

## 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](#)

```
[cpp] 1. //解码slice
2. //三个主要步骤：
3. //1. 熵解码（CAVLC/CABAC）
```

```

4. //2.宏块解码
5. //3.环路滤波
6. //此外还包含了错误隐藏代码
7. static int decode_slice(struct AVCodecContext *avctx, void *arg)
8. {
9.     H264Context *h = *(void **)arg;
10.    int lf_x_start = h->mb_x;
11.
12.    h->mb_skip_run = -1;
13.
14.    av_assert0(h->block_offset[15] == (4 * ((scan8[15] - scan8[0]) & 7) << h->pixel_shift) + 4 * h->linesize * ((scan8[15] - scan8[0]) >> 3));
15.
16.    h->is_complex = FRAME_MBAFF(h) || h->picture_structure != PICT_FRAME ||
17.        avctx->codec_id != AV_CODEC_ID_H264 ||
18.        (CONFIG_GRAY && (h->flags & CODEC_FLAG_GRAY));
19.
20.    if (!(h->avctx->active_thread_type & FF_THREAD_SLICE) && h->picture_structure == PICT_FRAME && h->er.error_status_table) {
21.        const int start_i = av_clip(h->resync_mb_x + h->resync_mb_y * h->mb_width, 0, h->mb_num - 1);
22.        if (start_i) {
23.            int prev_status = h->er.error_status_table[h->er.mb_index2xy[start_i - 1]];
24.            prev_status &= ~ VP_START;
25.            if (prev_status != (ER_MV_END | ER_DC_END | ER_AC_END))
26.                h->er.error_occurred = 1;
27.        }
28.    }
29.    //CABAC情况
30.    if (h->pps.cabac) {
31.        /* realign */
32.        align_get_bits(&h->gb);
33.
34.        /* init cabac */
35.        //初始化CABAC解码器
36.        ff_init_cabac_decoder(&h->cabac,
37.            h->gb.buffer + get_bits_count(&h->gb) / 8,
38.            (get_bits_left(&h->gb) + 7) / 8);
39.
40.        ff_h264_init_cabac_states(h);
41.        //循环处理每个宏块
42.        for (;;) {
43.            // START_TIMER
44.            //解码CABAC数据
45.            int ret = ff_h264_decode_mb_cabac(h);
46.            int eos;
47.            // STOP_TIMER("decode_mb_cabac")
48.            //解码宏块
49.            if (ret >= 0)
50.                ff_h264_hl_decode_mb(h);
51.
52.            // FIXME optimal? or let mb_decode decode 16x32 ?
53.            //宏块级帧场自适应。很少接触
54.            if (ret >= 0 && FRAME_MBAFF(h)) {
55.                h->mb_y++;
56.
57.                ret = ff_h264_decode_mb_cabac(h);
58.                //解码宏块
59.                if (ret >= 0)
60.                    ff_h264_hl_decode_mb(h);
61.                h->mb_y--;
62.            }
63.            eos = get_cabac_terminate(&h->cabac);
64.
65.            if ((h->workaround_bugs & FF_BUG_TRUNCATED) &&
66.                h->cabac.bytestream > h->cabac.bytestream_end + 2) {
67.                //错误隐藏
68.                er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x - 1,
69.                    h->mb_y, ER_MB_END);
70.                if (h->mb_x >= lf_x_start)
71.                    loop_filter(h, lf_x_start, h->mb_x + 1);
72.                return 0;
73.            }
74.            if (h->cabac.bytestream > h->cabac.bytestream_end + 2)
75.                av_log(h->avctx, AV_LOG_DEBUG, "bytestream overread %"PTRDIFF_SPECIFIER"\n", h->cabac.bytestream_end - h->cabac.bytestream);
76.            if (ret < 0 || h->cabac.bytestream > h->cabac.bytestream_end + 4) {
77.                av_log(h->avctx, AV_LOG_ERROR,
78.                    "error while decoding MB %d %d, bytestream %"PTRDIFF_SPECIFIER"\n",
79.                    h->mb_x, h->mb_y,
80.                    h->cabac.bytestream_end - h->cabac.bytestream);
81.                er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x,
82.                    h->mb_y, ER_MB_ERROR);
83.                return AVERROR_INVALIDDATA;
84.            }
85.            //mb_x自增
86.            //如果自增后超过了一行的mb个数
87.            if (++h->mb_x >= h->mb_width) {
88.                //环路滤波
89.                loop_filter(h, lf_x_start, h->mb_x);
90.                h->mb_x = lf_x_start = 0;
91.                decode_finish_row(h);
92.                //mb_y自增 (处理下一行)
93.                ++h->mb_y;

```

```

94.         //宏块级帧场自适应, 暂不考虑
95.         if (FIELD_OR_MBAFF_PICTURE(h)) {
96.             ++h->mb_y;
97.             if (FRAME_MBAFF(h) && h->mb_y < h->mb_height)
98.                 predict_field_decoding_flag(h);
99.         }
100.     }
101.     //如果mb_y超过了mb的行数
102.     if (eos || h->mb_y >= h->mb_height) {
103.         tprintf(h->avctx, "slice end %d %d\n",
104.             get_bits_count(&h->gb), h->gb.size_in_bits);
105.         er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x - 1,
106.             h->mb_y, ER_MB_END);
107.         if (h->mb_x > lf_x_start)
108.             loop_filter(h, lf_x_start, h->mb_x);
109.         return 0;
110.     }
111. }
112. } else {
113.     //CAVLC情况
114.     //循环处理每个宏块
115.     for (;;) {
116.         //解码宏块的CAVLC
117.         int ret = ff_h264_decode_mb_cavlc(h);
118.         //解码宏块
119.         if (ret >= 0)
120.             ff_h264_hl_decode_mb(h);
121.
122.         // FIXME optimal? or let mb_decode decode 16x32 ?
123.         if (ret >= 0 && FRAME_MBAFF(h)) {
124.             h->mb_y++;
125.             ret = ff_h264_decode_mb_cavlc(h);
126.
127.             if (ret >= 0)
128.                 ff_h264_hl_decode_mb(h);
129.             h->mb_y--;
130.         }
131.
132.         if (ret < 0) {
133.             av_log(h->avctx, AV_LOG_ERROR,
134.                 "error while decoding MB %d %d\n", h->mb_x, h->mb_y);
135.             er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x,
136.                 h->mb_y, ER_MB_ERROR);
137.             return ret;
138.         }
139.
140.         if (++h->mb_x >= h->mb_width) {
141.             //环路滤波
142.             loop_filter(h, lf_x_start, h->mb_x);
143.             h->mb_x = lf_x_start = 0;
144.             decode_finish_row(h);
145.             ++h->mb_y;
146.             if (FIELD_OR_MBAFF_PICTURE(h)) {
147.                 ++h->mb_y;
148.                 if (FRAME_MBAFF(h) && h->mb_y < h->mb_height)
149.                     predict_field_decoding_flag(h);
150.             }
151.             if (h->mb_y >= h->mb_height) {
152.                 tprintf(h->avctx, "slice end %d %d\n",
153.                     get_bits_count(&h->gb), h->gb.size_in_bits);
154.
155.                 if (get_bits_left(&h->gb) == 0
156.                     || get_bits_left(&h->gb) > 0 && !(h->avctx->err_recognition & AV_EF_AGGRESSIVE)) {
157.                     //错误隐藏
158.                     er_add_slice(h, h->resync_mb_x, h->resync_mb_y,
159.                         h->mb_x - 1, h->mb_y, ER_MB_END);
160.
161.                     return 0;
162.                 } else {
163.                     er_add_slice(h, h->resync_mb_x, h->resync_mb_y,
164.                         h->mb_x, h->mb_y, ER_MB_END);
165.
166.                     return AVERROR_INVALIDDATA;
167.                 }
168.             }
169.         }
170.
171.         if (get_bits_left(&h->gb) <= 0 && h->mb_skip_run <= 0) {
172.             tprintf(h->avctx, "slice end %d %d\n",
173.                 get_bits_count(&h->gb), h->gb.size_in_bits);
174.
175.             if (get_bits_left(&h->gb) == 0) {
176.                 er_add_slice(h, h->resync_mb_x, h->resync_mb_y,
177.                     h->mb_x - 1, h->mb_y, ER_MB_END);
178.                 if (h->mb_x > lf_x_start)
179.                     loop_filter(h, lf_x_start, h->mb_x);
180.
181.                 return 0;
182.             } else {
183.                 er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x,
184.                     h->mb_y, ER_MB_ERROR);

```

```

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

```

重复记录一下decode\_slice()的流程：

- (1) 判断H.264码流是CABAC编码还是CAVLC编码，进入不同的处理循环。
- (2) 如果是CABAC编码，首先调用ff\_init\_cabac\_decoder()初始化CABAC解码器。然后进入一个循环，依次对每个宏块进行以下处理：
  - a)调用ff\_h264\_decode\_mb\_cabac()进行CABAC熵解码
  - b)调用ff\_h264\_hl\_decode\_mb()进行宏块解码
  - c)解码一行宏块之后调用loop\_filter()进行环路滤波
  - d)此外还有可能调用er\_add\_slice()进行错误隐藏处理
- (3) 如果是CAVLC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。

```

1. //解码宏块
2. void ff_h264_hl_decode_mb(H264Context *h)
3. {
4.     //宏块序号 mb_xy = mb_x + mb_y*mb_stride
5.     const int mb_xy = h->mb_xy;
6.     //宏块类型
7.     const int mb_type = h->cur_pic.mb_type[mb_xy];
8.     //比较少见，PCM类型
9.     int is_complex = CONFIG_SMALL || h->is_complex ||
10.                    IS_INTRA_PCM(mb_type) || h->qscale == 0;
11.     //YUV444
12.     if (CHROMA444(h)) {
13.         if (is_complex || h->pixel_shift)
14.             hl_decode_mb_444_complex(h);
15.         else
16.             hl_decode_mb_444_simple_8(h);
17.     } else if (is_complex) {
18.         hl_decode_mb_complex(h); //PCM类型？
19.     } else if (h->pixel_shift) {
20.         hl_decode_mb_simple_16(h); //色彩深度为16
21.     } else
22.         hl_decode_mb_simple_8(h); //色彩深度为8
23. }

```

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。

```

1. //hl是什么意思？high level？
2. /*
3.  * 注释：雷霄骅
4.  * leixiaohua1020@126.com
5.  * http://blog.csdn.net/leixiaohua1020
6.  *
7.  * 宏块解码

```

```

8.  * 帧内宏块：帧内预测->残差DCT反变换
9.  * 帧间宏块：帧间预测（运动补偿）->残差DCT反变换
10.
11.  */
12. static av_noinline void FUNC(hl_decode_mb)(H264Context *h)
13. {
14.     //序号：x（行）和y（列）
15.     const int mb_x = h->mb_x;
16.     const int mb_y = h->mb_y;
17.     //宏块序号 mb_xy = mb_x + mb_y*mb_stride
18.     const int mb_xy = h->mb_xy;
19.     //宏块类型
20.     const int mb_type = h->cur_pic.mb_type[mb_xy];
21.     //这三个变量存储最后处理完成的像素值
22.     uint8_t *dest_y, *dest_cb, *dest_cr;
23.     int linesize, uvlinesize /*dct_offset*/;
24.     int i, j;
25.     int *block_offset = &h->block_offset[0];
26.     const int transform_bypass = !SIMPLE && (h->qscale == 0 && h->sps.transform_bypass);
27.     /* is_h264 should always be true if SVQ3 is disabled. */
28.     const int is_h264 = !CONFIG_SVQ3_DECODER || SIMPLE || h->avctx->codec_id == AV_CODEC_ID_H264;
29.     void (*idct_add)(uint8_t *dst, int16_t *block, int stride);
30.     const int block_h = 16 >> h->chroma_y_shift;
31.     const int chroma422 = CHROMA422(h);
32.     //存储Y, U, V像素的位置：dest_y, dest_cb, dest_cr
33.     //分别对应AVFrame的data[0], data[1], data[2]
34.     dest_y = h->cur_pic.f.data[0] + ((mb_x << PIXEL_SHIFT) + mb_y * h->linesize) * 16;
35.     dest_cb = h->cur_pic.f.data[1] + (mb_x << PIXEL_SHIFT) * 8 + mb_y * h->uvlinesize * block_h;
36.     dest_cr = h->cur_pic.f.data[2] + (mb_x << PIXEL_SHIFT) * 8 + mb_y * h->uvlinesize * block_h;
37.
38.     h->vdsp.prefetch(dest_y + (h->mb_x & 3) * 4 * h->linesize + (64 << PIXEL_SHIFT), h->linesize, 4);
39.     h->vdsp.prefetch(dest_cb + (h->mb_x & 7) * h->uvlinesize + (64 << PIXEL_SHIFT), dest_cr - dest_cb, 2);
40.
41.     h->list_counts[mb_xy] = h->list_count;
42.
43.     //系统中包含了
44.     //define SIMPLE 1
45.     //不会执行？
46.     if (!SIMPLE && MB_FIELD(h)) {
47.         linesize = h->mb_linesize = h->linesize * 2;
48.         uvlinesize = h->mb_uvlinesize = h->uvlinesize * 2;
49.         block_offset = &h->block_offset[48];
50.         if (mb_y & 1) { // FIXME move out of this function?
51.             dest_y -= h->linesize * 15;
52.             dest_cb -= h->uvlinesize * (block_h - 1);
53.             dest_cr -= h->uvlinesize * (block_h - 1);
54.         }
55.         if (FRAME_MBAFF(h)) {
56.             int list;
57.             for (list = 0; list < h->list_count; list++) {
58.                 if (!USES_LIST(mb_type, list))
59.                     continue;
60.                 if (IS_16X16(mb_type)) {
61.                     int8_t *ref = &h->ref_cache[list][scan8[0]];
62.                     fill_rectangle(ref, 4, 4, 8, (16 + *ref) ^ (h->mb_y & 1), 1);
63.                 } else {
64.                     for (i = 0; i < 16; i += 4) {
65.                         int ref = h->ref_cache[list][scan8[i]];
66.                         if (ref >= 0)
67.                             fill_rectangle(&h->ref_cache[list][scan8[i]], 2, 2,
68.                                           8, (16 + ref) ^ (h->mb_y & 1), 1);
69.                     }
70.                 }
71.             }
72.         }
73.     } else {
74.         linesize = h->mb_linesize = h->linesize;
75.         uvlinesize = h->mb_uvlinesize = h->uvlinesize;
76.         // dct_offset = s->linesize * 16;
77.     }
78.     //系统中包含了
79.     //define SIMPLE 1
80.     //不会执行？
81.     if (!SIMPLE && IS_INTRA_PCM(mb_type)) {
82.         const int bit_depth = h->sps.bit_depth_luma;
83.         if (PIXEL_SHIFT) {
84.             int j;
85.             GetBitContext gb;
86.             init_get_bits(&gb, h->intra_pcm_ptr,
87.                          ff_h264_mb_sizes[h->sps.chroma_format_idc] * bit_depth);
88.
89.             for (i = 0; i < 16; i++) {
90.                 uint16_t *tmp_y = (uint16_t *) (dest_y + i * linesize);
91.                 for (j = 0; j < 16; j++)
92.                     tmp_y[j] = get_bits(&gb, bit_depth);
93.             }
94.             if (SIMPLE || !CONFIG_GRAY || !(h->flags & CODEC_FLAG_GRAY)) {
95.                 if (!h->sps.chroma_format_idc) {
96.                     for (i = 0; i < block_h; i++) {
97.                         uint16_t *tmp_cb = (uint16_t *) (dest_cb + i * uvlinesize);
98.                         uint16_t *tmp_cr = (uint16_t *) (dest_cr + i * uvlinesize);

