电类工程导论(C 类)实验 2&3 报告

贾萧松 516030910548

一、实验概述

Lab2 中，主要讲了 html 协议和表单的相关知识，以及用 bfs、dfs 实现了一个简单的爬虫。

二、实验环境

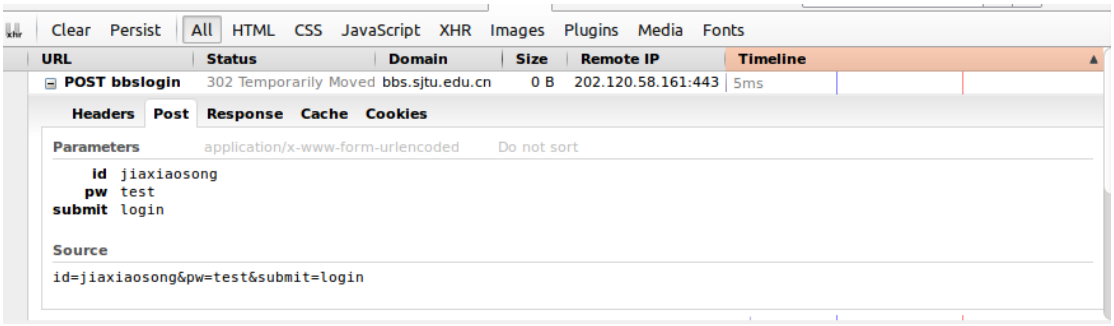
Ubuntu14.04+python2.7+beautifulsoup4

三、实验内容

1. Lab2 Ex1

使用自己的账号模拟登陆 BBS 后，修改个人说明档。

首先我通过一次在网页上的登录来观察 HTTP 请求，其中完成本练习关键的 request-body 如下：



打开 html 网页 登录相关代码如下：

```
▼<form action="/bbslogin" target="_top" method="post" name="form1"
id="Form1" class="inline">
    <label for="Text1">账号:</label>
    <input type="text" name="id" maxlength="12" size="12" class=
    "text" id="Text1">
    <label for="Password1">密码:</label>
    <input type="password" name="pw" maxlength="12" size="8" class=
    "password" id="Password1">
    <input type="checkbox" id="Checkbox1">
    <label for="Checkbox1">记住密码</label>
    <input type="submit" name="submit" class="button" value="login"
    id="Submit1">
  </form>
```

于是我明白了对于 name 为'id' 和 'pw' 的 input 标签，我需要赋给它们我需要输入的值

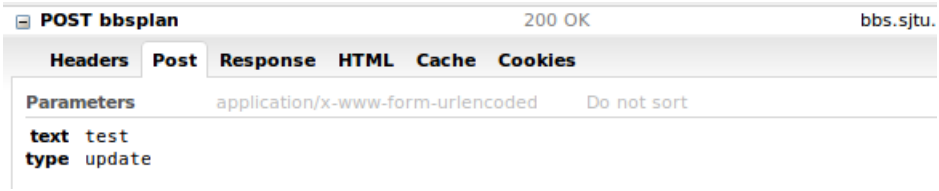
而对 name 为'submit'的 input 标签，我需要赋给它自带的'login'来传递我按下了登录按钮的消息。

明白了原理写出的 python 代码如下：

```python
cj = cookielib.CookieJar()
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
urllib2.install_opener(opener)
postdata = urllib.urlencode({'id':id, 'pw': pw, 'submit':'login'})
req = urllib2.Request(url = 'https://bbs.sjtu.edu.cn/bbslogin', data = postdata)
response = urllib2.urlopen(req)
```

其中 Cookie 那一段是因为访问 BBS 欢迎页时需要将个人信息发给网站，网站才会显示 ID。个人信息保存在 Cookie 中。

接下来，修改个人说明档也是同样的道理：



代码如下：

```
postdata2 = urllib.urlencode({'type':'update', 'text':text})
req2 = urllib2.Request(url = 'https://bbs.sjtu.edu.cn/bbsplan', data = postdata2)
response2 = urllib2.urlopen(req2).read()
```

2. Lab2 Ex2&3

修改 crawler_sample.py 中的 union_bfs 函数，完成 BFS 搜索

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

只需要按照原理做就可以

修改 crawler_sample.py 中的 crawl 函数，返回图的结构

```
            graph[page] = outlinks
```

