

FFmpeg的HEVC解码器源代码简单分析：CTU解码（CTU Decode）部分-TU

2015年06月16日 17:14:42 阅读数：6833

HEVC源代码分析文章列表：

【解码 -libavcodec HEVC 解码器】

[FFmpeg的HEVC解码器源代码简单分析：概述](#)

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

[FFmpeg的HEVC解码器源代码简单分析：解码器主干部分](#)

[FFmpeg的HEVC解码器源代码简单分析：CTU解码（CTU Decode）部分-PU](#)

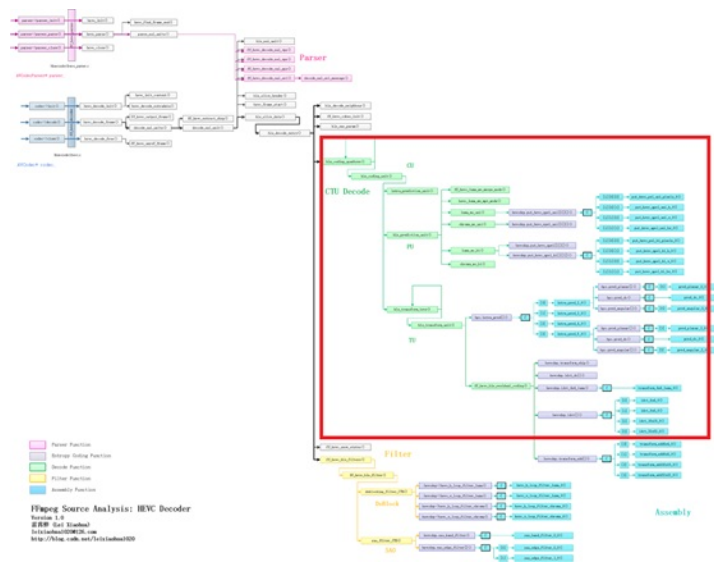
[FFmpeg的HEVC解码器源代码简单分析：CTU解码（CTU Decode）部分-TU](#)

[FFmpeg的HEVC解码器源代码简单分析：环路滤波（LoopFilter）](#)

本文分析FFmpeg的libavcodec中的HEVC解码器的CTU解码（CTU Decode）部分的源代码。FFmpeg的HEVC解码器调用hls_decode_entry()函数完成了Slice解码工作。hls_decode_entry()则调用了hls_coding_quadtree()完成了CTU解码工作。由于CTU解码部分的内容比较多，因此将这一部分内容拆分成两篇文章：一篇文章记录PU的解码，另一篇文章记录TU解码。本文记录TU的解码过程。

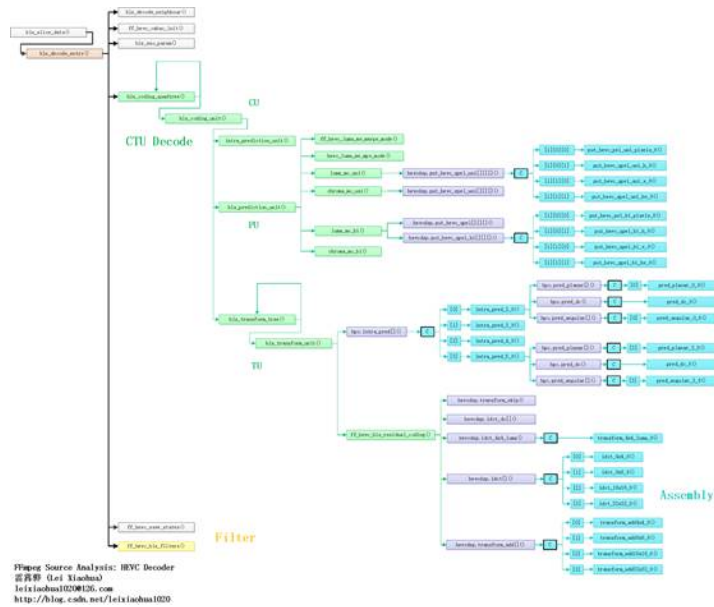
函数调用关系图

FFmpeg HEVC解码器的CTU解码（CTU Decoder）部分在整个HEVC解码器中的位置如下图所示。



[单击查看更清晰的大图](#)

CTU解码（CTU Decoder）部分的函数调用关系如下图所示。



单击查看更清晰的大图

从图中可以看出，CTU解码模块对应的函数是hls_coding_quadtree()。该函数是一个递归调用的函数，可以按照四叉树的句法格式解析CTU并获得其中的CU。对于每个CU会调用hls_coding_unit()进行解码。

hls_coding_unit()会调用hls_prediction_unit()对CU中的PU进行处理。hls_prediction_unit()调用luma_mc_unit()对亮度单向预测块进行运动补偿处理，调用chroma_mc_unit()对色度单向预测块进行运动补偿处理，调用luma_mc_bi()对亮度单向预测块进行运动补偿处理。

hls_coding_unit()会调用hls_transform_tree()对CU中的TU进行处理。hls_transform_tree()是一个递归调用的函数，可以按照四叉树的句法格式解析并获得其中的TU。对于每一个TU会调用hls_transform_unit()进行解码。hls_transform_unit()会进行帧内预测，并且调用ff_hevc_hls_residual_coding()解码DCT残差数据。

hls_decode_entry()

hls_decode_entry()是FFmpeg HEVC解码器中Slice解码的入口函数。该函数的定义如下所示。

```
[cpp]
1. //解码入口函数
2. static int hls_decode_entry(AVCodecContext *avctx, void *isFilterThread)
3. {
4.     HEVCContext *s = avctx->priv_data;
5.     //CTB尺寸
6.     int ctb_size = 1 << s->sps->log2_ctb_size;
7.     int more_data = 1;
8.     int x_ctb = 0;
9.     int y_ctb = 0;
10.    int ctb_addr_ts = s->pps->ctb_addr_rs_to_ts[s->sh.slice_ctb_addr_rs];
11.
12.    if (!ctb_addr_ts && s->sh.dependent_slice_segment_flag) {
13.        av_log(s->avctx, AV_LOG_ERROR, "Impossible initial tile.\n");
14.        return AVERROR_INVALIDDATA;
15.    }
16.
17.    if (s->sh.dependent_slice_segment_flag) {
18.        int prev_rs = s->pps->ctb_addr_rs_to_rs[ctb_addr_ts - 1];
19.        if (s->tab_slice_address[prev_rs] != s->sh.slice_addr) {
20.            av_log(s->avctx, AV_LOG_ERROR, "Previous slice segment missing\n");
21.            return AVERROR_INVALIDDATA;
22.        }
23.    }
24.
25.    while (more_data && ctb_addr_ts < s->sps->ctb_size) {
26.        int ctb_addr_rs = s->pps->ctb_addr_ts_to_rs[ctb_addr_ts];
27.        //CTB的位置x和y
28.        x_ctb = (ctb_addr_rs % ((s->sps->width + ctb_size - 1) >> s->sps->log2_ctb_size)) << s->sps->log2_ctb_size;
29.        y_ctb = (ctb_addr_rs / ((s->sps->width + ctb_size - 1) >> s->sps->log2_ctb_size)) << s->sps->log2_ctb_size;
30.        //初始化周围的参数
31.        hls_decode_neighbour(s, x_ctb, y_ctb, ctb_addr_ts);
32.        //初始化CABAC
33.        ff_hevc_cabac_init(s, ctb_addr_ts);
34.        //样点自适应补偿参数
35.        hls_sao_param(s, x_ctb >> s->sps->log2_ctb_size, y_ctb >> s->sps->log2_ctb_size);
36.
37.        s->deblock[ctb_addr_rs].beta_offset = s->sh.beta_offset;
38.        s->deblock[ctb_addr_rs].tc_offset = s->sh.tc_offset;
39.        s->filter_slice_edges[ctb_addr_rs] = s->sh.slice_loop_filter_across_slices_enabled_flag;
40.        /*
41.         * CU示意图
42.         *
```

```

43.      * 64x64块
44.      *
45.      * 深度d=0
46.      * split_flag=1时候划分为4个32x32
47.      *
48.      * +-----+-----+-----+-----+-----+-----+-----+
49.      * |                                                     |
50.      * |                                                     |
51.      * |                                                     |
52.      * | +-----+-----+-----+-----+-----+-----+
53.      * | |                                                     |
54.      * | |                                                     |
55.      * | |                                                     |
56.      * | | +-----+-----+-----+-----+-----+-----+
57.      * | | |                                                     |
58.      * | | |                                                     |
59.      * | | |                                                     |
60.      * | | | +-----+-----+-----+-----+-----+-----+
61.      * | | | |                                                     |
62.      * | | | |                                                     |
63.      * | | | |                                                     |
64.      * | | | | +-----+-----+-----+-----+-----+-----+
65.      * | | | | |                                                     |
66.      * | | | | |                                                     |
67.      * | | | | |                                                     |
68.      * | | | | | +-----+-----+-----+-----+-----+-----+
69.      * | | | | | |                                                     |
70.      * | | | | | |                                                     |
71.      * | | | | | |                                                     |
72.      * | | | | | | +-----+-----+-----+-----+-----+-----+
73.      * | | | | | | |                                                     |
74.      * | | | | | | |                                                     |
75.      * | | | | | | |                                                     |
76.      * | | | | | | | +-----+-----+-----+-----+-----+-----+
77.      * | | | | | | | |                                                     |
78.      * | | | | | | | |                                                     |
79.      * | | | | | | | |                                                     |
80.      * | | | | | | | | +-----+-----+-----+-----+-----+-----+
81.      *
82.      *
83.      * 32x32 块
84.      * 深度d=1
85.      * split_flag=1时候划分为4个16x16
86.      *
87.      * +-----+-----+-----+
88.      * |               |       |
89.      * |               |       |
90.      * |               |       |
91.      * | +-----+-----+
92.      * | |               |       |
93.      * | |               |       |
94.      * | |               |       |
95.      * | | +-----+-----+
96.      * | |               |       |
97.      * | |               |       |
98.      * | |               |       |
99.      * | | +-----+-----+
100.     * |               |       |
101.     * |               |       |
102.     * |               |       |
103.     * | +-----+-----+
104.     *
105.     *
106.     * 16x16 块
107.     * 深度d=2
108.     * split_flag=1时候划分为4个8x8
109.     *
110.     * +-----+-----+
111.     * |               |
112.     * |               |
113.     * |               |
114.     * | +-----+-----+
115.     * |               |
116.     * |               |
117.     * |               |
118.     * | +-----+-----+
119.     *
120.     *
121.     * 8x8块
122.     * 深度d=3
123.     * split_flag=1时候划分为4个4x4
124.     *
125.     * +---+---+
126.     * |   |   |
127.     * | +---+---+
128.     * |   |   |
129.     * | +---+---+
130.     *
131.     */
132.     /*
133.     * 解析四叉树结构，并且解码
134.     */

```

```

134.     *
135.     * hls_coding_quadtree(HEVCContext *s, int x0, int y0, int log2_cb_size, int cb_depth)中:
136.     * s: HEVCContext上下文结构体
137.     * x_ctb: CB位置的x坐标
138.     * y_ctb: CB位置的y坐标
139.     * log2_cb_size: CB大小取log2之后的值
140.     * cb_depth: 深度
141.     *
142.     */
143.     more_data = hls_coding_quadtree(s, x_ctb, y_ctb, s->sps->log2_ctb_size, 0);
144.     if (more_data < 0) {
145.         s->tab_slice_address[ctb_addr_rs] = -1;
146.         return more_data;
147.     }
148.
149.
150.     ctb_addr_ts++;
151.     //保存解码信息以供下次使用
152.     ff_hevc_save_states(s, ctb_addr_ts);
153.     //去块效应滤波
154.     ff_hevc_hls_filters(s, x_ctb, y_ctb, ctb_size);
155. }
156.
157. if (x_ctb + ctb_size >= s->sps->width &&
158.     y_ctb + ctb_size >= s->sps->height)
159.     ff_hevc_hls_filter(s, x_ctb, y_ctb, ctb_size);
160.
161. return ctb_addr_ts;
162. }

```

从源代码可以看出，hls_decode_entry()主要调用了2个函数进行解码工作：

- (1) 调用hls_coding_quadtree()解码CTU。其中包含了PU和TU的解码。
- (2) 调用ff_hevc_hls_filters()进行滤波。其中包含了去块效应滤波和SAO滤波。

本文分析第一步的CTU解码过程。

hls_coding_quadtree()

hls_coding_quadtree()用于解析CTU的四叉树句法结构。该函数的定义如下所示。

```

1.  /*
2.  * 解析四叉树结构，并且解码
3.  * 注意该函数是递归调用
4.  * 注释和处理：雷霄骅
5.  *
6.  *
7.  * s: HEVCContext上下文结构体
8.  * x_ctb: CB位置的x坐标
9.  * y_ctb: CB位置的y坐标
10. * log2_cb_size: CB大小取log2之后的值
11. * cb_depth: 深度
12. *
13. */
14. static int hls_coding_quadtree(HEVCContext *s, int x0, int y0,
15.                                int log2_cb_size, int cb_depth)
16. {
17.     HEVCLocalContext *lc = s->HEVCLc;
18.     //CB的大小,split flag=0
19.     //log2_cb_size为CB大小取log之后的结果
20.     const int cb_size = 1 << log2_cb_size;
21.     int ret;
22.     int qp_block_mask = (1<<(s->sps->log2_ctb_size - s->pps->diff_cu_qp_delta_depth)) - 1;
23.     int split_cu;
24.
25.     lc->ct_depth = cb_depth;
26.     if (x0 + cb_size <= s->sps->width &&
27.         y0 + cb_size <= s->sps->height &&
28.         log2_cb_size > s->sps->log2_min_cb_size) {
29.         split_cu = ff_hevc_split_coding_unit_flag_decode(s, cb_depth, x0, y0);
30.     } else {
31.         split_cu = (log2_cb_size > s->sps->log2_min_cb_size);
32.     }
33.     if (s->pps->cu_qp_delta_enabled_flag &&
34.         log2_cb_size >= s->sps->log2_ctb_size - s->pps->diff_cu_qp_delta_depth) {
35.         lc->tu.is_cu_qp_delta_coded = 0;
36.         lc->tu.cu_qp_delta = 0;
37.     }
38.
39.     if (s->sh.cu_chroma_qp_offset_enabled_flag &&
40.         log2_cb_size >= s->sps->log2_ctb_size - s->pps->diff_cu_chroma_qp_offset_depth) {
41.         lc->tu.is_cu_chroma_qp_offset_coded = 0;
42.     }
43.
44.     if (split_cu) {
45.         //如果CU还可以继续划分，则继续解析划分后的CU
46.         //注意这里是递归调用
47.

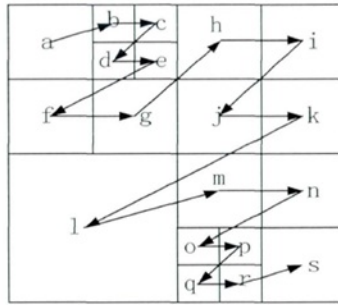
```

```

48.
49. //CB的大小,split flag=1
50. const int cb_size_split = cb_size >> 1;
51.
52. /*
53.  * (x0, y0) (x1, y0)
54.  * +-----+-----+
55.  * |           |           |
56.  * |           |           |
57.  * |           |           |
58.  * + -- --+ -- -- +
59.  * (x0, y1) (x1, y1) |
60.  * |           |           |
61.  * |           |           |
62.  * +-----+-----+
63.  *
64.  */
65. const int x1 = x0 + cb_size_split;
66. const int y1 = y0 + cb_size_split;
67.
68. int more_data = 0;
69.
70. //注意:
71. //CU大小减半, log2_cb_size-1
72. //深度d加1, cb_depth+1
73. more_data = hls_coding_quadtree(s, x0, y0, log2_cb_size - 1, cb_depth + 1);
74. if (more_data < 0)
75.     return more_data;
76.
77. if (more_data && x1 < s->sps->width) {
78.     more_data = hls_coding_quadtree(s, x1, y0, log2_cb_size - 1, cb_depth + 1);
79.     if (more_data < 0)
80.         return more_data;
81. }
82. if (more_data && y1 < s->sps->height) {
83.     more_data = hls_coding_quadtree(s, x0, y1, log2_cb_size - 1, cb_depth + 1);
84.     if (more_data < 0)
85.         return more_data;
86. }
87. if (more_data && x1 < s->sps->width &&
88.     y1 < s->sps->height) {
89.     more_data = hls_coding_quadtree(s, x1, y1, log2_cb_size - 1, cb_depth + 1);
90.     if (more_data < 0)
91.         return more_data;
92. }
93.
94. if(((x0 + (1<<log2_cb_size)) & qp_block_mask) == 0 &&
95.     ((y0 + (1<<log2_cb_size)) & qp_block_mask) == 0)
96.     lc->qPy_pred = lc->qp_y;
97.
98. if (more_data)
99.     return ((x1 + cb_size_split) < s->sps->width ||
100.        (y1 + cb_size_split) < s->sps->height);
101. else
102.     return 0;
103. } else {
104.
105.     /*
106.     * (x0, y0)
107.     * +-----+-----+
108.     * |           |           |
109.     * |           |           |
110.     * |           |           |
111.     * +           +
112.     * |           |           |
113.     * |           |           |
114.     * |           |           |
115.     * +-----+-----+
116.     *
117.     */
118.     //注意处理的是不可划分的CU单元
119.     //处理CU单元-真正的解码
120.     ret = hls_coding_unit(s, x0, y0, log2_cb_size);
121.     if (ret < 0)
122.         return ret;
123.     if (((!(x0 + cb_size) %
124.         (1 << (s->sps->log2_ctb_size))) ||
125.         (x0 + cb_size >= s->sps->width)) &&
126.         (!(y0 + cb_size) %
127.         (1 << (s->sps->log2_ctb_size))) ||
128.         (y0 + cb_size >= s->sps->height))) {
129.         int end_of_slice_flag = ff_hevc_end_of_slice_flag_decode(s);
130.         return !end_of_slice_flag;
131.     } else {
132.         return 1;
133.     }
134. }
135.
136. return 0;
137. }

```

从源代码可以看出，hls_coding_quadtree()首先调用ff_hevc_split_coding_unit_flag_decode()判断当前CU是否还需要划分。如果需要划分的话，就会递归调用4次hls_coding_quadtree()分别对4个子块继续进行二叉树解析；如果不需要划分，就会调用hls_coding_unit()对CU进行解码。总而言之，hls_coding_quadtree()会解析出来一个CTU中的所有CU，并且对每一个CU逐一调用hls_coding_unit()进行解码。一个CTU中CU的解码顺序如下图所示。图中a, b, c ...即代表了先后顺序。



hls_coding_unit()

