实验二报告

1.使用自己的账号模拟登陆BBS后，修改个人说明档

1.1.原理简述

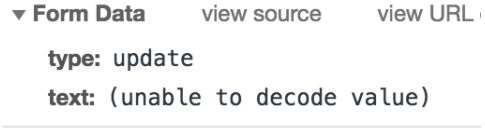
通过前期的学习我们已经知道，浏览器在向网页发出请求时，该请求除了简单的url地址外，还包含着headers头文件，头文件中包括request的方式（get还是post）、User-Agent、Cookie等，这些都是用来说明“我是浏览器的某个用户”的。我的认识是，在爬取带有认证（比如需要用户登录）的https网页时，有两项往往需要注意：
①让程序忽视SSL证书未被CA认证的错误，这就需要调用SSL库；
②设置好headers，尽量模仿像浏览器和人工用户

Headers中很重要的一项就是Cookie。Cookie是服务器在本地机器上存储的小段文本并随每一个请求发送至同一服务器，是让客户端保持状态的方案。其内容包括名字，值，过期时间，路径和域等。我们日常登录某网站之后一段时间内不用再重新登录、可以直接连上，就是Cookie在起作用。和Cookie有类似功能的还有Sesion，但其是由服务器在每次请求时产生，这里不赘述。本任务需要在登录状态下完成任务，需要设置好Cookie。

2.2.最初实现

我首先使用的方法比较简单粗暴：既然是Cookie说明了我的登录状态，我就在浏览器中登录状态下打开改说明的网页，然后找到头文件中的Cookie内容，其实就是一段字符串（或许经过映射转码），然后复制粘贴到我request的headers中，这样我的请求就是在登录状态下的了。经过尝试果然可以。

那么如何进行修改呢？再次观察调试窗口，当我在文本框内输入新的说明并点击“存盘”之后，页面网址并没有改变，但内容变成了“修改成功”。同时发现使用POST方式传入了我的修改操作，也就是这个文本框form，其data为：



注意到text部分不能被Chrome转码，这点稍后会详细叙述。目前只需知道，修改说明的操作，实际是通过传入一个POST完成的，而该POST的form格式也知道了，那么就可以带上POST打开网页，发出数据模拟修改request；修改完之后为了查看新的说明，需要重新打开一次网页，这次是不带POST的，那么就是普通的显示说明的页面，简单soup化、找到textarea内的内容就是当前说明。

代码实现如下：

```
import urllib.request
import urllib.parse
from bs4 import BeautifulSoup
import ssl
ssl._create_default_https_context = ssl._create_unverified_context

cookie='_ga=GA1.3.1630716770.1528006602; __utmc=259950169;
__utmz=259950169.1537367217.1.1.utmcsr=(direct)|utmccn=(direct)|
utmcmd=(none);
__utma=259950169.1630716770.1528006602.1537367217.1537370843.2;
utmpuserid=markdana; utmpnum=801; utmpkey=132767295'
```

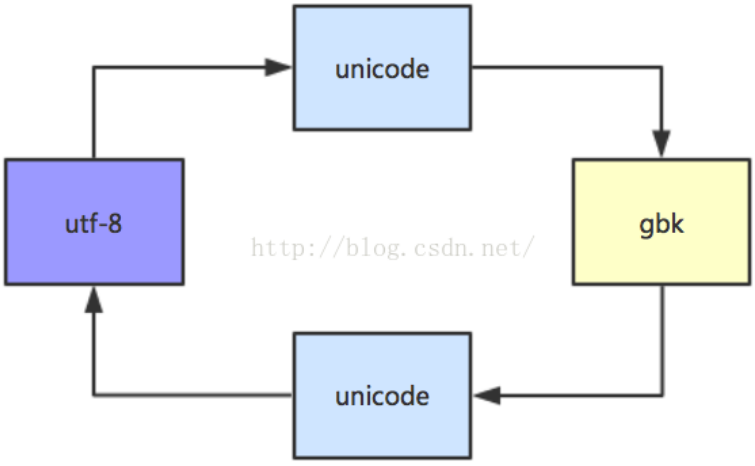
```
ua='Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_2) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/68.0.3440.106 Safari/537.36'

header={'User-Agent':ua,'Cookie':cookie}
postdata = urllib.parse.urlencode({    #根据发出数据模拟request-body
    'type': 'update',
    'text': 'hallooo'
}).encode(encoding='UTF8')

#rq=urllib.request.Request(url='https://bbs.sjtu.edu.cn/
bbsplan',headers=header,data=postdata)
rq=urllib.request.Request(url='https://bbs.sjtu.edu.cn/
bbsplan',headers=header)
data1=urllib.request.urlopen(rq).read()
print(data1)
soup = BeautifulSoup(data1,"lxml")
print (str(soup.find('textarea').string).strip())
```

