PJ2说明文档

一共分为四个内容：实现思路、代码分析、测试用例、遇到的问题（可跳过看后面的内容~）

实现思路

压缩：

单文件压缩：

首先以二进制的形式读取文件，统计不同字符出现的频率，共256个字符，所以可以用大小为256的int数组来存储，然后根据频率数组构建哈夫曼树，获得哈夫曼编码，获得哈夫曼编码以后可以开始写入新的文件，但是写入文件内容前，需要写入一些头数据，比如文件名、文件后缀名、字符与对应的哈夫曼编码（也可以选择存频率），然后读取文件，根据读取的字符，连接哈夫曼编码，每八位构成一个新的字符写入文件中，但是写到最后可能不足八位，需要补零至八位再存储。

目录（文件夹）压缩：
根据输入的路径判断其是目录还是文件，如果是文件的话，就调用对文件的压缩函数，如果是目录，则读取该目录下的内容，递归调用函数，直到把目录下的所有文件都压缩为止。把所有单文件的压缩写入同一个文件下，所以还需要存储文件的路径、大小等内容。最终把整个目录压缩完成。

空文件与空目录的处理：

对于空文件，只需要存储文件名，但为了适配目录压缩，可以再存入文件大小。对于空目录，存储目录即可，只有空目录存储路径，非空目录不需要存储目录的路径。

解压缩：

单文件解压缩：

首先是读取文件，然后根据头文件信息创建原文件，判断原文件是否为空文件，若为空文件，只负责创建文件即可。若非空，则读取哈夫曼编码，为了提升时间效率，根据哈夫曼编码重新构建哈夫曼树，然后按字节读取压缩后的文件，根据01左右探寻哈夫曼树，遇到叶子节点则得到对应的字符，将其写入原文件中即可。

目录（文件夹）解压缩：

读取文件，根据读取的路径依次判断顶层目录是否存在，如果不存在则创建新的目录（此即实现了多层目录），直到遇见文件名，创建原文件，进行单文件的解压缩，继续读取下一个路径，循环上述过程，最终可以实现目录的解压缩。因为空目录存的是路径，所以根据路径创建新的目录就可以实现空目录的解压缩。但其实最开始是不知道是文件还是目录的，实现的时候是一起实现的，这里只是为了说明不同的思想，所以写开了

关于写入与读取:

这里借鉴了Java中的缓冲区的概念，即先把数据存入一个数组中，然后一起打包写入，或者一起打包读出存入一个数组中，然后通过此数组读取相应数据。是一种提升效率的方式。

关于位运算：

写入的过程使用了位运算，而不是频繁反复地对字符串进行截取，拼接，大大加快了时间！！！！

代码分析

哈夫曼树的类

```
class HuffmanNode {  //哈夫曼结点的类定义
public:
    unsigned char ch;
    int freq;
    HuffmanNode* lft;
    HuffmanNode* rig;
public:
    unsigned char get_ch() { return ch; }
    int get_freq() { return freq; }
    HuffmanNode(HuffmanNode* test);
    HuffmanNode(pair<unsigned char, int>test);
    HuffmanNode();
    bool is_leaf();
};
class HuffmanTree {
public:
    HuffmanNode* root;
    string huffcode[256];  //存放生成的哈弗曼编码，以01串的形式储存
    int* huffcodeBits[256]; //按位存储哈弗曼编码以数字0,1存储
    int num;
    HuffmanTree(int* file); //根据字符频率数组构建哈夫曼树
    void get_huff_code(HuffmanNode*& rot, string huff); //得到哈夫曼编码，实例化
huffcode和huffcodeBits
    void build_from_code(const unsigned char& ch, const string& code);//根据01串构
建哈夫曼树，解压缩的时候要用
    void destroy(HuffmanNode* root);//析构函数
    HuffmanTree() { root = NULL; }
    ~HuffmanTree() { destroy(root); };
};
```

压缩compress类

```
class Compress {
public:
    string filename;//实际上，是相对路径
    int filedt[256] = { 0 };//存放读取的文件字符频率
    Compress(string filename) { this->filename = filename; }
    void compress();//总的压缩函数
    void files_compress(string name, FILE*& fw);//文件夹的压缩函数
    void file_compress(FILE*& fw, string filename, FILE* fp);//文件的压缩函数
    void read_file(FILE* fp, long int& length, unsigned char*& data);//读取文件数据
    void get_head_data(HuffmanTree* huffmanTree, unsigned char*& dt, long int&
index, int size);//准备好需要写入的头数据
    void get_file_data(HuffmanTree* huffmanTree, unsigned char*& dt, long int&
index, long int length, unsigned char* data);//准备好写入的文件数据，最后一起打包写入
};
```

