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_edgev()和filter_mb_edgeh()对亮度垂直边界和水平边界进行滤波，或者调用filter_mb_edgecv()和filter_mb_edgech()对色度的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
//解码slice
//三个主要步骤：
//1.熵解码（CAVLC/CABAC）
//2.宏块解码
//3.环路滤波
//此外还包含了错误隐藏代码
static int decode_slice(struct AVCodecContext *avctx, void *arg)
{
    H264Context *h = *(void **)arg;
    int lf_x_start = h->mb_x;

    h->mb_skip_run = -1;

    av_assert0(h->block_offset[15] == (4 * ((scan8[15] - scan8[0]) & 7) << h->pixel_shift) + 4 * h->linesize * ((scan8[15] - scan8[0]) >> 3));

    h->is_complex = FRAME_MBAFF(h) || h->picture_structure != PICT_FRAME ||
                    avctx->codec_id != AV_CODEC_ID_H264 ||
                    (CONFIG_GRAY && (h->flags & CODEC_FLAG_GRAY));

    if (!(h->avctx->active_thread_type & FF_THREAD_SLICE) && h->picture_structure == PICT_FRAME && h->er.error_status_table) {
        const int start_i  = av_clip(h->resync_mb_x + h->resync_mb_y * h->mb_width, 0, h->mb_num - 1);
        if (start_i) {
            int prev_status = h->er.error_status_table[h->er.mb_index2xy[start_i - 1]];
            prev_status &= ~ VP_START;
            if (prev_status != (ER_MV_END | ER_DC_END | ER_AC_END))
                h->er.error_occurred = 1;
        }
    }
    //CABAC情况
    if (h->pps.cabac) {
        /* realign */
        align_get_bits(&h->gb);

        /* init cabac */
        //初始化CABAC解码器
        ff_init_cabac_decoder(&h->cabac,
                              h->gb.buffer + get_bits_count(&h->gb) / 8,
                              (get_bits_left(&h->gb) + 7) / 8);

        ff_h264_init_cabac_states(h);
        //循环处理每个宏块
        for (;;) {
            // START_TIMER
            //解码CABAC数据
            int ret = ff_h264_decode_mb_cabac(h);
            int eos;
            // STOP_TIMER("decode_mb_cabac")
            //解码宏块
            if (ret >= 0)
                ff_h264_hl_decode_mb(h);

            // FIXME optimal? or let mb_decode decode 16x32 ?
            //宏块级帧场自适应。很少接触
            if (ret >= 0 && FRAME_MBAFF(h)) {
                h->mb_y++;

                ret = ff_h264_decode_mb_cabac(h);
                //解码宏块
                if (ret >= 0)
                    ff_h264_hl_decode_mb(h);
                h->mb_y--;
            }
            eos = get_cabac_terminate(&h->cabac);

            if ((h->workaround_bugs & FF_BUG_TRUNCATED) &&
                h->cabac.bytestream > h->cabac.bytestream_end + 2) {
                //错误隐藏
                er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x - 1,
                             h->mb_y, ER_MB_END);
                if (h->mb_x >= lf_x_start)
                    loop_filter(h, lf_x_start, h->mb_x + 1);
                return 0;
            }
            if (h->cabac.bytestream > h->cabac.bytestream_end + 2 )
                av_log(h->avctx, AV_LOG_DEBUG, "bytestream overread %"PTRDIFF_SPECIFIER"\n", h->cabac.bytestream_end - h->cabac.bytestream);
            if (ret < 0 || h->cabac.bytestream > h->cabac.bytestream_end + 4) {
                av_log(h->avctx, AV_LOG_ERROR,
                       "error while decoding MB %d %d, bytestream %"PTRDIFF_SPECIFIER"\n",
                       h->mb_x, h->mb_y,
```

```
                    h->cabac.bytestream_end - h->cabac.bytestream);
                er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x,
                             h->mb_y, ER_MB_ERROR);
                return AVERROR_INVALIDDATA;
            }
            //mb_x自增
            //如果自增后超过了一行的mb个数
            if (++h->mb_x >= h->mb_width) {
                //环路滤波
                loop_filter(h, lf_x_start, h->mb_x);
                h->mb_x = lf_x_start = 0;
                decode_finish_row(h);
                //mb_y自增（处理下一行）
                ++h->mb_y;
                //宏块级帧场自适应，暂不考虑
                if (FIELD_OR_MBAFF_PICTURE(h)) {
                    ++h->mb_y;
                    if (FRAME_MBAFF(h) && h->mb_y < h->mb_height)
                        predict_field_decoding_flag(h);
                }
            }
            //如果mb_y超过了mb的行数
            if (eos || h->mb_y >= h->mb_height) {
                tprintf(h->avctx, "slice end %d %d\n",
                        get_bits_count(&h->gb), h->gb.size_in_bits);
                er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x - 1,
                             h->mb_y, ER_MB_END);
                if (h->mb_x > lf_x_start)
                    loop_filter(h, lf_x_start, h->mb_x);
                return 0;
            }
        }
    } else {
        //CAVLC情况
        //循环处理每个宏块
        for (;;) {
            //解码宏块的CAVLC
            int ret = ff_h264_decode_mb_cavlc(h);
            //解码宏块
            if (ret >= 0)
                ff_h264_hl_decode_mb(h);

            // FIXME optimal? or let mb_decode decode 16x32 ?
            if (ret >= 0 && FRAME_MBAFF(h)) {
                h->mb_y++;
                ret = ff_h264_decode_mb_cavlc(h);

                if (ret >= 0)
                    ff_h264_hl_decode_mb(h);
                h->mb_y--;
            }

            if (ret < 0) {
                av_log(h->avctx, AV_LOG_ERROR,
                       "error while decoding MB %d %d\n", h->mb_x, h->mb_y);
                er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x,
                             h->mb_y, ER_MB_ERROR);
                return ret;
            }

            if (++h->mb_x >= h->mb_width) {
                //环路滤波
                loop_filter(h, lf_x_start, h->mb_x);
                h->mb_x = lf_x_start = 0;
                decode_finish_row(h);
                ++h->mb_y;
                if (FIELD_OR_MBAFF_PICTURE(h)) {
                    ++h->mb_y;
                    if (FRAME_MBAFF(h) && h->mb_y < h->mb_height)
                        predict_field_decoding_flag(h);
                }
                if (h->mb_y >= h->mb_height) {
                    tprintf(h->avctx, "slice end %d %d\n",
                            get_bits_count(&h->gb), h->gb.size_in_bits);

                    if (   get_bits_left(&h->gb) == 0
                        || get_bits_left(&h->gb) > 0 && !(h->avctx->err_recognition & AV_EF_AGGRESSIVE)) {
                        //错误隐藏
                        er_add_slice(h, h->resync_mb_x, h->resync_mb_y,
                                     h->mb_x - 1, h->mb_y, ER_MB_END);

