从零实现一个简单的全文搜索引擎

杨子凡

May 20, 2025

在信息爆炸的互联网时代，全文搜索引擎已成为处理海量文本数据的核心工具。从 Google 的网页搜索到 Elasticsearch 的企业级检索，其底层都建立在经典的倒排索引和相关性排序机制之上。本文将通过 Python 实现一个支持中文检索的简易搜索引擎，帮助开发者理解其核心原理与技术细节。

1 技术原理

全文搜索引擎的核心在于倒排索引这一数据结构。与传统书籍目录的「页码→内容」映射不同，倒排索引建立的是「关键词→文档集合」的反向映射。例如对于文档集合：

```
文档 1："搜索引擎原理与实践"
文档 2："Python 实现搜索引擎"
```

构建的倒排索引将呈现为：

```
"搜索引擎" → {文档 1:1，文档 2:1}
"Python" → {文档 2:1}
```

相关性排序通常采用 TF-IDF 算法，其公式为：

$$\text{TF-IDF} = \text{TF}(t,d) \times \text{IDF}(t)$$

其中 $\text{TF}(t,d)$ 表示词项 t 在文档 d 中的出现频率，$\text{IDF}(t) = \log(\frac{N}{n_t})$ 表示逆文档频率（N 为总文档数，n_t 为包含词项 t 的文档数）。该算法同时考虑词项的局部重要性和全局区分度。

2 环境准备

我们选择 Python 作为开发语言，因其丰富的文本处理库和简洁的语法特性。中文分词采用 `jieba` 库，其具备 98% 以上的分词准确率和自定义词典支持。通过以下命令安装依赖：

```
pip install jieba
```

3 核心实现步骤

3.1 文档预处理与倒排索引构建

搜索引擎首先需要将原始文本转化为结构化的索引数据。以下代码实现文档添加和索引构建：

```
from collections import defaultdict
import jieba

class SimpleSearchEngine:
    def __init__(self):
        self.inverted_index = defaultdict(dict) # 倒排索引结构
        self.documents = [] # 文档原始内容存储
        self.stop_words = set(["的", "是", "在"]) # 示例停用词表

    def add_document(self, doc_id, text):
        # 中文分词与清洗
        words = jieba.lcut(text)
        words = [word.lower() for word in words
                 if word not in self.stop_words and len(word) > 1]

        # 更新倒排索引
        for word in words:
            if doc_id not in self.inverted_index[word]:
                self.inverted_index[word][doc_id] = 0
            self.inverted_index[word][doc_id] += 1

        self.documents.append({"id": doc_id, "content": text})
```

代码解读：

1. defaultdict(dict) 创建嵌套字典，外层键为词项，内层键为文档 ID
2. jieba.lcut 执行中文分词，lower() 统一为小写形式
3. 停用词过滤移除「的」等无意义词汇，提升索引质量

3.2 搜索逻辑实现

当用户输入查询时，系统需要完成分词、索引查找和结果合并：

```
def search(self, query, page=1, per_page=10):
    query_terms = jieba.lcut(query)
    matched_docs = set()
```

```
4
    # 收集所有包含查询词的文档
6   for term in query_terms:
        if term in self.inverted_index:
8           matched_docs.update(self.inverted_index[term].keys())

10  # 计算相关性排序
    ranked = self.rank_documents(query_terms)
12
    # 分页处理
14  start = (page - 1) * per_page
    return ranked[start:start + per_page]
```

该实现采用 OR 逻辑合并结果，即返回包含任意查询词的文档。实际工业级系统通常支持更复杂的布尔运算。

3.3 相关性排序优化

基于 TF-IDF 的排序算法实现如下：

```
1  import math

3  def rank_documents(self, query_terms):
       scores = defaultdict(float)
5      total_docs = len(self.documents)

7      for term in query_terms:
           if term not in self.inverted_index:
9              continue

11         # 计算 IDF 值
           doc_count = len(self.inverted_index[term])
13         idf = math.log(total_docs / (doc_count + 1))

15         # 累加 TF-IDF 分数
           for doc_id, tf in self.inverted_index[term].items():
17             scores[doc_id] += tf * idf

19     # 按分数降序排列
       return sorted(scores.items(), key=lambda x: x[1], reverse=True)
```

代码关键点：

1. `math.log` 计算自然对数，避免零除错误加入 +1 平滑
2. 分数累加策略使包含多个查询词的文档获得更高排名
3. 排序时间复杂度为 $O(n \log n)$，适用于中小规模数据集

3.4 分页与结果展示

分页功能通过列表切片实现，以下代码演示结果格式化输出：

```
def format_results(self, ranked_docs):
    results = []
    for doc_id, score in ranked_docs:
        content = self.documents[doc_id]["content"]
        # 截取摘要（前 100 字符）
        snippet = content[:100] + "..." if len(content) > 100 else content
        results.append({
            "id": doc_id,
            "score": round(score, 2),
            "snippet": snippet
        })
    return results
```

4 优化与扩展方向

在基础版本之上，可通过以下方式提升系统性能与功能：

1. 前缀匹配优化：引入 Trie 树实现自动补全，将时间复杂度从 $O(n)$ 降至 $O(k)$（k 为查询词长度）
2. 缓存机制：使用 LRU 缓存存储高频查询结果，降低重复计算开销
3. 短语搜索：通过 Bigram 索引记录词语位置信息，支持精确短语匹配
4. 拼写纠错：基于编辑距离（Levenshtein Distance）实现查询词建议

例如拼写纠错的核心逻辑可表示为：

$$\text{编辑距离}(s,t) = \min \begin{cases} \text{删除操作} & \text{插入操作} \\ \text{替换操作} & \text{空字符串长度} \end{cases}$$

本文实现的搜索引擎虽然省略了分布式、实时更新等复杂特性，但完整呈现了倒排索引构建、TF-IDF 排序等核心机制。读者可通过扩展停用词表、引入 BM25 算法、增加持久化存储等方式继续完善系统。深入学习建议参考《信息检索导论》和 Lucene 源码，探索 PageRank 等更复杂的排序模型。