# 多线程和并发编程

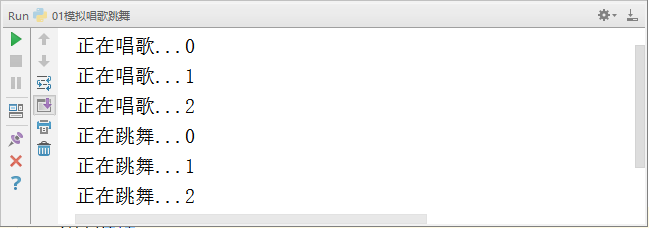
## 多任务

在现实生活中，有很多场景中的事情是同时进行的。例如：开车的时候，手和脚需要共同操作来完成驾驶。再比如演唱会中唱歌和跳舞也是同时进行的。如果把唱歌和跳舞这两件事情分开依次来完成，估计效果不是很好。示例代码如下：

**【示例】模拟唱歌跳舞**

|  |
| --- |
| from time import sleep  def sing():  for i in range(3):  print('正在唱歌...%d'%i)  sleep(1)  def dance():  for i in range(3):  print('正在跳舞...%d'%i)  sleep(1)  if \_\_name\_\_=='\_\_main\_\_':  sing()  dance() |

执行结果如图所示：



从上面的示例可以看出，程序并没有完成唱歌和跳舞同时进行的要求，如果想要实现“唱歌和跳舞”同时进行，那么就需要一个新的方法，叫做：多任务。

什么叫做多任务呢？简单的说，就是操作系统可以同时进行多个任务。例如：你一边在使用浏览器上网，一边在听MP3，一边在用word赶作业，这就是多任务。



现在，多核CPU已经非常普及了，但是，即使过去的单核CPU，也可以执行多任务。由于CPU执行代码都是顺序执行的，那么，单核CPU是怎么执行多任务的呢？

操作系统轮流让各个任务交替执行，任务1执行0.01秒，切换到任务2，任务2执行0.01秒，再切换到任务3，执行0.01秒……这样反复执行下去。表面上看，每个任务都是交替执行的，但是，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。

真正的并行执行多任务只能在多核CPU上实现，但是，由于任务数量远远多于CPU的核心数量，所以，操作系统也会自动把很多任务轮流调度到每个核心上执行。

## 进程

程序编写完没有运行称之为程序。正在运行的代码就是进程。在Python3语言中，对多进程支持的是multiprocessing模块和subprocess模块。multiprocessing模块为在子进程中运行任务、通讯和共享数据，以及执行各种形式的同步提供支持。

### 进程创建

Python提供了非常好用的多进程包multiprocessing，只需要定义一个函数，Python会完成其他所有事情。借助这个包，可以轻松完成从单进程到并发执行的转换。multiprocessing支持子进程、通信和共享数据。语法格式如下：

|  |
| --- |
| Process（[group [, target [, name [, args [, kwargs]]]]]） |

其中target表示调用对象，args表示调用对象的位置参数元组。kwargs表示调用对象的字典。name为别名。group参数未使用，值始终为None。

构造函数简单地构造了一个Process进程，Process的实例方法、Process的实例属性如下表所示。

**Process实例方法表**

|  |  |
| --- | --- |
| **方法** | **描述** |
| is\_alive() | 如果p仍然运行，返回True |
| join（[timeout]） | 等待进程p终止。Timeout是可选的超时期限，进程可以被链接无数次，但如果连接自身则会出错 |
| run() | 进程启动时运行的方法。默认情况下，会调用传递给Process构造函数的target。定义进程的另一种方法是继承Process类并重新实现run()函数 |
| start() | 启动进程，这将运行代表进程的子进程，并调用该子进程中的run()函数 |
| terminate() | 强制终止进程。如果调用此函数，进程p将被立即终止，同时不会进行任何清理动作。如果进程p创建了它自己的子进程，这些进程将变为僵尸进程。使用此方法时需要特别小心。如果p保存了一个锁或参与了进程间通信，那么终止它可能会导致死锁或I/O损坏 |

**【示例】创建子进程并执行**

|  |
| --- |
| from multiprocessing import Process  #定义子进程代码  def run\_proc():  print('子进程运行中')  if \_\_name\_\_=='\_\_main\_\_':  print('父进程运行')  p=Process(target=run\_proc)  print('子进程将要执行')  p.start() |

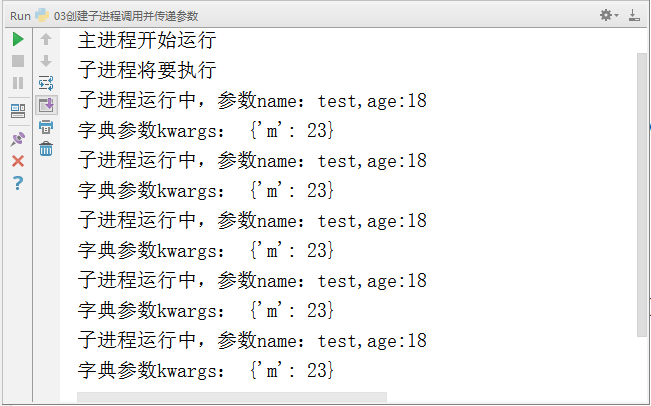
执行结果如图所示：



**【示例】创建子进程，传递参数**

|  |
| --- |
| from multiprocessing import Process  import os  from time import sleep  #创建子进程代码  def run\_proc(name,age,\*\*kwargs):  for i in range(5):  print('子进程运行中，参数name：%s,age:%d'%(name,age))  print('字典参数kwargs：',kwargs)  sleep(0.5)  if \_\_name\_\_=='\_\_main\_\_':  print('主进程开始运行')  p=Process(target=run\_proc,args=('test',18),kwargs={'m':23})  print('子进程将要执行')  p.start() |

