x264 源代码分析

来源：http://blog.lmtw.com/b/guanyhu/archives/2006/38868.html

相关说明:

1. 使用版本: x264-cvs-2004-05-11
2. 这次的分析基本上已经将代码中最难理解的部分做了阐释,对代码的主线也做了剖析,如果这个主线理解了,就容易设置几个区间,进行分工阅读,将各个区间击破了.
3. 需要学习的知识:

a) 编码器的工作流程.

b) H.264 的码流结构,像 x264_sps_t,x264_pps_t 等参数的定义基本上都完全符合标准文档中参数集的定义,抓住主要参数,次要参数也应该有所了解.

c) 数学知识,对 dct 变换等与数学相关的知识的编程实现要有较好理解.

d) C 语言的知识.涉及到 c 语言的较多不经常用的特性,如函数指针数组,移位运算,结构体的嵌套定义等.

e) 耐心,对 h.264 的复杂性要有清醒的认识.

3.参考资料:

a) 新一代视频压缩编码标准-h.264/avc 毕厚杰主编,人民邮电出版社.

b) 网上的流媒体论坛,百度,google 等搜索引擎.

4. 阅读代码的方法:

a) 较好的方法是利用 vc 的调试器,如果对某个函数感兴趣,可以将断点设置在它的前面.然后采用 step into,step over 等方法进去该函数一步步分析.当然本身要对程序执行流程要有较清楚认识,不然不知道何时 step into,何时 step over.

b) 建议应该先对照标准弄清各个结构体成员的意义.

源代码主要过程分析:

1. 进入 x264.c 中的 main 函数.

刚开始是读取默认参数,如果你设置了参数的话会修改 param 的.

```
i_ret = Encode( &param, fin, fout );
```

这条语句使过程进入 x264.c 中的 Encode 函数.

2. X.264 的 encode 函数.

A. i_frame_total = 0;

```
if( !fseek( fyuv, 0, SEEK_END ) )
    {
      int64_t i_size = ftell( fyuv );
      fseek( fyuv, 0, SEEK_SET );
      i_frame_total = i_size / ( param->i_width * param->i_height * 3 / 2 )
}
```

上面这段计算出输入文件的总帧数.

B. h = x264_encoder_open(param)这个函数是对不正确的参数进行修改,并对各结构体参数和 cabac 编码,预测等需要的参数进行初始化.

C. pic = x264_picture_new(h);

该函数定义在\CORE\common.c 中.首先分给能容纳 sizeof(x264_picture_t)字节数的空间,然后进行初始化.

这里看一下 x264_picture_t 和 x264_frame_t 的区别.前者是说明一个视频序列中每帧的特点.后者存放每帧实际的象素值.注意区分.

D. for(i_frame = 0, i_file = 0; i_ctrl_c == 0 ; i_frame++)

```
{
    int         i_nal;
    x264_nal_t   *nal;
int         i;
/* read a frame */
    if( fread( pic->plane[0], 1, param->i_width * param->i_height, fyuv ) <= 0 ||
        fread( pic->plane[1], 1, param->i_width * param->i_height / 4, fyuv ) <= 0 ||
        fread( pic->plane[2], 1, param->i_width * param->i_height / 4, fyuv ) <= 0 )
    {
        break;
    }
  //文件位置指示器自己变化了.

    if( x264_encoder_encode( h, &nal, &i_nal, pic ) < 0 )
    {
        fprintf( stderr, "x264_encoder_encode failed\n" );
    }
    ……
}
```

凡是出现 for 循环的地方都应该认真对待,这是我的一个体会,也是进入分层结构认真分析的好方法.

fread()函数一次读入一帧,分亮度和色度分别读取.这里要看到 c 语言中的 File 文件有一个文件位置指示器,调用 fread()函数会使文件指示器自动移位,这就是一帧一帧读取的实现过程.

E. 然后进入 x264_encoder_encode(h, &nal, &i_nal, pic)函数,该函数定义在/Enc/encoder.c 中.

开始进入比较复杂的地方了.

