



# 信息检索大作业实验报告

## 搜索引擎实现

作者：曹珉浩

组织：南开大学计算机学院

时间：December 17, 2023

学号：2113619



# 目录

<b>第 1 章 网页抓取</b>	<b>1</b>
1.1 对象选取 . . . . .	1
1.2 代码实现 . . . . .	2
1.3 数据清洗与预处理 . . . . .	3
<b>第 2 章 文本索引</b>	<b>5</b>
2.1 分词: NLP 工具 jieba . . . . .	5
2.2 倒排索引构建: 手动实现 . . . . .	6
2.3 TF-IDF . . . . .	8
<b>第 3 章 链接分析</b>	<b>9</b>
<b>第 4 章 查询服务</b>	<b>11</b>
4.1 普通查询 . . . . .	11
4.2 高级查询 . . . . .	13
4.2.1 时间查询: 仅保留时间范围内的搜索结果 . . . . .	13
4.2.2 站内查询: 仅保留来源为指定 URL 或域名的搜索结果 . . . . .	14
4.2.3 通配查询: 支持用户输入正则表达式进行模糊匹配 . . . . .	14
4.2.4 短语查询: 完全匹配、部分匹配、不匹配输入的若干词项 . . . . .	15
4.2.5 标题查询: 可以限制仅在标题中进行查询 . . . . .	15
4.3 查询日志 . . . . .	15
4.4 网页快照 . . . . .	16
4.5 个性化查询 . . . . .	17
<b>第 5 章 Web 页面及结果展示</b>	<b>19</b>
5.1 Web 页面总体展示与搭建过程简述 . . . . .	19
5.2 具体结果展示 . . . . .	20
<b>第 6 章 个性化推荐</b>	<b>24</b>

# 第1章 网页抓取

构建搜索引擎的第一步是进行网页抓取(爬虫)，网络爬虫可以自动化浏览网络中的信息，当然浏览信息的时候需要按照我们制定的规则进行，这些规则我们称之为网络爬虫算法。使用 Python 可以很方便地编写出爬虫程序，进行互联网信息的自动化检索。一般爬虫的流程图如下所示：

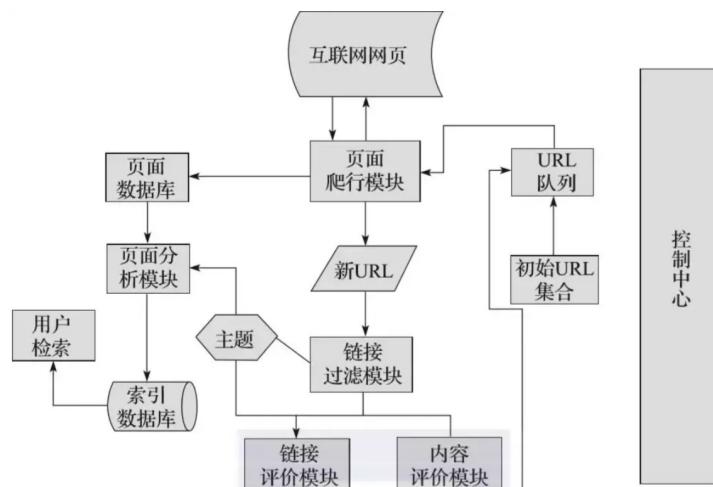


图 1.1：网络爬虫示意图

## 1.1 对象选取

在对象选取的方面，首先要考虑的是内容的丰富性和爬取的难易性，先尝试对豆瓣进行爬虫，但被反爬机制拒之门外，后尝试对各大新闻网站进行爬取，虽然成功，但考虑到新闻的实时性，爬取的信息大都是近几天发生的事情，涵盖面难免太少，且后期更新会产生更多的网页，这一想法也被废弃。最终，我们选取了南开要闻与媒体南开两个网址进行爬取，其中南开要闻网站上是若干校内新闻，而媒体南开则涵盖了若干新闻报社的国内与国际新闻。

确定了爬取对象后，接下来要观察网页的结构，打开开发者工具可以发现，网站的版心是一个大的表格元素，其中包含着若干表格项，其中表格项的元素都是链接标签 a。这种类似于排行榜的网页结构是很好进行爬取的，我们首先对这个 index 网页模拟请求，然后获取网页上所有链接的值与它的 href 属性，这样，我们就得到了下一步要进行爬取的所有网站的名字与 URL。

```
▶ <table width="98%" border="0" cellpadding="0" cellspacing="0">(...)</table>
▶ <table width="98%" border="0" cellpadding="0" cellspacing="0">(...)</table>
▼ <table width="98%" border="0" cellpadding="0" cellspacing="0">
  ▼ <tbody>
    ▼ <tr>
      ▼ <td width="80%" valign="middle">
        ▼ <div align="left">
          <a href="http://news.nankai.edu.cn/mtnk/system/2023/11/21/030058932.shtml"
             target="_blank">中国化工报：南开大学合成全金属富勒烯 为新材料创制提供新思路
          </a> == $0
        </div>
      </td>
      ▶ <td width="20%" align="right" valign="middle">(...)</td>
    </tr>
  </tbody>
</table>
▶ <table width="98%" border="0" cellpadding="0" cellspacing="0">(...)</table>
```

图 1.2：目标网站结构示意图 (以媒体南开为例)

将上一步得到的链接信息存储在一个字典中，接下来再次模拟请求，访问这些 URL，获取它们完整的 HTML 文档。在这个过程中，可能会产生重定向或者超时的问题，需要进一步解决。而且，我们需要确定好爬虫的频率，减轻目标服务器的压力，避免被封禁 IP。

## 1.2 代码实现

在实现方面，我们采用异步 + 协程的方式进行网页抓取，提速爬虫效率。异步爬虫是指使用异步编程技术进行网页抓取和数据提取的一种方式。传统的爬虫通常是同步的，即按顺序依次发出请求并等待每个请求的响应返回后再处理下一个请求。而异步爬虫则利用异步编程（在这里使用 **asyncio 库** 和 **aiofiles 库**）的特性，能够在发送一个请求后不必等待其返回，而是继续处理其他请求或任务。当响应返回时，再去处理这个请求的结果。而协程可以理解为一种轻量级的线程，由我们主动显式定义，用于和异步程序配合。

模拟请求完成后，使用 **parsel.Selector** 对象进行解析和提取 HTML 中的任意对象（通过 CSS 选择器语法来实现），在这里就是获取 a 的文本和跳转链接。

---

```

1 async def parse_catalogs(url):
2     async with coroutine:
3         async with httpx.AsyncClient() as client:
4             response = await client.get(url)
5             if response.status_code == 302:
6                 # 获取重定向的新 URL
7                 redirect_url = response.headers.get("Location")
8                 if redirect_url:
9                     # 使用新的 URL 发起请求
10                    response = await client.get(redirect_url)
11                    selector = Selector(response.text)
12                    print(response.text) # 打印爬取到的网页结构
13                    url_dict.update(zip(selector.css('a::attr(href)').getall(),
14                                         selector.css('a::text').getall())))

```

---

注意到有些网页的资源已经不在原来的服务器上了，进行访问会产生 302 重定向响应码，我们这里进行处理，找到这些资源目前真正的位置，然后使用新的 URL 再次发起请求。获取这些目录 URL 后，使用 selector 进行解析，将 a 的文本和跳转链接存储到字典当中，用于下一步再次发起模拟请求。

第二步的逻辑和第一步类似，只不过这次获取的是所有文本，并需要将这些文本保存到本地的 **\*.html** 文件中，由于爬取过程中可能得到某些异常网页，我们在此把这个过程中的网页名和网页链接存储到一个 DataFrame 中，用于后续的数据清洗，这个过程的关键代码如下：

---

```

1 selector = Selector(response.text)
2 title = selector.css('title::text').get()
3 async with aiofiles.open(f'./nk_news/{title}.html', mode='w', encoding='utf-8') as f:
4     await f.write(response.text)
5     url_df.loc[title] = url # 插入 df

```

---

在爬取过程结束后，HTML 文件夹和 DataFrame 的结构如图1.3所示：其中，两个网站爬取的是 **2021年初至2023年11月之间的所有校内新闻和媒体新闻**。