执行结果如图所示：



**【示例】join()方法的使用**

|  |
| --- |
| from multiprocessing import Process  from time import sleep  def worker(interval):  print("work start");  sleep(interval)  print("work end");  if \_\_name\_\_ == "\_\_main\_\_":  p = Process(target = worker, args = (3,))  p.start()  #等待进程p终止  p.join()  print("主进程结束!") |

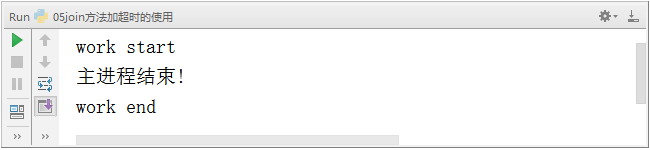
执行结果如图所示：



**【示例】join()方法中加超时的使用**

|  |
| --- |
| from multiprocessing import Process  from time import sleep  def worker(interval):  print("work start");  sleep(interval)  print("work end");  if \_\_name\_\_ == "\_\_main\_\_":  p = Process(target = worker, args = (5,))  p.start()  #等待进程p终止  p.join(3)  print("主进程结束!") |

执行结果如图所示：



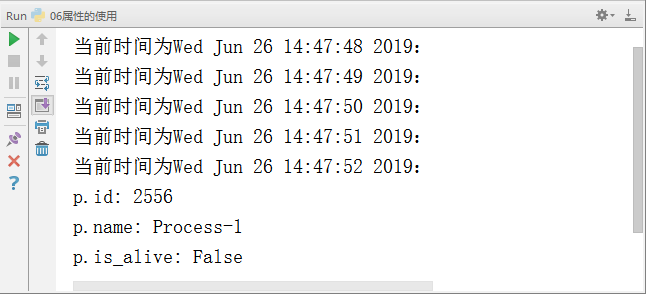
**Process实例属性表**

|  |  |
| --- | --- |
| **方法** | **描述** |
| name | 进程的名称 |
| pid | 进程的整数进程ID |

**【示例】属性的使用**

|  |
| --- |
| #导入模块  import multiprocessing  import time  #定义进程执行函数  def clock(interval):  for i in range(5):  print('当前时间为{0}：'.format(time.ctime()))  time.sleep(interval)  if \_\_name\_\_=='\_\_main\_\_':  #创建进程  p=multiprocessing.Process(target=clock,args=(1,))  #启动进程  p.start()  p.join()  #获取进程的ID  print('p.id:',p.pid)  #获取进程的名称  print('p.name:',p.name)  #判断进程是否运行  print('p.is\_alive:',p.is\_alive()) |

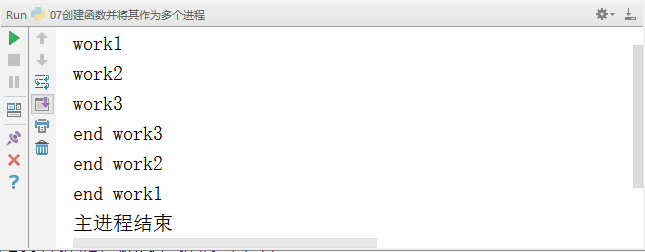
执行结果如图所示：



**【示例】创建函数并将其作为多个进程**

|  |
| --- |
| #导入模块  import multiprocessing  import time  #创建进程调用函数  def work1(interval):  print('work1')  time.sleep(interval)  print('end work1')  def work2(interval):  print('work2')  time.sleep(interval)  print('end work2')  def work3(interval):  print('work3')  time.sleep(interval)  print('end work3')  if \_\_name\_\_=='\_\_main\_\_':  #创建进程对象  p1=multiprocessing.Process(target=work1,args=(4,))  p2=multiprocessing.Process(target=work2,args=(3,))  p3=multiprocessing.Process(target=work3,args=(2,))  #启动进程  p1.start()  p2.start()  p3.start()  p1.join()  p2.join()  p3.join()  print('主进程结束') |

执行结果如下图所示：



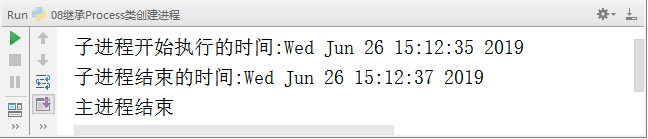
### 进程的创建-Process子类

创建进程的方式还可以使用类的方式，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，示例如下：

**【示例】继承Process的类，重写run()方法创建进程**

|  |
| --- |
| #导入模块  from multiprocessing import Process  import time  #定义线程类  class ClockProcess(Process):  def \_\_init\_\_(self,interval):  Process.\_\_init\_\_(self)  self.interval=interval  def run(self):  print('子进程开始执行的时间:{}'.format(time.ctime()))  time.sleep(self.interval)  print('子进程结束的时间:{}'.format(time.ctime()))  if \_\_name\_\_=='\_\_main\_\_':  #创建进程  p=ClockProcess(2)  #启动进程  p.start()  p.join()  print('主进程结束') |

执行结果如下图所示：



### 进程池

在利用Python进行系统管理的时候，特别是同时操作多个文件目录，或者远程控制多台主机，并行操作可以节约大量的时间。当被操作对象数目不大时，可以直接利用multiprocessing中的Process动态成生多个进程，十几个还好，但如果是上百个，上千个目标，手动的去限制进程数量却又太过繁琐，此时可以发挥进程池的功效。

