Python 多线与锁

Edit by Void 2015/08/30

一.python 的线程

python 的线程分两个模块，thread、threading，本文 python 版本为 Python 2.7.9。

---

1. thread

在 python 3.x 已经改名为_thread，作为开发者，建议用更高级的模块 threading。

核心模块 thread 是多线程的最基本模块，提供了低级别的、原始的线程，同时也有互斥锁（mutexes）和二态信号量（binary semaphores），threading 模块是基于此模块封装的更高级模块。

这个模块支持 windows，Linux，SGI IRIX，Solaries 2.x，等有 POSIX 线程的系统。

其定义的接口如下：

接口名称	接口作用
thread.ERROR	异常，抛出一个线程的错误
thread.LockType	线程同步锁的类型
thread.start_new_thread(function, args[, kwargs])	开始一个新的线程，并返回该线程的标识符（线程号），这个线程执行函数 function，其中参数为 args（tuple 类型），可选参数 kwargs 是指定字典类型参数。 当函数 function 返回时，这个线程还是后台存在。 当函数 function 出现一个未捕获的异常时，会打印堆栈跟踪错误，然后线程终止，但不影响其他线程。
thread.interrupt_main()	在主线程抛出一个 keyboardinterrupt，子线程可以用这个函数去中断主线程。
thread.exit()	抛出一个 Systemexit 异常，如果没有补获，线程静态终止。
thread.allocate_lock()	返回一个新的锁对象，这个所默认为 unlocked。

thread.get_ident()	返回当前线程的一个线程标识符，其值为无直接含义的正整数，用来在线程表数据字典来查询，线程标识符会在线程终止时，其他线程创建时回收。
thread.stack_size([size])	返回创建线程时线程栈的大小，可选项 size 可以指定创建的线程的栈的大小,其值必须[0,32768]，但不支持线程栈大小的修改。
lock.release()	释放锁，锁必须在释放之前获取。
lock.locked()	返回锁的状态，True 表示该所被线程获取，False 表示未线程获取。锁可以与环境管理器 with 一起 拿了 nianzhog 使用。 例如： import thread a_lock = thread.allocate_lock() with a_lock: print "a_lock is locked while this executes"

附加说明：

(1). 通常线程不要与中断一起使用，例如 KeyboardInterrupt 异常会被随机的一个线程获取，但是信号模块可以使用，信号中断会被主线程获取。

(2). 调用 sys.exit()或者抛出 SyetemExit 异常等效于调用了 Thread.exit()。

(3). 不能中断一个锁的 acquire()，当锁被获取后 KeyboardInterrupt 才会执行。

(4). 当主线程退出时，它能决定其他子线程是否继续存在，在大多数系统上，子线程会被杀死。

(5). 当主线程退出时，不做任何清理动作，甚至标准 IO 也不刷新。

(6). thread 中的锁，用 thread 模块来实现生产者，消费者问题，Lock 对象时 Python 提供的低级别的线程控制工具，使用起来只需下列三部：

>thread.allocate_lock()，返回一个新的 Lock 对象，即为一个新的锁；

>lock.acquire()，相当于 P 操作，得到一个锁。

>lock.release()，相当于 V 操作，释放一个锁。

例子：

#生产者与消费者

```
# -*- coding: cp936 -*-
import thread
import time
import random

dish = 0
```

```
lock = thread.allocate_lock()

def producerFunction():
      '''
      如果投的筛子比 0.1 大，则向盘子中增加一个苹果
      '''
      global lock, dish
      while True:
            if random.random() > 0.1:
                  lock.acquire()
                  if dish < 100:
                        dish += 1
                        print '生产者增加了一个苹果，现在有%d 个苹果
' % dish
                  lock.release()
                  time.sleep(random.random()*3)

def consumerFunction():
      '''
      如果投的筛子比 0.9 小，则向盘子中拿走一个苹果
      '''
      global lock, dish
      while True:
            if random.random() > 0.9:
                  lock.acquire()
                  if dish > 0:
                        dish -= 1
                        print '消费者拿走一个苹果，现在有%d 个苹果' %
dish
                  lock.release()
                  time.sleep(random.random()*3)

def begin():
      ident1 = thread.start_new_thread(producerFunction, ())
      ident2 = thread.start_new_thread(consumerFunction, ())

if __name__ == '__main__':
    begin()
```