```

```

99.         for (j = 0; j < 8; j++) {
100.             tmp_cb[j] = tmp_cr[j] = 1 << (bit_depth - 1);
101.         }
102.     }
103. } else {
104.     for (i = 0; i < block_h; i++) {
105.         uint16_t *tmp_cb = (uint16_t *) (dest_cb + i * uvlinesize);
106.         for (j = 0; j < 8; j++)
107.             tmp_cb[j] = get_bits(&gb, bit_depth);
108.     }
109.     for (i = 0; i < block_h; i++) {
110.         uint16_t *tmp_cr = (uint16_t *) (dest_cr + i * uvlinesize);
111.         for (j = 0; j < 8; j++)
112.             tmp_cr[j] = get_bits(&gb, bit_depth);
113.     }
114. }
115. }
116. } else {
117.     for (i = 0; i < 16; i++)
118.         memcpy(dest_y + i * linesize, h->intra_pcm_ptr + i * 16, 16);
119.     if (SIMPLE || !CONFIG_GRAY || !(h->flags & CODEC_FLAG_GRAY)) {
120.         if (!h->sps.chroma_format_idc) {
121.             for (i = 0; i < 8; i++) {
122.                 memset(dest_cb + i * uvlinesize, 1 << (bit_depth - 1), 8);
123.                 memset(dest_cr + i * uvlinesize, 1 << (bit_depth - 1), 8);
124.             }
125.         } else {
126.             const uint8_t *src_cb = h->intra_pcm_ptr + 256;
127.             const uint8_t *src_cr = h->intra_pcm_ptr + 256 + block_h * 8;
128.             for (i = 0; i < block_h; i++) {
129.                 memcpy(dest_cb + i * uvlinesize, src_cb + i * 8, 8);
130.                 memcpy(dest_cr + i * uvlinesize, src_cr + i * 8, 8);
131.             }
132.         }
133.     }
134. }
135. } else {
136.     //Intra类型
137.     //Intra4x4或者Intra16x16
138.
139.     if (IS_INTRA(mb_type)) {
140.         if (h->deblocking_filter)
141.             xchg_mb_border(h, dest_y, dest_cb, dest_cr, linesize,
142.                             uvlinesize, 1, 0, SIMPLE, PIXEL_SHIFT);
143.
144.         if (SIMPLE || !CONFIG_GRAY || !(h->flags & CODEC_FLAG_GRAY)) {
145.             h->hpc.pred8x8[h->chroma_pred_mode](dest_cb, uvlinesize);
146.             h->hpc.pred8x8[h->chroma_pred_mode](dest_cr, uvlinesize);
147.         }
148.         //帧内预测-亮度
149.         hl_decode_mb_predict_luma(h, mb_type, is_h264, SIMPLE,
150.                                   transform_bypass, PIXEL_SHIFT,
151.                                   block_offset, linesize, dest_y, 0);
152.
153.         if (h->deblocking_filter)
154.             xchg_mb_border(h, dest_y, dest_cb, dest_cr, linesize,
155.                             uvlinesize, 0, 0, SIMPLE, PIXEL_SHIFT);
156.     } else if (is_h264) {
157.         //Inter类型
158.
159.         //运动补偿
160.         if (chroma422) {
161.             FUNC(hl_motion_422)(h, dest_y, dest_cb, dest_cr,
162.                                 h->qpel_put, h->h264chroma.put_h264_chroma_pixels_tab,
163.                                 h->qpel_avg, h->h264chroma.avg_h264_chroma_pixels_tab,
164.                                 h->h264dsp.weight_h264_pixels_tab,
165.                                 h->h264dsp.biweight_h264_pixels_tab);
166.         } else {
167.             //“*_put”处理单向预测, “*_avg”处理双向预测, “weight”处理加权预测
168.             //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
169.             //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
170.             //h->qpel_avg[16]情况也类似
171.             FUNC(hl_motion_420)(h, dest_y, dest_cb, dest_cr,
172.                                 h->qpel_put, h->h264chroma.put_h264_chroma_pixels_tab,
173.                                 h->qpel_avg, h->h264chroma.avg_h264_chroma_pixels_tab,
174.                                 h->h264dsp.weight_h264_pixels_tab,
175.                                 h->h264dsp.biweight_h264_pixels_tab);
176.         }
177.     }
178.     //亮度的IDCT
179.     hl_decode_mb_idct_luma(h, mb_type, is_h264, SIMPLE, transform_bypass,
180.                             PIXEL_SHIFT, block_offset, linesize, dest_y, 0);
181.     //色度的IDCT（没有写在一个单独的函数中）
182.     if ((SIMPLE || !CONFIG_GRAY || !(h->flags & CODEC_FLAG_GRAY)) &&
183.         (h->cbp & 0x30)) {
184.         uint8_t *dest[2] = { dest_cb, dest_cr };
185.         //transform_bypass=0, 不考虑
186.         if (transform_bypass) {
187.             if (IS_INTRA(mb_type) && h->sps.profile_idc == 244 &&
188.                 (h->chroma_pred_mode == VERT_PRED8x8 ||
189.                  h->chroma_pred_mode == HOR_PRED8x8)) {

```

```

190.         h->hpc.pred8x8_add[h->chroma_pred_mode](dest[0],
191.                                                 block_offset + 16,
192.                                                 h->mb + (16 * 16 * 1 << PIXEL_SHIFT),
193.                                                 uvlinesize);
194.         h->hpc.pred8x8_add[h->chroma_pred_mode](dest[1],
195.                                                 block_offset + 32,
196.                                                 h->mb + (16 * 16 * 2 << PIXEL_SHIFT),
197.                                                 uvlinesize);
198.     } else {
199.         idct_add = h->h264dsp.h264_add_pixels4_clear;
200.         for (j = 1; j < 3; j++) {
201.             for (i = j * 16; i < j * 16 + 4; i++)
202.                 if (h->non_zero_count_cache[scan8[i]] ||
203.                     dctcoef_get(h->mb, PIXEL_SHIFT, i * 16))
204.                     idct_add(dest[j - 1] + block_offset[i],
205.                               h->mb + (i * 16 << PIXEL_SHIFT),
206.                               uvlinesize);
207.             if (chroma422) {
208.                 for (i = j * 16 + 4; i < j * 16 + 8; i++)
209.                     if (h->non_zero_count_cache[scan8[i + 4]] ||
210.                         dctcoef_get(h->mb, PIXEL_SHIFT, i * 16))
211.                         idct_add(dest[j - 1] + block_offset[i + 4],
212.                                   h->mb + (i * 16 << PIXEL_SHIFT),
213.                                   uvlinesize);
214.             }
215.         }
216.     }
217. } else {
218.     if (is_h264) {
219.         int qp[2];
220.         if (chroma422) {
221.             qp[0] = h->chroma_qp[0] + 3;
222.             qp[1] = h->chroma_qp[1] + 3;
223.         } else {
224.             qp[0] = h->chroma_qp[0];
225.             qp[1] = h->chroma_qp[1];
226.         }
227.         //色度的IDCT
228.
229.         //直流量分的hadamard变换
230.         if (h->non_zero_count_cache[scan8[CHROMA_DC_BLOCK_INDEX + 0]])
231.             h->h264dsp.h264_chroma_dc_dequant_idct(h->mb + (16 * 16 * 1 << PIXEL_SHIFT),
232.                                                     h->dequant4_coeff[IS_INTRA(mb_type) ? 1 : 4][qp[0]][0]);
233.         if (h->non_zero_count_cache[scan8[CHROMA_DC_BLOCK_INDEX + 1]])
234.             h->h264dsp.h264_chroma_dc_dequant_idct(h->mb + (16 * 16 * 2 << PIXEL_SHIFT),
235.                                                     h->dequant4_coeff[IS_INTRA(mb_type) ? 2 : 5][qp[1]][0]);
236.         //IDCT
237.         //最后的“8”代表内部循环处理8次 (U,V各4次)
238.         h->h264dsp.h264_idct_add8(dest, block_offset,
239.                                   h->mb, uvlinesize,
240.                                   h->non_zero_count_cache);
241.     } else if (CONFIG_SVQ3_DECODER) {
242.         h->h264dsp.h264_chroma_dc_dequant_idct(h->mb + 16 * 16 * 1,
243.                                                 h->dequant4_coeff[IS_INTRA(mb_type) ? 1 : 4][h->chroma_qp[0]][0]);
244.         h->h264dsp.h264_chroma_dc_dequant_idct(h->mb + 16 * 16 * 2,
245.                                                 h->dequant4_coeff[IS_INTRA(mb_type) ? 2 : 5][h->chroma_qp[1]][0]);
246.         for (j = 1; j < 3; j++) {
247.             for (i = j * 16; i < j * 16 + 4; i++)
248.                 if (h->non_zero_count_cache[scan8[i]] || h->mb[i * 16]) {
249.                     uint8_t *const ptr = dest[j - 1] + block_offset[i];
250.                     ff_svq3_add_idct_c(ptr, h->mb + i * 16,
251.                                         uvlinesize,
252.                                         ff_h264_chroma_qp[0][h->qscale + 12] - 12, 2);
253.                 }
254.             }
255.         }
256.     }
257. }
258. }
259. }

```

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

- 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。
- 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

- 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]
1. //帧内预测-亮度
2. //分成2种情况：Intra4x4和Intra16x16
3. static av_always_inline void hl_decode_mb_predict_luma(H264Context *h,
4.                                                         int mb_type, int is_h264,
5.                                                         int simple,
6.                                                         int transform_bypass,
7.                                                         int pixel_shift,
8.                                                         int *block_offset,
9.                                                         int linesize,
10.                                                         uint8_t *dest_y, int p)
11. {
12.     //用于DCT反变换
13.     void (*idct_add)(uint8_t *dst, int16_t *block, int stride);
14.     void (*idct_dc_add)(uint8_t *dst, int16_t *block, int stride);
15.     int i;
16.     int qscale = p == 0 ? h->qscale : h->chroma_qp[p - 1];
17.     //外部调用时候p=0
18.     block_offset += 16 * p;
19.     if (IS_INTRA4x4(mb_type)) {
20.         //Intra4x4帧内预测
21.
22.         if (IS_8x8DCT(mb_type)) {
23.             //如果使用了8x8的DCT，先不研究
24.             if (transform_bypass) {
25.                 idct_dc_add =
26.                     idct_add = h->h264dsp.h264_add_pixels8_clear;
27.             } else {
28.                 idct_dc_add = h->h264dsp.h264_idct8_dc_add;
29.                 idct_add = h->h264dsp.h264_idct8_add;
30.             }
31.             for (i = 0; i < 16; i += 4) {
32.                 uint8_t *const ptr = dest_y + block_offset[i];
33.                 const int dir = h->intra4x4_pred_mode_cache[scan8[i]];
34.                 if (transform_bypass && h->sps.profile_idc == 244 && dir <= 1) {
35.                     if (h->x264_build != -1) {
36.                         h->hpc.pred8x8l_add[dir](ptr, h->mb + (i * 16 + p * 256 << pixel_shift), linesize);
37.                     } else
38.                         h->hpc.pred8x8l_filter_add[dir](ptr, h->mb + (i * 16 + p * 256 << pixel_shift),
39.                                                         (h-> topleft_samples_available << i) & 0x8000,
40.                                                         (h->topright_samples_available << i) & 0x4000, linesize);
41.                 } else {
42.                     const int nnz = h->non_zero_count_cache[scan8[i + p * 16]];
43.                     h->hpc.pred8x8l[dir](ptr, (h->topleft_samples_available << i) & 0x8000,
44.                                         (h->topright_samples_available << i) & 0x4000, linesize);
45.                     if (nnz) {
46.                         if (nnz == 1 && dctcoef_get(h->mb, pixel_shift, i * 16 + p * 256))
47.                             idct_dc_add(ptr, h->mb + (i * 16 + p * 256 << pixel_shift), linesize);
48.                         else
49.                             idct_add(ptr, h->mb + (i * 16 + p * 256 << pixel_shift), linesize);
50.                     }
51.                 }
52.             }
53.         } else {
54.             /*
55.              * Intra4x4帧内预测：16x16 宏块被划分为16个4x4子块
56.              *
57.              * +---+---+---+---+
58.              * |   |   |   |   |
59.              * +---+---+---+---+
60.              * |   |   |   |   |
61.              * +---+---+---+---+
62.              * |   |   |   |   |
63.              * +---+---+---+---+
64.              * |   |   |   |   |
65.              * +---+---+---+---+
66.              *
67.              */
68.             //4x4的IDCT
69.             //transform_bypass=0, 不考虑
70.             if (transform_bypass) {
71.                 idct_dc_add =
72.                 idct_add = h->h264dsp.h264_add_pixels4_clear;
73.             } else {
74.                 //常见情况
75.                 idct_dc_add = h->h264dsp.h264_idct_dc_add;
76.                 idct_add = h->h264dsp.h264_idct_add;
77.             }
78.             //循环4x4=16个DCT块
79.             for (i = 0; i < 16; i++) {
80.                 //ptr指向输出的像素数据
81.                 uint8_t *const ptr = dest_y + block_offset[i];
82.                 //dir存储了帧内预测模式
83.                 const int dir = h->intra4x4_pred_mode_cache[scan8[i]];
84.
85.                 if (transform_bypass && h->sps.profile_idc == 244 && dir <= 1) {
```

