第 9 章 图象的压缩编码，JPEG 压缩编码标准

在介绍图象的压缩编码之前，先考虑一个问题：为什么要压缩？其实这个问题不用我回答，你也能想得到。因为图象信息的数据量实在是太惊人了。举一个例子就明白：一张 A4(210mm×297mm) 幅面的照片，若用中等分辨率(300dpi)的扫描仪按真彩色扫描，其数据量为多少？让我们来计算一下：共有(300×210/25.4) ×(300×297/25.4)个象素，每个象素占 3 个字节，其数据量为 26M 字节，其数据量之大可见一斑了。

如今在 Internet 上，传统基于字符界面的应用逐渐被能够浏览图象信息的 WWW(World Wide Web)方式所取代。WWW 尽管漂亮，但是也带来了一个问题：图象信息的数据量太大了，本来就已经非常紧张的网络带宽变得更加不堪重负，使得 World Wide Web 变成了 World Wide Wait。

总之，大数据量的图象信息会给存储器的存储容量，通信干线信道的带宽，以及计算机的处理速度增加极大的压力。单纯靠增加存储器容量，提高信道带宽以及计算机的处理速度等方法来解决这个问题是不现实的，这时就要考虑压缩。

压缩的理论基础是信息论。从信息论的角度来看，压缩就是去掉信息中的冗余，即保留不确定的信息，去掉确定的信息(可推知的)，也就是用一种更接近信息本质的描述来代替原有冗余的描述。这个本质的东西就是信息量(即不确定因素)。

压缩可分为两大类：第一类压缩过程是可逆的，也就是说，从压缩后的图象能够完全恢复出原来的图象，信息没有任何丢失，称为无损压缩；第二类压缩过程是不可逆的，无法完全恢复出原图象，信息有一定的丢失，称为有损压缩。选择哪一类压缩，要折衷考虑，尽管我们希望能够无损压缩，但是通常有损压缩的压缩比(即原图象占的字节数与压缩后图象占的字节数之比，压缩比越大，说明压缩效率越高)比无损压缩的高。

图象压缩一般通过改变图象的表示方式来达到，因此压缩和编码是分不开的。图象压缩的主要应用是图象信息的传输和存储，可广泛地应用于广播电视、电视会议、计算机通讯、传真、多媒体系统、医学图象、卫星图象等领域。

压缩编码的方法有很多，主要分成以下四大类：(1)象素编码；(2)预测编码；(3)变换编码；(4)其它方法。

所谓象素编码是指，编码时对每个象素单独处理，不考虑象素之间的相关性。在象素编码中常用的几种方法有：(1)脉冲编码调制(Pulse Code Modulation，简称 PCM)；(2)熵编码(Entropy Coding)；(3)行程编码(Run Length Coding)；(4)位平面编码(Bit Plane Coding)。其中我们要介绍的是熵编码中的哈夫曼(Huffman)编码和行程编码(以读取.PCX 文件为例)。

所谓预测编码是指，去除相邻象素之间的相关性和冗余性，只对新的信息进行编码。举个简单的例子，因为象素的灰度是连续的，所以在一片区域中，相邻象素之间灰度值的差别可能很小。如果我们只记录第一个象素的灰度，其它象素的灰度都用它与前一个象素灰度之差来表示，就能起到压缩的目的。如 248，2，1，0，1，3，实际上这 6 个象素的灰度是 248，250，251，251，252，255。表示 250 需要 8 个比特，而表示 2 只需要两个比特，这样就实现了压缩。

常用的预测编码有 Δ 调制(Delta Modulation，简称 DM)；微分预测编码(Differential Pulse Code Modulation，DPCM)，具体的细节在此就不详述了。

所谓变换编码是指，将给定的图象变换到另一个数据域(如频域)上，使得大量的信息能用较少的数据来表示，从而达到压缩的目的。变换编码有很多，如(1)离散傅立叶变换(Discrete Fourier Transform，简称 DFT)；(2)离散余弦变换(Discrete Cosine Transform，简称 DCT)；(3)离散哈达玛变换(Discrete Hadamard Transform，简称 DHT)。

其它的编码方法也有很多，如混合编码(Hybird Coding)、矢量量化(Vector Quantize，VQ) 、LZW 算法。在这里，我们只介绍 LZW 算法的大体思想。

值得注意的是，近些年来出现了很多新的压缩编码方法，如使用人工神经元网络(Artificial Neural Network，简称 ANN)的压缩编码算法、分形(Fractl)、小波(Wavelet) 、基于对象(Object Based)的压缩编码算法、基于模型(Model –Based)的压缩编码算法(应用在 MPEG4 及未来的视频压缩编码标准中)。这些都超出了本书的范围。

本章的最后，我们将以 JPEG 压缩编码标准为例，看看上面的几种编码方法在实际的压缩编码中是怎样应用的。

9.1 哈夫曼编码

哈夫曼(Huffman)编码是一种常用的压缩编码方法，是 Huffman 于 1952 年为压缩文本文件建立的。它的基本原理是频繁使用的数据用较短的代码代替，较少使用的数据用较长的代码代替，每个数据的代码各不相同。这些代码都是二进制码，且码的长度是可变的。举个例子：假设一个文件中出现了 8 种符号 S0,S1,S2,S3,S4,S5,S6,S7，那么每种符号要编码，至少需要 3 比特。假设编码成 000,001,010,011,100,101,110,111(称做码字)。那么符号序列 S0S1S7S0S1S6S2S2S3S4S5S0S0S1 编码后变成 000001111000001110010010011100101000000001，共用了 42 比特。我们发现 S0，S1，S2 这三个符号出现的频率比较大，其它符号出现的频率比较小，如果我们采用一种编码方案使得 S0，S1，S2 的码字短，其它符号的码字长，这样就能够减少占用的比特数。例如，我们采用这样的编码方案：S0 到 S7 的码字分别 01,11,101,0000,0001,0010,0011,100，那么上述符号序列变成 011110001110011101101000000010010010111，共用了 39 比特，尽管有些码字

