原 FFmpeg的H.264解码器源代码简单分析：解码器主干部分

2015年04月14日 16:04:50　阅读数：15858

==========

H.264源代码分析文章列表：

【编码 - x264】

x264源代码简单分析：概述

x264源代码简单分析：x264命令行工具（x264.exe）

x264源代码简单分析：编码器主干部分-1

x264源代码简单分析：编码器主干部分-2

x264源代码简单分析：x264_slice_write()

x264源代码简单分析：滤波（Filter）部分

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

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

x264源代码简单分析：宏块编码（Encode）部分

x264源代码简单分析：熵编码（Entropy Encoding）部分

FFmpeg与libx264接口源代码简单分析

【解码 - libavcodec H.264 解码器】

FFmpeg的H.264解码器源代码简单分析：概述

FFmpeg的H.264解码器源代码简单分析：解析器（Parser）部分

FFmpeg的H.264解码器源代码简单分析：解码器主干部分

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

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

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

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

==========

本文分析FFmpeg的H.264解码器的主干部分。“主干部分”是相对于“熵解码”、“宏块解码”、“环路滤波”这些细节部分而言的。它包含了H.264解码器直到decode_slice()前面的函数调用关系（decode_slice()后面就是H.264解码器的细节部分，主要包含了“熵解码”、“宏块解码”、“环路滤波”3个部分）。

函数调用关系图

解码器主干部分的源代码在整个H.264解码器中的位置如下图所示。

单击查看更清晰的图片

解码器主干部分的源代码的调用关系如下图所示。

单击查看更清晰的图片

从图中可以看出，H.264解码器（Decoder）在初始化的时候调用了ff_h264_decode_init()，ff_h264_decode_init()又调用了下面几个函数进行解码器汇编函数的初始化工作（仅举了几个例子）：

ff_h264dsp_init()：初始化DSP相关的汇编函数。包含了IDCT、环路滤波函数等。

ff_h264qpel_init()：初始化四分之一像素运动补偿相关的汇编函数。

ff_h264_pred_init()：初始化帧内预测相关的汇编函数。

H.264解码器在关闭的时候调用了h264_decode_end()，h264_decode_end()又调用了ff_h264_remove_all_refs()，ff_h264_free_context()等几个函数进行清理工作。

H.264解码器在解码图像帧的时候调用了h264_decode_frame()，h264_decode_frame()调用了decode_nal_units()，decode_nal_units()调用了两类函数——解析函数和解码函数，如下所示。

（1）解析函数（获取信息）：

ff_h264_decode_nal()：解析NALU Header。

ff_h264_decode_seq_parameter_set()：解析SPS。

ff_h264_decode_picture_parameter_set()：解析PPS。

ff_h264_decode_sei()：解析SEI。

ff_h264_decode_slice_header()：解析Slice Header。

（2）解码函数（解码获得图像）：

ff_h264_execute_decode_slices()：解码Slice。

其中ff_h264_execute_decode_slices()调用了decode_slice()，而decode_slice()中调用了解码器中细节处理的函数（暂不详细分析）：

ff_h264_decode_mb_cabac()：CABAC熵解码函数。

ff_h264_decode_mb_cavlc()：CAVLC熵解码函数。

ff_h264_hl_decode_mb()：宏块解码函数。

loop_filter()：环路滤波函数。

本文针对H.264解码器decode_slice()前面的函数调用关系进行分析。

ff_h264_decoder

ff_h264_decoder是FFmpeg的H.264解码器对应的AVCodec结构体。它的定义位于libavcodec\h264.c，如下所示。

```cpp
AVCodec ff_h264_decoder = {
    .name                  = "h264",
    .long_name             = NULL_IF_CONFIG_SMALL("H.264 / AVC / MPEG-4 AVC / MPEG-4 part 10"),
    .type                  = AVMEDIA_TYPE_VIDEO,
    .id                    = AV_CODEC_ID_H264,
    .priv_data_size        = sizeof(H264Context),
    .init                  = ff_h264_decode_init,
    .close                 = h264_decode_end,
    .decode                = h264_decode_frame,
    .capabilities          = /*CODEC_CAP_DRAW_HORIZ_BAND |*/ CODEC_CAP_DR1 |
                             CODEC_CAP_DELAY | CODEC_CAP_SLICE_THREADS |
                             CODEC_CAP_FRAME_THREADS,
    .flush                 = flush_dpb,
    .init_thread_copy      = ONLY_IF_THREADS_ENABLED(decode_init_thread_copy),
    .update_thread_context = ONLY_IF_THREADS_ENABLED(ff_h264_update_thread_context),
    .profiles              = NULL_IF_CONFIG_SMALL(profiles),
    .priv_class            = &h264_class,
};
```

从ff_h264_decoder的定义可以看出：解码器初始化的函数指针init()指向ff_h264_decode_init()函数，解码的函数指针decode()指向h264_decode_frame()函数，解码器关闭的函数指针close()指向h264_decode_end()函数。

ff_h264_decode_init()

ff_h264_decode_init()用于FFmpeg H.264解码器的初始化。该函数的定义位于libavcodec\h264.c，如下所示。

```cpp
//H.264解码器初始化函数
av_cold int ff_h264_decode_init(AVCodecContext *avctx)
{
    H264Context *h = avctx->priv_data;
    int i;
    int ret;

    h->avctx = avctx;
    //8颜色位深8bit
    h->bit_depth_luma    = 8;
    //1代表是YUV420P
    h->chroma_format_idc = 1;

    h->avctx->bits_per_raw_sample = 8;
    h->cur_chroma_format_idc = 1;
    //初始化DSP相关的汇编函数。包含了IDCT、环路滤波函数等
    ff_h264dsp_init(&h->h264dsp, 8, 1);
    av_assert0(h->sps.bit_depth_chroma == 0);
    ff_h264chroma_init(&h->h264chroma, h->sps.bit_depth_chroma);
    //初始化四分之一像素运动补偿相关的汇编函数
    ff_h264qpel_init(&h->h264qpel, 8);
    //初始化帧内预测相关的汇编函数
    ff_h264_pred_init(&h->hpc, h->avctx->codec_id, 8, 1);

    h->dequant_coeff_pps = -1;
    h->current_sps_id = -1;
```

```
    /* needed so that IDCT permutation is known early */
    if (CONFIG_ERROR_RESILIENCE)
        ff_me_cmp_init(&h->mecc, h->avctx);
    ff_videodsp_init(&h->vdsp, 8);

    memset(h->pps.scaling_matrix4, 16, 6 * 16 * sizeof(uint8_t));
    memset(h->pps.scaling_matrix8, 16, 2 * 64 * sizeof(uint8_t));

    h->picture_structure   = PICT_FRAME;
    h->slice_context_count = 1;
    h->workaround_bugs     = avctx->workaround_bugs;
    h->flags               = avctx->flags;

    /* set defaults */
    // s->decode_mb = ff_h263_decode_mb;
    if (!avctx->has_b_frames)
        h->low_delay = 1;

    avctx->chroma_sample_location = AVCHROMA_LOC_LEFT;
    //初始化熵解码器
    //CAVLC
    ff_h264_decode_init_vlc();
    //CABAC
    ff_init_cabac_states();
    //8-bit H264取0，大于 8-bit H264取1
    h->pixel_shift        = 0;
    h->sps.bit_depth_luma = avctx->bits_per_raw_sample = 8;

    h->thread_context[0] = h;
    h->outputed_poc      = h->next_outputed_poc = INT_MIN;
    for (i = 0; i < MAX_DELAYED_PIC_COUNT; i++)
        h->last_pocs[i] = INT_MIN;
    h->prev_poc_msb = 1 << 16;
    h->prev_frame_num = -1;
    h->x264_build   = -1;
    h->sei_fpa.frame_packing_arrangement_cancel_flag = -1;
    ff_h264_reset_sei(h);
    if (avctx->codec_id == AV_CODEC_ID_H264) {
        if (avctx->ticks_per_frame == 1) {
            if(h->avctx->time_base.den < INT_MAX/2) {
                h->avctx->time_base.den *= 2;
            } else
                h->avctx->time_base.num /= 2;
        }
        avctx->ticks_per_frame = 2;
    }
    //AVCodecContext中是否包含extradata？包含的话，则解析之
    if (avctx->extradata_size > 0 && avctx->extradata) {
        ret = ff_h264_decode_extradata(h, avctx->extradata, avctx->extradata_size);
        if (ret < 0) {
            ff_h264_free_context(h);
            return ret;
        }
    }

    if (h->sps.bitstream_restriction_flag &&
        h->avctx->has_b_frames < h->sps.num_reorder_frames) {
        h->avctx->has_b_frames = h->sps.num_reorder_frames;
        h->low_delay           = 0;
    }

    avctx->internal->allocate_progress = 1;

    ff_h264_flush_change(h);

    return 0;
}
```

从函数定义中可以看出，ff_h264_decode_init()一方面给H.264 解码器中一些变量（例如bit_depth_luma、chroma_format_idc等）设定了初始值，另一方面调用了一系列汇编函数的初始化函数（初始化函数的具体内容在后续文章中完成）。初始化汇编函数的的步骤是：首先将C语言版本函数赋值给相应模块的函数指针；然后检测平台的特性，如果不支持汇编优化（ARM、X86等），则不再做任何处理，如果支持汇编优化，则将相应的汇编优化函数赋值给相应模块的函数指针（替换掉C语言版本的效率较低的函数）。下面几个函数初始化了几个不同模块的汇编优化函数：

ff_h264dsp_init()：初始化DSP相关的汇编函数。包含了IDCT、环路滤波函数等。
ff_h264qpel_init()：初始化四分之一像素运动补偿相关的汇编函数。
ff_h264_pred_init()：初始化帧内预测相关的汇编函数。
可以举例看一下个ff_h264_pred_init()的代码。

ff_h264_pred_init()