WPS 云盘					
北方网：牢记嘱托 奋勇奋进（三）：“小...	2023/11/20 21:39	HTML 文件	20 KB	73 中华读书报：“晋南关注”读后感——从《海水在于木》看孙亚光先生的诗与美	<a href="http://news.nankai.edu.cn/ntwk/system/2021/03/10/030044860.shtml">http://news.nankai.edu.cn/ntwk/system/2021/03/10/030044860.shtml</a>
北方网：南开大学“体质健康测试报告出炉...”	2023/11/20 21:39	HTML 文件	17 KB	74 光明日报：深挖把脉新发展理念 为信息化建设新引擎——媒体南开·南开大学	<a href="http://news.nankai.edu.cn/ntwk/system/2021/03/09/030044796.shtml">http://news.nankai.edu.cn/ntwk/system/2021/03/09/030044796.shtml</a>
北方网：南开大学“正式命名！纪...”	2023/11/20 21:46	HTML 文件	43 KB	75 津云：“大家说理”：攻克克隆“媒体南开·南开大学”	<a href="http://news.nankai.edu.cn/ntwk/system/2021/03/09/030044900.shtml">http://news.nankai.edu.cn/ntwk/system/2021/03/09/030044900.shtml</a>
北方网：南开大学获批高校思政课教研研...”	2023/11/20 21:48	HTML 文件	18 KB	76 天津日报：南大成立“爱心助学基金”——媒体南开·南开大学	<a href="http://news.nankai.edu.cn/ntwk/system/2021/03/10/030044777.shtml">http://news.nankai.edu.cn/ntwk/system/2021/03/10/030044777.shtml</a>
北方网：南开大学助力“觉醒年代”邵阳市...	2023/11/20 21:50	HTML 文件	24 KB	77 天津日报：南大师生齐发声：永跟党走——媒体南开·南开大学	<a href="http://news.nankai.edu.cn/ntwk/system/2021/03/10/030044810.shtml">http://news.nankai.edu.cn/ntwk/system/2021/03/10/030044810.shtml</a>
北方网：南开大学成立“心理健康教育...”	2023/11/20 21:59	HTML 文件	18 KB	78 天津日报：南大“五项管理”——媒体南开·南开大学	<a href="http://news.nankai.edu.cn/ntwk/system/2021/03/10/030044781.shtml">http://news.nankai.edu.cn/ntwk/system/2021/03/10/030044781.shtml</a>
北方网：南开大学首批评选全国急救教育...”	2023/11/20 21:59	HTML 文件	17 KB	79 天津日报：南大“五项管理”——媒体南开·南开大学	<a href="http://news.nankai.edu.cn/ntwk/system/2021/03/10/030044782.shtml">http://news.nankai.edu.cn/ntwk/system/2021/03/10/030044782.shtml</a>
北方网：南开大学拟转化科技成果...”	2023/11/20 21:51	HTML 文件	17 KB	80 天津日报：“南大人在军车”：弘扬红色基因——媒体南开·南开大学	<a href="http://news.nankai.edu.cn/ntwk/system/2021/03/10/030044792.shtml">http://news.nankai.edu.cn/ntwk/system/2021/03/10/030044792.shtml</a>
北方网：南开大学副校长...”	2023/11/20 21:58	HTML 文件	18 KB	81 中国新闻网：“南大人在军车”：弘扬红色基因——媒体南开·南开大学	<a href="http://news.nankai.edu.cn/ntwk/system/2021/03/09/030044776.shtml">http://news.nankai.edu.cn/ntwk/system/2021/03/09/030044776.shtml</a>
北方网：南开大学男篮连获20年校赛冠军...”	2023/11/20 21:47	HTML 文件	21 KB	82 中国新闻网：“关爱哺乳期女教工”——媒体南开·南开大学	<a href="http://news.nankai.edu.cn/ntwk/system/2021/03/09/030044771.shtml">http://news.nankai.edu.cn/ntwk/system/2021/03/09/030044771.shtml</a>
北方网：南开大学科研团队研制新型人工血管...”	2023/11/20 21:45	HTML 文件	17 KB	83 天津电视台：守护人民健康 践行“德才兼备、厚德载物”——媒体南开·南开大学	<a href="http://news.nankai.edu.cn/ntwk/system/2021/03/09/030044798.shtml">http://news.nankai.edu.cn/ntwk/system/2021/03/09/030044798.shtml</a>
北方网：南开大学与开发区专业发展组织...”	2023/11/20 21:35	HTML 文件	17 KB	84 天津日报：南大“五项管理”——媒体南开·南开大学	<a href="http://news.nankai.edu.cn/ntwk/system/2021/03/10/030044799.shtml">http://news.nankai.edu.cn/ntwk/system/2021/03/10/030044799.shtml</a>
北方网：南开大学与市卫生健康委签约“教...”	2023/11/20 21:36	HTML 文件	17 KB	85 天津日报：南大“五项管理”——媒体南开·南开大学	<a href="http://news.nankai.edu.cn/ntwk/system/2021/02/01/030044504.shtml">http://news.nankai.edu.cn/ntwk/system/2021/02/01/030044504.shtml</a>
北方网：南开大学附属天津眼科医院联手...”	2023/11/20 21:31	HTML 文件	19 KB	86 天津日报：南大师生共绘“爱眼同心圆”——媒体南开·南开大学	<a href="http://news.nankai.edu.cn/ntwk/system/2021/01/31/030044497.shtml">http://news.nankai.edu.cn/ntwk/system/2021/01/31/030044497.shtml</a>
北方网：南开大学与天津联通携手...”	2023/11/20 21:28	HTML 文件	18 KB	87 中国新闻网：“南大师生携手共绘‘爱眼同心圆’——媒体南开·南开大学”	<a href="http://news.nankai.edu.cn/ntwk/system/2021/01/28/030044486.shtml">http://news.nankai.edu.cn/ntwk/system/2021/01/28/030044486.shtml</a>
北方网：南开大学与武清区签署战略合作...”	2023/11/20 21:48	HTML 文件	21 KB	88 天津日报：南大师生携手共绘“爱眼同心圆”——媒体南开·南开大学	<a href="http://news.nankai.edu.cn/ntwk/system/2021/01/29/030044489.shtml">http://news.nankai.edu.cn/ntwk/system/2021/01/29/030044489.shtml</a>
北方网：南开大学再添3座市级科普基地...”	2023/11/20 21:48	HTML 文件	18 KB	89 新华网客户端：“南大师生携手共绘‘爱眼同心圆’——媒体南开·南开大学”	<a href="http://news.nankai.edu.cn/ntwk/system/2021/02/01/030044509.shtml">http://news.nankai.edu.cn/ntwk/system/2021/02/01/030044509.shtml</a>
1,456 个项目					

图 1.3: 爬虫结果示意图 (以媒体南开为例)

## 1.3 数据清洗与预处理

### 一、数据清洗

对于**南开要闻**这个网站所爬取的 2925 个网页，中途没有产生异常，文件夹下也都是完好的 HTML 文档，无需另行处理，而对于**媒体南开**这个网站所爬取的网页，过程中存在一些异常，导致 csv 文件的长度和抓取网页数不一致，并且爬取到了一些非 HTML 文档，如下所示：

ChinaDaily	2023/11/23 14:44	文件	0 KB
CHINADAILY: Zhou Enlai exhibition ...	2023/11/23 14:46	HTML 文件	17 KB
Xinhua Headlines	2023/11/23 14:29	文件	0 KB
Xinhua News	2023/11/23 14:42	文件	0 KB
Xinhua News: 20th CPC National Co...	2023/11/23 14:39	文件	0 KB

图 1.4: 媒体南开网站爬取过程中的异常

在这里，我们考虑仅保留一些以媒体单位开头的网页，如北方网，光明日报，津云等，这些网页可以保证没有发生异常，并且也利于我们后续实现个性化推荐。

```

1 medias = ['北方网', '光明日报', '北京日报', '津云', '经济日报', '科技日报', '科学网',
2   '人民日报', '天津日报', '新华网', '中国教育报', '中国科学报', '中国青年报', '中国新闻网']
3 # 删除不以上面列表中任意元素开头的文件
4 for file in files:
5   if not any(file.startswith(media) for media in medias):
6     file_path = os.path.join(path,file)
7     os.remove(file_path)

```

接着处理对应的 csv 文件，逻辑类似，在此不多赘述。但在处理后我们又发现一个问题：**处理后 csv 的长度和文件夹下的文件个数不一致**，这是由于在爬取过程中，写入了诸如”<https://xs.nankai.edu.cn/>”这样的文件导致，它们并非 HTML 文件，但却可以被我们的计算机正确识别成一个网页，我们还应该给这样的文件进行删除，为了保证绝对的正确性，我们采用类似集合的思想，进行两次遍历删除，代码同样不在此赘述。

经过上面的处理，两个网页集合和对应的 csv 已经清洗干净，后续将为它们进行构建索引等操作，数据清洗的全部代码，存储在 `spider/adjust.ipynb` 文件中。

### 二、数据预处理

接下来，我们要对数据进行一些预处理，主要是提取 HTML 代码中的关键词，编辑作者以及发布时间等，便于后续进行分词然后构建索引。在这里以添加文档描述为例，观察我们爬取的网页结构可以发现，文档的摘要都存储在一个以”description”为类名的 `<meta>` 标签中，如图 1.5 所示：

图 1.5: 文档摘要位置示意图

所以，我们仍然使用 `parsel.Selector` 对象，对本地的这些文件进行抓取即可，最后更新我们的 csv，便于在下一阶段构建索引，关键代码如下所示：

```
1 def add_description(path="htmls"):
2     files = os.listdir(path)
3     for file_name in files:
4         file_path = os.path.join(path,file_name)
5         with open(file_path,'r',encoding='utf-8') as file:
6             content = file.read()
7             selector = Selector(content)
8             title = selector.css('title::text').get().replace('/', '_')
9             # 获取 head 内的以 description 为类名的 meta 标签内容
10            description = selector.css('meta[name="description"]::attr(content)').get()
11            if description is not None: # 去除空字符
12                description = description.replace('\r', '').replace('\n', '')
13                .replace('\t', '').replace('\n', '').replace(' ', '')
14            data.loc[title, 'description'] = description
```

图 1.6: 增加文档描述后, csv 文件结果展示

遵循类似的方法，我们还可以从 HTML 源文件中提取发布者以及发布时间等信息，在此不多赘述，这部分的源代码，也位于 `spider/adjust.ipynb` 文件中。

第2章 文本索引

在网页抓取模块实现完毕后，我们已经有了构建搜索引擎的必要资源，即所有的 HTML 文档，以及从中提取出来的关键信息，比如文章标题、URL、文档内容、发布时间、作者等。在本章中，我们将基于这些信息构建索引，具体做法是：首先利用 NLP 工具对文章标题、内容等信息进行分词；然后基于分词结果构建倒排索引；在倒排索引的构建过程中，还可以计算 TF 与 IDF 值，便于后续查询系统的实现。

