原 FFmpeg的HEVC解码器源代码简单分析：解析器（Parser）部分

2015年06月09日 23:19:48　阅读数：13199

==========

HEVC源代码分析文章列表：

【解码 -libavcodec HEVC 解码器】

FFmpeg的HEVC解码器源代码简单分析：概述

FFmpeg的HEVC解码器源代码简单分析：解析器（Parser）部分

FFmpeg的HEVC解码器源代码简单分析：解码器主干部分

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

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

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

==========

上篇文章概述了FFmpeg的libavcodec中HEVC（H.265）解码器的结构；从这篇文章开始，具体研究HEVC解码器的源代码。本文分析HEVC解码器中解析器（Parser）部分的源代码。这部分的代码用于分割HEVC的NALU，并且解析SPS、PPS、SEI等信息。解析HEVC码流（对应AVCodecParser结构体中的函数）和解码HEVC码流（对应AVCodec结构体中的函数）的时候都会调用该部分的代码完成相应的功能。

函数调用关系图

FFmpeg HEVC解析器（Parser）部分在整个HEVC解码器中的位置如下图所示。

单击查看更清晰的大图

HEVC解析器（Parser）部分的源代码的调用关系如下图所示。

单击查看更清晰的大图

从图中可以看出，HEVC解析器调用了parse_nal_units()，HEVC解码器调用了decode_nal_units()，而上述两个函数都调用了下面几个解析函数：

ff_hevc_decode_nal_vps()：解析VPS。

ff_hevc_decode_nal_sps()：解析SPS。

ff_hevc_decode_nal_pps()：解析PPS。

ff_hevc_decode_nal_sei()：解析SEI。

下文将会分别这几个函数。

ff_hevc_decoder

ff_hevc_decoder是HEVC解码器对应的AVCodec结构体。该结构体的定义位于libavcodec\hevc.c，如下所示。

```cpp
AVCodec ff_hevc_decoder = {
    .name                  = "hevc",
    .long_name             = NULL_IF_CONFIG_SMALL("HEVC (High Efficiency Video Coding)"),
    .type                  = AVMEDIA_TYPE_VIDEO,
    .id                    = AV_CODEC_ID_HEVC,
    .priv_data_size        = sizeof(HEVCContext),
    .priv_class            = &hevc_decoder_class,
    .init                  = hevc_decode_init,
    .close                 = hevc_decode_free,
    .decode                = hevc_decode_frame,
    .flush                 = hevc_decode_flush,
    .update_thread_context = hevc_update_thread_context,
    .init_thread_copy      = hevc_init_thread_copy,
    .capabilities          = CODEC_CAP_DR1 | CODEC_CAP_DELAY |
                             CODEC_CAP_SLICE_THREADS | CODEC_CAP_FRAME_THREADS,
    .profiles              = NULL_IF_CONFIG_SMALL(profiles),
};
```

从源代码中可以看出，HEVC解码器的解码函数是hevc_decode_frame()。由于本文主要分析HEVC解析器，所以不对解码函数进行分析。在这里只需要知道hevc_decode_frame()调用了decode_nal_units()，而decode_nal_units()最终调用了ff_hevc_decode_nal_sps()等解析函数即可。

ff_hevc_parser

ff_hevc_parser是HEVC解析器对应的AVCodecParser结构体。该结构体的定义位于libavcodec\hevc_parser.c，如下所示。

```cpp
AVCodecParser ff_hevc_parser = {
    .codec_ids      = { AV_CODEC_ID_HEVC },
    .priv_data_size = sizeof(HEVCParseContext),
    .parser_init    = hevc_init,
    .parser_parse   = hevc_parse,
    .parser_close   = hevc_close,
    .split          = hevc_split,
};
```

从源代码可以看出，HEVC解析器的初始化函数是hevc_init()，解析函数是hevc_parse()，关闭函数是hevc_close()。

hevc_init()

hevc_init()是HEVC解析器的初始化函数，该函数的定义如下所示。

```cpp
static int hevc_init(AVCodecParserContext *s)
{
    HEVCContext  *h  = &((HEVCParseContext *)s->priv_data)->h;
    h->HEVClc = av_mallocz(sizeof(HEVCLocalContext));
    h->skipped_bytes_pos_size = INT_MAX;

    return 0;
}
```

可以看出hevc_init()简单地给内部成员变量分配了内存。

hevc_close()

hevc_close()是HEVC解析器的关闭函数，该函数的定义如下所示。

```cpp
static void hevc_close(AVCodecParserContext *s)
{
    int i;
    HEVCContext  *h  = &((HEVCParseContext *)s->priv_data)->h;
    ParseContext *pc = &((HEVCParseContext *)s->priv_data)->pc;

    av_freep(&h->skipped_bytes_pos);
    av_freep(&h->HEVClc);
    av_freep(&pc->buffer);

    for (i = 0; i < FF_ARRAY_ELEMS(h->vps_list); i++)
        av_buffer_unref(&h->vps_list[i]);
    for (i = 0; i < FF_ARRAY_ELEMS(h->sps_list); i++)
        av_buffer_unref(&h->sps_list[i]);
    for (i = 0; i < FF_ARRAY_ELEMS(h->pps_list); i++)
        av_buffer_unref(&h->pps_list[i]);

    av_buffer_unref(&h->current_sps);
    h->sps = NULL;

    for (i = 0; i < h->nals_allocated; i++)
        av_freep(&h->nals[i].rbsp_buffer);
    av_freep(&h->nals);
    h->nals_allocated = 0;
}
```

可以看出hevc_close()释放了内部成员变量的内存。

hevc_parse()

hevc_parse()是HEVC解析器中最关键的解析函数。该函数的定义如下所示。

```cpp
/*
 * 解析码流
 *
 * 注释：雷霄骅
 * leixiaohua1020@126.com
 * http://blog.csdn.net/leixiaohua1020
 */
static int hevc_parse(AVCodecParserContext *s,
                      AVCodecContext *avctx,
                      const uint8_t **poutbuf, int *poutbuf_size,
                      const uint8_t *buf, int buf_size)
{
    int next;
    ParseContext *pc = &((HEVCParseContext *)s->priv_data)->pc;
    //PARSER_FLAG_COMPLETE_FRAMES为1的时候说明传入的就是完整的1帧数据
    //这时候不用再分割NALU
    //PARSER_FLAG_COMPLETE_FRAMES为0的时候说明传入的是任意一段数据
    //需要先分离出完整的NALU
    if (s->flags & PARSER_FLAG_COMPLETE_FRAMES) {
        next = buf_size;
    } else {
        //分割NALU
        //通过查找起始码0x000001的方法
        next = hevc_find_frame_end(s, buf, buf_size);
        //合并
        if (ff_combine_frame(pc, next, &buf, &buf_size) < 0) {
            *poutbuf      = NULL;
            *poutbuf_size = 0;
            return buf_size;
        }
    }
    //解析NALU内容（不解码）
    parse_nal_units(s, avctx, buf, buf_size);

    *poutbuf      = buf;
    *poutbuf_size = buf_size;
    return next;
}
```

从源代码可以看出，hevc_parse()主要做了两步工作：

（1）判断传入的flags 中是否包含PARSER_FLAG_COMPLETE_FRAMES。如果包含，则说明传入的是完整的一帧数据，不作任何处理；如果不包含，则说明传入的不是完整的一帧数据而是任意一段HEVC数据，则需要调用hevc_find_frame_end()通过查找“起始码”（0x00000001或者0x000001）的方法，分离出完整的一帧数据。

