

## 原 x264源代码简单分析：x264\_slice\_write()

2015年05月17日 18:36:39 阅读数：7836

=====

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](#)

=====

本文简单分析x264的x264\_slice\_write()的源代码。x264\_slice\_write()是x264项目的核心，它完成了编码了一个Slice的工作。根据功能的不同，该函数可以分为滤波（Filter），分析（Analysis），宏块编码（Encode）和熵编码（Entropy Encoding）几个子模块。本文首先对x264\_slice\_write()进行总体的概括，在后续文章中将会对上述几个子模块展开进行分析。

## 函数调用关系图

x264\_slice\_write()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](#)

x264\_slice\_write()的函数调用关系如下图所示。

从图中可以看出，x264\_slice\_write()调用了如下函数：

x264\_nal\_start()：开始写一个NALU。  
 x264\_macroblock\_thread\_init()：初始化宏块重建数据缓存fdec\_buf[]和编码数据缓存fenc\_buf[]。  
 x264\_slice\_header\_write()：输出 Slice Header。  
 x264\_fdec\_filter\_row()：滤波模块。该模块包含了环路滤波，半像素插值，SSIM/PSNR的计算。  
 x264\_macroblock\_cache\_load()：将要编码的宏块的周围的宏块的信息读进来。  
 x264\_macroblock\_analyse()：分析模块。该模块包含了帧内预测模式分析以及帧间运动估计等。  
 x264\_macroblock\_encode()：宏块编码模块。该模块通过对残差的DCT变换、量化等方式对宏块进行编码。  
 x264\_macroblock\_write\_cabac()：CABAC熵编码模块。  
 x264\_macroblock\_write\_cavlc()：CAVLC熵编码模块。  
 x264\_macroblock\_cache\_save()：保存当前宏块的信息。  
 x264\_ratecontrol\_mb()：码率控制。  
 x264\_nal\_end()：结束写一个NALU。

本文将会对上述函数进行分析。其中x264\_fdec\_filter\_row(), x264\_macroblock\_analyse(), x264\_macroblock\_encode(), x264\_macroblock\_write\_cabac()/x264\_macroblock\_write\_cavlc()只做概述，后续文章中再做分析。

## x264\_slice\_write()

x264\_slice\_write()用于编码一个Slice。该函数的定义位于encoder/encoder.c，如下所示。

```
[cpp]
1.  /*****
2.   * 真正的编码——编码1个Slice
3.   * 注释和处理：雷霄骅
4.   * http://blog.csdn.net/leixiaohua1020
5.   * leixiaohua1020@126.com
6.   *****/
7.  static intptr_t x264_slice_write( x264_t *h )
8.  {
9.      int i_skip;
10.     //宏块的序号，以及序号对应的x, y坐标
11.     int mb_xy, i_mb_x, i_mb_y;
12.     /* NALUs other than the first use a 3-byte startcode.
13.      * Add one extra byte for the rbsp, and one more for the final CABAC putbyte.
14.      * Then add an extra 5 bytes just in case, to account for random NAL escapes and
15.      * other inaccuracies. */
16.     int overhead_guess = (NALU_OVERHEAD - (h->param.b_annexb && h->out.i_nal)) + 1 + h->param.b_cabac + 5;
17.     int slice_max_size = h->param.i_slice_max_size > 0 ? (h->param.i_slice_max_size-overhead_guess)*8 : 0;
18.     int back_up_bitstream_cavlc = !h->param.b_cabac && h->sps->i_profile_idc < PROFILE_HIGH;
19.     int back_up_bitstream = slice_max_size || back_up_bitstream_cavlc;
20.     int starting_bits = bs_pos(&h->out.bs);
21.     int b_deblock = h->sh.i_disable_deblocking_filter_idc != 1;
22.     int b_hpel = h->fdec->b_kept_as_ref;
23.     int orig_last_mb = h->sh.i_last_mb;
24.     int thread_last_mb = h->i_threadslice_end * h->mb.i_mb_width - 1;
25.     uint8_t *last_emu_check;
26. #define BS_BAK_SLICE_MAX_SIZE 0
27. #define BS_BAK_CAVLC_OVERFLOW 1
28. #define BS_BAK_SLICE_MIN_MBS 2
29. #define BS_BAK_ROW_VBV 3
30.     x264_bs_bak_t bs_bak[4];
31.     b_deblock &= b_hpel || h->param.b_full_recon || h->param.psz_dump_yuv;
32.     bs_realign( &h->out.bs );
33.
34.     /* Slice */
35.     //开始输出一个NAL
36.     //后面对应着x264_nal_end()
37.     x264_nal_start( h, h->i_nal_type, h->i_nal_ref_idc );
38.     h->out.nal[h->out.i_nal].i_first_mb = h->sh.i_first_mb;
39.
40.     /* Slice header */
41.
42.     //存储宏块像素的缓存fdec_buf和fenc_buf的初始化
43.     //宏块编码缓存p_fenc[0], p_fenc[1], p_fenc[2]
44.     //宏块重建缓存p_fdec[0], p_fdec[1], p_fdec[2]
45.     //[0]存Y, [1]存U, [2]存V
46.     x264_macroblock_thread_init( h );
47.
48.     /* Set the QP equal to the first QP in the slice for more accurate CABAC initialization. */
49.     h->mb.i_mb_xy = h->sh.i_first_mb;
50.     h->sh.i_qp = x264_ratecontrol_mb_qp( h );
51.     h->sh.i_qp = SPEC_QP( h->sh.i_qp );
52.     h->sh.i_qp_delta = h->sh.i_qp - h->pps->i_pic_init_qp;
53.     //输出 slice header
```

```

54.     x264_slice_header_write( &h->out.bs, &h->sh, h->i_nal_ref_idc );
55.     //如果使用CABAC, 需要初始化
56.     if( h->param.b_cabac )
57.     {
58.         /* alignment needed */
59.         bs_align_1( &h->out.bs );
60.
61.         /* init cabac */
62.         x264_cabac_context_init( h, &h->cabac, h->sh.i_type, x264_clip3( h->sh.i_qp-QP_BD_OFFSET, 0, 51 ), h-
>sh.i_cabac_init_idc );
63.         x264_cabac_encode_init ( &h->cabac, h->out.bs.p, h->out.bs.p_end );
64.         last_emu_check = h->cabac.p;
65.     }
66.     else
67.     {
68.         last_emu_check = h->out.bs.p;
69.         h->mb.i_last_qp = h->sh.i_qp;
70.         h->mb.i_last_dqp = 0;
71.         h->mb.field_decoding_flag = 0;
72.         //宏块位置-纵坐标 (初始值)
73.         i_mb_y = h->sh.i_first_mb / h->mb.i_mb_width;
74.         //宏块位置-横坐标 (初始值)
75.         i_mb_x = h->sh.i_first_mb % h->mb.i_mb_width;
76.         i_skip = 0;
77.
78.         //一个大循环
79.         //对一个slice中每个宏块进行编码
80.         while( 1 )
81.         {
82.             //宏块序号。由i_mb_x和i_mb_y计算而来。
83.             mb_xy = i_mb_x + i_mb_y * h->mb.i_mb_width;
84.             int mb_spos = bs_pos(&h->out.bs) + x264_cabac_pos(&h->cabac);
85.             //一行的开始
86.             if( i_mb_x == 0 )
87.             {
88.                 if( x264_bitstream_check_buffer( h ) )
89.                     return -1;
90.                 if( !(i_mb_y & SLICE_MBAFF) && h->param.rc.i_vbv_buffer_size )
91.                     x264_bitstream_backup( h, &bs_bak[BS_BAK_ROW_VBV], i_skip, 1 );
92.                 //去块效应滤波、半像素插值、SSIM/PSNR计算等
93.                 //一次处理一行宏块
94.                 if( !h->mb.b_reencode_mb )
95.                     x264_fdec_filter_row( h, i_mb_y, 0 );
96.
97.                 if( back_up_bitstream )
98.                 {
99.                     if( back_up_bitstream_cavlc )
100.                        x264_bitstream_backup( h, &bs_bak[BS_BAK_CAVLC_OVERFLOW], i_skip, 0 );
101.                     if( slice_max_size && !(i_mb_y & SLICE_MBAFF) )
102.                     {
103.                         x264_bitstream_backup( h, &bs_bak[BS_BAK_SLICE_MAX_SIZE], i_skip, 0 );
104.                         if( (thread_last_mb+1-mb_xy) == h->param.i_slice_min_mbs )
105.                             x264_bitstream_backup( h, &bs_bak[BS_BAK_SLICE_MIN_MBS], i_skip, 0 );
106.                     }
107.                 }
108.
109.                 if( PARAM_INTERLACED )
110.                 {
111.                     if( h->mb.b_adaptive_mbafl )
112.                     {
113.                         if( !(i_mb_y&1) )
114.                         {
115.                             /* FIXME: VSAD is fast but fairly poor at choosing the best interlace type. */
116.                             h->mb.b_interlaced = x264_field_vsad( h, i_mb_x, i_mb_y );
117.                             memcpy( &h->zigzagf, MB_INTERLACED ? &h->zigzagf_interlaced : &h->zigzagf_progressive, sizeof(h->zigzagf) );
118.                             if( !MB_INTERLACED && (i_mb_y+2) == h->mb.i_mb_height )
119.                                 x264_expand_border_mbpair( h, i_mb_x, i_mb_y );
120.                         }
121.                     }
122.                     h->mb.field[mb_xy] = MB_INTERLACED;
123.                 }
124.
125.                 /* load cache */
126.                 //将要编码的宏块的周围的宏块的值读进来
127.                 //主要是上面、左边块的值
128.                 if( SLICE_MBAFF )
129.                     x264_macroblock_cache_load_interlaced( h, i_mb_x, i_mb_y );
130.                 else
131.                     x264_macroblock_cache_load_progressive( h, i_mb_x, i_mb_y );
132.                 //分析-帧内预测模式选择、帧间运动估计等
133.                 x264_macroblock_analyse( h );
134.
135.                 /* encode this macroblock -> be careful it can change the mb type to P_SKIP if needed */
136.                 reencode:
137.                 //编码-残差DCT变换、量化
138.                 x264_macroblock_encode( h );
139.                 //输出CABAC
140.                 if( h->param.b_cabac )
141.                 {
142.                     if( mb_xy > h->sh.i_first_mb && !(SLICE_MBAFF && (i_mb_y&1)) )
143.                         x264_cabac_encode_terminal( &h->cabac );
144.

```

```

144.
145.         if( IS_SKIP( h->mb.i_type ) )
146.             x264_cabac_mb_skip( h, 1 );
147.         else
148.         {
149.             if( h->sh.i_type != SLICE_TYPE_I )
150.                 x264_cabac_mb_skip( h, 0 );
151.             //输出
152.             x264_macroblock_write_cabac( h, &h->cabac );
153.         }
154.     }
155. else
156. {
157.     //输出CAVLC
158.     if( IS_SKIP( h->mb.i_type ) )
159.         i_skip++;
160.     else
161.     {
162.         if( h->sh.i_type != SLICE_TYPE_I )
163.         {
164.             bs_write_ue( &h->out.bs, i_skip ); /* skip run */
165.             i_skip = 0;
166.         }
167.         //输出
168.         x264_macroblock_write_cavlc( h );
169.         /* If there was a CAVLC level code overflow, try again at a higher QP. */
170.         if( h->mb.b_overflow )
171.         {
172.             h->mb.i_chroma_qp = h->chroma_qp_table[++h->mb.i_qp];
173.             h->mb.i_skip_intra = 0;
174.             h->mb.b_skip_mc = 0;
175.             h->mb.b_overflow = 0;
176.             x264_bitstream_restore( h, &bs_bak[BS_BAK_CAVLC_OVERFLOW], &i_skip, 0 );
177.             goto reencode;
178.         }
179.     }
180. }
181.
182. int total_bits = bs_pos(&h->out.bs) + x264_cabac_pos(&h->cabac);
183. int mb_size = total_bits - mb_spos;
184.
185. if( slice_max_size && (!SLICE_MBAFF || (i_mb_y&1)) )
186. {
187.     /* Count the skip run, just in case. */
188.     if( !h->param.b_cabac )
189.         total_bits += bs_size_ue_big( i_skip );
190.     /* Check for escape bytes. */
191.     uint8_t *end = h->param.b_cabac ? h->cabac.p : h->out.bs.p;
192.     for( ; last_emu_check < end - 2; last_emu_check++ )
193.         if( last_emu_check[0] == 0 && last_emu_check[1] == 0 && last_emu_check[2] <= 3 )
194.         {
195.             slice_max_size -= 8;
196.             last_emu_check++;
197.         }
198.     /* We'll just re-encode this last macroblock if we go over the max slice size. */
199.     if( total_bits - starting_bits > slice_max_size && !h->mb.b_reencode_mb )
200.     {
201.         if( !x264_frame_new_slice( h, h->fdec ) )
202.         {
203.             /* Handle the most obnoxious slice-min-mbs edge case: we need to end the slice
204.              * because it's gone over the maximum size, but doing so would violate slice-min-mbs.
205.              * If possible, roll back to the last checkpoint and try again.
206.              * We could try raising QP, but that would break in the case where a slice spans multiple
207.              * rows, which the re-encoding infrastructure can't currently handle. */
208.             if( mb_xy <= thread_last_mb && (thread_last_mb+1-mb_xy) < h->param.i_slice_min_mbs )
209.             {
210.                 if( thread_last_mb-h->param.i_slice_min_mbs < h->sh.i_first_mb+h->param.i_slice_min_mbs )
211.                 {
212.                     x264_log( h, X264_LOG_WARNING, "slice-max-size violated (frame %d, cause: slice-min-mbs)\n", h->i_frame
213. );
214.                     slice_max_size = 0;
215.                     goto cont;
216.                 }
217.                 x264_bitstream_restore( h, &bs_bak[BS_BAK_SLICE_MIN_MBS], &i_skip, 0 );
218.                 h->mb.b_reencode_mb = 1;
219.                 h->sh.i_last_mb = thread_last_mb-h->param.i_slice_min_mbs;
220.                 break;
221.             }
222.             if( mb_xy-SLICE_MBAFF*h->mb.i_mb_stride != h->sh.i_first_mb )
223.             {
224.                 x264_bitstream_restore( h, &bs_bak[BS_BAK_SLICE_MAX_SIZE], &i_skip, 0 );
225.                 h->mb.b_reencode_mb = 1;
226.                 if( SLICE_MBAFF )
227.                 {
228.                     // set to bottom of previous mbpair
229.                     if( i_mb_x )
230.                         h->sh.i_last_mb = mb_xy-1+h->mb.i_mb_stride*(!i_mb_y&1);
231.                     else
232.                         h->sh.i_last_mb = (i_mb_y-2+!(i_mb_y&1))*h->mb.i_mb_stride + h->mb.i_mb_width - 1;
233.                 }
234.                 else
235.                     h->sh.i_last_mb = mb_xy-1;

