JPEG 学习笔记

陈硕

chenshuo@chenshuo.com

2005 年 11 月

JPEG 静态图像压缩的基本(baseline)算法分 4 个步骤[1]：

1. 按 8 × 8 象素分块
2. 离散余弦变换(DCT)
3. 量化
4. 熵编码

如果图片是彩色的，那么通常在第一步同时做色彩空间变换（RGB → YCbCr）。编码算法的框图见图 1。

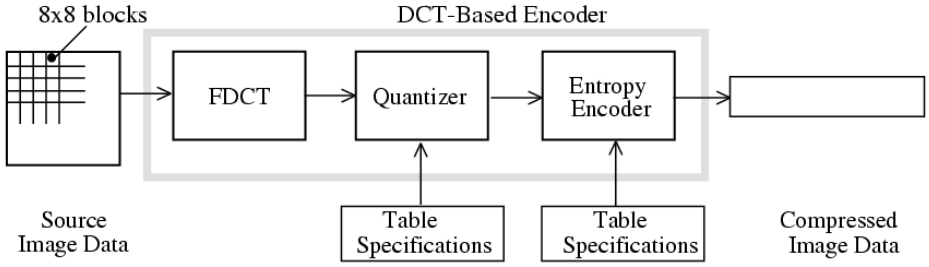

图 1 - JPEG 编码器框图[1]

解码算法是编码算法的逆过程，框图见图 2。

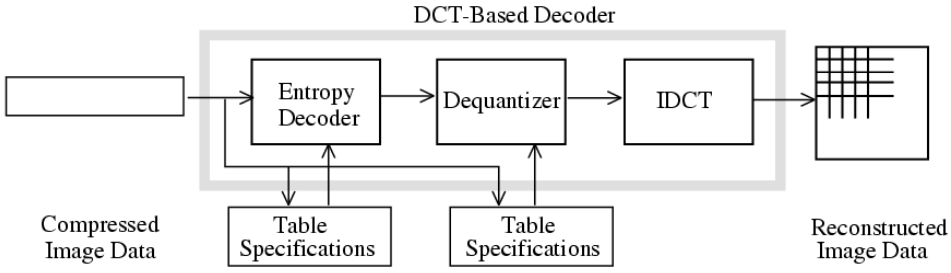

图 2 - JPEG 解码器框图[1]

JPEG 标准[2]中还定义了无损压缩算法，不过基本没人用。除了 baseline 方式，还有 progressive 方式和 hierarchical 方式，本文只谈 baseline 方式。

目 录

1. 算法描述与比较

1.1 分块及预处理

JPEG 可以压缩连续色调的灰度图像或彩色图像。对于灰度图像，每个象素可以是 8-bit 或 12-bit 的灰度值。baseline 模式只要求处理 8-bit 图像（某些医学影像是 12-bit 的）。对于彩色图像，原则上可以将每个色彩分量成分看作一幅灰度图像，再进行压缩。不过由于人眼对于亮度变化比色彩变化更敏感，因此通常把 RGB 彩色信号变换为亮度和色差信号，并对色差信号进行下采样(downsampling)，这样可以提高压缩率。具体来讲，如果输入图像用 RGB 色彩表示，那么要把它变换到 YCbCr 色彩空间。每个象素 (R, G, B) 对应的 (Y, Cb, Cr) 的计算公式[3]是：

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.2990 & 0.5870 & 0.1140 \\ -0.1687 & -0.3313 & 0.5000 \\ 0.5000 & -0.4187 & -0.0813 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix}$$

其中 R, G, B 和 Y, Cb, Cr 都是 8-bit 无符号整数，范围从 0 到 255。色彩变换基本不影响图像的质量，最多由于整数运算的截断会使解码的图像与原图有些微差别。对于 $N \times M$ 的图像，色彩变换的计算复杂度为 $O(MN)$。

JPEG 压缩编码的第一步是将输入图像按从左到右，从上到下的顺序分成 8×8 大小的块。编码器本质上是对一连串 8×8 大小的灰度图像进行编码。对于彩色图像，还涉及到各色彩分量的交织(interleaving)与下采样处理，这留在“实现细节”一节中讨论。

如果原图像的宽度或高度不是 8 的整数倍，需要对原图扩大一圈，让宽度和高度都是 8 的倍数，然后进行压缩。当然，在生成的 JPEG 文件中要记录原始图像的大小，并把解压缩的（大了一圈的）图像剪裁到原大小。图像的填补和分块伴随离散余弦变换一同进行，可不计复杂度。

1.2 变换编码

JPEG 是一种基于变换的图像压缩算法，它采用了离散余弦变换 (DCT)。基于变换的一般编码系统[9, Chap. 8]的框图见图 3。

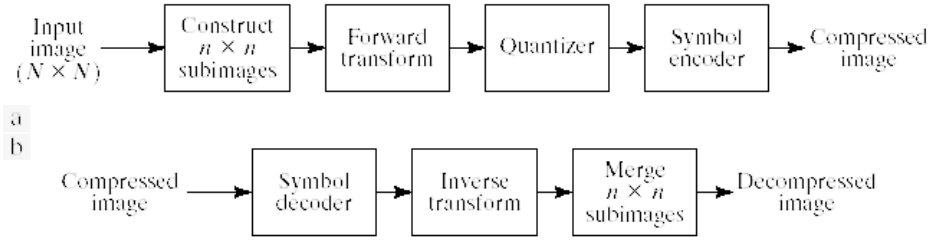

图 3 - 基于变换的编码系统：(a) 编码器；(b) 解码器。

“变换”的基本思想是，找到一组基，让图像在这组基下能量集中（少量系数值较大，其余系数接近 0）。为了达到分离能量的目的，“变换”一般应是正交的。变换必须是可逆的（否则无法解码），因此变换本身不能压缩信息。不过通过随后的“量化”步骤将影响不大的系数略去，即可达到压缩的目的。

适合图像压缩的变换有很多种[9, Chap. 8]，例如Karhunen-Loève 变换 (KLT，即主成分分析)、离散傅立叶变换 (DFT)、离散余弦变换 (DCT)、Walsh-Hadamard 变换 (WHT) 等等。

在数据压缩方面最佳的变换是 Karhunen-Loève 变换，但实际系统中一般不用它。原因有两点：1. KLT 得到的基与输入数据相关，意味着除了需要把系数传给解码器，还需要把基传给解码器。而且每个分块的基往往是不同的！这使得压缩效果大打折扣。2. KLT 需要计算协方差矩阵的特征向量，计算量相当大。

DFT、WHT、DCT 的基是固定的，与输入无关。在与输入无关的各种变换中，非正弦变换（如 WHT）最容易实现，正弦变换（如 DFT 和 DCT）的信息压缩能力更接近最佳的 KLT 方法。DFT 是酉空间的正交变换（系数是复数），DCT 是欧式空间的正交变换（系数是实数），因此 DFT 的系数比 DCT 多一倍。不过实数的 DFT 的系数是共扼对称的，实际所需的存储空间与 DCT 一样。

图像压缩的质量与所用的变换有关，也与分块的大小有关，最常采用的分块尺寸为 8×8 和 16×16。图 4 说明了分块尺寸对变换编码重构误差的影响[9, p. 478]。这里先计算各个分块的变换，截取 75% 的系数，然后对截取后的数组进行反变换，最后计算重构出的图像的误差。

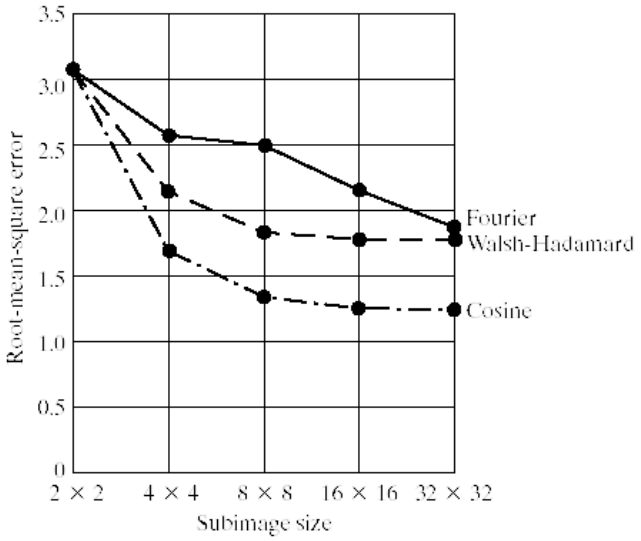

图 4 - 重构误差与分块尺寸的关系

从图 4 可以看出，据重构误差比较，DCT 比 DFT 和 WHT 要好。DCT 优于 DFT 的另外一个原因是[9, p. 477]，由于 DFT 的边缘振铃现象（Gibbs 现象），相邻图像分块之间的边界变得可见。DCT 减少了这种效应，一是它的固有周期比 DFT 长一倍，二是它的边界不间断，见图 5 的比较。