（2）调用parse_nal_units()完成了NALU的解析工作。

下面分别看一下这两步中的两个函数。

hevc_find_frame_end()

hevc_find_frame_end()用于从HEVC码流中分离出完整的NALU。该函数的定义位于libavcodec\hevc_parser.c，如下所示。

```cpp
/**
 * Find the end of the current frame in the bitstream.
 * @return the position of the first byte of the next frame, or END_NOT_FOUND
 */
//分割NALU
static int hevc_find_frame_end(AVCodecParserContext *s, const uint8_t *buf,
                               int buf_size)
{
    int i;
    ParseContext *pc = &((HEVCParseContext *)s->priv_data)->pc;
    //一个一个字节进行处理
    for (i = 0; i < buf_size; i++) {
        int nut;
        //state64可以存8个字节
        //buf[i]存入state64
        pc->state64 = (pc->state64 << 8) | buf[i];

        //起始码定义#define START_CODE 0x000001
        //state64右移24bit之后，再对比是否为起始码0x000001
        if (((pc->state64 >> 3 * 8) & 0xFFFFFF) != START_CODE)
            continue;
        //找到起始码之后
        /*
         * 此时state64内容如下：
         *                  |      Start Code    | NALU Header |
         * |------|------|------|------|------|------|------|------|
         *
         *                |  buf |  buf |  buf |  buf |  buf | buf  |
         *                | [t-5]| [t-4]| [t-3]| [t-2]| [t-1]|  [t] |
         *
         * Start Code:
         * 0x000001
         *
         * NALU Header:
         * forbidden_zero_bit: 1bit。取值0。
         * nal_unit_type: 6 bit。NALU类型。
         * nuh_layer_id: 6 bit。目前取值为0（保留以后使用）.
         * nuh_temporal_id_plus1: 3 bit。减1后为NALU时域层标识号TemporalID。
         *
         */
        //state64右移16bit之后，state64最低字节为起始码后面的1Byte。即为NALU Header的前一个字节
        //NALU Header的前一个字节中，第1bit为forbidden_zero_bit，取值为0；
        //2-7bit为nal_unit_type；第8bit为nuh_layer_id，取值为0。

        //在这里state64右移(16+1)bit，然后相与0x3F(00111111)
        //即得到了nal_unit_type
        nut = (pc->state64 >> 2 * 8 + 1) & 0x3F;

        // Beginning of access unit
        if ((nut >= NAL_VPS && nut <= NAL_AUD) || nut == NAL_SEI_PREFIX ||
            (nut >= 41 && nut <= 44) || (nut >= 48 && nut <= 55)) {
            if (pc->frame_start_found) {
                pc->frame_start_found = 0;
                //返回起始码开始位置
                return i - 5;
            }
        } else if (nut <= NAL_RASL_R ||
                   (nut >= NAL_BLA_W_LP && nut <= NAL_CRA_NUT)) {
            int first_slice_segment_in_pic_flag = buf[i] >> 7;
            if (first_slice_segment_in_pic_flag) {
                if (!pc->frame_start_found) {
                    pc->frame_start_found = 1;
                } else { // First slice of next frame found
                    pc->frame_start_found = 0;
                    //返回起始码开始位置
                    return i - 5;
                }
            }
        }
    }

    return END_NOT_FOUND;
}
```

从源代码可以看出，hevc_find_frame_end()使用ParseContext中的state64临时缓存读取的字节。state64是一个uint64_t类型的变量，一共可以存储8Byte的数据。函数体的for()循环一次读取一个字节，读取完成后将该字节放入state64变量中；接着与起始码“0x000001”进行比较，如果不相等则继续读取，如果相等的话则提取NALU Header中nal_unit_type信息做相应处理后返回起始码开始的位置。

parse_nal_units()

