

## 原 x264源代码简单分析：宏块编码（Encode）部分

2015年05月24日 13:47:40 阅读数：8205

=====

H.264源代码分析文章列表：

【编码 - x264】

[x264源代码简单分析：概述](#)

[x264源代码简单分析：x264命令行工具（x264.exe）](#)

[x264源代码简单分析：编码器主干部分-1](#)

[x264源代码简单分析：编码器主干部分-2](#)

[x264源代码简单分析：x264\\_slice\\_write\(\)](#)

[x264源代码简单分析：滤波（Filter）部分](#)

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

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

[x264源代码简单分析：宏块编码（Encode）部分](#)

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

[FFmpeg与libx264接口源代码简单分析](#)

【解码 - libavcodec H.264 解码器】

[FFmpeg的H.264解码器源代码简单分析：概述](#)

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

[FFmpeg的H.264解码器源代码简单分析：解码器主干部分](#)

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

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

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

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

=====

本文记录x264的 `x264_slice_write()`函数中调用的`x264_macroblock_encode()`的源代码。`x264_macroblock_encode()`对应着x264中的宏块编码模块。宏块编码模块主要完成了DCT变换和量化两个步骤。

## 函数调用关系图

宏块编码（Encode）部分的源代码在整个x264中的位置如下图所示。



宏块编码（Encode）部分的函数调用关系如下图所示。



从图中可以看出，宏块编码模块的`x264_macroblock_encode()`调用了`x264_macroblock_encode_internal()`，而`x264_macroblock_encode_internal()`完成了如下功能：

- `x264_macroblock_encode_skip()`：编码Skip类型宏块。
- `x264_mb_encode_i16x16()`：编码Intra16x16类型的宏块。该函数除了进行DCT变换之外，还对16个小块的DC系数进行了Hadamard变换。

x264\_mb\_encode\_i4x4()：编码Intra4x4类型的宏块。

帧间宏块编码：这一部分代码直接写在了函数体里面。

x264\_mb\_encode\_chroma()：编码色度块。

本文将会分析上述函数中除了色度编码外的几个函数。

## x264\_slice\_write()

x264\_slice\_write()是x264项目的核心，它完成了编码了一个Slice的工作。有关该函数的分析可以参考文章《x264源代码简单分析：x264\_slice\_write()》。本文分析其调用的x264\_macroblock\_encode()函数。

## x264\_macroblock\_encode()

x264\_macroblock\_encode()用于编码宏块。该函数的定义位于encoder/macroblock.c，如下所示。

```
[cpp]
1. //编码-残差DCT变换、量化
2. void x264_macroblock_encode( x264_t *h )
3. {
4.     //编码-内部函数
5.     //YUV444相当于把YUV3个分量都当做Y编码
6.     if( CHROMA444 )
7.         x264_macroblock_encode_internal( h, 3, 0 );
8.     else
9.         x264_macroblock_encode_internal( h, 1, 1 );
10. }
```

从源代码可以看出，x264\_macroblock\_encode()封装了x264\_macroblock\_encode\_internal()。如果色度模式是YUV444的话，传递的参数plane\_count=3而chroma=0；如果不是YUV444的话，传递的参数plane\_count=1而chroma=1。

## x264\_macroblock\_encode\_internal()

x264\_macroblock\_encode\_internal()是x264\_macroblock\_encode()的内部函数。该函数的定义位于encoder/macroblock.c，如下所示。

```
[cpp]
1. /*****
2.  * x264_macroblock_encode:
3.  * 编码-残差DCT变换、量化-内部函数
4.  *
5.  * 注释和处理：雷霄骅
6.  * http://blog.csdn.net/leixiaohua1020
7.  * leixiaohua1020@126.com
8.  *****/
9. static ALWAYS_INLINE void x264_macroblock_encode_internal( x264_t *h, int plane_count, int chroma )
10. {
11.     int i_qp = h->mb.i_qp;
12.     int b_decimate = h->mb.b_dct_decimate;
13.     int b_force_no_skip = 0;
14.     int nz;
15.     h->mb.i_cbp_luma = 0;
16.     for( int p = 0; p < plane_count; p++ )
17.         h->mb.cache.non_zero_count[x264_scan8[LUMA_DC+p]] = 0;
18.     //PCM, 不常见
19.     if( h->mb.i_type == I_PCM )
20.     {
21.         /* if PCM is chosen, we need to store reconstructed frame data */
22.         for( int p = 0; p < plane_count; p++ )
23.             h->mc.copy[PIXEL_16x16]( h->mb.pic.p_fdec[p], FDEC_STRIDE, h->mb.pic.p_fenc[p], FENC_STRIDE, 16 );
24.         if( chroma )
25.         {
26.             int height = 16 >> CHROMA_V_SHIFT;
27.             h->mc.copy[PIXEL_8x8]( h->mb.pic.p_fdec[1], FDEC_STRIDE, h->mb.pic.p_fenc[1], FENC_STRIDE, height );
28.             h->mc.copy[PIXEL_8x8]( h->mb.pic.p_fdec[2], FDEC_STRIDE, h->mb.pic.p_fenc[2], FENC_STRIDE, height );
29.         }
30.         return;
31.     }
32.
33.     if( !h->mb.b_allow_skip )
34.     {
35.         b_force_no_skip = 1;
36.         if( IS_SKIP(h->mb.i_type) )
37.         {
38.             if( h->mb.i_type == P_SKIP )
39.                 h->mb.i_type = P_L0;
40.             else if( h->mb.i_type == B_SKIP )
41.                 h->mb.i_type = B_DIRECT;
42.         }
43.     }
44.     //根据不同的宏块类型，进行编码
```

```

45.     if( h->mb.i_type == P_SKIP )
46.     {
47.         /* don't do pskip motion compensation if it was already done in macroblock_analyse */
48.         if( !h->mb.b_skip_mc )
49.         {
50.             int mvx = x264_clip3( h->mb.cache.mv[0][x264_scan8[0]][0],
51.                                   h->mb.mv_min[0], h->mb.mv_max[0] );
52.             int mvy = x264_clip3( h->mb.cache.mv[0][x264_scan8[0]][1],
53.                                   h->mb.mv_min[1], h->mb.mv_max[1] );
54.
55.             for( int p = 0; p < plane_count; p++ )
56.                 h->mc.mc_luma( h->mb.pic.p_fdec[p], FDEC_STRIDE,
57.                               &h->mb.pic.p_fref[0][0][p*4], h->mb.pic.i_stride[p],
58.                               mvx, mvy, 16, 16, &h->sh.weight[0][p] );
59.
60.             if( chroma )
61.             {
62.                 int v_shift = CHROMA_V_SHIFT;
63.                 int height = 16 >> v_shift;
64.
65.                 /* Special case for mv0, which is (of course) very common in P-skip mode. */
66.                 if( mvx | mvy )
67.                     h->mc.mc_chroma( h->mb.pic.p_fdec[1], h->mb.pic.p_fdec[2], FDEC_STRIDE,
68.                                       h->mb.pic.p_fref[0][0][4], h->mb.pic.i_stride[1],
69.                                       mvx, 2*mvy>>v_shift, 8, height );
70.                 else
71.                     h->mc.load_deinterleave_chroma_fdec( h->mb.pic.p_fdec[1], h->mb.pic.p_fref[0][0][4],
72.                                                           h->mb.pic.i_stride[1], height );
73.
74.                 if( h->sh.weight[0][1].weightfn )
75.                     h->sh.weight[0][1].weightfn[8>>2]( h->mb.pic.p_fdec[1], FDEC_STRIDE,
76.                                                           h->mb.pic.p_fdec[1], FDEC_STRIDE,
77.                                                           &h->sh.weight[0][1], height );
78.                 if( h->sh.weight[0][2].weightfn )
79.                     h->sh.weight[0][2].weightfn[8>>2]( h->mb.pic.p_fdec[2], FDEC_STRIDE,
80.                                                           h->mb.pic.p_fdec[2], FDEC_STRIDE,
81.                                                           &h->sh.weight[0][2], height );
82.             }
83.         }
84.         //编码skip类型宏块
85.         x264_macroblock_encode_skip( h );
86.         return;
87.     }
88.     if( h->mb.i_type == B_SKIP )
89.     {
90.         /* don't do bskip motion compensation if it was already done in macroblock_analyse */
91.         if( !h->mb.b_skip_mc )
92.             x264_mb_mc( h );
93.         x264_macroblock_encode_skip( h );
94.         return;
95.     }
96.
97.     if( h->mb.i_type == I_16x16 )
98.     {
99.         h->mb.b_transform_8x8 = 0;
100.        //Intra16x16宏块编码-需要Hadamard变换
101.        //分别编码Y, U, V
102.        /*
103.         * 16x16 宏块
104.         *
105.         * +-----+-----+
106.         * |               |
107.         * |               |
108.         * |               |
109.         * +       +       +
110.         * |               |
111.         * |               |
112.         * |               |
113.         * +-----+-----+
114.         */
115.        /*
116.         for( int p = 0; p < plane_count; p++, i_qp = h->mb.i_chroma_qp )
117.             x264_mb_encode_i16x16( h, p, i_qp );
118.         */
119.        else if( h->mb.i_type == I_8x8 )
120.        {
121.            h->mb.b_transform_8x8 = 1;
122.            /* If we already encoded 3 of the 4 i8x8 blocks, we don't have to do them again. */
123.            if( h->mb.i_skip_intra )
124.            {
125.                h->mc.copy[PIXEL_16x16]( h->mb.pic.p_fdec[0], FDEC_STRIDE, h->mb.pic.i8x8_fdec_buf, 16, 16 );
126.                M32( &h->mb.cache.non_zero_count[x264_scan8[ 0]] ) = h->mb.pic.i8x8_nnz_buf[0];
127.                M32( &h->mb.cache.non_zero_count[x264_scan8[ 2]] ) = h->mb.pic.i8x8_nnz_buf[1];
128.                M32( &h->mb.cache.non_zero_count[x264_scan8[ 8]] ) = h->mb.pic.i8x8_nnz_buf[2];
129.                M32( &h->mb.cache.non_zero_count[x264_scan8[10]] ) = h->mb.pic.i8x8_nnz_buf[3];
130.                h->mb.i_cbp_luma = h->mb.pic.i8x8_cbp;
131.                /* In RD mode, restore the now-overwritten DCT data. */
132.                if( h->mb.i_skip_intra == 2 )
133.                    h->mc.memcpy_aligned( h->dict.luma8x8, h->mb.pic.i8x8_dct_buf, sizeof(h->mb.pic.i8x8_dct_buf) );
134.            }
135.            for( int p = 0; p < plane_count; p++, i_qp = h->mb.i_chroma_qp )

