《源码探秘 CPython》96. 深入源码，探究垃圾回收的秘密

原创 古明地觉 古明地觉的编程教室 2022-05-22 09:30 发表于北京

收录于合集
#CPython
97个 >

楔子

Python 的垃圾回收是通过标记-清除和分代收集实现的，下面就通过源码来考察一下。

我们知道，清理一代链表时会顺带清理零代链表，总之就是把比自己"代"小的链子也清理了。那么这是怎么做到的呢？其实答案就在 gc_list_merge 函数中。

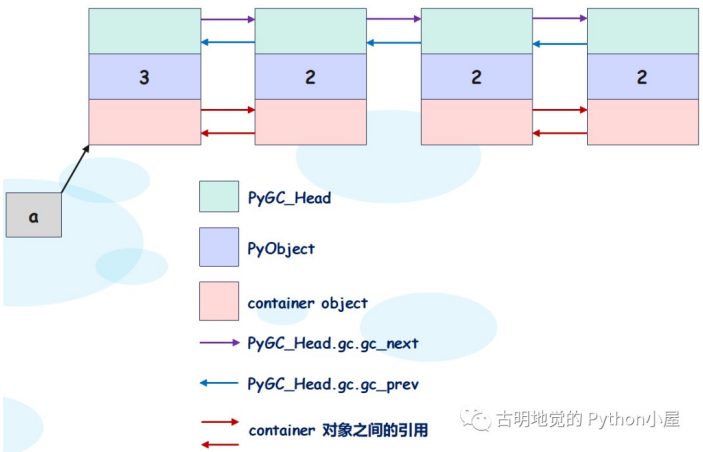
如果清理的是一代链表，那么在开始垃圾回收之前，Python 会将零代链表（比它年轻的），整个链接到一代链表之后，这样的话在清理一代的时候也会清理零代。当然啦，清理二代链表也是同理，会将一代链表和零代链表，整个链接到二代链表之后，这样清理二代的时候也会清理一代和零代。

```
//Modules/gcmodule.c
static void
gc_list_merge(PyGC_Head *from, PyGC_Head *to)
{
    assert(from != to);
    if (!gc_list_is_empty(from)) {
        PyGC_Head *to_tail = GC_PREV(to);
        PyGC_Head *from_head = GC_NEXT(from);
        PyGC_Head *from_tail = GC_PREV(from);
        assert(from_head != from);
        assert(from_tail != from);

        _PyGCHead_SET_NEXT(to_tail, from_head);
        _PyGCHead_SET_PREV(from_head, to_tail);

        _PyGCHead_SET_NEXT(from_tail, to);
        _PyGCHead_SET_PREV(to, from_tail);
    }
    gc_list_init(from);
}
```

假设我们清理的是零代链表，那么这里的 from 就是零代链表，to 就是一代链表，所以此后的标记-清除算法就将在 merge 之后的那一条链表上进行。

在探究标记-清除垃圾回收方法之前，我们需要建立一个简单的循环引用的例子。

```
lst1 = []
lst2 = []
lst1.append(lst2)
lst2.append(lst1)

# 注意这里多了一个外部引用
a = lst1

lst3 = []
lst4 = []
lst3.append(lst4)
lst4.append(lst3)
```



数字指的是当前对象的引用计数，显然 lst1 为 3，其它的都为 2，因为有一个额外的变量 a 也指向了 lst1 指向的对象。而 lst1 和 lst2，lst3 和 lst4 之间均发生了循环引用。

寻找 root object 集合

为了使用标记-清除算法，按照我们之前对垃圾收集算法的一般性描述，首先我们需要找到 root object。那么在上面的那幅图中，哪些是属于 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，在可收集对象链表的外部存在着某个对该对象的引用，删除这个对象会导致错误的行为，而在我们当前这个例子中显然只有 lst1 属于 root object。但这仅仅是观察的结果，那么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，但仅仅是它们之间存在着循环引用，那么这两个对象是需要被回收的。也就是说，尽管它们的引用计数表现为非0，但是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，为了获得有效的引用计数，必须将循环引用的影响消除，或者将这个闭环从引用中摘除（循环引用在有向图中会形成一个环），而具体的实现就是两个对象各自的引用值都减去1。这样一来，两个对象的引用计数都成为了0，我们便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 和 B，那么从 A 出发，由于它有一个对 B 的引用，则将 B 的引用计数减 1；然后顺着引用达到 B，发现它有一个对 A 的引用，那么同样会将 A 的引用减1，这样就完成了循环引用对象间环的删除。

