

# 面向英文数据的编辑与信息检索系统

学号 19071110  
姓名 孙天天  
指导教师 杜永萍

2022 年 1 月 11 日

# 目录

<b>1 需求分析</b>	<b>3</b>
1.1 题目需求描述 . . . . .	3
1.1.1 文件管理功能 . . . . .	3
1.1.2 文本编辑功能 . . . . .	3
1.1.3 编码译码功能 . . . . .	3
1.1.4 词频统计功能 . . . . .	3
1.1.5 高级检索功能 . . . . .	3
1.2 选用的技术栈 . . . . .	3
1.2.1 从课设的角度 . . . . .	3
1.2.2 从产品的角度 . . . . .	4
1.2.3 使用的技术栈 . . . . .	4
1.3 原型设计 . . . . .	4
<b>2 数据结构设计</b>	<b>8</b>
2.1 所定义的数据结构 . . . . .	8
2.1.1 文件树 . . . . .	8
2.1.2 哈希表、集合、映射表 . . . . .	9
2.1.3 堆 . . . . .	10
2.1.4 哈夫曼树 . . . . .	10
2.1.5 base64 编解码器 . . . . .	11
2.1.6 替换器 . . . . .	12
2.1.7 倒排索引 . . . . .	12
2.2 所实现的算法 . . . . .	13
2.2.1 表达式转换与求值 . . . . .	13
2.2.2 高级检索 . . . . .	13
2.3 系统整体结构以及各模块的功能描述 . . . . .	14
2.3.1 整体组件 App . . . . .	14
2.3.2 开始界面 StartPanel . . . . .	14
2.3.3 空白界面 DummyEditorBar . . . . .	15
2.3.4 编辑器界面 Editor . . . . .	15
2.3.5 目录预览界面 DivViewer . . . . .	17
2.3.6 文件侧栏 FileButtons 和 FileTree . . . . .	17
2.3.7 编码译码侧栏 EncodingPanel . . . . .	19
2.3.8 查找替换侧栏 FindAndReplace . . . . .	21
2.3.9 高级检索侧栏 AdvancedSearch . . . . .	22
2.3.10 词频侧栏 Frequency . . . . .	23
<b>3 详细设计</b>	<b>24</b>
3.1 KMP 算法 . . . . .	24
3.1.1 前缀函数 . . . . .	24
3.1.2 使用前缀函数匹配字符串 . . . . .	25
3.2 闭散列 . . . . .	25

3.2.1	闭散列框架 . . . . .	25
3.2.2	散列函数 h() . . . . .	26
3.2.3	探查函数 p() . . . . .	27
3.3	二级倒排索引 . . . . .	27
3.3.1	整体设计 . . . . .	27
3.3.2	数据结构设计 . . . . .	28
3.3.3	算法设计 . . . . .	28
3.4	相关性评价 . . . . .	29
3.5	流程图 . . . . .	29
3.5.1	整体流程图 . . . . .	29
3.5.2	各部分流程图 . . . . .	29
<b>4</b>	<b>测试</b>	<b>34</b>
4.1	功能性测试 . . . . .	34
4.1.1	普通测试用例 . . . . .	34
4.1.2	导致程序报错的用例 . . . . .	35
4.1.3	边界数据测试 . . . . .	36
4.2	非功能性测试 . . . . .	37
<b>5</b>	<b>总结与提高</b>	<b>38</b>
5.1	亲手实现基础数据结构的经验 . . . . .	38
5.2	分层次、分模块地构建一个多功能程序 . . . . .	38
5.3	鸣谢 . . . . .	38

# 1 需求分析

## 1.1 题目需求描述

题目要求文档中已经对需求进行了全面清晰的描述，这里结合我具体实现拓展要求的情况，对需求进行明确。

### 1.1.1 文件管理功能

基本要求中：对文件的创建、打开、保存功能。

我增加了：递归打开指定目录下所有文件、文件目录树状显示、显示未保存文件的功能。

### 1.1.2 文本编辑功能

基本需求中：文件内容编辑、单文件查找和替换的功能。

我增加了：在文本中自动选中搜索或替换结果、类似迭代器逐个替换、批量替换全部的功能。

### 1.1.3 编码译码功能

基本需求中：给定的文章片段按字频哈夫曼编码和译码，并图形化界面演示。

扩展要求中：对编码与译码的过程可以自行设计其他算法。对此我完成了哈夫曼编码译码和 base64 编码译码。

此外，我增加了：可视化编码中生成的哈夫曼树。

### 1.1.4 词频统计功能

基本要求中：对多篇文档统计词汇出现频率，排序并显示前 30。

我增加了：每个词汇按文档分类，显示文档中词汇出现的每个位置，点击跳转到词汇出现的位置，自定义显示数量。

### 1.1.5 高级检索功能

基本要求中：对多篇文档检索关键词，显示检索结果。

扩展要求中：支持检索表达式，对检索结果按相关性排序（这点我使用 TFIDF 为权重实现）。

我增加了：根据同义词数据库联系同义词一起检索，点击跳转到词汇出现位置，自定义显示数量。

## 1.2 选用的技术栈

### 1.2.1 从课设的角度

本课设的重点在于练习数据结构和算法的设计、开发能力，这点对于大部分编程语言均可实现。而其图形化界面的展示功能不是课设的重点，为此应该选择可以高效快速构建图形界面的技术栈。

这促使我想到了 web 前端技术栈，而考虑到前端开发通常可以使用组件库提高开发效率，我采用了前端框架 Vue 和组件库 Ant Design of Vue。

### 1.2.2 从产品的角度

若将本课设看作一个小规模的产品，分析可能真正使用它的需求。我想对于这样轻量级的工具，快速启动和随开随用十分关键。

选用 web 前端技术栈进行开发，可以将其部署在静态 web 服务器上，用户打开网页即可使用，同时也不会占有过多服务器资源。

### 1.2.3 使用的技术栈

本课设使用 web 前端技术栈开发，具体如下：

- 用户界面框架：Vue
- 网页结构：HTML
- 样式表：CSS、LESS
- 组件库：Ant Design of Vue
- 页面控制与动态反馈：JavaScript
- 数据结构与算法：JavaScript

## 1.3 原型设计

为了更准确地确定需求，我在开发之前进行了原型设计，绘制出了大致的图形界面效果，并说明其中的功能、用到的算法和数据结构。完成后我有老师就原型进行了讨论交流，确定了开发需求。

后来，在进行开发实践后，又对其中与实际不符的地方进行了简单修改。因此以下展示的原型功能与最终版本基本一致，但界面显示效果略有不同。

原型左侧为图形化界面的展示，右侧为对应的功能、实现的需求和用到的算法。其中黑色字为必须完成内容，深蓝色为拓展内容。



## 整体设计

### 使用技术栈:

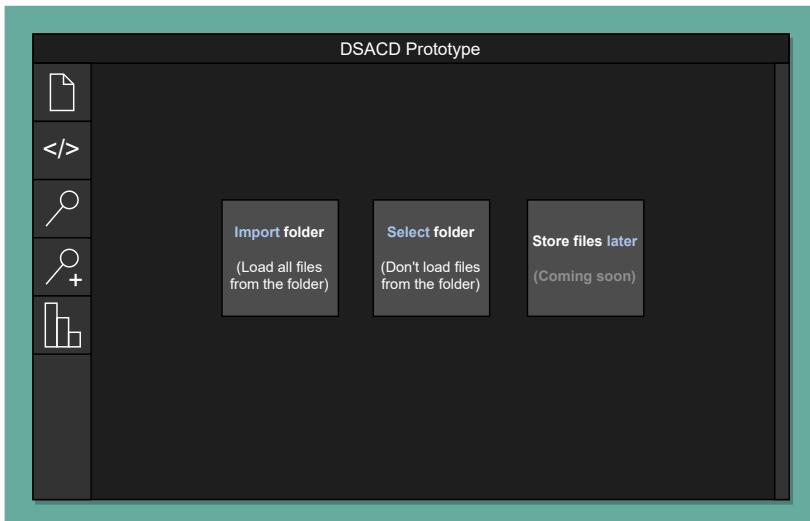
- \* Vue+AntDesign (HTML+CSS+JS)
- \* 除了组件库不使用其他库
- \* 使用同义词数据库 (json文件存储) 联想同义词

### 自定义扩展功能:

- \* 树状文件系统
- \* 高亮、自动选中检索/替换内容，一键全部替换

### 达成题目要求:

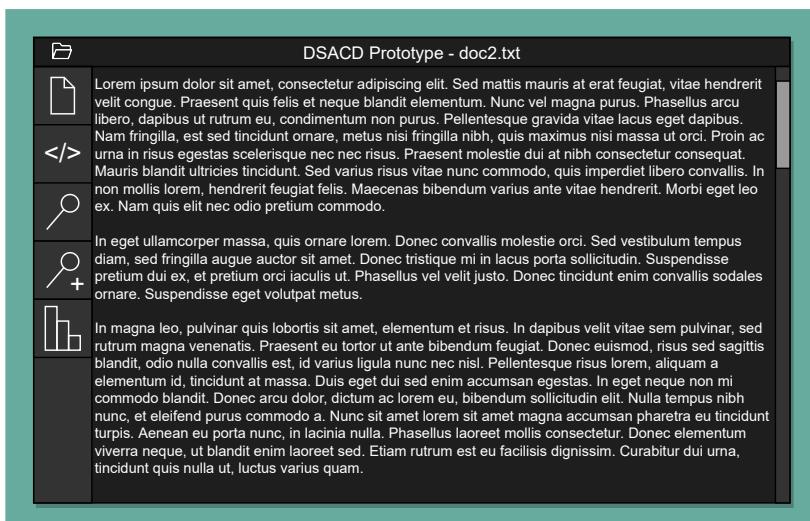
- \* 扩展要求(1) “界面设计的优化。”
- \* 扩展要求(5) “可以自行根据本题目程序的实际应用情况，扩展功能。”



## Start - 开始界面

### 选择一种存储文件方式:

- \* 载入目录：将目录所有文件递归载入
- \* 选择目录：选择存放文件的目录但不载入已有文件



## Editor - 文本编辑器

### 文本编辑器:

- \* 实现基础的文本增添、删除功能
- \* 使用textarea实现

### 左上角文件夹按钮:

- \* 回到开始页面，若有文件未保存会发出提醒

### 达成题目要求:

- \* 基础要求(1) “设计图形界面，可以实现英文文章的编辑和检索功能”



## File - 文件侧栏

### 侧栏树状显示目录与文件:

- \* 展开/收起文件夹
- \* 选择当前编辑的文件
- \* 提示尚未保存的文件  
(目前显示的最底层元素显示实心圆, 其父元素显示空心圆)

### 下方四个按钮:

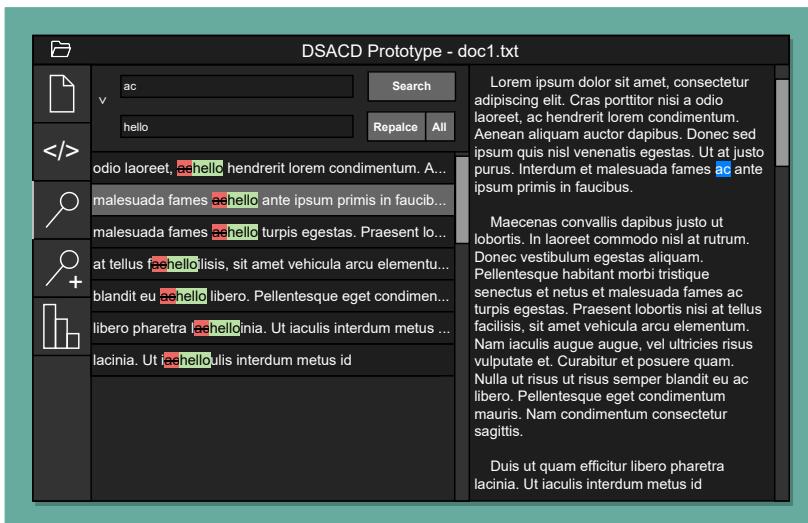
- \* 创建文件: 在当前选中目录/选中文件同目录下创建新文件
- \* 打开文件: 同样创建文件, 但内容初始化为选中的文件
- \* 保存当前文件
- \* 保存全部文件

### 达成题目需求:

- \* 基础要求(2)/1 "创建新文件; 打开文件; 保存文件。"

### 数据结构: 树

- \* 存储形式: 对象作为结点+指针
- \* 载入文件夹: 从根递归地创建树
- \* 选中/修改/保存某个文件: 操作指定结点
- \* 创建文件/打开文件: 为树创建新节点
- \* 显示未保存文件: 找某节点所有父元素
- \* 树的可视化: 由组件库实现



## Search/Replace - 检索替换侧栏

### 检索/替换侧栏:

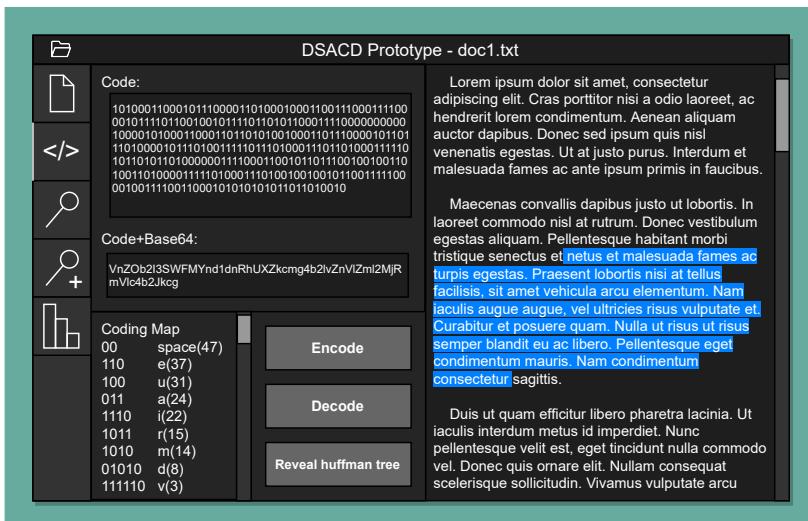
- \* 展开/收起替换框=检索/替换切换
- \* 点按钮检索/替换 (单个或所有) 当前文件中文本
- \* 列出所有检索结果, 高亮检索效果, 预览替换效果
- \* 点击某个结果, 右侧选中对应文本

