原 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）

2015年04月20日 14:55:51　阅读数：14580

==========

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）

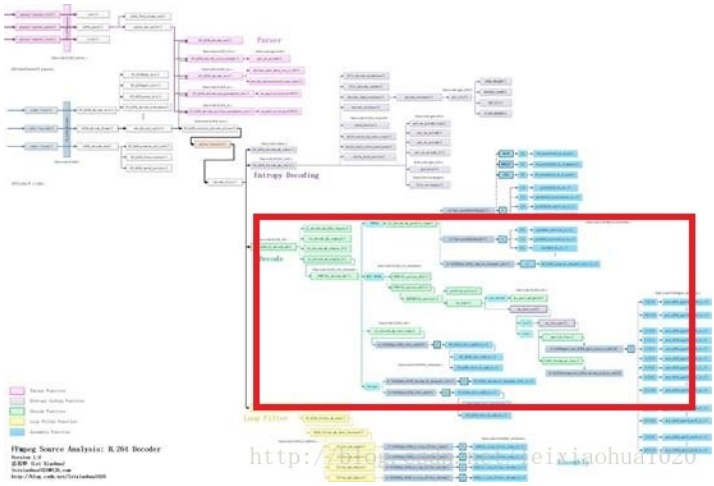
FFmpeg的H.264解码器源代码简单分析：环路滤波（Loop Filter）部分

==========

本文分析FFmpeg的H.264解码器的宏块解码（Decode）部分的源代码。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2个步骤。由于宏块解码部分的内容比较多，因此将本部分内容拆分成两篇文章：一篇文章记录帧内预测宏块（Intra）的宏块解码，另一篇文章记录帧间预测宏块（Inter）的宏块解码。

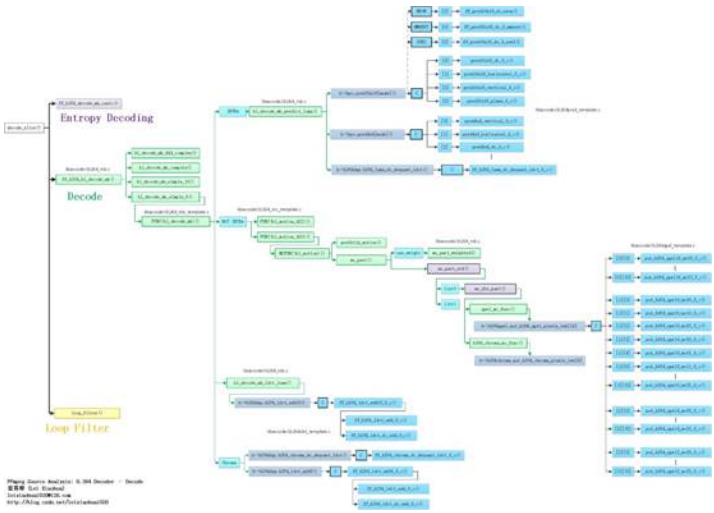
函数调用关系图

宏块解码（Decode）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

宏块解码（Decode）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

宏块解码函数（Decode）通过帧内预测、帧间预测、DCT反变换等方法解码压缩数据。解码函数是ff_h264_hl_decode_mb()。其中跟宏块类型的不同，会调用几个不同的函数，最常见的就是调用hl_decode_mb_simple_8()。

hl_decode_mb_simple_8()的定义是无法在源代码中直接找到的，这是因为它实际代码的函数名称是使用宏的方式写的。hl_decode_mb_simple_8()的源代码实际上就是FUNC(hl_decode_mb)()函数的源代码。

从函数调用图中可以看出，FUNC(hl_decode_mb)()根据宏块类型的不同作不同的处理：如果帧内预测宏块（INTRA），就会调用hl_decode_mb_predict_luma()进行帧内预测；如果是帧间预测宏块（INTER），就会调用FUNC(hl_motion_422)()或者FUNC(hl_motion_420)()进行四分之一像素运动补偿。

经过帧内预测或者帧间预测步骤之后，就得到了预测数据。随后FUNC(hl_decode_mb)()会调用hl_decode_mb_idct_luma()等几个函数对残差数据进行DCT反变换工作，并将变换后的数据叠加到预测数据上，形成解码后的图像数据。

由于帧内预测宏块和帧间预测宏块的解码工作都比较复杂，因此分成两篇文章记录这两部分的源代码。本文记录帧内预测宏块解码时候的源代码。

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;
```

```
                ++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);
```

```
185.
186.                     return AVERROR_INVALIDDATA;
187.                 }
188.             }
189.         }
190.     }
191. }
```

重复记录一下decode_slice()的流程：

（1）判断H.264码流是CABAC编码还是CAVLC编码，进入不同的处理循环。

（2）如果是CABAC编码，首先调用ff_init_cabac_decoder()初始化CABAC解码器。然后进入一个循环，依次对每个宏块进行以下处理：

a)调用ff_h264_decode_mb_cabac()进行CABAC熵解码

b)调用ff_h264_hl_decode_mb()进行宏块解码

c)解码一行宏块之后调用loop_filter()进行环路滤波

d)此外还有可能调用er_add_slice()进行错误隐藏处理

（3）如果是CABAC编码，直接进入一个循环，依次对每个宏块进行以下处理：

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，宏块解码函数是ff_h264_hl_decode_mb()。下面看一下这个函数。

ff_h264_hl_decode_mb()

ff_h264_hl_decode_mb()完成了宏块解码的工作。“宏块解码”就是根据前一步骤“熵解码”得到的宏块类型、运动矢量、参考帧、DCT残差数据等信息恢复图像数据的过程。该函数的定义位于libavcodec\h264_mb.c，如下所示。

```cpp
//解码宏块
void ff_h264_hl_decode_mb(H264Context *h)
{
    //宏块序号 mb_xy = mb_x + mb_y*mb_stride
    const int mb_xy   = h->mb_xy;
    //宏块类型
    const int mb_type = h->cur_pic.mb_type[mb_xy];
    //比较少见，PCM类型
    int is_complex    = CONFIG_SMALL || h->is_complex ||
                        IS_INTRA_PCM(mb_type) || h->qscale == 0;
    //YUV444
    if (CHROMA444(h)) {
        if (is_complex || h->pixel_shift)
            hl_decode_mb_444_complex(h);
        else
            hl_decode_mb_444_simple_8(h);
    } else if (is_complex) {
        hl_decode_mb_complex(h);   //PCM类型？
    } else if (h->pixel_shift) {
        hl_decode_mb_simple_16(h); //色彩深度为16

    } else
        hl_decode_mb_simple_8(h);  //色彩深度为8
}
```

可以看出ff_h264_hl_decode_mb()的定义很简单：通过系统的参数（例如颜色位深是不是8bit，YUV采样格式是不是4：4：4等）判断该调用哪一个函数作为解码函数。由于最普遍的情况是解码8bit的YUV420P格式的H.264数据，因此一般情况下会调用hl_decode_mb_simple_8()。这里有一点需要注意：如果我们直接查找hl_decode_mb_simple_8()的定义，会发现这个函数是找不到的。这个函数的定义实际上就是FUNC(hl_decode_mb)()函数。FUNC(hl_decode_mb)()函数名称中的宏“FUNC()”展开后就是hl_decode_mb_simple_8()。那么我们看一下FUNC(hl_decode_mb)()函数。

FUNC(hl_decode_mb)()

FUNC(hl_decode_mb)()的定义位于libavcodec\h264_mb_template.c。下面看一下FUNC(hl_decode_mb)()函数的定义。

PS：在这里需要注意，FFmpeg H.264解码器中名称中包含“_template”的C语言文件中的函数都是使用类似于“FUNC(name)()”的方式书写的，这样做的目的大概是为了适配各种各样的功能。例如在处理16bit的H.264码流的时候，FUNC(hl_decode_mb)()可以展开为hl_decode_mb_simple_16()函数；同理，FUNC(hl_decode_mb)()在其他条件下也可以展开为hl_decode_mb_complex()函数。

```cpp
//hl是什么意思？high level？
/*
 * 注释：雷霄骅
 * leixiaohua1020@126.com
 * http://blog.csdn.net/leixiaohua1020
 *
 * 宏块解码
```

```
 * 帧内宏块：帧内预测->残差DCT反变换
 * 帧间宏块：帧间预测（运动补偿）->残差DCT反变换
 *
 */
static av_noinline void FUNC(hl_decode_mb)(H264Context *h)
{
    //序号：x（行）和y（列）
    const int mb_x    = h->mb_x;
    const int mb_y    = h->mb_y;
    //宏块序号 mb_xy = mb_x + mb_y*mb_stride
    const int mb_xy   = h->mb_xy;
    //宏块类型
    const int mb_type = h->cur_pic.mb_type[mb_xy];
    //这三个变量存储最后处理完成的像素值
    uint8_t *dest_y, *dest_cb, *dest_cr;
    int linesize, uvlinesize /*dct_offset*/;
    int i, j;
    int *block_offset = &h->block_offset[0];
    const int transform_bypass = !SIMPLE && (h->qscale == 0 && h->sps.transform_bypass);
    /* is_h264 should always be true if SVQ3 is disabled. */
    const int is_h264 = !CONFIG_SVQ3_DECODER || SIMPLE || h->avctx->codec_id == AV_CODEC_ID_H264;
    void (*idct_add)(uint8_t *dst, int16_t *block, int stride);
    const int block_h   = 16 >> h->chroma_y_shift;
    const int chroma422 = CHROMA422(h);
    //存储Y，U，V像素的位置：dest_y，dest_cb，dest_cr
    //分别对应AVFrame的data[0]，data[1]，data[2]
    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 + mb_y * h->uvlinesize * block_h;
    dest_cr = h->cur_pic.f.data[2] +  (mb_x << PIXEL_SHIFT) * 8 + mb_y * h->uvlinesize * block_h;

    h->vdsp.prefetch(dest_y  + (h->mb_x & 3) * 4 * h->linesize   + (64 << PIXEL_SHIFT), h->linesize,       4);
    h->vdsp.prefetch(dest_cb + (h->mb_x & 7)     * h->uvlinesize + (64 << PIXEL_SHIFT), dest_cr - dest_cb, 2);

    h->list_counts[mb_xy] = h->list_count;

    //系统中包含了
    //#define SIMPLE 1
    //不会执行？
    if (!SIMPLE && MB_FIELD(h)) {
        linesize     = h->mb_linesize = h->linesize * 2;
        uvlinesize   = h->mb_uvlinesize = h->uvlinesize * 2;
        block_offset = &h->block_offset[48];
        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);
        }
        if (FRAME_MBAFF(h)) {
            int list;
            for (list = 0; list < h->list_count; list++) {
                if (!USES_LIST(mb_type, list))
                    continue;
                if (IS_16X16(mb_type)) {
                    int8_t *ref = &h->ref_cache[list][scan8[0]];
                    fill_rectangle(ref, 4, 4, 8, (16 + *ref) ^ (h->mb_y & 1), 1);
                } else {
                    for (i = 0; i < 16; i += 4) {
                        int ref = h->ref_cache[list][scan8[i]];
                        if (ref >= 0)
                            fill_rectangle(&h->ref_cache[list][scan8[i]], 2, 2,
                                           8, (16 + ref) ^ (h->mb_y & 1), 1);
                    }
                }
            }
        }
    } else {
        linesize   = h->mb_linesize   = h->linesize;
        uvlinesize = h->mb_uvlinesize = h->uvlinesize;
        // dct_offset = s->linesize * 16;
    }
    //系统中包含了
    //#define SIMPLE 1
    //不会执行？
    if (!SIMPLE && IS_INTRA_PCM(mb_type)) {
        const int bit_depth = h->sps.bit_depth_luma;
        if (PIXEL_SHIFT) {
            int j;
            GetBitContext gb;
            init_get_bits(&gb, h->intra_pcm_ptr,
                          ff_h264_mb_sizes[h->sps.chroma_format_idc] * bit_depth);

            for (i = 0; i < 16; i++) {
                uint16_t *tmp_y = (uint16_t *)(dest_y + i * linesize);
                for (j = 0; j < 16; j++)
                    tmp_y[j] = get_bits(&gb, bit_depth);
            }
            if (SIMPLE || !CONFIG_GRAY || !(h->flags & CODEC_FLAG_GRAY)) {
                if (!h->sps.chroma_format_idc) {
                    for (i = 0; i < block_h; i++) {
                        uint16_t *tmp_cb = (uint16_t *)(dest_cb + i * uvlinesize);
                        uint16_t *tmp_cr = (uint16_t *)(dest_cr + i * uvlinesize);
```

```
                    for (j = 0; j < 8; j++) {
                        tmp_cb[j] = tmp_cr[j] = 1 << (bit_depth - 1);
                    }
                }
            } else {
                for (i = 0; i < block_h; i++) {
                    uint16_t *tmp_cb = (uint16_t *)(dest_cb + i * uvlinesize);
                    for (j = 0; j < 8; j++)
                        tmp_cb[j] = get_bits(&gb, bit_depth);
                }
                for (i = 0; i < block_h; i++) {
                    uint16_t *tmp_cr = (uint16_t *)(dest_cr + i * uvlinesize);
                    for (j = 0; j < 8; j++)
                        tmp_cr[j] = get_bits(&gb, bit_depth);
                }
            }
        }
    } else {
        for (i = 0; i < 16; i++)
            memcpy(dest_y + i * linesize, h->intra_pcm_ptr + i * 16, 16);
        if (SIMPLE || !CONFIG_GRAY || !(h->flags & CODEC_FLAG_GRAY)) {
            if (!h->sps.chroma_format_idc) {
                for (i = 0; i < 8; i++) {
                    memset(dest_cb + i * uvlinesize, 1 << (bit_depth - 1), 8);
                    memset(dest_cr + i * uvlinesize, 1 << (bit_depth - 1), 8);
                }
            } else {
                const uint8_t *src_cb = h->intra_pcm_ptr + 256;
                const uint8_t *src_cr = h->intra_pcm_ptr + 256 + block_h * 8;
                for (i = 0; i < block_h; i++) {
                    memcpy(dest_cb + i * uvlinesize, src_cb + i * 8, 8);
                    memcpy(dest_cr + i * uvlinesize, src_cr + i * 8, 8);
                }
            }
        }
    }
} else {
    //Intra类型
    //Intra4x4或者Intra16x16

    if (IS_INTRA(mb_type)) {
        if (h->deblocking_filter)
            xchg_mb_border(h, dest_y, dest_cb, dest_cr, linesize,
                           uvlinesize, 1, 0, SIMPLE, PIXEL_SHIFT);

        if (SIMPLE || !CONFIG_GRAY || !(h->flags & CODEC_FLAG_GRAY)) {
            h->hpc.pred8x8[h->chroma_pred_mode](dest_cb, uvlinesize);
            h->hpc.pred8x8[h->chroma_pred_mode](dest_cr, uvlinesize);
        }
        //帧内预测-亮度
        hl_decode_mb_predict_luma(h, mb_type, is_h264, SIMPLE,
                                  transform_bypass, PIXEL_SHIFT,
                                  block_offset, linesize, dest_y, 0);

        if (h->deblocking_filter)
            xchg_mb_border(h, dest_y, dest_cb, dest_cr, linesize,
                           uvlinesize, 0, 0, SIMPLE, PIXEL_SHIFT);
    } else if (is_h264) {
        //Inter类型

