

原 x264源代码简单分析：滤波（Filter）部分

2015年05月20日 22:55:15 阅读数：7455

=====

H.264源代码分析文章列表：

【编码 - x264】

[x264源代码简单分析：概述](#)

[x264源代码简单分析：x264命令行工具（x264.exe）](#)

[x264源代码简单分析：编码器主干部分-1](#)

[x264源代码简单分析：编码器主干部分-2](#)

[x264源代码简单分析：x264_slice_write\(\)](#)

[x264源代码简单分析：滤波（Filter）部分](#)

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

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

[x264源代码简单分析：宏块编码（Encode）部分](#)

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

[FFmpeg与libx264接口源代码简单分析](#)

【解码 - libavcodec H.264 解码器】

[FFmpeg的H.264解码器源代码简单分析：概述](#)

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

[FFmpeg的H.264解码器源代码简单分析：解码器主干部分](#)

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

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

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

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

=====

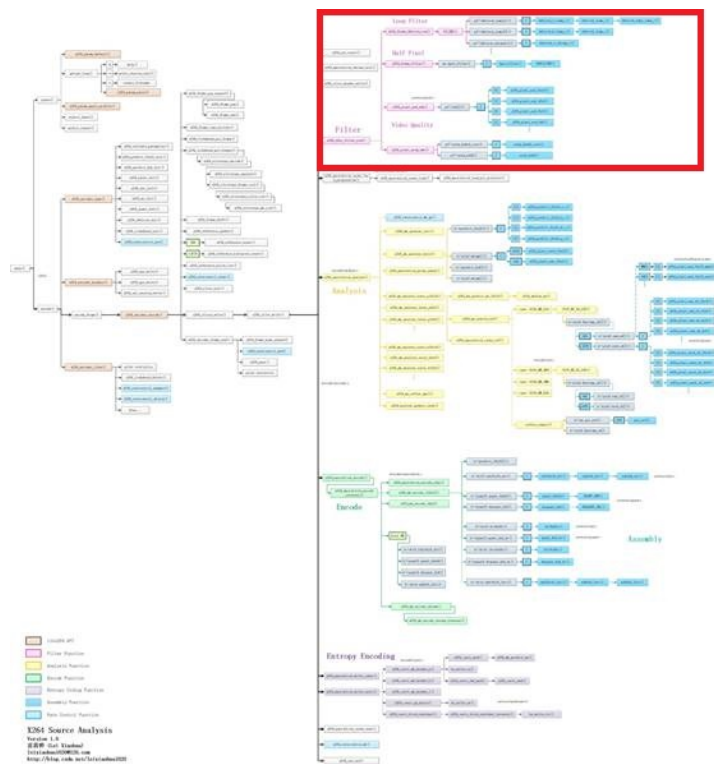
本文记录x264的x264_slice_write()函数中调用的x264_fdec_filter_row()的源代码。x264_fdec_filter_row()对应着x264中的滤波模块。滤波模块主要完成了下面3个方面的功能：

- （1）环路滤波（去块效应滤波）
- （2）半像素内插
- （3）视频质量指标PSNR和SSIM的计算

本文分别记录上述3个方面的源代码。

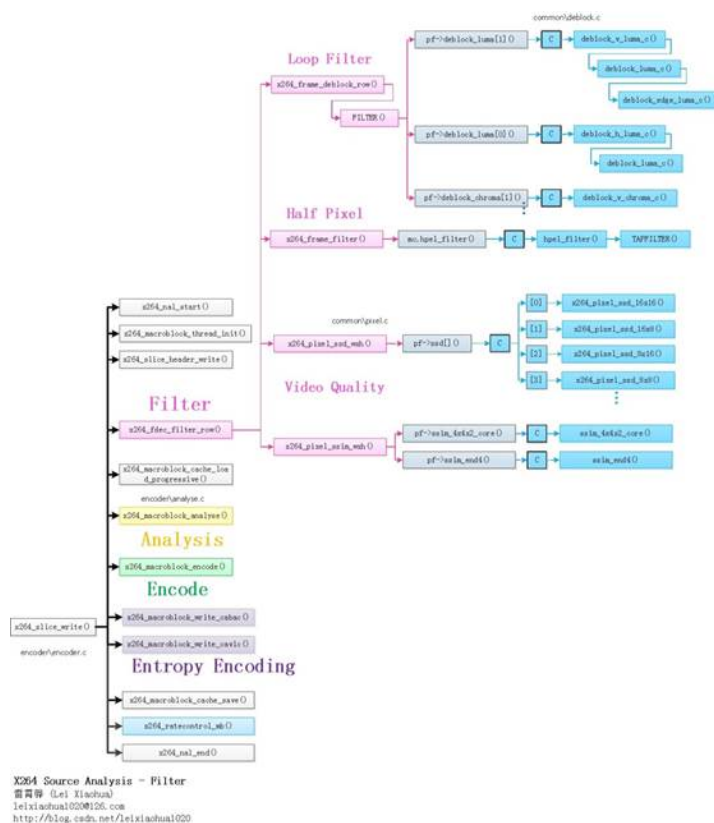
函数调用关系图

滤波（Filter）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

滤波（Filter）部分的函数调用关系如下图所示。



单击查看更清晰的图片

从图中可以看出，滤波模块对应的x264_fdec_filter_row调用了如下函数：

- x264_frame_deblock_row()：去块效应滤波器。
- x264_frame_filter()：半像素插值。
- x264_pixel_ssd_wxh()：PSNR计算。
- x264_pixel_ssim_wxh()：SSIM计算。


x264_slice_write()

x264_slice_write()是x264项目的核心，它完成了编码了一个Slice的工作。有关该函数的分析可以参考文章《[x264源代码简单分析：x264_slice_w](#)

ite() 》。本文分析其调用的x264_fdec_filter_row()函数。

x264_fdec_filter_row()

x264_fdec_filter_row()用于对一行宏块进行滤波。该函数的定义位于encoder\encoder.c，如下所示。

```
[cpp]  
1.  /*****
2.  * 滤波-去块效应滤波、半像素插值、SSIM/PSNR计算等
3.  * 一次处理一行宏块
4.  *
5.  * 注释和处理：雷霄骅
6.  * http://blog.csdn.net/leixiaohua1020
7.  * leixiaohua1020@126.com
8.  *****/
9.  static void x264_fdec_filter_row( x264_t *h, int mb_y, int pass )
10. {
11.     /* mb_y is the mb to be encoded next, not the mb to be filtered here */
12.     int b_hpel = h->fdec->b_kept_as_ref;
13.     int b_deblock = h->sh.i_disable_deblocking_filter_idc != 1;
14.     int b_end = mb_y == h->i_threadslice_end;
15.     int b_measure_quality = 1;
16.     int min_y = mb_y - (1 << SLICE_MBAFF);
17.     int b_start = min_y == h->i_threadslice_start;
18.     /* Even in interlaced mode, deblocking never modifies more than 4 pixels
19.     * above each MB, as b5=4 doesn't happen for the top of interlaced mbpairs. */
20.     int minpix_y = min_y*16 - 4 * !b_start;
21.     int maxpix_y = mb_y*16 - 4 * !b_end;
22.     b_deblock &= b_hpel || h->param.b_full_recon || h->param.psz_dump_yuv;
23.     if( h->param.b_sliced_threads )
24.     {
25.         switch( pass )
26.         {
27.             /* During encode: only do deblock if asked for */
28.             default:
29.             case 0:
30.                 b_deblock &= h->param.b_full_recon;
31.                 b_hpel = 0;
32.                 break;
33.             /* During post-encode pass: do deblock if not done yet, do hpel for all
34.             * rows except those between slices. */
35.             case 1:
36.                 b_deblock &= !h->param.b_full_recon;
37.                 b_hpel &= !(b_start && min_y > 0);
38.                 b_measure_quality = 0;
39.                 break;
40.             /* Final pass: do the rows between slices in sequence. */
41.             case 2:
42.                 b_deblock = 0;
43.                 b_measure_quality = 0;
44.                 break;
45.         }
46.     }
47.     if( mb_y & SLICE_MBAFF )
48.         return;
49.     if( min_y < h->i_threadslice_start )
50.         return;
51.     //去块效应滤波
52.     if( b_deblock )
53.         for( int y = min_y; y < mb_y; y += (1 << SLICE_MBAFF) )
54.             x264_frame_deblock_row( h, y );//处理一行
55.
56.     /* FIXME: Prediction requires different borders for interlaced/progressive mc,
57.     * but the actual image data is equivalent. For now, maintain this
58.     * consistency by copying deblocked pixels between planes. */
59.     if( PARAM_INTERLACED && (!h->param.b_sliced_threads || pass == 1) )
60.         for( int p = 0; p < h->fdec->i_plane; p++ )
61.             for( int i = minpix_y>>(CHROMA_V_SHIFT && p); i < maxpix_y>>(CHROMA_V_SHIFT && p); i++ )
62.                 memcpy( h->fdec->plane_fld[p] + i*h->fdec->i_stride[p],
63.                     h->fdec->plane[p] + i*h->fdec->i_stride[p],
64.                     h->mb.i_mb_width*16*sizeof(pixel) );
65.
66.     if( h->fdec->b_kept_as_ref && (!h->param.b_sliced_threads || pass == 1) )
67.         x264_frame_expand_border( h, h->fdec, min_y );
68.     //半像素内插
69.     if( b_hpel )
70.     {
71.         int end = mb_y == h->mb.i_mb_height;
72.         /* Can't do hpel until the previous slice is done encoding. */
73.         if( h->param.analyse.i_subpel_refine )
74.         {
75.             //半像素内插
76.             x264_frame_filter( h, h->fdec, min_y, end );
77.             x264_frame_expand_border_filtered( h, h->fdec, min_y, end );
78.         }
79.     }
80. }
```

```

81.     if( SLICE_MBAFF && pass == 0 )
82.         for( int i = 0; i < 3; i++ )
83.         {
84.             XCHG( pixel *, h->intra_border_backup[0][i], h->intra_border_backup[3][i] );
85.             XCHG( pixel *, h->intra_border_backup[1][i], h->intra_border_backup[4][i] );
86.         }
87.
88.     if( h->i_thread_frames > 1 && h->fdec->b_kept_as_ref )
89.         x264_frame_cond_broadcast( h->fdec, mb_y*16 + (b_end ? 10000 : -(X264_THREAD_HEIGHT << SLICE_MBAFF)) );
90.
91.     //计算编码的质量
92.     if( b_measure_quality )
93.     {
94.         maxpix_y = X264_MIN( maxpix_y, h->param.i_height );
95.         //如果需要打印输出PSNR
96.         if( h->param.analyse.b_psnr )
97.         {
98.             //实际上是计算SSD
99.             //输出的时候调用x264_psnr()换算SSD为PSNR
100.            /**
101.             * 计算PSNR的过程
102.             *
103.             * MSE = SSD*1/(w*h)
104.             * PSNR= 10*log10(MAX^2/MSE)
105.             *
106.             * 其中MAX指的是图像的灰度级, 对于8bit来说就是2^8-1=255
107.             */
108.            for( int p = 0; p < (CHROMA444 ? 3 : 1); p++ )
109.                h->stat.frame.i_ssd[p] += x264_pixel_ssd_wxh( &h->pixf,
110.                    h->fdec->plane[p] + minpix_y * h->fdec->i_stride[p], h->fdec->i_stride[p], //重建帧
111.                    h->fenc->plane[p] + minpix_y * h->fenc->i_stride[p], h->fenc->i_stride[p], //编码帧
112.                    h->param.i_width, maxpix_y-minpix_y );
113.            if( !CHROMA444 )
114.            {
115.                uint64_t ssd_u, ssd_v;
116.                int v_shift = CHROMA_V_SHIFT;
117.                x264_pixel_ssd_nv12( &h->pixf,
118.                    h->fdec->plane[1] + (minpix_y>>v_shift) * h->fdec->i_stride[1], h->fdec->i_stride[1],
119.                    h->fenc->plane[1] + (minpix_y>>v_shift) * h->fenc->i_stride[1], h->fenc->i_stride[1],
120.                    h->param.i_width>>1, (maxpix_y-minpix_y)>>v_shift, &ssd_u, &ssd_v );
121.                h->stat.frame.i_ssd[1] += ssd_u;
122.                h->stat.frame.i_ssd[2] += ssd_v;
123.            }
124.        }
125.        //如果需要打印输出SSIM
126.        if( h->param.analyse.b_ssim )
127.        {
128.            int ssim_cnt;
129.            x264_emms();
130.            /* offset by 2 pixels to avoid alignment of ssim blocks with dct blocks,
131.             * and overlap by 4 */
132.            minpix_y += b_start ? 2 : -6;
133.            //计算SSIM
134.            h->stat.frame.f_ssim +=
135.                x264_pixel_ssim_wxh( &h->pixf,
136.                    h->fdec->plane[0] + 2*minpix_y*h->fdec->i_stride[0], h->fdec->i_stride[0], //重建帧
137.                    h->fenc->plane[0] + 2*minpix_y*h->fenc->i_stride[0], h->fenc->i_stride[0], //编码帧
138.                    h->param.i_width-2, maxpix_y-minpix_y, h->scratch_buffer, &ssim_cnt );
139.            h->stat.frame.i_ssim_cnt += ssim_cnt;
140.        }
141.    }
142. }