```

86.         h->hpc.pred4x4_add[dir](ptr, h->mb + (i * 16 + p * 256 << pixel_shift), linesize);
87.     } else {
88.         uint8_t *topright;
89.         int nnz, tr;
90.         uint64_t tr_high;
91.         //这2种模式特殊的处理?
92.         if (dir == DIAG_DOWN_LEFT_PRED || dir == VERT_LEFT_PRED) {
93.             const int topright_avail = (h->topright_samples_available << i) & 0x8000;
94.             av_assert2(h->mb_y || linesize <= block_offset[i]);
95.             if (!topright_avail) {
96.                 if (pixel_shift) {
97.                     tr_high = ((uint16_t *)ptr)[3 - linesize / 2] * 0x0001000100010001ULL;
98.                     topright = (uint8_t *)&tr_high;
99.                 } else {
100.                     tr = ptr[3 - linesize] * 0x01010101u;
101.                     topright = (uint8_t *)&tr;
102.                 }
103.             } else
104.                 topright = ptr + (4 << pixel_shift) - linesize;
105.         } else
106.             topright = NULL;
107.         //汇编函数: 4x4帧内预测 (9种方式: Vertical, Horizontal, DC, Plane等等。。。)
108.         h->hpc.pred4x4[dir](ptr, topright, linesize);
109.         //每个4x4块的非0系数个数的缓存
110.         nnz = h->non_zero_count_cache[scan8[i + p * 16]];
111.         //有非0系数的时候才处理
112.         //h->mb中存储了DCT系数
113.         //输出存储在ptr
114.         if (nnz) {
115.             if (is_h264) {
116.                 if (nnz == 1 && dctcoef_get(h->mb, pixel_shift, i * 16 + p * 256))
117.                     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
118.                 else
119.                     idct_add(ptr, h->mb + (i * 16 + p * 256 << pixel_shift), linesize); //4x4DCT反变换
120.             } else if (CONFIG_SVQ3_DECODER)
121.                 ff_svq3_add_idct_c(ptr, h->mb + i * 16 + p * 256, linesize, qscale, 0);
122.         }
123.     }
124. }
125. }
126. } else {
127.     /*
128.      * Intra16x16帧内预测
129.      *
130.      * +-----+-----+
131.      * |           |           |
132.      * |           |           |
133.      * |           |           |
134.      * +     +     +
135.      * |           |           |
136.      * |           |           |
137.      * |           |           |
138.      * +-----+-----+
139.      *
140.      */
141.     //汇编函数: 16x16帧内预测 (4种方式: Vertical, Horizontal, DC, Plane)
142.     h->hpc.pred16x16[h->intra16x16_pred_mode](dest_y, linesize);
143.     if (is_h264) {
144.         if (h->non_zero_count_cache[scan8[LUMA_DC_BLOCK_INDEX + p]]) {
145.             //有非0系数的时候才处理
146.             //Hadamard反变换
147.             //h->mb中存储了DCT系数
148.             //h->mb_luma_dc中存储了16个DCT的直流分量
149.             if (!transform_bypass)
150.                 h->h264dsp.h264_luma_dc_dequant_idct(h->mb + (p * 256 << pixel_shift),
151.                                                         h->mb_luma_dc[p],
152.                                                         h->dequant4_coeff[p][qscale][0]);
153.             //注: 此处仅仅进行了Hadamard反变换, 并未进行DCT反变换
154.             //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
155.             else {
156.                 static const uint8_t dc_mapping[16] = {
157.                     0 * 16, 1 * 16, 4 * 16, 5 * 16,
158.                     2 * 16, 3 * 16, 6 * 16, 7 * 16,
159.                     8 * 16, 9 * 16, 12 * 16, 13 * 16,
160.                     10 * 16, 11 * 16, 14 * 16, 15 * 16
161.                 };
162.                 for (i = 0; i < 16; i++)
163.                     dctcoef_set(h->mb + (p * 256 << pixel_shift),
164.                                   pixel_shift, dc_mapping[i],
165.                                   dctcoef_get(h->mb_luma_dc[p],
166.                                                  pixel_shift, i));
167.             }
168.         }
169.     } else if (CONFIG_SVQ3_DECODER)
170.         ff_svq3_luma_dc_dequant_idct_c(h->mb + p * 256,
171.                                         h->mb_luma_dc[p], qscale);
172. }
173. }

```

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（Intra16x16），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

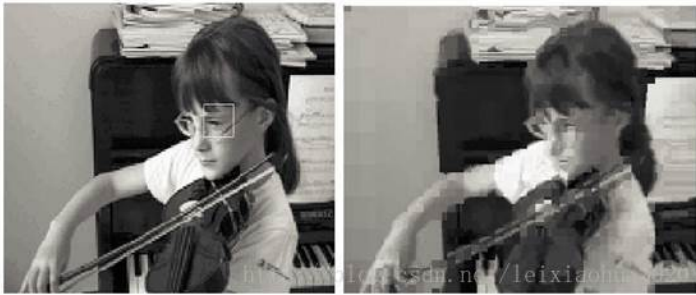
在这里需要注意，帧内4x4的宏块在执行完hl\_decode\_mb\_predict\_luma()之后实际上已经完成了“帧内预测+DCT反变换”的流程（解码完成）；而帧内16x16的宏块在执行完hl\_decode\_mb\_predict\_luma()之后仅仅完成了“帧内预测+Hadamard反变换”的流程，而并未进行“DCT反变换”的步骤，这一步骤需要在后续步骤中完成。

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）：

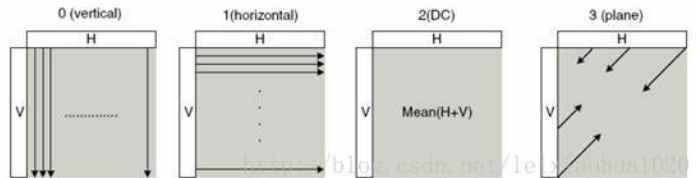
4x4帧内预测汇编函数：H264PredContext->pred4x4[dir]()  
16x16帧内预测汇编函数：H264PredContext->pred16x16[dir]()  
Hadamard反变换汇编函数：H264DSPContext->h264\_luma\_dc\_dequant\_idct()

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。



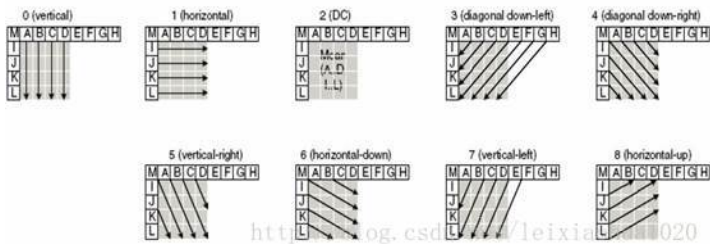
H.264中有两种帧内预测模式：16x16亮度帧内预测模式和4x4亮度帧内预测模式。其中16x16帧内预测模式一共有4种，如下图所示。



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]
1.  /**
2.   * Set the intra prediction function pointers.
3.   */
4.   //初始化帧内预测相关的汇编函数
5.   av_cold void ff_h264_pred_init(H264PredContext *h, int codec_id,
6.                                 const int bit_depth,
7.                                 int chroma_format_idc)
8.   {
9.   #undef FUNC
10.  #undef FUNCC
11.  #define FUNC(a, depth) a ## _ ## depth
12.  #define FUNCC(a, depth) a ## _ ## depth ## _c
13.  #define FUNCD(a) a ## _c
14.  //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!)
15.  //该宏用于给帧内预测模块的函数指针赋值
16.  //注意参数为颜色位深度
17.  #define H264_PRED(depth) \
18.  if(codec_id != AV_CODEC_ID_RV40){\
19.  if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {\
20.  h->pred4x4[VERT_PRED] = FUNCD(pred4x4_vertical_vp8);\
21.  h->pred4x4[HOR_PRED] = FUNCD(pred4x4_horizontal_vp8);\
22.  } else {\
23.  h->pred4x4[VERT_PRED] = FUNCC(pred4x4_vertical, depth);\
24.  h->pred4x4[HOR_PRED] = FUNCC(pred4x4_horizontal, depth);\
25.  }\
26.  h->pred4x4[DC_PRED] = FUNCC(pred4x4_dc, depth);\
27.  if(codec_id == AV_CODEC_ID_SVQ3)\
28.  h->pred4x4[DIAG_DOWN_LEFT_PRED] = FUNCD(pred4x4_down_left_svq3);\
29.  else\
30.  h->pred4x4[DIAG_DOWN_LEFT_PRED] = FUNCC(pred4x4_down_left, depth);\
31.  h->pred4x4[DIAG_DOWN_RIGHT_PRED] = FUNCC(pred4x4_down_right, depth);\
32.  h->pred4x4[VERT_RIGHT_PRED] = FUNCC(pred4x4_vertical_right, depth);\
33.  h->pred4x4[HOR_DOWN_PRED] = FUNCC(pred4x4_horizontal_down, depth);\
34.  if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {\
35.  h->pred4x4[VERT_LEFT_PRED] = FUNCD(pred4x4_vertical_left_vp8);\
36.  } else\
37.  h->pred4x4[VERT_LEFT_PRED] = FUNCC(pred4x4_vertical_left, depth);\
38.  h->pred4x4[HOR_UP_PRED] = FUNCC(pred4x4_horizontal_up, depth);\
39.  if (codec_id != AV_CODEC_ID_VP7 && codec_id != AV_CODEC_ID_VP8) {\
40.  h->pred4x4[LEFT_DC_PRED] = FUNCC(pred4x4_left_dc, depth);\
41.  h->pred4x4[TOP_DC_PRED] = FUNCC(pred4x4_top_dc, depth);\
42.  } else {\
43.  h->pred4x4[TM_VP8_PRED] = FUNCD(pred4x4_tm_vp8);\
44.  h->pred4x4[DC_127_PRED] = FUNCC(pred4x4_127_dc, depth);\
45.  h->pred4x4[DC_129_PRED] = FUNCC(pred4x4_129_dc, depth);\
46.  h->pred4x4[VERT_VP8_PRED] = FUNCC(pred4x4_vertical, depth);\
47.  h->pred4x4[HOR_VP8_PRED] = FUNCC(pred4x4_horizontal, depth);\
48.  }\
49.  if (codec_id != AV_CODEC_ID_VP8)\
50.  h->pred4x4[DC_128_PRED] = FUNCC(pred4x4_128_dc, depth);\
51.  }else{\
52.  h->pred4x4[VERT_PRED] = FUNCC(pred4x4_vertical, depth);\
53.  h->pred4x4[HOR_PRED] = FUNCC(pred4x4_horizontal, depth);\
54.  h->pred4x4[DC_PRED] = FUNCC(pred4x4_dc, depth);\
55.  h->pred4x4[DIAG_DOWN_LEFT_PRED] = FUNCD(pred4x4_down_left_rv40);\
56.  h->pred4x4[DIAG_DOWN_RIGHT_PRED] = FUNCC(pred4x4_down_right, depth);\
57.  h->pred4x4[VERT_RIGHT_PRED] = FUNCC(pred4x4_vertical_right, depth);\
58.  h->pred4x4[HOR_DOWN_PRED] = FUNCC(pred4x4_horizontal_down, depth);\
59.  h->pred4x4[VERT_LEFT_PRED] = FUNCD(pred4x4_vertical_left_rv40);\
60.  h->pred4x4[HOR_UP_PRED] = FUNCD(pred4x4_horizontal_up_rv40);\
61.  h->pred4x4[LEFT_DC_PRED] = FUNCC(pred4x4_left_dc, depth);\
62.  h->pred4x4[TOP_DC_PRED] = FUNCC(pred4x4_top_dc, depth);\
63.  h->pred4x4[DC_128_PRED] = FUNCC(pred4x4_128_dc, depth);\
64.  h->pred4x4[DIAG_DOWN_LEFT_PRED_RV40_NODOWN] = FUNCD(pred4x4_down_left_rv40_nodown);\
65.  h->pred4x4[HOR_UP_PRED_RV40_NODOWN] = FUNCD(pred4x4_horizontal_up_rv40_nodown);\
66.  h->pred4x4[VERT_LEFT_PRED_RV40_NODOWN] = FUNCD(pred4x4_vertical_left_rv40_nodown);\
67.  }\
68.  \
69.  h->pred8x8l[VERT_PRED] = FUNCC(pred8x8l_vertical, depth);\
70.  h->pred8x8l[HOR_PRED] = FUNCC(pred8x8l_horizontal, depth);\
71.  h->pred8x8l[DC_PRED] = FUNCC(pred8x8l_dc, depth);\
72.  h->pred8x8l[DIAG_DOWN_LEFT_PRED] = FUNCC(pred8x8l_down_left, depth);\
```

```

73. h->pred8x8l[DIAG_DOWN_RIGHT_PRED]= FUNCC(pred8x8l_down_right , depth);\
74. h->pred8x8l[VERT_RIGHT_PRED ]= FUNCC(pred8x8l_vertical_right , depth);\
75. h->pred8x8l[HOR_DOWN_PRED ]= FUNCC(pred8x8l_horizontal_down , depth);\
76. h->pred8x8l[VERT_LEFT_PRED ]= FUNCC(pred8x8l_vertical_left , depth);\
77. h->pred8x8l[HOR_UP_PRED ]= FUNCC(pred8x8l_horizontal_up , depth);\
78. h->pred8x8l[LEFT_DC_PRED ]= FUNCC(pred8x8l_left_dc , depth);\
79. h->pred8x8l[TOP_DC_PRED ]= FUNCC(pred8x8l_top_dc , depth);\
80. h->pred8x8l[DC_128_PRED ]= FUNCC(pred8x8l_128_dc , depth);\
81. \
82. if (chroma_format_idc <= 1) {\
83. h->pred8x8[VERT_PRED8x8 ]= FUNCC(pred8x8_vertical , depth);\
84. h->pred8x8[HOR_PRED8x8 ]= FUNCC(pred8x8_horizontal , depth);\
85. } else {\
86. h->pred8x8[VERT_PRED8x8 ]= FUNCC(pred8x16_vertical , depth);\
87. h->pred8x8[HOR_PRED8x8 ]= FUNCC(pred8x16_horizontal , depth);\
88. }\
89. if (codec_id != AV_CODEC_ID_VP7 && codec_id != AV_CODEC_ID_VP8) {\
90. if (chroma_format_idc <= 1) {\
91. h->pred8x8[PLANE_PRED8x8]= FUNCC(pred8x8_plane , depth);\
92. } else {\
93. h->pred8x8[PLANE_PRED8x8]= FUNCC(pred8x16_plane , depth);\
94. }\
95. } else{\
96. h->pred8x8[PLANE_PRED8x8]= FUNCC(pred8x8_tm_vp8);\
97. if (codec_id != AV_CODEC_ID_RV40 && codec_id != AV_CODEC_ID_VP7 && \
98. codec_id != AV_CODEC_ID_VP8) {\
99. if (chroma_format_idc <= 1) {\
100. h->pred8x8[DC_PRED8x8 ]= FUNCC(pred8x8_dc , depth);\
101. h->pred8x8[LEFT_DC_PRED8x8]= FUNCC(pred8x8_left_dc , depth);\
102. h->pred8x8[TOP_DC_PRED8x8 ]= FUNCC(pred8x8_top_dc , depth);\
103. h->pred8x8[ALZHEIMER_DC_L0T_PRED8x8 ]= FUNC(pred8x8_mad_cow_dc_l0t, depth);\
104. h->pred8x8[ALZHEIMER_DC_0LT_PRED8x8 ]= FUNC(pred8x8_mad_cow_dc_0lt, depth);\
105. h->pred8x8[ALZHEIMER_DC_L00_PRED8x8 ]= FUNC(pred8x8_mad_cow_dc_l00, depth);\
106. h->pred8x8[ALZHEIMER_DC_0L0_PRED8x8 ]= FUNC(pred8x8_mad_cow_dc_0l0, depth);\
107. } else {\
108. h->pred8x8[DC_PRED8x8 ]= FUNCC(pred8x16_dc , depth);\
109. h->pred8x8[LEFT_DC_PRED8x8]= FUNCC(pred8x16_left_dc , depth);\
110. h->pred8x8[TOP_DC_PRED8x8 ]= FUNCC(pred8x16_top_dc , depth);\
111. h->pred8x8[ALZHEIMER_DC_L0T_PRED8x8 ]= FUNC(pred8x16_mad_cow_dc_l0t, depth);\
112. h->pred8x8[ALZHEIMER_DC_0LT_PRED8x8 ]= FUNC(pred8x16_mad_cow_dc_0lt, depth);\
113. h->pred8x8[ALZHEIMER_DC_L00_PRED8x8 ]= FUNC(pred8x16_mad_cow_dc_l00, depth);\
114. h->pred8x8[ALZHEIMER_DC_0L0_PRED8x8 ]= FUNC(pred8x16_mad_cow_dc_0l0, depth);\
115. }\
116. }else{\
117. h->pred8x8[DC_PRED8x8 ]= FUNCC(pred8x8_dc_rv40);\
118. h->pred8x8[LEFT_DC_PRED8x8]= FUNCC(pred8x8_left_dc_rv40);\
119. h->pred8x8[TOP_DC_PRED8x8 ]= FUNCC(pred8x8_top_dc_rv40);\
120. if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {\
121. h->pred8x8[DC_127_PRED8x8]= FUNCC(pred8x8_127_dc , depth);\
122. h->pred8x8[DC_129_PRED8x8]= FUNCC(pred8x8_129_dc , depth);\
123. }\
124. }\
125. if (chroma_format_idc <= 1) {\
126. h->pred8x8[DC_128_PRED8x8 ]= FUNCC(pred8x8_128_dc , depth);\
127. } else {\
128. h->pred8x8[DC_128_PRED8x8 ]= FUNCC(pred8x16_128_dc , depth);\
129. }\
130. \
131. h->pred16x16[DC_PRED8x8 ]= FUNCC(pred16x16_dc , depth);\
132. h->pred16x16[VERT_PRED8x8 ]= FUNCC(pred16x16_vertical , depth);\
133. h->pred16x16[HOR_PRED8x8 ]= FUNCC(pred16x16_horizontal , depth);\
134. switch(codec_id){\
135. case AV_CODEC_ID_SVQ3:\
136. h->pred16x16[PLANE_PRED8x8 ]= FUNCC(pred16x16_plane_svq3);\
137. break;\
138. case AV_CODEC_ID_RV40:\
139. h->pred16x16[PLANE_PRED8x8 ]= FUNCC(pred16x16_plane_rv40);\
140. break;\
141. case AV_CODEC_ID_VP7:\
142. case AV_CODEC_ID_VP8:\
143. h->pred16x16[PLANE_PRED8x8 ]= FUNCC(pred16x16_tm_vp8);\
144. h->pred16x16[DC_127_PRED8x8]= FUNCC(pred16x16_127_dc , depth);\
145. h->pred16x16[DC_129_PRED8x8]= FUNCC(pred16x16_129_dc , depth);\
146. break;\
147. default:\
148. h->pred16x16[PLANE_PRED8x8 ]= FUNCC(pred16x16_plane , depth);\
149. break;\
150. }\
151. h->pred16x16[LEFT_DC_PRED8x8]= FUNCC(pred16x16_left_dc , depth);\
152. h->pred16x16[TOP_DC_PRED8x8 ]= FUNCC(pred16x16_top_dc , depth);\
153. h->pred16x16[DC_128_PRED8x8 ]= FUNCC(pred16x16_128_dc , depth);\
154. \
155. /* special lossless h/v prediction for h264 */ \
156. h->pred4x4_add [VERT_PRED ]= FUNCC(pred4x4_vertical_add , depth);\
157. h->pred4x4_add [ HOR_PRED ]= FUNCC(pred4x4_horizontal_add , depth);\
158. h->pred8x8l_add [VERT_PRED ]= FUNCC(pred8x8l_vertical_add , depth);\
159. h->pred8x8l_add [ HOR_PRED ]= FUNCC(pred8x8l_horizontal_add , depth);\
160. h->pred8x8l_filter_add [VERT_PRED ]= FUNCC(pred8x8l_vertical_filter_add , depth);\
161. h->pred8x8l_filter_add [ HOR_PRED ]= FUNCC(pred8x8l_horizontal_filter_add , depth);\
162. if (chroma_format_idc <= 1) {\
163. h->pred8x8_add [VERT_PRED8x8]= FUNCC(pred8x8_vertical_add , depth);\

