

信息系统检索

HW5 Web搜索引擎

姓名： 蒋浩南 学号： 2012948

- 一、网页抓取
- 二、文本索引
- 三、链接分析
- 四、查询服务
 - 1. 站内查询
 - 2. 通配查询
 - 3. 短语查询
 - 4. 用户功能
 - 5. 查询日志
 - 6. 网页快照
- 五、个性化查询
- 六、个性化推荐
- 七、Web页面，图形化界面

一、网页抓取

功能实现：

1. 从 `base_url` 开始。

- 每次访问网页，将该网页的源代码（html文件）保存到本地，为之后实现网页快照功能。
- 提取该网页的content，保存到本地。
- 提取该网页的title，加入 `url_title_list`，url和title的列表。
- 提取该网页的锚文本，并将对应的url，加入 `url_anchor_list`。同时将url加入到探索的url栈中，以继续探索。以及设置该网页指向的网页信息（`url_linkto`），为之后PageRank提供数据。（使用`IdMap()`实现对url的数字对应）

```
def get_html(url):
    # print("try to get: ", url)
    try:
        temp = requests.get(url, timeout=2)
        temp.encoding = 'utf-8'
    except:
        return None
    return temp

base_url = 'http://cc.nankai.edu.cn'
dir_path = "C:/Users/nan/Desktop/Web_Search_Engine/data/"
url_id_map = IdMap() # url to id

url_anchor_list=[]      #url和锚文本的列表
url_title_list=[]       #url和title的列表
url_linkto={}           #url的link
```

```

url_list=[] #url的栈
url_done=[] #已经探索过的url
url_list.append(base_url)
max_loop = 100

def spider():
    cur_loop=0
    while cur_loop<max_loop and len(url_list)!=0:
        cur=url_list[0]
        print("cur_loop",cur_loop)
        #防止重复探索一个url
        if cur in url_done:
            url_list.remove(cur)
            continue
        print("cur : ",cur)
        r=get_html(cur)
        if r==None:
            url_list.remove(cur)
            continue
        if r.status_code!=200:
            url_list.remove(cur)
            continue

        soup = BeautifulSoup(r.text, "lxml")

        url_id=url_id_map[cur]
        #source code

        doc=open(os.path.join(dir_path,'source_code/'+str(url_id)+'.html'),'w',encoding
='utf-8')
        doc.write(r.text)
        doc.close()

        #add title_url
        try:
            title = soup.find('title').text
            print(title)
            url_title_list.append([title,cur])
            print(url_title_list)
        except :
            url_list.remove(cur)
            continue

        #content
        data_list = soup.select('head')
        content=''
        for item in data_list:
            text=re.sub('[\r \n\t]', '', item.get_text())
            text = re.sub('[\s]', '', text)
            if text==None or text=='':
                continue
            content+=text

```

```

data_list = soup.select('body')
for item in data_list:
    text=re.sub('[\r \n\t]', '', item.get_text())
    text = re.sub('[\s]', '', text)
    if text==None or text=='':
        continue
    #print(text)
    content += text
    #print(content)

#remove
if os.path.exists(os.path.join(dir_path, 'content/' + str(url_id) +
'.txt')):
    os.remove(os.path.join(dir_path, 'content/' + str(url_id) + '.txt'))
    print('remove sucessful')
else:
    print("not exists")

with open(os.path.join(dir_path, 'content/' + str(url_id) + '.txt'),
'w',encoding='utf-8') as doc:
    doc.write(content)

data_list= soup.find_all('a')
#print(title_list.text)
linkto=[]
for item in data_list:
    #锚文本
    anchor = item.text.strip().replace('
','').replace('\n','').replace('\t','')
    #链接url
    str_url = str(item.get('href'))
    #print(str_url)
    if "http" in str_url:

        url_anchor_list.append([anchor, str_url])
        url_list.append(str_url)
        temp_url_id=url_id_map[str_url]
        linkto.append(temp_url_id)

    elif "htm" in str_url :

        com_str=str(base_url + str_url)
        url_anchor_list.append([anchor, com_str])
        url_list.append(com_str)
        temp_url_id=url_id_map[com_str]
        linkto.append(temp_url_id)

url_done.append(cur)
url_linkto[url_id_map[cur]]=linkto
url_list.remove(cur)
cur_loop+=1

```

2. 保存

对爬取过程中的数据进行保存。

```
def save():
    #####anchor_url#####
    # remove
    if os.path.exists(os.path.join(dir_path, 'anchor_url.csv')):
        os.remove(os.path.join(dir_path, 'anchor_url.csv'))
        print('remove sucessful')
    else:
        print("not exists")

    # save anchor_url
    df1 = pd.DataFrame(data=url_anchor_list, columns=["anchor", "url"])
    df1.to_csv(os.path.join(dir_path, 'anchor_url.csv'), encoding='utf_8_sig')
    #####

    #####title_url#####
    # remove
    if os.path.exists(os.path.join(dir_path, 'title_url.csv')):
        os.remove(os.path.join(dir_path, 'title_url.csv'))
        print('remove sucessful')
    else:
        print("not exists")

    # save anchor_url
    df1 = pd.DataFrame(data=url_title_list, columns=["title", "url"])
    df1.to_csv(os.path.join(dir_path, 'title_url.csv'), encoding='utf_8_sig')
    #####

    #序列化
    try:
        os.mkdir(os.path.join(dir_path, "pkl_dir"))
    except FileExistsError:
        pass
    #url_id_map
    with open(os.path.join(dir_path, "pkl_dir/" + 'url_id_map.pkl'), 'wb') as
doc:
        pickle.dump(url_id_map, doc)

    #url_linkto_dict
    with open(os.path.join(dir_path, "pkl_dir/" + "url_linkto_dict.pkl"), 'wb')
as doc:
        pickle.dump(url_linkto, doc)
```

二、文本索引

文本索引的构建分为两部分：

- 第一部分使用 `whoosh`, 分别对Title、锚文本、url和网页的总内容建立索引。由于直接通过爬取的原始数据构建索引，在后面实现短语查询。
- 第二部分为通过 `TfidfVectorizer` 对不同域建立向量空间模型。实现其他的查询功能。

