DS-PJ开发文档

> 20岁,是学生.

1. 代码结构概要说明

项目文件结构

- include
  - huf.h `//声明HufTool类`
- huf.cpp `//实现HufTool类的成员函数`
- main.cpp `//处理参数并调用相应功能`
- CMakeLists.txt

HufTool类声明

```
`class HufTool
{
private:
    struct HTNode // 哈夫曼树节点
    {
        int weight;
        int parent, lchild, rchild;
        int length;   // 编码长度
        int *hufCode; // 指向存放了长度为length的编码的区域
    };
    struct DirNode // 目录节点
    {
        string pathStr;
        unsigned char len; // 名字所占字节数
        unsigned char dep; // 深度
        bool isFile = false;
    };

    int nodeCnt[N];
    HTNode HT[MAX];          // HT是huffmanTree的缩写
    int root = N;            // 用于记录根节点
    char cover;              // 补位数
    vector<DirNode> dirVec; // 目录容器

    void find2Min(int num, int &m1, int &m2); // 找到权重最小的两个根节点

    void makeHT();                                // 初始化以及造树
    void encode();                                // 编码
    void nodeCount(string input);                 // 统计出现次数
    void writeHT(string input, ofstream &ofs); // 写入文件
    long long getLength(ifstream &ifs);           // 读取并计算返回文件长度
    void fileHuf(string input, ofstream &ofs); // 压缩文件

    void remakeHT(ifstream &ifs);                 // 重新造树
    int rwName(ifstream &ifs, string &output); // 读写文件名并返回名字长度
    void fileDehuf(ifstream &ifs);                // 解压缩文件

    void recursion(filesystem::path filePath, unsigned char &depth); // 递归取得目录信息
    void writeDir(ofstream &ofs);                                    // 写入目录
    void writeFiles(ofstream &ofs);                                  // 写入压缩后文件
    void folderHuf(filesystem::path filePath, ofstream &ofs);        // 文件夹压缩

    void readDir(ifstream &ifs);     // 读取目录
    void folderDehuf(ifstream &ifs); // 文件夹解压缩

    bool fileOverwrite(string str); // 文件覆写判断

public:
    void huf(string input, string output); // 压缩方法
    void dehuf(string input);              // 解压缩方法
    void preview(string input);            // 压缩包预览
```

```
};
```

