2. Crawler

- HTTP协议
- 爬虫的概念
- 哈希散列
- Bloomfilter
- 并发编程

本次实验满分15分，时间为3周。
PartA和PartB是选读内容，不影响PartC和PartD理解。
需要提交的练习在PPT最后几页（即：BloomFilter和单线程、多线程爬虫）。
标注了“不需要提交”的小练习不参与评分。

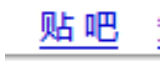
Part A: HTTP协议（选读）

HTTP协议：HTTP请求

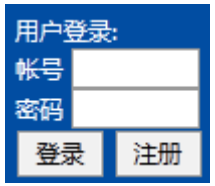
- 从使用者角度，一个HTTP请求起始于：

（1）浏览器输入网址:

（2）网页中的超链接:



（3）提交HTML表单:

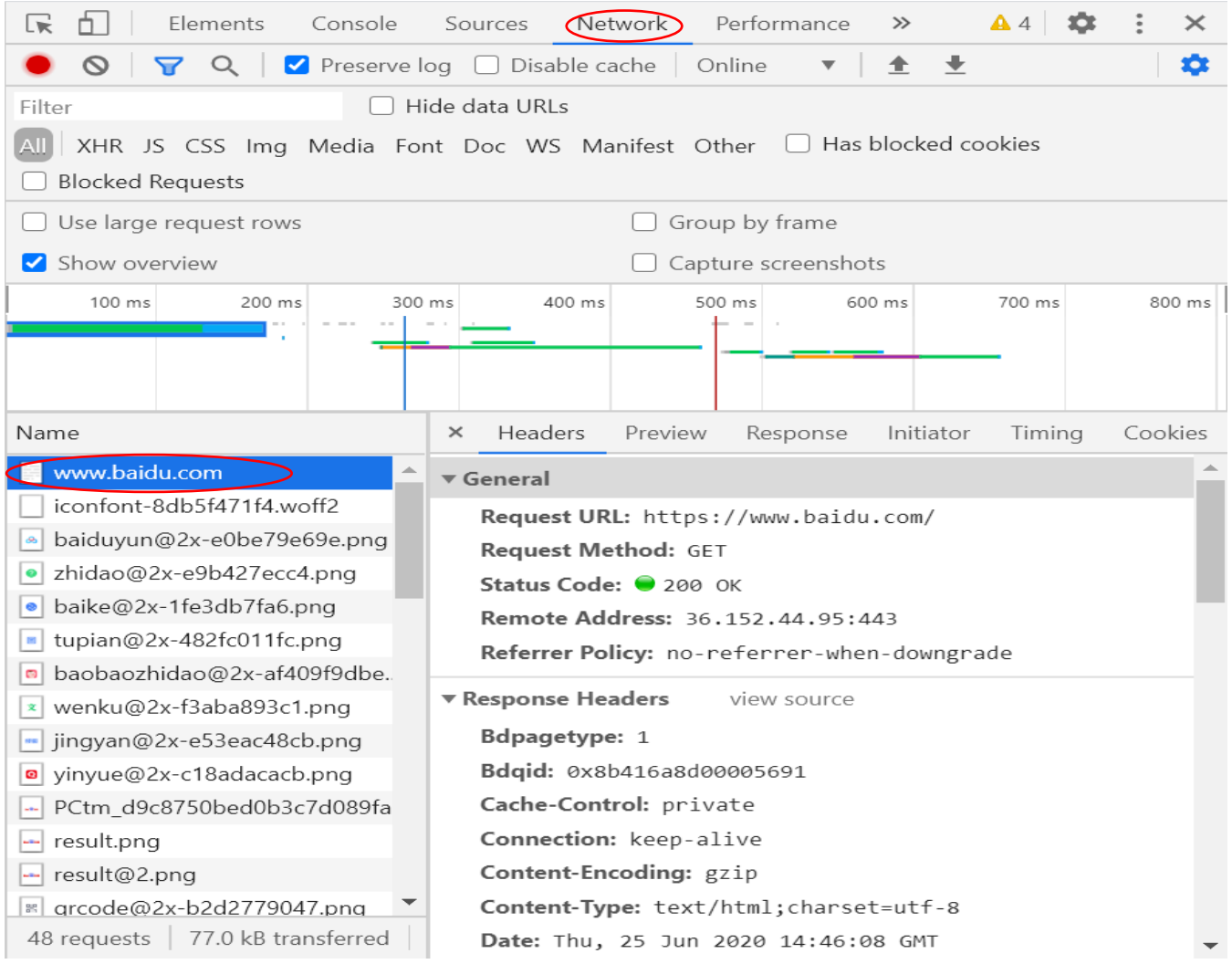


- 本质上说，一个HTTP请求起始于用户端向HTTP服务器发送的一个URL请求。一个标准的HTTP请求由以下几个部分组成：

<request-line>	请求行，用来说明请求类型、要访问的资源（URL）以及使用的HTTP版本
<headers>	头部信息，用来说明服务器要使用的附加信息
<CRLF>	回车换行符(Carriage-Return Line-Feed)(/r/n)，用于标明头部信息的结束
[<request-body><CRLF>]	主体数据（可不添加）

HTTP协议：HTTP请求

- 在Chrome中查看HTTP请求：在Chrome中按F12。选择Network选项卡，点刷新重新加载页面，选择左侧面板上相应网址的HTTP请求，该HTTP请求会在右侧面板中显示。



HTTP协议：HTTP请求

- 浏览器访问网址的HTTP请求

  <request-line>:

  <headers>:

- 上述信息没有request-body部分，这是以GET方式发送的HTTP请求。如果请求中需要附加主体数据，即增加request-body部分，则必须使用POST方式发送HTTP请求。HTML超链接（<a>）只能用GET方式提交HTTP请求，HTML表单（<form></form>）则可以使用两种方式提交HTTP请求。

- 其中，headers包括Cookies。Cookies是当你浏览某网站时，由Web服务器置于你硬盘上的一个非常小的文本文件。它可以记录你的用户ID、密码、浏览过的网页等信息。当你再次来到该网站时，网站通过读取Cookies，得知你的相关信息，就可以做出相应的动作，如在页面显示欢迎你的标语，或者让你不用输入ID、密码就直接登录等等。

HTTP协议：HTML表单

- 一个简单的表单（调用百度搜索框）（在basic.html中查看效果）

```
<form name="input" action="http://www.baidu.com/s" method="GET">
关键词:
<input type="text" name="wd" />
<input type="submit" value="百度一下" />
</form>
```

关键词: 百度一下

- 目标地址（URL）：action标签属性指定了表单提交的目标地址。这里是 http://www.baidu.com/s 。
- 发送方式：method标签属性指定了表单的发送方式，发送方式只有两种：GET及POST。
- 输入类型：type标签属性指定了输入类型。这里text类型为文本输入框，submit为提交按钮。
- 当以GET方式提交表单时，表单的name和value以URL方式提交。name与value均要进行URL Encoding处理。这个操作通常是由用户端浏览器完成的。name与value以name=value形式连接，表单数据和目标地址以?连接。多个name value以&连接（观察百度搜索生成的链接）。

http://www.baidu.com/s?wd=test

HTTP协议：HTML表单

以POST提交的表单

- 饮水思源BBS(bbs.sjtu.edu.cn)登录表单:

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

- 浏览器提交的request-body:

HTTP协议：Python模拟GET

- urllib是python自带的网络接口
- 模拟GET请求

```
>>> import urllib.request
>>> response = urllib.request.urlopen('http://www.baidu.com/')        #GET方式请求网页
>>> content = response.read()                                          #返回的网页在.read()中
```

- 模拟basic.html中的百度搜索框的GET请求

```
1  import urllib.request
2  import urllib.parse
3  from bs4 import BeautifulSoup
4
5  values = {'wd': '上海'}
6  data = urllib.parse.urlencode(values) #name和value以URL Encoding处理
7  constant = "&usm=3&rsv_idx=2&rsv_page=1" #url中加上constant的理由是避开百度安全验证的问题
8  url = 'http://www.baidu.com/s?%s' % data #将表单数据和目标地址以？连接
9  url = url + constant
10 response = urllib.request.urlopen(url).read()
11 soup = BeautifulSoup(response)
12 print(str(soup.title))  #查看网页标题
```

urllib: urllib

```
<title>上海_百度搜索</title>
```

表单

关键词: 上海 百度一下

Request URL: https://www.baidu.com/s?wd=%C9%CF%BA%A3

Request Method: GET

Status Code: 200 OK

Remote Address: 36.152.44.95:443

Referrer Policy: no-referrer-when-downgrade

HTTP协议：Python模拟header

- 模拟headers：某些网站只让浏览器访问，浏览器信息存放在headers中，可以将浏览器信息模拟放入headers中发出。

  尝试不加浏览器信息访问weibo.cn（手机版微博页面）：

  >>> response = urllib.request.urlopen('http://weibo.cn/')

  #HTTPError: HTTP Error 403: Forbidden

  将浏览器信息加入headers：

```
>>> req = urllib.request.Request('http://weibo.cn')                    #生成HTTP请求
>>> req.add_header('User-Agent', 'Mozilla/5.0 (Windows NT 6.1; rv:14.0) Gecko/20100101 Firefox/14.0.1')
#将浏览器信息加入headers，浏览器信息属性为 User-Agent。爬虫通过User-Agent告诉网站自己的身份
>>> content = urllib.request.urlopen(req).read()
```

HTTP协议：Python模拟POST

- 模拟POST请求：模拟人人网登陆

```python
import urllib.request
import urllib.parse
from http import cookiejar

#1. 构建一个CookieJar对象实例来保存cookie
cookie = cookiejar.CookieJar()
#2. 使用HTTPCookieProcessor()来创建cookie处理器对象，参数为CookieJar()对象
cookie_handler = urllib.request.HTTPCookieProcessor(cookie)
#3. 通过build_opener()来构建opener
opener = urllib.request.build_opener(cookie_handler)
#4. addheaders接受一个列表，里面每个元素都是一个headers信息的元组，opener附带headers信息
opener.addheaders =
[("User-Agent", "Mozilla/5.0 (X11; Fedora; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110Safari/537.36")]
#5. 需要登陆的账号和密码
data = {"email": "18202101965", "password": "sjtujyx"}
#6. 通过urlencode转码
postdata = urllib.parse.urlencode(data).encode("utf8")
#7. 构建Request请求对象，包含需要发送的用户名和密码
request = urllib.request.Request("http://www.renren.com/PLogin.do", data=postdata)
#8. 通过opener发送这个请求，并获取登陆后的Cookie值
opener.open(request)
#9. opener包含用户登陆后的Cookie值，可以直接访问那些登陆后才能访问的页面
response = opener.open("http://www.renren.com/974663514/profile")
fhandle = open('./test3.html','wb')
fhandle.write(response.read())
fhandle.close()
```

HTTP协议：Python模拟POST

- 使用浏览器打开上页代码运行之后得到的test3.html文件，发现内容与我们登录后看到的页面相同，说明python模拟登录成功。

- **重点**：Cookie 是指某些网站服务器为了辨别用户身份和进行Session跟踪，而储存在用户浏览器上的文本文件，Cookie可以保持登录信息到用户下次与服务器的会话。HTTP是无状态的面向连接的协议，为了保持连接状态，引入了Cookie机制。Cookie是http消息头中的一种属性。当你获取一个URL你使用一个opener，在前面，我们都是使用的默认的opener，也就是urlopen。它是一个特殊的opener，可以理解成opener的一个特殊实例，传入的参数仅仅是url，data，timeout。

小练习（不需要提交）

1．自己先注册一下人人网账号（http://www.renren.com/PLogin.d），接着仿照第10页PPT的做法（参考**partA/renren.py**），利用cookie登陆自己的个人主页（http://www.renren.com/974663514/profile）（注：网址中的数字部分每个人各不相同），并打印出自己个人主页中的姓名，学校，性别，生日和现居地址。

