

金角大王Alex

Chaos is a ladder.

博客园

首页

新随笔

联系

已订阅

管理

公告

最新自学视频 路飞Python免费小课

人生迷茫问题可以加Alex个人微信听鸡

Alex金角大王

扫一扫上面的二维码图案，加我微信

汤

网管到CEO的10年逆袭之路

Alex给你清晰的职业规划和执行策略

Alex

老男孩Python教学总监 | 路飞学城CEO

面向对象编程

Alex | 前汽车之家架构师

面向对象开发原来如此简单

16人在学

进阶

12小时

Python常用模块

Alex | 前汽车之家架构师

Python开发中最常用的11个模块精讲

10人在学

进阶

6小时

Python开发21天

Alex | 前汽车之家架构师

跟随Alex金角大王3周上手Python开发

124人在学

入门

19小时

我的标签

职业发展(3)

创业(2)

随笔分类

职业&生活随笔(22)

Python之路,Day9, 进程、线程、协程篇

随笔 - 29 文章 - 64 评论 - 980

Flack me on GitHub

Python之路,Day9, 进程、线程、协程篇

本节内容

1. 操作系统发展史介绍

2. 进程、与线程区别

3. python GIL全局解释器锁

4. 线程

1. 语法

2. join

3. 线程锁之Lock\Rlock\信号量

4. 将线程变为守护进程

5. Event事件

6. queue队列

7. 生产者消费者模型

8. Queue队列

9. 开发一个线程池

5. 进程

1. 语法

2. 进程间通讯

3. 进程池

操作系统发展史

手工操作（无操作系统）

1946年第一台计算机诞生--20世纪50年代中期，还未出现操作系统，计算机工作采用手工操作方式。

手工操作

程序员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。

手工操作计算机

手工操作方式两个特点：

(1) 用户独占全机。不会出现因资源已被其他用户占用而等待的现象，但资源的利用率低。

(2) CPU 等待手工操作。CPU的利用不充分。

20世纪50年代后期，出现人机矛盾：手工操作的慢速度和计算机的高速度之间形成了尖锐矛盾，手工操作方式已严重损害了系统资源的利用率（使资源利用率降为百分之几，甚至更低），不能容忍。唯一的解决办法：只有摆脱人的手工操作，实现作业的自动过渡。这样就出现了成批处理。

批处理系统

批处理系统：加载在计算机上的一个系统软件，在它的控制下，计算机能够自动地、成批地处理一个或多个用户的作业（这作业包括程序、数据和命令）。

https://www.cnblogs.com/alex3714/articles/5230609.html

1/19

文章分类

- Python全栈开发之路(12)
- Python学习目录(4)
- Python自动化开发之路(33)
- 爬虫(6)

最新评论

1. Re:编程要自学或报班这事你都想不明白, 那必然是你智商不够
牛逼克拉斯

--晋北高峰

2. Re:Django 14天从小白到进阶- Day3
搞定Views组件
不更新了吗

--30岁老古董

3. Re:编程要自学或报班这事你都想不明白, 那必然是你智商不够
大王性格直爽, 有目标, 肯付出行动! 偶像! 我要是早几年遇见大王就好了, 现在都已经三十了, 浪费了好青春!

--Xiyue666

4. Re:python 之路, Day11 - python
mysql and ORM
ALTER mytable ADD INDEX
[indexName] ON (username(length))
这句应改为: alter table mytable add
index index...

--原竹

5. Re:Python之路,Day3 - Python基础3
@我的恋人叫臭臭 淫角大王听说很厉害
吧? ...

--Xiyue666

网友排行榜

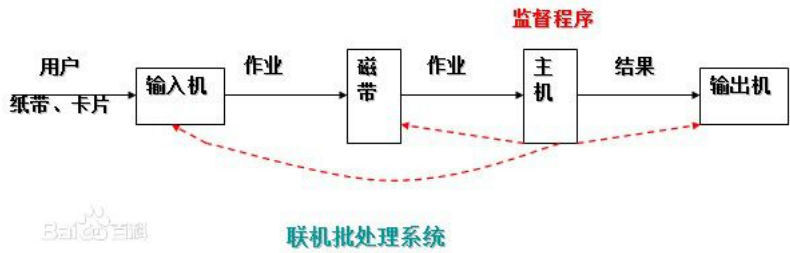
- 1. python 之路, 200行Python代码写了个打飞机游戏! (52782)
- 2. Django + Uwsgi + Nginx 实现生产环境部署(31842)
- 3. Python Select 解析(27203)
- 4. 为什么很多IT公司不喜欢进过培训机构的人呢? (20584)
- 5. 编程要自学或报班这事你都想不明白, 那必然是你智商不够(16948)

推荐排行榜

- 1. 给一位做技术迷茫的同学回信(63)
- 2. 你做了哪些事, 导致老板下调了对你的评价? (51)
- 3. 关于认识、格局、多维度发展的感触(46)
- 4. 为什么很多IT公司不喜欢进过培训机构的人呢? (37)
- 5. 编程要自学或报班这事你都想不明白, 那必然是你智商不够(36)

联机批处理系统

首先出现的是联机批处理系统, 即作业的输入/输出由CPU来处理。
主机与输入机之间增加一个存储设备——磁带, 在运行于主机上的监督程序的自动控制下, 计算机可自动完成: 成批地把输入机上的用户作业读入磁带, 依次把磁带上的用户作业读入主机内存并执行并把计算结果向输出机输出。完成了上一批作业后, 监督程序又从输入机上输入另一批作业, 保存在磁带上, 并按上述步骤重复处理。



