

原 FFMpeg的H.264解码器源代码简单分析：环路滤波（Loop Filter）部分

2015年04月23日 18:16:26 阅读数：10652

=====

H.264源代码分析文章列表：

[【编码 - x264】](#)

[x264源代码简单分析：概述](#)

[x264源代码简单分析：x264命令行工具（x264.exe）](#)

[x264源代码简单分析：编码器主干部分-1](#)

[x264源代码简单分析：编码器主干部分-2](#)

[x264源代码简单分析：x264_slice_write\(\)](#)

[x264源代码简单分析：滤波（Filter）部分](#)

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

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

[x264源代码简单分析：宏块编码（Encode）部分](#)

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

[FFmpeg与libx264接口源代码简单分析](#)

[【解码 - libavcodec H.264 解码器】](#)

[FFmpeg的H.264解码器源代码简单分析：概述](#)

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

[FFmpeg的H.264解码器源代码简单分析：解码器主干部分](#)

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

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

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

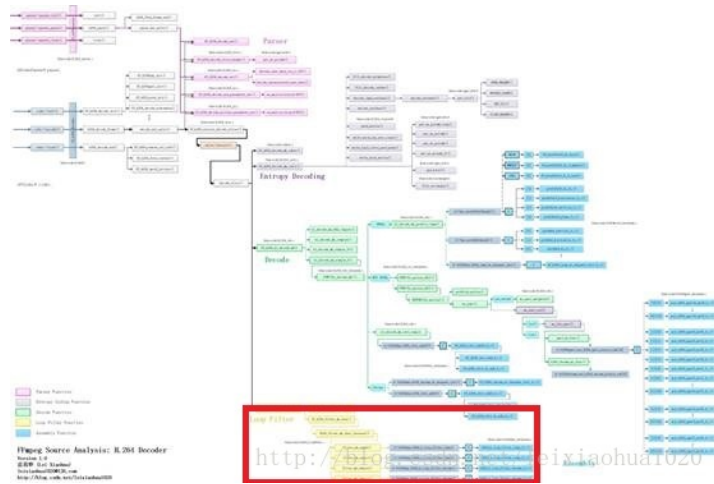
[FFmpeg的H.264解码器源代码简单分析：环路滤波（Loop Filter）部分](#)

=====

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

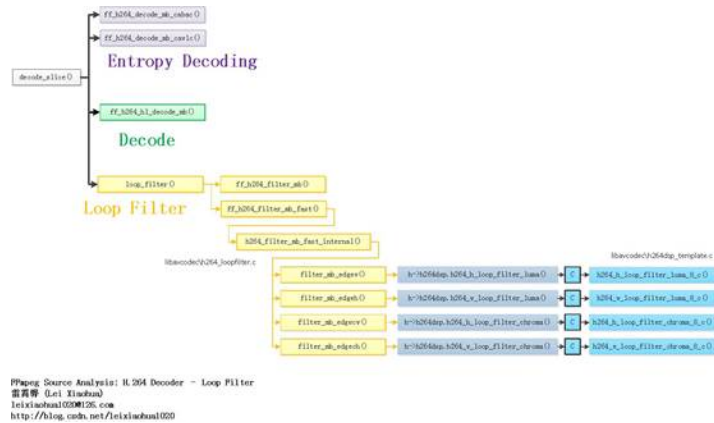
函数调用关系图

环路滤波（Loop Filter）部分的源代码在整个H.264解码器中的位置如下图所示。



[单击查看更清晰的图片](#)

环路滤波（Loop Filter）部分的源代码的调用关系如下图所示。



[单击查看更清晰的图片](#)

环路滤波主要用于滤除方块效应。decode_slice()在解码完一行宏块之后，会调用loop_filter()函数完成环路滤波功能。loop_filter()函数会遍历该行宏块中的每一个宏块，并且针对每一个宏块调用ff_h264_filter_mb_fast()。ff_h264_filter_mb_fast()又会调用h264_filter_mb_fast_internal()。

h264_filter_mb_fast_internal()完成了一个宏块的环路滤波工作。该函数调用filter_mb_edges()和filter_mb_edgesh()对亮度垂直边界和水平边界进行滤波，或者调用filter_mb_edgescv()和filter_mb_edgesch()对色度的垂直边界和水平边界进行滤波。

下面首先回顾一下decode_slice()函数。

decode_slice()

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

```
[cpp]
1. //解码slice
2. //三个主要步骤：
3. //1. 熵解码 (CAVLC/CABAC)
4. //2. 宏块解码
5. //3. 环路滤波
6. //此外还包含了错误隐藏代码
7. static int decode_slice(struct AVCodecContext *avctx, void *arg)
8. {
9.     H264Context *h = *(void **)arg;
10.    int lf_x_start = h->mb_x;
11.
12.    h->mb_skip_run = -1;
13.
14.    av_assert0(h->block_offset[15] == (4 * ((scan8[15] - scan8[0]) & 7) << h->pixel_shift) + 4 * h->linesize * ((scan8[15] - scan8[0]) >> 3));
15.
16.    h->is_complex = FRAME_MBAFF(h) || h->picture_structure != PICT_FRAME ||
17.        avctx->codec_id != AV_CODEC_ID_H264 ||
18.        (CONFIG_GRAY && (h->flags & CODEC_FLAG_GRAY));
19.
20.    if (!(h->avctx->active_thread_type & FF_THREAD_SLICE) && h->picture_structure == PICT_FRAME && h->er.error_status_table) {
21.        const int start_i = av_clip(h->resync_mb_x + h->resync_mb_y * h->mb_width, 0, h->mb_num - 1);
22.        if (start_i) {
23.            int prev_status = h->er.error_status_table[h->er.mb_index2xy[start_i - 1]];
```

```

24.     prev_status &= ~ VP_START;
25.     if (prev_status != (ER_MV_END | ER_DC_END | ER_AC_END))
26.         h->er.error_occurred = 1;
27.     }
28. }
29. //CABAC情况
30. if (h->pps.cabac) {
31.     /* realign */
32.     align_get_bits(&h->gb);
33.
34.     /* init cabac */
35.     //初始化CABAC解码器
36.     ff_init_cabac_decoder(&h->cabac,
37.                           h->gb.buffer + get_bits_count(&h->gb) / 8,
38.                           (get_bits_left(&h->gb) + 7) / 8);
39.
40.     ff_h264_init_cabac_states(h);
41.     //循环处理每个宏块
42.     for (;;) {
43.         // START_TIMER
44.         //解码CABAC数据
45.         int ret = ff_h264_decode_mb_cabac(h);
46.         int eos;
47.         // STOP_TIMER("decode_mb_cabac")
48.         //解码宏块
49.         if (ret >= 0)
50.             ff_h264_hl_decode_mb(h);
51.
52.         // FIXME optimal? or let mb_decode decode 16x32 ?
53.         //宏块级帧场自适应。很少接触
54.         if (ret >= 0 && FRAME_MBAFF(h)) {
55.             h->mb_y++;
56.
57.             ret = ff_h264_decode_mb_cabac(h);
58.             //解码宏块
59.             if (ret >= 0)
60.                 ff_h264_hl_decode_mb(h);
61.             h->mb_y--;
62.         }
63.         eos = get_cabac_terminate(&h->cabac);
64.
65.         if ((h->workaround_bugs & FF_BUG_TRUNCATED) &&
66.             h->cabac.bytestream > h->cabac.bytestream_end + 2) {
67.             //错误隐藏
68.             er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x - 1,
69.                          h->mb_y, ER_MB_END);
70.             if (h->mb_x >= lf_x_start)
71.                 loop_filter(h, lf_x_start, h->mb_x + 1);
72.             return 0;
73.         }
74.         if (h->cabac.bytestream > h->cabac.bytestream_end + 2 )
75.             av_log(h->avctx, AV_LOG_DEBUG, "bytestream overread %"PTRDIFF_SPECIFIER"\n", h->cabac.bytestream_end - h->cabac.bytestream);
76.         if (ret < 0 || h->cabac.bytestream > h->cabac.bytestream_end + 4) {
77.             av_log(h->avctx, AV_LOG_ERROR,
78.                  "error while decoding MB %d %d, bytestream %"PTRDIFF_SPECIFIER"\n",
79.                  h->mb_x, h->mb_y,
80.                  h->cabac.bytestream_end - h->cabac.bytestream);
81.             er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x,
82.                          h->mb_y, ER_MB_ERROR);
83.             return AVERERROR_INVALIDDATA;
84.         }
85.         //mb_x自增
86.         //如果自增后超过了一行的mb个数
87.         if (++h->mb_x >= h->mb_width) {
88.             //环路滤波
89.             loop_filter(h, lf_x_start, h->mb_x);
90.             h->mb_x = lf_x_start = 0;
91.             decode_finish_row(h);
92.             //mb_y自增 (处理下一行)
93.             ++h->mb_y;
94.             //宏块级帧场自适应，暂不考虑
95.             if (FIELD_OR_MBAFF_PICTURE(h)) {
96.                 ++h->mb_y;
97.                 if (FRAME_MBAFF(h) && h->mb_y < h->mb_height)
98.                     predict_field_decoding_flag(h);
99.             }
100.        }
101.        //如果mb_y超过了mb的行数
102.        if (eos || h->mb_y >= h->mb_height) {
103.            tprintf(h->avctx, "slice end %d %d\n",
104.                   get_bits_count(&h->gb), h->gb.size_in_bits);
105.            er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x - 1,
106.                         h->mb_y, ER_MB_END);
107.            if (h->mb_x > lf_x_start)
108.                loop_filter(h, lf_x_start, h->mb_x);
109.            return 0;
110.        }
111.    }
112. } else {
113.     //CAVLC情况