> 总结一下就是，Python 会寻找那些具有循环引用、但是没有被外部引用的对象，并尝试把它们的引用计数都减去 1。

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 的引用计数减去1，而最后 A 并没有被回收，那么显然，C 的引用计数会被错误地减少 1，这将导致未来的某个时刻对 C 的引用会出现悬空。这就要求我们必须在 A 没有被删除的情况下恢复 C 的引用计数，可是如果采用这样的方案的话，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，我们无论做什么样的改动，都不会影响对象生命周期的维护，因为它唯一的作用就是寻找root object集合，而这个副本就是PyGC_Head中的gc.gc_ref。在垃圾回收的第一步，就是遍历可收集对象链表，将每个对象的gc.gc_ref的值设置为其ob_refcnt的值。

```
1  //Modules/gcmodule.c
2  static void
3  update_refs(PyGC_Head *containers)
4  {
5      PyGC_Head *gc = GC_NEXT(containers);
6      //将对象的引用计数拷贝给 gc_refs 字段
7      for (; gc != containers; gc = GC_NEXT(gc)) {
8          gc_reset_refs(gc, Py_REFCNT(FROM_GC(gc)));
9          _PyObject_ASSERT(FROM_GC(gc), gc_get_refs(gc) != 0);
10     }
11 }
12
13 //而接下来的动作就是要将环引用摘除
14 static void
15 subtract_refs(PyGC_Head *containers)
16 {
17     traverseproc traverse;
18     PyGC_Head *gc = GC_NEXT(containers);
19     //遍历链表每一个对象
20     for (; gc != containers; gc = GC_NEXT(gc)) {
21         //遍历当前对象所引用的对象
22         //调用visit_decref将它们的引用计数减一
23         PyObject *op = FROM_GC(gc);
24         traverse = Py_TYPE(op)->tp_traverse;
25         (void) traverse(FROM_GC(gc),
26                         (visitproc)visit_decref,
27                         op);
28     }
29 }
```

我们注意到里面有一个tp_traverse，这个是和特定的container 对象有关的，在container对象的类型对象中定义。一般来说，tp_traverse的动作就是遍历container对象中的每一个引用，然后对引用执行 visit_decref 操作，它以一个回调函数的形式传递到traverse操作中。

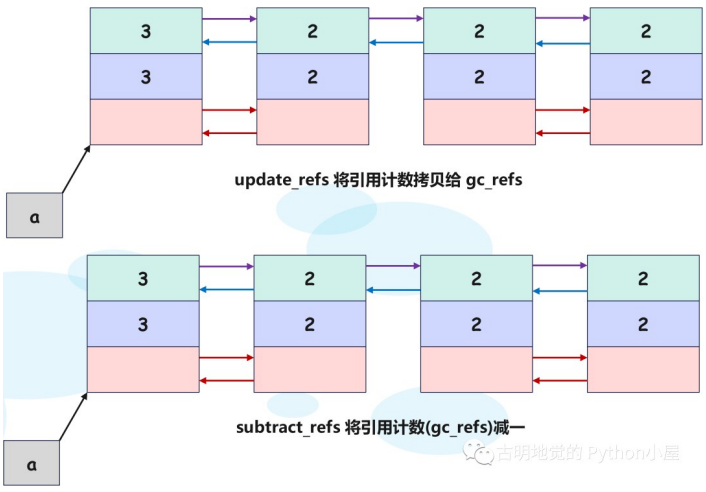
比如我们来看看PyListObject对象所定义traverse操作。

```
1  //Includeobject.h
2  typedef int (*visitproc)(PyObject *, void *);
3  typedef int (*traverseproc)(PyObject *, visitproc, void *);
```

```c
//Modules/listobject.c
PyTypeObject PyList_Type = {
    //...
    (traverseproc)list_traverse,     /* tp_traverse */
    //...
};

static int
list_traverse(PyListObject *o, visitproc visit, void *arg)
{
    Py_ssize_t i;

    for (i = Py_SIZE(o); --i >= 0; )
        //对列表中的每一个元素都进行回调的操作
        Py_VISIT(o->ob_item[i]);
    return 0;
}

//Modules/gcmodule.c
static int
visit_decref(PyObject *op, void *parent)
{
    _PyObject_ASSERT(_PyObject_CAST(parent), !_PyObject_IsFreed(op));

    //PyObject_IS_GC判断op指向的对象是否是可收集的(container对象)
    if (PyObject_IS_GC(op)) {
        //获取container对象PyGC_Head
        PyGC_Head *gc = AS_GC(op);
        if (gc_is_collecting(gc)) {
            //减少引用计数
            gc_decref(gc);
        }
    }
    return 0;
}
```

比如我们要删除一个列表，那么显然在删除之前，列表里面每个元素指向对象的引用计数肯定要减一。

在完成了subtract_refs之后，可收集对象链表中所有container对象之间的环引用就被摘除了。这时有一些container对象的PyGC_Head.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-清除算法的 root object。



举个栗子：

```c
import sys

lst1 = []
lst2 = []
lst1.append(lst2)
lst2.append(lst1)

# 注意这里多了一个外部引用
a = lst1

print(sys.getrefcount(a))  # 4
print(sys.getrefcount(lst1))  # 4
print(sys.getrefcount(lst2[0]))  # 4
```

由于sys.getrefcount函数本身会多一个引用，所以减去1的话，那么都是3，表示它们指向的对象的引用计数为3。所以这个时候 a 就想到了，除了我，还有两位老铁 list1 和 list2[0] 也指向了我指向的对象。

垃圾标记

还是以上面的代码为例。

```
lst1 = []
lst2 = []
lst1.append(lst2)
lst2.append(lst1)

a = lst1

lst3 = []
lst4 = []
lst3.append(lst4)
lst4.append(lst3)
```

假设我们现在执行了删除操作 del lst1, lst2, lst3, lst4，那么成功地寻找到root object集合之后，我们就可以从root object出发，沿着引用链，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不能回收。比如这里的 lst2，因为 lst1 不能回收，而 lst1 又 append 了 lst2，所以即便 del lst2，lst2 指向的内存也是不可以释放的。

