Python 垃圾回收机制

Garbage collection(GC)

现在的高级语言如 java，c#等，都采用了垃圾收集机制，而不再是 c，c++里用户自己管理维护内存的方式。自己管理内存极其自由，可以任意申请内存，但如同一把双刃剑，为大量内存泄露，悬空指针等 bug 埋下隐患。

对于一个字符串、列表、类甚至数值都是对象，且定位简单易用的语言，自然不会让用户去处理如何分配回收内存的问题。

python 里也同 java 一样采用了垃圾收集机制，不过不一样的是: python 采用的是引用计数机制为主，标记-清除和分代收集两种机制为辅的策略

1、引用计数机制

python 里每一个东西都是对象，它们的核心就是一个结构体：PyObject

```
typedef struct_object {
  int ob_refcnt;
  struct_typeobject *ob_type;
} PyObject;
```

PyObject 是每个对象必有的内容，其中 ob_*refcnt 就是做为引用计数。当一个对象有新的引用时，它的 ob*refcnt 就会增加，当引用它的对象被删除，它的 ob_refcnt 就会减少

```
#define Py_INCREF(op)   ((op)->ob_refcnt++) //增加计数
#define Py_DECREF(op)  //减少计数
    if (--(op)->ob_refcnt != 0)
        ;
    else
        __Py_Dealloc((PyObject *)(op))
```

当引用计数为 0 时，该对象生命就结束了。

引用计数机制的优点：

1.简单

2.实时性：一旦没有引用，内存就直接释放了。不用像其他机制等到特定时机。实时性还带来一个好处：处理回收内存的时间分摊到了平时。

引用计数机制的缺点：

1.维护引用计数消耗资源

2.循环引用

```
list1 = []
list2 = []
list1.append(list2)
list2.append(list1)
```

list1 与 list2 相互引用，如果不存在其他对象对它们的引用，list1 与 list2 的引用计数也仍然为 1，所占用的内存永远无法被回收，这将是致命的。

对于如今的强大硬件，缺点 1 尚可接受，但是循环引用导致内存泄露，注定 python 还将引入新的回收机制。(标记清除和分代收集)

GC 系统负责三个重要任务:

1、为新生成的对象分配内存

2、识别那些垃圾对象

3、从垃圾对象那回收内存。

如果将应用程序比作人的身体：所有你所写的那些优雅的代码，业务逻辑，算法，应该就是大脑。以此类推，垃圾回收机制应该是那个身体器官呢？

垃圾回收就象应用程序的心。像心脏为身体其他器官提供血液和营养物那样，垃圾回收器为你的应该程序提供内存和对象。如果心脏停跳，过不了几秒钟人就完了。如果垃圾回收器停止工作或运行迟缓,像动脉阻塞,你的应用程序效率也会下降，直至最终死掉。

2、Ruby 与 Python 垃圾回收

2.1 一个简单例子

运用实例一贯有助于理论的理解。下面是一个简单类，分别用 Python 和 Ruby 写成：

```python
class Node:
  def __init__(self, val):
    self.value = val

print(Node(1))
print(Node(2))
```

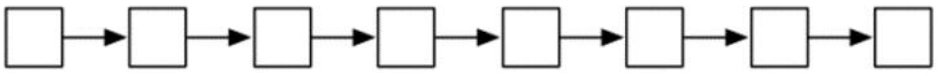
```ruby
class Node
  def initialize(val)
    @value = val
  end
end

p Node.new(1)
p Node.new(2)
```

Ruby 和 Python 在表达同一事物上真的只是略有不同。但是在这两种语言的内部实现上非常的相似。

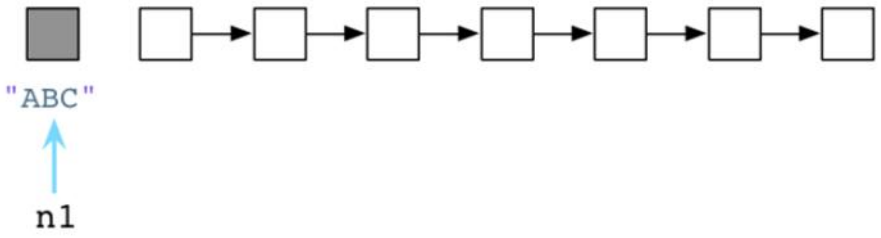
2.2 Ruby 的可用列表

当我们执行上面的 Node.new(1)时，Ruby 到底做了什么？Ruby 是如何为我们创建新的对象的呢？ 出乎意料的是它做的非常少。实际上，早在代码开始执行前，Ruby 就提前创建了成百上千个对象，并把它们串在链表上，名曰：可用列表。下图所示为可用列表的概念图：