```

```

114. //循环处理每个宏块
115. for (;;) {
116.     //解码宏块的CAVLC
117.     int ret = ff_h264_decode_mb_cavlc(h);
118.     //解码宏块
119.     if (ret >= 0)
120.         ff_h264_hl_decode_mb(h);
121.
122.     // FIXME optimal? or let mb_decode decode 16x32 ?
123.     if (ret >= 0 && FRAME_MBAFF(h)) {
124.         h->mb_y++;
125.         ret = ff_h264_decode_mb_cavlc(h);
126.
127.         if (ret >= 0)
128.             ff_h264_hl_decode_mb(h);
129.         h->mb_y--;
130.     }
131.
132.     if (ret < 0) {
133.         av_log(h->avctx, AV_LOG_ERROR,
134.             "error while decoding MB %d %d\n", h->mb_x, h->mb_y);
135.         er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x,
136.             h->mb_y, ER_MB_ERROR);
137.         return ret;
138.     }
139.
140.     if (++h->mb_x >= h->mb_width) {
141.         //环路滤波
142.         loop_filter(h, lf_x_start, h->mb_x);
143.         h->mb_x = lf_x_start = 0;
144.         decode_finish_row(h);
145.         ++h->mb_y;
146.         if (FIELD_OR_MBAFF_PICTURE(h)) {
147.             ++h->mb_y;
148.             if (FRAME_MBAFF(h) && h->mb_y < h->mb_height)
149.                 predict_field_decoding_flag(h);
150.         }
151.         if (h->mb_y >= h->mb_height) {
152.             tprintf(h->avctx, "slice end %d %d\n",
153.                 get_bits_count(&h->gb), h->gb.size_in_bits);
154.
155.             if (get_bits_left(&h->gb) == 0
156.                 || get_bits_left(&h->gb) > 0 && !(h->avctx->err_recognition & AV_EF_AGGRESSIVE)) {
157.                 //错误隐藏
158.                 er_add_slice(h, h->resync_mb_x, h->resync_mb_y,
159.                     h->mb_x - 1, h->mb_y, ER_MB_END);
160.
161.                 return 0;
162.             } else {
163.                 er_add_slice(h, h->resync_mb_x, h->resync_mb_y,
164.                     h->mb_x, h->mb_y, ER_MB_END);
165.
166.                 return AVERROR_INVALIDDATA;
167.             }
168.         }
169.     }
170.
171.     if (get_bits_left(&h->gb) <= 0 && h->mb_skip_run <= 0) {
172.         tprintf(h->avctx, "slice end %d %d\n",
173.             get_bits_count(&h->gb), h->gb.size_in_bits);
174.
175.         if (get_bits_left(&h->gb) == 0) {
176.             er_add_slice(h, h->resync_mb_x, h->resync_mb_y,
177.                 h->mb_x - 1, h->mb_y, ER_MB_END);
178.             if (h->mb_x > lf_x_start)
179.                 loop_filter(h, lf_x_start, h->mb_x);
180.
181.             return 0;
182.         } else {
183.             er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x,
184.                 h->mb_y, ER_MB_ERROR);
185.
186.             return AVERROR_INVALIDDATA;
187.         }
188.     }
189. }
190. }
191. }

```

重复记录一下decode_slice()的流程：

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

(3) 如果是CABAC编码,直接进入一个循环,依次对每个宏块进行以下处理:

- 调用ff_h264_decode_mb_cavlc()进行CAVLC熵解码
- 调用ff_h264_hl_decode_mb()进行宏块解码
- 解码一行宏块之后调用loop_filter()进行环路滤波
- 此外还有可能调用er_add_slice()进行错误隐藏处理

可以看出,环路滤波函数是loop_filter()。下面看一下这个函数。

loop_filter()

loop_filter()完成了环路滤波工作。该函数的定义位于libavcodec/h264_slice.c,如下所示。

```
[cpp]
1. //环路滤波
2. static void loop_filter(H264Context *h, int start_x, int end_x)
3. {
4.     uint8_t *dest_y, *dest_cb, *dest_cr;
5.     int linesize, uvlinesize, mb_x, mb_y;
6.     const int end_mb_y = h->mb_y + FRAME_MBAFF(h);
7.     const int old_slice_type = h->slice_type;
8.     const int pixel_shift = h->pixel_shift;
9.     const int block_h = 16 >> h->chroma_y_shift;
10.
11.     if (h->deblocking_filter) {
12.         //循环处理宏块
13.         //例如从一行开始的mb_x到一行结束的mb_x
14.         for (mb_x = start_x; mb_x < end_x; mb_x++)
15.             for (mb_y = end_mb_y - FRAME_MBAFF(h); mb_y <= end_mb_y; mb_y++) { //逐行扫描只有一行
16.                 int mb_xy, mb_type;
17.                 mb_xy = h->mb_xy = mb_x + mb_y * h->mb_stride;
18.                 h->slice_num = h->slice_table[mb_xy];
19.                 mb_type = h->cur_pic.mb_type[mb_xy];
20.                 h->list_count = h->list_counts[mb_xy];
21.
22.                 if (FRAME_MBAFF(h))
23.                     h->mb_mbafl =
24.                     h->mb_field_decoding_flag = !!IS_INTERLACED(mb_type);
25.
26.                 h->mb_x = mb_x;
27.                 h->mb_y = mb_y;
28.                 //像素数据
29.                 dest_y = h->cur_pic.f.data[0] +
30.                     ((mb_x << pixel_shift) + mb_y * h->linesize) * 16;
31.                 dest_cb = h->cur_pic.f.data[1] +
32.                     (mb_x << pixel_shift) * (8 << CHROMA444(h)) +
33.                     mb_y * h->uvlinesize * block_h;
34.                 dest_cr = h->cur_pic.f.data[2] +
35.                     (mb_x << pixel_shift) * (8 << CHROMA444(h)) +
36.                     mb_y * h->uvlinesize * block_h;
37.                 // FIXME simplify above
38.
39.                 if (MB_FIELD(h)) {
40.                     linesize = h->mb_linesize = h->linesize * 2;
41.                     uvlinesize = h->mb_uvlinesize = h->uvlinesize * 2;
42.                     if (mb_y & 1) { // FIXME move out of this function?
43.                         dest_y -= h->linesize * 15;
44.                         dest_cb -= h->uvlinesize * (block_h - 1);
45.                         dest_cr -= h->uvlinesize * (block_h - 1);
46.                     }
47.                 } else {
48.                     linesize = h->mb_linesize = h->linesize;
49.                     uvlinesize = h->mb_uvlinesize = h->uvlinesize;
50.                 }
51.                 backup_mb_border(h, dest_y, dest_cb, dest_cr, linesize,
52.                     uvlinesize, 0);
53.                 if (fill_filter_caches(h, mb_type))
54.                     continue;
55.                 h->chroma_qp[0] = get_chroma_qp(h, 0, h->cur_pic.qscale_table[mb_xy]);
56.                 h->chroma_qp[1] = get_chroma_qp(h, 1, h->cur_pic.qscale_table[mb_xy]);
57.                 //宏块滤波器
58.                 if (FRAME_MBAFF(h)) {
59.                     //宏块级帧场自适应才用, 不研究
60.                     ff_h264_filter_mb(h, mb_x, mb_y, dest_y, dest_cb, dest_cr,
61.                         linesize, uvlinesize);
62.                 } else {
63.                     //宏块滤波器 (快速?)
64.                     ff_h264_filter_mb_fast(h, mb_x, mb_y, dest_y, dest_cb,
65.                         dest_cr, linesize, uvlinesize);
66.                 }
67.             }
68.     }
69.     h->slice_type = old_slice_type;
70.     h->mb_x = end_x;
71.     h->mb_y = end_mb_y - FRAME_MBAFF(h);
72.     h->chroma_qp[0] = get_chroma_qp(h, 0, h->qscale);
73.     h->chroma_qp[1] = get_chroma_qp(h, 1, h->qscale);
74. }
```

