Python线程同步机制: Locks, RLocks, Semaphores, Conditions, Events和Queues

| Comments

翻译自Laurent Luce的博客

原文名称：Python threads synchronization: Locks, RLocks, Semaphores, Conditions, Events and Queues

原文连接：http://www.laurentluce.com/posts/python-threads-synchronization-locks-rlocks-semaphores-conditions-events-and-queues/

本文详细地阐述了Python线程同步机制。你将学习到以下有关Python线程同步机制：Lock，RLock，Semaphore，Condition，Event和Queue，还有Python的内部是如何实现这些机制的。 本文给出的程序的源代码可以在github上找到。

首先让我们来看一个没有使用线程同步的简单程序。

线程（Threading）

我们希望编程一个从一些URL中获得内容并且将内容写入文件的程序，完成这个程序可以不使用线程，为了加快获取的速度，我们使用2个线程，每个线程处理一半的URL。

注：完成这个程序的最好方式是使用一个URL队列，但是以下面的例子开始我的讲解更加合适。

类FetchUrls是threading.Thread的子类，他拥有一个URL列表和一个写URL内容的文件对象。

```
class FetchUrls(threading.Thread):
  """
  下载URL内容的线程
  """

  def __init__(self, urls, output):
    """
    构造器

    @param urls 需要下载的URL列表
    @param output 写URL内容的输出文件
    """
    threading.Thread.__init__(self)
    self.urls = urls
    self.output = output

  def run(self):
    """
    实现父类Thread的run方法，打开URL，并且一个一个的下载URL的内容
    """
```

```
    while self.urls:
      url = self.urls.pop()
      req = urllib2.Request(url)
      try:
        d = urllib2.urlopen(req)
      except urllib2.URLError, e:
        print 'URL %s failed: %s' % (url, e.reason)
      self.output.write(d.read())
      print 'write done by %s' % self.name
      print 'URL %s fetched by %s' % (url, self.name)
```

main函数启动了两个线程，之后让他们下载URL内容。

```
def main():
  # URL列表1
  urls1 = ['http://www.google.com', 'http://www.facebook.com']
  # URL列表2
  urls2 = ['http://www.yahoo.com', 'http://www.youtube.com']
  f = open('output.txt', 'w+')
  t1 = FetchUrls(urls1, f)
  t2 = FetchUrls(urls2, f)
  t1.start()
  t2.start()
  t1.join()
  t2.join()
  f.close()

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

上面的程序将出现两个线程同时写一个文件的情况，导致文件一团乱码。我们需要找到一种在给定的时间里只有一个线程写文件的方法。实现的方法就是使用像锁（Locks）这样的线程同步机制。

锁（Lock）

锁有两种状态：被锁（locked）和没有被锁（unlocked）。拥有acquire()和release()两种方法，并且遵循一下的规则：

- 如果一个锁的状态是unlocked，调用acquire()方法改变它的状态为locked；
- 如果一个锁的状态是locked，acquire()方法将会阻塞，直到另一个线程调用release()方法释放了锁；
- 如果一个锁的状态是unlocked调用release()会抛出RuntimeError异常；
- 如果一个锁的状态是locked，调用release()方法改变它的状态为unlocked。

解决上面两个线程同时写一个文件的问题的方法就是：我们给类FetchUrls的构造器中传入一个锁（lock），使用这个锁来保护文件操作，实现在给定的时间只有一个线程

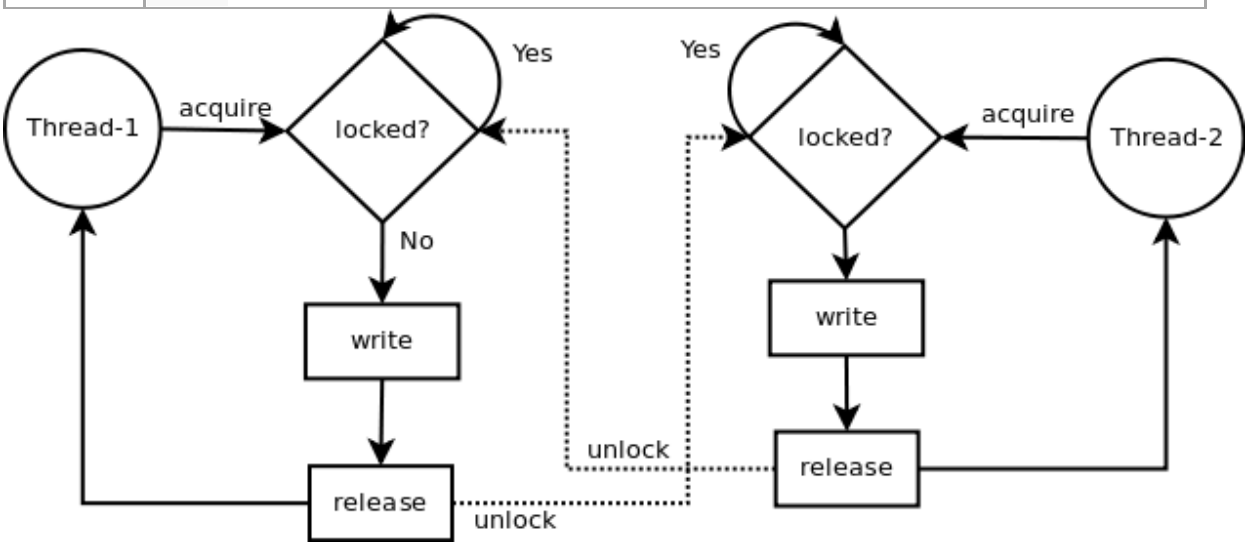
写文件。下面的代码只显示了关于lock部分的修改。完整的源码可以在threads/lock.py中找到。