        //运动补偿
        if (chroma422) {
            FUNC(hl_motion_422)(h, dest_y, dest_cb, dest_cr,
                                h->qpel_put, h->h264chroma.put_h264_chroma_pixels_tab,
                                h->qpel_avg, h->h264chroma.avg_h264_chroma_pixels_tab,
                                h->h264dsp.weight_h264_pixels_tab,
                                h->h264dsp.biweight_h264_pixels_tab);
        } else {
            //“*_put”处理单向预测，“*_avg”处理双向预测，“weight”处理加权预测
            //h->qpel_put[16]包含了单向预测的四分之一像素运动补偿所有样点处理的函数
            //两个像素之间横向的点（内插点和原始的点）有4个，纵向的点有4个，组合起来一共16个
            //h->qpel_avg[16]情况也类似
            FUNC(hl_motion_420)(h, dest_y, dest_cb, dest_cr,
                                h->qpel_put, h->h264chroma.put_h264_chroma_pixels_tab,
                                h->qpel_avg, h->h264chroma.avg_h264_chroma_pixels_tab,
                                h->h264dsp.weight_h264_pixels_tab,
                                h->h264dsp.biweight_h264_pixels_tab);
        }
    }
    //亮度的IDCT
    hl_decode_mb_idct_luma(h, mb_type, is_h264, SIMPLE, transform_bypass,
                           PIXEL_SHIFT, block_offset, linesize, dest_y, 0);
    //色度的IDCT（没有写在一个单独的函数中）
    if ((SIMPLE || !CONFIG_GRAY || !(h->flags & CODEC_FLAG_GRAY)) &&
        (h->cbp & 0x30)) {
        uint8_t *dest[2] = { dest_cb, dest_cr };
        //transform_bypass=0，不考虑
        if (transform_bypass) {
            if (IS_INTRA(mb_type) && h->sps.profile_idc == 244 &&
                (h->chroma_pred_mode == VERT_PRED8x8 ||
                 h->chroma_pred_mode == HOR_PRED8x8)) {
```

```
                    h->hpc.pred8x8_add[h->chroma_pred_mode](dest[0],
                                                            block_offset + 16,
                                                            h->mb + (16 * 16 * 1 << PIXEL_SHIFT),
                                                            uvlinesize);
                    h->hpc.pred8x8_add[h->chroma_pred_mode](dest[1],
                                                            block_offset + 32,
                                                            h->mb + (16 * 16 * 2 << PIXEL_SHIFT),
                                                            uvlinesize);
                } else {
                    idct_add = h->h264dsp.h264_add_pixels4_clear;
                    for (j = 1; j < 3; j++) {
                        for (i = j * 16; i < j * 16 + 4; i++)
                            if (h->non_zero_count_cache[scan8[i]] ||
                                dctcoef_get(h->mb, PIXEL_SHIFT, i * 16))
                                idct_add(dest[j - 1] + block_offset[i],
                                         h->mb + (i * 16 << PIXEL_SHIFT),
                                         uvlinesize);
                        if (chroma422) {
                            for (i = j * 16 + 4; i < j * 16 + 8; i++)
                                if (h->non_zero_count_cache[scan8[i + 4]] ||
                                    dctcoef_get(h->mb, PIXEL_SHIFT, i * 16))
                                    idct_add(dest[j - 1] + block_offset[i + 4],
                                             h->mb + (i * 16 << PIXEL_SHIFT),
                                             uvlinesize);
                        }
                    }
                }
            } else {
                if (is_h264) {
                    int qp[2];
                    if (chroma422) {
                        qp[0] = h->chroma_qp[0] + 3;
                        qp[1] = h->chroma_qp[1] + 3;
                    } else {
                        qp[0] = h->chroma_qp[0];
                        qp[1] = h->chroma_qp[1];
                    }
                    //色度的IDCT

                    //直流分量的hadamard变换
                    if (h->non_zero_count_cache[scan8[CHROMA_DC_BLOCK_INDEX + 0]])
                        h->h264dsp.h264_chroma_dc_dequant_idct(h->mb + (16 * 16 * 1 << PIXEL_SHIFT),
                                                               h->dequant4_coeff[IS_INTRA(mb_type) ? 1 : 4][qp[0]][0]);
                    if (h->non_zero_count_cache[scan8[CHROMA_DC_BLOCK_INDEX + 1]])
                        h->h264dsp.h264_chroma_dc_dequant_idct(h->mb + (16 * 16 * 2 << PIXEL_SHIFT),
                                                               h->dequant4_coeff[IS_INTRA(mb_type) ? 2 : 5][qp[1]][0]);
                    //IDCT
                    //最后的“8”代表内部循环处理8次（U,V各4次）
                    h->h264dsp.h264_idct_add8(dest, block_offset,
                                              h->mb, uvlinesize,
                                              h->non_zero_count_cache);
                } else if (CONFIG_SVQ3_DECODER) {
                    h->h264dsp.h264_chroma_dc_dequant_idct(h->mb + 16 * 16 * 1,
                                                           h->dequant4_coeff[IS_INTRA(mb_type) ? 1 : 4][h->chroma_qp[0]][0]);
                    h->h264dsp.h264_chroma_dc_dequant_idct(h->mb + 16 * 16 * 2,
                                                           h->dequant4_coeff[IS_INTRA(mb_type) ? 2 : 5][h->chroma_qp[1]][0]);
                    for (j = 1; j < 3; j++) {
                        for (i = j * 16; i < j * 16 + 4; i++)
                            if (h->non_zero_count_cache[scan8[i]] || h->mb[i * 16]) {
                                uint8_t *const ptr = dest[j - 1] + block_offset[i];
                                ff_svq3_add_idct_c(ptr, h->mb + i * 16,
                                                   uvlinesize,
                                                   ff_h264_chroma_qp[0][h->qscale + 12] - 12, 2);
                            }
                    }
                }
            }
        }
    }
}
```

下面简单梳理一下FUNC(hl_decode_mb)的流程（在这里只考虑亮度分量的解码，色度分量的解码过程是类似的）：

（1）预测

a)如果是帧内预测宏块（Intra），调用hl_decode_mb_predict_luma()进行帧内预测，得到预测数据。

b)如果不是帧内预测宏块（Inter），调用FUNC(hl_motion_420)()或者FUNC(hl_motion_422)()进行帧间预测（即运动补偿），得到预测数据。

（2）残差叠加

a)调用hl_decode_mb_idct_luma()对DCT残差数据进行DCT反变换，获得残差像素数据并且叠加到之前得到的预测数据上，得到最后的图像数据。

PS：该流程中有一个重要的贯穿始终的内存指针dest_y，其指向的内存中存储了解码后的亮度数据。

本文将会分析上述流程中的帧内预测和残差叠加两个部分。下面先看一下帧内预测函数hl_decode_mb_predict_luma()。

hl_decode_mb_predict_luma()

hl_decode_mb_predict_luma()对帧内宏块进行帧内预测，它的定义位于libavcodec\h264_mb.c，如下所示。

```cpp
//帧内预测-亮度
//分成2种情况：Intra4x4和Intra16x16
static av_always_inline void hl_decode_mb_predict_luma(H264Context *h,
                                                       int mb_type, int is_h264,
                                                       int simple,
                                                       int transform_bypass,
                                                       int pixel_shift,
                                                       int *block_offset,
                                                       int linesize,
                                                       uint8_t *dest_y, int p)
{
    //用于DCT反变换
    void (*idct_add)(uint8_t *dst, int16_t *block, int stride);
    void (*idct_dc_add)(uint8_t *dst, int16_t *block, int stride);
    int i;
    int qscale = p == 0 ? h->qscale : h->chroma_qp[p - 1];
    //外部调用时候p=0
    block_offset += 16 * p;
    if (IS_INTRA4x4(mb_type)) {
        //Intra4x4帧内预测

        if (IS_8x8DCT(mb_type)) {
            //如果使用了8x8的DCT，先不研究
            if (transform_bypass) {
                idct_dc_add =
                idct_add    = h->h264dsp.h264_add_pixels8_clear;
            } else {
                idct_dc_add = h->h264dsp.h264_idct8_dc_add;
                idct_add    = h->h264dsp.h264_idct8_add;
            }
            for (i = 0; i < 16; i += 4) {
                uint8_t *const ptr = dest_y + block_offset[i];
                const int dir      = h->intra4x4_pred_mode_cache[scan8[i]];
                if (transform_bypass && h->sps.profile_idc == 244 && dir <= 1) {
                    if (h->x264_build != -1) {
                        h->hpc.pred8x8l_add[dir](ptr, h->mb + (i * 16 + p * 256 << pixel_shift), linesize);
                    } else
                        h->hpc.pred8x8l_filter_add[dir](ptr, h->mb + (i * 16 + p * 256 << pixel_shift),
                                                        (h-> topleft_samples_available << i) & 0x8000,
                                                        (h->topright_samples_available << i) & 0x4000, linesize);
                } else {
                    const int nnz = h->non_zero_count_cache[scan8[i + p * 16]];
                    h->hpc.pred8x8l[dir](ptr, (h->topleft_samples_available << i) & 0x8000,
                                         (h->topright_samples_available << i) & 0x4000, linesize);
                    if (nnz) {
                        if (nnz == 1 && dctcoef_get(h->mb, pixel_shift, i * 16 + p * 256))
                            idct_dc_add(ptr, h->mb + (i * 16 + p * 256 << pixel_shift), linesize);
                        else
                            idct_add(ptr, h->mb + (i * 16 + p * 256 << pixel_shift), linesize);
                    }
                }
            }
        } else {
            /*
             * Intra4x4帧内预测：16x16 宏块被划分为16个4x4子块
             *
             * +----+----+----+----+
             * |    |    |    |    |
             * +----+----+----+----+
             * |    |    |    |    |
             * +----+----+----+----+
             * |    |    |    |    |
             * +----+----+----+----+
             * |    |    |    |    |
             * +----+----+----+----+
             *
             */
            //4x4的IDCT
            //transform_bypass=0，不考虑
            if (transform_bypass) {
                idct_dc_add  =
                idct_add     = h->h264dsp.h264_add_pixels4_clear;
            } else {
                //常见情况
                idct_dc_add = h->h264dsp.h264_idct_dc_add;
                idct_add    = h->h264dsp.h264_idct_add;
            }
            //循环4x4=16个DCT块
            for (i = 0; i < 16; i++) {
                //ptr指向输出的像素数据
                uint8_t *const ptr = dest_y + block_offset[i];
                //dir存储了帧内预测模式
                const int dir      = h->intra4x4_pred_mode_cache[scan8[i]];

