中山大学数据科学与计算机学院本科生实验报告

(2019-2020 学年第一学期)

课程名称： 数据结构与算法　　任课教师： 乔海燕

年级&班级	18 级计科 8 班	专业(方向)	计算机类
学号	18340208	姓名	张洪宾
学号	18340214	姓名	张千玉
开始日期	2019-10-10	完成日期	2019-10-13

一、 引言

我们使用电脑会有各种格式的文件，它们虽然形形色色，但是归根到底都是用二进制存储的。我们的目标是写一个通用的软件，使它可以对任意格式的文件进行压缩和无损解压，达到市面上压缩软件的基本功能。而为了实现这个目标，我们需要对原来的文件的二进制码进行重新编码，进行压缩，然后再找到一个方法，再由编码文件转回原文件。然后我们最后比较一下原来的二进制文件和解压缩后的二进制文件是否相同就行了。

二、 解决方法

众所周知，文件编码都是用字节存储的，而一个字节有 8 个位，也就是说，任意一个文件的二进制编码中，0 和 1 的总个数都是 8 的整数倍，那我们不妨以字节为单位，因为对于一个字节，总共也就只有$2^8 = 256$种情况，如果读入程序可以建立一个 0 到 255 的整数集与各种字节的一一对应。而且我们的 char 变量也是用字节存储的，我们可以用 C++的文件流一个一个读取 char 变量然后处理。这是处理文件的二进制编码的方法。

然后来到这篇报告的核心了——Huffman 算法。再离散数学中我们曾经学过根据编码的权值来构造最优二叉树，使得带权路径最小的 Huffman 编码算法，

在这里就是使用这种方法使得我们最后生成的压缩编码尽可能的小，将新的编码写进文件，就实现了文件的压缩。

具体的 Huffman 算法及其应用举例如下：

Huffman 算法具体的步骤如下：

1. 初始化： 根据给定的 n 个权值{w_1、w_2、……、w_n }构成 n 棵二叉树的集合 F={T_1、T_2、……、T_n}，其中每棵二叉树T_i中只有一个带权w_i的根结点，左右子树均空。

2. 找最小树：在 F 中选择两棵根结点权值最小的树作为左右子树构造一棵新的二叉树，且至新的二叉树的根结点的权值为其左右子树上根结点的权值之和。

3. 删除与加入：在 F 中删除这两棵树，并将新的二叉树加入 F 中。

4. 判断：重复前两步（2 和 3），直到 F 中只含有一棵树为止。该树即为 Huffman 树，而每个叶子存储着我们需要用 Huffman 编码表示的值。

为了构建 Huffman 树，我设计了一个类表示 Huffman 编码的节点，如下：

```
class node {
public:
    int value;
    int power;
    node* left;
    node* right;
    node() = default;
    node(int temp, int x, node* a, node* b) {
        value = temp;
        power = x;
        left = a;
        right = b;
    }
    //修改优先级，让比较权小的结点先合并
    friend bool operator < (const node& a, const node& b) {
        return a.power > b.power;
    }
};
```

节点类中有四个成员变量,value是一个整数,代表256中字节中的某一个,只有叶节点才会存储 value，其他节点的 value 我在后面都置为-1。而 power 则是该节点的权,即该字节在文件中出现的频率,还有两个是节点的左右儿子,这个是针对我们在建树过程中生成的新节点而写的，用于保存两个删除的树的根节点的地址。然后我重载了两个节点的<,用于比较两个节点的权的大小,便于建树。

而权值的获取其实也是非常有学问的内容，为了统计文件中的每一个字节出现的频率,我开了个大小为256的数组来统计各个字节出现的频率,若按char读取的文件出现在其中的时候，数组元素自增1。

获取字节出现频率数组的实现如下:

```
//频率数组
int res[256];
//获取每个字节及其出现的频数
void get_frequency(char* source_file_name) {
    ifstream in;
    in.open(source_file_name, ios::binary);
    char buffer;
    in.read(&buffer, sizeof(char));
    while (!in.eof()) {
        res[(unsigned char)(buffer)]++;
        //all_alpha.push_back(buffer);
        in.read(&buffer, sizeof(char));
    }
    in.close();
}
```

