

# 目录

<b>1 项目介绍</b>	<b>1</b>
1.1 项目亮点 . . . . .	1
1.2 项目（代码）框架 . . . . .	1
1.3 项目流程图 . . . . .	2
<b>2 爬虫</b>	<b>2</b>
2.1 基本功能 . . . . .	2
2.2 错误处理与断点续爬 . . . . .	5
2.3 多线程爬虫实现 . . . . .	6
<b>3 网页储存结构设计与信息提取</b>	<b>7</b>
3.1 树状存储结构 . . . . .	7
3.2 文本提取与分词 . . . . .	8
<b>4 搭建词频统计与倒排索引</b>	<b>8</b>
4.1 单一网站的词频统计与倒排索引搭建 . . . . .	8
4.2 多个网站的词频统计与倒排索引合并 . . . . .	9
<b>5 基于 tf-idf 的推荐</b>	<b>9</b>
5.1 tf-idf . . . . .	9
5.2 计算 tf-idf . . . . .	10
5.3 基于余弦相似度的推荐 . . . . .	10
5.4 搜索效果评估 . . . . .	11
<b>6 基于字符串匹配的推荐结果重排</b>	<b>12</b>
6.1 评分机制 . . . . .	12
6.2 搜索结果重排 . . . . .	13
<b>7 程序模块封装</b>	<b>13</b>
<b>8 Web UI 设计</b>	<b>14</b>
<b>9 思考与讨论</b>	<b>15</b>

# 1 项目介绍

本项目是暑期编程集训课程《人工智能综合设计》的课程项目，目标是实现基于（python）网络爬虫以及 TF-IDF 算法的小型搜索引擎。

## 1.1 项目亮点

1. 多线程爬虫实现，对比不同线程数爬虫的爬取速度
2. 使用树状储存结构对爬取到的网页进行储存管理，具有很强的可读性与可操作性
3. 两种爬虫策略（BFS、DFS）对比
4. 定制化搜索范围，可自定义多个域名下的搜索
5. 基于字符串匹配的小范围重排机制，提升推荐效果
6. 模块化程序设计，易于扩展与维护
7. Web UI 设计，便于用户快速匹配目标页面

## 1.2 项目（代码）框架

```
.
├── main.py // 主程序入口，模块功能封装，用于接入 Web UI 和评测模块
├── crawler.py // 爬虫模块
├── tokenizer.py // 基于 jieba 的分词模块
├── ii_tc.py // 建立倒排索引与词频统计
├── tf_idf.py // tf-idf 计算与保存
├── query.py // 查询模块
├── build.py // 控制单个域名下的模块进度
├── history.py // 控制搜索的 domain 组合的状态
├── utils.py // 实用函数
├── eval_client.py // 评测模块
└── eval_search_engine.py
├── app.py // 基于 flask 的 Web UI
└── static
    ├── script.js
    └── style.css
└── templates
    └── index.html
```

