并发编程（线程 进程 协程）

知识预览

- 操作系统

回到顶部

操作系统

一 为什么要有操作系统?

现代计算机系统是由一个或者多个处理器，主存，磁盘，打印机，键盘，鼠标显示器，网络接口以及各种其他输入输出设备组成的复杂系统，每位程序员不可能掌握所有系统实现的细节，并且管理优化这些部件是一件挑战性极强的工作。所以，我们需要为计算机安装一层软件，成为操作系统，任务就是用户程序提供一个简单清晰的计算机模型，并管理以上所有设备。

定义也就有了：操作系统是一个用来协调、管理和控制计算机硬件和软件资源的系统程序，它位于硬件和应用程序之间。

（程序是运行在系统上的具有某种功能的软件，比如说浏览器，音乐播放器等。）

操作系统的内核的定义：操作系统的内核是一个管理和控制程序，负责管理计算机的所有物理资源，其中包括：文件系统、内存管理、设备管理和进程管理。

二 操作系统历史

2.1 真空管与穿孔卡片（无操作系统）

过程：

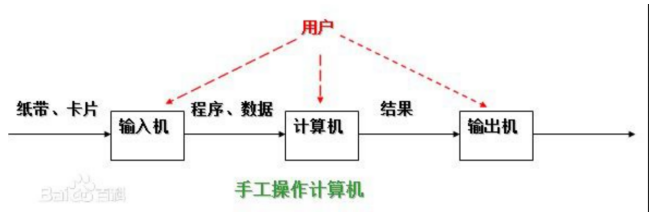
万能程序员们将对应于程序和数据的已穿孔的纸带（或卡片）装入输入机，然后启动输入机把程序和数据输入计算机内存，接着通过控制台开关启动程序针对数据运行；计算完毕，打印机输出计算结果；用户取走结果并卸下纸带（或卡片）后，才让下一个用户上机。

注意点：

1 程序员需要在墙上的计时表上预约时间
2 同一时刻只有一个程序在内存中被CPU调用运行（串行的）

优缺点：

优点：程序员在申请的时间段内独享整个资源，即时的调试自己的程序，如果有bug可以当场处理，
缺点：这对于计算机提供商来说是一种浪费（你买一台电脑4000块，那 一年中你用365比只用1天，肯定是省成本的，物尽其用）



2.2 晶体管和批处理系统

一代计算机的问题：

人机交互太多了（输入--->计算--->输出　输入--->计算--->输出 输入--->计算--->输出 ）

解决办法：

把一堆人的输入攒成一大波输入，然后顺序计算（这是有问题的，但是第二代计算没有解决）再把计算结果攒成一大波输出，这就是批处理系统

操作系统前身：

在收集了大约一个小时的批量作业之后，这些卡片被读入磁带，然后磁带被送到机房里并装到磁带上。然后磁带被送到机房里并装到磁带机上。随后，操作员装入一个特殊的程序（此乃现代操作系统的前身），它负责从磁带上读入第一个作业（job，一个或一组程序）并运行，其输出写到第二个磁带上，而且不打印。每个作业结束后，操作系统自动的从磁带上读入下一个作业并且运行。当一整批的作业全部结束后，操作员去下输入和输出磁带，讲输入磁带换成下一批作业，并且把输出磁带拿到一台1041机器上进行脱机（不与主计算机联机）打印

优点：批处理
缺点： 1 图的中间还有俩小人 2 仍然是顺序计算

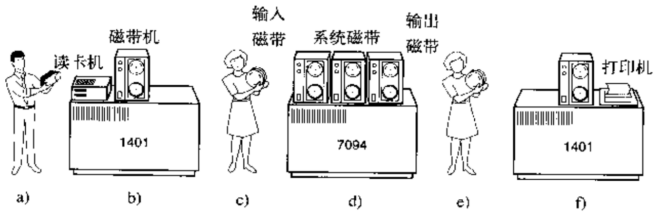


图1-3 一种早期的批处理系统：a) 程序员将卡片拿到1401机处；b) 1401机将批处理作业读到磁带上；c) 操作员将输入带送至7094机；d) 7094机进行计算；e) 操作员将输出磁带送到1401机；f) 1401机打印输出

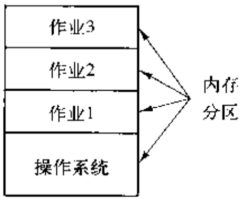


图1-5 一个内存中有三个作业的多道程序系统

2.3 集成电路芯片和多道程序设计

针对二代计算机的两个主要问题

开发出SPOOLING技术：
卡片被拿到机房后能够很快的将作业从卡片读入磁盘，于是任何时刻当一个作业结束时，操作系统就能将一个作业从磁带读出，装进空出啦的内存区域运行，这种技术叫做同时的外部设备联机操作：SPOOLING该技术同时用于输出。当采用了这种技术后，就不在需要IBM1401机了，也不必将磁带搬来搬去了（中间俩小人失业了），强化了操作系统的功能