函数用于初始化帧内预测相关的汇编函数，定位于libavcodec\h264pred.c，如下所示。

```
[cpp]
```

```cpp
/**
 * Set the intra prediction function pointers.
 */
//初始化帧内预测相关的汇编函数
av_cold void ff_h264_pred_init(H264PredContext *h, int codec_id,
                               const int bit_depth,
                               int chroma_format_idc)
{
#undef FUNC
#undef FUNCC
#define FUNC(a, depth) a ## _ ## depth
#define FUNCC(a, depth) a ## _ ## depth ## _c
#define FUNCD(a) a ## _c
//好长的宏定义...（这种很长的宏定义在H.264解码器中似乎很普遍！）
//该宏用于给帧内预测模块的函数指针赋值
//注意参数为颜色位深度
#define H264_PRED(depth) \
    if(codec_id != AV_CODEC_ID_RV40){\
        if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {\
            h->pred4x4[VERT_PRED       ]= FUNCD(pred4x4_vertical_vp8);\
            h->pred4x4[HOR_PRED        ]= FUNCD(pred4x4_horizontal_vp8);\
        } else {\
            h->pred4x4[VERT_PRED       ]= FUNCC(pred4x4_vertical          , depth);\
            h->pred4x4[HOR_PRED        ]= FUNCC(pred4x4_horizontal        , depth);\
        }\
        h->pred4x4[DC_PRED             ]= FUNCC(pred4x4_dc                , depth);\
        if(codec_id == AV_CODEC_ID_SVQ3)\
            h->pred4x4[DIAG_DOWN_LEFT_PRED ]= FUNCD(pred4x4_down_left_svq3);\
        else\
            h->pred4x4[DIAG_DOWN_LEFT_PRED ]= FUNCC(pred4x4_down_left     , depth);\
        h->pred4x4[DIAG_DOWN_RIGHT_PRED]= FUNCC(pred4x4_down_right        , depth);\
        h->pred4x4[VERT_RIGHT_PRED     ]= FUNCC(pred4x4_vertical_right    , depth);\
        h->pred4x4[HOR_DOWN_PRED       ]= FUNCC(pred4x4_horizontal_down   , depth);\
        if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {\
            h->pred4x4[VERT_LEFT_PRED  ]= FUNCD(pred4x4_vertical_left_vp8);\
        } else\
            h->pred4x4[VERT_LEFT_PRED  ]= FUNCC(pred4x4_vertical_left     , depth);\
        h->pred4x4[HOR_UP_PRED         ]= FUNCC(pred4x4_horizontal_up     , depth);\
        if (codec_id != AV_CODEC_ID_VP7 && codec_id != AV_CODEC_ID_VP8) {\
            h->pred4x4[LEFT_DC_PRED    ]= FUNCC(pred4x4_left_dc           , depth);\
            h->pred4x4[TOP_DC_PRED     ]= FUNCC(pred4x4_top_dc            , depth);\
        } else {\
            h->pred4x4[TM_VP8_PRED     ]= FUNCD(pred4x4_tm_vp8);\
            h->pred4x4[DC_127_PRED     ]= FUNCC(pred4x4_127_dc            , depth);\
            h->pred4x4[DC_129_PRED     ]= FUNCC(pred4x4_129_dc            , depth);\
            h->pred4x4[VERT_VP8_PRED   ]= FUNCC(pred4x4_vertical          , depth);\
            h->pred4x4[HOR_VP8_PRED    ]= FUNCC(pred4x4_horizontal        , depth);\
        }\
        if (codec_id != AV_CODEC_ID_VP8)\
            h->pred4x4[DC_128_PRED     ]= FUNCC(pred4x4_128_dc            , depth);\
    }else{\
        h->pred4x4[VERT_PRED           ]= FUNCC(pred4x4_vertical          , depth);\
        h->pred4x4[HOR_PRED            ]= FUNCC(pred4x4_horizontal        , depth);\
        h->pred4x4[DC_PRED             ]= FUNCC(pred4x4_dc                , depth);\
        h->pred4x4[DIAG_DOWN_LEFT_PRED ]= FUNCD(pred4x4_down_left_rv40);\
        h->pred4x4[DIAG_DOWN_RIGHT_PRED]= FUNCC(pred4x4_down_right        , depth);\
        h->pred4x4[VERT_RIGHT_PRED     ]= FUNCC(pred4x4_vertical_right    , depth);\
        h->pred4x4[HOR_DOWN_PRED       ]= FUNCC(pred4x4_horizontal_down   , depth);\
        h->pred4x4[VERT_LEFT_PRED      ]= FUNCD(pred4x4_vertical_left_rv40);\
        h->pred4x4[HOR_UP_PRED         ]= FUNCD(pred4x4_horizontal_up_rv40);\
        h->pred4x4[LEFT_DC_PRED        ]= FUNCC(pred4x4_left_dc           , depth);\
        h->pred4x4[TOP_DC_PRED         ]= FUNCC(pred4x4_top_dc            , depth);\
        h->pred4x4[DC_128_PRED         ]= FUNCC(pred4x4_128_dc            , depth);\
        h->pred4x4[DIAG_DOWN_LEFT_PRED_RV40_NODOWN]= FUNCD(pred4x4_down_left_rv40_nodown);\
        h->pred4x4[HOR_UP_PRED_RV40_NODOWN]= FUNCD(pred4x4_horizontal_up_rv40_nodown);\
        h->pred4x4[VERT_LEFT_PRED_RV40_NODOWN]= FUNCD(pred4x4_vertical_left_rv40_nodown);\
    }\
\
    h->pred8x8l[VERT_PRED           ]= FUNCC(pred8x8l_vertical            , depth);\
    h->pred8x8l[HOR_PRED            ]= FUNCC(pred8x8l_horizontal          , depth);\
    h->pred8x8l[DC_PRED             ]= FUNCC(pred8x8l_dc                  , depth);\
    h->pred8x8l[DIAG_DOWN_LEFT_PRED ]= FUNCC(pred8x8l_down_left           , depth);\
    h->pred8x8l[DIAG_DOWN_RIGHT_PRED]= FUNCC(pred8x8l_down_right          , depth);\
    h->pred8x8l[VERT_RIGHT_PRED     ]= FUNCC(pred8x8l_vertical_right      , depth);\
    h->pred8x8l[HOR_DOWN_PRED       ]= FUNCC(pred8x8l_horizontal_down     , depth);\
    h->pred8x8l[VERT_LEFT_PRED      ]= FUNCC(pred8x8l_vertical_left       , depth);\
    h->pred8x8l[HOR_UP_PRED         ]= FUNCC(pred8x8l_horizontal_up       , depth);\
    h->pred8x8l[LEFT_DC_PRED        ]= FUNCC(pred8x8l_left_dc             , depth);\
    h->pred8x8l[TOP_DC_PRED         ]= FUNCC(pred8x8l_top_dc              , depth);\
    h->pred8x8l[DC_128_PRED         ]= FUNCC(pred8x8l_128_dc              , depth);\
\
    if (chroma_format_idc <= 1) {\
        h->pred8x8[VERT_PRED8x8   ]= FUNCC(pred8x8_vertical               , depth);\
        h->pred8x8[HOR_PRED8x8    ]= FUNCC(pred8x8_horizontal             , depth);\
    } else {\
        h->pred8x8[VERT_PRED8x8   ]= FUNCC(pred8x16_vertical              , depth);\
        h->pred8x8[HOR_PRED8x8    ]= FUNCC(pred8x16_horizontal            , depth);\
    }\
    if (codec_id != AV_CODEC_ID_VP7 && codec_id != AV_CODEC_ID_VP8) {\
        if (chroma_format_idc <= 1) {\
```

```
            h->pred8x8[PLANE_PRED8x8]= FUNCC(pred8x8_plane                , depth);\
        } else {\
            h->pred8x8[PLANE_PRED8x8]= FUNCC(pred8x16_plane               , depth);\
        }\
    } else\
        h->pred8x8[PLANE_PRED8x8]= FUNCD(pred8x8_tm_vp8);\
    if (codec_id != AV_CODEC_ID_RV40 && codec_id != AV_CODEC_ID_VP7 && \
        codec_id != AV_CODEC_ID_VP8) {\
        if (chroma_format_idc <= 1) {\
            h->pred8x8[DC_PRED8x8     ]= FUNCC(pred8x8_dc                     , depth);\
            h->pred8x8[LEFT_DC_PRED8x8]= FUNCC(pred8x8_left_dc                , depth);\
            h->pred8x8[TOP_DC_PRED8x8 ]= FUNCC(pred8x8_top_dc                 , depth);\
            h->pred8x8[ALZHEIMER_DC_L0T_PRED8x8 ]= FUNC(pred8x8_mad_cow_dc_l0t, depth);\
            h->pred8x8[ALZHEIMER_DC_0LT_PRED8x8 ]= FUNC(pred8x8_mad_cow_dc_0lt, depth);\
            h->pred8x8[ALZHEIMER_DC_L00_PRED8x8 ]= FUNC(pred8x8_mad_cow_dc_l00, depth);\
            h->pred8x8[ALZHEIMER_DC_0L0_PRED8x8 ]= FUNC(pred8x8_mad_cow_dc_0l0, depth);\
        } else {\
            h->pred8x8[DC_PRED8x8     ]= FUNCC(pred8x16_dc                    , depth);\
            h->pred8x8[LEFT_DC_PRED8x8]= FUNCC(pred8x16_left_dc               , depth);\
            h->pred8x8[TOP_DC_PRED8x8 ]= FUNCC(pred8x16_top_dc                , depth);\
            h->pred8x8[ALZHEIMER_DC_L0T_PRED8x8 ]= FUNC(pred8x16_mad_cow_dc_l0t, depth);\
            h->pred8x8[ALZHEIMER_DC_0LT_PRED8x8 ]= FUNC(pred8x16_mad_cow_dc_0lt, depth);\
            h->pred8x8[ALZHEIMER_DC_L00_PRED8x8 ]= FUNC(pred8x16_mad_cow_dc_l00, depth);\
            h->pred8x8[ALZHEIMER_DC_0L0_PRED8x8 ]= FUNC(pred8x16_mad_cow_dc_0l0, depth);\
        }\
    }else{\
        h->pred8x8[DC_PRED8x8     ]= FUNCD(pred8x8_dc_rv40);\
        h->pred8x8[LEFT_DC_PRED8x8]= FUNCD(pred8x8_left_dc_rv40);\
        h->pred8x8[TOP_DC_PRED8x8 ]= FUNCD(pred8x8_top_dc_rv40);\
        if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {\
            h->pred8x8[DC_127_PRED8x8]= FUNCC(pred8x8_127_dc              , depth);\
            h->pred8x8[DC_129_PRED8x8]= FUNCC(pred8x8_129_dc              , depth);\
        }\
    }\
    if (chroma_format_idc <= 1) {\
        h->pred8x8[DC_128_PRED8x8 ]= FUNCC(pred8x8_128_dc                 , depth);\
    } else {\
        h->pred8x8[DC_128_PRED8x8 ]= FUNCC(pred8x16_128_dc                , depth);\
    }\
\
    h->pred16x16[DC_PRED8x8     ]= FUNCC(pred16x16_dc                     , depth);\
    h->pred16x16[VERT_PRED8x8   ]= FUNCC(pred16x16_vertical               , depth);\
    h->pred16x16[HOR_PRED8x8    ]= FUNCC(pred16x16_horizontal             , depth);\
    switch(codec_id){\
    case AV_CODEC_ID_SVQ3:\
       h->pred16x16[PLANE_PRED8x8  ]= FUNCD(pred16x16_plane_svq3);\
       break;\
    case AV_CODEC_ID_RV40:\
       h->pred16x16[PLANE_PRED8x8  ]= FUNCD(pred16x16_plane_rv40);\
       break;\
    case AV_CODEC_ID_VP7:\
    case AV_CODEC_ID_VP8:\
       h->pred16x16[PLANE_PRED8x8  ]= FUNCD(pred16x16_tm_vp8);\
       h->pred16x16[DC_127_PRED8x8]= FUNCC(pred16x16_127_dc              , depth);\
       h->pred16x16[DC_129_PRED8x8]= FUNCC(pred16x16_129_dc              , depth);\
       break;\
    default:\
       h->pred16x16[PLANE_PRED8x8  ]= FUNCC(pred16x16_plane              , depth);\
       break;\
    }\
    h->pred16x16[LEFT_DC_PRED8x8]= FUNCC(pred16x16_left_dc                , depth);\
    h->pred16x16[TOP_DC_PRED8x8 ]= FUNCC(pred16x16_top_dc                 , depth);\
    h->pred16x16[DC_128_PRED8x8 ]= FUNCC(pred16x16_128_dc                 , depth);\
\
    /* special lossless h/v prediction for h264 */ \
    h->pred4x4_add  [VERT_PRED   ]= FUNCC(pred4x4_vertical_add            , depth);\
    h->pred4x4_add  [ HOR_PRED   ]= FUNCC(pred4x4_horizontal_add          , depth);\
    h->pred8x8l_add [VERT_PRED   ]= FUNCC(pred8x8l_vertical_add           , depth);\
    h->pred8x8l_add [ HOR_PRED   ]= FUNCC(pred8x8l_horizontal_add         , depth);\
    h->pred8x8l_filter_add [VERT_PRED   ]= FUNCC(pred8x8l_vertical_filter_add           , depth);\
    h->pred8x8l_filter_add [ HOR_PRED   ]= FUNCC(pred8x8l_horizontal_filter_add         , depth);\
    if (chroma_format_idc <= 1) {\
    h->pred8x8_add  [VERT_PRED8x8]= FUNCC(pred8x8_vertical_add            , depth);\
    h->pred8x8_add  [ HOR_PRED8x8]= FUNCC(pred8x8_horizontal_add          , depth);\
    } else {\
        h->pred8x8_add  [VERT_PRED8x8]= FUNCC(pred8x16_vertical_add            , depth);\
        h->pred8x8_add  [ HOR_PRED8x8]= FUNCC(pred8x16_horizontal_add          , depth);\
    }\
    h->pred16x16_add[VERT_PRED8x8]= FUNCC(pred16x16_vertical_add          , depth);\
    h->pred16x16_add[ HOR_PRED8x8]= FUNCC(pred16x16_horizontal_add        , depth);\
    //注意这里使用了前面那个很长的宏定义
    //根据颜色位深的不同，初始化不同的函数
    //颜色位深默认值为8，所以一般情况下调用H264_PRED(8)
    switch (bit_depth) {
        case 9:
            H264_PRED(9)
            break;
        case 10:
            H264_PRED(10)
            break;
        case 12:
```

```
            H264_PRED(12)
            break;
        case 14:
            H264_PRED(14)
            break;
        default:
            av_assert0(bit_depth<=8);
            H264_PRED(8)
            break;
    }
    //如果支持汇编优化，则会调用相应的汇编优化函数
    //neon这些的
    if (ARCH_ARM) ff_h264_pred_init_arm(h, codec_id, bit_depth, chroma_format_idc);
    //mmx这些的
    if (ARCH_X86) ff_h264_pred_init_x86(h, codec_id, bit_depth, chroma_format_idc);
}
```