下面在从root object出发前，首先需要将现在的内存链表一分为二，一条链表维护root object集合，成为root链表；而另一条链表中维护剩下的对象，成为unreachable链表。

由于 unreachable 链表上面的对象都是可回收的垃圾，那么显然目前的 unreachable 链表是名不副实的，或者说不符合上面的对象都可回收的条件。因为里面可能存在被 root 链表中的对象直接或者间接引用的对象，这些对象是不可以回收的，因此一旦在标记中发现了这样的对象，那么就应该将其从 unreachable 移到 root 链表中。当完成标记之后，unreachable 链表中剩下的对象就是名副其实的垃圾对象了，那么接下来的垃圾回收只需要限制在 unreachable 链表中即可。

正如我们一开始介绍的三色标记模型，确定完 root object 集合之后，就假设剩下的对象都是不可达的。然后遍历，如果可达，证明我们冤枉它了，那么就为它平冤昭雪（标记为可达）。

为此Python专门准备了一条名为 unreachable 的链表，通过 move_unreachable 函数完成了对原始链表的切分。

```
//Modules/gcmodule.c
static void
move_unreachable(PyGC_Head *young, PyGC_Head *unreachable)
{
    PyGC_Head *prev = young;
    PyGC_Head *gc = GC_NEXT(young);

    // 遍历链表中的每个对象
    while (gc != young) {
        //如果是root object
        if (gc_get_refs(gc)) {
            PyObject *op = FROM_GC(gc);
            traverseproc traverse = Py_TYPE(op)->tp_traverse;
            //将它标记为可达
            _PyObject_ASSERT_WITH_MSG(op, gc_get_refs(gc) > 0,
                                      "refcount is too small");
            //遍历被它引用的对象
            //调用visit_reachable将被引用对象标记为可达
            (void) traverse(op,
                    (visitproc)visit_reachable,
                    (void *)young);
            _PyGCHead_SET_PREV(gc, prev);
            gc_clear_collecting(gc);
            prev = gc;
        }
        else {
            //对于非root object，移到unreachable链表中
            prev->_gc_next = gc->_gc_next;
            PyGC_Head *last = GC_PREV(unreachable);
            last->_gc_next = (NEXT_MASK_UNREACHABLE | (uintptr_t)gc);
            _PyGCHead_SET_PREV(gc, last);
            gc->_gc_next = (NEXT_MASK_UNREACHABLE | (uintptr_t)unreachab
le);
            unreachable->_gc_prev = (uintptr_t)gc;
        }
        gc = (PyGC_Head*)prev->_gc_next;
    }
    young->_gc_prev = (uintptr_t)prev;
}
```

如果一个对象可达，Python 还通过 tp_traverse 逐个遍历它引用的对象（因为它们也是可达的），并调用 visit_reachable 函数进行标记：

```
static int
visit_reachable(PyObject *op, PyGC_Head *reachable)
{
    if (!PyObject_IS_GC(op)) {
        return 0;
    }

    PyGC_Head *gc = AS_GC(op);
    const Py_ssize_t gc_refs = gc_get_refs(gc);

    // 忽略掉1代和2代、以及没有被gc跟踪的对象
    if (gc->_gc_next == 0 || !gc_is_collecting(gc)) {
        return 0;
    }

    if (gc->_gc_next & NEXT_MASK_UNREACHABLE) {
        //对于已经被挪到unreachable链表中的对象
        //将其再次挪动到原来的链表
        PyGC_Head *prev = GC_PREV(gc);
        PyGC_Head *next = (PyGC_Head*)(gc->_gc_next & ~NEXT_MASK_UNREACH
ABLE);
        _PyObject_ASSERT(FROM_GC(prev),
                         prev->_gc_next & NEXT_MASK_UNREACHABLE);
        _PyObject_ASSERT(FROM_GC(next),
                         next->_gc_next & NEXT_MASK_UNREACHABLE);
        prev->_gc_next = gc->_gc_next;  // copy NEXT_MASK_UNREACHABLE
        _PyGCHead_SET_PREV(next, prev);

        gc_list_append(gc, reachable);
        gc_set_refs(gc, 1);
    }
    else if (gc_refs == 0) {
        //对于还没有处理的对象，恢复其gc_refs
        gc_set_refs(gc, 1);
    }
    else {
        _PyObject_ASSERT_WITH_MSG(op, gc_refs > 0, "refcount is too smal
l");
    }
    return 0;
}
```

总结一下就是：如果被引用的对象的引用计数为 0 ，就将它的引用计数设为 1 ，之后它将在 move_unreachable 函数中被遍历到并设为可达；如果被引用的对象被临时移入 unreachable 链表，同样将它的引用计数设为 1 ，并从 unreachable 链表移回原链表尾部，之后同样在 move_unreachable 函数中被遍历到并设为可达。

当 move_unreachable 完成之后，最初的一条链表就被切分成了两条链表，在 unreachable 链表中，就是我们发现的垃圾对象，是垃圾回收的目标。

但是等一等，在 unreachable 链表中，所有的对象都可以安全回收吗？其实，垃圾回收在清理对象的时候，默认是会清理的。但一旦当该对象的类对象里面定义了 __del__ 函数，那么在清理该对象的时候就会调用 __del__，因此也叫析构函数，这是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。