                if (transform_bypass && h->sps.profile_idc == 244 && dir <= 1) {
```

```
                if (transform_bypass && h->sps.profile_idc == 244 && dir <= 1) {
                    h->hpc.pred4x4_add[dir](ptr, h->mb + (i * 16 + p * 256 << pixel_shift), linesize);
                } else {
                    uint8_t *topright;
                    int nnz, tr;
                    uint64_t tr_high;
                    //这2种模式特殊的处理？
                    if (dir == DIAG_DOWN_LEFT_PRED || dir == VERT_LEFT_PRED) {
                        const int topright_avail = (h->topright_samples_available << i) & 0x8000;
                        av_assert2(h->mb_y || linesize <= block_offset[i]);
                        if (!topright_avail) {
                            if (pixel_shift) {
                                tr_high  = ((uint16_t *)ptr)[3 - linesize / 2] * 0x0001000100010001ULL;
                                topright = (uint8_t *)&tr_high;
                            } else {
                                tr       = ptr[3 - linesize] * 0x01010101u;
                                topright = (uint8_t *)&tr;
                            }
                        } else
                            topright = ptr + (4 << pixel_shift) - linesize;
                    } else
                        topright = NULL;
                    //汇编函数：4x4帧内预测（9种方式：Vertical，Horizontal，DC，Plane等等。。。）
                    h->hpc.pred4x4[dir](ptr, topright, linesize);
                    //每个4x4块的非0系数个数的缓存
                    nnz = h->non_zero_count_cache[scan8[i + p * 16]];
                    //有非0系数的时候才处理
                    //h->mb中存储了DCT系数
                    //输出存储在ptr
                    if (nnz) {
                        if (is_h264) {
                            if (nnz == 1 && dctcoef_get(h->mb, pixel_shift, i * 16 + p * 256))
                                idct_dc_add(ptr, h->mb + (i * 16 + p * 256 << pixel_shift), linesize);//特殊：AC系数全为0时候调用
                            else
                                idct_add(ptr, h->mb + (i * 16 + p * 256 << pixel_shift), linesize);//4x4DCT反变换
                        } else if (CONFIG_SVQ3_DECODER)
                            ff_svq3_add_idct_c(ptr, h->mb + i * 16 + p * 256, linesize, qscale, 0);
                    }
                }
            }
        }
    } else {
        /*
         * Intra16x16帧内预测
         *
         * +--------+--------+
         * |                 |
         * |                 |
         * |                 |
         * +        +        +
         * |                 |
         * |                 |
         * |                 |
         * +--------+--------+
         *
         */
        //汇编函数：16x16帧内预测（4种方式：Vertical，Horizontal，DC，Plane）
        h->hpc.pred16x16[h->intra16x16_pred_mode](dest_y, linesize);
        if (is_h264) {
            if (h->non_zero_count_cache[scan8[LUMA_DC_BLOCK_INDEX + p]]) {
                //有非0系数的时候才处理
                //Hadamard反变换
                //h->mb中存储了DCT系数
                //h->mb_luma_dc中存储了16个DCT的直流分量
                if (!transform_bypass)
                    h->h264dsp.h264_luma_dc_dequant_idct(h->mb + (p * 256 << pixel_shift),
                                                         h->mb_luma_dc[p],
                                                         h->dequant4_coeff[p][qscale][0]);
                //注：此处仅仅进行了Hadamard反变换，并未进行DCT反变换
                //Intra16x16在解码过程中的DCT反变换并不是在这里进行，而是在后面进行
                else {
                    static const uint8_t dc_mapping[16] = {
                         0 * 16,  1 * 16,  4 * 16,  5 * 16,
                         2 * 16,  3 * 16,  6 * 16,  7 * 16,
                         8 * 16,  9 * 16, 12 * 16, 13 * 16,
                        10 * 16, 11 * 16, 14 * 16, 15 * 16
                    };
                    for (i = 0; i < 16; i++)
                        dctcoef_set(h->mb + (p * 256 << pixel_shift),
                                    pixel_shift, dc_mapping[i],
                                    dctcoef_get(h->mb_luma_dc[p],
                                                pixel_shift, i));
                }
            }
        } else if (CONFIG_SVQ3_DECODER)
            ff_svq3_luma_dc_dequant_idct_c(h->mb + p * 256,
                                           h->mb_luma_dc[p], qscale);
    }
}
```

下面根据原代码梳理一下hl_decode_mb_predict_luma()的主干：

（1）如果宏块是4x4帧内预测类型（Intra4x4），作如下处理：

a)循环遍历16个4x4的块，并作如下处理：

i.从intra4x4_pred_mode_cache中读取4x4帧内预测方法

ii.根据帧内预测方法调用H264PredContext中的汇编函数pred4x4()进行帧内预测

iii.调用H264DSPContext中的汇编函数h264_idct_add()对DCT残差数据进行4x4DCT反变换；如果DCT系数中不包含AC系数的话，则调用汇编函数h264_idct_dc_add()对残差数据进行4x4DCT反变换（速度更快）。

（2）如果宏块是16x16帧内预测类型（Intra4x4），作如下处理：

a)通过intra16x16_pred_mode获得16x16帧内预测方法

b)根据帧内预测方法调用H264PredContext中的汇编函数pred16x16 ()进行帧内预测

c)调用H264DSPContext中的汇编函数h264_luma_dc_dequant_idct ()对16个小块的DC系数进行Hadamard反变换

在这里需要注意，帧内4x4的宏块在执行完hl_decode_mb_predict_luma()之后实际上已经完成了“帧内预测+DCT反变换”的流程（解码完成）；而帧内16x16的宏块在执行完hl_decode_mb_predict_luma()之后仅仅完成了“帧内预测+Hadamard反变换”的流程，而并未进行“DCT反变换”的步骤，这一步骤需要在后续步骤中完成。

下文记录上述流程中涉及到的汇编函数（此处暂不记录DCT反变换的函数，在后文中再进行叙述）：

4x4帧内预测汇编函数：H264PredContext -> pred4x4[dir]()

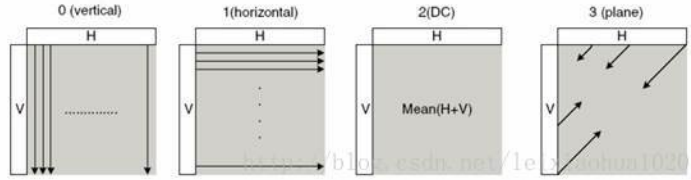
16x16帧内预测汇编函数：H264PredContext -> pred16x16[dir]()

Hadamard反变换汇编函数：H264DSPContext->h264_luma_dc_dequant_idct()

帧内预测小知识

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

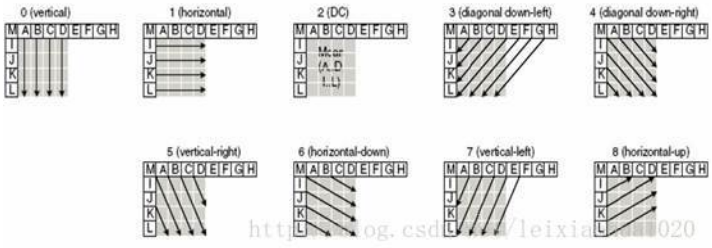
H.264中有两种帧内预测模式：16x16亮度帧内预测模式和4x4亮度帧内预测模式。其中16x16帧内预测模式一共有4种，如下图所示。



这4种模式列表如下。

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

4x4帧内预测模式一共有9种，如下图所示。



从图中可以看出，这9种模式中前4种和16x16帧内预测方法是一样的。后面增加了几种独特的方向——箭头不再位于“口”中，而是位于“日”中。

帧内预测汇编函数的初始化

FFmpeg H.264解码器中4x4帧内预测函数指针位于H264PredContext的pred4x4[]数组中，其中每一个元素指向一种4x4帧内预测模式。而16x16帧内预测函数指针位于H264PredContext的pred16x16[]数组中，其中每一个元素指向一种16x16帧内预测模式。

在FFmpeg H.264解码器初始化的时候，会调用ff_h264_pred_init()根据系统的配置对H264PredContext中的这些帧内预测函数指针进行赋值。下面简单看一下ff_h264_pred_init()的定义。

ff_h264_pred_init()

ff_h264_pred_init()的定义位于libavcodec\h264pred.c，如下所示。

```cpp
/**
 * Set the intra prediction function pointers.
 */
//初始化帧内预测相关的汇编函数
av_cold void ff_h264_pred_init(H264PredContext *h, int codec_id,
                               const int bit_depth,
                               int chroma_format_idc)
{
#undef FUNC
#undef FUNCC
#define FUNC(a, depth) a ## _ ## depth
#define FUNCC(a, depth) a ## _ ## depth ## _c
#define FUNCD(a) a ## _c
//好长的宏定义...（这种很长的宏定义在H.264解码器中似乎很普遍！）
//该宏用于给帧内预测模块的函数指针赋值
//注意参数为颜色位深度
#define H264_PRED(depth) \
    if(codec_id != AV_CODEC_ID_RV40){\
        if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {\
            h->pred4x4[VERT_PRED       ]= FUNCD(pred4x4_vertical_vp8);\
            h->pred4x4[HOR_PRED        ]= FUNCD(pred4x4_horizontal_vp8);\
        } else {\
            h->pred4x4[VERT_PRED       ]= FUNCC(pred4x4_vertical          , depth);\
            h->pred4x4[HOR_PRED        ]= FUNCC(pred4x4_horizontal        , depth);\
        }\
        h->pred4x4[DC_PRED             ]= FUNCC(pred4x4_dc                , depth);\
        if(codec_id == AV_CODEC_ID_SVQ3)\
            h->pred4x4[DIAG_DOWN_LEFT_PRED ]= FUNCD(pred4x4_down_left_svq3);\
        else\
            h->pred4x4[DIAG_DOWN_LEFT_PRED ]= FUNCC(pred4x4_down_left     , depth);\
        h->pred4x4[DIAG_DOWN_RIGHT_PRED]= FUNCC(pred4x4_down_right        , depth);\
        h->pred4x4[VERT_RIGHT_PRED     ]= FUNCC(pred4x4_vertical_right    , depth);\
        h->pred4x4[HOR_DOWN_PRED       ]= FUNCC(pred4x4_horizontal_down   , depth);\
        if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {\
            h->pred4x4[VERT_LEFT_PRED  ]= FUNCD(pred4x4_vertical_left_vp8);\
        } else\
            h->pred4x4[VERT_LEFT_PRED  ]= FUNCC(pred4x4_vertical_left     , depth);\
        h->pred4x4[HOR_UP_PRED         ]= FUNCC(pred4x4_horizontal_up     , depth);\
        if (codec_id != AV_CODEC_ID_VP7 && codec_id != AV_CODEC_ID_VP8) {\
            h->pred4x4[LEFT_DC_PRED    ]= FUNCC(pred4x4_left_dc           , depth);\
            h->pred4x4[TOP_DC_PRED     ]= FUNCC(pred4x4_top_dc            , depth);\
        } else {\
            h->pred4x4[TM_VP8_PRED     ]= FUNCD(pred4x4_tm_vp8);\
            h->pred4x4[DC_127_PRED     ]= FUNCC(pred4x4_127_dc            , depth);\
            h->pred4x4[DC_129_PRED     ]= FUNCC(pred4x4_129_dc            , depth);\
            h->pred4x4[VERT_VP8_PRED   ]= FUNCC(pred4x4_vertical          , depth);\
            h->pred4x4[HOR_VP8_PRED    ]= FUNCC(pred4x4_horizontal        , depth);\
        }\
        if (codec_id != AV_CODEC_ID_VP8)\
            h->pred4x4[DC_128_PRED     ]= FUNCC(pred4x4_128_dc            , depth);\
    }else{\
        h->pred4x4[VERT_PRED           ]= FUNCC(pred4x4_vertical          , depth);\
        h->pred4x4[HOR_PRED            ]= FUNCC(pred4x4_horizontal        , depth);\
        h->pred4x4[DC_PRED             ]= FUNCC(pred4x4_dc                , depth);\
        h->pred4x4[DIAG_DOWN_LEFT_PRED ]= FUNCD(pred4x4_down_left_rv40);\
        h->pred4x4[DIAG_DOWN_RIGHT_PRED]= FUNCC(pred4x4_down_right        , depth);\
        h->pred4x4[VERT_RIGHT_PRED     ]= FUNCC(pred4x4_vertical_right    , depth);\
        h->pred4x4[HOR_DOWN_PRED       ]= FUNCC(pred4x4_horizontal_down   , depth);\
        h->pred4x4[VERT_LEFT_PRED      ]= FUNCD(pred4x4_vertical_left_rv40);\
        h->pred4x4[HOR_UP_PRED         ]= FUNCD(pred4x4_horizontal_up_rv40);\
        h->pred4x4[LEFT_DC_PRED        ]= FUNCC(pred4x4_left_dc           , depth);\
        h->pred4x4[TOP_DC_PRED         ]= FUNCC(pred4x4_top_dc            , depth);\
        h->pred4x4[DC_128_PRED         ]= FUNCC(pred4x4_128_dc            , depth);\
        h->pred4x4[DIAG_DOWN_LEFT_PRED_RV40_NODOWN]= FUNCD(pred4x4_down_left_rv40_nodown);\
        h->pred4x4[HOR_UP_PRED_RV40_NODOWN]= FUNCD(pred4x4_horizontal_up_rv40_nodown);\
        h->pred4x4[VERT_LEFT_PRED_RV40_NODOWN]= FUNCD(pred4x4_vertical_left_rv40_nodown);\
    }\
\
    h->pred8x8l[VERT_PRED          ]= FUNCC(pred8x8l_vertical         , depth);\
    h->pred8x8l[HOR_PRED           ]= FUNCC(pred8x8l_horizontal       , depth);\
    h->pred8x8l[DC_PRED            ]= FUNCC(pred8x8l_dc               , depth);\
    h->pred8x8l[DIAG_DOWN_LEFT_PRED ]= FUNCC(pred8x8l_down_left       , depth);\
```

```
    h->pred8x8l[DIAG_DOWN_RIGHT_PRED]= FUNCC(pred8x8l_down_right          , depth);\
    h->pred8x8l[VERT_RIGHT_PRED     ]= FUNCC(pred8x8l_vertical_right      , depth);\
    h->pred8x8l[HOR_DOWN_PRED       ]= FUNCC(pred8x8l_horizontal_down     , depth);\
    h->pred8x8l[VERT_LEFT_PRED      ]= FUNCC(pred8x8l_vertical_left       , depth);\
    h->pred8x8l[HOR_UP_PRED         ]= FUNCC(pred8x8l_horizontal_up       , depth);\
    h->pred8x8l[LEFT_DC_PRED        ]= FUNCC(pred8x8l_left_dc             , depth);\
    h->pred8x8l[TOP_DC_PRED         ]= FUNCC(pred8x8l_top_dc              , depth);\
    h->pred8x8l[DC_128_PRED         ]= FUNCC(pred8x8l_128_dc              , depth);\
\
    if (chroma_format_idc <= 1) {\
        h->pred8x8[VERT_PRED8x8   ]= FUNCC(pred8x8_vertical               , depth);\
        h->pred8x8[HOR_PRED8x8    ]= FUNCC(pred8x8_horizontal             , depth);\
    } else {\
        h->pred8x8[VERT_PRED8x8   ]= FUNCC(pred8x16_vertical              , depth);\
        h->pred8x8[HOR_PRED8x8    ]= FUNCC(pred8x16_horizontal            , depth);\
    }\
    if (codec_id != AV_CODEC_ID_VP7 && codec_id != AV_CODEC_ID_VP8) {\
        if (chroma_format_idc <= 1) {\
            h->pred8x8[PLANE_PRED8x8]= FUNCC(pred8x8_plane                , depth);\
        } else {\
            h->pred8x8[PLANE_PRED8x8]= FUNCC(pred8x16_plane               , depth);\
        }\
    } else\
        h->pred8x8[PLANE_PRED8x8]= FUNCD(pred8x8_tm_vp8);\
    if (codec_id != AV_CODEC_ID_RV40 && codec_id != AV_CODEC_ID_VP7 && \
        codec_id != AV_CODEC_ID_VP8) {\
        if (chroma_format_idc <= 1) {\
            h->pred8x8[DC_PRED8x8     ]= FUNCC(pred8x8_dc                     , depth);\
            h->pred8x8[LEFT_DC_PRED8x8]= FUNCC(pred8x8_left_dc                , depth);\
            h->pred8x8[TOP_DC_PRED8x8 ]= FUNCC(pred8x8_top_dc                 , depth);\
            h->pred8x8[ALZHEIMER_DC_L0T_PRED8x8 ]= FUNC(pred8x8_mad_cow_dc_l0t, depth);\
            h->pred8x8[ALZHEIMER_DC_0LT_PRED8x8 ]= FUNC(pred8x8_mad_cow_dc_0lt, depth);\
            h->pred8x8[ALZHEIMER_DC_L00_PRED8x8 ]= FUNC(pred8x8_mad_cow_dc_l00, depth);\
            h->pred8x8[ALZHEIMER_DC_0L0_PRED8x8 ]= FUNC(pred8x8_mad_cow_dc_0l0, depth);\
        } else {\
            h->pred8x8[DC_PRED8x8     ]= FUNCC(pred8x16_dc                    , depth);\
            h->pred8x8[LEFT_DC_PRED8x8]= FUNCC(pred8x16_left_dc               , depth);\
            h->pred8x8[TOP_DC_PRED8x8 ]= FUNCC(pred8x16_top_dc                , depth);\
            h->pred8x8[ALZHEIMER_DC_L0T_PRED8x8 ]= FUNC(pred8x16_mad_cow_dc_l0t, depth);\
            h->pred8x8[ALZHEIMER_DC_0LT_PRED8x8 ]= FUNC(pred8x16_mad_cow_dc_0lt, depth);\
            h->pred8x8[ALZHEIMER_DC_L00_PRED8x8 ]= FUNC(pred8x16_mad_cow_dc_l00, depth);\
            h->pred8x8[ALZHEIMER_DC_0L0_PRED8x8 ]= FUNC(pred8x16_mad_cow_dc_0l0, depth);\
        }\
    }else{\
        h->pred8x8[DC_PRED8x8     ]= FUNCD(pred8x8_dc_rv40);\
        h->pred8x8[LEFT_DC_PRED8x8]= FUNCD(pred8x8_left_dc_rv40);\
        h->pred8x8[TOP_DC_PRED8x8 ]= FUNCD(pred8x8_top_dc_rv40);\
        if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {\
            h->pred8x8[DC_127_PRED8x8]= FUNCC(pred8x8_127_dc              , depth);\
            h->pred8x8[DC_129_PRED8x8]= FUNCC(pred8x8_129_dc              , depth);\
        }\
    }\
    if (chroma_format_idc <= 1) {\
        h->pred8x8[DC_128_PRED8x8 ]= FUNCC(pred8x8_128_dc                 , depth);\
    } else {\
        h->pred8x8[DC_128_PRED8x8 ]= FUNCC(pred8x16_128_dc                , depth);\
    }\
\
    h->pred16x16[DC_PRED8x8     ]= FUNCC(pred16x16_dc                     , depth);\
    h->pred16x16[VERT_PRED8x8   ]= FUNCC(pred16x16_vertical               , depth);\
    h->pred16x16[HOR_PRED8x8    ]= FUNCC(pred16x16_horizontal             , depth);\
    switch(codec_id){\
    case AV_CODEC_ID_SVQ3:\
       h->pred16x16[PLANE_PRED8x8  ]= FUNCD(pred16x16_plane_svq3);\
       break;\
    case AV_CODEC_ID_RV40:\
       h->pred16x16[PLANE_PRED8x8  ]= FUNCD(pred16x16_plane_rv40);\
       break;\
    case AV_CODEC_ID_VP7:\
    case AV_CODEC_ID_VP8:\
       h->pred16x16[PLANE_PRED8x8  ]= FUNCD(pred16x16_tm_vp8);\
       h->pred16x16[DC_127_PRED8x8]= FUNCC(pred16x16_127_dc              , depth);\
       h->pred16x16[DC_129_PRED8x8]= FUNCC(pred16x16_129_dc              , depth);\
       break;\
    default:\
       h->pred16x16[PLANE_PRED8x8  ]= FUNCC(pred16x16_plane              , depth);\
       break;\
    }\
    h->pred16x16[LEFT_DC_PRED8x8]= FUNCC(pred16x16_left_dc                , depth);\
    h->pred16x16[TOP_DC_PRED8x8 ]= FUNCC(pred16x16_top_dc                 , depth);\
    h->pred16x16[DC_128_PRED8x8 ]= FUNCC(pred16x16_128_dc                 , depth);\
\
    /* special lossless h/v prediction for h264 */ \
    h->pred4x4_add  [VERT_PRED   ]= FUNCC(pred4x4_vertical_add            , depth);\
    h->pred4x4_add  [ HOR_PRED   ]= FUNCC(pred4x4_horizontal_add          , depth);\
    h->pred8x8l_add [VERT_PRED   ]= FUNCC(pred8x8l_vertical_add           , depth);\
    h->pred8x8l_add [ HOR_PRED   ]= FUNCC(pred8x8l_horizontal_add         , depth);\
    h->pred8x8l_filter_add [VERT_PRED   ]= FUNCC(pred8x8l_vertical_filter_add          , depth);\
    h->pred8x8l_filter_add [ HOR_PRED   ]= FUNCC(pred8x8l_horizontal_filter_add        , depth);\
    if (chroma_format_idc <= 1) {\
    h->pred8x8_add  [VERT_PRED8x8]= FUNCC(pred8x8_vertical_add            , depth);\
```

