

第 12 章

优化：通用原则和剖析技术

“过早进行优化是编程中的万恶之源。”

——Donald Knuth

本章是关于优化的，将介绍一组通用原则和剖析技术。它讲解了开发人员应该掌握的 3 条规则，并提供了优化的指南。最后，它还将关注于如何查找瓶颈。

12.1 优化的 3 条规则

不管结果如何，优化工作都是有代价的。当代码工作正常时，不去理会它（有时）可能比不惜一切代价尝试使它运行得更快要好一些。进行优化时，需要记住几条原则：

- 首先要使它能够正常工作；
- 从用户的观点进行；
- 保持代码易读。

12.1.1 首先使它能够正常工作

尝试在编写代码的同时对其进行优化，是最常见的错误。这是不可能的，因为瓶颈常常会出现在意想不到的地方。

应用程序是由非常复杂的交互组成的，在实际使用之前不可能得到一个完整的视图。

当然，这不是不尝试尽可能编写更快的函数或方法的理由。应该认真地使其复杂度尽可能地降低，并避免无用的重复。但是，首要目标是使它能够正常工作，优化不应该阻碍这一目标的实现。

对于行级的代码，Python 的哲学是完成一个目标尽可能有且只有一个方法。所以，只要坚持第 2 章和第 3 章中介绍的 Python 风格的语法，代码就应该会很好。往往，编写的代码越少，代码就越好并且越快。

在使代码能够正常工作并且做好剖析的准备之前，不要做以下的事情：

- 开始编写为函数缓存数据的全局字典；
- 考虑以 C 语言或诸如 Pyrex 之类混杂语言来对代码的一个部分进行扩展；
- 寻找外部程序库来完成一些基本计算。

对于非常专业的程序，如科学计算程序或游戏，专业的程序库和扩展的使用从一开始就无法避免。另一方面，使用像 Numeric 这样的程序库，可以简化专业功能的开发并且能够生成更简单、更快速的代码。而且，不应该在有很好的程序库可利用的时候自己重新编写一个函数。

例如，Soya 3D 是一个基于 OpenGL 的游戏引擎（参见 <http://home.gna.org/oomadness/en/soya3d/index.html>），它是使用 C 和 Pyrex 开发的，用于在显示实时的 3D 图形时执行快速的矩阵操作。



优化应该在已经能够正常工作的程序上进行。
“先让它能够正常工作，然后让它变得更好，最好使它更快。”

—— Kent Beck

12.1.2 从用户的观点进行

我曾经看到过一些团队对应用程序服务器的启动时间进行优化，之后这些应用程序服务器在启动时真地表现得很好。他们提升了速度之后，就向他们的客户推销这一改进。但是，他们落空了，因为客户对此并不关心。这是因为速度提升的优化并不是由客户的反馈推动的，而是出自于开发人员的观点。构建该系统的人每天都要启动服务器，所以启动时间对他们很重要，但是对客户并不重要。

虽然程序能够更快地启动绝对是件好事，但是团队应该认真地安排优化的优先级，并且问自己以下问题：

- 客户是否要求提升它的速度？
- 谁发现程序慢了？
- 它真的很慢，还是可以接受？
- 提升它的速度需要多少成本？值得吗？哪部分需要提升速度？

记住，优化是有成本的，开发人员的观点对客户来说也许并没有意义，除非编写的是框架或程序库，客户也是开发人员。



优化不是一个游戏，它应该只在必要时进行。



12.1.3 保持代码易读（从而易于维护）

尽管 Python 尝试使公共的代码模式能够最快地执行，但优化工作可能使代码变得混乱，并且使它难以阅读。在生成易读从而易于维护的代码和使其运行速度更快之间，要找到一个平衡点。

当达到优化目标的 90% 时，如果剩下的 10% 会使代码完全无法理解，那么，停止优化工作并且寻求其他解决方案可能是个好主意。



优化不应该使代码难以理解。如果发生这种情况，应该寻求替代的方案，如扩展或重新设计。不过，在代码易读性和速度上总会有一个好的折衷方案。



12.2 优化策略

假设程序有一个需要解决的速度问题，不要尝试猜测如何才能提升它的速度。瓶颈往往难以在代码中找到，需要一组工具来找到真正的问题。

一个好的优化策略可以从以下 3 个步骤开始。

- 寻找其他原因：确定第三方服务器或资源不是问题所在。
- 度量硬件：确定资源足够用。
- 编写速度测试：创建带有速度要求的场景。

12.2.1 寻找其他原因

往往，性能问题会发生在生产环境中，客户会提醒软件的运行和测试环境中不一样。性能问题可能是因为应用程序没有考虑到现实世界中用户数或数据量不断增长的情况。

但是，如果应用程序存在与其他应用程序之间的交互，那么首先要做的是检查瓶颈是否出现在这些交互上。例如，数据库服务器或 LDAP 服务器可能带来额外的开销，并且使得速度变慢。

