第二次实验报告

学号：518030910308
姓名：刘文轩

一、实验准备

1、 实验环境介绍

操作系统：Windows 10
语言：Python 2 & Python 3
IDE：Pycharm 2018.2.4

2、 实验目的

2.1 学习 HTTP 协议，了解 HTTP 请求，并学习查看 HTTP 请求的方式，学会查看 HTML 表单
2.2 学会用 Python 模拟 GET 和 POST
2.3 了解爬虫的概念，了解广度优先搜索策略和深度优先搜索策略
2.4 了解时间复杂度的计算方式，学习哈希散列的查找方式
2.5 设计实现一个简单的布隆过滤器
2.6 实现一个并行的爬虫

3、实验思路

3.1 通过查看 HTML 表单的方法了解交大 BBS 以 POST 方式提交的数据，在 Python 中模拟 POST，先登录，在修改个人的说明文档

3.2 按照课上的要求，修改 union_bfs 函数，完成 BFS 搜索，修改 crawl 函数使它能够返回图的结构之后，修改 crawler.py，完成网页爬虫，需要注意的是，我们可以通过 urljoin 将相对路径变为绝对路径

3.3 基于 Bitarray 库，设计 BloomFilter 类，通过使用 mmh3 库，来自主选择调用的 Hash 函数，并且设计这个类的__iter__，add，__contains__函数，并统计错误率

3.4 结合之前设计的 crawler.py，通过 threading 和 Queue 库，设计并行化爬虫

二、实验过程

1、 练习一 模拟登录 BBS，修改个人说明档

1.1 设置 cookie

为了修改我们的个人说明档，我们首先要登录我们的交大 BBS，这个步骤在爬取各个需要登录以后才能查看信息的网站（如微博）的过程中都是需要的。

为了将保存在 cookie 中的个人信息发给网站，我们需要使用 urllib2，cookielib 库，我们首先使用 cj=cookielib.Cookiejar()，来初始化我们的 cookie。

接下来我们设置 opener，这就需要 urllib2 库的帮助，使用 opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj)) 设置好 opener 以后，通过 urllib2.install_opener(opener)我们就将 cookie 加入 opener 中了，至此我们的初始准备已经做好，cookie 也设置完毕了。

这一段的代码如下：

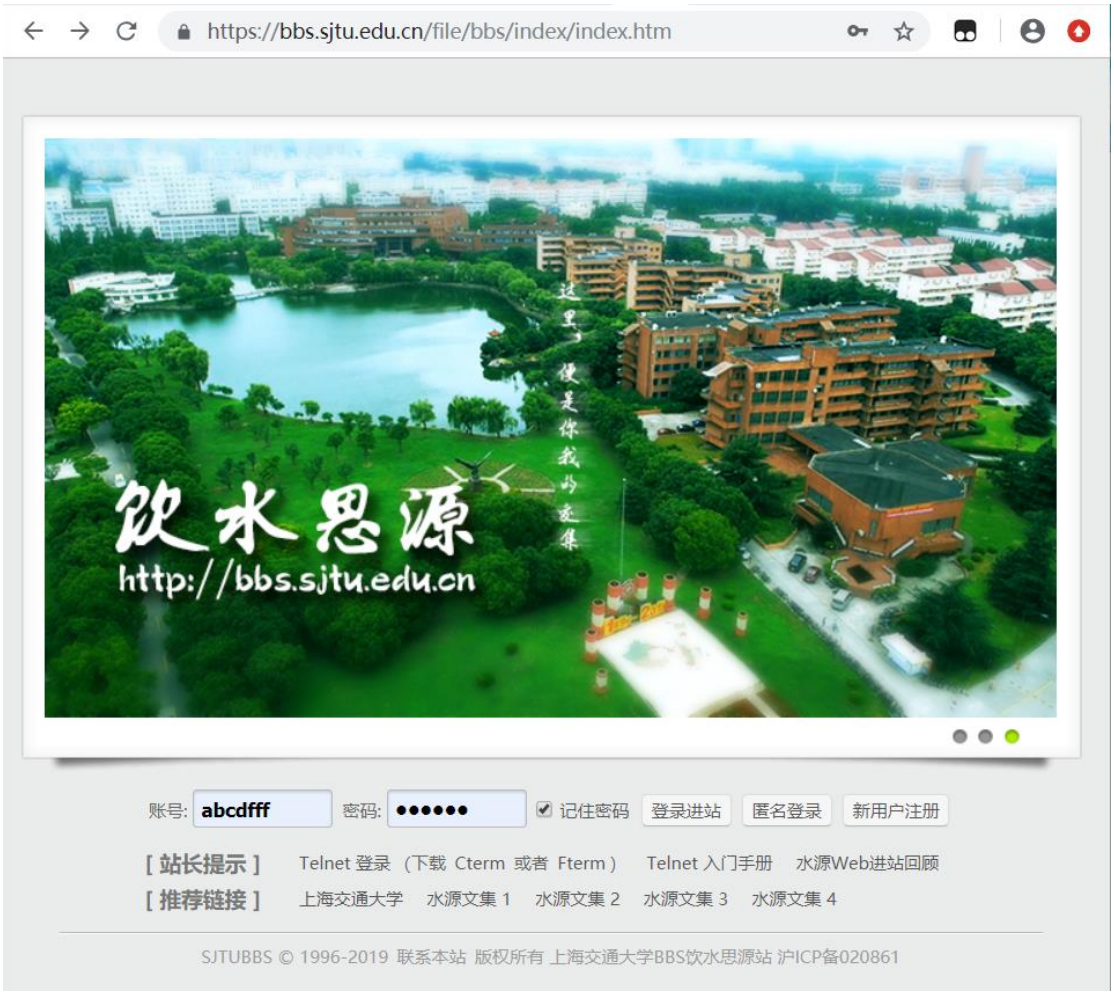
```
cj = cookielib.CookieJar()
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
urllib2.install_opener(opener)
```