```

```

136.     {
137.         for( int i = (p == 0 && h->mb.i_skip_intra) ? 3 : 0 ; i < 4; i++ )
138.         {
139.             int i_mode = h->mb.cache.intra4x4_pred_mode[x264_scan8[4*i]];
140.             x264_mb_encode_i8x8( h, p, i, i_qp, i_mode, NULL, 1 );
141.         }
142.     }
143. }
144. //Intra4x4类型
145. else if( h->mb.i_type == I_4x4 )
146. {
147.     /*
148.      * 帧内预测：16x16 宏块被划分为16个4x4子块
149.      *
150.      * +---+---+---+---+
151.      * |   |   |   |   |
152.      * +---+---+---+---+
153.      * |   |   |   |   |
154.      * +---+---+---+---+
155.      * |   |   |   |   |
156.      * +---+---+---+---+
157.      * |   |   |   |   |
158.      * +---+---+---+---+
159.      *
160.      */
161.     h->mb.b_transform_8x8 = 0;
162.     /* If we already encoded 15 of the 16 i4x4 blocks, we don't have to do them again. */
163.     if( h->mb.i_skip_intra )
164.     {
165.         h->mc.copy[PIXEL_16x16]( h->mb.pic.p_fdec[0], FDEC_STRIDE, h->mb.pic.i4x4_fdec_buf, 16, 16 );
166.         M32( &h->mb.cache.non_zero_count[x264_scan8[ 0]] ) = h->mb.pic.i4x4_nnz_buf[0];
167.         M32( &h->mb.cache.non_zero_count[x264_scan8[ 2]] ) = h->mb.pic.i4x4_nnz_buf[1];
168.         M32( &h->mb.cache.non_zero_count[x264_scan8[ 8]] ) = h->mb.pic.i4x4_nnz_buf[2];
169.         M32( &h->mb.cache.non_zero_count[x264_scan8[10]] ) = h->mb.pic.i4x4_nnz_buf[3];
170.         h->mb.i_cbp_luma = h->mb.pic.i4x4_cbp;
171.         /* In RD mode, restore the now-overwritten DCT data. */
172.         if( h->mb.i_skip_intra == 2 )
173.             h->mc.memcpy_aligned( h-> dct.luma4x4, h->mb.pic.i4x4_dct_buf, sizeof(h->mb.pic.i4x4_dct_buf) );
174.     }
175.     //分别编码Y,U,V
176.     for( int p = 0; p < plane_count; p++, i_qp = h->mb.i_chroma_qp )
177.     {
178.         //循环16次，编码16个Intra4x4宏块
179.         for( int i = (p == 0 && h->mb.i_skip_intra) ? 15 : 0 ; i < 16; i++ )
180.         {
181.             pixel *p_dst = &h->mb.pic.p_fdec[p][block_idx_xy_fdec[i]];
182.             int i_mode = h->mb.cache.intra4x4_pred_mode[x264_scan8[i]];
183.
184.             if( (h->mb.i_neighbour4[i] & (MB_TOPRIGHT|MB_TOP)) == MB_TOP )
185.                 /* emulate missing topright samples */
186.                 MPIXEL_X4( &p_dst[4-FDEC_STRIDE] ) = PIXEL_SPLAT_X4( p_dst[3-FDEC_STRIDE] );
187.             //Intra4x4宏块编码
188.             /*
189.              * +-----+
190.              * |   |   |
191.              * +-----+
192.              */
193.             x264_mb_encode_i4x4( h, p, i, i_qp, i_mode, 1 );
194.         }
195.     }
196. }
197. //包含帧间预测
198. else /* Inter MB */
199. {
200.     int i_decimate_mb = 0;
201.
202.     /* Don't repeat motion compensation if it was already done in non-RD transform analysis */
203.     if( !h->mb.b_skip_mc )
204.         x264_mb_mc( h );
205.
206.     if( h->mb.b_lossless )//lossless情况没研究过
207.     {
208.         if( h->mb.b_transform_8x8 )
209.             for( int p = 0; p < plane_count; p++ )
210.                 for( int i8x8 = 0; i8x8 < 4; i8x8++ )
211.                 {
212.                     int x = i8x8&1;
213.                     int y = i8x8>>1;
214.                     nz = h->zigzagf.sub_8x8( h->dct.luma8x8[p*4+i8x8], h->mb.pic.p_fenc[p] + 8*x + 8*y*FENC_STRIDE,
215.                                             h->mb.pic.p_fdec[p] + 8*x + 8*y*FDEC_STRIDE );
216.                     STORE_8x8_NNZ( p, i8x8, nz );
217.                     h->mb.i_cbp_luma |= nz << i8x8;
218.                 }
219.         else
220.             for( int p = 0; p < plane_count; p++ )
221.                 for( int i4x4 = 0; i4x4 < 16; i4x4++ )
222.                 {
223.                     nz = h->zigzagf.sub_4x4( h->dct.luma4x4[p*16+i4x4],
224.                                             h->mb.pic.p_fenc[p]+block_idx_xy_fenc[i4x4],
225.                                             h->mb.pic.p_fdec[p]+block_idx_xy_fdec[i4x4] );
226.                     h->mb.cache.non_zero_count[x264_scan8[p*16+i4x4]] = nz;
227.                     h->mb.i_cbp_luma |= nz << (i4x4>>2);

```

```

227.         h->mb.i_cbp_luma |= 1 << 8; (1+8+7+6+5+4+3+2+1);
228.     }
229. }
230. else if( h->mb.b_transform_8x8 )//DCT8x8情况暂时没研究过
231. {
232.     ALIGNED_ARRAY_N( dctcoef, dct8x8,[4],[64] );
233.     b_decimate &= !h->mb.b_trellis || !h->param.b_cabac; // 8x8 trellis is inherently optimal decimation for CABAC
234.
235.     for( int p = 0; p < plane_count; p++, i_qp = h->mb.i_chroma_qp )
236.     {
237.         CLEAR_16x16_NNZ( p );
238.         h->dctf.sub16x16_dct8( dct8x8, h->mb.pic.p_fenc[p], h->mb.pic.p_fdec[p] );
239.         h->nr_count[1+!!p*2] += h->mb.b_noise_reduction * 4;
240.
241.         int plane_cbp = 0;
242.         for( int idx = 0; idx < 4; idx++ )
243.         {
244.             nz = x264_quant_8x8( h, dct8x8[idx], i_qp, ctx_cat_plane[DCT_LUMA_8x8][p], 0, p, idx );
245.
246.             if( nz )
247.             {
248.                 h->zigzagf.scan_8x8( h->dct.luma8x8[p*4+idx], dct8x8[idx] );
249.                 if( b_decimate )
250.                 {
251.                     int i_decimate_8x8 = h->quantf.decimate_score64( h->dct.luma8x8[p*4+idx] );
252.                     i_decimate_mb += i_decimate_8x8;
253.                     if( i_decimate_8x8 >= 4 )
254.                         plane_cbp |= 1<<idx;
255.                 }
256.                 else
257.                     plane_cbp |= 1<<idx;
258.             }
259.         }
260.
261.         if( i_decimate_mb >= 6 || !b_decimate )
262.         {
263.             h->mb.i_cbp_luma |= plane_cbp;
264.             FOREACH_BIT( idx, 0, plane_cbp )
265.             {
266.                 h->quantf.dequant_8x8( dct8x8[idx], h->dequant8_mf[p?QOM_8PC:QOM_8PY], i_qp );
267.                 h->dctf.add8x8_idct8( &h->mb.pic.p_fdec[p][8*(idx&1) + 8*(idx>=1)*FDEC_STRIDE], dct8x8[idx] );
268.                 STORE_8x8_NNZ( p, idx, 1 );
269.             }
270.         }
271.     }
272. }
273. else//最普通的情况
274. {
275.     /*
276.     * 帧间预测：16x16 宏块被划分为8x8
277.     * 每个8x8再次被划分为4x4
278.     *
279.     * ++++++
280.     * || | | || | |
281.     * ++++++
282.     * || | | || | |
283.     * ++++++
284.     * || | | || | |
285.     * ++++++
286.     * || | | || | |
287.     * ++++++
288.     */
289.     /*
290.     ALIGNED_ARRAY_N( dctcoef, dct4x4,[16],[16] );
291.     for( int p = 0; p < plane_count; p++, i_qp = h->mb.i_chroma_qp )
292.     {
293.         CLEAR_16x16_NNZ( p );
294.         //16x16DCT (实际上分解为16个4x4DCT)
295.         //求编码帧p_fenc和重建帧p_fdec之间的残差, 然后进行DCT变换
296.         h->dctf.sub16x16_dct( dct4x4, h->mb.pic.p_fenc[p], h->mb.pic.p_fdec[p] );
297.
298.         if( h->mb.b_noise_reduction )
299.         {
300.             h->nr_count[0+!!p*2] += 16;
301.             for( int idx = 0; idx < 16; idx++ )
302.                 h->quantf.denoise_dct( dct4x4[idx], h->nr_residual_sum[0+!!p*2], h->nr_offset[0+!!p*2], 16 );
303.         }
304.
305.         int plane_cbp = 0;
306.         //16x16的块分成4个8x8的块
307.         for( int i8x8 = 0; i8x8 < 4; i8x8++ )
308.         {
309.             int i_decimate_8x8 = b_decimate ? 0 : 6;
310.             int nnz8x8 = 0;
311.             if( h->mb.b_trellis )
312.             {
313.                 for( int i4x4 = 0; i4x4 < 4; i4x4++ )
314.                 {
315.                     int idx = i8x8*4+i4x4;
316.                     if( x264_quant_4x4_trellis( h, dct4x4[idx], QOM_4PY, i_qp, ctx_cat_plane[DCT_LUMA_4x4]
[p], 0, !!p, p*16+idx ) )
317.