这个函数前面有一段注释(如下):

```
*************************************************************************
 * x264_encoder_encode:
 *   XXX: i_poc    : is the poc of the current given picture
 *        i_frame : is the number of the frame being coded
 *   ex:   type frame poc
 *        I      0    2*0//poc 是实际的帧的位置.
 *        P      1    2*3//frame 是编码的顺序.
 *        B      2    2*1
 *        B      3    2*2
```

```
 *      P      4   2*6
 *      B      5   2*4
 *      B      6   2*5
 ************************************************************************
********/
```

要搞清 poc 和 frame 的区别.
假设一个视频序列如下:
I B B P B B P
我们编码是按 I P B B P B B 的顺序,这就是 frame 的编号.
而我们视频序列的播放序号是 POC 的序号,这里是乘以了 2.
函数中先定义了如下三个参数:
int i_nal_type;
nal 存放的数据类型, 可以是 sps,pps 等多种.
int i_nal_ref_idc;
nal 的优先级,nal 重要性的标志位.
前面两个参数虽然简单,但如果不参照标准,也不容易理解,所以标准中的句法表是很重要的,可以说是最关键的.
int i_slice_type;
slice 的类型,在 x264 中我的感觉好像一帧只有一个 slice.如果确定了帧的类型,slice 的类型也就确定了.

我们来看看编码器是如何区分读入的一帧是 I 帧,P 帧,或者 B 帧,这个过程需要好好理解.
还以 I B B P B B P 为例.

```
if( h->i_frame % (h->param.i_iframe * h->param.i_idrframe) == 0 ){
```

确定这是立即刷新片.

```
}
```

这里很好理解.
但到了 if(h->param.i_bframe > 0)//可以 B 帧编码时.
就有问题了.
注意我们编完 I 帧后碰到了一个 B 帧,这时我们先不对它进编码.而是采用 frame = x264_encoder_frame_put_from_picture(h, h->frame_next, pic)函数将这个 B 帧放进 h->frame_next 中.
好,这里出现了 h->frame_next,在 h 中同时定义了下面几个帧数组用以实现帧的管理.

```
x264_frame_t    *bframe_current[X264_BFRAME_MAX]; /* store the sequence of b frame being encoded */
    x264_frame_t    *frame_next[X264_BFRAME_MAX+1];    /* store the next sequence of frames to be encoded *///搞清意义,下一个帧,而不一定是 B 帧.
    x264_frame_t    *frame_unused[X264_BFRAME_MAX+1]; /* store unused frames */
```

注意区分这 3 个数组.
同时还有下面 4 个函数(定义在\ENCODER\encoder.c 中).

x264_encoder_frame_put_from_picture();
x264_encoder_frame_put();
x264_encoder_frame_get();
x264_frame_copy_picture();
这 3 个数组和 4 个函数可以说完成了整个帧的类型的判定问题.这个里面 if ,else 语句较多,容易使人迷惑.但我们只要把握下面一个观点就可以看清实质:在不对 P 帧进行编码之前,我们不对 B 帧进行编码,只是把 B 帧放进缓冲区(就是前面提到的数组).
比如视频序列:I B B P B B P
先确立第一个帧的类型,然后进行编码.然后是 2 个 B 帧,我们把它放进缓冲区数组.然后是 P 帧,我们可以判定它的类型并进行编码.同时,我们将缓冲区的 B 帧放进 h->bframe_current[i],不过这时 P 帧前的两个 B 帧并没有编码.当读到 P 帧后面的第一个 B 帧时,我们实际上才将 h->bframe_current 数组中的第一个 B 帧编码,也就是将在 I 帧后面的第一个 B 帧(说成 P 帧前面的第一个 B 帧容易误解 J)编码.
依此类推,把握好上面 4 个函数的调用流程和指针操作的用法,就可以将帧的类型判定这个问题搞明白了.
F. 然后是速率控制(先不说这个,因为它对编码的流程影响不大),看看建立参考帧列表的操作,也就是
x264_reference_build_list(h, h->fdec->i_poc); (定义在\ENCODER\encoder.c 中).
光看这个函数是不行的,它是和后面的这个函数(如下)一起配合工作的.

```
if( i_nal_ref_idc != NAL_PRIORITY_DISPOSABLE )//B 帧时.
    {
        x264_reference_update( h );
}
```

If 条件是判断当前帧是否是 B 帧,如果是的话就不更新参考列表,因为 B 帧本来就不能作为参考帧嘛!如果是 I 帧或 P 帧的话,我们就更新参考帧列表.
我们看到了一个 for 循环,两个 do—while 循环.这是实现的关键,具体看代码,不好用语言说明白.

