

内容

407 116

学习 > Linux

Python 性能优化工具

定位程序性能瓶颈

Python 代码性能优化技巧

总结



张颖

2012 年 7 月 20 日发布

评论

Python 代码优化常见技巧

代码优化能够让程序运行更快，它是在不改变程序运行结果的情况下使得程序的运行效率更高，结构、优化、扩展以及文档相关的事情通常需要消耗 80% 的工作量。优化通常包含两方面的内容：提高运行效率。

改进算法，选择合适的数据结构

一个好的算法能够对性能起到关键作用，因此性能改进的首要点是对算法的改进。在算法的时间复杂度上，改进通常包含两方面的内容：提高运行效率。

$O(1) \rightarrow O(\lg n) \rightarrow O(n \lg n) \rightarrow O(n^2) \rightarrow O(n^3) \rightarrow O(n^k) \rightarrow O(k^n) \rightarrow O(n!)$

因此如果能够在时间复杂度上对算法进行一定的改进，对性能的提高不言而喻。但对具体算法的改进，读者可以自行参考这方面资料。下面的内容将集中讨论数据结构的选择。

- 字典 (dictionary) 与列表 (list)

Python 字典中使用了 hash table，因此查找操作的复杂度为 $O(1)$ ，而 list 实际是个数组，在 list 中查找操作的复杂度为 $O(n)$ ，因此对成员的查找访问等操作字典要比 list 更快。

清单 1. 代码 dict.py

```
1 | from time import time
```

```

2 | t = time()
3 | list = ['a','b','is','python','jason','hello','hill','with','phone','test',
4 | 'dfdf','apple','pddf','ind','basic','none','baecr','var','bana','dd','wrd']
5 | #list = dict.fromkeys(list,True)
6 | print list
7 | filter = []
8 | for i in range (1000000):
9 |     for find in ['is','hat','new','list','old','.']:
10 |         if find not in list:
11 |             filter.append(find)
12 | print "total run time:"
13 | print time()-t

```

Python 性能优化工具

上述代码运行大概需要 16.09seconds。如果去掉行 `#list = dict.fromkeys(list,True)` 的注释，将大约为 8.375 seconds，效率大概提高了一半。因此在需要多数据成员进行频繁的查找或者访问时，set 是一个较好的选择。

相关主题

- 集合 (set) 与列表 (list)

评论

set 的 union, intersection, difference 操作要比 list 的迭代要快。因此如果涉及到求 list 交集 set 来操作。

清单 2. 求 list 的交集：

```

1 | from time import time
2 | t = time()
3 | lista=[1,2,3,4,5,6,7,8,9,13,34,53,42,44]
4 | listb=[2,4,6,9,23]
5 | intersection=[]
6 | for i in range (1000000):
7 |     for a in lista:
8 |         for b in listb:
9 |             if a == b:
10 |                 intersection.append(a)
11 |
12 |
13 | print "total run time:"
14 | print time()-t

```

上述程序的运行时间大概为：

```

1 | total run time:
2 | 38.4070000648

```

清单 3. 使用 set 求交集

```

1 | from time import time
2 | t = time()
3 | lista=[1,2,3,4,5,6,7,8,9,13,34,53,42,44]
4 | listb=[2,4,6,9,23]
5 | intersection=[]
6 | for i in range (1000000):

```

```
7 |         list(set(lista)&set(listb))
8 |     print "total run time:"
9 |     print time()-t
```

内容
改为 set 后程序的运行时间缩减为 8.75，提高了 4 倍多，运行时间大大缩短。读者可以自行使

概览
表 1. set 常见用法

Python 代码优化常见技巧

语法	操作	说明
Python 性能优化工具 set(list1) set(list2)	union	包含 list1 和 list2
set(list1) & set(list2)	intersection	包含 list1 和 list2 的公共元素
评论 set(list1) - set(list2)	difference	在 list1 中出现但不在 list2 中的元素

