

原 x264源代码简单分析：熵编码（Entropy Encoding）部分

2015年05月24日 22:48:12 阅读数：7862

=====

H.264源代码分析文章列表：

【编码 - x264】

[x264源代码简单分析：概述](#)

[x264源代码简单分析：x264命令行工具（x264.exe）](#)

[x264源代码简单分析：编码器主干部分-1](#)

[x264源代码简单分析：编码器主干部分-2](#)

[x264源代码简单分析：x264_slice_write\(\)](#)

[x264源代码简单分析：滤波（Filter）部分](#)

[x264源代码简单分析：宏块分析（Analysis）部分-帧内宏块（Intra）](#)

[x264源代码简单分析：宏块分析（Analysis）部分-帧间宏块（Inter）](#)

[x264源代码简单分析：宏块编码（Encode）部分](#)

[x264源代码简单分析：熵编码（Entropy Encoding）部分](#)

[FFmpeg与libx264接口源代码简单分析](#)

【解码 - libavcodec H.264 解码器】

[FFmpeg的H.264解码器源代码简单分析：概述](#)

[FFmpeg的H.264解码器源代码简单分析：解析器（Parser）部分](#)

[FFmpeg的H.264解码器源代码简单分析：解码器主干部分](#)

[FFmpeg的H.264解码器源代码简单分析：熵解码（EntropyDecoding）部分](#)

[FFmpeg的H.264解码器源代码简单分析：宏块解码（Decode）部分-帧内宏块（Intra）](#)

[FFmpeg的H.264解码器源代码简单分析：宏块解码（Decode）部分-帧间宏块（Inter）](#)

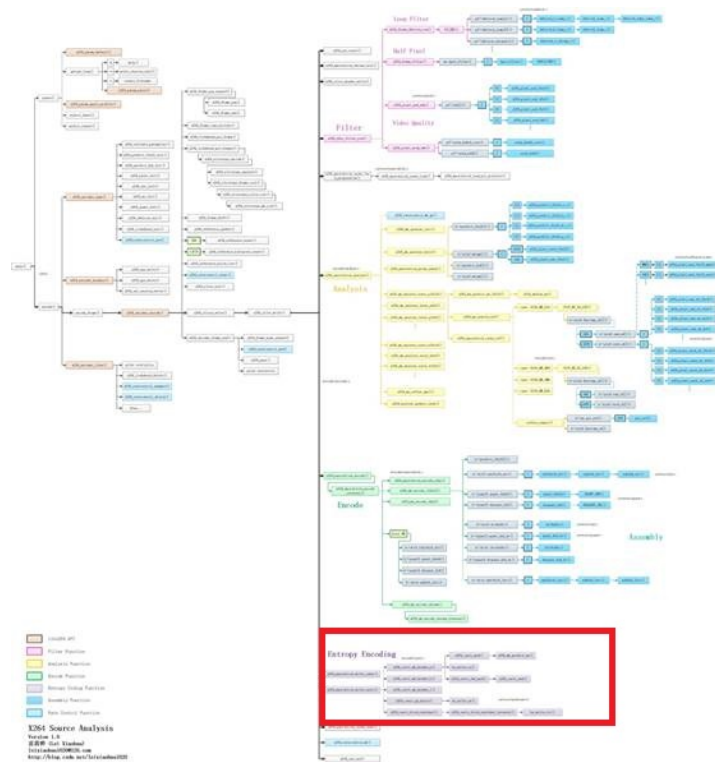
[FFmpeg的H.264解码器源代码简单分析：环路滤波（Loop Filter）部分](#)

=====

本文记录x264的 x264_slice_write()函数中调用的x264_macroblock_write_cavlc()的源代码。x264_macroblock_write_cavlc()对应着x264中的熵编码模块。熵编码模块主要完成了编码数据输出的功能。