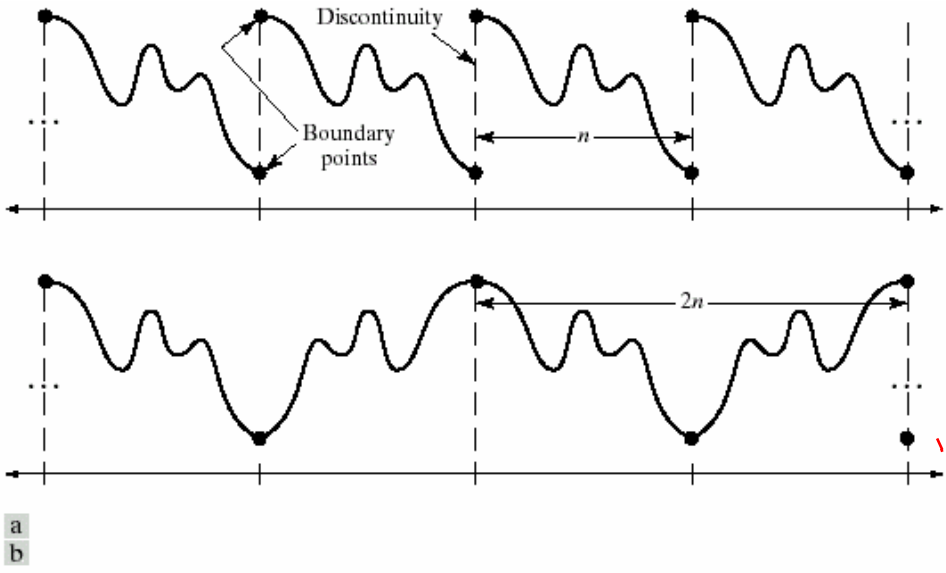

图 5 - 一维 DFT (a) 和 DCT (b) 的固有周期

因此为了兼顾压缩效果和计算复杂度，JPEG 采用 8×8 的2-D DCT 作为其核心，该变换的定义是

$$F(u,v) = \frac{1}{4}C(u)C(v)\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16}$$

其中当 $u, v = 0$ 时 $C(u), C(v) = \frac{1}{\sqrt{2}}$，否则 $C(u), C(v) = 1$。在计算 DCT 之前，要把输入图像的象素值减去 128，让它的取值范围从 $[0, 255]$ 移到 $[-128, 127]$。

每个 8×8 的分块经过 DCT 运算得到 64 个系数。其中 $F(0,0)$ 称为直流系数，其余 63 个系数称为交流系数。在典型的连续色调图像中，相邻象素之间的差别往往不大。这意味着空间频率的幅值集中在低频部分。对典型的 8×8 图像分块而言，大多数空间频率幅值是 0 或接近 0，在编码时可忽略。DCT 本身并不降低图像的质量，最多由于整数运算的舍入误差导致重构的图像与原始图像略有差别。

如果直接采用前面的公式计算，每个 DCT 系数 $F(u,v)$ 需要 64 次乘法，计算全部 64 个 DCT 系数需要 $64 \times 64 = 4096$ 次乘法。计算 DCT 是整个 JPEG 压缩过程中最耗时的一步，很多人研究快速算法[5, Item 25]，也有人往 CPU 增加新指令，使

之便于 DCT 运算。基本思路是把 2-D DCT 分解为 1-D DCT，再利用余弦函数的对称性来减少计算量。每个分块的计算复杂度为 $O(1)$。对于 $N \times M$ 的图像，共有 $\lceil \frac{N}{8} \rceil \times \lceil \frac{M}{8} \rceil$ 个分块，因此 DCT 这一步的计算复杂度为 $O(MN)$。

1.3 量化

量化(quantization)是 JPEG 压缩过程中惟一会大幅损失信息的步骤。64 个 DCT 系数会用 8×8 的量化表进行均匀量化，量化表中的每个元素是 1 到 255 之间的整数，表示对应的 DCT 系数的量化步长。量化的作用在于降低 DCT 系数的精度，从而达到更好的压缩率。量化是多对一映射，因此是有损的，它是基于变换的编码器中导致信息损失的主要步骤，也是用户惟一能参与控制压缩质量的步骤。

量化的过程是将每个 DCT 系数除以对应的量化步长，并四舍五入为整数：

$$F^Q(u,v) = \text{round}\Big(\frac{F(u,v)}{Q(u,v)}\Big)$$

反量化(dequantization)是把量化后的系数乘以对应的量化步长：

$$\hat{F}(u,v) = F^Q(u,v) \cdot Q(u,v)$$

随后 $\hat{F}(u,v)$ 将作为 IDCT 的输入。

量化表理论上应该根据输入图像确定，目标是在基本不影响图像的视觉效果的前提下，尽量提高压缩率。量化步长越大，压缩率越大，图像质量越低。JPEG 标准的正文中并没有规定或推荐使用哪个量化表，不过在 Annex K 中有一份量化表的例子（图 6），适用于大多数中等质量的图片。对于超高质量和超低质量的图片，这份量化表不是最优的[5, Item 75]。通常亮度分量和色差分量各有一张量化表，而且对色差分量的将忽略更多的高频成分。实际的 JPEG 实现都采用这一份量化表，因此压缩后的数据中无须包含量化表。如果把下表中的量化步长除以 2，那么图像质量就接近完美了。

$$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

(a) 亮度量化表

$$\begin{bmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{bmatrix}$$

(b) 色差量化表

图 6 - 缺省量化表

每个分块有 64 个系数需要量化，需用 64 次除法和取整操作，因此每个分块的计算复杂度为 $O(1)$。对于 $N \times M$ 的图像，共有 $\lceil \frac{N}{8} \rceil \times \lceil \frac{M}{8} \rceil$ 个分块，因此量化这一步的计算复杂度为 $O(MN)$。

1.4 行程编码与熵编码

JPEG 压缩的最后一步是对量化后的系数进行熵编码。这一步采用通用的无损数据压缩技术，对图像质量没有影响。

在熵编码之前，需要把 64 个 DCT 系数转换为一串中间符号。其中直流系数和交流系数的编码方式不同。直流系数表示当前分块中 64 个象素的平均值，相邻分块的直流系数具有很强的相关性，因此在编码时只需记录与前一分块的直流系数的差值，即直流系数的“中间符号”采用差分脉冲编码(DPCM)，见图 7(a)。

对 63 个交流系数用“之”字型扫描，让它变成一维数组，见图 7(b)。这样做的目的是将低频系数放在前面，高频系数放在后面。因为高频系数中有很多 0，为了节约空间，所以交流系数的“中间符号”用零行程码 (Zero Run Length) 表示。

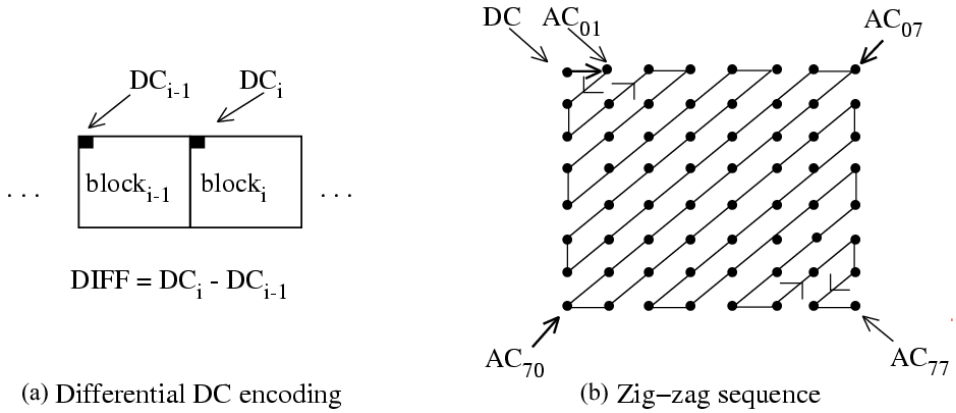

图 7 - 直流系数与交流系数的编码[1]

接下来对中间符号进行熵编码，这一步的目的是利用符号的统计特性，进一步提高压缩率。JPEG 标准规定的熵编码方式有两种：Huffman 编码和算术编码。这两种编码各有优劣，见图 8。

	Huffman 编码	算术编码
计算复杂度	小	大
压缩率	高	更高
扫描遍数	1 或 2	1
是否传码书	视情况	否

图 8 - 熵编码的比较

“正宗”的 Huffman 编码过程要对输入序列进行两遍扫描，第一遍统计各个符号出现的概率，构造 Huffman 树，得到码书 (codebook)；第二遍用码书对符号进行

