寻找自我的博客

python爬虫抓站的总结

分类： Python 2012-08-22 22:41 337人阅读 评论(0) 收藏 举报

1.最基本的抓站

```
import urllib2
content = urllib2.urlopen('http://XXXX').read()
```

2.使用代理服务器

这在某些情况下比较有用，比如IP被封了，或者比如IP访问的次数受到限制等等。

```
import urllib2
proxy support = urllib2.ProxyHandler({'http':'http://XX.XX.XX.XX:XXXX'})
opener = urllib2.build opener(proxy support, urllib2.HTTPHandler)
urllib2.install opener(opener)
content = urllib2.urlopen('http://XXXX').read()
```

3.需要登录的情况

登录的情况比较麻烦我把问题拆分一下：

3.1 cookie的处理

```
import urllib2, cookielib
cookie_support= urllib2.HTTPCookieProcessor(cookielib.CookieJar())
opener = urllib2.build_opener(cookie_support, urllib2.HTTPHandler)
urllib2.install_opener(opener)
content = urllib2.urlopen('http://XXXX').read()
```

是的没错，如果想同时用代理和cookie，那就加入proxy_support然后operner改为

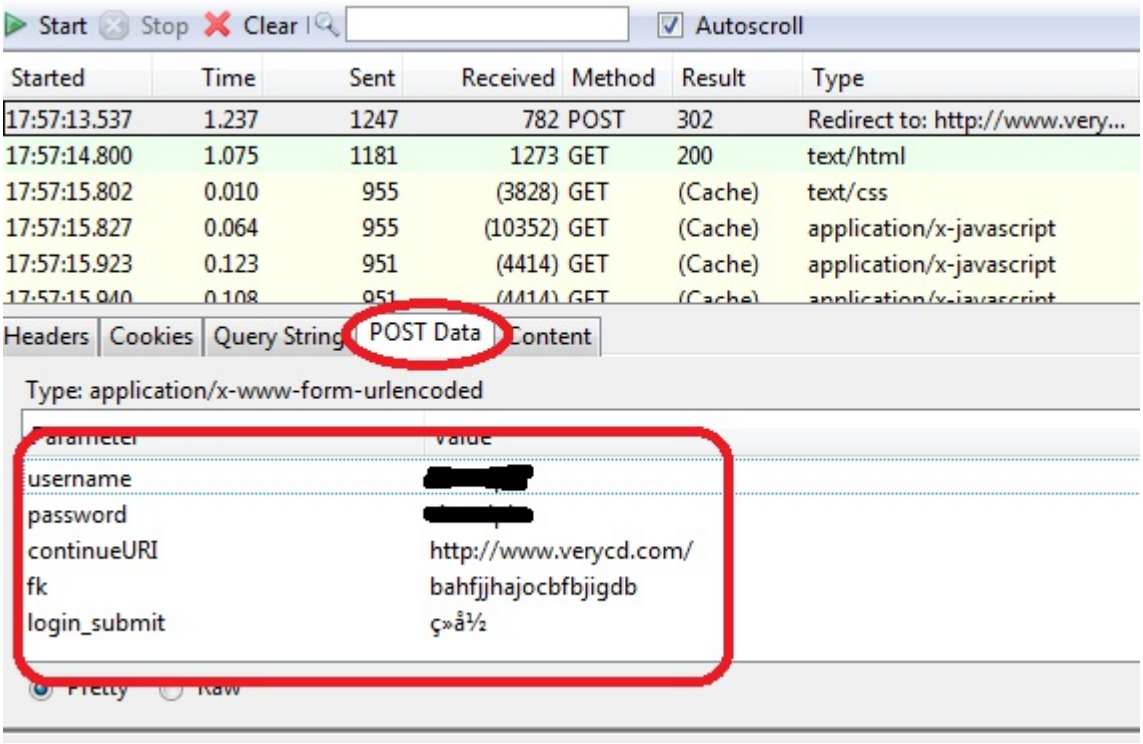
```
opener = urllib2.build_opener(proxy_support, cookie_support, urllib2.HTTPHandler)
```

