Python多线程鸡年不鸡肋

nmask / 2017-02-17 13:39:44 / 浏览数 3475 安全技术 技术讨论 顶(0) 踩(0)

当初在刚学习python多线程时，上网搜索资料几乎都是一片倒的反应python没有真正意义上的多线程，python多线程就是鸡肋。当时不明所以，只是了解到python带有GI

经过对比python与java的多线程测试，我发现python多线程的效率确实不如java，但远还没有达到鸡肋的程度，那么跟其他机制相比较呢？

观点：用多进程替代多线程需求

辗转了多篇博文，我看到了一些网友的观点，觉得应该使用python多进程来代替多线程的需求，因为多进程不受GIL的限制。于是我便动手使用多进程去解决一些并发问题，
那么是否多进程能完全替代多线程呢？别急，我们继续往下看。

观点：协程为最佳方案

协程的概念目前来说是比较火热的，协程不同于线程的地方在于协程不是操作系统进行切换，而是由程序员编码进行切换的，也就是说切换是由程序员控制的，这样就没有了

测试数据

好了，网上的观点无非是使用多进程或者协程来代替多线程（当然换编程语言，换解释器之类方法除外），那么我们就来测试下这三者的性能之差。既然要公平测试，就应该

IO密集型测试

测试IO密集型，我选择最常用的爬虫功能，计算爬虫访问bing所需要的时间。（主要测试多线程与协程，单线程与多进程就不测了，因为没有必要）
测试代码：

```
#! -*- coding:utf-8 -*-

from gevent import monkey;monkey.patch_all()
import gevent
import time
import threading
import urllib2

def urllib2_(url):
   try:
       urllib2.urlopen(url,timeout=10).read()
   except Exception,e:
       print e

def gevent_(urls):
   jobs=[gevent.spawn(urllib2_,url) for url in urls]
   gevent.joinall(jobs,timeout=10)
   for i in jobs:
       i.join()

def thread_(urls):
   a=[]
   for url in urls:
       t=threading.Thread(target=urllib2_,args=(url,))
       a.append(t)

   for i in a:
       i.start()
   for i in a:
       i.join()

if __name__=="__main__":
   urls=["https://www.bing.com/"]*10
   t1=time.time()
   gevent_(urls)
   t2=time.time()
   print 'gevent-time:%s' % str(t2-t1)
   thread_(urls)
   t4=time.time()
   print 'thread-time:%s' % str(t4-t2)
```

测试结果：
访问10次
gevent-time:0.380326032639
thread-time:0.376606941223
访问50次
gevent-time:1.3358900547
thread-time:1.59564089775
访问100次
gevent-time:2.42984986305
thread-time:2.5669670105
访问300次
gevent-time:6.66330099106
thread-time:10.7605059147
从结果可以看出，当并发数不断增大时，协程的效率确实比多线程要高，但在并发数不是那么高时，两者差异不大。

CPU密集型

CPU密集型，我选择科学计算的一些功能，计算所需时间。（主要测试单线程、多线程、协程、多进程）
测试代码：

```
#! -*- coding:utf-8 -*-

from multiprocessing import Process as pro
from multiprocessing.dummy import Process as thr
from gevent import monkey;monkey.patch_all()
import gevent

def run(i):
   lists=range(i)
   list(set(lists))

if __name__=="__main__":
   '''
   ■■■
   '''
   for i in range(30):      ##10-2.1s 20-3.8s 30-5.9s
       t=pro(target=run,args=(5000000,))
       t.start()

   '''
   ■■■
   '''
   # for i in range(30):    ##10-3.8s  20-7.6s  30-11.4s
   #   t=thr(target=run,args=(5000000,))
   #   t.start()

   '''
   ■■
   '''
   # jobs=[gevent.spawn(run,5000000) for i in range(30)]  ##10-4.0s 20-7.7s 30-11.5s
   # gevent.joinall(jobs)
   # for i in jobs:
   #   i.join()

   '''
   ■■■
   '''
   # for i in range(30):  ##10-3.5s  20-7.6s 30-11.3s
   #   run(5000000)
```

测试结果：

- 并发10次：【多进程】2.1s 【多线程】3.8s 【协程】4.0s 【单线程】3.5s
- 并发20次：【多进程】3.8s 【多线程】7.6s 【协程】7.7s 【单线程】7.6s
- 并发30次：【多进程】5.9s 【多线程】11.4s 【协程】11.5s 【单线程】11.3s

可以看到，在CPU密集型的测试下，多进程效果明显比其他的好，多线程、协程与单线程效果差不多。这是因为只有多进程完全使用了CPU的计算能力。在代码运行时，我们

本文结论

从两组数据我们不难发现，python多线程并没有那么鸡肋。如若不然，Python3为何不去除GIL呢？对于此问题，Python社区也有两派意见，这里不再论述，我们应该尊重
至于何时该用多线程，何时用多进程，何时用协程？想必答案已经很明显了。
当我们需要编写并发爬虫等IO密集型的程序时，应该选用多线程或者协程（亲测差距不是特别明显）；当我们需要科学计算，设计CPU密集型程序，应该选用多进程。当然
答案已经给出，本文是否就此收尾？既然已经论述Python多线程尚有用武之地，那么就来介绍介绍其用法吧。