```

```

318.         h->zigzagf.scan_4x4( h-> dct.luma4x4[p*16+idx], dct4x4[idx] );
319.         h-> quantf.dequant_4x4( dct4x4[idx], h-> dequant4_mf[p?CQM_4PC:CQM_4PY], i_qp );
320.         if( i_decimate_8x8 < 6 )
321.             i_decimate_8x8 += h-> quantf.decimate_score16( h-> dct.luma4x4[p*16+idx] );
322.         h-> mb.cache.non_zero_count[x264_scan8[p*16+idx]] = 1;
323.         nnz8x8 = 1;
324.     }
325. }
326. }
327. else
328. {
329.     //8x8的块分成4个4x4的块, 每个4x4的块再分别进行量化
330.     nnz8x8 = nz = h-> quantf.quant_4x4x4( &dct4x4[i8x8*4], h-> quant4_mf[CQM_4PY][i_qp], h-> quant4_bias[CQM_4PY][i_qp] );
331.     if( nz )
332.     {
333.         FOREACH_BIT( idx, i8x8*4, nz )
334.         {
335.             //这几步用于建立重建帧
336.             h-> zigzagf.scan_4x4( h-> dct.luma4x4[p*16+idx], dct4x4[idx] );
337.             //反量化
338.             h-> quantf.dequant_4x4( dct4x4[idx], h-> dequant4_mf[p?CQM_4PC:CQM_4PY], i_qp );
339.             if( i_decimate_8x8 < 6 )
340.                 i_decimate_8x8 += h-> quantf.decimate_score16( h-> dct.luma4x4[p*16+idx] );
341.             h-> mb.cache.non_zero_count[x264_scan8[p*16+idx]] = 1;
342.         }
343.     }
344. }
345. if( nnz8x8 )
346. {
347.     i_decimate_mb += i_decimate_8x8;
348.     if( i_decimate_8x8 < 4 )
349.         STORE_8x8_NNZ( p, i8x8, 0 );
350.     else
351.         plane_cbp |= 1<<i8x8;
352. }
353. }
354.
355. if( i_decimate_mb < 6 )
356. {
357.     plane_cbp = 0;
358.     CLEAR_16x16_NNZ( p );
359. }
360. else
361. {
362.     h-> mb.i_cbp_luma |= plane_cbp;
363.     FOREACH_BIT( i8x8, 0, plane_cbp )
364.     {
365.         //用于建立重建帧
366.         //残差进行DCT反变换之后, 叠加到预测数据上
367.         h-> dctf.add8x8_idct( &h-> mb.pic.p_fdec[p][(i8x8&1)*8 + (i8x8>>1)*8*FDEC_STRIDE], &dct4x4[i8x8*4] );
368.     }
369. }
370. }
371. }
372. }
373.
374. /* encode chroma */
375. if( chroma )
376. {
377.     if( IS_INTRA( h-> mb.i_type ) )
378.     {
379.         int i_mode = h-> mb.i_chroma_pred_mode;
380.         if( h-> mb.b_lossless )
381.             x264_predict_lossless_chroma( h, i_mode );
382.         else
383.         {
384.             h-> predict_chroma[i_mode]( h-> mb.pic.p_fdec[1] );
385.             h-> predict_chroma[i_mode]( h-> mb.pic.p_fdec[2] );
386.         }
387.     }
388.
389.     /* encode the 8x8 blocks */
390.     x264_mb_encode_chroma( h, !IS_INTRA( h-> mb.i_type ), h-> mb.i_chroma_qp );
391. }
392. else
393.     h-> mb.i_cbp_chroma = 0;
394.
395. /* store cbp */
396. int cbp = h-> mb.i_cbp_chroma << 4 | h-> mb.i_cbp_luma;
397. if( h-> param.b_cabac )
398.     cbp |= h-> mb.cache.non_zero_count[x264_scan8[LUMA_DC]] << 8
399.         | h-> mb.cache.non_zero_count[x264_scan8[CHROMA_DC+0]] << 9
400.         | h-> mb.cache.non_zero_count[x264_scan8[CHROMA_DC+1]] << 10;
401. h-> mb.cbp[h-> mb.i_mb_xy] = cbp;
402.
403. /* Check for P_SKIP
404.  * XXX: in the me perhaps we should take x264_mb_predict_mv_pskip into account
405.  *      (if multiple mv give same result)*/
406. if( !b_force_no_skip )
407. {

```

```

408.         if( h->mb.i_type == P_L0 && h->mb.i_partition == D_16x16 &&
409.            !(h->mb.i_cbp_luma | h->mb.i_cbp_chroma) &&
410.            M32( h->mb.cache.mv[0][x264_scan8[0]] ) == M32( h->mb.cache.pskip_mv )
411.            && h->mb.cache.ref[0][x264_scan8[0]] == 0 )
412.         {
413.             h->mb.i_type = P_SKIP;
414.         }
415.
416.         /* Check for B_SKIP */
417.         if( h->mb.i_type == B_DIRECT && !(h->mb.i_cbp_luma | h->mb.i_cbp_chroma) )
418.         {
419.             h->mb.i_type = B_SKIP;
420.         }
421.     }
422. }

```

从源代码可以看出，x264\_macroblock\_encode\_internal()的流程大致如下：

- (1) 如果是Skip类型，调用x264\_macroblock\_encode\_skip()编码宏块。
- (2) 如果是Intra16x16类型，调用x264\_mb\_encode\_i16x16()编码宏块。
- (3) 如果是Intra4x4类型，循环16次调用x264\_mb\_encode\_i4x4()编码宏块。
- (4) 如果是Inter类型，则不再调用子函数，而是直接进行编码：
  - a) 对16x16块调用x264\_dct\_function\_t的sub16x16\_dct()汇编函数，求得编码宏块数据p\_fenc与重建宏块数据p\_fdec之间的残差（“sub”），并对残差进行DCT变换。
  - b) 分成4个8x8的块，对每个8x8块分别调用x264\_quant\_function\_t的quant\_4x4x4()汇编函数进行量化。
  - c) 分成16个4x4的块，对每个4x4块分别调用x264\_quant\_function\_t的dequant\_4x4()汇编函数进行反量化（用于重建帧）。
  - d) 分成4个8x8的块，对每个8x8块分别调用x264\_dct\_function\_t的add8x8\_idct()汇编函数，对残差进行DCT反变换，并将反变换后的数据叠加（“add”）至预测数据上（用于重建帧）。
- (5)

如果对色度编码，调用x264\_mb\_encode\_chroma()。

从Inter宏块编码的步骤可以看出，编码就是“DCT变换+量化”两步的组合。下文将会按照顺序记录x264\_macroblock\_encode\_skip()，x264\_mb\_encode\_i16x16()，x264\_mb\_encode\_i4x4()三个函数。