## 2.1 分词：NLP 工具 jieba

`jieba` 是一个流行的中文文本处理库，用于中文分词、词性标注、关键词提取等自然语言处理任务。我们在本小节中，主要利用 `jieba` 库中的 `cut_for_search`，它是用于搜索引擎模式的分词方法，专门用于处理较长文本，并尝试提高搜索引擎中文本匹配的效果。

基于 `cut_for_search`, 我们只需要遍历数据 csv, 然后对每一行提取标题、描述和文本内容三个字段进行分词，并转换为列表；之后，由于 jieba 分词会保留中文标点符号和一些特殊的符号，我们要禁用这些标点和特殊符号，因为它们没有任何的实际意义；最后还要把这个列表变成字符串进行存储，关键代码如下所示：

```
1 cutted = [] # 用于存储分词后的结果
2 # 一次处理一行，对标题、描述、正文进行分词
3 for i in range(len(data)):
4     info = data.iloc[i]
5     title = list(cut_for_search(info.title))
6     description, content = str(info.description), str(info.content)
7     if description is not None:
8         description = list(cut_for_search(description))
9     if content is not None:
10        content = list(cut_for_search(content))
11    # 接下来把分词后的列表以字符串方式存储，并忽略无意义的标点符号
12    title = (re.sub(rf"\[punctuations\]", ' ', '#'.join(title)).replace('-', '')).split('#')
13    # 然后把新的分词结果按空格分割组合成新的字符串
14    title = ' '.join([word for word in title if (word != '' and word != ' ')])
15    # 接着对 description 和 content 做类似的操作，在此不多赘述
```

将分词后的结果保存到 `cutted.csv` 中，如图2.1所示，用于下一步构建倒排索引。

**图 2.1: jieba.cut for search 分词后结果示意图**

## 2.2 倒排索引构建：手动实现

在没有搜索引擎时，我们是直接输入一个网址，然后获取网站内容，这时我们的行为是：document → words，这种通过文章获取其中单词的形式就是正向索引。但我们的需求常常是，输入一个单词，搜索引擎能够找到含有这个单词，或者和这个单词有关系的文章，这时我们的行为是 word → documents，于是我们把这种倒过来的索引称为**倒排索引**，其一般结构如下图所示：



图 2.2: 倒排索引结构示意图

由于手动构建倒排索引并不复杂，在这里没有使用 Elasticsearch 工具而是自己手动实现。我们倒排索引的构建思路是遍历 **cutted.csv**，计算每个 HTML 中的词频（忽略停用词），然后据此构造倒排索引，最后把倒排索引字典转换为 json 格式进行存储，方便前端使用。

首先计算每个 HTML 文档中的词频：得益于上一步的处理，**cutted.csv** 中的每一行都对应每一个 HTML 文档的全部信息，并且描述、标题和正文都进行了分词，在此我们只需要遍历每一行，为出现的每个词项计数即可，但上一步分词后的结果可能会产生很多无用的此项，比如它他她，新闻，啊呀哦这样的词项，我们把这些词项加入停用词列表，不对他们进行计数。这一步的构建的字典架构图如图2.3(左) 所示，关键代码如下所示：

```

1 def calculateTFInHTML(data=data,title_only=False):
2     # 要返回的“正向”索引词典，映射关系为：url->{}, 而作为 value 的词典是一个词项到词频的映射
3     index = {}
4     # 遍历 data 的每一行，遍历标题，内容等信息，计算词频
5     for url,info in data.iterrows():
6         index[url] = {}
7         for word in info.title.split(" "):
8             # 不在停用词列表中的才计数
9             if word not in stopWords:
10                 if word not in index[url]:
11                     index[url][word] = 1
12                 else:
13                     index[url][word] += 1
14         # ... 其余类似，不在此赘述
15     return index

```

此外，注意到我们的函数中还有一个参数 **title\_only**，它用于表示是否只对标题中出现的词项进行计数，这是为了方便我们后续高级搜索功能模块的实现。

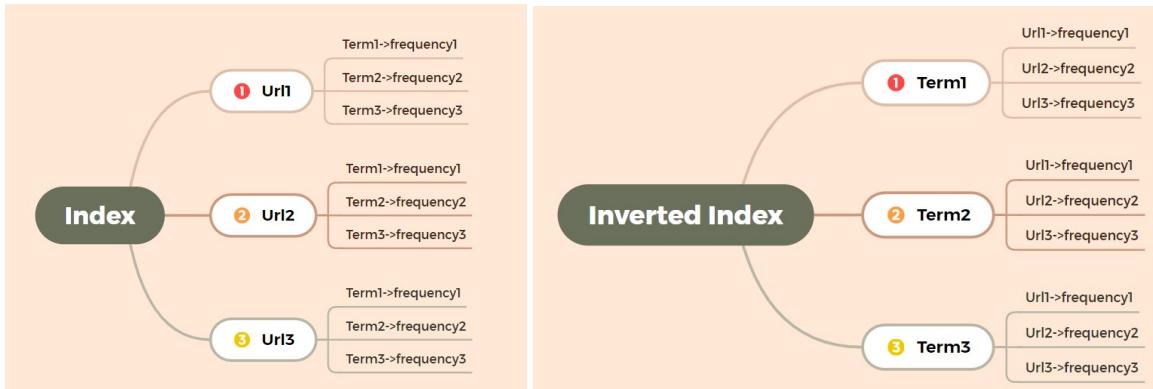


图 2.3: 文档词频计算函数构造出的字典架构示意图 (左) 及倒排索引架构示意图 (右)

接下来构造倒排索引，思路就是遍历上述得到的索引中的每个 URL 和对应的词汇及频率信息，如果当前词汇不在倒排索引中，就创建一个空字典作为该词的索引项，如果在就直接把这个映射赋值，倒排索引字典的架构图如图2.3的右图所示，代码如下所示：

```

1 def gen_inverted_index(index):
2     inverted_index = {}
3     # 遍历原始索引中的每个 URL 和对应的词汇及频率信息
4     for url, words in index.items():
5         for word, frequency in words.items():
6             # 如果当前词汇不在倒排索引中，就创建一个空字典作为该词的索引项
7             if word not in inverted_index:
8                 inverted_index[word] = {}
9                 inverted_index[word][url] = frequency
10            else:
11                inverted_index[word][url] = frequency
12    return inverted_index

```

经过这两步操作，我们就成功构建了共含 5644 个 HTML 文件的倒排索引，现在已经可以执行一些简单的查询工作了，比如我们查找“计算机”这个词项，返回的结果如图2.4所示，一共有 288 个结果，它们对应的 HTML 文档中，均含有计算机这个词项。

```

with open('jsons/invert_index.json', 'r', encoding='utf-8') as file:
    data_dict = json.load(file)
data_dict['计算机']
在 1s 的 1 Dec at 16:42:39 执行

{'http://news.nankai.edu.cn/ywsd/system/2021/01/02/030043730.shtml': 2,
 'http://news.nankai.edu.cn/ywsd/system/2021/09/27/030048112.shtml': 1,
 'http://news.nankai.edu.cn/ywsd/system/2022/03/08/030050510.shtml': 5,
 'http://news.nankai.edu.cn/ywsd/system/2023/07/11/030057037.shtml': 2,
 'http://news.nankai.edu.cn/ywsd/system/2022/03/30/030050743.shtml': 1,
 'http://news.nankai.edu.cn/ywsd/system/2021/09/11/030047882.shtml': 1,

```

图 2.4: 利用倒排索引搜索“计算机”得到的结果 (共 288 个文档含有)

## 2.3 TF-IDF

在上一小节的结尾中我们看到，搜索一个 term 会产生很多很多的结果。那么，这么多的结果应该如何返回给用户呢？一个很清晰的想法就是找到和用户描述最相似的文档返回给用户，这就是本节中 TF-IDF 所做的工作。当然，仅依靠 TF-IDF 还是不会产生令人满意的结果，后续我们还有链接分析等工作进一步优化返回结果。

### 定义 2.1

**tf**: 即词频 term frequency，也就是一个词在文档中出现的个数

**idf**: 逆文档频率 inverse document-frequency，其计算公式为：

$$idf(t) = \log \frac{n}{df(t)}$$

**TF-IDF 值**: 即 TF 值和 IDF 值的乘积:  $TF-IDF(t) = tf(t) \times idf(t)$



TF-IDF 被用来衡量一个词项在一个文档中的重要性。它假设如果一个词在一个文档中出现的频率较高，并且在整个文档集合中出现的频率相对较低，那么它对这个文档的重要性就更高。

TF-IDF 的实现很简单，代码如下所示，然后依然将结果保存到 json 文件中。

---

```

1 def get_IDF(index):
2     idf = {}
3     for url, frequency_dict in index.items():
4         for word, frequency in frequency_dict.items():
5             idf[word] = np.log(len(index) / frequency)
6     return idf
7 def get_TF_IDF(index, IDF):
8     tf_idf = {}
9     for url, words in index.items():
10        dict = {}
11        for word, frequency in words.items():
12            dict[word] = frequency * IDF[word]
13        tf_idf[url] = dict
14    return tf_idf

```

---

以随便一篇 HTML 文档为例，当作 key 传入字典就可以得到该篇文章内所有词项的 TF-IDF 值：

```

with open('jsons/tf-idf.json', 'r', encoding='utf-8') as file:
    tfidf_dict = json.load(file)
tfidf_dict['http://news.nankai.edu.cn/ywsd/system/2021/10/17/030048360.shtml']
在 1s 的 1 Dec at 19:11:27 执行
    '有机化学': 8.638348312972704,
    '重点': 8.638348312972704,
    '实验': 8.638348312972704,
    '实验室': 8.638348312972704,
    '强调': 8.638348312972704,