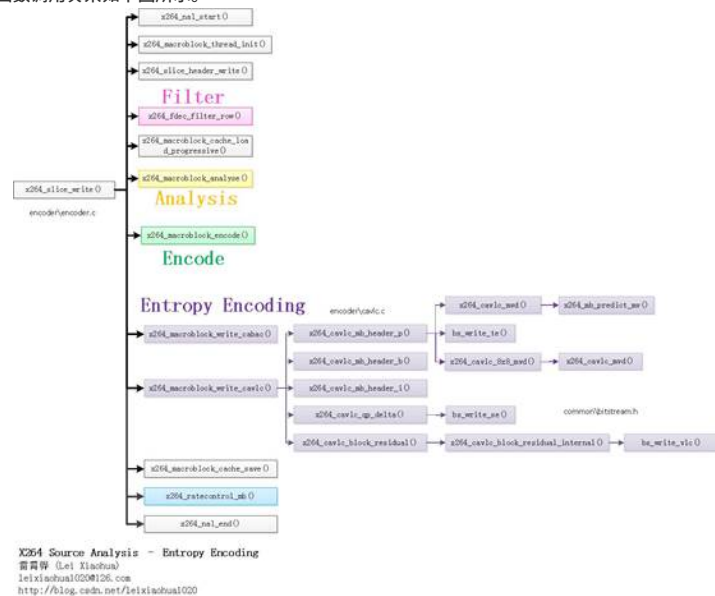
函数调用关系图

熵编码（Entropy Encoding）部分的源代码在整个x264中的位置如下图所示。



[单击查看更清晰的图片](#)

熵编码（Entropy Encoding）部分的函数调用关系如下图所示。



[单击查看更清晰的图片](#)

从图中可以看出，熵编码模块包含两个函数x264_macroblock_write_cabac()和x264_macroblock_write_cavlc()。如果输出设置为CABAC编码，则会调用x264_macroblock_write_cabac()；如果输出设置为CAVLC编码，则会调用x264_macroblock_write_cavlc()。本文选择CAVLC编码输出函数x264_macroblock_write_cavlc()进行分析。该函数调用了如下函数：

- x264_cavlc_mb_header_i(): 写入I宏块MB Header数据。包含帧内预测模式等。
- x264_cavlc_mb_header_p(): 写入P宏块MB Header数据。包含MVD、参考帧序号等。
- x264_cavlc_mb_header_b(): 写入B宏块MB Header数据。包含MVD、参考帧序号等。
- x264_cavlc_qp_delta(): 写入QP。
- x264_cavlc_block_residual(): 写入残差数据。

x264_slice_write()

x264_slice_write()是x264项目的核心，它完成了编码了一个Slice的工作。有关该函数的分析可以参考文章《x264源代码简单分析：x264_slice_write()》。本文分析其调用的x264_macroblock_write_cavlc()函数。

x264_macroblock_write_cavlc()