---

2. threading

源码位置：Lib/threading.py

threading 模块是基于 thread 的高级别的线程模块，dummy_threading 是不能使用 threading 的替代模块。

该模块是基于 java 的 threading 模块松散设计的。但是在 java 中，锁和条件变量是每个对象基本的操作，在 python 中，它们是分开的。Python 的 Thread 类支持 Java 线程类的子集操作。到目前，还没有优先级，线程组，线程被销毁，停止，挂起，继续，中断等操作。当这些 Java 的 Thread 类静态方法实现之后，会被映射到模块级别的函数中。

其定义的接口如下：

接口名称	接口作用
threading.active_count() threading.activeCount()	返回当前 alive 的线程对象数量，返回的数量等于 enumarate()返回列表的长度。
threading.Condition()	工厂函数，返回一个条件变量对象，该条件变量对象允许一个或多个线程阻塞，直到收到其他线程的信号。
threading.current_thread() threading.currentThread()	响应调用者线程，返回当前线程的对象。如果调用者线程不是通过 threading 模块创建的，则返回一个有限方法的 dummy 线程对象。
threading.enumarate()	返回一个当前 alive 线程的列表，这个列表包含后台线程，current_thread()创建的虚拟线程和主线程，也包含终止的线程和还未启动的线程。
thrading.Event()	一个工厂函数，返回一个事件对象。事件对象有一个标志位，可以被 set()方法设置为 true，或者被 clear()方法设置为 false，wait()方法则阻塞直到标志位为 true。
class threading.local	一个类代表线程的本地数据。去管理线程本地数据，只需要创建一个 local 对象（子对象）的实例，通过操作方法或者属性来完成。

	mydata = threading.local() mydate.x = 1
threading.Lock()	一个工厂函数，返回一个原始的锁对象，一旦线程获取该锁，随后就获取锁的块，直到锁被释放。任何线程可以释放它。
threading.RLock()	一个工厂函数，返回一个折返锁对象，折返锁必须被获取该锁的线程释放，不能被其他线程释放。一旦一个线程获取一个折返锁，这个线程可以非阻塞重新获取该锁，该线程每次获取该锁就必须释放该锁。
threading.Semaphore([value])	一个工厂函数，返回一个新的信号对象，一个信号管理一个计数，这个计数代表调用 release()的个数减去调用 acquire()的个数，再加上一个初始值。acquire()方法会被阻塞，直到方法返回，并且这个计数器值非负。如果计数器值没被设定，则默认为 1。
threading.BoundedSemaphore([value])	一个工厂函数，返回一个绑定的信号对象。一个绑定的信号会检测其当前的值不超过初始化的值，如果超过，抛出 ValueError。在大多数情况下，信号被用来管理监测有限的资源。如果信号被释放太多次数，则是一个 bug。如果信号的值未被给出，默认为 1。
class theading.Thread	一个线程类，该类可以被有限的安全继承。
class threading.Timer	一个线程计时器类。
threading.settrace(func)	设置一个所有从 threading 模块启动的线程的跟踪函数，这个函数 func 会在 run()方法调用前被传递给 sys.settrace()。
threading.setprofile(func)	设置一个所有从 threading 模块启动的线程的 profile 函数，这个函数 func 会在 run()方法调用前被传递给 sys.setprofile()。
threading.stack_size([size])	返回创建一个线程所需要栈的大小。可选项 size 指定后续创建线程的栈的大小。其值必须为自然数，在[0,32768]之间，即最大为 32KB，32KB 是保证解释器

	有效栈空间的最小大小。不支持改变线程栈的大小，强行改变会抛出 ThreadError 异常。如果指定的线程栈大小无效，抛出 ValueError，同时这个线程栈大小未被修改。
exception threading.ThreadError	抛出一系列的线程相关的错误。很多接口使用 RuntimeError 替代 ThreadError。

