1. **Python多线程**

**第一部分**

1. **进程的概念**

进程（Process）是正在运行的程序的实例，进程是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是[操作系统](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F" \t "https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B/_blank)结构的基础。在早期面向进程设计的计算机结构中，进程是程序的基本执行实体；在当代面向线程设计的计算机结构中，进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体。

有两点需要注意:

第一：进程是一个实体。每一个进程都有它自己的地址空间；

第二：进程是一个“执行中的程序”。程序是一个没有生命的实体，只有[处理](https://baike.baidu.com/item/%E5%A4%84%E7%90%86" \t "https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B/_blank)器赋予程序生命时（操作系统执行时），它才能成为一个活动的实体，我们称其为[进程](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B" \t "https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B/_blank)。

1. **多线程的基本概念**

线程，有时被称为轻量级进程(Lightweight Process，LWP），是程序执行流的最小单元。一个标准的线程由线程ID，当前指令[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88" \t "https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/_blank)(PC），[寄存器](https://baike.baidu.com/item/%E5%AF%84%E5%AD%98%E5%99%A8" \t "https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/_blank)集合和[堆栈](https://baike.baidu.com/item/%E5%A0%86%E6%A0%88" \t "https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/_blank)组成。另外，线程是进程中的一个实体，是被系统独立调度和分派的基本单位，线程自己不拥有系统资源，只拥有一点儿在运行中必不可少的资源，但它可与同属一个进程的其它线程共享进程所拥有的全部资源。一个线程可以创建和撤消另一个线程，同一进程中的多个线程之间可以并发执行。由于线程之间的相互制约，致使线程在运行中呈现出间断性。线程也有[就绪](https://baike.baidu.com/item/%E5%B0%B1%E7%BB%AA" \t "https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/_blank)、[阻塞](https://baike.baidu.com/item/%E9%98%BB%E5%A1%9E" \t "https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/_blank)和[运行](https://baike.baidu.com/item/%E8%BF%90%E8%A1%8C" \t "https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/_blank)三种基本状态。就绪状态是指线程具备运行的所有条件，逻辑上可以运行，在等待处理机；运行状态是指线程占有处理机正在运行；阻塞状态是指线程在等待一个事件（如某个信号量），逻辑上不可执行。每一个程序都至少有一个线程，若程序只有一个线程，那就是程序本身。

线程是程序中一个单一的顺序控制流程。进程内有一个相对独立的、可调度的执行单元，是系统独立调度和分派CPU的基本单位指令[运行](https://baike.baidu.com/item/%E8%BF%90%E8%A1%8C" \t "https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/_blank)时的程序的调度单位。在单个程序中同时运行多个线程完成不同的工作，称为[多线程](https://baike.baidu.com/item/%E5%A4%9A%E7%BA%BF%E7%A8%8B" \t "https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/_blank)。

1. **进程和线程关系及区别**
2. 定义

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。

2、关系

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。

相对进程而言，线程是一个更加接近于执行体的概念，它可以与同进程中的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。

3、区别

进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

1) 简而言之,一个程序至少有一个进程,一个进程至少有一个线程.

2) 线程的划分尺度小于进程，使得多线程程序的并发性高。

3) 另外，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。

4) 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。

5) 从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

4.优缺点

线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源的管理和保护；而进程正相反。

**第二部分**

1. **多线程的优点**

多线程类似于同时执行多个不同程序，多线程运行有如下优点：

1、使用线程可以把占据长时间的程序中的任务放到后台去处理。

2、用户界面可以更加吸引人，这样比如用户点击了一个按钮去触发某些事件的处理，可以弹出一个进度条来显示处理的进度。

3、程序的运行速度可能加快

4、在一些等待的任务实现上如用户输入、文件读写和网络收发数据等，线程就比较有用了。这种情况下我们可以释放一些珍贵的资源如内存占用等等。

1. **线程的分类**

内核线程：由操作系统内核创建和撤销。

用户线程：不需要内核支持而在用户程序中实现的线程。

1. **线程模块**

Python3 线程中常用的两个模块为：

1、 \_thread

2、 threading(推荐使用)

thread 模块已被废弃。用户可以使用 threading 模块代替。所以，在 Python3 中不能再使用"thread" 模块。为了兼容性，Python3 将 thread 重命名为 "\_thread"。

