DocAnalysis

环境

在`Windows 10`系统下，使用`Visual Studio 2019`编译运行的`C++`控制台程序。

任务分析

根据大作业要求，主要有以下两个任务需要完成：

1. 统计两个文档多少字符相同，多少字符不相同
2. 统计前十高频字或词

第一个任务比较简单，我们只需要记录第一个文档中各字符出现的次数，再和第二个文档进行比较即可。第二个任务我认为相对来说比较复杂。对于一篇汉语文档，我们并不能像对一篇英语文档那样通过标点和空格识别出词语，这就是摆在我面前的第一个难题——**分词**。同时，处理一篇汉语文档，势必要特别注意**编码**，这是我需要处理的第二个难题。

编码

现在比较常用的中文编码是`UTF-8`和`GBK`。

1. `UTF-8`编码，在`Linux`系统下很常见，采用1~4的变长字节，字符集全，是当下各领域比较通用、主流的编码；
2. `GBK`编码，是`Windows`系统下的默认编码，`Windows`下的文件路径、控制台等都默认采用这一编码，采用1~2的变长字节，涵盖绝大部分汉字。

除此之外，在`C++`环境下还支持`UTF-16`字符的相关操作，宽字符类型`wchar_t`的基本操作便是基于`UTF-16`编码的。但是这种编码不常使用，因此被我排除在外。

综合考量，**我选择了`GBK`编码作为本项目的统一编码。**本项目下所有代码源文件、`txt`文件均采用这一编码标准。选择这一编码的原因主要有以下几点：

1. 输入和输出较`UTF-8`编码方便，读入一个字符，判断最高位为1时(表示该字符为非`ASCII`字符)，我们只需要再读入一个字符便得到了该汉字的编码；而对于`UTF-8`编码，我们需要统计高位在0出现之前连续1的个数，以此决定再读入几个字符得到这个汉字的编码，输出同理；
2. 无论是`x86`还是`x64`，`MSVC`编译器对`wchar_t`类型的定义都是2字节的`unsigned short`类型，`GBK`由于最多只有两个字节可以方便地用`wchar_t`类型表示，而`UTF-8`则需要4个字节的`unsigned int`类型表示，这会让程序所占内存空间提高一倍；
3. 由于文件路径在`Windows`环境下采用`GBK`编码表示，如果我们统一采用`UTF-8`编码，当文件路径中有中文时可能会出现BUG，而我们直接采用`GBK`编码就可以避免这一问题。

在读入`GBK`字符时，核心操作如下：

```
wchar_t WInFile::convertUCharToWChar(unsigned char c)
{
    wchar_t wc = 0;
    if (c & 0x80) //首位为1
    {
        wc |= static_cast<wchar_t>(c) << 8;
        c = inFile->get(); //再读入一个字符作为完整汉字
        wc |= static_cast<wchar_t>(c);
    }
```

```
    else
    {
        wc = static_cast<wchar_t>(c) << 8;
    }
    return wc;
}
```

这个函数完成了将一个`unsigned char`类型字符转换为`wchar_t`类型字符，需要注意的是，`char`类型是有符号的，最高位为符号位，因此最高位不能直接进行诸如移位、按位于、按位或等位运算，需要使用`unsigned char`类型。

转换DOC文件到TXT

讨论完编码问题后，接踵而至的问题便是如何使用`C++`分析`DOC`文件。`DOC`文件的文件格式很复杂，对于`.docx`扩展名类型的文件，我们可以更改扩展名为`.zip`然后利用`C++`的`zlib`库将其解压成一个`xml`文件组成的文件夹，然后提取内容，但是对于`.doc`扩展名我没有找到合适的方法。最后，我决定先将`DOC`文件转换为`TXT`文件，然后再进行分析。因此，我写了一个`Windows PowerShell`脚本完成这一转换操作，位于`DocAnalysis\tools\doc2txt.ps1`。

分词

查阅相关资料，现在主要有以下几种分词方法：

1. 字符匹配，也叫做机械分词法，将待分析的汉字串与一个“充分大的”机器词典中的词条进行匹配；
2. 理解法，通过让计算机模拟人对句子的理解，达到识别词的效果，涉及句法、语义分析；
3. 统计法，涉及机器学习。

我挑选了其中最容易实现的机械分词法。

首先，我需要一个足够大的词典作为支撑。在github上我找到了《现代汉语词典（第七版）》XDHYCD7th，通过简单的字符串处理，我得到了一个只含词头的词典文件，在本项目中的位置是`DocAnalysis\dict\dict.txt`，该文件的每一行都是一个词。

接着，自然想到利用**串的模式匹配**把词典中每个词逐个与目标字符串作匹配。但是，在计算了时间复杂度后，我发现事情并没有这么简单。即使我们采用`KMP`算法，对于单个词条匹配的时间复杂度是`O(n + m)`，n是文档的字数，m是单词的字数，由于词典中大部分都是单双音节词，`m`是远小于`n`的，单次匹配的时间复杂度可以近似看成`O(n)`。词典中的词条的数量级为1e6，所以做完所有匹配的时间复杂度是`O(1e6 * n)`。考虑到一般文章的长度从几百字到几万字不等，我们这里假设`n`的数量级为`1e5`，所得**时间复杂度达到了**`O(1e11)`，而现代计算机1秒可以处理的时间复杂度数量级一半在`O(1e7)~O(1e8)`，显然，**采用直接模式匹配的方法并不可行。**