x264_macroblock_write_cavlc()用于以CAVLC编码的方式输出H.264码流。该函数的定义位于encoder/cavlc.c，如下所示。

```
[cpp]
1.  /*****
2.  * x264_macroblock_write:
3.  *
4.  * 注释和处理：雷霄骅
5.  * http://blog.csdn.net/leixiaohua1020
6.  * leixiaohua1020@126.com
7.  *****/
8.  void x264_macroblock_write_cavlc( x264_t *h )
9.  {
10.     bs_t *s = &h->out.bs;
11.     const int i_mb_type = h->mb.i_type;
12.     int plane_count = CHROMA444 ? 3 : 1;
13.     int chroma = !CHROMA444;
14.
15.     #if RDO_SKIP_BS
16.         s->i_bits_encoded = 0;
17.     #else
18.         const int i_mb_pos_start = bs_pos( s );
19.         int i_mb_pos_tex;
20.     #endif
21.
22.     if( SLICE_MBAFF
23.         && !(h->mb.i_mb_y & 1) || IS_SKIP(h->mb.type[h->mb.i_mb_xy - h->mb.i_mb_stride])) )
24.     {
25.         bs_writel( s, MB_INTERLACED );
26.     #if !RDO_SKIP_BS
27.         h->mb.field_decoding_flag = MB_INTERLACED;
28.     #endif
29.     }
30.
31.     #if !RDO_SKIP_BS
32.         if( i_mb_type == I_PCM )
33.         {
34.             static const uint8_t i_offsets[3] = {5,23,0};
35.             uint8_t *p_start = s->p_start;
36.             bs_write_ue( s, i_offsets[h->sh.i_type] + 25 );
37.             i_mb_pos_tex = bs_pos( s );
38.             h->stat.frame.i_mv_bits += i_mb_pos_tex - i_mb_pos_start;
39.
40.             bs_align_0( s );
41.
42.             for( int p = 0; p < plane_count; p++ )
43.                 for( int i = 0; i < 256; i++ )
44.                     bs_write( s, BIT_DEPTH, h->mb.pic.p_fenc[p][i] );
45.             if( chroma )
46.                 for( int ch = 1; ch < 3; ch++ )
47.                     for( int i = 0; i < 16>>CHROMA_V_SHIFT; i++ )
48.                         for( int j = 0; j < 8; j++ )
49.                             bs_write( s, BIT_DEPTH, h->mb.pic.p_fenc[ch][i*FENC_STRIDE+j] );
50.
51.             bs_init( s, s->p, s->p_end - s->p );
52.             s->p_start = p_start;
53.
54.             h->stat.frame.i_tex_bits += bs_pos(s) - i_mb_pos_tex;
55.             return;
56.         }
57.     #endif
58.
59.
60.     if( h->sh.i_type == SLICE_TYPE_P )
61.         x264_cavlc_mb_header_p( h, i_mb_type, chroma );//写入P宏块MB Header数据-CAVLC
62.     else if( h->sh.i_type == SLICE_TYPE_B )
63.         x264_cavlc_mb_header_b( h, i_mb_type, chroma );//写入B宏块MB Header数据-CAVLC
64.     else //if( h->sh.i_type == SLICE_TYPE_I )
65.         x264_cavlc_mb_header_i( h, i_mb_type, 0, chroma );//写入I宏块MB Header数据-CAVLC
66.
67.     #if !RDO_SKIP_BS
68.         i_mb_pos_tex = bs_pos( s );
69.         h->stat.frame.i_mv_bits += i_mb_pos_tex - i_mb_pos_start;
70.     #endif
71.
72.     /* Coded block pattern */
73.     if( i_mb_type != I_16x16 )
74.         bs_write_ue( s, cbp_to_golomb[chroma][IS_INTRA(i_mb_type)][(h->mb.i_cbp_chroma << 4)|h->mb.i_cbp_luma] );
75.
76.     /* transform size 8x8 flag */
77.     if( x264_mb_transform_8x8_allowed( h ) && h->mb.i_cbp_luma )
78.         bs_writel( s, h->mb.b_transform_8x8 );
79.
80.     if( i_mb_type == I_16x16 )
81.     {
82.         x264_cavlc_qp_delta( h );
83.
84.         /* DC Luma */
85.         for( int p = 0; p < plane_count; p++ )
86.         {
87.             x264_cavlc_block_residual( h, DCT_LUMA_DC, LUMA_DC+p, h-> dct.luma16x16_dc[p] );
88.         }
89.     }
90. }
```

```

88.
89.     /* AC Luma */
90.     if( h->mb.i_cbp_luma )
91.         for( int i = p*16; i < p*16+16; i++ )
92.             x264_cavlc_block_residual( h, DCT_LUMA_AC, i, h->dict.luma4x4[i]+1 );
93.     }
94. }
95. else if( h->mb.i_cbp_luma | h->mb.i_cbp_chroma )
96. {
97.     x264_cavlc_qp_delta( h );
98.     //残差数据
99.     x264_cavlc_macroblock_luma_residual( h, plane_count );
100. }
101. if( h->mb.i_cbp_chroma )
102. {
103.     /* Chroma DC residual present */
104.     x264_cavlc_block_residual( h, DCT_CHROMA_DC, CHROMA_DC+0, h->dict.chroma_dc[0] );
105.     x264_cavlc_block_residual( h, DCT_CHROMA_DC, CHROMA_DC+1, h->dict.chroma_dc[1] );
106.     if( h->mb.i_cbp_chroma == 2 ) /* Chroma AC residual present */
107.     {
108.         int step = 8 << CHROMA_V_SHIFT;
109.         for( int i = 16; i < 3*16; i += step )
110.             for( int j = i; j < i+4; j++ )
111.                 x264_cavlc_block_residual( h, DCT_CHROMA_AC, j, h->dict.luma4x4[j]+1 );
112.     }
113. }
114.
115. #if !RDO_SKIP_BS
116.     h->stat.frame.i_tex_bits += bs_pos(s) - i_mb_pos_tex;
117. #endif
118. }

