第14章内存管理

无论是C语言的手动管理，还是Jav日语言的垃圾回收机制，都彰显了内存管理是决定语 言性能的重要因素。Python作为一门动态语言，在程序运行的过程中，会不断地创建和销毁大 量的对象，所以内存管理在很大程度上决定了 Pythg的执行效率。接下来，带领大家看一下 Python中的内存管理方式。

内存管理慨述

14JJ 夠ft遂靈

内存管理是指软件运行时，对计算机内存资源的分配和使用的技术，其最主要的目的是高 效、快速地分配，并且在适当的时候释放和回收内存资源。

当程序开始运行时，它会在内存中申请一份合适大小的存储空间，由于硬件设备的内存是 有限的歹所以能分配给程序的内存资源更是寥寥无几。

程序运行的过程中，需要不断地创建新的对象，或者销毁不再使用的对象，这些对象都会 在内存中占据一定的存储空间。因此，程序要在有限的空间内部，不停地开辟新的空间，或者 回收垃圾以腾出不再使用的空间。

如果给对象分配的空间比实际使用的空间过大，就会造成内存资源的浪费；如果已经不再 被使用的对象滞留在内存的时间过长，就会引发内存泄漏的问题；如果回收了正在被使用的对 象的内存，程序就很可能会产生错误而终止运行。因此，内存管理显得尤为重要。在Python中, 一切类型的实例皆为对象，内存管理的目标就是这些对象。

在Python这门脚本语言中，程序员已经不用再关心如何管理内存了，这些工作都会交由 Pythori解释器负责。但是，对于实际编程来说，理解内存管理的能力与局限性将有助于程序员 更好地开发程序。

为了能更好地管理内存，Python提出了 3种管理内存的方式，具体如下:

274 w.. Pjthon襄鐵鑑圜 映罂攣Python

1. 引用计数机制

Pythoii内部记录着所有对象的引用数量，该数量使用一个内部跟踪变量来存储，这个跟踪 变量就是引用计数器。一旦对象被创建，它的引用计数器数值从。变成"对象的引用计数， 随着对象引用数量的增多而增多。对象的引用数量每增加一个，引用计数器的值就增加lo反之, 对象的引用数量每减少一个，引用计数器的值就减少L直到对象的引用计数器数值变成。时， 就会被当作垃圾进行回收。

1. 垃圾回收机制

当内存中不再使用某个对象时，垃圾收集器就会把它们清理掉，它会去检查那些引用计数 为。的对象，然后清理其所在的内存空间。

1. 内存池机制

Python的垃圾回收机制会清理不再使用的内存，但是它并不是将不用的内存返回给操作系 统，而是放到内存池中，内存池机制用于管理小块内存的申请和释放。

上面所讲的这些机制中，引用计数机制其实也是一种垃圾回收机制。但是，引用计数机制 潜伏着两个问题：第一，是降低垃圾回收的效率；第二，是解决不了循环引用的问题。为了弥 补这些不足，垃圾回收机制采用以引用计数技术为主，标记-清除和分代收集技术为辅的战略, 对垃圾收集的功能进行了完善，在后面的小节中，我们都会进行详细的介绍。

胸。2驯陶讲教现制

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。从广义上来讲， 引用计数机制也是一种垃圾回收机制，而且是一种最直观、最简单的垃圾收集技术。

什么是引用呢？在前面的章节中已经有所介绍，下面先举个例子来说明：

a = 1

上述示例中，整数1是一个对象，a是一个引用， 0xl0027b40

通过使用将&的值赋值为1,这也意味着引用a指 mn

a —► 1

向了对象"此时，弓I用&和对象1在内存空间存储的状 J

态如图14』所示。 图1心 引用a指向对象1的内存状态

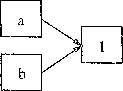
创建多个等于1的引用，实际上是让这些引用指向同一个对象。为了检验这些引用是否指 向了同一个对象，可以通过is语句进行判断，具体如下：

»>b=l

»> a is b #判断a和b的id是否相同

True

上述示例中，对象1又赋值给变量b。通过比较两个的内存地址可以发现，引用a和引用 b指向的是同一个对象1,说明赋值语句只是创建了新的引用，而不是创建了对象本身。此时， 引用L弓【用b和对象1在内存空间存储的状态如图14・2所示。

在Pythori中，整数(-5〜256)和字符串(只包含一个字 符，或者只包含字母、数字或下画线)都会被缓存起来，以 便能够重复使用。

0xl00275b40

除了这些整数和字符串以外，其他对象都可以使用赋值图心务和b指向1的内存状态 语句创建多个实例。这些实例虽然存储的内容是一样的，但 是它们的内存地址是不一样的，也就是说，它们是不同的对象。示例如下：

>>>a= 5hellomy name is Jane! \*

>>>b=,hellozmy name is Jane! !

