《JPEG 解码》调试报告

1 背景知识

1.1 DCT 系数熵编码原理

1.1.1 DC 系数编码：

由于直流系数 F(0,0)反映了该子图像中包含的直流成分，通常较大，又由于两个相邻的子图像的直流系数通常具有较大的相关性，所以对 DC 系数采用差值脉冲编码（DPCM）,即对本像素块直流系数与前一像素块直流系数的差值进行无损编码。

1.1.2 AC 系数编码：

首先，进行游程编码（RLC），并在最后加上块结束码（EOB）；然后，系数序列分组，将非零系数和它前面的相邻的全部零系数分在一组内；每组用两个符号表示[（Run，Size），（Amplitude）]

Amplitude：表示非零系数的幅度值；Run：表示零的游程即零的个数；Size：表示非零系数的幅度值的编码位数；

2 JPEG 的文件组织

2.1 Segment 的组织形式

JPEG 在文件中以 Segment 的形式组织，它具有以下特点：

- 均以 0xFF 开始，后跟 1 byte 的 Marker 和 2 byte 的 Segment length（包含表示 Length 本身所占用的 2 byte，不含“0xFF” + “Marker” 所占用的 2 byte）；
- 采用 Motorola 序（相对于 Intel 序），即保存时高位在前，低位在后；
- Data 部分中，0xFF 后若为 0x00，则跳过此字节不予处理；

2.2 JPEG 的 Segment Marker

Symbol (符号)	Code Assignment (0xFF + Marker) (标记代码)	Description (说明)
Start Of Frame markers, non-hierarchical Huffman coding		
SOF_0	0xFFC0	Baseline DCT
SOF_1	0xFFC1	Extended sequential DCT
SOF_2	0xFFC2	Progressive DCT
SOF_3	0xFFC3	Spatial (sequential) lossless
Start Of Frame markers, hierarchical Huffman coding		
SOF_5	0xFFC5	Differential sequential DCT
SOF_6	0xFFC6	Differential progressive DCT
SOF_7	0xFFC7	Differential spatial lossless

Start Of Frame markers, non-hierarchical arithmetic coding		
JPG	0xFFC8	Reserved for JPEG extensions
SOF_9	0xFFC9	Extended sequential DCT
SOF_{10}	0xFFCA	Progressive DCT
SOF_{11}	0xFFCB	Spatial (sequential) Lossless
Start Of Frame markers, hierarchical arithmetic coding		
SOF_{13}	0xFFCD	Differential sequential DCT
SOF_{14}	0xFFCE	Differential progressive DCT
SOF_{15}	0xFFCF	Differential spatial Lossless
Huffman table specification		
DHT	0xFFC4	Define Huffman table(s)
arithmetic coding conditioning specification		
DAC	0xFFCC	Define arithmetic conditioning table
Restart interval termination		
RSTm	0xFFD0～0xFFD7	Restart with modulo 8 counter m
Other marker		
SOI	0xFFD8	Start of image
EOI	0xFFD9	End of image
SOS	0xFFDA	Start of scan
DQT	0xFFDB	Define quantization table(s)
DNL	0xFFDC	Define number of lines
DRI	0xFFDD	Define restart interval
DHP	0xFFDE	Define hierarchical progression
EXP	0xFFDF	Expand reference image(s)
APP_n	0xFFE0～0xFFEF	Reserved for application use
JPG_n	0xFFF0～0xFFFD	Reserved for JPEG extension
COM	0xFFFE	Comment
Reserved markers		
TEM	0xFF01	For temporary use in arithmetic codi ng
RES	0xFF02～0xFFBF	Reserved

2.3 一个示例

FFD8: SOI，Start of Image，图像开始

所有的 JPEG 文件必须以 SOI 开始

FFE0: APP0，Application，应用程序保留标记 0
length: 16 byte　(2 byte)
00 10 4A 46 49 46 00 01 01 01 00 48 00 48 00 00
标识符: JFIF　　(5 byte)
Version: 0101　　(2 byte)
Units: 01 (1 byte)　X and Y are dots per inch
Xdensity: 72　(2 bytes) Horizontal pixel density(水平方向点密度)
Ydensity: 72　(2 bytes) Vertical pixel density(垂直方向点密度)
缩略图水平像素数目: 00　(1 byte)
缩略图垂直像素数目: 00　(1 byte)
缩略图 24bitRGB 点数目: 缩略图水平像素数目 * 缩略图垂直像素数目 = 00　(1 byte)