                        return 0;
                    } else {
                        er_add_slice(h, h->resync_mb_x, h->resync_mb_y,
                                     h->mb_x, h->mb_y, ER_MB_END);

                        return AVERROR_INVALIDDATA;
                    }
                }
            }
```

```
            if (get_bits_left(&h->gb) <= 0 && h->mb_skip_run <= 0) {
                tprintf(h->avctx, "slice end %d %d\n",
                        get_bits_count(&h->gb), h->gb.size_in_bits);

                if (get_bits_left(&h->gb) == 0) {
                    er_add_slice(h, h->resync_mb_x, h->resync_mb_y,
                                 h->mb_x - 1, h->mb_y, ER_MB_END);
                    if (h->mb_x > lf_x_start)
                        loop_filter(h, lf_x_start, h->mb_x);

                    return 0;
                } else {
                    er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x,
                                 h->mb_y, ER_MB_ERROR);

                    return AVERROR_INVALIDDATA;
                }
            }
        }
    }
}
```

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：

a)调用ff_h264_decode_mb_cavlc()进行CAVLC熵解码

b)调用ff_h264_hl_decode_mb()进行宏块解码

c)解码一行宏块之后调用loop_filter()进行环路滤波

d)此外还有可能调用er_add_slice()进行错误隐藏处理

可以看出，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
//环路滤波
static void loop_filter(H264Context *h, int start_x, int end_x)
{
    uint8_t *dest_y, *dest_cb, *dest_cr;
    int linesize, uvlinesize, mb_x, mb_y;
    const int end_mb_y       = h->mb_y + FRAME_MBAFF(h);
    const int old_slice_type = h->slice_type;
    const int pixel_shift    = h->pixel_shift;
    const int block_h        = 16 >> h->chroma_y_shift;

    if (h->deblocking_filter) {
        //循环处理宏块
        //例如从一行开始的mb_x到一行结束的mb_x
        for (mb_x = start_x; mb_x < end_x; mb_x++)
            for (mb_y = end_mb_y - FRAME_MBAFF(h); mb_y <= end_mb_y; mb_y++) {//逐行扫描只有一行
                int mb_xy, mb_type;
                mb_xy         = h->mb_xy = mb_x + mb_y * h->mb_stride;
                h->slice_num  = h->slice_table[mb_xy];
                mb_type       = h->cur_pic.mb_type[mb_xy];
                h->list_count = h->list_counts[mb_xy];

                if (FRAME_MBAFF(h))
                    h->mb_mbaff               =
                    h->mb_field_decoding_flag = !!IS_INTERLACED(mb_type);

                h->mb_x = mb_x;
                h->mb_y = mb_y;
                //像素数据
                dest_y  = h->cur_pic.f.data[0] +
                          ((mb_x << pixel_shift) + mb_y * h->linesize) * 16;
                dest_cb = h->cur_pic.f.data[1] +
                          (mb_x << pixel_shift) * (8 << CHROMA444(h)) +
                          mb_y * h->uvlinesize * block_h;
                dest_cr = h->cur_pic.f.data[2] +
                          (mb_x << pixel_shift) * (8 << CHROMA444(h)) +
                          mb_y * h->uvlinesize * block_h;
                // FIXME simplify above

                if (MB_FIELD(h)) {
                    linesize   = h->mb_linesize   = h->linesize   * 2;
                    uvlinesize = h->mb_uvlinesize = h->uvlinesize * 2;
                    if (mb_y & 1) { // FIXME move out of this function?
                        dest_y  -= h->linesize   * 15;
                        dest_cb -= h->uvlinesize * (block_h - 1);
                        dest_cr -= h->uvlinesize * (block_h - 1);
                    }
                } else {
                    linesize   = h->mb_linesize   = h->linesize;
                    uvlinesize = h->mb_uvlinesize = h->uvlinesize;
                }
                backup_mb_border(h, dest_y, dest_cb, dest_cr, linesize,
                                 uvlinesize, 0);
                if (fill_filter_caches(h, mb_type))
                    continue;
                h->chroma_qp[0] = get_chroma_qp(h, 0, h->cur_pic.qscale_table[mb_xy]);
                h->chroma_qp[1] = get_chroma_qp(h, 1, h->cur_pic.qscale_table[mb_xy]);
                //宏块滤波器
                if (FRAME_MBAFF(h)) {
                    //宏块级帧场自适应才用，不研究
                    ff_h264_filter_mb(h, mb_x, mb_y, dest_y, dest_cb, dest_cr,
                                      linesize, uvlinesize);
                } else {
                    //宏块滤波器（快速？）
                    ff_h264_filter_mb_fast(h, mb_x, mb_y, dest_y, dest_cb,
                                           dest_cr, linesize, uvlinesize);
                }
            }
    }
    h->slice_type   = old_slice_type;
    h->mb_x         = end_x;
    h->mb_y         = end_mb_y - FRAME_MBAFF(h);
    h->chroma_qp[0] = get_chroma_qp(h, 0, h->qscale);
    h->chroma_qp[1] = get_chroma_qp(h, 1, h->qscale);
}
```

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。

```cpp
//宏块滤波器（快速？）
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 linesize, unsigned int uvlinesize) {
    av_assert2(!FRAME_MBAFF(h));
    if(!h->h264dsp.h264_loop_filter_strength || h->pps.chroma_qp_diff) {
        ff_h264_filter_mb(h, mb_x, mb_y, img_y, img_cb, img_cr, linesize, uvlinesize);
        return;
    }

#if CONFIG_SMALL
    h264_filter_mb_fast_internal(h, mb_x, mb_y, img_y, img_cb, img_cr, linesize, uvlinesize, h->pixel_shift);
#else
    //宏块滤波器-internal（快速？）
    if(h->pixel_shift){
        h264_filter_mb_fast_internal(h, mb_x, mb_y, img_y, img_cb, img_cr, linesize, uvlinesize, 1);
    }else{
        h264_filter_mb_fast_internal(h, mb_x, mb_y, img_y, img_cb, img_cr, linesize, uvlinesize, 0);
    }
#endif
}
```