```

从源代码可以看出，x264_macroblock_write_cavlc()的流程大致如下：

- (1) 根据Slice类型的不同，调用不同的函数输出宏块头（MB Header）：
 - a)对于P Slice，调用x264_cavlc_mb_header_p()
 - b)对于B Slice，调用x264_cavlc_mb_header_b()
 - c)对于I Slice，调用x264_cavlc_mb_header_i()
- (2) 调用x264_cavlc_qp_delta()输出宏块QP值
- (3) 调用x264_cavlc_block_residual()输出CAVLC编码的残差数据

下文将会分别分析其中涉及到的几个函数。

x264_cavlc_mb_header_i()

x264_cavlc_mb_header_i()用于输出I Slice中宏块的宏块头（MB Header）。该函数的定义位于encoder/cavlc.c，如下所示。

```



1. //写入I宏块Header数据-CAVLC
2. static void x264_cavlc_mb_header_i( x264_t *h, int i_mb_type, int i_mb_i_offset, int chroma )
3. {
4.     bs_t *s = &h->out.bs;
5.     if( i_mb_type == I_16x16 )
6.     {
7.         bs_write_ue( s, i_mb_i_offset + 1 + x264_mb_pred_mode16x16_fix[h->mb.i_intra16x16_pred_mode] +
8.             h->mb.i_cbp_chroma * 4 + ( h->mb.i_cbp_luma == 0 ? 0 : 12 ) );
9.     }
10.    else //if( i_mb_type == I_4x4 || i_mb_type == I_8x8 )
11.    {
12.        int di = i_mb_type == I_8x8 ? 4 : 1;
13.        bs_write_ue( s, i_mb_i_offset + 0 );
14.        if( h->pps->b_transform_8x8_mode )
15.            bs_writel( s, h->mb.b_transform_8x8 );
16.
17.        /* Prediction: Luma */
18.        for( int i = 0; i < 16; i += di )
19.        {
20.            //写入Intra4x4宏块的帧内预测模式
21.
22.            //获得帧内模式的预测值（通过左边和上边的块）
23.            int i_pred = x264_mb_predict_intra4x4_mode( h, i );
24.            //获得当前帧内模式
25.            int i_mode = x264_mb_pred_mode4x4_fix( h->mb.cache.intra4x4_pred_mode[x264_scan8[i]] );
26.
27.            if( i_pred == i_mode )
28.                bs_writel( s, 1 ); //如果当前模式正好等于预测值/* b_prev_intra4x4_pred_mode */
29.            else
30.                bs_write( s, 4, i_mode - (i_mode > i_pred) );//否则传送差值（差值=当前模式-预测模式）
31.        }
32.    }
33.
34.    if( chroma )
35.        bs_write_ue( s, x264_mb_chroma_pred_mode_fix[h->mb.i_chroma_pred_mode] );
36. }

```

从源代码可以看出，x264_cavlc_mb_header_i()在宏块为Intra16x16和Intra4x4的时候做了不同的处理。在Intra4x4帧内编码的宏块中，每个4x4的子块都有自己的帧内预测方式。H.264码流中并不是直接保存了每个子块的帧内预测方式（不利于压缩）。而是优先通过有周围块的信息推测当前块的帧内预测模式。具体的方法就是获取到左边块和上边块的预测模式，然后取它们的最小值作为当前块的预测模式。X264中有关这一部分的实现位于x264_mb_predict_intra4x4_mode()函数中。

x264_mb_predict_intra4x4_mode()

x264_mb_predict_intra4x4_mode()用于在Intra4x4宏块中获得当前块模式的预测值，定义如下所示。

```
[cpp]    
1. //获得Intra4x4帧内模式的预测值  
2. static ALWAYS_INLINE int x264_mb_predict_intra4x4_mode( x264_t *h, int idx )  
3. {  
4.     //左边4x4块的帧内预测模式  
5.     const int ma = h->mb.cache.intra4x4_pred_mode[x264_scan8[idx] - 1];  
6.     //上边4x4块的帧内预测模式  
7.     const int mb = h->mb.cache.intra4x4_pred_mode[x264_scan8[idx] - 8];  
8.     //取左边和上边的最小值，作为预测值  
9.     const int m = X264_MIN( x264_mb_pred_mode4x4_fix(ma),  
10.                            x264_mb_pred_mode4x4_fix(mb) );  
11.  
12.     if( m < 0 )  
13.         return I_PRED_4x4_DC;  
14.  
15.     return m;  
16. }
```

