python-多进程和多线程

讲师：尚玉杰

CONTENTS

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec luctus nibh sit amet sem vulputate venenatis bibendum orci pulvinar.

多任务

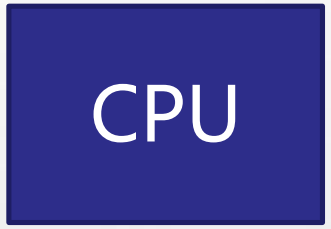
PART ONE

多任务处理

- 多任务处理：使得计算机可以同时处理多个任务
  - 听歌的同时QQ聊天、办公、下载文件

- 实现方式：多进程、多线程

串行
并发
并行



多进程

PART TWO

多进程

- 程序:是一个指令的集合
- 进程:正在执行的程序；或者说：当你运行一个程序，你就启动了一个进程
  - 编写完的代码，没有运行时，称为程序，正在运行的代码，称为进程
  - 程序是死的(静态的)，进程是活的(动态的)

- 操作系统轮流让各个任务交替执行，由于CPU的执行速度实在是太快了， 我们感觉就像所有任务都在同时执行一样

- 多进程中， 每个进程中所有数据（包括全局变量） 都各有拥有一份， 互不影响

多进程

- 模拟多任务处理：一边唱歌一边跳舞

```
from time import sleep
def sing():
    for i in range(3):
        print("正在唱歌")
        dance()
        sleep(1)

def dance():
    print("正在跳舞")
if __name__ == '__main__':
    sing()
```

多进程

- 程序开始运行时，首先会创建一个主进程
- 在主进程（父进程）下，我们可以创建新的进程（子进程），子进程依赖于主进程，如果主进程结束，程序会退出
- Python提供了非常好用的多进程包multiprocessing，借助这个包，可以轻松完成从单进程到并发执行的转换

多进程

- multiprocessing模块提供了一个Process类来创建一个进程对象

```
from multiprocessing import Process
def run(name):
    print("子进程运行中，name = %s"%(name))
if __name__ == "__main__":
    print("父进程启动")
    p = Process(target=run, args=('test',))
    #target表示调用对象，args表示调用对象的位置参数元组
    #（注意：元组中只有一个元素时结尾要加，）
    print("子进程将要执行")
    p.start()
    print(p.name)   #p.pid
    p.join()
    print("子进程结束")
```

多进程

if __name__ == “__main__”:说明

一个python的文件有两种使用的方法，第一是直接作为程序执行，第二是import到其他的python程序中被调用（模块重用）执行。

因此if __name__ == 'main': 的作用就是控制这两种情况执行代码的过程，__name__ 是内置变量，用于表示当前模块的名字

在if __name__ == 'main': 下的代码只有在文件作为程序直接执行才会被执行，而import到其他程序中是不会被执行的

在 Windows 上，子进程会自动 import 启动它的这个文件，而在 import 的时候是会执行这些语句的。
如果不加if __name__ == "__main__":的话就会无限递归创建子进程

所以必须把创建子进程的部分用那个 if 判断保护起来

import 的时候 __name__ 不是 __main__ ，就不会递归运行了

多进程

- Process(target , name , args)
- 参数介绍
  - target表示调用对象，即子进程要执行的任务
  - args表示调用对象的位置参数元组，args=(1,)
  - name为子进程的名称

多进程

- Process类常用方法：
  - p.start()：启动进程，并调用该子进程中的p.run()
  - p.run():进程启动时运行的方法，正是它去调用target指定的函数，我们自定义类的类中一定要实现该方法
  - p.terminate()（了解）强制终止进程p，不会进行任何清理操作
  - p.is_alive():如果p仍然运行，返回True.用来判断进程是否还在运行
  - p.join([timeout]):主进程等待p终止，timeout是可选的超时时间

多进程

- Process类常用属性：
  name： 当前进程实例别名， 默认为Process-N， N为从1开始递增的整数；
  pid： 当前进程实例的PID值

多进程

全局变量在多个进程中不共享：进程之间的数据是独立的，默认情况下互不影响

```
from multiprocessing import Process
num = 1
def run1():
    global num
    num += 5
    print("子进程1运行中，num = %d"%(num))
def run2():
    global num
    num += 10
    print("子进程2运行中，num = %d"%(num))
```

```
if __name__ == "__main__":
    print("父进程启动")
    p1 = Process(target=run1)
    p2 = Process(target=run2)
    print("子进程将要执行")
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    print("子进程结束")
```

