教程 | Python代码优化指南：从环境设置到内存分析（一）

2017-07-15 机器之心

选自pythonfiles.wordpress

机器之心编译

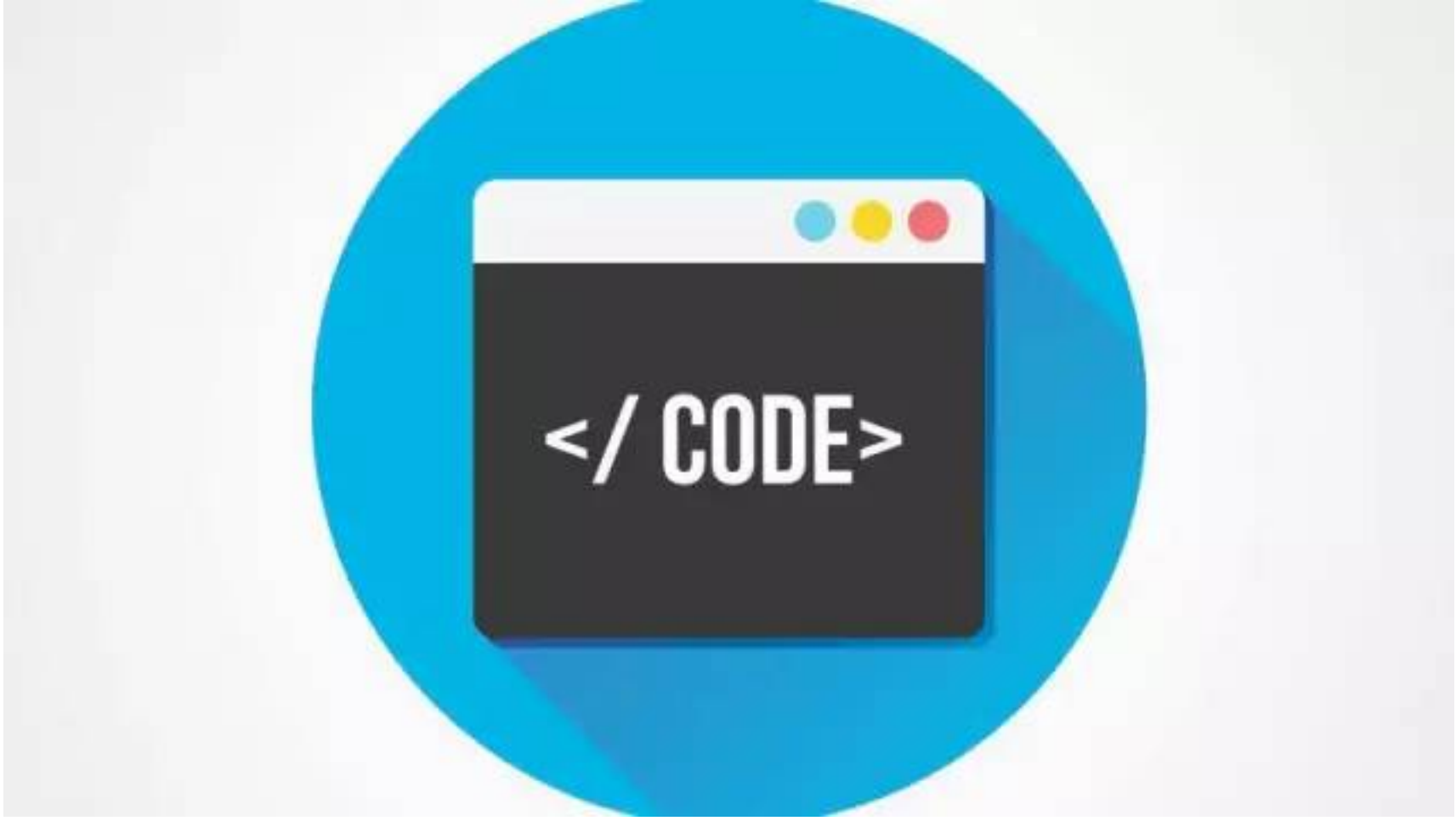
参与：Panda、蒋思源

> *近日，Python Files 博客发布了几篇主题为「Hunting Performance in Python Code」的系列文章，对提升 Python 代码的性能的方法进行了介绍。在其中的每一篇文章中，作者都会介绍几种可用于 Python 代码的工具和分析器，以及它们可以如何帮助你更好地在前端（Python 脚本）和/或后端（Python 解释器）中找到瓶颈。机器之心对这个系列文章进行了整理编辑，将其融合成了这一篇深度长文。本文的相关代码都已经发布在 GitHub 上。*

代码地址：https://github.com/apatrascu/hunting-python-performance

目录

本文是该教程的第一部分，主要从环境设置和内存分析两个方面探讨Python代码优化的路径。



一、环境设置

设置

在深入到基准测试和性能分析之前，首先我们需要一个合适的环境。这意味着我们需要为这项任务配置我们的机器和操作系统。
我的机器的规格如下：

- 处理器：Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
- 内存：32GB
- 操作系统：Ubuntu 16.04 LTS
- Kernel：4.4.0-75-generic

我们的目标是得到可复现的结果，因此要确保我们的数据不会受到其它后台进程、操作系统配置或任何其它硬件性能提升技术的影响。

让我们首先从配置用于性能分析的机器开始。

硬件功能

首先，禁用所有硬件性能功能，也就是说要禁用 Intel Turbo Boost 和 Hyper Threading from BIOS/UEFI。

正如其官方网页上说的那样，Turbo Boost 是「一种在处理器内核运行，并可以在低于功耗、电流和温度规格限制的情况下允许它们以高于额定频率的速度运行的技术。」此外，Hyper Threading 是「一种可以更高效地利用处理器资源的技术，能使每个内核都能多线程运行。」

这都是值得我们花钱购买的好东西。那为什么要在性能分析/基准测试中禁用它们呢？因为使用这些技术会让我们无法得到可靠的和可复现的结果。这会让运行过程发生变化。让我们看个小例子 primes.py，代码故意写得很糟糕。

```
import time
import statistics

def primes(n):
    if n==2:
    return [2]
elif n<2:
    return []
s=range(3,n+1,2)
mroot = n ** 0.5
half=(n+1)/2-1
i=0
m=3
while m <= mroot:
    if s[i]:
        j=(m*m-3)/2
        s[j]=0
        while j<half:
            s[j]=0
            j+=m
    i=i+1
    m=2*i+3
return [2]+[x for x in s if x]
```

```
def benchmark():
    results = []
    gstart = time.time()
    for _ in xrange(5):
        start = time.time()
        count = len(primes(1000000))
        end = time.time()
        results.append(end-start)
    gend = time.time()
    mean = statistics.mean(results)
    stdev = statistics.stdev(results)
    perc = (stdev * 100)/ mean
    print "Benchmark duration: %r seconds" % (gend-gstart)
    print "Mean duration: %r seconds" % mean
    print "Standard deviation: %r (%r %%)" % (stdev, perc)

benchmark()
```

这段代码可在 GitHub 上查看：https://github.com/apatrascu/hunting-python-performance/blob/master/01.primes.py。你需要运行以下命令安装一个依赖包：

```
pip install statistics
```

让我们在一个启用了 Turbo Boost 和 Hyper Threading 的系统中运行它：

```
python primes.py
Benchmark duration: 1.0644240379333496 seconds
Mean duration: 0.2128755569458008 seconds
Standard deviation: 0.032928838418120374 (15.468585914964498 %)
```

现在禁用该系统的睿频加速（Turbo Boost）和超线程（Hyper Threading），然后再次运行这段代码：

```
python primes.py
Benchmark duration: 1.2374498844146729 seconds
Mean duration: 0.12374367713928222 seconds
```

```
Standard deviation: 0.000684464852339824 (0.553131172568 %)
```

