多线程爬取和解析

python多线程基础

多线程类似于同时执行多个不同程序。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

每个线程都有他自己的一组CPU寄存器，称为线程的上下文，该上下文反映了线程上次运行该线程的CPU寄存器的状态。指令指针和堆栈指针寄存器是线程上下文中两个最重要的寄存器，线程总是在进程得到上下文中运行的，这些地址都用于标志拥有线程的进程地址空间中的内存。

- 线程可以被抢占（中断）。
- 在其他线程正在运行时，线程可以暂时搁置（也称为睡眠） -- 这就是线程的退让。

Python3 线程中常用的两个模块为:

- thread 已被废弃
- threading 推荐使用

多线程和多进程最大的不同在于，多进程中，同一个变量，各自有一份拷贝存在于每个进程中，互不影响，而多线程中，所有变量都由所有线程共享，所以，任何一个变量都可以被任何一个线程修改，因此，线程之间共享数据最大的危险在于多个线程同时改一个变量，把内容给改乱了。

使用**threading**创建线程类

Threading类提供了以下方法:

- run(): 用以表示线程活动的方法。
- start():启动线程活动。
- join([time]): 等待至线程中止。这阻塞调用线程直至线程的join() 方法被调用中止-正常退出或者抛出未处理的异常-或者是可选的超时发生。
- isAlive(): 返回线程是否活动的。
- getName(): 返回线程名。
- setName(): 设置线程名。

Python中使用线程有两种方式：函数或者用类来包装线程对象。我们下面介绍使用 threading 模块创建线程类。我们可以通过直接从 threading.Thread 继承创建一个新的子类，并实例化后调用 start() 方法启动新线程，即它调用了线程的 run() 方法。

In [2]:
```
import threading
import time

exitFlag = 0

class myThread(threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        #super(myThread).__init__(self)
        self.threadID = threadID
        self.name = name
```

```
        self.counter = counter
    def run(self):
        print("开始线程: " + self.name)
        print_time(self.name, self.counter, 5)
        print ("退出线程: " + self.name)

def print_time(threadName, delay, counter):
    while counter:
        if exitFlag:
            threadName.exit()
        time.sleep(delay)
        print ("%s: %s" % (threadName, time.ctime(time.time())))
        counter -= 1

# 创建新线程
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# 开启新线程
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print ("退出主线程")
```

```
开始线程: Thread-1开始线程: Thread-2

Thread-1: Tue Oct 23 19:33:35 2018
Thread-2: Tue Oct 23 19:33:36 2018Thread-1: Tue Oct 23 19:33:36 2018

Thread-1: Tue Oct 23 19:33:37 2018
Thread-1: Tue Oct 23 19:33:38 2018Thread-2: Tue Oct 23 19:33:38 2018

Thread-1: Tue Oct 23 19:33:39 2018
退出线程: Thread-1
Thread-2: Tue Oct 23 19:33:40 2018
Thread-2: Tue Oct 23 19:33:42 2018
Thread-2: Tue Oct 23 19:33:44 2018
退出线程: Thread-2
退出主线程
```

线程同步

如果多个线程共同对某个数据修改，则可能出现不可预料的结果，为了保证数据的正确性，需要对多个线程进行同步。

使用 Thread 对象的 Lock 和 Rlock 可以实现简单的线程同步，这两个对象都有 acquire 方法和 release 方法，对于那些需要每次只允许一个线程操作的数据，可以将其操作放到 acquire 和 release 方法之间。如下：

多线程的优势在于可以同时运行多个任务（至少感觉起来是这样）。但是当线程需要共享数据时，可能存在数据不同步的问题。

考虑这样一种情况：一个列表里所有元素都是0，线程"set"从后向前把所有元素改成1，而线程"print"负责从前往后读取列表并打印。

那么，可能线程"set"开始改的时候，线程"print"便来打印列表了，输出就成了一半0一半1，这就是数据的不同步。为了避免这种情况，引入了锁的概念。

锁有两种状态——锁定和未锁定。每当一个线程比如"set"要访问共享数据时，必须先获得锁定；如果已经有别的线程比如"print"获得锁定了，那么就让线程"set"暂停，也就是同步阻塞；等到线程"print"访问完毕，释放锁以后，再让线程"set"继续。