FFDB: DQT，Define Quantization Table，定义量化表
length: 67 byte　(2 byte)
00 43 00 08 06 06 07 06 05 08 07 07 07 09 09 08 0A 0C 14 0D 0C 0B 0B 0C 19 12 13 0F 14 1D 1A 1F 1E 1D 1A 1C 1C 20 24 2E 27 20 22 2C 23 1C 1C 28 37 29 2C 30 31 34 34 34 1F 27 39 3D 38 32 3C 2E 33 34 32
QT information - precision: 00　(Higher 4 bit)　　(8 bit)
QT information - index: 00　(Lower 4 bit)
qt_table:
i: 00　value:　　8
i: 01　value:　　6
i: 02　value:　　6
i: 03　value:　　7
i: 04　value:　　6
i: 05　value:　　5
i: 06　value:　　8
i: 07　value:　　7
i: 08　value:　　7
i: 09　value:　　7
i: 10　value:　　9
i: 11　value:　　9
i: 12　value:　　8
i: 13　value:　　10
i: 14　value:　　12
i: 15　value:　　20
i: 16　value:　　13
i: 17　value:　　12
i: 18　value:　　11
i: 19　value:　　11
i: 20　value:　　12

i: 21 value: 25
i: 22 value: 18
i: 23 value: 19
i: 24 value: 15
i: 25 value: 20
i: 26 value: 29
i: 27 value: 26
i: 28 value: 31
i: 29 value: 30
i: 30 value: 29
i: 31 value: 26
i: 32 value: 28
i: 33 value: 28
i: 34 value: 32
i: 35 value: 36
i: 36 value: 46
i: 37 value: 39
i: 38 value: 32
i: 39 value: 34
i: 40 value: 44
i: 41 value: 35
i: 42 value: 28
i: 43 value: 28
i: 44 value: 40
i: 45 value: 55
i: 46 value: 41
i: 47 value: 44
i: 48 value: 48
i: 49 value: 49
i: 50 value: 52
i: 51 value: 52
i: 52 value: 52
i: 53 value: 31
i: 54 value: 39
i: 55 value: 57
i: 56 value: 61
i: 57 value: 56
i: 58 value: 50
i: 59 value: 60
i: 60 value: 46
i: 61 value: 51
i: 62 value: 52
i: 63 value: 50

- **量化表标记代码：FFDB**

- **量化表长度：（17byte），长度本身占 2 字节，**
- **精度及量化表 ID：**

1byte，低位为量化表的 ID 的索引值，其取值范围为 0～3，所以说最多可能有四张量化表（亮度量化表，色度量化表，可能会有 R、G、B 各一张量化表）；而低位为量化精度，有两个可选值，0：8 位；1：16 位，这里是 00，代表量化精度为 8bit,即用一个字节来表示一个量化系数

- **数据：64 字节的量化系数，（一个字节对应一个量化系数，对于 8*8 的宏块来说，共 8*8=64 个量化系数）**

FFDB: DQT，Define Quantization Table，定义量化表

length: 67 byte (2 byte)

00 43 01 09 09 09 0C 0B 0C 18 0D 0D 18 32 21 1C 21 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32

QT information - precision: 00 (Higher 4 bit) (8 bit)

QT information - index: 01 (Lower 4 bit)

qt_table:

i: 00 value: 9
i: 01 value: 9
i: 02 value: 9
i: 03 value: 12
i: 04 value: 11
i: 05 value: 12
i: 06 value: 24
i: 07 value: 13
i: 08 value: 13
i: 09 value: 24
i: 10 value: 50
i: 11 value: 33
i: 12 value: 28
i: 13 value: 33
i: 14 value: 50
i: 15 value: 50
i: 16 value: 50
i: 17 value: 50
i: 18 value: 50
i: 19 value: 50
i: 20 value: 50
i: 21 value: 50
i: 22 value: 50
i: 23 value: 50
i: 24 value: 50
i: 25 value: 50
i: 26 value: 50

i: 27 value: 50
i: 28 value: 50
i: 29 value: 50
i: 30 value: 50
i: 31 value: 50
i: 32 value: 50
i: 33 value: 50
i: 34 value: 50
i: 35 value: 50
i: 36 value: 50
i: 37 value: 50
i: 38 value: 50
i: 39 value: 50
i: 40 value: 50
i: 41 value: 50
i: 42 value: 50
i: 43 value: 50
i: 44 value: 50
i: 45 value: 50
i: 46 value: 50
i: 47 value: 50
i: 48 value: 50
i: 49 value: 50
i: 50 value: 50
i: 51 value: 50
i: 52 value: 50
i: 53 value: 50
i: 54 value: 50
i: 55 value: 50
i: 56 value: 50
i: 57 value: 50
i: 58 value: 50
i: 59 value: 50
i: 60 value: 50
i: 61 value: 50
i: 62 value: 50
i: 63 value: 50