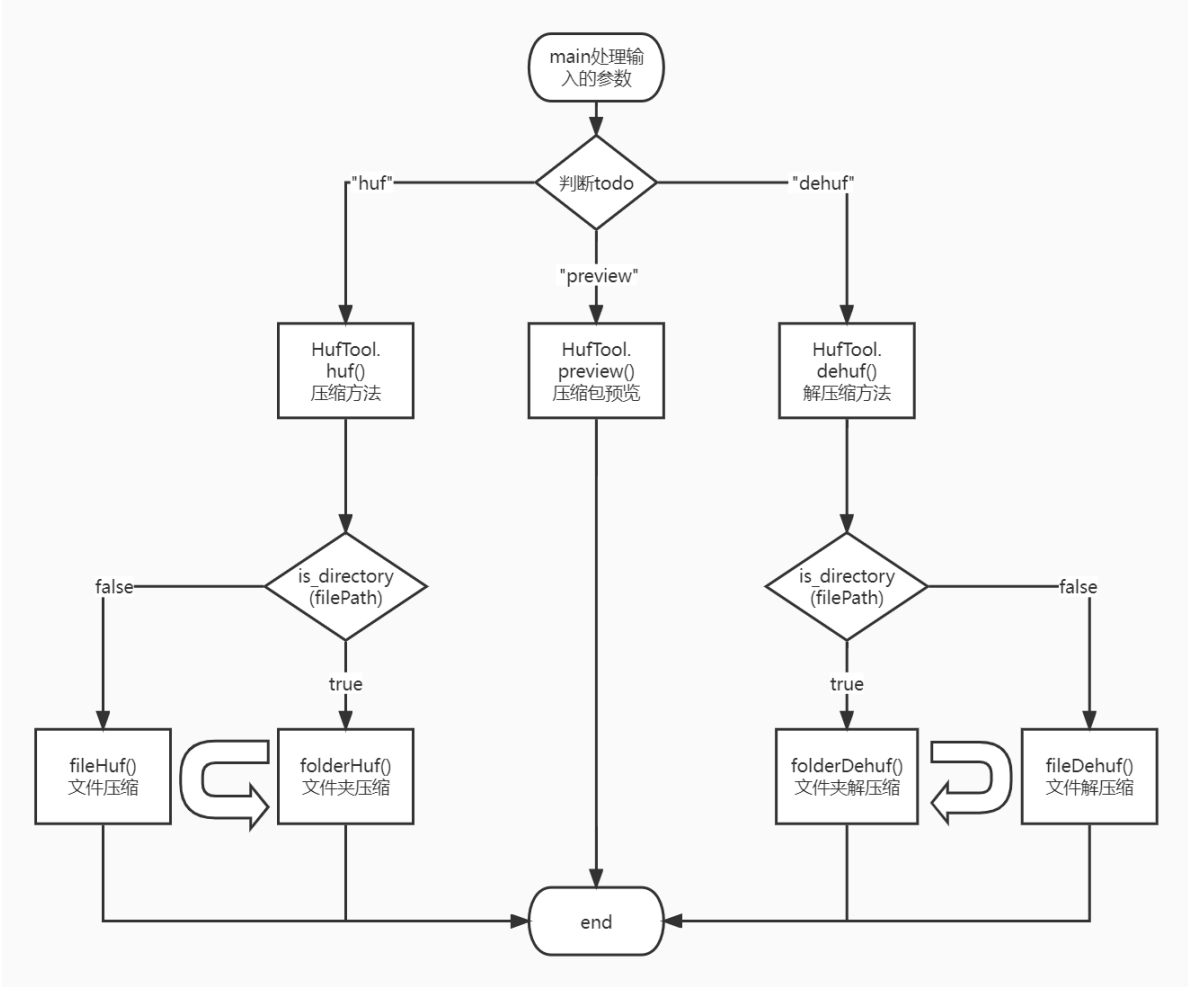
main.cpp

```
#include "huf.h"

int main(int argc, char *argv[])
{
    string input, output;
    string todo;
    HufTool hufTool;
    // 处理参数
    int i = 0;
    while (argv[1][i] != '\0')
    {
        todo.push_back(argv[1][i]);
        i++;
    }
    if (todo == "help")
    {
        cout << "the first argument(necessary) can be 'help' 'huf' 'dehuf' and 'preview'" << endl;
        cout << "the second argument(necessary for huf, dehuf and preview) is the name of input" << endl;
        cout << "the third argument(only optional for huf) is the name of output, if you forget to write suffix '.huf', I will do it for you" << endl;
        return 0;
    }
    i = 0;
    while (argv[2][i] != '\0')
    {
        input.push_back(argv[2][i]);
        i++;
    }
    i = 0;
    if (argv[3] != nullptr)
    {
        while (argv[3][i] != '\0')
        {
            output.push_back(argv[3][i]);
            i++;
        }
    }
    // 参数处理结束
    if (todo == "huf")
    {
        hufTool.huf(input, output); // 压缩方法
    }
    else if (todo == "dehuf")
    {
        hufTool.dehuf(input); // 解压缩方法
    }
    else if (todo == "preview")
    {
        hufTool.preview(input); // 压缩包预览
    }
    else
```

```
51      {
52          cout << "I don't know what you want to do." << endl;
53      }
54      return 0;
55  }
```

概要流程图



2. 需求设计与实现思路

{1} 核心需求

(1) 文件的压缩与解压

<1> 压缩

考虑到要能够压缩和解压不同格式文件,而他们都是01的组合,在二进制层面上没有什么区别;此外,文件以字节为最小单位存储,字节可以被char型变量完整接收,也符合文件流的各种操作,且一个字节只有256个可能的值;想到对文件以字节为单位进行哈夫曼编码,可以得到节点数确定的哈夫曼树(256 * 2 - 1 = 511).

