改进 Python 的内存分配器

埃文·琼斯 < ejones@uwaterloo.ca >
http://evanjones.ca/
[PDF 版]

抽象的

Python 使用引用计数自动管理内存。为了给典型程序提供良好的性能，Python 为小对象（≤ 256 字节）提供了自己的内存分配器。但是，原始实现不会向操作系统释放内存，这可能会导致性能问题。本文描述了原始实现是如何工作的，以及问题是如何解决的。它还讨论了可以改进 Python 内存管理的其他领域。

一、简介

Python 的众多优点之一是它可以自动管理内存，就像任何高级语言一样。它使用引用计数来确定何时可以回收对象占用的内存。然而，在幕后，该内存从未真正释放给操作系统。相反，Python 解释器会保留它，并会根据需要重用它。在许多情况下，这是一个很好的策略，因为它可以最大限度地减少内存分配中涉及的操作系统开销。但是，如果 Python 进程长时间运行，它将占用它所需的最大内存量。如果应用程序的峰值内存使用量远大于其平均使用量，这是一种浪费，并且会损害整体系统性能以及应用程序本身的性能。

解决此问题的方法是避免在长期进程中分配大量临时对象。例如，如果一个应用程序必须执行一些昂贵的一次性计算，它可以在另一个进程中使用 计算`fork()`，然后通过管道传回结果。或者，可以将数据存储在文件系统中，而不是使用一些大型临时列表或字典。这些黑客破坏了自动内存管理的目的。程序员不需要被迫考虑何时将内存归还给操作系统。

为了了解 Python 如何管理内存，我们首先概述 Python 的内存分配器是如何工作的，然后再描述解决问题的修改。然后，我们讨论了这种更改的优缺点，最后得出了一些可以进一步提高 Python 内存效率的改进。

2.pymalloc分配器

Python 的内存分配器，称为 pymalloc，由 Vladimir Marangozov 编写，最初是 Python 2.1 和 2.2 的实验性功能，在 2.3 中默认启用之前。Python 使用了很多经常被创建和销毁的小对象，并且为每个对象调用`malloc()`and会带来很大的开销。`free()`为了避免这种情况，pymalloc 以 256 kB 的块分配内存，称为 arenas。arena 被划分为 4 kB 的池，这些池又被细分为固定大小的块，如图 1 所示。这些块返回给应用程序。要了解分配器的详细信息，我们将逐步介绍在需要块时以及释放块时发生的过程。