## x264\_macroblock\_encode\_skip()

x264\_macroblock\_encode\_skip()用于编码Skip宏块。该函数的定义位于encoder\macroblock.c，如下所示。

```

1. //编码skip类型宏块
2. static void x264_macroblock_encode_skip( x264_t *h )
3. {
4.     /*
5.      * YUV420P的时候在这里相当于在non_zero_count[]填充了v (v=0) :
6.      * YUV422P, YUV444P的时候填充了w (w=0)
7.      * |
8.      * +-----+
9.      * | 0 0 0 0 0 0 0 0
10.     * | 0 0 0 0 v v v v
11.     * | 0 0 0 0 v v v v
12.     * | 0 0 0 0 v v v v
13.     * | 0 0 0 0 v v v v
14.     * | 0 0 0 0 0 0 0 0
15.     * | 0 0 0 0 v v v v
16.     * | 0 0 0 0 v v v v
17.     * | 0 0 0 0 w w w w
18.     * | 0 0 0 0 w w w w
19.     * | 0 0 0 0 0 0 0 0
20.     * | 0 0 0 0 v v v v
21.     * | 0 0 0 0 v v v v
22.     * | 0 0 0 0 w w w w
23.     * | 0 0 0 0 w w w w
24.     */
25.     //填充non_zero_count[]
26.     M32( &h->mb.cache.non_zero_count[x264_scan8[ 0]] ) = 0;
27.     M32( &h->mb.cache.non_zero_count[x264_scan8[ 2]] ) = 0;
28.     M32( &h->mb.cache.non_zero_count[x264_scan8[ 8]] ) = 0;
29.     M32( &h->mb.cache.non_zero_count[x264_scan8[10]] ) = 0;
30.     M32( &h->mb.cache.non_zero_count[x264_scan8[16+ 0]] ) = 0;
31.     M32( &h->mb.cache.non_zero_count[x264_scan8[16+ 2]] ) = 0;
32.     M32( &h->mb.cache.non_zero_count[x264_scan8[32+ 0]] ) = 0;
33.     M32( &h->mb.cache.non_zero_count[x264_scan8[32+ 2]] ) = 0;
34.     if( CHROMA_FORMAT >= CHROMA_422 )
35.     {
36.         M32( &h->mb.cache.non_zero_count[x264_scan8[16+ 8]] ) = 0;
37.         M32( &h->mb.cache.non_zero_count[x264_scan8[16+10]] ) = 0;
38.         M32( &h->mb.cache.non_zero_count[x264_scan8[32+ 8]] ) = 0;
39.         M32( &h->mb.cache.non_zero_count[x264_scan8[32+10]] ) = 0;
40.     }
41.     //CBP也赋值为0, 即不对亮度和色度编码
42.     h->mb.i_cbp_luma = 0;
43.     h->mb.i_cbp_chroma = 0;
44.     h->mb.cbp[h->mb.i_mb_xy] = 0;
45. }

```

从源代码可以看出，x264\_macroblock\_encode\_skip()的逻辑比较简单，就是将宏块的DCT非零系数缓存non\_zero\_count[]设置成了0，并且将宏块的CBP也设置为0（代表没有残差信息）。

## x264\_mb\_encode\_i16x16()

x264\_mb\_encode\_i16x16()用于编码Intra16x16的宏块。该函数的定义位于encoder\macroblock.c,如下所示。

```

1. //编码I16x16宏块-需要Hadamard变换
2. /*
3.  * 16x16 宏块
4.  *
5.  * +-----+-----+
6.  * |           |           D   D   D   D
7.  * |           |           D   D   D   D
8.  * |           |           D   D   D   D
9.  * +           +           +   +
10. * |           |           D   D   D   D
11. * |           |           D   D   D   D
12. * |           |           D   D   D   D
13. * +-----+-----+
14. *
15. */
16. //p代表分量
17. static void x264_mb_encode_i16x16( x264_t *h, int p, int i_qp )
18. {
19.     //编码帧
20.     pixel *p_src = h->mb.pic.p_fenc[p];
21.     //重建帧
22.     pixel *p_dst = h->mb.pic.p_fdec[p];
23.
24.     ALIGNED_ARRAY_N( dctcoef, dct4x4,[16],[16] );
25.     ALIGNED_ARRAY_N( dctcoef, dct_dc4x4,[16] );
26.
27.     int nz, block_cbp = 0;
28.     int decimate_score = h->mb.b_dct_decimate ? 0 : 9;
29.     int i_quant_cat = p ? CQM_4IC : CQM_4IY;
30.     int i_mode = h->mb.i_intra16x16_pred_mode;

```

23



```

31.
32.     if( h->mb.b_lossless )
33.         x264_predict_lossless_16x16( h, p, i_mode );
34.     else
35.         h->predict_16x16[i_mode]( h->mb.pic.p_fdec[p] );    //帧内预测.p_fdec是重建帧。p_fenc是编码帧。
36.
37.     if( h->mb.b_lossless )
38.     {
39.         for( int i = 0; i < 16; i++ )
40.         {
41.             int oe = block_idx_xy_fenc[i];
42.             int od = block_idx_xy_fdec[i];
43.             nz = h->zigzagf.sub_4x4ac( h-> dct.luma4x4[16*p+i], p_src+oe, p_dst+od, &dct_dc4x4[block_idx_yx_1d[i]] );
44.             h->mb.cache.non_zero_count[x264_scan8[16*p+i]] = nz;
45.             block_cbp |= nz;
46.         }
47.         h->mb.i_cbp_luma |= block_cbp * 0xf;
48.         h->mb.cache.non_zero_count[x264_scan8[LUMA_DC+p]] = array_non_zero( dct_dc4x4, 16 );
49.         h->zigzagf.scan_4x4( h-> dct.luma16x16_dc[p], dct_dc4x4 );
50.         return;
51.     }
52.
53.     CLEAR_16x16_NNZ( p );
54.
55.     h->dctf.sub16x16_dct( dct4x4, p_src, p_dst );    //求残差, 然后进行DCT变换
56.
57.     if( h->mb.b_noise_reduction )
58.         for( int idx = 0; idx < 16; idx++ )
59.             h->quantf.denoise_dct( dct4x4[idx], h->nr_residual_sum[0], h->nr_offset[0], 16 );
60.     //获取DC系数
61.     for( int idx = 0; idx < 16; idx++ )
62.     {
63.         //每个4x4DCT块的[0]元素
64.         dct_dc4x4[block_idx_xy_1d[idx]] = dct4x4[idx][0];
65.         //抽取出来之后, 赋值0
66.         dct4x4[idx][0] = 0;
67.     }
68.
69.     if( h->mb.b_trellis )
70.     {
71.         for( int idx = 0; idx < 16; idx++ )
72.             if( x264_quant_4x4_trellis( h, dct4x4[idx], i_quant_cat, i_qp, ctx_cat_plane[DCT_LUMA_AC][p], 1, !!p, idx ) )
73.             {
74.                 block_cbp = 0xf;
75.                 h->zigzagf.scan_4x4( h-> dct.luma4x4[16*p+idx], dct4x4[idx] );
76.                 h->quantf.dequant_4x4( dct4x4[idx], h->dequant4_mf[i_quant_cat], i_qp );
77.                 if( decimate_score < 6 ) decimate_score += h->quantf.decimate_score15( h-> dct.luma4x4[16*p+idx] );
78.                 h->mb.cache.non_zero_count[x264_scan8[16*p+idx]] = 1;
79.             }
80.     }
81.     else
82.     {
83.         //先分成4个8x8?
84.         for( int i8x8 = 0; i8x8 < 4; i8x8++ )
85.         {
86.             //每个8x8做4个4x4量化
87.             nz = h->quantf.quant_4x4x4( &dct4x4[i8x8*4], h->quant4_mf[i_quant_cat][i_qp], h->quant4_bias[i_quant_cat][i_qp] );
88.             if( nz )
89.             {
90.                 block_cbp = 0xf;
91.                 FOREACH_BIT( idx, i8x8*4, nz )
92.                 {
93.                     //建立重建的帧
94.                     //之子扫描
95.                     h->zigzagf.scan_4x4( h-> dct.luma4x4[16*p+idx], dct4x4[idx] );
96.                     //反量化, 用于重建图像
97.                     h->quantf.dequant_4x4( dct4x4[idx], h->dequant4_mf[i_quant_cat], i_qp );
98.                     if( decimate_score < 6 ) decimate_score += h->quantf.decimate_score15( h-> dct.luma4x4[16*p+idx] );
99.                     h->mb.cache.non_zero_count[x264_scan8[16*p+idx]] = 1;
100.                 }
101.             }
102.         }
103.     }
104.
105.     /* Writing the 16 CBFs in an i16x16 block is quite costly, so decimation can save many bits. */
106.     /* More useful with CAVLC, but still useful with CABAC. */
107.     if( decimate_score < 6 )
108.     {
109.         CLEAR_16x16_NNZ( p );
110.         block_cbp = 0;
111.     }
112.     else
113.         h->mb.i_cbp_luma |= block_cbp;
114.
115.     //16个DC系数-Hadamard变换
116.     h->dctf.dct4x4dc( dct_dc4x4 );
117.     if( h->mb.b_trellis )
118.         nz = x264_quant_luma_dc_trellis( h, dct_dc4x4, i_quant_cat, i_qp, ctx_cat_plane[DCT_LUMA_DC][p], 1, LUMA_DC+p );
119.     else
120.         //DC-Hadamard变换之后-量化
121.         nz = h->quantf.quant_4x4_dc( dct_dc4x4, h->quant4_mf[i_quant_cat][i_qp][0]>>1, h->quant4_bias[i_quant_cat][i_qp][0]<<1 );
122.

