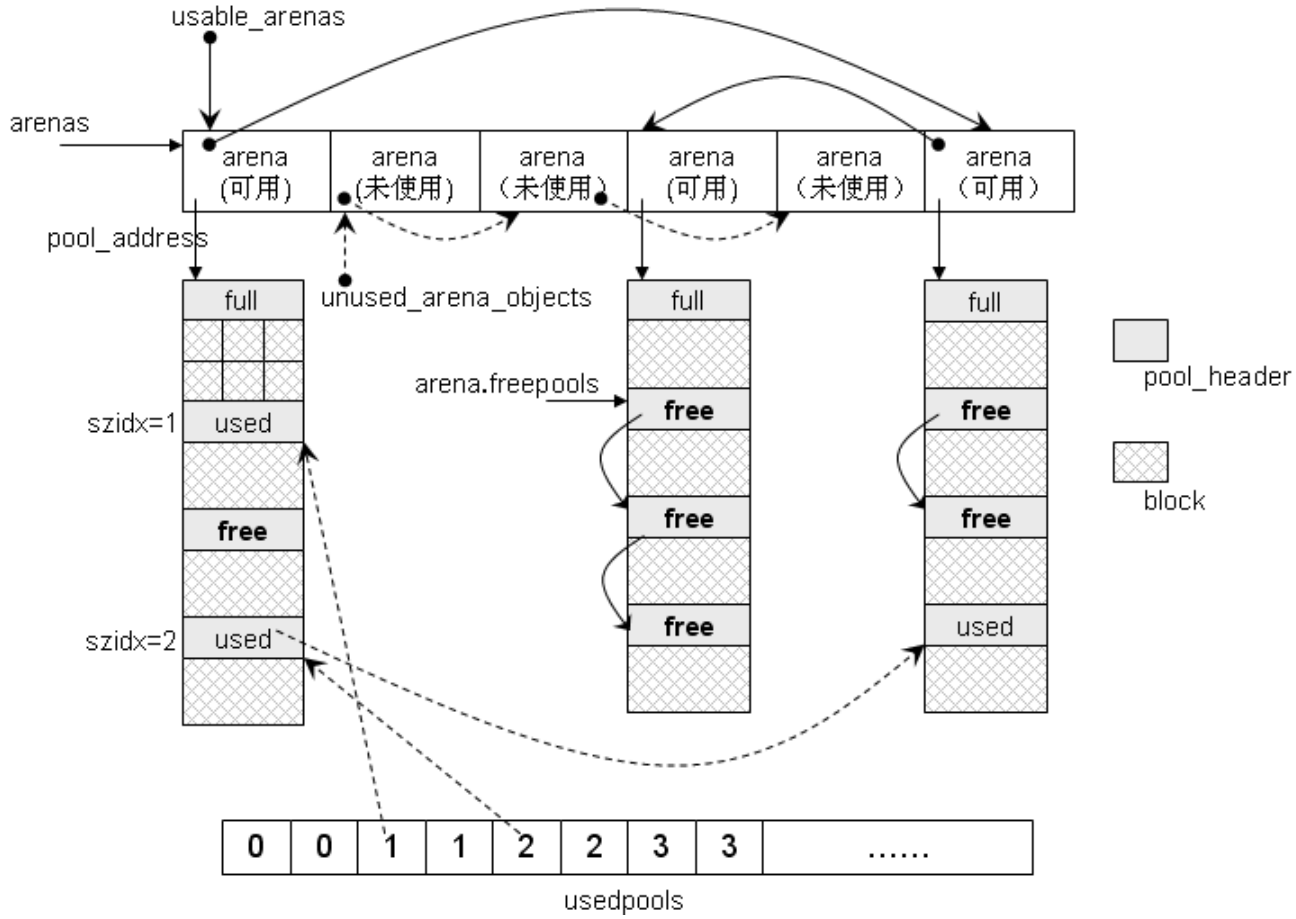

图 9 Python 的小块内存的内存池全景

3 循环引用的垃圾收集

3.1 引用计数与垃圾收集

尽管学术界对于垃圾收集技术的研究早在上个世纪 60 年代左右就拉开了帷幕，然而受限于当时的软硬件环境，垃圾收集还仅仅是一种看上去很美的技术。而挽起裤管，亲自下田，管理一个又一个的字节，不仅是某些应用所必需的，更成为了程序员自身技术水平的象征。然而与手动管理内存相伴随的 bug 也因此绵延几十年，无法根绝。直到上世纪 90 年代初，垃圾收集机制才随着 Java 的兴起，开始逐渐为工业界所接受。时至今日，随着软硬件环境的发展，垃

圾收集几乎已经成了现代主流开发语言不可或缺的特性，Java，C#，甚至以替代 C++为目标的 D 语言，都在语言层面引入了垃圾收集机制。

Python 同样也在语言层实现了内存的动态管理，从而将开发人员从管理内存的噩梦中解放出来。然而 Python 中的动态内存管理与 Java，C#有着很大的不同，在 Python 中，大多数对象的生命周期都是通过对象的引用计数来管理的，这一点，在本书前面的分析中，我们已经多次提到。从广义上来说，引用计数也是一种垃圾收集机制，而且也是一种最直观，最简单的垃圾收集技术。虽然引用计数必须在每次分配和释放内存的时候加入管理引用计数的动作，然而与其他主流的垃圾收集技术相比，引用计数方法有一个最大的优点，即实时性，任何内存，一旦没有指向它的引用，就会立即被回收。而其它的垃圾收集技术必须在某种特殊条件下（比如内存分配失败）才能进行无效内存的回收。

引用计数机制所带来的维护引用计数的额外操作与 Python 运行中所进行的内存分配和释放，引用赋值的次数是成正比的，这一点，相对于主流的垃圾回收技术，比如标记—清除（Mark—Sweep）、停止—复制（Stop—Copy）等方法相比，是一个弱点，因为这些技术所带来的额外操作基本上只与待回收的内存数量有关。为了与引用计数机制搭配，在内存的分配和释放上获得最高的效率，Python 因此设计了大量的内存池机制，在第 2 节中我们就看到了小块内存的内存池。而在之前对 Python 对象机制的剖析中，我们看到了对于 PyIntObject，PyStringObject，PyDictObject, PyListObject 等等都有与各种对象相关的内存池机制。这些大量使用的面向特定对象的对象内存池机制正是为了竭力弥补引用计数机制的软肋。

如果说执行效率还仅仅是引用计数机制的一个软肋的话，那么很不幸，引用计数还存在着一个致命的弱点，这一点虽然看似很小，然而其存在却几乎宣判了引用计数机制在垃圾收集技术中的死刑。也正是由于这一致命的弱点，使得狭义的垃圾收集研究从来没有将引用技术包含在内。这个致命的弱点就是循环引用。

我们知道，引用计数机制非常简单，当一个对象的引用被创建或复制时，对象的引用计数加 1；当一个对象的引用被销毁时，对象的引用计数减 1。如果对象的引用计数减少为 0，那么意味着对象已经不会被任何人使用，可以将其所占用的内存释放。问题的关键就在于，循环引用可以使一组对象的引用计数都不为 0，然而这些对象实际上并没有被任何外部变量引用，它们之间只是互相引用。这意味着不会再有人使用这组对象，应该回收这些对象所占用的内存，然而由于互相引用的存在，每一个对象的引用计数都不为 0，因此这些对象所占用的内存永远不会被回收。图 10 展示了 Python 中的一个循环引用。

```
>>> l1 = []
>>> l2 = []
>>> l1.append(l2)
>>> l2.append(l1)
>>> l1
[[[...]]]
>>> l2
[[[...]]]
```

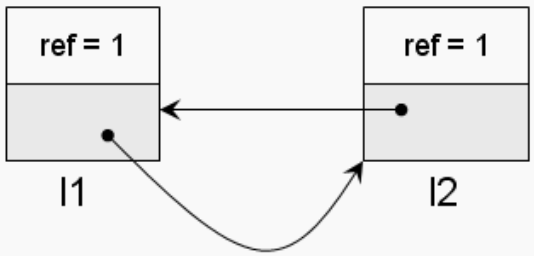

图 10 Python 中的循环引用

毫无疑问，这一点是致命的，这与手动进行内存管理所产生的内存泄露毫无区别。虽然在实际中，可以通过某种方法在语言一级保证不出现循环引用，然而这就要求开发者在将精力放到问题域的建模、实现上时，还需要花费额外的精力来精心设计代码结构，以保证不出现循环引用。这一点将立刻把所有的 Python 开发者都推到 Java，C#的阵营中。

