**一、问题描述**：

**名称**：基于哈夫曼编码的文件压缩解压

**目的**：利用哈夫曼编码压缩存储文件，节省空间

**输入**：任何格式的文件（压缩）或压缩文件（解压）

**输出**：压缩文件或解压后的原文件

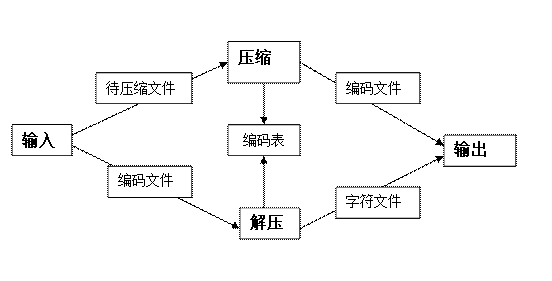
**功能**：利用哈夫曼编码压缩解压文件

**性能**：快速

**二、问题的初步讨论**：

为了建立哈夫曼树，首先扫描源文件，统计每类字符出现的频度（出现的次数），然后根据字符频度建立哈夫曼树，接着根据哈夫曼树生成哈夫曼编码。再次扫描文件，每次读取8bits，根据“字符—编码”表，匹配编码，并将编码存入压缩文件，同时存入编码表。解压时，读取编码表，然后读取编码匹配编码表找到对应字符，存入文件，完成解压。

**三、总的UML协同图**：

**[](http://images.cnitblog.com/blog/646820/201407/202125137408439.gif)**

**四、文件读取方式和处理单元的分析**：

压缩解压的第一步就是读取文件，为了能够处理任何格式的文件，采用二进制方式读写文件。以一个无符号字符（unsigned char）的长度8位为处理单元，最多有256（0~255）种组合，即256类字符。

**五、字符频度扫描的分析**：

要建立哈夫曼树，先要得到各类字符的频度，我想到了两种扫描方案：

1、利用链表存储，每扫描到一类新字符就动态分配内存；

2、利用数组，静态分配256个空间，对应256类字符，然后用下标随机存储。

链表在需要时才分配存储空间，可以节省内存，但是每加入一个新字符都要扫描一次链表，很费时；考虑到仅有256个字符种类，不是很多，使用静态数组，不会造成很大的空间浪费，而可以用数组的下标匹配字符，不需扫描数组就可以找到每类字符的位置，达到随机存储的目的，效率有很大的提高。当然，不一定每类字符都出现，所以，统计完后，需要排序，将字符频度为零的结点剔除。

我定义的数组类似这样：Node array[CHAR\_KINDS]，其中CHAR\_KINDS为8位无符号字符对应的256（0~255）种不同组合，这样每扫描到一个字符，直接将字符作为下标，就可以找到字符的位置。

**六、建立哈夫曼树的分析**：

哈夫曼树为二叉树，树结点含有权重（在这里为字符频度，同时也要把频度相关联的字符保存在结点中）、左右孩子、双亲等信息。

考虑到建立哈夫曼树所需结点会比较多，也比较大，如果静态分配，会浪费很大空间，故我们打算用动态分配的方法，并且，为了利用数组的随机访问特性，也将所需的所有树节点一次性动态分配，保证其内存的连续性。另外，结点中存储编码的域，由于长度不定，也动态分配内存。

**6.1、这时，针对上面的字符扫描结点就要做一些改动**：

将其定义成临时结点TmpNode，这个结点仅保存字符及对应频度，也用动态分配，但是一次性分配256个空间，统计并将信息转移到树结点后，就将这256个空间释放，既利用了数组的随机访问，也避免了空间的浪费。