>>>a is b

False

为了让每个对象知道指向它的引用数量，Python引入了引用计数器的概念。Python中的每 个对象都拥有一个引用计数器，表示有多少个引用指向该对象。引用计数器使用整数表示，可 以通过sys包的getrefcountO函数查看该计数器的数值。需要注意的是，当使用某个引用作为 参数传递给getrefcountO函数时，该参数会创建一个临时的引用，所以调用函数得到的结果会 比期望的值多1。示例如下：

>>>from sys import getrefcount

>>>list\_\_one= [ 1 z 2 z 3z 4 z 5] >>>getrefcount(list\_one)

2

>»list\_two=list\_one

>>>getrefcount(list\_two)

3

上述示例中，弓Iffi list„one指向了刚创建的列表，说明列表当前有1个引用；引用listjwo 赋值为引用list\_one9说明引用list\_tw。也指向了列表，列表当前有2个引用。从示例结果可以 看出9 list one和list two的引用计数器返回的是2和3,却不是期望的1和2。 ｛关5主意： ’

*勘瞄牌"*

由于Python会缓存一些数字和字符串,所以使用getrefcount函数获取到这些对象的引用 计数器数值会很大，是无法准确地获取引用计数器的数值的。

14.2.2蹭掘爾療酶引魇

每次创建或者复制一个指向对象的引用时，会让该对象的引用计数设置为1。如果其他对 象要使用该对象，就会增加该对象的引用数量，从而让该对象的引用计数器数值增加，具体增 加该对象的引用数量的操作包括如下场景：

1. 当同一个对象的引用或者对象又被赋值给其他变量时，会让该对象的引用计数器数 值增加1,示例如下：

»>a=l

»>b=a

1. 当对象作为参数传递给函数时，会让该对象的引用计数器数值增加1,示例如下：

>>>func(a)

1. 当某个对象加入容器中成为其内部的元素时，会让该对象的引用计数器数值增加， 示例如下:

b=3

a=[l,2,b]

14H2n3 SWBBSra

与增加引用操作相反，只要某个对象的引用被销毁时，就让对象的引用计数器数值减少。 下面列出的是减少引用的操作:

1. 当对象的引用离开其作用范围时，会让该对象的引用计数器数值减少1,这种情况经 常出现在函数运行结束时，其内部的所有局部变量都被自动销毁，示例如下：

>>>def func ():

"。如 a=0

.。. print(a)

>>>func ()

1. 使用del语句显式地销毁对象的引用，示例如下：

>>>del a

1. 当把另外一个对象赋值给变量后，原对象的引用计数器数值自动减少1,示例如下：

>>>word='a 1

>>>word\_other=word

»>word=' c 1

在上述示例中，对象W被创建并赋值给word,此时对象W的引用计数器值为1。增加一 个引用word\_other指向矿 使得该对象的引用计数器值变成2。重新给word赋值为矿 使得 指向对象农的引用减少了一个，所以该对象的引用计数器值变成lo

1. 当从容器对象中删除某个对象时，会让该对象的引用计数器数值减少1,示例如下：

>>>demo\_\_list= [ 17 2 f 3 ]

>>>demo\_list.remove(1)

1. 当容器对象本身被销毁，或者容器对象离开了其作用范围时，会让其内部的所有对 象的引用计数器数值减少1,示例如下：

»>demo\_list= [17 2Z 3]

>>>del demo\_list

只要某个对象的引用计数器数值为0,就意味着已经没有任何其他对象使用这个对象了， 此时可以将其占用的内存空间进行回收，示例如下：

>» class Test: # 创建一个 Test 类

def del (self) : #删除对象来释放类所占用资源时自动调用

.。。 print(，--end--5)