hls_coding_unit()用于解码一个CU。该函数的定义如下所示。

```
[cpp]
1. //处理CU单元-真正的解码
2. //注释和处理：雷霄骅
3. static int hls_coding_unit(HEVCContext *s, int x0, int y0, int log2_cb_size)
4. {
5.     //CB大小
6.     int cb_size = 1 << log2_cb_size;
7.     HEVCLocalContext *lc = s->HEVCLc;
8.     int log2_min_cb_size = s->sps->log2_min_cb_size;
9.     int length = cb_size >> log2_min_cb_size;
10.    int min_cb_width = s->sps->min_cb_width;
11.    //以最小的CB为单位（例如4x4）的时候，当前CB的位置—x坐标和y坐标
12.    int x_cb = x0 >> log2_min_cb_size;
13.    int y_cb = y0 >> log2_min_cb_size;
14.    int idx = log2_cb_size - 2;
15.    int qp_block_mask = (1 << (s->sps->log2_ctb_size - s->pps->diff_cu_qp_delta_depth)) - 1;
16.    int x, y, ret;
17.
18.    //设置CU的属性值
19.    lc->cu.x = x0;
20.    lc->cu.y = y0;
21.    lc->cu.pred_mode = MODE_INTRA;
22.    lc->cu.part_mode = PART_2Nx2N;
23.    lc->cu.intra_split_flag = 0;
24.
25.    SAMPLE_CTB(s->skip_flag, x_cb, y_cb) = 0;
26.
27.    for (x = 0; x < 4; x++)
28.        lc->pu.intra_pred_mode[x] = 1;
29.    if (s->pps->transquant_bypass_enable_flag) {
30.        lc->cu.cu_transquant_bypass_flag = ff_hevc_cu_transquant_bypass_flag_decode(s);
31.        if (lc->cu.cu_transquant_bypass_flag)
32.            set_deblocking_bypass(s, x0, y0, log2_cb_size);
33.    } else
34.        lc->cu.cu_transquant_bypass_flag = 0;
35.
36.    if (s->sh.slice_type != I_SLICE) {
37.        //Skip类型
38.        uint8_t skip_flag = ff_hevc_skip_flag_decode(s, x0, y0, x_cb, y_cb);
39.        //设置到skip_flag缓存中
40.        x = y_cb * min_cb_width + x_cb;
41.        for (y = 0; y < length; y++) {
42.            memset(&s->skip_flag[x], skip_flag, length);
43.            x += min_cb_width;
44.        }
45.        lc->cu.pred_mode = skip_flag ? MODE_SKIP : MODE_INTER;
46.    } else {
47.        x = y_cb * min_cb_width + x_cb;
48.        for (y = 0; y < length; y++) {
49.            memset(&s->skip_flag[x], 0, length);
50.            x += min_cb_width;
51.        }
52.    }
53.
54.    if (SAMPLE_CTB(s->skip_flag, x_cb, y_cb)) {
55.        hls_prediction_unit(s, x0, y0, cb_size, cb_size, log2_cb_size, 0, idx);
56.        intra_prediction_unit_default_value(s, x0, y0, log2_cb_size);
57.
58.        if (!s->sh.disable_deblocking_filter_flag)
59.            ff_hevc_deblocking_boundary_strengths(s, x0, y0, log2_cb_size);
60.    } else {
```

```

61. int pcm_flag = 0;
62.
63. //读取预测模式 (非 I Slice)
64. if (s->sh.slice_type != I_SLICE)
65.     lc->cu.pred_mode = ff_hevc_pred_mode_decode(s);
66.
67. //不是帧内预测模式的时候
68. //或者已经是最小CB的时候
69. if (lc->cu.pred_mode != MODE_INTRA ||
70.     log2_cb_size == s->sps->log2_min_cb_size) {
71.     //读取CU分割模式
72.     lc->cu.part_mode = ff_hevc_part_mode_decode(s, log2_cb_size);
73.     lc->cu.intra_split_flag = lc->cu.part_mode == PART_NxN &&
74.                             lc->cu.pred_mode == MODE_INTRA;
75. }
76.
77. if (lc->cu.pred_mode == MODE_INTRA) {
78.     //帧内预测模式
79.
80.     //PCM方式编码，不常见
81.     if (lc->cu.part_mode == PART_2Nx2N && s->sps->pcm_enabled_flag &&
82.         log2_cb_size >= s->sps->pcm.log2_min_pcm_cb_size &&
83.         log2_cb_size <= s->sps->pcm.log2_max_pcm_cb_size) {
84.         pcm_flag = ff_hevc_pcm_flag_decode(s);
85.     }
86.     if (pcm_flag) {
87.         intra_prediction_unit_default_value(s, x0, y0, log2_cb_size);
88.         ret = hls_pcm_sample(s, x0, y0, log2_cb_size);
89.         if (s->sps->pcm.loop_filter_disable_flag)
90.             set_deblocking_bypass(s, x0, y0, log2_cb_size);
91.
92.         if (ret < 0)
93.             return ret;
94.     } else {
95.         //帧内预测
96.         intra_prediction_unit(s, x0, y0, log2_cb_size);
97.     }
98. } else {
99.     //帧间预测模式
100.    intra_prediction_unit_default_value(s, x0, y0, log2_cb_size);
101.
102.    //帧间模式一共有8种划分模式
103.
104.    switch (lc->cu.part_mode) {
105.    case PART_2Nx2N:
106.        /*
107.         * PART_2Nx2N:
108.         * +-----+-----+
109.         * |               |
110.         * |               |
111.         * |               |
112.         * |   +       +   |
113.         * |               |
114.         * |               |
115.         * |               |
116.         * +-----+-----+
117.         */
118.        //处理PU单元-运动补偿
119.        hls_prediction_unit(s, x0, y0, cb_size, cb_size, log2_cb_size, 0, idx);
120.        break;
121.    case PART_2NxN:
122.        /*
123.         * PART_2NxN:
124.         * +-----+-----+
125.         * |               |
126.         * |               |
127.         * |               |
128.         * +-----+-----+
129.         * |               |
130.         * |               |
131.         * |               |
132.         * +-----+-----+
133.         */
134.        /*
135.         * hls_prediction_unit()参数:
136.         * x0 : PU左上角x坐标
137.         * y0 : PU左上角y坐标
138.         * nPbW : PU宽度
139.         * nPbH : PU高度
140.         * log2_cb_size : CB大小取log2()的值
141.         * partIdx : PU的索引号-分成4个块的时候取0-3, 分成两个块的时候取0和1
142.         */
143.        //上
144.        hls_prediction_unit(s, x0, y0, cb_size, cb_size / 2, log2_cb_size, 0, idx);
145.        //下
146.        hls_prediction_unit(s, x0, y0 + cb_size / 2, cb_size, cb_size / 2, log2_cb_size, 1, idx);
147.        break;
148.    case PART_Nx2N:
149.        /*
150.         * PART_Nx2N:
151.         * +-----+-----+
152.         * |               |
153.         * |               |
154.         * |               |
155.         * |               |
156.         * |               |
157.         * |               |
158.         * |               |
159.         * +-----+-----+
160.         */
161.        /*
162.         * hls_prediction_unit()参数:
163.         * x0 : PU左上角x坐标
164.         * y0 : PU左上角y坐标
165.         * nPbW : PU宽度
166.         * nPbH : PU高度
167.         * log2_cb_size : CB大小取log2()的值
168.         * partIdx : PU的索引号-分成4个块的时候取0-3, 分成两个块的时候取0和1
169.         */
170.        //左
171.        hls_prediction_unit(s, x0, y0, cb_size / 2, cb_size, log2_cb_size, 0, idx);
172.        //右
173.        hls_prediction_unit(s, x0 + cb_size / 2, y0, cb_size / 2, cb_size, log2_cb_size, 1, idx);
174.        break;
175.    case PART_NxN:
176.        /*
177.         * PART_NxN:
178.         * +-----+-----+
179.         * |               |
180.         * |               |
181.         * |               |
182.         * |               |
183.         * |               |
184.         * |               |
185.         * |               |
186.         * +-----+-----+
187.         */
188.        /*
189.         * hls_prediction_unit()参数:
190.         * x0 : PU左上角x坐标
191.         * y0 : PU左上角y坐标
192.         * nPbW : PU宽度
193.         * nPbH : PU高度
194.         * log2_cb_size : CB大小取log2()的值
195.         * partIdx : PU的索引号-分成4个块的时候取0-3, 分成两个块的时候取0和1
196.         */
197.        //左上
198.        hls_prediction_unit(s, x0, y0, cb_size / 2, cb_size / 2, log2_cb_size, 0, idx);
199.        //右上
200.        hls_prediction_unit(s, x0 + cb_size / 2, y0, cb_size / 2, cb_size / 2, log2_cb_size, 1, idx);
201.        //左下
202.        hls_prediction_unit(s, x0, y0 + cb_size / 2, cb_size / 2, cb_size / 2, log2_cb_size, 2, idx);
203.        //右下
204.        hls_prediction_unit(s, x0 + cb_size / 2, y0 + cb_size / 2, cb_size / 2, cb_size / 2, log2_cb_size, 3, idx);
205.        break;
206.    }
207. }
208. }
209. }
210. }
211. }
212. }
213. }
214. }
215. }
216. }
217. }
218. }
219. }
220. }
221. }
222. }
223. }
224. }
225. }
226. }
227. }
228. }
229. }
230. }
231. }
232. }
233. }
234. }
235. }
236. }
237. }
238. }
239. }
240. }
241. }
242. }
243. }
244. }
245. }
246. }
247. }
248. }
249. }
250. }
251. }
252. }
253. }
254. }
255. }
256. }
257. }
258. }
259. }
260. }
261. }
262. }
263. }
264. }
265. }
266. }
267. }
268. }
269. }
270. }
271. }
272. }
273. }
274. }
275. }
276. }
277. }
278. }
279. }
280. }
281. }
282. }
283. }
284. }
285. }
286. }
287. }
288. }
289. }
290. }
291. }
292. }
293. }
294. }
295. }
296. }
297. }
298. }
299. }
300. }
301. }
302. }
303. }
304. }
305. }
306. }
307. }
308. }
309. }
310. }
311. }
312. }
313. }
314. }
315. }
316. }
317. }
318. }
319. }
320. }
321. }
322. }
323. }
324. }
325. }
326. }
327. }
328. }
329. }
330. }
331. }
332. }
333. }
334. }
335. }
336. }
337. }
338. }
339. }
340. }
341. }
342. }
343. }
344. }
345. }
346. }
347. }
348. }
349. }
350. }
351. }
352. }
353. }
354. }
355. }
356. }
357. }
358. }
359. }
360. }
361. }
362. }
363. }
364. }
365. }
366. }
367. }
368. }
369. }
370. }
371. }
372. }
373. }
374. }
375. }
376. }
377. }
378. }
379. }
380. }
381. }
382. }
383. }
384. }
385. }
386. }
387. }
388. }
389. }
390. }
391. }
392. }
393. }
394. }
395. }
396. }
397. }
398. }
399. }
400. }
401. }
402. }
403. }
404. }
405. }
406. }
407. }
408. }
409. }
410. }
411. }
412. }
413. }
414. }
415. }
416. }
417. }
418. }
419. }
420. }
421. }
422. }
423. }
424. }
425. }
426. }
427. }
428. }
429. }
430. }
431. }
432. }
433. }
434. }
435. }
436. }
437. }
438. }
439. }
440. }
441. }
442. }
443. }
444. }
445. }
446. }
447. }
448. }
449. }
450. }
451. }
452. }
453. }
454. }
455. }
456. }
457. }
458. }
459. }
460. }
461. }
462. }
463. }
464. }
465. }
466. }
467. }
468. }
469. }
470. }
471. }
472. }
473. }
474. }
475. }
476. }
477. }
478. }
479. }
480. }
481. }
482. }
483. }
484. }
485. }
486. }
487. }
488. }
489. }
490. }
491. }
492. }
493. }
494. }
495. }
496. }
497. }
498. }
499. }
500. }
501. }
502. }
503. }
504. }
505. }
506. }
507. }
508. }
509. }
510. }
511. }
512. }
513. }
514. }
515. }
516. }
517. }
518. }
519. }
520. }
521. }
522. }
523. }
524. }
525. }
526. }
527. }
528. }
529. }
530. }
531. }
532. }
533. }
534. }
535. }
536. }
53
```

```

152.         * +-----+-----+
153.         * |         |         |
154.         * |         |         |
155.         * |         |         |
156.         * +         +         +
157.         * |         |         |
158.         * |         |         |
159.         * |         |         |
160.         * +-----+-----+
161.         *
162.         */
163.         //左
164.         hls_prediction_unit(s, x0,                y0, cb_size / 2, cb_size, log2_cb_size, 0, idx - 1);
165.         //右
166.         hls_prediction_unit(s, x0 + cb_size / 2, y0, cb_size / 2, cb_size, log2_cb_size, 1, idx - 1);
167.         break;
168.     case PART_2NxNU:
169.         /*
170.          * PART_2NxNU (Upper) :
171.          * +-----+-----+
172.          * |         |         |
173.          * +-----+-----+
174.          * |         |         |
175.          * +         +         +
176.          * |         |         |
177.          * |         |         |
178.          * |         |         |
179.          * +-----+-----+
180.          *
181.          */
182.          //上
183.          hls_prediction_unit(s, x0, y0,                cb_size, cb_size / 4, log2_cb_size, 0, idx);
184.          //下
185.          hls_prediction_unit(s, x0, y0 + cb_size / 4, cb_size, cb_size * 3 / 4, log2_cb_size, 1, idx);
186.          break;
187.     case PART_2NxND:
188.         /*
189.          * PART_2NxND (Down) :
190.          * +-----+-----+
191.          * |         |         |
192.          * |         |         |
193.          * |         |         |
194.          * +         +         +
195.          * |         |         |
196.          * +-----+-----+
197.          * |         |         |
198.          * +-----+-----+
199.          *
200.          */
201.          //上
202.          hls_prediction_unit(s, x0, y0,                cb_size, cb_size * 3 / 4, log2_cb_size, 0, idx);
203.          //下
204.          hls_prediction_unit(s, x0, y0 + cb_size * 3 / 4, cb_size, cb_size / 4, log2_cb_size, 1, idx);
205.          break;
206.     case PART_nLx2N:
207.         /*
208.          * PART_nLx2N (Left):
209.          * +---+---+-----+
210.          * |   |   |         |
211.          * |   |   |         |
212.          * |   |   |         |
213.          * +   +   +         +
214.          * |   |   |         |
215.          * |   |   |         |
216.          * |   |   |         |
217.          * +---+---+-----+
218.          *
219.          */
220.          //左
221.          hls_prediction_unit(s, x0,                y0, cb_size / 4, cb_size, log2_cb_size, 0, idx - 2);
222.          //右
223.          hls_prediction_unit(s, x0 + cb_size / 4, y0, cb_size * 3 / 4, cb_size, log2_cb_size, 1, idx - 2);
224.          break;
225.     case PART_nRx2N:
226.         /*
227.          * PART_nRx2N (Right):
228.          * +-----+---+---+
229.          * |         |   |   |
230.          * |         |   |   |
231.          * |         |   |   |
232.          * +         +   +   +
233.          * |         |   |   |
234.          * |         |   |   |
235.          * |         |   |   |
236.          * +-----+---+---+
237.          *
238.          */
239.          //左
240.          hls_prediction_unit(s, x0,                y0, cb_size * 3 / 4, cb_size, log2_cb_size, 0, idx - 2);
241.          //右
242.          hls_prediction_unit(s, x0 + cb_size * 3 / 4, y0, cb_size / 4, cb_size, log2_cb_size, 1, idx - 2);
243.          break;

```



```

244.         case PART_NxN:
245.             /*
246.              * PART_NxN:
247.              * +-----+-----+
248.              * |         |         |
249.              * |         |         |
250.              * |         |         |
251.              * +-----+-----+
252.              * |         |         |
253.              * |         |         |
254.              * |         |         |
255.              * +-----+-----+
256.              */
257.             hls_prediction_unit(s, x0, y0, cb_size / 2, cb_size / 2, log2_cb_size, 0, idx - 1);
258.             hls_prediction_unit(s, x0 + cb_size / 2, y0, cb_size / 2, cb_size / 2, log2_cb_size, 1, idx - 1);
259.             hls_prediction_unit(s, x0, y0 + cb_size / 2, cb_size / 2, cb_size / 2, log2_cb_size, 2, idx - 1);
260.             hls_prediction_unit(s, x0 + cb_size / 2, y0 + cb_size / 2, cb_size / 2, cb_size / 2, log2_cb_size, 3, idx - 1);
261.             break;
262.         }
263.     }
264. }
265.
266. if (!pcm_flag) {
267.     int rqt_root_cbf = 1;
268.
269.     if (lc->cu.pred_mode != MODE_INTRA &&
270.         !(lc->cu.part_mode == PART_2Nx2N && lc->pu.merge_flag)) {
271.         rqt_root_cbf = ff_hevc_no_residual_syntax_flag_decode(s);
272.     }
273.     if (rqt_root_cbf) {
274.         const static int cbf[2] = { 0 };
275.         lc->cu.max_trafo_depth = lc->cu.pred_mode == MODE_INTRA ?
276.             s->sps->max_transform_hierarchy_depth_intra + lc->cu.intra_split_flag :
277.             s->sps->max_transform_hierarchy_depth_inter;
278.         //处理TU四叉树
279.         ret = hls_transform_tree(s, x0, y0, x0, y0, x0, y0,
280.                                 log2_cb_size,
281.                                 log2_cb_size, 0, 0, cbf, cbf);
282.         if (ret < 0)
283.             return ret;
284.     } else {
285.         if (!s->sh.disable_deblocking_filter_flag)
286.             ff_hevc_deblocking_boundary_strengths(s, x0, y0, log2_cb_size);
287.     }
288. }
289. }
290.
291. if (s->pps->cu_qp_delta_enabled_flag && lc->tu.is_cu_qp_delta_coded == 0)
292.     ff_hevc_set_qPy(s, x0, y0, log2_cb_size);
293.
294. x = y_cb * min_cb_width + x_cb;
295. for (y = 0; y < length; y++) {
296.     memset(&s->qp_y_tab[x], lc->qp_y, length);
297.     x += min_cb_width;
298. }
299.
300. if (((x0 + (1 < log2_cb_size)) & qp_block_mask) == 0 &&
301.     ((y0 + (1 < log2_cb_size)) & qp_block_mask) == 0) {
302.     lc->qPy_pred = lc->qp_y;
303. }
304.
305. set_ct_depth(s, x0, y0, log2_cb_size, lc->ct_depth);
306.
307. return 0;
308. }

```

从源代码可以看出，hls_coding_unit()主要进行了两个方面的处理：

- (1) 调用hls_prediction_unit()处理PU。
- (2) 调用hls_transform_tree()处理TU树。

本文分析第二个函数hls_transform_tree()中相关的代码。

hls_transform_tree()

hls_transform_tree()用于解析TU四叉树句法。该函数的定义如下所示。

```

1. //处理TU四叉树
2. static int hls_transform_tree(HEVCContext *s, int x0, int y0,
3.                               int xBase, int yBase, int cb_xBase, int cb_yBase,
4.                               int log2_cb_size, int log2_trafo_size,
5.                               int trafo_depth, int blk_idx,
6.                               const int *base_cbf_cb, const int *base_cbf_cr)
7. {
8.     HEVCLocalContext *lc = s->HEVClc;
9.     uint8_t split_transform_flag;
10.    int cbf_cb[2];

```

```

11.     int cbf_cr[2];
12.     int ret;
13.
14.     cbf_cb[0] = base_cbf_cb[0];
15.     cbf_cb[1] = base_cbf_cb[1];
16.     cbf_cr[0] = base_cbf_cr[0];
17.     cbf_cr[1] = base_cbf_cr[1];
18.
19.     if (lc->cu.intra_split_flag) {
20.         if (trafo_depth == 1) {
21.             lc->tu.intra_pred_mode = lc->pu.intra_pred_mode[blk_idx];
22.             if (s->sps->chroma_format_idc == 3) {
23.                 lc->tu.intra_pred_mode_c = lc->pu.intra_pred_mode_c[blk_idx];
24.                 lc->tu.chroma_mode_c = lc->pu.chroma_mode_c[blk_idx];
25.             } else {
26.                 lc->tu.intra_pred_mode_c = lc->pu.intra_pred_mode_c[0];
27.                 lc->tu.chroma_mode_c = lc->pu.chroma_mode_c[0];
28.             }
29.         }
30.     } else {
31.         lc->tu.intra_pred_mode = lc->pu.intra_pred_mode[0];
32.         lc->tu.intra_pred_mode_c = lc->pu.intra_pred_mode_c[0];
33.         lc->tu.chroma_mode_c = lc->pu.chroma_mode_c[0];
34.     }
35.
36.     if (log2_trafo_size <= s->sps->log2_max_trafo_size &&
37.         log2_trafo_size > s->sps->log2_min_tb_size &&
38.         trafo_depth < lc->cu.max_trafo_depth &&
39.         !(lc->cu.intra_split_flag && trafo_depth == 0)) {
40.         split_transform_flag = ff_hevc_split_transform_flag_decode(s, log2_trafo_size);
41.     } else {
42.         int inter_split = s->sps->max_transform_hierarchy_depth_inter == 0 &&
43.             lc->cu.pred_mode == MODE_INTER &&
44.             lc->cu.part_mode != PART_2Nx2N &&
45.             trafo_depth == 0;
46.         //split_transform_flag标记当前TU是否要进行四叉树划分
47.         //为1则需要划分为4个大小相等的, 为0则不再划分
48.         split_transform_flag = log2_trafo_size > s->sps->log2_max_trafo_size ||
49.             (lc->cu.intra_split_flag && trafo_depth == 0) ||
50.             inter_split;
51.     }
52.
53.     if (log2_trafo_size > 2 || s->sps->chroma_format_idc == 3) {
54.         if (trafo_depth == 0 || cbf_cb[0]) {
55.             cbf_cb[0] = ff_hevc_cbf_cb_cr_decode(s, trafo_depth);
56.             if (s->sps->chroma_format_idc == 2 && (!split_transform_flag || log2_trafo_size == 3)) {
57.                 cbf_cb[1] = ff_hevc_cbf_cb_cr_decode(s, trafo_depth);
58.             }
59.         }
60.
61.         if (trafo_depth == 0 || cbf_cr[0]) {
62.             cbf_cr[0] = ff_hevc_cbf_cb_cr_decode(s, trafo_depth);
63.             if (s->sps->chroma_format_idc == 2 && (!split_transform_flag || log2_trafo_size == 3)) {
64.                 cbf_cr[1] = ff_hevc_cbf_cb_cr_decode(s, trafo_depth);
65.             }
66.         }
67.     }
68.
69.     //如果当前TU要进行四叉树划分
70.     if (split_transform_flag) {
71.         const int trafo_size_split = 1 << (log2_trafo_size - 1);
72.         const int x1 = x0 + trafo_size_split;
73.         const int y1 = y0 + trafo_size_split;
74.
75.         #define SUBDIVIDE(x, y, idx) \
76.         do { \
77.             ret = hls_transform_tree(s, x, y, x0, y0, cb_xBase, cb_yBase, log2_cb_size, \
78.                 log2_trafo_size - 1, trafo_depth + 1, idx, \
79.                 cbf_cb, cbf_cr); \
80.             if (ret < 0) \
81.                 return ret; \
82.         } while (0)
83.         //递归调用
84.         SUBDIVIDE(x0, y0, 0);
85.         SUBDIVIDE(x1, y0, 1);
86.         SUBDIVIDE(x0, y1, 2);
87.         SUBDIVIDE(x1, y1, 3);
88.
89.         #undef SUBDIVIDE
90.     } else {
91.         int min_tu_size = 1 << s->sps->log2_min_tb_size;
92.         int log2_min_tu_size = s->sps->log2_min_tb_size;
93.         int min_tu_width = s->sps->min_tb_width;
94.         int cbf_luma = 1;
95.
96.         if (lc->cu.pred_mode == MODE_INTRA || trafo_depth != 0 ||
97.             cbf_cb[0] || cbf_cr[0] ||
98.             (s->sps->chroma_format_idc == 2 && (cbf_cb[1] || cbf_cr[1]))) {
99.             cbf_luma = ff_hevc_cbf_luma_decode(s, trafo_depth);
100.        }
101.        //处理TU-帧内预测、DCT反变换

```

```

102.         ret = hls_transform_unit(s, x0, y0, xBase, yBase, cb_xBase, cb_yBase,
103.                                log2_cb_size, log2_trafo_size,
104.                                blk_idx, cbf_luma, cbf_cb, cbf_cr);
105.
106.         if (ret < 0)
107.             return ret;
108.         // TODO: store cbf_luma somewhere else
109.         if (cbf_luma) {
110.             int i, j;
111.             for (i = 0; i < (1 << log2_trafo_size); i += min_tu_size)
112.                 for (j = 0; j < (1 << log2_trafo_size); j += min_tu_size) {
113.                     int x_tu = (x0 + j) >> log2_min_tu_size;
114.                     int y_tu = (y0 + i) >> log2_min_tu_size;
115.                     s->cbf_luma[y_tu * min_tu_width + x_tu] = 1;
116.                 }
117.         }
118.         if (!s->sh.disable_deblocking_filter_flag) {
119.             ff_hevc_deblocking_boundary_strengths(s, x0, y0, log2_trafo_size);
120.             if (s->pps->transquant_bypass_enable_flag &&
121.                 lc->cu.transquant_bypass_flag)
122.                 set_deblocking_bypass(s, x0, y0, log2_trafo_size);
123.         }
124.         return 0;
125.     }

```

从源代码可以看出，hls_transform_tree()首先调用ff_hevc_split_transform_flag_decode()判断当前TU是否还需要划分。如果需要划分的话，就会递归调用4次hls_transform_tree()分别对4个子块继续进行二叉树解析；如果不需要划分，就会调用hls_transform_unit()对TU进行解码。总而言之，hls_transform_tree()会解析出来一个TU树中的所有TU，并且对每一个TU逐一调用hls_transform_unit()进行解码。

hls_transform_unit()

hls_transform_unit()用于解码一个TU，该函数的定义如下所示。

```

1.  //处理TU- 帧内预测、DCT反变换
2.  static int hls_transform_unit(HEVCContext *s, int x0, int y0,
3.                               int xBase, int yBase, int cb_xBase, int cb_yBase,
4.                               int log2_cb_size, int log2_trafo_size,
5.                               int blk_idx, int cbf_luma, int *cbf_cb, int *cbf_cr)
6.  {
7.      HEVCLocalContext *lc = s->HEVC1c;
8.      const int log2_trafo_size_c = log2_trafo_size - s->sps->hshift[1];
9.      int i;
10.
11.      if (lc->cu.pred_mode == MODE_INTRA) {
12.          int trafo_size = 1 << log2_trafo_size;
13.          ff_hevc_set_neighbour_available(s, x0, y0, trafo_size, trafo_size);
14.
15.          //注意：帧内预测也是在这里完成
16.          //帧内预测
17.          //log2_trafo_size为当前TU大小取log2()之后的值
18.          s->hpc.intra_pred[log2_trafo_size - 2](s, x0, y0, 0);
19.      }
20.
21.      if (cbf_luma || cbf_cb[0] || cbf_cr[0] ||
22.          (s->sps->chroma_format_idc == 2 && (cbf_cb[1] || cbf_cr[1]))) {
23.          int scan_idx = SCAN_DIAG;
24.          int scan_idx_c = SCAN_DIAG;
25.          int cbf_chroma = cbf_cb[0] || cbf_cr[0] ||
26.                          (s->sps->chroma_format_idc == 2 &&
27.                           (cbf_cb[1] || cbf_cr[1]));
28.
29.          if (s->pps->cu_qp_delta_enabled_flag && !lc->tu.is_cu_qp_delta_coded) {
30.              lc->tu.cu_qp_delta = ff_hevc_cu_qp_delta_abs(s);
31.              if (lc->tu.cu_qp_delta != 0)
32.                  if (ff_hevc_cu_qp_delta_sign_flag(s) == 1)
33.                      lc->tu.cu_qp_delta = -lc->tu.cu_qp_delta;
34.              lc->tu.is_cu_qp_delta_coded = 1;
35.
36.              if (lc->tu.cu_qp_delta < -(26 + s->sps->qp_bd_offset / 2) ||
37.                  lc->tu.cu_qp_delta > (25 + s->sps->qp_bd_offset / 2)) {
38.                  av_log(s->avctx, AV_LOG_ERROR,
39.                       "The cu_qp_delta %d is outside the valid range "
40.                       "[%d, %d].\n",
41.                       lc->tu.cu_qp_delta,
42.                       -(26 + s->sps->qp_bd_offset / 2),
43.                       (25 + s->sps->qp_bd_offset / 2));
44.                  return AERROR_INVALIDDATA;
45.              }
46.
47.              ff_hevc_set_qPy(s, cb_xBase, cb_yBase, log2_cb_size);
48.          }
49.
50.          if (s->sh.cu_chroma_qp_offset_enabled_flag && cbf_chroma &&
51.              !lc->cu.transquant_bypass_flag && !lc->tu.is_cu_chroma_qp_offset_coded) {
52.              int cu_chroma_qp_offset_flag = ff_hevc_cu_chroma_qp_offset_flag(s);
53.              if (cu_chroma_qp_offset_flag) {