想象一下每个白色方格上都标着一个"未使用预创建对象"。当我们调用 Node.new ,Ruby 只需取一个预创建对象给我们使用即可：

```ruby
n1 = Node.new("ABC")
```



这个简单的用链表来预分配对象的算法已经发明了超过 50 年，而发明人这是赫赫有名的计算机科学家 John McCarthy，一开始是用 Lisp 实现的。Lisp 不仅是最早的函数式编程语言，在计算机科学领域也有许多创举。其一就是利用垃圾回收机制自动化进行程序内存管理的概念。

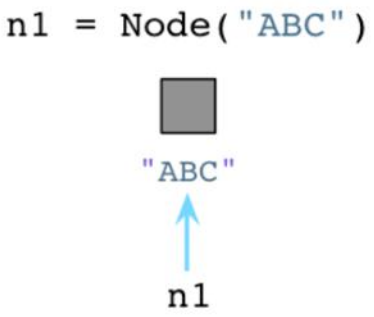
标准版的 Ruby，也就是众所周知的"Matz's Ruby Interpreter"(MRI),所使用的 GC 算法与 McCarthy 在 1960 年的实现方式很类似。无论好坏，Ruby 的垃圾回收机制已经 53 岁高龄了。像 Lisp 一样，Ruby 预先创建一些对象，然后在你分配新对象或者变量的时候供你使用。

2.3 Python 的对象分配

我们已经了解了 Ruby 预先创建对象并将它们存放在可用列表中。那 Python 又怎么样呢？

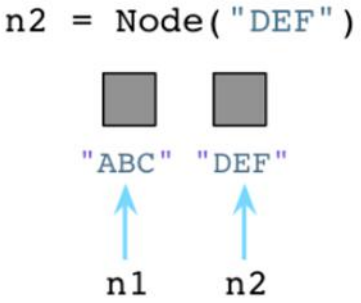
尽管由于许多原因 Python 也使用可用列表(用来回收一些特定对象比如 list)，但在为新对象和变量分配内存的方面 Python 和 Ruby 是不同的。

例如我们用 Pyhon 来创建一个 Node 对象：



与 Ruby 不同，当创建对象时 Python 立即向操作系统请求内存。(Python 实际上实现了一套自己的内存分配系统，在操作系统堆之上提供了一个抽象层。但是我今天不展开说了。)

当我们创建第二个对象的时候，再次像 OS 请求内存：

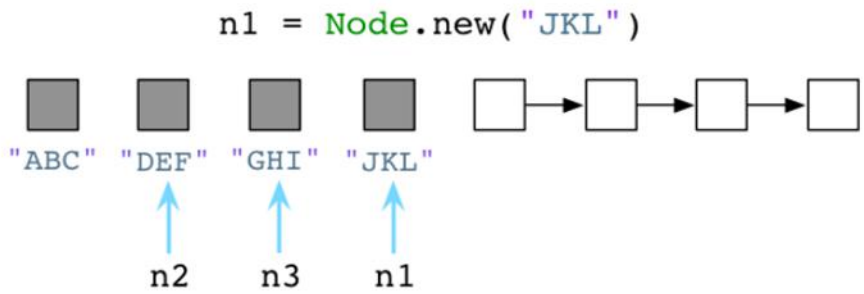


看起来够简单吧，在我们创建对象的时候，Python 会花些时间为我们找到并分配内存。

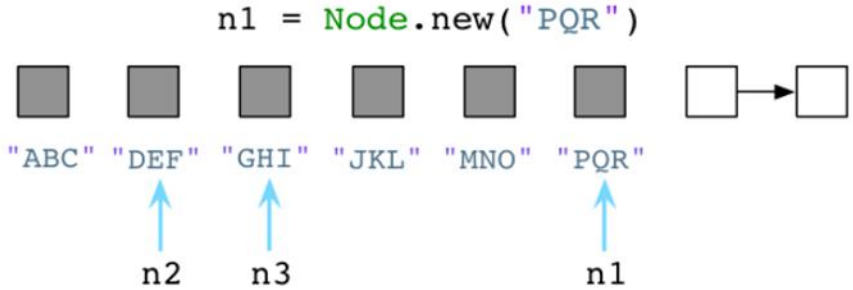
2.4Ruby 开发者住在凌乱的房间里

Ruby 把无用的对象留在内存里，直到下一次 GC 执行

回过来看 Ruby。随着我们创建越来越多的对象，Ruby 会持续寻可用列表里取预创建对象给我们。因此，可用列表会逐渐变短:



...然后更短:

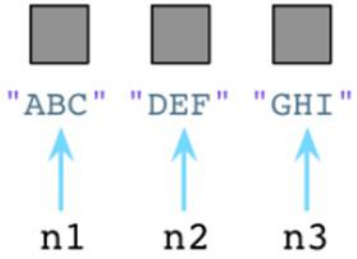


请注意我一直在为变量 n1 赋新值，Ruby 把旧值留在原处。"ABC","JKL"和"MNO"三个 Node 实例还滞留在内存中。Ruby 不会立即清除代码中不再使用的旧对象！Ruby 开发者们就像是住在一间凌乱的房间，地板上摞着衣服，要么洗碗池里都是脏盘子。作为一个 Ruby 程序员，无用的垃圾对象会一直环绕着你。

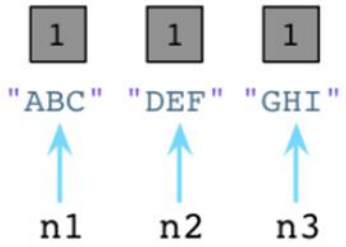
2.5Python 开发者住在卫生之家庭

用完的垃圾对象会立即被 Python 打扫干净

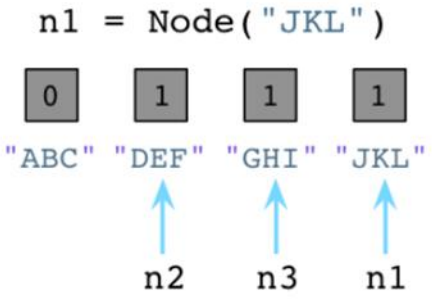
Python 与 Ruby 的垃圾回收机制颇为不同。让我们回到前面提到的三个 Python Node 对象：



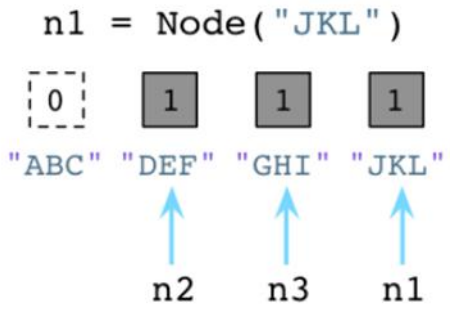
在内部，创建一个对象时，Python 总是在对象的 C 结构体里保存一个整数，称为 引用数。期初，Python 将这个值设置为 1：



值为 1 说明分别有个一个指针指向或是引用这三个对象。假如我们现在创建一个新的 Node 实例，JKL：

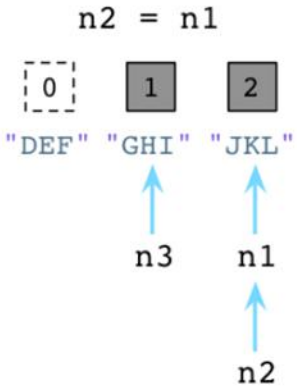


与之前一样，Python 设置 JKL 的引用数为 1。然而，请注意由于我们改变了 n1 指向了 JKL，不再指向 ABC，Python 就把 ABC 的引用数置为 0 了。 此刻，Python 垃圾回收器立刻挺身而出！每当对象的引用数减为 0，Python 立即将其释放，把内存还给操作系统：



Python 的这种垃圾回收算法被称为引用计数。

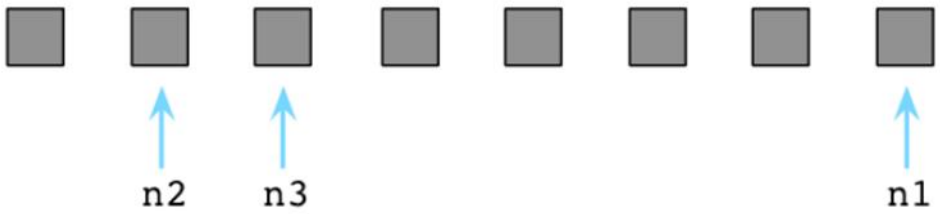
现在来看第二例子。加入我们让 n2 引用 n1：



上图中左边的 DEF 的引用数已经被 Python 减少了，垃圾回收器会立即回收 DEF 实例。同时 JKL 的引用数已经变为了 2 ，因为 n1 和 n2 都指向它。