3.2 表单的处理

登录必要填表，表单怎么填？首先利用工具截取所要填表的内容。

比如我一般用firefox+httpfox插件来看看自己到底发送了些什么包

这个我就举个例子好了，以verycd为例，先找到自己发的POST请求，以及POST表单项：



可以看到verycd的话需要填username,password,continueURI,fk,login_submit这几项，其中fk是随机生成的（其实不太随机，看上去像是把epoch时间经过简单的编码生成的），需要从网页获取，也就是说得先访问一次网页，用正则表达式等工具截取返回数据中的fk项。continueURI顾名思义可以随便写，login_submit是固定的，这从源码可以看出。还有username，password那就很显然了。

好的，有了要填写的数据，我们就要生成postdata

```
import urllib
postdata=urllib.urlencode({
    'username':'XXXXX',
    'password':'XXXXX',
    'continueURI':'http://www.verycd.com/',
    'fk':fk,
    'login_submit':'登录'
})
```

然后生成http请求，再发送请求：

```
req = urllib2.Request(
    url = 'http://secure.verycd.com/signin/*/http://www.verycd.com/',
    data = postdata
)
result = urllib2.urlopen(req).read()
```

3.3 伪装成浏览器访问

某些网站反感爬虫的到访，于是对爬虫一律拒绝请求。这时候我们需要伪装成浏览器，这可以通过修改http包中的header来实现：

```
headers = {
    'User-Agent':'Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.1.6) Gec
}
req = urllib2.Request(
    url = 'http://secure.verycd.com/signin/*/http://www.verycd.com/',
    data = postdata,
    headers = headers
```

```
)
```

3.4 反"反盗链"

某些站点有所谓的反盗链设置，其实说穿了很简单，就是检查你发送请求的header里面，referer站点是不是他自己，所以我们只需要像3.3一样，把headers的referer改成该网站即可，以黑幕著称地cnbeta为例：

```
headers = {
    'Referer':'http://www.cnbeta.com/articles'
}
```

headers是一个dict数据结构，你可以放入任何想要的header，来做一些伪装。例如，有些自作聪明的网站总喜欢窥人隐私，别人通过代理访问，他偏偏要读取header中的X-Forwarded-For来看看人家的真实IP，没话说，那就直接把X-Forwarde-For改了吧，可以改成随便什么好玩的东东来欺负欺负他，呵呵。

3.5 终极绝招

有时候即使做了3.1-3.4，访问还是会被据，那么没办法，老老实实把httpfox中看到的headers全都写上，那一般也就行了。 再不行，那就只能用终极绝招了，selenium直接控制浏览器来进行访问，只要浏览器可以做到的，那么它也可以做到。类似的还有pamie，watir，等等等等。

4.多线程并发抓取

单线程太慢的话，就需要多线程了，这里给个简单的线程池模板 这个程序只是简单地打印了1-10，但是可以看出是并发地。

```
from threading import Thread
from Queue import Queue
from time import sleep
#q是任务队列
#NUM是并发线程总数
#JOBS是有多少任务
q = Queue()
NUM = 2
JOBS = 10
#具体的处理函数，负责处理单个任务
def do somthing using(arguments):
    print arguments
#这个是工作进程，负责不断从队列取数据并处理
def working():
    while True:
        arguments = q.get()
        do somthing using(arguments)
        sleep(1)
        q.task done()
#fork NUM个线程等待队列
for i in range(NUM):
    t = Thread(target=working)
    t.setDaemon(True)
    t.start()
#把JOBS排入队列
for i in range(JOBS):
    q.put(i)
#等待所有JOBS完成
q.join()
```

5.验证码的处理

碰到验证码咋办？这里分两种情况处理：

- google那种验证码，凉拌
- 简单的验证码：字符个数有限，只使用了简单的平移或旋转加噪音而没有扭曲的，这种还是有可能可以处理的，一般思路是旋转的转回来，噪音去掉，然后划分单个字符，划分好了以后再通过特征提取的方法(例如PCA)降维并生成特征库，然后把验证码和特征库进行比较。这个比较复杂，一篇博文是说不完的，这里就不展开了，具体做法请弄本相关教科书好好研究一下。
- 事实上有些验证码还是很弱的，这里就不点名了，反正我通过2的方法提取过准确度非常高的验证码，所以2事实上是可行的。

6 gzip/deflate支持

现在的网页普遍支持gzip压缩，这往往可以解决大量传输时间，以VeryCD的主页为例，未压缩版本247K，压缩了以后45K，为原来的1/5。这就意味着抓取速度会快5倍。