```

```

164.     h->pred8x8_add [ HOR_PRED8x8]= FUNCC(pred8x8_horizontal_add , depth);\
165. } else {\
166.     h->pred8x8_add [VERT_PRED8x8]= FUNCC(pred8x16_vertical_add , depth);\
167.     h->pred8x8_add [ HOR_PRED8x8]= FUNCC(pred8x16_horizontal_add , depth);\
168. }\
169. h->pred16x16_add[VERT_PRED8x8]= FUNCC(pred16x16_vertical_add , depth);\
170. h->pred16x16_add[ HOR_PRED8x8]= FUNCC(pred16x16_horizontal_add , depth);\
171. //注意这里使用了前面那个很长的宏定义
172. //根据颜色位深的不同, 初始化不同的函数
173. //颜色位深默认值为8, 所以一般情况下调用H264_PRED(8)
174. switch (bit_depth) {
175.     case 9:
176.         H264_PRED(9)
177.         break;
178.     case 10:
179.         H264_PRED(10)
180.         break;
181.     case 12:
182.         H264_PRED(12)
183.         break;
184.     case 14:
185.         H264_PRED(14)
186.         break;
187.     default:
188.         av_assert0(bit_depth<=8);
189.         H264_PRED(8)
190.         break;
191. }
192.
193.
194. //如果支持汇编优化, 则会调用相应的汇编优化函数
195. //neon这些的
196. if (ARCH_ARM) ff_h264_pred_init_arm(h, codec_id, bit_depth, chroma_format_idc);
197. //mmx这些的
198. if (ARCH_X86) ff_h264_pred_init_x86(h, codec_id, bit_depth, chroma_format_idc);
199. }

```

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。

```

1.  if(codec_id != AV_CODEC_ID_RV40){
2.      if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {
3.          h->pred4x4[0]   ]= pred4x4_vertical_vp8_c;
4.          h->pred4x4[1]   ]= pred4x4_horizontal_vp8_c;
5.      } else {
6.          //帧内4x4的Vertical预测方式
7.          h->pred4x4[0]   ]= pred4x4_vertical_8_c;
8.          //帧内4x4的Horizontal预测方式
9.          h->pred4x4[1]   ]= pred4x4_horizontal_8_c;
10.     }
11.     //帧内4x4的DC预测方式
12.     h->pred4x4[2]       ]= pred4x4_dc_8_c;
13.     if(codec_id == AV_CODEC_ID_SVQ3)
14.         h->pred4x4[3] ]= pred4x4_down_left_svq3_c;
15.     else
16.         h->pred4x4[3] ]= pred4x4_down_left_8_c;
17.     h->pred4x4[4]= pred4x4_down_right_8_c;
18.     h->pred4x4[5]   ]= pred4x4_vertical_right_8_c;
19.     h->pred4x4[6]   ]= pred4x4_horizontal_down_8_c;
20.     if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {
21.         h->pred4x4[7] ]= pred4x4_vertical_left_vp8_c;
22.     } else
23.         h->pred4x4[7] ]= pred4x4_vertical_left_8_c;
24.     h->pred4x4[8]   ]= pred4x4_horizontal_up_8_c;
25.     if (codec_id != AV_CODEC_ID_VP7 && codec_id != AV_CODEC_ID_VP8) {
26.         h->pred4x4[9] ]= pred4x4_left_dc_8_c;
27.         h->pred4x4[10] ]= pred4x4_top_dc_8_c;
28.     } else {
29.         h->pred4x4[9]   ]= pred4x4_tm_vp8_c;
30.         h->pred4x4[12]   ]= pred4x4_127_dc_8_c;
31.         h->pred4x4[13]   ]= pred4x4_129_dc_8_c;
32.         h->pred4x4[10]   ]= pred4x4_vertical_8_c;
33.         h->pred4x4[14]   ]= pred4x4_horizontal_8_c;
34.     }
35.     if (codec_id != AV_CODEC_ID_VP8)
36.         h->pred4x4[11]   ]= pred4x4_128_dc_8_c;
37. }

