**Python多进程和多线程**

参考资料：

# 概念

**任务**可以理解为进程（process），如打开一个word就是启动一个word进程。在一个word进程之中不只是进行打字输入，还需要拼写检查、打印等子任务，我们可以把进程中的这些子任务称为线程（thread）。

由于每个进程至少要干一件事，那么一个进程至少有一个线程，有时候有的复杂进程有多个线程，在进程中的多个线程是可以同时执行的。多线程的执行方式和多进程是一样的，也是由操作系统在多个线程之间快速切换，让每个线程都短暂地交替运行，看起来就像同时执行一样。当然，**真正地同时执行多线程需要多核CPU才可能实现。**

总结一下就是，多任务的实现有3种方式：

* **多进程模式；**

启动多个进程，每个进程虽然只有一个线程，但多个进程可以一块执行多个任务。

* **多线程模式**

启动一个进程，在一个进程内启动多个线程，这样，多个线程也可以一块执行多个任务。

* **多进程+多线程模式。**

启动多个进程，每个进程再启动多个线程，这样同时执行的任务就更多了，当然这种模型更复杂，实际很少采用。同时执行多个任务通常各个任务之间并不是没有关联的，而是需要相互通信和协调，有时，**任务1**必须暂停等待**任务2**完成后才能继续执行，有时，任务3和任务4又不能同时执行，所以，多进程和多线程的程序的复杂度要远远高于我们前面写的单进程单线程的程序。

## 多进程

要让Python程序实现多进程（multiprocessing），我们先了解操作系统的相关知识。Unix/Linux操作系统提供了一个fork()系统调用，它非常特殊。普通的函数调用，调用一次，返回一次，但是fork()调用一次，返回两次，因为操作系统自动把当前进程（称为父进程）复制了一份（称为子进程），然后，分别在父进程和子进程内返回。

子进程永远返回0，而父进程返回子进程的ID。这样做的理由是，一个父进程可以fork出很多子进程，所以，父进程要记下每个子进程的ID，而子进程只需要调用getppid()就可以拿到父进程的ID。由于Python是跨平台的，自然也应该提供一个跨平台的多进程支持。multiprocessing模块就是跨平台版本的多进程模块。

multiprocessing模块提供了一个Process类来代表一个进程对象。

创建子进程时，只需要传入一个执行函数和函数的参数，创建一个Process实例，用start()方法启动，这样创建进程比fork()还要简单。join()方法可以等待子进程结束后再继续往下运行，通常用于进程间的同步。

         multiprocessing模块提供了一个Pool进程池的方式批量创建子进程。

from multiprocessing import Pool

import os

import time

def long\_time\_task(name):

    print('运行任务编号 %s (线程号: %s)...' % (name, os.getpid()))

    start = time.time()

    time.sleep(5)

    end = time.time()

    print('任务编号 %s 运行时间 %0.2f 秒.' % (name, (end - start)))

if \_\_name\_\_ == '\_\_main\_\_':

    print('父进程号 %s.' % os.getpid())

    p = Pool(5)  # 设置线程池的最大容量

    for i in range(10):

        #传入一个执行函数和函数的参数

        p.apply\_async(long\_time\_task, args=(i,))

    print('等待所有的进程运行 ...')

    # 调用join()之前必须先调用close()，

    # 调用close()之后就不能继续添加新的Process了

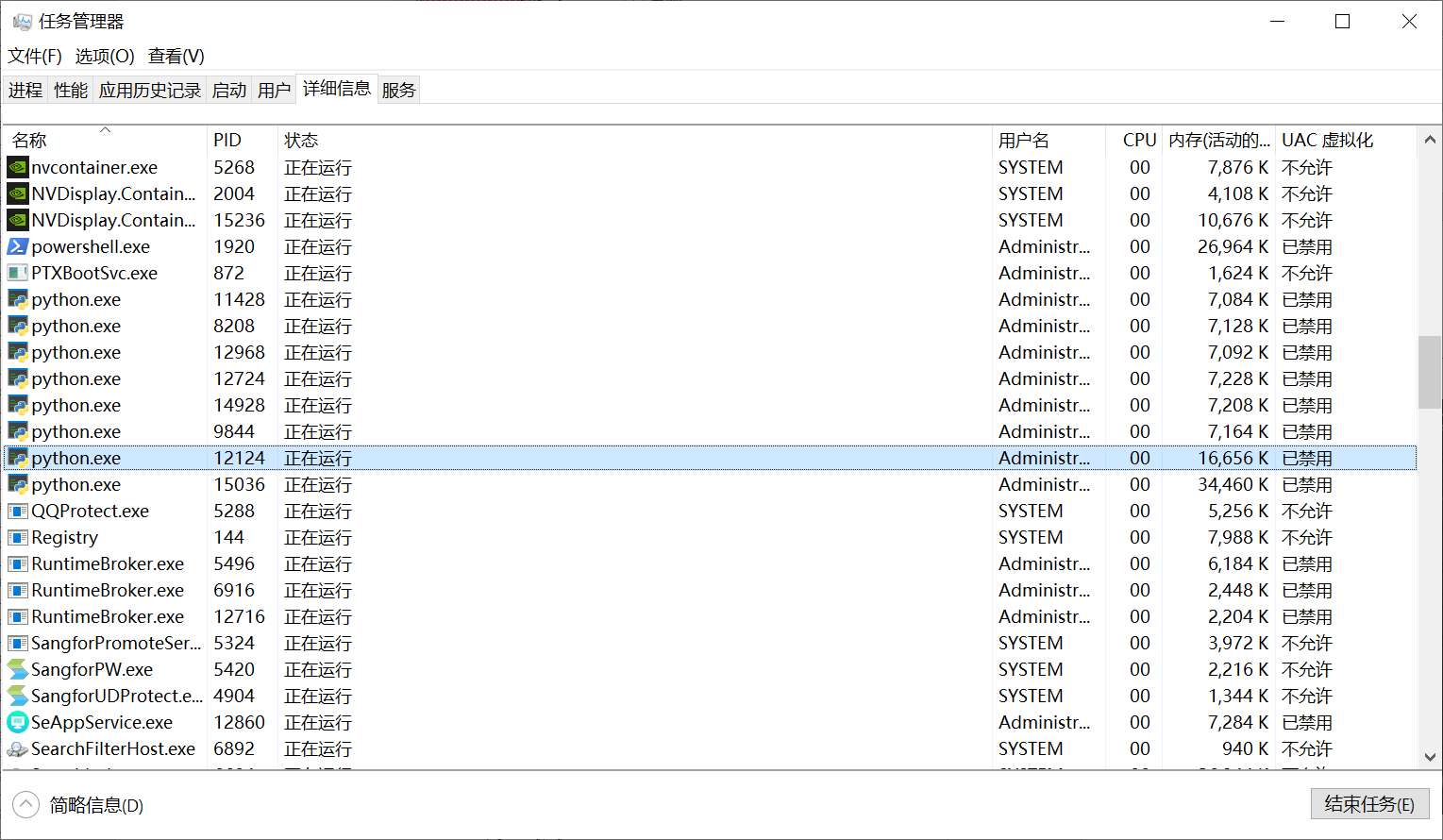
    p.close()