解压缩decompress类

```
class Decompress {
public:
    string filename;
    HuffmanTree* huffmanTree = NULL;
    Decompress(string filename) { this->filename = filename; }
    string to_string(int a);//把读取出的字节转换成01字符串
    void decompress();//解压缩函数，调用文件的解压缩函数
    int file_decompress(FILE* fp, int idx);//文件的解压缩函数
    void decode_huffman_tree(unsigned char* data, long int& index);//读取哈弗曼编码并构建哈夫曼树
    void write_new_file(string lastname, unsigned char*& data, long int& index, long int length);\\写新的文件，即得到原文件
};
```

一些重要函数

HuffmanTree(int* file);

```
HuffmanTree::HuffmanTree(int* file) {   //create huffmantree
    vector<HuffmanNode*> vec;
    for (int i = 0; i < 256; i++) {
        if (file[i] != 0) {
            pair<unsigned char, int>temp((unsigned char)i, file[i]);
            HuffmanNode* t = new HuffmanNode(temp);
            vec.push_back(t);
        }
        this->huffcodeBits[i] = NULL;
    }  //把频率非零的存入vector中
    if (vec.size() == 1) {
        HuffmanNode* last = new HuffmanNode;
        last->lft = new HuffmanNode(vec[0]);
    } //只有一个元素的处理
    while (vec.size() != 1) {
        sort(vec.begin(), vec.end(), comp);
        HuffmanNode* last = new HuffmanNode;
        last->lft = new HuffmanNode(vec[0]);
        last->rig = new HuffmanNode(vec[1]);
        last->freq = last->lft->freq + last->rig->freq;
        vec.erase(vec.begin(), vec.begin() + 2);
        vec.push_back(last);
    }   //每次对vector排序得到最小的前两个，然后为左右子树，把合并了的加入vector，重复上述过程，直到只剩下最后一个，即为根节点。
    root = vec[0];
    vec.erase(vec.begin(), vec.begin() + 1);//释放vector空间
}
```

get_huff_code(HuffmanNode*& rot, string huff);

```
void HuffmanTree::get_huff_code(HuffmanNode*& rot, string huff) {
    if (rot->is_leaf()) {//如果是叶子结点的话，就存入哈弗曼编码，还有huffcodeBits的实例化
        huffcode[rot->ch] = huff;
        huffcodeBits[rot->ch] = new int[huff.length()];
        for (int i = 0; i < huff.length(); i++) {
            huffcodeBits[rot->ch][i] = huff[i] - '0';
```

```
        }
        num++;
        return;
    }
    else {
        if (rot->lft) {
            get_huff_code(rot->lft, huff + "0");\\左节点，加0
        }
        if (rot->rig) {
            get_huff_code(rot->rig, huff + "1");\\右节点，加1
        }
    }
}
```

void build_from_code(const unsigned char& ch, const string& code);

```
void HuffmanTree::build_from_code(const unsigned char& ch, const string& code) {
    if (this->root == NULL) {
        root = new HuffmanNode;
    }
    HuffmanNode* curr = root;  \\每次从根节点出发探查
    for (int i = 0; i < code.length(); i++) {
        if (code[i] == '0') {   \\0的话，向左探查，若左节点为空则开辟空间
            if (curr->lft == NULL) {
                curr->lft = new HuffmanNode;
            }
            curr = curr->lft;
        }
        else {    \\1的话，向右探查，若右节点为空则开辟空间
            if (curr->rig == NULL) {
                curr->rig = new HuffmanNode;
            }
            curr = curr->rig;
        }
    }
    curr->ch = ch;//赋值
}
```