```
    h->pred8x8_add  [ HOR_PRED8x8]= FUNCC(pred8x8_horizontal_add          , depth);\
    } else {\
        h->pred8x8_add  [VERT_PRED8x8]= FUNCC(pred8x16_vertical_add           , depth);\
        h->pred8x8_add  [ HOR_PRED8x8]= FUNCC(pred8x16_horizontal_add         , depth);\
    }\
    h->pred16x16_add[VERT_PRED8x8]= FUNCC(pred16x16_vertical_add          , depth);\
    h->pred16x16_add[ HOR_PRED8x8]= FUNCC(pred16x16_horizontal_add        , depth);\
    //注意这里使用了前面那个很长的宏定义
    //根据颜色位深的不同，初始化不同的函数
    //颜色位深默认值为8，所以一般情况下调用H264_PRED(8)
    switch (bit_depth) {
        case 9:
            H264_PRED(9)
            break;
        case 10:
            H264_PRED(10)
            break;
        case 12:
            H264_PRED(12)
            break;
        case 14:
            H264_PRED(14)
            break;
        default:
            av_assert0(bit_depth<=8);
            H264_PRED(8)
            break;
    }


    //如果支持汇编优化，则会调用相应的汇编优化函数
    //neon这些的
    if (ARCH_ARM) ff_h264_pred_init_arm(h, codec_id, bit_depth, chroma_format_idc);
    //mmx这些的
    if (ARCH_X86) ff_h264_pred_init_x86(h, codec_id, bit_depth, chroma_format_idc);
}
```

从源代码可以看出，ff_h264_pred_init()函数中包含一个名为“H264_PRED(depth)”的很长的宏定义。该宏定义中包含了C语言版本的帧内预测函数的初始化代码。ff_h264_pred_init()会根据系统的颜色位深bit_depth初始化相应的C语言版本的帧内预测函数。在函数的末尾则包含了汇编函数的初始化函数：如果系统是ARM架构的，则会调用ff_h264_pred_init_arm()初始化ARM平台下经过汇编优化的帧内预测函数；如果系统是X86架构的，则会调用ff_h264_pred_init_x86()初始化X86平台下经过汇编优化的帧内预测函数。
下面看一下C语言版本的帧内预测函数。

C语言版本帧内预测函数

“H264_PRED(depth)”宏用于初始化C语言版本的帧内预测函数。其中“depth”表示颜色位深。以最常见的8bit位深为例，展开“H264_PRED(8)”宏定义之后的代码如下所示。

```cpp
if(codec_id != AV_CODEC_ID_RV40){
    if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {
        h->pred4x4[0       ]= pred4x4_vertical_vp8_c;
        h->pred4x4[1        ]= pred4x4_horizontal_vp8_c;
    } else {
        //帧内4x4的Vertical预测方式
        h->pred4x4[0       ]= pred4x4_vertical_8_c;
        //帧内4x4的Horizontal预测方式
        h->pred4x4[1        ]= pred4x4_horizontal_8_c;
    }
    //帧内4x4的DC预测方式
    h->pred4x4[2             ]= pred4x4_dc_8_c;
    if(codec_id == AV_CODEC_ID_SVQ3)
        h->pred4x4[3 ]= pred4x4_down_left_svq3_c;
    else
        h->pred4x4[3 ]= pred4x4_down_left_8_c;
    h->pred4x4[4]= pred4x4_down_right_8_c;
    h->pred4x4[5     ]= pred4x4_vertical_right_8_c;
    h->pred4x4[6       ]= pred4x4_horizontal_down_8_c;
    if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {
        h->pred4x4[7  ]= pred4x4_vertical_left_vp8_c;
    } else
        h->pred4x4[7  ]= pred4x4_vertical_left_8_c;
    h->pred4x4[8         ]= pred4x4_horizontal_up_8_c;
    if (codec_id != AV_CODEC_ID_VP7 && codec_id != AV_CODEC_ID_VP8) {
        h->pred4x4[9    ]= pred4x4_left_dc_8_c;
        h->pred4x4[10     ]= pred4x4_top_dc_8_c;
    } else {
        h->pred4x4[9     ]= pred4x4_tm_vp8_c;
        h->pred4x4[12     ]= pred4x4_127_dc_8_c;
        h->pred4x4[13     ]= pred4x4_129_dc_8_c;
        h->pred4x4[10   ]= pred4x4_vertical_8_c;
        h->pred4x4[14    ]= pred4x4_horizontal_8_c;
    }
    if (codec_id != AV_CODEC_ID_VP8)
        h->pred4x4[11     ]= pred4x4_128_dc_8_c;
}else{
```

```
}else{
    h->pred4x4[0           ]= pred4x4_vertical_8_c;
    h->pred4x4[1            ]= pred4x4_horizontal_8_c;
    h->pred4x4[2             ]= pred4x4_dc_8_c;
    h->pred4x4[3 ]= pred4x4_down_left_rv40_c;
    h->pred4x4[4]= pred4x4_down_right_8_c;
    h->pred4x4[5     ]= pred4x4_vertical_right_8_c;
    h->pred4x4[6       ]= pred4x4_horizontal_down_8_c;
    h->pred4x4[7      ]= pred4x4_vertical_left_rv40_c;
    h->pred4x4[8         ]= pred4x4_horizontal_up_rv40_c;
    h->pred4x4[9        ]= pred4x4_left_dc_8_c;
    h->pred4x4[10         ]= pred4x4_top_dc_8_c;
    h->pred4x4[11         ]= pred4x4_128_dc_8_c;
    h->pred4x4[12]= pred4x4_down_left_rv40_nodown_c;
    h->pred4x4[13]= pred4x4_horizontal_up_rv40_nodown_c;
    h->pred4x4[14]= pred4x4_vertical_left_rv40_nodown_c;
}

h->pred8x8l[0           ]= pred8x8l_vertical_8_c;
h->pred8x8l[1            ]= pred8x8l_horizontal_8_c;
h->pred8x8l[2             ]= pred8x8l_dc_8_c;
h->pred8x8l[3 ]= pred8x8l_down_left_8_c;
h->pred8x8l[4]= pred8x8l_down_right_8_c;
h->pred8x8l[5     ]= pred8x8l_vertical_right_8_c;
h->pred8x8l[6       ]= pred8x8l_horizontal_down_8_c;
h->pred8x8l[7      ]= pred8x8l_vertical_left_8_c;
h->pred8x8l[8         ]= pred8x8l_horizontal_up_8_c;
h->pred8x8l[9        ]= pred8x8l_left_dc_8_c;
h->pred8x8l[10         ]= pred8x8l_top_dc_8_c;
h->pred8x8l[11         ]= pred8x8l_128_dc_8_c;

if (chroma_format_idc <= 1) {
    h->pred8x8[2   ]= pred8x8_vertical_8_c;
    h->pred8x8[1    ]= pred8x8_horizontal_8_c;
} else {
    h->pred8x8[2   ]= pred8x16_vertical_8_c;
    h->pred8x8[1    ]= pred8x16_horizontal_8_c;
}
if (codec_id != AV_CODEC_ID_VP7 && codec_id != AV_CODEC_ID_VP8) {
    if (chroma_format_idc <= 1) {
        h->pred8x8[3]= pred8x8_plane_8_c;
    } else {
        h->pred8x8[3]= pred8x16_plane_8_c;
    }
} else
    h->pred8x8[3]= pred8x8_tm_vp8_c;
if (codec_id != AV_CODEC_ID_RV40 && codec_id != AV_CODEC_ID_VP7 &&
    codec_id != AV_CODEC_ID_VP8) {
    if (chroma_format_idc <= 1) {
        h->pred8x8[0     ]= pred8x8_dc_8_c;
        h->pred8x8[4]= pred8x8_left_dc_8_c;
        h->pred8x8[5 ]= pred8x8_top_dc_8_c;
        h->pred8x8[7 ]= pred8x8_mad_cow_dc_l0t_8;
        h->pred8x8[8 ]= pred8x8_mad_cow_dc_0lt_8;
        h->pred8x8[9 ]= pred8x8_mad_cow_dc_l00_8;
        h->pred8x8[10 ]= pred8x8_mad_cow_dc_0l0_8;
    } else {
        h->pred8x8[0     ]= pred8x16_dc_8_c;
        h->pred8x8[4]= pred8x16_left_dc_8_c;
        h->pred8x8[5 ]= pred8x16_top_dc_8_c;
        h->pred8x8[7 ]= pred8x16_mad_cow_dc_l0t_8;
        h->pred8x8[8 ]= pred8x16_mad_cow_dc_0lt_8;
        h->pred8x8[9 ]= pred8x16_mad_cow_dc_l00_8;
        h->pred8x8[10 ]= pred8x16_mad_cow_dc_0l0_8;
    }
}else{
    h->pred8x8[0     ]= pred8x8_dc_rv40_c;
    h->pred8x8[4]= pred8x8_left_dc_rv40_c;
    h->pred8x8[5 ]= pred8x8_top_dc_rv40_c;
    if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {
        h->pred8x8[7]= pred8x8_127_dc_8_c;
        h->pred8x8[8]= pred8x8_129_dc_8_c;
    }
}
if (chroma_format_idc <= 1) {
    h->pred8x8[6 ]= pred8x8_128_dc_8_c;
} else {
    h->pred8x8[6 ]= pred8x16_128_dc_8_c;
}

h->pred16x16[0     ]= pred16x16_dc_8_c;
h->pred16x16[2   ]= pred16x16_vertical_8_c;
h->pred16x16[1    ]= pred16x16_horizontal_8_c;
switch(codec_id){
case AV_CODEC_ID_SVQ3:
   h->pred16x16[3  ]= pred16x16_plane_svq3_c;
   break;
case AV_CODEC_ID_RV40:
   h->pred16x16[3  ]= pred16x16_plane_rv40_c;
   break;
case AV_CODEC_ID_VP7:
case AV_CODEC_ID_VP8:
```

```
    h->pred16x16[3  ]= pred16x16_tm_vp8_c;
    h->pred16x16[7]= pred16x16_127_dc_8_c;
    h->pred16x16[8]= pred16x16_129_dc_8_c;
    break;
default:
    h->pred16x16[3  ]= pred16x16_plane_8_c;
    break;
}
h->pred16x16[4]= pred16x16_left_dc_8_c;
h->pred16x16[5 ]= pred16x16_top_dc_8_c;
h->pred16x16[6 ]= pred16x16_128_dc_8_c;

