**Huffman编码实验报告**

1. **功能实现**
2. **代码解释**

实验采用python语言。主要分为Huffman树、文件读写以及编码译码三部分。具体解释主要在代码注释里。

（1）Huffman树

首先定义Huffman结点类，如图1所示。

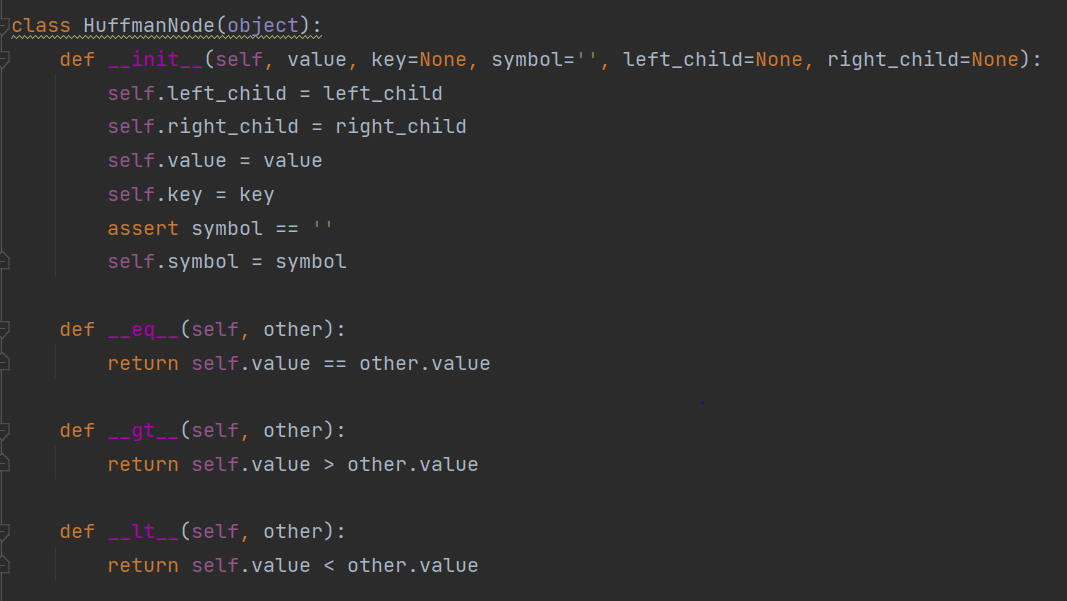


图1 Huffman结点类

然后进行Huffman树的创建，如图2所示。调用queue库，借助优先级队列实现对符号频率的排序。每一次选频率最小的两个值，作为二叉树的两个叶子节点，将和作为它们的父节点，这两个叶子节点不再参与比较，新的父节点参与比较