1. 如下为第一部分，使用 `whoosh` 建立索引并保存至本地。使用第一部分使用爬虫爬取的数据。以下四个函数分别为对title、锚文本、url和content建立索引的函数。基本相同，都是读取对应数据，创建索引结构，对原数据进行一定的处理后，加入索引中。

```
def build_Title_Index():

    id_map=read_IdMap(dir_pkl_path+'/'+ 'url_id_map.pkl')
    title_url=read_csv(dir_path+'/'+ 'title_url.csv')
    title_url=title_url.values.tolist()

    schema = Schema(url=NUMERIC(stored=True), title=TEXT(stored=True,
analyzer=ChineseAnalyzer())) # 创建索引结构

    ix = create_in(os.path.join(dir_index_path), schema=schema,
indexname='url_title_source')
    writer = ix.writer()

    for term in title_url:

        if term[1] != '':
            url_temp=id_map[term[2]]
            title_temp=str(term[1])
            title_temp = re.sub(r"[{}、,。! ? · 【】 ) 》 ; ; - 《 ” :
(-)"+".format(punctuation), "", title_temp)
            title_temp = title_temp.lower()
            #words = ' '.join(jieba.lcut_for_search(title_temp))
            writer.add_document(url=url_temp, title=title_temp )
    writer.commit()

def build_Anchor_Index():

    id_map=read_IdMap(dir_pkl_path+'/'+ 'url_id_map.pkl')
    anchor_url=read_csv(dir_path+'/'+ 'anchor_url.csv')
    anchor_url=anchor_url.values.tolist()

    schema = Schema(url=NUMERIC(stored=True), anchor=TEXT(stored=True,
analyzer=ChineseAnalyzer())) # 创建索引结构

    ix = create_in(os.path.join(dir_index_path), schema=schema,
indexname='url_anchor_source')
    writer = ix.writer()

    for term in anchor_url:

        #term = re.sub(r"[{}、,。! ? · 【】 ) 》 ; ; - 《 ” :
(-)"+".format(punctuation), "", term)
        if term[1] != '':
            url_temp = id_map[term[2]]
            anchor_temp = str(term[1])
            anchor_temp = re.sub(r"[{}、,。! ? · 【】 ) 》 ; ; - 《 ” :
(-)"+".format(punctuation), "", anchor_temp)
            anchor_temp = anchor_temp.lower()
            #words = ' '.join(jieba.lcut_for_search(anchor_temp))
            print(id_map[term[2]],term[1])
            writer.add_document(url=url_temp, anchor=anchor_temp )
    writer.commit()
```

```

def build_url_index():

    id_map=read_IdMap(dir_pkl_path+'/'+ 'url_id_map.pkl')
    # anchor_url=read_csv(dir_path+'/'+ 'anchor_url.csv')
    # anchor_url=anchor_url.values.tolist()

    # schema = Schema(url=NUMERIC(stored=True), anchor=TEXT(stored=True,
analyzer=ChineseAnalyzer())) # 创建索引结构
    schema = Schema(url=NUMERIC(stored=True), url_text=TEXT(stored=True)) # 创建
索引结构

    ix = create_in(os.path.join(dir_index_path), schema=schema,
indexname='url_source')
    writer = ix.writer()

    for term in (id_map.str_to_id).keys():
        # print(id_map[term],term)
        writer.add_document(url=id_map[term], url_text=term )
    writer.commit()

def build_Content_Index():

    #id_map=read_IdMap(dir_pkl_path+'/'+ 'url_id_map.pkl')

    schema = Schema(url=NUMERIC(stored=True), content=TEXT(stored=True,
analyzer=ChineseAnalyzer())) # 创建索引结构

    ix = create_in(os.path.join(dir_index_path), schema=schema,
indexname='url_content_source')
    writer = ix.writer()

    max_doc = 108
    temp_doc=0
    while temp_doc<max_doc:
        doc=str(temp_doc)+'.txt'
        print(dir_content_path + '/' +doc)
        doc_dir=dir_content_path + '/' + doc

        term=''
        try:
            for content in open(dir_content_path + '/' + doc, encoding='utf-
8').readlines():
                # print(content)
                content = re.sub(r"[{}、,。！？·【】）》；；-《“”：
(-)]+ ".format(punctuation), "", content)
                content = content.lower()
                term=term+content
                writer.add_document(url=temp_doc, content=term)
                temp_doc+=1
        except:
            temp_doc+=1
            continue

```

```
writer.commit()
```

2. 如下为第二部分，使用 `TfidfVectorizer` 对不同域建立向量空间模型。

- 实现思路都为首先对原始的数据进行处理（如去掉中文符号，最小化），生成每个url（网页）对于的list，之后组合形成 `doc_list`。建立 `tfidf_vectorizer = TfidfVectorizer(min_df=1)`，之后 `tfidf_matrix = tfidf_vectorizer.fit_transform(doc_list)` 得到 `tfidf_vectorizer` 为之后生成查询向量。 `tfidf_matrix` 为tfidf矩阵，为之后计算分数。
`tfidf_vectorizer.get_feature_names_out()` 为词袋，为之后通配查询时匹配用。序列化保存。

```
def genContentVSM():
    doc_list = []
    url_list = []
    temp_doc=0

    while temp_doc<max_doc:
        doc=str(temp_doc)+''.txt'
        print(dir_content_path + '/' + doc)

        try:
            for content in open(dir_content_path + '/' + doc, encoding='utf-8').readlines():
                content = re.sub(r"[{}、,。！？·【】）》；；-《”：(-)]+", "", content)
                content = content.lower()
                words = ' '.join(jieba.lcut_for_search(content))
                doc_list.append(words)
        except:
            content = ' '
            words = ' '.join(jieba.lcut_for_search(content))
            doc_list.append(words)
            temp_doc += 1
            url_list.append(doc[0:-4])
            continue
    temp_doc+=1

    tfidf_vectorizer = TfidfVectorizer(min_df=1)
    tfidf_matrix = tfidf_vectorizer.fit_transform(doc_list)
    print(tfidf_matrix)
    print(tfidf_vectorizer.get_feature_names_out())

    # print(tfidf_vectorizer.get_feature_names_out().shape)

    with open(os.path.join(dir_path, "pk1_dir/" + 'tfidf_vectorizer.pkl'), 'wb') as doc:
        pickle.dump(tfidf_vectorizer, doc)

    with open(os.path.join(dir_path, "pk1_dir/" + 'tfidf.pkl'), 'wb') as doc:
        pickle.dump(tfidf_matrix, doc)
```

```

with open(os.path.join(dir_path, "pkl_dir/" + 'words_bag.pkl'), 'wb') as
doc:
    pickle.dump(tfidf_vectorizer.get_feature_names_out(), doc)

def gen_Title_Anchor_VSM():

    title_dict, anchor_dict = csv_process()
    title_list = []
    for term in title_dict.keys():
        content = re.sub(r"[{}、,。！？·【】）》；；-《“”：
(-)]+".format(punctuation), "", title_dict[term])
        content = content.lower()
        words = ' '.join(jieba.lcut_for_search(content))
        title_list.append(words)
    print(title_list)

    anchor_list = []
    w = ' '.join(jieba.lcut_for_search(' '))
    anchor_list.append(w)
    for term in anchor_dict.keys():
        # if term > 250:
        #     break
        content = re.sub(r"[{}、,。！？·【】）》；；-《“”：
(-)]+".format(punctuation), "", anchor_dict[term])
        content = content.lower()
        words = ' '.join(jieba.lcut_for_search(content))

        anchor_list.append(words)
    print(anchor_list)

    t_tfidf_vectorizer = TfidfVectorizer(min_df=1)
    t_tfidf_matrix = t_tfidf_vectorizer.fit_transform(title_list)
    print(t_tfidf_matrix)
    print(t_tfidf_vectorizer.get_feature_names_out())

    a_tfidf_vectorizer = TfidfVectorizer(min_df=1)
    a_tfidf_matrix = a_tfidf_vectorizer.fit_transform(anchor_list)
    print(a_tfidf_matrix)
    print(a_tfidf_vectorizer.get_feature_names_out())

    with open(os.path.join(dir_path, "pkl_dir/" +
'title_tfidf_vectorizer_matrix.pkl'), 'wb') as doc:
        pickle.dump((t_tfidf_vectorizer, t_tfidf_matrix), doc)

    with open(os.path.join(dir_path, "pkl_dir/" +
'anchor_tfidf_vectorizer_matrix.pkl'), 'wb') as doc:
        pickle.dump((a_tfidf_vectorizer, a_tfidf_matrix), doc)