**6.1 、threading模块**

Python3 通过两个标准库 \_thread 和 threading 提供对线程的支持。

\_thread 提供了低级别的、原始的线程以及一个简单的锁，它相比于 threading 模块的功能还是比较有限的。

threading 模块除了包含 \_thread 模块中的所有方法外，还提供的其他方法：

1、threading.currentThread(): 返回当前的线程变量。

2、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。

3、threading.activeCount(): 返回正在运行的线程数量，与len(threading.enumerate())有相同的结果。

除了threading 模块的方法外，线程模块同样提供了Thread类来处理线程，Thread类提供了以下方法:

1、run(): 用以表示线程活动的方法。

2、start():启动线程活动。

3、join([time]): 等待至线程中止。这阻塞调用线程直至线程的join() 方法被调用中止-正常退出或者抛出未处理的异常-或者是可选的超时发生。

4、isAlive(): 返回线程是否活动。

5、getName(): 返回线程名。

6、setName(): 设置线程名。

实例

*#encoding=utf-8***import** threading  
*#打印主线程*print(threading.main\_thread())  
*#判断主线程是否运行*print(threading.main\_thread().isAlive())  
*#获取当前线程的名字*print(threading.currentThread().getName())  
*#计算当前线程活动个数*print(threading.activeCount())  
*#枚举当前线程***for** t **in** threading.enumerate():  
 print(type(t))  
 print(t.getName())

执行结果：

<\_MainThread(MainThread, started 8852)>

True

MainThread

1

<class 'threading.\_MainThread'>

MainThread

**第三部分**

1. **使用多线程**

Python中使用线程有两种方式：函数或者用类来包装线程对象。

7、1 函数式使用线程

语法：

函数式：调用 \_thread 模块中的start\_new\_thread()函数来产生新线程。语法如下:

\_thread.start\_new\_thread ( function, args[, kwargs] )

参数说明:

 function - 线程函数。

 args - 传递给线程函数的参数,它必须是个tuple类型。

kwargs - 可选参数。

实例1：

不使用多线程的例子：

*#encoding=utf-8***import** \_thread  
**import** time  
*#定义一个函数，用来显示线程的名字和当前时间***def** showTime2(name,delayTime):  
 count=1;  
 **while** count<=6:  
 *#时间延后delayTime，单位为秒* time.sleep(delayTime)  
 print(**"执行体的名字%s，现在的时间 %s"** %(name,time.ctime(time.time())))  
 count=count+1  
*#不使用多线程*print(**"不使用多线程时，开始执行，当前时间为%s"** %(time.ctime(time.time())))  
showTime2(**"thread02"**,4)  
showTime2(**"thread01"**,2)  
showTime2(**"thread03"**,1)  
print(**"不使用多线程时，执行结束，当前时间为%s"** %(time.ctime(time.time())))

特点：1、每时每刻，只能运行一个，每个执行体都是串行的

2、共花费42秒左右

实例2:

*#encoding=utf-8***import** \_thread  
**import** time  
m=0  
*#定义一个函数，用来显示线程的名字和当前时间***def** showTime2(name,delayTime):  
 count=1;  
 **while** count<=6:  
 *#时间延后delayTime，单位为秒* time.sleep(delayTime)  
 print(**"执行体的名字%s，现在的时间 %s"** %(name,time.ctime(time.time())))  
 count=count+1  
 **global** m  
 m=m+1  
 print(**"{0}结束"**.format(name))  
 print(m)  
*#创建三个线程*begin=time.time()  
print(**"使用多线程时，开始执行，当前时间为%s"** %(time.ctime(begin)))  
**try**:  
 \_thread.start\_new\_thread(showTime2, (**"thread02"**,4))  
 \_thread.start\_new\_thread(showTime2, (**"thread01"**,2))  
 \_thread.start\_new\_thread(showTime2, (**"thread03"**,1))  
 end=time.time()  
 print(**"使用多线程时，结束执行，当前时间为%s"** %(time.ctime(end)))  
**except**:  
 print(**"error"**)  
**while** 1:  
 **if** m==18:  
 **break  
 else**:  
 **pass**特点：1、在某一时刻，可以运行多个执行体，每个执行体都是并行的

