[转载]Python垃圾回收机制--完美讲解!

东皇Amrzs 关注

3 2016.05.24 10:19:08 字数 8,130 阅读 102,454

虽然是自己转载的但是是真的好的一篇图文并茂的对垃圾回收机制的讲解!!!

先来个概述，第二部分的画述才是厉害的。

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**

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 = []
```

推荐阅读

投递过程问题总结
阅读 3,716

（1）线程和进程的定义和区别
阅读 1,916

Ubuntu+GPU+Tensorflow运行tf-faster-rcnn【光纤分类项目】
阅读 72

GCN for Text Classification<总结>
阅读 51

世界上最短的婚姻，只有三分钟
阅读 29,780

[转载]Python垃圾回收机制--完美讲解!

东皇Amrzs 关注 赞赏支持

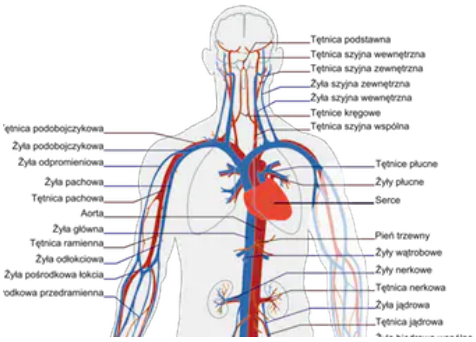
list1与list2相互引用，如果不存在其他对象对它们的引用，list1与list2的引用计数也仍然为1，所占用的内存永远无法被回收，这将是致命的。
对于如今的强大硬件，缺点1尚可接受，但是循环引用导致内存泄露，注定python还将引入新的回收机制。(标记清除和分代收集)

转载地址:http://my.oschina.net/hebianxizao/blog/57367?fromerr=KJozamtm

画说 Ruby 与 Python 垃圾回收

英文原文: visualizing garbage collection in ruby and python
中文:画说 Ruby 与 Python 垃圾回收

本文基于我在刚刚过去的在布达佩斯举行的RuPy上的演讲。我觉得趁热打铁写成帖子应该会比只留在幻灯片上更有意义。你也可以看看演讲录像。再跟你说件事，我在Ruby大会也会做一个相似的演讲，但是我不会去说Python的事儿，相反我会对比一下MRI,JRuby和Rubinius的垃圾回收机制。
想了解Ruby垃圾回收机制和Ruby内部实现更详尽的阐述，请关注即将问世的拙作《Ruby Under a Microscope》。



如果将算法和业务逻辑比作应用程序的大脑，垃圾回收对应哪个器官呢?

既然是"Ruby Python"大会，我觉得对比一下Ruby和Python的垃圾回收机制应该会很有趣。在此之前，到底为什么要计较垃圾回收呢？毕竟，这不是什么光鲜亮丽激动人心的主题，对吧。你们大家有多少人对垃圾回收感冒？(竟然有不少RuPyde与会者举手了!)

最近Ruby社区发表了一篇博文,是关于如何通过更改Ruby GC设置来为单元测试提速的。我认为这篇文章是极好的。对于想让单元测试跑得更快和让程序GC暂停更少的人来说很有裨益，但是GC并没能引起我的兴趣。第一瞥GC就像是一个让人昏昏欲睡的、干巴巴的技术主题。

但是实际上垃圾回收是一个迷人的主题：GC算法不仅是计算机科学史的重要组成部分，也是一个前沿课题。举例来说，MRI Ruby使用的标记-清除算法已经年逾五旬了，而Ruby的替代语言Rubinius使用的GC算法在不久前的2008年才被发明出来。

然而, "垃圾回收"这个词其实有些用词不当。

应用程序那颗跃动的心

GC系统所承担的工作远比"垃圾回收"多得多。实际上，它们负责三个重要任务。它们

- 为新生成的对象分配内存
- 识别那些垃圾对象，并且
- 从垃圾对象那回收内存。

推荐阅读
投递过程问题总结
阅读 3,716
（1）线程和进程的定义和区别
阅读 1,916
Ubuntu+GPU+Tensorflow运行tf-faster-rcnn【光纤分类项目】
阅读 72
GCN for Text Classification<总结>
阅读 51
世界上最短的婚姻，只有三分钟
阅读 29,780

[转载]Python垃圾回收机制--完美讲解!

东皇Amrzs 关注 赞赏支持

案：腰子、白血球 :))

我认为垃圾回收就是应用程序那颗跃动的心。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。

一个简单的例子

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，我们今天就以此为例：

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

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

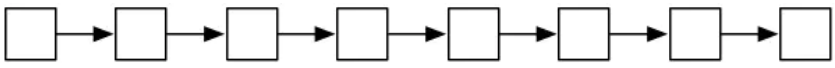
```ruby
class Node
  def initialize(val)
    @value = val
  end
end

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

顺便提一句，两种语言的代码竟能如此相像：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是否也如此相似呢?