多进程

- 创建新的进程还能够使用类的方式， 可以自定义一个类， 继承Process类， 每次实例化这个类的时候， 就等同于实例化一个进程对象

```
import multiprocessing
import time
class ClockProcess(multiprocessing.Process):
def run(self):
        n = 5
        while n > 0:
            print(n)
            time.sleep(1)
            n -= 1
if __name__ == '__main__':
    p = ClockProcess()
    p.start()
    p.join()
```

进程池

- 进程池：用来创建多个进程

- 当需要创建的子进程数量不多时， 可以直接利用multiprocessing中的Process动态生成多个进程， 但如果是上百甚至上千个目标， 手动的去创建进程的工作量巨大，此时就可以用到multiprocessing模块提供的Pool

- 初始化Pool时， 可以指定一个最大进程数， 当有新的请求提交到Pool中时，如果池还没有满， 那么就会创建一个新的进程用来执行该请求； 但如果池中的进程数已经达到指定的最大值， 那么该请求就会等待， 直到池中有进程结束， 才会创建新的进程来执行

进程池

```
from multiprocessing import Pool
import random,time

def work(num):
    print(random.random()*num)
    time.sleep(3)
if __name__ == "__main__":
    po = Pool(3)        #定义一个进程池，最大进程数为3，默认大小为CPU核数
    for i in range(10):
        po.apply_async(work,(i,))      #apply_async选择要调用的目标，每次循环会用空出来的子进程去调用目标
    po.close()       #进程池关闭之后不再接收新的请求
    po.join()        #等待po中所有子进程结束，必须放在close后面
```

在多进程中：主进程一般用来等待，真正的任务都在子进程中执行

进程池

- multiprocessing.Pool常用函数解析：
  - apply_async(func[, args[, kwds]])： 使用非阻塞方式调用func（并行执行， 堵塞方式必须等待上一个进程退出才能执行下一个进程） ， args为传递给func的参数列表， kwds为传递给func的关键字参数列表；
  - apply(func[, args[, kwds]])（了解即可） 使用阻塞方式调用func
  - close()： 关闭Pool， 使其不再接受新的任务；
  - join()： 主进程阻塞， 等待子进程的退出， 必须在close或terminate之后使用；

进程间通信-Queue

- 多进程之间，默认是不共享数据的
- 通过Queue（队列Q）可以实现进程间的数据传递
- Q本身是一个消息队列
- 如何添加消息（入队操作）：

```
from multiprocessing import Queue
q = Queue(3)      #初始化一个Queue对象，最多可接受3条消息
q.put(“消息1”)     #添加的消息数据类型不限
q.put("消息2")
q.put("消息3")
print(q.full())
```

进程间通信-Queue

- 可以使用multiprocessing模块的Queue实现多进程之间的数据传递
- 初始化Queue()对象时（例如： q=Queue()） ， 若括号中没有指定最大可接收的消息数量， 或数量为负值， 那么就代表可接受的消息数量没有上限
- Queue.qsize()： 返回当前队列包含的消息数量
- Queue.empty()： 如果队列为空， 返回True， 反之False
- Queue.full()： 如果队列满了， 返回True,反之False
- Queue.get([block[, timeout]])： 获取队列中的一条消息， 然后将其从列队中移除， block默认值为True
  - 如果block使用默认值， 且没有设置timeout（单位秒） ， 消息列队如果为空， 此时程序将被阻塞（停在读取状态） ， 直到从消息列队读到消息为止，如果设置了timeout， 则会等待timeout秒， 若还没读取到任何消息， 则抛出"Queue.Empty"异常
  - 如果block值为False， 消息列队如果为空， 则会立刻抛出“Queue.Empty”异常

进程间通信-Queue

- Queue.get_nowait()： 相当Queue.get(False)
- Queue.put(item,[block[, timeout]])： 将item消息写入队列， block默认值为True
  - 如果block使用默认值， 且没有设置timeout（单位秒） ， 消息列队如果已经没有空间可写入， 此时程序将被阻塞（停在写入状态） ， 直到从消息列队腾出空间为止， 如果设置了True和timeout， 则会等待timeout秒， 若还没空间， 则抛出"Queue.Full"异常
  - 如果block值为False， 消息列队如果没有空间可写入， 则会立刻抛出"Queue.Full"异常
- Queue.put_nowait(item)： 相当Queue.put(item, False)；