```

```

54.     int cu_chroma_qp_offset_idx = 0;
55.     if (s->pps->chroma_qp_offset_list_len_minus1 > 0) {
56.         cu_chroma_qp_offset_idx = ff_hevc_cu_chroma_qp_offset_idx(s);
57.         av_log(s->avctx, AV_LOG_ERROR,
58.             "cu_chroma_qp_offset_idx not yet tested.\n");
59.     }
60.     lc->tu.cu_qp_offset_cb = s->pps->cb_qp_offset_list[cu_chroma_qp_offset_idx];
61.     lc->tu.cu_qp_offset_cr = s->pps->cr_qp_offset_list[cu_chroma_qp_offset_idx];
62. } else {
63.     lc->tu.cu_qp_offset_cb = 0;
64.     lc->tu.cu_qp_offset_cr = 0;
65. }
66. lc->tu.is_cu_chroma_qp_offset_coded = 1;
67. }
68.
69. if (lc->cu.pred_mode == MODE_INTRA && log2_trafo_size < 4) {
70.     if (lc->tu.intra_pred_mode >= 6 &&
71.         lc->tu.intra_pred_mode <= 14) {
72.         scan_idx = SCAN_VERT;
73.     } else if (lc->tu.intra_pred_mode >= 22 &&
74.         lc->tu.intra_pred_mode <= 30) {
75.         scan_idx = SCAN_HORIZ;
76.     }
77.
78.     if (lc->tu.intra_pred_mode_c >= 6 &&
79.         lc->tu.intra_pred_mode_c <= 14) {
80.         scan_idx_c = SCAN_VERT;
81.     } else if (lc->tu.intra_pred_mode_c >= 22 &&
82.         lc->tu.intra_pred_mode_c <= 30) {
83.         scan_idx_c = SCAN_HORIZ;
84.     }
85. }
86.
87. lc->tu.cross_pf = 0;
88.
89. //读取残差数据, 进行反量化, DCT反变换
90.
91. //亮度Y
92. if (cbf_luma)
93.     ff_hevc_hls_residual_coding(s, x0, y0, log2_trafo_size, scan_idx, 0); //最后1个参数为颜色分量号
94.
95. if (log2_trafo_size > 2 || s->sps->chroma_format_idc == 3) {
96.     int trafo_size_h = 1 << (log2_trafo_size_c + s->sps->hshift[1]);
97.     int trafo_size_v = 1 << (log2_trafo_size_c + s->sps->vshift[1]);
98.     lc->tu.cross_pf = (s->pps->cross_component_prediction_enabled_flag && cbf_luma &&
99.         (lc->cu.pred_mode == MODE_INTER ||
100.         (lc->tu.chroma_mode_c == 4)));
101.
102.     if (lc->tu.cross_pf) {
103.         hls_cross_component_pred(s, 0);
104.     }
105.     //色度U
106.     for (i = 0; i < (s->sps->chroma_format_idc == 2 ? 2 : 1); i++) {
107.         if (lc->cu.pred_mode == MODE_INTRA) {
108.             ff_hevc_set_neighbour_available(s, x0, y0 + (i << log2_trafo_size_c), trafo_size_h, trafo_size_v);
109.             s->hpc.intra_pred[log2_trafo_size_c - 2](s, x0, y0 + (i << log2_trafo_size_c), 1);
110.         }
111.         if (cbf_cb[i])
112.             ff_hevc_hls_residual_coding(s, x0, y0 + (i << log2_trafo_size_c),
113.                 log2_trafo_size_c, scan_idx_c, 1); //最后1个参数为颜色分量号
114.     } else
115.         if (lc->tu.cross_pf) {
116.             ptrdiff_t stride = s->frame->linesize[1];
117.             int hshift = s->sps->hshift[1];
118.             int vshift = s->sps->vshift[1];
119.             int16_t *coeffs_y = (int16_t*)lc->edge_emu_buffer;
120.             int16_t *coeffs = (int16_t*)lc->edge_emu_buffer2;
121.             int size = 1 << log2_trafo_size_c;
122.
123.             uint8_t *dst = &s->frame->data[1][(y0 >> vshift) * stride +
124.                 ((x0 >> hshift) << s->sps->pixel_shift)];
125.             for (i = 0; i < (size * size); i++) {
126.                 coeffs[i] = ((lc->tu.res_scale_val * coeffs_y[i]) >> 3);
127.             }
128.             //叠加残差数据
129.             s->hevcdsp.transform_add[log2_trafo_size_c-2](dst, coeffs, stride);
130.         }
131.     }
132.
133.     if (lc->tu.cross_pf) {
134.         hls_cross_component_pred(s, 1);
135.     }
136.     //色度V
137.     for (i = 0; i < (s->sps->chroma_format_idc == 2 ? 2 : 1); i++) {
138.         if (lc->cu.pred_mode == MODE_INTRA) {
139.             ff_hevc_set_neighbour_available(s, x0, y0 + (i << log2_trafo_size_c), trafo_size_h, trafo_size_v);
140.             s->hpc.intra_pred[log2_trafo_size_c - 2](s, x0, y0 + (i << log2_trafo_size_c), 2);
141.         }
142.         //色度Cr
143.         if (cbf_cr[i])
144.             ff_hevc_hls_residual_coding(s, x0, y0 + (i << log2_trafo_size_c),

```

```

145.                                     log2_trafo_size_c, scan_idx_c, 2);
146.
147.     else
148.     {
149.         if (lc->tu.cross_pf) {
150.             ptrdiff_t stride = s->frame->linesize[2];
151.             int hshift = s->sps->hshift[2];
152.             int vshift = s->sps->vshift[2];
153.             int16_t *coeffs_y = (int16_t*)lc->edge_emu_buffer;
154.             int16_t *coeffs = (int16_t*)lc->edge_emu_buffer2;
155.             int size = 1 << log2_trafo_size_c;
156.
157.             uint8_t *dst = &s->frame->data[2][(y0 >> vshift) * stride +
158.                                             ((x0 >> hshift) << s->sps->pixel_shift)];
159.             for (i = 0; i < (size * size); i++) {
160.                 coeffs[i] = ((lc->tu.res_scale_val * coeffs_y[i]) >> 3);
161.             }
162.             s->hevcdsp.transform_add[log2_trafo_size_c-2](dst, coeffs, stride);
163.         }
164.     }
165. } else if (blk_idx == 3) {
166.     int trafo_size_h = 1 << (log2_trafo_size + 1);
167.     int trafo_size_v = 1 << (log2_trafo_size + s->sps->vshift[1]);
168.     for (i = 0; i < (s->sps->chroma_format_idc == 2 ? 2 : 1); i++) {
169.         if (lc->cu.pred_mode == MODE_INTRA) {
170.             ff_hevc_set_neighbour_available(s, xBase, yBase + (i << log2_trafo_size),
171.                                             trafo_size_h, trafo_size_v);
172.             s->hpc.intra_pred[log2_trafo_size - 2](s, xBase, yBase + (i << log2_trafo_size), 1);
173.         }
174.         if (cbf_cb[i])
175.             ff_hevc_hls_residual_coding(s, xBase, yBase + (i << log2_trafo_size),
176.                                         log2_trafo_size, scan_idx_c, 1);
177.     }
178.     for (i = 0; i < (s->sps->chroma_format_idc == 2 ? 2 : 1); i++) {
179.         if (lc->cu.pred_mode == MODE_INTRA) {
180.             ff_hevc_set_neighbour_available(s, xBase, yBase + (i << log2_trafo_size),
181.                                             trafo_size_h, trafo_size_v);
182.             s->hpc.intra_pred[log2_trafo_size - 2](s, xBase, yBase + (i << log2_trafo_size), 2);
183.         }
184.         if (cbf_cr[i])
185.             ff_hevc_hls_residual_coding(s, xBase, yBase + (i << log2_trafo_size),
186.                                         log2_trafo_size, scan_idx_c, 2);
187.     }
188. }
189. } else if (lc->cu.pred_mode == MODE_INTRA) {
190.     if (log2_trafo_size > 2 || s->sps->chroma_format_idc == 3) {
191.         int trafo_size_h = 1 << (log2_trafo_size_c + s->sps->hshift[1]);
192.         int trafo_size_v = 1 << (log2_trafo_size_c + s->sps->vshift[1]);
193.         ff_hevc_set_neighbour_available(s, x0, y0, trafo_size_h, trafo_size_v);
194.         s->hpc.intra_pred[log2_trafo_size_c - 2](s, x0, y0, 1);
195.         s->hpc.intra_pred[log2_trafo_size_c - 2](s, x0, y0, 2);
196.         if (s->sps->chroma_format_idc == 2) {
197.             ff_hevc_set_neighbour_available(s, x0, y0 + (1 << log2_trafo_size_c),
198.                                             trafo_size_h, trafo_size_v);
199.             s->hpc.intra_pred[log2_trafo_size_c - 2](s, x0, y0 + (1 << log2_trafo_size_c), 1);
200.             s->hpc.intra_pred[log2_trafo_size_c - 2](s, x0, y0 + (1 << log2_trafo_size_c), 2);
201.         }
202.     } else if (blk_idx == 3) {
203.         int trafo_size_h = 1 << (log2_trafo_size + 1);
204.         int trafo_size_v = 1 << (log2_trafo_size + s->sps->vshift[1]);
205.         ff_hevc_set_neighbour_available(s, xBase, yBase,
206.                                         trafo_size_h, trafo_size_v);
207.         s->hpc.intra_pred[log2_trafo_size - 2](s, xBase, yBase, 1);
208.         s->hpc.intra_pred[log2_trafo_size - 2](s, xBase, yBase, 2);
209.         if (s->sps->chroma_format_idc == 2) {
210.             ff_hevc_set_neighbour_available(s, xBase, yBase + (1 << (log2_trafo_size)),
211.                                             trafo_size_h, trafo_size_v);
212.             s->hpc.intra_pred[log2_trafo_size - 2](s, xBase, yBase + (1 << (log2_trafo_size)), 1);
213.             s->hpc.intra_pred[log2_trafo_size - 2](s, xBase, yBase + (1 << (log2_trafo_size)), 2);
214.         }
215.     }
216. }
217. return 0;
}

```

从源代码可以看出，如果是帧内CU的话，hls_transform_unit()会调用HEVCPredContext的intra_pred[]汇编函数进行帧内预测；然后不论帧内预测还是帧间CU都会调用ff_hevc_hls_residual_coding()解码残差数据，并叠加在预测数据上。

ff_hevc_hls_residual_coding()

ff_hevc_hls_residual_coding()用于读取残差数据并进行DCT反变换。该函数的定义如下所示。

```

1. //读取残差数据，DCT反变换
2. void ff_hevc_hls_residual_coding(HEVCContext *s, int x0, int y0,
3.                                 int log2_trafo_size, enum ScanType scan_idx,
4.                                 int c_idx)
5. {
6.     #define GET_COEFF(coeff) s->hls_residual_coding[log2_trafo_size][scan_idx][c_idx][coeff]

```

```

0. #define GET_COORD(v,iset, n) \
7. do { \
8.     x_c = (x_cg << 2) + scan_x_off[n]; \
9.     y_c = (y_cg << 2) + scan_y_off[n]; \
10. } while (0)
11. HEVCLocalContext *lc = s->HEVCLc;
12. int transform_skip_flag = 0;
13.
14. int last_significant_coeff_x, last_significant_coeff_y;
15. int last_scan_pos;
16. int n_end;
17. int num_coeff = 0;
18. int greater1_ctx = 1;
19.
20. int num_last_subset;
21. int x_cg_last_sig, y_cg_last_sig;
22.
23. const uint8_t *scan_x_cg, *scan_y_cg, *scan_x_off, *scan_y_off;
24.
25. ptrdiff_t stride = s->frame->linesize[c_idx];
26. int hshift = s->sps->hshift[c_idx];
27. int vshift = s->sps->vshift[c_idx];
28. uint8_t *dst = &s->frame->data[c_idx][(y0 >> vshift) * stride +
29.                                     (x0 >> hshift) << s->sps->pixel_shift)];
30. int16_t *coeffs = (int16_t*)(c_idx ? lc->edge_emu_buffer2 : lc->edge_emu_buffer);
31. uint8_t significant_coeff_group_flag[8][8] = {{0}};
32. int explicit_rdpem_flag = 0;
33. int explicit_rdpem_dir_flag;
34.
35. int trafo_size = 1 << log2_trafo_size;
36. int i;
37. int qp, shift, add, scale, scale_m;
38. const uint8_t level_scale[] = { 40, 45, 51, 57, 64, 72 };
39. const uint8_t *scale_matrix = NULL;
40. uint8_t dc_scale;
41. int pred_mode_intra = (c_idx == 0) ? lc->tu.intra_pred_mode :
42.                         lc->tu.intra_pred_mode_c;
43.
44. memset(coeffs, 0, trafo_size * trafo_size * sizeof(int16_t));
45.
46. // Derive QP for dequant
47. if (!lc->cu.cu.transquant_bypass_flag) {
48.     static const int qp_c[] = { 29, 30, 31, 32, 33, 33, 34, 34, 35, 35, 36, 36, 37, 37 };
49.     static const uint8_t rem6[51 + 4 * 6 + 1] = {
50.         0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 0, 1, 2,
51.         3, 4, 5, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5,
52.         0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3,
53.         4, 5, 0, 1, 2, 3, 4, 5, 0, 1
54.     };
55.
56.     static const uint8_t div6[51 + 4 * 6 + 1] = {
57.         0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 3, 3, 3,
58.         3, 3, 3, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6,
59.         7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 10, 10, 10, 10,
60.         10, 10, 11, 11, 11, 11, 11, 11, 12, 12
61.     };
62.     int qp_y = lc->qp_y;
63.
64.     if (s->pps->transform_skip_enabled_flag &&
65.         log2_trafo_size <= s->pps->log2_max_transform_skip_block_size) {
66.         transform_skip_flag = ff_hevc_transform_skip_flag_decode(s, c_idx);
67.     }
68.
69.     if (c_idx == 0) {
70.         qp = qp_y + s->sps->qp_bd_offset;
71.     } else {
72.         int qp_i, offset;
73.
74.         if (c_idx == 1)
75.             offset = s->pps->cb_qp_offset + s->sh.slice_cb_qp_offset +
76.                     lc->tu.cu_qp_offset_cb;
77.         else
78.             offset = s->pps->cr_qp_offset + s->sh.slice_cr_qp_offset +
79.                     lc->tu.cu_qp_offset_cr;
80.
81.         qp_i = av_clip(qp_y + offset, - s->sps->qp_bd_offset, 57);
82.         if (s->sps->chroma_format_idc == 1) {
83.             if (qp_i < 30)
84.                 qp = qp_i;
85.             else if (qp_i > 43)
86.                 qp = qp_i - 6;
87.             else
88.                 qp = qp_c[qp_i - 30];
89.         } else {
90.             if (qp_i > 51)
91.                 qp = 51;
92.             else
93.                 qp = qp_i;
94.         }
95.
96.         qp += s->sps->qp_bd_offset;
97.     }

```

```

98.
99.     shift    = s->sps->bit_depth + log2_trafo_size - 5;
100.     add      = 1 << (shift-1);
101.     scale    = level_scale[rem6[qp]] << (div6[qp]);
102.     scale_m  = 16; // default when no custom scaling lists.
103.     dc_scale = 16;
104.
105.     if (s->sps->scaling_list_enable_flag && !(transform_skip_flag && log2_trafo_size > 2)) {
106.         const ScalingList *sl = s->pps->scaling_list_data_present_flag ?
107.             &s->pps->scaling_list : &s->sps->scaling_list;
108.         int matrix_id = lc->cu.pred_mode != MODE_INTRA;
109.
110.         matrix_id = 3 * matrix_id + c_idx;
111.
112.         scale_matrix = sl->sl[log2_trafo_size - 2][matrix_id];
113.         if (log2_trafo_size >= 4)
114.             dc_scale = sl->sl_dc[log2_trafo_size - 4][matrix_id];
115.     }
116. } else {
117.     shift    = 0;
118.     add      = 0;
119.     scale    = 0;
120.     dc_scale = 0;
121. }
122.
123. if (lc->cu.pred_mode == MODE_INTER && s->sps->explicit_rdpdm_enabled_flag &&
124.     (transform_skip_flag || lc->cu.transquant_bypass_flag)) {
125.     explicit_rdpdm_flag = explicit_rdpdm_flag_decode(s, c_idx);
126.     if (explicit_rdpdm_flag) {
127.         explicit_rdpdm_dir_flag = explicit_rdpdm_dir_flag_decode(s, c_idx);
128.     }
129. }
130.
131. last_significant_coeff_xy_prefix_decode(s, c_idx, log2_trafo_size,
132.     &last_significant_coeff_x, &last_significant_coeff_y);
133.
134. if (last_significant_coeff_x > 3) {
135.     int suffix = last_significant_coeff_suffix_decode(s, last_significant_coeff_x);
136.     last_significant_coeff_x = (1 << ((last_significant_coeff_x >> 1) - 1)) *
137.         (2 + (last_significant_coeff_x & 1)) +
138.         suffix;
139. }
140.
141. if (last_significant_coeff_y > 3) {
142.     int suffix = last_significant_coeff_suffix_decode(s, last_significant_coeff_y);
143.     last_significant_coeff_y = (1 << ((last_significant_coeff_y >> 1) - 1)) *
144.         (2 + (last_significant_coeff_y & 1)) +
145.         suffix;
146. }
147.
148. if (scan_idx == SCAN_VERT)
149.     FFSWAP(int, last_significant_coeff_x, last_significant_coeff_y);
150.
151. x_cg_last_sig = last_significant_coeff_x >> 2;
152. y_cg_last_sig = last_significant_coeff_y >> 2;
153.
154. switch (scan_idx) {
155. case SCAN_DIAG: {
156.     int last_x_c = last_significant_coeff_x & 3;
157.     int last_y_c = last_significant_coeff_y & 3;
158.
159.     scan_x_off = ff_hevc_diag_scan4x4_x;
160.     scan_y_off = ff_hevc_diag_scan4x4_y;
161.     num_coeff = diag_scan4x4_inv[last_y_c][last_x_c];
162.     if (trafo_size == 4) {
163.         scan_x_cg = scan_lx1;
164.         scan_y_cg = scan_ly1;
165.     } else if (trafo_size == 8) {
166.         num_coeff += diag_scan2x2_inv[y_cg_last_sig][x_cg_last_sig] << 4;
167.         scan_x_cg = diag_scan2x2_x;
168.         scan_y_cg = diag_scan2x2_y;
169.     } else if (trafo_size == 16) {
170.         num_coeff += diag_scan4x4_inv[y_cg_last_sig][x_cg_last_sig] << 4;
171.         scan_x_cg = ff_hevc_diag_scan4x4_x;
172.         scan_y_cg = ff_hevc_diag_scan4x4_y;
173.     } else { // trafo_size == 32
174.         num_coeff += diag_scan8x8_inv[y_cg_last_sig][x_cg_last_sig] << 4;
175.         scan_x_cg = ff_hevc_diag_scan8x8_x;
176.         scan_y_cg = ff_hevc_diag_scan8x8_y;
177.     }
178.     break;
179. }
180. case SCAN_HORIZ:
181.     scan_x_cg = horiz_scan2x2_x;
182.     scan_y_cg = horiz_scan2x2_y;
183.     scan_x_off = horiz_scan4x4_x;
184.     scan_y_off = horiz_scan4x4_y;
185.     num_coeff = horiz_scan8x8_inv[last_significant_coeff_y][last_significant_coeff_x];
186.     break;
187. default: //SCAN_VERT
188.     scan_x_cg = horiz_scan2x2_y;