应用程序之间的物理链接也应该考虑：可能应用程序服务器和其他企业内网中的服务器

之间的网络链接因为错误配置而变慢，或者一个多疑的防病毒软件对所有 TCP 包进行扫描而导致速度降低。

设计文档应该提供描述所有交互和每个链接特性的一个图表，以获得对系统的全面认识，并且在解决速度问题时提供帮助。



如果应用程序使用了第三方服务器或资源，那么应该评估每个交互，以确定瓶颈不在那里。

12.2.2 度量硬件

当内存不够用时，系统会使用硬盘来存储数据，这也就是页面交换（swapping）。

这会带来许多的开销，性能将急剧下降。从用户的角度看，系统在这种时候会被认为处于死机状态。所以，度量硬件的能力以避免这种情况是很重要的。

虽然系统上有足够的内存很重要，但是确保应用程序不要疯狂表演，并吞噬太多内存也是很重要的。例如，如果程序将使用数百兆大小的视频文件，那么它不应该将该文件全部装入内存中，而应该分块载入或者使用磁盘流。

磁盘的使用也很重要。如果代码中隐藏 I/O 错误，尝试重复写入磁盘，那么一个已满的分区也可能降低应用程序的速度。而且，即使代码只写入一次，硬件和操作系统也可能会尝试多次写入。



Munin 是一个很好的系统监控工具，可以用它来获得系统健康情况的一个快照，在<http://munin.projects.linpro.no>上可以找到它。应确保系统健康并且符合应用程序需求。不过，仍应确保应用程序不要像怪物一样消费内存和磁盘空间。

12.2.3 编写速度测试

开始优化工作时，应该在测试程序上花工夫，而不是不断进行手工的测试。一个好的做法是在应用程序中专门制作一个测试模块，编写一系列针对优化的调用。这种方案可在优化应用程序的同时跟踪进度。

甚至可以编写一些断言，在这里设置一些速度要求。为了避免速度退化，这些测试可以留到代码被优化之后，如下所示。

```
>>> def test_speed():
...     import time
...     start = time.time()
...     the_code()
...     end = time.time() - start
...     assert end < 10, \
...         "sorry this code should not take 10 seconds !"
... 
```



度量执行速度取决于 CPU 的运行能力。在下一节将看到如何编写通用的持续时间度量方法。

12.3 查找瓶颈

查找瓶颈可以通过以下几个步骤来完成：

- 剖析 CPU 的使用情况；
- 剖析内存的使用情况；
- 剖析网络的使用情况。

12.3.1 剖析 CPU 的使用情况

瓶颈的第一个来源是代码本身。标准程序库提供了执行代码剖析所需的所有工具，它们都是基于确定性方法的。



确定性剖析程序通过在最低层级上添加一个定时器来度量每个函数所花费的时间。这将带来一些开销，但是它是获取时间消耗信息的一个好思路。另一方面，统计剖析程序将采样指令指针的使用情况而不采样代码。后者比较不精确，但是能够全速运行目标程序。

剖析代码有以下两种方法：

- 宏观剖析 在使用的同时剖析整个程序，并生成统计；

- 微观剖析 人工执行以度量程序中某个确定的部分。

1. 宏观剖析

宏观剖析通过在特殊的模式下运行应用程序来完成，在这种模式下解释程序将收集代码使用情况的统计信息。Python 为此提供了多个工具：

- profile 一个纯 Python 实现的工具；
- cProfile 用 C 实现的，界面和 profile 相同，开销更小；
- hotshot 另一个用 C 实现的工具，可能会被从标准程序库中删除。

除非程序是在 Python 2.5 以下版本中运行，否则建议使用 cProfile。

以下是拥有一个主函数的 myapp.py 模块。

```
import time
def medium():
    time.sleep(0.01)
def light():
    time.sleep(0.001)
def heavy():
    for i in range(100):
        light()
        medium()
        medium()
    time.sleep(2)
def main():
    for i in range(2):
        heavy()
if __name__ == '__main__':
    main()
```