在Python3中，即使我们重写了也没事，因为Python会把含有 __del__ 方法的对象都统统移动到一个名为 garbage 的 PyListObject 对象中。

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，前面我们已经阐述了如何打破循环引用的办法，下面来看看具体的销毁过程。

```
//Modules/gcmodule.c
static inline int
gc_list_is_empty(PyGC_Head *list)
{
    return (list->_gc_next == (uintptr_t)list);
}

static void
delete_garbage(struct _gc_runtime_state *state,
```

```
                PyGC_Head *collectable, PyGC_Head *old)
{
    assert(!PyErr_Occurred());

    while (!gc_list_is_empty(collectable)) {
        PyGC_Head *gc = GC_NEXT(collectable);
        PyObject *op = FROM_GC(gc);

        _PyObject_ASSERT_WITH_MSG(op, Py_REFCNT(op) > 0,
                                  "refcount is too small");

        if (state->debug & DEBUG_SAVEALL) {
            assert(state->garbage != NULL);
            if (PyList_Append(state->garbage, op) < 0) {
                PyErr_Clear();
            }
        }
        else {
            inquiry clear;
            if ((clear = Py_TYPE(op)->tp_clear) != NULL) {
                Py_INCREF(op);
                (void) clear(op);
                if (PyErr_Occurred()) {
                    _PyErr_WriteUnraisableMsg("in tp_clear of",
                                              (PyObject*)Py_TYPE(op));
                }
                Py_DECREF(op);
            }
        }
        if (GC_NEXT(collectable) == gc) {
            /* object is still alive, move it, it may die later */
            gc_list_move(gc, old);
        }
    }
}
```

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对象中引用的对象的引用计数值，从而打破完成循环的最终目标。还是以PyListObject为例：

```
//listobject.c
static int
_list_clear(PyListObject *a)
{
    Py_ssize_t i;
    PyObject **item = a->ob_item;
    if (item != NULL) {
        i = Py_SIZE(a);
        //将ob_size调整为0
        Py_SIZE(a) = 0;
        //ob_item是一个二级指针，本来指向一个数组的指针
        //现在指向为NULL
        a->ob_item = NULL;
        //容量也设置为0
        a->allocated = 0;
        while (--i >= 0) {
            //数组里面元素也全部减少引用计数
            Py_XDECREF(item[i]);
        }
        //释放数组
        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_garbage的old参数)中。这是由于进行clear动作时，如果成功进行，通常一个对象会把自己从垃圾回收机制维护的链表中摘除(也就是这里的可收集链表)。

但由于某些原因，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到reachable链表中。

然后我们知道当对象被销毁时，肯定要调用析构函数，我们在上面看到了_list_clear。假设是调用了lst3的_list_clear，那么不好意思，接下来是要调用lst4的析构函数。因为lst3和lst4存在循环引用，所以调用了lst3的_list_clear会减少lst4的引用计数。

由于这两位老铁都被删除了，还惺惺相惜赖在内存里面不走，所以将lst4的引用计数减少1之后，只能归于湮灭了。然后会调用其 list_dealloc，注意：这时候调用的是lst4的list_dealloc。

```
//listobjct.c
static void
list_dealloc(PyListObject *op)
{
```

```
    Py_ssize_t i;
    //从可收集链表中移除
    PyObject_GC_UnTrack(op);
    Py_TRASHCAN_BEGIN(op)
    if (op->ob_item != NULL) {
        //依次遍历，减少内部元素的引用计数
        i = Py_SIZE(op);
        while (--i >= 0) {
            Py_XDECREF(op->ob_item[i]);
        }
        //释放内存
        PyMem_FREE(op->ob_item);
    }
    //缓冲池机制
    if (numfree < PyList_MAXFREELIST && PyList_CheckExact(op))
        free_list[numfree++] = op;
    else
        Py_TYPE(op)->tp_free((PyObject *)op);
    Py_TRASHCAN_END(op)
}
```

我们知道调用lst3的_list_clear，减少内部元素引用计数的时候，会导致lst4引用计数为0。而一旦lst4的引用计数为0，那么是不是也要执行和lst3一样的_list_clear动作呢？然后会发现lst3的引用计数也为0了，因此lst3也会被销毁，准确的说是指向的对象被销毁。

循环引用，彼此共生，销毁之路，怎能独自前行？最终lst3和lst4都会执行内部的list_dealloc，释放内部元素，调整参数，当然还有所谓的缓存池机制等等。总之如此一来，lst3和lst4指向的对象就都被安全地回收了。

虽然有很多对象挂在垃圾收集机制监控的链表上，但大部分时间都是引用计数在维护这些对象，只有引用计数无能为力的循环引用，垃圾收集机制才会起到作用。

这里没有把引用计数看成垃圾回收机制的一种，事实上，如果不是循环引用的话，那么垃圾回收完全不用出马。因为没有循环引用的话，垃圾回收的作用相当于只是将对象标记为可达，并移入下一代链表。

所以挂在垃圾回收监控链表上的对象都是引用计数不为0的，如果为0早被引用计数机制干掉了。

而引用计数不为0的情况也有两种：一种是被程序使用的对象，二是循环引用中的对象。被程序使用的对象是不能被回收的，所以垃圾回收只能处理那些循环引用的对象。

这里多提一句，可收集对象链表中的对象越多，那么垃圾回收发动一次的开销就越大。假设有一个类的实例对象，显然它也是需要被 GC 跟踪的，但如果我们能保证这个对象一定不会发生循环引用，那么可不可以不让它参与 GC 呢？因为不会发生循环引用，GC 检测也只是在做无用功，这样还不如不检测。

答案是可以的，当我们写 C 扩展的时候可以这么做，但是纯 Python 代码不行，解释器没有在 Python 的层面将这一特性暴露出来。因为解释器并不知道实际会不会产生循环引用，所以只要是有能力产生循环引用的 container 对象，统统会被 GC 跟踪，也就是会被挂在可收集对象链表上。而我们在用 C 或 Cython 写扩展时是可以实现的，如果能够人为保证某个对象一定不会出现循环引用，那么可以不让它参与 GC，从而降低 GC 的成本。

对象的弱引用

这里再来提一个不常用的知识，就是对象的弱引用。默认情况下，引用都是强引用，会导致对象的引用计数加 1；而弱引用则不会导致对象的引用计数增加。

Python 的标准库里面有一个和弱引用相关的模块叫 weakref，我们通过介绍这个模块来学习一下弱引用。