### 1.3 项目流程图

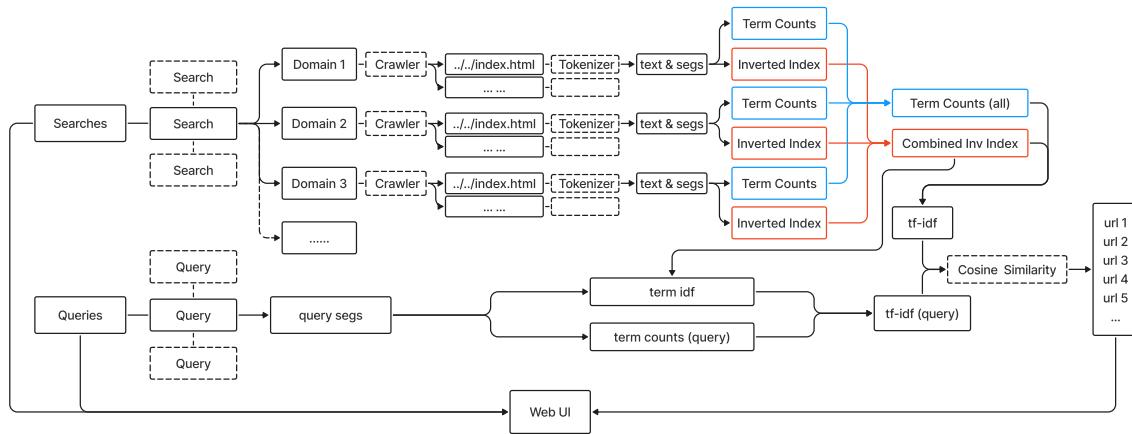


图 1 项目流程图

## 2 爬虫

### 2.1 基本功能

#### 2.1.1 单一页面爬取

使用 `requests` 库中 `requests.get()` 方法获取网页内容(`crawler.soup_maker` 函数), `BeautifulSoup` 进行网页内容解析 (`crawler.links_scraper_sp` 函数)

经过测试, 如果不设置 `headers` 或 `headers` 设置过于简单, 会出现 `403` 错误, 及服务器拒绝访问。参考互联网资料, 将 `headers` 设置为:

```
HEADERS = {
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
        AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.0.0
        Safari/537.36"
} # 为 crawler 设置 headers
```

可以有效解决这一问题

### 2.1.2 使用广度优先搜索（BFS）实现爬虫

算法框架：

---

#### Algorithm 1 爬虫（BFS）

---

**Require:** url: 起始 url (爬虫起点, 默认与 domain 相同)

**Require:** domain: 爬虫目标域名

**Require:** save\_path: 爬取结果保存路径

**Require:** max\_depth: 最大爬取深度

**Require:** fp\_links: 储存已经爬取的链接 (可选, 默认为  $\emptyset$ )

```

1: queue  $\leftarrow$  deque([(url, 0)])                                ▷ 初始化队列
2: if fp_links = None then
3:     fp_links  $\leftarrow$   $\emptyset$ 
4: end if
5: while queue is not empty do
6:     (current_url, current_depth)  $\leftarrow$  queue.popleft()
7:     if current_url  $\in$  fp_links or current_depth > max_depth then
8:         continue
9:     end if
10:    soup  $\leftarrow$  soup_maker(current_url)                      ▷ 爬取当前页面内容 (soup)
11:    save_soup(soup.prettify(), current_url, save_path) ▷ 保存页面到对应网页路径
      下的本地文件夹
12:    fp_links.add(current_url)                               ▷ 对已爬去过的 url 进行标记
13:    if current_depth < max_depth then
14:        found_links  $\leftarrow$  links_scraper_sp(soup, current_url, domain)
15:        new_links  $\leftarrow$  found_links - fp_links            ▷ 更新待爬取的链接
16:        for all link  $\in$  new_links do
17:            queue.append((link, current_depth + 1))
18:        end for
19:    end if
20:    sleep(0.1)
21: end while
22: return fp_links

```

---

### 2.1.3 不同搜索算法的爬虫实现与对比

使用深度优先搜索（DFS）实现爬虫

---

#### Algorithm 2 爬虫（DFS）

---

**Require:** url: 起始 url（爬虫起点，默认与 domain 相同）

**Require:** domain: 爬虫目标域名

**Require:** save\_path: 爬取结果保存路径

**Require:** fp\_links: 储存已经爬取的链接（可选，默认为  $\emptyset$ ）

```

1: function links_scaper_dfs(url, domain, save_path, fp_links)
2: if fp_links = None then
3:   fp_links  $\leftarrow \emptyset$ 
4: end if
5: soup  $\leftarrow$  soup_maker(current_url)                                ▷ 爬取当前页面内容 (soup)
6: if soup = None then
7:   return fp_links
8: end if
9: save_soup(soup.prettify(), current_url, save_path)    ▷ 保存页面到对应网页路径下的本地文件夹
10: fp_links.add(current_url)                                 ▷ 对已爬去过的 url 进行标记
11: found_links  $\leftarrow$  links_scaper_sp(soup, url, domain)
12: new_links  $\leftarrow$  found_links - fp_links                  ▷ 更新待爬取的链接
13: if new_links =  $\emptyset$  then                                ▷ 递归终止条件（没有新链接）
14:   return fp_links
15: end if
16: for all link  $\in$  new_links do                         ▷ 递归地爬取新链接
17:   sleep(0.1)
18:   fp_links  $\leftarrow$  links_scaper_dfs(link, domain, save_path, fp_links)
19: end for
20: return fp_links