要解决这个问题，必须引入其它的垃圾收集技术来打破循环引用，Python 中引入了主流垃圾收集技术中的标记—清除和分代收集两种技术来填补其内存管理机制中最后的也是最致命的漏洞。

3.2 三色标记模型

无论何种垃圾收集机制，一般都分为两个阶段：垃圾检测和垃圾回收。垃圾检测是从所有的已分配的内存中区别出可以回收的内存和不可回收的内存，而垃圾回收则是使系统重新掌握在垃圾检测阶段所标识出来的可回收内存块。在本节，我们将来看一看标记—清除（Mark—Sweep）方法是如何实现的，并为这个过程建立一个三色标记模型。Python 中的垃圾收集正是基于这个模型完成的。

从具体的实现上来讲，标记—清除方法同样遵循垃圾收集的两个阶段，其简要工作过程如下：

1、寻找根对象（root object）的集合，所谓的 root object 即是一些全局引用和函数栈中的引用。这些引用所引用的对象是不可被删除的。而这个 root object 集合也是垃圾检测动作的起点

2、从 root object 集合出发，沿着 root object 集合中的每一个引用，如果能到达某个对象 A，则 A 称为可达的（reachable），可达的对象也不可被删除。这个阶段就是垃圾检测阶段。

3、当垃圾检测阶段结束后，所有的对象分为了可达的和不可达的(unreachable)两部分，所有的可达对象都必须予以保留，而所有的不可达对象所占用的内存将被回收，这就是垃圾回收阶段。

在垃圾收集动作被激活之前，系统中所分配的所有对象和对象之间的引用组成了一张有向图，其中对象是图中的节点，而对象间的引用是图的边。我们在这个有向图的基础上建立一个三色标注模型，更形象地展示垃圾收集的整个动作。当垃圾收集开始时，我们假设系统中的所有对象都是不可达的，对应在有向图上，即所有的节点都标注为白色。随后，垃圾收集的动作开始，沿着始于 root object 集合中的某个 object 的引用链，在某个时刻到达了对象 A，那么我们将 A 标记为灰色，灰色表示一个对象是可达的，但是其所包含的引用还没有检查。当我们检

查了对象 A 中所包含的所有引用之后，A 将被标记为黑色，以示其包含的所有引用已经被检查过了。显然，这时，A 中引用所引用的对象则被标记为了灰色。假如我们从 root object 集合出发，采用先广搜索的策略，可以想象，灰色节点对象集合就如同一个波的阵面一样，不断向外扩散，随着所有的灰色节点都变为了黑色节点，也就意味着垃圾检测阶段结束了。图 11 展示了垃圾收集的某个时刻有向图的一个局部。

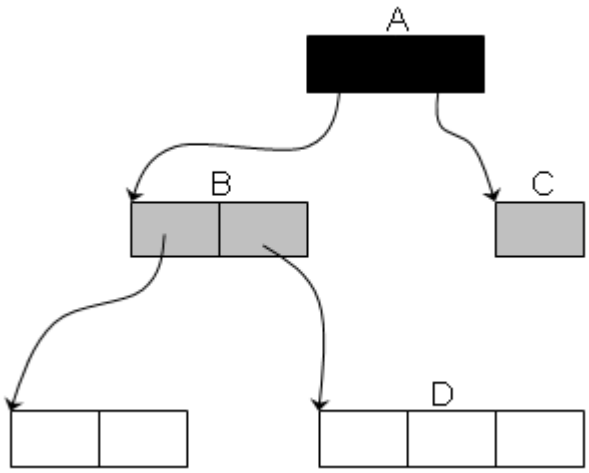

图 11 垃圾收集过程中某个时刻的多个对象组成的有向图

4 Python 中的垃圾收集

如前所述，在 Python 中，主要的内存管理手段是引用计数机制，而标记—清除和分代收集只是为了打破循环引用而引入的补充技术。这一事实意味着 Python 中的垃圾收集只关注可能会产生循环引用的对象。很显然，像 PyIntObject，PyStringObject 这些对象是绝不可能产生循环引用的，因为它们内部不可能持有对其他对象的引用。Python 中的循环引用总是发生在 container 对象之间，所谓 container 对象即是内部可持有对其他对象的引用的对象，比如 list，dict，class，instance 等等。当 Python 的垃圾收集机制运行时，只需要去检查这些 container 对象，而对 PyIntObject，PyStringObject 这些对象则不需理会，这使得垃圾收集带来的开销只

依赖于 container 对象的数量，而非所有对象的数量。为了达到这一点，Python 必须跟踪所创建的每一个 container 对象，并将这些对象组织到一个集合中，只有如此，才能将垃圾收集的动作限制在这些对象上。那么 Python 采用了什么结构将这些 container 对象组织在一起呢？Python 采用了一个双向链表，所有的 container 对象在创建之后，都会被插入到这个链表中。

4.1 可收集对象链表

在对 Python 对象机制的分析中我们已经看到，任何一个 Python 对象都分为两部分，一部分是 PyObject_HEAD，而另一部分是对象自身的数据。然而，对于一个需要被垃圾收集机制跟踪的 container 对象而言，这还不够，因为这个对象还必须链入到 Python 内部的可收集对象链表中去。一个 container 对象想要成为一个可收集的对象，则必须加入另外的信息，这个信息位于 PyObject_HEAD 之前，称为 PyGC_Head。

```
[objimpl.h]
typedef union _gc_head {
    struct {
        union _gc_head *gc_next;
        union _gc_head *gc_prev;
        int gc_refs;
    } gc;
    long double dummy;  /* force worst-case alignment */
} PyGC_Head;
```

所以，对于 Python 所创建的可收集 container 对象，其内存分布与我们之前所了解的内存布局是不同的，我们可以从可收集 container 对象的创建过程中窥见其内存分布。

```
[gcmodule.c]
PyObject* _PyObject_GC_New(PyTypeObject *tp)
{
    PyObject *op = _PyObject_GC_Malloc(_PyObject_SIZE(tp));
    if (op != NULL)
        op = PyObject_INIT(op, tp);
    return op;
```

```
}

#define _PyGC_REFS_UNTRACKED                    (-2)
#define GC_UNTRACKED                    _PyGC_REFS_UNTRACKED

PyObject* _PyObject_GC_Malloc(size_t basicsize)
{
    PyObject *op;

    //[1]：为对象本身及 PyGC_Head 申请内存

    PyGC_Head *g = PyObject_MALLOC(sizeof(PyGC_Head) + basicsize);
    g->gc.gc_refs = GC_UNTRACKED;//[2]
     generations[0].count++; /* number of allocated GC objects */
     //[3]
    if (generations[0].count > generations[0].threshold &&
        enabled &&
        generations[0].threshold &&
        !collecting &&
        !PyErr_Occurred()) {
        collecting = 1;
        collect_generations();
        collecting = 0;
    }
    op = FROM_GC(g);//[4]
    return op;
}
```

从代码[1]处可以清楚地看到，当 Python 为可收集的 container 对象申请内存空间时，为 PyGC_Head 也申请了内存空间，并且其位置在 container 对象之前。所以我们现在对于 PyListObject，PyDictObject 等对象的内存分布的推测应该变成如图 12 所示的情形。需要注意，在申请内存时，使用的是 PyObject_MALLOC，这将最终调用我们在上一节花费巨大精力分析的 PyObject_Malloc。

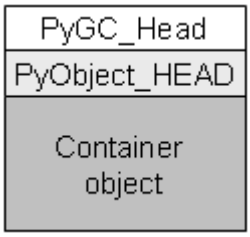

图 12 被垃圾收集机制监控的 container 对象

在可收集 container 对象的内存分布中，内存分为三个部分，首先的一块用于垃圾收集机制，然后紧跟着的是 Python 中所有对象都会有的 PyObject_HEAD，最后才是属于 container 对象自身的数据，这里的 Container Object 既可以是 PyDictObject，也可以是 PyListObject，等等等等。