监督程序不停地处理各个作业, 从而实现了作业到作业的自动转接, 减少了作业建立时间和手工操作时间, 有效克服了人机矛盾, 提高了计算机的利用率。

但是, 在作业输入和结果输出时, 主机的高速CPU仍处于空闲状态, 等待慢速的输入/输出设备完成工作: 主机处于“忙等”状态。

脱机批处理系统

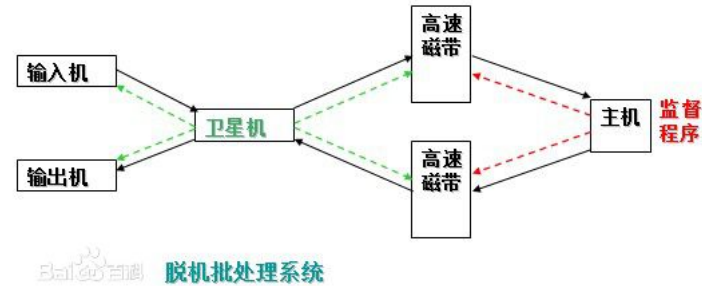
为克服与缓解高速主机与慢速外设的矛盾, 提高CPU的利用率, 又引入了脱机批处理系统, 即输入/输出脱离主机控制。

这种方式的显著特征是: 增加一台不与主机直接相连而专门用于与输入/输出设备打交道的卫星机。

其功能是:

- (1) 从输入机上读取用户作业并放到输入磁带上。
- (2) 从输出磁带上读取执行结果并传给输出机。

这样, 主机不是直接与慢速的输入/输出设备打交道, 而是与速度相对较快的磁带机发生关系, 有效缓解了主机与设备的矛盾。主机与卫星机可并行工作, 二者分工明确, 可以充分发挥主机的高速计算能力。



脱机批处理系统: 20世纪60年代应用十分广泛, 它极大缓解了人机矛盾及主机与外设的矛盾。IBM-7090/7094: 配备的监督程序就是脱机批处理系统, 是现代操作系统的原型。

不足: 每次主机内存中仅存放一道作业, 每当它运行期间发出输入/输出 (I/O) 请求后, 高速的CPU便处于等待低速的I/O完成状态, 致使CPU空闲。

为改善CPU的利用率, 又引入了多道程序系统。

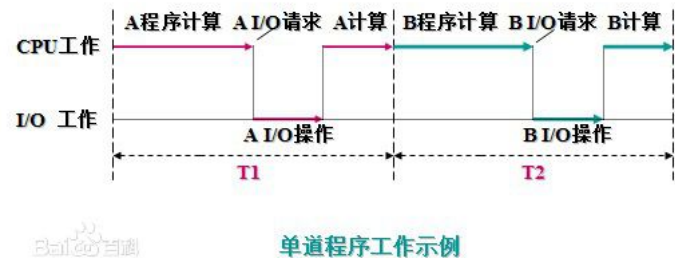
多道程序系统

多道程序设计技术

所谓多道程序设计技术, 就是指允许多个程序同时进入内存并运行。即同时把多个程序放入内存, 并允许它们交替在CPU中运行, 它们共享系统中的各种硬、软件资源。当一道程序因I/O请求而暂停运行时, CPU便立即转去运行另一道程序。

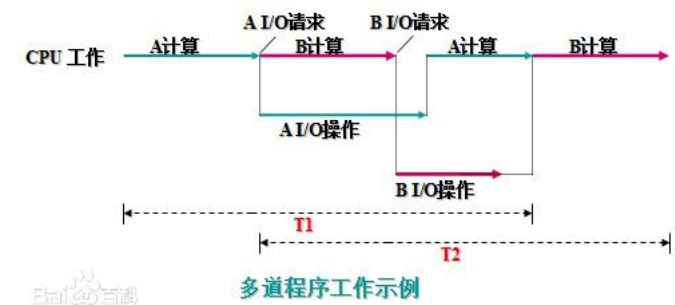
单道程序的运行过程:

在A程序计算时, I/O空闲, A程序I/O操作时, CPU空闲 (B程序也是同样); 必须A工作完成后, B才能进入内存中开始工作, 两者是串行的, 全部完成共需时间=T1+T2。



多道程序的运行过程：

将A、B两道程序同时存放在内存中，它们在系统的控制下，可相互穿插、交替地在CPU上运行：当A程序因请求I/O操作而放弃CPU时，B程序就可占用CPU运行，这样 CPU不再空闲，而正进行A I/O操作的I/O设备也不空闲，显然，CPU和I/O设备都处于“忙”状态，大大提高了资源的利用率，从而也提高了系统的效率，A、B全部完成所需时间 $< T_1 + T_2$ 。



多道程序设计技术不仅使CPU得到充分利用，同时改善I/O设备和内存的利用率，从而提高了整个系统的资源利用率和系统吞吐量（单位时间内处理作业（程序）的个数），最终提高了整个系统的效率。

- 单处理机系统中多道程序运行时特点：
- (1) 多道：计算机内存中同时存放几道相互独立的程序；
 - (2) 宏观上并行：同时进入系统的几道程序都处于运行过程中，即它们先后开始了各自的运行，但都未运行完毕；
 - (3) 微观上串行：实际上，各道程序轮流地用CPU，并交替运行。

多道程序系统的出现，标志着操作系统渐趋成熟的阶段，先后出现了作业调度管理、处理机管理、存储器管理、外部设备管理、文件系统管理等功能。

多道批处理系统

20世纪60年代中期，在前述的批处理系统中，引入多道程序设计技术后形成多道批处理系统（简称：批处理系统）。

- 它有两个特点：
- (1) 多道：系统内可同时容纳多个作业。这些作业放在外存中，组成一个后备队列，系统按一定的调度原则每次从后备作业队列中选取一个或多个作业进入内存运行，运行作业结束、退出运行和后备作业进入运行均由系统自动实现，从而在系统中形成一个自动转接的、连续的作业流。
 - (2) 成批：在系统运行过程中，不允许用户与其作业发生交互作用，即：作业一旦进入系统，用户就不能直接干预其作业的运行。