>>>from sys import getrefcount

|  |  |
| --- | --- |
| >>> test=Test() | #创建Test类对象，并赋值给引用test,引用计数变成1 |
| >>>getrefcount(test)  2  »> del test  --end-- | #获取引用计数的值  #使用del语句删除引用，使得引用计数变成0 #对象销毁，自动调用了 \_del\_()方法 |

在上述示例中,定义了一个Test类，该类中只有一个\_del\_()方法,此方法会在该类的实 例销毁时自动调用。创建一个指向TM类对象的引用test,此时TM类对象的引用计数器值为

使用del语句显式地销毁了引用於st,说明当前没有任何引用指向Test类对象，此时Test 类对象的引用计数器值由1变为0。

从示例的结果可以看岀，Test类对象已经被销毁了，垃圾收集器会把它当作垃圾进行回收。

引用计数机制必须在每次分配和释放内存时，加入管理引用计数的动作，这在一定程度上 会降低程序的效率。不过，与其他主流的垃圾收集技术相比，引用计数最大的优点就是“实时 性”，一旦没有任何引用指向一块内存，该内存就会被当作垃圾进行回收，而其他的垃圾收集 技术只能在某种特殊条件下，比如内存分配失败，才能进行无效内存的回收。

容器对象弓撫的M环驯康

在Python中9能够引用其他对象的对象被称为容器，比如列表、字典、兀组等°容器对 象中可以包含多个元素，有时包含的元素的本身，有时包含的是指向这些元素的引用，例如， 列表[l.b]中包含了一个整数1和一个引用bo

当某个对象或者引用加入到容器中，该对象或者该引用所指向的对象的引用计数器的值会 增加，具体示例如下。

>>> from sys import getrefcount

>>> demo\_tuple = (1〃 2) # 创建一个元组

>>> getrefcount (demo\_tuple) #获取元组的引用计数器的值

2

>>> demo\_list = [demo\_tuple7 1f demo\_tuplef 3]

>>> getrefcount(demo\_tuple)

4

上述示例中，创建元组对象并赋值给变量demo\_tuple 9表示引用demo\_tuple指向了该元组 对象，此时该元组的引用计数器值由0变为"

创建一个容器对象[demo\_tuple49demo\_tuple?3]9该容器对象中的两个元素为引用demo\_tuple 指向的元组对象。换言之，元组对象的弓I用数量又增加了两个，此时它的弓【用计数器值由1变为3。

14.3.2醐録窘廁II源蟀引魇

循环引用是指多个对象之间互相引用，比如对象list\_one引用了对象list\_two9对象list\_ two又引用了对象list oneo循环引用的一组对象引用计数都不为0,然而并没有任何外部对象引用它们，这就意味着外部对象不会再使用这组对象，应该及时回收这组对象所占用的内存空 间。不过由于循环引用的存在，这组对象的任意一个的引用计数都不为0,所以这组对象占用 的内存空间永远得不到释放。

对象之间循环引用的示例如下:

>>>list\_\_a =[]

>>>list\_b =[]

>>>list\_a。append(1ist\_b)

»>list\_b。append (list\_\_a)

>>>del list\_a

>>>del 1i s t\_b

•上述示例中，弓list\_a指向了一个列表，弓Illi list\_b指向了另一个空列表。

指向的列表添加了引用list\_b,说明list\_b引用的对象成了该列表的一个元素；list\_b 指向的列表添加了引用list\_\_a,说明list\_a引用的对象成了该列表的一个元素。

此时，两个列表的元素对象分别为这两个列表本身，它们之间互相引用了对方，如图14-3

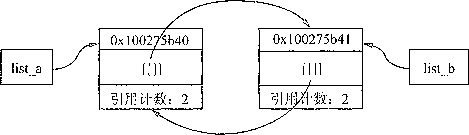
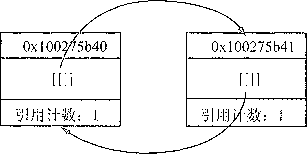


图14-3对象list\_a和对象list\_b的内存状态

使用如语句删除引用list\_a和引用list\_b以后， 由于没有任何引用指向它们之前引用的列表对象，导 致程序无法再访问到这组对象，此时可以把这组对象 当作垃圾进行回收。不过两个列表对象之间存在循环 引用，使得它们的引用计数器值不为0,所以它们不