编码。这么做的话，时间空间开销都较大。两遍扫描意味着要把整幅图像的“中间符号”都记录下来，不能“随到随编 (on-the-fly)”。而且要把码书传给解码器，这会增加压缩文件的大小。

JPEG 的实际实现一般支持两种 Huffman 方式，一种是前述的“正宗” Huffman 编码（称为 optimized 方式），另一种则采用 JPEG 标准 Annex K 中给出的缺省码书。采用缺省码书的好处是输入序列只用扫描一遍，空间和实际开销都较小；缺点是，由于码书不是根据当前图像的统计信息得到的，那么压缩率会比“正宗” Huffman 编码低一些。不过由于 Huffman 编码对概率误差不敏感，因此实践中常常采用缺省码书进行编码（这样还能省下保存码书的空间）。

算术编码和 Huffman 编码都是变长码，符号出现概率越高，码字越短。不同之处在于，Huffman 编码每个符号对应的码字是确定的，每个码字的 bit 数为整数。算术编码的基本思想是用一个精度足够高的属于 $[0,1)$ 的实数来表示整个输入序列，输入序列中每个符号对应的码字是不确定的，总体上每个符号对应的码字长度等于其信息熵（均以 bit 计），码字平均长度可能是小数。算术编码的压缩率通常高于 Huffman 编码[11]。算术编码的另一个好处是，它很容易做成自适应的(adaptive)，因此只需扫描一遍输入序列，空间开销小很多。算术编码还有一个小问题，IBM 掌握了一大堆与算术编码相关的专利[5, Item 8]。要想实现 JPEG 中的算术编码，不可能绕过这些专利。

Huffman 编码和算术编码在任何一本信息论[11]或数据压缩[12]教材中都能找到，其实现细节不是本文的重点。这两种编码方法的计算复杂度都和输入序列的长度成正比，每个分块最多有 64 个符号（符号数通常远小于此数，因为 DCT 系数当中有很多连续的 0），因此每个分块的复杂度为 $O(1)$。对于 $N \times M$ 的图像，熵编码的计算复杂度为 $O(MN)$。

2. 实现细节

我主要参考了两份 JPEG 实现，分别来自 Independent JPEG Group (IJG)[7] 和 Stanford Portable Video Research Group (PVRG)[8]，以下简称 ijg-jpeg 和 pvrg-jpeg。这两份实现都采用 C 语言，版本号分别为 version 6b 和 v1.2.1，最后更新时间分别为 1998 年 3 月和 1995 年 3 月。

从总体上看，ijg-jpeg 是具有工业强度的程序库，被广泛使用，久经考验；而 pvrg-jpeg 是实验性的库，主要用作研究。这两个库都很久没有更新了，因为 JPEG 标准已经出台 10 多年，库的功能都稳定了。从我阅读的体验看，ijg-jpeg 的代码要复杂得多，也完善得多，代码行数几乎是 pvrg-jpeg 的三倍。ijg-jpeg 代码中大量使用了函数指针，因此光阅读代码往往不知道它调用的到底是哪个函数，它这么做

是为了使用上的灵活性，同一步骤有多种实现（对应多个函数），在运行时选择到底采用哪种实现，并把函数指针指向选定的函数。相比之下，pvrg-jpeg 的代码要直接了当得多，结构也清晰得多。

我在 Visual C++ 2005 Express Edition 中为两个程序库分别建立了调试环境，通过单步跟踪确定 ijg-jpeg 到底调用哪个函数，在代码中标注出来，并归纳成图表。以下主要分析 ijg-jpeg，适当辅以 pvrg-jpeg。这两个函数库大体上都可分为编码器和解码器两部分。

2.1 ijg-jpeg 实现分析

下面以 ijg-jpeg 自带的 `testimg.bmp` 的压缩过程（Huffman 编码使用缺省码书）为例，介绍编码器工作流程。ijg-jpeg 编码的总体流程为：

1. 预处理
   (a) 色彩空间转换
   (b) 边界扩展
   (c) 下采样
2. JPEG 压缩
   (a) DCT
   (b) 量化
   (c) “之”字型扫描
   (d) 熵编码

编码主循环为：从输入文件读入一行行象素，送到编码器中缓存起来，编码器凑够 8 行就进行一次压缩。

main():*ijg-jpeg/cjpeg.c*

```
579  /* Start compressor */
580  jpeg_start_compress(&cinfo, TRUE);
581
582  /* Process data */
583  while (cinfo.next_scanline < cinfo.image_height) {
584    num_scanlines = (*src_mgr->get_pixel_rows) (&cinfo, src_mgr);
585    (void) jpeg_write_scanlines(&cinfo, src_mgr->buffer, num_scanlines);
586  }
587
588  /* Finish compression and release memory */
589  (*src_mgr->finish_input) (&cinfo, src_mgr);
590  jpeg_finish_compress(&cinfo);
```

main():*ijg-jpeg/cjpeg.c*

以上代码片段来自 ijg-jpeg 中 `cjpeg.c` 文件的第 579 行至 590 行，位于函数 `main()` 中（后略）。第 584 行的 `src_mgr->get_pixel_rows` 是函数指针，根据输入图像

文件类型的不同指向不同的文件读取函数。该函数的作用是读取一行象素，象素的 RGB 值保存在数组 src_mgr->buffer 中，供 jpeg_write_scanlines() 使用。

jpeg_write_scanlines():*ijg-jpeg/jcapistd.c*

```
107   row_ctr = 0;
108   (*cinfo->main->process_data) (cinfo, scanlines, &row_ctr, num_lines);
109   cinfo->next_scanline += row_ctr;
110   return row_ctr;
```

jpeg_write_scanlines():*ijg-jpeg/jcapistd.c*

而 jpeg_write_scanlines() 把实际工作都交给 cinfo->main->process_data 处理，这个函数指针只会指向 process_data_simple_main()。

process_data_simple_main():*ijg-jpeg/jcmainct.c*

```
106 /*
107  * Process some data.
108  * This routine handles the simple pass-through mode,
109  * where we have only a strip buffer.
110  */
111
112 METHODDEF(void)
113 process_data_simple_main (j_compress_ptr cinfo,
114                           JSAMPARRAY input_buf, JDIMENSION *in_row_ctr,
115                           JDIMENSION in_rows_avail)
116 {
117   my_main_ptr main = (my_main_ptr) cinfo->main;
118
119   while (main->cur_iMCU_row < cinfo->total_iMCU_rows) {
120     /* Read input data if we haven't filled the main buffer yet */
121     if (main->rowgroup_ctr < DCTSIZE)   // DCTSIZE == 8
122       (*cinfo->prep->pre_process_data) (cinfo,
123                                         input_buf, in_row_ctr, in_rows_avail,
124                                         main->buffer, &main->rowgroup_ctr,
125                                         (JDIMENSION) DCTSIZE);
126
127     /* If we don't have a full iMCU row buffered, return to application for
128      * more data.  Note that preprocessor will always pad to fill the iMCU row
129      * at the bottom of the image.
130      */
131     if (main->rowgroup_ctr != DCTSIZE)
132       return;
133
134     /* Send the completed row to the compressor */
135     if (! (*cinfo->coef->compress_data) (cinfo, main->buffer)) {
136       /* If compressor did not consume the whole row, then we must need to
137        * suspend processing and return to the application.  In this situation
138        * we pretend we didn't yet consume the last input row; otherwise, if
139        * it happened to be the last row of the image, the application would
140        * think we were done.
141        */
142       if (! main->suspended) {
143         (*in_row_ctr)--;
144         main->suspended = TRUE;
145       }
146       return;
```

```
147     }
148     /* We did finish the row.  Undo our little suspension hack if a previous
149      * call suspended; then mark the main buffer empty.
150      */
151     if (main->suspended) {
152       (*in_row_ctr)++;
153       main->suspended = FALSE;
154     }
155     main->rowgroup_ctr = 0;
156     main->cur_iMCU_row++;
157   }
158 }
```

process_data_simple_main():*ijg-jpeg/jcmainct.c*

从这个函数可以看出，每读入一行象素，就进行一次预处理（第 122 行），放入缓冲区。如果缓冲区中的数据不够 8 行则返回（第 131 行），否则就进行压缩（第 135 行）。