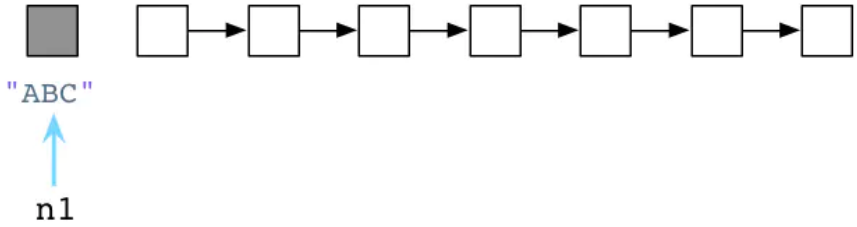
可用列表

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：

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

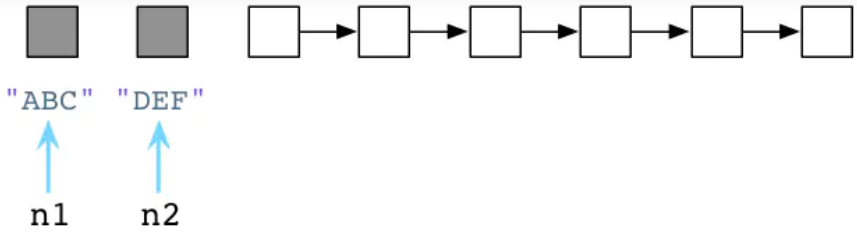


上图中左侧灰格表示我们代码中使用的当前对象，同时其他白格是未使用对象。(请注意：无疑我的示意图是对实际的简化。实际上，Ruby会用另一个对象来装载字符串"ABC",另一个对象装载Node类定义，还有一个对象装载了代码中分析出的抽象语法树，等等)

推荐阅读
投递过程问题总结
阅读 3,716
（1）线程和进程的定义和区别
阅读 1,916
Ubuntu+GPU+Tensorflow运行tf-faster-rcnn【光纤分类项目】
阅读 72
GCN for Text Classification<总结>
阅读 51
世界上最短的婚姻，只有三分钟
阅读 29,780

[转载]Python垃圾回收机制--完美讲解!

东皇Amrzs 关注 赞赏支持



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。

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。

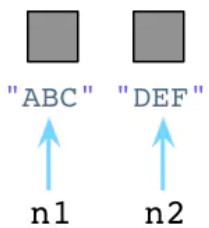
例如我们用Pyhon来创建一个Node对象:

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

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:

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



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。

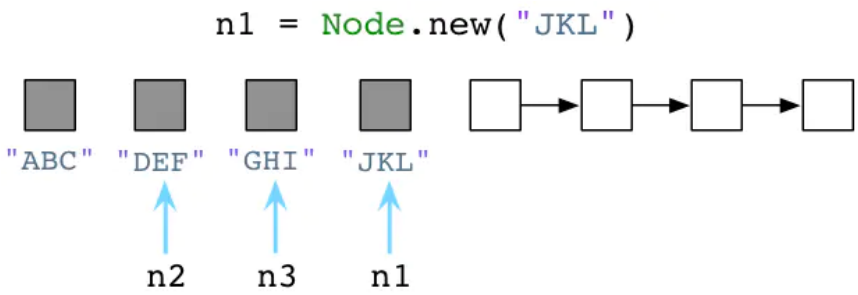
推荐阅读
投递过程问题总结
阅读 3,716
（1）线程和进程的定义和区别
阅读 1,916
Ubuntu+GPU+Tensorflow运行tf-faster-rcnn【光纤分类项目】
阅读 72
GCN for Text Classification<总结>
阅读 51
世界上最短的婚姻，只有三分钟
阅读 29,780

[转载]Python垃圾回收机制--完美讲解!

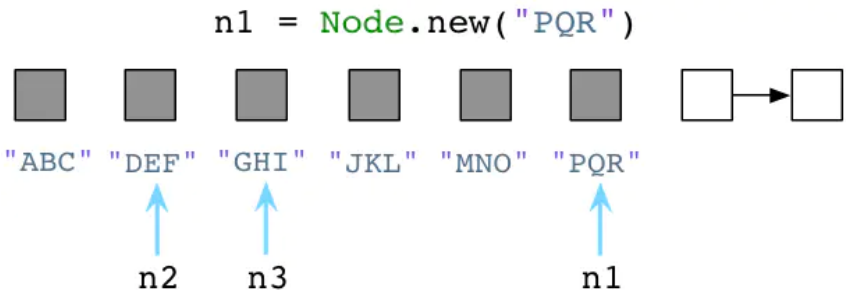
东皇Amrzs 关注 赞赏支持

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:

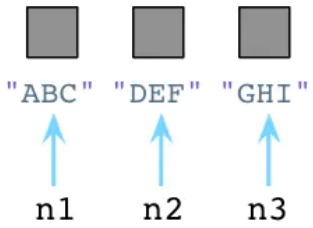


请注意我一直在为变量n1赋新值，Ruby把旧值留在原处。"ABC","JKL"和"MNO"三个Node实例还滞留在内存中。Ruby不会立即清除代码中不再使用的旧对象！Ruby开发者们就像是住在一间凌乱的房间，地板上摞着衣服，要么洗碗池里都是脏盘子。作为一个Ruby程序员，无用的垃圾对象会一直环绕着你。

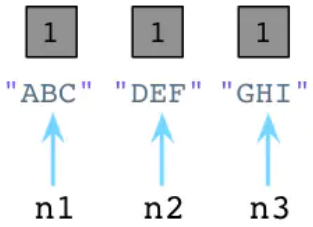
Python 开发者住在卫生之家庭

[转载]Python垃圾回收机制--完美讲解!

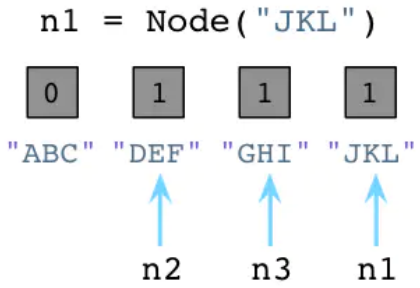
东皇Amrzs 关注 赞赏支持



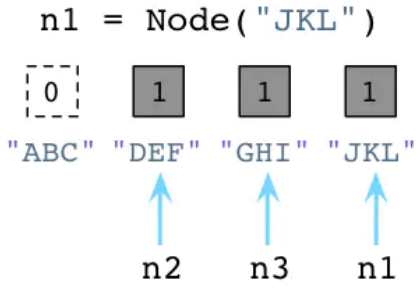
在内部，创建一个对象时，Python总是在对象的C结构体里保存一个整数，称为 *引用数*。期初，Python将这个值设置为1：



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：

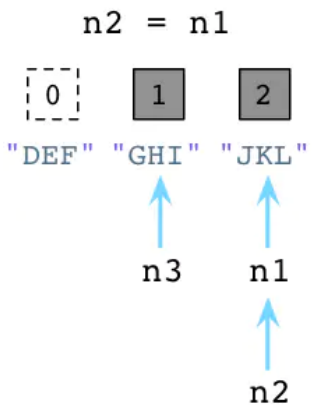


上面Python回收了ABC Node实例使用的内存。记住，Ruby弃旧对象原地于不顾，也不释放它们的内存。

Python的这种垃圾回收算法被称为*引用计数*。是George-Collins在1960年发明的，恰巧与John McCarthy发明的*可用列表算法*在同一年出现。就像Mike-Bernstein在6月份哥谭市Ruby大会杰出的垃圾回收机制演讲中说的: "1960年是垃圾收集器的黄金年代..."

[转载]Python垃圾回收机制--完美讲解!

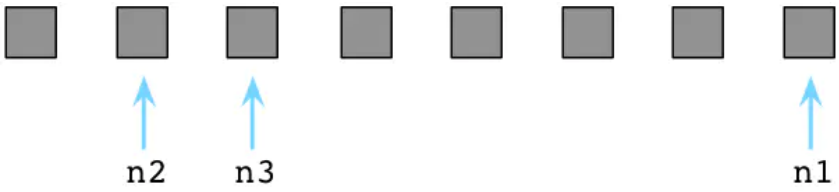
东皇Amrzs 关注 赞赏支持



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。

标记-清除

最终那间凌乱的房间充斥着垃圾，再不能岁月静好了。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通过自身的虚拟机便利内部指针。标记出这些指针引用的每个对象。我在图中使用M表示。

上图中那三个被标M的对象是程序还在使用的。在内部，Ruby实际上使用一串位值，被称为:可用位图(译注：还记得《编程珠玑》里的为突发排序吗，这对离散度不高的有限整数集合具有很强的压缩效果，用以节约机器的资源。)，来跟踪对象是否被标记了。

推荐阅读

投递过程问题总结
阅读 3,716

（1）线程和进程的定义和区别
阅读 1,916

Ubuntu+GPU+Tensorflow运行tf-faster-rcnn【光纤分类项目】
阅读 72

GCN for Text Classification<总结>
阅读 51

世界上最短的婚姻，只有三分钟
阅读 29,780

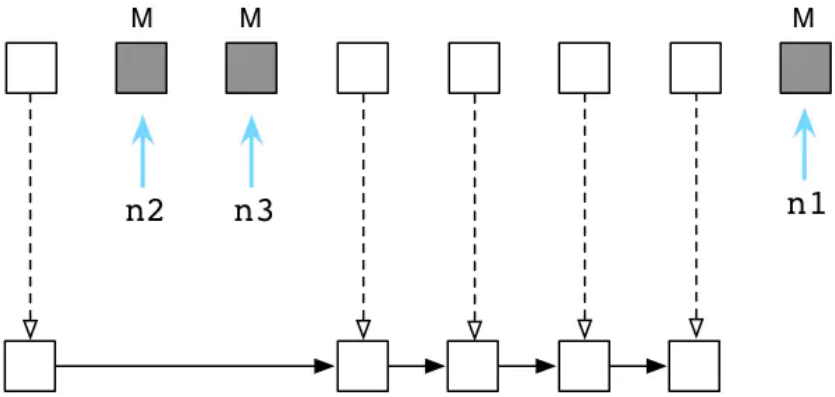
[转载]Python垃圾回收机制--完美讲解!