```

```

123.     h->mb.cache.non_zero_count[x264_scan8[LUMA_DC+p]] = nz;
124.     if( nz )
125.     {
126.
127.         //之子扫描
128.         h->zigzagf.scan_4x4( h-> dct.luma16x16_dc[p], dct_dc4x4 );
129.
130.         /* output samples to fdec */
131.         //DC-反变换
132.         h->dctf.idct4x4dc( dct_dc4x4 );
133.         //DC-反量化
134.         h->quantf.dequant_4x4_dc( dct_dc4x4, h->dequant4_mf[i_quant_cat], i_qp ); /* XXX not inversed */
135.         if( block_cbp )
136.             for( int i = 0; i < 16; i++ )//循环16个4x4DCT块
137.                 dct4x4[i][0] = dct_dc4x4[block_idx_xy_1d[i]]; //把DC系数重新赋值到每个DCT数组的[0]元素上
138.     }
139.
140.     /* put pixels to fdec */
141.     // fdec代表重建帧
142.     if( block_cbp )
143.         h->dctf.add16x16_idct( p_dst, dct4x4 ); //DCT反变换后，叠加到预测数据上（通用）
144.     else if( nz )
145.         h->dctf.add16x16_idct_dc( p_dst, dct_dc4x4 ); //DCT反变换后，叠加到预测数据上（只有DC系数的时候）
146. }

```

简单整理一下x264\_mb\_encode\_i16x16()的逻辑，如下所示：

- (1) 调用predict\_16x16()汇编函数对重建宏块数据p\_fdec进行帧内预测。
- (2) 调用x264\_dct\_function\_t的sub16x16\_dct()汇编函数，计算重建宏块数据p\_fdec与编码宏块数据p\_fenc之间的残差，然后对残差做DCT变换。
- (3) 抽取出来16个4x4DCT小块的DC系数，存储于dct\_dc4x4[]。
- (4) 分成4个8x8的块，对每个8x8块分别调用x264\_quant\_function\_t的quant\_4x4x4()汇编函数进行量化。
- (5) 分成16个4x4的块，对每个4x4块分别调用x264\_quant\_function\_t的dequant\_4x4()汇编函数进行反量化（用于重建帧）。
- (6) 对于dct\_dc4x4[]中16个小块的DC系数作如下处理：
  - a)调用x264\_dct\_function\_t的dct4x4dc()汇编函数进行Hadamard变换。
  - b)调用x264\_quant\_function\_t的quant\_4x4\_dc()汇编函数进行DC系数的量化。
  - c)调用x264\_dct\_function\_t的idct4x4dc()汇编函数进行Hadamard反变换。
  - d)调用x264\_quant\_function\_t的dequant\_4x4\_dc()汇编函数进行DC系数的反量化。
  - e)将反量化后的DC系数重新放到16x16块对应的位置上。
- (7) 调用x264\_dct\_function\_t的add16x16\_idct()汇编函数，对残差进行DCT反变换，并将反变换后的数据叠加（“add”）至预测数据上（用于重建帧）。

可以看出Intra16x16编码的过程就是一个“DCT变换 + 量化 + Hadamard变换”的流程。其中“DCT变换 + 量化”是一个通用的编码步骤，而“Hadamard变换”是专属于Intra16x16宏块的步骤。

## x264\_mb\_encode\_i4x4()

x264\_mb\_encode\_i4x4()用于编码Intra4x4的宏块。该函数的定义位于encoder\macroblock.c，如下所示。

```

1. //编码Intra4x4
2. /*
3.  * +----+
4.  * |    |
5.  * +----+
6.  */
7. static ALWAYS_INLINE void x264_mb_encode_i4x4( x264_t *h, int p, int idx, int i_qp, int i_mode, int b_predict )
8. {
9.     int nz;
10.    //编码帧
11.    pixel *p_src = &h->mb.pic.p_fenc[p][block_idx_xy_fenc[idx]];
12.    //重建帧
13.    pixel *p_dst = &h->mb.pic.p_fdec[p][block_idx_xy_fdec[idx]];
14.    ALIGNED_ARRAY_N( dctcoef, dct4x4,[16] );
15.
16.    if( b_predict )
17.    {
18.        if( h->mb.b_lossless )
19.            x264_predict_lossless_4x4( h, p_dst, p, idx, i_mode );
20.        else
21.            h->predict_4x4[i_mode]( p_dst );//帧内预测, 存于p_dst
22.    }
23.
24.    if( h->mb.b_lossless )
25.    {
26.        nz = h->zigzagf.sub_4x4( h->dct.luma4x4[p*16+idx], p_src, p_dst );
27.        h->mb.cache.non_zero_count[x264_scan8[p*16+idx]] = nz;
28.        h->mb.i_cbp_luma |= nz<<(idx>>2);
29.        return;
30.    }
31.
32.    h->dctf.sub4x4_dct( dct4x4, p_src, p_dst );//求p_src与p_dst之间的残差, 并且进行DCT变换
33.    //量化
34.    nz = x264_quant_4x4( h, dct4x4, i_qp, ctx_cat_plane[DCT_LUMA_4x4][p], 1, p, idx );
35.    h->mb.cache.non_zero_count[x264_scan8[p*16+idx]] = nz;
36.    if( nz )
37.    {
38.        //解码并且建立重建帧 (p_dst)
39.        h->mb.i_cbp_luma |= 1<<(idx>>2);
40.        //DCT系数重新排个序-从之子扫描变换为普通扫描
41.        h->zigzagf.scan_4x4( h->dct.luma4x4[p*16+idx], dct4x4 );
42.        //反量化
43.        h->quantf.dequant_4x4( dct4x4, h->dequant4_mf[p?CQM_4IC:CQM_4IY], i_qp );
44.        //DCT残差反变换, 并且叠加到预测数据上, 形成重建帧
45.        h->dctf.add4x4_idct( p_dst, dct4x4 );
46.    }
47. }

```

简单整理一下x264\_mb\_encode\_i4x4()的逻辑, 如下所示：

- (1) 调用predict\_4x4[]()汇编函数对重建宏块数据p\_fdec进行帧内预测。
- (2) 调用x264\_dct\_function\_t的sub4x4\_dct()汇编函数, 计算重建宏块数据p\_fdec与编码宏块数据p\_fenc之间的残差, 然后对残差做DCT变换。
- (3) 调用x264\_quant\_function\_t的quant\_4x4()汇编函数进行量化。
- (4) 调用x264\_quant\_function\_t的dequant\_4x4()汇编函数进行反量化（用于重建帧）。
- (5) 调用x264\_dct\_function\_t的add4x4\_idct()汇编函数, 对残差进行DCT反变换, 并将反变换后的数据叠加（“add”）至预测数据上（用于重建帧）。

可以看出Intra4x4编码的过程就是一个“DCT变换 + 量化”的流程。

## DCT和量化的知识

宏块的编码过程就是一个“DCT变换+量化”的过程。简单记录一下相关的知识。

### DCT变换

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

□

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

□

4x4整数DCT变换的公式如下所示。

□

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

同理，DCT反变换就是DCT变换的逆变换。DCT反变换的公式如下所示。

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

除了4x4DCT变换之外，新版本的H.264标准中还引入了一种8x8DCT。目前针对这种8x8DCT我还没有做研究，暂时不做记录。

## 量化

量化是H.264视频压缩编码中对视频质量影响最大的地方，也是会导致“信息丢失”的地方。量化的原理可以表示为下面公式：

$$FQ=\text{round}(y/Q\text{step})$$

其中，y 为输入样本点编码，Qstep为量化步长，FQ 为y 的量化值，round()为取整函数（其输出为与输入实数最近的整数）。其相反过程，即反量化为：

$$y'=FQ/Q\text{step}$$

如果Qstep较大，则量化值FQ取值较小，其相应的编码长度较小，但是但反量化时损失较多的图像细节信息。简而言之，Qstep越大，视频压缩编码后体积越小，视频质量越差。

在H.264 中，量化步长Qstep 共有52 个值，如下表所示。其中QP 是量化参数，是量化步长的序号。当QP 取最小值0 时代表最精细的量化，当QP 取最大值51 时代表最粗糙的量化。QP 每增加6，Qstep 增加一倍。

《H.264标准》中规定，量化过程除了完成本职工作外，还需要完成它前一步DCT变换中“系数相乘”的工作。这一步骤的推导过程不再记录，直接给出最终的公式（这个公式完全为整数运算，同时避免了除法的使用）：

$$\begin{aligned} |Z_{ij}| &= (|W_{ij}|*MF + f) \gg \text{qbits} \\ \text{sign}(Z_{ij}) &= \text{sign}(W_{ij}) \end{aligned}$$

其中：

sign()为符号函数。

Wij为DCT变换后的系数。

MF的值如下表所示。表中只列出对应QP 值为0 到5 的MF 值。QP大于6之后，将QP实行对6取余数操作，再找到MF的值。

qbits计算公式为“qbits = 15 + floor(QP/6)”。即它的值随QP 值每增加6 而增加1。

f 是偏移量（用于改善恢复图像的视觉效果）。对帧内预测图像块取2^qbits/3，对帧间预测图像块取2^qbits/6。

为了更形象的显示MF的取值，做了下面一张示意图。图中深蓝色代表MF取值较大的点，而浅蓝色代表MF取值较小的点。

## DCT相关的源代码

DCT模块的初始化函数是x264\_dct\_init()。该函数对x264\_dct\_function\_t结构体中的函数指针进行了赋值。X264运行的过程中只要调用x264\_dct\_function\_t的函数指针就可以完成相应的功能。