/* special lossless h/v prediction for h264 */
h->pred4x4_add  [0   ]= pred4x4_vertical_add_8_c;
h->pred4x4_add  [ 1   ]= pred4x4_horizontal_add_8_c;
h->pred8x8l_add [0   ]= pred8x8l_vertical_add_8_c;
h->pred8x8l_add [ 1   ]= pred8x8l_horizontal_add_8_c;
h->pred8x8l_filter_add [0   ]= pred8x8l_vertical_filter_add_8_c;
h->pred8x8l_filter_add [ 1   ]= pred8x8l_horizontal_filter_add_8_c;
if (chroma_format_idc <= 1) {
h->pred8x8_add  [2]= pred8x8_vertical_add_8_c;
h->pred8x8_add  [ 1]= pred8x8_horizontal_add_8_c;
} else {
    h->pred8x8_add  [2]= pred8x16_vertical_add_8_c;
    h->pred8x8_add  [ 1]= pred8x16_horizontal_add_8_c;
}
h->pred16x16_add[2]= pred16x16_vertical_add_8_c;
h->pred16x16_add[ 1]= pred16x16_horizontal_add_8_c;
```

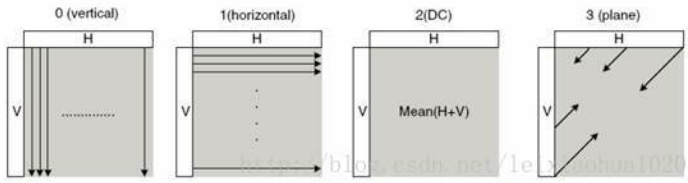
可以看出在H264_PRED(8)展开后的代码中，帧内预测模块的函数指针都被赋值以xxxx_8_c()的函数。例如pred4x4[0]（帧内4x4的模式0）被赋值以pred4x4_vertical_8_c()；pred4x4[1]（帧内4x4的模式1）被赋值以pred4x4_horizontal_8_c()；pred4x4[2]（帧内4x4的模式2）被赋值以pred4x4_cd_8_c()，如下所示。

```cpp
//帧内4x4的Vertical预测方式
h->pred4x4[0]= pred4x4_vertical_8_c;
//帧内4x4的Horizontal预测方式
h->pred4x4[1]= pred4x4_horizontal_8_c;
//帧内4x4的DC预测方式
h->pred4x4[2]= pred4x4_dc_8_c;
```

下面看一下这些4x4帧内预测函数的代码。

4x4帧内预测汇编函数：H264PredContext -> pred4x4[dir]()

4x4帧内预测函数指针位于H264PredContext的pred4x4[]数组中。pred4x4[]数组中每一个元素指向一种帧内预测模式。上文中提到的3个帧内预测函数实现的帧内预测功能如下图所示。



下面分别看看上文提到的3个4x4帧内预测函数。

pred4x4_vertical_8_c()

pred4x4_vertical_8_c()实现了4x4块Vertical模式的帧内预测，该函数的定义位于libavcodec\h264pred_template.c，如下所示。

```cpp
/* 帧内预测
 *
 * 注释：雷霄骅
 * leixiaohua1020@126.com
 * http://blog.csdn.net/leixiaohua1020
 *
 * 参数：
 * _src：输入数据
 * _stride：一行像素的大小
 *
 */
//垂直预测
//由上边像素推出像素值
static void FUNCC(pred4x4_vertical)(uint8_t *_src, const uint8_t *topright,
                                    ptrdiff_t _stride)
{
    pixel *src = (pixel*)_src;
    int stride = _stride>>(sizeof(pixel)-1);

    /*
     * Vertical预测方式
     *   |X1 X2 X3 X4
     * --+-----------
     *   |X1 X2 X3 X4
     *   |X1 X2 X3 X4
     *   |X1 X2 X3 X4
     *   |X1 X2 X3 X4
     *
     */

    //pixel4代表4个像素值。1个像素值占用8bit，4个像素值占用32bit。
    const pixel4 a= AV_RN4PA(src-stride);
    /* 宏定义展开后：
     * const uint32_t a=(((const av_alias32*)(src-stride))->u32);
     * 注：av_alias32是一个union类型的变量，存储4byte数据。
     * -stride代表了上一行对应位置的像素
     * 即a取的是上1行像素的值。
     */
    AV_WN4PA(src+0*stride, a);
    AV_WN4PA(src+1*stride, a);
    AV_WN4PA(src+2*stride, a);
    AV_WN4PA(src+3*stride, a);

    /* 宏定义展开后：
     * (((av_alias32*)(src+0*stride))->u32 = (a));
     * (((av_alias32*)(src+1*stride))->u32 = (a));
     * (((av_alias32*)(src+2*stride))->u32 = (a));
     * (((av_alias32*)(src+3*stride))->u32 = (a));
     * 即把a的值赋给下面4行。
     */

}
```

从源代码可以看出，pred4x4_vertical_8_c()首先取了当前4x4块上一行的4个像素存入a变量，然后将a变量的值分别赋值给了当前块的4行。在这一有一点要注意：stride代表了一行像素的大小，“src+stride”代表了位于src正下方的像素。

pred4x4_horizontal_8_c()

pred4x4_horizontal_8_c()实现了4x4块Horizontal模式的帧内预测，该函数的定义位于libavcodec\h264pred_template.c，如下所示。

```cpp
//水平预测
//由左边像素推出像素值
static void FUNCC(pred4x4_horizontal)(uint8_t *_src, const uint8_t *topright,
                                      ptrdiff_t _stride)
{
    pixel *src = (pixel*)_src;
    int stride = _stride>>(sizeof(pixel)-1);

    /*
     * Horizontal预测方式
     *   |
     * --+-----------
     * X5|X5 X5 X5 X5
     * X6|X6 X6 X6 X6
     * X7|X7 X7 X7 X7
     * X8|X8 X8 X8 X8
     *
     */

    AV_WN4PA(src+0*stride, PIXEL_SPLAT_X4(src[-1+0*stride]));
    AV_WN4PA(src+1*stride, PIXEL_SPLAT_X4(src[-1+1*stride]));
    AV_WN4PA(src+2*stride, PIXEL_SPLAT_X4(src[-1+2*stride]));
    AV_WN4PA(src+3*stride, PIXEL_SPLAT_X4(src[-1+3*stride]));
    /* 宏定义展开后：
     * (((av_alias32*)(src+0*stride))->u32 = (((src[-1+0*stride])*0x01010101U)));
     * (((av_alias32*)(src+1*stride))->u32 = (((src[-1+1*stride])*0x01010101U)));
     * (((av_alias32*)(src+2*stride))->u32 = (((src[-1+2*stride])*0x01010101U)));
     * (((av_alias32*)(src+3*stride))->u32 = (((src[-1+3*stride])*0x01010101U)));
     *
     * PIXEL_SPLAT_X4()的作用应该是把最后一个像素（最后8位）拷贝给前面3个像素（前24位）
     * 即把0x0100009F变成0x9F9F9F9F
     * 推导：
     * 前提是x占8bit（对应1个像素）
     * y=x*0x01010101
     *  =x*(0x00000001+0x00000100+0x00010000+0x01000000)
     *  =x<<0+x<<8+x<<16+x<<24
     *
     * 每行把src[-1]中像素值例如0x02赋值给src[0]开始的4个像素中，形成0x02020202
     */
}
```

从源代码可以看出，pred4x4_horizontal_8_c()将4x4块每行像素左边的一个像素拷贝了4份之后赋值给了当前行。其中PIXEL_SPLAT_X4()宏的定义如下：

```cpp
#   define PIXEL_SPLAT_X4(x) ((x)*0x01010101U)
```

经过研究后发现该宏用于将32bit数据的最后8位复制3份分别赋值到原数据的8-16位、16-24位以及24-32位，即将低位8bit数据“复制”3份到高位上。详细的推导过程已经写在了代码注释中，就不再重复了。

pred4x4_dc_8_c()

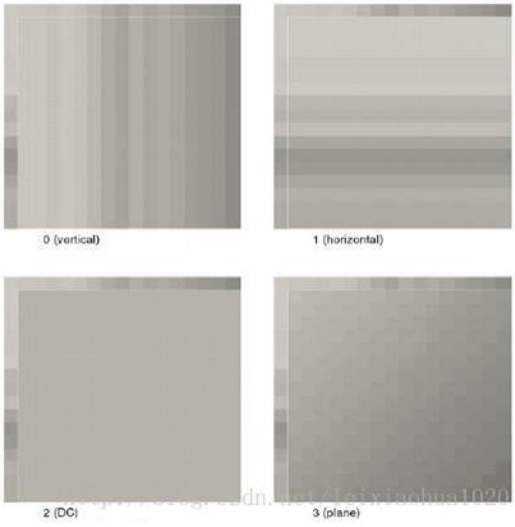
pred4x4_dc_8_c()实现了4x4块DC模式的帧内预测，该函数的定义位于libavcodec\h264pred_template.c，如下所示。

```cpp
//DC预测
//由左边和上边像素平均值推出像素值
static void FUNCC(pred4x4_dc)(uint8_t *_src, const uint8_t *topright,
                              ptrdiff_t _stride)
{
    pixel *src = (pixel*)_src;
    int stride = _stride>>(sizeof(pixel)-1);
    /*
     * DC预测方式
     *   |X1 X2 X3 X4
     * --+-----------
     * X5|
     * X6|     Y
     * X7|
     * X8|
     *
     * Y=(X1+X2+X3+X4+X5+X6+X7+X8)/8
     */
    const int dc= (  src[-stride] + src[1-stride] + src[2-stride] + src[3-stride]
                   + src[-1+0*stride] + src[-1+1*stride] + src[-1+2*stride] + src[-1+3*stride] + 4) >>3;
    const pixel4 a = PIXEL_SPLAT_X4(dc);

    AV_WN4PA(src+0*stride, a);
    AV_WN4PA(src+1*stride, a);
    AV_WN4PA(src+2*stride, a);
    AV_WN4PA(src+3*stride, a);
    /* 宏定义展开后：
     * (((av_alias32*)(src+0*stride))->u32 = (a))
     * (((av_alias32*)(src+1*stride))->u32 = (a))
     * (((av_alias32*)(src+2*stride))->u32 = (a))
     * (((av_alias32*)(src+3*stride))->u32 = (a))
     */
}
```

从源代码可以看出，pred4x4_dc_8_c()将4x4块左边和上边8个点的像素值相加后取了平均值，然后赋值到该4x4块中的所有像素点上。
分析完4x4帧内预测模式的C语言函数之后，我们再看一下16x16帧内预测模式的C语言函数。

16x16帧内预测汇编函数：H264PredContext -> pred16x16[dir] ()

16x16帧内预测模式一共有4种。它们的效果和4x4帧内预测是类似的，如下所示。



pred16x16_vertical_8_c()

下面举例看一个16x16帧内预测Vertical模式的C语言函数pred16x16_vertical_8_c()，如下所示。

```cpp
//垂直预测
//由上面的函数推出像素值
static void FUNCC(pred16x16_vertical)(uint8_t *_src, ptrdiff_t _stride)
{
    /*
     * Vertical预测方式
     *   |X1 X2 X3 X4
     * --+-----------
     *   |X1 X2 X3 X4
     *   |X1 X2 X3 X4
     *   |X1 X2 X3 X4
     *   |X1 X2 X3 X4
     *
     */
    int i;
    pixel *src = (pixel*)_src;
    int stride = _stride>>(sizeof(pixel)-1);
    //pixel4实际上就是uint32_t，存储4个像素的值（每个像素8bit）
    //src-stride表示取上面一行像素的值
    //在这里取了16个像素的值，分别存入a，b，c，d四个变量
    const pixel4 a = AV_RN4PA(((pixel4*)(src-stride))+0);
    const pixel4 b = AV_RN4PA(((pixel4*)(src-stride))+1);
    const pixel4 c = AV_RN4PA(((pixel4*)(src-stride))+2);
    const pixel4 d = AV_RN4PA(((pixel4*)(src-stride))+3);
    //循环16行
    for(i=0; i<16; i++){
        //分别赋值每行（每次赋值4个像素，赋值4次）
        AV_WN4PA(((pixel4*)(src+i*stride))+0, a);
        AV_WN4PA(((pixel4*)(src+i*stride))+1, b);
        AV_WN4PA(((pixel4*)(src+i*stride))+2, c);
        AV_WN4PA(((pixel4*)(src+i*stride))+3, d);
    }
}
```

可以看出pred16x16_vertical_8_c()首先取了16x16块上面的一行像素（16个像素），然后循环16次分别赋值给了宏块的16行。

Hadamard反变换汇编函数：H264DSPContext->h264_luma_dc_dequant_idct()

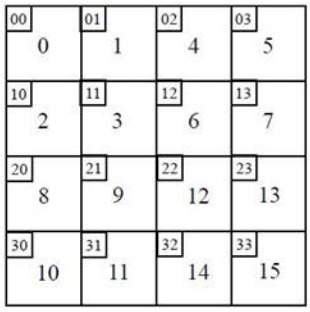
在记录Hadamard反变换的源代码之前，先简单记录Hadamard变换的原理。

Hadamard变换小知识

在H.264标准中，如果当前处理的图像宏块是色度块或帧内 16x16模式的亮度块，则需要在DCT变换后将其中各图像块的DCT变换系数矩阵W 中的DC系数按对应图像块顺序排序，组成新的矩阵Wd，再对Wd进行Hadamard 变换及量化。Hadamard变换的公式如下所示。

$$Y_D = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} W_D \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right) /2$$

Hadamard变换的取值方法如图所示。16x16的亮度块中有16 个4x4 图像亮度块（16个大方块），每个4x4亮度块都包含了一个DC系数（每个大方块左上角的小方块）。在编码的过程中，需要将00，01，02…等一共16个系数组成一个4x4的矩阵再次进行Hadamard变换。

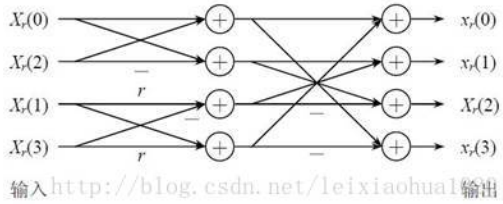


因此在解码帧内16x16亮度块的时候，需要在DCT反变换前先对DC系数进行Hadamard反变换。Hadamard反变换的公式如下所示。

$$W_{QD} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} Z_{rD} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix}$$

可以将该矩阵乘法改造成为2次一维变换，例如先对量化矩阵的每行进行一维变换，然后对经行变换所得数据块的每列再应用一维变换。而一维变换又可以采用蝶形快

速算法，如下图所示。



图中r=1的时候即Hadamard反变换（r=1/2是逆整数DCT 变换，后文再详细叙述）。

Hadamard反变换的C语言版本函数

FFmpeg H.264解码器中Hadamard反变换的函数指针是H264DSPContext的h264_luma_dc_dequant_idct()。在初始化的时候，其指向ff_h264_chroma_dc_dequant_idct_8_c()函数。下面看一下该函数的定义。

ff_h264_chroma_dc_dequant_idct_8_c()

ff_h264_chroma_dc_dequant_idct_8_c()完成了Hadamard反变换，其定义位于libavcodec\h264idct_template.c，如下所示。