批处理系统的追求目标：提高系统资源利用率和系统吞吐量，以及作业流程的自动化。

批处理系统的一个重要缺点：不提供人机交互能力，给用户使用计算机带来不便。

虽然用户独占全机资源，并且直接控制程序的运行，可以随时了解程序运行情况。但这种工作方式因独占全机造成资源效率极低。

一种新的追求目标：既能保证计算机效率，又能方便用户使用计算机。20世纪60年代中期，计算机技术和软件技术的发展使这种追求成为可能。

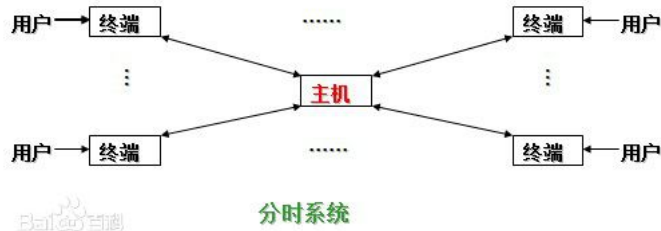
分时系统

由于CPU速度不断提高和采用分时技术，一台计算机可同时连接多个用户终端，而每个用户可在自己的终端上联机使用计算机，好象自己独占机器一样。

分时技术：把处理机的运行时间分成很短的时间片，按时间片轮流把处理机分配给各联机作业使用。

若某个作业在分配给它的时间片内不能完成其计算，则该作业暂时中断，把处理机让给另一作业使用，等待下一轮时再继续其运行。由于计算机速度很快，作业运行轮转得很快，给每个用户的印象是，好象他独占了一台计算机。而每个用户可以通过自己的终端向系统发出各种操作控制命令，在充分的人机交互情况下，完成作业的运行。

具有上述特征的计算机系统称为分时系统，它允许多个用户同时联机使用计算机。



特点：

- (1) 多路性。若干个用户同时使用一台计算机。微观上看是各用户轮流使用计算机；宏观上看是各用户并行工作。
- (2) 交互性。用户可根据系统对请求的响应结果，进一步向系统提出新的请求。这种能使用户与系统进行人机对话的工作方式，明显地有别于批处理系统，因而，分时系统又被称为交互式系统。
- (3) 独立性。用户之间可以相互独立操作，互不干扰。系统保证各用户程序运行的完整性，不会发生相互混淆或破坏现象。
- (4) 及时性。系统可对用户的输入及时作出响应。分时系统性能的主要指标之一是响应时间，它是指：从终端发出命令到系统予以应答所需的时间。

分时系统的主要目标：对用户响应的及时性，即不至于用户等待每一个命令的处理时间过长。

分时系统可以同时接纳数十个甚至上百个用户，由于内存空间有限，往往采用对换（又称交换）方式的存储方法。即将未“轮到”的作业放入磁盘，一旦“轮到”，再将其调入内存；而时间片用完后，又将作业存回磁盘（俗称“滚进”、“滚出”法），使同一存储区域轮流为多个用户服务。

多用户分时系统是当今计算机操作系统中最普遍使用的一类操作系统。

实时系统

虽然多道批处理系统和分时系统能获得较令人满意的资源利用率和系统响应时间，但却不能满足实时控制与实时信息处理两个应用领域的需求。于是就产生了实时系统，即系统能够及时响应随机发生的外部事件，并在严格的时间范围内完成对该事件的处理。

实时系统在一个特定的应用中常作为一种控制设备来使用。

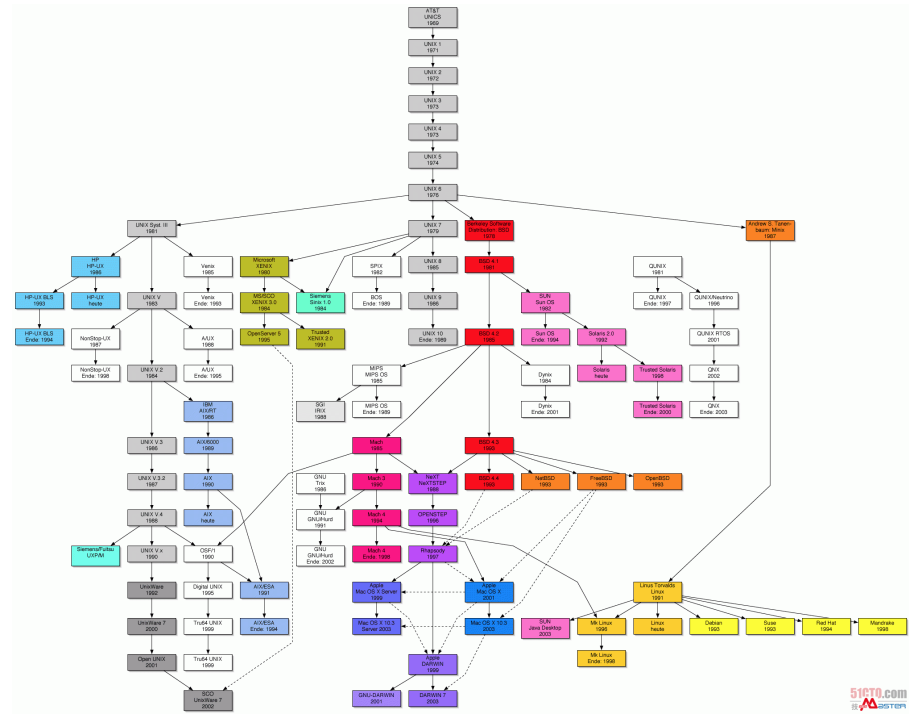
实时系统可分成两类：

- (1) 实时控制系统。当用于飞机飞行、导弹发射等的自动控制时，要求计算机能尽快处理测量系统测得的数据，及时地对飞机或导弹进行控制，或将有关信息通过显示终端提供给决策人员。当用于轧钢、石化等工业生产过程控制时，也要求计算机能及时处理由各类传感器送来的数据，然后控制相应的执行机构。
- (2) 实时信息处理系统。当用于预定飞机票、查询有关航班、航线、票价等事宜时，或当用于银行系统、情报检索系统时，都要求计算机能对终端设备发来的服务请求及时予以正确的回答。此类对响应及时性的要求稍弱于第一类。

实时操作系统的主要特点：

- (1) 及时响应。每一个信息接收、分析处理和发送的过程必须在严格的时间限制内完成。
- (2) 高可靠性。需采取冗余措施，双机系统前后台工作，也包括必要的保密措施等。

操作系统发展图谱



进程与线程

什么是进程(process)?