**七、生成哈夫曼编码的分析**：

每类字符对应一串编码，故从叶子结点（字符所在结点）由下往上生成每类字符对应的编码，左‘0’，右‘1’。为了得到正向的编码，设置一个编码缓存数组，从后往前保存，然后从前往后拷贝到叶子结点对应编码域中，根据上面“建立哈夫曼树的协商”的约定，需要根据得到的编码长度为编码域分配空间。对于缓存数组的大小，由于字符种类最多为256种，构建的哈夫曼树最多有256个叶子结点，树的深度最大为255，故编码最长为255，所以分配256个空间，最后一位用于保存结束标志。

**八、文件压缩的分析**：

上面协定以8位的字符为单元编码，这里压缩当然也以8位为处理单元。

首先将字符及种类和编码（编码表）存储于压缩文件中，供解压时使用。

然后以二进制打开源文件，每次读取一个8位的无符号字符，循环扫描匹配存储于哈夫曼树节点中的编码信息。

由于编码长度不定，故需要一个编码缓存，待编码满足8位时才写入，文件结束时缓存中可能不足8位，在后面补0，凑足8位写入，并将编码的长度随后存入文件。

在哈夫曼树节点中，编码的每一位都是以字符形式保存的，占用空间很大，不可以直接写入压缩文件，故需要转为二进制形式写入；至于如何实现，可以定义一个函数，将保存编码的字符数组转为二进制，但是比较麻烦，效率也不高；正好，可以利用C语言提供的位操作（与、或、移位）来实现，每匹配一位，用“或”操作存入低位，并左移一位，为下一位腾出空间，依次循环，满足8位就写入一次。

**8.1、压缩文件的存储结构**：

**[clip_image002](http://images.cnitblog.com/blog/646820/201407/202125141623610.gif)**

结构说明：字符种类用来判断读取的字符、频度序偶的个数，同时用来计算哈夫曼结点的个数；文件长度用来控制解码生成的字符个数，即判断解码结束。

**九、文件解压的分析**：

以二进制方式打开压缩文件，首先将文件前端的字符种类数读取出来，据此动态分配足够空间，然后将随后的字符—编码表读取处理保存到动态分配的结点中，然后以8位为处理单元，依次读取随后的编码匹配对应的字符，这里对比编码依然用在文件压缩中所用的方法，就是用C语言的位操作，同0x80与操作，判断8bits字符的最高位是否为‘1’，对比一位后，左移一位，将最高位移除，次高位移到最高位，依次对比。这次是从编码到字符反向匹配，与压缩时有一点不同，需要用读取的编码逐位与编码表中的编码进行对比，对比一位后，增加一位再对比，而且每次对比都是一个循环（与每个字符的编码对比），效率很低。

于是，我思考另外的方法，可以将哈夫曼树保存到文件中，解码时，从树根到叶子对比编码，只要一次遍历就可以找到编码对应的存于叶子结点中的字符，极大提高了效率。

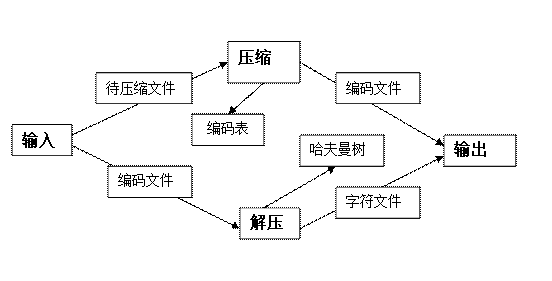
然而，我们发现树结点中有字符、编码、左右孩子、双亲，而且孩子和双亲还必须是整型的（树节点最多为256\*2-1=511个），占用空间很大，会导致压缩文件变大，这是不可取的，因为我们的目的就是压缩文件。

我们进一步考虑，可以仅存储字符及对应频度（频度为unsigned long，一般情况下与int占用空间一样，同为4个字节），解码时读取数据重建哈夫曼树，这样就解决了空间问题。

虽然重建哈夫曼树（双重循环，每个循环的次数最大为511）也要花费一定的时间，但是相对上面的与编码表匹配（每位编码都要循环匹配所有字符（最多为256种）一次，而总的编码位数一般很大，且随着文件变大而增长）所花费的时间更少。