Pool可以提供指定数量的进程，供用户调用，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规定最大值，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。Pool的语法格式如下：

|  |
| --- |
| Pool（[numprocess [, initializer [, initargs]]]） |

其中numprocess是要创建的进程数。如果省略此参数，将使用cpu\_count()的值。Initializer是每个工作进程启动时要执行的可调用对象。Initargs是要传递给initializer的参数元祖。Initializer默认为None。

Pool类的实例方法如下表所示。

**Pool实例方法表**

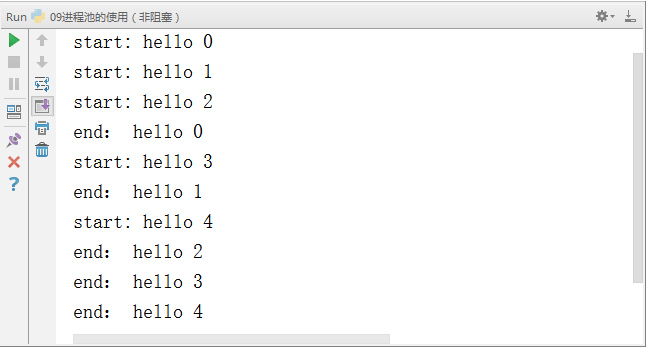
|  |  |
| --- | --- |
| **方法** | **描述** |
| apply（func [,args [,kwargs]]） | 在一个池工作进程中执行函数（\*args，\*\*kwargs），然后返回结果。 |
| apply\_async（func [, args [,kwargs [,callback ] ] ]） | 在一个池工作进程中异步地执行函数（\*args，\*\*kwargs），然后返回结果。此方法的结果是AsyncResult类的实例，稍后可用于获得最终结果。Callback是可调用对象，接受输入参数。当func的结果变为可用时，将立即传递给callback。Callback禁止执行任何阻塞操作，否则将阻塞接收其他异步操作中的结果 |
| close() | 关闭进程池，防止进行进一步操作。如果还有挂起的操作，它们将在工作进程终止之前完成 |
| join() | 等待所有工作进程退出。此方法只能在close（）或者terminate（）方法之后调用 |
| imap( func，iterable [ ,chunksize] ) | map（）函数的版本之一，返回迭代器而非结果列表 |
| imap\_unordered( func，iterable [，chunksize] ) | 同imap（）函数一样，只是结果的顺序根据从工作进程接收到的时间任意确定 |
| map( func，iterable [，chunksize] ) | 将可调用对象func应用给iterable中的所有项，然后以列表的形式返回结果。通过将iterable划分为多块并将工作分派给工作进程，可以并行地执行这项操作。chunksize指定每块中的项数。如果数量较大，可以增大chunksize的值来提升性能 |
| map\_async( func，iterable [，chunksize [，callback]] ) | 同map（）函数，但结果的返回是异步的。返回值是AsyncResult类的实例，稍后可用与获取结果。Callback是指接受一个参数的可调对象。如果提供callable，当结果变为可用时，将使用结果调用callable |
| terminate() | 立即终止所有工作进程，同时不执行任何清理或结束任何挂起工作。如果p被垃圾回收，将自动调用此函数 |
| get( [ timeout] ) | 返回结果，如果有必要则等待结果到达。Timeout是可选的超时。如果结果在指定时间内没有到达，将引发multiprocessing.TimeoutError异常。如果远程操作中引发了异常，它将在调用此方法时再次被引发 |
| ready() | 如果调用完成，则返回True |
| sucessful() | 如果调用完成且没有引发异常，返回True。如果在结果就绪之前调用此方法，将引发AssertionError异常 |
| wait( [timeout] ) | 等待结果变为可用。Timeout是可选的超时 |

|  |
| --- |
| **注意：**  apply\_async(func[, args[, kwds[, callback]]]) 它是非阻塞，apply(func[, args[, kwds]])是阻塞的 |

**【示例】进程池的使用（非阻塞）**

|  |
| --- |
| import multiprocessing  import time  def func(msg):  print("start:", msg)  time.sleep(3)  print("end：",msg)  if \_\_name\_\_ == "\_\_main\_\_":  pool = multiprocessing.Pool(processes = 3)  for i in range(5):  msg = "hello %d" %(i)  #维持执行的进程总数为processes，当一个进程执行完毕后会添加新的进程进去  pool.apply\_async(func, (msg, ))  pool.close()#进程池关闭之后不再接收新的请求  #调用join之前，先调用close函数，否则会出错。  # 执行完close后不会有新的进程加入到pool,join函数等待所有子进程结束  pool.join() |

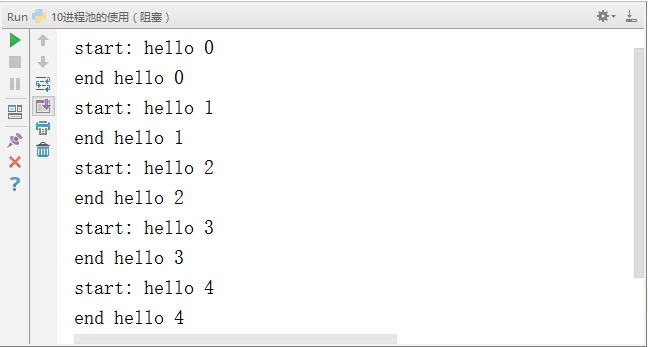
执行结果如图所示：



**【示例】使用进程池（阻塞）**

|  |
| --- |
| import multiprocessing  import time  def func(msg):  print("start:", msg)  time.sleep(3)  print("end",msg)  if \_\_name\_\_ == "\_\_main\_\_":  pool = multiprocessing.Pool(processes = 3)  for i in range(5):  msg = "hello %d" %(i)  #维持执行的进程总数为processes，当一个进程执行完毕后会添加新的进程进去  pool.apply(func, (msg, ))  pool.close()  #调用join之前，先调用close函数，否则会出错。  # 执行完close后不会有新的进程加入到pool,join函数等待所有子进程结束  pool.join() |