G. 进入另一个复杂的领域:写 slice 的操作,刚开使挺简单,如我下面的注释.

```
/* ---------------------- Write the bitstream ------------------------- */
    /* Init bitstream context */
    h->out.i_nal = 0;//out 的声明在 bs.h 中.
    bs_init( &h->out.bs, h->out.p_bitstream, h->out.i_bitstream );//空出 8 位.

    /* Write SPS and PPS */
    if( i_nal_type == NAL_SLICE_IDR )//不是每次都要写 SPS and PPS,只有碰见立即刷新片时才写.
    {
        /* generate sequence parameters */
        x264_nal_start( h, NAL_SPS, NAL_PRIORITY_HIGHEST );
        x264_sps_write( &h->out.bs, h->sps );
        x264_nal_end( h );
```

```
        /* generate picture parameters */
        x264_nal_start( h, NAL_PPS, NAL_PRIORITY_HIGHEST );
        x264_pps_write( &h->out.bs, h->pps );
        x264_nal_end( h );
}
```
不过看下面那个函数(就进入了复杂的领域).

H. x264_slice_write()(定义在\ENCODER\encoder.c 中),这里面是编码的最主要部分,下面仔细分析.
前面不说,看下面这个循环,它是采用 for 循环对一帧图像的所有块依次进行编码.
for(mb_xy = 0, i_skip = 0; mb_xy < h->sps->i_mb_width * h->sps->i_mb_height; mb_xy++)//h->sps->i_mb_width 指的是从宽度上说有多少个宏快.对于宽度也就是 288 / 16 = 18
```
    {
        const int i_mb_y = mb_xy / h->sps->i_mb_width;
        const int i_mb_x = mb_xy % h->sps->i_mb_width;//这两个变量是定义宏块的位置.而不是指宏块中元素的位置.

        /* load cache */
        x264_macroblock_cache_load( h, i_mb_x, i_mb_y );//是把当前宏块的 up 宏块和 left 宏块的 intra4x4_pred_mode，non_zero_count 加载进来，放到一个数组里面，这个数组用来直接得到当前宏块的左侧和上面宏块的相关值.要想得到当前块的预测值，要先知道上面，左面的预测值，它的目的是替代 getneighbour 函数.
/* analyse parameters
         * Slice I: choose I_4x4 or I_16x16 mode
         * Slice P: choose between using P mode or intra (4x4 or 16x16)
         * */
        TIMER_START( i_mtime_analyse );
        x264_macroblock_analyse( h );//定义在 analyse.h 中.
        TIMER_STOP( i_mtime_analyse );

        /* encode this macrobock -> be carefull it can change the mb type to P_SKIP if needed */
        TIMER_START( i_mtime_encode );
        x264_macroblock_encode( h );//定义在 Enc/encoder.c 中.
        TIMER_STOP( i_mtime_encode );
```
截止到这就已经完成编码的主要过程了,后面就是熵编码的过程了(我也没看到那,但认为前面才是编码的主要过程).下面对这个过程进行分析.
A. x264_macroblock_cache_load(h, i_mb_x, i_mb_y);它是将要编码的宏块的周围的宏块的值读进来, 要想得到当前块的预测值，要先知道上面，左面的预测值，它的作用相当于 jm93 中的 getneighbour 函数.
B. 进入 x264_macroblock_analyse(h)函数(定义在\Enc\analyse.c 中,这里涉及到了函数指针数组,需要好好复习,个人认为这也是 x264 代码最为复杂的一个地方了).既然已经将该宏块周围的宏块的值读了出来,我们就可以对该宏块进行分