东皇Amrzs 关注 赞赏支持

Ruby将这个可用位图存放在独立的内存区域中，以便充分利用Unix的写时拷贝化。有关此事的更多内容请关注我另一博文《Why You Should Be Excited About Garbage Collection in Ruby 2.0》

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。
现在等到下回再创建对象的时候Ruby又可以把这些垃圾对象分给我们使用了。在Ruby里，对象们六道轮回，转世投胎，享受多次人生。

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。

推荐阅读
投递过程问题总结
阅读 3,716
（1）线程和进程的定义和区别
阅读 1,916
Ubuntu+GPU+Tensorflow运行tf-faster-rcnn【光纤分类项目】
阅读 72
GCN for Text Classification<总结>
阅读 51
世界上最短的婚姻，只有三分钟
阅读 29,780

[转载]Python垃圾回收机制--完美讲解!

东皇Amrzs 关注 赞赏支持

最后，它不是总奏效的。在我的下一篇包含了我这个演讲剩余部分笔记的文章中，我们会看到，引用计数不能处理环形数据结构--也就是含有循环引用的数据结构。

下回分解

下周我会分解演讲的剩余部分。我会讨论一下Python如何摆平环形数据类型及GC在即将出炉的Ruby2.1发行版中是如何工作的。

对比Ruby和Python的垃圾回收（2）

英文原文地址: Generational GC in Python and Ruby
中文原文: 对比Ruby和Python的垃圾回收（2）：代式垃圾回收机制

上周，我根据之前在RuPy上做的一个名为"Visualizing Garbage Collection in Ruby and Python."的报告写了这篇文章的上半部分。在上篇中，我解释了标准Ruby(也被称为Matz的Ruby解释器或是MRI)是如何使用名为标记回收(Mark and Sweep)的垃圾回收算法，这个算法是为1960年原版本的Lisp所开发。同样，我也介绍了Python使用一种有53年历史的GC算法，这种算法的思路非常不同，称之为引用计数。

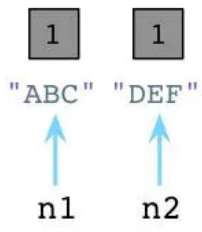
事实证明，Python在引用计数之外，还用了另一个名为 `Generational Garbage Collection` 的算法。这意味着Python的垃圾回收器用不同的方式对待新创建的以及旧有的对象。并且在即将到来的2.1版本的MRI Ruby中也首次引入了Generational Garbage Collection 的垃圾回收机制(在另两个Ruby的实现：JRuby和Rubinius中，已经使用这种GC机制很多年了，我将在下周的RubyConf大会上将它是如何在这两种Ruby实现中工作的)。

当然，这句话"用不同的方式对待新创建的以及旧有的对象"是有点模糊不清，比如如何定义新、旧对象？又比如对于Ruby和Python来说具体是如何采取不同的对待方式？今天，我们就来谈谈这两种语言GC机制的运行原理，回答上边那些疑问。但是在我们开始谈论Generational GC之前，我们先要花点时间谈论下Python的引用计数算法的一个严重的理论问题。

Python中的循环数据结构以及引用计数

通过上篇，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")
```

推荐阅读

投递过程问题总结
阅读 3,716

（1）线程和进程的定义和区别
阅读 1,916

Ubuntu+GPU+Tensorflow运行tf-faster-rcnn【光纤分类项目】
阅读 72

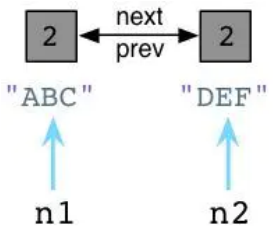
GCN for Text Classification<总结>
阅读 51

世界上最短的婚姻，只有三分钟
阅读 29,780

[转载]Python垃圾回收机制--完美讲解!

东皇Amrzs 关注 赞赏支持

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。(声明下我不是一个Python程序员，所以可能会存在一些命名方面的错误)。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)。


