[**Python多线程学习**](http://www.cnblogs.com/tqsummer/archive/2011/01/25/1944771.html)

 一、Python中的线程使用：

    Python中使用线程有两种方式：函数或者用类来包装线程对象。

1、  函数式：调用thread模块中的start\_new\_thread()函数来产生新线程。如下例：

1. **import** time
2. **import** thread
3. **def** timer(no, interval):
4. cnt = 0
5. **while** cnt<10:
6. **print** 'Thread:(%d) Time:%s\n'%(no, time.ctime())
7. time.sleep(interval)
8. cnt+=1
9. thread.exit\_thread()

12. **def** test(): #Use thread.start\_new\_thread() to create 2 new threads
13. thread.start\_new\_thread(timer, (1,1))
14. thread.start\_new\_thread(timer, (2,2))
16. **if** \_\_name\_\_=='\_\_main\_\_':
17. test()

    上面的例子定义了一个线程函数timer,它打印出10条时间记录后退出，每次打印的间隔由interval参数决定。thread.start\_new\_thread(function, args[, kwargs])的第一个参数是线程函数（本例中的timer方法），第二个参数是传递给线程函数的参数，它必须是tuple类型，kwargs是可选参数。

    线程的结束可以等待线程自然结束，也可以在线程函数中调用thread.exit()或thread.exit\_thread()方法。

2、  创建threading.Thread的子类来包装一个线程对象，如下例：

1. **import** threading
2. **import** time
3. **class** timer(threading.Thread): #The timer class is derived from the class threading.Thread
4. **def** \_\_init\_\_(self, num, interval):
5. threading.Thread.\_\_init\_\_(self)
6. self.thread\_num = num
7. self.interval = interval
8. self.thread\_stop = False
10. **def** run(self): #Overwrite run() method, put what you want the thread do here
11. **while** **not** self.thread\_stop:
12. **print** 'Thread Object(%d), Time:%s\n' %(self.thread\_num, time.ctime())
13. time.sleep(self.interval)
14. **def** stop(self):
15. self.thread\_stop = True

18. **def** test():
19. thread1 = timer(1, 1)
20. thread2 = timer(2, 2)
21. thread1.start()
22. thread2.start()
23. time.sleep(10)
24. thread1.stop()
25. thread2.stop()
26. **return**
28. **if** \_\_name\_\_ == '\_\_main\_\_':
29. test()

    就我个人而言，比较喜欢第二种方式，即创建自己的线程类，必要时重写threading.Thread类的方法，线程的控制可以由自己定制。

threading.Thread类的使用：

1，在自己的线程类的\_\_init\_\_里调用threading.Thread.\_\_init\_\_(self, name = threadname)

Threadname为线程的名字

2， run()，通常需要重写，编写代码实现做需要的功能。

3，getName()，获得线程对象名称

4，setName()，设置线程对象名称

5，start()，启动线程

6，jion([timeout])，等待另一线程结束后再运行。

7，setDaemon(bool)，设置子线程是否随主线程一起结束，必须在start()之前调用。默认为False。

8，isDaemon()，判断线程是否随主线程一起结束。

9，isAlive()，检查线程是否在运行中。

    此外threading模块本身也提供了很多方法和其他的类，可以帮助我们更好的使用和管理线程。可以参看<http://www.python.org/doc/2.5.2/lib/module-threading.html>。

假设两个线程对象t1和t2都要对num=0进行增1运算，t1和t2都各对num修改10次，num的最终的结果应该为20。但是由于是多线程访问，有可能出现下面情况：在num=0时，t1取得num=0。系统此时把t1调度为”sleeping”状态，把t2转换为”running”状态，t2页获得num=0。然后t2对得到的值进行加1并赋给num，使得num=1。然后系统又把t2调度为”sleeping”，把t1转为”running”。线程t1又把它之前得到的0加1后赋值给num。这样，明明t1和t2都完成了1次加1工作，但结果仍然是num=1。

    上面的case描述了多线程情况下最常见的问题之一：数据共享。当多个线程都要去修改某一个共享数据的时候，我们需要对数据访问进行同步。

1、  简单的同步

最简单的同步机制就是“锁”。锁对象由threading.RLock类创建。线程可以使用锁的acquire()方法获得锁，这样锁就进入“locked”状态。每次只有一个线程可以获得锁。如果当另一个线程试图获得这个锁的时候，就会被系统变为“blocked”状态，直到那个拥有锁的线程调用锁的release()方法来释放锁，这样锁就会进入“unlocked”状态。“blocked”状态的线程就会收到一个通知，并有权利获得锁。如果多个线程处于“blocked”状态，所有线程都会先解除“blocked”状态，然后系统选择一个线程来获得锁，其他的线程继续沉默（“blocked”）。