2.3.优化实现

最初的代码确实完成了任务，但是显然达不到要求，没有广泛的普适性——比如我们要登录多个肉鸡的账号进行操作，不可能依次去浏览器登录然后复制Cookie信息；再比如要面向用户，用户的输入应该是自己的账号和密码，而不是需要用户提供整个header信息。因此我们的程序需要将Cookie这一重要参数通过账号密码获得。可是修改个人说明这个页面并没有登录窗口，因此我们需要先通过在登录页面登录的方式获得网页反馈来的Cookie（说明登录状态），然后再带着Cookie去发出新的请求。处理爬虫时Cookie的问题，Cookiejar库很方便。我们首先建立一个Cookie对象，然后将这个对象加入到urllib.request的opener中，之后每次发送请求与收到反馈时，该Cookie就像浏览器一样会被修改、储存等。因此首先建立Cookie对象在opener中，然后通过POST方式在login页面填好登录的用户名和密码，那么Cookie就可以说明登录状态，其后的步骤就类似于2.2中了。

2.4.难点问题

关于中文的编码问题是该实验中的一个难点。首先我们要知道unicode是一种二进制编码，要变成能显示的汉字需要encode到汉字编码，比如UTF-8, GBK等，其中UTF-8是Unix下的通用编码，这也是为何我们写Python代码时习惯在开头加一句`# -*- coding:utf-8 -*-`。而GBK是Win下对汉字的编码。要将汉字的编码转换，可参考该图示：



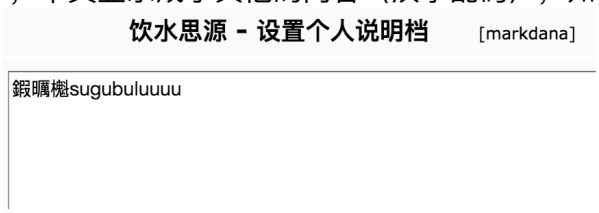
从Unicode到其他编码需要encode函数，从其他编码转换成Unicode需要decode函数。不过需要注意的是，Python3中没有了decode函数，因为3中字符串实际上也是Unicode编码的。

回到刚才发现的一个问题，我在页面填入想要的信息后，Chrome抓的包中无法解析text的内容，

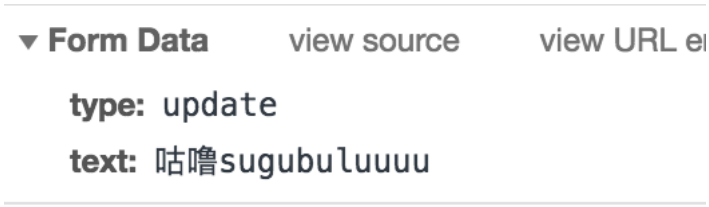
而当我通过写的爬虫传入我的POST信息后（代码如下），

```
updatedata = urllib.parse.urlencode({
        'type': 'update',
        'text': ("%s"%text)
         }).encode(encoding='UTF-8')
```

发现更新的我的个人说明中，中文显示成了其他的内容（汉字乱码），如下图，



有趣的是，现在再存盘，查看浏览器抓取的POST内容，text就可以显示了并且是我的原意内容，如下图，



通过分析我认为，是该网址使用的编码方式和我给文字编码的方式不同，而我给文字编码的方式和Chrome浏览器使用的编码方式相同。通过查看网页源代码，我发现其head部分说明了`charset='gb2312'`，这是GBK下的一种对汉字的编码方式。在UTF-8中，三个%节代表一个汉字，而在GB2312中，两个%节代表一个汉字，这也是为何我两个字节的'咕噜'变成了三字节的乱码，两个字节的'你好'，在UTF-8下编码为`=%E4%BD%A0%E5%A5%BD'`，而被GB2312解码成了`<table><tr><td><texta 浣犲ソsugulutu</textar`。