```

```

189.     scan_y_cg = horiz_scan2x2_x;
190.     scan_x_off = horiz_scan4x4_y;
191.     scan_y_off = horiz_scan4x4_x;
192.     num_coeff = horiz_scan8x8_inv[last_significant_coeff_x][last_significant_coeff_y];
193.     break;
194. }
195. num_coeff++;
196. num_last_subset = (num_coeff - 1) >> 4;
197.
198. for (i = num_last_subset; i >= 0; i--) {
199.     int n, m;
200.     int x_cg, y_cg, x_c, y_c, pos;
201.     int implicit_non_zero_coeff = 0;
202.     int64_t trans_coeff_level;
203.     int prev_sig = 0;
204.     int offset = i << 4;
205.     int rice_init = 0;
206.
207.     uint8_t significant_coeff_flag_idx[16];
208.     uint8_t nb_significant_coeff_flag = 0;
209.
210.     x_cg = scan_x_cg[i];
211.     y_cg = scan_y_cg[i];
212.
213.     if ((i < num_last_subset) && (i > 0)) {
214.         int ctx_cg = 0;
215.         if (x_cg < (1 << (log2_trafo_size - 2)) - 1)
216.             ctx_cg += significant_coeff_group_flag[x_cg + 1][y_cg];
217.         if (y_cg < (1 << (log2_trafo_size - 2)) - 1)
218.             ctx_cg += significant_coeff_group_flag[x_cg][y_cg + 1];
219.
220.         significant_coeff_group_flag[x_cg][y_cg] =
221.             significant_coeff_group_flag_decode(s, c_idx, ctx_cg);
222.         implicit_non_zero_coeff = 1;
223.     } else {
224.         significant_coeff_group_flag[x_cg][y_cg] =
225.             ((x_cg == x_cg_last_sig && y_cg == y_cg_last_sig) ||
226.              (x_cg == 0 && y_cg == 0));
227.     }
228.
229.     last_scan_pos = num_coeff - offset - 1;
230.
231.     if (i == num_last_subset) {
232.         n_end = last_scan_pos - 1;
233.         significant_coeff_flag_idx[0] = last_scan_pos;
234.         nb_significant_coeff_flag = 1;
235.     } else {
236.         n_end = 15;
237.     }
238.
239.     if (x_cg < ((1 << log2_trafo_size) - 1) >> 2)
240.         prev_sig = !significant_coeff_group_flag[x_cg + 1][y_cg];
241.     if (y_cg < ((1 << log2_trafo_size) - 1) >> 2)
242.         prev_sig += (!significant_coeff_group_flag[x_cg][y_cg + 1] << 1);
243.
244.     if (significant_coeff_group_flag[x_cg][y_cg] && n_end >= 0) {
245.         static const uint8_t ctx_idx_map[] = {
246.             0, 1, 4, 5, 2, 3, 4, 5, 6, 6, 8, 8, 7, 7, 8, 8, // log2_trafo_size == 2
247.             1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, // prev_sig == 0
248.             2, 2, 2, 2, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, // prev_sig == 1
249.             2, 1, 0, 0, 2, 1, 0, 0, 2, 1, 0, 0, 2, 1, 0, 0, // prev_sig == 2
250.             2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 // default
251.         };
252.         const uint8_t *ctx_idx_map_p;
253.         int scf_offset = 0;
254.         if (s->sps->transform_skip_context_enabled_flag &&
255.             (transform_skip_flag || lc->cu.transquant_bypass_flag)) {
256.             ctx_idx_map_p = (uint8_t*) &ctx_idx_map[4 * 16];
257.             if (c_idx == 0) {
258.                 scf_offset = 40;
259.             } else {
260.                 scf_offset = 14 + 27;
261.             }
262.         } else {
263.             if (c_idx != 0)
264.                 scf_offset = 27;
265.             if (log2_trafo_size == 2) {
266.                 ctx_idx_map_p = (uint8_t*) &ctx_idx_map[0];
267.             } else {
268.                 ctx_idx_map_p = (uint8_t*) &ctx_idx_map[(prev_sig + 1) << 4];
269.                 if (c_idx == 0) {
270.                     if ((x_cg > 0 || y_cg > 0))
271.                         scf_offset += 3;
272.                     if (log2_trafo_size == 3) {
273.                         scf_offset += (scan_idx == SCAN_DIAG) ? 9 : 15;
274.                     } else {
275.                         scf_offset += 21;
276.                     }
277.                 } else {
278.                     if (log2_trafo_size == 3)
279.                         scf_offset += 9;

```



```

280.         else
281.             scf_offset += 12;
282.     }
283. }
284. }
285. for (n = n_end; n > 0; n--) {
286.     x_c = scan_x_off[n];
287.     y_c = scan_y_off[n];
288.     if (significant_coeff_flag_decode(s, x_c, y_c, scf_offset, ctx_idx_map_p)) {
289.         significant_coeff_flag_idx[nb_significant_coeff_flag] = n;
290.         nb_significant_coeff_flag++;
291.         implicit_non_zero_coeff = 0;
292.     }
293. }
294. if (implicit_non_zero_coeff == 0) {
295.     if (s->sps->transform_skip_context_enabled_flag &&
296.         (transform_skip_flag || lc->cu.transquant_bypass_flag)) {
297.         if (c_idx == 0) {
298.             scf_offset = 42;
299.         } else {
300.             scf_offset = 16 + 27;
301.         }
302.     } else {
303.         if (i == 0) {
304.             if (c_idx == 0)
305.                 scf_offset = 0;
306.             else
307.                 scf_offset = 27;
308.         } else {
309.             scf_offset = 2 + scf_offset;
310.         }
311.     }
312.     if (significant_coeff_flag_decode_0(s, c_idx, scf_offset) == 1) {
313.         significant_coeff_flag_idx[nb_significant_coeff_flag] = 0;
314.         nb_significant_coeff_flag++;
315.     }
316. } else {
317.     significant_coeff_flag_idx[nb_significant_coeff_flag] = 0;
318.     nb_significant_coeff_flag++;
319. }
320. }
321.
322. n_end = nb_significant_coeff_flag;
323.
324.
325. if (n_end) {
326.     int first_nz_pos_in_cg;
327.     int last_nz_pos_in_cg;
328.     int c_rice_param = 0;
329.     int first_greater1_coeff_idx = -1;
330.     uint8_t coeff_abs_level_greater1_flag[8];
331.     uint16_t coeff_sign_flag;
332.     int sum_abs = 0;
333.     int sign_hidden;
334.     int sb_type;
335.
336.
337.     // initialize first elem of coeff_abs_level_greater1_flag
338.     int ctx_set = (i > 0 && c_idx == 0) ? 2 : 0;
339.
340.     if (s->sps->persistent_rice_adaptation_enabled_flag) {
341.         if (!transform_skip_flag && !lc->cu.transquant_bypass_flag)
342.             sb_type = 2 * (c_idx == 0 ? 1 : 0);
343.         else
344.             sb_type = 2 * (c_idx == 0 ? 1 : 0) + 1;
345.         c_rice_param = lc->stat_coeff[sb_type] / 4;
346.     }
347.
348.     if (!(i == num_last_subset) && greater1_ctx == 0)
349.         ctx_set++;
350.     greater1_ctx = 1;
351.     last_nz_pos_in_cg = significant_coeff_flag_idx[0];
352.
353.     for (m = 0; m < (n_end > 8 ? 8 : n_end); m++) {
354.         int inc = (ctx_set << 2) + greater1_ctx;
355.         coeff_abs_level_greater1_flag[m] =
356.             coeff_abs_level_greater1_flag_decode(s, c_idx, inc);
357.         if (coeff_abs_level_greater1_flag[m]) {
358.             greater1_ctx = 0;
359.             if (first_greater1_coeff_idx == -1)
360.                 first_greater1_coeff_idx = m;
361.             } else if (greater1_ctx > 0 && greater1_ctx < 3) {
362.                 greater1_ctx++;
363.             }
364.         }
365.     }
366.     first_nz_pos_in_cg = significant_coeff_flag_idx[n_end - 1];
367.
368.     if (lc->cu.transquant_bypass_flag ||
369.         (lc->cu.pred_mode == MODE_INTRA &&
370.          s->sps->implicit_rdpem_enabled_flag && transform_skip_flag &&
371.          (pred_mode_intra == 10 || pred_mode_intra == 26))) ||

```

```

371.         explicit_rdpdm_flag)
372.         sign_hidden = 0;
373.     else
374.         sign_hidden = (last_nz_pos_in_cg - first_nz_pos_in_cg >= 4);
375.
376.         if (first_greater1_coeff_idx != -1) {
377.             coeff_abs_level_greater1_flag[first_greater1_coeff_idx] += coeff_abs_level_greater2_flag_decode(s, c_idx, ctx_set);
378.         }
379.         if (!s->pps->sign_data_hiding_flag || !sign_hidden) {
380.             coeff_sign_flag = coeff_sign_flag_decode(s, nb_significant_coeff_flag) << (16 - nb_significant_coeff_flag);
381.         } else {
382.             coeff_sign_flag = coeff_sign_flag_decode(s, nb_significant_coeff_flag - 1) << (16 - (nb_significant_coeff_flag - 1));
383.         }
384.
385.         for (m = 0; m < n_end; m++) {
386.             n = significant_coeff_flag_idx[m];
387.             GET_COORD(offset, n);
388.             if (m < 8) {
389.                 trans_coeff_level = 1 + coeff_abs_level_greater1_flag[m];
390.                 if (trans_coeff_level == ((m == first_greater1_coeff_idx) ? 3 : 2)) {
391.                     int last_coeff_abs_level_remaining = coeff_abs_level_remaining_decode(s, c_rice_param);
392.
393.                     trans_coeff_level += last_coeff_abs_level_remaining;
394.                     if (trans_coeff_level > (3 << c_rice_param))
395.                         c_rice_param = s->sps-
>persistent_rice_adaptation_enabled_flag ? c_rice_param + 1 : FFMIN(c_rice_param + 1, 4);
396.                     if (s->sps->persistent_rice_adaptation_enabled_flag && !rice_init) {
397.                         int c_rice_p_init = lc->stat_coeff[sb_type] / 4;
398.                         if (last_coeff_abs_level_remaining >= (3 << c_rice_p_init))
399.                             lc->stat_coeff[sb_type]++;
400.                         else if (2 * last_coeff_abs_level_remaining < (1 << c_rice_p_init))
401.                             if (lc->stat_coeff[sb_type] > 0)
402.                                 lc->stat_coeff[sb_type]--;
403.                         rice_init = 1;
404.                     }
405.                 }
406.             } else {
407.                 int last_coeff_abs_level_remaining = coeff_abs_level_remaining_decode(s, c_rice_param);
408.
409.                 trans_coeff_level = 1 + last_coeff_abs_level_remaining;
410.                 if (trans_coeff_level > (3 << c_rice_param))
411.                     c_rice_param = s->sps-
>persistent_rice_adaptation_enabled_flag ? c_rice_param + 1 : FFMIN(c_rice_param + 1, 4);
412.                 if (s->sps->persistent_rice_adaptation_enabled_flag && !rice_init) {
413.                     int c_rice_p_init = lc->stat_coeff[sb_type] / 4;
414.                     if (last_coeff_abs_level_remaining >= (3 << c_rice_p_init))
415.                         lc->stat_coeff[sb_type]++;
416.                     else if (2 * last_coeff_abs_level_remaining < (1 << c_rice_p_init))
417.                         if (lc->stat_coeff[sb_type] > 0)
418.                             lc->stat_coeff[sb_type]--;
419.                     rice_init = 1;
420.                 }
421.             }
422.             if (s->pps->sign_data_hiding_flag && sign_hidden) {
423.                 sum_abs += trans_coeff_level;
424.                 if (n == first_nz_pos_in_cg && (sum_abs & 1))
425.                     trans_coeff_level = -trans_coeff_level;
426.             }
427.             if (coeff_sign_flag >> 15)
428.                 trans_coeff_level = -trans_coeff_level;
429.             coeff_sign_flag <<= 1;
430.             if (!lc->cu->transquant_bypass_flag) {
431.                 if (s->sps->scaling_list_enable_flag && !(transform_skip_flag && log2_trafo_size > 2)) {
432.                     if (y_c || x_c || log2_trafo_size < 4) {
433.                         switch(log2_trafo_size) {
434.                             case 3: pos = (y_c << 3) + x_c; break;
435.                             case 4: pos = ((y_c >> 1) << 3) + (x_c >> 1); break;
436.                             case 5: pos = ((y_c >> 2) << 3) + (x_c >> 2); break;
437.                             default: pos = (y_c << 2) + x_c; break;
438.                         }
439.                         scale_m = scale_matrix[pos];
440.                     } else {
441.                         scale_m = dc_scale;
442.                     }
443.                 }
444.                 trans_coeff_level = (trans_coeff_level * (int64_t)scale * (int64_t)scale_m + add) >> shift;
445.                 if (trans_coeff_level < 0) {
446.                     if ((~trans_coeff_level) & 0xffffffff8000)
447.                         trans_coeff_level = -32768;
448.                 } else {
449.                     if (trans_coeff_level & 0xffffffff8000)
450.                         trans_coeff_level = 32767;
451.                 }
452.             }
453.             coeffs[y_c * trafo_size + x_c] = trans_coeff_level;
454.         }
455.     }
456. }
457.
458. if (lc->cu->transquant_bypass_flag) {
459.     if (explicit_rdpdm_flag || (s->sps->implicit_rdpdm_enabled_flag &&

```

```

460.         (pred_mode_intra == 10 || pred_mode_intra == 26))) {
461.             int mode = s->sps->implicit_rdpdm_enabled_flag ? (pred_mode_intra == 26) : explicit_rdpdm_dir_flag;
462.
463.             s->hevcdsp.transform_rdpdm(coeffs, log2_trafo_size, mode);
464.         }
465.     } else {
466.         if (transform_skip_flag) {
467.             int rot = s->sps->transform_skip_rotation_enabled_flag &&
468.                 log2_trafo_size == 2 &&
469.                 lc->cu.pred_mode == MODE_INTRA;
470.             if (rot) {
471.                 for (i = 0; i < 8; i++)
472.                     FFSWAP(int16_t, coeffs[i], coeffs[16 - i - 1]);
473.             }
474.
475.             s->hevcdsp.transform_skip(coeffs, log2_trafo_size);
476.
477.             if (explicit_rdpdm_flag || (s->sps->implicit_rdpdm_enabled_flag &&
478.                 lc->cu.pred_mode == MODE_INTRA &&
479.                 (pred_mode_intra == 10 || pred_mode_intra == 26))) {
480.                 int mode = explicit_rdpdm_flag ? explicit_rdpdm_dir_flag : (pred_mode_intra == 26);
481.
482.                 s->hevcdsp.transform_rdpdm(coeffs, log2_trafo_size, mode);
483.             }
484.         } else if (lc->cu.pred_mode == MODE_INTRA && c_idx == 0 && log2_trafo_size == 2) {
485.             //这里是4x4DST
486.             s->hevcdsp.idct_4x4_luma(coeffs);
487.         } else {
488.             int max_xy = FFMAX(last_significant_coeff_x, last_significant_coeff_y);
489.             if (max_xy == 0)
490.                 s->hevcdsp.idct_dc[log2_trafo_size-2](coeffs); //只对DC系数做IDCT的比較快的算法
491.             else {
492.                 int col_limit = last_significant_coeff_x + last_significant_coeff_y + 4;
493.                 if (max_xy < 4)
494.                     col_limit = FFMIN(4, col_limit);
495.                 else if (max_xy < 8)
496.                     col_limit = FFMIN(8, col_limit);
497.                 else if (max_xy < 12)
498.                     col_limit = FFMIN(24, col_limit);
499.                 s->hevcdsp.idct[log2_trafo_size-2](coeffs, col_limit); //普通的IDCT
500.             }
501.         }
502.     }
503.     if (lc->tu.cross_pf) {
504.         int16_t *coeffs_y = (int16_t*)lc->edge_emu_buffer;
505.
506.         for (i = 0; i < (trafo_size * trafo_size); i++) {
507.             coeffs[i] = coeffs[i] + ((lc->tu.res_scale_val * coeffs_y[i]) >> 3);
508.         }
509.     }
510.     //将IDCT的结果叠加到预测数据上
511.     s->hevcdsp.transform_add[log2_trafo_size-2](dst, coeffs, stride);
512. }

```

ff_hevc_hls_residual_coding()前半部分的一大段代码应该是用于解析残差数据的（目前还没有细看），后半部分的代码则用于对残差数据进行DCT变换。在DCT反变换的时候，调用了如下几种功能的汇编函数：

HEVCDSPContext-> idct_4x4_luma()：4x4DST反变换
HEVCDSPContext-> idct_dc[X]()：特殊的只包含DC系数的DCT反变换
HEVCDSPContext-> idct[X]()：普通的DCT反变换
HEVCDSPContext-> transform_add [X]()：残差像素数据叠加

其中不同的[X]取值代表了不同尺寸的系数块：

[0]代表4x4；
[1]代表8x8；
[2]代表16x16；
[3]代表32x32；

后文将会对上述汇编函数进行详细分析。

帧内预测和DCT反变换知识

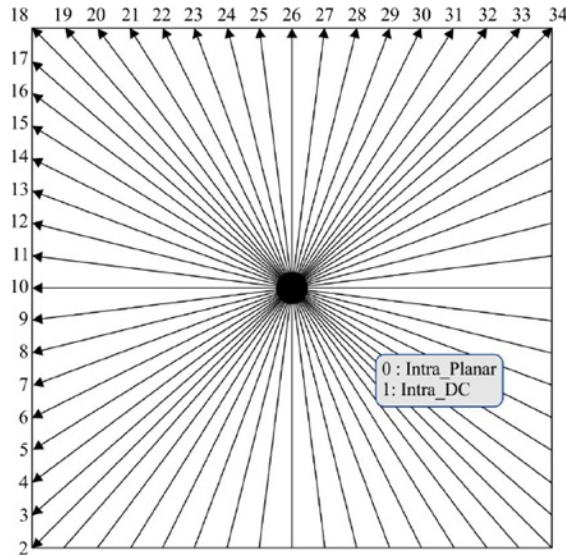
HEVC标准中的帧内预测和DCT反变换都是以TU为单位进行的，因此将这两部分知识放到一起记录。

帧内预测知识

HEVC的帧内预测共有35中预测模式，如下表所示：

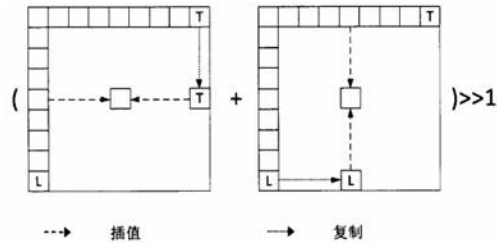
模式编号	模式名称
0	Planar
1	DC

其中第2-34种预测方式的角度如下所示。



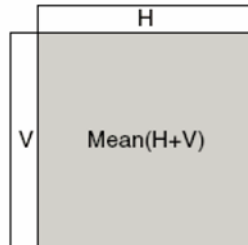
HEVC的角度预测方向相对于H.264增加到了33种。这样做的好处是能够更有效低表示图像的纹理特征，提高预测精度。其中编号2到17的角度预测模式为水平类模式，编号为18到34的角度预测模式为垂直类模式。编号为10的为水平预测，编号为26的为垂直预测模式。

Planar模式的计算方式如下图所示。



从图中可以看出，Planar模式首先将左边一列像素最下面一个像素值水平复制一行，将上边一行像素最右边一个像素值垂直复制一列；然后使用类似于双线性插值的方式，获得预测数据。这一预测方式综合了水平和垂直预测的特点。

DC模式的计算方法如下图所示。



从图中可以看出，DC模式计算方式原理很简单：直接将当前块上方一行以及左边一列像素求得平均值后，赋值给当前块中的每一个像素。

DCT变换

H.264中采用了4x4整数DCT变换，在HEVC中沿用了这种整数变换方法，但是其主要有以下几点不同：

(1) 变换尺寸不再限于4x4，而是包括了4x4，8x8，16x16，32x32几种方式。

(2) 变换系数值变大了很多，这样使得整数DCT的结果更接近浮点DCT的结果。注意在变换完成后会乘以修正矩阵（对于4x4变换来说，统一乘以1/128；对于尺寸N，修正系数值为 $1/(64 \cdot \sqrt{N})$ ）将放大后的结果修正回来。

(3) 在Intra4x4亮度残差变换的时候使用了一种比较特殊的4x4DST（离散正弦变换，中间的“S”代表“sin()”），在后文会记录该种变换。

HEVC支持最大为32x32的DCT变换。该变换矩阵的系数值如下图所示。其中第一张图为左边的16列数值，第二张图为右边的16列数值。

64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64
90	90	88	85	82	78	73	67	61	54	46	38	31	22	13	4	
90	87	80	70	57	43	25	9	-9	-25	-43	-57	-70	-80	-87	-90	
90	82	67	46	22	-4	-31	-54	-73	-85	-90	-88	-78	-61	-38	-13	
89	75	50	18	-18	-50	-75	-89	-89	-75	-50	-18	18	50	75	89	
88	67	31	-13	-54	-82	-90	-78	-46	-4	38	73	90	85	61	22	
87	57	9	-43	-80	-90	-70	-25	25	70	90	80	43	-9	-57	-87	
85	46	-13	-67	-90	-73	-22	38	82	88	54	-4	-61	-90	-78	-31	
83	36	-36	-83	-83	-36	36	83	83	36	-36	-83	-83	-36	36	83	
82	22	-54	-90	-61	13	78	85	31	-46	-90	-67	4	73	88	38	
80	9	-70	-87	-25	57	90	43	-43	-90	-57	25	87	70	-9	-80	
78	-4	-82	-73	13	85	67	-22	-88	-61	31	90	54	-38	-90	-46	
75	-18	-89	-50	50	89	18	-75	-75	18	89	50	-50	-89	-18	75	
73	-31	-90	-22	78	67	-38	-90	-13	82	61	-46	-88	-4	85	54	
70	-43	-87	9	90	25	-80	-57	57	80	-25	-90	-9	87	43	-70	
67	-54	-78	38	85	-22	-90	4	90	13	-88	-31	82	46	-73	-61	
64	-64	-64	64	64	-64	-64	64	64	-64	-64	64	64	-64	-64	64	
61	-73	-46	82	31	-88	-13	90	-4	-90	22	85	-38	-78	54	67	
57	-80	-25	90	-9	-87	43	70	-70	-43	87	9	-90	25	80	-57	
54	-85	-4	88	-46	-61	82	13	-90	38	67	-78	-22	90	-31	-73	
50	-89	18	75	-75	-18	89	-50	-50	89	-18	-75	75	18	-89	50	
46	-90	38	54	-90	31	61	-88	22	67	-85	13	73	-82	4	78	
43	-90	57	25	-87	70	9	-80	80	-9	-70	87	-25	-57	90	-43	
38	-88	73	-4	-67	90	-46	-31	85	-78	13	61	-90	54	22	-82	
36	-83	83	-36	-36	83	-83	36	36	-83	83	-36	-36	83	-83	36	
31	-78	90	-61	4	54	-88	82	-38	-22	73	-90	67	-13	-46	85	
25	-70	90	-80	43	9	-57	87	-87	57	-9	-43	80	-90	70	-25	
22	-61	85	-90	73	-38	-4	46	-78	90	-82	54	-13	-31	67	-88	
18	-50	75	-89	89	-75	50	-18	-18	50	-75	89	-89	75	-50	18	
13	-38	61	-78	88	-90	85	-73	54	-31	4	22	-46	67	-82	90	
9	-25	43	-57	70	-80	87	-90	90	-87	80	-70	57	-43	25	-9	
4	-13	22	-31	38	-46	54	-61	67	-73	78	-82	85	-88	90	-90	