图 2 创建Huffman树

最后是在编码过程中访问树并不断记录编码。左孩子记0，右孩子记1。

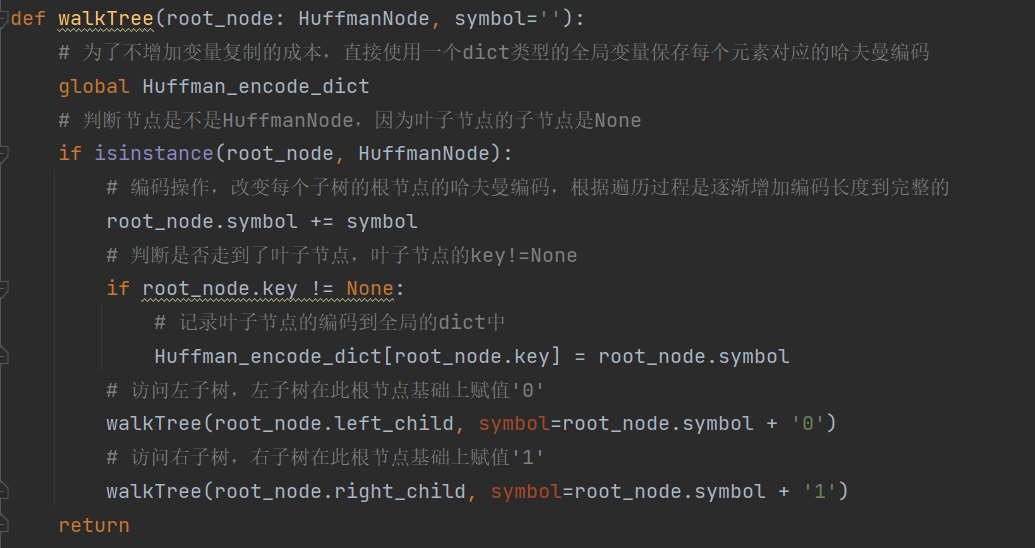
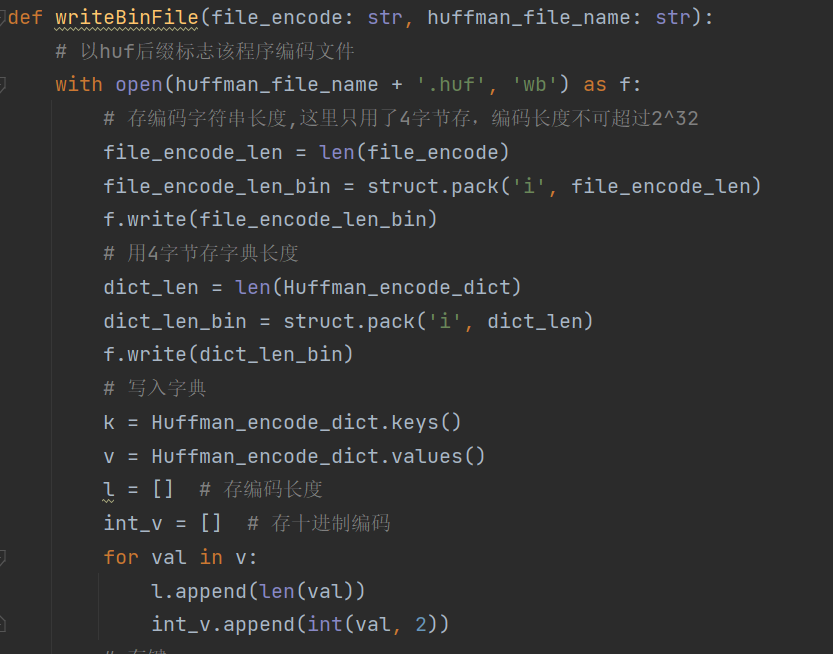