如 S3，S4，S5，S6 变长了(由 3 位变成 4 位)，但使用频繁的几个码字如 S0，S1 变短了，所以实现了压缩。

上述的编码是如何得到的呢？随意乱写是不行的。编码必须保证不能出现一个码字和另一个的前几位相同的情况，比如说，如果 S0 的码字为 01，S2 的码字为 011，那么当序列中出现 011 时，你不知道是 S0 的码字后面跟了个 1，还是完整的一个 S2 的码字。我们给出的编码能够保证这一点。

下面给出具体的 Huffman 编码算法。

(1) 首先统计出每个符号出现的频率，上例 S0 到 S7 的出现频率分别为 4/14，3/14，2/14，1/14，1/14，1/14，1/14，1/14。

(2) 从左到右把上述频率按从小到大的顺序排列。

(3) 每一次选出最小的两个值，作为二叉树的两个叶子节点，将和作为它们的根节点，这两个叶子节点不再参与比较，新的根节点参与比较。

(4) 重复(3)，直到最后得到和为 1 的根节点。

(5) 将形成的二叉树的左节点标 0，右节点标 1。把从最上面的根节点到最下面的叶子节点途中遇到的 0,1 序列串起来，就得到了各个符号的编码。

上面的例子用 Huffman 编码的过程如图 9.1 所示，其中圆圈中的数字是新节点产生的顺序。可见，我们上面给出的编码就是这么得到的。

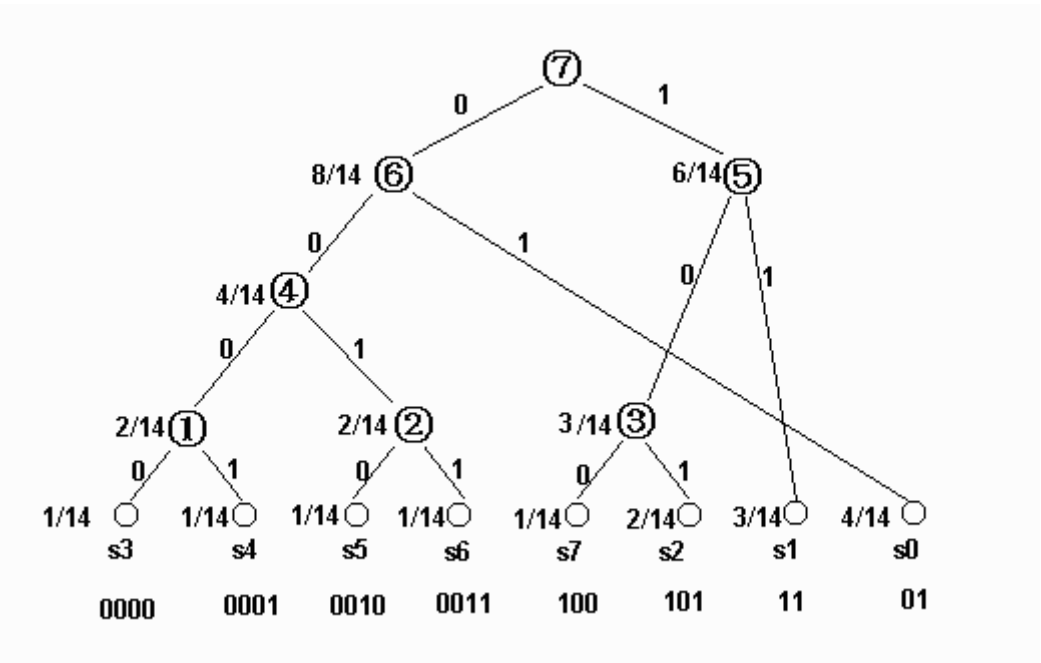

图 9.1　Huffman 编码的示意图

产生 Huffman 编码需要对原始数据扫描两遍。第一遍扫描要精确地统计出原始数据中，每个值出现的频率，第二遍是建立 Huffman 树并进行编码。由于需要建立二叉树并遍历二叉树生成编码，因此数据压缩和还原速度都较慢，但简单有效，因而得到广泛的应用。

源程序就不给出了，有兴趣的读者可以自己实现。

9.2 行程编码

行程编码(Run Length Coding)的原理也很简单：将一行中颜色值相同的相邻象素用一个计数值和该颜色值来代替。例如 aaabcccccccddeee 可以表示为 3a1b6c2d3e。如果一幅图象是由很多块颜色相同的大面积区域组成，那么采用行程编码的压缩效率是惊人的。然而，该算法也导致了一个致命弱点，如果图象中每两个相邻点的颜色都不同，用这种算法不但不能压缩，反而数据量增加一倍。所以现在单纯采用行程编码的压缩算法用得并不多，PCX 文件算是其中的一种。

PCX 文件最早是 PC Paintbrush 软件所采用的一种文件格式，由于压缩比不高，现在用的并不是很多了。它也是由头信息、调色板、实际的图象数据三个部分组成。其中头信息的结构为：

```
typedef struct{
        char manufacturer;
        char version;
        char encoding;
        char bits_per_pixel;
        WORD xmin,ymin;
        WORD xmax,ymax;
        WORD hres;
        WORD vres;
        char palette[48];
        char reserved;
```

```
    char colour_planes;
    WORD bytes_per_line;
    WORD palette_type;
    char filler[58];
} PCXHEAD;
```

其中值得注意的是以下几个数据：manufacturer 为 PCX 文件的标识，必须为 0x0a；xmin 为最小的 x 坐标，xmax 最大的 x 坐标，所以图象的宽度为 xmax-xmin+1，同样图象的高度为 ymax-yin+1；bytes_per_line 为每个编码行所占的字节数，下面将详细介绍。