```

```

37.     }
38.     h->pred4x4[0]      ]= pred4x4_vertical_8_c;
39.     h->pred4x4[1]      ]= pred4x4_horizontal_8_c;
40.     h->pred4x4[2]      ]= pred4x4_dc_8_c;
41.     h->pred4x4[3] ]= pred4x4_down_left_rv40_c;
42.     h->pred4x4[4]= pred4x4_down_right_8_c;
43.     h->pred4x4[5]      ]= pred4x4_vertical_right_8_c;
44.     h->pred4x4[6]      ]= pred4x4_horizontal_down_8_c;
45.     h->pred4x4[7]      ]= pred4x4_vertical_left_rv40_c;
46.     h->pred4x4[8]      ]= pred4x4_horizontal_up_rv40_c;
47.     h->pred4x4[9]      ]= pred4x4_left_dc_8_c;
48.     h->pred4x4[10]     ]= pred4x4_top_dc_8_c;
49.     h->pred4x4[11]     ]= pred4x4_128_dc_8_c;
50.     h->pred4x4[12]= pred4x4_down_left_rv40_nodown_c;
51.     h->pred4x4[13]= pred4x4_horizontal_up_rv40_nodown_c;
52.     h->pred4x4[14]= pred4x4_vertical_left_rv40_nodown_c;
53. }
54.
55. h->pred8x8l[0]      ]= pred8x8l_vertical_8_c;
56. h->pred8x8l[1]      ]= pred8x8l_horizontal_8_c;
57. h->pred8x8l[2]      ]= pred8x8l_dc_8_c;
58. h->pred8x8l[3] ]= pred8x8l_down_left_8_c;
59. h->pred8x8l[4]= pred8x8l_down_right_8_c;
60. h->pred8x8l[5]      ]= pred8x8l_vertical_right_8_c;
61. h->pred8x8l[6]      ]= pred8x8l_horizontal_down_8_c;
62. h->pred8x8l[7]      ]= pred8x8l_vertical_left_8_c;
63. h->pred8x8l[8]      ]= pred8x8l_horizontal_up_8_c;
64. h->pred8x8l[9]      ]= pred8x8l_left_dc_8_c;
65. h->pred8x8l[10]     ]= pred8x8l_top_dc_8_c;
66. h->pred8x8l[11]     ]= pred8x8l_128_dc_8_c;
67.
68. if (chroma_format_idc <= 1) {
69.     h->pred8x8[2] ]= pred8x8_vertical_8_c;
70.     h->pred8x8[1] ]= pred8x8_horizontal_8_c;
71. } else {
72.     h->pred8x8[2] ]= pred8x16_vertical_8_c;
73.     h->pred8x8[1] ]= pred8x16_horizontal_8_c;
74. }
75. if (codec_id != AV_CODEC_ID_VP7 && codec_id != AV_CODEC_ID_VP8) {
76.     if (chroma_format_idc <= 1) {
77.         h->pred8x8[3]= pred8x8_plane_8_c;
78.     } else {
79.         h->pred8x8[3]= pred8x16_plane_8_c;
80.     }
81. } else
82.     h->pred8x8[3]= pred8x8_tm_vp8_c;
83. if (codec_id != AV_CODEC_ID_RV40 && codec_id != AV_CODEC_ID_VP7 &&
84.     codec_id != AV_CODEC_ID_VP8) {
85.     if (chroma_format_idc <= 1) {
86.         h->pred8x8[0]      ]= pred8x8_dc_8_c;
87.         h->pred8x8[4]= pred8x8_left_dc_8_c;
88.         h->pred8x8[5] ]= pred8x8_top_dc_8_c;
89.         h->pred8x8[7] ]= pred8x8_mad_cow_dc_l0t_8;
90.         h->pred8x8[8] ]= pred8x8_mad_cow_dc_0lt_8;
91.         h->pred8x8[9] ]= pred8x8_mad_cow_dc_l00_8;
92.         h->pred8x8[10] ]= pred8x8_mad_cow_dc_0l0_8;
93.     } else {
94.         h->pred8x8[0]      ]= pred8x16_dc_8_c;
95.         h->pred8x8[4]= pred8x16_left_dc_8_c;
96.         h->pred8x8[5] ]= pred8x16_top_dc_8_c;
97.         h->pred8x8[7] ]= pred8x16_mad_cow_dc_l0t_8;
98.         h->pred8x8[8] ]= pred8x16_mad_cow_dc_0lt_8;
99.         h->pred8x8[9] ]= pred8x16_mad_cow_dc_l00_8;
100.        h->pred8x8[10] ]= pred8x16_mad_cow_dc_0l0_8;
101.    }
102. } else {
103.     h->pred8x8[0]      ]= pred8x8_dc_rv40_c;
104.     h->pred8x8[4]= pred8x8_left_dc_rv40_c;
105.     h->pred8x8[5] ]= pred8x8_top_dc_rv40_c;
106.     if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {
107.         h->pred8x8[7]= pred8x8_127_dc_8_c;
108.         h->pred8x8[8]= pred8x8_129_dc_8_c;
109.     }
110. }
111. if (chroma_format_idc <= 1) {
112.     h->pred8x8[6] ]= pred8x8_128_dc_8_c;
113. } else {
114.     h->pred8x8[6] ]= pred8x16_128_dc_8_c;
115. }
116.
117. h->pred16x16[0]      ]= pred16x16_dc_8_c;
118. h->pred16x16[2]      ]= pred16x16_vertical_8_c;
119. h->pred16x16[1]      ]= pred16x16_horizontal_8_c;
120. switch(codec_id){
121. case AV_CODEC_ID_SVQ3:
122.     h->pred16x16[3] ]= pred16x16_plane_svq3_c;
123.     break;
124. case AV_CODEC_ID_RV40:
125.     h->pred16x16[3] ]= pred16x16_plane_rv40_c;
126.     break;
127. case AV_CODEC_ID_VP7:
128. case AV_CODEC_ID_VP8:

```



```

129.     h->pred16x16[3] = pred16x16_tm_vp8_c;
130.     h->pred16x16[7] = pred16x16_127_dc_8_c;
131.     h->pred16x16[8] = pred16x16_129_dc_8_c;
132.     break;
133. default:
134.     h->pred16x16[3] = pred16x16_plane_8_c;
135.     break;
136. }
137. h->pred16x16[4] = pred16x16_left_dc_8_c;
138. h->pred16x16[5] = pred16x16_top_dc_8_c;
139. h->pred16x16[6] = pred16x16_128_dc_8_c;
140.
141. /* special lossless h/v prediction for h264 */
142. h->pred4x4_add [0] = pred4x4_vertical_add_8_c;
143. h->pred4x4_add [1] = pred4x4_horizontal_add_8_c;
144. h->pred8x8l_add [0] = pred8x8l_vertical_add_8_c;
145. h->pred8x8l_add [1] = pred8x8l_horizontal_add_8_c;
146. h->pred8x8l_filter_add [0] = pred8x8l_vertical_filter_add_8_c;
147. h->pred8x8l_filter_add [1] = pred8x8l_horizontal_filter_add_8_c;
148. if (chroma_format_idc <= 1) {
149.     h->pred8x8_add [2] = pred8x8_vertical_add_8_c;
150.     h->pred8x8_add [1] = pred8x8_horizontal_add_8_c;
151. } else {
152.     h->pred8x8_add [2] = pred8x16_vertical_add_8_c;
153.     h->pred8x8_add [1] = pred8x16_horizontal_add_8_c;
154. }
155. h->pred16x16_add[2] = pred16x16_vertical_add_8_c;
156. 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\_dc\_8\_c()，如下所示。

```

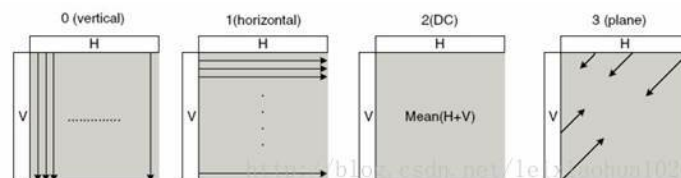
1. //帧内4x4的Vertical预测方式
2. h->pred4x4[0] = pred4x4_vertical_8_c;
3. //帧内4x4的Horizontal预测方式
4. h->pred4x4[1] = pred4x4_horizontal_8_c;
5. //帧内4x4的DC预测方式
6. h->pred4x4[2] = pred4x4_dc_8_c;

```

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。



```

1.  /* 帧内预测
2.  *
3.  * 注释：雷霄骅
4.  * leixiaohua1020@126.com
5.  * http://blog.csdn.net/leixiaohua1020
6.  *
7.  * 参数：
8.  * _src：输入数据
9.  * _stride：一行像素的大小
10. *
11. */
12. //垂直预测
13. //由上边像素推出像素值
14. static void FUNC(pred4x4_vertical)(uint8_t * _src, const uint8_t *topright,
15.                                   ptrdiff_t _stride)
16. {
17.     pixel *_src = (pixel*)_src;
18.     int stride = _stride>>(sizeof(pixel)-1);
19.
20.     /*
21.      * Vertical预测方式
22.      * |X1 X2 X3 X4
23.      * --+-----
24.      * |X1 X2 X3 X4
25.      * |X1 X2 X3 X4
26.      * |X1 X2 X3 X4
27.      * |X1 X2 X3 X4
28.      *
29.      */
30.
31.     //pixel4代表4个像素值。1个像素值占用8bit，4个像素值占用32bit。
32.     const pixel4 a= AV_RN4PA(src-stride);
33.     /* 宏定义展开后：
34.      * const uint32_t a=((const av_alias32*)(src-stride))->u32);
35.      * 注：av_alias32是一个union类型的变量，存储4byte数据。
36.      * -stride代表了上一行对应位置的像素
37.      * 即a取的是上1行像素的值。
38.      */
39.     AV_WN4PA(src+0*stride, a);
40.     AV_WN4PA(src+1*stride, a);
41.     AV_WN4PA(src+2*stride, a);
42.     AV_WN4PA(src+3*stride, a);
43.
44.     /* 宏定义展开后：
45.      * (((av_alias32*)(src+0*stride))->u32 = (a));
46.      * (((av_alias32*)(src+1*stride))->u32 = (a));
47.      * (((av_alias32*)(src+2*stride))->u32 = (a));
48.      * (((av_alias32*)(src+3*stride))->u32 = (a));
49.      * 即把a的值赋给下面4行。
50.      */
51.
52. }

```

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。

```

1. //水平预测
2. //由左边像素推出像素值
3. static void FUNC(pred4x4_horizontal)(uint8_t *_src, const uint8_t *topright,
4.                                   ptrdiff_t _stride)
5. {
6.     pixel *_src = (pixel*)_src;
7.     int stride = _stride>>(sizeof(pixel)-1);
8.
9.     /*
10.      * Horizontal预测方式
11.      * |
12.      * --+-----
13.      * X5|X5 X5 X5 X5
14.      * X6|X6 X6 X6 X6
15.      * X7|X7 X7 X7 X7
16.      * X8|X8 X8 X8 X8
17.      *
18.      */
19.
20.     AV_WN4PA(src+0*stride, PIXEL_SPLAT_X4(src[-1+0*stride]));
21.     AV_WN4PA(src+1*stride, PIXEL_SPLAT_X4(src[-1+1*stride]));
22.     AV_WN4PA(src+2*stride, PIXEL_SPLAT_X4(src[-1+2*stride]));
23.     AV_WN4PA(src+3*stride, PIXEL_SPLAT_X4(src[-1+3*stride]));
24.     /* 宏定义展开后：
25.      * (((av_alias32*)(src+0*stride))->u32 = (((src[-1+0*stride])*0x01010101U)));
26.      * (((av_alias32*)(src+1*stride))->u32 = (((src[-1+1*stride])*0x01010101U)));
27.      * (((av_alias32*)(src+2*stride))->u32 = (((src[-1+2*stride])*0x01010101U)));
28.      * (((av_alias32*)(src+3*stride))->u32 = (((src[-1+3*stride])*0x01010101U)));
29.      *
30.      * 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）
31.      * 即把0x0100009F变成0x9F9F9F9F
32.      * 推导：
33.      * 前提是x占8bit（对应1个像素）
34.      * y=x*0x01010101
35.      * =x*(0x00000001+0x00000100+0x00010000+0x01000000)
36.      * =x<<0+x<<8+x<<16+x<<24
37.      *
38.      * 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
39.      */
40. }

```

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：

```

1. # 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。

```

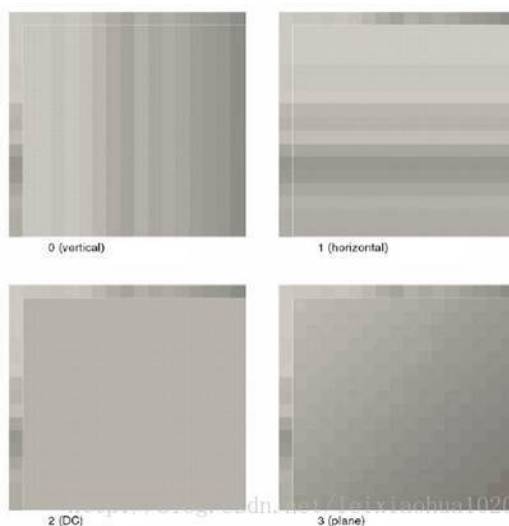
1. //DC预测
2. //由左边和上边像素平均值推出像素值
3. static void FUNC(pred4x4_dc)(uint8_t *_src, const uint8_t *topright,
4.                             ptrdiff_t _stride)
5. {
6.     pixel *_src = (pixel*)_src;
7.     int stride = _stride>>(sizeof(pixel)-1);
8.     /*
9.      * DC预测方式
10.     * |X1 X2 X3 X4
11.     * --+-----
12.     * X5|
13.     * X6|   Y
14.     * X7|
15.     * X8|
16.     *
17.     * Y=(X1+X2+X3+X4+X5+X6+X7+X8)/8
18.     */
19.     const int dc= ( _src[-stride] + src[1-stride] + src[2-stride] + src[3-stride]
20.                   + src[-1+0*stride] + src[-1+1*stride] + src[-1+2*stride] + src[-1+3*stride] + 4) >>3;
21.     const pixel4 a = PIXEL_SPLAT_X4(dc);
22.
23.     AV_WN4PA(src+0*stride, a);
24.     AV_WN4PA(src+1*stride, a);
25.     AV_WN4PA(src+2*stride, a);
26.     AV_WN4PA(src+3*stride, a);
27.     /* 宏定义展开后:
28.     * (((av_alias32*)(src+0*stride))->u32 = (a))
29.     * (((av_alias32*)(src+1*stride))->u32 = (a))
30.     * (((av_alias32*)(src+2*stride))->u32 = (a))
31.     * (((av_alias32*)(src+3*stride))->u32 = (a))
32.     */
33. }