图 3 填充Huffman树

1. 文件读写

文件读写都采用二进制字节流形式。根据实验要求，写文件（即生成压缩文件）时将Huffman字典也拆开一并写入。采用struct.pack打包不同类型数据。



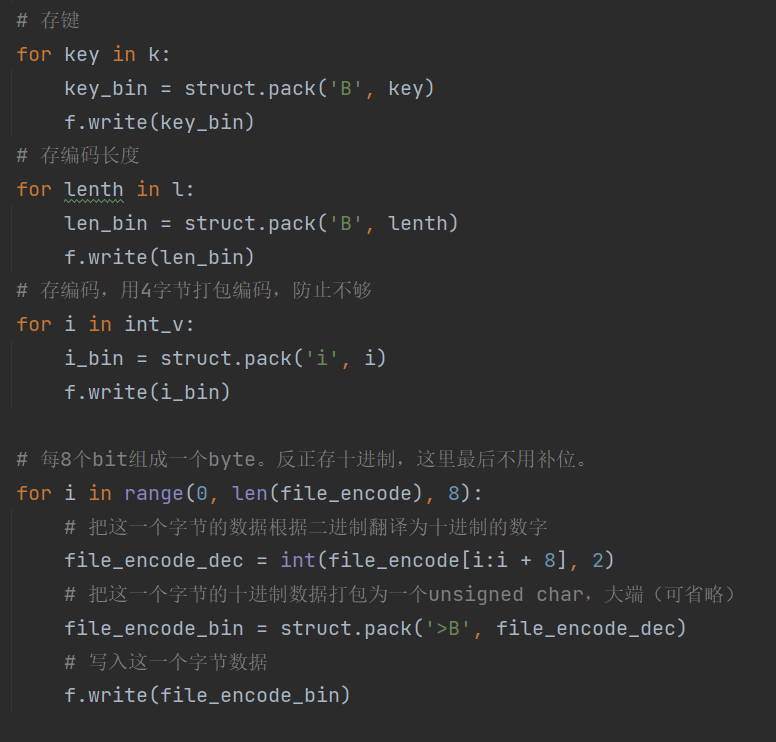


图 4 写二进制流文件

如图5，该函数用于读取编码文件中的二进制数据。根据写文件时打包格式与顺序解包，注意单个字节型解出来是十进制数，转二进制要注意补0到8位。最后一字节要去掉高位多余0，因为写的时候也没补最后一字节。



图 5 读二进制流文件

1. 编码译码

编码时将原文件每个字符在编码字典中查找，获得编码字符串。

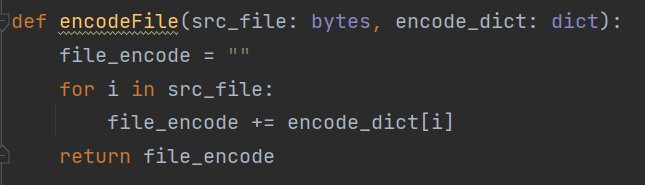


图 6 编码原文件

译码稍麻烦，先将编码字典颠倒过来，获得键为01编码，值为字符的译码字典。按01串长度对比输入的二进制串看是否相同。由于Huffman是前缀码，只要一段编码与译码字典中某键长度相同且相等，则可确定该段编码为该键对应符号编码而来。返回字节流方便再写入译码文件。

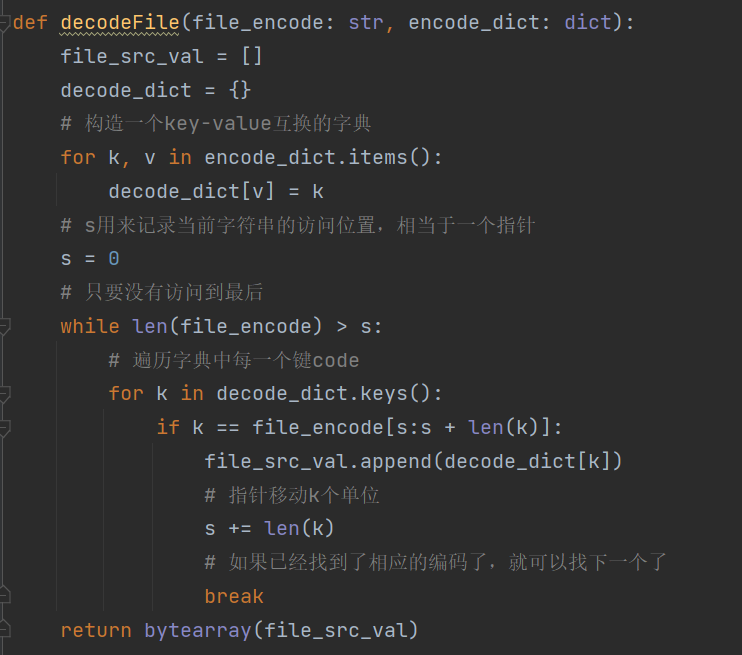


图 7 译码二进制数据

真正实现完全编码功能的函数为encode和decode，集成上述函数功能，且涉及用户交互和鲁棒性处理。encode包括频率字典生成，构造Huffman树，遍历Huffman树，原文件编码，写二进制流文件以及一些结果计算展示。译码函数decode需要读取字典各部分信息并恢复，恢复二进制流数据，译码二进制数据并恢复原文件。

1. **运行测试截图**

本次实验实现了使用Huffman码对任意文件进行编码译码。下面是对testpic.bmp进行编码译码的测试截图。本次实验所有原文件都在testfiles文件夹中。