第二张量化表，不同的是上张表的索引号为 00，这张表的索引号为 01，在后面的 SOF0 的部分中我们将会知道上张表对应亮度量化表，这张表对应色度量化表，对这张图来说就这两张量化表。

FFC0: SOF0 ，Start of Frame， 基线离散余弦变换
length: 17 byte (2 byte)

00 11 08 01 C2 01 3B 03 01 11 00 02 11 01 03 11 01
图像精度（每个数据样本的位数）: 8
Image Height: 450　(2 byte)
Image Width: 315　(2 byte)
颜色分量数: 03（YCrCb）　(1 byte)

颜色分量 ID: 01　(1 byte) (Y)
SampRate_Y_H: 01　(Higher 4 bit)
SampRate_Y_V: 01　(Lower 4 bit)
Y QtTableID: 00　(1 byte)

颜色分量 ID: 02　(1 byte) (U)
SampRate_U_H: 01　(Higher 4 bit)
SampRate_U_V: 01　(Lower 4 bit)
U QtTableID: 01　(1 byte)

颜色分量 ID: 03　(1 byte) (V)
SampRate_V_H: 01　(Higher 4 bit)
SampRate_V_V: 01　(Lower 4 bit)
V QtTableID: 01　(1 byte)

- **SOF0：　start of frame,帧图像开始**
- **数据长度：长度本身占 2byte**
- **图像精度: 1byte，这里是 08，即精度为一个字节。**
- **图像高度：2 byte，以像素为单位。**
- **图像宽度：2 byte，以像素为单位。**
- **颜色分量数：一个字节，这里是 03，代表有三个分量，YCrCb。**
- **颜色分量信息：每个分量有三个字节，第一个为分量的 ID，01：Y 02：U 03：V；第二个字节高位为水平采样因子，低位为垂直采样因子，这里三个分量的采样率相同，所以采样格式为 4：4：4；第三个字节代表这个分量对应的量化表 ID，可以看出，Y 对应的量化表 ID 索引值为 00,而 UV 对应的量化表 ID 都为 01，即它们共用一张量化表。**
- **所以，除了标记外，总共的长度就为 2+1+5+3*3=17**

FFC4: DHT，Define Huffman Table，定义 Huffman 树表
00 1C 00 00 02 03 01 01 01 01 00 00 00 00 00 00 00 00 03 04 01 02 05 06 00 07 08
length: 28 byte　(2 byte)
Huffman 表类型: 0　(Higher 4 bit)　(DC)
Huffman 表 ID: 0　(Lower 4 bit)　(0 号表)
HuffmanTableIndex: 0
code_len_table: (16 byte)
CodeLength: 01　00 个
CodeLength: 02　02 个

CodeLength: 03　　03 个
CodeLength: 04　　01 个
CodeLength: 05　　01 个
CodeLength: 06　　01 个
CodeLength: 07　　01 个
CodeLength: 08　　00 个
CodeLength: 09　　00 个
CodeLength: 10　　00 个
CodeLength: 11　　00 个
CodeLength: 12　　00 个
CodeLength: 13　　00 个
CodeLength: 14　　00 个
CodeLength: 15　　00 个
CodeLength: 16　　00 个

- **FFC4：标记代码，2 字节，代表定义 Huffman 表。**
- **数据长度：　2 字节　这里是 28 字节的长度(包括长度自身)**
- **Huffman 表 ID 号和类型：1 字节，高 4 位为表的类型，0：DC 直流；1：AC 交流　可以看出这里是直流表；低四位为 Huffman 表 ID。**
- **可以看出这张表是直流 DC 的第 0 张表，在后面的扫描开始的部分中我们可以获右为亮度的直流系数表。**
- **不同长度 Huffman 的码字数量：固定为 16 个字节，每个字节代表从长度为 1 到长度为　16　的码字的个数，以表中的分析，这　16　个字节之后的 2+3+1+1+1+1=9　个字节对应的就是每个符字对应的权值，这些权值的含义即为 DC 系数经 DPCM 编码后幅度值的位长。**
- **通过上面的码长与码字个数的关系来生成相应码长的码字，再对应上之后的权值即位长**
- **根据解码得到的位长来读取之后相应长度的码字，再查上面这张可变长二进制编码表，就可以得到直流系数的幅度值，注意这个幅度值是经过 DPCM 差分编码得到的。**
- **这张图片中共有四张 Huffman 表**