x264_cavlc_mb_header_i()会将x264_mb_predict_intra4x4_mode()得到的预测值与当前宏块实际的预测模式进行比较，如果正好相等则可以略去不传，如果不等的话则传送它们的差值。

x264_cavlc_mb_header_p()

x264_cavlc_mb_header_p()用于输出P Slice中宏块的宏块头（MB Header）。该函数的定义位于encoder/cavlc.c，如下所示。

```
[cpp]    
1. //写入P宏块Header数据-CAVLC  
2. static ALWAYS_INLINE void x264_cavlc_mb_header_p( x264_t *h, int i_mb_type, int chroma )  
3. {  
4.     bs_t *s = &h->out.bs;  
5.     if( i_mb_type == P_L0 )  
6.     {  
7.         if( h->mb.i_partition == D_16x16 )  
8.         {  
9.             bs_writel( s, 1 );  
10.            //写入参考帧序号  
11.            if( h->mb.pic.i_fref[0] > 1 )  
12.                bs_write_te( s, h->mb.pic.i_fref[0] - 1, h->mb.cache.ref[0][x264_scan8[0]] );  
13.  
14.            /*  
15.             * 向码流中写入MVD  
16.             *  
17.             * 运动矢量缓存mv[]  
18.             * 第3个参数是mv数据的起始点（scan8[]序号），在这里是mv[scan8[0]]  
19.             *  
20.             * 写入了Y  
21.             * |  
22.             * +-----+  
23.             * | 0 0 0 0 0 0 0 0  
24.             * | 0 0 0 0 Y 1 1 1  
25.             * | 0 0 0 0 1 1 1 1  
26.             * | 0 0 0 0 1 1 1 1  
27.             * | 0 0 0 0 1 1 1 1  
28.             */  
29.            x264_cavlc_mvd( h, 0, 0, 4 );  
30.  
31.        }  
32.        else if( h->mb.i_partition == D_16x8 )  
33.        {  
34.            bs_write_ue( s, 1 );  
35.            /*  
36.             * 向码流中写入参考帧序号、MVD  
37.             * 写入了Y  
38.             * |  
39.             * +-----+  
40.             * | 0 0 0 0 0 0 0 0  
41.             * | 0 0 0 0 Y 1 1 1  
42.             * | 0 0 0 0 1 1 1 1  
43.             * | 0 0 0 0 Y 2 2 2  
44.             * | 0 0 0 0 2 2 2 2  
45.             */  
46.            x264_cavlc_mvd( h, 0, 0, 4 );  
47.        }  
48.    }  
49. }
```

```

45.         */
46.         if( h->mb.pic.i_fref[0] > 1 )
47.         {
48.             bs_write_te( s, h->mb.pic.i_fref[0] - 1, h->mb.cache.ref[0][x264_scan8[0]] );
49.             bs_write_te( s, h->mb.pic.i_fref[0] - 1, h->mb.cache.ref[0][x264_scan8[8]] );
50.         }
51.
52.         x264_cavlc_mvd( h, 0, 0, 4 );
53.         x264_cavlc_mvd( h, 0, 8, 4 );
54.     }
55.     else if( h->mb.i_partition == D_8x16 )
56.     {
57.         bs_write_ue( s, 2 );
58.         /*
59.          * 向码流中写入参考帧序号、MVD
60.          * 写入了Y
61.          * |
62.          * ---+-----
63.          * | 0 0 0 0 0 0 0 0
64.          * | 0 0 0 0 Y 1 Y 2
65.          * | 0 0 0 0 1 1 2 2
66.          * | 0 0 0 0 1 1 2 2
67.          * | 0 0 0 0 1 1 2 2
68.          */
69.
70.         if( h->mb.pic.i_fref[0] > 1 )
71.         {
72.             bs_write_te( s, h->mb.pic.i_fref[0] - 1, h->mb.cache.ref[0][x264_scan8[0]] );
73.             bs_write_te( s, h->mb.pic.i_fref[0] - 1, h->mb.cache.ref[0][x264_scan8[4]] );
74.         }
75.
76.         x264_cavlc_mvd( h, 0, 0, 2 );
77.         x264_cavlc_mvd( h, 0, 4, 2 );
78.     }
79. }
80. else if( i_mb_type == P_8x8 )
81. {
82.     int b_sub_ref;
83.     if( (h->mb.cache.ref[0][x264_scan8[0]] | h->mb.cache.ref[0][x264_scan8[ 4]] |
84.         h->mb.cache.ref[0][x264_scan8[8]] | h->mb.cache.ref[0][x264_scan8[12]]) == 0 )
85.     {
86.         bs_write_ue( s, 4 );
87.         b_sub_ref = 0;
88.     }
89.     else
90.     {
91.         bs_write_ue( s, 3 );
92.         b_sub_ref = 1;
93.     }
94.
95.     /* sub mb type */
96.     if( h->param.analyse.inter & X264_ANALYSE_PSUB8x8 )
97.         for( int i = 0; i < 4; i++ )
98.             bs_write_ue( s, subpartition_p_to_golomb[ h->mb.i_sub_partition[i] ] );
99.     else
100.        bs_write( s, 4, 0xf );
101.
102.     /* ref0 */
103.     //参考帧序号
104.     if( b_sub_ref )
105.     {
106.         bs_write_te( s, h->mb.pic.i_fref[0] - 1, h->mb.cache.ref[0][x264_scan8[0]] );
107.         bs_write_te( s, h->mb.pic.i_fref[0] - 1, h->mb.cache.ref[0][x264_scan8[4]] );
108.         bs_write_te( s, h->mb.pic.i_fref[0] - 1, h->mb.cache.ref[0][x264_scan8[8]] );
109.         bs_write_te( s, h->mb.pic.i_fref[0] - 1, h->mb.cache.ref[0][x264_scan8[12]] );
110.     }
111.
112.     //写入8x8块的子块的MVD
113.     for( int i = 0; i < 4; i++ )
114.         x264_cavlc_8x8_mvd( h, i );
115. }
116. else //if( IS_INTRA( i_mb_type ) )
117.     x264_cavlc_mb_header_i( h, i_mb_type, 5, chroma );
118. }

