# 代码性能分析

清华大学软件学院 刘强 潘天翔

效率是程序员之间永恒的话题

如何才能编写出运行更快、效率更高的程序

这是每一个优秀程序员不懈追求的目标！

# 代码性能优化

优化是对代码进行等价变换，使得变换后的代码运行结果与变换前的代码运行结果相同，但执行速度加快或存储开销减少。


- 代码性能优化是一门复杂的学问。
- 根据 80/20 原则，实现程序的重构、优化、扩展以及文档相关的事情通常需要消耗 80% 的工作量。

# 代码性能优化

- 在满足正确性、可靠性、健壮性、可读性等质量因素的前提下，设法提高程序的效率
- 以提高程序的全局效率为主，提高局部效率为辅
- 在优化程序效率时，应先找出限制效率的“瓶颈”
- 先优化数据结构和算法，再优化执行代码
- 时间效率和空间效率可能是对立的，应当分析哪一个因素更重要，再做出适当的折衷

# 代码性能优化

- 从一开始就要考虑程序性能，不要期待在开发结束后再做一些快速调整
- 正确的代码要比速度快的代码重要，任何优化都不能破坏代码的正确性


- 认真选择测试数据，使其能够代表实际的使用状况
- 永远不要在没有执行前后性能评估的情况下尝试对代码进行优化

# 案例分析

编写程序：读入一个文本文件，统计在该文本文件中每个英文单词出现的频率，并输出单词频率最高的100个单词。其中，单词的定义是连续的若干个小写英文字母。

如何有效地提高代码的执行效率？

# 案例分析

编写程序：读入一个文本文件，统计在该文本文件中每个英文单词出现的频率，并输出单词频率最高的100个单词。其中，单词的定义是连续的若干个小写英文字母。

单词示例

- 1个单词： as
- 2个单词： as,asd
- 4个单词： sa,fdf.fdf fdfdf

```python
# -*- coding: utf-8 -*-
# __author__ = 'Pan Tianxiang'
# __email__ = 'ptx9363@gmail.com'
```

文件头注释，说明了该文件的编码格式，作者，联系方式等

```python
import re
```

包引用，导入文件需要的各种包文件

```python
def SplitWords(InputFile):

    #读入文件
    fileobject = open(InputFile)
    try:
        alltext = fileobject.read()
    finally:
        fileobject.close()

    #分割单词
    words = re.split('[^a-zA-Z]+', alltext)

    #统计单词词频
    dic = {}
    for word in words:
        if word in dic.keys():
            dic[word]+= 1
        else:
            dic[word] = 1

    #排序
    result = sorted(dic.items(), key=lambda dic:dic[1], reverse=True)

    print(result[1:100])
```

分词函数，用于实现我们实际的程序功能

```python
if __name__ == '__main__':
    SplitWords('input.txt')
```

主函数，程序的入口，相当于c++中的main函数

```
7  def SplitWords(InputFile):
8
9      #读入文件
10     fileobject = open(InputFile)
11     try:
12         alltext = fileobject.read()
13     finally:
14         fileobject.close()
15
16     #分割单词
17     words = re.split('[^a-zA-Z]+', alltext)
18
19     #统计单词词频
20     dic = {}
21     for word in words:
22         if word in dic.keys():
23             dic[word]+= 1
24         else:
25             dic[word] = 1
26
27     #排序
28     result = sorted(dic.items(), key=lambda dic:dic[1], reverse=True)
29
30     print(result[1:100])
```

文件读入

分割单词，使用正则表达式

词频统计，利用词典类型进行判重的操作

排序，这里用python内置的排序算法来实现

# 性能测试工具

Profile是Python语言内置的性能分析工具，它能够有效地描述程序运行的性能状况，提供各种统计数据帮助程序员找出程序中的性能瓶颈。

```
import profile

def profileTest():
    Total =1;
    for i in range(10):
        Total=Total*(i+1)
        print(Total)
    return Total

if __name__ == "__main__":
    profile.run("profileTest()")
```

左图是一个求阶乘的小程序，Profile工具仅需要一行代码就可以对所测试函数进行代码性能测试。

```
E:\Python\python.exe E:/venv/Profiletest.py
1
2
6
24
120
720
5040
40320
362880
3628800
         15 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 :0(exec)
       10    0.000    0.000    0.000    0.000 :0(print)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 Profiletest.py:7(profileTest)
        1    0.000    0.000    0.000    0.000 profile:0(profileTest())
        0    0.000             0.000          profile:0(profiler)
```