```
class FetchUrls(threading.Thread):
  ...

  def __init__(self, urls, output, lock):
    ...
    self.lock = lock     #传入的lock对象

  def run(self):
    ...
    while self.urls:
      ...
      self.lock.acquire()          #获得lock对象，lock状态变为locked，并且阻塞其他线程获取lock对象（写文件的权利）
      print 'lock acquired by %s' % self.name
      self.output.write(d.read())
      print 'write done by %s' % self.name
      print 'lock released by %s' % self.name
      self.lock.release()          #释放lock对象，lock状态变为unlocked，其他的线程可以重新获取lock对象
      ...

def main():
  ...
  lock = threading.Lock()
  ...
  t1 = FetchUrls(urls1, f, lock)
  t2 = FetchUrls(urls2, f, lock)
  ...
```



以下是程序的输出：

```
$ python locks.py
lock acquired by Thread-2
write done by Thread-2
lock released by Thread-2
URL http://www.youtube.com fetched by Thread-2
lock acquired by Thread-1
write done by Thread-1
lock released by Thread-1
```

```
URL http://www.facebook.com fetched by Thread-1
lock acquired by Thread-2
write done by Thread-2
lock released by Thread-2
URL http://www.yahoo.com fetched by Thread-2
lock acquired by Thread-1
write done by Thread-1
lock released by Thread-1

URL http://www.google.com fetched by Thread-1
```

从上面的输出我们可以看出，写文件的操作被锁保护，没有出现两个线程同时写一个文件的现象。

下面我们看一下Python内部是如何实现锁（Lock）的。我正在使用的Python版本是Linux操作系统上的Python 2.6.6。

threading模块的Lock()方法就是thread.allocate_lock，代码可以在Lib/threading.py中找到。

```
Lock = _allocate_lock
_allocate_lock = thread.allocate_lock
```

C的实现在Python/thread_pthread.h中。程序假定你的系统支持POSIX信号量（semaphores）。sem_init()初始化锁（Lock）所在地址的信号量。初始的信号量值是1，意味着锁没有被锁（unlocked）。信号量将在处理器的不同线程之间共享。

```
PyThread_type_lock
PyThread_allocate_lock(void)
{

    ...
    lock = (sem_t *)malloc(sizeof(sem_t));

    if (lock) {
        status = sem_init(lock,0,1);
        CHECK_STATUS("sem_init");
        ....
    }
    ...
}
```

当acquire()方法被调用时，下面的C代码将被执行。默认的waitflag值是1，表示调用将被被阻塞直到锁被释放。sem_wait()方法减少信号量的值或者被阻塞直到信号量大于零。

```
int
PyThread_acquire_lock(PyThread_type_lock lock, int waitflag)
{
    ...
    do {
        if (waitflag)
            status = fix_status(sem_wait(thelock));
        else
            status = fix_status(sem_trywait(thelock));
    } while (status == EINTR); /* Retry if interrupted by a
signal */
    ....
```

