什么是进程、线程

　　进程：一个整体的形式暴露给操作系统管理，里面包含对各种资源的调用，内存的管理，网络接口的调用等，对各种资源管理的基本单位。

　　线程：操作系统最小的调度单位， 是一串指令的集合，进程中的一个执行单元。

　　一个进程至少有一个线程。

　　全局解释器锁（GIL）：python虚拟机的访问由全局解释器锁控制，这个锁能保证同一时刻只有一个线程运行。

进程与线程的区别

　　◐ 线程之间共享内存空间，而进程的内存是独立，即使是父子进程

　　◐ 同一个进程的线程之间可以直接交流，两个进程想通信，必须通过一个中间代理来实现

　　◐ 创建新线程很简单，但是创建一个新进程需要克隆一次父进程

　　◐ 一个线程可以控制和操作同一个进程内的其他线程，而进程只能操作子进程

多线程环境中，python虚拟机的执行方式

一、线程

Part1 简单的线程

1 import threading

2 import time

3

4 def test(n):

5 print("thread",n)

6 time.sleep(3)

7

8 t1 = threading.Thread(target=test,args=("tt1",)) # 启动一个线程t1，执行test函数，参数为字符串tt1

9 t2 = threading.Thread(target=test,args=("tt2",))

10 t1.start() # 使用两个线程执行这个函数，cpu在t1执行完print后，遇到sleep，就会切换到t2执行print

11 t2.start()

12

13 # test("tt1") # 而直接调用两个函数执行，cpu会先执行完第一个，再执行下一个，这样比多线程多了个执行sleep的时间

14 # test("tt2")

Part2 用类的形式启动线程

1 # 用类的形式启动线程

2 import threading

3

4 class MyThread(threading.Thread):

5 def \_\_init\_\_(self,n):

6 super(MyThread,self).\_\_init\_\_()

7 self.n = n

8

9 def run(self): # 这里必须是run函数，不能取其他名，里面写死了会调用run函数

10 print("run thread",self.n)

11

12 t1 = MyThread("tt1")

13 t2 = MyThread("tt2")

14 t1.start()

15 t2.start()

Part3 主线程等待子线程执行完成

1 import threading

2 import time

3

4 def run(n):

5 print("th:",n,threading.current\_thread()) # 可以查看当前进程是为主线程还是子线程

6 time.sleep(2)

7

8 start\_time = time.time()

9 t\_list = [] # 定义一个空列表，用来存启动的子线程

10 for i in range(50):

11 t = threading.Thread(target=run,args=("t-%d" %i,))

12 t.start()

13 t\_list.append(t)

14 # print("cost time:",time.time()-start\_time)

15 # 按目前所知可得，这里打印的时间是整个程序执行一共花的时间。但是最终执行下来时间只有0.02左右，里面怎么没有包含函数sleep的时间呢

16 # 一个进程至少有一个线程，从执行这个程序开始，就启动了一个主线程，而主线程中启动了50个子线程，而子线程启动后和主线程独立没有影响（主子并行）

17 # 其中的print语句也是主线程部分，sleep语句是子线程部分，所以打印的时间是主线程启动了50个子线程的时间，主线程并不会等待子线程执行完了再执行后面的程序

18 print(threading.active\_count()) # 输出51，表示当前存活的线程，包含主线程

19 for t in t\_list:

20 t.join()

21

22 print("cost time:",time.time()-start\_time,threading.current\_thread())

23 # 这里就使用join来等待子线程的完成，其中等待是指主线程等待子线程执行完后再继续执行，默认程序最后都有一个join

24 # 不能直接在启动线程的循环里写join，那样会变成串行，因为每启动一个线程，都要等待执行完成后才启动下一个线程

25 # 这里直接循环每个已经启动了的线程，主线程会等所有的子线程执行完后再执行print时间

Part4 守护线程

1 # 守护线程，主线程执行完了，不管守护线程有没有执行完都退出

2 import threading

3 import time

4

5 def run(n):

6 print("th:",n)

7 time.sleep(2)

8

9 start\_time = time.time()

10 for i in range(50):

11 t = threading.Thread(target=run,args=("t%d" %i,))

12 t.setDaemon(True) # 把当前线程设置为守护线程，必须在start之前设置

13 t.start()

14 print("cost time:",time.time()-start\_time)

15 # 主线程不是守护线程，程序会等主线程执行完之后，不会等待守护线程，也就是子线程，就直接程序退出了

Part5 使用全局解释器锁

1 # 设置全局解释器锁

2 import threading

3 import time

4

5 def run(n):

6 lock.acquire() # 设置锁

7 print("th:",n)

8 global num

9 num += 1

10 lock.release() # 释放锁

11 time.sleep(2)

12

13 lock = threading.Lock() # 生成一个锁

14 num = 0

15 start\_time = time.time()

16 for i in range(50):

17 t = threading.Thread(target=run,args=("t%d" %i,))

18 t.start()

19 print(num)

20 print("cost time:",time.time()-start\_time)

例2：

　　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，每个线程可以访问同一份数据，所以当多个线程同时要修改同一份数据时，就会出现错误