提示：（1）姓名的标签形如<a class=”hd-name”>

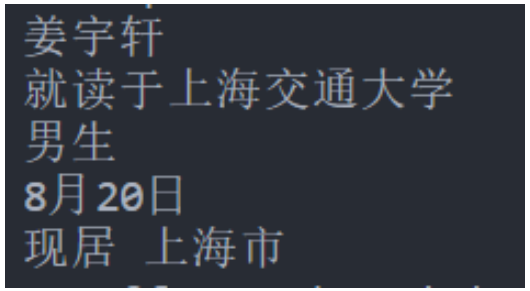
（2）学校的标签形如<li class=”school”>

（2）性别和生日同是标签<li class=”birthday”>的子标签，子标签索引分别为1和3

（3）现居地址的标签形如<li class=”address”>

（4）记得去除生日输出前面的逗号

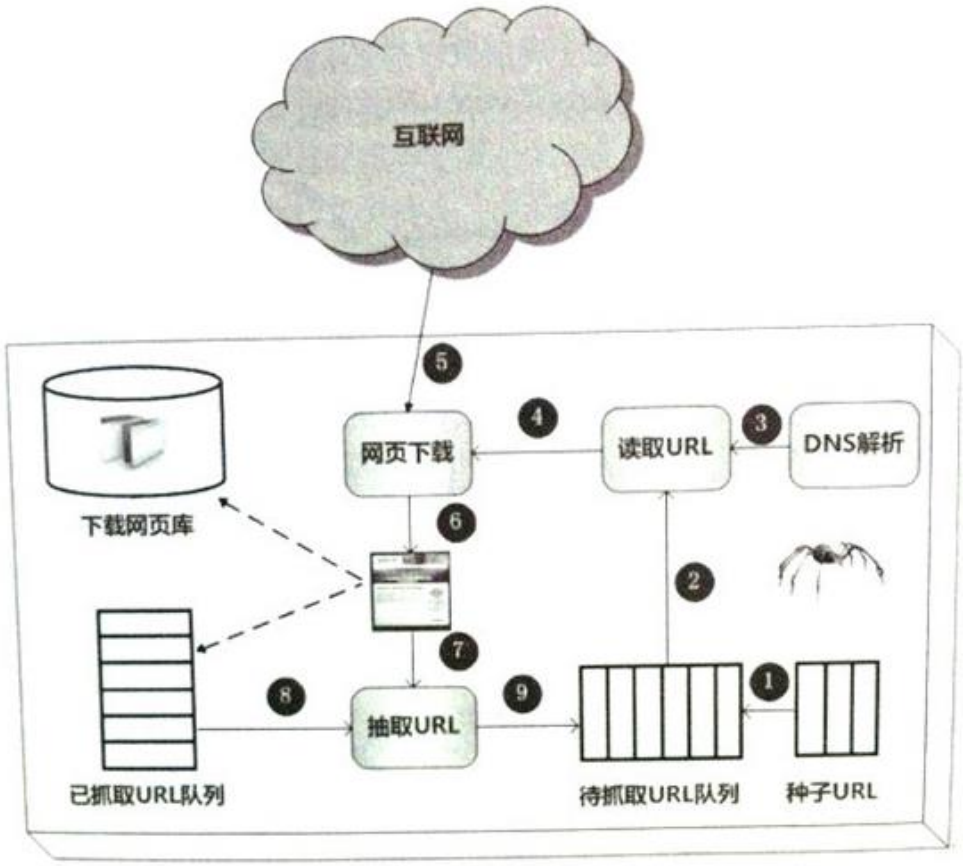
输出结果形式如下：



Part B: 爬虫的概念（选读）

爬虫的概念

- **什么是爬虫：**网络爬虫是一种按照一定的规则，自动抓取万维网信息的程序或者脚本。



爬虫框架

爬虫的概念：Robots.txt

- Robots协议：robots协议也就是robots.txt，网站通过robots协议告诉搜索引擎哪些页面可以抓取，哪些页面不能抓取。Robots协议是网站国际互联网界通行的道德规范，其目的是保护网站数据和敏感信息、确保用户个人信息和隐私不被侵犯。因其不是命令，故需要搜索引擎自觉遵守。

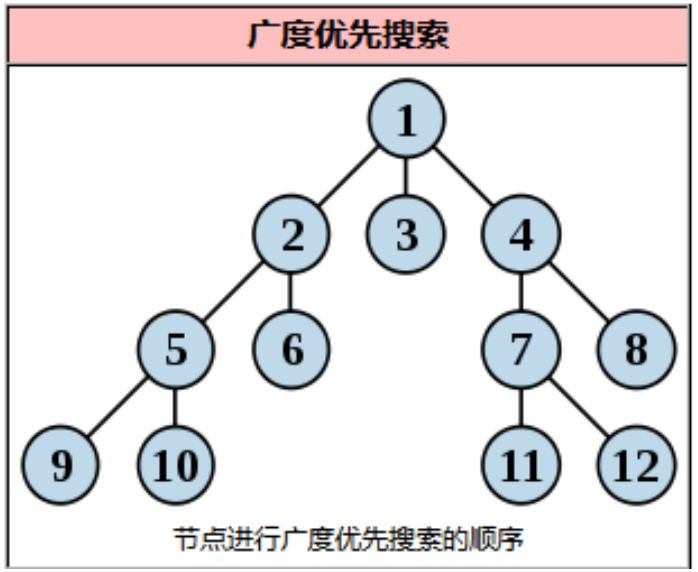
- 一些网站的robots举例

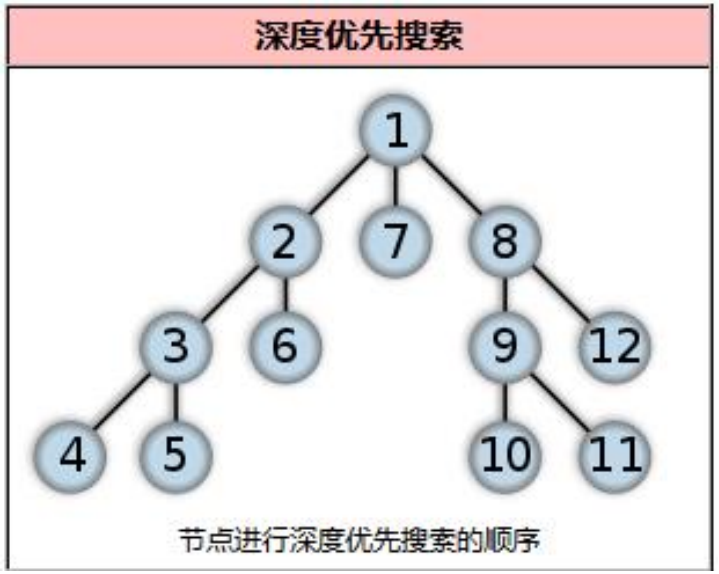
http://www.taobao.com/robots.txt	User-agent: Baiduspider Disallow: /	淘宝屏蔽百度蜘蛛 未屏蔽谷歌 http://tech.163.com/08/0907/07/4L7LDM33000915BF.html
http://renren.com/robots.txt	User-agent: * Allow: / Disallow: /profile.do* Disallow: /getuser.do*	人人禁止搜索引擎爬取个人页面
http://www.360buy.com/robots.txt http://www.suning.com/robots.txt	User-agent: EtaoSpider Disallow: /	京东，苏宁等B2C网站屏蔽一淘比价 http://tech.sina.com.cn/i/2012-08-28/01257551648.shtml

爬虫的概念：抓取策略

- 广度优先搜索策略BFS（Breadth First Search）：BFS是从根节点开始，沿着树的宽度遍历树的节点。如果所有节点均被访问，则算法中止。

- 深度优先搜索策略DFS（Depth First Search）：DFS是沿着树的深度遍历树的节点，尽可能深的搜索树的分支。当节点v的所有边都已被探寻过，搜索将回溯到发现节点v的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。



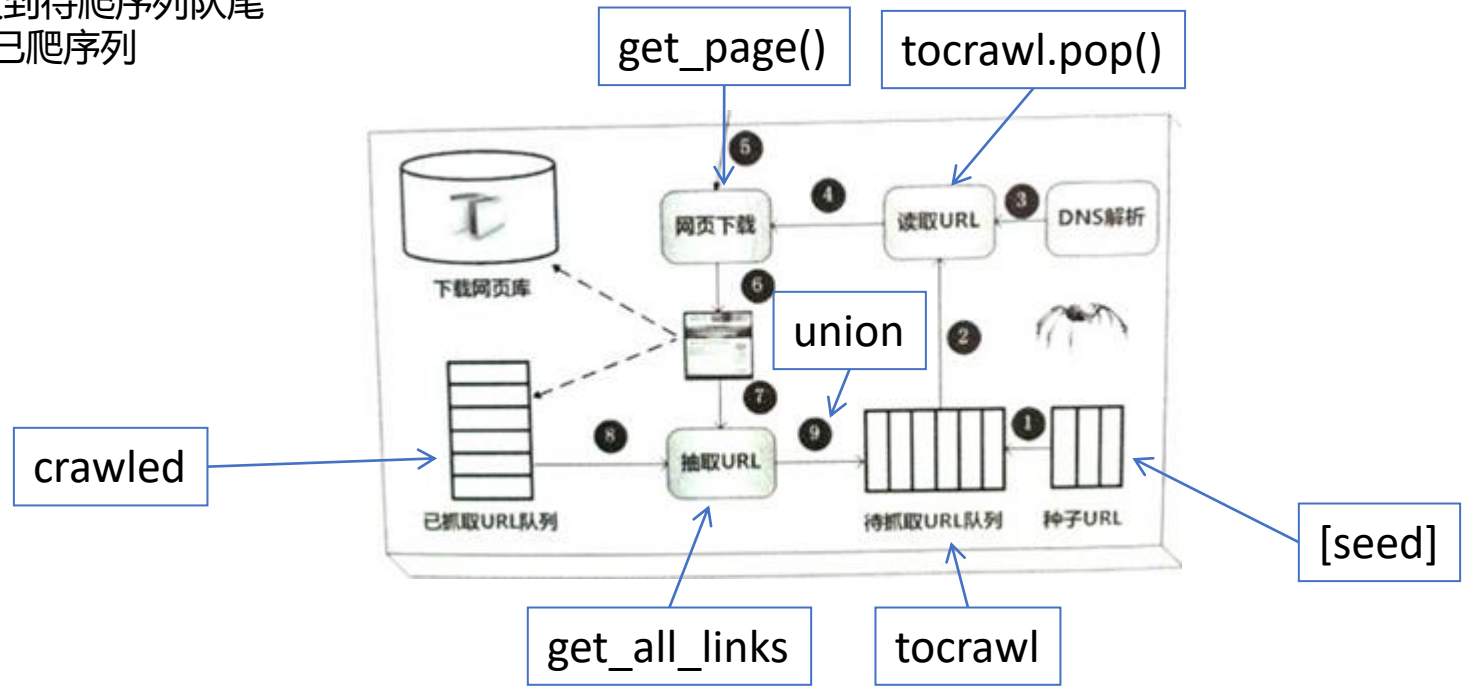


爬虫的概念：抓取策略

- DFS的简单实现（查看crawler_sample.py）

算法的Python代码：

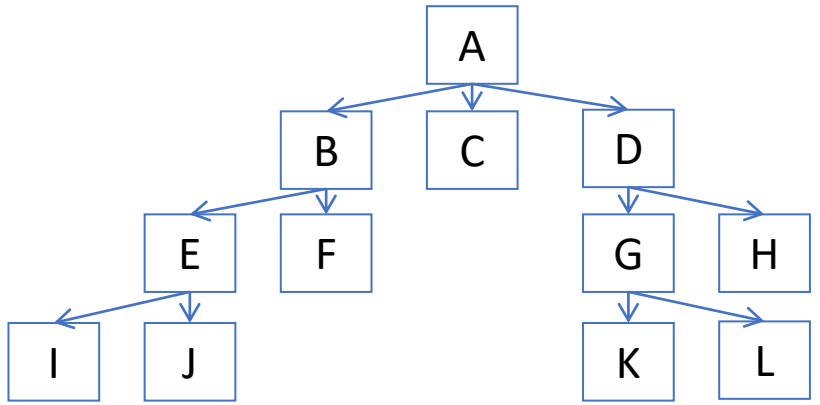
```
tocrawl = [seed]                        #待爬序列初始化为种子序列
crawled = []                            #已爬序列初始化为空
while tocrawl:                          #待爬序列非空
    page = tocrawl.pop()                #弹出待爬堆栈顶端URL
    if page not in crawled:             #如果page未被爬过
        content = get_page(page)                    #爬取page
        outlinks = get_all_links(content)   #找到page中所有链接
        union_dfs(tocrawl, outlinks)        #将outlinks放到待爬序列队尾
        crawled.append(page)                #将page放入已爬序列
return crawled
```



爬虫的概念：抓取策略

- DFS的简单实现

  假设现在有A,B,C...L网页。A→B表示A中有B的链接。将左图表示为字典结构。其中key:[value1, value2...]，表示key中有指向value1,value2的链接。




```
>>> g =  {'A':['B', 'C', 'D'],
'B':['E', 'F'],
'D':['G', 'H'],
'E':['I', 'J'],
'G':['K', 'L']}
```

待爬队列	A	BCD	BCGH	BCG	BCKL	BCK	BC	B	EF	E	IJ	I	
已爬队列		A	AD	ADH	ADHG	ADHGL	ADHGLK	ADHGLKC	ADHGLKCB	ADHGLKCBF	ADHGLKCBFE	ADHGLKCBFEJ	ADHGLKCBFEJI