第 122 行的 `cinfo->prep->pre_process_data` 指向 `pre_process_data()`。

第 135 行的 `cinfo->coef->compress_data` 指向 `compress_data()`。

`process_data_simple_main()` 的函数调用关系为（红色方块为函数指针，椭圆为普通函数）：

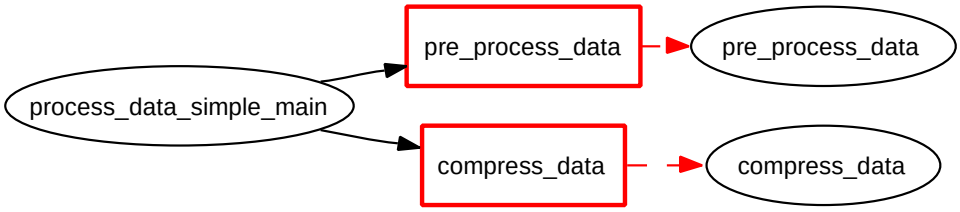

图 9 - `process_data_simple_main()` 的调用关系

预处理

预处理在 `pre_process_data()` 中进行，主要做色彩转换、边界扩展、下采样，它的函数调用关系为：

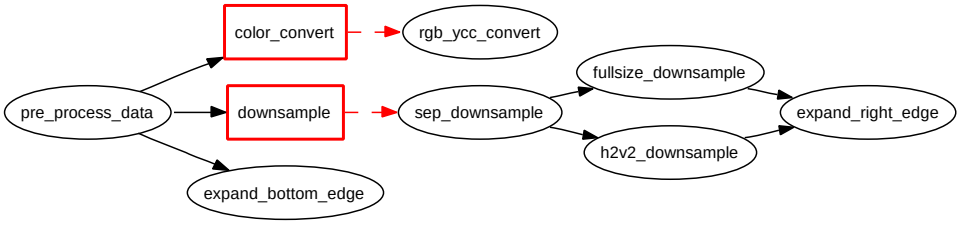

图 10 - `pre_process_data()` 的调用关系

pre_process_data():*ijg-jpeg/jcprepct.c*

```
/*
 * Process some data in the simple no-context case.
 *
 * Preprocessor output data is counted in "row groups".  A row group
 * is defined to be v_samp_factor sample rows of each component.
 * Downsampling will produce this much data from each max_v_samp_factor
 * input rows.
 */

METHODDEF(void)
pre_process_data (j_compress_ptr cinfo,
                  JSAMPARRAY input_buf, JDIMENSION *in_row_ctr,
                  JDIMENSION in_rows_avail,
                  JSAMPIMAGE output_buf, JDIMENSION *out_row_group_ctr,
                  JDIMENSION out_row_groups_avail)
{
  my_prep_ptr prep = (my_prep_ptr) cinfo->prep;
  int numrows, ci;
  JDIMENSION inrows;
  jpeg_component_info * compptr;

  while (*in_row_ctr < in_rows_avail &&
         *out_row_group_ctr < out_row_groups_avail) {
    /* Do color conversion to fill the conversion buffer. */
    inrows = in_rows_avail - *in_row_ctr;
    numrows = cinfo->max_v_samp_factor - prep->next_buf_row;
    numrows = (int) MIN((JDIMENSION) numrows, inrows);
    (*cinfo->cconvert->color_convert) (cinfo, input_buf + *in_row_ctr,
                                       prep->color_buf,
                                       (JDIMENSION) prep->next_buf_row,
                                       numrows);
    *in_row_ctr += numrows;
    prep->next_buf_row += numrows;
    prep->rows_to_go -= numrows;
    /* If at bottom of image, pad to fill the conversion buffer. */
    if (prep->rows_to_go == 0 &&
        prep->next_buf_row < cinfo->max_v_samp_factor) {
      for (ci = 0; ci < cinfo->num_components; ci++) {
        expand_bottom_edge(prep->color_buf[ci], cinfo->image_width,
                           prep->next_buf_row, cinfo->max_v_samp_factor);
      }
      prep->next_buf_row = cinfo->max_v_samp_factor;
    }
    /* If we've filled the conversion buffer, empty it. */
    if (prep->next_buf_row == cinfo->max_v_samp_factor) {
      (*cinfo->downsample->downsample) (cinfo,
                                        prep->color_buf, (JDIMENSION) 0,
                                        output_buf, *out_row_group_ctr);
      prep->next_buf_row = 0;
      (*out_row_group_ctr)++;
    }
    /* If at bottom of image, pad the output to a full iMCU height.
     * Note we assume the caller is providing a one-iMCU-height output buffer!
     */
    if (prep->rows_to_go == 0 &&
        *out_row_group_ctr < out_row_groups_avail) {
```

```
174       for (ci = 0, compptr = cinfo->comp_info; ci < cinfo->num_components;
175            ci++, compptr++) {
176         expand_bottom_edge(output_buf[ci],
177                            compptr->width_in_blocks * DCTSIZE,
178                            (int) (*out_row_group_ctr * compptr->v_samp_factor),
179                            (int) (out_row_groups_avail * compptr->v_samp_factor));
180       }
181       *out_row_group_ctr = out_row_groups_avail;
182       break;                    /* can exit outer loop without test */
183     }
184   }
185 }
```

pre_process_data():*ijg-jpeg/jcprepct.c*

这个函数中只做了下边界扩展 `expand_bottom_edge()`，而右边界扩展留在下采样时进行。

第 145 行的 `cinfo->cconvert->color_convert` 指向 `rgb_ycc_convert()`。

第 163 行的 `cinfo->downsample->downsample` 指向 `sep_downsample()`。

色彩转换

RGB → YCbCr 的色彩转换由 `rgb_ycc_convert()` 负责，转换公式在第 3 页给出。这里为了加快速度，预先把乘积算出来存入 9 张表中 “`rgb_ycc_start()`”，在转换时只用查表操作。

rgb_ycc_convert():*ijg-jpeg/jccolor.c*

```
148   for (col = 0; col < num_cols; col++) {
149     r = GETJSAMPLE(inptr[RGB_RED]);
150     g = GETJSAMPLE(inptr[RGB_GREEN]);
151     b = GETJSAMPLE(inptr[RGB_BLUE]);
152     inptr += RGB_PIXELSIZE;
153     /* If the inputs are 0..MAXJSAMPLE, the outputs of these equations
154      * must be too; we do not need an explicit range-limiting operation.
155      * Hence the value being shifted is never negative, and we don't
156      * need the general RIGHT_SHIFT macro.
157      */
158     /* Y */
159     outptr0[col] = (JSAMPLE)
160                 ((ctab[r+R_Y_OFF] + ctab[g+G_Y_OFF] + ctab[b+B_Y_OFF])
161                  >> SCALEBITS);
162     /* Cb */
163     outptr1[col] = (JSAMPLE)
164                 ((ctab[r+R_CB_OFF] + ctab[g+G_CB_OFF] + ctab[b+B_CB_OFF])
165                  >> SCALEBITS);
166     /* Cr */
167     outptr2[col] = (JSAMPLE)
168                 ((ctab[r+R_CR_OFF] + ctab[g+G_CR_OFF] + ctab[b+B_CR_OFF])
169                  >> SCALEBITS);
170   }
```

rgb_ycc_convert():*ijg-jpeg/jccolor.c*

扩展边界

在 ijg-jpeg 的具体实现中，它先扩展每一行的右边界“expand_right_edge()”，让行的长度（也就是图像的宽度）等于 8 的倍数。然后在图像底部填充一些行，扩展其下边界“expand_bottom_edge()”，让整个图像的高度是 8 的倍数。这样得到扩展之后的图像，它的宽度和高度都是 8 的倍数，便于分块。

expand_right_edge():*ijg-jpeg/jcsample.c*

```
/*
 * Expand a component horizontally from width input_cols to width output_cols,
 * by duplicating the rightmost samples.
 */

LOCAL(void)
expand_right_edge (JSAMPARRAY image_data, int num_rows,
                   JDIMENSION input_cols, JDIMENSION output_cols)
{
  register JSAMPROW ptr;
  register JSAMPLE pixval;
  register int count;
  int row;
  int numcols = (int) (output_cols - input_cols);

  if (numcols > 0) {
    for (row = 0; row < num_rows; row++) {
      ptr = image_data[row] + input_cols;
      pixval = ptr[-1];            /* don't need GETJSAMPLE() here */
      for (count = numcols; count > 0; count--)
        *ptr++ = pixval;
    }
  }
}
```

expand_right_edge():*ijg-jpeg/jcsample.c*

expand_bottom_edge():*ijg-jpeg/jcprepct.c*

```
/*
 * Expand an image vertically from height input_rows to height output_rows,
 * by duplicating the bottom row.
 */

LOCAL(void)
expand_bottom_edge (JSAMPARRAY image_data, JDIMENSION num_cols,
                    int input_rows, int output_rows)
{
  register int row;

  for (row = input_rows; row < output_rows; row++) {
    jcopy_sample_rows(image_data, input_rows-1, image_data, row,
                      1, num_cols);
  }
}
```

expand_bottom_edge():*ijg-jpeg/jcprepct.c*

下采样

下采样的工作由 sep_downsample() 分派给别的函数来做，第 127 行用到的 downsample->methods[] 是函数指针数组，指向各分量所用的下采样函数。