```


三、链接分析

- 使用包 `import networkx as nx`
- 读取网页爬取时保存的 `url_linkto`
- `G=nx.DiGraph()` 加边
- `pr = nx.pagerank(G, alpha=0.85)` 得到pageRank的矩阵
- 序列化保存，作为之后搜索排序时网页得分的一部分。

```
def pageRank():
    url_linkto=read_url_linkto()
    G=nx.DiGraph()
    #加边
    for i in url_linkto:
        for j in url_linkto[i]:
            G.add_edge(i,j)
    #计算，阻尼因子 0.85
    pr = nx.pagerank(G, alpha=0.85)
    print(pr)
    prdic = {}
    for node, pageRankValue in pr.items():
        prdic[node] = pageRankValue * 1e4
    print(prdic)

    #序列化
    with open(os.path.join(dir_pkl_path , 'pageRRank.pkl'), 'wb') as doc:
        pkl.dump(prdic, doc)
```

四、查询服务

查询服务实现主要在queryUnit.py的类Query中。

- 该类的初始化:

```
class Query:

    def __init__(self ,hobby=None):

        #content的tfidf和tfidf_vectorizer
        with open(os.path.join(dir_path, "pkl_dir/" + 'tfidf_vectorizer.pkl'),
'rb') as doc:
            self.content_tfidf_vectorizer=pkl.load(doc)
        with open(os.path.join(dir_path, "pkl_dir/" + 'tfidf.pkl'), 'rb') as
doc:
            self.content_tfidf=pkl.load(doc)
        # titleh 和anchor 的tfidf和tfidf_vectorizer
        with open(os.path.join(dir_path, "pkl_dir/" +
'title_tfidf_vectorizer_matrix.pkl'), 'rb') as doc:
            self.t_tfidf_vectorizer, self.t_tfidf_matrix= pkl.load(doc)

        with open(os.path.join(dir_path, "pkl_dir/" +
'anchor_tfidf_vectorizer_matrix.pkl'), 'rb') as doc:
            self.a_tfidf_vectorizer, self.a_tfidf_matrix=pkl.load(doc)
```

```

        #url_id_map
        self.url_id_map=IdMap.IdMap()
        with open(os.path.join(dir_path, "pk1_dir/" + 'url_id_map.pkl'), 'rb')
as doc:
            self.url_id_map = pickle.load(doc)

        #词袋
        with open(os.path.join(dir_path, "pk1_dir/" + 'words_bag.pkl'), 'rb') as
doc:
            self.words_bag = pickle.load(doc)
        #url的pagerank
        with open(os.path.join(dir_path, "pk1_dir/" + 'pageRRank.pkl'), 'rb') as
doc:
            self.pageRank = pickle.load(doc)

        self.hobby=hobby

```

- 该类利用函数 `query` 来调用类内的短语查询、通配查询和站内查询。
 - 函数输入：
 - `input_query`: 查询输入。
 - `query_type`: 查询的类型，判别是短语查询、通配查询还是站内查询。
 - `positin_type`: 查询位置，决定查询的域，是整个网页内容、仅title、仅锚文本还是仅url
 - `hobby`: 为当前登入用户的兴趣标签
 - `history`: 为当前登入用户的查询历史
 - 短语查询：
 - 调用该类的短语查询函数 `Query.parse_query`，得到查询得到的url的list。
 - 调用 `urlid_to_page`，输入url_list，返回web网页查询结果输出的数据格式。之后介绍。
 - 通配查询：
 - 调用该类的通配查询函数 `Query.wildcard_query`，得到查询的url_score的字典。
 - 调用该类的个性化查询函数 `Query.add_personal_queries`，修改上一步url_score部分url对应的分数。
 - 调用该类的pageRank函数 `Query.add_pageRank`，对url_score字典增加对于网页的pageRank分数。
 - 调用函数 `Query.query_result_sort`，得到排序后url的id的list
 - 调用函数 `urlid_to_page`，输入url_list，返回web网页查询结果输出的数据格式。之后介绍。
 - 站内查询：
 - 调用该类的站内查询函数 `Query.common_query`，得到查询的url_score的字典。
 - 调用该类的个性化推荐函数 `Query.add_personal_recommendation`，根据输入的hobby和history对上文的url_score字典进行扩充。增加推荐的网页。
 - 调用该类的个性化查询函数 `Query.add_personal_queries`，修改上一步url_score部分url对应的分数。
 - 调用该类的pageRank函数 `Query.add_pageRank`，对url_score字典增加对于网页的pageRank分数。
 - 调用函数 `Query.query_result_sort`，得到排序后url的id的list