```
import weakref

class RefObject:

    def __del__(self):
        print("del executed")

obj = RefObject()
print(obj)  # <__main__.RefObject object at 0x000001B7C573A5E0>
# 对象的弱引用要通过 weakref.ref 来创建
r = weakref.ref(obj)
# 如果是 r = obj，那么都会强引用指向的对象
# 但此时的 r 是弱引用，并且显示关联 RefObject
print(r)  # <weakref at 0x000...; to 'RefObject' at 0x000...>
```

```
15
16 # 对引用进行调用的话，即可得到原对象
17 print(r() is obj)  # True
18
19 # 删除 obj 会执行析构函数
20 del obj  # del executed
21
22 # 之前说过 r() 等价于 obj，但是obj被删除了，所以返回None
23 # 从这里返回 None 也能看出弱引用是不会增加引用计数的
24 print("r():", r())  # r(): None
25 # 打印弱引用，告诉我们状态已经变成了 dead
26 print(r)  # <weakref at 0x000001B7DCAE19A0; dead>
```

通过弱引用我们可以实现缓存的效果，当它弱引用的对象存在时，则弱引用可用；当对象不存在时，则返回 None，程序不会因此而报错。这个缓存本质上是一样的，也是一个有则用、无则重新获取的技术。

此外 weak.ref 的构造函数还可以接收一个可选的回调函数，删除引用所指向的对象时就会调用这个回调函数。

```
1  import weakref
2
3  class RefObject:
4
5      def __del__(self):
6          print("del executed")
7
8  obj = RefObject()
9  r = weakref.ref(obj, lambda ref: print("引用被删除了", ref))
10 del obj
11 print("r():", r())
12 """
13 del executed
14 引用被删除了 <weakref at 0x0000021A69681900; dead>
15 r(): None
16 """
```

回调函数接收一个参数，也就是死亡之后的弱引用。

清理弱引用时要对资源完成更健壮的管理，可以使用 finalize 将回调与对象关联。finalize 实例会一直保留（直到所关联的对象被删除），即使没有保留它的引用。

```
1  import weakref
2
3  class RefObject:
4
5      def __del__(self):
6          print("del executed")
7
8  def on_finalize(*args):
9      print(f"on_finalize", args)
10
11 obj = RefObject()
12 weakref.finalize(obj, on_finalize, "arg1", "arg2", "arg3")
13 del obj
14 """
15 del executed
16 on_finalize ('arg1', 'arg2', 'arg3')
17 """
```

finalize 的参数包括要跟踪的对象，对象被垃圾回收时要调用的 callback，以及传递给 callback 的参数。

有时候使用代理比使用弱引用更方便，使用代理可以像使用原对象一样，而且不要求在访问对象之前先调用代理。这说明，可以将代理传递到一个库，并且这个库并不知道它接收的是一个代理，而不是一个真正的对象。

```
1  import weakref
2
3  class RefObject:
4
5      def __init__(self, name):
6          self.name = name
7
8      def __del__(self):
9          print("del executed")
10
11 obj = RefObject("my obj")
12 r = weakref.ref(obj)
13 p = weakref.proxy(obj)
```

```
14
15 # 可以看到弱引用加上()才相当于原来的对象
16 # 而代理不需要，直接和原来的对象保持一致
17 print(obj.name)  # my obj
18 print(r().name)  # my obj
19 print(p.name)  # my obj
20
21 # 但是注意：弱引用在调用之后就是原对象，而代理不是
22 print(r() is obj)  # True
23 print(p is obj)  # False
24
25 del obj  # del executed
26
27 try:
28     # 删除对象之后，再调用弱引用，打印None
29     print(r())  # None
30     # 如果是使用代理，则会报错
31     print(p)
32 except Exception as e:
33     print(e)  # weakly-referenced object no longer exists
```

weakref.proxy 和 weakref.ref 一样，也可以接收一个额外的回调函数。此外我们还可以查看一个对象被弱引用的次数，以及弱引用组成的列表。