开发出多道程序设计，用于解决顺序执行的问题：

在7094机上（程序运行的机器），若当前作业因等待磁带或等待其他IO操作而暂停，CPU就处于休闲状态直至IO操作完成，对于CPU密集的科学计算，IO操作少，浪费时间不明显，对于商业数据处理，IO等待能到达80%～90%，所以必须解决CPU浪费的现象。

解决方案：将内存分为几个部分，每一部分存放不同的作业，如图1-5所示。当一个作业等待IO完成时，另一个作业可以使用CPU，内存中放足够的作业，则CPU的利用率能接近100%

此时的第三代计算机适合大型科学计算和繁忙的商务数据处理，但，本质上其仍是一个批处理系统。
虽然解决了诸如以上问题，但多个作业必须在全部运行结束后，才能得到结果，从一个作业的提交到运算结果取回往往长达数小时。
想象一个场景：A君 B君 C君 三个程序员同时在调试程序，一旦A君写错一个逗号，那么可能需要半天的时间才能看到结果，因为B君C君的结果也同时运算出来了。时间必然要长。一言以蔽之：大家一起存作业，大家一起去数据（磁带）

许多程序员怀念第一代独享的计算机，可以即时调试自己的程序。为了满足程序员们很快可以得到响应，出现了分时操作系统

分时操作系统：多个联机终端+多道技术

20个客户端同时加载到内存，有17在思考，3个在运行，cpu就采用多道的方式处理内存中的这3个程序，由于客户提交的一般都是简短的指令而且很少有耗时长的，索引计算机能够为许多用户提供快速的交互式服务，所有的用户都以为自己独享了计算机资源

2.4 个人计算机

随着大规模集成电路的发展，每平方厘米的硅片芯片上可以集成数千个晶体管，个人计算机的时代就此到来。

三 进程和线程

进程

假如有两个程序A和B，程序A在执行到一半的过程中，需要读取大量的数据输入（I/O操作），
而此时CPU只能静静地等待任务A读取完数据才能继续执行，这样就白白浪费了CPU资源。
是不是在程序A读取数据的过程中，让程序B去执行，当程序A读取完数据之后，让
程序B暂停，然后让程序A继续执行?

当然没问题，但这里有一个关键词：切换
既然是切换，那么这就涉及到了状态的保存，状态的恢复，加上程序A与程序B所需要的系统资源（内存，硬盘，键盘等等）是不一样的。自然而然的就需要有一个东西去记录程序A和程序B分别需要什么资源，怎样去识别程序A和程序B等等，所以就有了一个叫进程的抽象

进程定义：

进程就是一个程序在一个数据集上的一次动态执行过程。
进程一般由程序、数据集、进程控制块三部分组成。
我们编写的程序用来描述进程要完成哪些功能以及如何完成；
数据集则是程序在执行过程中所需要使用的资源；
进程控制块用来记录进程的外部特征，描述进程的执行变化过程，系统可以利用它来控制和管理进程，它是系统感知进程存在的唯一标志。

举一例说明进程：

想象一位有一手好厨艺的计算机科学家正在为他的女儿烘制生日蛋糕。他有做生日蛋糕的食谱，厨房里有所需的原料：面粉、鸡蛋、糖、香草汁等。在这个比喻中，做蛋糕的食谱就是程序（即用适当形式描述的算法）计算机科学家就是处理器（cpu），而做蛋糕的各种原料就是输入数据。进程就是厨师阅读食谱、取来各种原料以及烘制蛋糕等一系列动作的总和。现在假设计算机科学家的儿子哭着跑了进来，说他的头被一只蜜蜂蛰了。计算机科学家就记录下他照着食谱做到哪儿了（保存进程的当前状态），然后拿出一本急救手册，按照其中的指示处理蛰伤。这里，我们看到处理机从一个进程（做蛋糕）切换到另一个高优先级的进程（实施医疗救治），每个进程拥有各自的程序（食谱和急救手册）。当蜜蜂蛰伤处理完之后，这位计算机科学家又回来做蛋糕，从他离开时的那一步继续做下去。