```

---

在目标都是爬取所有网页的情况下，DFS 和 BFS 最终能够实现相同的效果。然而，从爬取策略的角度来看，BFS 更符合人类使用网页时的检索方式，且更契合网页的**层次化设计**。BFS 逐层扩展爬取，有助于全面探索同一深度的网页。相比之下，DFS 由于其深度优先的策略，可能会在某一分支上深入探索，忽略其他路径。**在时间和计算资源有限的情况下**，DFS 可能会遗漏一些重要的网页，从而导致覆盖不全的可能。

### 2.1.4 爬虫规则设计

爬虫规则可以分为两类：`exclude` 和 `include`。这些规则分别用于排除不需要的页面以及保留需要爬取的页面。通过设置 `exclude` 规则，爬虫能够避免抓取不相关或不必要的内容，而 `include` 规则则确保爬虫只抓取符合特定需求的目标网页。这样的设计有助于提高爬虫的效率与精确度，减少无效页面的抓取。本项目中，为了像对简化规则设置，在 `links_scraper_sp` 函数中使用了 `include` 的方式，保留了几类常见的网页后缀：

```
def links_scraper_sp(soup: BeautifulSoup, url: str, domain: str) ->
    set:
    ...
    if href:
        href, _ = urldefrag(href)
        if not href.startswith(("http://", "https://", "//")):
            href = urljoin(url, href)
        if not href.startswith(domain):
            continue

        parsed_href = urlparse(href)
        if not (
            parsed_href.path.endswith((".html", ".htm", "/"))
            or "." not in parsed_href.path
        ):
            continue
        ...
    ...
```

模块化的设计使得后续添加和修改爬虫规则变得更加灵活与便捷。只需在函数中规则设置部分修改规则即可实现对爬取行为的调整，无需大规模修改代码结构。此外，这里还可以引入动态爬取规则，进一步提高爬虫能力。

## 2.2 错误处理与断点续爬

在爬虫的实现中，需要考虑到网络请求可能会出现的错误，如 `HTTPError`, `RequestException` 等。为了保证爬虫的稳定性，需要对这些错误进行处理，避免因为网络问题导致爬虫中断。在本项目中，使用 `try-except` 语句捕获异常，并在异常发生时记录错误信息，以便后续分析和处理。

爬虫模块使用 `pickle` 库来保存爬虫运行时的关键数据结构（如 `queue` 和 `fp_links`），实现爬虫的断点续爬。通过将这些数据结构序列化保存至文件，当爬虫中断后，可以从保存的状态恢复继续执行，从而避免重复爬取已经访问的网页。

## 2.3 多线程爬虫实现

使用 concurrent.futures 中的 ThreadPoolExecutor 实现多线程并行爬取

```

def process_link( current_url: str, current_depth: int, domain:
    ↳ str, save_path: str, fp_links: set, max_depth: int, lock:
    ↳ threading.Lock) -> tuple:
    # 设计 fp_links 操作时使用线程锁，避免多线程操作时出现问题
    ...
    with lock:
        if current_url in fp_links or current_depth > max_depth:
            return None, None
    ...
    with lock:
        fp_links.add(current_url)
    ...

def links_scraper_bfs_parallel(url: str, domain: str, save_path:
    ↳ str, max_depth: int = 12, max_workers: int = 6)->None:
    fp_links, queue = load_State(save_path) # 读取上次爬取状态
    if not queue:
        queue = deque([(url, 0)])
    if not fp_links:
        fp_links = set()
    lock = threading.Lock()
    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        futures = []
        while queue or futures:
            while queue:
                current_url, current_depth = queue.popleft()
                future = executor.submit(process_link,
                    ↳ current_url, current_depth, domain,
                    ↳ save_path, fp_links, max_depth, lock)
                futures.append(future)
            for future in as_completed(futures):
                futures.remove(future)
                result = future.result()
                # 更新 queue
        wait(futures)
    
```

