

## 原 x264源代码简单分析：编码器主干部分-2

2015年05月14日 14:14:43 阅读数：9442

=====

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](#)

=====

本文继续记录x264编码器主干部分的源代码。上一篇文章记录x264\_encoder\_open(), x264\_encoder\_headers(), 和x264\_encoder\_close()这三个函数，本文记录x264\_encoder\_encode()函数。

## 函数调用关系图

X264编码器主干部分的源代码在整个x264中的位置如下图所示。



[单击查看更清晰的图片](#)

从图中可以看出, x264主干部分最复杂的函数就是x264\_encoder\_encode(), 该函数完成了编码一帧YUV为H.264码流的工作。与之配合的还有打开编码器的函数x264\_encoder\_open(), 关闭编码器的函数x264\_encoder\_close(), 以及输出SPS/PPS/SEI这样的头信息的x264\_encoder\_he

aders()。

x264\_encoder\_open()用于打开编码器，其中初始化了libx264编码所需要的各种变量。它调用了下面的函数：

- x264\_validate\_parameters()：检查输入参数（例如输入图像的宽高是否为正数）。
- x264\_predict\_16x16\_init()：初始化Intra16x16帧内预测汇编函数。
- x264\_predict\_4x4\_init()：初始化Intra4x4帧内预测汇编函数。
- x264\_pixel\_init()：初始化像素值计算相关的汇编函数（包括SAD、SATD、SSD等）。
- x264\_dct\_init()：初始化DCT变换和DCT反变换相关的汇编函数。
- x264\_mc\_init()：初始化运动补偿相关的汇编函数。
- x264\_quant\_init()：初始化量化和反量化相关的汇编函数。
- x264\_deblock\_init()：初始化去块效应滤波器相关的汇编函数。
- x264\_lookahead\_init()：初始化Lookahead相关的变量。
- x264\_ratecontrol\_new()：初始化码率控制相关的变量。

x264\_encoder\_headers()输出SPS/PPS/SEI这些H.264码流的头信息。它调用了下面的函数：

- x264\_sps\_write()：输出SPS
- x264\_pps\_write()：输出PPS
- x264\_sei\_version\_write()：输出SEI

x264\_encoder\_encode()编码一帧YUV为H.264码流。它调用了下面的函数：

- x264\_frame\_pop\_unused()：获取1个x264\_frame\_t类型结构体fenc。如果frames.unused[]队列不为空，就调用x264\_frame\_pop()从unused[]队列取1个现成的；否则就调用x264\_frame\_new()创建一个新的。
- x264\_frame\_copy\_picture()：将输入的图像数据拷贝至fenc。
- x264\_lookahead\_put\_frame()：将fenc放入lookahead.next.list[]队列，等待确定帧类型。
- x264\_lookahead\_get\_frames()：通过lookahead分析帧类型。该函数调用了x264\_slicetype\_decide()，x264\_slicetype\_analyse()和x264\_slicetype\_frame\_cost()等函数。经过一系列分析之后，最终确定了帧类型信息，并且将帧放入frames.current[]队列。
- x264\_frame\_shift()：从frames.current[]队列取出1帧用于编码。
- x264\_reference\_update()：更新参考帧队列。
- x264\_reference\_reset()：如果为IDR帧，调用该函数清空参考帧列表。
- x264\_reference\_hierarchy\_reset()：如果是非IDR的I帧、P帧、B帧（可做为参考帧），调用该函数。
- x264\_reference\_build\_list()：创建参考帧列表list0和list1。
- x264\_ratecontrol\_start()：开启码率控制。
- x264\_slice\_init()：创建 Slice Header。
- x264\_slices\_write()：编码数据（最关键的步骤）。其中调用了x264\_slice\_write()完成了编码的工作（注意"x264\_slices\_write()"和"x264\_slice\_write()"名字差了一个"s"）。
- x264\_encoder\_frame\_end()：编码结束后做一些后续处理，例如记录一些统计信息。其中调用了x264\_encoder\_encapsulate\_nals()封装NALU（添加起始码），调用x264\_frame\_push\_unused()将fenc重新放回frames.unused[]队列，并且调用x264\_ratecontrol\_end()结束码率控制。

x264\_encoder\_close()用于关闭解码器，其中释放了libx264初始化的时候使用的各种变量。它调用了下面的函数：

- x264\_lookahead\_delete()：释放Lookahead相关的变量。
- x264\_ratecontrol\_summary()：汇总码率控制信息。
- x264\_ratecontrol\_delete()：关闭码率控制。

上一篇文章已经记录了x264\_encoder\_open()，x264\_encoder\_headers()，和x264\_encoder\_close()这三个函数的源代码。本文继续上一篇文章的内容，记录x264\_encoder\_encode()函数的源代码。

## x264\_encoder\_encode()

x264\_encoder\_encode()是libx264的API函数，用于编码一帧YUV为H.264码流。该函数的声明如下所示。

[cpp]  

```
1. /* x264_encoder_encode:
2.  *   encode one picture.
3.  *   *pi_nal is the number of NAL units outputted in pp_nal.
4.  *   returns the number of bytes in the returned NALs.
5.  *   returns negative on error and zero if no NAL units returned.
6.  *   the payloads of all output NALs are guaranteed to be sequential in memory. */
7. int x264_encoder_encode( x264_t *, x264_nal_t **pp_nal, int *pi_nal, x264_picture_t *pic_in, x264_picture_t *pic_out );
```

x264\_encoder\_encode()的定义如下所示。

[cpp]  

```
1. /*****
2.  * x264_encoder_encode:
3.  *   XXX: i_poc : is the poc of the current given picture
4.  *   i_frame : is the number of the frame being coded
5.  *   ex:  type frame poc
6.  *       I      0  2*0
7.  *       P      1  2*3
8.  *       B      2  2*1
9.  *       B      3  2*2
10.  *       P      4  2*6
11.  *       B      5  2*4
12.  *       B      6  2*5
13.  *
14.  * 注释和处理: 雷霄骅
15.  *  http://blog.csdn.net/leixiaohua1020
16.  *  leixiaohua1020@126.com
17.  *****/
18. //编码一帧数据
19. int x264_encoder_encode( x264_t *h,
20.                          x264_nal_t **pp_nal, int *pi_nal,
21.                          x264_picture_t *pic_in,
22.                          x264_picture_t *pic_out )
23. {
24.     x264_t *thread_current, *thread_prev, *thread_oldest;
25.     int i_nal_type, i_nal_ref_idc, i_global_qp;
26.     int overhead = NALU_OVERHEAD;
27.
28. #if HAVE_OPENCL
29.     if( h->opencl.b_fatal_error )
30.         return -1;
31. #endif
32.
33.     if( h->i_thread_frames > 1 )
34.     {
35.         thread_prev = h->thread[ h->i_thread_phase ];
36.         h->i_thread_phase = (h->i_thread_phase + 1) % h->i_thread_frames;
37.         thread_current = h->thread[ h->i_thread_phase ];
38.         thread_oldest = h->thread[ (h->i_thread_phase + 1) % h->i_thread_frames ];
39.         x264_thread_sync_context( thread_current, thread_prev );
40.         x264_thread_sync_ratecontrol( thread_current, thread_prev, thread_oldest );
41.         h = thread_current;
42.     }
43.     else
44.     {
45.         thread_current =
46.         thread_oldest = h;
47.     }
48.     h->i_cpb_delay_pir_offset = h->i_cpb_delay_pir_offset_next;
49.
50.     /* no data out */
51.     *pi_nal = 0;
52.     *pp_nal = NULL;
53.
54.     /* ----- Setup new frame from picture ----- */
55.     if( pic_in != NULL )
56.     {
57.         /* 1: Copy the picture to a frame and move it to a buffer */
58.         //步骤1
59.         //fenc存储了编码帧
60.         //获取一帧的空间fenc, 用来存放待编码的帧
61.         x264_frame_t *fenc = x264_frame_pop_unused( h, 0 );
62.         if( !fenc )
63.             return -1;
64.
65.         //外部像素数据传递到内部系统
66.         //pic_in (外部结构体x264_picture_t) 到fenc (内部结构体x264_frame_t)
67.         if( x264_frame_copy_picture( h, fenc, pic_in ) < 0 )
68.             return -1;
69.
70.         //宽和高都确保是16的整数倍 (宏块宽度的整数倍)
71.         if( h->param.i_width != 16 * h->mb.i_mb_width ||
72.             h->param.i_height != 16 * h->mb.i_mb_height )
73.             x264_frame_expand_border_mod16( h, fenc );//扩展至16整数倍
74.
75.         fenc->i_frame = h->frames.i_input++;
76.
77.         if( fenc->i_frame == 0 )
78.             h->frames.i_first_pts = fenc->i_pts;
```

```

70.         h->frames.i_bframe_delay && fenc->i_pts == h->frames.i_bframe_delay ;
71.         h->frames.i_bframe_delay_time = fenc->i_pts - h->frames.i_first_pts;
72.
73.
74.         if( h->param.b_vfr_input && fenc->i_pts <= h->frames.i_largest_pts )
75.             x264_log( h, X264_LOG_WARNING, "non-strictly-monotonic PTS\n" );
76.
77.         h->frames.i_second_largest_pts = h->frames.i_largest_pts;
78.         h->frames.i_largest_pts = fenc->i_pts;
79.
80.         if( (fenc->i_pic_struct < PIC_STRUCT_AUTO) || (fenc->i_pic_struct > PIC_STRUCT_TRIPLE) )
81.             fenc->i_pic_struct = PIC_STRUCT_AUTO;
82.
83.         if( fenc->i_pic_struct == PIC_STRUCT_AUTO )
84.         {
85. #if HAVE_INTERLACED
86.             int b_interlaced = fenc->param ? fenc->param->b_interlaced : h->param.b_interlaced;
87. #else
88.             int b_interlaced = 0;
89. #endif
90.             if( b_interlaced )
91.             {
92.                 int b_tff = fenc->param ? fenc->param->b_tff : h->param.b_tff;
93.                 fenc->i_pic_struct = b_tff ? PIC_STRUCT_TOP_BOTTOM : PIC_STRUCT_BOTTOM_TOP;
94.             }
95.             else
96.                 fenc->i_pic_struct = PIC_STRUCT_PROGRESSIVE;
97.         }
98.
99.         if( h->param.rc.b_mb_tree && h->param.rc.b_stat_read )
100.         {
101.             if( x264_macroblock_tree_read( h, fenc, pic_in->prop.quant_offsets ) )
102.                 return -1;
103.         }
104.         else
105.             x264_stack_align( x264_adaptive_quant_frame, h, fenc, pic_in->prop.quant_offsets );
106.
107.         if( pic_in->prop.quant_offsets_free )
108.             pic_in->prop.quant_offsets_free( pic_in->prop.quant_offsets );
109.         //降低分辨率处理（原来的一半），线性内插
110.         //注意这里并不是6抽头滤波器的半像素内插
111.         if( h->frames.b_have_lowres )
112.             x264_frame_init_lowres( h, fenc );
113.
114.         /* 2: Place the frame into the queue for its slice type decision */
115.         //步骤2
116.         //fenc放入lookahead.next.list[]队列，等待确定帧类型
117.         x264_lookahead_put_frame( h, fenc );
118.
119.         if( h->frames.i_input <= h->frames.i_delay + 1 - h->i_thread_frames )
120.         {
121.             /* Nothing yet to encode, waiting for filling of buffers */
122.             pic_out->i_type = X264_TYPE_AUTO;
123.             return 0;
124.         }
125.         else
126.         {
127.             //输入数据为空的时候（Flush Encoder?），不需要lookahead
128.
129.             /* signal kills for lookahead thread */
130.             x264_pthread_mutex_lock( &h->lookahead->ifbuf.mutex );
131.             h->lookahead->b_exit_thread = 1;
132.             x264_pthread_cond_broadcast( &h->lookahead->ifbuf.cv_fill );
133.             x264_pthread_mutex_unlock( &h->lookahead->ifbuf.mutex );
134.         }
135.
136.         h->i_frame++;
137.         /* 3: The picture is analyzed in the lookahead */
138.         //步骤3
139.         //通过lookahead分析帧类型
140.         if( !h->frames.current[0] )
141.             x264_lookahead_get_frames( h );
142.
143.         if( !h->frames.current[0] && x264_lookahead_is_empty( h ) )
144.             return x264_encoder_frame_end( thread_oldest, thread_current, pp_nal, pi_nal, pic_out );
145.
146.         /* ----- Get frame to be encoded ----- */
147.         /* 4: get picture to encode */
148.         //从frames.current[]队列取出1帧[0]用于编码
149.         h->fenc = x264_frame_shift( h->frames.current );
150.
151.         /* If applicable, wait for previous frame reconstruction to finish */
152.         if( h->param.b_sliced_threads )
153.             if( x264_threadpool_wait_all( h ) < 0 )
154.                 return -1;
155.
156.         if( h->i_frame == h->i_thread_frames - 1 )
157.             h->i_reordered_pts_delay = h->fenc->i_reordered_pts;
158.         if( h->reconfig )
159.         {
160.             x264_encoder_reconfig_apply( h, &h->reconfig_h->param );
161.             h->reconfig = 0;
162.

```

```

170.     }
171.     if( h->fenc->param )
172.     {
173.         x264_encoder_reconfig_apply( h, h->fenc->param );
174.         if( h->fenc->param->param_free )
175.         {
176.             h->fenc->param->param_free( h->fenc->param );
177.             h->fenc->param = NULL;
178.         }
179.     }
180.
181.     // ok to call this before encoding any frames, since the initial values of fdec have b_kept_as_ref=0
182.     //更新参考帧队列frames.reference[] .若为B帧则不更新
183.     //重建帧fdec移植参考帧列表, 新建一个fdec
184.     if( x264_reference_update( h ) )
185.         return -1;
186.     h->fdec->i_lines_completed = -1;
187.
188.     if( !IS_X264_TYPE_I( h->fenc->i_type ) )
189.     {
190.         int valid_refs_left = 0;
191.         for( int i = 0; h->frames.reference[i]; i++ )
192.             if( !h->frames.reference[i]->b_corrupt )
193.                 valid_refs_left++;
194.         /* No valid reference frames left: force an IDR. */
195.         if( !valid_refs_left )
196.         {
197.             h->fenc->b_keyframe = 1;
198.             h->fenc->i_type = X264_TYPE_IDR;
199.         }
200.     }
201.
202.     if( h->fenc->b_keyframe )
203.     {
204.         h->frames.i_last_keyframe = h->fenc->i_frame;
205.         if( h->fenc->i_type == X264_TYPE_IDR )
206.         {
207.             h->i_frame_num = 0;
208.             h->frames.i_last_idr = h->fenc->i_frame;
209.         }
210.     }
211.     h->sh.i_mmco_command_count =
212.     h->sh.i_mmco_remove_from_end = 0;
213.     h->b_ref_reorder[0] =
214.     h->b_ref_reorder[1] = 0;
215.     h->fdec->i_poc =
216.     h->fenc->i_poc = 2 * ( h->fenc->i_frame - X264_MAX( h->frames.i_last_idr, 0 ) );
217.
218.     /* ----- Setup frame context ----- */
219.     /* 5: Init data dependent of frame type */
220.     if( h->fenc->i_type == X264_TYPE_IDR )
221.     {
222.         //I与IDR区别
223.         //注意IDR会导致参考帧列表清空, 而I不会
224.         //I图像之后的图像可以引用I图像之间的图像做运动参考
225.         /* reset ref pictures */
226.         i_nal_type = NAL_SLICE_IDR;
227.         i_nal_ref_idc = NAL_PRIORITY_HIGHEST;
228.         h->sh.i_type = SLICE_TYPE_I;
229.         //若是IDR帧, 则清空所有参考帧
230.         x264_reference_reset( h );
231.         h->frames.i_poc_last_open_gop = -1;
232.     }
233.     else if( h->fenc->i_type == X264_TYPE_I )
234.     {
235.         //I与IDR区别
236.         //注意IDR会导致参考帧列表清空, 而I不会
237.         //I图像之后的图像可以引用I图像之间的图像做运动参考
238.         i_nal_type = NAL_SLICE;
239.         i_nal_ref_idc = NAL_PRIORITY_HIGH; /* Not completely true but for now it is (as all I/P are kept as ref)*/
240.         h->sh.i_type = SLICE_TYPE_I;
241.         x264_reference_hierarchy_reset( h );
242.         if( h->param.b_open_gop )
243.             h->frames.i_poc_last_open_gop = h->fenc->b_keyframe ? h->fenc->i_poc : -1;
244.     }
245.     else if( h->fenc->i_type == X264_TYPE_P )
246.     {
247.         i_nal_type = NAL_SLICE;
248.         i_nal_ref_idc = NAL_PRIORITY_HIGH; /* Not completely true but for now it is (as all I/P are kept as ref)*/
249.         h->sh.i_type = SLICE_TYPE_P;
250.         x264_reference_hierarchy_reset( h );
251.         h->frames.i_poc_last_open_gop = -1;
252.     }
253.     else if( h->fenc->i_type == X264_TYPE_BREF )
254.     {
255.         //可以作为参考帧的B帧, 这是个特色
256.
257.         i_nal_type = NAL_SLICE;
258.         i_nal_ref_idc = h->param.i_bframe_pyramid == X264_B_PYRAMID_STRICT ? NAL_PRIORITY_LOW : NAL_PRIORITY_HIGH;
259.         h->sh.i_type = SLICE_TYPE_B;
260.         x264_reference_hierarchy_reset( h );