```

从源代码可以看出, x264_fdec_filter_row()完成了三步工作:

- (1) 环路滤波 (去块效应滤波)。通过调用x264_frame_deblock_row()实现。
- (2) 半像素内插。通过调用x264_frame_filter()实现。
- (3) 视频质量SSIM和PSNR计算。PSNR在这里只计算了SSD, 通过调用x264_pixel_ssd_wxh()实现; SSIM的计算则是通过x264_pixel_ssim_wxh()实现。

x264_frame_deblock_row()

x264_frame_deblock_row()用于进行环路滤波 (去块效应滤波)。该函数的定义位于common\deblock.c, 如下所示。

```

[cpp]
1. //去块效应滤波
2. void x264_frame_deblock_row( x264_t *h, int mb_y )
3. {
4.     int b_interlaced = SLICE_MBAFF;
5.     int a = h->sh.i_alpha_c0_offset - QP_BD_OFFSET;
6.     int b = h->sh.i_beta_offset - QP_BD_OFFSET;
7.     int qp_thresh = 15 - X264_MIN( a, b ) - X264_MAX( 0, h->pps->i_chroma_qp_index_offset );
8.     int stridey = h->fdec->i_stride[0];
9.     int strideuv = h->fdec->i_stride[1];
10.    int chroma444 = CHROMA444;
11.    int chroma_height = 16 >> CHROMA_V_SHIFT;
12.    intptr_t uvdiff = chroma444 ? h->fdec->plane[2] - h->fdec->plane[1] : 1;

```

```

13.
14.     for( int mb_x = 0; mb_x < h->mb.i_mb_width; mb_x += (~b_interlaced | mb_y)&1, mb_y ^= b_interlaced )
15.     {
16.         x264_prefetch_fenc( h, h->fdec, mb_x, mb_y );
17.         x264_macroblock_cache_load_neighbours_deblock( h, mb_x, mb_y );
18.
19.         int mb_xy = h->mb.i_mb_xy;
20.         int transform_8x8 = h->mb.mb_transform_size[mb_xy];
21.         int intra_cur = IS_INTRA( h->mb.type[mb_xy] );
22.         uint8_t (*bs)[8][4] = h->deblock_strength[mb_y&1][h->param.b_sliced_threads?mb_xy:mb_x];
23.         //找到像素数据 (宏块的大小是16x16)
24.         pixel *pixy = h->fdec->plane[0] + 16*mb_y*stridey + 16*mb_x;
25.         pixel *pixuv = h->fdec->plane[1] + chroma_height*mb_y*strideuv + 16*mb_x;
26.
27.         if( mb_y & MB_INTERLACED )
28.         {
29.             pixy -= 15*stridey;
30.             pixuv -= (chroma_height-1)*strideuv;
31.         }
32.
33.         int stride2y = stridey << MB_INTERLACED;
34.         int stride2uv = strideuv << MB_INTERLACED;
35.         //QP, 用于计算环路滤波的门槛值alpha和beta
36.         int qp = h->mb.qp[mb_xy];
37.         int qpc = h->chroma_qp_table[qp];
38.         int first_edge_only = (h->mb.partition[mb_xy] == D_16x16 && !h->mb.cbp[mb_xy] && !intra_cur) || qp <= qp_thresh;
39.
40.         /*
41.          * 滤波顺序如下所示 (大方框代表16x16块)
42.          *
43.          * +- -4-+- -4-+- -4-+- -4-+
44.          * 0  1  2  3  |
45.          * +- -5-+- -5-+- -5-+- -5-+
46.          * 0  1  2  3  |
47.          * +- -6-+- -6-+- -6-+- -6-+
48.          * 0  1  2  3  |
49.          * +- -7-+- -7-+- -7-+- -7-+
50.          * 0  1  2  3  |
51.          * +-----+-----+-----+
52.          *
53.          */
54.
55.         //一个比较长的宏, 用于进行环路滤波
56.         //根据不同的情况传递不同的参数
57.         //几个参数的含义:
58.         //intra:
59.         //为“intra”的时候:
60.         //其中的“deblock_edge##intra()”展开为函数deblock_edge_intra()
61.         //其中的“h->loopf.deblock_luma##intra[dir]”展开为强滤波汇编函数h->loopf.deblock_luma_intra[dir]()
62.         //为“” (空), 其中的“deblock_edge##intra()”展开为函数deblock_edge()
63.         //其中的“h->loopf.deblock_luma##intra[dir]”展开为普通滤波汇编函数h->loopf.deblock_luma[dir]()
64.         //dir:
65.         //决定了滤波的方向: 0为水平滤波器 (垂直边界), 1为垂直滤波器 (水平边界)
66.         #define FILTER( intra, dir, edge, qp, chroma_qp )\
67.         do\
68.         {\
69.             if( !(edge & 1) || !transform_8x8 )\
70.             {\
71.                 deblock_edge##intra( h, pixy + 4*edge*(dir?stride2y:1),\
72.                                     stride2y, bs[dir][edge], qp, a, b, 0,\
73.                                     h->loopf.deblock_luma##intra[dir] );\
74.                 if( CHROMA_FORMAT == CHROMA_444 )\
75.                 {\
76.                     deblock_edge##intra( h, pixuv + 4*edge*(dir?stride2uv:1),\
77.                                           stride2uv, bs[dir][edge], chroma_qp, a, b, 0,\
78.                                           h->loopf.deblock_luma##intra[dir] );\
79.                     deblock_edge##intra( h, pixuv + uvdiff + 4*edge*(dir?stride2uv:1),\
80.                                           stride2uv, bs[dir][edge], chroma_qp, a, b, 0,\
81.                                           h->loopf.deblock_luma##intra[dir] );\
82.                 }\
83.                 else if( CHROMA_FORMAT == CHROMA_420 && !(edge & 1) )\
84.                 {\
85.                     deblock_edge##intra( h, pixuv + edge*(dir?2*stride2uv:4),\
86.                                           stride2uv, bs[dir][edge], chroma_qp, a, b, 1,\
87.                                           h->loopf.deblock_chroma##intra[dir] );\
88.                 }\
89.             }\
90.             if( CHROMA_FORMAT == CHROMA_422 && (dir || !(edge & 1)) )\
91.             {\
92.                 deblock_edge##intra( h, pixuv + edge*(dir?4*stride2uv:4),\
93.                                     stride2uv, bs[dir][edge], chroma_qp, a, b, 1,\
94.                                     h->loopf.deblock_chroma##intra[dir] );\
95.             }\
96.         }\
97.         } while(0)
98.
99.         if( h->mb.i_neighbour & MB_LEFT )
100.        {
101.            if( b_interlaced && h->mb.field[h->mb.i_mb_left_xy[0]] != MB_INTERLACED )
102.            {
103.                //隔行的

```

```

104.     int luma_qp[2];
105.     int chroma_qp[2];
106.     int left_qp[2];
107.     x264_deblock_inter_t luma_deblock = h->loopf.deblock_luma_mbaff;
108.     x264_deblock_inter_t chroma_deblock = h->loopf.deblock_chroma_mbaff;
109.     x264_deblock_intra_t luma_intra_deblock = h->loopf.deblock_luma_intra_mbaff;
110.     x264_deblock_intra_t chroma_intra_deblock = h->loopf.deblock_chroma_intra_mbaff;
111.     int c = chroma444 ? 0 : 1;
112.
113.     left_qp[0] = h->mb.qp[h->mb.i_mb_left_xy[0]];
114.     luma_qp[0] = (qp + left_qp[0] + 1) >> 1;
115.     chroma_qp[0] = (qpc + h->chroma_qp_table[left_qp[0]] + 1) >> 1;
116.     if( intra_cur || IS_INTRA( h->mb.type[h->mb.i_mb_left_xy[0]] ) )
117.     {
118.         deblock_edge_intra( h, pixy,          2*stridey, bs[0][0], luma_qp[0],  a, b, 0, luma_intra_deblock );
119.         deblock_edge_intra( h, pixuv,        2*strideuv, bs[0][0], chroma_qp[0], a, b, c, chroma_intra_deblock );
120.         if( chroma444 )
121.             deblock_edge_intra( h, pixuv + uvdiff, 2*strideuv, bs[0]
122. [0], chroma_qp[0], a, b, c, chroma_intra_deblock );
123.     }
124.     else
125.     {
126.         deblock_edge( h, pixy,          2*stridey, bs[0][0], luma_qp[0],  a, b, 0, luma_deblock );
127.         deblock_edge( h, pixuv,        2*strideuv, bs[0][0], chroma_qp[0], a, b, c, chroma_deblock );
128.         if( chroma444 )
129.             deblock_edge( h, pixuv + uvdiff, 2*strideuv, bs[0][0], chroma_qp[0], a, b, c, chroma_deblock );
130.     }
131.     int offy = MB_INTERLACED ? 4 : 0;
132.     int offuv = MB_INTERLACED ? 4-CHROMA_V_SHIFT : 0;
133.     left_qp[1] = h->mb.qp[h->mb.i_mb_left_xy[1]];
134.     luma_qp[1] = (qp + left_qp[1] + 1) >> 1;
135.     chroma_qp[1] = (qpc + h->chroma_qp_table[left_qp[1]] + 1) >> 1;
136.     if( intra_cur || IS_INTRA( h->mb.type[h->mb.i_mb_left_xy[1]] ) )
137.     {
138.         deblock_edge_intra( h, pixy          + (stridey<<offy), 2*stridey, bs[0]
139. [4], luma_qp[1],  a, b, 0, luma_intra_deblock );
140.         deblock_edge_intra( h, pixuv          + (strideuv<<offuv), 2*strideuv, bs[0]
141. [4], chroma_qp[1], a, b, c, chroma_intra_deblock );
142.         if( chroma444 )
143.             deblock_edge_intra( h, pixuv + uvdiff + (strideuv<<offuv), 2*strideuv, bs[0]
144. [4], chroma_qp[1], a, b, c, chroma_intra_deblock );
145.     }
146.     else
147.     {
148.         deblock_edge( h, pixy          + (stridey<<offy), 2*stridey, bs[0]
149. [4], luma_qp[1],  a, b, 0, luma_deblock );
150.         deblock_edge( h, pixuv          + (strideuv<<offuv), 2*strideuv, bs[0]
151. [4], chroma_qp[1], a, b, c, chroma_deblock );
152.         if( chroma444 )
153.             deblock_edge( h, pixuv + uvdiff + (strideuv<<offuv), 2*strideuv, bs[0]
154. [4], chroma_qp[1], a, b, c, chroma_deblock );
155.     }
156.     }
157.     else
158.     {
159.         //逐行的
160.
161.         //左边宏块的qp
162.         int qpl = h->mb.qp[h->mb.i_mb_xy-1];
163.         int qp_left = (qp + qpl + 1) >> 1;
164.         int qpc_left = (qpc + h->chroma_qp_table[qpl] + 1) >> 1;
165.         //Intra宏块左边宏块的qp
166.         int intra_left = IS_INTRA( h->mb.type[h->mb.i_mb_xy-1] );
167.         int intra_deblock = intra_cur || intra_left;
168.
169.         /* Any MB that was coded, or that analysis decided to skip, has quality commensurate with its QP.
170.          * But if deblocking affects neighboring MBs that were force-skipped, blur might accumulate there.
171.          * So reset their effective QP to max, to indicate that lack of guarantee. */
172.         if( h->fdec->mb_info && M32( bs[0][0] ) )
173.         {
174.             #define RESET_EFFECTIVE_QP(xy) h->fdec->effective_qp[xy] |= 0xff * !(h->fdec->mb_info[xy] & X264_MBINFO_CONSTANT);
175.             RESET_EFFECTIVE_QP(mb_xy);
176.             RESET_EFFECTIVE_QP(h->mb.i_mb_left_xy[0]);
177.         }
178.
179.         if( intra_deblock )
180.             FILTER( _intra, 0, 0, qp_left, qpc_left );// [0] 强滤波, 水平滤波器 (垂直边界)
181.         else
182.             FILTER( , 0, 0, qp_left, qpc_left );// [0] 普通滤波, 水平滤波器 (垂直边界)
183.     }
184. }
185.
186. if( !first_edge_only )
187. {
188.     //普通滤波, 水平滤波器 (垂直边界)
189.     FILTER( , 0, 1, qp, qpc );// [1]
190.     FILTER( , 0, 2, qp, qpc );// [2]
191.     FILTER( , 0, 3, qp, qpc );// [3]
192. }
193.
194. if( h->mb.i_neighbour & MB_TOP )