对比多线程爬虫爬取同一网站时间与线程数的关系，经过多次实验，在尽量保证同一网络与硬件环境的条件下，得到如下结果：

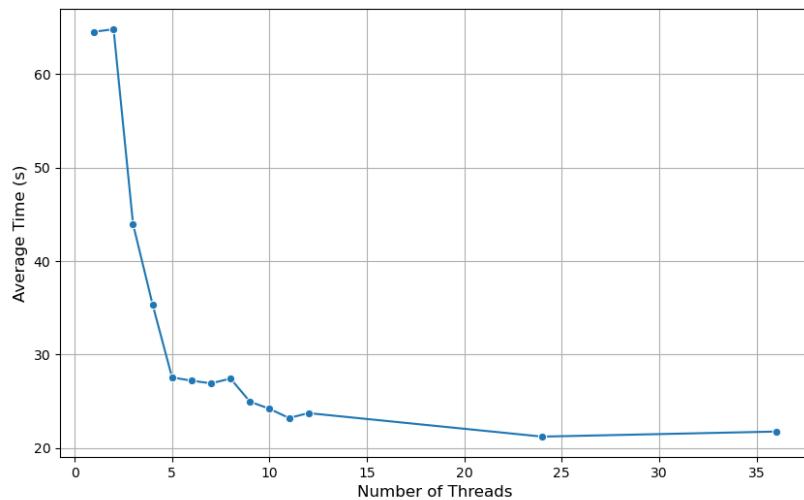


图 2 爬取时间-线程数

结合图2，最终设置多线程爬虫线程数为 6，主要因为在这个线程数下，爬取时间较短，且在之后的线程数增加，爬取时间增加的速度明显变慢

### 3 网页储存结构设计与信息提取

#### 3.1 树状存储结构

使用 url 中的 path 分别储存每个页面的 html 文件。以 <http://ai.ruc.edu.cn> 为例，爬取到的网页文件部分结构大致如下：

```
http_ai.ruc.edu.cn
└── newlist
    ├── lecture
    └── newsdetail
└── research
    ├── gsaiacademic
    └── science
└── student
    ├── doctor
    ├── master
    └── undergraduate
...
...
```

这种储存方式，网页和储存位置一一对应，便于快速查找和管理，同时方便后续爬虫更新

## 3.2 文本提取与分词

使用 `bs4.BeautifulSoup.find_all` 进行 html 标签提取，进而获取网页的文本内容，并用 `jieba` 库对文本内容进行分词

文本提取，根据 html 中不同标签进行提取，如标题 `<h1>`、`<h2>`、`<h3>`、`<h4>`、`<h5>`、`<h6>`，段落 `<p>`，要点 `<li>` 等，对不同的标签，进行相应标记，为后面进行字符串匹配评分作准备

```
def extract_text(file_path: str) -> str:
    with open(file_path, "r", encoding="utf-8") as file:
        soup = BeautifulSoup(file, "html.parser")
        content = []
        if soup.title:
            content.append(f"# {soup.title.string.strip()}") # 对
            # title 进行标记
        for element in soup.find_all(["h1", "h2", "h3", "h4",
                                     "h5", "h6", "p", "li"]):
            ... # 对 h、p、li 进行提取，并且加以标记
```

使用 `jieba.cut_for_search()` 进行分词并且加载停用词表进行过滤

```
filtered_words = [word for word in words if word not in stopwords
                  and word.strip()]
```