```
}
```

当release()方法被调用时，下面的C代码将被执行。sem_post()方法增加信号量。

```
void
PyThread_release_lock(PyThread_type_lock lock)
{
    ...
    status = sem_post(thelock);
    ...
}
```

可以将锁（Lock）与“with”语句一起使用，锁可以作为上下文管理器（context manager）。使用“with”语句的好处是：当程序执行到“with”语句时，acquire()方法将被调用，当程序执行完“with”语句时，release()方法会被调用（译注：这样我们就不用显示地调用acquire()和release()方法，而是由“with”语句根据上下文来管理锁的获取和释放。）下面我们用“with”语句重写FetchUrls类。

```
class FetchUrls(threading.Thread):
  ...
  def run(self):
    ...
    while self.urls:
      ...
      with self.lock:    #使用“with”语句管理锁的获取和释放
        print 'lock acquired by %s' % self.name
        self.output.write(d.read())
        print 'write done by %s' % self.name
        print 'lock released by %s' % self.name
      ...
```

可重入锁（RLock）

RLock是可重入锁（reentrant lock），acquire()能够不被阻塞的被同一个线程调用多次。要注意的是release()需要调用与acquire()相同的次数才能释放锁。

使用Lock，下面的代码第二次调用acquire()时将被阻塞：

```
lock = threading.Lock()
lock.acquire()
lock.acquire()
```

如果你使用的是RLock，下面的代码第二次调用acquire()不会被阻塞:

```
rlock = threading.RLock()
rlock.acquire()
rlock.acquire()
```

RLock使用的同样是thread.allocate_lock()，不同的是他跟踪宿主线程（the owner thread）来实现可重入的特性。下面是RLock的acquire()实现。如果调用acquire()的线程是资源的所有者，记录调用acquire()次数的计数器就会加1。如果不是，就将试图去获取锁。线程第一次获得锁时，锁的拥有者将会被保存，同时计数器初始化为1。

```
def acquire(self, blocking=1):
    me = _get_ident()
    if self.__owner == me:
        self.__count = self.__count + 1
```

```
            self.__count = self.__count + 1
            ...
            return 1
        rc = self.__block.acquire(blocking)

        if rc:
            self.__owner = me
            self.__count = 1
            ...
        ...
        return rc
```

下面我们看一下可重入锁（RLock）的release()方法。首先它会去确认调用者是否是锁的拥有者。如果是的话，计数器减1；如果计数器为0，那么锁将会被释放，这时其他线程就可以去获取锁了。

```
def release(self):
    if self.__owner != _get_ident():
        raise RuntimeError("cannot release un-acquired lock")
    self.__count = count = self.__count - 1
    if not count:
        self.__owner = None
        self.__block.release()
        ...
    ...
```

条件（Condition）

条件同步机制是指：一个线程等待特定条件，而另一个线程发出特定条件满足的信号。 解释条件同步机制的一个很好的例子就是生产者/消费者（producer/consumer）模型。生产者随机的往列表中“生产”一个随机整数，而消费者从列表中“消费”整数。完整的源码可以在threads/condition.py中找到

在producer类中，producer获得锁，生产一个随机整数，通知消费者有了可用的“商品”，并且释放锁。producer无限地向列表中添加整数，同时在两个添加操作中间随机的停顿一会儿。

```
class Producer(threading.Thread):
  """
  向列表中生产随机整数
  """

  def __init__(self, integers, condition):
    """
    构造器

    @param integers 整数列表
    @param condition 条件同步对象
    """
    threading.Thread.__init__(self)
    self.integers = integers
    self.condition = condition