    #join()方法会等待所有子进程执行完毕

    p.join()

print('所有的线程运行完毕.')

执行结果：



父进程号 11428.

等待所有的线程运行 ...

运行任务编号 0 (线程号: 8208)...

运行任务编号 1 (线程号: 12968)...

运行任务编号 2 (线程号: 12724)...

运行任务编号 3 (线程号: 14928)...

运行任务编号 4 (线程号: 9844)...

任务编号 0 运行时间 5.00 秒.

运行任务编号 5 (线程号: 8208)...

任务编号 1 运行时间 5.00 秒.

运行任务编号 6 (线程号: 12968)...

任务编号 2 运行时间 5.00 秒.

运行任务编号 7 (线程号: 12724)...

任务编号 3 运行时间 5.00 秒.

运行任务编号 8 (线程号: 14928)...

任务编号 4 运行时间 5.00 秒.

运行任务编号 9 (线程号: 9844)...

任务编号 5 运行时间 5.00 秒.

运行任务编号 10 (线程号: 8208)...

任务编号 6 运行时间 5.00 秒.

运行任务编号 11 (线程号: 12968)...

任务编号 7 运行时间 5.00 秒.

运行任务编号 12 (线程号: 12724)...

任务编号 8 运行时间 5.00 秒.

运行任务编号 13 (线程号: 14928)...

任务编号 9 运行时间 5.00 秒.

运行任务编号 14 (线程号: 9844)...

任务编号 10 运行时间 5.00 秒.

运行任务编号 15 (线程号: 8208)...

任务编号 11 运行时间 5.00 秒.

运行任务编号 16 (线程号: 12968)...

任务编号 12 运行时间 5.00 秒.

运行任务编号 17 (线程号: 12724)...

任务编号 13 运行时间 5.00 秒.

运行任务编号 18 (线程号: 14928)...

任务编号 14 运行时间 5.00 秒.

运行任务编号 19 (线程号: 9844)...

任务编号 15 运行时间 5.00 秒.

任务编号 16 运行时间 5.00 秒.

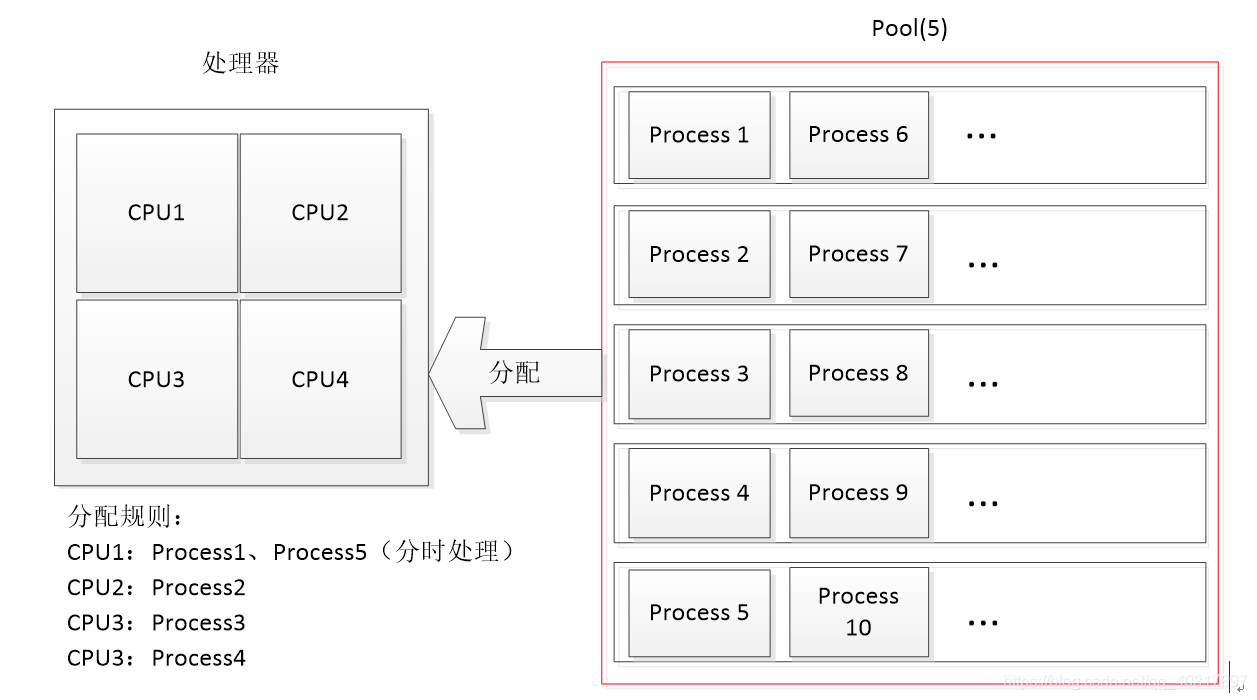
任务编号 17 运行时间 5.00 秒.