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

64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64
-4	-13	-22	-31	-38	-46	-54	-61	-67	-73	-78	-82	-85	-88	-90	-90	
-90	-87	-80	-70	-57	-43	-25	-9	9	25	43	57	70	80	87	90	
13	38	61	78	88	90	85	73	54	31	4	-22	-46	-67	-82	-90	
89	75	50	18	-18	-50	-75	-89	-89	-75	-50	-18	18	50	75	89	
-22	-61	-85	-90	-73	-38	4	46	78	90	82	54	13	-31	-67	-88	
-87	-57	-9	43	80	90	70	25	-25	-70	-90	-80	-43	9	57	87	
31	78	90	61	4	-54	-88	-82	-38	22	73	90	67	13	-46	-85	
83	36	-36	-83	-83	-36	36	83	83	36	-36	-83	-83	-36	36	83	
-38	-88	-73	-4	67	90	46	-31	-85	-78	-13	61	90	54	-22	-82	
-80	-9	70	87	25	-57	-90	-43	43	90	57	-25	-87	-70	9	80	
46	90	38	-54	-90	-31	61	88	22	-67	-85	-13	73	82	4	-78	
75	-18	-89	-50	50	89	18	-75	-75	18	89	50	-50	-89	-18	75	
-54	-85	4	88	46	-61	-82	13	90	38	-67	-78	22	90	31	-73	
-70	43	87	-9	-90	-25	80	57	-57	-80	25	90	9	-87	-43	70	
61	73	-46	-82	31	88	-13	-90	-4	90	22	-85	-38	78	54	-67	
64	-64	-64	64	64	-64	-64	64	64	-64	-64	64	64	-64	-64	64	
-67	-54	78	38	-85	-22	90	4	-90	13	88	-31	-82	46	73	-61	
-57	80	25	-90	9	87	-43	-70	70	43	-87	-9	90	-25	-80	57	
73	31	-90	22	78	-67	-38	90	-13	-82	61	46	-88	4	85	-54	
50	-89	18	75	-75	-18	89	-50	-50	89	-18	-75	75	18	-89	50	
-78	-4	82	-73	-13	85	-67	-22	88	-61	-31	90	-54	-38	90	-46	
-43	90	-57	-25	87	-70	-9	80	-80	9	70	-87	25	57	-90	43	
82	-22	-54	90	-61	-13	78	-85	31	46	-90	67	4	-73	88	-38	
36	-83	83	-36	-36	83	-83	36	36	-83	83	-36	-36	83	-83	36	
-85	46	13	-67	90	-73	22	38	-82	88	-54	-4	61	-90	78	-31	
-25	70	-90	80	-43	-9	57	-87	87	-57	9	43	-80	90	-70	25	
88	-67	31	13	-54	82	-90	78	-46	4	38	-73	90	-85	61	-22	
18	-50	75	-89	89	-75	50	-18	-18	50	-75	89	-89	75	-50	18	
-90	82	-67	46	-22	-4	31	-54	73	-85	90	-88	78	-61	38	-13	
-9	25	-43	57	-70	80	-87	90	-90	87	-80	70	-57	43	-25	9	
90	-90	88	-85	82	-78	73	-67	61	-54	46	-38	31	-22	13	-4	

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

4x4DCT变换的系数来自于为32x32系数矩阵中第0, 8, 16, 24行元素中的前4个元素, 在图中以红色方框表示出来。由此可知4x4DCT系数矩阵为:

64 64 64 64
83 36 -36 -83
64 -64 -64 64
36 -83 83 -36

8x8DCT变换的系数来自于32x32系数矩阵中第0, 4, 8, 12, 16, 20, 24, 28行元素中的前8个元素, 在图中以黄色方框表示出来。由此可知8x8 DCT系数矩阵为:

64 64 64 64 64 64 64 64
89 75 50 18 -18 -50 -75 -89
83 36 -36 -83 -83 -36 36 83
75 -18 -89 -50 50 89 18 -75
64 -64 -64 64 64 -64 -64 64
50 -89 18 75 -75 -18 89 -50
36 -83 83 -36 -36 83 -83 36
18 -50 75 -89 89 -75 50 -18

16x16 DCT变换的系数来自于32x32系数矩阵中第0,2,4...,28,30行元素中的前16个元素,在图中以绿色方框表示出来。由于系数数量较大,就不再列出了。

在编码Intra4x4的残差数据的时候,使用了一种比较特殊的4x4DST。该种变换的系数矩阵如下所示。相关的实验表明,在编码Intra4x4的时候使用4x4DST可以提升约0.8%的编码效率。

29	55	74	84
74	74	0	-74
84	-29	-74	55
55	-84	74	-29

帧内预测实例

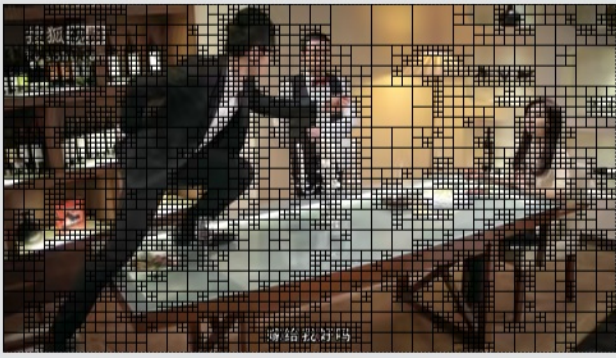
本节以一小段视频的码流为例,看一下HEVC码流中的帧内预测相关的信息。

【示例1】

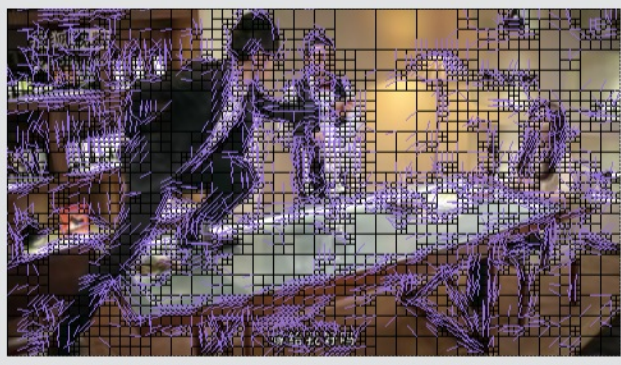
下图为一个I帧解码后的图像。



下图为该帧CTU的划分方式。可以看出画面复杂的地方CTU划分比较细。



下图的蓝色线条显示了帧内预测的方向。



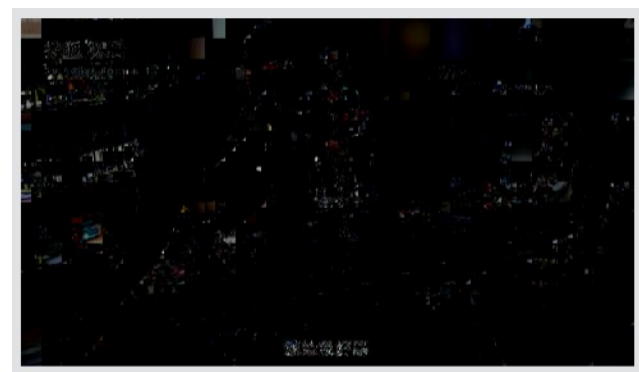
下图显示了帧内预测方向与图像内容之间的关系。可以看出帧内预测方向基本上和图像纹理方向是一致的。



下图为经过帧内预测，没有经过残差叠加处理的视频内容。

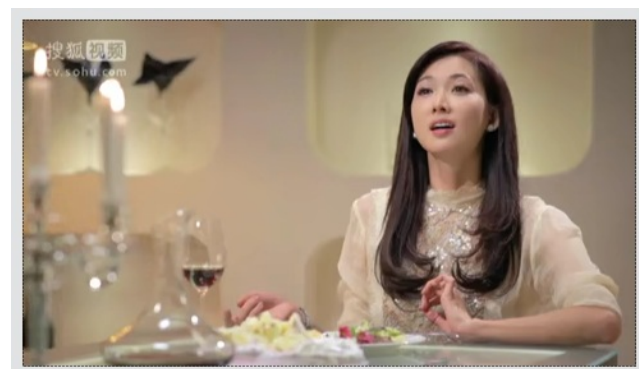


下图为该帧的残差信息。

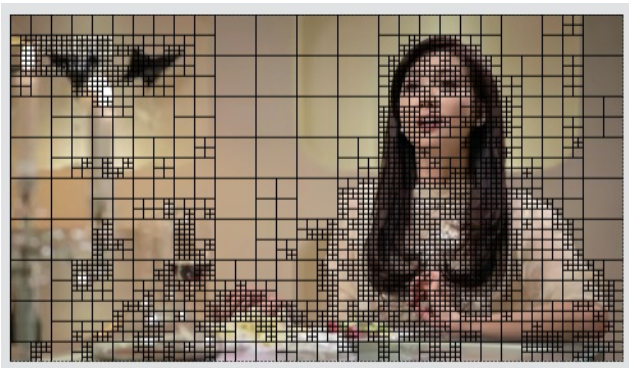


【示例2】

下图为一个I帧解码后的图像。



下图为该帧CTU的划分方式。



下图的蓝色线条显示了帧内预测的方向。



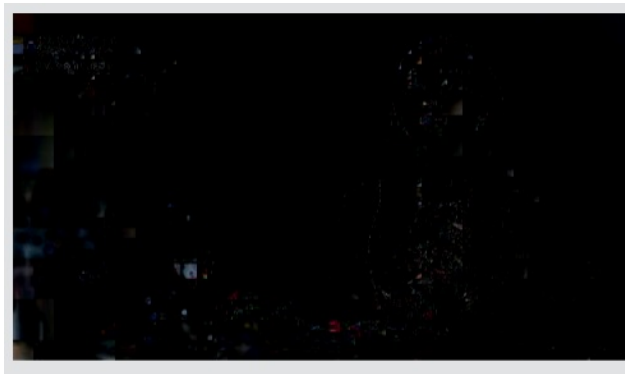
下图显示了帧内预测方向与图像内容之间的关系。



下图为经过帧内预测，没有经过残差叠加处理的视频内容。

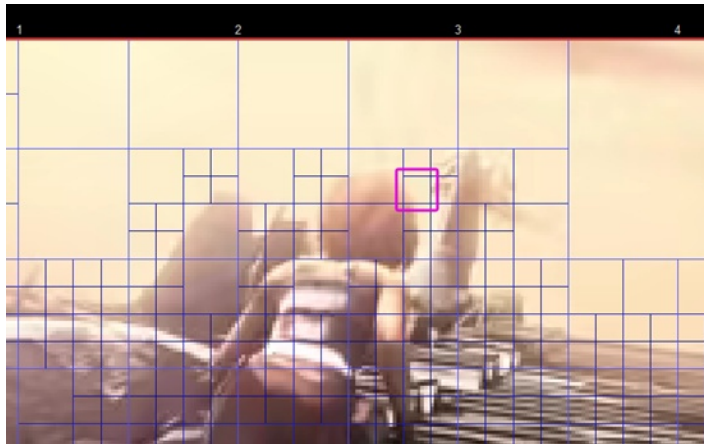


下图为该帧的残差信息。

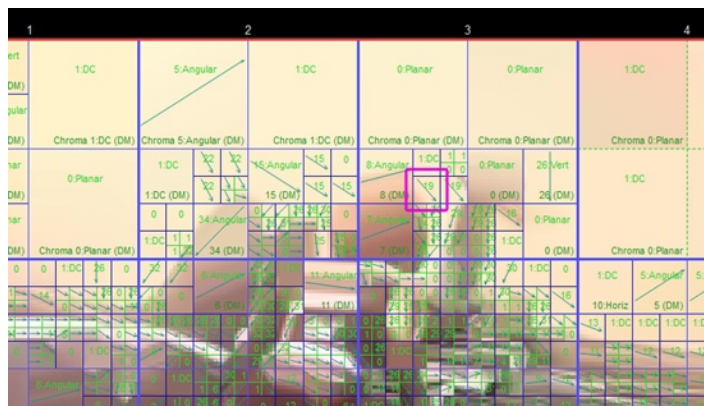


【示例3-帧内滤波信息】

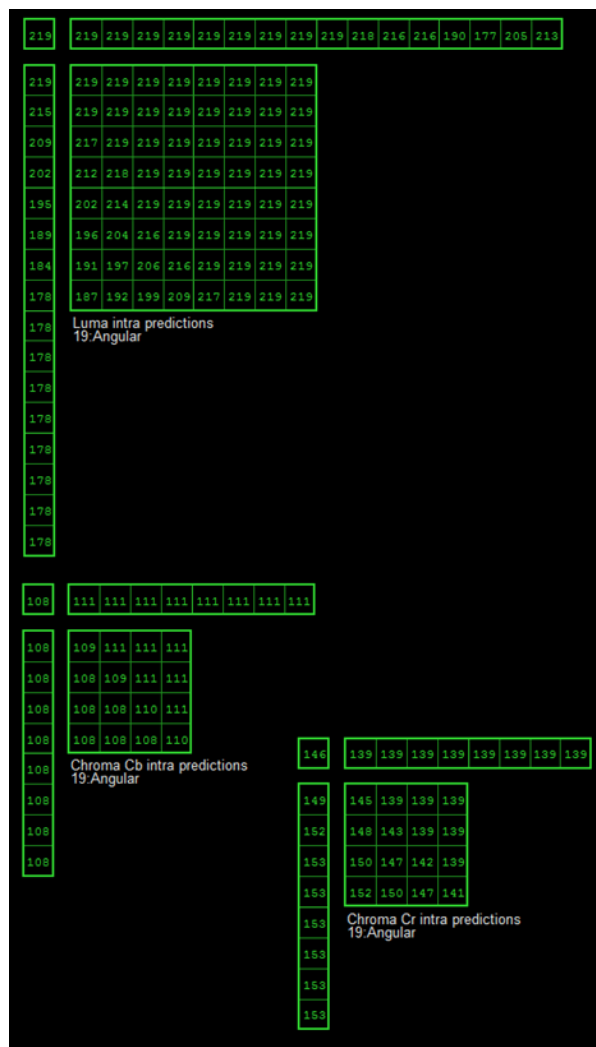
本节以一段《Sintel》动画的码流为例，看一下HEVC码流中的帧内滤波具体的信息。下图为I帧解码后的图像。



下图为没有叠加残差数据的帧内预测的结果。在这里我们选择一个8x8 CU（图中以紫色方框标出）看一下其中具体的信息。该CU采用了19号帧内预测模式（属于角度Angular预测模式）。

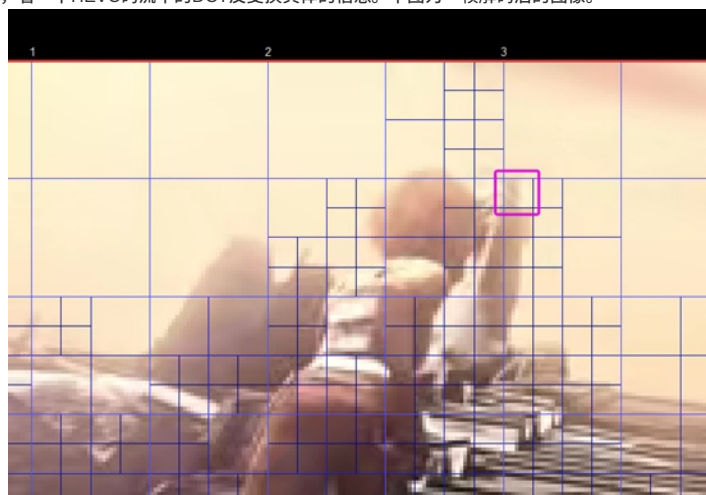


该8x8 CU的帧内预测信息如下图所示。

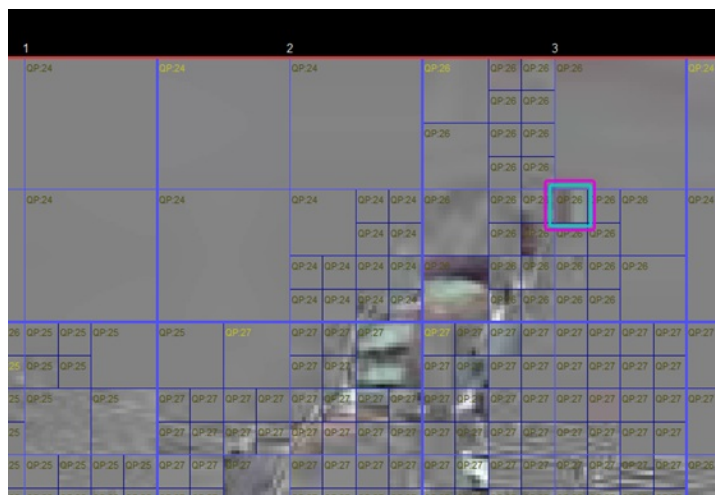


【示例4-DCT反变换示例】

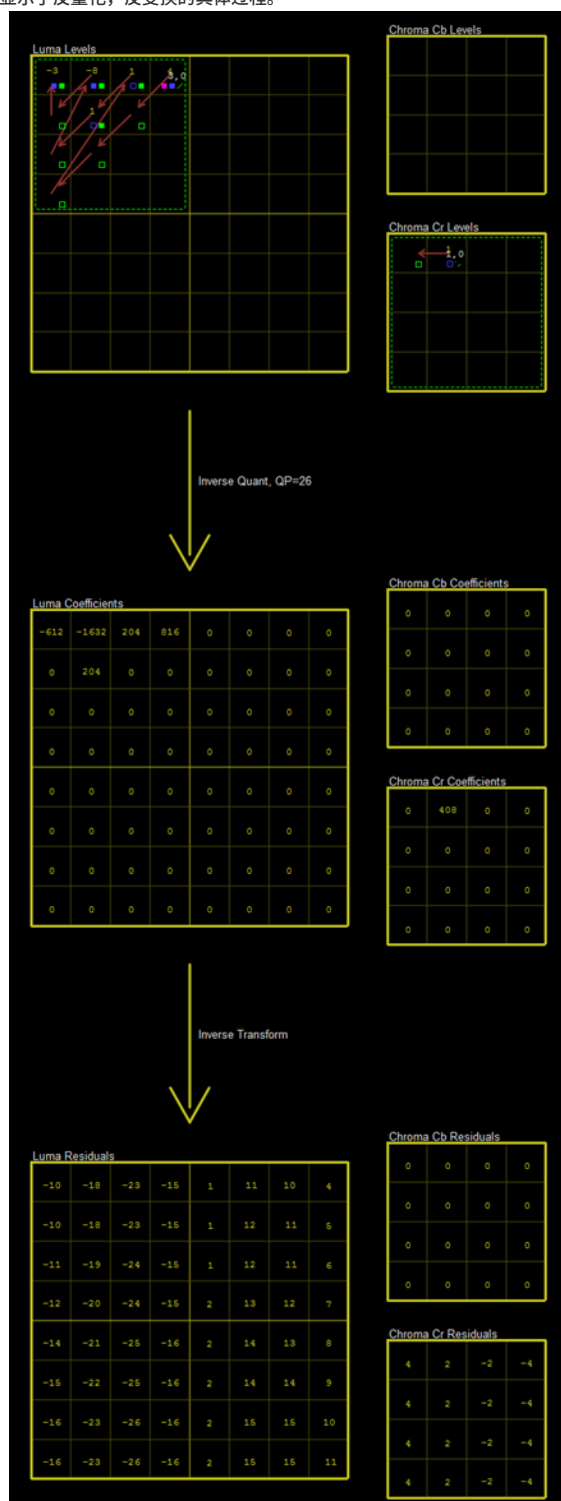
本节还是以《Sintel》动画的码流为例，看一下HEVC码流中的DCT反变换具体的信息。下图为一帧解码后的图像。



下图为该帧图像的残差数据。在这里我们选择一个8x8 CU（图中以紫色方框标出）看一下其中具体的信息。



该8x8 CU的DCT反变换信息如下图所示。图中显示了反量化，反变换的具体过程。





帧内预测汇编函数源代码


帧内预测相关的汇编函数位于HEVCPredContext中。HEVCPredContext的初始化函数是ff_hevc_pred_init()。该函数对HEVCPredContext结构体中的函数指针进行了赋值。FFmpeg HEVC解码器运行的过程中只要调用HEVCPredContext的函数指针就可以完成相应的功能。

ff_hevc_pred_init()



ff_hevc_pred_init()用于初始化HEVCPredContext结构体中的汇编函数指针。该函数的定义如下所示。

```
[cpp]    
1. //帧内预测函数初始化  
2. void ff_hevc_pred_init(HEVCPredContext *hpc, int bit_depth)  
3. {  
4. #undef FUNC  
5. #define FUNC(a, depth) a ## _ ## depth  
6.  
7. #define HEVC_PRED(depth) \\\br/>8.     hpc->intra_pred[0] = FUNC(intra_pred_2, depth); \\\br/>9.     hpc->intra_pred[1] = FUNC(intra_pred_3, depth); \\\br/>10.    hpc->intra_pred[2] = FUNC(intra_pred_4, depth); \\\br/>11.    hpc->intra_pred[3] = FUNC(intra_pred_5, depth); \\\br/>12.    hpc->pred_planar[0] = FUNC(pred_planar_0, depth); \\\br/>13.    hpc->pred_planar[1] = FUNC(pred_planar_1, depth); \\\br/>14.    hpc->pred_planar[2] = FUNC(pred_planar_2, depth); \\\br/>15.    hpc->pred_planar[3] = FUNC(pred_planar_3, depth); \\\br/>16.    hpc->pred_dc = FUNC(pred_dc, depth); \\\br/>17.    hpc->pred_angular[0] = FUNC(pred_angular_0, depth); \\\br/>18.    hpc->pred_angular[1] = FUNC(pred_angular_1, depth); \\\br/>19.    hpc->pred_angular[2] = FUNC(pred_angular_2, depth); \\\br/>20.    hpc->pred_angular[3] = FUNC(pred_angular_3, depth);  
21.  
22.    switch (bit_depth) {  
23.    case 9:  
24.        HEVC_PRED(9);  
25.        break;  
26.    case 10:  
27.        HEVC_PRED(10);  
28.        break;  
29.    case 12:  
30.        HEVC_PRED(12);  
31.        break;  
32.    default:  
33.        HEVC_PRED(8);  
34.        break;  
35.    }  
36. }
```

从源代码可以看出，ff_hevc_pred_init()函数中包含一个名为“HEVC_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_hevc_dsp_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。下面以8bit颜色位深为例，看一下“HEVC_PRED(8)”的展开结果。

```
[cpp]    
1. hpc->intra_pred[0] = intra_pred_2_8;  
2. hpc->intra_pred[1] = intra_pred_3_8;  
3. hpc->intra_pred[2] = intra_pred_4_8;  
4. hpc->intra_pred[3] = intra_pred_5_8;  
5. hpc->pred_planar[0] = pred_planar_0_8;  
6. hpc->pred_planar[1] = pred_planar_1_8;  
7. hpc->pred_planar[2] = pred_planar_2_8;  
8. hpc->pred_planar[3] = pred_planar_3_8;  
9. hpc->pred_dc = pred_dc_8;  
10. hpc->pred_angular[0] = pred_angular_0_8;  
11. hpc->pred_angular[1] = pred_angular_1_8;  
12. hpc->pred_angular[2] = pred_angular_2_8;  
13. hpc->pred_angular[3] = pred_angular_3_8;
```

可以看出“HEVC_PRED(8)”初始化了帧内预测模块的C语言版本函数。HEVCPredContext的定义如下。

```
[cpp]    
1. typedef struct HEVCPredContext {  
2.     void (*intra_pred[4])(struct HEVCContext *s, int x0, int y0, int c_idx);  
3.  
4.     void (*pred_planar[4])(uint8_t *src, const uint8_t *top,  
5.         const uint8_t *left, ptrdiff_t stride);  
6.     void (*pred_dc)(uint8_t *src, const uint8_t *top, const uint8_t *left,  
7.         ptrdiff_t stride, int log2_size, int c_idx);  
8.     void (*pred_angular[4])(uint8_t *src, const uint8_t *top,  
9.         const uint8_t *left, ptrdiff_t stride,  
10.         int c_idx, int mode);  
11. } HEVCPredContext;
```