2.6 标记-清除

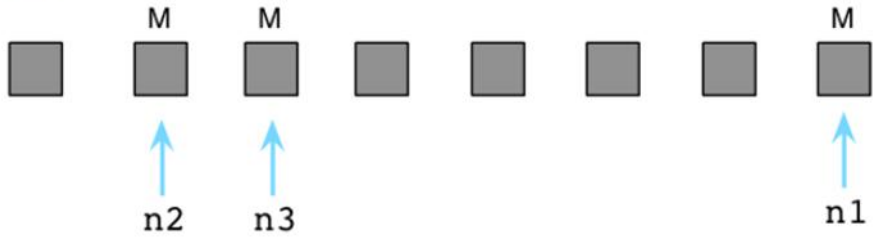
在 Ruby 程序运行了一阵子以后，可用列表最终被用光光了



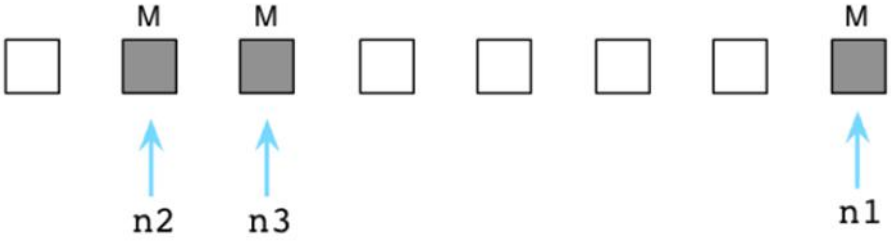
此刻所有 Ruby 预创建对象都被程序用过了(它们都变灰了)，可用列表里空空如也（没有白格子了）。

此刻 Ruby 祭出另一 McCarthy 发明的算法，名曰：标记-清除。首先 Ruby 把程序停下来，Ruby 用"地球停转垃圾回收大法"。之后 Ruby 轮询所有指针，变量和代码产生别的引用对象和其他

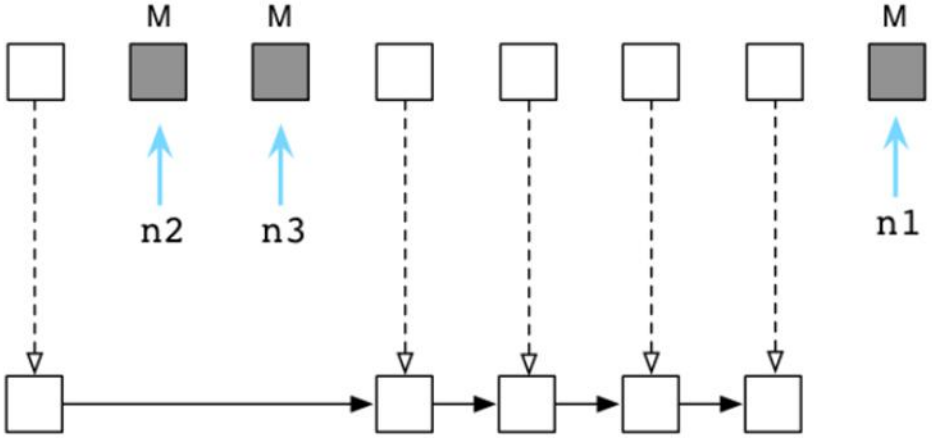
值。同时 Ruby 通过自身的虚拟机遍历内部指针。标记出这些指针引用的每个对象。我在图中使用 M 表示。



如果说被标记的对象是存活的，剩下的未被标记的对象只能是垃圾，这意味着我们的代码不再会使用它了。我会在下图中用白格子表示垃圾对象：



接下来 Ruby 清除这些无用的垃圾对象，把它们送回到可用列表中：



在内部这一切发生得迅雷不及掩耳，因为 Ruby 实际上不会吧对象从这拷贝到那。而是通过调整内部指针，将其指向一个新链表的方式，来将垃圾对象归位到可用列表中的。

2.7 标记-删除 vs. 引用计数

乍一看，Python 的 GC 算法貌似远胜于 Ruby 的：宁舍洁宇而居秽室乎？为什么 Ruby 宁愿定期强制程序停止运行，也不使用 Python 的算法呢？

然而，引用计数并不像第一眼看上去那样简单。有许多原因使得不许多语言不像 Python 这样使用引用计数 GC 算法：

首先，它不好实现。Python 不得不在每个对象内部留一些空间来处理引用数。这样付出了一小点儿空间上的代价。但更糟糕的是，每个简单的操作（像修改变量或引用）都会变成一个更复杂的操作，因为 Python 需要增加一个计数，减少另一个，还可能释放对象。

第二点，它相对较慢。虽然 Python 随着程序执行 GC 很稳健（一把脏碟子放在洗碗盆里就开始洗啦），但这并不一定更快。Python 不停地更新着众多引用数值。特别是当你不再使用一个大数据结构的时候，比如一个包含很多元素的列表，Python 可能必须一次性释放大量对象。减少引用数就成了一项复杂的递归过程了。