```
1  import weakref
2
3  class RefObject:
4
5      def __del__(self):
6          print("del executed")
7
8  obj = RefObject()
9  r1 = weakref.ref(obj)
10 r2 = weakref.ref(obj)
11 print(weakref.getweakrefcount(obj))  # 1
12
13 # 可能有人好奇为什么是 1
14 # 那是因为 r1 和 r2 指向的是同一个对象
15 print(r1 is r2)  # True
16
17 p1 = weakref.proxy(obj)
18 p2 = weakref.proxy(obj)
19 # 此时变成了 2，被代理的话也可以看成是被弱引用了
20 print(weakref.getweakrefcount(obj))  # 2
21 print(p1 is p2)  # True
22
23 # 查看 obj 被弱引用的对象
24 print(weakref.getweakrefs(obj))
25 # [<weakref at 0x0000...; to 'RefObject' at 0x000001EFC060A5E0>,
26 #  <weakproxy at 0x0000... to RefObject at 0x000001EFC060A5E0>]
27
28 del obj
29 """
30 del executed
31 """
```

我们可以创建一个 key 为弱引用或者 value 为弱引用的字典。

```
1  class A:
2
3      def __del__(self):
4          print("__del__")
5
6  a = A()
7  # 创建一个普通字典
8  d = {}
9  # 由于 a 作为了字典的 key
10 # 那么 a 指向对象的引用计数会加 1
11 d[a] = "xxx"
12
13 # 删除 a，对对象无影响，不会触发析构函数
14 del a
15 print(d)
16 """
17 {<__main__.A object at 0x000002092669A5E0>: 'xxx'}
18 __del__
19 """
20 # 最后打印的 __del__ 是程序结束了，将对象回收时打印的
```

但如果是 key 为弱引用的字典，就不一样了。

```python
import weakref

class A:

    def __del__(self):
        print("__del__")

a = A()

# 创建一个弱引用字典，它的 api 和普通字典一样
d = weakref.WeakKeyDictionary()
print("d:", d)

# 此时 a 指向对象的引用计数不会增加
# 因为 d 指向的是一个 key 为弱引用的字典
d[a] = "xxx"
print("before del a:", list(d.items()))

# 删除 a，对象会被回收
del a
print("after del a:", list(d.items()))
"""
d: <WeakKeyDictionary at 0x2a2dc048700>
before del a: [(<__main__.A object at 0x000002A2DC15A5E0>, 'xxx')]
__del__
after del a: []
"""
# 并且我们看到也没有 a 这个 key 了
```

注意：这里只是对 key 进行弱引用，但是 value 不会；如果想对 value 进行弱引用的话，我们需要使用 WeakValueDictionary，方法是一样的。当然还有 WeakSet，用法也是类似的，只不过它是一个集合，放入该集合中的元素不会增加引用计数。

当我们自定义一个类的时候，如果为了省内存，那么会不给它的实例对象赋予 __dict__属性。因为字典使用的是哈希表，这是一个空间换时间的数据结构。如果想省内存的话，那么我们通常的做法是在类里面定义 __slots__，然后其实例对象就不会再有 __dict__ 属性了。

不过这会带来一个问题：

```python
class A:

    __slots__ = ("name", "age")

    def __init__(self):
        self.name = "古明地觉"
        self.age = 16

import weakref
a = A()

try:
    weakref.ref(a)
except Exception as e:
    print(e)  # cannot create weak reference to 'A' object

try:
    weakref.proxy(a)
except Exception as e:
    print(e)  # cannot create weak reference to 'A' object

try:
    d = weakref.WeakSet()
    d.add(a)
except Exception as e:
    print(e)  # cannot create weak reference to 'A' object
```

此时我们发现，A 的实例对象没办法被弱引用，因为我们指定了 __slots__。那么要怎么解决呢？很简单，直接在 __slots__ 里面加一个属性就好了。

```python
class A:

    # 多指定一个__weakref__，表示支持弱引用
    __slots__ = ("name", "age", "__weakref__")

    def __init__(self):
        self.name = "夏色祭"
        self.age = 16


```

```
import weakref
a = A()

weakref.ref(a)
weakref.proxy(a)
d = weakref.WeakSet()
d.add(a)
```

没有报错，可以看到此时就支持弱引用了。

弱引用是怎么实现的？

弱引用看起来很神奇，但实现起来比想象中简单的多。首先弱引用保存了原对象的指针和一个回调函数（但是不增加对象的引用计数），而在弱引用创建之后，会组织成一个列表保存在原对象中。

```
typedef struct {
    PyObject_HEAD
    PyObject *func_code;        /* A code
    PyObject *func_globals;     /* A dicti
    PyObject *func_defaults;    /* NULL or
    PyObject *func_kwdefaults;  /* NULL or
    PyObject *func_closure;     /* NULL or
    PyObject *func_doc;         /* The __d
    PyObject *func_name;        /* The __n
    PyObject *func_dict;        /* The __d
    PyObject *func_weakreflist; /* List of
    PyObject *func_module;      /* The __m
    PyObject *func_annotations; /* Annotat
    PyObject *func_qualname;    /* The qua
    vectorcallfunc vectorcall;