```
n1 = None
n2 = None
```

好了，Python会像往常一样将每个节点的引用计数减少到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：

推荐阅读

投递过程问题总结
阅读 3,716

（1）线程和进程的定义和区别
阅读 1,916

Ubuntu+GPU+Tensorflow运行tf-faster-rcnn【光纤分类项目】
阅读 72

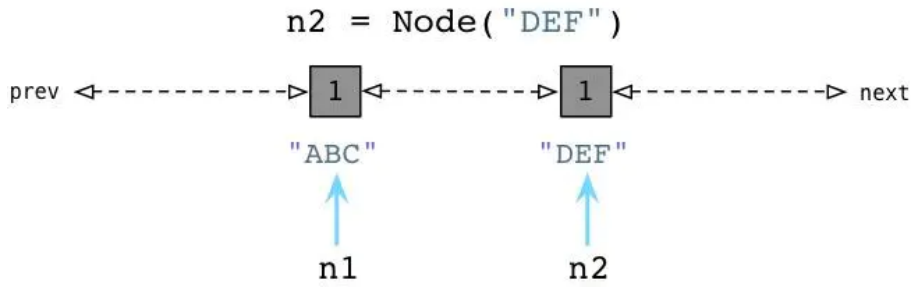
GCN for Text Classification<总结>
阅读 51

世界上最短的婚姻，只有三分钟
阅读 29,780

[转载]Python垃圾回收机制--完美讲解!

东皇Amrzs 关注 赞赏支持

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：

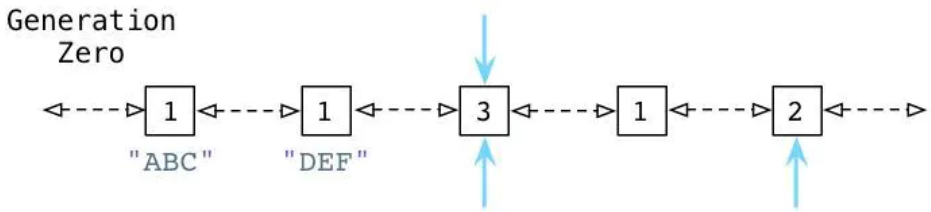


现在零代包含了两个节点对象。(他还将包含Python创建的每个其他值，与一些Python自己使用的内部值。)

检测循环引用

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：

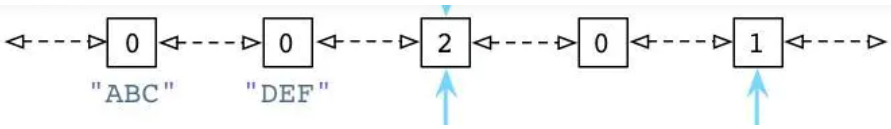


从上面可以看到 ABC 和 DEF 节点包含的引用数为1.有三个其他的对象同时存在于零代链表中，蓝色的箭头指示了有一些对象正在被零代链表之外的其他对象所引用。(接下来我们会看到，Python中同时存在另外两个分别被称为一代和二代的链表)。这些对象有着更高的引用计数因为它们正在被其他指针所指向着。

接下来你会看到Python的GC是如何处理零代链表的。

[转载]Python垃圾回收机制--完美讲解!

东皇Amrzs 关注 赞赏支持



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。

Python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。

弱代假说

来看看代垃圾回收算法的核心行为：垃圾回收器会更频繁的处理新对象。一个新的对象即是你的程序刚刚创建的，而一个来的对象则是经过了几个时间周期之后仍然存在的对象。Python会在当一个对象从零代移动到一代，或是从一代移动到二代的过程中提升(promote)这个对象。

为什么要这么做？这种算法的根源来自于弱代假说(weak generational hypothesis)。这个假说由两个观点构成：首先是年亲的对象通常死得也快，而老对象则很有可能存活更长的时间。

假定现在我用Python或是Ruby创建一个新对象：

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

根据假说，我的代码很可能仅仅会使用ABC很短的时间。这个对象也许仅仅只是一个方法中的中间结果，并且随着方法的返回这个对象就将变成垃圾了。大部分的新对象都是如此般地很快变成垃圾。然而，偶尔程序会创建一些很重要的，存活时间比较长的对象-例如web应用中的session变量或是配置项。

通过频繁的处理零代链表中的新对象，Python的垃圾收集器将把时间花在更有意义的地方：它

推荐阅读
投递过程问题总结
阅读 3,716
（1）线程和进程的定义和区别
阅读 1,916