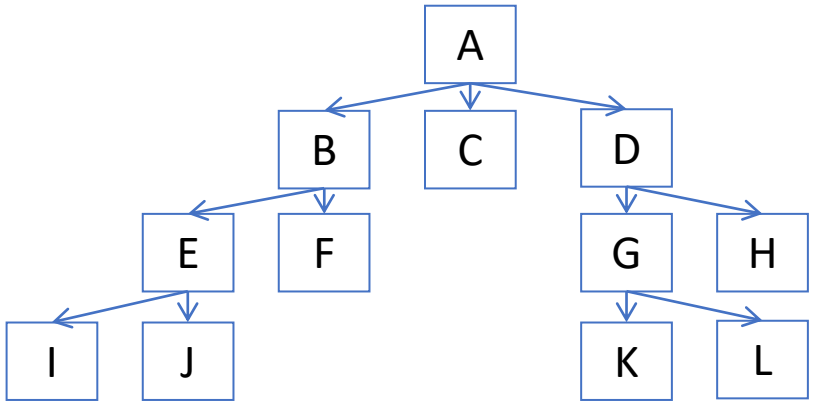
爬取A，将A中链接BCD放入待爬队列队尾

爬取D，将D中链接GH放入待爬队列队尾

DFS

爬虫的概念：抓取策略

BFS与DFS的不同：DFS每次将爬到的链接放到待爬队列队尾，BFS每次将爬到的链接放到待爬队列队首。
（DFS每次爬取最后看到的链接，因此优先往深度方向爬。BFS每次爬取最早看见的链接，因此优先爬层级比较浅的网页。）




```
def union_dfs(a,b):
    for e in b:
        if e not in a:
            a.append(e)      #将链接放到待爬序列队尾
```

待爬队列	A	DCB	FEDC	FED	HGFE	JIHGF	JIHG	LKJIH	LKJI	LKJ	LK	L	
已爬队列		A	AB	ABC	ABCD	ABCDE	ABCDEF	ABCDEFG	ABCDEFGH	ABCDEFGHI	ABCDEFGHIJ	ABCDEFGHIJK	ABCDEFGHIJKL

爬取A，将A中链接BCD放入待爬队列队首

爬取B，将B中链接EF放入待爬队列队首

BFS

小练习（不需要提交）

1. 修改partB/crawler_sample.py中的union_bfs函数，完成BFS搜索

def union_bfs(a,b):

其中a，b为list，函数将b中的元素插在a之前。注意排除重复元素。
例如：
提示：list的insert操作可以将元素插在指定位置。

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

graph结构与crawler_sample.py中g的结构相同。
完成后运行graph, crawled = crawl('A', 'bfs')
查看graph 中的图结构，以及crawled中的爬取结果顺序。

```
>>> graph_dfs, crawled_dfs = crawl('A', 'dfs')
>>> graph_dfs
{'A': ['B', 'C', 'D'], 'C': [], 'B': ['E', 'F'], 'E': ['I', 'J'], 'D': ['G', 'H'
], 'G': ['K', 'L'], 'F': [], 'I': [], 'H': [], 'K': [], 'J': [], 'L': []}
>>> crawled_dfs
['A', 'D', 'H', 'G', 'L', 'K', 'C', 'B', 'F', 'E', 'J', 'I']
>>> graph_bfs, crawled_bfs = crawl('A', 'bfs')
>>> graph_bfs
{'A': ['B', 'C', 'D'], 'C': [], 'B': ['E', 'F'], 'E': ['I', 'J'], 'D': ['G', 'H'
], 'G': ['K', 'L'], 'F': [], 'I': [], 'H': [], 'K': [], 'J': [], 'L': []}
>>> crawled_bfs
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L']
```

Part C: 哈希散列 & Bloom Filter

Hashtable作业**不需要提交**

Bloom filter作业**需要提交**

哈希散列：时间复杂度

•在Python中查询程序运行时间

```
>>> import time
>>> start = time.time()                  #记录开始时间（单位为秒）
>>> ...                                  #需要运行的程序
>>> run_time = time.time() – start       #运行时间 = 结束时间 - 开始时间
```

✓写成函数的形式

```
def time_execution(code):
    start = time.time()
    result = eval(code)        #按需求设定，运行字符串code中的命令（可以是函数）
    run_time = time.time() - start
    return result, run_time  #返回运行结果和时间
```

```
>>> time_execution('1+1')
(2, 5.6355336710112l7e-05)
>>> time_execution('1-1')
(0, 5.461476757773198e-05)
```

✓测试函数的运行时间

```
def loop(n):
    i = 0
    while i<n:
        i += 1
    return i
```

```
>>> time_execution('loop(1000)')
(1000, 0.00013994783512316644)
>>> time_execution('loop(10000)')
(10000, 0.0010387590264l4482)
>>> time_execution('loop(10**5)')
(100000, 0.007880161516368389)
>>> time_execution('loop(10**6)')
(1000000, 0.0834762640442932)
>>> time_execution('loop(10**7)')
(10000000, 0.8340115870778391)
>>> time_execution('loop(10**8)')
```

哈希散列：时间复杂度

- **算法的时间复杂度：**

如果一个问题的规模是n，解这一问题的某一算法所需要的时间为T(n)，它是n的某一函数 T(n)称为这一算法的“时间复杂性”。常用大O表示法表示时间复杂性，在这种描述中使用的基本参数是 n，即问题实例的规模，把复杂性或运行时间表达为n的函数。

✓O(n)

```
def loop(n):
    i = 0          #执行1次
    while i<n:     #执行n次
        i += 1     #执行n次
    return i       ##T(n)=1+n+n=O(n)
```

```
>>> time_execution('loop(10**5)')
(100000, 0.007880161516368389)
>>> time_execution('loop(10**6)')
(1000000, 0.0834762640442932)
>>> time_execution('loop(10**7)')
(10000000, 0.8340115870778391)
>>> time_execution('loop(10**8)')
(100000000, 8.417622597313311)
```

时间以n倍变化

✓$O(n^2)$

```
•def loop_1(n):
•    s = 0                  #1次
•    for i in range(n):     #n次
•        for j in range(n): #n²次
•            s += 1         #n²次
•    return s               #T(n)=1+n+n²+n²=O(n²)
```

```
>>> time_execution('loop_1(10**1)')
(100, 9.262139064958319e-05)
>>> time_execution('loop_1(10**2)')
(10000, 0.0010084520472446457)
>>> time_execution('loop_1(10**3)')
(1000000, 0.08799290228489554)
>>> time_execution('loop_1(10**4)')
(100000000, 9.67143691813908)
```

时间以n^2倍变化

哈希散列：让查找更快

- **爬虫中的时间复杂度**

✓ 爬虫代码中的一段：

```
tocrawl = [seed]
crawled = []                    #已爬序列，list方式存储
while tocrawl:
        page = tocrawl.pop()
        if page not in crawled:    #查看page是否在已爬序列中
                ...                          #抓取page
                crawled.append(page)    #将page加入已爬序列中
```

按list方式存储时，判断page是否在crawled中，程序需要逐个比对crawled中的元素，直到找到为止。

在实际应用中，已爬序列中的网页通常非常多，遍历方式的判断效率不高。

哈希散列：让查找更快

- **爬虫中的时间复杂度**

  模拟最差情况（新的page都不在crawled中，每次都要遍历crawled）

```
def crawl(tocrawl):
    crawled = []
    while tocrawl:
        page = tocrawl.pop()
        if page not in crawled:
            #crawl page
            crawled.append(page)
```

```
tocrawl = [str(i) for i in range(10**2)]
time_execution('crawl(tocrawl)')
```

```
0.000198364257812 5
```

```
tocrawl = [str(i) for i in range(10**3)]
time_execution('crawl(tocrawl)')
```

```
0.010037660598754883
```

```
tocrawl = [str(i) for i in range(10**4)]
time_execution('crawl(tocrawl)')
```

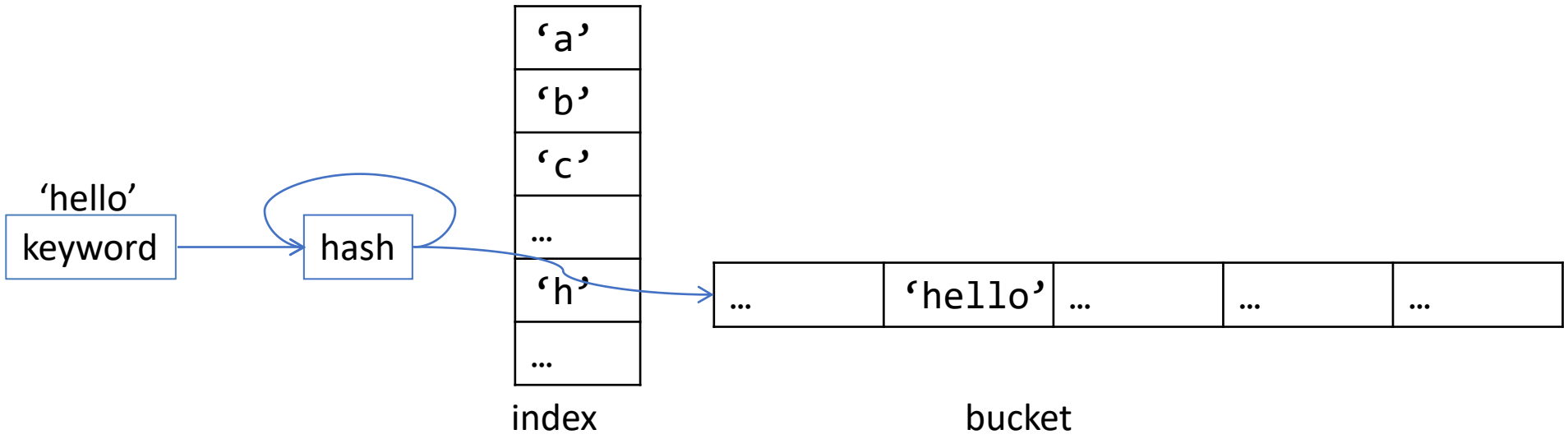
```
0.9217512607574463
```

[str(i) for i in range(n)] 生成['0', '1', '2',...'n-1']的字符串list

哈希散列：让查找更快

Hash Table

- list相当于没有目录的字典，查找单词时需要从第一页开始，一页一页翻，找到为止。
- 我们可以将字符串根据规则（hash function）放在b个不同的bucket里，这样查找时只要在对应的bucket中进行查找。这样可以缩小搜索范围。理想情况下可以将搜索范围减小到原来的1/b。
- 英语字典的hash function是 单词首字母。给定keyword，就从单词首字母所在的bucket进行查找。



哈希散列：让查找更快

Hash Function

我们定义hash function的输入为字符串keyword，输出为一个数，该数字告诉我们应该在第几个bucket中查找keyword。

Q: 假设有k个keyword，b个bucket，k>b。Hash function应该有以下哪些性质?

a.输出0~k-1范围的数字

b.输出0~b-1范围的数字

c.将大约k/b个keywords映射到第0个bucket

d.将大约k/b个keywords映射到第1个bucket

e.映射到第0个bucket的keywords比映射到第1个bucket中的多

Answer: b, c, d.

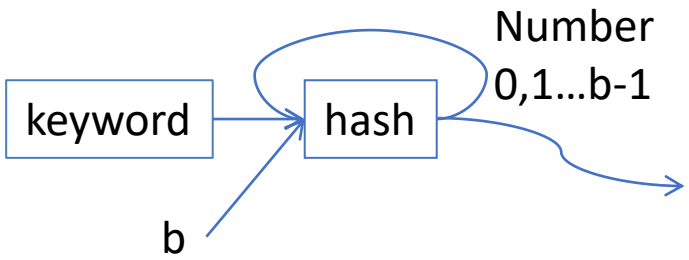
a.k-1超过了bucket数的最大范围

e.理想情况下，hash function应该将keywords平均的映射到各个bucket中，这样才能保证平均搜索范围是原来的1/b。

哈希散列：让查找更快

Hash Function

hash function的输入为字符串keyword，bucket数b，输出为0~b-1范围内的数。



✓一个简单的Hash function

```
def simple_hash_string(keyword, b):
    if keyword != '':
        return ord(keyword[0])% b          #ord(s)返回字符s的ASCII码，例如ord('A')=65 。
    else:                                  #ord(s)是chr(i)的逆。
        return 0
```