```

```

235.         break;
236.     }
237.     else
238.         h->sh.i_last_mb = mb_xy;
239.     }
240.     else
241.         slice_max_size = 0;
242.     }
243. }
244. cont:
245.     h->mb.b_reencode_mb = 0;
246.
247.     /* save cache */
248.     //保存当前宏块的值, 用于以后的宏块的编码
249.     //包括Intra4x4宏块帧内预测模式, DCT非零系数, 运动矢量, 参考帧序号等等
250.     x264_macroblock_cache_save( h );
251.     //码率控制
252.     if( x264_ratecontrol_mb( h, mb_size ) < 0 )
253.     {
254.         x264_bitstream_restore( h, &bs_bak[BS_BAK_ROW_VBV], &i_skip, 1 );
255.         h->mb.b_reencode_mb = 1;
256.         i_mb_x = 0;
257.         i_mb_y = i_mb_y - SLICE_MBAFF;
258.         h->mb.i_mb_prev_xy = i_mb_y * h->mb.i_mb_stride - 1;
259.         h->sh.i_last_mb = orig_last_mb;
260.         continue;
261.     }
262.
263.     /* accumulate mb stats */
264.     //后面很大一段代码都是对stat结构体中的统计信息进行赋值=====
265.     h->stat.frame.i_mb_count[h->mb.i_type]++;
266.
267.     int b_intra = IS_INTRA( h->mb.i_type );
268.     int b_skip = IS_SKIP( h->mb.i_type );
269.     if( h->param.i_log_level >= X264_LOG_INFO || h->param.rc.b_stat_write )
270.     {
271.         if( !b_intra && !b_skip && !IS_DIRECT( h->mb.i_type ) )
272.         {
273.             if( h->mb.i_partition != D_8x8 )
274.                 h->stat.frame.i_mb_partition[h->mb.i_partition] += 4;
275.             else
276.                 for( int i = 0; i < 4; i++ )
277.                     h->stat.frame.i_mb_partition[h->mb.i_sub_partition[i]] ++;
278.             if( h->param.i_frame_reference > 1 )
279.                 for( int i_list = 0; i_list <= (h->sh.i_type == SLICE_TYPE_B); i_list++ )
280.                     for( int i = 0; i < 4; i++ )
281.                     {
282.                         int i_ref = h->mb.cache.ref[i_list][ x264_scan8[4*i] ];
283.                         if( i_ref >= 0 )
284.                             h->stat.frame.i_mb_count_ref[i_list][i_ref] ++;
285.                     }
286.         }
287.     }
288.
289.     if( h->param.i_log_level >= X264_LOG_INFO )
290.     {
291.         if( h->mb.i_cbp_luma | h->mb.i_cbp_chroma )
292.         {
293.             if( CHROMA444 )
294.             {
295.                 for( int i = 0; i < 4; i++ )
296.                     if( h->mb.i_cbp_luma & (1 << i) )
297.                         for( int p = 0; p < 3; p++ )
298.                         {
299.                             int s8 = i*4+p*16;
300.                             int nnz8x8 = M16( &h->mb.cache.non_zero_count[x264_scan8[s8]+0] )
301.                                 | M16( &h->mb.cache.non_zero_count[x264_scan8[s8]+8] );
302.                             h->stat.frame.i_mb_cbp[!b_intra + p*2] += !!nnz8x8;
303.                         }
304.             }
305.             else
306.             {
307.                 int cbpsum = (h->mb.i_cbp_luma&1) + ((h->mb.i_cbp_luma>>1)&1)
308.                     + ((h->mb.i_cbp_luma>>2)&1) + (h->mb.i_cbp_luma>>3);
309.                 h->stat.frame.i_mb_cbp[!b_intra + 0] += cbpsum;
310.                 h->stat.frame.i_mb_cbp[!b_intra + 2] += !!h->mb.i_cbp_chroma;
311.                 h->stat.frame.i_mb_cbp[!b_intra + 4] += h->mb.i_cbp_chroma >> 1;
312.             }
313.         }
314.         if( h->mb.i_cbp_luma && !b_intra )
315.         {
316.             h->stat.frame.i_mb_count_8x8dct[0] ++;
317.             h->stat.frame.i_mb_count_8x8dct[1] += h->mb.b_transform_8x8;
318.         }
319.         if( b_intra && h->mb.i_type != I_PCM )
320.         {
321.             if( h->mb.i_type == I_16x16 )
322.                 h->stat.frame.i_mb_pred_mode[0][h->mb.i_intra16x16_pred_mode]++;
323.             else if( h->mb.i_type == I_8x8 )
324.                 for( int i = 0; i < 16; i += 4 )
325.                     h->stat.frame.i_mb_pred_mode[i][h->mb.cache.intra4x4_pred_mode[x264_scan8[i]]]++;

```

```

326.         else //if( h->mb.i_type == I_4x4 )
327.             for( int i = 0; i < 16; i++ )
328.                 h->stat.frame.i_mb_pred_mode[2][h->mb.cache.intra4x4_pred_mode[x264_scan8[i]]]++;
329.                 h->stat.frame.i_mb_pred_mode[3][x264_mb_chroma_pred_mode_fix[h->mb.i_chroma_pred_mode]]++;
330.             }
331.             h->stat.frame.i_mb_field[b_intra?0:b_skip?2:1] += MB_INTERLACED;
332.         }
333.         //=====
334.
335.         /* calculate deblock strength values (actual deblocking is done per-row along with hpel) */
336.         //计算去块效应滤波器强度Bs
337.         //这里没有滤波
338.         if( b_deblock )
339.             x264_macroblock_deblock_strength( h );
340.
341.         //如果处理完最后一个宏块，就跳出大循环
342.         if( mb_xy == h->sh.i_last_mb )
343.             break;
344.
345.         if( SLICE_MBAFF )
346.         {
347.             i_mb_x += i_mb_y & 1;
348.             i_mb_y ^= i_mb_x < h->mb.i_mb_width;
349.         }
350.         else
351.             i_mb_x++; //宏块序号x加1
352.         //处理完一行宏块
353.         if( i_mb_x == h->mb.i_mb_width )
354.         {
355.             //该处理下一行了
356.             i_mb_y++; //宏块序号y加1
357.             i_mb_x = 0; //宏块序号x设置为0
358.         }
359.     }
360.     if( h->sh.i_last_mb < h->sh.i_first_mb )
361.         return 0;
362.
363.     h->out.nal[h->out.i_nal].i_last_mb = h->sh.i_last_mb;
364.
365.     //熵编码的收尾工作
366.     if( h->param.b_cabac )
367.     {
368.         x264_cabac_encode_flush( h, &h->cabac );
369.         h->out.bs.p = h->cabac.p;
370.     }
371.     else
372.     {
373.         if( i_skip > 0 )
374.             bs_write_ue( &h->out.bs, i_skip ); /* last skip run */
375.         /* rbsp_slice_trailing_bits */
376.         bs_rbsp_trailing( &h->out.bs );
377.         bs_flush( &h->out.bs );
378.     }
379.     //结束输出一个NAL
380.     //前面对应着x264_nal_start()
381.     if( x264_nal_end( h ) )
382.         return -1;
383.     //多线程并行处理？
384.     if( h->sh.i_last_mb == (h->i_threadslice_end * h->mb.i_mb_width - 1) )
385.     {
386.         h->stat.frame.i_misc_bits = bs_pos( &h->out.bs )
387.             + (h->out.i_nal*NALU_OVERHEAD * 8)
388.             - h->stat.frame.i_tex_bits
389.             - h->stat.frame.i_mv_bits;
390.         x264_fdec_filter_row( h, h->i_threadslice_end, 0 );
391.
392.         if( h->param.b_sliced_threads )
393.         {
394.             /* Tell the main thread we're done. */
395.             x264_threadslice_cond_broadcast( h, 1 );
396.             /* Do hpel now */
397.             for( int mb_y = h->i_threadslice_start; mb_y <= h->i_threadslice_end; mb_y++ )
398.                 x264_fdec_filter_row( h, mb_y, 1 );
399.             x264_threadslice_cond_broadcast( h, 2 );
400.             /* Do the first row of hpel, now that the previous slice is done */
401.             if( h->i_thread_idx > 0 )
402.             {
403.                 x264_threadslice_cond_wait( h->thread[h->i_thread_idx-1], 2 );
404.                 x264_fdec_filter_row( h, h->i_threadslice_start + (1 << SLICE_MBAFF), 2 );
405.             }
406.         }
407.
408.         /* Free mb info after the last thread's done using it */
409.         if( h->fdec->mb_info_free && (!h->param.b_sliced_threads || h->i_thread_idx == (h->param.i_threads-1)) )
410.         {
411.             h->fdec->mb_info_free( h->fdec->mb_info );
412.             h->fdec->mb_info = NULL;
413.             h->fdec->mb_info_free = NULL;
414.         }
415.     }
416.

```

```
417.     return 0;
418. }
```

根据源代码简单梳理了x264\_slice\_write()的流程，如下所示：

- (1) 调用x264\_nal\_start()开始输出一个NALU。
- (2) x264\_macroblock\_thread\_init()：初始化宏块重建像素缓存fdec\_buf[]和编码像素缓存fenc\_buf[]。
- (3) 调用x264\_slice\_header\_write()输出 Slice Header。
- (4) 进入一个循环，该循环每执行一遍编码一个宏块：
  - a) 每处理一行宏块，调用一次x264\_fdec\_filter\_row()执行滤波模块。
  - b) 调用x264\_macroblock\_cache\_load\_progressive()将要编码的宏块的周围的宏块的信息读进来。
  - c) 调用x264\_macroblock\_analyse()执行分析模块。
  - d) 调用x264\_macroblock\_encode()执行宏块编码模块。
  - e) 调用x264\_macroblock\_write\_cabac()/x264\_macroblock\_write\_cavlc()执行熵编码模块。
  - f) 调用x264\_macroblock\_cache\_save()保存当前宏块的信息。
  - g) 调用x264\_ratecontrol\_mb()执行码率控制。
  - h) 准备处理下一个宏块。
- (5) 调用x264\_nal\_end()结束输出一个NALU。

下文分别从数据结构和函数两个方面分析x264\_slice\_write()的源代码。

## 重要的数据结构

X264在宏块编码方面涉及到下面几个比较重要的结构体：

宏块像素存储缓存fenc\_buf[]和fdec\_buf[]——位于x264\_t.mb.pic中，用于存储宏块的亮度和色度像素。

宏块各种信息的缓存Cache——位于x264\_t.mb.pic中，用于存储宏块的信息例如4x4帧内预测模式、DCT的非0系数个数、运动矢量、参考帧序号等。

图像半像素点存储空间filtered[]——位于x264\_frame\_t中，用于存储半像素插值后的点。

## 宏块像素存储缓存fenc\_buf[]和fdec\_buf[]

fenc\_buf[]和fdec\_buf[]为x264\_t.mb.cache中的结构体，用于存储一个宏块的像素数据。其中fenc\_buf[]用于存储宏块编码像素数据，而fdec\_buf[]用于存储宏块重建像素数据。他们的定义如下所示。

```
[cpp]
1.  /* space for p_fenc and p_fdec */
2.  #define FENC_STRIDE 16
3.  #define FDEC_STRIDE 32
4.  //存储编码宏块fenc和重建宏块fdec的内存
5.  uint8_t fenc_buf[48*FENC_STRIDE]
6.  uint8_t fdec_buf[52*FDEC_STRIDE]
```

从定义可以看出，fenc\_buf[]每行16个数据；而fdec\_buf[]每行32个数据。在x264\_t.mb.cache中和fenc\_buf[]和fdec\_buf[]相关的指针数组还有p\_fenc[3]和p\_fdec[3]，它们中的3个元素[0]、[1]、[2]分别指向分别指向对应缓存buf的Y、U、V分量。下图画出了像素格式为YUV420P的时候fenc\_buf[]的存储示意图。图中灰色区域存储Y，蓝色区域存储U，粉红区域存储V。p\_fenc[0]指向Y的存储区域，p\_fenc[1]指向U的存储区域，p\_fenc[2]指向V的存储区域，在图中以方框的形式标注了出来。

下图画出了像素格式为YUV420P的时候fdec\_buf[]的存储示意图。图中灰色区域存储Y，蓝色区域存储U，粉红区域存储V。p\_fenc[0]指向Y的存储区域，p\_fenc[1]指向U的存储区域，p\_fenc[2]指向V的存储区域，在图中以方框的形式标注了出来。

从图中可以看出，fdec\_buf[]和fenc\_buf[]主要的区别在于fdec\_buf[]像素块的左边和上边包含了左上方相邻块用于预测的像素。

## 宏块各种信息的缓存Cache

在x264中x264\_t.mb.cache结构体中包含了存储宏块信息的各种各样的缓存Cache。例如：

intra4x4\_pred\_mode：Intra4x4帧内预测模式的缓存

non\_zero\_count：DCT的非0系数个数的缓存

mv：运动矢量缓存

ref：运动矢量参考帧的缓存

这几个Cache的定义如下所示。

```
1.  /* 宏块信息缓存cache */
2.  struct
3.  {
4.      /* real intra4x4_pred_mode if I_4X4 or I_8X8, I_PRED_4x4_DC if mb available, -1 if not */
5.      /*
6.       * mb.cache.intra4x4_pred_mode[]格式如下
7.       * |
8.       * ---+-----
9.       * | 0 0 0 y y y y y
10.      * | 0 0 0 y Y Y Y Y
11.      * | 0 0 0 y Y Y Y Y
12.      * | 0 0 0 y Y Y Y Y
13.      * | 0 0 0 y Y Y Y Y
14.      */
15.      ALIGNED_8( int8_t intra4x4_pred_mode[X264_SCAN8_LUMA_SIZE] );
16.
17.      /* i_non_zero_count if available else 0x80 */
18.      /*
19.       * mb.cache.non_zero_count[]格式如下
20.       * |
21.       * ---+-----
22.       * | 0 0 0 y y y y y
23.       * | 0 0 0 y Y Y Y Y
24.       * | 0 0 0 y Y Y Y Y
25.       * | 0 0 0 y Y Y Y Y
26.       * | 0 0 0 y Y Y Y Y
27.       * | 0 0 0 u u u u u
28.       * | 0 0 0 u U U U U
29.       * | 0 0 0 u U U U U
30.       * | 0 0 0 u U U U U
31.       * | 0 0 0 u U U U U
32.       * | 0 0 0 v v v v v
33.       * | 0 0 0 v V V V V
34.       * | 0 0 0 v V V V V
35.       * | 0 0 0 v V V V V
36.       * | 0 0 0 v V V V V
37.       */
38.      ALIGNED_16( uint8_t non_zero_count[X264_SCAN8_SIZE] );
39.
40.      /* -1 if unused, -2 if unavailable */
41.      /*
42.       * mb.cache.ref[0][]格式如下
43.       * |
44.       * ---+-----
45.       * | 0 0 0 y y y y y
46.       * | 0 0 0 y Y Y Y Y
47.       * | 0 0 0 y Y Y Y Y
48.       * | 0 0 0 y Y Y Y Y
49.       * | 0 0 0 y Y Y Y Y
50.       */
51.      ALIGNED_4( int8_t ref[2][X264_SCAN8_LUMA_SIZE] );
52.
53.      /* 0 if not available */
54.      /*
55.       * mb.cache.mv[0][]格式如下
56.       * |
57.       * ---+-----
58.       * | 0 0 0 y y y y y
59.       * | 0 0 0 y Y Y Y Y
60.       * | 0 0 0 y Y Y Y Y
61.       * | 0 0 0 y Y Y Y Y
62.       * | 0 0 0 y Y Y Y Y
63.       */
64.      ALIGNED_16( int16_t mv[2][X264_SCAN8_LUMA_SIZE][2] );
65.      ALIGNED_8( uint8_t mvd[2][X264_SCAN8_LUMA_SIZE][2] );
66.
67.      /* 1 if SKIP or DIRECT. set only for B-frames + CABAC */
68.      ALIGNED_4( int8_t skip[X264_SCAN8_LUMA_SIZE] );
69.
70.      ALIGNED_4( int16_t direct_mv[2][4][2] );
71.      ALIGNED_4( int8_t direct_ref[2][4] );
72.      int      direct_partition;
73.      ALIGNED_4( int16_t pskip_mv[2] );
74.
75.      /* number of neighbors (top and left) that used 8x8 dct */
76.      int      i_neighbour_transform_size;
77.      int      i_neighbour_skip;
78.
79.      /* neighbor CBPs */
80.      int      i_cbp_top;
81.      int      i_cbp_left;
82.
83.      /* extra data required for mbaff in mv prediction */
84.      int16_t topright_mv[2][3][2];
85.      int8_t topright_ref[2][3];
86.
87.      /* current mb deblock strength */
88.      uint8_t (*deblock_strength)[8][4];
89.  } cache;
```



