[python 线程的同步和互斥](http://blog.csdn.net/xluren/article/details/40978971)

标签： [线程](http://www.csdn.net/tag/%e7%ba%bf%e7%a8%8b)

2014-11-10 13:33 2295人阅读 [评论](http://blog.csdn.net/xluren/article/details/40978971#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/xluren/article/details/40978971#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

python 周边（45） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg 个人总结（66） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录[(?)[+]](http://blog.csdn.net/xluren/article/details/40978971)

文章借用了别人的总结，但是代码demo都是我自己想的。

参考链接：http://www.cnblogs.com/huxi/archive/2010/06/26/1765808.html

http://blog.csdn.net/iamaiearner/article/details/9363837

http://www.tuicool.com/articles/zAJjYj

http://doudouclever.blog.163.com/blog/static/1751123102012111192621448/

http://c4fun.cn/blog/2014/05/06/[**python**](http://lib.csdn.net/base/11)-threading/

先从一个简单的demo开始说起：

**[python]** [view plain](http://blog.csdn.net/xluren/article/details/40978971) [copy](http://blog.csdn.net/xluren/article/details/40978971)

[在CODE上查看代码片](https://code.csdn.net/snippets/513991)

1. (pythonenv)[xluren@test thread\_communicate]$ cat   demo\_thread.py
2. #!/usr/bin/python
3. **import** time
4. **def** add\_sum(n):
5. **global** sum,mutex
6. **for** i **in** range(0,n):
7. #        mutex.acquire()
8. sum+=int(i)
9. #print threading.currentThread().getName()+"#"+str(sum)+"#"+str(time.time())
10. #        mutex.release()

13. **import** threading
14. threads=[]
15. **global** sum,mutex
16. mutex=threading.Lock()

19. sum=0
20. **for** i **in** range(100):
21. threads.append(threading.Thread(target=add\_sum,args=(10000,)))
22. **for** i **in** threads:
23. i.start()

26. **for** i **in** threads:
27. i.join()
28. **print** sum

运行结果：

**[python]** [view plain](http://blog.csdn.net/xluren/article/details/40978971) [copy](http://blog.csdn.net/xluren/article/details/40978971)

[在CODE上查看代码片](https://code.csdn.net/snippets/513991)

1. (pythonenv)[xluren@test thread\_communicate]$ python demo\_thread.py
2. 4070480886
3. (pythonenv)[xluren@test thread\_communicate]$ python demo\_thread.py
4. 3492920367

两次运行的结果不一样。

打开这句的注释你会发现 #print threading.currentThread().getName()+"#"+str(sum)+"#"+str(time.time())

**[python]** [view plain](http://blog.csdn.net/xluren/article/details/40978971) [copy](http://blog.csdn.net/xluren/article/details/40978971)

[在CODE上查看代码片](https://code.csdn.net/snippets/513991)

1. (pythonenv)[xluren@test thread\_communicate]$ python demo\_thread.py |awk -F'#' '{if(NF>5) print}'
2. Thread-3#498629#1415598231.61Thread-2#377314#1415598231.6Thread-5#252126#1415598231.61
3. Thread-4#516899#1415598231.61Thread-6#377314#1415598231.61Thread-1#462700#1415598231.62
4. Thread-4#752637#1415598231.69Thread-7#589330#1415598231.68Thread-2#1227700#1415598231.7
5. Thread-2#1331434#1415598231.7Thread-5#1227700#1415598231.7Thread-1#1175952#1415598231.69

原因：也就是说一个线程在计算的过程中用尽了自己的时间片，然后让出了执行的权利，并且此时存在了多个线程同时运行的情况。

也就是说两个线程会同时的去使用同一个数据，而如果是这种情况

线程一：计算了结果，放到内存中，sum在放到内存中前是1000，而线程二，去内存取数据，但是进程一还没有放进去的时候，他取到的是sum=999.在sum=999的基础上线程二去进行计算，必然导致计算的结果是错误的。可以参看UNIX高级环境编程的298页，有详细的介绍，这里涉及到了线程之间的同步和互斥。

线程间的同步和互斥有以下几种情况

**1.mutex 互斥锁,**

通常用来保护多个线程的共享数据的：

代码：

**[python]** [view plain](http://blog.csdn.net/xluren/article/details/40978971) [copy](http://blog.csdn.net/xluren/article/details/40978971)

[在CODE上查看代码片](https://code.csdn.net/snippets/513991)