Q: 这一hash function是否满足前面的性质?
a.输出0~b-1范围的数字
b.将keywords平均分布到不同的bucket中

Answer:
a.满足。%b 将数字变为0~b-1范围内
b.不满足，在英文里，某些字母开头的单词可能更多。

哈希散列：让查找更快

Hash Function

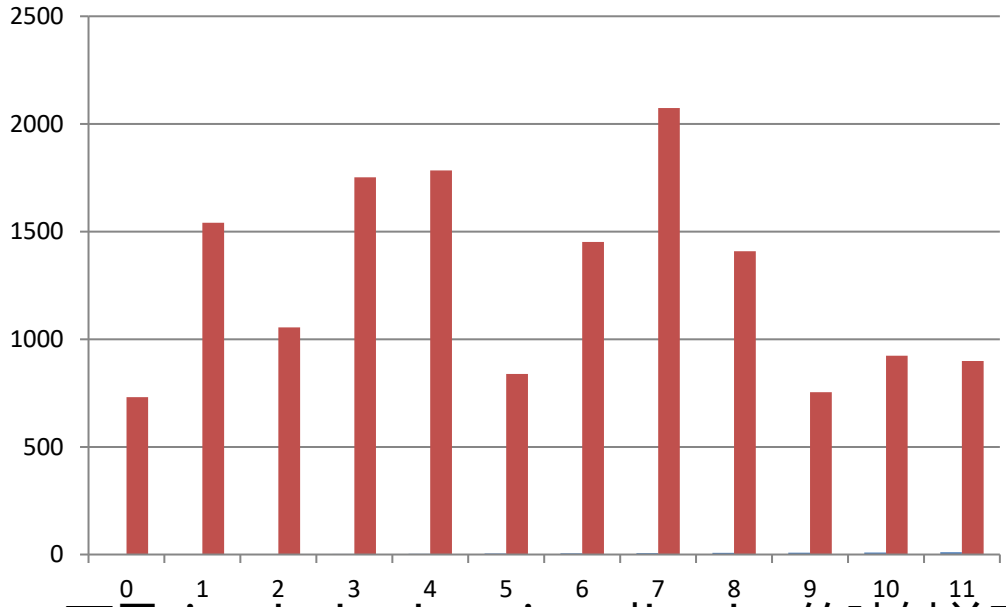
将英文文本中的单词提取出（已经排除重复单词），测试Hash Function是否将keywords平均分布到不同的bucket中。

```
def test_hash_function(func, size, filename):
#func为需要测试的hash function, size为bucket数，filename为英文文本
    words = []
    f = open(filename,'r')
    for line in f.readlines():      #逐行读取文本
        for word in line.strip().split(' '):    #以空格分割，将文本变为词
            words.append(word)      #将词加入words
    f.close()
    results = [0] * size        #统计不同bucket中的字符串数
    words_used = []
    for word in words:
        if word not in words_used:  #已经统计过的word不再统计
            bucket = func(word, size)
            results[bucket] += 1
            words_used.append(word)
    return results
```

哈希散列：让查找更快

- Hash Function

```
>>> test_hash_function(simple_hash_string, 12, 'pg1661.txt')
[731, 1541, 1055, 1752, 1784, 839, 1452, 2074, 1409, 754, 924, 899]
```



可见simple_hash_string对bucket的映射并不均匀。你能设计一个更好的hash_function吗?

```
def hash_string(keyword,b):          #输入为keyword, bucket数b，输出为0~b-1的数
    ...
    return number
```

哈希散列：让查找更快

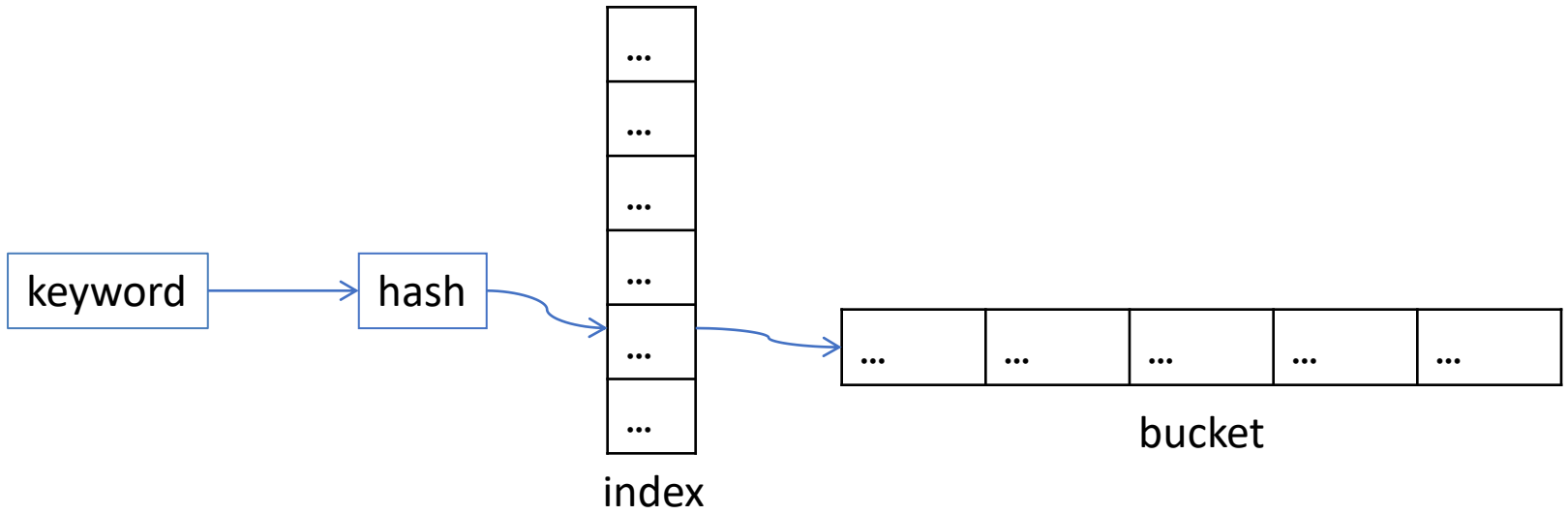
- **Hash Table的存储结构**

  Q: hash table的存储结构应该如何表示

  a.[<word>, <word>,...]

  b.[[<word>], [<word>],...]

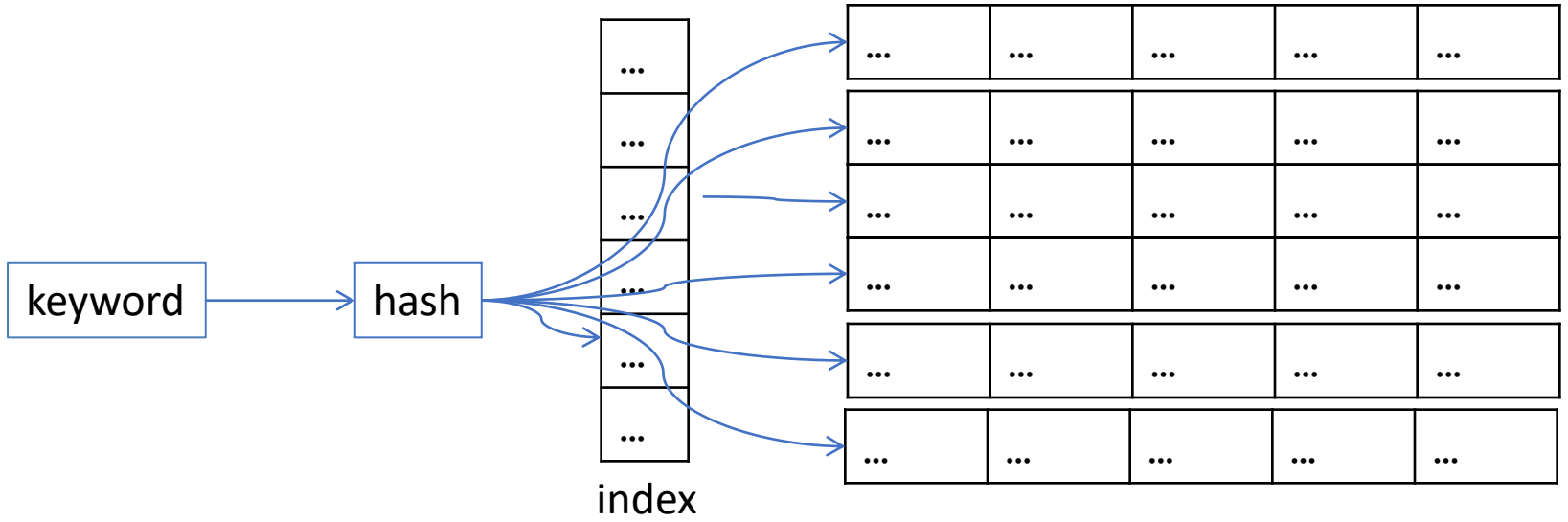
  c.[[<word>, <word>,...], [<word>, <word>,...],...]



Answer: c. hash table的每个bucket以list方式存储。

哈希散列：让查找更快

- 初始化Hash Table



请完成初始化Hash Table的函数
def make_hashtable(b): #输入为bucket数b，输出为空的Hash Table
    ...
    return table
例如

```
>>> table = make_hashtable(4)
>>> table
[[], [], [], []]
```

哈希散列：让查找更快

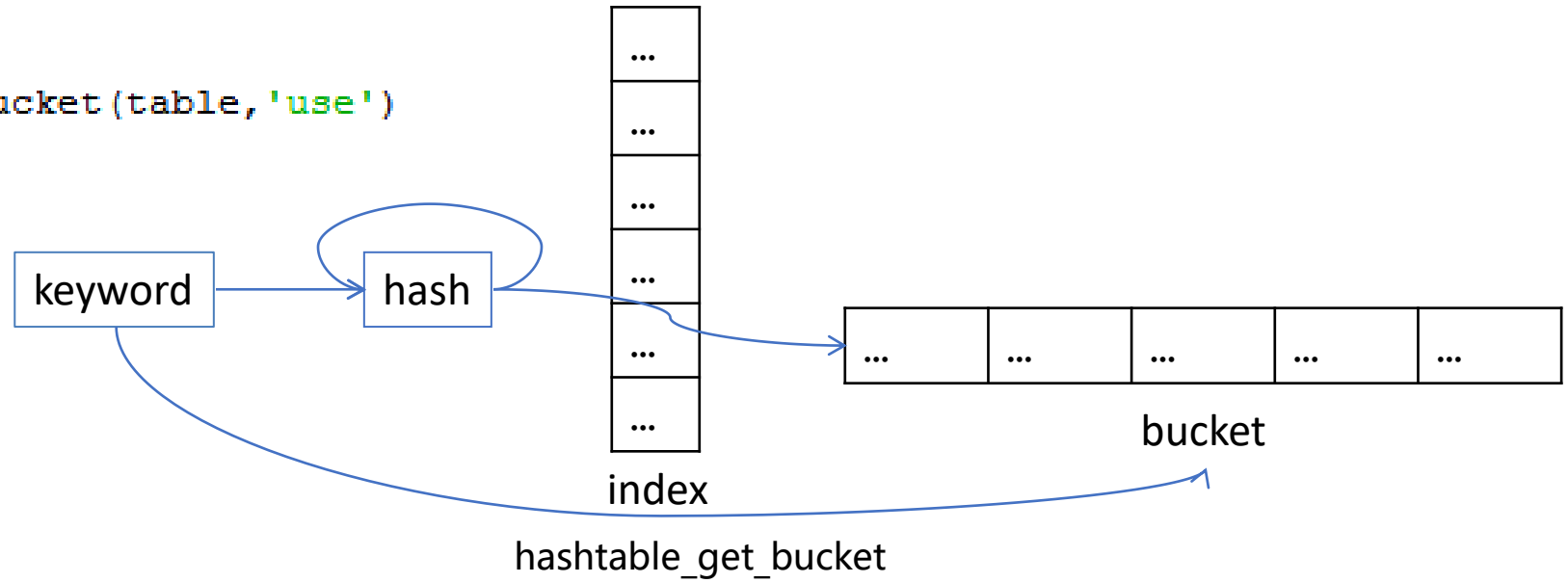
- 从Hash Table中得到bucket

  请完成返回bucket的函数

  def hashtable_get_bucket(table,keyword):　　#输入为Hash Table，keyword

  　return …　　#输出为bucket。keyword不在table中，也应返回hash function计算的bucket

```
>>> table
[['this', 'is', 'anyone'], ['the'], ['for'], ['ebook', 'use', 'of'], []]
>>> bucket = hashtable_get_bucket(table,'user')
>>> bucket
['for']
>>> bucket = hashtable_get_bucket(table,'use')
>>> bucket
['ebook', 'use', 'of']
```



哈希散列：让查找更快

- 查找HashTable中的元素

  请完成查找函数

  def hashtable_lookup(table,keyword):　　#输入为Hash Table，keyword

  　　return …　　#keyword在Table中输出True，否则输出False

```
>>> table
[['this', 'is', 'anyone'], ['the'], ['for'], ['ebook', 'use', 'of'], []]
>>> hashtable_lookup(table,'user')
False
>>> hashtable_lookup(table,'use')
True
```

哈希散列：让查找更快

- 添加keyword到HashTable中

请完成添加元素的函数

def hashtable_add(table,keyword):　　#输入为HashTable，keyword

　　...　　#将keyword添加到Table中，注意函数内部不用做keyword是否在#Table中的判定（而在函数外部进行判断）。将keyword添加到Table中

在添加keyword前，请先用lookup查看keyword是否在hashtable中。

（为什么需要把判定放在hashtable_add外面？）

```
>>> table = [['this', 'is', 'anyone'], ['the'], ['for'], ['ebook', 'use', 'of'], []]
>>> table
[['this', 'is', 'anyone'], ['the'], ['for'], ['ebook', 'use', 'of'], []]
>>> keyword = 'use'
>>> if not hashtable_lookup(table, keyword):
        hashtable_add(table, keyword)


>>> table
[['this', 'is', 'anyone'], ['the'], ['for'], ['ebook', 'use', 'of'], []]
>>> keyword = 'user'
>>> if not hashtable_lookup(table, keyword):
        hashtable_add(table, keyword)


>>> table
[['this', 'is', 'anyone'], ['the'], ['for', 'user'], ['ebook', 'use', 'of'], []]
```