回到POST内容，在updatedata后面改成`encode(encoding='gb2312')`，再次实验，发现仍然无用。或许因为内部已经是utf-8字符串，不能直接encode。最后从根部解决，`'text':("%s"%text).encode(encoding='gb2312')`，text字符串的编码就是gb2312了。问题解决。

2.实验2和3：完成BFS搜索、DFS搜索，使crawl函数返回graph结构

2.1.原理简述

这两个任务完成起来较为容易。重要的是理解编程思想。我认为这两任务中值得学习的编程思想有两个，

①利用算法实现图论中的内容。离散数学中BFS、DFS搜索都学过，但是没有想到过使用编程来实现。编程的实现精妙，图的结构通过字典来实现，而深度优先还是广度优先可以通过向待搜列表中添加元素的位置不同来实现。添加在即将搜索位，就是沿深度一直搜完；添加在最后，就是先搜完当前位置的，也就是广度优先。

②复杂程序的分层、分步设计。直接搜索爬虫比较复杂，但这里我们将复杂问题拆分。首先搭起具体的框架，也就是通过一个个函数实现的算法实现。这些函数的传入实参先只是标识符，数据结构正确即可，比如这里的ABCD代表稍后会参与运算的url地址。然后在后续步骤中，再进行实际填充完善，也就是实验4中的爬虫函数等。

2.2.运行结果

```
graph_dfs: {'A': ['B', 'C', 'D'], 'D': ['G', 'H'], 'H': [], 'G': ['K',
'L'], 'L': [], 'K': [], 'C': [], 'B': ['E', 'F'], 'F': [], 'E': ['I',
'J'], 'J': [], 'I': []}
crawled_dfs: ['A', 'D', 'H', 'G', 'L', 'K', 'C', 'B', 'F', 'E', 'J', 'I']
graph_bfs: {'A': ['B', 'C', 'D'], 'B': ['E', 'F'], 'C': [], 'D': ['G',
'H'], 'E': ['I', 'J'], 'F': [], 'G': ['K', 'L'], 'H': [], 'I': [], 'J':
[], 'K': [], 'L': []}
crawled_bfs: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L']
```

3.实验4：完成网页爬虫

3.1.任务简述

该任务需要从一个url开始作为种子，爬取其页面中的所有超链接。而这些所有超链接又作为种子，分别爬取各自页面中的超链接。如此深入，可选择利用BFS还是DFS，直至没有新的超链接可爬或是爬取的总超链接数达到了实现设定的max范围。对于每个爬取的网页，其名字（也就是经过可成文件名修改后的url）和url会写入一个txt文件，而以该名字命名的爬取的内容储存在HTML文件中，再放入一个文件夹中。这就是整个操作流程。由于算法实现、爬虫内容实现在之前均详讲过，这里只需要注意一些细节问题。