亮度分量无需下采样，于是调用 fullsize_downsample()。色差分量有 4:1:1 和 4:2:2 两种下采样方式，分别对应 h2v2_downsample() 和 h2v1_downsample()。这里实际调用的是 h2v2_downsample()。

sep_downsample():*ijg-jpeg/jcsample.c*

```
107 /*
108  * Do downsampling for a whole row group (all components).
109  *
110  * In this version we simply downsample each component independently.
111  */
112 
113 METHODDEF(void)
114 sep_downsample (j_compress_ptr cinfo,
115                 JSAMPIMAGE input_buf, JDIMENSION in_row_index,
116                 JSAMPIMAGE output_buf, JDIMENSION out_row_group_index)
117 {
118   my_downsample_ptr downsample = (my_downsample_ptr) cinfo->downsample;
119   int ci;
120   jpeg_component_info * compptr;
121   JSAMPARRAY in_ptr, out_ptr;
122 
123   for (ci = 0, compptr = cinfo->comp_info; ci < cinfo->num_components;
124        ci++, compptr++) {
125     in_ptr = input_buf[ci] + in_row_index;
126     out_ptr = output_buf[ci] + (out_row_group_index * compptr->v_samp_factor);
127     (*downsample->methods[ci]) (cinfo, compptr, in_ptr, out_ptr);
128   }
129 }
```

sep_downsample():*ijg-jpeg/jcsample.c*

fullsize_downsample() 只是简单地将输入拷贝到输出，并做右边界扩展。

fullsize_downsample():*ijg-jpeg/jcsample.c*

```
180 /*
181  * Downsample pixel values of a single component.
182  * This version handles the special case of a full-size component,
183  * without smoothing.
184  */
185 
186 METHODDEF(void)
187 fullsize_downsample (j_compress_ptr cinfo, jpeg_component_info * compptr,
188                      JSAMPARRAY input_data, JSAMPARRAY output_data)
189 {
190   /* Copy the data */
191   jcopy_sample_rows(input_data, 0, output_data, 0,
192                     cinfo->max_v_samp_factor, cinfo->image_width);
193   /* Edge-expand */
194   expand_right_edge(output_data, cinfo->max_v_samp_factor,
195                     cinfo->image_width, compptr->width_in_blocks * DCTSIZE);
196 }
```

fullsize_downsample():*ijg-jpeg/jcsample.c*

`h2v2_downsample()` 把输入图像分量按 2×2 分块，计算每块样本的平均值，送到输出中（第 272 行）。这样得到的样本数为原来的 1/4。

h2v2_downsample():*ijg-jpeg/jcsample.c*

```
242 /*
243  * Downsample pixel values of a single component.
244  * This version handles the standard case of 2:1 horizontal and 2:1 vertical,
245  * without smoothing.
246  */
247 
248 METHODDEF(void)
249 h2v2_downsample (j_compress_ptr cinfo, jpeg_component_info * compptr,
250                  JSAMPARRAY input_data, JSAMPARRAY output_data)
251 {
252   int inrow, outrow;
253   JDIMENSION outcol;
254   JDIMENSION output_cols = compptr->width_in_blocks * DCTSIZE;
255   register JSAMPROW inptr0, inptr1, outptr;
256   register int bias;
257 
258   /* Expand input data enough to let all the output samples be generated
259    * by the standard loop.  Special-casing padded output would be more
260    * efficient.
261    */
262   expand_right_edge(input_data, cinfo->max_v_samp_factor,
263                     cinfo->image_width, output_cols * 2);
264 
265   inrow = 0;
266   for (outrow = 0; outrow < compptr->v_samp_factor; outrow++) {
267     outptr = output_data[outrow];
268     inptr0 = input_data[inrow];
269     inptr1 = input_data[inrow+1];
270     bias = 1;                   /* bias = 1,2,1,2,... for successive samples */
271     for (outcol = 0; outcol < output_cols; outcol++) {
272       *outptr++ = (JSAMPLE) ((GETJSAMPLE(*inptr0) + GETJSAMPLE(inptr0[1]) +
273                               GETJSAMPLE(*inptr1) + GETJSAMPLE(inptr1[1])
274                               + bias) >> 2);
275       bias ^= 3;                /* 1=>2, 2=>1 */
276       inptr0 += 2; inptr1 += 2;
277     }
278     inrow += 2;
279   }
280 }
```

h2v2_downsample():*ijg-jpeg/jcsample.c*

设输入为 $N \times M$ 的 RGB 图像，共有 $3MN$ 个输入样本；经过色彩转换后，Y、Cr、Cb 分量各有 MN 个样本；再经过下采样，Y 分量有 MN 个样本，Cr、Cb 分量各有大约 $\frac{N}{2} \cdot \frac{M}{2}$ 个样本。输出样本总数约为 $\frac{3}{2}MN$，是输入的一半。

预处理中的每一个步骤的计算复杂度都与输入样本的个数成正比，对于 $N \times M$ 的图像，预处理的计算复杂度为 $O(MN)$。

JPEG 压缩

预处理之后，JPEG 压缩由 `compress_data()` 实现。它调用 `forward_DCT()` 做离散余弦变换和量化。它在调用 `forward_DCT()` 之前，会对各色彩分量进行交织。所谓“交织”，就是说不是压缩完 Y 分量再压缩 Cr 和 Cb 分量，而是每次从 Y 分量中选几个块，和 Cr、Cb 中选出的分块组成 MCU (Minimum Coded Unit，最小编码单元)。然后对 MCU 中的分块进行 DCT，再以 MCU 为单位进行熵编码。交织的主要目的是为了在解码时能一边解压一边显示（或进行后续处理），而不用等整幅图片都解压完才显示。（光解出 Y 分量和 Cr 分量是无法显示的。）

`compress_data()` 然后调用 `encode_mcu_huff()` 对一个 MCU 进行 Huffman 编码。后者会对 MCU 中的各个分块调用 `encode_one_block()`，由它来进行 Huffman 编码，这个函数还顺便进行“之”字型扫描。

`compress_data()` 的函数调用关系为：

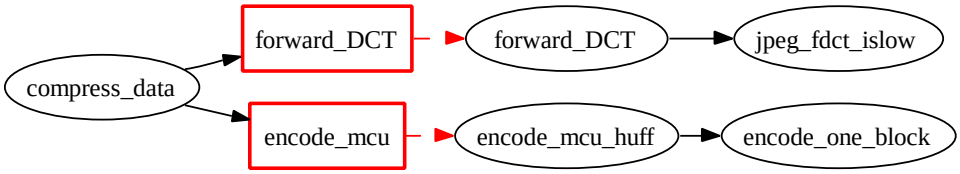

图 11 - `compress_data()` 的调用关系

`compress_data()` 比较长，我没有把它完整列出来。

compress_data():*ijg-jpeg/jccoefct.c*

```
132 /*
133  * Process some data in the single-pass case.
134  * We process the equivalent of one fully interleaved MCU row ("iMCU" row)
135  * per call, ie, v_samp_factor block rows for each component in the image.
136  * Returns TRUE if the iMCU row is completed, FALSE if suspended.
137  *
138  * NB: input_buf contains a plane for each component in image,
139  * which we index according to the component's SOF position.
140  */
141
142 METHODDEF(boolean)
143 compress_data (j_compress_ptr cinfo, JSAMPIMAGE input_buf)
144 {
```

// 对每个 MCU 中的每个分块：

```
177             (*cinfo->fdct->forward_DCT) (cinfo, compptr,
178                                          input_buf[compptr->component_index],
179                                          coef->MCU_buffer[blkn],
180                                          ypos, xpos, (JDIMENSION) blockcnt);
```

// 对每个 MCU：

```
201       /* Try to write the MCU.  In event of a suspension failure, we will
202        * re-DCT the MCU on restart (a bit inefficient, could be fixed...)
203        */
204       if (! (*cinfo->entropy->encode_mcu) (cinfo, coef->MCU_buffer)) {
205         /* Suspension forced; update state counters and exit */
206         coef->MCU_vert_offset = yoffset;
207         coef->mcu_ctr = MCU_col_num;
208         return FALSE;
209       }

217   return TRUE;
218 }
```

compress_data():*ijg-jpeg/jccoefct.c*

第 177 行的 `cinfo->fdct->forward_DCT` 指向 `forward_DCT()`。

第 204 行的 `cinfo->entropy->encode_mcu` 指向 `encode_mcu_huff()`。

DCT 与量化

`forward_DCT` 把 DCT 运算交给 `do_dct` 所指向的函数来做（第 224 行），它自己只做量化，这个函数的调用频率非常高，所以写得很紧凑。

forward_DCT():*ijg-jpeg/jcdctmgr.c*

```
171 /*
172  * Perform forward DCT on one or more blocks of a component.
173  *
174  * The input samples are taken from the sample_data[] array starting at
175  * position start_row/start_col, and moving to the right for any additional
176  * blocks. The quantized coefficients are returned in coef_blocks[].
177  */
178
179 METHODDEF(void)
180 forward_DCT (j_compress_ptr cinfo, jpeg_component_info * compptr,
181              JSAMPARRAY sample_data, JBLOCKROW coef_blocks,
182              JDIMENSION start_row, JDIMENSION start_col,
183              JDIMENSION num_blocks)
184 /* This version is used for integer DCT implementations. */
185 {
186   /* This routine is heavily used, so it's worth coding it tightly. */
187   my_fdct_ptr fdct = (my_fdct_ptr) cinfo->fdct;
188   forward_DCT_method_ptr do_dct = fdct->do_dct;
189   DCTELEM * divisors = fdct->divisors[compptr->quant_tbl_no];
190   DCTELEM workspace[DCTSIZE2];     /* work area for FDCT subroutine */
191   JDIMENSION bi;                   // DCTSIZE2 = 64
```