  def run(self):
    """
```

```
    实现Thread的run方法。在随机时间向列表中添加一个随机整数
    """
    while True:
      integer = random.randint(0, 256)
      self.condition.acquire()   #获取条件锁
      print 'condition acquired by %s' % self.name
      self.integers.append(integer)
      print '%d appended to list by %s' % (integer,
self.name)
      print 'condition notified by %s' % self.name
      self.condition.notify()    #唤醒消费者线程
      print 'condition released by %s' % self.name
      self.condition.release()   #释放条件锁
      time.sleep(1)              #暂停1秒钟
```

下面是消费者（consumer）类。它获取锁，检查列表中是否有整数，如果没有，等待生产者的通知。当消费者获取整数之后，释放锁。

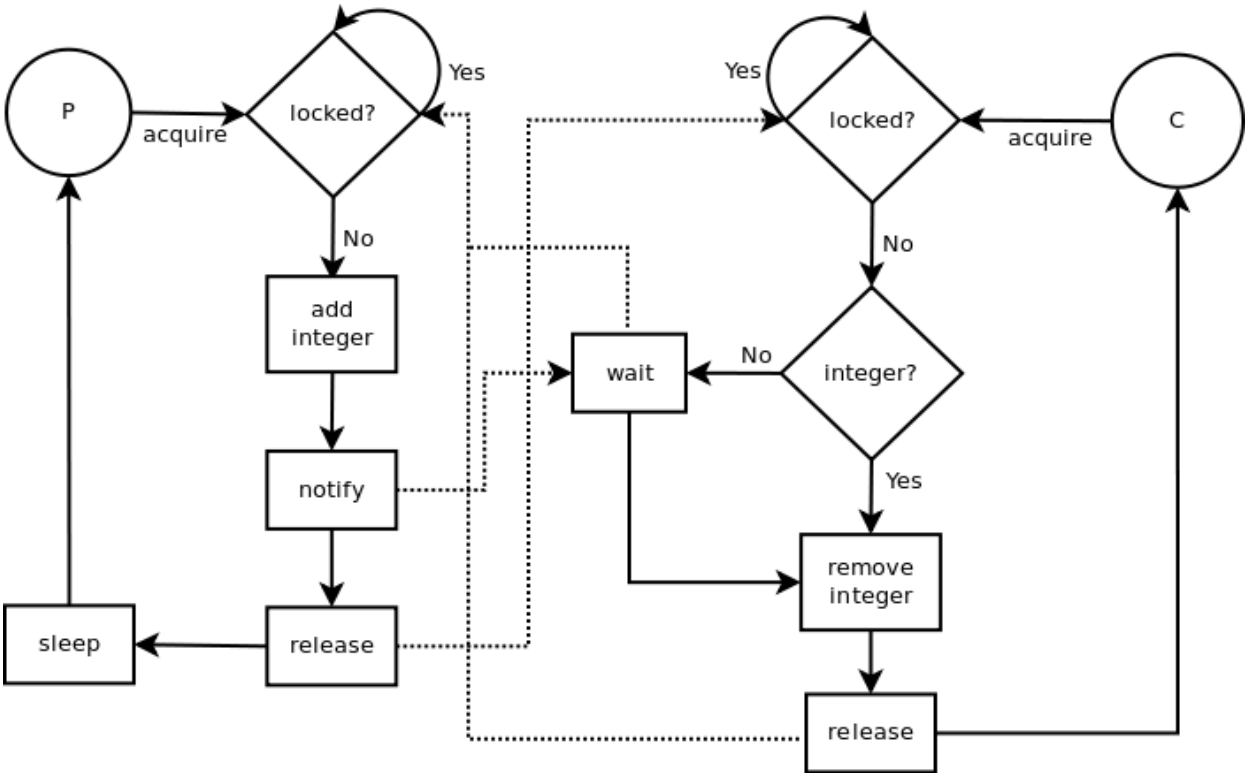
注意在wait()方法中会释放锁，这样生产者就能获得资源并且生产“商品”。

```
class Consumer(threading.Thread):
  """
  从列表中消费整数
  """

  def __init__(self, integers, condition):
    """
    构造器

    @param integers 整数列表
    @param condition 条件同步对象
    """
    threading.Thread.__init__(self)
    self.integers = integers
    self.condition = condition

  def run(self):
    """
    实现Thread的run()方法，从列表中消费整数
    """
    while True:
      self.condition.acquire()   #获取条件锁
      print 'condition acquired by %s' % self.name
      while True:
        if self.integers:        #判断是否有整数
          integer = self.integers.pop()
          print '%d popped from list by %s' % (integer,
self.name)
          break
        print 'condition wait by %s' % self.name
        self.condition.wait()    #等待商品，并且释放资源
      print 'condition released by %s' % self.name
      self.condition.release()   #最后释放条件锁
```



下面我们编写main方法，创建两个线程：

```
def main():
  integers = []
  condition = threading.Condition()
  t1 = Producer(integers, condition)
  t2 = Consumer(integers, condition)
  t1.start()
  t2.start()
  t1.join()
  t2.join()


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

下面是程序的输出：

```
$ python condition.py
condition acquired by Thread-1
159 appended to list by Thread-1
condition notified by Thread-1
condition released by Thread-1
condition acquired by Thread-2
159 popped from list by Thread-2
condition released by Thread-2
condition acquired by Thread-2
condition wait by Thread-2
condition acquired by Thread-1
116 appended to list by Thread-1
condition notified by Thread-1
condition released by Thread-1
116 popped from list by Thread-2
condition released by Thread-2
condition acquired by Thread-2
condition wait by Thread-2
```