从源代码可以看出，`loop_filter()`循环遍历一行宏块，并且针对每一个宏块调用了`ff_h264_filter_mb_fast()`函数。

ff_h264_filter_mb_fast()

ff_h264_filter_mb_fast()用于对一个宏块进行环路滤波工作。该函数的定义位于libavcodec\h264_loopfilter.c，如下所示。

```

1. //宏块滤波器 (快速?)
2. void ff_h264_filter_mb_fast( H264Context *h, int mb_x, int mb_y, uint8_t *img_y, uint8_t *img_cb, uint8_t *img_cr, unsigned int line
size, unsigned int uvlinesize) {
3.     av_assert2(!FRAME_MBAFF(h));
4.     if(!h->h264dsp.h264_loop_filter_strength || h->pps.chroma_qp_diff) {
5.         ff_h264_filter_mb(h, mb_x, mb_y, img_y, img_cb, img_cr, linesize, uvlinesize);
6.         return;
7.     }
8.
9. #if CONFIG_SMALL
10.     h264_filter_mb_fast_internal(h, mb_x, mb_y, img_y, img_cb, img_cr, linesize, uvlinesize, h->pixel_shift);
11. #else
12.     //宏块滤波器-internal (快速?)
13.     if(h->pixel_shift){
14.         h264_filter_mb_fast_internal(h, mb_x, mb_y, img_y, img_cb, img_cr, linesize, uvlinesize, 1);
15.     }else{
16.         h264_filter_mb_fast_internal(h, mb_x, mb_y, img_y, img_cb, img_cr, linesize, uvlinesize, 0);
17.     }
18. #endif
19. }

```

可以看出ff_h264_filter_mb_fast()代码比较简单，其中调用了另一个函数h264_filter_mb_fast_internal()。

h264_filter_mb_fast_internal()

`h264_filter_mb_fast_internal()`用于对一个宏块进行环路滤波。该函数的定义位于`libavcodec/h264_loopfilter.c`，如下所示。

```

1. //宏块滤波器-internal (快速?)
2. static av_always_inline void h264_filter_mb_fast_internal(H264Context *h,
3.                                                         int mb_x, int mb_y,
4.                                                         uint8_t *img_y,
5.                                                         uint8_t *img_cb,
6.                                                         uint8_t *img_cr,
7.                                                         unsigned int linesize,
8.                                                         unsigned int uvlinesize,
9.                                                         int pixel_shift)
10. {
11.     int chroma = CHROMA(h) && !(CONFIG_GRAY && (h->flags&CODEC_FLAG_GRAY));
12.     int chroma444 = CHROMA444(h);
13.     int chroma422 = CHROMA422(h);
14.     //宏块序号
15.     int mb_xy = h->mb_xy;
16.     int left_type= h->left_type[LTOP];
17.     int top_type= h->top_type;
18.
19.     int qp_bd_offset = 6 * (h->sps.bit_depth_luma - 8);
20.     int a = 52 + h->slice_alpha_c0_offset - qp_bd_offset;
21.     int b = 52 + h->slice_beta_offset - qp_bd_offset;
22.     //宏块类型
23.     int mb_type = h->cur_pic.mb_type[mb_xy];
24.     //量化参数
25.     //qp用于推导alpha,beta (判断是否滤波的门限值)
26.     int qp      = h->cur_pic.qscale_table[mb_xy];
27.     int qp0     = h->cur_pic.qscale_table[mb_xy - 1];
28.     int qp1     = h->cur_pic.qscale_table[h->top_mb_xy];
29.     int qpc = get_chroma_qp( h, 0, qp );
30.     int qpc0 = get_chroma_qp( h, 0, qp0 );
31.     int qpc1 = get_chroma_qp( h, 0, qp1 );
32.     qp0 = (qp + qp0 + 1) >> 1;
33.     qp1 = (qp + qp1 + 1) >> 1;
34.     qpc0 = (qpc + qpc0 + 1) >> 1;
35.     qpc1 = (qpc + qpc1 + 1) >> 1;
36.     //Intra类型
37.     if( IS_INTRA(mb_type) ) {
38.         static const int16_t bS4[4] = {4,4,4,4};
39.         static const int16_t bS3[4] = {3,3,3,3};
40.         const int16_t *bSH = FIELD_PICTURE(h) ? bS3 : bS4;
41.         /*
42.          * 帧内宏块滤波
43.          * 滤波顺序如下所示 (大方框代表16x16块)
44.          *
45.          * +-+4+--+4+--+4+--+4+
46.          * 0  1  2  3  |
47.          * +-+5+--+5+--+5+--+5+

```

```

48.      * 0   1   2   3   |
49.      * --6--6--6--6--+
50.      * 0   1   2   3   |
51.      * --7--7--7--7--+
52.      * 0   1   2   3   |
53.      * +---+---+---+---+
54.      *
55.      */
56.  if(left_type)
57.      //宏块的左边界, 强度bs为4的滤波 (Vertical)
58.      filter_mb_edgev( &img_y[4*0<<pixel_shift], linesize, bS4, qp0, a, b, h, 1); //0
59.  //不考虑8x8DCT
60.  if( IS_8x8DCT(mb_type) ) {
61.      filter_mb_edgev( &img_y[4*2<<pixel_shift], linesize, bS3, qp, a, b, h, 0);
62.      if(top_type){
63.          filter_mb_edgeh( &img_y[4*0*linesize], linesize, bSH, qp1, a, b, h, 1);
64.      }
65.      filter_mb_edgeh( &img_y[4*2*linesize], linesize, bS3, qp, a, b, h, 0);
66.  } else {
67.      //宏块内部强度bs为3的滤波 (Vertical)
68.      filter_mb_edgev( &img_y[4*1<<pixel_shift], linesize, bS3, qp, a, b, h, 0); //1
69.      filter_mb_edgev( &img_y[4*2<<pixel_shift], linesize, bS3, qp, a, b, h, 0); //2
70.      filter_mb_edgev( &img_y[4*3<<pixel_shift], linesize, bS3, qp, a, b, h, 0); //3
71.      if(top_type){
72.          //宏块的上边界, 强度bs为4的滤波 (逐行扫描) (Horizontal)
73.          filter_mb_edgeh( &img_y[4*0*linesize], linesize, bSH, qp1, a, b, h, 1); //4
74.      }
75.      //宏块内部强度bs为3的滤波 (Horizontal)
76.      filter_mb_edgeh( &img_y[4*1*linesize], linesize, bS3, qp, a, b, h, 0); //5
77.      filter_mb_edgeh( &img_y[4*2*linesize], linesize, bS3, qp, a, b, h, 0); //6
78.      filter_mb_edgeh( &img_y[4*3*linesize], linesize, bS3, qp, a, b, h, 0); //7
79.  }
80.  if(chroma){
81.      if(chroma444){
82.          if(left_type){
83.              filter_mb_edgev( &img_cb[4*0<<pixel_shift], linesize, bS4, qp0, a, b, h, 1);
84.              filter_mb_edgev( &img_cr[4*0<<pixel_shift], linesize, bS4, qp0, a, b, h, 1);
85.          }
86.          if( IS_8x8DCT(mb_type) ) {
87.              filter_mb_edgev( &img_cb[4*2<<pixel_shift], linesize, bS3, qpc, a, b, h, 0);
88.              filter_mb_edgev( &img_cr[4*2<<pixel_shift], linesize, bS3, qpc, a, b, h, 0);
89.              if(top_type){
90.                  filter_mb_edgeh( &img_cb[4*0*linesize], linesize, bSH, qp1, a, b, h, 1 );
91.                  filter_mb_edgeh( &img_cr[4*0*linesize], linesize, bSH, qp1, a, b, h, 1 );
92.              }
93.              filter_mb_edgeh( &img_cb[4*2*linesize], linesize, bS3, qpc, a, b, h, 0);
94.              filter_mb_edgeh( &img_cr[4*2*linesize], linesize, bS3, qpc, a, b, h, 0);
95.          } else {
96.              filter_mb_edgev( &img_cb[4*1<<pixel_shift], linesize, bS3, qpc, a, b, h, 0);
97.              filter_mb_edgev( &img_cr[4*1<<pixel_shift], linesize, bS3, qpc, a, b, h, 0);
98.              filter_mb_edgev( &img_cb[4*2<<pixel_shift], linesize, bS3, qpc, a, b, h, 0);
99.              filter_mb_edgev( &img_cr[4*2<<pixel_shift], linesize, bS3, qpc, a, b, h, 0);
100.             filter_mb_edgev( &img_cb[4*3<<pixel_shift], linesize, bS3, qpc, a, b, h, 0);
101.             filter_mb_edgev( &img_cr[4*3<<pixel_shift], linesize, bS3, qpc, a, b, h, 0);
102.             if(top_type){
103.                 filter_mb_edgeh( &img_cb[4*0*linesize], linesize, bSH, qp1, a, b, h, 1);
104.                 filter_mb_edgeh( &img_cr[4*0*linesize], linesize, bSH, qp1, a, b, h, 1);
105.             }
106.             //水平horizontal
107.             filter_mb_edgeh( &img_cb[4*1*linesize], linesize, bS3, qpc, a, b, h, 0);
108.             filter_mb_edgeh( &img_cr[4*1*linesize], linesize, bS3, qpc, a, b, h, 0);
109.             filter_mb_edgeh( &img_cb[4*2*linesize], linesize, bS3, qpc, a, b, h, 0);
110.             filter_mb_edgeh( &img_cr[4*2*linesize], linesize, bS3, qpc, a, b, h, 0);
111.             filter_mb_edgeh( &img_cb[4*3*linesize], linesize, bS3, qpc, a, b, h, 0);
112.             filter_mb_edgeh( &img_cr[4*3*linesize], linesize, bS3, qpc, a, b, h, 0);
113.         }
114.     } else if(chroma422){
115.         if(left_type){
116.             filter_mb_edgcev(&img_cb[2*0<<pixel_shift], uvlinesize, bS4, qp0, a, b, h, 1);
117.             filter_mb_edgcev(&img_cr[2*0<<pixel_shift], uvlinesize, bS4, qp0, a, b, h, 1);
118.         }
119.         filter_mb_edgcev(&img_cb[2*2<<pixel_shift], uvlinesize, bS3, qpc, a, b, h, 0);
120.         filter_mb_edgcev(&img_cr[2*2<<pixel_shift], uvlinesize, bS3, qpc, a, b, h, 0);
121.         if(top_type){
122.             filter_mb_edgeh(&img_cb[4*0*uvlinesize], uvlinesize, bSH, qp1, a, b, h, 1);
123.             filter_mb_edgeh(&img_cr[4*0*uvlinesize], uvlinesize, bSH, qp1, a, b, h, 1);
124.         }
125.         filter_mb_edgeh(&img_cb[4*1*uvlinesize], uvlinesize, bS3, qpc, a, b, h, 0);
126.         filter_mb_edgeh(&img_cr[4*1*uvlinesize], uvlinesize, bS3, qpc, a, b, h, 0);
127.         filter_mb_edgeh(&img_cb[4*2*uvlinesize], uvlinesize, bS3, qpc, a, b, h, 0);
128.         filter_mb_edgeh(&img_cr[4*2*uvlinesize], uvlinesize, bS3, qpc, a, b, h, 0);
129.         filter_mb_edgeh(&img_cb[4*3*uvlinesize], uvlinesize, bS3, qpc, a, b, h, 0);
130.         filter_mb_edgeh(&img_cr[4*3*uvlinesize], uvlinesize, bS3, qpc, a, b, h, 0);
131.     } else{
132.         if(left_type){
133.             filter_mb_edgcev( &img_cb[2*0<<pixel_shift], uvlinesize, bS4, qp0, a, b, h, 1);
134.             filter_mb_edgcev( &img_cr[2*0<<pixel_shift], uvlinesize, bS4, qp0, a, b, h, 1);
135.         }
136.         filter_mb_edgcev( &img_cb[2*2<<pixel_shift], uvlinesize, bS3, qpc, a, b, h, 0);
137.         filter_mb_edgcev( &img_cr[2*2<<pixel_shift], uvlinesize, bS3, qpc, a, b, h, 0);
138.         if(top_type){

