

原 FFMpeg的H.264解码器源代码简单分析：熵解码（Entropy Decoding）部分

2015年04月18日 17:19:15 阅读数：11444

=====

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](#)

=====

本文分析FFmpeg的H.264解码器的熵解码(Entropy Decoding)部分的源代码。FFmpeg的H.264解码器调用decode_slice()函数完成了解码工作。这些解码工作可以大体上分为3个步骤：熵解码，宏块解码以及环路滤波。本文分析这3个步骤中的第1个步骤。

函数调用关系图

熵解码（Entropy Decoding）部分的源代码在整个H.264解码器中的位置如下图所示。

[单击查看更清晰的图片](#)

熵解码（Entropy Decoding）部分的源代码的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，FFmpeg的熵解码方面的函数有两个：ff_h264_decode_mb_cabac()和ff_h264_decode_mb_cavlc()。ff_h264_decode_mb_cabac()用于解码CABAC编码方式的H.264数据，ff_h264_decode_mb_cavlc()用于解码CAVLC编码方式的H.264数据。本文挑选了ff_h264_decode_mb_cavlc()函数进行分析。

ff_h264_decode_mb_cavlc()调用了很多的读取指数哥伦布编码数据的函数，例如get_ue_golomb_long(), get_ue_golomb(), get_se_golomb(), get_ue_golomb_31()等。此外在解码残差数据的时候，调用了decode_residual()函数，而decode_residual()会调用get_vlc2()函数读取CAVLC编码数据。

总而言之，“熵解码”部分的作用就是按照H.264语法和语义的规定，读取数据（宏块类型、运动矢量、参考帧、残差等）并且赋值到FFmpeg H.264解码器中相应的变量上。需要注意的是，“熵解码”部分并不使用这些变量还原视频数据。还原视频数据的功能在下一步“宏块解码”步骤中完成。

在开始看ff_h264_decode_mb_cavlc()之前先回顾一下decode_slice()函数。

decode_slice()

decode_slice()用于解码H.264的Slice。该函数完成了“熵解码”、“宏块解码”、“环路滤波”的功能。它的定义位于libavcodec\h264_slice.c，如下所示。

```
[cpp]
1. //解码slice
2. //三个主要步骤：
3. //1.熵解码（CAVLC/CABAC）
4. //2.宏块解码
5. //3.环路滤波
6. //此外还包含了错误隐藏代码
7. static int decode_slice(struct AVCodecContext *avctx, void *arg)
8. {
9.     H264Context *h = *(void **)arg;
10.    int lf_x_start = h->mb_x;
11.
12.    h->mb_skip_run = -1;
13.
14.    av_assert0(h->block_offset[15] == (4 * ((scan8[15] - scan8[0]) & 7) << h->pixel_shift) + 4 * h->linesize * ((scan8[15] - scan8[0]) >> 3));
15.
16.    h->is_complex = FRAME_MBAFF(h) || h->picture_structure != PICT_FRAME ||
17.        avctx->codec_id != AV_CODEC_ID_H264 ||
18.        (CONFIG_GRAY && (h->flags & CODEC_FLAG_GRAY));
19.
20.    if (!(h->avctx->active_thread_type & FF_THREAD_SLICE) && h->picture_structure == PICT_FRAME && h->er.error_status_table) {
21.        const int start_i = av_clip(h->resync_mb_x + h->resync_mb_y * h->mb_width, 0, h->mb_num - 1);
22.        if (start_i) {
23.            int prev_status = h->er.error_status_table[h->er.mb_index2xy[start_i - 1]];
24.            prev_status &= ~ VP_START;
25.            if (prev_status != (ER_MV_END | ER_DC_END | ER_AC_END))
26.                h->er.error_occurred = 1;
27.        }
28.    }
29.    //CABAC情况
30.    if (h->pps.cabac) {
31.        /* realign */
32.        align_get_bits(&h->gb);
33.
34.        /* init cabac */
35.        //初始化CABAC解码器
36.        ff_init_cabac_decoder(&h->cabac,
37.                             h->gb.buffer + get_bits_count(&h->gb) / 8,
38.                             (get_bits_left(&h->gb) + 7) / 8);
39.
40.        ff_h264_init_cabac_states(h);
41.        //循环处理每个宏块
42.        for (;;) {
43.            // START_TIMER
44.            //解码CABAC数据
45.            int ret = ff_h264_decode_mb_cabac(h);
46.            int eos;
47.            // STOP_TIMER("decode_mb_cabac")
48.            //解码宏块
49.            if (ret >= 0)
50.                ff_h264_hl_decode_mb(h);
51.
52.            // FIXME optimal? or let mb_decode decode 16x32 ?
53.            //宏块级帧场自适应。很少接触
54.            if (ret >= 0 && FRAME_MBAFF(h)) {
55.                h->mb_y++;
56.
57.                ret = ff_h264_decode_mb_cabac(h);
58.                //解码宏块
59.                if (ret >= 0)
60.                    ff_h264_hl_decode_mb(h);
61.                h->mb_y--;
62.            }
63.            eos = get_cabac_terminate(&h->cabac);
64.
65.            if ((h->workaround_bugs & FF_BUG_TRUNCATED) &&
66.                h->cabac.bytestream > h->cabac.bytestream_end + 2) {
67.                //错误隐藏
68.                er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x - 1,
69.                            h->mb_y, ER_MB_END);
70.                if (h->mb_x >= lf_x_start)
71.                    loop_filter(h, lf_x_start, h->mb_x + 1);
72.                return 0;
73.            }
74.            if (h->cabac.bytestream > h->cabac.bytestream_end + 2)
75.                av_log(h->avctx, AV_LOG_DEBUG, "bytestream overread %"PTRDIFF_SPECIFIER"\n", h->cabac.bytestream_end - h->cabac.bytestream);

```

```

76.         if (ret < 0 || h->cabac.bytestream > h->cabac.bytestream_end + 4) {
77.             av_log(h->avctx, AV_LOG_ERROR,
78.                 "error while decoding MB %d %d, bytestream %"PTRDIFF_SPECIFIER"\n",
79.                 h->mb_x, h->mb_y,
80.                 h->cabac.bytestream_end - h->cabac.bytestream);
81.             er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x,
82.                 h->mb_y, ER_MB_ERROR);
83.             return AVERROR_INVALIDDATA;
84.         }
85.         //mb_x自增
86.         //如果自增后超过了一行的mb个数
87.         if (++h->mb_x >= h->mb_width) {
88.             //环路滤波
89.             loop_filter(h, lf_x_start, h->mb_x);
90.             h->mb_x = lf_x_start = 0;
91.             decode_finish_row(h);
92.             //mb_y自增 (处理下一行)
93.             ++h->mb_y;
94.             //宏块级帧场自适应, 暂不考虑
95.             if (FIELD_OR_MBAFF_PICTURE(h)) {
96.                 ++h->mb_y;
97.                 if (FRAME_MBAFF(h) && h->mb_y < h->mb_height)
98.                     predict_field_decoding_flag(h);
99.             }
100.        }
101.        //如果mb_y超过了mb的行数
102.        if (eos || h->mb_y >= h->mb_height) {
103.            tprintf(h->avctx, "slice end %d %d\n",
104.                get_bits_count(&h->gb), h->gb.size_in_bits);
105.            er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x - 1,
106.                h->mb_y, ER_MB_END);
107.            if (h->mb_x > lf_x_start)
108.                loop_filter(h, lf_x_start, h->mb_x);
109.            return 0;
110.        }
111.    }
112.    } else {
113.        //CAVLC情况
114.        //循环处理每个宏块
115.        for (;;) {
116.            //解码宏块的CAVLC
117.            int ret = ff_h264_decode_mb_cavlc(h);
118.            //解码宏块
119.            if (ret >= 0)
120.                ff_h264_hl_decode_mb(h);
121.
122.            // FIXME optimal? or let mb_decode decode 16x32 ?
123.            if (ret >= 0 && FRAME_MBAFF(h)) {
124.                h->mb_y++;
125.                ret = ff_h264_decode_mb_cavlc(h);
126.
127.                if (ret >= 0)
128.                    ff_h264_hl_decode_mb(h);
129.                h->mb_y--;
130.            }
131.
132.            if (ret < 0) {
133.                av_log(h->avctx, AV_LOG_ERROR,
134.                    "error while decoding MB %d %d\n", h->mb_x, h->mb_y);
135.                er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x,
136.                    h->mb_y, ER_MB_ERROR);
137.                return ret;
138.            }
139.
140.            if (++h->mb_x >= h->mb_width) {
141.                //环路滤波
142.                loop_filter(h, lf_x_start, h->mb_x);
143.                h->mb_x = lf_x_start = 0;
144.                decode_finish_row(h);
145.                ++h->mb_y;
146.                if (FIELD_OR_MBAFF_PICTURE(h)) {
147.                    ++h->mb_y;
148.                    if (FRAME_MBAFF(h) && h->mb_y < h->mb_height)
149.                        predict_field_decoding_flag(h);
150.                }
151.                if (h->mb_y >= h->mb_height) {
152.                    tprintf(h->avctx, "slice end %d %d\n",
153.                        get_bits_count(&h->gb), h->gb.size_in_bits);
154.
155.                    if ( get_bits_left(&h->gb) == 0
156.                        || get_bits_left(&h->gb) > 0 && !(h->avctx->err_recognition & AV_EF_AGGRESSIVE)) {
157.                        //错误隐藏
158.                        er_add_slice(h, h->resync_mb_x, h->resync_mb_y,
159.                            h->mb_x - 1, h->mb_y, ER_MB_END);
160.
161.                        return 0;
162.                    } else {
163.                        er_add_slice(h, h->resync_mb_x, h->resync_mb_y,
164.                            h->mb_x, h->mb_y, ER_MB_END);
165.
166.                        return AVERROR_INVALIDDATA;

```

```

167.         }
168.     }
169. }
170.
171.     if (get_bits_left(&h->gb) <= 0 && h->mb_skip_run <= 0) {
172.         tprintf(h->avctx, "slice end %d %d\n",
173.             get_bits_count(&h->gb), h->gb.size_in_bits);
174.
175.         if (get_bits_left(&h->gb) == 0) {
176.             er_add_slice(h, h->resync_mb_x, h->resync_mb_y,
177.                 h->mb_x - 1, h->mb_y, ER_MB_END);
178.             if (h->mb_x > lf_x_start)
179.                 loop_filter(h, lf_x_start, h->mb_x);
180.
181.             return 0;
182.         } else {
183.             er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x,
184.                 h->mb_y, ER_MB_ERROR);
185.
186.             return AVERROR_INVALIDDATA;
187.         }
188.     }
189. }
190. }
191. }

```

可以看出decode_slice()的流程如下：

- (1) 判断H.264码流是CABAC编码还是CAVLC编码，进入不同的处理循环。
- (2) 如果是CABAC编码，首先调用ff_init_cabac_decoder()初始化CABAC解码器。然后进入一个循环，依次对每个宏块进行以下处理：
 - a)调用ff_h264_decode_mb_cabac()进行CABAC熵解码
 - b)调用ff_h264_hl_decode_mb()进行宏块解码
 - c)解码一行宏块之后调用loop_filter()进行环路滤波
 - d)此外还有可能调用er_add_slice()进行错误隐藏处理
- (3) 如果是CAVLC编码，直接进入一个循环，依次对每个宏块进行以下处理：
 - a)调用ff_h264_decode_mb_cavlc()进行CAVLC熵解码
 - b)调用ff_h264_hl_decode_mb()进行宏块解码
 - c)解码一行宏块之后调用loop_filter()进行环路滤波
 - d)此外还有可能调用er_add_slice()进行错误隐藏处理

可以看出，出了熵解码以外，宏块解码和环路滤波的函数是一样的。下面详细看一下CAVLC熵解码函数ff_h264_decode_mb_cavlc()。

ff_h264_decode_mb_cavlc()

ff_h264_decode_mb_cavlc()完成了FFmpeg H.264解码器中“熵解码”的功能。“熵解码”部分的作用就是按照H.264语法和语义的规定，读取数据（宏块类型、运动矢量、参考帧、残差等）并且赋值到FFmpeg H.264解码器中相应的变量上。具体说来就是完成了解析H.264码流中Slice Data的功能。该函数比较复杂，它的定义位于libavcodec/h264_cavlc.c，如下所示。

```

1.  /*
2.   * 注释：雷霄骅
3.   *  leixiaohua1020@126.com
4.   *  http://blog.csdn.net/leixiaohua1020
5.   *
6.   *  解码宏块的CAVLC数据
7.   *  解码Slice Data（注意不包含Slice Header）
8.   *
9.   */
10. int ff_h264_decode_mb_cavlc(H264Context *h){
11.     int mb_xy;
12.     int partition_count;
13.     unsigned int mb_type, cbp;
14.     int dct8x8_allowed= h->pps.transform_8x8_mode;
15.     //如果是YUV420或者YUV422，需要处理色度（YUV444中的UV直接当亮度处理）
16.     int decode_chroma = h->sps.chroma_format_idc == 1 || h->sps.chroma_format_idc == 2;
17.     const int pixel_shift = h->pixel_shift;
18.     unsigned local_ref_count[2];
19.     //mb_xy的计算方法
20.     mb_xy = h->mb_xy = h->mb_x + h->mb_y*h->mb_stride;
21.
22.     tprintf(h->avctx, "pic:%d mb:%d/%d\n", h->frame_num, h->mb_x, h->mb_y);
23.     cbp = 0; /* avoid warning. FIXME: find a solution without slowing