初看一眼ff_h264_pred_init()定义会给人一种很奇怪的感觉：前面的这个H264_PRED(depth)的宏定义怎么这么长？！实际上在FFmpeg的H.264解码器中这种很长的宏定义是很常见的。我个人觉得这么做主要是为了方便为不同的颜色位深的码流初始化不同的功能函数。例如，对于常见的8bit码流，调用H264_PRED(8)就可以初始化相应的函数；对于比较新的10bit码流，调用H264_PRED(10)就可以初始化相应的函数。

ff_h264_pred_init()的代码是开始于switch()语句的，可以看出该函数根据不同的bit_depth（颜色位深）调用了不同的H264_PRED(bit_depth)宏。我们不妨展开一个H264_PRED()宏看看里面的代码究竟是什么。在这里我们选择最常见的8bit为例，看看H264_PRED(8)宏展开后的结果。

H264_PRED(8)

H264_PRED(8)用于初始化8bit颜色位深C语言版本的帧内预测的函数。该宏定义展开后的结果如下所示。

```cpp
if(codec_id != AV_CODEC_ID_RV40){
    if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {
        h->pred4x4[0       ]= pred4x4_vertical_vp8_c;
        h->pred4x4[1        ]= pred4x4_horizontal_vp8_c;
    } else {
        //帧内4x4的Vertical预测方式
        h->pred4x4[0       ]= pred4x4_vertical_8_c;
        //帧内4x4的Horizontal预测方式
        h->pred4x4[1        ]= pred4x4_horizontal_8_c;
    }
    //帧内4x4的DC预测方式
    h->pred4x4[2             ]= pred4x4_dc_8_c;
    if(codec_id == AV_CODEC_ID_SVQ3)
        h->pred4x4[3 ]= pred4x4_down_left_svq3_c;
    else
        h->pred4x4[3 ]= pred4x4_down_left_8_c;
    h->pred4x4[4]= pred4x4_down_right_8_c;
    h->pred4x4[5     ]= pred4x4_vertical_right_8_c;
    h->pred4x4[6       ]= pred4x4_horizontal_down_8_c;
    if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {
        h->pred4x4[7  ]= pred4x4_vertical_left_vp8_c;
    } else
        h->pred4x4[7  ]= pred4x4_vertical_left_8_c;
    h->pred4x4[8         ]= pred4x4_horizontal_up_8_c;
    if (codec_id != AV_CODEC_ID_VP7 && codec_id != AV_CODEC_ID_VP8) {
        h->pred4x4[9    ]= pred4x4_left_dc_8_c;
        h->pred4x4[10     ]= pred4x4_top_dc_8_c;
    } else {
        h->pred4x4[9     ]= pred4x4_tm_vp8_c;
        h->pred4x4[12     ]= pred4x4_127_dc_8_c;
        h->pred4x4[13     ]= pred4x4_129_dc_8_c;
        h->pred4x4[10   ]= pred4x4_vertical_8_c;
        h->pred4x4[14    ]= pred4x4_horizontal_8_c;
    }
    if (codec_id != AV_CODEC_ID_VP8)
        h->pred4x4[11     ]= pred4x4_128_dc_8_c;
}else{
    h->pred4x4[0           ]= pred4x4_vertical_8_c;
    h->pred4x4[1            ]= pred4x4_horizontal_8_c;
    h->pred4x4[2             ]= pred4x4_dc_8_c;
    h->pred4x4[3 ]= pred4x4_down_left_rv40_c;
    h->pred4x4[4]= pred4x4_down_right_8_c;
    h->pred4x4[5     ]= pred4x4_vertical_right_8_c;
    h->pred4x4[6       ]= pred4x4_horizontal_down_8_c;
    h->pred4x4[7      ]= pred4x4_vertical_left_rv40_c;
    h->pred4x4[8         ]= pred4x4_horizontal_up_rv40_c;
    h->pred4x4[9        ]= pred4x4_left_dc_8_c;
    h->pred4x4[10         ]= pred4x4_top_dc_8_c;
    h->pred4x4[11         ]= pred4x4_128_dc_8_c;
    h->pred4x4[12]= pred4x4_down_left_rv40_nodown_c;
    h->pred4x4[13]= pred4x4_horizontal_up_rv40_nodown_c;
    h->pred4x4[14]= pred4x4_vertical_left_rv40_nodown_c;
}

h->pred8x8l[0           ]= pred8x8l_vertical_8_c;
h->pred8x8l[1            ]= pred8x8l_horizontal_8_c;
h->pred8x8l[2             ]= pred8x8l_dc_8_c;
```

```
h->pred8x8l[3 ]= pred8x8l_down_left_8_c;
h->pred8x8l[4]= pred8x8l_down_right_8_c;
h->pred8x8l[5     ]= pred8x8l_vertical_right_8_c;
h->pred8x8l[6       ]= pred8x8l_horizontal_down_8_c;
h->pred8x8l[7      ]= pred8x8l_vertical_left_8_c;
h->pred8x8l[8         ]= pred8x8l_horizontal_up_8_c;
h->pred8x8l[9        ]= pred8x8l_left_dc_8_c;
h->pred8x8l[10         ]= pred8x8l_top_dc_8_c;
h->pred8x8l[11         ]= pred8x8l_128_dc_8_c;

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

h->pred16x16[0     ]= pred16x16_dc_8_c;
h->pred16x16[2   ]= pred16x16_vertical_8_c;
h->pred16x16[1    ]= pred16x16_horizontal_8_c;
switch(codec_id){
case AV_CODEC_ID_SVQ3:
   h->pred16x16[3  ]= pred16x16_plane_svq3_c;
   break;
case AV_CODEC_ID_RV40:
   h->pred16x16[3  ]= pred16x16_plane_rv40_c;
   break;
case AV_CODEC_ID_VP7:
case AV_CODEC_ID_VP8:
   h->pred16x16[3  ]= pred16x16_tm_vp8_c;
   h->pred16x16[7]= pred16x16_127_dc_8_c;
   h->pred16x16[8]= pred16x16_129_dc_8_c;
   break;
default:
   h->pred16x16[3  ]= pred16x16_plane_8_c;
   break;
}
h->pred16x16[4]= pred16x16_left_dc_8_c;
h->pred16x16[5 ]= pred16x16_top_dc_8_c;
h->pred16x16[6 ]= pred16x16_128_dc_8_c;

/* special lossless h/v prediction for h264 */
h->pred4x4_add  [0   ]= pred4x4_vertical_add_8_c;
h->pred4x4_add  [ 1   ]= pred4x4_horizontal_add_8_c;
h->pred8x8l_add [0   ]= pred8x8l_vertical_add_8_c;
h->pred8x8l_add [ 1   ]= pred8x8l_horizontal_add_8_c;
h->pred8x8l_filter_add [0   ]= pred8x8l_vertical_filter_add_8_c;
h->pred8x8l_filter_add [ 1   ]= pred8x8l_horizontal_filter_add_8_c;
if (chroma_format_idc <= 1) {
```

```
h->pred8x8_add  [2]= pred8x8_vertical_add_8_c;
h->pred8x8_add  [ 1]= pred8x8_horizontal_add_8_c;
} else {
    h->pred8x8_add  [2]= pred8x16_vertical_add_8_c;
    h->pred8x8_add  [ 1]= pred8x16_horizontal_add_8_c;
}
h->pred16x16_add[2]= pred16x16_vertical_add_8_c;
h->pred16x16_add[ 1]= pred16x16_horizontal_add_8_c;
```

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

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

在这里我们可以简单看一下pred4x4_vertical_8_c()函数。该函数完成了4x4帧内Vertical模式的预测。

pred4x4_vertical_8_c()

pred4x4_vertical_8_c()的定义位于libavcodec\h264pred_template.c，如下所示。

```cpp
//垂直预测
//由上边像素推出像素值
static void pred4x4_vertical_8_c (uint8_t *_src, const uint8_t *topright,
                                  ptrdiff_t _stride)
{
    pixel *src = (pixel*)_src;
    int stride = _stride>>(sizeof(pixel)-1);

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

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

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

}
```

有关pred4x4_vertical_8_c()的代码在后续文章中再做详细分析，在这里就不再做过多解释了。

ff_h264_pred_init_x86()

当系统支持ARM汇编优化的时候（ARCH_ ARM取值为1），就会调用ff_h264_pred_init_arm()初始化ARM平台下帧内预测汇编优化的函数；当系统支持X86汇编优化的时候（ARCH_X86取值为1），就会调用ff_h264_pred_init_x86()初始化X86平台下帧内预测汇编优化的函数。在这里我们简单看一下ff_h264_pred_init_x86()的定义。ff_h264_pred_init_x86()的定义位于libavcodec\x86\h264_intrapred_init.c，如下所示。

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

```
{
    int cpu_flags = av_get_cpu_flags();

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

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

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

        if (EXTERNAL_SSE2(cpu_flags)) {
            h->pred16x16[DC_PRED8x8           ] = ff_pred16x16_dc_8_sse2;
            h->pred8x8l [DIAG_DOWN_LEFT_PRED  ] = ff_pred8x8l_down_left_8_sse2;
            h->pred8x8l [DIAG_DOWN_RIGHT_PRED ] = ff_pred8x8l_down_right_8_sse2;
            h->pred8x8l [VERT_RIGHT_PRED      ] = ff_pred8x8l_vertical_right_8_sse2;
            h->pred8x8l [VERT_LEFT_PRED       ] = ff_pred8x8l_vertical_left_8_sse2;
```

```
            h->pred8x8l [HOR_DOWN_PRED        ] = ff_pred8x8l_horizontal_down_8_sse2;
            if (codec_id == AV_CODEC_ID_VP7 || codec_id == AV_CODEC_ID_VP8) {
                h->pred16x16[PLANE_PRED8x8    ] = ff_pred16x16_tm_vp8_8_sse2;
                h->pred8x8  [PLANE_PRED8x8    ] = ff_pred8x8_tm_vp8_8_sse2;
            } else {
                if (chroma_format_idc <= 1)
                    h->pred8x8  [PLANE_PRED8x8] = ff_pred8x8_plane_8_sse2;
                if (codec_id == AV_CODEC_ID_SVQ3) {
                    h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_svq3_8_sse2;
                } else if (codec_id == AV_CODEC_ID_RV40) {
                    h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_rv40_8_sse2;
                } else {
                    h->pred16x16[PLANE_PRED8x8] = ff_pred16x16_plane_h264_8_sse2;
                }
            }
        }

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

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

            h->pred8x8l[DC_128_PRED        ] = ff_pred8x8l_128_dc_10_mmxext;

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

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

            h->pred8x8l[VERT_PRED           ] = ff_pred8x8l_vertical_10_sse2;
            h->pred8x8l[HOR_PRED            ] = ff_pred8x8l_horizontal_10_sse2;
            h->pred8x8l[DC_PRED             ] = ff_pred8x8l_dc_10_sse2;
            h->pred8x8l[DC_128_PRED         ] = ff_pred8x8l_128_dc_10_sse2;
            h->pred8x8l[TOP_DC_PRED         ] = ff_pred8x8l_top_dc_10_sse2;
            h->pred8x8l[DIAG_DOWN_LEFT_PRED ] = ff_pred8x8l_down_left_10_sse2;
            h->pred8x8l[DIAG_DOWN_RIGHT_PRED] = ff_pred8x8l_down_right_10_sse2;
            h->pred8x8l[VERT_RIGHT_PRED     ] = ff_pred8x8l_vertical_right_10_sse2;
            h->pred8x8l[HOR_UP_PRED         ] = ff_pred8x8l_horizontal_up_10_sse2;

            h->pred16x16[DC_PRED8x8        ] = ff_pred16x16_dc_10_sse2;
            h->pred16x16[TOP_DC_PRED8x8    ] = ff_pred16x16_top_dc_10_sse2;
```

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

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

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

从源代码可以看出，ff_h264_pred_init_x86()首先调用av_get_cpu_flags()获取标记CPU特性的cpu_flags，然后根据cpu_flags初始化不同的函数，包括{xxx}_mmx()，{xxx}_mmxext()，{xxx}_sse()，{xxx}_sse2()，{xxx}_ssse3()，{xxx}_avx()几种采用不同会变指令的函数。

h264_decode_end()

h264_decode_end()用于关闭FFmpeg的H.264解码器。该函数的定义位于libavcodec\h264.c，如下所示。

```cpp
//关闭解码器
static av_cold int h264_decode_end(AVCodecContext *avctx)
{
    H264Context *h = avctx->priv_data;
    //移除参考帧
    ff_h264_remove_all_refs(h);
    //释放H264Context
    ff_h264_free_context(h);

    ff_h264_unref_picture(h, &h->cur_pic);

    return 0;
}
```