```

```

188.     {
189.         if( b_interlaced && !(mb_y&1) && !MB_INTERLACED && h->mb.field[h->mb.i_mb_top_xy] )
190.         {
191.             int mbn_xy = mb_xy - 2 * h->mb.i_mb_stride;
192.
193.             for( int j = 0; j < 2; j++, mbn_xy += h->mb.i_mb_stride )
194.             {
195.                 int qpt = h->mb.qp[mbn_xy];
196.                 int qp_top = (qp + qpt + 1) >> 1;
197.                 int qpc_top = (qpc + h->chroma_qp_table[qpt] + 1) >> 1;
198.                 int intra_top = IS_INTRA( h->mb.type[mbn_xy] );
199.                 if( intra_cur || intra_top )
200.                     M32( bs[1][4*j] ) = 0x03030303;
201.
202.                 // deblock the first horizontal edge of the even rows, then the first horizontal edge of the odd rows
203.                 deblock_edge( h, pixy + j*stridey, 2* stridey, bs[1][4*j], qp_top, a, b, 0, h->loopf.deblock_luma[1] );
204.                 if( chroma444 )
205.                 {
206.                     deblock_edge( h, pixuv + j*strideuv, 2*strideuv, bs[1][4*j], qp_top, a, b, 0, h->loopf.deblock_lum
a[1] );
207.                     deblock_edge( h, pixuv + uvdiff + j*strideuv, 2*strideuv, bs[1][4*j], qp_top, a, b, 0, h->loopf.deblock_lum
a[1] );
208.                 }
209.                 else
210.                     deblock_edge( h, pixuv + j*strideuv, 2*strideuv, bs[1][4*j], qp_top, a, b, 1, h->loopf.deblock_chr
oma[1] );
211.             }
212.         }
213.         else
214.         {
215.             int qpt = h->mb.qp[h->mb.i_mb_top_xy];
216.             int qp_top = (qp + qpt + 1) >> 1;
217.             int qpc_top = (qpc + h->chroma_qp_table[qpt] + 1) >> 1;
218.             int intra_top = IS_INTRA( h->mb.type[h->mb.i_mb_top_xy] );
219.             int intra_deblock = intra_cur || intra_top;
220.
221.             /* This edge has been modified, reset effective qp to max. */
222.             if( h->fdec->mb_info && M32( bs[1][0] ) )
223.             {
224.                 RESET_EFFECTIVE_QP(mb_xy);
225.                 RESET_EFFECTIVE_QP(h->mb.i_mb_top_xy);
226.             }
227.
228.             if( (!b_interlaced || (!MB_INTERLACED && !h->mb.field[h->mb.i_mb_top_xy])) && intra_deblock )
229.             {
230.                 FILTER( _intra, 1, 0, qp_top, qpc_top );// 【4】 普通滤波, 垂直滤波器 (水平边界)
231.             }
232.             else
233.             {
234.                 if( intra_deblock )
235.                     M32( bs[1][0] ) = 0x03030303;
236.                 FILTER( , 1, 0, qp_top, qpc_top );// 【4】 普通滤波, 垂直滤波器 (水平边界)
237.             }
238.         }
239.     }
240.
241.     if( !first_edge_only )
242.     {
243.         //普通滤波, 垂直滤波器 (水平边界)
244.         FILTER( , 1, 1, qp, qpc );// 【5】
245.         FILTER( , 1, 2, qp, qpc );// 【6】
246.         FILTER( , 1, 3, qp, qpc );// 【7】
247.     }
248.
249.     #undef FILTER
250. }
251. }

```

从源代码可以看出，x264_frame_deblock_row()中有一个很长的宏定义“FILTER()”定义了函数调用的方式。FILTER(intra, dir, edge, qp, chroma_qp)中：

“intra”指定了是普通滤波（Bs=1, 2, 3）还是强滤波（Bs=4）；

“dir”指定了滤波器的方向。0为水平滤波器（垂直边界），1为垂直滤波器（水平边界）；

“edge”指定了边界的位置。“0”，“1”，“2”，“3”分别代表了水平（或者垂直）的4条边界；

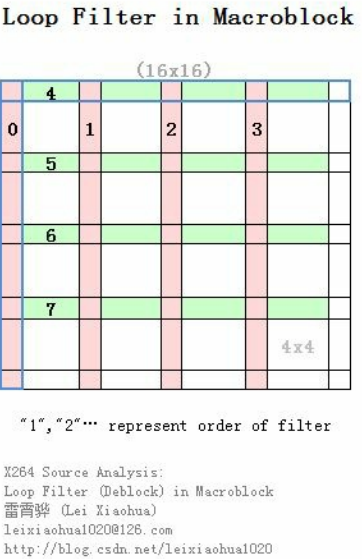
滤波的主干代码如下所示。

```

[cpp]
1.  FILTER( _intra, 0, 0, qp_left, qpc_left );// 【0】 强滤波, 水平滤波器 (垂直边界)
2.  //普通滤波, 水平滤波器 (垂直边界)
3.  FILTER( , 0, 1, qp, qpc );// 【1】
4.  FILTER( , 0, 2, qp, qpc );// 【2】
5.  FILTER( , 0, 3, qp, qpc );// 【3】
6.  FILTER( _intra, 1, 0, qp_top, qpc_top );// 【4】 普通滤波, 垂直滤波器 (水平边界)
7.  //普通滤波, 垂直滤波器 (水平边界)
8.  FILTER( , 1, 1, qp, qpc );// 【5】
9.  FILTER( , 1, 2, qp, qpc );// 【6】
10. FILTER( , 1, 3, qp, qpc );// 【7】

```

上述代码滤波的顺序如下图所示。图中蓝色边缘的边界是强滤波，其他边界是普通滤波。



下面分别看一下两个宏“FILTER(_intra, 0, 0, qp_left, qpc_left)”和“FILTER(, 0, 1, qp, qpc)”展开后的代码。

FILTER(_intra, 0, 0, qp_left, qpc_left)

FILTER(_intra, 0, 0, qp_left, qpc_left)用于对上文图中“0”号垂直边界进行强滤波（Bs=4）。该宏的展开结果如下所示。

```
[cpp]
1. do
2. {
3.     if( !(0 & 1) || !transform_8x8 )
4.     {
5.         deblock_edge_intra( h, pixy + 4*0*(0?stride2y:1),
6.                             stride2y, bs[0][0], qp_left, a, b, 0,
7.                             h->loopf.deblock_luma_intra[0] );
8.         if( h->sps->i_chroma_format_idc == CHROMA_444 )
9.         {
10.            deblock_edge_intra( h, pixuv + 4*0*(0?stride2uv:1),
11.                                stride2uv, bs[0][0], qpc_left, a, b, 0,
12.                                h->loopf.deblock_luma_intra[0] );
13.            deblock_edge_intra( h, pixuv + uvdiff + 4*0*(0?stride2uv:1),
14.                                stride2uv, bs[0][0], qpc_left, a, b, 0,
15.                                h->loopf.deblock_luma_intra[0] );
16.        }
17.        else if( h->sps->i_chroma_format_idc == CHROMA_420 && !(0 & 1) )
18.        {
19.            deblock_edge_intra( h, pixuv + 0*(0?4*stride2uv:4),
20.                                stride2uv, bs[0][0], qpc_left, a, b, 1,
21.                                h->loopf.deblock_chroma_intra[0] );
22.        }
23.    }
24.    if( h->sps->i_chroma_format_idc == CHROMA_422 && (0 || !(0 & 1)) )
25.    {
26.        deblock_edge_intra( h, pixuv + 0*(0?4*stride2uv:4),
27.                            stride2uv, bs[0][0], qpc_left, a, b, 1,
28.                            h->loopf.deblock_chroma_intra[0] );
29.    }
30. } while(0)
```

从代码中可以看出，FILTER(_intra, 0, 0, qp_left, qpc_left)调用了deblock_edge_intra()完成了强滤波。该函数的最后一个参数指定了环路滤波的汇编函数，在这里是h->loopf.deblock_luma_intra[0]()。有关h->loopf.deblock_luma_intra[0]()的代码在后面进行分析。

deblock_edge_intra()

deblock_edge_intra()通过调用相应的滤波函数完成强滤波（Bs=4）。该函数的定义位于common\deblock.c，如下所示。


```

1. //强滤波 (Bs取值为4)
2. static ALWAYS_INLINE void deblock_edge_intra( x264_t *h, pixel *pix, intptr_t i_stride, uint8_t bs[4], int i_qp,
3.                                               int a, int b, int b_chroma, x264_deblock_intra_t pf_intra )
4. {
5.     int index_a = i_qp + a;
6.     int index_b = i_qp + b;
7.     //根据QP, 通过查表的方法获得是否滤波的门限值alpha和beta
8.     //alpha为边界两边2点的门限值
9.     //beta为边界一边最靠近边界的2点的门限值
10.    //总体说来, QP越大, alpha和beta越大, 越有可能滤波
11.    int alpha = alpha_table(index_a) << (BIT_DEPTH-8);
12.    int beta  = beta_table(index_b) << (BIT_DEPTH-8);
13.    //alpha或者beta有一个门限为0的时候, 根本不用滤波
14.    if( !alpha || !beta )
15.        return;
16.    //滤波函数, 通过传参而来
17.    pf_intra( pix, i_stride, alpha, beta );
18. }

```

从源代码可以看出, deblock_edge_intra()首先计算滤波的门限值alpha和beta, 然后调用通过参数传过来的pf_intra()汇编函数完成滤波。

FILTER(, 0, 1, qp, qpc)

FILTER(, 0, 1, qp, qpc)用于对上文图中“1”号垂直边界进行普通滤波 (Bs=1, 2, 3) 。该宏的展开结果如下所示。

```

1. do
2. {
3.     if( !(1 & 1) || !transform_8x8 )
4.     {
5.         deblock_edge( h, pixy + 4*1*(0?stride2y:1),
6.                       stride2y, bs[0][1], qp, a, b, 0,
7.                       h->loopf.deblock_luma[0] );
8.         if( h->sps->i_chroma_format_idc == CHROMA_444 )
9.         {
10.            deblock_edge( h, pixuv + 4*1*(0?stride2uv:1),
11.                          stride2uv, bs[0][1], qpc, a, b, 0,
12.                          h->loopf.deblock_luma[0] );
13.            deblock_edge( h, pixuv + uvdiff + 4*1*(0?stride2uv:1),
14.                          stride2uv, bs[0][1], qpc, a, b, 0,
15.                          h->loopf.deblock_luma[0] );
16.        }
17.        else if( h->sps->i_chroma_format_idc == CHROMA_420 && !(1 & 1) )
18.        {
19.            deblock_edge( h, pixuv + 1*(0?2*stride2uv:4),
20.                          stride2uv, bs[0][1], qpc, a, b, 1,
21.                          h->loopf.deblock_chroma[0] );
22.        }
23.    }
24.    if( h->sps->i_chroma_format_idc == CHROMA_422 && (0 || !(1 & 1)) )
25.    {
26.        deblock_edge( h, pixuv + 1*(0?4*stride2uv:4),
27.                      stride2uv, bs[0][1], qpc, a, b, 1,
28.                      h->loopf.deblock_chroma[0] );
29.    }
30. } while(0)

```

从代码中可以看出, FILTER(, 0, 1, qp, qpc)调用了deblock_edge()完成了普通滤波(Bs=1,2,3)。该函数的最后一个参数指定了环路滤波的汇编函数, 在这里是h->loopf.deblock_luma[0]()。有关h->loopf.deblock_luma[0]()的代码在后面进行分析。

deblock_edge()

deblock_edge()通过调用相应的滤波函数完成强滤波 (Bs=4) 。该函数的定义位于common\deblock.c, 如下所示。

```

1. //普通滤波 (Bs取值1-3)
2. static ALWAYS_INLINE void deblock_edge( x264_t *h, pixel *pix, intptr_t i_stride, uint8_t bs[4], int i_qp,
3.                                         int a, int b, int b_chroma, x264_deblock_inter_t pf_inter )
4. {
5.     int index_a = i_qp + a;
6.     int index_b = i_qp + b;
7.     //根据QP, 通过查表的方法获得是否滤波的门限值alpha和beta
8.     //alpha为边界两边2点的门限值
9.     //beta为边界一边最靠近边界的2点的门限值
10.    //总体说来, QP越大, alpha和beta越大, 越有可能滤波
11.    int alpha = alpha_table(index_a) << (BIT_DEPTH-8);
12.    int beta  = beta_table(index_b) << (BIT_DEPTH-8);
13.    int8_t tc[4];
14.    //alpha或者beta有一个门限为0的时候, 根本不用滤波
15.    if( !M32(bs) || !alpha || !beta )
16.        return;
17.
18.    tc[0] = (tc0_table(index_a)[bs[0]] << (BIT_DEPTH-8)) + b_chroma;
19.    tc[1] = (tc0_table(index_a)[bs[1]] << (BIT_DEPTH-8)) + b_chroma;
20.    tc[2] = (tc0_table(index_a)[bs[2]] << (BIT_DEPTH-8)) + b_chroma;
21.    tc[3] = (tc0_table(index_a)[bs[3]] << (BIT_DEPTH-8)) + b_chroma;
22.    //滤波函数, 通过传参而来
23.    pf_inter( pix, i_stride, alpha, beta, tc );
24. }

```

从源代码可以看出, deblock_edge()首先计算滤波的门限值alpha和beta, 然后计算tc[]的取值, 最后调用通过参数传过来的pf_inter()汇编函数完成滤波。
下文开始分析环路滤波模块调用的汇编函数。

环路滤波小知识

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

- (1) DCT变换后的量化造成误差 (主要原因)
- (2) 运动补偿

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



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

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。

环路滤波的门槛



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进行环路滤波。只有满足下面三个条件的时候才能进行环路滤波：