### 达成题目需求:

- \* 基础要求(2)/2 "查找: 输入单词在当前打开的文档中进行查找, 并将结果显示在界面中。"
- \* 基础要求(2)/3 "替换: 将文章中给定的单词替换为另外一个单词, 再保存等。"

### 算法: KMP

- \* 每次搜索时对当前文本进行KMP匹配运算
- \* 列出所有匹配结果 (向前寻找3个单词或指定最大字符数)
- \* 列出结果中关键部分使用特殊格式
- \* 默认选中第一个检索结果, 点击某个时跳转至那一个



## Encode/Decode - 编码解码侧栏

### 编码解码侧栏:

- \* Code栏展示/输入编码结果
- \* Code+Base64栏展示/输入编码的Base64结果
- \* Coding Map栏展示编码表
- \* Encode: 将右侧选中内容哈夫曼编码生成到左侧
- \* Decode: 将左侧输入编码生成到右侧  
(替换选中内容或增加到光标位置)

### 达成题目需求:

- \* 基础要求(3) "对于给定的文章片段 (30<单词数量<100), 统计该片段中每个字符出现的次数, 然后以它们作为权值, 对每一个字符进行编码, 编码完成后将对编码进行译码。在图形界面中演示该过程。"
- \* 拓展需求(2) "对于编码与译码过程, 可以自行设计其他算法。" --Base64 (不太确定十分符合要求)

### 算法: 桶排序+哈夫曼编码

- \* 统计词频使用桶排序
- \* 编码时用词频构建哈夫曼树
- \* 编码表为哈夫曼树前序遍历
- \* 译码时用编码表构建哈夫曼树

The screenshot shows a search interface for multiple files. The search bar at the top contains the query "story AND friend NOT animal". Below the search bar, there are buttons for "Search", "Show -1 files, with 4 words each file.", and "Use 2 synonyms (with IDF)". The results pane displays the following text:

This is it. This is the one. This is the worst movie ever made. Ever. It beats everything. I have never seen worse. Retire the trophy and give it to these people....there's just no comparison.<br /><br />Even three days after watching this (for some reason I still don't know why) I cannot believe how insanely horrific this movie is/was. Its so bad. So far from anything that could be considered a movie, a story or anything that should have ever been created and brought into our existence.<br /><br />This made me question whether or not humans are truly put on this earth to do good. It made me feel disgusted with ourselves and our progress as a species in this universe. This type of movie sincerely hurts us as a society. We should be ashamed. I really cannot emphasize that our global responsibility as people living here and creating art, is that we need to prevent the creation of these gross distortions of our reality for our own good. It's an embarrassment. I don't know how on earth any of these actors, writers, or the director of this film sleeps at night knowing that they had a role in making "Loaded". I don't know what type of disgusting monsters enjoy

## Search+ - 多文件检索侧栏

### 多文件检索侧栏:

- \* 按下按钮对多文件检索，并将结果列在下方
- \* 每条结果包括文件路径、检索结果，高亮所有匹配结果
- \* 选中某条结果自动打开对应文件，光标选中第一个词
- \* 支持逻辑表达式
- \* 支持相近词检索

### 达成题目要求:

- \* 基础要求(5) “对于给定的多篇文章构成的文档集中，建立倒排索引，实现按关键词的检索（包括单个关键词；或者2个及以上关键词的联合检索，要求检索结果中同时出现所有的关键词），并在界面中显示检索的结果（如：关键词出现的文档编号以及所在的句子片段，可以将关键词高亮显示）”
- \* 拓展要求(3) “高级检索，采用逻辑表达式来表示检索需求，例如：(关键词A) AND (关键词B) OR (关键词C) AND (关键词D)”
- \* 拓展要求(4) “优化检索，对于检索结果的相关性排序，例如：包含关键词的数量等信息为依据。” -- TFIDF算法+相关词联想

### 算法：二级倒排索引

- \* 一级索引：统计每个文件中各个单词出现的位置
- \* 二级索引：记录每个单词在各个文件中出现的词频
- \* 逻辑表达式检索
  - \* 每个文档单独搜索求出集合，将逻辑运算转换为集合运算
  - \* 选出计算后符合条件的文档集合，再对文档进行排序
- \* 相关性检索
  - \* 使用TF\*IDF数值代替原本的TF数值进行逻辑表达式运算
  - \* 联想同义词参与检索（仅考虑相关性给定数量的同义词）

### 算法：中缀表达式求值

- \* 先将中缀表达式转为后缀（使用堆栈）
- \* 再对每个文件进行后缀表达式求值（使用堆栈）

The screenshot shows a frequency sidebar for the word "movie". The sidebar includes buttons for "Show 5 words, 3 results/file." and "Get words". The results pane displays the following text:

**movie (TF=461)**  
time. This movie is so wasteful of talent, it is truly... one. The movie moves at a snail's pace, is photo... When a movie is this worthless, it doesn't require... Avoid this movie... through the movie but just enough to catch watch...

**story (TF=138)**  
Story of a man who has unnatural feelings for a p... an uplifting story of curing illiteracy, watching it is ... of the story dictating the style in which the movie ... wrote a story to suite it. And he has failed in it ver... forcing the story along the path where they want i...

**good (TF=123)**  
with some good cinematography by future great ... but some good acting.<br /><br />4/10 Director, ... not in good way. Because of the style this film ha... It's always good that movie uses music to make t... a single good line or character in the whole mess...

**great (TF=53)**  
comment said "great acting." In my opinion, the...

## Frequency - 词频边栏

### 词频排序:

- \* 实现按TF检索最高词频，可设置显示条数
- \* 选中指定条目在右侧显示第一个匹配的文档，自动标注第一个匹配的词

### 达成题目要求:

- \* 基础要求(4) “对于给定的多篇文章构成的文档集中，统计不同词汇的出现频率，并进行排序，在界面中显示TOP 30的排序结果。”

### 算法：二级倒排索引

- \* 同多文件检索栏使用的二级倒排索引
- \* 二级索引统计单词出现总次数，并进行排序

## 2 数据结构设计

此部分要求说明所定义的数据结构和系统的主要模块。这正好对应我 *src/lib* 目录下作为基础的数据结构模块和 *src/components* 目录下实现业务逻辑的组件部分。但与数据结构地位相当的一些算法难以恰当地列入这个分类中，为此我增加了所实现的算法部分。下面逐一进行说明。

### 2.1 所定义的数据结构

由于使用 JavaScript 语言开发，需要首先说明两点：

- JavaScript 为动态类型语言，以下给出的类型仅为设计中字段应该具有的类型
- JavaScript 中的对象操作均为引用，所有对象字段均应理解为对象的引用

#### 2.1.1 文件树

此部分主要在 *src/lib/files.js* 中，我定义了文件结点类 `FileNode`，每个对应一个文件或目录，以及文件树类 `FileTree`，用来表示一整棵文件树。

**FileNode 类** 主要包含以下字段：

表 1: FileNode 类的字段

字段名	类型	含义说明
name	String	文件或目录名称
parent	FileNode	父级目录
children	FileNode[ ]	子级目录或文件
handler	FileSystemHandle	文件 API 的句柄
kind	String	结点类型（文件或目录）
key	String	唯一标识符
loaded	Boolean	文件内容是否已载入
content	String	文件内容
hash	Number	文件内容哈希值
lastSavedHash	Number	上次保存文件时的哈希值
saved	Boolean	内容已保存
workspace	WorkspaceIndex	全局索引
unsavedChildren	FileNode Set	未保存的子目录或文件
title	String	显示的文件标题
slots	Object	附加显示属性

文件结点类 `FileNode` 实现了以下功能：

- 扫描目录：递归地扫描目录下所有的子目录和文件
- 载入、保存：从文件系统中加载、保存到文件系统
- 显示保存状态：通过哈希值判断是否已保存，同时递归更新到父结点

- 更新索引：内容变化时创新新的倒排索引

**FileTree** 类 包含以下字段：

表 2: FileTree 类的字段

字段名	类型	含义说明
root	FileNode	根目录结点

文件树类 FileTree 实现了以下功能：

- 打开目录：调用操作系统提供的 API 打开指定目录并获得读写权限
- 扫描目录：从根目录开始递归扫描所有目录
- 全部载入、全部保存：递归地载入全部文件（并建立索引）、保存全部文件

### 2.1.2 哈希表、集合、映射表

此部分主要在 *src/lib/hash.js* 中，我定义了哈希表类 HashTable，实现了字符串的闭散列，集合类 HashSet，用于表示字符串的集合，映射表类 HashMap，用于表示字符串到任意对象的映射表（字典）。

**HashTable** 类 包含以下字段：

表 3: HashTable 类的字段

字段名	类型	含义说明
capacity	Number	闭散列最大容量
array	String[ ]	闭散列数组
h	function(String) Number	哈希码函数
p	function(Number) Number	探查函数
size	Number	当前容量
maxTryTimes	Number	最大探查次数

哈希表类 HashTable 实现了以下功能：

- 创建：支持设定容量和探测函数（内置了线性探测和二次探测）
- 查找、获取：查找指定字符串，返回状态或哈希编码
- 增加、删除：向哈希表内增加字符串或删除字符串
- 转换为广义表：遍历得到所有存储的字符串并转换为广义表

**HashSet** 类 包含以下字段：

表 4: HashSet 类的字段

字段名	类型	含义说明
hashTable	HashTable	哈希表

集合类 HashSet 实现了以下功能：

- 创建：创建一个集合，调用哈希表实现，还可以从广义表或映射表（只保留键）转换得来
- 查找、获取、增加、删除、转换为广义表：调用哈希表实现
- 交、并、补：集合的交并补运算

**HashMap** 类 包含以下字段：

表 5: HashMap 类的字段

字段名	类型	含义说明
hashTable	HashTable	哈希表
array	Object[ ]	值数组

映射表类 HashMap 实现了以下功能：

- 创建：创建一个映射表，调用哈希表实现
- 查找、获取、设置、删除、转换为广义表：调用哈希表实现
- 转换为广义表并排序：转换为广义表后，用堆进行排序

### 2.1.3 堆

此部分主要在 *src/lib/heap.js* 中，我定义了堆类 Heap，实现了任意类型的堆。

**Heap** 类 包含以下字段：

表 6: Heap 类的字段

字段名	类型	含义说明
items	Object[ ]	元素数组
greater	function(Object, Object)	元素比较函数
size	Number	当前堆大小

堆类 Heap 实现了以下功能：

- 创建：由任意类型数组创建堆，同时提供比较函数
- ShiftDown、ShiftUp：实现两种维护有序性的转移操作
- 压入、弹出：压入或弹出元素的同时维护堆的有序性
- 堆排序：数组建堆后逐一弹出实现排序

### 2.1.4 哈夫曼树

此部分主要在 *src/lib/huffman.js* 中，我定义了哈夫曼结点类 HuffmanNode 和哈夫曼树类 HuffmanTree，实现了哈夫曼编码和译码。

**HuffmanNode 类** 包含以下字段：

表 7: HuffmanNode 类的字段

字段名	类型	含义说明
char	Number	表示字符的 ASCII 码 (中间结点为-1)
weight	Number	权重
children	HuffmanNode[ ]	子节点
parent	HuffmanNode	父节点
code	String	编码结果

哈夫曼结点类 HuffmanNode 实现了以下功能：

- 创建：创建为对应字符的叶子结点或中间结点
- 合并：构造两个哈夫曼结点的父结点

**HuffmanTree 类** 包含以下字段：

表 8: HuffmanTree 类的字段

字段名	类型	含义说明
root	HuffmanNode	根目录结点

哈夫曼树类 HuffmanTree 实现了以下功能：

- 由字符和频率创建：根据字符和频率先生成叶子结点再逐步合并生成一棵树
- 由对应表导入：根据字符和前缀码通过前序遍历生成一棵树
- 生成对应表：遍历树来实现字符和前缀码的对应表
- 生成图数据：中序遍历树生成可视化使用的结点关系和坐标

由哈夫曼树类，可实现哈夫曼编码译码功能：

- 编码：由字符和频率创建树，生成对应表和编码后的比特流
- 译码：由对应表创建树，生成译码后的文段

**base64 类** 为静态类，不包含字段。

### 2.1.5 base64 编解码器

此部分主要在 `src/lib/base64.js` 中，我定义了静态的 Base64 编解码类 base64。Base64 编解码类 base64 实现了以下功能：

- 编码：将任意比特流编码为 base64 形式
- 解码：将 base64 形式字符串解码为比特流

### 2.1.6 替换器

此部分主要在 `src/lib/KMP.js` 中，我定义了替换器类 `Replacer`，实现了自动修正位置的单次替换、迭代器式替换、全部替换。

**KMP 算法** 替换器依赖于 KMP 算法，因此需要先介绍实现的 KMP 算法：

- `prefixFunction` 函数：生成前缀数组
- `findAll` 函数、`split` 函数：基于前缀数组实现字符串匹配、拆分

**Replacer 类** 包含以下字段：

表 9: Replacer 类的字段

字段名	类型	含义说明
str	String	处理的字符串
source	String	替换前的关键词
target	String	替换后的关键词
lastReplaced	Number	上次替换位置

