林禹臣 5140309507

October 6, 2015

电类工程导论C 实验报告2&3

——crawler

目录页

1. 引言（Introduction）

在lab2里，首先我学习了get请求和post请求的模拟方法，知道了如何利用浏览器开发工具去查找到请求内容。接下来根据bfs和dfs两大原理编写了一个简单的爬虫程序。

lab3中，我主要学会了如何更好的去优化我的爬虫程序，尤其是效率上的高级优化，比如利用布隆过滤器和多线程并发编程等等。

衷心感谢张娅老师，姚助教和王助教的耐心指导和辛勤付出，让我获益匪浅。

2. 实验环境 （Testbed）

python 2.7.10（narrow）+ Sublime Text 3 + Mac OS X 10.11

(提示：5.2节中会提到，Python中time.clock()函数以及time.time()函数在windows系统和unix系系统的不同之处。)

3. 预备知识 （Propaedeutics）

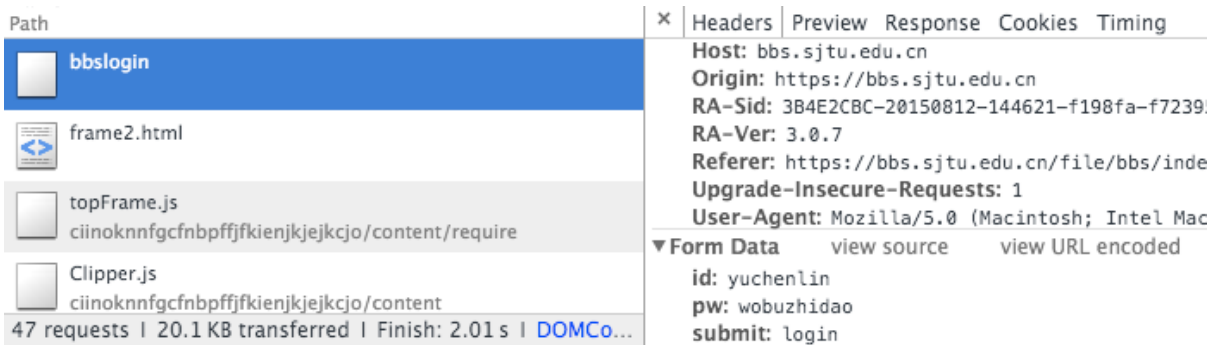
- HTTP请求的两种方式之间的区别，以及如何利用浏览器的开发者工具去寻找请求的内容，并在python中模拟提交请求
- 网络爬虫基本原理
- 布隆过滤器基本原理
- Python中线程的概念，以及基本的多线程处理方法

4. 实验过程（Experiment）

4.1. lab1 模拟Post请求 （Post Request）

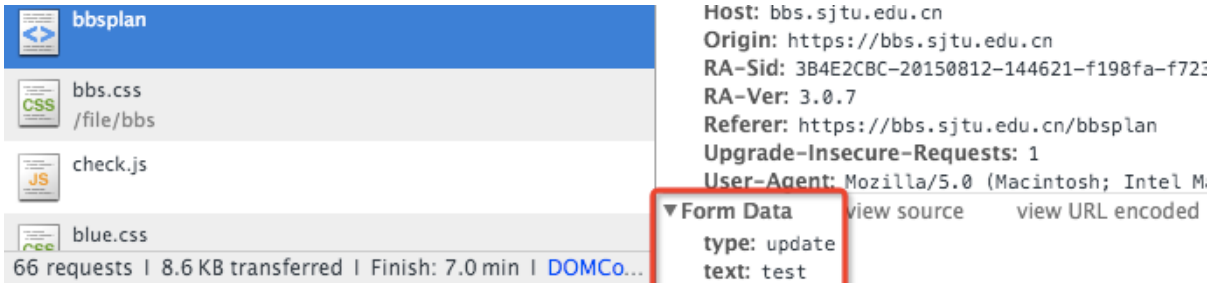
实验目的：模拟post请求登录交大BBS，然后再模拟一个请求修改说明档（bbsplan）。