程序输出结果

程序执行时间

详细的函数性能数据报表

ncalls 函数的被调用次数
tottime 函数总计运行时间，这里除去函数中调用的其他函数运行时间
percall 函数运行一次的平均时间，等于tottime/ncalls
cumtime 函数总计运行时间，这里包含调用的其他函数运行时间
percall 函数运行一次的平均时间，等于cumtime/ncalls
filename:lineno(function) 函数所在的文件名，函数的行号，函数名

```
         902806 function calls (902803 primitive calls) in 4.562 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 :0(_getdefaultlocale)
       31    0.000    0.000    0.000    0.000 :0(append)
                        ......
   500000    1.141    0.000    1.141    0.000 :0(keys)
    41/40    0.000    0.000    0.000    0.000 :0(len)
        4    0.000    0.000    0.000    0.000 :0(min)
        1    0.000    0.000    0.000    0.000 :0(open)
        4    0.000    0.000    0.000    0.000 :0(ord)
        1    0.000    0.000    0.000    0.000 :0(print)
        1    0.094    0.094    0.094    0.094 :0(read)
                        ......
        1    0.750    0.750    1.203    1.203 :0(sorted)
        1    0.312    0.312    0.312    0.312 :0(split)
                        ......
        1    0.000    0.000    0.000    0.000 sre_parse.py:750(parse)
        2    0.000    0.000    0.000    0.000 sre_parse.py:90(__init__)
```

程序总耗时 4.562s

Keys()函数执行500000次，耗时 1.141s

输出几乎不耗时

输入耗时 0.094s

排序函数，耗时1.203s

分词函数，耗时 0.312s

# 案例：原因分析

为什么 keys() 函数的调用复杂度过高？

> keys(): Return a new view of the dictionary's keys (docs.python.org)

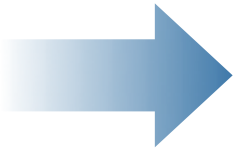
- 原因：每调用一次keys()函数，系统就会生成一个新的字典迭代器，如果这个生成过程重复50万次，……
- 优化：使用 in 操作符直接代替keys()，不再每次生成新的迭代器。

```
8  def SplitWords(InputFile):
9
10     #读入文件
11     fileobject = open(InputFile)
12     try:
13         alltext = fileobject.read()
14     finally:
15         fileobject.close()
16
17     #分割单词
18     words = re.split('[^a-zA-Z]+', alltext)
19
20     #统计单词词频
21     dic = {}
22     for word in words:
23         #if word in dic.keys():
24         if word in dic:
25             dic[word]+= 1
26         else:
27             dic[word] = 1
28
29     result = sorted(dic.items(), key=lambda dic:dic[1], reverse=True)
30
31     print(result[1:100])
```

```
402806 function calls (402803 primitive calls) in 2.250 seconds
```

测试发现：程序的总执行时间降低到 2.250s

# 案例：优化结果

优化前 4.562 s  2.250 s 优化后

- 性能优化的关键是如何发现问题，寻找解决问题的方法。
- 有效的测试是不可缺少的，通过测试找出真正的瓶颈，并分析优化结果
- 要避免不必要的优化，避免不成熟的优化，不成熟的优化是错误的来源

# Python 代码性能优化

## 改进算法，选择合适的数据结构

- 良好的算法对性能起到关键作用，因此性能改进的首要点是对算法改进
- 算法时间复杂性的排序依次是

  O(1) ➔ O(lg n) ➔ O(n lg n) ➔ O(n^2) ➔ O(n^3) ➔ O(n^k) ➔ O(k^n) ➔ O(n!)

- 对成员的查找访问等操作，字典（dictionary）要比列表（list）更快
- 集合（set）的并、交、差的操作比列表（list）的迭代要快

# Python 代码性能优化

循环优化的基本原则：尽量减少循环过程中的计算量，在多重循环的时候，尽量将内层的计算提到上一层。

字符串的优化： Python的字符串对象是不可改变的。字符串连接的使用尽量使用 join() 而不是 +。当对字符串可以使用正则表达式或者内置函数处理时，选择内置函数。

使用列表解析和生成器表达式：列表解析要比在循环中重新构建一个新的 list 更为高效，因此可以利用这一特性来提高运行的效率。

# 谢谢大家！

THANKS