Python性能分析

stevegao(高家华)

课程介绍

1. Python性能分析
2. Python性能优化的技巧
3. Python性能优化实践

背景知识

适合

- 写过一点python
- 思考python的性能问题
- 有python性能优化需求
- 想写出性能更好的python代码

不适合

- 不是一门python编程入门，从来没写过python的人不适合
- python初级偏中级一点的课程，python老鸟不适合
- 对比课程的章节目，如果对涉及到的技术点都了解，也不适合这个课程

目录介绍

概述

- 什么是性能分析
- 性能分析的一般过程
    - 程序运行的速度如何
    - 时间瓶颈在哪里/内存瓶颈在哪里
    - 性能瓶颈的改进方案

正文：

1. 运行时间分析

- 运行时间复杂度

名称	复杂度	算法举例
常数时间	O(1)	判断一个数是基数还是偶数
对数时间	O(logn)	二分查找
线性时间	O(n)	查找无序列表的最小元素
线性对数时间	O(n logn)	快速排序(平均时间)
平方时间	O(n**2)	冒泡排序

1.1 Shell 命令time

linux shell time命令常用于测量一个命令的运行时间，注意不是用来显示和修改系统时间的，不仅仅用于python。

windows下使用：

1. 虚拟机
2. **Cygwin**这样的模拟环境
3. git bash

```
stevegao@stevegao-PC4 ~
$ time ls
a.exe  helloworld.c

real    0m0.111s
user    0m0.000s
sys     0m0.015s
```

```
#test_shell_time0.py
rs = 0
for i in xrange(100*100):
    rs += i
print rs
```

```
stevegao_tech@stevegao-NB1 /cygdrive/f/Codes/python性能优化/demos
$ time python test_shell_time0.py
49995000

real    0m0.153s
user    0m0.015s
sys     0m0.093s
```

```python
#test_shell_time1.py
from time import *
sleep(2)
```

```
stevegao_tech@stevegao-NB1 /cygdrive/f/Codes/python性能优化/demos
$ time python test_shell_time1.py

real    0m2.178s
user    0m0.031s
sys     0m0.077s
```

- 一点结论
  - real != user + sys
  - real 和 user + sys的值越接近，证明程序越重计算，反之说明程序更重IO

1.2 Python自带模块time

- time函数的功能

```python
#time_demo.py
import time
print time.time()
print time.asctime( time.localtime(time.time()) )
print time.asctime( time.localtime(0) )
```

time. time() 返回当前时间的时间戳（1970纪元后经过的浮点秒数）

```
1493454752.49
Sat Apr 29 16:32:32 2017
Thu Jan 01 08:00:00 1970
```

- time.time()的简单应用

```python
import time
t0 = time.time()
doSomething()
t1 = time.time()
print t1 - t0
```

- time.time()的封装使用

  _enter_和_exit_配合with关键字使用

```
#timer.py
import time
class Timer(object):
        def __init__(self, verbose=False):
                self.verbose = verbose

        def __enter__(self):
                self.start = time.time()
                return self

        def __exit__(self, *args):
                self.end = time.time()
                self.secs = self.end - self.start
                self.msecs = self.secs * 1000 # millisecs
                if self.verbose:
                        print 'elapsed time: %f ms' % self.msecs
```

```
#python_time_test0.py
from timer import Timer
from redis import Redis
rdb = Redis()

with Timer() as t:
    rdb.lpush("foo", "bar")
print "=> elasped lpush: %s s" % t.secs

with Timer() as t:
    print rdb.lpop("foo")
print "=> elasped lpop: %s s" % t.secs
```

> 结果演示

windows下推荐用time.clock代替time.time，更精确

1.3 python模块timeit

测量一段代码的运行时间，在python内可以直接使用timeit。

```
import timeit
timeit.timeit("x = range(100)")
```

```
0.6274833867336724
```

大家觉得上边这行代码执行0.6s时间消耗高不高

```
default_number = 1000000
```

上边讲的三种方法都比较简单，适合做粗略统计，下面讲两个性能分析器

1.4 Python默认性能分析器cProfile

cProfile自Python 2.5以来就是标准版Python解释器默认的性能分析器，测量CPU运行时间，统计函数调用次数，不关心内存相关信息。尽管如此，它是性能优化过程中一个近似于标准化的起点，绝大多数时候这个都能为我们的分析工作提供有力支持。

- 在py代码中使用

```
#cprofiler_inpy.py
import cProfile
import re
def test():
    for i in xrange(10**6):
        re.compile("foo|bar")
cProfile.run('test()')
```

