EDU

CSDN学院　IT实战派

多线程爬虫

讲师：黄老师

本章目标

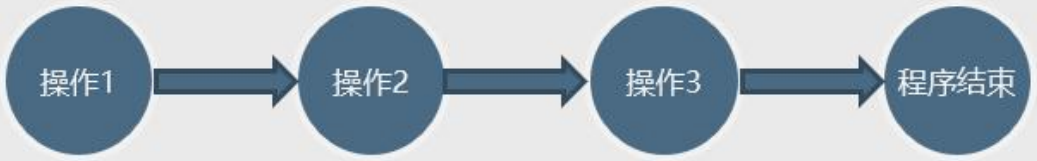
1. 理解多线程。
2. 掌握threading模块的使用。
3. 掌握生产者和消费者模式。
4. 理解GIL。
5. 能用多线程写爬虫。

本节目标

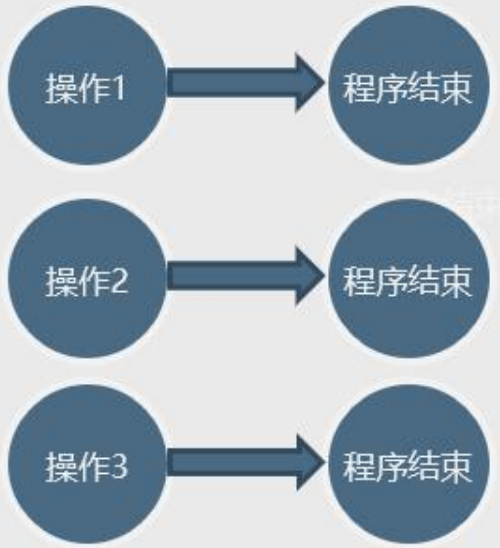
1. 了解什么是多线程
2. 学会使用threading模块基本使用

什么是多线程

1. 理解：默认情况下，一个程序只有一个进程和一个线程，代码是依次线性执行的。而多线程则可以并发执行，一次性多个人做多件事，自然比单线程更快。
2. 官方：https://baike.baidu.com/item/多线程/1190404?fr=aladdin



单线程执行方式



多线程执行方式

用threading模块写一个多线程程序

threading模块是python中专门提供用来做多线程编程的模块。threading模块中最常用的类是Thread。以下看一个简单的多线程程序：

```python
import time

def coding():
    for x in range(3):
        print('%s正在写代码' % x)
        time.sleep(1)

def drawing():
    for x in range(3):
        print('%s正在画图' % x)
        time.sleep(1)


def single_thread():
    coding()
    drawing()

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

单线程执行

```python
import threading
import time

def coding():
    for x in range(3):
        print('%s正在写代码' % x)
        time.sleep(1)

def drawing():
    for x in range(3):
        print('%s正在画图' % x)
        time.sleep(1)


def single_thread():
    coding()
    drawing()

def multi_thread():
    t1 = threading.Thread(target=coding)
    t2 = threading.Thread(target=drawing)

    t1.start()
    t2.start()

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

多线程执行

继承自threading.Thread类

首先来了解两个小知识点：

1. 使用threading.current_thread()可以看到当前线程的信息。
2. 使用threading.enumerate()函数便可以看到当前的线程。

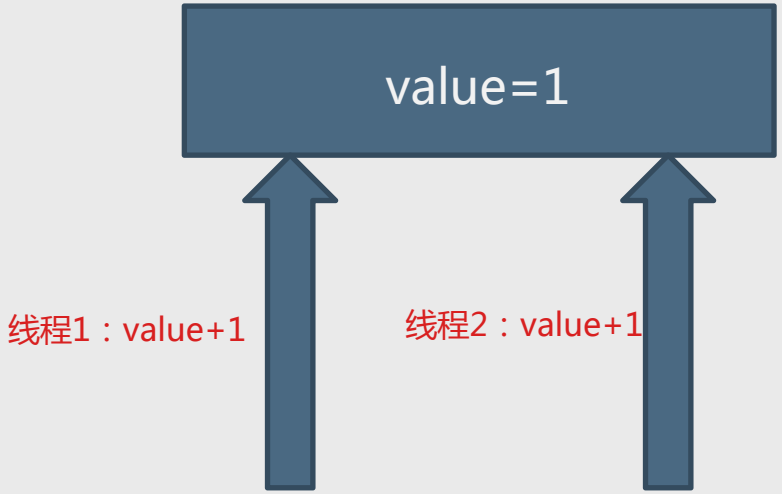
为了让线程代码更好的封装。可以使用threading模块下的Thread类，继承自这个类，然后实现run方法，线程就会自动运行run方法中的代码。示例代码如下：

```
import threading
import time

class CodingThread(threading.Thread):
    def run(self):
        for x in range(3):
            print('%s正在写代码' % threading.current_thread())
            time.sleep(1)

class DrawingThread(threading.Thread):
    def run(self):
        for x in range(3):
            print('%s正在画图' % threading.current_thread())
            time.sleep(1)

def multi_thread():
    t1 = CodingThread()
    t2 = DrawingThread()

    t1.start()
    t2.start()

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

多线程共享全局变量的问题

多线程都是在同一个进程中运行的。因此在进程中的全局变量所有线程都是可共享的。这就造成了一个问题，因为线程执行的顺序是无序的。有可能会造成数据错误。比如以下代码：