执行结果如图所示：



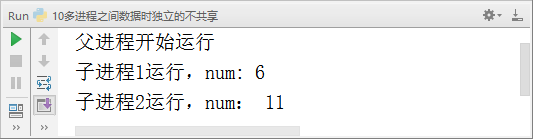
### 进程间通信

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。

**【示例】多个进程中数据不共享**

|  |
| --- |
| from multiprocessing import Process  num=1  def work1():  global num  num+=5  print('子进程1运行，num:',num)  def work2():  global num  num += 10  print('子进程2运行，num：',num)  if \_\_name\_\_=='\_\_main\_\_':  print('父进程开始运行')  p1=Process(target=work1)  p2=Process(target=work2)  p1.start()  p2.start()  p1.join()  p2.join() |

执行结果如图所示：



Queue是多进程安全的队列，可以使用Queue实现多进程之间的数据传递。put方法用以插入数据到队列中，put方法还有两个可选参数：blocked和timeout。如果blocked为True（默认值），并且timeout为正值，该方法会阻塞timeout指定的时间，直到该队列有剩余的空间。如果超时，会抛出Queue.full异常。如果blocked为False，但该Queue已满，会立即抛出Queue.full异常。

get方法可以从队列读取并且删除一个元素。同样，get方法有两个可选参数：blocked和timeout。如果blocked为True（默认值），并且timeout为正值，那么在等待时间内没有取到任何元素，会抛出Queue.Empty异常。如果blocked为False，有两种情况存在，如果Queue有一个值可用，则立即返回该值，否则，如果队列为空，则立即抛出Queue.Empty异常。

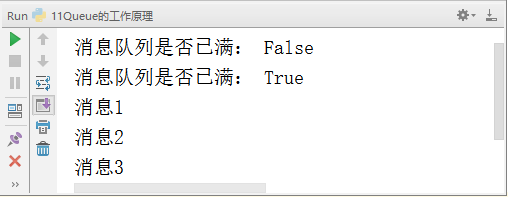
**Queue实例方法表**

|  |  |
| --- | --- |
| **方法** | **描述** |
| cancle\_join\_thread() | 不会在进程退出时自动连接后台线程。这可以防止join\_thread()方法阻塞 |
| close() | 关闭队列，防止队列中加入更多数据。调用此方法时，后台线程将继续写入那些已入队列尚未写入数据，但将在此方法完成时马上关闭 |
| empty() | 如果调用此方法时q为空，返回True |
| full() | 如果q已满，返回True |
| get([block [,timeout]) | 返回q中的一个项。如果q为空，此方法将阻塞，直到队列中有项可用为止。Block用于控制阻塞行为，默认为True。如果设置为False，将引发Queue.Empty异常(定义在Queue模块中)。Timeout是可选超时时间，用在阻塞模式中。如果在指定的时间间隔内没有项变为可用，将引发Queue.Empty异常 |
| join\_thread() | 连接队列的后台线程。此方法用于在调用q.close()方法之后，等待所有队列项被消耗。默认情况下此方法由不是q的原始创建者的所有进程调用。调用q.cancle\_join\_thread()方法可以禁止这种行为 |
| put(item [ , block [, timeout]]) | 将item放入队列。如果队列已满，此方法将阻塞至有空间可用为止。Block控制阻塞行为，默认为True。如果设置为False，将引发Queue.Empty异常(定义在Queue模块中)。Timeout指定在阻塞模式中等待可用时空间的时间长短。超时后将引发Queue.Full异常。 |
| qsize() | 返回目前队列中项的正确数量。 |
| joinableQueue([maxsize]) | 创建可连接的共享进程队列。这就像是一个Queue对象，但队列允许项的消费者通知生产者项已经被成功处理。通知进程是使用共享的信号和条件变量来实现的 |
| task\_done() | 消费者使用此方法发出信号，表示q.get()返回的项已经被处理。如果调用此方法的次数大于从队列中删除的项的数量，将引发ValueError异常 |
| join() | 生产者使用此方法进行阻塞，知道队列中的所有项均被处理。阻塞将持续到位队列中的每个项均调用q.task\_done()方法为止 |

**【示例】Queue的方法使用**

|  |
| --- |
| from multiprocessing import Queue  q=Queue(3)  q.put('消息1')  q.put('消息2')  print('消息队列是否已满：',q.full())  q.put('消息3')  print('消息队列是否已满：',q.full())  # q.put('消息4')因为消息队列已满，需要直接写入需要等待，如果超时会抛出异常，  # 所以写入时候需判断，消息队列是否已满  if not q.full():  q.put('消息4')  #同理读取消息时，先判断消息队列是否为空，再读取  if not q.empty():  for i in range(q.qsize()):  print(q.get()) |