该模块可以直接从提示符调用，在此摘要显示了它的执行结果。

```
$ python -m cProfile myapp.py
      1212 function calls in 10.120 CPU seconds

Ordered by: standard name
```

| ncalls | totttime | cumtime | percall | file |
|--------|----------|---------|---------|---------------------|
| 1 | 0.000 | 10.117 | 10.117 | myapp.py:16(main) |
| 400 | 0.004 | 4.077 | 0.010 | myapp.py:3(lighter) |
| 200 | 0.002 | 2.035 | 0.010 | myapp.py:6(light) |
| 2 | 0.005 | 10.117 | 5.058 | myapp.py:9(heavy) |

```

3      0.000      0.000      0.000 {range}
602  10.106  10.106      0.017 {time.sleep}

```

所提供的统计是一个由剖析程序生成的统计对象的打印视图。手工调用该工具的结果可能如下所示。

```

>>> from myapp import main
>>> import cProfile
>>> profiler = cProfile.Profile()
>>> profiler.runcall(main)
>>> profiler.print_stats()
      1209 function calls in 10.140 CPU seconds
Ordered by: standard name
ncalls  tottime  cumtime  percall   file
    1      0.000   10.140    10.140 myapp.py:16(main)
   400      0.005    4.093     0.010 myapp.py:3(medium)
   200      0.002    2.042     0.010 myapp.py:6(light)
     2      0.005   10.140     5.070 myapp.py:9(heavy)
     3      0.000    0.000     0.000 {range}
   602    10.128   10.128     0.017 {time.sleep}

```

统计也可以保存到一个文件中，然后由 `pstats` 模块阅读。这个模块提供一个处理剖析文件的类，并给出了一些诸如排序之类的助手方法，如下所示。

```

>>> cProfile.run('main()', 'myapp.stats')
>>> import pstats
>>> p = pstats.Stats('myapp.stats')
>>> p.total_calls
1210
>>> p.sort_stats('time').print_stats(3)
Thu Jun 19 23:56:08 2008      myapp.stats
      1210 function calls in 10.240 CPU seconds
Ordered by: internal time
List reduced from 8 to 3 due to restriction <3>
ncalls  tottime  cumtime  percall filename:lineno(function)
   602    10.231   10.231    0.017 {time.sleep}
     2      0.004   10.240    5.120 myapp.py:9(heavy)
   400      0.004    4.159    0.010 myapp.py:3(medium)

```

由此，可以打印输出每个函数的调用者和被调用者，以便浏览代码，如下所示。


```
>>> p.print_callees('medium')
Ordered by: internal time
List reduced from 8 to 1 due to restriction <'medium'>
Function          called...
                  ncalls      tottime cumtime
myapp.py:3(lighter) ->    400      4.155   4.155   {time.sleep}
>>> p.print_callers('light')
Ordered by: internal time
List reduced from 8 to 2 due to restriction <'light'>
Function          was called by...
                  ncalls      tottime cumtime
myapp.py:3(medium) <-    400      0.004   4.159   myapp.py:9(heavy)
myapp.py:6(light) <-    200      0.002   2.073   myapp.py:9(heavy)
```

对输出进行排序，可以从不同的视图来查找瓶颈。例如：

- 当调用数量确实很高，并且占据了最多的时间，那么该函数或方法可能进入了一个循环，可以尝试对其进行优化；
- 当一个函数运行时间很长，如果可能，缓存可能是个不错的选择。

另一个从剖析数据中显示瓶颈的好办法是将其转化为图表。用 Gprof2Dot 工具 (<http://code.google.com/p/jrfonseca/wiki/Gprof2Dot>) 就可以将剖析数据以散点图形式展现（如图 12.1 所示）。可以从 <http://jrfonseca.googlecode.com/svn/trunk/gprof2dot/gprof2dot.py> 下载这个简单的脚本，然后在统计时使用它，只要安装了 Graphviz 就可以（参见 <http://www.graphviz.org/>）。

```
$ wget http://jrfonseca.googlecode.com/svn/trunk/gprof2dot/
gprof2dot.py
$ python2.5 gprof2dot.py -f pstats myapp.stats | dot -Tpng -o output.png
```

 KcacheGrind 也是一个很好的可视化显示剖析数据的工具（参见 <http://kcachegrind.sourceforge.net/cgi-bin/show.cgi>）。