从函数定义中可以看出，h264_decode_end()调用了ff_h264_remove_all_refs()移除了所有的参考帧，然后又调用了ff_h264_free_context()释放了H264Context里面的所有内存。下面看一下这两个函数的定义。

ff_h264_remove_all_refs()

ff_h264_remove_all_refs()的定义如下所示。

```cpp
//移除参考帧
void ff_h264_remove_all_refs(H264Context *h)
{
    int i;
    //循环16次
    //长期参考帧
    for (i = 0; i < 16; i++) {
        remove_long(h, i, 0);
    }
    assert(h->long_ref_count == 0);
    //短期参考帧
    for (i = 0; i < h->short_ref_count; i++) {
        unreference_pic(h, h->short_ref[i], 0);
        h->short_ref[i] = NULL;
    }
    h->short_ref_count = 0;

    memset(h->default_ref_list, 0, sizeof(h->default_ref_list));
    memset(h->ref_list, 0, sizeof(h->ref_list));
}
```

从ff_h264_remove_all_refs()的定义中可以看出，该函数调用了remove_long()释放了长期参考帧，调用unreference_pic()释放了短期参考帧。

ff_h264_free_context()

ff_h264_free_context()的定义如下所示。

```cpp
//释放H264Context
av_cold void ff_h264_free_context(H264Context *h)
{
    int i;
    //释放各种内存
    ff_h264_free_tables(h, 1); // FIXME cleanup init stuff perhaps
    //释放SPS缓存
    for (i = 0; i < MAX_SPS_COUNT; i++)
        av_freep(h->sps_buffers + i);
    //释放PPS缓存
    for (i = 0; i < MAX_PPS_COUNT; i++)
        av_freep(h->pps_buffers + i);
}
```

从ff_h264_free_context()的定义可以看出，该函数调用了ff_h264_free_tables()释放H264Context中的各种内存。可以看一下该函数的定义。

ff_h264_free_tables()

ff_h264_free_tables()的定义如下所示。

```cpp
//释放各种内存
void ff_h264_free_tables(H264Context *h, int free_rbsp)
{
    int i;
    H264Context *hx;

    av_freep(&h->intra4x4_pred_mode);
    av_freep(&h->chroma_pred_mode_table);
    av_freep(&h->cbp_table);
    av_freep(&h->mvd_table[0]);
    av_freep(&h->mvd_table[1]);
    av_freep(&h->direct_table);
    av_freep(&h->non_zero_count);
    av_freep(&h->slice_table_base);
    h->slice_table = NULL;
    av_freep(&h->list_counts);

    av_freep(&h->mb2b_xy);
    av_freep(&h->mb2br_xy);

    av_buffer_pool_uninit(&h->qscale_table_pool);
    av_buffer_pool_uninit(&h->mb_type_pool);
    av_buffer_pool_uninit(&h->motion_val_pool);
    av_buffer_pool_uninit(&h->ref_index_pool);

    if (free_rbsp && h->DPB) {
        for (i = 0; i < H264_MAX_PICTURE_COUNT; i++)
            ff_h264_unref_picture(h, &h->DPB[i]);
        memset(h->delayed_pic, 0, sizeof(h->delayed_pic));
        av_freep(&h->DPB);
    } else if (h->DPB) {
        for (i = 0; i < H264_MAX_PICTURE_COUNT; i++)
            h->DPB[i].needs_realloc = 1;
    }

    h->cur_pic_ptr = NULL;

    for (i = 0; i < H264_MAX_THREADS; i++) {
        hx = h->thread_context[i];
        if (!hx)
            continue;
        av_freep(&hx->top_borders[1]);
        av_freep(&hx->top_borders[0]);
        av_freep(&hx->bipred_scratchpad);
        av_freep(&hx->edge_emu_buffer);
        av_freep(&hx->dc_val_base);
        av_freep(&hx->er.mb_index2xy);
        av_freep(&hx->er.error_status_table);
        av_freep(&hx->er.er_temp_buffer);
        av_freep(&hx->er.mbintra_table);
        av_freep(&hx->er.mbskip_table);

        if (free_rbsp) {
            av_freep(&hx->rbsp_buffer[1]);
            av_freep(&hx->rbsp_buffer[0]);
            hx->rbsp_buffer_size[0] = 0;
            hx->rbsp_buffer_size[1] = 0;
        }
        if (i)
            av_freep(&h->thread_context[i]);
    }
}
```

可以看出ff_h264_free_tables()调用了av_freep()等函数释放了H264Context中的各个内存。

h264_decode_frame()

h264_decode_frame()用于解码一帧图像数据。该函数的定义位于libavcodec\h264.c，如下所示。

```cpp
//H.264解码器-解码
static int h264_decode_frame(AVCodecContext *avctx, void *data,
                             int *got_frame, AVPacket *avpkt)
{
    //赋值。buf对应的就是AVPacket的data
    const uint8_t *buf = avpkt->data;
    int buf_size       = avpkt->size;
    //指向AVCodecContext的priv_data
    H264Context *h     = avctx->priv_data;
    AVFrame *pict      = data;
    int buf_index      = 0;
    H264Picture *out;
    int i, out_idx;
    int ret;

    h->flags = avctx->flags;
```

```
    /* reset data partitioning here, to ensure GetBitContexts from previous
     * packets do not get used. */
    h->data_partitioning = 0;

    /* end of stream, output what is still in the buffers */
    // Flush Decoder的时候会调用，此时输入为空的AVPacket====================
    if (buf_size == 0) {
out:

        h->cur_pic_ptr = NULL;
        h->first_field = 0;

        // FIXME factorize this with the output code below
        //输出out，源自于h->delayed_pic[]
        //初始化
        out     = h->delayed_pic[0];
        out_idx = 0;
        for (i = 1;
             h->delayed_pic[i] &&
             !h->delayed_pic[i]->f.key_frame &&
             !h->delayed_pic[i]->mmco_reset;
             i++)
            if (h->delayed_pic[i]->poc < out->poc) {
                //输出out，源自于h->delayed_pic[]
                //逐个处理
                out     = h->delayed_pic[i];
                out_idx = i;
            }

        for (i = out_idx; h->delayed_pic[i]; i++)
            h->delayed_pic[i] = h->delayed_pic[i + 1];

        if (out) {
            out->reference &= ~DELAYED_PIC_REF;
            //输出
            //out输出到pict
            //即H264Picture到AVFrame
            ret = output_frame(h, pict, out);
            if (ret < 0)
                return ret;
            *got_frame = 1;
        }

        return buf_index;
    }
    //==============================================================

    if (h->is_avc && av_packet_get_side_data(avpkt, AV_PKT_DATA_NEW_EXTRADATA, NULL)) {
        int side_size;
        uint8_t *side = av_packet_get_side_data(avpkt, AV_PKT_DATA_NEW_EXTRADATA, &side_size);
        if (is_extra(side, side_size))
            ff_h264_decode_extradata(h, side, side_size);
    }
    if(h->is_avc && buf_size >= 9 && buf[0]==1 && buf[2]==0 && (buf[4]&0xFC)==0xFC && (buf[5]&0x1F) && buf[8]==0x67){
        if (is_extra(buf, buf_size))
            return ff_h264_decode_extradata(h, buf, buf_size);
    }

    //关键：解码NALU最主要的函数
    //==============================================================
    buf_index = decode_nal_units(h, buf, buf_size, 0);
    //==============================================================
    if (buf_index < 0)
        return AVERROR_INVALIDDATA;

    if (!h->cur_pic_ptr && h->nal_unit_type == NAL_END_SEQUENCE) {
        av_assert0(buf_index <= buf_size);
        goto out;
    }

    if (!(avctx->flags2 & CODEC_FLAG2_CHUNKS) && !h->cur_pic_ptr) {
        if (avctx->skip_frame >= AVDISCARD_NONREF ||
            buf_size >= 4 && !memcmp("Q264", buf, 4))
            return buf_size;
        av_log(avctx, AV_LOG_ERROR, "no frame!\n");
        return AVERROR_INVALIDDATA;
    }

    if (!(avctx->flags2 & CODEC_FLAG2_CHUNKS) ||
        (h->mb_y >= h->mb_height && h->mb_height)) {
        if (avctx->flags2 & CODEC_FLAG2_CHUNKS)
            decode_postinit(h, 1);

        ff_h264_field_end(h, 0);

        /* Wait for second field. */
        //设置got_frame为0
        *got_frame = 0;
        if (h->next_output_pic && (
                                   h->next_output_pic->recovered)) {
            if (!h->next_output_pic->recovered)
```

```
                h->next_output_pic->f.flags |= AV_FRAME_FLAG_CORRUPT;
            //输出Frame
            //即H264Picture到AVFrame
            ret = output_frame(h, pict, h->next_output_pic);
            if (ret < 0)
                return ret;
            //设置got_frame为1
            *got_frame = 1;
            if (CONFIG_MPEGVIDEO) {
                ff_print_debug_info2(h->avctx, pict, h->er.mbskip_table,
                                     h->next_output_pic->mb_type,
                                     h->next_output_pic->qscale_table,
                                     h->next_output_pic->motion_val,
                                     &h->low_delay,
                                     h->mb_width, h->mb_height, h->mb_stride, 1);
            }
        }
    }

    assert(pict->buf[0] || !*got_frame);

    return get_consumed_bytes(buf_index, buf_size);
}
```

从源代码可以看出，h264_decode_frame()根据输入的AVPacket的data是否为空作不同的处理：

（1）若果输入的AVPacket的data为空，则调用output_frame()输出delayed_pic[]数组中的H264Picture，即输出解码器中缓存的帧（对应的是通常称为“Flush Decoder”的功能）。