编码结果可在“中间文件”文件夹中查看testpic\_encode.huf。

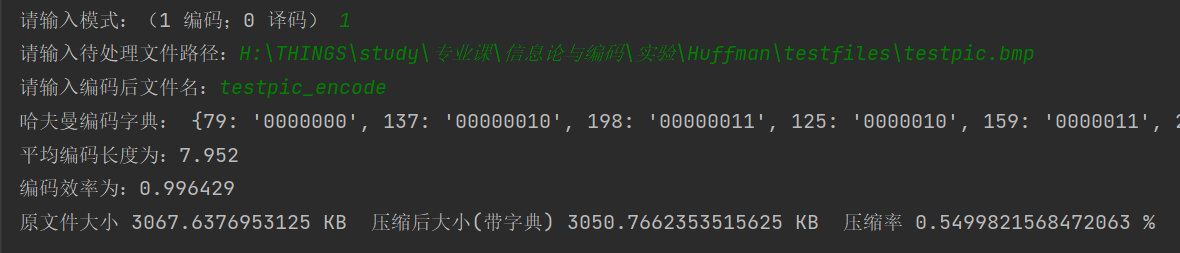


图 8 编码测试

译码结果可在“结果文件”文件夹中查看testpic\_decode.bmp。



图 9 译码测试

如图10，编码前、译码后文件的SHA256值均为

9c7d41230dd8d84996cdd33a62ef76e2be845394923f18dc64a1f22727ee1e39

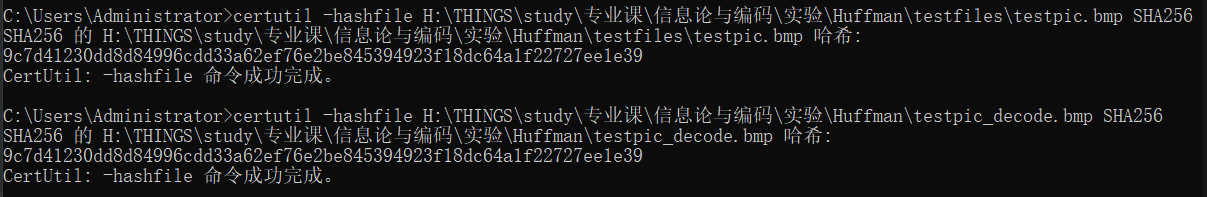


图 10 SHA256值相同

（另外，这张图本来是jpg格式，我直接将后缀改成了bmp，由结果来看改这个后缀没什么用处。）

1. **代码风格与用户交互**

代码风格上，本代码有清楚的变量命名、恰当的注释和合理的结构设计。变量命名均采用下划线分隔单词，可顾名思义。定义函数时均已写明相关参数类型，便于查看修改。重要功能语句均有注释解释，而且注释了一些调试用print代码，需要的话可取消注释查看中间结果。先定义类，再定义相关函数，按Huffman树、二进制流文件读写、编码译码顺序，结构清晰明了。

如图8、9所示，本代码设计了CLI终端交互。可以输入1或0选择编码或译码模式。用户需要输入待处理的文件路径，可自定义文件名。译码模式时用户输入文件名时需要加上后缀便于恢复原文件。编码和译码时都会显示Huffman编码字典。编码时还会显示平均编码长度、编码效率、原文件大小、压缩后大小和压缩率。交互代码如图11所示。