FFC4: DHT，Define Huffman Table，定义 Huffman 树表
00 45 10 00 01 03 02 04 03 05 05 05 06 05 02 06 03 00 00 01 00 02 03 04 11 05 12 21 31 41 51 61 06 13 22 71 81 32 91 A1 B1 C1 14 23 42 52 D1 07 15 72 E1 F0 F1 33 43 53 62 82 16 92 24 34 73 A2 B2 C2 63 74 83
length: 69 byte　(2 byte)
Huffman 表类型: 1　(Higher 4 bit)　　(AC)
Huffman 表 ID: 0　(Lower 4 bit)　(0 号表)
HuffmanTableIndex: 2
code_len_table: (16 byte)
CodeLength: 01　　00 个
CodeLength: 02　　01 个

CodeLength: 03　　03 个
CodeLength: 04　　02 个
CodeLength: 05　　04 个
CodeLength: 06　　03 个
CodeLength: 07　　05 个
CodeLength: 08　　05 个
CodeLength: 09　　05 个
CodeLength: 10　　06 个
CodeLength: 11　　05 个
CodeLength: 12　　02 个
CodeLength: 13　　06 个
CodeLength: 14　　03 个
CodeLength: 15　　00 个
CodeLength: 16　　00 个

交流系数表:
它和直流系数表的权值代表的含义与解码方式有一定的差别，交流系数权值的高位代表游程 run 的值，低位与直流系数相同，代表幅度的位长 size. 在得到位长后还是要查可变长二制编码表来得到真正的 AC 幅值。

FFC4: DHT，Define Huffman Table，定义 Huffman 树表
00 19 01 00 03 01 01 01 00 00 00 00 00 00 00 00 00 00 00 01 02 03 04 05
length: 25 byte　(2 byte)
Huffman 表类型: 0　(Higher 4 bit)　　(DC)
Huffman 表 ID: 1　(Lower 4 bit)　(1 号表)
HuffmanTableIndex: 1
code_len_table: (16 byte)
CodeLength: 01　　00 个
CodeLength: 02　　03 个
CodeLength: 03　　01 个
CodeLength: 04　　01 个
CodeLength: 05　　01 个
CodeLength: 06　　00 个
CodeLength: 07　　00 个
CodeLength: 08　　00 个
CodeLength: 09　　00 个
CodeLength: 10　　00 个
CodeLength: 11　　00 个
CodeLength: 12　　00 个
CodeLength: 13　　00 个
CodeLength: 14　　00 个
CodeLength: 15　　00 个
CodeLength: 16　　00 个

FFC4: DHT，Define Huffman Table，定义 Huffman 树表
00 2A 11 01 01 00 02 02 02 02 02 03 00 02 02 03 01 00 00 00 01 02 11 03 21 12 31 41 51 04 22 13 32 61 42 71 33 52 05 14 23 A1
length: 42 byte　(2 byte)
Huffman 表类型: 1　(Higher 4 bit)　(AC)
Huffman 表 ID: 1　(Lower 4 bit)　(1 号表)
HuffmanTableIndex: 3
code_len_table: (16 byte)
CodeLength: 01　01 个
CodeLength: 02　01 个
CodeLength: 03　00 个
CodeLength: 04　02 个
CodeLength: 05　02 个
CodeLength: 06　02 个
CodeLength: 07　02 个
CodeLength: 08　02 个
CodeLength: 09　03 个
CodeLength: 10　00 个
CodeLength: 11　02 个
CodeLength: 12　02 个
CodeLength: 13　03 个
CodeLength: 14　01 个
CodeLength: 15　00 个
CodeLength: 16　00 个

FFDA:　SOS，Start of Scan，扫描开始
length: 12 byte　(2 byte)
00 0C 03 01 00 02 11 03 11 00 3F 00
颜色分量 ID: 1　(1 byte) (Y)
Y Dc HuffmanTreeIndex: 0　(Higher 4 bit)
Y Ac HuffmanTreeIndex: 2　(Lower 4 bit)

颜色分量 ID: 2　(1 byte) (U or V)
UV Dc HuffmanTreeIndex: 1　(Higher 4 bit)
UV Ac HuffmanTreeIndex: 3　(Lower 4 bit)