线程

线程的出现是为了降低上下文切换的消耗，提高系统的并发性，并突破一个进程只能干一样事的缺陷，使到进程内并发成为可能。

假设，一个文本程序，需要接受键盘输入，将内容显示在屏幕上，还需要保存信息到硬盘中。若只有一个进程，势必造成同一时间只能干一样事的尴尬（当保存时，就不能通过键盘输入内容）。若有多个进程，每个进程负责一个任务，进程A负责接收键盘输入的任务，进程B负责将内容显示在屏幕上的任务，进程C负责保存内容到硬盘中的任务。这里进程A，B，C间的协作涉及到了进程通信问题，而且有共同都需要拥有的东西--------文本内容，不停的切换造成性能上的损失。若有一种机制，可以使任务A，B，C共享资源，这样上下文切换所需要保存和恢复的内容就少了，同时又可以减少通信所带来的性能损耗，那就好了。是的，这种机制就是线程。

线程也叫轻量级进程，它是一个基本的CPU执行单元，也是程序执行过程中的最小单元，由线程ID、程序计数器、寄存器集合和堆栈共同组成。线程的引入减小了程序并发执行时的开销，提高了操作系统的并发性能。线程没有自己的系统资源。

线程进程的关系区别

1. Threads share the address space of the process that created it; processes have their own address space.
2. Threads have direct access to the data segment of its process; processes have their own copy of the data segment of the parent process.
3. Threads can directly communicate with other threads of its process; processes must use interprocess communication to communicate with sibling processes.
4. New threads are easily created; new processes require duplication of the parent process.
5. Threads can exercise considerable control over threads of the same process; processes can only exercise control over child processes.
6. Changes to the main thread (cancellation, priority change, etc.) may affect the behavior of the other threads of the process; changes to the parent process does not affect child processes.

1 一个程序至少有一个进程，一个进程至少有一个线程.（进程可以理解成线程的容器）

2 进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

3 线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

4 进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，进程是系统进行资源分配和调度的一个独立单位.
线程是进程的一个实体，是CPU调度和分派的基本单位，它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源（如程序计数器，一组寄存器和栈）但是

它可与同属一个进程的其他的线程共享进程所拥有的全部资源.
一个线程可以创建和撤销另一个线程;同一个进程中的多个线程之间可以并发执行.

python的GIL

In CPython, the global interpreter lock, or GIL, is a mutex that prevents multiple native threads from executing Python bytecodes at once. This lock is necessary mainly because CPython's memory management is not thread-safe. (However, since the GIL exists, other features have grown to depend on the guarantees that it enforces.)

上面的核心意思就是，**无论你启多少个线程，你有多少个cpu, Python在执行的时候会淡定的在同一时刻只允许一个线程运行**

<python的线程与threading模块>

一 线程的两种调用方式

threading 模块建立在thread 模块之上。thread模块以低级、原始的方式来处理和控制线程，而threading 模块通过对thread进行二次封装，

提供了更方便的api来处理线程。

直接调用：

```
import threading
import time

def sayhi(num): #定义每个线程要运行的函数

    print("running on number:%s" %num)

    time.sleep(3)

if __name__ == '__main__':

    t1 = threading.Thread(target=sayhi,args=(1,)) #生成一个线程实例
    t2 = threading.Thread(target=sayhi,args=(2,)) #生成另一个线程实例

    t1.start() #启动线程
    t2.start() #启动另一个线程

    print(t1.getName()) #获取线程名
    print(t2.getName())
```

继承式调用：

```
import threading
import time


class MyThread(threading.Thread):
    def __init__(self,num):
        threading.Thread.__init__(self)
        self.num = num

    def run(self):#定义每个线程要运行的函数

        print("running on number:%s" %self.num)

        time.sleep(3)

if __name__ == '__main__':

    t1 = MyThread(1)
    t2 = MyThread(2)
    t1.start()
    t2.start()

    print("ending......")
```

二 threading.thread的实例方法

join&Daemon方法

```
import threading
from time import ctime,sleep
import time

def ListenMusic(name):

        print ("Begin listening to %s. %s" %(name,ctime()))
        sleep(3)
        print("end listening %s"%ctime())

def RecordBlog(title):

        print ("Begin recording the %s! %s" %(title,ctime()))
        sleep(5)
        print('end recording %s'%ctime())


threads = []


t1 = threading.Thread(target=ListenMusic,args=('水手',))
t2 = threading.Thread(target=RecordBlog,args=('python线程',))

threads.append(t1)
threads.append(t2)

if __name__ == '__main__':

    for t in threads:
        #t.setDaemon(True) #注意:一定在start之前设置
        t.start()
        # t.join()
    # t1.join()
    t1.setDaemon(True)

    #t2.join()########考虑这三种join位置下的结果?
    print ("all over %s" %ctime())
```

join()：在子线程完成运行之前，这个子线程的父线程将一直被阻塞。

setDaemon(True)：

将线程声明为守护线程，必须在start() 方法调用之前设置， 如果不设置为守护线程程序会被无限挂起。这个方法基本和join是相反的。

当我们 在程序运行中，执行一个主线程，如果主线程又创建一个子线程，主线程和子线程 就分兵两路，分别运行，那么当主线程完成想退出时，会检验子线程是否完成。如 果子线程未完成，则主线程会等待子线程完成后再退出。但是有时候我们需要的是 只要主线程完成了，不管子线程是否完成，都要和主线程一起退出，这时就可以 用setDaemon方法啦