看看第一个案例的标准差为 15%。这是一个很大的值！假设我们的优化只能带来 6% 的加速，那我们怎么能将运行过程中的变化（run to run variation）和你的实现的差异区分开？相对而言，在第二个例子中，标准差减少到了大约 0.6%，我们的新优化方案效果清晰可见。

CPU 节能

禁用所有的 CPU 节能设置，并使用固定的 CPU 频率。这可以通过在 Linux 功率调节器（power governor）中将 intel_pstate 改成 acpi_cpufreq 而实现。

intel_pstate 驱动使用英特尔内核（Sandy Bridge 或更新）处理器的内部调节器实现了一个缩放驱动。acpi_cpufreq 使用了 ACPI Processor Performance States。
下面让我们先来检查一下：

```
$ cpupower frequency-info
analyzing CPU 0:
 driver: intel_pstate
 CPUs which run at the same hardware frequency: 0
 CPUs which need to have their frequency coordinated by software: 0
 maximum transition latency: 0.97 ms.
 hardware limits: 1.20 GHz - 3.60 GHz
 available cpufreq governors: performance, powersave
 current policy: frequency should be within 1.20 GHz and 3.60 GHz.
                 The governor "powersave" may decide which speed to use
                 within this range.
 current CPU frequency is 1.20 GHz.
 boost state support:
     Supported: yes
     Active: yes
```

可以看到这里所使用的调节器被设置成了节能模式，而 CPU 的频率范围在 1.20 GHz 到 3.60 GHz 之间。这个设置对日常应用来说是很不错的，但却会影响到基准测试的结果。

那么应该给调节器设置什么值呢？如果我们浏览一下文档，我们可以看到我们可以使用以下设置：

- 高性能（performance）：以最大频率运行 CPU
- 节能（powersave）：以最小频率运行 CPU
- 自定义（userspace）：按用户指定的频率运行 CPU
- 按需（ondemand）：根据当前负载动态调节频率。可能跳至最高频率，空闲时又会降低
- 保守（conservative）：根据当前负载动态调节频率。相比于按需模式，其频率调节更加渐进

我们要使用性能调节器（performance governor），并将频率设置成 CPU 支持的最大频率。如下所示：

```
$ cpupower frequency-info
analyzing CPU 0:
 driver: acpi-cpufreq
 CPUs which run at the same hardware frequency: 0
 CPUs which need to have their frequency coordinated by software: 0
 maximum transition latency: 10.0 us.
 hardware limits: 1.20 GHz - 2.30 GHz
 available frequency steps: 2.30 GHz, 2.20 GHz, 2.10 GHz, 2.00 GHz, 1.90 GHz, 1.80 GHz, 1.70 GH
z, 1.60 GHz, 1.50 GHz, 1.40 GHz, 1.30 GHz, 1.20 GHz
 available cpufreq governors: conservative, ondemand, userspace, powersave, performance
 current policy: frequency should be within 2.30 GHz and 2.30 GHz.
                 The governor "performance" may decide which speed to use
                 within this range.
 current CPU frequency is 2.30 GHz.
 cpufreq stats: 2.30 GHz:100.00%, 2.20 GHz:0.00%, 2.10 GHz:0.00%, 2.00 GHz:0.00%, 1.90 GHz:0.00
%, 1.80 GHz:0.00%, 1.70 GHz:0.00%, 1.60 GHz:0.00%, 1.50 GHz:0.00%, 1.40 GHz:0.00%, 1.30 GHz:0.0
0%, 1.20 GHz:0.00% (174)
 boost state support:
     Supported: no
     Active: no
```

现在你已经使用性能调节器将频率设置成了固定的 2.3 GHz。这是最大的可设置的值，没有睿频加速（Turbo Boost），它可以被用在 Xeon E5-2699 v3 上。

要完成设置，请使用管理员权限运行以下命令：

```
cpupower frequency-set -g performance
cpupower frequency-set --min 2300000 --max 2300000
```