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

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

x264_deblock_init()

x264_deblock_init()用于初始化去块效应滤波器相关的汇编函数。该函数的定义位于common\deblock.c，如下所示。

```
[cpp]  
1. //去块效应滤波
2. void x264_deblock_init( int cpu, x264_deblock_function_t *pf, int b_mbafl )
3. {
4.     //注意：标记“v”的垂直滤波器是处理水平边界用的
5.     //亮度-普通滤波器-边界强度Bs=1,2,3
6.     pf->deblock_luma[1] = deblock_v_luma_c;
7.     pf->deblock_luma[0] = deblock_h_luma_c;
8.     //色度的
9.     pf->deblock_chroma[1] = deblock_v_chroma_c;
10.    pf->deblock_h_chroma_420 = deblock_h_chroma_c;
11.    pf->deblock_h_chroma_422 = deblock_h_chroma_422_c;
12.    //亮度-强滤波器-边界强度Bs=4
13.    pf->deblock_luma_intra[1] = deblock_v_luma_intra_c;
14.    pf->deblock_luma_intra[0] = deblock_h_luma_intra_c;
15.    pf->deblock_chroma_intra[1] = deblock_v_chroma_intra_c;
16.    pf->deblock_h_chroma_420_intra = deblock_h_chroma_intra_c;
17.    pf->deblock_h_chroma_422_intra = deblock_h_chroma_422_intra_c;
18.    pf->deblock_luma_mbafl = deblock_h_luma_mbafl_c;
19.    pf->deblock_chroma_420_mbafl = deblock_h_chroma_mbafl_c;
20.    pf->deblock_luma_intra_mbafl = deblock_h_luma_intra_mbafl_c;
21.    pf->deblock_chroma_420_intra_mbafl = deblock_h_chroma_intra_mbafl_c;
22.    pf->deblock_strength = deblock_strength_c;
23.
24.    #if HAVE_MMX
25.        if( cpu & X264_CPU_MMX2 )
26.        {
27.            #if ARCH_X86
28.                pf->deblock_luma[1] = x264_deblock_v_luma_mmx2;
29.                pf->deblock_luma[0] = x264_deblock_h_luma_mmx2;
30.                pf->deblock_chroma[1] = x264_deblock_v_chroma_mmx2;
31.                pf->deblock_h_chroma_420 = x264_deblock_h_chroma_mmx2;
32.                pf->deblock_chroma_420_mbafl = x264_deblock_h_chroma_mbafl_mmx2;
33.                pf->deblock_h_chroma_422 = x264_deblock_h_chroma_422_mmx2;
34.                pf->deblock_h_chroma_422_intra = x264_deblock_h_chroma_422_intra_mmx2;
35.                pf->deblock_luma_intra[1] = x264_deblock_v_luma_intra_mmx2;
36.                pf->deblock_luma_intra[0] = x264_deblock_h_luma_intra_mmx2;
37.                pf->deblock_chroma_intra[1] = x264_deblock_v_chroma_intra_mmx2;
38.                pf->deblock_h_chroma_420_intra = x264_deblock_h_chroma_intra_mmx2;
39.                pf->deblock_chroma_420_intra_mbafl = x264_deblock_h_chroma_intra_mbafl_mmx2;
40.            #endif
41.            //此处省略大量的X86、ARM等平台的汇编函数初始化代码
42.        }
43.    }
```

从源代码可以看出，x264_deblock_init()中初始化了一系列环路滤波函数。这些函数名称的规则如下：

- (1) 包含“v”的是垂直滤波器，用于处理水平边界；包含“h”的是水平滤波器，用于处理垂直边界。
- (2) 包含“luma”的是亮度滤波器，包含“chroma”的是色度滤波器。
- (3) 包含“intra”的是处理边界强度Bs为4的强滤波器，不包含“intra”的是普通滤波器。

x264_deblock_init()的输入参数x264_deblock_function_t是一个结构体，其中包含了环路滤波器相关的函数指针。x264_deblock_function_t的定义如下所示。

```
[cpp]
1. typedef struct
2. {
3.     x264_deblock_inter_t deblock_luma[2];
4.     x264_deblock_inter_t deblock_chroma[2];
5.     x264_deblock_inter_t deblock_h_chroma_420;
6.     x264_deblock_inter_t deblock_h_chroma_422;
7.     x264_deblock_intra_t deblock_luma_intra[2];
8.     x264_deblock_intra_t deblock_chroma_intra[2];
9.     x264_deblock_intra_t deblock_h_chroma_420_intra;
10.    x264_deblock_intra_t deblock_h_chroma_422_intra;
11.    x264_deblock_inter_t deblock_luma_mbaff;
12.    x264_deblock_inter_t deblock_chroma_mbaff;
13.    x264_deblock_inter_t deblock_chroma_420_mbaff;
14.    x264_deblock_inter_t deblock_chroma_422_mbaff;
15.    x264_deblock_intra_t deblock_luma_intra_mbaff;
16.    x264_deblock_intra_t deblock_chroma_intra_mbaff;
17.    x264_deblock_intra_t deblock_chroma_420_intra_mbaff;
18.    x264_deblock_intra_t deblock_chroma_422_intra_mbaff;
19.    void (*deblock_strength) ( uint8_t nnz[X264_SCAN8_SIZE], int8_t ref[2][X264_SCAN8_LUMA_SIZE],
20.                             int16_t mv[2][X264_SCAN8_LUMA_SIZE][2], uint8_t bs[2][8][4], int mvy_limit,
21.                             int bframe );
22. } x264_deblock_function_t;
```

x264_deblock_init()的工作就是对x264_deblock_function_t中的函数指针进行赋值。可以看出x264_deblock_function_t中很多的元素是一个包含2个元素的数组，例如deblock_luma[2]，deblock_luma_intra[2]等。这些数组中的元素[0]一般是水平滤波器，而元素[1]是垂直滤波器。下面记录几个最有代表性的滤波函数：普通滤波函数deblock_v_luma_c()和deblock_h_luma_c()，以及强滤波函数deblock_v_luma_intra_c()和deblock_h_luma_intra_c()。

普通滤波函数（Bs=1,2,3）

deblock_v_luma_c()

deblock_v_luma_c()是一个普通强度的垂直滤波器，用于处理边界强度Bs为1，2，3的水平边界。该函数的定义位于common\deblock.c，如下所示。

```
[cpp]
1. //去块效应滤波-普通滤波，Bs为1,2,3
2. //垂直（Vertical）滤波器
3. // 边界
4. //      x
5. //      x
6. // 边界-----
7. //      x
8. //      x
9. //
10. //
11. static void deblock_v_luma_c( pixel *pix, intptr_t stride, int alpha, int beta, int8_t *tc0 )
12. {
13.     //xstride=stride（用于选择滤波的像素）
14.     //ystride=1
15.     deblock_luma_c( pix, stride, 1, alpha, beta, tc0 );
16. }
```

可以看出deblock_v_luma_c()调用了另一个函数deblock_luma_c()。需要注意deblock_luma_c()是一个水平滤波器和垂直滤波器都会调用的“通用”滤波器函数。在这里传递给deblock_luma_c()第二个参数xstride的值为stride，第三个参数ystride的值为1。

deblock_luma_c()

deblock_luma_c()是一个通用的滤波器函数，定义如下所示。

```
[cpp]
1. //去块效应滤波-普通滤波，Bs为1,2,3
2. static inline void deblock_luma_c( pixel *pix, intptr_t xstride, intptr_t ystride, int alpha, int beta, int8_t *tc0 )
3. {
4.     for( int i = 0; i < 4; i++ )
5.     {
6.         if( tc0[i] < 0 )
7.         {
8.             pix += 4*ystride;
9.             continue;
10.        }
11.        //滤4个像素
12.        for( int d = 0; d < 4; d++, pix += ystride )
13.            deblock_edge_luma_c( pix, xstride, alpha, beta, tc0[i] );
14.    }
15. }
```

从源代码中可以看出，具体的滤波在deblock_edge_luma_c()中完成。处理完一个像素后，会继续处理与当前像素距离为ystride的像素。

deblock_edge_luma_c()

deblock_edge_luma_c()用于完成一个点的滤波工作。该函数的定义如下所示。

```

1.  /* From ffmpeg */
2.  //去块效应滤波-普通滤波, Bs为1,2,3
3.  //从FFmpeg复制过来的?
4.  static ALWAYS_INLINE void deblock_edge_luma_c( pixel *pix, intptr_t xstride, int alpha, int beta, int8_t tc0 )
5.  {
6.      //p和q
7.      //如果xstride=stride, ystride=1
8.      //就是处理纵向的6个像素
9.      //对应的是方块的横向边界的滤波, 即如下所示:
10.     //      p2
11.     //      p1
12.     //      p0
13.     //=====图像边界=====
14.     //      q0
15.     //      q1
16.     //      q2
17.     //
18.     //如果xstride=1, ystride=stride
19.     //就是处理纵向的6个像素
20.     //对应的是方块的横向边界的滤波, 即如下所示:
21.     //      ||
22.     // p2 p1 p0 || q0 q1 q2
23.     //      ||
24.     //      边界
25.
26.     //注意: 这里乘的是xstride
27.
28.     int p2 = pix[-3*xstride];
29.     int p1 = pix[-2*xstride];
30.     int p0 = pix[-1*xstride];
31.     int q0 = pix[ 0*xstride];
32.     int q1 = pix[ 1*xstride];
33.     int q2 = pix[ 2*xstride];
34.     //计算方法参考相关的标准
35.     //alpha和beta是用于检查图像内容的2个参数
36.     //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
37.     if( abs( p0 - q0 ) < alpha && abs( p1 - p0 ) < beta && abs( q1 - q0 ) < beta )
38.     {
39.         int tc = tc0;
40.         int delta;
41.         //上面2个点(p0, p2)满足条件的时候, 滤波p1
42.         //int x264_clip3( int v, int i_min, int i_max )用于限幅
43.         if( abs( p2 - p0 ) < beta )
44.         {
45.             if( tc0 )
46.                 pix[-2*xstride] = p1 + x264_clip3( (( p2 + ((p0 + q0 + 1) >> 1)) >> 1) - p1, -tc0, tc0 );
47.             tc++;
48.         }
49.         //下面2个点(q0, q2)满足条件的时候, 滤波q1
50.         if( abs( q2 - q0 ) < beta )
51.         {
52.             if( tc0 )
53.                 pix[ 1*xstride] = q1 + x264_clip3( (( q2 + ((p0 + q0 + 1) >> 1)) >> 1) - q1, -tc0, tc0 );
54.             tc++;
55.         }
56.
57.         delta = x264_clip3( (((q0 - p0) << 2) + (p1 - q1) + 4) >> 3, -tc, tc );
58.         //p0
59.         pix[-1*xstride] = x264_clip_pixel( p0 + delta ); /* p0' */
60.         //q0
61.         pix[ 0*xstride] = x264_clip_pixel( q0 - delta ); /* q0' */
62.     }
63. }

```

从源代码可以看出, deblock_edge_luma_c()实现了前文记录的普通强度的滤波公式。

deblock_h_luma_c()

deblock_h_luma_c()是一个普通强度的水平滤波器, 用于处理边界强度Bs为1, 2, 3的垂直边界。该函数的定义如下所示。

```

1.  //去块效应滤波-普通滤波, Bs为1,2,3
2.  //水平 (Horizontal) 滤波器
3.  //      边界
4.  //      |
5.  // x x x | x x x
6.  //      |
7.  static void deblock_h_luma_c( pixel *pix, intptr_t stride, int alpha, int beta, int8_t *tc0 )
8.  {
9.      //xstride=1 (用于选择滤波的像素)
10.     //ystride=stride
11.     deblock_luma_c( pix, 1, stride, alpha, beta, tc0 );
12. }

```

从源代码可以看出, 和deblock_v_luma_c()类似, deblock_h_luma_c()同样调用了deblock_luma_c()函数。唯一的不同在于它传递给deblock_luma_c()的第2个参数xstride为1, 第3个参数ystride为stride。

强滤波函数（Bs=4）

deblock_v_luma_intra_c()

deblock_v_luma_intra_c()是一个强滤波的垂直滤波器，用于处理边界强度Bs为4的水平边界。该函数的定义位于common\deblock.c，如下所示。

```
[cpp]
1. //垂直 (Vertical) 强滤波器-Bs为4
2. //    边界
3. //        x
4. //        x
5. // 边界-----
6. //        x
7. //        x
8. static void deblock_v_luma_intra_c( pixel *pix, intptr_t stride, int alpha, int beta )
9. {
10.     //注意
11.     //xstride=stride
12.     //ystride=1
13.     //处理完1个像素点之后, pix增加ystride
14.
15.     //水平滤波和垂直滤波通用的强滤波函数
16.     deblock_luma_intra_c( pix, stride, 1, alpha, beta );
17. }
```

可以看出deblock_v_luma_intra_c()调用了另一个函数deblock_luma_intra_c()。需要注意deblock_luma_intra_c()是一个水平滤波器和垂直滤波器都会调用的“通用”滤波器函数。在这里传递给deblock_luma_intra_c()第二个参数xstride的值为stride，第三个参数ystride的值为1。

deblock_luma_intra_c()

deblock_luma_intra_c()是一个通用的滤波器函数，定义如下所示。

```
[cpp]
1. //水平滤波和垂直滤波通用的强滤波函数-Bs为4
2. static inline void deblock_luma_intra_c( pixel *pix, intptr_t xstride, intptr_t ystride, int alpha, int beta )
3. {
4.     //循环处理16个点
5.     //处理完1个像素点之后, pix增加ystride
6.     for( int d = 0; d < 16; d++, pix += ystride )
7.         deblock_edge_luma_intra_c( pix, xstride, alpha, beta );    //每次处理1个点
8. }
```