> 结果太长，实际演示

```
ncalls：表示函数调用的次数；
tottime：表示指定函数的总的运行时间，除掉函数中调用子函数的运行时间；
percall：（第一个percall）等于 tottime/ncalls；
cumtime：表示该函数及其所有子函数的调用运行的时间，即函数开始调用到返回的时间；
percall：（第二个percall）即函数运行一次的平均时间，等于 cumtime/ncalls；
filename:lineno(function)：每个函数调用的具体信息；
```

tips：原生（**primitive**）调用，表明这些调用不涉及递归

- 在命令行使用

  使用的Python脚本就是刚才在讲time模块的时候Redis的例子

```
# 直接把分析结果打印到控制台
python -m cProfile python_time_test0.py
# 把分析结果保存到文件中
python -m cProfile -o result.prf python_time_test0.py
# 增加排序方式
python -m cProfile -s tottime python_time_test0.py
```

```
         6189 function calls (6119 primitive calls) in 1.105 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        2    1.025    0.513    1.025    0.513 {method 'connect' of '_socket.socket' objects}
        1    0.012    0.012    0.012    0.012 socket.py:45(<module>)
        1    0.010    0.010    0.010    0.010 {_socket.getaddrinfo}
        1    0.007    0.007    0.026    0.026 urllib.py:23(<module>)
        1    0.006    0.006    0.032    0.032 _compat.py:1(<module>)
        1    0.006    0.006    0.064    0.064 client.py:1(<module>)
        1    0.005    0.005    0.009    0.009 connection.py:1(<module>)
        1    0.004    0.004    0.004    0.004 __init__.py:4(<module>)
        1    0.003    0.003    0.003    0.003 collections.py:1(<module>)
        4    0.002    0.001    0.003    0.001 collections.py:293(namedtuple)
        1    0.002    0.002    0.010    0.010 uuid.py:45(<module>)
        1    0.002    0.002    0.002    0.002 utils.py:1(<module>)
        1    0.002    0.002    1.105    1.105 python_time_test0.py:1(<module>)
```

1.5 第三方性能分析器line_profiler

核心就在于line这个单词，这个性能分析器和cProfile不同。它可以帮助你一行一行地分析函数性能。cProfile主要关注函数的性能，如果你的程序性能瓶颈出现在某一行python代码中，line_profiler显得非常恰当。

- 安装
  使用pip安装，linux直接pip install line_profiler

  windows下pip安装，可能的失败情况：

```
error: Microsoft Visual C++ 9.0 is required
```

Windows下，依赖VS编译，可以调整环境变量

```
VS90COMNTOOLS
#系统中的VS安装路径
C:\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\Tools
```

或者访问 http://aka.ms/vcpython27 去下载编译支持包

- 使用：line_profiler的作者建议使用其中的kernprof工具，下边的介绍也是基于kernprof的

(1) 修改源代码, 待测试函数上增加@profile

```
#line_profiler_test.py
@profile
def line_profiler():
    rs = 0
    for i in range(100*100):
        rs += i
    print rs
if __name__ == "__main__":
    line_profiler()
```

(2) 命令行调用

```
python  kernprof.py   -l  -v  line_profiler_test.py

#-l 选项通知kernprof注入@profile装饰器

#-v 选项通知kernprof在脚本执行完毕的时候显示计时信息
```

- 效果

```
Total time: 0.0112683 s
File: line_profiler_test.py
Function: line_profiler at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           @profile
     2                                           def line_profiler():
     3         1            4      4.0      0.0      rs = 0
     4     10001        13151      1.3     49.9      for i in range(100*100):
     5     10000        12401      1.2     47.1          rs += i
     6         1          795    795.0      3.0      print rs
```

2. 内存分析

2.1 宏观分析memory_profiler

现在机器学习和深度学习很火热，很多学习任务比较吃内存，memory_profiler这种场景下可以起到一定作用

- 安装

```
pip install memory_profiler
pip install psutil
pip install matplotlib# 如果出现安装失败的情况，先更新pip: pip install --upgrade pip
```

- 命令行使用

```
#memory_profiler_test0.py
@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a
if __name__ == '__main__':
    my_func()
```

```
python -m memory_profiler memory_profiler_test.py
```

- 效果

```
Line #    Mem usage    Increment   Line Contents
================================================
     1   34.758 MiB    0.000 MiB   @profile
     2                             def my_func():
     3   42.406 MiB    7.648 MiB       a = [1] * (10 ** 6)
     4  195.293 MiB  152.887 MiB       b = [2] * (2 * 10 ** 7)
     5   42.406 MiB -152.887 MiB       del b
     6   42.406 MiB    0.000 MiB       return a
```

- 在Python脚本中使用

```
#memory_profiler_test1.py
from memory_profiler import profile
@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a
if __name__ == '__main__':
    my_func()
```