```

图 2.5: 获取文档内所有词项的 TF-IDF 值示例

## 第3章 链接分析

PageRank 是一种用于衡量网页重要性和影响力的方法，它通过分析网页间的链接关系，将一个网页的重要性定义为其他页面链接到它的数量和质量。如果一个网页被其他重要的网页链接，那么它的 PageRank 值会更高，被视为更重要的页面。在搜索引擎中，PageRank 被用来对搜索结果进行排序。搜索引擎根据页面的 PageRank 值将搜索结果呈现给用户，更重要的页面通常会在搜索结果中排名更靠前。

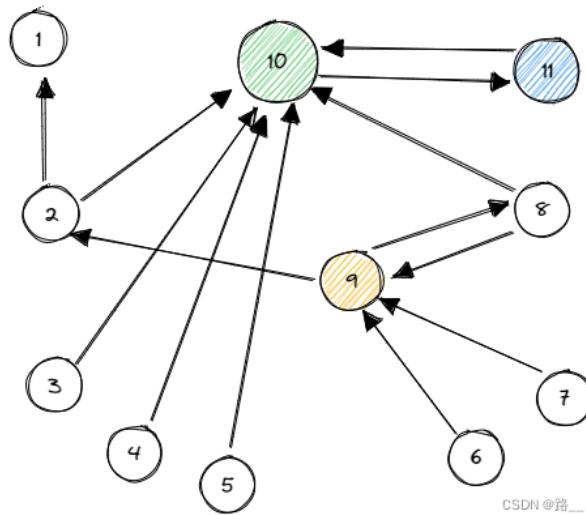


图 3.1: PageRank 示意图

如上图所示，PageRank 实现中最重要的就是一个有向图的构建：将互联网视为一个有向图，网页作为图中的节点，超链接作为图中的有向边。如果网页 A 有一个指向网页 B 的链接，则有一条从节点 A 指向节点 B 的有向边，接着初始化权重，并反复迭代计算每个网页节点的 PageRank 值直到收敛，然后更新权重，并引入一个阻尼因子 (damping factor)，通常设定为 0.85，表示用户有 15% 的概率随机访问其他页面而不是通过链接跳转，最后当节点的 PageRank 值不再发生显著变化或达到预设的迭代次数时，停止迭代。

在这里，我们使用 **networkx** 库进行实现，这个库可以帮助我们建立有向图、添加有向边，并且传入一个阻尼因子还能帮我们计算 PageRank 值直到算法收敛。但在此之前，我们还需要建立一个 url->url\_list 的映射，即一个 HTML 文档中可以链接到哪些 URL，依然使用 `parsel.Selector` 抓取文档的链接标签，代码如下所示：

```
1 for file in files:
2     title = str(file).split(".html")[0]
3     url = df.loc[title].url
4     with open(os.path.join(path,file), 'r', encoding='utf-8-sig') as f :
5         text = f.read()
6         selector = Selector(text)
7         url_list = []
8         url_list.extend(selector.css('a::attr(href)').getall())
9         url_dict[url] = url_list
```

有了这个词典，并借助 `networkx` 库，PageRank 的实现就很简单了，代码如下所示：

```
1 def get_page_rank():
2     # 创建一个有向图
```

```

3  digraph = networkx.DiGraph()
4  # 在有向图中添加有向边
5  for url, url_list in url_dict.items():
6      for url2 in url_list:
7          if url2 in df.url.values:
8              digraph.add_edge(url, url2)
9  # 引入阻尼因子，利用库函数计算 pagerank
10 pagerank = networkx.pagerank(digraph, alpha=0.85)
11 page_rank_df = pd.Series(pagerank, name='page_rank')
12 # 对 PageRank 进行处理，缩放并限制数值范围，避免权重过大
13 page_rank_df = page_rank_df.apply(lambda x: math.log(x * 10000, 10) + 1)
14 page_rank_df.to_csv("./page_rank.csv", encoding='utf-8-sig')

```

这样，我们就计算了 5644 个文档的 PageRank 值，排名靠前的一些网页如下所示：

	url	page_rank
1	http://news.nankai.edu.cn/index.shtml	3.643053
2	http://news.nankai.edu.cn/ywsd/index.shtml	3.643053
3	http://news.nankai.edu.cn/mtnk/index.shtml	3.643053
4	http://news.nankai.edu.cn/gynk/index.shtml	3.643053
5	http://news.nankai.edu.cn/nkrw/index.shtml	3.643053
6	http://news.nankai.edu.cn/nkdxb/index.shtml	3.643053
7	http://news.nankai.edu.cn/sp/index.shtml	3.643053
8	http://news.nankai.edu.cn/gb/index.shtml	3.643053
13	http://news.nankai.edu.cn/ywsd/system/2023/11/17/030058850.shtml	3.623492

图 3.2: PageRank 计算结果示意图：查看排名靠前的几个网页

可以发现，排名靠前的网页大多都是列表页（索引页），它们的内容是很多的链接，而且这些链接跳转的网页，一般也都有跳转回索引页的功能，因此它们的入边和出边都很多，网页的质量较高，PageRank 靠前。

在后续的查询服务实现中，我们会综合利用本章实现的 PageRank，上一章实现的 TF-IDF，以及相似度定义函数（如余弦相似度）等，尽力优化我们搜索引擎的返回结果。

关于 PageRank 部分的完整代码，请参考 [pageRank.ipynb](#)。

# 第4章 查询服务

我们的查询服务将基于前面的三个模块开展，对于用户输入的关键字，进行分词索引，根据余弦相似度以及PageRank综合计算出一个得分，返回相应的结果，我们把这个工作称作普通查询。而对于站内查询、短语查询、通配查询等高级搜索功能，将基于普通查询的结果进行过滤，最终返回给用户过滤后的结果。

## 4.1 普通查询

普通查询是我们检索系统最主要的一个函数，它基于向量空间模型实现。向量空间模型是一种用于表示文本文档的数学模型，主要用于计算文档间的相似度。在这个模型中，每个文档被表示为一个向量，向量的每个维度对应于词汇表中的一个词。而相似度的度量方法有很多，在这里我们采用余弦相似度进行度量，余弦相似度测量了两个向量在空间中的夹角，其值的范围从-1（完全不相似）到1（完全相同），通常情况下，相似的文档会有较高的余弦相似度值，这个值将对返回结果的顺序产生重要影响。

定义好一些辅助函数(比如计算TF、TF-IDF值的函数以及计算向量长度的函数)后，开始设计普通查询函数，它目前接受三个参数：用户查询的字符串，以及一个标志位，表示是否仅在标题中进行检索，还有一个num，表示返回的结果数量，默认为100。而函数的返回值是一个num长度的列表(如果查询成功的话)，列表中的元素是一个元组，其内容为网址和相似度，先来看关键代码：

---

```
1 # 对输入和历史记录进行分词
2 spilt_input = sorted(list(cut_for_search(input)))
3 spilt_input = [term for term in spilt_input if term not in ["", " "]]
4 # 判断用户需要的搜索模式，设置数据集
5 if onlyTitle == True:
6     tf_dict,idf_dict,words = tf_title,idf_title,word_set_title
7 else:
8     tf_dict,idf_dict,words = tf,idf,word_set
9 # 对于每个文档，计算其所有词项的 TF-IDF 值，并选取 TF-IDF 值最高的前 num 个词项
10 key_tfidf_dict = {}
11 for key,value in tfidf_dict.items():
12     key_tfidf_dict[key] = sorted(tfidf_dict[key].items(),key=lambda item:item[1],reverse=True)[0:num]
13 # 对用户输入和历史记录计算 TF-IDF 值
14 tf_input = getTF(words,spilt_input)
15 tfidf_input = getTF_IDF(tf_input,idf_dict)
16 key_input = sorted(tfidf_input.items(),key=lambda item:item[1],reverse=True)[0:num]
17 len_key_input = getVecLength(key_input)
18 # 计算余弦相似度
19 key_results = [] # 用于存储余弦相似度
20 key_results_index = [] # 记录文档索引
21 for i in range(len(key_tfidf_dict_keys)):
22     length = 0
23     temp_list = key_tfidf_dict_values[i]
24     # 遍历每个输入关键词
25     for key in key_input:
26         if key[1] !=0: # tf-idf 值不为 0 才存在相似度
```

```

27     # 遍历文档内的每个关键词
28     for value in temp_list:
29         if key[0] == value[0]:
30             length = length + key[1]*value[1]
31     # 余弦相似度
32     sim = round(length/(len_key_input*getVecLength(temp_list)),4)
33     key_results.append((key_tfidf_dict_keys[i],sim))
34     if sim > 0:
35         key_results_index.append(i)
36 # 返回结果, 其形式为 (url,similarity)
37 ls = []
38 for res in results:
39     if res[1]>0 :
40         ls.append((res[0],res[1]))

```

---

主要的逻辑就是上面代码注释的这六步,由于这个函数代码太长,在此不完全展示,完整代码可以参考search.py,在后续为了实现个性化查询,我们还将对这个函数进行优化,主要是用于记录用户的查询历史,通过查询历史来对结果排名作用,起到个性化的目的。此外,在search.py中,我们还为每个查询功能的函数实现了一些测试函数,用以检查我们的实现是否达到预期。

