

python 性能优化实践

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

1. 避免重复计算

代码

```
#lcm_test0.py
def lowest_common_multiplier(arg1, arg2):
    i = max(arg1, arg2)
    while i < (arg1 * arg2):
        if i % min(arg1, arg2) == 0:
            return i
        i += max(arg1, arg2)
    return(arg1 * arg2)

print lowest_common_multiplier(41391237, 2830338)
```

```
#lcm_test1.py
def lowest_common_multiplier(arg1, arg2):
    i = max(arg1, arg2)
    _max = i
    _min = min(arg1, arg2)
    while i < (arg1 * arg2):
        if i % _min == 0:
            return i
        i += _max
    return(arg1 * arg2)

print lowest_common_multiplier(41391237, 2830338)
```

测试

```

F:\Codes\python\python_PERF_OPT\demos\L3>python -m cProfile lcm_test0.py
39050396982702
1886895 function calls in 0.692 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    0.001    0.001    0.692    0.692 lcm_test0.py:2(<module>)
      1    0.514    0.514    0.692    0.692 lcm_test0.py:2(lowest_common_multiplier)
943446    0.089    0.000    0.089    0.000 <max>
      1    0.000    0.000    0.000    0.000 <method 'disable' of '_lsprof.Profiler' objects>
943446    0.089    0.000    0.089    0.000 <min>

F:\Codes\python\python_PERF_OPT\demos\L3>python -m cProfile lcm_test1.py
39050396982702
5 function calls in 0.287 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    0.000    0.000    0.287    0.287 lcm_test1.py:2(<module>)
      1    0.286    0.286    0.286    0.286 lcm_test1.py:2(lowest_common_multiplier)
      1    0.000    0.000    0.000    0.000 <max>
      1    0.000    0.000    0.000    0.000 <method 'disable' of '_lsprof.Profiler' objects>
      1    0.000    0.000    0.000    0.000 <min>

```

2. 真假多线程

开发多线程的应用程序，是日常软件开发中经常会遇到的需求。

Python支持多线程threading模块

```

from timer import Timer
import threading
import time
def compute():
    for i in xrange(10000):
        rs = 0
        for i in xrange(1000):
            rs += i
def mysleep():
    time.sleep(1)

def test(func):
    print func.__name__
    with Timer() as t:
        for i in xrange(10):
            func()
    print "> single thread 10 times : %s s" % t.secs

    with Timer() as t:
        thread_list = []
        for i in xrange(10):
            th = threading.Thread(target = func, args=())
            th.start()
            thread_list.append(th)
        for th in thread_list:
            th.join()
    print "> multiple threads 1 time : %s s" % t.secs

if __name__ == "__main__":
    test(compute)
    test(mysleep)

```

```

F:\Codes\python\python_PERF_OPT\demos\L3>python threading_test.py
compute
=> single thread 10 times : 2.76899981499 s
=> multiple threads 1 time : 6.64299988747 s
mysleep
=> single thread 10 times : 10.0009999275 s
=> multiple threads 1 time : 1.0039999485 s

```

GIL

（Global Interpreter Lock）全局解释器锁

Python解释器被GIL保护，该锁定只允许一次执行一个线程，即便存在多个可用的处理器。在计算型密集的程序中，严重限制了线程的作用。实际上，就像上边我们看到的那样，在密集计算型程序中使用多线程，经常比顺序执行慢很多。

缺点：

1. 不顺应计算机的发展潮流
2. 限制多线程程序的速度

存在的理由:

1. 写python的扩展(module)时会遇到锁的问题,程序员需要繁琐地加解锁来保证线程安全
2. 会较大幅度地减低单线程程序的速度

threading模块的意义何在

既然多线程会慢,那么threading模块的存在意义是什么

对于爬虫或者下载这种I/O密集型的程序,使用threading模块是有意义的

计算密集型程序的并发

使用C扩展,或者multiprocessing模块,重复下上边的例子

```
from timer import Timer
import multiprocessing
import time
def compute():
    for i in xrange(10000):
        rs = 0
        for i in xrange(1000):
            rs += i

def mysleep():
    time.sleep(1)

def test(func):
    print func.__name__
    with Timer() as t:
        for i in xrange(10):
            func()
    print "> single Process 10 times : %s s" % t.secs

    with Timer() as t:
        process_list = []
        for i in xrange(10):
            th = multiprocessing.Process(target = func, args=())
            th.start()
            process_list.append(th)
        for th in process_list:
            th.join()
    print "> multiple Processes 1 time : %s s" % t.secs

if __name__ == "__main__":
    test(compute)
    test(mysleep)
```

- 测试结果

```
F:\Codes\python\python_PERF_OPT\demos\L3>python multiprocessing_test.py
compute
=> single Process 10 times : 2.70399999619 s
=> multiple Processes 1 time : 0.824000120163 s
mysleep
=> single Process 10 times : 10.00099999275 s
=> multiple Processes 1 time : 1.15300011635 s
```

multiprocessing可以绕过GIL锁实现真正的并发，但是创建进程这个操作比创建线程重很多

两个关键点：

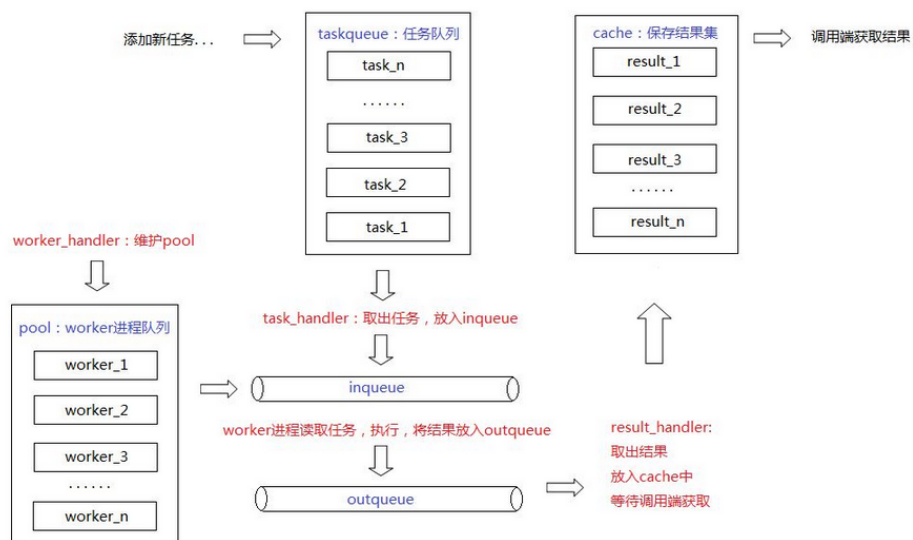
第一是衡量你的程序是I/O密集还是计算密集

第二是不要频繁去创建新进程

小结

多进程的优势	多进程的劣势
避开GIL的限制，可以使用多核操作系统	更多的内存消耗
进程使用独立的内存空间，避免竞态问题	进程间的数据共享变得更加困难
子进程容易中断（killable）	IPC（Interprocess communication，进程间通信）处理比线程困难

进程池



```
#import multiprocessing
g_inputq = multiprocessing.JoinableQueue(Queue.MAXSIZE)
g_outputq = multiprocessing.JoinableQueue(Queue.MAXSIZE)
#初始化进程池
for i in range(0, POOLSIZE):
    t = multiprocessing.Process(target=Distribute, args=(g_inputq,g_outputq))
    t.daemon = True
    t.start()
```