我们不妨拿举个例子：假设我们读一个文件，它的字节分布如下

00000001 有 1 个，00000010 有 3 个，00001000 有 4 个，00000100 有 7 个。然后我们来构造它的 Huffman 编码:

首先，我们将二进制编码转化为十进制可以知道:

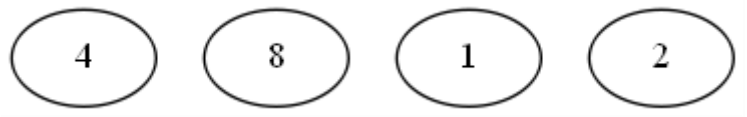
节点 1 的 power = 1

节点 2 的 power = 3

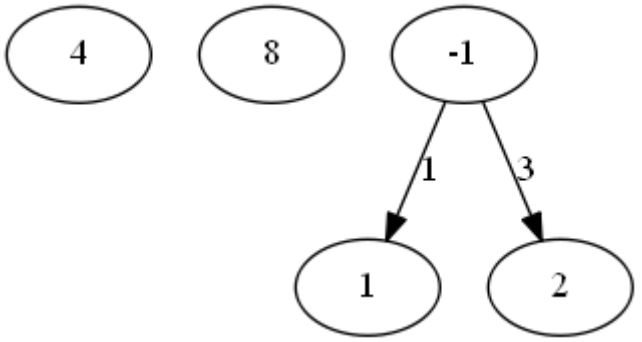
节点 8 的 power = 4

节点 4 的 power = 7

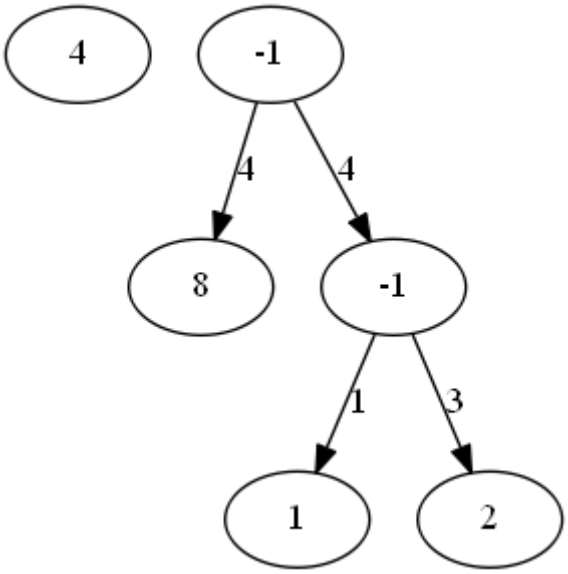
Step1：



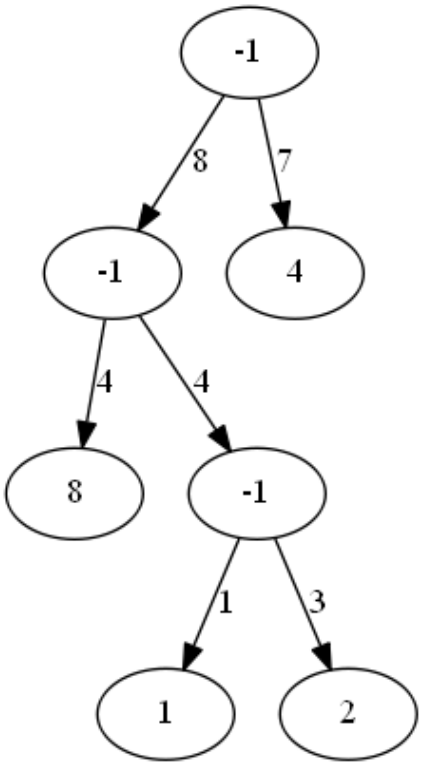
Step2：取出其中节点数最小的两个，即节点 1 和 2：



Step3：然后再在当前树中取根的权值最小的两个：



Step4：以此类推，不断操作可以得到 Huffman 树：



1：010

2：011

4：1

8：00

通过这个例子，我们可以得到，最重要的过程还是找到当前根权值最小的两个子树。为了解决这个问题，我研究了 STL 的优先队列，它可以完美的解决这个问题。

优先队列是元素带有优先级的特殊队列。可以让优先级高的先出队，优先在 top()函数出现。这与我们的要求是一样的。我在 Huffman 节点的定义中重载了<符合，也是为了将权值小的节点的优先级更大，然后我们只需要一次取两个节点出来，权值相加生成新的-1 节点并成为该-1 节点的左右儿子即可，再将新节点入队。而每次这样的操作都会使队列中节点数减 1，所以不断操作直至队列中只有一个节点时，唯一的节点就是最顶端的-1 节点。

而 STL 里面提供了优先队列的模板 priority_queue，我们只需要先根据字节统计数组构造一个优先队列，然后再根据这个优先队列生成 Huffman 树：

生成优先队列及用优先队列构造树的代码如下：

```
//由频率数组得到叶节点的优先队列
priority_queue<node> get_queue(int*res) {
    priority_queue<node> temp;
    for (int i = 0; i < 256; i++) {
        if (res[i]) {
            node newone(i, res[i], nullptr, nullptr);
            temp.push(newone);
        }
    }
    return temp;
}
//由优先队列建树
node* Huffman_Tree(priority_queue<node> queue) {
    while (queue.size() > 1) {
        node* temp1 = new node(queue.top());
        queue.pop();
        node* temp2 = new node(queue.top());
        queue.pop();
        node temp(-1, temp1->power + temp2->power, temp2, temp1);
        queue.push(temp);
    }
    node* head = new node(queue.top());
