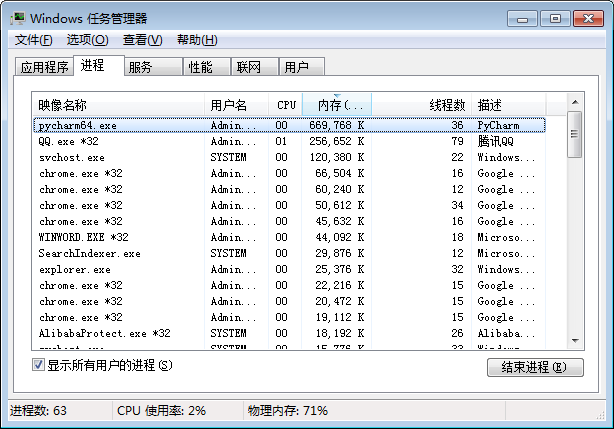
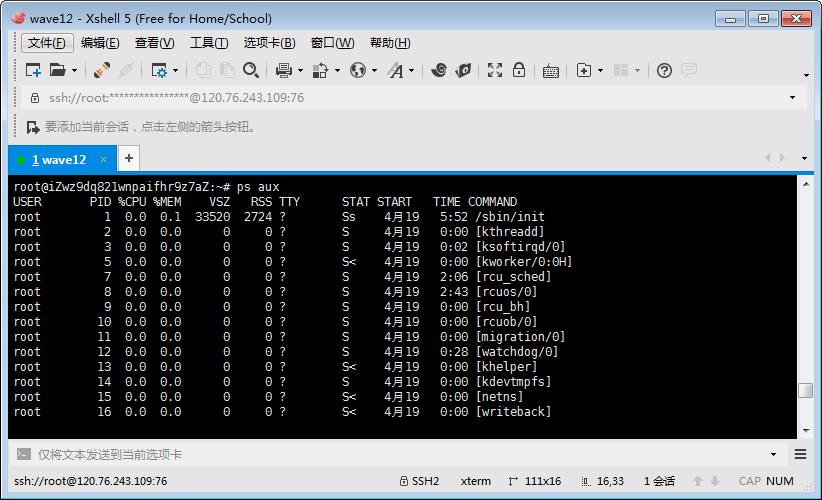
# 多线程和多进程编程

一个计算机可以运行多个进程，一个进程里可以运行多个线程，至少有一个主线程。



在linux系统下查看进程列表：ps aux



## 多线程编程

python中，有两个标准模块thread和threading可以实现多线程，thread 模块在3.0里已被废弃，不建议使用。这里使用更高级的threading模块。

Python中使用线程有两种方式：线程函数或者用线程类来包装线程对象。

### 利用线程函数

thread1.py

#!/usr/bin/python3

**import** threading  
**import** time  
  
**def** fun(counter):  
 **while** counter:  
 print((**"%s:%d"**) % (threading.current\_thread().name,counter))  
 *#time.sleep(3)* counter = counter - 1  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 print((**"%s"**) % (threading.current\_thread().name))  
 t = threading.Thread(target=fun, args=(100,))  
 t2 = threading.Thread(target=fun, args=(100,))  
 t.start()  
 t2.start()

### 利用线程类

thread2.py

**import** threading  
**import** time  
  
  
**class** MyThread(threading.Thread):  
 **def** \_\_init\_\_(self, n):  
 super(MyThread, self).\_\_init\_\_() *# 重构run函数必须要写* self.name = n *#使用自己的线程名称* **def** run(self):  
 counter = 100  
 **while** counter:  
 print((**"%s:%d"**) % (threading.current\_thread().name, counter))  
 counter = counter - 1  
 print((**"%s exit!"**) % (threading.current\_thread().name))  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 print((**"%s"**) % (threading.current\_thread().name))  
 t1 = MyThread(**"t1"**)  
 t2 = MyThread(**"t2"**)  
 t1.start()  
 t2.start()  
 print((**"%s exit!"**) % (threading.current\_thread().name))

### 设置守护线程

setDaemon方法设置此线程是否为守护线程。设置为true即被主线程守护回收。默认False不回收。设为True相当于向主线程中注册守护，主线程结束时会将其一并回收。这样做的意义在于：避免子线程无限死循环，导致退不出程序，即出现孤儿进程。

setDaemon方法需要在 start 方法前调用。

join方法：等待线程正常结束。

**import** threading  
**import** time  
  
  
**class** MyThread(threading.Thread):  
 **def** \_\_init\_\_(self, n):  
 super(MyThread, self).\_\_init\_\_() *# 重构run函数必须要写* self.name = n *#使用自己的线程名称* **def** run(self):  
 counter = 100  
 **while** counter:  
 print((**"%s:%d"**) % (threading.current\_thread().name, counter))  
 counter = counter - 1  
 print((**"%s exit!"**) % (threading.current\_thread().name))  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 print((**"%s"**) % (threading.current\_thread().name))  
 t1 = MyThread(**"t1"**)  
 t2 = MyThread(**"t2"**)  
 t1.setDaemon(**True**)  
 t2.setDaemon(**True**)  
 t1.start()  
 t2.start()  
 t1.join()  
 t2.join()  
 print((**"%s exit!"**) % (threading.current\_thread().name))

t1,t2都设定为守护线程，主程序退出时回强制回收t1,t2资源，这里加上join是等待t1,t2正常执行完任务。

super(MyThread, self).\_\_init\_\_() 也可以写成super().\_\_init\_\_()。前者是python2.0的写法，后者是python3.0的写法。这里，3.0兼容2.0的写法，即两种写法均可。