```

从源代码可以看出，x264_cavlc_mb_header_p()主要完成了输出P宏块参考帧序号和运动矢量的功能。对于P16x16、P16x8、P8x16、P8x8这几种方式采用了类似的输出方式。需要注意运动矢量信息在H.264中是以MVD（运动矢量差值）的方式存储的（而不是直接存储）。一个宏块真正的运动矢量应该使用下式计算：

$$MV = \text{预测}MV + MVD$$

其中“预测MV”是由当前宏块的左边，上边，以及右上方宏块的MV预测而来。预测的方式就是取这3个块的中值（注意不是平均值）。X264中输出MVD的函数是x264_cavlc_mvd()。

x264_cavlc_mvd()

x264_cavlc_mvd()用于输出运动矢量的MVD信息。该函数的定义如下所示。

```

1. //写入MVD
2. static void x264_cavlc_mvd( x264_t *h, int i_list, int idx, int width )
3. {
4.     bs_t *s = &h->out.bs;
5.     ALIGNED_4( int16_t mvp[2] );
6.     //获得预测MV
7.     x264_mb_predict_mv( h, i_list, idx, width, mvp );
8.     //实际存储MVD
9.     //MVD=MV-预测MV
10.    bs_write_se( s, h->mb.cache.mv[i_list][x264_scan8[idx]][0] - mvp[0] );
11.    bs_write_se( s, h->mb.cache.mv[i_list][x264_scan8[idx]][1] - mvp[1] );
12. }