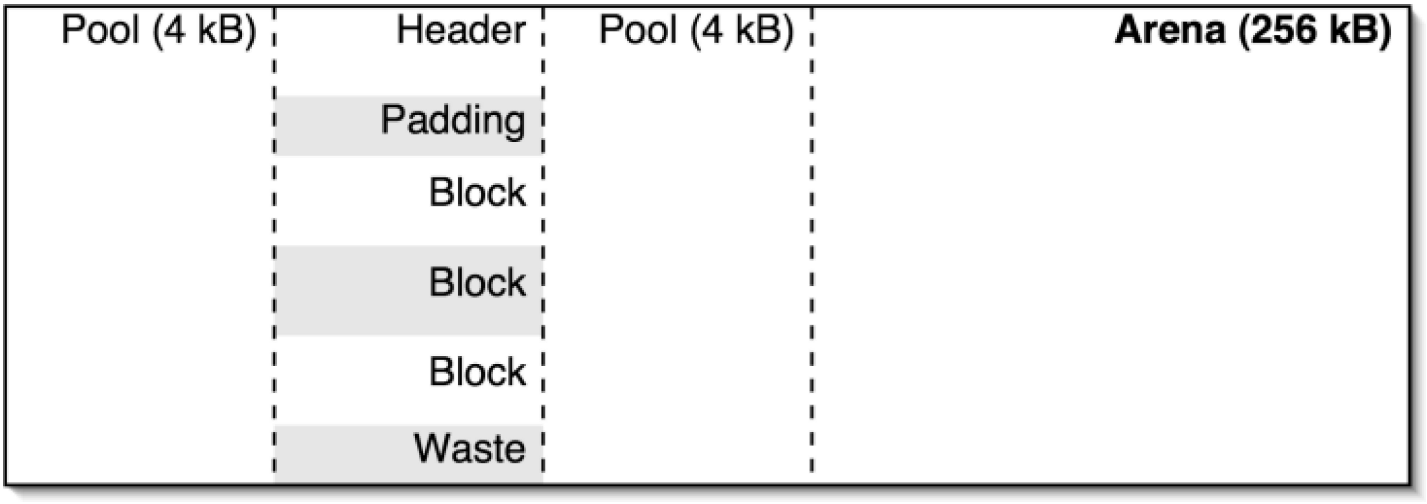

图 **1**：竞技场、池和块的内存布局。

2.1。分配内存

在分配新对象时，第一个分配器会查看是否有任何池已被划分为所需大小的块。该`usedpools`数组，如图 2 所示，包含具有各种大小的块的池的链接列表。每个池都有一个可用块的单链表。如果有一个池，我们从它的列表中弹出一个块。这是最常见的情况，它非常快，因为它只需要几次内存读取和一次写入。如果该块是池中的最后一个，那么它现在已被完全分配。在这种情况下，我们也将池弹出。最后，我们将块返回给应用程序。

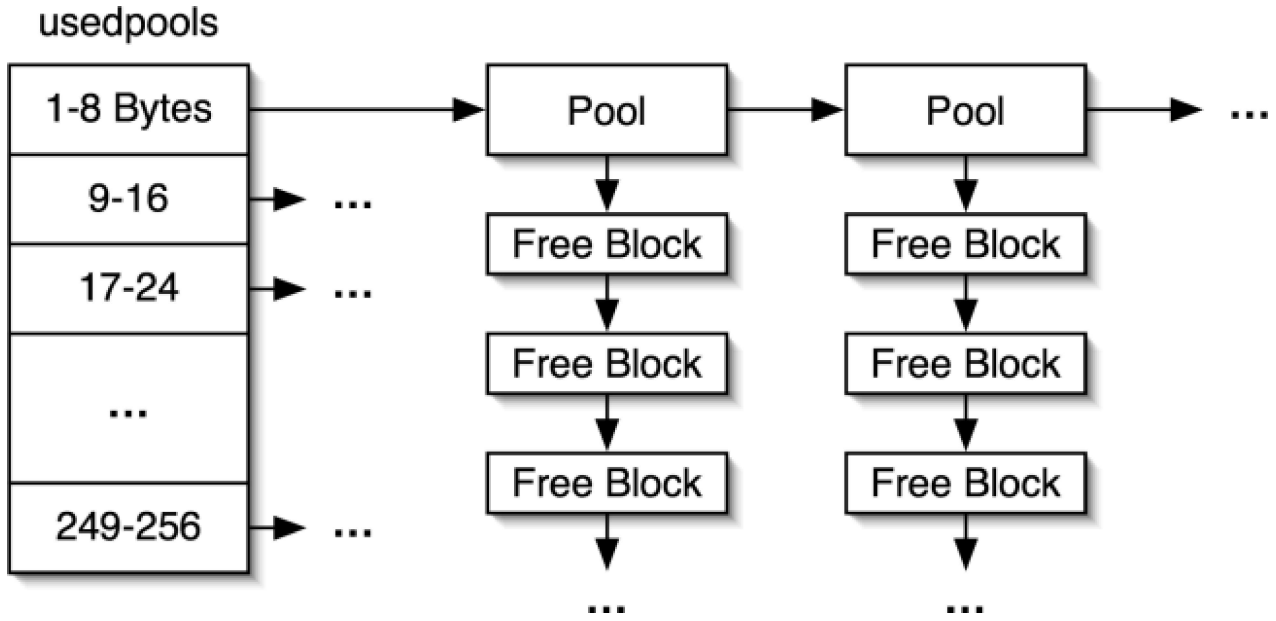

图 **2**：**`usedpools`**用于存储部分分配的池的数组

如果没有正确大小的池，我们需要找到一个可用的池。图 3 所示的`freepools`链表包含可用的池。如果列表上有一个池，我们将其弹出。否则，我们需要创建一个新池。如果在我们分配的最后一个 arena 的末尾还有空间，我们可以使用`arenabase`指针切断另一个池，如图 3 所示。如果没有空间，我们调用`malloc()`分配一个新的 arena。现在我们终于有了一个池，我们将它分成固定大小的块，放入池中，`usedpools`最后将一个块返回给应用程序。