    return head;
}
```

加上程序结束时情况这段内存的 del 函数以及获取 Huffman 编码的 get_code 函数，这两个都是用递归函数实现的，不会很难，故不赘述。就这样与 Huffman 树有关的内容就算是写完了。

然后我们就要研究怎么将原来的文件转换为用 Huffman 编码编写的压缩文件就是一个比较大的问题了。我最开始将生成的新编码以 string 的形式传给新的文件，结果发现压缩后的文件比原来的文件大了好几倍。一开始我是懵逼的，然后仔细想了一下也确实如此：原文件是按位来存储的，而我如果以 string 写入新文件，相当于我用了原来 8 倍的大小表示一个 0 和 1，这显然是极其浪费空间的事情。于是我只能将 string 中的每一个 0 和 1 压缩到每一位中，使用位运算，将新的编码一位一位压进去。然后以 8 位（即一个 char

变量）为单位，不断的调用定位。然后把新的编码以多个 char 的形式写入新文件。

但是仅仅这样子我们还不足以实现解压。因为我们只有将解压的“密码”传进压缩文件的时候，压缩文件才可以通过解压程序生成原来的文件。关键的问题就是，我们用什么来存储这个解压方式。最开始我想要设计一个类，然后把各个字节对应的 Huffman 编码存进去，但是发现这样子占用的空间太多，特别是原文件比较大的时候，256 个字节都出现过的时候，这时候生成的 Huffman 编码都比较长，这样子会使得存编码占据比较大的空间，不划算。

这时候我看到了之前开来表示频率的数组。我想出一个方法，不如直接把频率数组写入文件，解压的时候再用一次之前构建好的 Huffman 树的代码，再次生成各个字节对应的 Huffman 编码，然后再翻译即可。所以我将频率数组中非 0 元素的个数，非 0 元素的索引及其值写入文件，所以写文件的全部代码如下：

```
//将频率数组与哈夫曼编码按位和二进制文件的方式写进文件
void write_out(char* source_file_name) {
    ifstream in;
    in.open(source_file_name, ios::binary);
    char buffer;
    string temp;
    char* output = new char[_file_length];
    memset(output, 0, sizeof(char) * _file_length);
    //重新读文件，因为如果将所有的字节直接读进数组或 vector 内存可能会炸
    in.read(&buffer, sizeof(char));
    int count = 0;
    long long int buffer_ptr = 0;
    string total;
    while (!in.eof()) {
        //在这里面将原字节转换为哈夫曼编码并压进新的字节
        string temp = code[(unsigned char)buffer];
        total += temp;
        for (int i = 0; i < temp.size(); i++) {
            int x = temp[i] - '0';
            x = x << (7 - count);
            output[buffer_ptr] = output[buffer_ptr] | x;
            count++;
```