如果你没有 cpupower，可使用以下命令安装：

```
sudo apt-get install linux-tools-common linux-header-`uname -r` -y
```

功率调节器对 CPU 的工作方式有很大的影响。该调节器的默认设置是自动调节频率以减少功耗。我们不想要这样的设置，所以从 GRUB 中禁用它。只需要编辑 /boot/grub/grub.cfg（但是如果你在 kernel 升级上很小心，那么这将会消失）或在 /etc/grub.d/40_custom 中创建一个新的 kernel 入口。我们的 boot 行中必须包含这个 flag： intel_pstate=disable，如下所示：

```
linux   /boot/vmlinuz-4.4.0-78-generic.efi.signed root=UUID=86097ec1-3fa4-4d00-97c7-3bf91787be8
3 ro intel_pstate=disable quiet splash $vt_handoff
```

ASLR（地址空间配置随机发生器）

这个设置是有争议的，参见 Victor Stinner 的博客： https://haypo.github.io/journey-to-stable-benchmark-average.html。当我首次建议在基准测试时禁用 ASLR 时，那是为了进一步提升对那时在 CPython 中存在的 Profile Guided Optimizations 的支持。

我为什么要说这个呢？因为在上面给出的特定硬件上，禁用 ASLR 可以将运行之间的标准差降低至 0.4%。另一方面，根据在我的个人计算机（Intel Core i7 4710MQ）上的测试，禁用 ASLR 会导致 Victor 所提到的同样的问题。在更小的 CPU（比如 Intel Atom）上的测试会带来甚至更大的运行间标准差。

因为这似乎并不是普遍适用的真理，而且很大程度上依赖于硬件/软件配置，所以对于这个设置，我在启用后测量一次，再禁用后测量一次，之后再进行比较。在我的机器上，我通过在 /etc/sysctl.conf. 中加入以下命令禁用了 ASLR。使用 sudo sysctl -p 进行应用。

```
kernel.randomize_va_space = 0
```

如果你想在运行时禁用它：

```
sudo bash -c 'echo 0 >| /proc/sys/kernel/randomize_va_space'
```

如果你想重新启用：

```
sudo bash -c 'echo 2 >| /proc/sys/kernel/randomize_va_space'
```

二、内存分析

在这一节，我将介绍一些有助于我们解决 Python 中（尤其是使用 PyPy 时）的内存消耗难题的工具。我们为什么要关心这个问题？为什么我们不仅仅就关心性能？这些问题的答案相当复杂，但我会总结出来。

PyPy 是一个可选的 Python 解释器，其相对于 CPython 有一些巨大的优势：速度（通过其 Just in Time 编译器）、兼容性（几乎可以替代 CPython）和并发性（使用 stackless 和 greenlets）。

PyPy 的一个缺点是因为其 JIT 和垃圾一样的回收站实现，它通常会使用比 CPython 更多的内存。但是在某些案例中，其的内存消耗会比 CPython 少。下面我们来看看你可以如何测量你的应用使用了多少内存。

诊断内存使用

memory_profiler 是一个可用来测量解释器运行一个负载时的内存用量的库。你可以通过 pip 安装它：

```
pip install memory_profiler
```

另外还要安装 psutil 依赖包：

```
pip install psutil
```

这个工具的优点是它会在一个 Python 脚本中一行行地显示内存消耗。这可以让我们找到脚本中可以被我们重写的位置。但这种分析有一个缺点。你的代码的运行速度比一般脚本慢 10 到 20 倍。

怎么使用它？你只需要在你需要测量的函数上直接加上 @profile() 即可。让我们看看实际怎么操作！我们将使用

之前用过的素材脚本作为模型，但做了一点修改，移除了统计部分。代码也可在 GitHub 查看：

https://github.com/apatrascu/hunting-python-performance/blob/master/02.primes-v1.py

```
from memory_profiler import profile



@profile(precision=6)
def primes(n):
    if n == 2:
        return [2]
    elif n < 2:
        return []
    s = range(3, n + 1, 2)
    mroot = n ** 0.5
    half = (n + 1) / 2 - 1
    i = 0
    m = 3
    while m <= mroot:
        if s[i]:
            j = (m * m - 3) / 2
            s[j] = 0
            while j < half:
                s[j] = 0
                j += m
        i = i + 1
        m = 2 * i + 3
    return [2] + [x for x in s if x]



len(primes(100000))
```