**9.1由于解压的方式变了，在这里要对上面的协商作一些修改**：

1、修改后的总UML协同图：

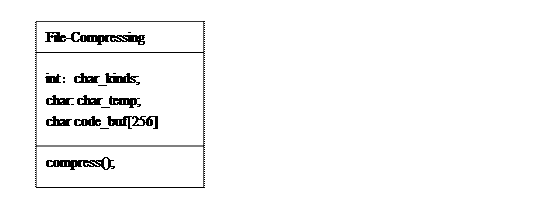
**[](http://images.cnitblog.com/blog/646820/201407/202125147878252.gif)**

2、在文件压缩时，就不需要保存编码表，改为保存字符及对应权重。

3、在文件压缩时，处理最后不足8位的编码后，不再需要保存编码的长度，因为解压时从树根向下匹配，到达叶子就停止（所有叶子结点都在连续分配的树结点空间的低端，故可以用结点下标判断是否到达叶子结点），不会超过而读取最后的无效编码。

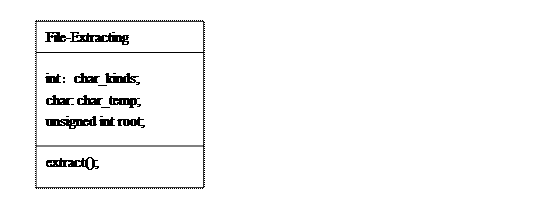
**十、定义所需类：**

**10.1、文件压缩所需类**：

**[](http://images.cnitblog.com/blog/646820/201407/202125152716937.gif)**

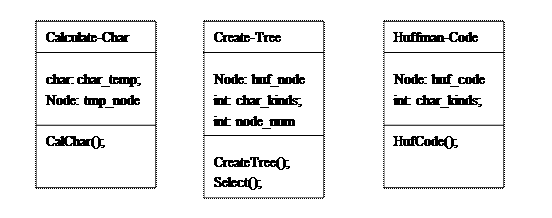
行为说明：char\_kinds保存出现的字符种类；char\_temp用来保暂存字符；code\_buf暂存匹配出来的编码；compres()是主压缩函数，接收两个文件名，一个输入，可以是任何格式的待压缩文件，一个输出，为压缩后的编码文件；

**10.2、文件解压所需类**：

**[](http://images.cnitblog.com/blog/646820/201407/202125157718851.gif)**

行为说明：char\_kinds保存出现的字符种类；char\_temp用来保暂存字符；root保存解码时的当前结点索引，用来判断是否达到叶子结点；extract()是主压缩函数，接收两个文件名，一个输入，为压缩后的编码文件，一个输出，为解码后的原文件；