对循环的优化

对循环的优化所遵循的原则是尽量减少循环过程中的计算量，有多重循环的尽量将内层的计算移到循环外部，以减少重复计算。循环优化后所带来的性能的提高。程序清单 4 中，如果不进行循环优化，其大概的运行时间约：

清单 4. 为进行循环优化前

```
1 | from time import time
2 | t = time()
3 | lista = [1,2,3,4,5,6,7,8,9,10]
4 | listb =[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,0.01]
5 | for i in range (1000000):
6 |     for a in range(len(lista)):
7 |         for b in range(len(listb)):
8 |             x=lista[a]+listb[b]
9 |     print "total run time:"
10 | print time()-t
```

现在进行如下优化，将长度计算提到循环外，range 用 xrange 代替，同时将第三层的计算 lista

清单 5. 循环优化后

```
1 | from time import time
2 | t = time()
3 | lista = [1,2,3,4,5,6,7,8,9,10]
4 | listb =[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,0.01]
5 | len1=len(lista)
6 | len2=len(listb)
7 | for i in xrange (1000000):
```

```

8         for a in xrange(len1):
9             temp=lista[a]
10            for b in xrange(len2):
11                x=temp+listb[b]
12    print "total run time:"
13    print time()-t

```

概览

上述优化后的程序其运行时间缩短为 102.171999931。在清单 4 中 lista[a] 被计算的次数为 1000000*10，计算次数大幅度缩短，因此性能有所提升。

定位程序性能瓶颈

充分利用 Lazy if-evaluation 的特性

总结

python 中条件表达式是 lazy evaluation 的，也就是说如果存在条件表达式 if x and y，在 x 为 false 时，y 不会被计算，因此可以利用该特性在一定程度上提高程序效率。

清单 6. 利用 Lazy if-evaluation 的特性

```

1  from time import time
2  t = time()
3  abbreviations = ['cf.', 'e.g.', 'ex.', 'etc.', 'fig.', 'i.e.', 'Mr.', 'vs.']
4  for i in range (1000000):
5      for w in ('Mr.', 'Hat', 'is', 'chasing', 'the', 'black', 'cat', '.'):
6          if w in abbreviations:
7              #if w[-1] == '.' and w in abbreviations:
8                  pass
9  print "total run time:"
10 print time()-t

```

在未进行优化之前程序的运行时间大概为 8.84，如果使用注释行代替第一个 if，运行的时间大概为 0.016s。

字符串的优化

python 中的字符串对象是不可改变的，因此对任何字符串的操作如拼接，修改等都将产生一个字符串，因此这种持续的 copy 会在一定程度上影响 python 的性能。对字符串的优化也是改善性能较佳的情况下。字符串的优化主要集中在以下几个方面：

1. 在字符串连接的使用尽量使用 join() 而不是 +：在代码清单 7 中使用 + 进行字符串连接大概需要 0.016s。因此在字符串的操作上 join 比 + 要快，因此要尽量使用 join 而不是 +。

清单 7. 使用 join 而不是 + 连接字符串

```

1  from time import time
2
3  t = time()
4  s = ""
5  list = ['a', 'b', 'b', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n']

```

```

6 | for i in range (10000):
7 |     for substr in list:
8 |         s+= substr
9 | print "total run time:"
10| print time()-t

```

内容

概览 同时要避免：

```

Pyth 1 | s = ""
      2 | for x in list:
      3 |     s += func(x)
定位

```

Python 性能优化工具

总结 而是要使用：

```

相关 1 | slist = [func(elt) for elt in somelist]
      2 | s = "".join(slist)

```

评论

1. 当对字符串可以使用正则表达式或者内置函数来处理的时候，选择内置函数。如 `str.isalpha`、`str.startswith(('x', 'yz'))`、`str.endswith(('x', 'yz'))`

1. 对字符进行格式化比直接串联读取要快，因此要使用

```

1 | out = "<html>%s%s%s%s</html>" % (head, prologue, query, tail)