```

```

261.     }
262.     else /* B frame */
263.     {
264.         //最普通
265.
266.         i_nal_type = NAL_SLICE;
267.         i_nal_ref_idc = NAL_PRIORITY_DISPOSABLE;
268.         h->sh.i_type = SLICE_TYPE_B;
269.     }
270.     //重建帧与编码帧的赋值...
271.     h->fdec->i_type = h->fenc->i_type;
272.     h->fdec->i_frame = h->fenc->i_frame;
273.     h->fenc->b_kept_as_ref =
274.     h->fdec->b_kept_as_ref = i_nal_ref_idc != NAL_PRIORITY_DISPOSABLE && h->param.i_keyint_max > 1;
275.
276.     h->fdec->mb_info = h->fenc->mb_info;
277.     h->fdec->mb_info_free = h->fenc->mb_info_free;
278.     h->fenc->mb_info = NULL;
279.     h->fenc->mb_info_free = NULL;
280.
281.     h->fdec->i_pts = h->fenc->i_pts;
282.     if( h->frames.i_bframe_delay )
283.     {
284.         int64_t *prev_reordered_pts = thread_current->frames.i_prev_reordered_pts;
285.         h->fdec->i_dts = h->i_frame > h->frames.i_bframe_delay
286.             ? prev_reordered_pts[ (h->i_frame - h->frames.i_bframe_delay) % h->frames.i_bframe_delay ]
287.             : h->fenc->i_reordered_pts - h->frames.i_bframe_delay_time;
288.         prev_reordered_pts[ h->i_frame % h->frames.i_bframe_delay ] = h->fenc->i_reordered_pts;
289.     }
290.     else
291.         h->fdec->i_dts = h->fenc->i_reordered_pts;
292.     if( h->fenc->i_type == X264_TYPE_IDR )
293.         h->i_last_idr_pts = h->fdec->i_pts;
294.
295.     /* ----- Init ----- */
296.     /* build ref list 0/1 */
297.     //创建参考帧列表list0和list1
298.     x264_reference_build_list( h, h->fdec->i_poc );
299.
300.     /* ----- Write the bitstream ----- */
301.     /* Init bitstream context */
302.     //用于输出
303.     if( h->param.b_sliced_threads )
304.     {
305.         for( int i = 0; i < h->param.i_threads; i++ )
306.         {
307.             bs_init( &h->thread[i]->out.bs, h->thread[i]->out.p_bitstream, h->thread[i]->out.i_bitstream );
308.             h->thread[i]->out.i_nal = 0;
309.         }
310.     }
311.     else
312.     {
313.         bs_init( &h->out.bs, h->out.p_bitstream, h->out.i_bitstream );
314.         h->out.i_nal = 0;
315.     }
316.
317.     if( h->param.b_aud )
318.     {
319.         int pic_type;
320.
321.         if( h->sh.i_type == SLICE_TYPE_I )
322.             pic_type = 0;
323.         else if( h->sh.i_type == SLICE_TYPE_P )
324.             pic_type = 1;
325.         else if( h->sh.i_type == SLICE_TYPE_B )
326.             pic_type = 2;
327.         else
328.             pic_type = 7;
329.
330.         x264_nal_start( h, NAL_AUD, NAL_PRIORITY_DISPOSABLE );
331.         bs_write( &h->out.bs, 3, pic_type );
332.         bs_rbsp_trailing( &h->out.bs );
333.         if( x264_nal_end( h ) )
334.             return -1;
335.         overhead += h->out.nal[h->out.i_nal-1].i_payload + NALU_OVERHEAD;
336.     }
337.
338.     h->i_nal_type = i_nal_type;
339.     h->i_nal_ref_idc = i_nal_ref_idc;
340.
341.     if( h->param.b_intra_refresh )
342.     {
343.         if( IS_X264_TYPE_I( h->fenc->i_type ) )
344.         {
345.             h->fdec->i_frames_since_pir = 0;
346.             h->b_queued_intra_refresh = 0;
347.             /* PIR is currently only supported with ref == 1, so any intra frame effectively refreshes
348.              * the whole frame and counts as an intra refresh. */
349.             h->fdec->f_pir_position = h->mb.i_mb_width;
350.         }
351.         else if( h->fenc->i_type == X264_TYPE_P )

```

```

352.     {
353.         int pocdiff = (h->fdec->i_poc - h->fref[0][0]->i_poc)/2;
354.         float increment = X264_MAX( ((float)h->mb.i_mb_width-1) / h->param.i_keyint_max, 1 );
355.         h->fdec->f_pir_position = h->fref[0][0]->f_pir_position;
356.         h->fdec->i_frames_since_pir = h->fref[0][0]->i_frames_since_pir + pocdiff;
357.         if( h->fdec->i_frames_since_pir >= h->param.i_keyint_max ||
358.             (h->b_queued_intra_refresh && h->fdec->f_pir_position + 0.5 >= h->mb.i_mb_width) )
359.         {
360.             h->fdec->f_pir_position = 0;
361.             h->fdec->i_frames_since_pir = 0;
362.             h->b_queued_intra_refresh = 0;
363.             h->fenc->b_keyframe = 1;
364.         }
365.         h->fdec->i_pir_start_col = h->fdec->f_pir_position+0.5;
366.         h->fdec->f_pir_position += increment * pocdiff;
367.         h->fdec->i_pir_end_col = h->fdec->f_pir_position+0.5;
368.         /* If our intra refresh has reached the right side of the frame, we're done. */
369.         if( h->fdec->i_pir_end_col >= h->mb.i_mb_width - 1 )
370.         {
371.             h->fdec->f_pir_position = h->mb.i_mb_width;
372.             h->fdec->i_pir_end_col = h->mb.i_mb_width - 1;
373.         }
374.     }
375. }
376.
377. if( h->fenc->b_keyframe )
378. {
379.     //每个关键帧前面重复加上SPS和PPS
380.     /* Write SPS and PPS */
381.     if( h->param.b_repeat_headers )
382.     {
383.         /* generate sequence parameters */
384.         x264_nal_start( h, NAL_SPS, NAL_PRIORITY_HIGHEST );
385.         x264_sps_write( &h->out.bs, h->sps );
386.         if( x264_nal_end( h ) )
387.             return -1;
388.         /* Pad AUD/SPS to 256 bytes like Panasonic */
389.         if( h->param.i_avcintra_class )
390.             h->out.nal[h->out.i_nal-1].i_padding = 256 - bs_pos( &h->out.bs ) / 8 - 2*NALU_OVERHEAD;
391.         overhead += h->out.nal[h->out.i_nal-1].i_payload + h->out.nal[h->out.i_nal-1].i_padding + NALU_OVERHEAD;
392.
393.         /* generate picture parameters */
394.         x264_nal_start( h, NAL_PPS, NAL_PRIORITY_HIGHEST );
395.         x264_pps_write( &h->out.bs, h->sps, h->pps );
396.         if( x264_nal_end( h ) )
397.             return -1;
398.         if( h->param.i_avcintra_class )
399.             h->out.nal[h->out.i_nal-1].i_padding = 256 - h->out.nal[h->out.i_nal-1].i_payload - NALU_OVERHEAD;
400.         overhead += h->out.nal[h->out.i_nal-1].i_payload + h->out.nal[h->out.i_nal-1].i_padding + NALU_OVERHEAD;
401.     }
402.
403.     /* when frame threading is used, buffering period sei is written in x264_encoder_frame_end */
404.     if( h->i_thread_frames == 1 && h->sps->vui.b_nal_hrd_parameters_present )
405.     {
406.         x264_hrd_fullness( h );
407.         x264_nal_start( h, NAL_SEI, NAL_PRIORITY_DISPOSABLE );
408.         x264_sei_buffering_period_write( h, &h->out.bs );
409.         if( x264_nal_end( h ) )
410.             return -1;
411.         overhead += h->out.nal[h->out.i_nal-1].i_payload + SEI_OVERHEAD;
412.     }
413. }
414.
415. /* write extra sei */
416. //下面很大一段代码用于写入SEI（一部分是为了适配其他的解码器）=====
417. for( int i = 0; i < h->fenc->extra_sei.num_payloads; i++ )
418. {
419.     x264_nal_start( h, NAL_SEI, NAL_PRIORITY_DISPOSABLE );
420.     x264_sei_write( &h->out.bs, h->fenc->extra_sei.payloads[i].payload, h->fenc->extra_sei.payloads[i].payload_size,
421.                    h->fenc->extra_sei.payloads[i].payload_type );
422.     if( x264_nal_end( h ) )
423.         return -1;
424.     overhead += h->out.nal[h->out.i_nal-1].i_payload + SEI_OVERHEAD;
425.     if( h->fenc->extra_sei.sei_free )
426.     {
427.         h->fenc->extra_sei.sei_free( h->fenc->extra_sei.payloads[i].payload );
428.         h->fenc->extra_sei.payloads[i].payload = NULL;
429.     }
430. }
431.
432. if( h->fenc->extra_sei.sei_free )
433. {
434.     h->fenc->extra_sei.sei_free( h->fenc->extra_sei.payloads );
435.     h->fenc->extra_sei.payloads = NULL;
436.     h->fenc->extra_sei.sei_free = NULL;
437. }
438. /*特殊的SEI信息（Avid等解码器需要）
439. if( h->fenc->b_keyframe )
440. {
441.     /* Avid's decoder strictly wants two SEIs for AVC-Intra so we can't insert the x264 SEI */
442.     if( h->param.b_repeat_headers && h->fenc->i_frame == 0 && !h->param.i_avcintra_class )

```



```

443.     {
444.         /* identify ourself */
445.         x264_nal_start( h, NAL_SEI, NAL_PRIORITY_DISPOSABLE );
446.         if( x264_sei_version_write( h, &h->out.bs ) )
447.             return -1;
448.         if( x264_nal_end( h ) )
449.             return -1;
450.         overhead += h->out.nal[h->out.i_nal-1].i_payload + SEI_OVERHEAD;
451.     }
452.
453.     if( h->fenc->i_type != X264_TYPE_IDR )
454.     {
455.         int time_to_recovery = h->param.b_open_gop ? 0 : X264_MIN( h->mb.i_mb_width - 1, h->param.i_keyint_max ) + h->param.i_bframe - 1;
456.         x264_nal_start( h, NAL_SEI, NAL_PRIORITY_DISPOSABLE );
457.         x264_sei_recovery_point_write( h, &h->out.bs, time_to_recovery );
458.         if( x264_nal_end( h ) )
459.             return -1;
460.         overhead += h->out.nal[h->out.i_nal-1].i_payload + SEI_OVERHEAD;
461.     }
462. }
463.
464. if( h->param.i_frame_packing >= 0 && (h->fenc->b_keyframe || h->param.i_frame_packing == 5) )
465. {
466.     x264_nal_start( h, NAL_SEI, NAL_PRIORITY_DISPOSABLE );
467.     x264_sei_frame_packing_write( h, &h->out.bs );
468.     if( x264_nal_end( h ) )
469.         return -1;
470.     overhead += h->out.nal[h->out.i_nal-1].i_payload + SEI_OVERHEAD;
471. }
472.
473. /* generate sei pic timing */
474. if( h->sps->vui.b_pic_struct_present || h->sps->vui.b_nal_hrd_parameters_present )
475. {
476.     x264_nal_start( h, NAL_SEI, NAL_PRIORITY_DISPOSABLE );
477.     x264_sei_pic_timing_write( h, &h->out.bs );
478.     if( x264_nal_end( h ) )
479.         return -1;
480.     overhead += h->out.nal[h->out.i_nal-1].i_payload + SEI_OVERHEAD;
481. }
482.
483. /* As required by Blu-ray. */
484. if( !IS_X264_TYPE_B( h->fenc->i_type ) && h->b_sh_backup )
485. {
486.     h->b_sh_backup = 0;
487.     x264_nal_start( h, NAL_SEI, NAL_PRIORITY_DISPOSABLE );
488.     x264_sei_dec_ref_pic_marking_write( h, &h->out.bs );
489.     if( x264_nal_end( h ) )
490.         return -1;
491.     overhead += h->out.nal[h->out.i_nal-1].i_payload + SEI_OVERHEAD;
492. }
493.
494. if( h->fenc->b_keyframe && h->param.b_intra_refresh )
495.     h->i_cpb_delay_pir_offset_next = h->fenc->i_cpb_delay;
496.
497. /* Filler space: 10 or 18 SEIs' worth of space, depending on resolution */
498. if( h->param.i_avcintra_class )
499. {
500.     /* Write an empty filler NAL to mimic the AUD in the P2 format*/
501.     x264_nal_start( h, NAL_FILLER, NAL_PRIORITY_DISPOSABLE );
502.     x264_filler_write( h, &h->out.bs, 0 );
503.     if( x264_nal_end( h ) )
504.         return -1;
505.     overhead += h->out.nal[h->out.i_nal-1].i_payload + NALU_OVERHEAD;
506.
507.     /* All lengths are magic lengths that decoders expect to see */
508.     /* "UMID" SEI */
509.     x264_nal_start( h, NAL_SEI, NAL_PRIORITY_DISPOSABLE );
510.     if( x264_sei_avcintra_umid_write( h, &h->out.bs ) < 0 )
511.         return -1;
512.     if( x264_nal_end( h ) )
513.         return -1;
514.     overhead += h->out.nal[h->out.i_nal-1].i_payload + SEI_OVERHEAD;
515.
516.     int unpadded_len;
517.     int total_len;
518.     if( h->param.i_height == 1080 )
519.     {
520.         unpadded_len = 5780;
521.         total_len = 17*512;
522.     }
523.     else
524.     {
525.         unpadded_len = 2900;
526.         total_len = 9*512;
527.     }
528.     /* "VANC" SEI */
529.     x264_nal_start( h, NAL_SEI, NAL_PRIORITY_DISPOSABLE );
530.     if( x264_sei_avcintra_vanc_write( h, &h->out.bs, unpadded_len ) < 0 )
531.         return -1;
532.     if( x264_nal_end( h ) )
533.         return -1;

```

```

535.         return -1;
536.
537.         h->out.nal[h->out.i_nal-1].i_padding = total_len - h->out.nal[h->out.i_nal-1].i_payload - SEI_OVERHEAD;
538.         overhead += h->out.nal[h->out.i_nal-1].i_payload + h->out.nal[h->out.i_nal-1].i_padding + SEI_OVERHEAD;
539.     }
540.     //写入SEI代码结束=====
541.
542.     /* Init the rate control */
543.     /* FIXME: Include slice header bit cost. */
544.     //码率控制单元初始化
545.     x264_ratecontrol_start( h, h->fenc->i_qpplus1, overhead*8 );
546.     i_global_qp = x264_ratecontrol_qp( h );
547.
548.     pic_out->i_qpplus1 =
549.     h->fdec->i_qpplus1 = i_global_qp + 1;
550.
551.     if( h->param.rc.b_stat_read && h->sh.i_type != SLICE_TYPE_I )
552.     {
553.         x264_reference_build_list_optimal( h );
554.         x264_reference_check_reorder( h );
555.     }
556.
557.     if( h->i_ref[0] )
558.         h->fdec->i_poc_l0ref0 = h->fref[0][0]->i_poc;
559.
560.     /* ----- Create slice header ----- */
561.     //创建Slice Header
562.     x264_slice_init( h, i_nal_type, i_global_qp );
563.
564.     /*----- Weights ----- */
565.     //加权预测
566.
567.     if( h->sh.i_type == SLICE_TYPE_B )
568.         x264_macroblock_bipred_init( h );
569.
570.     x264_weighted_pred_init( h );
571.
572.     if( i_nal_ref_idc != NAL_PRIORITY_DISPOSABLE )
573.         h->i_frame_num++;
574.
575.     /* Write frame */
576.     h->i_threadslice_start = 0;
577.     h->i_threadslice_end = h->mb.i_mb_height;
578.
579.     if( h->i_thread_frames > 1 )
580.     {
581.         x264_threadpool_run( h->threadpool, (void*)x264_slices_write, h );
582.         h->b_thread_active = 1;
583.     }
584.     else if( h->param.b_sliced_threads )
585.     {
586.         if( x264_threaded_slices_write( h ) )
587.             return -1;
588.     }
589.     else{
590.         //真正的编码——编码1个图像帧（注意这里“slices”后面有“s”）
591.         if( (intptr_t)x264_slices_write( h ) )
592.             return -1;
593.     }
594.     //结束的时候做一些处理，记录一些统计信息
595.     //输出NALU
596.     //输出重建帧
597.     return x264_encoder_frame_end( thread_oldest, thread_current, pp_nal, pi_nal, pic_out );
598. }

```

从源代码可以看出，x264\_encoder\_encode()的流程大致如下：

- (1) 调用x264\_frame\_pop\_unused获取一个空的fenc（x264\_frame\_t类型）用于存储一帧编码像素数据。
- (2) 调用x264\_frame\_copy\_picture()将外部结构体的pic\_in（x264\_picture\_t类型）的数据拷贝给内部结构体的fenc（x264\_frame\_t类型）。
- (3) 调用x264\_lookahead\_put\_frame()将fenc放入Lookahead模块的队列中，等待确定帧类型。
- (4) 调用x264\_lookahead\_get\_frames()分析Lookahead模块中一个帧的帧类型。分析后的帧保存在frames.current[]中。
- (5) 调用x264\_frame\_shift()从frames.current[]中取出分析帧类型之后的fenc。
- (6) 调用x264\_reference\_update()更新参考帧队列frames.reference[]。
- (7) 如果编码帧fenc是IDR帧，调用x264\_reference\_reset()清空参考帧队列frames.reference[]。
- (8) 调用x264\_reference\_build\_list()创建参考帧列表List0和List1。
- (9) 根据选项做一些配置：
  - a)如果b\_aud不为0，输出AUD类型NALU
  - b)在当前帧是关键帧的情况下，如果b\_repeat\_headers不为0，调用x264\_sps\_write()和x264\_pps\_write()输出SPS和PPS。
  - c)输出一些特殊的SEI信息，用于适配各种解码器。
- (10) 调用x264\_slice\_init()初始化Slice Header信息。
- (11) 调用x264\_slices\_write()进行编码。该部分是libx264的核心，在后续文章中会详细分析。
- (12) 调用x264\_encoder\_frame\_end()做一些编码后的后续处理。

下文将会按照步骤对上述函数进行简单的分析。

## x264\_frame\_pop\_unused()

x264\_frame\_pop\_unused()用于获取1个x264\_frame\_t类型结构体fenc。该函数的定义位于common\frame.c，如下所示。

```
1. //获取一帧的编码帧fenc或者重建帧fdec
2. x264_frame_t *x264_frame_pop_unused( x264_t *h, int b_fdec )
3. {
4.     x264_frame_t *frame;
5.     if ( h->frames.unused[b_fdec][0] )//unused队列不为空
6.         frame = x264_frame_pop( h->frames.unused[b_fdec] );//从unused队列取
7.     else
8.         frame = x264_frame_new( h, b_fdec );//分配一帧空间
9.     if ( !frame )
10.        return NULL;
11.    frame->b_last_minigop_bframe = 0;
12.    frame->i_reference_count = 1;
13.    frame->b_intra_calculated = 0;
14.    frame->b_scenecut = 1;
15.    frame->b_keyframe = 0;
16.    frame->b_corrupt = 0;
17.    frame->i_slice_count = h->param.b_sliced_threads ? h->param.i_threads : 1;
18.
19.    memset( frame->weight, 0, sizeof(frame->weight) );
20.    memset( frame->f_weighted_cost_delta, 0, sizeof(frame->f_weighted_cost_delta) );
21.
22.    return frame;
23. }
```

从源代码可以看出，如果frames.unused[]队列不为空，x264\_frame\_pop\_unused()就调用x264\_frame\_pop()从unused[]队列取1个现成的；否则就调用x264\_frame\_new()创建一个新的。下面看一下这两个函数。

## x264\_frame\_pop()

x264\_frame\_pop()用于从一个队列的尾部取出一个帧。该函数的定义位于common\frame.c, 如下所示。

```
1. //从队列的尾部取出一帧
2. x264_frame_t *x264_frame_pop( x264_frame_t **list )
3. {
4.     x264_frame_t *frame;
5.     int i = 0;
6.     assert( list[0] );
7.     while( list[i+1] ) i++;
8.     frame = list[i];
9.     list[i] = NULL;
10.    return frame;
11. }
```