An executing instance of a program is called a process.

Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.

程序并不能单独运行，只有将程序装载到内存中，系统为它分配资源才能运行，而这种执行的程序就称之为进程。程序和进程的区别就在于：程序是指令的集合，它是进程运行的静态描述文本；进程是程序的一次执行活动，属于动态概念。

在多道编程中，我们允许多个程序同时加载到内存中，在操作系统的调度下，可以实现并发地执行。这是这样的设计，大大提高了CPU的利用率。进程的出现让每个用户感觉自己独享CPU，因此，进程就是为了在CPU上实现多道编程而提出的。

有了进程为什么还要线程？

进程有很多优点，它提供了多道编程，让我们感觉我们每个人都拥有自己的CPU和其他资源，可以提高计算机的利用率。很多人就不理解了，既然进程这么优秀，为什么还要线程呢？其实，仔细观察就会发现进程还是有很多缺陷的，主要体现在两点上：

- 进程只能在一个时间干一件事，如果想同时干两件事或多件事，进程就无能为力了。
- 进程在执行的过程中如果阻塞，例如等待输入，整个进程就会挂起，即使进程中有些工作不依赖于输入的数据，也将无法执行。

例如，我们在使用qq聊天，qq做为一个独立进程如果同一时间只能干一件事，那他如何实现在同一时刻即能监听键盘输入、又能监听其它人给你发的消息、同时还能把别人发的消息显示在屏幕上呢？你会说，操作系统不是有分时么？但我的亲，分时是指在不同进程间的分时呀，即操作系统处理一会你的qq任务，又切换到word文档任务上了，每个cpu时间片分给你的qq程序时，你的qq还是只能同时干一件事呀。

再直白一点，一个操作系统就像是一个工厂，工厂里面有很多个生产车间，不同的车间生产不同的产品，每个车间就相当于一个进程，且你的工厂又穷，供电不足，同一时间只能给一个车间供电，为了能让所有车间都能同时生产，你的工厂的电工只能给不同的车间分时供电，但是轮到你的qq车间时，发现只有一个干活的工

人, 结果生产效率极低, 为了解决这个问题, 应该怎么办呢?。。。没错, 你肯定想到了, 就是多加几个工人, 让几个人工人并行工作, 这每个工人, 就是线程!

什么是线程(thread)?

线程是操作系统能够进行运算调度的最小单位。它被包含在进程之中, 是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流, 一个进程中可以并发多个线程, 每条线程并行执行不同的任务

A thread is an execution context, which is all the information a CPU needs to execute a stream of instructions.

Suppose you're reading a book, and you want to take a break right now, but you want to be able to come back and resume reading from the exact point where you stopped. One way to achieve that is by jotting down the page number, line number, and word number. So your execution context for reading a book is these 3 numbers.

If you have a roommate, and she's using the same technique, she can take the book while you're not using it, and resume reading from where she stopped. Then you can take it back, and resume it from where you were.

Threads work in the same way. A CPU is giving you the illusion that it's doing multiple computations at the same time. It does that by spending a bit of time on each computation. It can do that because it has an execution context for each computation. Just like you can share a book with your friend, many tasks can share a CPU.

On a more technical level, an execution context (therefore a thread) consists of the values of the CPU's registers.

Last: threads are different from processes. A thread is a context of execution, while a process is a bunch of resources associated with a computation. A process can have one or many threads.

Clarification: the resources associated with a process include memory pages (all the threads in a process have the same view of the memory), file descriptors (e.g., open sockets), and security credentials (e.g., the ID of the user who started the process).

进程与线程的区别?

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.

Python GIL(Global Interpreter Lock)

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, 擦。。。, 那这还叫什么多线程呀? 莫如此早的下结论, 听我现场讲。

首先需要明确的一点是GIL并不是Python的特性，它是在实现Python解析器(CPython)时所引入的一个概念。就好比C++是一套语言（语法）标准，但是可以用不同的编译器来编译成可执行代码。有名的编译器例如GCC，INTEL C++，Visual C++等。Python也一样，同样一段代码可以通过CPython，PyPy，Psyco等不同的Python执行环境来执行。像其中的JPython就没有GIL。然而因为CPython是大部分环境下默认的Python执行环境。所以在很多人的概念里CPython就是Python，也就想当然的把GIL归结为Python语言的缺陷。所以这里要先明确一点：GIL并不是Python的特性，Python完全可以不依赖于GIL

这篇文章透彻的剖析了GIL对python多线程的影响，强烈推荐看一下：

<http://www.dabeaz.com/python/UnderstandingGIL.pdf>

Python threading模块

线程有2种调用方式，如下：

直接调用

```
1 import threading
2 import time
3
4 def sayhi(num): #定义每个线程要运行的函数
5
6     print("running on number:%s" %num)
7
8     time.sleep(3)
9
10 if __name__ == '__main__':
11
12     t1 = threading.Thread(target=sayhi,args=(1,)) #生成一个线程实例
13     t2 = threading.Thread(target=sayhi,args=(2,)) #生成另一个线程实例
14
15     t1.start() #启动线程
16     t2.start() #启动另一个线程
17
18     print(t1.getName()) #获取线程名
19     print(t2.getName())
```

继承式调用

```
1 import threading
2 import time
3
4
5 class MyThread(threading.Thread):
6     def __init__(self,num):
7         threading.Thread.__init__(self)
8         self.num = num
9
10     def run(self):#定义每个线程要运行的函数
11
12         print("running on number:%s" %self.num)
13
14         time.sleep(3)
15
16 if __name__ == '__main__':
17
18     t1 = MyThread(1)
19     t2 = MyThread(2)
20     t1.start()
21     t2.start()
```

Join & Daemon

Some threads do background tasks, like sending keepalive packets, or performing periodic garbage collection, or whatever. These are only useful when the main program is running, and it's okay to kill them off once the other, non-daemon, threads have exited.