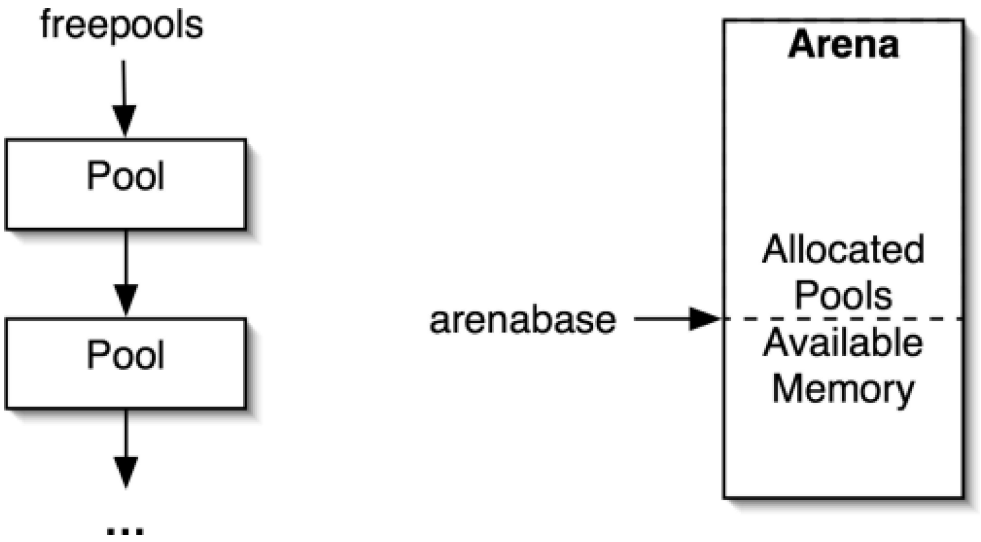

图 3：**freepools**列表和分配新池。

2.2. 释放内存

当应用程序释放块时，过程非常相似。首先，为了确定一个块属于哪个池（参见obmalloc.c 血腥细节），执行了一些魔法。然后将该块放置在池的空闲列表中。如果池已完全分配，则将其添加到usedpools数组中。如果池现在完全可用，则将其从列表中删除usedpools 并添加到freepools 列表中。

3. 解决问题

问题是该过程在那里停止。没有代码可以将空闲池与其所属的 arena 匹配，然后将空闲的 arenas 返回给操作系统。为此，需要将一个免费池专门针对它来自的竞技场放在一个列表中。一旦竞技场中的所有池都在该列表中，就可以释放整个竞技场。

这意味着某些数据结构必须与每个 arena 相关联，以跟踪可用和分配的池。维护了一系列 arena 结构，以便在操作 arena 时提供良好的内存局部性。下一个挑战是从一个游泳池到分配它的竞技场。为此，池标头被扩展为在数组中包含竞技场的索引。此外，freepools 列表已更改为partially_allocated_arenas 列表。此列表（如图 4 所示）跟踪具有可用池的竞技场。通过这些更改，当需要空闲池时，需要修改第 2.1 节中描述的分配内存的过程。不是从freepools列表中获取池，而是从列表中的第一个竞技场中获取池partially_allocated_arenas 。