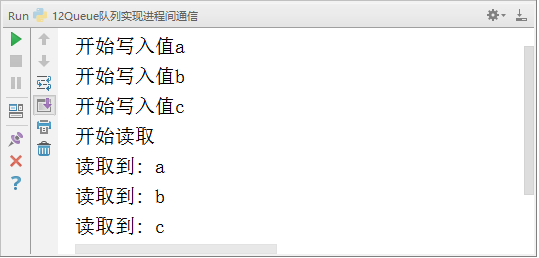
执行结果如图所示：



**【示例】Queue队列实现进程间通信**

|  |
| --- |
| from multiprocessing import \*  import time  def write(q):  #将列表中的元素写入队列中  for i in ["a","b","c"]:  print('开始写入值%s' % i)  q.put(i)  time.sleep(1)  #读取  def read(q):  print('开始读取')  while True:  if not q.empty():  print('读取到:',q.get())  time.sleep(1)  else:  break  if \_\_name\_\_=='\_\_main\_\_':  #创建队列  q=Queue()  #创建写入进程  pw=Process(target=write,args=(q,))  pr=Process(target=read,args=(q,))  #启动进程  pw.start()  pw.join()  pr.start()  pr.join() |

执行结果如图所示：



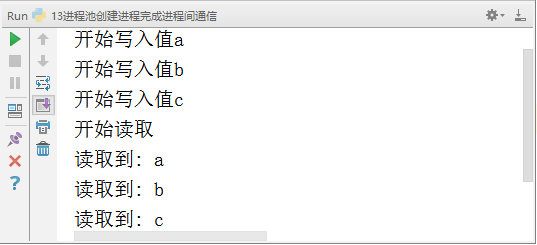
如果使用Pool创建进程，就需要使用multiprocessing.Manager()中的Queue()来完成进程间的通信，而不是multiprocessing.Queue()，否则会抛出如下异常。

|  |
| --- |
| RuntimeError: Queue objects should only be shared between processes through inheritance |

**【示例】进程池创建进程完成进程之间的通信**

|  |
| --- |
| from multiprocessing import Manager,Pool  import time  def write(q):  #将列表中的元素写入队列中  for i in ["a","b","c"]:  print('开始写入值%s' % i)  q.put(i)  time.sleep(1)  #读取  def read(q):  print('开始读取')  while True:  if not q.empty():  print('读取到:',q.get())  time.sleep(1)  else:  break  if \_\_name\_\_=='\_\_main\_\_':  #创建队列  q=Manager().Queue()  #创建进程池  p=Pool(3)  #使用阻塞模式创建进程  p.apply(write,(q,))  p.apply(read,(q,))  p.close()  p.join() |

执行结果如图所示：



## 线程

线程也是实现多任务的一种方式，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。一个进程可以拥有多个并行的线程，其中每一个线程，共享当前进程的资源。

|  |  |  |
| --- | --- | --- |
| 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 | 每一个进程都有独立的代码和数据空间，进程间的切换会有较大的开销 | 线程可以看出是轻量级的进程，多个线程共享内存，线程切换的开销小 |
| 所处环境 | 在操作系统中，同时运行的多个任务 | 在程序中多个顺序流同时执行 |
| 分配内存 | 系统在运行的时候为每一个进程分配不同的内存区域 | 线程所使用的资源是他所属进程的资源 |
| 包含关系 | 一个进程内可以拥有多个线程 | 线程是进程的一部分，所有线程有时候称为是轻量级的进程 |

进程和线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。

在Python程序中，可以通过“\_thread”和threading（推荐使用）这两个模块来处理线程。在Python3中，thread模块已经废弃。可以使用threading模块代替。所以，在Python3中不能再使用thread模块，但是为了兼容Python3以前的程序，在Python3中将thread模块重命名为“\_thread”。

### \_thread模块

在Python程序中，可以通过两种方式来使用线程：使用函数或者使用类来包装线程对象。当使用thread模块来处理线程时，可以调用里面的函数start\_new\_thread()来生成一个新的线程，语法格式如下：

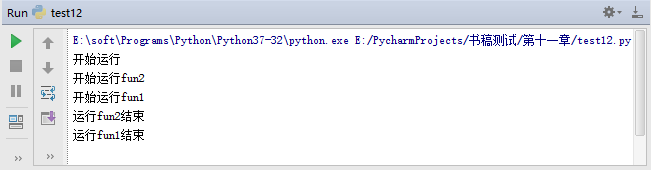
|  |
| --- |
| \_thread.start\_new\_thread ( function, args[, kwargs] ) |

其中function是线程函数；args表示传递给线程函数的参数，他必须是个tuple类型；kwargs 是可选参数。

**【示例】使用\_thread模块创建线程**

|  |
| --- |
| import \_thread  import time  def fun1():  print('开始运行fun1')  time.sleep(4)  print('运行fun1结束')  def fun2():  print('开始运行fun2')  time.sleep(2)  print('运行fun2结束')  if \_\_name\_\_=='\_\_main\_\_':  print('开始运行')  #启动一个线程运行函数fun1  \_thread.start\_new\_thread(fun1,())  #启动一个线程运行函数fun2  \_thread.start\_new\_thread(fun2,())  time.sleep(6) |

执行结果如图所示：

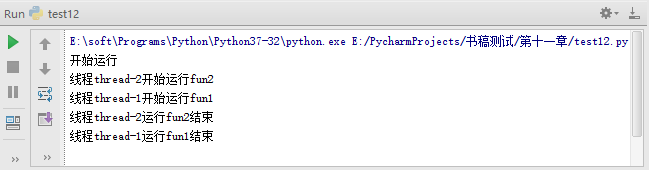


从程序运行结果可以看出，在fun2函数中调用了sleep函数休眠，当休眠期间，会释放CPU的计算资源，这时fun1抢占了CPU资源开始执行。

**【示例】为线程传递参数**

|  |
| --- |
| import \_thread  import time  def fun1(thread\_name,delay):  print('线程{0}开始运行fun1'.format(thread\_name))  time.sleep(delay)  print('线程{0}运行fun1结束'.format(thread\_name))  def fun2(thread\_name,delay):  print('线程{0}开始运行fun2'.format(thread\_name))  time.sleep(2)  print('线程{0}运行fun2结束'.format(thread\_name))  if \_\_name\_\_=='\_\_main\_\_':  print('开始运行')  #启动一个线程运行函数fun1  \_thread.start\_new\_thread(fun1,('thread-1',4))  #启动一个线程运行函数fun2  \_thread.start\_new\_thread(fun2,('thread-2',2))  time.sleep(6) |