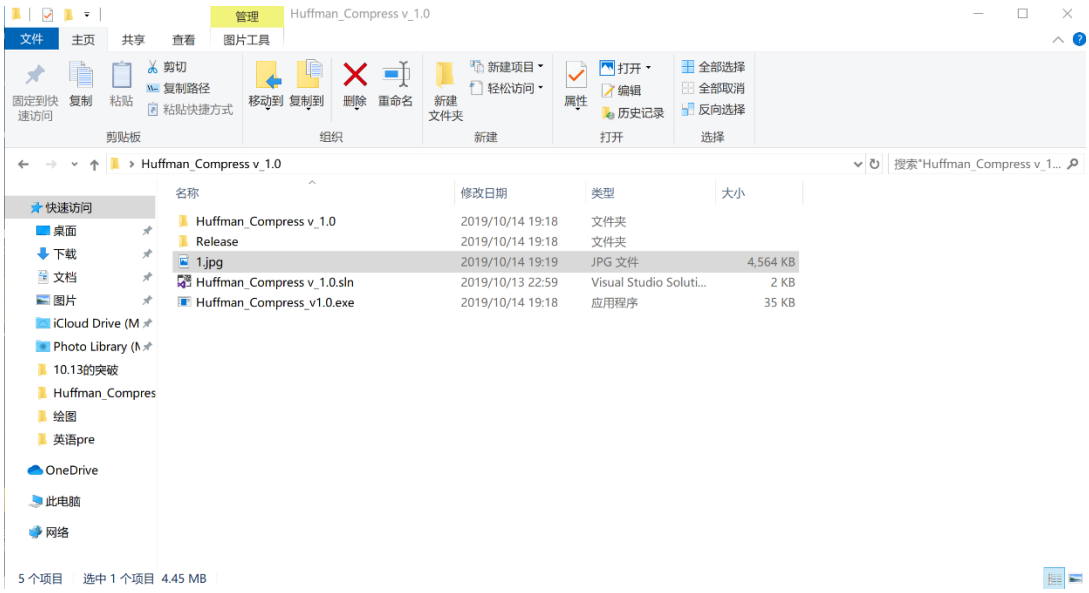
```
            if (count == 8) {
                count = 0;
                buffer_ptr++;
            }
        }
        in.read(&buffer, sizeof(char));
    }
    in.close();
    //开始输出
    ofstream out;
    string out_name = source_file_name;
    out_name += ".zhbzqyzip";//加上后缀名
    out.open(out_name, ios::out | ios::binary);
    //统计出现多少种字节
    int total_num = 0;
    for (int i = 0; i < 256; i++) {
        if (res[i])total_num++;
    }
    //将所有字节数存入文件，在读取的时候就可以用 for 循环定位
    out.write((char*)& total_num, sizeof(int));
    for (int i = 0; i < 256; i++) {
        //将该文件出现过的字节极其对应的哈夫曼编码存进文件，在解码的时候再构建树获取
        if (res[i]) {
            buffer = i;
            int num = res[i];
            out.write(&buffer, sizeof(char));
            out.write((char*)& num, sizeof(int));
        }
    }
    out.write((char*)& buffer_ptr, sizeof(int));
    out.write((char*)& count, sizeof(int));
    for (int i = 0; i <= buffer_ptr; i++) {
        char temp = output[i];
        int x = (unsigned char)output[i];
        out.write(&temp, sizeof(char));
    }
    out.write((char*)& _file_length, sizeof(int));
    out.close();
    delete[]output;
}
```

解压的步骤其实大同小异，首先是读文件，将文件中的频率数组读出来，然后构建树，获取各个字节的 Huffman 编码，然后用与压缩时一样的方法将 Huffman 编码转换为原文件中的编码，然后再将文件写入输出文件就行了。