模拟post请求的第一步首先我们要利用浏览器的开发者模式中Network界面找到我们post的data是什么，如图所示：



模拟登入之后，我们发现这个form data 有三项，id，pw和submit。其中id和pw就是用户名密码，而submit是固定值login。

我们再继续模拟修改说明档，找到form data。




```
def bbsSet(id, pwd , text):
    cj = cookielib.CookieJar()
    opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
    urllib2.install_opener(opener)
    postdata = urllib.urlencode({'id':'yuchenlin','pw':'wobuzhidao','submit':'login'})
    req = urllib2.Request(url="https://bbs.sjtu.edu.cn/bbslogin",data=postdata)#this is
    response = urllib2.urlopen(req) #login

    toUpdate = urllib.urlencode({'type':'update','text':text.encode('gb2312')})
    #encode
    req = urllib2.Request(url='https://bbs.sjtu.edu.cn/bbsplan',data=toUpdate)
    response = url  lib2.urlopen(req)
    content = urllib2.urlopen('https://bbs.sjtu.edu.cn/bbsplan').read()
    bs = BeautifulSoup(content)
    print bs.find('textarea').string.strip().decode('decode')
```

有了这两个信息之后我们就可以去模拟爬虫的信息了，代码如下。

两个postdata的key和value都是由我们之前在浏览器开发者工具里面获取到的，这里注意的是我们一定要把提交的text设置为gb2312的编码模式才可以，因为这是被指定的，虽然用gbk也没有报错，但是不保险。

```
<html>
<meta HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=gb2312" />
<link rel="shortcut icon" type="image/x-icon" href="favicon.ico" />
<script src="/js/jquery.js"></script>
```

剩下的步骤就很简单了，和正常的访问差不多，只不过多了CookieJar的使用以及Request函数的参数。

4.2. lab1 图论基础 （Graph Theory）

实验目的：实现bfs的搜索函数以及图结构的重建。

```
def union_bfs(a,b):
    for e in b:
        if e not in a:
            a.insert(0,e)
```

这个函数很简单，把b里的元素都插在a之前，插的时候要注意排除重复。我们当然有更好的做法，不过这个做法确实是最显而易见容易懂的。由于我们的效率瓶颈并不集中在此处，故先这样简单的处理了。

返回搜索过程中产生的图结构只要在搜索的过程中记录一下即可。

```
if page not in crawled:
    content = get_page(page)
    outlinks = get_all_links(content)
    graph[page] = outlinks
    globals()['union_%s' % method](tocrawl,
    crawled.append(page)
```

4.3. lab1 简单爬虫（Basic Crawler）

在前两个练习的基础上，我们稍加改动即可爬去真正的网页，这里值得一提的是get_all_links(content, page)这个函数，在lab1的代码中，我先采用了BeautifulSoup的parse模式，但是效率奇低，在lab2的并行爬虫那一节我采用了另一种方式，并且会在那一节里比较两种方式，这里就暂时不说细节问题了。

有一个需要注意的地方就是，在爬虫中如果加上timeout的设置时，一定要考虑一个问题那就是seed是不可以被timeout的设置所放弃爬取的，如果seed网页被放弃了，那么接下来将会少很多有价值的页面，所以我又在getPage的函数里加了一个参数来判断是否应该开启timeout设置。经测试，当timeout=3的时候效果比较好，当然这样取决于测试环境的网络环境。

另外值得一提的是getPage函数内部需要进行异常处理，对于404、502等等错误进行报错但是要继续爬取，所以当时应该返回一个None值，然后在接受方进行响应的跳过处理。（见ex4.py）

4.4. lab2 布隆过滤器（Bloom Filter）

实验目的是创建一个简单的布隆过滤器并且测试其 false positive rate。我选择的方案是建立一个类MyBloomFilter来封装这些过程。MyBloomFilter里我设置了这么一些属性和方法：

- size制定了的总的bit数目，也就是m
- k表示的是hash函数的个数，
- bitmap是我们的存储结构，它是一个Bitarray的对象。
- add方法是用来添加元素的。
- lookup是用来检查key是否存在的。

除此之外，我还建立了一个list，用来给hash函数传递不同的seed。（至于BKDRHash的seed为何要选成31之类的数目，在实验扩展中将会介绍）

类的代码如下：

```
class MyBloomFilter(object):
    """docstring for MyBloomFilter"""
    seeds = [31,7,15,127,131,1313,131311,1313113,1311113113,1731,71,177]
    def __init__(self, size,k = 8):
        self.bitmap = Bitarray(size)
        self.m = size
        self.k = k
    def add(self,key):  #增加key到我们的位图中
        #进行k次hash
        for i in range(self.k):
            hv = BKDRHash(key,MyBloomFilter.seeds[i],self.m)
            self.bitmap.set(hv)
    def lookup(self,key):
        for i in range(self.k):
            hv = BKDRHash(key,MyBloomFilter.seeds[i],self.m)
            if(not self.bitmap.get(hv)): return False
        return True
```

