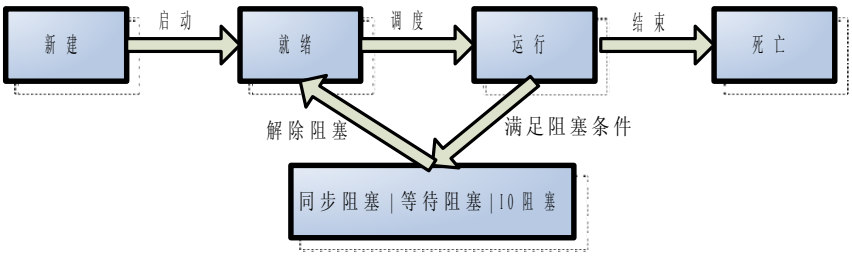
Python3 的 threading 学习笔记

作者：罗寅兵　　日期：2012-11-14

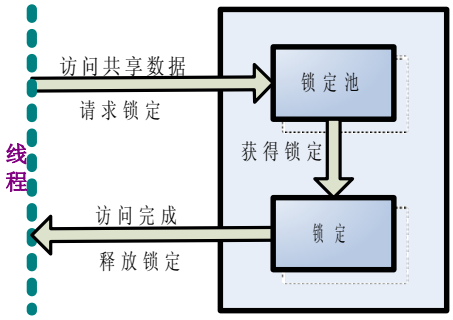
一、线程基础

1、线程状态



2、线程互斥锁同步控制

多个线程需要修改同一共享数据时，需要进行同步控制，引入了锁的概念。

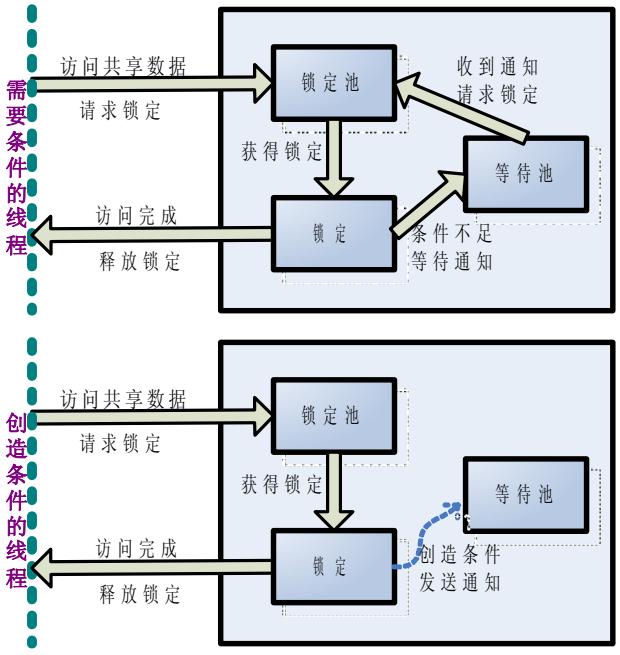


a、同一时间可能有多个线程在锁定池中，它们处于同步阻塞状态竞争锁定；

b、同一时间只能有一个线程获得锁定处于运行状态。

3、条件变量（线程通信）

有的线程需要预备条件才能干活。



二、threading：线程创建、启动、睡眠、退出

1、方法一：将要执行的函数作为参数传递给 threading.Thread()

```
#!/usr/bin/env python3
# -*- coding:utf-8 -*-
import threading, time

def func(n):
    global count
    time.sleep(0.1)
    for i in range(n):
        count += 1

if __name__=’__main__’:
    count = 0
    threads = []
    for i in range(5):
        threads.append(threading.Thread(target=func, args=(1000,)))
    for t in threads:
        t.start()
    time.sleep(5)
    print(‘count:’,count)
```

以上例子创建了 5 个线程去执行 func 函数。**获得的结果可能是 5000，但也有时候会出现错误，解决方法请继续阅读下文。**

要点：

a. threading.Thread(target=func, args=(10,))：func 为函数名，args 为函数参数（必须以元组的形式传递）；
b. t.start(): 启动函数，等待操作系统调度；
c. 函数运行结束，线程也就结束；
d. time.sleep(): 线程进入睡眠，处于 IO 阻塞状态。