**10.3、其他重要类**：

**[](http://images.cnitblog.com/blog/646820/201407/202125165063993.gif)**

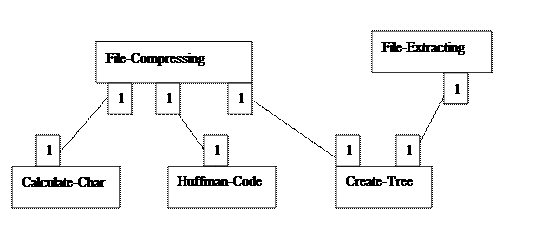
行为说明：

1、tmp\_nodes用来保存字符频度，动态一次性分配256个空间，统计后删除；CalChar()用于生成8位的256个字符及对应频度（出现次数）；

2、node\_num保存树结点总数，CreateTree()建立哈夫曼树，select()函数用来找最小的两个结点；

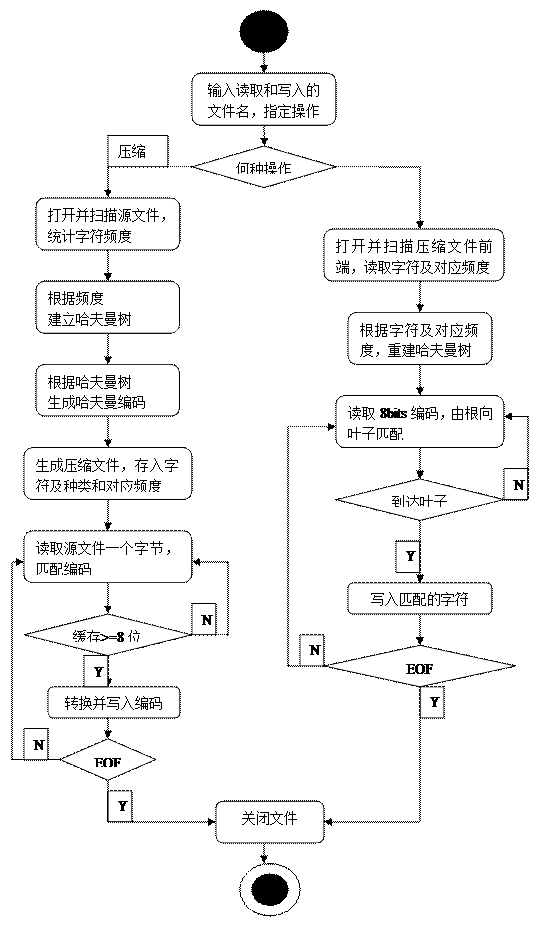
3、huf\_node树结点用来保存编码信息，HufCode()生成哈夫曼编码；

**10.4、类的关联图**：

**[](http://images.cnitblog.com/blog/646820/201407/202125169432394.gif)**

行为说明：CreateTree()和HufCode()供compress调用，前者建立哈夫曼树，后者生成哈夫曼编码；CreateTree()供extrac()调用，重建哈夫曼树，用于解码；

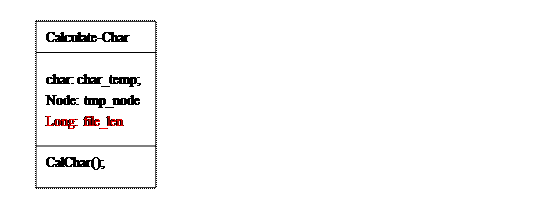
**十一、编码行为状态图**：

**[clip_image008](http://images.cnitblog.com/blog/646820/201407/202125174746064.gif)[](http://images.cnitblog.com/blog/646820/201407/202125179282992.gif)**

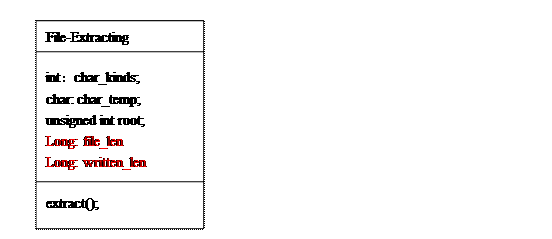
后来我在初步编码时，发现一些问题：解码后无法得到完全正确的源文件，经过排查，发现以EOF判断压缩文件的结束不可取，因为压缩文件是二进制文件，而EOF一般用来判断非二进制文件的结束，所以我们加入了文件长度来控制。