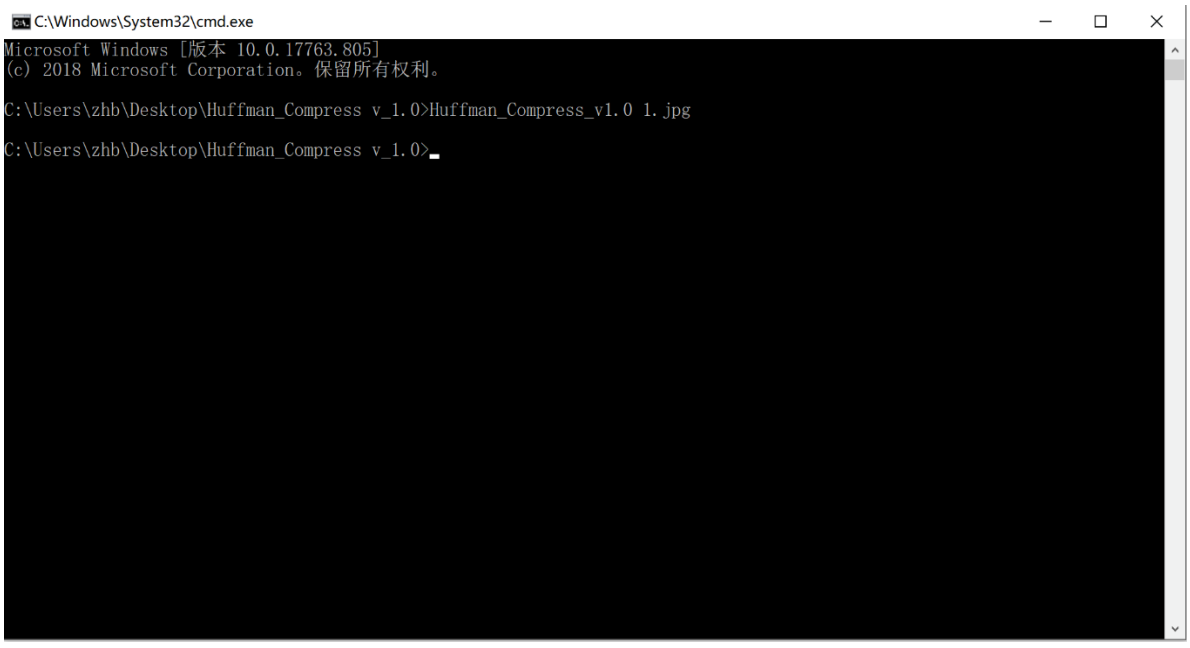
三、 程序使用和测试说明

为了可以使用命令行以及直接将待处理文件拖至 exe 文件，我在 main 函数传了参数，然后就可以直接运行。

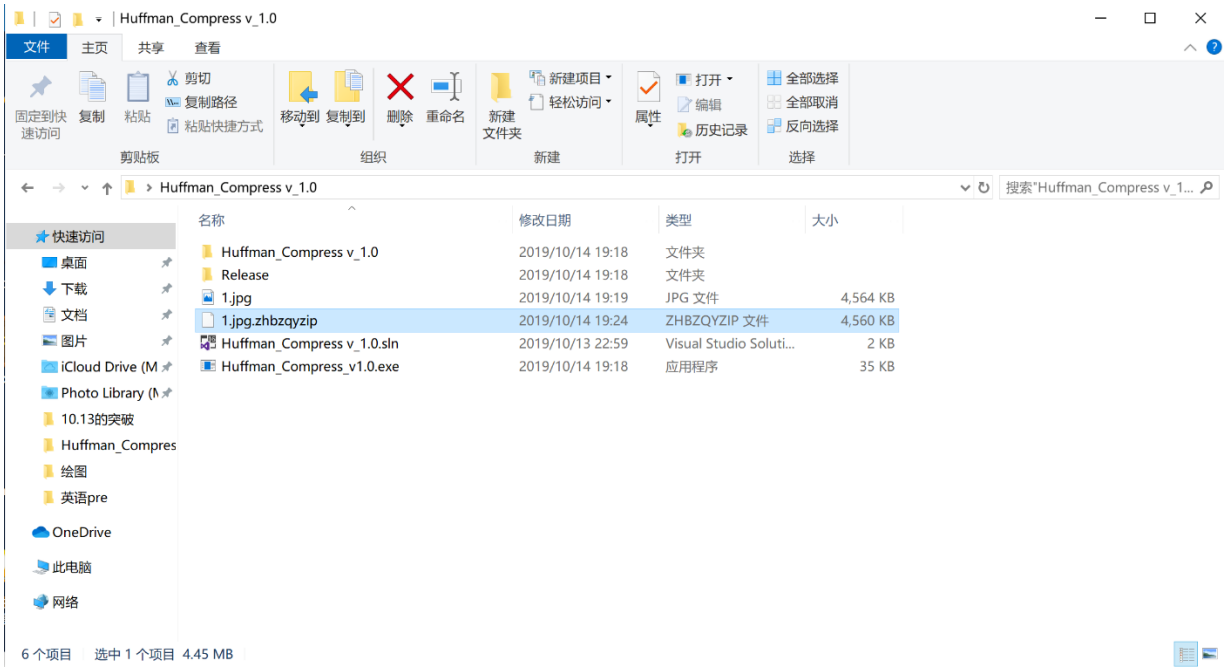
程序使用有两种方法，在此我举个例子。我首先拍了一张我们宿舍一位年级前十的大佬的照片，然后传到电脑上，将文件名改为 1.jpg，如图：