Without daemon threads, you'd have to keep track of them, and tell them to exit, before your program can completely quit. By setting them as daemon threads, you can let them run and forget about them, and when your program quits, any daemon threads are killed automatically.

```
1 # *_coding:utf-8_*
2 __author__ = 'Alex Li'
```

```

3
4 import time
5 import threading
6
7
8 def run(n):
9
10     print('[%s]-----running----\n' % n)
11     time.sleep(2)
12     print('--done--')
13
14 def main():
15     for i in range(5):
16         t = threading.Thread(target=run,args=[i,])
17         t.start()
18         t.join(1)
19         print('starting thread', t.getName())
20
21
22 m = threading.Thread(target=main,args=[])
23 m.setDaemon(True) #将main线程设置为Daemon线程,它做为程序主线程的守护线程,当主线程退出时,m线程也会退
24 m.start()
25 m.join(timeout=2)
26 print("---main thread done---")

```

Note: Daemon threads are abruptly stopped at shutdown. Their resources (such as open files, database t

线程锁(互斥锁Mutex)

一个进程下可以启动多个线程，多个线程共享父进程的内存空间，也就意味着每个线程可以访问同一份数据，此时，如果2个线程同时要修改同一份数据，会出现什么状况？

```

1 import time
2 import threading
3
4 def addNum():
5     global num #在每个线程中都获取这个全局变量
6     print('--get num:', num )
7     time.sleep(1)
8     num -=1 #对此公共变量进行-1操作
9
10 num = 100 #设定一个共享变量
11 thread_list = []
12 for i in range(100):
13     t = threading.Thread(target=addNum)
14     t.start()
15     thread_list.append(t)
16
17 for t in thread_list: #等待所有线程执行完毕
18     t.join()
19
20
21 print('final num:', num )

```

正常来讲，这个num结果应该是0，但在python 2.7上多运行几次，会发现，最后打印出来的num结果不总是0，为什么每次运行的结果不一样呢？哈，很简单，假设你有A,B两个线程，此时都要对num进行减1操作，由于2个线程是并发同时运行的，所以2个线程很有可能同时拿走了num=100这个初始变量交给cpu去运算，当A线程去减完的结果是99，但此时B线程运算完的结果也是99，两个线程同时CPU运算的结果再赋值给num变量后，结果就都是99。那怎么办呢？很简单，每个线程在要修改公共数据时，为了避免自己在还没改完的时候别人也来修改此数据，可以给这个数据加一把锁，这样其它线程想修改此数据时必须等待你修改完毕并把锁释放掉后才能再访问此数据。

*注：不要在3.x上运行，不知为什么，3.x上的结果总是正确的，可能是自动加了锁

加锁版本

```

1 import time
2 import threading

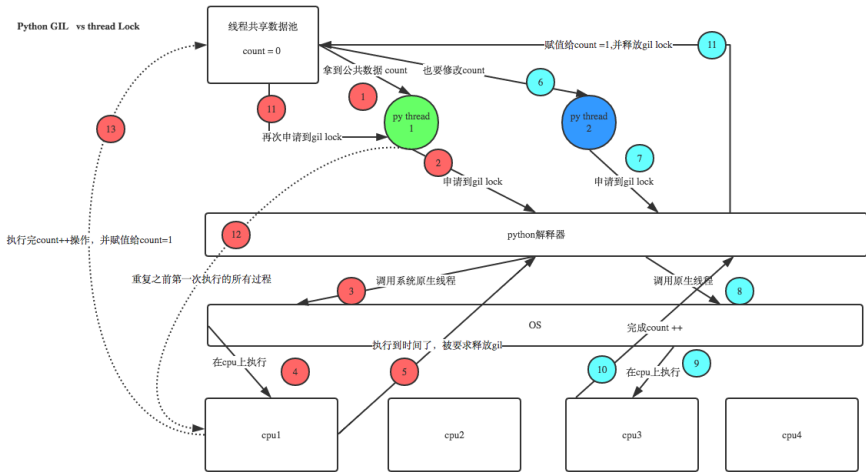
```



```
3
4 def addNum():
5     global num #在每个线程中都获取这个全局变量
6     print('--get num:',num )
7     time.sleep(1)
8     lock.acquire() #修改数据前加锁
9     num -=1 #对此公共变量进行-1操作
10    lock.release() #修改后释放
11
12 num = 100 #设定一个共享变量
13 thread_list = []
14 lock = threading.Lock() #生成全局锁
15 for i in range(100):
16     t = threading.Thread(target=addNum)
17     t.start()
18     thread_list.append(t)
19
20 for t in thread_list: #等待所有线程执行完毕
21     t.join()
22
23 print('final num:', num )
```

GIL VS Lock

机智的同学可能会问到这个问题，就是既然你之前说过了，Python已经有一个GIL来保证同一时间只能有一个线程来执行了，为什么这里还需要lock? 注意啦，这里的lock是用户级的lock,跟那个GIL没关系，具体我们通过下图来看一下+配合我现场讲给大家，就明白了。