```

从源代码可以看出，x264_cavlc_mvd()首先调用x264_mb_predict_mv()通过左边，上边和右上宏块的运动矢量推算出预测运动矢量，然后将当前实际运动矢量与预测运动矢量相减后输出。

x264_mb_predict_mv()

x264_mb_predict_mv()用于获得预测的运动矢量。该函数的定义如下所示。

```

1. //获得预测的运动矢量MV (通过取中值)
2. void x264_mb_predict_mv( x264_t *h, int i_list, int idx, int i_width, int16_t mvp[2] )
3. {
4.     const int i8 = x264_scan8[idx];
5.     const int i_ref = h->mb.cache.ref[i_list][i8];
6.     int i_refa = h->mb.cache.ref[i_list][i8 - 1];
7.     int16_t *mv_a = h->mb.cache.mv[i_list][i8 - 1];
8.     int i_refb = h->mb.cache.ref[i_list][i8 - 8];
9.     int16_t *mv_b = h->mb.cache.mv[i_list][i8 - 8];
10.    int i_refc = h->mb.cache.ref[i_list][i8 - 8 + i_width];
11.    int16_t *mv_c = h->mb.cache.mv[i_list][i8 - 8 + i_width];
12.
13.    // Partitions not yet reached in scan order are unavailable.
14.    if( (idx&3) >= 2 + (i_width&1) || i_refc == -2 )
15.    {
16.        i_refc = h->mb.cache.ref[i_list][i8 - 8 - 1];
17.        mv_c = h->mb.cache.mv[i_list][i8 - 8 - 1];
18.
19.        if( SLICE_MBAFF
20.            && h->mb.cache.ref[i_list][x264_scan8[0]-1] != -2
21.            && MB_INTERLACED != h->mb.field[h->mb.i_mb_left_xy[0]] )
22.        {
23.            if( idx == 2 )
24.            {
25.                mv_c = h->mb.cache.topright_mv[i_list][0];
26.                i_refc = h->mb.cache.topright_ref[i_list][0];
27.            }
28.            else if( idx == 8 )
29.            {
30.                mv_c = h->mb.cache.topright_mv[i_list][1];
31.                i_refc = h->mb.cache.topright_ref[i_list][1];
32.            }
33.            else if( idx == 10 )
34.            {
35.                mv_c = h->mb.cache.topright_mv[i_list][2];
36.                i_refc = h->mb.cache.topright_ref[i_list][2];
37.            }
38.        }
39.    }
40.    if( h->mb.i_partition == D_16x8 )
41.    {
42.        if( idx == 0 )
43.        {
44.            if( i_refb == i_ref )
45.            {
46.                CP32( mvp, mv_b );
47.                return;
48.            }
49.        }
50.        else
51.        {
52.            if( i_refa == i_ref )
53.            {
54.                CP32( mvp, mv_a );
55.                return;
56.            }
57.        }
58.    }
59.    else if( h->mb.i_partition == D_8x16 )
60.    {
61.        if( idx == 0 )
62.        {
63.            if( i_refa == i_ref )

```

```

64.         {
65.             CP32( mvp, mv_a );
66.             return;
67.         }
68.     }
69.     else
70.     {
71.         if( i_refc == i_ref )
72.         {
73.             CP32( mvp, mv_c );
74.             return;
75.         }
76.     }
77. }
78.
79. int i_count = (i_refa == i_ref) + (i_refb == i_ref) + (i_refc == i_ref);
80. //如果可参考运动矢量的个数大于1个
81. if( i_count > 1 )
82. {
83. median:
84.     //取中值
85.     //x264_median_mv()内部调用了2次x264_median(), 分别求了运动矢量的x分量和y分量的中值
86.     x264_median_mv( mvp, mv_a, mv_b, mv_c );
87. }
88. else if( i_count == 1 ) //如果可参考运动矢量的个数只有1个
89. {
90.     //直接赋值
91.     if( i_refa == i_ref )
92.         CP32( mvp, mv_a );
93.     else if( i_refb == i_ref )
94.         CP32( mvp, mv_b );
95.     else
96.         CP32( mvp, mv_c );
97. }
98. else if( i_refb == -2 && i_refc == -2 && i_refa != -2 )
99.     CP32( mvp, mv_a );
100. else
101.     goto median;
102. }