```

```

139.         filter_mb_edgch( &img_cb[2*0*uvlinesize], uvlinesize, bSH, qpcl, a, b, h, 1);
140.         filter_mb_edgch( &img_cr[2*0*uvlinesize], uvlinesize, bSH, qpcl, a, b, h, 1);
141.     }
142.     filter_mb_edgch( &img_cb[2*2*uvlinesize], uvlinesize, bS3, qpc, a, b, h, 0);
143.     filter_mb_edgch( &img_cr[2*2*uvlinesize], uvlinesize, bS3, qpc, a, b, h, 0);
144. }
145. }
146. return;
147. } else {
148.     //非Intra类型
149.     LOCAL_ALIGNED_8(int16_t, bS, [2], [4][4]);
150.     int edges;
151.     if( IS_8x8DCT(mb_type) && (h->cbp&7) == 7 && !chroma444 ) {
152.         edges = 4;
153.         AV_WN64A(bS[0][0], 0x0002000200020002ULL);
154.         AV_WN64A(bS[0][2], 0x0002000200020002ULL);
155.         AV_WN64A(bS[1][0], 0x0002000200020002ULL);
156.         AV_WN64A(bS[1][2], 0x0002000200020002ULL);
157.     } else {
158.         int mask_edge1 = (3*
159. ((5*mb_type)>>5)&1) | (mb_type>>4); // (mb_type & (MB_TYPE_16x16 | MB_TYPE_8x16)) ? 3 : (mb_type & MB_TYPE_16x8) ? 1 : 0;
160.         int mask_edge0 = 3*((mask_edge1>>1) & ((5*left_type)>>5)&1); // (mb_type & (MB_TYPE_16x16 | MB_TYPE_8x16)) && (h->left_t
161. ype[LTOP] & (MB_TYPE_16x16 | MB_TYPE_8x16)) ? 3 : 0;
162.         int step = 1+(mb_type>>24); // IS_8x8DCT(mb_type) ? 2 : 1;
163.         edges = 4 - 3*((mb_type>>3) & !(h->cbp & 15)); // (mb_type & MB_TYPE_16x16) && !(h->cbp & 15) ? 1 : 4;
164.         h->h264dsp.h264_loop_filter_strength( bS, h->non_zero_count_cache, h->ref_cache, h->mv_cache,
165. h->list_count==2, edges, step, mask_edge0, mask_edge1, FIELD_PICTURE(h));
166.     }
167.     if( IS_INTRA(left_type) )
168.         AV_WN64A(bS[0][0], 0x0004000400040004ULL);
169.     if( IS_INTRA(top_type) )
170.         AV_WN64A(bS[1][0], FIELD_PICTURE(h) ? 0x0003000300030003ULL : 0x0004000400040004ULL);
171.     //专门定义了一个宏?
172. #define FILTER(hv,dir,edge,intra)\
173.     if(AV_RN64A(bS[dir][edge])) {\
174.         filter_mb_edgch##hv( &img_y[4*edge*(dir?linesize:1<<pixel_shift)], linesize, bS[dir]
175. [edge], edge ? qp : qp##dir, a, b, h, intra );\
176.         if(chroma){\
177.             if(chroma444){\
178.                 filter_mb_edgch##hv( &img_cb[4*edge*(dir?linesize:1<<pixel_shift)], linesize, bS[dir]
179. [edge], edge ? qpc : qpc##dir, a, b, h, intra );\
180.                 filter_mb_edgch##hv( &img_cr[4*edge*(dir?linesize:1<<pixel_shift)], linesize, bS[dir]
181. [edge], edge ? qpc : qpc##dir, a, b, h, intra );\
182.             } else if(!(edge&1)) {\
183.                 filter_mb_edgch##hv( &img_cb[2*edge*(dir?uvlinesize:1<<pixel_shift)], uvlinesize, bS[dir]
184. [edge], edge ? qpc : qpc##dir, a, b, h, intra );\
185.                 filter_mb_edgch##hv( &img_cr[2*edge*(dir?uvlinesize:1<<pixel_shift)], uvlinesize, bS[dir]
186. [edge], edge ? qpc : qpc##dir, a, b, h, intra );\
187.             }\
188.         }\
189.     }\
190. }
191. /*
192. * 非Intra宏块滤波
193. * 滤波顺序如下所示 (大方框代表16x16块)
194. *
195. * +-+4+---4+---4+---4+
196. * 0 1 2 3 |
197. * +-+5+---5+---5+---5+
198. * 0 1 2 3 |
199. * +-+6+---6+---6+---6+
200. * 0 1 2 3 |
201. * +-+7+---7+---7+---7+
202. * 0 1 2 3 |
203. * +---+-----+
204. *
205. */
206. if(left_type)
207.     FILTER(v,0,0,1); //0
208. if( edges == 1 ) {
209.     if(top_type)
210.         FILTER(h,1,0,1);
211. } else if( IS_8x8DCT(mb_type) ) {
212.     FILTER(v,0,2,0);
213.     if(top_type)
214.         FILTER(h,1,0,1);
215.     FILTER(h,1,2,0);
216. } else {
217.     FILTER(v,0,1,0); //1
218.     FILTER(v,0,2,0); //2
219.     FILTER(v,0,3,0); //3
220.     if(top_type)
221.         FILTER(h,1,0,1); //4
222.     FILTER(h,1,1,0); //5
223.     FILTER(h,1,2,0); //6
224.     FILTER(h,1,3,0); //7
225. }
226. #undef FILTER
227. }
228. }

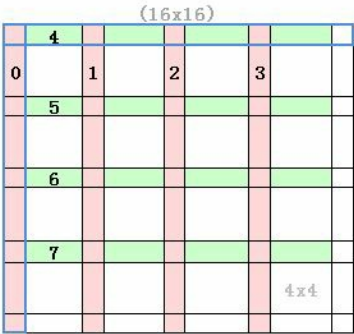
```


通过源代码整理出来h264_filter_mb_fast_internal()的流程如下：

- (1) 读取QP等几个参数，用于推导滤波门限值alpha, beta。
- (2) 如果是帧内宏块（Intra），作如下处理：
 - a)对于水平的边界，调用filter_mb_edghev()进行滤波。
 - b)对于垂直的边界，调用filter_mb_edghev()进行滤波。帧内宏块滤波过程中，对于在宏块边界上的边界（最左边的垂直边界和最上边的水平边界），采用滤波强度Bs为4的滤波；对于其它边界则采用滤波强度Bs为3的滤波。
- (3) 如果是其他宏块，作如下处理：
 - a)对于水平的边界，调用filter_mb_edghev()进行滤波。
 - b)对于垂直的边界，调用filter_mb_edghev()进行滤波。此类宏块的滤波强度需要另作判断。