任务编号 18 运行时间 5.00 秒.

任务编号 19 运行时间 5.00 秒.

所有的线程运行完毕.

使用Pool对象，Pool中的参数表示调用多少个并行进程进行程序运行，pool的默认容量为CPU的计算核的数量，我们可以这样理解pool池：



直观上的理解为指定了可以容纳最多多少个进程进行并行运算。而p.apply\_async()函数给进程池添加进程任务如上图的Process1, Process2 … ，它的参数包括待运行程序和程序的传入参数。**对Pool对象调用join()方法会等待所有子进程执行完毕，调用join()之前必须先调用close()，调用close()之后就不能继续添加新的Process了。**

请注意输出的结果，process 0，1，2，3，4是立刻执行的，而process  5要等待前面某个process完成后才执行，这是因为Pool的大小为5，在电脑上默认为CPU的核数。因此，最多同时执行5个进程。这是Pool有意设计的限制，并不是操作系统的限制。如果改成：

p = Pool(10)就可以同时跑10个进程。

## 子进程

很多时候，子进程并不是自身，而是一个外部进程。我们创建了子进程后，还需要控制子进程的输入和输出。Subprocess模块可以让我们非常方便地启动一个子进程，然后控制其输入和输出。如果子进程还需要输入，则可以通过communicate()方法输入：

## 进程间的通信

Process之间肯定是需要通信的，操作系统提供了很多机制来实现进程间的通信。Python的multiprocessing模块包装了底层的机制，提供了**Queue、Pipes**等多种方式来交换数据。

我们以Queue为例，在父进程中创建两个子进程，一个往Queue里写数据，一个从Queue里读数据：

from multiprocessing import Process, Queue

import os, time, random

*# 写数据进程执行的代码:*

def write(q):

print('Process to write: %s' % os.getpid())

for value in ['A', 'B', 'C']:

print('Put %s to queue...' % value)

q.put(value)

time.sleep(random.random())

*# 读数据进程执行的代码:*

def read(q):

print('Process to read: %s' % os.getpid())

while True:

value = q.get(True)

print('Get %s from queue.' % value)

if \_\_name\_\_=='\_\_main\_\_':

*# 父进程创建Queue，并传给各个子进程：*

q = Queue()

pw = Process(target=write, args=(q,))

pr = Process(target=read, args=(q,))

*# 启动子进程pw，写入:*

pw.start()

*# 启动子进程pr，读取:*

pr.start()

*# 等待pw结束:*

pw.join()

*# pr进程里是死循环，无法等待其结束，只能强行终止:*

pr.terminate()

运行结果：

Process to write: 15728

Put A to queue...

Process to read: 15748

Get A from queue.

Put B to queue...

Get B from queue.

Put C to queue...

Get C from queue.

Pipes如何使用？

## 多线程

多任务可以由多进程完成，也可以由一个进程内的多线程完成。一个进程至少有一个线程，由于线程是操作系统直接支持的执行单元，因此，高级语言通常都内置多线程的支持，Python也不例外，并且，Python的线程是真正的Posix Thread，而不是模拟出来的线程。**Python的标准库提供了两个模块：\_thread和threading，\_thread是低级模块，threading是高级模块，对\_thread进行了封装。绝大多数情况下，我们只需要使用threading这个高级模块。启动一个线程就是把一个函数传入并创建Thread实例，**

**如:threading.Thread(target=函数名,args=(参数,)，name=“ThreadName” )**

然后调用**start()**开始执行：

import time, threading

# 新线程执行的代码:

def loop():

    print('thread %s is running...' % threading.current\_thread().name)

    n = 0

    while n < 5:

        n = n + 1

        print('thread %s >>> %s' % (threading.current\_thread().name, n))

        time.sleep(1)

    print('thread %s ended.' % threading.current\_thread().name)

#主程序

print('thread %s is running...' % threading.current\_thread().name)

t = threading.Thread(target=loop, name='LoopThread')

t.start()

t.join()

print('thread %s ended.' % threading.current\_thread().name)

执行结果：

thread MainThread is running...

thread LoopThread is running...

thread LoopThread >>> 1

thread LoopThread >>> 2

thread LoopThread >>> 3

thread LoopThread >>> 4

thread LoopThread >>> 5

thread LoopThread ended.

thread MainThread ended.