经过这样的处理，打印列表时要么全部输出0，要么全部输出1，不会再出现一半0一半1的尴尬场面。

In [3]:
```
import threading
import time

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print ("开启线程:  " + self.name)
        # 加锁，用于线程同步
        threadLock.acquire()
        print_time(self.name, self.counter, 3)
        # 释放锁，开启下一个线程
        threadLock.release()

def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        print ("%s: %s" % (threadName, time.ctime(time.time())))
        counter -= 1

threadLock = threading.Lock()
threads = []

# 创建新线程
thread1 = myThread(1, "Thread-1", 1)

thread2 = myThread(2, "Thread-2", 2)

# 开启新线程
thread1.start()
#time.sleep(1)
thread2.start()

# 添加线程到线程列表
threads.append(thread1)
threads.append(thread2)

# 等待所有线程完成
for t in threads:
    t.join()
print ("退出主线程")
```

```
开启线程:  Thread-1开启线程:  Thread-2

Thread-1: Tue Oct 23 19:38:42 2018
Thread-1: Tue Oct 23 19:38:43 2018
Thread-1: Tue Oct 23 19:38:45 2018
Thread-2: Tue Oct 23 19:38:47 2018
Thread-2: Tue Oct 23 19:38:49 2018
Thread-2: Tue Oct 23 19:38:51 2018
```

退出主线程

线程优先级队列（ **Queue**）

Python 的 Queue 模块中提供了同步的、线程安全的队列类。

queue模块中实现了3种不同的队列，区别仅在于获取队列中元素的顺序不同。

- FIFO队列，先进先出，经典队列；
  - class queue.Queue(maxsize=0)
- LIFO队列，后进先出，类似常见的堆栈；
  - class queue.LifoQueue(maxsize=0)
- priority队列，元素经排序后存放在队列里（使用heapq模块），值最小的元素最先被取出。
  - class queue.PriorityQueue(maxsize=0)

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。 模块设计了两个异常类，用于判断队列满或空：

- exception queue.Empty
  - Exception raised when non-blocking get() (or get_nowait()) is called on a Queue object which is empty.
- exception queue.Full
  - Exception raised when non-blocking put() (or put_nowait()) is called on a Queue object which is full.

queue.Queue对象的常用方法

- Queue.qsize() 返回队列的大小
- Queue.empty() 如果队列为空，返回True,反之False
- Queue.full() 如果队列满了，返回True,反之False
- Queue.full 与 maxsize 大小对应
- Queue.get([block[, timeout]])获取队列，timeout等待时间
- Queue.put_nowait(item) Equivalent to put(item, False).
- Queue.get_nowait() Equivalent to get(False).
- Queue.task_done() 在完成一项工作之后，Queue.task_done()函数向任务已经完成的队列发送一个信号
- Queue.join() 实际上意味着等到队列为空，再执行别的操作

In [10]:
```
"""FIFO队列对象的基本使用"""
import queue

q = queue.Queue()

for i in range(5):
    q.put(i)

while not q.empty():
    print(q.get())
```

```
0
1
2
3
4
```

In [11]:
```
"""LIFO队列对象的基本使用"""
import queue

q = queue.LifoQueue()

for i in range(5):
    q.put(i)

while not q.empty():
    print( q.get())
```

```
4
3
2
1
0
```

In [9]:
```
import queue
import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, q):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.q = q
    def run(self):
        print ("开启线程: " + self.name)
        process_data(self.name, self.q)
        print ("退出线程: " + self.name)

def process_data(threadName, q):
    while not exitFlag:
        queueLock.acquire()
        if not workQueue.empty():
            data = q.get()
            queueLock.release()
            print ("%s processing %s" % (threadName, data))
        else:
            queueLock.release()
        time.sleep(1)


threadList = ["Thread-1", "Thread-2", "Thread-3"]
nameList = ["One", "Two", "Three", "Four", "Five"]
queueLock = threading.Lock()
workQueue = queue.Queue(10)
threads = []
threadID = 1

# 创建新线程
for tName in threadList:
    thread = myThread(threadID, tName, workQueue)
    thread.start()
    threads.append(thread)
    threadID += 1
```

```
# 填充队列
queueLock.acquire()
for word in nameList:
    workQueue.put(word)
queueLock.release()

# 等待队列清空
while not workQueue.empty():
    pass

# 通知线程是时候退出
exitFlag = 1

# 等待所有线程完成
for t in threads:
    t.join()
print ("退出主线程")
```

```
开启线程: Thread-1
开启线程: Thread-2
开启线程: Thread-3
Thread-3 processing One
Thread-1 processing TwoThread-2 processing Three

Thread-3 processing Four
Thread-1 processing Five
退出线程: Thread-3
退出线程: Thread-2
退出线程: Thread-1
退出主线程
```

多线程网络爬虫示例

下面介绍一个采集百度贴吧页面并解析的多线程程序。这个程序有以下两个线程类和一个main函数组成。

- class Fecther
  - 采集页面类，用于获取URL对应的页面，这个类基于threading.Thread，可以生成采集线程1，2，3，...
- class Parser
  - 内容解析类，用于解析已爬取到的网页，这个类基于threading.Thread，可以生成采集线程1，2，3，...
- main函数：
  - 调度程序执行过程，生成并启动线程