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。

```cpp
//宏块滤波器-internal（快速？）
static av_always_inline void h264_filter_mb_fast_internal(H264Context *h,
                                                          int mb_x, int mb_y,
                                                          uint8_t *img_y,
                                                          uint8_t *img_cb,
                                                          uint8_t *img_cr,
                                                          unsigned int linesize,
                                                          unsigned int uvlinesize,
                                                          int pixel_shift)
{
    int chroma = CHROMA(h) && !(CONFIG_GRAY && (h->flags&CODEC_FLAG_GRAY));
    int chroma444 = CHROMA444(h);
    int chroma422 = CHROMA422(h);
    //宏块序号
    int mb_xy = h->mb_xy;
    int left_type= h->left_type[LTOP];
    int top_type= h->top_type;

    int qp_bd_offset = 6 * (h->sps.bit_depth_luma - 8);
    int a = 52 + h->slice_alpha_c0_offset - qp_bd_offset;
    int b = 52 + h->slice_beta_offset - qp_bd_offset;
    //宏块类型
    int mb_type = h->cur_pic.mb_type[mb_xy];
    //量化参数
    //qp用于推导alpha,beta（判断是否滤波的门限值）
    int qp      = h->cur_pic.qscale_table[mb_xy];
    int qp0     = h->cur_pic.qscale_table[mb_xy - 1];
    int qp1     = h->cur_pic.qscale_table[h->top_mb_xy];
    int qpc = get_chroma_qp( h, 0, qp );
    int qpc0 = get_chroma_qp( h, 0, qp0 );
    int qpc1 = get_chroma_qp( h, 0, qp1 );
    qp0 = (qp + qp0 + 1) >> 1;
    qp1 = (qp + qp1 + 1) >> 1;
    qpc0 = (qpc + qpc0 + 1) >> 1;
    qpc1 = (qpc + qpc1 + 1) >> 1;
    //Intra类型
    if( IS_INTRA(mb_type) ) {
        static const int16_t bS4[4] = {4,4,4,4};
        static const int16_t bS3[4] = {3,3,3,3};
        const int16_t *bSH = FIELD_PICTURE(h) ? bS3 : bS4;
        /*
         * 帧内宏块滤波
         * 滤波顺序如下所示（大方框代表16x16块）
         *
         * +--4-+--4-+--4-+--4-+
         * 0    1    2    3    |
         * +--5-+--5-+--5-+--5-+
         * 0    1    2    3    |
         * +--6-+--6-+--6-+--6-+
         * 0    1    2    3    |
         * +--7-+--7-+--7-+--7-+
         * 0    1    2    3    |
         * +----+----+----+----+
         *
         */
        if(left_type)
            //宏块的左边边界，强度bs为4的滤波（Vertical）
            filter_mb_edgev( &img_y[4*0<<pixel_shift], linesize, bS4, qp0, a, b, h, 1); //0
```

```
        //不考虑8x8DCT
        if( IS_8x8DCT(mb_type) ) {
            filter_mb_edgev( &img_y[4*2<<pixel_shift], linesize, bS3, qp, a, b, h, 0);
            if(top_type){
                filter_mb_edgeh( &img_y[4*0*linesize], linesize, bSH, qp1, a, b, h, 1);
            }
            filter_mb_edgeh( &img_y[4*2*linesize], linesize, bS3, qp, a, b, h, 0);
        } else {
            //宏块内部强度bs为3的滤波（Vertical）
            filter_mb_edgev( &img_y[4*1<<pixel_shift], linesize, bS3, qp, a, b, h, 0);  //1
            filter_mb_edgev( &img_y[4*2<<pixel_shift], linesize, bS3, qp, a, b, h, 0);  //2
            filter_mb_edgev( &img_y[4*3<<pixel_shift], linesize, bS3, qp, a, b, h, 0);  //3
            if(top_type){
                //宏块的上边边界，强度bs为4的滤波（逐行扫描）（Horizontal）
                filter_mb_edgeh( &img_y[4*0*linesize], linesize, bSH, qp1, a, b, h, 1); //4
            }
            //宏块内部强度bs为3的滤波（Horizontal）
            filter_mb_edgeh( &img_y[4*1*linesize], linesize, bS3, qp, a, b, h, 0);      //5
            filter_mb_edgeh( &img_y[4*2*linesize], linesize, bS3, qp, a, b, h, 0);      //6
            filter_mb_edgeh( &img_y[4*3*linesize], linesize, bS3, qp, a, b, h, 0);      //7
        }
        if(chroma){
            if(chroma444){
                if(left_type){
                    filter_mb_edgev( &img_cb[4*0<<pixel_shift], linesize, bS4, qpc0, a, b, h, 1);
                    filter_mb_edgev( &img_cr[4*0<<pixel_shift], linesize, bS4, qpc0, a, b, h, 1);
                }
                if( IS_8x8DCT(mb_type) ) {
                    filter_mb_edgev( &img_cb[4*2<<pixel_shift], linesize, bS3, qpc, a, b, h, 0);
                    filter_mb_edgev( &img_cr[4*2<<pixel_shift], linesize, bS3, qpc, a, b, h, 0);
                    if(top_type){
                        filter_mb_edgeh( &img_cb[4*0*linesize], linesize, bSH, qpc1, a, b, h, 1 );
                        filter_mb_edgeh( &img_cr[4*0*linesize], linesize, bSH, qpc1, a, b, h, 1 );
                    }
                    filter_mb_edgeh( &img_cb[4*2*linesize], linesize, bS3, qpc, a, b, h, 0);
                    filter_mb_edgeh( &img_cr[4*2*linesize], linesize, bS3, qpc, a, b, h, 0);
                } else {
                    filter_mb_edgev( &img_cb[4*1<<pixel_shift], linesize, bS3, qpc, a, b, h, 0);
                    filter_mb_edgev( &img_cr[4*1<<pixel_shift], linesize, bS3, qpc, a, b, h, 0);
                    filter_mb_edgev( &img_cb[4*2<<pixel_shift], linesize, bS3, qpc, a, b, h, 0);
                    filter_mb_edgev( &img_cr[4*2<<pixel_shift], linesize, bS3, qpc, a, b, h, 0);
                    filter_mb_edgev( &img_cb[4*3<<pixel_shift], linesize, bS3, qpc, a, b, h, 0);
                    filter_mb_edgev( &img_cr[4*3<<pixel_shift], linesize, bS3, qpc, a, b, h, 0);
                    if(top_type){
                        filter_mb_edgeh( &img_cb[4*0*linesize], linesize, bSH, qpc1, a, b, h, 1);
                        filter_mb_edgeh( &img_cr[4*0*linesize], linesize, bSH, qpc1, a, b, h, 1);
                    }
                    //水平horizontal
                    filter_mb_edgeh( &img_cb[4*1*linesize], linesize, bS3, qpc, a, b, h, 0);
                    filter_mb_edgeh( &img_cr[4*1*linesize], linesize, bS3, qpc, a, b, h, 0);
                    filter_mb_edgeh( &img_cb[4*2*linesize], linesize, bS3, qpc, a, b, h, 0);
                    filter_mb_edgeh( &img_cr[4*2*linesize], linesize, bS3, qpc, a, b, h, 0);