通过观察上面的定义，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）。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x264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.  /* Scan8 organization:
2.  *   0 1 2 3 4 5 6 7
3.  * 0  DY   y y y y y
4.  * 1       y Y Y Y Y
5.  * 2       y Y Y Y Y
6.  * 3       y Y Y Y Y
7.  * 4       y Y Y Y Y
8.  * 5  DU   u u u u u
9.  * 6       u U U U U
10. * 7       u U U U U
11. * 8       u U U U U
12. * 9       u U U U U
13. * 10 DV    v v v v v
14. * 11       v V V V V
15. * 12       v V V V V
16. * 13       v V V V V
17. * 14       v V V V V
18. * DY/DU/DV are for luma/chroma DC.
19. */
20. /*
21. * 扫描方式：
22. * 0-0 0-0
23. * / / /
24. * 0-0 0-0
25. * ,---'
26. * 0-0 0-0
27. * / / /
28. * 0-0 0-0
29. */
30. /*
31. * 关于多次出现的scan8
32. *
33. * cache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）。
34. * scan8[]则存储了宏块信息在cache中的索引值
35. *
36. * scan8[]中的“8”，意思应该是按照8x8为单元来扫描？
37. * 因此可以理解为“按照8x8为单元来扫描4x4的块”？
38. *
39. * 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。
40. *
41. * 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。
42. * 其中数字代表了scan8[]中元素的索引值
43. *
44. * +---+---+---+---+---+---+---+---+
45. * |   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
46. * +---+---+---+---+---+---+---+---+
47. * | 0 | 48|   |   |   | y| y| y| y|
48. * | 1 |   |   |   | y| 0| 1| 4| 5|
49. * | 2 |   |   |   | y| 2| 3| 6| 7|
50. * | 3 |   |   |   | y| 8| 9| 12| 13|
51. * | 4 |   |   |   | y| 10| 11| 14| 15|
52. * | 5 | 49|   |   |   | u| u| u| u|
53. * | 6 |   |   |   | u| 16| 17| 20| 21|
54. * | 7 |   |   |   | u| 18| 19| 22| 23|
55. * | 8 |   |   |   | u| 24| 25| 28| 29|
56. * | 9 |   |   |   | u| 26| 27| 30| 31|
57. * |10 | 50|   |   |   | v| v| v| v|
58. * |11 |   |   |   | v| 32| 33| 36| 37|
59. * |12 |   |   |   | v| 34| 35| 38| 39|
60. * |13 |   |   |   | v| 40| 41| 44| 45|
61. * |14 |   |   |   | v| 42| 43| 46| 47|
62. * |---+---+---+---+---+---+---+---+
63. * |   |
64. *
65. */
66.
67. #define LUMA_DC   48
68. #define CHROMA_DC 49
69.
70. static const uint8_t x264_scan8[16*3 + 3] =
71. {
72.     4+ 1*8, 5+ 1*8, 4+ 2*8, 5+ 2*8,
73.     6+ 1*8, 7+ 1*8, 6+ 2*8, 7+ 2*8,
74.     4+ 3*8, 5+ 3*8, 4+ 4*8, 5+ 4*8,
75.     6+ 3*8, 7+ 3*8, 6+ 4*8, 7+ 4*8,
76.     4+ 6*8, 5+ 6*8, 4+ 7*8, 5+ 7*8,
77.     6+ 6*8, 7+ 6*8, 6+ 7*8, 7+ 7*8,
78.     4+ 8*8, 5+ 8*8, 4+ 9*8, 5+ 9*8,
79.     6+ 8*8, 7+ 8*8, 6+ 9*8, 7+ 9*8,
80.     4+11*8, 5+11*8, 4+12*8, 5+12*8,
81.     6+11*8, 7+11*8, 6+12*8, 7+12*8,
82.     4+13*8, 5+13*8, 4+14*8, 5+14*8,
83.     6+13*8, 7+13*8, 6+14*8, 7+14*8,
84.     0+ 0*8, 0+ 5*8, 0+10*8
85. };

```

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[], ref[]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。

## 图像半像素点存储缓存filtered[]

X264中在图像运动搜索的过程中，需要使用1/4像素精度的运动补偿。其中半像素点的内插工作是提前完成的。每一帧的半像素点存储在x264\_frame\_t的filtered[3][4]变量中。其中前面的“[3]”代表Y, U, V三个分量, 后面的“[4]”分别存储了整像素, H半像素, V半像素, C (对角线) 半像素的数据。

下面的图以4x4图像块为例演示了filtered[][4]中几种半像素点与整像素点之间的位置关系。图中灰色的点为整像素点，黄色的点为半像素点。filtered[][0]存储了整像素点数据，filtered[][1]存储了H半像素点数据，filtered[][2]存储了V半像素点数据，filtered[][3]存储了C（对角线）半像素点数据。

## 重要的函数

下文简单记录x264\_slice\_write()中调用的几个函数：

- x264\_macroblock\_thread\_init()：初始化宏块重建数据缓存fdec\_buf[]和编码数据缓存fenc\_buf[]。
- x264\_slice\_header\_write()：输出 Slice Header。
- x264\_macroblock\_cache\_load()：将要编码的宏块的周围的宏块的信息读进来。
- x264\_macroblock\_cache\_save()：保存当前宏块的信息。

另外还有一些关键模块对应的函数将会在后续文章中进行分析：

- x264\_fdec\_filter\_row()：滤波模块。该模块包含了环路滤波，半像素插值，SSIM/PSNR的计算。
- x264\_macroblock\_analyse()：分析模块。该模块包含了帧内预测模式分析以及帧间运动估计等。
- x264\_macroblock\_encode()：宏块编码模块。该模块通过对残差的DCT变换、量化等方式对宏块进行编码。
- x264\_macroblock\_write\_cabac()：CABAC熵编码模块。
- x264\_macroblock\_write\_cavlc()：CAVLC熵编码模块。

## x264\_macroblock\_thread\_init()

x264\_macroblock\_thread\_init()用于初始化宏块重建数据缓存fdec\_buf[]和编码数据缓存fenc\_buf[]。该函数的定义位于common\macroblock.c，如下所示。

```
[cpp]
1. //存储宏块像素的缓存fdec_buf和fenc_buf的初始化
2. //设定宏块编码缓存p_fenc[0], p_fenc[1], p_fenc[2]
3. //设定宏块重建缓存p_fdec[0], p_fdec[1], p_fdec[2]
4. // [0]存Y, [1]存U, [2]存V
5. void x264_macroblock_thread_init( x264_t *h )
6. {
7.     h->mb.i_me_method = h->param.analyse.i_me_method;
```



```

99.      *      ...
100.     *      UV一共8行
101.     *      ...
102.    */
103.
104.    //fenc (编码帧) 结构比较简单, Y、U、V像素“挨着”存放
105.    h->mb.pic.p_fenc[0] = h->mb.pic.fenc_buf;
106.    //fdec (重建帧) 结构比较复杂, 需要当前宏块左边以及上边宏块的信息
107.    //第1行为空, 第2行用于存储上边宏块下边缘的像素
108.    h->mb.pic.p_fdec[0] = h->mb.pic.fdec_buf + 2*FDEC_STRIDE;
109.
110.    //U
111.    h->mb.pic.p_fenc[1] = h->mb.pic.fenc_buf + 16*FENC_STRIDE;
112.    h->mb.pic.p_fdec[1] = h->mb.pic.fdec_buf + 19*FDEC_STRIDE;
113.
114.    //V
115.    if( CHROMA444 )
116.    {
117.        h->mb.pic.p_fenc[2] = h->mb.pic.fenc_buf + 32*FENC_STRIDE;
118.        h->mb.pic.p_fdec[2] = h->mb.pic.fdec_buf + 36*FDEC_STRIDE;
119.    }
120.    else
121.    {
122.        //注意+8和+16
123.        h->mb.pic.p_fenc[2] = h->mb.pic.fenc_buf + 16*FENC_STRIDE + 8;
124.        h->mb.pic.p_fdec[2] = h->mb.pic.fdec_buf + 19*FDEC_STRIDE + 16;
125.    }
126. }

```

从源代码可以看出, x264\_macroblock\_thread\_init()设定了宏块编码数据指针p\_fenc[0], p\_fenc[1], p\_fenc[2]在fenc\_buf[]中的位置, 以及宏块重建数据指针p\_fdec[0], p\_fdec[1], p\_fdec[2]在fdec\_buf[]中的位置。由于前文中已经介绍过fenc\_buf[]和fdec\_buf[]的结构, 在这里不再重复。

## x264\_slice\_header\_write()

x264\_slice\_header\_write()用于输出Slice Header。该函数的定义位于encoder\encoder.c, 如下所示。

```

[cpp]
1. //输出 slice header
2. static void x264_slice_header_write( bs_t *s, x264_slice_header_t *sh, int i_nal_ref_idc )
3. {
4.     if( sh->b_mbaff )
5.     {
6.         int first_x = sh->i_first_mb % sh->sps->i_mb_width;
7.         int first_y = sh->i_first_mb / sh->sps->i_mb_width;
8.         assert( (first_y&1) == 0 );
9.         bs_write_ue( s, (2*first_x + sh->sps->i_mb_width*(first_y&1) + (first_y&1)) >> 1 );
10.    }
11.    else
12.    {
13.        bs_write_ue( s, sh->i_first_mb );//first_mb_in_slice: Slice中的第一个宏块的地址
14.        //slice_type: Slice类型 (I,B,P,SI,SP)
15.        bs_write_ue( s, sh->i_type + 5 ); /* same type things */
16.        //pic_parameter_set_id: PPS的索引号
17.        bs_write_ue( s, sh->i_pps_id );
18.        //frame_num: 指明了各图像的解码顺序
19.        bs_write( s, sh->sps->i_log2_max_frame_num, sh->i_frame_num & ((1<<sh->sps->i_log2_max_frame_num)-1) );
20.
21.        if( !sh->sps->b_frame_mbs_only )
22.        {
23.            bs_writel( s, sh->b_field_pic );
24.            if( sh->b_field_pic )
25.                bs_writel( s, sh->b_bottom_field );
26.        }
27.
28.        if( sh->i_idr_pic_id >= 0 ) /* NAL IDR */
29.            bs_write_ue( s, sh->i_idr_pic_id );//idr_pic_id: IDR图像的标识
30.
31.        if( sh->sps->i_poc_type == 0 )
32.        {
33.            bs_write( s, sh->sps->i_log2_max_poc_lsb, sh->i_poc & ((1<<sh->sps->i_log2_max_poc_lsb)-1) );
34.            if( sh->pps->b_pic_order && !sh->b_field_pic )
35.                bs_write_se( s, sh->i_delta_poc_bottom );
36.        }
37.
38.        if( sh->pps->b_redundant_pic_cnt )
39.            bs_write_ue( s, sh->i_redundant_pic_cnt );
40.
41.        if( sh->i_type == SLICE_TYPE_B )
42.            bs_writel( s, sh->b_direct_spatial_mv_pred );
43.
44.        if( sh->i_type == SLICE_TYPE_P || sh->i_type == SLICE_TYPE_B )
45.        {
46.            bs_writel( s, sh->b_num_ref_idx_override );
47.            if( sh->b_num_ref_idx_override )
48.            {
49.                bs_write_ue( s, sh->i_num_ref_idx_l0_active - 1 );
50.                if( sh->i_type == SLICE_TYPE_B )
51.                    bs_write_ue( s, sh->i_num_ref_idx_l1_active - 1 );

```

```

52.     }
53.
54.     /* ref pic list reordering */
55.     if( sh->i_type != SLICE_TYPE_I )
56.     {
57.         bs_writel( s, sh->b_ref_pic_list_reordering[0] );
58.         if( sh->b_ref_pic_list_reordering[0] )
59.         {
60.             for( int i = 0; i < sh->i_num_ref_idx_l0_active; i++ )
61.             {
62.                 bs_write_ue( s, sh->ref_pic_list_order[0][i].idc );
63.                 bs_write_ue( s, sh->ref_pic_list_order[0][i].arg );
64.             }
65.             bs_write_ue( s, 3 );
66.         }
67.     }
68.     if( sh->i_type == SLICE_TYPE_B )
69.     {
70.         bs_writel( s, sh->b_ref_pic_list_reordering[1] );
71.         if( sh->b_ref_pic_list_reordering[1] )
72.         {
73.             for( int i = 0; i < sh->i_num_ref_idx_l1_active; i++ )
74.             {
75.                 bs_write_ue( s, sh->ref_pic_list_order[1][i].idc );
76.                 bs_write_ue( s, sh->ref_pic_list_order[1][i].arg );
77.             }
78.             bs_write_ue( s, 3 );
79.         }
80.     }
81.
82.     sh->b_weighted_pred = 0;
83.     if( sh->pps->b_weighted_pred && sh->i_type == SLICE_TYPE_P )
84.     {
85.         sh->b_weighted_pred = sh->weight[0][0].weightfn || sh->weight[0][1].weightfn || sh->weight[0][2].weightfn;
86.         /* pred_weight_table() */
87.         bs_write_ue( s, sh->weight[0][0].i_denom );
88.         bs_write_ue( s, sh->weight[0][1].i_denom );
89.         for( int i = 0; i < sh->i_num_ref_idx_l0_active; i++ )
90.         {
91.             int luma_weight_l0_flag = !!sh->weight[i][0].weightfn;
92.             int chroma_weight_l0_flag = !!sh->weight[i][1].weightfn || !!sh->weight[i][2].weightfn;
93.             bs_writel( s, luma_weight_l0_flag );
94.             if( luma_weight_l0_flag )
95.             {
96.                 bs_write_se( s, sh->weight[i][0].i_scale );
97.                 bs_write_se( s, sh->weight[i][0].i_offset );
98.             }
99.             bs_writel( s, chroma_weight_l0_flag );
100.            if( chroma_weight_l0_flag )
101.            {
102.                for( int j = 1; j < 3; j++ )
103.                {
104.                    bs_write_se( s, sh->weight[i][j].i_scale );
105.                    bs_write_se( s, sh->weight[i][j].i_offset );
106.                }
107.            }
108.        }
109.    }
110.    else if( sh->pps->b_weighted_bipred == 1 && sh->i_type == SLICE_TYPE_B )
111.    {
112.        /* TODO */
113.    }
114.
115.    if( i_nal_ref_idc != 0 )
116.    {
117.        if( sh->i_idr_pic_id >= 0 )
118.        {
119.            bs_writel( s, 0 ); /* no output of prior pics flag */
120.            bs_writel( s, 0 ); /* long term reference flag */
121.        }
122.        else
123.        {
124.            bs_writel( s, sh->i_mmco_command_count > 0 ); /* adaptive_ref_pic_marking_mode_flag */
125.            if( sh->i_mmco_command_count > 0 )
126.            {
127.                for( int i = 0; i < sh->i_mmco_command_count; i++ )
128.                {
129.                    bs_write_ue( s, 1 ); /* mark short term ref as unused */
130.                    bs_write_ue( s, sh->mmco[i].i_difference_of_pic_nums - 1 );
131.                }
132.                bs_write_ue( s, 0 ); /* end command list */
133.            }
134.        }
135.    }
136.
137.    if( sh->pps->b_cabac && sh->i_type != SLICE_TYPE_I )
138.        bs_write_ue( s, sh->i_cabac_init_idc );
139.
140.    //slice_qp_delta: 指出在用于当前片的所有宏块的量化参数的初始值
141.    //SliceQP = 26 + pic_init_qp_minus26 + slice_qp_delta
142.    bs_write_se( s, sh->i_qp_delta ); /* slice qp delta */

```