颜色分量 ID: 3　(1 byte) (U or V)
UV Dc HuffmanTreeIndex: 1　(Higher 4 bit)
UV Ac HuffmanTreeIndex: 3　(Lower 4 bit)

- **FFDA：标记代码 SOS，Start of Scan，扫描开始**

- 数据长度：2 字节，这里是长度为 12 字节。
- 颜色分量数：1 字节　应该和 SOF 中的颜色分量数相同

- 颜色分量信息：每个分量对应 3 个字节，第一个字节是颜色分量 ID，1，2，3 对应 YUV，第二个字节高位为直流分量使用的哈夫曼树编号，这里 Y 的直流分量用的是 DC 的第 0 张表，低四位代表交流分量使用的哈夫曼树编号，这里 Y 的交流分量用的是 AC 的第 0 张表，而两个色度信号的直流分量都用的是 DC 的第 1 张表，交流分量用的是 AC 的第 1 张表.
- 压缩图像数据

  a)谱选择开始　　1 字节　固定值 0x00

  b)谱选择结束　　1 字节　固定值 0x3F

  c)谱选择　　1 字节　在基本 JPEG 中总为 00

[熵编码数据]

图片的熵编码数据

FFD9: End of Image，图像结束

结束符，JPEG 文件必须以 EOI 结束

3 JPEG 的解码流程

3.1 读取文件

3.2 解析 Segment Marker

3.2.1 解析 SOI

3.2.2 解析 APP0

- 检查标识“JFIF”及版本
- 得到一些参数

3.2.3 解析 DQT

- 得到量化表长度（可能包含多张量化表）
- 得到量化表的精度
- 得到及检查量化表的序号（只能是 0 —— 3）
- 得到量化表内容（64 个数据）

3.2.4 解析 SOF0

- 得到每个 sample 的比特数、长宽、颜色分量数
- 得到每个颜色分量的 ID、水平采样因子、垂直采样因子、使用的量化表序号（与 DQT 中序号对应）

3.2.5 解析 DHT

- 得到 Huffman 表的类型（AC、DC）、序号
- 依据数据重建 Huffman 表

3.2.6 解析 SOS

- 得到解析每个颜色分量的 DC、AC 值所使用的 Huffman 表序号（与 DHT 中序号对应）

3.3 依据每个分量的水平、垂直采样因子计算 MCU 的大小，并得到每个 MCU 中 8*8 宏块的个数

3.4 对每个 MCU 解码（依照各分量水平、垂直采样因子对 MCU 中每个分量宏块解码）

3.4.1 对每个宏块进行 Huffman 解码，得到 DCT 系数

3.4.2 对每个宏块的 DCT 系数进行 IDCT，得到 Y、Cb、Cr

3.4.3 遇到 Segment Marker RST 时，清空之前的 DC DCT 系数

3.5 解析到 EOI，解码结束

3.6 将 Y、Cb、Cr 转化为需要的色彩空间并保存。

4 程序实现

4.1 读取文件

4.2 解析 Segment Marker

```
while (!sos_marker_found)
    {
      if (*stream++ != 0xff)
        goto bogus_jpeg_format;
      /* Skip any padding ff byte (this is normal) */
      while (*stream == 0xff)
        stream++;

      marker = *stream++;
      chuck_len = be16_to_cpu(stream);
      next_chunck = stream + chuck_len;
      switch (marker)
       {
        case SOF:
      if (parse_SOF(priv, stream) < 0)
        return -1;
      break;
        case DQT:
      if (parse_DQT(priv, stream) < 0)
        return -1;
      break;
        case SOS:
      if (parse_SOS(priv, stream) < 0)
        return -1;
      sos_marker_found = 1;
      break;
        case DHT:
      if (parse_DHT(priv, stream) < 0)
        return -1;
      dht_marker_found = 1;
      break;
        case DRI:
      if (parse_DRI(priv, stream) < 0)
```

```
            return -1;
        break;
            default:
#if TRACE
        fprintf(p_trace,"> Unknown marker %2.2x\n", marker);
        fflush(p_trace);
#endif
        break;
         }

        stream = next_chunck;
    }
```

4.2.1 解析 SOI

```
if ((buf[0] != 0xFF) || (buf[1] != SOI))   //JPEG 文件必须以 0xFFD8（SOI）起始
        snprintf(error_string, sizeof(error_string),"Not a JPG file ?\n");
```