- 效果

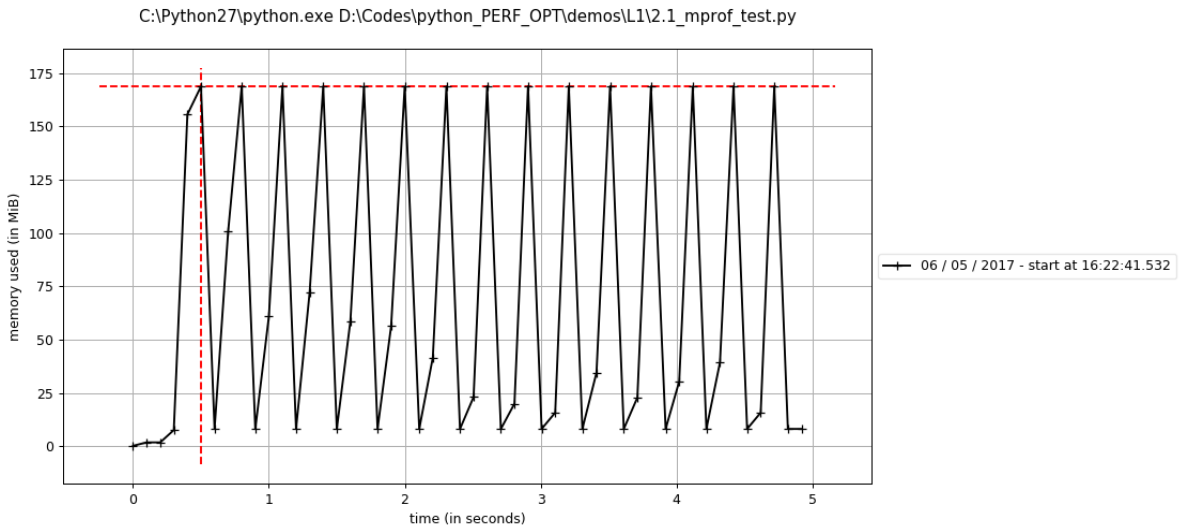
  直接运行和命令行增加 -m memory_profiler方式的效果相同。

- mprof 基于时间的内存测量

```
python mprof run xxx.py
python mprof plot
```

```
#mprof_test.py
import time
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a
if __name__ == '__main__':
    for i in xrange(15):
        my_func()
        time.sleep(0.1)
```

- 效果



2.2 微观分析objgraph

- 安装

```
pip install objgraph
```

- python可能出现的内存问题：
  (1)所用到的用 C 语言开发的底层模块中出现了内存问题

  (2)代码中用到了全局的 list、 dict 或其它容器，不停的往这些容器中插入对象，而忘记了在使用完之后进行删除回收

```
#memLeak.py
import pdb
class MyBigFatObject(object):
    def __init__(self):
        self.data =  [2] * (2 * 10 ** 7)

def computate_something(_cache={}):
    _cache["default"] = dict(foo=MyBigFatObject(),
                      bar=MyBigFatObject())
    x = MyBigFatObject()

def test():
    pdb.set_trace()
    print "b"
    computate_something()
    print "f"
if __name__ == '__main__':
    test()
```

- 借助pdb调试，常用的pdb命令

  p(print) 查看一个变量值
  n(next) 下一步
  s(step) 单步,可进入函数
  c(continue)继续前进
  l(list)看源代码

```
#显示距离上次执行此命令之间生成的对象
objgraph.show_growth()
```

3. 可视化工具

3.1 log分析Runsnakerun

- 安装
  依赖wxpython
  可以直接在网页上下载安装包https://www.wxpython.org/（注意区分64位和32位要和安装的Python版本对应）

然后安装runsnakerun

```
pip install runsnakerun
```

安装完之后会生成一个runsnake.py

- 使用
  python runsnake.py result.prf

3.2 可视化工具pycallgraph

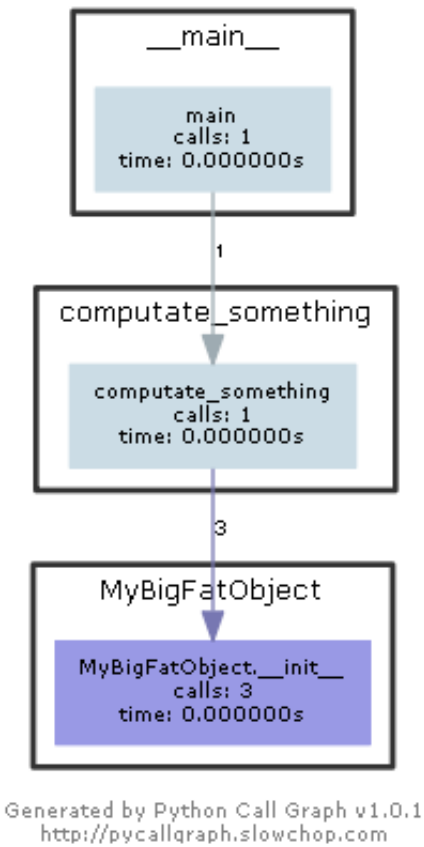
依赖graphviz，网上下载安装包http://www.graphviz.org/，安装完成后把安装路径的bin文件夹路径添加到环境变量中。

然后pip安装pycallgraph，pip install pycallgraph

- python脚本内使用

```
#code2pycallGraph.py
from pycallgraph import PyCallGraph
from pycallgraph.output import GraphvizOutput

with PyCallGraph(output=GraphvizOutput()):
    code_to_profile()
```



- 命令行使用

```
python pycallgraph graphviz -- xxx.py
```