Thread-1添加159到列表中，通知消费者同时释放锁，Thread-2获得锁，取回159，并且释放锁。此时因为执行time.sleep(1)，生产者正在睡眠，当消费者再次试图获取整数时，列表中并没有整数，这时消费者进入等待状态，等待生产者的通知。当wait()被调用时，它会释放资源，从而生产者能够利用资源生产整数。

下面我们看一下Python内部是如何实现条件同步机制的。如果用户没有传入锁（lock）对象，condition类的构造器创建一个可重入锁（RLock），这个锁将会在调用acquire()和release()时使用。

```
class _Condition(_Verbose):

    def __init__(self, lock=None, verbose=None):
        _Verbose.__init__(self, verbose)
        if lock is None:
            lock = RLock()
        self.__lock = lock
```

接下来是wait()方法。为了简化说明，我们假定在调用wait()方法时不使用timeout参数。wait()方法创建了一个名为waiter的锁，并且设置锁的状态为locked。这个waiter锁用于线程间的通讯，这样生产者（在生产完整数之后）就可以通知消费者释放waiter()锁。锁对象将会被添加到等待者列表，并且在调用waiter.acquire()时被阻塞。一开始condition锁的状态被保存，并且在wait()结束时被恢复。

```
def wait(self, timeout=None):
    ...

    waiter = _allocate_lock()
    waiter.acquire()
    self.__waiters.append(waiter)
    saved_state = self._release_save()
    try:    # 无论如何恢复状态（例如，KeyboardInterrupt）
        if timeout is None:
            waiter.acquire()
            ...
        ...
    finally:
        self._acquire_restore(saved_state)
```

当生产者调用notify()方法时，notify()释放waiter锁，唤醒被阻塞的消费者。

```
def notify(self, n=1):
    ...
    __waiters = self.__waiters
    waiters = __waiters[:n]
    ...
    for waiter in waiters:
        waiter.release()
        try:
            __waiters.remove(waiter)
        except ValueError:
            pass
```

同样Condition对象也可以和“with”语句一起使用，这样“with”语句上下文会帮我们调用acquire()和release()方法。下面的代码使用“with”语句改写了生产者和消费者类。

```
class Producer(threading.Thread):
  ...
  def run(self):
    while True:
      integer = random.randint(0, 256)
      with self.condition:
        print 'condition acquired by %s' % self.name
        self.integers.append(integer)
        print '%d appended to list by %s' % (integer, self.name)
        print 'condition notified by %s' % self.name
        self.condition.notify()
        print 'condition released by %s' % self.name
      time.sleep(1)

class Consumer(threading.Thread):
  ...
  def run(self):
    while True:
      with self.condition:
        print 'condition acquired by %s' % self.name
        while True:
          if self.integers:
            integer = self.integers.pop()
            print '%d popped from list by %s' % (integer,
self.name)
            break
          print 'condition wait by %s' % self.name
          self.condition.wait()
        print 'condition released by %s' % self.name
```

信号量（Semaphore）

信号量同步基于内部计数器，每调用一次acquire()，计数器减1；每调用一次release()，计数器加1.当计数器为0时，acquire()调用被阻塞。这是迪科斯彻（Dijkstra）信号量概念P()和V()的Python实现。信号量同步机制适用于访问像服务器这样的有限资源。

信号量同步的例子：

```
semaphore = threading.Semaphore()
semaphore.acquire()
 # 使用共享资源
...
semaphore.release()
```

让我们看一下信号量同步在Python内部是如何实现的。构造器使用参数value来表示计数器的初始值，默认值为1。一个条件锁实例用于保护计数器，同时当信号量被释放时通知其他线程。

```
class _Semaphore(_Verbose):
    ...
    def __init__(self, value=1, verbose=None):
        _Verbose.__init__(self, verbose)
```

```
        self.__cond = Condition(Lock())
        self.__value = value
        ...
```

acquire()方法。如果信号量为0，线程被条件锁的wait()方法阻塞，直到被其他线程唤醒；如果计数器大于0，调用acquire()使计数器减1。

```
def acquire(self, blocking=1):
    rc = False
    self.__cond.acquire()
    while self.__value == 0:
        ...
        self.__cond.wait()
    else:
        self.__value = self.__value - 1
        rc = True
    self.__cond.release()
    return rc
```