1.2 查看 Headers，获取登录所需数据

我们首先来到 bbs.sjtu.edu.cn 这个网站上，通过 Chrome 浏览器的检查，查看我们登录

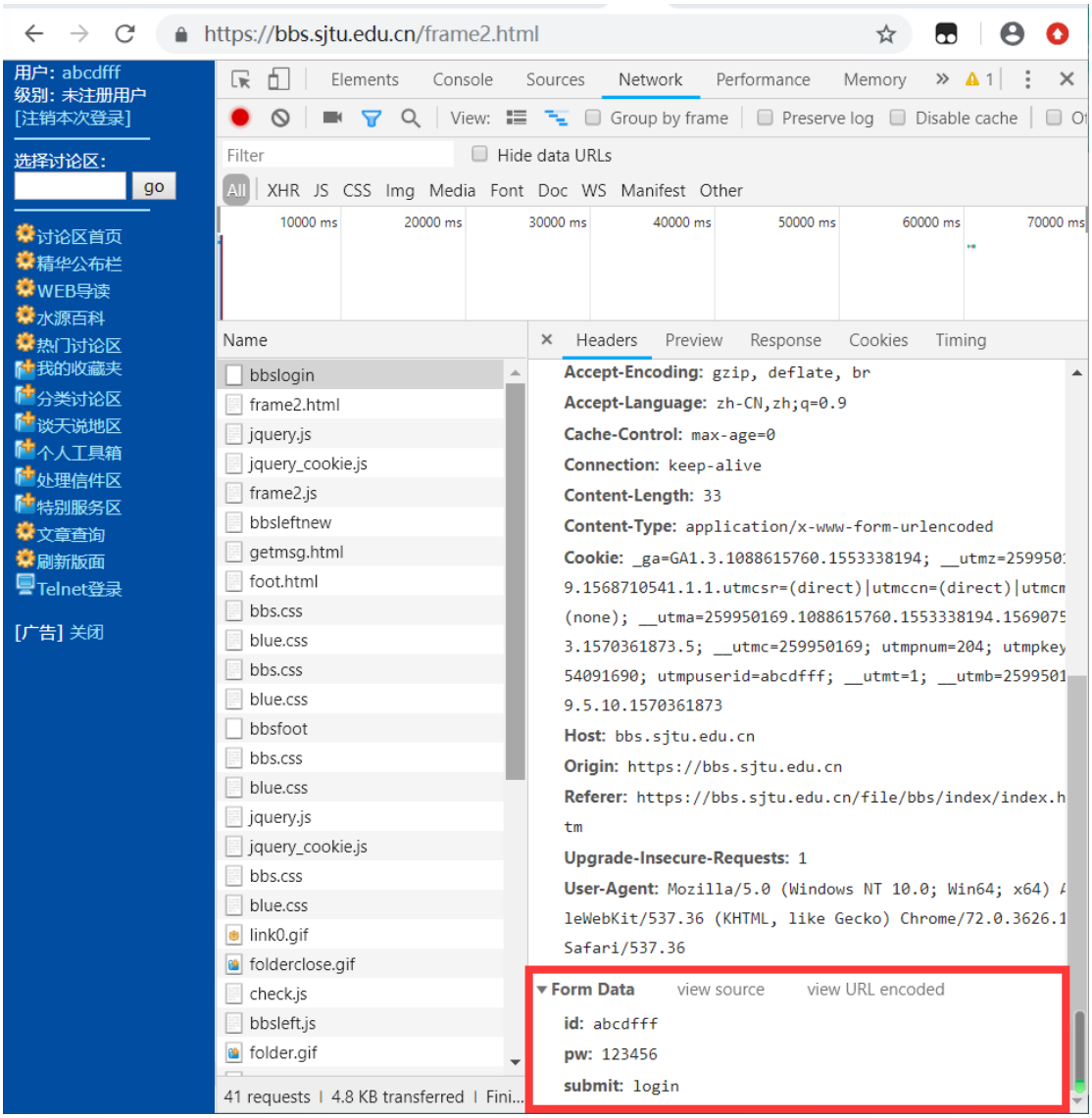
时都提交了哪些数据，在这里，我的实验用的账号为 abcdfff，密码为 123456。

我们来到的界面如下：



我们右键鼠标选择检查，首先选择右上角为 Network，然后登录进站，观察检查页的变化。

因为我们要查看登录所需要的数据，因此我们在左侧的 Name 列中选中第一个 bbslogin，在默认的 Headers 选项中，我们将滚动条拉到最后，看到了“Form Data”这一项，里面显示了我们提交的数据有 id，pw，submit 这三项，接下来我们要做的就是在我们的代码中模拟这些了。