由于任何进程默认就会启动一个线程，**我们把该线程称为主线程**，主线程又可以启动新的线程，Python的threading模块有个current\_thread()函数，它永远返回当前线程的实例。**主线程实例的名字叫MainThread，子线程的名字在创建时指定**，我们用LoopThread命名子线程。名字仅仅在打印时用来显示，完全没有其他意义，**如果不起名字Python就自动给线程命名为Thread-1，Thread-2……。**在上述程序中我们的主线程是整个程序，子线程执行loop()函数，输出的LoopThread是子线程的名字，输出的1,2,3,… 不是线程的数量。

## Lock：

**多线程和多进程最大的不同在于，多进程中，同一个变量，各自有一份拷贝存在于每个进程中，互不影响，而多线程中，所有变量都由所有线程共享，所以，任何一个变量都可以被任何一个线程修改，因此，线程之间共享数据最大的危险在于多个线程同时改一个变量，把内容给改乱了。**（对于变量：进程à复制；线程à共享）

对于多线程将程序变量改乱了的例子：

import time, threading

*# 假定这是你的银行存款:*

balance = 0

def change\_it(n):

*# 先存后取，结果应该为0:*

global balance

balance = balance + n

balance = balance - n

def run\_thread(n):

for i in range(100000):

change\_it(n)

t1 = threading.Thread(target=run\_thread, args=(5,))

t2 = threading.Thread(target=run\_thread, args=(8,))

t1.start()

t2.start()

t1.join()

t2.join()

print(balance)

我们定义了一个共享变量balance，初始值为0，并且启动两个线程，先存后取，理论上结果应该为0，但是，由于线程的调度是由操作系统决定的，当t1、t2交替执行时，只要循环次数足够多，balance的结果就不一定是0了。原因是因为高级语言的一条语句在CPU执行时是若干条语句，即使一个简单的计算：

Balance = Balance + n

都需要分两步计算：

1. 计算临时变量temp = Balance + n
2. 将临时变量temp赋值给Balance

**当两个线程正常顺序工作时，并不会造成冲突，但是，如果两个线程中的CPU执行语句不是按照线程顺序执行时有可能会造成冲突：**

初始值 balance = 0

t1: x1 = balance + 5  # x1 = 0 + 5 = 5

t2: x2 = balance + 8  # x2 = 0 + 8 = 8

t2: balance = x2      # balance = 8

t1: balance = x1      # balance = 5

t1: x1 = balance - 5  # x1 = 5 - 5 = 0

t1: balance = x1      # balance = 0

t2: x2 = balance - 8  # x2 = 0 - 8 = -8

t2: balance = x2   # balance = -8

结果 balance = -8

究其原因，是因为修改balance需要多条语句，而执行这几条语句时，线程可能中断，从而导致多个线程把同一个对象的内容改乱了。

两个线程同时一存一取，就可能导致余额不对，你肯定不希望你的银行存款莫名其妙地变成了负数，所以，我们必须确保一个线程在修改balance的时候，别的线程一定不能改。

如果我们要确保balance计算正确，就要给change\_it()上一把锁，当某个线程开始执行change\_it()时，我们说，该线程因为获得了锁，因此其他线程不能同时执行change\_it()，只能等待，直到锁被释放后，获得该锁以后才能改。由于锁只有一个，无论多少线程，同一时刻最多只有一个线程持有该锁，所以，不会造成修改的冲突。创建一个锁就是通过threading.Lock()来实现：

balance = 0

lock = threading.Lock()

def run\_thread(n):

for i in range(100000):

*# 先要获取锁:*

lock.acquire()

try:

*# 放心地改吧:*

change\_it(n)

finally:

*# 改完了一定要释放锁:*

lock.release()

当多个线程同时执行lock.acquire()时，只有一个线程能成功地获取锁，然后继续执行代码，其他线程就继续等待直到获得锁为止。

获得锁的线程用完后一定要释放锁，否则那些苦苦等待锁的线程将永远等待下去，成为死线程。所以我们用try...finally来确保锁一定会被释放。

锁的好处就是确保了某段关键代码只能由一个线程从头到尾完整地执行，坏处当然也很多，首先是**阻止了多线程并发执行**，包含锁的某段代码实际上只能以单线程模式执行，效率就大大地下降了。其次，由于可以存在多个锁，不同的线程持有不同的锁，并试图获取对方持有的锁时，可能会造成死锁，导致多个线程全部挂起，既不能执行，也无法结束，只能靠操作系统强制终止。

## 多核CPU

如果你不幸拥有一个多核CPU，你肯定在想，多核应该可以同时执行多个线程。如果写一个死循环的话，会出现什么情况呢？打开Mac OS X的Activity Monitor，或者Windows的Task Manager，都可以监控某个进程的CPU使用率。我们可以监控到一个死循环线程会100%占用一个CPU。如果有两个死循环线程，在多核CPU中，可以监控到会占用200%的CPU，也就是占用两个CPU核心。要想把N核CPU的核心全部跑满，就必须启动N个死循环线程。试试用Python写个死循环：

import threading, multiprocessing

def loop():

x = 0

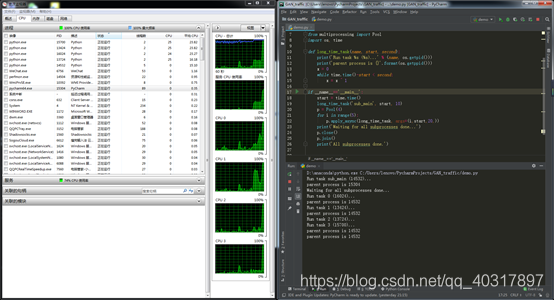
while True:

x = x ^ 1

for i in range(multiprocessing.cpu\_count()):

t = threading.Thread(target=loop)

t.start()