信号量类的release()方法增加计数器的值并且唤醒其他线程。

```
def release(self):
    self.__cond.acquire()
    self.__value = self.__value + 1
    self.__cond.notify()
    self.__cond.release()
```

还有一个“有限”(bounded)信号量类，可以确保release()方法的调用次数不能超过给定的初始信号量数值(value参数)，下面是“有限”信号量类的Python代码：

```
class _BoundedSemaphore(_Semaphore):
    """检查release()的调用次数是否小于等于acquire()次数"""
    def __init__(self, value=1, verbose=None):
        _Semaphore.__init__(self, value, verbose)
        self._initial_value = value

    def release(self):
        if self._Semaphore__value >= self._initial_value:
            raise ValueError, "Semaphore released too many
times"
        return _Semaphore.release(self)
```

同样信号量(Semaphore)对象可以和“with”一起使用：

```
semaphore = threading.Semaphore()
with semaphore:
  # 使用共享资源
  ...
```

事件（Event）

基于事件的同步是指：一个线程发送/传递事件，另外的线程等待事件的触发。 让我们再来看看前面的生产者和消费者的例子，现在我们把它转换成使用事件同步而不是条件同步。完整的源码可以在threads/event.py里面找到。

首先是生产者类，我们传入一个Event实例给构造器而不是Condition实例。一旦整数被添加进列表，事件(event)被设置和发送去唤醒消费者。注意事件(event)实例默认是

被发送的。

```
class Producer(threading.Thread):
    """
    向列表中生产随机整数
    """

    def __init__(self, integers, event):
        """
        构造器

        @param integers 整数列表
        @param event 事件同步对象
        """
        threading.Thread.__init__(self)
        self.integers = integers
        self.event = event

    def run(self):
        """
        实现Thread的run方法。在随机时间向列表中添加一个随机整数
        """
        while True:
            integer = random.randint(0, 256)
            self.integers.append(integer)
            print '%d appended to list by %s' % (integer, self.name)
            print 'event set by %s' % self.name
            self.event.set()                    #设置事件
            self.event.clear()                  #发送事件
            print 'event cleared by %s' % self.name
            time.sleep(1)
```

同样我们传入一个Event实例给消费者的构造器，消费者阻塞在wait()方法，等待事件被触发，即有可供消费的整数。

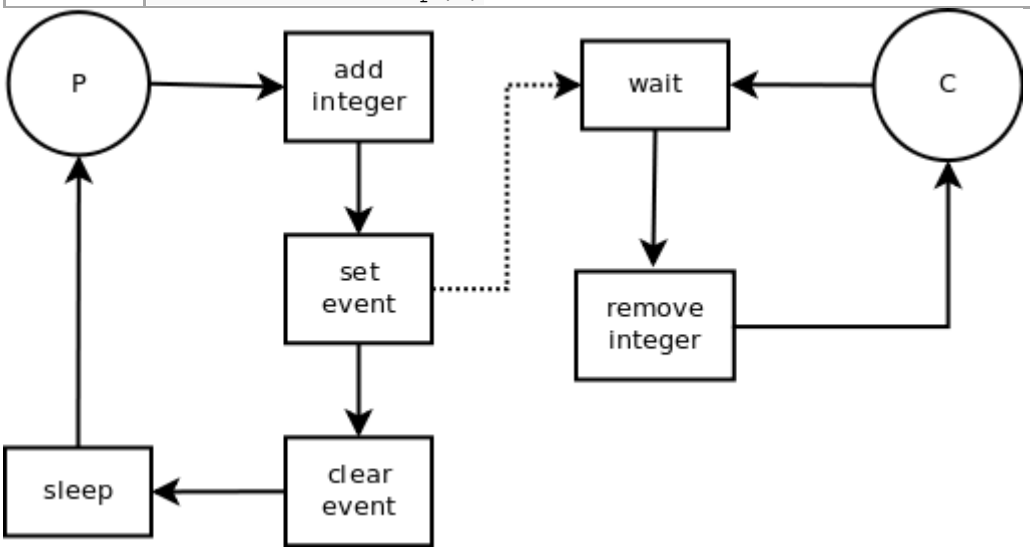
```
class Consumer(threading.Thread):
    """
     从列表中消费整数
    """

    def __init__(self, integers, event):
        """
        构造器

        @param integers 整数列表
        @param event 事件同步对象
        """
        threading.Thread.__init__(self)
        self.integers = integers
        self.event = event

    def run(self):
        """
        实现Thread的run()方法，从列表中消费整数
        """
        while True:
            self.event.wait()  #等待事件被触发
```

```
23          try:
24            integer = self.integers.pop()
25            print '%d popped from list by %s' % (integer, self.name)
26          except IndexError:
27            # catch pop on empty list
28            time.sleep(1)
```