（2）若果输入的AVPacket的data不为空，则首先调用decode_nal_units()解码AVPacket的data，然后再调用output_frame()输出解码后的视频帧（有一点需要注意：由于帧重排等因素，输出的AVFrame并非对应于输入的AVPacket）。

下面看一下解码压缩编码数据时候用到的函数decode_nal_units()。

decode_nal_units()

decode_nal_units()是用于解码NALU的函数。函数定义位于libavcodec\h264.c，如下所示。

```cpp
//解码NALU最主要的函数
//h264_decode_frame()中：
//buf一般是AVPacket->data
//buf_size一般是AVPacket->size
static int decode_nal_units(H264Context *h, const uint8_t *buf, int buf_size,
                            int parse_extradata)
{
    AVCodecContext *const avctx = h->avctx;
    H264Context *hx; ///< thread context
    int buf_index;
    unsigned context_count;
    int next_avc;
    int nals_needed = 0; ///< number of NALs that need decoding before the next frame thread starts
    int nal_index;
    int idr_cleared=0;
    int ret = 0;

    h->nal_unit_type= 0;

    if(!h->slice_context_count)
         h->slice_context_count= 1;
    h->max_contexts = h->slice_context_count;
    if (!(avctx->flags2 & CODEC_FLAG2_CHUNKS)) {
        h->current_slice = 0;
        if (!h->first_field)
            h->cur_pic_ptr = NULL;
        ff_h264_reset_sei(h);
    }

    //AVC1和H264的区别：
    //AVC1 描述:H.264 bitstream without start codes.是不带起始码0x00000001的。FLV/MKV/MOV种的H.264属于这种
    //H264 描述:H.264 bitstream with start codes.是带有起始码0x00000001的。H.264裸流，MPEGTS种的H.264属于这种
    //
    //通过VLC播放器，可以查看到具体的格式。打开视频后，通过菜单【工具】/【编解码信息】可以查看到【编解码器】具体格式，举例如下，编解码器信息：
    //编码: H264 – MPEG-4 AVC (part 10) (avc1)
    //编码: H264 – MPEG-4 AVC (part 10) (h264)
    //
    if (h->nal_length_size == 4) {
        if (buf_size > 8 && AV_RB32(buf) == 1 && AV_RB32(buf+5) > (unsigned)buf_size) {
            //前面4位是起始码0x00000001
            h->is_avc = 0;
        }else if(buf_size > 3 && AV_RB32(buf) > 1 && AV_RB32(buf) <= (unsigned)buf_size)
            //前面4位是长度数据
            h->is_avc = 1;
    }

    if (avctx->active_thread_type & FF_THREAD_FRAME)
        nals_needed = get_last_needed_nal(h, buf, buf_size);

    {
```

```
    {
        buf_index     = 0;
        context_count = 0;
        next_avc      = h->is_avc ? 0 : buf_size;
        nal_index     = 0;
        for (;;) {
            int consumed;
            int dst_length;
            int bit_length;
            const uint8_t *ptr;
            int nalsize = 0;
            int err;

            if (buf_index >= next_avc) {
                nalsize = get_avc_nalsize(h, buf, buf_size, &buf_index);
                if (nalsize < 0)
                    break;
                next_avc = buf_index + nalsize;
            } else {
                buf_index = find_start_code(buf, buf_size, buf_index, next_avc);
                if (buf_index >= buf_size)
                    break;
                if (buf_index >= next_avc)
                    continue;
            }

            hx = h->thread_context[context_count];
            //解析得到NAL（获得nal_unit_type等信息）
            ptr = ff_h264_decode_nal(hx, buf + buf_index, &dst_length,
                                     &consumed, next_avc - buf_index);
            if (!ptr || dst_length < 0) {
                ret = -1;
                goto end;
            }

            bit_length = get_bit_length(h, buf, ptr, dst_length,
                                        buf_index + consumed, next_avc);

            if (h->avctx->debug & FF_DEBUG_STARTCODE)
                av_log(h->avctx, AV_LOG_DEBUG,
                       "NAL %d/%d at %d/%d length %d\n",
                       hx->nal_unit_type, hx->nal_ref_idc, buf_index, buf_size, dst_length);

            if (h->is_avc && (nalsize != consumed) && nalsize)
                av_log(h->avctx, AV_LOG_DEBUG,
                       "AVC: Consumed only %d bytes instead of %d\n",
                       consumed, nalsize);

            buf_index += consumed;
            nal_index++;

            if (avctx->skip_frame >= AVDISCARD_NONREF &&
                h->nal_ref_idc == 0 &&
                h->nal_unit_type != NAL_SEI)
                continue;

again:
            if (   !(avctx->active_thread_type & FF_THREAD_FRAME)
                || nals_needed >= nal_index)
                h->au_pps_id = -1;
            /* Ignore per frame NAL unit type during extradata
             * parsing. Decoding slices is not possible in codec init
             * with frame-mt */
            if (parse_extradata) {
                switch (hx->nal_unit_type) {
                case NAL_IDR_SLICE:
                case NAL_SLICE:
                case NAL_DPA:
                case NAL_DPB:
                case NAL_DPC:
                    av_log(h->avctx, AV_LOG_WARNING,
                           "Ignoring NAL %d in global header/extradata\n",
                           hx->nal_unit_type);
                    // fall through to next case
                case NAL_AUXILIARY_SLICE:
                    hx->nal_unit_type = NAL_FF_IGNORE;
                }
            }

            err = 0;
            //根据不同的 NALU Type，调用不同的函数
            switch (hx->nal_unit_type) {
            //IDR帧
            case NAL_IDR_SLICE:
                if ((ptr[0] & 0xFC) == 0x98) {
                    av_log(h->avctx, AV_LOG_ERROR, "Invalid inter IDR frame\n");
                    h->next_outputed_poc = INT_MIN;
                    ret = -1;
                    goto end;
                }
                if (h->nal_unit_type != NAL_IDR_SLICE) {
                    av_log(h->avctx, AV_LOG_ERROR
```

```
                    av_log(h->avctx, AV_LOG_ERROR,
                           "Invalid mix of idr and non-idr slices\n");
                    ret = -1;
                    goto end;
                }
                if(!idr_cleared)
                    idr(h); // FIXME ensure we don't lose some frames if there is reordering
                idr_cleared = 1;
                h->has_recovery_point = 1;
                //注意没有break
            case NAL_SLICE:
                init_get_bits(&hx->gb, ptr, bit_length);
                hx->intra_gb_ptr      =
                hx->inter_gb_ptr      = &hx->gb;
                hx->data_partitioning = 0;
                //解码Slice Header
                if ((err = ff_h264_decode_slice_header(hx, h)))
                    break;

                if (h->sei_recovery_frame_cnt >= 0) {
                    if (h->frame_num != h->sei_recovery_frame_cnt || hx->slice_type_nos != AV_PICTURE_TYPE_I)
                        h->valid_recovery_point = 1;

                    if (   h->recovery_frame < 0
                        || ((h->recovery_frame - h->frame_num) & ((1 << h->sps.log2_max_frame_num)-1)) > h->sei_recovery_frame_cnt) {
                        h->recovery_frame = (h->frame_num + h->sei_recovery_frame_cnt) &
                                            ((1 << h->sps.log2_max_frame_num) - 1);

                        if (!h->valid_recovery_point)
                            h->recovery_frame = h->frame_num;
                    }
                }

                h->cur_pic_ptr->f.key_frame |=
                    (hx->nal_unit_type == NAL_IDR_SLICE);

                if (hx->nal_unit_type == NAL_IDR_SLICE ||
                    h->recovery_frame == h->frame_num) {
                    h->recovery_frame         = -1;
                    h->cur_pic_ptr->recovered = 1;
                }
                // If we have an IDR, all frames after it in decoded order are
                // "recovered".
                if (hx->nal_unit_type == NAL_IDR_SLICE)
                    h->frame_recovered |= FRAME_RECOVERED_IDR;
                h->frame_recovered |= 3*!!(avctx->flags2 & CODEC_FLAG2_SHOW_ALL);
                h->frame_recovered |= 3*!!(avctx->flags & CODEC_FLAG_OUTPUT_CORRUPT);
#if 1