void Compress::file_compress(FILE*& *fw, string filename, FILE* fp) ;

```
void Compress::file_compress(FILE*& fw, string filename, FILE* fp) {
    unsigned char* data;
    long int length;
    assert(fp != NULL);
    fseek(fp, 0, SEEK_END);
    length = ftell(fp);
    if (length == 0) { //对空文件的处理
        fclose(fp);
        unsigned char a = (unsigned char)filename.size();
        data = new unsigned char[100];
        length = -1;
        data[++length] = a;
        for (int i = 0; i < filename.size(); i++) {
            data[++length] = filename[i];
        }
        data[++length] = (unsigned char)0;
        fwrite(data, 1, ++length, fw);
```

```
        int v = 0;
        fwrite(&v, 8, 1, fw);
        fclose(fw);
        delete[]data;
    }
    else {
        read_file(fp, length, data);
        HuffmanTree tree(filedt);
        string huff = "";
        tree.get_huff_code(tree.root, huff);
        unsigned char* dt = new unsigned char[2000 + 5 * length];
        long int index = -1;
        get_head_data(&tree, dt, index, tree.num);
        get_file_data(&tree, dt, index, length, data);
        unsigned char a = (unsigned char)filename.size();
        fwrite(&a, 1, 1, fw);
        unsigned char* t = new unsigned char[filename.size() + 1];
        for (int i = 0; i < filename.size(); i++) {
            t[i] = filename[i];
        }
        t[filename.size()] = (unsigned char)0;
        fwrite(t, 1, filename.size() + 1, fw); //打包写入文件名
        fwrite(&(++index), 8, 1, fw); //写入压缩后文件大小
        fwrite(dt, 1, index, fw); //打包写入数据
        finish = clock();
        delete[]data;
        delete[]dt;
    }
}
```

void Compress::get_file_data(HuffmanTree* huffmanTree, unsigned char*& *dt, long int& index, long int length, unsigned char* data);

```
void Compress::get_file_data(HuffmanTree* huffmanTree, unsigned char*& dt, long
int& index, long int length, unsigned char* data) {
    int len = 0;
    int val = 0;
    int** huffcodeBits = huffmanTree->huffcodeBits;
    for (long int i = 0; i < length; i++) {
        int* temp = huffcodeBits[data[i]];  //data[i]表示读到哪一个字节，temp存储该字
节对应的按位存储的哈弗曼编码
        for (int idx = 0; idx < huffmanTree->huffcode[data[i]].length(); idx++)
{ //查找对应的字符
            val = (val << 1) + temp[idx]; //相当于给val接上temp 的对应位
            len++;
            if (len >= 8) { //如果大于等于八位说明此时应该生成一个字符写入到新的文件中
                dt[++index] = (unsigned char)val;
                val = 0;   //val重新置为0
                len = 0;   //len重新置为0
            }
        }
    }
    if (len > 0) {  //最后如果不足八位的话就补零，共需8-len个0需要补足
        val = val << (8 - len);
        dt[++index] = (unsigned char)val;
```

```
    }
}
```

void Compress::files_compress(string name, FILE*& fw) ;

```
void Compress::files_compress(string name, FILE*& fw) {
    int flag = 0;
    struct stat filedata;
    stat(name.c_str(), &filedata);
    if (S_ISDIR(filedata.st_mode)) { //判断是否为目录
        cout << name << "  files write success!!" << endl;
        struct dirent* file;
        DIR* dp = opendir(name.c_str());
        file = readdir(dp);
        while (file) {
            char file_path[200] = {};
            strcat_s(file_path, name.c_str());
            strcat_s(file_path, "/");
            strcat_s(file_path, file->d_name);
            if (strcmp(file->d_name, ".") != 0 && strcmp(file->d_name, "..") !=
0) {
                stat(file_path, &filedata);
                if (S_ISDIR(filedata.st_mode)) {  //内层还有目录，递归调用
                    files_compress(file_path, fw);
                }
                if (S_ISREG(filedata.st_mode)) {  //是文件
                    cout << file_path << " write success!!"<< endl;
                    FILE* fp;
                    fopen_s(&fp, file_path, "rb");
                    file_compress(fw, file_path, fp); //调用单文件压缩函数
                    for (int i = 0; i < 256; i++) {
                        filedt[i] = 0;
                    }
                }
            }
            if (strcmp(file->d_name, "..") == 0) {
                file = readdir(dp);
                if (file == NULL) {  //对空目录的处理
                    cout << " empty files " << endl;
                    unsigned char a = (unsigned char)name.size();
                    fwrite(&a, 1, 1, fw);
                    unsigned char* t = new unsigned char[name.size() + 1];
                    for (int i = 0; i < name.size(); i++) {
                        t[i] = name[i];
                    }
                    t[name.size()] = '/';
                    fwrite(t, 1, name.size() + 1, fw);
                    int v = 0;
                    fwrite(&v, 8, 1, fw);
                }
            }
            else {
                file = readdir(dp);
            }
        }
    }
}
```