这部分工作完成后,我们其实已经实现了最简单的检索系统,用户输入查询字符串,我们会按相似度从高到低返回结果,但是这种体验不是很好,因为我们只给用户返回了一大堆的网页链接,用户还要自己一个一个点开看是否是自己需求的或者感兴趣的。因此我们还要对结果进行拓展,即利用URL查我们前面处理好的数据,然后拓展返回结果(比如添加标题和描述等等),目的效果类似于百度搜索后,如图4.1所示的结果:

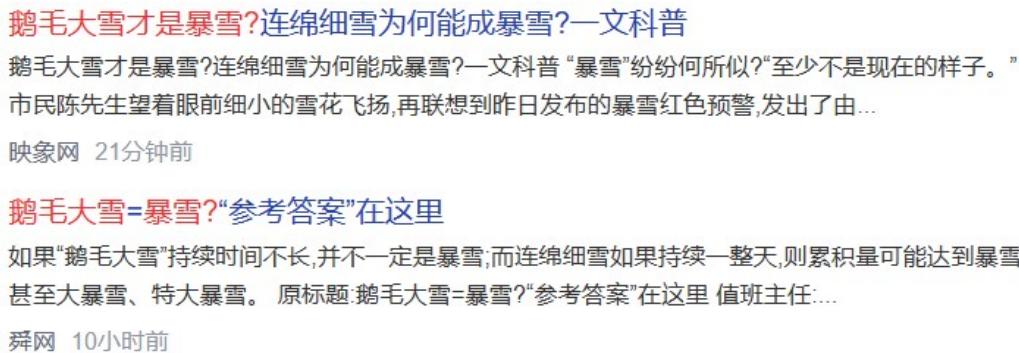


图 4.1: 结果拓展, 目的效果示意图, 添加了网页标题以及描述等

实现代码比较简单,就是查表插入的过程,我们还增加了一个新的列,代表网页的最终评分,他综合了余弦相似度与PageRank,是网页更加可信的排名,具体代码如下所示:

```

1 def expand_results(results:list):
2     expanded = []
3     for res in results:
4         url = res[0]
5         row = allInfo.loc[url].fillna('')
6         title = str(row['title']).replace("_","/")
7         dsp = str(row['description'])
8         # 计算网页综合得分: 0.7* 余弦相似度 + 0.3*pagerank 值

```

```

9     score = res[1]*0.7 + 0.3*page_rank.loc[url]['page_rank']
10    expanded.append((title,url,dsp,score))
11    # 按照综合得分降序排列并返回
12    return sorted(expanded,key=lambda item:item[-1],reverse=True)

```

## 4.2 高级查询

在本节中，我们将仿照百度的高级搜索，实现我们搜索引擎的功能。



图 4.2: 百度的高级搜索功能

### 4.2.1 时间查询：仅保留时间范围内的搜索结果

我们也按照百度高级搜索的使用方法，到时在搭建 Web 页面时会创建一个表单，时间检索这里会放一个下拉列表，为用户提供若干时间限制选项。在本小节和后面其他的高级搜索功能中，逻辑其实就是不断的对普通查询结果进行过滤的过程，因此我们设计的函数都是：传入结果的一行，判断是否满足限制条件，之后高级搜索过滤时，仅保留函数值为 True 的结果行即可。下面是时间限制的函数实现：

```

1 def check_time(result,limit):
2     row = allInfo.loc[result[1]] # result[1] 是 URL, 即 allInfo 的键
3     if str(row['date_timestamp']) != "nan":
4         # 将时间戳转换为 datetime
5         articleTime = datetime.fromtimestamp(int(row['date_timestamp']))
6         res = datetime.now() - articleTime
7         if limit == "一周内":
8             if res > timedelta(days=7):
9                 return False
10            elif limit == "一个月内":
11                if res > timedelta(days=30):
12                    return False
13            elif limit == "一年内":
14                if res > timedelta(days=365):
15                    return False
16            if str(row['date_timestamp']) == "nan":

```

---

```

17     return False
18     return True

```

---

我们将在 Web 页面搭建完成后，统一展示这些高级搜索的结果，此外，我们在 search.py 中编写了很多测试函数，拿到数据后可以直接运行测试。

### 4.2.2 站内查询：仅保留来源为指定 URL 或域名的搜索结果

如上所述，我们也只保留 URL 中含这个字段的结果即可，函数实现如下：

---

```

1 # 预处理，从 <input ...> 标签中提取它的 value 字段，代表 URL 或域名
2 website = form.site_or_domain
3 domain = str(website).split("value")[-1][2:].split("\\"")[0]
4 def check_website(result, name):
5     if name not in result[1]:
6         return False
7     return True

```

---

### 4.2.3 通配查询：支持用户输入正则表达式进行模糊匹配

我们将在本小节中实现功能以支持正则表达式的搜索，并在下一小节短语查询中实现更广义的通配查询，即判断文档中是否完全匹配、部分匹配、不匹配输入的词项等。

我们把正则表达式搜索集成在了简单查询中，即对用户输入的字符串判断其中是否含有正则表达式匹配符，如果不含有那么和我们上一节中的函数功能一致，如果含有那么就缓存这个字符串的一个副本，接着去除原字符串中所有的正则匹配符，进入一轮简单查询，最后对简单查询的结果根据原缓存字符串（即正则匹配中的模式）进行匹配，如果匹配成功就保留结果，否则就删去，在 simple\_search 中改动的代码如下：

---

```

1 # 对输入的查询字符串进行预处理，判断是否开启正则模式
2 regex = r'[\.\^\$\*\+\?\{\}\[\]\|\(\)]'
3 origin_input = input
4 isRe = re.search(regex, input)
5 if isRe is not None:
6     input = re.sub(regex, '', input)
7 # ..... 普通查询逻辑，同上
8 # 根据是否开启正则匹配进行不同的处理
9 # 如果开启正则模式，那么要进行一轮筛选，对模式串和文档的内容进行匹配
10 if isRe is not None:
11     for item in ls:
12         row = allInfo.loc[item[0]]
13         if re.search(origin_input, str(row.title)) is not None or re.search(origin_input,
14             str(row.description)) is not None or re.search(origin_input, str(row.content)) is not None:
15             ans.append(item)
16 if isRe is None:
17     return ls
18 return ans

```

---

#### 4.2.4 短语查询：完全匹配、部分匹配、不匹配输入的若干词项

在短语查询中，我们主要实现三种功能：

- **完全匹配**：对用户输入的若干词项，找到包含全部这些词的文档
- **部分匹配**：对用户输入的若干词项，找到至少包含这些词其中之一的文档
- **不匹配**：对用户输入的若干词项，找到不包含全部这些词的文档

它们的实现都很简单，逻辑也很类似，以完全匹配和部分匹配为例（它们是一个函数，根据参数区别）：

---

```

1 def check_match_words(result, input, complete=True):
2     row = allInfo.loc[result[1]]
3     text = f'{row['title']}#{row['description']}#{row['content']}#{row['editor']}'
4     ls = str(input).split(" ")
5     for word in ls:
6         if word == '#':
7             pass
8         if word not in text:
9             if complete == True:
10                 return False
11         if word in text:
12             if complete == False:
13                 return True
14     if complete == True:
15         return True
16     return False

```

---

在后续搭建 Web 页面时，我们会通过一个表单来操控这些功能，会提示用户将输入的词项用空格进行分割，因此这里调用 `split` 后会得到一个列表，接着判断是否满足条件即可，代码逻辑十分简单。

#### 4.2.5 标题查询：可以限制仅在标题中进行查询

由于我们在数据处理中对没加入描述以及正文的数据进行了一次处理，因此我们还有另一套仅含标题的工作集，用户到时可以根据表单中的单选框选择是在全文中进行检索还是仅在标题中进行检索，这部分的代码都是塞在各处的一个布尔判断，然后在不同的工作集上进行查询，在此不赘述代码。

### 4.3 查询日志

对于查询日志功能的实现，我们主要做了两份工作。一份工作是记录了用户的搜索历史，并通过 cookie 形式存储，有效期为 30 天，当用户再次打开搜索引擎时，会看到自己 30 天内的全部浏览记录。此外，这些浏览记录也方便了我们后续实现个性化查询和个性化推荐的功能。

---

```

1 # 搜索引擎初始化，加载搜索历史
2 if request.cookies.get('search_history'):
3     # 从 cookie 中获取搜索历史
4     search_history: list = json.loads(request.cookies.get('search_history'))
5     # ..... 若干搜索和过滤操作

```

```

6   # 如果以前没搜过，那么更新搜索历史
7   if words not in search_history:
8     search_history.append(words)
9   if len(search_history) > 10:
10    search_history.pop(0)
11 # 重新设置 cookie，下一次发生请求时就能看到这次的记录
12 resp.set_cookie('search_history', json.dumps(search_history), max_age=60 * 60 * 24 * 30)

```

另一份工作是真的”日志”，由于我们后续会用 Flask 搭建框架，我们每发生一次前后端的交互，在终端中都会产生一些日志输出，而 Flask 给了我们一个接口，我们可以将这些输入保存到本地的一个文件当中，这样可以时时查看用户的操作以及我们系统产生 bug 的原因：

```

1 # 设置日志记录器
2 if not app.debug:
3   file_handler = RotatingFileHandler('search.log', maxBytes=1024 * 1024, backupCount=1)
4   file_handler.setFormatter(logging.Formatter(
5     '%(asctime)s %(levelname)s: %(message)s [in %(pathname)s:%(lineno)d]')
6   ))
7   file_handler.setLevel(logging.INFO)
8   app.logger.addHandler(file_handler)
9   app.logger.setLevel(logging.INFO)
10  app.logger.info('Flask application started')