宏观剖析对于发现有问题的函数，或者至少是与问题临近的函数而言是很好方法。当发现问题时，可以转用微观剖析。

2. 微观剖析

当找到速度很慢的函数时，有时还必须做更多只测试程序某个部分的剖析工作。这需要

通过手动对一部分代码进行速度测试来完成。

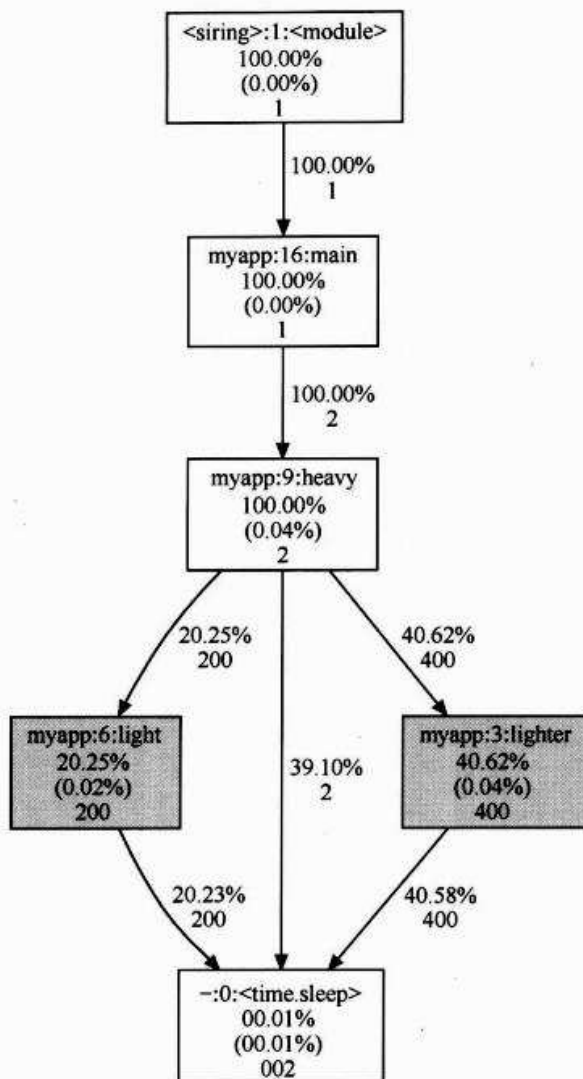


图 12.1

例如，可以在一个装饰器中使用 `cProfile`，如下所示。

```
>>> import tempfile, os, cProfile, pstats
>>> def profile(column='time', list=5):
...     def _profile(function):
...         def __profile(*args, **kw):
...             s = tempfile.mktemp()
...             profiler = cProfile.Profile()
...             profiler.runcall(function, *args, **kw)
...             profiler.dump_stats(s)
...             p = pstats.Stats(s)
```

```

...         p.sort_stats(column).print_stats(list)
...         return __profile
...     return _profile
...
>>> from myapp import main
>>> @profile('time', 6)
... def main_profiled():
...     return main()
...
>>> main_profiled()
Fri Jun 20 00:30:36 2008 ...
    1210 function calls in 10.129 CPU seconds
Ordered by: internal time
List reduced from 8 to 6 due to restriction <6>
ncalls  tottime  cumtime  percall  filename:lineno(function)
    602   10.118   10.118   0.017   {time.sleep}
      2    0.005   10.129   5.065   myapp.py:9(heavy)
    400    0.004    4.080   0.010   myapp.py:3(lighter)
    200    0.002    2.044   0.010   myapp.py:6(light)
      1    0.000   10.129  10.129   myapp.py:16(main)
      3    0.000    0.000   0.000   {range}
>>> from myapp import lighter
>>> p = profile()(lighter)
>>> p()
Fri Jun 20 00:32:40 2008    /var/folders/31/
31iTrMYWHny8cxfjH5VuTk+++TI/-Tmp-/tmpQjstAG
    3 function calls in 0.010 CPU seconds
Ordered by: internal time
ncalls  tottime  cumtime  percall  filename:lineno(function)
      1    0.010    0.010   0.010   {time.sleep}
      1    0.000    0.010   0.010   myapp.py:3(lighter)