进程间通信-Queue

```
from multiprocessing import Queue, Process
import time

def write(q):
    for value in ["a","b","c"]:
        print("开始写入：",value)
        q.put(value)
        time.sleep(1)

def read(q):
    while True:
        if not q.empty():
            print("读取到的是",q.get())
            time.sleep(1)
        else:
            break
```

```
if __name__ == "__main__":
    q = Queue()
    pw = Process(target=write, args=(q,))
    pr = Process(target=read, args=(q,))
    pw.start()
    pw.join()#等待接收完毕
    pr.start()
    pr.join()
    print("接收完毕！")
```

问题：
如果有两个接收方怎么办？（多任务之间配合）

进程间通信-Queue

- 进程池创建的进程之间通信：如果要使用Pool创建进程， 就需要使用multiprocessing.Manager()中的Queue()
  而不是multiprocessing.Queue()

- 否则会得到一条如下的错误信息：
  RuntimeError: Queue objects should only be shared between processes through inheritance.

进程间通信-Queue

```
from multiprocessing import Manager,Pool
import time

def writer(q):
    for i in "welcome":
        print("开始写入",i)
        q.put(i)

def reader(q):
    time.sleep(3)
    for i in range(q.qsize()):
        print("得到消息",q.get())
```

```
if __name__ == "__main__":
    print("主进程启动")
    q = Manager().Queue()
    po = Pool()
    po.apply_async(writer,(q,))
    po.apply_async(reader,(q,))
    po.close()
    po.join()
```

多线程

PART THREE

多线程

- **线程:实现多任务的另一种方式**
- **一个进程中，也经常需要同时做多件事，就需要同时运行多个‘子任务’，这些子任务，就是线程**
- 线程又被称为**轻量级进程**(lightweight process)，是更小的执行单元

  - 一个进程可拥有多个并行的(concurrent)线程，当中每一个线程，共享当前进程的资源
  - 一个进程中的线程共享相同的内存单元/内存地址空间→可以访问相同的变量和对象，而且它们从同一堆中分配对象→通信、数据交换、同步操作
  - 由于线程间的通信是在同一地址空间上进行的，所以不需要额外的通信机制，这就使得通信更简便而且信息传递的速度也更快

线程和进程的区别

- 进程是系统进行资源分配和调度的一个独立单位
- 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
- 一个程序至少有一个进程,一个进程至少有一个线程
- 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，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
- 线程的划分尺度小于进程(资源比进程少)， 使得多线程程序的并发性高
- 线程不能够独立执行， 必须依存在进程中
- 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

线程和进程的区别

区别	进程	线程
根本区别	作为资源分配的单位	调度和执行的单位
开　　销	每个进程都有独立的代码和数据空间(进程上下文)，进程间的切换会有较大的开销。	线程可以看成时轻量级的进程，同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器(PC)，线程切换的开销小。
所处环境	在操作系统中能同时运行多个任务(程序)	在同一应用程序中有多个顺序流同时执行
分配内存	系统在运行的时候会为每个进程分配不同的内存区域	除了CPU之外，不会为线程分配内存（线程所使用的资源是它所属的进程的资源），线程组只能共享资源
包含关系	没有线程的进程是可以被看作单线程的，如果一个进程内拥有多个线程，则执行过程不是一条线的，而是多条线（线程）共同完成的。	线程是进程的一部分，所以线程有的时候被称为是轻权进程或者轻量级进程。

一般来讲：我们把进程用来分配资源，线程用来具体执行（CPU调度）

多线程

- python的thread模块是比较底层的模块，在各个操作系统中表现形式不同（低级模块）
- python的threading模块是对thread做了一些包装的， 可以更加方便的被使用（高级模块）
- thread 有一些缺点，在threading 得到了弥补，所以我们直接学习threading

```
import threading
if __name__ == "__main__":
    #任何进程默认会启动一个线程，这个线程称为主线程，主线程可以启动新的子线程
    #current_thread():返回当前线程的实例
    #.name： 当前线程的名称
    print('主线程%s启动' %(threading.current_thread().name))
```

