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

原创 古明地觉 古明地觉的编程教室 2022-05-19 03:36 发表于北京

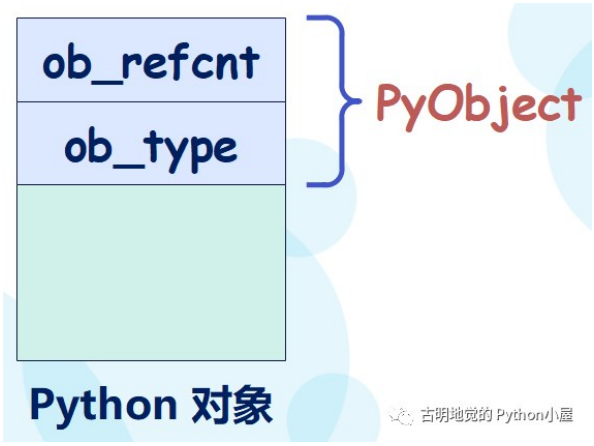


收录于合集
#CPython
97个 >

## 引用计数

现在绝大部分高级语言都自带了垃圾回收机制，将开发者从繁重的内存管理工作中解放了出来。Python 作为一门高级语言，同样自带了垃圾回收，然而 Python 的垃圾回收和 Java，C# 等语言有一个很大的不同，那就是 Python 中大多数对象的生命周期是通过对象的引用计数来管理的。

Python 对象的基石 PyObject 有两个属性，一个是该对象的类型，还有一个就是引用计数。




所以从广义上讲，引用计数也算是一种垃圾回收机制，而且是一种最简单最直观的垃圾回收机制。尽管需要一个值来维护引用计数，但是引用计数有一个最大的优点：实时性。任何内存，一旦没有指向它的引用，那么就会被回收。而其它的垃圾回收技术必须在某种特定条件下(比如内存分配失败)才能进行无效内存的回收。

引用计数机制会带来一些额外开销，因为要时刻维护引用计数的值，并且与 Python 运行中所进行的内存分配、释放、引用赋值的次数成正比。这一点和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，因为它们带来的额外开销只和内存数量有关，至于多少人引用了这块内存则不关心。

因此为了与引用计数搭配，在内存的分配和释放上获得最高的效率，Python 设计了大量的缓存池机制，比如小整数对象池、字符串的 intern 机制，列表的 freelist 缓存池等等，这些大量使用的面向特定对象的缓存机制弥补了引用计数的软肋。

那么引用计数什么时候会增加，什么时候会减少呢？

### 引用计数加1

- 对象被创建：a=1；
- 对象被引用：b=a；
- 对象的引用作为参数传到一个函数中，func(a)；
- 对象的引用作为列表、元组等容器里面的元素；

### 引用计数减1

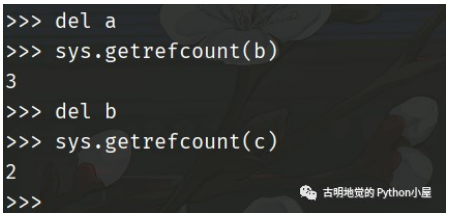
- 指向对象的变量(符号)被显式的销毁：del a；
- 对象的引用指向了其它的对象：a=2；
- 对象的引用离开了它的作用域，比如函数的局部变量，在函数执行完毕的时候会被销毁(如果没有获取栈帧的话)；
- 对象的引用所在的容器被销毁，或者从容器中删除等等；

### 查看引用计数

查看一个对象的引用计数，可以通过 sys.getrefcount(obj)，但是由于作为 getrefcount 这个函数的参数，所以引用计数会多 1。

```
>>> import sys
>>>
>>> a = "古明地觉"
>>> sys.getrefcount(a)
2
>>> b = a
>>> sys.getrefcount(a)
3
>>> c = b
>>> sys.getrefcount(a)
```


Python 的变量只是一个和对象绑定的符号，在底层都是 PyObject * 泛型指针，b = a 在底层则表示把指针变量 a 存储的地址拷贝给了指针变量 b，所以此时 b 也指向了 a 指向的对象。因此字符串对象的引用计数就会加 1，此时变为 2。