4.2.2 解析 APP0

- 检查标识“JFIF”及版本
- 得到一些参数

4.2.3 解析 DQT

- 得到量化表长度（可能包含多张量化表）
- 得到量化表的精度
- 得到及检查量化表的序号（只能是 0 —— 3）
- 得到量化表内容（64 个数据）

```
dqt_block_end = stream + be16_to_cpu(stream); // 得到量化表长度（可能包含多张量化表）
    stream += 2;     /* Skip length */

    while (stream < dqt_block_end) // 检查是否还有表
     {
        qi = *stream++;
        if (qi>>4)
            snprintf(error_string, sizeof(error_string),"16 bits quantization table is not supported\n"); // 得到量化表的精度（高四位）
        if (qi>4)
            snprintf(error_string, sizeof(error_string),"No more 4 quantization table is supported (got %d)\n", qi); // 得到量化表序号（低四位）
        table = priv->Q_tables[qi];
        build_quantization_table(table, stream); // 得到量化表内容
        stream += 64;
     }
```

```
    static const unsigned char zigzag[64] =
    { // zig-zag 排序
```

```
    0,  1,  5,  6, 14, 15, 27, 28,
    2,  4,  7, 13, 16, 26, 29, 42,
    3,  8, 12, 17, 25, 30, 41, 43,
    9, 11, 18, 24, 31, 40, 44, 53,
   10, 19, 23, 32, 39, 45, 52, 54,
   20, 22, 33, 38, 46, 51, 55, 60,
   21, 34, 37, 47, 50, 56, 59, 61,
   35, 36, 48, 49, 57, 58, 62, 63
};
```

```
static void build_quantization_table(float *qtable, const unsigned char *ref_table)
{
  /* Taken from libjpeg. Copyright Independent JPEG Group's LLM idct.
   * For float AA&N IDCT method, divisors are equal to quantization
   * coefficients scaled by scalefactor[row]*scalefactor[col], where
   *    scalefactor[0] = 1
   *    scalefactor[k] = cos(k*PI/16) * sqrt(2)    for k=1..7
   * We apply a further scale factor of 8.
   * What's actually stored is 1/divisor so that the inner loop can
   * use a multiplication rather than a division.
   */
  int i, j;
  static const double aanscalefactor[8] = {
      1.0, 1.387039845, 1.306562965, 1.175875602,
      1.0, 0.785694958, 0.541196100, 0.275899379
  };
  const unsigned char *zz = zigzag;

  for (i=0; i<8; i++) {
      for (j=0; j<8; j++) {
        *qtable++ = ref_table[*zz++] * aanscalefactor[i] * aanscalefactor[j]; // 以 zig-zag 序存储
      }
   }

}
```

4.2.4 解析 SOF0

- 得到每个 sample 的比特数、长宽、颜色分量数
- 得到每个颜色分量的 ID、水平采样因子、垂直采样因子、使用的量化表序号（与 DQT 中序号对应）

```
height = be16_to_cpu(stream+3);   // 得到当前图片的高度
width  = be16_to_cpu(stream+5); // 得到当前图片的宽度
nr_components = stream[7]; // 得到颜色分量数（Y、Cb、Cr，共计 3）
```

```
stream += 8;
for (i=0; i<nr_components; i++) { // 得到每个分量的 ID 等
    cid = *stream++;   // 该分量的 ID
    sampling_factor = *stream++; // 该分量的水平、垂直采样因子
    Q_table = *stream++;   // 该分量使用的量化表序号
    c = &priv->component_infos[i];
    c->cid = cid;
    if (Q_table >= COMPONENTS)
        snprintf(error_string,  sizeof(error_string),"Bad  Quantization  table
index (got %d, max allowed %d)\n", Q_table, COMPONENTS-1);
    c->Vfactor = sampling_factor&0xf;   // 垂直采样因子（低四位）
    c->Hfactor = sampling_factor>>4;   // 水平采样因子（高四位）
    c->Q_table = priv->Q_tables[Q_table];
}
priv->width = width;
priv->height = height;
```

4.2.5 解析 DHT

- 得到 Huffman 表的类型（AC、DC）、序号
- 依据数据重建 Huffman 表

```
length = be16_to_cpu(stream) - 2;   // 表长(可能包含多张表)
stream += 2;    /* Skip length */
while (length>0) {   // 是否还有表
    index = *stream++;

    /* We need to calculate the number of bytes 'vals' will takes */
    huff_bits[0] = 0;
    count = 0;
    for (i=1; i<17; i++) {
   huff_bits[i] = *stream++;
   count += huff_bits[i];
    }
    if (index & 0xf0 ) // AC 表
                        // (index&0xf)，  Huffman 表序号
        build_huffman_table(huff_bits, stream, &priv->HTAC[index&0xf]);
    else   // DC 表
        build_huffman_table(huff_bits, stream, &priv->HTDC[index&0xf]);

    length -= 1;
    length -= 16;
    length -= count;
    stream += count;
}
```