那你又问了，既然用户程序已经自己有锁了，那为什么C python还需要GIL呢？加入GIL主要的原因是为了降低程序的开发的复杂度，比如现在的你写python不需要关心内存回收的问题，因为Python解释器帮你自动定期进行内存回收，你可以理解为python解释器里有一个独立的线程，每过一段时间它起wake up做一次全局轮询看看哪些内存数据是可以被清空的，此时你自己的程序里的线程和py解释器自己的线程是并发运行的，假设你的线程删除了一个变量，py解释器的垃圾回收线程在清空这个变量的过程中的clearing时刻，可能一个其它线程正好又重新给这个还没来得及清空的内存空间赋值了，结果就有可能新赋值的数据被删除了，为了解决类似的问题，python解释器简单粗暴的加了锁，即当一个线程运行时，其它人都不能动，这样就解决了上述的问题，这可以说是Python早期版本的遗留问题。

RLock (递归锁)

说白了就是在一个大锁中还要再包含子锁

```
1 import threading,time
2
3 def run1():
4     print("grab the first part data")
5     lock.acquire()
6     global num
7     num +=1
```

```

8     lock.release()
9     return num
10 def run2():
11     print("grab the second part data")
12     lock.acquire()
13     global num2
14     num2+=1
15     lock.release()
16     return num2
17 def run3():
18     lock.acquire()
19     res = run1()
20     print('-----between run1 and run2-----')
21     res2 = run2()
22     lock.release()
23     print(res,res2)
24
25
26 if __name__ == '__main__':
27
28     num,num2 = 0,0
29     lock = threading.RLock()
30     for i in range(10):
31         t = threading.Thread(target=run3)
32         t.start()
33
34 while threading.active_count() != 1:
35     print(threading.active_count())
36 else:
37     print('---all threads done---')
38     print(num,num2)

```

Semaphore(信号量)

互斥锁 同时只允许一个线程更改数据，而Semaphore是同时允许一定数量的线程更改数据，比如厕所所有3个坑，那最多只允许3个人上厕所，后面的人只能等里面有人出来了才能再进去。

```

1 import threading,time
2
3 def run(n):
4     semaphore.acquire()
5     time.sleep(1)
6     print("run the thread: %s\n" %n)
7     semaphore.release()
8
9 if __name__ == '__main__':
10
11     num= 0
12     semaphore = threading.BoundedSemaphore(5) #最多允许5个线程同时运行
13     for i in range(20):
14         t = threading.Thread(target=run,args=(i,))
15         t.start()
16
17 while threading.active_count() != 1:
18     pass #print threading.active_count()
19 else:
20     print('---all threads done---')
21     print(num)

```

Timer

This class represents an action that should be run only after a certain amount of time has passed

Timers are started, as with threads, by calling their `start()` method. The timer can be stopped (before its action has begun) by calling the `cancel()` method. The interval the timer will wait before executing its action may not be exactly the same as the interval specified by the user.

```

1 def hello():
2     print("hello, world")
3
4 t = Timer(30.0, hello)

```

```
5 | t.start() # after 30 seconds, "hello, world" will be printed
```

Events

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.

通过Event来实现两个或多个线程间的交互，下面是一个红绿灯的例子，即启动一个线程做交通指挥灯，生成几个线程做车辆，车辆行驶按红灯停，绿灯行的规则。

```
1 | import threading,time
2 | import random
3 | def light():
4 |     if not event.isSet():
5 |         event.set() #wait就不阻塞 #绿灯状态
6 |     count = 0
7 |     while True:
8 |         if count < 10:
9 |             print('\033[42;1m--green light on---\033[0m')
10 |         elif count <13:
11 |             print('\033[43;1m--yellow light on---\033[0m')
12 |         elif count <20:
13 |             if event.isSet():
14 |                 event.clear()
15 |             print('\033[41;1m--red light on---\033[0m')
16 |         else:
17 |             count = 0
18 |             event.set() #打开绿灯
19 |             time.sleep(1)
20 |             count +=1
21 | def car(n):
22 |     while 1:
23 |         time.sleep(random.randrange(10))
24 |         if event.isSet(): #绿灯
25 |             print("car [%s] is running.." % n)
26 |         else:
27 |             print("car [%s] is waiting for the red light.." %n)
28 | if __name__ == '__main__':
29 |     event = threading.Event()
30 |     Light = threading.Thread(target=light)
31 |     Light.start()
32 |     for i in range(3):
33 |         t = threading.Thread(target=car,args=(i,))
34 |         t.start()
```

这里还有一个event使用的例子，员工进公司门要刷卡，我们这里设置一个线程是“门”，再设置几个线程为“员工”，员工看到门没打开，就刷卡，刷完卡，门开了，员工就可以通过。

[View Code](#)

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

class queue.Queue(*maxsize=0*) #先入先出

class queue.LifoQueue(*maxsize=0*) #last in first out

class queue.PriorityQueue(*maxsize=0*) #存储数据时可设置优先级的队列

Constructor for a priority queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

The lowest valued entries are retrieved first (the lowest valued entry is the one returned by `sorted(list(entries))[0]`). A typical pattern for entries is a tuple in the form: (priority_number, data).

exception queue.Empty

Exception raised when non-blocking `get()` (or `get_nowait()`) is called on a `Queue` object which is empty.

exception queue.Full

Exception raised when non-blocking `put()` (or `put_nowait()`) is called on a `Queue` object which is full.

Queue.qsize()

Queue.empty() #return True if empty

Queue.full() # return True if full

Queue.put(*item*, *block=True*, *timeout=None*)

Put *item* into the queue. If optional args *block* is true and *timeout* is None (the default), block if necessary until a free slot is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the `Full` exception if no free slot was available within that time. Otherwise (*block* is false), put an item on the queue if a free slot is immediately available, else raise the `Full` exception (*timeout* is ignored in that case).

Queue.put_nowait(*item*)

Equivalent to `put(item, False)`.

Queue.get(*block=True*, *timeout=None*)

Remove and return an item from the queue. If optional args *block* is true and *timeout* is None (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the `Empty` exception if no item was available within that time. Otherwise (*block* is false), return an item if one is immediately available, else raise the `Empty` exception (*timeout* is ignored in that case).