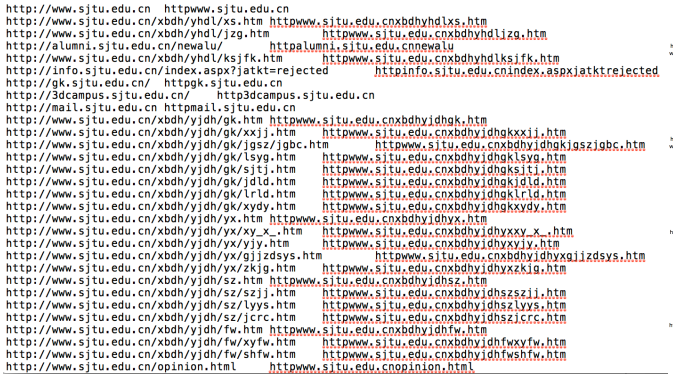
3.2.细节问题

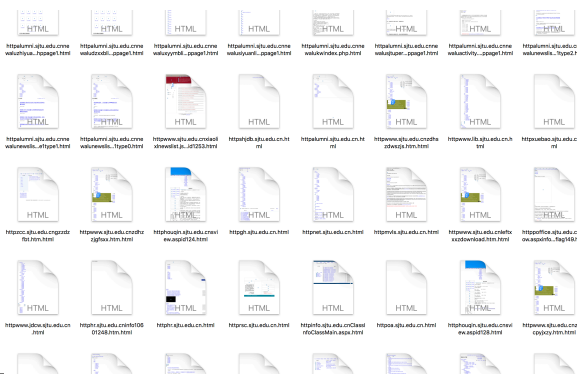
①对超链接进行标准化处理。某页面爬取的所有超链接中，可能某些是相对地址，某些是绝对地址，在这种情况下，需要调用urljoin库进行标准化处理，都变成绝对地址。

②增强爬取的鲁棒性。某页面的所有超链接中，可能就会有一些打不开或者打开很慢的地址。部分是因为其超链接为javascript动态而非url地址，部分是坏死节点。为了不拖慢整个过程或中途报错，我设置了`timeout`参数，并且使用`try`模块

```
try:
        content=urllib.request.urlopen(rq,timeout=1).read()
        return content
except:
        print("FAILED WHEN OPENING "+url)
        pass
```

也就是如果打开时间超过了1s或者其他各种问题时，就打印出提醒，同时返回值为默认None；否则返回正常读取的页面内容。之后的函数要传入页面内容时，就先判断该内容是否为None。以下是index文本内容、文件夹内的内容、以及随程序运行输出的提示内容。





```
http://alumni.sjtu.edu.cn/newalu/
http://www.sjtu.edu.cn/xbdh/yhdl/ksjfk.htm
http://info.sjtu.edu.cn/index.aspx?jatkt=rejected
http://gk.sjtu.edu.cn/
http://3dcampus.sjtu.edu.cn/
http://en.sjtu.edu.cn/
FAILED WHEN OPENING http://en.sjtu.edu.cn/
http://www.jwc.sjtu.edu.cn/web/sjtu/198109.htm
FAILED WHEN OPENING http://www.jwc.sjtu.edu.cn/web/sjtu/198109.htm
http://mail.sjtu.edu.cn
http://www.sjtu.edu.cn/xbdh/yjdh/gk.htm
http://www.sjtu.edu.cn/xbdh/yjdh/gk/xxjj.htm
http://www.sjtu.edu.cn/xbdh/yjdh/gk/jgsz/jgbc.htm
http://www.sjtu.edu.cn/xbdh/yjdh/gk/lsyg.htm
http://www.sjtu.edu.cn/xbdh/yjdh/gk/sjtj.htm
http://www.sjtu.edu.cn/xbdh/yjdh/gk/jdld.htm
http://www.sjtu.edu.cn/xbdh/yjdh/gk/lrld.htm
http://www.sjtu.edu.cn/xbdh/yjdh/gk/xydy.htm
http://www.sjtu.edu.cn/xbdh/yjdh/yx.htm
http://www.sjtu.edu.cn/xbdh/yjdh/yx/xy_x_.htm
http://www.sjtu.edu.cn/xbdh/yjdh/yx/yjy.htm
http://www.sjtu.edu.cn/xbdh/yjdh/yx/gjizdsys.htm
http://www.sjtu.edu.cn/xbdh/yjdh/yx/zkjg.htm
http://www.sjtu.edu.cn/xbdh/yjdh/sz.htm
http://www.sjtu.edu.cn/xbdh/yjdh/sz/szjj.htm
http://www.sjtu.edu.cn/xbdh/yjdh/sz/lyys.htm
http://www.sjtu.edu.cn/xbdh/yjdh/sz/jcrc.htm
http://www.sjtu.edu.cn/xbdh/yjdh/fw.htm
http://www.sjtu.edu.cn/xbdh/yjdh/fw/xyfw.htm
http://www.sjtu.edu.cn/xbdh/yjdh/fw/shfw.htm
http://www.sjtu.edu.cn/opinion.html
http://news.sjtu.edu.cn/jdyw/20180405/11199.html
http://join.sjtu.edu.cn/
http://foundation.sjtu.edu.cn/
http://news.sjtu.edu.cn/jdyw/20180917/83269.html
http://news.sjtu.edu.cn/jdyw/20180918/83405.html
https://www.sjtu.edu.cn/
http://news.sjtu.edu.cn/jdyw/20180920/83671.html
http://news.sjtu.edu.cn/ztzl_jdms/20180919/83517.html
http://news.sjtu.edu.cn/
http://www.sjtu.edu.cn/xinban/lslby.jsp?urltype=tree.TreeTempUrl&wbtreeid=1639
http://news.sjtu.edu.cn/tsfx/index.html
http://news.sjtu.edu.cn/ztzl/index.html
http://media.sjtu.edu.cn
FAILED WHEN OPENING http://media.sjtu.edu.cn
http://news.sjtu.edu.cn/jdyw/20180917/83265.html
http://news.sjtu.edu.cn/jdyw/20180919/83501.html
http://news.sjtu.edu.cn/jdyw/20180915/83197.html
```