然而python的urllib/urllib2默认都不支持压缩，要返回压缩格式，必须在request的header里面写明'accept-encoding'，然后读取response后更要检查header查看是否有'content-encoding'一项来判断是否需要解码，很繁琐琐碎。如何让urllib2自动支持gzip, defalte呢？

其实可以继承BaseHanlder类，然后build_opener的方式来处理：

```
import urllib2
from gzip import GzipFile
from StringIO import StringIO
class ContentEncodingProcessor(urllib2.BaseHandler):
  """A handler to add gzip capabilities to urllib2 requests """

  # add headers to requests
  def http_request(self, req):
    req.add_header("Accept-Encoding", "gzip, deflate")
    return req

  # decode
  def http_response(self, req, resp):
    old_resp = resp
    # gzip
    if resp.headers.get("content-encoding") == "gzip":
        gz = GzipFile(
                    fileobj=StringIO(resp.read()),
                    mode="r"
                  )
        resp = urllib2.addinfourl(gz, old_resp.headers, old_resp.url, old_resp.code)
        resp.msg = old_resp.msg
    # deflate
    if resp.headers.get("content-encoding") == "deflate":
        gz = StringIO( deflate(resp.read()) )
        resp = urllib2.addinfourl(gz, old_resp.headers, old_resp.url, old_resp.code)
        resp.msg = old_resp.msg
    return resp

# deflate support
import zlib
def deflate(data):   # zlib only provides the zlib compress format, not the deflate
  try:               # so on top of all there's this workaround:
    return zlib.decompress(data, -zlib.MAX_WBITS)
  except zlib.error:
    return zlib.decompress(data)
```

然后就简单了，

```
encoding_support = ContentEncodingProcessor
```

```
opener = urllib2.build opener( encoding support, urllib2.HTTPHandler )

#直接用opener打开网页，如果服务器支持gzip/defalte则自动解压缩
content = opener.open(url).read()
```

7. 更方便地多线程

总结一文的确提及了一个简单的多线程模板，但是那个东东真正应用到程序里面去只会让程序变得支离破碎，不堪入目。在怎么更方便地进行多线程方面我也动了一番脑筋。先想想怎么进行多线程调用最方便呢？

1、用twisted进行异步I/O抓取

事实上更高效的抓取并非一定要用多线程，也可以使用异步I/O法：直接用twisted的getPage方法，然后分别加上异步I/O结束时的callback和errback方法即可。例如可以这么干：

```
from twisted.web.client import getPage
from twisted.internet import reactor

links = [ 'http://www.verycd.com/topics/%d/'%i for i in range(5420,5430) ]

def parse page(data,url):
    print len(data),url

def fetch error(error,url):
    print error.getErrorMessage(),url

# 批量抓取链接
for url in links:
    getPage(url,timeout=5) \
        .addCallback(parse_page,url) \ #成功则调用parse_page方法
        .addErrback(fetch_error,url)      #失败则调用fetch_error方法

reactor.callLater(5, reactor.stop) #5秒钟后通知reactor结束程序
reactor.run()
```

twisted人如其名，写的代码实在是太扭曲了，非正常人所能接受，虽然这个简单的例子看上去还好；每次写twisted的程序整个人都扭曲了，累得不得了，文档等于没有，必须得看源码才知道怎么整，唉不提了。

如果要支持gzip/deflate，甚至做一些登陆的扩展，就得为twisted写个新的HTTPClientFactory类诸如此类，我这眉头真是大皱，遂放弃。有毅力者请自行尝试。

这篇讲怎么用twisted来进行批量网址处理的文章不错，由浅入深，深入浅出，可以一看。

2、设计一个简单的多线程抓取类

还是觉得在urllib之类python"本土"的东东里面折腾起来更舒服。试想一下，如果有个Fetcher类，你可以这么调用

```
f = Fetcher(threads=10) #设定下载线程数为10
for url in urls:
    f.push(url)  #把所有url推入下载队列
while f.taskleft(): #若还有未完成下载的线程
    content = f.pop()  #从下载完成队列中取出结果
    do_with(content) # 处理content内容
```