哈希散列：让查找更快

- 设计实验对比list和HashTable的速度 （不需要提交）

  例如：
  可以将 crawl(tocrawl) 的list形式的crawled修改为hashtable，对比速度。
  也可设计别的实验。

- 许多代码语言中的dict和set就是用hashtable实现的。

BloomFilter

- 在爬虫中，用HashTable记录已爬URL太消耗内存。随着URL的增多，占用的内存会越来越多。就算只有1亿个URL，每个URL只算50个字符，就需要5GB内存。

- Gmail等Email提供商，需要过滤垃圾邮件。一个办法就是记录下那些发垃圾邮件的 email 地址。每存储一亿个 email 地址， 就需要 1.6GB 的内存。而全世界至少有几十亿个发垃圾邮件的地址。

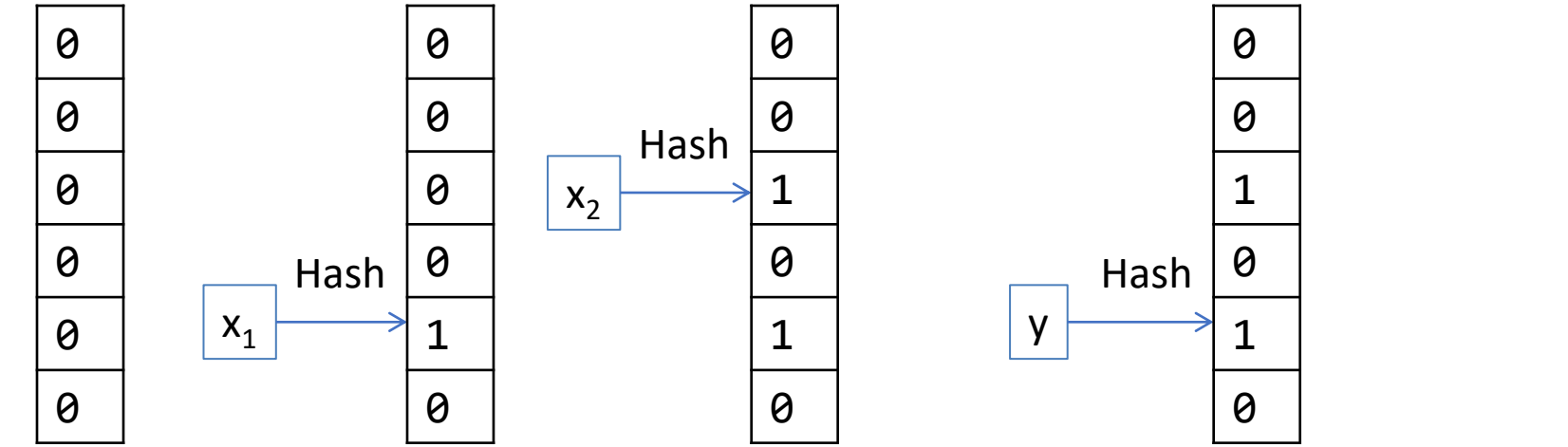
- 在上述应用中，需要快速判断某个元素是否属于集合，但是并不严格要求100%正确。例如将未爬网页误判为已爬网页的代价只是少爬几个网页；将正常邮件的地址误判为垃圾邮件地址，可以用通过建立白名单（存储那些可能误判的邮件地址）的方式补救。另外，我们不关心集合里具体有哪些元素。例如我们不关心垃圾邮件集合里具体有哪些垃圾邮件地址。

BloomFilter

- 一个简单方案

  Bit-Map方法：建立一个BitSet，将每个URL经过一个哈希函数映射到某一位。

  初始状态时，BitSet是一个包含m位的位数组，每一位都置0。添加元素x时，通过哈希函数将x映射到0~m-1的某个位置h(x)，将该位置置1。查找元素y时，对y用哈希函数，如果h(y)位置为1，则很有可能y属于集合。如果h(y)位置为0，则y肯定不属于集合。



初始状态　　添加x_1　　添加x_2

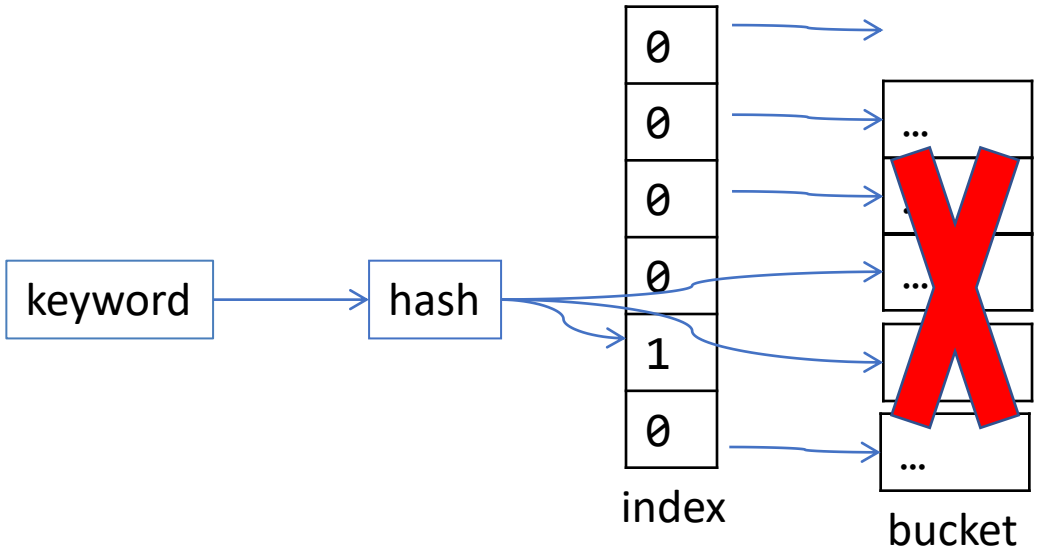
判断y是否属于集合，由于
y经过Hash后的映射位是1，
y有很大概率属于集合{x_1, x_2}

BloomFilter

- 通过HashTable解释

在HashTable中，假设有n个keyword，m个bucket，我们取m>>n。这样我们给index增加0,1属性，一旦添加元素x，则index中的h(x)位置置1。当我们要判断y是否在集合中，假如h(y)所在的位置index为1，由于m>>n，一个bucket中存储1个以上keyword的概率很小，在不查看bucket的情况下我们可以判断y很可能在集合中。

Bit-Map相当于没有bucket，而且m>>n的HashTable。由于没有存储keyword的bucket，bitmap更节省空间。若要降低冲突发生的概率到1%，就要将BitSet的长度m设置为keyword个数n的100倍。

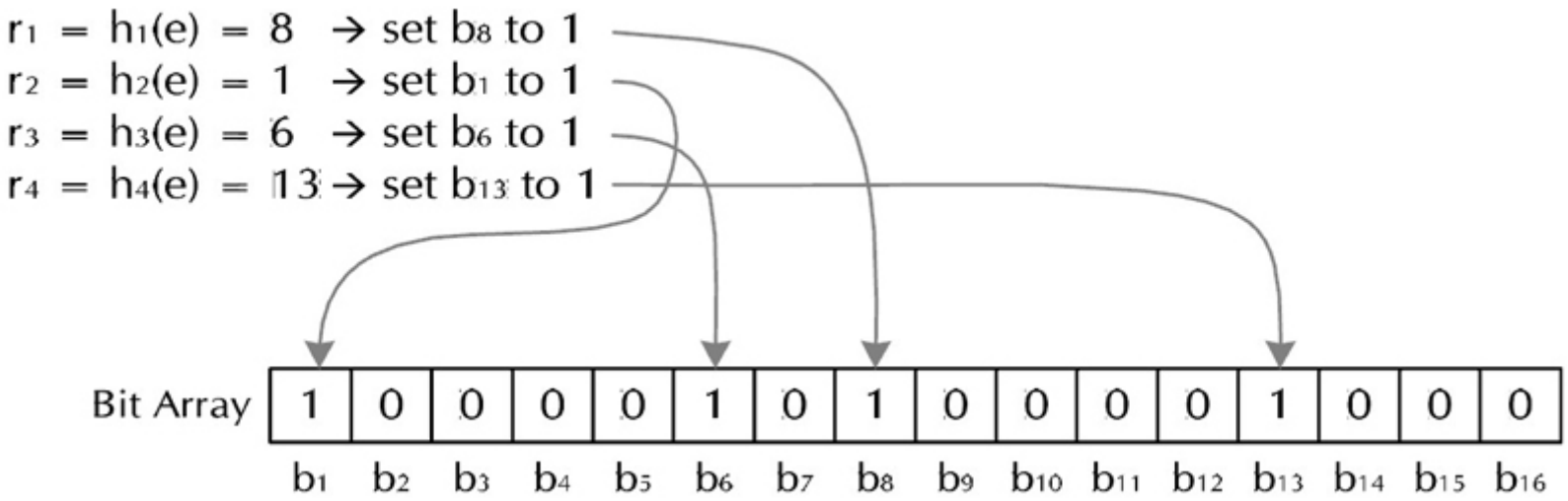


BloomFilter：实现

- BitMap缺点是冲突概率高，为了降低冲突的概念，Bloom Filter使用了k个哈希函数，而不是一个。假设有k个哈希函数，位数组大小为m，加入keyword的数量为n。

1. 加入keyword e的过程

   k个哈希函数记为h_1, h_2, ..., h_k, 对于keyword e，分别计算$h_1(e)$，$h_2(e)$，...，$h_k(e)$，然后将BitSet的$h_1(e)$，$h_2(e)$，...，$h_k(e)$位设为1。



图中k=4, m=16

这样就将keyword e映射到BitSet中的k个二进制位了。

BloomFilter：实现

2. 检查keyword str是否存在的过程

   对于keyword str，分别计算$h_1(str)$，$h_2(str)$，$h_3(str)$，$h_4(str)$。然后检查BitSet的第$h_1(str)$，$h_2(str)$，$h_3(str)$，$h_4(str)$位是否为1，若其中任何一位不为1则可以判定str一定没有被记录过。若全部位都是1，则“认为”字符串str存在。

   若一个字符串对应的Bit不全为1，则可以肯定该字符串一定没有被BloomFilter记录过。（这是显然的，因为字符串被记录过，其对应的二进制位肯定全部被设为1了）

   但是若一个字符串对应的Bit全为1，实际上是不能100%的肯定该字符串被BloomFilter记录过的。（因为有可能该字符串的所有位都刚好是被其他字符串所对应）这种将该字符串划分错的情况，称为false positive 。

- 不同于其他结构，BloomFilter中的keyword加入了就被不能删除了，因为删除会影响到其他keyword 。
- BloomFilter跟单哈希函数Bit-Map不同之处在于：Bloom Filter使用了k个哈希函数，每个字符串跟k个bit对应。从而降低了冲突的概率。
- BloomFilter的在线演示

➢ http://billmill.org/bloomfilter-tutorial/

➢ http://www.jasondavies.com/bloomfilter/

BloomFilter：参数选择

- 哈希函数选择

哈希函数的选择对性能的影响应该是很大的，一个好的哈希函数要能近似等概率的将字符串映射到各个Bit。选择k个不同的哈希函数比较麻烦，一种简单的方法是选择一个哈希函数，然后送入k个不同的参数。

参考(General Purpose Hash Function Algorithms，即http://www.partow.net/programming/hashfunctions/)中给出了几种Hash Function的写法（GeneralHashFunctions_-_Python.zip文件）。使用时可以选择多个哈希函数。也可以选择其中一种，给定不同的seed生成多个哈希函数。

例如其中的BKDRHash，通过改变seed可以得到不同的哈希函数

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

使用时通过取模得到0~m-1的hash输出

```
>>> keyword = '123'
>>> m = 100
>>> BKDRHash(keyword)%m
82
```

BloomFilter：参数选择

- 哈希函数个数k、位数组大小m选择

  哈希函数个数k、位数组大小m、加入的字符串数量n的关系可以查看(BloomFilters- the math)。该文献证明了对于给定的m、n，当 k = ln(2)* m/n 时出错的概率是最小的。

  同时该文献还给出特定的k，m，n的出错概率。例如：根据(BloomFilters- the math)，哈希函数个数k取10，位数组大小m设为字符串个数n的20倍时，false positive发生的概率是0.0000889，这个概率基本能满足爬虫的需求了。

- Python中对位数组的操作

  Bitarray.py提供了一个位数组操作的类。

提供了初始化，set和get操作。set操作将该位置的值设为1。get得到该位置的值（1为True，0为False）

```
>>> from Bitarray import Bitarray        #导入位数组库
>>> bitarray_obj = Bitarray(32000)       #初始化一个长度是32000的位数组
>>> bitarray_obj.set(3)                  #将第三个位置设置为1
>>> bitarray_obj.get(3)                  #得到第三个位置的值，
True
>>> bitarray_obj.get(4)                  #得到第四个位置的值
False
>>> bitarray_obj.set(4)                  #将第四个位置设置为1
>>> bitarray_obj.get(4)                  #得到第四个位置的值
True
```