// 第 192～222 行准备 `workspace`，并将样本取值范围从 $[0, 255]$ 移到 $[-128, 127]$。

```
223     /* Perform the DCT */
224     (*do_dct) (workspace);
225
226     /* Quantize/descale the coefficients, and store into coef_blocks[] */
227     { register DCTELEM temp, qval;
228       register int i;
229       register JCOEFPTR output_ptr = coef_blocks[bi];
```

```
230
231       for (i = 0; i < DCTSIZE2; i++) {
232         qval = divisors[i];
233         temp = workspace[i];
234         /* Divide the coefficient value by qval, ensuring proper rounding.
235          * Since C does not specify the direction of rounding for negative
236          * quotients, we have to force the dividend positive for portability.
237          *
238          * In most files, at least half of the output values will be zero
239          * (at default quantization settings, more like three-quarters...)
240          * so we should ensure that this case is fast.  On many machines,
241          * a comparison is enough cheaper than a divide to make a special test
242          * a win.  Since both inputs will be nonnegative, we need only test
243          * for a < b to discover whether a/b is 0.
244          * If your machine's division is fast enough, define FAST_DIVIDE.
245          */
246 #ifdef FAST_DIVIDE
247 #define DIVIDE_BY(a,b)  a /= b
248 #else
249 #define DIVIDE_BY(a,b)  if (a >= b) a /= b; else a = 0
250 #endif
251         if (temp < 0) {
252           temp = -temp;
253           temp += qval>>1;      /* for rounding */
254           DIVIDE_BY(temp, qval);
255           temp = -temp;
256         } else {
257           temp += qval>>1;      /* for rounding */
258           DIVIDE_BY(temp, qval);
259         }
260         output_ptr[i] = (JCOEF) temp;
261       }
262     }
263   }
264 }
```

forward_DCT():*ijg-jpeg/jcdctmgr.c*

量化是整个 JPEG 压缩中惟一用到除法的地方，而各种机器的整数除法略有区别，因此需要特殊对待，代码中的注释说得很清楚了。通常 C 语言中的整数除法是向下取整，并非四舍五入，因此第 253 行和第 257 行先给被除数加上除数的一半，这样就能实现四舍五入。

第 224 行的 `do_dct` 指向快速 DCT 的具体实现，ijg-jpeg 提供有 C. Loeffler、A. Ligtenberg、G. Moschytz 等的实现（`jpeg_fdct_islow`）和 Arai、Agui、Nakajima 等的实现（`jpeg_fdct_ifast`）。这里实际调用的是 `jpeg_fdct_islow`。这两份实现都是就地 (in-place) 计算的，代码从略。

`forward_DCT()` 的运行时间为常数，它被调用的次数等于输入图像的分块数。对于 $N \times M$ 的图像，DCT 的计算复杂度为 $O(MN)$。

在执行完 `forward_DCT` 之后，就该做熵编码了。

“之” 字型扫描与熵编码

`compress_data()` 在第 204 行通过函数指针 `encode_mcu` 调用 `encode_mcu_huff()` 进行 Huffman 编码。`encode_mcu_huff()` 对 MCU 中的各个分块调用 `encode_one_block()`，“之” 字型扫描和 Huffman 编码的具体工作由后者完成。

encode_mcu_huff():*ijg-jpeg/jchuff.c*

```
496   /* Encode the MCU data blocks */
497   for (blkn = 0; blkn < cinfo->blocks_in_MCU; blkn++) {
498     ci = cinfo->MCU_membership[blkn];
499     compptr = cinfo->cur_comp_info[ci];
500     if (! encode_one_block(&state,
501                            MCU_data[blkn][0], state.cur.last_dc_val[ci],
502                            entropy->dc_derived_tbls[compptr->dc_tbl_no],
503                            entropy->ac_derived_tbls[compptr->ac_tbl_no]))
504       return FALSE;
505     /* Update last_dc_val */
506     state.cur.last_dc_val[ci] = MCU_data[blkn][0][0];
507   }
```

encode_mcu_huff():*ijg-jpeg/jchuff.c*

注意在以上代码中，直流和交流的码书不同，各个色彩分量的码书也可能不同。

`encode_one_block()` 很长，它的实际作用是产生 “中间符号”，然后调用 `emit_bits()` 输出 bit 流。直流系数采用差分编码（第 362 行）。

encode_one_block():*ijg-jpeg/jchuff.c*

```
350 /* Encode a single block's worth of coefficients */
351
352 LOCAL(boolean)
353 encode_one_block (working_state * state, JCOEFPTR block, int last_dc_val,
354                   c_derived_tbl *dctbl, c_derived_tbl *actbl)
355 {
356   register int temp, temp2;
357   register int nbits;
358   register int k, r, i;
359
360   /* Encode the DC coefficient difference per section F.1.2.1 */
361
362   temp = temp2 = block[0] - last_dc_val;
363
364   if (temp < 0) {
365     temp = -temp;               /* temp is abs value of input */
366     /* For a negative input, want temp2 = bitwise complement of abs(input) */
367     /* This code assumes we are on a two's complement machine */
368     temp2--;
369   }
370
371   /* Find the number of bits needed for the magnitude of the coefficient */
372   nbits = 0;
373   while (temp) {
374     nbits++;
375     temp >>= 1;
376   }
```

```
377   /* Check for out-of-range coefficient values.
378    * Since we're encoding a difference, the range limit is twice as much.
379    */
380   if (nbits > MAX_COEF_BITS+1)
381     ERREXIT(state->cinfo, JERR_BAD_DCT_COEF);
382
383   /* Emit the Huffman-coded symbol for the number of bits */
384   if (! emit_bits(state, dctbl->ehufco[nbits], dctbl->ehufsi[nbits]))
385     return FALSE;
386
387   /* Emit that number of bits of the value, if positive, */
388   /* or the complement of its magnitude, if negative. */
389   if (nbits)                          /* emit_bits rejects calls with size 0 */
390     if (! emit_bits(state, (unsigned int) temp2, nbits))
391       return FALSE;
392
```

encode_one_block():*ijg-jpeg/jchuff.c*

交流系数先进行“之”字型扫描，再做零行程编码，这两步集中在 397～399 三行。然后构造中间符号，并对其进行 Huffman 编码（第 400 行起）。

encode_one_block():*ijg-jpeg/jchuff.c*

```
393   /* Encode the AC coefficients per section F.1.2.2 */
394
395   r = 0;                          /* r = run length of zeros */
396
397   for (k = 1; k < DCTSIZE2; k++) {
398     if ((temp = block[jpeg_natural_order[k]]) == 0) {
399       r++;
400     } else {
401       /* if run length > 15, must emit special run-length-16 codes (0xF0) */
402       while (r > 15) {
403         if (! emit_bits(state, actbl->ehufco[0xF0], actbl->ehufsi[0xF0]))
404           return FALSE;
405         r -= 16;
406       }
407
408       temp2 = temp;
409       if (temp < 0) {
410         temp = -temp;                 /* temp is abs value of input */
411         /* This code assumes we are on a two's complement machine */
412         temp2--;
413       }
414
415       /* Find the number of bits needed for the magnitude of the coefficient */
416       nbits = 1;                  /* there must be at least one 1 bit */
417       while ((temp >>= 1))
418         nbits++;
419       /* Check for out-of-range coefficient values */
420       if (nbits > MAX_COEF_BITS)
421         ERREXIT(state->cinfo, JERR_BAD_DCT_COEF);
422
423       /* Emit Huffman symbol for run length / number of bits */
424       i = (r << 4) + nbits;
425       if (! emit_bits(state, actbl->ehufco[i], actbl->ehufsi[i]))
426         return FALSE;
```

```
427
428         /* Emit that number of bits of the value, if positive, */
429         /* or the complement of its magnitude, if negative. */
430         if (! emit_bits(state, (unsigned int) temp2, nbits))
431           return FALSE;
432
433         r = 0;
434       }
435     }
436
437     /* If the last coef(s) were zero, emit an end-of-block code */
438     if (r > 0)
439       if (! emit_bits(state, actbl->ehufco[0], actbl->ehufsi[0]))
440         return FALSE;
441
442     return TRUE;
443 }
```