多线程

```
import threading,time
def saySorry():
    print("子线程%s启动" %(threading.current_thread().name))
    time.sleep(1)
    print("亲爱的，我错了，我能吃饭了吗？")

if __name__ == "__main__":
    print('主线程%s启动' %(threading.current_thread().name))
    for i in range(5):
        t = threading.Thread(target=saySorry))#Thread()：指定线程要执行的代码
        t.start()
```

查看当前线程数量

```
import threading
import time

def sing():
    for i in range(3):
        print("正在唱歌...%d" %i)
        time.sleep(1)
def dance():
    for i in range(2):
        print("正在跳舞...%d" %i)
        time.sleep(1)
```

```
if __name__ == "__main__":
    print("开始：%s" %time.time())
    t1 = threading.Thread(target=sing)
    t2 = threading.Thread(target=dance)
    t1.start()
    t2.start()

    while True:
        length = len(threading.enumerate())
        #threading.enumerate():返回当前运行中的Thread对象列表
        print("当前线程数为：%d" %length)
        if length<=1:
            break
        time.sleep(1)
```

多线程

- 创建线程的两种方式：
  - 第一：通过 threading.Thread 直接在线程中运行函数；
  - 第二：通过继承 threading.Thread 类来创建线程
    - 这种方法只需要重载 threading.Thread 类的 run 方法，然后调用 start()开启线程就可以了

```
import threading
class MyThread(threading.Thread):
    def run(self):
        for i in range(5):
            print(i)
if __name__ == "__main__":
    t1 = MyThread()
    t2 = MyThread()
    t1.start()
    t2.start()
```

多线程

```
import threading
import time
class MyThread(threading.Thread):
    def run(self):
        for i in range(3):
            time.sleep(1)
            msq = "I`m" + self.name + "@" + str(i)
            #name属性中保存了当前线程的名字
            print(msq)
if __name__ == "__main__":
    t = MyThread()
    t.start()
```

线程的五种状态

- 1、新状态：线程对象已经创建，还没有在其上调用start()方法。
- 2、可运行状态：当线程有资格运行，但调度程序还没有把它选定为运行线程时线程所处的状态。当start()方法调用时，线程首先进入可运行状态。在线程运行之后或者从阻塞、等待或睡眠状态回来后，也返回到可运行状态。
- 3、运行状态：线程调度程序从可运行池中选择一个线程作为当前线程时线程所处的状态。这也是线程进入运行状态的唯一一种方式。
- 4、等待/阻塞/睡眠状态：这是线程有资格运行时它所处的状态。实际上这个三状态组合为一种，其共同点是：线程仍旧是活的（可运行的），但是当前没有条件运行。但是如果某件事件出现，他可能返回到可运行状态。
- 5、死亡态：当线程的run()方法完成时就认为它死去。这个线程对象也许是活的，但是，它已经不是一个单独执行的线程。线程一旦死亡，就不能复生。如果在一个死去的线程上调用start()方法，会抛出异常。
-

线程共享全局变量

- 在一个进程内的所有线程共享全局变量， 多线程之间的数据共享（这点要比多进程要好）
- 缺点就是， 可能造成多个线程同时修改一个变量（即线程非安全），可能造成混乱

线程共享全局变量

```
import threading
import time

num = 100
def work1():
    global num
    for i in range(3):
        num += 1
    print("---in work1,num is %d" %num)
def work2():
    global num
    print("---in work2,num is %d" %num)
print("---线程创建之前 num is %d"
%num)
t1 = threading.Thread(target=work1)
t1.start()
time.sleep(1)
#延时一会保证线程1中的任务做完
t2 = threading.Thread(target=work2)
t2.start()
```

线程共享全局变量

```
import threading,time
def work1(nums):
    nums.append(44)
    print('-----in work1-----' ,nums)
def work2(nums):
    time.sleep(1)
    #延时一会保证另一线程执行
    print('-----in work2-----', nums)

g_nums = [11,22,33]
t1 = threading.Thread(target=work1,args=(g_nums,))
t1.start()
t2 = threading.Thread(target=work2,args=(g_nums,))
t2.start()
```

执行1000000次的bug

```
import threading
num = 0
def test1():
    global num
    for i in range(100):#一百万错误
                num += 1
def test2():
    global num
    for i in range(100):#一百万错误
                num += 1
p1 = threading.Thread(target=test1)
p1.start()
p2 = threading.Thread(target=test2)
p2.start()
print("---num = %d---" %num)
```