void Decompress::decompress();

```
void Decompress::decompress() {
    int a = filename.rfind(".");
    string pat = filename.substr(a + 1, filename.size() - a);
    if (strcmp(pat.c_str(), "lxy")) {
        cout << "error! the file form is not right!" << endl;
    }
    FILE* fp;
    fopen_s(&fp, filename.c_str(), "rb");
    assert(fp != NULL);
    long int idx = 0;
    long int Length;
    fseek(fp, 0, SEEK_END);
    Length = ftell(fp);
    cout << Length << endl;
    rewind(fp);
    while (idx < Length) { //得到解压缩文件后的大小，看是否还有文件需要进行解压缩
        fseek(fp, idx, SEEK_SET);
        idx = idx + file_decompress(fp, idx) + 8;
    }
    fclose(fp);
}
```

void Decompress::decode_huffman_tree(unsigned char* data, long int& index);

```
void Decompress::decode_huffman_tree(unsigned char* data, long int& index) {
    int ch = (int)data[++index];
    ch++;
    for (int i = 0; i < ch; i++) {
        unsigned char temp = data[++index]; //读取是哪一个字符
        unsigned char a = data[++index];  //读取该字符占据几个字节
        unsigned char b = data[++index];  //读取补零数
        string huff = "";
        for (int j = 0; j < a; j++) {
            huff += to_string(data[++index]);
        }
        huff = huff.substr(0, huff.size() - b); //得到哈弗曼编码，01串
        huffmanTree->build_from_code(temp, huff); //根据01串创建 哈夫曼结点，进而得到
哈夫曼树
    }
    finish = clock();
}
```

void Decompress::write_new_file(string lastname, unsigned char*& data, long int& index, long int length) ;

```
void Decompress::write_new_file(string lastname, unsigned char*& data, long int&
index, long int length) {
    FILE* fw;
    fopen_s(&fw, lastname.c_str(), "wb");
    unsigned char* dt = new unsigned char[10 * length];
    long int index1 = 0;
    int comp = data[length - 1];
    HuffmanNode* curr = this->huffmanTree->root; //从哈夫曼树的根节点开始探查
```

```
    index++;
    while (index < length ) {
        int code = data[index++];
        int mask = 1 << 7; //mask置为10000000
        for (int i = 0; i < 8; i++) {
            if ((mask & code) == 0) { //代表对应位是0，因为mask对应位为1
                curr = curr->lft;
            }
            else {  //代表对应位为1，向右探查
                curr = curr->rig;
            }
            if (curr!=NULL&&curr->is_leaf()) { //curr不是NULL且为叶子结点，找到字符，
写入即可
                dt[index1++] = curr->ch;
                curr = this->huffmanTree->root; curr重新回到根节点
            }
            mask = mask >> 1; mask每向右移一位，就代表探查curr的右边一位
        }
    }
    fwrite(dt, 1, index1, fw); //打包写入，节省时间
    fclose(fw);
    delete[] dt;
}
```

file_decompress()的处理，部分代码如下

```
    for (int i = 0; i < lastname.size(); i++) {
        if (lastname[i] == '/') {
            a[j] = i;
            j++;
        }
    }
    for (int i = 1; i < j; i++) {
        int b = a[i];
        string temp = lastname.substr(0, b + 1);
        DIR* mydir = opendir(temp.c_str());
        if (mydir == NULL) {
            _mkdir(temp.c_str());
        }
    } //看父目录是否存在，不存在则创建
    if (lastname[lastname.size() - 1] == '/') {
        return sz + 1;
    } //看是否是空目录，如果是直接返回
    lastname[sz - 1] = '\0';
    fseek(fp, idx + sz + 1, SEEK_SET);
    fread(&length, 8, 1, fp);
    if (length == 0) { //对空文件的处理
        FILE* fw;
        fopen_s(&fw, lastname.c_str(), "wb");
        fclose(fw);
        return sz + 1;
    }
    data = new unsigned char[length];
    fread(data, 1, length, fp);
    index = -1;
    huffmanTree = new HuffmanTree;
    decode_huffman_tree(data, index);
```

```
    write_new_file(lastname, data, index, length);
```