实验三报告

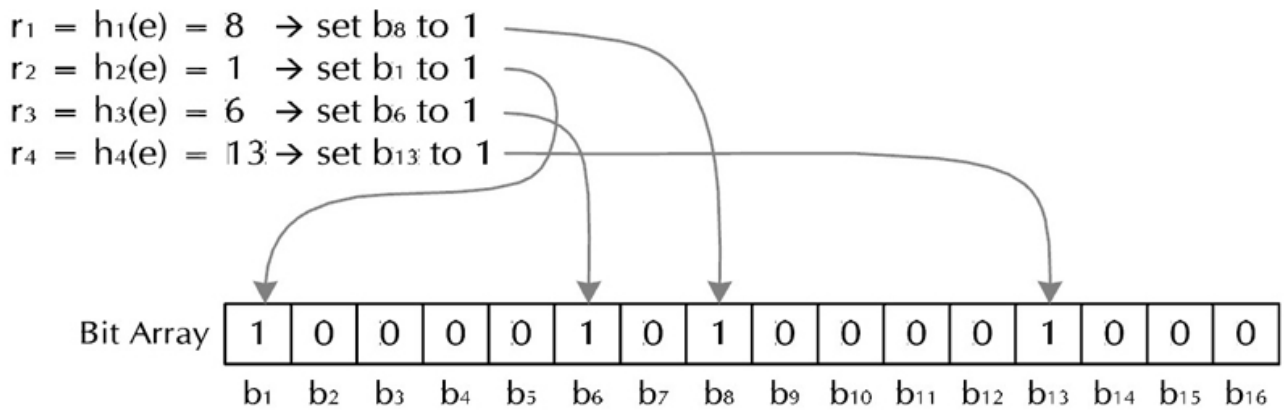
1.实现BloomFilter

1.1.原理简述

在之前的PPT教程中我们已经知道了哈希函数的概念和特性。它可以以极小的准确度牺牲来换取极大的查找时间加快。引入部分我们使用了Hash_Table，Table分为若干个不同的bucket，哈希值相同的元素放入一个bucket中，查找某元素是否在Table中，可以先算出其哈希值，类似于字典中的索引，然后在bucket中再去查找。搜寻范围被极大缩小，则搜索速度有所加快。

但是我们还面临着一个问题，搜索速度确实加快了，但是对数据的储存并没有优化。所有元素仍然需要储存在引擎端的服务器内，造成了成本浪费。因此数学家们发明了布隆过滤器。布隆过滤器也可以理解为许多bucket，但是这些bucket不储存实际元素，而是一个二进制位，只储存0或者1。将一个元素放进集合中，实际就是先算出其哈希值，然后将Table中哈希值对应的bucket位设置为1。哈希值是元素的一个映射，我们想要这个映射尽可能是一对一映射，这样从哈希值反推回元素就会

尽可能的准确。怎样实现呢？一方面我们可以设计更加优秀的哈希函数，另一方面我们可以通过k个而非仅仅1个哈希函数的组合来映射一个元素。两个元素对某哈希函数的哈希值一样，或许还有不同；但选用不同的k个哈希函数，两个元素的k个哈希值都相同，我们就很有把握说，这两个元素是相同的。在实际使用中，k个不同的哈希函数可以依靠不同的seed来实现。