2、共花费20秒左右

从上面的实验可以得出这样的结论：

多线程可以使多个线程并发执行，比如上面的例子，三个线程同时执行，大大减少了执行等待的时间，提高CPU的使用率和用户体验。

**7、2 类来包装线程对象**

使用 threading 模块创建线程,我们可以通过直接从 threading.Thread 继承创建一个新的子类，并实例化后调用 start() 方法启动新线程，即它调用了线程的 run() 方法：

*# encoding=utf-8***import** threading  
**import** time  
**class** selfThread(threading.Thread):  
 **def** \_\_init\_\_(self, tname, dtime):  
 threading.Thread.\_\_init\_\_(self)  
 self.name = tname  
 self.delayTime = dtime  
  
 **def** run(self):  
 print(**"多线程%s开始执行"** % (self.name))  
 showTime2(self.name, self.delayTime)  
 print(**"多线程%s结束执行"** % (self.name))  
  
  
*# 定义一个函数，用来显示线程的名字和当前时间***def** showTime2(name, delayTime):  
 count =1;  
 **while** count <= 6:  
 *# 时间延后delayTime，单位为秒* time.sleep(delayTime)  
 print(**"执行体的名字%s，现在的时间 %s"** % (name, time.ctime(time.time())))  
 count = count + 1  
  
  
thread1 = selfThread(**"thread01"**, 4)  
thread2 = selfThread(**"thread02"**, 2)  
thread3 = selfThread(**"thread03"**, 1)  
  
thread1.start()  
thread2.start()  
thread3.start()

执行结果：

多线程thread01开始执行

多线程thread02开始执行

多线程thread03开始执行

多线程的名字thread03，现在的时间 Tue Nov 28 16:05:05 2017

多线程的名字thread02，现在的时间 Tue Nov 28 16:05:06 2017

多线程的名字thread03，现在的时间 Tue Nov 28 16:05:06 2017

多线程的名字thread03，现在的时间 Tue Nov 28 16:05:07 2017

多线程的名字thread01，现在的时间 Tue Nov 28 16:05:08 2017

多线程的名字thread02，现在的时间 Tue Nov 28 16:05:08 2017

多线程的名字thread03，现在的时间 Tue Nov 28 16:05:08 2017

多线程的名字thread03，现在的时间 Tue Nov 28 16:05:09 2017

多线程的名字thread02，现在的时间 Tue Nov 28 16:05:10 2017

多线程的名字thread03，现在的时间 Tue Nov 28 16:05:10 2017

多线程thread03结束执行

多线程的名字thread01，现在的时间 Tue Nov 28 16:05:12 2017

多线程的名字thread02，现在的时间 Tue Nov 28 16:05:12 2017

多线程的名字thread02，现在的时间 Tue Nov 28 16:05:14 2017

多线程的名字thread01，现在的时间 Tue Nov 28 16:05:16 2017

多线程的名字thread02，现在的时间 Tue Nov 28 16:05:16 2017

多线程thread02结束执行

多线程的名字thread01，现在的时间 Tue Nov 28 16:05:20 2017

多线程的名字thread01，现在的时间 Tue Nov 28 16:05:24 2017

多线程的名字thread01，现在的时间 Tue Nov 28 16:05:28 2017

多线程thread01结束执行

**第四部分**

**八、多线程同步**

多线程的优势在于可以同时运行多个任务（至少感觉起来是这样）。但是当线程需要共享数据时，可能存在数据不同步的问题。考虑这样一种情况：一个列表里所有元素都是0，线程"set"从后向前把所有元素改成1，而线程"print"负责从前往后读取列表并打印。那么，可能线程"set"开始改的时候，线程"print"便来打印列表了，输出就成了一半0一半1，这就是数据的不同步。