只需将 page 和 page 中的链接相关联

3. Lab2 Ex4

输入网页内容 content，网页内容所在的网址 page，以 list 形式返回网页中所有链接。建议匹配所有绝对网址和相对网址。
题目提示：soup.findAll(‘a’,{‘href’ : re.compile(‘^http|^/’)}) 可以匹配以 http 开头的绝对链接和以/开头的相对链接。urljoin 可以将相对链接变为绝对链接。

我创建了名为 tmp 的 set 以避免重复，然后按照提示寻找 url，除此之外我去掉了结尾的’/’以减少 url 重复。

```
def get_all_links(content, page):
    links = []
    tmp = set()
    soup = BeautifulSoup(content)
    for goal in soup.findAll('a',{'href':re.compile('^http|^/')}):
        url = goal.get('href')
        if(url[0]=='j' or len(url)<6):
            continue
        if(url[0] == '/'):
            url = urlparse.urljoin(page, url)
        if(url[-1] == '/'):
            url = url[:-1]
        if(url[0] == 'h'):
            tmp.add(url)
    links = list(tmp)
    return links
```

输入网址 page，返回网页内容 content。注意做异常处理（try/except，防止网页无法访问），建议在 urlopen 时加超时参数 timeout。

经过上网查询我学习到：设置了 timeout 超时参数，read 超时的时候会抛出异常，想要程序稳定，还需要给 urlopen 加上异常处理，代码如下：

```
def get_page(page):
    content = ''
    try:
        content = urllib2.urlopen(page, timeout = 5).read()
    except Exception, e:
        print "Open Url Error"
        return None
    return content
```

5 只是我的一个测试值，具体数值需要根据自身网速和目标网页来进行调试。

最后为主函数：

```
def crawl(seed, method, max_page):
    tocrawl = [seed]
    crawled = []
    graph = {}
    count = 0
    max_page = int(max_page)
    while tocrawl:
        page = tocrawl.pop()
        if page not in crawled:
            crawled.append(page)
            content = get_page(page)
            if(content == None):  #不再爬取该网页
                continue
            count += 1
            add_page_to_folder(page, content)
            outlinks = get_all_links(content, page)
            graph[page] = outlinks
            globals()['union_%s' % method](tocrawl, outlinks)
            print(count >= max_page)
            if(count == max_page):
                break
    return graph, crawled
```

我在提供的 sample 的基础上，增加了对 content 值的判定，如果为 None 也就是这个网页不需要爬取，而对 count 的值的递增和判定也就实现了要求的 maxpage。

4. Lab3 Ex1

实现一个简单的 BloomFilter。
设计一个实验统计你的 BloomFilter 的错误率(false positive rate)。

我采用的的方法是用一个类 bloomfilter 实现。
代码如下：

```
def BKDRHash(the_seed,key):
    seed = the_seed # 31 131 1313 13131 131313 etc..
    hash = 0
    for i in range(len(key)):
      hash = (hash * seed) + ord(key[i])
    return hash

class bloomfilter:
    def __init__(self, m, k):
        self.bitset = Bitarray(m)
        self.k = (k if k<=10 else 10)
        self.m = m
        self.seed = [31, 131, 1313, 13131, 131313, 1313131, 13131313, 131313131,

    def set(self, key):
        for i in range(self.k):
            value = BKDRHash(self.seed[i], key)%(self.m)
            self.bitset.set(value)

    def check(self, key):
        for i in range(self.k):
            value = BKDRHash(self.seed[i], key)%(self.m)
            if(not self.bitset.get(value)):
                return False
        return True
```

其中:

BKDRHash 函数可以通过改变 seed 来得到不同的哈希函数

在 bloomfilter 类中，成员有 bitset，k（哈希函数个数，最大为 10）, m（bitset 的大小），seed（存储了 10 个用于 BKDRHash 函数的 seed）； 函数有 set（将 key 对应的 k 个哈希值%m 对应的 bit 位设为 1），check（给定一个 key，检查是否已经记录）。