Ubuntu+GPU+Tensorflow运行tf-faster-rcnn【光纤分类项目】
阅读 72
GCN for Text Classification<总结>
阅读 51
世界上最短的婚姻，只有三分钟
阅读 29,780

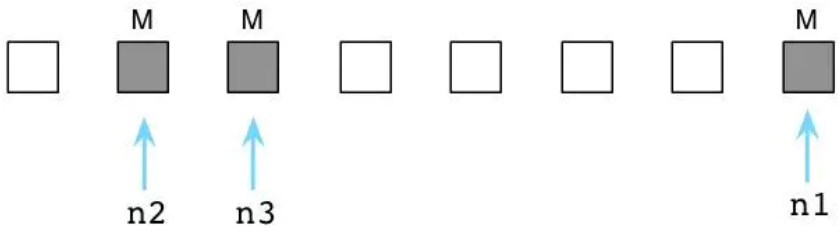
[转载]Python垃圾回收机制--完美讲解!

东皇Amrzs 关注 赞赏支持

回到Ruby的自由链

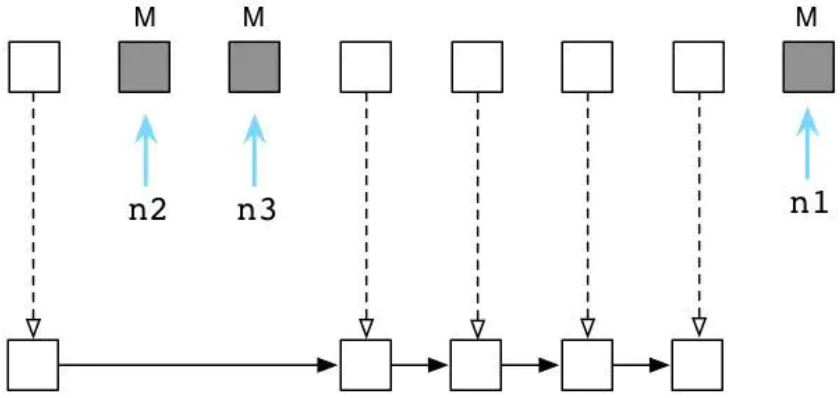
即将到来的Ruby 2.1版本将会首次使用基于代的垃圾回收算法！(请注意的是，其他的Ruby实现，例如JRuby和Rubinius已经使用这个算法许多年了)。让我们回到上篇博文中提到的自由链的图来看看它到底是怎么工作的。

请回忆当自由链使用完之后，Ruby会标记你的程序仍然在使用的对象。



从这张图上我们可以看到有三个活跃的对象，因为指针n1、n2、n3仍然指向着它们。剩下的用白色矩形表示的对象即是垃圾。(当然，实际情况会复杂得多，自由链可能会包含上千个对象，并且有复杂的引用指向关系，这里的简图只是帮助我们了解Ruby的GC机制背后的简单原理，而不会将我们陷入细节之中)

同样，我们说过Ruby会将垃圾对象移动回自由链中，这样的话它们就能在程序申请新对象的时候被循环使用了。



Ruby2.1基于代的GC机制

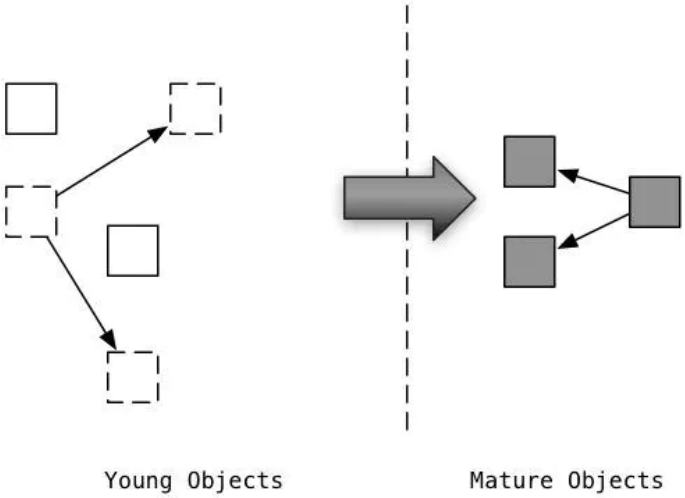
从2.1版本开始，Ruby的GC代码增加了一些附加步骤：它将留下来的活跃对象晋升(promote)到成熟代(mature generation)中。(在MRI的C源码中使用了old这个词而不是mature)，接下来的图展示了两个Ruby2.1对象代的概念图：

[转载]Python垃圾回收机制--完美讲解!

东皇Amrzs 关注 赞赏支持

在左边是一个跟自由链不相同的场景，我们可以看到垃圾对象是用白色表示的，剩下的是灰色的活跃对象。灰色的对象刚刚被标记。

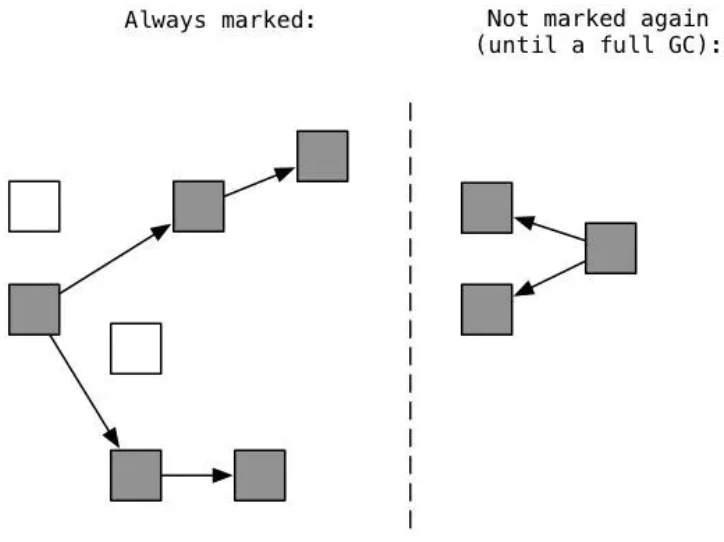
一旦"标记清除"过程结束，Ruby2.1将剩下的标记对象移动到成熟区：



跟Python中使用三代来划分不同，Ruby2.1只用了两代，左边是年轻的新一代对象，而右边是成熟代的老对象。一旦Ruby2.1标记了对象一次，它就会被认为是成熟的。Ruby会打赌剩下的活跃对象在相当长的一段时间内不会很快变成垃圾对象。

重要提醒：Ruby2.1并不会真的在内存中拷贝对象，这些代表不同代的区域并不是由不同的物理内存区域构成。(有一些别的编程语言的GC实现或是Ruby的其他实现，可能会在对象晋升的时候采取拷贝的操作)。Ruby2.1的内部实现不会将在标记&清除过程中预先标记的对象包含在内。一旦一个对象已经被标记过一次了，那么那将不会被包含在接下来的标记清除过程中。

现在，假定你的Ruby程序接着运行着，创造了更多新的，更年轻的对象。则GC的过程将会在新的一代中出现，如图：



推荐阅读

投递过程问题总结
阅读 3,716

（1）线程和进程的定义和区别
阅读 1,916

Ubuntu+GPU+Tensorflow运行tf-faster-rcnn【光纤分类项目】
阅读 72

GCN for Text Classification<总结>
阅读 51

世界上最短的婚姻，只有三分钟
阅读 29,780

[转载]Python垃圾回收机制--完美讲解!

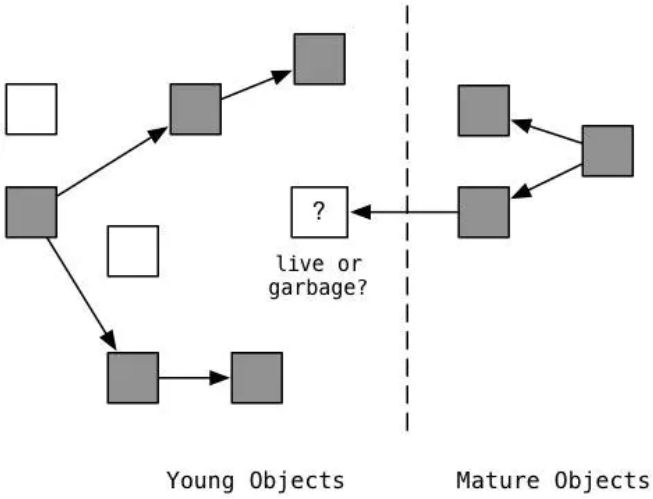