parse_nal_units()用于解析一些NALU（VPS、SPS、PPS）的信息。该函数的定义位于libavcodec\hevc_parser.c，如下所示。

```cpp
/**
```

```
 * Parse NAL units of found picture and decode some basic information.
 *
 * @param s parser context.
 * @param avctx codec context.
 * @param buf buffer with field/frame data.
 * @param buf_size size of the buffer.
 *
 * 解析NALU内容（不解码）
 * 注释：雷霄骅
 * leixiaohua1020@126.com
 * http://blog.csdn.net/leixiaohua1020
 *
 */
static inline int parse_nal_units(AVCodecParserContext *s, AVCodecContext *avctx,
                      const uint8_t *buf, int buf_size)
{
    HEVCContext   *h  = &((HEVCParseContext *)s->priv_data)->h;
    GetBitContext *gb = &h->HEVClc->gb;
    SliceHeader   *sh = &h->sh;
    const uint8_t *buf_end = buf + buf_size;
    int state = -1, i;
    HEVCNAL *nal;

    /* set some sane default values */
    s->pict_type         = AV_PICTURE_TYPE_I;
    s->key_frame         = 0;
    s->picture_structure = AV_PICTURE_STRUCTURE_UNKNOWN;

    h->avctx = avctx;

    if (!buf_size)
        return 0;

    if (h->nals_allocated < 1) {
        HEVCNAL *tmp = av_realloc_array(h->nals, 1, sizeof(*tmp));
        if (!tmp)
            return AVERROR(ENOMEM);
        h->nals = tmp;
        memset(h->nals, 0, sizeof(*tmp));
        h->nals_allocated = 1;
    }

    nal = &h->nals[0];

    for (;;) {
        int src_length, consumed;
        buf = avpriv_find_start_code(buf, buf_end, &state);
        if (--buf + 2 >= buf_end)
            break;
        src_length = buf_end - buf;

        h->nal_unit_type = (*buf >> 1) & 0x3f;
        h->temporal_id   = (*(buf + 1) & 0x07) - 1;
        if (h->nal_unit_type <= NAL_CRA_NUT) {
            // Do not walk the whole buffer just to decode slice segment header
            if (src_length > 20)
                src_length = 20;
        }
        //类似于H.264解析器中的ff_h264_decode_nal()
        consumed = ff_hevc_extract_rbsp(h, buf, src_length, nal);
        if (consumed < 0)
            return consumed;

        init_get_bits8(gb, nal->data + 2, nal->size);
        /*
         * 几种NALU之间的关系
         *                            +--SS1
         *                            |
         *                +--PPS1<--+
         *                |         |
         *       +--SPS1<--+         +--SS2
         *       |         |
         * VPS<--+         +--PPS2
         *       |
         *       +--SPS2
         *
         */
        //解析不同种类的NALU
        switch (h->nal_unit_type) {
        case NAL_VPS:
            //解析VPS
            //VPS主要传输视频分级信息，有利于兼容可分级视频编码以及多视点视频编码
            ff_hevc_decode_nal_vps(h);
            break;
        case NAL_SPS:
            //解析SPS
            ff_hevc_decode_nal_sps(h);
            break;
        case NAL_PPS:
            //解析PPS
            ff_hevc_decode_nal_pps(h);
            break;
```

```
            break;
        case NAL_SEI_PREFIX:
        case NAL_SEI_SUFFIX:
            //解析SEI
            ff_hevc_decode_nal_sei(h);
            break;
        case NAL_TRAIL_N:
        case NAL_TRAIL_R:
        case NAL_TSA_N:
        case NAL_TSA_R:
        case NAL_STSA_N:
        case NAL_STSA_R:
        case NAL_RADL_N:
        case NAL_RADL_R:
        case NAL_RASL_N:
        case NAL_RASL_R:
        case NAL_BLA_W_LP:
        case NAL_BLA_W_RADL:
        case NAL_BLA_N_LP:
        case NAL_IDR_W_RADL:
        case NAL_IDR_N_LP:
        case NAL_CRA_NUT:

            //解析 SS Header

            //按照解码顺序，当前SS是否为第1个SS（Slice Segment）
            sh->first_slice_in_pic_flag = get_bits1(gb);
            s->picture_structure = h->picture_struct;
            s->field_order = h->picture_struct;

            //IRAP, Intra Random Access Point, 随机介入点
            //包括 IDR, CRA, BLA
            if (IS_IRAP(h)) {
                //设置关键帧
                s->key_frame = 1;
                sh->no_output_of_prior_pics_flag = get_bits1(gb);
            }
            //当前Slice引用的PPS的ID号
            sh->pps_id = get_ue_golomb(gb);
            if (sh->pps_id >= MAX_PPS_COUNT || !h->pps_list[sh->pps_id]) {
                av_log(h->avctx, AV_LOG_ERROR, "PPS id out of range: %d\n", sh->pps_id);
                return AVERROR_INVALIDDATA;
            }
            h->pps = (HEVCPPS*)h->pps_list[sh->pps_id]->data;

            if (h->pps->sps_id >= MAX_SPS_COUNT || !h->sps_list[h->pps->sps_id]) {
                av_log(h->avctx, AV_LOG_ERROR, "SPS id out of range: %d\n", h->pps->sps_id);
                return AVERROR_INVALIDDATA;
            }
            if (h->sps != (HEVCSPS*)h->sps_list[h->pps->sps_id]->data) {
                h->sps = (HEVCSPS*)h->sps_list[h->pps->sps_id]->data;
                h->vps = (HEVCVPS*)h->vps_list[h->sps->vps_id]->data;
            }
            //当前Slice不是第一个SS
            if (!sh->first_slice_in_pic_flag) {
                int slice_address_length;
                //当前SS是否依赖SS
                if (h->pps->dependent_slice_segments_enabled_flag)
                    sh->dependent_slice_segment_flag = get_bits1(gb);
                else
                    sh->dependent_slice_segment_flag = 0;

                slice_address_length = av_ceil_log2_c(h->sps->ctb_width *
                                                      h->sps->ctb_height);
                //当前SS中第一个CTU的地址
                sh->slice_segment_addr = get_bits(gb, slice_address_length);
                if (sh->slice_segment_addr >= h->sps->ctb_width * h->sps->ctb_height) {
                    av_log(h->avctx, AV_LOG_ERROR, "Invalid slice segment address: %u.\n",
                           sh->slice_segment_addr);
                    return AVERROR_INVALIDDATA;
                }
            } else
                sh->dependent_slice_segment_flag = 0;//独立SS

            if (sh->dependent_slice_segment_flag)//依赖SS
                break;

            for (i = 0; i < h->pps->num_extra_slice_header_bits; i++)
                skip_bits(gb, 1); // slice_reserved_undetermined_flag[]

            //slice type定义：
            //  0: B Slice
            //  1: P Slice
            //  2: I Slice
            sh->slice_type = get_ue_golomb(gb);//
            if (!(sh->slice_type == I_SLICE || sh->slice_type == P_SLICE ||
                  sh->slice_type == B_SLICE)) {
                av_log(h->avctx, AV_LOG_ERROR, "Unknown slice type: %d.\n",
                       sh->slice_type);
                return AVERROR_INVALIDDATA;
            }
            s->pict_type = sh->slice_type == B_SLICE ? AV_PICTURE_TYPE_B :
```

```
                                sh->slice_type == P_SLICE ? AV_PICTURE_TYPE_P :
                                                            AV_PICTURE_TYPE_I;

            if (h->pps->output_flag_present_flag)
                sh->pic_output_flag = get_bits1(gb);

            if (h->sps->separate_colour_plane_flag)
                sh->colour_plane_id = get_bits(gb, 2);

            if (!IS_IDR(h)) {
                //不是IDR，则计算POC
                sh->pic_order_cnt_lsb = get_bits(gb, h->sps->log2_max_poc_lsb);
                s->output_picture_number = h->poc = ff_hevc_compute_poc(h, sh->pic_order_cnt_lsb);
            } else
                s->output_picture_number = h->poc = 0;

            if (h->temporal_id == 0 &&
                h->nal_unit_type != NAL_TRAIL_N &&
                h->nal_unit_type != NAL_TSA_N &&
                h->nal_unit_type != NAL_STSA_N &&
                h->nal_unit_type != NAL_RADL_N &&
                h->nal_unit_type != NAL_RASL_N &&
                h->nal_unit_type != NAL_RADL_R &&
                h->nal_unit_type != NAL_RASL_R)
                h->pocTid0 = h->poc;

            return 0; /* no need to evaluate the rest */
        }
        buf += consumed;
    }
    /* didn't find a picture! */
    av_log(h->avctx, AV_LOG_ERROR, "missing picture in access unit\n");
    return -1;
}
```

从源代码可以看出，parse_nal_units()根据nal_unit_type的不同，调用不同的解析函数进行处理。例如：

a)解析VPS的时候调用ff_hevc_decode_nal_vps()

b)解析SPS的时候调用ff_hevc_decode_nal_sps()

c)解析PPS的时候调用ff_hevc_decode_nal_pps()

d)解析SEI的时候调用ff_hevc_decode_nal_sei()

e)解析SS Header的一部分信息。

下文简单分析这几种NALU的解析函数。

ff_hevc_decode_nal_vps()

目前还没有研究过VPS，所以没有分析该函数。

ff_hevc_decode_nal_sps()