1、方法二：继承 threading.Thread()类，并重写 run()【推荐使用】

```
#!/usr/bin/env python3
# -*- coding:utf-8 -*-
import threading, time

class myThread(threading.Thread):
    def __init__(self, n):
        threading.Thread.__init__(self)
        self.myThread_n = n
    def run(self):
        global count
        for i in range(self.myThread_n):
            count += 1

if __name__=’__main__’:
    count = 0
    threads = []
    for i in range(5):
        threads.append(myThread(1000))
    for t in threads:
        t.start()
    time.sleep(5)
    print(‘count:’,count)
```

要点：

a. threading.Thread.__init__(self)：回调父类方法。如果你重写了__init__()方法，这一步是必须的，否则出错。

三、threading：线程同步锁互斥控制

因为线程是乱序执行的，在某种情况下上面的两个例子，输出的结果可能不是预期的值。

我将第 2 例的线程类修改下，让问题更加突出，然后你每次运行的时候再把线程数也修改下，并计算出预期结果和运行结果对比。一定要多试几次哦。

```
class myThread(threading.Thread):
    def __init__(self, n):
        threading.Thread.__init__(self)
        self.myThread_n = n
    def run(self):
        global count
        for i in range(self.myThread_n):
            __Temp = count
            time.sleep(0.0001)
            count = __Temp + 1
```

是不是输出的结果和预期结果不一致啊，呵呵！因为多个线程都在同时操作同一个共享资源，所以造成资源破坏。不过我们可以通过同步锁来解决这种问题：

```
#!/usr/bin/env python3
# -*- coding:utf-8 -*-
import threading, time

class myThread(threading.Thread):
    def __init__(self, n):
        threading.Thread.__init__(self)
        self.myThread_n = n
    def run(self):
        global count
        for i in range(self.myThread_n):
            if lock.acquire():
                __Temp = count
                time.sleep(0.0001)
                count = __Temp + 1
                lock.release()

if __name__==’__main__’:
    count = 0
    lock = threading.Lock()      #同步锁，也称互斥量

    threads = []
    for i in range(5):
        threads.append(myThread(1000))
    for t in threads:
        t.start()
    time.sleep(5)
    print(‘count:’,count)
```

要点：

a. lock = threading.Lock()：创建锁;

b. lock.acquire([timeount])：请求锁定，如果设定了 timeout，则在超时后通过返回值可以判断是否得到了锁，

从而可以进行一些其他的处理;

c. lock.release()：释放锁定。

四、threading：线程死锁和递归锁

1、死锁

a. 在线程间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，因为系统判断这部分资源都正在使用，所有这两个线程在无外力作用下将一直等待下去。下面是一个死锁的例子：

```
#!/usr/bin/env python3
# -*- coding:utf-8 -*-
import threading, time

class myThread(threading.Thread):
    def doA(self):
        if lockA.acquire():
            print(self.name,"got lockA.")
            time.sleep(0.0001)
            if lockB.acquire():
                print(self.name,"got lockB")
                lockB.release()
            lockA.release()
    def doB(self):
        if lockB.acquire():
            print(self.name,"got lockB.")
            time.sleep(0.0001)
            if lockA.acquire():
                print(self.name,"got lockA")
                lockA.release()
            lockB.release()
    def run(self):
        self.doA()
        self.doB()
if __name__=='__main__':
    lockA = threading.Lock()
    lockB = threading.Lock()

    threads = []
    for i in range(5):
        threads.append(myThread())
    for t in threads:
        t.start()
    for t in threads:
        t.join()    #等待线程结束，后面再讲。
```

b. 当一个线程已经获得了锁，再此请求锁也会出现死锁，请看下面的例子：

```
#!/usr/bin/env python3
# -*- coding:utf-8 -*-
import threading, time
import random

class myThread(threading.Thread):
    def toHex(self):
        global L
        if lock.acquire(1):
            for i in range(len(L)):
                if type(L[i]) is int: L[i]="{:X}".format(L[i])
            lock.release()
        else:
            print(self.name,"错误，系统忙")
    def run(self):
        global L
        if lock.acquire(1):
            L.append(random.randint(0, 15))
            self.toHex()
            lock.release()
        else:
            print(self.name,"错误，系统忙")
if __name__=='__main__':
    L = []
    lock = threading.Lock()

    threads = []
    for i in range(5):
        threads.append(myThread())
    for t in threads:
        t.start()
    for t in threads:
        t.join()   #等待线程结束，后面再讲。
```