会被当作垃圾回收，如图14-4所示。 图14-4对象和对象互相引用

循环引用的问题并不是只有两个对象能引起的，即使只有一个对或这个对象岳己引用了 自己，也会造成循环引用的问题，示例如下：

>>>demo\_list = []

>>>demo\_list。append(demo\_list)

上述示例中，弓Idemojist指向了一个列表，然后在该列表中添加了引用demojist,表 示demojist引用的对象成了该列表的一个元素，这个元素就是列表本身。程序结束后，由于 列表的元素与列表本身相互引用对方，导致该列表不会被当作垃圾回收。

对于垃圾回收机制来说，循环引用的问题是致命的，它与手动内存管理产生的内存泄漏没 有任何区别。为了解决循环引用的问题，Python提供了弱引用技术和标记清除技术。其中，弱 引用技术的局限性比较大，它只能解决自定义对象间的循环引用问题，而标记清除技术专门用 于解决容器对象可能产生的循环引用问题，后续的章节会有详细介绍。

1<4飜引阚解礫自定义对象胸循环朝關

循环引用的一组对象，它们既不能被其他对象使用，又不能当作垃圾回收，它们会一直驻 留在内存中，直至程序结束。程序如果持续产生这种不可使用的对象，就会发生内存泄漏。因此， Pythoii引入了弱引用技术，解决对象之间产生的循环引用。接下来，本节将针对弱引用的内容 进行详细的介绍。

靄引li襪逵 「 「

在计算机程序设计中，弱引用是指不能确保其指向的对象不会被垃圾收集器回收的引 用，也就是说，如果一个对象只被弱引用所引用，则该对象可能在任何时刻被回收。强引用与 弱引用正好相反，一个对象只要被强引用所引用，则该对象肯定不会被回收。Python中使用 wealcref模块创建对象的弱引用，一旦某个对象的引用计数器值为0,或者只存在对象的弱引用 时，就会回收该对象占用的内存空间。

创建对象的弱引用可以使用如下两种方式：

方式一:weakref.ref函数创建弱引用

可以通过调用weakref模块的ref(obj[?callback])函数来创建一个弱引用，其语法格式如下：

weakref.ref(obj ect[z callback])

在上述语法中，冀f函数有两个参数。其中，object为弱引用的对象，callback是一个可选 的函数，当没有任何引用指向的对象销毁时调用，并且把object作为唯一的参数进行传递。

通过ref函数创建弱引用的示例如下:

>>>import weakref

>>>class Test:

。.. pass

>>>test=Test()

>>>weak=weakref . ref (test) # 创建个弱引用 t 指向 Test 对象

>>>del test

上述示例中，自定义了一个T©就类。首先，创建一个T©就类对象并且赋值给tesL此时 Test类对象的引用为强引用testo

然后，通过weal<re£ref创建了弱引用weak,其指向的是test引用的对象。此时Test类对 象的引用有两个，其中一个为弱引用wsk,另一个为强引用翊t。

当如 语句删除强引用曲t后，由于Test类对象的引用只有一个弱引用wsk,所以该对象 会被当作垃圾回收。

如何判断弱引用指向的对象销毁了呢？这里，我们可以通过“弱引用名称()”的形式获取 弱引用指向的对象，如果该对象还存在于内存中，就返回该对象，否则返回None。

在上述示例的末尾，验证弱引用指向的对象是否被销毁，示例如下：

>>>print(weak())

None

方式二:weakref.proxy函数创建弱引用

前面所讲的ref函数有个不便之处，就是每次在获取弱引用所指向的对象时，都要进行初 始化操作，即在弱引用的后面加上括号，这就使得弱引用和强引用在代码使用层面上产生了差 异。因此Python提供了另一个函数proxy(object[? callback]),用于创建一个弱引用，其语法格 式如下：

weakref.proxy(obj ect[/ callback])

上述函数跟mf函数的参数都是一样的，这里就不再赘述了。

通过proxy函数创建弱引用的示例如下：

>>>class Test:

。。 pass

>>>import weakref

>>>test=Test()

>>> proxy=weakref.proxy(test) #

创建一个弱引用，指向Test对象 获取弱引用指向的对象

>» print (proxy) #

< main .Test object at 0xl006e69e8> >>> del test #

删除强引用

再次获取弱引用指向的对象

>>> print(proxy) #

Traceback (most recent call last):