PCX 的调色板在文件的最后。以 256 色 PCX 文件为例，倒数第 769 个字节为颜色数的标识，256 时该字节必须为 12，剩下的 768(256×3)为调色板的 RGB 值。

为了叙述方便，我们针对 256 色 PCX 文件，介绍一下它的解码过程。编码是解码的逆过程，有兴趣的读者可以试着自己来完成。

解码是以行为单位的，该行所占的字节数由 bytes_per_line 给定。为此，我们开一个大小为 bytes_per_line 的解码缓冲区。一开始，将缓冲区的所有内容清零。从文件中读出一个字节 C，若 C>0xc0，说明是行程(Run Length)信息，即 C 的低 6 位表示后面连续的字节个数(所以最多 63 个连续颜色相同的象素，若还有颜色相同的象素，将在下一个行程处理)，文件的下一个字节就是实际的图象数据(即该颜色在调色板中的索引值)。若 C<0xc0，则表示 C 是实际的图象数据。如此反复，直到这 bytes_per_line 个字节处理完，这一行的解码完成。PCX 就是有若干个这样的解码行组成。

下面是实现 256 色 PCX 文件解码的源程序，其中第二个函数对一行进行解码，应该把阅读的重点放在这个函数上。要注意的是，执行时文件 C:\\test.pcx 必须存在，而且是一个 256 色 PCX 文件。

```
unsigned int     PcxBytesPerLine;

BOOL LoadPcxFile (HWND hWnd,char *PcxFileName)

{
    FILE                    *PCXfp;
    PCXHEAD                 header;
```

```
    LOGPALETTE                  *pPal;
    HPALETTE                    hPrevPalette;
    HDC                         hDc;
    HLOCAL                      hPal;
    DWORD                       ImgSize;
    DWORD                   OffBits,BufSize;
    LPBITMAPINFOHEADER      lpImgData;
    DWORD                   i;
    LONG                        x,y;
    int                     PcxTag;
    unsigned char               LineBuffer[6400];
    LPSTR                       lpPtr;
    HFILE                   hfbmp;
if((PCXfp=fopen(PcxFileName,"rb"))==NULL){ //文件没有找到
MessageBox(hWnd,"File c:\\test.pcx not found!","Error Message",
MB_OK|MB_ICONEXCLAMATION);
return FALSE;
    }
    //读出头信息
    fread((char*)&header,1,sizeof(PCXHEAD),PCXfp);
    if(header.manufacturer!=0x0a){ //不是一个合法的 PCX 文件
MessageBox(hWnd,"Not a valid Pcx file!","Error Message",
```

```
MB_OK|MB_ICONEXCLAMATION);
fclose(PCXfp);
            return FALSE;
        }
        //将文件指针指向调色板开始处
        fseek(PCXfp,-769L,SEEK_END);
        //获取颜色数信息
        PcxTag=fgetc(PCXfp)&0xff;
        if(PcxTag!=12){ //非 256 色，返回
            MessageBox(hWnd,"Not a 256 colors Pcx file!","Error Message",
MB_OK|MB_ICONEXCLAMATION);
fclose(PCXfp);
             return FALSE;
        }
//创建新的 BITMAPFILEHEADER 和 BITMAPINFOHEADER
        memset((char *)&bf,0,sizeof(BITMAPFILEHEADER));
        memset((char *)&bi,0,sizeof(BITMAPINFOHEADER));
        //填写 BITMAPINFOHEADER 头信息
        bi.biSize=sizeof(BITMAPINFOHEADER);
        //得到图象的宽和高
        bi.biWidth=header.xmax-header.xmin+1;
        bi.biHeight=header.ymax-header.ymin+1;
```

```
        bi.biPlanes=1;
        bi.biBitCount=8;
        bi.biCompression=BI_RGB;
        ImgWidth=bi.biWidth;
        ImgHeight=bi.biHeight;
        NumColors=256;
        LineBytes=(DWORD)WIDTHBYTES(bi.biWidth*bi.biBitCount);
        ImgSize=(DWORD)LineBytes*bi.biHeight;
        //填写 BITMAPFILEHEADER 头信息
        bf.bfType=0x4d42;
        bf.bfSize=sizeof(BITMAPFILEHEADER)+sizeof(BITMAPINFOHEADER)+
NumColors*sizeof(RGBQUAD)+ImgSize;
        bf.bfOffBits=(DWORD)(NumColors*sizeof(RGBQUAD)+
sizeof(BITMAPFILEHEADER)+sizeof(BITMAPINFOHEADER));
        //为新图分配缓冲区
        if((hImgData=GlobalAlloc(GHND,(DWORD)
(sizeof(BITMAPINFOHEADER)+
NumColors*sizeof(RGBQUAD)+ImgSize)))==NULL)
        {
MessageBox(hWnd,"Error alloc memory!","ErrorMessage",
MB_OK|MB_ICONEXCLAMATION);
            fclose(PCXfp);
```

```
            return FALSE;
        }
        lpImgData=(LPBITMAPINFOHEADER)GlobalLock(hImgData);
        //拷贝头信息
        memcpy(lpImgData,(char *)&bi,sizeof(BITMAPINFOHEADER));
        lpPtr=(char *)lpImgData+sizeof(BITMAPINFOHEADER);
        //为 256 色调色板分配内存
       hPal=LocalAlloc(LHND,sizeof(LOGPALETTE)+
NumColors* sizeof(PALETTEENTRY));
pPal =(LOGPALETTE *)LocalLock(hPal);
       pPal->palNumEntries =256;
        pPal->palVersion = 0x300;
        for (i = 0; i < 256; i++) {
            //读取调色板中的 RGB 值
           pPal->palPalEntry[i].peRed=(BYTE)fgetc(PCXfp);
            pPal->palPalEntry[i].peGreen=(BYTE)fgetc(PCXfp);
            pPal->palPalEntry[i].peBlue=(BYTE)fgetc(PCXfp);
            pPal->palPalEntry[i].peFlags=(BYTE)0;
            *(lpPtr++)=(unsigned char)pPal->palPalEntry[i].peBlue;
            *(lpPtr++)=(unsigned char)pPal->palPalEntry[i].peGreen;
            *(lpPtr++)=(unsigned char)pPal->palPalEntry[i].peRed;
            *(lpPtr++)=0;
```

```
        }
        //产生新的逻辑调色板
        hPalette=CreatePalette(pPal);
        LocalUnlock(hPal);
        LocalFree(hPal);
        hDc=GetDC(hWnd);
        if(hPalette){
hPrevPalette=SelectPalette(hDc,hPalette,FALSE);
            RealizePalette(hDc);
        }
        //解码行所占的字节数
        PcxBytesPerLine=(unsigned int)header.bytes_per_line;
        //将文件指针指向图象数据的开始处
        fseek(PCXfp,(LONG)sizeof(PCXHEAD),SEEK_SET);
        //缓冲区大小
OffBits=bf.bfOffBits-sizeof(BITMAPFILEHEADER);
//BufSize 为缓冲区大小
        BufSize=OffBits+bi.biHeight*LineBytes;
        for(y=0;y<bi.biHeight;y++){
            //指向新图中相应的位置
            lpPtr=(char *)lpImgData+BufSize-LineBytes-y*LineBytes;
            //解码该行，放在数组 LineBuffer 中
```

```
            ReadPcxLine(LineBuffer,PCXfp);
            for(x=0;x<bi.biWidth;x++)
                *(lpPtr++)=LineBuffer[x]; //将该行存储到位图数据中
        }
        //创建新的位图
        hBitmap=CreateDIBitmap(hDc,(LPBITMAPINFOHEADER)lpImgData,
(LONG)CBM_INIT,
(LPSTR)lpImgData+
sizeof(BITMAPINFOHEADER)+
NumColors*sizeof(RGBQUAD),
(LPBITMAPINFO)lpImgData,
DIB_RGB_COLORS);
        if(hPalette && hPrevPalette){
            SelectPalette(hDc,hPrevPalette,FALSE);
            RealizePalette(hDc);
        }
        hfbmp=_lcreat("c:\\pcx2bmp.bmp",0);
        _lwrite(hfbmp,(LPSTR)&bf,sizeof(BITMAPFILEHEADER));
        _lwrite(hfbmp,(LPSTR)lpImgData,BufSize);
        _lclose(hfbmp);
        fclose(PCXfp);
        //释放内存和资源
```

```
    ReleaseDC(hWnd,hDc);
    GlobalUnlock(hImgData);
    return TRUE;
}
//对每一行进行解码，结果存储到指针 p 指向的内存中
void ReadPcxLine(unsigned char *p,FILE *fp)
{
    unsigned int    n=0,i;
    char            c;
    memset(p,0,PcxBytesPerLine);
    do{
        //读出一个字节
        c=fgetc(fp)&0xff;
if((c&0xc0)==0xc0){ //是个形成字节
//i 为 c 的低六位
i=c&0x3f;
            //下一个字节为实际的图象数据
            c=fgetc(fp);
            while(i--) p[n++]=c; //填充连续的 i 个字节到 p 中
        }
        else p[n++]=c; //否则是实际的图象数据，直接填入到 p 中
    }while (n<PcxBytesPerLine); //共读取 PcxBytesPerLine 个字节
```