1. (pythonenv)[xluren@test thread\_communicate]$ cat demo\_thread\_mutex.py
2. #!/usr/bin/python
3. **import** time
4. **def** add\_sum(n):
5. **global** sum,mutex
6. **for** i **in** range(0,n):
7. mutex.acquire()
8. sum+=int(i)
9. **print** threading.currentThread().getName()+"#"+str(sum)+"#"+str(time.time())
10. mutex.release()
12. **import** threading
13. threads=[]
14. **global** sum,mutex
15. mutex=threading.Lock()
17. sum=0
18. **for** i **in** range(100):
19. threads.append(threading.Thread(target=add\_sum,args=(10000,)))
20. **for** i **in** threads:
21. i.start()
23. **for** i **in** threads:
24. i.join()
25. **print** sum

运行结果：

**[python]** [view plain](http://blog.csdn.net/xluren/article/details/40978971) [copy](http://blog.csdn.net/xluren/article/details/40978971)

[在CODE上查看代码片](https://code.csdn.net/snippets/513991)

1. (pythonenv)[xluren@test thread\_communicate]$ python demo\_thread\_mutex.py |awk -F'#' '{if(NF>5) print}'
2. (pythonenv)[xluren@test thread\_communicate]$ python demo\_thread\_mutex.py |awk -F'#' '{print NF}'|sort |uniq -c
3. 1 1
4. 1000000 3

**2.Condition**

Condition（条件变量）通常与一个锁关联。需要在多个Contidion中共享一个锁时，可以传递一个Lock/RLock实例给构造方法，否则它将自己生成一个RLock实例。

可以认为，除了Lock带有的锁定池外，Condition还包含一个等待池，池中的线程处于状态图中的等待阻塞状态，直到另一个线程调用notify()/notifyAll()通知；得到通知后线程进入锁定池等待锁定。

实例方法：

例子是很常见的生产者/消费者模式：

**[python]** [view plain](http://blog.csdn.net/xluren/article/details/40978971) [copy](http://blog.csdn.net/xluren/article/details/40978971)

[在CODE上查看代码片](https://code.csdn.net/snippets/513991)

1. #!/usr/bin/python
2. # encoding: UTF-8
3. **import** threading
4. **import** time
5. **def** produce():
6. **global** product
7. **if** con.acquire():
8. **while** True:
9. **if** product **is** None:
10. **print** 'produce...'
11. product = 'anything'
12. con.notify()
13. con.wait()
14. time.sleep(2)
15. **def** consume():
16. **global** product
17. **if** con.acquire():
18. **while** True:
19. **if** product **is** **not** None:
20. **print** 'consume...'
21. product = None
22. con.notify()
23. con.wait()
24. time.sleep(2)
25. product = None
26. con = threading.Condition()
27. t1 = threading.Thread(target=produce)
28. t2 = threading.Thread(target=consume)
29. t1.start()
30. t2.start()

**3.semaphore信号量**

Semaphore（信号量）是计算机科学史上最古老的同步指令之一。Semaphore管理一个内置的计数器，每当调用acquire()时-1，调用release() 时+1。计数器不能小于0；当计数器为0时，acquire()将阻塞线程至同步锁定状态，直到其他线程调用release()。

实例方法：

**[python]** [view plain](http://blog.csdn.net/xluren/article/details/40978971) [copy](http://blog.csdn.net/xluren/article/details/40978971)

[在CODE上查看代码片](https://code.csdn.net/snippets/513991)

1. [xluren@test thread\_communicate]$ cat demo\_thread\_semaphore.py
2. #!/usr/bin/python
3. **import** time
4. **def** add\_sum(n):
5. **global** sum,sem
6. **for** i **in** range(0,n):
7. **if** sem.acquire():
8. sum+=int(i)
9. sem.release()
10. **import** threading
11. threads=[]
12. **global** sum,sem
13. sem=threading.Semaphore(1)
15. sum=0
16. **for** i **in** range(100):
17. threads.append(threading.Thread(target=add\_sum,args=(10000,)))
18. **for** i **in** threads:
19. i.start()
21. **for** i **in** threads:
22. i.join()
23. **print** sum

测试结果：

**[python]** [view plain](http://blog.csdn.net/xluren/article/details/40978971) [copy](http://blog.csdn.net/xluren/article/details/40978971)

[在CODE上查看代码片](https://code.csdn.net/snippets/513991)

1. [xluren@test thread\_communicate]$ python  demo\_thread\_semaphore.py
2. 4999500000
3. [xluren@test thread\_communicate]$