总体说来，一个宏块内部的滤波顺序如下图所示。图中的“0”、“1”、“2”、“3”为滤波的顺序。可以看出首先对垂直边界进行滤波，然后对水平边界进行滤波。垂直边界滤波按照从左到右的顺序进行，而水平边界的滤波按照从上到下的顺序进行。

Loop Filter in Macroblock



“1”, “2”... represent order of filter

FFmpeg Source Analysis: H.264 Decoder
Loop Filter in Macroblock
雷霄骅 (Lei Xiaohua)
leixiaohua1020@126.com
<http://blog.csdn.net/leixiaohua1020>

下面分别看一下对水平边界滤波的函数filter_mb_edghev()以及对垂直边界滤波的函数filter_mb_edghev()。

filter_mb_edghev()

filter_mb_edghev()用于对水平边界进行滤波。该函数定义位于libavcodec\h264_loopfilter.c，如下所示。

```

1. //滤波水平边界 (Horizontal) -亮度
2. //垂直 (Vertical) 滤波器
3. //    边界
4. //    x
5. //    x
6. // 边界-----
7. //    x
8. //    x
9. static av_always_inline void filter_mb_edgch(uint8_t *pix, int stride,
10.                                              const int16_t bs[4],
11.                                              unsigned int qp, int a, int b,
12.                                              H264Context *h, int intra)
13. {
14.     //alpha,beta为判断是否滤波的门限值
15.     //它们是通过将 (qp+offset)作为索引查表得到的
16.     //qp大 (压缩大), 门限高, 更容易发生滤波
17.     const unsigned int index_a = qp + a;
18.     const int alpha = alpha_table[index_a];
19.     const int beta = beta_table[qp + b];
20.
21.     //门限为0, 不用滤波了
22.     if (alpha == 0 || beta == 0) return;
23.
24.     if( bs[0] < 4 || !intra ) {
25.         int8_t tc[4];
26.         tc[0] = tc0_table[index_a][bs[0]];
27.         tc[1] = tc0_table[index_a][bs[1]];
28.         tc[2] = tc0_table[index_a][bs[2]];
29.         tc[3] = tc0_table[index_a][bs[3]];
30.         //边界强度3以下 (弱滤波)
31.         h->h264dsp.h264_v_loop_filter_luma(pix, stride, alpha, beta, tc);
32.     } else {
33.         //边界强度为4个滤波 (强滤波)
34.         h->h264dsp.h264_v_loop_filter_luma_intra(pix, stride, alpha, beta);
35.     }
36. }

```

从filter_mb_edgch()的定义可以看出, 该函数首先计算了alpha,beta两个滤波的门限值, 然后根据输入信息判断是否需要强滤波。如果需要强滤波 (Bs取值为4), 就调用H264DSPContext中的滤波汇编函数h264_v_loop_filter_luma_intra(); 如果不需要强滤波 (Bs取值为1、2、3), 就调用H264DSPContext中的滤波汇编函数h264_v_loop_filter_luma()。

在这里有一点需要注意, 对水平边界进行滤波的函数 (函数名中包含“_edgch”), 调用的是垂直滤波函数 (函数名中包含“_v”)。

filter_mb_edgev()

filter_mb_edgev()用于对垂直边界进行滤波。该函数定义位于libavcodec/h264_loopfilter.c, 如下所示。

```

1. //滤波垂直边界 (Vertical) -亮度
2. //水平 (Horizontal) 滤波器
3. //    边界
4. //    |
5. // x x x | x x x
6. //    |
7. static av_always_inline void filter_mb_edgev(uint8_t *pix, int stride,
8.                                              const int16_t bs[4],
9.                                              unsigned int qp, int a, int b,
10.                                              H264Context *h, int intra)
11. {
12.     const unsigned int index_a = qp + a;
13.     const int alpha = alpha_table[index_a];
14.     const int beta = beta_table[qp + b];
15.     if (alpha == 0 || beta == 0) return;
16.
17.     if( bs[0] < 4 || !intra ) {
18.         int8_t tc[4];
19.         tc[0] = tc0_table[index_a][bs[0]];
20.         tc[1] = tc0_table[index_a][bs[1]];
21.         tc[2] = tc0_table[index_a][bs[2]];
22.         tc[3] = tc0_table[index_a][bs[3]];
23.         //Bs取值为1,2,3的弱滤波
24.         h->h264dsp.h264_h_loop_filter_luma(pix, stride, alpha, beta, tc);
25.     } else {
26.         //Bs取值为4的强滤波
27.         h->h264dsp.h264_h_loop_filter_luma_intra(pix, stride, alpha, beta);
28.     }
29. }

```

可以看出filter_mb_edgev()的定义与filter_mb_edgch()是类似的。也是先计算了alpha,beta两个滤波的门限值, 然后根据输入信息判断是否需要强滤波。如果需要强滤波 (Bs取值为4), 就调用H264DSPContext中的滤波汇编函数h264_h_loop_filter_luma_intra(); 如果不需要强滤波 (Bs取值为1、2、3), 就调用H264DSPContext中的滤波汇编函数h264_h_loop_filter_luma()。下文将会对H264DSPContext中的h264_h_loop_filter_luma()和h264_h_loop_filter_luma_intra()这两个汇编函数进行分析。

环路滤波小知识

H.264解码器在解码后的数据一般情况下会出现方块效应。产生这种效应的原因主要有两个：

- (1) DCT变换后的量化造成误差（主要原因）。
- (2) 运动补偿

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



环路滤波分类

环路滤波器根据滤波的强度可以分为两种：

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

$$p0' = p0 + (((q0 - p0) << 2) + (p1 - q1) + 4) >> 3$$
$$p1' = (p2 + ((p0 + q0 + 1) >> 1) - 2p1) >> 1$$

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

$$p0' = (p2 + 2*p1 + 2*p0 + 2*q0 + q1 + 4) >> 3$$
$$p1' = (p2 + p1 + p0 + q0 + 2) >> 2$$
$$p2' = (2*p3 + 3*p2 + p1 + p0 + q0 + 4) >> 3$$

其中上文中提到的边界强度Bs的判定方式如下。

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

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

环路滤波的门限

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

$$|p0 - q0| < \alpha$$
$$|p1 - p0| < \beta$$
$$|q1 - q0| < \beta$$

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

有关环路滤波的基本知识就记录到这里，下文开始分析和环路滤波相关的汇编函数的源代码。

环路滤波汇编函数

首先看一下环路滤波汇编函数的初始化函数ff_h264dsp_init()。

ff_h264dsp_init()