对于错误率的检验：

我采用的方法是下载好一个没有重复的单词表（共 45094 个单词），然后对每一个单词先 check 如果为 true 表示这是一个错误（因为实际上没有重复的单词），再在 bitset 中 set 该单词，这样就可以得出在 45094 个单词中产生了几个错误。代码如下：

```
test_file = open('45094.txt')
words = test_file.read().splitlines()
test_file.close()

def false_positive_rate(filter):
    cnt = 0
    for word in words:
        if(filter.check(word)):
            cnt += 1
        else:
            filter.set(word)
    return (cnt/45094.0)

for i in range(2,10):
    for j in range(1,10):
        print false_positive_rate(bloomfilter(i*45094, j))
```

通过改变 k 与 m 的大小得到的表如下：

m/n	m*ln2/n	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8	k=9
2	1.386294	0.213288	0.166231	0.175034	0.200337	0.237526	0.278862	0.318446	0.354593	0.392358
3	2.079442	0.15104	0.092496	0.085377	0.089458	0.108108	0.123897	0.144232	0.166718	0.194083
4	2.772589	0.114849	0.058278	0.044263	0.043265	0.047833	0.05422	0.063068	0.074245	0.08877
5	3.465736	0.093848	0.039917	0.026345	0.022375	0.024127	0.026278	0.029383	0.035526	0.040271
6	4.158883	0.080011	0.029294	0.01814	0.014392	0.014525	0.014392	0.015035	0.016654	0.019692
7	4.85203	0.068302	0.022664	0.012507	0.009026	0.008759	0.008227	0.008671	0.009403	0.010245
8	5.545177	0.060385	0.017519	0.007784	0.0051	0.004324	0.004346	0.00408	0.00428	0.004923
9	6.238325	0.054353	0.014658	0.00581	0.004213	0.004191	0.003371	0.002994	0.002661	0.002927

通过观察可以发现，对于相同的 m/n 的大小，当 k 在 m*ln2/n 左右时取最小，当然这有一定的误差，我猜测原因主要有一是数据量太小，二是选取的样本都为单词，而不是随机字符串。不过，总体来说这个 Bloomfilter 还是基本达到了要求。

5. Lab3 Ex2：

将实验二中的 crawler.py 改为并行化实现。
首先，按提示用字符串改写 get_alllinks 函数，使用正则函数即可：

```
def get_all_links(content, page):
    tmp = set()
    for url in re.compile(r'a href=\"(.+?)\"').findall(content):
        if(url[0]=='j' or len(url)<6):
            continue
        if(url[0] == '/'):
            url = urlparse.urljoin(page, url)
        if(url[-1] == '/'):
            url = url[:-1]
        if(url[0:4] == 'http'):
            tmp.add(url)
    links = list(tmp)
    return links
```

接下来是设置线程和队列：

```
seed = "http://www.baidu.com"
q = Queue.Queue()
q.put(seed)
count = 1
max_page = 50
bf = bloomfilter.BloomFilter(20*max_page, 5)

THREAD_NUM = 10
threads = []
varLock = threading.Lock()
for i in range(THREAD_NUM):
    t = threading.Thread(target = working)
    t.setDaemon(True)
    threads.append(t)
    t.start()


q.join()
```

这里我使用 bloomfilter 来储存被爬取过的 url，其他都按照 sample 中提示的设置。

最后是 working 函数：

```
def working():
    global q, crawled, count, max_page
    while True:
        page = q.get()
        if not crawled.check(page):
            content = get_page(page)

            #fail to read the page
            if(content == None):
                crawled.set(page)
                q.task_done()
                continue

            add_page_to_folder(page, content)
            outlinks = get_all_links(content, page)

            if varLock.acquire():
                print page
                for url in outlinks:
                    if(count <= max_page):
                        count += 1
                        q.put(url)
                    else:
                        break
                crawled.set(page)
                varLock.release()
        q.task_done()
```

思路就是每一个线程循环执行函数体进行爬取，直到 queue 中为空，这里我在向 queue 中添加时就检测是否已经达到最大的页数（也就是说不会添加过量的 url）。

四、问题与解决

1. Lab2 Ex4 : urlopen 中超时抛出异常为空

开始时，urlopen 在遇到异常时，我选择的是 print(str(e))，于是在输出结果时，会有一行为空，起初我以为是 url 读取失败，就把空的 url 读取进去了，后来经过反复调试才发现：抛出的异常只有部分有内容，有部分可能是无法判断，于是 str(e)内容为空，最终我选择遇到异常时统一输出”Open Url Error”