然后打开命令行，输入 exe 文件的名字（建议将 exe 文件的名字改成简单的 1，2 之类的，便于输入），然后再输入 1.jpg，按下回车，如图：

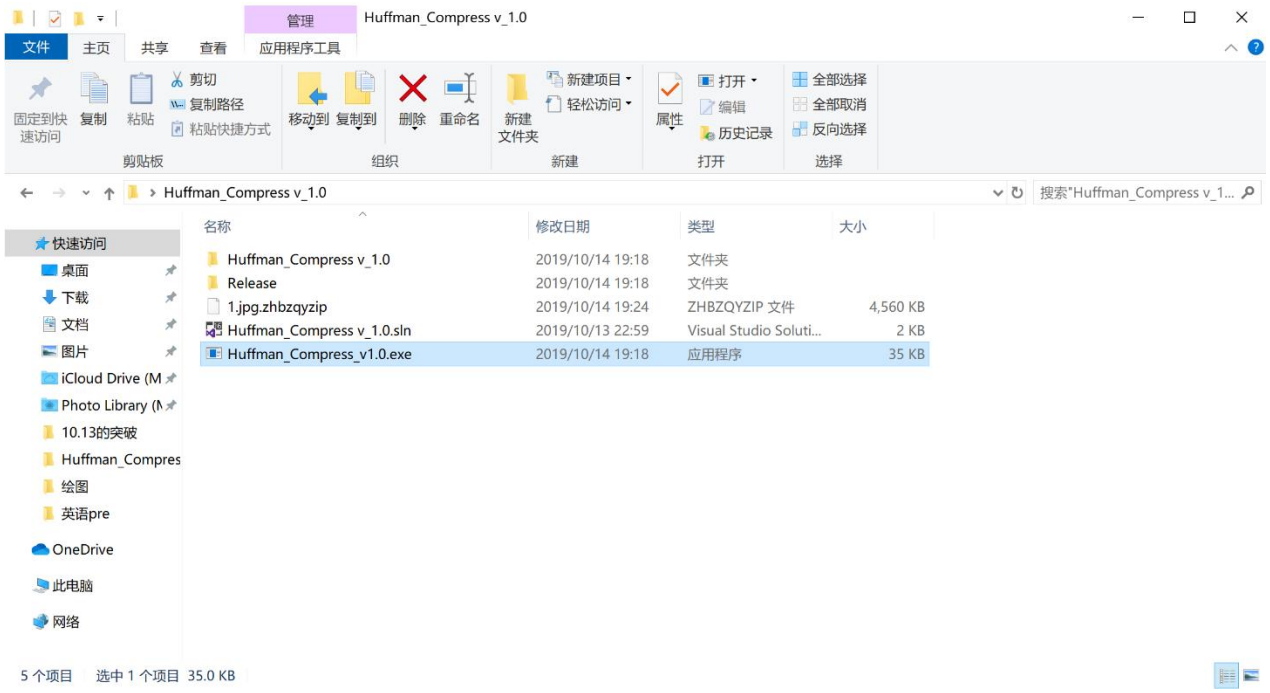


然后回到文件夹，如图：

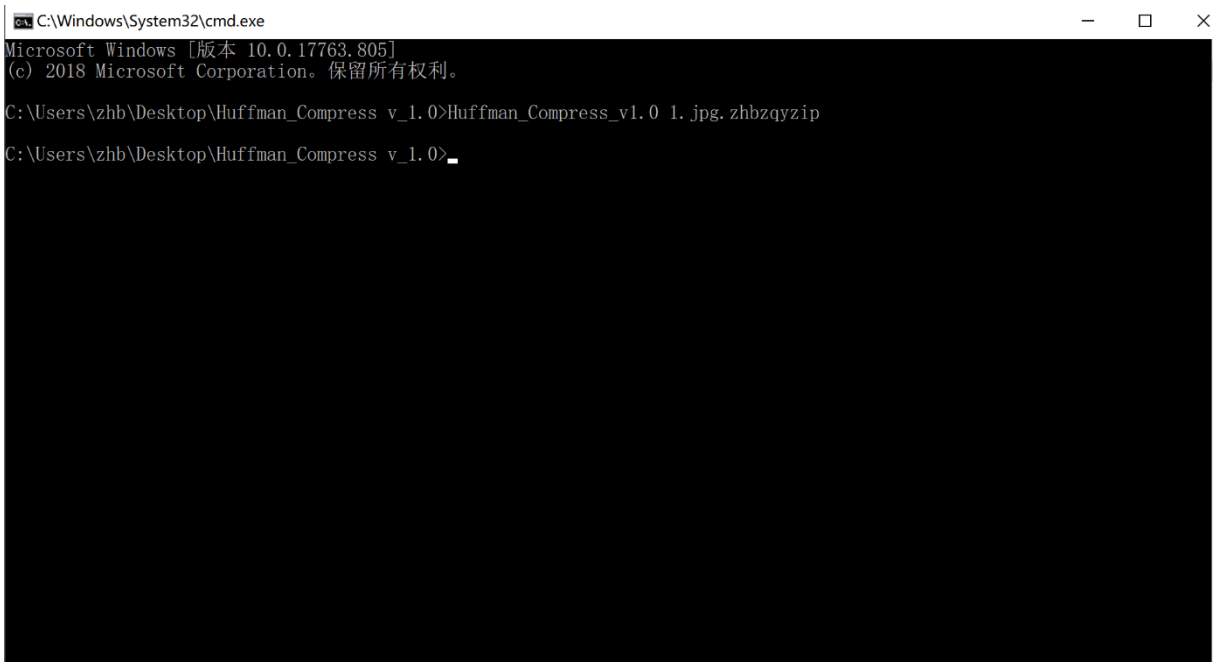


这时候多了一个名为 1.jpg.zhbzqyzip 的文件，这个就是我们生成的压缩文件，后缀名是为了让程序识别这是个压缩文件。

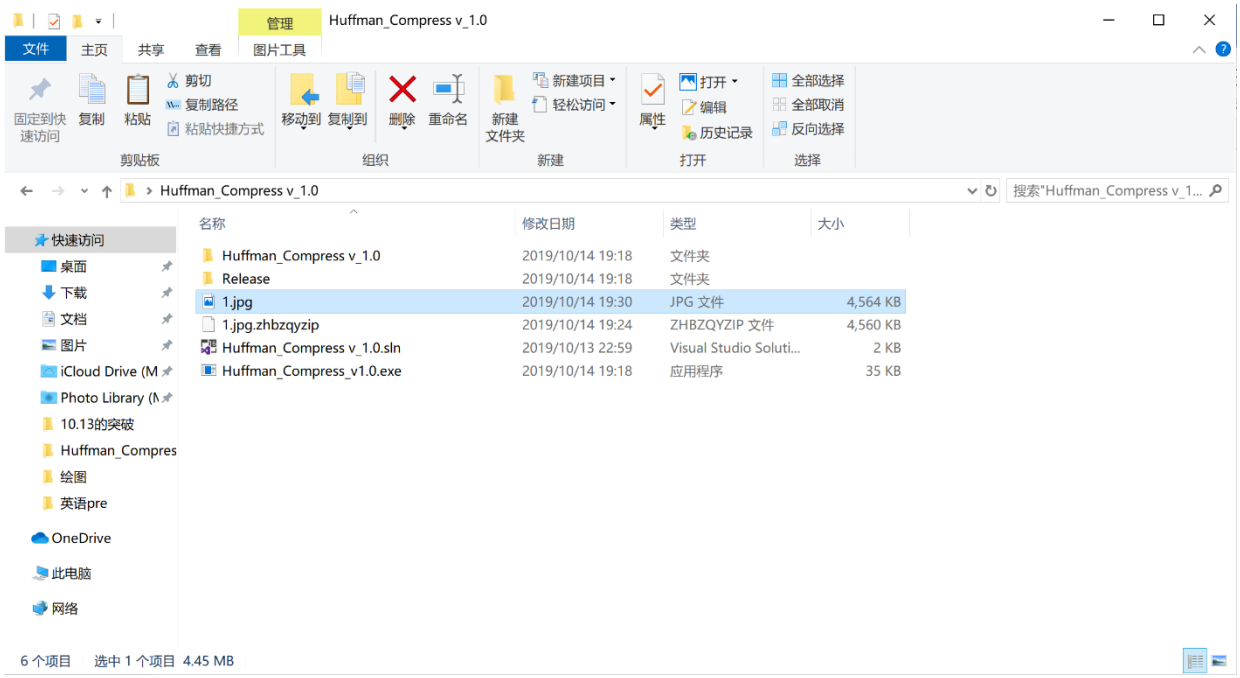
然后我们执行解压步骤，先将 1.jpg 删除，如图：



然后我们再次进入命令行，输入 exe 的名字，然后再输入 1.jpg.zhbzqyzip，按下回车键：