                    filter_mb_edgeh( &img_cb[4*3*linesize], linesize, bS3, qpc, a, b, h, 0);
                    filter_mb_edgeh( &img_cr[4*3*linesize], linesize, bS3, qpc, a, b, h, 0);
                }
            }else if(chroma422){
                if(left_type){
                    filter_mb_edgecv(&img_cb[2*0<<pixel_shift], uvlinesize, bS4, qpc0, a, b, h, 1);
                    filter_mb_edgecv(&img_cr[2*0<<pixel_shift], uvlinesize, bS4, qpc0, a, b, h, 1);
                }
                filter_mb_edgecv(&img_cb[2*2<<pixel_shift], uvlinesize, bS3, qpc, a, b, h, 0);
                filter_mb_edgecv(&img_cr[2*2<<pixel_shift], uvlinesize, bS3, qpc, a, b, h, 0);
                if(top_type){
                    filter_mb_edgech(&img_cb[4*0*uvlinesize], uvlinesize, bSH, qpc1, a, b, h, 1);
                    filter_mb_edgech(&img_cr[4*0*uvlinesize], uvlinesize, bSH, qpc1, a, b, h, 1);
                }
                filter_mb_edgech(&img_cb[4*1*uvlinesize], uvlinesize, bS3, qpc, a, b, h, 0);
                filter_mb_edgech(&img_cr[4*1*uvlinesize], uvlinesize, bS3, qpc, a, b, h, 0);
                filter_mb_edgech(&img_cb[4*2*uvlinesize], uvlinesize, bS3, qpc, a, b, h, 0);
                filter_mb_edgech(&img_cr[4*2*uvlinesize], uvlinesize, bS3, qpc, a, b, h, 0);
                filter_mb_edgech(&img_cb[4*3*uvlinesize], uvlinesize, bS3, qpc, a, b, h, 0);
                filter_mb_edgech(&img_cr[4*3*uvlinesize], uvlinesize, bS3, qpc, a, b, h, 0);
            }else{
                if(left_type){
                    filter_mb_edgecv( &img_cb[2*0<<pixel_shift], uvlinesize, bS4, qpc0, a, b, h, 1);
                    filter_mb_edgecv( &img_cr[2*0<<pixel_shift], uvlinesize, bS4, qpc0, a, b, h, 1);
                }
                filter_mb_edgecv( &img_cb[2*2<<pixel_shift], uvlinesize, bS3, qpc, a, b, h, 0);
                filter_mb_edgecv( &img_cr[2*2<<pixel_shift], uvlinesize, bS3, qpc, a, b, h, 0);
                if(top_type){
                    filter_mb_edgech( &img_cb[2*0*uvlinesize], uvlinesize, bSH, qpc1, a, b, h, 1);
                    filter_mb_edgech( &img_cr[2*0*uvlinesize], uvlinesize, bSH, qpc1, a, b, h, 1);
                }
                filter_mb_edgech( &img_cb[2*2*uvlinesize], uvlinesize, bS3, qpc, a, b, h, 0);
                filter_mb_edgech( &img_cr[2*2*uvlinesize], uvlinesize, bS3, qpc, a, b, h, 0);
            }
        }
        return;
    } else {
        //非Intra类型
        LOCAL_ALIGNED_8(int16_t, bS, [2], [4][4]);
```

```
        int edges;
        if( IS_8x8DCT(mb_type) && (h->cbp&7) == 7 && !chroma444 ) {
            edges = 4;
            AV_WN64A(bS[0][0], 0x0002000200020002ULL);
            AV_WN64A(bS[0][2], 0x0002000200020002ULL);
            AV_WN64A(bS[1][0], 0x0002000200020002ULL);
            AV_WN64A(bS[1][2], 0x0002000200020002ULL);
        } else {
            int mask_edge1 = (3*(((5*mb_type)>>5)&1)) | (mb_type>>4); //(mb_type & (MB_TYPE_16x16 | MB_TYPE_8x16)) ? 3 : (mb_type & MB_TYPE_16x8) ? 1 : 0;
            int mask_edge0 = 3*((mask_edge1>>1) & ((5*left_type)>>5)&1); // (mb_type & (MB_TYPE_16x16 | MB_TYPE_8x16)) && (h->left_type[LTOP] & (MB_TYPE_16x16 | MB_TYPE_8x16)) ? 3 : 0;
            int step =  1+(mb_type>>24); //IS_8x8DCT(mb_type) ? 2 : 1;
            edges = 4 - 3*((mb_type>>3) & !(h->cbp & 15)); //(mb_type & MB_TYPE_16x16) && !(h->cbp & 15) ? 1 : 4;
            h->h264dsp.h264_loop_filter_strength( bS, h->non_zero_count_cache, h->ref_cache, h->mv_cache,
                                                  h->list_count==2, edges, step, mask_edge0, mask_edge1, FIELD_PICTURE(h));
        }
        if( IS_INTRA(left_type) )
            AV_WN64A(bS[0][0], 0x0004000400040004ULL);
        if( IS_INTRA(top_type) )
            AV_WN64A(bS[1][0], FIELD_PICTURE(h) ? 0x0003000300030003ULL : 0x0004000400040004ULL);
        //专门定义了一个宏？
#define FILTER(hv,dir,edge,intra)\
        if(AV_RN64A(bS[dir][edge])) {                                   \
            filter_mb_edge##hv( &img_y[4*edge*(dir?linesize:1<<pixel_shift)], linesize, bS[dir][edge], edge ? qp : qp##dir, a, b, h, intra );\
            if(chroma){\
                if(chroma444){\
                    filter_mb_edge##hv( &img_cb[4*edge*(dir?linesize:1<<pixel_shift)], linesize, bS[dir][edge], edge ? qpc : qpc##dir, a, b, h, intra );\
                    filter_mb_edge##hv( &img_cr[4*edge*(dir?linesize:1<<pixel_shift)], linesize, bS[dir][edge], edge ? qpc : qpc##dir, a, b, h, intra );\
                } else if(!(edge&1)) {\
                    filter_mb_edgec##hv( &img_cb[2*edge*(dir?uvlinesize:1<<pixel_shift)], uvlinesize, bS[dir][edge], edge ? qpc : qpc##dir, a, b, h, intra );\
                    filter_mb_edgec##hv( &img_cr[2*edge*(dir?uvlinesize:1<<pixel_shift)], uvlinesize, bS[dir][edge], edge ? qpc : qpc##dir, a, b, h, intra );\
                }\
            }\
        }
        /*
         * 非Intra宏块滤波
         * 滤波顺序如下所示（大方框代表16x16块）
         *
         * +--4-+--4-+--4-+--4-+
         * 0    1    2    3    |
         * +--5-+--5-+--5-+--5-+
         * 0    1    2    3    |
         * +--6-+--6-+--6-+--6-+
         * 0    1    2    3    |
         * +--7-+--7-+--7-+--7-+
         * 0    1    2    3    |
         * +----+----+----+----+
         *
         */
        if(left_type)
            FILTER(v,0,0,1);    //0
        if( edges == 1 ) {
            if(top_type)
                FILTER(h,1,0,1);
        } else if( IS_8x8DCT(mb_type) ) {
            FILTER(v,0,2,0);
            if(top_type)
                FILTER(h,1,0,1);
            FILTER(h,1,2,0);
        } else {
            FILTER(v,0,1,0);    //1
            FILTER(v,0,2,0);    //2
            FILTER(v,0,3,0);    //3
            if(top_type)
                FILTER(h,1,0,1);//4
            FILTER(h,1,1,0);    //5
            FILTER(h,1,2,0);    //6
            FILTER(h,1,3,0);    //7
        }
#undef FILTER
    }
}
```