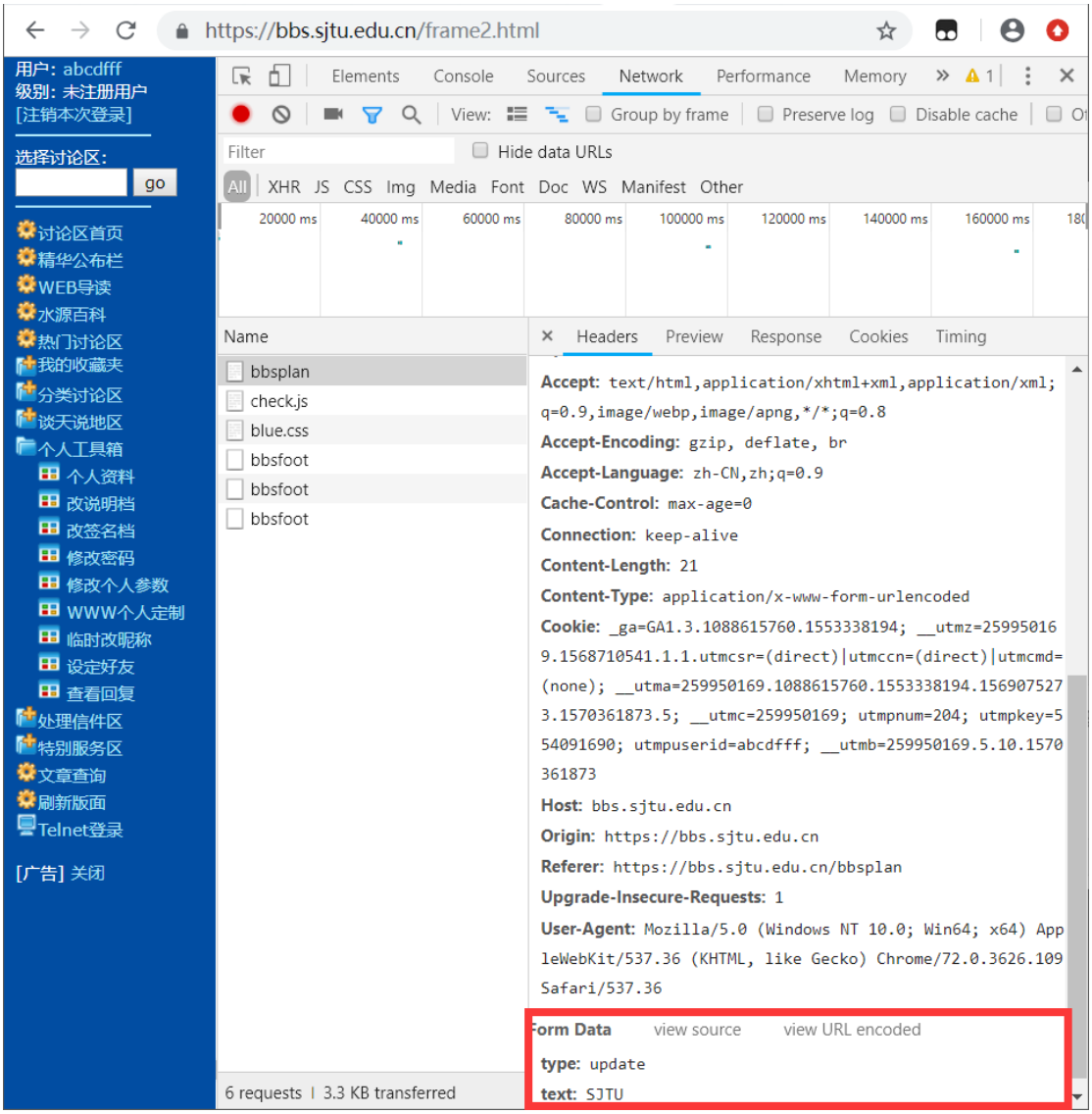
1.3 bbs_set 函数设计

我们注意到，在登录界面，我们提交的数据中需要变动的只有 id 和 pw，同时我们还需要给出一定的文字内容来修改我们的个人说明档，因此 bbs_set 函数需要传入 id，pw，text 三个参数。

我们用 postdata 来保存我们提交的数据，我们使用 urllib.urlencode()函数，其中'id'就设置为传入的 id，'pw'就设置为传入的 pw，'submit'为默认值'login'。

我们首先使用 req = urllib2.Request(url = 'https://bbs.sjtu.edu.cn/bbslogin',data = postdata)和 urllib2.urlopen(req)来登录 bbs。

我们再以上面相同的方式来修改一次个人说明档。查看我们需要提交的数据，我们发现，当我们输入要修改的内容为"SJTU"时，提交的数据分别有 type 和 text 类别。



我们再以和上面相同的方式将数据传入 https://bbs.sjtu.edu.cn/bbsplan 后，用我们在第一次实验中学习的方式，通过 Beautifulsoup 库来查看我们的个人说明档是否修改成功。

需要注意的是，我们需要删掉原先代码中 print 部分的 str()函数，不然由于类型问题将会报错。

完整代码如下：

```python
def bbs_set(id, pw, text):
    import urllib2, cookielib, urllib
    from bs4 import BeautifulSoup
    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)
    urllib2.urlopen(req)
    postdata2 = urllib.urlencode({'type':'update','text':text},)
    req2 = urllib2.Request(url='https://bbs.sjtu.edu.cn/bbsplan',data=postdata2)
    urllib2.urlopen(req2)
    content = urllib2.urlopen('https://bbs.sjtu.edu.cn/bbsplan').read()
    soup = BeautifulSoup(content,features="html.parser")
    print soup.find('textarea').string.strip()
```

1.4 命令行界面操控

为了便于在命令行界面中操控，我们使用 sys 库，sys.argv 获取运行此 python 文件时的命令行参数，我们分别传进 id，pw，text，然后运行函数，查看运行结果。