File n<stdin>° line 1f in <module>

ReferenceError: weakly-referenced object no longer exists

当弱引用proxy引用的对象销毁以后，再访问proxy,则会抛出ReferenceError异常。为了 解决上述问题，需要判定proxy指向的对象是否有效，可以使用try异常处理语句尝试捕获， 示例如下：

try:

proxy. name return True

except ReferenceError : return False

既然弱引用指向的对象已经销毁了，这些弱引用就没用了，它们是什么时候清理的呢？在 各种Python类型的xx\_dealloc\_方法中，会清理这些弱引用。

值得一提的是，wealaef模块只能创建自定义对象的弱引用，诸如列表、元组、字典、 数字、字符串或None是不能创建的。一旦使用wskref创建了这些对象的弱引用，就会引发 TypeErroi•异常，具体示例如下。

»> import weakref # 导入 weakref 模块

»> demo\_list = [ ] #创建一个强引用demo\_list,指向刚创建的列表对象

»> proxy = weakref .proxy(demo\_list) #创建一个弱引用proxy,指向强引用指向的列表对象 Traceback (most recent call last):

File "<stdin>°, line *lf* in <module>

TypeError: cannot create weak reference to Jlist? object

(泠注意：

(1 )如果将弱引用不断传递，则容易遇到ReferenceError异常。

(2)如果弱引用指向了一个对象，那么该对象的引用计数器值不变。

14.4.2

当两个对象之间出现循环引用时，可以让一方使用强引用指向，另外一方使用弱引用指向。 这样，当没有任何其他引用指向这两个对象时，由于弱引用没有持有对象的能力，所以它指向 的对象的内存会优先得到释放。

为了大家更好地理解，下面以proxy函数为例，介绍如何使用弱引用处理两个对象间产生 的循环引用，示例如下：

>>> import weakref

>>> class Test(object):

... def init (self):

.。. print (° 创建,内存地址为 %d"%id (self))

... def del (self):

…. print ("销毁,内存地址为%dn%id (self))

>>> def func():

strong\_a = Test()

strong\_b = *Test()*

strong\_a weak\_b = weakref.proxy(strong\_b) strong\_b e weak\_a = weakref 8proxy(strong\_a) func()

»>

创建，内存地址为4302804920 创建，内存地址为4302860752 销毁，内存地址为4302804920 销毁，内存地址为4302860752

上述示例中，在T顷类中声明了 \_init\_()和\_d<\_()方法，分别用于在创建对象和销毁 对象时输出打印语句。

在feme函数中，强引用strong\_a指向了 Test类对象，强引用strong\_b指向了另外一个Test 类对象。

创建弱引用为strong\_a.weak\_b,创建另一个弱引用为strong\_b.weak\_a0此时,strong\_a*弓*I 用对象的属性weak\_b为引用strong\_b指向的对象，而strong\_b引用对象的属性weak\_a为引用 strong/指向的对象。即使两个对象之间互相引用了对方，但是它们之间使用的是弱引用，不 会出现循环引用的问题。

当Pythoii中的对象占据越来越大的内存时，不必过于担心Pythoii的"体形”，原因在于 它会在适当的时机“减肥”,即启动垃圾回收机制，把那些没用的对象清除。在许多语言中都 有垃圾回收机制，比如Java和Ruby,尽管它们最终达到的目的都是“塑造苗条的体形”，但 是不同语言的“减肥方案”有着很大的差异。接下来，带领大家认识Python的垃圾回收机制。

14n5J

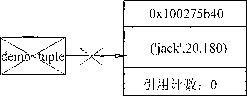
前面所介绍的引用计数是最为直观、简单的垃圾回收技术，其基本的原理就是，当某个对 象的引用计数器值为。时，说明没有任何引用指向该对象，此时，这个对象就会被当作垃圾进 行回收。

例如，为新建的对象分配引用，使得该对象的引用计数器值变为1;删除指向刚刚分配给 该对象的引用，使得该对象的引用计数器值变为0,这个对象就是待回收的垃圾，示例如下：

>>>demo\_tuple=: (1 Jack^20^ 180)