```

```

2023-12-14 20:27:30,638 INFO: Request: GET http://127.0.0.1:5000/bootstrap/static/js/bootstrap.min.js [in D:\documents\Github\Information-Retrieval\Search Engine\Web\_init_.py:27]
2023-12-14 20:27:30,639 INFO: Request: GET http://127.0.0.1:5000/static/js/recommend.js [in D:\documents\Github\Information-Retrieval\Search Engine\Web\_init_.py:27]
2023-12-14 20:27:30,641 INFO: Request: GET http://127.0.0.1:5000/static/img/bg.png [in D:\documents\Github\Information-Retrieval\Search Engine\Web\_init_.py:27]
2023-12-14 20:27:30,990 INFO: Request: GET http://127.0.0.1:5000/suggest?keywords=运动会 [in D:\documents\Github\Information-Retrieval\Search Engine\Web\_init_.py:27]
2023-12-14 20:27:30,992 INFO: Request: GET http://127.0.0.1:5000/personalized_recommendation [in D:\documents\Github\Information-Retrieval\Search Engine\Web\_init_.py:27]
2023-12-15 08:54:43,832 INFO: Flask application started [in D:\documents\Github\Information-Retrieval\Search Engine\Web\_init_.py:38]
2023-12-15 08:54:49,737 INFO: Flask application started [in D:\documents\Github\Information-Retrieval\Search Engine\Web\_init_.py:38]
2023-12-15 08:54:51,321 INFO: Request: GET http://127.0.0.1:5000/ [in D:\documents\Github\Information-Retrieval\Search Engine\Web\_init_.py:27]
2023-12-15 08:54:51,343 INFO: Request: GET http://127.0.0.1:5000/bootstrap/static/css/bootstrap.min.css [in D:\documents\Github\Information-Retrieval\Search Engine\Web\_init_.py:27]

```

图 4.3: Flask 日志输出重定向，一些交互示例

## 4.4 网页快照

网页快照是指对网页内容在某一特定时间点的完整拷贝或保存状态，其主要功能是提供当时的备份：网页快照为用户提供了网页在被搜索引擎爬取时的样子，并且搜索引擎在搜索结果中通常会提供一个链接到网页快照的选项。这使用户能够查看在索引时网页的内容，即使原网页已经改变或不再存在。因为我们在前面的爬虫工作中获取了完整的 HTML 代码，因此我们只需要通过某种手段，将其重新渲染给用户即可。

我们使用 Flask 搭建 Web 框架，对网页快照功能建立了一个路由’/snapshot’，它会从我们爬虫阶段整理好的 **htmls** 文件夹下获取指定的 HTML 文档，并通过模板重新渲染给用户。和大多数的搜索引擎一样，我们的快照服务对于一些音视频资源也需要从原始服务器上获取，如果获取不到则不会显示(因为我们没有在本地保存这些资源，并且保存这些资源太浪费空间，它们比 HTML 代码要大很多)，实现代码如下：

```

1 @front.route('/snapshot') # front 是我们为 Flask 注册的蓝图名
2 def _snapshot():

```

```

3   if url := request.args.get('url'):
4       title = allInfo.loc[url]['title']
5       with open(rf'../spider/htmls/{title}.html',encoding='utf-8') as f:
6           snapshot = f.read()
7           # 向前端以网页的形式返回快照
8           return render_template(r'snapshot.html', snapshot=snapshot)
9   else:
10      return "invalid arguments!"

```

---

## 4.5 个性化查询

在我们的个性化查询中，采用指导书提出的第二种方法，即记录用户的查询历史，通过历史查询来提供个性化的查询结果。在前面我们也提到，我们在前端是采用 cookie 进行的存储(有效期为 30 天)，会记录用户一段时间内的行为，且不会因为我们的程序结束了结果就消失。历史记录提供个性化结果的原理是，前端每次会把历史记录也传递给后端，然后通过历史行为赋予查询一些额外的权重，即最终的结果会受相似度、PageRank 和历史行为三者的影响。

具体的实现思路是：在用户查询时，不仅对输入的字符串进行分词，还会对前端传来的历史记录也进行分词并且加权到最终的相似度上(当然这个权重会比较小)，起到历史影响的作用，关键代码如下：

```

1 # 最终完整的函数头
2 def simple_search(input: str, history: list, onlyTitle: bool = False,num:int = 100):
3     # ..... 对输入进行分词，并计算 TF-IDF
4     spilt_history = []
5     for i in range(len(history)):
6         ls = list(cut_for_search(history[i]))
7         ls = [term for term in ls if term not in ["", " "]]
8         spilt_history.extend(ls)
9     # 计算历史记录的 TFIDF
10    tf_history = getTF(words,spilt_history)
11    tfidf_history = getTF_IDF(tf_history,idf_dict)
12    key_history = sorted(tfidf_history.items(),key=lambda item:item[1],reverse=True)[0:num]
13    len_key_history = getVecLength(key_history)
14    # ..... 对输入的其他处理，在普通查询中已经介绍过
15    # 如果存在历史记录，那么也计算一个相似度，之后加权平均得到总相似度
16    if len(history) > 0:
17        history_results_dict = {}
18        for item in key_results_index:
19            length = 0
20            temp_list = key_tfidf_dict_values[item]
21            for _key_history in key_history:
22                if _key_history[1] != 0:
23                    for value in temp_list:
24                        if _key_history[0] == value[0]:
25                            length = length + _key_history[1]*value[1]

```

```

26         sim = round(length/(len_key_history*getVecLength(temp_list)),4)
27         history_results_dict[item] = ((key_tfidf_dict_keys[item],sim))
28 # 加权平均，得到总相似度
29 for i in range(len(key_tfidf_dict_keys)):
30     if key_results[i][1] == 0:
31         pass
32     elif j:=history_results_dict.get(i):
33         # 设置历史记录的权重为 0.1
34         results.append((key_results[i][0],key_results[i][1]+j[1]/10))
35     else:
36         results.append((key_results[i][0],key_results[i][1]))
37 results = sorted(results,key=lambda item:item[1],reverse=True)
38 # ..... 对返回结果的后续处理
39
40 # 前端向后端传递数据：当产生搜索行为时，从 cookie 获取历史记录，和其他参数一起打包传递给 simple_search
41 @front.route('/advanced_search', methods=['GET', 'POST'])
42 def _advanced_search():
43     form = AdvancedSearchForm(is_title_only='全部网页')
44     if form.validate_on_submit(): # 提取表单数据
45         t = time.perf_counter()
46         all_these_words = form.all_these_words.data
47     if request.method == 'GET':
48         if request.args.get('keywords'):
49             form.all_these_words.data = request.args.get('keywords')
50     if request.cookies.get('search_history'):
51         # 从 cookie 中获取搜索历史
52         search_history: list = json.loads(request.cookies.get('search_history'))
53     else:
54         search_history = []
55     result_list = simple_search(all_these_words, search_history)
56     # 其他高级搜索过滤并返回结果，在此不作赘述

```

我们将个性化查询的功能，即受历史记录影响的查询功能也集成在了普通查询中，更详细的代码以及注释请参考 `search.py` 中的 `simple_search` 函数，在此不作赘述。

# 第5章 Web 页面及结果展示

## 5.1 Web 页面总体展示与搭建过程简述

我们采用简单的 Flask 框架进行搭建，并使用其内置的 jinja 框架系统进行基础渲染，再通过一些 css 和 js 进行一些额外的网页美化与交互功能实现，我们在框架中主要实现了四个页面：分别是首页面（普通查询页面）、高级搜索页面、快照页面以及结果页面，最终的效果图如下面四张图所示：