```

这种方法能够对应用程序的各部分进行测试，使统计的输出更加精确。

但是在这个阶段，被调用者的列表可能不太令人感兴趣，因为函数已经被作为需要优化的部分。唯一令人感兴趣的是它有多快，然后改进它。

`timeit` 更适合这一需求，它提供一个简单的度量小型代码片断执行时间的方法，它将使用宿主系统提供的最好的底层定时器（`time.time` 或 `time.clock`），如下所示。

```
>>> from myapp import light
>>> import timeit
>>> t = timeit.Timer('main()')
>>> t.timeit(number=5)
10000000 loops, best of 3: 0.0269 usec per loop
10000000 loops, best of 3: 0.0268 usec per loop
10000000 loops, best of 3: 0.0269 usec per loop
10000000 loops, best of 3: 0.0268 usec per loop
10000000 loops, best of 3: 0.0269 usec per loop
5.6196951866149902
```

该模块允许重复调用，它适用于对已隔离的代码片断进行测试。这在应用程序上下文之外非常有用，如在命令提示符下，但是在一个现有的应用程序中使用其并不是很方便。



确定性剖析程序提供的结果依赖于运行它的电脑，因此每次生成的结果都不一样。重复相同的测试并且取其平均值能提供更精确的结果。而且，有些电脑提供了特殊的 CPU 特性，如 SpeedStep。如果电脑在测试启动时处于闲置状态，也将会改变结果。所以对于小的代码片断持续地重复测试是一种好习惯。另外，别忘了有各种缓存，如 DNS 或 CPU 缓存。

和上面相似的装饰器（decorator）是计算应用程序某部分执行时间的更简单方法。这个装饰器将收集程序运行的持续时间，如下所示。

```
>>> import time
>>> import sys
>>> if sys.platform == 'win32':      # same condition in timeit
...     timer = time.clock
... else:
...     timer = time.time
>>> stats = {}
>>> def duration(name='stats', stats=stats):
...     def _duration(function):
...         def __duration(*args, **kw):
...             start_time = timer()
...             try:
```

```

...         return function(*args, **kw)
...         finally:
...             stats[name] = timer() - start_time
...         return __duration
...     return _duration
...
>>> from myapp import heavy
>>> heavy = duration('this_func_is')(heavy)
>>> heavy()
>>> print stats['this_func_is']
1.50201916695

```

在执行代码时，装饰器将填充 stats 词典的内容，这在函数执行完成之后就可以阅读。

使用这样一个装饰器，可以在应用程序中添加嵌入的测试机制，而且不会破坏应用程序本身。

```

>>> stats = {}
>>> from myapp import light
>>> import myapp
>>> myapp.light = duration('myapp.light')(myapp.light)
>>> myapp.main()
>>> stats
{'myapp.light': 0.05014801025390625}

```

这可以在速度测试的上下文中完成。

3. 测量 Pystones

测量执行时间时，结果取决于电脑硬件。为了能产生一个通用的度量值，最简单的方法是测量固定代码序列的基准速度，从而得出一个系数。由此，函数所耗费的时间就可以被转换为一个通用值，以便与其他电脑进行比较。



可用的基准工具有许多，例如创建于 1972 年的 Whetstone，它是一个用 Algol60 编写的电脑性能分析程序（参见 http://en.wikipedia.org/wiki/Whetstone_%28benchmark%29）。它将测量每秒能够执行几百万个 Whetstone 指令 (MWIPS)。它在 <http://freespace.virgin.net/roy.longbottom/whetstone%20results.htm> 上维护了一个结果列表。

Python 在 `test` 包中提供了一个基准测试工具，用来测量一个精心挑选的操作序列的执行消耗时间。结果是电脑每秒能够执行的 `pystones` 数量，以及用于执行基准测试的时间——在当今的硬件环境中的结果通常在 1 秒左右，如下所示。

```
>>> from test import pystone
>>> pystone.pystones()
(1.0500000000000007, 47619.047619047589)
```

这个系数可以用来将剖析得到的消耗时间转化为 `pystones` 数，如下所示。

```
>>> from test import pystone
>>> benchtime, pystones = pystone.pystones()
>>> def seconds_to_kpystones(seconds):
...     return (pystones*seconds) / 1000
...
...
>>> seconds_to_kpystones(0.03)
1.4563106796116512
>>> seconds_to_kpystones(1)
48.543689320388381
>>> seconds_to_kpystones(2)
97.087378640776762
```