    /* Invariant:
     *     func_closure contains the bindi
     *     PyTuple_Size(func_closure) == P
     *     (func_closure may be NULL if Py
     */
} PyFunctionObject;
```


以函数为例，观察它的结构体，我们看到内部有一个 func_weakreflist，这个字段就负责保存弱引用组成的列表。当原对象销毁之后，虚拟机会遍历这个列表，清理弱引用保存的指针并执行回调函数（如果设置了）。

但是奇怪了，我们之前在介绍整数、浮点数等结构的时候，没有看到类似 weakreflist 这样的字段啊。是的，所以它们无法被弱引用。

```
import weakref

try:
    r1 = weakref.ref([])
except TypeError as e:
    print(e)
# TypeError: cannot create weak reference to 'list' object
```


关于弱引用的底层实现，可以通过以下几个文件查看：

```
Include/weakrefobject.h
Objects/weakrefobject.c
Modules/clinic/_weakref.c.h
Modules/_weakref.c
```

这里就不再多说了。

Python 的 gc 模块

这个 gc 模块之前提到过，它是用 C 编写的，源码对应 Modules/gcmodule.c，当 Python 编译好时，就内嵌在解释器里面了。我们可以导入它，但是在 Python 安装目录里面是看不到的。

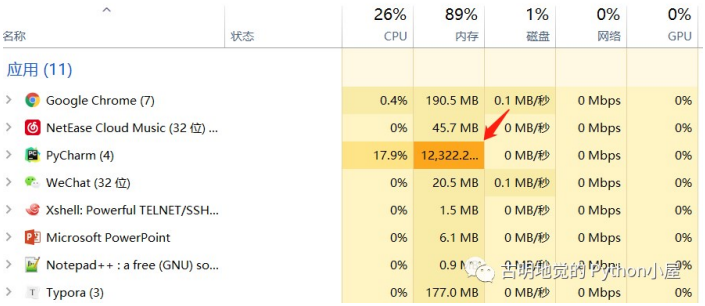
gc.enable()：开启垃圾回收

这个函数表示开启垃圾回收机制，默认是自动开启的。

gc.disable()：关闭垃圾回收

```python
import gc

class A:
    pass
# 关掉gc
gc.disable()


while True:
    a1 = A()
    a2 = A()
    # 此时内部出现了循环引用
    a1.__dict__["attr"] = a2
    a2.__dict__["attr"] = a1

    # 由于循环引用，所以del a1, a2无效
    # 因为光靠引用计数是删不掉的
    # 需要垃圾回收，但是我们给关闭了
    del a1, a2
```

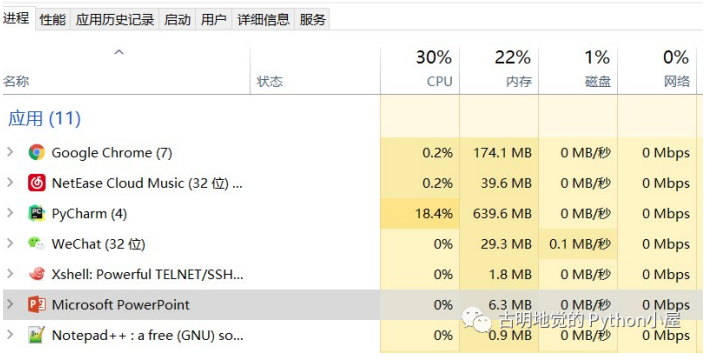
看一下内存利用率：



无限循环，并且每次循环都会创建新的对象，最终导致内存无限增大。

```python
import gc

class A:
    pass


# 关掉 GC
gc.disable()

while True:
    a1 = A()
    a2 = A()
```

再次查看内存利用率：



这里我们关闭了 GC，但是每一次循环都会指向一个新的对象，而之前的对象由于没有人指向了，那么引用计数为0，直接就被引用计数机制干掉了，内存会一直稳定，不会出现增长。

所以即使关闭了 GC，但是对于那些引用计数为0的，该删除还是会删除的。因此引用计数很简单，就是按照对应的规则该加1加1，该减1减1，一旦为0，直接销毁。而当出现循环引用的时候，才需要 GC 闪亮登场。因此这里虽然关闭了 GC，不过没有循环引用，所以没事。

但上一个例子出现了循环引用，而引用计数机制只会根据引用计数来判断，发现引用计数不为0，所以就一直傻傻地不回收，程序又一直创建新的对象，最终导致内存越用越多。如果上一个例子开启了GC，那么就会通过标记-清除的方式将产生循环引用的对象的引用计数减1，而引用计数机制发现引用计数为0了，就会将对象回收掉。

gc.isenabled()：判断 GC 是否开启

```python
import gc
print(gc.isenabled())  # True
gc.disable()
print(gc.isenabled())  # False
```

默认是开启的。

gc.collect()：立刻触发垃圾回收

我们说，垃圾回收的触发是有条件的，但是这个函数可以强制触发垃圾回收。

```python
import gc

class A:
    def __init__(self, name):
        self.name = name
    def __del__(self):
        print(f"{self.name} 被删除了")

a1 = A("古明地觉")
a2 = A("古明地恋")

# 发生循环引用
a1.obj = a2
a2.obj = a1

# 无事发生
del a1, a2

print("------")
# 强行触发垃圾回收，参数表示指定的"代"
# 目前对象显然是在零代链表上，应该清理零代
# 但是清理一代和二代也可以，因为会顺带清理零代
gc.collect(0)
"""
------
古明地觉 被删除了
古明地恋 被删除了
"""
```

gc.get_threshold()：返回每一代的阈值