```

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]
1. //垂直预测
2. //由上面的函数推出像素值
3. static void FUNC(pred16x16_vertical)(uint8_t *_src, ptrdiff_t _stride)
4. {
5.     /*
6.      * Vertical预测方式
7.      * |X1 X2 X3 X4
8.      * --+-----
9.      * |X1 X2 X3 X4
10.     * |X1 X2 X3 X4
11.     * |X1 X2 X3 X4
12.     * |X1 X2 X3 X4
13.     */
14.     /*
15.     int i;
16.     pixel *_src = (pixel*)_src;
17.     int stride = _stride>>(sizeof(pixel)-1);
18.     //pixel4实际上就是uint32_t, 存储4个像素的值 (每个像素8bit)
19.     //src-stride表示取上面一行像素的值
20.     //在这里取了16个像素的值, 分别存入a, b, c, d四个变量
21.     const pixel4 a = AV_RN4PA(((pixel4*)(src-stride))+0);
22.     const pixel4 b = AV_RN4PA(((pixel4*)(src-stride))+1);
23.     const pixel4 c = AV_RN4PA(((pixel4*)(src-stride))+2);
24.     const pixel4 d = AV_RN4PA(((pixel4*)(src-stride))+3);
25.     //循环16行
26.     for(i=0; i<16; i++){
27.         //分别赋值每行 (每次赋值4个像素, 赋值4次)
28.         AV_WN4PA(((pixel4*)(src+i*stride))+0, a);
29.         AV_WN4PA(((pixel4*)(src+i*stride))+1, b);
30.         AV_WN4PA(((pixel4*)(src+i*stride))+2, c);
31.         AV_WN4PA(((pixel4*)(src+i*stride))+3, d);
32.     }
33. }
```

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。

$$Y_D = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} W_D \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} / 2$$

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。

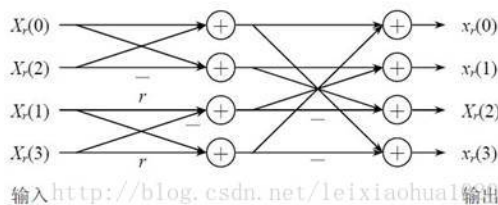
00	01	02	03
0	1	4	5
10	11	12	13
2	3	6	7
20	21	22	23
8	9	12	13
30	31	32	33
10	11	14	15

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。

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

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]
1.  /**
2.   * IDCT transforms the 16 dc values and dequantizes them.
3.   * @param qmul quantization parameter
4.   */
5.   //DCT直流系数的Hadamard反变换-亮度
6.   //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
7.   //
8.   void FUNC(ff_h264_luma_dc_dequant_idct)(int16_t *_output, int16_t *_input, int qmul){
9.   #define stride 16
10.    int i;
11.    int temp[16];
12.    static const uint8_t x_offset[4]={0, 2*stride, 8*stride, 10*stride};
13.    dctcoef *input = (dctcoef*)_input;
14.    dctcoef *output = (dctcoef*)_output;
15.
16.    for(i=0; i<4; i++){
17.        const int z0= input[4*i+0] + input[4*i+1];
18.        const int z1= input[4*i+0] - input[4*i+1];
19.        const int z2= input[4*i+2] - input[4*i+3];
20.        const int z3= input[4*i+2] + input[4*i+3];
21.
22.        temp[4*i+0]= z0+z3;
23.        temp[4*i+1]= z0-z3;
24.        temp[4*i+2]= z1-z2;
25.        temp[4*i+3]= z1+z2;
26.    }
27.
28.    for(i=0; i<4; i++){
29.        const int offset= x_offset[i];
30.        const int z0= temp[4*0+i] + temp[4*2+i];
31.        const int z1= temp[4*0+i] - temp[4*2+i];
32.        const int z2= temp[4*1+i] - temp[4*3+i];
33.        const int z3= temp[4*1+i] + temp[4*3+i];
34.
35.        output[stride* 0+offset]= (((z0 + z3)*qmul + 128 ) >> 8));
36.        output[stride* 1+offset]= (((z1 + z2)*qmul + 128 ) >> 8));
37.        output[stride* 4+offset]= (((z1 - z2)*qmul + 128 ) >> 8));
38.        output[stride* 5+offset]= (((z0 - z3)*qmul + 128 ) >> 8));
39.    }
40.    #undef stride
41. }
```

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]
1.  av_cold void ff_h264_pred_init_x86(H264PredContext *h, int codec_id,
2.                                     const int bit_depth,
3.                                     const int chroma_format_idc)
4.  {
5.      int cpu_flags = av_get_cpu_flags();
6.
7.      if (bit_depth == 8) {
8.          if (EXTERNAL_MMX(cpu_flags)) {
9.              h->pred16x16[VERT_PRED8x8] = ff_pred16x16_vertical_8_mmx;
10.             h->pred16x16[HOR_PRED8x8] = ff_pred16x16_horizontal_8_mmx;
11.             if (chroma_format_idc <= 1) {
12.                 h->pred8x8 [VERT_PRED8x8] = ff_pred8x8_vertical_8_mmx;
13.                 h->pred8x8 [HOR_PRED8x8] = ff_pred8x8_horizontal_8_mmx;
14.             }
15.             if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {
16.                 h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_tm_vp8_8_mmx;
17.                 h->pred8x8 [PLANE_PRED8x8] = ff_pred8x8_tm_vp8_8_mmx;
18.                 h->pred4x4 [TM_VP8_PRED] = ff_pred4x4_tm_vp8_8_mmx;
19.             } else {
20.                 if (chroma_format_idc <= 1)
21.                     h->pred8x8 [PLANE_PRED8x8] = ff_pred8x8_plane_8_mmx;
22.                 if (codec_id == AV_CODEC_ID_SVQ3) {
23.                     if (cpu_flags & AV_CPU_FLAG_CMOV)
24.                         h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_svq3_8_mmx;
25.                     } else if (codec_id == AV_CODEC_ID_RV40) {
26.                         h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_rv40_8_mmx;
27.                     } else {
28.                         h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_h264_8_mmx;
29.                     }
30.                 }
31.             }
32.
33.             if (EXTERNAL_MMXEXT(cpu_flags)) {
34.                 h->pred16x16[HOR_PRED8x8] = ff_pred16x16_horizontal_8_mmxext;
35.                 h->pred16x16[DC_PRED8x8] = ff_pred16x16_dc_8_mmxext;
36.                 if (chroma_format_idc <= 1)
37.                     h->pred8x8[HOR_PRED8x8] = ff_pred8x8_horizontal_8_mmxext;
38.                 h->pred8x8l [TOP_DC_PRED] = ff_pred8x8l_top_dc_8_mmxext;
39.                 h->pred8x8l [DC_PRED] = ff_pred8x8l_dc_8_mmxext;
40.                 h->pred8x8l [HOR_PRED] = ff_pred8x8l_horizontal_8_mmxext;
41.                 h->pred8x8l [VERT_PRED] = ff_pred8x8l_vertical_8_mmxext;
42.                 h->pred8x8l [DIAG_DOWN_RIGHT_PRED] = ff_pred8x8l_down_right_8_mmxext;
43.                 h->pred8x8l [VERT_RIGHT_PRED] = ff_pred8x8l_vertical_right_8_mmxext;
44.                 h->pred8x8l [HOR_UP_PRED] = ff_pred8x8l_horizontal_up_8_mmxext;
45.                 h->pred8x8l [DIAG_DOWN_LEFT_PRED] = ff_pred8x8l_down_left_8_mmxext;
46.                 h->pred8x8l [HOR_DOWN_PRED] = ff_pred8x8l_horizontal_down_8_mmxext;
47.                 h->pred4x4 [DIAG_DOWN_RIGHT_PRED] = ff_pred4x4_down_right_8_mmxext;
48.                 h->pred4x4 [VERT_RIGHT_PRED] = ff_pred4x4_vertical_right_8_mmxext;
49.                 h->pred4x4 [HOR_DOWN_PRED] = ff_pred4x4_horizontal_down_8_mmxext;
50.                 h->pred4x4 [DC_PRED] = ff_pred4x4_dc_8_mmxext;
51.                 if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8 ||
52.                     codec_id == AV_CODEC_ID_H264) {
53.                     h->pred4x4 [DIAG_DOWN_LEFT_PRED] = ff_pred4x4_down_left_8_mmxext;
54.                 }
55.                 if (codec_id == AV_CODEC_ID_SVQ3 || codec_id == AV_CODEC_ID_H264) {
56.                     h->pred4x4 [VERT_LEFT_PRED] = ff_pred4x4_vertical_left_8_mmxext;
57.                 }
58.                 if (codec_id != AV_CODEC_ID_RV40) {
59.                     h->pred4x4 [HOR_UP_PRED] = ff_pred4x4_horizontal_up_8_mmxext;
60.                 }
61.                 if (codec_id == AV_CODEC_ID_SVQ3 || codec_id == AV_CODEC_ID_H264) {
62.                     if (chroma_format_idc <= 1) {
63.                         h->pred8x8[TOP_DC_PRED8x8] = ff_pred8x8_top_dc_8_mmxext;
64.                         h->pred8x8[DC_PRED8x8] = ff_pred8x8_dc_8_mmxext;
65.                     }
66.                 }
67.                 if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {
68.                     h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_tm_vp8_8_mmxext;
69.                     h->pred8x8 [DC_PRED8x8] = ff_pred8x8_dc_rv40_8_mmxext;
70.                     h->pred8x8 [PLANE_PRED8x8] = ff_pred8x8_tm_vp8_8_mmxext;
71.                     h->pred4x4 [TM_VP8_PRED] = ff_pred4x4_tm_vp8_8_mmxext;
72.                     h->pred4x4 [VERT_PRED] = ff_pred4x4_vertical_vp8_8_mmxext;
73.                 } else {
74.                     if (chroma_format_idc <= 1)
75.                         h->pred8x8 [PLANE_PRED8x8] = ff_pred8x8_plane_8_mmxext;
76.                     if (codec_id == AV_CODEC_ID_SVQ3) {
77.                         h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_svq3_8_mmxext;
78.                     } else if (codec_id == AV_CODEC_ID_RV40) {
79.                         h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_rv40_8_mmxext;
80.                     } else {
81.                         h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_h264_8_mmxext;
82.                     }
83.                 }
84.             }
85.
86.             if (EXTERNAL_SSE(cpu_flags)) {
87.                 h->pred16x16[VERT_PRED8x8] = ff_pred16x16_vertical_8_sse;
88.             }
89.         }
90.     }
91. }
```

```

88.     }
89.
90.     if (EXTERNAL_SSE2(cpu_flags)) {
91.         h->pred16x16[DC_PRED8x8] = ff_pred16x16_dc_8_sse2;
92.         h->pred8x8l [DIAG_DOWN_LEFT_PRED] = ff_pred8x8l_down_left_8_sse2;
93.         h->pred8x8l [DIAG_DOWN_RIGHT_PRED] = ff_pred8x8l_down_right_8_sse2;
94.         h->pred8x8l [VERT_RIGHT_PRED] = ff_pred8x8l_vertical_right_8_sse2;
95.         h->pred8x8l [VERT_LEFT_PRED] = ff_pred8x8l_vertical_left_8_sse2;
96.         h->pred8x8l [HOR_DOWN_PRED] = ff_pred8x8l_horizontal_down_8_sse2;
97.         if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {
98.             h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_tm_vp8_8_sse2;
99.             h->pred8x8 [PLANE_PRED8x8] = ff_pred8x8_tm_vp8_8_sse2;
100.        } else {
101.            if (chroma_format_idc <= 1)
102.                h->pred8x8 [PLANE_PRED8x8] = ff_pred8x8_plane_8_sse2;
103.            if (codec_id == AV_CODEC_ID_SVQ3) {
104.                h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_svq3_8_sse2;
105.            } else if (codec_id == AV_CODEC_ID_RV40) {
106.                h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_rv40_8_sse2;
107.            } else {
108.                h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_h264_8_sse2;
109.            }
110.        }
111.    }
112.
113.    if (EXTERNAL_SSSE3(cpu_flags)) {
114.        h->pred16x16[HOR_PRED8x8] = ff_pred16x16_horizontal_8_ssse3;
115.        h->pred16x16[DC_PRED8x8] = ff_pred16x16_dc_8_ssse3;
116.        if (chroma_format_idc <= 1)
117.            h->pred8x8 [HOR_PRED8x8] = ff_pred8x8_horizontal_8_ssse3;
118.        h->pred8x8l [TOP_DC_PRED] = ff_pred8x8l_top_dc_8_ssse3;
119.        h->pred8x8l [DC_PRED] = ff_pred8x8l_dc_8_ssse3;
120.        h->pred8x8l [HOR_PRED] = ff_pred8x8l_horizontal_8_ssse3;
121.        h->pred8x8l [VERT_PRED] = ff_pred8x8l_vertical_8_ssse3;
122.        h->pred8x8l [DIAG_DOWN_LEFT_PRED] = ff_pred8x8l_down_left_8_ssse3;
123.        h->pred8x8l [DIAG_DOWN_RIGHT_PRED] = ff_pred8x8l_down_right_8_ssse3;
124.        h->pred8x8l [VERT_RIGHT_PRED] = ff_pred8x8l_vertical_right_8_ssse3;
125.        h->pred8x8l [VERT_LEFT_PRED] = ff_pred8x8l_vertical_left_8_ssse3;
126.        h->pred8x8l [HOR_UP_PRED] = ff_pred8x8l_horizontal_up_8_ssse3;
127.        h->pred8x8l [HOR_DOWN_PRED] = ff_pred8x8l_horizontal_down_8_ssse3;
128.        if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {
129.            h->pred8x8l [PLANE_PRED8x8] = ff_pred8x8_tm_vp8_8_ssse3;
130.            h->pred4x4 [TM_VP8_PRED] = ff_pred4x4_tm_vp8_8_ssse3;
131.        } else {
132.            if (chroma_format_idc <= 1)
133.                h->pred8x8 [PLANE_PRED8x8] = ff_pred8x8_plane_8_ssse3;
134.            if (codec_id == AV_CODEC_ID_SVQ3) {
135.                h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_svq3_8_ssse3;
136.            } else if (codec_id == AV_CODEC_ID_RV40) {
137.                h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_rv40_8_ssse3;
138.            } else {
139.                h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_h264_8_ssse3;
140.            }
141.        }
142.    }
143. } else if (bit_depth == 10) {
144.     if (EXTERNAL_MMEXT(cpu_flags)) {
145.         h->pred4x4[DC_PRED] = ff_pred4x4_dc_10_mmext;
146.         h->pred4x4[HOR_UP_PRED] = ff_pred4x4_horizontal_up_10_mmext;
147.
148.         if (chroma_format_idc <= 1)
149.             h->pred8x8[DC_PRED8x8] = ff_pred8x8_dc_10_mmext;
150.
151.         h->pred8x8l[DC_128_PRED] = ff_pred8x8l_128_dc_10_mmext;
152.
153.         h->pred16x16[DC_PRED8x8] = ff_pred16x16_dc_10_mmext;
154.         h->pred16x16[TOP_DC_PRED8x8] = ff_pred16x16_top_dc_10_mmext;
155.         h->pred16x16[DC_128_PRED8x8] = ff_pred16x16_128_dc_10_mmext;
156.         h->pred16x16[LEFT_DC_PRED8x8] = ff_pred16x16_left_dc_10_mmext;
157.         h->pred16x16[VERT_PRED8x8] = ff_pred16x16_vertical_10_mmext;
158.         h->pred16x16[HOR_PRED8x8] = ff_pred16x16_horizontal_10_mmext;
159.     }
160.     if (EXTERNAL_SSE2(cpu_flags)) {
161.         h->pred4x4[DIAG_DOWN_LEFT_PRED] = ff_pred4x4_down_left_10_sse2;
162.         h->pred4x4[DIAG_DOWN_RIGHT_PRED] = ff_pred4x4_down_right_10_sse2;
163.         h->pred4x4[VERT_LEFT_PRED] = ff_pred4x4_vertical_left_10_sse2;
164.         h->pred4x4[VERT_RIGHT_PRED] = ff_pred4x4_vertical_right_10_sse2;
165.         h->pred4x4[HOR_DOWN_PRED] = ff_pred4x4_horizontal_down_10_sse2;
166.
167.         if (chroma_format_idc <= 1) {
168.             h->pred8x8[DC_PRED8x8] = ff_pred8x8_dc_10_sse2;
169.             h->pred8x8[TOP_DC_PRED8x8] = ff_pred8x8_top_dc_10_sse2;
170.             h->pred8x8[PLANE_PRED8x8] = ff_pred8x8_plane_10_sse2;
171.             h->pred8x8[VERT_PRED8x8] = ff_pred8x8_vertical_10_sse2;
172.             h->pred8x8[HOR_PRED8x8] = ff_pred8x8_horizontal_10_sse2;
173.         }
174.
175.         h->pred8x8l[VERT_PRED] = ff_pred8x8l_vertical_10_sse2;
176.         h->pred8x8l[HOR_PRED] = ff_pred8x8l_horizontal_10_sse2;
177.         h->pred8x8l[DC_PRED] = ff_pred8x8l_dc_10_sse2;
178.         h->pred8x8l[DC_128_PRED] = ff_pred8x8l_128_dc_10_sse2;
179.         h->pred8x8l[TOP_DC_PRED] = ff_pred8x8l_top_dc_10_sse2;

```