其它方法

```
# run():  线程被cpu调度后自动执行线程对象的run方法
# start():启动线程活动。
# isAlive(): 返回线程是否活动的。
# getName(): 返回线程名。
# setName(): 设置线程名。

threading模块提供的一些方法:
# threading.currentThread(): 返回当前的线程变量。
# threading.enumerate(): 返回一个包含正在运行的线程的list。正在运行指线程启动后、结束前，不包括启动前和终止后的线程。
# threading.activeCount(): 返回正在运行的线程数量，与len(threading.enumerate())有相同的结果。
```

三 同步锁(Lock)

```
import time
import threading
```

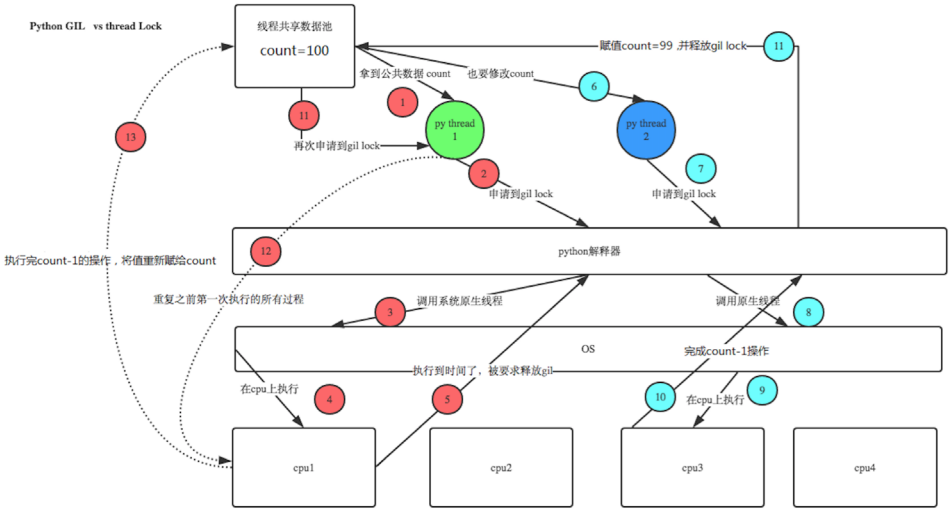
```
def addNum():
    global num #在每个线程中都获取这个全局变量
    #num-=1

    temp=num
    #print('--get num:',num )
    time.sleep(0.1)
    num =temp-1 #对此公共变量进行-1操作

num = 100  #设定一个共享变量
thread_list = []
for i in range(100):
    t = threading.Thread(target=addNum)
    t.start()
    thread_list.append(t)

for t in thread_list: #等待所有线程执行完毕
    t.join()

print('final num:', num )
```



观察：time.sleep(0.1) /0.001/0.0000001 结果分别是多少?

多个线程都在同时操作同一个共享资源，所以造成了资源破坏，怎么办呢？(join会造成串行，失去所线程的意义)

我们可以通过同步锁来解决这种问题

```
R=threading.Lock()

####
def sub():
    global num
    R.acquire()
    temp=num-1
    time.sleep(0.1)
    num=temp
    R.release()
```

四 线程死锁和递归锁

在线程间共享多个资源的时候，如果两个线程分别占有一部分资源并且同时等待对方的资源，就会造成死锁，因为系统判断这部分资源都正在使用，所有这两个线程在无外力作用下将一直等待下去。下面是一个死锁的例子：

```
import threading,time

class myThread(threading.Thread):
    def doA(self):
        lockA.acquire()
        print(self.name,"gotlockA",time.ctime())
```

```
        time.sleep(3)
        lockB.acquire()
        print(self.name,"gotlockB",time.ctime())
        lockB.release()
        lockA.release()

    def doB(self):
        lockB.acquire()
        print(self.name,"gotlockB",time.ctime())
        time.sleep(2)
        lockA.acquire()
        print(self.name,"gotlockA",time.ctime())
        lockA.release()
        lockB.release()

    def run(self):
        self.doA()
        self.doB()
if __name__=="__main__":

    lockA=threading.Lock()
    lockB=threading.Lock()
    threads=[]
    for i in range(5):
        threads.append(myThread())
    for t in threads:
        t.start()
    for t in threads:
        t.join()#等待线程结束，后面再讲。
```