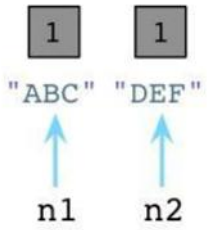
最后，它不是总奏效的。

3、Python 中的循环数据结构以及引用计数

我们知道在 Python 中，每个对象都保存了一个称为引用计数的整数值，来追踪到底有多少引用指向了这个对象。无论何时，如果我们程序中的一个变量或其他对象引用了目标对象，Python 将会增加这个计数值，而当程序停止使用这个对象，则 Python 会减少这个计数值。一旦计数值被减到零，Python 将会释放这个对象以及回收相关内存空间。

从六十年代开始，计算机科学界就面临了一个严重的理论问题，那就是针对引用计数这种算法来说，如果一个数据结构引用了它自身，即如果这个数据结构是一个循环数据结构，那么某些引用计数值是肯定无法变成零的。为了更好地理解这个问 0

题，让我们举个例子。下面的代码展示了一些上周我们所用到的节点类：

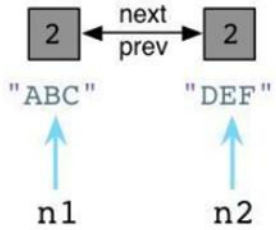



```
class Node:
  def __init__(self, val):
    self.value = val

n1 = Node("ABC")
n2 = Node("DEF")
```

我们有一个构造器(在 Python 中叫做 init)，在一个实例变量中存储一个单独的属性。在类定义之后我们创建两个节点，ABC 以及 DEF，在图中为左边的矩形框。两个节点的引用计数都被初始化为 1，因为各有两个引用指向各个节点(n1 和 n2)。

现在，让我们在节点中定义两个附加的属性，next 以及 prev：




```
n1.next = n2
n2.prev = n1
```

跟 Ruby 不同的是，Python 中你可以在代码运行的时候动态定义实例变量或对象属性。这看起来似乎有点像 Ruby 缺失了某些有趣的魔法。我们设置 n1.next 指向 n2，同时设置 n2.prev 指回 n1。现在，我们的两个节点使用循环引用的方式构成了一个双端链表。同时请注意到 ABC 以及 DEF 的引用计数值已经增加到了 2。这里有两个指针指向了每个节点：首先是 n1 以及 n2，其次就是 next 以及 prev。

现在，假定我们的程序不再使用这两个节点了，我们将 n1 和 n2 都设置为 null(Python 中是 None)。

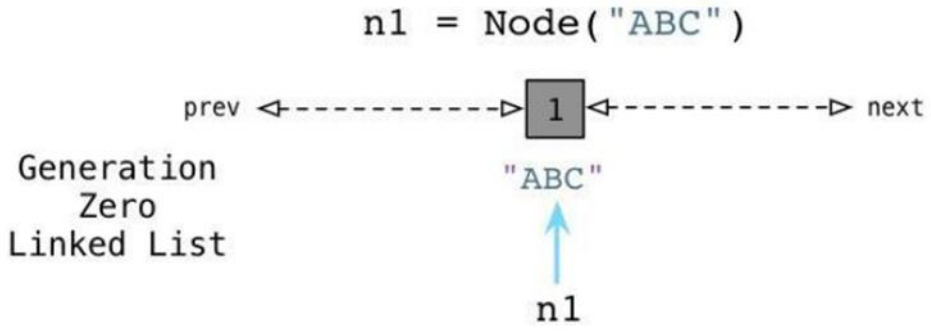
Python 会像往常一样将每个节点的引用计数减少到 1。

3.1 在 Python 中的零代(Generation Zero)

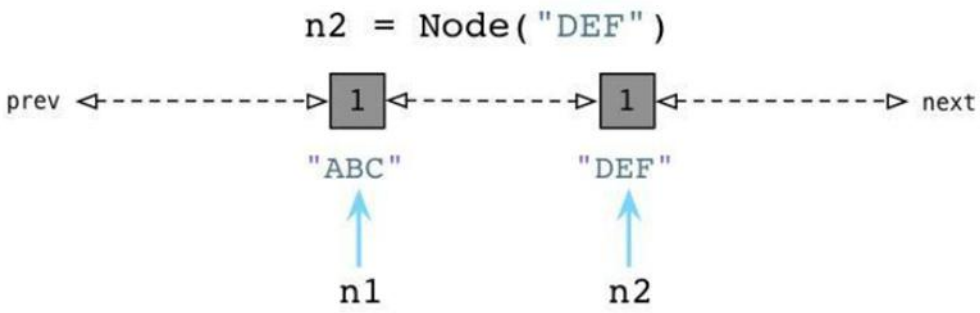
请注意在以上刚刚说到的例子中，我们以一个不是很常见的情况结尾：我们有一个“孤岛”或是一组未使用的、互相指向的对象，但是谁都没有外部引用。换句话说，我们的程序不再使用这些节点对象了，所以我们希望 Python 的垃圾回收机制能够足够智能去释放这些对象并回收它们占用的内存空间。但是这不可能，因为所有的引用计数都是 1 而不是 0。Python 的引用计数算法不能够处理互相指向自己的对象。