```cpp
/**
 * IDCT transforms the 16 dc values and dequantizes them.
 * @param qmul quantization parameter
 */
//DCT直流系数的Hadamard反变换-亮度
//16x16宏块中一共有16个4x4的图像块，因此包含了16个DCT直流系数
//
void FUNCC(ff_h264_luma_dc_dequant_idct)(int16_t *_output, int16_t *_input, int qmul){
#define stride 16
    int i;
    int temp[16];
    static const uint8_t x_offset[4]={0, 2*stride, 8*stride, 10*stride};
    dctcoef *input = (dctcoef*)_input;
    dctcoef *output = (dctcoef*)_output;

    for(i=0; i<4; i++){
        const int z0= input[4*i+0] + input[4*i+1];
        const int z1= input[4*i+0] - input[4*i+1];
        const int z2= input[4*i+2] - input[4*i+3];
        const int z3= input[4*i+2] + input[4*i+3];

        temp[4*i+0]= z0+z3;
        temp[4*i+1]= z0-z3;
        temp[4*i+2]= z1-z2;
        temp[4*i+3]= z1+z2;
    }

    for(i=0; i<4; i++){
        const int offset= x_offset[i];
        const int z0= temp[4*0+i] + temp[4*2+i];
        const int z1= temp[4*0+i] - temp[4*2+i];
        const int z2= temp[4*1+i] - temp[4*3+i];
        const int z3= temp[4*1+i] + temp[4*3+i];

        output[stride* 0+offset]= ((((z0 + z3)*qmul + 128 ) >> 8));
        output[stride* 1+offset]= ((((z1 + z2)*qmul + 128 ) >> 8));
        output[stride* 4+offset]= ((((z1 - z2)*qmul + 128 ) >> 8));
        output[stride* 5+offset]= ((((z0 - z3)*qmul + 128 ) >> 8));
    }
#undef stride
}
```

从源代码可以看出，ff_h264_chroma_dc_dequant_idct_8_c()实现了Hadamard反变换的蝶形算法。

Intel汇编语言版本帧内预测函数：基于MMX指令集和SSE指令集

前文记录的都是C语言版本的帧内预测函数，作为对比，在此记录2个Intel汇编语言版本帧内预测函数ff_pred16x16_vertical_8_mmx()和ff_pred16x16_vertical_8_sse()。在记录该函数之前首先回顾一下前文中提到的帧内预测初始化函数ff_h264_pred_init()。该函数的末尾进行了一个判断：如果支持ARM架构，则会调用ff_h264_pred_init_arm()初始化ARM平台的汇编函数；如果支持X86架构，则会调用ff_h264_pred_init_x86()初始化X86平台的汇编函数。在这里以X86平台为例，看一下X86平台帧内预测初始化函数ff_h264_pred_init_x86()。

ff_h264_pred_init_x86()

ff_h264_pred_init_x86()的定义位于libavcodec\x86\h264_intrapred_init.c（注意位于libavcodec的子文件夹x86下），如下所示。

```cpp
av_cold void ff_h264_pred_init_x86(H264PredContext *h, int codec_id,
                                   const int bit_depth,
                                   const int chroma_format_idc)
{
    int cpu_flags = av_get_cpu_flags();

    if (bit_depth == 8) {
        if (EXTERNAL_MMX(cpu_flags)) {
            h->pred16x16[VERT_PRED8x8         ] = ff_pred16x16_vertical_8_mmx;
            h->pred16x16[HOR_PRED8x8          ] = ff_pred16x16_horizontal_8_mmx;
            if (chroma_format_idc <= 1) {
                h->pred8x8  [VERT_PRED8x8     ] = ff_pred8x8_vertical_8_mmx;
                h->pred8x8  [HOR_PRED8x8      ] = ff_pred8x8_horizontal_8_mmx;
            }
            if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {
                h->pred16x16[PLANE_PRED8x8    ] = ff_pred16x16_tm_vp8_8_mmx;
                h->pred8x8  [PLANE_PRED8x8    ] = ff_pred8x8_tm_vp8_8_mmx;
                h->pred4x4  [TM_VP8_PRED      ] = ff_pred4x4_tm_vp8_8_mmx;
            } else {
                if (chroma_format_idc <= 1)
                    h->pred8x8  [PLANE_PRED8x8] = ff_pred8x8_plane_8_mmx;
                if (codec_id == AV_CODEC_ID_SVQ3) {
                    if (cpu_flags & AV_CPU_FLAG_CMOV)
                        h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_svq3_8_mmx;
                } else if (codec_id == AV_CODEC_ID_RV40) {
                    h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_rv40_8_mmx;
                } else {
                    h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_h264_8_mmx;
                }
            }
        }

        if (EXTERNAL_MMXEXT(cpu_flags)) {
            h->pred16x16[HOR_PRED8x8            ] = ff_pred16x16_horizontal_8_mmxext;
            h->pred16x16[DC_PRED8x8             ] = ff_pred16x16_dc_8_mmxext;
            if (chroma_format_idc <= 1)
                h->pred8x8[HOR_PRED8x8          ] = ff_pred8x8_horizontal_8_mmxext;
            h->pred8x8l [TOP_DC_PRED            ] = ff_pred8x8l_top_dc_8_mmxext;
            h->pred8x8l [DC_PRED                ] = ff_pred8x8l_dc_8_mmxext;
            h->pred8x8l [HOR_PRED               ] = ff_pred8x8l_horizontal_8_mmxext;
            h->pred8x8l [VERT_PRED              ] = ff_pred8x8l_vertical_8_mmxext;
            h->pred8x8l [DIAG_DOWN_RIGHT_PRED   ] = ff_pred8x8l_down_right_8_mmxext;
            h->pred8x8l [VERT_RIGHT_PRED        ] = ff_pred8x8l_vertical_right_8_mmxext;
            h->pred8x8l [HOR_UP_PRED            ] = ff_pred8x8l_horizontal_up_8_mmxext;
            h->pred8x8l [DIAG_DOWN_LEFT_PRED    ] = ff_pred8x8l_down_left_8_mmxext;
            h->pred8x8l [HOR_DOWN_PRED          ] = ff_pred8x8l_horizontal_down_8_mmxext;
            h->pred4x4  [DIAG_DOWN_RIGHT_PRED   ] = ff_pred4x4_down_right_8_mmxext;
            h->pred4x4  [VERT_RIGHT_PRED        ] = ff_pred4x4_vertical_right_8_mmxext;
            h->pred4x4  [HOR_DOWN_PRED          ] = ff_pred4x4_horizontal_down_8_mmxext;
            h->pred4x4  [DC_PRED                ] = ff_pred4x4_dc_8_mmxext;
            if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8 ||
                codec_id == AV_CODEC_ID_H264) {
                h->pred4x4  [DIAG_DOWN_LEFT_PRED] = ff_pred4x4_down_left_8_mmxext;
            }
            if (codec_id == AV_CODEC_ID_SVQ3 || codec_id == AV_CODEC_ID_H264) {
                h->pred4x4  [VERT_LEFT_PRED     ] = ff_pred4x4_vertical_left_8_mmxext;
            }
            if (codec_id != AV_CODEC_ID_RV40) {
                h->pred4x4  [HOR_UP_PRED        ] = ff_pred4x4_horizontal_up_8_mmxext;
            }
            if (codec_id == AV_CODEC_ID_SVQ3 || codec_id == AV_CODEC_ID_H264) {
                if (chroma_format_idc <= 1) {
                    h->pred8x8[TOP_DC_PRED8x8   ] = ff_pred8x8_top_dc_8_mmxext;
                    h->pred8x8[DC_PRED8x8       ] = ff_pred8x8_dc_8_mmxext;
                }
            }
            if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {
                h->pred16x16[PLANE_PRED8x8      ] = ff_pred16x16_tm_vp8_8_mmxext;
                h->pred8x8  [DC_PRED8x8         ] = ff_pred8x8_dc_rv40_8_mmxext;
                h->pred8x8  [PLANE_PRED8x8      ] = ff_pred8x8_tm_vp8_8_mmxext;
                h->pred4x4  [TM_VP8_PRED        ] = ff_pred4x4_tm_vp8_8_mmxext;
                h->pred4x4  [VERT_PRED          ] = ff_pred4x4_vertical_vp8_8_mmxext;
            } else {
                if (chroma_format_idc <= 1)
                    h->pred8x8  [PLANE_PRED8x8] = ff_pred8x8_plane_8_mmxext;
                if (codec_id == AV_CODEC_ID_SVQ3) {
                    h->pred16x16[PLANE_PRED8x8  ] = ff_pred16x16_plane_svq3_8_mmxext;
                } else if (codec_id == AV_CODEC_ID_RV40) {
                    h->pred16x16[PLANE_PRED8x8  ] = ff_pred16x16_plane_rv40_8_mmxext;
                } else {
                    h->pred16x16[PLANE_PRED8x8  ] = ff_pred16x16_plane_h264_8_mmxext;
                }
            }
        }

        if (EXTERNAL_SSE(cpu_flags)) {
            h->pred16x16[VERT_PRED8x8] = ff_pred16x16_vertical_8_sse;
```

```
        }

        if (EXTERNAL_SSE2(cpu_flags)) {
            h->pred16x16[DC_PRED8x8           ] = ff_pred16x16_dc_8_sse2;
            h->pred8x8l [DIAG_DOWN_LEFT_PRED  ] = ff_pred8x8l_down_left_8_sse2;
            h->pred8x8l [DIAG_DOWN_RIGHT_PRED ] = ff_pred8x8l_down_right_8_sse2;
            h->pred8x8l [VERT_RIGHT_PRED      ] = ff_pred8x8l_vertical_right_8_sse2;
            h->pred8x8l [VERT_LEFT_PRED       ] = ff_pred8x8l_vertical_left_8_sse2;
            h->pred8x8l [HOR_DOWN_PRED        ] = ff_pred8x8l_horizontal_down_8_sse2;
            if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {
                h->pred16x16[PLANE_PRED8x8    ] = ff_pred16x16_tm_vp8_8_sse2;
                h->pred8x8  [PLANE_PRED8x8    ] = ff_pred8x8_tm_vp8_8_sse2;
            } else {
                if (chroma_format_idc <= 1)
                    h->pred8x8  [PLANE_PRED8x8] = ff_pred8x8_plane_8_sse2;
                if (codec_id == AV_CODEC_ID_SVQ3) {
                    h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_svq3_8_sse2;
                } else if (codec_id == AV_CODEC_ID_RV40) {
                    h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_rv40_8_sse2;
                } else {
                    h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_h264_8_sse2;
                }
            }
        }

        if (EXTERNAL_SSSE3(cpu_flags)) {
            h->pred16x16[HOR_PRED8x8          ] = ff_pred16x16_horizontal_8_ssse3;
            h->pred16x16[DC_PRED8x8           ] = ff_pred16x16_dc_8_ssse3;
            if (chroma_format_idc <= 1)
                h->pred8x8  [HOR_PRED8x8      ] = ff_pred8x8_horizontal_8_ssse3;
            h->pred8x8l [TOP_DC_PRED          ] = ff_pred8x8l_top_dc_8_ssse3;
            h->pred8x8l [DC_PRED              ] = ff_pred8x8l_dc_8_ssse3;
            h->pred8x8l [HOR_PRED             ] = ff_pred8x8l_horizontal_8_ssse3;
            h->pred8x8l [VERT_PRED            ] = ff_pred8x8l_vertical_8_ssse3;
            h->pred8x8l [DIAG_DOWN_LEFT_PRED  ] = ff_pred8x8l_down_left_8_ssse3;
            h->pred8x8l [DIAG_DOWN_RIGHT_PRED ] = ff_pred8x8l_down_right_8_ssse3;
            h->pred8x8l [VERT_RIGHT_PRED      ] = ff_pred8x8l_vertical_right_8_ssse3;
            h->pred8x8l [VERT_LEFT_PRED       ] = ff_pred8x8l_vertical_left_8_ssse3;
            h->pred8x8l [HOR_UP_PRED          ] = ff_pred8x8l_horizontal_up_8_ssse3;
            h->pred8x8l [HOR_DOWN_PRED        ] = ff_pred8x8l_horizontal_down_8_ssse3;
            if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {
                h->pred8x8  [PLANE_PRED8x8    ] = ff_pred8x8_tm_vp8_8_ssse3;
                h->pred4x4  [TM_VP8_PRED      ] = ff_pred4x4_tm_vp8_8_ssse3;
            } else {
                if (chroma_format_idc <= 1)
                    h->pred8x8  [PLANE_PRED8x8] = ff_pred8x8_plane_8_ssse3;
                if (codec_id == AV_CODEC_ID_SVQ3) {
                    h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_svq3_8_ssse3;
                } else if (codec_id == AV_CODEC_ID_RV40) {
                    h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_rv40_8_ssse3;
                } else {
                    h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_h264_8_ssse3;
                }
            }
        }
    } else if (bit_depth == 10) {
        if (EXTERNAL_MMXEXT(cpu_flags)) {
            h->pred4x4[DC_PRED             ] = ff_pred4x4_dc_10_mmxext;
            h->pred4x4[HOR_UP_PRED         ] = ff_pred4x4_horizontal_up_10_mmxext;

            if (chroma_format_idc <= 1)
                h->pred8x8[DC_PRED8x8      ] = ff_pred8x8_dc_10_mmxext;

            h->pred8x8l[DC_128_PRED        ] = ff_pred8x8l_128_dc_10_mmxext;

            h->pred16x16[DC_PRED8x8        ] = ff_pred16x16_dc_10_mmxext;
            h->pred16x16[TOP_DC_PRED8x8    ] = ff_pred16x16_top_dc_10_mmxext;
            h->pred16x16[DC_128_PRED8x8    ] = ff_pred16x16_128_dc_10_mmxext;
            h->pred16x16[LEFT_DC_PRED8x8   ] = ff_pred16x16_left_dc_10_mmxext;
            h->pred16x16[VERT_PRED8x8      ] = ff_pred16x16_vertical_10_mmxext;
            h->pred16x16[HOR_PRED8x8       ] = ff_pred16x16_horizontal_10_mmxext;
        }
        if (EXTERNAL_SSE2(cpu_flags)) {
            h->pred4x4[DIAG_DOWN_LEFT_PRED ] = ff_pred4x4_down_left_10_sse2;
            h->pred4x4[DIAG_DOWN_RIGHT_PRED] = ff_pred4x4_down_right_10_sse2;
            h->pred4x4[VERT_LEFT_PRED      ] = ff_pred4x4_vertical_left_10_sse2;
            h->pred4x4[VERT_RIGHT_PRED     ] = ff_pred4x4_vertical_right_10_sse2;
            h->pred4x4[HOR_DOWN_PRED       ] = ff_pred4x4_horizontal_down_10_sse2;

            if (chroma_format_idc <= 1) {
                h->pred8x8[DC_PRED8x8      ] = ff_pred8x8_dc_10_sse2;
                h->pred8x8[TOP_DC_PRED8x8  ] = ff_pred8x8_top_dc_10_sse2;
                h->pred8x8[PLANE_PRED8x8   ] = ff_pred8x8_plane_10_sse2;
                h->pred8x8[VERT_PRED8x8    ] = ff_pred8x8_vertical_10_sse2;
                h->pred8x8[HOR_PRED8x8     ] = ff_pred8x8_horizontal_10_sse2;
            }

            h->pred8x8l[VERT_PRED           ] = ff_pred8x8l_vertical_10_sse2;
            h->pred8x8l[HOR_PRED            ] = ff_pred8x8l_horizontal_10_sse2;
            h->pred8x8l[DC_PRED             ] = ff_pred8x8l_dc_10_sse2;
            h->pred8x8l[DC_128_PRED         ] = ff_pred8x8l_128_dc_10_sse2;
            h->pred8x8l[TOP_DC_PRED         ] = ff_pred8x8l_top_dc_10_sse2;
```