为了避免数据不同步的情况，引入了锁(threading.Lock())的概念。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（锁定）和 release（未锁定） 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。

锁有两种状态——锁定和未锁定。每当一个线程比如"set"要访问共享数据时，必须先获得锁定；如果已经有别的线程比如"print"获得锁定了，那么就让线程"set"暂停，也就是同步阻塞；等到线程"print"访问完毕，释放锁以后，再让线程"set"继续。

经过这样的处理，打印列表时要么全部输出0，要么全部输出1，不会再出现一半0一半1的尴尬场面。

开发实例：

有余票2张，4个人同时抢，如果不用多线程同步，编码如下：

*# encoding=utf-8***import** threading  
**import** time  
count = 2  
  
**class** saleTicketsThread(threading.Thread):  
 **def** \_\_init\_\_(self, name, time):  
 threading.Thread.\_\_init\_\_(self)  
 self.name = name  
 self.count = count  
 self.time = time  
  
 **def** run(self):  
 print(**"开启线程： "** + self.name)  
 saleTickets(self)  
  
**def** saleTickets(self):  
 **global** count  
 **if** count >= 1:  
 *# 这个很重要，查询休息一下 ，在休息的过程中，票已经被人抢了* time.sleep(self.time)  
 count = count - 1  
 print(**"%s已经买到票,还有余票%s"** % (self.name, str(count)))  
 **else**:  
 print(**"余票不足,%s没有买到票"** % (self.name))  
  
threadLock = threading.Lock()  
*# 创建新线程*thread1 = saleTicketsThread(**"Thread-1"**, 1)  
thread2 = saleTicketsThread(**"Thread-2"**, 1)  
thread3 = saleTicketsThread(**"Thread-3"**, 1)  
thread4 = saleTicketsThread(**"Thread-4"**, 1)  
*# 开启新线程*thread1.start()  
thread2.start()  
thread3.start()  
thread4.start()

执行结果：

开启线程： Thread-1

开启线程： Thread-2

开启线程： Thread-3

开启线程： Thread-4

Thread-2已经买到票,还有余票0

Thread-1已经买到票,还有余票0

Thread-3已经买到票,还有余票-1

Thread-4已经买到票,还有余票-2

这显然是错误的。

同样的代码，采用同步机制：

#encoding=utf-8

import threading

import time

count=2

class saleTicketsThread (threading.Thread):

def \_\_init\_\_(self, name,time):

threading.Thread.\_\_init\_\_(self)

self.name = name

self.count = count

self.time=time

def run(self):

print ("开启线程： " + self.name)

# 获取锁，用于线程同步

threadLock.acquire()

saleTickets(self)

# 释放锁，开启下一个线程

threadLock.release()

def saleTickets(self):

global count

if count>=1:

#这个很重要，查询休息一下 ，在休息的过程中，票已经被人抢了

time.sleep(self.time)

count=count-1

print("%s已经买到票,还有余票%s" %(self.name,str(count)))

else:

print("余票不足,%s没有买到票" %(self.name))

threadLock = threading.Lock()

# 创建新线程

thread1 = saleTicketsThread("Thread-1",1)

thread2 =saleTicketsThread("Thread-2",1)

thread3 = saleTicketsThread("Thread-3",1)

thread4 = saleTicketsThread("Thread-4",1)

# 开启新线程

thread1.start()

thread2.start()

thread3.start()

thread4.start()