例：加入某元素的操作

在搜索某元素时，先算出其k个哈希值，然后搜索Table中对应的bucket。如果存在某bucket位值为0，那么肯定该元素不存在于集合中；如果每个bucket都为1，那么我们有大概率认为该元素在集合中。现在什么影响搜索的准确率？一方面是哈希函数的选择，另一方面，直觉告诉我们，k越大，也就是选择的不同哈希函数越多，搜索越准确。同时哈希值实际上最终做了一个取模运算，为了能够放入有m个bucket的Table中，需要%m。那么bucket的数量，也就是m（位数）越大，搜索也应该越精确。PPT中讲，对于加入n个字符串，当k=ln2*m/n时出错的概率最小。当k取10，m设置为n的20倍时，出错概率0.0000889已经能够满足爬虫需求了。

1.2.储存结构

上面形象化讲述的Table和bucket，我们可以使用一个类Bitarray来实现Table的功能，而bucket就是一个bit为0或1。分析提供的Bitarray类，有两点需要注意：

①要开辟size位的数组空间，使用了操作`self.bitarray = bytearray(size/8)`。这是因为我们需要size位的二进制bit，而一个byte是字节，也就是8位元组。则我们只需要`size/8`个byte来达到size个bit的需求。由于我使用的是Python3，这里需要写成`size//8`才能返回整商部分。另外，当size不整除8时，实际上应该向上取整来开辟空间，同时考虑到%m的边界条件，在这里为了增加健壮性，我实际使用的`bytearray(size//8+1)`

②该类的set函数只能够set0到1，不能够改回来，也就是元素只能添加进来，不能删除出去。原因联系到使用k个哈希值来映射一个字符串，将某元素删除出去需要将k位bit置0，可能其中一两个就影响到了其他的元素。因此删除元素是不被允许的。

1.3.哈希函数

接下来是哈希函数的设置。我整合成了一个函数`BKDRHash(key,k,m)`，其中传入参数key为要映射的字符串，k为选取的不同哈希函数个数，m为Bitarray的size，也就是返回哈希值的上限。需要注意的是，我的seeds只设置了10个，则应有k≤10。实际使用中10个已经能够满足准确率了。若要更大的k来满足更大的数据集，在BKDR哈希函数中，seed为31，131，1313，13131 etc.. 则可以考虑利用循环生成文本字节，再格式化为数字。

该函数的返回值是一个k维数组，储存着k个不同seed算出来的哈希值。我本以为不同seed下的哈希值应该是很不同的，但是实际测试发现了一个有意思的现象：

```
>>>BKDRHash(“happy”,10,100)
[36, 36, 58, 36, 58, 36, 58, 36, 58, 36]

>>>BKDRHash(“happy”,10,1000)
[136, 436, 458, 436, 458, 436, 458, 436, 458, 436]

>>>BKDRHash(“happy",10,1000000000000)
[99047136, 30847993436, 314947395458, 266620112436, 88585545458,
153132012436, 642400545458, 804322012436, 23900545458, 923322012436]

>>>BKDRHash(“happy",10,103)
[70, 93, 49, 5, 71, 2, 88, 84, 6, 18]
```

当m为10的次方时，取模得到的哈希值实际上就是结果的后几位。而在这个BKDR哈希函数中，得到结果是利用了循环乘，seed尾数又带有循环节，因此不同seed的结果的尾数部分有很大的相似度。在m比较小的情况下，不同seed产生的哈希值可能有很多重复，对效果有不佳影响。如何处理呢？可以尽量将m放大，如1000000000000的情况；也可以不取m为10的次方，这样取模就不是尾数，如取103的情况。以上实例告诉我们，选取m时候同时也要考虑到使用的哈希函数和对应seeds的设置，比如这里用了循环乘，m就不要取10的次方，那样会造成尾数的重复情况。

1.4.测试函数

测试函数`test_hash_function(filename)`。其中传入参数filename是测试文本文件的路径。有两个字符串集，`some_words`和`test_words`，对于该文本文件中的每一个单词，随机将其放入其中一个集合（不重复的）。然后根据上文提及的BloomFilter算法，储存`some_words`中的每一个单词到Bitarray对象`bitarray_obj`中，按PPT的提示，`bitarray_obj`的size是20倍的单词个数（`some_words`的长度）。然后对于`test_words`中的每一个单词，计算出其k个哈希值，和`bitarray_obj`比对，预测该单词是否存在于`some_words`中；然后和实际是否存在比较，得出正确率。