}

对一幅的 PCX 文件格式的图象解码后，结果如图 9.2 所示。显示的是我最喜欢的法国影星阿佳妮·伊莎贝拉。

图 9.2　一幅 PCX 文件格式的图象

9.3 LZW 算法的大体思想

LZW 是一种比较复杂的压缩算法，其压缩效率也比较高。我们在这里只介绍一下它的基本原理：LZW 把每一个第一次出现的字符串用一个数值来编码，在还原程序中再将这个数值还成原来的字符串。例如：用数值 0x100 代替字符串“abccddeee”，每当出现该字符串时，都用 0x100 代替，这样就起到了压缩的作用。至于 0x100 与字符串的对应关系则是在压缩过程中动态生成的，而且这种对应关系隐含在压缩数据中，随着解压缩的进行这张编码表会从压缩数据中逐步得到恢复，后面的压缩数据再根据前面数据产生的对应关系产生更多的对应关系，直到压缩文件结束为止。LZW 是无损的。GIF 文件采用了这种压缩算法。

要注意的是，LZW 算法由 Unisys 公司在美国申请了专利，要使用它首先要获得该公司的认可。

9.4 JPEG 压缩编码标准

JPEG 是联合图象专家组(Joint Picture Expert Group)的英文缩写，是国际标准化组织(ISO)和 CCITT 联合制定的静态图象的压缩编码标准。和相同图象质量的其它常用文件格式(如 GIF，

TIFF，PCX)相比，JPEG 是目前静态图象中压缩比最高的。我们给出具体的数据来对比一下。例图采用 Windows95 目录下的 Clouds.bmp，原图大小为 640*480，256 色。用工具 SEA(version1.3)将其分别转成 24 位色 BMP、24 位色 JPEG、GIF(只能转成 256 色)压缩格式、24 位色 TIFF 压缩格式、24 位色 TGA 压缩格式。得到的文件大小(以字节为单位)分别为：921,654，17,707，177,152，923,044，768,136。可见 JPEG 比其它几种压缩比要高得多，而图象质量都差不多(JPEG 处理的颜色只有真彩和灰度图)。

正是由于 JPEG 的高压缩比，使得它广泛地应用于多媒体和网络程序中，例如 HTML 语法中选用的图象格式之一就是 JPEG(另一种是 GIF)。这是显然的，因为网络的带宽非常宝贵，选用一种高压缩比的文件格式是十分必要的。

