**python3 多线程**

**一 概念**

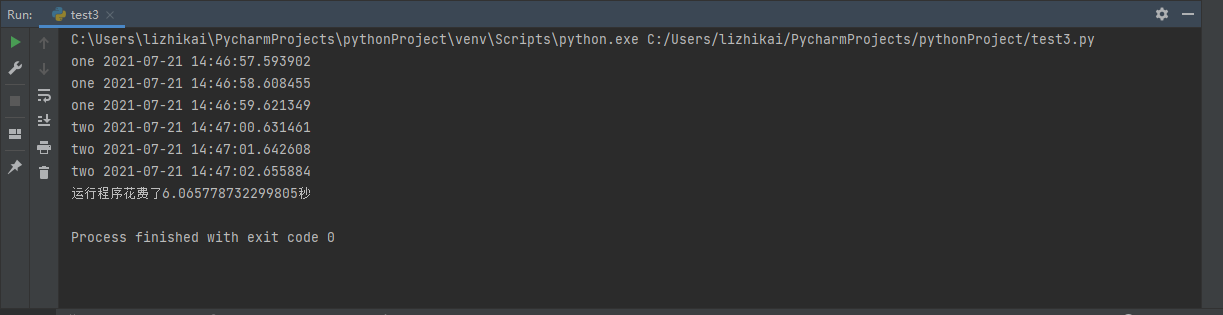
多线程相当于同时执行多个程序，大多数情况是出现在多个程序需要同时调用同一个资源。

**二 单线程**

代码示例

|  |
| --- |
| import \_thread  import time  from datetime import datetime  def Test(name):  for i in range(3):  print(name, datetime.now())  time.sleep(1)  def main():  Test("one")  Test("two")  if \_\_name\_\_ == '\_\_main\_\_':  start = time.time()  main()  end = time.time()  print("运行程序花费了%s秒" % (end - start)) |

运行结果



**三 多线程**

**1 模块**

import threading

**2 模块方法**

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列表。正在运行指线程启动后、结束前。

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

**3 创建方法**

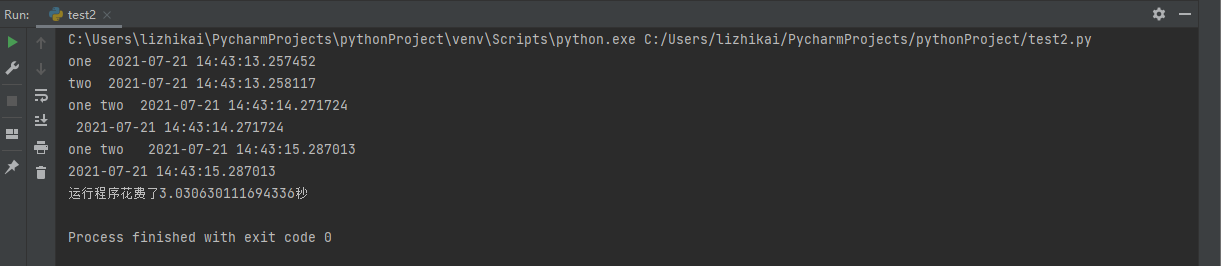
1 创建线程可以通过继承threading.Thread 类并重写start函数来实现

2 也可以直接通过创建函数并调用threading.Thread()函数来实现

**继承重写法**

|  |
| --- |
| 代码示例：  import threading  import time  from datetime import datetime  #继承threading.Thread类并重写run函数  class myThread(threading.Thread):  def \_\_init\_\_(self, name):  threading.Thread.\_\_init\_\_(self)  self.name = name  #run函数是线程执行的任务，因此需要根据自身需求来重写run函数  def run(self):  #需求：输出传入的name和当前时间  for i in range(3):  print(self.name, datetime.now())  time.sleep(1)  def main():  #实例化我们继承的myThread类，并传入参数，这里需要几个线程就需要实例化多少个类  thread1 = myThread("one ")  thread2 = myThread("two ")  # 启动线程任务，即启动myThread类中的run方法  thread1.start() # 启动线程，代用了该线程的 run() 方法  thread2.start()  thread1.join() # 必须等子线程运行完了父线程才可以运行  thread2.join()  if \_\_name\_\_ == '\_\_main\_\_':  start = time.time()  main()  end = time.time()  print("运行程序花费了%s秒" % (end - start)) |