从源代码中可以看出，x264 frame pop()首先通过一个while()循环找到队列尾部的元素，然后将该元素作为返回值返回。

## x264\_frame\_new()

x264\_frame\_new()用于新建一个x264\_frame\_t。该函数的定义位于common/frame.c，如下所示。

```

1. //新建一个帧
2. //b_fdec : 取1的时候为重建帧fdec, 取0的时候为编码帧fenc
3. static x264_frame_t *x264_frame_new( x264_t *h, int b_fdec )
4. {
5.     x264_frame_t *frame;
6.     //注意转换后只有3种colospace : X264_CSP_NV12(对应YUV420), X264_CSP_NV16(对应YUV422), X264_CSP_I444(对应YUV444)
7.     int i_csp = x264_frame_internal_csp( h->param.i_csp );
8.     int i_mb_count = h->mb.i_mb_count;
9.     int i_stride, i_width, i_lines, luma_plane_count;
10.    int i_padv = PADV << PARAM_INTERLACED;
11.    int align = 16;
12.    #if ARCH_X86 || ARCH_X86_64
13.        if( h->param.cpu&X264_CPU_CACHELINE_64 )
14.            align = 64;
15.        else if( h->param.cpu&X264_CPU_CACHELINE_32 || h->param.cpu&X264_CPU_AVX2 )
16.            align = 32;
17.    #endif
18.    #if ARCH_PPC
19.        int disalign = 1<<9;

```

```

20.  #endif
21.  int disalign = 1<<10;
22.  #endif
23.  //给frame分配内存, 并置零
24.  CHECKED_MALLOCZERO( frame, sizeof(x264_frame_t) );
25.  PREALLOC_INIT
26.
27.  /* allocate frame data (+64 for extra data for me) */
28.  //以像素为单位的宽高
29.  i_width  = h->mb.i_mb_width*16;
30.  i_lines  = h->mb.i_mb_height*16;
31.  i_stride = align_stride( i_width + 2*PADH, align, disalign );
32.
33.  if( i_csp == X264_CSP_NV12 || i_csp == X264_CSP_NV16 )
34.  {
35.      //YUV422,YUV420情况
36.
37.      luma_plane_count = 1;
38.      frame->i_plane = 2;
39.      for( int i = 0; i < 2; i++ )
40.      {
41.          frame->i_width[i] = i_width >> i;
42.          frame->i_lines[i] = i_lines >> (i && i_csp == X264_CSP_NV12);
43.          frame->i_stride[i] = i_stride;
44.      }
45.  }
46.  else if( i_csp == X264_CSP_I444 )
47.  {
48.      //YUV444情况
49.      luma_plane_count = 3;
50.      frame->i_plane = 3;
51.      for( int i = 0; i < 3; i++ )
52.      {
53.          frame->i_width[i] = i_width;
54.          frame->i_lines[i] = i_lines;
55.          frame->i_stride[i] = i_stride;
56.      }
57.  }
58.  else
59.      goto fail;
60.  //赋值赋值赋值...
61.  frame->i_csp = i_csp;
62.  frame->i_width_lowres = frame->i_width[0]/2;
63.  frame->i_lines_lowres = frame->i_lines[0]/2;
64.  frame->i_stride_lowres = align_stride( frame->i_width_lowres + 2*PADH, align, disalign<<1 );
65.
66.  for( int i = 0; i < h->param.i_bframe + 2; i++ )
67.      for( int j = 0; j < h->param.i_bframe + 2; j++ )
68.          PREALLOC( frame->i_row_satds[i][j], i_lines/16 * sizeof(int) );
69.
70.  frame->i_poc = -1;
71.  frame->i_type = X264_TYPE_AUTO;
72.  frame->i_qpplus1 = X264_QP_AUTO;
73.  frame->i_pts = -1;
74.  frame->i_frame = -1;
75.  frame->i_frame_num = -1;
76.  frame->i_lines_completed = -1;
77.  frame->b_fdec = b_fdec;
78.  frame->i_pic_struct = PIC_STRUCT_AUTO;
79.  frame->i_field_cnt = -1;
80.  frame->i_duration =
81.  frame->i_cpb_duration =
82.  frame->i_dpb_output_delay =
83.  frame->i_cpb_delay = 0;
84.  frame->i_coded_fields_lookahead =
85.  frame->i_cpb_delay_lookahead = -1;
86.
87.  frame->orig = frame;
88.
89.  if( i_csp == X264_CSP_NV12 || i_csp == X264_CSP_NV16 )
90.  {
91.      int chroma_padv = i_padv >> (i_csp == X264_CSP_NV12);
92.      int chroma_plane_size = (frame->i_stride[1] * (frame->i_lines[1] + 2*chroma_padv));
93.      PREALLOC( frame->buffer[1], chroma_plane_size * sizeof(pixel) );
94.      if( PARAM_INTERLACED )
95.          PREALLOC( frame->buffer_fld[1], chroma_plane_size * sizeof(pixel) );
96.  }
97.
98.  /* all 4 luma planes allocated together, since the cacheline split code
99.   * requires them to be in-phase wrt cacheline alignment. */
100.
101.  for( int p = 0; p < luma_plane_count; p++ )
102.  {
103.      int luma_plane_size = align_plane_size( frame->i_stride[p] * (frame->i_lines[p] + 2*i_padv), disalign );
104.      if( h->param.analyse.i_subpel_refine && b_fdec )
105.      {
106.          /* FIXME: Don't allocate both buffers in non-adaptive MBAFF. */
107.          PREALLOC( frame->buffer[p], 4*luma_plane_size * sizeof(pixel) );
108.          if( PARAM_INTERLACED )
109.              PREALLOC( frame->buffer_fld[p], 4*luma_plane_size * sizeof(pixel) );
110.      }
111.  }

```

```

112.     {
113.         PREALLOC( frame->buffer[p], luma_plane_size * sizeof(pixel) );
114.         if( PARAM_INTERLACED )
115.             PREALLOC( frame->buffer_fld[p], luma_plane_size * sizeof(pixel) );
116.     }
117. }
118.
119. frame->b_duplicate = 0;
120.
121. if( b_fdec ) /* fdec frame */
122. {
123.     //重建帧fdec
124.     PREALLOC( frame->mb_type, i_mb_count * sizeof(int8_t) );
125.     PREALLOC( frame->mb_partition, i_mb_count * sizeof(uint8_t) );
126.     PREALLOC( frame->mv[0], 2*16 * i_mb_count * sizeof(int16_t) );
127.     PREALLOC( frame->mv16x16, 2*(i_mb_count+1) * sizeof(int16_t) );
128.     PREALLOC( frame->ref[0], 4 * i_mb_count * sizeof(int8_t) );
129.     if( h->param.i_bframe )
130.     {
131.         PREALLOC( frame->mv[1], 2*16 * i_mb_count * sizeof(int16_t) );
132.         PREALLOC( frame->ref[1], 4 * i_mb_count * sizeof(int8_t) );
133.     }
134.     else
135.     {
136.         frame->mv[1] = NULL;
137.         frame->ref[1] = NULL;
138.     }
139.     PREALLOC( frame->i_row_bits, i_lines/16 * sizeof(int) );
140.     PREALLOC( frame->f_row_qp, i_lines/16 * sizeof(float) );
141.     PREALLOC( frame->f_row_qscale, i_lines/16 * sizeof(float) );
142.     if( h->param.analyse.i_me_method >= X264_ME_ESA )
143.         PREALLOC( frame->buffer[3], frame->i_stride[0] * (frame->i_lines[0] + 2*i_padv) * sizeof(uint16_t) << h->frames.b_have_s
ub8x8_esa );
144.     if( PARAM_INTERLACED )
145.         PREALLOC( frame->field, i_mb_count * sizeof(uint8_t) );
146.     if( h->param.analyse.b_mb_info )
147.         PREALLOC( frame->effective_qp, i_mb_count * sizeof(uint8_t) );
148. }
149. else /* fenc frame */
150. {
151.     //编码帧fenc
152.     if( h->frames.b_have_lowres )
153.     {
154.         int luma_plane_size = align_plane_size( frame->i_stride_lowres * (frame->i_lines[0]/2 + 2*PADV), disalign );
155.
156.         PREALLOC( frame->buffer_lowres[0], 4 * luma_plane_size * sizeof(pixel) );
157.
158.         for( int j = 0; j <= !h->param.i_bframe; j++ )
159.             for( int i = 0; i <= h->param.i_bframe; i++ )
160.             {
161.                 PREALLOC( frame->lowres_mv[j][i], 2*h->mb.i_mb_count*sizeof(int16_t) );
162.                 PREALLOC( frame->lowres_mv_costs[j][i], h->mb.i_mb_count*sizeof(int) );
163.             }
164.         PREALLOC( frame->i_propagate_cost, (i_mb_count+7) * sizeof(uint16_t) );
165.         for( int j = 0; j <= h->param.i_bframe+1; j++ )
166.             for( int i = 0; i <= h->param.i_bframe+1; i++ )
167.                 PREALLOC( frame->lowres_costs[j][i], (i_mb_count+3) * sizeof(uint16_t) );
168.
169.     }
170.     if( h->param.rc.i_aq_mode )
171.     {
172.         PREALLOC( frame->f_qp_offset, h->mb.i_mb_count * sizeof(float) );
173.         PREALLOC( frame->f_qp_offset_aq, h->mb.i_mb_count * sizeof(float) );
174.         if( h->frames.b_have_lowres )
175.             PREALLOC( frame->i_inv_qscale_factor, (h->mb.i_mb_count+3) * sizeof(uint16_t) );
176.     }
177. }
178.
179. PREALLOC_END( frame->base );
180.
181. if( i_csp == X264_CSP_NV12 || i_csp == X264_CSP_NV16 )
182. {
183.     int chroma_padv = i_padv >> (i_csp == X264_CSP_NV12);
184.     frame->plane[1] = frame->buffer[1] + frame->i_stride[1] * chroma_padv + PADH;
185.     if( PARAM_INTERLACED )
186.         frame->plane_fld[1] = frame->buffer_fld[1] + frame->i_stride[1] * chroma_padv + PADH;
187. }
188.
189. for( int p = 0; p < luma_plane_count; p++ )
190. {
191.     int luma_plane_size = align_plane_size( frame->i_stride[p] * (frame->i_lines[p] + 2*i_padv), disalign );
192.     if( h->param.analyse.i_subpel_refine && b_fdec )
193.     {
194.         for( int i = 0; i < 4; i++ )
195.         {
196.             frame->filtered[p][i] = frame->buffer[p] + i*luma_plane_size + frame->i_stride[p] * i_padv + PADH;
197.             frame->filtered_fld[p][i] = frame->buffer_fld[p] + i*luma_plane_size + frame->i_stride[p] * i_padv + PADH;
198.         }
199.         frame->plane[p] = frame->filtered[p][0];
200.         frame->plane_fld[p] = frame->filtered_fld[p][0];
201.     }

```

```

202.     else
203.     {
204.         frame->filtered[p][0] = frame->plane[p] = frame->buffer[p] + frame->i_stride[p] * i_padv + PADH;
205.         frame->filtered_fld[p][0] = frame->plane_fld[p] = frame->buffer_fld[p] + frame->i_stride[p] * i_padv + PADH;
206.     }
207. }
208.
209. if( b_fdec )
210. {
211.     M32( frame->mv16x16[0] ) = 0;
212.     frame->mv16x16++;
213.
214.     if( h->param.analyse.i_me_method >= X264_ME_ESA )
215.         frame->integral = (uint16_t*)frame->buffer[3] + frame->i_stride[0] * i_padv + PADH;
216. }
217. else
218. {
219.     if( h->frames.b_have_lowres )
220.     {
221.         int luma_plane_size = align_plane_size( frame->i_stride_lowres * (frame->i_lines[0]/2 + 2*PADV), disalign );
222.         for( int i = 0; i < 4; i++ )
223.             frame->lowres[i] = frame->buffer_lowres[0] + (frame->i_stride_lowres * PADV + PADH) + i * luma_plane_size;
224.
225.         for( int j = 0; j <= !h->param.i_bframe; j++ )
226.             for( int i = 0; i <= h->param.i_bframe; i++ )
227.                 memset( frame->lowres_mvs[j][i], 0, 2*h->mb.i_mb_count*sizeof(int16_t) );
228.
229.         frame->i_intra_cost = frame->lowres_costs[0][0];
230.         memset( frame->i_intra_cost, -1, (i_mb_count+3) * sizeof(uint16_t) );
231.
232.         if( h->param.rc.i_aq_mode )
233.             /* shouldn't really be initialized, just silences a valgrind false-positive in x264_mbtrees_propagate_cost_sse2 */
234.             memset( frame->i_inv_qscales_factor, 0, (h->mb.i_mb_count+3) * sizeof(uint16_t) );
235.     }
236. }
237.
238. if( x264_pthread_mutex_init( &frame->mutex, NULL ) )
239.     goto fail;
240. if( x264_pthread_cond_init( &frame->cv, NULL ) )
241.     goto fail;
242.
243. #if HAVE_OPENCL
244.     frame->opencl.ocl = h->opencl.ocl;
245. #endif
246.
247.     return frame;
248.
249. fail:
250.     x264_free( frame );
251.     return NULL;
252. }

```

从源代码中可以看出，x264\_frame\_new()声明了一个frame指针，并在后续过程中对该frame的成员变量进行内存分配和注释。需要注意的是编码帧fenc和重建帧fdec初始化的变量是不一样的——函数的输入参数b\_fdec不为0的时候初始化重建帧，否则初始化编码帧。在这个函数中涉及到一个简单的函数x264\_frame\_internal\_csp()，用于把种类繁多的外部Colorspace转换为简单的内部Colorspace。

#### x264\_frame\_internal\_csp()

x264\_frame\_internal\_csp()用于把外部Colorspace转换为内部Colorspace。该函数的定义如下所示。

```

[cpp]
1. //注意转换后只有3种内部colorspace: X264_CSP_NV12(对应YUV420), X264_CSP_NV16(对应YUV422), X264_CSP_I444(对应YUV444)
2. static int x264_frame_internal_csp( int external_csp )
3. {
4.     switch( external_csp & X264_CSP_MASK )
5.     {
6.         case X264_CSP_NV12:
7.         case X264_CSP_I420:
8.         case X264_CSP_YV12:
9.             return X264_CSP_NV12;
10.        case X264_CSP_NV16:
11.        case X264_CSP_I422:
12.        case X264_CSP_YV16:
13.        case X264_CSP_V210:
14.            return X264_CSP_NV16;
15.        case X264_CSP_I444:
16.        case X264_CSP_YV24:
17.        case X264_CSP_BGR:
18.        case X264_CSP_BGRA:
19.        case X264_CSP_RGB:
20.            return X264_CSP_I444;
21.        default:
22.            return X264_CSP_NONE;
23.    }
24. }

```

## x264\_frame\_copy\_picture()

x264\_frame\_copy\_picture()用于将外部结构体x264\_picture\_t的数据拷贝给内部结构体x264\_frame\_t。该函数的定义位于common\frame.c，如下所示。

```
[cpp]  
1. //拷贝帧数据
2. //src (外部结构体x264_picture_t) 到dst (内部结构体x264_frame_t)
3. int x264_frame_copy_picture( x264_t *h, x264_frame_t *dst, x264_picture_t *src )
4. {
5.     int i_csp = src->img.i_csp & X264_CSP_MASK;
6.     //注意转换后只有3种内部colorspace: X264_CSP_NV12(对应YUV420), X264_CSP_NV16(对应YUV422), X264_CSP_I444(对应YUV444)
7.     if( dst->i_csp != x264_frame_internal_csp( i_csp ) )
8.     {
9.         x264_log( h, X264_LOG_ERROR, "Invalid input colorspace\n" );
10.        return -1;
11.    }
12.
13.    #if HIGH_BIT_DEPTH
14.        if( !(src->img.i_csp & X264_CSP_HIGH_DEPTH) )
15.        {
16.            x264_log( h, X264_LOG_ERROR, "This build of x264 requires high depth input. Rebuild to support 8-bit input.\n" );
17.            return -1;
18.        }
19.    #else
20.        if( src->img.i_csp & X264_CSP_HIGH_DEPTH )
21.        {
22.            x264_log( h, X264_LOG_ERROR, "This build of x264 requires 8-bit input. Rebuild to support high depth input.\n" );
23.            return -1;
24.        }
25.    #endif
26.
27.    if( BIT_DEPTH != 10 && i_csp == X264_CSP_V210 )
28.    {
29.        x264_log( h, X264_LOG_ERROR, "v210 input is only compatible with bit-depth of 10 bits\n" );
30.        return -1;
31.    }
32.    //赋值赋值赋值
33.    dst->i_type = src->i_type;
34.    dst->i_qpplus1 = src->i_qpplus1;
35.    dst->i_pts = dst->i_reordered_pts = src->i_pts;
36.    dst->param = src->param;
37.    dst->i_pic_struct = src->i_pic_struct;
38.    dst->extra_sei = src->extra_sei;
39.    dst->opaque = src->opaque;
40.    dst->mb_info = h->param.analyse.b_mb_info ? src->prop.mb_info : NULL;
41.    dst->mb_info_free = h->param.analyse.b_mb_info ? src->prop.mb_info_free : NULL;
42.
43.    uint8_t *pix[3];
44.    int stride[3];
45.    if( i_csp == X264_CSP_V210 )
46.    {
47.        stride[0] = src->img.i_stride[0];
48.        pix[0] = src->img.plane[0];
49.
50.        h->mc.plane_copy_deinterleave_v210( dst->plane[0], dst->i_stride[0],
51.                                           dst->plane[1], dst->i_stride[1],
52.                                           (uint32_t *)pix[0], stride[0]/sizeof(uint32_t), h->param.i_width, h->param.i_height );
53.    }
54.    else if( i_csp >= X264_CSP_BGR )
55.    {
56.        stride[0] = src->img.i_stride[0];
57.        pix[0] = src->img.plane[0];
58.        if( src->img.i_csp & X264_CSP_VFLIP )
59.        {
60.            pix[0] += (h->param.i_height-1) * stride[0];
61.            stride[0] = -stride[0];
62.        }
63.        int b = i_csp==X264_CSP_RGB;
64.        h->mc.plane_copy_deinterleave_rgb( dst->plane[1+b], dst->i_stride[1+b],
65.                                         dst->plane[0], dst->i_stride[0],
66.                                         dst->plane[2-b], dst->i_stride[2-b],
67.                                         (pixel*)pix[0], stride[0]/sizeof(pixel), i_csp==X264_CSP_BGRA ? 4 : 3, h->param.i_width,
68.                                         h->param.i_height );
69.    }
70.    else
71.    {
72.        int v_shift = CHROMA_V_SHIFT;
73.        get_plane_ptr( h, src, &pix[0], &stride[0], 0, 0, 0 );
74.        //拷贝像素
75.        h->mc.plane_copy( dst->plane[0], dst->i_stride[0], (pixel*)pix[0],
76.                        stride[0]/sizeof(pixel), h->param.i_width, h->param.i_height );
77.        if( i_csp == X264_CSP_NV12 || i_csp == X264_CSP_NV16 )
78.        {
79.            get_plane_ptr( h, src, &pix[1], &stride[1], 1, 0, v_shift );
80.            h->mc.plane_copy( dst->plane[1], dst->i_stride[1], (pixel*)pix[1],
81.                            stride[1]/sizeof(pixel), h->param.i_width, h->param.i_height>>v_shift );
82.        }
83.    }
84. }
```

```

81.     }
82.     else if( i_csp == X264_CSP_I420 || i_csp == X264_CSP_I422 || i_csp == X264_CSP_YV12 || i_csp == X264_CSP_YV16 )
83.     {
84.         int uv_swap = i_csp == X264_CSP_YV12 || i_csp == X264_CSP_YV16;
85.         get_plane_ptr( h, src, &pix[1], &stride[1], uv_swap ? 2 : 1, 1, v_shift );
86.         get_plane_ptr( h, src, &pix[2], &stride[2], uv_swap ? 1 : 2, 1, v_shift );
87.         h->mc.plane_copy_interleave( dst->plane[1], dst->i_stride[1],
88.                                     (pixel*)pix[1], stride[1]/sizeof(pixel),
89.                                     (pixel*)pix[2], stride[2]/sizeof(pixel),
90.                                     h->param.i_width>>1, h->param.i_height>>v_shift );
91.     }
92.     else //if( i_csp == X264_CSP_I444 || i_csp == X264_CSP_YV24 )
93.     {
94.         get_plane_ptr( h, src, &pix[1], &stride[1], i_csp==X264_CSP_I444 ? 1 : 2, 0, 0 );
95.         get_plane_ptr( h, src, &pix[2], &stride[2], i_csp==X264_CSP_I444 ? 2 : 1, 0, 0 );
96.         h->mc.plane_copy( dst->plane[1], dst->i_stride[1], (pixel*)pix[1],
97.                         stride[1]/sizeof(pixel), h->param.i_width, h->param.i_height );
98.         h->mc.plane_copy( dst->plane[2], dst->i_stride[2], (pixel*)pix[2],
99.                         stride[2]/sizeof(pixel), h->param.i_width, h->param.i_height );
100.    }
101.    }
102.    return 0;
103. }