程序：

import threading, multiprocessing

'''

#一个线程运行

def loop():

    x = 0

    while True:

        x = x ^ 1

for i in range(multiprocessing.cpu\_count()):

    t = threading.Thread(target=loop)

t.start()

'''

import os

import time

#多个线程运行

def long\_time\_task(name,second):

    print(' %s 任务 (线程号: %s)...' % (name, os.getpid()))

    print(' %s 任务 (父进程号) %s ...' %  (name,os.getppid()))

    start = time.time()

    x = 0

    while time.time() -start < second:

        x = x ^ 1

    end = time.time()

    print(' %s 任务 运行时间 %0.2f 秒.' % (name, (end - start)))

    print(' %s 任务...结束' % (name))

if \_\_name\_\_ == '\_\_main\_\_':

    start = time.time()

    long\_time\_task('sub\_main',5)

    p = multiprocessing.Pool(4)  # 设置线程池的最大容量

    print ('CPU 数量:' ,multiprocessing.cpu\_count())

    for i in range(multiprocessing.cpu\_count()):

        #传入一个执行函数和函数的参数

        p.apply\_async(long\_time\_task, args=(i,20))

    print('等待所有的线程运行 ...')

    # 调用join()之前必须先调用close()，

    # 调用close()之后就不能继续添加新的Process了

    p.close()

    #join()方法会等待所有子进程执行完毕

    p.join()

    print('所有的线程运行完毕.')

    end = time.time()

print('主程序运行时间 %0.2f 秒.' % (end - start))

进程：

sub\_main 任务 (线程号: 15272)...

sub\_main 任务 (父进程号) 1104 ...

sub\_main 任务 运行时间 5.00 秒.

sub\_main 任务...结束

CPU 数量: 12

等待所有的线程运行 ...

0 任务 (线程号: 13628)...

1 任务 (线程号: 9664)...

0 任务 (父进程号) 15272 ...

1 任务 (父进程号) 15272 ...

2 任务 (线程号: 9312)...

3 任务 (线程号: 12684)...

2 任务 (父进程号) 15272 ...

3 任务 (父进程号) 15272 ...

4 任务 (线程号: 6780)...

5 任务 (线程号: 14112)...

4 任务 (父进程号) 15272 ...

5 任务 (父进程号) 15272 ...

6 任务 (线程号: 10928)...

7 任务 (线程号: 2876)...

6 任务 (父进程号) 15272 ...

7 任务 (父进程号) 15272 ...

8 任务 (线程号: 9440)...

8 任务 (父进程号) 15272 ...

9 任务 (线程号: 12324)...

9 任务 (父进程号) 15272 ...

10 任务 (线程号: 7908)...

10 任务 (父进程号) 15272 ...

11 任务 (线程号: 2068)...

11 任务 (父进程号) 15272 ...

0 任务 运行时间 20.00 秒.

0 任务...结束

1 任务 运行时间 20.00 秒.

1 任务...结束

2 任务 运行时间 20.00 秒.

2 任务...结束

3 任务 运行时间 20.00 秒.

3 任务...结束

4 任务 运行时间 20.00 秒.

4 任务...结束

5 任务 运行时间 20.00 秒.

5 任务...结束

6 任务 运行时间 20.00 秒.

6 任务...结束

7 任务 运行时间 20.00 秒.

7 任务...结束

8 任务 运行时间 20.00 秒.

8 任务...结束

9 任务 运行时间 20.00 秒.

9 任务...结束

10 任务 运行时间 20.00 秒.