Multiprocessing.dummy模块

Multiprocessing.dummy用法与多进程Multiprocessing用法类似，只是在import包的时候，加上.dummy。
用法参考Multiprocessing用法

threading模块

这是python自带的threading多线程模块，其创建多线程主要有2种方式。一种为继承threading类，另一种使用threading.Thread函数，接下来将会分别介绍这两种用法。

Usage【1】

利用threading.Thread()函数创建线程。
代码：

```
def run(i):
   print i

for i in range(10):
   t=threading.Thread(target=run,args=(i,))
   t.start()
```

说明：Thread()函数有2个参数，一个是target，内容为子线程要执行的函数名称；另一个是args，内容为需要传递的参数。创建完子线程，将会返回一个对象，调用对象的

线程对象的方法：

- Start() 开始线程的执行
- Run() 定义线程的功能的函数
- Join(timeout=None) 程序挂起，直到线程结束；如果给了timeout，则最多阻塞timeout秒
- getName() 返回线程的名字
- setName() 设置线程的名字
- isAlive() 布尔标志，表示这个线程是否还在运行
- isDaemon() 返回线程的daemon标志
- setDaemon(daemonic) 把线程的daemon标志设为daemonic（一定要在start（）函数前调用）
- t.setDaemon(True) 把父线程设置为守护线程，当父进程结束时，子进程也结束。

threading类的方法：

- threading.enumerate() 正在运行的线程数量

Usage【2】

通过继承threading类，创建线程。
代码：

```
import threading

class test(threading.Thread):
   def __init__(self):
       threading.Thread.__init__(self)

   def run(self):
       try:
           print "code one"
       except:
           pass

for i in range(10):
   cur=test()
   cur.start()
for i in range(10):
   cur.join()
```

说明：此方法继承了threading类，并且重构了run函数功能。

获取线程返回值问题

有时候，我们往往需要获取每个子线程的返回值。然而通过调用普通函数，获取return值的方式在多线程中并不适用。因此需要一种新的方式去获取子线程返回值。
代码：

```
import threading

class test(threading.Thread):
   def __init__(self):
       threading.Thread.__init__(self)

   def run(self):
       self.tag=1

   def get_result(self):
       if self.tag==1:
           return True
       else:
           return False

f=test()
f.start()
while f.isAlive():
   continue
print f.get_result()
```

说明：多线程获取返回值的首要问题，就是子线程什么时候结束？我们应该什么时候去获取返回值？可以使用isAlive()方法判断子线程是否存活。

控制线程运行数目

当需要执行的任务非常多时，我们往往需要控制线程的数量，threading类自带有控制线程数量的方法。
代码：

```
import threading

maxs=10  ##■■■■■■■
threadLimiter=threading.BoundedSemaphore(maxs)
class test(threading.Thread):
   def __init__(self):
       threading.Thread.__init__(self)

   def run(self):
       threadLimiter.acquire()  #■■
       try:
           print "code one"
       except:
           pass
       finally:
           threadLimiter.release() #■■

for i in range(100):
   cur=test()
   cur.start()
for i in range(100):
   cur.join()
```

说明：以上程序可以控制多线程并发数为10，超过这个数量会引发异常。
除了自带的方法，我们还可以设计其他方案：

```
threads=[]
'''
■■■■■■
'''
for i in range(10):
   t=threading.Thread(target=run,args=(i,))
   threads.append(t)
'''
■■■■■■■■
'''
for t in threads:
   t.start()
   while True:
```

```
        #■■■■■■■■■■■■,■■■■■5■■■while■■,
        #■■for■■■■■■■■■.■■■■■■■while■■■■■■■■
        if(len(threading.enumerate())<5):
            break
```

以上两种方式皆可以，本人更喜欢用下面那种方式。

线程池

```
import threadpool
def ThreadFun(arg1,arg2):
   pass
def main():
   device_list=[object1,object2,object3......,objectn]#■■■■■■■■■■
   task_pool=threadpool.ThreadPool(8)#8■■■■■■■■■■■■
   request_list=[]#■■■■■■■
   #■■■■■■■■■
   for device in device_list:
       request_list.append(threadpool.makeRequests(ThreadFun,[((device, ), {})]))
   #■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■map■■■■■■■■■■■■■■■■■
   map(task_pool.putRequest,request_list)
   #■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
   task_pool.poll()
if __name__=="__main__":
   main()
```

多进程问题，可以赶赴Python多进程现场，其他关于多线程问题，可以下方留言讨论

申明：本文谈不上原创，其中借鉴了网上很多大牛的文章，本人只是在此测试论述Python多线程相关问题，并简单介绍Python多线程的基本用法，为新手朋友解惑。

点击收藏 | 0 关注 | 0

上一篇：大概统计了下，奖金、积分和月度奖励... 下一篇：内网漫游之SOCKS代理大结局

1. 1 条回复

cover 2017-02-17 15:02:33

协程这地方可以在深入一点，应该吧协程原理多讲一下，在IO延迟很大的情况下，基本上线程就不如协程了。

0 回复Ta

目录