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_edgeh()进行滤波。

b)对于垂直的边界，调用filter_mb_edgev()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_edgeh()进行滤波。
b)对于垂直的边界，调用filter_mb_edgev()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。

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

filter_mb_edgeh()

filter_mb_edgeh()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。

```cpp
//滤波水平边界（Horizontal）-亮度
//垂直（Vertical）滤波器
//      边界
//         x
//         x
// 边界----------
//         x
//         x
static av_always_inline void filter_mb_edgeh(uint8_t *pix, int stride,
                                             const int16_t bS[4],
                                             unsigned int qp, int a, int b,
                                             H264Context *h, int intra)
{
    //alpha,beta为判断是否滤波的门限值
    //它们是通过将(qp+offset)作为索引查表得到的
    //qp大（压缩大），门限高，更容易发生滤波
    const unsigned int index_a = qp + a;
    const int alpha = alpha_table[index_a];
    const int beta  = beta_table[qp + b];

    //门限为0，不用滤波了
    if (alpha ==0 || beta == 0) return;

    if( bS[0] < 4 || !intra ) {
        int8_t tc[4];
        tc[0] = tc0_table[index_a][bS[0]];
        tc[1] = tc0_table[index_a][bS[1]];
        tc[2] = tc0_table[index_a][bS[2]];
        tc[3] = tc0_table[index_a][bS[3]];
        //边界强度3以下（弱滤波）
        h->h264dsp.h264_v_loop_filter_luma(pix, stride, alpha, beta, tc);
    } else {
        //边界强度为4个滤波（强滤波）
        h->h264dsp.h264_v_loop_filter_luma_intra(pix, stride, alpha, beta);
    }
}
```

从filter_mb_edgeh()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“_edgeh”），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。

```cpp
//滤波垂直边界（Vertical）-亮度
//水平（Horizontal）滤波器
//      边界
//       |
// x x x | x x x
//       |
static av_always_inline void filter_mb_edgev(uint8_t *pix, int stride,
                                             const int16_t bS[4],
                                             unsigned int qp, int a, int b,
                                             H264Context *h, int intra)
{
    const unsigned int index_a = qp + a;
    const int alpha = alpha_table[index_a];
    const int beta  = beta_table[qp + b];
    if (alpha ==0 || beta == 0) return;

    if( bS[0] < 4 || !intra ) {
        int8_t tc[4];
        tc[0] = tc0_table[index_a][bS[0]];
        tc[1] = tc0_table[index_a][bS[1]];
        tc[2] = tc0_table[index_a][bS[2]];
        tc[3] = tc0_table[index_a][bS[3]];
        //Bs取值为1,2,3的弱滤波
        h->h264dsp.h264_h_loop_filter_luma(pix, stride, alpha, beta, tc);
    } else {
        //Bs取值为4的强滤波
        h->h264dsp.h264_h_loop_filter_luma_intra(pix, stride, alpha, beta);
    }
}
```

可以看出filter_mb_edgev()的定义与filter_mb_edgeh()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来判决边界是否需要进行环路滤波。只有满足下面三个条件的时候才能进行环路滤波：

| 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进行环路滤波。

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
//初始化DSP相关的函数。包含了IDCT、环路滤波函数等
av_cold void ff_h264dsp_init(H264DSPContext *c, const int bit_depth,
                             const int chroma_format_idc)
{
#undef FUNC
#define FUNC(a, depth) a ## _ ## depth ## _c

#define ADDPX_DSP(depth) \
    c->h264_add_pixels4_clear = FUNC(ff_h264_add_pixels4, depth);\
    c->h264_add_pixels8_clear = FUNC(ff_h264_add_pixels8, depth)

    if (bit_depth > 8 && bit_depth <= 16) {
        ADDPX_DSP(16);
    } else {
        ADDPX_DSP(8);
    }