要进行构造哈夫曼树,就需要提前统计词频;在构造完哈夫曼树后,就需要写入文件了,为了方便解压,就需要统计压缩文件的字节数存入开头(为了防止溢出使用long long型,占8个字节);考虑到哈夫曼编码会导致单个字符所占bit数不同,可能会导致压缩文件的最后一个字节没被填满而引发意想不到的问题,所以还要在开头存放补位数(占1个字节);为了在解压时能还原出哈夫曼编码,还要将哈夫曼树的信息存入开头,为了节约空间,我最终只存入了根节点id以及每个节点的父节点id(共占512个字节)并修改构造树的过程,使得任何左孩子都小于等于右孩子(这样只需要有父节点信息就能还原哈夫曼树了);此外,为了在解压时能够还原文件名,还要将文件名存入开头,但考虑到文件名长度不定,如果用固定字节数存储要么容易溢出,要么太占空

间,所以想到了在名字前存放名字长度的方式.而文件内容的写入就将读取的字节都替换成对应的哈夫曼编码,在凑齐一个字节后写入压缩文件中,最后缺少的bit全部用0补上.

*为了在处理文件夹压缩问题时不至于重写这部分代码,最好能直接调用已有的函数,就需要传入ofstream的引用而不是string output,并且要在最后将ofstream.put的位置移到文件末尾从而能够连续调用.

部分代码:

```
void HufTool::fileHuf(string input, ofstream &ofs) // 压缩文件
{
    nodeCount(input);// 统计出现次数
    makeHT();// 初始化以及造树
    encode();// 编码
    writeHT(input, ofs);// 写入文件
}

void HufTool::writeHT(string input, ofstream &ofs) // 写入文件
{
    ifstream ifs;
    char ch;
    char byte = 0;
    int cnt = 0;
    long long length = 0;
    ifs.open(input, ios_base::binary);
    for (int i = 0; i < 8; i++) // 留给之后写入压缩后文件内容的字节数
    {
        ofs.put(0);
    }
    ofs.put(0);            // 留给之后写入补位数
    ofs.put(root - PMOD); // 写入根节点id
    // 写入树
    for (int i = 0; i < N * 2 - 1; i++)
    {
        if (HT[i].parent != -1)
        {
            ofs.put(HT[i].parent - PMOD);
        }
        else
        {
            ofs.put(0x7F); // 0X7F代表无父节点的节点
        }
    }
    // 写入名字
    unsigned char nameLength = input.size();
    ofs.put(nameLength);
    for (int i = 0; i < nameLength; i++)
    {
        ofs.put(input[i]);
    }
    // 写入压缩后内容
    while (ifs.get(ch))
    {
        for (int i = 0; i < HT[(unsigned char)ch].length; i++)
        {
            if (cnt == 8)
            {
                ofs.put(byte);
                byte = 0;
```

```
                cnt = 0;
                length++;
            }
            byte = (byte << 1) | HT[(unsigned char)ch].hufCode[i];
            cnt++;
        }
    }
    // 补0以凑齐一个字节
    bool flag = false;
    if (cnt > 0)
    {
        for (int i = cnt; i < 8; i++)
        {
            byte = byte << 1;
        }
        ofs.put(byte);
        length++;
        ofs.seekp(-length - nameLength - 514, ios_base::cur);
        ofs.put(8 - cnt); // 把补0数写到前面
        flag = true;
    }
    //调整put的位置以适应文件夹压缩
    if (flag)
    {
        ofs.seekp(-9, ios_base::cur);
    }
    else
    {
        ofs.seekp(-length - nameLength - 522, ios_base::cur);
    }
    for (int i = 0; i < 8; i++) // 把length按小端存入开头
    {
        int temp = length % 256;
        if (temp > 127)
        {
            temp -= 256;
        }
        char ch = temp;
        ofs.put(ch);
        length /= 256;
    }
    ofs.seekp(0, ios_base::end);
    ifs.close();
}
```

<2> 解压

解压建立在压缩格式的基础之上,压缩时在开头存入了解压所需的信息,且除了名字都是定长的(名字也有长度信息);所以可以先读取这些信息,重建哈夫曼树,获取文件名,文件长度,补位数等,最终通过逐字节写入的方式即可还原压缩前的文件.