注：替换器 `Replacer` 从 `source` 关键词替换为 `target` 关键词。

替换器类 `Replacer` 实现了以下功能：

- 查找：查找所有匹配位置
- 单次替换：替换当前位置，并记录文本长度变化，保证下一次替换位置正确
- 全部替换：不断调用单次替换直到全部替换完成

### 2.1.7 倒排索引

此部分主要在 `src/lib/invertedIndex.js` 中，我定义了文件索引类 `FileIndex`，实现了对单文件内各个单词位置的索引和多文件索引类 `WorkspaceIndex`，实现了对单文件索引的索引。整体结构为一个二级索引。

**FileIndex 类** 包含以下字段：

表 10: FileIndex 类的字段

字段名	类型	含义说明
map	HashMap	词到词位置与词频的映射表
hash	Number	索引内容的哈希码
workspace	WorkspaceIndex	全局索引
fileNode	FileNode	文件结点

文件索引类 `FileIndex` 实现了以下功能：

- 创建：根据文件内容（字符串）创建倒排索引，每次文件内容变化时重新创建。

- 获取词频 TF：实现词频统计，可以方便地获取某文件内指定词的词频
- 获取位置：获取某文件内指定词出现的各个位置

**WorkspaceIndex 类** 主要包含以下字段：

表 11: WorkspaceIndex 类的字段

字段名	类型	含义说明
map	HashMap	词到各文件词频和总词频的映射表
fileIndexes	HashMap	内容哈希到文件索引的映射表

多文件索引类 WorkspaceIndex 实现了以下功能：

- 增、删文件索引：增添包含的文件索引，使用文件哈希作一致性判断
- 统计词频：索引文件索引的同时，统计多文件整体的词频
- 获取逆文本频率指数 IDF：根据多文件 TF 计算指定词的 IDF 值

## 2.2 所实现的算法

### 2.2.1 表达式转换与求值

此部分主要在 *src/lib/expression.js* 中，我实现了中缀转后缀表达式 InFixExpressionToPostfix 和后缀表达式求值 ExecutePostfixExpression。

**中缀转后缀表达式 InFixExpressionToPostfix 算法** 使用一个运算符堆栈实现，可以实现支持单种括号匹配、运算符优先级的中缀表达式转后缀表达式。

**后缀表达式求值 ExecutePostfixExpression 算法** 使用一个运算数堆栈实现，可以实现自定义运算符、运算函数的后缀表达式求值。

### 2.2.2 高级检索

此部分主要在 *src/lib/advancedSearch.js* 中，我主要实现了匹配文档集合运算 CalculateSet、相关性权重计算 calculateWeight、同义词扩展 AddSynonyms。

**匹配文档集合运算 CalculateSet 算法** 将检索表达式中的 AND, OR, NOT 视为求交集、求并集和全集求补集运算，筛选出匹配的文档。

**相关性权重计算 calculateWeight 算法** 将检索表达式中的 AND, OR, NOT 视为求最小值、求最大值和求差值运算，计算出各个文档的相关性权重。

**同义词扩展 AddSynonyms 算法** 启用同义词拓展时，将每个检索关键词转换为其本身和同义词相或的表达式，再进行计算。

## 2.3 系统整体结构以及各模块的功能描述

此部分将系统按功能分为大体模块，并描述每个模块。这种划分与我使用 Vue 框架对系统进行组件划分的思路基本一致，因此以下按组件划分进行说明。

### 2.3.1 整体组件 App

整个系统均包含在这个组件中，它复制系统组件间的信息通讯、页面整体布局、全局初始化等功能。

下图中展示了整体的布局，可见界面首先在上、下切出了两块页眉 Header 和页脚 Footer 部分；之后再从左到右划分为侧栏切换菜单 Menu、侧栏 Side Bar 和编辑器栏 Editor Bar。

其中，侧栏和编辑器栏可以切换不同的功能。侧栏使用左侧的侧栏切换菜单进行切换，侧栏和编辑器栏的占比可以随意调整，也可以彻底收起侧栏；而编辑器栏会自动根据当前选择的文件或目录情况进行切换。

下面前四个以“界面”结尾的组件均用于编辑器栏，而后面五个以“侧栏”结尾的组件均用于侧栏。

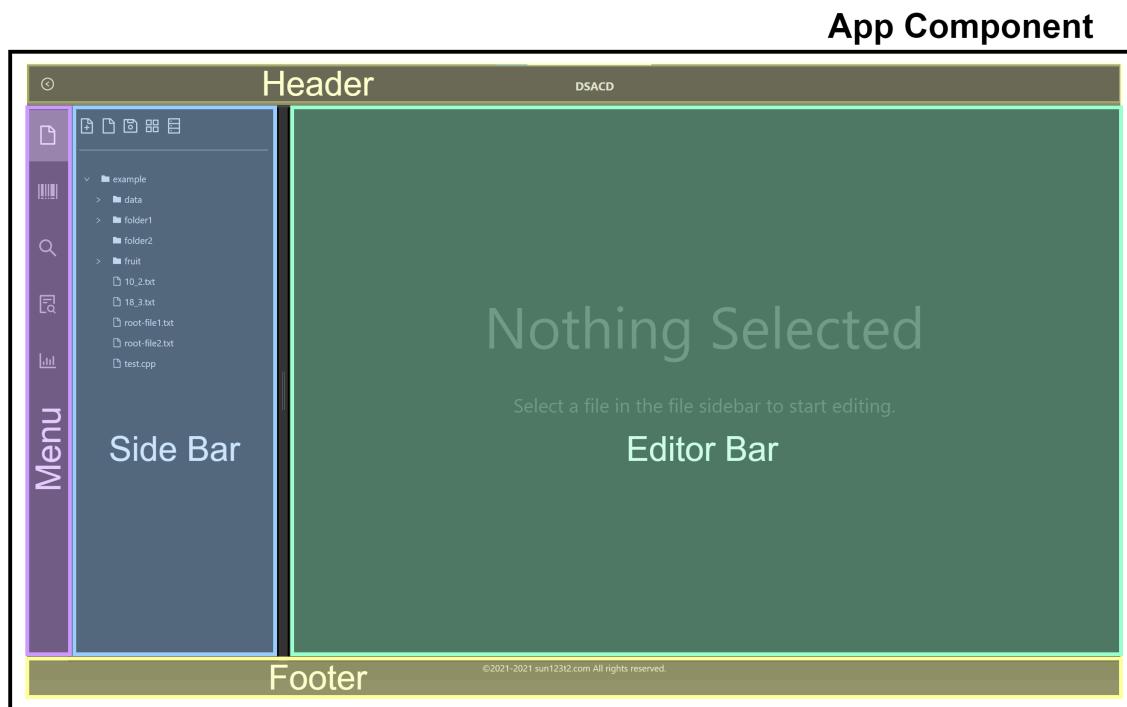


图 1: 整体布局

### 2.3.2 开始界面 StartPanel

进入系统后，用户侧栏关闭，并进入开始界面。必须选择打开某文件夹及其内部内容，或选择某文件夹作为存储位置（不扫描其内部内容）。在选择文件夹前无法进行其他操作，包括打开侧栏和任何文件编辑功能。

**打开文件夹 Import folder** 该功能会创建文件树、创建多文件索引，并递归地扫描目录以创建文件树中各结点（并不会读取文件内容）。完成以上工作后，会自动切换到文件侧栏和空白界面，引导用户开始编辑文件。

**选择存储目录 Select folder** 该功能与打开文件夹类似，只是不会递归扫描目录，仅创建选择的文件夹作为文件树的根节点。

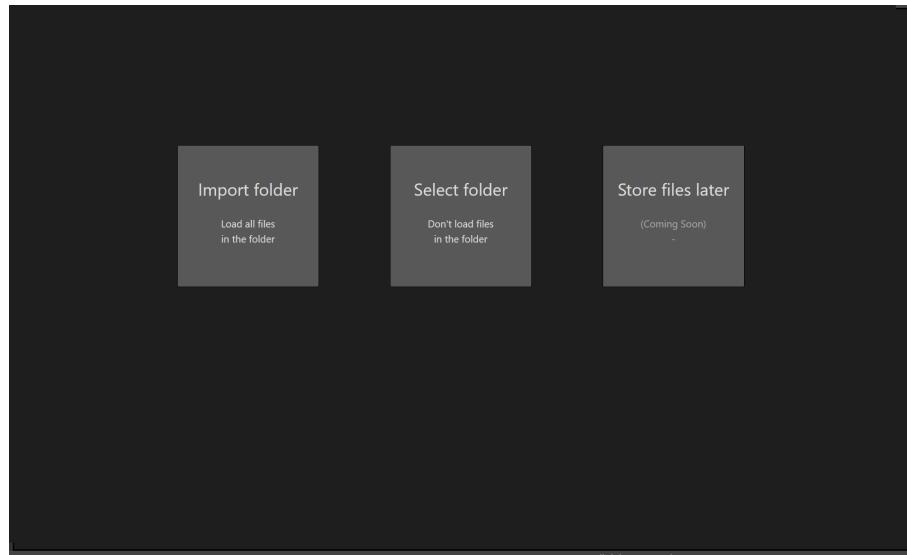


图 2: 开始界面

### 2.3.3 空白界面 DummyEditorBar

当用户未选择任何文件时，右侧会显示空白界面，提示用户从左侧的文件侧栏选择文件。



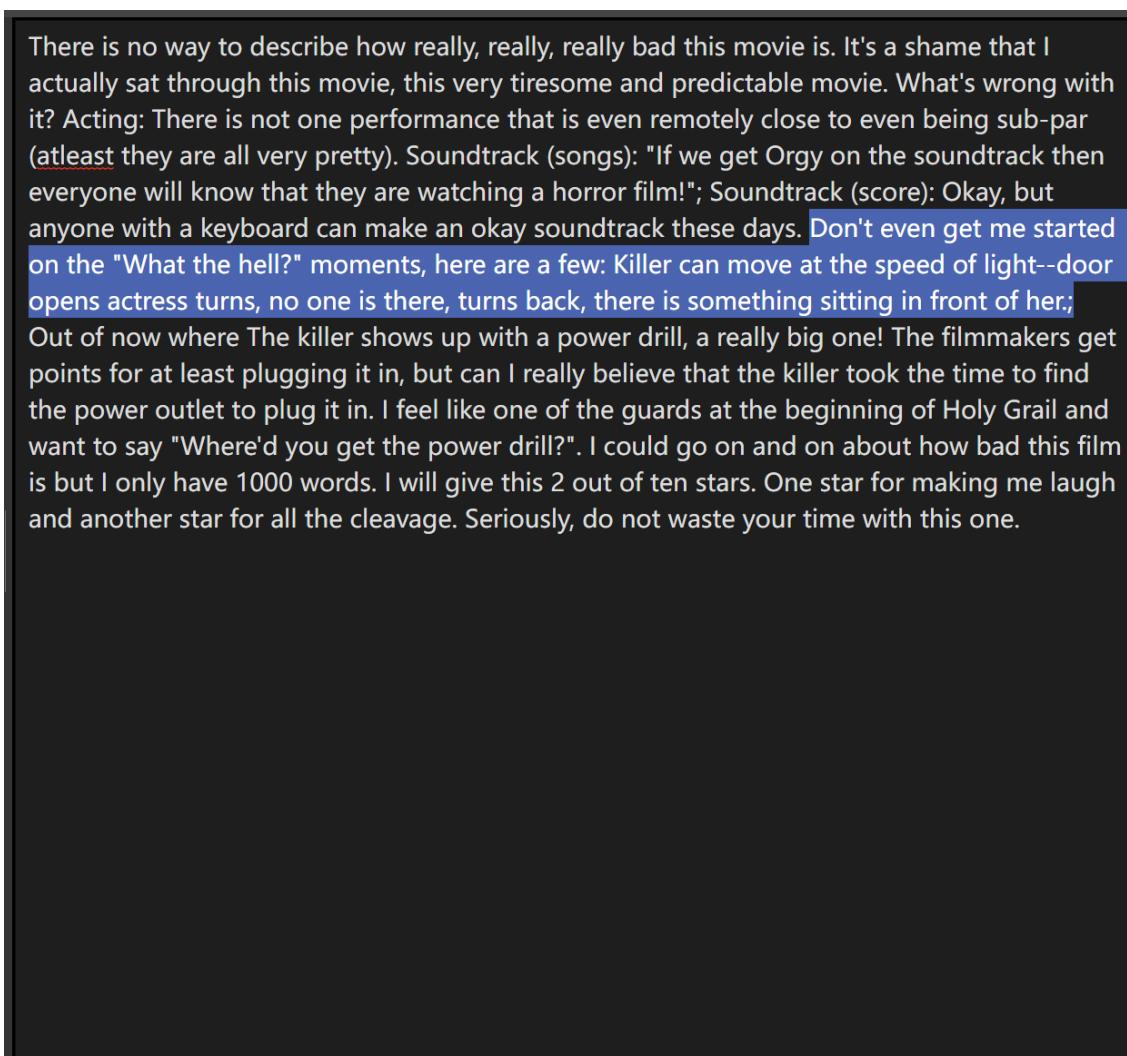
图 3: 空白界面

### 2.3.4 编辑器界面 Editor

当用户选择一个文件后，会自动切换到此界面。用户可以在其中随意修改文件中的文本内容。此外，编辑器界面还提供以下功能和其他模块交互。

- 内容更新通知：主要用于和文件模块交互，当文件内容被改变，且用户失去编辑器焦点或切换文件时，会向文件模块传递信息以更新文件未保存状态
- 文件切换：切换文件时，会自动缓存前一个文件，并打开另一个文件

- 内容选取：可以根据开始和终止位置自动选取指定位置的文字
- 内容提取和覆盖：可以对出用户当前选中的文字进行提取或覆盖



There is no way to describe how really, really, really bad this movie is. It's a shame that I actually sat through this movie, this very tiresome and predictable movie. What's wrong with it? Acting: There is not one performance that is even remotely close to even being sub-par (atleast they are all very pretty). Soundtrack (songs): "If we get Orgy on the soundtrack then everyone will know that they are watching a horror film!"; Soundtrack (score): Okay, but anyone with a keyboard can make an okay soundtrack these days. Don't even get me started on the "What the hell?" moments, here are a few: Killer can move at the speed of light--door opens actress turns, no one is there, turns back, there is something sitting in front of her; Out of now where The killer shows up with a power drill, a really big one! The filmmakers get points for at least plugging it in, but can I really believe that the killer took the time to find the power outlet to plug it in. I feel like one of the guards at the beginning of Holy Grail and want to say "Where'd you get the power drill?". I could go on and on about how bad this film is but I only have 1000 words. I will give this 2 out of ten stars. One star for making me laugh and another star for all the cleavage. Seriously, do not waste your time with this one.