```

143.
144.     if( sh->pps->b_deblocking_filter_control )
145.     {
146.         bs_write_ue( s, sh->i_disable_deblocking_filter_idc );
147.         if( sh->i_disable_deblocking_filter_idc != 1 )
148.         {
149.             bs_write_se( s, sh->i_alpha_c0_offset >> 1 );
150.             bs_write_se( s, sh->i_beta_offset >> 1 );
151.         }
152.     }
153. }

```

有关x264\_slice\_header\_write()的源代码不再做详细的分析。其中Slice Header的结构参考《H.264标准》即可。

## x264\_fdec\_filter\_row()

x264\_fdec\_filter\_row()属于滤波模块，完成几种滤波工作：

- (1) 半像素内插
- (2) 环路滤波
- (3) PSNR/SSIM计算

下面简单记录一下半像素内插和环路滤波的概念（后续文章再对源代码进行分析）。

### (1) 半像素插值知识简述

简单记录一下半像素插值的知识。《H.264标准》中规定，运动估计为1/4像素精度。因此在H.264编码和解码的过程中，需要将画面中的像素进行插值——简单地说就是把原先的1个像素点拓展成4x4一共16个点。下图显示了H.264编码和解码过程中像素插值情况。可以看出原先的G点的右下方通过插值的方式产生了a、b、c、d等一共16个点。

□

如图所示，1/4像素内插一般分成两步：

- (1) 半像素内插。这一步通过6抽头滤波器获得5个半像素点。
- (2) 线性内插。这一步通过简单的线性内插获得剩余的1/4像素点。

图中半像素内插点为b、m、h、s、j五个点。半像素内插方法是对整像素点进行6抽头滤波得出，滤波器的权重为(1/32, -5/32, 5/8, 5/8, -5/32, 1/32)。例如b的计算公式为：

$$b = \text{round}((E - 5F + 20G + 20H - 5I + J) / 32)$$

剩下几个半像素点的计算关系如下：

m：由B、D、H、N、S、U计算

h：由A、C、G、M、R、T计算

s：由K、L、M、N、P、Q计算

j：由cc、dd、h、m、ee、ff计算。需要注意j点的运算量比较大，因为cc、dd、ee、ff都需要通过半像素内插方法进行计算。

在获得半像素点之后，就可以通过简单的线性内插获得1/4像素内插点了。1/4像素内插的方式如下图所示。例如图中a点的计算公式如下：

$$A = \text{round}((G+b)/2)$$

在这里有一点需要注意：位于4个角的e、g、p、r四个点并不是通过j点计算计算的，而是通过b、h、s、m四个半像素点计算的。

□

### (2) 环路滤波相关知识简述

简单记录一下环路滤波（去块效应滤波）的知识。X264的重建帧（通过解码得到）一般情况下会出现方块效应。产生这种效应的原因主要有两个：

- (1) DCT变换后的量化造成误差（主要原因）。
- (2) 运动补偿

正是由于这种块效应的存在，才需要添加环路滤波器调整相邻的“块”边缘上的像素值以减轻这种视觉上的不连续感。下面一张图显示了环路滤波的效果。图中左边的图没有使用环路滤波，而右边的图使用了环路滤波。

□

#### 环路滤波分类

环路滤波器根据滤波的强度可以分为两种：

(1) 普通滤波器。针对边界的Bs（边界强度）为1、2、3的滤波器。此时环路滤波涉及到方块边界周围的6个点（边界两边各3个点）：p2, p1, p0, q0, q1, q2。需要处理4个点（边界两边各2个点，只以p点为例）：

$$\begin{aligned}
 p0' &= p0 + (((q0 - p0) << 2) + (p1 - q1) + 4) >> 3 \\
 p1' &= (p2 + ((p0 + q0 + 1) >> 1) - 2p1) >> 1
 \end{aligned}$$

(2) 强滤波器。针对边界的Bs（边界强度）为4的滤波器。此时环路滤波涉及到方块边界周围的8个点（边界两边各4个点）：p3, p2, p1, p0, q0, q1, q2, q3。需要处理6个点（边界两边各3个点，只以p点为例）：

$$\begin{aligned}
 p0' &= (p2 + 2*p1 + 2*p0 + 2*q0 + q1 + 4) >> 3 \\
 p1' &= (p2 + p1 + p0 + q0 + 2) >> 2 \\
 p2' &= (2*p3 + 3*p2 + p1 + p0 + q0 + 4) >> 3
 \end{aligned}$$

其中上文中提到的边界强度Bs的判定方式如下。

条件（针对两边的图像块）	Bs
有一个块为帧内预测 + 边界为宏块边界	4
有一个块为帧内预测	3
有一个块对残差编码	2
运动矢量差不小于1像素	1
运动补偿参考帧不同	1
其它	0

总体说来，与帧内预测相关的图像块（帧内预测块）的边界强度比较大，取值为3或者4；与运动补偿相关的图像块（帧间预测块）的边界强度比较小，取值为1。

### 环路滤波的门限

并不是所有的块的边界处都需要环路滤波。例如画面中物体的边界正好和块的边界重合的话，就不能进行滤波，否则会使画面中物体的边界变模糊。因此需要区别开物体边界和块效应边界。一般情况下，物体边界两边的像素值差别很大，而块效应边界两边像素值差别比较小。《H.264标准》以这个特点定义了2个变量alpha和beta来判决边界是否需要进行环路滤波。只有满足下面三个条件的时候才能进行环路滤波：

$$\begin{aligned} &|p0 - q0| < \alpha \\ &|p1 - p0| < \beta \\ &|q1 - q0| < \beta \end{aligned}$$

简而言之，就是边界两边的两个点的像素值不能太大，即不能超过alpha；边界一边的前两个点之间的像素值也不能太大，即不能超过beta。其中alpha和beta是根据量化参数QP推算出来（具体方法不再记录）。总体说来QP越大，alpha和beta的值也越大，也就越容易触发环路滤波。由于QP越大表明压缩的程度越大，所以也可以得知高压缩比的情况下更需要进行环路滤波。

## x264\_macroblock\_cache\_load()

x264\_slice\_write()根据是否包含隔行扫描，会分别调用x264\_macroblock\_cache\_load\_progressive()或者x264\_macroblock\_cache\_load\_interlaced()加载当前宏块的周边宏块的信息。这两个函数都会调用同一个函数x264\_macroblock\_cache\_load()。上述两个函数的定义位于common\macroblock.c，如下所示。

```
[cpp]
1. //加载Cache-逐行扫描
2. //即将要编码的宏块的周围的宏块的值读进来
3. void x264_macroblock_cache_load_progressive( x264_t *h, int mb_x, int mb_y )
4. {
5.     x264_macroblock_cache_load( h, mb_x, mb_y, 0 );
6. }
7. //加载Cache-隔行扫描
8. void x264_macroblock_cache_load_interlaced( x264_t *h, int mb_x, int mb_y )
9. {
10.     x264_macroblock_cache_load( h, mb_x, mb_y, 1 );
11. }
```

x264\_macroblock\_cache\_load()的定义位于common\macroblock.c，如下所示。

```
[cpp]
1. //加载Cache
2. //即将要编码的宏块的周围的宏块的值读进来
3. static void ALWAYS_INLINE x264_macroblock_cache_load( x264_t *h, int mb_x, int mb_y, int b_mbaff )
4. {
5.     x264_macroblock_cache_load_neighbours( h, mb_x, mb_y, b_mbaff );
6.
7.     //左边宏块
8.     int *left = h->mb.i_mb_left_xy;
9.     //上边宏块
10.    int top = h->mb.i_mb_top_xy;
11.    int top_y = h->mb.i_mb_top_y;
12.    int s8x8 = h->mb.i_b8_stride;
13.    int s4x4 = h->mb.i_b4_stride;
14.    int top_8x8 = (2*top_y+1) * s8x8 + 2*mb_x;
15.    int top_4x4 = (4*top_y+3) * s4x4 + 4*mb_x;
16.    int lists = (1 << h->sh.i_type) & 3;
17.
18.    /* GCC pessimizes direct loads from heap-allocated arrays due to aliasing. */
19.    /* By only dereferencing them once, we avoid this issue. */
20.    int8_t (*i4x4)[8] = h->mb.intra4x4_pred_mode;
21.    //DCT非0系数个数
22.    uint8_t (*nnz)[48] = h->mb.non_zero_count;
23.    //CBP值
24.    int16_t *cbp = h->mb.cbp;
25.
26.    const x264_left_table_t *left_index_table = h->mb.left_index_table;
27.
28.    h->mb.cache.deblock_strength = h->deblock_strength[mb_y&1][h->param.b_sliced_threads?h->mb.i_xy:mb_x];
29.
30.    /*
31.     *
```



```

31.      *
32.      * 关于多次出现的scan8
33.      *
34.      * scan8是和cache配合使用的
35.      * cache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亮度块”（H.264中最小的亮度处理单位）。
36.      * scan8[]则存储了宏块信息在cache中的索引值
37.      *
38.      * scan8[]中的“8”，意思应该是按照8x8为单元来扫描？
39.      * 因此可以理解为“按照8x8为单元来扫描4x4的块”？
40.      *
41.      * scan8中按照顺序分别存储了Y, U, V信息在cache中的索引值。具体的存储还是在相应的cache中。
42.      *
43.      * 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。
44.      * 其中数字代表了scan8[]中元素的索引值
45.      *
46.      * +---+---+---+---+---+---+---+---+
47.      * |   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
48.      * +---+---+---+---+---+---+---+---+
49.      * | 0 | 48|   |   |   | y| y| y| y|
50.      * | 1 |   |   |   |   | y| 0| 1| 4| 5|
51.      * | 2 |   |   |   |   | y| 2| 3| 6| 7|
52.      * | 3 |   |   |   |   | y| 8| 9| 12| 13|
53.      * | 4 |   |   |   |   | y| 10| 11| 14| 15|
54.      * | 5 | 49|   |   |   | u| u| u| u|
55.      * | 6 |   |   |   |   | u| 16| 17| 20| 21|
56.      * | 7 |   |   |   |   | u| 18| 19| 22| 23|
57.      * | 8 |   |   |   |   | u| 24| 25| 28| 29|
58.      * | 9 |   |   |   |   | u| 26| 27| 30| 31|
59.      * |10 | 50|   |   |   | v| v| v| v|
60.      * |11 |   |   |   |   | v| 32| 33| 36| 37|
61.      * |12 |   |   |   |   | v| 34| 35| 38| 39|
62.      * |13 |   |   |   |   | v| 40| 41| 44| 45|
63.      * |14 |   |   |   |   | v| 42| 43| 46| 47|
64.      * +---+---+---+---+---+---+---+---+
65.      * |   |
66.      *
67.      * 扫描方式：
68.      * 0-0 0-0
69.      * / / /
70.      * 0-0 0-0
71.      * ,---'
72.      * 0-0 0-0
73.      * / / /
74.      * 0-0 0-0
75.      *
76.      */
77.
78.      /* load cache */
79.      if( h->mb.i_neighbour & MB_TOP )
80.      {
81.          h->mb.cache.i_cbp_top = cbp[top];
82.          /* load intra4x4 */
83.          /*
84.           * 填充intra4x4_pred_mode[]
85.           * 在这里相当于在intra4x4_pred_mode[]填充了“y”，如下所示（没有U、V）
86.           * |
87.           * +---+---+---+---+
88.           * | 0 0 0 0 y y y y
89.           * | 0 0 0 0 Y Y Y Y
90.           * | 0 0 0 0 Y Y Y Y
91.           * | 0 0 0 0 Y Y Y Y
92.           * | 0 0 0 0 Y Y Y Y
93.           */
94.          CP32( &h->mb.cache.intra4x4_pred_mode[x264_scan8[0] - 8], &i4x4[top][0] );
95.
96.          /* load non_zero_count */
97.          /*
98.           * 填充non_zero_count[]
99.           * 在这里相当于在non_zero_count[]填充了“y”，如下所示（只列出了Y。U、V是类似的）
100.          * |
101.          * +---+---+---+---+
102.          * | 0 0 0 0 y y y y
103.          * | 0 0 0 0 Y Y Y Y
104.          * | 0 0 0 0 Y Y Y Y
105.          * | 0 0 0 0 Y Y Y Y
106.          * | 0 0 0 0 Y Y Y Y
107.          */
108.          CP32( &h->mb.cache.non_zero_count[x264_scan8[0] - 8], &nnz[top][12] ); //Y
109.          CP32( &h->mb.cache.non_zero_count[x264_scan8[16] - 8], &nnz[top][16-4 + (16>>CHROMA_V_SHIFT)] ); //U
110.          CP32( &h->mb.cache.non_zero_count[x264_scan8[32] - 8], &nnz[top][32-4 + (16>>CHROMA_V_SHIFT)] ); //V
111.
112.          /* Finish the prefetching */
113.          for( int l = 0; l < lists; l++ )
114.          {
115.              x264_prefetch( &h->mb.mv[l][top_4x4-1] );
116.              /* Top right being not in the same cacheline as top left will happen
117.               * once every 4 MBs, so one extra prefetch is worthwhile */
118.              x264_prefetch( &h->mb.mv[l][top_4x4+4] );
119.              x264_prefetch( &h->mb.ref[l][top_8x8-1] );
120.              x264_prefetch( &h->mb.mvd[l][top] );
121.          }
122.

```

```

123.     else
124.     {
125.         //没有相关信息的时候，填充下列数据
126.
127.         h->mb.cache.i_cbp_top = -1;
128.
129.         /* load intra4x4 */
130.         M32( &h->mb.cache.intra4x4_pred_mode[x264_scan8[0] - 8] ) = 0xFFFFFFFFU;
131.
132.         /* load non_zero_count */
133.         M32( &h->mb.cache.non_zero_count[x264_scan8[0] - 8] ) = 0x80808080U;
134.         M32( &h->mb.cache.non_zero_count[x264_scan8[16] - 8] ) = 0x80808080U;
135.         M32( &h->mb.cache.non_zero_count[x264_scan8[32] - 8] ) = 0x80808080U;
136.     }
137.
138.     if( h->mb.i_neighbour & MB_LEFT )
139.     {
140.         int ltop = left[LTOP];
141.         int lbot = b_mbaff ? left[LBOT] : ltop;
142.         if( b_mbaff )
143.         {
144.             const int16_t top_luma = (cbp[ltop] >> (left_index_table->mv[0]&(~1))) & 2;
145.             const int16_t bot_luma = (cbp[lbot] >> (left_index_table->mv[2]&(~1))) & 2;
146.             h->mb.cache.i_cbp_left = (cbp[ltop] & 0xfff0) | (bot_luma<2) | top_luma;
147.         }
148.         else
149.             h->mb.cache.i_cbp_left = cbp[ltop];
150.
151.         /* load intra4x4 */
152.         /*
153.          * 填充intra4x4_pred_mode[]
154.          * 在这里相当于在intra4x4_pred_mode[]填充了“y”，如下所示（没有U、V）
155.          * |
156.          * ---+-----
157.          * | 0 0 0 0 0 0 0 0
158.          * | 0 0 0 y Y Y Y Y
159.          * | 0 0 0 y Y Y Y Y
160.          * | 0 0 0 y Y Y Y Y
161.          * | 0 0 0 y Y Y Y Y
162.          */
163.         h->mb.cache.intra4x4_pred_mode[x264_scan8[0] - 1] = i4x4[ltop][left_index_table->intra[0]];
164.         h->mb.cache.intra4x4_pred_mode[x264_scan8[2] - 1] = i4x4[ltop][left_index_table->intra[1]];
165.         h->mb.cache.intra4x4_pred_mode[x264_scan8[8] - 1] = i4x4[lbot][left_index_table->intra[2]];
166.         h->mb.cache.intra4x4_pred_mode[x264_scan8[10] - 1] = i4x4[lbot][left_index_table->intra[3]];
167.
168.         /* load non_zero_count */
169.         /*
170.          * 填充non_zero_count[]
171.          * 在这里相当于在non_zero_count[]填充了“y”，如下所示（只列出了Y，U、V是类似的）
172.          * |
173.          * ---+-----
174.          * | 0 0 0 0 0 0 0 0
175.          * | 0 0 0 y Y Y Y Y
176.          * | 0 0 0 y Y Y Y Y
177.          * | 0 0 0 y Y Y Y Y
178.          * | 0 0 0 y Y Y Y Y
179.          */
180.         h->mb.cache.non_zero_count[x264_scan8[0] - 1] = nnz[ltop][left_index_table->nnz[0]];
181.         h->mb.cache.non_zero_count[x264_scan8[2] - 1] = nnz[ltop][left_index_table->nnz[1]];
182.         h->mb.cache.non_zero_count[x264_scan8[8] - 1] = nnz[lbot][left_index_table->nnz[2]];
183.         h->mb.cache.non_zero_count[x264_scan8[10] - 1] = nnz[lbot][left_index_table->nnz[3]];
184.
185.         if( CHROMA_FORMAT >= CHROMA_422 )
186.         {
187.             int offset = (4>>CHROMA_H_SHIFT) - 4;
188.             h->mb.cache.non_zero_count[x264_scan8[16+0] - 1] = nnz[ltop][left_index_table->nnz[0]+16+offset];
189.             h->mb.cache.non_zero_count[x264_scan8[16+2] - 1] = nnz[ltop][left_index_table->nnz[1]+16+offset];
190.             h->mb.cache.non_zero_count[x264_scan8[16+8] - 1] = nnz[lbot][left_index_table->nnz[2]+16+offset];
191.             h->mb.cache.non_zero_count[x264_scan8[16+10] - 1] = nnz[lbot][left_index_table->nnz[3]+16+offset];
192.             h->mb.cache.non_zero_count[x264_scan8[32+0] - 1] = nnz[ltop][left_index_table->nnz[0]+32+offset];
193.             h->mb.cache.non_zero_count[x264_scan8[32+2] - 1] = nnz[ltop][left_index_table->nnz[1]+32+offset];
194.             h->mb.cache.non_zero_count[x264_scan8[32+8] - 1] = nnz[lbot][left_index_table->nnz[2]+32+offset];
195.             h->mb.cache.non_zero_count[x264_scan8[32+10] - 1] = nnz[lbot][left_index_table->nnz[3]+32+offset];
196.         }
197.         else
198.         {
199.             h->mb.cache.non_zero_count[x264_scan8[16+0] - 1] = nnz[ltop][left_index_table->nnz_chroma[0]];
200.             h->mb.cache.non_zero_count[x264_scan8[16+2] - 1] = nnz[lbot][left_index_table->nnz_chroma[1]];
201.             h->mb.cache.non_zero_count[x264_scan8[32+0] - 1] = nnz[ltop][left_index_table->nnz_chroma[2]];
202.             h->mb.cache.non_zero_count[x264_scan8[32+2] - 1] = nnz[lbot][left_index_table->nnz_chroma[3]];
203.         }
204.     }
205.     else
206.     {
207.         //没有相关信息的时候，填充下列数据
208.
209.         h->mb.cache.i_cbp_left = -1;
210.
211.         h->mb.cache.intra4x4_pred_mode[x264_scan8[0] - 1] =
212.         h->mb.cache.intra4x4_pred_mode[x264_scan8[2] - 1] =
213.         h->mb.cache.intra4x4_pred_mode[x264_scan8[8] - 1] =

```