```python
import threading

tickets = 0

def get_ticket():
    global tickets
    for x in range(1000000):
        tickets += 1
    print('tickets:%d'%tickets)

def main():
    for x in range(2):
        t = threading.Thread(target=get_ticket)
        t.start()

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



以上结果正常来讲应该是1000000和2000000，但是因为多线程运行的不确定性。因此最后的结果可能是随机的。

锁机制和threading.Lock类

为了解决共享全局变量的问题。threading提供了一个Lock类，这个类可以在某个线程访问某个变量的时候加锁，其他线程此时就不能进来，直到当前线程处理完后，把锁释放了，其他线程才能进来处理。示例代码如下：

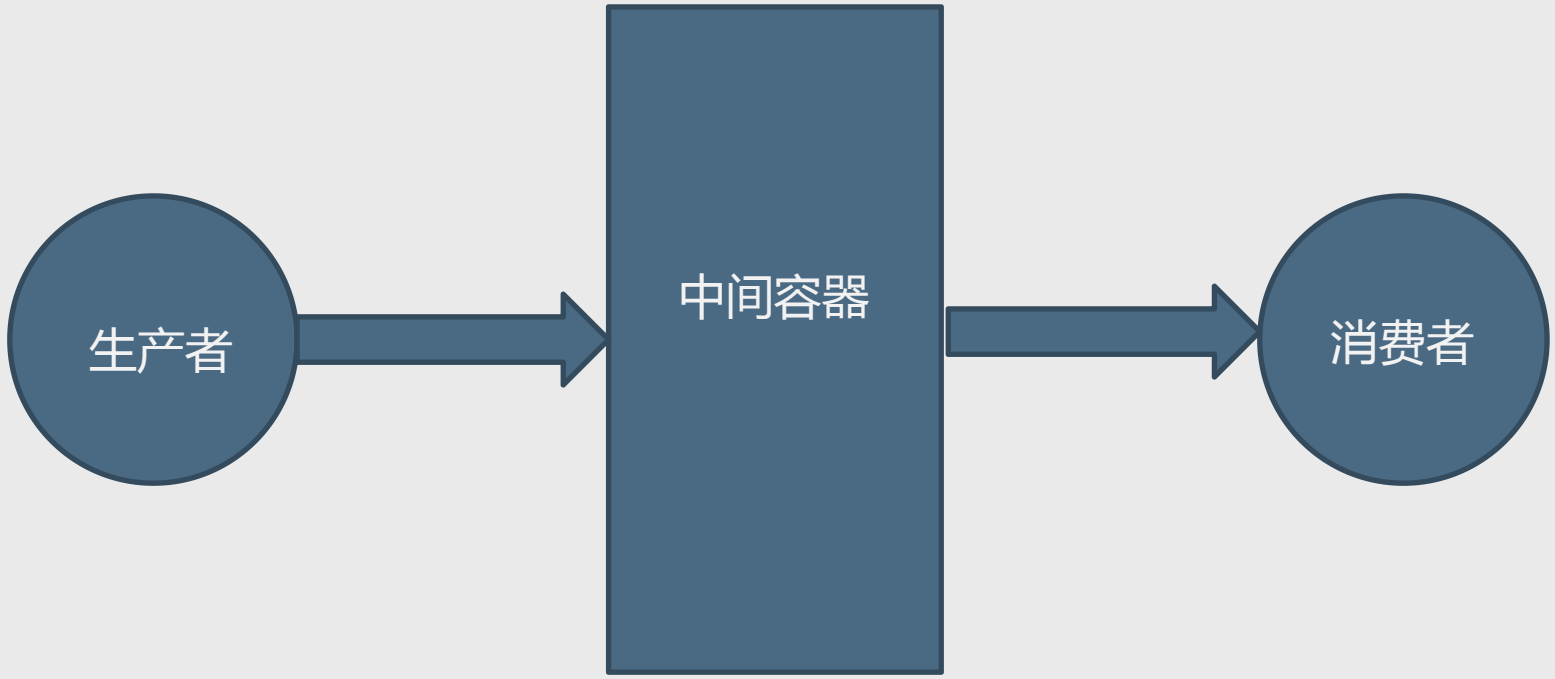
```python
import threading

VALUE = 0

gLock = threading.Lock()

def add_value():
    global VALUE
    gLock.acquire()  # 加锁
    for x in range(1000000):
        VALUE += 1
    gLock.release()  # 释放锁
    print('value : %d'%VALUE)

def main():
    for x in range(2):
        t = threading.Thread(target=add_value)
        t.start()

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

使用锁的原则：

1. **把尽量少的和不耗时的代码放到锁中执行。**
2. **代码执行完成后要记得释放锁。**

生产者和消费者模式

生产者和消费者模式是多线程开发中经常见到的一种模式。生产者的线程专门用来生产一些数据，然后存放到一个中间的变量中。消费者再从这个中间的变量中取出数据进行消费。通过生产者和消费者模式，可以让代码达到高内聚低耦合的目标，程序分工更加明确，线程更加方便管理。



Lock版的生产者和消费者模式

生产者和消费者因为要使用中间变量，中间变量经常是一些全局变量，因此需要使用锁来保证数据完整性。以下是使用threading.Lock锁实现的“生产者与消费者模式”的一个例子：