### x264\_dct\_init()

x264\_dct\_init()用于初始化DCT变换和DCT反变换相关的汇编函数。该函数的定义位于common\dct.c，如下所示。

[cpp]  

```
1.  /*****
2.  * x264_dct_init:
3.  *****/
4.  void x264_dct_init( int cpu, x264_dct_function_t *dctf )
5.  {
6.      //C语言版本
7.      //4x4DCT变换
8.      dctf->sub4x4_dct = sub4x4_dct;
9.      dctf->add4x4_idct = add4x4_idct;
10.     //8x8块：分解成4个4x4DCT变换，调用4次sub4x4_dct()
11.     dctf->sub8x8_dct = sub8x8_dct;
12.     dctf->sub8x8_dct_dc = sub8x8_dct_dc;
13.     dctf->add8x8_idct = add8x8_idct;
14.     dctf->add8x8_idct_dc = add8x8_idct_dc;
15.
16.     dctf->sub8x16_dct_dc = sub8x16_dct_dc;
17.     //16x16块：分解成4个8x8块，调用4次sub8x8_dct()
18.     //实际上每个sub8x8_dct()又分解成4个4x4DCT变换，调用4次sub4x4_dct()
19.     dctf->sub16x16_dct = sub16x16_dct;
20.     dctf->add16x16_idct = add16x16_idct;
21.     dctf->add16x16_idct_dc = add16x16_idct_dc;
22.     //8x8DCT，注意：后缀是_dct8
23.     dctf->sub8x8_dct8 = sub8x8_dct8;
24.     dctf->add8x8_idct8 = add8x8_idct8;
25.
26.     dctf->sub16x16_dct8 = sub16x16_dct8;
27.     dctf->add16x16_idct8 = add16x16_idct8;
28.     //Hadamard变换
29.     dctf->dct4x4dc = dct4x4dc;
30.     dctf->idct4x4dc = idct4x4dc;
31.
32.     dctf->dct2x4dc = dct2x4dc;
33.
34. #if HIGH_BIT_DEPTH
35. #if HAVE_MMX
36.     if( cpu & X264_CPU_MMX )
37.     {
38.         dctf->sub4x4_dct = x264_sub4x4_dct_mmx;
39.         dctf->sub8x8_dct = x264_sub8x8_dct_mmx;
40.         dctf->sub16x16_dct = x264_sub16x16_dct_mmx;
41.     }
42.     if( cpu & X264_CPU_SSE2 )
43.     {
44.         dctf->add4x4_idct = x264_add4x4_idct_sse2;
45.         dctf->dct4x4dc = x264_dct4x4dc_sse2;
46.         dctf->idct4x4dc = x264_idct4x4dc_sse2;
47.         dctf->sub8x8_dct8 = x264_sub8x8_dct8_sse2;
48.         dctf->sub16x16_dct8 = x264_sub16x16_dct8_sse2;
49.         dctf->add8x8_idct = x264_add8x8_idct_sse2;
50.         dctf->add16x16_idct = x264_add16x16_idct_sse2;
51.         dctf->add8x8_idct8 = x264_add8x8_idct8_sse2;
52.         dctf->add16x16_idct8 = x264_add16x16_idct8_sse2;
53.         dctf->sub8x8_dct_dc = x264_sub8x8_dct_dc_sse2;
54.         dctf->add8x8_idct_dc = x264_add8x8_idct_dc_sse2;
55.         dctf->sub8x16_dct_dc = x264_sub8x16_dct_dc_sse2;
56.         dctf->add16x16_idct_dc = x264_add16x16_idct_dc_sse2;
57.     }
58.     if( cpu & X264_CPU_SSE4 )
59.     {
60.         dctf->sub8x8_dct8 = x264_sub8x8_dct8_sse4;
61.         dctf->sub16x16_dct8 = x264_sub16x16_dct8_sse4;
62.     }
63.     if( cpu & X264_CPU_AVX )
64.     {
65.         dctf->add4x4_idct = x264_add4x4_idct_avx;
66.         dctf->dct4x4dc = x264_dct4x4dc_avx;
67.         dctf->idct4x4dc = x264_idct4x4dc_avx;
68.         dctf->sub8x8_dct8 = x264_sub8x8_dct8_avx;
69.         dctf->sub16x16_dct8 = x264_sub16x16_dct8_avx;
70.         dctf->add8x8_idct = x264_add8x8_idct_avx;
71.         dctf->add16x16_idct = x264_add16x16_idct_avx;
72.         dctf->add8x8_idct8 = x264_add8x8_idct8_avx;
73.         dctf->add16x16_idct8 = x264_add16x16_idct8_avx;
74.         dctf->add8x8_idct_dc = x264_add8x8_idct_dc_avx;
75.         dctf->sub8x16_dct_dc = x264_sub8x16_dct_dc_avx;
76.         dctf->add16x16_idct_dc = x264_add16x16_idct_dc_avx;
77.     }
78. #endif // HAVE_MMX
79. #else // !HIGH_BIT_DEPTH
80.     //MMX版本
81. #if HAVE_MMX
82.     if( cpu & X264_CPU_MMX )
83.     {
84.         dctf->sub4x4_dct = x264_sub4x4_dct_mmx;
85.         dctf->add4x4_idct = x264_add4x4_idct_mmx;
86.         dctf->idct4x4dc = x264_idct4x4dc_mmx;
87.         dctf->sub8x8_dct_dc = x264_sub8x8_dct_dc_mmx2;
88.         //此处省略大量的X86、ARM等平台的汇编函数初始化代码
89.     }

```

从源代码可以看出，x264\_dct\_init()初始化了一系列的DCT变换的函数，这些DCT函数名称有如下规律：

- (1) DCT函数名称前面有“sub”，代表对两块像素相减得到残差之后，再进行DCT变换。
- (2) DCT反变换函数名称前面有“add”，代表将DCT反变换之后的残差数据叠加到预测数据上。
- (3) 以“dct8”为结尾的函数使用了8x8DCT（未研究过），其余函数是用的都是4x4DCT。

x264\_dct\_init()的输入参数x264\_dct\_function\_t是一个结构体，其中包含了各种DCT函数的接口。x264\_dct\_function\_t的定义如下所示。

```
[cpp]
1.  typedef struct
2.  {
3.      // pix1 stride = FENC_STRIDE
4.      // pix2 stride = FDEC_STRIDE
5.      // p_dst stride = FDEC_STRIDE
6.      void (*sub4x4_dct) ( dctcoef dct[16], pixel *pix1, pixel *pix2 );
7.      void (*add4x4_idct) ( pixel *p_dst, dctcoef dct[16] );
8.
9.      void (*sub8x8_dct) ( dctcoef dct[4][16], pixel *pix1, pixel *pix2 );
10.     void (*sub8x8_dct_dc)( dctcoef dct[4], pixel *pix1, pixel *pix2 );
11.     void (*add8x8_idct) ( pixel *p_dst, dctcoef dct[4][16] );
12.     void (*add8x8_idct_dc) ( pixel *p_dst, dctcoef dct[4] );
13.
14.     void (*sub8x16_dct_dc)( dctcoef dct[8], pixel *pix1, pixel *pix2 );
15.
16.     void (*sub16x16_dct) ( dctcoef dct[16][16], pixel *pix1, pixel *pix2 );
17.     void (*add16x16_idct)( pixel *p_dst, dctcoef dct[16][16] );
18.     void (*add16x16_idct_dc) ( pixel *p_dst, dctcoef dct[16] );
19.
20.     void (*sub8x8_dct8) ( dctcoef dct[64], pixel *pix1, pixel *pix2 );
21.     void (*add8x8_idct8) ( pixel *p_dst, dctcoef dct[64] );
22.
23.     void (*sub16x16_dct8) ( dctcoef dct[4][64], pixel *pix1, pixel *pix2 );
24.     void (*add16x16_idct8)( pixel *p_dst, dctcoef dct[4][64] );
25.
26.     void (*dct4x4dc) ( dctcoef d[16] );
27.     void (*idct4x4dc)( dctcoef d[16] );
28.
29.     void (*dct2x4dc)( dctcoef dct[8], dctcoef dct4x4[8][16] );
30.
31. } x264_dct_function_t;
```

x264\_dct\_init()的工作就是对x264\_dct\_function\_t中的函数指针进行赋值。由于DCT函数很多，不便于一一研究，下文仅举例分析几个典型的4x4DCT函数：4x4DCT变换函数sub4x4\_dct(), 4x4IDCT变换函数add4x4\_idct(), 8x8块的4x4DCT变换函数sub8x8\_dct(), 16x16块的4x4DCT变换函数sub16x16\_dct(), 4x4Hadamard变换函数dct4x4dc()。

## sub4x4\_dct()

sub4x4\_dct()可以将两块4x4的图像相减求残差后，进行DCT变换。该函数的定义位于common\dct.c，如下所示。

```

1.  /*
2.  * 求残差用
3.  * 注意求的是一个“方块”形像素
4.  *
5.  * 参数的含义如下：
6.  * diff：输出的残差数据
7.  * i_size：方块的大小
8.  * pix1：输入数据1
9.  * i_pix1：输入数据1一行像素大小（stride）
10. * pix2：输入数据2
11. * i_pix2：输入数据2一行像素大小（stride）
12. *
13. */
14. static inline void pixel_sub_wxh( dctcoef *diff, int i_size,
15.                                   pixel *pix1, int i_pix1, pixel *pix2, int i_pix2 )
16. {
17.     for( int y = 0; y < i_size; y++ )
18.     {
19.         for( int x = 0; x < i_size; x++ )
20.             diff[x + y*i_size] = pix1[x] - pix2[x]; //求残差
21.         pix1 += i_pix1; //前进到下一行
22.         pix2 += i_pix2;
23.     }
24. }
25. //4x4DCT变换
26. //注意首先获取pix1和pix2两块数据的残差，然后再进行变换
27. //返回dct[16]
28. static void sub4x4_dct( dctcoef dct[16], pixel *pix1, pixel *pix2 )
29. {
30.     dctcoef d[16];
31.     dctcoef tmp[16];
32.     //获取残差数据，存入d[16]
33.     //pix1一般为编码帧（enc）
34.     //pix2一般为重建帧（dec）
35.     pixel_sub_wxh( d, 4, pix1, FENC_STRIDE, pix2, FDEC_STRIDE );
36.
37.     //处理残差d[16]
38.     //蝶形算法：横向4个像素
39.     for( int i = 0; i < 4; i++ )
40.     {
41.         int s03 = d[i*4+0] + d[i*4+3];
42.         int s12 = d[i*4+1] + d[i*4+2];
43.         int d03 = d[i*4+0] - d[i*4+3];
44.         int d12 = d[i*4+1] - d[i*4+2];
45.
46.         tmp[0*4+i] = s03 + s12;
47.         tmp[1*4+i] = 2*d03 + d12;
48.         tmp[2*4+i] = s03 - s12;
49.         tmp[3*4+i] = d03 - 2*d12;
50.     }
51.     //蝶形算法：纵向
52.     for( int i = 0; i < 4; i++ )
53.     {
54.         int s03 = tmp[i*4+0] + tmp[i*4+3];
55.         int s12 = tmp[i*4+1] + tmp[i*4+2];
56.         int d03 = tmp[i*4+0] - tmp[i*4+3];
57.         int d12 = tmp[i*4+1] - tmp[i*4+2];
58.
59.         dct[i*4+0] = s03 + s12;
60.         dct[i*4+1] = 2*d03 + d12;
61.         dct[i*4+2] = s03 - s12;
62.         dct[i*4+3] = d03 - 2*d12;
63.     }
64. }