下面是程序的输出，Thread-1添加124到整数列表中，然后设置事件并且唤醒消费者。消费者从wait()方法中唤醒，在列表中获取到整数。

```
1  $ python event.py
2  124 appended to list by Thread-1
3  event set by Thread-1
4  event cleared by Thread-1
5  124 popped from list by Thread-2
6  223 appended to list by Thread-1
7  event set by Thread-1
8  event cleared by Thread-1
9  223 popped from list by Thread-2
```

事件锁的Python内部实现，首先是Event锁的构造器。构造器中创建了一个条件（Condition）锁，来保护事件标志（event flag）,同事唤醒其他线程当事件被设置时。

```
1  class _Event(_Verbose):
2      def __init__(self, verbose=None):
3          _Verbose.__init__(self, verbose)
4          self.__cond = Condition(Lock())
5          self.__flag = False
```

接下来是set()方法，它设置事件标志为True，并且唤醒其他线程。条件锁对象保护程序修改事件标志状态的关键部分。

```
1  def set(self):
2      self.__cond.acquire()
3      try:
4          self.__flag = True
5          self.__cond.notify_all()
6      finally:
7          self.__cond.release()
```

而clear()方法正好相反，它设置时间标志为False。

```
def clear(self):
    self.__cond.acquire()
    try:
        self.__flag = False
    finally:
        self.__cond.release()
```

最后，wait()方法将阻塞直到调用了set()方法，当事件标志为True时，wait()方法就什么也不做。

```
def wait(self, timeout=None):
    self.__cond.acquire()
    try:
        if not self.__flag:      #如果flag不为真
            self.__cond.wait(timeout)
    finally:
        self.__cond.release()
```

队列（Queue）

队列是一个非常好的线程同步机制，使用队列我们不用关心锁，队列会为我们处理锁的问题。 队列(Queue)有以下4个用户感兴趣的方法：

- **put:** 向队列中添加一个项；
- **get:** 从队列中删除并返回一个项；
- **task_done:** 当某一项任务完成时调用；
- **join:** 阻塞知道所有的项目都被处理完。

下面我们将上面的生产者/消费者的例子转换成使用队列。源代码可以在threads/queue.py中找到。

首先是生产者类，我们不需要传入一个整数列表，因为我们使用队列就可以存储生成的整数。生产者线程在一个无限循环中生成整数并将生成的整数添加到队列中。

```
class Producer(threading.Thread):
    """
    向队列中生产随机整数
    """

    def __init__(self, queue):
        """
        构造器

        @param integers 整数列表      #译注：不需要这个参数
        @param queue 队列同步对象
        """
        threading.Thread.__init__(self)
        self.queue = queue

    def run(self):
        """
        实现Thread的run方法。在随机时间向队列中添加一个随机整数
```

```
        """
        while True:
            integer = random.randint(0, 256)
            self.queue.put(integer)    #将生成的整数添加到队列
            print '%d put to queue by %s' % (integer, self.name)
            time.sleep(1)
```

下面是消费者类。线程从队列中获取整数，并且在任务完成时调用task_done()方法。

```
class Consumer(threading.Thread):
    """
    从队列中消费整数
    """

    def __init__(self, queue):
        """
        构造器

        @param integers 整数列表      #译注：不需要这个参数
        @param queue 队列同步对象
        """
        threading.Thread.__init__(self)
        self.queue = queue

    def run(self):
        """
        实现Thread的run()方法，从队列中消费整数
        """
        while True:
            integer = self.queue.get()
            print '%d popped from list by %s' % (integer,
self.name)
            self.queue.task_done()
```

以下是程序的输出：

```
$ python queue.py
61 put to queue by Thread-1
61 popped from list by Thread-2
6 put to queue by Thread-1
6 popped from list by Thread-2
```