## 4 搭建词频统计与倒排索引

### 4.1 单一网站的词频统计与倒排索引搭建

对同一个网站下的每个网页，结合文本与分词结果，对（一个域名下）所有网页进行词频统计，即可得到词频字典 (`term_counts`) 首先明确这里“词频”  $f_{t_i, d_j}$  是指词语  $t_i$  在文档  $d_j$  中出现的次数，从而

$$\text{term\_counts}(d_i) = \{(t_1, f_{t_1, d_i}), (t_2, f_{t_2, d_i}), \dots\}$$

具体实现见 `ii_tc.build_term_counts` 函数

**倒排索引** (inverted index)

$$\text{inverted\_index}(t) = \{(t_1, (d_{t_1,1}, d_{t_1,2}, \dots)), \dots, (t_k, (d_{t_k,1}, d_{t_k,2}, \dots))\}$$

其中  $(d_{t_i,1}, d_{t_i,2}, \dots)$  是所有包含词语  $t_i$  的文档

具体实现见 `ii_tc.build_ii_tc` 函数

## 4.2 多个网站的词频统计与倒排索引合并



图 3 词频、倒排索引的合并

词频以**文档 (url)**为单元，可直接合并；倒排索引以**词语**为单位，合并时需要对相同词语的**文档集合**进行并集操作。由于倒排索引的数据结构通常由字典和集合构成，基于哈希结构的集合并集操作能够保证较高的合并效率。

## 5 基于 tf-idf 的推荐

### 5.1 tf-idf

tf-idf (term frequency-inverse document frequency) 是一种用于信息检索与文本挖掘的常用加权技术 [1] 。

**词频** (term frequency) 指某一个给定的词语在该文件中出现的频率，其有多种定义方式（不同的标准化方式），本项目中，采取对数标准化 (log normalization)，公式如下：

$$\text{tf}(t, d) = \log(1 + f_{t,d}) \quad (1)$$

其中  $f_{t,d}$  为词语  $t$  在文档  $d$  中出现的次数，即前文中的 **term\_counts**。

**逆文档频率** (inverse document frequency) 度量的是一个词能够提供的信息量，公式如下：

$$\text{idf}(t, D) = \log\left(\frac{N}{1 + |\{d : d \in D \text{ and } t \in d\}|}\right) \quad (2)$$

其中  $N$  为文档总数， $|\{d : d \in D \text{ and } t \in d\}|$  表示包含  $t$  所出现的文档数

结合 tf (1) 与 idf (2)，得到 tf-idf：

$$\text{tf-idf}(t, d, D) = \text{tf}(t, d) \cdot \text{idf}(t, D) \quad (3)$$

## 5.2 计算 tf-idf

### 5.2.1 建立网页的 tf-idf 库

可以发现，tf-idf 库与 term\_counts 类似，都是以文档为单位的，因此可以直接在 term\_counts 的基础上，结合（整合后的）inverted\_index 进行计算，即可得到 tf-idf 库

```
def combine_tf_idf(tc_list: list[dict], ii_list: list[dict],
→   tf_idf_save_path: str) -> dict:
    total_documents = sum(len(tc) for tc in tc_list)
    tf_idf = defaultdict(lambda: defaultdict(dict))
    for tc in tc_list:
        for doc, term in tc.items():
            for term2, tc2 in term["tc"].items():
                idf = math.log(total_documents / (1 +
→      len(combined_inverted_index[term2])))
                tf = math.log(1 + tc2) # 对数标准化后的词频统计
                tf_idf[doc]["tf_idf"][term2] = tf * idf
```

### 5.2.2 建立 query 的 tf-idf

结合 inverted\_index 与 query，即可得到 query 的 tf-idf，

```
def compute_query_tf_idf(inverted_index: dict, query_segs: str,
→   query_tc: dict, total_documents: int) -> dict:
    query_idf = {}
    query_segs = query_segs.split("/") # 假设词是以 "/" 分隔的
    for term in query_segs:
        if term in inverted_index:
            query_idf[term] = math.log(total_documents / (1 +
→      len(inverted_index[term])))
        else:
            query_idf[term] = 0
    return {term: (1 + math.log(query_tc.get(term, 0))) * idf
            for term, idf in query_idf.items()}
```