线程同步

PART FOUR

线程同步

- 当多个线程几乎同时修改某一个共享数据的时候， 需要进行同步控制
- 线程同步能够保证多个线程安全访问竞争资源， 最简单的同步机制是引入互斥锁
- 互斥锁保证了每次只有一个线程进行写入操作，从而保证了多线程情况下数据的正确性（原子性）
- 互斥锁为资源引入一个状态：锁定/非锁定。某个线程要更改共享数据时，先将其锁定，此时资源的状态为“锁定”，其他线程不能更改；直到该线程释放资源，将资源的状态变成“非锁定”，其他的线程才能再次锁定该资源。互斥锁保证了每次只有一个线程进行写入操作，从而保证了多线程情况下数据的正确性。

- threading模块中定义了Lock类， 可以方便的处理锁定

线程同步

- 创建锁
  mutex = threading.Lock()
- 锁定
  mutex.acquire()

- 释放
  mutex.release() #解锁

线程同步-互斥锁

```
import threading
num = 0
def test1():
    global num
    if mutex.acquire():
        for i in range(1000):
            num += 1
    mutex.release()
def test2():
    global num
    if mutex.acquire():
        for i in range(1000):
            num += 1
    mutex.release()
```

```
mutex = threading.Lock()
p1 = threading.Thread(target=test1)
p1.start()
p2 = threading.Thread(target=test2)
p2.start()
print(num)
```

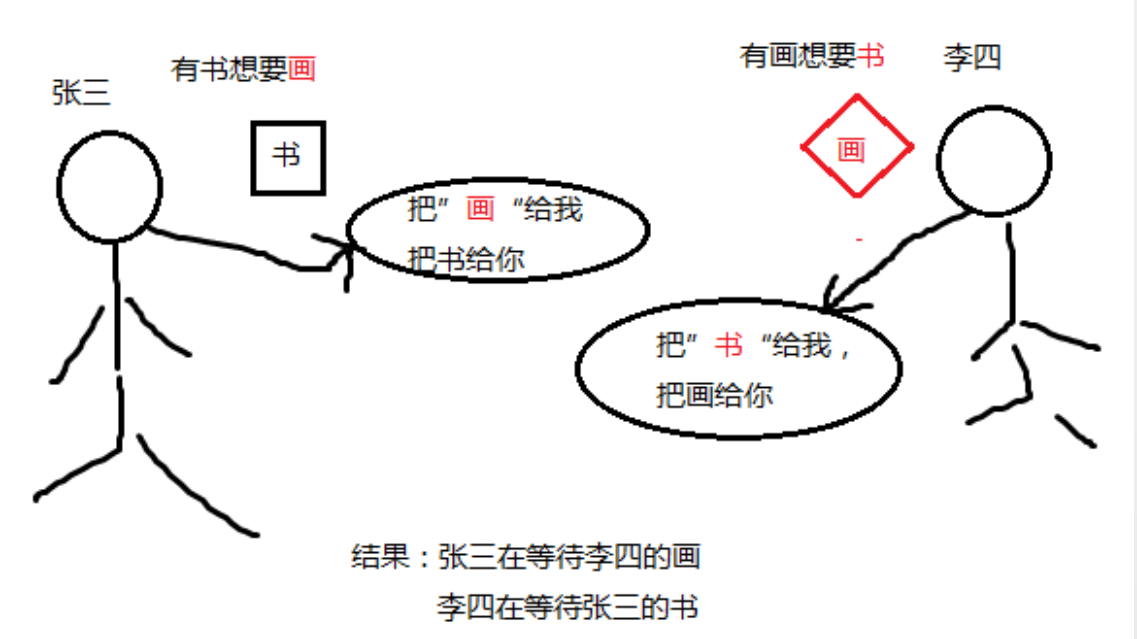
当一个线程调用Lock对象的acquire()方法获得锁时，这把锁就进入“locked”状态。因为每次只有一个线程可以获得锁，所以如果此时另一个线程2试图获得这个锁，该线程2就会变为同步阻塞状态

直到拥有锁的线程1调用锁的release()方法释放锁之后，该锁进入“unlocked”状态。线程调度程序继续从处于同步阻塞状态的线程中选择一个来获得锁，并使得该线程进入运行（running）状态