在 PyGC_Head 部分，除了两个用于建立链表结构的前向和后向指针外，还有一个 gc_ref，在代码[2]处我们看到，这个值被初始化为 GC_UNTRACED。这个变量对于垃圾收集的运行至关重要，但是现在，在深入这个变量以及垃圾收集之前，我们还需要了解其他一些事实，这里我们先将其放下。同样对于代码[3]，我们这里也先不剖析，不过别急，马上我们就会回来。

当垃圾收集机制运行期间，我们需要在一个可收集的 container 对象的 PyGC_Head 部分和 PyObject_HEAD 部分来回转换，更清楚地说，某些时候，我们持有一个对象 A 的 PyObject_HEAD 的地址，但是我们需要根据此地址获得 A 的 PyGC_Head 的地址；而在某些时候，我们又需要进行这一动作的逆运算。Python 提供了在两个地址之间的转换算法，代码[4]处使用的是从 PyGC_Head 地址转换为 PyObject_HEAD 地址的算法。

```
[gcmodule.c]
/* Get an object's GC head */
#define AS_GC(o) ((PyGC_Head *)(o)-1)
/* Get the object given the GC head */
#define FROM_GC(g) ((PyObject *)(((PyGC_Head *)g)+1))

[objimpl.h]
#define _Py_AS_GC(o) ((PyGC_Head *)(o)-1)
```

在 PyGC_Head 中，出现了用于建立链表的两个指针，只有将创建的可收集 container 对象链接到 Python 内部维护的可收集对象链表中，Python 的垃圾收集机制才能跟踪和处理这个 container 对象。但是我们发现，在创建可收集 container 对象之时，并没有立即将这个对象链接到链表中。实际上，这个动作是发生在创建某个 container 对象的最后一步，从 PyDictObject 的

创建过程，我们可以清楚地看到这一点。

```
[dictobject.c]
PyObject* PyDict_New(void)
{
    register dictobject *mp;

    ……

    mp = PyObject_GC_New(dictobject, &PyDict_Type);

    ……

    _PyObject_GC_TRACK(mp);
    return (PyObject *)mp;
}
```

在创建 container 对象的最后一步，Python 通过_PyObject_GC_TRACD 将所创建的 container 对象链接到了 Python 中的可收集对象链表中。

```
[objimpl.h]
#define _PyObject_GC_TRACK(o) do { \
    PyGC_Head *g = _Py_AS_GC(o); \
    if (g->gc.gc_refs != _PyGC_REFS_UNTRACKED) \
        Py_FatalError("GC object already tracked"); \
    g->gc.gc_refs = _PyGC_REFS_REACHABLE; \
    g->gc.gc_next = _PyGC_generation0; \
    g->gc.gc_prev = _PyGC_generation0->gc.gc_prev; \
    g->gc.gc_prev->gc.gc_next = g; \
    _PyGC_generation0->gc.gc_prev = g; \
    } while (0);
```

前面我们说过，Python 会将自己的垃圾收集机制限制在其维护的可收集对象链表上，因为所有的循环引用一定是发生在这个链表中的一群对象之间。在_PyObject_GC_TRACK 之后，我们所创建的 container 对象也就置于 Python 垃圾收集机制的掌控之中了。

同样，Python 还提供了将一个 container 对象从链表中摘除的方法，显然，这个方法应该在对象被销毁时调用。

```
[objimpl.h]
#define _PyObject_GC_UNTRACK(o) do { \
    PyGC_Head *g = _Py_AS_GC(o); \
```

```
    assert(g->gc.gc_refs != _PyGC_REFS_UNTRACKED); \
    g->gc.gc_refs = _PyGC_REFS_UNTRACKED; \
    g->gc.gc_prev->gc.gc_next = g->gc.gc_next; \
    g->gc.gc_next->gc.gc_prev = g->gc.gc_prev; \
    g->gc.gc_next = NULL; \
    } while (0);
```

很明显，_PyObject_GC_UNTRACK 仅仅是_PyObject_GC_TRACK 的逆运算而已。在图 12 中，我们展示了 Python 运行过程的某个时刻，所建立起来的可收集对象链表。

图 12 可收集对象链表

4.2 分代的垃圾收集

分代的垃圾收集技术是在上个世纪 80 年代初发展起来的一种垃圾收集机制，一系列的研究表明，无论使用何种语言开发，无论开发的是何种类型，何种规模的程序，都存在这样一点相同之处，即，一定比例的内存块的生存周期都比较短，通常是几百万条机器指令的时间，而剩下的内存块，其生存周期会比较长，甚至会从程序开始一直持续到程序结束。研究表明，对于不同的语言，不同的应用程序，这个比例会在 80%到 98%之间游走。

这一发现对于垃圾收集技术有着重要的意义。从前面的分析我们已经看到，象标记—清除这样的垃圾收集所带来的额外操作实际上与系统中总的内存块的数量的相关的，当需回收的内存块越多时，垃圾检测带来的额外操作就越多，而垃圾回收带来的额外操作就越少；反之，当需回收的内存块越少时，垃圾检测就将比垃圾回收带来更少的额外操作。而无论如何，我们可以看到，当系统中使用的内存越少时，整个垃圾收集所带来的额外操作也就越少。为了使垃

圾收集的效率提高，基于研究人员所发现的统计规律，我们就可以采用一种以空间换时间的策略。这种以空间换时间的分代收集的技术正是当前支撑着 Java 的关键技术。

这种以空间换时间的总体思想是：将系统中的所有内存块根据其存活时间划分为不同的集合，每一个集合就称为一个“代”，垃圾收集的频率随着“代”的存活时间的增大而减小，也就是说，活得越长的对象，就越可能不是垃圾，就应该越少去收集。那么这个存活时间是如何来衡量的呢，通常是利用经过了几次垃圾收集动作来衡量，如果一个对象经过的垃圾收集次数越多，那么显然，其存活时间就越长。

举个具体的例子来说，当某些内存块 M 经过了 3 次垃圾收集的洗礼还依然存活时，我们就将 M 划到一个集合 A 中去，而新分配的内存都划到集合 B 中去，当垃圾收集开始工作时，大多数情况下都只对集合 B 进行垃圾回收，而对 A 的回收要等到过了相当长一段时间才进行，这就使得垃圾收集需要处理的内存变少了，效率则得到提高。可以想见，B 中的内存在经过几次收集之后，有一些内存块会被转移到 A 中，而在 A 中，实际上确实会存在一些垃圾，这些垃圾的回收因为这种分代的机制会被延迟。这就是我们所说的以空间换时间的策略。

在 Python 中，也引入了分代的垃圾收集机制，总共有三个“代”。在 _PyObject_GC_TRACK 中我们看到了一个名为_PyGC_generation0 的神秘变量，这个变量是 Python 内部维护的一个指针，指向的正是 Python 中第 0 代的内存块集合。

“代”似乎是一个很抽象的概念，实际上，在 Python 中，一个“代”就是一个链表，所有属于同一“代”的内存块都链接在同一个链表中。既然 Python 中总共有 3“代”，那么很显然，Python 中实际是维护了三条链表。更明确地说，一“代”就是我们在上一节中所提到的一条可收集对象链表，在前面所介绍的链表的基础上，为了支持分代机制，需要的仅仅是一个额外的表头而已。

[gcmodule.c]

```
struct gc_generation {
    PyGC_Head head;
    int threshold; /* collection threshold */
    int count; /* count of allocations or collections of younger
            generations */
};
```

Python 中有一个维护了三个 gc_generation 结构的数组，通过这个数组控制了三条可收集对象链表，这就是 Python 中用于分代垃圾收集的三个“代”。

```
[gcmodule.c]
#define NUM_GENERATIONS 3
#define GEN_HEAD(n) (&generations[n].head)

/* linked lists of container objects */
static struct gc_generation generations[NUM_GENERATIONS] = {
    /* PyGC_Head,                                    threshold,  count */
    {{GEN_HEAD(0), GEN_HEAD(0), 0}},   700,         0},
    {{GEN_HEAD(1), GEN_HEAD(1), 0}},   10,                0},
    {{GEN_HEAD(2), GEN_HEAD(2), 0}},   10,          0},
};

PyGC_Head *_PyGC_generation0 = GEN_HEAD(0);
```