事实上，若哈希值预测出不存在，就一定是不存在的，也无需再搜索比对；只需要关注预测存在的部分单词。计算正确率，也只需要考虑预测为存在的部分单词。

在测试函数中，有两个参数可以调整。我默认设置了k=10，m=20*n。那么在不同大小的m情况下正确率有什么区别呢？以下是我的实验结果：

m取值	准确率
m=n	0.4759519038076152
m=2*n	0.47974722884077486
m=5*n	0.7949882273797511
m=10*n	0.9853066439522998
m=20*n	0.9997860962566845
m=50*n	0.9997844827586206
m=500*n	1.0

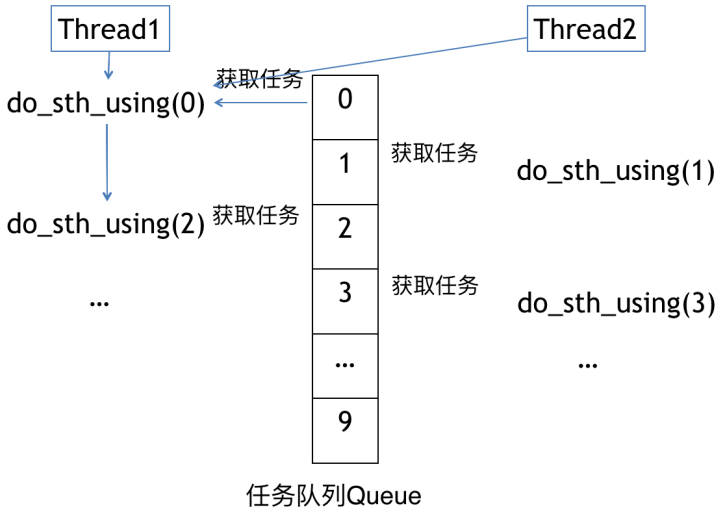
显然当m足够大时，准确率可以无限接近100%；但是由此带来的内存消耗有违于初衷，而带来的准确率提升却并不大，所以我们采用折中的办法，同时兼顾准确率和内存消耗。m=20*n的准确率已经足够满足。

2.实现并行爬虫

2.1.原理简述

本实验我们的目的是要实现并行爬虫。对于一个特定网页的爬取函数在上次实验中已经实现，则该实验的重点落在如何优化to_crawl部分，也就是不同函数任务之间的顺序关系。在之前实现的单线程爬虫中，我们建立了一个to_crawl队列，每次弹出一个url，等到该url被爬取完毕，再弹出下一个url。

而在并发线程的爬虫中，我们设置了Queue类对象q，本质上还是to_crawl队列。所不同的是，我们同时调用threading库，对于NUM线程并行，定义了NUM个thread对象，每个对象在完成手上任务后都去向q申请下一个任务；而只要有thread来申请任务，q都会弹出新任务。这样就实现了NUM线程的并行。需要注意的有两点：
①这里的爬虫方式类似于BFS，因为数据是先进先出的。那么是否可以DFS，也就是后进先出呢？这样对于多线程，由于在队列一段可能有多个线程同时来申请新url和放入新url，或许会造成冲突，对库的实现难度增大。或许需要其他数据结构。
②为了防止不同线程同时申请到一个url，需要添加互斥锁。也就是一方面url在q中排队，一方面不同线程也在排队申请任务。



多任务并行示意图

2.2.编程实现

首先对于不同的线程，要设置他们的任务函数，也就是`working()`。`working()`部分就是对于队列q弹出的url，首先分析其是否被爬取过。如果没有，就打印其名字，爬取其页面内容，放入指定文件夹；并对内容进行文本分析得到页面内所有的超链接；当互斥锁打开时，将他们都放入待爬序列q中。最终将url序列标记为已经爬取。整个任务完成后，通过`q.task_done()`函数向队列发出信号，队列记录该任务已经被完成，彻底弹出，同时该线程再去申请新任务。需要注意的是：