开始测量时，使用以下 PyPy 命令：

```
pypy -m memory_profiler 02.primes-v3.py
```

或者直接在脚本中导入 memory_profiler：

```
pypy -m memory_profiler 02.primes-v3.py
```

在执行完这行代码之后，我们可以看到 PyPy 得到这样的结果：

```
Line #    Mem usage    Increment   Line Contents
================================================
    54  35.312500 MiB   0.000000 MiB   @profile(precision=6)
    55                                 def primes(n):
    56  35.351562 MiB   0.039062 MiB       if n == 2:
    57                                     return [2]
    58  35.355469 MiB   0.003906 MiB       elif n < 2:
    59                                     return []
    60  35.355469 MiB   0.000000 MiB       s = []
    61  59.515625 MiB  24.160156 MiB       for i in range(3, n+1):
    62  59.515625 MiB   0.000000 MiB           if i % 2 != 0:
    63  59.515625 MiB   0.000000 MiB               s.append(i)
    64  59.546875 MiB   0.031250 MiB       mroot = n ** 0.5
    65  59.550781 MiB   0.003906 MiB       half = (n + 1) / 2 - 1
    66  59.550781 MiB   0.000000 MiB       i = 0
    67  59.550781 MiB   0.000000 MiB       m = 3
    68  59.554688 MiB   0.003906 MiB       while m <= mroot:
    69  59.554688 MiB   0.000000 MiB           if s[i]:
    70  59.554688 MiB   0.000000 MiB               j = (m * m - 3) / 2
    71  59.554688 MiB   0.000000 MiB               s[j] = 0
    72  59.554688 MiB   0.000000 MiB               while j < half:
    73  59.554688 MiB   0.000000 MiB                   s[j] = 0
    74  59.554688 MiB   0.000000 MiB                   j += m
    75  59.554688 MiB   0.000000 MiB           i = i + 1
```

```
    76   59.554688 MiB    0.000000 MiB           m = 2 * i + 3
    77   59.554688 MiB    0.000000 MiB       l = [2]
    78   59.679688 MiB    0.125000 MiB       for x in s:
    79   59.679688 MiB    0.000000 MiB           if x:
    80   59.679688 MiB    0.000000 MiB               l.append(x)
    81   59.683594 MiB    0.003906 MiB       return l
```

我们可以看到这个脚本使用了 24.371094 MiB 的 RAM。让我们简单分析一下。我们看到其中大多数都用在了数值数组的构建中。它排除了偶数数值，保留了所有其它数值。

我们可以通过调用 range 函数而对其进行一点改进，其使用一个增量参数。在这个案例中，该脚本看起来像是这样：

```
from memory_profiler import profile

@profile(precision=6)

def primes(n):

    if n == 2:

        return [2]

    elif n < 2:

        return []

    s = range(3, n + 1, 2)

    mroot = n ** 0.5

    half = (n + 1) / 2 - 1

    i = 0

    m = 3

    while m <= mroot:

        if s[i]:

            j = (m * m - 3) / 2

            s[j] = 0

            while j < half:
```

```
                s[j] = 0
                j += m
        i = i + 1
        m = 2 * i + 3
    l = [2]
    for x in s:
        if x:
            l.append(x)
    return l


len(primes(100000))
```