解决办法：使用递归锁，将

```
lockA=threading.Lock()
lockB=threading.Lock()<br>#--------------<br>lock=threading.RLock()
```

为了支持在同一线程中多次请求同一资源，python提供了“可重入锁”：threading.RLock。RLock内部维护着一个Lock和一个counter变量，counter记录了acquire的次数，从而使得资源可以被多次acquire。直到一个线程所有的acquire都被release，其他的线程才能获得资源。

应用

View Code

同步条件(Event)

An event is a simple synchronization object;the event represents an internal flag,

and threads can wait for the flag to be set, or set or clear the flag themselves.

event = threading.Event()

# a client thread can wait for the flag to be set
event.wait()

# a server thread can set or reset it
event.set()
event.clear()

If the flag is set, the wait method doesn’t do anything.
If the flag is cleared, wait will block until it becomes set again.
Any number of threads may wait for the same event.

```
import threading,time
class Boss(threading.Thread):
    def run(self):
        print("BOSS: 今晚大家都要加班到22:00。")
        print(event.isSet())
        event.set()
        time.sleep(5)
        print("BOSS: <22:00>可以下班了。")
```

```
        print(event.isSet())
        event.set()
class Worker(threading.Thread):
    def run(self):
        event.wait()
        print("Worker: 哎……命苦啊！ ")
        time.sleep(1)
        event.clear()
        event.wait()
        print("Worker: OhYeah!")
if __name__=="__main__":
    event=threading.Event()
    threads=[]
    for i in range(5):
        threads.append(Worker())
    threads.append(Boss())
    for t in threads:
        t.start()
    for t in threads:
        t.join()
```

信号量(Semaphore)

信号量用来控制线程并发数的，BoundedSemaphore或Semaphore管理一个内置的计数 器，每当调用acquire()时-1，调用release()时+1。

计数器不能小于0，当计数器为 0时，acquire()将阻塞线程至同步锁定状态，直到其他线程调用release()。(类似于停车位的概念)

BoundedSemaphore与Semaphore的唯一区别在于前者将在调用release()时检查计数 器的值是否超过了计数器的初始值，如果超过了将抛出一个异常。

```
import threading,time
class myThread(threading.Thread):
    def run(self):
        if semaphore.acquire():
            print(self.name)
            time.sleep(5)
            semaphore.release()
if __name__=="__main__":
    semaphore=threading.Semaphore(5)
    thrs=[]
    for i in range(100):
        thrs.append(myThread())
    for t in thrs:
        t.start()
```

多线程利器 - - - 队列(queue)

列表是不安全的数据结构

View Code

思考：如何通过对列来完成上述功能?

queue is especially useful in threaded programming when information must be exchanged safely between multiple threads.

queue列队类的方法

```
创建一个"队列"对象
import Queue
q = Queue.Queue(maxsize = 10)
Queue.Queue类即是一个队列的同步实现。队列长度可为无限或者有限。可通过Queue的构造函数的可选参数maxsize来设定队列长度。如果maxsize小于1就表示队列长度无

将一个值放入队列中
q.put(10)
调用队列对象的put()方法在队尾插入一个项目。put()有两个参数，第一个item为必需的，为插入项目的值；第二个block为可选参数，默认为
1。如果队列当前为空且block为1，put()方法就使调用线程暂停,直到空出一个数据单元。如果block为0，put方法将引发Full异常。

将一个值从队列中取出
q.get()
调用队列对象的get()方法从队头删除并返回一个项目。可选参数为block，默认为True。如果队列为空且block为True，
get()就使调用线程暂停，直至有项目可用。如果队列为空且block为False，队列将引发Empty异常。
```

```
Python Queue模块有三种队列及构造函数：
1、Python Queue模块的FIFO队列先进先出。    class queue.Queue(maxsize)
2、LIFO类似于堆，即先进后出。                 class queue.LifoQueue(maxsize)
3、还有一种是优先级队列级别越低越先出来。        class queue.PriorityQueue(maxsize)

此包中的常用方法(q = Queue.Queue()):
q.qsize() 返回队列的大小
q.empty() 如果队列为空，返回True,反之False
q.full() 如果队列满了，返回True,反之False
q.full 与 maxsize 大小对应
q.get([block[, timeout]]) 获取队列，timeout等待时间
q.get_nowait() 相当q.get(False)
非阻塞 q.put(item) 写入队列，timeout等待时间
q.put_nowait(item) 相当q.put(item, False)
q.task_done() 在完成一项工作之后，q.task_done() 函数向任务已经完成的队列发送一个信号
q.join() 实际上意味着等到队列为空，再执行别的操作
```

other mode:

View Code

生产者消费者模型：

为什么要使用生产者和消费者模式

在线程世界里，生产者就是生产数据的线程，消费者就是消费数据的线程。在多线程开发当中，如果生产者处理速度很快，而消费者处理速度很慢，那么生产者就必须等待消费者处理完，才能继续生产数据。同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。为了解决这个问题于是引入了生产者和消费者模式。

什么是生产者消费者模式

生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

这就像，在餐厅，厨师做好菜，不需要直接和客户交流，而是交给前台，而客户去饭菜也不需要不找厨师，直接去前台领取即可，这也是一个结耦的过程。

```python
import time,random
import queue,threading

q = queue.Queue()

def Producer(name):
  count = 0
  while count <10:
    print("making........")
    time.sleep(random.randrange(3))
    q.put(count)
    print('Producer %s has produced %s baozi..' %(name, count))
    count +=1
    #q.task_done()
    #q.join()
    print("ok......")
def Consumer(name):
  count = 0
  while count <10:
    time.sleep(random.randrange(4))
    if not q.empty():
        data = q.get()
        #q.task_done()
        #q.join()
        print(data)
        print('\033[32;1mConsumer %s has eat %s baozi...\033[0m' %(name, data))
    else:
        print("-----no baozi anymore----")
    count +=1

p1 = threading.Thread(target=Producer, args=('A',))
c1 = threading.Thread(target=Consumer, args=('B',))
# c2 = threading.Thread(target=Consumer, args=('C',))
```

```
# c3 = threading.Thread(target=Consumer, args=('D',))
p1.start()
c1.start()
# c2.start()
# c3.start()
```

<三 多进程模块 multiprocessing>

M ultiprocessing is a package that supports spawning processes using an API similar to the threading module. The multiprocessing package offers both local and remote concurrency,effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads. Due to this, the multiprocessing module allows the programmer to fully leverage multiple processors on a given machine. It runs on both Unix and Windows.

由于GIL的存在，python中的多线程其实并不是真正的多线程，如果想要充分地使用多核CPU的资源，在python中大部分情况需要使用多进程。

multiprocessing包是Python中的多进程管理包。与threading.Thread类似，它可以利用multiprocessing.Process对象来创建一个进程。该进程可以运行在Python程序内部编写的函数。该Process对象与Thread对象的用法相同，也有start(), run(), join()的方法。此外multiprocessing包中也有Lock/Event/Semaphore/Condition类(这些对象可以像多线程那样，通过参数传递给各个进程)，用以同步进程，其用法与threading包中的同名类一致。所以，multiprocessing的很大一部份与threading使用同一套API，只不过换到了多进程的情境。

一 进程的调用

调用方式1

```
from multiprocessing import Process
import time
def f(name):
    time.sleep(1)
    print('hello', name,time.ctime())

if __name__ == '__main__':
    p_list=[]
    for i in range(3):
        p = Process(target=f, args=('alvin',))
        p_list.append(p)
        p.start()
    for i in p_list:
        p.join()
    print('end')
```

调用方式2

```
from multiprocessing import Process
import time

class MyProcess(Process):
    def __init__(self):
        super(MyProcess, self).__init__()
        #self.name = name

    def run(self):
        time.sleep(1)
        print ('hello', self.name,time.ctime())


if __name__ == '__main__':
    p_list=[]
    for i in range(3):
        p = MyProcess()
        p.start()
        p_list.append(p)

    for p in p_list:
        p.join()

    print('end')
```

To show the individual process IDs involved, here is an expanded example:

```
from multiprocessing import Process
import os
import time
def info(title):

    print("title:",title)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main process line')
    time.sleep(1)
    print("------------------")
    p = Process(target=info, args=('yuan',))
    p.start()
    p.join()
```

二 Process类

构造方法：

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

group: 线程组，目前还没有实现，库引用中提示必须是None；
target: 要执行的方法；
name: 进程名；
args/kwargs: 要传入方法的参数。