→ 练习1

Part D: 多线程爬虫

如何构建一个简单的爬虫？

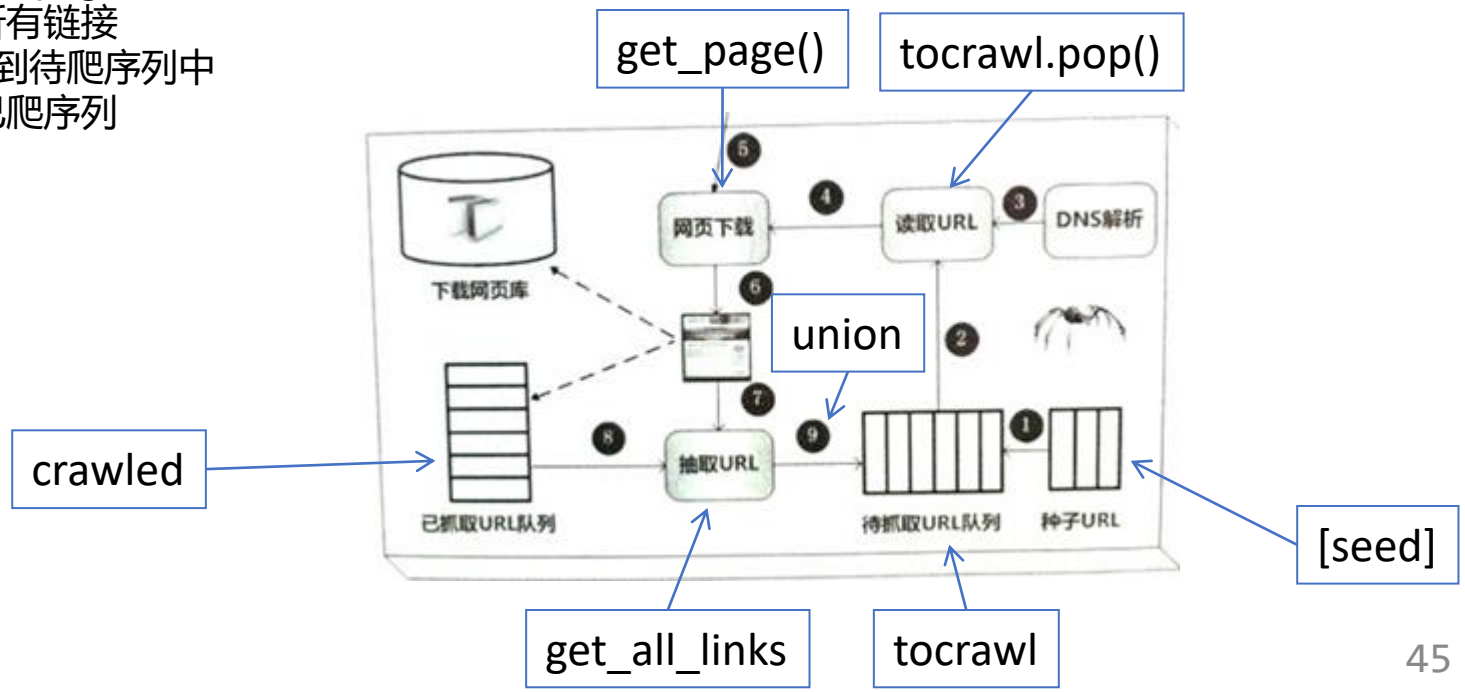
一个简单的爬虫实现步骤（查看crawler_sample.py）：

→ 练习2

- 从一个种子网页出发（例如www.sjtu.edu.cn），找出该网页上的所有链接。访问其中一个链接，得到更多的新链接；然后再访问一个之前没有访问过的链接，继续得到新链接…如此循环下去，就能爬取到大量网页。

算法的伪代码：

```
tocrawl = [seed]                          #待爬序列初始化为种子序列
crawled = []                              #已爬序列初始化为空
while tocrawl:                            #待爬序列非空
    page  = tocrawl.pop()                 #从待爬堆栈中取出一个page，并把它从tocrawl中删除
    if page not in crawled:               #如果page未被爬过
        content = get_page(page)                        #爬取page
        outlinks = get_all_links(content)   #找到page中所有链接
        union操作: add outlinks into tocrawl #将outlinks放到待爬序列中
        add page into crawled             #将page放入已爬序列
return crawled
```



并发编程

- 迅雷等下载工具可以设置线程数。它会将文件分成与线程数相同的部分，然后每个线程下载自己的那一部分，这样下载效率就有可能提高。爬虫程序也是如此。

- 假设每个网页需要下载0.5s，在crawler_sample.py中将get_page函数修改为

```
def get_page(page):
    print  ('downloading page %s' % page)
    time.sleep(0.5)                          #等待0.5s
    return g.get(page, [])
```

这样一个串行的爬虫抓取12个网页大约要6s。

可以让爬虫并发爬取网页加快速度。

```
downloading page A
downloading page D
downloading page H
downloading page G
downloading page L
downloading page K
downloading page C
downloading page B
downloading page F
downloading page E
downloading page J
downloading page I
6.44135764948
```

并发编程

- Python中并发编程可以通过threading和queue两个模块来实现。threading用来操作线程，queue用来维护任务队列。
- 下面我们来对比一下单线程和多线程的运行时间，先看一个单线程的例子。在程序的主线程中依次执行两个任务，任务1时间为4秒，任务2时间为2秒。由于两个任务是依次执行，从控制台输出可以看出所有任务完成总共花费了6秒。

```
import time;

#定义任务1
def work1():
    print("任务1开始了：",time.ctime());
    time.sleep(4)
    print("任务1结束了：",time.ctime());
#定义任务2
def work2():
    print("任务2开始了：", time.ctime());
    time.sleep(2)
    print("任务2结束了：", time.ctime());
#定义主程序 在主程序中依次执行两个任务
def main():
    print("主函数开始了：",time.ctime());
    work1();
    work2();
    print("主函数结束了：", time.ctime());
#执行主程序
main();
```

```
主函数开始了：  Fri Jun 26 15:05:30 2020
任务1开始了：  Fri Jun 26 15:05:30 2020
任务1结束了：  Fri Jun 26 15:05:34 2020
任务2开始了：  Fri Jun 26 15:05:34 2020
任务2结束了：  Fri Jun 26 15:05:36 2020
主函数结束了：  Fri Jun 26 15:05:36 2020
```

并发编程

- 再将上述例子改成多线程。在主程序中单独开启两个线程，将两个任务分别放在两个线程中去执行。这样两个任务并发执行，可以看出所有任务完成花费了4秒。因为在执行任务1时任务2也在执行，两个任务之间不需要等待。

```python
import time;
import threading;
def work1(in1):
    print("任务1开始啦：{0}\n".format(time.ctime()));
    print("我是参数",in1);
    time.sleep(4);
    print("任务1结束了：{0}".format(time.ctime()));
def work2(in1,in2):
    print("任务2开始啦：{0}\n".format(time.ctime()));
    print("我是参数",in1,in2);
    time.sleep(2);
    print("任务2结束了：{0}".format(time.ctime()));
def main():
    print("主程序开始啦：{0}".format(time.ctime()));
    #创建子线程
    t1 = threading.Thread(target=work1,args=("SJTU",));
    t2 = threading.Thread(target=work2, args=("SJTU","JYX"));
    #启动子线程
    t1.start();
    t2.start();
    #等子线程执行完毕再退出
    t1.join();
    t2.join();
    print("主程序结束了：{0}".format(time.ctime()));
main();
```

```
主程序开始啦：Fri Jun 26 15:14:52 2020
任务1开始啦：Fri Jun 26 15:14:52 2020

我是参数 SJTU
任务2开始啦：Fri Jun 26 15:14:52 2020

我是参数 SJTU JYX
任务2结束了：Fri Jun 26 15:14:54 2020
任务1结束了：Fri Jun 26 15:14:56 2020
主程序结束了：Fri Jun 26 15:14:56 2020
```

并发编程

Python3 通过两个标准库 _thread 和 threading 提供对线程的支持。_thread 提供了低级别的、原始的线程以及一个简单的锁，它相比于 threading 模块的功能还是比较有限的。threading 模块除了包含 _thread 模块中的所有方法外，还提供的其他方法：

- threading.currentThread(): 返回当前的线程变量。
- threading.enumerate(): 返回一个包含正在运行的线程的list。正在运行指线程启动后、结束前，不包括启动前和终止后的线程。
- threading.activeCount(): 返回正在运行的线程数量，与len(threading.enumerate())有相同的结果。

除了使用方法外，线程模块同样提供了Thread类来处理线程，Thread类提供了以下方法：

- **run()**: 用以表示线程活动的方法。
- **start()**:启动线程活动。
- **join([time])**: 等待至线程中止。这阻塞调用线程直至线程的join() 方法被调用中止-正常退出或者抛出未处理的异常-或者是可选的超时发生。
- **isAlive()**: 返回线程是否活动的。
- **getName()**: 返回线程名。
- **setName()**: 设置线程名。

并发编程

如何使用Threading模块创建线程

1. 直接实例化threading.Thread线程对象,实现多线程

```python
import threading
import time
def print_age(who, age):
    """
    需要用多线程调用的函数
    :param who:
    :param age:
    :return:
    """
    print("Hello,every one!")
    time.sleep(1)
    print("%s is %s years old !" % (who, age))

if __name__ == "__main__":
    t1 = threading.Thread(target=print_age, args=("jet", 18, ))      # 创建线程1
    t2 = threading.Thread(target=print_age, args=("jack", 25, ))     # 创建线程2
    t3 = threading.Thread(target=print_age, args=("jack", 25,))      # 创建线程3
    t1.start()    # 运行线程1
    t2.start()    # 运行线程2
    t3.start()    # 运行线程3
    print("over...")
```

如何使用Threading模块创建线程

2．通过继承threading.Thread，并重写run()方法，来实现多线程

```python
import threading
import time
class MyThread(threading.Thread):
    """
    使用继承的方式实现多线程
    """
    def __init__(self, who):
        super().__init__()    # 必须调用父类的构造方法
        self.name = who

    def run(self):
        print("%s is run..." % self.name)
        time.sleep(3)

if __name__ == "__main__":
    t1 = MyThread("Jet")    # 创建线程1
    t2 = MyThread("Jack")   # 创建线程2
    t1.start()              # 运行线程1
    t2.start()              # 运行线程2
    print("over...")
```

守护线程与非守护线程

- 当主线程执行完毕时，守护线程也会中止执行（陪葬），而非守护线程与主线程是彼此独立的，主线程结束之后非守护线程会继续执行。
- 创建一个非守护线程，由输出结果可以看出当主线程结束时，子线程还在休眠，而子线程休眠过后会继续执行。

```
import time;
import threading;
def fun():
    print("子线程开始执行");
    time.sleep(4);
    print("子线程执行完毕");

print("主线程开始执行");

#执行守护线程
t = threading.Thread(target=fun,args=());
t.start();

#主线程休眠两秒
time.sleep(2);
print("主线程执行结束");
```

```
主线程开始执行
子线程开始执行
主线程执行结束
子线程执行完毕
```

守护线程与非守护线程

- 创建一个守护线程，通过setDaemon(True)函数设置守护线程。由输出结果可以看出当主线程结束时，子线程还在休眠，但子线程休眠过后不会执行，它会随着主线程的中止而中止。

```python
import time;
import threading;

def fun():
    print("子线程开始执行");
    time.sleep(4);
    print("子线程执行完毕");

print("主线程开始执行");

#执行守护线程
t = threading.Thread(target=fun,args=());
t.setDaemon(True);
t.start();

#主线程休眠两秒
time.sleep(2);
print("主线程执行结束");
```

```
主线程开始执行
子线程开始执行
主线程执行结束
```

并发编程

线程间共享变量问题

- 当多个线程在同一时刻同时访问同一个变量时，就会产生不可预期的错误，这称为共享变量问题。
- 先看一个例子。定义一个全局变量，任务1是引用这个变量让它自增1000000次，任务2是引用这个变量让它自减1000000次，让这两个任务依次执行。

```
1   import threading;
2   count = 0;
3   def work1():
4       global count;
5       for i in range(0,1000000):
6           count = count + 1;
7           # print("任务1: count={0}\n".format(count));
8   def work2():
9       global count;
10      for i in range(0,1000000):
11          count = count - 1;
12          # print("任务2: count={0}\n".format(count));
13  t1 = threading.Thread(target=work1,args=());
14  t2 = threading.Thread(target=work2, args=());
15  #启动子线程
16  t1.start();
17  t2.start();
18  #等子线程执行完毕再退出
19  t1.join();
20  t2.join();
21  print("最终结果: {0}".format(count))
```

```
最终结果: 142532
最终结果: -469330
最终结果: 180608
最终结果: -280336
最终结果: -119930
```

我们看输出结果就会发现，count的最终值几乎不会是0，而且每一次运行的结果都相差很大，这就证明了多线程共享变量的问题确实是存在的。

并发编程

通过加锁来解决共享变量问题

- 锁其实是一个标志，它表示一个变量正在占用一些资源。当一个线程要访问共享资源前，先申请锁，等访问结束后再释放锁。当一个线程申请锁后，其他线程就不能访问这个共享资源，只有等待这个线程释放锁之后才能访问。

```
import threading;
count = 0;
lock = threading.Lock();    #创建一个Lock对象
def work1():
    global count;
    for i in range(0,1000000):
        #申请锁
        lock.acquire();
        count = count + 1;
        #释放锁
        lock.release(); # print("任务1: count={0}\n".format(count));
def work2():
    global count;
    for i in range(0,1000000):
        lock.acquire(); #申请锁
        count = count - 1;
        lock.release();  #释放锁
        # print("任务2: count={0}\n".format(count));
t1 = threading.Thread(target=work1,args=());
t2 = threading.Thread(target=work2, args=());
#启动子线程
t1.start();
t2.start();
#等子线程执行完毕再退出
t1.join();
t2.join();
print("最终结果: {0}".format(count))
```