ff_hevc_decode_nal_sps()用于解析HEVC码流中的SPS。该函数的定义位于libavcodec\hevc_ps.c，如下所示。

```cpp
//解析SPS
int ff_hevc_decode_nal_sps(HEVCContext *s)
{
    const AVPixFmtDescriptor *desc;
    GetBitContext *gb = &s->HEVClc->gb;
    int ret = 0;
    unsigned int sps_id = 0;
    int log2_diff_max_min_transform_block_size;
    int bit_depth_chroma, start, vui_present, sublayer_ordering_info;
    int i;

    HEVCSPS *sps;
    AVBufferRef *sps_buf = av_buffer_allocz(sizeof(*sps));

    if (!sps_buf)
        return AVERROR(ENOMEM);
    sps = (HEVCSPS*)sps_buf->data;

    av_log(s->avctx, AV_LOG_DEBUG, "Decoding SPS\n");

    // Coded parameters
    // 当前引用的VPS的ID
    sps->vps_id = get_bits(gb, 4);
    if (sps->vps_id >= MAX_VPS_COUNT) {
        av_log(s->avctx, AV_LOG_ERROR, "VPS id out of range: %d\n", sps->vps_id);
        ret = AVERROR_INVALIDDATA;
        goto err;
    }

    if (!s->vps_list[sps->vps_id]) {
```

```
        av_log(s->avctx, AV_LOG_ERROR, "VPS %d does not exist\n",
               sps->vps_id);
        ret = AVERROR_INVALIDDATA;
        goto err;
    }
    //时域子层的最大数目
    sps->max_sub_layers = get_bits(gb, 3) + 1;
    if (sps->max_sub_layers > MAX_SUB_LAYERS) {
        av_log(s->avctx, AV_LOG_ERROR, "sps_max_sub_layers out of range: %d\n",
               sps->max_sub_layers);
        ret = AVERROR_INVALIDDATA;
        goto err;
    }

    skip_bits1(gb); // temporal_id_nesting_flag

    if (parse_ptl(s, &sps->ptl, sps->max_sub_layers) < 0)
        goto err;
    //当前SPS的ID
    sps_id = get_ue_golomb_long(gb);
    if (sps_id >= MAX_SPS_COUNT) {
        av_log(s->avctx, AV_LOG_ERROR, "SPS id out of range: %d\n", sps_id);
        ret = AVERROR_INVALIDDATA;
        goto err;
    }
    /*
     * chroma_format_idc色度取样格式
     * 0: Y
     * 1: YUV420P
     * 2: YUV422P
     * 3: YUV444P
     */
    sps->chroma_format_idc = get_ue_golomb_long(gb);
    if (!(sps->chroma_format_idc == 1 || sps->chroma_format_idc == 2 || sps->chroma_format_idc == 3)) {
        avpriv_report_missing_feature(s->avctx, "chroma_format_idc != {1, 2, 3}\n");
        ret = AVERROR_PATCHWELCOME;
        goto err;
    }
    //YUV444的时候，标记是否对3个分量单独编码
    if (sps->chroma_format_idc == 3)
        sps->separate_colour_plane_flag = get_bits1(gb);

    if (sps->separate_colour_plane_flag)
        sps->chroma_format_idc = 0;
    //宽和高
    sps->width  = get_ue_golomb_long(gb);
    sps->height = get_ue_golomb_long(gb);
    if ((ret = av_image_check_size(sps->width,
                                   sps->height, 0, s->avctx)) < 0)
        goto err;
    //裁剪相关
    if (get_bits1(gb)) { // pic_conformance_flag
        //TODO: * 2 is only valid for 420
        sps->pic_conf_win.left_offset   = get_ue_golomb_long(gb) * 2;
        sps->pic_conf_win.right_offset  = get_ue_golomb_long(gb) * 2;
        sps->pic_conf_win.top_offset    = get_ue_golomb_long(gb) * 2;
        sps->pic_conf_win.bottom_offset = get_ue_golomb_long(gb) * 2;

        if (s->avctx->flags2 & CODEC_FLAG2_IGNORE_CROP) {
            av_log(s->avctx, AV_LOG_DEBUG,
                   "discarding sps conformance window, "
                   "original values are l:%u r:%u t:%u b:%u\n",
                   sps->pic_conf_win.left_offset,
                   sps->pic_conf_win.right_offset,
                   sps->pic_conf_win.top_offset,
                   sps->pic_conf_win.bottom_offset);

            sps->pic_conf_win.left_offset   =
            sps->pic_conf_win.right_offset  =
            sps->pic_conf_win.top_offset    =
            sps->pic_conf_win.bottom_offset = 0;
        }
        sps->output_window = sps->pic_conf_win;
    }
    //亮度像素的颜色位深
    sps->bit_depth   = get_ue_golomb_long(gb) + 8;
    //色度像素的颜色位深
    bit_depth_chroma = get_ue_golomb_long(gb) + 8;
    if (bit_depth_chroma != sps->bit_depth) {
        av_log(s->avctx, AV_LOG_ERROR,
               "Luma bit depth (%d) is different from chroma bit depth (%d), "
               "this is unsupported.\n",
               sps->bit_depth, bit_depth_chroma);
        ret = AVERROR_INVALIDDATA;
        goto err;
    }
    //根据颜色位深和色度采样格式设定pix_fmt
    switch (sps->bit_depth) {
    case 8:
        if (sps->chroma_format_idc == 1) sps->pix_fmt = AV_PIX_FMT_YUV420P;
        if (sps->chroma_format_idc == 2) sps->pix_fmt = AV_PIX_FMT_YUV422P;
```

```
        if (sps->chroma_format_idc == 3) sps->pix_fmt = AV_PIX_FMT_YUV444P;
       break;
    case 9:
        if (sps->chroma_format_idc == 1) sps->pix_fmt = AV_PIX_FMT_YUV420P9;
        if (sps->chroma_format_idc == 2) sps->pix_fmt = AV_PIX_FMT_YUV422P9;
        if (sps->chroma_format_idc == 3) sps->pix_fmt = AV_PIX_FMT_YUV444P9;
        break;
    case 10:
        if (sps->chroma_format_idc == 1) sps->pix_fmt = AV_PIX_FMT_YUV420P10;
        if (sps->chroma_format_idc == 2) sps->pix_fmt = AV_PIX_FMT_YUV422P10;
        if (sps->chroma_format_idc == 3) sps->pix_fmt = AV_PIX_FMT_YUV444P10;
        break;
    case 12:
        if (sps->chroma_format_idc == 1) sps->pix_fmt = AV_PIX_FMT_YUV420P12;
        if (sps->chroma_format_idc == 2) sps->pix_fmt = AV_PIX_FMT_YUV422P12;
        if (sps->chroma_format_idc == 3) sps->pix_fmt = AV_PIX_FMT_YUV444P12;
        break;
    default:
        av_log(s->avctx, AV_LOG_ERROR,
               "4:2:0, 4:2:2, 4:4:4 supports are currently specified for 8, 10 and 12 bits.\n");
        ret = AVERROR_PATCHWELCOME;
        goto err;
    }

    desc = av_pix_fmt_desc_get(sps->pix_fmt);
    if (!desc) {
        ret = AVERROR(EINVAL);
        goto err;
    }

    sps->hshift[0] = sps->vshift[0] = 0;
    sps->hshift[2] = sps->hshift[1] = desc->log2_chroma_w;
    sps->vshift[2] = sps->vshift[1] = desc->log2_chroma_h;

    sps->pixel_shift = sps->bit_depth > 8;

    //用于计算POC
    sps->log2_max_poc_lsb = get_ue_golomb_long(gb) + 4;
    if (sps->log2_max_poc_lsb > 16) {
        av_log(s->avctx, AV_LOG_ERROR, "log2_max_pic_order_cnt_lsb_minus4 out range: %d\n",
               sps->log2_max_poc_lsb - 4);
        ret = AVERROR_INVALIDDATA;
        goto err;
    }

    sublayer_ordering_info = get_bits1(gb);
    start = sublayer_ordering_info ? 0 : sps->max_sub_layers - 1;
    for (i = start; i < sps->max_sub_layers; i++) {