#define H264_DSP(depth) \
    c->h264_idct_add= FUNC(ff_h264_idct_add, depth);\
    c->h264_idct8_add= FUNC(ff_h264_idct8_add, depth);\
    c->h264_idct_dc_add= FUNC(ff_h264_idct_dc_add, depth);\
    c->h264_idct8_dc_add= FUNC(ff_h264_idct8_dc_add, depth);\
    c->h264_idct_add16     = FUNC(ff_h264_idct_add16, depth);\
    c->h264_idct8_add4     = FUNC(ff_h264_idct8_add4, depth);\
    if (chroma_format_idc <= 1)\
        c->h264_idct_add8  = FUNC(ff_h264_idct_add8, depth);\
    else\
        c->h264_idct_add8  = FUNC(ff_h264_idct_add8_422, depth);\
    c->h264_idct_add16intra= FUNC(ff_h264_idct_add16intra, depth);\
    c->h264_luma_dc_dequant_idct= FUNC(ff_h264_luma_dc_dequant_idct, depth);\
    if (chroma_format_idc <= 1)\
        c->h264_chroma_dc_dequant_idct= FUNC(ff_h264_chroma_dc_dequant_idct, depth);\
    else\
        c->h264_chroma_dc_dequant_idct= FUNC(ff_h264_chroma422_dc_dequant_idct, depth);\
\
    c->weight_h264_pixels_tab[0]= FUNC(weight_h264_pixels16, depth);\
    c->weight_h264_pixels_tab[1]= FUNC(weight_h264_pixels8, depth);\
    c->weight_h264_pixels_tab[2]= FUNC(weight_h264_pixels4, depth);\
    c->weight_h264_pixels_tab[3]= FUNC(weight_h264_pixels2, depth);\
    c->biweight_h264_pixels_tab[0]= FUNC(biweight_h264_pixels16, depth);\
    c->biweight_h264_pixels_tab[1]= FUNC(biweight_h264_pixels8, depth);\
    c->biweight_h264_pixels_tab[2]= FUNC(biweight_h264_pixels4, depth);\
    c->biweight_h264_pixels_tab[3]= FUNC(biweight_h264_pixels2, depth);\
\
    c->h264_v_loop_filter_luma= FUNC(h264_v_loop_filter_luma, depth);\
    c->h264_h_loop_filter_luma= FUNC(h264_h_loop_filter_luma, depth);\
    c->h264_h_loop_filter_luma_mbaff= FUNC(h264_h_loop_filter_luma_mbaff, depth);\
    c->h264_v_loop_filter_luma_intra= FUNC(h264_v_loop_filter_luma_intra, depth);\
    c->h264_h_loop_filter_luma_intra= FUNC(h264_h_loop_filter_luma_intra, depth);\
```

```
    c->h264_h_loop_filter_luma_mbaff_intra= FUNC(h264_h_loop_filter_luma_mbaff_intra, depth);\
    c->h264_v_loop_filter_chroma= FUNC(h264_v_loop_filter_chroma, depth);\
    if (chroma_format_idc <= 1)\
        c->h264_h_loop_filter_chroma= FUNC(h264_h_loop_filter_chroma, depth);\
    else\
        c->h264_h_loop_filter_chroma= FUNC(h264_h_loop_filter_chroma422, depth);\
    if (chroma_format_idc <= 1)\
        c->h264_h_loop_filter_chroma_mbaff= FUNC(h264_h_loop_filter_chroma_mbaff, depth);\
    else\
        c->h264_h_loop_filter_chroma_mbaff= FUNC(h264_h_loop_filter_chroma422_mbaff, depth);\
    c->h264_v_loop_filter_chroma_intra= FUNC(h264_v_loop_filter_chroma_intra, depth);\
    if (chroma_format_idc <= 1)\
        c->h264_h_loop_filter_chroma_intra= FUNC(h264_h_loop_filter_chroma_intra, depth);\
    else\
        c->h264_h_loop_filter_chroma_intra= FUNC(h264_h_loop_filter_chroma422_intra, depth);\
    if (chroma_format_idc <= 1)\
        c->h264_h_loop_filter_chroma_mbaff_intra= FUNC(h264_h_loop_filter_chroma_mbaff_intra, depth);\
    else\
        c->h264_h_loop_filter_chroma_mbaff_intra= FUNC(h264_h_loop_filter_chroma422_mbaff_intra, depth);\
    c->h264_loop_filter_strength= NULL;
    //根据颜色位深，初始化不同的函数
    //一般为8bit，即执行H264_DSP(8)
    switch (bit_depth) {
    case 9:
        H264_DSP(9);
        break;
    case 10:
        H264_DSP(10);
        break;
    case 12:
        H264_DSP(12);
        break;
    case 14:
        H264_DSP(14);
        break;
    default:
        av_assert0(bit_depth<=8);
        H264_DSP(8);
        break;
    }
    //这个函数查找startcode的时候用到
    //在这里竟然单独列出
    c->startcode_find_candidate = ff_startcode_find_candidate_c;
    //如果系统支持，则初始化经过汇编优化的函数
    if (ARCH_AARCH64) ff_h264dsp_init_aarch64(c, bit_depth, chroma_format_idc);
    if (ARCH_ARM) ff_h264dsp_init_arm(c, bit_depth, chroma_format_idc);
    if (ARCH_PPC) ff_h264dsp_init_ppc(c, bit_depth, chroma_format_idc);
    if (ARCH_X86) ff_h264dsp_init_x86(c, bit_depth, chroma_format_idc);
}
```

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。

```cpp
c->h264_idct_add= ff_h264_idct_add_8_c;
c->h264_idct8_add= ff_h264_idct8_add_8_c;
c->h264_idct_dc_add= ff_h264_idct_dc_add_8_c;
c->h264_idct8_dc_add= ff_h264_idct8_dc_add_8_c;
c->h264_idct_add16     = ff_h264_idct_add16_8_c;
c->h264_idct8_add4     = ff_h264_idct8_add4_8_c;
if (chroma_format_idc <= 1)
    c->h264_idct_add8  = ff_h264_idct_add8_8_c;
else
    c->h264_idct_add8  = ff_h264_idct_add8_422_8_c;
c->h264_idct_add16intra= ff_h264_idct_add16intra_8_c;
c->h264_luma_dc_dequant_idct= ff_h264_luma_dc_dequant_idct_8_c;
if (chroma_format_idc <= 1)
    c->h264_chroma_dc_dequant_idct= ff_h264_chroma_dc_dequant_idct_8_c;
else
    c->h264_chroma_dc_dequant_idct= ff_h264_chroma422_dc_dequant_idct_8_c;

c->weight_h264_pixels_tab[0]= weight_h264_pixels16_8_c;
c->weight_h264_pixels_tab[1]= weight_h264_pixels8_8_c;
c->weight_h264_pixels_tab[2]= weight_h264_pixels4_8_c;
c->weight_h264_pixels_tab[3]= weight_h264_pixels2_8_c;
c->biweight_h264_pixels_tab[0]= biweight_h264_pixels16_8_c;
c->biweight_h264_pixels_tab[1]= biweight_h264_pixels8_8_c;
c->biweight_h264_pixels_tab[2]= biweight_h264_pixels4_8_c;
c->biweight_h264_pixels_tab[3]= biweight_h264_pixels2_8_c;