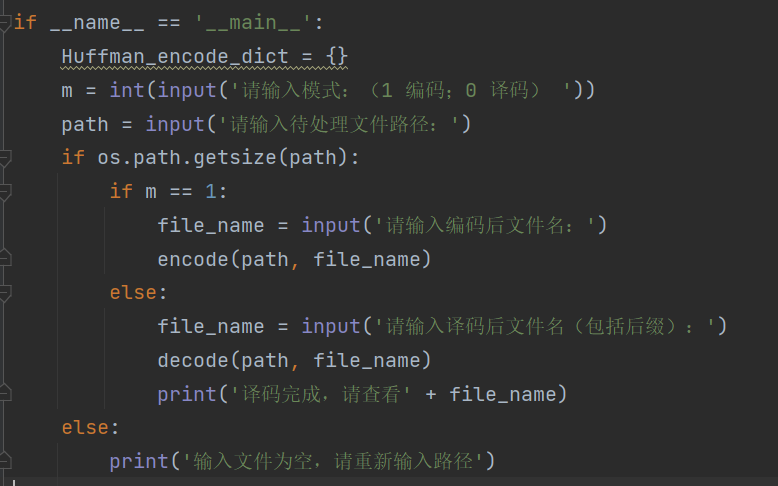


图 11 交互部分代码

1. **代码鲁棒性与安全性**

本程序考虑了一些边界情况和错误。

空文件无需编译码。先通过os.path.getsize()获取文件大小，为空则不进行编码译码，直接返回错误提示，如图12所示。



图 12 空文件测试

若文件仅含有一种字节，则Huffman字典编码为空，无法编码译码。若频率字典长度为1，说明文件只有一种字节。如图13所示，此时直接令该字节编码为‘0’，将文件中所有字节都变成‘0’，不进行正常Huffman编码译码操作。

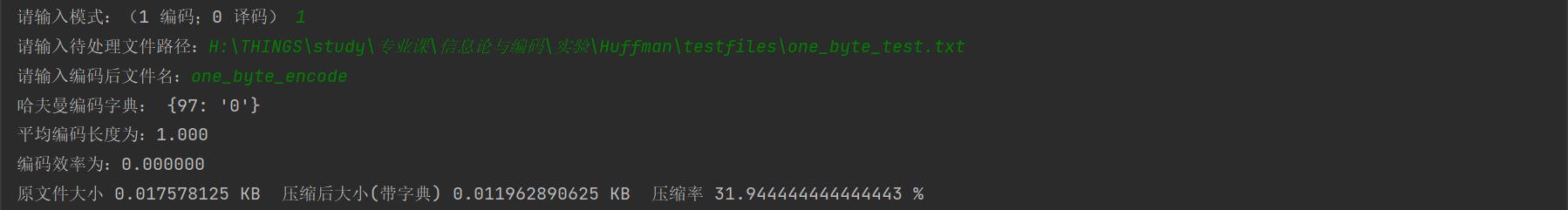


图 13 单字节测试

本程序不能解译非本程序编码的文件。为标识本程序产生的编码文件，一种方法是在写文件时一起写入标识，但这样做会使文件变大，压缩效率变低。因此本程序采取特定后缀的方式标识编码文件。编码时只要输入文件名而不要后缀，是因为本程序会自动使编码文件后缀为.huf。如图14所示，译码时传入文件路径最后后缀不是.huf，则可确定文件不能被本程序译码，返回错误提示。