```

179.         h->pred8x8l[TOP_DC_PRED] = ff_pred8x8l_top_dc_10_sse2;
180.         h->pred8x8l[DIAG_DOWN_LEFT_PRED] = ff_pred8x8l_down_left_10_sse2;
181.         h->pred8x8l[DIAG_DOWN_RIGHT_PRED] = ff_pred8x8l_down_right_10_sse2;
182.         h->pred8x8l[VERT_RIGHT_PRED] = ff_pred8x8l_vertical_right_10_sse2;
183.         h->pred8x8l[HOR_UP_PRED] = ff_pred8x8l_horizontal_up_10_sse2;
184.
185.         h->pred16x16[DC_PRED8x8] = ff_pred16x16_dc_10_sse2;
186.         h->pred16x16[TOP_DC_PRED8x8] = ff_pred16x16_top_dc_10_sse2;
187.         h->pred16x16[DC_128_PRED8x8] = ff_pred16x16_128_dc_10_sse2;
188.         h->pred16x16[LEFT_DC_PRED8x8] = ff_pred16x16_left_dc_10_sse2;
189.         h->pred16x16[VERT_PRED8x8] = ff_pred16x16_vertical_10_sse2;
190.         h->pred16x16[HOR_PRED8x8] = ff_pred16x16_horizontal_10_sse2;
191.     }
192.     if (EXTERNAL_SSSE3(cpu_flags)) {
193.         h->pred4x4[DIAG_DOWN_RIGHT_PRED] = ff_pred4x4_down_right_10_ssse3;
194.         h->pred4x4[VERT_RIGHT_PRED] = ff_pred4x4_vertical_right_10_ssse3;
195.         h->pred4x4[HOR_DOWN_PRED] = ff_pred4x4_horizontal_down_10_ssse3;
196.
197.         h->pred8x8l[HOR_PRED] = ff_pred8x8l_horizontal_10_ssse3;
198.         h->pred8x8l[DIAG_DOWN_LEFT_PRED] = ff_pred8x8l_down_left_10_ssse3;
199.         h->pred8x8l[DIAG_DOWN_RIGHT_PRED] = ff_pred8x8l_down_right_10_ssse3;
200.         h->pred8x8l[VERT_RIGHT_PRED] = ff_pred8x8l_vertical_right_10_ssse3;
201.         h->pred8x8l[HOR_UP_PRED] = ff_pred8x8l_horizontal_up_10_ssse3;
202.     }
203.     if (EXTERNAL_AVX(cpu_flags)) {
204.         h->pred4x4[DIAG_DOWN_LEFT_PRED] = ff_pred4x4_down_left_10_avx;
205.         h->pred4x4[DIAG_DOWN_RIGHT_PRED] = ff_pred4x4_down_right_10_avx;
206.         h->pred4x4[VERT_LEFT_PRED] = ff_pred4x4_vertical_left_10_avx;
207.         h->pred4x4[VERT_RIGHT_PRED] = ff_pred4x4_vertical_right_10_avx;
208.         h->pred4x4[HOR_DOWN_PRED] = ff_pred4x4_horizontal_down_10_avx;
209.
210.         h->pred8x8l[VERT_PRED] = ff_pred8x8l_vertical_10_avx;
211.         h->pred8x8l[HOR_PRED] = ff_pred8x8l_horizontal_10_avx;
212.         h->pred8x8l[DC_PRED] = ff_pred8x8l_dc_10_avx;
213.         h->pred8x8l[TOP_DC_PRED] = ff_pred8x8l_top_dc_10_avx;
214.         h->pred8x8l[DIAG_DOWN_RIGHT_PRED] = ff_pred8x8l_down_right_10_avx;
215.         h->pred8x8l[DIAG_DOWN_LEFT_PRED] = ff_pred8x8l_down_left_10_avx;
216.         h->pred8x8l[VERT_RIGHT_PRED] = ff_pred8x8l_vertical_right_10_avx;
217.         h->pred8x8l[HOR_UP_PRED] = ff_pred8x8l_horizontal_up_10_avx;
218.     }
219. }
220. }

```

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。



```

1. ;-----
2. ; void ff_pred16x16_vertical_8(uint8_t *src, int stride)
3. ; 注释：雷霄骅
4. ; 16x16帧内预测-Vertical
5. ;
6. ; Vertical预测方式
7. ; |X1 X2 X3 X4
8. ; -----
9. ; |X1 X2 X3 X4
10. ; |X1 X2 X3 X4
11. ; |X1 X2 X3 X4
12. ; |X1 X2 X3 X4
13. ;
14. ;-----
15. ;mmx指令优化
16. INIT_MMX mmx
17. cglobal pred16x16_vertical_8, 2,3
18. ;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）.....
19.     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）
20. ;此时r0指向16x16块上面一行像素数据
21.     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
22.     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。
23. ;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）
24. ;注：MOV-1~2字节（word），MOVD-4字节（Dword），MOVQ-8字节（Qword）
25.     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
26. ;即将宏块上面1行像素中后8个像素传入mm1
27. .loop:                          ;循环
28.     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。
29.     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。
30.     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。
31.     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。
32. ;总而言之，一次处理2行，16行像素一共处理8次
33.     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
34.     dec     r2                  ;r2--;
35.     jg     .loop                ;r2=0时候，不再跳转
36.     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。

```

1. ;sse指令优化
2. INIT_XMM sse
3. cglobal pred16x16_vertical_8, 2,3
4.     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
5.     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
6.     movaps  xmm0, [r0]           ;类似于memcpy(&xmm0,r0,16)。movaps传递128bit（16字节，对应16像素）数据
7. .loop:                          ;循环
8.     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。
9.     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。
10.     lea     r0, [r0+r1*2]        ;r0=r0+r1*2。r0前移2行。
11.     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。
12.     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。
13.     lea     r0, [r0+r1*2]        ;r0=r0+r1*2。r0再次前移2行。
14. ;注：一次循环处理了4行像素。16行像素一共处理4次
15.     dec     r2                  ;r2--;
16.     jg     .loop                ;r2=0时候，不再跳转
17.     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。

```

1. //亮度的IDCT
2. static av_always_inline void hl_decode_mb_idct_luma(H264Context *h, int mb_type,
3.                                     int is_h264, int simple,
4.                                     int transform_bypass,
5.                                     int pixel_shift,
6.                                     int *block_offset,
7.                                     int linesize,
8.                                     uint8_t *dest_y, int p)
9. {
10.     //用于IDCT
11.     void (*idct_add)(uint8_t *dst, int16_t *block, int stride);
12.     int i;
13.     block_offset += 16 * p;
14.     //Intra4x4的DCT反变换在pred部分已经完成, 这里就不需要处理了
15.     if (!IS_INTRA4x4(mb_type)) {
16.         if (is_h264) {
17.             //Intra16x16宏块
18.             if (IS_INTRA16x16(mb_type)) {
19.                 //transform_bypass=0, 不考虑
20.                 if (transform_bypass) {
21.                     if (h->sps.profile_idc == 244 &&
22.                         (h->intra16x16_pred_mode == VERT_PRED8x8 ||
23.                          h->intra16x16_pred_mode == HOR_PRED8x8)) {
24.                         h->hpc.pred16x16_add[h->intra16x16_pred_mode](dest_y, block_offset,
25.                                                                       h->mb + (p * 256 << pixel_shift),
26.                                                                       linesize);
27.                     } else {
28.                         for (i = 0; i < 16; i++)
29.                             if (h->non_zero_count_cache[scan8[i + p * 16]] ||
30.                                 dctcoef_get(h->mb, pixel_shift, i * 16 + p * 256))
31.                                 h->h264dsp.h264_add_pixels4_clear(dest_y + block_offset[i],
32.                                                                    h->mb + (i * 16 + p * 256 << pixel_shift),
33.                                                                    linesize);
34.                     }
35.                 } else {
36.                     //Intra16x16的DCT反变换
37.                     //最后的“16”代表内部循环处理16次
38.                     h->h264dsp.h264_idct_add16intra(dest_y, block_offset,
39.                                                       h->mb + (p * 256 << pixel_shift),
40.                                                       linesize,
41.                                                       h->non_zero_count_cache + p * 5 * 8);
42.                 }
43.             } else if (h->cbp & 15) { //15=1111, 即残差全部传送了
44.                 //Inter类型的宏块
45.
46.
47.                 //transform_bypass=0, 不考虑
48.                 if (transform_bypass) {
49.                     const int di = IS_8x8DCT(mb_type) ? 4 : 1;
50.                     idct_add = IS_8x8DCT(mb_type) ? h->h264dsp.h264_add_pixels8_clear
51.                                                       : h->h264dsp.h264_add_pixels4_clear;
52.                     for (i = 0; i < 16; i += di)
53.                         if (h->non_zero_count_cache[scan8[i + p * 16]])
54.                             idct_add(dest_y + block_offset[i],
55.                                       h->mb + (i * 16 + p * 256 << pixel_shift),
56.                                       linesize);
57.                 } else {
58.                     //8x8的IDCT
59.                     if (IS_8x8DCT(mb_type))
60.                         h->h264dsp.h264_idct8_add4(dest_y, block_offset,
61.                                                       h->mb + (p * 256 << pixel_shift),
62.                                                       linesize,
63.                                                       h->non_zero_count_cache + p * 5 * 8);
64.
65.                     //处理16x16宏块
66.                     //采用4x4的IDCT
67.                     //最后的“16”代表内部循环处理16次
68.                     //输出结果到dest_y
69.                     //h->mb中存储了DCT系数
70.                     else
71.                         h->h264dsp.h264_idct_add16(dest_y, block_offset,
72.                                                       h->mb + (p * 256 << pixel_shift),
73.                                                       linesize,
74.                                                       h->non_zero_count_cache + p * 5 * 8);
75.                 }
76.             }
77.         } else if (CONFIG_SVQ3_DECODER) {
78.             for (i = 0; i < 16; i++)
79.                 if (h->non_zero_count_cache[scan8[i + p * 16]] || h->mb[i * 16 + p * 256]) {
80.                     //FIXME benchmark weird rule, & below
81.                     uint8_t *const ptr = dest_y + block_offset[i];
82.                     ff_svq3_add_idct_c(ptr, h->mb + i * 16 + p * 256, linesize,
83.                                         h->qscale, IS_INTRA(mb_type) ? 1 : 0);
84.                 }
85.         }
86.     }

```

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

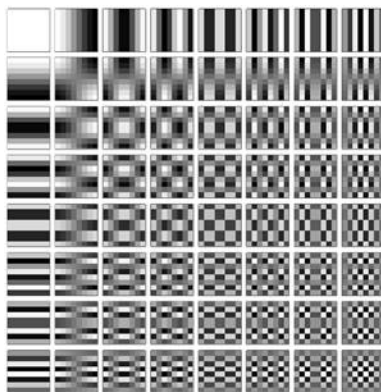
b) Inter类型：调用H264DSPContext的汇编函数h264\_idct\_add16()进行DCT反变换

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。

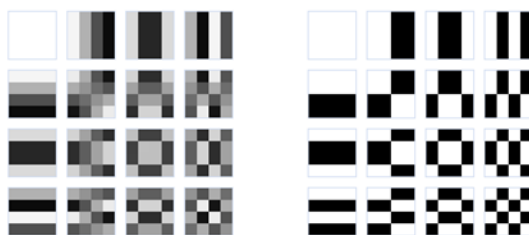
下文记录DCT反变化函数相关的源代码。

## 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）。



http://blog.csdn.net/visual\_2010  
DCT Walsh-Hadamard

4x4整数DCT变换的公式如下所示。

$$Y = (C_f X C_f^T)$$

$$= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} X \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{bmatrix}$$

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。

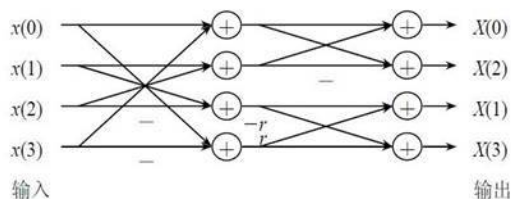


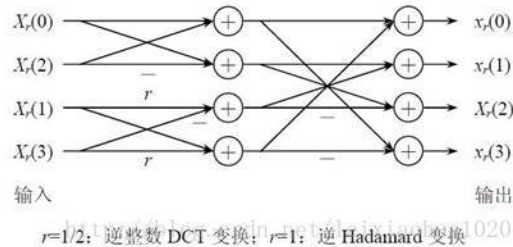
图2-2 整数 DCT 变换；图2-3 Hadamard 变换

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。

$$X_r = C_i^T(W)C_i$$

$$= \begin{bmatrix} 1 & 1 & 1 & \frac{1}{2} \\ 1 & \frac{1}{2} & -1 & -1 \\ 1 & -\frac{1}{2} & -1 & 1 \\ 1 & -1 & 1 & -\frac{1}{2} \end{bmatrix} W \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \frac{1}{2} & -\frac{1}{2} & -1 \\ 1 & -1 & -1 & 1 \\ \frac{1}{2} & -1 & 1 & -\frac{1}{2} \end{bmatrix}$$

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。



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。