JPEG 有几种模式，其中最常用的是基于 DCT 变换的顺序型模式，又称为基线系统(Baseline)，以下将针对这种格式进行讨论。

1. JPEG 的压缩原理

JPEG 的压缩原理其实上面介绍的那些原理的综合，博采众家之长，这也正是 JPEG 有高压缩比的原因。其编码器的流程为：

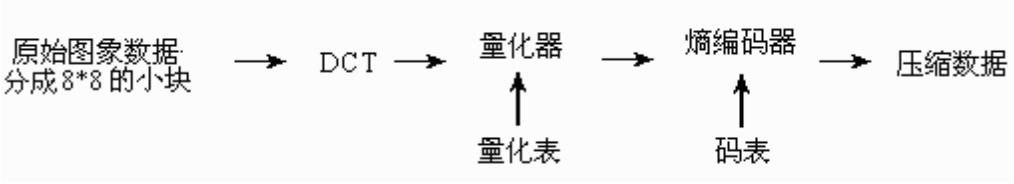

图 9.3 JPEG 编码器流程

解码器基本上为上述过程的逆过程：

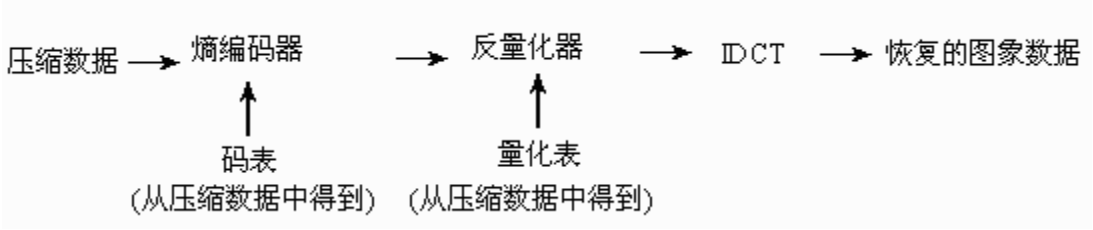

图 9.4 解码器流程

8×8 的图象经过 DCT 变换后，其低频分量都集中在左上角，高频分量分布在右下角(DCT 变换实际上是空间域的低通滤波器)。由于该低频分量包含了图象的主要信息(如亮度)，而高频与之相比，就不那么重要了，所以我们可以忽略高频分量，从而达到压缩的目的。如何将高频分量去掉，这就要用到量化，它是产生信息损失的根源。这里的量化操作，就是将某一个值除以量化表中对应的值。由于量化表左上角的值较小，右上角的值较大，这样就起到了

保持低频分量，抑制高频分量的目的。JPEG 使用的颜色是 YUV 格式。我们提到过，Y 分量代表了亮度信息，UV 分量代表了色差信息。相比而言，Y 分量更重要一些。我们可以对 Y 采用细量化，对 UV 采用粗量化，可进一步提高压缩比。所以上面所说的量化表通常有两张，一张是针对 Y 的；一张是针对 UV 的。

上面讲了，经过 DCT 变换后，低频分量集中在左上角，其中 F(0，0)(即第一行第一列元素)代表了直流(DC)系数，即 8×8 子块的平均值，要对它单独编码。由于两个相邻的 8×8 子块的 DC 系数相差很小，所以对它们采用差分编码 DPCM，可以提高压缩比，也就是说对相邻的子块 DC 系数的差值进行编码。8×8 的其它 63 个元素是交流(AC)系数，采用行程编码。这里出现一个问题：这 63 个系数应该按照怎么样的顺序排列？为了保证低频分量先出现，高频分量后出现，以增加行程中连续“0”的个数，这 63 个元素采用了“之”字型(Zig-Zag)的排列方法，如图 9.5 所示。

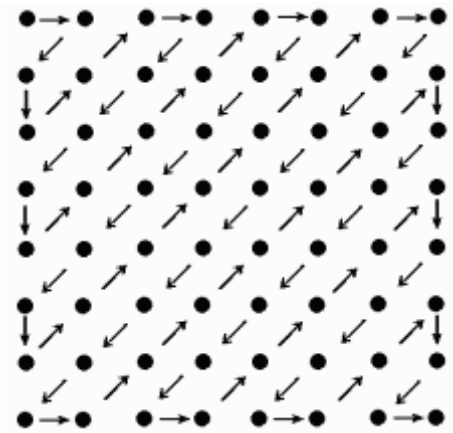

图 9.5　Zig-Zag

这 63 个 AC 系数行程编码的码字用两个字节表示，如图 9.6 所示。

图 9.6　行程编码

上面，我们得到了 DC 码字和 AC 行程码字。为了进一步提高压缩比，需要对其再进行熵编码，这里选用 Huffman 编码，分成两步：

(1)熵编码的中间格式表示

对于 AC 系数，有两个符号。符号 1 为行程和尺寸，即上面的(RunLength，Size)。(0，0)和(15，0)是两个比较特殊的情况。(0，0)表示块结束标志(EOB)，(15，0)表示 ZRL，当行程长度超过 15 时，用增加 ZRL 的个数来解决，所以最多有三个 ZRL(3×16+15=63)。符号 2 为幅度值(Amplitude)。

对于 DC 系数，也有两个符号。符号 1 为尺寸(Size)；符号 2 为幅度值(Amplitude)。

(2)熵编码

对于 AC 系数，符号 1 和符号 2 分别进行编码。零行程长度超过 15 个时，有一个符号(15，0)，块结束时只有一个符号(0，0)。

对符号 1 进行 Hufffman 编码(亮度，色差的 Huffman 码表不同)。对符号 2 进行变长整数 VLI 编码。举例来说：Size=6 时，Amplitude 的范围是-63~-32，以及 32~63，对绝对值相同，符号相反的码字之间为反码关系。所以 AC 系数为 32 的码字为 100000，33 的码字为 100001，-32 的码字为 011111，-33 的码字为 011110。符号 2 的码字紧接于符号 1 的码字之后。