从源代码中可以看出，HEVCPredContext中存储了4个汇编函数指针（数组）：

intra_pred[4]()：帧内预测的入口函数，该函数执行过程中调用了后面3个函数指针。数组中4个函数分别处理4x4，8x8，16x16，32x32几种块。

pred_planar[4]()：Planar预测模式函数。数组中4个函数分别处理4x4，8x8，16x16，32x32几种块。

pred_dc()：DC预测模式函数。

pred_angular[4]()：角度预测模式。数组中4个函数分别处理4x4，8x8，16x16，32x32几种块。

下文按照顺序分别介绍这几种函数。

HEVCPredContext ->intra_pred[4]()

intra_pred[4]()是帧内预测的入口函数，该函数执行过程中调用了Planar、DC或者角度预测函数。数组中4个元素分别处理4x4，8x8，16x16，32x32几种块。这几种块的具体处理函数为：

intra_pred_2_8()——4x4块

intra_pred_3_8()——8x8块

intra_pred_4_8()——16x16块

intra_pred_5_8()——32x32块

PS：函数命名时候中间的数字是块的边长取log2()之后的数值。

上面这几个函数的定义如下所示。

```
[cpp]
1. #define INTRA_PRED(size) \
2. static void FUNC(intra_pred_ ## size)(HEVCContext *s, int x0, int y0, int c_idx) \
3. { \
4.     FUNC(intra_pred)(s, x0, y0, size, c_idx); \
5. }
6.
7. /* 几种不同大小的方块对应的帧内预测函数
8.  * 参数是方块像素数取对数之后的值
9.  * 例如“INTRA_PRED(2)”即为4x4块的帧内预测函数
10.  *
11.  * “INTRA_PRED(2)”展开后的函数是
12.  * static void intra_pred_2_8(HEVCContext *s, int x0, int y0, int c_idx)
13.  * {
14.  *     intra_pred_8(s, x0, y0, 2, c_idx);
15.  * }
16.  */
17. INTRA_PRED(2)
18. INTRA_PRED(3)
19. INTRA_PRED(4)
20. INTRA_PRED(5)
```

从源代码中可以看出，intra_pred_2_8()、intra_pred_3_8()等函数都是通过“INTRA_PRED()”宏进行定义的。intra_pred_2_8()、intra_pred_3_8()的函数的内部都调用了同一个函数intra_pred_8()。这几个函数唯一的不同在于，调用intra_pred_8()时候第4个参数size的值不一样。

intra_pred_8()

intra_pred_8()完成了帧内预测前的滤波等准备工作，并根据帧内预测类型的不同（Planar、DC、角度）调用不同的帧内预测函数。该函数的定义如下所示。

```
[cpp]
1. static av_always_inline void FUNC(intra_pred)(HEVCContext *s, int x0, int y0,
2.                                             int log2_size, int c_idx)
3. {
4.     #define PU(x) \
5.         ((x) >> s->sps->log2_min_pu_size)
6.     #define MVF(x, y) \
7.         (s->ref->tab_mvf[(x) + (y) * min_pu_width])
8.     #define MVF_PU(x, y) \
9.         MVF(PU(x0 + ((x) << hshift)), PU(y0 + ((y) << vshift)))
10.    #define IS_INTRA(x, y) \
11.        (MVF_PU(x, y).pred_flag == PF_INTRA)
12.    #define MIN_TB_ADDR_ZS(x, y) \
13.        s->pps->min_tb_addr_zs[(y) * (s->sps->tb_mask+2) + (x)]
14.    #define EXTEND(ptr, val, len) \
15.        do { \
16.            pixel4 pix = PIXEL_SPLAT_X4(val); \
17.            for (i = 0; i < (len); i += 4) \
18.                AV_WN4P(ptr + i, pix); \
19.        } while (0)
20.
21.    #define EXTEND_RIGHT_CIP(ptr, start, length) \
22.        for (i = start; i < (start) + (length); i += 4) \
23.            if (!IS_INTRA(i, -1)) \
24.                AV_WN4P(&ptr[i], a); \
25.            else \
26.                a = PIXEL_SPLAT_X4(ptr[i+3])
27.    #define EXTEND_LEFT_CIP(ptr, start, length) \
28.        for (i = start; i > (start) - (length); i--) \
29.            if (!IS_INTRA(i - 1, -1)) \
30.                ptr[i - 1] = ptr[i]
31.    #define EXTEND_UP_CIP(ptr, start, length) \
32.        for (i = (start); i > (start) - (length); i -= 4) \
33.            if (!IS_INTRA(-1, i - 3)) \
34.                AV_WN4P(&ptr[i - 3], a); \
```

```

35.         else \
36.             a = PIXEL_SPLAT_X4(ptr[i - 3])
37. #define EXTEND_DOWN_CIP(ptr, start, length) \
38.     for (i = start; i < (start) + (length); i += 4) \
39.         if (!IS_INTRA(-1, i)) \
40.             AV_WN4P(&ptr[i], a); \
41.         else \
42.             a = PIXEL_SPLAT_X4(ptr[i + 3])
43.
44.     HEVCLocalContext *lc = s->HEVCLc;
45.     int i;
46.     int hshift = s->sps->hshift[c_idx];
47.     int vshift = s->sps->vshift[c_idx];
48.     int size = (1 << log2_size);
49.     int size_in_luma_h = size << hshift;
50.     int size_in_tbs_h = size_in_luma_h >> s->sps->log2_min_tb_size;
51.     int size_in_luma_v = size << vshift;
52.     int size_in_tbs_v = size_in_luma_v >> s->sps->log2_min_tb_size;
53.     int x = x0 >> hshift;
54.     int y = y0 >> vshift;
55.     int x_tb = (x0 >> s->sps->log2_min_tb_size) & s->sps->tb_mask;
56.     int y_tb = (y0 >> s->sps->log2_min_tb_size) & s->sps->tb_mask;
57.
58.     int cur_tb_addr = MIN_TB_ADDR_ZS(x_tb, y_tb);
59.     //注意c_idx标志了颜色分量
60.     ptrdiff_t stride = s->frame->linesize[c_idx] / sizeof(pixel);
61.     pixel *src = (pixel*)s->frame->data[c_idx] + x + y * stride;
62.
63.     int min_pu_width = s->sps->min_pu_width;
64.
65.     enum IntraPredMode mode = c_idx ? lc->tu.intra_pred_mode_c :
66.                                     lc->tu.intra_pred_mode;
67.     pixel4 a;
68.     pixel left_array[2 * MAX_TB_SIZE + 1];
69.     pixel filtered_left_array[2 * MAX_TB_SIZE + 1];
70.     pixel top_array[2 * MAX_TB_SIZE + 1];
71.     pixel filtered_top_array[2 * MAX_TB_SIZE + 1];
72.
73.     pixel *left = left_array + 1;
74.     pixel *top = top_array + 1;
75.     pixel *filtered_left = filtered_left_array + 1;
76.     pixel *filtered_top = filtered_top_array + 1;
77.     int cand_bottom_left = lc->na.cand_bottom_left && cur_tb_addr > MIN_TB_ADDR_ZS(x_tb - 1, (y_tb + size_in_tbs_v) & s->sps->tb_ma
sk);
78.     int cand_left = lc->na.cand_left;
79.     int cand_up_left = lc->na.cand_up_left;
80.     int cand_up = lc->na.cand_up;
81.     int cand_up_right = lc->na.cand_up_right && cur_tb_addr > MIN_TB_ADDR_ZS((x_tb + size_in_tbs_h) & s->sps->tb_mask, y_tb -
1);
82.
83.     int bottom_left_size = (FFMIN(y0 + 2 * size_in_luma_v, s->sps->height) -
84.                             (y0 + size_in_luma_v)) >> vshift;
85.     int top_right_size = (FFMIN(x0 + 2 * size_in_luma_h, s->sps->width) -
86.                            (x0 + size_in_luma_h)) >> hshift;
87.
88.     if (s->pps->constrained_intra_pred_flag == 1) {
89.         int size_in_luma_pu_v = PU(size_in_luma_v);
90.         int size_in_luma_pu_h = PU(size_in_luma_h);
91.         int on_pu_edge_x = !(x0 & ((1 << s->sps->log2_min_pu_size) - 1));
92.         int on_pu_edge_y = !(y0 & ((1 << s->sps->log2_min_pu_size) - 1));
93.         if (!size_in_luma_pu_h)
94.             size_in_luma_pu_h++;
95.         if (cand_bottom_left == 1 && on_pu_edge_x) {
96.             int x_left_pu = PU(x0 - 1);
97.             int y_bottom_pu = PU(y0 + size_in_luma_v);
98.             int max = FFMIN(size_in_luma_pu_v, s->sps->min_pu_height - y_bottom_pu);
99.             cand_bottom_left = 0;
100.            for (i = 0; i < max; i += 2)
101.                cand_bottom_left |= (MVF(x_left_pu, y_bottom_pu + i).pred_flag == PF_INTRA);
102.        }
103.        if (cand_left == 1 && on_pu_edge_x) {
104.            int x_left_pu = PU(x0 - 1);
105.            int y_left_pu = PU(y0);
106.            int max = FFMIN(size_in_luma_pu_v, s->sps->min_pu_height - y_left_pu);
107.            cand_left = 0;
108.            for (i = 0; i < max; i += 2)
109.                cand_left |= (MVF(x_left_pu, y_left_pu + i).pred_flag == PF_INTRA);
110.        }
111.        if (cand_up_left == 1) {
112.            int x_left_pu = PU(x0 - 1);
113.            int y_top_pu = PU(y0 - 1);
114.            cand_up_left = MVF(x_left_pu, y_top_pu).pred_flag == PF_INTRA;
115.        }
116.        if (cand_up == 1 && on_pu_edge_y) {
117.            int x_top_pu = PU(x0);
118.            int y_top_pu = PU(y0 - 1);
119.            int max = FFMIN(size_in_luma_pu_h, s->sps->min_pu_width - x_top_pu);
120.            cand_up = 0;
121.            for (i = 0; i < max; i += 2)
122.                cand_up |= (MVF(x_top_pu + i, y_top_pu).pred_flag == PF_INTRA);
123.        }

```

```

124.     if (cand_up_right == 1 && on_pu_edge_y) {
125.         int y_top_pu = PU(y0 - 1);
126.         int x_right_pu = PU(x0 + size_in_luma_h);
127.         int max = FFMIN(size_in_luma_pu_h, s->sps->min_pu_width - x_right_pu);
128.         cand_up_right = 0;
129.         for (i = 0; i < max; i += 2)
130.             cand_up_right |= (MV(x_right_pu + i, y_top_pu).pred_flag == PF_INTRA);
131.     }
132.     memset(left, 128, 2 * MAX_TB_SIZE * sizeof(pixel));
133.     memset(top, 128, 2 * MAX_TB_SIZE * sizeof(pixel));
134.     top[-1] = 128;
135. }
136. if (cand_up_left) {
137.     left[-1] = POS(-1, -1);
138.     top[-1] = left[-1];
139. }
140. if (cand_up)
141.     memcpy(top, src - stride, size * sizeof(pixel));
142. if (cand_up_right) {
143.     memcpy(top + size, src - stride + size, size * sizeof(pixel));
144.     EXTEND(top + size + top_right_size, POS(size + top_right_size - 1, -1),
145.           size - top_right_size);
146. }
147. if (cand_left)
148.     for (i = 0; i < size; i++)
149.         left[i] = POS(-1, i);
150. if (cand_bottom_left) {
151.     for (i = size; i < size + bottom_left_size; i++)
152.         left[i] = POS(-1, i);
153.     EXTEND(left + size + bottom_left_size, POS(-1, size + bottom_left_size - 1),
154.           size - bottom_left_size);
155. }
156.
157. if (s->pps->constrained_intra_pred_flag == 1) {
158.     if (cand_bottom_left || cand_left || cand_up_left || cand_up || cand_up_right) {
159.         int size_max_x = x0 + ((2 * size) << hshift) < s->sps->width ?
160.             2 * size : (s->sps->width - x0) >> hshift;
161.         int size_max_y = y0 + ((2 * size) << vshift) < s->sps->height ?
162.             2 * size : (s->sps->height - y0) >> vshift;
163.         int j = size + (cand_bottom_left ? bottom_left_size : 0) - 1;
164.         if (!cand_up_right) {
165.             size_max_x = x0 + ((size) << hshift) < s->sps->width ?
166.                 size : (s->sps->width - x0) >> hshift;
167.         }
168.         if (!cand_bottom_left) {
169.             size_max_y = y0 + ((size) << vshift) < s->sps->height ?
170.                 size : (s->sps->height - y0) >> vshift;
171.         }
172.         if (cand_bottom_left || cand_left || cand_up_left) {
173.             while (j > -1 && !IS_INTRA(-1, j))
174.                 j--;
175.             if (!IS_INTRA(-1, j)) {
176.                 j = 0;
177.                 while (j < size_max_x && !IS_INTRA(j, -1))
178.                     j++;
179.                 EXTEND_LEFT_CIP(top, j, j + 1);
180.                 left[-1] = top[-1];
181.             }
182.         } else {
183.             j = 0;
184.             while (j < size_max_x && !IS_INTRA(j, -1))
185.                 j++;
186.             if (j > 0)
187.                 if (x0 > 0) {
188.                     EXTEND_LEFT_CIP(top, j, j + 1);
189.                 } else {
190.                     EXTEND_LEFT_CIP(top, j, j);
191.                     top[-1] = top[0];
192.                 }
193.             left[-1] = top[-1];
194.         }
195.         left[-1] = top[-1];
196.         if (cand_bottom_left || cand_left) {
197.             a = PIXEL_SPLAT_X4(left[-1]);
198.             EXTEND_DOWN_CIP(left, 0, size_max_y);
199.         }
200.         if (!cand_left)
201.             EXTEND(left, left[-1], size);
202.         if (!cand_bottom_left)
203.             EXTEND(left + size, left[size - 1], size);
204.         if (x0 != 0 && y0 != 0) {
205.             a = PIXEL_SPLAT_X4(left[size_max_y - 1]);
206.             EXTEND_UP_CIP(left, size_max_y - 1, size_max_y);
207.             if (!IS_INTRA(-1, -1))
208.                 left[-1] = left[0];
209.         } else if (x0 == 0) {
210.             EXTEND(left, 0, size_max_y);
211.         } else {
212.             a = PIXEL_SPLAT_X4(left[size_max_y - 1]);
213.             EXTEND_UP_CIP(left, size_max_y - 1, size_max_y);
214.         }

```

```

215.         top[-1] = left[-1];
216.         if (y0 != 0) {
217.             a = PIXEL_SPLAT_X4(left[-1]);
218.             EXTEND_RIGHT_CIP(top, 0, size_max_x);
219.         }
220.     }
221. }
222. // Infer the unavailable samples
223. if (!cand_bottom_left) {
224.     if (cand_left) {
225.         EXTEND(left + size, left[size - 1], size);
226.     } else if (cand_up_left) {
227.         EXTEND(left, left[-1], 2 * size);
228.         cand_left = 1;
229.     } else if (cand_up) {
230.         left[-1] = top[0];
231.         EXTEND(left, left[-1], 2 * size);
232.         cand_up_left = 1;
233.         cand_left = 1;
234.     } else if (cand_up_right) {
235.         EXTEND(top, top[size], size);
236.         left[-1] = top[size];
237.         EXTEND(left, left[-1], 2 * size);
238.         cand_up = 1;
239.         cand_up_left = 1;
240.         cand_left = 1;
241.     } else { // No samples available
242.         left[-1] = (1 << (BIT_DEPTH - 1));
243.         EXTEND(top, left[-1], 2 * size);
244.         EXTEND(left, left[-1], 2 * size);
245.     }
246. }
247.
248. if (!cand_left)
249.     EXTEND(left, left[size], size);
250. if (!cand_up_left) {
251.     left[-1] = left[0];
252. }
253. if (!cand_up)
254.     EXTEND(top, left[-1], size);
255. if (!cand_up_right)
256.     EXTEND(top + size, top[size - 1], size);
257.
258. top[-1] = left[-1];
259.
260. // Filtering process
261. // 滤波
262. if (!s->sps->intra_smoothing_disabled_flag && (c_idx == 0 || s->sps->chroma_format_idc == 3)) {
263.     if (mode != INTRA_DC && size != 4) {
264.         int intra_hor_ver_dist_thresh[] = { 7, 1, 0 };
265.         int min_dist_vert_hor = FFMIN(FFABS((int)(mode - 26U)),
266.                                         FFABS((int)(mode - 10U)));
267.         if (min_dist_vert_hor > intra_hor_ver_dist_thresh[log2_size - 3]) {
268.             int threshold = 1 << (BIT_DEPTH - 5);
269.             if (s->sps->sps_strong_intra_smoothing_enable_flag && c_idx == 0 &&
270.                 log2_size == 5 &&
271.                 FFABS(top[-1] + top[63] - 2 * top[31]) < threshold &&
272.                 FFABS(left[-1] + left[63] - 2 * left[31]) < threshold) {
273.                 // We can't just overwrite values in top because it could be
274.                 // a pointer into src
275.                 filtered_top[-1] = top[-1];
276.                 filtered_top[63] = top[63];
277.                 for (i = 0; i < 63; i++)
278.                     filtered_top[i] = ((64 - (i + 1)) * top[-1] +
279.                                           (i + 1) * top[63] + 32) >> 6;
280.                 for (i = 0; i < 63; i++)
281.                     left[i] = ((64 - (i + 1)) * left[-1] +
282.                                 (i + 1) * left[63] + 32) >> 6;
283.                 top = filtered_top;
284.             } else {
285.                 filtered_left[2 * size - 1] = left[2 * size - 1];
286.                 filtered_top[2 * size - 1] = top[2 * size - 1];
287.                 for (i = 2 * size - 2; i >= 0; i--)
288.                     filtered_left[i] = (left[i + 1] + 2 * left[i] +
289.                                           left[i - 1] + 2) >> 2;
290.                 filtered_top[-1] =
291.                     filtered_left[-1] = (left[0] + 2 * left[-1] + top[0] + 2) >> 2;
292.                 for (i = 2 * size - 2; i >= 0; i--)
293.                     filtered_top[i] = (top[i + 1] + 2 * top[i] +
294.                                           top[i - 1] + 2) >> 2;
295.                 left = filtered_left;
296.                 top = filtered_top;
297.             }
298.         }
299.     }
300. }
301. /*
302. * 根据不同的帧内预测模式，调用不同的处理函数
303. * pred_planar[4], pred_angular[4]中的"[4]"代表了儿种不同大小的方块
304. * [0]:4x4块
305. * [1]:8x8块
306. * [2]:16x16块

```



```

306.     * [2]:16x16块
307.     * [3]:32x32块
308.     *
309.     * log2size为方块边长取对数。
310.     * 4x4块, log2size=log2(4)=2
311.     * 8x8块, log2size=log2(8)=3
312.     * 16x16块, log2size=log2(16)=4
313.     * 32x32块, log2size=log2(32)=5
314.     *
315.     */
316.     switch (mode) {
317.     case INTRA_PLANAR:
318.         s->hpc.pred_planar[log2_size - 2]((uint8_t *)src, (uint8_t *)top,
319.                                         (uint8_t *)left, stride);
320.         break;
321.     case INTRA_DC:
322.         s->hpc.pred_dc((uint8_t *)src, (uint8_t *)top,
323.                       (uint8_t *)left, stride, log2_size, c_idx);
324.         break;
325.     default:
326.         s->hpc.pred_angular[log2_size - 2]((uint8_t *)src, (uint8_t *)top,
327.                                           (uint8_t *)left, stride, c_idx,
328.                                           mode);
329.         break;
330.     }
331. }

```

intra_pred_8()前面部分的代码还没有细看，大致做了一些帧内预测的准备工作；它的后面有一个switch()语句，根据帧内预测模式的不同作不同的处理：

- (1) Planar模式，调用HEVCContext-> pred_planar()
- (2) DC模式，调用HEVCContext-> pred_dc()
- (3) 其他模式（剩余都是角度模式），调用HEVCContext-> pred_angular()

HEVC解码器中帧内预测模式的定义于IntraPredMode变量，如下所示。

```

1.  enum IntraPredMode {
2.      INTRA_PLANAR = 0,
3.      INTRA_DC,
4.      INTRA_ANGULAR_2,
5.      INTRA_ANGULAR_3,
6.      INTRA_ANGULAR_4,
7.      INTRA_ANGULAR_5,
8.      INTRA_ANGULAR_6,
9.      INTRA_ANGULAR_7,
10.     INTRA_ANGULAR_8,
11.     INTRA_ANGULAR_9,
12.     INTRA_ANGULAR_10,
13.     INTRA_ANGULAR_11,
14.     INTRA_ANGULAR_12,
15.     INTRA_ANGULAR_13,
16.     INTRA_ANGULAR_14,
17.     INTRA_ANGULAR_15,
18.     INTRA_ANGULAR_16,
19.     INTRA_ANGULAR_17,
20.     INTRA_ANGULAR_18,
21.     INTRA_ANGULAR_19,
22.     INTRA_ANGULAR_20,
23.     INTRA_ANGULAR_21,
24.     INTRA_ANGULAR_22,
25.     INTRA_ANGULAR_23,
26.     INTRA_ANGULAR_24,
27.     INTRA_ANGULAR_25,
28.     INTRA_ANGULAR_26,
29.     INTRA_ANGULAR_27,
30.     INTRA_ANGULAR_28,
31.     INTRA_ANGULAR_29,
32.     INTRA_ANGULAR_30,
33.     INTRA_ANGULAR_31,
34.     INTRA_ANGULAR_32,
35.     INTRA_ANGULAR_33,
36.     INTRA_ANGULAR_34,
37. };

```

下面分别看一下3种帧内预测函数。

HEVCPredContext -> pred_planar[4]()

HEVCPredContext -> pred_planar[4]()指向了帧内预测Planar模式的汇编函数。数组中4个元素分别处理4x4，8x8，16x16，32x32几种块。这几种块的具体C语言版本处理函数为：

```

    pred_planar_0_8()——4x4块；
    pred_planar_1_8()——8x8块；
    pred_planar_2_8()——16x16块；
    pred_planar_3_8()——32x32块；

```

这四个函数的定义如下所示。

```
[cpp]
1. #define PRED_PLANAR(size)\
2. static void FUNC(pred_planar_ ## size)(uint8_t *src, const uint8_t *top, \
3.                                     const uint8_t *left, ptrdiff_t stride) \
4. { \
5.     FUNC(pred_planar)(src, top, left, stride, size + 2); \
6. }
7. /* 几种不同大小的方块对应的Planar预测函数
8.  * 参数取值越大，代表的方块越大：
9.  * [0]:4x4块
10. * [1]:8x8块
11. * [2]:16x16块
12. * [3]:32x32块
13. *
14. * "PRED_PLANAR(0)"展开后的函数是
15. * static void pred_planar_0_8(uint8_t *src, const uint8_t *top,
16. *                             const uint8_t *left, ptrdiff_t stride)
17. * {
18. *     pred_planar_8(src, top, left, stride, 0 + 2);
19. * }
20. */
21. PRED_PLANAR(0)
22. PRED_PLANAR(1)
23. PRED_PLANAR(2)
24. PRED_PLANAR(3)
```