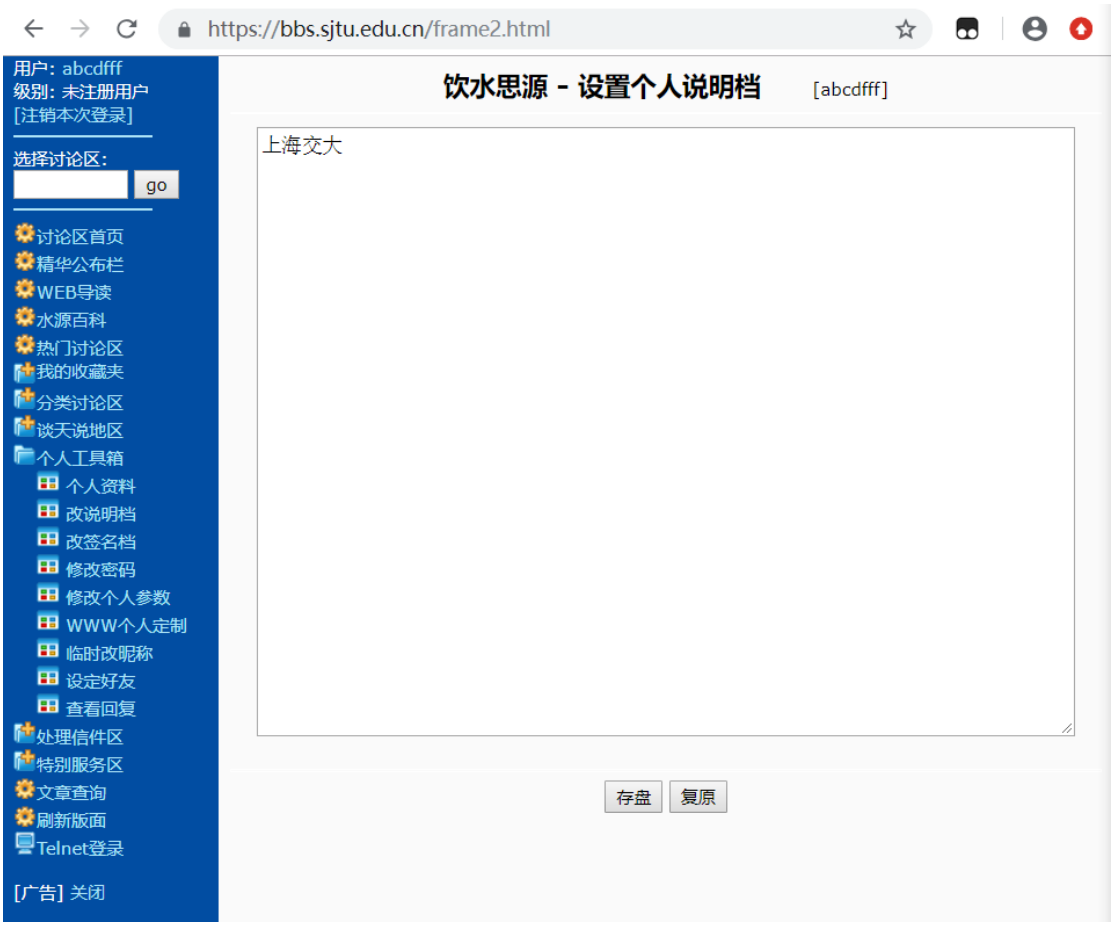
同时需要注意的是，代码是在 windows 环境下运行的，在代码开头已经通过# -*- coding:utf-8 -*-设置了 coding 的情况下，对于 text 我们不再需要 decode 然后再 encode，完整代码如下:

```python
if __name__ == '__main__':

    id = sys.argv[1]
    pw = sys.argv[2]
    text = sys.argv[3] # windows下不需要decode后encode

    bbs_set(id, pw, text)
```

1.5 代码运行结果

在命令行界面运行，运行过程与结果如图所示。

```
C:\Users\63503\Desktop\ex1>python2 bbs_set_sample.py abcdfff 123456 上海交大
上海交大
```



可见，我们的运行结果是非常合适的，对于中文也很适配。

2、练习二 完成 BFS 搜索，返回图结构后，完成爬虫搜索

2.1 完成 BFS 搜索

这个函数要求对于 list a 和 b，在排除了重复元素后，将 b 中的元素插在 a 之前。

这是很容易的，对于 b 中的每一个元素 e，如果它不在 a 中，我们可以通过 a.insert(0,e) 将它放在列表 a 的首位，如此反复我们就将 b 中的元素插在了 a 之前。

完整代码如下：

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

2.2 修改 crawl 函数，返回图结构

这一段代码的修改相对很简单，我们注意到，graph 为字典类型，一个相对的根网站对应着从它爬取得到的网站列表，我们把完整的修改好的 crawl 函数展示出来：

```python
def crawl(seed, method, max_page):
    tocrawl = [seed]
    crawled = []
    graph = {}
    count = 0

    while tocrawl:
        page = tocrawl.pop()
        if count >= int(max_page):
            return graph, crawled
        if page not in crawled:
            count += 1
            print page
            content = get_page(page)
            add_page_to_folder(page,content)
            outlinks = get_all_links(content, page)
            globals()['union_%s' % method](tocrawl, outlinks)
            crawled.append(page)
            graph[page] = outlinks
```