2.1 threading.Thread

threading.Thread 是管理一个线程活动的控制类。有两种方法：通过一个可调用的构造函数，或者在子类重载 run()方法。子类中不能重载（除了构造函数）其他方法，只能重载__init__，run()方法。

创建一个线程对象，其活动状态只能通过调用 start()方法来启动。这个方法在一个进程里面调用 run()。

线程开始之后，线程状态可以被认为是“alive”,alive 状态时，当 run()方法终止时，线程停止，或者抛出一个未处理的异常。is_alive()方法用来检测线程是否是 alive。

其他线程可以调用 join()方法，这个方法阻塞调用 join()的线程，直到 join()方法终止。

一个线程有一个名字，这个名字可以传递给构造函数，通过 name 变量来读取或者修改线程的名字。

一个线程可以被标志位守护线程，这个标志的含义就是整个 Python 程序退出时，守护线程还存在，这个初始值继承自创建线程的时候，这个标志位可以通过

daemon 来设置。守护线程会在关机时退出。其资源（比如打开的文件，数据库事务等）可能不会被释放。如果要线程优雅的退出，就让其为非守护线程或者使用适当的信令机制，比如事件。

主线程对象，对应于 python 程序初始控制线程，其不是守护线程。

虚拟线程，其可以被创建，这种线程对应于非 threading 模块创建的外来线程，比如 C 语言创建的线程。虚拟线程功能有限，被授予 alive、守护性的，不能调用 join()方法或行使 join()功能。其不会被删除，因为其不能检测到外来线程是否终止。

其定义的接口如下：

接口名称	接口作用
class threading.Thread(group=None, target=None, name=None, args=(), kwargs={})	这个构造函数被称为关键字参数，参数如下： group 应该为 None，保留字段作为以后 python 线程组实现以后的扩展。 target 是一个可调用对象，由 run()方法来调用，默认值为 None。 name 是线程的名称，默认情况下，一个线程的名称构造如“Thread-N”，N 是小十进制数。 args 是 target 调用对象的参数，为元组，默认为()。 kwargs 是 target 调用对象的关键字参数字典，默认为{}。 如果子类继承 threading.Thread，重载构造函数，必须保证在做其他事情之前调用构造函数（Thread.__init__()）。
start()	开始线程，使线程状态 alive。 它必须在每个线程对象调用。s 在一个单独的线程中，start()方法会调用 run()。

	在同一个线程中调用两次会抛出 RuntimeError。
run()	该方法代表线程的活动。 可以在子类重载该方法，run()方法调用可调用的对象，并以 target 参数传递给对象构造器，参数会顺序从 args、kwargs 里面获取。
join([timeout])	等待线程终止，它会阻塞调用线程，直到 join()方法调用终止。 当 timeout 参数非 None 时，其值为一个超时的浮点秒数，join()方法返回 None，需要调用 isAlive()方法去判断超时是否发生，如果为 alive，则 join()调用了 time out。 当 timeout 参数为 None 时，join()方法会阻塞直到线程终止。 一个线程可以调用 join()多次。 如果调用 join()造成死锁，则会抛出 RuntimeError()。在线程启动之前调用 join()同样会抛出 RuntimeError()。
name	一个字符串去标志线程，无语义，多线程可能给出相同的名字，初始化的名字被构造函数设置。
getName()	获取线程的名称。
setName()	设置线程的名称。
ident	线程的标志，如果线程没启动则为 None，这是个一个自然数，类似于 thread.get_ident()，线程标志器可以在线称退出和其他线程创建时再次使用。这个标志器可以再线程退出时获取。
is_alive() isAlive()	返回线程是否 alive，True/False 在线程执行 run()之前到 run()终止之后，都返回 True。模块函数 enumerate()返回活动的线程列表。
deamon	一个布尔变量（True/False），指明这个线程是否守护线程。这个必须在 start()调用之前设置，否则会抛 RuntimeError。其初始值继承自创建的线程，主线程不是守护线程的，那么主线程创建的其他线程默认为 False。
isDaemon()	判断是否守护进程。