而每当减少一个引用，引用计数就会减少 1。尽管我们用 sys.getrefcount 得到的结果是 2，但是当这个函数执行完，由于局部变量的销毁，其实结果已经变成了 1。

因此引用计数机制非常简单，就是多一个引用，引用计数加 1；少一个引用，引用计数减 1；如果引用计数为 0，说明对象已经没有人引用了，那么就直接销毁。这就是引用计数机制的实现原理，简单且直观。

从目前来看，引用计数机制貌似还挺不错的，虽然需要额外用一个字段（ob_refcnt）来时刻维护引用计数的值，但对于现在的 CPU 和内存来说，完全不是事儿。最主要的是，引用计数机制真的很简单、很直观。

但可惜的是，它存在一个致命的缺陷，这一缺陷几乎将引用计数在垃圾回收机制中判了"死刑"，这一缺陷就是"循环引用"。而且也正是因为"循环引用"这个致命伤，导致在狭义上并不把引用计数看成是垃圾回收机制的一种。

```
lst1 = []
lst2 = []

lst1.append(lst2)
lst2.append(lst1)

del lst1, lst2
```

初始的时候，lst1 和 lst2 指向的对象的引用计数都为 1，而在 lst1.append(lst2) 之后，lst2 指向对象的引用计数就变成了 2；同理，lst2.append(lst1) 导致 lst1 指向对象的引用计数也变成了 2。

因此当我们 del lst1, lst2 的时候，引用计数会从 2 变成 1，由于不为 0，所以 lst1 和 lst2 指向的对象都不会被回收，但我们是希望回收的。所以此时我们就说 lst1 和 lst2 指向的对象之间发生了循环引用，如果只有引用计数机制的话，那么显然这两者是回收不了的。

为了更直观的观察到这个现象，我们用 ctypes 来模拟一下这个过程。

```
from ctypes import *
import gc

class PyObject(Structure):

    _fields_ = [
        ("ob_refcnt", c_ssize_t),
        ("ob_type", c_void_p)
    ]

# 创建两个列表
lst1 = []
lst2 = []

# 获取它们的 PyObject *
# 注意:这一步不会改变对象的引用计数
py_lst1 = PyObject.from_address(id(lst1))
py_lst2 = PyObject.from_address(id(lst2))

# 显然初始的时候，引用计数都为 1
print(py_lst1.ob_refcnt)  # 1
print(py_lst2.ob_refcnt)  # 1

# lst2 作为列表的一个元素
# 所以指向对象的引用计数加 1
lst1.append(lst2)
print(py_lst1.ob_refcnt)  # 1
print(py_lst2.ob_refcnt)  # 2

# lst1 作为列表的一个元素
# 所以指向对象的引用计数加 1
lst2.append(lst1)
print(py_lst1.ob_refcnt)  # 2
```

```
print(py_lst2.ob_refcnt)  # 2

# 删除 lst1、lst2
# 发现引用计数还为 1
del lst1, lst2
print(py_lst1.ob_refcnt)  # 1
print(py_lst2.ob_refcnt)  # 1

# 显然我们希望的结果是引用计数为 0
# 但是现在不为 0，原因就是发生了循环引用
# 程序如果一直运行下去，就会出现内存泄露
# 于是 Python 的垃圾回收就登场了
# 发动一次 gc
gc.collect()
print(py_lst1.ob_refcnt)  # 0
print(py_lst2.ob_refcnt)  # 0

# nice，我们看到此时引用计数都变成了 0
# 此时两个对象也都会被回收
```

这里提前给出结论，一个对象是否被回收只取决于它的引用计数（ob_refcnt）是否为 0，只要为 0 就回收，不为 0 则存活。但由于对象之间会发生循环引用，导致引用计数失效，所以严格意义上不能把引用计数机制看成是垃圾回收机制的一种。