从源代码中可以看出，具体的滤波在deblock_edge_luma_intra_c()中完成。处理完一个像素后，会继续处理与当前像素距离为ystride的像素。

deblock_edge_luma_intra_c()

deblock_edge_luma_intra_c()用于完成一个点的滤波工作。该函数的定义如下所示。

```

1. //水平滤波和垂直滤波通用的强滤波函数-处理1个点-Bs为4
2. //注意涉及到8个像素
3. static ALWAYS_INLINE void deblock_edge_luma_intra_c( pixel *pix, intptr_t xstride, int alpha, int beta )
4. {
5.     //如果xstride=stride, ystride=1
6.     //就是处理纵向的6个像素
7.     //对应的是方块的横向边界的滤波。如下所示：
8.     //      p2
9.     //      p1
10.    //      p0
11.    //=====图像边界=====
12.    //      q0
13.    //      q1
14.    //      q2
15.    //
16.    //如果xstride=1, ystride=stride
17.    //就是处理纵向的6个像素
18.    //对应的是方块的横向边界的滤波，即如下所示：
19.    //      ||
20.    // p2 p1 p0 || q0 q1 q2
21.    //      ||
22.    //      边界
23.
24.    //注意：这里乘的是xstride
25.    int p2 = pix[-3*xstride];
26.    int p1 = pix[-2*xstride];
27.    int p0 = pix[-1*xstride];
28.    int q0 = pix[ 0*xstride];
29.    int q1 = pix[ 1*xstride];
30.    int q2 = pix[ 2*xstride];
31.    //满足条件的时候，才滤波
32.    if( abs( p0 - q0 ) < alpha && abs( p1 - p0 ) < beta && abs( q1 - q0 ) < beta )
33.    {
34.        if( abs( p0 - q0 ) < ((alpha >> 2) + 2) )
35.        {
36.            if( abs( p2 - p0 ) < beta ) /* p0', p1', p2' */
37.            {
38.                const int p3 = pix[-4*xstride];
39.                pix[-1*xstride] = ( p2 + 2*p1 + 2*p0 + 2*q0 + q1 + 4 ) >> 3;
40.                pix[-2*xstride] = ( p2 + p1 + p0 + q0 + 2 ) >> 2;
41.                pix[-3*xstride] = ( 2*p3 + 3*p2 + p1 + p0 + q0 + 4 ) >> 3;
42.            }
43.            else /* p0' */
44.                pix[-1*xstride] = ( 2*p1 + p0 + q1 + 2 ) >> 2;
45.            if( abs( q2 - q0 ) < beta ) /* q0', q1', q2' */
46.            {
47.                const int q3 = pix[3*xstride];
48.                pix[0*xstride] = ( p1 + 2*p0 + 2*q0 + 2*q1 + q2 + 4 ) >> 3;
49.                pix[1*xstride] = ( p0 + q0 + q1 + q2 + 2 ) >> 2;
50.                pix[2*xstride] = ( 2*q3 + 3*q2 + q1 + q0 + p0 + 4 ) >> 3;
51.            }
52.            else /* q0' */
53.                pix[0*xstride] = ( 2*q1 + q0 + p1 + 2 ) >> 2;
54.        }
55.        else /* p0', q0' */
56.        {
57.            pix[-1*xstride] = ( 2*p1 + p0 + q1 + 2 ) >> 2;
58.            pix[ 0*xstride] = ( 2*q1 + q0 + p1 + 2 ) >> 2;
59.        }
60.    }
61. }

```

从源代码可以看出，deblock_edge_luma_intra_c()实现了前文记录的强滤波公式。

至此有关环路滤波的源代码就分析完毕了。

x264_frame_filter()

x264_frame_filter()用于完成半像素内插的工作。该函数的定义位于common\mc.c，如下所示。

```

1. //半像素内插
2. void x264_frame_filter( x264_t *h, x264_frame_t *frame, int mb_y, int b_end )
3. {
4.     const int b_interlaced = PARAM_INTERLACED;
5.     int start = mb_y*16 - 8; // buffer = 4 for deblock + 3 for 6tap, rounded to 8
6.     int height = (b_end ? frame->i_lines[0] + 16*PARAM_INTERLACED : (mb_y+b_interlaced)*16) + 8;
7.
8.     if( mb_y & b_interlaced )
9.         return;
10.
11.     for( int p = 0; p < (CHROMA444 ? 3 : 1); p++ )
12.     {
13.         int stride = frame->i_stride[p];
14.         const int width = frame->i_width[p];
15.         int offs = start*stride - 8; // buffer = 3 for 6tap, aligned to 8 for simd
16.         //半像素内插
17.         if( !b_interlaced || h->mb.b_adaptive_mbaff )
18.             h->mc.hpel_filter(
19.                 frame->filtered[p][1] + offs, //水平半像素内插
20.                 frame->filtered[p][2] + offs, //垂直半像素内插
21.                 frame->filtered[p][3] + offs, //中间半像素内插
22.                 frame->plane[p] + offs,
23.                 stride, width + 16, height - start,
24.                 h->scratch_buffer );
25.
26.         if( b_interlaced )
27.         {
28.             /* MC must happen between pixels in the same field. */
29.             stride = frame->i_stride[p] << 1;
30.             start = (mb_y*16 >> 1) - 8;
31.             int height_fld = ((b_end ? frame->i_lines[p] : mb_y*16) >> 1) + 8;
32.             offs = start*stride - 8;
33.             for( int i = 0; i < 2; i++, offs += frame->i_stride[p] )
34.             {
35.                 h->mc.hpel_filter(
36.                     frame->filtered_fld[p][1] + offs,
37.                     frame->filtered_fld[p][2] + offs,
38.                     frame->filtered_fld[p][3] + offs,
39.                     frame->plane_fld[p] + offs,
40.                     stride, width + 16, height_fld - start,
41.                     h->scratch_buffer );
42.             }
43.         }
44.     }
45.
46.     /* generate integral image:
47.      * frame->integral contains 2 planes. in the upper plane, each element is
48.      * the sum of an 8x8 pixel region with top-left corner on that point.
49.      * in the lower plane, 4x4 sums (needed only with --partitions p4x4). */
50.
51.     if( frame->integral )
52.     {
53.         int stride = frame->i_stride[0];
54.         if( start < 0 )
55.         {
56.             memset( frame->integral - PADV * stride - PADH, 0, stride * sizeof(uint16_t) );
57.             start = -PADV;
58.         }
59.         if( b_end )
60.             height += PADV-9;
61.         for( int y = start; y < height; y++ )
62.         {
63.             pixel *pix = frame->plane[0] + y * stride - PADH;
64.             uint16_t *sum8 = frame->integral + (y+1) * stride - PADH;
65.             uint16_t *sum4;
66.             if( h->frames.b_have_sub8x8_esa )
67.             {
68.                 h->mc.integral_init4h( sum8, pix, stride );
69.                 sum8 -= 8*stride;
70.                 sum4 = sum8 + stride * (frame->i_lines[0] + PADV*2);
71.                 if( y >= 8-PADV )
72.                     h->mc.integral_init4v( sum8, sum4, stride );
73.             }
74.             else
75.             {
76.                 h->mc.integral_init8h( sum8, pix, stride );
77.                 if( y >= 8-PADV )
78.                     h->mc.integral_init8v( sum8-8*stride, stride );
79.             }
80.         }
81.     }
82. }

```

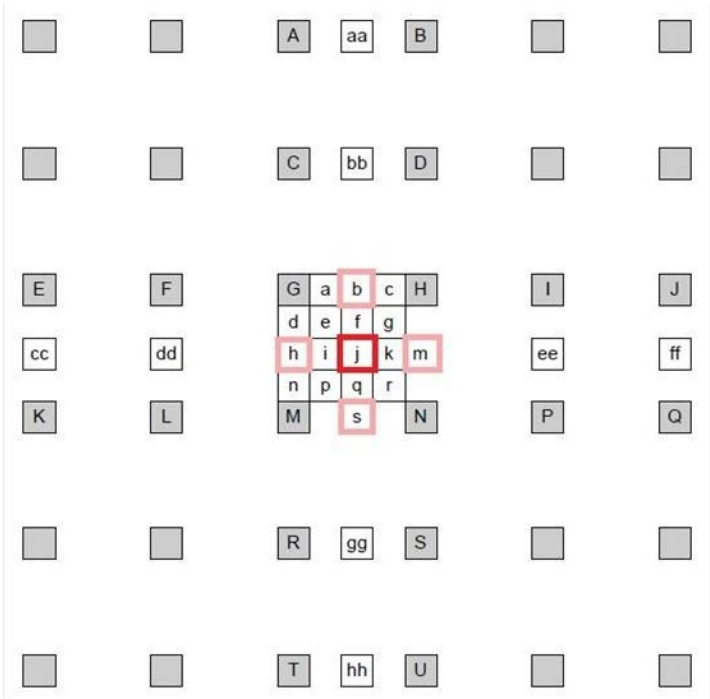
从源代码中可以看出，x264_frame_filter()调用了汇编函数h->mc.hpel_filter()完成了半像素内插的工作。经过汇编半像素内插函数处理之后，得到的水平半像素内差点存储在x264_frame_t的filtered[][1]中，垂直半像素内差点存储在x264_frame_t的filtered[][2]中，对角线半像素内差点存储在x264_frame_t的filtered[][3]中（整像素点存储在x264_frame_t的filtered[][0]中）。

下文开始分析半像素内插模块调用的汇编函数。

1/4像素内插小知识

(1) 半像素内插

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



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

如图所示，1/4像素内插一般分成两步：

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

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

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

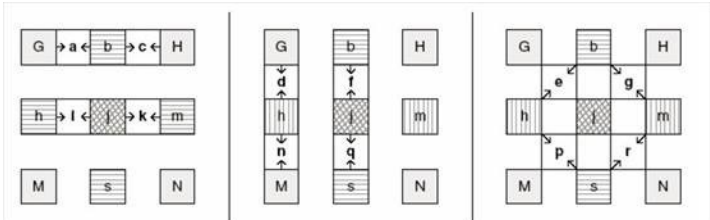
剩下几个半像素点的计算关系如下：

- m：由B、D、H、N、S、U计算
- h：由A、C、G、M、R、T计算
- s：由K、L、M、N、P、Q计算
- j：由cc、dd、h、m、ee、ff计算。需要注意j点的运算量比较大，因为cc、dd、ee、ff都需要通过半像素内插方法进行计算。

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

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

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



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

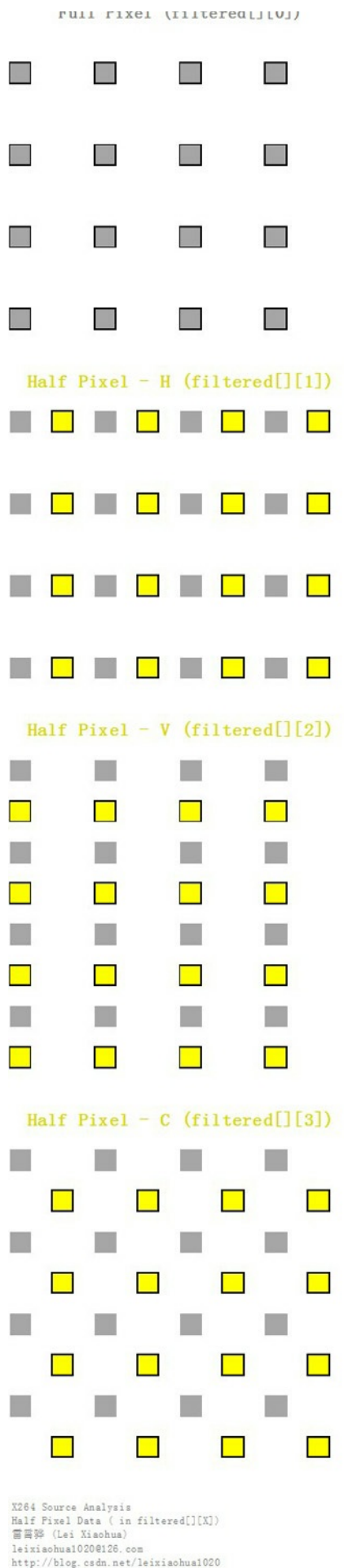
(2) 半像素点实例

下图显示了一个4x4图像块经过半像素内插处理后，得到的半像素与整像素点之间的位置关系。

Half Pixel Data (4x4)