实例方法：

is_alive()：返回进程是否在运行。

join([timeout])：阻塞当前上下文环境的进程程，直到调用此方法的进程终止或到达指定的timeout（可选参数）。

start()：进程准备就绪，等待CPU调度

run()：strat()调用run方法，如果实例进程时未制定传入target，这star执行t默认run()方法。

terminate()：不管任务是否完成，立即停止工作进程

属性：

daemon：和线程的setDeamon功能一样

name：进程名字。

pid：进程号。

View Code

三 进程间通讯

3.1 进程对列Queue

```
from multiprocessing import Process, Queue
import queue

def f(q,n):
    #q.put([123, 456, 'hello'])
    q.put(n*n+1)
    print("son process",id(q))

if __name__ == '__main__':
    q = Queue()  #try: q=queue.Queue()
    print("main process",id(q))

    for i in range(3):
        p = Process(target=f, args=(q,i))
```

```
        p.start()

    print(q.get())
    print(q.get())
    print(q.get())
```

3.2 管道

The `Pipe()` function returns a pair of connection objects connected by a pipe which by default is duplex (two-way). For example:

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([12, {"name":"yuan"}, 'hello'])
    response=conn.recv()
    print("response",response)
    conn.close()
    print("q_ID2:",id(child_conn))

if __name__ == '__main__':

    parent_conn, child_conn = Pipe()
    print("q_ID1:",id(child_conn))
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # prints "[42, None, 'hello']"
    parent_conn.send("儿子你好!")
    p.join()
```

The two connection objects returned by `Pipe()` represent the two ends of the pipe. Each connection object has `send()` and `recv()` methods (among others). Note that data in a pipe may become corrupted if two processes (or threads) try to read from or write to the *same* end of the pipe at the same time. Of course there is no risk of corruption from processes using different ends of the pipe at the same time.

3.3 Managers

Queue和pipe只是实现了数据交互，并没实现数据共享，即一个进程去更改另一个进程的数据。

A manager object returned by `Manager()` controls a server process which holds Python objects and allows other processes to manipulate them using proxies.

A manager returned by `Manager()` will support types `list`, `dict`, `Namespace`, `Lock`, `RLock`, `Semaphore`, `BoundedSemaphore`, `Condition`, `Event`, `Barrier`, `Queue`, `Value` and `Array`. For example:

```
from multiprocessing import Process, Manager

def f(d, l,n):
    d[n] = '1'
    d['2'] = 2
    d[0.25] = None
    l.append(n)
    #print(l)

    print("son process:",id(d),id(l))

if __name__ == '__main__':

    with Manager() as manager:

        d = manager.dict()

        l = manager.list(range(5))

        print("main process:",id(d),id(l))

        p_list = []

        for i in range(10):
            p = Process(target=f, args=(d,l,i))
            p.start()
            p_list.append(p)
```

```
        for res in p_list:
            res.join()

        print(d)
        print(l)
```

四 进程同步

Without using the lock output from the different processes is liable to get all mixed up.

```
from multiprocessing import Process, Lock

def f(l, i):

    with l.acquire():
        print('hello world %s'%i)

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

五 进程池

进程池内部维护一个进程序列，当使用时，则去进程池中获取一个进程，如果进程池序列中没有可供使用的进进程，那么程序就会等待，直到进程池中有可用进程为止。

进程池中有两个方法：

- apply
- apply_async

```
from  multiprocessing import Process,Pool
import time,os

def Foo(i):
    time.sleep(1)
    print(i)
    return i+100

def Bar(arg):

    print(os.getpid())
    print(os.getppid())
    print('logger:',arg)

pool = Pool(5)

Bar(1)
print("-----------------")

for i in range(10):
    #pool.apply(func=Foo, args=(i,))
    #pool.apply_async(func=Foo, args=(i,))
    pool.apply_async(func=Foo, args=(i,),callback=Bar)

pool.close()
pool.join()
print('end')
```

<四 Python中的上下文管理器(contextlib模块)>

上下文管理器的任务是：代码块执行前准备，代码块执行后收拾

1 如何使用上下文管理器：

如何打开一个文件，并写入"hello world"

```
filename="my.txt"
mode="w"
f=open(filename,mode)
f.write("hello world")
f.close()
```

当发生异常时（如磁盘写满），就没有机会执行第5行。当然，我们可以采用try-finally语句块进行包装：

```
writer=open(filename,mode)
try:
    writer.write("hello world")
finally:
    writer.close()
```

当我们进行复杂的操作时，try-finally语句就会变得丑陋，采用with语句重写：

```
with open(filename,mode) as writer:
    writer.write("hello world")
```

as指代了从open()函数返回的内容，并把它赋给了新值。with完成了try-finally的任务。

2 自定义上下文管理器

with语句的作用类似于try-finally，提供一种上下文机制。要应用with语句的类，其内部必须提供两个内置函数__enter__和__exit__。前者在主体代码执行前执行，后者在主体代码执行后执行。as后面的变量，是在__enter__函数中返回的。

```
class echo():
    def output(self):
        print "hello world"
    def __enter__(self):
        print "enter"
        return self  #可以返回任何希望返回的东西
    def __exit__(self,exception_type,value,trackback):
        print "exit"
        if exception_type==ValueError:
            return True
        else:
            return Flase

>>>with echo as e:
    e.output()

输出：
enter
hello world
exit
```