2.3 网站地址的优化

我们获取的网址，可能是相对路径，也可能是一些我们不需要的杂乱的信息，这里我们就需要用正则表达式库和 urlparse 库。

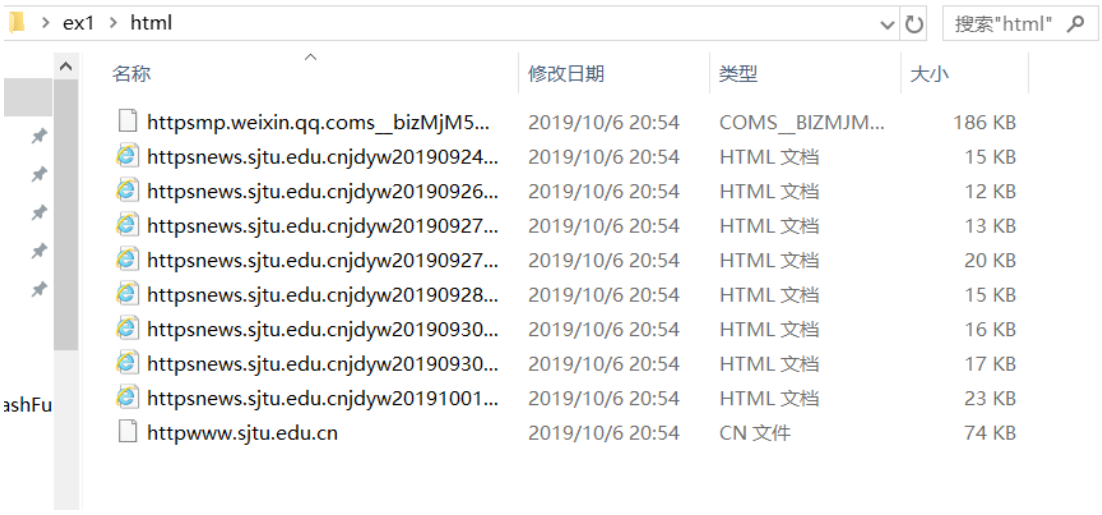
```python
def get_all_links(content, page):
    import urlparse
    links = []
    soup = BeautifulSoup(content, features="html.parser",from_encoding="gbk")
    for i in soup.findAll('a'):
        temp = urlparse.urljoin(page,i.get('href'))
        if re.compile('^http').match(temp):
            links.append(temp)
    return links
```

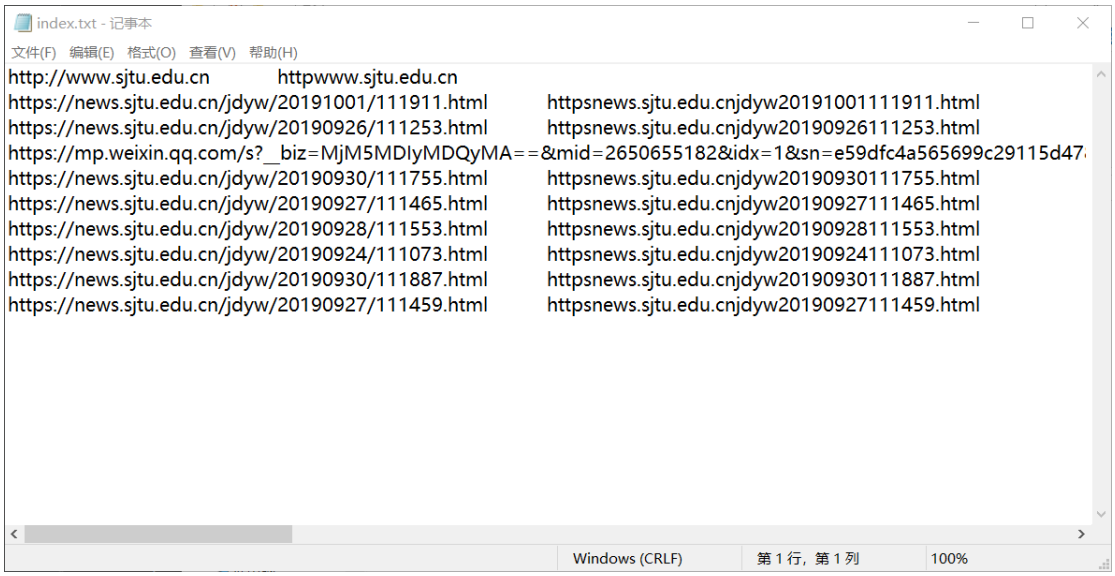
经过实验发现，urlparse.urljoin()是个非常强大的函数，它不仅仅能够将相对路径改写为绝对路径，还能将多级子路径改写为完整的路径，还能够去除一些杂乱的、非网站地址的数据的干扰，在运用了这个函数之后，我们将所有我们需要的网址加入到返回的列表中。

2.4 代码运行结果

我们首先以 bfs 爬取 http://www.sjtu.edu.cn ,我们共爬取十个网站，结果如下。

```
C:\Users\63503\Desktop\ex1>python2 crawler.py http://www.sjtu.edu.cn bfs 10
http://www.sjtu.edu.cn
https://news.sjtu.edu.cn/jdyw/20191001/111911.html
https://news.sjtu.edu.cn/jdyw/20190926/111253.html
https://mp.weixin.qq.com/s?__biz=MjM5MDIyMDQyMA==&mid=2650655182&idx=1&sn=e59dfc4a565699c2911
5d4786d81332c&chksm=be4140258936c9339efccf1ef49d019c56dea87024b8d6acecb90a8db74408a0ab0ebf2e0
b96&mpshare=1&
https://news.sjtu.edu.cn/jdyw/20190930/111755.html
https://news.sjtu.edu.cn/jdyw/20190927/111465.html
https://news.sjtu.edu.cn/jdyw/20190928/111553.html
https://news.sjtu.edu.cn/jdyw/20190924/111073.html
https://news.sjtu.edu.cn/jdyw/20190930/111887.html
https://news.sjtu.edu.cn/jdyw/20190927/111459.html

C:\Users\63503\Desktop\ex1>
```