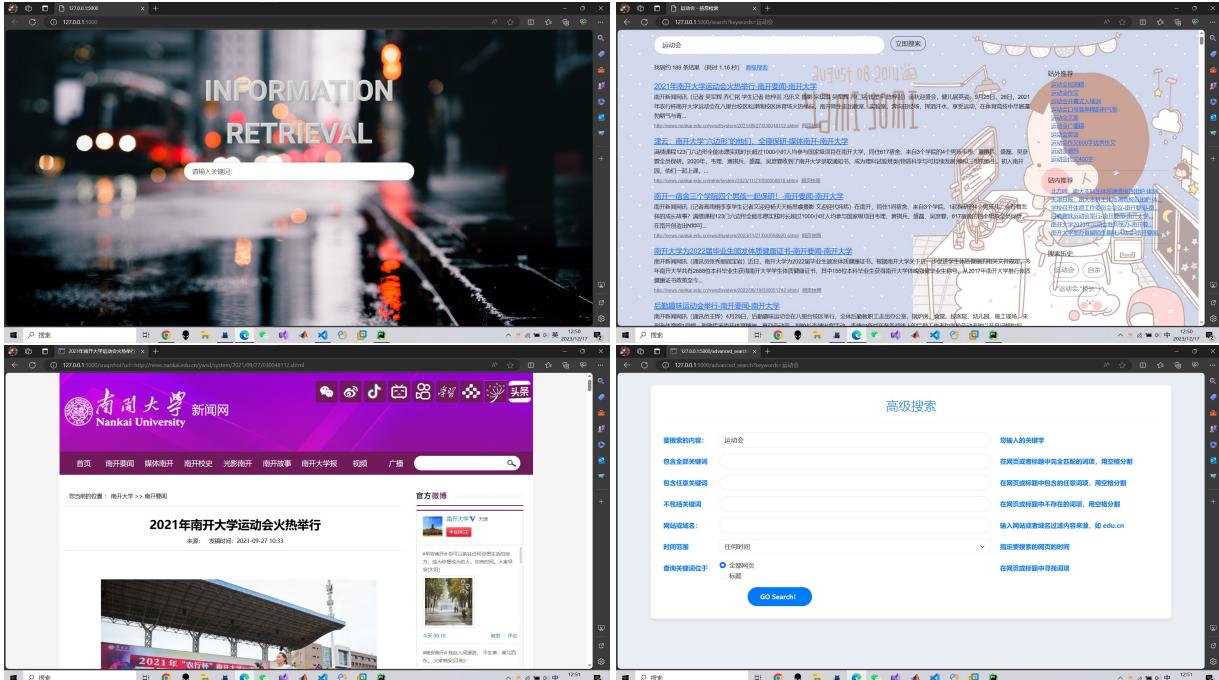


图 5.1: Web 框架搭建效果：从左到右从上到下依次为：主页面、结果页面、快照页面、高级搜索页面

由于 Web 页面的搭建并非此次实验的重点，在此不花大篇幅讲解搭建过程。简单讲解一下搭建思路和流程：总的来说，我们在 **Web.templates** 模块下存放了要展示的网页，这些 HTML 文件是网页的基本框架，但它们没有要展示的数据，那么这些数据从哪里获取呢？我们在 **Web.front** 模块下定义了这些网页获取它们数据的逻辑，比如在主页面中，我们将鼠标放在搜索框时会弹出搜索历史列表，这就是这个页面所需要的全部数据，那么我们在对应的 `Web.front.index.py` 中创建好 Flask 路由后，只需要拿到数据传送给 `index.html` 即可：

```
1 @front.route('/')
2 def _index():
3     # 从 cookie 中获取搜索历史
4     if request.cookies.get('search_history'):
5         search_history: list = json.loads(request.cookies.get('search_history'))
6     else:
7         search_history = []
8     # 利用 Flask, 便捷的传递数据给对应的 HTML
9     return render_template(r'index.html', search_history=search_history)
```

相应地，我们在 `webSearch.py` 中定义了高级搜索页面所需要的资源获取逻辑以及结果返回逻辑；在 `result.py` 里定义了结果页面的资源获取逻辑，在 `snapshot.py` 中定义了网页快照的逻辑（即上哪里去找当初爬下来的 HTML），

这四个文件就是上面四张图的最主要的效果实现。此外，我们还有一个 suggest.py，它用于实现后面要说的个性化推荐，我们放到之后详细说明，这部分的数据展现不是单独一个网页，而是结果页面右边的那一栏内容。

## 5.2 具体结果展示

下面，我们进行一些结果演示，验证前面实现的各种功能的实现效果。

### 一、普通查询及正则匹配结果演示

普通查询支持用户输入一段文字，或者是一个正则表达式进行模糊匹配，比如我们搜索运动会：



图 5.2: 普通查询结果展示

可以看到在 5644 个网页中很快返回了 189 条结果，并且越靠前的网页内容越相关。在基础搜索页面，我们也可以输入正则表达式，比如我们搜索 `.* 运动会.* 校长`，这会搜索先出现运动会后出现校长的所有网页（但一般没人这么干，没有顺序限制的在高级搜索的短语查询中实现），效果如图 5.3 所示。

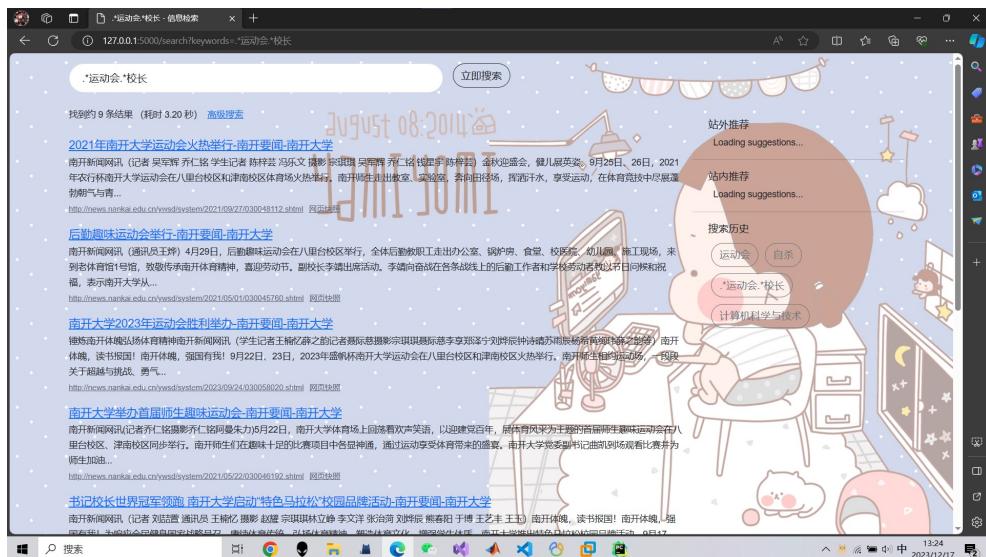


图 5.3: 高级搜索：正则表达式模糊匹配结果展示

只剩下了九个网页，并且它们的网页内容确实都可以对正则表达式进行匹配。

## 二、各种高级搜索功能结果展示

**站内搜索：**加入我们现在只想要媒体南发布的有关运动会的内容，而不想要南开要闻版面的，可以在基础搜索完成之后，点击高级搜索链接，然后在网站或域名那一栏填写 **mtnk**，高级搜索页面以及过滤来源后的结果如下所示：

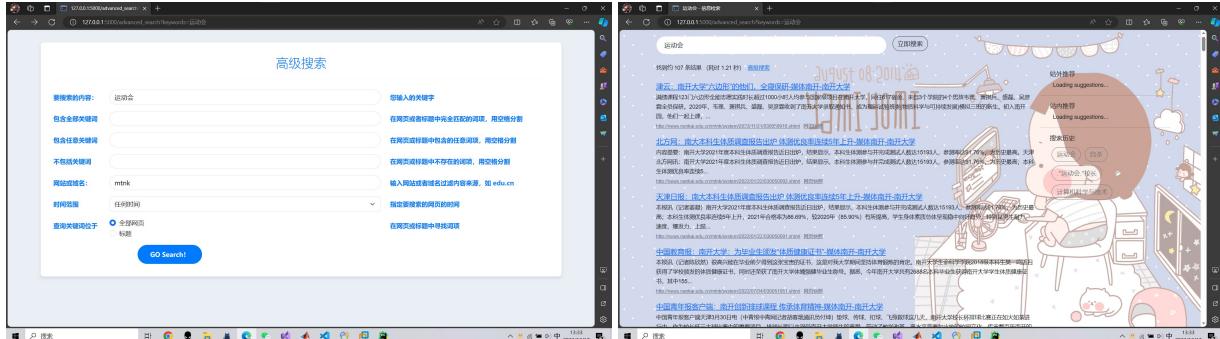


图 5.4: 高级搜索: 站内查询结果演示

可以看到，只剩下 107 条结果，过滤了 80 多个来自南开要闻版面的结果，剩下的结果均来源于媒体南开。

**短语查询：**我们还是以上面的**运动会、校长为例**，先来测试一下**完全匹配**，之前的正则式的含义是运动会必须在校长前面，而我们在**包含全部关键词**一栏中以空格分割输入运动会、校长时，它们是没有顺序要求的，这也更符合人们的搜索习惯，如下图所示：



图 5.5: 高级搜索: 短语查询——完全匹配结果演示

可以看到，确实比正则匹配多返回了一些结果，下面以这个结果为基础，测试**不匹配**：搜索结果的第二条有“后勤字样”，我们在**不包括关键词**这一栏填入后勤，结果不再含有后勤词项的网页：