4.2.6 解析 SOS

- 得到解析每个颜色分量的 DC、AC 值所使用的 Huffman 表序号（与 DHT

中序号对应）

```
    unsigned int nr_components = stream[2]; // 得到颜色分量数
    stream += 3;
    for (i=0;i<nr_components;i++) {    // 对所有分量
        cid = *stream++; // 得到 ID
        table = *stream++; // Huffman 表序号，高四位：DC 低四位：AC
        priv->component_infos[i].AC_table = &priv->HTAC[table&0xf];
        priv->component_infos[i].DC_table = &priv->HTDC[table>>4];
    }
    priv->stream = stream+3;
```

4.3 依据每个分量的水平、垂直采样因子计算 MCU 的大小，并得到每个 MCU 中 8*8 宏块的个数

```
xstride_by_mcu = ystride_by_mcu = 8; // 初始化为 4:4:4 时的情况
    if ((priv->component_infos[cY].Hfactor | priv->component_infos[cY].Vfactor)
== 1) {
        decode_MCU = decode_mcu_table[0]; // 4:4:4
        convert_to_pixfmt = colorspace_array_conv[0];
#if TRACE
        fprintf(p_trace,"Use decode 1x1 sampling\n");
    fflush(p_trace);
#endif
    } else if (priv->component_infos[cY].Hfactor == 1) {
        decode_MCU = decode_mcu_table[1];
        convert_to_pixfmt = colorspace_array_conv[1];
        ystride_by_mcu = 16;    // 一个 MCU 的高为 16
#if TRACE
        fprintf(p_trace,"Use decode 1x2 sampling (not supported)\n");
    fflush(p_trace);
#endif
    } else if (priv->component_infos[cY].Vfactor == 2) {
        decode_MCU = decode_mcu_table[3];
        convert_to_pixfmt = colorspace_array_conv[3];
        xstride_by_mcu = 16;    // 一个 MCU 的宽为 16
        ystride_by_mcu = 16;    // 一个 MCU 的高为 16
#if TRACE
    fprintf(p_trace,"Use decode 2x2 sampling\n");
    fflush(p_trace);
#endif
    } else {
        decode_MCU = decode_mcu_table[2];
        convert_to_pixfmt = colorspace_array_conv[2];
        xstride_by_mcu = 16;    // 一个 MCU 的宽为 16
#if TRACE
        fprintf(p_trace,"Use decode 2x1 sampling\n");
```

```
    fflush(p_trace);
#endif
  }
```