- 调用函数 `urlid_to_page`，输入 `url_list`，返回web网页查询结果输出的数据格式。之后介绍。

```
def
query(self, input_query, query_type=0, positin_type=0, hobby=None, history=None):

    page=[]
    #短语查询
    if query_type==1:

        url_list=Query.pharse_query(self, query=input_query, position_type=positin_type)
        print('url_list', url_list)
        page=urlid_to_page(url_list)
        return page

    #通配查询
    if query_type==2:
        # 通配查询

        url_score=Query.wildcard_query(self, input_query, positin_type=positin_type)
        # 个性化查询

        Query.add_personal_queries(self, qr=url_score, hobby=hobby, history=history)
        # pageRank
        Query.add_pageRank(self, qr=url_score)
        # 得到排序后url的id的list
        sorted_score_id=Query.query_result_sort(self, url_score)
        page=urlid_to_page(sorted_score_id)
        return page

    #站内查询
    if query_type == 3:
        url_score =
        Query.common_query(self, input_query, positin_type=positin_type)

        # # 个性化推荐
        print('个性化推荐')
        Query.add_personal_recommendation(self, qr=url_score, hobby=hobby,
        history=history)

        # 个性化查询

        Query.add_personal_queries(self, qr=url_score, hobby=hobby, history=history)
        # pageRank
        Query.add_pageRank(self, qr=url_score)
        # 得到排序后url的id的list
        sorted_score_id=Query.query_result_sort(self, url_score)
        page=urlid_to_page(sorted_score_id)
        return page
```

上文中的函数 `urlid_to_page`，url的id的list生成页面显示的page结构。

为{'url:', 'title:', 'content:', 'id: ', []}

```
# url的id的list生成页面显示的page结构
def urlid_to_page(idlist):
    with open(os.path.join(dir_path, "pkl_dir/" + 'url_id_map.pkl'), 'rb') as doc:
        url_id_map = pickle.load(doc)

    title_df = buildIndex.read_csv(dir_path + '/' + 'title_url.csv')

    page=[]
    for id in idlist:

        page_item=[]
        page_item.append(url_id_map[id])

        index=title_df[title_df.url ==url_id_map[id]].index.tolist()[0]

        page_item.append(title_df.title.loc[index])

        for content in open(dir_content_path + '/' + str(id) + '.txt',
encoding='utf-8').readlines():
            page_item.append(content[:100])

        page_item.append(id)
        page.append(page_item)

    return page
```

上文中的函数 `add_pageRank` , 为网页增加pageRank权重。

```
#pageRank的分数加权
def add_pageRank(self,qr,para=100):
    # print(type(self.pageRank))
    # print(self.pageRank)

    for temp in qr.keys():
        # print(qr[temp],self.pageRank[temp])
        qr[temp]+=self.pageRank[temp]/para
        # print(qr[temp])
```

上文中的函数 `query_result_sort` , 根据网页分数对其排序, 返回排序后的url_list。

```

#排序
def query_result_sort(self,url_score):

    new_url_score = sorted(url_score.items(), key=lambda score: score[1],
reverse=True)
    sorted_score_id = [score[0] for score in new_url_score ]
    # new_url_score = sorted(url_score.values())
    print(new_url_score)
    print(sorted_score_id)
    # return new_url_score, sorted_score_id
    return sorted_score_id

```

之后对短语查询、通配查询、站内查询、查询日志和网页快照功能的实现做介绍。

1.站内查询

实现：

- 对输入查询文本的处理，去中文符号，lower，利用jieba分词，返回分词后的列表。
- 根据positin_type，即要查询的域，调用对应的 `tfidf_vectorizer`，得到查询的向量。
- 根据positin_type，即要查询的域，调用对应的 `tfidf_matrix`，vsm计算得分。
- 得到得分大于零的url

```

#常规站内查找
def common_query(self,query,positin_type=0,type='common'):
    # 输入文本处理
    if type!='wildcard':
        query = re.sub(r"[{}、,。！？·【】）》；；《”（-]+".format(punctuation), ""
        , query)
        query = query.lower()
        query_words = ' '.join(jieba.lcut_for_search(query))
        query = []
        query.append(query_words)
    print(query)

    # 得到输入向量
    if positin_type==0:
        new_term_freq_matrix = self.content_tfidf_vectorizer.transform(query)
        print(new_term_freq_matrix)
        query_vec = np.array((new_term_freq_matrix.todense().tolist())[0])
    if positin_type==1:
        new_term_freq_matrix = self.t_tfidf_vectorizer.transform(query)
        print(new_term_freq_matrix)
        query_vec = np.array((new_term_freq_matrix.todense().tolist())[0])
    if positin_type==2:
        new_term_freq_matrix = self.a_tfidf_vectorizer.transform(query)
        print(new_term_freq_matrix)
        query_vec = np.array((new_term_freq_matrix.todense().tolist())[0])

    # vsm计算得分
    # print(self.content_tfidf.shape[1])
    if positin_type==0:
        num_doc=self.content_tfidf.shape[0]

```

```

if positin_type==1:
    num_doc=self.t_tfidf_matrix.shape[0]
if positin_type==2:
    num_doc=self.a_tfidf_matrix.shape[0]

score = np.zeros(num_doc)

if positin_type==0:
    tf_idf=self.content_tfidf.toarray()
if positin_type==1:
    tf_idf=self.t_tfidf_matrix.toarray()
if positin_type==2:
    tf_idf=self.a_tfidf_matrix.toarray()
get_score(num_doc,score,tf_idf,query_vec)

list_url_id = []
list_url=[]
url_score={}
#得到得分大于零的url
for i in range(num_doc):
    if score[i]>0:
        url_score[i]=score[i]
        list_url_id.append(i)
        list_url.append(self.url_id_map[i])
set(list_url)
set(list_url_id)
set(url_score)
print(url_score)
return url_score

```

上文中VSM计算得分的函数:

```

# cos
def cosine_similarity(x, y):
    num = x.dot(y.T)
    denom = np.linalg.norm(x) * np.linalg.norm(y)
    return num / denom

# 分数计算
def get_score(len, score, tf_idf_x, query):
    for i in range(len):
        # doc_vec = np.array(tf_idf_x[i])
        # print(doc_vec)
        score[i] = cosine_similarity(tf_idf_x[i], query)

```

2.通配查询

首先利用包 `fnmatch` ,实现对输入在词袋中查找匹配的词。

词袋为索引构建时建立 `TfidfVectorizer` 得到 (`tfidf_vectorizer.get_feature_names_out()`)

```

from fnmatch import fnmatchcase as match

def wildcardLookup(st, wbags):
    matches=[string for string in wbags if match( string,st)]
    return matches

```

之后利用在词袋中查找匹配的term，调用函数 `Query.common_query`，即调用站内查询函数。

```

#通配查找
def wildcard_query(self,query,positin_type=0):

    #通配根据输入在词袋中查找匹配的term
    wquery=wildcarding.wildcardLookup(query,self.words_bag)
    print(wquery)

    url_score=Query.common_query(self,query=wquery,positin_type=positin_type,type='
wildcard')
    return url_score