```

从源代码可以看出，x264\_frame\_t和x264\_picture\_t结构体中很多字段是一模一样的，x264\_frame\_copy\_picture()只是简单地将x264\_picture\_t中字段的值赋值给了x264\_frame\_t。

## x264\_lookahead\_put\_frame()

x264\_lookahead\_put\_frame()将编码帧放入Lookahead模块的队列中，等待确定帧类型。该函数的定义位于encoder/lookahead.c，如下所示。

```

1. //x264_frame_t放入x264_sync_frame_list_t队列
2. void x264_lookahead_put_frame( x264_t *h, x264_frame_t *frame )
3. {
4.     if( h->param.i_sync_lookahead )
5.         x264_sync_frame_list_push( &h->lookahead->ifbuf, frame );
6.     else
7.         x264_sync_frame_list_push( &h->lookahead->next, frame );//放入next队列
8. }

```

从源代码可以看出，i\_sync\_lookahead不为0的时候，会将编码帧放入lookahead.ifbuf[]中，否则会将编码帧放入lookahead.next[]中。放入帧的时候会调用x264\_sync\_frame\_list\_push()。

## x264\_sync\_frame\_list\_push()

x264\_sync\_frame\_list\_push()用于向x264\_sync\_frame\_list\_t类型的队列中放入一个帧。该函数的定义位于common/frame.c，如下所示。

```

1. void x264_sync_frame_list_push( x264_sync_frame_list_t *slist, x264_frame_t *frame )
2. {
3.     x264_pthread_mutex_lock( &slist->mutex );
4.     while( slist->i_size == slist->i_max_size )
5.         x264_pthread_cond_wait( &slist->cv_empty, &slist->mutex );
6.     //放入
7.     slist->list[ slist->i_size++ ] = frame;
8.     x264_pthread_mutex_unlock( &slist->mutex );
9.     x264_pthread_cond_broadcast( &slist->cv_fill );
10. }

```

从源代码中可以看出，x264\_sync\_frame\_list\_push()将frame放在了x264\_sync\_frame\_list\_t.list的尾部。

## x264\_lookahead\_get\_frames()

x264\_lookahead\_get\_frames()通过lookahead模块分析帧类型。该函数的定义位于encoder/lookahead.c，如下所示。



```

1. //通过lookahead分析帧类型
2. void x264_lookahead_get_frames( x264_t *h )
3. {
4.     if( h->param.i_sync_lookahead )
5.     { /* We have a lookahead thread, so get frames from there */
6.         x264_pthread_mutex_lock( &h->lookahead->obuf.mutex );
7.         while( !h->lookahead->obuf.i_size && h->lookahead->b_thread_active )
8.             x264_pthread_cond_wait( &h->lookahead->obuf.cv_fill, &h->lookahead->obuf.mutex );
9.         x264_lookahead_encoder_shift( h );
10.        x264_pthread_mutex_unlock( &h->lookahead->obuf.mutex );
11.    }
12.    else
13.    { /* We are not running a lookahead thread, so perform all the slicetype decide on the fly */
14.        //current[]必须为空, next不能为空?
15.        if( h->frames.current[0] || !h->lookahead->next.i_size )
16.            return;
17.        //分析lookahead->next->list帧的类型
18.        x264_stack_align( x264_slicetype_decide, h );
19.        //更新lookahead->last_nonb
20.        x264_lookahead_update_last_nonb( h, h->lookahead->next.list[0] );
21.        int shift_frames = h->lookahead->next.list[0]->i_bframes + 1;
22.        //lookahead->next.list移动到lookahead->obuf.list
23.        x264_lookahead_shift( &h->lookahead->obuf, &h->lookahead->next, shift_frames );
24.
25.        /* For MB-tree and VBV lookahead, we have to perform propagation analysis on I-frames too. */
26.        if( h->lookahead->b_analyse_keyframe && IS_X264_TYPE_I( h->lookahead->last_nonb->i_type ) )
27.            x264_stack_align( x264_slicetype_analyse, h, shift_frames );
28.
29.        //lookahead->obuf.list帧移动到frames->current
30.        x264_lookahead_encoder_shift( h );
31.    }
32. }

```

从源代码中可以看出, x264\_lookahead\_get\_frames()调用了x264\_slicetype\_decide()用于确定帧类型。在这里需要注意, Lookahead模块的代码量比较大, 暂时不做详细的分析, 仅简单理一下脉络。

## x264\_slicetype\_decide()

x264\_slicetype\_decide()用于确定帧类型, 该函数的定义位于encoder\slicetype.c, 如下所示。

```

1. //确定帧的类型 (I、B、P)
2. void x264_slicetype_decide( x264_t *h )
3. {
4.     x264_frame_t *frames[X264_BFRAME_MAX+2];
5.     x264_frame_t *frm;
6.     int bframes;
7.     int brefs;
8.
9.     if( !h->lookahead->next.i_size )
10.        return;
11.
12.     int lookahead_size = h->lookahead->next.i_size;
13.
14.     //遍历next队列
15.     for( int i = 0; i < h->lookahead->next.i_size; i++ )
16.     {
17.         if( h->param.b_vfr_input )
18.         {
19.             if( lookahead_size-- > 1 )
20.                 h->lookahead->next.list[i]->i_duration = 2 * (h->lookahead->next.list[i+1]->i_pts - h->lookahead->next.list[i]->i_pts);
21.             else
22.                 h->lookahead->next.list[i]->i_duration = h->i_prev_duration;
23.         }
24.         else
25.             h->lookahead->next.list[i]->i_duration = delta_tfi_divisor[h->lookahead->next.list[i]->i_pic_struct];
26.         h->i_prev_duration = h->lookahead->next.list[i]->i_duration;
27.         h->lookahead->next.list[i]->f_duration = (double)h->lookahead->next.list[i]->i_duration
28.             * h->sps->vui.i_num_units_in_tick
29.             / h->sps->vui.i_time_scale;
30.
31.         if( h->lookahead->next.list[i]->i_frame > h->i_disp_fields_last_frame && lookahead_size > 0 )
32.         {
33.             h->lookahead->next.list[i]->i_field_cnt = h->i_disp_fields;
34.             h->i_disp_fields += h->lookahead->next.list[i]->i_duration;
35.             h->i_disp_fields_last_frame = h->lookahead->next.list[i]->i_frame;
36.         }
37.         else if( lookahead_size == 0 )
38.         {
39.             h->lookahead->next.list[i]->i_field_cnt = h->i_disp_fields;
40.             h->lookahead->next.list[i]->i_duration = h->i_prev_duration;
41.         }
42.     }
43.
44.     if( h->param.rc.b_stat_read )

```

```

45. {
46.     //b_stat_read在2pass模式的第2遍才不为0
47.
48.     /* Use the frame types from the first pass */
49.     for( int i = 0; i < h->lookahead->next.i_size; i++ )
50.         h->lookahead->next.list[i]->i_type =
51.             x264_ratecontrol_slice_type( h, h->lookahead->next.list[i]->i_frame );
52. }
53. else if( (h->param.i_bframe && h->param.i_bframe_adaptive)
54.         || h->param.i_scenecut_threshold
55.         || h->param.rc.b_mb_tree
56.         || (h->param.rc.i_vbv_buffer_size && h->param.rc.i_lookahead) )
57.     x264_slicetype_analyse( h, 0 );//分析帧的类型 (I、B、P)
58. //=====
59.
60. for( bframes = 0, brefs = 0;; bframes++ )
61. {
62.     //从next队列取出1个
63.     frm = h->lookahead->next.list[bframes];
64.     //BREF的处理
65.     if( frm->i_type == X264_TYPE_BREF && h->param.i_bframe_pyramid < X264_B_PYRAMID_NORMAL &&
66.         brefs == h->param.i_bframe_pyramid )
67.     {
68.         //BREF改成B
69.         frm->i_type = X264_TYPE_B;
70.         x264_log( h, X264_LOG_WARNING, "B-ref at frame %d incompatible with B-pyramid %s \n",
71.             frm->i_frame, x264_b_pyramid_names[h->param.i_bframe_pyramid] );
72.     }
73.     /* pyramid with multiple B-refs needs a big enough dpb that the preceding P-frame stays available.
74.        smaller dpb could be supported by smart enough use of mmco, but it's easier just to forbid it. */
75.     else if( frm->i_type == X264_TYPE_BREF && h->param.i_bframe_pyramid == X264_B_PYRAMID_NORMAL &&
76.         brefs && h->param.i_frame_reference <= (brefs+3) )
77.     {
78.         frm->i_type = X264_TYPE_B;
79.         x264_log( h, X264_LOG_WARNING, "B-ref at frame %d incompatible with B-pyramid %s and %d reference frames\n",
80.             frm->i_frame, x264_b_pyramid_names[h->param.i_bframe_pyramid], h->param.i_frame_reference );
81.     }
82.     //Keyframe处理
83.     if( frm->i_type == X264_TYPE_KEYFRAME )
84.         frm->i_type = h->param.b_open_gop ? X264_TYPE_I : X264_TYPE_IDR;
85.
86.     /* Limit GOP size */
87.     if( (!h->param.b_intra_refresh || frm->i_frame == 0) && frm->i_frame - h->lookahead->i_last_keyframe >= h->param.i_keyint_ma
x )
88.     {
89.         if( frm->i_type == X264_TYPE_AUTO || frm->i_type == X264_TYPE_I )
90.             frm->i_type = h->param.b_open_gop && h->lookahead->i_last_keyframe >= 0 ? X264_TYPE_I : X264_TYPE_IDR;
91.         int warn = frm->i_type != X264_TYPE_IDR;
92.         if( warn && h->param.b_open_gop )
93.             warn &= frm->i_type != X264_TYPE_I;
94.         if( warn )
95.         {
96.             x264_log( h, X264_LOG_WARNING, "specified frame type (%d) at %d is not compatible with keyframe interval\n", frm->i_
type, frm->i_frame );
97.             frm->i_type = h->param.b_open_gop && h->lookahead->i_last_keyframe >= 0 ? X264_TYPE_I : X264_TYPE_IDR;
98.         }
99.     }
100.     if( frm->i_type == X264_TYPE_I && frm->i_frame - h->lookahead->i_last_keyframe >= h->param.i_keyint_min )
101.     {
102.         if( h->param.b_open_gop )
103.         {
104.             h->lookahead->i_last_keyframe = frm->i_frame; // Use display order
105.             if( h->param.b_bluray_compat )
106.                 h->lookahead->i_last_keyframe -= bframes; // Use bluray order
107.             frm->b_keyframe = 1;
108.         }
109.         else
110.             frm->i_type = X264_TYPE_IDR;
111.     }
112.     if( frm->i_type == X264_TYPE_IDR )
113.     {
114.         /* Close GOP */
115.         //设置当前帧为“上一个关键帧”
116.         h->lookahead->i_last_keyframe = frm->i_frame;
117.         frm->b_keyframe = 1;
118.         if( bframes > 0 )
119.         {
120.             bframes--;
121.             h->lookahead->next.list[bframes]->i_type = X264_TYPE_P;
122.         }
123.     }
124.
125.     if( bframes == h->param.i_bframe ||
126.         !h->lookahead->next.list[bframes+1] )
127.     {
128.         if( IS_X264_TYPE_B( frm->i_type ) )
129.             x264_log( h, X264_LOG_WARNING, "specified frame type is not compatible with max B-frames\n" );
130.         if( frm->i_type == X264_TYPE_AUTO
131.             || IS_X264_TYPE_B( frm->i_type ) )
132.             frm->i_type = X264_TYPE_P;
133.     }

```

```

134.
135.     if( frm->i_type == X264_TYPE_BREF )
136.         brefs++;
137.
138.     if( frm->i_type == X264_TYPE_AUTO )
139.         frm->i_type = X264_TYPE_B;
140.
141.     else if( !IS_X264_TYPE_B( frm->i_type ) ) break;
142. }
143.
144. if( bframes )
145.     h->lookahead->next.list[bframes-1]->b_last_minigop_bframe = 1;
146. h->lookahead->next.list[bframes]->i_bframes = bframes;
147.
148. /* insert a bref into the sequence */
149. if( h->param.i_bframe_pyramid && bframes > 1 && !brefs )
150. {
151.     h->lookahead->next.list[bframes/2]->i_type = X264_TYPE_BREF;
152.     brefs++;
153. }
154.
155. /* calculate the frame costs ahead of time for x264_rc_analyse_slice while we still have lowres */
156. if( h->param.rc.i_rc_method != X264_RC_CQP )
157. {
158.     x264_mb_analysis_t a;
159.     int p0, p1, b;
160.     p1 = b = bframes + 1;
161.
162.     x264_lowres_context_init( h, &a );
163.
164.     frames[0] = h->lookahead->last_nonb;
165.     memcpy( &frames[1], h->lookahead->next.list, (bframes+1) * sizeof(x264_frame_t*) );
166.     if( IS_X264_TYPE_I( h->lookahead->next.list[bframes]->i_type ) )
167.         p0 = bframes + 1;
168.     else // P
169.         p0 = 0;
170.
171.     x264_slicetype_frame_cost( h, &a, frames, p0, p1, b, 0 );
172.
173.     if( (p0 != p1 || bframes) && h->param.rc.i_vbv_buffer_size )
174.     {
175.         /* We need the intra costs for row SATDs. */
176.         x264_slicetype_frame_cost( h, &a, frames, b, b, b, 0 );
177.
178.         /* We need B-frame costs for row SATDs. */
179.         p0 = 0;
180.         for( b = 1; b <= bframes; b++ )
181.         {
182.             if( frames[b]->i_type == X264_TYPE_B )
183.                 for( p1 = b; frames[p1]->i_type == X264_TYPE_B; )
184.                     p1++;
185.             else
186.                 p1 = bframes + 1;
187.             x264_slicetype_frame_cost( h, &a, frames, p0, p1, b, 0 );
188.             if( frames[b]->i_type == X264_TYPE_BREF )
189.                 p0 = b;
190.         }
191.     }
192. }
193.
194. /* Analyse for weighted P frames */
195. if( !h->param.rc.b_stat_read && h->lookahead->next.list[bframes]->i_type == X264_TYPE_P
196.     && h->param.analyse.i_weighted_pred >= X264_WEIGHTP_SIMPLE )
197. {
198.     x264_emms();
199.     x264_weights_analyse( h, h->lookahead->next.list[bframes], h->lookahead->last_nonb, 0 );
200. }
201.
202. /* shift sequence to coded order.
203.    use a small temporary list to avoid shifting the entire next buffer around */
204. int i_coded = h->lookahead->next.list[0]->i_frame;
205. if( bframes )
206. {
207.     int idx_list[] = { brefs+1, 1 };
208.     for( int i = 0; i < bframes; i++ )
209.     {
210.         int idx = idx_list[h->lookahead->next.list[i]->i_type == X264_TYPE_BREF]++;
211.         frames[idx] = h->lookahead->next.list[i];
212.         frames[idx]->i_reordered_pts = h->lookahead->next.list[idx]->i_pts;
213.     }
214.     frames[0] = h->lookahead->next.list[bframes];
215.     frames[0]->i_reordered_pts = h->lookahead->next.list[0]->i_pts;
216.     memcpy( h->lookahead->next.list, frames, (bframes+1) * sizeof(x264_frame_t*) );
217. }
218.
219. for( int i = 0; i <= bframes; i++ )
220. {
221.     h->lookahead->next.list[i]->i_coded = i_coded++;
222.     if( i )
223.     {
224.         x264_calculate_durations( h, h->lookahead->next.list[i], h->lookahead->next.list[i-1], &h->i_cpb_delay, &h->i_coded_fiel

```

```

ds );
225.         h->lookahead->next.list[0]->f_planned_cpb_duration[i-1] = (double)h->lookahead->next.list[i]->i_cpb_duration *
226.                                     h->sps->vui.i_num_units_in_tick / h->sps->vui.i_time_scale;
227.     }
228.     else
229.         x264_calculate_durations( h, h->lookahead->next.list[i], NULL, &h->i_cpb_delay, &h->i_coded_fields );
230. }
231. }

```

x264\_slicetype\_decide()源代码比较长，还没有细看。该函数中调用了一个比较重要的函数x264\_slicetype\_analyse()。