Python中的thread模块和Lock对象是Python提供的低级线程控制工具，使用起来非常简单。如下例所示：

1. **import** thread
2. **import** time
3. mylock = thread.allocate\_lock()  #Allocate a lock
4. num=0  #Shared resource
6. **def** add\_num(name):
7. **global** num
8. **while** True:
9. mylock.acquire() #Get the lock
10. # Do something to the shared resource
11. **print** 'Thread %s locked! num=%s'%(name,str(num))
12. **if** num >= 5:
13. **print** 'Thread %s released! num=%s'%(name,str(num))
14. mylock.release()
15. thread.exit\_thread()
16. num+=1
17. **print** 'Thread %s released! num=%s'%(name,str(num))
18. mylock.release()  #Release the lock.
20. **def** test():
21. thread.start\_new\_thread(add\_num, ('A',))
22. thread.start\_new\_thread(add\_num, ('B',))
24. **if** \_\_name\_\_== '\_\_main\_\_':
25. test()

Python 在thread的基础上还提供了一个高级的线程控制库，就是之前提到过的threading。Python的threading module是在建立在thread module基础之上的一个module，在threading module中，暴露了许多thread module中的属性。在thread module中，python提供了用户级的线程同步工具“Lock”对象。而在threading module中，python又提供了Lock对象的变种: RLock对象。RLock对象内部维护着一个Lock对象，它是一种可重入的对象。对于Lock对象而言，如果一个线程连续两次进行acquire操作，那么由于第一次acquire之后没有release，第二次acquire将挂起线程。这会导致Lock对象永远不会release，使得线程死锁。RLock对象允许一个线程多次对其进行acquire操作，因为在其内部通过一个counter变量维护着线程acquire的次数。而且每一次的acquire操作必须有一个release操作与之对应，在所有的release操作完成之后，别的线程才能申请该RLock对象。

下面来看看如何使用threading的RLock对象实现同步。

1. **import** threading
2. mylock = threading.RLock()
3. num=0
5. **class** myThread(threading.Thread):
6. **def** \_\_init\_\_(self, name):
7. threading.Thread.\_\_init\_\_(self)
8. self.t\_name = name
10. **def** run(self):
11. **global** num
12. **while** True:
13. mylock.acquire()
14. **print** '\nThread(%s) locked, Number: %d'%(self.t\_name, num)
15. **if** num>=4:
16. mylock.release()
17. **print** '\nThread(%s) released, Number: %d'%(self.t\_name, num)
18. **break**
19. num+=1
20. **print** '\nThread(%s) released, Number: %d'%(self.t\_name, num)
21. mylock.release()
23. **def** test():
24. thread1 = myThread('A')
25. thread2 = myThread('B')
26. thread1.start()
27. thread2.start()
29. **if** \_\_name\_\_== '\_\_main\_\_':
30. test()

我们把修改共享数据的代码成为“临界区”。必须将所有“临界区”都封闭在同一个锁对象的acquire和release之间。

2、  条件同步

锁只能提供最基本的同步。假如只在发生某些事件时才访问一个“临界区”，这时需要使用条件变量Condition。

Condition对象是对Lock对象的包装，在创建Condition对象时，其构造函数需要一个Lock对象作为参数，如果没有这个Lock对象参数，Condition将在内部自行创建一个Rlock对象。在Condition对象上，当然也可以调用acquire和release操作，因为内部的Lock对象本身就支持这些操作。但是Condition的价值在于其提供的wait和notify的语义。

条件变量是如何工作的呢？首先一个线程成功获得一个条件变量后，调用此条件变量的wait()方法会导致这个线程释放这个锁，并进入“blocked”状态，直到另一个线程调用同一个条件变量的notify()方法来唤醒那个进入“blocked”状态的线程。如果调用这个条件变量的notifyAll()方法的话就会唤醒所有的在等待的线程。

如果程序或者线程永远处于“blocked”状态的话，就会发生死锁。所以如果使用了锁、条件变量等同步机制的话，一定要注意仔细检查，防止死锁情况的发生。对于可能产生异常的临界区要使用异常处理机制中的finally子句来保证释放锁。等待一个条件变量的线程必须用notify()方法显式的唤醒，否则就永远沉默。保证每一个wait()方法调用都有一个相对应的notify()调用，当然也可以调用notifyAll()方法以防万一。