当然，上边举的是一个故意设计的例子，但是你的代码也许会在不经意间包含循环引用并且你并未意识到。事实上，当你的 Python 程序运行的时候它将会建立一定数量的“浮点数垃圾”，Python 的 GC 不能够处理未使用的对象因为应用计数值不会到零。

这就是为什么 Python 要引入 Generational GC 算法的原因！正如 Ruby 使用一个链表(free list)来持续追踪未使用的、自由的对象一样，Python 使用一种不同的链表来持续追踪活跃的对象。而不将其称之为“活跃列表”，Python 的内部 C 代码将其称为零代(Generation Zero)。每次当你创建一个对象或其他什么值的时候，Python 会将其加入零代链表：



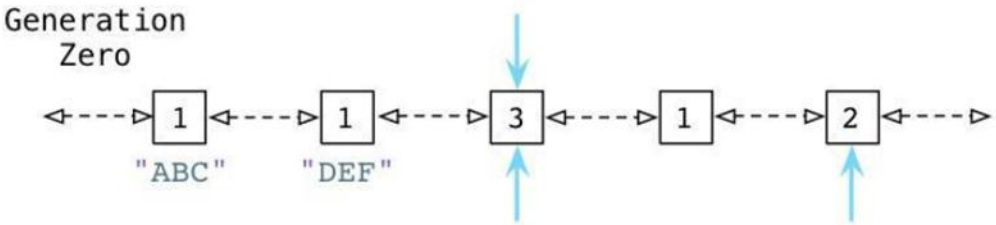
从上边可以看到当我们创建 ABC 节点的时候，Python 将其加入零代链表。请注意到这并不是一个真正的列表，并不能直接在你的代码中访问，事实上这个链表是一个完全内部的 Python 运行时。 相似的，当我们创建 DEF 节点的时候，Python 将其加入同样的链表：



现在零代包含了两个节点对象。(他还将包含 Python 创建的每个其他值，与一些 Python 自己使用的内部值。)

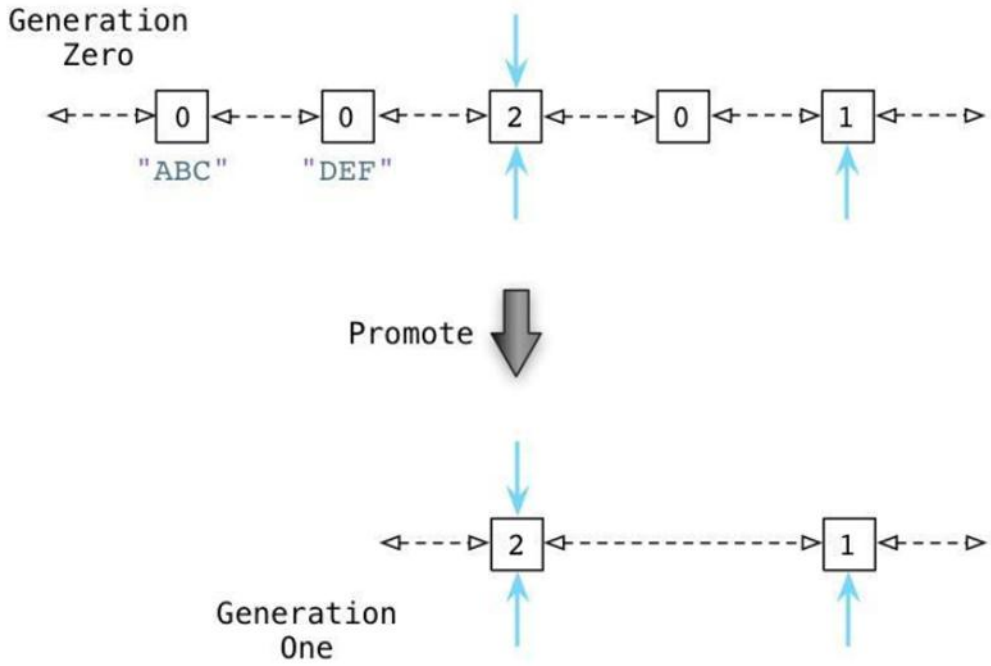
随后，Python 会循环遍历零代列表上的每个对象，检查列表中每个互相引用的对象，根据规则减掉其引用计数。在这个过程中，Python 会一个接一个的统计内部引用的数量以防过早地释放对象。