```

而避免

```

1 | out = "<html>" + head + prologue + query + tail + "</html>"

```

使用列表解析（list comprehension）和生成器表达式（generator expression）

列表解析要比在循环中重新构建一个新的 list 更为高效，因此我们可以利用这一特性来提高运行效率。

```

1 | from time import time
2 | t = time()
3 | list = ['a', 'b', 'is', 'python', 'jason', 'hello', 'hill', 'with', 'phone', 'test',
4 | 'dfdf', 'apple', 'pddf', 'ind', 'basic', 'none', 'baecr', 'var', 'bana', 'dd', 'wrd']
5 | total=[]
6 | for i in range (1000000):
7 |     for w in list:
8 |         total.append(w)
9 | print "total run time:"
10| print time()-t

```

使用列表解析：

```
1 for i in range (1000000):
2     a = [w for w in list]
```

概述代码直接运行大概需要 17s，而改为使用列表解析后，运行时间缩短为 9.29s。将近提高中引入的新内容，语法和列表解析类似，但是在大数据量处理时，生成器表达式的优势较为明显。Python 代码优化常见技巧，回一个生成器，因此效率较高。在上述例子上中代码 `a = [w for w in list]` 修改为 `a = (w for w in list)` 约为 2.98s，约 1/5 的性能瓶颈。

Python 性能优化工具

其他优化技巧

相关主题

1. 如果需要交换两个变量的值使用 `a,b=b,a` 而不是借助中间变量 `t=a;a=b;b=t`;

评论

```
1 >>> from timeit import Timer
2 >>> Timer("t=a;a=b;b=t", "a=1;b=2").timeit()
3 0.25154118749729365
4 >>> Timer("a,b=b,a", "a=1;b=2").timeit()
5 0.17156677734181258
6 >>>
```

1. 在循环的时候使用 `xrange` 而不是 `range`；使用 `xrange` 可以节省大量的系统内存，因为 `xrange` 只返回一个整数元素。而 `range()` 将直接返回完整的元素列表，用于循环时会有不必要的开销。在 `python3` 中 `range` 提供一个可以遍历任意长度的范围的 iterator。
1. 使用局部变量，避免 "global" 关键字。python 访问局部变量会比全局变量要快得多，因此
1. `if done is not None` 比语句 `if done != None` 更快，读者可以自行验证；
1. 在耗时较多的循环中，可以把函数的调用改为内联的方式；
1. 使用级联比较 `"x < y < z"` 而不是 `"x < y and y < z"`；
1. `while 1` 要比 `while True` 更快（当然后者的可读性更好）；
1. `build in` 函数通常较快，`add(a,b)` 要优于 `a+b`。

定位程序性能瓶颈

对代码优化的前提是需要了解性能瓶颈在什么地方，程序运行的主要时间是消耗在哪里，对于定位，python 内置了丰富的性能分析工具，如 `profile`, `cProfile` 与 `hotshot` 等。其中 `Profiler` 描述程序运行时候的性能，并提供各种统计帮助用户定位程序的性能瓶颈。Python 标准模块提及 `hotshot`。