- Full pixel
- Half Pixel

Full Pixel (64x64) / 4 = 16x16

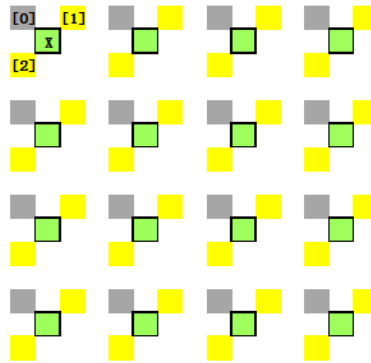


(3) 1/4像素内插

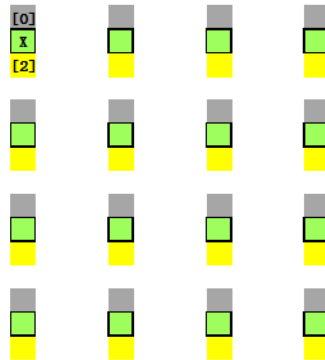
1/4像素内插点是通过半像素点之间（或者和整像素点）线性内插获得。下图显示了一个4x4图像块进行1/4像素内插的过程。上面一张图中水平半像素点（存储于filter[][1]）和垂直半像素点（存储于filter[][2]）线性内插后得到了绿色的1/4像素内插点X。下面一张图中整像素点（存储于filter[][0]）和垂直半像素点（存储于filter[][2]）线性内插后得到了绿色的1/4像素内插点X。

Quarter Pixel Data (4x4)

Use Half-H and Half-V ([1] and [2])
 $X = ([1] + [2]) / 2$



Use Full and Half-V ([0] and [2])
 $X = ([0] + [2]) / 2$



X264 Source Analysis
Quarter Pixel Data (4x4 block)
雷霄骅 (Lei Xiaohua)
leixiaohua1020@126.com
<http://blog.csdn.net/leixiaohua1020>

x264_mc_init()

x264_mc_init()用于初始化运动补偿相关的汇编函数。该函数的定义位于common\mc.c，如下所示。

```

1. //运动补偿
2. void x264_mc_init( int cpu, x264_mc_functions_t *pf, int cpu_independent )
3. {
4.     //亮度运动补偿
5.     pf->mc_luma = mc_luma;
6.     //获得匹配块
7.     pf->get_ref = get_ref;
8.
9.     pf->mc_chroma = mc_chroma;
10.    //求平均
11.    pf->avg[PIXEL_16x16]= pixel_avg_16x16;
12.    pf->avg[PIXEL_16x8] = pixel_avg_16x8;
13.    pf->avg[PIXEL_8x16] = pixel_avg_8x16;
14.    pf->avg[PIXEL_8x8]  = pixel_avg_8x8;
15.    pf->avg[PIXEL_8x4]  = pixel_avg_8x4;
16.    pf->avg[PIXEL_4x16] = pixel_avg_4x16;
17.    pf->avg[PIXEL_4x8]  = pixel_avg_4x8;
18.    pf->avg[PIXEL_4x4]  = pixel_avg_4x4;
19.    pf->avg[PIXEL_4x2]  = pixel_avg_4x2;
20.    pf->avg[PIXEL_2x8]  = pixel_avg_2x8;
21.    pf->avg[PIXEL_2x4]  = pixel_avg_2x4;
22.    pf->avg[PIXEL_2x2]  = pixel_avg_2x2;
23.    //加权相关
24.    pf->weight = x264_mc_weight_wtab;
25.    pf->offsetadd = x264_mc_weight_wtab;
26.    pf->offsetsub = x264_mc_weight_wtab;
27.    pf->weight_cache = x264_weight_cache;
28.    //赋值-只包含了方形的
29.    pf->copy_16x16_unaligned = mc_copy_w16;
30.    pf->copy[PIXEL_16x16] = mc_copy_w16;
31.    pf->copy[PIXEL_8x8]   = mc_copy_w8;
32.    pf->copy[PIXEL_4x4]   = mc_copy_w4;
33.
34.    pf->store_interleave_chroma = store_interleave_chroma;
35.    pf->load_deinterleave_chroma_fenc = load_deinterleave_chroma_fenc;
36.    pf->load_deinterleave_chroma_fdec = load_deinterleave_chroma_fdec;
37.    //拷贝像素-不论像素块大小
38.    pf->plane_copy = x264_plane_copy_c;
39.    pf->plane_copy_interleave = x264_plane_copy_interleave_c;
40.    pf->plane_copy_deinterleave = x264_plane_copy_deinterleave_c;
41.    pf->plane_copy_deinterleave_rgb = x264_plane_copy_deinterleave_rgb_c;
42.    pf->plane_copy_deinterleave_v210 = x264_plane_copy_deinterleave_v210_c;
43.    //关键：半像素内插
44.    pf->hpel_filter = hpel_filter;
45.    //几个空函数
46.    pf->prefetch_fenc_420 = prefetch_fenc_null;
47.    pf->prefetch_fenc_422 = prefetch_fenc_null;
48.    pf->prefetch_ref = prefetch_ref_null;
49.    pf->memcpy_aligned = memcpy;
50.    pf->memzero_aligned = memzero_aligned;
51.    //降低分辨率-线性内插（不是半像素内插）
52.    pf->frame_init_lowres_core = frame_init_lowres_core;
53.
54.    pf->integral_init4h = integral_init4h;
55.    pf->integral_init8h = integral_init8h;
56.    pf->integral_init4v = integral_init4v;
57.    pf->integral_init8v = integral_init8v;
58.
59.    pf->mbtree_propagate_cost = mbtree_propagate_cost;
60.    pf->mbtree_propagate_list = mbtree_propagate_list;
61.    //各种汇编版本
62.    #if HAVE_MMX
63.        x264_mc_init_mmx( cpu, pf );
64.    #endif
65.    #if HAVE_ALTIVEC
66.        if( cpu & X264_CPU_ALTIVEC )
67.            x264_mc_altivec_init( pf );
68.    #endif
69.    #if HAVE_ARMV6
70.        x264_mc_init_arm( cpu, pf );
71.    #endif
72.    #if ARCH_AARCH64
73.        x264_mc_init_aarch64( cpu, pf );
74.    #endif
75.
76.    if( cpu_independent )
77.    {
78.        pf->mbtree_propagate_cost = mbtree_propagate_cost;
79.        pf->mbtree_propagate_list = mbtree_propagate_list;
80.    }
81. }

```

从源代码可以看出，x264_mc_init()中包含了大量的像素内插、拷贝、求平均的函数。这些函数都是用于在H.264编码过程中进行运动估计和运动补偿的。其中半像素内插函数是hpel_filter()。

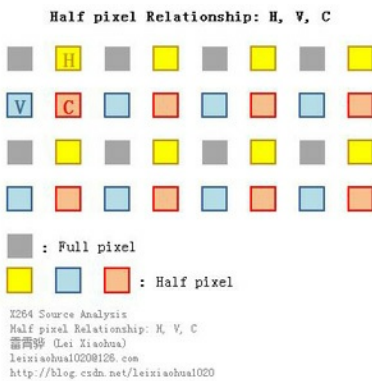
hpel_filter()

hpel_filter()用于进行半像素插值。该函数的定义位于common\mc.c，如下所示。

[cpp]  

```
1. //半像素插值公式
2. //b= (E - 5F + 20G + 20H - 5I + J)/32
3. //          x
4. //d取1, 水平滤波器; d取stride, 垂直滤波器 (这里没有除以32)
5. #define TAPFILTER(pix, d) ((pix)[x-2*d] + (pix)[x+3*d] - 5*((pix)[x-d] + (pix)[x+2*d]) + 20*((pix)[x] + (pix)[x+d]))
6.
7. /*
8.  * 半像素插值
9.  * dsth: 水平滤波得到的半像素点(aa,bb,b,s,gg,hh)
10.  * dstv: 垂直滤波得到的半像素点(cc,dd,h,m,ee,ff)
11.  * dstc: “水平+垂直”滤波得到的位于4个像素中间的半像素点 (j)
12.  *
13.  * 半像素插值示意图如下:
14.  *
15.  *      A aa B
16.  *
17.  *      C bb D
18.  *
19.  * E  F  G  b H  I  J
20.  *
21.  * cc dd  h  j m  ee ff
22.  *
23.  * K  L  M  s N  P  Q
24.  *
25.  *      R gg S
26.  *
27.  *      T hh U
28.  *
29.  * 计算公式如下:
30.  * b=round( (E - 5F + 20G + 20H - 5I + J ) / 32)
31.  *
32.  * 剩下几个半像素点的计算关系如下:
33.  * m: 由B、D、H、N、S、U计算
34.  * h: 由A、C、G、M、R、T计算
35.  * s: 由K、L、M、N、P、Q计算
36.  * j: 由cc、dd、h、m、ee、ff计算。需要注意j点的运算量比较大, 因为cc、dd、ee、ff都需要通过半像素内插方法进行计算。
37.  *
38.  */
39. static void hpel_filter( pixel *dsth, pixel *dstv, pixel *dstc, pixel *src,
40.                          intptr_t stride, int width, int height, int16_t *buf )
41. {
42.     const int pad = (BIT_DEPTH > 9) ? (-10 * PIXEL_MAX) : 0;
43.     /*
44.      * 几种半像素点之间的位置关系
45.      *
46.      * X: 像素点
47.      * H: 水平滤波半像素点
48.      * V: 垂直滤波半像素点
49.      * C: 中间位置半像素点
50.      *
51.      * X  H  X      X      X
52.      *
53.      * V  C
54.      *
55.      * X      X      X      X
56.      *
57.      *
58.      *
59.      * X      X      X      X
60.      *
61.      */
62.     //一行一行处理
63.     for( int y = 0; y < height; y++ )
64.     {
65.         //一个一个点处理
66.         //每个整数点都对应h, v, c三个半像素点
67.         //v
68.         for( int x = -2; x < width+3; x++ )//(aa,bb,b,s,gg,hh),结果存入buf
69.         {
70.             //垂直滤波半像素点
71.             int v = TAPFILTER(src,stride);
72.             dstv[x] = x264_clip_pixel( (v + 16) >> 5 );
73.             /* transform v for storage in a 16-bit integer */
74.             //这应该是给dstc计算使用的?
75.             buf[x+2] = v + pad;
76.         }
77.         //c
78.         for( int x = 0; x < width; x++ )
79.             dstc[x] = x264_clip_pixel( (TAPFILTER(buf+2,1) - 32*pad + 512) >> 10 );//四个相邻像素中间的半像素点
80.         //h
81.         for( int x = 0; x < width; x++ )
82.             dsth[x] = x264_clip_pixel( (TAPFILTER(src,1) + 16) >> 5 );//水平滤波半像素点
83.         dsth += stride;
84.         dstv += stride;
85.         dstc += stride;
86.         src += stride;
87.     }
88. }
```

从源代码可以看出，hpel_filter()中包含了一个宏TAPFILTER()用来完成半像素点像素值的计算。在完成半像素插值工作后，dsth中存储的是经过水平插值后的半像素点，dstv中存储的是经过垂直插值后的半像素点，dstc中存储的是位于4个相邻像素点中间位置的半像素点。这三块内存中的点的位置关系如下图所示（灰色的点是整像素点）。



视频质量计算-PSNR和SSIM

X264中支持两种视频质量计算方法：PSNR和SSIM。这两种的方法都是在x264_fdec_filter_row()中计算完成的。其中PSNR在此只计算了SSD，在编码一帧结束之后的x264_encoder_frame_end()中，调用x264_psnr()完成计算。

视频质量评价的知识

PSNR知识

PSNR（Peak Signal to Noise Ratio，峰值信噪比）是最基础的视频质量评价方法。它的取值一般在20-50之间，值越大代表受损图片越接近原图片。PSNR通过对原始图像和失真图像进行像素的逐点对比，计算两幅图像像素点之间的误差，并由这些误差最终确定失真图像的质量评分。该方法由于计算简便、数学意义明确，在图像处理领域中应用最为广泛。

一幅MxN尺寸的图像的PSNR的计算公式如下所示：

$$\text{MSE} = \frac{1}{M \times N} \sum_{i=1}^M \sum_{j=1}^N (x_{ij} - y_{ij})^2$$
$$\text{PSNR} = 10 \lg \left(\frac{L^2}{\text{MSE}} \right)$$

其中x_{ij} 和y_{ij} 分别表示失真图像和原始图像对应像素点的灰度值;i,j 分别代表图像的行和列;L 是图像灰度值可达到的动态范围,8位的灰度图像的L=2⁸-1=255。如果已知SSD，MxN尺寸图像的PSNR公式如下所示。

$$\text{MSE} = \text{SSD} * 1 / (M * N)$$
$$\text{PSNR} = 10 * \lg(255^2 / \text{MSE})$$

但是PSNR仅仅计算了图像像素点间的绝对误差，没有考虑像素点间的视觉相关性，更没顾及人类视觉系统的感知特性，所以其评价结果与主观感受往往相差较大。例如下图两张图片的PSNR取值都在23.6左右，但是给人的感觉却是（a）图比（b）图清晰得多。



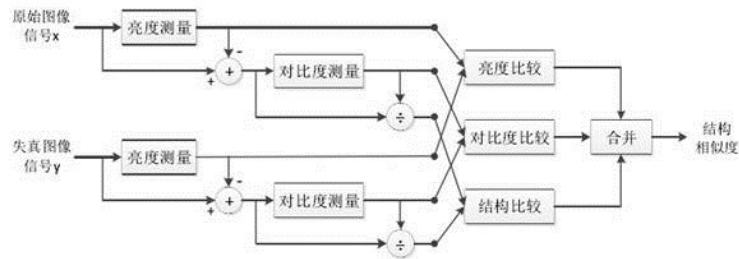
（a）PSNR = 23.681 高斯白噪声失真 （b）PSNR = 23.621 高斯模糊失真