        sps->temporal_layer[i].max_dec_pic_buffering = get_ue_golomb_long(gb) + 1;
        sps->temporal_layer[i].num_reorder_pics      = get_ue_golomb_long(gb);
        sps->temporal_layer[i].max_latency_increase  = get_ue_golomb_long(gb) - 1;
        if (sps->temporal_layer[i].max_dec_pic_buffering > MAX_DPB_SIZE) {
            av_log(s->avctx, AV_LOG_ERROR, "sps_max_dec_pic_buffering_minus1 out of range: %d\n",
                   sps->temporal_layer[i].max_dec_pic_buffering - 1);
            ret = AVERROR_INVALIDDATA;
            goto err;
        }
        if (sps->temporal_layer[i].num_reorder_pics > sps->temporal_layer[i].max_dec_pic_buffering - 1) {
            av_log(s->avctx, AV_LOG_WARNING, "sps_max_num_reorder_pics out of range: %d\n",
                   sps->temporal_layer[i].num_reorder_pics);
            if (s->avctx->err_recognition & AV_EF_EXPLODE ||
                sps->temporal_layer[i].num_reorder_pics > MAX_DPB_SIZE - 1) {
                ret = AVERROR_INVALIDDATA;
                goto err;
            }
            sps->temporal_layer[i].max_dec_pic_buffering = sps->temporal_layer[i].num_reorder_pics + 1;
        }
    }

    if (!sublayer_ordering_info) {
        for (i = 0; i < start; i++) {
            sps->temporal_layer[i].max_dec_pic_buffering = sps->temporal_layer[start].max_dec_pic_buffering;
            sps->temporal_layer[i].num_reorder_pics      = sps->temporal_layer[start].num_reorder_pics;
            sps->temporal_layer[i].max_latency_increase  = sps->temporal_layer[start].max_latency_increase;
        }
    }
    //亮度编码块-最小尺寸
    sps->log2_min_cb_size                    = get_ue_golomb_long(gb) + 3;
    //亮度编码块-最大尺寸和最小尺寸插值
    sps->log2_diff_max_min_coding_block_size = get_ue_golomb_long(gb);
    //亮度变换块-最小尺寸
    sps->log2_min_tb_size                    = get_ue_golomb_long(gb) + 2;
    //亮度变换块-最大尺寸和最小尺寸插值
    log2_diff_max_min_transform_block_size   = get_ue_golomb_long(gb);
    sps->log2_max_trafo_size                 = log2_diff_max_min_transform_block_size +
                                               sps->log2_min_tb_size;

    if (sps->log2_min_tb_size >= sps->log2_min_cb_size) {
        av_log(s->avctx, AV_LOG_ERROR, "Invalid value for log2_min_tb_size");
        ret = AVERROR_INVALIDDATA;
        goto err;
```

```
    }
    //帧间预测变换块-最大划分深度
    sps->max_transform_hierarchy_depth_inter = get_ue_golomb_long(gb);
    //帧内预测变换块-最大划分深度
    sps->max_transform_hierarchy_depth_intra = get_ue_golomb_long(gb);
    //是否使用量化矩阵
    sps->scaling_list_enable_flag = get_bits1(gb);
    if (sps->scaling_list_enable_flag) {
        set_default_scaling_list_data(&sps->scaling_list);

        if (get_bits1(gb)) {
            ret = scaling_list_data(s, &sps->scaling_list, sps);
            if (ret < 0)
                goto err;
        }
    }
    //是否使用非对称划分模式
    sps->amp_enabled_flag = get_bits1(gb);
    //是否在去块效应滤波过程中使用样点自适应补偿SAO
    sps->sao_enabled      = get_bits1(gb);
    //允许PCM编码
    sps->pcm_enabled_flag = get_bits1(gb);
    //有关PCM编码的参数
    if (sps->pcm_enabled_flag) {
        sps->pcm.bit_depth   = get_bits(gb, 4) + 1;
        sps->pcm.bit_depth_chroma = get_bits(gb, 4) + 1;
        sps->pcm.log2_min_pcm_cb_size = get_ue_golomb_long(gb) + 3;
        sps->pcm.log2_max_pcm_cb_size = sps->pcm.log2_min_pcm_cb_size +
                                        get_ue_golomb_long(gb);
        if (sps->pcm.bit_depth > sps->bit_depth) {
            av_log(s->avctx, AV_LOG_ERROR,
                   "PCM bit depth (%d) is greater than normal bit depth (%d)\n",
                   sps->pcm.bit_depth, sps->bit_depth);
            ret = AVERROR_INVALIDDATA;
            goto err;
        }

        sps->pcm.loop_filter_disable_flag = get_bits1(gb);
    }
    //短期参考num_short_term_ref_pic_set
    sps->nb_st_rps = get_ue_golomb_long(gb);
    if (sps->nb_st_rps > MAX_SHORT_TERM_RPS_COUNT) {
        av_log(s->avctx, AV_LOG_ERROR, "Too many short term RPS: %d.\n",
               sps->nb_st_rps);
        ret = AVERROR_INVALIDDATA;
        goto err;
    }
    for (i = 0; i < sps->nb_st_rps; i++) {
        if ((ret = ff_hevc_decode_short_term_rps(s, &sps->st_rps[i],
                                                 sps, 0)) < 0)
            goto err;
    }
    //长期参考num_long_term_ref_pic_set
    sps->long_term_ref_pics_present_flag = get_bits1(gb);
    if (sps->long_term_ref_pics_present_flag) {
        sps->num_long_term_ref_pics_sps = get_ue_golomb_long(gb);
        if (sps->num_long_term_ref_pics_sps > 31U) {
            av_log(0, AV_LOG_ERROR, "num_long_term_ref_pics_sps %d is out of range.\n",
                   sps->num_long_term_ref_pics_sps);
            goto err;
        }
        for (i = 0; i < sps->num_long_term_ref_pics_sps; i++) {
            sps->lt_ref_pic_poc_lsb_sps[i]       = get_bits(gb, sps->log2_max_poc_lsb);
            sps->used_by_curr_pic_lt_sps_flag[i] = get_bits1(gb);
        }
    }
    //是否使用时域MV预测
    sps->sps_temporal_mvp_enabled_flag          = get_bits1(gb);
    //滤波过程是否使用双线性插值
    sps->sps_strong_intra_smoothing_enable_flag = get_bits1(gb);
    sps->vui.sar = (AVRational){0, 1};
    vui_present = get_bits1(gb);
    if (vui_present)
        decode_vui(s, sps);

    if (get_bits1(gb)) { // sps_extension_flag
        int sps_extension_flag[1];
        for (i = 0; i < 1; i++)
            sps_extension_flag[i] = get_bits1(gb);
        skip_bits(gb, 7); //sps_extension_7bits = get_bits(gb, 7);
        if (sps_extension_flag[0]) {
            int extended_precision_processing_flag;
            int high_precision_offsets_enabled_flag;
            int cabac_bypass_alignment_enabled_flag;

            sps->transform_skip_rotation_enabled_flag = get_bits1(gb);
            sps->transform_skip_context_enabled_flag  = get_bits1(gb);
            sps->implicit_rdpcm_enabled_flag = get_bits1(gb);

            sps->explicit_rdpcm_enabled_flag = get_bits1(gb);

            extended_precision_processing_flag = get_bits1(gb);
```

```
            extended_precision_processing_flag = get_bits1(gb);
            if (extended_precision_processing_flag)