10 任务...结束

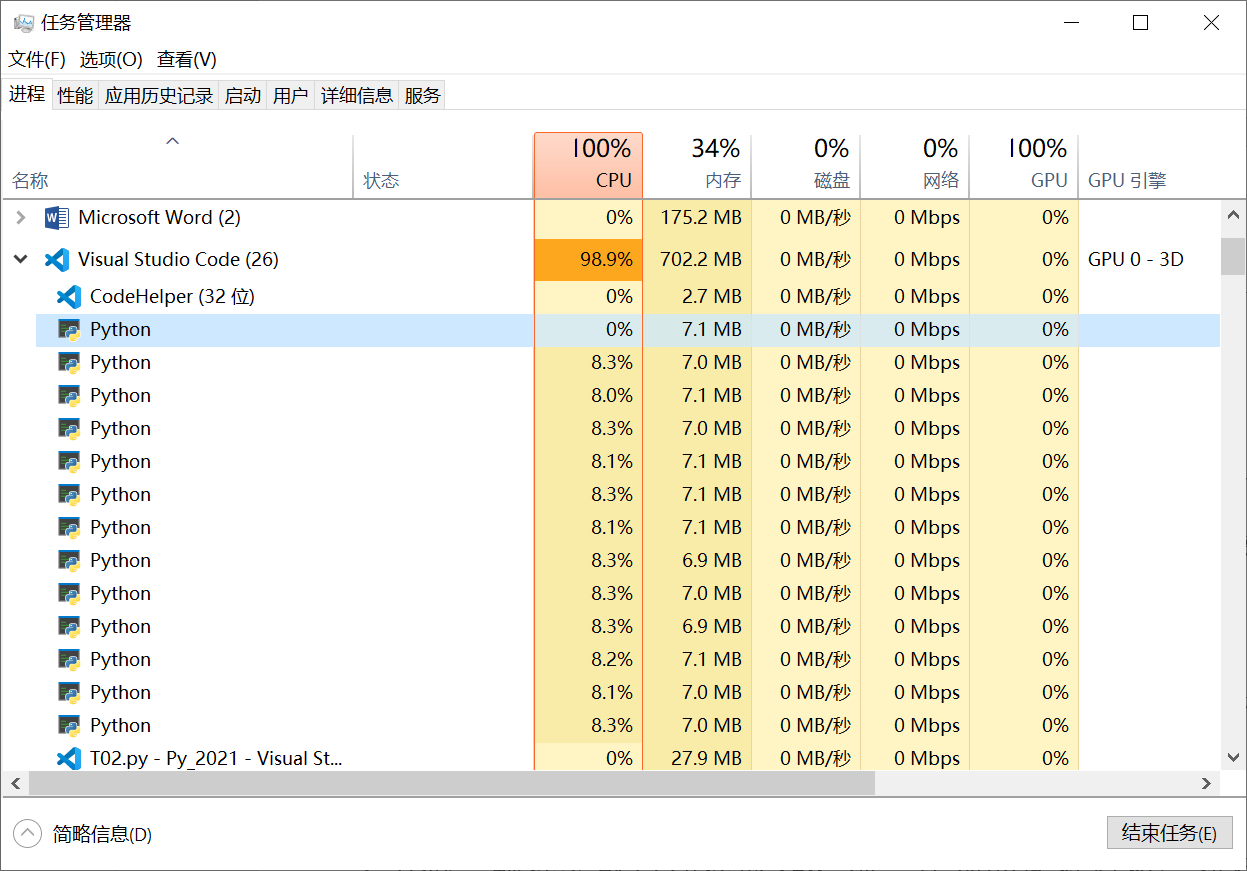
11 任务 运行时间 20.00 秒.

11 任务...结束

所有的线程运行完毕.

主程序运行时间 25.28 秒.

资源监视器：



启动与CPU核心数量相同的N个线程，在4核CPU上可以监控到CPU占用率仅有100%，也就是**仅使用了一核**。但是用C、C++或Java来改写相同的死循环，直接可以把全部核心跑满，4核就跑到400%，8核就跑到800%，为什么Python不行呢？因为Python的线程虽然是真正的线程，但解释器执行代码时，有一个GIL锁：Global Interpreter Lock，任何Python线程执行前，必须先获得GIL锁，然后，每执行100条字节码，解释器就自动释放GIL锁，让别的线程有机会执行。这个GIL全局锁实际上把所有线程的执行代码都给上了锁，所以，多线程在Python中只能交替执行，即使100个线程跑在100核CPU上，也只能用到1个核。GIL是Python解释器设计的历史遗留问题，通常我们用的解释器是官方实现的CPython，要真正利用多核，除非重写一个不带GIL的解释器。所以，在Python中，可以使用多线程，但**不要指望能有效利用多核**。如果一定要通过多线程利用多核，那只能通过C扩展来实现，不过这样就失去了Python简单易用的特点。不过，也不用过于担心，P**ython虽然不能利用多线程实现多核任务，但可以通过多进程实现多核任务。**多个Python进程有各自独立的GIL锁，互不影响。

END

## 进程VS.线程

我们介绍了多进程和多线程，这是实现多任务最常用的两种方式。现在，我们来讨论一下这两种方式的优缺点。

首先，要实现多任务，通常我们会设计Master-Worker模式，Master负责分配任务，Worker负责执行任务，因此，多任务环境下，通常是一个Master，多个Worker。

如果用多进程实现Master-Worker，主进程就是Master，其他进程就是Worker。

如果用多线程实现Master-Worker，主线程就是Master，其他线程就是Worker。

多进程模式最大的优点就是稳定性高，因为一个子进程崩溃了，不会影响主进程和其他子进程。（当然主进程挂了所有进程就全挂了，但是Master进程只负责分配任务，挂掉的概率低）著名的Apache最早就是采用多进程模式。

多进程模式的缺点是创建进程的代价大，在Unix/Linux系统下，用fork调用还行，在Windows下创建进程开销巨大。另外，操作系统能同时运行的进程数也是有限的，在内存和CPU的限制下，如果有几千个进程同时运行，操作系统连调度都会成问题。

多线程模式通常比多进程快一点，但是也快不到哪去，而且，多线程模式致命的缺点就是任何一个线程挂掉都可能直接造成整个进程崩溃，因为所有线程共享进程的内存。在Windows上，如果一个线程执行的代码出了问题，你经常可以看到这样的提示：“该程序执行了非法操作，即将关闭”，其实往往是某个线程出了问题，但是操作系统会强制结束整个进程。

在Windows下，多线程的效率比多进程要高，所以微软的IIS服务器默认采用多线程模式。由于多线程存在稳定性的问题，IIS的稳定性就不如Apache。为了缓解这个问题，IIS和Apache现在又有多进程+多线程的混合模式，真是把问题越搞越复杂。

**线程切换：**

无论是多进程还是多线程，只要数量一多，效率肯定上不去，为什么呢？我们打个比方，假设你不幸正在准备中考，每天晚上需要做语文、数学、英语、物理、化学这5科的作业，每项作业耗时1小时。