                h->cur_pic_ptr->recovered |= h->frame_recovered;
#else
                h->cur_pic_ptr->recovered |= !!(h->frame_recovered & FRAME_RECOVERED_IDR);
#endif

                if (h->current_slice == 1) {
                    if (!(avctx->flags2 & CODEC_FLAG2_CHUNKS))
                        decode_postinit(h, nal_index >= nals_needed);

                    if (h->avctx->hwaccel &&
                        (ret = h->avctx->hwaccel->start_frame(h->avctx, NULL, 0)) < 0)
                        return ret;
                    if (CONFIG_H264_VDPAU_DECODER &&
                        h->avctx->codec->capabilities & CODEC_CAP_HWACCEL_VDPAU)
                        ff_vdpau_h264_picture_start(h);
                }

                if (hx->redundant_pic_count == 0) {
                    if (avctx->hwaccel) {
                        ret = avctx->hwaccel->decode_slice(avctx,
                                                           &buf[buf_index - consumed],
                                                           consumed);
                        if (ret < 0)
                            return ret;
                    } else if (CONFIG_H264_VDPAU_DECODER &&
                               h->avctx->codec->capabilities & CODEC_CAP_HWACCEL_VDPAU) {
                        ff_vdpau_add_data_chunk(h->cur_pic_ptr->f.data[0],
                                                start_code,
                                                sizeof(start_code));
                        ff_vdpau_add_data_chunk(h->cur_pic_ptr->f.data[0],
                                                &buf[buf_index - consumed],
                                                consumed);
                    } else
                        context_count++;
                }
                break;
            case NAL_DPA:
                if (h->avctx->flags & CODEC_FLAG2_CHUNKS) {
                    av_log(h->avctx, AV_LOG_ERROR,
                           "Decoding in chunks is not supported for "
                           "partitioned slices.\n");
                    return AVERROR(ENOSYS);
                }
```

```
232.                 }
233.
234.                 init_get_bits(&hx->gb, ptr, bit_length);
235.                 hx->intra_gb_ptr =
236.                 hx->inter_gb_ptr = NULL;
237.                 //解码Slice Header
238.                 if ((err = ff_h264_decode_slice_header(hx, h))) {
239.                     /* make sure data_partitioning is cleared if it was set
240.                      * before, so we don't try decoding a slice without a valid
241.                      * slice header later */
242.                     h->data_partitioning = 0;
243.                     break;
244.                 }
245.
246.                 hx->data_partitioning = 1;
247.                 break;
248.             case NAL_DPB:
249.                 init_get_bits(&hx->intra_gb, ptr, bit_length);
250.                 hx->intra_gb_ptr = &hx->intra_gb;
251.                 break;
252.             case NAL_DPC:
253.                 init_get_bits(&hx->inter_gb, ptr, bit_length);
254.                 hx->inter_gb_ptr = &hx->inter_gb;
255.
256.                 av_log(h->avctx, AV_LOG_ERROR, "Partitioned H.264 support is incomplete\n");
257.                 break;
258.
259.                 if (hx->redundant_pic_count == 0 &&
260.                     hx->intra_gb_ptr &&
261.                     hx->data_partitioning &&
262.                     h->cur_pic_ptr && h->context_initialized &&
263.                     (avctx->skip_frame < AVDISCARD_NONREF || hx->nal_ref_idc) &&
264.                     (avctx->skip_frame < AVDISCARD_BIDIR  ||
265.                      hx->slice_type_nos != AV_PICTURE_TYPE_B) &&
266.                     (avctx->skip_frame < AVDISCARD_NONINTRA ||
267.                      hx->slice_type_nos == AV_PICTURE_TYPE_I) &&
268.                     avctx->skip_frame < AVDISCARD_ALL)
269.                     context_count++;
270.                 break;
271.             case NAL_SEI:
272.                 init_get_bits(&h->gb, ptr, bit_length);
273.                 //解析SEI补充增强信息单元
274.                 ret = ff_h264_decode_sei(h);
275.                 if (ret < 0 && (h->avctx->err_recognition & AV_EF_EXPLODE))
276.                     goto end;
277.                 break;
278.             case NAL_SPS:
279.                 init_get_bits(&h->gb, ptr, bit_length);
280.                 //解析SPS序列参数集
281.                 if (ff_h264_decode_seq_parameter_set(h) < 0 && (h->is_avc ? nalsize : 1)) {
282.                     av_log(h->avctx, AV_LOG_DEBUG,
283.                            "SPS decoding failure, trying again with the complete NAL\n");
284.                     if (h->is_avc)
285.                         av_assert0(next_avc - buf_index + consumed == nalsize);
286.                     if ((next_avc - buf_index + consumed - 1) >= INT_MAX/8)
287.                         break;
288.                     init_get_bits(&h->gb, &buf[buf_index + 1 - consumed],
289.                                   8*(next_avc - buf_index + consumed - 1));
290.                     ff_h264_decode_seq_parameter_set(h);
291.                 }
292.
293.                 break;
294.                 //
295.             case NAL_PPS:
296.                 init_get_bits(&h->gb, ptr, bit_length);
297.                 //解析PPS图像参数集
298.                 ret = ff_h264_decode_picture_parameter_set(h, bit_length);
299.                 if (ret < 0 && (h->avctx->err_recognition & AV_EF_EXPLODE))
300.                     goto end;
301.                 break;
302.             case NAL_AUD:
303.             case NAL_END_SEQUENCE:
304.             case NAL_END_STREAM:
305.             case NAL_FILLER_DATA:
306.             case NAL_SPS_EXT:
307.             case NAL_AUXILIARY_SLICE:
308.                 break;
309.             case NAL_FF_IGNORE:
310.                 break;
311.             default:
312.                 av_log(avctx, AV_LOG_DEBUG, "Unknown NAL code: %d (%d bits)\n",
313.                        hx->nal_unit_type, bit_length);
314.             }
315.
316.             if (context_count == h->max_contexts) {
317.                 ret = ff_h264_execute_decode_slices(h, context_count);
318.                 if (ret < 0 && (h->avctx->err_recognition & AV_EF_EXPLODE))
319.                     goto end;
320.                 context_count = 0;
321.             }
322.
323.             if (err < 0 || err == SLICE_SKIPED) {
```

```
                if (err < 0)
                    av_log(h->avctx, AV_LOG_ERROR, "decode_slice_header error\n");
                h->ref_count[0] = h->ref_count[1] = h->list_count = 0;
            } else if (err == SLICE_SINGLETHREAD) {
                /* Slice could not be decoded in parallel mode, copy down
                 * NAL unit stuff to context 0 and restart. Note that
                 * rbsp_buffer is not transferred, but since we no longer
                 * run in parallel mode this should not be an issue. */
                h->nal_unit_type = hx->nal_unit_type;
                h->nal_ref_idc   = hx->nal_ref_idc;
                hx               = h;
                goto again;
            }
        }
    }
    if (context_count) {
        //真正的解码
        ret = ff_h264_execute_decode_slices(h, context_count);
        if (ret < 0 && (h->avctx->err_recognition & AV_EF_EXPLODE))
            goto end;
    }

    ret = 0;
end:
    /* clean up */
    if (h->cur_pic_ptr && !h->droppable) {
        ff_thread_report_progress(&h->cur_pic_ptr->tf, INT_MAX,
                                  h->picture_structure == PICT_BOTTOM_FIELD);
    }

    return (ret < 0) ? ret : buf_index;
}
```