　　例如：

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

import threading

import time

num = 100 #设置一个共享变量

def show():

global num #在函数内操作函数外变量，需设置为全局变量

time.sleep(1)

num -= 1

list=[]

for i in range(100):

t = threading.Thread(target=show)

t.start()

list.append(t)

for t in list:

t.join()

print(num)

　　上面的例子在正常执行完成后的num的结果应该是0，但实际上每次的执行结果都不太一样，因为当多个线程同时要修改同一份数据时，就会出现一些错误（只有在python2.x运行才会出现错误，python3.x中不会），所以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，可以加上线程锁(全局解释锁)来确保每次修改数据时只有一个线程在操作。

　　加锁代码

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

import threading

import time

num = 100 #设置一个共享变量

lock=threading.Lock() #生成全局锁

def show():

global num #在函数内操作函数外变量，需设置为全局变量

time.sleep(1)

lock.acquire() #修改前加锁

num -= 1

lock.release() #修改后解锁

list=[]

for i in range(100):

t = threading.Thread(target=show)

t.start()

list.append(t)

for t in list:

t.join()

print(num)

Part6 使用递归锁

　　就是在一个大锁中再包含子锁

import threading

#递归锁

def run1():

lock.acquire() #小锁

global num

num +=1

lock.release()

return num

def run2():

lock.acquire() #小锁

global num2

num2+=1

lock.release()

return num2

def run3():

lock.acquire() #大锁

res = run1()

res2 = run2()

lock.release()

print(res,res2)

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

num,num2 = 0,0

lock = threading.RLock() #生成Rlock

for i in range(10):

t = threading.Thread(target=run3)

t.start()

while threading.active\_count() != 1:#如果不等于1，说明子线程还没执行完毕

pass #打印进程数

else:

print(num,num2)

Part7 信号量

同时允许一定数量的线程更改数据

1 # 信号量 一般用于连接池，并发数

2 import threading, time

3

4 def run(n):

5 semaphore.acquire()

6 time.sleep(1)

7 print("run the thread: %s\n" % n)

8 semaphore.release()

9

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

11 semaphore = threading.BoundedSemaphore(5) # 最多允许5个线程同时运行，并非5个执行完了再执行下5个，是保持在5个

12 for i in range(22):

13 t = threading.Thread(target=run, args=(i,))

14 t.start()

15 while threading.active\_count() != 1: #如果不等于1，说明子线程还没执行完毕

16 pass # print threading.active\_count()

17 else:

18 print('----all threads done---')

Part8 队列queue

1.Queue模块常用来处理队列相关问题。

2.常用于生产消费者模型，主要功能是提供效率和程序解耦。

import queue

q = queue.Queue() # 实例化队列，数据先入先出

# q = queue.Queue(maxsize=3) 最多存放3个数据，put第四个时候就会卡住，等数据有被取走，就放进去

# q = queue.LifoQueue() 数据后入先出

# q = queue.PriorityQueue() 设置优先级

q.put("a") # 存入数据

q.put(123)

#q.put("a",block=False) # 放进数据超过指定最大数量就会报异常

#q.put("a",timeout=3) # q满了，等待3秒还是不能放进去的话就报错

#q.put((2,"p1")) 传入元组，第一个元素是优先级，从小到大取数据

#q.put((-1,"p1"))

#q.put((6,"p1"))

print(q.qsize()) # 返回队列里元素数量

print(q.get()) # 获取一个数据，如果队列里没有数据就会卡住

#q.get(timeout=3) # 有数据就立刻获取返回，如果没有数据就等待3秒，若依然没有数据就报异常

#q.get(block=False) # 如果队列里没有数据就会报异常，默认为True

#q.get\_nowait() # 如果队列里没有数据就会报异常

Part9 生产消费模型

1 import threading

2 import time

3 import queue

4

5 q = queue.Queue(maxsize=10)

6

7 def productData(name):

8 i = 1

9 while True:

10 time.sleep(0.4)

11 q.put("数据%s" %i)

12 print("[%s] 生产了 数据[%s]" %(name,i))

13 i += 1

14

15 def consumeData(name):

16 while True:

17 print("[%s] 消费了 [%s]" %(name,q.get()))

18 time.sleep(1)

19

20 p = threading.Thread(target=productData,args=("p1",))

21 c1 = threading.Thread(target=consumeData,args=("c1",))

22 c2 = threading.Thread(target=consumeData,args=("c2",))

23 p.start()

24 c1.start()

25 c2.start()

Part10 线程事件 Event

event

　　实现两个或多个线程间的交互，提供了三个方法 set、wait、clear，默认碰到event.wait 方法时就会阻塞。

　　event.set()，设定后遇到wait不阻塞

　　event.clear()，设定后遇到wait后阻塞

　　event.isSet()，判断有没有被设定

import threading

def start():

print("---start---1")

event.wait() #阻塞

print("---start---2")

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

event = threading.Event()

t = threading.Thread(target=start)

t.start()

result=input(">>:")

if result == "set":

event.set() #设定set，wait不阻塞