2、递归锁（也称可重入锁）

上一例子的死锁，我们可以用递归锁解决：

```
#lock = threading.Lock()  注释掉此行，并加入下行。
lock = threading.RLock()
```

为了支持在同一线程中多次请求同一资源，python 提供了“可重入锁”：threading.RLock。RLock 内部维护着一个 Lock 和一个 counter 变量，counter 记录了 acquire 的次数，从而使得资源可以被多次 require。直到一个线程所有的 acquire 都被 release，其他的线程才能获得资源。

递归锁一般应用于复杂的逻辑。

五、threading：条件变量同步

有一类线程需要满足条件之后才能够继续执行，Python 提供了 threading.Condition 对象用于条件变量线程的支持，它除了能提供 RLock()或 Lock()的方法外，还提供了 wait()、notify()、notifyAll()方法。

lock_con = threading.Condition([Lock/Rlock])：锁是可选选项，不传人锁，对象自动创建一个 RLock()。

wait()：条件不满足时调用，线程会释放锁并进入等待阻塞；

notify()：条件创造后调用，通知等待池激活一个线程；

notifyAll()：条件创造后调用，通知等待池激活所有线程。

```
#!/usr/bin/env python3
# -*- coding:utf-8 -*-
import threading, time
from random import randint

class Producer(threading.Thread):
    def run(self):
        global L
        while True:
            val = randint(0,100)
            print(self.name, ":Append "+str(val), L)
            if lock_con.acquire():
                l.append(val)
                lock_con.notify()
                lock_con.release()
            time.sleep(3)
class Consumer(threading.Thread):
    global L
    while True:
        if lock_con.acquire():
            if len(L)==0:
                lock_con.wait()
            else:
                print(self.name, ":DELETE "+L[0],L)
                del L[0]
                lock_con.release()
        time.sleep(0.25)
if __name__=='__main__':
    L = []
    lock_con = threading.Condition()

    threads = []
    for i in range(5):
        threads.append(Producer())
    threads.append(Consumer())
    for t in threads:
        t.start()
    for t in threads:
        t.join()
```

六、threading：条件同步

条件同步和条件变量同步差不多意思，只是少了锁功能，因为条件同步设计于不访问共享资源的条件环境。

event = threading.Event()：条件环境对象，初始值为 False；

event.isSet()：返回 event 的状态值；

event.wait()：如果 event.isSet()==False 将阻塞线程；

event.set()：设置 event 的状态值为 True，所有阻塞池的线程激活进入就绪状态，等待操作系统调度；

event.clear()：恢复 event 的状态值为 False。

实例：

```
#!/usr/bin/env python3
# -*- coding:utf-8 -*-
import threading, time

class Boss(threading.Thread):
    def run(self):
        print("BOSS：今晚大家都要加班到 22:00。")
        event.isSet() or event.set()
        time.sleep(5)
        print("BOSS：<22:00>可以下班了。")
        event.isSet() or event.set()
class Worker(threading.Thread):
    def run(self):
        event.wait()
        print("Worker：哎……命苦啊！")
        time.sleep(0.25)
        event.clear()
        event.wait()
        print("Worker：Oh Yeah !")
if __name__=='__main__':
    event = threading.Event()

    threads = []
    for i in range(5):
        threads.append(Worker())
    threads.append(Boss())
    for t in threads:
        t.start()
    for t in threads:
        t.join()
```