从源代码可以看出，decode_nal_units()首先调用ff_h264_decode_nal()判断NALU的类型，然后根据NALU类型的不同调用了不同的处理函数。这些处理函数可以分为两类——解析函数和解码函数，如下所示。

（1）解析函数（获取信息）：

ff_h264_decode_seq_parameter_set()：解析SPS。

ff_h264_decode_picture_parameter_set()：解析PPS。

ff_h264_decode_sei()：解析SEI。

ff_h264_decode_slice_header()：解析Slice Header。

（2）解码函数（解码得到图像）：

ff_h264_execute_decode_slices()：解码Slice。

其中解析函数在文章《FFmpeg的H.264解码器源代码简单分析：解析器（Parser）部分》部分已经有过介绍，就不再重复叙述了。解码函数ff_h264_execute_decode_slices()完成了解码Slice的工作，下面看一下该函数的定义。

ff_h264_execute_decode_slices()

ff_h264_execute_decode_slices()用于解码获取图像信息，定义位于libavcodec\h264_slice.c，如下所示。

```cpp
/**
 * Call decode_slice() for each context.
 *
 * @param h h264 master context
 * @param context_count number of contexts to execute
 */
//真正的解码
int ff_h264_execute_decode_slices(H264Context *h, unsigned context_count)
{
    AVCodecContext *const avctx = h->avctx;
    H264Context *hx;
    int i;

    av_assert0(h->mb_y < h->mb_height);

    if (h->avctx->hwaccel ||
        h->avctx->codec->capabilities & CODEC_CAP_HWACCEL_VDPAU)
        return 0;
    //context_count的数量
    if (context_count == 1) {
        //解码Slice
        return decode_slice(avctx, &h);
    } else {
        av_assert0(context_count > 0);
        for (i = 1; i < context_count; i++) {
            hx                 = h->thread_context[i];
            if (CONFIG_ERROR_RESILIENCE) {
                hx->er.error_count = 0;
            }
            hx->x264_build     = h->x264_build;
        }

        avctx->execute(avctx, decode_slice, h->thread_context,
                       NULL, context_count, sizeof(void *));

        /* pull back stuff from slices to master context */
        hx                   = h->thread_context[context_count - 1];
        h->mb_x              = hx->mb_x;
        h->mb_y              = hx->mb_y;
        h->droppable         = hx->droppable;
        h->picture_structure = hx->picture_structure;
        if (CONFIG_ERROR_RESILIENCE) {
            for (i = 1; i < context_count; i++)
                h->er.error_count += h->thread_context[i]->er.error_count;
        }
    }

    return 0;
}
```