执行结果：

开启线程： Thread-1

开启线程： Thread-2

开启线程： Thread-3

开启线程： Thread-4

Thread-1已经买到票,还有余票1

Thread-2已经买到票,还有余票0

余票不足,Thread-3没有买到票

余票不足,Thread-4没有买到票

**第五部分**

使用多线程模拟售票过程：

*#encoding=utf-8***import** threading  
**import** time  
**import** random  
**class** Worker(threading.Thread):  
 *'''售票员'''* **def** \_\_init\_\_(self, wait\_num=5, index=0):  
 threading.Thread.\_\_init\_\_(self)  
 self.wait\_num = wait\_num *# 当前排队人数* self.setName(**'窗口'** + str(index)) *# 窗口号* **def** run(self):  
 **global** counter, mutex  
 **while** counter **and** self.wait\_num: *# 若有余票且有人排队  
 # ============================  
 # 窗口问询  
 # ============================* time.sleep(random.randrange(2, 10)) *# 比较耗时，时间随机  
 # ============================  
 # 出票  
 # ============================* mutex.acquire() *# 锁住①* **if** counter == 0: *# 如果票数为0* mutex.release() *# 解锁③ -----> 这句很重要！！！  
 #print(self.getName(), '：sorry，票已售完')* **print**(**'{},sorry，票已售完'**.format(self.getName()))  
 **break** counter = counter - 1 *# 票数减一  
 # print('{}：当前余票 {} 张'.format(self.getName(), counter))* **print**(**'{}：当前余票 {} 张'**.format(self.getName(), counter))  
 mutex.release() *# 解锁③  
 # ============================  
 # 排队人数变化  
 # ============================* self.wait\_num -= 1 *# 排队人数减一* self.wait\_num += random.randrange(0, 2) *# 排队人数随机增加***if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 *# 剩余车票数* counter = 20  
 *# 创建锁* mutex = threading.Lock()  
 *# 开4个售票窗口* workers = []  
 **for** i **in** range(4):  
 wait\_num = random.randrange(2, 10) *# 窗口前排队人数随机* workers.append(Worker(wait\_num, i + 1))  
 *# 开始售票* **for** w **in** workers:  
 w.start()  
  
 *# 阻塞主程/后台静默？自己选  
 #for w in workers:  
 # w.join()*

**第六部分**

1. **生产-消费模型**

我们已经知道，对公共资源进行互斥访问，可以使用Lock上锁，或者使用RLock去重入锁。但是这些都只是方便于处理简单的同步现象，我们甚至还不能很合理的去解决使用Lock锁带来的死锁问题。要解决更复杂的同步问题，就必须考虑别的办法了。

threading提供的Condition对象提供了对复杂线程同步问题的支持。

Condition被称为条件变量，除了提供与Lock类似的acquire和release方法外，还提供了wait和notify方法。

使用Condition的主要方式为：

线程首先acquire一个条件变量，然后判断一些条件。如果条件不满足则wait；如果条件满足，进行一些 处理改变条件后，通过notify方法通知其他线程，其他处于wait状态的线程接到通知后会重新判断条件。不断的重复这一过程，从而解决复杂的同步问 题。