                av_log(s->avctx, AV_LOG_WARNING,
                   "extended_precision_processing_flag not yet implemented\n");

            sps->intra_smoothing_disabled_flag       = get_bits1(gb);
            high_precision_offsets_enabled_flag  = get_bits1(gb);
            if (high_precision_offsets_enabled_flag)
                av_log(s->avctx, AV_LOG_WARNING,
                   "high_precision_offsets_enabled_flag not yet implemented\n");

            sps->persistent_rice_adaptation_enabled_flag = get_bits1(gb);

            cabac_bypass_alignment_enabled_flag  = get_bits1(gb);
            if (cabac_bypass_alignment_enabled_flag)
                av_log(s->avctx, AV_LOG_WARNING,
                   "cabac_bypass_alignment_enabled_flag not yet implemented\n");
        }
    }
    if (s->apply_defdispwin) {
        sps->output_window.left_offset   += sps->vui.def_disp_win.left_offset;
        sps->output_window.right_offset  += sps->vui.def_disp_win.right_offset;
        sps->output_window.top_offset    += sps->vui.def_disp_win.top_offset;
        sps->output_window.bottom_offset += sps->vui.def_disp_win.bottom_offset;
    }
    if (sps->output_window.left_offset & (0x1F >> (sps->pixel_shift)) &&
        !(s->avctx->flags & CODEC_FLAG_UNALIGNED)) {
        sps->output_window.left_offset &= ~(0x1F >> (sps->pixel_shift));
        av_log(s->avctx, AV_LOG_WARNING, "Reducing left output window to %d "
               "chroma samples to preserve alignment.\n",
               sps->output_window.left_offset);
    }
    sps->output_width  = sps->width -
                         (sps->output_window.left_offset + sps->output_window.right_offset);
    sps->output_height = sps->height -
                         (sps->output_window.top_offset + sps->output_window.bottom_offset);
    if (sps->output_width <= 0 || sps->output_height <= 0) {
        av_log(s->avctx, AV_LOG_WARNING, "Invalid visible frame dimensions: %dx%d.\n",
               sps->output_width, sps->output_height);
        if (s->avctx->err_recognition & AV_EF_EXPLODE) {
            ret = AVERROR_INVALIDDATA;
            goto err;
        }
        av_log(s->avctx, AV_LOG_WARNING,
               "Displaying the whole video surface.\n");
        memset(&sps->pic_conf_win, 0, sizeof(sps->pic_conf_win));
        memset(&sps->output_window, 0, sizeof(sps->output_window));
        sps->output_width               = sps->width;
        sps->output_height              = sps->height;
    }

    // Inferred parameters
    // 推算出来的参数
    sps->log2_ctb_size = sps->log2_min_cb_size +
                         sps->log2_diff_max_min_coding_block_size;
    sps->log2_min_pu_size = sps->log2_min_cb_size - 1;

    sps->ctb_width  = (sps->width  + (1 << sps->log2_ctb_size) - 1) >> sps->log2_ctb_size;
    sps->ctb_height = (sps->height + (1 << sps->log2_ctb_size) - 1) >> sps->log2_ctb_size;
    sps->ctb_size   = sps->ctb_width * sps->ctb_height;

    sps->min_cb_width  = sps->width  >> sps->log2_min_cb_size;
    sps->min_cb_height = sps->height >> sps->log2_min_cb_size;
    sps->min_tb_width  = sps->width  >> sps->log2_min_tb_size;
    sps->min_tb_height = sps->height >> sps->log2_min_tb_size;
    sps->min_pu_width  = sps->width  >> sps->log2_min_pu_size;
    sps->min_pu_height = sps->height >> sps->log2_min_pu_size;
    sps->tb_mask       = (1 << (sps->log2_ctb_size - sps->log2_min_tb_size)) - 1;

    sps->qp_bd_offset = 6 * (sps->bit_depth - 8);

    if (sps->width  & ((1 << sps->log2_min_cb_size) - 1) ||
        sps->height & ((1 << sps->log2_min_cb_size) - 1)) {
        av_log(s->avctx, AV_LOG_ERROR, "Invalid coded frame dimensions.\n");
        goto err;
    }

    if (sps->log2_ctb_size > MAX_LOG2_CTB_SIZE) {
        av_log(s->avctx, AV_LOG_ERROR, "CTB size out of range: 2^%d\n", sps->log2_ctb_size);
        goto err;
    }
    if (sps->max_transform_hierarchy_depth_inter > sps->log2_ctb_size - sps->log2_min_tb_size) {
        av_log(s->avctx, AV_LOG_ERROR, "max_transform_hierarchy_depth_inter out of range: %d\n",
               sps->max_transform_hierarchy_depth_inter);
        goto err;
    }
    if (sps->max_transform_hierarchy_depth_intra > sps->log2_ctb_size - sps->log2_min_tb_size) {
        av_log(s->avctx, AV_LOG_ERROR, "max_transform_hierarchy_depth_intra out of range: %d\n",
               sps->max_transform_hierarchy_depth_intra);
        goto err;
    }
    if (sps->log2_max_trafo_size > FFMIN(sps->log2_ctb_size, 5)) {
```

```
    if (sps->log2_max_trafo_size > FFMIN(sps->log2_ctb_size, 5)) {
        av_log(s->avctx, AV_LOG_ERROR,
               "max transform block size out of range: %d\n",
               sps->log2_max_trafo_size);
        goto err;
    }

    if (get_bits_left(gb) < 0) {
        av_log(s->avctx, AV_LOG_ERROR,
               "Overread SPS by %d bits\n", -get_bits_left(gb));
        goto err;
    }

    if (s->avctx->debug & FF_DEBUG_BITSTREAM) {
        av_log(s->avctx, AV_LOG_DEBUG,
               "Parsed SPS: id %d; coded wxh: %dx%d; "
               "cropped wxh: %dx%d; pix_fmt: %s.\n",
               sps_id, sps->width, sps->height,
               sps->output_width, sps->output_height,
               av_get_pix_fmt_name(sps->pix_fmt));
    }

    /* check if this is a repeat of an already parsed SPS, then keep the
     * original one.
     * otherwise drop all PPSes that depend on it */
    if (s->sps_list[sps_id] &&
        !memcmp(s->sps_list[sps_id]->data, sps_buf->data, sps_buf->size)) {
        av_buffer_unref(&sps_buf);
    } else {
        for (i = 0; i < FF_ARRAY_ELEMS(s->pps_list); i++) {
            if (s->pps_list[i] && ((HEVCPPS*)s->pps_list[i]->data)->sps_id == sps_id)
                av_buffer_unref(&s->pps_list[i]);
        }
        if (s->sps_list[sps_id] && s->sps == (HEVCSPS*)s->sps_list[sps_id]->data) {
            av_buffer_unref(&s->current_sps);
            s->current_sps = av_buffer_ref(s->sps_list[sps_id]);
            if (!s->current_sps)
                s->sps = NULL;
        }
        av_buffer_unref(&s->sps_list[sps_id]);
        s->sps_list[sps_id] = sps_buf;
    }

    return 0;

err:
    av_buffer_unref(&sps_buf);
    return ret;
}
```