`seconds_to_kpystones` 将返回 kilo `pystones` 数（即千个 `pystones`）。这个转换可以包含在 `duration` 装饰器中，以求得以 `stones` 为单位表示的值。

```
>>> def duration(name='stats', stats=stats):
...     def _duration(function):
...         def __duration(*args, **kw):
...             start_time = timer()
...             try:
...                 return function(*args, **kw)
...             finally:
...                 total = timer() - start_time
...                 kstones = seconds_to_kpystones(total)
...                 stats[name] = total, kstones
...         return __duration
...     return _duration
>>> @duration()
... def some_code():
```

```
...     time.sleep(0.5)
...
>>> some_code()
>>> stats
{'stats': (0.50012803077697754, 24.278059746455238)}
```

有了 `pystones` 值，就可以在测试中使用这个装饰器，这样可以在执行时间上设置断言。这些测试将可以在任何电脑上运行，从而使开发人员能够有效避免速度退化。当应用程序的一部分被优化之后，他们能在测试中设置其最大执行时间，以确定进一步的修改没有对它产生影响。

12.3.2 剖析内存使用情况

另一个问题是内存消耗。如果一个程序在运行时消耗太多的内存而导致系统出现页面交换，那么可能是应用程序创建了太多的对象。这往往很容易通过经典的剖析来发现，因为如果程序对内存消耗过大，以至于使系统需要进行页面交换的程度，那么也就会检测到 CPU 所要执行的大量工作。但是有时候它也不明显，因此必须对内存的使用情况进行剖析。

1. Python 处理内存的方式

使用 CPython 时，内存的使用可能是最难剖析的。虽然 C 这样的语言允许获得任何元素所占用的内存大小，但 Python 从不会让你知道指定对象消耗了多少内存。这是因为该语言所具有的动态特性，以及对象实例化的自动管理——垃圾回收。例如，两个指向相同字符串的变量在内存中不一定也指向相同的字符串对象。

这种内存管理器所采用的方法大约是基于这样一个简单的命题：如果指定的对象不再被引用，那么将删除它。所有函数的局部引用在解释程序出现下述情况时将被删除：

- 离开该函数；
- 确定该对象不再需要使用。



在正常情况下，收集器能够工作得很好，但是可以使用 `del` 调用来协助垃圾回收手动删除对象的引用。

所以留在内存中的对象将是：

- 全局对象；
- 仍然以某种方式被引用的对象。

对参数输入输出的边界值要小心。如果一个对象是在参数中创建的，并且函数返回了该对象，那么该参数引用将仍然存在。如果它被作为默认值使用，可能导致不可预测的结果，如下所示。

```
>>> def my_function(argument={}): #不良实践
...     if '1' in argument:
...         argument['1'] = 2
...     argument['3'] = 4
...     return argument
...
>>> my_function()
{'3': 4}
>>> res = my_function()
>>> res['4'] = 'I am still alive!'
>>> print my_function()
{'3': 4, '4': 'I am still alive!'}
```

这就是始终使用非易变对象的原因，示例如下。

```
>>> def my_function(argument=None): # 较好的实践
...     if argument is None:
...         argument = {} # 每次都创建一个刷新的词典
...     if '1' in argument:
...         argument['1'] = 2
...         argument['3'] = 4
...     return argument
...
>>> my_function()
{'3': 4}
>>> res = my_function()
>>> res['4'] = 'I am still alive!'
>>> print my_function()
{'3': 4}
```

垃圾回收给开发人员带来了方便，可以避免跟踪对象并且手工销毁它们。但是这将引入其他问题，因为开发人员从不在内存中清除对象实例，因此如果不注意使用数据结构的方式，可能会变得不可控制。

通常消耗内存的是：

- 增长速度不可控制的缓存；

- 注册全局对象并且不跟踪其使用的对象工厂，如数据库连接创建器，每当查询被调用时将立刻执行；
- 没有正常结束的线程；
- 具有 `__del__` 方法并且涉及循环的对象也是主要的内存消耗者。Python 垃圾回收不会打断该循环，因为它不能确定对象是否应该被先删除。因此，将导致内存泄漏。在任何情况下使用这个方法都是坏主意。

2. 剖析内存

想知道垃圾回收控制了多少对象，以及它实际的大小是需要技巧的。例如，要知道指定的对象有多少字节，这就涉及到统计其所有的特性，处理交叉引用，然后累加所有值。如果再考虑对象间可能存在的互相引用，这将是相当困难的问题。`gc` 模块不提供高层级的函数，并且要求 Python 在调试模式下编译，以获得完整的信息。