如果你先花1小时做语文作业，做完了，再花1小时做数学作业，这样，依次全部做完，一共花5小时，这种方式称为单任务模型，或者批处理任务模型。

假设你打算切换到多任务模型，可以先做1分钟语文，再切换到数学作业，做1分钟，再切换到英语，以此类推，只要切换速度足够快，这种方式就和单核CPU执行多任务是一样的了，以幼儿园小朋友的眼光来看，你就正在同时写5科作业。

但是，切换作业是有代价的，比如从语文切到数学，要先收拾桌子上的语文书本、钢笔（这叫保存现场），然后，打开数学课本、找出圆规直尺（这叫准备新环境），才能开始做数学作业。操作系统在切换进程或者线程时也是一样的，它需要先保存当前执行的现场环境（CPU寄存器状态、内存页等），然后，把新任务的执行环境准备好（恢复上次的寄存器状态，切换内存页等），才能开始执行。这个切换过程虽然很快，但是也需要耗费时间。如果有几千个任务同时进行，操作系统可能就主要忙着切换任务，根本没有多少时间去执行任务了，这种情况最常见的就是硬盘狂响，点窗口无反应，系统处于假死状态。

所以，多任务一旦多到一个限度，就会消耗掉系统所有的资源，结果效率急剧下降，所有任务都做不好。

## 计算密集型 vs. IO密集型：

是否采用多任务的第二个考虑是任务的类型。我们可以把任务分为计算密集型和IO密集型。

计算密集型任务的特点是要进行大量的计算，消耗CPU资源，比如计算圆周率、对视频进行高清解码等等，全靠CPU的运算能力。这种计算密集型任务虽然也可以用多任务完成，但是任务越多，花在任务切换的时间就越多，CPU执行任务的效率就越低，所以，要最高效地利用CPU，计算密集型任务同时进行的数量应当等于CPU的核心数。

计算密集型任务由于主要消耗CPU资源，因此，代码运行效率至关重要。Python这样的脚本语言运行效率很低，完全不适合计算密集型任务。对于计算密集型任务，最好用C语言编写。

第二种任务的类型是IO密集型，涉及到网络、磁盘IO的任务都是IO密集型任务，这类任务的特点是CPU消耗很少，任务的大部分时间都在等待IO操作完成（因为IO的速度远远低于CPU和内存的速度）。对于IO密集型任务，任务越多，CPU效率越高，但也有一个限度。常见的大部分任务都是IO密集型任务，比如Web应用。

IO密集型任务执行期间，99%的时间都花在IO上，花在CPU上的时间很少，因此，用运行速度极快的C语言替换用Python这样运行速度极低的脚本语言，完全无法提升运行效率。对于IO密集型任务，最合适的语言就是开发效率最高（代码量最少）的语言，脚本语言是首选，C语言最差。

## 异步IO：

考虑到CPU和IO之间巨大的速度差异，一个任务在执行的过程中大部分时间都在等待IO操作，单进程单线程模型会导致别的任务无法并行执行，因此，我们才需要多进程模型或者多线程模型来支持多任务并发执行。

现代操作系统对IO操作已经做了巨大的改进，最大的特点就是支持异步IO。如果充分利用操作系统提供的异步IO支持，就可以用单进程单线程模型来执行多任务，这种全新的模型称为事件驱动模型，Nginx就是支持异步IO的Web服务器，它在单核CPU上采用单进程模型就可以高效地支持多任务。在多核CPU上，可以运行多个进程（数量与CPU核心数相同），充分利用多核CPU。由于系统总的进程数量十分有限，因此操作系统调度非常高效。用异步IO编程模型来实现多任务是一个主要的趋势。

对应到Python语言，单线程的异步编程模型称为协程，有了协程的支持，就可以基于事件驱动编写高效的多任务程序。

## 小结

多线程编程，模型复杂，容易发生冲突，必须用锁加以隔离，同时，又要小心死锁的发生。Python解释器由于设计时有GIL全局锁，导致了多线程无法利用多核。多线程的并发在Python中就是一个美丽的梦。

### 分布式进程

在Thread和Process中，应当优选Process，因为Process更稳定，而且，Process可以分布到多台机器上，而Thread最多只能分布到同一台机器的多个CPU上。

Python的multiprocessing模块不但支持多进程，其中managers子模块还支持把多进程分布到多台机器上。一个服务进程可以作为调度者，将任务分布到其他多个进程中，依靠网络通信。由于managers模块封装很好，不必了解网络通信的细节，就可以很容易地编写分布式多进程程序。

举个例子：如果我们已经有一个通过Queue通信的多进程程序在同一台机器上运行，现在，由于处理任务的进程任务繁重，希望把发送任务的进程和处理任务的进程分布到两台机器上。怎么用分布式进程实现？

原有的Queue可以继续使用，但是，通过managers模块把Queue通过网络暴露出去，就可以让其他机器的进程访问Queue了。

我们先看服务进程，服务进程负责启动Queue，把Queue注册到网络上，然后往Queue里面写入任务：

*# task\_master.py*

import random, time, queue

from multiprocessing.managers import BaseManager

*# 发送任务的队列:*

task\_queue = queue.Queue()

*# 接收结果的队列:*

result\_queue = queue.Queue()

*# 从BaseManager继承的QueueManager:*

class QueueManager(BaseManager):

pass

*# 把两个Queue都注册到网络上, callable参数关联了Queue对象:*

QueueManager.register('get\_task\_queue', callable=lambda: task\_queue)

QueueManager.register('get\_result\_queue', callable=lambda: result\_queue)

*# 绑定端口5000, 设置验证码'abc':*

manager = QueueManager(address=('', 5000), authkey=b'abc')

*# 启动Queue:*

manager.start()

*# 获得通过网络访问的Queue对象:*

task = manager.get\_task\_queue()

result = manager.get\_result\_queue()

*# 放几个任务进去:*

for i in range(10):

n = random.randint(0, 10000)

print('Put task %d...' % n)

task.put(n)

*# 从result队列读取结果:*

print('Try get results...')

for i in range(10):

r = result.get(timeout=10)

print('Result: %s' % r)

*# 关闭:*

manager.shutdown()

print('master exit.')