图 4: 编辑器界面

### 2.3.5 目录预览界面 DivViewer

当用户选择一个文件后，会自动切换到此界面。此界面会预览此目录直接包含的目录和文件，若较多则仅显示前 5 个。



图 5: 目录预览界面

### 2.3.6 文件侧栏 FileButtons 和 FileTree

文件侧栏可分为 FileButtons 和 FileTree 两个部分，中间有一条分界线用于划分。

**文件操作按钮 FileButtons** 用于实现文件的创建、打开、保存、全部保存和未保存信息更新。

操作的目录称为“当前目录”，在选中文件时是选中文件同目录，选中目录时是选中的目录，未选中时是根目录。具体功能如下：

- 新建文件：网页内弹出窗口输入名称，在当前目录下新建文件。
- 打开文件：打开磁盘任意位置文件，并将其放入当前目录。
- 保存文件：保存当前打开的文件。
- 全部保存：保存所有未保存的文件。
- 更新未保存状态：更新所有文件的未保存状态（所有文件检查哈希）。

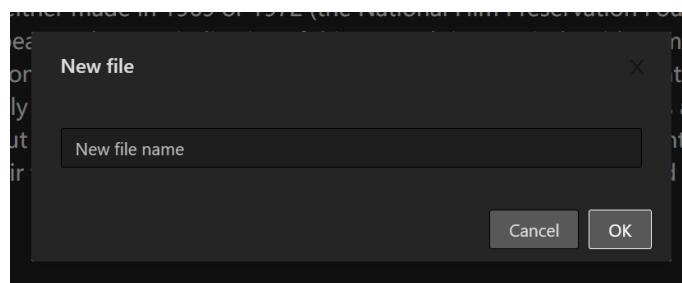


图 6: 新建文件的网页内弹窗

**文件树 FileTree** 用于树状展示文件的目录结构，选择当前编辑的文件，可收起或展开任意一个目录结点。

如下图中的左图展示了该模块树状展示文件目录结构，选择指定文件或目录的功能。当进行文件切换时，会自动切换编辑器栏，对应无选择、选择文件、选择目录，显示空白界面、编辑器界面、目录预览界面。切换到未载入的文件时，会先进行载入和索引。

如下图中的右图展示了标记未保存文件的功能。未保存的文件会被标为蓝色，同时其祖先目录也都会被标为蓝色。这样的设计主要用于解决文件被因此的情况（如图中的 file 文件夹，内部有文件未保存可以通过文件夹的蓝色发现）。

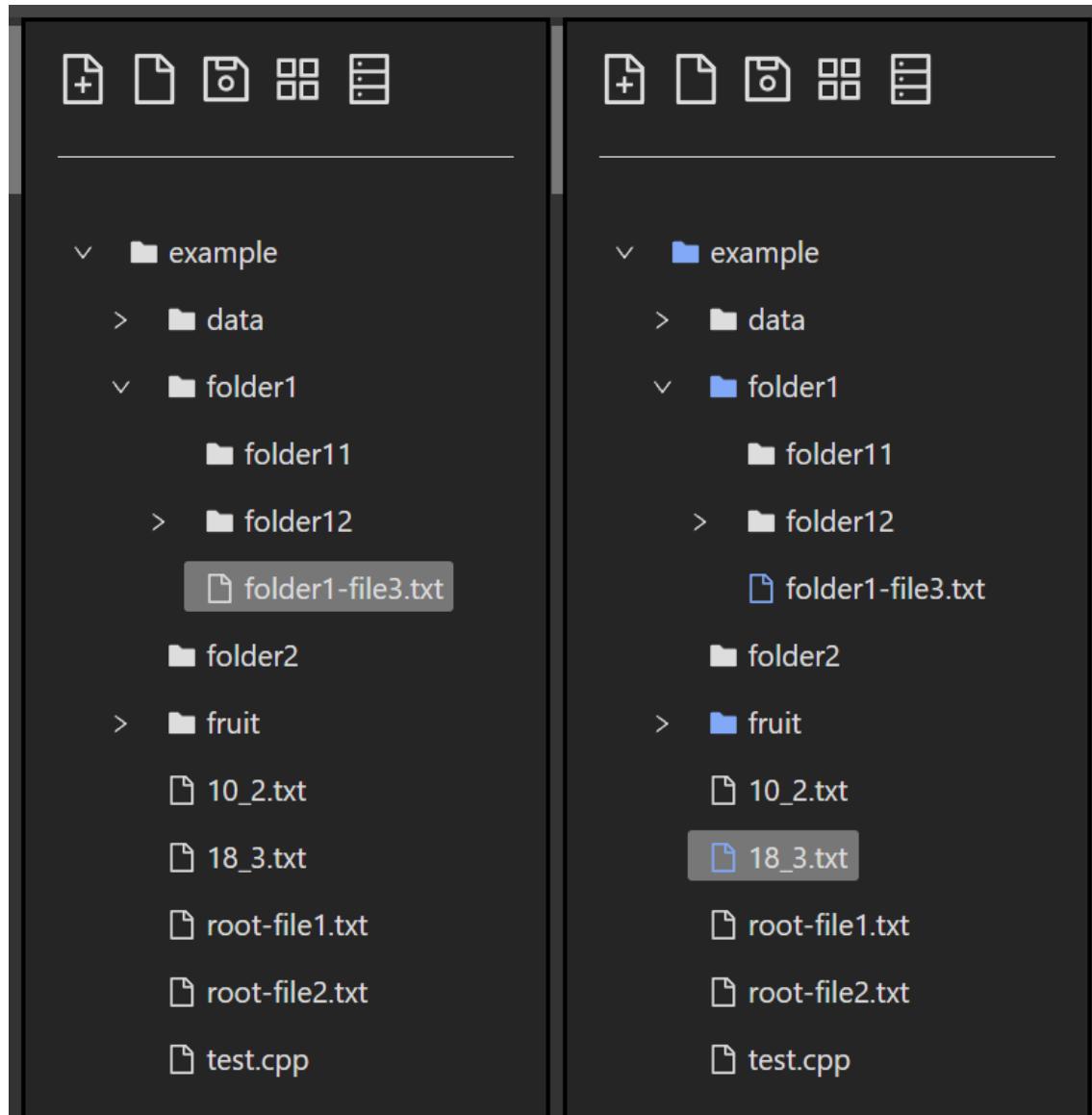


图 7: 文件侧栏

### 2.3.7 编码译码侧栏 EncodingPanel

编码译码侧栏如下图所示，分为原文 Content、哈夫曼编码比特流 Code、哈夫曼编码后 base64 结果 Code+base64、哈夫曼编码对应表 CodingMap 几部分。这几部分之间相互关联，下面具体阐述。

**Content 和编辑器中选中的文字** 原文文本框下方有两个按钮，其中 Load 表示将编辑器中选中的部分放到原文这里，而 Store 表示将原文内容替换掉编辑器中。

**Code** 和它的 base64 编码 **Code+base64** 这两个文本框时刻保证一致性。当 Code 发生变化时，会自动编码 base64 更新 Code+base64，反之当 Code+base64 发生变化时也会自动解码 base64 更新 Code。

base64 原本是对任意 ASCII 字符串进行编码，可以生成只由 64 种字符组成的字符串。而我这里对它进行了改造，可以对任意比特流编码为仅由 64 种字符组成的字符串。具体而言，对于任意比特流，首先要将其对齐到 8 的整数倍长度。为此我设定每个比特流前面增加一串数个 0 和一个 1 的填充部分，根据比特流长度不同填入 1 至 8 个比特。这样即可以顺利地进行 base64 编解码了。

**Content、Code 和 CodingMap** 这三者之间的关系就是哈夫曼编解码的关系。Content 处输入任意原文，系统会自动统计符号频率并生成它的哈夫曼编码 Code 和字符编码对应表 CodingMap。反过来，当输入对应表 CodingMap 和哈夫曼编码 Code 时，可以解码处原文 Content。编码和解码的过程分别由下方的按钮触发。

**哈夫曼树可视化** 只有对应表 CodingMap 非空，点击显示哈夫曼树的按钮即可出现一个页内弹窗显示哈夫曼树。这个哈夫曼树的结点位置通过中序遍历生成，每两个中序遍历相邻的结点距离一定。展示的哈夫曼树显示各结点前缀码、权重和叶子结点代表的字符，可以用鼠标缩放、移动。