**11.1、于是上面的协商需要一些改动**：

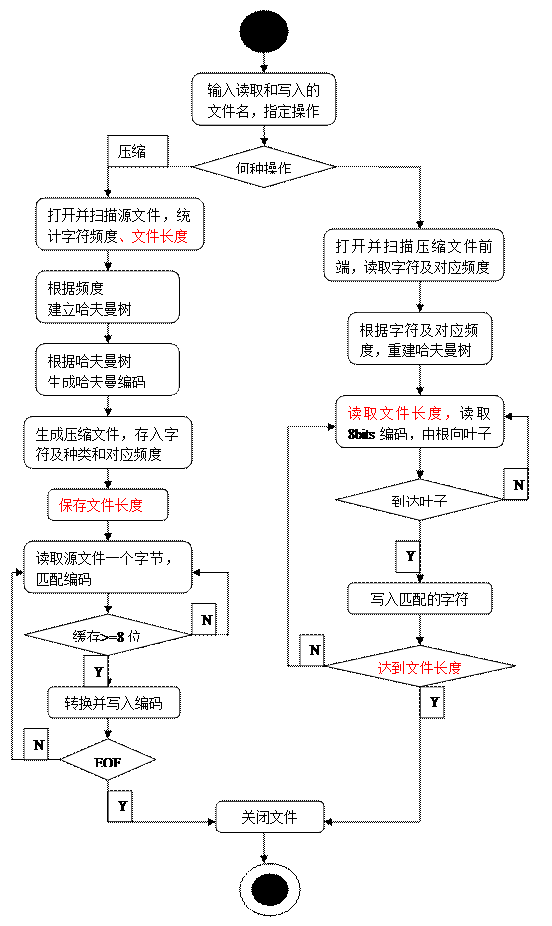
1、修改后的字符统计类：

**[](http://images.cnitblog.com/blog/646820/201407/202125183651393.gif)**

2、修改后的文件压缩类：

**[](http://images.cnitblog.com/blog/646820/201407/202125187712335.gif)**

3修改后的编码行为状态图：

**[clip_image008[1]](http://images.cnitblog.com/blog/646820/201407/202125191783277.gif)[](http://images.cnitblog.com/blog/646820/201407/202125198037919.gif)**

**十二、函数实现：**

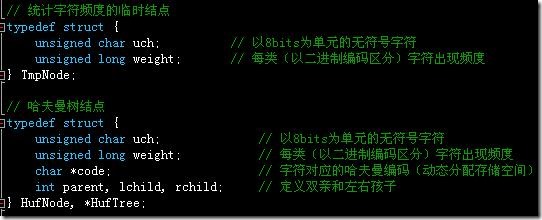
**12.1、实现语言及编码环境：**

**实现语言**：C语言，兼容嵌入式，运行效率高

**编码环境**：XP+VS2010(debug模式)

**12.2、结构体及函数定义**：

两个重要的结点结构体：

**[](http://images.cnitblog.com/blog/646820/201407/202125203495819.jpg)**

三个函数用于建立哈夫曼树和生成哈夫曼编码：

**[clip_image016](http://images.cnitblog.com/blog/646820/201407/202125209597232.jpg)**

**[clip_image018](http://images.cnitblog.com/blog/646820/201407/202125215218361.jpg)**

**[clip_image020](http://images.cnitblog.com/blog/646820/201407/202125220533032.jpg)**

两个主要函数——压缩解压函数：

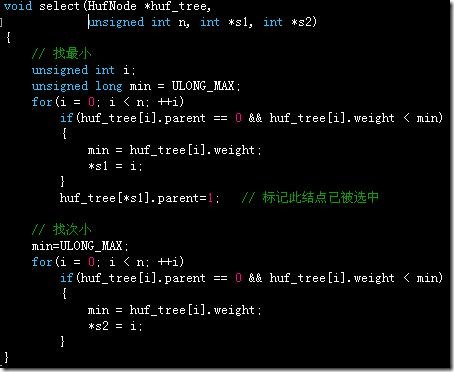
**[clip_image022](http://images.cnitblog.com/blog/646820/201407/202125225846702.jpg)**