profile 的使用非常简单，只需要在使用之前进行 import 即可。具体实例如下：

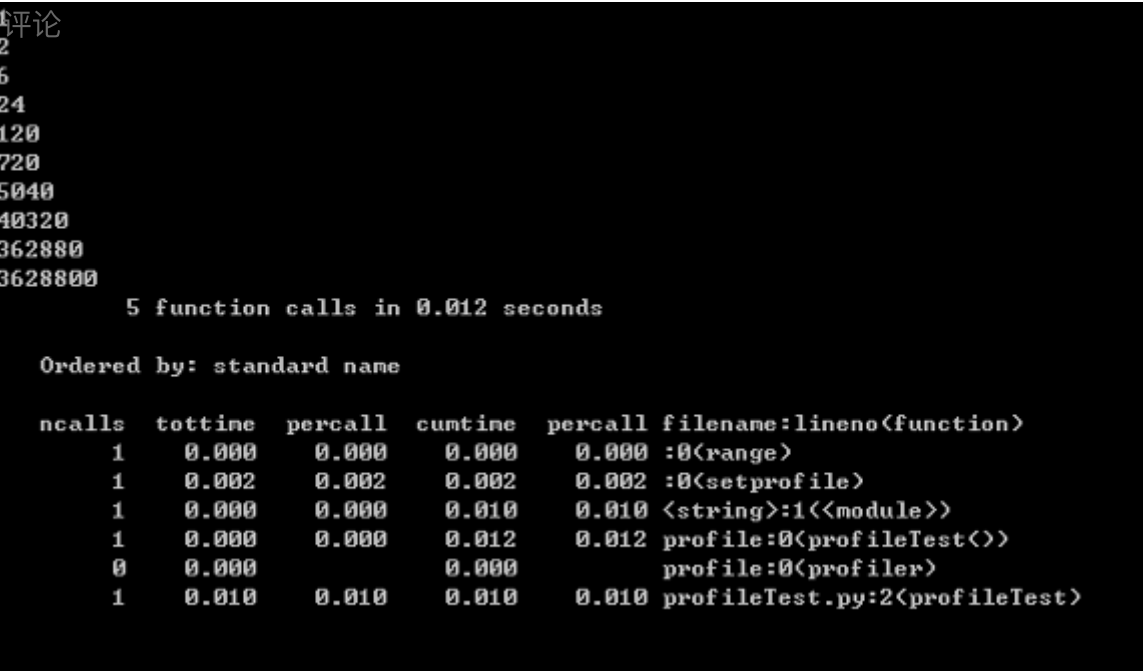
清单 8. 使用 profile 进行性能分析

```
1 import profile
2 def profileTest():
3     Total =1;
4     for i in range(10):
5         Total=Total*(i+1)
6         print Total
7     return Total
8 if __name__ == "__main__":
9     profile.run("profileTest()")
```

Python 性能优化工具

程序的运行结果如下：

图 8.1 性能分析结果



对于 profile 的剖析数据，如果以二进制文件的时候保存结果的时候，可以通过 pstats 模块进行的报表输出，是文本界面下一个较为实用的工具。使用非常简单：

```
1 import pstats
2 p = pstats.Stats('testprof')
3 p.sort_stats("name").print_stats()
```

其中 `sort_stats()` 方法能够对剖分数据进行排序，可以接受多个排序字段，如 `sort_stats('name')` 按照函数名进行排序，然后再按照文件名进行排序。常见的排序字段有 `calls` (被调用的次数)，`time` (函数内部总时间) 等。此外 pstats 也提供了命令行交互工具，执行 `python -m pstats` 后可以通过 `help`

Python 性能优化工具

对于大型应用程序，如果能够将性能分析的结果以图形的方式呈现，将会非常实用和直观，常见的工具有 `Visualpytune`，`KCacheGrind` 等，读者可以自行查阅相关官网，本文不做详细讨论。

相关主题

Python 性能优化工具

Python 性能优化除了改进算法，选用合适的数据结构之外，还有几种关键的技术，比如将关键块，或者选用在性能上更为优化的解释器等，这些在本文中统称为优化工具。python 有很多自 `Cython`，`Pyrex` 等，这些优化工具各有千秋，本节选择几种进行介绍。

Psyco

psyco 是一个 just-in-time 的编译器，它能够在不改变源代码的情况下提高一定的性能，Psyco 其操作分成三个不同的级别，有"运行时"、"编译时"和"虚拟时"变量。并根据需要提高和降低 Python 解释器处理的原始字节码和对象结构。一旦 Psyco 将操作编译成机器码，那么编译时变问的内存位置中表示。同时 python 能高速缓存已编译的机器码以备今后重用，这样能节省一点本身运行所占内存较大。目前 psyco 已经不在 python2.7 中支持，而且不再提供维护和更新了 <http://psyco.sourceforge.net/>