可见我们爬取的结果是很完整正确的。

3、练习三 实现 BloomFilter

3.1 设计 BloomFilter 类

我们设计的 BloomFilter 类中的位操作使用教师提供的 Bitarray.py，对于其中的 Hash 函数的选择，为了增强程序对于不同数量哈希函数的需求的适应性，我们使用 mmh3 库。

对于 BloomFilter 类，我们需要传入的参数有位数组的大小以及哈希函数的数量，因此我们的初始函数设计如下：

```
    def __init__(self,size,hash_num):
        super(BloomFilter,self).__init__()
        self.bitarray = Bitarray(size)
        self.size = size
        self.hash_num = hash_num
```

对于__len__和__iter__这两个函数，非常简单，我们展示代码之后不再赘述。

```
def __len__(self):
    return self.size

def __iter__(self):
    return iter(self.bitarray)
```

接下来我们首先设计的就是加入元素的函数，依据我们选择的哈希函数的数量，我们会将这些计算出来的索引依次映射到位数组的位中，将这些位的值置为 1。

同理，为了判断某一个元素是否已经被包含，我们需要根据哈希函数的数量，依次判断相应的位是不是为 0，这两部分的函数设计如下：

```
def add(self,item):
    for i in range(self.hash_num):
        index = mmh3.hash(item,i) %self.size
        self.bitarray.set(index)

    return self

def __contains__(self, item):
    flag = True
    for i in range(self.hash_num):
        index = mmh3.hash(item,i)%self.size
        if self.bitarray.get(index) == 0:
            flag = False

    return flag
```

3.2 main 函数设计

因为在本题中，我们选择了 24 个字符串，因此按照课上的介绍，我们在选择了 10 个哈希函数的情况下， 应该将位数组设置为 500 位，既不浪费空间，误报率又很小。

我们设置误报率为误报的数量与添加的字符集和测试集两个集合的数量之比，主函数代码如下：

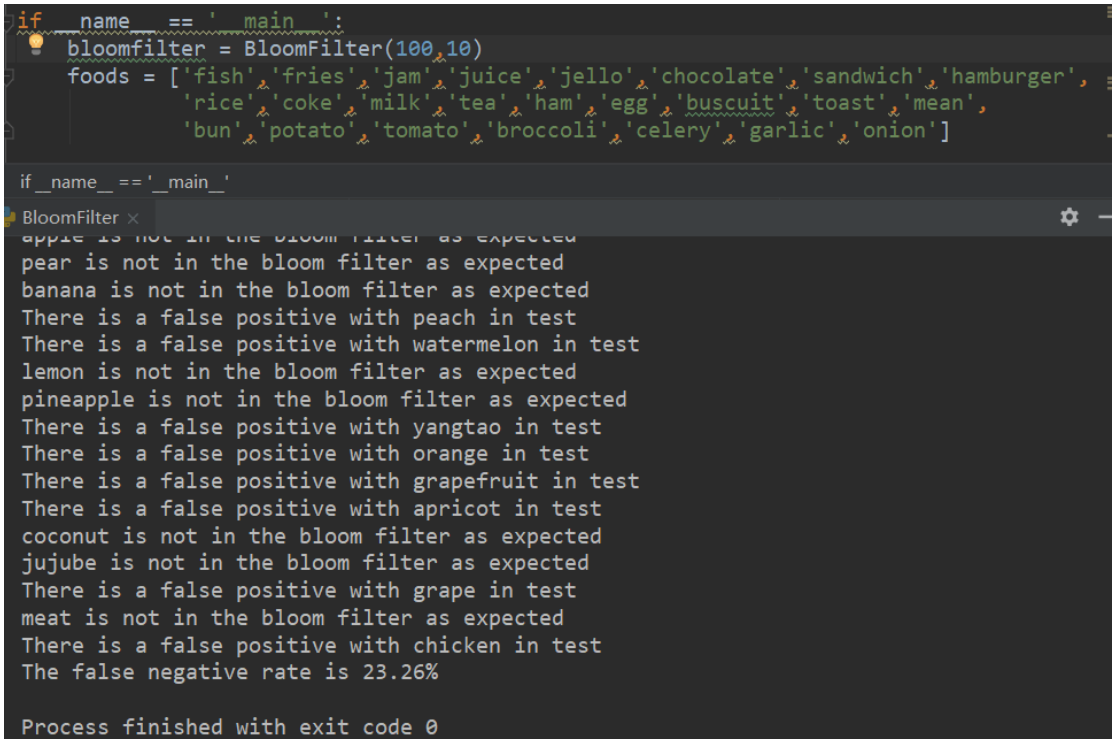
```python
if __name__ == '__main__':
    bloomfilter = BloomFilter(500,10)
    foods = ['fish','fries','jam','juice','jello','chocolate','sandwich','hamburger',
             'rice','coke','milk','tea','ham','egg','buscuit','toast','mean',
             'bun','potato','tomato','broccoli','celery','garlic','onion']

    for food in foods:
        bloomfilter.add(food)

    count = 0
    for food in foods:
        if food in bloomfilter:
            print('{} is in bloom filter as expected'.format(food))
        else:
            print('There is a false positive with {}'.format(food))
            count += 1

    tests = ['cowpea','cabbage','soybean','apple','pear','banana','peach',
             'watermelon','lemon','pineapple','yangtao','orange',
             'grapefruit','apricot','coconut','jujube','grape','meat','chicken']

    total = len(foods)+len(tests)
    for test in tests:
        if test in bloomfilter:
            print('There is a false positive with {} in test'.format(test))
            count += 1
        else:
            print('{} is not in the bloom filter as expected'.format(test))

    print("The false negative rate is {:.2%}".format(count/total))
```