encode_one_block():*ijg-jpeg/jchuff.c*

以上代码进行“之”字型扫描的方法很巧妙，它使用了一个数组 jpeg_natural_order[]，记录“之”字型扫描序列中第 i 个元素在原分块中的位置，例如第 2 个元素的位置是 8。这个数组定义在文件 jutils.c 中。

jpeg_natural_order[]:*ijg-jpeg/jutils.c*

```
39 /*
40  * jpeg_natural_order[i] is the natural-order position of the i'th element
41  * of zigzag order.
42  *
43  * When reading corrupted data, the Huffman decoders could attempt
44  * to reference an entry beyond the end of this array (if the decoded
45  * zero run length reaches past the end of the block).  To prevent
46  * wild stores without adding an inner-loop test, we put some extra
47  * "63"s after the real entries.  This will cause the extra coefficient
48  * to be stored in location 63 of the block, not somewhere random.
49  * The worst case would be a run-length of 15, which means we need 16
50  * fake entries.
51  */
52
53 const int jpeg_natural_order[DCTSIZE2+16] = {
54   0,  1,  8, 16,  9,  2,  3, 10,
55  17, 24, 32, 25, 18, 11,  4,  5,
56  12, 19, 26, 33, 40, 48, 41, 34,
57  27, 20, 13,  6,  7, 14, 21, 28,
58  35, 42, 49, 56, 57, 50, 43, 36,
59  29, 22, 15, 23, 30, 37, 44, 51,
60  58, 59, 52, 45, 38, 31, 39, 46,
61  53, 60, 61, 54, 47, 55, 62, 63,
62  63, 63, 63, 63, 63, 63, 63, 63, /* extra entries for safety in decoder */
63  63, 63, 63, 63, 63, 63, 63, 63
64 };
```

jpeg_natural_order[]:*ijg-jpeg/jutils.c*

encode_one_block() 的运行时间与输入有关，被调用次数等于图像分块数。在最坏情况下（所有系数均不为 0），它调用 128 次 emit_bits()，因此运行时间

小于某一常数。对于 $N \times M$ 的图像，熵编码的计算复杂度为 $O(MN)$。

至此，整个 JPEG 编码过程结束。根据以上分析，整个 JPEG 压缩（含预处理）的运行时间与输入数据的大小成线性关系。对于 $N \times M$ 的图像，JPEG 压缩的计算复杂度为 $O(MN)$。

量化表的缩放

前面提到，整个 JPEG 编码过程中惟一控制压缩质量的办法是调整量化表。设置压缩质量的函数是 `jpeg_set_quality()`，用户期望的压缩质量用 0～100 间的整数表示（称为质量百分数）。

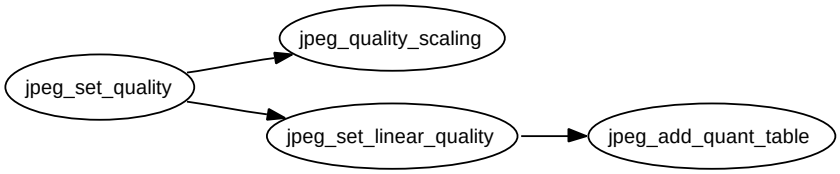

图 12 - `jpeg_set_quality()` 的调用关系

jpeg_set_quality():*ijg-jpeg/jcparam.c*

```
131 GLOBAL(void)
132 jpeg_set_quality (j_compress_ptr cinfo, int quality, boolean force_baseline)
133 /* Set or change the 'quality' (quantization) setting, using default tables.
134  * This is the standard quality-adjusting entry point for typical user
135  * interfaces; only those who want detailed control over quantization tables
136  * would use the preceding three routines directly.
137  */
138 {
139   /* Convert user 0-100 rating to percentage scaling */
140   quality = jpeg_quality_scaling(quality);
141
142   /* Set up standard quality tables */
143   jpeg_set_linear_quality(cinfo, quality, force_baseline);
144 }
```

jpeg_set_quality():*ijg-jpeg/jcparam.c*

`jpeg_set_quality()` 调用 `jpeg_quality_scaling()` 将用户输入的质量百分数 `quality` 转换为量化步长的缩放百分数 `scale_factor`。如果质量百分数为 75（这是默认设置），那么对应的量化步长缩放百分数为 50，表明实际量化步长是标准步长的 50%。

jpeg_quality_scaling():*ijg-jpeg/jcparam.c*

```
105 GLOBAL(int)
106 jpeg_quality_scaling (int quality)
107 /* Convert a user-specified quality rating to a percentage scaling factor
108  * for an underlying quantization table, using our recommended scaling curve.
109  * The input 'quality' factor should be 0 (terrible) to 100 (very good).
110  */
```

```
{
  /* Safety limit on quality factor.  Convert 0 to 1 to avoid zero divide. */
  if (quality <= 0) quality = 1;
  if (quality > 100) quality = 100;

  /* The basic table is used as-is (scaling 100) for a quality of 50.
   * Qualities 50..100 are converted to scaling percentage 200 - 2*Q;
   * note that at Q=100 the scaling is 0, which will cause jpeg_add_quant_table
   * to make all the table entries 1 (hence, minimum quantization loss).
   * Qualities 1..50 are converted to scaling percentage 5000/Q.
   */
  if (quality < 50)
    quality = 5000 / quality;
  else
    quality = 200 - quality*2;

  return quality;
}
```
jpeg_quality_scaling():*ijg-jpeg/jcparam.c*

jpeg_set_quality() 接下来调用 jpeg_set_linear_quality()，后者调用 jpeg_add_quant_table() 做实际的缩放操作，并装载量化表。从以下代码看出 ijg-jpeg 使用量化表与 JPEG 标准 Annex K 中的一致（图 6），而且亮度与色度各有一张量化表。

jpeg_set_linear_quality():*ijg-jpeg/jcparam.c*
```
GLOBAL(void)
jpeg_set_linear_quality (j_compress_ptr cinfo, int scale_factor,
                         boolean force_baseline)
/* Set or change the 'quality' (quantization) setting, using default tables
 * and a straight percentage-scaling quality scale.  In most cases it's better
 * to use jpeg_set_quality (below); this entry point is provided for
 * applications that insist on a linear percentage scaling.
 */
{
  /* These are the sample quantization tables given in JPEG spec section K.1.
   * The spec says that the values given produce "good" quality, and
   * when divided by 2, "very good" quality.
   */
  static const unsigned int std_luminance_quant_tbl[DCTSIZE2] = {
    16,  11,  10,  16,  24,  40,  51,  61,
    12,  12,  14,  19,  26,  58,  60,  55,
    14,  13,  16,  24,  40,  57,  69,  56,
    14,  17,  22,  29,  51,  87,  80,  62,
    18,  22,  37,  56,  68, 109, 103,  77,
    24,  35,  55,  64,  81, 104, 113,  92,
    49,  64,  78,  87, 103, 121, 120, 101,
    72,  92,  95,  98, 112, 100, 103,  99
  };
  static const unsigned int std_chrominance_quant_tbl[DCTSIZE2] = {
    17,  18,  24,  47,  99,  99,  99,  99,
    18,  21,  26,  66,  99,  99,  99,  99,
    24,  26,  56,  99,  99,  99,  99,  99,
    47,  66,  99,  99,  99,  99,  99,  99,
    99,  99,  99,  99,  99,  99,  99,  99,
```

```
91      99,  99,  99,  99,  99,  99,  99,  99,
92      99,  99,  99,  99,  99,  99,  99,  99,
93      99,  99,  99,  99,  99,  99,  99,  99
94    };
95
96    /* Set up two quantization tables using the specified scaling */
97    jpeg_add_quant_table(cinfo, 0, std_luminance_quant_tbl,
98                         scale_factor, force_baseline);
99    jpeg_add_quant_table(cinfo, 1, std_chrominance_quant_tbl,
100                        scale_factor, force_baseline);
101 }
```