## x264\_slicetype\_analyse()

x264\_slicetype\_analyse()用于分析帧类型。该函数的定义位于encoder\slicetype.c，如下所示。

```

[cpp]
1. //分析帧的类型 (I、B、P)
2. void x264_slicetype_analyse( x264_t *h, int intra_minigop )
3. {
4.     x264_mb_analysis_t a;
5.     x264_frame_t *frames[X264_LOOKAHEAD_MAX+3] = { NULL, };
6.     int num_frames, orig_num_frames, keyint_limit, framecnt;
7.     int i_mb_count = NUM_MBS;
8.     int cost1p0, cost2p0, cost1b1, cost2p1;
9.     // 确定最大的搜索长度
10.    // 在我的调试当中, h->lookahead->next.i_size = 4
11.    int i_max_search = X264_MIN( h->lookahead->next.i_size, X264_LOOKAHEAD_MAX );
12.    int vbv_lookahead = h->param.rc.i_vbv_buffer_size && h->param.rc.i_lookahead;
13.    /* For determinism we should limit the search to the number of frames lookahead has for sure
14.     * in h->lookahead->next.list buffer, except at the end of stream.
15.     * For normal calls with (intra_minigop == 0) that is h->lookahead->i_slicetype_length + 1 frames.
16.     * And for I-frame calls (intra_minigop != 0) we already removed intra_minigop frames from there. */
17.    if( h->param.b_deterministic )
18.        i_max_search = X264_MIN( i_max_search, h->lookahead->i_slicetype_length + 1 - intra_minigop );
19.    int keyframe = !intra_minigop;
20.
21.    assert( h->frames.b_have_lowres );
22.
23.    if( !h->lookahead->last_nonb )
24.        return;
25.    //frames[0]指向上一次的非B帧
26.    frames[0] = h->lookahead->last_nonb;
27.    //frames[] 依次指向 lookahead->next链表中的帧
28.    for( framecnt = 0; framecnt < i_max_search && h->lookahead->next.list[framecnt]->i_type == X264_TYPE_AUTO; framecnt++ )
29.        frames[framecnt+1] = h->lookahead->next.list[framecnt];
30.
31.    x264_lowres_context_init( h, &a );
32.
33.    if( !framecnt )
34.    {
35.        if( h->param.rc.b_mb_tree )
36.            x264_macroblock_tree( h, &a, frames, 0, keyframe );
37.        return;
38.    }
39.
40.    keyint_limit = h->param.i_keyint_max - frames[0]->i_frame + h->lookahead->i_last_keyframe - 1;
41.    orig_num_frames = num_frames = h->param.b_intra_refresh ? framecnt : X264_MIN( framecnt, keyint_limit );
42.
43.    /* This is important psy-wise: if we have a non-scenecut keyframe,
44.     * there will be significant visual artifacts if the frames just before
45.     * go down in quality due to being referenced less, despite it being
46.     * more RD-optimal. */
47.    if( (h->param.analyse.b_psy && h->param.rc.b_mb_tree) || vbv_lookahead )
48.        num_frames = framecnt;
49.    else if( h->param.b_open_gop && num_frames < framecnt )
50.        num_frames++;
51.    else if( num_frames == 0 )
52.    {
53.        frames[1]->i_type = X264_TYPE_I;
54.        return;
55.    }
56.
57.    int num_bframes = 0;
58.    int num_analysed_frames = num_frames;
59.    int reset_start;
60.    //通过scenecut()函数判断是否有场景切换,从而确定I帧
61.    if( h->param.i_scenecut_threshold && scenecut( h, &a, frames, 0, 1, 1, orig_num_frames, i_max_search ) )
62.    {
63.        frames[1]->i_type = X264_TYPE_I;
64.        return;
65.    }
66.
67.    #if HAVE_OPENCL
68.        x264_openc1_slicetype_prep( h, frames, num_frames, a.i_lambda );
69.    #endif
70.    //允许有B帧的时候
71.    if( h->param.i_bframe )
72.

```

```

73.         if( h->param.i_bframe_adaptive == X264_B_ADAPT_TRELLIS )
74.         {
75.             if( num_frames > 1 )
76.             {
77.                 char best_paths[X264_BFRAME_MAX+1][X264_LOOKAHEAD_MAX+1] = { "", "P" };
78.                 int best_path_index = num_frames % (X264_BFRAME_MAX+1);
79.
80.                 /* Perform the frametype analysis. */
81.                 for( int j = 2; j <= num_frames; j++ )
82.                     x264_slicetype_path( h, &a, frames, j, best_paths );
83.
84.                 num_bframes = strspn( best_paths[best_path_index], "B" );
85.                 /* Load the results of the analysis into the frame types. */
86.                 for( int j = 1; j < num_frames; j++ )
87.                     frames[j]->i_type = best_paths[best_path_index][j-1] == 'B' ? X264_TYPE_B : X264_TYPE_P;
88.             }
89.             frames[num_frames]->i_type = X264_TYPE_P;
90.         }
91.     else if( h->param.i_bframe_adaptive == X264_B_ADAPT_FAST )
92.     {
93.         for( int i = 0; i <= num_frames-2; )
94.         {
95.             //i+2作为P帧编码的代价
96.             //注：i+2始终为P帧
97.             cost2p1 = x264_slicetype_frame_cost( h, &a, frames, i+0, i+2, i+2, 1 );
98.             if( frames[i+2]->i_intra_mbs[2] > i_mb_count / 2 )
99.             {
100.                 frames[i+1]->i_type = X264_TYPE_P;
101.                 frames[i+2]->i_type = X264_TYPE_P;
102.                 i += 2;
103.                 continue;
104.             }
105.
106. #if HAVE_OPENCL
107.             if( h->param.b_opencl )
108.             {
109.                 int b_work_done = 0;
110.                 b_work_done |= x264_opencl_precalculate_frame_cost(h, frames, a.i_lambda, i+0, i+2, i+1 );
111.                 b_work_done |= x264_opencl_precalculate_frame_cost(h, frames, a.i_lambda, i+0, i+1, i+1 );
112.                 b_work_done |= x264_opencl_precalculate_frame_cost(h, frames, a.i_lambda, i+1, i+2, i+2 );
113.                 if( b_work_done )
114.                     x264_opencl_flush( h );
115.             }
116. #endif
117.
118.             //计算代价
119.             //x264_slicetype_frame_cost(,,,p0,p1,b,)
120.             //p0 b p1
121.             //p1!=b为B帧, 否则为P帧
122.
123.             // i + 1 作为B帧编码的代价
124.             cost1b1 = x264_slicetype_frame_cost( h, &a, frames, i+0, i+2, i+1, 0 );
125.             // i + 1 作为P帧编码的代价
126.             cost1p0 = x264_slicetype_frame_cost( h, &a, frames, i+0, i+1, i+1, 0 );
127.             // i + 2 作为P帧编码的代价
128.             cost2p0 = x264_slicetype_frame_cost( h, &a, frames, i+1, i+2, i+2, 0 );
129.             //如果i+1作为P帧编码的代价 + i+2作为P帧编码的代价
130.             //小于 i+1作为B帧编码的代价 + i+2作为P帧编码的代价
131.             if( cost1p0 + cost2p0 < cost1b1 + cost2p1 )
132.             {
133.                 //那么i+1将作为P帧编码
134.                 //然后直接continue
135.                 frames[i+1]->i_type = X264_TYPE_P;
136.                 i += 1;
137.                 continue;
138.             }
139.
140.             // arbitrary and untuned
141.             #define INTER_THRESH 300
142.             #define P_SENS_BIAS (50 - h->param.i_bframe_bias)
143.
144.             // i+1 将作为B帧编码
145.             frames[i+1]->i_type = X264_TYPE_B;
146.
147.             int j;
148.             for( j = i+2; j <= X264_MIN( i+h->param.i_bframe, num_frames-1 ); j++ )
149.             {
150.                 int pthresh = X264_MAX(INTER_THRESH - P_SENS_BIAS * (j-i-1), INTER_THRESH/10);
151.                 // 预测j+1作为P帧编码代价
152.                 int pcost = x264_slicetype_frame_cost( h, &a, frames, i+0, j+1, j+1, 1 );
153.                 // 如果pcost 满足下述条件, 则确定了一个P帧, 跳出循环
154.                 if( pcost > pthresh*i_mb_count || frames[j+1]->i_intra_mbs[j-i+1] > i_mb_count/3 )
155.                     break;
156.                 // 否则就是B帧
157.                 frames[j]->i_type = X264_TYPE_B;
158.             }
159.             // 将j帧确定为P帧
160.             frames[j]->i_type = X264_TYPE_P;
161.             i = j;
162.         }
163.         // 最后一帧确定为P帧
164.         frames[num_frames]->i_type = X264_TYPE_P;

```



```

7.  {
8.      int i_score = 0;
9.      int do_search[2];
10.     const x264_weight_t *w = x264_weight_none;
11.     x264_frame_t *fenc = frames[b];
12.
13.     /* Check whether we already evaluated this frame
14.      * If we have tried this frame as P, then we have also tried
15.      * the preceding frames as B. (is this still true?) */
16.     /* Also check that we already calculated the row SATDs for the current frame. */
17.     //如果已经计算过就不用算了
18.     if( fenc->i_cost_est[b-p0][p1-b] >= 0 && (!h->param.rc.i_vbv_buffer_size || fenc->i_row_satds[b-p0][p1-b][0] != -1) )
19.         i_score = fenc->i_cost_est[b-p0][p1-b];
20.     else
21.     {
22.         int dist_scale_factor = 128;
23.
24.         /* For each list, check to see whether we have lowres motion-searched this reference frame before. */
25.         do_search[0] = b != p0 && fenc->lowres_mvs[0][b-p0-1][0][0] == 0x7FFF;
26.         do_search[1] = b != p1 && fenc->lowres_mvs[1][p1-b-1][0][0] == 0x7FFF;
27.         if( do_search[0] )
28.         {
29.             if( h->param.analyse.i_weighted_pred && b == p1 )
30.             {
31.                 x264_emms();
32.                 x264_weights_analyse( h, fenc, frames[p0], 1 );
33.                 w = fenc->weight[0];
34.             }
35.             fenc->lowres_mvs[0][b-p0-1][0][0] = 0;
36.         }
37.         if( do_search[1] ) fenc->lowres_mvs[1][p1-b-1][0][0] = 0;
38.
39.         if( p1 != p0 )
40.             dist_scale_factor = ( ((b-p0) << 8) + ((p1-p0) >> 1) ) / (p1-p0);
41.
42.         int output_buf_size = h->mb.i_mb_height + (NUM_INTS + PAD_SIZE) * h->param.i_lookahead_threads;
43.         int *output_inter[X264_LOOKAHEAD_THREAD_MAX+1];
44.         int *output_intra[X264_LOOKAHEAD_THREAD_MAX+1];
45.         output_inter[0] = h->scratch_buffer2;
46.         output_intra[0] = output_inter[0] + output_buf_size;
47.
48. #if HAVE_OPENCL
49.         if( h->param.b_opencl )
50.         {
51.             x264_opengl_lowres_init(h, fenc, a->i_lambda );
52.             if( do_search[0] )
53.             {
54.                 x264_opengl_lowres_init( h, frames[p0], a->i_lambda );
55.                 x264_opengl_motionsearch( h, frames, b, p0, 0, a->i_lambda, w );
56.             }
57.             if( do_search[1] )
58.             {
59.                 x264_opengl_lowres_init( h, frames[p1], a->i_lambda );
60.                 x264_opengl_motionsearch( h, frames, b, p1, 1, a->i_lambda, NULL );
61.             }
62.             if( b != p0 )
63.                 x264_opengl_finalize_cost( h, a->i_lambda, frames, p0, p1, b, dist_scale_factor );
64.             x264_opengl_flush( h );
65.
66.             i_score = fenc->i_cost_est[b-p0][p1-b];
67.         }
68.     else
69. #endif
70.     {
71.         if( h->param.i_lookahead_threads > 1 )
72.         {
73.             x264_slicetype_slice_t s[X264_LOOKAHEAD_THREAD_MAX];
74.
75.             for( int i = 0; i < h->param.i_lookahead_threads; i++ )
76.             {
77.                 x264_t *t = h->lookahead_thread[i];
78.
79.                 /* FIXME move this somewhere else */
80.                 t->mb.i_me_method = h->mb.i_me_method;
81.                 t->mb.i_subpel_refine = h->mb.i_subpel_refine;
82.                 t->mb.b_chroma_me = h->mb.b_chroma_me;
83.
84.                 s[i] = (x264_slicetype_slice_t){ t, a, frames, p0, p1, b, dist_scale_factor, do_search, w,
85.                     output_inter[i], output_intra[i] };
86.
87.                 t->i_threadslice_start = ((h->mb.i_mb_height * i + h->param.i_lookahead_threads/2) / h-
88. >param.i_lookahead_threads);
89.                 t->i_threadslice_end = ((h->mb.i_mb_height * (i+1) + h->param.i_lookahead_threads/2) / h-
90. >param.i_lookahead_threads);
91.
92.                 int thread_height = t->i_threadslice_end - t->i_threadslice_start;
93.                 int thread_output_size = thread_height + NUM_INTS;
94.                 memset( output_inter[i], 0, thread_output_size * sizeof(int) );
95.                 memset( output_intra[i], 0, thread_output_size * sizeof(int) );
96.                 output_inter[i][NUM_ROWS] = output_intra[i][NUM_ROWS] = thread_height;
97.                 output_intra[i][1] = output_inter[i] + thread_output_size + PAD_SIZE;

```

```

90.         output_inter[i+1] = output_inter[i] + thread_output_size + PAD_SIZE;
91.         output_intra[i+1] = output_intra[i] + thread_output_size + PAD_SIZE;
92.
93.         x264_threadpool_run( h->lookaheadpool, (void*)x264_slicetype_slice_cost, &s[i] );
94.     }
95.     for( int i = 0; i < h->param.i_lookahead_threads; i++ )
96.         x264_threadpool_wait( h->lookaheadpool, &s[i] );
97. }
98.
99. else
100. {
101.     h->i_threadslice_start = 0;
102.     h->i_threadslice_end = h->mb.i_mb_height;
103.     memset( output_inter[0], 0, (output_buf_size - PAD_SIZE) * sizeof(int) );
104.     memset( output_intra[0], 0, (output_buf_size - PAD_SIZE) * sizeof(int) );
105.     output_inter[0][NUM_ROWS] = output_intra[0][NUM_ROWS] = h->mb.i_mb_height;
106.     //作为参数的结构体
107.     x264_slicetype_slice_t s = (x264_slicetype_slice_t){ h, a, frames, p0, p1, b, dist_scale_factor, do_search, w,
108.         output_inter[0], output_intra[0] };
109.     //一个slice的开销
110.     //输入输出参数都在s结构体中
111.     x264_slicetype_slice_cost( &s );
112. }
113.
114. /* Sum up accumulators */
115. if( b == p1 )
116.     fenc->i_intra_mbs[b-p0] = 0;
117. if( !fenc->b_intra_calculated )
118. {
119.     fenc->i_cost_est[0][0] = 0;
120.     fenc->i_cost_est_aq[0][0] = 0;
121. }
122. fenc->i_cost_est[b-p0][p1-b] = 0;
123. fenc->i_cost_est_aq[b-p0][p1-b] = 0;
124.
125. int *row_satd_inter = fenc->i_row_satds[b-p0][p1-b];
126. int *row_satd_intra = fenc->i_row_satds[0][0];
127. for( int i = 0; i < h->param.i_lookahead_threads; i++ )
128. {
129.     //累加output_inter[]或output_intra[]
130.     //这2个变量中存储了整帧的开销
131.     if( b == p1 )
132.         fenc->i_intra_mbs[b-p0] += output_inter[i][INTRA_MBS];
133.     if( !fenc->b_intra_calculated )
134.     {
135.         //帧内编码的代价
136.         fenc->i_cost_est[0][0] += output_intra[i][COST_EST];
137.         fenc->i_cost_est_aq[0][0] += output_intra[i][COST_EST_AQ];
138.     }
139.
140.     //帧间编码的代价
141.     fenc->i_cost_est[b-p0][p1-b] += output_inter[i][COST_EST];
142.     fenc->i_cost_est_aq[b-p0][p1-b] += output_inter[i][COST_EST_AQ];
143.
144.     if( h->param.rc.i_vbv_buffer_size )
145.     {
146.         int row_count = output_inter[i][NUM_ROWS];
147.         memcpy( row_satd_inter, output_inter[i] + NUM_INTS, row_count * sizeof(int) );
148.         if( !fenc->b_intra_calculated )
149.             memcpy( row_satd_intra, output_intra[i] + NUM_INTS, row_count * sizeof(int) );
150.         row_satd_inter += row_count;
151.         row_satd_intra += row_count;
152.     }
153. }
154. //一帧的开销
155. i_score = fenc->i_cost_est[b-p0][p1-b];
156. if( b != p1 )//B帧
157.     i_score = (uint64_t)i_score * 100 / (120 + h->param.i_bframe_bias);
158. else
159.     fenc->b_intra_calculated = 1;
160.
161. fenc->i_cost_est[b-p0][p1-b] = i_score;
162. x264_emms();
163. }
164. }
165.
166. if( b_intra_penalty )
167. {
168.     // arbitrary penalty for I-blocks after B-frames
169.     int nmb = NUM_MBS;
170.     i_score += (uint64_t)i_score * fenc->i_intra_mbs[b-p0] / (nmb * 8);
171. }
172. //返回一帧的开销值
173. return i_score;
174. }



```

从源代码可以看出，x264\_slicetype\_analyse()调用了x264\_slicetype\_slice\_cost()来计算一个slice的开销。

## x264\_slicetype\_slice\_cost()





x264\_slicetype\_slice\_cost()用来计算一个slice的开销。该函数的定义位于encoder\slicetype.c，如下所示。

```
[cpp]  
1. //一个slice的开销
2. static void x264_slicetype_slice_cost( x264_slicetype_slice_t *s )
3. {
4.     x264_t *h = s->h;
5.
6.     /* Lowres lookahead goes backwards because the MVs are used as predictors in the main encode.
7.      * This considerably improves MV prediction overall. */
8.
9.     /* The edge mbs seem to reduce the predictive quality of the
10.      * whole frame's score, but are needed for a spatial distribution. */
11.     int do_edges = h->param.rc.b_mb_tree || h->param.rc.i_vbv_buffer_size || h->mb.i_mb_width <= 2 || h->mb.i_mb_height <= 2;
12.
13.     int start_y = X264_MIN( h->i_threadslice_end - 1, h->mb.i_mb_height - 2 + do_edges );
14.     int end_y = X264_MAX( h->i_threadslice_start, 1 - do_edges );
15.     int start_x = h->mb.i_mb_width - 2 + do_edges;
16.     int end_x = 1 - do_edges;
17.
18.     //逐个计算每个MB的开销
19.     for( h->mb.i_mb_y = start_y; h->mb.i_mb_y >= end_y; h->mb.i_mb_y-- )
20.         for( h->mb.i_mb_x = start_x; h->mb.i_mb_x >= end_x; h->mb.i_mb_x-- )
21.             x264_slicetype_mb_cost( h, s->a, s->frames, s->p0, s->p1, s->b, s->dist_scale_factor,
22.                                     s->do_search, s->w, s->output_inter, s->output_intra );
23. }
```