完备的__exit__函数如下：

```
def __exit__(self,exc_type,exc_value,exc_tb)
```

其中，exc_type:异常类型；exc_value:异常值；exc_tb:异常追踪信息

当__exit__返回True时，异常不传播

3 contextlib模块

contextlib模块的作用是提供更易用的上下文管理器，它是通过Generator实现的。contextlib中的contextmanager作为装饰器来提供一种针对函数级别的上下文管理机制，常用框架如下：

```
from contextlib import contextmanager
@contextmanager
def make_context():
    print 'enter'
    try:
```

```
        yield "ok"
    except RuntimeError,err:
        print 'error',err
    finally:
        print 'exit'

>>>with make_context() as value:
    print value

输出为:
    enter
    ok
    exit
```

其中，yield写入try-finally中是为了保证异常安全（能处理异常）as后的变量的值是由yield返回。yield前面的语句可看作代码块执行前操作，yield之后的操作可以看作在__exit__函数中的操作。

以线程锁为例：

```
@contextlib.contextmanager
def loudLock():
    print 'Locking'
    lock.acquire()
    yield
    print 'Releasing'
    lock.release()

with loudLock():
    print 'Lock is locked: %s' % lock.locked()
    print 'Doing something that needs locking'

#Output:
#Locking
#Lock is locked: True
#Doing something that needs locking
#Releasing
```

4 contextlib.nested减少嵌套

对于：

```
with open(filename,mode) as reader:
    with open(filename1,mode1) as writer:
        writer.write(reader.read())
```

可以通过contextlib.nested进行简化：

```
with contextlib.nested(open(filename,mode),open(filename1,mode1)) as (reader,writer):
    writer.write(reader.read())
```

在python 2.7及以后，被一种新的语法取代：

```
with open(filename,mode) as reader,open(filename1,mode1) as writer:
    writer.write(reader.read())
```

5 contextlib.closing()

file类直接支持上下文管理器API，但有些表示打开句柄的对象并不支持，如urllib.urlopen()返回的对象。还有些遗留类，使用close()方法而不支持上下文管理器API。为了确保关闭句柄，需要使用closing()为它创建一个上下文管理器（调用类的close方法）。

View Code

协程

协程，又称微线程，纤程。英文名Coroutine。

优点1: 协程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显。

优点2: 不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

因为协程是一个线程执行，那怎么利用多核CPU呢？最简单的方法是多进程+协程，既充分利用多核，又充分发挥协程的高效率，可获得极高的性能。

yield的简单实现

```
import time
import queue

def consumer(name):
    print("--->ready to eat baozi...")
    while True:
        new_baozi = yield
        print("[%s] is eating baozi %s" % (name,new_baozi))
        #time.sleep(1)

def producer():

    r = con.__next__()
    r = con2.__next__()
    n = 0
    while 1:
        time.sleep(1)
        print("\033[32;1m[producer]\033[0m is making baozi %s and %s" %(n,n+1) )
        con.send(n)
        con2.send(n+1)

        n +=2


if __name__ == '__main__':
    con = consumer("c1")
    con2 = consumer("c2")
    p = producer()
```

Greenlet

greenlet是一个用C实现的协程模块，相比与python自带的yield，它可以使你在任意函数之间随意切换，而不需把这个函数先声明为generator

```
from greenlet import greenlet


def test1():
    print(12)
    gr2.switch()
    print(34)
    gr2.switch()


def test2():
    print(56)
    gr1.switch()
    print(78)


gr1 = greenlet(test1)
gr2 = greenlet(test2)
gr1.switch()
```

Gevent

```
import gevent

import requests,time
```

```
start=time.time()

def f(url):
    print('GET: %s' % url)
    resp =requests.get(url)
    data = resp.text
    print('%d bytes received from %s.' % (len(data), url))

gevent.joinall([

        gevent.spawn(f, 'https://www.python.org/'),
        gevent.spawn(f, 'https://www.yahoo.com/'),
        gevent.spawn(f, 'https://www.baidu.com/'),
        gevent.spawn(f, 'https://www.sina.com.cn/'),

])

# f('https://www.python.org/')
#
# f('https://www.yahoo.com/')
#
# f('https://baidu.com/')
#
# f('https://www.sina.com.cn/')

print("cost time:",time.time()-start)
```