一个线程有锁时
别的线程只能在
外面等着

线程同步（死锁）

- 死锁（错误情况，理解即可）
  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



线程同步（死锁）

```
import threading
import time
class MyThread1(threading.Thread):
    def run(self):
        if mutexA.acquire():
            print(self.name + '---do1---up---')
            time.sleep(1)
            if mutexB.acquire():
                print(self.name + '---do1---down---')
                mutexB.release()
        mutexA.release()
```

在多线程程序中，死锁问题很大一部分是由于线程同时获取多个锁造成的。如一个线程获取了第一个锁，然后在获取第二个锁的时候发生阻塞，那么这个线程就可能阻塞其他线程的执行，从而导致整个程序假死（两个人每人一根筷子）

```
class MyThread2(threading.Thread):
    def run(self):
            time.sleep(1)
            if mutexB.acquire():
                    print(self.name + '---do2---up---')
                    if mutexA.acquire():
                            print(self.name + '---do2---d
                            mutexA.release()
            mutexB.release()

if __name__ == '__main__':
    mutexA = threading.Lock()
    mutexB = threading.Lock()
    t1 = MyThread1()
    t2 = MyThread2()
    t1.start()
    t2.start()
```

信号量

- 信号量semaphore： 用于控制一个时间点内线程进入数量的锁，信号量是用来控制线程并发数的

- 使用场景举例：在读写文件的时候，一般只有一个线程在写，而读可以有多个线程同时进行，如果需要限制同时读文件的线程个数，这时候就可以用到信号量了（如果用互斥锁，就是限制同一时刻只能有一个线程读取文件）

信号量

```
import time
import threading

def foo():
    time.sleep(2)
    print("ok",time.ctime())

for i in range(100):
    t1=threading.Thread(target=foo)
    t1.start()  #此时无法控制同时进入的线程数
```

信号量

```
import time
import threading
s1=threading.Semaphore(5)        (赛么佛)
def foo():
    s1.acquire()
    time.sleep(2)
    print("ok")
    s1.release()
for i in range(20):
    t1=threading.Thread(target=foo,args=())
    t1.start()    #此时可以控制同时进入的线程数
```

GIL全局解释器锁

- Cpython独有的锁，牺牲效率保证数据安全
  - GIL锁是一把双刃剑，它带来优势的同时也带来一些问题
  - 首先：执行Python文件是什么过程？谁把进程起来的？
    - 操作系统将你的应用程序从硬盘加载到内存。运行python文件，在内存中开辟一个进程空间，将你的Python解释器以及py文件加载进去，解释器运行py文件
    - Python解释器分为两部分，先将你的代码通过编译器编译成C的字节码，然后你的虚拟机拿到你的C的字节码，输出机器码，再配合操作系统把你的这个机器码扔给cpu去执行
    - 你的py文件中有一个主线程，主线程做的就是这个过程。如果开多线程，每个线程都要进行这个过程

GIL全局解释器锁

- 理想的情况：
  - 三个线程，得到三个机器码，然后交由CPU，三个线程同时扔给三个CPU，然后同时进行，最大限度的提高效率，但是CPython多线程应用不了多核
- CPython到底干了一件什么事情导致用不了多核?
  - Cpython在所有线程进入解释器之前加了一个全局解释器锁（GIL锁）。这个锁是互斥锁，是加在解释器上的，导致同一时间只有一个线程在执行所以你用不了多核

- 为什么这么干?
  - 之前写python的人只有一个cpu。。。
  - 所以加了一个锁，保证了数据的安全，而且在写python解释器时，更加好写了
- 为什么不取消这个锁?
  - 解释器内部的管理全部是针对单线程写的

GIL全局解释器锁

- 能不能不用Cpython?
  - 官方推荐Cpython，处理速度快，相对其他解释器较完善。其他解释器比如pypy规则和漏洞很多
- 那我该怎么办?
  - 虽然多线程无法应用多核，但是多进程可以应用多核（开销大）
- 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，为什么我还要学互斥锁?
  - 首先我们需要达成共识：锁的目的是为了保护共享的数据，同一时间只能有一个线程来修改共享的数据
  - 然后，我们可以得出结论：保护不同的数据就应该加不同的锁。
  - 所以，GIL 与Lock是两把锁，保护的数据不一样，前者是解释器级别的（当然保护的就是解释器级别的数据，比如垃圾回收的数据），后者是保护用户自己开发的应用程序的数据，很明显GIL不负责这件事，只能用户自定义加锁处理