ff_h264dsp_init()用于初始化环路滤波函数（实际上该函数也用于初始化DCT反变换和Hadamard反变换函数）。该函数的定义位于libavcodec/h264dsp.c，如下所示。

```
[cpp]  
1. //初始化DSP相关的函数。包含了IDCT、环路滤波函数等
2. av_cold void ff_h264dsp_init(H264DSPContext *c, const int bit_depth,
3.                               const int chroma_format_idc)
4. {
5.     #undef FUNC
6.     #define FUNC(a, depth) a ## _ ## depth ## _c
7.
8.     #define ADDPX_DSP(depth) \
9.         c->h264_add_pixels4_clear = FUNC(ff_h264_add_pixels4, depth);\
10.        c->h264_add_pixels8_clear = FUNC(ff_h264_add_pixels8, depth)
11.
12.        if (bit_depth > 8 && bit_depth <= 16) {
13.            ADDPX_DSP(16);
14.        } else {
15.            ADDPX_DSP(8);
16.        }
17.
18.        #define H264_DSP(depth) \
19.            c->h264_idct_add= FUNC(ff_h264_idct_add, depth);\
20.            c->h264_idct8_add= FUNC(ff_h264_idct8_add, depth);\
21.            c->h264_idct_dc_add= FUNC(ff_h264_idct_dc_add, depth);\
22.            c->h264_idct8_dc_add= FUNC(ff_h264_idct8_dc_add, depth);\
23.            c->h264_idct_add16 = FUNC(ff_h264_idct_add16, depth);\
24.            c->h264_idct8_add4 = FUNC(ff_h264_idct8_add4, depth);\
25.            if (chroma_format_idc <= 1)\
26.                c->h264_idct_add8 = FUNC(ff_h264_idct_add8, depth);\
27.            else\
28.                c->h264_idct_add8 = FUNC(ff_h264_idct_add8_422, depth);\
29.            c->h264_idct_add16intra= FUNC(ff_h264_idct_add16intra, depth);\
30.            c->h264_luma_dc_dequant_idct= FUNC(ff_h264_luma_dc_dequant_idct, depth);\
31.            if (chroma_format_idc <= 1)\
32.                c->h264_chroma_dc_dequant_idct= FUNC(ff_h264_chroma_dc_dequant_idct, depth);\
33.            else\
34.                c->h264_chroma_dc_dequant_idct= FUNC(ff_h264_chroma422_dc_dequant_idct, depth);\
35.        \
36.        c->weight_h264_pixels_tab[0]= FUNC(weight_h264_pixels16, depth);\
37.        c->weight_h264_pixels_tab[1]= FUNC(weight_h264_pixels8, depth);\
38.        c->weight_h264_pixels_tab[2]= FUNC(weight_h264_pixels4, depth);\
39.        c->weight_h264_pixels_tab[3]= FUNC(weight_h264_pixels2, depth);\
40.        c->biweight_h264_pixels_tab[0]= FUNC(biweight_h264_pixels16, depth);\
41.        c->biweight_h264_pixels_tab[1]= FUNC(biweight_h264_pixels8, depth);\
42.        c->biweight_h264_pixels_tab[2]= FUNC(biweight_h264_pixels4, depth);\
43.        c->biweight_h264_pixels_tab[3]= FUNC(biweight_h264_pixels2, depth);\
44.        \
45.        c->h264_v_loop_filter_luma= FUNC(h264_v_loop_filter_luma, depth);\
46.        c->h264_h_loop_filter_luma= FUNC(h264_h_loop_filter_luma, depth);\
47.        c->h264_h_loop_filter_luma_mbaff= FUNC(h264_h_loop_filter_luma_mbaff, depth);\
48.        c->h264_v_loop_filter_luma_intra= FUNC(h264_v_loop_filter_luma_intra, depth);\
49.        c->h264_h_loop_filter_luma_intra= FUNC(h264_h_loop_filter_luma_intra, depth);\
50.        c->h264_h_loop_filter_luma_mbaff_intra= FUNC(h264_h_loop_filter_luma_mbaff_intra, depth);\
51.        c->h264_v_loop_filter_chroma= FUNC(h264_v_loop_filter_chroma, depth);\
52.        if (chroma_format_idc <= 1)\
53.            c->h264_h_loop_filter_chroma= FUNC(h264_h_loop_filter_chroma, depth);\
54.        else\
55.            c->h264_h_loop_filter_chroma= FUNC(h264_h_loop_filter_chroma422, depth);\
56.        if (chroma_format_idc <= 1)\
57.            c->h264_h_loop_filter_chroma_mbaff= FUNC(h264_h_loop_filter_chroma_mbaff, depth);\
58.        else\
59.            c->h264_h_loop_filter_chroma_mbaff= FUNC(h264_h_loop_filter_chroma422_mbaff, depth);\
60.        c->h264_v_loop_filter_chroma_intra= FUNC(h264_v_loop_filter_chroma_intra, depth);\
61.        if (chroma_format_idc <= 1)\
62.            c->h264_h_loop_filter_chroma_intra= FUNC(h264_h_loop_filter_chroma_intra, depth);\
63.        else\
64.            c->h264_h_loop_filter_chroma_intra= FUNC(h264_h_loop_filter_chroma422_intra, depth);\
65.        if (chroma_format_idc <= 1)\
66.            c->h264_h_loop_filter_chroma_mbaff_intra= FUNC(h264_h_loop_filter_chroma_mbaff_intra, depth);\
67.        else\
68.            c->h264_h_loop_filter_chroma_mbaff_intra= FUNC(h264_h_loop_filter_chroma422_mbaff_intra, depth);\
69.        c->h264_loop_filter_strength= NULL;
70.        //根据颜色位深，初始化不同的函数
71.        //一般为8bit，即执行H264_DSP(8)
72.        switch (bit_depth) {
73.            case 9:
74.                H264_DSP(9);
75.                break;
```

```

76.     case 10:
77.         H264_DSP(10);
78.         break;
79.     case 12:
80.         H264_DSP(12);
81.         break;
82.     case 14:
83.         H264_DSP(14);
84.         break;
85.     default:
86.         av_assert0(bit_depth<=8);
87.         H264_DSP(8);
88.         break;
89. }
90. //这个函数查找startcode的时候用到
91. //在这里竟然单独列出
92. c->startcode_find_candidate = ff_startcode_find_candidate_c;
93. //如果系统支持，则初始化经过汇编优化的函数
94. if (ARCH_AARCH64) ff_h264dsp_init_aarch64(c, bit_depth, chroma_format_idc);
95. if (ARCH_ARM) ff_h264dsp_init_arm(c, bit_depth, chroma_format_idc);
96. if (ARCH_PPC) ff_h264dsp_init_ppc(c, bit_depth, chroma_format_idc);
97. if (ARCH_X86) ff_h264dsp_init_x86(c, bit_depth, chroma_format_idc);
98. }

```

从源代码可以看出，ff_h264dsp_init()初始化了环路滤波函数，DCT反变换函数和Hadamard反变换函数。下面展开“H264_DSP(8)”宏看一下C语言版本函数初始化的代码。

```

1.  c->h264_idct_add= ff_h264_idct_add_8_c;
2.  c->h264_idct8_add= ff_h264_idct8_add_8_c;
3.  c->h264_idct_dc_add= ff_h264_idct_dc_add_8_c;
4.  c->h264_idct8_dc_add= ff_h264_idct8_dc_add_8_c;
5.  c->h264_idct_add16 = ff_h264_idct_add16_8_c;
6.  c->h264_idct8_add4 = ff_h264_idct8_add4_8_c;
7.  if (chroma_format_idc <= 1)
8.      c->h264_idct_add8 = ff_h264_idct_add8_8_c;
9.  else
10.     c->h264_idct_add8 = ff_h264_idct_add8_422_8_c;
11. c->h264_idct_add16intra= ff_h264_idct_add16intra_8_c;
12. c->h264_luma_dc_dequant_idct= ff_h264_luma_dc_dequant_idct_8_c;
13. if (chroma_format_idc <= 1)
14.     c->h264_chroma_dc_dequant_idct= ff_h264_chroma_dc_dequant_idct_8_c;
15. else
16.     c->h264_chroma_dc_dequant_idct= ff_h264_chroma422_dc_dequant_idct_8_c;
17.
18. c->weight_h264_pixels_tab[0]= weight_h264_pixels16_8_c;
19. c->weight_h264_pixels_tab[1]= weight_h264_pixels8_8_c;
20. c->weight_h264_pixels_tab[2]= weight_h264_pixels4_8_c;
21. c->weight_h264_pixels_tab[3]= weight_h264_pixels2_8_c;
22. c->biweight_h264_pixels_tab[0]= biweight_h264_pixels16_8_c;
23. c->biweight_h264_pixels_tab[1]= biweight_h264_pixels8_8_c;
24. c->biweight_h264_pixels_tab[2]= biweight_h264_pixels4_8_c;
25. c->biweight_h264_pixels_tab[3]= biweight_h264_pixels2_8_c;
26.
27. c->h264_v_loop_filter_luma= h264_v_loop_filter_luma_8_c;
28. c->h264_h_loop_filter_luma= h264_h_loop_filter_luma_8_c;
29. c->h264_h_loop_filter_luma_mbaff= h264_h_loop_filter_luma_mbaff_8_c;
30. c->h264_v_loop_filter_luma_intra= h264_v_loop_filter_luma_intra_8_c;
31. c->h264_h_loop_filter_luma_intra= h264_h_loop_filter_luma_intra_8_c;
32. c->h264_h_loop_filter_luma_mbaff_intra=
33. h264_h_loop_filter_luma_mbaff_intra_8_c;
34. c->h264_v_loop_filter_chroma= h264_v_loop_filter_chroma_8_c;
35. if (chroma_format_idc <= 1)
36.     c->h264_h_loop_filter_chroma= h264_h_loop_filter_chroma_8_c;
37. else
38.     c->h264_h_loop_filter_chroma= h264_h_loop_filter_chroma422_8_c;
39. if (chroma_format_idc <= 1)
40.     c->h264_h_loop_filter_chroma_mbaff=
41. h264_h_loop_filter_chroma_mbaff_8_c;
42. else
43.     c->h264_h_loop_filter_chroma_mbaff=
44. h264_h_loop_filter_chroma422_mbaff_8_c;
45. c->h264_v_loop_filter_chroma_intra= h264_v_loop_filter_chroma_intra_8_c;
46. if (chroma_format_idc <= 1)
47.     c->h264_h_loop_filter_chroma_intra=
48. h264_h_loop_filter_chroma_intra_8_c;
49. else
50.     c->h264_h_loop_filter_chroma_intra=
51. h264_h_loop_filter_chroma422_intra_8_c;
52. if (chroma_format_idc <= 1)
53.     c->h264_h_loop_filter_chroma_mbaff_intra=
54. h264_h_loop_filter_chroma_mbaff_intra_8_c;
55. else
56.     c->h264_h_loop_filter_chroma_mbaff_intra=
57. h264_h_loop_filter_chroma422_mbaff_intra_8_c;
58. c->h264_loop_filter_strength= ((void *)0);

```

从“H264_DSP(8)”宏展开的结果可以看出，和亮度环路滤波有关的C语言函数有如下4个：

h264_v_loop_filter_luma_8_c()：亮度垂直的普通滤波。
h264_h_loop_filter_luma_8_c()：亮度水平的普通滤波。
h264_v_loop_filter_luma_intra_8_c()：亮度垂直的强滤波。
h264_h_loop_filter_luma_intra_8_c()：亮度水平的强滤波。