>>>del demo\_tuple

上述示例中，引用demojuple指向了新建的元组对象。由于使用del语句删除引用a,所 以此时没有任何引用指向刚才建立的元组，如图14.5所示。

当垃圾回收机制启动以后，垃圾收集器会扫描到这个 引用计数为0的对象，并且将该对象所占用的内存全部回收。 垃圾收集器是一块独立的代码，用于寻找引用计数器值为0 的对象。

不过，当容器对象产生循环引用时，引用计数技术就 无能为力了。为了解决上述问题，Pythg又在gc模块中补 充了标记清除技术和分代垃圾回收技术。其中，“标记■清除”技术负责解决容器对象产生的 循环引用的问题，“分代回收”技术负责提高垃圾回收的效率。

图14・5没有引用指向建立的元组

14.5.2糠词清除援术

标记，清除技术是一种基于追踪回收技术实现的垃圾回收算法，它依赖于对所有存活的对 象进行一次全局遍历，确定哪些对象可以回收。整个遍历的过程分为两个阶段：

第一阶段：标记阶段，gc模块会把所有存活的对象打上标记。

第二阶段：清除阶段，把那些没有标记的垃圾对象进行回收。

那么，gc模块是如何判断哪些是活动对象，哪些是非活动对象呢？

对象间通过引用连在一起的，从而构成一个有向图。对象则构成这个有向图的结点，引用 关系构成这个有向图的边。先要寻找root object (全局变量、调用栈、寄存器，不能被回收)， 从这个根对象出发须沿着有向边遍历对象，可达的对象标记为活动对象，不可达的对象就是要 被清除的非活动对象，如图14弟所示。

在图14-6中，我们把黑点视为全局变量(root object)，从小 黑点开始，对象1是可达的，它会被标记；对象2和3是间接可达的， 同样会被标记；而对象4和5是不可达的。所以对象1、2、3是活 动对象，4和5是非活动对象，会被视为垃圾回收。

实际上，标记■清除不会改动真实的引用计数，而是将集合中 对象的引用计数复制一份副本，改动该对象引用的副本。这样，对 副本做任何的改动，都不再影响到对象的生命周期的维护。这个引

ff

图14-6有向图示例

用副本的唯一作用就是寻找root object集合，当成功找到根对象以后，会把现在的内存链表一 分为二，一条链表中维护root object集合，成为root链表，而另外一条链表中维护剩下的对象, 成为unreachable链表。

为什么要分成两个链表呢？是基于这样一种考虑：现在的unreachable链表很有可能存在 被roof链表中的对象直接或间接引用的对象须这些对象是不能被回收的，一旦在标记的过程中 发现这样的对象，就要把它从unreachable链表中移到root链表中。

完成标记以后,unreachable链表中剩下的所有对象就是名副其实的垃圾对象了，所以垃圾 回收只需限制在unreachable链表中即可。把图14-6的对象进行划分,如图14-7所示。

存活的对象：I j FT］ 匸］E1

root 链表：［JJ4ZHZI

r 1 I—~I 口 : 被标记对象

I 4 H 5 I M：标记

unreachable 链表:

图 14-7 root 链表和 unreachable 链表

与引用计数相比，标记-清除算法可以很自然地处理引用环的问题狩在创建和销毁对象时 少了操作引用计数值的开销。但是在垃圾回收器运行过程中，应用程序必须暂时停止，而分代 垃圾回收正好减少了它的停顿时间，下面会做具体的介绍。

1感霑戻xtsww

从“标记-清除”这样的垃圾回收机制来看，它所带来的额外操作实际上与系统中总的内 存块数量是相关的。当需要回收的内存块越多，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，则垃圾检测将比垃圾回收带来更少的额外操作。为了提高垃圾 收集的效率，gc模块利用“空间换时间”的策略（分代回收），减少垃圾回收中扫描的频率。

那么，分代回收是如何做到的呢？基本原理如下：

Python将系统中的所有内存块根据其存活的时间划分为不同的集合，每个集合就成为一个 代。内存分为了3个代,分别为第0代（年轻代）、第1代（中年代）、第2代（老年代）， 它们对应着三个链表。