从源代码可以看出，x264\_slicetype\_slice\_cost()循环遍历了每一个宏块，针对每一个宏块调用了x264\_slicetype\_mb\_cost()。

## x264\_slicetype\_mb\_cost()

x264\_slicetype\_mb\_cost()用于计算一个宏块的编码代价。该函数的定义位于encoder\slicetype.c，如下所示。

```
[cpp]  
1. //一个MB的开销
2. static void x264_slicetype_mb_cost( x264_t *h, x264_mb_analysis_t *a,
3.                                     x264_frame_t **frames, int p0, int p1, int b,
4.                                     int dist_scale_factor, int do_search[2], const x264_weight_t *w,
5.                                     int *output_inter, int *output_intra )
6. {
7.     x264_frame_t *fref0 = frames[p0];
8.     x264_frame_t *fref1 = frames[p1];
9.     x264_frame_t *fenc = frames[b];
10.     const int b_bidir = (b < p1);
11.     const int i_mb_x = h->mb.i_mb_x;
12.     const int i_mb_y = h->mb.i_mb_y;
13.     const int i_mb_stride = h->mb.i_mb_width;
14.     const int i_mb_xy = i_mb_x + i_mb_y * i_mb_stride;
15.     const int i_stride = fenc->i_stride_lowres;
16.     const int i_pel_offset = 8 * (i_mb_x + i_mb_y * i_stride);
17.     const int i_bipred_weight = h->param.analyse.b_weighted_bipred ? 64 - (dist_scale_factor >> 2) : 32;
18.     int16_t (*fenc_mvs[2])[2] = { &fenc->lowres_mvs[0][b-p0-1][i_mb_xy], &fenc->lowres_mvs[1][p1-b-1][i_mb_xy] };
19.     int (*fenc_costs[2]) = { &fenc->lowres_mv_costs[0][b-p0-1][i_mb_xy], &fenc->lowres_mv_costs[1][p1-b-1][i_mb_xy] };
20.     int b_frame_score_mb = (i_mb_x > 0 && i_mb_x < h->mb.i_mb_width - 1 &&
21.                             i_mb_y > 0 && i_mb_y < h->mb.i_mb_height - 1) ||
22.                             h->mb.i_mb_width <= 2 || h->mb.i_mb_height <= 2;
23.
24.     ALIGNED_ARRAY_16( pixel, pix1, [9*FDEC_STRIDE] );
25.     pixel *pix2 = pix1+8;
26.     x264_me_t m[2];
27.     int i_bcost = COST_MAX;
28.     int list_used = 0;
29.     /* A small, arbitrary bias to avoid VBV problems caused by zero-residual lookahead blocks. */
30.     int lowres_penalty = 4;
31.     //计算只涉及一个分量
32.     h->mb.pic.p_fenc[0] = h->mb.pic.fenc_buf;
33.     //从低分辨率（1/2线性内插）图像中拷贝数据
34.     h->mc.copy[PIXEL_8x8]( h->mb.pic.p_fenc[0], FENC_STRIDE, &fenc->lowres[0][i_pel_offset], i_stride, 8 );
35.
36.     if( p0 == p1 )
37.         goto lowres_intra_mb;
38.
39.     // no need for h->mb.mv_min[]
40.     h->mb.mv_limit_fpel[0][0] = -8*h->mb.i_mb_x - 4;
41.     h->mb.mv_limit_fpel[1][0] = 8*( h->mb.i_mb_width - h->mb.i_mb_x - 1 ) + 4;
42.     h->mb.mv_min_spel[0] = 4*( h->mb.mv_limit_fpel[0][0] - 8 );
43.     h->mb.mv_max_spel[0] = 4*( h->mb.mv_limit_fpel[1][0] + 8 );
44.     if( h->mb.i_mb_x >= h->mb.i_mb_width - 2 )
45.     {
46.         h->mb.mv_limit_fpel[0][1] = -8*h->mb.i_mb_y - 4;
47.         h->mb.mv_limit_fpel[1][1] = 8*( h->mb.i_mb_height - h->mb.i_mb_y - 1 ) + 4;
48.         h->mb.mv_min_spel[1] = 4*( h->mb.mv_limit_fpel[0][1] - 8 );
49.         h->mb.mv_max_spel[1] = 4*( h->mb.mv_limit_fpel[1][1] + 8 );
50.     }
51.
52. #define LOAD_HPELS_LUMA(dst, src) \
53.     { \
```

```

54.     (dst)[0] = &(src)[0][i_pel_offset]; \
55.     (dst)[1] = &(src)[1][i_pel_offset]; \
56.     (dst)[2] = &(src)[2][i_pel_offset]; \
57.     (dst)[3] = &(src)[3][i_pel_offset]; \
58. }
59. #define LOAD_WPELS_LUMA(dst,src) \
60.     (dst) = &(src)[i_pel_offset];
61.
62. #define CLIP_MV( mv ) \
63.     { \
64.         mv[0] = x264_clip3( mv[0], h->mb.mv_min_spel[0], h->mb.mv_max_spel[0] ); \
65.         mv[1] = x264_clip3( mv[1], h->mb.mv_min_spel[1], h->mb.mv_max_spel[1] ); \
66.     }
67. #define TRY_BIDIR( mv0, mv1, penalty ) \
68.     { \
69.         int i_cost; \
70.         if( h->param.analyse.i_subpel_refine <= 1 ) \
71.         { \
72.             int hpel_idx1 = (((mv0)[0]&2)>>1) + ((mv0)[1]&2); \
73.             int hpel_idx2 = (((mv1)[0]&2)>>1) + ((mv1)[1]&2); \
74.             pixel *src1 = m[0].p_fref[hpel_idx1] + ((mv0)[0]>>2) + ((mv0)[1]>>2) * m[0].i_stride[0]; \
75.             pixel *src2 = m[1].p_fref[hpel_idx2] + ((mv1)[0]>>2) + ((mv1)[1]>>2) * m[1].i_stride[0]; \
76.             h->mc.avg[PIXEL_8x8]( pix1, 16, src1, m[0].i_stride[0], src2, m[1].i_stride[0], i_bipred_weight ); \
77.         } \
78.         else \
79.         { \
80.             intptr_t stride1 = 16, stride2 = 16; \
81.             pixel *src1, *src2; \
82.             src1 = h->mc.get_ref( pix1, &stride1, m[0].p_fref, m[0].i_stride[0], \
83.                                 (mv0)[0], (mv0)[1], 8, 8, w ); \
84.             src2 = h->mc.get_ref( pix2, &stride2, m[1].p_fref, m[1].i_stride[0], \
85.                                 (mv1)[0], (mv1)[1], 8, 8, w ); \
86.             h->mc.avg[PIXEL_8x8]( pix1, 16, src1, stride1, src2, stride2, i_bipred_weight ); \
87.         } \
88.         i_cost = penalty * a->i_lambda + h->pixf.mbcmp[PIXEL_8x8]( \
89.             m[0].p_fenc[0], FENC_STRIDE, pix1, 16 ); \
90.         COPY2_IF_LT( i_bcost, i_cost, list_used, 3 ); \
91.     }
92.
93. //帧间编码 (后面还有帧内编码)
94.
95. //处理m[0]
96. m[0].i_pixel = PIXEL_8x8;
97. m[0].p_cost_mv = a->p_cost_mv;
98. m[0].i_stride[0] = i_stride;
99. m[0].p_fenc[0] = h->mb.pic.p_fenc[0];
100. m[0].weight = w;
101. m[0].i_ref = 0;
102. //加载1/2插值像素点
103. LOAD_HPELS_LUMA( m[0].p_fref, fref0->lowres );
104. m[0].p_fref_w = m[0].p_fref[0];
105. if( w[0].weightfn )
106.     LOAD_WPELS_LUMA( m[0].p_fref_w, fenc->weighted[0] );
107. //双线预测, 处理m[1]
108. if( b_bidir )
109. {
110.     int16_t *mvr = fref1->lowres_mvs[0][p1-p0-1][i_mb_xy];
111.     ALIGNED_ARRAY_8( int16_t, dmv,[2],[2] );
112.
113.     m[1].i_pixel = PIXEL_8x8;
114.     m[1].p_cost_mv = a->p_cost_mv;
115.     m[1].i_stride[0] = i_stride;
116.     m[1].p_fenc[0] = h->mb.pic.p_fenc[0];
117.     m[1].i_ref = 0;
118.     m[1].weight = x264_weight_none;
119.     LOAD_HPELS_LUMA( m[1].p_fref, fref1->lowres );
120.     m[1].p_fref_w = m[1].p_fref[0];
121.
122.     dmv[0][0] = ( mvr[0] * dist_scale_factor + 128 ) >> 8;
123.     dmv[0][1] = ( mvr[1] * dist_scale_factor + 128 ) >> 8;
124.     dmv[1][0] = dmv[0][0] - mvr[0];
125.     dmv[1][1] = dmv[0][1] - mvr[1];
126.     CLIP_MV( dmv[0] );
127.     CLIP_MV( dmv[1] );
128.     if( h->param.analyse.i_subpel_refine <= 1 )
129.         M64( dmv ) &= ~0x0001000100010001ULL; /* mv & ~1 */
130.
131.     //双向预测, 其中包含了mc.avg[PIXEL_8x8]()
132.     TRY_BIDIR( dmv[0], dmv[1], 0 );
133.     if( M64( dmv ) )
134.     {
135.         int i_cost;
136.         h->mc.avg[PIXEL_8x8]( pix1, 16, m[0].p_fref[0], m[0].i_stride[0], m[1].p_fref[0], m[1].i_stride[0], i_bipred_weight );
137.         i_cost = h->pixf.mbcmp[PIXEL_8x8]( m[0].p_fenc[0], FENC_STRIDE, pix1, 16 );
138.         COPY2_IF_LT( i_bcost, i_cost, list_used, 3 );
139.     }
140. }
141.
142. for( int l = 0; l < 1 + b_bidir; l++ )
143. {
144.     if( do_search[l] )

```

```

145.     {
146.         int i_mvc = 0;
147.         int16_t (*fenc_mv)[2] = fenc_mvs[l];
148.         ALIGNED_4( int16_t mvc[4][2] );
149.
150.         /* Reverse-order MV prediction. */
151.         M32( mvc[0] ) = 0;
152.         M32( mvc[2] ) = 0;
153. #define MVC(mv) { CP32( mvc[i_mvc], mv ); i_mvc++; }
154.         if( i_mb_x < h->mb.i_mb_width - 1 )
155.             MVC( fenc_mv[l] );
156.         if( i_mb_y < h->i_threadslice_end - 1 )
157.         {
158.             MVC( fenc_mv[i_mb_stride] );
159.             if( i_mb_x > 0 )
160.                 MVC( fenc_mv[i_mb_stride-1] );
161.             if( i_mb_x < h->mb.i_mb_width - 1 )
162.                 MVC( fenc_mv[i_mb_stride+1] );
163.         }
164. #undef MVC
165.         if( i_mvc <= 1 )
166.             CP32( m[l].mvp, mvc[0] );
167.         else
168.             x264_median_mv( m[l].mvp, mvc[0], mvc[1], mvc[2] );
169.
170.         /* Fast skip for cases of near-zero residual.  Shortcut: don't bother except in the mv0 case,
171.          * since anything else is likely to have enough residual to not trigger the skip. */
172.         if( !M32( m[l].mvp ) )
173.         {
174.             m[l].cost = h->pixf.mbcmp[PIXEL_8x8]( m[l].p_fenc[0], FENC_STRIDE, m[l].p_fref[0], m[l].i_stride[0] );
175.             if( m[l].cost < 64 )
176.             {
177.                 M32( m[l].mv ) = 0;
178.                 goto skip_motionest;
179.             }
180.         }
181.         //运动搜索, 开销存在m[l].cost中
182.         x264_me_search( h, &m[l], mvc, i_mvc );
183.         m[l].cost -= a->p_cost_mv[0]; // remove mvcost from skip mbs
184.         if( M32( m[l].mv ) )
185.             m[l].cost += 5 * a->i_lambda;
186.
187. skip_motionest:
188.         CP32( fenc_mvs[l], m[l].mv );
189.         *fenc_costs[l] = m[l].cost;
190.     }
191.     else
192.     {
193.         CP32( m[l].mv, fenc_mvs[l] );
194.         m[l].cost = *fenc_costs[l];
195.     }
196.     //如果更小就拷贝
197.     //帧内编码开销, 存储于i_bcost
198.     COPY2_IF_LT( i_bcost, m[l].cost, list_used, l+1 );
199. }
200.
201. if( b_bidir && ( M32( m[0].mv ) || M32( m[1].mv ) ) )
202.     TRY_BIDIR( m[0].mv, m[1].mv, 5 );
203.
204. lowres_intra_mb:
205.     //帧内编码
206.     if( !fenc->b_intra_calculated )
207.     {
208.         ALIGNED_ARRAY_16( pixel, edge, [36] );
209.         pixel *pix = &pixl[8+FDEC_STRIDE];
210.         pixel *src = &fenc->lowres[0][i_pel_offset];
211.         const int intra_penalty = 5 * a->i_lambda;
212.         int satds[3];
213.         int pixoff = 4 / sizeof(pixel);
214.
215.         /* Avoid store forwarding stalls by writing larger chunks */
216.         memcpy( pix-FDEC_STRIDE, src-i_stride, 16 * sizeof(pixel) );
217.         for( int i = -1; i < 8; i++ )
218.             M32( &pix[i*FDEC_STRIDE-pixoff] ) = M32( &src[i*i_stride-pixoff] );
219.
220.         //8x8块的SAD/SATD计算
221.         //x3打表计算了V, H, DC三种模式, 开销存储在satds[3]数组的3个元素中
222.         h->pixf.intra_mbcmp_x3_8x8c( h->mb.pic.p_fenc[0], pix, satds );
223.         //帧内编码开销, 存储于i_icost
224.         int i_icost = X264_MIN3( satds[0], satds[1], satds[2] );
225.
226.         if( h->param.analyse.i_subpel_refine > 1 )
227.         {
228.             h->predict_8x8c[I_PRED_CHROMA_P]( pix );
229.             int satd = h->pixf.mbcmp[PIXEL_8x8]( pix, FDEC_STRIDE, h->mb.pic.p_fenc[0], FENC_STRIDE );
230.             i_icost = X264_MIN( i_icost, satd );
231.             h->predict_8x8_filter( pix, edge, ALL_NEIGHBORS, ALL_NEIGHBORS );
232.             for( int i = 3; i < 9; i++ )
233.             {
234.                 h->predict_8x8[i]( pix, edge );
235.                 satd = h->pixf.mbcmp[PIXEL_8x8]( pix, FDEC_STRIDE, h->mb.pic.p_fenc[0], FENC_STRIDE );

```

```

236.         i_icost = X264_MIN( i_icost, satd );
237.     }
238. }
239.
240.     i_icost += intra_penalty + lowres_penalty;
241.     //存一下
242.     fenc->i_intra_cost[i_mb_xy] = i_icost;
243.     int i_icost_aq = i_icost;
244.     if( h->param.rc.i_aq_mode )
245.         i_icost_aq = ( i_icost_aq * fenc->i_inv_qscale_factor[i_mb_xy] + 128 ) >> 8;
246.     output_intra[ROW_SATD] += i_icost_aq;
247.     if( b_frame_score_mb )
248.     {
249.         //累加。[COST_EST]用于整帧的开销计算
250.         output_intra[COST_EST] += i_icost;
251.         output_intra[COST_EST_AQ] += i_icost_aq;
252.     }
253. }
254. i_bcost += lowres_penalty;
255.
256. /* forbid intra-mbs in B-frames, because it's rare and not worth checking */
257. /* FIXME: Should we still forbid them now that we cache intra scores? */
258. if( !b_bidir )
259. {
260.     int i_icost = fenc->i_intra_cost[i_mb_xy];
261.     //帧内开销比帧间更小, b_intra就会取1
262.     int b_intra = i_icost < i_bcost;
263.     if( b_intra )
264.     {
265.         //赋值给i_bcost
266.         i_bcost = i_icost;
267.         list_used = 0;
268.     }
269.     if( b_frame_score_mb )
270.         output_inter[INTRA_MBS] += b_intra; // [INTRA_MBS]统计有多少个帧内模式的宏块
271. }
272.
273. /* In an I-frame, we've already added the results above in the intra section. */
274. if( p0 != p1 )
275. {
276.     int i_bcost_aq = i_bcost;
277.     if( h->param.rc.i_aq_mode )
278.         i_bcost_aq = ( i_bcost_aq * fenc->i_inv_qscale_factor[i_mb_xy] + 128 ) >> 8;
279.     output_inter[ROW_SATD] += i_bcost_aq;
280.     if( b_frame_score_mb )
281.     {
282.         /* Don't use AQ-weighted costs for slicetype decision, only for ratecontrol. */
283.         //累加。[COST_EST]用于整帧的开销计算
284.         output_inter[COST_EST] += i_bcost;
285.         output_inter[COST_EST_AQ] += i_bcost_aq;
286.     }
287. }
288. //存储开销i_bcost
289. fenc->lowres_costs[b-p0][p1-b][i_mb_xy] = X264_MIN( i_bcost, LOWRES_COST_MASK ) + (list_used << LOWRES_COST_SHIFT);
290. }
291. #undef TRY_BIDIR

```

宏块开销这里在源代码上写了比较详细的注释，不再详细记录。在这里有一点需要注意：处理的图像是经过1/2线性差值的“低分辨率（lowres）”图片（这样速度更快？），而其中宏块的大小也是以8x8而不是16x16为单位的。

## x264\_frame\_shift()

x264\_frame\_shift()用于从队列头部取出1帧。该函数的定义位于common\frame.c，如下所示。

```



1. //从队列的头部取出一帧
2. x264_frame_t *x264_frame_shift( x264_frame_t **list )
3. {
4.     x264_frame_t *frame = list[0];
5.     int i;
6.     for( i = 0; list[i]; i++ )
7.         list[i] = list[i+1];
8.     assert(frame);
9.     return frame;
10. }