```

可以看出x264_mb_predict_mv()去了左边，上边，右上宏块运动矢量的中值作为预测的运动矢量。其中的x264_median_mv()是一个取中值的函数。

x264_cavlc_qp_delta()

x264_cavlc_qp_delta()用于输出宏块的QP信息。该函数的定义如下所示。

```



1. //QP
2. static void x264_cavlc_qp_delta( x264_t *h )
3. {
4.     bs_t *s = &h->out.bs;
5.     //相减
6.     int i_dqp = h->mb.i_qp - h->mb.i_last_qp;
7.
8.     /* Avoid writing a delta quant if we have an empty 16x16 block, e.g. in a completely
9.      * flat background area. Don't do this if it would raise the quantizer, since that could
10.     * cause unexpected deblocking artifacts. */
11.     if( h->mb.i_type == I_16x16 && !(h->mb.i_cbp_luma | h->mb.i_cbp_chroma)
12.         && !h->mb.cache.non_zero_count[x264_scan8[LUMA_DC]]
13.         && !h->mb.cache.non_zero_count[x264_scan8[CHROMA_DC+0]]
14.         && !h->mb.cache.non_zero_count[x264_scan8[CHROMA_DC+1]]
15.         && h->mb.i_qp > h->mb.i_last_qp )
16.     {
17. #if !RDO_SKIP_BS
18.         h->mb.i_qp = h->mb.i_last_qp;
19. #endif
20.         i_dqp = 0;
21.     }
22.
23.     if( i_dqp )
24.     {
25.         if( i_dqp < -(QP_MAX_SPEC+1)/2 )
26.             i_dqp += QP_MAX_SPEC+1;
27.         else if( i_dqp > QP_MAX_SPEC/2 )
28.             i_dqp -= QP_MAX_SPEC+1;
29.     }
30.     bs_write_se( s, i_dqp );
31. }

```

在这里需要注意，QP信息在H.264码流中是以“QP偏移值”的形式存储的。“QP偏移值”指的是当前宏块和上一个宏块之间的差值。因此x264_cavlc_qp_delta()中使用当前宏块的QP减去上一个宏块的QP之后再进行输出。

x264_cavlc_macroblock_luma_residual()

x264_cavlc_macroblock_luma_residual()用于将残差数据以CAVLC编码的方式输出出来。该函数的定义如下所示。

```
[cpp]    
1. static ALWAYS_INLINE void x264_cavlc_macroblock_luma_residual( x264_t *h, int plane_count )  
2. {  
3.     if( h->mb.b_transform_8x8 )  
4.     {  
5.         /* shuffle 8x8 dct coeffs into 4x4 lists */  
6.         for( int p = 0; p < plane_count; p++ )  
7.             for( int i8 = 0; i8 < 4; i8++ )  
8.                 if( h->mb.cache.non_zero_count[x264_scan8[p*16+i8*4]] )  
9.                     h->zigzagf.interleave_8x8_cavlc( h-> dct.luma4x4[p*16+i8*4], h-> dct.luma8x8[p*4+i8],  
10.                    &h->mb.cache.non_zero_count[x264_scan8[p*16+i8*4]] );  
11.     }  
12.  
13.     for( int p = 0; p < plane_count; p++ )  
14.         FOREACH_BIT( i8, 0, h->mb.i_cbp_luma )  
15.             for( int i4 = 0; i4 < 4; i4++ )  
16.                 x264_cavlc_block_residual( h, DCT_LUMA_4x4, i4+i8*4+p*16, h-> dct.luma4x4[i4+i8*4+p*16] );  
17. }
```

从源代码可以看出，x264_cavlc_macroblock_luma_residual()调用了x264_cavlc_block_residual()进行残差数据的输出。由于x264_cavlc_block_residual()的源代码还没有看过，就不再深入分析了。

至此有关x264熵编码模块的源代码就分析完毕了。

雷霄骅

leixiaohua1020@126.com

<http://blog.csdn.net/leixiaohua1020>

版权声明：本文为博主原创文章，未经博主允许不得转载。 <https://blog.csdn.net/leixiaohua1020/article/details/45944811>

文章标签： [x264](#) [MVD](#) [输出](#) [CAVLC](#) [CABAC](#)

个人分类： [x264](#)

所属专栏： [开源多媒体项目源代码分析](#)

此PDF由spygg生成,请尊重原作者版权!!!

我的邮箱:liushidc@163.com