```python
import gc

print(gc.get_threshold())  # (700, 10, 10)
```

gc.set_threshold()：设置每一代的阈值

```python
import gc

gc.set_threshold(1000, 100, 100)
print(gc.get_threshold())  # (1000, 100, 100)
```

gc.get_count()：查看每一代的值达到了多少

```python
import gc

# 你的结果可能和我这里不一样
print(gc.get_count())  # (675, 8, 6)
```

gc.get_stats()：返回每一代的具体信息

```python
from pprint import pprint
import gc

pprint(gc.get_stats())
"""
[{'collected': 680, 'collections': 74, 'uncollectable': 0},
 {'collected': 153, 'collections': 6, 'uncollectable': 0},
 {'collected': 0, 'collections': 0, 'uncollectable': 0}]
"""
```

gc.get_objects()：返回被垃圾回收器追踪的所有对象，一个列表

```python
import gc

print(gc.get_objects())
```

不要执行，打印的内容会非常多，因为有大量的 container 对象在被跟踪。

gc.is_tracked(obj)：查看对象obj是否被垃圾回收器追踪

```python
import gc

a = 1
b = []

print(gc.is_tracked(a))  # False
print(gc.is_tracked(b))  # True

# 只有那些有能力产生循环引用的对象才会被垃圾回收器跟踪
```

gc.get_referrers(obj)：返回所有引用了obj的对象
gc.get_referents(obj)：返回所有被obj引用了的对象
gc.freeze()：冻结所有被垃圾回收器跟踪的对象并在以后的垃圾回收中不被处理
gc.unfreeze()：取消所有冻结的对象，让它们继续参数垃圾回收
gc.get_freeze_count()：获取冻结的对象的个数

```python
import gc
# 不需要参数，会自动找到被垃圾回收器跟踪的对象
gc.freeze()
# 说明有很多内置对象在被跟踪，但被我们冻结了
print(gc.get_freeze_count())  # 24397

b = []
gc.freeze()
# 只要打印的结果比上面多 1 就行
print(gc.get_freeze_count())  # 24398

# 取消冻结
gc.unfreeze()
print(gc.get_freeze_count())  # 0
```

gc.get_debug()：获取debug级别

```python
import gc
print(gc.get_debug())  # 0
```

gc.set_debug()：设置debug级别

```python
import gc

"""
DEBUG_STATS - 在垃圾收集过程中打印所有统计信息
DEBUG_COLLECTABLE - 打印发现的可收集对象
DEBUG_UNCOLLECTABLE - 打印unreachable对象(除了uncollectable对象)
DEBUG_SAVEALL - 将对象保存到gc.garbage(一个列表)里面，而不是释放它
DEBUG_LEAK - 对内存泄漏的程序进行debug (everything but STATS).
"""
class A:
    pass

class B:
    pass

a = A()
b = B()

gc.set_debug(gc.DEBUG_STATS | gc.DEBUG_SAVEALL)
print(gc.garbage)  # []
a.b = b
b.a = a
del a, b
gc.collect()  # 强制触发垃圾回收
# 下面都是自动打印的
"""
gc: collecting generation 2...
gc: objects in each generation: 123 3732 20563
gc: objects in permanent generation: 0
gc: done, 4 unreachable, 0 uncollectable, 0.0000s elapsed
gc: collecting generation 2...
gc: objects in each generation: 0 0 24249
gc: objects in permanent generation: 0
gc: done, 0 unreachable, 0 uncollectable, 0.0150s elapsed
gc: collecting generation 2...
gc: objects in each generation: 525 0 23752
gc: objects in permanent generation: 0
gc: done, 7062 unreachable, 0 uncollectable, 0.0000s elapsed
```

```
gc: collecting generation 2...
gc: objects in each generation: 0 0 21941
gc: objects in permanent generation: 0
gc: done, 4572 unreachable, 0 uncollectable, 0.0000s elapsed
"""
print(gc.garbage)
"""
[<__main__.A object at 0x0000020CFDB50250>,
<__main__.B object at 0x0000020CFDB50340>,
{'b': <__main__.B object at 0x0000020CFDB50340>},
{'a': <__main__.A object at 0x0000020CFDB50250>}]
"""
```

以上就是 gc 模块相关的内容，对于一般的业务开发来说，使用频率不高。

Python采用了最经典的(最土的)引用计数机制来作为自动管理内存的方案，但由于存在循环引用，于是又引入标记-清除、分代收集，进行了极大的完善。

尽管引用计数机制需要花费额外的开销来维护引用计数，但是在如今这个年代，这点开销算个啥。而且引用计数也有好处，不然早就随着时代的前进而被扫进历史的垃圾堆里面了。至于好处有两点：第一，引用计数机制很方便，很直观，由于大部分对象都不出现循环引用，所以引用计数机制能够直接解决，不需要什么复杂的操作；第二，引用计数将对象回收的开销分摊在了整个运行时，这对 Python 的响应是有好处的。

当然内存管理和垃圾回收是一门非常精细和繁琐的技术，有兴趣的话可以自己大刀阔斧地冲进 Python 的源码中自由翱翔。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》完结撒花，个人想说的一些话

下一篇 ›
《源码探秘 CPython》95. Python 的分代收集技术