我们在_PyObject_GC_TRACK 中所见的_PyGC_generation0 不偏不斜，指向的正是第 0 代内存集合。图 13 展示了用于控制三个“代”的 generations。

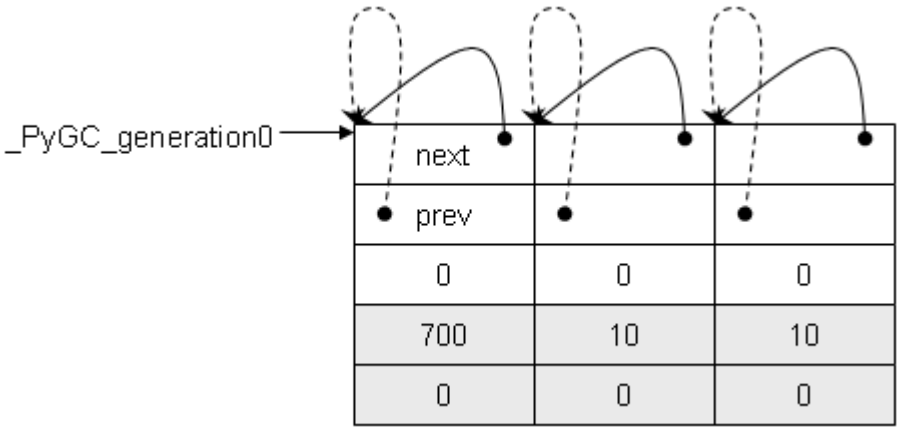

图 13 Python 中维护 3 代内存的控制结构

对于每一个 gc_generation，其中的 count 记录了当前这条可收集对象链表中一共有多少个

可收集对象，在_PyObject_GC_Malloc 中，我们可以看到，再分配了内存之后，都会进行 generations[0].count++的动作，将第 0 代内存链表中所维护的内存块的数量加 1，这预示着所有新创建的对象实际上都会被加入到第 0 代可收集对象链表中，这一点在_PyObject_GC_TRACK 中被证实了。细心的朋友可能已经发现，这个递增 count 的动作实际上是被提前了，因为直到 _PyObject_GC_TRACK 时，所创建的可收集 container 对象才会真正被链接到第 0 代内存链表中。

在 gc_generation 中，threshold 纪录了该条可收集对象链表中最多可容纳多少个可收集对象，从 Python 的实现代码中，可以发现，第 0 代链表中最多可以容纳 700 个 container 对象，一旦第 0 代内存链表的 count 超过了 700 这个极限值，则会立刻触发垃圾回收机制，这一点正是 _PyObject_GC_Malloc 在代码[3]处所表现出来的行为。

```
[gcmodule.c]
static Py_ssize_t collect_generations(void)
{
    int i;
    Py_ssize_t n = 0;

    /* Find the oldest generation (higest numbered) where the count
     * exceeds the threshold.  Objects in the that generation and
     * generations younger than it will be collected. */
    for (i = NUM_GENERATIONS-1; i >= 0; i--) {
        if (generations[i].count > generations[i].threshold) {
            n = collect(i);
            break;
        }
    }
    return n;
}
```

在_PyObject_GC_Malloc 中，虽然是由第 0 代内存链表的越界触发了垃圾收集，但是 Python 会借此时机，对所有“代”内存链表都进行垃圾收集，当然，这只能在与某“代”对应的链表的 count 值越界的条件满足时才进行。从 Python 源码中的注释里我们看到，在

collect_generations 中，Python 将寻找满足 count 值越界条件的最“老”的那一“代”（也就是 generations 数组中序号最高的那一“代”），然后回收这“代”对应的内存和所有比它年轻的“代”对应的内存。但是我们在源码中却明明白白地看见，找到最老的那“代”，并进行处理之后，就潇洒地一个 break 动作，拍拍屁股走人了，比它年轻的“代”根本没处理啊，Python 源码里的注释不是睁眼说瞎话么。实际上，问题的关键出在哪个 collect 和它接受的参数上。这个函数是 Python 中垃圾收集机制的关键实现所在，下一节将详细剖析这个函数。

4.3 Python 中的标记—清除方法

前面我们提到，Python 采用了三代的分代收集机制，如果当前收集的是第 1 代，那么在开始垃圾收集之前，Python 会将比其“年轻”的所有代的内存链表（当然，在这里只有第 0 代）整个地链接到第 1 代内存链表之后，这个操作是通过 gc_list_merge 实现的。

```
[gcmodule.c]
static void gc_list_init(PyGC_Head *list)
{
    list->gc.gc_prev = list;
    list->gc.gc_next = list;
}

static void gc_list_merge(PyGC_Head *from, PyGC_Head *to)
{
    PyGC_Head *tail;
    if (!gc_list_is_empty(from)) {
        tail = to->gc.gc_prev;
        tail->gc.gc_next = from->gc.gc_next;
        tail->gc.gc_next->gc.gc_prev = tail;
        to->gc.gc_prev = from->gc.gc_prev;
        to->gc.gc_prev->gc.gc_next = to;
    }
    gc_list_init(from);
}
```

在我们的例子中，from 就是第 0 代内存链表，而 to 就是第 1 代内存链表。图 14 展示了

merge 的结果。

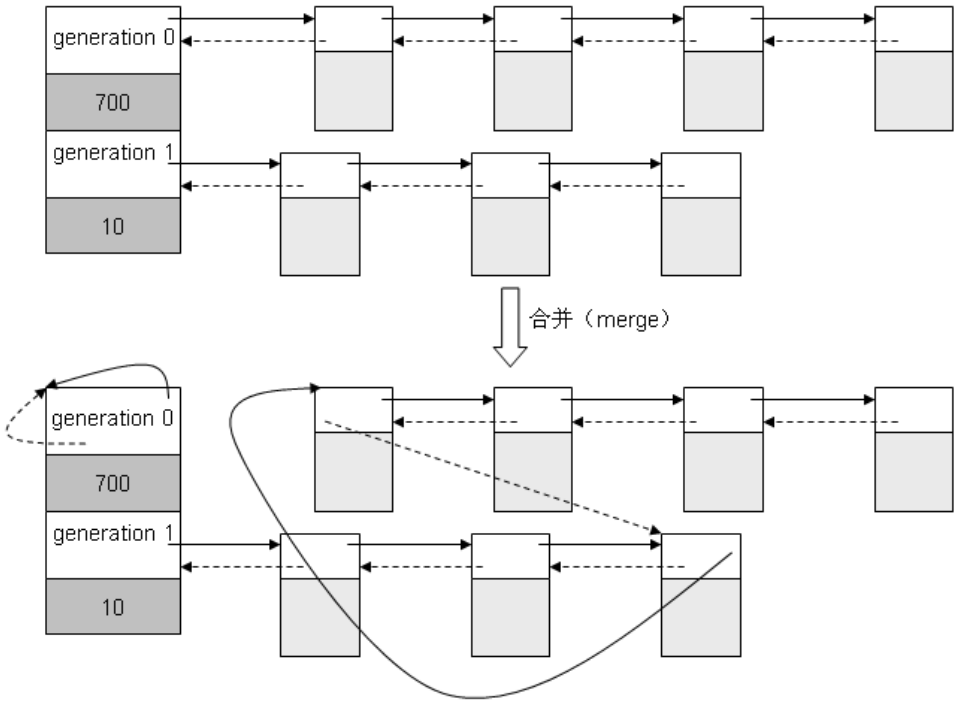

图 14 对可收集对象链表的合并操作

此后的标记—清除算法就将在 merge 之后所得到的那一条内存链表上进行。同时图 14 也能说明在 4.2 节末尾的 collect_generations 函数中，为什么 Python 拍拍屁股走人后，还敢大言不惭地说它对符合垃圾回收条件的最“老”的“代”以及所有比它年轻的“代”都进行了回收。