setDaemon()	设置守护进程。

2.2 锁

锁名称	锁的描述
Lock objects	(1)一个基本的锁是不被一个特定的线程一直持有的同步机制。在 python 中，它是单签最低级的同步机制，直接由 thread 模块扩展实现。 (2)原始的锁只有两种状态，“locked”和“unlocked”。它创建的时候是“unlocked”状态，其有两个基本的方法，acquire()和 release()。 (3)当锁的状态为“unlocked”的时候，acquire()将锁改为“locked”，而且快速返回。 (4)当锁的状态为“locked”的时候，acquire()方法阻塞线程直到其他线程调用 release()，将锁状态改为“unlocked”，然后 acquire()调用重置锁的状态为“locked”，然后返回。 (5)release()方法只能在“locked”状态下调用，它将锁的状态改为“unlocked”，然后直接返回。 (6)如果试图去释放一个“unlocked”锁，会抛出 ThreadError 异常。 (7)当多个线程在 acquire()阻塞，等待状态由“locked”转变“unlocked”，当 release()调用重置锁状态为“unlocked”时，只有一个线程获得锁。 方法： Lock.acquire([blocking]) 获取一个锁，阻塞或者非阻塞。 当调用了带 blocking 参数并设置为 True（默认）时，其会阻塞直到锁被释放，然后设置锁为 locked，并返回 True。 当调用了带 blocking 参数并设置为 False 时，非阻塞，设置 lock 为“locked”，然后返回 True。 Lock.release() 释放一个锁。 当锁为“locked”的时候，重置为“unlocked”并返回。 当多个线程阻塞，等待锁被释放时，只有一个线程可获得锁。 无返回值。
RLock objects	(1)可重入锁是可以被同一个线程获取多次的同步机制，在内部，它使用线程占有和递归级别的概念去管理“locked”和“unlocked”状态。 (2)在“locked”状态下，某些线程占有这个可重入的锁，在“unlocked”状态下，没有线程占有。 (3)去锁定一个锁，线程调用 acquire()方法，如果线程占有该锁，这

	个方法返回一次。如果去释放锁，线程调用 release()方法。 (4)acquire()和 release()方法可以嵌套，但是必须配对。 RLock.acquire([blocking=1]) 当无参数调用该方法，若线程占有锁，增加一级嵌套并返回。否则，当另外一个线程占有该锁时，会被阻塞直到锁被释放。当锁被释放（并不被其他线程占有时），这个线程获取锁的所有权，并设置嵌套级别为 1 并返回。 当多个线程阻塞，等待锁被释放，只有一个线程会获取锁的占有权，这种情况下该方法无返回值。 当 blocking 设置为 true 时，跟无 blocking 参数是一样的，返回 true。 当 blocking 设置为 false 时，非阻塞。 RLock.release() 释放锁，并减小嵌套级别，减少至 0 时，重置锁为“unlocked”，若嵌套级别非 0，则该线程继续占有该锁，状态还是为“locked”。 释放一个“unolcked”的可重入锁时，会抛出 RuntimeError。 无返回值。

例子：

```
#-*- coding: utf-8 -*-

import threading
import time

count = 0

class TreadingTest(threading.Thread):
    def __init__(self, lock, threadName):
```

```
        '''
        :param lock: 锁对象
        :param threadName: 线程名称
        :return: None
        '''
        super(TreadingTest, self).__init__(name=threadName)
        self.lock = lock

    def run(self):
        '''
        重写父类 run 方法
        :return:
        '''
        global count
        self.lock.acquire()
        for i in xrange(10000):
            count += 1
        self.lock.release()
        print "self.getName(): %s" % self.getName()
        print "self.name: %s" % self.name
```

```
lock = threading.Lock()
for i in range(5):
    TreadingTest(lock, "thread-"+str(i)).start()
    time.sleep(2)
    print count
     print '------------------------'
```