PyPy

PyPy 表示 "用 Python 实现的 Python"，但实际上它是使用一个称为 RPython 的 Python 子集实现 C，.NET，Java 等语言和平台的代码。PyPy 集成了一种即时 (JIT) 编译器。和许多编译器，解的词法分析和语法树。因为它用 Python 语言写的，所以它直接利用 Python 语言的 Code Object 节码的表示，也就是说，PyPy 直接分析 Python 代码所对应的字节码，这些字节码即不是以二进制保存在文件中，而在 Python 运行环境中。目前版本是 1.8. 支持不同的平台安装，windows 上 <https://bitbucket.org/pypy/pypy/downloads/pypy-1.8-win32.zip>，然后解压到相关的目录，并

path 中即可。在命令行运行 pypy，如果出现如下错误："没有找到 MSVCR100.dll, 因此这个程序可能会修复此问题"，则还需要在微软的官网上下载 VS 2010 runtime libraries 解决该问题。

<http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=5555>

内容

安装成功后在命令行里运行 pypy，输出结果如下：

概览

```
1 C:\Documents and Settings\Administrator>pypy
2 Python 2.7.2 (0e28b379d8b3, Feb 09 2012, 18:31:47)
3 [PyPy 1.8.0 with MSC v.1500 32 bit] on win32
4 Type "help", "copyright", "credits" or "license" for more information.
5 And now for something completely different: ``PyPy is vast, and contains
6 multitudes''
7 >>>>
```

总结

以清单 5 的循环为例子，使用 python 和 pypy 分别运行，得到的运行结果分别如下：

相关主题

```
1 C:\Documents and Settings\Administrator\ 桌面 \doc\python>pypy loop.py
2 total run time:
3 8.42199993134
4
5 C:\Documents and Settings\Administrator\ 桌面 \doc\python>python loop.py
6 total run time:
7 106.391000032
```

可见使用 pypy 来编译和运行程序，其效率大大的提高。

Cython

Cython 是用 python 实现的一种语言，可以用来写 python 扩展，用它写出来的库都可以通过 i 的快。cython 里可以载入 python 扩展（比如 import math），也可以载入 c 的库的头文件（比如外也可以用它来写 python 代码。将关键部分重写成 C 扩展模块

Linux Cpython 的安装：

第一步：下载

```
1 [root@v5254085f259 cpython]# wget -N http://cython.org/release/Cython-0.15.1.
2 --2012-04-16 22:08:35-- http://cython.org/release/Cython-0.15.1.zip
3 Resolving cython.org... 128.208.160.197
4 Connecting to cython.org|128.208.160.197|:80... connected.
5 HTTP request sent, awaiting response... 200 OK
6 Length: 2200299 (2.1M) [application/zip]
7 Saving to: `Cython-0.15.1.zip'
8
9 100%[=====>] 2,200,299 1.96M/s in 1.1s
10
11 2012-04-16 22:08:37 (1.96 MB/s) - `Cython-0.15.1.zip' saved [2200299/2200299]
```