为了便于理解，来看一个例子：



从上面可以看到 ABC 和 DEF 节点包含的引用数为 1.有三个其他的对象同时存在于零代链表中，蓝色的箭头指示了有一些对象正在被零代链表之外的其他对象所引用。(接下来我们会看到，Python 中同时存在另外两个分别被称为一代和二代的链表)。这些对象有着更高的引用计数因为它们正在被其他指针所指向着。

接下来你会看到 Python 的 GC 是如何处理零代链表的。



通过识别内部引用，Python 能够减少许多零代链表对象的引用计数。在上图的第一行中你能够看见 ABC 和 DEF 的引用计数已经变为零了，这意味着收集器可以释放它们并回收内存空间了。剩下的活跃的对象则被移动到一个新的链表：一代链表。

从某种意义上说，Python 的 GC 算法类似于 Ruby 所用的标记回收算法。周期性地从一个对象到另一个对象追踪引用以确定对象是否还是活跃的，正在被程序所使用的，这正类似于 Ruby 的标记过程。

3.2Python 中的 GC 阈值

Python 什么时候会进行这个标记过程？随着你的程序运行，Python 解释器保持对新创建的对象，以及因为引用计数为零而被释放掉的对象的追踪。从理论上说，这两个值应该保持一致，因为程序新建的每个对象都应该最终被释放掉。

当然，事实并非如此。因为循环引用的原因，并且因为你的程序使用了一些比其他对象存在时间更长的对象，从而被分配对象的计数值与被释放对象的计数值之间的差异在逐渐增长。一旦这个差异累计超过某个阈值，则 Python 的收集机制就启动了，并且触发上边所说到的零代算法，释放“浮动的垃圾”，并且将剩下的对象移动到一代列表。

随着时间的推移，程序所使用的对象逐渐从零代列表移动到一代列表。而 Python 对于一代列表中对象的处理遵循同样的方法，一旦被分配计数值与被释放计数值累计到达一定阈值，Python 会将剩下的活跃对象移动到二代列表。

通过这种方法，你的代码所长期使用的对象，那些你的代码持续访问的活跃对象，会从零代链表转移到一代再转移到二代。通过不同的阈值设置，Python 可以在不同的时间间隔处理这些对象。Python 处理零代最为频繁，其次是一代然后才是二代。

4、Python 中的 GC 模块

4.1Python 垃圾回收机制

Python 中的垃圾回收是以引用计数为主，分代收集为辅

4.2 导致引用计数+1 的情况

- 对象被创建
- 对象被引用
- 对象被作为参数，传入到一个函数中
- 对象作为一个元素，存储在容器中

4.3 导致引用计数-1 的情况

- 对象的别名被显式销毁
- 对象的别名被赋予新的对象
- 一个对象离开它的作用域，例如 f 函数执行完毕时，func 函数中的局部变量（全局变量不会）
- 对象所在的容器被销毁，或从容器中删除对象

4.4 查看一个对象的引用计数

```
import sys
a = "hello world"
sys.getrefcount(a)
```

可以查看 a 对象的引用计数，但是比正常计数大 1，因为调用函数的时候传入 a，这会让 a 的引用计数+1

4.5 内存泄漏

申请了某些内存，但是忘记了释放，那么这就造成了内存的浪费，久而久之内存就不够用了

4.6 内存泄露演示

```
import gc

class ClassA():
    def __init__(self):
        print('object born,id:%s'%str(id(self)))

def f2():
    while True:
        c1 = ClassA()
        c2 = ClassA()
        c1.t = c2
        c2.t = c1
        del c1
        del c2

#python 默认是开启垃圾回收的，可以通过下面代码来将其关闭
gc.disable()

f2()
```

执行 f2()，进程占用的内存会不断增大。

创建了 c1，c2 后这两块内存的引用计数都是 1，执行 c1.t=c2 和 c2.t=c1 后，这两块内存的引用计数变成 2.