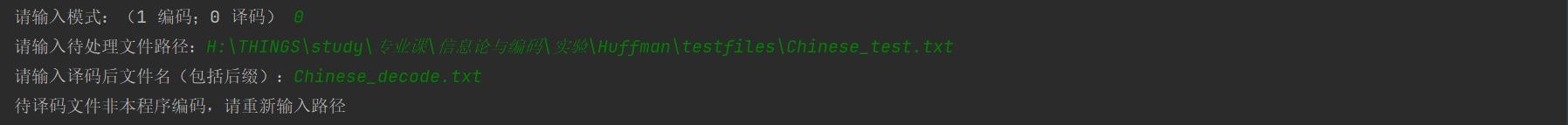


图 14 文件标识测试

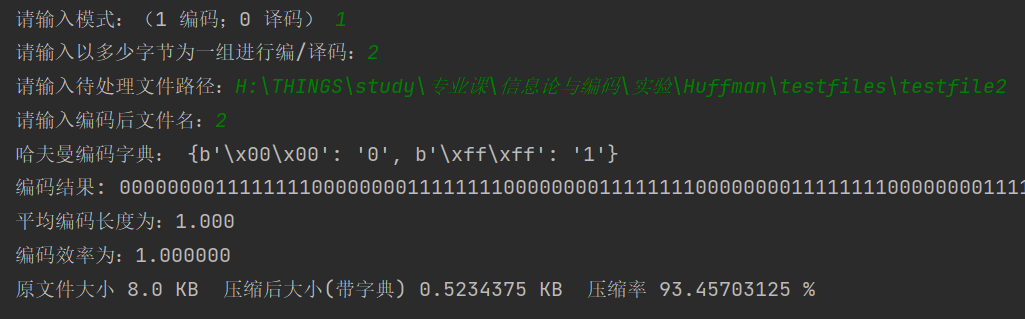
1. **实践问题**
2. **重复性的文件结构**

编码testfile1结果如图15所示。由图可见原文件大小为64KB，压缩后反而变成了65.5KB。原来的文件结构就是所有符号等概，编码完平均码长还是8，和原来没有变化，对于该文件Huffman编码不能起到压缩作用。由于还要往编码文件里写入字典等信息，反而使编码后文件更大了。



图 15 testfile1编码结果

分别以2、4、8、13、128字节为单位对testfile2编码，结果如图16-20所示。

图 16 2字节编码结果

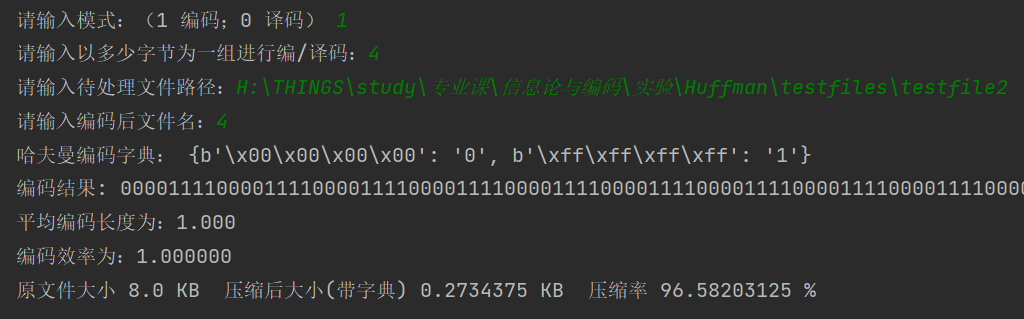


图 17 4字节编码结果

图 18 8字节编码结果



图 19 13字节编码结果



图 20 128字节编码结果

这里压缩率的定义是编码前后文件大小变化除以原文件大小。由以上结果可见，以多字节为单位字符对testfile2编码可大大提高文件压缩率，这是因为testfile2有高度连续重复结构，多字节将被编码成更小的字符串。进一步，当我们以2的幂次个字节为单位时，编码效率能达到100%，即平均码长和熵相等。非2的幂次时，如13字节反而压缩率比8字节低，是testfile2以32为周期重复内容的特殊结构导致的。

以2字节为单位已经有很高的压缩率了，经试验2、4、8、16字节压缩率依次上升。16字节最大，压缩率达98.9%。但这也不意味着单位字符越大越好，当以128字节为单位时，压缩率大幅降低，连2字节都不如了。这是因为我们需要往编码文件里存字典等信息，单位字符越大，Huffman字典越大，占用了很多空间。经试验，32字节再继续增大2的幂次，压缩率将保持87%，因为之后都是只有1种字符的情况。这启示我们单位字符的大小选取最好是原文件重复内容周期的最大因数。