下面分别分析这4个函数的源代码。

h264_v_loop_filter_luma_8_c()

h264_v_loop_filter_luma_8_c()实现了亮度边界垂直普通滤波器（处理水平边界）。该函数的定义位于libavcodec/h264dsp_template.c，如下所示。

```
[cpp]
1. //垂直 (Vertical) 普通滤波器
2. //    边界
3. //        x
4. //        x
5. // 边界-----
6. //        x
7. //        x
8. static void h264_v_loop_filter_luma_8_c(uint8_t *pix, int stride, int alpha, int beta, int8_t *tc0)
9. {
10.     //xstride=stride (用于选择滤波的像素)
11.     //ystride=1
12.     //inner_iters=4
13.     h264_loop_filter_luma_8_c(pix, stride, sizeof(pixel), 4, alpha, beta, tc0);
14. }
```

从源代码中可以看出，h264_v_loop_filter_luma_8_c()调用了另一个函数h264_loop_filter_luma_8_c()。需要注意在调用h264_loop_filter_luma_8_c()的时候传递的3个主要的参数：

```
[plain]
1. xstride=stride
2. ystride=1
3. inner_iters=4
```

这几个参数中的xstride，ystride决定了滤波器的方向。下面看一下垂直和水平方向通用的普通滤波函数h264_loop_filter_luma_8_c()的定义。

h264_loop_filter_luma_8_c()

h264_loop_filter_luma_8_c()用于垂直或者水平滤波的普通滤波器（Bs取值为1、2、3）函数。该函数的定义位于libavcodec/h264dsp_template.c。原函数中包含了一些宏定义，宏定义展开后的结果如下所示。

```
[cpp]
1. //-----
2. //代码中函数名包含大量的“FUNC”的宏，该宏的定义如下所示
3. //“FUNC(XXX)”展开后的结果为“XXX_8_c”，即在“XXX”后面加上“_8_c”
4. //下面的代码中为了阅读方便，手动展开了一些重要函数的“FUNC”宏。
5. //但是手动展开宏比较麻烦，所以还是有一些“FUNC”宏没有展开
6. /*
7.  * 环路滤波函数 (Loop Filter) 展开结果
8.  *
9.  * 源代码注释和处理：雷霄骅
10.  * leixiaohua1020@126.com
11.  * http://blog.csdn.net/leixiaohua1020
12.  */
13.
14. //亮度的环路滤波器-普通滤波器
15. //边界强度Bs取1,2,3
16. //参数：
17. //p_pix：像素数据
18. //xstride, ystride：决定了是横向边界滤波器还是纵向边界滤波器
19. //inner_iters：逐行扫描为4
20. //alpha, beta：决定滤波器是否滤波的门限值，由QP确定。QP大，门限会高一些，更有可能滤波。
21. //tc0：限幅值，由QP确定。QP大，限幅值会高一些，相对宽松。此外边界强度Bs大，限幅值也会大。
22.
23. //普通滤波涉及到方块边界周围的6个点（边界两边各3个点）：p2, p1, p0, q0, q1, q2。
24. static av_always_inline av_flatten void h264_loop_filter_luma_8_c(uint8_t *p_pix, int xstride, int ystride, int inner_iters, int alpha, int beta, int8_t *tc0)
25. {
26.     //pixel代表了一个像素，在这里是uint8_t，定义如下所示
27.     // #define pixel uint8_t
28.
29.     pixel *pix = (pixel*)p_pix;
30.     int i, d;
31.     //不右移
32.     xstride >>= sizeof(pixel)-1;
```

```

33.     ystride >= sizeof(pixel)-1;
34.     //BIT_DEPTH在这里取值为8, 定义如下所示
35.     //define BIT_DEPTH 8
36.     alpha <= BIT_DEPTH - 8;
37.     beta <= BIT_DEPTH - 8;
38.
39.     //循环一共4x4=16次, 相当于处理了16个点, 与宏块的宽度是相同的
40.     /*
41.      * [滤波示例] 大方框代表一个宏块
42.      *
43.      * xstride=1, ystride=stride
44.      *
45.      * +---+---+---+---+
46.      * |   X   |   |   |
47.      * +---+---+---+---+
48.      * |   X   |   |   |
49.      * +---+---+---+---+
50.      * |   X   |   |   |
51.      * +---+---+---+---+
52.      * |   X   |   |   |
53.      * +---+---+---+---+
54.      *
55.      * xstride=stride, ystride=1
56.      *
57.      * +---+---+---+---+
58.      * |   |   |   |   |
59.      * +--X--+X--+X--+X--+
60.      * |   |   |   |   |
61.      * +---+---+---+---+
62.      * |   |   |   |   |
63.      * +---+---+---+---+
64.      * |   |   |   |   |
65.      * +---+---+---+---+
66.      */
67.     //外部循环4次
68.     for( i = 0; i < 4; i++ ) {
69.         const int tc_orig = tc0[i] << (BIT_DEPTH - 8);
70.         if( tc_orig < 0 ) {
71.             pix += inner_iters*ystride;
72.             continue;
73.         }
74.
75.         //一般inner_iters=4
76.         for( d = 0; d < inner_iters; d++ ) {
77.             //p和q
78.             //如果xstride=stride, ystride=1
79.             //就是处理纵向的6个像素
80.             //对应的是方块的横向边界的滤波 (后文以此举例子)。如下所示:
81.             //      p2
82.             //      p1
83.             //      p0
84.             //====图像边界====
85.             //      q0
86.             //      q1
87.             //      q2
88.             //
89.             //如果xstride=1, ystride=stride
90.             //就是处理纵向的6个像素
91.             //对应的是方块的横向边界的滤波, 即如下所示:
92.             //      ||
93.             // p2 p1 p0 || q0 q1 q2
94.             //      ||
95.             //      边界
96.
97.             //注意: 这里乘的是xstride
98.             const int p0 = pix[-1*xstride];
99.             const int p1 = pix[-2*xstride];
100.            const int p2 = pix[-3*xstride];
101.            const int q0 = pix[0];
102.            const int q1 = pix[1*xstride];
103.            const int q2 = pix[2*xstride];
104.            //计算方法参考相关的标准
105.            //alpha和beta是用于检查图像内容的2个参数
106.            //只有满足if()里面3个取值条件的时候 (只涉及边界旁边的4个点), 才会滤波
107.            if( FFABS( p0 - q0 ) < alpha &&
108.                FFABS( p1 - p0 ) < beta &&
109.                FFABS( q1 - q0 ) < beta ) {
110.
111.                int tc = tc_orig;
112.                int i_delta;
113.                //上面2个点 (p0, p2) 满足条件的时候, 滤波p1
114.                if( FFABS( p2 - p0 ) < beta ) {
115.                    //av_clip(int a, int amin, int amax)用于限幅: Clip a signed integer value into the amin-amax range.
116.                    if(tc_orig)
117.                        pix[-2*xstride] = p1 + av_clip( (( p2 + ( ( p0 + q0 + 1 ) >> 1 ) ) >> 1) - p1, -tc_orig, tc_orig );
118.                    tc++;
119.                }
120.                //下面2个点 (q0, q2) 满足条件的时候, 滤波q1
121.                if( FFABS( q2 - q0 ) < beta ) {
122.                    //q1
123.                    if(tc_orig)
124.                        pix[-xstride] = q1 + av_clip( (( q2 + ( ( q0 + q0 + 1 ) >> 1 ) >> 1) - q1, -tc_orig, tc_orig );

```

```

124.         pix[ -xstride] = q1 + av_clip( ( ( q2 + ( ( p0 + q0 + 1 ) >> 1 ) >> 1 ) - q1, -tc, tc );
125.         tc++;
126.     }
127.
128.     i_delta = av_clip( (((q0 - p0 ) << 2) + (p1 - q1) + 4) >> 3, -tc, tc );
129.     //p0
130.     pix[-xstride] = av_clip_pixel( p0 + i_delta );    /* p0' */
131.     //q0
132.     pix[0]      = av_clip_pixel( q0 - i_delta );    /* q0' */
133. }
134. //移动指针
135. //注意：这里加的是ystride
136. pix += ystride;
137. }
138. }
139. }

```

由于源代码中写了比较充分的注释，在这里就不再逐行解析代码了。可以看出函数中包含了两个嵌套的for()循环，每个for()循环循环4次，合计运行16次。for()循环执行一遍即完成了一次水平（或者垂直）的滤波，所以for()循环执行完毕的时候，就完成了对宏块中一个纵向边界（或者横向边界）的滤波。

函数的输入参数xstride和ystride决定了函数是水平滤波器还是垂直滤波器。如果xstride=stride、ystride=1，滤波器处理垂直的6个像素，为垂直滤波器；xstride=1、ystride=stride，滤波器处理水平的6个像素，为水平滤波器。

函数在确定了处理的6个点之后，就会根据滤波的门限值alpha和beta判定边界是否满足滤波条件。如果满足条件，就会根据下面的公式进行滤波（只列出p点的，q点类似）：

$$p0' = p0 + (((q0 - p0) << 2) + (p1 - q1) + 4) >> 3$$

$$p1' = (p2 + ((p0 + q0 + 1) >> 1) - 2p1) >> 1$$

h264_h_loop_filter_luma_8_c()

h264_h_loop_filter_luma_8_c()实现了亮度边界水平普通滤波器（处理垂直边界）。该函数的定义位于libavcodec/h264dsp_template.c，如下所示。

```

1. //水平 (Horizontal) 普通滤波器
2. // 边界
3. // |
4. // x x x | x x x
5. // |
6. static void h264_h_loop_filter_luma_8_c(uint8_t *pix, int stride, int alpha, int beta, int8_t *tc0)
7. {
8.     //xstride=1 (用于选择滤波的像素)
9.     //ystride=stride
10.    //inner_iters=4
11.    h264_loop_filter_luma_8_c(pix, sizeof(pixel), stride, 4, alpha, beta, tc0);
12. }