第二步：解压

```
1 | [root@v5254085f259 cpython]# unzip -o Cython-0.15.1.zip
```

第三步：安装

Python 代码优化堂见技巧

```
1 | python setup.py install
```

Cython 性能优化工具 安装完成后直接输入 cython，如果出现如下内容则表明安装成功。

```
1 | [root@v5254085f259 Cython-0.15.1]# cython
2 | Cython (http://cython.org) is a compiler for code written in the
3 | Cython language.  Cython is based on Pyrex by Greg Ewing.
4 |
5 | Usage: cython [options] sourcefile.{pyx,py} ...
6 |
7 | Options:
8 |   -V, --version                Display version number of cython compiler
9 |   -l, --create-listing         Write error messages to a listing file
10 |  -I, --include-dir <directory> Search for include files in named directory
11 |                               (multiple include directories are allowed).
12 |  -o, --output-file <filename> Specify name of generated C file
13 |  -t, --timestamps             Only compile newer source files
14 |  -f, --force                  Compile all source files (overrides implied -
15 |  -q, --quiet                  Don't print module names in recursive mode
16 |  -v, --verbose                Be verbose, print file names on multiple comp
17 |  -p, --embed-positions        If specified, the positions in Cython files o
18 | function definition is embedded in its docstring.
19 |  --cleanup <level>
20 | Release interned objects on python exit, for memory debugging.
21 |   Level indicates aggressiveness, default 0 releases nothing.
22 |  -w, --working <directory>
23 | Sets the working directory for Cython (the directory modules are searched fr
24 |  --gdb Output debug information for cygdb
25 |  -D, --no-docstrings
26 |       Strip docstrings from the compiled module.
27 |  -a, --annotate
28 |       Produce a colorized HTML version of the source.
29 |  --line-directives
30 |       Produce #line directives pointing to the .pyx source
31 |  --cplus
32 |       Output a C++ rather than C file.
33 |  --embed[=<method_name>]
34 |       Generate a main() function that embeds the Python interpreter.
35 |  -2                            Compile based on Python-2 syntax and code seman tics.
36 |  -3                            Compile based on Python-3 syntax and code seman tics.
37 |  --fast-fail                  Abort the compilation on the first error
38 |  --warning-error, -Werror      Make all warnings into errors
39 |  --warning-extra, -Wextra      Enable extra warnings
40 |  -X, --directive <name>=<value>
41 | [,<name=value,...] Overrides a compiler directive
```

其他平台上的安装可以参考文档：<http://docs.cython.org/src/quickstart/install.html>

Cython 代码与 python 不同，必须先编译，编译一般需要经过两个阶段，将 pyx 文件编译为 .c 文件。编译有多种方法：

内容通过命令行编译：

假设有如下测试代码，使用命令行编译为 .c 文件。

概览

```

1 | def sum(int a,int b):
2 |     print a+b
3 |
4 | [root@v5254085f259 test]# cython sum.pyx
5 | [root@v5254085f259 test]# ls
6 | total 76
7 | 4 drwxr-xr-x 2 root root 4096 Apr 17 02:45 .
8 | 4 drwxr-xr-x 4 root root 4096 Apr 16 22:20 ..
9 | 4 -rw-r--r-- 1 root root 35 Apr 17 02:45 1
10 | 60 -rw-r--r-- 1 root root 55169 Apr 17 02:45 sum.c
11 | 4 -rw-r--r-- 1 root root 35 Apr 17 02:45 sum.pyx

```

相关

评论

在 linux 上利用 gcc 编译为 .so 文件：

```

1 | [root@v5254085f259 test]# gcc -shared -pthread -fPIC -fwrapv -O2
2 | -Wall -fno-strict-aliasing -I/usr/include/python2.4 -o sum.so sum.c
3 | [root@v5254085f259 test]# ls
4 | total 96
5 | 4 drwxr-xr-x 2 root root 4096 Apr 17 02:47 .
6 | 4 drwxr-xr-x 4 root root 4096 Apr 16 22:20 ..
7 | 4 -rw-r--r-- 1 root root 35 Apr 17 02:45 1
8 | 60 -rw-r--r-- 1 root root 55169 Apr 17 02:45 sum.c
9 | 4 -rw-r--r-- 1 root root 35 Apr 17 02:45 sum.pyx
10 | 20 -rwxr-xr-x 1 root root 20307 Apr 17 02:47 sum.so