*为了在处理文件夹解压问题时不至于重写这部分代码,就需要传入ifstream的引用而不是string input

部分代码:

```
void HufTool::fileDehuf(ifstream &ifs) // 解压缩文件
{
    ofstream ofs;
```

```
    string output;
    long long length = getLength(ifs);    // 获取长度
    remakeHT(ifs);                        // 重新造树
    int nameLength = rwName(ifs, output); // 读写名字
    if (fileOverwrite(output))
    {
        ofs.open(output, ios_base::binary);
        if (!ofs)
        {
            ifs.close();
            exit(1);
        }
        int now, bit, cnt = 0;
        char byte;
        for (long long i = 0; i < length * 8 - cover;)
        {
            now = root;
            while (HT[now].lchild != -1)
            {
                if (cnt % 8 == 0)
                {
                    ifs.get(byte); // 每次读一个字节
                    cnt = 0;
                }
                bit = byte & 0x80; // 获取当前的比特位,也就是byte的首位
                if (bit == 0)
                {
                    now = HT[now].lchild;
                }
                else
                {
                    now = HT[now].rchild;
                }
                byte = byte << 1; // 字节左移一位，以便下一次获取1bit
                cnt++;
                i++;
            }
            if (now > 127) // 范围控制在-128到127之间,防止输入值超出char型的范围
            {
                now -= 256;
            }
            ofs.put(now);
        }
    }
    ofs.close();
}
```

(2) 文件夹的压缩和解压

<1> 压缩

为了在解压时能够还原出文件夹的目录结构,就需要存储文件夹的目录信息(包括相对路径,路径所占字节数,深度以及是文件或文件夹的标志),又因为文件夹的深度不确定,所以获取目录信息时需要递归.在存入目录信息之后,只需要通过调用文件压缩函数来分别存入每个文件的压缩后信息即可

部分代码:

```
```

```
void HufTool::folderHuf(filesystem::path filePath, ofstream &ofs) // 文件夹压缩
{
    unsigned char depth = 0;
    recursion(filePath, depth); // 递归取得目录信息
    writeDir(ofs);              // 写入目录
    writeFiles(ofs);            // 写入压缩后文件
}

void HufTool::recursion(filesystem::path filePath, unsigned char &depth) // 递归取得目录信息
{
    DirNode temp;
    temp.pathStr = filePath.string();
    temp.len = filePath.string().size();
    temp.dep = depth;
    dirVec.push_back(temp);
    for (const auto &entry : filesystem::directory_iterator(filePath))
    {
        if (entry.is_directory()) // 若是文件夹
        {
            filesystem::path subPath = entry;
            depth++;
            recursion(subPath, depth); // 递归
            depth--;
        }
        else // 若是文件
        {
            temp.pathStr = entry.path().string();
            temp.len = entry.path().string().size();
            temp.dep = depth + 1;
            temp.isFile = true;
            dirVec.push_back(temp);
        }
    }
}

void HufTool::writeFiles(ofstream &ofs) // 写入压缩后文件
{
    for (int i = 0; i < dirVec.size(); i++)
    {
        if (dirVec[i].isFile)
        {
            fileHuf(dirVec[i].pathStr, ofs); // 文件压缩
        }
    }
}
```

<2> 解压

由于压缩过程中的设计,解压较容易实现,只需从压缩文件读取目录信息并重建目录,然后遍历目录容器中的每一个元素,是文件夹类型.就创建目录,是文件类型,就调用文件解压函数.

部分代码:

```
void HufTool::folderDehuf(ifstream &ifs) // 文件夹解压缩
{
```

```
    readDir(ifs); // 读取目录
    for (int i = 0; i < dirVec.size(); i++)
    {
        filesystem::path filePath = dirVec[i].pathStr;
        if (dirVec[i].isFile)
        {
            fileDehuf(ifs); // 文件解压
        }
        else
        {
            filesystem::create_directory(filePath); // 创建目录
        }
    }
}
```

(3) 性能

使用带缓冲的输入输出减少在 IO 上的时间开销;提前设置一些变量存储,减少过程调用.

(4) 代码风格

> 你的程序应保持良好的面向对象风格，良好的代码风格、注释习惯，具备较强的可读性，并符合标准 命名规范。不宜出现过长的类或方法，过量的耦合，或是大篇幅的重复代码。

按照要求来写即可

{2} 其他需求

(1) 使用CLI与用戶交互

考虑到使用类似Linux 下 tar, zip 等工具的输入输出方式:例如`zip png.zip 1.png`可能需要修改环境变量,比较麻烦;所以想到直接利用int main(int argc, char *argv[])中的argv[]来获取参数.此外,我还设置了help参数来提供说明,只需要在pj文件夹build目录下执行`.\huf.exe help`即可.不过这样的副作用是debugger功能变得难以使用(至少我不会使用).