在详细剖析 Python 中用于打破循环引用的标记—清除垃圾收集方法之前，需要先建立一个循环引用的最简单的例子，基于这个例子，我们将描述 Python 中使用的标记—清除算法。这个例子与图 10 所示的例子相类似，但是不同的是，它多了一个外部引用，如图 15 所示。

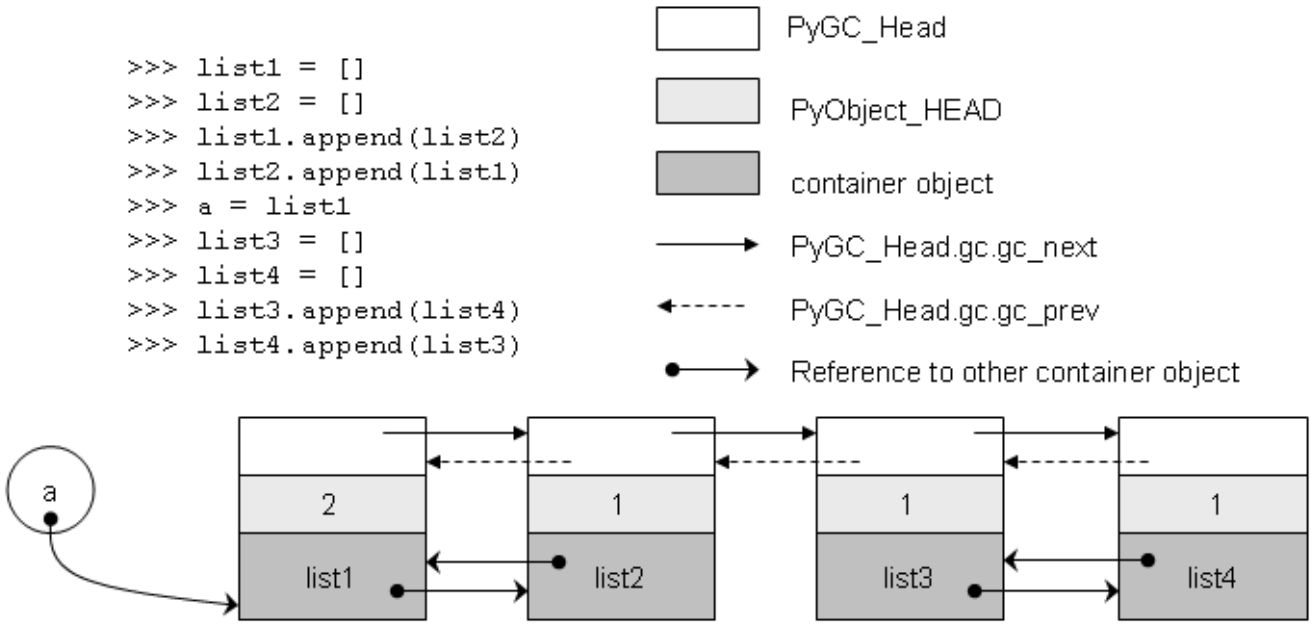

图 15 用于演示标记一清除算法的例子

其中，在 PyObject_HEAD 部分标出的数值表示对象的引用计数 ob_refcnt 的值。

4.3.1 寻找 Root Object 集合

为了使用标记一清除算法，按照我们前面对垃圾收集算法的一般性描述，首先我们需要寻找出 root object 集合。那么在图 15 中，那些 container 对象应该属于 root object 呢?

让我们换个角度来思考，前面提到，root object 是不能被删除的对象，也就是说，有可收集对象链表外部的某个引用在引用这个对象，删除这个对象会导致错误的行为，从图 15 中可以看到只有 list1 应该属于 root object。但是，这只是观察的结果，应该如何设计一种算法来得到这个结果呢?

我们注意到这样一个事实，如果两个对象的引用计数都为 1，但是仅仅存在它们之间的循环引用，那么这两个对象都是需要被回收的，也就是说，虽然它们的引用计数虽然表现为非 0，但实际上有效的引用计数为 0。这里，我们提出了**有效引用计数**的概念，为了从引用计数

获得有效引用计数，必须将循环引用的影响去除，也就是说，将环从引用中摘取，具体的实现就是两个对象各自的引用计数值都减去 1，这样一来，两个对象的引用计数都成为了 0，我们挥去了循环引用的迷雾，使有效引用计数现出了真身。那么如何使两个对象的引用计数都减 1 呢，很简单，假设这两个对象为 A，B，我们从 A 出发，因为它有一个对 B 的引用，则将 B 的引用计数减 1，然后顺着引用达到 B，因为 B 有一个对 A 的引用，同样将 A 的引用减 1，这样，就完成了循环引用对象间环的摘除。

但是这样就引出了一个问题，假设可收集对象链表中的 container 对象 A 有一个对对象 C 的引用，而 C 并不在这个链表中，如果将 C 的引用计数减 1，而最后 A 并没有被回收，那么显然，C 的引用计数被错误地减少了 1，这将导致在未来的某个时刻出现一个对 C 的悬空引用。这就要求我们必须在 A 没有被删除的情况下复原 C 的引用计数，如果采用这样的方案，那么维护引用计数的复杂度将成倍增长。换一个角度，其实我们有更好的做法，我们并不改动真实的引用计数，而是改动引用计数的副本，对于副本无论做任何的改动，都不会影响到对象生命周期的维护，因为这个副本的唯一作用就是寻找 root object 集合。这个副本就是 PyGC_Head 中的 gc.gc_ref。在垃圾收集的第一步，就是遍历可收集对象链表，将每个对象的 gc.gc_ref 值设置为其 ob_refcnt 值。

```
[gcmodule.c]
static void update_refs(PyGC_Head *containers)
{
    PyGC_Head *gc = containers->gc.gc_next;
    for (; gc != containers; gc = gc->gc.gc_next) {
        assert(gc->gc.gc_refs == GC_REACHABLE);
        gc->gc.gc_refs = FROM_GC(gc)->ob_refcnt;
    }
}
```

接下来的动作就是要将环引用从引用中摘除。

```
[gcmodule.c]
static void subtract_refs(PyGC_Head *containers)
```

```
{
    traverseproc traverse;
    PyGC_Head *gc = containers->gc.gc_next;
    for (; gc != containers; gc=gc->gc.gc_next) {
        traverse = FROM_GC(gc)->ob_type->tp_traverse;
        (void) traverse(FROM_GC(gc),(visitproc)visit_decref, NULL);
    }
}
```

其中的 traverse 是与特定的 container 对象相关的，在 container 对象的类型对象中定义，一般来说，traverse 的动作都是遍历 container 对象中的每一个引用，然后对引用进行某种动作，而这个动作在 subtract_refs 中就是 visit_decref，它以一个回调函数的形式传递到 traverse 操作中。作为例子，我们来看看 PyDictObject 对象所定义的 traverse 操作。

```
[object.h]
typedef int (*visitproc)(PyObject *, void *);
typedef int (*traverseproc)(PyObject *, visitproc, void *);

[dictobject.c]
PyTypeObject PyDict_Type = {

    ......

    (traverseproc)dict_traverse,        /* tp_traverse */

    ......

};

static int dict_traverse(PyObject *op, visitproc visit, void *arg)
{
    int i = 0, err;
    PyObject *pk;
    PyObject *pv;

    while (PyDict_Next(op, &i, &pk, &pv)) {
        visit(pk, arg);
        visit(pv, arg);
    }
}
```

对于 dict 中的所有键和所有值都回调用回调函数，即 subtract_refs 中传递进来的

visit_decref。

```
[gcmodule.c]
static int visit_decref(PyObject *op, void *data)
{

    //PyObject_IS_GC 判断 op 指向的对象是不是被垃圾收集监控的

    //通常在 container 对象的 type 对象中有 Py_TPFLAGS_HAVE_GC 符号

    //标识 container 对象是被垃圾收集监控的
    if(PyObject_IS_GC(op)) {
        PyGC_Head *gc = AS_GC(op);
        if (gc->gc.gc_refs > 0)
            gc->gc.gc_refs--;
    }
    return 0;
}
```