1. **不同格式的压缩**

（1）图片压缩

bmp格式压缩文件结果如图21所示。

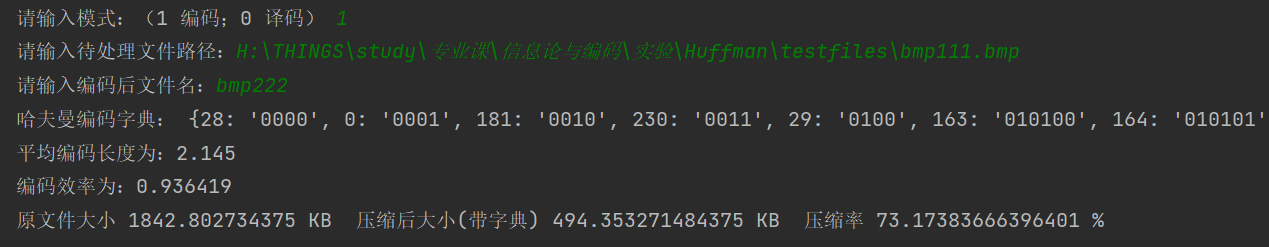


图 21 bmp压缩结果

同一张图jpeg格式压缩结果如图22所示。

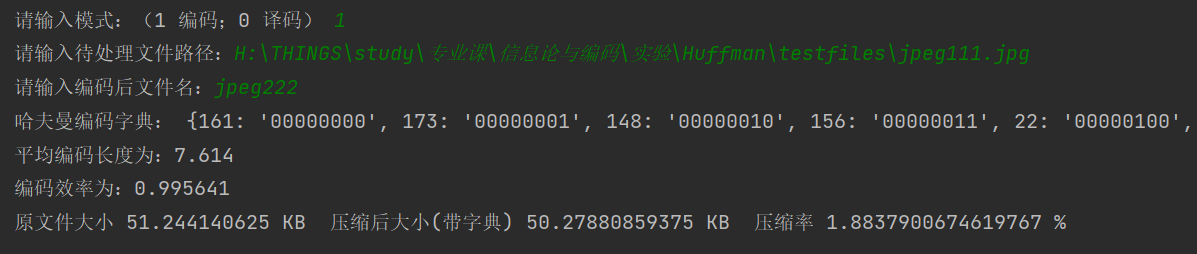


图 22 jpeg压缩结果

由结果可见bmp格式压缩率很高，而jpg格式压缩率极低。压缩前bmp文件就比jpg文件大，因为bmp是未压缩色域的格式，jpg在保存时已经压缩过了，部分像素值会有较小波动。测试文件有大量白色，因此bmp压缩时白色对应字符编码长度就会最短，从而大大压缩文件。而jpg由于本身的失真，即使同样是白色可能对应字符也不同，破坏了字符重复结构，因此压缩率很低。