```
            h->pred8x8l[TOP_DC_PRED         ] = ff_pred8x8l_top_dc_10_sse2;
            h->pred8x8l[DIAG_DOWN_LEFT_PRED ] = ff_pred8x8l_down_left_10_sse2;
            h->pred8x8l[DIAG_DOWN_RIGHT_PRED] = ff_pred8x8l_down_right_10_sse2;
            h->pred8x8l[VERT_RIGHT_PRED     ] = ff_pred8x8l_vertical_right_10_sse2;
            h->pred8x8l[HOR_UP_PRED         ] = ff_pred8x8l_horizontal_up_10_sse2;

            h->pred16x16[DC_PRED8x8        ] = ff_pred16x16_dc_10_sse2;
            h->pred16x16[TOP_DC_PRED8x8    ] = ff_pred16x16_top_dc_10_sse2;
            h->pred16x16[DC_128_PRED8x8    ] = ff_pred16x16_128_dc_10_sse2;
            h->pred16x16[LEFT_DC_PRED8x8   ] = ff_pred16x16_left_dc_10_sse2;
            h->pred16x16[VERT_PRED8x8      ] = ff_pred16x16_vertical_10_sse2;
            h->pred16x16[HOR_PRED8x8       ] = ff_pred16x16_horizontal_10_sse2;
        }
        if (EXTERNAL_SSSE3(cpu_flags)) {
            h->pred4x4[DIAG_DOWN_RIGHT_PRED] = ff_pred4x4_down_right_10_ssse3;
            h->pred4x4[VERT_RIGHT_PRED     ] = ff_pred4x4_vertical_right_10_ssse3;
            h->pred4x4[HOR_DOWN_PRED       ] = ff_pred4x4_horizontal_down_10_ssse3;

            h->pred8x8l[HOR_PRED            ] = ff_pred8x8l_horizontal_10_ssse3;
            h->pred8x8l[DIAG_DOWN_LEFT_PRED ] = ff_pred8x8l_down_left_10_ssse3;
            h->pred8x8l[DIAG_DOWN_RIGHT_PRED] = ff_pred8x8l_down_right_10_ssse3;
            h->pred8x8l[VERT_RIGHT_PRED     ] = ff_pred8x8l_vertical_right_10_ssse3;
            h->pred8x8l[HOR_UP_PRED         ] = ff_pred8x8l_horizontal_up_10_ssse3;
        }
        if (EXTERNAL_AVX(cpu_flags)) {
            h->pred4x4[DIAG_DOWN_LEFT_PRED ] = ff_pred4x4_down_left_10_avx;
            h->pred4x4[DIAG_DOWN_RIGHT_PRED] = ff_pred4x4_down_right_10_avx;
            h->pred4x4[VERT_LEFT_PRED      ] = ff_pred4x4_vertical_left_10_avx;
            h->pred4x4[VERT_RIGHT_PRED     ] = ff_pred4x4_vertical_right_10_avx;
            h->pred4x4[HOR_DOWN_PRED       ] = ff_pred4x4_horizontal_down_10_avx;

            h->pred8x8l[VERT_PRED           ] = ff_pred8x8l_vertical_10_avx;
            h->pred8x8l[HOR_PRED            ] = ff_pred8x8l_horizontal_10_avx;
            h->pred8x8l[DC_PRED             ] = ff_pred8x8l_dc_10_avx;
            h->pred8x8l[TOP_DC_PRED         ] = ff_pred8x8l_top_dc_10_avx;
            h->pred8x8l[DIAG_DOWN_RIGHT_PRED] = ff_pred8x8l_down_right_10_avx;
            h->pred8x8l[DIAG_DOWN_LEFT_PRED ] = ff_pred8x8l_down_left_10_avx;
            h->pred8x8l[VERT_RIGHT_PRED     ] = ff_pred8x8l_vertical_right_10_avx;
            h->pred8x8l[HOR_UP_PRED         ] = ff_pred8x8l_horizontal_up_10_avx;
        }
    }
}
```

从源代码可以看出，ff_h264_pred_init_x86()根据平台支持指令集的不同，将很多形如“XXX_mmx()”，“XXX_sse2()”，“XXX_ssse3()”，“XXX_avx()”的函数赋值给了H264PredContext中的帧内预测函数指针。在这里我们看2个针对16x16帧内预测Vertical模式优化的汇编函数：基于MMX指令集ff_pred16x16_vertical_8_mmx()和基于SSE指令集的ff_pred16x16_vertical_8_sse()。

ff_pred16x16_vertical_8_mmx()

ff_pred16x16_vertical_8_mmx()的函数定义位于libavcodec\x86\h264_intrapred.asm，如下所示。

```
;-----------------------------------------------------------------------------
; void ff_pred16x16_vertical_8(uint8_t *src, int stride)
; 注释：雷霄骅
; 16x16帧内预测-Vertical
;
; Vertical预测方式
;   |X1 X2 X3 X4
; --+-----------
;   |X1 X2 X3 X4
;   |X1 X2 X3 X4
;   |X1 X2 X3 X4
;   |X1 X2 X3 X4
;
;-----------------------------------------------------------------------------
;mmx指令优化
INIT_MMX mmx
cglobal pred16x16_vertical_8, 2,3
                            ;C语言调用汇编的时候，r0接收第1个参数（src），r1接收第2个参数（stride）......
    sub   r0, r1            ;r0=r0-r1。只有r0和r1可以作为地址寄存器，在这里里面存储的是地址（r2-r7不可以）
                            ;此时r0指向16x16块上面一行像素数据
    mov   r2, 8             ;r2=8;r2为循环计数器，每次循环减1，r2为0的时候，循环停止
    movq mm0, [r0+0]        ;类似于memcpy(&mm0,r0,8)。movq传递64bit（8字节，对应8像素）数据。“[]”代表传送r0地址的数据.
                            ;即将宏块上面1行像素中前8个像素传入mm0（用于循环中的赋值）
                            ;注：MOV-1~2字节（word），MOVD-4字节（Dword），MOVQ-8字节（Qword）
    movq mm1, [r0+8]        ;类似于memcpy(&mm1,r0+8,8)。2次movq传递128bit（16个像素）。mm0和mm1中存储了宏块上方一行像素的值
                            ;即将宏块上面1行像素中后8个像素传入mm1
.loop:                      ;循环
    movq [r0+r1*1+0], mm0  ;类似于memcpy(r0+r1,&mm0,8)。第1次循环，拷贝mm0至宏块第1行前8个像素。
    movq [r0+r1*1+8], mm1  ;类似于memcpy(r0+r1+8,&mm1,8)。第1次循环，拷贝mm1至宏块第1行后8个像素。
    movq [r0+r1*2+0], mm0  ;类似于memcpy(r0+r1*2,&mm0,8)。第1次循环，拷贝mm0至宏块第2行前8个像素。
    movq [r0+r1*2+8], mm1  ;类似于memcpy(r0+r1*2+8,&mm1,8)。第1次循环，拷贝mm1至宏块第2行后8个像素。
                            ;总而言之，一次处理2行，16行像素一共处理8次
    lea   r0, [r0+r1*2]    ;r0=r0+r1*2。r0前移2行。注意“lea”是传送地址的指令
    dec   r2                ;r2--;
    jg .loop                ;r2=0时候，不再跳转
    REP_RET
```

由于对汇编语言并不算很熟悉，因此对ff_pred16x16_vertical_8_mmx()中的每行函数都进行了注释并类比了C语言中等同的方法。从代码中可以看出，由于MMX指令集支持通过“MOVQ”指令一次性处理64bit（8字节，即8个像素点）数据，所以基于MMX指令集优化后的函数调用32次“MOVQ”即可完成16x16帧内预测Vertical模式的像素赋值工作（循环“loop”执行8次）。
下面再看一个针对SSE指令集优化过的函数ff_pred16x16_vertical_8_sse()。由于SSE指令集比MMX指令集更为先进，所以ff_pred16x16_vertical_8_sse()的效率比ff_pred16x16_vertical_8_mmx()还要高。

ff_pred16x16_vertical_8_sse()

ff_pred16x16_vertical_8_sse()的函数定义位于libavcodec\x86\h264_intrapred.asm，如下所示。

```
;sse指令优化
INIT_XMM sse
cglobal pred16x16_vertical_8, 2,3
    sub   r0, r1            ;r0=r0-r1。r0取值为src；r1取值为stride。此时r0指向16x16块上面一行像素数据
    mov   r2, 4             ;r2=4;r2为循环计数器，每次循环减1，r2为0的时候，循环停止
    movaps xmm0, [r0]       ;类似于memcpy(&xmm0,r0,16)。movaps传递128bit（16字节，对应16像素）数据
.loop:                      ;循环
    movaps [r0+r1*1], xmm0 ;类似于memcpy(r0+r1,&xmm0,16)。第1次循环，拷贝xmm0至宏块第1行16个像素。
    movaps [r0+r1*2], xmm0 ;类似于memcpy(r0+r1*2,&xmm0,16)。第1次循环，拷贝xmm0至宏块第2行16个像素。
    lea   r0, [r0+r1*2]    ;r0=r0+r1*2。r0前移2行。
    movaps [r0+r1*1], xmm0 ;类似于memcpy(r0+r1,&xmm0,16)。第1次循环，拷贝xmm0至宏块第1行16个像素。
    movaps [r0+r1*2], xmm0 ;类似于memcpy(r0+r1*2,&xmm0,16)。第1次循环，拷贝xmm0至宏块第2行16个像素。
    lea   r0, [r0+r1*2]    ;r0=r0+r1*2。r0再次前移2行。
                            ;注：一次循环处理了4行像素。16行像素一共处理4次
    dec   r2                ;r2--;
    jg .loop                ;r2=0时候，不再跳转
    REP_RET
```

从源代码可以看出，由于SSE指令集支持通过“MOVAPS”指令支持一次性处理128bit（16字节，即16个像素点）数据，所以基于MMX指令集优化后的函数调用16次“MOVAPS”即可完成16x16帧内预测Vertical模式的像素赋值工作（循环“loop”执行4次）。

至此有关FFmpeg H.264解码器帧内预测的部分的源代码就基本分析完毕了。下面分析处理帧内预测宏块的第二个步骤：残差数据的DCT反变化和叠加。

hl_decode_mb_idct_luma()

hl_decode_mb_idct_luma()对宏块的亮度残差进行进行DCT反变换，并且将残差数据叠加到前面阵内或者帧间预测得到的预测数据上（需要注意实际上“DCT反变换”和“叠加”两个步骤是同时完成的）。该函数的定义位于libavcodec\h264_mb.c，如下所示。

```cpp
//亮度的IDCT
static av_always_inline void hl_decode_mb_idct_luma(H264Context *h, int mb_type,
                                                    int is_h264, int simple,
                                                    int transform_bypass,
                                                    int pixel_shift,
                                                    int *block_offset,
                                                    int linesize,
                                                    uint8_t *dest_y, int p)
{
    //用于IDCT
    void (*idct_add)(uint8_t *dst, int16_t *block, int stride);
    int i;
    block_offset += 16 * p;
    //Intra4x4的DCT反变换在pred部分已经完成，这里就不需要处理了
    if (!IS_INTRA4x4(mb_type)) {
        if (is_h264) {
            //Intra16x16宏块
            if (IS_INTRA16x16(mb_type)) {
                //transform_bypass=0，不考虑
                if (transform_bypass) {
                    if (h->sps.profile_idc == 244 &&
                        (h->intra16x16_pred_mode == VERT_PRED8x8 ||
                         h->intra16x16_pred_mode == HOR_PRED8x8)) {
                        h->hpc.pred16x16_add[h->intra16x16_pred_mode](dest_y, block_offset,
                                                                      h->mb + (p * 256 << pixel_shift),
                                                                      linesize);
                    } else {
                        for (i = 0; i < 16; i++)
                            if (h->non_zero_count_cache[scan8[i + p * 16]] ||
                                dctcoef_get(h->mb, pixel_shift, i * 16 + p * 256))
                                h->h264dsp.h264_add_pixels4_clear(dest_y + block_offset[i],
                                                                  h->mb + (i * 16 + p * 256 << pixel_shift),
                                                                  linesize);
                    }
                } else {
                    //Intra16x16的DCT反变换
                    //最后的“16”代表内部循环处理16次
                    h->h264dsp.h264_idct_add16intra(dest_y, block_offset,
                                                    h->mb + (p * 256 << pixel_shift),
                                                    linesize,
                                                    h->non_zero_count_cache + p * 5 * 8);
                }
            } else if (h->cbp & 15) {//15=1111，即残差全部传送了
                //Inter类型的宏块