```

214.     h->mb.cache.intra4x4_pred_mode[x264_scan8[10] - 1] = -1;
215.
216.     /* load non_zero_count */
217.     h->mb.cache.non_zero_count[x264_scan8[ 0] - 1] =
218.     h->mb.cache.non_zero_count[x264_scan8[ 2] - 1] =
219.     h->mb.cache.non_zero_count[x264_scan8[ 8] - 1] =
220.     h->mb.cache.non_zero_count[x264_scan8[10] - 1] =
221.     h->mb.cache.non_zero_count[x264_scan8[16+ 0] - 1] =
222.     h->mb.cache.non_zero_count[x264_scan8[16+ 2] - 1] =
223.     h->mb.cache.non_zero_count[x264_scan8[32+ 0] - 1] =
224.     h->mb.cache.non_zero_count[x264_scan8[32+ 2] - 1] = 0x80;
225.     if( CHROMA_FORMAT >= CHROMA_422 )
226.     {
227.         h->mb.cache.non_zero_count[x264_scan8[16+ 8] - 1] =
228.         h->mb.cache.non_zero_count[x264_scan8[16+10] - 1] =
229.         h->mb.cache.non_zero_count[x264_scan8[32+ 8] - 1] =
230.         h->mb.cache.non_zero_count[x264_scan8[32+10] - 1] = 0x80;
231.     }
232. }
233.
234. if( h->pps->b_transform_8x8_mode )
235. {
236.     h->mb.cache.i_neighbour_transform_size =
237.         ( (h->mb.i_neighbour & MB_LEFT) && h->mb.mb_transform_size[left[0]] )
238.         + ( (h->mb.i_neighbour & MB_TOP) && h->mb.mb_transform_size[top] );
239. }
240.
241. if( b_mbaff )
242. {
243.     h->mb.pic.i_fref[0] = h->i_ref[0] << MB_INTERLACED;
244.     h->mb.pic.i_fref[1] = h->i_ref[1] << MB_INTERLACED;
245. }
246.
247. if( !b_mbaff )
248. {
249.     //没有“宏块级帧场自适应”情况的时候
250.
251.     //亮度
252.     //拷贝上一个宏块最右边一列（共16个）像素（p_fdec[0]+15）
253.     //作为这一个宏块最左边再靠左的一列像素（p_fdec[0]-1）
254.     //一次拷贝8个（起始点上面4个下面4个），拷贝2次
255.
256.     x264_copy_column8( h->mb.pic.p_fdec[0]-1+ 4*FDEC_STRIDE, h->mb.pic.p_fdec[0]+15+ 4*FDEC_STRIDE );
257.     x264_copy_column8( h->mb.pic.p_fdec[0]-1+12*FDEC_STRIDE, h->mb.pic.p_fdec[0]+15+12*FDEC_STRIDE );
258.     //加载图像相关的指针
259.     //第4个参数：指明了第几个分量（Y、U、V）
260.     //第5个参数：指明了是否为色度
261.     x264_macroblock_load_pic_pointers( h, mb_x, mb_y, 0, 0, 0 );
262.     if( CHROMA444 )
263.     {
264.         x264_copy_column8( h->mb.pic.p_fdec[1]-1+ 4*FDEC_STRIDE, h->mb.pic.p_fdec[1]+15+ 4*FDEC_STRIDE );
265.         x264_copy_column8( h->mb.pic.p_fdec[1]-1+12*FDEC_STRIDE, h->mb.pic.p_fdec[1]+15+12*FDEC_STRIDE );
266.         x264_copy_column8( h->mb.pic.p_fdec[2]-1+ 4*FDEC_STRIDE, h->mb.pic.p_fdec[2]+15+ 4*FDEC_STRIDE );
267.         x264_copy_column8( h->mb.pic.p_fdec[2]-1+12*FDEC_STRIDE, h->mb.pic.p_fdec[2]+15+12*FDEC_STRIDE );
268.         x264_macroblock_load_pic_pointers( h, mb_x, mb_y, 1, 0, 0 );
269.         x264_macroblock_load_pic_pointers( h, mb_x, mb_y, 2, 0, 0 );
270.     }
271.     else
272.     {
273.         //U和V
274.         //YUV420P的情况下
275.         //拷贝上一个宏块最右边一列（共8个）像素
276.         //作为这一个宏块最左边再靠左的一列像素
277.         //一次拷贝8个
278.         x264_copy_column8( h->mb.pic.p_fdec[1]-1+ 4*FDEC_STRIDE, h->mb.pic.p_fdec[1]+ 7+ 4*FDEC_STRIDE );
279.         x264_copy_column8( h->mb.pic.p_fdec[2]-1+ 4*FDEC_STRIDE, h->mb.pic.p_fdec[2]+ 7+ 4*FDEC_STRIDE );
280.         if( CHROMA_FORMAT == CHROMA_422 )
281.         {
282.             x264_copy_column8( h->mb.pic.p_fdec[1]-1+12*FDEC_STRIDE, h->mb.pic.p_fdec[1]+ 7+12*FDEC_STRIDE );
283.             x264_copy_column8( h->mb.pic.p_fdec[2]-1+12*FDEC_STRIDE, h->mb.pic.p_fdec[2]+ 7+12*FDEC_STRIDE );
284.         }
285.         x264_macroblock_load_pic_pointers( h, mb_x, mb_y, 1, 1, 0 );
286.     }
287. }
288. else
289. {
290.     x264_macroblock_load_pic_pointers( h, mb_x, mb_y, 0, 0, 1 );
291.     if( CHROMA444 )
292.     {
293.         x264_macroblock_load_pic_pointers( h, mb_x, mb_y, 1, 0, 1 );
294.         x264_macroblock_load_pic_pointers( h, mb_x, mb_y, 2, 0, 1 );
295.     }
296.     else
297.         x264_macroblock_load_pic_pointers( h, mb_x, mb_y, 1, 1, 1 );
298. }
299.
300. if( h->fdec->integral )
301. {
302.     int offset = 16 * (mb_x + mb_y * h->fdec->i_stride[0]);
303.     for( int list = 0; list < 2; list++ )
304.         for( int i = 0; i < h->mb.pic.i_fref[list]; i++ )

```

```

305.         h->mb.pic.p_integral[list][i] = &h->fref[list][i]->integral[offset];
306.     }
307.
308.     x264_prefetch_fenc( h, h->fenc, mb_x, mb_y );
309.
310.     /* load ref/mv/mvd */
311.     for( int l = 0; l < lists; l++ )
312.     {
313.         int16_t (*mv)[2] = h->mb.mv[l];
314.         int8_t *ref = h->mb.ref[l];
315.
316.         int i8 = x264_scan8[0] - 1 - 1*8;
317.         if( h->mb.i_neighbour & MB_TOPLEFT )
318.         {
319.             //填充宏块左上方信息
320.
321.             int ir = b_mbaff ? 2*(s8x8*h->mb.i_mb_topleft_y + mb_x-1)+1+s8x8 : top_8x8 - 1;
322.             int iv = b_mbaff ? 4*(s4x4*h->mb.i_mb_topleft_y + mb_x-1)+3+s4x4 : top_4x4 - 1;
323.             if( b_mbaff && h->mb.topleft_partition )
324.             {
325.                 /* Take motion vector from the middle of macroblock instead of
326.                  * the bottom right as usual. */
327.                 iv -= 2*s4x4;
328.                 ir -= s8x8;
329.             }
330.             /*
331.              * 填充参考帧序号ref[]
332.              * 在这里相当于在ref[]填充了“y”,
333.              * |
334.              * ---+-----
335.              * | 0 0 0 y 0 0 0 0
336.              * | 0 0 0 0 Y Y Y Y
337.              * | 0 0 0 0 Y Y Y Y
338.              * | 0 0 0 0 Y Y Y Y
339.              * | 0 0 0 0 Y Y Y Y
340.              */
341.             //参考帧序号
342.             h->mb.cache.ref[l][i8] = ref[ir];
343.             /*
344.              * 填充运动矢量mv[]
345.              * 在这里相当于在mv[]填充了“y”,
346.              * |
347.              * ---+-----
348.              * | 0 0 0 y 0 0 0 0
349.              * | 0 0 0 0 Y Y Y Y
350.              * | 0 0 0 0 Y Y Y Y
351.              * | 0 0 0 0 Y Y Y Y
352.              * | 0 0 0 0 Y Y Y Y
353.              */
354.             //运动矢量
355.             CP32( h->mb.cache.mv[l][i8], mv[iv] );
356.         }
357.         else
358.         {
359.             h->mb.cache.ref[l][i8] = -2;
360.             M32( h->mb.cache.mv[l][i8] ) = 0;
361.         }
362.
363.         i8 = x264_scan8[0] - 8;
364.         if( h->mb.i_neighbour & MB_TOP )
365.         {
366.             //填充宏块上方信息
367.
368.             /*
369.              * 填充参考帧序号ref[]
370.              * 在这里相当于在ref[]分别填充了“1”和“2”,
371.              * |
372.              * ---+-----
373.              * | 0 0 0 0 1 1 2 2
374.              * | 0 0 0 0 Y Y Y Y
375.              * | 0 0 0 0 Y Y Y Y
376.              * | 0 0 0 0 Y Y Y Y
377.              * | 0 0 0 0 Y Y Y Y
378.              */
379.             h->mb.cache.ref[l][i8+0] =
380.             h->mb.cache.ref[l][i8+1] = ref[top_8x8 + 0];
381.             h->mb.cache.ref[l][i8+2] =
382.             h->mb.cache.ref[l][i8+3] = ref[top_8x8 + 1];
383.             /*
384.              * 填充运动矢量mv[]
385.              * 在这里相当于在mv[]填充了y,
386.              * |
387.              * ---+-----
388.              * | 0 0 0 0 y y y y
389.              * | 0 0 0 0 Y Y Y Y
390.              * | 0 0 0 0 Y Y Y Y
391.              * | 0 0 0 0 Y Y Y Y
392.              * | 0 0 0 0 Y Y Y Y
393.              */
394.             CP128( h->mb.cache.mv[l][i8], mv[top_4x4] );
395.         }

```

```

396.     else
397.     {
398.         M128( h->mb.cache.mv[l][i8] ) = M128_ZERO;
399.         M32( &h->mb.cache.ref[l][i8] ) = (uint8_t)(-2) * 0x01010101U;
400.     }
401.
402.     i8 = x264_scan8[0] + 4 - 1*8;
403.     if( h->mb.i_neighbour & MB_TOPRIGHT )
404.     {
405.         //填充宏块右上方信息
406.         int ir = b_mbaff ? 2*(s8x8*h->mb.i_mb_topright_y + (mb_x+1))+s8x8 : top_8x8 + 2;
407.         int iv = b_mbaff ? 4*(s4x4*h->mb.i_mb_topright_y + (mb_x+1))+3*s4x4 : top_4x4 + 4;
408.         h->mb.cache.ref[l][i8] = ref[ir];
409.         CP32( h->mb.cache.mv[l][i8], mv[iv] );
410.     }
411.     else
412.         h->mb.cache.ref[l][i8] = -2;
413.
414.     i8 = x264_scan8[0] - 1;
415.     if( h->mb.i_neighbour & MB_LEFT )
416.     {
417.         //填充宏块左边信息
418.
419.         if( b_mbaff )
420.         {
421.             h->mb.cache.ref[l][i8+0*8] = ref[h->mb.left_b8[LTOP] + 1 + s8x8*left_index_table->ref[0]];
422.             h->mb.cache.ref[l][i8+1*8] = ref[h->mb.left_b8[LTOP] + 1 + s8x8*left_index_table->ref[1]];
423.             h->mb.cache.ref[l][i8+2*8] = ref[h->mb.left_b8[LBOT] + 1 + s8x8*left_index_table->ref[2]];
424.             h->mb.cache.ref[l][i8+3*8] = ref[h->mb.left_b8[LBOT] + 1 + s8x8*left_index_table->ref[3]];
425.
426.             CP32( h->mb.cache.mv[l][i8+0*8], mv[h->mb.left_b4[LTOP] + 3 + s4x4*left_index_table->mv[0]] );
427.             CP32( h->mb.cache.mv[l][i8+1*8], mv[h->mb.left_b4[LTOP] + 3 + s4x4*left_index_table->mv[1]] );
428.             CP32( h->mb.cache.mv[l][i8+2*8], mv[h->mb.left_b4[LBOT] + 3 + s4x4*left_index_table->mv[2]] );
429.             CP32( h->mb.cache.mv[l][i8+3*8], mv[h->mb.left_b4[LBOT] + 3 + s4x4*left_index_table->mv[3]] );
430.         }
431.         else
432.         {
433.             //不考虑“宏块级帧场自适应”的时候
434.
435.             const int ir = h->mb.i_b8_xy - 1;
436.             const int iv = h->mb.i_b4_xy - 1;
437.
438.             /*
439.              * 填充参考帧序号ref[]
440.              * 在这里相当于在ref[]分别填充了“1”和“2”，
441.              * |
442.              * --+-----
443.              * | 0 0 0 0 0 0 0 0
444.              * | 0 0 0 1 Y Y Y Y
445.              * | 0 0 0 1 Y Y Y Y
446.              * | 0 0 0 2 Y Y Y Y
447.              * | 0 0 0 2 Y Y Y Y
448.              */
449.             h->mb.cache.ref[l][i8+0*8] =
450.             h->mb.cache.ref[l][i8+1*8] = ref[ir + 0*s8x8];
451.             h->mb.cache.ref[l][i8+2*8] =
452.             h->mb.cache.ref[l][i8+3*8] = ref[ir + 1*s8x8];
453.
454.             /*
455.              * 填充运动矢量mv[]
456.              * 在这里相当于在mv[]填充了y，
457.              * |
458.              * --+-----
459.              * | 0 0 0 0 0 0 0 0
460.              * | 0 0 0 y Y Y Y Y
461.              * | 0 0 0 y Y Y Y Y
462.              * | 0 0 0 y Y Y Y Y
463.              * | 0 0 0 y Y Y Y Y
464.              */
465.             CP32( h->mb.cache.mv[l][i8+0*8], mv[iv + 0*s4x4] );
466.             CP32( h->mb.cache.mv[l][i8+1*8], mv[iv + 1*s4x4] );
467.             CP32( h->mb.cache.mv[l][i8+2*8], mv[iv + 2*s4x4] );
468.             CP32( h->mb.cache.mv[l][i8+3*8], mv[iv + 3*s4x4] );
469.         }
470.     }
471.     else
472.     {
473.         for( int i = 0; i < 4; i++ )
474.         {
475.             h->mb.cache.ref[l][i8+i*8] = -2;
476.             M32( h->mb.cache.mv[l][i8+i*8] ) = 0;
477.         }
478.     }
479.
480.     /* Extra logic for top right mv in mbaff.
481.      * . . . d . . a .
482.      * . . . e . . . .
483.      * . . . f b . c .
484.      * . . . . . . .
485.      *
486.      * If the top right of the 4x4 partitions labeled a, b and c in the

```