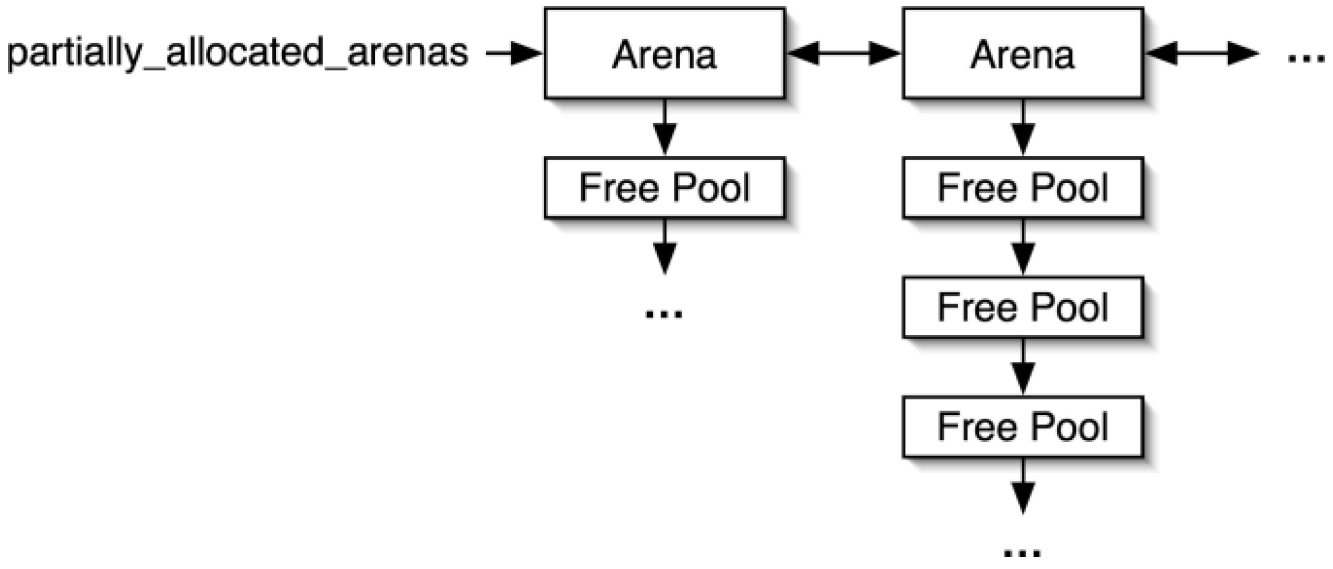

图 4：该**partially_allocated_arenas**列表包含带有免费池的竞技场

3.1。释放内存需要两个

对释放内存的过程进行了更重大的更改。如果一个池是完全空闲的，则该池被放置在一个专门针对其竞技场的空闲列表中。如果竞技场已完全分配，则将其添加到`partially_allocated_arenas` 列表中。该链表包含包含一些空闲池的 arenas。如果竞技场现在完全可用，则将其从`partially_allocated_arenas`列表中删除，并通过调用将其释放到操作系统`free()`。另一个调整是`partially_allocated_arenas` 列表保持排序，以便在需要池时首先使用分配最多的竞技场。这使几乎空无一物的 arena 有机会完全释放并返回到操作系统，并通过实验确定提高可以释放的内存量。

4. 影响

这些更改不会影响程序的正确性，只会影响它们的性能。新版本的分配器能够将内存返回给操作系统，这应该会提高以突发方式分配内存的应用程序的整体系统性能。缺点是释放块时需要一些额外的开销，以跟踪它们的来源。对于循环分配和释放内存的程序，还有重复调用`malloc()` 和的额外开销`free()`。以前不存在开销，因为 Python 会保持所需的最大 RAM 量。然而，这个额外的开销应该很小，因为内存分配器中的公共代码路径没有改变。

不幸的是，这个补丁只能在不再分配更多对象的情况下释放竞技场。这意味着碎片化是一个大问题。一个应用程序可能有许多兆字节的空闲内存，分散在所有领域，但它无法释放任何一个。这是所有内存分配器都会遇到的问题。解决它的唯一方法是移动到一个压缩垃圾收集器，它能够移动内存中的对象。这将需要对 Python 解释器进行重大更改。

4.1。示例应用程序

为了说明这个补丁的效果，考虑图 5 所示的简单程序。它分配大量字典，释放它们，最后再次分配它们。图 6 显示了一段时间内的内存使用情况。理想的分配器显示程序分配了大约 550 MB，然后释放除 50 MB 之外的所有内存，重新分配，最后退出。原始分配器始终保持最大内存量，而新分配器紧跟理想的分配器行为。对于这个应用程序，新的分配器实际上是有害的，因为内存会立即被重用。但是，如果两次分配之间的时间很长，那么新的分配器将非常有用。

```
迭代= 2000000 l = [] for i in xrange (迭代): l 。append ( None ) for i in xrange (
iterations ): l [ i ] = {} for i in xrange ( iterations ): l [ i ] = None for i in xrange (
iterations ): l [

        我] = {}
```