如果我们再次测量，我们可以得到以下结果：

```
Line #    Mem usage    Increment   Line Contents
================================================
    27  35.343750 MiB   0.000000 MiB   @profile(precision=6)
    28                                 def primes(n):
    29  35.382812 MiB   0.039062 MiB       if n == 2:
    30                                     return [2]
    31  35.382812 MiB   0.000000 MiB       elif n < 2:
    32                                     return []
    33  35.386719 MiB   0.003906 MiB       s = range(3, n + 1, 2)
    34  35.417969 MiB   0.031250 MiB       mroot = n ** 0.5
    35  35.417969 MiB   0.000000 MiB       half = (n + 1) / 2 - 1
    36  35.417969 MiB   0.000000 MiB       i = 0
    37  35.421875 MiB   0.003906 MiB       m = 3
    38  58.019531 MiB  22.597656 MiB       while m <= mroot:
    39  58.019531 MiB   0.000000 MiB           if s[i]:
    40  58.019531 MiB   0.000000 MiB               j = (m * m - 3) / 2
```

```
    41  58.019531 MiB   0.000000 MiB                s[j] = 0
    42  58.019531 MiB   0.000000 MiB                while j < half:
    43  58.019531 MiB   0.000000 MiB                    s[j] = 0
    44  58.019531 MiB   0.000000 MiB                    j += m
    45  58.019531 MiB   0.000000 MiB            i = i + 1
    46  58.019531 MiB   0.000000 MiB            m = 2 * i + 3
    47  58.019531 MiB   0.000000 MiB        l = [2]
    48  58.089844 MiB   0.070312 MiB        for x in s:
    49  58.089844 MiB   0.000000 MiB            if x:
    50  58.089844 MiB   0.000000 MiB                l.append(x)
    51  58.093750 MiB   0.003906 MiB        return l
```

很好，现在我们的内存消耗下降到了 22.75 MiB。使用列表解析（list comprehension），我们还可以将消耗再降低一点。

```
from memory_profiler import profile

@profile(precision=6)
def primes(n):
    if n == 2:
        return [2]
    elif n < 2:
        return []
    s = range(3, n + 1, 2)
    mroot = n ** 0.5
    half = (n + 1) / 2 - 1
    i = 0
    m = 3
    while m <= mroot:
        if s[i]:
```

```
            j = (m * m - 3) / 2
            s[j] = 0
            while j < half:
                s[j] = 0
                j += m
        i = i + 1
        m = 2 * i + 3
    return [2] + [x for x in s if x]


len(primes(100000))
```

再次测量:

```
Line #    Mem usage    Increment   Line Contents
================================================
 4  35.425781 MiB   0.000000 MiB   @profile(precision=6)
 5                                 def primes(n):
 6  35.464844 MiB   0.039062 MiB       if n == 2:
 7                                     return [2]
 8  35.464844 MiB   0.000000 MiB       elif n < 2:
 9                                     return []
10  35.464844 MiB   0.000000 MiB       s = range(3, n + 1, 2)
11  35.500000 MiB   0.035156 MiB       mroot = n ** 0.5
12  35.500000 MiB   0.000000 MiB       half = (n + 1) / 2 - 1
13  35.500000 MiB   0.000000 MiB       i = 0
14  35.500000 MiB   0.000000 MiB       m = 3
15  57.683594 MiB  22.183594 MiB       while m <= mroot:
16  57.683594 MiB   0.000000 MiB           if s[i]:
17  57.683594 MiB   0.000000 MiB               j = (m * m - 3) / 2
18  57.683594 MiB   0.000000 MiB               s[j] = 0
```

```
    19   57.683594 MiB    0.000000 MiB                while j < half:
    20   57.683594 MiB    0.000000 MiB                    s[j] = 0
    21   57.683594 MiB    0.000000 MiB                    j += m
    22   57.683594 MiB    0.000000 MiB            i = i + 1
    23   57.683594 MiB    0.000000 MiB            m = 2 * i + 3
    24   57.847656 MiB    0.164062 MiB        return [2] + [x for x in s if x]
```

我们最后的脚本仅消耗 22.421875 MiB。相比于第一个版本，差不多下降了 10%。

原文链接：https://pythonfiles.wordpress.com/

阅读原文