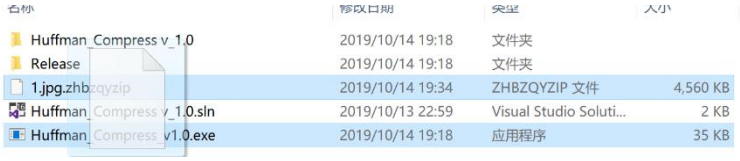


然后见证奇迹的时候到了！经过几秒钟的等待，我们返回文件夹：



打开 1. jpg，帅气的大佬回来了。于是我们完成了无损解压工作。

第二种运行程序的方法操作起来相对简单很多，只要直接将待处理的文件用鼠标拖动至 exe 文件处就行了：



当然为了证明是无损解压，用将原文件和解压后的文件按字节比较就行了，我写了一个 tset.cpp，也证明了这是无损解压，在这里就不赘述了。

四、 总结和讨论

这个项目是近期比较难的项目，我们的分工是这样的，张洪宾负责构建树，压缩和实验报告，张千玉负责写解压部分，但是我们实际操作起来发现两个人的合作其实不太适应，特别是两个人对同一个算法的理解不太一样，对接口的设计不一样的时候，就会出现代码不兼容的情况，总体来说第一次做合作项目才明白合作的重要性，也明白了为什么要写好注释，要让代码可以让别人看懂，只有编程能力是远远不够的，要学会与别人合作，写出让人可以理解的代码，才是一个优秀程序员应该追求的事情。

当然这个项目还是很有意思的，虽然我们熬了很多夜，为它掉了很多头发，但是当压缩的文件成功解压的那一刻，真的很 amazing。

但是我们还是有很多不足之处的，比如对图片和 pdf 的压缩效率比较低，而且对大的文件压缩效率不高，解压的时候也可能会因为堆内存不足而程序崩溃解压失败，所以如果有时间我会将这个程序优化做出 2.0 版。

五、 参考文献

《数据结构与算法实验实践教程》，乔海燕、蒋爱军、高集荣和刘晓铭编著，清华大学出版社出版，2012。