垃圾收集的频率随着代的存活时间的增大而减小。新创建的对象都会分配在年轻代，年 轻代链表的总数达到上限时，垃圾收集机制就会被触发，把那些可以被回收的对象回收掉， 而那些不会回收的对象就会移到中年代。假设年轻代链表的总数为5,当达到上限后如图14.8 所示。

以此类推，老年代中的对象是存活时间最久的对象。那么如何来衡量这个存活时间呢？通 常是利用几次垃圾收集动作进行抉择，如果一个对象经过的垃圾收集次数越多，可以得出该对 象存活的时间越长。活的越长的对象，就应该减少对它的垃圾收集频率。

接下来，举个例子进行说明。

年轻代(第o代)：

中年代(第]代)M |

区：回收的对象

老年代(第2代)： □:不回收的对象

图14-8从年轻(第零)代移到中年(第一)代

有个内存块N,该块在经过了3次垃圾收集的清洗之后依然存活，我们就将内存块N划分 到集合A中，新分配的内存都划分到集合B中。

当垃圾收集开始工作时•大多数情况下只会对集合B进行垃圾回收，而对集合A进行垃 圾回收要隔很长一段时间后才进行，这使得垃圾收集机制需要处理的内存变少，效率自然就提 高了。

在这个过程中，集合B中的某些内存块由于存活时间长而被转移到集合A中。当然，集 合A中实际上也存在一些垃圾,不过这些垃圾的回收会因为这种分代机制被延迟。

14d5d4闢◎模煥疆述

Python为垃圾收集器提供了设置的接口 gc模块，该模块用于禁用收集器，调整收集频率， 以及设置调试选项。另外，这个模块还可以对被垃圾收集器发现但不能释放的对象进行访问。

由于垃圾收集器补充了已经使用的引用计数技术，只要能确保程序中没有创建循环引用的 对象，就可以禁用垃圾收集器。因此，该模块提供了用于设置垃圾收集器的一些函数，具体如下：

* gc.enable()：启动自动垃圾回收。

® gcJisable():禁用自动垃圾回收。

* gcJsenabled()：如果启动自动垃圾回收，则返回True。

® gcxollect(generation=2):显式地进行垃圾回收。generation参数取值为从0到2的整数， 用于指定要收集垃圾的代数。其中，0代表只检查第一代对象，1代表检查第一代和第 二代对象，2代表检查第一、二、三代对象。如果没有传入参数，相当于传入的是默认 值2「程序会执行完整地收集垃圾，返回所有不可达对象的数目。

® gc.set\_debug(flags)：设置 gc 的 debug 0 志,一般设置*为* gc.DEBUGJLEAK。

* gc.set\_threshold(thresholdO[, threshold 1[, threshold?]])：设置垃圾收集的阈值(收集频率)。

® gc.get\_threshold():将当前收集阈值作为元组(thresholdO，threshold 1, threshold?) 返回。

® gc.get\_count():将当前的收集计数作为元组(countO, count 19 count2 )返回。

14^5.5 k

Pythoii的垃圾回收机制，既能够让程序员手动启动，又能自动启动，主要依赖于上述所讲 的三种垃圾回收的技术。当程序结束时，会清理所有不再使用的资源，此外还可以通过gc模 块启动垃圾回收，具体分为如下两种情况：

1孥薊回收垃圾

程序员可以导入gc模块，主动调用collectQ函数手动实现垃圾回收。例如：

»>import gc # 导入模块

>>>gc . collect () # 手动

5 #返回不可达对象的数目

2当容器对象的数目达到某个阈值卧 会国动调願鲍丽说0回收垃圾

在运行Python程序时，会记录其中分配对象(object allocation)和取消分配对象(object de疝ocation)的次数。一旦两者的差值高于某个阈值的时候，垃圾回收才会启动。

可以使用gc模块的get\_threshold()函数查看该阈值：

>>>import gc

>>>gc a get\_threshold () # 查看阈值

(700, 10, 10)

其中，700是垃圾回收启动的阈值，后面的两个数值10是与分代回收相关的阈值，后面再 做介绍。这个回收启动的阈值并非一成不变的，可通过gc模块的set\_thresliold()函数重新进行 设置。

示例如下:

>>>gc。set\_threshold(750)

>»gc. get\_\_threshold () # 重新设置阈值

(750, 10, 10)