1. 不同格式文件压缩

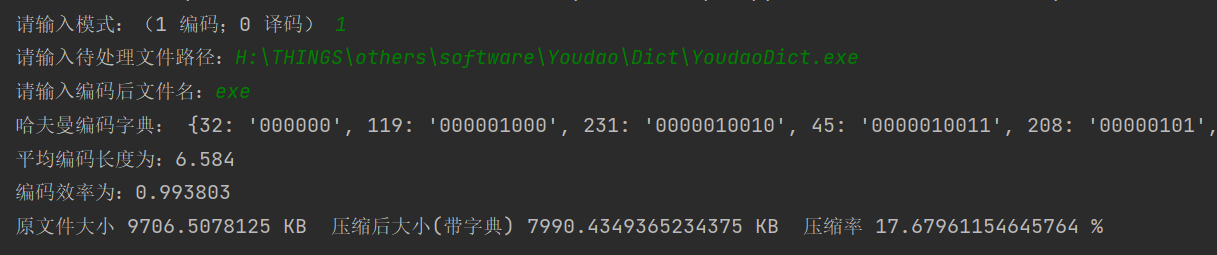


图 24 exe压缩结果



图 24 png压缩结果

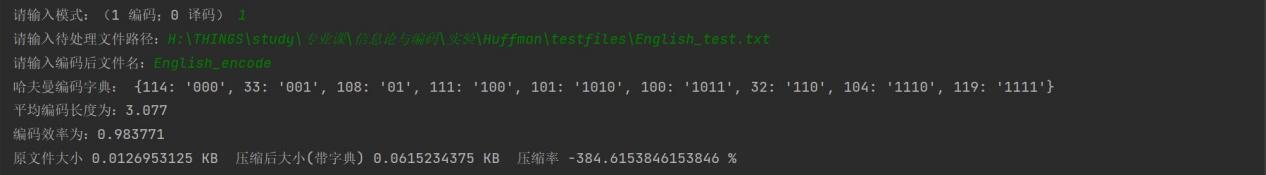


图 25 英文txt压缩结果



图 26 中文txt压缩结果

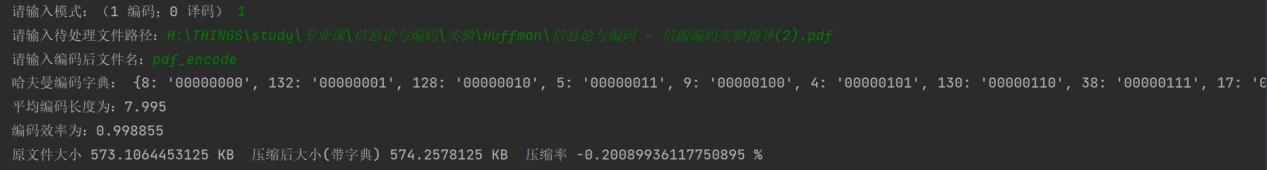


图 27 pdf压缩结果

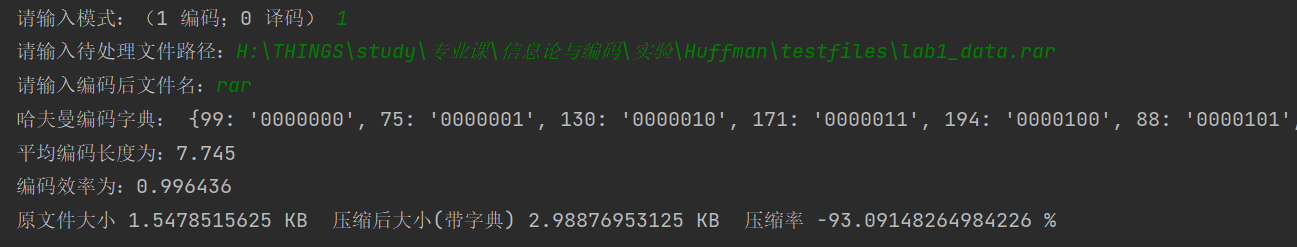


图 28 rar压缩结果

1. **黑洞**

对上一问中的bmp文件压缩10次，结果如图29所示。

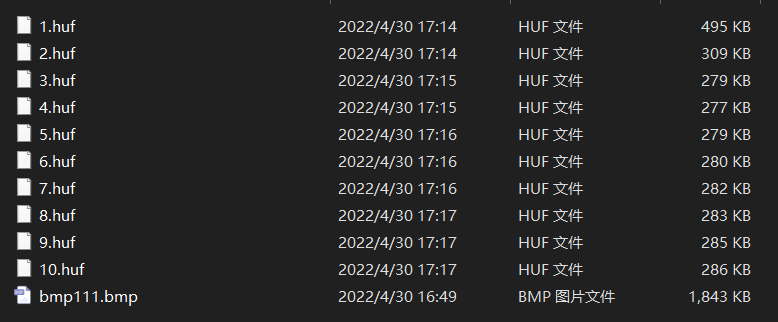


图 29 bmp压缩10次结果

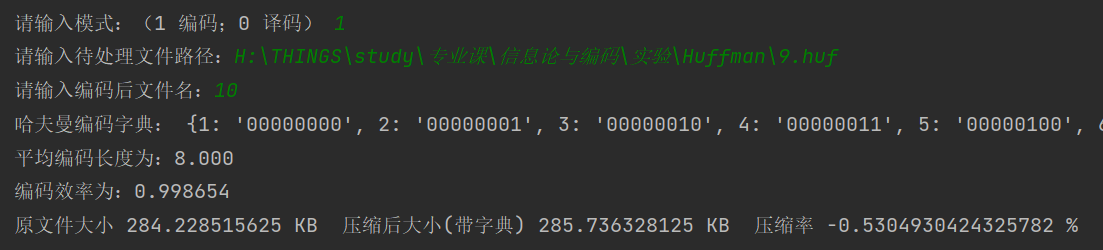


图 30 bmp压缩第10次结果

由图可见，压缩到第4次时文件最小，之后开始慢慢增大，文件并不会越压缩越小。这是因为信源熵是平均码长的下界，第四次及以后编码效率都接近1，已经不能再继续压缩下去了。至于后来越压越大，是因为每次编码还要存字典，之后每次基本都是-0.5%的压缩率。