东皇Amrzs 关注 赞赏支持

一次GC过程发生后创建的新的、年轻的对象包含在接下来的标记清除过程中。这是因为很多新对象很可能马上就会变成垃圾(白色标记)。Ruby不会重复标记右边的成熟对象。因为他们已经在一次GC过程中存活下来了，在相当长的一段时间内不会很快变成垃圾。因为只需要标记新对象，所以Ruby 的GC能够运行得更快。它完全跳过了成熟对象，减少了代码等待GC完成的时间。

偶然的Ruby会运行一次"全局回收"，重标记(re-marking)并重清除(re-sweeping)，这次包括所有的成熟对象。Ruby通过监控成熟对象的数目来确定何时运行全局回收。当成熟对象的数目双倍于上次全局回收的数目时，Ruby会清理所有的标记并且将所有的对象都视为新对象。

白障

这个算法的一个重要挑战是值得深入解释的：假定你的代码创建了一个新的年轻的对象，并且将其作为一个已存在的成熟对象的子嗣加入。举个例子，这种情况将会发生在，当你往一个已经存在了很长时间的数组中增加了一个新值的时候：



让我们来看看图，左边的是新对象，而成熟的对象在右边。在左边标记过程已经识别出了5个新的对象目前仍然是活跃的(灰色)。但有两个对象已经变成垃圾了(白色)。但是如何处理正中间这个新建对象？这是刚刚那个问题提到的对象，它是垃圾还是活跃对象呢?

当然它是活跃对象了，因为有一个从右边成熟对象的引用指向它。但是我们前面说过已经被标记的成熟对象是不会被包含在标记清除过程中的(一直到全局回收)。这意味着类似这种的新建对象会被错误的认为是垃圾而被释放，从而造成数据丢失。

Ruby2.1 通过监视成熟对象，观察你的代码是否会添加一个从它们到新建对象的引用来克服这个问题。Ruby2.1 使用了一个名为白障(white barriers)的老式GC技术来监视成熟对象的变化 – 无论任意时刻当你添加了从一个对象指向另一个对象的引用时(无论是新建或是修改一个对象)，白障就会被触发。白障将会检测是否源对象是一个成熟对象，如果是的话则将这个成熟对象添加到一个特殊的列表中。随后，Ruby2.1会将这些满足条件的成熟对象包括到下一次标记清除的范围内，以防止新建对象被错误的标记为垃圾而清除。

Ruby2.1 的白障实现相当复杂，主要是因为已有的C扩展并未包含这部分功能。Koichi Sasada以及Ruby的核心团队使用了一个比较巧妙的方案来解决这个问题。如果想了解更多的内容，请阅读这些相关材料：Koichi在EuRuKo 2013上的演讲Koichi's fascinating presentation。

站在巨人的肩膀上

[转载]Python垃圾回收机制--完美讲解!

来处理循环引用，而两者同时以相似的方式使用基于代的垃圾回收算法。Python划分了三代，而Ruby只有两代。

这种相似性应该不会让人感到意外。两种编程语言都使用了几十年前的计算机科学研究成果来进行设计，这些成果早在语言成型之前就已经被做出来了。我比较惊异的是当你掀开不同编程语言的表面而深入底层，你总能够发现一些相似的基础理念和算法。现代编程语言应该感激那些六七十年代由麦卡锡等计算机先贤所作出的计算机科学开创性研究。

[转载]Python垃圾回收机制--完美讲解!

老哥你确定你是干这行的吗😂

回复

暖遇
2018.03.26 11:00

@东皇Amrzs 小白 小白 老铁 可以加一下 QQ 或者微信吗 有不懂的

回复

东皇Amrzs 作者
2018.03.26 11:23

@暖遇 微信:xzp199498

回复

添加新评论 | 还有1条评论，查看更多

Hanaasagi