析了(其实主要就是通过计算 sad 值分析是否要将 16*16 的宏块进行分割和采用哪种分割方式合适).
看似很复杂,但我们只要把握一个东西就有利于理解了:
举个生活中的例子来说:
如果你有 2 元钱,你可以去买 2 袋 1 元钱的瓜子,也可以买一袋 2 元钱的瓜子,如果 2 袋 1 元钱的瓜子数量加起来比 1 袋 2 元钱的瓜子数量多,你肯定会买 2 袋 1 元的.反之你会去买那 2 元 1 袋的.
具体来说,对于一个 16*16 的块,
如果它是 I 帧的块,我们可以将它分割成 16 个 4*4 的块,如果这 16 个块的 sad 加起来小于按 16*16 的方式计算出来的 sad 值,我们就将这个 16*16 的块分成 16 个 4*4 的块进行编码(在计算每个 4*4 的块的最小 sad 值时已经知道它采用何种编码方式最佳了),否则采用 16*16 的方式编码(同样我们也已知道对它采用哪种编码方式最为合适了.
如果它是 P 帧或 B 帧的块,同样是循环套循环,但更为复杂了,可以看我在 analyse.c 中的注释.
这里还要注意的是提到了
x264_predict_t　　predict_16x16[4+3];
typedef void (*x264_predict_t)(uint8_t *src, int i_stride);
这是函数指针数组,有很多对它的调用.
C.　退出 x264_macroblock_analyse(h)函数,进入 x264_macroblock_encode( )函数(定义在\ENCODER\macroblock.c 中).
我拿宏块类型为 I_16*16 为例.

```
if( h->mb.i_type == I_16x16 )
    {
        const int i_mode = h->mb.i_intra16x16_pred_mode;
        /* do the right prediction */
        h->predict_16x16[i_mode]( h->mb.pic.p_fdec[0], h->mb.pic.i_fdec[0] );//这两个参数的关系.
                                                    //涉及到 x264_predict_t(函数指针数组),声明在 core/predict.h 中,core/predict.c 里有不同定义.
        /* encode the 16x16 macroblock */
        x264_mb_encode_i16x16( h, i_qscale );//
/* fix the pred mode value */
        …　　}
```

我们看到 h->predict_16x16[i_mode](h->mb.pic.p_fdec[0], h->mb.pic.i_fdec[0]);只调用了一次,这是因为在 x264_macroblock_analyse(　)中我们已经确定了采用 4 种方式中的哪种最合适.而在 x264_macroblock_analyse(　)中判定一个块是否为 I_16*16,我们调用了四次.这是因为当时我们需要拿最小的 sad 值进行比较.
继续,是 x264_mb_encode_i16x16(h, i_qscale)函数(定义在\ENCODER\macroblock.c 中).在这个函数中我们就可以看到量化,zig-扫描等函数了,这些都是直来直去的,需要的只是我们的细心和对数学知识的掌握了

c)　　到这里还没完,我们接着看

```
void x264_macroblock_encode( x264_t *h ){
```

…….前面省略.
执行到下面这条语句,看看下面是干啥的.

```
    /* encode chroma */
    i_qscale = i_chroma_qp_table[x264_clip3( i_qscale +
h->pps->i_chroma_qp_index_offset, 0, 51 )];
    if( IS_INTRA( h->mb.i_type ) )
    {
        const int i_mode = h->mb.i_chroma_pred_mode;
        /* do the right prediction */
        h->predict_8x8[i_mode]( h->mb.pic.p_fdec[1], h->mb.pic.i_fdec[1] );
        h->predict_8x8[i_mode]( h->mb.pic.p_fdec[2], h->mb.pic.i_fdec[2] );

        /* fix the pred mode value */
        h->mb.i_chroma_pred_mode = x264_mb_pred_mode8x8_fix[i_mode];
    }

    /* encode the 8x8 blocks */
x264_mb_encode_8x8( h, !IS_INTRA( h->mb.i_type ), i_qscale );//对色度块进行编
码了.
```