在完成了 subtract_refs 之后，可收集对象链表中所有 contianer 对象之间的环引用都被摘除了，这时，有一些 container 对象的 PyGC_Head.gc.gc_ref 还不为 0，这就意味着存在对这些对象的外部引用，这些对象，就是开始标记一清除算法的 root object 集合。

图 16 展示了图 15 所示的例子在经过了 update_refs 和 subtract_refs 两步处理后所得到的 root object 集合。

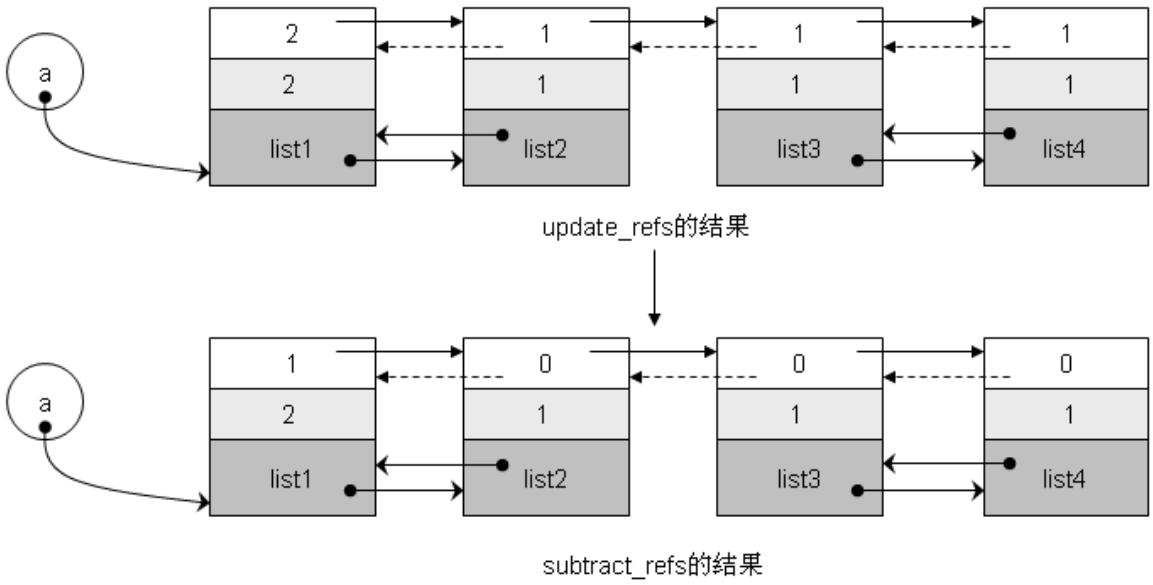

图 16 update_refs 和 subtract_refs 的执行结果

4.3.2 垃圾标记

成功地寻找到了 root object 集合之后，我们就可以从 root object 出发，沿着引用链出发，一个接一个地标记不能回收的内存，由于 root object 集合中的对象是不能回收的，因此，被这些对象直接或间接引用的对象也是不能回收的。在从 root object 出发之前，我们首先要将现在的内存链表一分为 2，一条链表中维护 root object 集合，成为 root 链表，而另一条链表中维护剩下的对象，称为 unreachable 链表。之所以要剖成两个链表，是基于这样的一种考虑，显然，现在的 unreachable 链表是名不副实的，其中可能存在被 root 链表中的对象直接或间接引用的对象，这些对象也是不能回收的，一旦在标记的过程中，发现了这样的对象，就将其从 unreachable 链表中移到 root 链表中，当完成标记后，unreachable 链表中剩下的对象就是名副其实的垃圾对象了，接下来的垃圾回收只需限制在 unreachable 链表中即可。

为此，Python 准备了一条名为 unreachable 的链表，通过 move_unreachable 完成对原始链表的剖分。

```
[gcmodule.c]
static void move_unreachable(PyGC_Head *young, PyGC_Head *unreachable)
{
    PyGC_Head *gc = young->gc.gc_next;
    while (gc != young) {
        PyGC_Head *next;

        //[1]：对于root object，设置其gc_refs为GC_REACHABLE标志

        if (gc->gc.gc_refs) {
            PyObject *op = FROM_GC(gc);
            traverseproc traverse = op->ob_type->tp_traverse;
            gc->gc.gc_refs = GC_REACHABLE;
            (void) traverse(op, (visitproc)visit_reachable, (void *)young);
            next = gc->gc.gc_next;
        }
```

```
        //[2]：对于非 root 对象，移到 unreachable 链表中，

        //并标记为 GC_TENTATIVELY_UNREACHABLE
        else {
            next = gc->gc.gc_next;
            gc_list_move(gc, unreachable);
            gc->gc.gc_refs = GC_TENTATIVELY_UNREACHABLE;
        }
        gc = next;
    }
}

static int visit_reachable(PyObject *op, PyGC_Head *reachable)
{
    if (PyObject_IS_GC(op)) {
        PyGC_Head *gc = AS_GC(op);
        const int gc_refs = gc->gc.gc_refs;

        //[3]：对于还没有处理的对象，恢复其 gc_refs
        if (gc_refs == 0) {
            gc->gc.gc_refs = 1;
        }

        //[4]：对于已经被挪到 unreachable 链表中的对象，将其再次挪到原来的链表
        else if (gc_refs == GC_TENTATIVELY_UNREACHABLE) {
            gc_list_move(gc, reachable);
            gc->gc.gc_refs = 1;
        }
         else {
            assert(gc_refs > 0 || gc_refs == GC_REACHABLE || gc_refs ==
GC_UNTRACKED);
         }
    }
    return 0;
}
```

在 move_unreachable 中，沿着可收集对象链表依次向前，并检查其 PyGC_Head.gc.gc_ref 值，我们发现，这里的动作是遍历链表，而并非从 root object 集合出发，遍历引用链。这将导致一个微妙的结果，即当检查到一个 gc_ref 为 0 的对象时，我们并不能立即断定这个对象就是垃

圾对象，因为这个对象之后的对象链表上，也许还会遇到一个 root object，而这个 root object 将引用该对象。所以，这个对象只是一个可能的垃圾对象，因此在代码[2]处将其暂时性的标注为 GC_TENTATIVELY_UNREACHABLE，但是还是通过 gc_list_move 将其搬移到了 unreachable 对象链表中，不过不要紧，马上我们就能看到，Python 留下了一条后路。

当在 move_unreachable 中遇到一个 gc_refs 不为 0 的对象 A 时，显然，A 是 root object 或从某个 root object 能引用到的对象，而 A 所引用的所有对象也都是不可回收的对象。因此，在代码[1]处，会再次调用与特定对象相关的 traverse 操作，依次对 A 中所引用的对象进行调用 visit_reachable。在 visit_reachable 的代码[4]处我们就可以发现，如果 A 所引用的对象之前曾被标注为 GC_TENTATIVELY_UNREACHABLE，那么现在能通过 A 访问到它，就意味着它也是一个不可回收对象，所以 Python 会重新将其从 unreachable 链表中搬移回原来的列表。注意，这里的 reachable，即是 move_unreachabl 中的 young，也就是我们所谓的 root object 链表。Python 还会将其 gc_refs 设置为 1，表示该对象是一个不可回收的对象。同样，在代码[1]处，我们看到对 A 所引用的 gc_refs 为 0 的对象，也将其 gc_refs 设置为了 1。想一想，这样的对象是什么对象呢？显然，它是在链表中 move_unreachable 操作还没有访问到的对象，这样，Python 就直接掐断了之后 move_unreachable 访问它时将其移动到 unreachable 链表的诱因。图 17 显示了链表被剖分后的结果。