```

3.短语查询

- 建立索引时利用whoosh，领用原数据建立索引。
- 再利用whoosh的匹配查询函数实现精准的查询匹配。
- 根据position_type分别调用

`buildIndex.query_Content`，`buildIndex.query_Title`，`buildIndex.query_Anchor`，`buildIndex.query_Url`。及分别调用以content、title、anchor和url域建立的索引的匹配查询。

```

#短语查询
def parse_query(self,query,position_type):
    q_list=[]
    if position_type==0:
        q_list = buildIndex.query_Content(query)
    if position_type==1:
        q_list = buildIndex.query_Title(query)
    if position_type==2:
        q_list = buildIndex.query_Anchor(query)
    if position_type==3:
        q_list = buildIndex.query_Url(query)

    query_list=[]
    for q in q_list:
        query_list.append(q['url'])

    url_list=[]
    for i in query_list:
        url_list.append(self.url_id_map[i])
    print(query_list)
    print(url_list)
    return query_list

```

以下给出query_Title，其余类似。

```
def query_Title(query):
    new_list = []
    index = open_dir(dir_index_path, indexname='url_title_source') # 读取建立好的索引
    print('query_Title')
    with index.searcher() as searcher:
        parser = QueryParser("title", index.schema)
        myquery = parser.parse(query)
        facet = FieldFacet("url", reverse=True) # 按序排列搜索结果
        results = searcher.search(myquery, limit=None, sortedby=facet) # limit为搜索结果的限制，默认为10
        print(results)
        for result1 in results:
            print(dict(result1))
            new_list.append(dict(result1))
    return new_list
```

4.用户功能

为实现个性化推荐、个性化查询、查询日志功能。增加用户功能。

用来管理当前用户，保存用户、密码、查询历史、兴趣标签。

```
class UserUnit:

    def __init__(self):
        self.user='root'
        self.usr_pd = {}
        self.usr_hobby={}
        self.usr_hs = {}
    def load(self):

        with open(os.path.join(users_dir_path, 'usr_pd.pkl'), 'rb') as doc:
            self.usr_pd =pk1.load( doc)
        with open(os.path.join(users_dir_path, 'usr_hs.pkl'), 'rb') as doc:
            self.usr_hs=pk1.load( doc)
        with open(os.path.join(users_dir_path, 'usr_hobby.pkl'), 'rb') as doc:
            self.usr_hobby=pk1.load( doc)

    def save(self):
        with open(os.path.join(users_dir_path, 'usr_pd.pkl'), 'wb') as doc:
            pk1.dump(self.usr_pd, doc)
        with open(os.path.join(users_dir_path, 'usr_hs.pkl'), 'wb') as doc:
            pk1.dump(self.usr_hs, doc)
        with open(os.path.join(users_dir_path, 'usr_hobby.pkl'), 'wb') as doc:
            pk1.dump(self.usr_hobby, doc)

    def change_usr_pd(self,usr_pd):
        self.usr_pd=usr_pd
    def change_usr_hs(self,usr_hs):
        self.usr_hs=usr_hs
```



```

def change_usr_hobby(self,usr_hobby):
    self.usr_hobby=usr_hobby
def change_usr(self,usr):
    self.user=usr

def get_usr_pd(self):
    return self.usr_pd
def get_usr_hs(self):
    return self.usr_hs
def get_usr_hobby(self):
    return self.usr_hobby
def get_usr(self):
    return self.user

#登入判断
def login_judge(self,username,pd):
    if username in list(self.usr_pd.keys()):
        if pd == self.usr_pd[username]:
            return True
    return False

```

5.查询日志

- 利用用户功能，保存的登录用户的查询历史。
- 提供查询日志网页，现实该用户的查询历史。

```

# 转到查询日志
@app.route('/turn_to_search_log',methods=['POST','GET'])
def turn_to_search_log():
    # print('111')
    search_log=global_user.get_usr_hs()
    search_log=search_log[global_user.get_usr()]
    search_log=reversed(search_log)
    return render_template('search_log.html',page_list=search_log)

```

效果如下：

龙湖

程明明

网络攻防与系统安全

龙湖

龙湖

程明明

网络攻防与系统安全

网络攻防与系统安全

6.网页快照

- 爬虫爬取数据时将爬取网页的html文件保存到了本地。
- 在每一条查询结果下提供网页快照按钮，点击打开本地保存的对应网页的历史html文件。

```
#网页快照跳转
@app.route('/turn_to_web_page_snapshot', methods=['POST', 'GET'])
def turn_to_web_page_snapshot():
    id = request.args.get('id')
    st='source_code/'+str(id)+'.html'
    return render_template(st)
```

效果如下：

- [联系我们](#)
 - [机构设置](#)
 - [院长信箱](#)
- [English](#)

师资队伍

- [教授/研究员](#)
- [副教授/副研究员](#)
- [讲师](#)
- [实验教学队伍](#)
- [博士后](#)
- [兼职教授](#)

姓名	职称	所属部门	研究方向
郭春乐	博士后	计算机科学与技术系	深度学习，图像增强，图像复原
贾岩	博士后	信息安全系	漏洞挖掘，物联网安全，网络攻防与系统安全
吕思艺	博士后	计算机科学与技术系	
宋珂慧	博士后	计算机科学与技术系	
谢学说	博士后	计算机科学与技术系	
徐思涵	博士后	计算机科学与技术系	智能软件安全；软件工程；软件测试
徐夏	博士后	计算机科学与技术系	
张翼	博士后	计算机科学与技术系	

Copyright © 2018 - 2019 南开大学计算机学院. All Rights Reserved 浏览次数: 1000

五、个性化查询

函数输入为：

- qr：为查询返回的字典，其key为url（使用IdMap（），即url对于的数字）。values为前面计算得到的分数。
- hobby：为用户在注册时选择的兴趣标签。
- history：为用户的查询历史。

实现思路：

- 利用查询分别对hobby和history做查询，得到对应的查询结果字典。
- 如果hobby和history中的key也存在于输入的字典中，则对输入的字典中该key对应的分数有加分，即`qr[key]+=url_score_hobby[key]/para_hobby`。
- 实现的效果为对含有兴趣标签和查询历史特征的网页有一定加分。

```
#个性化查询,为不同的用户提供不同的内容排序
def
add_personal_queries(self,qr,hobby=None,history=None,para_hobby=10,para_history=
20):

    str_hobby=''
    for i in hobby:
        str_hobby+=i+' '
    str_history=''
    for i in history:
        str_history+=i+' '

    # print(str_hobby)
    # print(str_history)
    url_score_hobby=Query.common_query(self,str_hobby)
    url_score_history=Query.common_query(self,str_history)
    # print(url_score_hobby)
    # print(url_score_history)

    qr_keys=qr.keys()
    url_score_hobby_keys=url_score_hobby.keys()
    url_score_history_keys=url_score_history.keys()
```

```

for key in url_score_hobby_keys:
    if key in qr_keys:
        qr[key]+=url_score_hobby[key]/para_hobby

for key in url_score_history_keys:
    if key in qr_keys:
        qr[key]+=url_score_history[key]/para_history