①和上次将url放入一个集合crawled，然后遍历集合查找url是否存在不同，这次我使用了BloomFilter的方法。设置全局变量Bitarray类对象，来记录已经爬取的url。首先设置max_size，然后设置Bitarray的size为20*max_size，这样可以确保准确率。

②如何在爬取max_size之后就停下来？我设置了全局变量count来计数已经爬取过的url数量（由于没有已爬集合，必须使用计数器），在`working()`部分，只有在`count<=max_page`的条件下操作才会进行下去。这样我测试的结果是——确实在爬取了超过max_page数量之后就不再进行爬虫操作了，但是程序并没有停止运行。这是为何呢？这就需要注意到结尾部分的`q.join()`函数了，这是一个阻塞函数，可以理解为它装着剩余任务数，也就是当前的q队列长度。只有在队列长度为0后，它才会往下运行，也就是任务全部完成。而在`working()`部分结尾的`q.task_done()`函数则是用来发出“已完成一个任务”的信号，使`q.join()`记录的当前任务数减1。因此，在`working()`部分，当`count>max_size`后，运行

```
while 1:
        q.task_done()
```

并不进行工作，也就是不向q中放入新任务，只是不停地发出信号，将剩余任务数减到0，使程序结束。

2.3.结果分析

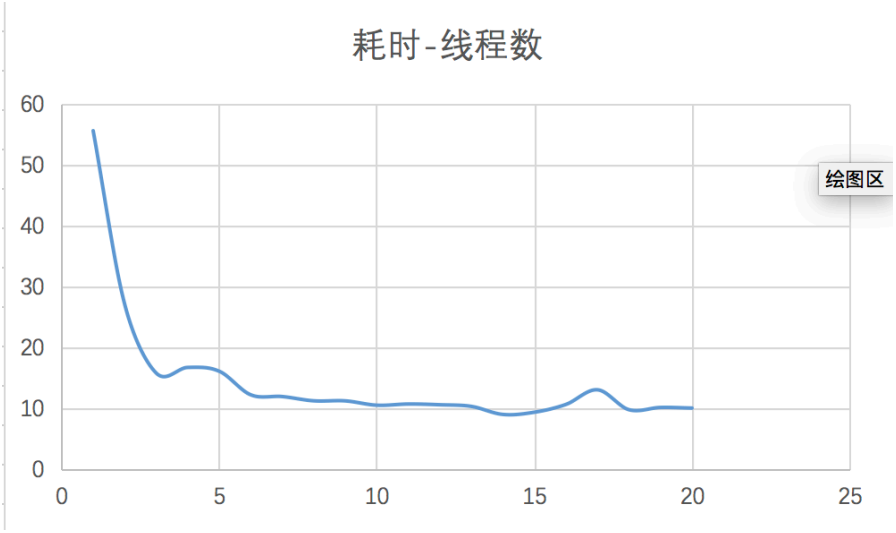
不同的线程数对爬虫的效率有什么影响呢？

```
times=[]
for i in range (1,21):
    bitarray_obj = Bitarray(m)#每次需要更新
    q=Queue()
    varLock = threading.Lock()
    graph={}
    count=0

    start=time.time()
    crawl(seed, i)
    end=time.time()
    times.append(end-start)

print(times)
```

按照以上代码，我进行了测验：以同一个url为seed，max_page都设置为300，依次记录线程为1，2，…，20的情况下，爬完300个网页的耗时。结果如下

线程数	耗时	线程数	耗时
1	55.56924486160278	11	10.863524913787842
2	27.187553882598877	12	10.74594497680664
3	15.930575132369995	13	10.477574110031128
4	16.849175691604614	14	9.161355018615723
5	16.225544929504395	15	9.524857997894287
6	12.375440120697021	16	10.797739028930664
7	12.086585998535156	17	13.192623138427734
8	11.387887954711914	18	9.936655044555664
9	11.378950119018555	19	10.302669763565063

线程数	耗时	线程数	耗时
10	10.670843839645386	**20**	10.203727960586548



耗时-线程数示意图

可以发现，当线程数较小时，增多线程数可以显著提高爬取效率；但当线程数多到一定值之后，牵制爬取速度的就不是线程数了，而是受其他牵制，比如网速、程序中单个任务working()的速度、计算机处理器能力等。