同步和异步

- 同步调用：确定调用的顺序
  - 提交一个任务,自任务开始运行直到此任务结束,我再提交下一个任务
  - 按顺序购买四大名著
- 异步调用：不确定顺序
  - 一次提交多个任务,然后我就直接执行下一行代码
  - 你喊你朋友吃饭， 你朋友说知道了， 待会忙完去找你， 你就去做别的了

- 给三个老师发布任务:
  - 同步: 先告知第一个老师完成写书的任务,我原地等待,等他两个月之后完成了,我再发布下一个任务……
  - 异步:直接将三个任务告知三个老师,我就忙我的我,直到三个老师完成之后,告知我

同步和异步

- 同步意味着顺序、统一的时间轴
  - 场景1：是指完成事务的逻辑，先执行第一个事务，如果阻塞了，会一直等待，直到这个事务完成，再执行第二个事务，协同步调，按预定的先后次序进行运行
  - 场景2：一个任务的完成需要依赖另外一个任务时，只有等待被依赖的任务完成后，依赖的任务才能算完成，这是一种可靠的任务序列

线程同步-多个线程有序执行

```
import threading,time
class Task1(threading.Thread):
    def run(self):
        while True:
            if lock1.acquire():
                print('-----Task1-----')
                time.sleep(1)
                lock2.release()
class Task2(threading.Thread):
    def run(self):
        while True:
            if lock2.acquire():
                print('-----Task2-----')
                time.sleep(1)
                lock3.release()
class Task3(threading.Thread):
    def run(self):
        while True:
            if lock3.acquire():
                print('-----Task3-----')
                time.sleep(1)
                lock1.release()
lock1 = threading.Lock()
#创建另外一把锁，并且锁上
lock2 = threading.Lock()
lock2.acquire()
#再创建一把锁并锁上
lock3 = threading.Lock()
lock3.acquire()
t1 = Task1()
t2 = Task2()
t3 = Task3()
t1.start()    t2.start()      t3.start()
```

线程同步-消息队列

- Python的Queue模块：实现了3种类型的队列来实现线程同步，包括：
  - FIFO（先入先出) 队列 Queue，
  - LIFO（后入先出） 栈 LifoQueue，
  - 优先级队列 PriorityQueue
  - 区别在于队列中条目检索的顺序不同
  - 在FIFO队列中，按照先进先出的顺序检索条目
  - 在LIFO队列中，最后添加的条目最先检索到（操作类似一个栈）
  - 在优先级队列中，条目被保存为有序的（使用heapq模块）并且最小值的条目被最先检索
- 这些队列都实现了锁原语（可以理解为原子操作， 即要么不做， 要么就做完），能够在多线程中直接使用

- 现阶段只要求掌握其中一种，FIFO队列

生产者消费者模式

- 生产者消费者模式
  - 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（做包子，吃包子）
    - 经常会出现生产数据的速度大于消费数据的速度，或者生产速度跟不上消费速度
  - 生产者消费者模式是通过一个容器（缓冲区）来解决生产者和消费者的强耦合问题
    - 例如两个线程共同操作一个列表，一个放数据，一个取数据
  - 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

线程同步--消息队列

- **class queue.Queue(maxsize=0)**
  - FIFO队列的构造器。maxsize为一个整数,表示队列的最大条目数，可用来限制内存的使用。
  - 一旦队列满，插入将被阻塞直到队列中存在空闲空间。如果maxsize小于等于0，队列大小为无限。maxsize默认为0

线程同步--消息队列

```
import threading
import time
from queue import Queue
class Pro(threading.Thread):
    def run(self):
        global queue
        count = 0
        while True:
            if queue.qsize()<1000:
                for i in range(100):
                    count = count + 1
                    msg = '生成产品' + str(count)
                    queue.put(msg)#队列中添加新产品
                    print(msg)
            time.sleep(1)
```

```
class Con(threading.Thread):
    def run(self):
        global queue
        while True:
            if queue.qsize() > 100:
                for i in range(3):
                    msg = self.name + '消费了' + queue.get()
                    print(msg)
            time.sleep(1)
if __name__ == "__main__":
    queue = Queue()
    #创建一个队列。线程中能用，进程中不能使用
    for i in range(500):#创建500个产品放到队列里
        queue.put( '初始产品'  + str(i))#字符串放进队列
    for i in range(2):#创建了两个线程
        p = Pro()
        p.start()
    for i in range(5):#5个线程
        c = Con()
        c.start()
```