经常使用 threading.Thread 就用类来继承，并重载 run()函数。

2.3 Condition、Semaphore、Event 对象

接口名称	接口作用
Condition　Oblect	(1)一个条件变量总是与某种锁联系在一起。这个锁可以是被传递过来的或者默认创建的，当多个线程共享同一个锁时，传递条件变量非常有用。 (2)condition 有与锁相关的 acquire()、release()方法。也有 wait()、notify()和 notifyAll()方法，这三个方法必须在线程获取到锁后调用，不然会抛 RuntimeError。 (3)wait()方法，先释放锁，然后阻塞直到线程被其他线程的 notify()或者 notifyAll()方法唤醒。一旦唤醒，该线程获取锁并返回，也可以指定一个超时时间。 (4))notify()方法唤醒一个等待条件变量的线程。如果多个线程都在等待，则 notifyall()方法唤醒所有的等待条件变量的线程。notify()和 notifyAll()方法并不释放锁， 　　这意味着被唤醒的线程不会立即从 wait()调用中返回，当且仅当调用 notify()和 notifyAll()的线程释放锁。 (5)条件变量典型的被用来与锁一起使用，作为一种同步方法去共享状态。线程可以重复调用 wait()去查询资源，也可以在改变状态时，调用 notify()或者 notifyAll()去通知等待者。 (6)如果 lock 参数给出且非 None，则该参数必须为 Lock 或者 RLock 对象，而且它被作为一个底层锁。否则创建一个新的 RLock 对象作为底层锁。

	方法 class threading.Condition([lock]) 如果 lock 参数给出而且非空，那么它必须为 Lock 或者一个 RLock 对象，并且被用作底层锁，否则，创建一个新的 RLock 作为底层锁。 acquire(*args) 获取底层锁，这个方法调用对应锁的 acquire()方法，返回锁方法的返回值。 release() 释放底层锁，无返回值。 wait([timeout]) 等待被通知或者超时。如果线程没有获取锁时，调用该方法，会抛 RuntimeError。 wait()方法，先释放锁，然后阻塞直到线程被其他线程的 notify()、notifyAll()方法唤醒或者超时。一旦唤醒，该线程获取锁并返回，也可以指定一个超时时间。 当 timeout 参数给出且非空时，其值为浮点数，指定超时的秒数。 当锁为 RLock 时，只有嵌套锁结束时才会被释放。然而一个内部用 RLock 的接口，即使在锁有几层嵌套的情况下，仍然会被释放，另外一个内部接口获取该锁时会存贮该锁的嵌套层数。 notify(n=1) 唤醒一个等待条件的线程，如果调用此方法的线程没有获得锁，则会抛 RuntimeError。 该方法唤醒做多 n 个等待条件变量的线程，如果没有线程等待，空操作。 如果等待的线程大于 n，则可以准备的唤醒 n 个线程，但是这个操作不安全。未来将优化可以唤醒 n+个。一个被唤醒的线程不会立即返回，除非获取了锁，notify()不会释放锁。 notify_all() notifyAll() 唤醒所有等待条件变量的线程，跟 notify()很相似，如果调用该方法的线程没有获取锁，则会抛 RuntimeError。
Semaphore Oblect	这个是计算机科学上的最古老的同步机制之一，由荷兰计算机科学家 Edsger W. Dijkstra 发明，他用 P()和 V()代替 acquire()和 release()。 一个信号量管理一个内部的计数器，这个计数器会在调用 acquire()时递减，在调用 release()的时候递增。计数器不能小于零，当 acquire()发现其值为 0，则阻塞，直到其他