```

487.     * above diagram do not exist, but the entries d, e and f exist (in
488.     * the macroblock to the left) then use those instead.
489.     */
490.     if( b_mbafl && (h->mb.i_neighbour & MB_LEFT) )
491.     {
492.         if( MB_INTERLACED && !h->mb.field[h->mb.i_mb_xy-1] )
493.         {
494.             h->mb.cache.topright_ref[l][0] = ref[h->mb.left_b8[0] + 1 + s8x8*0];
495.             h->mb.cache.topright_ref[l][1] = ref[h->mb.left_b8[0] + 1 + s8x8*1];
496.             h->mb.cache.topright_ref[l][2] = ref[h->mb.left_b8[1] + 1 + s8x8*0];
497.             CP32( h->mb.cache.topright_mv[l][0], mv[h->mb.left_b4[0] + 3 + s4x4*(left_index_table->mv[0]+1)] );
498.             CP32( h->mb.cache.topright_mv[l][1], mv[h->mb.left_b4[0] + 3 + s4x4*(left_index_table->mv[1]+1)] );
499.             CP32( h->mb.cache.topright_mv[l][2], mv[h->mb.left_b4[1] + 3 + s4x4*(left_index_table->mv[2]+1)] );
500.         }
501.         else if( !MB_INTERLACED && h->mb.field[h->mb.i_mb_xy-1] )
502.         {
503.             // Looking at the bottom field so always take the bottom macroblock of the pair.
504.             h->mb.cache.topright_ref[l][0] = ref[h->mb.left_b8[0] + 1 + s8x8*2 + s8x8*left_index_table->ref[0]];
505.             h->mb.cache.topright_ref[l][1] = ref[h->mb.left_b8[0] + 1 + s8x8*2 + s8x8*left_index_table->ref[0]];
506.             h->mb.cache.topright_ref[l][2] = ref[h->mb.left_b8[0] + 1 + s8x8*2 + s8x8*left_index_table->ref[2]];
507.             CP32( h->mb.cache.topright_mv[l][0], mv[h->mb.left_b4[0] + 3 + s4x4*4 + s4x4*left_index_table->mv[0]] );
508.             CP32( h->mb.cache.topright_mv[l][1], mv[h->mb.left_b4[0] + 3 + s4x4*4 + s4x4*left_index_table->mv[1]] );
509.             CP32( h->mb.cache.topright_mv[l][2], mv[h->mb.left_b4[0] + 3 + s4x4*4 + s4x4*left_index_table->mv[2]] );
510.         }
511.     }
512.
513.     //使用了CABAC的时候才会运行
514.     if( h->param.b_cabac )
515.     {
516.         uint8_t (*mvd)[8][2] = h->mb.mvd[l];
517.         if( h->mb.i_neighbour & MB_TOP )
518.             CP64( h->mb.cache.mvd[l][x264_scan8[0] - 8], mvd[top][0] );
519.         else
520.             M64( h->mb.cache.mvd[l][x264_scan8[0] - 8] ) = 0;
521.
522.         if( h->mb.i_neighbour & MB_LEFT && (!b_mbafl || h->mb.cache.ref[l][x264_scan8[0]-1] >= 0) )
523.         {
524.             CP16( h->mb.cache.mvd[l][x264_scan8[0] - 1], mvd[left[LTOP]][left_index_table->intra[0]] );
525.             CP16( h->mb.cache.mvd[l][x264_scan8[2] - 1], mvd[left[LTOP]][left_index_table->intra[1]] );
526.         }
527.         else
528.         {
529.             M16( h->mb.cache.mvd[l][x264_scan8[0]-1+0*8] ) = 0;
530.             M16( h->mb.cache.mvd[l][x264_scan8[0]-1+1*8] ) = 0;
531.         }
532.         if( h->mb.i_neighbour & MB_LEFT && (!b_mbafl || h->mb.cache.ref[l][x264_scan8[0]-1+2*8] >=0) )
533.         {
534.             CP16( h->mb.cache.mvd[l][x264_scan8[8] - 1], mvd[left[LBOT]][left_index_table->intra[2]] );
535.             CP16( h->mb.cache.mvd[l][x264_scan8[10] - 1], mvd[left[LBOT]][left_index_table->intra[3]] );
536.         }
537.         else
538.         {
539.             M16( h->mb.cache.mvd[l][x264_scan8[0]-1+2*8] ) = 0;
540.             M16( h->mb.cache.mvd[l][x264_scan8[0]-1+3*8] ) = 0;
541.         }
542.     }
543.
544.     /* If motion vectors are cached from frame macroblocks but this
545.     * macroblock is a field macroblock then the motion vector must be
546.     * halved. Similarly, motion vectors from field macroblocks are doubled. */
547.     if( b_mbafl )
548.     {
549. #define MAP_MVS\
550.         if( FIELD_DIFFERENT(h->mb.i_mb_topleft_xy) )\
551.             MAP_F2F(mv, ref, x264_scan8[0] - 1 - 1*8)\
552.         if( FIELD_DIFFERENT(top) )\
553.         {\
554.             MAP_F2F(mv, ref, x264_scan8[0] + 0 - 1*8)\
555.             MAP_F2F(mv, ref, x264_scan8[0] + 1 - 1*8)\
556.             MAP_F2F(mv, ref, x264_scan8[0] + 2 - 1*8)\
557.             MAP_F2F(mv, ref, x264_scan8[0] + 3 - 1*8)\
558.         }\
559.         if( FIELD_DIFFERENT(h->mb.i_mb_topright_xy) )\
560.             MAP_F2F(mv, ref, x264_scan8[0] + 4 - 1*8)\
561.         if( FIELD_DIFFERENT(left[0]) )\
562.         {\
563.             MAP_F2F(mv, ref, x264_scan8[0] - 1 + 0*8)\
564.             MAP_F2F(mv, ref, x264_scan8[0] - 1 + 1*8)\
565.             MAP_F2F(mv, ref, x264_scan8[0] - 1 + 2*8)\
566.             MAP_F2F(mv, ref, x264_scan8[0] - 1 + 3*8)\
567.             MAP_F2F(topright_mv, topright_ref, 0)\
568.             MAP_F2F(topright_mv, topright_ref, 1)\
569.             MAP_F2F(topright_mv, topright_ref, 2)\
570.         }
571.
572.         if( MB_INTERLACED )
573.         {
574. #define FIELD_DIFFERENT(macroblock) (macroblock >= 0 && !h->mb.field[macroblock])
575. #define MAP_F2F(varmv, varref, index)\
576.             if( h->mb.cache.varref[l][index] >= 0 )\
577.             {\

```

```

578.             h->mb.cache.varref[l][index] <= 1;\
579.             h->mb.cache.varmv[l][index][1] /= 2;\
580.             h->mb.cache.mvd[l][index][1] >= 1;\
581.         }
582.         MAP_MVS
583. #undef MAP_F2F
584. #undef FIELD_DIFFERENT
585.     }
586.     else
587.     {
588. #define FIELD_DIFFERENT(macroblock) (macroblock >= 0 && h->mb.field[macroblock])
589. #define MAP_F2F(varmv, varref, index)\
590.         if( h->mb.cache.varref[l][index] >= 0 )\
591.         {\
592.             h->mb.cache.varref[l][index] >= 1;\
593.             h->mb.cache.varmv[l][index][1] <= 1;\
594.             h->mb.cache.mvd[l][index][1] <= 1;\
595.         }
596.         MAP_MVS
597. #undef MAP_F2F
598. #undef FIELD_DIFFERENT
599.     }
600. }
601. }
602.
603. if( b_mbaff && mb_x == 0 && !(mb_y&1) )
604. {
605.     if( h->mb.i_mb_top_xy >= h->sh.i_first_mb )
606.         h->mb.field_decoding_flag = h->mb.field[h->mb.i_mb_top_xy];
607.     else
608.         h->mb.field_decoding_flag = 0;
609. }
610.
611. /* Check whether skip here would cause decoder to predict interlace mode incorrectly.
612.  * FIXME: It might be better to change the interlace type rather than forcing a skip to be non-skip. */
613. h->mb.b_allow_skip = 1;
614. if( b_mbaff )
615. {
616.     if( MB_INTERLACED != h->mb.field_decoding_flag &&
617.         (mb_y&1) && IS_SKIP(h->mb.type[h->mb.i_mb_xy - h->mb.i_mb_stride]) )
618.         h->mb.b_allow_skip = 0;
619. }
620.
621. //使用了CABAC的时候才会运行
622. if( h->param.b_cabac )
623. {
624.     if( b_mbaff )
625.     {
626.         int left_xy, top_xy;
627.         /* Neighbours here are calculated based on field_decoding_flag */
628.         int mb_xy = mb_x + (mb_y&1)*h->mb.i_mb_stride;
629.         left_xy = mb_xy - 1;
630.         if( (mb_y&1) && mb_x > 0 && h->mb.field_decoding_flag == h->mb.field[left_xy] )
631.             left_xy += h->mb.i_mb_stride;
632.         if( h->mb.field_decoding_flag )
633.         {
634.             top_xy = mb_xy - h->mb.i_mb_stride;
635.             if( !(mb_y&1) && top_xy >= 0 && h->mb.slice_table[top_xy] == h->sh.i_first_mb && h->mb.field[top_xy] )
636.                 top_xy -= h->mb.i_mb_stride;
637.         }
638.         else
639.             top_xy = mb_x + (mb_y-1)*h->mb.i_mb_stride;
640.
641.         h->mb.cache.i_neighbour_skip = (mb_x > 0 && h->mb.slice_table[left_xy] == h->sh.i_first_mb && !IS_SKIP( h->mb.type[le
ft_xy] ))
642.             + (top_xy >= 0 && h->mb.slice_table[top_xy] == h->sh.i_first_mb && !IS_SKIP( h->mb.type[top_xy] ));
643.     }
644.     else
645.     {
646.         h->mb.cache.i_neighbour_skip = ((h->mb.i_neighbour & MB_LEFT) && !IS_SKIP( h->mb.i_mb_type_left[0] ))
647.             + ((h->mb.i_neighbour & MB_TOP) && !IS_SKIP( h->mb.i_mb_type_top ));
648.     }
649. }
650.
651. /* load skip */
652. //处理"skip"类型宏块
653. if( h->sh.i_type == SLICE_TYPE_B )
654. {
655.     h->mb.bipred_weight = h->mb.bipred_weight_buf[MB_INTERLACED][MB_INTERLACED&(mb_y&1)];
656.     h->mb.dist_scale_factor = h->mb.dist_scale_factor_buf[MB_INTERLACED][MB_INTERLACED&(mb_y&1)];
657.     if( h->param.b_cabac )
658.     {
659.         uint8_t skipbp;
660.         x264_macroblock_cache_skip( h, 0, 0, 4, 4, 0 );
661.         if( b_mbaff )
662.         {
663.             skipbp = (h->mb.i_neighbour & MB_LEFT) ? h->mb.skipbp[left[LTOP]] : 0;
664.             h->mb.cache.skip[x264_scan8[0] - 1] = (skipbp >> (1+(left_index_table->mv[0]&~1))) & 1;
665.             skipbp = (h->mb.i_neighbour & MB_LEFT) ? h->mb.skipbp[left[LBOT]] : 0;
666.             h->mb.cache.skip[x264_scan8[8] - 1] = (skipbp >> (1+(left_index_table->mv[2]&~1))) & 1;
667.         }

```

```

668.         else
669.         {
670.             skipbp = (h->mb.i_neighbour & MB_LEFT) ? h->mb.skipbp[left[0]] : 0;
671.             h->mb.cache.skip[x264_scan8[0] - 1] = skipbp & 0x2;
672.             h->mb.cache.skip[x264_scan8[8] - 1] = skipbp & 0x8;
673.         }
674.         skipbp = (h->mb.i_neighbour & MB_TOP) ? h->mb.skipbp[top] : 0;
675.         h->mb.cache.skip[x264_scan8[0] - 8] = skipbp & 0x4;
676.         h->mb.cache.skip[x264_scan8[4] - 8] = skipbp & 0x8;
677.     }
678. }
679.
680. if( h->sh.i_type == SLICE_TYPE_P )
681.     x264_mb_predict_mv_pskip( h, h->mb.cache.pskip_mv );
682.
683. /*
684.  * i_neighbour8把一个宏块分成4个8x8的子块，编号如下，用于记录它们邻块的可用性
685.  * +-----+-----+
686.  * |         |         |
687.  * |    0    |    1    |
688.  * |         |         |
689.  * +-----+-----+
690.  * |         |         |
691.  * |    2    |    3    |
692.  * |         |         |
693.  * +-----+-----+
694.  *
695.  * i_neighbour4把一个宏块分成16个4x4的子块，编号如下，用于记录它们邻块的可用性
696.  * (实际上也是类似scan8[]读取cache的顺序)
697.  * +-----+-----+
698.  * | 0 | 1 | 4 | 5 |
699.  * +-----+-----+
700.  * | 2 | 3 | 6 | 7 |
701.  * +-----+-----+
702.  * | 8 | 9 |12|13|
703.  * +-----+-----+
704.  * |10|11|14|15|
705.  * +-----+-----+
706.  *
707.  */
708. h->mb.i_neighbour4[0] =
709. h->mb.i_neighbour8[0] = (h->mb.i_neighbour_intra & (MB_TOP|MB_LEFT|MB_TOPLEFT))
710. | ((h->mb.i_neighbour_intra & MB_TOP) ? MB_TOPRIGHT : 0);
711. h->mb.i_neighbour4[4] =
712. h->mb.i_neighbour4[1] = MB_LEFT | ((h->mb.i_neighbour_intra & MB_TOP) ? (MB_TOP|MB_TOPLEFT|MB_TOPRIGHT) : 0);
713. h->mb.i_neighbour4[2] =
714. h->mb.i_neighbour4[8] =
715. h->mb.i_neighbour4[10] =
716. h->mb.i_neighbour8[2] = MB_TOP|MB_TOPRIGHT | ((h->mb.i_neighbour_intra & MB_LEFT) ? (MB_LEFT|MB_TOPLEFT) : 0);
717. h->mb.i_neighbour4[5] =
718. h->mb.i_neighbour8[1] = MB_LEFT | (h->mb.i_neighbour_intra & MB_TOPRIGHT)
719. | ((h->mb.i_neighbour_intra & MB_TOP) ? MB_TOP|MB_TOPLEFT : 0);
720. }

```

x264\_macroblock\_cache\_load()源代码比较长，比较关键的地方都做了注释，在这里就不详细记录了。总体说来该函数的流程如下所示：

- (1) 加载Intra4x4帧内预测模式intra4x4\_pred\_mode[]和DCT非零系数non\_zero\_count[]缓存Cache的宏块周边信息。加载顺序为：上->左->左上。
- (2) 加载宏块重建像素p\_fdec[]的周边像素，以及宏块编码像素p\_fenc[]。对于p\_fdec[]来说，在本函数中直接加载当前宏块左边的像素；调用函数x264\_macroblock\_load\_pic\_pointers()加载当前宏块上面的像素。对于p\_fenc[]来说，调用x264\_macroblock\_load\_pic\_pointers()从图像上拷贝数据。
- (3) 加载参考帧序号ref[]和运动矢量mv[]缓存Cache的宏块周边信息。加载顺序为：左上->上->左。
- (4) 加载其它信息。

下面简单浏览一下x264\_macroblock\_load\_pic\_pointers()的源代码。

## x264\_macroblock\_load\_pic\_pointers()

x264\_macroblock\_load\_pic\_pointers()用于给宏块重建像素p\_fdec[]和宏块编码像素p\_fenc[]加载数据，并且加载图像的半像素数据。它的定义位于common\macroblock.c，如下所示。