```

从源代码可以看出，sub4x4\_dct()首先调用pixel\_sub\_wxh()求出两个输入图像块的残差，然后使用蝶形快速算法计算残差图像的DCT系数。

## add4x4\_idct()

add4x4\_idct()可以将残差数据进行DCT反变换，并将变换后得到的残差像素数据叠加到预测数据上。该函数的定义位于common\dct.c，如下所示。

```

1. //4x4DCT反变换 (“add”代表叠加到已有的像素上)
2. static void add4x4_idct( pixel *p_dst, dctcoef dct[16] )
3. {
4.     dctcoef d[16];
5.     dctcoef tmp[16];
6.
7.     for( int i = 0; i < 4; i++ )
8.     {
9.         int s02 =  dct[0*4+i]    +  dct[2*4+i];
10.        int d02 =  dct[0*4+i]    -  dct[2*4+i];
11.        int s13 =  dct[1*4+i]    +  (dct[3*4+i]>>1);
12.        int d13 = (dct[1*4+i]>>1) -  dct[3*4+i];
13.
14.        tmp[i*4+0] = s02 + s13;
15.        tmp[i*4+1] = d02 + d13;
16.        tmp[i*4+2] = d02 - d13;
17.        tmp[i*4+3] = s02 - s13;
18.    }
19.
20.    for( int i = 0; i < 4; i++ )
21.    {
22.        int s02 =  tmp[0*4+i]    +  tmp[2*4+i];
23.        int d02 =  tmp[0*4+i]    -  tmp[2*4+i];
24.        int s13 =  tmp[1*4+i]    +  (tmp[3*4+i]>>1);
25.        int d13 = (tmp[1*4+i]>>1) -  tmp[3*4+i];
26.
27.        d[0*4+i] = ( s02 + s13 + 32 ) >> 6;
28.        d[1*4+i] = ( d02 + d13 + 32 ) >> 6;
29.        d[2*4+i] = ( d02 - d13 + 32 ) >> 6;
30.        d[3*4+i] = ( s02 - s13 + 32 ) >> 6;
31.    }
32.
33.
34.    for( int y = 0; y < 4; y++ )
35.    {
36.        for( int x = 0; x < 4; x++ )
37.            p_dst[x] = x264_clip_pixel( p_dst[x] + d[y*4+x] );
38.        p_dst += FDEC_STRIDE;
39.    }
40. }

```

从源代码可以看出，add4x4\_idct()首先采用快速蝶形算法对DCT系数进行DCT反变换后得到残差像素数据，然后再将残差数据叠加到p\_dst指向的像素上。需要注意这里是“叠加”而不是“赋值”。

## sub8x8\_dct()

sub8x8\_dct()可以将两块8x8的图像相减求残差后，进行4x4DCT变换。该函数的定义位于common\dct.c，如下所示。

```

1. //8x8块：分解成4个4x4DCT变换，调用4次sub4x4_dct()
2. //返回dct[4][16]
3. static void sub8x8_dct( dctcoef dct[4][16], pixel *pix1, pixel *pix2 )
4. {
5.     /*
6.      * 8x8 宏块被划分为4个4x4子块
7.      *
8.      * +---+---+
9.      * | 0 | 1 |
10.     * +---+---+
11.     * | 2 | 3 |
12.     * +---+---+
13.     */
14.     /*
15.     sub4x4_dct( dct[0], &pix1[0], &pix2[0] );
16.     sub4x4_dct( dct[1], &pix1[4], &pix2[4] );
17.     sub4x4_dct( dct[2], &pix1[4*FENC_STRIDE+0], &pix2[4*FDEC_STRIDE+0] );
18.     sub4x4_dct( dct[3], &pix1[4*FENC_STRIDE+4], &pix2[4*FDEC_STRIDE+4] );
19.     */
20. }

```

从源代码可以看出，sub8x8\_dct()将8x8的图像块分成4个4x4的图像块，分别调用了sub4x4\_dct()。

## sub16x16\_dct()

sub16x16\_dct()可以将两块16x16的图像相减求残差后，进行4x4DCT变换。该函数的定义位于common\dct.c，如下所示。



```

1. //16x16块：分解成4个8x8的块做DCT变换，调用4次sub8x8_dct()
2. //返回dct[16][16]
3. static void sub16x16_dct( dctcoef dct[16][16], pixel *pix1, pixel *pix2 )
4. {
5.     /*
6.      * 16x16 宏块被划分为4个8x8子块
7.      *
8.      * +-----+-----+
9.      * |           |           |
10.     * |    0      |    1      |
11.     * |           |           |
12.     * +-----+-----+
13.     * |           |           |
14.     * |    2      |    3      |
15.     * |           |           |
16.     * +-----+-----+
17.     */
18.     /*
19.     sub8x8_dct( &dct[ 0], &pix1[0], &pix2[0] ); //0
20.     sub8x8_dct( &dct[ 4], &pix1[8], &pix2[8] ); //1
21.     sub8x8_dct( &dct[ 8], &pix1[8*FENC_STRIDE+0], &pix2[8*FDEC_STRIDE+0] ); //2
22.     sub8x8_dct( &dct[12], &pix1[8*FENC_STRIDE+8], &pix2[8*FDEC_STRIDE+8] ); //3
23.     */
}

```

从源代码可以看出，sub8x8\_dct()将16x16的图像块分成4个8x8的图像块，分别调用了sub8x8\_dct()。而sub8x8\_dct()实际上又调用了4次sub4x4\_dct()。所以可以得知，不论sub16x16\_dct()，sub8x8\_dct()还是sub4x4\_dct()，本质都是进行4x4DCT。

## dct4x4dc()

dct4x4dc()可以将输入的4x4图像块进行Hadamard变换。该函数的定义位于common\dct.c，如下所示。

```

1. //Hadamard变换
2. static void dct4x4dc( dctcoef d[16] )
3. {
4.     dctcoef tmp[16];
5.
6.     //蝶形算法：横向的4个像素
7.     for( int i = 0; i < 4; i++ )
8.     {
9.
10.         int s01 = d[i*4+0] + d[i*4+1];
11.         int d01 = d[i*4+0] - d[i*4+1];
12.         int s23 = d[i*4+2] + d[i*4+3];
13.         int d23 = d[i*4+2] - d[i*4+3];
14.
15.         tmp[0*4+i] = s01 + s23;
16.         tmp[1*4+i] = s01 - s23;
17.         tmp[2*4+i] = d01 - d23;
18.         tmp[3*4+i] = d01 + d23;
19.     }
20.     //蝶形算法：纵向
21.     for( int i = 0; i < 4; i++ )
22.     {
23.         int s01 = tmp[i*4+0] + tmp[i*4+1];
24.         int d01 = tmp[i*4+0] - tmp[i*4+1];
25.         int s23 = tmp[i*4+2] + tmp[i*4+3];
26.         int d23 = tmp[i*4+2] - tmp[i*4+3];
27.
28.         d[i*4+0] = ( s01 + s23 + 1 ) >> 1;
29.         d[i*4+1] = ( s01 - s23 + 1 ) >> 1;
30.         d[i*4+2] = ( d01 - d23 + 1 ) >> 1;
31.         d[i*4+3] = ( d01 + d23 + 1 ) >> 1;
32.     }
33. }

```

从源代码可以看出，dct4x4dc()实现了Hadamard快速蝶形算法。

## 量化相关的源代码

量化模块的初始化函数是x264\_quant\_init()。该函数对x264\_quant\_function\_t结构体中的函数指针进行了赋值。X264运行的过程中只要调用x264\_quant\_function\_t的函数指针就可以完成相应的功能。