```

从源代码可以看出，x264\_frame\_shift()取出了list[0]并且作为返回值返回。

## x264\_reference\_update()

x264\_reference\_update()用于更新参考帧队列（将重建帧fdec加入参考帧队列）。该函数的定义位于encoder\encoder.c，如下所示。

```
[cpp]  
1. //更新参考帧队列，若为非参考B帧则不更新
2. //重建帧移植参考帧列表，新建一个重建帧
3. static inline int x264_reference_update( x264_t *h )
4. {
5.     //如果不是被参考的帧
6.     if( !h->fdec->b_kept_as_ref )
7.     {
8.         if( h->i_thread_frames > 1 )
9.         {
10.            x264_frame_push_unused( h, h->fdec );
11.            h->fdec = x264_frame_pop_unused( h, 1 );
12.            if( !h->fdec )
13.                return -1;
14.        }
15.        return 0;
16.    }
17.
18.    /* apply mmco from previous frame. */
19.    for( int i = 0; i < h->sh.i_mmco_command_count; i++ )
20.        for( int j = 0; h->frames.reference[j]; j++ )
21.            if( h->frames.reference[j]->i_poc == h->sh.mmco[i].i_poc )
22.                x264_frame_push_unused( h, x264_frame_shift( &h->frames.reference[j] ) );
23.
24.    /* move frame in the buffer */
25.    //重建帧加入参考帧列表
26.    x264_frame_push( h->frames.reference, h->fdec );
27.    //列表满了，则要移除1帧
28.    if( h->frames.reference[h->sps->i_num_ref_frames] )
29.        x264_frame_push_unused( h, x264_frame_shift( h->frames.reference ) );
30.    //重新初始化重建帧fdec
31.    h->fdec = x264_frame_pop_unused( h, 1 );
32.    if( !h->fdec )
33.        return -1;
34.    return 0;
35. }
```

从源代码可以看出，如果重建帧fdec是不被参考的B帧，则直接返回；如果fdec是被参考的帧，则会调用x264\_frame\_push()将该帧加入frames.reference[]队列的尾部。如果frames.reference[]已经满了，则会调用x264\_frame\_shift()和x264\_frame\_push\_unused()将frames.reference[]队列头部的帧移动到frames.unused[]队列。最后函数还会调用x264\_frame\_pop\_unused()获取一个新的重建帧fdec。

## x264\_reference\_reset()

如果编码帧为IDR帧，就会调用x264\_reference\_reset()函数清空参考帧列表。该函数定义位于encoder\encoder.c，如下所示。

```
[cpp]  
1. //清空所有参考帧
2. static inline void x264_reference_reset( x264_t *h )
3. {
4.     //把frames.reference[]中所有帧移动到frames.unused[]
5.     while( h->frames.reference[0] )
6.         x264_frame_push_unused( h, x264_frame_pop( h->frames.reference ) );
7.     h->fdec->i_poc =
8.     h->fenc->i_poc = 0;
9. }
```

从源代码可以看出，x264\_reference\_reset()中调用x264\_frame\_pop()和x264\_frame\_push\_unused()将frames.reference[]队列中的帧移动到frames.unused[]队列中。

## x264\_slice\_init()

x264\_slice\_init()用于创建Slice Header，初始化其中的信息。该函数的定义位于encoder\encoder.c，如下所示。

```

1. //创建Slice Header
2. static inline void x264_slice_init( x264_t *h, int i_nal_type, int i_global_qp )
3. {
4.     /* ----- Create slice header ----- */
5.     if( i_nal_type == NAL_SLICE_IDR )
6.     {
7.         //I帧
8.
9.         //对x264_slice_header_t进行赋值
10.        x264_slice_header_init( h, &h->sh, h->sps, h->pps, h->i_idr_pic_id, h->i_frame_num, i_global_qp );
11.
12.        /* alternate id */
13.        if( h->param.i_avcintra_class )
14.        {
15.            switch( h->i_idr_pic_id )
16.            {
17.                case 5:
18.                    h->i_idr_pic_id = 3;
19.                    break;
20.                case 3:
21.                    h->i_idr_pic_id = 4;
22.                    break;
23.                case 4:
24.                default:
25.                    h->i_idr_pic_id = 5;
26.                    break;
27.            }
28.        }
29.        else
30.            h->i_idr_pic_id ^= 1;
31.    }
32.    else
33.    {
34.        //非IDR帧
35.        x264_slice_header_init( h, &h->sh, h->sps, h->pps, -1, h->i_frame_num, i_global_qp );
36.        //参考帧列表
37.        h->sh.i_num_ref_idx_l0_active = h->i_ref[0] <= 0 ? 1 : h->i_ref[0];
38.        h->sh.i_num_ref_idx_l1_active = h->i_ref[1] <= 0 ? 1 : h->i_ref[1];
39.        if( h->sh.i_num_ref_idx_l0_active != h->pps->i_num_ref_idx_l0_default_active ||
40.            (h->sh.i_type == SLICE_TYPE_B && h->sh.i_num_ref_idx_l1_active != h->pps->i_num_ref_idx_l1_default_active) )
41.        {
42.            h->sh.b_num_ref_idx_override = 1;
43.        }
44.    }
45.
46.    if( h->fenc->i_type == X264_TYPE_BREF && h->param.b_bluray_compat && h->sh.i_mmco_command_count )
47.    {
48.        h->b_sh_backup = 1;
49.        h->sh_backup = h->sh;
50.    }
51.
52.    h->fdec->i_frame_num = h->sh.i_frame_num;
53.
54.    if( h->sps->i_poc_type == 0 )
55.    {
56.        h->sh.i_poc = h->fdec->i_poc;
57.        if( PARAM_INTERLACED )
58.        {
59.            h->sh.i_delta_poc_bottom = h->param.b_tff ? 1 : -1;
60.            h->sh.i_poc += h->sh.i_delta_poc_bottom == -1;
61.        }
62.        else
63.            h->sh.i_delta_poc_bottom = 0;
64.        h->fdec->i_delta_poc[0] = h->sh.i_delta_poc_bottom == -1;
65.        h->fdec->i_delta_poc[1] = h->sh.i_delta_poc_bottom == 1;
66.    }
67.    else
68.    {
69.        /* Nothing to do ? */
70.    }
71.    //主要对mb结构体赋初值
72.    x264_macroblock_slice_init( h );
73. }

```

从源代码可以看出，x264\_slice\_init()调用x264\_slice\_header\_init()完成了Slice Header“通用”的初始化工作，然后根据帧类型的不同，做了一些特殊参数的设置。下面看一下x264\_slice\_header\_init()。

## x264\_slice\_header\_init()

x264\_slice\_header\_init()用于对Slice Header进行初始化工作。该函数的定义如下所示。

```

1. /* Fill "default" values */
2. //对x264_slice_header_t进行赋值
3. static void x264_slice_header_init( x264_t *h, x264_slice_header_t *sh

```

```

3. static void x264_slice_header_init( x264_t *h, x264_slice_header_t *sh,
4.                                   x264_sps_t *sps, x264_pps_t *pps,
5.                                   int i_idr_pic_id, int i_frame, int i_qp )
6. {
7.     x264_param_t *param = &h->param;
8.
9.     /* First we fill all fields */
10.    sh->sps = sps;
11.    sh->pps = pps;
12.
13.    sh->i_first_mb = 0;
14.    sh->i_last_mb = h->mb.i_mb_count - 1;
15.    sh->i_pps_id = pps->i_id;
16.
17.    sh->i_frame_num = i_frame;
18.
19.    sh->b_mbaff = PARAM_INTERLACED;
20.    sh->b_field_pic = 0; /* no field support for now */
21.    sh->b_bottom_field = 0; /* not yet used */
22.
23.    sh->i_idr_pic_id = i_idr_pic_id;
24.
25.    /* poc stuff, fixed later */
26.    sh->i_poc = 0;
27.    sh->i_delta_poc_bottom = 0;
28.    sh->i_delta_poc[0] = 0;
29.    sh->i_delta_poc[1] = 0;
30.
31.    sh->i_redundant_pic_cnt = 0;
32.
33.    h->mb.b_direct_auto_write = h->param.analyse.i_direct_mv_pred == X264_DIRECT_PRED_AUTO
34.                                && h->param.i_bframe
35.                                && ( h->param.rc.b_stat_write || !h->param.rc.b_stat_read );
36.
37.    if( !h->mb.b_direct_auto_read && sh->i_type == SLICE_TYPE_B )
38.    {
39.        if( h->fref[1][0]->i_poc_l0ref0 == h->fref[0][0]->i_poc )
40.        {
41.            if( h->mb.b_direct_auto_write )
42.                sh->b_direct_spatial_mv_pred = ( h->stat.i_direct_score[1] > h->stat.i_direct_score[0] );
43.            else
44.                sh->b_direct_spatial_mv_pred = ( param->analyse.i_direct_mv_pred == X264_DIRECT_PRED_SPATIAL );
45.        }
46.        else
47.        {
48.            h->mb.b_direct_auto_write = 0;
49.            sh->b_direct_spatial_mv_pred = 1;
50.        }
51.    }
52.    /* else b_direct_spatial_mv_pred was read from the 2pass statsfile */
53.
54.    sh->b_num_ref_idx_override = 0;
55.    sh->i_num_ref_idx_l0_active = 1;
56.    sh->i_num_ref_idx_l1_active = 1;
57.
58.    sh->b_ref_pic_list_reordering[0] = h->b_ref_reorder[0];
59.    sh->b_ref_pic_list_reordering[1] = h->b_ref_reorder[1];
60.
61.    /* If the ref list isn't in the default order, construct reordering header */
62.    for( int list = 0; list < 2; list++ )
63.    {
64.        if( sh->b_ref_pic_list_reordering[list] )
65.        {
66.            int pred_frame_num = i_frame;
67.            for( int i = 0; i < h->i_ref[list]; i++ )
68.            {
69.                int diff = h->fref[list][i]->i_frame_num - pred_frame_num;
70.                sh->ref_pic_list_order[list][i].idc = ( diff > 0 );
71.                sh->ref_pic_list_order[list][i].arg = (abs(diff) - 1) & ((1 << sps->i_log2_max_frame_num) - 1);
72.                pred_frame_num = h->fref[list][i]->i_frame_num;
73.            }
74.        }
75.    }
76.
77.    sh->i_cabac_init_idc = param->i_cabac_init_idc;
78.
79.    sh->i_qp = SPEC_QP(i_qp);
80.    sh->i_qp_delta = sh->i_qp - pps->i_pic_init_qp;
81.    sh->b_sp_for_swidth = 0;
82.    sh->i_qs_delta = 0;
83.
84.    int deblock_thresh = i_qp + 2 * X264_MIN(param->i_deblocking_filter_alphac0, param->i_deblocking_filter_beta);
85.    /* If effective qp <= 15, deblocking would have no effect anyway */
86.    if( param->b_deblocking_filter && (h->mb.b_variable_qp || 15 < deblock_thresh ) )
87.        sh->i_disable_deblocking_filter_idc = param->b_sliced_threads ? 2 : 0;
88.    else
89.        sh->i_disable_deblocking_filter_idc = 1;
90.    sh->i_alpha_c0_offset = param->i_deblocking_filter_alphac0 << 1;
91.    sh->i_beta_offset = param->i_deblocking_filter_beta << 1;
92. }

```

可以看出x264\_slice\_header\_init()对x264\_slice\_header\_t结构体的成员变量进行了赋值。

## x264\_slices\_write()

编码数据（最关键的步骤）。其中调用了x264\_slice\_write()完成了编码的工作（注意“x264\_slices\_write()”和“x264\_slice\_write()”名字差了一个“s”）。

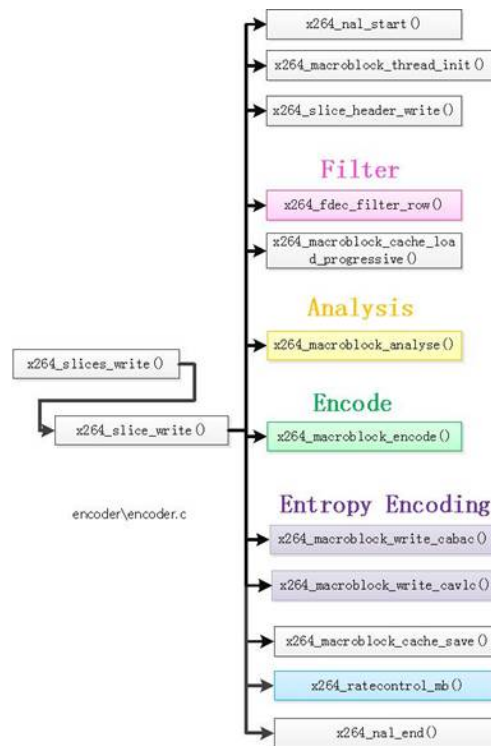
```
[cpp]
1. //真正的编码—编码1个图像帧
2. //注意“slice”后面有一个“s”
3. //它其中又调用了x264_slice_write()
4. //这一点要区分开
5. static void *x264_slices_write( x264_t *h )
6. {
7.     int i_slice_num = 0;
8.     int last_thread_mb = h->sh.i_last_mb;
9.
10.    /* init stats */
11.    memset( &h->stat.frame, 0, sizeof(h->stat.frame) );
12.    h->mb.b_reencode_mb = 0;
13.    //循环每一个slice (一幅图像可以由多个Slice构成)
14.    while( h->sh.i_first_mb + SLICE_MBAFF*h->mb.i_mb_stride <= last_thread_mb )
15.    {
16.        h->sh.i_last_mb = last_thread_mb;
17.        if( !i_slice_num || !x264_frame_new_slice( h, h->fdec ) )
18.        {
19.            if( h->param.i_slice_max_mbs )
20.            {
21.                if( SLICE_MBAFF )
22.                {
23.                    // convert first to mbaff form, add slice-max-mbs, then convert back to normal form
24.                    int last_mbaff = 2*(h->sh.i_first_mb % h->mb.i_mb_width)
25.                        + h->mb.i_mb_width*(h->sh.i_first_mb / h->mb.i_mb_width)
26.                        + h->param.i_slice_max_mbs - 1;
27.                    int last_x = (last_mbaff % (2*h->mb.i_mb_width))/2;
28.                    int last_y = (last_mbaff / (2*h->mb.i_mb_width))*2 + 1;
29.                    h->sh.i_last_mb = last_x + h->mb.i_mb_stride*last_y;
30.                }
31.                else
32.                {
33.                    h->sh.i_last_mb = h->sh.i_first_mb + h->param.i_slice_max_mbs - 1;
34.                    if( h->sh.i_last_mb < last_thread_mb && last_thread_mb - h->sh.i_last_mb < h->param.i_slice_min_mbs )
35.                        h->sh.i_last_mb = last_thread_mb - h->param.i_slice_min_mbs;
36.                }
37.                i_slice_num++;
38.            }
39.            else if( h->param.i_slice_count && !h->param.b_sliced_threads )
40.            {
41.                int height = h->mb.i_mb_height >> PARAM_INTERLACED;
42.                int width = h->mb.i_mb_width << PARAM_INTERLACED;
43.                i_slice_num++;
44.                h->sh.i_last_mb = (height * i_slice_num + h->param.i_slice_count/2) / h->param.i_slice_count * width - 1;
45.            }
46.        }
47.        h->sh.i_last_mb = X264_MIN( h->sh.i_last_mb, last_thread_mb );
48.        //真正的编码—编码1个Slice
49.        //x264_stack_align()应该是平台优化过程中内存对齐的工作
50.        //实际上就是调用x264_slice_write()
51.        if( x264_stack_align( x264_slice_write, h ) )
52.            goto fail;
53.        //注意这里对i_first_mb进行了赋值
54.        h->sh.i_first_mb = h->sh.i_last_mb + 1;
55.        // if i_first_mb is not the last mb in a row then go to the next mb in MBAFF order
56.        if( SLICE_MBAFF && h->sh.i_first_mb % h->mb.i_mb_width )
57.            h->sh.i_first_mb -= h->mb.i_mb_stride;
58.    }
59.
60.    return (void *)0;
61.
62. fail:
63.    /* Tell other threads we're done, so they wouldn't wait for it */
64.    if( h->param.b_sliced_threads )
65.        x264_threadslice_cond_broadcast( h, 2 );
66.    return (void *)-1;
67. }
```

在这里需要注意，x264\_slices\_write()调用了x264\_slice\_write()。其中x264\_slices\_write()的单位为帧，而x264\_slice\_write()的单位为Slice。最常见的情况下一个帧里面只有一个Slice，但是也有可能一个帧里面有多个Slice。

## x264\_slice\_write()

x264\_slice\_write()是完成编码工作的函数。该函数中包含了去块效应滤波，运动估计，宏块编码，熵编码等模块，它的调用结构大致如下图所示。





X264 Source Analysis - x264\_slice\_write()  
 雷霄骅 (Lei Xiaohua)  
 leixiaohual020@126.com  
<http://blog.csdn.net/leixiaohual020>

本文暂不分析x264\_slice\_write()函数。从下一篇文章开始将会对该函数进行详细的分析。

## x264\_encoder\_frame\_end()

x264\_encoder\_frame\_end()用于在编码结束后做一些后续处理，例如封装NALU（加上起始码），释放一些中间变量，记录一些统计信息等。该函数的定义位于encoder\nencoder.c，如下所示。

```

1. //结束的时候做一些处理，记录一些统计信息
2. //pp_nal: 输出的NALU
3. //pic_out: 输出的重建帧
4. static int x264_encoder_frame_end( x264_t *h, x264_t *thread_current,
5.                                   x264_nal_t **pp_nal, int *pi_nal,
6.                                   x264_picture_t *pic_out )
7. {
8.     char psz_message[80];
9.
10.    if( !h->param.b_sliced_threads && h->b_thread_active )
11.    {
12.        h->b_thread_active = 0;
13.        if( (intptr_t)x264_threadpool_wait( h->threadpool, h ) )
14.            return -1;
15.    }
16.    if( !h->out.i_nal )
17.    {
18.        pic_out->i_type = X264_TYPE_AUTO;
19.        return 0;
20.    }
21.
22.    x264_emms();
23.
24.    /* generate buffering period sei and insert it into place */
25.    if( h->i_thread_frames > 1 && h->fenc->b_keyframe && h->sps->vui.b_nal_hrd_parameters_present )
26.    {
27.        x264_hrd_fullness( h );
28.        x264_nal_start( h, NAL_SEI, NAL_PRIORITY_DISPOSABLE );
29.        x264_sei_buffering_period_write( h, &h->out.bs );
30.        if( x264_nal_end( h ) )
31.            return -1;
32.        /* buffering period sei must follow AUD, SPS and PPS and precede all other SEIs */
33.        int idx = 0;
34.        while( h->out.nal[idx].i_type == NAL_AUD ||
35.              h->out.nal[idx].i_type == NAL_SPS ||
36.              h->out.nal[idx].i_type == NAL_PPS )
37.            idx++;
38.        x264_nal_t nal_tmp = h->out.nal[h->out.i_nal-1];
39.        memmove( &h->out.nal[idx+1], &h->out.nal[idx], (h->out.i_nal-idx-1)*sizeof(x264_nal_t) );
40.        h->out.nal[idx] = nal_tmp;
41.    }
  