执行结果如图所示：



从输出结果可以看出，由于每个线程函数的休眠时间可能都不相同，所以随机输出了这个结果，每次运行程序，输出的结果是不一样的。

### 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：

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

在Python3程序中，对多线程支持最好的是threading模块，使用这个模块，可以灵活地创建多线程程序，并且可以在多线程之间进行同步和通信。在Python3程序中，可以通过如下两种方式来创建线程：

* 通过threading.Thread直接在线程中运行函数
* 通过继承类threading.Thread来创建线程

在Python中使用threading.Thread的基本语法格式如下所示：

|  |
| --- |
| Thread(group=None, target=None, name=None, args=(), kwargs={}) |

其中target: 要执行的方法；name: 线程名；args/kwargs: 要传入方法的参数。

Thread类的方法如表所示：

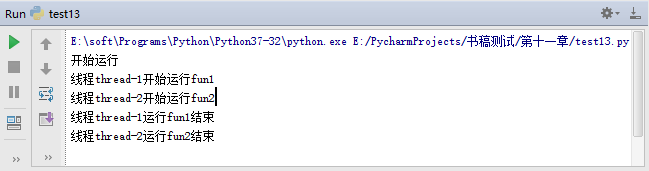
**Thread类的方法**

|  |  |
| --- | --- |
| **方法名** | **描述** |
| run() | 用以表示线程活动的方法 |
| start() | 启动线程活动 |
| 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 |
| isAlive() | 返回线程是否活动的 |
| getName() | 返回线程名 |
| setName() | 设置线程名 |

**【示例】threading.Thread直接创建线程**

|  |
| --- |
| import threading  import time  def fun1(thread\_name,delay):  print('线程{0}开始运行fun1'.format(thread\_name))  time.sleep(delay)  print('线程{0}运行fun1结束'.format(thread\_name))  def fun2(thread\_name,delay):  print('线程{0}开始运行fun2'.format(thread\_name))  time.sleep(delay)  print('线程{0}运行fun2结束'.format(thread\_name))  if \_\_name\_\_=='\_\_main\_\_':  print('开始运行')  #创建线程  t1=threading.Thread(target=fun1,args=('thread-1',2))  t2=threading.Thread(target=fun2,args=('thread-2',4))  t1.start()  t2.start() |

执行结果如图所示：

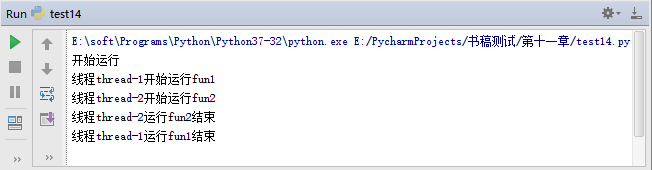


在Python中，通过继承类threading.Thread的方式来创建一个线程。这种方法只要重写类threading.Thread中的方法run()，然后再调用方法start()就能创建线程，并运行方法run()中的代码。

**【示例】继承threading.Thread类创建线程**

|  |
| --- |
| import threading  import time  def fun1(delay):  print('线程{0}开始运行fun1'.format(threading.current\_thread().getName()))  time.sleep(delay)  print('线程{0}运行fun1结束'.format(threading.current\_thread().getName()))  def fun2(delay):  print('线程{0}开始运行fun2'.format(threading.current\_thread().getName()))  time.sleep(2)  print('线程{0}运行fun2结束'.format(threading.current\_thread().getName()))  #创建线程类继承threading.Thread  class MyThread(threading.Thread):  #重写父类的构造方法，其中func是线程函数，args是传入线程的参数,name是线程名  def \_\_init\_\_(self,func,name,args):  super().\_\_init\_\_(target=func,name=name,args=args)  #重写父类的run()方法  def run(self):  self.\_target(\*self.\_args)  if \_\_name\_\_=='\_\_main\_\_':  print('开始运行')  #创建线程  t1=MyThread(fun1,'thread-1',(2,))  t2=MyThread(fun2,'thread-2',(4,))  t1.start()  t2.start() |

执行结果如图所示：



在调用Thread类的构造方法时，需要将线程函数、参数等值传入构造方法，其中name表示线程的名字，如果不指定这个参数，默认的线程名字格式为Thread-1、Thread-2。每一个传入构造方法的参数值，在Thread类中都有对应的成员变量保存这些值，这些成员变量都以下划线(\_)开头，如果\_target、\_args等。在run方法中需要使用这些变量调用传入的线程函数，并为线程函数传递参数。