至此，我们得到了网页的 tf-idf 库与 query 的 tf-idf

## 5.3 基于余弦相似度的推荐

在本项目中，查询的 tf-idf 与每个网页的 tf-idf 本质上是两个向量， $\omega_q$  和  $\omega_p$ ，因此可以通过计算余弦相似度 (cosine\_similarity)，排序得到推荐结果。

$$\text{cosine\_similarity}(\omega_q, \omega_p) = \frac{\omega_q \cdot \omega_p}{\|\omega_q\| \|\omega_p\|}$$

由于单一文档中的词汇在整个文档集合中的占比通常较低，导致这两个向量往往呈现出显著的稀疏性（sparse）。如果直接使用原始向量进行存储和计算，将会造成大量的存储和计算资源浪费。为此，本项目延续了之前的字典设计方案，用以词作为索引，tf-idf 值作为键值的字典来存储这两个向量。在相似度计算时，只需要考虑两个字典中交集部分（非零部分）的余弦相似度，从而有效减少储存与计算开销。

```
def cosine_similarity(tf_idf: dict, query_tf_idf: dict) -> float:
    all_terms = set(tf_idf.keys()).union(set(query_tf_idf.keys()))
    tf_idf = {term: tf_idf.get(term, 0) for term in all_terms}
    query_tf_idf = {term: query_tf_idf.get(term, 0) for term in
                    all_terms}
    dot_product = sum(tf_idf[key] * query_tf_idf[key] for key in
                      tf_idf)
    magnitude1 = math.sqrt(sum(value**2 for value in
                               tf_idf.values()))
    magnitude2 = math.sqrt(sum(value**2 for value in
                               query_tf_idf.values()))
    return dot_product / (magnitude1 * magnitude2)
```

## 5.4 搜索效果评估

### 5.4.1 搜索结果评价指标

Mean Reciprocal Rank (MRR) 是一种用于评估搜索引擎效果的指标 [2]，其定义如下：

$$\text{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i} \quad (4)$$

其中  $Q$  为查询集合， $\text{rank}_i$  为第  $i$  个查询的第一个正确结果的排名。MRR 越高，说明搜索引擎的效果越好。

### 5.4.2 tf-idf 搜索效果评估

使用课程提供的评估模块，对上述基于 tf-idf 的搜索效果进行评估，得到 MRR 值为 0.54。根据 MRR 的定义，这表明搜索引擎推荐的第一个结果在正确答案中的平均排名为 1.85。这意味着当前搜索引擎能够在前几条推荐结果中较为准确地包含正确答案，但在排名靠前的精确性方面仍有提升空间。

## 6 基于字符串匹配的推荐结果重排

基于 5.4.2 中的分析，为了解决小范围搜索结果排序问题，以进一步提高检索效果，本项目提出了一种基于字符串匹配的局部评分机制，用于对通过 tf-idf 获取的前 n 个搜索结果进行重排，以进一步提升搜索引擎的推荐效果。该方法旨在将查询 (query) 及其分词与网页文本进行匹配，并设计相应的评分机制来优化搜索结果的排序。

### 6.1 评分机制

在设计字符串匹配的评分机制时，主要考虑以下因素：

1. **分词**：为确保分词的准确性与全面性，使用 `jieba.cut(text, cut_all=False)` 分词
2. **出现次数**：考虑字符串在网页文本中出现的次数，频率越高，得分越高。
3. **字符串长度**：认为较长的字符串在文本中出现时，其重要性高。
4. **出现位置**：检查字符串是否出现在标题、要点等关键部分，出现位置越关键，得分越高