```python
import threading
import random
import time

gMoney = 1000
gLock = threading.Lock()
# 记录生产者生产的次数，达到10次就不再生产
gTimes = 0

class Producer(threading.Thread):
    def run(self):
        global gMoney
        global gLock
        global gTimes
        while True:
            money = random.randint(100, 1000)
            gLock.acquire()
            # 如果已经达到10次了，就不再生产了
            if gTimes >= 10:
                gLock.release()
                break
            gMoney += money
            print('%s当前存入%s元钱，剩余%s元钱' % (threading.current_thread(), money, gMoney))
            gTimes += 1
            time.sleep(0.5)
            gLock.release()

class Consumer(threading.Thread):
    def run(self):
        global gMoney
        global gLock
        global gTimes
        while True:
            money = random.randint(100, 500)
            gLock.acquire()
            if gMoney > money:
                gMoney -= money
                print('%s当前取出%s元钱，剩余%s元钱' % (threading.current_thread(), money, gMoney))
                time.sleep(0.5)
            else:
                # 如果钱不够了，有可能是已经超过了次数，这时候就判断一下
                if gTimes >= 10:
                    gLock.release()
                    break
                print("%s当前想取%s元钱，剩余%s元钱，不足！" % (threading.current_thread(),money,gMoney))
            gLock.release()

def main():
    for x in range(5):
        Consumer(name='消费者线程%d'%x).start()

    for x in range(5):
        Producer(name='生产者线程%d'%x).start()

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

Condition版的生产者和消费者模式

Lock版本的生产者与消费者模式可以正常的运行。但是存在一个不足，在消费者中，总是通过while True死循环并且上锁的方式去判断钱够不够。上锁是一个很耗费CPU资源的行为。因此这种方式不是最好的。还有一种更好的方式便是使用threading.Condition来实现。threading.Condition可以在没有数据的时候处于阻塞等待状态。一旦有合适的数据了，还可以使用notify相关的函数来通知其他处于等待状态的线程。这样就可以不用做一些无用的上锁和解锁的操作。可以提高程序的性能。首先对threading.Condition相关的函数做个介绍，threading.Condition类似threading.Lock，可以在修改全局数据的时候进行上锁，也可以在修改完毕后进行解锁。以下将一些常用的函数做个简单的介绍：

1. acquire：上锁。
2. release：解锁。
3. wait：将当前线程处于等待状态，并且会释放锁。可以被其他线程使用notify和notify_all函数唤醒。被唤醒后会继续等待上锁，上锁后继续执行下面的代码。
4. notify：通知某个正在等待的线程，默认是第1个等待的线程。
5. notify_all：通知所有正在等待的线程。notify和notify_all不会释放锁。并且需要在release之前调用。