核心在于处理进程间通信，加上socket通信，可以搭建一个简单的分布式系统

- 注意事项

3. JSON解析

JSON文件解析是日常工作比较常见的一个任务。

[JSON](#)([JavaScript](#) Object Notation, JS 对象标记) 是一种轻量级的数据交换格式。它基于 [ECMAScript](#) 规范的一个子集，采用完全独立于编程语言的文本格式来存储和表示数据。简洁和清晰的层次结构使得 JSON 成为理想的数据交换语言。易于人阅读和编写，同时也易于机器解析和生成，并有效地提升网络传输效率。

JSON demo

```
"Resource": [{
    "ID": "3277400",
    "Type": "Cursor",
    "Size": "134"
},
{
    "ID": "3277400",
    "Type": "Cursor",
    "Size": "134"
},
{
    "ID": "3277400",
    "Type": "Cursor",
    "Size": "134"
}
]
```

测试代码

```
#json_test.py
import json
from timer import *
import sys
with Timer() as t:
    for i in xrange(10):
        json.loads(open(sys.path[0]+"//test.json").read())
print "json loads %s" % t.secs
```

测试结果

python2.7.13中运行

```
F:\Codes\python\python_PERF_OPT\demos\L3>python json_test.py
json loads 1.1369998455

F:\Codes\python\python_PERF_OPT\demos\L3>python
Python 2.7.13 <v2.7.13:a06454b1afa1, Dec 17 2016, 20:53:40> [MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
```

python2.6.5中运行

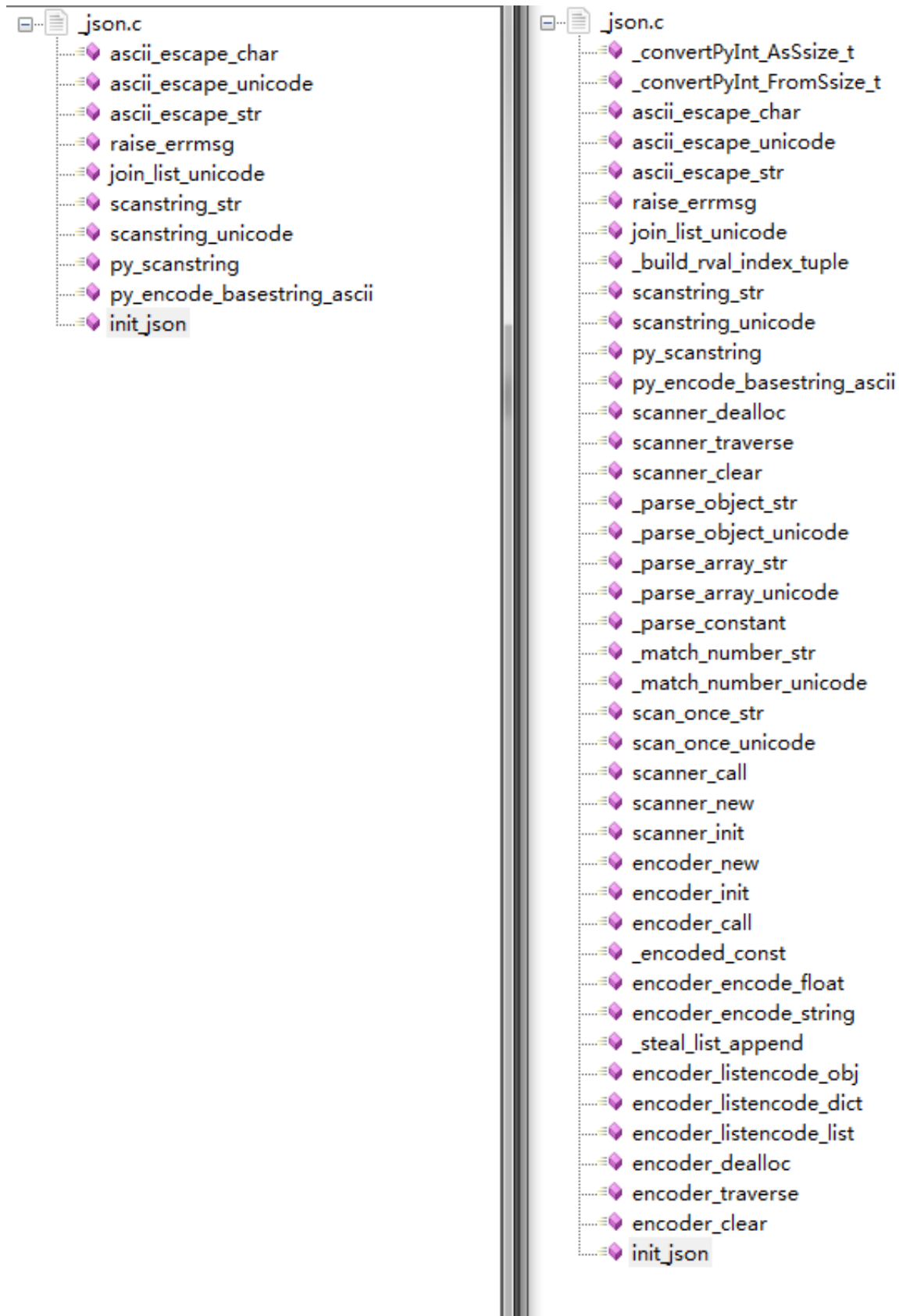
```
C:\Python26>python.exe F:\Codes\python\python_PERF_OPT\demos\L3\json_test.py
json loads 10.6470000744

C:\Python26>python
Python 2.6.5 <r265:79096, Mar 19 2010, 18:02:59> [MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
```