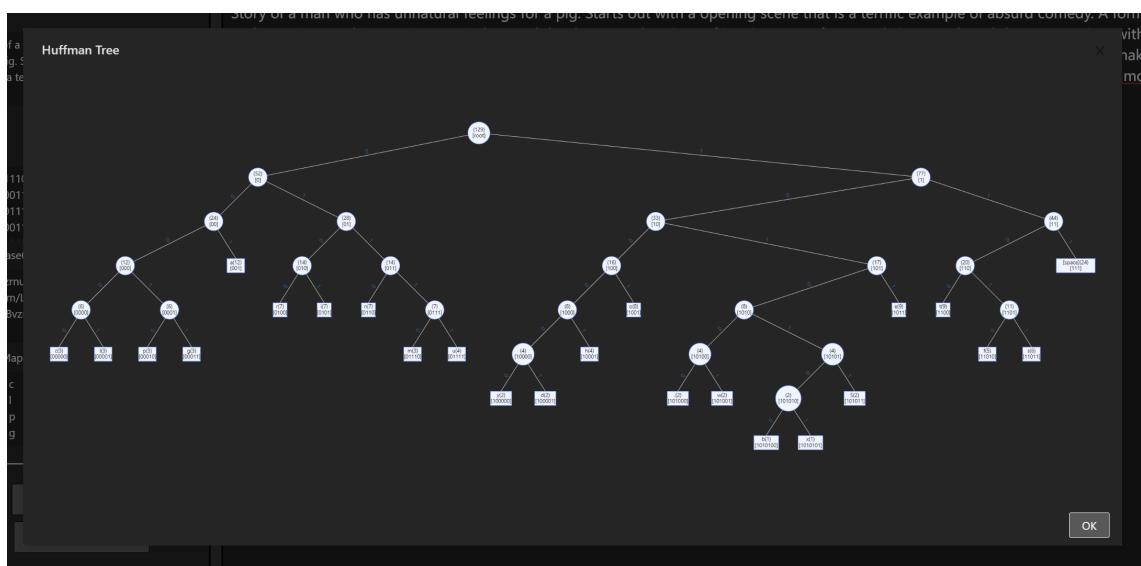


图 8: 哈夫曼树可视化页内弹窗

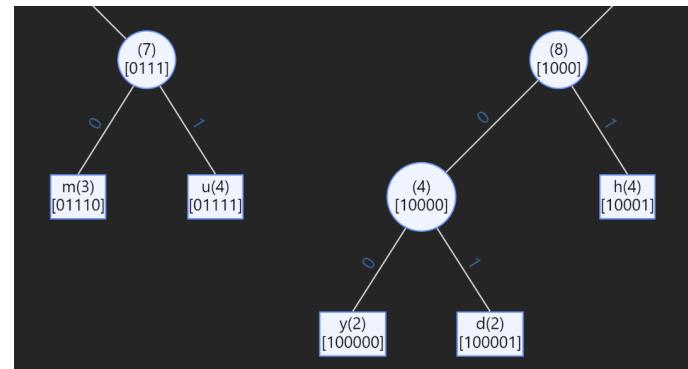


图 9: 哈夫曼树可视化局部

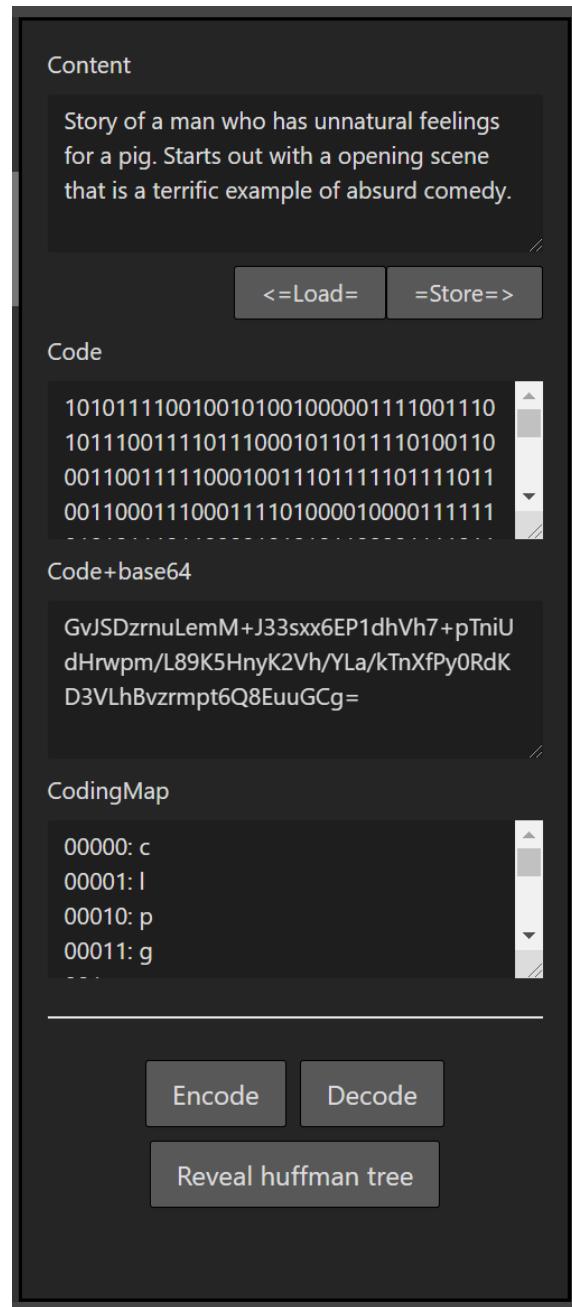


图 10: 编码译码侧栏

### 2.3.8 查找替换侧栏 FindAndReplace

下图中展示了查找替换侧栏的两个状态，查找和替换状态。两个状态下的数据相互独立，可以在两个模式下随意切换，每个模式各自保留数据。

**查找状态** 在当前打开的文件内检索指定内容，其功能如下：

- 按下按钮后即会使用 KMP 算法进行匹配检索，并将结果列在下方
- 匹配的字符串会用黄底黑字高亮显示，同时自动解析前后若干个词（空格分隔）或达到指定最长字符数
- 上下按钮点击切换或直接点击某个结果，均可以自动在编辑器区域选中对应内容

**替换状态** 在当前打开的文件内替换指定内容，其功能如下：

- 同样使用 KMP 算法匹配，高亮并预览替换结果显示在下方
- 上下按钮点击切换或直接点击某个结果，均可以自动在编辑器区域选中对应内容
- 点击 Replace 按钮会替换当前文本，并自动使用替换后的文本长度变化，跳转到下一个替换处
- 点击 Replace All 按钮会替换掉所有可替换目标（仅会替换一次，如 a 替换为 ab 只会替换最初文本中的 a，完成后会再次检索）

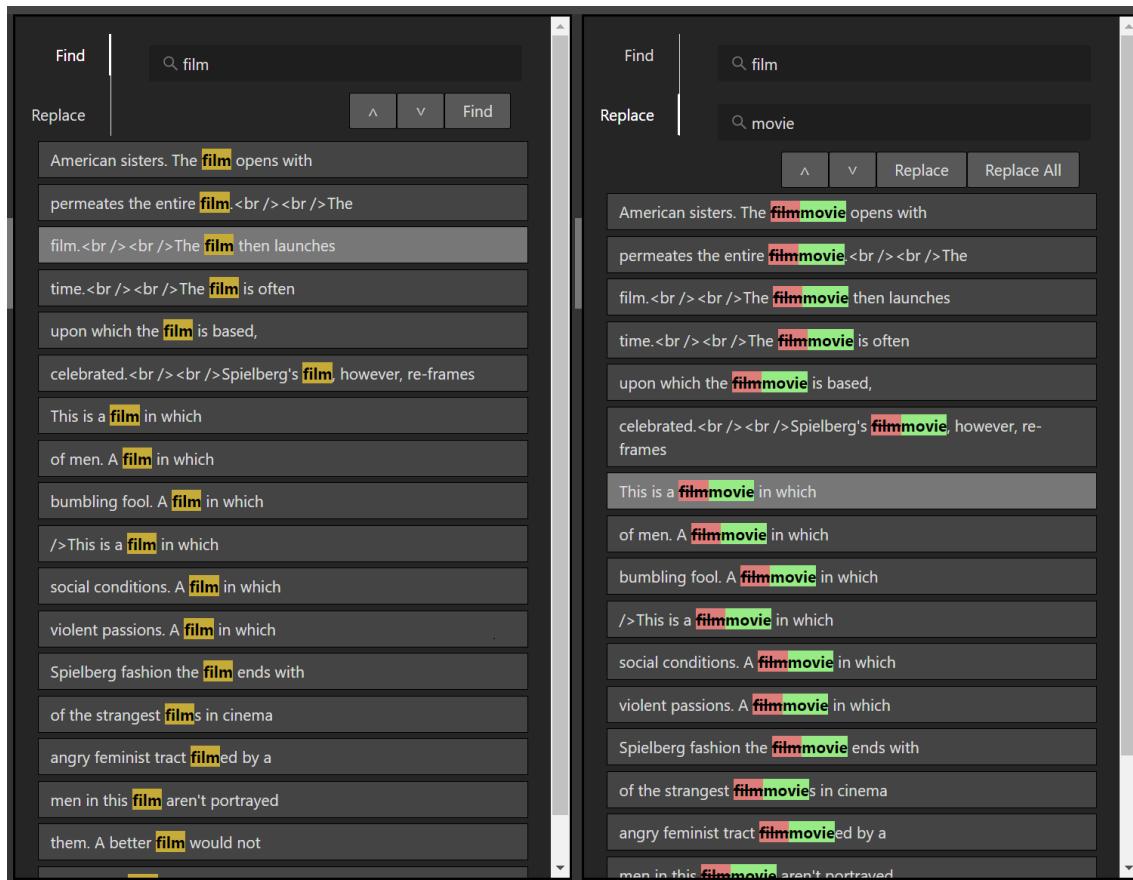


图 11: 查找替换侧栏

### 2.3.9 高级检索侧栏 AdvancedSearch

高级检索侧栏对目前工作区中所有文件（即最开始打开文件夹时扫描到的所有文件和后添加的文件）进行搜索。

当切换到此界面时，对工作区全部未索引文件进行索引，高级检索基于二级索引的结果，因此只有在索引全部完成后才能进行。

**表达式检索** 在搜索框中输入检索表达式，点击搜索后会自动对表达式进行解析，根据集合运算找出匹配的文档，再根据 TF-IDF 权重进行相关性排序。最终显示到下方，每个文件显示各个匹配的位置，点击每条匹配词汇可以跳转到对应文件并自动选中对应位置。