那么，问题出在哪里了呢？试想，对于文档字符串中的字符“一”，从这个字符开始，它只用匹配“一”开头的所有词就行了，而在上述算法中它和非“一”开头的词进行了大量无用的匹配。那么如何避免这种无用的匹配呢？自然想到使用`Tire树`（字典树）。

以前，我只写过英文的`Tire树`，通常一个`Tire树`节点的定义如下：

```
struct TrieTreeNode
{
    char c; //当前节点所存字符
    bool isEnd; //是否是单词的结尾
    TrieTreeNode *next[26]; //next[c]表示下个字符的地址
};
```

根据需要，如果区分大小写，可能指针数组`next`要开到52的大小。然而，这对于汉字来说仍然远远不够。对于`GBK`编码的汉字，我使用两字节的`wchar_t`来存储，难道我们要开一个2^16长度的`next`数组吗？这样的解决策略显然不行，后面的结点用不了这么大的空间，我们也没有这么充裕的空间。难道我们要采用变长数组吗？考虑到查询的时间复杂度，也不是一种高效的解决方案。如果**既要求`next`大小动态变化，又要求能够在较低的时间复杂度内完成查询，自然想到采用红黑树来存储`next`**。这样，插入和查找都可以在`log`的时间复杂度下完成，且可以动态变化。分析至此，我们的树节点应定义如下：

```
template<typename T>
struct TrieTreeNode
{
    T c; //当前节点所存字符
    bool isEnd; //是否是单词的结尾
    std::map<T, TrieTreeNode<T>*> next; //next[c]表示下个字符对应树节点的地址
    int freq; //单词的频数
    TrieTreeNode(T c_, bool isEnd_) : c(c_), isEnd(isEnd_), freq(0) {};
};
```

`C++`的`map`类型采用红黑树实现，为了方便以后在别的项目中使用，这里并没有指定字符`c`的类型为`wchar_t`，而是采用了`C++`的模板`template`完成对这一数据结构的定义。

由于词典中的字大约在1e5数量级，分支最多的根节点一次查询的时间复杂度为`O(log(1e5))`，而词典中的词大多为2~4字，最多不超过10字，且其后节点分支的数量级一般在100左右，这里可以近似将从某个字符开始在字典树中匹配的时间复杂度视为`O(k)`，k的数量级为`O(log(1e5))`，总时间复杂度为`O(k * n)`，这个时间复杂度我们可以在1s左右处理百万字级别的文章。

停止词

统计词频之后，我们会发现，诸如“你”、“我”、“这个”、“或者”、“然后”等**对理解文章内容没有意义的词**大量出现，这些词称为停止词(stopword)，统计词频时需要将这类词去掉。在github上的stopwords仓库中有已整理好的停止词，我选择了百度的停止词典，在本项目中的位置是`DocAnalysis\dict\baidu_stopwords.txt`。程序开始，我们不止给分词词典建立字典树，同样给停止词典建立一个字典树，在统计完词频输出结果时，如果对应词条可以与停止词典字典树中的词条匹配，那么就忽略这一词条。

相同和不相同字符数统计

统计字符数，我们需要建立一张表，可以根据汉字找到其对应出现的次数，可以有三种方式实现：

1. 直接使用静态的顺序表，以汉字字符的编码直接作为下标进行索引，这样我们需要建立一个包含13万左右元素的`int`类型的数组，使用起来比较方便，但是有大量空间未利用。
2. 使用哈希表建立起汉字编码 -> 数组下标的映射关系，我们知道`GBK`的汉字编码绝大多数集中在一个区间内，只需找到合适的哈希映射函数就可以节省大量空间。
3. 使用`C++`标准库提供的`std::map`（红黑树）存储汉字编码 -> 汉字频数这个关系。

其实这里直接使用第一种方法经计算也只占用0.5MB左右的空间，但是如果使用`UTF-8`编码而不是`GBK`编码空间就会不够用了。这里我采用了第三种方法来存储汉字对应的字频。

统计字频时，我还进行了另外的过滤操作，我认为统计像标点符号、换行符、空格符这样的字符没有意义，因此统计时将这些字符排除在外：

```
bool DocAnalysis::isWCharValid(wchar_t wc) //判断是否为有效字符
{
    if (!(wc & 0x00ff)) //不是汉字
    {
        char c = wc >> 8;
        //下面的判断表示wc是除字母、数字外的无效字符
        if (!((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c
<= '9')))
        {
            return false;
        }
    }
    return true;
}
```