编程人员往往只查询在指定操作执行前后他们应用程序的内存使用情况。但是这一度量只是个近似值，而且很大程度上依赖于系统级别上的内存管理方式。例如，在 Linux 下使用 `top` 命令或者在 Windows 下使用任务管理器都可能发现明显的内存问题，但是要跟踪错误的代码块，就需要在代码方面进行令人痛苦的工作。

幸运的是，有一些工具可用来创建内存快照，并计算载入对象的大小。但是请记住，Python 不会简单地释放内存，而是继续保存在内存中以防再次需要。

(1) Guppy-PE 入门

Guppy-PE (<http://guppy-pe.sourceforge.net>) 是一个框架，提供了被称为 Heap 的内存剖析程序及其他特性。

Guppy 可以通过 `easy_install` 来安装，命令如下。

```
$ sudo easy_install guppy
```

现在，在 `guppy` 命名空间下就可以使用 `hpy` 函数了。它将返回一个对象，该对象能够显示出内存的一个快照，如下所示。

```
>>> from guppy import hpy
>>> profiler = hpy()
>>> profiler.heap()
Partition of a set of 22723 objects. Total size = 1660748 bytes.
      Index Count %      Size % Cumulative % Kind (class / dict of
class)
0          9948    44 775680  47 775680  47   str
1          5162    23 214260  13 989940  60  tuple
2          1404     6  95472   6 1085412  65 types.CodeType
```



```

3         61      0 91484    6 1176896 71 dict of module
4        152      1 84064    5 1260960 76 dict of type
5       1333      6 79980    5 1340940 81 function
6        168      1 72620    4 1413560 85 type
7        119      1 68884    4 1482444 89 dict of class
8         76      0 51728    3 1534172 92 dict (no owner)
9        959      4 38360    2 1572532 95 __builtin__.wrapper_
descriptor
<43 more rows. Type e.g. '_more' to view.>

```

这个输出展示了当前内存的使用情况，并且按照大小排序，按照对象类型分组。该对象提供了许多特性，可以用来定义列表显示和使用方式，就像显示 CPU 时间信息的 `pstats` 那样。

可以使用 `iso` 方法来度量具体对象的大小，如下所示。

```

>>> import random
>>> def eat_memory():
...     memory = []
...     def _get_char():
...         return chr(random.randint(97, 122))
...     for i in range(100):
...         size = random.randint(20, 150)
...         data = [_get_char() for i in xrange(size)]
...         memory.append(''.join(data))
...     return '\n'.join(memory)
...
>>> profiler.iso(eat_memory())
Partition of a set of 1 object. Total size = 8840 bytes.
  Index  Count   %   Size      % Cumulative   % Kind
      0      1 100   8840     100         8840   100 str
>>> profiler.iso(eat_memory()+eat_memory())
Partition of a set of 1 object. Total size = 17564 bytes.
  Index  Count   %   Size      % Cumulative   % Kind
      0      1 100  17564     100        17564   100 str

```

`setrelheap` 方法用来重置这个剖析程序，这样可以使用 `heap` 捕获独立代码块的内存使用情况。但是这个初始化并不完美，因为上下文总是有多个载入的元素。这就是刚刚初始化的 `hpy` 实例不是 0 的原因，如下所示。

```
>>> g = hpy()
```

```
>>> g.setrelheap()
>>> g.heap().size
1120
>>> g.heap().size
1200
>>> g.heap().size
1144
```

heap 方法返回一个提供以下信息的 Usage 对象。

- size 以字节表示的内存总消耗量；
- get_rp() 告知对象代码在模块中位置的一个很好的遍历方法；
- 多种结果的排序方法，如 bytype。

(2) 使用 Heapy 跟踪内存使用情况

Heapy 不容易使用，需要一些练习。本小节将只介绍一些可以用来提供内存使用信息的简单函数。由此，跟踪代码必须通过 Heapy API 浏览对象来完成。



在 pkgcore 项目网站上有关于 Heapy 如何用来跟踪内存使用情况的一个很好的实例，参见 <http://www.pkgcore.org/trac/pkgcore/doc/dev-notes/heapy.rst>。这是使用该工具一个很好的出发点。