于是 Python 除了引用计数机制之外，还提供了真正的垃圾回收技术（标记-清除和分代收集），来弥补这一漏洞。其工作方式也很简单，就是找出那些发生循环引用的对象，然后将循环引用导致增加的引用计数再给减掉，这样对象的引用计数不就正常了吗？

比如上面代码中 lst1 和 lst2 指向的对象，当 gc 触发时，垃圾回收器发现循环引用导致它们的引用计数增加了 1，于是会再将它们的引用计数减去 1，然后变成 0。而引用计数机制发现引用计数变为 0，便会将对象回收。

所以对象回收与否，完全是由它的引用计数决定的，垃圾回收只是在给引用计数机制擦屁股。如果程序不出现循环引用，那么引用计数机制足矣；但当出现了循环引用，垃圾回收机制就要出来解决这一点，将循环引用造成的影响抵消掉，从而让引用计数机制能够正常工作。

那么接下来的重点，就是要看看 Python 的垃圾回收是怎么解决循环引用的？

## 三色标记模型

无论何种垃圾回收机制，一般都分为两个阶段：垃圾检测和垃圾回收。垃圾检测是从所有已经分配的内存中区别出可回收和不可回收的内存，而垃圾回收则是使操作系统重新掌握垃圾检测阶段所标识出来的可回收内存块（或者缓存起来）。所以垃圾回收，并不是说直接把这块内存的数据清空了，而是将使用权重新交给了操作系统，不会自己霸占了。

而 Python 的垃圾回收采用的标记-清除和分代收集，分代收集我们一会再说，先来看看标记-清除是怎么实现的，并为这个过程建立一个三色标记模型，Python 的垃圾回收正是基于这个模型完成的。

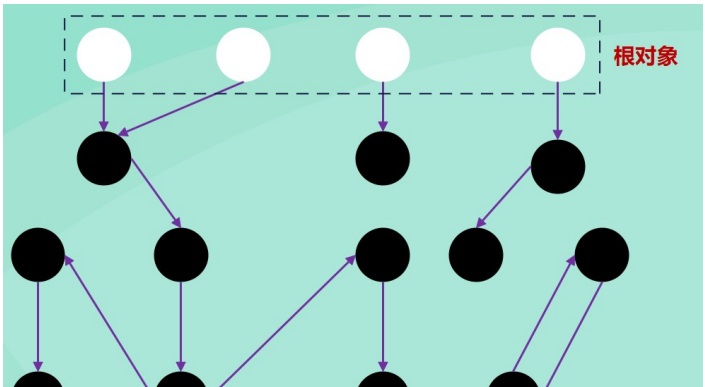
**从具体的实现上来讲，标记-清除方法同样遵循垃圾回收的两个阶段，其简要过程如下：**

1）寻找根对象（root object）集合，所谓的 root object 集合就是一些全局引用和函数栈的引用，这些引用指向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进行探索，如果能到达某个对象 A，则称 A 是可达的（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，所有的对象被分为了可达的（reachable）和不可达的（unreachable）。所有可达对象都必须予以保留，而不可达对象所占用的内存将被回收；