```

- 使用 distutils 编译

建立一个 setup.py 的脚本：

```

1 | from distutils.core import setup
2 | from distutils.extension import Extension
3 | from Cython.Distutils import build_ext
4 |
5 | ext_modules = [Extension("sum", ["sum.pyx"])]
6 |
7 | setup(
8 |     name = 'sum app',
9 |     cmdclass = {'build_ext': build_ext},
10 |     ext_modules = ext_modules
11 | )
12 |
13 |
14 | [root@v5254085f259 test]# python setup.py build_ext --inplace
15 | running build_ext
16 | cythoning sum.pyx to sum.c
17 | building 'sum' extension
18 | gcc -pthread -fno-strict-aliasing -fPIC -g -O2 -DNDEBUG -g -fwrapv -O3
19 | -Wall -Wstrict-prototypes -fPIC -I/opt/ActivePython-2.7/include/python2.7
20 | -c sum.c -o build/temp.linux-x86_64-2.7/sum.o

```

```
21 | gcc -pthread -shared build/temp.linux-x86_64-2.7/sum.o
22 | -o /root/cpython/test/sum.so
```

内容

编译完成之后可以导入到 python 中使用：

概览

```
1 | [root@v5254085f259 test]# python
2 | ActivePython 2.7.2.5 (ActiveState Software Inc.) based on
3 | Python 2.7.2 (default, Jun 24 2011, 11:24:26)
4 | [GCC 4.0.2 20051125 (Red Hat 4.0.2-8)] on linux2
5 | Type "help", "copyright", "credits" or "license" for more information.
6 | >>> import pyximport; pyximport.install()
7 | >>> import sum
8 | >>> sum.sum(1,3)
```

稍后再进行一个简单的性能比较：

评论

清单 9. Cython 测试代码

```
1 | from time import time
2 | def test(int n):
3 |     cdef int a =0
4 |     cdef int i
5 |     for i in xrange(n):
6 |         a+= i
7 |     return a
8 |
9 | t = time()
10 | test(10000000)
11 | print "total run time:"
12 | print time()-t
```

测试结果：

```
1 | [GCC 4.0.2 20051125 (Red Hat 4.0.2-8)] on linux2
2 | Type "help", "copyright", "credits" or "license" for more information.
3 | >>> import pyximport; pyximport.install()
4 | >>> import ctest
5 | total run time:
6 | 0.00714015960693
```

清单 10. Python 测试代码

```
1 | from time import time
2 | def test(n):
3 |     a =0;
4 |     for i in xrange(n):
5 |         a+= i
6 |     return a
7 |
8 | t = time()
9 | test(10000000)
```

```
10 | print "total run time:"
11 | print time()-t
12 |
13 | [root@v5254085f259 test]# python test.py
14 | total run time:
15 | 0.971596002579
```

概览

从上述对比可以看到使用 Cython 的速度提高了将近 100 多倍。

Python 代码优化常见技巧

总结

定位程序性能瓶颈

Python 性能优化工具

本文初步探讨了 python 常见的性能优化技巧以及如何借助工具来定位和分析程序的性能瓶颈，
总结的工具或语言，希望能够更相关人员一些参考。

相关主题

评论

相关主题

- 参考 [Cython 官网](#)，了解更多关于 Cython 的信息。
 - 参考 [Pypy 官网](#)，了解更多关于 pypy 的知识。
 - 查看文章"[可爱的 Python: 用 Psyco 让 Python 运行得像 C 一样快](#)"，了解如何使用 Psyco 提
 - 参考 [Python 官网](#)，了解更多 Python 最新的动态。
 - 获取 [visualpytune](#)进行试用。
 - 了解更多关于 [Gprof2Dot](#)的信息。
 - 在 [developerWorks Linux 专区](#) 寻找为 Linux 开发人员（包括 [Linux 新手入门](#)）准备的更多[文章和教程](#)。
 - 在 developerWorks 上查阅所有 [Linux 技巧](#) 和 [Linux 教程](#)。
 - 随时关注 developerWorks [技术活动](#)和[网络广播](#)。
-

评论

添加或订阅评论，请先[登录](#)或[注册](#)。

☐ 有新评论时提醒我