运行结果：



1. 线程调用方法1精炼

a 创建继承类，根据需求重写run方法

class myThread(threading.Thread):

def run(self):

……

b 在主函数或者其他需要调用线程的函数中实例化继承类对象

thread1 = myThread("one ")

c 启动线程

thread1.start()

d 设置阻塞，让父线程在子线程完成之后再结束

thread1.join()

**创建函数调用法**

|  |
| --- |
| 代码示例：  import threading  import time  from datetime import datetime  def Test(name):  for i in range(3):  print(name, datetime.now())  time.sleep(1)  def main():  # 调用threading.Thread函数，target参数是要执行的函数，args是要传入的参数，为元组类型，有多少参数就往里写多少  t1 = threading.Thread(target=Test, args=("one",))  t2 = threading.Thread(target=Test, args=("two",))  t1.start() # 启动线程  t2.start()  t1.join() # 必须等子线程运行完了父进程才可以运行  t2.join()  if \_\_name\_\_ == '\_\_main\_\_':  start = time.time()  main()  end = time.time()  print("运行程序花费了%s秒" % (end - start)) |

1. 创建函数调用法精炼

a 根据自身需求创建线程任务函数供线程调用

def Test(name):

for i in range(3):

print(name, datetime.now())

time.sleep(1)

b 在调用线程的主函数或者其他函数中调用threading.thread方法创建线程

t1 = threading.Thread(target=Test, args=("one",))

t2 = threading.Thread(target=Test, args=("two",))

c 启动线程并阻塞父线程直至子线程完成后结束父线程

t1.start()

t1.join()

**4 线程同步**

当多个线程需要修改同一个资源（例如同一个变量的值），如果这些线程同时对该数据进行修改，那么会出现数据错乱的情况，因此需要考虑同步问题。

线程同步的是现实是通过锁来实现的：当一个线程对共享资源进行修改时对该资源上锁，其他线程无法访问该资源，直到这个线程完成了修改，退出访问之后对该资源解锁，其他线程才能访问修改该资源。

上面提到的资源很多情况下是指代的代码块。

**对象**

threading.Lock

**使用方法**

1 实例化对象

lock = threading.Lock()

2 调用acquire()方法上锁

lock.acquire()

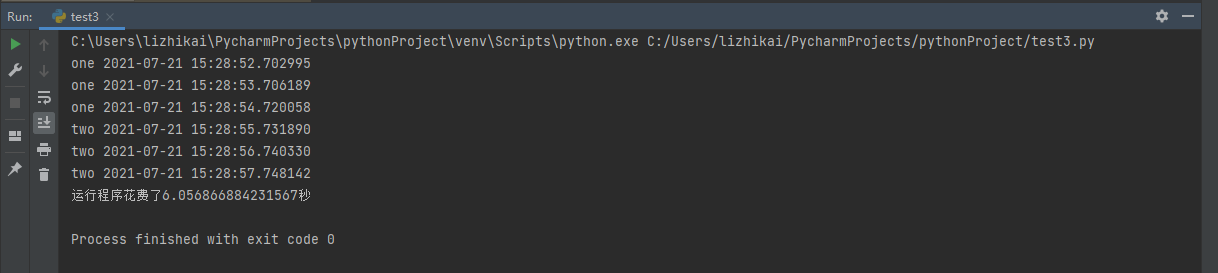
3 调用release()方法解锁

lock.acquire()