到这我们可以看到原来我们在这前面是对宏块中的亮度系数进行了编码,我们到上面那个函数才开始对色度系数进行编码.进入 x264_mb_encode_8x8()函数看到 for 循环里面有个 2 可以证明是对 2 个色度系数进行编码,想法没错.
那下面这些又是干啥的呢?它们是计算 cbp 系数看需要对残差(包括 ac,dc)中的哪个系数进行传输的.

```
    /* Calculate the Luma/Chroma patern and non_zero_count */
    if( h->mb.i_type == I_16x16 )
    {
        h->mb.i_cbp_luma = 0x00;
        for( i = 0; i < 16; i++ )
        {
            const int nz = array_non_zero_count( h->dct.block[i].residual_ac, 15 );
            h->mb.cache.non_zero_count[x264_scan8[i]] = nz;
            if( nz > 0 )
            {
                h->mb.i_cbp_luma = 0x0f;
            }
        }
    }
    else
    {
        h->mb.i_cbp_luma = 0x00;
        for( i = 0; i < 16; i++ )
```

```
        {
            const int nz = array_non_zero_count( h->dct.block[i].luma4x4, 16 );//统计非
0 个数.
            h->mb.cache.non_zero_count[x264_scan8[i]] = nz;
            if( nz > 0 )
            {
                h->mb.i_cbp_luma |= 1 << (i/4);// %16 的意义.
            }
        }
    }

    /* Calculate the chroma patern *///色度的 cbp 有 3 种方式.
    h->mb.i_cbp_chroma = 0x00;
    for( i = 0; i < 8; i++ )
    {
        const int nz = array_non_zero_count( h->dct.block[16+i].residual_ac, 15 );
        h->mb.cache.non_zero_count[x264_scan8[16+i]] = nz;
        if( nz > 0 )
        {
            h->mb.i_cbp_chroma = 0x02;     /* dc+ac (we can't do only ac) */
        }
    }
    if( h->mb.i_cbp_chroma == 0x00 &&
        ( array_non_zero_count( h->dct.chroma_dc[0], 4 ) > 0 ||
array_non_zero_count( h->dct.chroma_dc[1], 4 ) ) > 0 )
    {
        h->mb.i_cbp_chroma = 0x01;     /* dc only */
    }

    if( h->param.b_cabac )
    {
        if( h->mb.i_type == I_16x16 && array_non_zero_count( h->dct.luma16x16_dc,
16 ) > 0 )
            i_cbp_dc = 0x01;
        else
            i_cbp_dc = 0x00;

        if( array_non_zero_count( h->dct.chroma_dc[0], 4 ) > 0 )
            i_cbp_dc |= 0x02;
        if( array_non_zero_count( h->dct.chroma_dc[1], 4 ) > 0 )
            i_cbp_dc |= 0x04;
    }

    /* store cbp */
```

h->mb.cbp[h->mb.i_mb_xy] = (i_cbp_dc << 8) | (h->mb.i_cbp_chroma << 4) | h->mb.i_cbp_luma;

到这,基本上 x264_macroblock_encode(h)(定义在 Enc/encoder.c)基本上就分析完了.剩下的就是熵编码的部分了.以后的部分更需要的应该是耐心和数学知识吧,相对前面来说应该简单些.

l 总结:

1. 我对代码的理解应该还算比较深入,把代码的主线已经分析了出来,对代码中几个最难理解的地方(最难理解的地方就是帧的类型的判定,参考帧是如何管理的,一个 16*16 的块是采用到底需不需要分割,分割的话分成什么大小的,子块又采用何种预测方式,这些实际上就是整个编码的主线.)基本上已经明白,但有些过分复杂的函数的实现(或者涉及数学知识较多的地方)还有待深入研究,但我相信沿着这条主线应该能够继续深入下去,自己需要的是更多的时间和耐心. 自己需要的是更多的时间和耐心,争取以后能写出更详细更准确的流程分析,并尽量思考能改进的地方.

2.层次性,就像网络的 7 层结构一样,每一帧图像也可以分成很多层,只有对每层的语法结构(具体来说就是各个结构体中变量的意思)有了很好的理解,才有可能真正认清代码,这需要对标准认真研习.比如说量化参数,就在 3 个地方有定义,不读标准根本不会明白意思.

3. 很多过分复杂的东西不容易在本文中表达出来(比如说预测部分),只有通过自己的钻研才能真正悟到,直觉也很重要,还有就是信心了.看这种程序的收获就好像是真地肉眼看到了原子那样.

4.由于代码过分复杂,对某些函数的实现过程还没能彻底理解,比如说 x264_macroblock_cache_load()函数的具体实现过程,我只是知道它的功能,实现过程还有待认真理解.dct 变换是如何实现的,是如何计算残差的等等,这些都需要很多功夫,当然这里也需要大家的共同学习和交流.实现分工阅读不同代码部分并进行交流,才有可能对代码做到彻底的理解.