下面我们通过很著名的“生产者-消费者”模型来来演示下，在Python中使用Condition实现复杂同步。

生产者和消费者，各一个线程，双方将会围绕products来产生同步问题，首先是2个生成者生产products ，而接下来的10个消费者将会消耗products。

*#encoding=utf-8***import** threading  
**import** time  
condition = threading.Condition()  
products = 0  
**class** Producer(threading.Thread):  
 *'''生产者'''* ix = [0] *# 生产者实例个数  
 # 闭包，必须是数组，不能直接 ix = 0* **def** \_\_init\_\_(self, ix=0):  
 threading.Thread.\_\_init\_\_(self)  
 self.ix[0] += 1  
 self.setName(**'生产者'** + str(self.ix[0]))  
 **def** run(self):  
 **global** condition, products  
 **while** True:  
 **if** condition.acquire():  
 **if** products < 10:  
 products += 1;  
 **print**(**"{}：库存不足,我努力生产了1件产品，现在产品总数量 {}"**.format(self.getName(), products))  
 condition.notify()  
 **else**:  
 **print**(**"{}：库存充足,让我休息会儿，现在产品总数量 {}"**.format(self.getName(), products))  
 condition.wait();  
 condition.release()  
 time.sleep(2)  
**class** Consumer(threading.Thread):  
 *'''消费者'''* ix = [0] *# 消费者实例个数  
 # 闭包，必须是数组，不能直接 ix = 0* **def** \_\_init\_\_(self):  
 threading.Thread.\_\_init\_\_(self)  
 self.ix[0] += 1  
 self.setName(**'消费者'** + str(self.ix[0]))  
 **def** run(self):  
 **global** condition, products  
 **while** True:  
 **if** condition.acquire():  
 **if** products > 1:  
 products -= 1  
 **print**(**"{}：我消费了1件产品，现在产品数量 {}"**.format(self.getName(), products))  
 condition.notify()  
 **else**:  
 **print**(**"{}：只剩下1件产品，我停止消费。现在产品数量 {}"**.format(self.getName(), products))  
 condition.wait();  
 condition.release()  
 time.sleep(2)  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 **for** i **in** range(2):  
 p = Producer()  
 p.start()  
 **for** i **in** range(10):  
 c = Consumer()  
 c.start()

**第七章 Python中多线程的缺陷**

“Python下多线程是鸡肋，推荐使用多进程！”那当然有同学会问了，为啥？

1、GIL是什么？

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。

2、每个CPU在同一时间只能执行一个线程（在单核CPU下的多线程其实都只是并发，不是并行，并发和并行从宏观上来讲都是同时处理多路请求的概念。但并发和并行又有区别，并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔内发生。）

在Python多线程下，每个线程的执行方式：

1、获取GIL：执行代码直到sleep或者是python虚拟机将其挂起。

2、释放GIL：可见，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。

在Python2.x里，GIL的释放逻辑是当前线程遇见IO操作或者ticks计数达到100（ticks可以看作是Python自身的一个计数器，专门做用于GIL，每次释放后归零，这个计数可以通过 sys.setcheckinterval 来调整），进行释放。而每次释放GIL锁，线程进行锁竞争、切换线程，会消耗资源。并且由于GIL锁存在，python里一个进程永远只能同时执行一个线程(拿到GIL的线程才能执行)，这就是为什么在多核CPU上，python的多线程效率并不高。

那么是不是python的多线程就完全没用了呢？

在这里我们进行分类讨论：

1、CPU密集型代码(各种循环处理、计数等等)，在这种情况下，由于计算工作多，ticks计数很快就会达到阈值，然后触发GIL的释放与再竞争（多个线程来回切换当然是需要消耗资源的），所以python下的多线程对CPU密集型代码并不友好。

2、IO密集型代码(文件处理、网络爬虫等)，多线程能够有效提升效率(单线程下有IO操作会进行IO等待，造成不必要的时间浪费，而开启多线程能在线程A等待时，自动切换到线程B，可以不浪费CPU的资源，从而能提升程序执行效率)。

所以python的多线程对IO密集型代码比较友好。

而在python3.x中，GIL不使用ticks计数，改为使用计时器（执行时间达到阈值后，当前线程释放GIL），这样对CPU密集型程序更加友好，但依然没有解决GIL导致的同一时间只能执行一个线程的问题，所以效率依然不尽如人意。

多核性能

多核多线程比单核多线程更差，原因是单核下多线程，每次释放GIL，唤醒的那个线程都能获取到GIL锁，所以能够无缝执行，但多核下，CPU0释放GIL后，其他CPU上的线程都会进行竞争，

但GIL可能会马上又被CPU0拿到，导致其他几个CPU上被唤醒后的线程会醒着等待到切换时间后又进入待调度状态，这样会造成线程颠簸(thrashing)，导致效率更低

多进程为什么不会这样？

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，所以在python中，

多进程的执行效率优于多线程(仅仅针对多核CPU而言)。

所以在这里说结论：多核下，想做并行提升效率，比较通用的方法是使用多进程，能够有效提高执行效率。