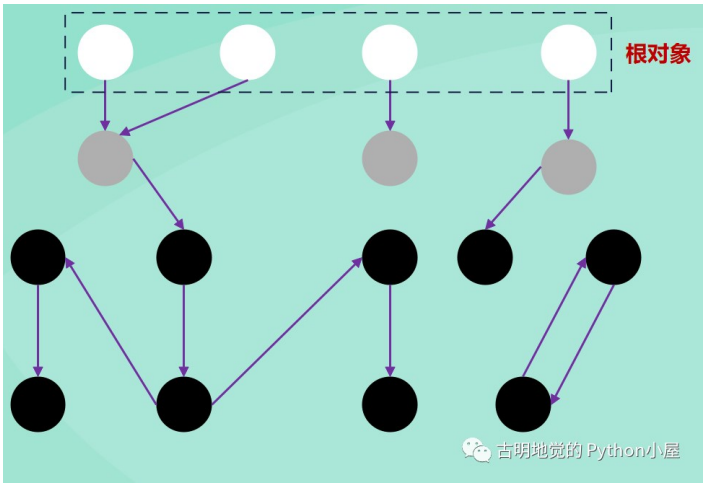
我们用图形的方式，来描述一下。



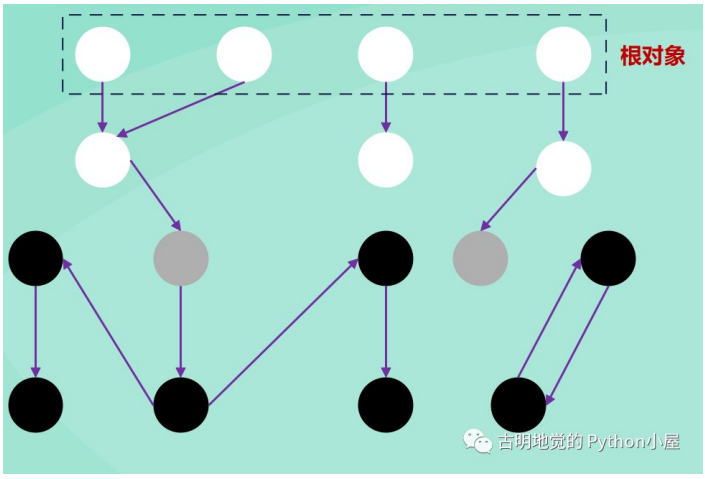

在垃圾回收动作被激活之前，系统中所分配的对象之间的引用关系组成了一张有向图，对象是图中的节点，而对象间的引用则是节点和节点之间的连线。

白色的节点表示被引用的活跃对象，也就是可达；黑色的节点表示需要回收、但是由于循环引用而无法回收的垃圾对象，也就是不可达。由于根对象本身就是可达的，所以它被标记为白色。然后我们假设除了根对象之外都是不可达的，所以下面都标记成了黑色，至于它到底是不是黑色，需要通过遍历才知道。

然后我们开始遍历了，显然从根对象开始遍历，根对象是可达的，被根对象引用的对象同样也是可达的。所以当我们从根对象出发，沿着引用关系遍历，能够遍历到的对象都是可达的，我们将其标记为灰色。

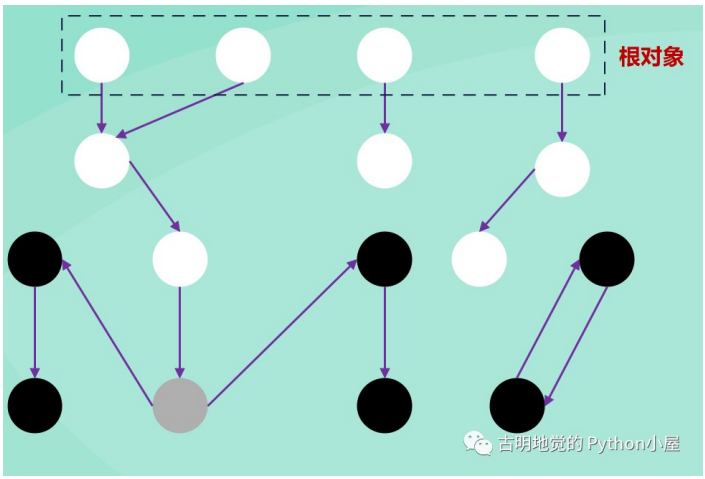



咦，可达的不应该是白色吗？为啥又冒出来一个灰色。因为被根对象直接引用的对象，可能还会继续引用其它对象，我们需要一层一层遍历。因此暂时将它标记为灰色，但很明显灰色节点也是可达的。