	线程调用 release()。 信号量经常被用来监督有限的资源，例如数据库服务器。在其他资源容量固定的情况下，应该用有界信号量，在创建其他线程之前，在主线程中应该初始化信号量。 使用有界信号量可以减少信号量被释放的次数大于获取的次数错误的概率。 class threading.Semaphore([value]) value 参数是可选的，如果给出，则是给出内部计数器的初始值，默认为 1，如果 value 给的值小于 0，则抛 ValueError。 acquire([blocking]) 获取一个信号量。 当无参数调用时，如果内部的计数器大于 0，则计数器减一，然后返回，如果等于 0，则阻塞直到其他线程调用 release()，让计数器值大于 0。在多个线程调用 acquire()时，会被阻塞，release()会随机唤醒其中之一，跟线程阻塞的顺序无关，在这种情况下无返回值。 当 blocking 参数设置为 true，跟没有该参数一样，返回 true。 当 blocking 参数设置为 false 时，不阻塞。 release() 释放一个信号量，内部计数器加一。当计数器的值为 0，而且有其他线程在等待计数器大于 0，则唤醒该线程。
Event Objects	这是最线程间最简单的通信机制，一个线程创建一个事件，然后其他线程等待获取。 一个事件对象管理一个内部的标志位，这个标志位可以调用 set()设置为 true，调用 clear()设置为 false。wait()方法阻塞，直到标志位为 true。 class threading.Event 标志位初始化为 false is_set() isSet() 如果标志位为 true，则返回 true。 set() 设置标志位为 true，所有的线程等待标志位变为 true，然后被唤醒。线程调用 wait()方法后，标志位设为 true 之后，线程就不会阻塞。

	wait([timeout]) 阻塞，直到标志位为 true，如果标志位为 true，立即返回。否则阻塞，直到其他线程调用 set()将标准为设置为 true 或者超时。 超时时间 timeout 参数可选的，其值应该为浮点数，指出超时的时间（秒）。 这个方法退出的时候返回标志位，所以基本返回 true，除非超时时返回 false。

2.4 Timer 对象

这个类代表一个操作需要等待一定时间后才能执行，即定时器。

Timer 是 Thread 的一个子类，因此也可以作为创建子线程的一个例子。

计数器随着线程启动，通过调用 start()方法。计时器可以调用 cancel()方法停止。在执行这个操作之前，定时器等待的的时间间隔可能跟用户指定的不一样。

接口名称	接口作用
class threading.Timer(interval, function, args=(), kwasrgs={})	创建一个定时器，会在 interval 秒后执行带参数 args 和关键字 kwargs 的 function。
cancel()	停止定时器，停止定时器执行的操作，只能在定时器还在等待期间有用。

2.5 with/as 环境管理器

例子:

```
import threading

some_rlock = threading.RLock()
```

```
    with some_rlock:
    print "some_rlock is locked while this executes"
```

2.6 例子

```
# -*- coding: utf-8 -*-
import threading
import time

cond = threading.Condition()

class kongbaige(threading.Thread):
    def __init__(self, cond, name):
        super(kongbaige, self).__init__(name=name)
        #threading.Thread.__init__(self, name = name)
        self.cond = cond

    def run(self):
        self.cond.acquire()
        self.cond.acquire()
        print self.getName() + ':一支穿云箭'
        self.cond.notify()
        self.cond.wait()
```

```
            print self.name + ':山无棱，天地合，乃敢与君绝!'
            self.cond.notify()
            self.cond.wait()

            print self.getName() + ':紫薇！！！'
            self.cond.notify()
            self.cond.wait()

            print self.name + ':是你'
            self.cond.notify()
            self.cond.wait()

            print self.getName() + ':有钱吗，借点'
            self.cond.notify()
            self.cond.release()
            self.cond.release()

class ximige(threading.Thread):
    def __init__(self, cond, name):
        super(ximige, self).__init__(name=name)
        #threading.Thread.__init__(self, name = name)
```

```
        self.cond = cond

    def run(self):
        self.cond.acquire()
        self.cond.wait()

        print self.name + ':千军万马来相见'
        self.cond.notify()
        self.cond.wait()

        print self.name + ':海可枯，石可烂，激情永不散!'
        self.cond.notify()
        self.cond.wait()

        print self.getName() + ':尔康！！！'
        self.cond.notify()
        self.cond.wait()

        print self.name + ':是我'
        self.cond.notify()
        self.cond.wait()
```

```
        print self.name + ':滚'
        self.cond.release()


kongbai = kongbaige(cond, '空白哥')
ximi = ximige(cond, '西米')
ximi.start()
kongbai.start()
```

参考文献：

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

http://orangeholic.iteye.com/blog/1720421

http://blog.sina.com.cn/s/blog_9f488855010198uw.html

http://zeping.blog.51cto.com/6140112/1258977

http://blog.csdn.net/jgood/article/details/4305604