```

```

24.         down the code */
25.         //slice_type_nos意思是SI/SP 被映射为 I/P （即没有SI/SP这种帧）
26.         //处理Skip宏块 - 不携带任何数据
27.         //解码器通过周围已重建的宏块的数据来恢复skip块
28.         if(h->slice_type_nos != AV_PICTURE_TYPE_I){
29.             //熵编码为CAVLC时候特有的字段
30.             if(h->mb_skip_run== -1)
31.                 h->mb_skip_run= get_ue_golomb_long(&h->gb);
32.
33.             if (h->mb_skip_run-- ) {
34.                 if(FRAME_MBAFF(h) && (h->mb_y&1) == 0){
35.                     if(h->mb_skip_run==0)
36.                         h->mb_mbafl = h->mb_field_decoding_flag = get_bits1(&h->gb);
37.                 }
38.                 decode_mb_skip(h);
39.                 return 0;
40.             }
41.         }
42.         if (FRAME_MBAFF(h)) {
43.             if( (h->mb_y&1) == 0 )
44.                 h->mb_mbafl = h->mb_field_decoding_flag = get_bits1(&h->gb);
45.         }
46.
47.         h->prev_mb_skipped= 0;
48.         //获取宏块类型 (I,B,P)
49.         //I片中只允许出现I宏块
50.         //P片中即可以出现P宏块也可以出现I宏块
51.         //B片中即可以出现B宏块也可以出现I宏块
52.         //这个语义含义比较复杂, 需要查表
53.         mb_type= get_ue_golomb(&h->gb);
54.         //B
55.         if(h->slice_type_nos == AV_PICTURE_TYPE_B){
56.             //b_mb_type_info存储了B宏块的类型
57.             //type代表宏块类型
58.             //partition_count代表宏块分区数目
59.             if(mb_type < 23){
60.                 partition_count= b_mb_type_info[mb_type].partition_count;
61.                 mb_type= b_mb_type_info[mb_type].type;
62.             }else{
63.                 mb_type -= 23;
64.                 goto decode_intra_mb;
65.             }
66.             //P
67.         }else if(h->slice_type_nos == AV_PICTURE_TYPE_P){
68.             //p_mb_type_info存储了P宏块的类型
69.             //type代表宏块类型
70.             //partition_count代表宏块分区数目 (一般为1, 2, 4)
71.             if(mb_type < 5){
72.                 partition_count= p_mb_type_info[mb_type].partition_count;
73.                 mb_type= p_mb_type_info[mb_type].type;
74.             }else{
75.                 mb_type -= 5;
76.                 goto decode_intra_mb;
77.             }
78.         }else{
79.             //i_mb_type_info存储了I宏块的类型
80.             //注意i_mb_type_info和p_mb_type_info、b_mb_type_info是不一样的：
81.             //type：宏块类型。只有MB_TYPE_INTRA4x4, MB_TYPE_INTRA16x16 (基本上都是这种), MB_TYPE_INTRA_PCM三种
82.             //pred_mode：帧内预测方式 (四种：DC, Horizontal, Vertical, Plane)。
83.             //cbp：指亮度和色度分量的各小块的残差的编码方案, 所谓编码方案有以下几种：
84.             //      0) 所有残差 (包括 DC、AC) 都不编码。
85.             //      1) 只对 DC 系数编码。
86.             //      2) 所有残差 (包括 DC、AC) 都编码。
87.             av_assert2(h->slice_type_nos == AV_PICTURE_TYPE_I);
88.             if(h->slice_type == AV_PICTURE_TYPE_SI && mb_type)
89.                 mb_type--;
90.         decode_intra_mb:
91.             if(mb_type > 25){
92.                 av_log(h->avctx, AV_LOG_ERROR, "mb_type %d in %c slice too large at %d %d\n", mb_type, av_get_picture_type_char(h->slice_type), h->mb_x, h->mb_y);
93.                 return -1;
94.             }
95.             partition_count=0;
96.             cbp= i_mb_type_info[mb_type].cbp;
97.             h->intra16x16_pred_mode= i_mb_type_info[mb_type].pred_mode;
98.             mb_type= i_mb_type_info[mb_type].type;
99.         }
100.        //隔行
101.        if(MB_FIELD(h))
102.            mb_type |= MB_TYPE_INTERLACED;
103.
104.        h->slice_table[ mb_xy ]= h->slice_num;
105.        //I_PCM不常见
106.        if(IS_INTRA_PCM(mb_type)){
107.            const int mb_size = ff_h264_mb_sizes[h->sps.chroma_format_idc] *
108.                h->sps.bit_depth_luma;
109.
110.            // We assume these blocks are very rare so we do not optimize it.
111.            h->intra_pcm_ptr = align_get_bits(&h->gb);
112.            if (get_bits_left(&h->gb) < mb_size) {
113.                av_log(h->avctx, AV_LOG_ERROR, "Not enough data for an intra PCM block.\n");

```

```

114.         return AVERKUR_INVALIDDATA;
115.     }
116.     skip_bits_long(&h->gb, mb_size);
117.
118.     // In deblocking, the quantizer is 0
119.     h->cur_pic.qscale_table[mb_xy] = 0;
120.     // All coeffs are present
121.     memset(h->non_zero_count[mb_xy], 16, 48);
122.     //赋值
123.     h->cur_pic.mb_type[mb_xy] = mb_type;
124.     return 0;
125. }
126.
127. //
128. local_ref_count[0] = h->ref_count[0] << MB_MBAFF(h);
129. local_ref_count[1] = h->ref_count[1] << MB_MBAFF(h);
130.
131. /* 设置上左, 上, 上右, 左宏块的索引值和宏块类型
132. * 这4个宏块在解码过程中会用到
133. * 位置如下图所示
134. *
135. * +---+---+---+
136. * | UL | U | UR |
137. * +---+---+---+
138. * | L |   |   |
139. * +---+---+---+
140. */
141. fill_decode_neighbors(h, mb_type);
142. //填充Cache
143. fill_decode_caches(h, mb_type);
144.
145. /*
146. *
147. * 关于多次出现的scan8
148. *
149. * scan8[]是一个表格。表格中存储了一整个宏块的信息，每一个元素代表了一个“4x4块”（H.264中最小的处理单位）。
150. * scan8[]中的“8”，意思应该是按照8x8为单元来扫描？
151. * 因此可以理解为“按照8x8为单元来扫描4x4的块”？
152. *
153. * scan8中按照顺序分别存储了Y, U, V的索引值。具体的存储还是在相应的cache中。
154. *
155. * PS：“4x4”貌似是H.264解码器中最小的“块”单位
156. *
157. * cache中首先存储Y，然后存储U和V。cache中的存储方式如下所示。
158. * 其中数字代表了scan8[]中元素的索引值
159. * scan8[]中元素的值则代表了其代表的变量在cache中的索引值
160. * +---+---+---+---+---+---+---+---+---+---+---+---+
161. * |   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |   |
162. * +---+---+---+---+---+---+---+---+---+---+---+---+
163. * | 0 | 48|   |   |   | y| y| y| y|
164. * | 1 |   |   |   | y| 0| 1| 4| 5|
165. * | 2 |   |   |   | y| 2| 3| 6| 7|
166. * | 3 |   |   |   | y| 8| 9| 12| 13|
167. * | 4 |   |   |   | y| 10| 11| 14| 15|
168. * | 5 | 49|   |   | u| u| u| u|
169. * | 6 |   |   |   | u| 16| 17| 20| 21|
170. * | 7 |   |   |   | u| 18| 19| 22| 23|
171. * | 8 |   |   |   | u| 24| 25| 28| 29|
172. * | 9 |   |   |   | u| 26| 27| 30| 31|
173. * | 10| 50|   |   | v| v| v| v|
174. * | 11|   |   |   | v| 32| 33| 36| 37|
175. * | 12|   |   |   | v| 34| 35| 38| 39|
176. * | 13|   |   |   | v| 40| 41| 44| 45|
177. * | 14|   |   |   | v| 42| 43| 46| 47|
178. * |---+---+---+---+---+---+---+---+---+---+---+---+
179. * |   |   |
180. *
181. */
182.
183.
184. //mb_pred
185. //分成3种情况进行预测工作：
186. //1. 帧内预测
187. //2. 划分为4个块（此时每个8x8的块可以再次划分为4种类型）
188. //3. 其他类型（包括16x16, 16x8, 8x16，这些划分不可再次划分）
189. if(IS_INTRA(mb_type)){
190.     //情况1：帧内宏块
191.     int pred_mode;
192.     // init_top_left_availability(h);
193.     //如果是帧内4x4，帧内预测方式需要特殊处理（9种）
194.     if(IS_INTRA4x4(mb_type)){
195.         int i;
196.         int di = 1;
197.         //先不考虑这种相对特殊情况，认为di=1
198.         if(dct8x8_allowed && get_bits1(&h->gb)){
199.             mb_type |= MB_TYPE_8x8DCT;
200.             di = 4;
201.         }
202.
203.         // fill_intra4x4_pred_table(h);
204.         //对于一个宏块（16x16）来说，包含了4*4=16个4x4帧内预测的块
205.         //所以循环16次

```

```

205. //帧内预测16x16
206. /*
207.  * 帧内预测：16x16 宏块被划分为16个4x4子块
208.  *
209.  * +---+---+---+---+
210.  * |   |   |   |   |
211.  * +---+---+---+---+
212.  * |   |   |   |   |
213.  * +---+---+---+---+
214.  * |   |   |   |   |
215.  * +---+---+---+---+
216.  * |   |   |   |   |
217.  * +---+---+---+---+
218.  *
219.  */
220. for(i=0; i<16; i+=di){
221.     //获得对Intra4x4的预测模式的预测值（挺绕口，确实是这样）
222.     //这个预测模式由左边和上边块的预测模式（取最小值）推导出来
223.     int mode= pred_intra_mode(h, i);
224.     //这1bit是dcPredModePredictedFlag，如果为1，则直接使用推导出来的预测模式
225.     if(!get_bits1(&h->gb)){
226.         //否则就使用读取出来的预测模式
227.         const int rem_mode= get_bits(&h->gb, 3);
228.         mode = rem_mode + (rem_mode >= mode);
229.     }
230.
231.     if(di==4)
232.         fill_rectangle( &h->intra4x4_pred_mode_cache[ scan8[i] ], 2, 2, 8, mode, 1 );
233.     else
234.         h->intra4x4_pred_mode_cache[ scan8[i] ] = mode;//赋值
235.     /*
236.     * 将mode填充至intra4x4_pred_mode_cache
237.     *
238.     * 用简单图形表示intra4x4_pred_mode_cache如下。数字代表填充顺序（一共填充16次）
239.     * |
240.     * --+-----
241.     * | 0 0 0 0 0 0 0 0
242.     * | 0 0 0 0 1 2 5 6
243.     * | 0 0 0 0 3 4 7 8
244.     * | 0 0 0 0 9 10 13 14
245.     * | 0 0 0 0 11 12 15 16
246.     *
247.     */
248.
249. }
250. //将宏块的Cache中的intra4x4_pred_mode拷贝至整张图片的intra4x4_pred_mode变量中
251. write_back_intra_pred_mode(h);
252. if( ff_h264_check_intra4x4_pred_mode(h) < 0)
253.     return -1;
254. }else{
255.     //帧内16x16的检测：检查宏块上方和左边的数据是否可用
256.     h->intra16x16_pred_mode= ff_h264_check_intra_pred_mode(h, h->intra16x16_pred_mode, 0);
257.     if(h->intra16x16_pred_mode < 0)
258.         return -1;
259. }
260. if(decode_chroma){
261.     //色度帧内预测的检测，和亮度一样
262.     pred_mode= ff_h264_check_intra_pred_mode(h, get_ue_golomb_31(&h->gb), 1);
263.     if(pred_mode < 0)
264.         return -1;
265.     h->chroma_pred_mode= pred_mode;
266. } else {
267.     h->chroma_pred_mode = DC_128_PRED8x8;
268. }
269. }else if(partition_count==4){
270.     //情况2：宏块划分为4
271.     //为什么宏块划分为4的时候要单独处理？因为宏块划分为4的时候，每个8x8的子宏块还可以进一步划分为2个4x8，2个8x4（4x8），或者4个4x4。
272.     //而其他方式的宏块划分（例如16x16,16x8,8x16等）是不可以这样再次划分的
273.     /*
274.     * 16x16 宏块被划分为4个8x8子块
275.     *
276.     * +-----+-----+
277.     * |   |   |   |   |
278.     * | 0   |   | 1   |   |
279.     * |   |   |   |   |
280.     * +-----+-----+
281.     * |   |   |   |   |
282.     * | 2   |   | 3   |   |
283.     * |   |   |   |   |
284.     * +-----+-----+
285.     *
286.     */
287.     int i, j, sub_partition_count[4], list, ref[2][4];
288.     //获得8x8子块的宏块类型
289.     //后续的很多代码都是循环处理4个8x8子块
290.     //所以很多for()循环的次数都是为4
291.     if(h->slice_type_nos == AV_PICTURE_TYPE_B){
292.         //B宏块
293.         //4个子块
294.         for(i=0; i<4; i++){
295.             //子宏块的预测类型
296.             h->sub_mb_type[i]= get_ue_golomb_31(&h->gb);