对于 DC 系数，Y 和 UV 的 Huffman 码表也不同。

掉了这么半天的书包，你可能已经晕了，呵呵。举个例子来说明上述过程就容易明白了。

下面为 8×8 的亮度(Y)图象子块经过量化后的系数。

15 0 -1 0 0 0 0 0

-2 -1 0 0 0 0 0 0

-1 -1 0 0 0 0 0 0

0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0

可见量化后只有左上角的几个点(低频分量)不为零，这样采用行程编码就很有效。

第一步，熵编码的中间格式表示：先看 DC 系数。假设前一个 8×8 子块 DC 系数的量化值为 12，则本块 DC 系数与它的差为 3，根据下表

Size	Amplitude
0	0
1	–1,1
2	–3,-2,2,3
3	–7~-4，4~7
4	–15~-8，8~15
5	–31~-16，16~31
6	–63~-32，32~63
7	–127~-64，64~127
8	–255~-128，128~255
9	–511~-256，256~511
10	–1023~512，512~1023
11	–2047~-1024，1024~2047

查表得 Size=2，Amplitude=3，所以 DC 中间格式为(2)(3)。

下面对 AC 系数编码。经过 Zig-Zag 扫描后，遇到的第一个非零系数为-2，其中遇到零的个数为 1(即 RunLength)，根据下面这张 AC 系数表：

Size	Amplitude
1	–1,1
2	–3,-2,2,3
3	–7~-4，4~7
4	–15~-8，8~15

5 –31~-16，16~31

6 –63~-32，32~63

7 –127~-64，64~127

8 –255~-128，128~255

9 –511~-256，256~511

10 –1023~512，512~1023

查表得 Size=2。所以 RunLength=1,Size=2,Amplitude=3，所以 AC 中间格式为(1,2)(-2)。

其余的点类似，可以求得这个 8×8 子块熵编码的中间格式为

(DC)(2)(3),(1,2)(-2),(0,1)(-1),(0,1)(-1),(0,1)(-1),(2,1)(-1),(EOB)(0,0)

第二步，熵编码：

对于(2)(3)：2 查 DC 亮度 Huffman 表得到 11，3 经过 VLI 编码为 011；

对于(1,2)(-2)：(1,2)查 AC 亮度 Huffman 表得到 11011，-2 是 2 的反码，为 01；

对于(0,1)(-1)：(0,1)查 AC 亮度 Huffman 表得到 00，-1 是 1 的反码，为 0；

……

最后，这一 8×8 子块亮度信息压缩后的数据流为 11011, 1101101, 000, 000, 000, 111000,1010。总共 31 比特，其压缩比是 64×8/31=16.5，大约每个象素用半个比特。

可以想见，压缩比和图象质量是呈反比的，以下是压缩效率与图象质量之间的大致关系，可以根据你的需要，选择合适的压缩比。

表 9.1　压缩比与图象质量的关系

压缩效率(单位：bits/pixel)	图象质量
0.25~0.50	中~好，可满足某些应用
0.50~0.75	好~很好，满足多数应用
0.75~1.5	极好，满足大多数应用
1.5~2.0	与原始图象几乎一样

以上我们介绍了 JPEG 压缩的原理，其中 DC 系数使用了预测编码 DPCM，AC 系数使用了变换编码 DCT，二者都使用了熵编码 Huffman，可见几乎所有传统的压缩方法在这里都用到了。这几种方法的结合正是产生 JPEG 高压缩比的原因。顺便说一下，该标准是 JPEG 小组从很多种不同中方案中比较测试得到的，并非空穴来风。

上面介绍了 JPEG 压缩的基本原理，下面介绍一下 JPEG 的文件格式。

2. JPEG 的文件格式

JPEG 文件大体上可以分成以下两个部分：标记码(Tag)加压缩数据。先介绍标记码部分。

标记码部分给出了 JPEG 图象的所有信息(有点类似于 BMP 中的头信息，但要复杂的多)，如图象的宽、高、Huffman 表、量化表等等。标记码有很多，但绝大多数的 JPEG 文件只包含几种。标记码的结构为：

SOI

DQT

DRI

SOF0

DHT

SOS

…

EOI

标记码由两个字节组成，高字节为 0XFF，每个标记码之前可以填上个数不限的填充字节 0XFF。

下面介绍一些常用的标记码的结构及其含义。

(1)SOI(Start of Image)

标记结构	字节数
0XFF	1

0XD8 1

可作为 JPEG 格式的判据(JFIF 还需要 APP0 的配合)

(2)APP0(Application)

标记结构 字节数 意义

0XFF 1

0XE0 1

Lp 2 APP0 标记码长度，不包括前两个字节 0XFF，0XE0

Identifier 5 JFIF 识别码 0X4A，0X46，0X49，0X46，0X00

Version 2 JFIF 版本号 可为 0X0101 或者 0X0102

Units 1 单位，等于零时表示未指定，为 1 表示英寸，为 2 表示

厘米

Xdensity 2 水平分辨率

Ydensity 2 垂直分辨率

Xthumbnail 1 水平点数

Ythumbnail 1 垂直点数

RGB0 3 RGB 的值

RGB1 3 RGB 的值

…

RGBn 3 RGB 的值，n=Xthumbnail*Ythumbnail

APP0 是 JPEG 保留给 Application 所使用的标记码，而 JFIF 将文件的相关信息定义在此标记中。

(3)DQT(Define Quantization Table)