图 5.6: 高级搜索: 短语查询——不匹配结果演示

部分匹配也类似，它是比前面的完全匹配和完全不匹配更弱的条件，效果也正常：

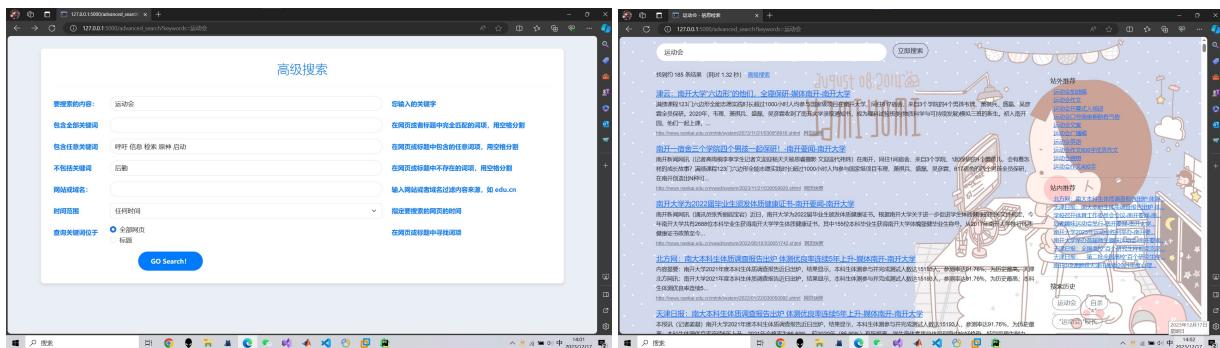


图 5.7：高级搜索：短语查询——部分匹配结果演示

**时间限制及标题搜索：**可以看到，在这个表单中有一个检索位置，默认是搜索全部内容，我们也可以在高级搜索中调整其为仅出现在标题中，比如我们仅搜索出现在标题中的与运动会有关的网页，结果如下，可以看到明显少了很多：



图 5.8：高级搜索：标题检索结果演示

此外还有带时间限制的检索，我们为用户提供了不限制、一个月内、一年内三种选项，下面我们把时间限制为一年之内发布的网页，结果如下，实现正常：

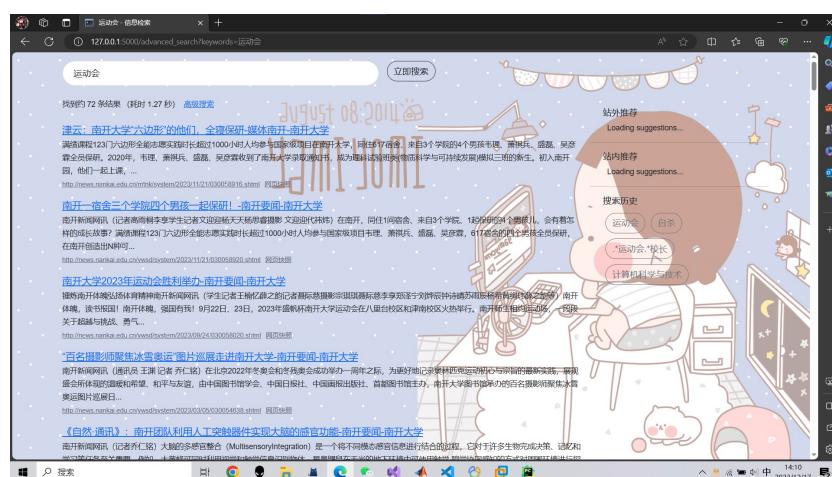


图 5.9：高级搜索：时间检索结果演示

### 三、网页快照、历史记录、查询日志

在结果页面的每个结果项中，都提供了查看网页快照的功能，它会显示我们本地当时爬取来的网页：

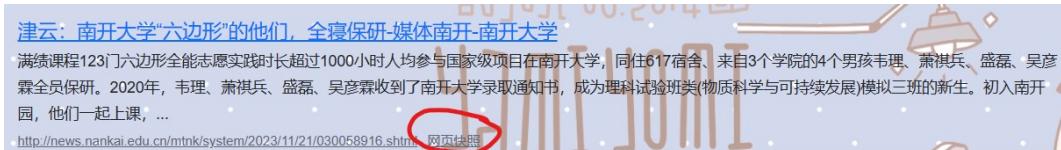


图 5.10: 网页快照功能

点开网页快照，会从本地拿 HTML 进行渲染，而图片等资源会到原服务器进行获取，如果原服务器发生了改变或者删除了资源，那么就不会显示，此外使用网页快照的打开速度也比点链接要快一点：



图 5.11: 点击网页快照页结果演示

对于历史记录，它帮助了我们实现了个性化推荐的功能以及部分查询日志的功能(还有后面的个性化推荐功能)，我们把他放在了主搜索页面上进行展示，保留最近的十条搜索记录，有效期为 30 天：

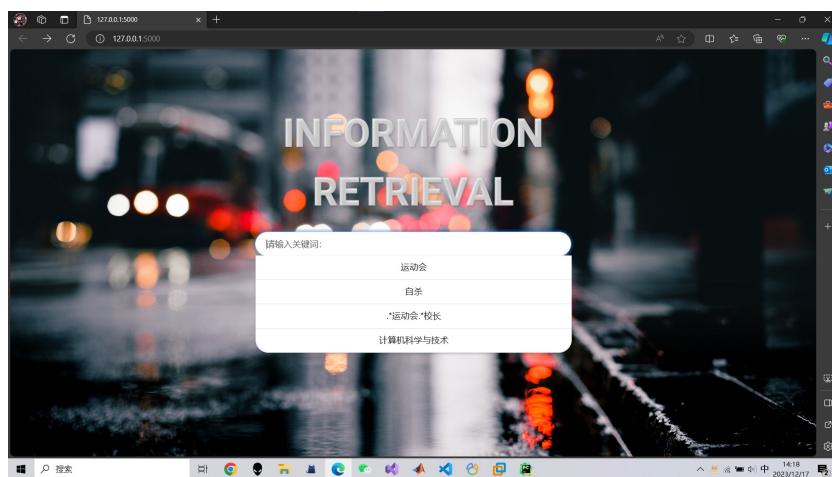


图 5.12: 历史记录演示

我们还在 Web 目录下保存了一份查询日志，是 Flask 的操作记录，在前文有所展示，在此不加赘述。

## 第6章 个性化推荐

我们的个性化推荐分为[站外推荐](#)和[站内推荐](#)，站外推荐的思路就是，虽然我们有5600多个网页，但内容太过局限，所以我们考虑从外部获取一些新鲜的网页返回给用户，具体做法就是根据用户此次的输入内容，从其他网站上进行联网查询，返回最优的几条结果（标题名+跳转链接），同时为了用户的体验，不会等到这部分内容加载完才展示结果页，而是通过一个window.onload允许后加载；而站内推荐的思路就是，根据用户的搜索历史，在网页加载之后对搜索历史进行一些查询，[返回用户曾经感兴趣的一些话题](#)，我们以站内推荐为例讲解一下实现的做法，首先是后端的处理，代码如下所示：

---

```
1 # 实现站内个性化推荐
2 @front.route('/personalized_recommendation')
3 def personalized_recommendation():
4     search_history = json.loads(request.cookies.get('search_history', '[]'))
5     recommendations = []
6     for query in search_history:
7         results = simple_search(query, [])[:3]
8         results = [result[0] for result in results]
9         urls = []
10        for i in range(len(results)):
11            urls.append(allInfo.loc[results[i]]['title'])
12        recommendations.extend([[results[i], urls[i]] for i in range(len(results))])
13        if len(recommendations) > 8:
14            break
15    return jsonify(recommendations)
```

---

后端的工作就是读取用户历史记录，然后拓展前端要展示的信息，最后jsonify交付给前端；而前端的工作就是按照json文件的格式，读取数据并组合成链接标签a进行展示，代码如下所示：

---

```
1 function personal_suggest() {
2     let suggestion_dom = document.getElementById('personal_suggestion');
3     suggestion_dom.innerHTML = '<li>Loading suggestions...</li>';
4     $.ajax({
5         type : "GET",
6         async: true,
7         url : "/personalized_recommendation",
8         success : function(data){
9             let tag = '';
10            for (let i = 0; i < data.length; i++) {
11                let url = data[i][0];
12                let name = data[i][1];
13                if(name.length>20) {
14                    name = name.substr(0,20);
15                }
16            }
17        }
18    })
19}
```

---

```

16     name += "...";
17     tag += `<li><a href="${url}" target="_blank">${name}</a></li>`;
18   }
19   suggestion_dom.innerHTML = tag;
20   console.log(tag);
21 },
22 error: function(){
23   suggestion_dom.innerHTML = '<li>Error loading suggestions</li>';
24 }
25 });
26 }

```

同时如上所述，为了用户的体验，不必等待这个操作结束在展示网页：

```

1 <script>
2   window.onload = function () {
3     query_suggest('{{ keywords }}'); // 站外推荐
4     personal_suggest(); // 站内推荐
5   }
6 </script>

```

站外推荐的实现过程和它类似，只不过是多了一个访问外部网页的过程，和爬虫阶段的处理差不多，在此不再赘述，详细代码可以参考 **recommend.js** 和 **suggest.py**，下面进行个性化推荐展示：

我们来搜索计算机科学与技术，右侧个性化推荐栏的结果如下图所示：

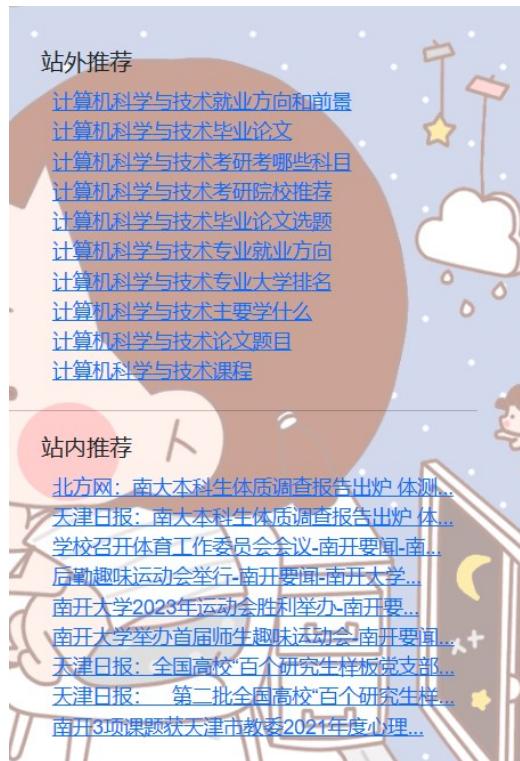


图 6.1: 个性化推荐演示

可以看到，站外推荐的内容和搜索内容大幅度相关，我们随便点开一个，会跳转到外部网页：



图 6.2: 个性化推荐演示：站外推荐效果

可以看到，通过站外的个性化推荐，极大的丰富了用户的检索需求。下面来看站内的推荐：由于我们前面的演示搜索了大量的运动会词项，因此根据历史记录，站内个性化推荐会选择最近的几条搜索记录中，综合得分最高的网页进行推荐，这部分内容是用户曾经感兴趣的：



图 6.3: 个性化推荐演示：站内推荐效果