                //transform_bypass=0，不考虑
                if (transform_bypass) {
                    const int di = IS_8x8DCT(mb_type) ? 4 : 1;
                    idct_add = IS_8x8DCT(mb_type) ? h->h264dsp.h264_add_pixels8_clear
                                                  : h->h264dsp.h264_add_pixels4_clear;
                    for (i = 0; i < 16; i += di)
                        if (h->non_zero_count_cache[scan8[i + p * 16]])
                            idct_add(dest_y + block_offset[i],
                                     h->mb + (i * 16 + p * 256 << pixel_shift),
                                     linesize);
                } else {
                    //8x8的IDCT
                    if (IS_8x8DCT(mb_type))
                        h->h264dsp.h264_idct8_add4(dest_y, block_offset,
                                                   h->mb + (p * 256 << pixel_shift),
                                                   linesize,
                                                   h->non_zero_count_cache + p * 5 * 8);
                    //处理16x16宏块
                    //采用4x4的IDCT
                    //最后的“16”代表内部循环处理16次
                    //输出结果到dest_y
                    //h->mb中存储了DCT系数
                    else
                        h->h264dsp.h264_idct_add16(dest_y, block_offset,
                                                   h->mb + (p * 256 << pixel_shift),
                                                   linesize,
                                                   h->non_zero_count_cache + p * 5 * 8);
                }
            }
        } else if (CONFIG_SVQ3_DECODER) {
            for (i = 0; i < 16; i++)
                if (h->non_zero_count_cache[scan8[i + p * 16]] || h->mb[i * 16 + p * 256]) {
                    // FIXME benchmark weird rule, & below
                    uint8_t *const ptr = dest_y + block_offset[i];
                    ff_svq3_add_idct_c(ptr, h->mb + i * 16 + p * 256, linesize,
                                       h->qscale, IS_INTRA(mb_type) ? 1 : 0);
                }
        }
    }
}
```

下面根据源代码简单梳理一下hl_decode_mb_idct_luma()的流程：

（1）判断宏块是否属于Intra4x4类型，如果是，函数直接返回（Intra4x4比较特殊，它的DCT反变换已经前文所述的“帧内预测”部分完成）。

（2）根据不同的宏块类型作不同的处理：

a)Intra16x16：调用H264DSPContext的汇编函数h264_idct_add16intra()进行DCT反变换

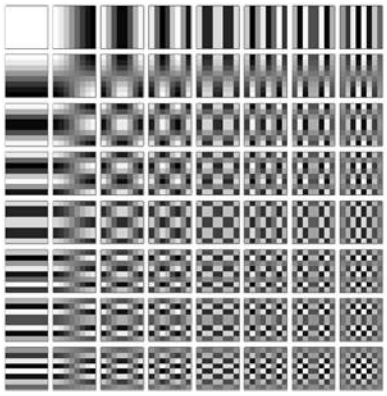
b)Inter类型：调用H264DSPContext的汇编函数h264_idct_add16()进行DCT反变换

PS：需要注意的是h264_idct_add16intra()和h264_idct_add16()只有微小的区别，它们的基本逻辑都是把16x16的块划分为16个4x4的块再进行DCT反变换。此外还有一点需要注意：函数名中的“add”的含义是将DCT反变换之后的残差像素数据直接叠加到已有数据之上。

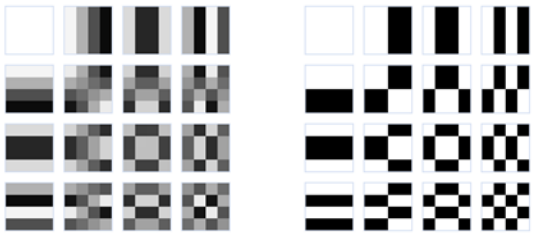
下文记录DCT反变化函数相关的源代码。

DCT反变化小知识

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



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

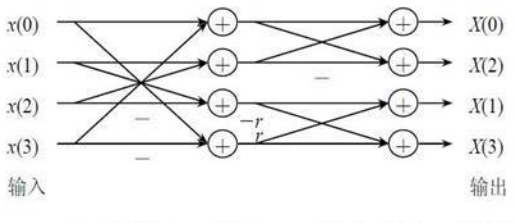


4x4整数DCT变换的公式如下所示。

$$\boldsymbol{Y} = (\boldsymbol{C}_f \boldsymbol{X} \boldsymbol{C}_f^{\mathrm{T}})$$

$$= \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \boldsymbol{X} \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{bmatrix} \right)$$

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

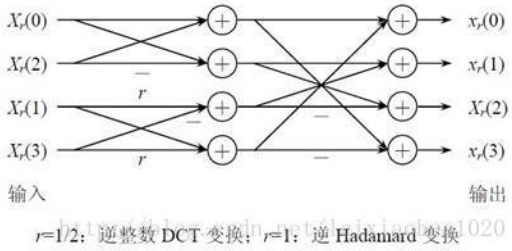


r=2：整数DCT变换；r=1：Hadamard变换

同理，DCT反变换就是DCT变换的逆变换。DCT反变换的公式如下所示。

$$X_r = C_i^{\mathrm{T}}(W)C_i$$
$$= \begin{bmatrix} 1 & 1 & 1 & \frac{1}{2} \\ 1 & \frac{1}{2} & -1 & -1 \\ 1 & -\frac{1}{2} & -1 & 1 \\ 1 & -1 & 1 & -\frac{1}{2} \end{bmatrix} W \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \frac{1}{2} & -\frac{1}{2} & -1 \\ 1 & -1 & -1 & 1 \\ \frac{1}{2} & -1 & 1 & -\frac{1}{2} \end{bmatrix}$$

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



r=1/2：逆整数 DCT 变换；r=1：逆 Hadamard 变换

下文记录的源代码正是实现了上述4x4整数DCT反变换。

DCT反变换汇编函数的初始化

FFmpeg H.264解码器中4x4DCT反变换（也称为“IDCT”）汇编函数指针位于H264DSPContext中。在FFmpeg H.264解码器初始化的时候，会调用ff_h264dsp_init()根据系统的配置对H264DSPContext中的这些IDCT函数指针进行赋值（H264DSPContext中实际上不仅仅包含DCT反变换函数，还包含了Hadamard反变换函数，环路滤波函数，在这里不详细讨论）。下面简单看一下ff_h264_pred_init()的定义。

ff_h264dsp_init()

ff_h264_pred_init()用于初始化DCT反变换函数，Hadamard反变换函数，环路滤波函数。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ff_h264dsp_init()的定义可以看出，该函数通过调用“H264_DSP(depth)”宏完成C语言版本的DCT反变换函数，Hadamard反变换函数，环路滤波函数的初始化。在函数的末尾还会判断系统的特性，如果允许的话会初始化效率更高的经过汇编优化的函数。
下面我们展开“H264_DSP(8)”宏看看C语言版本函数的初始化过程。

```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：

（1）4x4块的DCT反变换函数指针h264_idct_add()指向ff_h264_idct_add_8_c()

（2）只包含DC系数的4x4块的DCT反变换函数指针h264_idct_dc_add()指向ff_h264_idct_dc_add_8_c()

（3）16x16块的DCT反变换函数指针h264_idct_add16()指向ff_h264_idct_add16_8_c()

（4）16x16的Intra块的DCT反变换函数指针h264_idct_add16intra()指向ff_h264_idct_add16intra_8_c()

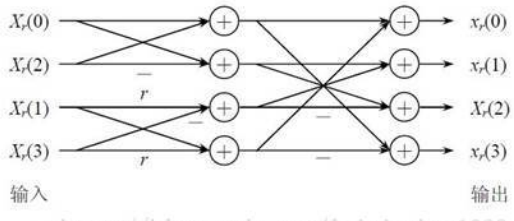
下文将会简单分析上述几个函数。

ff_h264_idct_add_8_c()

ff_h264_idct_add_8_c()用于进行4x4整数DCT反变换，该函数的定义位于libavcodec\h264idct_template.c，如下所示。

```cpp
/* IDCT
 *
 * 注释：雷霄骅
 * leixiaohua1020@126.com
 * http://blog.csdn.net/leixiaohua1020
 *
 * 参数：
 * _block：输入DCT系数
 * _dst：输出像素数据
 * stride：一行图像数据的大小
 *
 */
//IDCT反变换（4x4）
//“add”的意思是在像素数据（通过预测获得）上面叠加（而不是赋值）IDCT的结果
void FUNCC(ff_h264_idct_add)(uint8_t *_dst, int16_t *_block, int stride)
{

    /*
     *     | 1  1  1  1 |   | 1  2  1  1 |
     *     | 2  1 -1 -2 |   | 1  1 -1 -2 |
     * Y = | 1 -1 -1  2 | X | 1 -1 -1  2 |
     *     | 1 -2  2 -1 |   | 1 -2  1 -1 |
     *
     */
    int i;
    pixel *dst = (pixel*)_dst;
    dctcoef *block = (dctcoef*)_block;
    stride >>= sizeof(pixel)-1;

    block[0] += 1 << 5;
    //蝶形算法（一维变换，纵向）
    //---+----------
    // 0 |   |   |   |
    // 4 |
    // 8 |
    // 12|
    //---+----------
    for(i=0; i<4; i++){
        //[0]和[2]
        const int z0=  block[i + 4*0]     +  block[i + 4*2];
        const int z1=  block[i + 4*0]     -  block[i + 4*2];
        //[1]和[3]
        const int z2= (block[i + 4*1]>>1) -  block[i + 4*3];
        const int z3=  block[i + 4*1]     + (block[i + 4*3]>>1);

        block[i + 4*0]= z0 + z3;
        block[i + 4*1]= z1 + z2;
        block[i + 4*2]= z1 - z2;
        block[i + 4*3]= z0 - z3;
    }
    //蝶形算法（另一维变换，横向）
    //---+----------
    // 0 | 1 | 2 | 3 |
    //   |
    //   |
    //   |
    //---+----------
    for(i=0; i<4; i++){
        const int z0=  block[0 + 4*i]     +  block[2 + 4*i];
        const int z1=  block[0 + 4*i]     -  block[2 + 4*i];
        const int z2= (block[1 + 4*i]>>1) -  block[3 + 4*i];
        const int z3=  block[1 + 4*i]     + (block[3 + 4*i]>>1);
        //av_clip_pixel(): 把一个整形转换取值范围为0-255的数值，用于限幅
        //注意是累加而不是赋值到dst上（所以函数名中包含“add”）

        //转置？！
        //一列一列处理
        dst[i + 0*stride]= av_clip_pixel(dst[i + 0*stride] + ((z0 + z3) >> 6));
        dst[i + 1*stride]= av_clip_pixel(dst[i + 1*stride] + ((z1 + z2) >> 6));
        dst[i + 2*stride]= av_clip_pixel(dst[i + 2*stride] + ((z1 - z2) >> 6));
        dst[i + 3*stride]= av_clip_pixel(dst[i + 3*stride] + ((z0 - z3) >> 6));
    }
    //清零
    memset(block, 0, 16 * sizeof(dctcoef));
}
```

从ff_h264_idct_add_8_c()的定义可以看出，该函数首先对4x4系数块中纵向的4列数据进行了一维DCT反变换，然后又对4x4系数块中横向的4行数据进行了DCT一维反变换，最后将变换后的残差图像数据叠加到了原有数据之上。
在这里一维DCT反变换采用了蝶形快速算法，如下图所示。



r=1/2：逆整数 DCT 变换；r=1：逆 Hadamard 变换

下面分析上文提到的几个函数。

h264_idct_dc_add()

ff_h264_idct_dc_add_8_c()用于对只有DC系数的4x4矩阵进行4x4整数DCT反变换，该函数的定义位于libavcodec\h264idct_template.c，如下所示。

```cpp
// assumes all AC coefs are 0
//DCT反变换，特殊情况：
//AC系数全部为0（没有传递AC系数，只有DC系数）
void FUNCC(ff_h264_idct_dc_add)(uint8_t *_dst, int16_t *_block, int stride){
    int i, j;
    pixel *dst = (pixel*)_dst;
    dctcoef *block = (dctcoef*)_block;
    //DC系数
    int dc = (block[0] + 32) >> 6;
    stride /= sizeof(pixel);
    //设置为0
    block[0] = 0;
    //在4x4块的每个像素上面累加（注意不是赋值）dc系数
    for( j = 0; j < 4; j++ )
    {
        for( i = 0; i < 4; i++ )
            dst[i] = av_clip_pixel( dst[i] + dc );//av_clip_pixel(): 把一个整形转换取值范围为0-255的数值，用于限幅
        dst += stride;//下一行
    }
}
```

可以看出只有DC系数的DCT反变换相比前面“正式”的DCT反变换来说简单了很多，只需要把DC系数赋值到4x4块中的每个像素上就可以了。

ff_h264_idct_add16_8_c()

ff_h264_idct_add16_8_c()用于对16x16的块进行4x4整数DCT反变换，该函数的定义位于libavcodec\h264idct_template.c，如下所示。

```cpp
//处理16x16宏块
//采用4x4的IDCT
//最后的“16”代表内部循环处理16次
//输入为block，输出为dst
void FUNCC(ff_h264_idct_add16)(uint8_t *dst, const int *block_offset, int16_t *block, int stride, const uint8_t nnzc[15*8]){
    int i;
    //循环16次
    for(i=0; i<16; i++){
        //非零系数个数
        int nnz = nnzc[ scan8[i] ];
        //非零系数个数不为0才处理
        if(nnz){
            //特殊：只有DC系数
            if(nnz==1 && ((dctcoef*)block)[i*16]) FUNCC(ff_h264_idct_dc_add)(dst + block_offset[i], block + i*16*sizeof(pixel), stride);
            //一般的情况
            else                                  FUNCC(ff_h264_idct_add   )(dst + block_offset[i], block + i*16*sizeof(pixel), stride);
        }
    }
}
```

从源代码可以看出，16x16块的4x4DCT反变换的实质就是把16x16的块分割为16个4x4的块，然后分别进行4x4DCT反变换。

h264_idct_add16intra()

h264_idct_add16intra()用于对16x16的帧内预测（Intra）的块进行4x4整数DCT反变换，该函数的定义位于libavcodec\h264idct_template.c，如下所示。

```cpp
//处理Intra16x16宏块
//采用4x4的IDCT
//最后的“16”代表内部循环处理16次
//输入为block，输出为dst
void FUNCC(ff_h264_idct_add16intra)(uint8_t *dst, const int *block_offset, int16_t *block, int stride, const uint8_t nnzc[15*8]){
    int i;
    for(i=0; i<16; i++){
        if(nnzc[ scan8[i] ])             FUNCC(ff_h264_idct_add   )(dst + block_offset[i], block + i*16*sizeof(pixel), stride);
        else if(((dctcoef*)block)[i*16]) FUNCC(ff_h264_idct_dc_add)(dst + block_offset[i], block + i*16*sizeof(pixel), stride);
    }
}
```

可以看出h264_idct_add16intra()的机制与ff_h264_idct_add16_8_c()是类似的，只是有一些细节的差别：它们都是把16x16的块分割为16个4x4的块，然后分别进行4x4 DCT反变换。

至此FFmpeg H.264解码器的帧内宏块（Intra）解码相关的代码就基本分析完毕了。总而言之帧内预测宏块的解码就是“预测+残差”的处理流程。下一篇文章分析帧间宏块（Inter）解码的代码。

雷霄骅
leixiaohua1020@126.com
http://blog.csdn.net/leixiaohua1020

文章标签： FFmpeg 宏块 解码 帧内预测 DCT

个人分类： FFMPEG

所属专栏： FFmpeg