```

```

297.         if(h->sub_mb_type[i] >=13){
298.             av_log(h->avctx, AV_LOG_ERROR, "B sub_mb_type %u out of range at %d %d\n", h->sub_mb_type[i], h->mb_x, h->mb_y);
299.
300.             return -1;
301.         }
302.         sub_partition_count[i]= b_sub_mb_type_info[ h->sub_mb_type[i] ].partition_count;
303.         h->sub_mb_type[i]=      b_sub_mb_type_info[ h->sub_mb_type[i] ].type;
304.     }
305.     if( IS_DIRECT(h->sub_mb_type[0]|h->sub_mb_type[1]|h->sub_mb_type[2]|h->sub_mb_type[3])) {
306.         ff_h264_pred_direct_motion(h, &mb_type);
307.         h->ref_cache[0][scan8[4]] =
308.         h->ref_cache[1][scan8[4]] =
309.         h->ref_cache[0][scan8[12]] =
310.         h->ref_cache[1][scan8[12]] = PART_NOT_AVAILABLE;
311.     }
312. }else{
313.     av_assert2(h->slice_type_nos == AV_PICTURE_TYPE_P); //FIXME SP correct ?
314.     //P宏块
315.     //4个子块
316.     for(i=0; i<4; i++){
317.         h->sub_mb_type[i]= get_ue_golomb_31(&h->gb);
318.         if(h->sub_mb_type[i] >=4){
319.             av_log(h->avctx, AV_LOG_ERROR, "P sub_mb_type %u out of range at %d %d\n", h->sub_mb_type[i], h->mb_x, h->mb_y);
320.
321.             return -1;
322.         }
323.         //p_sub_mb_type_info存储了P子宏块的类型, 和前面的p_mb_type_info类似
324.         //type代表宏块类型
325.         //partition_count代表宏块分区数目
326.         sub_partition_count[i]= p_sub_mb_type_info[ h->sub_mb_type[i] ].partition_count;
327.         h->sub_mb_type[i]=      p_sub_mb_type_info[ h->sub_mb_type[i] ].type;
328.     }
329. }
330. //8x8块的子宏块的参考帧序号
331. for(list=0; list<h->list_count; list++){
332.     int ref_count = IS_REF0(mb_type) ? 1 : local_ref_count[list];
333.     //4个子块
334.     for(i=0; i<4; i++){
335.         if(IS_DIRECT(h->sub_mb_type[i])) continue;
336.         if(IS_DIR(h->sub_mb_type[i], 0, list)){
337.             unsigned int tmp;
338.             if(ref_count == 1){
339.                 tmp= 0;
340.             }else if(ref_count == 2){
341.                 tmp= get_bits1(&h->gb)^1;
342.             }else{
343.                 //参考帧序号
344.                 tmp= get_ue_golomb_31(&h->gb);
345.                 if(tmp>ref_count){
346.                     av_log(h->avctx, AV_LOG_ERROR, "ref %u overflow\n", tmp);
347.                     return -1;
348.                 }
349.             }
350.             //存储
351.             ref[list][i]= tmp;
352.         }else{
353.             //FIXME
354.             ref[list][i] = -1;
355.         }
356.     }
357. }
358.
359. if(dct8x8_allowed)
360.     dct8x8_allowed = get_dct8x8_allowed(h);
361.
362. //8x8块的子宏块的运动矢量
363. //依次处理L0和L1
364. for(list=0; list<h->list_count; list++){
365.     //4个子块
366.     for(i=0; i<4; i++){
367.         if(IS_DIRECT(h->sub_mb_type[i])) {
368.             h->ref_cache[list][ scan8[4*i] ] = h->ref_cache[list][ scan8[4*i+1] ];
369.             continue;
370.         }
371.         h->ref_cache[list][ scan8[4*i] ]=h->ref_cache[list][ scan8[4*i+1] ]=
372.         h->ref_cache[list][ scan8[4*i+8] ]=h->ref_cache[list][ scan8[4*i+9] ]= ref[list][i];
373.
374.         if(IS_DIR(h->sub_mb_type[i], 0, list)){
375.             const int sub_mb_type= h->sub_mb_type[i];
376.             const int block_width= (sub_mb_type & (MB_TYPE_16x16|MB_TYPE_16x8)) ? 2 : 1;
377.             //8x8块的子块 (可能是8x8,8x4,4x8,4x4) 的运动矢量
378.             //依次处理, 数量为sub_partition_count
379.             for(j=0; j<sub_partition_count[i]; j++){
380.                 int mx, my;
381.                 //scan8索引
382.                 const int index= 4*i + block_width*j;
383.                 int16_t (* mv_cache)[2]= &h->mv_cache[list][ scan8[index] ];
384.                 //先获取“预测MV”(取中值), 结果存入mx, my
385.                 pred_motion(h, index, block_width, list, h->ref_cache[list][ scan8[index] ], &mx, &my);
386.                 //获取MVD并且累加至“预测MV”
387.                 //MV=预测MV+MVD

```



```

386. mx += get_se_golomb(&h->gb);
387. my += get_se_golomb(&h->gb);
388. tprintf(h->avctx, "final mv:%d %d\n", mx, my);
389.
390. if(IS_SUB_8X8(sub_mb_type)){
391.     //8x8子宏块的宏块划分方式为8x8（等同于没划分）
392.     //则把mv_cache中的4个块对应的值都赋值成一样的
393.     //即：[0], [1], [0+8], [1+8]
394.     //PS：stride（代表一行元素个数）为8（即“+8”代表是下一行）
395.     /*
396.     * +-----+
397.     * |           |
398.     * +   +   +
399.     * |           |
400.     * +-----+
401.     *
402.     */
403.     mv_cache[ 1 ][0]=
404.     mv_cache[ 8 ][0]= mv_cache[ 9 ][0]= mx;
405.     mv_cache[ 1 ][1]=
406.     mv_cache[ 8 ][1]= mv_cache[ 9 ][1]= my;
407. }else if(IS_SUB_8X4(sub_mb_type)){
408.     //如果是8x4子宏块
409.     //则把mv_cache中的横向的2个块对应的值都赋值成一样的
410.     //即：[0], [1]
411.     /*
412.     * +-----+
413.     * |           |
414.     * +-----+
415.     * |           |
416.     * +-----+
417.     *
418.     */
419.     mv_cache[ 1 ][0]= mx;
420.     mv_cache[ 1 ][1]= my;
421. }else if(IS_SUB_4X8(sub_mb_type)){
422.     //如果是4x8子宏块
423.     //则把mv_cache中纵向的2个块对应的值都赋值成一样的
424.     //即：[0], [0+8]
425.     /*
426.     * +-----+
427.     * |   |   |
428.     * +   +   +
429.     * |   |   |
430.     * +-----+
431.     *
432.     */
433.     mv_cache[ 8 ][0]= mx;
434.     mv_cache[ 8 ][1]= my;
435. }
436. //赋值
437. //PS：如果是4x4子宏块划分的话，则不会触发上面的if else语句，即分别得到4个4x4块的运动矢量
438. mv_cache[ 0 ][0]= mx;
439. mv_cache[ 0 ][1]= my;
440.
441.
442. /*
443. * mv_cache赋值方式如下
444. * scan8[0]代表了cache里面亮度Y的起始点，取值12
445. * 如果全部都是4x4划分的话，mv_cache填充顺序即按照scan8中元素中的顺序：
446. * scan8[0], scan8[1], scan8[2], scan8[3], scan8[4], scan8[5].....
447. * 即：
448. * 4 + 1 * 8, 5 + 1 * 8, 4 + 2 * 8, 5 + 2 * 8,
449. * 6 + 1 * 8, 7 + 1 * 8, 6 + 2 * 8, 7 + 2 * 8,
450. * 4 + 3 * 8, 5 + 3 * 8, 4 + 4 * 8, 5 + 4 * 8,.....
451. * 用简单图形表示mv_cache如下。数字代表填充顺序（一共填充16次）
452. * |
453. * +-----+
454. * | 0 0 0 0 0 0 0 0
455. * | 0 0 0 0 1 2 5 6
456. * | 0 0 0 0 3 4 7 8
457. * | 0 0 0 0 9 10 13 14
458. * | 0 0 0 0 11 12 15 16
459. *
460. * 如果全部是8x8划分的话，mv_cache填充顺序即按照scan8中元素中的顺序：
461. * scan8[0], scan8[4], scan8[8], scan8[16].....
462. * 填充后赋值3个元素
463. * 用简单图形表示mv_cache如下。数字代表填充顺序（一共填充4次）
464. * |
465. * +-----+
466. * | 0 0 0 0 0 0 0 0
467. * | 0 0 0 0 1 1 2 2
468. * | 0 0 0 0 1 1 2 2
469. * | 0 0 0 0 3 3 4 4
470. * | 0 0 0 0 3 3 4 4
471. *
472. * 如果全部是8x4划分的话，mv_cache填充顺序即按照scan8中元素中的顺序：
473. * scan8[0], scan8[2], scan8[4], scan8[6].....
474. * 填充后赋值右边1个元素
475. * 用简单图形表示mv_cache如下。数字代表填充顺序（一共填充8次）
476. * |
477. * +-----+

```

```

477.         * | 0 0 0 0 0 0 0 0
478.         * | 0 0 0 0 1 1 3 3
479.         * | 0 0 0 0 2 2 4 4
480.         * | 0 0 0 0 5 5 7 7
481.         * | 0 0 0 0 6 6 8 8
482.         *
483.         * 如果全部是4x8划分的话, mv_cache填充顺序即按照scan8中元素中的顺序:
484.         * scan8[0], scan8[1], scan8[4], scan8[5], scan8[8], scan8[9] .....
485.         * 填充后赋值下边1个元素
486.         * 用简单图形表示mv_cache如下。数字代表填充顺序 (一共填充8次)
487.         * |
488.         * +-----+
489.         * | 0 0 0 0 0 0 0 0
490.         * | 0 0 0 0 1 2 3 4
491.         * | 0 0 0 0 1 2 3 4
492.         * | 0 0 0 0 5 6 7 8
493.         * | 0 0 0 0 5 6 7 8
494.         *
495.         * 其他划分的不同组合, 可以参考上面的填充顺序
496.         */
497.     }
498. }else{
499.     uint32_t *p= (uint32_t *)&h->mv_cache[list][ scan8[4*i] ][0];
500.     p[0] = p[1]=
501.     p[8] = p[9]= 0;
502. }
503. }
504. }
505. }else if(IS_DIRECT(mb_type)){
506.     //Direct模式
507.     ff_h264_pred_direct_motion(h, &mb_type);
508.     dct8x8_allowed &= h->sps.direct_8x8_inference_flag;
509. }else{
510.     //情况3: 既不是帧内宏块 (情况1), 宏块划分数目也不为4 (情况2)
511.     //这种情况下不存在8x8的子宏块再次划分这样的事情
512.     int list, mx, my, i;
513.     //FIXME we should set ref_idx_l? to 0 if we use that later ...
514.     if(IS_16X16(mb_type)){
515.         /*
516.          * 16x16 宏块
517.          *
518.          * +-----+-----+
519.          * |               |
520.          * |               |
521.          * |               |
522.          * |               +
523.          * |               |
524.          * |               |
525.          * |               |
526.          * +-----+-----+
527.          */
528.         /*
529.          //运动矢量对应的参考帧
530.          //L0和L1
531.          for(list=0; list<h->list_count; list++){
532.              unsigned int val;
533.              if(IS_DIR(mb_type, 0, list)){
534.                  if(local_ref_count[list]==1){
535.                      val= 0;
536.                  } else if(local_ref_count[list]==2){
537.                      val= get_bits1(&h->gb)^1;
538.                  }else{
539.                      //参考帧图像序号
540.                      val= get_ue_golomb_31(&h->gb);
541.                      if (val >= local_ref_count[list]){
542.                          av_log(h->avctx, AV_LOG_ERROR, "ref %u overflow\n", val);
543.                          return -1;
544.                      }
545.                  }
546.              }
547.              //填充ref_cache
548.              //fill_rectangle(数据起始点, 宽, 高, 一行数据个数, 数据值, 每个数据占用的byte)
549.              //scan8[0]代表了cache里面亮度Y的起始点
550.              /*
551.               * 在这里相当于在ref_cache[list]填充了这样的一份数据 (val=v):
552.               * |
553.               * +-----+
554.               * | 0 0 0 0 0 0 0 0
555.               * | 0 0 0 0 v v v v
556.               * | 0 0 0 0 v v v v
557.               * | 0 0 0 0 v v v v
558.               * | 0 0 0 0 v v v v
559.               */
560.              fill_rectangle(&h->ref_cache[list][ scan8[0] ], 4, 4, 8, val, 1);
561.          }
562.          //运动矢量
563.          for(list=0; list<h->list_count; list++){
564.              if(IS_DIR(mb_type, 0, list)){
565.                  //预测MV (取中值)
566.                  pred_motion(h, 0, 4, list, h->ref_cache[list][ scan8[0] ], &mx, &my);
567.                  //MVD从码流中获取