main.cpp中相关代码:

```
// 处理参数
    int i = 0;
    while (argv[1][i] != '\0')
    {
        todo.push_back(argv[1][i]);
        i++;
    }
    if (todo == "help")
    {
        cout << "the first argument(necessary) can be 'help' 'huf' 'dehuf' and 'preview'" << endl;
        cout << "the second argument(necessary for huf, dehuf and preview) is the name of input" << endl;
        cout << "the third argument(only optional for huf) is the name of output, if you forget to write suffix '.huf', I will do it for you" << endl;
        return 0;
    }
    i = 0;
    while (argv[2][i] != '\0')
    {
```

```
        input.push_back(argv[2][i]);
        i++;
    }
    i = 0;
    if (argv[3] != nullptr)
    {
        while (argv[3][i] != '\0')
        {
            output.push_back(argv[3][i]);
            i++;
        }
    }
    // 参数处理结束
```

(2) 检验压缩包来源是否是自己的压缩工具

该功能主要在dehuf()中实现,除了最基础的后缀名判断,我还添加了首位的标志字节的判断,即不占用多少时间,也能极大地降低判断错误率,具体细节已在代码注释中给出:

> // 所有huf文件开头都有占1个字节的标志,用来区分是文件压缩还是文件夹压缩,它的值只可能是0或1,可以应对绝大部分其他来源却有相同后缀的情况

相关代码:

```
void HufTool::dehuf(string input) // 解压缩方法
{
    string suffix;
    int size = input.size();
    for (int i = size - 4; i < size; i++)
    {
        suffix.push_back(input[i]);
    }
    // 判断后缀,后缀不同,肯定是不同来源
    if (suffix == ".huf")
    {
        ifstream ifs;
        ifs.open(input, ios_base::binary);
        if (!ifs)
        {
            exit(1);
        }
        char ch;
        ifs.get(ch);
        // 所有huf文件开头都有占1个字节的标志,用来区分是文件压缩还是文件夹压缩,它的值只可能是0或1,可以应对绝大部分其他来源却有相同后缀的情况
        if (ch == 0 || ch == 1)
        {
            bool isFolder;
            isFolder = ch;
            if (isFolder)
            {
                folderDehuf(ifs);
            }
            else
            {
                fileDehuf(ifs);
            }
```

```
        }
        else
        {
            cout << "This is not created by me, unable to dehuf(decompress)"
<< endl;
        }
    }
    else
    {
        cout << "This is not created by me, unable to dehuf(decompress)" <<
endl;
    }
}
```

(3) 文件覆盖问题

首先,只有文件需要考虑覆盖的情况,

相关代码:

```
bool HufTool::fileOverwrite(string str) // 文件覆写判断
{
    bool flag = true;
    filesystem::path filePath = ".\\";
    filePath.append(str);
    if (filesystem::is_regular_file(filePath))
    { // 压缩后的文件已存在
        cout << str << " already exists. Overwrite it?" << endl;
        cout << "type 'y' or 'yes' if you want; type anything else like 'n'
or 'no' if you don't want" << endl;
        string ans;
        cin >> ans;
        if (ans != "y" && ans != "yes")
        {
            flag = false;
        }
    }
    return flag;
}
```

(4) 压缩包预览

如果压缩包内是单个文件,不用多说,很简单;如果是文件夹,则需要调用readDir()读取目录信息,并依据深度信息确定文件名或文件夹名前的空格数,由于示例中给出的字符我不知道如何打出,所以我使用'|'和'__'来"绘制"目录