这么个多线程调用简单明了，那么就这么设计吧，首先要有两个队列，用Queue搞定，多线程的基本架构也和"技巧总结"一文类似，push方法和pop方法都比较好处理，都是直接用Queue的方法，taskleft则是如果有"正在运行的任务"或者"队列中的任务"则为是，也好办，于是代码如下：

```
import urllib2
from threading import Thread,Lock
from Queue import Queue
import time

class Fetcher:
    def __init__(self,threads):
        self.opener = urllib2.build_opener(urllib2.HTTPHandler)
        self.lock = Lock() #线程锁
        self.q_req = Queue() #任务队列
        self.q_ans = Queue() #完成队列
        self.threads = threads
        for i in range(threads):
            t = Thread(target=self.threadget)
            t.setDaemon(True)
            t.start()
        self.running = 0

    def __del__(self): #解构时需等待两个队列完成
        time.sleep(0.5)
        self.q_req.join()
        self.q_ans.join()

    def taskleft(self):
        return self.q_req.qsize()+self.q_ans.qsize()+self.running

    def push(self,req):
        self.q_req.put(req)

    def pop(self):
        return self.q_ans.get()

    def threadget(self):
        while True:
            req = self.q_req.get()
            with self.lock: #要保证该操作的原子性，进入critical area
                self.running += 1
            try:
                ans = self.opener.open(req).read()
            except Exception, what:
                ans = ''
                print what
            self.q_ans.put((req,ans))
            with self.lock:
                self.running -= 1
            self.q_req.task_done()
            time.sleep(0.1) # don't spam

if __name__ == "__main__":
    links = [ 'http://www.verycd.com/topics/%d/'%i for i in range(5420,5430) ]
    f = Fetcher(threads=10)
    for url in links:
        f.push(url)
    while f.taskleft():
        url,content = f.pop()
        print url,len(content)
```

8. 一些琐碎的经验

1、连接池：

opener.open和urllib2.urlopen一样，都会新建一个http请求。通常情况下这不是什么问题，因为线性环境下，一秒钟可能也就新生成一个请求；然而在多线程环境下，每秒钟可以是几十上百个请求，这么干只要几分钟，正常的有理智的服务器一定会封禁你的。

然而在正常的html请求时，保持同时和服务器几十个连接又是很正常的一件事，所以完全可以手动维护一个HttpConnection的池，然后每次抓取时从连接池里面选连接进行连接即可。

这里有一个取巧的方法，就是利用squid做代理服务器来进行抓取，则squid会自动为你维护连接池，还附带数据缓存功能，而且squid本来就是我每个服务器上面必装的东东，何必再自找麻烦写连接池呢。

2、设定线程的栈大小

栈大小的设定将非常显著地影响python的内存占用，python多线程不设置这个值会导致程序占用大量内存，这对openvz的vps来说非常致命。stack_size必须大于32768，实际上应该总要32768*2以上

```
from threading import stack size
stack_size(32768*16)
```

3、设置失败后自动重试

```
    def get(self,req,retries=3):
        try:
            response = self.opener.open(req)
            data = response.read()
        except Exception , what:
            print what,req
            if retries>0:
                return self.get(req,retries-1)
            else:
                print 'GET Failed',req
                return ''
        return data
```

4、设置超时

```
    import socket
    socket.setdefaulttimeout(10) #设置10秒后连接超时
```

5、登陆

登陆更加简化了，首先build_opener中要加入cookie支持，参考“总结”一文；如要登陆VeryCD，给Fetcher新增一个空方法login，并在init()中调用，然后继承Fetcher类并override login方法：

```
def login(self,username,password):
    import urllib
    data=urllib.urlencode({'username':username,
                           'password':password,
                           'continue':'http://www.verycd.com/',
                           'login_submit':u'登录'.encode('utf-8'),
                           'save_cookie':1,})
    url = 'http://www.verycd.com/signin'
    self.opener.open(url,data).read()
```

于是在Fetcher初始化时便会自动登录VeryCD网站。

9. 总结

如此，把上述所有小技巧都糅合起来就和我目前的私藏最终版的Fetcher类相差不远了，它支持多线程，gzip/deflate压缩，超时设置，自动重试，设置栈大小，自动登录等功能；代码简单，使用方便，性能也不俗，可谓居家旅行，杀人放火，咳咳，之必备工具。

之所以说和最终版差得不远，是因为最终版还有一个保留功能“马甲术”：多代理自动选择。看起来好像仅仅是一个random.choice的区别，其实包含了代理获取，代理验证，代理测速等诸多环节，这就是另一个故事了。