### 线程同步

如果多个线程共同对某个数据修改，则可能出现不可预料的结果，为了保证数据的正确性，需要对多个线程进行同步。

使用 Thread 对象的 Lock 和 Rlock 可以实现简单的线程同步，这两个对象都有 acquire 方法和 release 方法，对于那些需要每次只允许一个线程操作的数据，可以将其操作放到 acquire 和 release 方法之间。

**import** threading  
**import** time  
  
gl\_num = 0  
  
lock = threading.RLock()  
  
**def** show():  
 **global** gl\_num  
 lock.acquire()  
 gl\_num +=1  
 time.sleep(1)  
 print((**'%s:%d'**) %(threading.current\_thread().name, gl\_num))  
 lock.release()  
  
t = []  
**for** i **in** range(20):  
 t.append(threading.Thread(target=show))  
**for** i **in** range(20):  
 t[i].start()  
  
print(**'main thread stop'**)

threading.Lock() 加载线程的锁对象，是一个基本的锁对象，一次只能一个锁定，其余锁请求，需等待锁释放后才能获取

threading.RLock() 多重锁，在同一线程中可用被多次acquire。如果使用RLock，那么acquire和release必须成对出现，   
调用了n次acquire锁请求，则必须调用n次的release才能在线程中释放锁对象

其他线程同步对象： Condition, Semaphore, Event，思想与Lock类似。

### GIL全局解释器锁

在非python环境中，单核情况下，同时只能有一个任务执行。多核时可以支持多个线程同时执行。但是在python中，无论有多少核，同时只能执行一个线程。究其原因，这就是由于GIL的存在导致的。

GIL的全称是Global Interpreter Lock(全局解释器锁)，来源是python设计之初的考虑，为了数据安全所做的决定。某个线程想要执行，必须先拿到GIL，我们可以把GIL看作是“通行证”，并且在一个python进程中，GIL只有一个。拿不到通行证的线程，就不允许进入CPU执行。

Python多线程很适合用在IO密集型任务中。I/O密集型执行期间大部分是时间都用在I/O上，如读写文件I/O、数据库I/O，较少时间用在CPU计算上。因此该应用场景可以使用Python多线程，当一个任务阻塞在IO操作上时，我们可以立即切换执行其他线程上执行其他IO操作请求。

## 多进程编程

python下想要充分利用多核CPU，就用多进程。因为每个进程有各自独立的GIL，互不干扰，这样就可以真正意义上的并行执行，在python中，多进程的执行效率优于多线程(仅仅针对多核CPU而言)。**对于计算密集型任务，要实现并行执行，更应该使用Python多进程。**

### 使用multiprocessing

*#!/bin/env python***from** multiprocessing **import** Process  
**import** os  
**import** time  
**import** random  
  
**class** Piao(Process):  
 **def** \_\_init\_\_(self,name):  
 self.name=name  
 super().\_\_init\_\_()  
  
 **def** run(self):  
 print(**'%s is piaoing'** % self.name)  
 time.sleep(random.randrange(1,3))  
 print(**'%s is piao end'** % self.name)  
  
**def** run\_proc(name):  
 time.sleep(3)  
 print(**'Run child process %s (%s)...'** % (name, os.getpid()))  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 print(**'Parent process %s.'** % os.getpid())  
 **for** i **in** range(5):  
 p = Process(target=run\_proc, args=(**'test'**,))  
 print(**'Process will start.'**)  
 p.start()  
  
 p = Piao(**'egon'**)  
 p.start()  
  
 print( **'Process end.'**)

### 使用守护进程

调用setDaemon方法。

**from** multiprocessing **import** Process  
**import** time  
**import** random  
**import** os  
  
**class** Piao(Process):  
 **def** \_\_init\_\_(self,name):  
 self.name=name  
 super().\_\_init\_\_()  
  
 **def** run(self):  
 print(**'%s is piaoing'** % self.name)  
 time.sleep(random.randrange(1,3))  
 print(**'%s is piao end'** % self.name)  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 print(**'Parent process %s.'** % os.getpid())  
 p = Piao(**'egon'**)  
 *#p.daemon=True #一定要在p.start()前设置,设置p为守护进程,禁止p创建子进程,并且父进程代码执行结束,p即终止运行* p.start()  
 *#p.join(0.5)* print(**'主进程结束！'**)

### 进程池

使用 multiprocessing.Pool

from multiprocessing import Pool

import os, time, random

def long\_time\_task(name):

print('Run task %s (%s)...' % (name, os.getpid()))

start = time.time()

time.sleep(random.random() \* 3)

end = time.time()

print('Task %s runs %0.2f seconds.' % (name, (end - start)))

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

print('Parent process %s.' % os.getpid())

p = Pool(4)

for i in range(5):

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

print('Waiting for all subprocesses done...')

p.close()

p.join()

print('All subprocesses done.')

apply\_async异步启动进程。Apply则是同步启动进程。

### 多进程同步

多进程写同一个文件

import multiprocessing

import sys

def worker\_with(lock, f):

with lock:

fs = open(f, 'a+')

n = 10

while n > 1:

fs.write("Lockd acquired via with\n")

n -= 1

fs.close()

def worker\_no\_with(lock, f):

lock.acquire()

try:

fs = open(f, 'a+')

n = 10

while n > 1:

fs.write("Lock acquired directly\n")

n -= 1

fs.close()

finally:

lock.release()

if \_\_name\_\_ == "\_\_main\_\_":

lock = multiprocessing.Lock()

f = "file.txt"

w = multiprocessing.Process(target = worker\_with, args=(lock, f))

nw = multiprocessing.Process(target = worker\_no\_with, args=(lock, f))

w.start()

nw.start()

print

### 进程间通信

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。

**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\_\_'**:  
 print(**'Process Main: %s'** % os.getpid())  
 *# 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()  
 print(**'Process Main: exit'**)