相关代码:

```
void HufTool::preview(string input) // 压缩包预览
{
    string suffix;
    int size = input.size();
    for (int i = size - 4; i < size; i++)
    {
        suffix.push_back(input[i]);
    }
    // 判断后缀,后缀不同,肯定是不同来源
```

```
    if (suffix == ".huf")
    {
        ifstream ifs;
        ifs.open(input, ios_base::binary);
        if (!ifs)
        {
            exit(1);
        }
        char ch;
        ifs.get(ch);
        // 所有huf文件开头都有占1个字节的标志,用来区分是文件压缩还是文件夹压缩,它的值只可能是0或1,可以应对绝大部分其他来源却有相同后缀的情况
        if (ch == 0 || ch == 1)
        {
            bool isFolder;
            isFolder = ch;

            if (isFolder)
            {
                readDir(ifs); // 读取目录
                vector<string> strVec;
                for (int i = 0; i < dirVec.size(); i++)
                {
                    string str;
                    int dep = dirVec[i].dep;
                    if (dep > 0)
                    {
                        for (int j = 1; j < dep; j++)
                        {
                            str += "    ";
                        }
                        str += "|__ ";
                    }
                    filesystem::path filePath = dirVec[i].pathStr;
                    str += filePath.filename().string();
                    strVec.push_back(str);
                }
                for (int row = strVec.size() - 1; row > 0; row--)
                {
                    int size = strVec[row].size();
                    for (int column = 0; column < size; column += 4)
                    {
                        if (strVec[row][column] == '|' && strVec[row - 1][column] == ' ')
                        {
                            strVec[row - 1][column] = '|';
                        }
                    }
                }
                for (int i = 0; i < strVec.size(); i++)
                {
                    cout << strVec[i] << endl;
                }
            }
            else
            {
                string name;
                ifs.seekg(521, ios_base::cur);
```

```
                rwName(ifs, name); // 读写文件名
                cout << "name: '" << name << "'" << endl;
            }
        }
        else
        {
            cout << "This is not created by me, unable to preview" << endl;
        }
    }
    else
    {
        cout << "This is not created by me, unable to preview" << endl;
    }
}
```

(5) 与其他压缩工具的压缩率和压缩时间比较

我是在Windows Powershell下用Measure-Command命令进行测试,为了能够使用haozip,winrar和7z的命令模式,需要添加环境变量.

为了保证相对公平需要控制变量:

- 测试时,我的笔记本电脑始终保持在高性能模式
- 确保不存在覆写问题影响压缩时间
- 测试时不打开和关闭其他软件
- 多次测试取平均时间

为了保证一定的全面性,选取了不同大小不同格式的文件和文件夹

完整表格如下,还可以看pj文件夹中的excel文件

time/ms	huf	haozip	winrar	7z	size/KB	.huf	.zip	.rar	.7z	percent/%	huf	haozip	winrar	7z	dehuf.time/ms
3.csv	23933	9023	8957	31964	628333	402067	73788	49695	54326	3.csv	63.98948	11.74345	7.909023	8.646052	31225
36.txt	613	772	414	3721	14386	10459	6373	4216	3753	36.txt	72.70263	44.30001	29.30627	26.08786	903
1.jpg	914	653	656	1561	20262	20206	20204	20246	20235	1.jpg	99.72362	99.71375	99.92103	99.86675	1887
1.txt	99	195	100	356	1939	1086	714	602	558	1.txt	56.00825	36.8231	31.04693	28.77772	126
7.pdf	36	124	81	48	127	103	92	92	89	7.pdf	81.10236	72.44094	72.44094	70.07874	31
3-f	23223	1684	13476	14965	430390	432610	429167	429979	426875	3-f	100.5158	99.71584	99.90451	99.1833	42083
2-f	222	121	180	552	4131	2741	956	871	554	2-f	66.35197	23.1421	21.08448	13.4108	276

(6) 开发文档

略

PS: 我直接调用本文;在开发文档里写开发文档的设计和实现思路,搁这递归呢

3. 开发环境和工具以及如何编译和运行程序

开发环境和开发工具: Windows 11下的Visual Studio Code,使用的扩展如下图

编译和运行程序: 编译器使用[GCC 9.2.0 x86_64-w64-mingw32]