可以看出ff_h264_execute_decode_slices()调用了decode_slice()函数。在decode_slice()函数中完成了熵解码，宏块解码，环路滤波，错误隐藏等解码的细节工作。由于decode_slice()的内容比较多，本文暂不详细分析该函数，仅简单看一下该函数的定义。

decode_slice()

decode_slice()完成了熵解码，宏块解码，环路滤波，错误隐藏等解码的细节工作。该函数的定义位于定义位于libavcodec\h264_slice.c，如下所示。

```cpp
//解码slice
//三个主要步骤：
//1.熵解码（CAVLC/CABAC）
//2.宏块解码
//3.环路滤波
//此外还包含了错误隐藏代码
static int decode_slice(struct AVCodecContext *avctx, void *arg)
{
    H264Context *h = *(void **)arg;
    int lf_x_start = h->mb_x;

    h->mb_skip_run = -1;

    av_assert0(h->block_offset[15] == (4 * ((scan8[15] - scan8[0]) & 7) << h->pixel_shift) + 4 * h->linesize * ((scan8[15] - scan8[0]) >> 3));

    h->is_complex = FRAME_MBAFF(h) || h->picture_structure != PICT_FRAME ||
                    avctx->codec_id != AV_CODEC_ID_H264 ||
                    (CONFIG_GRAY && (h->flags & CODEC_FLAG_GRAY));

    if (!(h->avctx->active_thread_type & FF_THREAD_SLICE) && h->picture_structure == PICT_FRAME && h->er.error_status_table) {
        const int start_i  = av_clip(h->resync_mb_x + h->resync_mb_y * h->mb_width, 0, h->mb_num - 1);
        if (start_i) {
            int prev_status = h->er.error_status_table[h->er.mb_index2xy[start_i - 1]];
            prev_status &= ~ VP_START;
            if (prev_status != (ER_MV_END | ER_DC_END | ER_AC_END))
                h->er.error_occurred = 1;
        }
```

```
    }
    //CABAC情况
    if (h->pps.cabac) {
        /* realign */
        align_get_bits(&h->gb);

        /* init cabac */
        //初始化CABAC解码器
        ff_init_cabac_decoder(&h->cabac,
                              h->gb.buffer + get_bits_count(&h->gb) / 8,
                              (get_bits_left(&h->gb) + 7) / 8);

        ff_h264_init_cabac_states(h);
        //循环处理每个宏块
        for (;;) {
            // START_TIMER
            //解码CABAC数据
            int ret = ff_h264_decode_mb_cabac(h);
            int eos;
            // STOP_TIMER("decode_mb_cabac")
            //解码宏块
            if (ret >= 0)
                ff_h264_hl_decode_mb(h);

            // FIXME optimal? or let mb_decode decode 16x32 ?
            //宏块级帧场自适应。很少接触
            if (ret >= 0 && FRAME_MBAFF(h)) {
                h->mb_y++;

                ret = ff_h264_decode_mb_cabac(h);
                //解码宏块
                if (ret >= 0)
                    ff_h264_hl_decode_mb(h);
                h->mb_y--;
            }
            eos = get_cabac_terminate(&h->cabac);

            if ((h->workaround_bugs & FF_BUG_TRUNCATED) &&
                h->cabac.bytestream > h->cabac.bytestream_end + 2) {
                //错误隐藏
                er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x - 1,
                             h->mb_y, ER_MB_END);
                if (h->mb_x >= lf_x_start)
                    loop_filter(h, lf_x_start, h->mb_x + 1);
                return 0;
            }
            if (h->cabac.bytestream > h->cabac.bytestream_end + 2 )
                av_log(h->avctx, AV_LOG_DEBUG, "bytestream overread %"PTRDIFF_SPECIFIER"\n", h->cabac.bytestream_end - h->cabac.byte
stream);
            if (ret < 0 || h->cabac.bytestream > h->cabac.bytestream_end + 4) {
                av_log(h->avctx, AV_LOG_ERROR,
                       "error while decoding MB %d %d, bytestream %"PTRDIFF_SPECIFIER"\n",
                       h->mb_x, h->mb_y,
                       h->cabac.bytestream_end - h->cabac.bytestream);
                er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x,
                             h->mb_y, ER_MB_ERROR);
                return AVERROR_INVALIDDATA;
            }
            //mb_x自增
            //如果自增后超过了一行的mb个数
            if (++h->mb_x >= h->mb_width) {
                //环路滤波
                loop_filter(h, lf_x_start, h->mb_x);
                h->mb_x = lf_x_start = 0;
                decode_finish_row(h);
                //mb_y自增（处理下一行）
                ++h->mb_y;
                //宏块级帧场自适应，暂不考虑
                if (FIELD_OR_MBAFF_PICTURE(h)) {
                    ++h->mb_y;
                    if (FRAME_MBAFF(h) && h->mb_y < h->mb_height)
                        predict_field_decoding_flag(h);
                }
            }
            //如果mb_y超过了mb的行数
            if (eos || h->mb_y >= h->mb_height) {
                tprintf(h->avctx, "slice end %d %d\n",
                        get_bits_count(&h->gb), h->gb.size_in_bits);
                er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x - 1,
                             h->mb_y, ER_MB_END);
                if (h->mb_x > lf_x_start)
                    loop_filter(h, lf_x_start, h->mb_x);
                return 0;
            }
        }
    } else {
        //CAVLC情况
        //循环处理每个宏块
        for (;;) {
            //解码宏块的CAVLC
            int ret = ff_h264_decode_mb_cavlc(h);
```

```
            //解码宏块
            if (ret >= 0)
                ff_h264_hl_decode_mb(h);

            // FIXME optimal? or let mb_decode decode 16x32 ?
            if (ret >= 0 && FRAME_MBAFF(h)) {
                h->mb_y++;
                ret = ff_h264_decode_mb_cavlc(h);

                if (ret >= 0)
                    ff_h264_hl_decode_mb(h);
                h->mb_y--;
            }

            if (ret < 0) {
                av_log(h->avctx, AV_LOG_ERROR,
                       "error while decoding MB %d %d\n", h->mb_x, h->mb_y);
                er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x,
                             h->mb_y, ER_MB_ERROR);
                return ret;
            }

            if (++h->mb_x >= h->mb_width) {
                //环路滤波
                loop_filter(h, lf_x_start, h->mb_x);
                h->mb_x = lf_x_start = 0;
                decode_finish_row(h);
                ++h->mb_y;
                if (FIELD_OR_MBAFF_PICTURE(h)) {
                    ++h->mb_y;
                    if (FRAME_MBAFF(h) && h->mb_y < h->mb_height)
                        predict_field_decoding_flag(h);
                }
                if (h->mb_y >= h->mb_height) {
                    tprintf(h->avctx, "slice end %d %d\n",
                            get_bits_count(&h->gb), h->gb.size_in_bits);

                    if (   get_bits_left(&h->gb) == 0
                        || get_bits_left(&h->gb) > 0 && !(h->avctx->err_recognition & AV_EF_AGGRESSIVE)) {
                        //错误隐藏
                        er_add_slice(h, h->resync_mb_x, h->resync_mb_y,
                                     h->mb_x - 1, h->mb_y, ER_MB_END);

                        return 0;
                    } else {
                        er_add_slice(h, h->resync_mb_x, h->resync_mb_y,
                                     h->mb_x, h->mb_y, ER_MB_END);

                        return AVERROR_INVALIDDATA;
                    }
                }
            }

            if (get_bits_left(&h->gb) <= 0 && h->mb_skip_run <= 0) {
                tprintf(h->avctx, "slice end %d %d\n",
                        get_bits_count(&h->gb), h->gb.size_in_bits);

                if (get_bits_left(&h->gb) == 0) {
                    er_add_slice(h, h->resync_mb_x, h->resync_mb_y,
                                 h->mb_x - 1, h->mb_y, ER_MB_END);
                    if (h->mb_x > lf_x_start)
                        loop_filter(h, lf_x_start, h->mb_x);

                    return 0;
                } else {
                    er_add_slice(h, h->resync_mb_x, h->resync_mb_y, h->mb_x,
                                 h->mb_y, ER_MB_ERROR);

                    return AVERROR_INVALIDDATA;
                }
            }
        }
    }
}
```

从源代码可以看出，decode_slice()按照宏块（16x16）的方式处理输入的视频流。每个宏块的压缩数据经过以下3个基本步骤的处理，得到解码后的数据：

（1）熵解码。如果熵编码为CABAC，则调用ff_h264_decode_mb_cabac()；如果熵编码为CAVLC，则调用ff_h264_decode_mb_cavlc()

（2）宏块解码。这一步骤调用ff_h264_hl_decode_mb()

（3）环路滤波。这一步骤调用loop_filter()

此外，还有可能调用错误隐藏函数er_add_slice()。

至此，decode_nal_units()函数的调用流程就基本分析完毕了。h264_decode_frame()在调用完decode_nal_units()之后，还需要把解码后得到的H264Picture转换为AVFrame输出出来，这时候会调用一个相对比较简单的函数output_frame()。

output_frame()

output_frame()用于将一个H264Picture结构体转换为一个AVFrame结构体。该函数的定义位于libavcodec\h264.c，如下所示。

```cpp
//Flush Decoder的时候用到
//srcp输出到dst
//即H264Picture到AVFrame
static int output_frame(H264Context *h, AVFrame *dst, H264Picture *srcp)
{
    //src即H264Picture中的f
    AVFrame *src = &srcp->f;
    const AVPixFmtDescriptor *desc = av_pix_fmt_desc_get(src->format);
    int i;
    int ret = av_frame_ref(dst, src);
    if (ret < 0)
        return ret;

    av_dict_set(&dst->metadata, "stereo_mode", ff_h264_sei_stereo_mode(h), 0);

    if (srcp->sei_recovery_frame_cnt == 0)
        dst->key_frame = 1;
    if (!srcp->crop)
        return 0;

    for (i = 0; i < desc->nb_components; i++) {
        int hshift = (i > 0) ? desc->log2_chroma_w : 0;
        int vshift = (i > 0) ? desc->log2_chroma_h : 0;
        int off    = ((srcp->crop_left >> hshift) << h->pixel_shift) +
                     (srcp->crop_top  >> vshift) * dst->linesize[i];
        dst->data[i] += off;
    }
    return 0;
}
```

从源代码中可以看出，output_frame()实际上就是把H264Picture结构体中的“f”（AVFrame结构体）输出了出来。

至此，H.264解码器的主干部分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/45042755

文章标签： FFmpeg 解码器 NALU 初始化 源代码

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