解析SPS源代码并不是很有“技术含量”。只要参考ITU-T的《HEVC标准》就可以理解了，不再做过多详细的分析。

ff_hevc_decode_nal_pps()

ff_hevc_decode_nal_pps()用于解析HEVC码流中的PPS。该函数的定义位于libavcodec\hevc_ps.c，如下所示。

```cpp
//解析PPS
int ff_hevc_decode_nal_pps(HEVCContext *s)
{
    GetBitContext *gb = &s->HEVClc->gb;
    HEVCSPS      *sps = NULL;
    int pic_area_in_ctbs;
    int log2_diff_ctb_min_tb_size;
    int i, j, x, y, ctb_addr_rs, tile_id;
    int ret = 0;
    unsigned int pps_id = 0;

    AVBufferRef *pps_buf;
    HEVCPPS *pps = av_mallocz(sizeof(*pps));

    if (!pps)
        return AVERROR(ENOMEM);

    pps_buf = av_buffer_create((uint8_t *)pps, sizeof(*pps),
                               hevc_pps_free, NULL, 0);
    if (!pps_buf) {
        av_freep(&pps);
        return AVERROR(ENOMEM);
    }

    av_log(s->avctx, AV_LOG_DEBUG, "Decoding PPS\n");

    // Default values
    // 默认值
    pps->loop_filter_across_tiles_enabled_flag = 1;
    pps->num_tile_columns                      = 1;
    pps->num_tile_rows                         = 1;
```

```
    pps->uniform_spacing_flag                 = 1;
    pps->disable_dbf                          = 0;
    pps->beta_offset                          = 0;
    pps->tc_offset                            = 0;
    pps->log2_max_transform_skip_block_size    = 2;

    // Coded parameters
    //当前PPS的ID
    pps_id = get_ue_golomb_long(gb);
    if (pps_id >= MAX_PPS_COUNT) {
        av_log(s->avctx, AV_LOG_ERROR, "PPS id out of range: %d\n", pps_id);
        ret = AVERROR_INVALIDDATA;
        goto err;
    }
    //引用的SPS的ID
    pps->sps_id = get_ue_golomb_long(gb);
    if (pps->sps_id >= MAX_SPS_COUNT) {
        av_log(s->avctx, AV_LOG_ERROR, "SPS id out of range: %d\n", pps->sps_id);
        ret = AVERROR_INVALIDDATA;
        goto err;
    }
    if (!s->sps_list[pps->sps_id]) {
        av_log(s->avctx, AV_LOG_ERROR, "SPS %u does not exist.\n", pps->sps_id);
        ret = AVERROR_INVALIDDATA;
        goto err;
    }
    sps = (HEVCSPS *)s->sps_list[pps->sps_id]->data;
    //判断当前Slice是否包含依赖片
    pps->dependent_slice_segments_enabled_flag = get_bits1(gb);
    pps->output_flag_present_flag              = get_bits1(gb);
    pps->num_extra_slice_header_bits           = get_bits(gb, 3);

    pps->sign_data_hiding_flag = get_bits1(gb);
    //在CABAC中用何种方式确定上下文变量的初始值
    pps->cabac_init_present_flag = get_bits1(gb);
    //list0中参考图像数目的默认最大值
    pps->num_ref_idx_l0_default_active = get_ue_golomb_long(gb) + 1;
    //list1中参考图像数目的默认最大值
    pps->num_ref_idx_l1_default_active = get_ue_golomb_long(gb) + 1;
    //亮度分量QP的初始值
    pps->pic_init_qp_minus26 = get_se_golomb(gb);

    pps->constrained_intra_pred_flag = get_bits1(gb);
    pps->transform_skip_enabled_flag = get_bits1(gb);

    pps->cu_qp_delta_enabled_flag = get_bits1(gb);
    pps->diff_cu_qp_delta_depth   = 0;
    if (pps->cu_qp_delta_enabled_flag)
        pps->diff_cu_qp_delta_depth = get_ue_golomb_long(gb);

    if (pps->diff_cu_qp_delta_depth < 0 ||
        pps->diff_cu_qp_delta_depth > sps->log2_diff_max_min_coding_block_size) {
        av_log(s->avctx, AV_LOG_ERROR, "diff_cu_qp_delta_depth %d is invalid\n",
               pps->diff_cu_qp_delta_depth);
        ret = AVERROR_INVALIDDATA;
        goto err;
    }

    pps->cb_qp_offset = get_se_golomb(gb);
    if (pps->cb_qp_offset < -12 || pps->cb_qp_offset > 12) {
        av_log(s->avctx, AV_LOG_ERROR, "pps_cb_qp_offset out of range: %d\n",
               pps->cb_qp_offset);
        ret = AVERROR_INVALIDDATA;
        goto err;
    }
    pps->cr_qp_offset = get_se_golomb(gb);
    if (pps->cr_qp_offset < -12 || pps->cr_qp_offset > 12) {
        av_log(s->avctx, AV_LOG_ERROR, "pps_cr_qp_offset out of range: %d\n",
               pps->cr_qp_offset);
        ret = AVERROR_INVALIDDATA;
        goto err;
    }
    pps->pic_slice_level_chroma_qp_offsets_present_flag = get_bits1(gb);
    //P Slice是否使用加权预测
    pps->weighted_pred_flag   = get_bits1(gb);
    //B Slice是否使用加权预测
    pps->weighted_bipred_flag = get_bits1(gb);

    pps->transquant_bypass_enable_flag    = get_bits1(gb);
    //是否使用tile
    pps->tiles_enabled_flag               = get_bits1(gb);
    pps->entropy_coding_sync_enabled_flag = get_bits1(gb);

    if (pps->tiles_enabled_flag) {
        //Tile的列数
        pps->num_tile_columns = get_ue_golomb_long(gb) + 1;
        //Tile的行数
        pps->num_tile_rows    = get_ue_golomb_long(gb) + 1;
        if (pps->num_tile_columns == 0 ||
            pps->num_tile_columns >= sps->width) {
            av_log(s->avctx, AV_LOG_ERROR, "num_tile_columns_minus1 out of range: %d\n",
```

```
                pps->num_tile_columns - 1);
            ret = AVERROR_INVALIDDATA;
            goto err;
        }
        if (pps->num_tile_rows == 0 ||
            pps->num_tile_rows >= sps->height) {
            av_log(s->avctx, AV_LOG_ERROR, "num_tile_rows_minus1 out of range: %d\n",
                   pps->num_tile_rows - 1);
            ret = AVERROR_INVALIDDATA;
            goto err;
        }

        pps->column_width = av_malloc_array(pps->num_tile_columns, sizeof(*pps->column_width));
        pps->row_height   = av_malloc_array(pps->num_tile_rows,    sizeof(*pps->row_height));
        if (!pps->column_width || !pps->row_height) {
            ret = AVERROR(ENOMEM);
            goto err;
        }

        pps->uniform_spacing_flag = get_bits1(gb);
        if (!pps->uniform_spacing_flag) {
            uint64_t sum = 0;
            //每个Tile的宽度和高度
            for (i = 0; i < pps->num_tile_columns - 1; i++) {
                pps->column_width[i] = get_ue_golomb_long(gb) + 1;
                sum                 += pps->column_width[i];
            }
            if (sum >= sps->ctb_width) {
                av_log(s->avctx, AV_LOG_ERROR, "Invalid tile widths.\n");
                ret = AVERROR_INVALIDDATA;
                goto err;
            }
            pps->column_width[pps->num_tile_columns - 1] = sps->ctb_width - sum;

            sum = 0;
            for (i = 0; i < pps->num_tile_rows - 1; i++) {
                pps->row_height[i] = get_ue_golomb_long(gb) + 1;
                sum               += pps->row_height[i];
            }
            if (sum >= sps->ctb_height) {
                av_log(s->avctx, AV_LOG_ERROR, "Invalid tile heights.\n");
                ret = AVERROR_INVALIDDATA;
                goto err;
            }
            pps->row_height[pps->num_tile_rows - 1] = sps->ctb_height - sum;