3.3 代码运行结果

我们首先检测在 500 位，10 个哈希函数，储存了 24 个字符串的情况下的错误率。

```
BloomFilter ×
onion is in bloom filter as expected
cowpea is not in the bloom filter as expected
cabbage is not in the bloom filter as expected
soybean is not in the bloom filter as expected
apple is not in the bloom filter as expected
pear is not in the bloom filter as expected
banana is not in the bloom filter as expected
peach is not in the bloom filter as expected
watermelon is not in the bloom filter as expected
lemon is not in the bloom filter as expected
pineapple is not in the bloom filter as expected
yangtao is not in the bloom filter as expected
orange is not in the bloom filter as expected
grapefruit is not in the bloom filter as expected
apricot is not in the bloom filter as expected
coconut is not in the bloom filter as expected
jujube is not in the bloom filter as expected
grape is not in the bloom filter as expected
meat is not in the bloom filter as expected
chicken is not in the bloom filter as expected
The false negative rate is 0.00%

Process finished with exit code 0
```

由图可见，在 500 位的情况下，误报率为 0，结果是相当喜人的，然而在我们设置为 100 位的情况下：



误报率就达到了 23.26%，可见课上的数据是很正确的，在哈希函数为 10 个的情况下，我们选择位数组为字符串个数的 20 倍，能控制误报率在我们可接受的范围内。

4、练习四 实现并行的爬虫

4.1 working 函数设计

为了限制爬虫在我们获取需要的数据量的情况下停止，我们设置了 count，max_page，crawled，q 为全局变量。当爬取数量小于目标数量，队列长度大于零时，我们进行循环。

为了增强程序的适应性，我们使用 try-except 结构，需要注意的是互斥锁的使用，为防止不同线程同时操作一个 crawled 列表产生冲突，我们为 crawled 变量加互斥锁。

其中很重要的一步是，为了让 q.join()判断队列长度为 0 而顺利结束，我们在第一个循环结束后，由于大部分情况下队列里都会有很多的未完成的任务，我们再次加入循环，将它们通过 q.task_down()结束任务，这样子才能让程序顺利运行结束。

修改部分的完整代码如下：

```
def working():
    global count,max_page,crawled,q
    while count<max_page and q.qsize > 0:
        page = q.get()
        if page not in crawled:
            try:
                content = get_page(page)
                add_page_to_folder(page, content)
                crawled.append(page)
                outlinks = get_all_links(content, page)
                for link in outlinks:
                    q.put(link)
                print page
                if varLock.acquire():
                    count += 1
                    varLock.release()
                q.task_done()
            except:
                continue

    while q.unfinished_tasks:
        if varLock.acquire():
            q.task_done()
            varLock.release()
```