```

```

568. //MV=预测MV+MVD
569. mx += get_se_golomb(&h->gb);
570. my += get_se_golomb(&h->gb);
571. tprintf(h->avctx, "final mv:%d %d\n", mx, my);
572. //填充mv_cache
573. //fill_rectangle(数据起始点, 宽, 高, 一行数据个数, 数据值, 每个数据占用的byte)
574. //scan8[0]代表了cache里面亮度Y的起始点
575. /*
576.  * 在这里相当于在mv_cache[list]填充了这样的一份数据 (val=v) :
577.  * |
578.  * +-----+
579.  * | 0 0 0 0 0 0 0 0
580.  * | 0 0 0 0 v v v v
581.  * | 0 0 0 0 v v v v
582.  * | 0 0 0 0 v v v v
583.  * | 0 0 0 0 v v v v
584.  */
585. fill_rectangle(h->mv_cache[list][ scan8[0] ], 4, 4, 8, pack16to32(mx,my), 4);
586. }
587. }
588. }
589. else if(IS_16X8(mb_type)){ //16x8
590. /*
591.  * 16x8 宏块划分
592.  *
593.  * +-----+-----+
594.  * |         |         |
595.  * |         |         |
596.  * |         |         |
597.  * +-----+-----+
598.  */
599. /*
600. //运动矢量对应的参考帧
601. for(list=0; list<h->list_count; list++){
602. //横着的2个
603. for(i=0; i<2; i++){
604. //存储在val
605. unsigned int val;
606. if(IS_DIR(mb_type, i, list)){
607. if(local_ref_count[list] == 1) {
608. val= 0;
609. } else if(local_ref_count[list] == 2) {
610. val= get_bits1(&h->gb)^1;
611. }else{
612. val= get_ue_golomb_31(&h->gb);
613. if (val >= local_ref_count[list]){
614. av_log(h->avctx, AV_LOG_ERROR, "ref %u overflow\n", val);
615. return -1;
616. }
617. }
618. }else
619. val= LIST_NOT_USED&0xFF;
620. //填充ref_cache
621. //fill_rectangle(数据起始点, 宽, 高, 一行数据个数, 数据值, 每个数据占用的byte)
622. //scan8[0]代表了cache里面亮度Y的起始点
623. /*
624.  * 在这里相当于在ref_cache[list]填充了这样的一份数据 (第一次循环val=1, 第二次循环val=2) :
625.  * |
626.  * +-----+
627.  * | 0 0 0 0 0 0 0 0
628.  * | 0 0 0 0 1 1 1 1
629.  * | 0 0 0 0 1 1 1 1
630.  * | 0 0 0 0 2 2 2 2
631.  * | 0 0 0 0 2 2 2 2
632.  */
633. fill_rectangle(&h->ref_cache[list][ scan8[0] + 16*i ], 4, 2, 8, val, 1);
634. }
635. }
636. //运动矢量
637. for(list=0; list<h->list_count; list++){
638. //2个
639. for(i=0; i<2; i++){
640. //存储在val
641. unsigned int val;
642. if(IS_DIR(mb_type, i, list)){
643. //预测MV
644. pred_16x8_motion(h, 8*i, list, h->ref_cache[list][scan8[0] + 16*i], &mx, &my);
645. //MV=预测MV+MVD
646. mx += get_se_golomb(&h->gb);
647. my += get_se_golomb(&h->gb);
648. tprintf(h->avctx, "final mv:%d %d\n", mx, my);
649. //打包?
650. val= pack16to32(mx,my);
651. }else
652. val=0;
653. //填充mv_cache
654. //fill_rectangle(数据起始点, 宽, 高, 一行数据个数, 数据值, 每个数据占用的byte)
655. //scan8[0]代表了cache里面亮度Y的起始点
656. /*
657.  * 在这里相当于在ref_cache[list]填充了这样的一份数据 (第一次循环val=1, 第二次循环val=2) :
658.  * |
659.  * +-----+

```

```

659.         *  --+-----+
660.         *  | 0 0 0 0 0 0 0 0
661.         *  | 0 0 0 0 1 1 1 1
662.         *  | 0 0 0 0 1 1 1 1
663.         *  | 0 0 0 0 2 2 2 2
664.         *  | 0 0 0 0 2 2 2 2
665.         */
666.         fill_rectangle(h->mv_cache[list][ scan8[0] + 16*i ], 4, 2, 8, val, 4);
667.     }
668. }
669. }else{ //8x16?
670.     /*
671.     * 8x16 宏块划分
672.     *
673.     * +-----+
674.     * |         |
675.     * |         |
676.     * |         |
677.     * +-----+
678.     * |         |
679.     * |         |
680.     * |         |
681.     * +-----+
682.     *
683.     */
684.     av_assert2(IS_8X16(mb_type));
685.     for(list=0; list<h->list_count; list++){
686.         //竖着的2个
687.         for(i=0; i<2; i++){
688.             unsigned int val;
689.             if(IS_DIR(mb_type, i, list)){ //FIXME optimize
690.                 if(local_ref_count[list]==1){
691.                     val= 0;
692.                 } else if(local_ref_count[list]==2){
693.                     val= get_bits1(&h->gb)^1;
694.                 }else{
695.                     val= get_ue_golomb_31(&h->gb);
696.                     if (val >= local_ref_count[list]){
697.                         av_log(h->avctx, AV_LOG_ERROR, "ref %u overflow\n", val);
698.                         return -1;
699.                     }
700.                 }
701.             }else
702.                 val= LIST_NOT_USED&0xFF;
703.             //填充ref_cache
704.             //fill_rectangle(数据起始点, 宽, 高, 一行数据个数, 数据值, 每个数据占用的byte)
705.             //scan8[0]代表了cache里面亮度Y的起始点
706.             /*
707.             * 在这里相当于在ref_cache[list]填充了这样的一份数据 (第一次循环val=1, 第二次循环val=2) :
708.             *
709.             *  --+-----+
710.             *  | 0 0 0 0 0 0 0 0
711.             *  | 0 0 0 0 1 1 2 2
712.             *  | 0 0 0 0 1 1 2 2
713.             *  | 0 0 0 0 1 1 2 2
714.             *  | 0 0 0 0 1 1 2 2
715.             */
716.             fill_rectangle(&h->ref_cache[list][ scan8[0] + 2*i ], 2, 4, 8, val, 1);
717.         }
718.     }
719.     for(list=0; list<h->list_count; list++){
720.         for(i=0; i<2; i++){
721.             unsigned int val;
722.             if(IS_DIR(mb_type, i, list)){
723.                 //预测MV
724.                 pred_8x16_motion(h, i*4, list, h->ref_cache[list][ scan8[0] + 2*i ], &mx, &my);
725.                 //MV=预测MV+MVD
726.                 mx += get_se_golomb(&h->gb);
727.                 my += get_se_golomb(&h->gb);
728.                 tprintf(h->avctx, "final mv:%d %d\n", mx, my);
729.
730.                 val= pack16to32(mx,my);
731.             }else
732.                 val=0;
733.             //填充mv_cache
734.             //fill_rectangle(数据起始点, 宽, 高, 一行数据个数, 数据值, 每个数据占用的byte)
735.             //scan8[0]代表了cache里面亮度Y的起始点
736.             /*
737.             * 在这里相当于在mv_cache[list]填充了这样的一份数据 (第一次循环val=1, 第二次循环val=2) :
738.             *
739.             *  --+-----+
740.             *  | 0 0 0 0 0 0 0 0
741.             *  | 0 0 0 0 1 1 2 2
742.             *  | 0 0 0 0 1 1 2 2
743.             *  | 0 0 0 0 1 1 2 2
744.             *  | 0 0 0 0 1 1 2 2
745.             */
746.             fill_rectangle(h->mv_cache[list][ scan8[0] + 2*i ], 2, 4, 8, val, 4);
747.         }
748.     }
749. }
750. }

```

```

750.     }
751.     //将宏块的Cache中的MV拷贝至整张图片的motion_val变量中
752.     if(IS_INTER(mb_type))
753.         write_back_motion(h, mb_type);
754.
755.     //Intra16x16的CBP位于mb_type中，其他类型的宏块的CBP需要单独读取
756.     if(!IS_INTRA16x16(mb_type)){
757.         //获取CBP
758.         cbp= get_ue_golomb(&h->gb);
759.
760.         if(decode_chroma){
761.             //YUV420,YUV422的情况
762.             if(cbp > 47){
763.                 av_log(h->avctx, AV_LOG_ERROR, "cbp too large (%u) at %d %d\n", cbp, h->mb_x, h->mb_y);
764.                 return -1;
765.             }
766.             //获取CBP
767.             if(IS_INTRA4x4(mb_type)) cbp= golomb_to_intra4x4_cbp[cbp];
768.             else cbp= golomb_to_inter_cbp [cbp];
769.         }else{
770.             if(cbp > 15){
771.                 av_log(h->avctx, AV_LOG_ERROR, "cbp too large (%u) at %d %d\n", cbp, h->mb_x, h->mb_y);
772.                 return -1;
773.             }
774.             if(IS_INTRA4x4(mb_type)) cbp= golomb_to_intra4x4_cbp_gray[cbp];
775.             else cbp= golomb_to_inter_cbp_gray[cbp];
776.         }
777.     } else {
778.         if (!decode_chroma && cbp>15) {
779.             av_log(h->avctx, AV_LOG_ERROR, "gray chroma\n");
780.             return AVERROR_INVALIDDATA;
781.         }
782.     }
783.
784.     if(dct8x8_allowed && (cbp&15) && !IS_INTRA(mb_type)){
785.         mb_type |= MB_TYPE_8x8DCT*get_bits1(&h->gb);
786.     }
787.     //赋值CBP
788.     h->cbp=
789.     h->cbp_table[mb_xy]= cbp;
790.     //赋值mb_type
791.     h->cur_pic.mb_type[mb_xy] = mb_type;
792.
793.     /*
794.     * 亮度cbp取值（只有低4位有意义）：
795.     * 变量的最低位比特从最低位开始，每1位对应1个子宏块，该位等于1时表明对应子宏块残差系数被传送；
796.     * 该位等于0时表明对应子宏块残差全部不被传送
797.     * 色度cbp取值：
798.     * 0，代表所有残差都不被传送
799.     * 1，只传送DC
800.     * 2，传送DC+AC
801.     */
802.
803.     //cbp不为0，才有残差信息
804.     if(cbp || IS_INTRA16x16(mb_type)){
805.         int i4x4, i8x8, chroma_idx;
806.         int dquant;
807.         int ret;
808.         GetBitContext *gb= IS_INTRA(mb_type) ? h->intra_gb_ptr : h->inter_gb_ptr;
809.         const uint8_t *scan, *scan8x8;
810.         const int max_qp = 51 + 6*(h->sps.bit_depth_luma-8);
811.
812.         if(IS_INTERLACED(mb_type)){
813.             scan8x8= h->qscale ? h->field_scan8x8_cavlc : h->field_scan8x8_cavlc_q0;
814.             scan= h->qscale ? h->field_scan : h->field_scan_q0;
815.         }else{
816.             scan8x8= h->qscale ? h->zigzag_scan8x8_cavlc : h->zigzag_scan8x8_cavlc_q0;
817.             scan= h->qscale ? h->zigzag_scan : h->zigzag_scan_q0;
818.         }
819.         //QP量化参数的偏移值
820.         dquant= get_se_golomb(&h->gb);
821.         //由前一个宏块的量化参数累加得到本宏块的QP
822.         h->qscale += dquant;
823.         //注：slice中第1个宏块的计算方法（不存在前一个宏块了）：
824.         //QP = 26 + pic_init_qp_minus26 + slice_qp_delta
825.
826.         if(((unsigned)h->qscale) > max_qp){
827.             if(h->qscale<0) h->qscale+= max_qp+1;
828.             else h->qscale-= max_qp+1;
829.             if(((unsigned)h->qscale) > max_qp){
830.                 av_log(h->avctx, AV_LOG_ERROR, "dquant out of range (%d) at %d %d\n", dquant, h->mb_x, h->mb_y);
831.                 return -1;
832.             }
833.         }
834.
835.         h->chroma_qp[0]= get_chroma_qp(h, 0, h->qscale);
836.         h->chroma_qp[1]= get_chroma_qp(h, 1, h->qscale);
837.         //解码残差-亮度
838.         if( (ret = decode_luma_residual(h, gb, scan, scan8x8, pixel_shift, mb_type, cbp, 0)) < 0 ){
839.             return -1;
840.         }
841.         h->chn_table[mb_x][mb_y] |= ret << 12;