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

正是由于PSNR方法存在上述的问题，人们才开始研究与人类视觉系统特性相关的质量评价方法。SSIM就是一种典型的与人类视觉系统特性结合的质量评价方法。

SSIM知识

SSIM（Structural SIMilarity，结构相似度）是一种结合了亮度信息，对比度信息以及结构信息的视频质量评价方法。它的取值在0-1之间，值越大代表受损图片越接近原图片。该方法的模型图如下所示。



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

从模型图可以看出，SSIM 评价方法中的结构相似度由三个层次的结构信息共同决定。首先假设 x 、 y 分别是原始图像信号和失真图像信号，然后分别计算这两个信号的亮度比较函数 $l(x,y)$ 、对比度比较函数 $c(x,y)$ 以及结构比较函数 $s(x,y)$ ，最后经过加权合并计算得出图像结构相似度评价结果。这3个比较函数具体的公式如下所示。

(1) 亮度比较函数 $l(x,y)$

亮度均值 μ_x 如下所示。

$$\mu_x = \frac{1}{N} \sum_{i=1}^N x_i$$

亮度比较函数的公式如下所示。其中 C_1 为常量。

$$l(x, y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}$$

(2) 对比度比较函数 $c(x,y)$

亮度标准差 σ_x 如下所示。

$$\sigma_x = \left(\frac{1}{N-1} \sum_{i=1}^N (x_i - \mu_x)^2 \right)^{1/2}$$

对比度比较函数的公式如下所示。其中 C_2 为常量。

$$c(x, y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}$$

(3) 结构比较函数 $s(x,y)$

两个图像信号的相关系数 σ_{xy} 如下所示。

$$\sigma_{xy} = \frac{1}{N-1} \sum_{i=1}^N (x_i - \mu_x)(y_i - \mu_y)$$

结构比较函数定义如下所示。其中 C_3 为常量。

$$s(x, y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}$$

SSIM就是将上述三个公式相乘，公式如下所示。

$$SSIM(x, y) = [l(x, y)]^\alpha \cdot [c(x, y)]^\beta \cdot [s(x, y)]^\gamma$$

为了便于计算，将 α 、 β 、 γ 的值都设为 1，并且令 $C_3 = C_2/2$ ，则上式的简化为下式。

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}$$

实际经验中，对整幅图像直接使用 SSIM 模型，不如局部分块使用最后综合的效果好。因此SSIM的计算都是按照一个小一个小方块（例如8x8这种的方块）进行计算的。

PS：有关PSNR和SSIM和人眼主观感受之间的关系可以参考文章《[全参考视频质量评价方法（PSNR，SSIM）以及相关数据库](#)》

视频质量评价的源代码

X264中计算PSNR使用了两个函数：x264_pixel_ssd_wxh()和x264_psnr()；而计算SSIM使用了一个函数x264_pixel_ssim_wxh()。

x264_pixel_ssd_wxh()

x264_pixel_ssd_wxh()用于计算SSD（用于以后计算PSNR）。该函数的定义位于common\pixel.c，如下所示。

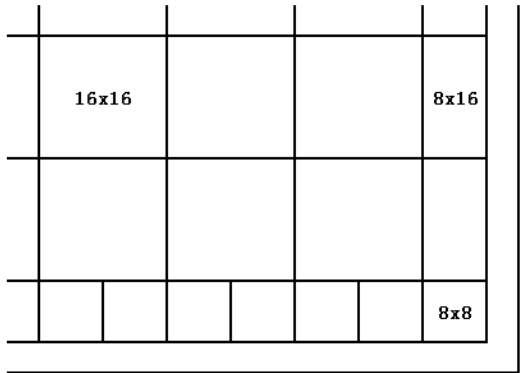

```

1.  /*
2.   * 计算SSD（可用于计算PSNR）
3.   * pix1: 受损数据
4.   * pix2: 原始数据
5.   * i_width: 图像宽
6.   * i_height: 图像高
7.   */
8.  uint64_t x264_pixel_ssd_wxh( x264_pixel_function_t *pf, pixel *pix1, intptr_t i_pix1,
9.                               pixel *pix2, intptr_t i_pix2, int i_width, int i_height )
10. {
11.     //计算结果都累加到i_ssd变量上
12.     uint64_t i_ssd = 0;
13.     int y;
14.     int align = !(((intptr_t)pix1 | (intptr_t)pix2 | i_pix1 | i_pix2) & 15);
15.
16.     #define SSD(size) i_ssd += pf->ssd[size]( pix1 + y*i_pix1 + x, i_pix1, \
17.                                              pix2 + y*i_pix2 + x, i_pix2 );
18.
19.
20.     /*
21.      * SSD计算过程：
22.      * 从左上角开始，绝大部分块使用16x16的SSD计算
23.      * 右边边界部分可能用16x8的SSD计算
24.      * 下边边界可能用8x8的SSD计算
25.      * 注意：这么做主要是出于汇编优化的考虑
26.      *
27.      * +-----+-----+-----+-----+-----+-----+
28.      * |               |               |               |
29.      * +               +               +               +
30.      * |               |               |               |
31.      * +      16x16      +      16x16      +      8x16      +
32.      * |               |               |               |
33.      * +               +               +               +
34.      * |               |               |               |
35.      * +-----+-----+-----+-----+-----+-----+
36.      * |               |
37.      * +      8x8      +
38.      * |               |
39.      * +-----+-----+
40.      * +               +
41.      */
42.     for( y = 0; y < i_height-15; y += 16 )
43.     {
44.         int x = 0;
45.         //大部分使用16x16的SSD
46.         if( align )
47.             for( ; x < i_width-15; x += 16 )
48.                 SSD(PIXEL_16x16);           //i_ssd += pf->ssd[PIXEL_16x16]();
49.         //右边边界部分可能用8x16的SSD
50.         for( ; x < i_width-7; x += 8 )
51.             SSD(PIXEL_8x16);               //i_ssd += pf->ssd[PIXEL_8x16]();
52.     }
53.     //下边边界部分可能用到8x8的SSD
54.     if( y < i_height-7 )
55.         for( int x = 0; x < i_width-7; x += 8 )
56.             SSD(PIXEL_8x8);               //i_ssd += pf->ssd[PIXEL_8x8]();
57.     #undef SSD
58.
59.     #define SSD1 { int d = pix1[y*i_pix1+x] - pix2[y*i_pix2+x]; i_ssd += d*d; }
60.
61.     //如果像素不是16/8的整数倍，边界上的点需要单独算
62.     if( i_width & 7 )
63.     {
64.         for( y = 0; y < (i_height & ~7); y++ )
65.             for( int x = i_width & ~7; x < i_width; x++ )
66.                 SSD1;
67.     }
68.     if( i_height & 7 )
69.     {
70.         for( y = i_height & ~7; y < i_height; y++ )
71.             for( int x = 0; x < i_width; x++ )
72.                 SSD1;
73.     }
74.     #undef SSD1
75.
76.     return i_ssd;
77. }

```

从源代码可以看出，x264_pixel_ssd_wxh()在计算大部分块的SSD的时候是以16x16的块为单位；当宽度不是16的整数倍的时候，在左侧边缘处不足16像素的地方使用了8x16的块进行计算；当高度不是16的整数倍的时候，在下方不足16像素的地方使用了8x8的块进行计算；当宽高不是8的整数倍的时候，则再单独计算。计算方法示意图如下所示。

SSD Calculation (PSNR)



X264 Source Analysis
SSD Calculation (PSNR)
雷霄骅 (Lei Xiaohua)
leixiaohua1020@126.com
<http://blog.csdn.net/leixiaohua1020>

源代码中计算16x16块的SSD的宏“SSD(PIXEL_16x16)”展开的结果如下所示。

```
[cpp]    
1. i_ssd += pf->ssd[PIXEL_16x16]( pix1 + y*i_pix1 + x, i_pix1, pix2 + y*i_pix2 + x, i_pix2 );
```

而pf->ssd[PIXEL_16x16]()指向的C语言版本的SSD计算函数为x264_pixel_ssd_16x16()。

x264_pixel_ssd_16x16()

x264_pixel_ssd_16x16()用于计算16x16的两个像素块的SSD。它的源代码如下所示。

```
[cpp]    
1. static int x264_pixel_ssd_16x16( pixel *pix1, intptr_t i_stride_pix1,  
2. pixel *pix2, intptr_t i_stride_pix2 )  
3. {  
4.     int i_sum = 0;  
5.     for( int y = 0; y < 16; y++ )  
6.     {  
7.         for( int x = 0; x < 16; x++ )  
8.         {  
9.             int d = pix1[x] - pix2[x];  
10.            i_sum += d*d;  
11.        }  
12.        pix1 += i_stride_pix1;  
13.        pix2 += i_stride_pix2;  
14.    }  
15.    return i_sum;  
16. }
```

从源代码可以看出，x264_pixel_ssd_16x16()将两个16x16块的对应点相减之后求平方，然后累加。其他尺寸的块的计算也是类似的，再看一个4x4块的例子。

x264_pixel_ssd_4x4()

x264_pixel_ssd_4x4()用于计算4x4的两个像素块的SSD。它的源代码如下所示。

```
[cpp]    
1. static int x264_pixel_ssd_4x4( pixel *pix1, intptr_t i_stride_pix1,  
2. pixel *pix2, intptr_t i_stride_pix2 )  
3. {  
4.     int i_sum = 0;  
5.     for( int y = 0; y < 4; y++ ) //4个像素  
6.     {  
7.         for( int x = 0; x < 4; x++ ) //4个像素  
8.         {  
9.             int d = pix1[x] - pix2[x]; //相减  
10.            i_sum += d*d; //平方之后，累加  
11.        }  
12.        pix1 += i_stride_pix1;  
13.        pix2 += i_stride_pix2;  
14.    }  
15.    return i_sum;  
16. }
```

可以看出4x4的块和16x16的块的计算方法是类似的，不再重复叙述。在计算完一幅图片的SSD之后，就可以将该值换算成为PSNR了。将SSD换算成PSNR的函数并不在滤波函数x264_fdec_filter_row()中，而是在x264_slice_write()执行完成之后的x264_encoder_frame_end()函数中。

x264_encoder_frame_end()中的x264_psnr()

x264_encoder_frame_end()中的x264_psnr()用于将SSD换算成为PSNR，该函数的定义如下所示。

```
[cpp]
1. //通过SSD换算PSNR
2. static double x264_psnr( double sqe, double size )
3. {
4.     /**
5.      * 计算PSNR的过程
6.      *
7.      * MSE = SSD*1/(w*h)
8.      * PSNR= 10*log10(MAX^2/MSE)
9.      *
10.     * 其中MAX指的是图像的灰度级，对于8bit来说就是2^8-1=255
11.     */
12.     //PIXEL_MAX=255
13.     double mse = sqe / (PIXEL_MAX*PIXEL_MAX * size);
14.     if( mse <= 0.0000000001 ) /* Max 100dB */
15.         return 100;
16.     //MSE转换为PSNR
17.     return -10.0 * log10( mse );
18. }
```

从源代码中可以看出，x264_psnr()实现了上文中提到的MxN尺寸图像的PSNR计算公式：

$$\begin{aligned} \text{MSE} &= \text{SSD} * 1 / (M * N) \\ \text{PSNR} &= 10 * \lg(255^2 / \text{MSE}) \end{aligned}$$

PS：实现过程看上去有点不同，实际上是一样的。

x264_pixel_ssim_wxh()

x264_pixel_ssim_wxh()用于计算SSIM。该函数的定义位于common/pixel.c，如下所示。

```
[cpp]
1. /*
2.  * 计算SSIM
3.  * pix1: 受损数据
4.  * pix2: 原始数据
5.  * i_width: 图像宽
6.  * i_height: 图像高
7.  */
8. float x264_pixel_ssim_wxh( x264_pixel_function_t *pf,
9.                             pixel *pix1, intptr_t stride1,
10.                             pixel *pix2, intptr_t stride2,
11.                             int width, int height, void *buf, int *cnt )
12. {
13.     /*
14.      * SSIM公式
15.      * SSIM = ((2*ux*uy+C1)(2*oxy+C2))/((ux^2+uy^2+C1)(ox^2+oy^2+C2))
16.      *
17.      * 其中
18.      * ux=E(x)
19.      * uy=E(y)
20.      * oxy=cov(x,y)=E(XY)-ux*uy
21.      * ox^2=E(x^2)-E(x)^2
22.      *
23.      */
24.     int z = 0;
25.     float ssim = 0.0;
26.     //这是数组指针，注意和指针数组的区别
27.     //数组指针就是指向数组的指针
28.     int (*sum0)[4] = buf;
29.     /*
30.      * sum0是一个数组指针，其中存储了一个4元素数组的地址
31.      * 换句话说，sum0[]中每一个元素对应一个4x4块的信息（该信息包含4个元素）。
32.      *
33.      * 4个元素中：
34.      * [0]原始像素之和
35.      * [1]受损像素之和
36.      * [2]原始像素平方之和+受损像素平方之和
37.      * [3]原始像素*受损像素的值的和
38.      *
39.      */
40.     int (*sum1)[4] = sum0 + (width >> 2) + 3;
41.     //除以4，编程以“4x4块”为单位
42.     width >>= 2;
43.     height >>= 2;
44.     //以8*8的块为单位计算SSIM值。然后以4个像素为step滑动窗口
45.     for( int y = 1; y < height; y++ )
46.     {
47.         //下面这个循环，只有在第一次执行的时候执行2次，处理第1行和第2行的块
48.         //后面的都只会执行一次
49.         for( ; z <= y; z++ )
```