七、threading：线程合并与后台线程

1、线程合并

join()方法，使得一个线程可以等待另一个线程执行结束后再继续运行。这个方法还可以设定一个 timeout 参数，避免无休止的等待。因为两个线程顺序完成，看起来象一个线程，所以称为线程的合并。【前面有演示，就不多写了】

2、后台线程

默认情况下，主线程在退出时会等待所有子线程的结束。如果希望主线程不等待子线程，而是在退出时自动结束所有的子线程，就需要设置子线程为后台线程：setDaemon(True)　**(在 Windows 下的 Py3.3 调试失败，Linux 下面 OK)**

```
import threading, time
class myThread(threading.Thread):
    def run(self):
        print(self.name, "is Start.")
        time.sleep(5)
        print(self.name, "is Finished..")
if __name__=='__main__':
        t = myThread()
        t.setDaemon(True)
        t.start()
        print("main thread is finished.")
```

八、threading：信号量

信号量用来控制线程并发数的，BoundedSemaphore 或 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()。

BoundedSemaphore 与 Semaphore 的唯一区别在于前者将在调用 release()时检查计数器的值是否超过了计数器的初始值，如果超过了将抛出一个异常。

```
#!/usr/bin/env python3
# -*- coding:utf-8 -*-
import threading, time

class myThread(threading.Thread):
    def run(self):
        if semaphore.acquire():
            print(self.name)
            time.sleep(5)
            semaphore.release()
if __name__==’__main__’:
    semaphore = threading.Semaphore(5)
    thrs = []
    for i in range(100):
        thrs.append(myThread())
    for t in thrs:
        t.start()
```

九、threading：定时器

timer()是 Thread 的派生类，用于在指定时间后调用一个方法。【我不知道用继承的方法写】

```
#!/usr/bin/env python3
# -*- coding:utf-8 -*-
import threading

def func():
    print(“Hello World.”)
t = threading.Timer(5, func)
t.start()
```

十、threading：线程局部数据

local()类用于存储线程局部数据，即线程和其他线程之间直接无法共享，互相不影响使用。

```
import threading

class myThread(threading.Thread):
    def run(self):
        local.name = “Obama”
        print(local.name)
if __name__==”__main__”:
    local = threading.local()
    local.name = “Lady Gaga”
    myThread().start()
    print(local.name)
```

十一、threading：其他函数

threading.isAlive()	判断线程是否还在运行中
threading.isDaemon()	返回线程的 daemon 标志
threading.getName()	返回线程名
threading.current_thread()	返回当前线程句柄
threading.enumerate()	返回正在运行的线程列表
threading.activeCount()	返回正在运行的线程数量

十二、多线程利器：列队同步（queue）

python 有 queue 模块方便线程的生产者与使用者信息列队的安全交换。

queue 定义了 3 种信息列队模式类：

a. Queue([maxsize]):FIFO 列队模式，[maxsize]定义列队容量，缺省为 0，即无穷大；
b. LifoQueue([maxsize]):LIFO 列队模式；
c. PriorityQueue([maxsize]):优先级列队模式，使用此列队时，项目应该是(priority,data)的形式。

queue 列队类的方法：

a. q.put(item[,block,[timeout]]):将 item 放入列队，如果 block 设置为 True[默认]时,列队满了，调用者将被阻塞，否则抛出 Full 异常。timeout 设置阻塞超时；
b. q.get([block,[timeout]]):从列队中取出一个项目；
c. q.qsize():返回列队大小；
d. q.full():如果列队满了返回 True，否则为 False；
e. q.empty():如果列队为空返回 True，否则为 False；
f. q.queue.*:列队项目操作方法集合<和列表方法很类似>；

queue 列队实例：

```
#!/usr/bin/env python3
# -*- coding:utf-8 -*-
import threading
from time import sleep
from random import randint

class Production(threading.Thread):
    def run(self):
        while True:
            q.put(randint(0, 100))
            Sleep(3)
class Proces(threading.Thread):
    def run(self):
        while True:
            re = q.get()
            if re % 2 == 0: print(re)
if __name__=’__main__’:
    q = queue.Queue(10)
    threads = [Production(), Proces()]
    for t in threads:
        t.start()
```