```

六、个性化推荐

该部分使用 `synonyms` 包，生成输入的hobby和history（兴趣标签和查询历史）的近义词列表。利用近义词列表进行查询，得到推荐的网页字典。与输入的未经过推荐查询的字典结合，做set。得到个性化推荐后的查询字典。

函数输入为：

- qr：为查询返回的字典其key为url（使用IdMap（），即url对于的数字）。values为前面计算得到的分数。
- hobby：为用户在注册时选择的兴趣标签。
- history：为用户的查询历史。

实现思路：

- 对输入的hobby和history进行预处理。生成近义词列表，得到符合查询的输入。
- 利用查询分别对hobby和history做查询，得到对应的查询结果字典。
- 与输入的未经过推荐查询的字典结合，做set。得到个性化推荐后的查询字典。
- 返回结合后的查询字典(经过个性化推荐，有所扩充)

```

# 个性化推荐
def add_personal_recommendation(self ,qr,hobby=None,history=None):

    print('start personal_recommendation')

    #根据爱好标签
    rec_hobby=[]
    for h in hobby:
        # hh=synonyms.display(h,size=3)
        rec_hobby.extend(synonyms.nearby(h,size=5)[0])
    print('rec hobby',rec_hobby)

    # 根据查询历史
    r_history=list(reversed(history))
    if len(r_history)>5:
        r_history=r_history[0:5]
    rec_history = []
    for hh in r_history:
        h = jieba.lcut(hh)
        for _ in h:
            rec_history.extend(synonyms.nearby(_, size=3)[0])
    print('rec history', rec_history)

    st_hobby=''

```

```

for temp in rec_hobby:
    st_hobby+=temp+' '
st_history=''
for temp in rec_history:
    st_history+=temp+' '
# 推荐内容的查询
url_score_hobby=Query.common_query(self,st_hobby)
url_score_history=Query.common_query(self,st_history)

after_add_personal_recommendation = dict(qr)

for i in url_score_hobby.keys():
    url_score_hobby[i] = url_score_hobby[i] / 10
for i in url_score_history.keys():
    url_score_history[i] = url_score_history[i] / 50

after_add_personal_recommendation.update(url_score_hobby)
after_add_personal_recommendation.update(url_score_history)
set(after_add_personal_recommendation)
print(after_add_personal_recommendation)
print('end personal_recommendation')
return after_add_personal_recommendation