在3.2中文本提取过程中提出对不同标签进行标记，在这里可以用来判断字符串是否出现在特殊标签 (tags) 中，如标题、要点等，对得分给予不同的权重

$$S(\text{seg}) = \begin{cases} \text{count} \cdot \text{len}^2, & \text{if } \text{seg} \neq \text{query} \text{ and } \text{seg} \in \text{text}, \\ \text{count} \cdot \text{len}^3, & \text{if } \text{seg} \neq \text{query} \text{ and } \text{seg} \in \text{tags}, \\ \text{count} \cdot \text{len}^4, & \text{if } \text{seg} = \text{query} \text{ and } \text{seg} \in \text{text}, \\ \text{count} \cdot \text{len}^5, & \text{if } \text{seg} = \text{query} \text{ and } \text{seg} \in \text{tags}. \end{cases}$$

$$\text{Score} = \sum_{\text{seg} \in \text{segs}} S(\text{seg})$$

其中：**query** 表示查询关键词，**segs** 表示 query 的所有分词（包括 query 本身），**seg** 表示 query 中的单个分词，**len** 表示 seg 的字符串长度，**count** 表示 seg 在文本中出现的次数，**S(seg)** 表示 seg 的得分，**Score** 表示 query 的总得分。

从图 4 可以看出，在此打分机制下，当完整的 query 出现，或 seg 出现在标题 (title) 中时，对最终得分具有决定性影响。

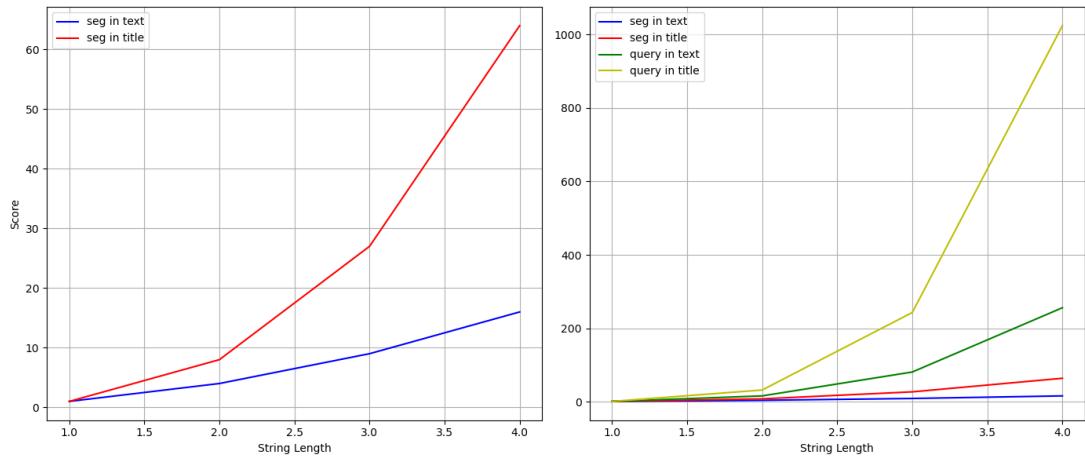


图 4 打分机制

## 6.2 搜索结果重排

引入上述评分机制后，5.4.2中的 MRR 得分提升至 **0.786**。观察输出结果后，发现很多 query 的推荐首选是带有目录性质的界面——因为这些界面会含有较多标题标签，导致评分偏高，排到了真正有效页面前。在重排过程中引入相应过滤机制后，MRR 得分得到进一步提升，达到 **0.883**。具体实现见 `query.query_booster` 函数。