标记结构	字节数	意义
0XFF	1	
0XDB	1	
Lq	2	DQT 标记码长度，不包括前两个字节 0XFF，0XDB
(Pq,Tq)	1	高四位 Pq 为量化表的数据精确度，Pq=0 时，Q0~Qn 的值为 8 位，Pq=1 时，Qt 的值为 16 位，Tq 表示量化表的编号，为 0~3。在基本系统中，Pq=0，Tq=0~1，也就是说最多有两个量化表。
Q0	1 或 2	量化表的值，Pq=0 时；为一个字节，Pq=1 时，为两个字节
Q1	1 或 2	量化表的值，Pq=0 时；为一个字节，Pq=1 时，为两个字节
…		
Qn	1 或 2	量化表的值，Pq=0 时，为一个字节；Pq=1 时，为两个字节。n 的值为 0~63，表示量化表中 64 个值(之字形排列)

(4)DRI(Define Restart Interval)

此标记需要用到最小编码单元(MCU，Minimum Coding Unit)的概念。前面提到，Y 分量数据重要，UV 分量的数据相对不重要，所以可以只取 UV 的一部分，以增加压缩比。目前支持 JPEG 格式的软件通常提供两种取样方式 YUV411 和 YUV422，其含义是 YUV 三个分量的数据取样比例。举例来说，如果 Y 取四个数据单元，即水平取样因子 Hy 乘以垂直取样因子 Vy 的值为 4，而 U 和 V 各取一个数据单元，即 Hu×Vu=1,Hv×Vv=1。那么这种部分取样就称为 YUV411。如图 9.7 所示：

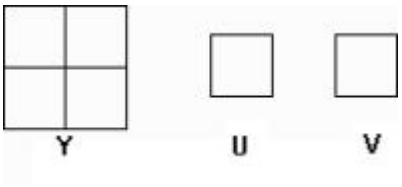

图 9.7　YUV411 的示意图

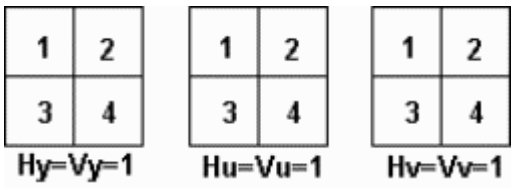

图 9.8　YUV111 的排列顺序

易知 YUV411 有 50%的压缩比(原来有 12 个数据单元，现在有 6 个数据单元)，YUV422 有 33%的压缩比(原来有 12 个数据单元，现在有 8 个数据单元)。

那么你可能会想，YUV911，YUV1611 压缩比不是更高嘛？但是要考虑到图象质量的因素。所以 JPEG 标准规定了最小编码单元 MCU，要求 Hy×Vy+Hu×Vu+Hv×Vv≤10。

MCU 中块的排列方式与 H，V 的值有密切关系，如图 9.8、图 9.9、图 9.10 所示。

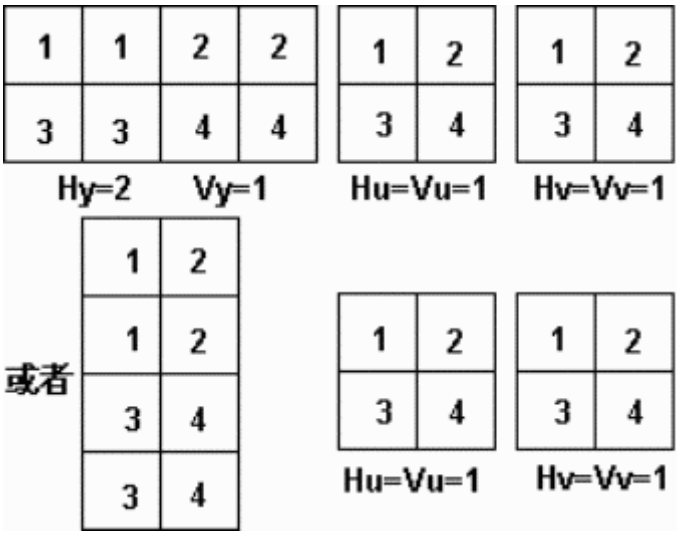

图 9.9　YUV211 的排列顺序

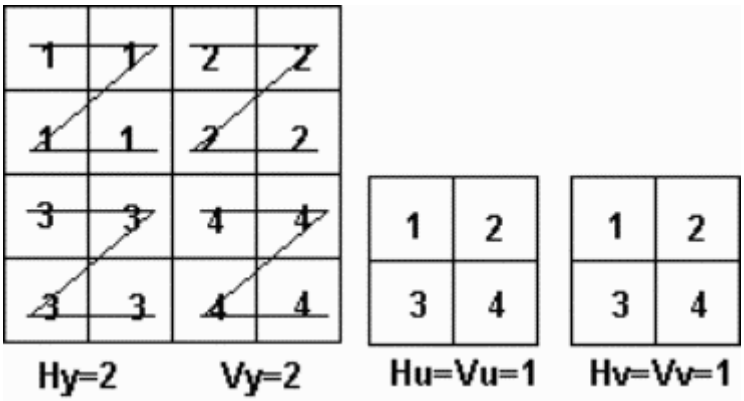

图 9.10　YUV411 的排列顺序

标记结构　　字节数　意义

0XFF　　1

0XDD　　1

Lr　　2　　DRI 标记码长度，不包括前两个字节 0XFF，0XDD

Ri　　2　　重入间隔的 MCU 个数，Ri 必须是一 MCU 行中 MCU 个数的整数，最后一个零头不一定刚好是 Ri 个 MCU。

每个重入间隔各自独立编码。

(5)SOF(Start of Frame) 在基本系统中，只处理 SOF0

标记结构　　字节数　意义

0XFF　　1

0XC0　　1

Lf　　2　　SOF 标记码长度，不包括前两个字节 0XFF，0XC0

P　　1　　基本系统中，为 0X08

Y　　2　　图象高度

X　　2　　图象宽度

Nf　　1　　Frame 中的成分个数，一般为 1 或 3，1 代表灰度图，3 代表真彩图

C1　　1　　成分编号 1

(H1,V1)　1　　第一个水平和垂直采样因子

Tq1　　1　　该量化表编号

C2　　1　　成分编号 2

(H2,V2)　1　　第二个水平和垂直采样因子

Tq2　　1　　该量化表编号

…

Cn　　1　　成分编号 n

(Hn,Vn)　1　　第 n 个水平和垂直采样因子

Tqn　　1　　该量化表编号

(6)DHT(Define Huffman Table)