```

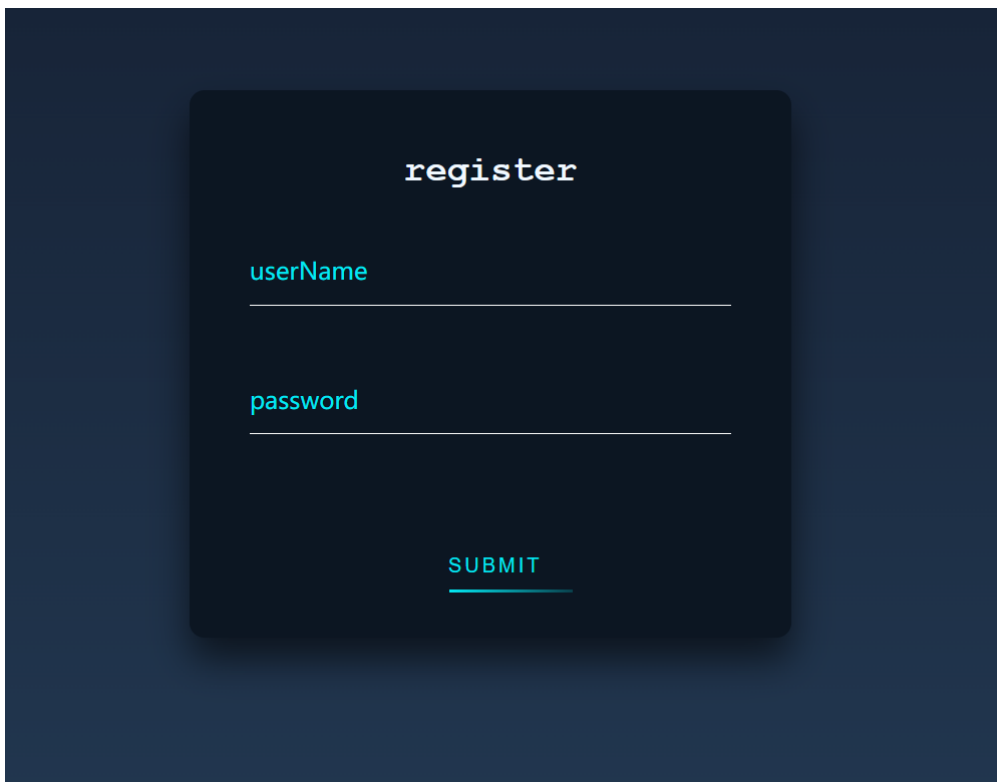
七、Web页面，图形化界面

网页分为以下部分：

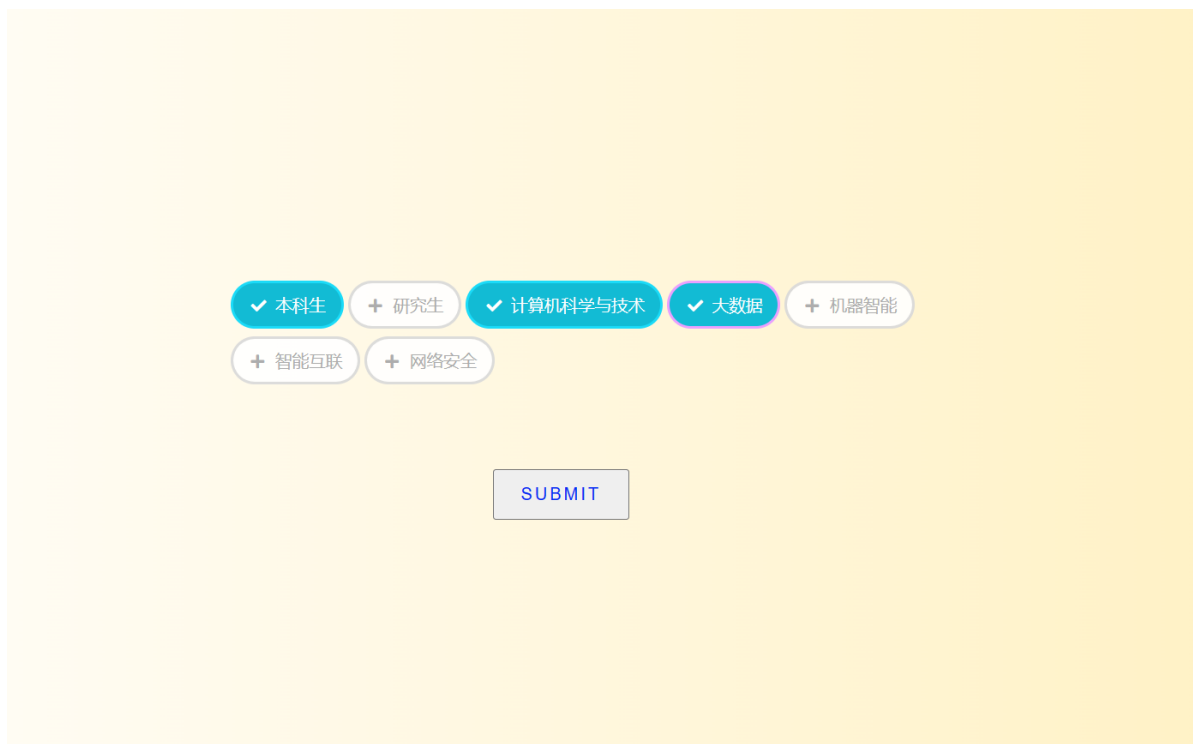
- 开始的是登入页面，可以选择登入或者注册。

The image shows a login form on a dark blue background. The form is a dark gray rounded rectangle with the title 'login' at the top. It contains two input fields: 'userName' and 'password', both with light blue labels. Below the password field is a 'SUBMIT' button with a light blue underline. To the right of the form is a light blue link that says 'CLICK TO REGISTER'.

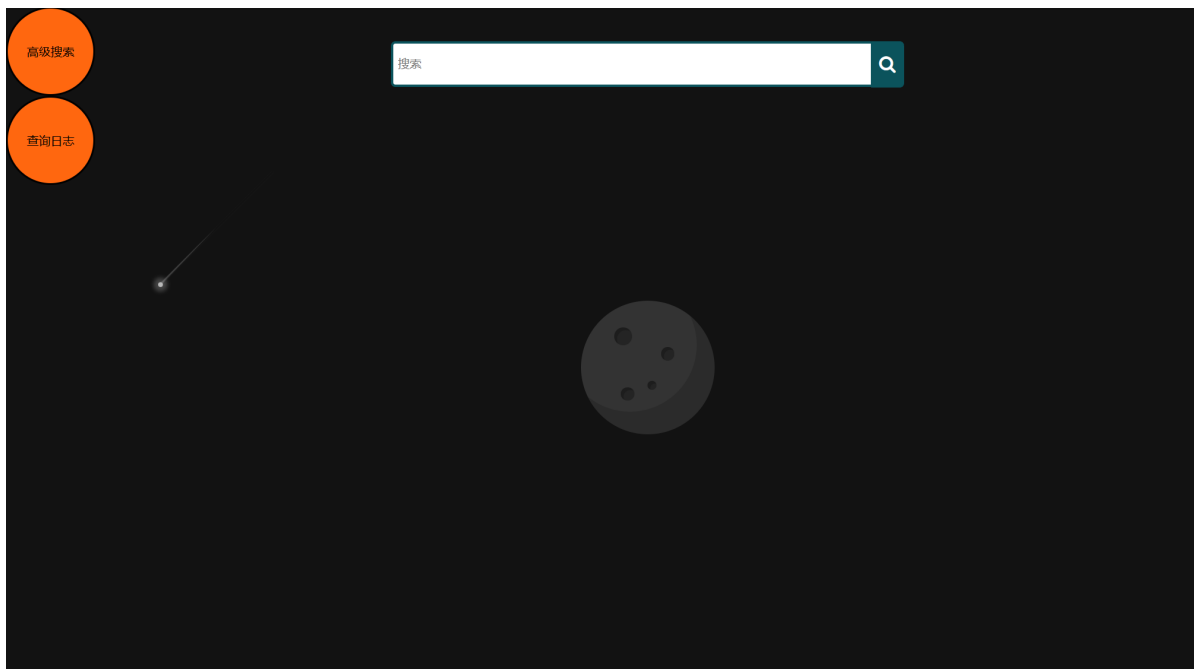
- 注册页面：进行注册，注册后转到兴趣选择页面。



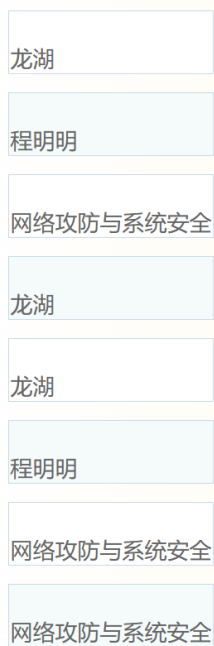
- 兴趣选择页面：选择感兴趣的标签。提交后转回到登入页面。



- 常规搜索页面：进行站内查询，查询范围为网页的全部内容。左上角高级搜索和查询历史。



- 查询历史：可以显示当前登录用户的查询历史。



- 高级搜索：可以进行供站内查询、短语查询、通配查询。可以选择查询得范围。以及设置站内查询地址。



- 查询结果界面：现实查询结果。点击标题跳转，显示部分网页内容。网页快照按钮可以打开本地存储的对应网页。

博士后 博士后首页导航首页学院概况学院概况学院领导学院党委组织结构学科学术分委会学位评定分委会教代会、工会委员会专业技术职务评审分委员会学科建设学科结构计算机科学与技术学科师资队伍教授/研究员副教授/副研究员 网页快照
教授/研究员 教授/研究员首页导航首页学院概况学院概况学院领导学院党委组织结构学科学术分委会学位评定分委会教代会、工会委员会专业技术职务评审分委员会学科建设学科结构计算机科学与技术学科师资队伍教授/研究员副教授/副研 网页快照
博士生导师 博士生导师首页导航首页学院概况学院概况学院领导学院党委组织结构学科学术分委会学位评定分委会教代会、工会委员会专业技术职务评审分委员会学科建设学科结构计算机科学与技术学科师资队伍教授/研究员副教授/副研 网页快照
硕士生导师 硕士生导师首页导航首页学院概况学院概况学院领导学院党委组织结构学科学术分委会学位评定分委会教代会、工会委员会专业技术职务评审分委员会学科建设学科结构计算机科学与技术学科师资队伍教授/研究员副教授/副研 网页快照
讲师 讲师首页导航首页学院概况学院概况学院领导学院党委组织结构学科学术分委会学位评定分委会教代会、工会委员会专业技术职务评审分委员会学科建设学科结构计算机科学与技术学科师资队伍教授/研究员副教授/副研究员 网页快照
副教授/副研究员 副教授/副研究员首页导航首页学院概况学院概况学院领导学院党委组织结构学科学术分委会学位评定分委会教代会、工会委员会专业技术职务评审分委员会学科建设学科结构计算机科学与技术学科师资队伍教授/研究员副教授 网页快照
计算机科学与技术系 计算机科学与技术系首页导航首页学院概况学院概况学院领导学院党委组织结构学科学术分委会学位评定分委会教代会、工会委员会专业技术职务评审分委员会学科建设学科结

