实验四报告-202408040228-符航康

词频统计

一、问题分析

1. **要处理的对象（数据）是什么**

   本次实验目标是对一个英文文本文件中的单词进行**频率统计**——需要处理的是文本文件中的每个单词，并统计其出现次数。本问题中，单词定义为由字母组成的字符序列，且需要将大写字母转为小写字母进行统一统计；而非字母字符则被视为分隔符（例如空格、标点符号等）。

   为何选择使用这种数据结构?

   程序使用了 `unordered_map` 来存储每个单词及其对应的频率，其查询和插入操作具有**常数时间复杂度[O(1)]**，适合用于频率统计，能够高效地存储和处理大量数据。

2. 程序需要实现以下**功能**:

   从文件中**读取**英文文本。

   统计每个单词的出现频率，并按照频率从高到低输出。

   处理文件中的**非字母**字符，将其视为单词分隔符。

   输出前100个出现频率最高的单词及其出现次数，若单词数量不足100个，则输出实际的数量。

3. **处理后的结果**按照以下规则排序:

   先按频率从高到低排序；若频率相同，则按字典顺序排列。

4. **样例求解**

   输入

   输入

   ```
   I will give you some advice about life.
   Eat more roughage;
   Do more than others expect you to do and do it pains;
   Remember what life tells you;
   do not take to heart everything you hear.
   do not spend all that you have.
   ```

```
7 do not sleep as long as you want.
```

执行步骤

1. 程序读取文件并将文本转换为小写字母。

2.使用正则表达式从每行文本中提取单词。

3.统计每个单词出现的频率，并存储在 `unordered_map` 中。

4.将统计结果转换为 `vector` 并根据频率排序，若频率相同，则按字典序排序。

5.输出前100个最频繁的单词及其频率。

输出

输出

```
1 do 6
2 you 6
3 not 3
4 as 2
5 life 2
6 more 2
7 ...
8 /*
9 排序优先级：
10 主排序：词频降序
11 次排序：字典序升序
12 */
```

二、数据结构和算法设计

1. **抽象数据类型设计**

使用 `unordered_map` 来存储单词及其出现的频率；每个键值对表示一个单词及其频率。`unordered_map` 提供了常数时间查找和插入操作，适合频繁访问和修改这些值。

2. **物理数据对象设计**

`unordered_map<string, int>`：存储单词及其频率。

`vector<pair<string, int>>`：存储所有单词及频率的列表，并按频率排序。

3. **算法思想设计**

通过**正则表达式**从每一行文本中提取出所有有效的单词，忽略其他非字母字符。

> 使用 `unordered_map` 统计每个单词的频率。
>
> 将 `unordered_map` 中的数据转换为 `vector`，然后按照频率进行排序。若频率相同，按字典顺序排序。

4. **关键功能的算法步骤**

> **读取文件并处理文本**：逐行读取文件内容，将每行转换为小写字母。
>
> **提取单词并统计频率**：使用正则表达式 `[a-zA-Z]+` 提取所有由字母组成的单词，将每个单词的出现次数增加。
>
> **排序并输出**：将 `unordered_map` 中的单词及其频率转换为 `vector`，然后按照频率降序排序，频率相同的单词按字典序升序排序。最后输出前100个最频繁的单词及其频率。

三、算法性能分析

1. **时间复杂度**：

> 读取文件并逐行处理文本：O(n)，n 是总字符数。
>
> 使用正则表达式提取单词：对行进行处理的时间复杂度为 O(n)，n为行数。对每个单词的匹配是 O(m)，其中 m 是行的长度。
>
> 插入到 `unordered_map`：每次插入操作的时间复杂度是 O(1)，插入操作的总时间复杂度是 O(m)，m 是单词的个数。
>
> 排序：对所有单词及其频率进行排序的时间复杂度为 O(m log m)， m 是文件中单词的总数。

2. **空间复杂度**：

> `unordered_map` 存储每个单词及其频率，占 O(n) 空间。
>
> `vector` 存储所有单词及其频率，占 O(n) 空间。

算法的总体时间复杂度为 O(n + m log m)，空间复杂度为 O(m)。

四、实验总结与收获

本实验通过实现一个基于 `unordered_map` 和正则表达式的词频统计程序，我熟悉了如何高效地处理和分析文本数据，以及如何利用 C++ 标准库中的容器（如 `unordered_map` 和 `vector`）以及算法（如 `sort`）来解决实际问题；并让我更加深刻理解了正则表达式在文本处理中的应用——提取符合特定规则的单词。

附完整实验代码：

完整实验代码

```cpp
#include <iostream>
#include <fstream>
#include <unordered_map>
#include <vector>
#include <string>
#include <algorithm>
#include <regex>

using namespace std;

int main() {
    ifstream ifs("in.txt");  // 打开文件
    if (!ifs.is_open()) {
        cerr << "文件打开失败。" << endl;
        return -1;
    }

    unordered_map<string, int> wordFreq;  // 存储单词及其频率
    string line;
    regex wordRegex("[a-zA-Z]+");  // 匹配单词的正则表达式

    while (getline(ifs, line)) {
        transform(line.begin(), line.end(), line.begin(), ::tolower);  // 将整个行转换为小写字母
        auto words_begin = sregex_iterator(line.begin(), line.end(), wordRegex);
        auto words_end = sregex_iterator();

        for (sregex_iterator i = words_begin; i != words_end; ++i) {
            string word = (*i).str();
            wordFreq[word]++;  // 统计单词出现的频率
        }
    }

    ifs.close();  // 关闭文件

    // 将unordered_map转换为vector，并按频率排序
    vector<pair<string, int>> sortedWords;
    for (const auto &entry : wordFreq) {
        sortedWords.push_back(entry);
    }

    sort(sortedWords.begin(), sortedWords.end(), [](const pair<string, int>& a, const pair<string, int>& b) {
        if (a.second != b.second)
            return a.second > b.second;  // 按频率降序排列
```

```
        else
            return a.first < b.first;  // 频率相同按字典顺序升序排列
    });

    for (int i = 0; i < min(100, (int)sortedWords.size()); ++i) {
        cout << sortedWords[i].first << " " << sortedWords[i].second << endl;
    }

    return 0;
}

```