队列同步的最大好处就是队列帮我们处理了锁。现在让我们去看看在Python内部是如何实现队列同步机制的。

队列（Queue）构造器创建一个锁，保护队列元素的添加和删除操作。同时创建了一些条件锁对象处理队列事件，比如队列不空事件（削除get()的阻塞），队列不满事件（削除put()的阻塞）和所有项目都被处理完事件（削除join()的阻塞）。

```
class Queue:
    def __init__(self, maxsize=0):
        ...
        self.mutex = threading.Lock()
        self.not_empty = threading.Condition(self.mutex)
        self.not_full = threading.Condition(self.mutex)
        self.all_tasks_done = threading.Condition(self.mutex)
        self.unfinished_tasks = 0
```

put()方法向队列中添加一个项，或者阻塞如果队列已满。这时队列非空，它唤醒阻塞在get()方法中的线程。更多关于Condition锁的内容请查看上面的讲解。

```
def put(self, item, block=True, timeout=None):
    ...
    self.not_full.acquire()
    try:
        if self.maxsize > 0:
            ...
            elif timeout is None:
                while self._qsize() == self.maxsize:
                    self.not_full.wait()
        self._put(item)
        self.unfinished_tasks += 1
        self.not_empty.notify()
    finally:
        self.not_full.release()
```

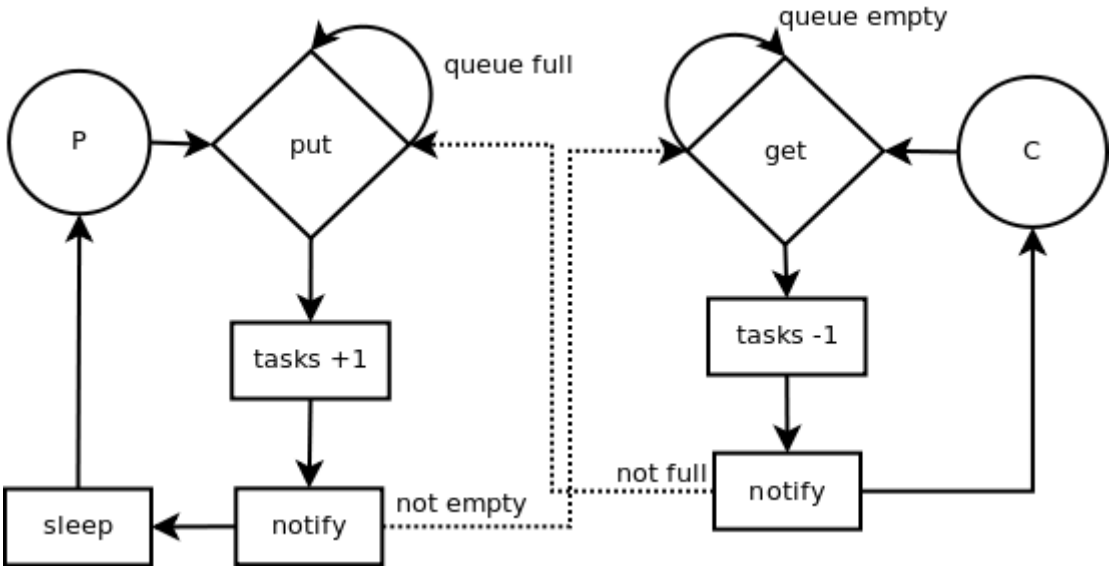
get()方法从队列中获得并删除一个项，或者阻塞当队列为空时。这时队列不满，他唤醒阻塞在put()方法中的线程。

```
def get(self, block=True, timeout=None):
    ...
    self.not_empty.acquire()
    try:
        ...
        elif timeout is None:
            while not self._qsize():
                self.not_empty.wait()
        item = self._get()
        self.not_full.notify()
        return item
    finally:
        self.not_empty.release()
```

当调用task_done()方法时，未完成任务的数量减1。如果未完成任务的数量为0，线程等待队列完成join()方法。

```
def task_done(self):
    self.all_tasks_done.acquire()
    try:
        unfinished = self.unfinished_tasks - 1
        if unfinished <= 0:
            if unfinished < 0:
                raise ValueError('task_done() called too many
times')
            self.all_tasks_done.notify_all()
        self.unfinished_tasks = unfinished
    finally:
        self.all_tasks_done.release()

def join(self):
    self.all_tasks_done.acquire()
    try:
        while self.unfinished_tasks:
            self.all_tasks_done.wait()
    finally:
        self.all_tasks_done.release()
```



本文到此结束，希望您喜欢这篇文章。欢迎您的留言和反馈。

---EOF---