2. Lab2 Ex4：在 terminal 中执行的问题

在写好爬虫后，在 IDE 执行中正常，如果改成在 terminal 中调试，`python2.7 4.py http://www.baidu.com bfs 10`，会出现程序无法停止的状况。经过调试发现了问题所在:terminal 中传递的 10 是 str 类型，如果将’10’与 int 型的 count 比较永远是 false。于是，我就在程序内部加了这样一句，`max_page = int(max_page)`，问题也就解决了。

上面这个问题，提醒我 python 中传递参数时无需类型声明，这带来便利的同时，也需要程序员自己时时思考，明白自己传递的参数类型，否则就会出现像上面一样看似莫名其妙的问题。

3. Lab3 Ex1: IndexError: bytearray index out of range

在生成 BloomFilter 过程中，我发现如果我把 m(bitset)的大小设置的较小，就会报出这个错误，于是我检查了 Bitarray 的代码：

```
class Bitarray:
    def __init__(self, size):
        """ Create a bit array of a specific size """
        self.size = size
        self.bitarray = bytearray(size/8)

    def set(self, n):
        """ Sets the nth element of the bitarray """

        index = n / 8
        position = n % 8
        self.bitarray[index] = self.bitarray[index] | 1 << (7 - position)
File "/home/jia/Desktop/电工C/hw3/hw1.py", line 15, in set

    def get(self, n):
        """ Gets the nth element of the bitarray """

        index = n / 8
        position = n % 8
        return (self.bitarray[index] & (1 << (7 - position))) > 0
```

发现了错误，在构造函数中 bytearray 的大小为 size/8（向下取整），而在 set 和 get 函数中的 index 同样为 n/8（向下取整），也就是说如果 size 比较小，就会出现 index 越界的现象（因为 python 中的 index 是从 0 开始的！）。
上述问题可以有两种解决方案，保证 size 大小为 8 的倍数或者是将 bytearray 的大小加一，我选择了后者，便于调试。

4. Lab3 Ex3:关于线程结束的问题

这个练习中，大部分内容只需要按照提示做就行，由于我选择的是阻塞 Queue，所以重难点是搞清楚线程之间是怎么协调工作的，这样才能保证在工作结束的时候每个线程能够正确退出。

开始的时候，我设计 working 函数的思路是把爬取到的所有 outlink 都添加到 queue 中，然后在每处理一个 page，就把 count+=1，直到 count>=max_page 时，清空 queue，结束线程。

然而，在实际操作中，这里存在一个 bug，就是当你在其中一个线程中清空 queue 时（使用 crawler_2_note 中提供的 clear 函数），queue 的阻塞就结束了，这时候那些全局变量开始开始被销毁，而线程此时还尚未被杀死（即使设置了 setDaemon(True)）,于是在线程调用全局变量时会报错：'NoneType' object has no attribute …。

我的解决思路就是不向 queue 中添加过量的 url，这样在 queue 为空也就是阻塞结束时，此时，线程也就处理完了所有 url。

PS:后来经过和同学探讨，发现阻塞线程要比阻塞队列，在使用上要方便不少，避免了清空队列、处理线程的过程，提交的作业中附有阻塞线程实现的爬虫代码。

五、总结

通过 Lab2, 我学到了 html 协议和表单的相关知识，并用 python 模拟了提交 POST 的过程，用 python 实现了一个简单的爬虫。

通过 Lab3，在 Ex1 中我对 hash 的工作原理有了初步了解，并运用 hash 的理念写了一个简单的 BloomFilter。在 Ex2 中，我完成了一个简单的并行爬虫。

此外，通过在网上查阅资料，我初步了解进程，线程，协程的概念以及工作方式，了解到在 python 中其实多线程由于 GIL(Global Interpreter Lock)的存在其实效率并不是特别高，只不过由于爬虫中有网络请求密集型的操作，所以多线程才可以提高效率。在 CPU 密集型和 IO 密集型的操作中，多进程才可以明显的提高效率。

最后，衷心感谢老师和助教们的精心准备和辛勤付出，你们整理完善的 ppt 和到位的答疑大大提高了我学习的效率，感谢~

贾萧松 516030910548
2017.10.8