可以看到add函数其实就是进行了k次hash函数，分别把得到的hash值（模m后）对应的bit位设置为1，所以调用了bitmap的set函数。

lookup函数更加简单，同样对传进来的key进行k次hash取值，注意的是，一旦发现有其中某一次对应的bit位是0，那么就说明这个key肯定不存在，所以直接返回False避免造成多余的去调用hash函数。如果全都存在，那么我们就可以『认为』这个key是存在的。

下面要做的事就是测量我们的False Positive Rate （假阳性率）。我的测量方案是这样的，首先找到一个单词表（词数为n），然后我们产生大量的随机字符串（假设个数为 T ），找到其中满足不真的在单词表中，但是我们『认为』它在的，个数假设为 t 。

所以我们就有

$$P = t/T$$

按照这个思路，我们第一步要找到单词表，这里我选择的是GRE的单词表文本，经过处理（processGREwords.py）我把它转换为了全英文的gre.txt，其中有7500个单词，最短3个字母，最长的17个字母。所以我们生产的随机字符串的要求就是，长度在3到17之间，由a~z组成的字符串。代码如下：

（注：此处代码为单线程核心原理演示代码，在6.3节中会给出更完善且加入了多线程的测试代码，提交的源代码文件中保留的是多线程版本，与此处截图不同。）

```
mbf = MyBloomFilter(32000,3) # k: (m/n)ln(2)
for w in words: mbf.add(w)
falseCount = 0
for i in range(10000):
    ranLen = random.randint(minl,maxl+1)
    strr = string.join(random.sample(['z','y','x','w','v','u','t','s','r','q
        ,'j','i','h','g','f','e','d','c','b','a'], ranLen)).replace(' ','')
    if((strr not in words) and (mbf.lookup(strr))):
        falseCount += 1
print falseCount/10000.0
```

这里我们看到了判断条件很简单，随机字符串strr 不在 words中但是却符合我们的lookup函数，所以说明这是个False Positive的item，就让falseCount自增1。

那么如何降低布隆过滤器的False Positive Rate呢？根据概率论的推导（见扩展部分），当k，m，n满足

$$k = (m/n) * \ln 2$$

时，rate是最小的。为了测试方便，我选的数据量偏小，这里m=32000，n=7500，ln2 = 0.7 ，所以m/n = 4.2 估算出来 k = 3

经过多次实验测量，k取3的时候确实是最小的，rate约为0.11 和WISC给出下表基本相符

m/n	*k*	*k*=1	*k*=2	*k*=3	*k*=4	*k*=5	*k*=6	*k*=7	*k*=8
2	1.39	0.393	0.400						
3	2.08	0.283	0.237	0.253					
4	2.77	0.221	0.155	0.147	0.160				
5	3.46	0.181	0.109	0.092	0.092	0.101			
6	4.16	0.154	0.0804	0.0609	0.0561	0.0578	0.0638		
7	4.85	0.133	0.0618	0.0423	0.0359	0.0347	0.0364		
8	5.55	0.118	0.0489	0.0306	0.024	0.0217	0.0216	0.0229	

相对来说，我们按照一定规则提高m和k肯定是可以降低rate的，经过多次选取不同的m/n和k，实验得到的rate确实基本符合这个表格，侧面说明了我的布隆过滤器基本合格。要注意的是，在测试的时候不可避免要提高m和k，从而需要更大的测试量才可以，否则会导致因为测试数据规模较小，错误率一直为0的情况。也因此，提高数据规模会耗费更多的时间，所以要进行多线程优化，优化部分见**6.3节**。

4.5. lab2 并行爬虫 (Parallel Crawler)

实验目的是重写crawler，使之成为并行的爬虫程序，并且为了提高效率，要重写get_all_links 函数。

4.5.1. 关于并行。我一开始选择了其他更便捷的并行库来做实验，发现出现了很多问题没有办法一时解决，最后还是选择了用最通用的thread和Queue这种模式来进行操作，核心代码如下：（MultiThreadCrawler.py）

值得一提地方在于如何结束进程，这里其实有两种思路都可以进行对爬去页面个数的限制。

方案1：在while循环中每次get之后对已经爬取页面的个数进行判断，如果超过了阈值，就done掉当前的task，并直接continue，这样循环done掉所有的不需要的task就可以退出线程。但是这个方案有一个严重的缺点就是因为在put的时候并没有进行检查，多个线程会在临近阈值时还在大量的put新的page进去，导致最后的循环done的词数特别多，浪费了效率。

方案2：在get_all_links函数调用之后的那个for循环里，我们put之前对count进行控制，如果超过了阈值，我们就不再向q里put新的任务进去了。这样就能保证q里的任务不会过多，从根本上解决了问题。