**[clip_image024](http://images.cnitblog.com/blog/646820/201407/202125230848616.jpg)**

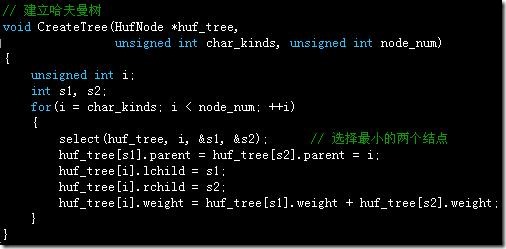
**12.3、函数说明**：

**12.3.1、其他函数：**

Select函数供CreateTree函数调用，找两个最小的结点，找到第一个后需要将其parent设为‘1’（初始化后为‘0’）表明此结点已被选中：

**[](http://images.cnitblog.com/blog/646820/201407/202125236782502.jpg)**

建立哈夫曼树，每次用select()函数找两个最小结点：

**[](http://images.cnitblog.com/blog/646820/201407/202125244909088.jpg)**

生成哈夫曼编码，由叶子到根反向生成编码，左‘0’，右‘1’，每个code域的内存动态分配：

**[](http://images.cnitblog.com/blog/646820/201407/202125252719215.jpg)**

**12.3.2、压缩函数中的几个部分的说明**：

动态分配256个暂存结点，用下标索引统计字符频度：

**[](http://images.cnitblog.com/blog/646820/201407/202125258964858.jpg)**

这里以feof来判断文件结束，是由于eof判断的文件类型比较局限，而feof在读完最后一个字节之后，再次读文件时才会设置结束标志，所以需要在while循环之前读一次，然后每次在循环的最后读取文件，这样可以正确判断文件结束；以位操作来匹配编码，每次存入最低位，然后左移一位，依次循环处理，满8位保存一次：

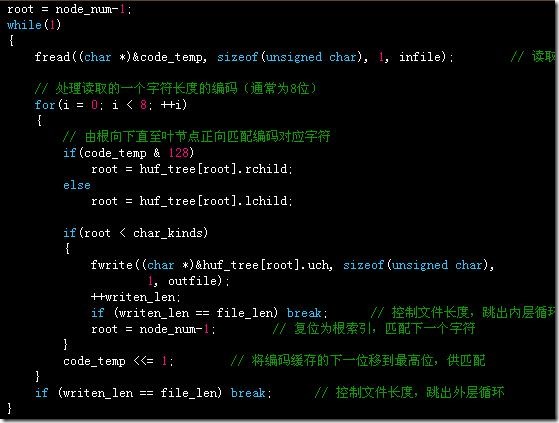
**[](http://images.cnitblog.com/blog/646820/201407/202125265997243.jpg)**

最后缓冲中不足8位，补0凑足8位（左移）：

**[](http://images.cnitblog.com/blog/646820/201407/202125273817370.jpg)**

**12.3.3、解压函数中的说明**：

压缩文件为二进制文件，feof在这里无法正确判断结束，故用一个死循环处理编码，以压缩时存储的文件长度来控制循环的结束。每当root小于char\_kinds，就匹配到了一个字符，是因为字符的下标范围是0~char\_kinds-1。

**[](http://images.cnitblog.com/blog/646820/201407/202125281934957.jpg)**

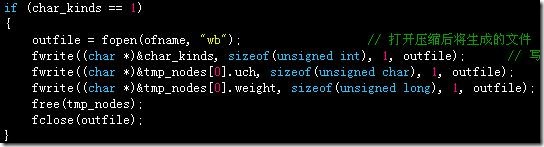
**十三、程序健壮性考虑**：

**13.1、字符种类为‘1’：**

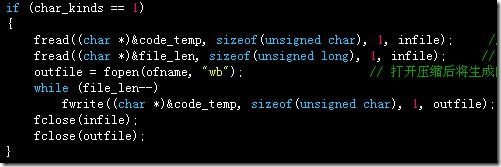
当字符种类为‘1’时，只有一个哈夫曼结点，无法构造哈夫曼编码，但是可以直接处理，依次保存字符种类数、字符、字符频度（此时就是文件长度）即可，解压时仍然先读取字符种类数，为‘1’则特殊处理，读取字符和频度（此时就是文件长度），利用频度控制循环，输出字符到文件即可，此时压缩文件的存储结构为：