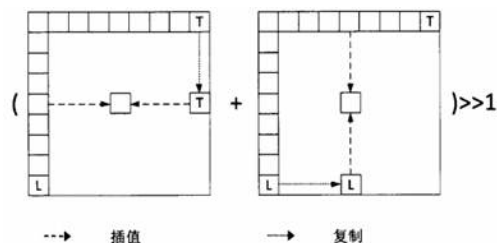
从源代码中可以看出，pred_planar_0_8()、pred_planar_1_8()等函数都是通过“PRED_PLANAR ()”宏进行定义的。pred_planar_0_8()、pred_planar_1_8()等函数的内部都调用了同一个函数pred_planar_8()。这几个函数唯一的不同在于，调用pred_planar_8()时候第5个参数trafo_size的值不一样。

pred_planar_8()

pred_planar_8()实现了Planar帧内预测模式，该函数的定义如下所示。

```
[cpp]
1. #define POS(x, y) src[(x) + stride * (y)]
2.
3. //Planar预测模式
4. static av_always_inline void FUNC(pred_planar)(uint8_t *src, const uint8_t *top,
5.                                     const uint8_t *left, ptrdiff_t stride,
6.                                     int trafo_size)
7. {
8.     int x, y;
9.     pixel *src = (pixel *)_src;
10.    //上面1行像素
11.    const pixel *top = (const pixel *)_top;
12.    //左边1列像素
13.    const pixel *left = (const pixel *)_left;
14.    int size = 1 << trafo_size;
15.    //双线性插值
16.    //注意[size]为最后一个元素
17.    for (y = 0; y < size; y++)
18.        for (x = 0; x < size; x++)
19.            POS(x, y) = ((size - 1 - x) * left[y] + (x + 1) * top[size] +
20.                        (size - 1 - y) * top[x] + (y + 1) * left[size] + size) >> (trafo_size + 1);
21. }
```

从源代码可以看出，pred_planar_8()以一种类似双线性插值的方式完成了预测数据的填充。其中src指向方块的像素区域，left指向方块左边一列像素，top指向方块上边一行像素。Planar模式的计算方式如下图所示。



从图中可以看出，Planar模式首先将左边一列像素最下面一个像素值水平复制一行，将上边一行像素最右边一个像素值垂直复制一列；然后使用类似于双线性插值的方式，获得预测数据。

HEVCPredContext -> pred_dc ()

HEVCPredContext -> pred_dc()指向了帧内预测DC模式的汇编函数。具体的C语言版本的处理函数是pred_dc_8()。pred_dc_8()的定义如下。

```

1. #define POS(x, y) src[(x) + stride * (y)]
2.
3. //DC预测模式
4. static void FUNC(pred_dc)(uint8_t *_src, const uint8_t *_top,
5.                          const uint8_t *_left,
6.                          ptrdiff_t stride, int log2_size, int c_idx)
7. {
8.     int i, j, x, y;
9.     int size      = (1 << log2_size);
10.    pixel *_src    = (pixel *)_src;
11.    const pixel *_top = (const pixel *)_top;
12.    const pixel *_left = (const pixel *)_left;
13.    int dc         = size;
14.    //pixel4为uint32_t, 即存储了4个像素
15.    pixel4 a;
16.    //累加左边1列, 和上边1行
17.    for (i = 0; i < size; i++)
18.        dc += left[i] + top[i];
19.    //求平均
20.    dc >>= log2_size + 1;
21.    //取出来值
22.    a = PIXEL_SPLAT_X4(dc);
23.    //赋值到像素块中的每个点
24.    for (i = 0; i < size; i++)
25.        for (j = 0; j < size; j+=4)
26.            AV_WN4P(&POS(j, i), a);
27.
28.    if (c_idx == 0 && size < 32) {
29.        POS(0, 0) = (left[0] + 2 * dc + top[0] + 2) >> 2;
30.        for (x = 1; x < size; x++)
31.            POS(x, 0) = (top[x] + 3 * dc + 2) >> 2;
32.        for (y = 1; y < size; y++)
33.            POS(0, y) = (left[y] + 3 * dc + 2) >> 2;
34.    }
35. }

```

从源代码可以看出，pred_dc_8()首先求得了左边一行像素和上边一行像素的平均值，然后将该值作为预测数据赋值给了整个方块。

HEVCPredContext -> pred_angular ()

HEVCPredContext -> pred_angular[4]()指向了帧内预测角度（Angular）模式的汇编函数。数组中4个元素分别处理4x4，8x8，16x16，32x32几种块。这几种块的具体C语言版本处理函数为：

pred_angular_0_8()——4x4块；
 pred_angular_1_8()——8x8块；
 pred_angular_2_8()——16x16块；
 pred_angular_3_8()——32x32块；

这四个函数的定义如下所示。

```

1.  /* 几种不同大小的方块对应的Angular预测函数
2.  * 数字取值越大，代表的方块越大：
3.  * [0]:4x4块
4.  * [1]:8x8块
5.  * [2]:16x16块
6.  * [3]:32x32块
7.  *
8.  */
9.  static void FUNC(pred_angular_0)(uint8_t *src, const uint8_t *top,
10.                                const uint8_t *left,
11.                                ptrdiff_t stride, int c_idx, int mode)
12.  {
13.      FUNC(pred_angular)(src, top, left, stride, c_idx, mode, 1 << 2);
14.  }
15.
16.  static void FUNC(pred_angular_1)(uint8_t *src, const uint8_t *top,
17.                                const uint8_t *left,
18.                                ptrdiff_t stride, int c_idx, int mode)
19.  {
20.      FUNC(pred_angular)(src, top, left, stride, c_idx, mode, 1 << 3);
21.  }
22.
23.  static void FUNC(pred_angular_2)(uint8_t *src, const uint8_t *top,
24.                                const uint8_t *left,
25.                                ptrdiff_t stride, int c_idx, int mode)
26.  {
27.      FUNC(pred_angular)(src, top, left, stride, c_idx, mode, 1 << 4);
28.  }
29.
30.  static void FUNC(pred_angular_3)(uint8_t *src, const uint8_t *top,
31.                                const uint8_t *left,
32.                                ptrdiff_t stride, int c_idx, int mode)
33.  {
34.      FUNC(pred_angular)(src, top, left, stride, c_idx, mode, 1 << 5);
35.  }

```

从源代码可以看出，pred_angular_0_8()、pred_angular_1_8()等函数的内部都调用了同样的一个函数pred_angular_8()。它们之间的不同在于传递给pred_angular_8()的最后一个参数size取值的不同。

pred_angular_8()

pred_planar_8()实现了角度（Angular）帧内预测模式，该函数的定义如下所示。

```

1.  #define POS(x, y) src[(x) + stride * (y)]
2.
3.  static av_always_inline void FUNC(pred_angular)(uint8_t *src,
4.                                                  const uint8_t *top,
5.                                                  const uint8_t *left,
6.                                                  ptrdiff_t stride, int c_idx,
7.                                                  int mode, int size)
8.  {
9.      int x, y;
10.     pixel *src = (pixel *)src;
11.     const pixel *top = (const pixel *)top;
12.     const pixel *left = (const pixel *)left;
13.     //角度
14.     static const int intra_pred_angle[] = {
15.         32, 26, 21, 17, 13, 9, 5, 2, 0, -2, -5, -9, -13, -17, -21, -26, -32,
16.         -26, -21, -17, -13, -9, -5, -2, 0, 2, 5, 9, 13, 17, 21, 26, 32
17.     };
18.     static const int inv_angle[] = {
19.         -4096, -1638, -910, -630, -482, -390, -315, -256, -315, -390, -482,
20.         -630, -910, -1638, -4096
21.     };
22.     //mode的前两种是Planar和DC，不属于角度预测
23.     int angle = intra_pred_angle[mode - 2];
24.     pixel ref_array[3 * MAX_TB_SIZE + 4];
25.     pixel *ref_tmp = ref_array + size;
26.     const pixel *ref;
27.     int last = (size * angle) >> 5;
28.
29.     if (mode >= 18) {
30.         //垂直类模式
31.
32.         ref = top - 1;
33.         if (angle < 0 && last < -1) {
34.             for (x = 0; x <= size; x += 4)
35.                 AV_WN4P(&ref_tmp[x], AV_RN4P(&top[x - 1]));
36.             for (x = last; x <= -1; x++)
37.                 ref_tmp[x] = left[-1 + ((x * inv_angle[mode - 11] + 128) >> 8)];
38.             ref = ref_tmp;
39.         }
40.
41.         for (y = 0; y < size; y++) {
42.             int idx = ((y + 1) * angle) >> 5;
43.             int fact = ((y + 1) * angle) & 31;
44.             if (fact) {

```

```

44.         if (fact) {
45.             for (x = 0; x < size; x += 4) {
46.                 POS(x, y) = ((32 - fact) * ref[x + idx + 1] +
47.                             fact * ref[x + idx + 2] + 16) >> 5;
48.                 POS(x + 1, y) = ((32 - fact) * ref[x + 1 + idx + 1] +
49.                             fact * ref[x + 1 + idx + 2] + 16) >> 5;
50.                 POS(x + 2, y) = ((32 - fact) * ref[x + 2 + idx + 1] +
51.                             fact * ref[x + 2 + idx + 2] + 16) >> 5;
52.                 POS(x + 3, y) = ((32 - fact) * ref[x + 3 + idx + 1] +
53.                             fact * ref[x + 3 + idx + 2] + 16) >> 5;
54.             }
55.         } else {
56.             for (x = 0; x < size; x += 4)
57.                 AV_WN4P(&POS(x, y), AV_RN4P(&ref[x + idx + 1]));
58.         }
59.     }
60.     if (mode == 26 && c_idx == 0 && size < 32) {
61.         for (y = 0; y < size; y++)
62.             POS(0, y) = av_clip_pixel(top[0] + ((left[y] - left[-1]) >> 1));
63.     }
64. } else {
65.     //水平类模式
66.
67.     ref = left - 1;
68.     if (angle < 0 && last < -1) {
69.         for (x = 0; x <= size; x += 4)
70.             AV_WN4P(&ref_tmp[x], AV_RN4P(&left[x - 1]));
71.         for (x = last; x <= -1; x++)
72.             ref_tmp[x] = top[-1] + ((x * inv_angle[mode - 11] + 128) >> 8)];
73.         ref = ref_tmp;
74.     }
75.
76.     for (x = 0; x < size; x++) {
77.         int idx = ((x + 1) * angle) >> 5;
78.         int fact = ((x + 1) * angle) & 31;
79.         if (fact) {
80.             for (y = 0; y < size; y++) {
81.                 POS(x, y) = ((32 - fact) * ref[y + idx + 1] +
82.                             fact * ref[y + idx + 2] + 16) >> 5;
83.             }
84.         } else {
85.             for (y = 0; y < size; y++)
86.                 POS(x, y) = ref[y + idx + 1];
87.         }
88.     }
89.     if (mode == 10 && c_idx == 0 && size < 32) {
90.         for (x = 0; x < size; x += 4) {
91.             POS(x, 0) = av_clip_pixel(left[0] + ((top[x] - top[-1]) >> 1));
92.             POS(x + 1, 0) = av_clip_pixel(left[0] + ((top[x + 1] - top[-1]) >> 1));
93.             POS(x + 2, 0) = av_clip_pixel(left[0] + ((top[x + 2] - top[-1]) >> 1));
94.             POS(x + 3, 0) = av_clip_pixel(left[0] + ((top[x + 3] - top[-1]) >> 1));
95.         }
96.     }
97. }
98. }

```

pred_planar_8()的代码还没有细看，以后有时间再做分析。
至此有关帧内预测方面的源代码就基本分析完了。后文继续分析DCT反变换相关的源代码。

DCT反变换汇编函数源代码

DCT反变换相关的汇编函数位于HEVCDSPContext中。HEVCDSPContext的初始化函数是ff_hevc_dsp_init()。该函数对HEVCDSPContext结构体中的函数指针进行了赋值。FFmpeg HEVC解码器运行的过程中只要调用HEVCDSPContext的函数指针就可以完成相应的功能。

ff_hevc_dsp_init()

ff_hevc_dsp_init()用于初始化HEVCDSPContext结构体中的汇编函数指针。该函数的定义如下所示。

```

1. void ff_hevc_dsp_init(HEVCDSPContext *hevcdsp, int bit_depth)
2. {
3.     #undef FUNC
4.     #define FUNC(a, depth) a ## _ ## depth
5.
6.     #undef PEL_FUNC
7.     #define PEL_FUNC(dst1, idx1, idx2, a, depth) \
8.         for(i = 0 ; i < 10 ; i++) \
9.         { \
10.             hevcdsp->dst1[i][idx1][idx2] = a ## _ ## depth; \
11.         }
12.
13.     #undef EPEL_FUNCS
14.     #define EPEL_FUNCS(depth) \
15.         PEL_FUNC(put_hevc_epel, 0, 0, put_hevc_pel_pixels, depth); \
16.         PEL_FUNC(put_hevc_epel, 0, 1, put_hevc_epel_h, depth); \

```

```

17. PEL_FUNC(put_hevc_epel, 1, 0, put_hevc_epel_v, depth); \
18. PEL_FUNC(put_hevc_epel, 1, 1, put_hevc_epel_hv, depth)
19.
20. #undef EPEL_UNI_FUNCS
21. #define EPEL_UNI_FUNCS(depth) \
22.     PEL_FUNC(put_hevc_epel_uni, 0, 0, put_hevc_pel_uni_pixels, depth); \
23.     PEL_FUNC(put_hevc_epel_uni, 0, 1, put_hevc_epel_uni_h, depth); \
24.     PEL_FUNC(put_hevc_epel_uni, 1, 0, put_hevc_epel_uni_v, depth); \
25.     PEL_FUNC(put_hevc_epel_uni, 1, 1, put_hevc_epel_uni_hv, depth); \
26.     PEL_FUNC(put_hevc_epel_uni_w, 0, 0, put_hevc_pel_uni_w_pixels, depth); \
27.     PEL_FUNC(put_hevc_epel_uni_w, 0, 1, put_hevc_epel_uni_w_h, depth); \
28.     PEL_FUNC(put_hevc_epel_uni_w, 1, 0, put_hevc_epel_uni_w_v, depth); \
29.     PEL_FUNC(put_hevc_epel_uni_w, 1, 1, put_hevc_epel_uni_w_hv, depth)
30.
31. #undef EPEL_BI_FUNCS
32. #define EPEL_BI_FUNCS(depth) \
33.     PEL_FUNC(put_hevc_epel_bi, 0, 0, put_hevc_pel_bi_pixels, depth); \
34.     PEL_FUNC(put_hevc_epel_bi, 0, 1, put_hevc_epel_bi_h, depth); \
35.     PEL_FUNC(put_hevc_epel_bi, 1, 0, put_hevc_epel_bi_v, depth); \
36.     PEL_FUNC(put_hevc_epel_bi, 1, 1, put_hevc_epel_bi_hv, depth); \
37.     PEL_FUNC(put_hevc_epel_bi_w, 0, 0, put_hevc_pel_bi_w_pixels, depth); \
38.     PEL_FUNC(put_hevc_epel_bi_w, 0, 1, put_hevc_epel_bi_w_h, depth); \
39.     PEL_FUNC(put_hevc_epel_bi_w, 1, 0, put_hevc_epel_bi_w_v, depth); \
40.     PEL_FUNC(put_hevc_epel_bi_w, 1, 1, put_hevc_epel_bi_w_hv, depth)
41.
42. #undef QPEL_FUNCS
43. #define QPEL_FUNCS(depth) \
44.     PEL_FUNC(put_hevc_qpel, 0, 0, put_hevc_pel_pixels, depth); \
45.     PEL_FUNC(put_hevc_qpel, 0, 1, put_hevc_qpel_h, depth); \
46.     PEL_FUNC(put_hevc_qpel, 1, 0, put_hevc_qpel_v, depth); \
47.     PEL_FUNC(put_hevc_qpel, 1, 1, put_hevc_qpel_hv, depth)
48.
49. #undef QPEL_UNI_FUNCS
50. #define QPEL_UNI_FUNCS(depth) \
51.     PEL_FUNC(put_hevc_qpel_uni, 0, 0, put_hevc_pel_uni_pixels, depth); \
52.     PEL_FUNC(put_hevc_qpel_uni, 0, 1, put_hevc_qpel_uni_h, depth); \
53.     PEL_FUNC(put_hevc_qpel_uni, 1, 0, put_hevc_qpel_uni_v, depth); \
54.     PEL_FUNC(put_hevc_qpel_uni, 1, 1, put_hevc_qpel_uni_hv, depth); \
55.     PEL_FUNC(put_hevc_qpel_uni_w, 0, 0, put_hevc_pel_uni_w_pixels, depth); \
56.     PEL_FUNC(put_hevc_qpel_uni_w, 0, 1, put_hevc_qpel_uni_w_h, depth); \
57.     PEL_FUNC(put_hevc_qpel_uni_w, 1, 0, put_hevc_qpel_uni_w_v, depth); \
58.     PEL_FUNC(put_hevc_qpel_uni_w, 1, 1, put_hevc_qpel_uni_w_hv, depth)
59.
60. #undef QPEL_BI_FUNCS
61. #define QPEL_BI_FUNCS(depth) \
62.     PEL_FUNC(put_hevc_qpel_bi, 0, 0, put_hevc_pel_bi_pixels, depth); \
63.     PEL_FUNC(put_hevc_qpel_bi, 0, 1, put_hevc_qpel_bi_h, depth); \
64.     PEL_FUNC(put_hevc_qpel_bi, 1, 0, put_hevc_qpel_bi_v, depth); \
65.     PEL_FUNC(put_hevc_qpel_bi, 1, 1, put_hevc_qpel_bi_hv, depth); \
66.     PEL_FUNC(put_hevc_qpel_bi_w, 0, 0, put_hevc_pel_bi_w_pixels, depth); \
67.     PEL_FUNC(put_hevc_qpel_bi_w, 0, 1, put_hevc_qpel_bi_w_h, depth); \
68.     PEL_FUNC(put_hevc_qpel_bi_w, 1, 0, put_hevc_qpel_bi_w_v, depth); \
69.     PEL_FUNC(put_hevc_qpel_bi_w, 1, 1, put_hevc_qpel_bi_w_hv, depth)
70.
71. #define HEVC_DSP(depth) \
72.     hevcdsp->put_pcm = FUNC(put_pcm, depth); \
73.     hevcdsp->transform_add[0] = FUNC(transform_add4x4, depth); \
74.     hevcdsp->transform_add[1] = FUNC(transform_add8x8, depth); \
75.     hevcdsp->transform_add[2] = FUNC(transform_add16x16, depth); \
76.     hevcdsp->transform_add[3] = FUNC(transform_add32x32, depth); \
77.     hevcdsp->transform_skip = FUNC(transform_skip, depth); \
78.     hevcdsp->transform_rdp_pcm = FUNC(transform_rdp_pcm, depth); \
79.     hevcdsp->idct_4x4_luma = FUNC(transform_4x4_luma, depth); \
80.     hevcdsp->idct[0] = FUNC(idct_4x4, depth); \
81.     hevcdsp->idct[1] = FUNC(idct_8x8, depth); \
82.     hevcdsp->idct[2] = FUNC(idct_16x16, depth); \
83.     hevcdsp->idct[3] = FUNC(idct_32x32, depth); \
84. \
85.     hevcdsp->idct_dc[0] = FUNC(idct_4x4_dc, depth); \
86.     hevcdsp->idct_dc[1] = FUNC(idct_8x8_dc, depth); \
87.     hevcdsp->idct_dc[2] = FUNC(idct_16x16_dc, depth); \
88.     hevcdsp->idct_dc[3] = FUNC(idct_32x32_dc, depth); \
89. \
90.     hevcdsp->sao_band_filter = FUNC(sao_band_filter_0, depth); \
91.     hevcdsp->sao_edge_filter[0] = FUNC(sao_edge_filter_0, depth); \
92.     hevcdsp->sao_edge_filter[1] = FUNC(sao_edge_filter_1, depth); \
93. \
94.     QPEL_FUNCS(depth); \
95.     QPEL_UNI_FUNCS(depth); \
96.     QPEL_BI_FUNCS(depth); \
97.     EPEL_FUNCS(depth); \
98.     EPEL_UNI_FUNCS(depth); \
99.     EPEL_BI_FUNCS(depth); \
100. \
101.     hevcdsp->hevc_h_loop_filter_luma = FUNC(hevc_h_loop_filter_luma, depth); \
102.     hevcdsp->hevc_v_loop_filter_luma = FUNC(hevc_v_loop_filter_luma, depth); \
103.     hevcdsp->hevc_h_loop_filter_chroma = FUNC(hevc_h_loop_filter_chroma, depth); \
104.     hevcdsp->hevc_v_loop_filter_chroma = FUNC(hevc_v_loop_filter_chroma, depth); \
105.     hevcdsp->hevc_h_loop_filter_luma_c = FUNC(hevc_h_loop_filter_luma, depth); \
106.     hevcdsp->hevc_v_loop_filter_luma_c = FUNC(hevc_v_loop_filter_luma, depth); \
107.     hevcdsp->hevc_h_loop_filter_chroma_c = FUNC(hevc_h_loop_filter_chroma, depth); \

```

```

108.     hevcdsp->hevc_v_loop_filter_chroma_c = FUNC(hevc_v_loop_filter_chroma, depth)
109.     int i = 0;
110.
111.     switch (bit_depth) {
112.     case 9:
113.         HEVC_DSP(9);
114.         break;
115.     case 10:
116.         HEVC_DSP(10);
117.         break;
118.     case 12:
119.         HEVC_DSP(12);
120.         break;
121.     default:
122.         HEVC_DSP(8);
123.         break;
124.     }
125.
126.     if (ARCH_X86)
127.         ff_hevc_dsp_init_x86(hevcdsp, bit_depth);
128. }

```

从源代码可以看出，ff_hevc_dsp_init()函数中包含一个名为“HEVC_DSP(depth)”的很长的宏定义。该宏定义中包含了C语言版本的各种函数的初始化代码。ff_hevc_dsp_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X86架构的，则会调用ff_hevc_dsp_init_x86()初始化X86平台下经过汇编优化的函数。下面以8bit颜色位深为例，看一下“HEVC_DSP(8)”的展开结果中和DCT相关的函数。

```

[cpp]
1.  hevcdsp->transform_add[0]      = transform_add4x4_8;
2.  hevcdsp->transform_add[1]      = transform_add8x8_8;
3.  hevcdsp->transform_add[2]      = transform_add16x16_8;
4.  hevcdsp->transform_add[3]      = transform_add32x32_8;
5.  hevcdsp->transform_skip        = transform_skip_8;
6.  hevcdsp->transform_rdpcom      = transform_rdpcom_8;
7.  hevcdsp->idct_4x4_luma         = transform_4x4_luma_8;
8.  hevcdsp->idct[0]               = idct_4x4_8;
9.  hevcdsp->idct[1]               = idct_8x8_8;
10. hevcdsp->idct[2]               = idct_16x16_8;
11. hevcdsp->idct[3]               = idct_32x32_8;
12.
13. hevcdsp->idct_dc[0]             = idct_4x4_dc_8;
14. hevcdsp->idct_dc[1]             = idct_8x8_dc_8;
15. hevcdsp->idct_dc[2]             = idct_16x16_dc_8;
16. hevcdsp->idct_dc[3]             = idct_32x32_dc_8;
17. //略...
    <span style="font-family: Arial, Helvetica, sans-serif;">