**同义词拓展** 当启用同义词非 0 时，系统会查询存储的同义词数据库，寻找指定数量的同义词，以或的方式加入到检索表达式中。每个关键词找到的同义词会被列在检索结果上方，同时附上 IDF 值。

**选择显示数量** 用户可以选择显示的总文件数、单文件匹配位置数，实现自定义的显示效果。其中-1 表示不限制，即显示所有的元素。

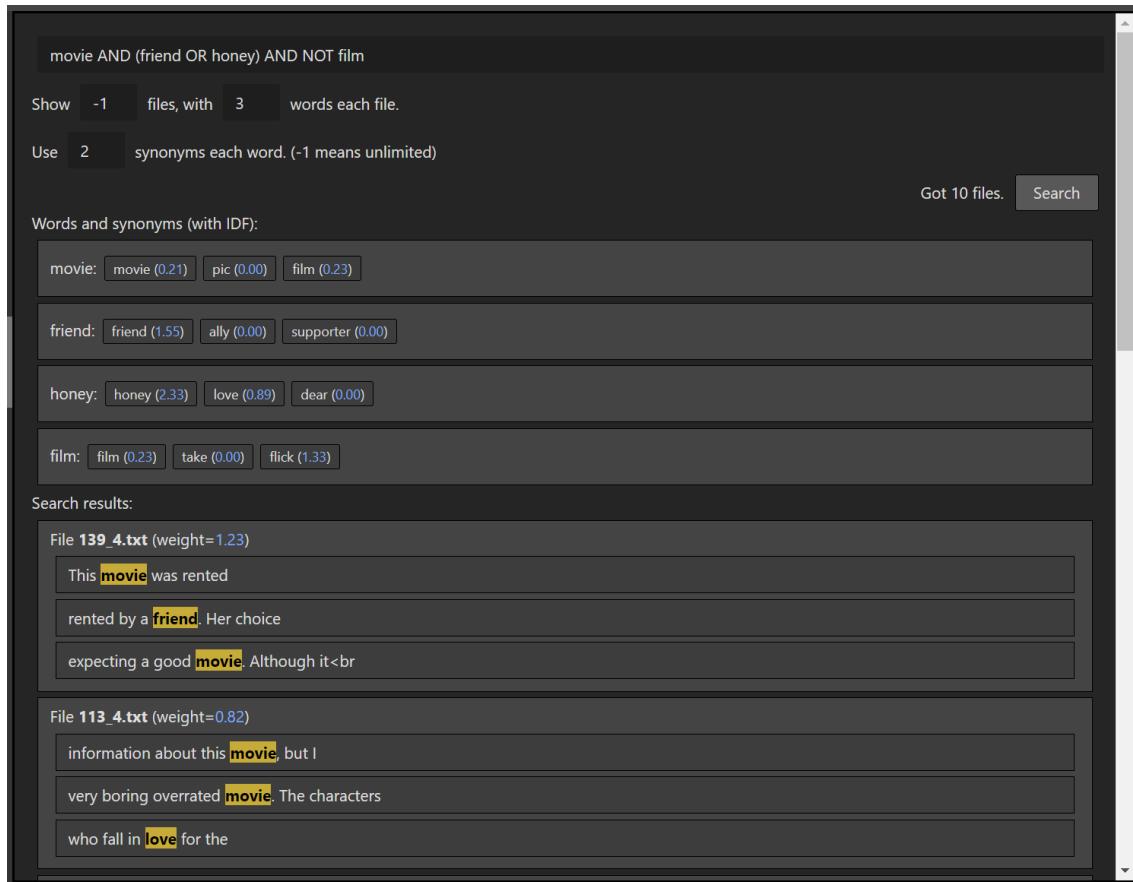


图 12: 高级检索侧栏

### 2.3.10 词频侧栏 Frequency

词频侧栏对目前工作区中所有文件进行词频排序。

类似高级检索，当切换到此界面时，对工作区全部未索引文件进行索引。二级倒排索引会对词频进行统计，因此词频模块基本上就是将统计的结果显示出来。

**三级显示层次** 词频统计分为三级显示，首先是按总词频统计对单词排序，然后每个词内部按文件词频对文件排序，最内部按词汇出现位置按先后排序。

**选择显示数量** 用户可以选择显示的总词汇数、每个词汇显示的文件数、每个文件显示的匹配数。其中-1 表示不限制，即显示所有的元素。

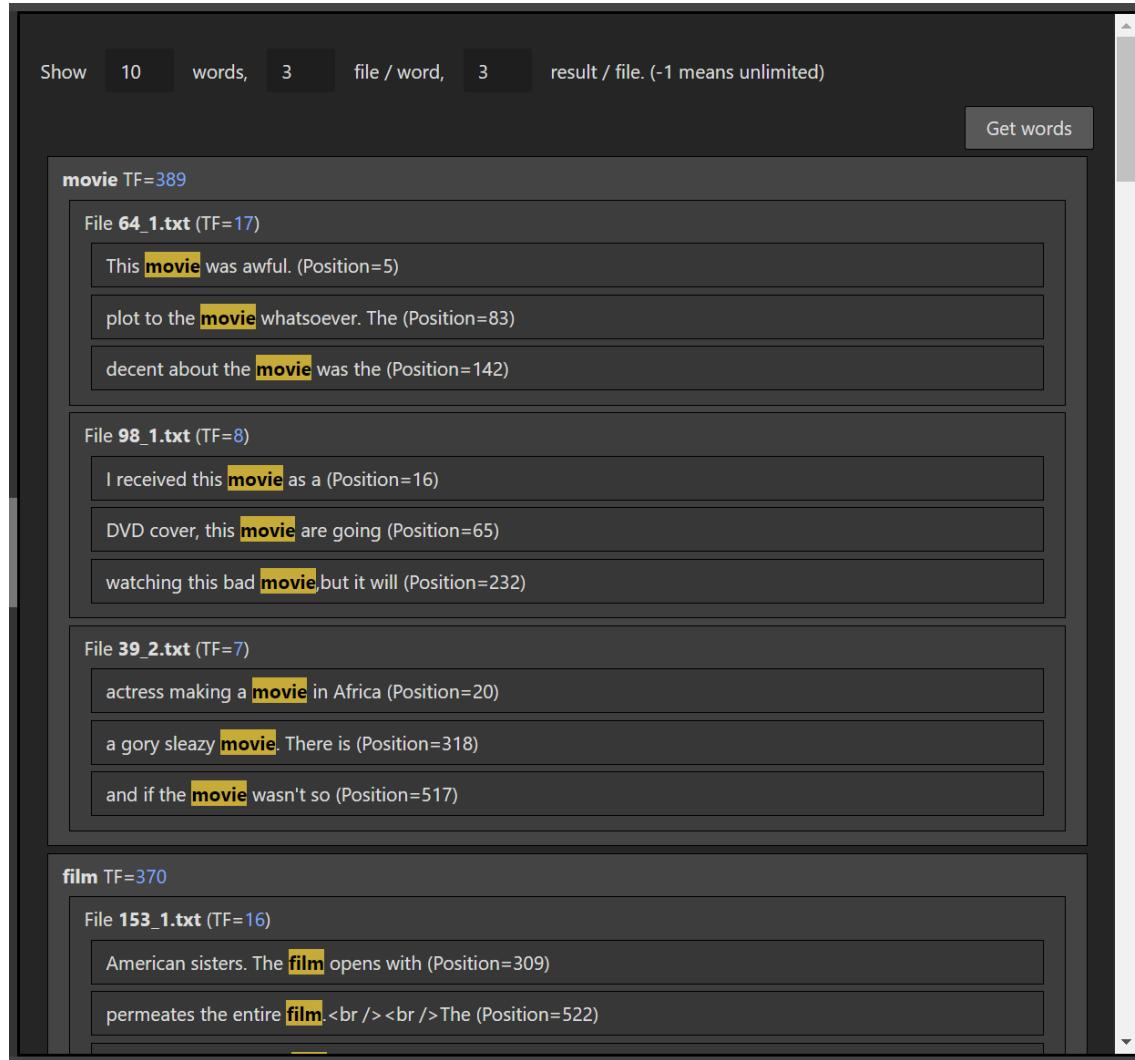


图 13: 词频侧栏

### 3 详细设计

详细设计中对主要的算法进行说明，包括数据结构的成员方法和独立的函数，较为次要的实现细节不再赘述。此外，给出系统整体和各部分的流程图，说明各部分直接的运行关系。

#### 3.1 KMP 算法

KMP 算法依赖前缀函数实现字符串的匹配。这里重点说明前缀函数的求取算法，随后简单说明如何使用前缀函数匹配字符串。

##### 3.1.1 前缀函数

**前缀函数的定义** 任取长度为  $n$  的字符串  $s$ ，其前缀函数为一个长度为  $n$  的数组  $\pi$ 。其中  $\pi[i]$  等于子串  $s[0 \dots i]$  的最长相等真前缀、真后缀的长度（其中，真前缀、真后缀指与原字符串不相等的前缀、后缀字符串）。特别地，规定  $\pi[0] = 0$ 。

用数学语言可描述如下：

$$\begin{aligned} \forall s, \text{length}(s) = n, \\ \exists \pi[i] = \max_{0 \leq k \leq i-1} (\{k+1 | s[0 \dots k] = s[i-k \dots i]\} \cup \{0\}), 0 \leq i \leq n \end{aligned}$$

**求前缀函数** 思想主要包括以下几点：

- 从  $\pi[1]$  到  $\pi[n]$  逐一计算，前缀函数值每次最多增加 1，分为增加一和其他两种情况
- 增加一：当  $s[i+1] = s[\pi[i]]$  时， $\pi[i+1] = \pi[i] + 1$
- 其他：当  $s[i+1] \neq s[\pi[i]]$  时，我们寻找最大的  $j \leq \pi[i]$ ，满足  $s[0 \dots j-1] = s[i-j+1 \dots i]$ 
  - 由  $\pi[i]$  的定义，有  $s[0 \dots \pi[i]-1] = s[i-\pi[i]+1 \dots i]$
  - 因为  $j \leq \pi[i]$ ， $s[i-j+1 \dots i]$  是  $s[i-\pi[i]+1 \dots i]$  的子串
  - 故两等式联合有  $s[0 \dots j-1] = s[i-j+1 \dots i] = s[\pi[i]-j \dots \pi[i]-1]$
  - 根据  $\pi$  的定义，由上式可知  $j = \pi[\pi[i]-1]$

以上算法中“其他”情况下可以直观地参照下式理解：

- 首先，已知两个  $\pi[i]$  长的字符串（上方大括号）相等，可以知道（从左到右数）第 2 个和第 3 个标记出的长度为  $j$  的字符串相等
- 随后，要求第 1 个和第 3 个长度为  $j$  的字符串相等，就等价于要求第 1 个和第 2 个长度为  $j$  的字符串相等，即是求  $\pi[\pi[i]-1]$  时的要求

$$\underbrace{s_0 s_1 s_2 s_3}_{j} \underbrace{s_4 s_5}_{j} \dots \underbrace{s_{i-5} s_{i-4} s_{i-3}}_{j} \underbrace{s_{i-2} s_{i-1} s_i}_{j} s_{i+1}$$

求解前缀函数算法的伪代码如下：

---

**Algorithm 1** 求前缀函数
 

---

**Input:** A string  $s$

**Output:** The prefix array  $\pi$

```

1: procedure PREFIXFUNCTION( $s$ )
2:    $\pi \leftarrow []$ 
3:    $\pi.\text{PUSHBACK}(0)$ 
4:   for  $i$  in 0 to LENGTH( $s$ ) do
5:      $j \leftarrow \pi[i-1]$ 
6:     while  $j > 0$  and  $\pi[i] \neq \pi[j]$  do  $j = \pi[j - 1]$ 
7:     end while
8:     if  $s[i] = s[j]$  then
9:        $\pi.\text{PUSHBACK}(j + 1)$ 
10:    else
11:       $\pi.\text{PUSHBACK}(0)$ 
12:    end if
13:   end for
14: end procedure

```

---

### 3.1.2 使用前缀函数匹配字符串

使用前缀函数求解算法可以轻易地匹配字符串。

首先，拼接出形如  $Target \phi Content$  的字符串，其中  $Target$  是搜索目标， $Content$  是检索的原文，而  $\phi$  是二者中都不可能出现的特殊字符。

其次，对拼接出的字符串求前缀函数，当出现  $i > length(Target)$  且  $\pi[i] = length(Target)$  的时候，意味着在  $Content$  中  $i - 2 \times length(Target) - 1$  的位置出现了  $Target$ 。

## 3.2 闭散列

我将功能划分为散列函数  $h()$ 、探查函数  $p()$ 、闭散列框架三部分。其中前两者可根据需求进行替换。这样实现了高自由度的闭散列功能。

### 3.2.1 闭散列框架

闭散列框架的核心功能是查找函数  $find()$ 。它可以借助散列函数  $h()$ 、探查函数  $p()$ ，来探查目标元素，并返回探查的结果（找到、目标为空或查找次数过多）和目标元素的位置。其他功能调用查找函数  $find()$  实现功能。

**查找函数  $find()$**  本质上是一个有最多次数限制的探查循环。从哈希码开始，每次循环后用哈希码、探查次数、目标元素通过探查函数  $p()$  获取下一个探查位置。其伪代码如下：

**Algorithm 2** 查找函数 *find()***Input:** Target Element e**Output:** Result, position of e

---

```

1: function FIND(e)
2:   d0  $\leftarrow$  H(e)
3:   d  $\leftarrow$  d0
4:   for i in 1 to MaxTryTimes do
5:     if array[d] = e then
6:       return FOUND, d
7:     else if array[d] = NULL then
8:       return EMPTY, d
9:     end if
10:    d  $\leftarrow$  (d0 + P(i, e) mod capacity + capacity) mod capacity
11:   end for
12:   return TIMES_EXCEED, d
13: end function

```

---

**对外提供功能** 闭散列基于查找函数 *find()* 对外提供了增加元素、查找元素、删除元素和列出现有所有元素的功能。

其中，前三者均会使用提供的字符串调用 *find()*，然后对结果和位置 *d* 进行不同的处理。具体如下：

表 12: 添加、查找、删除元素功能实现

功能名称	返回 FOUND	返回 EMPTY	返回 TIMES_EXCEED
添加元素	返回 <i>d</i>	设置 array[ <i>d</i> ] 为新元素 返回 <i>d</i>	返回 NULL
查找元素	返回 <i>d</i>	返回 NULL	返回 NULL
删除元素	设置 array[ <i>d</i> ] 为墓碑 返回	返回	返回

而列出现有所有元素的功能只需要遍历 array 即可实现。

**构造函数** 构造闭散列表时可以设定容量、最大检索次数和探查函数 *p()*。由于课设中所有散列表均针对字符串使用，因此固定了散列函数 *h()*。

### 3.2.2 散列函数 *h()*

由于结果不需要用于密码学相关用途，对字符串的散列只需要保证字符串的每位均会对结果产生较显著的影响即可。为此我采用 Stack Overflow 上经过讨论认为速度较快的算法。

主要思想是先将字符串变为字符数组，再逐一对字符的 ASCII 码进行  $result \leftarrow result \times 31 + new$  的迭代运算。伪代码如下：

---

**Algorithm 3** 散列函数  $h()$ 

---

**Input:** A string s**Output:** Hash code of s

```

1: function FIND(s)
2:   hash ← 0
3:   for i in 1 to LENGTH(s) do
4:     hash ← (hash « 5) - hash + ASCII(s[i])
5:   end for
6:   return hash
7: end function

```

---

3.2.3 探查函数  $p()$ 

实践中发现，由于默认使用的闭散列容量较大，且字符串的哈希码不太容易出现碰撞，较为简单的线性探查、二次探查就已经足够使用。

**线性探查** 单向逐一探查，公式如下：

$$p(i, e) = i$$

**二次探查** 越来越远地双向探查，公式如下：

$$p(i, e) = (-1)^{i+1} \times \left\lceil \frac{i}{2} \right\rceil^2$$

## 3.3 二级倒排索引

## 3.3.1 整体设计

二级倒排索引的目的是对多文档中词出现的频率、位置进行索引，并当部分文档变化时高效更新。具体设计如下：

- 一级索引为文件索引（FileIndex 类）
  - 对单文件的内容索引，记录每个词出现的所有位置
  - 对于每个词，出现位置的个数即是词频
  - 只读。本质上是对文本内容索引，当出现新文档或文档内容变更时（用户修改并移开焦点或搜索替换后）重新进行索引
- 二级索引为全局索引（WorkspaceIndex 类）
  - 对文件索引进行索引，记录每个词在不同出现的所有文件和数量
  - 对于每个词，维护出现它的文件索引的引用和总出现次数
  - 记录当前包含的所有文件索引，仅需在新增或移除文件索引时更新

注：这里提到的“词”指标准化处理后的词，系统会首先对文本中提取的每个词进行去停用词、转小写等操作。

### 3.3.2 数据结构设计

**文件索引** 包含以下内容：

表 13: 文件索引存储内容

名称	类型	含义
词位置索引	对应表: 字符串 → 数组	词 → 出现的所有位置
哈希值	数值	索引内容的哈希码，唯一标识文件索引
索引的文件结点	文件结点	用于提取内容等操作

**全局索引** 包含以下内容：

表 14: 全局索引存储内容

名称	类型	用途
词索引	对应表: 字符串 → 文件索引组和总词频	词 → 出现的文件和总词频
全部文件索引的索引	对应表: 数值 → 文件索引	文件索引哈希值 → 文件索引

### 3.3.3 算法设计

**文件索引** 文件索引是只读的，因此只需要每次根据内容构造以及查询词汇。

- **根据内容构造** 首先对内容按空格分割，对每个词进行标准化，随后不断从对应表中获取并添加位置。

- **查询词汇** 使用对应表的查询功能，返回所有位置的数组，对数组取长度可以得到词频。

**全局索引** 全局索引需要维护当前包含的文件索引，还要实现全局的查询词汇。

- **维护当前包含的索引** 即需要实现添加、删除索引。

全部文件索引的索引用于保证此操作合法。

需要对新添加或删除索引中的每个词，对应地在全局索引中进行处理，在对应词中新增或删除这个文件索引，并维护总词频。

- **全局查询词汇** 调用词索引中的查询功能即可，可以查询出现的各个文件，进而查询出现的各个位置；也可以快速地运用总词频排序。

### 3.4 相关性评价

在进行多文档的相关性检索时，需要对根据检索表达式对文档进行相关性排序。而排序的依据是相关性评价的值。

**表达式解析** 表达式由关键词、括号和 *AND*、*OR*、*NOT* 三种运算符组成。

由于表达式解析只支持二元运算符，我将所有的 *NOT X* 转换为 *ALL SUB X*。

在开启近义词扩展时，我会将每个关键词 *X* 变为其近义词  $X_1, X_2, \dots$  的或表达式。如  $X \rightarrow (X \text{ OR } X_1 \text{ OR } X_2 \text{ OR } \dots)$ 。