可以对 duration 装饰器（前面已经提出）做些修改，以使其提供函数使用的内存大小（以字节表示）。这个信息可以在词典中和持续时间、pystone 值一起读出。在 profiler.py 模块中的完整装饰器如下所示。

```
import time
import sys
from test import pystone
from guppy import hpy
benchtime, stones = pystone.pystones()
def secs_to_kstones(seconds):
    return (stones*seconds) / 1000
stats = {}
if sys.platform == 'win32':
    timer = time.clock
else:
    timer = time.time
```

```

def profile(name='stats', stats=stats):
    """Calculates a duration and a memory size."""
    def _profile(function):
        def __profile(*args, **kw):
            start_time = timer()
            profiler = hpy()
            profiler.setref()
            # 12是调用了 setref 之后的初始内存大小
            start = profiler.heap().size + 12
            try:
                return function(*args, **kw)
            finally:
                total = timer() - start_time
                kstones = secs_to_kstones(total)
                memory = profiler.heap().size - start
                stats[name] = {'time': total,
                              'stones': kstones,
                              'memory': profiler.heap().size}
        return __profile
    return _profile

```

变量 `start` 用来确认所计算的内存不包含调用 `setref` 时 `Heapy` 所消耗的那部分内存。

使用具有 `eat_memory` 的装饰器，除了以秒表示的持续时间和 `pystones` 值以外，还将提供函数消耗的内存数量，如下所示。

```

>>> import profiler
>>> import random
>>> eat_it = profiler.profile('you bad boy!')(eat_memory)
>>> please = eat_it()
>>> profiler.stats
{'you bad boy!': {'stones': 14.306935999128555, 'memory': 8680,
                  'time': 0.30902981758117676}}

```

当然，如果多次运行它，那么在涉及非易变的对象时将导致不同的内存大小。但是，它仍然是个良好的指示器。

`Heapy` 的另一种有趣的用法是检查函数是否释放了它所使用的内存，例如在它产生缓存或注册的元素时。这可以通过重复函数调用并观察使用的内存是否增长来实现。

以下就是一个用于该目的的简单函数。


```
>>> REPETITIONS = 100
>>> def memory_grow(function, *args, **kw):
...     """checks if a function makes the memory grows"""
...     profiler = hpy()
...     profiler.setref()
...     # 12 是调用了 setref 之后的初始内存大小
...     start = profiler.heap().size + 12
...     for i in range(REPEAT):
...         function(*args, **kw)
...     return profiler.heap().size - start
...
>>> def stable():
...     return "some"*10000
...
>>> d = []
>>> def greedy():
...     for i in range(100):
...         d.append('garbage data'*i)
...
>>> memory_grow(stable)
24
>>> memory_grow(greedy)
5196468
```

(3) C 代码内存泄漏

如果 Python 代码看上去很好，但是循环执行被隔离的函数时内存占用仍然在增加，那么导致内存泄漏的问题可能位于用 C 写的那部分代码。当诸如 Py_DECREF 之类的调用丢失时，就会发生这样的情况。

Python 核心代码相当健壮，并且进行了内存泄漏的测试。如果使用具有 C 扩展的包，那么应该首先关注这些扩展。

12.3.3 剖析网络使用情况

正如之前说过的，与诸如数据库、LDAP 服务器之类的第三程序通信的应用程序，可能会受到这些第三程序执行速度的影响。这可以在应用程序端用常规的代码剖析方法进行跟踪。但是如果第三方软件工作得很好，那么问题可能出在网络上。

问题有可能是错误配置的 hub，网络连接带宽较低，甚至是导致使电脑多次发送相同数据包通信冲突。

这里有几个要素是切入点。为了找到问题所在，可以调查以下 3 个方面。

- 使用以下工具查看网络流量。
 - ntop <http://www.ntop.org> (仅针对 Linux)
 - wireshark www.wireshark.org (以前叫 Ethereal)
- 使用 net-snmp 跟踪不健康或错误配置的设备。
- 使用 Pathrate 统计工具来估计两台计算机之间的带宽。

如果想要进一步查看网络性能问题，还可阅读 Richard Blum 所著的 *Network Performance Open Source Toolkit* 一书。这本书展示了针对大量使用网络的应用程序的调整策略，并且它是扫描复杂网络问题的一本很好的教程。

在编写使用 MySQL 的应用程序时，Jeremy Zawodny 所著的 *High Performance MySQL* 也是值得一读的好书。

12.4 小 结

本章介绍了以下内容。

- 优化的 3 条规则：首先使它能够正常工作；从用户的观点出来；保持代码易读。
- 基于编写带速度要求的使用场的优化策略。
- 剖析代码内存的方法，以及网络剖析的一些技巧。

知道如何查找问题后，下一章将提供解决问题的具体方案。