请注意，当我们在一台机器上写多进程程序时，创建的Queue可以直接拿来用，但是，在分布式多进程环境下，添加任务到Queue不可以直接对原始的task\_queue进行操作，那样就绕过了QueueManager的封装，必须通过manager.get\_task\_queue()获得的Queue接口添加。然后，在另一台机器上启动任务进程（本机上启动也可以）：

*# task\_worker.py*

import time, sys, queue

from multiprocessing.managers import BaseManager

*# 创建类似的QueueManager:*

class QueueManager(BaseManager):

pass

*# 由于这个QueueManager只从网络上获取Queue，所以注册时只提供名字:*

QueueManager.register('get\_task\_queue')

QueueManager.register('get\_result\_queue')

*# 连接到服务器，也就是运行task\_master.py的机器:*

server\_addr = '127.0.0.1'

print('Connect to server %s...' % server\_addr)

*# 端口和验证码注意保持与task\_master.py设置的完全一致:*

m = QueueManager(address=(server\_addr, 5000), authkey=b'abc')

*# 从网络连接:*

m.connect()

*# 获取Queue的对象:*

task = m.get\_task\_queue()

result = m.get\_result\_queue()

*# 从task队列取任务,并把结果写入result队列:*

for i in range(10):

try:

n = task.get(timeout=1)

print('run task %d \* %d...' % (n, n))

r = '%d \* %d = %d' % (n, n, n\*n)

time.sleep(1)

result.put(r)

except Queue.Empty:

print('task queue is empty.')

*# 处理结束:*

print('worker exit.')

任务进程要通过网络连接到服务进程，所以要指定服务进程的IP。

现在，可以试试分布式进程的工作效果了。先启动task\_master.py服务进程：

$ python3 task\_master.py

Put task 3411...

Put task 1605...

Put task 1398...

Put task 4729...

Put task 5300...

Put task 7471...

Put task 68...

Put task 4219...

Put task 339...

Put task 7866...

Try get results...

task\_master.py进程发送完任务后，开始等待result队列的结果。现在启动task\_worker.py进程：

$ python3 task\_worker.py

Connect to server 127.0.0.1...

run task 3411 \* 3411...

run task 1605 \* 1605...

run task 1398 \* 1398...

run task 4729 \* 4729...

run task 5300 \* 5300...

run task 7471 \* 7471...

run task 68 \* 68...

run task 4219 \* 4219...

run task 339 \* 339...

run task 7866 \* 7866...

worker exit.

task\_worker.py进程结束，在task\_master.py进程中会继续打印出结果：

Result: 3411 \* 3411 = 11634921

Result: 1605 \* 1605 = 2576025

Result: 1398 \* 1398 = 1954404

Result: 4729 \* 4729 = 22363441

Result: 5300 \* 5300 = 28090000

Result: 7471 \* 7471 = 55815841

Result: 68 \* 68 = 4624

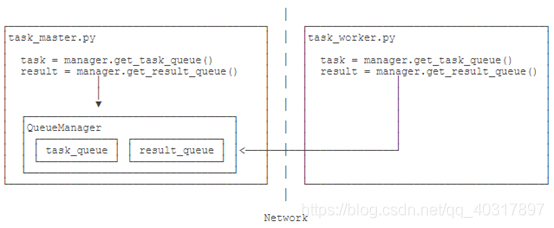
Result: 4219 \* 4219 = 17799961

Result: 339 \* 339 = 114921

Result: 7866 \* 7866 = 61873956

这个简单的Master/Worker模型有什么用？其实这就是一个简单但真正的分布式计算，把代码稍加改造，启动多个worker，就可以把任务分布到几台甚至几十台机器上，比如把计算n\*n的代码换成发送邮件，就实现了邮件队列的异步发送。

Queue对象存储在哪？注意到task\_worker.py中根本没有创建Queue的代码，所以，Queue对象存储在task\_master.py进程中：



而Queue之所以能通过网络访问，就是通过QueueManager实现的。由于QueueManager管理的不止一个Queue，所以，要给每个Queue的网络调用接口起个名字，比如get\_task\_queue。

authkey有什么用？这是为了保证两台机器正常通信，不被其他机器恶意干扰。如果task\_worker.py的authkey和task\_master.py的authkey不一致，肯定连接不上。

## 小结：

Python的分布式进程接口简单，封装良好，适合需要把繁重任务分布到多台机器的环境下。

注意Queue的作用是用来传递任务和接收结果，每个任务的描述数据量要尽量小。比如发送一个处理日志文件的任务，就不要发送几百兆的日志文件本身，而是发送日志文件存放的完整路径，由Worker进程再去共享的磁盘上读取文件。