```

```

842.
843.     if (CHROMA444(h)) {
844.         //YUV444, 把U, V都当成亮度处理
845.         if( decode_luma_residual(h, gb, scan, scan8x8, pixel_shift, mb_type, cbp, 1) < 0 ){
846.             return -1;
847.         }
848.         if( decode_luma_residual(h, gb, scan, scan8x8, pixel_shift, mb_type, cbp, 2) < 0 ){
849.             return -1;
850.         }
851.     } else {
852.         //解码残差-色度
853.         const int num_c8x8 = h->sps.chroma_format_idc;
854.         //色度CBP位于高4位
855.         //0:不传
856.         //1:只传DC
857.         //2:DC+AC
858.         if(cbp&0x30){
859.             //如果传了的话
860.             //就要解码残差数据
861.             //2个分量
862.             for(chroma_idx=0; chroma_idx<2; chroma_idx++){
863.                 if (decode_residual(h, gb, h->mb + ((256 + 16*16*chroma_idx) << pixel_shift),
864.                                     CHROMA_DC_BLOCK_INDEX+chroma_idx,
865.                                     CHROMA422(h) ? chroma422_dc_scan : chroma_dc_scan,
866.                                     NULL, 4*num_c8x8) < 0) {
867.                     return -1;
868.                 }
869.             }
870.             //如果传递了AC系数
871.             if(cbp&0x20){
872.                 //2个分量
873.                 for(chroma_idx=0; chroma_idx<2; chroma_idx++){
874.                     const uint32_t *qmul = h->dequant4_coeff[chroma_idx+1+(IS_INTRA( mb_type ) ? 0:3)][h->chroma_qp[chroma_idx]];
875.                     int16_t *mb = h->mb + (16*(16 + 16*chroma_idx) << pixel_shift);
876.                     for (i8x8 = 0; i8x8<num_c8x8; i8x8++) {
877.                         for (i4x4 = 0; i4x4 < 4; i4x4++) {
878.                             const int index = 16 + 16*chroma_idx + 8*i8x8 + i4x4;
879.                             if (decode_residual(h, gb, mb, index, scan + 1, qmul, 15) < 0)
880.                                 return -1;
881.                             mb += 16 << pixel_shift;
882.                         }
883.                     }
884.                 }
885.             } else{
886.                 /*
887.                  * non_zero_count_cache:
888.                  * 每个4x4块的非0系数个数的缓存
889.                  *
890.                  * 在这里把U, V都填充为0
891.                  * non_zero_count_cache[]内容如下所示
892.                  * 图中v=0, 上面的块代表Y, 中间的块代表U, 下面的块代表V
893.                  *
894.                  *  +-+-----+
895.                  *  | 0 0 0 0 0 0 0 0
896.                  *  | 0 0 0 0 x x x x
897.                  *  | 0 0 0 0 x x x x
898.                  *  | 0 0 0 0 x x x x
899.                  *  | 0 0 0 0 x x x x
900.                  *  | 0 0 0 0 0 0 0 0
901.                  *  | 0 0 0 0 v v v v
902.                  *  | 0 0 0 0 v v v v
903.                  *  | 0 0 0 0 v v v v
904.                  *  | 0 0 0 0 v v v v
905.                  *  | 0 0 0 0 0 0 0 0
906.                  *  | 0 0 0 0 v v v v
907.                  *  | 0 0 0 0 v v v v
908.                  *  | 0 0 0 0 v v v v
909.                  *  | 0 0 0 0 v v v v
910.                  */
911.                 fill_rectangle(&h->non_zero_count_cache[scan8[16]], 4, 4, 8, 0, 1);
912.                 fill_rectangle(&h->non_zero_count_cache[scan8[32]], 4, 4, 8, 0, 1);
913.             }
914.         }
915.     } else{
916.         /*
917.          * non_zero_count_cache:
918.          * 每个4x4块的非0系数个数的缓存
919.          *
920.          * cbp为0时, 既不传DC, 也不传AC, 即全部赋值为0
921.          *
922.          * non_zero_count_cache[]内容如下所示
923.          * 图中v=0, 上面的块代表Y, 中间的块代表U, 下面的块代表V
924.          *
925.          *  +-+-----+
926.          *  | 0 0 0 0 0 0 0 0
927.          *  | 0 0 0 0 v v v v
928.          *  | 0 0 0 0 v v v v
929.          *  | 0 0 0 0 v v v v
930.          *  | 0 0 0 0 v v v v
931.          *  | 0 0 0 0 v v v v
932.          *  | 0 0 0 0 0 0 0 0

```

```

933.      * | 0 0 0 0 v v v v
934.      * | 0 0 0 0 v v v v
935.      * | 0 0 0 0 v v v v
936.      * | 0 0 0 0 v v v v
937.      * | 0 0 0 0 0 0 0 0
938.      * | 0 0 0 0 v v v v
939.      * | 0 0 0 0 v v v v
940.      * | 0 0 0 0 v v v v
941.      * | 0 0 0 0 v v v v
942.      *
943.      */
944.      fill_rectangle(&h->non_zero_count_cache[scan8[ 0]], 4, 4, 8, 0, 1);
945.      fill_rectangle(&h->non_zero_count_cache[scan8[16]], 4, 4, 8, 0, 1);
946.      fill_rectangle(&h->non_zero_count_cache[scan8[32]], 4, 4, 8, 0, 1);
947.  }
948.  //赋值QP
949.  h->cur_pic.qscale_table[mb_xy] = h->qscale;
950.
951.  //将宏块的non_zero_count_cache拷贝至整张图片的non_zero_count变量中
952.  write_back_non_zero_count(h);
953.
954.  return 0;
955. }

```

ff_h264_decode_mb_cavlc()的定义有将近1000行代码，算是一个比较复杂的函数了。我在其中写了不少注释，因此不再对源代码进行详细的分析。下面先简单梳理一下它的流程：

- (1) 解析Skip类型宏块
- (2) 获取mb_type
- (3) 填充当前宏块左边和上边宏块的信息（后面的预测中会用到）
- (4) 根据mb_type的不同，分成三种情况进行预测工作：
 - a)宏块是帧内预测
 - i.如果宏块是Intra4x4类型，则需要单独解析帧内预测模式。
 - ii.如果宏块是Intra16x16类型，则不再做过多处理。

b)宏块划分为4个块（此时每个8x8的块可以再次划分为4种类型）

这个时候每个8x8的块可以再次划分为8x8、8x4、4x8、4x4几种子块。需要分别处理这些小的子块：

- i.解析子块的参考帧序号
- ii.解析子块的运动矢量

c)其它类型（包括16x16，16x8，8x16几种划分，这些划分不可再次划分）

这个时候需要判断宏块的类型为16x16，16x8还是8x16，然后作如下处理：

- i.解析子宏块的参考帧序号
- ii.解析子宏块的运动矢量

- (5) 解码残差信息
- (6) 将宏块的各种信息输出到整个图片相应的变量中

下面简单总结一下ff_h264_decode_mb_cavlc()中涉及到的一些知识点。

mb_type

mb_type是宏块的类型的索引。FFmpeg H.264解码器中使用i_mb_type_info[]存储了I宏块的类型信息；使用p_mb_type_info[]存储了P宏块的类型信息；使用b_mb_type_info[]存储了B宏块的类型信息。使用“X_mb_type_info[mb_type]”的方式（“X”可以取“i”、“p”、“b”）可以获得该类型宏块的信息。例如获得B宏块的分块数可以使用下面这句代码。

```

1.  int partition_count= b_mb_type_info[mb_type].partition_count;

```



下面看一下这几个数组的定义。

i_mb_type_info[]

i_mb_type_info[]存储了I宏块的类型。其中的元素为IMbInfo类型的结构体。IMbInfo类型结构体的定义如下所示。

```
[cpp]    
1. typedef struct IMbInfo {  
2.     uint16_t type;  
3.     uint8_t pred_mode;//帧内预测模式  
4.     uint8_t cbp;// Coded Block Pattern, 高4位为色度, 低4位为亮度  
5. } IMbInfo;
```

i_mb_type_info[]的定义如下。

```
[cpp]    
1. //I宏块的mb_type  
2. /*  
3.  * 规律：  
4.  * pred_mode总是Vertical->Horizontal->DC->Plane(记住帧内预测中Vertical排在第0个)  
5.  * cbp:传送数据量越来越大(前半部分不传亮度残差)  
6.  * 按照数据量排序  
7.  *  
8.  * 只有Intra_16x16宏块类型, CBP的值不是由语法元素给出, 而是通过mb_type得到。  
9.  *  
10. * CBP(Coded Block Pattern)  
11. * 色度CBP含义：  
12. * 0:不传残差  
13. * 1:只传DC  
14. * 2:传送DC+AC  
15. * 亮度CBP(只有最低4位有定义)含义：  
16. * 变量的最低位比特从最低位开始, 每一位对应一个子宏块, 该位等于1时表明对应子宏块残差系数被传送；该位等于0  
17. * 时表明对应子宏块残差全部不被传送, 解码器把这些残差系数赋为0。  
18. */  
19.  
20. static const IMbInfo i_mb_type_info[26] = {  
21.     { MB_TYPE_INTRA4x4,  -1,  -1 },//pred_mode还需要单独获取  
22.     { MB_TYPE_INTRA16x16, 2,   0 },//cbp:0000+0  
23.     { MB_TYPE_INTRA16x16, 1,   0 },  
24.     { MB_TYPE_INTRA16x16, 0,   0 },  
25.     { MB_TYPE_INTRA16x16, 3,   0 },  
26.     { MB_TYPE_INTRA16x16, 2,  16 },//cbp:0000+1<<4  
27.     { MB_TYPE_INTRA16x16, 1,  16 },  
28.     { MB_TYPE_INTRA16x16, 0,  16 },  
29.     { MB_TYPE_INTRA16x16, 3,  16 },  
30.     { MB_TYPE_INTRA16x16, 2,  32 },//cbp:0000+2<<4  
31.     { MB_TYPE_INTRA16x16, 1,  32 },  
32.     { MB_TYPE_INTRA16x16, 0,  32 },  
33.     { MB_TYPE_INTRA16x16, 3,  32 },  
34.     { MB_TYPE_INTRA16x16, 2,  15 + 0 },//cbp:1111+0<<4  
35.     { MB_TYPE_INTRA16x16, 1,  15 + 0 },  
36.     { MB_TYPE_INTRA16x16, 0,  15 + 0 },  
37.     { MB_TYPE_INTRA16x16, 3,  15 + 0 },  
38.     { MB_TYPE_INTRA16x16, 2,  15 + 16 },//cbp:1111+1<<4  
39.     { MB_TYPE_INTRA16x16, 1,  15 + 16 },  
40.     { MB_TYPE_INTRA16x16, 0,  15 + 16 },  
41.     { MB_TYPE_INTRA16x16, 3,  15 + 16 },  
42.     { MB_TYPE_INTRA16x16, 2,  15 + 32 },//cbp:1111+2<<4  
43.     { MB_TYPE_INTRA16x16, 1,  15 + 32 },  
44.     { MB_TYPE_INTRA16x16, 0,  15 + 32 },  
45.     { MB_TYPE_INTRA16x16, 3,  15 + 32 },  
46.     { MB_TYPE_INTRA_PCM,  -1,  -1 },//特殊  
47. };
```

p_mb_type_info[]

p_mb_type_info[]存储了P宏块的类型。其中的元素为PMbInfo类型的结构体。PMbInfo类型结构体的定义如下所示。

```
[cpp]    
1. typedef struct PMbInfo {  
2.     uint16_t type;//宏块类型  
3.     uint8_t partition_count;//分块数量  
4. } PMbInfo;
```

p_mb_type_info[]的定义如下。


```
[cpp]
1. //P宏块的mb_type
2. /*
3.  * 规律：
4.  * 宏块划分尺寸从大到小（子宏块数量逐渐增多）
5.  * 先是“胖”（16x8）的，再是“瘦”（8x16）的
6.  * MB_TYPE_PXL0中的“X”代表宏块的第几个分区，只能取0或者1
7.  * MB_TYPE_P0LX中的“X”代表宏块参考的哪个List。P宏块只能参考list0
8.  *
9.  */
10. static const PMbInfo p_mb_type_info[5] = {
11.     { MB_TYPE_16x16 | MB_TYPE_P0L0, 1 }, //没有“P1”
12.     { MB_TYPE_16x8 | MB_TYPE_P0L0 | MB_TYPE_P1L0, 2 },
13.     { MB_TYPE_8x16 | MB_TYPE_P0L0 | MB_TYPE_P1L0, 2 },
14.     { MB_TYPE_8x8 | MB_TYPE_P0L0 | MB_TYPE_P1L0, 4 },
15.     { MB_TYPE_8x8 | MB_TYPE_P0L0 | MB_TYPE_P1L0 | MB_TYPE_REF0, 4 },
16. };
```

b_mb_type_info[]

b_mb_type_info[]存储了B宏块的类型。其中的元素为PMbInfo类型的结构体。在这里需要注意，p_mb_type_info[]和b_mb_type_info[]中的元素的类型是一样的，都是PMbInfo类型的结构体。