**[clip_image039](http://images.cnitblog.com/blog/646820/201407/202125288189598.gif)**

在压缩函数前部加入特殊情况的判断和处理：

**[](http://images.cnitblog.com/blog/646820/201407/202125293182513.jpg)**

在解压函数前部加入特殊情况判断和处理：

**[](http://images.cnitblog.com/blog/646820/201407/202125301467626.jpg)**

**13.2、输入文件不存在：**

由于压缩或解压时，输入文件必须是存在的，而用户可能会输错，因此有必要加入输入文件的存在性进行判断，防止文件不存在而导致程序异常退出：

1、将压缩解压的返回值改为int：

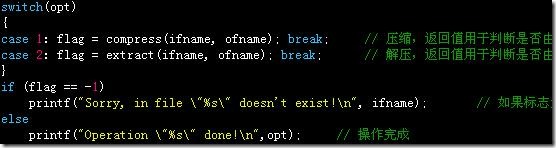
**[clip_image045](http://images.cnitblog.com/blog/646820/201407/202125307876498.jpg)**

**[clip_image047](http://images.cnitblog.com/blog/646820/201407/202125314433897.jpg)**

2、在压缩和解压函数中加入：

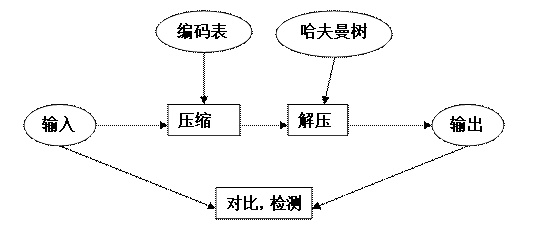
**[clip_image049](http://images.cnitblog.com/blog/646820/201407/202125319435811.jpg)**

3、在main函数中加入压缩解压函数是否异常退出的判断：

**[](http://images.cnitblog.com/blog/646820/201407/202125325062238.jpg)**

**十四、系统测试：**

**14.1、测试流程图：**

**[](http://images.cnitblog.com/blog/646820/201407/202125332094623.gif)**

**14.2、代码运行测试**：

**14.2.1、使用说明：**

（编译链接生成的可执行文件为：hufzip.exe）

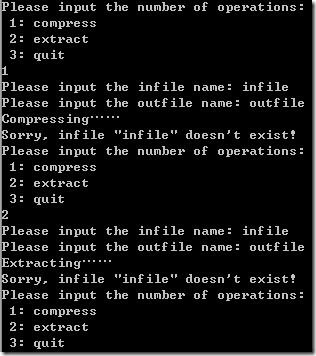
双击hufzip.exe运行，输入所选择操作类型的数字代号：

1：compress（压缩）

2：extract（解压）

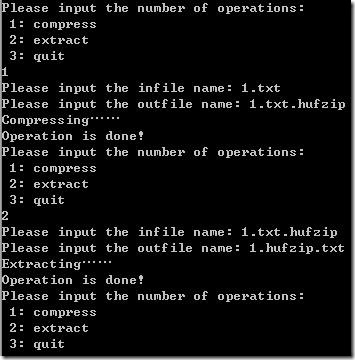
3：quit（退出）

然后提示输入源文件和目标文件，可以输入完整的路径名加文件名，也可以仅输入一个文件名（默认在当前运行目录下寻找），如果不小心输错源文件名或源文件不存在，将提示出错，然后可以再次输入，如下图所示：