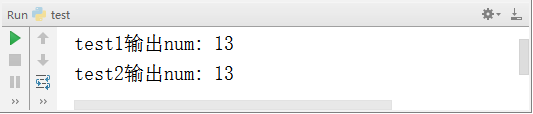
## 线程共享全局变量

在一个进程内所有线程共享全局变量，多线程之间的数据共享比多进程要好。但是可能造成多个进程同时修改一个变量（即线程非安全），可能造成混乱。

**【示例】线程共享全局变量**

|  |
| --- |
| import time  from threading import \*  #定义全局变量num  num=10  def test1():  global num  for i in range(3):  num+=1  print('test1输出num:',num)  def test2():  global num  print('test2输出num:',num)  if \_\_name\_\_=='\_\_main\_\_':  t1=Thread(target=test1)  t2=Thread(target=test2)  t1.start()  t1.join()  t2.start()  t2.join() |

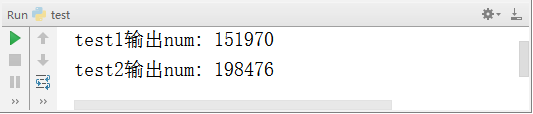
执行结果如图所示：



**【示例】线程共享全局变量存在问题**

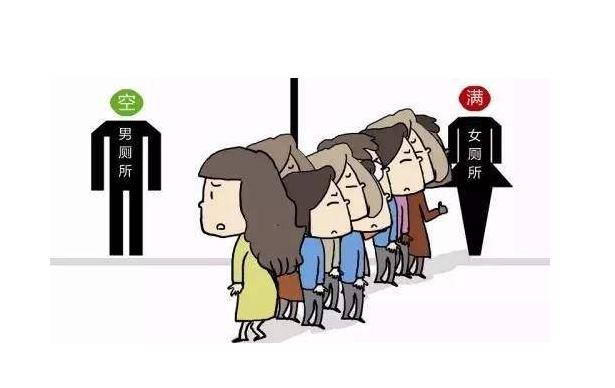
|  |
| --- |
| import time  from threading import \*  #定义全局变量num  num=0  def test1():  global num  for i in range(100000):  num+=1  print('test1输出num:',num)  def test2():  global num  for i in range(100000):  num+=1  print('test2输出num:',num)  if \_\_name\_\_=='\_\_main\_\_':  t1=Thread(target=test1)  t2=Thread(target=test2)  t1.start()  t2.start()  t1.join()  t2.join() |

执行结果如图所示：



## 互斥锁

如果多个线程共同对某个数据修改，则可能出现不可预料的结果，为了保证数据的正确性，需要对多个线程进行同步。最简单的同步机制就是引入互斥锁。



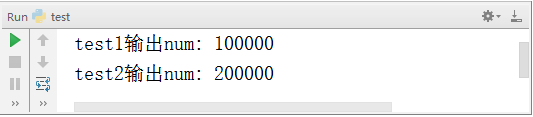
锁有两种状态——锁定和未锁定。某个线程要更改共享数据时，先将其锁定，此时资源的状态为“锁定”，其他线程不能更改；直到该线程释放资源，将资源的状态变成“非锁定”状态，其他的线程才能再次锁定该资源。互斥锁保证了每次只有一个线程进行写入操作，从而保证了多线程情况下数据的正确性。

使用 Thread 对象的 Lock 可以实现简单的线程同步，有上锁 acquire 方法和 释放release 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。

**【示例】互斥锁**

|  |
| --- |
| import time  from threading import Thread,Lock  #定义全局变量num  num=0  #创建一把互斥锁  mutex=Lock()  def test1():  global num  '''  在两个线程中都调用上锁的方法，则这两个线程就会抢着上锁，  如果有1方成功上锁，那么导致另外一方会堵塞（一直等待）直到这个锁被解开  '''  mutex.acquire()#上锁  for i in range(100000):  num+=1  mutex.release()  print('test1输出num:',num)  def test2():  global num  mutex.acquire() # 上锁  for i in range(100000):  num+=1  mutex.release()  print('test2输出num:',num)  if \_\_name\_\_=='\_\_main\_\_':  t1=Thread(target=test1)  t2=Thread(target=test2)  t1.start()  t2.start()  t1.join()  t2.join() |

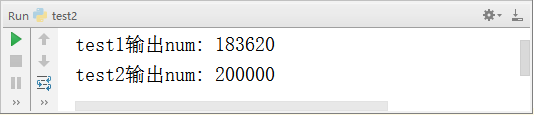
执行结果如图所示：



**【示例】互斥锁改进**

|  |
| --- |
| import time  from threading import Thread,Lock  #定义全局变量num  num=0  #创建一把互斥锁  mutex=Lock()  def test1():  global num  '''  在两个线程中都调用上锁的方法，则这两个线程就会抢着上锁，  如果有1方成功上锁，那么导致另外一方会堵塞（一直等待）直到这个锁被解开  '''  for i in range(100000):  mutex.acquire() # 上锁  num+=1  mutex.release()  print('test1输出num:',num)  def test2():  global num  for i in range(100000):  mutex.acquire() # 上锁  num+=1  mutex.release()  print('test2输出num:',num)  if \_\_name\_\_=='\_\_main\_\_':  t1=Thread(target=test1)  t2=Thread(target=test2)  t1.start()  t2.start()  t1.join()  t2.join() |