b_mb_type_info[]的定义如下。

```
[cpp]
1. //B宏块的mb_type
2. /*
3.  * 规律：
4.  * 宏块划分尺寸从大到小（子宏块数量逐渐增多）
5.  * 先是“胖”（16x8）的，再是“瘦”（8x16）的
6.  * 每个分区参考的list越来越多（意见越来越不一致了）
7.  *
8.  * MB_TYPE_PXL0中的“X”代表宏块的第几个分区，只能取0或者1
9.  * MB_TYPE_P0LX中的“X”代表宏块参考的哪个List。B宏块参考list0和list1
10.  *
11.  */
12. static const PMbInfo b_mb_type_info[23] = {
13.     { MB_TYPE_DIRECT2 | MB_TYPE_L0L1, 1, },
14.     { MB_TYPE_16x16 | MB_TYPE_P0L0, 1, }, //没有“P1”
15.     { MB_TYPE_16x16 | MB_TYPE_P0L1, 1, },
16.     { MB_TYPE_16x16 | MB_TYPE_P0L0 | MB_TYPE_P0L1, 1, },
17.     { MB_TYPE_16x8 | MB_TYPE_P0L0 | MB_TYPE_P1L0, 2, }, //两个分区（每个分区两个参考帧）都参考list0
18.     { MB_TYPE_8x16 | MB_TYPE_P0L0 | MB_TYPE_P1L0, 2, },
19.     { MB_TYPE_16x8 | MB_TYPE_P0L1 | MB_TYPE_P1L1, 2, }, //两个分区（每个分区两个参考帧）都参考list1
20.     { MB_TYPE_8x16 | MB_TYPE_P0L1 | MB_TYPE_P1L1, 2, },
21.     { MB_TYPE_16x8 | MB_TYPE_P0L0 | MB_TYPE_P1L1, 2, }, //0分区（两个参考帧）参考list0,1分区（两个参考帧）
    参考list1
22.     { MB_TYPE_8x16 | MB_TYPE_P0L0 | MB_TYPE_P1L1, 2, },
23.     { MB_TYPE_16x8 | MB_TYPE_P0L1 | MB_TYPE_P1L0, 2, },
24.     { MB_TYPE_8x16 | MB_TYPE_P0L1 | MB_TYPE_P1L0, 2, },
25.     { MB_TYPE_16x8 | MB_TYPE_P0L0 | MB_TYPE_P1L0 | MB_TYPE_P1L1, 2, },
26.     { MB_TYPE_8x16 | MB_TYPE_P0L0 | MB_TYPE_P1L0 | MB_TYPE_P1L1, 2, },
27.     { MB_TYPE_16x8 | MB_TYPE_P0L1 | MB_TYPE_P1L0 | MB_TYPE_P1L1, 2, },
28.     { MB_TYPE_8x16 | MB_TYPE_P0L1 | MB_TYPE_P1L0 | MB_TYPE_P1L1, 2, },
29.     { MB_TYPE_16x8 | MB_TYPE_P0L0 | MB_TYPE_P0L1 | MB_TYPE_P1L0, 2, },
30.     { MB_TYPE_8x16 | MB_TYPE_P0L0 | MB_TYPE_P0L1 | MB_TYPE_P1L0, 2, },
31.     { MB_TYPE_16x8 | MB_TYPE_P0L0 | MB_TYPE_P0L1 | MB_TYPE_P1L1, 2, },
32.     { MB_TYPE_8x16 | MB_TYPE_P0L0 | MB_TYPE_P0L1 | MB_TYPE_P1L1, 2, },
33.     { MB_TYPE_16x8 | MB_TYPE_P0L0 | MB_TYPE_P0L1 | MB_TYPE_P1L0 | MB_TYPE_P1L1, 2, },
34.     { MB_TYPE_8x16 | MB_TYPE_P0L0 | MB_TYPE_P0L1 | MB_TYPE_P1L0 | MB_TYPE_P1L1, 2, },
35.     { MB_TYPE_8x8 | MB_TYPE_P0L0 | MB_TYPE_P0L1 | MB_TYPE_P1L0 | MB_TYPE_P1L1, 4, },
36. };
```

填充当前宏块左边和上边宏块的信息

在宏块预测的时候需要用到当前宏块左边、上左、上边，上右位置的宏块有关的信息。因此在预测前需要先填充这些信息。H.264解码器中调用了fill_decode_neighbors()和fill_decode_caches()两个函数填充这些信息。fill_decode_caches()函数我目前还没有仔细看，在这里简单分析一下fill_decode_neighbors()函数

fill_decode_neighbors()

fill_decode_neighbors()用于设置当前宏块左边、上左、上边，上右位置的宏块的索引值和宏块类型，定义位于libavcodec\h264_mvpred.h，如下所示。

```
[cpp]
1. /* 设置上左，上，上右，左宏块的索引值和宏块类型
2.  * 这4个宏块在解码过程中会用到
3.  * 位置如下图所示
4.  *
5.  * +-----+-----+
6.  * | UL | U | UR |
```

```

7.  * +---+---+---+
8.  * | L |   |
9.  * +---+---+
10. */
11. static void fill_decode_neighbors(H264Context *h, int mb_type)
12. {
13.     const int mb_xy = h->mb_xy;
14.     int topleft_xy, top_xy, topright_xy, left_xy[LEFT_MBS];
15.     static const uint8_t left_block_options[4][32] = {
16.         { 0, 1, 2, 3, 7, 10, 8, 11, 3 + 0 * 4, 3 + 1 * 4, 3 + 2 * 4, 3 + 3 * 4, 1 + 4 * 4, 1 + 8 * 4, 1 + 5 * 4, 1 + 9 * 4 },
17.         { 2, 2, 3, 3, 8, 11, 8, 11, 3 + 2 * 4, 3 + 2 * 4, 3 + 3 * 4, 3 + 3 * 4, 1 + 5 * 4, 1 + 5 * 4, 1 + 9 * 4, 1 + 9 * 4 },
18.         { 0, 0, 1, 1, 7, 10, 7, 10, 3 + 0 * 4, 3 + 0 * 4, 3 + 1 * 4, 3 + 1 * 4, 1 + 4 * 4, 1 + 8 * 4, 1 + 4 * 4, 1 + 8 * 4 },
19.         { 0, 2, 0, 2, 7, 10, 7, 10, 3 + 0 * 4, 3 + 2 * 4, 3 + 0 * 4, 3 + 2 * 4, 1 + 4 * 4, 1 + 8 * 4, 1 + 4 * 4, 1 + 8 * 4 }
20.     };
21.
22.     h->topleft_partition = -1;
23.     //上方宏块。当前宏块减去一行
24.     //top_xy=mb_xy-mb_stride
25.     top_xy = mb_xy - (h->mb_stride << MB_FIELD(h));
26.
27.     /* Wow, what a mess, why didn't they simplify the interlacing & intra
28.      * stuff, I can't imagine that these complex rules are worth it. */
29.     //左上宏块。上方宏块减1
30.     topleft_xy = top_xy - 1;
31.     //上右宏块。上方宏块加1
32.     topright_xy = top_xy + 1;
33.     //左边宏块。当前宏块减1
34.     left_xy[LBOT] = left_xy[LTOP] = mb_xy - 1;
35.     h->left_block = left_block_options[0];
36.
37.     if (FRAME_MBAFF(h)) {
38.         const int left_mb_field_flag = IS_INTERLACED(h->cur_pic.mb_type[mb_xy - 1]);
39.         const int curr_mb_field_flag = IS_INTERLACED(mb_type);
40.         if (h->mb_y & 1) {
41.             if (left_mb_field_flag != curr_mb_field_flag) {
42.                 left_xy[LBOT] = left_xy[LTOP] = mb_xy - h->mb_stride - 1;
43.                 if (curr_mb_field_flag) {
44.                     left_xy[LBOT] += h->mb_stride;
45.                     h->left_block = left_block_options[3];
46.                 } else {
47.                     topleft_xy += h->mb_stride;
48.                     /* take top left mv from the middle of the mb, as opposed
49.                      * to all other modes which use the bottom right partition */
50.                     h->topleft_partition = 0;
51.                     h->left_block = left_block_options[1];
52.                 }
53.             }
54.         } else {
55.             if (curr_mb_field_flag) {
56.                 topleft_xy += h->mb_stride & (((h->cur_pic.mb_type[top_xy - 1] >> 7) & 1) - 1);
57.                 topright_xy += h->mb_stride & (((h->cur_pic.mb_type[top_xy + 1] >> 7) & 1) - 1);
58.                 top_xy += h->mb_stride & (((h->cur_pic.mb_type[top_xy] >> 7) & 1) - 1);
59.             }
60.             if (left_mb_field_flag != curr_mb_field_flag) {
61.                 if (curr_mb_field_flag) {
62.                     left_xy[LBOT] += h->mb_stride;
63.                     h->left_block = left_block_options[3];
64.                 } else {
65.                     h->left_block = left_block_options[2];
66.                 }
67.             }
68.         }
69.     }
70.     //宏块索引值
71.     //左上
72.     h->topleft_mb_xy = topleft_xy;
73.     //上
74.     h->top_mb_xy = top_xy;
75.     //上右
76.     h->topright_mb_xy = topright_xy;
77.     //左。逐行扫描时候left_xy[LTOP]=left_xy[LBOT]
78.     h->left_mb_xy[LTOP] = left_xy[LTOP];
79.     h->left_mb_xy[LBOT] = left_xy[LBOT];
80.     //FIXME do we need all in the context?
81.
82.     //宏块类型
83.     h->topleft_type = h->cur_pic.mb_type[topleft_xy];
84.     h->top_type = h->cur_pic.mb_type[top_xy];
85.     h->topright_type = h->cur_pic.mb_type[topright_xy];
86.     h->left_type[LTOP] = h->cur_pic.mb_type[left_xy[LTOP]];
87.     h->left_type[LBOT] = h->cur_pic.mb_type[left_xy[LBOT]];
88.
89.     if (FMO) {
90.         if (h->slice_table[topleft_xy] != h->slice_num)
91.             h->topleft_type = 0;
92.         if (h->slice_table[top_xy] != h->slice_num)
93.             h->top_type = 0;
94.         if (h->slice_table[left_xy[LTOP]] != h->slice_num)
95.             h->left_type[LTOP] = h->left_type[LBOT] = 0;
96.     } else {
97.         if (h->slice_table[topleft_xy] != h->slice_num) {

```

```

98.         n->topleft_type = 0;
99.         if (h->slice_table[top_xy] != h->slice_num)
100.             h->top_type = 0;
101.         if (h->slice_table[left_xy[LTOP]] != h->slice_num)
102.             h->left_type[LTOP] = h->left_type[LBOT] = 0;
103.     }
104. }
105. if (h->slice_table[topright_xy] != h->slice_num)
106.     h->topright_type = 0;
107. }

```

从源代码中可以看出，fill_decode_neighbors()设置了下面几个索引值：

toleft_mb_xy, top_mb_xy, topright_mb_xy, left_mb_xy[LTOP]和left_mb_xy[LBOT]

设置了下面几个宏块的类型：

toleft_type, top_type, topright_type, left_type[LTOP], left_type[LBOT]

需要注意的是，在逐行扫面的情况下left_xy[LTOP]和left_xy[LBOT]是相等的。

各种Cache（缓存）

在H.264解码器中包含了各种各样的Cache（缓存）。例如：

intra4x4_pred_mode_cache：Intra4x4帧内预测模式的缓存

non_zero_count_cache：每个4x4块的非0系数个数的缓存

mv_cache：运动矢量缓存

ref_cache：运动矢量参考帧的缓存

这几个Cache的定义如下所示。

```

1.  /**
2.   * Intra4x4帧内预测模式的缓存
3.   * 结构如下所示
4.   * |
5.   * --+-----
6.   * | 0 0 0 0 0 0 0 0
7.   * | 0 0 0 0 Y Y Y Y
8.   * | 0 0 0 0 Y Y Y Y
9.   * | 0 0 0 0 Y Y Y Y
10.  * | 0 0 0 0 Y Y Y Y
11.  */
12.  int8_t intra4x4_pred_mode_cache[5 * 8];
13.
14.  /**
15.   * non zero coeff count cache.
16.   * is 64 if not available.
17.   * 每个4x4块的非0系数个数的缓存
18.   */
19.  uint8_t __attribute__((aligned (8))) non_zero_count_cache[15 * 8];
20.
21.  /**
22.   * Motion vector cache.
23.   * 运动矢量缓存[list][data][x,y]
24.   * list:L0或者L1
25.   * data:共5x8个元素（注意是int16_t类型）
26.   * 结构如下所示
27.   * |
28.   * --+-----
29.   * | 0 0 0 0 0 0 0 0
30.   * | 0 0 0 0 Y Y Y Y
31.   * | 0 0 0 0 Y Y Y Y
32.   * | 0 0 0 0 Y Y Y Y
33.   * | 0 0 0 0 Y Y Y Y
34.   * x,y：运动矢量的横坐标和纵坐标
35.   */
36.  int16_t __attribute__((aligned (16))) mv_cache[2][5 * 8][2];
37.
38.  /**
39.   * 运动矢量参考帧的缓存，与mv_cache配合使用（注意数据是int8_t类型）
40.   * 结构如下所示
41.   * |
42.   * --+-----
43.   * | 0 0 0 0 0 0 0 0
44.   * | 0 0 0 0 Y Y Y Y
45.   * | 0 0 0 0 Y Y Y Y
46.   * | 0 0 0 0 Y Y Y Y
47.   * | 0 0 0 0 Y Y Y Y
48.   */
49.  int8_t __attribute__((aligned (8))) ref_cache [2][5 * 8];