在 del c1 后，引用计数变为 1，由于不是为 0，所以 c1 对象不会被销毁;同理，c2 对象的引用数也是 1。

python 默认是开启垃圾回收功能的，但是由于以上程序已经将其关闭，因此导致垃圾回收器都不会回收它们，所以就会导致内存泄露。

4.7 手动调用 gc 回收垃圾

```
class ClassA():
    def __init__(self):
        print('id = %s'%str(id(self)))

def f2():
    while True:
        c1 = ClassA()
        c2 = ClassA()
        c1.t = c2
        c2.t = c1
        del c1
        del c2
        gc.collect()#手动调用垃圾回收功能，这样在自动垃圾回收被关闭的情况下，也会进行回收

#python 默认是开启垃圾回收的，可以通过下面代码来将其关闭
gc.disable()

f2()
```

有三种情况会触发垃圾回收：

当 gc 模块的计数器达到阀值的时候，自动回收垃圾

调用 gc.collect()，手动回收垃圾

程序退出的时候，python 解释器来回收垃圾

4.8 gc 模块的自动垃圾回收触发机制

在 Python 中，采用分代收集的方法。把对象分为三代，一开始，对象在创建的时候，放在一代中，如果在一次一代的垃圾检查中，该对象存活下来，就会被放到二代中，同理在一次二代的垃圾检查中，该对象存活下来，就会被放到三代中。

gc 模块里面会有一个长度为 3 的列表的计数器，可以通过 gc.get_count()获取。

例如(200, 8, 3)，其中 200 是指距离上一次一代垃圾检查，Python 分配内存的数目减去释放内存的数目，注意是内存分配，而不是引用计数的增加。 8 是指距离上一次二代垃圾检查，一代垃圾检查的次数，同理，3 是指距离上一次三代垃圾检查，二代垃圾检查的次数。

```
>>> class A():
...     pass
...
>>>
>>> import gc
>>> print(gc.get_count())
(200, 8, 3)
>>>
>>> a = A()
>>>
>>> print(gc.get_count())
(201, 8, 3)
>>>
>>>
>>> del a
>>>
>>>
>>> print(gc.get_count())
(200, 8, 3)
>>>
>>>
```

gc 模快有一个自动垃圾回收的阀值，即通过 gc.get_threshold 函数获取到的长度为 3 的元组，例如(700,10,10) 每一次计数器的增加，gc 模块就会检查增加后的计数是否达到阀值的数目，如果是，就会执行对应的代数的垃圾检查，然后重置计数器

```
>>> import gc
>>> print(gc.get_threshold())
(700, 10, 10)
```

当计数器从(699,3,0)增加到(700,3,0)，gc 模块就会执行 gc.collect(0),即检查一代对象的垃圾，并重置计数器为(0,4,0)

当计数器从(699,9,0)增加到(700,9,0)，gc 模块就会执行 gc.collect(1),即检查一、二代对象的垃圾，并重置计数器为(0,0,1)

当计数器从(699,9,9)增加到(700,9,9)，gc 模块就会执行 gc.collect(2),即检查一、二、三代对象的垃圾，并重置计数器为(0,0,0)

5、Python 内存优化

5.1 小整数与大整数对象池

Python 为了优化速度，使用了小整数对象池， 避免为整数频繁申请和销毁内存空间。 Python 对小整数的定义是 [-5, 256] 这些整数对象是提前建立好的，不会被垃圾回收。

```
# 小整数
>>> a = 1
>>> id(a)
10919424
>>> b = 1
>>> id(b)
10919424
>>> type(a)
<class 'int'>
>>>

# 大整数
# 每一个大整数，均创建一个新的对象。
>>> a = 1000
>>> id(a)
140489361555248
>>> b = 1000
>>> id(b)
140489337201840
>>>
```

5.2 python intern 机制

值同样的字符串对象仅仅会保存一份。是共用的，这也决定了字符串必须是不可变对象。就跟数值类型一样，同样的数值仅仅要保存一份即可了，不是必需用不同对象来区分。

intern 机制的优点是。须要值同样的字符串的时候（比方标识符）。直接从池里拿来用。避免频繁的创建和销毁，通过引用计数来维护这个字符串对象是否需要销毁，

须要小心的坑。并非全部的字符串都会採用 intern 机制。仅仅包括下划线、数字、字母的字符串才会被 intern。如果字符串中含有空格，不开启 intern 机制。

```
>>> a = 'abcd'
>>> b= 'abcd'  # 无空格开启 intern 机制
>>> id(a)
140489337335736
>>> id(b)
140489337335736
>>>
>>> a = 'hello world'    # 字符串含有空格，不开启 intsern 机制
>>> b = 'hello world'
>>> id(a)
140489337347696
>>> id(b)
140489337347760
```

不可变类型。是无法修改这个对象的值，每一次改变实际上是创建了一个新的对象。

```
>>> a = 1
>>> id(a)
10919424
>>> a += 1
>>> id(a)
10919456
>>>
>>> b='abcd'
>>> id(b)
140489337335736
>>> b='abcd'+'efg'
>>> id(b)
140489337360488
>>>
```