文件结构

下面展示的是`DocAnalysis/DocAnalysis`目录下的文件结构：

```
.
+-- main.cpp // 包含main函数定义，程序入口

// DocAnalysis类包含了程序运行的主逻辑
+-- DocAnalysis.h
+-- DocAnalysis.cpp

// WInFile类完成了基本的从文件中读取GBK编码字符并存储为宽字符类型的操作
+-- WInFile.h
+-- WInFile.cpp

// WOutFile类完成了输出GBK编码宽字符到文件的操作
+-- WOutFile.h
+-- WOutFile.cpp

// TrieTree类定义了字典树这一数据结构及其基本操作，由于template类的声明与实现必须放在同一个文
件当中，所以这里采用.hpp文件类型
+-- TrieTree.hpp

+-- tools // tools文件夹存储相关工具
|   +-- doc2txt.ps1 // Windows PowerShell命令行脚本，用于完成.doc或.docx文件转换为GBK
编码的txt文件的操作

+-- dict // dict文件夹存储相关字典
|   +-- baidu_stopwords.txt // 百度停止词词典
|   +-- dict.txt // 现代汉语词典

+-- 资本论 //输入文件1，用于Release下测试的长文档
|   +-- 资本论.docx
|   +-- 资本论.txt //1.docx转换之后的txt文件
|   +-- 资本论_result.txt //输出结果文件，包含词频和字频

+-- 共产党宣言 //输入文件2，用于Release下测试的长文档
|   +-- 共产党宣言.docx
|   +-- 共产党宣言.txt
|   +-- 共产党宣言_result.txt
```

```
+-- 1 //输入文件1，用于Debug下测试的短文档
|   +-- 1.docx
|   +-- 1.txt //1.docx转换之后的txt文件
|   +-- 1_result.txt //输出结果文件，包含词频和字频

+-- 2 //输入文件2，用于Debug下测试的短文档
|   +-- 2.docx
|   +-- 2.txt
|   +-- 2_result.txt
```

输出结果

对于词频和字频，由于要统计出前十，这里我采用了优先队列来存储频数表，优先队列类型如下：

```
typedef std::priority_queue<std::pair<int, std::vector<T>>> qtype;
```

使用了`C++`标准库的`vector`，`pair`和`priority_queue`。在统计完词频后遍历一遍字典树，将频数大于0的词条以`std::pair<int, std::vector<T>>`(这里`T`的类型是`wchar_t`)存储在优先队列中，`std::pair`的第一个`int`类型关键字为频数，第二个关键字是`wchar_t`类型字符的数组，表示一个词，采用`std::pair`的原因是其自动按照第一关键字、第二关键字的顺序比较大小，因此无需重载比较运算符。最后，依次`pop`出优先队列的元素即可得到前十词频和字频。

对`资本论.docx`的分析如下：

> 前十高频词:
> 资本 18273次
> 生产 12384次
> 价值 10518次
> 劳动 9528次
> 商品 6728次
> 货币 5590次
> 工人 4360次
> 形式 4011次
> 剩余 3482次
> 产品 3332次
>
> 前十高频字:
> 资 23366次
> 产 18583次
> 不 16957次
> 生 16681次
> 价 14592次
> 动 12319次
> 人 11920次
> 工 11324次
> 值 10841次
> 品 10783次

结果存储在`资本论\资本论_result.txt`当中。对于字符相等与不相等数目的比较则直接显示在控制台窗口中。

根据`VS`的代码分析，本程序大部分时间都花在`doc`文件与`txt`文件的转换上（大约需要5-7秒钟）。若是直接分析`txt`文件，**在`Release`模式下程序仅需0.9秒左右即可完成所有操作。**如果要直接分析`txt`文件，可以将对应`txt`文件放在对应文件夹中，在主程序执行以下语句：

```
DocAnalysis::useTxtFileDirectly(); //直接使用txt文件请执行本函数，否则将本行代码注释掉
```

如果要使用别的文件名，比如`foo.docx`，注意三个地方：

1. 在`DocAnalysis`文件夹下创建对应文件夹（文件夹名为`foo`）
2. 对应文件夹下放对应`docx`文件（`foo`文件夹下放`foo.docx`，如果是`txt`文件那么放`foo.txt`）
3. 在`main.cpp`中，更改对应`inFileName`如下：

```
const char* inFileAName = "foo";
```

如果要使用`.doc`的后缀名，注意在主函数运行语句：

```
DocAnalysis::changeDocExtension(".doc"); //设置文件后缀名，如果为.doc请执行本函数
```

问题与不足

由于我统计词频时分词的策略是遇到一个在字典树中的词就将这个词的词频加1，所以如果原文当中多次出现了词条`中国特色社会主义`，我会将词条`中国`，`特色`，`社会`，`主义`，`社会主义`，`中国特色社会主义`的词频都加一，然而这里显然只需要将词条`中国特色社会主义`的词频加一即可。除了这种策略外，还有正向最大匹配、反向最大匹配的策略，这些策略可能一定程度上避免这种问题，但是有时也会产生歧义。若想实现更加准确的分词与词频统计，可能需要借助人工智能与机器学习中`NLP`领域的相关方法。