```

通过观察上面的定义，我们会发现Cache都是一个包含x*8个元素的一维数组（x取15或者5）。按照我自己的理解，我觉得Cache使用一维数组比较形象的存储了二维图像的信息。从上面的代码可以看出Cache中存储有效数据的地方是一个位于右下角的“方形区域”，这一部分实际上对应一维数组中第12-15，20-23，28-31，36-39的元素。这个“方形区域”代表了一个宏块的亮度相关的信息，其中一共包含16个元素。由于1个宏块的亮度数据是1个16x16的块，所以这个“方形区域”里面1个元素实际上代表了一个4x4的块的信息（“4x4”的亮度块应该也是H.264压缩编码中最小的处理单元）。

如果我们使用12-15，20-23，28-31，36-39这些范围内的下标引用Cache中的元素，确实是不太方便。由此也引出了FFmpeg H.264解码器中另一个关键的变量——scan8[]数组。

scan8[]

scan8[]存储的是缓存的序号值，它一般情况下是与前面提到的Cache配合使用的。scan8[]的定义位于libavcodec/h264.h，如下所示。

```

1.  /*
2.   * 扫描方式：
3.   * 0-0 0-0
4.   * / / /
5.   * 0-0 0-0
6.   * ,--'
7.   * 0-0 0-0
8.   * / / /
9.   * 0-0 0-0
10.  */
11.
12.  /* Scan8 organization:
13.   *   0 1 2 3 4 5 6 7
14.   * 0  DY   y y y y y
15.   * 1       y Y Y Y Y
16.   * 2       y Y Y Y Y
17.   * 3       y Y Y Y Y
18.   * 4       y Y Y Y Y
19.   * 5  DU   u u u u u
20.   * 6       u U U U U
21.   * 7       u U U U U
22.   * 8       u U U U U
23.   * 9       u U U U U
24.   * 10 DV  v v v v v
25.   * 11      v V V V V
26.   * 12      v V V V V
27.   * 13      v V V V V
28.   * 14      v V V V V
29.   * DY/DU/DV are for luma/chroma DC.
30.  */
31.
32.  // This table must be here because scan8[constant] must be known at compiletime
33.  //scan8[]通常与各种cache配合使用(mv_cache,ref_cache等)
34.  static const uint8_t scan8[16 * 3 + 3] = {
35.      4 + 1 * 8, 5 + 1 * 8, 4 + 2 * 8, 5 + 2 * 8,
36.      6 + 1 * 8, 7 + 1 * 8, 6 + 2 * 8, 7 + 2 * 8,
37.      4 + 3 * 8, 5 + 3 * 8, 4 + 4 * 8, 5 + 4 * 8,
38.      6 + 3 * 8, 7 + 3 * 8, 6 + 4 * 8, 7 + 4 * 8,
39.      4 + 6 * 8, 5 + 6 * 8, 4 + 7 * 8, 5 + 7 * 8,
40.      6 + 6 * 8, 7 + 6 * 8, 6 + 7 * 8, 7 + 7 * 8,
41.      4 + 8 * 8, 5 + 8 * 8, 4 + 9 * 8, 5 + 9 * 8,
42.      6 + 8 * 8, 7 + 8 * 8, 6 + 9 * 8, 7 + 9 * 8,
43.      4 + 11 * 8, 5 + 11 * 8, 4 + 12 * 8, 5 + 12 * 8,
44.      6 + 11 * 8, 7 + 11 * 8, 6 + 12 * 8, 7 + 12 * 8,
45.      4 + 13 * 8, 5 + 13 * 8, 4 + 14 * 8, 5 + 14 * 8,
46.      6 + 13 * 8, 7 + 13 * 8, 6 + 14 * 8, 7 + 14 * 8,
47.      0 + 0 * 8, 0 + 5 * 8, 0 + 10 * 8
48.  };

```

可以看出scan8[]数组中元素的值都是以“a+b*8”的形式写的，我们不妨计算一下前面16个元素的值：

scan8[0]=12

scan8[1]= 13

scan8[2]= 20

scan8[3]= 21

scan8[4]= 14

scan8[5]= 15

scan8[6]= 22

scan8[7]= 23

scan8[8]= 28

scan8[9]= 29

scan8[10]= 36

scan8[11]= 37

scan8[12]= 30

scan8[13]= 31

scan8[14]= 38

scan8[15]= 39

如果把scan8[]数组这些元素的值，作为Cache（例如mv_cache，ref_cache等）的序号，会发现他们的在Cache中代表的元素的位置如下图所示。



上图中灰色背景的元素即为Cache中有效的元素（不使用左边的空白区域的元素可能是由于历史原因）。直接使用Cache元素序号可能感觉比较抽象，下图使用scan8[]数组元素序号表示Cache中存储的数据，则结果如下图所示。



图中每个元素代表了一个4x4的块的信息，每个由16个元素组成的“大方块”代表了1个宏块的1个分量的信息。灰色背景的“大方块”存储的是宏块中亮度Y相关的信息，蓝色背景的“大方块”存储的是宏块中色度U相关的信息，粉红背景的“大方块”存储的是宏块中色度V相关的信息。

PS：有关scan8[]数组在网上能查到一点资料。但是经过源代码比对之后，我发现网上的资料已经过时了。旧版本scan8[]代表的Cache的存储方式如下所示。



可以看出旧版本的scan8[]中U、V是存储在Y的左边的区域，而且每个分量只有4个元素，而新版本的scan8[]中U、V是存储在Y的下边的区域，而且每个分量有16个元素。

推测Intra4x4帧内预测模式

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。H.264解码器中有关这部分功能的实现代码位于pred_intra_mode()函数中，如下所示。

```
[cpp]
1.  /**
2.   * Get the predicted intra4x4 prediction mode.
3.   */
4.   //获得对Intra4x4的预测模式的预测值（挺绕口，确实是这样）
5.   //这个预测模式由左边和上边块的预测模式（取最小值）推导主来
6.   static av_always_inline int pred_intra_mode(H264Context *h, int n)
7.   {
8.       const int index8 = scan8[n];
9.       //左边块的预测方式
10.      const int left  = h->intra4x4_pred_mode_cache[index8 - 1];
11.      //上边块的预测方式
12.      const int top   = h->intra4x4_pred_mode_cache[index8 - 8];
13.      //获得左边和上边的最小值
14.      const int min   = FFMIN(left, top);
15.
16.      tprintf(h->avctx, "mode:%d %d min:%d\n", left, top, min);
17.      //返回
18.      if (min < 0)
19.          return DC_PRED;
20.      else
21.          return min;
22.  }
```

参考帧序号和运动矢量的获取

无论处理哪种类型的宏块，H.264解码器都是首先获得宏块的参考帧序号，然后获得宏块的运动矢量。获取参考帧序号和运动矢量的代码占用了ff_h264_decode_mb_cavlc()最大的篇幅。在这里我们看一段最简单的例子——帧间16x16宏块参考帧序号和运动矢量获取。该部分的代码如下所示。

```

1.  if(IS_16X16(mb_type)){
2.      /*
3.       * 16x16 宏块
4.       *
5.       * +-----+-----+
6.       * |               |
7.       * |               |
8.       * |               |
9.       * |   +       +   |
10.      * |               |
11.      * |               |
12.      * |               |
13.      * +-----+-----+
14.      *
15.      */
16.      //运动矢量对应的参考帧
17.      //L0和L1
18.      for(list=0; list<h->list_count; list++){
19.          unsigned int val;
20.          if(IS_DIR(mb_type, 0, list)){
21.              if(local_ref_count[list]==1){
22.                  val= 0;
23.              } else if(local_ref_count[list]==2){
24.                  val= get_bits1(&h->gb)^1;
25.              }else{
26.                  //参考帧图像序号
27.                  val= get_ue_golomb_31(&h->gb);
28.                  if (val >= local_ref_count[list]){
29.                      av_log(h->avctx, AV_LOG_ERROR, "ref %u overflow\n", val);
30.                      return -1;
31.                  }
32.              }
33.              //填充ref_cache
34.              //fill_rectangle(数据起始点, 宽, 高, 一行数据个数, 数据值, 每个数据占用的byte)
35.              //scan8[0]代表了cache里面亮度Y的起始点
36.              /*
37.               * 在这里相当于在ref_cache[list]填充了这样的一份数据 (val=v) :
38.               * |
39.               * +-----+
40.               * | 0 0 0 0 0 0 0 0
41.               * | 0 0 0 0 v v v v
42.               * | 0 0 0 0 v v v v
43.               * | 0 0 0 0 v v v v
44.               * | 0 0 0 0 v v v v
45.               */
46.              fill_rectangle(&h->ref_cache[list][ scan8[0] ], 4, 4, 8, val, 1);
47.          }
48.      }
49.      //运动矢量
50.      for(list=0; list<h->list_count; list++){
51.          if(IS_DIR(mb_type, 0, list)){
52.              //预测MV (取中值)
53.              pred_motion(h, 0, 4, list, h->ref_cache[list][ scan8[0] ], &mx, &my);
54.              //MVD从码流中获取
55.              //MV=预测MV+MVD
56.              mx += get_se_golomb(&h->gb);
57.              my += get_se_golomb(&h->gb);
58.              tprintf(h->avctx, "final mv:%d %d\n", mx, my);
59.              //填充mv_cache
60.              //fill_rectangle(数据起始点, 宽, 高, 一行数据个数, 数据值, 每个数据占用的byte)
61.              //scan8[0]代表了cache里面亮度Y的起始点
62.              /*
63.               * 在这里相当于在mv_cache[list]填充了这样的一份数据 (val=v) :
64.               * |
65.               * +-----+
66.               * | 0 0 0 0 0 0 0 0
67.               * | 0 0 0 0 v v v v
68.               * | 0 0 0 0 v v v v
69.               * | 0 0 0 0 v v v v
70.               * | 0 0 0 0 v v v v
71.               */
72.              fill_rectangle(h->mv_cache[list][ scan8[0] ], 4, 4, 8, pack16to32(mx,my), 4);
73.          }
74.      }
75.  }

```

从代码中可以看出，H.264解码器首先读取了参考帧图像序号（val变量）并且存入了ref_cache缓存中，然后读取了运动矢量（mx，my变量）并且存入了mv_cache缓存中。在读取运动矢量的时候，有一点需要注意：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。因此一个宏块真正的运动矢量应该使用下式计算：

$$\text{MV} = \text{预测MV} + \text{MVD}$$

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）。例如下图中，E的运动矢量的预测值就取自于A、B、C三个块MV的中值。

在FFmpeg H.264解码器中，运动矢量预测部分的代码在pred_motion()函数中实现。该函数定义位于libavcodec/h264_mvpred.h，如下所示。

```

1.  /**
2.   * Get the predicted MV.
3.   * @param n the block index
4.   * @param part_width the width of the partition (4, 8,16) -> (1, 2, 4)
5.   * @param mx the x component of the predicted motion vector
6.   * @param my the y component of the predicted motion vector
7.   */
8.   //获取预测MV（取中值），结果存入mx, my
9.   static av_always_inline void pred_motion(H264Context *const h, int n,
10.                                           int part_width, int list, int ref,
11.                                           int *const mx, int *const my)
12.   {
13.       const int index8      = scan8[n];
14.       const int top_ref      = h->ref_cache[list][index8 - 8];
15.       const int left_ref     = h->ref_cache[list][index8 - 1];
16.       //左侧MV
17.       const int16_t *const A = h->mv_cache[list][index8 - 1];
18.       //上方MV
19.       const int16_t *const B = h->mv_cache[list][index8 - 8];
20.       //右上MV?
21.       const int16_t *C;
22.       int diagonal_ref, match_count;
23.
24.       av_assert2(part_width == 1 || part_width == 2 || part_width == 4);
25.
26.       /* mv_cache
27.        * B . . A T T T T
28.        * U . . L . . . .
29.        * U . . L . . . .
30.        * U . . L . . . .
31.        * . . . L . . . .
32.        */
33.
34.       diagonal_ref = fetch_diagonal_mv(h, &C, index8, list, part_width);
35.       match_count = (diagonal_ref == ref) + (top_ref == ref) + (left_ref == ref);
36.       tprintf(h->avctx, "pred_motion match_count=%d\n", match_count);
37.       if (match_count > 1) { //most common
38.           //取A,B,C中值
39.           *mx = mid_pred(A[0], B[0], C[0]);
40.           *my = mid_pred(A[1], B[1], C[1]);
41.       } else if (match_count == 1) {
42.           //只取其中的一个值
43.           if (left_ref == ref) {
44.               *mx = A[0];
45.               *my = A[1];
46.           } else if (top_ref == ref) {
47.               *mx = B[0];
48.               *my = B[1];
49.           } else {
50.               *mx = C[0];
51.               *my = C[1];
52.           }
53.       } else {
54.           if (top_ref == PART_NOT_AVAILABLE &&
55.               diagonal_ref == PART_NOT_AVAILABLE &&
56.               left_ref != PART_NOT_AVAILABLE) {
57.               *mx = A[0];
58.               *my = A[1];
59.           } else {
60.               *mx = mid_pred(A[0], B[0], C[0]);
61.               *my = mid_pred(A[1], B[1], C[1]);
62.           }
63.       }
64.
65.       tprintf(h->avctx,
66.           "pred_motion (%2d %2d %2d) (%2d %2d %2d) (%2d %2d %2d) -> (%2d %2d %2d) at %2d %2d %d list %d\n",
67.           top_ref, B[0], B[1], diagonal_ref, C[0], C[1], left_ref,
68.           A[0], A[1], ref, *mx, *my, h->mb_x, h->mb_y, n, list);
69.   }