- 网页快照：

- [联系我们](#)
 - [机构设置](#)
 - [院长信箱](#)
- [English](#)

师资队伍

- [教授/研究员](#)
- [副教授/副研究员](#)
- [讲师](#)
- [实验教学队伍](#)
- [博士后](#)
- [兼职教授](#)

姓名	职称	所属部门	研究方向
郭春乐	博士后	计算机科学与技术系	深度学习，图像增强，图像复原
贾岩	博士后	信息安全系	漏洞挖掘，物联网安全，网络攻防与系统安全
吕思艺	博士后	计算机科学与技术系	
宋珂慧	博士后	计算机科学与技术系	
谢学说	博士后	计算机科学与技术系	
徐思涵	博士后	计算机科学与技术系	智能软件安全；软件工程；软件测试
徐夏	博士后	计算机科学与技术系	
张翼弓	博士后	计算机科学与技术系	

Copyright © 2018 - 2019 南开大学计算机学院. All Rights Reserved 浏览次数: 1000

代码实现：

使用 flask 框架编写。

```
from flask import Flask, redirect, url_for, request, render_template
from flask import render_template
import queryUnit
import UserUnit

app = Flask(__name__)

global_user=UserUnit.UserUnit()
global_user.load()
query_unit=queryUnit.Query()
# global_user_name='root'
usr_pd = {}
```



```

usr_hobby = {}
usr_hs = {}

#开始
@app.route('/')
def start():
    return render_template('login.html')
    # return render_template('advanced_search.html')
    # return render_template('search.html')

# 跳转至注册
@app.route('/turn_to_register',methods=['POST'])
def turn_to_register():
    return render_template('register.html')

# 注册
@app.route('/register',methods=['POST'])
def register():

    user = request.form['nm']
    password= request.form['pd']

    print((user,password))

    global_user.user=user
    usr_pd[user]=password
    global_user.change_usr_pd(usr_pd)

    print(global_user.usr_pd)
    return render_template('choose.html')

# 选择爱好标签
@app.route('/choose',methods=['POST'])
def choose():
    joy = request.form.getlist('cb')
    print(joy)
    usr_hobby[global_user.user]=joy
    global_user.change_usr_hobby(usr_hobby)
    print(global_user.usr_hobby)
    global_user.save()

    return render_template('login.html')

#登入
@app.route('/login',methods=['POST'])
def login():
    username = request.form['nm']
    password= request.form['pd']

    if global_user.login_judge(username,password):
        global_user.user=username
        return render_template('search.html')

```

```

    #return "login false"
    return render_template('login.html')

#网页快照跳转
@app.route('/turn_to_web_page_snapshot',methods=['POST','GET'])
def turn_to_web_page_snapshot():
    id = request.args.get('id')
    print(id)
    st='source_code/'+str(id)+'.html'
    print(st)
    return render_template(st)

# 常规站内搜索
@app.route('/common_search',methods=['POST'])
def common_search():
    input_sr=request.form['input_search']
    print(input_sr)
    if input_sr!='':
        usr_hs= global_user.get_usr_hs()
        if global_user.user not in usr_hs.keys():
            list=[]
            list.append(input_sr)
            usr_hs[global_user.user]=list
        else:
            usr_hs[global_user.user].append(input_sr)
    global_user.change_usr_hs(usr_hs)
    global_user.save()

    # global_user.user = '1'
    usr_hobby = global_user.get_usr_hobby()
    hobby = usr_hobby[global_user.user]
    print(hobby)
    history = global_user.get_usr_hs()
    history = history[global_user.user]
    print(history)

page=query_unit.query(input_query=input_sr,query_type=3,hobby=hobby,history=history)

# print(page)
print(global_user.usr_hs[global_user.user])

return render_template('result.html', page_list=page)

# 转到查询日志
@app.route('/turn_to_search_log',methods=['POST','GET'])
def turn_to_search_log():
    # print('111')
    search_log=global_user.get_usr_hs()
    search_log=search_log[global_user.get_usr()]
    search_log=reversed(search_log)
    return render_template('search_log.html',page_list=search_log)

```

```

# 转到高级搜索
@app.route('/turn_to_advanced_search')
def turn_to_advanced_search():
    # print('222')
    return render_template('advanced_search.html')

# 高级搜索
@app.route('/advanced_search', methods=['POST' ])
def advanced_search():

    phrase=request.form['q1']
    wildcard = request.form['q2']
    inside_station = request.form['q3']
    # q4 = request.form['q4']
    q5 = request.form.get('q5')
    q6 = request.form['q6']

    query=''
    query_type=0
    if phrase!='' and wildcard=='' and inside_station=='':
        query=phrase
        query_type=1
    elif phrase == '' and wildcard != '' and inside_station == '':
        query = wildcard
        query_type=2
    elif phrase == '' and wildcard == '' and inside_station != '':
        query=inside_station
        query_type=3

    position_type=0
    if q5=='0':
        position_type=0
    if q5=='1':
        position_type=1
    if q5=='2':
        position_type=2
    if q5=='3':
        position_type=3

    print(phrase)
    print(wildcard)
    print(inside_station)
    # print(q4)
    print(q5)
    print(q6)
    print(query)
    print(query_type)

```

```

print(position_type)

# global_user.user='1'
usr_hobby=global_user.get_usr_hobby()
hobby=usr_hobby[global_user.user]
print(hobby)
history=global_user.get_usr_hs()
history=history[global_user.user]
print(history)

page=query_unit.query(input_query=query,query_type=query_type,positin_type=posit
ion_type,hobby=hobby,history=history)
print(page)

return render_template('result.html', page_list=page)

# 高级搜索返回常规搜索
@app.route('/return_to_common_search')
def return_to_common_search():
    return render_template('search.html')

if __name__ == '__main__':
    app.run()

```