**[](http://images.cnitblog.com/blog/646820/201407/202125337562523.jpg)**

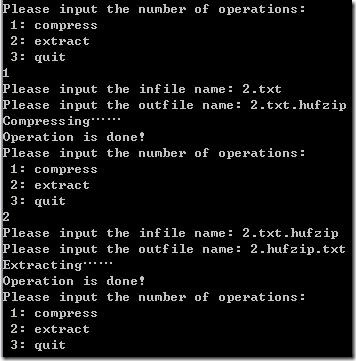
**14.2.2、测试的几个文件**：

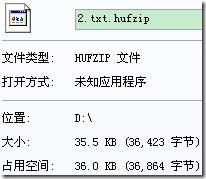
“1.txt”中为全为字符‘a’（共1024\*1024个），由于只有一种字符，压缩文件只保存了字符种类（4byte）、字符（1byte）和字符频度（4byte），故为9字节，控制台及文件压缩情况如下（1.txt.hufzip为压缩文件，1.hufzip.txt为解压后的文件）：

**[](http://images.cnitblog.com/blog/646820/201407/202125345069893.jpg)**

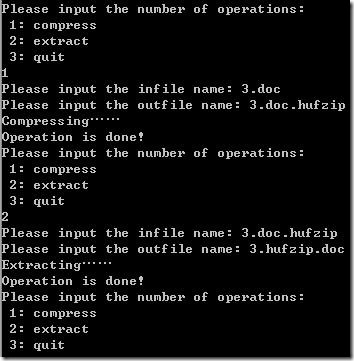
**[](http://images.cnitblog.com/blog/646820/201407/202125355372180.jpg)** **[](http://images.cnitblog.com/blog/646820/201407/202125363969050.jpg)[](http://images.cnitblog.com/blog/646820/201407/202125368817735.jpg)**

“2.txt”为0~255的整数，其中0出现1次，1出现2次，……，255出现256次，其控制台及压缩情况如下（2.txt.hufzip为压缩文件，2.hufzip.txt为解压后的文件）：

**[](http://images.cnitblog.com/blog/646820/201407/202125375216606.jpg)**

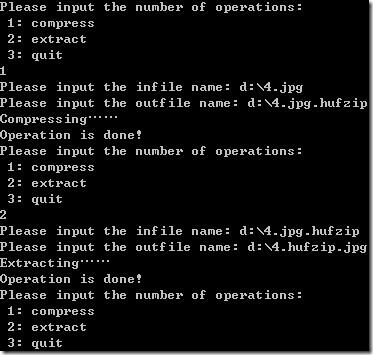
**[](http://images.cnitblog.com/blog/646820/201407/202125383189963.jpg)[](http://images.cnitblog.com/blog/646820/201407/202125388341405.jpg)** **[](http://images.cnitblog.com/blog/646820/201407/202125393659776.jpg)**

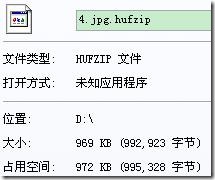
“3.doc”为一个随意的word文档，其控制台及压缩情况如下（3.doc.hufzip为压缩文件，3.hufzip.doc为解压后的文件）：

**[](http://images.cnitblog.com/blog/646820/201407/202125399286204.jpg)**

**[](http://images.cnitblog.com/blog/646820/201407/202125406151062.jpg)** **[](http://images.cnitblog.com/blog/646820/201407/202125414594703.jpg)[](http://images.cnitblog.com/blog/646820/201407/202125420379061.jpg)**

“4.jpg”是一个图像文件，输入绝对路径对4.jpg进行压缩，其控制台及压缩情况如下（4.jpg.hufzip为压缩文件，4.hufzip.jpg为解压后的文件）：

**[](http://images.cnitblog.com/blog/646820/201407/202125426783231.jpg)**

**[](http://images.cnitblog.com/blog/646820/201407/202125433968845.jpg)****[](http://images.cnitblog.com/blog/646820/201407/202125438817529.jpg)** **[](http://images.cnitblog.com/blog/646820/201407/202125444438658.jpg)**

**14.2.3、由上面的几种文件的压缩前后的对比可以得出**：

哈夫曼编码对文本文件，一般可以达到大约2:1的压缩比，特别是有规律的文本文件，可以达到高于2:1的压缩比，而对于图像等特殊文件压缩比几乎为1:1，效果不理想。