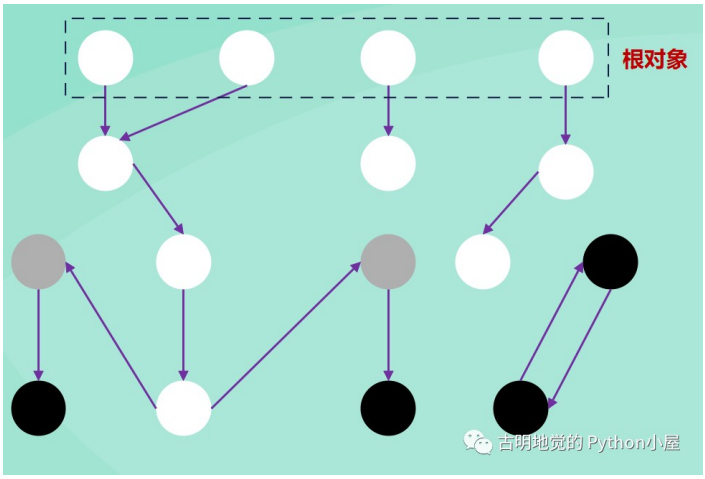



我们看到当灰色直接引用的对象检测完毕时，灰色就被标记成了白色，然后它所引用的对象也从黑色变成了灰色。所以下面该谁了呢？显然从新的被标记成灰色的对象开始继续往下找，就是一层一层遍历嘛。

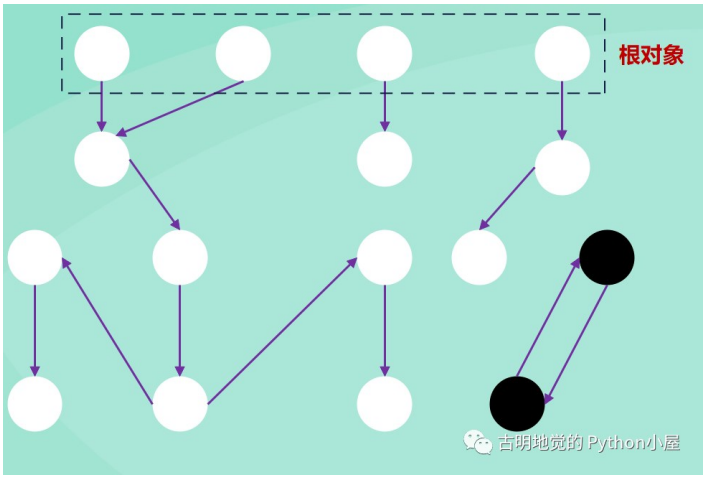
> 根对象之外的可达对象会先由黑色变成灰色，当其直接引用的对象被检查完毕时，再由灰色变成白色。与此同时，其直接引用的对象也从黑色变成灰色。




凡是被引用到了的对象，都是可达的，不能删除。




如果从根集合开始，按照广度优先的策略进行搜索的话，那么不难想象，灰色节点集合就如同波纹一样不断向外扩散。凡是被灰色波纹触碰到的就会变成白色，没有被触碰到的则还是原来的黑色。




遍历完所有的对象之后，说明垃圾检测阶段结束了。如果是黑色，说明是不可达的，会被回收；白色，这说明是可达的，不会被回收。

比如图中的两个黑色节点，从任何一个根节点出发都遍历不到它，所以它们是因为循环引用而无法被回收的垃圾对象。这时垃圾回收器会将它们的引用计数减一，所以上面说的回收并不是真的就回收了，而是减少它的引用计数。至于对象的回收，则是由引用计数机制发现对象的引用计数是否为 0、然后调用它的 tp_dealloc 实现的。

> 正如一开始说的那样，垃圾回收只是在给引用计数机制擦屁股。垃圾回收做的工作就是修正对象的引用计数，解决循环引用带来的问题；而对象的回收，则由引用计数机制负责。

以上就是垃圾回收中的标记-清除法，思想其实很简单，Python 内部也是采用这个办法来识别、回收垃圾对象。

由于 Python 中的对象都是分配在堆上的，根对象集合其实不太直观，Python是先通过一个算法找出根对象，然后再从根对象出发找到可达的活跃对象。

整体来说应该不难理解，下一篇来介绍分代收集。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》95. Python 的分代收集技术

下一篇 ›
《源码探秘 CPython》93. Python 是如何管理内存的？（下）