原因分析

Python2.6的json解析并不是没有使用C语言加速，而是C语言加速模块效果不如Python2.7的好。

直观点看，左边是2.6的C加速代码，右边是2.7的。



选用合适的第三方库

这里介绍一个simplejson， pip 安装


```
C:\Python27\Scripts>pip install simplejson
Collecting simplejson
  Downloading simplejson-3.10.0-cp27-cp27m-win_amd64.whl (67kB)
    100% |#####| 71kB 363kB/s
Installing collected packages: simplejson
Successfully installed simplejson-3.10.0
```

```
#json_test.py
import json
from timer import *
import sys
with Timer() as t:
    for i in xrange(10):
        json.loads(open(sys.path[0]+"//test.json").read())
print "json loads %s" % t.secs

import simplejson
with Timer() as t:
    for i in xrange(10):
        simplejson.loads(open(sys.path[0]+"//test.json").read())
print "simplejson loads %s" % t.secs
```

加载速度测试：

```
F:\Codes\python\python_PERF_OPT\demos\L3>python json_test.py
json loads 1.10600018501
simplejson loads 0.37700009346
```

小结

- 去了解你import进来的模块，随便装个轮子你的程序可以跑，但不一定跑得远跑得快，通过测试对比选取合适的库
- 不要功能实现了就觉得万事大吉，多做一点，很多时候进步是由这一点带来的
- 有条件的情况下，尽量去升级到较新的Python版本和库的版本

PS:必须采用编译安装的方式（编译安装会开启c语言优化，而源码拷贝功能是正常的，但是由python实现的）效率是高于python2.7自带的json库的。

4. URL中提取域名

前段时间一个网址安全检测的项目里边需要从网址中提取域名，eg：

<http://sports.qq.com/a/20170511/003170.htm>，域名就是qq.com。看起来挺简单的一个算法，Python中也有专门的库，tldextract。

安装

```
>>pip install tldextract
***
***
Successfully installed tldextract-2.0.2
```