死锁问题

- 当线程1占用了锁1，线程2占用了锁2，此时两个线程都想申请对方的锁，都在等着申请到对方的锁才肯释放自己的锁，这样两个线程就会一直盲等下去，产生死锁问题

```
def work1():
    print("-----线程1开始运行");
    lock1.acquire();
    print("线程1申请了锁1");
    time.sleep(1);
    print("线程1等待其他线程释放锁2");
    lock2.acquire();
    print("线程1申请了锁2");
    # time.sleep(1);
    lock2.release();
    print("线程1释放了锁2");
    lock1.release();
    print("线程1释放了锁1");
def work2():
    print("-----线程2开始运行");
    lock2.acquire();
    print("线程2申请了锁2");
    time.sleep(1);
    print("线程2等待其他线程释放锁1");
    lock1.acquire();
    print("线程2申请了锁1");
    # time.sleep(2);
    lock2.release();
    print("线程2释放了锁1");
    lock1.release();
    print("线程2释放了锁2");
```

```
if __name__ == '__main__':
    print("主程序启动")
    t1 = threading.Thread(target=work1, args=());
    t2 = threading.Thread(target=work2, args=());
    t1.start();
    t2.start();
    t1.join();
    t2.join();
    print("主程序结束");
```

```
主程序启动
-----线程1开始运行
线程1申请了锁1
-----线程2开始运行
线程2申请了锁2
线程1等待其他线程释放锁2
线程2等待其他线程释放锁1
```

死锁问题的解决：最简单的办法就是，尽量不要让一个线程申请多个锁。或者设置时间限制，如果线程占用锁太长时间就中止这个线程。

并发编程

queue队列

- queue.Queue：对应队列类（FIFO先进先出）
- queue.LifoQueue：对应后进先出队列类
- queue.PriorityQueue：优先级队列
- queue.SimpleQueue：无边界FIFO简单队列类

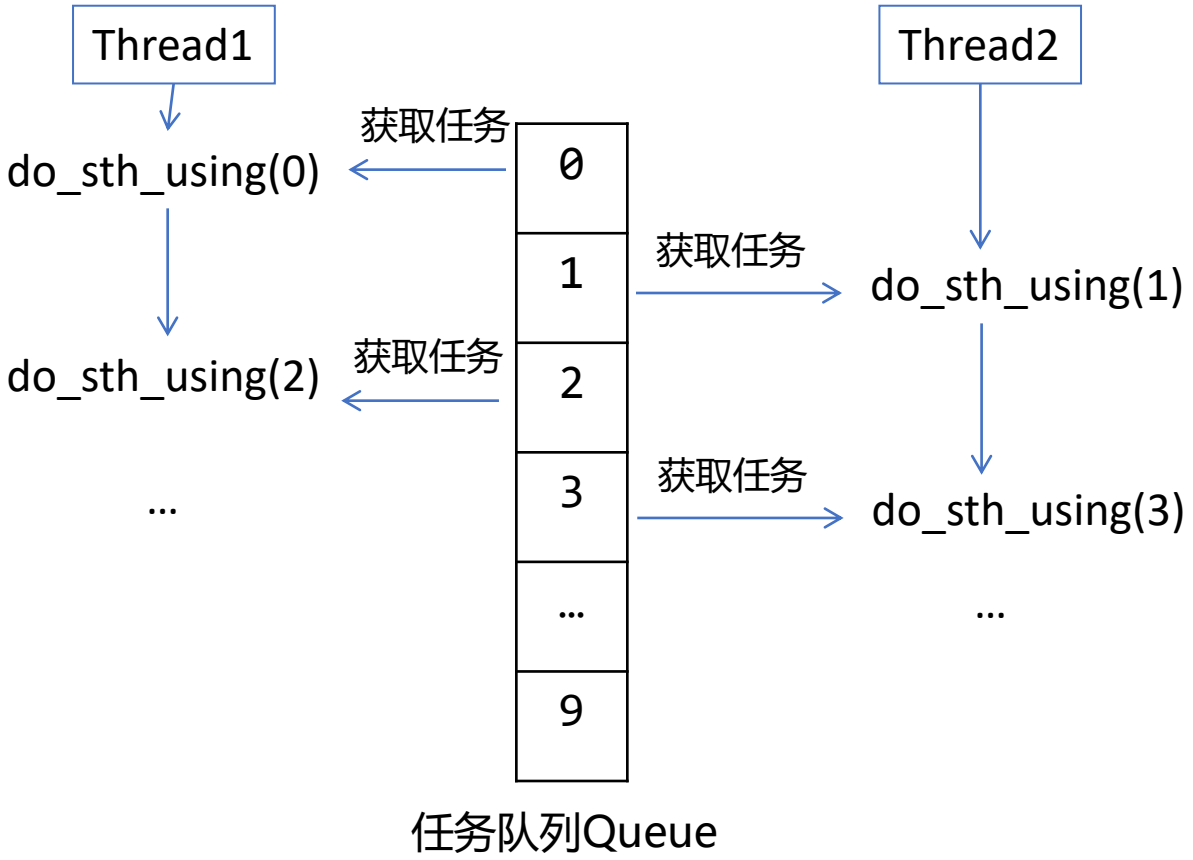
Queue模块中的常用方法：

- Queue.qsize() 返回队列的大小
- Queue.empty() 如果队列为空，返回True,反之False
- Queue.full() 如果队列满了，返回True,反之False
- Queue.full 与 maxsize 大小对应
- Queue.get([block[, timeout]])获取队列，timeout等待时间
- Queue.get_nowait() 相当Queue.get(False)
- Queue.put(item) 写入队列，timeout等待时间
- Queue.put_nowait(item) 相当Queue.put(item, False)
- Queue.task_done() 在完成一项工作之后，Queue.task_done()函数向任务已经完成的队列发送一个信号
- Queue.join() 实际上意味着等到队列为空，再执行别的操作

threading和queue相结合的并行化

- 简单的并行化编程

  将这一任务并行到2个线程上，并行化的原理图如下（处理任务的顺序可能不同）。



并发编程

- 简单的并行化编程（参见partD/parallel_example.py）

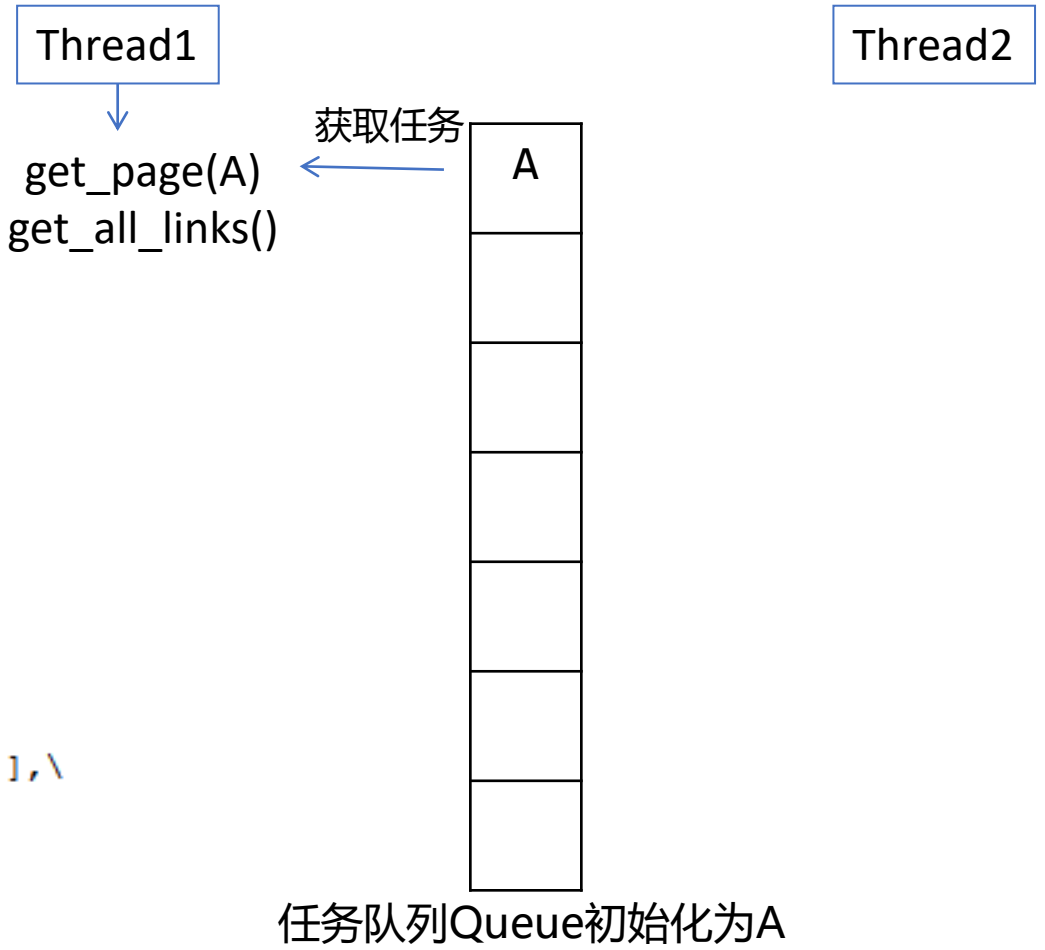
每个thread不停的从Queue中获取任务，处理任务

```
  def working():
      while True:
          arguments = q.get()                  #获取任务
          do_something_using(arguments)    #处理任务
          q.task_done()                            #任务结束

  for i in range(NUM):                  #生成NUM个线程等待队列
      t = Thread(target=working)  #每个线程的工作进程为working()函数
      t.setDaemon(True)
      t.start()

  for i in range(JOBS):
      q.put(i)                                  #将任务放入队列
  q.join()                                  #阻塞，等待所有任务完成
```

并发编程

- 将crawler_sample并行化(查看crawler_multi_thread.py)

  在并行化的爬虫中，队列初始只有seed，thread在爬取的过程中不停的往队列中添加待爬URL。

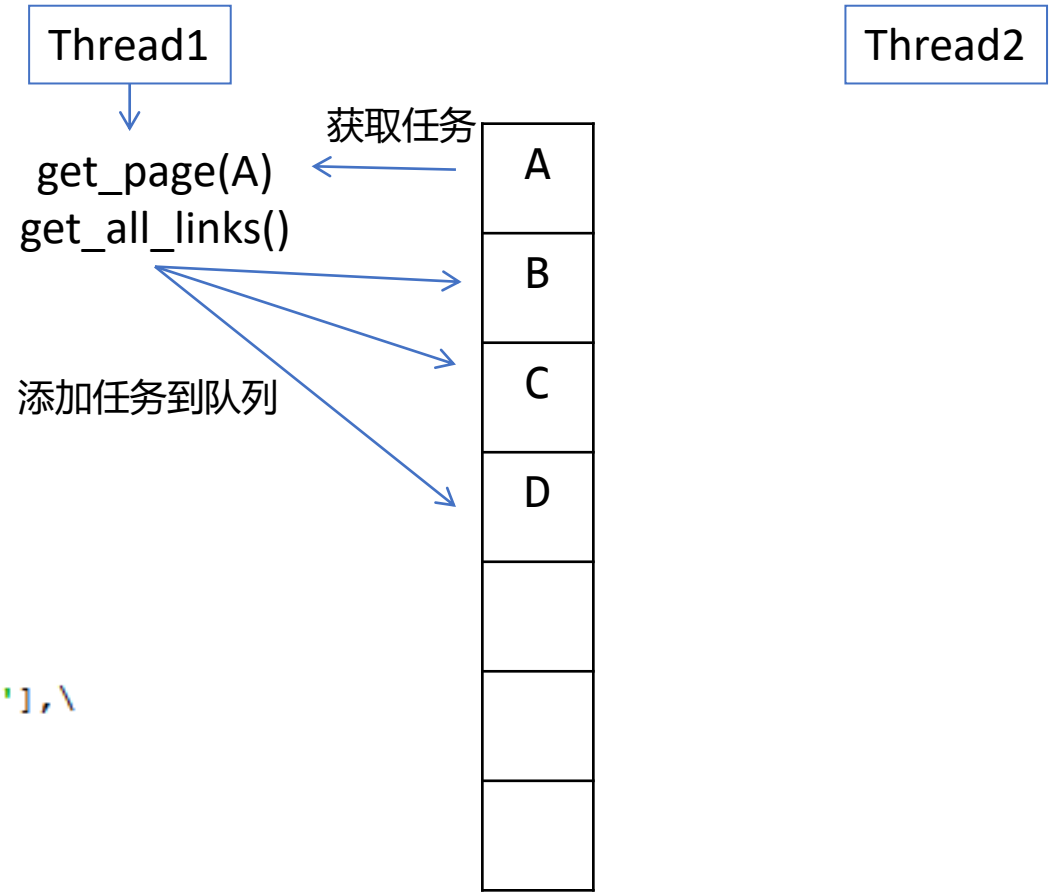



```
g = {'A':['B', 'C', 'D'],\
     'B':['E', 'F'],\
     'D':['G', 'H'],\
     'E':['I', 'J'],\
     'G':['K', 'L']}
```