执行结果如图所示：

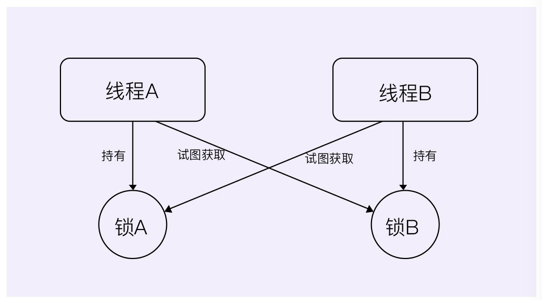


## 死锁

在线程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。

**【示例】死锁**

|  |
| --- |
| import time  from threading import Thread,Lock  import threading  mutexA=threading.Lock()  mutexB=threading.Lock()  class MyThread1(Thread):  def run(self):  if mutexA.acquire():  print(self.name,'执行')  time.sleep(1)  if mutexB.acquire():  print(self.name,'执行')  mutexB.release()  mutexA.release()  class MyThread2(Thread):  def run(self):  if mutexB.acquire():  print(self.name,'执行')  time.sleep(1)  if mutexA.acquire():  print(self.name,'执行')  mutexA.release()  mutexB.release()  if \_\_name\_\_ == '\_\_main\_\_':  t1=MyThread1()  t2=MyThread2()  t1.start()  t2.start() |



## 线程同步的应用

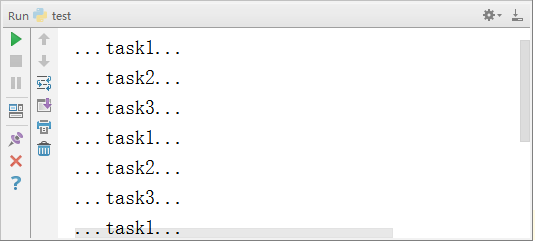
同步就是协同步调，按预定的先后次序进行运行。例如：开会。“同”字指协同、协助、互相配合。

如进程、线程同步，可以理解为进程或线程A和B一块配合，A执行到一定程度时要依靠B的某个结果，于是停下来，示意B运行，B运行后将结果给A，A继续运行。

**【示例】线程同步应用**

|  |
| --- |
| import time  from threading import Thread,Lock  import threading  lock1=Lock()  lock2=Lock()  lock3=Lock()  lock2.acquire()  lock3.acquire()  class Task1(Thread):  def run(self):  while True:  if lock1.acquire():  print('...task1...')  time.sleep(1)  lock2.release()  class Task2(Thread):  def run(self):  while True:  if lock2.acquire():  print('...task2...')  time.sleep(1)  lock3.release()  class Task3(Thread):  def run(self):  while True:  if lock3.acquire():  print('...task3...')  time.sleep(1)  lock1.release()  if \_\_name\_\_ == '\_\_main\_\_':  t1=Task1()  t2=Task2()  t3=Task3()  t1.start()  t2.start()  t3.start() |

执行结果如图所示：



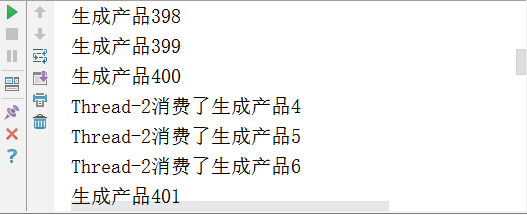
## 生产者消费者模式

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生产者和消费者模式

生产者消费者模式通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者之间不直接通信。生产者生产商品，然后将其放到类似队列的数据结构中，消费者不找生产者要数据，而是直接从队列中取。这里使用queue模块来提供线程间通信的机制，也就是说，生产者和消费者共享一个队列。生产者生产商品后，会将商品添加到队列中。消费者消费商品，会从队列中取出商品。  
**【示例】生产者-消费者模型**

|  |
| --- |
| import time  import threading  from queue import Queue  class Producer(threading.Thread):  def run(self):  global queue  count=0  while True:  if queue.qsize()<1000:  for i in range(100):  count += 1  msg = '生成产品' + str(count)  queue.put(msg)  print(msg)  time.sleep(0.5)  class Consumer(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()  p=Producer()  p.start()  time.sleep(1)  c=Consumer()  c.start() |

执行结果如图所示：



## ThreadLocal

我们知道多线程环境下，每一个线程均可以使用所属进程的全局变量。如果一个线程对全局变量进行了修改，将会影响到其他所有的线程对全局变量的计算操作，从而出现数据混乱，即为脏数据。为了避免多个线程同时对变量进行修改，引入了线程同步机制，通过互斥锁来控制对全局变量的访问。所以有时候线程使用局部变量比全局变量好，因为局部变量只有线程自身可以访问，同一个进程下的其他线程不可访问。

但是局部变量也是有问题，就是在函数调用的时候，传递起来很麻烦。示例如下：

**【示例】局部变量作为参数传递**

|  |
| --- |
| def process\_student(name):  std=Student(name)  do\_task1(std)  do\_task2(std)    def do\_task1(std):  do\_sub\_task1(std)  do\_sub\_task2(std)    def do\_task2(std):  do\_sub\_task1(std)  do\_sub\_task2(std) |

从上面的实例可以看到每个函数一层一层调用都需要传递std参数，非常麻烦，如果使用全局变量也不行，因为每个线程处理不同的Student对象，不能共享。因此 Python 还提供了ThreadLocal 变量，它本身是一个全局变量，但是每个线程却可以利用它来保存属于自己的私有数据，这些私有数据对其他线程也是不可见的。

**【示例】ThreadLocal的使用**

|  |
| --- |
| import threading  # 创建全局ThreadLocal对象:  local = threading.local()  def process\_student():  # 获取当前线程关联的name:  student\_name = local.name  print('线程名：%s 学生姓名:%s' % (threading.current\_thread().name,student\_name))  def process\_thread(name):  # 绑定ThreadLocal的name:  local.name = name  process\_student()  t1 = threading.Thread(target=process\_thread, args=('张三',), name='Thread-A')  t2 = threading.Thread(target=process\_thread, args=('李四',), name='Thread-B')  t1.start()  t2.start()  t1.join()  t2.join() |

执行结果如下图：