测试

```
#tldextract_test.py
import tldextract
from timer import *
urls = ["https://www.baidu.com/s?wd=asdf", "http://sports.qq.com/a/20170511/003170.htm",
"abc.sd.def.ru", "http://blog.csdn.net/arbel/article/details/7957782",
        "http://www.vrplumber.com/programming/runsnakerun/"
        ]

with Timer() as t:
    for i in xrange(10):
        for url in urls:
            ext = tldextract.extract(url)
            #print ext.domain
print "tldextract %s"%t.secs
```

```
F:\Codes\python\python_PERF_OPT\demos\L3>python tldextract_test.py
tldextract 2.02999997139
```

5个url取域名，重复运行10次，耗时2s以上。

原因分析

```
python -m cProfile -o tld.prof tldextract_test.py
```

- runsnake

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）

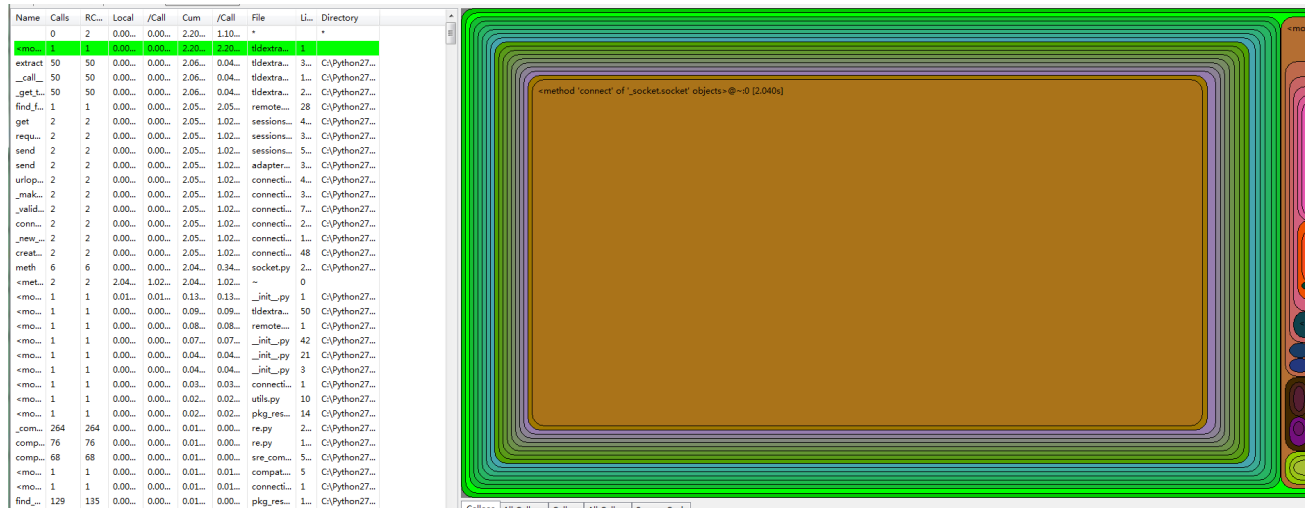
然后安装runsnakerun

```
pip install runsnakerun
```