```

从源代码中可以看出，h264_h_loop_filter_luma_8_c()和h264_v_loop_filter_luma_8_c()类似，也调用了h264_loop_filter_luma_8_c()。需要注意在调用h264_loop_filter_luma_8_c()的时候传递的3个主要的参数：

```

1. xstride=1
2. ystride=stride
3. inner_iters=4

```

h264_v_loop_filter_luma_intra_8_c()

h264_v_loop_filter_luma_intra_8_c()实现了亮度边界垂直强滤波器（处理水平边界）。该函数的定义位于libavcodec/h264dsp_template.c，如下所示。

```

1. //垂直 (Vertical) 强滤波器
2. // 边界
3. // x
4. // x
5. // 边界-----
6. // x
7. // x
8. static void h264_v_loop_filter_luma_intra_8_c(uint8_t *pix, int stride, int alpha, int beta)
9. {
10.    //xstride=stride
11.    //ystride=1
12.    //inner_iters=4
13.    h264_loop_filter_luma_intra_8_c(pix, stride, sizeof(pixel), 4, alpha, beta);
14. }

```

可以看出h264_v_loop_filter_luma_intra_8_c()调用了水平垂直通用的强滤波器函数h264_loop_filter_luma_intra_8_c()。并传递了以下参数：

[plain]  

```
1. xstride=stride
2. ystride=1
3. inner_iters=4
```

h264_loop_filter_luma_intra_8_c()

h264_loop_filter_luma_intra_8_c()是用于垂直或者水平滤波的强滤波器（Bs取值为4）函数。该函数的定义位于libavcodec/h264dsp_template.c，定义如下所示。

[cpp]  

```
1. //亮度的环路滤波器-强滤波器
2. //边界强度Bs取4（最强）
3. //强滤波涉及到方块边界周围的8个点（边界两边各4个点）：p3, p2, p1, p0, q0, q1, q2, q3
4. static av_always_inline av_flatten void h264_loop_filter_luma_intra_8_c(uint8_t *p_pix, int xstride, int ystride, int inner_iters, int alpha, int beta)
5. {
6.     pixel *pix = (pixel*)p_pix;
7.     int d;
8.     xstride >>= sizeof(pixel)-1;
9.     ystride >>= sizeof(pixel)-1;
10.    alpha <=<= BIT_DEPTH - 8;
11.    beta <=<= BIT_DEPTH - 8;
12.
13.    //循环一共16次，相当于处理了16个点，与宏块的宽度是相同的
14.    /*
15.     * [滤波示例] 大方框代表一个宏块
16.     *
17.     * xstride=1, ystride=stride
18.     *
19.     * +---+---+---+---+
20.     * X   |   |   |   |
21.     * +---+---+---+---+
22.     * X   |   |   |   |
23.     * +---+---+---+---+
24.     * X   |   |   |   |
25.     * +---+---+---+---+
26.     * X   |   |   |   |
27.     * +---+---+---+---+
28.     *
29.     * xstride=stride, ystride=1
30.     *
31.     * ---X---X---X---X---
32.     * |   |   |   |   |
33.     * +---+---+---+---+
34.     * |   |   |   |   |
35.     * +---+---+---+---+
36.     * |   |   |   |   |
37.     * +---+---+---+---+
38.     * |   |   |   |   |
39.     * +---+---+---+---+
40.     */
41.    //一般inner_iters=4
42.    for( d = 0; d < 4 * inner_iters; d++ ) {
43.        //p和q
44.        //如果xstride=stride, ystride=1
45.        //就是处理纵向的6个像素
46.        //对应的是方块的横向边界的滤波（后文以此举例子）。如下所示：
47.        //      p2
48.        //      p1
49.        //      p0
50.        //=====图像边界=====
51.        //      q0
52.        //      q1
53.        //      q2
54.        //
55.        //如果xstride=1, ystride=stride
56.        //就是处理纵向的6个像素
57.        //对应的是方块的横向边界的滤波，即如下所示：
58.        //      ||
59.        // p2 p1 p0 || q0 q1 q2
60.        //      ||
61.        //      边界
62.
63.        //注意：这里乘的是xstride
64.        const int p2 = pix[-3*xstride];
65.        const int p1 = pix[-2*xstride];
66.        const int p0 = pix[-1*xstride];
67.
68.        const int q0 = pix[ 0*xstride];
69.        const int q1 = pix[ 1*xstride];
70.        const int q2 = pix[ 2*xstride];
71.
72.        if( FFABS( p0 - q0 ) < alpha &&
73.            FFABS( p1 - p0 ) < beta &&
74.            FFABS( q1 - q0 ) < beta ) {
75.            //满足条件的时候，使用强滤波器
76.            if(FFABS( p0 - q0 ) < (( alpha >> 2 ) + 2 )){
77.                //

```

```

77. //p
78. if( FFABS( p2 - p0 ) < beta)
79. {
80.     const int p3 = pix[-4*xstride];
81.     /* p0', p1', p2' */
82.     pix[-1*xstride] = ( p2 + 2*p1 + 2*p0 + 2*q0 + q1 + 4 ) >> 3;
83.     pix[-2*xstride] = ( p2 + p1 + p0 + q0 + 2 ) >> 2;
84.     pix[-3*xstride] = ( 2*p3 + 3*p2 + p1 + p0 + q0 + 4 ) >> 3;
85. } else {
86.     //不满足条件的时候
87.     /* p0' */
88.     pix[-1*xstride] = ( 2*p1 + p0 + q1 + 2 ) >> 2;
89. }
90. //q
91. if( FFABS( q2 - q0 ) < beta)
92. {
93.     const int q3 = pix[3*xstride];
94.     /* q0', q1', q2' */
95.     pix[0*xstride] = ( p1 + 2*p0 + 2*q0 + 2*q1 + q2 + 4 ) >> 3;
96.     pix[1*xstride] = ( p0 + q0 + q1 + q2 + 2 ) >> 2;
97.     pix[2*xstride] = ( 2*q3 + 3*q2 + q1 + q0 + p0 + 4 ) >> 3;
98. } else {
99.     /* q0' */
100.    pix[0*xstride] = ( 2*q1 + q0 + p1 + 2 ) >> 2;
101. }
102. }else{
103.     //不满足条件的时候, 使用下式修正
104.     /* p0', q0' */
105.     pix[-1*xstride] = ( 2*p1 + p0 + q1 + 2 ) >> 2;
106.     pix[ 0*xstride] = ( 2*q1 + q0 + p1 + 2 ) >> 2;
107. }
108. }
109. pix += ystride;
110. }
111. }

```

由于源代码中写了比较充分的注释, 在这里就不再逐行解析代码了。可以看出函数中包含了一个会执行16次的for()循环。for()循环执行一遍即完成了一次水平（或者垂直）的滤波, 所以for()循环执行完毕的时候, 就完成了对宏块中一个纵向边界（或者横向边界）的滤波。

函数的输入参数xstride和ystride决定了函数是水平滤波器还是垂直滤波器。如果xstride=stride、ystride=1, 滤波器处理垂直的8个像素, 为垂直滤波器；xstride=1、ystride=stride, 滤波器处理水平的8个像素, 为水平滤波器。

函数在确定了处理的8个点之后, 就会根据滤波的门限值alpha和beta判定边界是否满足滤波条件。如果满足条件, 就会根据下面的公式进行滤波（只列出p点的, q点类似）：

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

h264_h_loop_filter_luma_intra_8_c()

h264_v_loop_filter_luma_intra_8_c()实现了亮度边界水平强滤波器（处理垂直边界）。该函数的定义位于libavcodec/h264dsp_template.c, 如下所示。

```

1. //水平 (Horizontal) 强滤波器
2. // 边界
3. // |
4. // x x x | x x x
5. // |
6. static void h264_h_loop_filter_luma_intra_8_c(uint8_t *pix, int stride, int alpha, int beta)
7. {
8.     //xstride=1
9.     //ystride=stride
10.    //inner_iters=4
11.    h264_loop_filter_luma_intra_8_c(pix, sizeof(pixel), stride, 4, alpha, beta);
12. }

```

可以看出h264_h_loop_filter_luma_intra_8_c()和h264_v_loop_filter_luma_intra_8_c()类似, 都调用了h264_loop_filter_luma_intra_8_c()。

至此FFmpeg H.264解码器熵解码的部分就分析完毕了。

雷霄骅

leixiaohua1020@126.com

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

版权声明：本文为博主原创文章，未经博主允许不得转载。<https://blog.csdn.net/leixiaohua1020/article/details/45224579>

文章标签：[FFmpeg](#) [解码](#) [环路滤波](#) [源代码](#) [H.264](#)

个人分类：[FFMPEG](#)

所属专栏：[FFmpeg](#)

此PDF由[spygg](#)生成,请尊重原作者版权!!!

我的邮箱:liushidc@163.com