```

解码残差

H.264解码器首先判断CBP是否为0。如果CBP不为0，则解码CAVLC编码的残差数据；如果CBP为0，则直接将non_zero_count_cache[]全部赋值为0。

CBP

CBP全称为Coded Block Pattern，指亮度和色度分量的各小块的残差的编码方案。H.264解码器中cbp变量（一个uint8_t类型变量）高4位存储了色度CBP，低4位存储了亮度CBP。色度CBP和亮度CBP的含义是不一样的：

亮度CBP数据从最低位开始，每1位对应1个子宏块，该位等于1时表明对应子宏块残差系数被传送。（因此亮度CBP数据通常需要当成二进制数据来看）

色度CBP包含3种取值：

0：代表所有残差都不被传送

1：只传送DC系数

2：传送DC系数以及AC系数

（因此色度CBP数据通常可以当成十进制数据来看）

decode_luma_residual()

当CBP不为0的时候，会调用decode_luma_residual()解码亮度残差数据。此外如果包含色度残差的话，还会调用decode_residual()解码色度残差数据。decode_luma_residual()的定义如下所示。

```

1. //解码残差-亮度
2. static av_always_inline int decode_luma_residual(H264Context *h, GetBitContext *gb, const uint8_t *scan, const uint8_t *scan8x8, int
   pixel_shift, int mb_type, int cbp, int p){
3.     int i4x4, i8x8;
4.     int qscale = p == 0 ? h->qscale : h->chroma_qp[p-1];
5.     if(IS_INTRA16x16(mb_type)){
6.         //Intra16x16类型
7.         AV_ZERO128(h->mb_luma_dc[p]+0);
8.         AV_ZERO128(h->mb_luma_dc[p]+8);
9.         AV_ZERO128(h->mb_luma_dc[p]+16);
10.        AV_ZERO128(h->mb_luma_dc[p]+24);
11.        //解码残差
12.        //在这里是解码Hadamard变换后的系数?
13.        if( decode_residual(h, h->intra_gb_ptr, h->mb_luma_dc[p], LUMA_DC_BLOCK_INDEX+p, scan, NULL, 16) < 0){
14.            return -1; //FIXME continue if partitioned and other return -1 too
15.        }
16.
17.        av_assert2((cbp&15) == 0 || (cbp&15) == 15);
18.
19.        //cbp=15=1111
20.        if(cbp&15){
21.            //如果子宏块亮度残差全都编码了
22.            for(i8x8=0; i8x8<4; i8x8++){
23.                for(i4x4=0; i4x4<4; i4x4++){
24.                    //循环16次
25.                    const int index= i4x4 + 4*i8x8 + p*16;
26.                    if( decode_residual(h, h->intra_gb_ptr, h->mb + (16*index << pixel_shift),
27.                        index, scan + 1, h->dequant4_coeff[p][qscale], 15) < 0 ){
28.                        return -1;
29.                    }
30.                }
31.            }
32.            return 0xf;
33.        }else{
34.            //如果子宏块亮度残差没有编码
35.            //就把non_zero_count_cache亮度部分全部填上0
36.            fill_rectangle(&h->non_zero_count_cache[scan8[p*16]], 4, 4, 8, 0, 1);
37.            return 0;
38.        }
39.    }else{
40.        int cqm = (IS_INTRA( mb_type ) ? 0:3)+p;
41.        /* For CAVLC 4:4:4, we need to keep track of the luma 8x8 CBP for deblocking nnz purposes. */
42.        int new_cbp = 0;
43.        for(i8x8=0; i8x8<4; i8x8++){
44.            if(cbp & (1<<i8x8)){
45.                if(IS_8x8DCT(mb_type)){
46.                    int16_t *buf = &h->mb[64*i8x8+256*p << pixel_shift];
47.                    uint8_t *nnz;
48.                    for(i4x4=0; i4x4<4; i4x4++){
49.                        const int index= i4x4 + 4*i8x8 + p*16;
50.                        if( decode_residual(h, gb, buf, index, scan8x8+16*i4x4,
51.                            h->dequant8_coeff[cqm][qscale], 16) < 0 )
52.                            return -1;
53.                    }
54.                    nnz= &h->non_zero_count_cache[ scan8[4*i8x8+p*16] ];
55.                    nnz[0] += nnz[1] + nnz[8] + nnz[9];
56.                    new_cbp |= !nnz[0] << i8x8;
57.                }else{
58.                    for(i4x4=0; i4x4<4; i4x4++){
59.                        const int index= i4x4 + 4*i8x8 + p*16;
60.                        //解码残差
61.                        if( decode_residual(h, gb, h->mb + (16*index << pixel_shift), index,
62.                            scan, h->dequant4_coeff[cqm][qscale], 16) < 0 ){
63.                            return -1;
64.                        }
65.                        new_cbp |= h->non_zero_count_cache[ scan8[index] ] << i8x8;
66.                    }
67.                }
68.            }else{
69.                uint8_t * const nnz= &h->non_zero_count_cache[ scan8[4*i8x8+p*16] ];
70.                nnz[0] = nnz[1] = nnz[8] = nnz[9] = 0;
71.            }
72.        }
73.        return new_cbp;
74.    }
75. }

```

从源代码可以看出，decode_luma_residual()内部实际上也是调用了decode_residual()解码残差数据。decode_residual()内部则调用了get_vlc2()解析CAVLC数据。由于decode_residual()内部还没有仔细看，所以暂时不进行详细分析。

宏块的各种信息输出到整个图片相应的内存中

ff_h264_decode_mb_cavlc()中包含了很多名称为write_back_{XXX}()的函数。这些函数用于将Cache中当前宏块的信息拷贝至整张图片的相应的变量中。例如如下几个函数：

write_back_intra_pred_mode(): 将intra4x4_pred_mode_cache中的数据拷贝至intra4x4_pred_mode。

write_back_motion(): 将mv_cache中的数据拷贝至cur_pic结构体中的motion_val; 然后将ref_cache中的数据拷贝至cur_pic结构体中的ref_index。

write_back_non_zero_count(): 将non_zero_count_cache中的数据拷贝至non_zero_count。

在这里我们选择write_back_motion()看看它的源代码。

write_back_motion()

write_back_motion()可以将宏块的Cache中的MV拷贝至整张图片的motion_val变量中。

```
[cpp]
1. //将宏块的Cache中的MV拷贝至整张图片的motion_val变量中
2. static av_always_inline void write_back_motion(H264Context *h, int mb_type)
3. {
4.     const int b_stride = h->b_stride;
5.     const int b_xy = 4 * h->mb_x + 4 * h->mb_y * h->b_stride; // try mb2b(8)_xy
6.     const int b8_xy = 4 * h->mb_xy;
7.     //L0: 将宏块的Cache中的MV拷贝至整张图片的motion_val变量中
8.     if (USES_LIST(mb_type, 0)) {
9.         write_back_motion_list(h, b_stride, b_xy, b8_xy, mb_type, 0);
10.    } else {
11.        fill_rectangle(&h->cur_pic.ref_index[0][b8_xy],
12.                       2, 2, 2, (uint8_t)LIST_NOT_USED, 1);
13.    }
14.    //L1: 将宏块的Cache中的MV拷贝至整张图片的motion_val变量中 (最后一个参数不同)
15.    if (USES_LIST(mb_type, 1))
16.        write_back_motion_list(h, b_stride, b_xy, b8_xy, mb_type, 1);
17.
18.    if (h->slice_type_nos == AV_PICTURE_TYPE_B && CABAC(h)) {
19.        if (IS_8X8(mb_type)) {
20.            uint8_t *direct_table = &h->direct_table[4 * h->mb_xy];
21.            direct_table[1] = h->sub_mb_type[1] >> 1;
22.            direct_table[2] = h->sub_mb_type[2] >> 1;
23.            direct_table[3] = h->sub_mb_type[3] >> 1;
24.        }
25.    }
26. }
```

从源代码可以看出，如果使用了List0，会调用一次write_back_motion_list()函数（注意最后一个参数为“0”）；如果使用了List1（双向预测），又会调用一次write_back_motion_list()函数（注意最后一个参数为“1”）。下面再看一下write_back_motion_list()函数。

write_back_motion_list()

write_back_motion_list()是将宏块的Cache中的MV拷贝至整张图片的motion_val变量中的执行函数。该函数定义如下所示。

```

1. //将宏块的Cache中的MV拷贝至整张图片的motion_val变量中-这是具体的执行函数
2. static av_always_inline void write_back_motion_list(H264Context *h,
3.                                                     int b_stride,
4.                                                     int b_xy, int b8_xy,
5.                                                     int mb_type, int list)
6. {
7.     //目的：整张图片的motion_val
8.     int16_t(*mv_dst)[2] = &h->cur_pic.motion_val[list][b_xy];
9.     //源：宏块的Cache，从scan8[0]开始
10.    int16_t(*mv_src)[2] = &h->mv_cache[list][scan8[0]];
11.    //一个运动矢量的坐标（x或者y）占用一个int16_t
12.    //一个宏块一行有4个运动矢量
13.    //每个运动矢量包含2个坐标（x和y）
14.    //一个宏块一行运动矢量的数据量=16*4*2=128
15.    //因此这里拷贝128bit
16.    AV_COPY128(mv_dst + 0 * b_stride, mv_src + 8 * 0);
17.    //每个宏块有4行4列的运动矢量（总计16个）
18.    //因此要分别拷贝4行
19.    //b_stride代表了1行图像中运动矢量的个数
20.    AV_COPY128(mv_dst + 1 * b_stride, mv_src + 8 * 1);
21.    AV_COPY128(mv_dst + 2 * b_stride, mv_src + 8 * 2);
22.    AV_COPY128(mv_dst + 3 * b_stride, mv_src + 8 * 3);
23.    if (CABAC(h)) {
24.        uint8_t (*mvd_dst)[2] = &h->mvd_table[list][FM0 ? 8 * h->mb_xy
25.                                                : h->mb2br_xy[h->mb_xy]];
26.        uint8_t(*mvd_src)[2] = &h->mvd_cache[list][scan8[0]];
27.        if (IS_SKIP(mb_type)) {
28.            AV_ZERO128(mvd_dst);
29.        } else {
30.            AV_COPY64(mvd_dst, mvd_src + 8 * 3);
31.            AV_COPY16(mvd_dst + 3 + 3, mvd_src + 3 + 8 * 0);
32.            AV_COPY16(mvd_dst + 3 + 2, mvd_src + 3 + 8 * 1);
33.            AV_COPY16(mvd_dst + 3 + 1, mvd_src + 3 + 8 * 2);
34.        }
35.    }
36.
37.    {
38.        //拷贝参考帧序号
39.        //目的：整张图片的ref_index
40.        int8_t *ref_index = &h->cur_pic.ref_index[list][b8_xy];
41.        //源：宏块的Cache，从scan8[0]开始
42.        int8_t *ref_cache = h->ref_cache[list];
43.        ref_index[0 + 0 * 2] = ref_cache[scan8[0]];
44.        ref_index[1 + 0 * 2] = ref_cache[scan8[4]];
45.        ref_index[0 + 1 * 2] = ref_cache[scan8[8]];
46.        ref_index[1 + 1 * 2] = ref_cache[scan8[12]];
47.    }
48. }

```

由于源代码中作了比较详细的注释，这里就不在过多解释了。从源代码中可以得知write_back_motion_list()首先将mv_cache中的运动矢量信息拷贝至cur_pic（H264Picture类型）的motion_val中（motion_val中存储了整张图片的运动矢量信息）；然后将ref_cache中的参考帧序号信息拷贝至cur_pic（H264Picture类型）的ref_index中（ref_index中存储了整张图片的参考帧信息）。

至此FFmpeg H.264解码器的熵解码部分就基本上分析完毕了。

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/45114453>

文章标签： [FFmpeg](#) [CAVLC](#) [源代码](#) [视频解码](#) [H.264](#)

个人分类： [FFMPEG](#)

所属专栏： [FFmpeg](#)

此PDF由spyyg生成,请尊重原作者版权!!!

我的邮箱:liushidc@163.com