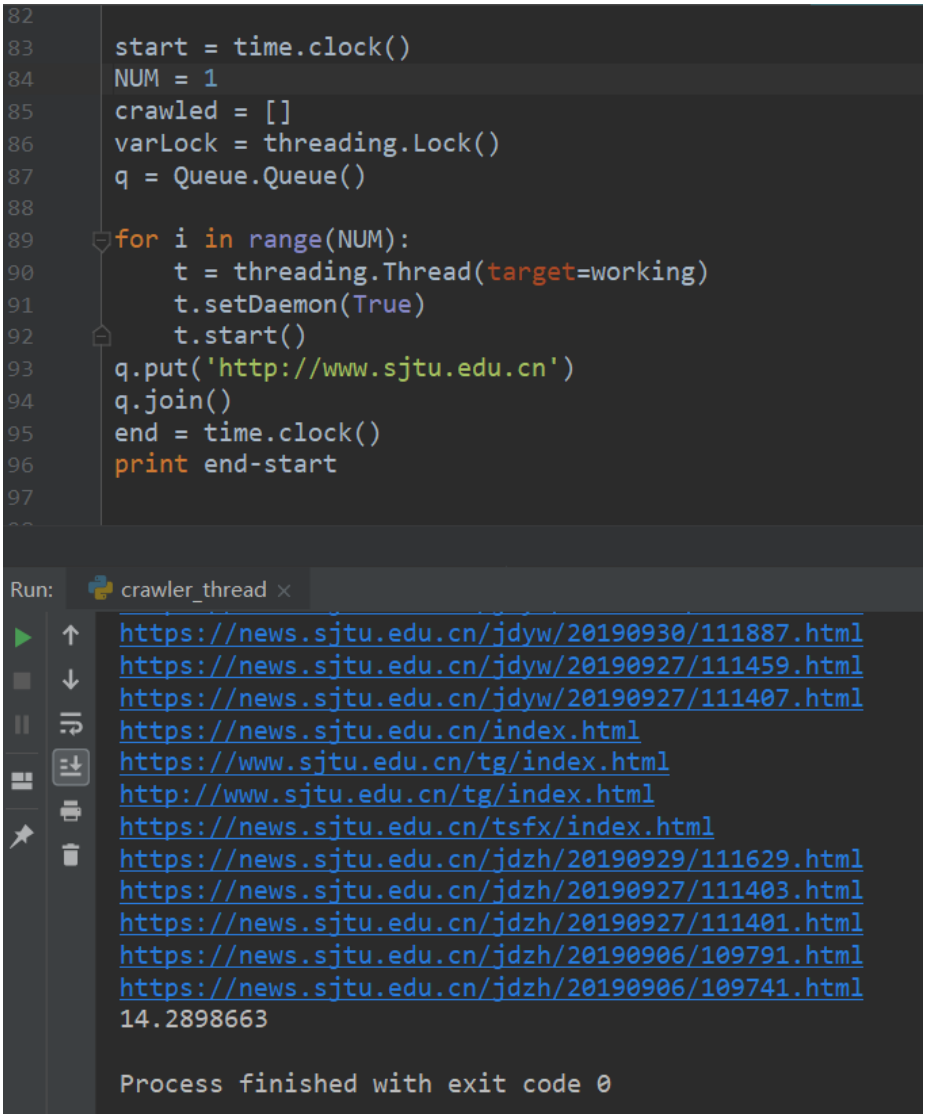
4.2 代码运行结果

在爬取 20 个网站的情况下，我们分别对比八线程与单线程运行的结果：

```
82
83      start = time.clock()
84      NUM = 8
85      crawled = []
86      varLock = threading.Lock()
87      q = Queue.Queue()
88
89      for i in range(NUM):
90          t = threading.Thread(target=working)
91          t.setDaemon(True)
92          t.start()
93      q.put('http://www.sjtu.edu.cn')
94      q.join()
95      end = time.clock()
96      print end-start
97
```

Run: crawler_thread ×

```
https://news.sjtu.edu.cn/jdyw/20191001/111911.html
https://news.sjtu.edu.cn/jdyw/20190927/111459.html
https://www.sjtu.edu.cn/tg/index.html
https://news.sjtu.edu.cn/jdzh/20190927/111403.html
https://news.sjtu.edu.cn/jdzh/20190929/111629.html
https://news.sjtu.edu.cn/jdyw/20190927/111407.html
http://www.sjtu.edu.cn/tg/index.html
https://news.sjtu.edu.cn/tsfx/index.html
https://news.sjtu.edu.cn/jdzh/20190927/111401.html
https://news.sjtu.edu.cn/index.html
https://news.sjtu.edu.cn/jdzh/20190906/109791.html
https://news.sjtu.edu.cn/jdzh/20190906/109741.html
3.4016202

Process finished with exit code 0
```



由实验结果可知，8 线程所需时间约为 3.4 秒，而单线程需要 14.3 秒，运行时间差异显著，并行爬虫优化效率十分显著。

三、实验总结

1、实验概述

本次实验的主要任务，可以总结为通过 POST 方式提交数据给网页修改内容，设计简单的爬虫，布隆过滤器的实现，并行爬虫的设计四个实验，四道练习分别从不同的角度，让我们熟悉了相关的代码设计。

2、感想总结

在这次的实验中，学会了的东西有很多，其中最重要的就是提高了自己处理问题、收集相关资料、解决问题的能力，这在我们将来的学习和工作生活中都是很重要的。而具体细化开来，在本次实验中：

2.1 学会了使用 POST 方式向网站提交数据，修改内容

2.2 学会了查看 HTML 表单

2.3 了解了 DFS、BFS，学会了以这两种方式进行网页爬取的简单实现

2.4 学会了布隆过滤器的 Python 简单实现

2.5 了解了 Queue，threading 库，实现了并行的爬虫，并且进行时间复杂度的比较

3、创新点

首先是对于异常处理，我们不希望当程序出现错误时在用户界面显示一大堆报错相关信息，因此我在并行爬虫的程序中使用了 try-except 来完成对程序异常的处理。

```
try:
    content = get_page(page)
    add_page_to_folder(page, content)
    crawled.append(page)
    outlinks = get_all_links(content, page)
    for link in outlinks:
        q.put(link)
    print page
    if varLock.acquire():
        count += 1
        varLock.release()
    q.task_done()
except:
    continue
```

同时通过对 urlparse.urljoin()函数和 re.compile().match()的使用，我掌握了一种更加完整和更加简洁的将各种相对路径改写为绝对路径，去除无关数据的干扰的方式。

```
for i in soup.findAll('a'):
    temp = urlparse.urljoin(page, i.get('href'))
    if re.compile('^http').match(temp):
        links.append(temp)
```

同时在获取链接时，考虑到了很多不合规的情形，并且通过正则表达式，re.match()这个方法将其一一过滤或者修改，具体的情形已经在上面进行了详细的分析。

4、遇到的问题

很多老师提供的代码，在使用过程中由于平台和语言版本的不同都需要修改，如在 windows 下，在开头指定了编码为 utf-8 的情况下，代码中不需要将获得的中文文本内容先 decode 再 encode。

同时一个困扰了我很长时间的问题，就是在并行爬虫时，如何控制爬虫在获得我需要的链接数的情况下停止，这就需要我对 Queue 有一个更为深入的了解。

对于 q = Queue.Queue()，q.task_down()使得队列里的任务减 1，同时 q.join()会选择当队列长度为 0 时结束，因而在达到预期链接数量后，我们可以通过再加入一个循环，将其中的未完成的任务都结束，这样子就能让程序顺利终止了。

```
while q.unfinished_tasks:
    if varLock.acquire():
        q.task_done()
        varLock.release()
```