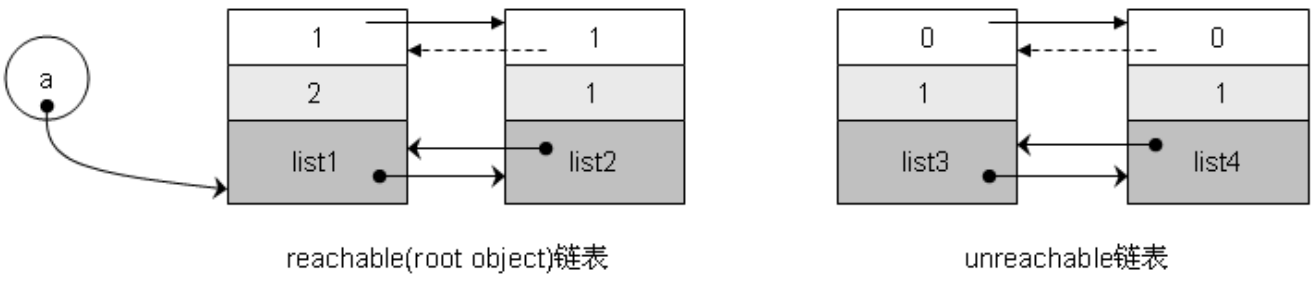

图 17 最终获得的 reachable 链表和 unreachable 链表

当 move_unreachable 完成之后，最初的一条链表就被剖分成了两条链表，在 unreachable 链

表中，就是我们所发现的垃圾对象，是垃圾回收的目标。但是，等一等，在 unreachable 链表中，所有的对象都能被安全地回收吗？恐怕未必。当二十世纪初人们以为物理学的大厦已经建立完毕时，这座大厦的上空又出现了小小的三朵乌云，而这些乌云最终将会把经典物理学的大厦摧枯拉朽般地摧毁。现在，我们也遇到了这样的乌云。

问题出在一种特殊的 container 对象，即从类对象实例化得到的实例对象。当我们用 Python 定义一个 class 时，可以为这个 class 定义一个特殊的方法：__del__，这在 Python 中被称为 finalizer。当一个拥有 finalizer 的实例对象被销毁时，首先会调用这个 finalizer，因为这个__del__就是 Python 为开发人员提供的在对象被销毁时进行某些资源释放的 Hook 机制。现在问题来了，现在我们已经知道，最终在 unreachable 链表中出现的对象都是只存在循环引用的对象，需要被销毁。但是假如现在在 unreachbale 中，有两个对象，对象 B 在 finalizer 中调用了对象 A 的某个操作，这意味着安全的垃圾回收必须保证对象 A 一定要在对象 B 之后被回收，但是 Python 无法做到这一点，Python 在回收垃圾时不能保证回收的顺序。于是，有可能在 A 被销毁之后，B 在销毁时访问已经不存在的 A，毫无疑问，Python 遇到麻烦了。虽然同时满足存在 finalizer 和循环引用这两个条件的概率非常低，但 Python 不能对此置之不理。这是一个非常棘手的问题，从 Python 中拿掉__del__显然是很愚蠢的，所以 Python 采用了一种保守的做法，即将 unreachable 链表中的拥有 finalizer 的 PyInstanceObject 对象统统都移到一个名为 garbage 的 PyListObject 对象中。

4.3.3 垃圾回收

要回收 unreachable 链表中的垃圾对象，就必须先打破对象间的循环引用，前面我们已经详细地阐述了如何打破循环引用的算法。在寻找 root object 时，我们引入了 gc_refs 来模拟这个

打破过程，现在我们要真刀真枪对 ob_refcnt 下手了，直到 unreachable 链表中的每一个对象的 ob_refcnt 变为 0，引发对象的销毁。

```
[gcmodule.c]
static int
gc_list_is_empty(PyGC_Head *list)
{
        return (list->gc.gc_next == list);
}

static void delete_garbage(PyGC_Head *collectable, PyGC_Head *old)
{
    inquiry clear;

    while (!gc_list_is_empty(collectable)) {
        PyGC_Head *gc = collectable->gc.gc_next;
        PyObject *op = FROM_GC(gc);

        if ((clear = op->ob_type->tp_clear) != NULL) {
            Py_INCREF(op);
            clear(op);
            Py_DECREF(op);
        }

        if (collectable->gc.gc_next == gc) {
            /* object is still alive, move it, it may die later */
            gc_list_move(gc, old);
            gc->gc.gc_refs = GC_REACHABLE;
        }
    }
}
```

其中会调用 container 对象的类型对象中的 tp_clear 操作，这个操作会调整 container 对象中每个引用所引用的对象的引用计数值，从而完成打破循环的最终目标。作为一个例子，我们来看看 PyListObject 的 clear 操作。

```
[listobject.c]
static int list_clear(PyListObject *a)
{
    int i;
    PyObject **item = a->ob_item;
```

```
    if (item != NULL) {
        i = a->ob_size;
        a->ob_size = 0;
        a->ob_item = NULL;
        a->allocated = 0;
        while (--i >= 0) {
            Py_XDECREF(item[i]);
        }
        PyMem_FREE(item);
    }
    return 0;
}
```

我们注意到，在 delete_garbage 中，有一些 unreachable 链表中的对象会被重新送回到 reachable 链表（即 delete_garbase 函数的 old 参数）中，这是由于在进行 clear 动作时，如果成功进行，则通常一个对象会把自己从垃圾收集机制维护的链表中摘除（也就是这里的 collectable 链表）。由于某些原因，对象可能在 clear 动作时，没有成功完成必要的动作，从而没有将自己从 collectable 链表摘除，这表示对象认为自己还不能被销毁，所以 Python 需要将这种对象放回 reachable 链表中。

我们来看看图 17 所示的 unreachable 链表中的 list3 和 list4 是如何被回收的。首先，在 delete_garbage 中，假如首先处理 list3，调用其 list_clear，那么会减少 list4 的引用计数，这导致 list4 的 ob_refcnt 为 0，引发对象销毁动作，会调用 list4 的 list_dealloc。

```
[listobjec.c]
static void list_dealloc(PyListObject *op)
{
    int i;
    PyObject_GC_UnTrack(op);
    if (op->ob_item != NULL) {
        i = op->ob_size;
        while (--i >= 0) {
            Py_XDECREF(op->ob_item[i]);
        }
        PyMem_FREE(op->ob_item);
    }
```

```
    ……
}
```

首先会将 list4 从可收集对象链表中摘除，然后如同 list_clear 所作的，会调整 list4 所引用的所有对象的引用计数，这个动作立即就影响到了 list3，并使其引用计数变为 0，同样，list3 的销毁动作也被触发了。如此一来，list3 和 list4 就都被安全地回收了。

4.4 垃圾收集全景

到此，我们已经详细地剖析了 Python 中垃圾收集机制的所有细节及隐秘之处，作为这些细节的综合，是时候来看一看 Python 中那个实际完成垃圾收集的 collect 是如何实现的。了解了 collect，就功德圆满了。

```
[gcmodule.c]
static long collect(int generation)
{
    int i;
    long m = 0; /* # objects collected */
    long n = 0; /* # unreachable objects that couldn't be collected */
    PyGC_Head *young; /* the generation we are examining */
    PyGC_Head *old; /* next older generation */
    PyGC_Head unreachable; /* non-problematic unreachable trash */
    PyGC_Head finalizers;  /* objects with, & reachable from, __del__ */
    PyGC_Head *gc;

    if (delstr == NULL) {
        delstr = PyString_InternFromString("__del__");
        if (delstr == NULL)
            Py_FatalError("gc couldn't allocate \"__del__\"");
    }

    /* update collection and allocation counters */
    if (generation+1 < NUM_GENERATIONS)
        generations[generation+1].count += 1;
    for (i = 0; i <= generation; i++)
        generations[i].count = 0;
```

```
/* merge younger generations with one we are currently collecting */

/* 将比当前处理的“代”更年轻的“代”的链表合并到当前“代”中

for (i = 0; i < generation; i++) {
    gc_list_merge(GEN_HEAD(i), GEN_HEAD(generation));
}

/* handy references */
young = GEN_HEAD(generation);
if (generation < NUM_GENERATIONS-1)
    old = GEN_HEAD(generation+1);
else
    old = young;

// 在待处理链表上进行打破循环的模拟，寻找 root object

update_refs(young);
subtract_refs(young);

//将待处理链表中的unreachable object 转移到 unreachable 链表中

//处理完成后，当前“代”中只剩下 reachable object 了

gc_list_init(&unreachable);
move_unreachable(young, &unreachable);

//如果可能，将当前“代”中的 reachable object 合并到更老的“代”中

if (young != old)
    gc_list_merge(young, old);

//对于 unreachable 链表中的对象，如果其带有__del__函数，则不能安全回收

//需要将这些对象收集到 finalizers 链表中，因此，这些对象引用的对象也不能

//回收，也需要放入 finalizers 链表中

gc_list_init(&finalizers);
move_finalizers(&unreachable, &finalizers);
move_finalizer_reachable(&finalizers);
```

```
    //处理弱引用（weakref），如果可能，调用弱引用中注册的 callback 操作
    handle_weakrefs(&unreachable, old);

    //对 unreachable 链表上的对象进行垃圾回收操作
    delete_garbage(&unreachable, old);

    //将含有__del__操作的实例对象收集到 Python 内部维护的名为 garbage 的链表中
    //同时将 finalizers 链表中所有对象加入 old 链表中
    (void)handle_finalizers(&finalizers, old);
    ......
}
```