```

```

42. //封装一帧数据对应的NALU.
43. //例如给NALU添加起始码0x00000001
44. int frame_size = x264_encoder_encapsulate_nals( h, 0 );
45. if( frame_size < 0 )
46.     return -1;
47.
48. /* Set output picture properties */
49. //pic_out为x264_picture_t类型结构体。是libx264对外的结构体
50. //fenc,fdec是x264_frame_t类型结构体。是libx264的内部结构体
51. pic_out->i_type = h->fenc->i_type;
52.
53. pic_out->b_keyframe = h->fenc->b_keyframe;
54. pic_out->i_pic_struct = h->fenc->i_pic_struct;
55.
56. pic_out->i_pts = h->fdec->i_pts;
57. pic_out->i_dts = h->fdec->i_dts;
58.
59. if( pic_out->i_pts < pic_out->i_dts )
60.     x264_log( h, X264_LOG_WARNING, "invalid DTS: PTS is less than DTS\n" );
61.
62. pic_out->opaque = h->fenc->opaque;
63.
64. pic_out->img.i_csp = h->fdec->i_csp;
65. #if HIGH_BIT_DEPTH
66. pic_out->img.i_csp |= X264_CSP_HIGH_DEPTH;
67. #endif
68. pic_out->img.i_plane = h->fdec->i_plane;
69. //图像数据
70. for( int i = 0; i < pic_out->img.i_plane; i++ )
71. {
72.     pic_out->img.i_stride[i] = h->fdec->i_stride[i] * sizeof(pixel);
73.     pic_out->img.plane[i] = (uint8_t*)h->fdec->plane[i];
74. }
75. //回收用过的编码帧fenc
76. x264_frame_push_unused( thread_current, h->fenc );
77.
78. /* ----- Update encoder state ----- */
79.
80. /* update rc */
81. int filler = 0;
82. if( x264_ratecontrol_end( h, frame_size * 8, &filler ) < 0 )
83.     return -1;
84.
85. pic_out->hrd_timing = h->fenc->hrd_timing;
86. pic_out->prop.f_crf_avg = h->fdec->f_crf_avg;
87.
88. /* Filler in AVC-Intra mode is written as zero bytes to the last slice
89.  * We don't know the size of the last slice until encapsulation so we add filler to the encapsulated NAL */
90. if( h->param.i_avcintra_class )
91. {
92.     x264_t *h0 = h->thread[0];
93.     int ret = x264_check_encapsulated_buffer( h, h0, h->out.i_nal, frame_size, frame_size + filler );
94.     if( ret < 0 )
95.         return -1;
96.     memset( h->out.nal[0].p_payload + frame_size, 0, filler );
97.     h->out.nal[h->out.i_nal-1].i_payload += filler;
98.     h->out.nal[h->out.i_nal-1].i_padding = filler;
99.     frame_size += filler;
100. }
101. else
102. {
103.     while( filler > 0 )
104.     {
105.         int f, overhead;
106.         overhead = (FILLER_OVERHEAD - h->param.b_annexb);
107.         if( h->param.i_slice_max_size && filler > h->param.i_slice_max_size )
108.         {
109.             int next_size = filler - h->param.i_slice_max_size;
110.             int overflow = X264_MAX( overhead - next_size, 0 );
111.             f = h->param.i_slice_max_size - overhead - overflow;
112.         }
113.         else
114.             f = X264_MAX( 0, filler - overhead );
115.
116.         if( x264_bitstream_check_buffer_filler( h, f ) )
117.             return -1;
118.         x264_nal_start( h, NAL_FILLER, NAL_PRIORITY_DISPOSABLE );
119.         x264_filler_write( h, &h->out.bs, f );
120.         if( x264_nal_end( h ) )
121.             return -1;
122.         int total_size = x264_encoder_encapsulate_nals( h, h->out.i_nal-1 );
123.         if( total_size < 0 )
124.             return -1;
125.         frame_size += total_size;
126.         filler -= total_size;
127.     }
128. }
129.
130. /* End bitstream, set output */
131. *pi_nal = h->out.i_nal;
132. *pp_nal = h->out.nal;

```

```

133.
134.     h->out.i_nal = 0;
135.
136.     x264_noise_reduction_update( h );
137.
138.     /* ----- Compute/Print statistics ----- */
139.     x264_thread_sync_stat( h, h->thread[0] );
140.
141.     /* Slice stat */
142.     //stat中存储了统计信息
143.     //帧数+1 (根据类型)
144.     h->stat.i_frame_count[h->sh.i_type]++;
145.     //帧大小
146.     h->stat.i_frame_size[h->sh.i_type] += frame_size;
147.     h->stat.f_frame_qp[h->sh.i_type] += h->fdec->f_qp_avg_aq;
148.     //统计MB个数, 把不同类型的累加起来
149.     for( int i = 0; i < X264_MBTYPE_MAX; i++ )
150.         h->stat.i_mb_count[h->sh.i_type][i] += h->stat.frame.i_mb_count[i];
151.     for( int i = 0; i < X264_PARTTYPE_MAX; i++ )
152.         h->stat.i_mb_partition[h->sh.i_type][i] += h->stat.frame.i_mb_partition[i];
153.     for( int i = 0; i < 2; i++ )
154.         h->stat.i_mb_count_8x8dct[i] += h->stat.frame.i_mb_count_8x8dct[i];
155.     for( int i = 0; i < 6; i++ )
156.         h->stat.i_mb_cbp[i] += h->stat.frame.i_mb_cbp[i];
157.     for( int i = 0; i < 4; i++ )
158.         for( int j = 0; j < 13; j++ )
159.             h->stat.i_mb_pred_mode[i][j] += h->stat.frame.i_mb_pred_mode[i][j];
160.     if( h->sh.i_type != SLICE_TYPE_I )
161.         for( int i_list = 0; i_list < 2; i_list++ )
162.             for( int i = 0; i < X264_REF_MAX*2; i++ )
163.                 h->stat.i_mb_count_ref[h->sh.i_type][i_list][i] += h->stat.frame.i_mb_count_ref[i_list][i];
164.     for( int i = 0; i < 3; i++ )
165.         h->stat.i_mb_field[i] += h->stat.frame.i_mb_field[i];
166.     if( h->sh.i_type == SLICE_TYPE_P && h->param.analyse.i_weighted_pred >= X264_WEIGHTP_SIMPLE )
167.     {
168.         h->stat.i_wpred[0] += !!h->sh.weight[0][0].weightfn;
169.         h->stat.i_wpred[1] += !!h->sh.weight[0][1].weightfn || !!h->sh.weight[0][2].weightfn;
170.     }
171.     if( h->sh.i_type == SLICE_TYPE_B )
172.     {
173.         h->stat.i_direct_frames[ h->sh.b_direct_spatial_mv_pred ] ++;
174.         if( h->mb.b_direct_auto_write )
175.         {
176.             //FIXME somewhat arbitrary time constants
177.             if( h->stat.i_direct_score[0] + h->stat.i_direct_score[1] > h->mb.i_mb_count )
178.                 for( int i = 0; i < 2; i++ )
179.                     h->stat.i_direct_score[i] = h->stat.i_direct_score[i] * 9/10;
180.             for( int i = 0; i < 2; i++ )
181.                 h->stat.i_direct_score[i] += h->stat.frame.i_direct_score[i];
182.         }
183.     }
184.     else
185.         h->stat.i_consecutive_bframes[h->fenc->i_bframes]++;
186.
187.     psz_message[0] = '\0';
188.     double dur = h->fenc->f_duration;
189.     h->stat.f_frame_duration[h->sh.i_type] += dur;
190.
191.     //需要计算PSNR
192.     if( h->param.analyse.b_psnr )
193.     {
194.         //SSD (Sum of Squared Difference) 即差值的平方和
195.         int64_t ssd[3] =
196.         {
197.             h->stat.frame.i_ssd[0],
198.             h->stat.frame.i_ssd[1],
199.             h->stat.frame.i_ssd[2],
200.         };
201.         int luma_size = h->param.i_width * h->param.i_height;
202.         int chroma_size = CHROMA_SIZE( luma_size );
203.
204.         //SSD是已经在“滤波”环节计算过的
205.         //SSD简单换算成PSNR, 调用x264_psnr()
206.         pic_out->prop.f_psnr[0] = x264_psnr( ssd[0], luma_size );
207.         pic_out->prop.f_psnr[1] = x264_psnr( ssd[1], chroma_size );
208.         pic_out->prop.f_psnr[2] = x264_psnr( ssd[2], chroma_size );
209.         //平均值
210.         pic_out->prop.f_psnr_avg = x264_psnr( ssd[0] + ssd[1] + ssd[2], luma_size + chroma_size*2 );
211.         //mean系列的累加
212.         h->stat.f_ssd_global[h->sh.i_type] += dur * (ssd[0] + ssd[1] + ssd[2]);
213.         h->stat.f_psnr_average[h->sh.i_type] += dur * pic_out->prop.f_psnr_avg;
214.         h->stat.f_psnr_mean_y[h->sh.i_type] += dur * pic_out->prop.f_psnr[0];
215.         h->stat.f_psnr_mean_u[h->sh.i_type] += dur * pic_out->prop.f_psnr[1];
216.         h->stat.f_psnr_mean_v[h->sh.i_type] += dur * pic_out->prop.f_psnr[2];
217.
218.         snprintf( psz_message, 80, " PSNR Y:%5.2f U:%5.2f V:%5.2f", pic_out->prop.f_psnr[0],
219.                 pic_out->prop.f_psnr[1],
220.                 pic_out->prop.f_psnr[2] );
221.     }
222.
223.     //需要计算SSIM
224.     if( h->param.analyse.b_ssim )

```

```

224.         int( n->param.analyse.o_ssim )
225.     {
226.         //SSIM是已经在“滤波”环节计算过的
227.         pic_out->prop.f_ssim = h->stat.frame.f_ssim / h->stat.frame.i_ssim_cnt;
228.         //mean系列的需要累加
229.         h->stat.f_ssim_mean_y[h->sh.i_type] += pic_out->prop.f_ssim * dur;
230.         snprintf( psz_message + strlen(psz_message), 80 - strlen(psz_message),
231.             " SSIM Y:%.5f", pic_out->prop.f_ssim );
232.     }
233.     psz_message[79] = '\0';
234.     //Debug时候输出
235.     x264_log( h, X264_LOG_DEBUG,
236.         "frame=%4d QP=%.2f NAL=%d Slice:%c Poc:%-3d I:%-4d P:%-4d SKIP:%-4d size=%d bytes%s\n",
237.             h->i_frame,
238.             h->fdec->f_qp_avg_aq,
239.             h->i_nal_ref_idc,
240.             h->sh.i_type == SLICE_TYPE_I ? 'I' : (h->sh.i_type == SLICE_TYPE_P ? 'P' : 'B' ),
241.             h->fdec->i_poc,
242.             h->stat.frame.i_mb_count_i,
243.             h->stat.frame.i_mb_count_p,
244.             h->stat.frame.i_mb_count_skip,
245.             frame_size,
246.             psz_message );
247.
248.     // keep stats all in one place
249.     x264_thread_sync_stat( h->thread[0], h );
250.     // for the use of the next frame
251.     x264_thread_sync_stat( thread_current, h );
252.
253. #ifdef DEBUG_MB_TYPE
254. {
255.     static const char mb_chars[] = { 'i', 'i', 'I', 'C', 'P', '8', 'S',
256.         'D', '<', 'X', 'B', 'X', '>', 'B', 'B', 'B', 'B', '8', 'S' };
257.     for( int mb_xy = 0; mb_xy < h->mb.i_mb_width * h->mb.i_mb_height; mb_xy++ )
258.     {
259.         if( h->mb.type[mb_xy] < X264_MBTYPE_MAX && h->mb.type[mb_xy] >= 0 )
260.             fprintf( stderr, "%c ", mb_chars[ h->mb.type[mb_xy] ] );
261.         else
262.             fprintf( stderr, "? " );
263.
264.         if( (mb_xy+1) % h->mb.i_mb_width == 0 )
265.             fprintf( stderr, "\n" );
266.     }
267. }
268. #endif
269.
270. /* Remove duplicates, must be done near the end as breaks h->fref0 array
271.  * by freeing some of its pointers. */
272. for( int i = 0; i < h->i_ref[0]; i++ )
273.     if( h->fref[0][i] && h->fref[0][i]->b_duplicate )
274.     {
275.         x264_frame_push_blank_unused( h, h->fref[0][i] );
276.         h->fref[0][i] = 0;
277.     }
278.
279. if( h->param.psz_dump_yuv )
280.     x264_frame_dump( h );
281. x264_emms();
282.
283. return frame_size;
284. }

```

从源代码可以看出，x264\_encoder\_frame\_end()中大部分代码用于把统计信息记录到x264\_t的stat中。此外做了一些后续处理：调用了x264\_encoder\_encapsulate\_nals()封装NALU（添加起始码），调用x264\_frame\_push\_unused()将fenc重新放回frames.unused[]队列，并且调用x264\_ratecontrol\_end()结束码率控制。

## x264\_encoder\_encapsulate\_nals()

x264\_encoder\_encapsulate\_nals()用于封装一帧数据对应的NALU，其代码如下所示。

```

1. //封装一帧数据对应的NALU.
2. //例如给NALU添加起始码0x00000001
3. static int x264_encoder_encapsulate_nals( x264_t *h, int start )
4. {
5.     x264_t *h0 = h->thread[0];
6.     int nal_size = 0, previous_nal_size = 0;
7.
8.     if( h->param.nalu_process )
9.     {
10.         for( int i = start; i < h->out.i_nal; i++ )
11.             nal_size += h->out.nal[i].i_payload;
12.         return nal_size;
13.     }
14.
15.     for( int i = 0; i < start; i++ )
16.         previous_nal_size += h->out.nal[i].i_payload;
17.
18.     for( int i = start; i < h->out.i_nal; i++ )
19.         nal_size += h->out.nal[i].i_payload;
20.
21.     /* Worst-case NAL unit escaping: reallocate the buffer if it's too small. */
22.     int necessary_size = previous_nal_size + nal_size * 3/2 + h->out.i_nal * 4 + 4 + 64;
23.     for( int i = start; i < h->out.i_nal; i++ )
24.         necessary_size += h->out.nal[i].i_padding;
25.     if( x264_check_encapsulated_buffer( h, h0, start, previous_nal_size, necessary_size ) )
26.         return -1;
27.
28.     uint8_t *nal_buffer = h0->nal_buffer + previous_nal_size;
29.
30.     //一个一个NALU处理
31.     for( int i = start; i < h->out.i_nal; i++ )
32.     {
33.         int old_payload_len = h->out.nal[i].i_payload;
34.         h->out.nal[i].b_long_startcode = !i || h->out.nal[i].i_type == NAL_SPS || h->out.nal[i].i_type == NAL_PPS ||
35.             h->param.i_avcintra_class;
36.         //添加起始码
37.         x264_nal_encode( h, nal_buffer, &h->out.nal[i] );
38.         nal_buffer += h->out.nal[i].i_payload;
39.         if( h->param.i_avcintra_class )
40.         {
41.             h->out.nal[i].i_padding -= h->out.nal[i].i_payload - (old_payload_len + NALU_OVERHEAD);
42.             if( h->out.nal[i].i_padding > 0 )
43.             {
44.                 memset( nal_buffer, 0, h->out.nal[i].i_padding );
45.                 nal_buffer += h->out.nal[i].i_padding;
46.                 h->out.nal[i].i_payload += h->out.nal[i].i_padding;
47.             }
48.             h->out.nal[i].i_padding = X264_MAX( h->out.nal[i].i_padding, 0 );
49.         }
50.     }
51.
52.     x264_emms();
53.
54.     return nal_buffer - (h0->nal_buffer + previous_nal_size);
55. }

```

从源代码中可以看出，x264\_encoder\_encapsulate\_nals()调用了另外一个函数x264\_nal\_encode()逐个给一帧数据中的各个NALU添加起始码以及NALU Header等。

#### x264\_nal\_encode()

x264\_nal\_encode()用于给NALU添加起始码以及NALU Header等。该函数的定义位于common\bitstream.c，如下所示。

```

1.  /*****
2.  * x264_nal_encode:
3.  *****/
4.  //添加起始码
5.  void x264_nal_encode( x264_t *h, uint8_t *dst, x264_nal_t *nal )
6.  {
7.      uint8_t *src = nal->p_payload;
8.      uint8_t *end = nal->p_payload + nal->i_payload;
9.      uint8_t *orig_dst = dst;
10.     //起始码 =====
11.     //annexb格式, 起始码为0x00000001
12.     if( h->param.b_annexb )
13.     {
14.         if( nal->b_long_startcode )
15.             *dst++ = 0x00;
16.             *dst++ = 0x00;
17.             *dst++ = 0x00;
18.             *dst++ = 0x01;
19.         }
20.         else /* save room for size later */
21.             dst += 4; //mp4格式
22.
23.         //NALU Header =====
24.         /* nal header */
25.         *dst++ = ( 0x00 << 7 ) | ( nal->i_ref_idc << 5 ) | nal->i_type;
26.
27.         dst = h->bsf.nal_escape( dst, src, end );
28.         int size = (dst - orig_dst) - 4;
29.
30.         /* Write the size header for mp4/etc */
31.         //重新回到起始码的位置, 写入mp4格式的起始码 (size大小, 不含起始码)
32.         if( !h->param.b_annexb )
33.         {
34.             /* Size doesn't include the size of the header we're writing now. */
35.             orig_dst[0] = size>>24;
36.             orig_dst[1] = size>>16;
37.             orig_dst[2] = size>> 8;
38.             orig_dst[3] = size>> 0;
39.         }
40.         //NALU负载大小, 包含起始码
41.         nal->i_payload = size+4;
42.         nal->p_payload = orig_dst;
43.         x264_emms();
44.     }

```

从源代码可以看出，x264\_nal\_encode()给NALU数据添加了起始码以及NALU Header。在这里简单总结一下起始码的添加过程。

H.264码流有两种格式：

(1) annexb模式（传统模式）。这种模式下每个NALU包含起始码0x00000001；而且SPS、PPS存储在ES码流中。常见的H.264裸流就是属于这种格式。

(2) mp4模式。这种模式下每个NALU不包含起始码，原本存储起始码前4个字节存储的是这个NALU的长度（不包含前4字节）；而且SPS、PPS被单独放在容器的其他位置上。这种H.264一般存储在某些容器中，例如MP4中。

从源代码中可以看出，x264\_nal\_encode()根据H.264码流格式的不同分成两种情况给NALU添加起始码：

(1) annexb模式下，在每个NALU前面添加0x00000001。

(2) mp4模式下，先计算NALU的长度（不包含前4字节），再将长度信息写入NALU前面的4个字节

至此有关编码器主干部分有关x264\_encoder\_encode()的源代码就分析完了。从下一篇文章开始将会开始分析编码Slice的函数——x264\_slice\_write()。

**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/45719905>

文章标签： x264 编码 H.264 Slice NALU

个人分类： x264

所属专栏： 开源多媒体项目源代码分析