图 **5**：分配和释放大量内存的示例程序

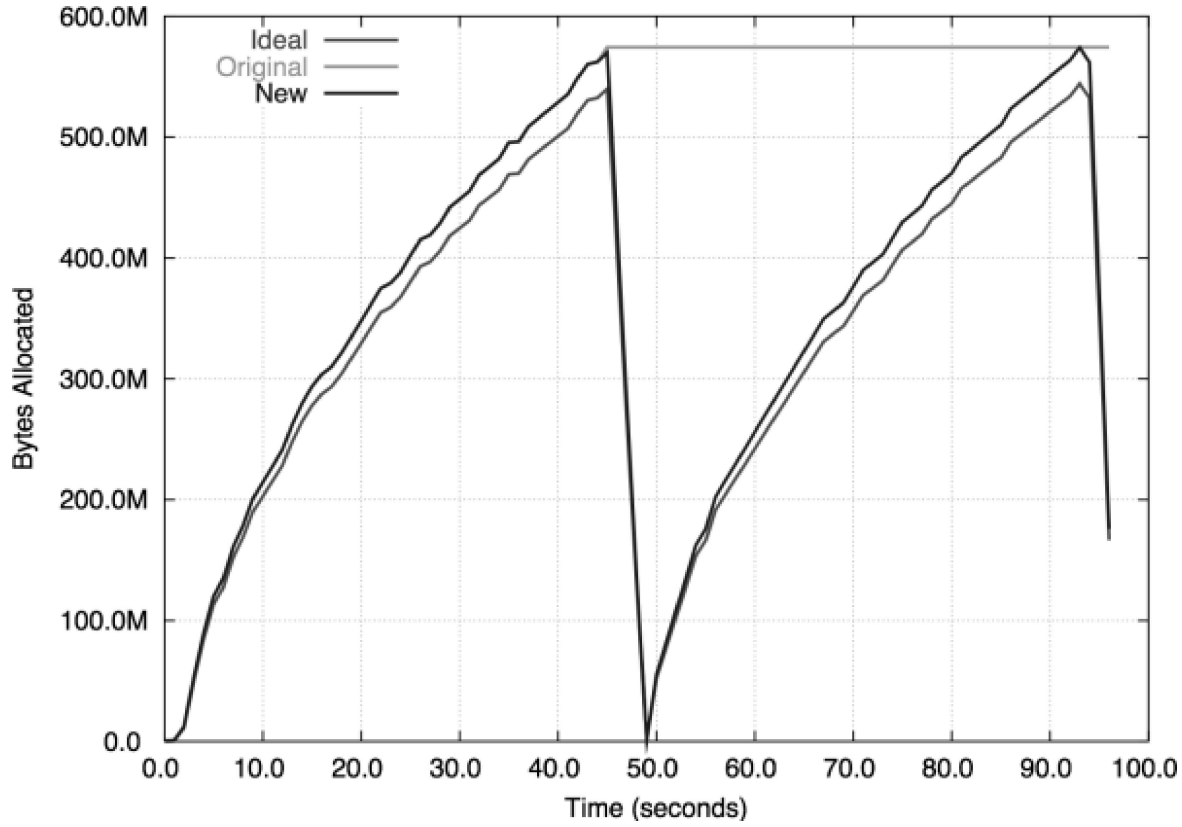

图 **6**：示例程序的内存分配器行为

5. 未来的改进

Python 的内存管理还有一个地方可以改进。有一些对象不使用 pymalloc 分配器。最重要的是 Python 的整数、浮点数、列表和字典。这些数据类型维护自己的空闲对象列表，以便为这些非常常见的对象更有效地利用空间和时间。但是，当前的实现可能会导致与本文中描述的更改试图解决的相同问题。

最严重的违规者是整数和浮点数。这两种对象类型分配它们自己的大约 1kB 的内存块，这些内存块是随系统分配的`malloc()`。这些块用作整数和浮点对象的数组，这避免了 pymalloc 将对象大小四舍五入到最接近的 8 倍数造成的浪费。然后将这些对象链接到一个简单的空闲列表。当需要一个对象时，从列表中取出一个或分配一个新块。当一个对象被释放时，它会返回到空闲列表中。

该方案非常简单且非常快速，但是，它存在一个重大问题：分配给整数的内存永远不能用于其他任何事情。这意味着，如果您编写一个程序，分配 1 000 000 个整数，然后释放它们并分配 1 000 000 个浮点数，Python 将为 2 000 000 个数字对象保留足够的内存。解决方案是应用与上述类似的方法。可以从 pymalloc 请求池，因此它们正确对齐。释放整数或浮点数时，该对象将被放入其特定池的空闲列表中。当不再需要该池时，可以将其返回给 pymalloc。挑战在于这些类型的对象经常使用，因此需要注意确保良好的性能。

字典和列表使用不同的方案。Python 总是最多保留 80 个空闲列表和字典，任何多余的都会被释放。这不是最优的，因为某些应用程序在列表较大时会表现得更好，而其他应用程序则需要较少。自我调整列表大小可能会更有效。

6。结论

对内存分配器的更改使 Python 可以将内存返回给操作系统。对于峰值内存使用量远高于其平均使用量的长时间运行的应用程序来说，这是一个问题。对于这些应用程序，这可以大大提高系统的整体性能。可以进一步改进 Python 的内存分配器。特别是，在某些 Python 程序中，为整数和浮点数维护的空闲列表可能会浪费大量内存。