编译前需要配置好CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(huf)
add_executable(huf huf.cpp main.cpp)
include_directories(include)
```

鼠标单击下图中的三角形即可编译

然后在pj文件夹build目录下执行`./huf.exe 参数1 参数2 参数3`即可运行

例如我要压缩1.txt,不指定压缩文件名,则键入`./huf.exe huf 1.txt`

压缩3.csv,指定文件名3c.huf,则键入`./huf.exe huf 3.csv 3c.huf`

解压1.huf,则键入`./huf.exe dehuf 1.huf`

预览3.huf,则键入`./huf.exe preview 3.huf`

需要帮助,则键入`./huf.exe help`

4. 性能测试结果

表格如下,还可以看pj文件夹中的excel文件的Sheet2

时间/ms	huf	大小/KB	.huf	压缩率/%	huf	huf解压时间/ms
3.csv	23933	628333	402067	3.csv	63.98948	31225
36.txt	613	14386	10459	36.txt	72.70263	903
1.jpg	914	20262	20206	1.jpg	99.72362	1887
1.txt	99	1939	1086	1.txt	56.00825	126
7.pdf	36	127	103	7.pdf	81.10236	31
3-folder	23223	430390	432610	3-folder	100.5158	42083
2-folder	222	4131	2741	2-folder	66.35197	276
empty.txt	25	0	0.52	empty.txt	#DIV/0!	29
Empty-f	24	0	0.015	Empty-f	#DIV/0!	22
1-folder	59	574	377	1-folder	65.67944	61
1.csv	16053	430418	270558	1.csv	62.85936	23422

5. 与其他压缩工具的压缩率和压缩时间比较

完整表格如下,还可以看pj文件夹中的excel文件

time/ms	huf	haozip	winrar	7z	size/KB	.huf	.zip	.rar	.7z	percent/%	huf	haozip	winrar	7z	dehuf.time/ms
3.csv	23933	9023	8957	31964	628333	402067	73788	49695	54326	3.csv	63.98948	11.74345	7.909023	8.646052	31225
36.txt	613	772	414	3721	14386	10459	6373	4216	3753	36.txt	72.70263	44.30001	29.30627	26.08786	903
1.jpg	914	653	656	1561	20262	20206	20204	20246	20235	1.jpg	99.72362	99.71375	99.92103	99.86675	1887
1.txt	99	195	100	356	1939	1086	714	602	558	1.txt	56.00825	36.8231	31.04693	28.77772	126
7.pdf	36	124	81	48	127	103	92	92	89	7.pdf	81.10236	72.44094	72.44094	70.07874	31
3-f	23223	1684	13476	14965	430390	432610	429167	429979	426875	3-f	100.5158	99.71584	99.90451	99.1833	42083
2-f	222	121	180	552	4131	2741	956	871	554	2-f	66.35197	23.1421	21.08448	13.4108	276

*由于WinRAR的算法是闭源的,故下面不做分析

(1) 压缩时间

从上表可以看到在文件小于2000KB,我的程序相比于haozip和7z有明显的优势,但随着文件的增大,到3.csv(628333KB),虽然相较于7z仍然有优势,但所用时间已经是haozip的2.65倍了,可以看出我的程序时间复杂度相较于haozip更大

(2)压缩率

多数情况下,7z的压缩率都是最优的;haozip次之,且与7z相差较小;我的程序压缩率最大,且与前两者相差较大,在csv格式的大文件中尤其明显.不过在jpg文件和以jpg文件为主的文件夹上,压缩率都接近100%,因为jpg本身就是压缩率极高的图片格式了.

(3)简要分析

查阅资料得知haozip使用的是deflate算法,该算法结合了哈夫曼编码和LZ77算法,其中LZ77算法利用字符串有很多子串是重复出现的这一特点来压缩数据。压缩过程中，会从已压缩的数据中查找该字符是否在前面出现过，如果出现过，则只需保存该字符与以前出现字符的距离以及字符长度。其中已压缩的数据会有最大长度，如果超出该长度则滑动窗需要前移。

由此不难想象haozip的压缩率远小于我的程序,也因为haozip的算法更复杂,在文件很小时,耗时要多于我的程序;但是关于时间复杂度的问题我没有想清楚,也许出在优化上?

而7z使用的是LZMA算法,该算法有极高的压缩比,根据资料,它是是LZ77的改进版它使用链式压缩方法，在比特而非字节级别上应用修改后的LZ77算法。该压缩算法的输出稍后被算数编码进行处理以便后续进一步压缩。

6. 遇到的问题和解决方案

(1)"灵异"事件

在编写完文件压缩和解压后进行测试时发现,压缩1.txt然后解压得到的文件相比原始文件在开都多出了"eeo",检查1.huf,发现编码正常;又去用其他测试用例测试,也发现了类似现象,在把一些代码封装成新的函数后,这个现象又消失了;之后在我编写文件夹压缩时又发生了.

解决方案

通过断点调试发现是fileDehuf()中的`int cnt`忘记初始了,调试时居然是以`cnt = 5`开始,不出错才显得奇怪,所以还是要养成良好的初始化习惯,否则就不时地发生"灵异"事件.

相关代码如下:

```
void HufTool::fileDehuf(ifstream &ifs) // 解压缩文件
{
    ofstream ofs;
    string output;
    long long length = getLength(ifs);     // 获取长度
    remakeHT(ifs);                         // 重新造树
    int nameLength = rwName(ifs, output); // 读写文件名并返回名字长度
    if (fileOverwrite(output))
    {
        ofs.open(output, ios_base::binary);
        if (!ofs)
        {
            ifs.close();
            exit(1);
        }
        int now, bit, cnt = 0;
        char byte;
        for (long long i = 0; i < length * 8 - cover;)
        {
            now = root;
            while (HT[now].lchild != -1)
            {
                if (cnt % 8 == 0)
                {
                    ifs.get(byte); // 每次读一个字节
                    cnt = 0;
                }
                bit = byte & 0x80; // 获取当前的比特位,也就是byte的首位
```

```
                if (bit == 0)
                {
                    now = HT[now].lchild;
                }
                else
                {
                    now = HT[now].rchild;
                }
                byte = byte << 1; // 字节左移一位，以便下一次获取1bit
                cnt++;
                i++;
            }
            if (now > 127) // 范围控制在-128到127之间,防止输入值超出char型的范围
            {
                now -= 256;
            }
            ofs.put(now);
        }
    }
    ofs.close();
}
```

(2)断开的目录

下面是第一次做出的目录预览的效果,直接裂开了(一语双关)

```
test
|__ 1
    |__ 11
        |__ 111
        |__ 112.txt
        |__ 113
    |__ 12.txt
    |__ 13
|__ 2.txt
|__ 3
    |__ 31
    |__ 32.txt
    |__ 33
```

下面是修改后的

```
test
|__ 1
|   |__ 11
|   |   |__ 111
|   |   |__ 112.txt
|   |   |__ 113
|   |__ 12.txt
|   |__ 13
|__ 2.txt
|__ 3
|   |__ 31
|   |__ 32.txt
|   |__ 33
```

解决方案

之后我放弃了读一个目录节点输出一个目录节点的方式,改为存到一个vector的容器中,然后再遍历除了第一行的每一个字符,如果该字符为'|'其它上面的字符为' ',则将上面的字符改为'|',最后一起输出

```
test
|__ 1
|   |__ 11
|   |   |__ 111
|   |   |__ 112.txt
|   |   |__ 113
|   |__ 12.txt
|   |__ 13
|__ 2.txt
|__ 3
    |__ 31
    |__ 32.txt
    |__ 33
```

相关代码如下:

```
readDir(ifs); // 读取目录
vector<string> strVec;
for (int i = 0; i < dirVec.size(); i++)
{
    string str;
    int dep = dirVec[i].dep;
    if (dep > 0)
    {
        for (int j = 1; j < dep; j++)
        {
            str += "    ";
        }
        str += "|__ ";
    }
    filesystem::path filePath = dirVec[i].pathStr;
    str += filePath.filename().string();
    strVec.push_back(str);
}
for (int row = strVec.size() - 1; row > 0; row--)
{
    int size = strVec[row].size();
    for (int column = 0; column < size; column += 4)
    {
        if (strVec[row][column] == '|' && strVec[row - 1][column] == ' ')
        {
            strVec[row - 1][column] = '|';
        }
    }
}
for (int i = 0; i < strVec.size(); i++)
{
    cout << strVec[i] << endl;
}
```

7. 其他想说明的问题

在测试程序压缩和解压的内存占用时没有找到特别好的方法,于是就通过任务管理器查看内存是否有被大量的占用,发现即便是压缩解压大文件,内存占用率也没有明显增加

`filesystem::path`对于路径的大小写似乎有一些我不清楚的机制,也许会导致一些奇怪的问题,比如在压缩empty.txt得到empty.huf后,压缩键入`./huf.exe huf Empty Empty.huf`会提示是否要覆写,如果输入y或者yes,最终并没有生成Empty.huf,empty.huf的修改日期却被更新了;在删除empty.huf后再次键入`./huf.exe huf Empty Empty.huf`就能正常得到Empty.huf.