Queue.get_nowait()

Equivalent to `get(False)`.

Two methods are offered to support tracking whether enqueued tasks have been fully processed by daemon consumer threads.

Queue.task_done()

Indicate that a formerly enqueued task is complete. Used by queue consumer threads. For each `get()` used to fetch a task, a subsequent call to `task_done()` tells the queue that the processing on the task is complete.

If a `join()` is currently blocking, it will resume when all items have been processed (meaning that a `task_done()` call was received for every item that had been `put()` into the queue).

Raises a `ValueError` if called more times than there were items placed in the queue.

Queue.join() block直到queue被消费完毕

生产者消费者模型

在并发编程中使用生产者和消费者模式能够解决绝大多数并发问题。该模式通过平衡生产线程和消费线程的工作能力来提高程序的整体处理数据的速度。

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。

下面来学习一个最基本的生产者消费者模型的例子

```
1 import threading
2 import queue
3
4 def producer():
5     for i in range(10):
6         q.put("骨头 %s" % i)
7
8     print("开始等待所有的骨头被取走...")
9     q.join()
10    print("所有的骨头被取完了...")
11
12
13 def consumer(n):
14
15     while q.qsize() > 0:
16
17         print("%s 取到" % n, q.get())
18         q.task_done() #告知这个任务执行完了
19
20
21 q = queue.Queue()
22
23
24
25 p = threading.Thread(target=producer,)
26 p.start()
27
28 c1 = consumer("李闯")
```

```
1 import time, random
2 import queue, threading
3 q = queue.Queue()
4 def Producer(name):
5     count = 0
6     while count < 20:
7         time.sleep(random.randrange(3))
8         q.put(count)
9         print('Producer %s has produced %s baozi..' % (name, count))
10        count += 1
11 def Consumer(name):
12     count = 0
13     while count < 20:
14         time.sleep(random.randrange(4))
15         if not q.empty():
16             data = q.get()
17             print(data)
18             print('\033[32;1mConsumer %s has eat %s baozi...\033[0m' % (name, data))
19         else:
20             print("-----no baozi anymore----")
21         count += 1
22 p1 = threading.Thread(target=Producer, args=('A',))
23 c1 = threading.Thread(target=Consumer, args=('B',))
24 p1.start()
25 c1.start()
```

多进程multiprocessing

`multiprocessing` 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.

```
1 from multiprocessing import Process
2 import time
3 def f(name):
4     time.sleep(2)
5     print('hello', name)
6
7 if __name__ == '__main__':
8     p = Process(target=f, args=('bob',))
9     p.start()
10    p.join()
```

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

```
1 from multiprocessing import Process
2 import os
3
4 def info(title):
5     print(title)
6     print('module name:', __name__)
7     print('parent process:', os.getppid())
8     print('process id:', os.getpid())
9     print("\n\n")
10
11 def f(name):
12     info('\033[31;1mfunction f\033[0m')
13     print('hello', name)
14
15 if __name__ == '__main__':
16     info('\033[32;1mmain process line\033[0m')
17     p = Process(target=f, args=('bob',))
18     p.start()
19     p.join()
```

进程间通讯

不同进程间内存是不共享的，要想实现两个进程间的数据交换，可以用以下方法：

Queues

使用方法跟threading里的queue差不多

```
1 from multiprocessing import Process, Queue
2
3 def f(q):
4     q.put([42, None, 'hello'])
5
6 if __name__ == '__main__':
7     q = Queue()
8     p = Process(target=f, args=(q,))
9     p.start()
10    print(q.get())    # prints "[42, None, 'hello']"
11    p.join()
```

Pipes

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

```
1 from multiprocessing import Process, Pipe
2
3 def f(conn):
4     conn.send([42, None, 'hello'])
5     conn.close()
```



```

6
7 if __name__ == '__main__':
8     parent_conn, child_conn = Pipe()
9     p = Process(target=f, args=(child_conn,))
10    p.start()
11    print(parent_conn.recv()) # prints "[42, None, 'hello']"
12    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.

Managers

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,

```

1 from multiprocessing import Process, Manager
2
3 def f(d, l):
4     d[1] = '1'
5     d['2'] = 2
6     d[0.25] = None
7     l.append(1)
8     print(l)
9
10 if __name__ == '__main__':
11     with Manager() as manager:
12         d = manager.dict()
13
14         l = manager.list(range(5))
15         p_list = []
16         for i in range(10):
17             p = Process(target=f, args=(d, l))
18             p.start()
19             p_list.append(p)
20         for res in p_list:
21             res.join()
22
23         print(d)
24         print(l)

```

进程同步

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

```

1 from multiprocessing import Process, Lock
2
3 def f(l, i):
4     l.acquire()
5     try:
6         print('hello world', i)
7     finally:
8         l.release()
9
10 if __name__ == '__main__':
11     lock = Lock()
12
13     for num in range(10):
14         Process(target=f, args=(lock, num)).start()

```

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