## x264\_quant\_init()

x264\_quant\_init()初始化量化和反量化相关的汇编函数。该函数的定义位于common\quant.c，如下所示。

```

1. //量化
2. void x264_quant_init( x264_t *h, int cpu, x264_quant_function_t *pf )
3. {
4.     //这个好像是针对8x8DCT的
5.     pf->quant_8x8 = quant_8x8;
6.
7.     //量化4x4=16个
8.     pf->quant_4x4 = quant_4x4;
9.     //注意：处理4个4x4的块
10.    pf->quant_4x4x4 = quant_4x4x4;
11.    //Intra16x16中，16个DC系数Hadamard变换后对的它们量化
12.    pf->quant_4x4_dc = quant_4x4_dc;
13.    pf->quant_2x2_dc = quant_2x2_dc;
14.    //反量化4x4=16个
15.    pf->dequant_4x4 = dequant_4x4;
16.    pf->dequant_4x4_dc = dequant_4x4_dc;
17.    pf->dequant_8x8 = dequant_8x8;
18.
19.    pf->idct_dequant_2x4_dc = idct_dequant_2x4_dc;
20.    pf->idct_dequant_2x4_donly = idct_dequant_2x4_donly;
21.
22.    pf->optimize_chroma_2x2_dc = optimize_chroma_2x2_dc;
23.    pf->optimize_chroma_2x4_dc = optimize_chroma_2x4_dc;
24.
25.    pf->denoise_dct = x264_denoise_dct;
26.    pf->decimate_score15 = x264_decimate_score15;
27.    pf->decimate_score16 = x264_decimate_score16;
28.    pf->decimate_score64 = x264_decimate_score64;
29.
30.    pf->coeff_last4 = x264_coeff_last4;
31.    pf->coeff_last8 = x264_coeff_last8;
32.    pf->coeff_last[ DCT_LUMA_AC ] = x264_coeff_last15;
33.    pf->coeff_last[ DCT_LUMA_4x4 ] = x264_coeff_last16;
34.    pf->coeff_last[ DCT_LUMA_8x8 ] = x264_coeff_last64;
35.    pf->coeff_level_run4 = x264_coeff_level_run4;
36.    pf->coeff_level_run8 = x264_coeff_level_run8;
37.    pf->coeff_level_run[ DCT_LUMA_AC ] = x264_coeff_level_run15;
38.    pf->coeff_level_run[ DCT_LUMA_4x4 ] = x264_coeff_level_run16;
39.
40.    #if HIGH_BIT_DEPTH
41.    #if HAVE_MMX
42.        INIT_TRELLIS( sse2 );
43.        if( cpu & X264_CPU_MMX2 )
44.        {
45.            #if ARCH_X86
46.                pf->denoise_dct = x264_denoise_dct_mmx;
47.                pf->decimate_score15 = x264_decimate_score15_mmx2;
48.                pf->decimate_score16 = x264_decimate_score16_mmx2;
49.                pf->decimate_score64 = x264_decimate_score64_mmx2;
50.                pf->coeff_last8 = x264_coeff_last8_mmx2;
51.                pf->coeff_last[ DCT_LUMA_AC ] = x264_coeff_last15_mmx2;
52.                pf->coeff_last[ DCT_LUMA_4x4 ] = x264_coeff_last16_mmx2;
53.                pf->coeff_last[ DCT_LUMA_8x8 ] = x264_coeff_last64_mmx2;
54.                pf->coeff_level_run8 = x264_coeff_level_run8_mmx2;
55.                pf->coeff_level_run[ DCT_LUMA_AC ] = x264_coeff_level_run15_mmx2;
56.                pf->coeff_level_run[ DCT_LUMA_4x4 ] = x264_coeff_level_run16_mmx2;
57.            #endif
58.            pf->coeff_last4 = x264_coeff_last4_mmx2;
59.            pf->coeff_level_run4 = x264_coeff_level_run4_mmx2;
60.            if( cpu & X264_CPU_LZCNT )
61.                pf->coeff_level_run4 = x264_coeff_level_run4_mmx2_lzcnt;
62.        }
63.        //此处省略大量的X86、ARM等平台的汇编函数初始化代码
64.    }

```

从源代码可以看出，x264\_quant\_init()初始化了一系列的量化相关的函数。它的输入参数x264\_quant\_function\_t是一个结构体，其中包含了和量化相关各种函数指针。x264\_quant\_function\_t的定义如下所示。

```

1. typedef struct
2. {
3.     int (*quant_8x8) ( dctcoef dct[64], udctcoef mf[64], udctcoef bias[64] );
4.     int (*quant_4x4) ( dctcoef dct[16], udctcoef mf[16], udctcoef bias[16] );
5.     int (*quant_4x4x4)( dctcoef dct[4][16], udctcoef mf[16], udctcoef bias[16] );
6.     int (*quant_4x4_dc)( dctcoef dct[16], int mf, int bias );
7.     int (*quant_2x2_dc)( dctcoef dct[4], int mf, int bias );
8.
9.     void (*dequant_8x8)( dctcoef dct[64], int dequant_mf[6][64], int i_qp );
10.    void (*dequant_4x4)( dctcoef dct[16], int dequant_mf[6][16], int i_qp );
11.    void (*dequant_4x4_dc)( dctcoef dct[16], int dequant_mf[6][16], int i_qp );
12.
13.    void (*idct_dequant_2x4_dc)( dctcoef dct[8], dctcoef dct4x4[8][16], int dequant_mf[6][16], int i_qp );
14.    void (*idct_dequant_2x4_donly)( dctcoef dct[8], int dequant_mf[6][16], int i_qp );
15.
16.    int (*optimize_chroma_2x2_dc)( dctcoef dct[4], int dequant_mf );
17.    int (*optimize_chroma_2x4_dc)( dctcoef dct[8], int dequant_mf );
18.
19.    void (*denoise_dct)( dctcoef *dct, uint32_t *sum, udctcoef *offset, int size );
20.
21.    int (*decimate_score15)( dctcoef *dct );
22.    int (*decimate_score16)( dctcoef *dct );
23.    int (*decimate_score64)( dctcoef *dct );
24.    int (*coeff_last14)( dctcoef *dct );
25.    int (*coeff_last4)( dctcoef *dct );
26.    int (*coeff_last8)( dctcoef *dct );
27.    int (*coeff_level_run13)( dctcoef *dct, x264_run_level_t *runlevel );
28.    int (*coeff_level_run4)( dctcoef *dct, x264_run_level_t *runlevel );
29.    int (*coeff_level_run8)( dctcoef *dct, x264_run_level_t *runlevel );
30.
31.    #define TRELLIS_PARAMS const int *unquant_mf, const uint8_t *zigzag, int lambda2,\
32.        int last_nnz, dctcoef *coefs, dctcoef *quant_coefs, dctcoef *dct,\
33.        uint8_t *cabac_state_sig, uint8_t *cabac_state_last,\
34.        uint64_t level_state0, uint16_t level_state1
35.    int (*trellis_cabac_4x4)( TRELLIS_PARAMS, int b_ac );
36.    int (*trellis_cabac_8x8)( TRELLIS_PARAMS, int b_interlaced );
37.    int (*trellis_cabac_4x4_psy)( TRELLIS_PARAMS, int b_ac, dctcoef *fenc_dct, int psy_trellis );
38.    int (*trellis_cabac_8x8_psy)( TRELLIS_PARAMS, int b_interlaced, dctcoef *fenc_dct, int psy_trellis );
39.    int (*trellis_cabac_dc)( TRELLIS_PARAMS, int num_coefs );
40.    int (*trellis_cabac_chroma_422_dc)( TRELLIS_PARAMS );
41. } x264_quant_function_t;

```

x264\_quant\_init()的工作就是对x264\_quant\_function\_t中的函数指针进行赋值。下文分析其中2个函数：4x4矩阵量化函数quant\_4x4()，4个4x4矩阵量化函数quant\_4x4x4()。

## quant\_4x4()

quant\_4x4()用于对4x4的DCT残差矩阵进行量化。该函数的定义位于common/quant.c，如下所示。

```

1. //4x4量化
2. //输入输出都是dct[16]
3. static int quant_4x4( dctcoef dct[16], udctcoef mf[16], udctcoef bias[16] )
4. {
5.     int nz = 0;
6.     //循环16个元素
7.     for( int i = 0; i < 16; i++ )
8.         QUANT_ONE( dct[i], mf[i], bias[i] );
9.     return !!nz;
10. }

```

可以看出quant\_4x4()循环16次调用了QUANT\_ONE()完成了量化工作。并且将DCT系数值，MF值，bias偏移值直接传递给了该宏。

## QUANT\_ONE()

QUANT\_ONE()完成了一个DCT系数的量化工作，它的定义如下。

```



1. //量化1个元素
2. #define QUANT_ONE( coef, mf, f ) \
3. { \
4.     if( (coef) > 0 ) \
5.         (coef) = (f + (coef)) * (mf) >> 16; \
6.     else \
7.         (coef) = - ((f - (coef)) * (mf) >> 16); \
8.     nz |= (coef); \
9. }

```

从QUANT\_ONE()的定义可以看出，它实现了上文提到的H.264标准中的量化公式。

## quant\_4x4x4()

quant\_4x4x4()用于对4个4x4的DCT残差矩阵进行量化。该函数的定义位于common\quant.c，如下所示。

```
[cpp]    
1. //处理4个4x4量化  
2. //输入输出都是dct[4][16]  
3. static int quant_4x4x4( dctcoef dct[4][16], udctcoef mf[16], udctcoef bias[16] )  
4. {  
5.     int nza = 0;  
6.     //处理4个  
7.     for( int j = 0; j < 4; j++ )  
8.     {  
9.         int nz = 0;  
10.        //量化  
11.        for( int i = 0; i < 16; i++ )  
12.            QUANT_ONE( dct[j][i], mf[i], bias[i] );  
13.        nza |= (!!nz)<<j;  
14.    }  
15.    return nza;  
16. }
```

从quant\_4x4x4()的定义可以看出，该函数相当于调用了4次quant\_4x4()函数。

至此有关x264中的宏块编码模块的源代码就分析完毕了。

**雷霄骅**

leixiaohua1020@126.com

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

版权声明：本文为博主原创文章，未经博主允许不得转载。 <https://blog.csdn.net/leixiaohua1020/article/details/45938927>

文章标签： [x264](#) [DCT](#) [量化](#) [残差](#) [编码](#)

个人分类： [x264](#)

所属专栏： [开源多媒体项目源代码分析](#)

此PDF由[spygg](#)生成,请尊重原作者版权!!!

我的邮箱:liushidc@163.com