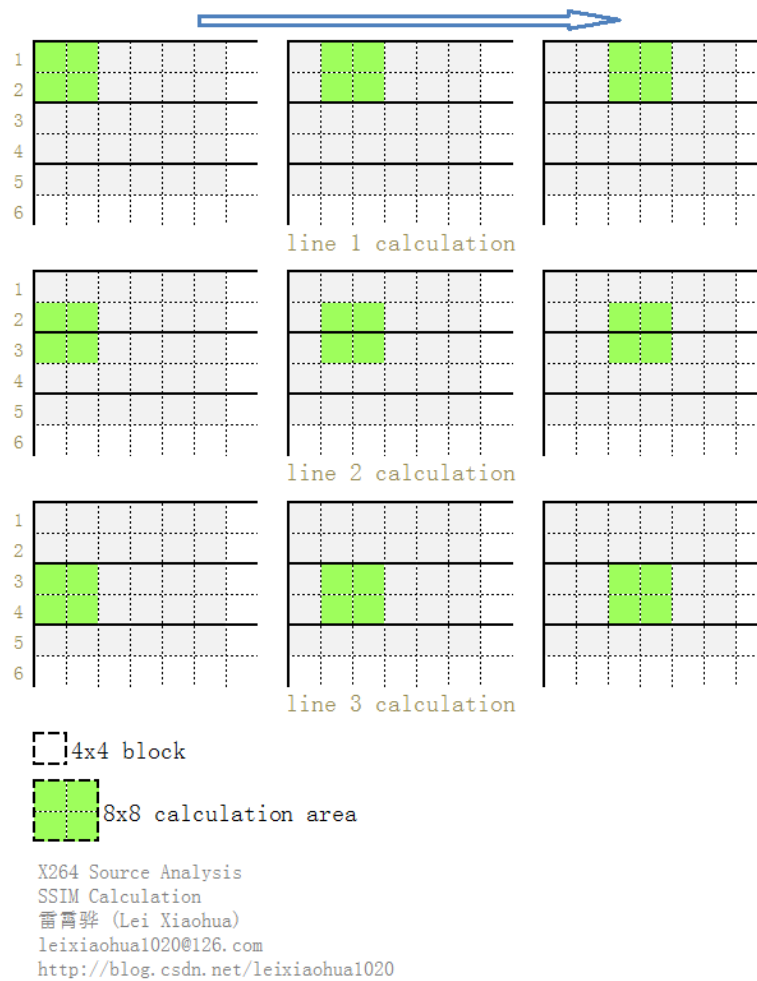
```

50.     {
51.         //执行完XCHG()之后, sum1[]存储上1行块的值 (在上面), 而sum0[]等待ssim_4x4x2_core()计算当前行的值 (在下面)
52.         XCHG( void*, sum0, sum1 );
53.         //获取4x4块的信息 (这里并没有代入公式计算SSIM结果)
54.         //结果存储在sum0[]中。从左到右每个4x4的块依次存储在sum0[0], sum0[1], sum0[2]...
55.         //每次x前进2个块
56.         /*
57.          * ssim_4x4x2_core() : 计算2个4x4块
58.          * +---+---+
59.          * |   |   |
60.          * +---+---+
61.          */
62.         for( int x = 0; x < width; x+=2 )
63.             pf->ssim_4x4x2_core( &pixl[4*(x+z*stride1)], stride1, &pix2[4*(x+z*stride2)], stride2, &sum0[x] );
64.     }
65.     //x每次增加4, 前进4个块
66.     //以8*8的块为单位计算
67.     /*
68.      * sum1[]为上一行4x4块信息, sum0[]为当前行4x4块信息
69.      * 示例 (line以4x4块为单位)
70.      * 第1次运行
71.      *      +---+---+---+---+
72.      * 1line |   sum1[]
73.      *      +---+---+---+---+
74.      * 2line |   sum0[]
75.      *      +---+---+---+---+
76.      *
77.      * 第2次运行
78.      *      +
79.      * 1line |
80.      *      +---+---+---+---+
81.      * 2line |   sum1[]
82.      *      +---+---+---+---+
83.      * 3line |   sum0[]
84.      *      +---+---+---+---+
85.      */
86.     for( int x = 0; x < width-1; x += 4 )
87.         ssim += pf->ssim_end4( sum0+x, sum1+x, X264_MIN(4,width-x-1) );//累加
88.     }
89.     *cnt = (height-1) * (width-1);
90.     return ssim;
91. }

```

计算SSIM这段代码虽然看上去比较短,但是却不太容易理解。总体说来这段代码实现的SSIM的计算是以8x8的块为单元,而以4为滑动窗口的滑动步长。计算的示意图如下所示,图中每一个小方块代表一个4x4的像素块,绿色方块是正在计算区域。

SSIM Calculation



x264_pixel_ssim_wxh()中是按照4x4的块对像素进行处理的。使用sum1[]保存上一行块的“信息”，sum0[]保存当前一行块的“信息”。“信息”包含4个元素：

- s1: 原始像素之和
- s2: 受损像素之和
- ss: 原始像素平方之和+受损像素平方之和
- s12: 原始像素*受损像素的值的和

ssim_4x4x2_core()用于获取上述信息；而ssim_end4()用于根据这些信息计算SSIM。

ssim_4x4x2_core()

ssim_4x4x2_core()用于获取2个4x4块计算SSIM时候需要用到信息。该函数的定义如下所示。

```

1.  /*****
2.  * structural similarity metric
3.  * 获取2个4x4的块的信息
4.  *****/
5.  static void ssim_4x4_core( const pixel *pix1, intptr_t stride1,
6.                           const pixel *pix2, intptr_t stride2,
7.                           int sums[2][4] )
8.  {
9.      //计算2个块, 分别存在sums[0]和sums[1]
10.     for( int z = 0; z < 2; z++ )
11.     {
12.         uint32_t s1 = 0, s2 = 0, ss = 0, s12 = 0;
13.         /*
14.          * 计算4x4块
15.          * +---+
16.          * |   |
17.          * +---+
18.          */
19.         for( int y = 0; y < 4; y++ )
20.             for( int x = 0; x < 4; x++ )
21.             {
22.                 //两个图像上分别取一个点
23.                 int a = pix1[x+y*stride1];
24.                 int b = pix2[x+y*stride2];
25.                 //累加
26.                 s1 += a;
27.                 s2 += b;
28.                 //平方累加
29.                 ss += a*a;
30.                 ss += b*b;
31.                 //相乘累加
32.                 s12 += a*b;
33.             }
34.         /*
35.          * [0]原始像素之和
36.          * [1]受损像素之和
37.          * [2]原始像素平方之和+受损像素平方之和
38.          * [3]原始像素*受损像素的值的和
39.          *
40.          * [0]为a00+a01+a02...
41.          * [1]为b00+b01+b02...
42.          * [2]为a00^2 +a01^2+...+b00^2+b01^2+...
43.          * [3]为a00*b00+a01*b01+...
44.          */
45.         sums[z][0] = s1;
46.         sums[z][1] = s2;
47.         sums[z][2] = ss;
48.         sums[z][3] = s12;
49.         //右移4个像素
50.         pix1 += 4;
51.         pix2 += 4;
52.     }
53. }

```

从源代码可以看出，ssim_4x4_core()计算了2个4x4的下列信息：

- s1: 原始像素之和
- s2: 受损像素之和
- ss: 原始像素平方之和+受损像素平方之和
- s12: 原始像素*受损像素的值的和

ssim_end4()

ssim_end4()用于计算SSIM，它的定义如下所示。

```

1. //width一般取4
2. static float ssim_end4( int sum0[5][4], int sum1[5][4], int width )
3. {
4.     float ssim = 0.0;
5.     //循环计算8x8块的SSIM（通过4个4x4块），并且累加
6.     /*
7.      *      +-----+-----+
8.      * sum1 | 0 | 1 | 2 | 3 | 4 |
9.      *      +-----+-----+
10.     * sum0 | 0 | 1 | 2 | 3 | 4 |
11.     *      +-----+-----+
12.     *
13.     *      +-----+
14.     * sum1 | 0 | 1 |
15.     *      +-----+
16.     * sum0 | 0 | 1 |
17.     *      +-----+
18.     *
19.     *          +-----+
20.     * sum1      | 1 | 2 |
21.     *          +-----+
22.     * sum0      | 1 | 2 |
23.     *          +-----+
24.     *
25.     *          +-----+
26.     * sum1      | 2 | 3 |
27.     *          +-----+
28.     * sum0      | 2 | 3 |
29.     *          +-----+
30.     *
31.     *          +-----+
32.     * sum1      | 3 | 4 |
33.     *          +-----+
34.     * sum0      | 3 | 4 |
35.     *          +-----+
36.     *
37.     */
38.     for( int i = 0; i < width; i++ )
39.         ssim += ssim_end1( sum0[i][0] + sum0[i+1][0] + sum1[i][0] + sum1[i+1][0],
40.                             sum0[i][1] + sum0[i+1][1] + sum1[i][1] + sum1[i+1][1],
41.                             sum0[i][2] + sum0[i+1][2] + sum1[i][2] + sum1[i+1][2],
42.                             sum0[i][3] + sum0[i+1][3] + sum1[i][3] + sum1[i+1][3] );
43.     return ssim;
44. }

```

该函数中，sum0[]存储了当前一行4x4块的信息，sum1[]存储了上一行4x4块的信息，将sum0[i]，sum0[i+1]，sum1[i]，sum1[i]四个4x4块结合之后就形成了1个8x8的块，传递给ssim_end1()进行计算。

ssim_end1()

ssim_end1()根据SSIM的公式计算1个块的SSIM。该函数的定义如下所示。

```

1. //计算1个块的SSIM
2. static float ssim_end1( int s1, int s2, int ss, int s12 )
3. {
4.     /* Maximum value for 10-bit is: ss*64 = (2^10-1)^2*16*4*64 = 4286582784, which will overflow in some cases.
5.      * s1*s1, s2*s2, and s1*s2 also obtain this value for edge cases: ((2^10-1)*16*4)^2 = 4286582784.
6.      * Maximum value for 9-bit is: ss*64 = (2^9-1)^2*16*4*64 = 1069551616, which will not overflow. */
7.     #if BIT_DEPTH > 9
8.     #define type float
9.         static const float ssim_c1 = .01*.01*PIXEL_MAX*PIXEL_MAX*64;
10.        static const float ssim_c2 = .03*.03*PIXEL_MAX*PIXEL_MAX*64*63;
11.     #else
12.     #define type int
13.        //常量C1,C2
14.        static const int ssim_c1 = (int)(.01*.01*PIXEL_MAX*PIXEL_MAX*64 + .5);
15.        static const int ssim_c2 = (int)(.03*.03*PIXEL_MAX*PIXEL_MAX*64*63 + .5);
16.     #endif
17.
18.     /*
19.      * SSIM公式
20.      * SSIM = ((2*ux*uy+C1)(2*oxy+C2))/((ux^2+uy^2+C1)(ox^2+oy^2+C2))
21.      * 其中
22.      * ux=E(x)
23.      * uy=E(y)
24.      * oxy=cov(x,y)=E(XY)-ux*uy
25.      * ox^2=E(x^2)-E(x)^2
26.      *
27.      * 4个元素中：
28.      * [0]原始像素之和
29.      * [1]受损像素之和
30.      * [2]原始像素平方之和+受损像素平方之和
31.      * [3]原始像素*受损像素的值的和
32.      *
33.      */
34.     //注意：这里都没有求平均值
35.     //E(x)
36.     type fs1 = s1;
37.     //E(y)
38.     type fs2 = s2;
39.     type fss = ss;
40.     type fs12 = s12;
41.     //E(x^2)-E(x)^2+E(y^2)-E(y)^2
42.     type vars = fss*64 - fs1*fs1 - fs2*fs2;
43.     //cov(x,y)
44.     type covar = fs12*64 - fs1*fs2;
45.
46.     //计算公式在这里
47.     return (float)(2*fs1*fs2 + ssim_c1) * (float)(2*covar + ssim_c2)
48.         / ((float)(fs1*fs1 + fs2*fs2 + ssim_c1) * (float)(vars + ssim_c2));
49.     #undef type
50. }

```

从源代码可以看出，ssim_end1()实现了上文所述的SSIM计算公式。

至此有关x264中的滤波模块的源代码就分析完毕了。

雷霄骅

leixiaohua1020@126.com

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

版权声明：本文为博主原创文章，未经博主允许不得转载。 <https://blog.csdn.net/leixiaohua1020/article/details/45870269>

文章标签：

x264

半像素内插

滤波

视频质量

个人分类：

x264

所属专栏：[开源多媒体项目源代码分析](#)

此PDF由spygg生成, 请尊重原作者版权!!!

我的邮箱:liushidc@163.com