4.4 对每个 MCU 解码（依照各分量水平、垂直采样因子对 MCU 中每个分量宏块解码）

4.4.1 对每个宏块进行 Huffman 解码，得到 DCT 系数

4.4.2 对每个宏块的 DCT 系数进行 IDCT，得到 Y、Cb、Cr

4.4.3 遇到 Segment Marker RST 时，清空之前的 DC DCT 系数

```
/*
 * Decode all the 3 components for 1x1
 */
static void decode_MCU_1x1_3planes(struct jdec_private *priv) // 4:4:4
{
  // Y
  process_Huffman_data_unit(priv, cY); // 以 8*8 宏块为单位进行 Huffman 解码
  IDCT(&priv->component_infos[cY], priv->Y, 8); // 对得到的 DCT 系数进行 IDCT

  // Cb
  process_Huffman_data_unit(priv, cCb);
  IDCT(&priv->component_infos[cCb], priv->Cb, 8);

  // Cr
  process_Huffman_data_unit(priv, cCr);
  IDCT(&priv->component_infos[cCr], priv->Cr, 8);
}


/*
 * Decode a 2x1
 *   .-------.
 *   | 1 | 2 |
 *   `-------'
 */
static void decode_MCU_2x1_3planes(struct jdec_private *priv) // 4:2:2
{
  // Y
  process_Huffman_data_unit(priv, cY);
  IDCT(&priv->component_infos[cY], priv->Y, 16);
  process_Huffman_data_unit(priv, cY);
  IDCT(&priv->component_infos[cY], priv->Y+8, 16);

  // Cb
  process_Huffman_data_unit(priv, cCb);
```

```
    IDCT(&priv->component_infos[cCb], priv->Cb, 8);

    // Cr
    process_Huffman_data_unit(priv, cCr);
    IDCT(&priv->component_infos[cCr], priv->Cr, 8);
}


/*
 * Decode a 2x2
 *    .-------.
 *    | 1 | 2 |
 *    |---+---|
 *    | 3 | 4 |
 *    `-------'
 */
static void decode_MCU_2x2_3planes(struct jdec_private *priv) // 4:2:0
{
    // Y
    process_Huffman_data_unit(priv, cY);
    IDCT(&priv->component_infos[cY], priv->Y, 16);
    process_Huffman_data_unit(priv, cY);
    IDCT(&priv->component_infos[cY], priv->Y+8, 16);
    process_Huffman_data_unit(priv, cY);
    IDCT(&priv->component_infos[cY], priv->Y+64*2, 16);
    process_Huffman_data_unit(priv, cY);
    IDCT(&priv->component_infos[cY], priv->Y+64*2+8, 16);

    // Cb
    process_Huffman_data_unit(priv, cCb);
    IDCT(&priv->component_infos[cCb], priv->Cb, 8);

    // Cr
    process_Huffman_data_unit(priv, cCr);
    IDCT(&priv->component_infos[cCr], priv->Cr, 8);
}
```

```
static void process_Huffman_data_unit(struct jdec_private *priv, int component)
{      // 以 8*8 宏块为单位进行 Huffman 解码
    unsigned char j;
    unsigned int huff_code;
    unsigned char size_val, count_0;

    struct component *c = &priv->component_infos[component];
```

```
  short int DCT[64];


  /* Initialize the DCT coef table */
  memset(DCT, 0, sizeof(DCT));

  /* DC coefficient decoding */
  huff_code = get_next_huffman_code(priv, c->DC_table);
  if (huff_code) {
      get_nbits(priv->reservoir, priv->nbits_in_reservoir, priv->stream, huff_code,
DCT[0]); // 查表的 DC DCT 系数（残值）
      DCT[0] += c->previous_DC; // DC 系数采用差分编码，恢复原值
      c->previous_DC = DCT[0];
  } else {
      DCT[0] = c->previous_DC;
  }

  /* AC coefficient decoding */
  j = 1;
  while (j<64)
   {
      huff_code = get_next_huffman_code(priv, c->AC_table);

      size_val = huff_code & 0xF; // Amplitude 幅度
      count_0 = huff_code >> 4;   // 零游程长度

      if (size_val == 0)   // 0 不是一个有效的 Amplitude 值，这里做零游程标志
       { /* 零游程 */
          if (count_0 == 0)
                break;    /* EOB found, go out */
          else if (count_0 == 0xF)
             j += 16;     /* skip 16 zeros */
       }
      else
       {
          j += count_0; /* 忽略零游程 */
          if (__unlikely(j >= 64))   // 出错了
                {
                   snprintf(error_string, sizeof(error_string), "Bad huffman data
(buffer overflow)");
                   break;
            }
          get_nbits(priv->reservoir,      priv->nbits_in_reservoir,      priv->stream,
size_val, DCT[j]);   // 查表得到 AC DCT 系数
```

```
            j++;
          }
      }

    for (j = 0; j < 64; j++)
       c->DCT[j] = DCT[zigzag[j]]; //  以 zig-zag 序保存
}
```

4.5　解完所有 MCU，解码结束

```
for (y=0; y < priv->height/ystride_by_mcu; y++)   //  行循环
    {
       //trace("Decoding row %d\n", y);
       priv->plane[0] = priv->components[0] + (y * bytes_per_blocklines[0]);
       priv->plane[1] = priv->components[1] + (y * bytes_per_blocklines[1]);
       priv->plane[2] = priv->components[2] + (y * bytes_per_blocklines[2]);
       for (x=0; x < priv->width; x+=xstride_by_mcu) //  列循环
        {
      decode_MCU(priv); //  解码（Huffman 解码  + IDCT）
      convert_to_pixfmt(priv);
      priv->plane[0] += bytes_per_mcu[0];
      priv->plane[1] += bytes_per_mcu[1];
      priv->plane[2] += bytes_per_mcu[2];
      if (priv->restarts_to_go>0)
       {
          priv->restarts_to_go--;
          if (priv->restarts_to_go == 0)
           {
              priv->stream -= (priv->nbits_in_reservoir/8);
              resync(priv); //  清空 preDC（所有颜色分量）
              if (find_next_rst_marker(priv) < 0) //  查找 RST 标记
            return -1;
            }
       }
        }
    }
```

4.6　将 Y、Cb、Cr 转化为需要的色彩空间并保存。