## 7 程序模块封装

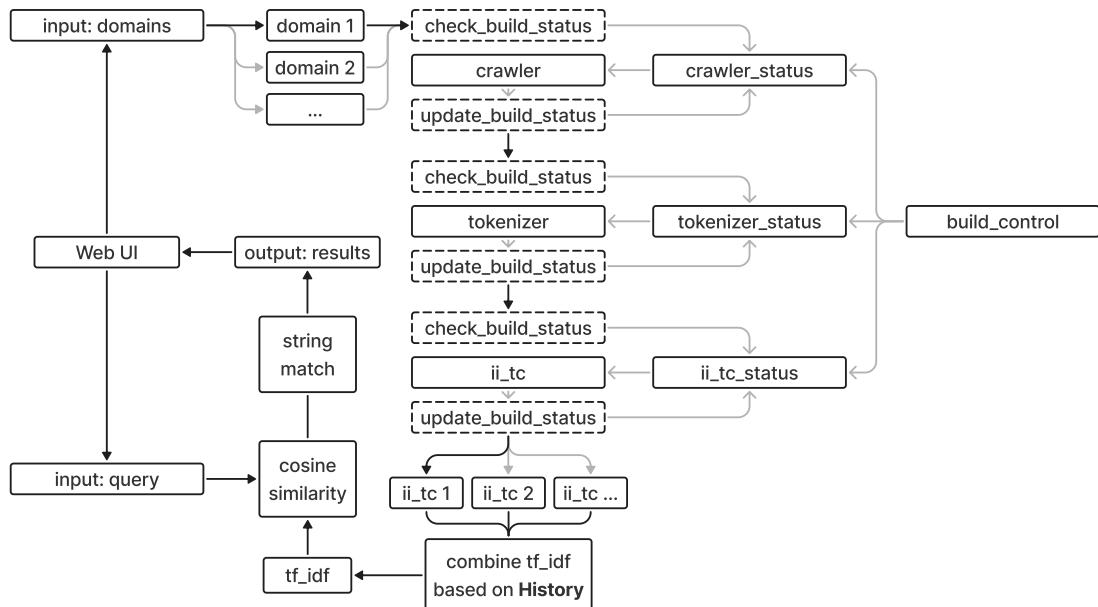


图 5 模块管理

图 5 展示了 1.2 中程序的模块管理流程。将搜索系统对单个网站爬取、分词、倒排索引的过程划分成了三个模块：crawler, tokenizer, ii\_tc(inverted index & term counts)，在设计上，各模块采用独立管理的方式，包括检查（当前模块的状态是否标记为完成）、状态更新（将当前模块的状态标记为完成），在 build\_control 中，可以对各个模块的状态进行调整（从而控制是否在一次搜索中进行爬取、分词、倒排索引的操作），这样设计有利于模块的独立管理，方便后续的扩展与维护。

图 5 中 History 用于判断当前搜索的域名组合是否出现过，具体而言，对于一次搜索，可分为两种情况：

1. 待搜索域名组合之前未搜索过，即第一次搜索，需计算 tf-idf
2. 待搜索域名组合之前搜索过，可直接调用之前的 tf-idf 进行搜索

这里采用“空间换时间”的策略，避免重复计算 tf-idf，提高检索速度

## 8 Web UI 设计

本项目使用 Flask 框架构建 Web UI，并通过调用本地 Python API 实现后端功能交互（app.py）。在页面设计上，使用“卡片”的形式呈现搜索结果，每个卡片包含了网页标题与网页文本内容，用户可直接在卡片中快速预览网页信息；query 中的关键词会以高亮的形式标记。网页同时适用于 PC 端和移动端，具有较好的响应式设计。图 6 分别展示了 PC 与移动端浏览器中 Web UI 的效果。



图 6 Web UI

## 9 思考与讨论

1. **tf-idf**: 本搜索系统主要使用了 tf-idf 算法。然而，tf-idf 算法仍存在一些问题，如对于长文档的处理效果不佳、没有考虑词语的位置信息等。在后续工作中可以考虑使用 BM25 算法替代 tf-idf 算法，以提高搜索效果。
2. **爬虫规则**: 本项目实现的搜索系统在《综合设计》测试集上的 MRR 得分为 0.81，与 6.2 中的结果相比，有所下降。这可能是由于 2.1.4 中爬虫规则过于简单，导致爬取的网页不全，从而影响了搜索结果的准确性。在后续工作中，可以考虑优化爬虫规则、支持动态爬取，提高爬取的网页覆盖率。

## 参考文献

- [1] Wikipedia. tf-idf. <https://en.wikipedia.org/wiki/Tf-idf>. Accessed: 05-September-2024.
- [2] Wikipedia. Mean reciprocal rank. [https://en.wikipedia.org/wiki/Mean\\_reciprocal\\_rank](https://en.wikipedia.org/wiki/Mean_reciprocal_rank). Accessed: 05-September-2024.