综合方案1和方案2，我觉得方案2在性能上更优，所以选择了它。

完整的working函数代码如下：

```
def working():
    global count,crawled,max_page,q
    while count <= max_page:
        if (count==0 or q.qsize>0):
            page = q.get()
        else:
            break
        if not mbf.lookup(page):
            content = get_page(page)
            if(content!=None):

                if(count <= max_page):
                    print count,page
                    add_page_to_folder(page,content)
                    outlinks = get_all_links(content,page)
                    for link in outlinks:
                        if(q.qsize() + count > max_page):
                            break
                        q.put(link)
                    if varLock.acquire():
                        count += 1
                        graph[page] = outlinks
                        mbf.add(page)
                        varLock.release()
            if q.unfinished_tasks:
                q.task_done()
    while q.unfinished_tasks:
        if varLock.acquire():
            q.task_done()
            varLock.release()
```

注意：

1. q.get()之前一定要判断q是不是空的，否则会引起Empty异常，但是当count=0的时候，这时因为可能有多个线程在put到q中，所以此时要给予特权。

2. 由于是多线程的并发处理，所以我们还要在整个函数的结尾处加上一个while循环来不断的消除所有多余的任务。（这一步在我原本的代码中即使没有也是可以正常结束进程的，但是在帮其他同学debug的时候，发现这一步确实非常重要，很多同学的代码没有这一布就不能正常的终止进程。）

4.5.2.重写get_all_links。因为bs的效率太低了，我选择了手写的方式来获取url，最后证明效率提高了数倍。

思路很简单，是先用正则表达式findAll所有的a标签，然后进行容错处理，把javascript去掉，把以#开头的锚标记去掉，把相对地址补全。尤其是相对地址，由于lab1中用的正则表达式的规则，它只能find到以/开头的相对地址，但是还是有大量其他的相对地址没有处理，而现在是可以全部处理的。还有一点就是我是利用set的排异性，防止重复的连接过多，最后在把set转换为list。代码如下：

```
#without bs
def get_all_links(content, page):
    if content == None:
        return []
    import re
    urlset = set()
    urls = re.findall(r"<a.*?href=.*?<\/a>",content,re.I)
    for i in urls:
        u=""
        href = re.findall(r'href=".*?"',i,re.I)
        if(len(href)==1):
            url = href[0].split('"')[1]
            if(len(url)<5): continue #某些太短的无意义代码
            if('javascript' in url or url[0]=='#'): #js和锚标记
                continue
            elif(url[0:4]=='http'):#绝对地址
                u = url
            else:#相对地址
                u = urlparse.urljoin(page,url)
        if(u!=""):
            urlset.add(u)
    links = list(urlset)
    return links
```

下面是爬取并解析10个网页的效率（两者都加入了多线程）的差别：

```
2.4394929409
```
（Beautiful Soup的处理时间）

```
0.495404005051
```
（手动Parse的处理时间）

可以明显地看到效率的提升，如果爬去的网页更多的话，这个时间差将会更加明显。

5. 遇到的困难和解决方案（Problems & Solutions）

5.1. Python 2.7.9 + 中SSL的报错问题

问题是这样的，在2.7.10中urlopen一个关于bbs 的https的站点的时候出现了SSL证书验证不通过的事情。报错信息如下：urllib2.URLError: <urlopen error [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed (_ssl.c:590)>

经过查询，终于发现，在2.7.6下，打开https站点如果涉及证书问题，且对方站点是自签名的证书则直接通过。但是在2.7.9及之后的版本中，python加了一个新的特性，为了安全起见，遇到这种情况直接报错，认为不通过SSL验证。所以需要在全局变量中把SSL验证关闭。即加入这两行代码即可：

```
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
```

5.2. Python计时函数在不同系统下的区别

刚开始的时候，我也是和ppt一样，用两次time.clock（）的差值来表示时间，但是发现在多线程的时候，我的这个差值在不同的线程数下始终保持近似相等，但是通过个人观察和感觉，明显所用自然时间是不同的。

注意到time中还有个time()函数，于是我换用了它，发现这个是准确的，不同的线程个数，会有不同的时间。

查了下资料，这是因为clock在不同的系统中表示的是不一样的。在win32系统下，这个函数返回的是真实时间（wall time），而在Unix/Linux下返回的是CPU时间(process time)。因为Mac OS X是基于Unix的，所以出现了这个问题，其实应该用time.time()来做差就好了。