随后使用一个存放括号和运算符的栈即可将中缀表达式转换为后缀表达式。后缀表达式中没有括号，可以便捷地进行计算，表达式解析完成。

**集合运算：文档范围筛选** 对表达式进行集合运算，以此筛选出符合要求的文档。

每个关键词被它出现过的文档集合代替，*ALL* 对应全集。其中集合使用我基于闭散列实现的集合类 *HashSet* 对象。

运算 *AND*、*OR* 和 *SUB* 则分别对应集合的交、并、补运算。

**权重运算：相关性数值** 对表达式进行权重运算，以此计算出相关性数值用于排序。

每个关键词被它在该文档中出现的次数代替，*ALL* 赋予一个很大的常数。

运算 *AND*、*OR* 和 *SUB* 则分别对应数值的最小值、最大值、减运算。

**整体流程** 进行相关性检索式，首先解析表达式，然后用集合运算筛选出符合条件的文档，最后用权重运算为符合条件的文档计算相关值，按相关值排序显示在搜索结果中。

### 3.5 流程图

#### 3.5.1 整体流程图

系统的整体流程图如下所示，各部分流程图分别说明各步骤更细致的流程。

#### 3.5.2 各部分流程图

各部分流程图包括开始部分、文件管理、编码译码、搜索替换、词频统计、相关搜索和文本编辑部分。

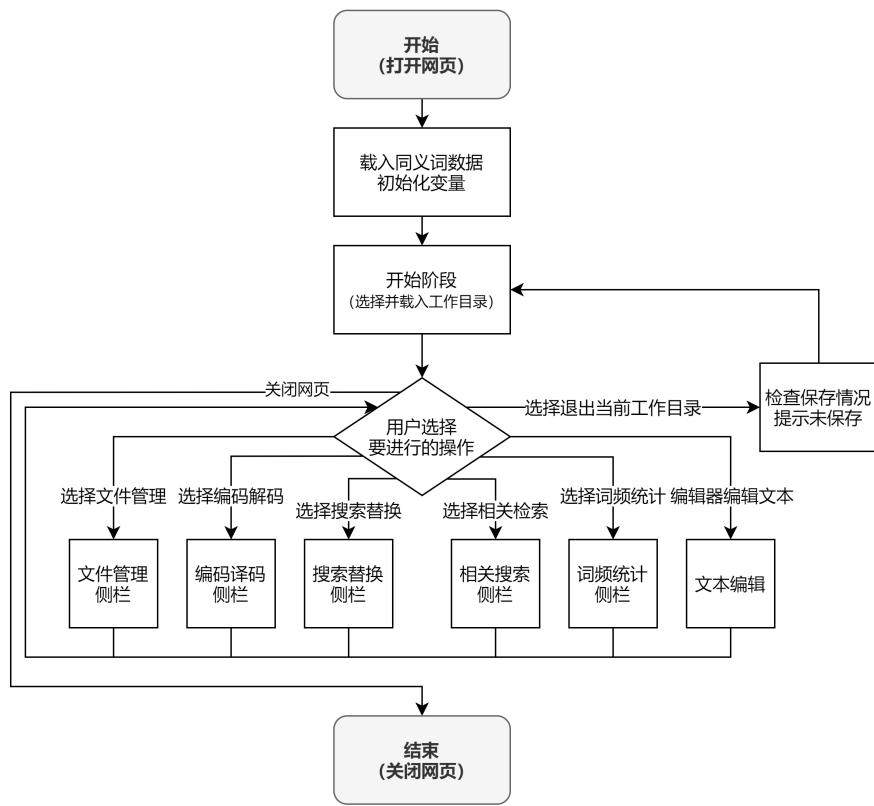


图 14: 整体流程图

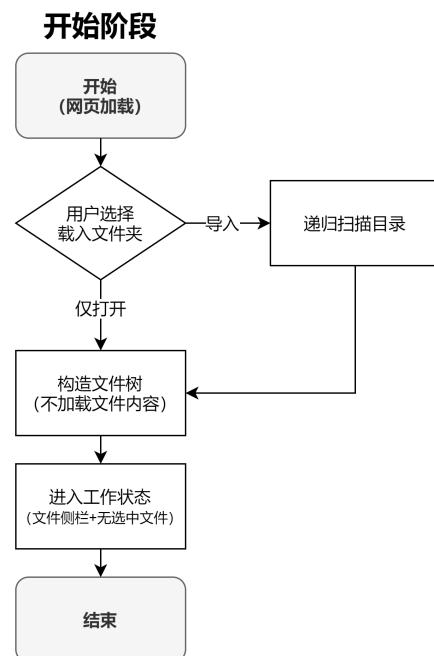


图 15: 开始阶段流程图

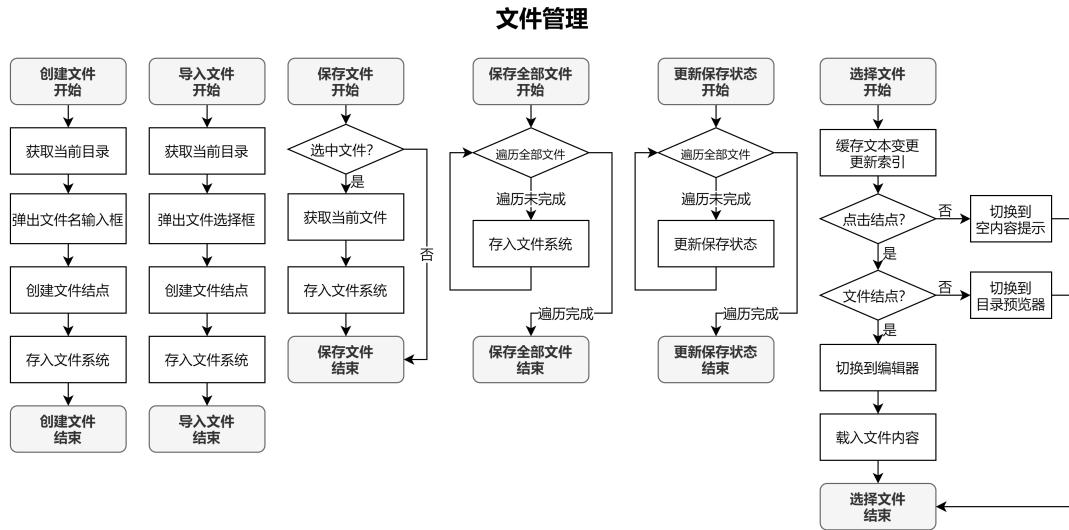


图 16: 文件管理流程图

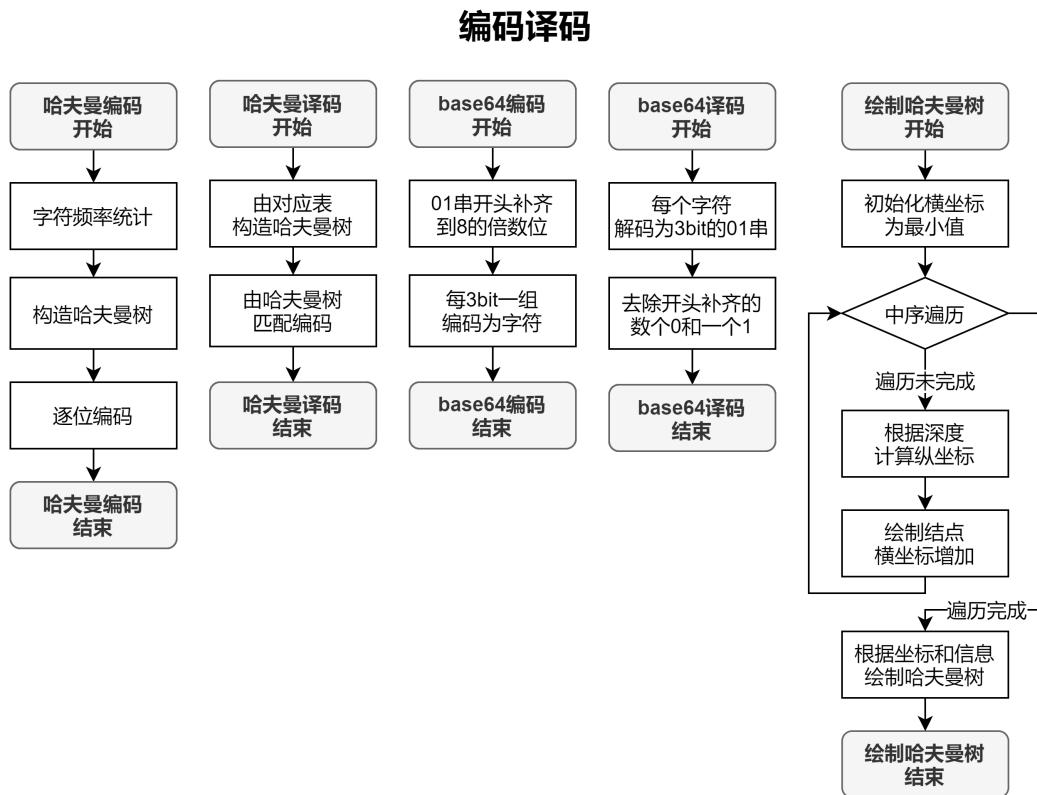


图 17: 编码译码流程图

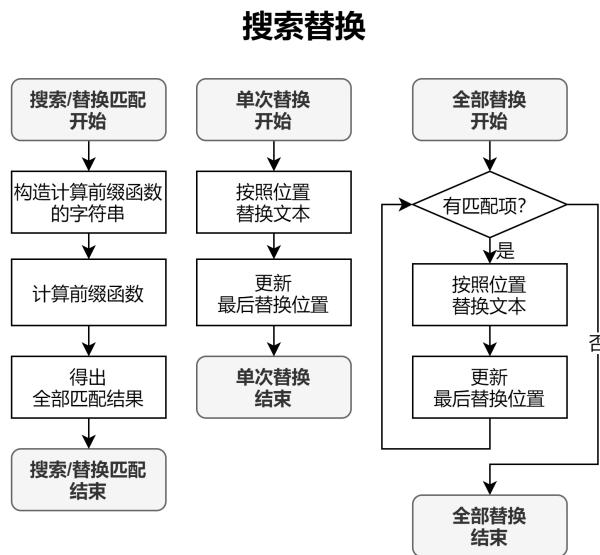


图 18: 搜索替换流程图

## 相关搜索

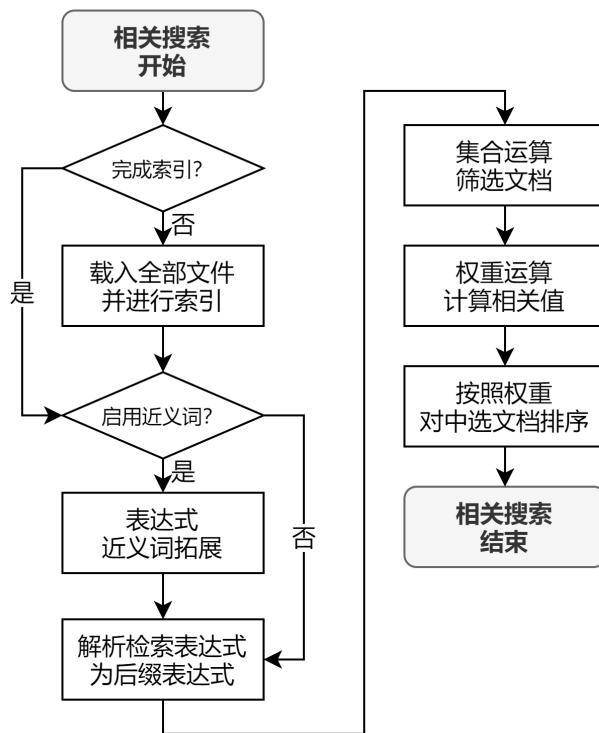


图 19: 相关搜索流程图

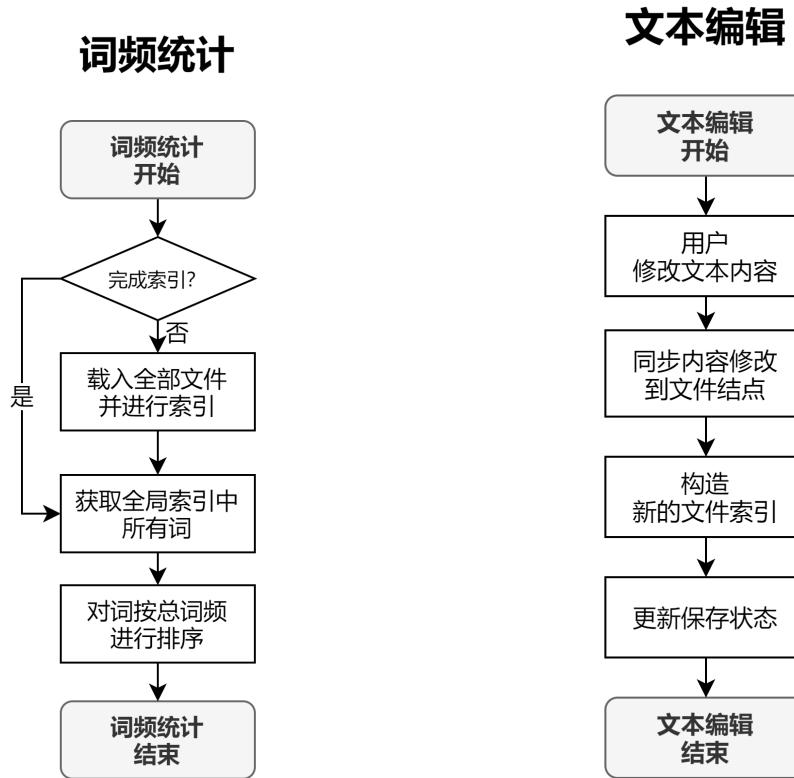


图 20: 词频统计流程图

图 21: 文本编辑流程图

## 4 测试

### 4.1 功能性测试

#### 4.1.1 普通测试用例

使用普通测试用例对各侧栏进行功能性测试，均工作正常。由于前面“系统整体结构以及各模块的功能描述”中详细描述了各模块的功能，这里不再赘述。

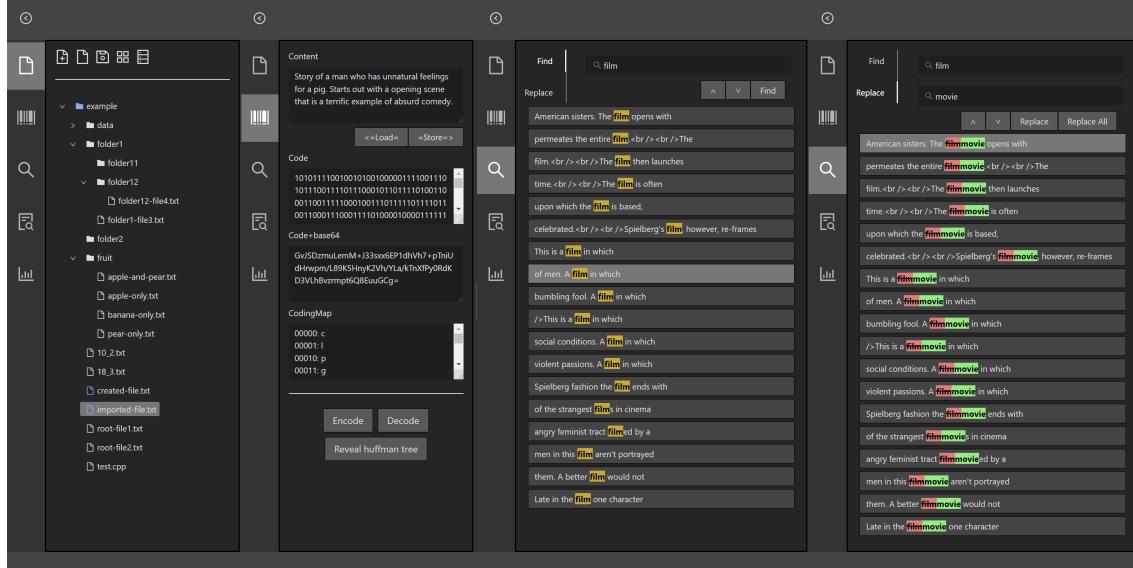


图 22: 前三个侧栏的测试截图

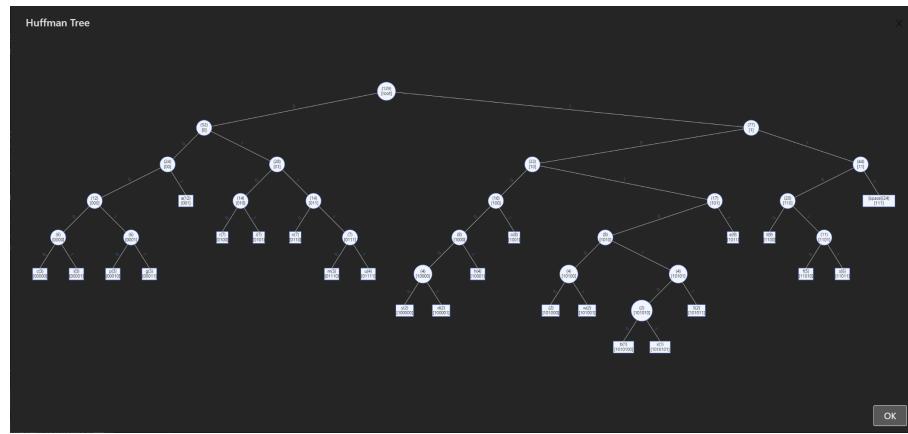


图 23: 编码译码侧栏中展示的哈夫曼树

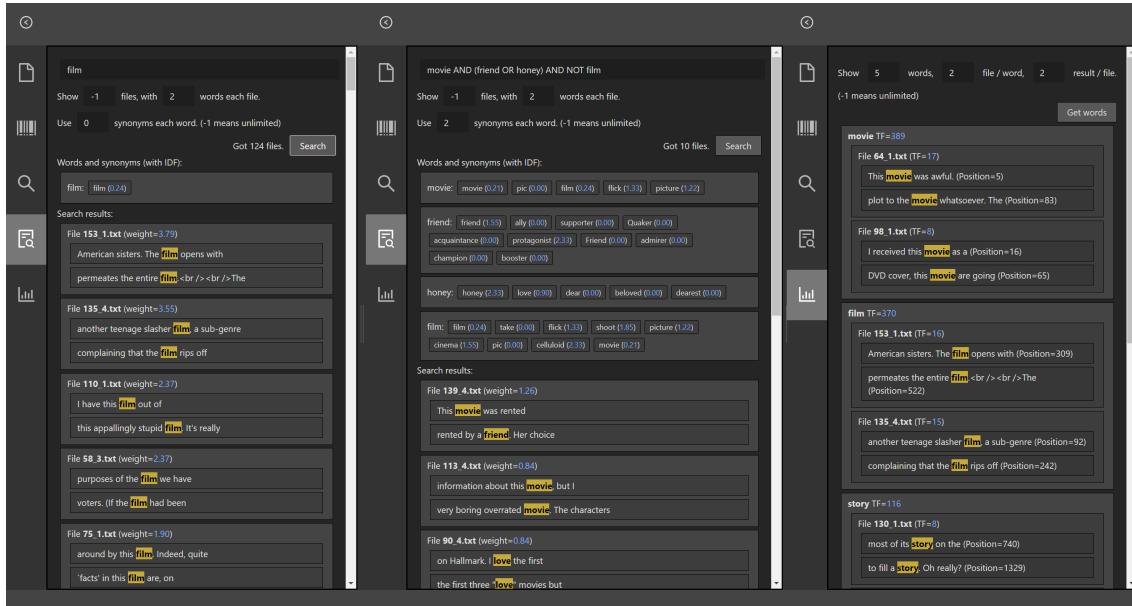


图 24: 后两个侧栏的测试截图

#### 4.1.2 导致程序报错的用例

此部分展示系统对特殊输入的报错和警告。

**编码、搜索空内容报错** 在编码译码侧栏中，若用户尝试编码空内容，会在页面上方提示报错。同样地，在搜索替换侧栏中搜索空内容也会产生类似的报错。

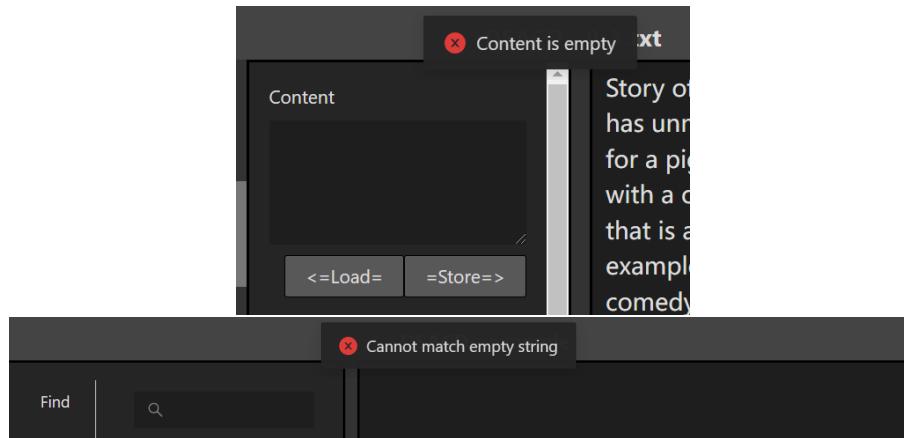