7楼 2017.06.19 11:26

那个翻译成白障的应该是 write barrier 而不是 white barrier

赞 回复

Jay54520
6楼 2017.05.06 10:54

感觉 标记-清除 与 分代可以看成一个整体
标记-清除过程先从零代开始，清理最频繁的也是零代，然后是第一代、第二代

赞 回复

暖小希
5楼 2017.05.05 16:33

说的很明白

赞 回复

Luat物联网通信模块
4楼 2017.02.21 01:35

可以关注下Luat开源项目，基于Air200 GPRS模块（价格才12块多），基于Lua开发应用软件，超级容易上手。 完美支持mqtt，而且开发点阵LCD 的UI 非常方便，有丰富的底层库，支持中文很方便。

赞 回复

我是萌大叔
3楼 2016.11.23 10:05

完美

赞 回复

一叶染秋
2楼 2016.11.21 00:07

细致好文啊，我自己找，一下找不到的

赞 回复

东皇Amrzs 作者
2016.11.21 09:11

推荐阅读

投递过程问题总结
阅读 3,716

（1）线程和进程的定义和区别
阅读 1,916

Ubuntu+GPU+Tensorflow运行tf-faster-rcnn【光纤分类项目】
阅读 72

GCN for Text Classification<总结>
阅读 51

世界上最短的婚姻，只有三分钟
阅读 29,780

[转载]Python垃圾回收机制--完美讲解!

东皇Amrzs 关注 赞赏支持

添加新评论

被以下专题收入，发现更多相似内容

Python之旅 python 机器学习 Python python 全栈 Python 展开更多

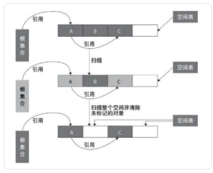
推荐阅读

更多精彩内容>

浅析JAVA的垃圾回收机制（GC）

1.什么是垃圾回收？ 垃圾回收(Garbage Collection)是Java虚拟机(JVM)垃圾回收器提供...

简欲明心 阅读 44,270 评论 18 赞 264



python 高级编程④元类、垃圾回收

一元类 1类也是对象 在大多数编程语言中，类就是一组用来描述如何生成一个对象的代码段。在Python中这一点仍然成...

五行缺觉 阅读 348 评论 0 赞 1

python高级编程2

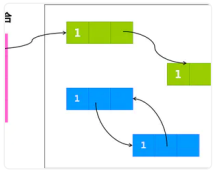
1.元类 1.1.1类也是对象 在大多数编程语言中，类就是一组用来描述如何生成一个对象的代码段。在Python中这...

TENG书 阅读 575 评论 0 赞 3

JAVA垃圾回收机制

来自： Android梦想特工队作者： Aaron主页： http://www.wxtlife.com/原...

技术特工队 阅读 2,679 评论 0 赞 28



一直在路上

你在忙着去哪？ 去远方？ 忙着生？忙着死？ 生活如此的简单，生活又如此的艰难。 好久没有碰过熨斗了，离开有空调的办...

邓宇捷 阅读 142 评论 0 赞 5

推荐阅读

投递过程问题总结
阅读 3,716

（1）线程和进程的定义和区别
阅读 1,916

Ubuntu+GPU+Tensorflow运行tf-faster-rcnn【光纤分类项目】
阅读 72

GCN for Text Classification<总结>
阅读 51

世界上最短的婚姻，只有三分钟
阅读 29,780

写下你的评论... 评论14 赞157 ···