只有引入gc模块，并且isenabledO函数返回True时才会启动自动垃圾回收，这个机制的 主要作用是发现并处理不可达的垃圾对象。自动垃圾回收共包括两个过程：垃圾检查和回收 垃圾。

我们知道，采用分代收集的方法把对象分为三代。起初，在创建对象时会放到一代中，如 果在一次一代的垃圾检查中，这个对象存活下来，就会被放到二代中，依此类推，如果通过二 代的垃圾检查后继续存活下来，就会被放到三代中。

在gc模块里面有个长度为3的列表计数器(get\_count函数)，例如(488,3,0)，其中488 是指距离上一次一代垃圾检查后的分配内存减去释放内存的数目。

示例如下:

>>>class Test:

。. pass

>>>gc。get\_count()

(59, 0, 1)

>>>test=Test ()

>>>gc»get\_count()

(60, 0, 1)

>>>del test

>>>gc.get\_count()

(59, 0, 1)

在gc模块中还有自动垃圾回收的阈值(getjhreshold函数)，同样是长度为3的元组，例 如(500,10,10)。每次计数器的值增加膏尝模块都会检查增加后的计数是否达到阈值的数目，如 果达到了就会执行相应代数的垃圾检查，然后重置计数器。

例如,阈值为(500,10,10)：

•当计数器从(499,3,0)增加到(500,3,0)，gc模块就会执行gcxollect(0)3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(499,9,0)增加到(500,9,0), gc模块就会执行gcxollect(l)?即检查一、二代对 象的垃圾，并重置计数器为(001)

•当计数器从(499,9,9)增加到(500,9,9) ? gc模块就会执行gcxollect(2)5即检查一、二、三 代对象的垃圾，并重置计数器为(000)

通常情况下，Python对象申请的内存都是小块内存(比如整数、字符串)，这些小块内存 在申请以后很快又会被释放。因此，程序在运行期间会一直申请和释放内存，严重影响了程序 的执行效率。为了提升程序的性能,Python引入了对象池和intern机制,下面针对两个机制为 大家进行详细介绍。

我们先来看一段示例程序：

»> a = 100

»> b = 100

>>> a is b

True

上述示例中，引用&和引用b都指向了整数对象100,之后使用is语句判断这两个引用是 否指向同一个对象。通过输出的结果可以看出，弓I用a和引用b指向了同一个对象。

整数在程序中使用非常广泛，为了避免整数频繁地申请和销毁内存空间，Python对小整数 使用了对象池技术，所谓小整数是指位于［-5, 256］范围内的整数，这些整数对象是提前创建好 的，不会被当作垃圾回收。因此，位于此范围的整数使用的是同一个对象。

对于其他整数而言，使用赋值语句都会创建一个新的对象，示例如下：

»> a = 257

»> b = 257

»> a is b

False

需要注意的是，只包含单个字符的字符串是共用的，会一直驻留在内存中，示例如下：

»> sl = ° !"

»> s2 = ° ! °

>>> sl is s2

True

14d®d2 mtem

我们来看一段示例程序:

>>> words\_one = "HelloWorld"

>>> words\_two = "HelioWorld"

>>> words\_three = "HelloWorld"

在上述示例中,引用wordsone、wordstwo和words\_three分别指向字符串对象 ?sHelloWorld\试想一下，如果这三个引用指向的不是同一个内存空间，要是有更多的引用指 向字符串!?HelloWorld!\那么就要创建更多的内存空间。显而易见，这样的方式是不可取的。

下面对上述示例进行测试M

>>> words\_one is words\_two

True

>>> words\_two is words\_three

True

通过上述运行结果可以看出，上述三个引用指向的是同一个字符串对象。

在Python^,字符串是最常用的数据类型之一。Python解释器为了提高字符串的使用效率9 提供了 intoni （字符串驻留）技术。intern机制是指同样的字符串对象只保存一份，该字符串会 放到缓冲池中，是共用的。

字符串对象不会一直留在缓冲池中，它使用引用计数进行维护。只要这个字符串对象的引 用计数器为0,就会从缓冲池中销毁。

不过，如果字符串中含有空格、#、!等特殊符号，则Python解释器不会开启intaii机制， 多个引用不会共用该字符串对象，示例如下：

>>> a = !hellozmy name is Jane! f

>>> b = ?hellozmy name is Jane! !

>>> a is b

False