—compress()与_ decompress()函数

```
void _compress() {
    clock_t start, finish;
    double time;
    string filename;
    cout << "please input file's name to compress" << endl;
    getline(cin,filename);
    start = clock();
    Compress tet(filename);
    tet.compress();
    finish = clock();
    time = (double)(finish - start) / CLOCKS_PER_SEC;
    cout << "compress time:  " << time << endl;
}
void _decompress() {
    clock_t start, finish;
    double time;
    string test;
    cout << "please input file's name to decompress" << endl;
    geiline(cin,test);
    start = clock();
    Decompress tet(test);
    tet.decompress();
    finish = clock();
    time = (double)(finish - start) / CLOCKS_PER_SEC;
    cout << "decompress time:  " << time << endl;
}
```

main函数

```
int main() {
    _compress();
    _decompress();
}
```

测试用例

PS：测试用例的文件已经打包在提交的zip文件中

G3

	A	B	C	D	E	F	G
1	文件名	压缩时间	解压缩时间	压缩率	WINRAR压缩率	文件大小	
2	1.txt	0.134	0.132	0.566227	0.307458	1971kb	
3	1(目录)	0.224	0.21	0.632500	0.217917	2400kb	
4	3.csv	8.583	7.975	0.637020	0.077914	113792kb	
5	2（目录）	0.841	0.788	0.638110	0.221199	9340kb	
6	2.csv	8.243	8.918	0.641499	0.200536	113865kb	
7	3（目录）	0.233	0.229	0.650406	0.217886	2460kb	
8	25.aiff	0.009	0.004	0.689060	0.454545	22kb	
9	7.pdf	0.015	0.014	0.807764	0.716535	127kb	
10	35.avi	0.016	0.006	0.907343	0.735294	34kb	
11	27.mp3	0.141	0.179	0.992237	0.973303	1311kb	
12	21.mpeg	0.021	0.016	0.992359	0.871560	109kb	
13	1.jpg	2.009	2.736	0.997251	1.029069	20262kb	
14	17.tif	0.009	0.004	1.037970	0.066875	16kb	
15	13.jpg	0.012	0.006	1.050490	0.995238	21kb	
16	29.wav	0.008	0.003	1.205830	0.850000	5kb	
17	12.gif	0.008	0.003	1.336960	0.940000	3kb	
18	14.png	0.006	0.003	1.632970	0.595000	2kb	
19	4.gz	0.003	0.001	4.146850		1kb	
20							
21							

根据实验结果可以发现，哈弗曼编码比较适合对文本文件进行压缩，像图片文件，或者是音频，视频文件，由于其储存方式的差异，压缩率就没有那么理想。此外，对于小的文件，很有可能会出现写入的数据比原数据还要多的情况，对于小文件，其实也是没有必要进行压缩的。

遇到的问题

一、二进制读文件的相关问题

最开始不是很清楚fwrite和fprintf的区别，在试了一个简单的例子后，发现，前者是二进制读入，后者是预处理的文件读入。此外，之前是认为只可以按照顺序，依次读取文件数据而不可以找到具体的位置，经过此次PJ，学会了fseek函数的用法，对文件的一些宏定义，比如SEEK_END，有了比较清晰的认知，也学会了ftell来获取长度。

二、缓冲区buffer的理解

在Java中了解到，可以利用缓冲区来提高时间效率，C++中没有对应的缓冲区，其实，缓冲区就是一个中间介质，因为多次频繁地从文件中读取比较耗时间，不如只读一次，然后从中间介质中进行读取，虽然中间开辟了一些空间，但是相比之下，还是有缓冲区比较好，同样的道理，写入的时候，也可以利用buffer，打包一起写入。

三、关于位运算和字符串

最开始实现的时候，非常笨拙的频繁使用字符串的截取，拼接，以至于速度超级慢，甚至一个2MB的文本文档，都可以跑8秒，后来了解到，可以利用位运算，通过移位<<,>>等方式来实现，改进了之后，程序变得快了很多。由此可见，字符串的一些操作是很耗时的，其实也和字符串的函数的实现方法有关。在之前的学习中，不是很频繁的用到位运算，这次对位运算有了新的理解。

四、内存的申请与释放

这次PJ有好几处都用到了new来申请内存，但是由于马虎和粗心，不是忘记释放，就是释放多了，或者申请的不够，导致debug的时候非常痛苦，真的是一些警示，以后申请内存，一定要记得释放，一定要申请足够的内存。