c->h264_v_loop_filter_luma= h264_v_loop_filter_luma_8_c;
c->h264_h_loop_filter_luma= h264_h_loop_filter_luma_8_c;
c->h264_h_loop_filter_luma_mbaff= h264_h_loop_filter_luma_mbaff_8_c;
c->h264_v_loop_filter_luma_intra= h264_v_loop_filter_luma_intra_8_c;
c->h264_h_loop_filter_luma_intra= h264_h_loop_filter_luma_intra_8_c;
c->h264_h_loop_filter_luma_mbaff_intra=
h264_h_loop_filter_luma_mbaff_intra_8_c;
c->h264_v_loop_filter_chroma= h264_v_loop_filter_chroma_8_c;
if (chroma_format_idc <= 1)
    c->h264_h_loop_filter_chroma= h264_h_loop_filter_chroma_8_c;
else
    c->h264_h_loop_filter_chroma= h264_h_loop_filter_chroma422_8_c;
if (chroma_format_idc <= 1)
    c->h264_h_loop_filter_chroma_mbaff=
h264_h_loop_filter_chroma_mbaff_8_c;
else
    c->h264_h_loop_filter_chroma_mbaff=
h264_h_loop_filter_chroma422_mbaff_8_c;
c->h264_v_loop_filter_chroma_intra= h264_v_loop_filter_chroma_intra_8_c;
if (chroma_format_idc <= 1)
    c->h264_h_loop_filter_chroma_intra=
h264_h_loop_filter_chroma_intra_8_c;
else
    c->h264_h_loop_filter_chroma_intra=
h264_h_loop_filter_chroma422_intra_8_c;
if (chroma_format_idc <= 1)
    c->h264_h_loop_filter_chroma_mbaff_intra=
h264_h_loop_filter_chroma_mbaff_intra_8_c;
else
    c->h264_h_loop_filter_chroma_mbaff_intra=
h264_h_loop_filter_chroma422_mbaff_intra_8_c;
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
//垂直（Vertical）普通滤波器
//      边界
//         x
//         x
// 边界----------
//         x
//         x
static void h264_v_loop_filter_luma_8_c(uint8_t *pix, int stride, int alpha, int beta, int8_t *tc0)
{
    //xstride=stride（用于选择滤波的像素）
    //ystride=1
    //inner_iters=4
    h264_loop_filter_luma_8_c(pix, stride, sizeof(pixel), 4, alpha, beta, tc0);
}
```

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
xstride=stride
ystride=1
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
//-----------------------------------------------------------------------------
//代码中函数名包含大量的“FUNCC”的宏，该宏的定义如下所示
//“FUNCC(xxx)”展开后的结果为“xxx_8_c”，即在“xxx”后面加上“_8_c”
//下面的代码中为了阅读方便，手动展开了一些重要函数的“FUNCC”宏。
//但是手动展开宏比较麻烦，所以还是有一些“FUNCC”宏没有展开
/*
 * 环路滤波函数（Loop Filter）展开结果
 *
 * 源代码注释和处理：雷霄骅
 * leixiaohua1020@126.com
 * http://blog.csdn.net/leixiaohua1020
 */

//亮度的环路滤波器-普通滤波器
//边界强度Bs取1,2,3
//参数：
//p_pix：像素数据
//xstride，ystride：决定了是横向边界滤波器还是纵向边界滤波器
//inner_iters：逐行扫描为4
//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。
//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。

//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。
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)
{
//pixel代表了一个像素，在这里是uint8_t，定义如下所示
//#   define pixel  uint8_t

    pixel *pix = (pixel*)p_pix;
    int i, d;
    //不右移
    xstride >>= sizeof(pixel)-1;
    ystride >>= sizeof(pixel)-1;
    //BIT_DEPTH在这里取值为8，定义如下所示
    //#define BIT_DEPTH 8
    alpha <<= BIT_DEPTH - 8;
    beta  <<= BIT_DEPTH - 8;

    //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
    /*
     * [滤波示例] 大方框代表一个宏块
     *
     * xstride=1, ystride=stride
     *
     * +----+----+----+----+
     * |    X    |    |    |
     * +----+----+----+----+
     * |    X    |    |    |
     * +----+----+----+----+
     * |    X    |    |    |
     * +----+----+----+----+
```

```
     * |    X    |    |    |
     * +----+----+----+----+
     *
     * xstride=stride，ystride=1
     *
     * +----+----+----+----+
     * |    |    |    |    |
     * +--X-+--X-+--X-+--X-+
     * |    |    |    |    |
     * +----+----+----+----+
     * |    |    |    |    |
     * +----+----+----+----+
     * |    |    |    |    |
     * +----+----+----+----+
     */
    //外部循环4次
    for( i = 0; i < 4; i++ ) {
        const int tc_orig = tc0[i] << (BIT_DEPTH - 8);
        if( tc_orig < 0 ) {
            pix += inner_iters*ystride;
            continue;
        }

        //一般inner_iters=4
        for( d = 0; d < inner_iters; d++ ) {
        //p和q
        //如果xstride=stride，ystride=1
        //就是处理纵向的6个像素
        //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：
        //        p2
        //        p1
        //        p0
        //=====图像边界=====
        //        q0
        //        q1
        //        q2
        //
        //如果xstride=1，ystride=stride
        //就是处理纵向的6个像素
        //对应的是方块的横向边界的滤波，即如下所示：
        //          ||
        // p2 p1 p0 || q0 q1 q2
        //          ||
        //         边界

        //注意：这里乘的是xstride
            const int p0 = pix[-1*xstride];
            const int p1 = pix[-2*xstride];
            const int p2 = pix[-3*xstride];
            const int q0 = pix[0];
            const int q1 = pix[1*xstride];
            const int q2 = pix[2*xstride];
        //计算方法参考相关的标准
        //alpha和beta是用于检查图像内容的2个参数
        //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
            if( FFABS( p0 - q0 ) < alpha &&
                FFABS( p1 - p0 ) < beta &&
                FFABS( q1 - q0 ) < beta ) {

                int tc = tc_orig;
                int i_delta;
                //上面2个点（p0，p2）满足条件的时候，滤波p1
                if( FFABS( p2 - p0 ) < beta ) {
                //av_clip(int a, int amin, int amax)用于限幅: Clip a signed integer value into the amin-amax range.
                    if(tc_orig)
                    pix[-2*xstride] = p1 + av_clip( (( p2 + ( ( p0 + q0 + 1 ) >> 1 ) ) >> 1) - p1, -tc_orig, tc_orig );
                    tc++;
                }
                //下面2个点（q0，q2）满足条件的时候，滤波q1
                if( FFABS( q2 - q0 ) < beta ) {
                    //q1
                    if(tc_orig)
                    pix[   xstride] = q1 + av_clip( (( q2 + ( ( p0 + q0 + 1 ) >> 1 ) ) >> 1) - q1, -tc_orig, tc_orig );
                    tc++;
                }

                i_delta = av_clip( (((q0 - p0 ) << 2) + (p1 - q1) + 4) >> 3, -tc, tc );
                //p0
                pix[-xstride] = av_clip_pixel( p0 + i_delta );    /* p0' */
                //q0
                pix[0]        = av_clip_pixel( q0 - i_delta );    /* q0' */
            }
            //移动指针
            //注意：这里加的是ystride
            pix += ystride;
        }
    }
}
```