6. 实验扩展（Extensions）

6.1. 关于 k = (m/n) * ln2 的推导和布隆过滤器的理解

正好前两天刚刚买了《数学之美》（第二版），里面有一节就是关于布隆过滤器的。虽然内容浅短，但是给出了这个公式的证明。其实还是很好理解的，主要利用了概率学的基本知识，加上一定的估算就可以得到这个结果。也可以根据WISC的那个页面上的公式结合理解。总体上来说布隆过滤器的数学原理就是两个足够随机的数完全相等的可能性很低。如果空间和时间满足要求，可以加大m/n，同时提高k，将会让假阳性率非常非常小，是一个非常优美的数据结构。其实还可以把布隆过滤器理解成bitmap的高纬度扩展。（假象有多层bitmap在不同纬度重叠）

6.2. 关于 BKDRHash函数的seed选择问题

一开始并不知道为什么要去31,131,1311等等这样的数，所以查了一下资料。简单来说，我们可以把数字分成三类，一类是2的幂，一类是非2的幂的偶数，一类是奇数。经过数学推导就会发现，当选取的seed是2的幂的时候，如果字符串很长，则很快就会导致因为溢出(python没有这个问题)从而被抛弃高位的数，导致最后的hash值是一样的。而非2幂的偶数也同样会有类似的问题，只不过在不同的高度上出现而

已。所以只能是奇数，但是奇数中最好还是用31这种2的幂-1的奇数，也许是从计算效率的角度来讲的，不过对于今天的计算机而言并没有太大的意义，所以seed的选取只要是奇数就可以了，不必强求。

6.3. 关于 multiprocessing.dummy 中map函数的多线程问题

通过在网络上的学习，我找到了一个非常快捷的库来进行多线程处理，即dummy。一开始想用这个模块来进行多线程处理，因为非常简单。比如这个样例代码，就可以完成多线程的分布。

```
pool = ThreadPool(9)
results = pool.map(test, urls)
pool.close()
pool.join()
```

核心的代码就一行，就是调用pool的map函数，将test函数和一个list作为参数传入，它会把这个list中的每个元素分配到不同的线程下调用test函数，从而完成多线程。

但是当我实际使用它来进行爬虫的处理的时候发现了一个致命的问题就是当这个list是要在test函数里动态增删的时候，会发生bug。因为这个map函数虽然有一定的lock机制，但是在多线程操作的时候仍然无法协调好各线程的关系，而且无法灵活的加入线程停止条件。它更适合替代原有的for循环，因为for循环的任务列表是固定的，而爬虫这种迭代的任务列表套进去会有很多不兼容的bug。

既然只能替换for循环，我想到了可以在测试布隆过滤器错误率的时候，加入map函数的多线程，让整个测试过程变得更快捷，代码如下。（见TestMyBloomFilter.py）

```
import time
from multiprocessing import Pool
from multiprocessing.dummy import Pool as TPool

def work(i):
    global mbf
    ranLen = random.randint(minl,maxl+1)
    strr = string.join(random.sample(['z','y','x','w',
    return (strr not in words) and (mbf.lookup(strr))

start = time.time()

p = TPool(13)#设置线程池
res = p.map(work,range(1,amount))
p.close()
p.join()

falseCount = sum(res)
print 'falseCount = ',falseCount
print 'amount = ',amount
print 'The Rate is',falseCount/(amount+0.0)
print 'using ',time.time() -  start ,'s'
```

如此加入了多线程之后，代码的效率提升得非常快，把这个库推荐给其他同学之后也广受好评。

6.4. 布隆过滤器和并发爬虫的结合

这一步并没有太大的新意，只是想把两个内容结合到一起来提高效率，所以这里我就去掉了crawled这个list，而是利用了布隆过滤器的一个对象mbf来进行处理，但是发现效率提升并不明显，这也许是因为我爬虫爬取页面的个数不够多。

7. 总结 (Conclusions)

这两次实验让我更加了解了做一个接近工业级的爬虫需要考虑哪些方面的问题，尤其是关于布隆过滤器的实验，让我体验到了优秀数据结构带来的魅力。关于多线程编程的练习也让我对提高CPU资源利用率有一个初步的见识。

另一个非常深的感受就是，帮其他同学debug能让我收获很多，比如解决线程不能停止的地方，虽然我自己的代码一开始并没有出现问题，但是通过帮其他同学debug，让我知道了我的写法还是有隐患的。

8. 参考（References）

http://tieba.baidu.com/p/3596194843

https://www.python.org/dev/peps/pep-0476/

http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html

http://www.cnblogs.com/heaad/archive/2011/01/02/1924195.html

http://blog.jobbole.com/58700/

http://blog.chedushi.com/archives/9158

最后，再次感谢张娅老师和两位助教的耐心指导。

林禹臣

5140309507

10月6日，2015年