jpeg_set_linear_quality():*ijg-jpeg/jcparam.c*

缩放操作在 jpeg_add_quant_table() 第 49 行。

jpeg_add_quant_table():*ijg-jpeg/jcparam.c*

```
22 GLOBAL(void)
23 jpeg_add_quant_table (j_compress_ptr cinfo, int which_tbl,
24                       const unsigned int *basic_table,
25                       int scale_factor, boolean force_baseline)
26 /* Define a quantization table equal to the basic_table times
27  * a scale factor (given as a percentage).
28  * If force_baseline is TRUE, the computed quantization table entries
29  * are limited to 1..255 for JPEG baseline compatibility.
30  */
31 {
32   JQUANT_TBL ** qtblptr;
33   int i;
34   long temp;
35
36   /* Safety check to ensure start_compress not called yet. */
37   if (cinfo->global_state != CSTATE_START)
38     ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);
39
40   if (which_tbl < 0 || which_tbl >= NUM_QUANT_TBLS)
41     ERREXIT1(cinfo, JERR_DQT_INDEX, which_tbl);
42
43   qtblptr = & cinfo->quant_tbl_ptrs[which_tbl];
44
45   if (*qtblptr == NULL)
46     *qtblptr = jpeg_alloc_quant_table((j_common_ptr) cinfo);
47
48   for (i = 0; i < DCTSIZE2; i++) {
49     temp = ((long) basic_table[i] * scale_factor + 50L) / 100L;
50     /* limit the values to the valid range */
51     if (temp <= 0L) temp = 1L;
52     if (temp > 32767L) temp = 32767L; /* max quantizer needed for 12 bits */
53     if (force_baseline && temp > 255L)
54       temp = 255L;                /* limit to baseline range if requested */
55     (*qtblptr)->quantval[i] = (UINT16) temp;
56   }
57
58   /* Initialize sent_table FALSE so table will be written to JPEG file. */
59   (*qtblptr)->sent_table = FALSE;
60 }
```

jpeg_add_quant_table():*ijg-jpeg/jcparam.c*

以上设置量化表的执行时间为常数。

ijg-jpeg 压缩编码的完整调用关系见下图（略去函数指针的间接层），对这个图进行深度优先搜索就能得到函数的实际执行顺序。

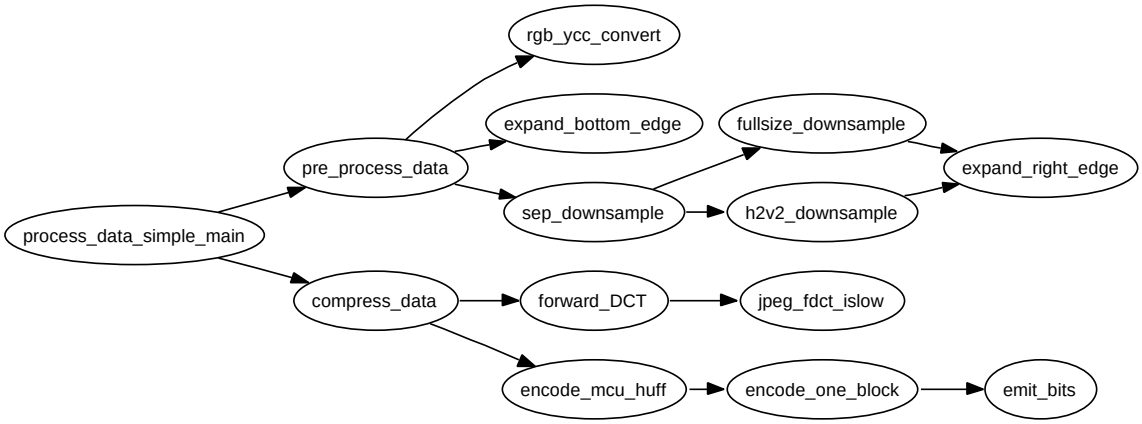

图 13 - ijg-jpeg 完整调用关系

2.2 pvrg-jpeg 实现分析

pvrg-jpeg 的结构比 ijg-jpeg 要简单得多，代码更清晰，容易阅读。

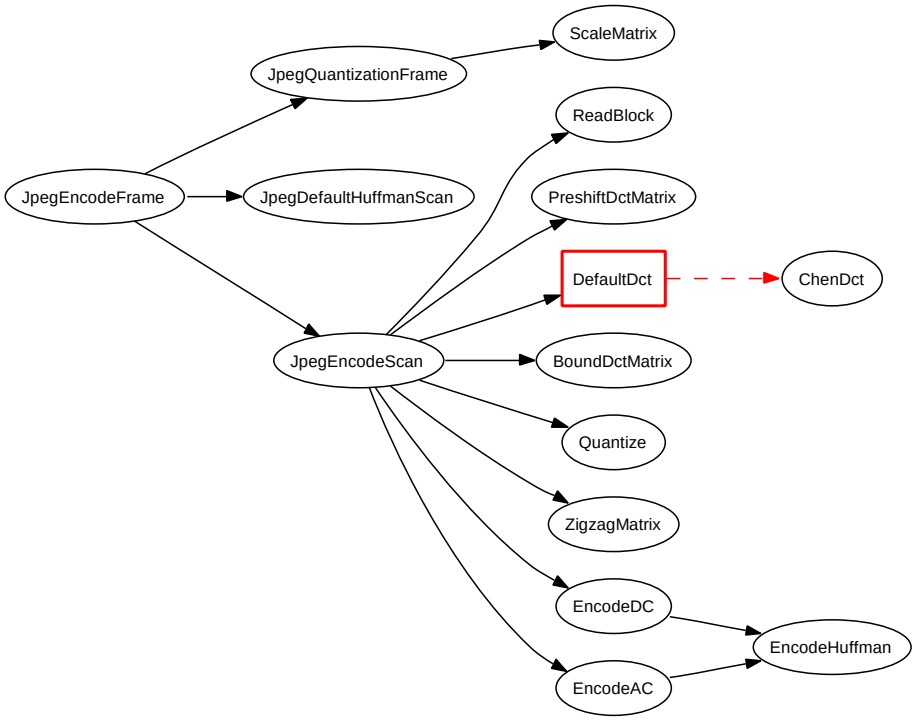

图 14 - pvrg-jpeg 一遍扫描的调用关系

图 14 是 pvrg-jpeg 采用缺省 Huffman 码书压缩时的函数调用关系，注意到它没有色彩变换这一步骤。`JpegEncodeFrame()` 先调用 `JpegDefaultHuffmanScan()`

装载缺省 Huffman 码书，然后调用 JpegEncodeScan() 进行实际编码。

图 15 是采用 “正宗” Huffman 编码时的函数调用关系，JpegFrequencyScan() 的步骤与 JpegEncodeScan() 基本相同，只不过它没有调用 EncodeDC() / EncodeAC() 来进行实际编码，而是调用 FrequencyDC() / FrequencyAC() 以统计各 “中间符号” 出现的次数，JpegEncodeFrame() 然后调用 JpegCustomScan() 根据统计数据生成 Huffman 码书，接着调用 JpegEncodeScan() 进行实际编码。

注意 JpegFrequencyScan 并没有把中间符号记录下来，JpegEncodeScan() 不能像图 15 中那样立即调用 EncodeDC() / EncodeAC() 来进行实际编码，它还得再次调用 ReadBlock()、PreshiftDctMatrix()、DefaultDct()、BoundDctMatrix()、Quantize()、ZigzagMatrix() 以获得中间符号序列（就像图 14 一样）。我认为这是该程序库值得改进的地方之一。

图 15 - pvrg-jpeg 两遍扫描的调用关系

3. MATLAB 实现

使用 MATLAB 能非常方便地实现除熵编码以外的所有 JPEG 编码解码步骤，熵编码（无论 Huffman 编码还是算术编码）更适合用 C 语言实现。我主要参考文献 [10] 第 8.5 节，用 MATLAB 编写了 JPEG 编码器与解码器（不含熵编码）。

4. 参考资料

[1] Gregory K. Wallace. The JPEG Still Picture Compression Standard. *Commun. ACM*, 34(4):30–44, 1991. http://www.ijg.org/files/wallace.ps.gz.

[2] ISO DIS 10918-1: *Digital Compression and Coding of Continuous-Tone Still Images, Part 1: Requirements and Guidelines*, 1991. The same as CCITT Recommendation T.81. http://www.w3.org/Graphics/JPEG/itu-t81.pdf.

[3] Eric Hamilton. *JPEG File Interchange Format, Version 1.02*. http://www.jpeg.org/public/jfif.pdf.

[4] [日] 小野定康，铃木纯司著，叶明译. 《JPEG/MPEG2 技术》. 北京：科学出版社, 2004.

[5] *Compression FAQ*. http://www.faqs.org/faqs/compression-faq/.

[6] 云风. 《JPEG 简易文档V2.14》. http://www.codingnow.com/2000/download/jpeg.txt.

[7] Independent JPEG Group's JPEG software, version 6b. http://www.ijg.org.

[8] Andy C. Hung. PVRG-JPEG CODEC 1.1. http://www.dclunie.com/jpegge.html.

[9] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (2nd ed.)*. Prentice Hall, 2002.

[10] Rafael C. Gonzalez, Richard E. Woods, and Steven L. Eddins. *Digital Image Processing Using MATLAB*. Prentice Hall, 2004.

[11] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. John Wiley & Sons, 1991.

[12] Mark Nelson and Jean-Loup Gailly. *The Data Compression Book (2nd ed.)*. M&T Books, New York, NY, USA, 1996.