```

通过上述代码可以总结出下面几个IDCT函数（数组）：

HEVCDSPContext -> idct[4]()：DCT反变换函数。数组中4个函数分别处理4x4，8x8，16x16，32x32几种块。

HEVCDSPContext -> idct_dc[4]()：只有DC系数时候的DCT反变换函数（运算速度比普通DCT快一些）。数组中4个函数分别处理4x4，8x8，16x16，32x32几种块。

HEVCDSPContext -> idct_4x4_luma()：特殊的4x4DST反变换函数。在处理Intra4x4块的时候，HEVC使用了一种比较特殊的DST（而不是DCT），可以微量的提高编码效率。

HEVCDSPContext -> transform_add[4]()：残差叠加函数，用于将IDCT之后的残差像素数据叠加到预测像素数据之上。数组中4个函数分别处理4x4，8x8，16x16，32x32几种块。

PS：还有几种IDCT函数目前还没有看，就不列出了。

下面分别看一下上述的几种函数。

HEVCDSPContext -> idct[4]()

HEVCPredContext -> idct[4]()指向了DCT反变换的汇编函数。数组中4个元素分别处理4x4，8x8，16x16，32x32几种块。这几种块的具体C语言版本处理函数为：

idct_4x4_8()——4x4块；

idct_8x8_8()——8x8块；

idct_16x16_8()——16x16块；

idct_32x32_8()——32x32块；

这四个函数的定义如下所示。

```

1. #define SET(dst, x) (dst) = (x)
2. #define SCALE(dst, x) (dst) = av_clip_int16(((x) + add) >> shift)
3. #define ADD_AND_SCALE(dst, x) \
4.     (dst) = av_clip_pixel((dst) + av_clip_int16(((x) + add) >> shift))
5.
6. #define IDCT_VAR4(H) \
7.     int limit2 = FFMIN(col_limit + 4, H)
8. #define IDCT_VAR8(H) \
9.     int limit = FFMIN(col_limit, H); \
10.    int limit2 = FFMIN(col_limit + 4, H)
11. #define IDCT_VAR16(H) IDCT_VAR8(H)
12. #define IDCT_VAR32(H) IDCT_VAR8(H)
13.
14. //其中的“H”取4,8,16,32
15. //可以拼凑出不同的函数
16. #define IDCT(H) \
17. static void FUNC(idct_##H ##x ##H) { \
18.     int16_t *coeffs, int col_limit) { \
19.     int i; \
20.     int shift = 7; \
21.     int add = 1 << (shift - 1); \
22.     int16_t *src = coeffs; \
23.     IDCT_VAR ##H(H); \
24.     \
25.     for (i = 0; i < H; i++) { \
26.         TR_ ## H(src, src, H, H, SCALE, limit2); \
27.         if (limit2 < H && i%4 == 0 && !!i) \
28.             limit2 -= 4; \
29.         src++; \
30.     } \
31.     \
32.     shift = 20 - BIT_DEPTH; \
33.     add = 1 << (shift - 1); \
34.     for (i = 0; i < H; i++) { \
35.         TR_ ## H(coeffs, coeffs, 1, 1, SCALE, limit); \
36.         coeffs += H; \
37.     } \
38. }
39.
40. //几种不同尺度的IDCT
41. IDCT( 4)
42. IDCT( 8)
43. IDCT(16)
44. IDCT(32)

```

从源代码可以看出，idct_4x4_8()、idct_8x8_8()等函数的定义是通过“IDCT()”宏实现的。而“IDCT(H)”宏中又调用了另外一个宏“TR_ ## H()”。“TR_ ## H()”根据“H”取值的不同，可以调用：

TR_4()——用于4x4DCT
 TR_8()——用于8x8DCT
 TR_16()——用于16x16DCT
 TR_32()——用于32x32DCT

TR4()、TR8()、TR16()、TR32()的定义如下所示。


```

1.  /*
2.  * 4x4DCT
3.  *
4.  *      | 64  64  64  64 |
5.  * H = | 83  36 -36 -83 |
6.  *      | 64 -64 -64  64 |
7.  *      | 36 -83  83 -36 |
8.  *
9.  */
10. #define TR_4(dst, src, dstep, sstep, assign, end) \
11.     do { \
12.         const int e0 = 64 * src[0 * sstep] + 64 * src[2 * sstep]; \
13.         const int e1 = 64 * src[0 * sstep] - 64 * src[2 * sstep]; \
14.         const int o0 = 83 * src[1 * sstep] + 36 * src[3 * sstep]; \
15.         const int o1 = 36 * src[1 * sstep] - 83 * src[3 * sstep]; \
16.         \
17.         assign(dst[0 * dstep], e0 + o0); \
18.         assign(dst[1 * dstep], e1 + o1); \
19.         assign(dst[2 * dstep], e1 - o1); \
20.         assign(dst[3 * dstep], e0 - o0); \
21.     } while (0)
22.
23. /*
24. * 8x8DCT
25. *
26. * transform[]存储了32x32DCT变换系数
27. * 8x8DCT变换的系数来自于32x32系数矩阵中第0, 4, 8, 12, 16, 20, 24, 28行元素中的前8个元素
28. *
29. */
30. #define TR_8(dst, src, dstep, sstep, assign, end) \
31.     do { \
32.         int i, j; \
33.         int e_8[4]; \
34.         int o_8[4] = { 0 }; \
35.         for (i = 0; i < 4; i++) \
36.             for (j = 1; j < end; j += 2) \
37.                 o_8[i] += transform[4 * j][i] * src[j * sstep]; \
38.         TR_4(e_8, src, 1, 2 * sstep, SET, 4); \
39.         \
40.         for (i = 0; i < 4; i++) { \
41.             assign(dst[i * dstep], e_8[i] + o_8[i]); \
42.             assign(dst[(7 - i) * dstep], e_8[i] - o_8[i]); \
43.         } \
44.     } while (0)
45.
46. /*
47. * 16x16DCT
48. * 16x16 DCT变换的系数来自于32x32系数矩阵中第0, 2, 4..., 28, 30行元素中的前16个元素
49. *
50. */
51. #define TR_16(dst, src, dstep, sstep, assign, end) \
52.     do { \
53.         int i, j; \
54.         int e_16[8]; \
55.         int o_16[8] = { 0 }; \
56.         for (i = 0; i < 8; i++) \
57.             for (j = 1; j < end; j += 2) \
58.                 o_16[i] += transform[2 * j][i] * src[j * sstep]; \
59.         TR_8(e_16, src, 1, 2 * sstep, SET, 8); \
60.         \
61.         for (i = 0; i < 8; i++) { \
62.             assign(dst[i * dstep], e_16[i] + o_16[i]); \
63.             assign(dst[(15 - i) * dstep], e_16[i] - o_16[i]); \
64.         } \
65.     } while (0)
66.
67. /*
68. * 32x32DCT
69. *
70. */
71. #define TR_32(dst, src, dstep, sstep, assign, end) \
72.     do { \
73.         int i, j; \
74.         int e_32[16]; \
75.         int o_32[16] = { 0 }; \
76.         for (i = 0; i < 16; i++) \
77.             for (j = 1; j < end; j += 2) \
78.                 o_32[i] += transform[j][i] * src[j * sstep]; \
79.         TR_16(e_32, src, 1, 2 * sstep, SET, end/2); \
80.         \
81.         for (i = 0; i < 16; i++) { \
82.             assign(dst[i * dstep], e_32[i] + o_32[i]); \
83.             assign(dst[(31 - i) * dstep], e_32[i] - o_32[i]); \
84.         } \
85.     } while (0)

```

有关这一部分的源代码目前还没有细看，以后有时间再进行补充。从TR8()、TR16()等的定义中可以看出，它们的DCT系数来自于一个transform[32][32]数组。

transform[32][32]

transform[32][32] 的定义如下所示，其中存储了32x32DCT的系数。使用该系数矩阵，也可以推导获得16x16DCT、8x8DCT、4x4DCT的系数。

```
[cpp]
1. //32x32DCT变换系数
2. static const int8_t transform[32][32] = {
3.     { 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
4.       64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64 },
5.     { 90, 90, 88, 85, 82, 78, 73, 67, 61, 54, 46, 38, 31, 22, 13, 4,
6.       -4, -13, -22, -31, -38, -46, -54, -61, -67, -73, -78, -82, -85, -88, -90, -90 },
7.     { 90, 87, 80, 70, 57, 43, 25, 9, -9, -25, -43, -57, -70, -80, -87, -90,
8.       -90, -87, -80, -70, -57, -43, -25, -9, 9, 25, 43, 57, 70, 80, 87, 90 },
9.     { 90, 82, 67, 46, 22, -4, -31, -54, -73, -85, -90, -88, -78, -61, -38, -13,
10.      13, 38, 61, 78, 88, 90, 85, 73, 54, 31, 4, -22, -46, -67, -82, -90 },
11.     { 89, 75, 50, 18, -18, -50, -75, -89, -89, -75, -50, -18, 18, 50, 75, 89,
12.      89, 75, 50, 18, -18, -50, -75, -89, -89, -75, -50, -18, 18, 50, 75, 89 },
13.     { 88, 67, 31, -13, -54, -82, -90, -78, -46, -4, 38, 73, 90, 85, 61, 22,
14.      -22, -61, -85, -90, -73, -38, 4, 46, 78, 90, 82, 54, 13, -31, -67, -88 },
15.     { 87, 57, 9, -43, -80, -90, -70, -25, 25, 70, 90, 80, 43, -9, -57, -87,
16.      -87, -57, -9, 43, 80, 90, 70, 25, -25, -70, -90, -80, -43, 9, 57, 87 },
17.     { 85, 46, -13, -67, -90, -73, -22, 38, 82, 88, 54, -4, -61, -90, -78, -31,
18.      31, 78, 90, 61, 4, -54, -88, -82, -38, 22, 73, 90, 67, 13, -46, -85 },
19.     { 83, 36, -36, -83, -83, -36, 36, 83, 83, 36, -36, -83, -83, -36, 36, 83,
20.      83, 36, -36, -83, -83, -36, 36, 83, 83, 36, -36, -83, -83, -36, 36, 83 },
21.     { 82, 22, -54, -90, -61, 13, 78, 85, 31, -46, -90, -67, 4, 73, 88, 38,
22.      -38, -88, -73, -4, 67, 90, 46, -31, -85, -78, -13, 61, 90, 54, -22, -82 },
23.     { 80, 9, -70, -87, -25, 57, 90, 43, -43, -90, -57, 25, 87, 70, -9, -80,
24.      -80, -9, 70, 87, 25, -57, -90, -43, 43, 90, 57, -25, -87, -70, 9, 80 },
25.     { 78, -4, -82, -73, 13, 85, 67, -22, -88, -61, 31, 90, 54, -38, -90, -46,
26.      46, 90, 38, -54, -90, -31, 61, 88, 22, -67, -85, -13, 73, 82, 4, -78 },
27.     { 75, -18, -89, -50, 50, 89, 18, -75, -75, 18, 89, 50, -50, -89, -18, 75,
28.      75, -18, -89, -50, 50, 89, 18, -75, -75, 18, 89, 50, -50, -89, -18, 75 },
29.     { 73, -31, -90, -22, 78, 67, -38, -90, -13, 82, 61, -46, -88, -4, 85, 54,
30.      -54, 85, 4, 88, 46, -61, -82, 13, 90, 38, -67, -78, 22, 90, 31, -73 },
31.     { 70, -43, -87, 9, 90, 25, -80, -57, 57, 80, -25, -90, -9, 87, 43, -70,
32.      -70, 43, 87, -9, -90, -25, 80, 57, -57, -80, 25, 90, 9, -87, -43, 70 },
33.     { 67, -54, -78, 38, 85, -22, -90, 4, 90, 13, -88, -31, 82, 46, -73, -61,
34.      61, 73, -46, -82, 31, 88, -13, -90, -4, 90, 22, -85, -38, 78, 54, -67 },
35.     { 64, -64, -64, 64, 64, -64, -64, 64, 64, -64, -64, 64, 64, -64, -64, 64,
36.      64, -64, -64, 64, 64, -64, -64, 64, 64, -64, -64, 64, 64, -64, -64, 64 },
37.     { 61, -73, -46, 82, 31, -88, -13, 90, -4, -90, 22, 85, -38, -78, 54, 67,
38.      -67, -54, 78, 38, -85, -22, 90, 4, -90, 13, 88, -31, -82, 46, 73, -61 },
39.     { 57, -80, -25, 90, -9, -87, 43, 70, -70, -43, 87, 9, -90, 25, 80, -57,
40.      -57, 80, 25, -90, 9, 87, -43, -70, 70, 43, -87, -9, 90, -25, -80, 57 },
41.     { 54, -85, -4, 88, -46, -61, 82, 13, -90, 38, 67, -78, -22, 90, -31, -73,
42.      73, 31, -90, 22, 78, -67, -38, 90, -13, -82, 61, 46, -88, 4, 85, -54 },
43.     { 50, -89, 18, 75, -75, -18, 89, -50, -50, 89, -18, -75, 75, 18, -89, 50,
44.      50, -89, 18, 75, -75, -18, 89, -50, -50, 89, -18, -75, 75, 18, -89, 50 },
45.     { 46, -90, 38, 54, -90, 31, 61, -88, 22, 67, -85, 13, 73, -82, 4, 78,
46.      -78, -4, 82, -73, -13, 85, -67, -22, 88, -61, -31, 90, -54, -38, 90, -46 },
47.     { 43, -90, 57, 25, -87, 70, 9, -80, 80, -9, -70, 87, -25, -57, 90, -43,
48.      -43, 90, -57, -25, 87, -70, -9, 80, -80, 9, 70, -87, 25, 57, -90, 43 },
49.     { 38, -88, 73, -4, -67, 90, -46, -31, 85, -78, 13, 61, -90, 54, 22, -82,
50.      82, -22, -54, 90, -61, -13, 78, -85, 31, 46, -90, 67, 4, -73, 88, -38 },
51.     { 36, -83, 83, -36, -36, 83, -83, 36, 36, -83, 83, -36, -36, 83, -83, 36,
52.      36, -83, 83, -36, -36, 83, -83, 36, 36, -83, 83, -36, -36, 83, -83, 36 },
53.     { 31, -78, 90, -61, 4, 54, -88, 82, -38, -22, 73, -90, 67, -13, -46, 85,
54.      -85, 46, 13, -67, 90, -73, 22, 38, -82, 88, -54, -4, 61, -90, 78, -31 },
55.     { 25, -70, 90, -80, 43, 9, -57, 87, -87, 57, -9, -43, 80, -90, 70, -25,
56.      -25, 70, -90, 80, -43, -9, 57, -87, 87, -57, 9, 43, -80, 90, -70, 25 },
57.     { 22, -61, 85, -90, 73, -38, -4, 46, -78, 90, -82, 54, -13, -31, 67, -88,
58.      88, -67, 31, 13, -54, 82, -90, 78, -46, 4, 38, -73, 90, -85, 61, -22 },
59.     { 18, -50, 75, -89, 89, -75, 50, -18, -18, 50, -75, 89, -89, 75, -50, 18,
60.      18, -50, 75, -89, 89, -75, 50, -18, -18, 50, -75, 89, -89, 75, -50, 18 },
61.     { 13, -38, 61, -78, 88, -90, 85, -73, 54, -31, 4, 22, -46, 67, -82, 90,
62.      -90, 82, -67, 46, -22, -4, 31, -54, 73, -85, 90, -88, 78, -61, 38, -13 },
63.     { 9, -25, 43, -57, 70, -80, 87, -90, 90, -87, 80, -70, 57, -43, 25, -9,
64.      -9, 25, -43, 57, -70, 80, -87, 90, -90, 87, -80, 70, -57, 43, -25, 9 },
65.     { 4, -13, 22, -31, 78, -46, 54, -61, 67, -73, 78, -82, 85, -88, 90, -90,
66.      90, -90, 88, -85, 82, -78, 73, -67, 61, -54, 46, -38, 31, -22, 13, -4 },
67. };
```

HEVCDSPContext -> idct_dc[4]()

HEVCPredContext -> idct_dc[4]()指向了只有DC系数时候的DCT反变换的汇编函数。只有DC系数的DCT反变换属于一种比较特殊的情况，在这种情况下使用idct_dc[4]()的速度会比idct[4]()要快一些。数组中4个元素分别处理4x4, 8x8, 16x16, 32x32几种块。这几种块的具体C语言版本处理函数为：

idct_4x4_dc_8()——4x4块；

idct_8x8_dc_8()——8x8块；

idct_16x16_dc_8()——16x16块；

idct_32x32_dc_8()——32x32块；

这四个函数的定义如下所示。

```

1. #define IDCT_DC(H) \
2. static void FUNC(idct_##H ##x ##H ##_dc)( \
3.     int16_t *coeffs) { \
4.     int i, j; \
5.     int shift = 14 - BIT_DEPTH; \
6.     int add = 1 << (shift - 1); \
7.     int coeff = (((coeffs[0] + 1) >> 1) + add) >> shift; \
8. \
9.     for (j = 0; j < H; j++) { \
10.         for (i = 0; i < H; i++) { \
11.             coeffs[i+j*H] = coeff; \
12.         } \
13.     } \
14. } \
15. \
16. \
17. //只包含DC系数时候的比较快速的IDCT
18. IDCT_DC( 4)
19. IDCT_DC( 8)
20. IDCT_DC(16)
21. IDCT_DC(32)

```

可以看出idct_4x4_dc_8()、idct_8x8_dc_8()等函数的初始化是通过“IDCT_DC()”宏完成的。可以看出“IDCT_DC()”首先通过DC系数coeffs[0]换算得到值coeff，然后将coeff赋值给系数矩阵中的每个系数。

HEVCDSPContext -> idct_4x4_luma()

HEVCDSPContext -> idct_4x4_luma()指向处理Intra4x4的CU的DST反变换。相比于视频编码中常见的DCT反变换，DST反变换是一种比较特殊的变换。4x4DST反变换的C语言版本函数是transform_4x4_luma_8()，它的定义如下所示。

```

1. #define SCALE(dst, x) (dst) = av_clip_int16(((x) + add) >> shift)
2.
3. /*
4.  * 4x4DST
5.  *
6.  * | 29 55 74 84 |
7.  * H = | 74 74 0 -74 |
8.  * | 84 -29 -74 55 |
9.  * | 55 -84 74 -29 |
10.  *
11.  */
12. #define TR_4x4_LUMA(dst, src, step, assign) \
13. do { \
14.     int c0 = src[0 * step] + src[2 * step]; \
15.     int c1 = src[2 * step] + src[3 * step]; \
16.     int c2 = src[0 * step] - src[3 * step]; \
17.     int c3 = 74 * src[1 * step]; \
18. \
19.     assign(dst[2 * step], 74 * (src[0 * step] - \
20.         src[2 * step] + \
21.         src[3 * step])); \
22.     assign(dst[0 * step], 29 * c0 + 55 * c1 + c3); \
23.     assign(dst[1 * step], 55 * c2 - 29 * c1 + c3); \
24.     assign(dst[3 * step], 55 * c0 + 29 * c2 - c3); \
25. } while (0)
26.
27. //4x4DST
28. static void FUNC(transform_4x4_luma)(int16_t *coeffs)
29. {
30.     int i;
31.     int shift = 7;
32.     int add = 1 << (shift - 1);
33.     int16_t *src = coeffs;
34.
35.     for (i = 0; i < 4; i++) {
36.         TR_4x4_LUMA(src, src, 4, SCALE);
37.         src++;
38.     }
39.
40.     shift = 20 - BIT_DEPTH;
41.     add = 1 << (shift - 1);
42.     for (i = 0; i < 4; i++) {
43.         TR_4x4_LUMA(coeffs, coeffs, 1, SCALE);
44.         coeffs += 4;
45.     }
46. }
47.
48. #undef TR_4x4_LUMA

```



从源代码可以看出，transform_4x4_luma_8()调用TR_4x4_LUMA()完成了4x4DST的工作。

HEVCDSPContext -> transform_add[4]()

HEVCDSPContext -> transform_add[4]()指向了叠加残差数据的汇编函数。这些函数用于将残差像素数据叠加到预测像素数据上，形成最后的解码图像数据。数组中4个元素分别处理4x4，8x8，16x16，32x32几种块。这几种块的具体C语言版本处理函数为：

```
transform_add4x4_8()——4x4块；  
transform_add8x8_8()——8x8块；  
transform_add16x16_8()——16x16块；  
transform_add32x32_8()——32x32块；
```

这四个函数的定义如下所示。

```
[cpp]    
1. //叠加4x4方块的残差数据  
2. static void FUNC(transform_add4x4)(uint8_t *_dst, int16_t *coeffs,  
3.                                ptrdiff_t stride)  
4. {  
5.     //最后一个参数为4  
6.     FUNC(transquant_bypass)(_dst, coeffs, stride, 4);  
7. }  
8. //叠加8x8方块的残差数据  
9. static void FUNC(transform_add8x8)(uint8_t *_dst, int16_t *coeffs,  
10.                                ptrdiff_t stride)  
11. {  
12.     //最后一个参数为8  
13.     FUNC(transquant_bypass)(_dst, coeffs, stride, 8);  
14. }  
15. //叠加16x16方块的残差数据  
16. static void FUNC(transform_add16x16)(uint8_t *_dst, int16_t *coeffs,  
17.                                ptrdiff_t stride)  
18. {  
19.     //最后一个参数为16  
20.     FUNC(transquant_bypass)(_dst, coeffs, stride, 16);  
21. }  
22. //叠加32x32方块的残差数据  
23. static void FUNC(transform_add32x32)(uint8_t *_dst, int16_t *coeffs,  
24.                                ptrdiff_t stride)  
25. {  
26.     //最后一个参数为32  
27.     FUNC(transquant_bypass)(_dst, coeffs, stride, 32);  
28. }
```

从源代码可以看出，transform_add4x4_8()、transform_add8x8_8()等函数内部都调用了同样一个函数transquant_bypass_8()，它们的不同在于传递给transquant_bypass_8()的最后一个参数size的值不同。

transquant_bypass_8()

transquant_bypass_8()完成了残差像素数据叠加的工作。该函数的定义如下所示。

```
[cpp]    
1. //叠加残差数据, transquant_bypass_8()  
2. static av_always_inline void FUNC(transquant_bypass)(uint8_t *_dst, int16_t *coeffs,  
3.                                ptrdiff_t stride, int size)  
4. {  
5.     int x, y;  
6.     pixel *_dst = (pixel *)_dst;  
7.  
8.     stride /= sizeof(pixel);  
9.     //逐个叠加每个点  
10.    for (y = 0; y < size; y++) {  
11.        for (x = 0; x < size; x++) {  
12.            dst[x] = av_clip_pixel(dst[x] + *coeffs); //叠加, av_clip_pixel()用于限幅。处理的数据一直存储于dst  
13.            coeffs++;  
14.        }  
15.        dst += stride;  
16.    }  
17. }
```

从源代码中可以看出，transquant_bypass_8()将残差数据coeff依次叠加到了预测数据dst之上。

至此有关IDCT方面的源代码就基本分析完毕了。

雷霄骅

leixiaohua1020@126.com

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

版权声明：本文为博主原创文章，未经博主允许不得转载。<https://blog.csdn.net/leixiaohua1020/article/details/46451119>

文章标签：[FFmpeg](#) [libavcodec](#) [HEVC](#) [解码器](#) [TU](#)

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