**代码示例**

|  |
| --- |
| import time  from datetime import datetime  from threading import Lock  lock = Lock()  def Test(name):  lock.acquire()  for i in range(3):  print(name, datetime.now())  time.sleep(1)  lock.release()  def main():  # 调用threading.Thread函数，target参数是要执行的函数，args是要传入的参数，为元组类型，有多少参数就往里写多少  t1 = threading.Thread(target=Test, args=("one",))  t2 = threading.Thread(target=Test, args=("two",))  t1.start() # 启动线程  t2.start()  t1.join() # 必须等子线程运行完了父进程才可以运行  t2.join()  if \_\_name\_\_ == '\_\_main\_\_':  start = time.time()  main()  end = time.time()  print("运行程序花费了%s秒" % (end - start)) |

**执行结果**



**四 线程池 进程池**

线程池是为了解决多线程不断进行启动、运行、结束动作消耗内存的问题而产生的，思想是一个线程完成了当前任务后不再直接结束，而是接着去执行下一个未被完成的任务，知道所有的任务都被执行完毕，线程池中的线程才会结束。

**调用模块**

concurrent.futures

**实例化对象**

concurrent.futures.ThreadPoolExecutor(max\_workers=10)

**常用方法**

submit():用于多线程调用不同任务的场景，但是也使用调用一个任务的场景

map():用于多线程调用同一个任务的场景

上述两种方法用法是一样的，只不过使用场景不同而已。

**代码实例**

|  |
| --- |
| #!/usr/local/python3.6.3/bin/python3.6  # coding = utf-8  import socket  import datetime  import re  from concurrent.futures import ThreadPoolExecutor, wait  DEBUG = False  # 判断ip地址输入是否符合规范  def check\_ip(ipAddr):  compile\_ip = re.compile(  '^(1\d{2}|2[0-4]\d|25[0-5]|[1-9]\d|[1-9])\.(1\d{2}|2[0-4]\d|25[0-5]|[1-9]\d|\d)\.(1\d{2}|2[0-4]\d|25[0-5]|[1-9]\d|\d)\.(1\d{2}|2[0-4]\d|25[0-5]|[1-9]\d|\d)$')  if compile\_ip.match(ipAddr):  return True  else:  return False  # 扫描端口程序  def portscan(ip, port):  try:  s = socket.socket()  s.settimeout(0.2)  s.connect((ip, port))  openstr = f'[+] {ip} port:{port} open'  print(openstr)  except Exception as e:  if DEBUG is True:  print(ip + str(port) + str(e))  else:  return f'[+] {ip} port:{port} error'  finally:  s.close  # 主程序,利用ThreadPoolExecutor创建600个线程同时扫描端口  def main():  while True:  ip = input("请输入ip地址:")  if check\_ip(ip):  start\_time = datetime.datetime.now()  executor = ThreadPoolExecutor(max\_workers=600)  t = [executor.submit(portscan, ip, n) for n in range(1, 65536)]  if wait(t, return\_when='ALL\_COMPLETED'):  end\_time = datetime.datetime.now()  print("扫描完成,用时:", (end\_time - start\_time).seconds)  break  if \_\_name\_\_ == '\_\_main\_\_':  main() |

1. 创建线程池方法精炼

a 实例化对象，同时调用线程执行任务；使用with方法进行实例化，这样不需要手动关闭线程池。

with ThreadPoolExecutor(max\_workers=600) as executor:

t = [executor.submit(portscan, ip, n) for n in range(1, 65536)]

也可以这样：

executor = ThreadPoolExecutor(max\_workers=600)

t = [executor.submit(portscan, ip, n) for n in range(1, 65536)]

b 等待线程结束获取数据

调用wait()函数，该函数返回值为布尔值：

方法和参数

wait(fs, timeout=None, return\_when=All\_COMPLETED)

fs 表示需要执行的序列

timeout 等待的最大时间，如果超过这个时间

return\_when 表示wait返回结果的结果，默认为ALLP\_COMPLETED全部执行完成后再返回结果

代码示例：

if wait(t, return\_when='ALL\_COMPLETED'):

end\_time = datetime.datetime.now()

print("扫描完成,用时:", (end\_time - start\_time).seconds)

break