生产者与消费者问题是典型的同步问题。这里简单介绍两种不同的实现方法。

1，  条件变量

1. **import** threading
3. **import** time
5. **class** Producer(threading.Thread):
7. **def** \_\_init\_\_(self, t\_name):
9. threading.Thread.\_\_init\_\_(self, name=t\_name)


13. **def** run(self):
15. **global** x
17. con.acquire()
19. **if** x > 0:
21. con.wait()
23. **else**:
25. **for** i **in** range(5):
27. x=x+1
29. **print** "producing..." + str(x)
31. con.notify()
33. **print** x
35. con.release()


39. **class** Consumer(threading.Thread):
41. **def** \_\_init\_\_(self, t\_name):
43. threading.Thread.\_\_init\_\_(self, name=t\_name)
45. **def** run(self):
47. **global** x
49. con.acquire()
51. **if** x == 0:
53. **print** 'consumer wait1'
55. con.wait()
57. **else**:
59. **for** i **in** range(5):
61. x=x-1
63. **print** "consuming..." + str(x)
65. con.notify()
67. **print** x
69. con.release()


73. con = threading.Condition()
75. x=0
77. **print** 'start consumer'
79. c=Consumer('consumer')
81. **print** 'start producer'
83. p=Producer('producer')


87. p.start()
89. c.start()
91. p.join()
93. c.join()
95. **print** x

    上面的例子中，在初始状态下，Consumer处于wait状态，Producer连续生产（对x执行增1操作）5次后，notify正在等待的Consumer。Consumer被唤醒开始消费（对x执行减1操作）

2，  同步队列

Python中的Queue对象也提供了对线程同步的支持。使用Queue对象可以实现多个生产者和多个消费者形成的FIFO的队列。

生产者将数据依次存入队列，消费者依次从队列中取出数据。

1. # producer\_consumer\_queue
3. **from** Queue **import** Queue
5. **import** random
7. **import** threading
9. **import** time


13. #Producer thread
15. **class** Producer(threading.Thread):
17. **def** \_\_init\_\_(self, t\_name, queue):
19. threading.Thread.\_\_init\_\_(self, name=t\_name)
21. self.data=queue
23. **def** run(self):
25. **for** i **in** range(5):
27. **print** "%s: %s is producing %d to the queue!\n" %(time.ctime(), self.getName(), i)
29. self.data.put(i)
31. time.sleep(random.randrange(10)/5)
33. **print** "%s: %s finished!" %(time.ctime(), self.getName())


37. #Consumer thread
39. **class** Consumer(threading.Thread):
41. **def** \_\_init\_\_(self, t\_name, queue):
43. threading.Thread.\_\_init\_\_(self, name=t\_name)
45. self.data=queue
47. **def** run(self):
49. **for** i **in** range(5):
51. val = self.data.get()
53. **print** "%s: %s is consuming. %d in the queue is consumed!\n" %(time.ctime(), self.getName(), val)
55. time.sleep(random.randrange(10))
57. **print** "%s: %s finished!" %(time.ctime(), self.getName())


61. #Main thread
63. **def** main():
65. queue = Queue()
67. producer = Producer('Pro.', queue)
69. consumer = Consumer('Con.', queue)
71. producer.start()
73. consumer.start()
75. producer.join()
77. consumer.join()
79. **print** 'All threads terminate!'


83. **if** \_\_name\_\_ == '\_\_main\_\_':
85. main()

在上面的例子中，Producer在随机的时间内生产一个“产品”，放入队列中。Consumer发现队列中有了“产品”，就去消费它。本例中，由于Producer生产的速度快于Consumer消费的速度，所以往往Producer生产好几个“产品”后，Consumer才消费一个产品。

Queue模块实现了一个支持多producer和多consumer的FIFO队列。当共享信息需要安全的在多线程之间交换时，Queue非常有用。Queue的默认长度是无限的，但是可以设置其构造函数的maxsize参数来设定其长度。Queue的put方法在队尾插入，该方法的原型是：

put( item[, block[, timeout]])

如果可选参数block为true并且timeout为None（缺省值），线程被block，直到队列空出一个数据单元。如果timeout大于0，在timeout的时间内，仍然没有可用的数据单元，Full exception被抛出。反之，如果block参数为false（忽略timeout参数），item被立即加入到空闲数据单元中，如果没有空闲数据单元，Full exception被抛出。

Queue的get方法是从队首取数据，其参数和put方法一样。如果block参数为true且timeout为None（缺省值），线程被block，直到队列中有数据。如果timeout大于0，在timeout时间内，仍然没有可取数据，Empty exception被抛出。反之，如果block参数为false（忽略timeout参数），队列中的数据被立即取出。如果此时没有可取数据，Empty exception也会被抛出。