安装完之后会生成一个runsnake.py

- 使用

```
python runsnake.py xxxx.prof
```



Python27\Lib\site-packages\tlidextract\tlid_set

Python27\Lib\site-packages\tlidextract\tlid_set_snapshot

小结

从耗时分析入手，找到程序运营不正常的原因

5. simhash算法

原理

局部敏感性哈希

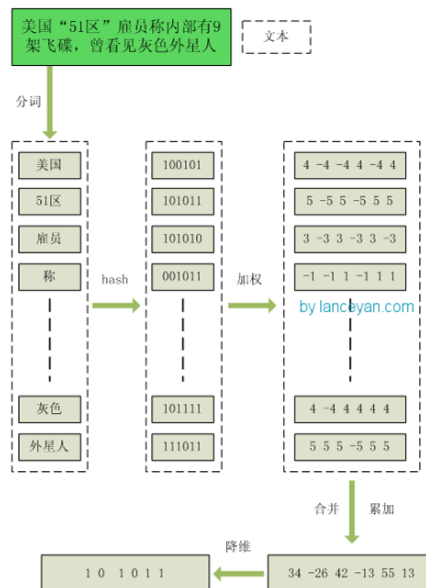
- 1. 局部敏感性哈希(Locality-Sensitive Hashing, LSH)
- 2. simhash、海明距离：google在 网页去重中使用

```
if __name__ == '__main__':  
    s = 'This is a test string for testing0'  
    hash1 = simhash(s.split())  
    s = 'This is a test string for testingA'  
    hash2 = simhash(s.split())  
    ...  
    print "_____result_____  
    #print type(hash1)  
    print "hash1 : ", hash1.hash  
    print "hash2 : ", hash2.hash  
    print (100*float(hash1.similarity(hash2)), "percent similar")  
    ...  
    print (hash1.hamming_distance(hash2), "bits differ out of", hash1.hashbits)
```

```
hash1 : 3839656072  
hash2 : 3839656168  
(99.99999749977613, 'percent similar')  
(2, 'bits differ out of', 32)
```

simhash原理

降维



代码

```

from hashtype import hashtype

class simhash(hashtype):
    def create_hash(self, tokens):
        """Calculates a Charikar simhash with appropriate bitlength.

        Input can be any iterable, but for strings it will automatically
        break it into words first, assuming you don't want to iterate
        over the individual characters. Returns nothing.

        Reference used: http://dsrg.mff.cuni.cz/~holub/sw/shash
        """
        if type(tokens) == str:
            tokens = tokens.split()
        v = [0]*self.hashbits
        for t in [self._string_hash(x) for x in tokens]:
            bitmask = 0
            for i in xrange(self.hashbits):
                bitmask = 1 << i
                if t & bitmask:
                    v[i] += 1
                else:
                    v[i] -= 1

        fingerprint = 0
        for i in xrange(self.hashbits):
            if v[i] >= 0:
                fingerprint += 1 << i
        self.hash = fingerprint

    def _string_hash(self, v):
        """A variable-length version of Python's builtin hash. Neat!"""
        if v == "":
            return 0
        else:
            x = ord(v[0])<<7
            m = 1000003
            mask = 2**self.hashbits-1
            for c in v:
                x = ((x*m)^ord(c)) & mask
            x ^= len(v)
            if x == -1:
                x = -2
            return x

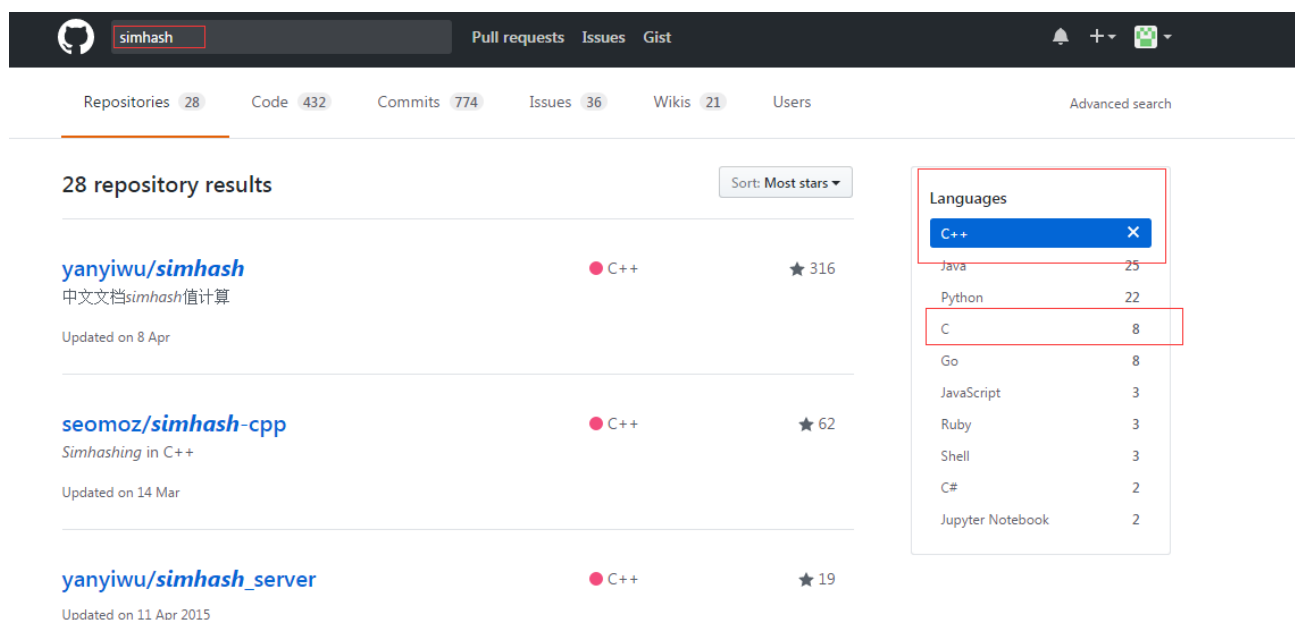
    def similarity(self, other_hash):
        """Calculate how different this hash is from another simhash.
        Returns a float from 0.0 to 1.0 (inclusive)
        """
        if type(other_hash) != simhash:

```

```
        raise Exception('Hashes must be of same type to find similarity')
    b = self.hashbits
    if b!= other_hash.hashbits:
        raise Exception('Hashes must be of equal size to find similarity')
    return float(b - self.hamming_distance(other_hash)) / b
```

优化

使用C/C++版本



28 repository results

Sort: Most stars ▼

Repository	Language	Stars
yanyiwu/simhash	C++	316
seomoz/simhash-cpp	C++	62
yanyiwu/simhash_server	C++	19

Languages

- C++
- Java 25
- Python 22
- C 8
- Go 8
- JavaScript 3
- Ruby 3
- Shell 3
- C# 2
- Jupyter Notebook 2

6. 极速数据处理

Numba

概述

Numba (<http://numba.pydata.org/>) 是一个模块，让你能够（通过装饰器）控制Python解释器把函数转变成机器码。因此，Numba实现了与C和Cython同样的性能，但是不需要用新的解释器或者写C代码。

这个模块可以按需生成优化的机器码，甚至可以编译成CPU或GPU可执行代码。

```
#numba_test.py
from numba import jit
from numpy import arange

# jit装饰器告诉Numba编译函数
# 当函数被调用时，Numba会把参数类型引入
@jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i, j]
    return result

a = arange(9).reshape(3, 3)
print(sum2d(a))
```

测试

```
D:\ProgramData\Anaconda2>python -m cProfile F:\Codes\python\python_PERF_OPT\demos\L3\numba_test.py> 1.txt
D:\ProgramData\Anaconda2>python -m cProfile F:\Codes\python\python_PERF_OPT\demos\L3\numba_test_withoutJIT.py >2.txt
```