标记结构　　字节数　意义

0XFF　　1

0XC4　　1

Lh　　2　　DHT 标记码长度，不包括前两个字节 0XFF，0XC4

(Tc,Th)　　1

L1　　1

L2　　1

…

L16　　1

V1　　1

V2　　1

…

Vt　　1

Tc 为高 4 位，Th 为低 4 位。在基本系统中，Tc 为 0 或 1，为 0 时，指 DC 所用的 Huffman 表，为 1 时，指 AC 所用的 Huffman 表。Th 表示 Huffman 表的编号，在基本系统中，其值为 0 或 1。所以，在基本系统中，最多有 4 个 Huffman 表，如下所示：

Tc　Th　Huffman 表编号(2×Tc+Th)

0 0

1 1

0 2

1 1 3

Ln 表示每个 n 比特的 Huffman 码字的个数，n=1~16

Vt 表示每个 Huffman 码字所对应的值，也就是我们前面所讲的符号 1，对 DC 来说该值为(Size)，对 AC 来说该值为(RunLength，Size)。

t=L1+L2+…L16

(7)SOS(Start of Scan)

标记结构	字节数	意义
0XFF	1	
0XDA	1	
Ls	2	DHT 标记码长度，不包括前两个字节 0XFF，0XDA
Ns	1	
Cs1	1	
(Td1,Ta1)	1	
Cs2	1	
(Td2,Ta2)	1	
…		
CsNs	1	
(TdNs,TaNs)	1	
Ss	1	

Se 1

(Ah，Al) 1

Ns 为 Scan 中成分的个数，在基本系统中，Ns=Nf(Frame 中成分个数)。CSNs 为在 Scan 中成分的编号。TdNs 为高 4 位，TaNs 为低 4 位，分别表示 DC 和 AC 编码表的编号。在基本系统中 Ss=0，Se=63，Ah=0，Al=0。

(8)EOI(End of Image) 结束标志

标记结构 字节数 意义

0XFF 1

0XD9 1

3. JPEG 基本系统解码器的实现

笔者曾经实现了一个 Windows 下 JPEG 基本系统的解码器，限于篇幅，这里就不给源程序了，只给出大体上的程序流程图(见图 9.11)。

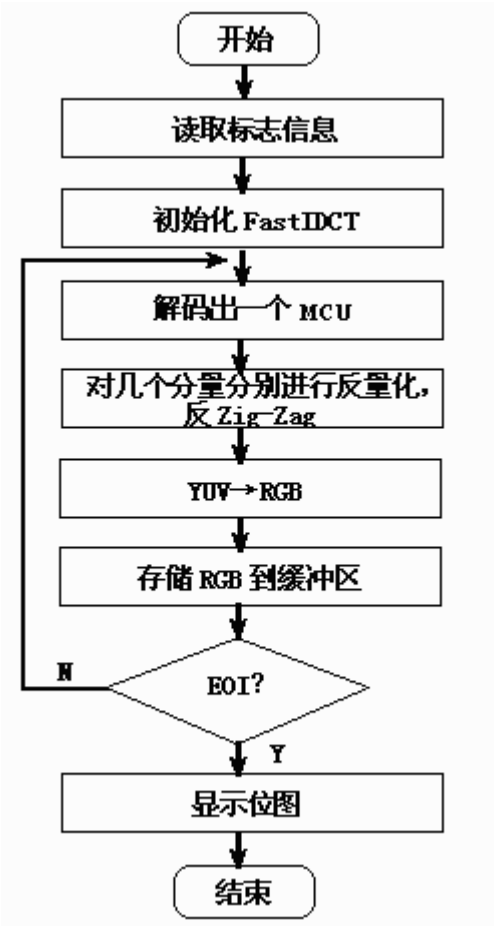

图 9.11　JPEG 解码器的程序流程图

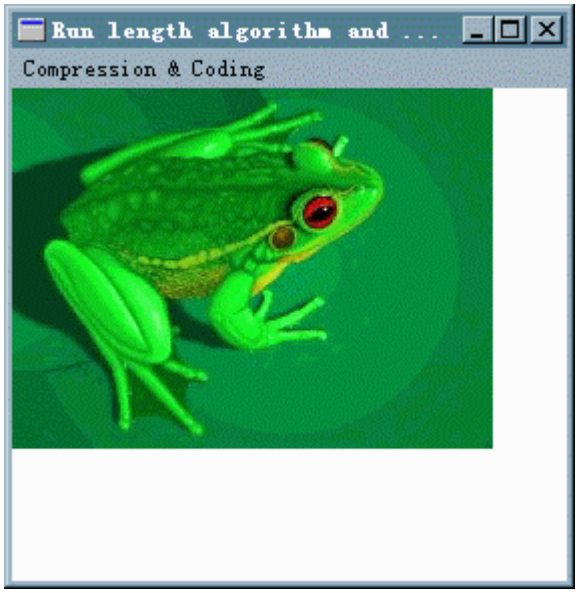

图 9.12　程序运行时的画面

由于没有用到什么优化算法，该解码器的速度并不高，在用 VC 的性能评测工具 Profile 评测该程序时我发现最耗时的地方是反离散余弦变换(IDCT)那里，其实这是显然的，浮点数的指令条数要比整数的多得多，因此采用一种快速的 IDCT 算法能很大的提高性能，我这里采用是目前被认为比较好的一种快速 IDCT 算法，其主要思想是把二维 IDCT 分解成行和列两个一维 IDCT。图 9.12 是程序运行时的画面。