In [ ]:

```
"""多线程爬虫实例"""
import queue
import threading
import time
import json
import requests
from bs4 import BeautifulSoup
import time

class Fecther(threading.Thread):
    """
    百度贴吧https://tieba.baidu.com/f?kw=%E7%BD%91%E7%BB%9C%E7%88%AC%E8%9
```

```
9%AB&页面获取类
    """
    def __init__(self,threadName,pageQueue,htmlQueue):
        #threading.Thread.__init__(self)
        #下面调用父类初始化的方法更好。
        super(Fecther,self).__init__()
        self.threadName = threadName
        # page number queue
        self.pageQueue = pageQueue
        # page content queue
        self.htmlQueue = htmlQueue

    def run(self):
        print('Starting fetcher thread %s' % self.threadName)
        while not Fetch_Exit:
            """从pageQueue中取出一个页码

            注意：队列get方法有一个block参数，默认为True，此时若队列为空，run过程不会结束，而是进入阻塞状态，等待队列有新的数据；
            如果block = False，队列为空时，就弹出Empty异常。

            """
            try:
                page = self.pageQueue.get(block = False)
                url = 'https://tieba.baidu.com/f?kw=%E7%BD%91%E7%BB%9C%E7%88%AC%E8%99%AB&ie=utf-8&pn='+ str((page-1)*50) + '/'
                headers = {
                    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.2743.116 Safari/537.36 ',
                    'Accept-Language': 'zh-CN,zh;q=0.8'}
                print('    baidutieba_crawler = %s, page = %s' % (self.threadName,page))

                r = requests.get(url,headers = headers)
                r.raise_for_status()
                r.encoding = r.apparent_encoding
                self.htmlQueue.put(r.text)
            except requests.RequestException as e:
                #print(e)#'requests.RequestException raised.')
                pass
            except queue.Empty as e:
                pass #print("  pageQueue is empty.")

            except :
                print('    Exception raised in fetcher thread %s.' % self.threadName)

        print('Exiting fetcher thread %s' % self.threadName)


class Parser(threading.Thread):
    """
    内容解析类
    """
    def __init__(self,threadName,htmlQueue,lock,filename):

        super(Parser,self).__init__()
        self.threadName = threadName
        # 解析内容队列
```

```
        self.htmlQueue = htmlQueue
        #self.lock = lock
        # 保存数据的文件名
        self.filename = filename

        self.lock = lock

    def run(self):
        print('Starting parser thread %s ...' % self.threadName)
        global Parse_Exit
        while not Parse_Exit:
            try:
                html = self.htmlQueue.get(False)# 若队列空则报异常
                if not html:
                    pass

                bsobj = BeautifulSoup(html,'html.parser')
                str = bsobj.find("title") + str(page)
                #在完成一项工作之后，Queue.task_done()函数向任务已经完成的队列发送一个信号
                self.queue.task_done()

                with self.lock:
                    with open(filename,'a') as f:
                        f.write(str)
            except Exception as e:
                print(e)
        print('Exiting parser thread', self.threadName)

Fetch_Exit = False
Parse_Exit = False
lock = threading.Lock()

def main():
    """
    pageQueue : 用于存放待爬取到的页数（整数值）
    htmlQueue : 用于存储网页内容（html文本）
    """
    #初始化网页页码page从1-10个页面
    pageQueue = queue.Queue(10)
    for page in range(1, 11):
        pageQueue.put(page)

    htmlQueue = queue.Queue()

    # 初始化采集线程的名字，以方便我们观察和理解
    fetcherlist = ["crawl-1", "crawl-2", "crawl-3"]
    fetcherthreads = []
    #依次启动3个Fetcher线程
    for threadName in fetcherlist:
        thread = Fecther(threadName, pageQueue,htmlQueue)
        thread.start() # 启动线程，对应类的run方法
        time.sleep(1)
        fetcherthreads.append(thread)

    #初始化解析线程parserList
    parserthreads = []
    parserList = ["parser-1", "parser-2", "parser-3"]
    global lock
    #分别启动parserList
```

```
    with open('tiebabaidu.txt','a') as outputf:
        for threadName in parserList:
            thread = Parser(threadName, htmlQueue, lock, outputf)
            thread.start()
            parserthreads.append(thread)

    # 不为空表示需要继续处理,为空后说明处理完毕，可以向后执行。
    while not pageQueue.empty():
        pass
    # 阻塞Fetcher 线程，等待所有线程完成
    for t in fetcherthreads:
        #print('%s is joined' % t)
        t.join()


    while not dataQueue.empty():
        pass
    # 增加阻塞，为了等待parser线程完成任务。
    for t in parserthreads:
        t.join()


    # 通知fetcher线程退出
    global Fetch_Exit
    Fetch_Exit = True
    # 通知parser线程退处
    global Parse_Exit
    Parse_Exit = True



    print( "Exiting Main Thread")
    with lock:
        output.close()


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