并发编程

- 将crawler_sample并行化(查看crawler_multi_thread.py)

  在并行化的爬虫中，队列初始只有seed，thread在爬取的过程中不停的往队列中添加待爬URL。



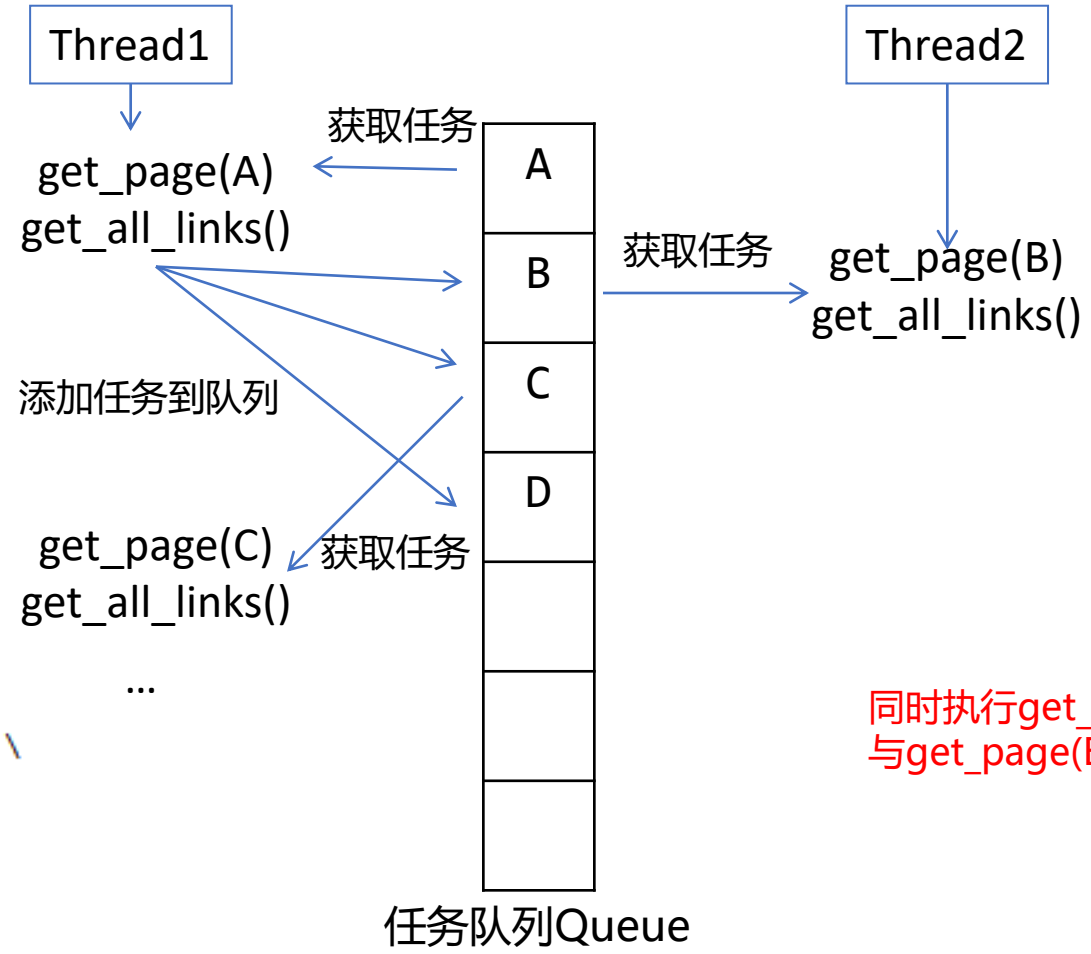

```
g = {'A':['B', 'C', 'D'],\
     'B':['E', 'F'],\
     'D':['G', 'H'],\
     'E':['I', 'J'],\
     'G':['K', 'L']}
```

任务队列Queue

并发编程

- 将crawler_sample并行化(查看crawler_multi_thread.py)

  在并行化的爬虫中，队列初始只有seed，thread在爬取的过程中不停的往队列中添加待爬URL。



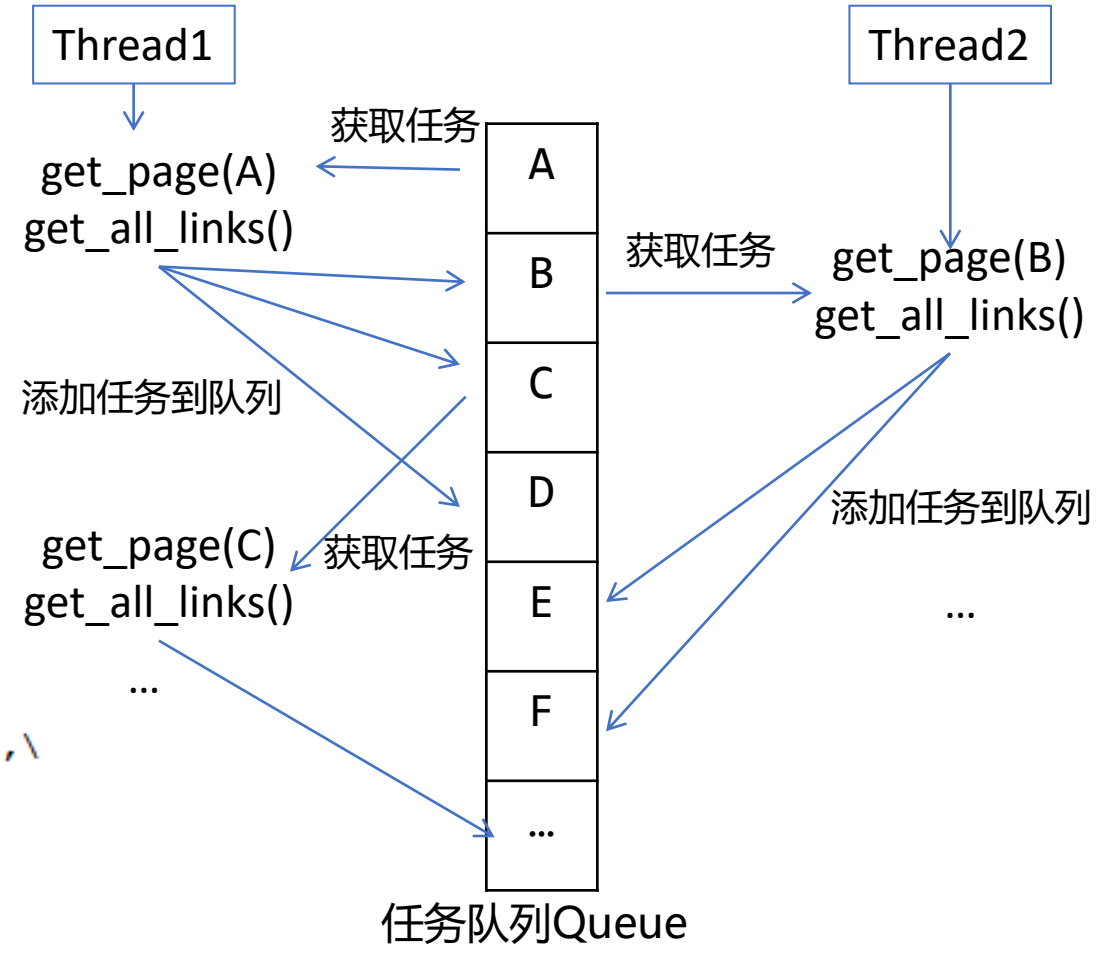

```
g = {'A':['B', 'C', 'D'],\
     'B':['E', 'F'],\
     'D':['G', 'H'],\
     'E':['I', 'J'],\
     'G':['K', 'L']}
```

同时执行get_page(C)
与get_page(B)

并发编程

- 将crawler_sample并行化(查看crawler_multi_thread.py)

  在并行化的爬虫中，队列初始只有seed，thread在爬取的过程中不停的往队列中添加待爬URL。




```
g = {'A':['B', 'C', 'D'],\
     'B':['E', 'F'],\
     'D':['G', 'H'],\
     'E':['I', 'J'],\
     'G':['K', 'L']}
```

并发编程

- 将crawler_sample并行化(查看crawler_multi_thread.py)

程序需要动态添加任务到queue。不同的线程需要操作相同的已爬网址列表crawled，为防止不同线程同时操作一个crawled列表产生冲突，可以为crawled变量加互斥锁。

```
crawled = []        #将crawled作为全局变量，这样每个线程都可以对其操作
varLock = threading.Lock()  #生成变量锁

def working():
    while True:
        page = q.get()          #从Queue中得到需要抓取的网址
        if page not in crawled:
            content = get_page(page)
            outlinks = get_all_links(content)
            for link in outlinks:
                q.put(link)     #将链接放入待爬队列Queue
            if varLock.acquire():       #一个线程操作全局变量前先将变量锁住，防止其
                #他线程同时操作，产生冲突
                graph[page] = outlinks
                crawled.append(page)
                varLock.release()       #操作结束解锁。
            q.task_done()
```

queue中的各种队列都实现了锁原语，能够在多线程中直接使用，可以使用队列来实现线程间的同步，因此不需要显示的加锁。

并发编程

- 将crawler_sample并行化(查看crawler_multi_thread.py)

→ 练习3

相比之前的串行程序，一个4线程的爬虫，只需要2s就可以完成爬取。

```
'''
downloading page A
downloading page Cdownloading page Bdownloading page D


downloading page Edownloading page F

downloading page H
downloading page G
downloading page I
 downloading page J
downloading page Kdownloading page L

2.11368308587
```

Q: 这种并行化的爬虫的搜索规则类似于DFS还是BFS？ （参见PartB部分的介绍）

A:类似于BFS，因为queue.Queue是先入先出的

需要提交的练习

练习

1.实现BloomFilter

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)。

提示：可以用函数实现（例如hashtable里，用函数操作table的做法），也可以用类实现（例如Bitarray.py的实现，可以修改Bitarray.py完成Bloomfilter）。

如果你的程序依赖于我们提供的一些代码（比如GeneralHashFucntion.py），请在提交作业时也附带上，方便我们直接运行程序测试。

练习

2．实现一个简单的网页爬虫（修改crawler.py）

其中：

```
def get_all_links(content, page):
    links = []
    ...
    return links
```

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。

例如，匹配形如

```
<a target="_blank" href="http://m.qiushibaike.com"></a>
<a href="/pic/page/2?s=4492933">2</a>
```

的网址。

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。

```
1  import urllib.parse
2
3  page = "http://www.qiushibaike.com/pic"
4  url = "pic/page/2?s=4492933"
5  print(urllib.parse.urljoin(page, url))
```

→ http://www.qiushibaike.com/pic/page/2?s=4492933

练习

需要修改的函数：

```
def get_page(page):
    content = ''
    ...
    return content
```

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。

```
def crawl(seed, max_page):
```

seed为种子网址，max_page为最多爬取的网页数。

练习

3．实现一个并行的爬虫

将练习2中的crawler.py改为并行化实现(partD/crawler_multi_thread.py)。

需要实现的功能：

queue.Queue初始时给入一个seed网址，从这个网站开始爬取一定数量的网页。

```
>>>q = queue.Queue()
>>>q.put('http://www.sjtu.edu.cn')
```

提示：

在爬取网页的时候建议将网址打出

需要加一个计数器，爬取一定数量的网页后停止抓取，或者是爬取一定深度后停止抓取。

函数输入输出没有特定要求，保存网页的模块可以用crawler.py中的add_page_to_folder。

如果觉得threading和queue的操作方式不便，也可以使用别的并行库。

BeautifulSoup对速度影响较大，可以用字符串方式改写get_all_links(content,page)函数，来加快速度。

拓展思考

本部分仅供同学们拓展学习，不作为作业。

1. 练习2给出的代码是DFS还是BFS？
2. 你能把bloom filter和多线程爬虫结合吗？
3. Python中还提供了multiprocessing等高级多线程/多进程工具（这两者有什么区别呢？），感兴趣的同学可以自行探索。
4. Python开启了N个线程，消耗时间可以变为原来的1/N吗？
5. 有一些爬虫框架可以方便地实现网页爬取，如Scrapy等，感兴趣的同学可以自行探索。

参考

- BloomFilter：https://blog.csdn.net/jiaomeng/article/details/1495500
  https://www.cnblogs.com/allensun/archive/2011/02/16/1956532.html
  https://www.zhihu.com/question/38211640
  http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html

- Python3中的多线程：https://www.runoob.com/python3/python3-multithreading.html
  https://docs.python.org/zh-cn/3.7/library/threading.html