```

[cpp]
1. //加载图像相关的指针
2. static void ALWAYS_INLINE x264_macroblock_load_pic_pointers( x264_t *h, int mb_x, int mb_y, int i, int b_chroma, int b_mbaff )
3. {
4.     int mb_interlaced = b_mbaff && MB_INTERLACED;
5.     int height = b_chroma ? 16 >> CHROMA_V_SHIFT : 16;
6.     int i_stride = h->fdec->i_stride[i];
7.     int i_stride2 = i_stride << mb_interlaced;
8.     int i_pix_offset = mb_interlaced
9.         ? 16 * mb_x + height * (mb_y&~1) * i_stride + (mb_y&1) * i_stride
10.        : 16 * mb_x + height * mb_y * i_stride;

```



```

11. //从一整个重建帧中读取一部分像素，赋值到重建帧宏块中
12. //i_pix_offset为宏块相对于整个帧起始位置的偏移量
13. pixel *plane_fdec = &h->fdec->plane[i][i_pix_offset];
14. int fdec_idx = b_mbaff ? (mb_interlaced ? (3 + (mb_y&1)) : (mb_y&1)) : (mb_y&1);
15. //前一行宏块的底部边界像素
16. pixel *intra_fdec = &h->intra_border_backup[fdec_idx][i][mb_x*16];
17. int ref_pix_offset[2] = { i_pix_offset, i_pix_offset };
18. /* ref_pix_offset[0] references the current field and [1] the opposite field. */
19. if( mb_interlaced )
20.     ref_pix_offset[1] += (1-2*(mb_y&1)) * i_stride;
21. h->mb.pic.i_stride[i] = i_stride2;
22. h->mb.pic.p_fenc_plane[i] = &h->fenc->plane[i][i_pix_offset];
23. if( b_chroma )
24. {
25.     //色度
26.     //编码帧p_fenc
27.     h->mc.load_deinterleave_chroma_fenc( h->mb.pic.p_fenc[1], h->mb.pic.p_fenc_plane[1], i_stride2, height );
28.     //重建帧p_fdec
29.     memcpy( h->mb.pic.p_fdec[1]-FDEC_STRIDE, intra_fdec, 8*sizeof(pixel) );
30.     memcpy( h->mb.pic.p_fdec[2]-FDEC_STRIDE, intra_fdec+8, 8*sizeof(pixel) );
31.     h->mb.pic.p_fdec[1][-FDEC_STRIDE-1] = intra_fdec[-1-8];
32.     h->mb.pic.p_fdec[2][-FDEC_STRIDE-1] = intra_fdec[-1];
33. }
34. else
35. {
36.     //编码帧p_fenc
37.     h->mc.copy_PIXEL_16x16( h->mb.pic.p_fenc[i], FENC_STRIDE, h->mb.pic.p_fenc_plane[i], i_stride2, 16 );
38.     //重建帧p_fdec
39.     //上一行宏块的底部边界像素
40.     memcpy( h->mb.pic.p_fdec[i]-FDEC_STRIDE, intra_fdec, 24*sizeof(pixel) );
41.     //左上角的1个像素
42.     h->mb.pic.p_fdec[i][-FDEC_STRIDE-1] = intra_fdec[-1];
43. }
44. if( b_mbaff || h->mb.b_reencode_mb )
45. {
46.     for( int j = 0; j < height; j++ )
47.         if( b_chroma )
48.         {
49.             //把一整个重建帧中宏块对应像素的地址，赋值到宏块的重建帧指针上
50.             h->mb.pic.p_fdec[1][-1+j*FDEC_STRIDE] = plane_fdec[-2+j*i_stride2];
51.             h->mb.pic.p_fdec[2][-1+j*FDEC_STRIDE] = plane_fdec[-1+j*i_stride2];
52.         }
53.         else
54.             h->mb.pic.p_fdec[i][-1+j*FDEC_STRIDE] = plane_fdec[-1+j*i_stride2];
55. }
56. pixel *plane_src, **filtered_src;
57.
58. //P Slice 的情况
59. //fref[0]后面的[0]代表list0
60. for( int j = 0; j < h->mb.pic.i_fref[0]; j++ )
61. {
62.     // Interpolate between pixels in same field.
63.     if( mb_interlaced )
64.     {
65.         plane_src = h->fref[0][j>>1]->plane_fld[i];
66.         filtered_src = h->fref[0][j>>1]->filtered_fld[i];
67.     }
68.     else
69.     {
70.         plane_src = h->fref[0][j]->plane[i];
71.         /*
72.          * filtered_src指向半像素插值之后的得到的内插点数据
73.          * filtered_src[1]存储了H半像素内插点
74.          * filtered_src[2]存储了V半像素内插点
75.          * filtered_src[3]存储了C半像素（对角线）内插点
76.          *
77.          * 示意（“X”代表像素点）：
78.          * X   H   X
79.          *
80.          * V   C
81.          *
82.          * X       X
83.          *
84.          */
85.         filtered_src = h->fref[0][j]->filtered[i];
86.     }
87.     //注意
88.     //i为输入参数，代表处理的分量（0代表Y，1代表U，2代表V）
89.     //j为参考帧序号
90.     //处理亮度的时候相当于将plane_src赋值给了p_fref[0][j][0]
91.     h->mb.pic.p_fref[0][j][i*4] = plane_src + ref_pix_offset[j&1];
92.
93.     if( !b_chroma )
94.     {
95.         //半像素插值之后的filtered_src赋值给mb.pic.p_fref
96.         //注意
97.         //i为输入参数，代表处理的分量（0代表Y，1代表U，2代表V）
98.         //j为参考帧序号
99.         //处理亮度的时候相当于将filtered_src[1]赋值给了p_fref[0][j][1]；
100.        //filtered_src[2]赋值给了p_fref[0][j][2]；filtered_src[3]赋值给了p_fref[0][j][3]
101.        //
102.        for( int k = 1; k < 4; k++ )

```

```
102.         for( int k = 1; k < 4; k++ )
103.             h->mb.pic.p_fref[0][j][i*4+k] = filtered_src[k] + ref_pix_offset[j&1];
104.         if( !i )
105.         {
106.             if( h->sh.weight[j][0].weightfn )
107.                 h->mb.pic.p_fref_w[j] = &h->fenc->weighted[j >> mb_interlaced][ref_pix_offset[j&1]];
108.             else
109.                 h->mb.pic.p_fref_w[j] = h->mb.pic.p_fref[0][j][0];
110.         }
111.     }
112. }
113.
114. //B Slice 的情况
115. //fref[1]后面的[1]代表list1
116. //与P Slice处理方式类似
117. if( h->sh.i_type == SLICE_TYPE_B )
118.     for( int j = 0; j < h->mb.pic.i_fref[1]; j++ )
119.     {
120.         if( mb_interlaced )
121.         {
122.             plane_src = h->fref[1][j>>1]->plane_fld[i];
123.             filtered_src = h->fref[1][j>>1]->filtered_fld[i];
124.         }
125.         else
126.         {
127.             plane_src = h->fref[1][j]->plane[i];
128.             filtered_src = h->fref[1][j]->filtered[i];
129.         }
130.         h->mb.pic.p_fref[1][j][i*4] = plane_src + ref_pix_offset[j&1];
131.
132.         if( !b_chroma )
133.             for( int k = 1; k < 4; k++ )
134.                 h->mb.pic.p_fref[1][j][i*4+k] = filtered_src[k] + ref_pix_offset[j&1];
135.     }
136. }
```

从源代码可以看出，x264\_macroblock\_load\_pic\_pointers()最主要做了两个步骤：

- （1）加载编码宏块mb.pic.p\_fenc[]的像素数据，以及重建宏块mb.pic.p\_fenc[]上边的像素数据。
- （2）加载参考帧的半像素数据（除了整像素外，还包含了：H，V，C三组半像素数据点）。

## x264\_macroblock\_analyse()

x264\_macroblock\_analyse()用于分析宏块的编码模式。对于帧内宏块来说，主要分析使用Intra16x16合适还是使用Intra4x4合适；对于帧间宏块来说，主要分析它的划分模式，并且进行运动估计。下面简单整理一下相关的知识（具体的源代码在后续文章中再进行分析）。

### （1）帧内预测

#### 简述

简单记录一下帧内预测的方法。帧内预测根据宏块左边和上边的边界像素值推算宏块内部的像素值，帧内预测的效果如下图所示。其中左边的图为图像原始画面，右边的图为经过帧内预测后没有叠加残差的画面。



H.264中有两种帧内预测模式：16x16亮度帧内预测模式和4x4亮度帧内预测模式。其中16x16帧内预测模式一共有4种，如下图所示。



这4种模式列表如下。

模式	描述
Vertical	由上边像素推出相应像素值
Horizontal	由左边像素推出相应像素值
DC	由上边和左边像素平均值推出相应像素值
Plane	由上边和左边像素推出相应像素值

4x4帧内预测模式一共有9种，如下图所示。



简单记录几个编码模式判断中两个像素计算的方法：SAD和SATD。它们的定义如下：

SAD (Sum of Absolute Difference) 也可以称为SAE (Sum of Absolute Error)，即绝对误差和。它的计算方法就是求出两个像素块对应像素点的差值，将这些差值分别求绝对值之后再进行累加。

SATD (Sum of Absolute Transformed Difference) 即Hadamard变换后再绝对值求和。它和SAD的区别在于多了一个“变换”。

H.264中使用SAD和SATD进行宏块预测模式的判断。早期的编码器使用SAD进行计算，近期的编码器多使用SATD进行计算。为什么使用SATD而不使用SAD呢？关键原因在于编码之后码流的大小是和图像块DCT变换后频域信息紧密相关的，而和变换前的时域信息关联性小一些。SAD只能反应时域信息；SATD却可以反映频域信息，而且计算复杂度也低于DCT变换，因此是比较合适的模式选择的依据。

使用SAD进行模式选择的示例如下所示。下面这张图代表了一个普通的Intra16x16的宏块的像素。它的下方包含了使用Vertical，Horizontal，DC和Plane四种帧内预测模式预测的像素。通过计算可以得到这几种预测像素和原始像素之间的SAD (SAE) 分别为3985，5097，4991，2539。由于Plane模式的SAD取值最小，由此可以断定Plane模式对于这个宏块来说是最好的帧内预测模式。

□

□

## (2) 帧间预测知识简述

### 简述

运动估计的理论基础就是活动图像邻帧中的景物存在着一定的相关性。因此在压缩编码中不需要传递每一帧的所有信息，而只需要传递帧与帧之间差值就可以了（可以想象，如果画面背景是静止的，那么只需要传递很少的数据）。

在视频编码的运动估计步骤中，会查找与当前宏块或者子宏块“长得像”的宏块作为“匹配块”，然后编码传输匹配块的位置（运动矢量，参考帧）和当前宏块与匹配块之间的微小差别（残差数据）。例如下图中，当前宏块中一个“车轮”在参考帧中找到了形状同样为一个“轮子”的匹配块。

□

最早视频编码标准中都是以整像素的方式进行运动估计的。这样处理的好处是计算简单，坏处是不够精确。随着硬件技术的进步，比较新的视频编码标准（例如MP EG2）中使用1/2像素精度的方式进行运动估计。这样做计算相对复杂，但是计算也相对准确。1/2像素精度运动估计如下图所示。

□

《H.264标准》中对运动估计的精度要求又有了提升，变成了1/4像素精度。因此H.264编码器对系统性能要求又有了更高的要求。一些实验证明，1/4像素精度基本上达到了运动估计性能提升的极限。更高精度的运动估计并不能更明显的提升性能，却会导致计算复杂度的显著提升。因此现存主流的编解码标准在运动估计方面都采用了1/4精度。曾经有人压缩对比过1/2、1/4、1/8精度的运动估计下编码的视频质量，如下图所示。

□

从图中可以看出：1/4精度相比于1/2精度来说有显著的提升，但是1/8精度实际上和1/4精度是差不多的。

### 宏块划分（帧间预测）

《H.264标准》中规定，每个16x16的宏块可以划分为16x16，16x8，8x16，8x8四种类型。而如果宏块划分为8x8类型的时候，每个8x8宏块又可以划分为8x8，8x4，4x8，4x4四种小块。它们之间的关系下图所示。

□

上图中这些子宏块都包含了自己的运动矢量和参考帧序号，并且根据这两个信息获得最终的预测数据。总体说来，大的子宏块适合平坦区域，而小的子宏块适合多细节区域。例如下面这张图是一张没有进行运动补偿的残差帧的宏块分割方式图，可以看出平坦区域使用了较大的16x16分割方式，而细节区域使用了相对较小的宏块分割方式。

### 单向预测与双向预测

在运动估计的过程中，不仅仅只可以选择一个图像作为参考帧（P帧），而且还可以选择两张图片作为参考帧（B帧）。使用一张图像作为参考帧称为单向预测，而使用一张图像作为参考帧称为双向预测。使用单向预测的时候，直接将参考帧上的匹配块的数据“搬移下来”作后续的处理（“赋值”），而使用双向预测的时候，需要首先将两个参考帧上的匹配块的数据求平均值（“求平均”），然后再做后续处理。毫无疑问双向预测可以得到更好的压缩效果，但是也会使码流变得复杂一些。双向预测的示意图如下所示。

□

## x264\_macroblock\_encode()

x264\_macroblock\_encode()用于编码宏块。它主要完成了两个工作：编码（DCT变换和量化）和重建（DCT反变换和反量化）。在这里简单总结一下这方面的知识（后续文章中再对源代码进行详细分析）。

## (1) DCT相关知识简述

简单记录一下DCT相关的知识。DCT变换的核心理念就是把图像的低频信息（对应大面积平坦区域）变换到系数矩阵的左上角，而把高频信息变换到系数矩阵的右下角，这样就可以在压缩的时候（量化）去除掉人眼不敏感的高频信息（位于矩阵右下角的系数）从而达到压缩数据的目的。二维8x8DCT变换常见的示意图如下所示。

□

早期的DCT变换都使用了8x8的矩阵（变换系数为小数）。在H.264标准中新提出了一种4x4的矩阵。这种4x4 DCT变换的系数都是整数，一方面提高了运算的准确性，一方面也利于代码的优化。4x4整数DCT变换的示意图如下所示（作为对比，右侧为4x4块的Hadamard变换的示意图）。

4x4整数DCT变换的公式如下所示。

对该公式中的矩阵乘法可以转换为2次一维DCT变换：首先对4x4块中的每行像素进行一维DCT变换，然后再对4x4块中的每列像素进行一维DCT变换。而一维的DCT变换是可以改造成为蝶形快速算法的，如下所示。

同理，DCT反变换就是DCT变换的逆变换。DCT反变换的公式如下所示。

同理，DCT反变换的矩阵乘法也可以改造成为2次一维IDCT变换：首先对4x4块中的每行像素进行一维IDCT变换，然后再对4x4块中的每列像素进行一维IDCT变换。而一维的IDCT变换也可以改造成为蝶形快速算法，如下所示。

除了4x4DCT变换之外，新版本的H.264标准中还引入了一种8x8DCT。目前针对这种8x8DCT我还没有做研究，暂时不做记录。

## （2）量化相关知识简述

简单记录一下量化的概念。量化是H.264视频压缩编码中对视频质量影响最大的地方，也是会导致“信息丢失”的地方。量化的原理可以表示为下面公式：

$$FQ=\text{round}(y/Q\text{step})$$

其中，y 为输入样本点编码，Qstep为量化步长，FQ 为y 的量化值，round()为取整函数（其输出为与输入实数最近的整数）。其相反过程，即反量化为：

$$y'=FQ/Q\text{step}$$

如果Qstep较大，则量化值FQ取值较小，其相应的编码长度较小，但是但反量化时损失较多的图像细节信息。简而言之，Qstep越大，视频压缩编码后体积越小，视频质量越差。

在H.264 中，量化步长Qstep 共有52个值，如下表所示。其中QP 是量化参数，是量化步长的序号。当QP 取最小值0 时代表最精细的量化，当QP 取最大值51 时代表最粗糙的量化。QP 每增加6，Qstep 增加一倍。

《H.264标准》中规定，量化过程除了完成本职工作外，还需要完成它前一步DCT变换中“系数相乘”的工作。这一步骤的推导过程不再记录，直接给出最终的公式（这个公式完全为整数运算，同时避免了除法的使用）：

$$\begin{aligned} |Z_{ij}| &= (|W_{ij}| \cdot MF + f) \gg \text{qbits} \\ \text{sign}(Z_{ij}) &= \text{sign}(W_{ij}) \end{aligned}$$

其中：

sign()为符号函数。

Wij为DCT变换后的系数。

MF的值如下表所示。表中只列出对应QP 值为0 到5 的MF 值。QP大于6之后，将QP实行对6取余数操作，再找到MF的值。

qbits计算公式为“qbits = 15 + floor(QP/6)”。即它的值随QP 值每增加6 而增加1。

f 是偏移量（用于改善恢复图像的视觉效果）。对帧内预测图像块取2^qbits/3，对帧间预测图像块取2^qbits/6。

为了更形象的显示MF的取值，做了下面一张示意图。图中深蓝色代表MF取值较大的点，而浅蓝色代表MF取值较小的点。

## x264\_macroblock\_write\_cabac()

x264\_macroblock\_write\_cabac()属于“熵编码”模块，这一部分内容在后续文章中再进行详细分析。

## x264\_macroblock\_write\_cavlc()

x264\_macroblock\_write\_cavlc()属于“熵编码”模块，这一部分内容在后续文章中再进行详细分析。

## x264\_macroblock\_cache\_save()

x264\_macroblock\_cache\_save()用于保存当前宏块的信息以供后面宏块编码作为参考。它的作用与x264\_macroblock\_cache\_load()是相对应的。该函数定义位于comm onmacroblock.c，如下所示。

