**multiprocessing**

Process

Process([group [, target [, name [, args [, kwargs]]]]])

#target表示调用对象，你可以传入方法的名字

#args表示被调用对象的位置参数**元组** ,单个参数是（m,）

#kwargs表示调用对象的字典

#name是别名，相当于给这个进程取一个名字

#group分组，实际上不使用

cpu.count()

#获取当前机器的 CPU 核心数量

active\_children()

#获取目前所有的运行的进程

继承Process类

from multiprocessing import Process

import time

class MyProcess(Process):

def \_\_init\_\_(self, loop):

**Process.\_\_init\_\_(self)**

self.loop = loop

def run(self):

for count in range(self.loop):

time.sleep(1)

print('Pid: ' + str(self.pid) + ' LoopCount: ' + str(count))

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

for i in range(2, 5):

p = MyProcess(i)

p.start()

start()

#启动多个进程

daemon

#设置为True，则父进程结束后子进程自动终止

join()

#所有子进程都执行完再结束

Lock()

#同一时间只能一个进程输出，其他进程等待

#获得锁 Lock(),acquire()

#释放锁 Lock().release()

Semaphore(信号量)

#做到同步和互斥，及控制临界资源数量

Queue

#进程间的通信

**from** multiprocessing **import** Process, Semaphore, Lock, Queue  
**import** time  
**from** random **import** random  
  
buffer=Queue(10)  
empty=Semaphore(2) #缓冲区空余数  
full=Semaphore(0) #缓冲区占用数  
lock=Lock()  
  
**class** Consumer(Process):  
  
 **def** run(self):  
 **global** buffer, empty, full, lock  
 **while True**:  
 full.acquire()  
 lock.acquire()  
 print(**"Consumer get"**,buffer.get())  
 time.sleep(1)  
 lock.release()  
 empty.release()  
  
**class** Producer(Process):  
  
 **def** run(self):  
 **global** buffer, empty, full, lock  
 **while True**:  
 empty.acquire() #占用一个缓冲区位置,缓冲区空余数-1  
 lock.acquire()  
 num=random()  
 print(**"Producer put"**,num)  
 buffer.put(num) #对缓冲区进行操作  
 time.sleep(1)  
 lock.release()  
 full.release() #缓冲区占用数+1  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 p=Producer()  
 c=Consumer()  
 p.daemon=c.daemon=**True** p.start()  
 c.start()  
 p.join()  
 c.join()  
 print(**"Ended!"**)

运行结果

Producer put 0.719213647437

Producer put 0.44287326683

Consumer get 0.719213647437

Consumer get 0.44287326683

Producer put 0.722859424381

Producer put 0.525321338921

Consumer get 0.722859424381

Consumer get 0.525321338921

Pipe(管道)

#默认双向；mutiprocessing.Pipe(duplex=False)创建单向管道

**from** multiprocessing **import** Process, Pipe  
  
**class** Consumer(Process):  
 **def** \_\_init\_\_(self,pipe):  
 Process.\_\_init\_\_(self)  
 self.pipe=pipe  
  
 **def** run(self):  
 self.pipe.send(**"Consumer words"**)  
 print(**"Consumer received: "**,self.pipe.recv())  
  
**class** Producer(Process):  
 **def** \_\_init\_\_(self, pipe):  
 Process.\_\_init\_\_(self)  
 self.pipe = pipe  
  
 **def** run(self):  
 self.pipe.send(**"Producer words"**)  
 print(**"Producer received: "**,self.pipe.recv())  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 pipe=Pipe()  
 p=Producer(pipe[0]) #将管道的两端分别传给两个进程  
 c=Consumer(pipe[1])  
 p.daemon=c.daemon=**True** p.start()  
 c.start()  
 p.join()  
 c.join()  
 print(**"Ended"**)

输出结果：

Producer received: Consumer words

Consumer received: Producer words

Ended

Pool

#进程池

分为阻塞和非阻塞两种方式:非阻塞(Pool().apply\_async())即为添加进程后，不一定非要等到改进程执行完就添加其他进程运行，阻塞(Pool().apply())则相反。

**阻塞**

**from** multiprocessing **import** Pool  
**import** time  
  
**def** fun(x):  
 print(x)  
 time.sleep(1)  
  
pool=Pool(processes=5)  
start=time.time()  
**for** i **in** range(10):  
 pool.apply(fun,(i,))  
  
pool.close()  
pool.join()  
end=time.time()  
print(end-start)

输出结果：

0

1

2

3

4

5

6

7

8

9

10.056557178497314

非阻塞

**from** multiprocessing **import** Pool  
**import** time  
  
**def** fun(x):  
 print(x)  
 time.sleep(1)  
  
pool=Pool(processes=5)  
start=time.time()  
**for** i **in** range(10):  
 pool.apply\_async(fun,(i,))  
  
pool.close()  
pool.join()  
end=time.time()  
print(end-start)

输出结果

0

1

2

3

4

5

6

7

8

9

2.0517051219940186

批量处理还可以使用Pool().map(func,[])

pool=Pool(processes=5)

pool.map(fun,range(10))

多进程写入同一文件，需要使用multiprocessing库的**回调函数功能**。

#把进程需要写入文件的内容作为返回值返回给回调函数，使用回调函数向文件中写入内容

def mycallback(x):

        output.write(x+"\n")

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

        pool=multiprocessing.Pool(processes=12)

        for item in strain\_combination:

                pool.apply\_async(compare2.calculate\_distance,(item,),callback=mycallback)

        pool.close()

        pool.join()