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。

```cpp
//水平（Horizontal）普通滤波器
//      边界
//       |
// x x x | x x x
//       |
static void h264_h_loop_filter_luma_8_c(uint8_t *pix, int stride, int alpha, int beta, int8_t *tc0)
{
    //xstride=1（用于选择滤波的像素）
    //ystride=stride
    //inner_iters=4
    h264_loop_filter_luma_8_c(pix, sizeof(pixel), stride, 4, alpha, beta, tc0);
}
```

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：

```plain
xstride=1
ystride=stride
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。

```cpp
//垂直（Vertical）强滤波器
//      边界
//         x
//         x
// 边界----------
//         x
//         x
static void h264_v_loop_filter_luma_intra_8_c(uint8_t *pix, int stride, int alpha, int beta)
{
    //xstride=stride
    //ystride=1
    //inner_iters=4
    h264_loop_filter_luma_intra_8_c(pix, stride, sizeof(pixel), 4, alpha, beta);
}
```

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
xstride=stride
ystride=1
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
//亮度的环路滤波器-强滤波器
//边界强度Bs取4（最强）
//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
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)
{
```

```
    pixel *pix = (pixel*)p_pix;
    int d;
    xstride >>= sizeof(pixel)-1;
    ystride >>= sizeof(pixel)-1;
    alpha <<= BIT_DEPTH - 8;
    beta  <<= BIT_DEPTH - 8;

    //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
    /*
     * [滤波示例] 大方框代表一个宏块
     *
     * xstride=1, ystride=stride
     *
     * +----+----+----+----+
     * X    |    |    |    |
     * +----+----+----+----+
     * X    |    |    |    |
     * +----+----+----+----+
     * X    |    |    |    |
     * +----+----+----+----+
     * X    |    |    |    |
     * +----+----+----+----+
     *
     * xstride=stride, ystride=1
     *
     * +--X-+--X-+--X-+--X-+
     * |    |    |    |    |
     * +----+----+----+----+
     * |    |    |    |    |
     * +----+----+----+----+
     * |    |    |    |    |
     * +----+----+----+----+
     * |    |    |    |    |
     * +----+----+----+----+
     */
    //一般inner_iters=4
    for( d = 0; d < 4 * inner_iters; d++ ) {
        //p和q
        //如果xstride=stride, ystride=1
        //就是处理纵向的6个像素
        //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：
        //        p2
        //        p1
        //        p0
        //=====图像边界=====
        //        q0
        //        q1
        //        q2
        //
        //如果xstride=1, ystride=stride
        //就是处理纵向的6个像素
        //对应的是方块的横向边界的滤波，即如下所示：
        //          ||
        // p2 p1 p0 || q0 q1 q2
        //          ||
        //         边界

        //注意：这里乘的是xstride
        const int p2 = pix[-3*xstride];
        const int p1 = pix[-2*xstride];
        const int p0 = pix[-1*xstride];

        const int q0 = pix[ 0*xstride];
        const int q1 = pix[ 1*xstride];
        const int q2 = pix[ 2*xstride];

        if( FFABS( p0 - q0 ) < alpha &&
            FFABS( p1 - p0 ) < beta &&
            FFABS( q1 - q0 ) < beta ) {
            //满足条件的时候，使用强滤波器
            if(FFABS( p0 - q0 ) < (( alpha >> 2 ) + 2 )){
                //p
                if( FFABS( p2 - p0 ) < beta)
                {
                    const int p3 = pix[-4*xstride];
                    /* p0', p1', p2' */
                    pix[-1*xstride] = ( p2 + 2*p1 + 2*p0 + 2*q0 + q1 + 4 ) >> 3;
                    pix[-2*xstride] = ( p2 + p1 + p0 + q0 + 2 ) >> 2;
                    pix[-3*xstride] = ( 2*p3 + 3*p2 + p1 + p0 + q0 + 4 ) >> 3;
                } else {
                    //不满足条件的时候
                    /* p0' */
                    pix[-1*xstride] = ( 2*p1 + p0 + q1 + 2 ) >> 2;
                }
                //q
                if( FFABS( q2 - q0 ) < beta)
                {
                    const int q3 = pix[3*xstride];
                    /* q0', q1', q2' */
                    pix[0*xstride] = ( p1 + 2*p0 + 2*q0 + 2*q1 + q2 + 4 ) >> 3;
                    pix[1*xstride] = ( p0 + q0 + q1 + q2 + 2 ) >> 2;
```

```
                    pix[2*xstride] = ( 2*q3 + 3*q2 + q1 + q0 + p0 + 4 ) >> 3;
                } else {
                    /* q0' */
                    pix[0*xstride] = ( 2*q1 + q0 + p1 + 2 ) >> 2;
                }
            }else{
                //不满足条件的时候，使用下式修正
                /* p0', q0' */
                pix[-1*xstride] = ( 2*p1 + p0 + q1 + 2 ) >> 2;
                pix[ 0*xstride] = ( 2*q1 + q0 + p1 + 2 ) >> 2;
            }
        }
        pix += ystride;
    }
}
```

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）：

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

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。

```cpp
//水平（Horizontal）强滤波器
//      边界
//       |
// x x x | x x x
//       |
static void h264_h_loop_filter_luma_intra_8_c(uint8_t *pix, int stride, int alpha, int beta)
{
    //xstride=1
    //ystride=stride
    //inner_iters=4
    h264_loop_filter_luma_intra_8_c(pix, sizeof(pixel), stride, 4, alpha, beta);
}
```

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

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