原因分析：因为sem设置的是1，所以同一时刻只有一个线程访问同一变量。这就可以防止内存的数据读写不一致了。

**4.Event事件**

Event（事件）是最简单的线程通信机制之一：一个线程通知事件，其他线程等待事件，Event其实就是一个简化版的 Condition。Event没有锁，无法使线程进入同步阻塞状态，源代码中调用了norifyall

实例代码:

**[python]** [view plain](http://blog.csdn.net/xluren/article/details/40978971) [copy](http://blog.csdn.net/xluren/article/details/40978971)

[在CODE上查看代码片](https://code.csdn.net/snippets/513991)

1. [xluren@test thread\_communicate]$ cat  demo\_event.py
2. **import** threading
3. **import** random
4. **import** time
5. **def** deal\_event(thread\_event):
6. **print** threading.currentThread().getName(),"waiting......."
7. thread\_event.wait()
8. **print** threading.currentThread().getName(),"done.........."
10. thread\_event = threading.Event()
11. **for** i **in** range(100):
12. t = threading.Thread(target=deal\_event,args=(thread\_event,))
13. t.start()
15. thread\_event.set()

**5.Queue队列**

Queue模块中提供了同步的、线程安全的队列类，包括FIFO（先入先出)队列Queue，LIFO（后入先出）队列LifoQueue，和优先级队列PriorityQueue。这些队列都实现了锁原语，能够在多线程中直接使用。可以使用队列来实现线程间的同步。内部实现使用了mutex和condition

示例代码：

**[python]** [view plain](http://blog.csdn.net/xluren/article/details/40978971) [copy](http://blog.csdn.net/xluren/article/details/40978971)

[在CODE上查看代码片](https://code.csdn.net/snippets/513991)

1. [xluren@test thread\_communicate]$ cat demo\_thread\_queue.py
2. #!/usr/bin/python
3. **import** time
4. **def** produce\_queue(number\_queue):
5. **for** i **in** range(100):
6. **print** threading.currentThread().getName(),"produce ",i
7. number\_queue.put(i)
8. time.sleep(1)
9. **def** consume\_queue(number\_queue):
10. **while** 1:
11. n=number\_queue.get()
12. **if** n>90:
13. **break**
14. **else**:
15. **print**  threading.currentThread().getName(),"consume",n
16. **import** threading,Queue
17. threads=[]
18. number\_queue=Queue.Queue()
19. threads.append(threading.Thread(target=produce\_queue,args=(number\_queue,)))
20. threads.append(threading.Thread(target=consume\_queue,args=(number\_queue,)))
21. **for** i **in** threads:
22. i.start()
23. **for** i **in** threads:
24. i.join()

Thread类的构造函数定义如下

class threading.Thread(group=None, target=None, name=None, args=(), kwargs={})

group： 留作ThreadGroup扩展使用，一般没什么用

target：新线程的任务函数名

name： 线程名，一般也没什么用

args： tuple参数

kwargs：dictionary参数

Thread类的成员变量和函数如下

start() 启动一个线程

run() 线程执行体，也是一般要重写的内容

join([timeout]) 等待线程结束

name 线程名

ident 线程ID

daemon 是否守护线程

isAlive()、is\_alive() 线程是否存活

getName()、setName() Name的get&set方法

isDaemon()、setDaemon() daemon的get&set方法

互斥锁

threading中定义了两种锁：threading.Lock和threading.RLock。两者的不同在于后者是可重入锁，也就是说在一个线程内重复LOCK同一个锁不会发生死锁，这与POSIX中的PTHREAD\_MUTEX\_RECURSIVE也就是可递归锁的概念是相同的。

关于互斥锁的API很简单，只有三个函数————分配锁，上锁，解锁。

threading.Lock() 分配一个互斥锁

acquire([blocking=1]) 上锁(阻塞或者非阻塞，非阻塞时相当于try\_lock，通过返回False表示已经被其它线程锁住。)

release() 解锁

条件变量

条件变量总是与互斥锁一起使用的，threading中的条件变量默认绑定了一个RLock，也可以在初始化条件变量的时候传进去一个自己定义的锁。可用的函数如下

threading.Condition([lock]) 分配一个条件变量

acquire(\*args) 条件变量上锁

release() 条件变量解锁

wait([timeout]) 等待唤醒，timeout表示超时

notify(n=1) 唤醒最大n个等待的线程

notifyAll()、notify\_all() 唤醒所有等待的线程