图 25: 编码空内容报错

**保存提醒** 在用户未对所有文件保存就要退出当前文件夹时，会弹出保存提醒框。用户可以选择保存所有文件、不保存，或者返回编辑界面继续修改。

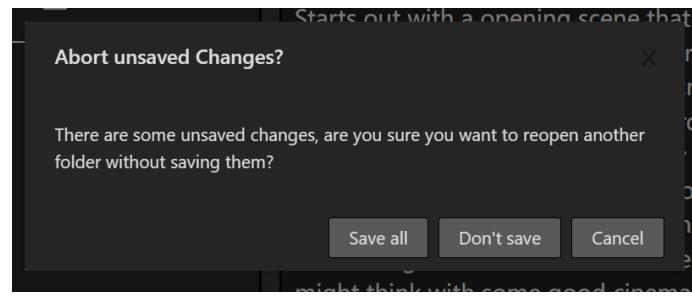


图 26: 保存提醒

**搜索无匹配** 当用户搜索的内容没有匹配项时，程序会在页面弹出警告，说明本次搜索未匹配任何内容。

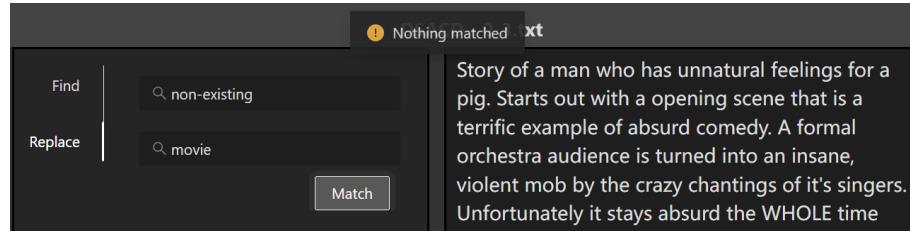


图 27: 搜索无匹配

#### 4.1.3 边界数据测试

**相关性检索空文件夹** 我尝试让系统检索空文件夹，结果与预想的一致，没有任何检索结果。

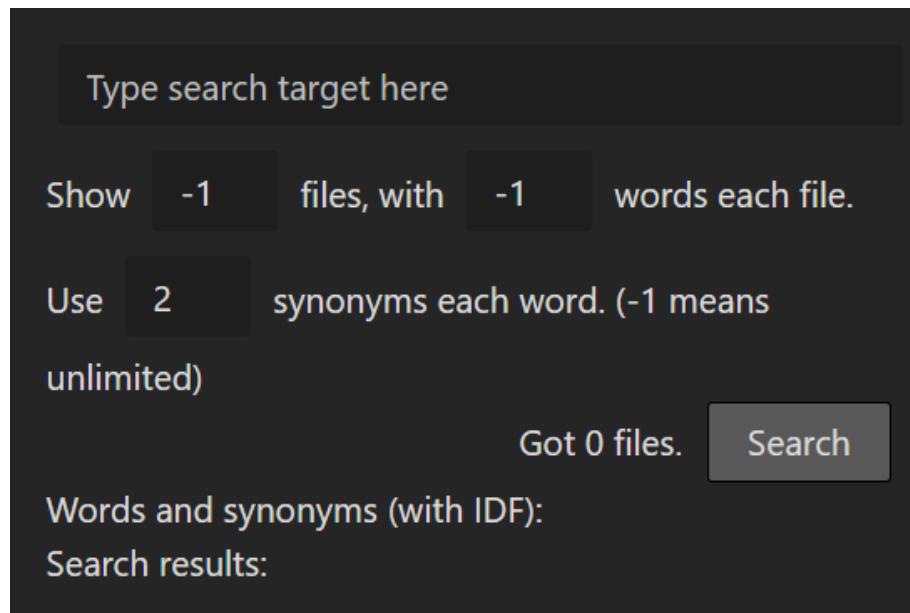


图 28: 搜索无匹配

此外，许多边界测试，如搜索空内容、编码空内容，都在前面的部分进行了测试，系统会针对这些边界情况进行特殊的处理和报错。这些测试同样属于边界数据测试，但在此不再重复说明。

## 4.2 非功能性测试

测试对提高的文档进行索引，全部完成共耗时约 5.6s。考虑到索引过程中也可以先对已完成索引部分先进行检索，程序运行效率可以接受。

此外，程序在载入和检索文件时，页面上方会弹出提示，向用户提示此过程的进行。

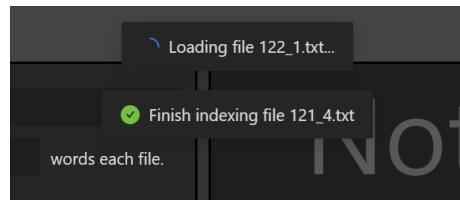


图 29: 载入检索文件

## 5 总结与提高

### 5.1 亲手实现基础数据结构的经验

在通常的开发过程中，亲自去实现基础数据结构是比较罕见的。在面向工程的场景下，我们通常可以使用可靠的标准库，还能使用充分验证、功能丰富的包；而在面向竞赛的场景下，我们几乎一定会使用 STL 等标准库以节约开发和运行的时间。

因此，亲手去开发如哈希表、对应表、堆等基础的数据结构是十分难得的。亲自实现这些基础数据结构让我对它们加深了理解，在将来的使用中也能更多地理解它们内部的原理，进而有助于进行调试、优化等工作。

### 5.2 分层次、分模块地构建一个多功能程序

之前在开发 EduOJ 等程序时，也离不开分层次、分模块地对程序进行构建，但此课设中，这一点体现得更加透彻。

**层次** 课设分为三个层次，从低到高分别是：自己实现的基础数据结构与算法、面向应用的组合数据结构、业务逻辑与交互界面。

例如，对于相关性检索功能。最底层的是基础的中缀表达式转后缀、后缀表达式求值；中间的是面向相关性搜索构建的集合运算、权重运算；最上层的是与用户交互的界面、对检索结果的排序和分层次展示。

**模块** 课设在图形化界面、数据结构上都分为了多个模块。图形化界面中每个 Vue 的 Component 就是一个模块，每种数据结构是一个模块。

**“分治”的开发过程** 经过对一个大系统分层次、分模块的划分，我得以从低到高，逐个模块地对系统进行开发。每次开发一个层次中的一个模块，这有助于我全身心地思考模块的设计，也易于对各个模块进行测试。

### 5.3 鸣谢

在此，十分感谢本课设的指导老师，也是教授我们数据结构课设与算法的老师。

上学期的数据结构与算法课程中，杜老师对数据结构和算法的讲解十分清晰明了，本课设中大部分的数据结构均按照课堂讲授的思路实现。

此外，在课设教学中，杜老师从原型设计到验收检查都为我提供了明确的指导，对此课设有很大的帮助。