```

1. //初始化DSP相关的函数。包含了IDCT、环路滤波函数等
2. av_cold void ff_h264dsp_init(H264DSPContext *c, const int bit_depth,
3.                               const int chroma_format_idc)
4. {
5.     #undef FUNC
6.     #define FUNC(a, depth) a ## _ ## depth ## _c
7.
8.     #define ADDPX_DSP(depth) \
9.         c->h264_add_pixels4_clear = FUNC(ff_h264_add_pixels4, depth);\
10.        c->h264_add_pixels8_clear = FUNC(ff_h264_add_pixels8, depth)
11.
12.     if (bit_depth > 8 && bit_depth <= 16) {
13.         ADDPX_DSP(16);
14.     } else {
15.         ADDPX_DSP(8);
16.     }
17.
18.     #define H264_DSP(depth) \
19.         c->h264_idct_add= FUNC(ff_h264_idct_add, depth);\
20.         c->h264_idct8_add= FUNC(ff_h264_idct8_add, depth);\
21.         c->h264_idct_dc_add= FUNC(ff_h264_idct_dc_add, depth);\
22.         c->h264_idct8_dc_add= FUNC(ff_h264_idct8_dc_add, depth);\
23.         c->h264_idct_add16 = FUNC(ff_h264_idct_add16, depth);\
24.         c->h264_idct8_add4 = FUNC(ff_h264_idct8_add4, depth);\
25.         if (chroma_format_idc <= 1)\
26.             c->h264_idct_add8 = FUNC(ff_h264_idct_add8, depth);\
27.         else\
28.             c->h264_idct_add8 = FUNC(ff_h264_idct_add8_422, depth);\
29.         c->h264_idct_add16intra= FUNC(ff_h264_idct_add16intra, depth);\
30.         c->h264_luma_dc_dequant_idct= FUNC(ff_h264_luma_dc_dequant_idct, depth);\
31.         if (chroma_format_idc <= 1)\
32.             c->h264_chroma_dc_dequant_idct= FUNC(ff_h264_chroma_dc_dequant_idct, depth);\
33.         else\
34.             c->h264_chroma_dc_dequant_idct= FUNC(ff_h264_chroma422_dc_dequant_idct, depth);\
35.     \
36.     c->weight_h264_pixels_tab[0]= FUNC(weight_h264_pixels16, depth);\
37.     c->weight_h264_pixels_tab[1]= FUNC(weight_h264_pixels8, depth);\
38.     c->weight_h264_pixels_tab[2]= FUNC(weight_h264_pixels4, depth);\
39.     c->weight_h264_pixels_tab[3]= FUNC(weight_h264_pixels2, depth);\
40.     c->biweight_h264_pixels_tab[0]= FUNC(biweight_h264_pixels16, depth);\
41.     c->biweight_h264_pixels_tab[1]= FUNC(biweight_h264_pixels8, depth);\
42.     c->biweight_h264_pixels_tab[2]= FUNC(biweight_h264_pixels4, depth);\
43.     c->biweight_h264_pixels_tab[3]= FUNC(biweight_h264_pixels2, depth);\
44.     \
45.     c->h264_v_loop_filter_luma= FUNC(h264_v_loop_filter_luma, depth);\
46.     c->h264_h_loop_filter_luma= FUNC(h264_h_loop_filter_luma, depth);\
47.     c->h264_h_loop_filter_luma_mbaff= FUNC(h264_h_loop_filter_luma_mbaff, depth);\

```

```

48. c->h264_v_loop_filter_luma_intra= FUNC(h264_v_loop_filter_luma_intra, depth);\
49. c->h264_h_loop_filter_luma_intra= FUNC(h264_h_loop_filter_luma_intra, depth);\
50. c->h264_h_loop_filter_luma_mbaff_intra= FUNC(h264_h_loop_filter_luma_mbaff_intra, depth);\
51. c->h264_v_loop_filter_chroma= FUNC(h264_v_loop_filter_chroma, depth);\
52. if (chroma_format_idc <= 1)\
53.     c->h264_h_loop_filter_chroma= FUNC(h264_h_loop_filter_chroma, depth);\
54. else\
55.     c->h264_h_loop_filter_chroma= FUNC(h264_h_loop_filter_chroma422, depth);\
56. if (chroma_format_idc <= 1)\
57.     c->h264_h_loop_filter_chroma_mbaff= FUNC(h264_h_loop_filter_chroma_mbaff, depth);\
58. else\
59.     c->h264_h_loop_filter_chroma_mbaff= FUNC(h264_h_loop_filter_chroma422_mbaff, depth);\
60. c->h264_v_loop_filter_chroma_intra= FUNC(h264_v_loop_filter_chroma_intra, depth);\
61. if (chroma_format_idc <= 1)\
62.     c->h264_h_loop_filter_chroma_intra= FUNC(h264_h_loop_filter_chroma_intra, depth);\
63. else\
64.     c->h264_h_loop_filter_chroma_intra= FUNC(h264_h_loop_filter_chroma422_intra, depth);\
65. if (chroma_format_idc <= 1)\
66.     c->h264_h_loop_filter_chroma_mbaff_intra= FUNC(h264_h_loop_filter_chroma_mbaff_intra, depth);\
67. else\
68.     c->h264_h_loop_filter_chroma_mbaff_intra= FUNC(h264_h_loop_filter_chroma422_mbaff_intra, depth);\
69. c->h264_loop_filter_strength= NULL;
70. //根据颜色位深，初始化不同的函数
71. //一般为8bit，即执行H264_DSP(8)
72. switch (bit_depth) {
73. case 9:
74.     H264_DSP(9);
75.     break;
76. case 10:
77.     H264_DSP(10);
78.     break;
79. case 12:
80.     H264_DSP(12);
81.     break;
82. case 14:
83.     H264_DSP(14);
84.     break;
85. default:
86.     av_assert0(bit_depth<=8);
87.     H264_DSP(8);
88.     break;
89. }
90. //这个函数查找startcode的时候用到
91. //在这里竟然单独列出
92. c->startcode_find_candidate = ff_startcode_find_candidate_c;
93. //如果系统支持，则初始化经过汇编优化的函数
94. if (ARCH_AARCH64) ff_h264dsp_init_aarch64(c, bit_depth, chroma_format_idc);
95. if (ARCH_ARM) ff_h264dsp_init_arm(c, bit_depth, chroma_format_idc);
96. if (ARCH_PPC) ff_h264dsp_init_ppc(c, bit_depth, chroma_format_idc);
97. if (ARCH_X86) ff_h264dsp_init_x86(c, bit_depth, chroma_format_idc);
98. }

```

从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。

```

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

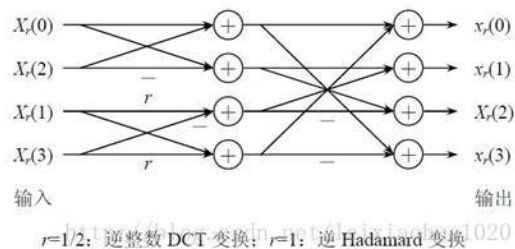
```

1.  /* IDCT
2.  *
3.  * 注释：雷霄骅
4.  * leixiaohua1020@126.com
5.  * http://blog.csdn.net/leixiaohua1020
6.  *
7.  * 参数：
8.  * _block：输入DCT系数
9.  * _dst：输出像素数据
10. * stride：一行图像数据的大小
11. *
12. */
13. //IDCT反变换（4x4）
14. //“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
15. void FUNC(ff_h264_idct_add)(uint8_t *_dst, int16_t *_block, int stride)
16. {
17.
18.     /*
19.     *      | 1  1  1  1 |   | 1  2  1  1 |
20.     *      | 2  1 -1 -2 |   | 1  1 -1 -2 |
21.     * Y = | 1 -1 -1 -2 | X | 1 -1 -1  2 |
22.     *      | 1 -2  2 -1 |   | 1 -2  1 -1 |
23.     *
24.     */
25.     int i;
26.     pixel *_dst = (pixel*)_dst;
27.     dctcoef *_block = (dctcoef*)_block;
28.     stride >>= sizeof(pixel)-1;
29.
30.     block[0] += 1 << 5;
31.     //蝶形算法（一维变换，纵向）
32.     //---+-----
33.     // 0 |   |   |
34.     // 4 |
35.     // 8 |
36.     // 12|
37.     //---+-----
38.     for(i=0; i<4; i++){
39.         //[0]和[2]
40.         const int z0=  block[i + 4*0]    +  block[i + 4*2];
41.         const int z1=  block[i + 4*0]    -  block[i + 4*2];
42.         //[1]和[3]
43.         const int z2= (block[i + 4*1]>>1) -  block[i + 4*3];
44.         const int z3=  block[i + 4*1]    + (block[i + 4*3]>>1);
45.
46.         block[i + 4*0]= z0 + z3;
47.         block[i + 4*1]= z1 + z2;
48.         block[i + 4*2]= z1 - z2;
49.         block[i + 4*3]= z0 - z3;
50.     }
51.     //蝶形算法（另一维变换，横向）
52.     //---+-----
53.     // 0 | 1 | 2 | 3 |
54.     //   |
55.     //   |
56.     //   |
57.     //---+-----
58.     for(i=0; i<4; i++){
59.         const int z0=  block[0 + 4*i]    +  block[2 + 4*i];
60.         const int z1=  block[0 + 4*i]    -  block[2 + 4*i];
61.         const int z2= (block[1 + 4*i]>>1) -  block[3 + 4*i];
62.         const int z3=  block[1 + 4*i]    + (block[3 + 4*i]>>1);
63.         //av_clip_pixel(): 把一个整形转换取值范围为0-255的数值，用于限幅
64.         //注意是累加而不是赋值到dst上（所以函数名中包含“add”）
65.
66.         //转置？！
67.         //一列一列处理
68.         dst[i + 0*stride]= av_clip_pixel(dst[i + 0*stride] + ((z0 + z3) >> 6));
69.         dst[i + 1*stride]= av_clip_pixel(dst[i + 1*stride] + ((z1 + z2) >> 6));
70.         dst[i + 2*stride]= av_clip_pixel(dst[i + 2*stride] + ((z1 - z2) >> 6));
71.         dst[i + 3*stride]= av_clip_pixel(dst[i + 3*stride] + ((z0 - z3) >> 6));
72.     }
73.     //清零
74.     memset(block, 0, 16 * sizeof(dctcoef));
75. }

```

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。



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]
1. // assumes all AC coefs are 0
2. //DCT反变换，特殊情况：
3. //AC系数全部为0（没有传递AC系数，只有DC系数）
4. void FUNC(ff_h264_idct_dc_add)(uint8_t *_dst, int16_t *_block, int stride){
5.     int i, j;
6.     pixel *_dst = (pixel*)_dst;
7.     dctcoef *_block = (dctcoef*)_block;
8.     //DC系数
9.     int dc = (block[0] + 32) >> 6;
10.    stride /= sizeof(pixel);
11.    //设置为0
12.    block[0] = 0;
13.    //在4x4块的每个像素上面累加（注意不是赋值）dc系数
14.    for( j = 0; j < 4; j++ )
15.    {
16.        for( i = 0; i < 4; i++ )
17.            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
18.        dst += stride;//下一行
19.    }
20. }
```

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]
1. //处理16x16宏块
2. //采用4x4的IDCT
3. //最后的“16”代表内部循环处理16次
4. //输入为block，输出为dst
5. void FUNC(ff_h264_idct_add16)(uint8_t *_dst, const int *_block_offset, int16_t *_block, int stride, const uint8_t nnzc[15*8]){
6.     int i;
7.     //循环16次
8.     for(i=0; i<16; i++){
9.         //非零系数个数
10.        int nnz = nnzc[ scan8[i] ];
11.        //非零系数个数不为0才处理
12.        if(nnz){
13.            //特殊：只有DC系数
14.            if(nnz==1 && ((dctcoef*)_block)[i*16]) FUNC(ff_h264_idct_dc_add)(dst + block_offset[i], block + i*16*sizeof(pixel), stride);
15.            //一般的情况
16.            else FUNC(ff_h264_idct_add_8_c)(dst + block_offset[i], block + i*16*sizeof(pixel), stride);
17.        }
18.    }
19. }
```

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。



```

1. //处理Intra16x16宏块
2. //采用4x4的IDCT
3. //最后的“16”代表内部循环处理16次
4. //输入为block, 输出为dst
5. void FUNC(ff_h264_idct_add16intra)(uint8_t *dst, const int *block_offset, int16_t *block, int stride, const uint8_t nnzc[15*8]){
6.     int i;
7.     for(i=0; i<16; i++){
8.         if(nnzc[ scan8[i] ])          FUNC(ff_h264_idct_add  )(dst + block_offset[i], block + i*16*sizeof(pixel), stride);
9.         else if(((dctcoef*)block)[i*16]) FUNC(ff_h264_idct_dc_add)(dst + block_offset[i], block + i*16*sizeof(pixel), stride);
10.    }
11. }

```

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>

版权声明：本文为博主原创文章，未经博主允许不得转载。 <https://blog.csdn.net/leixiaohua1020/article/details/45143075>

文章标签： [FFmpeg](#) [宏块](#) [解码](#) [帧内预测](#) [DCT](#)

个人分类： [FFMPEG](#)

所属专栏： [FFmpeg](#)

此PDF由spygg生成,请尊重原作者版权!!!

我的邮箱:liushidc@163.com