Condition版的生产者和消费者模式


```python
import threading
import random
import time

gMoney = 1000
gCondition = threading.Condition()
gTimes = 0
gTotalTimes = 5

class Producer(threading.Thread):
    def run(self):
        global gMoney
        global gCondition
        global gTimes
        while True:
            money = random.randint(100, 1000)
            gCondition.acquire()
            if gTimes >= gTotalTimes:
                gCondition.release()
                print('当前生产者总共生产了%s次'%gTimes)
                break
            gMoney += money
            print('%s当前存入%s元钱，剩余%s元钱' % (threading.current_thread(), money, gMoney))
            gTimes += 1
            time.sleep(0.5)
            gCondition.notify_all()
            gCondition.release()

class Consumer(threading.Thread):
    def run(self):
        global gMoney
        global gCondition
        while True:
            money = random.randint(100, 500)
            gCondition.acquire()
            # 这里要给个while循环判断，因为等轮到这个线程的时候
            # 条件有可能又不满足了
            while gMoney < money:
                if gTimes >= gTotalTimes:
                    gCondition.release()
                    return
                print('%s准备取%s元钱，剩余%s元钱，不足！'%(threading.current_thread(),money,gMoney))
                gCondition.wait()
            gMoney -= money
            print('%s当前取出%s元钱，剩余%s元钱' % (threading.current_thread(), money, gMoney))
            time.sleep(0.5)
            gCondition.release()

def main():
    for x in range(5):
        Consumer(name='消费者线程%d'%x).start()

    for x in range(2):
        Producer(name='生产者线程%d'%x).start()

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

Queue线程安全队列

在线程中，访问一些全局变量，加锁是一个经常的过程。如果你是想把一些数据存储到某个队列中，那么Python内置了一个线程安全的模块叫做queue模块。Python中的queue模块中提供了同步的、线程安全的队列类，包括FIFO（先进先出）队列Queue，LIFO（后入先出）队列LifoQueue。这些队列都实现了锁原语（可以理解为原子操作，即要么不做，要么都做完），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。相关的函数如下：

初始化Queue(maxsize)：创建一个先进先出的队列。

1. qsize()：返回队列的大小。
2. empty()：判断队列是否为空。
3. full()：判断队列是否满了。
4. get()：从队列中取最后一个数据。
5. put()：将一个数据放到队列中。

多线程下载王者荣耀高清壁纸

1. 网址：https://pvp.qq.com/web201605/wallpaper.shtml
2. 真正获取壁纸的网址：https://apps.game.qq.com/cgi-bin/ams/module/ishow/V1.0/query/workList_inc.cgi?activityId=2735&sVerifyCode=ABCD&sDataType=JSON&iListNum=20&totalpage=0&page=0&iOrder=0&iSortNumClose=1&jsoncallback=jQuery171095256804866647 83_1554453019020&iAMSActivityId=51991&_everyRead=true&iTypeId=2&iFlowId=267733&iActId=2735&iModuleId=2735&_=1554453019266
3. 其中的page，代表的是第几页。
4. 用多线程，生产者和消费者模式，以及多线程安全的队列Queue来实现。

多线程的GIL锁

Python自带的解释器是CPython。CPython解释器的多线程实际上是一个假的多线程（在多核CPU中，只能利用一核，不能利用多核）。同一时刻只有一个线程在执行，为了保证同一时刻只有一个线程在执行，在CPython解释器中有一个东西叫做GIL（Global Intepreter Lock），叫做全局解释器锁。这个解释器锁是有必要的。因为CPython解释器的内存管理不是线程安全的。当然除了CPython解释器，还有其他的解释器，有些解释器是没有GIL锁的，见下面：

1. Jython：用Java实现的Python解释器。不存在GIL锁。更多详情请见：
   https://zh.wikipedia.org/wiki/Jython
2. IronPython：用.net实现的Python解释器。不存在GIL锁。更多详情请见：
   https://zh.wikipedia.org/wiki/IronPython
3. PyPy：用Python实现的Python解释器。存在GIL锁。更多详情请见：
   https://zh.wikipedia.org/wiki/PyPy

GIL虽然是一个假的多线程。但是在处理一些IO操作（比如文件读写和网络请求）还是可以在很大程度上提高效率的。在IO操作上建议使用多线程提高效率。在一些CPU计算操作上不建议使用多线程，而建议使用多进程。

多线程的GIL锁

A=1

作业

1. 用多线程的方式爬取“百思不得姐”段子作业。
2. 网址：http://www.budejie.com/text/
3. 爬取下来后需要用csv保存下来。

EDU

CSDN学院　IT实战派