```
1 from multiprocessing import Process, Pool
2 import time
3
4 def Foo(i):
5     time.sleep(2)
6     return i+100
7
8 def Bar(arg):
9     print('-->exec done:',arg)
10
11 pool = Pool(5)
12
13 for i in range(10):
14     pool.apply_async(func=Foo, args=(i,), callback=Bar)
15     #pool.apply(func=Foo, args=(i,))
16
17 print('end')
18 pool.close()
19 pool.join()#进程池中进程执行完毕后再关闭，如果注释，那么程序直接关闭。
```

作业需求：

题目:简单主机批量管理工具

需求：

1. 主机分组
2. 主机信息配置文件用configparser解析
3. 可批量执行命令、发送文件，结果实时返回，执行格式如下
 1. batch_run -h h1,h2,h3 -g web_clusters,db_servers -cmd "df -h"
 2. batch_scp -h h1,h2,h3 -g web_clusters,db_servers -action put -local test.py -remote /tmp/
4. 主机用户名密码、端口可以不同
5. 执行远程命令使用paramiko模块
6. 批量命令需使用multiprocessing并发

分类: [Python自动化开发之路](#)

好文要顶

已关注

收藏该文



金角大王

关注 - 5

粉丝 - 10911

我在关注他 [取消关注](#)

7

0

posted @ 2016-03-01 13:18 金角大王 阅读(41634) 评论(14) 编辑 收藏

评论列表

#1楼 2016-12-29 22:26 真是好人

[回复](#) [引用](#)

总结的真详细

支持(0) 反对(0)

#2楼 2017-03-06 13:22 ~艾比郎~

[回复](#) [引用](#)

太牛逼了

	支持(0) 反对(0)
<div>#3楼 2017-07-12 15:54 航大</div> <div>写的很详细，不过英文不好，英文部分还需要谷歌翻译</div>	<div>回复 引用</div> <div>支持(0) 反对(0)</div>
<div>#4楼 2017-12-06 11:08 胡嘉衍</div> <div>总结的很牛逼</div>	<div>回复 引用</div> <div>支持(0) 反对(0)</div>
<div>#5楼 2017-12-10 00:17 周资源</div> <div>写的很好</div>	<div>回复 引用</div> <div>支持(0) 反对(0)</div>
<div>#6楼 2018-01-03 15:14 yyshuke</div> <div>很喜欢听Alex将东西，不仅教知识，更注重对学生的人品教育</div>	<div>回复 引用</div> <div>支持(2) 反对(0)</div>
<div>#7楼 2018-01-05 16:55 MeggieJ</div> <div>我在执行Manager时出现下面的异常，我是直接复制上面的Manager代码在pycharm上运行的，python版本是3.6，不知道什么原因，哪位大神帮忙回答一下啊，跪谢了！ Traceback (most recent call last): File "C:/Users/Administrator/Desktop/python3/day10/进程与manager.py", line 34, in <module> d = manager.dict() File "D:\Python36\lib\multiprocessing\managers.py", line 662, in temp token, exp = self._create(typeid, *args, **kwargs) File "D:\Python36\lib\multiprocessing\managers.py", line 554, in _create conn = self._Client(self._address, authkey=self._authkey) File "D:\Python36\lib\multiprocessing\connection.py", line 493, in Client answer_challenge(c, authkey) File "D:\Python36\lib\multiprocessing\connection.py", line 735, in answer_challenge digest = hmac.new(authkey, message, 'md5').digest() AttributeError: module 'hmac' has no attribute 'new'</div>	<div>回复 引用</div> <div>支持(0) 反对(0)</div>
<div>#8楼 2018-03-18 12:06 不识i</div> <div>强烈推荐我们看的是全英文的啊,完全看不懂</div>	<div>回复 引用</div> <div>支持(0) 反对(0)</div>
<div>#9楼 2018-06-11 15:40 vitoi</div> <div>汝之秀，吾何时能及？！</div>	<div>回复 引用</div> <div>支持(0) 反对(0)</div>
<div>#10楼 2018-07-19 10:58 hexintong</div> <div>大王牛逼。</div>	<div>回复 引用</div> <div>支持(0) 反对(0)</div>
<div>#11楼 2018-08-06 14:49 Elience</div> <div>Alex老师，您好！上边的继承式调用中，类里边写多个函数怎么还是只执行一个run()函数，谢谢！！</div>	<div>回复 引用</div> <div>支持(0) 反对(0)</div>
<div>#12楼 2018-10-25 17:23 Happy_Bing</div> <div>Hello Alex对于你演示的线程锁，我代码更改了一下，这个例子很容易就演示出来你说的那种现象了，并且Python3也可以复制出来这种现象~</div>	<div>回复 引用</div>

```
1  __author__ = 'Brook Zhao'
2
3  import threading
4  import time
5
6  num=0
7  thread_list=[]
8  lock=threading.Lock()
9
10 def holdon():
11     time.sleep(0.01)
12     return 1
13
14 def addNum():
15     global num
16     print("--get num",num)
17     time.sleep(1)
18     #lock.acquire()
19     num=num+holdon()
20     #lock.release()
```

```
21
22 for i in range(100):
23     t=threading.Thread(target=addNum)
24     t.start()
25     thread_list.append(t)
26
27 for t in thread_list:
28     t.join()
29
30 print("final num",num)
```

支持(0) 反对(0)

#13楼 2018-11-20 16:21 蔚蓝的蓝

回复 引用

老哥是有多懒，连翻译都不想翻译，就直接粘英文文档

支持(0) 反对(0)

#14楼 2019-06-24 22:19 我也不想这么菜

回复 引用





作业，作业，作业 重要的事情说三遍

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

发表评论

编辑 预览

B    

支持 Markdown

提交评论

退出 订阅评论

[Ctrl+Enter快捷键提交]

- 【推荐】超50万行VC++源码：大型组态工控、电力仿真CAD与GIS源码库
- 【推荐】了不起的开发者，挡不住的华为，园子里的品牌专区
- 【推荐】开放下载 | 多场景多实战《阿里云AIoT造物秘籍》，值得收藏！

相关博文：

- Python之路,Day9,进程、线程、协程篇
 - Python之路,Day9,进程、线程、协程篇
 - Python之路,Day9, 进程、线程、协程篇
 - python之路-进程、线程、协程篇
 - python之路-进程、线程
- » 更多推荐...

最新 IT 新闻：

- 黎巴嫩首都爆炸能量有多大？物理学家看视频计算：300吨TNT！
 - 马斯克到底从特斯拉赚了多少钱？他是最富的穷光蛋
 - 特朗普封禁微信，张小龙至少有两张牌可打
 - “抖音点赞员”月入1万+，靠谱吗？
 - 减持500亿元、死磕马斯克，贝索斯在下一盘大棋？
- » 更多新闻...