同步和异步

- 异步则意味着乱序、效率优先的时间轴
  - 处理调用这个事务的之后，不会等待这个事务的处理结果，直接处理第二个事务去了，通过状态、回调来通知调用者处理结果
  - 对于I/O相关的程序来说，异步编程可以大幅度的提高系统的吞吐量，因为在某个I/O操作的读写过程中，系统可以先去处理其它的操作（通常是其它的I/O操作）
  - 不确定执行顺序

异步1，无需等待线程执行

```
import threading, time
def thead(num):
    print("线程%s开始执行" % num)
    time.sleep(3)
    print("线程%s执行完毕" % num)


print("主方法开始执行")
poll = []
for i in range(1, 3):
    thead_one = threading.Thread(target=thead, args=(i,))
    poll.append(thead_one)
for n in poll:
    n.start()  # 准备就绪,等待cpu执行

print("主方法执行完毕")
```

异步2，通过循环控制

```
import threading,time
num = 0
def test1():
    global num
    while True:
        if mutex.acquire(False):
            for i in range(1000000):
                num += 1
            print("1", num)
            mutex.release()
            break
        else:
            print("该干嘛干嘛")
def test2():
    global num
    while True:
        if mutex.acquire(False):
            for i in range(1000000):
                num += 1
            print("2", num)
            mutex.release()
            break
        else:
            print("该干嘛干嘛")
mutex = threading.Lock()
p1 = threading.Thread(target=test1)
p2 = threading.Thread(target=test2)
p1.start()
p2.start()
```

异步3，通过回调机制

- ```
  from multiprocessing import Pool
  import random
  import time
  def download(f):
      for i in range(1,4):
          print(f"{f}下载文件{i}")
          time.sleep(random.randint(1,3))
      return "下载完成"

  def alterUser(msg):
      print(msg)
  if __name__ == "__main__":
      p = Pool(3)
      # 当func执行完毕后，return的东西会给到回调函数callback
      p.apply_async(func= download,args=("线程1",),callback = alterUser)
      p.apply_async(func=download, args=("线程2",),callback =alterUser)
      p.apply_async(func=download, args=("线程3",),callback =alterUser)
      p.close()
      p.join()
  ```
  ```

协程

- 比线程更小的执行单元（微线程）
- 一个线程作为一个容器里面可以放置多个协程

- 只切换函数调用即可完成多线程，可以减少CPU的切换
- 协程自己主动让出CPU

协程

- 安装模块： pip3 install greenlet

```
from greenlet import greenlet
import time
def t1():
    while True:
        print("........A.........")
        gr2.switch()
        time.sleep(1)
def t2():
    while True:
        print("........b.........")
        gr1.switch()#调到上次执行的地方继续执行
        time.sleep(1)
gr1 = greenlet(t1)#创建一个greenlet对象
gr2 = greenlet(t2)
gr1.switch()#此时会执行1函数
```

协程

- python还有一个比greenlet更强大的并且能够自动切换任务的模块 gevent
- 原理是当一个greenlet遇到IO(指的是input output 输入输出)操作时， 比如访问网络， 就自动切换到其他的greenlet， 等到IO操作完成， 再在适当的时候切换回来继续执行

- 进程线程的任务切换是由操作系统自行切换的，你自己不能控制

- 协程可以通过自己的程序（代码）来进行切换，自己能够控制

- gevent只有遇到模块能够识别的IO操作的时候，程序才会进行任务切换，实现并发效果，如果所有程序都没有IO操作，那么就基本属于串行执行了

协程

```
import gevent
def A():
    while True:
        print(".....A.....")
        gevent.sleep(1)#用来模拟一个耗时操作
        #gevent中：当一个协程遇到耗时操作会自动交出控制权给其他协程
def B():
    while True:
        print(".....B.....")
        gevent.sleep(1)#每当遇到耗时操作，会自用转到其他协程
g1 = gevent.spawn(A) # 创建一个gevent对象（创建了一个协程），此时就已经开始执行A
g2 = gevent.spawn(B)
g1.join()  #等待协程执行结束
g2.join()  #会等待协程运行结束后再退出
```