1.txt						2.txt					
1	36.0	248181 function calls (240659 primitive calls) in 0.373 seconds				1	36.0	127901 function calls (126409 primitive calls) in 0.215 seconds			
2		Ordered by: standard name				2		Ordered by: standard name			
3						3					
ncalls	tottime	pertime	cumtime	pertime	filename:lineno(function)	ncalls	tottime	pertime	cumtime	pertime	filename:lineno(function)
1	0.000	0.000	0.000	0.000	<string>:1(<module>)	1	0.000	0.000	0.000	0.000	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	<string>:1(ArgInfo)	1	0.000	0.000	0.000	0.000	<string>:1(ArgInfo)
1	0.000	0.000	0.000	0.000	<string>:1(ArgSpec)	1	0.000	0.000	0.000	0.000	<string>:1(ArgSpec)
1	0.000	0.000	0.000	0.000	<string>:1(Arguments)	1	0.000	0.000	0.000	0.000	<string>:1(Arguments)
1	0.000	0.000	0.000	0.000	<string>:1(Attribute)	1	0.000	0.000	0.000	0.000	<string>:1(Attribute)
1	0.000	0.000	0.000	0.000	<string>:1(Indexing)	1	0.000	0.000	0.000	0.000	<string>:1(Loop)
1	0.000	0.000	0.000	0.000	<string>:1(Loop)	1	0.000	0.000	0.000	0.000	<string>:1(Match)
1	0.000	0.000	0.000	0.000	<string>:1(Mismatch)	1	0.000	0.000	0.000	0.000	<string>:1(Mismatch)
1	0.000	0.000	0.000	0.000	<string>:1(ModuleInfo)	1	0.000	0.000	0.000	0.000	<string>:1(ModuleInfo)
1	0.000	0.000	0.000	0.000	<string>:1(Partition)	1	0.000	0.000	0.000	0.000	<string>:1(Partition)
1	0.000	0.000	0.000	0.000	<string>:1(QuickSortImplementation)	1	0.000	0.000	0.000	0.000	<string>:1(QuickSortImplementation)
1	0.000	0.000	0.000	0.000	<string>:1(SetLoop)	1	0.000	0.000	0.000	0.000	<string>:1(SetLoop)
1	0.000	0.000	0.000	0.000	<string>:1(Status)	1	0.000	0.000	0.000	0.000	<string>:1(Status)
1	0.000	0.000	0.000	0.000	<string>:1(Traceback)	1	0.000	0.000	0.000	0.000	<string>:1(Traceback)

pandas

311 服务请求数据统计

```

#panda_test.py
import pandas as pd
import time
import csv
import collections

SOURCE_FILE = './311.csv'

def readCSV(fname):
    with open(fname, 'rb') as csvfile:
        reader = csv.DictReader(csvfile)
        lines = [line for line in reader]
        return lines

def process(fname):
    content = readCSV(fname)
    incidents_by_zipcode = collections.defaultdict(int)
    for record in content:
        incidents_by_zipcode[toFloat(record['Incident Zip'])] += 1
    return sorted(incidents_by_zipcode.items(), reverse=True, key=lambda a: int(a[1]))[:10]

def toFloat(number):
    try:
        return int(float(number))
    except:
        return 0

def process_pandas(fname):
    df = pd.read_csv(fname, dtype={
        'Incident Zip': str, 'Landmark': str, 'Vehicle Type': str, 'Ferry
Direction': str})
    df['Incident Zip'] = df['Incident Zip'].apply(toFloat)
    column_names = list(df.columns.values)
    column_names.remove("Incident Zip")
    column_names.remove("Unique Key")
    return df.drop(column_names, axis=1).groupby(['Incident Zip'],
sort=False).count().sort('Unique Key', ascending=False).head(10)

init = time.clock()
total = process(SOURCE_FILE)
endtime = time.clock() - init
for item in total:
    print "%s\t%s" % (item[0], item[1])

print "(Pure Python) time: %s" % (endtime)

```



```
init = time.clock()
total = process_pandas(SOURCE_FILE)
endtime = time.clock() - init
print total
print "(Pandas) time: %s" % (endtime)
```

性能对比

```
D:\ProgramData\Anaconda2>python F:\Codes\python\python_PERF_OPT\demos\L3\panda_test.py
0      119344
11226   37903
10467   31997
11207   28624
10458   27543
11221   25597
10453   24704
10468   24037
10457   23574
11233   23302
<Pure Python> time: 240.849001647
```

```
Incident Zip
0      119344
11226   37903
10467   31997
11207   28624
10458   27543
11221   25597
10453   24704
10468   24037
10457   23574
11233   23302
<Pandas> time: 28.2113526559
```