我们注意到在 collect 函数中，还有对 Python 中弱引用(weakref)的处理，因为 weakref 能够注册 callback 操作，所以这个行为有点类似于带有__del__的实例对象。但是它们还是有本质的不同，weakref 能够被正确地清理掉，虽然必须引入一些额外的繁琐的操作，这些操作就隐身在 handle_weakrefs 中。而带有__del__的实例对象是不能自动被清除的，最终将被放入 garbage 链表中。对于弱引用的处理，这里就不深入了，有兴趣的读者可以自己参考 Python 源码。

到了这里，我们需要指出一点，Python 的垃圾收集机制完全是为了处理循环引用而设计的，虽然几乎大多数对象在创建时都会通过 PyObject_GC_New，并最终调用_PyObject_GC_New，将创建的对象纳入垃圾收集机制的监控中，但是有趣的是，被垃圾收集监控的对象并非只有垃圾收集机制才能回收，正常的引用计数就能销毁一个被纳入垃圾收集机制监控的对象。比如我们来看看 PyFunction 对象的正常销毁。

```
[funcobject.c]
static void func_dealloc(PyFunctionObject *op)
{
   _PyObject_GC_UNTRACK(op);
    ......
   PyObject_GC_Del(op);
```

```
}

[gcmodule.c]
void PyObject_GC_Del(void *op)
{
   PyGC_Head *g = AS_GC(op);
   if (IS_TRACKED(op))
      gc_list_remove(g);
   if (generations[0].count > 0) {
      generations[0].count--;
   }

   PyObject_FREE(g); //最终调用 PyObject_Free
}
```

如果 PyFunctionObject 对象因为正常的引用计数维护到达引用计数为 0 的状态，就会调用 func_dealloc，我们看到，PyFunctionObject 对象主动将自己从垃圾收集监控的链表中摘除，然后调用 PyObject_GC_Del 释放内存，之所以需要调用 PyObject_GC_Del，主要是为了将指向 PyObject 的指针调整为指向 PyGC_Head 的指针，以释放正确的内存。

所以，虽然有很多对象挂在了垃圾收集机制监控的链表上，但实际上更多时候，是引用计数机制在维护这些对象，只有对引用计数无能为力的循环引用，垃圾收集机制才会起作用。事实上，对循环引用之外的对象，垃圾收集是无能为力的，因为挂在垃圾收集机制上的对象都是引用计数不为 0 的，因为如果为 0，早就被引用计数机制干掉了。而引用计数不为 0 的对象只有两种情况：1、被程序使用的对象；2、循环引用中的对象。被程序使用的对象是不能被回收的，所以垃圾回收能且只能处理循环引用中的对象。

另一点需要说明的是，PyObject_GC_New 底层是以我们之前剖析的 PyObject_Malloc 作为真正申请内存的接口的，这就意味着在大多数情况下，Python 都在使用内存池。本书中我们剖析过的最大的对象就是 PyTypeObject，而这个对象也不过 200 个字节，小于 256 个字节，同样可以使用内存池。所以我们可以将垃圾收集和内存管理完全融为一体了。

4.5 Python 中的 gc 模块

Python 中通过 gc 模块为程序员提供了观察和手动使用 gc 机制的接口，这一节，我们通过 gc 模块进行一些观察，以加深对垃圾收集机制的理解。本节不对 gc 模块的使用进行介绍，关于 gc 模块的使用，请参考 Python 文档。

```
[gc1.py]
import gc
class A(object):
   pass

class B(object):
   pass

gc.set_debug(gc.DEBUG_STATS | gc.DEBUG_LEAK)
a = A()
b = B()
del a
del b
gc.collect()


结果:
F:\PythonBook\Src\memory>python gc1.py
gc: collecting generation 2...
gc: objects in each generation: 341 2692 0
gc: done.
gc: collecting generation 2...
gc: objects in each generation: 0 0 3027
gc: done.
```

通过 gc1.py 的演示，证明了对于引用计数机制能正常维护的对象，垃圾收集确实起不到任何的作用。

```
[gc2.py]
import gc
class A(object):
   pass
```

```
class B(object):
   pass

gc.set_debug(gc.DEBUG_STATS | gc.DEBUG_LEAK)
a = A()
b = B()
a.b = b
b.a = a
del a
del b
gc.collect()
```

运行结果：

```
F:\PythonBook\Src\memory>python gc2.py
gc: collecting generation 2...
gc: objects in each generation: 345 2692 0
gc: collectable <A 00B46850>
gc: 0.0160s elapsed.
gc: collectable <B 00B46870>
gc: 1201346676.5930s elapsed.
gc: collectable <dict 00B48300>
gc: 0.0160s elapsed.
gc: collectable <dict 00B48270>
gc: 1201346676.5930s elapsed.
gc: done, 4 unreachable, 0 uncollectable.
gc: collecting generation 2...
gc: objects in each generation: 0 0 3031
gc: done.
```

正如 gc2.py 的运行结果显示的，当存在循环引用时，引用计数确实不起作用了，而垃圾收集则能正确回收内存。

```
[gc3.py]
import gc
class A(object):
   def __del__(self):
      pass

class B(object):
   def __del__(self):
      pass
```

```
gc.set_debug(gc.DEBUG_STATS | gc.DEBUG_LEAK)
a = A()
b = B()
a.b = b
b.a = a
del a
del b
gc.collect()
```

运行结果：

```
F:\PythonBook\Src\memory>python gc3.py
gc: collecting generation 2...
gc: objects in each generation: 353 2692 0
gc: uncollectable <A 00B46990>
gc: uncollectable <B 00B469B0>
gc: uncollectable <dict 00B43E40>
gc: uncollectable <dict 00B481E0>
gc: done, 4 unreachable, 4 uncollectable.
gc: collecting generation 2...
gc: objects in each generation: 0 0 3037
gc: done.
```

由于我们执意在类对象中添加了__del__操作，所以 GC 很生气，后果很严重。不管是对象 a 本身（需要注意，显示结果中的 A 不是说 A 不能回收，而是**类型为 A 的 a** 不能回收），就连 a 对象内维护的__dict__也不能回收。真的是非常严重的后果，这不就是内存泄漏么？所以，没什么事，千万不要轻易启用__del__操作。

4.6 总结

尽管 Python 采用了最经典的（换句话说，最土的）引用计数来作为自动内存管理的方案，但是 Python 采用了多种方式来弥补引用计数的不足，内存池的大量使用，标记-清除垃圾收集技术的使用都极大地完善了 Python 的内存管理机制。尽管引用计数还存在着需要花费额外的内存维护引用计数值的毛病，但现在已经是 2008 年了，已经不是 64K 的年代了，这点花销，我

想，我们还是支付得起的。而另一方面，引用计数也有其优点，倘若它没有有点，早就被扫进历史的垃圾堆了。比如，引用计数将垃圾收集的开销分摊在了整个运行时，这对于 Python 的响应性能是非常有好处的。又比如，当前的研究表明，在分布式环境下，引用计数是目前最有效的垃圾收集技术。当然，这已经不是我们需要关心的话题了。

内存管理，垃圾收集，是一门非常精细和繁琐的技术，本章进行的剖析无法覆盖 Python 内存管理机制的所有细微之处，如果你不满足于本章的剖析，那么请打开你的代码浏览工具，在本章的基础上继续深入。