```
[cpp]
1. //保存当前宏块的的值，用于以后的宏块的编码
2. //包括Intra4x4宏块帧内预测模式，DCT非零系数，运动矢量，参考帧序号等等
3. void x264_macroblock_cache_save( x264_t *h )
4. {
5.     const int i_mb_xy = h->mb.i_mb_xy;
6.     const int i_mb_type = x264_mb_type_fix[h->mb.i_type];
7.     const int s8x8 = h->mb.i_b8_stride;
8.     const int s4x4 = h->mb.i_b4_stride;
9.     const int i_mb_4x4 = h->mb.i_b4_xy;
10.    const int i_mb_8x8 = h->mb.i_b8_xy;
11.
12.    /* GCC pessimizes direct stores to heap-allocated arrays due to aliasing. */
13.    /* By only dereferencing them once, we avoid this issue. */
14.    /* ... ... ... */
```

```

14. // Intra4x4帧内预测模式
15. int8_t *i4x4 = h->mb.intra4x4_pred_mode[i_mb_xy];
16. // DCT非零系数
17. uint8_t *nnz = h->mb.non_zero_count[i_mb_xy];
18.
19. if( SLICE_MBFAFF )
20. {
21.     x264_macroblock_backup_intra( h, h->mb.i_mb_x, h->mb.i_mb_y, 1 );
22.     x264_macroblock_store_pic( h, h->mb.i_mb_x, h->mb.i_mb_y, 0, 0, 1 );
23.     if( CHROMA444 )
24.     {
25.         x264_macroblock_store_pic( h, h->mb.i_mb_x, h->mb.i_mb_y, 1, 0, 1 );
26.         x264_macroblock_store_pic( h, h->mb.i_mb_x, h->mb.i_mb_y, 2, 0, 1 );
27.     }
28.     else
29.         x264_macroblock_store_pic( h, h->mb.i_mb_x, h->mb.i_mb_y, 1, 1, 1 );
30. }
31. else
32. {
33.     x264_macroblock_backup_intra( h, h->mb.i_mb_x, h->mb.i_mb_y, 0 );
34.     // 将当前宏块重建的像素数据保存到整张图片的像素数据中
35.     x264_macroblock_store_pic( h, h->mb.i_mb_x, h->mb.i_mb_y, 0, 0, 0 );
36.     if( CHROMA444 )
37.     {
38.         x264_macroblock_store_pic( h, h->mb.i_mb_x, h->mb.i_mb_y, 1, 0, 0 );
39.         x264_macroblock_store_pic( h, h->mb.i_mb_x, h->mb.i_mb_y, 2, 0, 0 );
40.     }
41.     else
42.         x264_macroblock_store_pic( h, h->mb.i_mb_x, h->mb.i_mb_y, 1, 1, 0 );
43. }
44.
45. x264_prefetch_fenc( h, h->fdec, h->mb.i_mb_x, h->mb.i_mb_y );
46.
47. h->mb.type[i_mb_xy] = i_mb_type;
48. h->mb.slice_table[i_mb_xy] = h->sh.i_first_mb;
49. h->mb.partition[i_mb_xy] = IS_INTRA( i_mb_type ) ? D_16x16 : h->mb.i_partition;
50. h->mb.i_mb_prev_xy = i_mb_xy;
51.
52. /* save intra4x4 */
53. // 保存Intra4x4预测模式
54. if( i_mb_type == I_4x4 )
55. {
56.     CP32( &i4x4[0], &h->mb.cache.intra4x4_pred_mode[x264_scan8[10]] );
57.     M32( &i4x4[4] ) = pack8to32( h->mb.cache.intra4x4_pred_mode[x264_scan8[5]] ],
58.                                     h->mb.cache.intra4x4_pred_mode[x264_scan8[7]] ],
59.                                     h->mb.cache.intra4x4_pred_mode[x264_scan8[13]] ], 0 );
60. }
61. else if( !h->param.b_constrained_intra || IS_INTRA(i_mb_type) )
62.     M64( i4x4 ) = I_PRED_4x4_DC * 0x0101010101010101ULL;
63. else
64.     M64( i4x4 ) = (uint8_t)(-1) * 0x0101010101010101ULL;
65.
66.
67. if( i_mb_type == I_PCM )
68. {
69.     h->mb.qp[i_mb_xy] = 0;
70.     h->mb.i_last_dqp = 0;
71.     h->mb.i_cbp_chroma = CHROMA444 ? 0 : 2;
72.     h->mb.i_cbp_luma = 0xf;
73.     h->mb.cbp[i_mb_xy] = (h->mb.i_cbp_chroma << 4) | h->mb.i_cbp_luma | 0x700;
74.     h->mb.b_transform_8x8 = 0;
75.     for( int i = 0; i < 48; i++ )
76.         h->mb.cache.non_zero_count[x264_scan8[i]] = h->param.b_cabac ? 1 : 16;
77. }
78. else
79. {
80.     if( h->mb.i_type != I_16x16 && h->mb.i_cbp_luma == 0 && h->mb.i_cbp_chroma == 0 )
81.         h->mb.i_qp = h->mb.i_last_qp;
82.     // 保存QP
83.     h->mb.qp[i_mb_xy] = h->mb.i_qp;
84.     h->mb.i_last_dqp = h->mb.i_qp - h->mb.i_last_qp;
85.     h->mb.i_last_qp = h->mb.i_qp;
86. }
87.
88. /* save non zero count */
89. // 保存DCT非零系数
90. CP32( &nnz[ 0+0*4], &h->mb.cache.non_zero_count[x264_scan8[ 0]] );
91. CP32( &nnz[ 0+1*4], &h->mb.cache.non_zero_count[x264_scan8[ 2]] );
92. CP32( &nnz[ 0+2*4], &h->mb.cache.non_zero_count[x264_scan8[ 8]] );
93. CP32( &nnz[ 0+3*4], &h->mb.cache.non_zero_count[x264_scan8[10]] );
94. CP32( &nnz[16+0*4], &h->mb.cache.non_zero_count[x264_scan8[16+0]] );
95. CP32( &nnz[16+1*4], &h->mb.cache.non_zero_count[x264_scan8[16+2]] );
96. CP32( &nnz[32+0*4], &h->mb.cache.non_zero_count[x264_scan8[32+0]] );
97. CP32( &nnz[32+1*4], &h->mb.cache.non_zero_count[x264_scan8[32+2]] );
98. if( CHROMA_FORMAT >= CHROMA_422 )
99. {
100.     CP32( &nnz[16+2*4], &h->mb.cache.non_zero_count[x264_scan8[16+ 8]] );
101.     CP32( &nnz[16+3*4], &h->mb.cache.non_zero_count[x264_scan8[16+10]] );
102.     CP32( &nnz[32+2*4], &h->mb.cache.non_zero_count[x264_scan8[32+ 8]] );
103.     CP32( &nnz[32+3*4], &h->mb.cache.non_zero_count[x264_scan8[32+10]] );
104. }
105.

```

```

106.     if( h->mb.i_cbp_luma == 0 && h->mb.i_type != I_8x8 )
107.         h->mb.b_transform_8x8 = 0;
108.     h->mb.mb_transform_size[i_mb_xy] = h->mb.b_transform_8x8;
109.     //不是Intra宏块的时候, 保存运动矢量, 参考帧序号
110.     if( h->sh.i_type != SLICE_TYPE_I )
111.     {
112.         int16_t (*mv0)[2] = &h->mb.mv[0][i_mb_4x4];
113.         int16_t (*mv1)[2] = &h->mb.mv[1][i_mb_4x4];
114.         int8_t *ref0 = &h->mb.ref[0][i_mb_8x8];
115.         int8_t *ref1 = &h->mb.ref[1][i_mb_8x8];
116.         if( !IS_INTRA( i_mb_type ) )
117.         {
118.             ref0[0+0*8x8] = h->mb.cache.ref[0][x264_scan8[0]];
119.             ref0[1+0*8x8] = h->mb.cache.ref[0][x264_scan8[4]];
120.             ref0[0+1*8x8] = h->mb.cache.ref[0][x264_scan8[8]];
121.             ref0[1+1*8x8] = h->mb.cache.ref[0][x264_scan8[12]];
122.             CP128( &mv0[0*4x4], h->mb.cache.mv[0][x264_scan8[0]+8*0] );
123.             CP128( &mv0[1*4x4], h->mb.cache.mv[0][x264_scan8[0]+8*1] );
124.             CP128( &mv0[2*4x4], h->mb.cache.mv[0][x264_scan8[0]+8*2] );
125.             CP128( &mv0[3*4x4], h->mb.cache.mv[0][x264_scan8[0]+8*3] );
126.             if( h->sh.i_type == SLICE_TYPE_B )
127.             {
128.                 ref1[0+0*8x8] = h->mb.cache.ref[1][x264_scan8[0]];
129.                 ref1[1+0*8x8] = h->mb.cache.ref[1][x264_scan8[4]];
130.                 ref1[0+1*8x8] = h->mb.cache.ref[1][x264_scan8[8]];
131.                 ref1[1+1*8x8] = h->mb.cache.ref[1][x264_scan8[12]];
132.                 CP128( &mv1[0*4x4], h->mb.cache.mv[1][x264_scan8[0]+8*0] );
133.                 CP128( &mv1[1*4x4], h->mb.cache.mv[1][x264_scan8[0]+8*1] );
134.                 CP128( &mv1[2*4x4], h->mb.cache.mv[1][x264_scan8[0]+8*2] );
135.                 CP128( &mv1[3*4x4], h->mb.cache.mv[1][x264_scan8[0]+8*3] );
136.             }
137.         }
138.         else
139.         {
140.             M16( &ref0[0*8x8] ) = (uint8_t)(-1) * 0x0101;
141.             M16( &ref0[1*8x8] ) = (uint8_t)(-1) * 0x0101;
142.             M128( &mv0[0*4x4] ) = M128_ZERO;
143.             M128( &mv0[1*4x4] ) = M128_ZERO;
144.             M128( &mv0[2*4x4] ) = M128_ZERO;
145.             M128( &mv0[3*4x4] ) = M128_ZERO;
146.             if( h->sh.i_type == SLICE_TYPE_B )
147.             {
148.                 M16( &ref1[0*8x8] ) = (uint8_t)(-1) * 0x0101;
149.                 M16( &ref1[1*8x8] ) = (uint8_t)(-1) * 0x0101;
150.                 M128( &mv1[0*4x4] ) = M128_ZERO;
151.                 M128( &mv1[1*4x4] ) = M128_ZERO;
152.                 M128( &mv1[2*4x4] ) = M128_ZERO;
153.                 M128( &mv1[3*4x4] ) = M128_ZERO;
154.             }
155.         }
156.     }
157.
158.     if( h->param.b_cabac )
159.     {
160.         uint8_t (*mvd0)[2] = h->mb.mvd[0][i_mb_xy];
161.         uint8_t (*mvd1)[2] = h->mb.mvd[1][i_mb_xy];
162.         if( IS_INTRA(i_mb_type) && i_mb_type != I_PCM )
163.             h->mb.chroma_pred_mode[i_mb_xy] = x264_mb_chroma_pred_mode_fix[h->mb.i_chroma_pred_mode];
164.         else
165.             h->mb.chroma_pred_mode[i_mb_xy] = I_PRED_CHROMA_DC;
166.
167.         if( (0x3FF30 >> i_mb_type) & 1 ) /* !INTRA && !SKIP && !DIRECT */
168.         {
169.             CP64( mvd0[0], h->mb.cache.mvd[0][x264_scan8[10]] );
170.             CP16( mvd0[4], h->mb.cache.mvd[0][x264_scan8[5]] );
171.             CP16( mvd0[5], h->mb.cache.mvd[0][x264_scan8[7]] );
172.             CP16( mvd0[6], h->mb.cache.mvd[0][x264_scan8[13]] );
173.             if( h->sh.i_type == SLICE_TYPE_B )
174.             {
175.                 CP64( mvd1[0], h->mb.cache.mvd[1][x264_scan8[10]] );
176.                 CP16( mvd1[4], h->mb.cache.mvd[1][x264_scan8[5]] );
177.                 CP16( mvd1[5], h->mb.cache.mvd[1][x264_scan8[7]] );
178.                 CP16( mvd1[6], h->mb.cache.mvd[1][x264_scan8[13]] );
179.             }
180.         }
181.         else
182.         {
183.             M128( mvd0[0] ) = M128_ZERO;
184.             if( h->sh.i_type == SLICE_TYPE_B )
185.                 M128( mvd1[0] ) = M128_ZERO;
186.         }
187.
188.         if( h->sh.i_type == SLICE_TYPE_B )
189.         {
190.             if( i_mb_type == B_SKIP || i_mb_type == B_DIRECT )
191.                 h->mb.skipbp[i_mb_xy] = 0xf;
192.             else if( i_mb_type == B_8x8 )
193.             {
194.                 int skipbp = ( h->mb.i_sub_partition[0] == D_DIRECT_8x8 ) << 0;
195.                 skipbp |= ( h->mb.i_sub_partition[1] == D_DIRECT_8x8 ) << 1;
196.                 skipbp |= ( h->mb.i_sub_partition[2] == D_DIRECT_8x8 ) << 2;

```

```
197.         skipbp |= ( h->mb.i_sub_partition[3] == D_DIRECT_8x8 ) << 3;
198.         h->mb.skipbp[i_mb_xy] = skipbp;
199.     }
200.     else
201.         h->mb.skipbp[i_mb_xy] = 0;
202. }
203. }
204. }
```

从源代码可以看出，x264\_macroblock\_cache\_save()保存了当前宏块的各种信息，例如Intra4x4宏块帧内预测模式，DCT非零系数，运动矢量，参考帧序号等等。

至此有关x264\_slice\_write()的结构分析就基本完成了。从下一篇文章开始，将会对x264\_slice\_write()中的几个关键模块进行分析：

分析（Analysis）模块

宏块编码（Encode）模块

熵编码（Entropy Encoding）模块

滤波（Filter）模块

**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/45790195>

文章标签： [x264](#) [半像素内插](#) [环路滤波](#) [DCT变换](#) [量化](#)

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