        }
        pps->loop_filter_across_tiles_enabled_flag = get_bits1(gb);
    }

    pps->seq_loop_filter_across_slices_enabled_flag = get_bits1(gb);
    //是否存在去方块滤波的控制信息
    pps->deblocking_filter_control_present_flag = get_bits1(gb);
    if (pps->deblocking_filter_control_present_flag) {
        pps->deblocking_filter_override_enabled_flag = get_bits1(gb);
        //是否使用去方块滤波
        pps->disable_dbf                             = get_bits1(gb);
        if (!pps->disable_dbf) {
            pps->beta_offset = get_se_golomb(gb) * 2;
            pps->tc_offset = get_se_golomb(gb) * 2;
            if (pps->beta_offset/2 < -6 || pps->beta_offset/2 > 6) {
                av_log(s->avctx, AV_LOG_ERROR, "pps_beta_offset_div2 out of range: %d\n",
                       pps->beta_offset/2);
                ret = AVERROR_INVALIDDATA;
                goto err;
            }
            if (pps->tc_offset/2 < -6 || pps->tc_offset/2 > 6) {
                av_log(s->avctx, AV_LOG_ERROR, "pps_tc_offset_div2 out of range: %d\n",
                       pps->tc_offset/2);
                ret = AVERROR_INVALIDDATA;
                goto err;
            }
        }
    }

    pps->scaling_list_data_present_flag = get_bits1(gb);
    if (pps->scaling_list_data_present_flag) {
        set_default_scaling_list_data(&pps->scaling_list);
        ret = scaling_list_data(s, &pps->scaling_list, sps);
        if (ret < 0)
            goto err;
    }
    pps->lists_modification_present_flag = get_bits1(gb);
    pps->log2_parallel_merge_level       = get_ue_golomb_long(gb) + 2;
    if (pps->log2_parallel_merge_level > sps->log2_ctb_size) {
        av_log(s->avctx, AV_LOG_ERROR, "log2_parallel_merge_level_minus2 out of range: %d\n",
               pps->log2_parallel_merge_level - 2);
        ret = AVERROR_INVALIDDATA;
        goto err;
    }

    pps->slice_header_extension_present_flag = get_bits1(gb);
```

```
    if (get_bits1(gb)) { // pps_extension_present_flag
        int pps_range_extensions_flag = get_bits1(gb);
        /* int pps_extension_7bits = */ get_bits(gb, 7);
        if (sps->ptl.general_ptl.profile_idc == FF_PROFILE_HEVC_REXT && pps_range_extensions_flag) {
            if ((ret = pps_range_extensions(s, pps, sps)) < 0)
                goto err;
        }
    }

    // Inferred parameters
    pps->col_bd   = av_malloc_array(pps->num_tile_columns + 1, sizeof(*pps->col_bd));
    pps->row_bd   = av_malloc_array(pps->num_tile_rows + 1,    sizeof(*pps->row_bd));
    pps->col_idxX = av_malloc_array(sps->ctb_width,    sizeof(*pps->col_idxX));
    if (!pps->col_bd || !pps->row_bd || !pps->col_idxX) {
        ret = AVERROR(ENOMEM);
        goto err;
    }

    if (pps->uniform_spacing_flag) {
        if (!pps->column_width) {
            pps->column_width = av_malloc_array(pps->num_tile_columns, sizeof(*pps->column_width));
            pps->row_height   = av_malloc_array(pps->num_tile_rows,    sizeof(*pps->row_height));
        }
        if (!pps->column_width || !pps->row_height) {
            ret = AVERROR(ENOMEM);
            goto err;
        }

        for (i = 0; i < pps->num_tile_columns; i++) {
            pps->column_width[i] = ((i + 1) * sps->ctb_width) / pps->num_tile_columns -
                                   (i * sps->ctb_width) / pps->num_tile_columns;
        }

        for (i = 0; i < pps->num_tile_rows; i++) {
            pps->row_height[i] = ((i + 1) * sps->ctb_height) / pps->num_tile_rows -
                                 (i * sps->ctb_height) / pps->num_tile_rows;
        }
    }

    pps->col_bd[0] = 0;
    for (i = 0; i < pps->num_tile_columns; i++)
        pps->col_bd[i + 1] = pps->col_bd[i] + pps->column_width[i];

    pps->row_bd[0] = 0;
    for (i = 0; i < pps->num_tile_rows; i++)
        pps->row_bd[i + 1] = pps->row_bd[i] + pps->row_height[i];

    for (i = 0, j = 0; i < sps->ctb_width; i++) {
        if (i > pps->col_bd[j])
            j++;
        pps->col_idxX[i] = j;
    }

    /**
     * 6.5
     */
    pic_area_in_ctbs     = sps->ctb_width    * sps->ctb_height;

    pps->ctb_addr_rs_to_ts = av_malloc_array(pic_area_in_ctbs,    sizeof(*pps->ctb_addr_rs_to_ts));
    pps->ctb_addr_ts_to_rs = av_malloc_array(pic_area_in_ctbs,    sizeof(*pps->ctb_addr_ts_to_rs));
    pps->tile_id           = av_malloc_array(pic_area_in_ctbs,    sizeof(*pps->tile_id));
    pps->min_tb_addr_zs_tab = av_malloc_array((sps->tb_mask+2) * (sps->tb_mask+2), sizeof(*pps->min_tb_addr_zs_tab));
    if (!pps->ctb_addr_rs_to_ts || !pps->ctb_addr_ts_to_rs ||
        !pps->tile_id || !pps->min_tb_addr_zs_tab) {
        ret = AVERROR(ENOMEM);
        goto err;
    }

    for (ctb_addr_rs = 0; ctb_addr_rs < pic_area_in_ctbs; ctb_addr_rs++) {
        int tb_x   = ctb_addr_rs % sps->ctb_width;
        int tb_y   = ctb_addr_rs / sps->ctb_width;
        int tile_x = 0;
        int tile_y = 0;
        int val    = 0;

        for (i = 0; i < pps->num_tile_columns; i++) {
            if (tb_x < pps->col_bd[i + 1]) {
                tile_x = i;
                break;
            }
        }

        for (i = 0; i < pps->num_tile_rows; i++) {
            if (tb_y < pps->row_bd[i + 1]) {
                tile_y = i;
                break;
            }
        }

        for (i = 0; i < tile_x; i++)
            val += pps->row_height[tile_y] * pps->column_width[i];
```

```
            val += pps->row_height[tile_y] * pps->column_width[i];
        for (i = 0; i < tile_y; i++)
            val += sps->ctb_width * pps->row_height[i];

        val += (tb_y - pps->row_bd[tile_y]) * pps->column_width[tile_x] +
               tb_x - pps->col_bd[tile_x];

        pps->ctb_addr_rs_to_ts[ctb_addr_rs] = val;
        pps->ctb_addr_ts_to_rs[val]         = ctb_addr_rs;
    }

    for (j = 0, tile_id = 0; j < pps->num_tile_rows; j++)
        for (i = 0; i < pps->num_tile_columns; i++, tile_id++)
            for (y = pps->row_bd[j]; y < pps->row_bd[j + 1]; y++)
                for (x = pps->col_bd[i]; x < pps->col_bd[i + 1]; x++)
                    pps->tile_id[pps->ctb_addr_rs_to_ts[y * sps->ctb_width + x]] = tile_id;

    pps->tile_pos_rs = av_malloc_array(tile_id, sizeof(*pps->tile_pos_rs));
    if (!pps->tile_pos_rs) {
        ret = AVERROR(ENOMEM);
        goto err;
    }

    for (j = 0; j < pps->num_tile_rows; j++)
        for (i = 0; i < pps->num_tile_columns; i++)
            pps->tile_pos_rs[j * pps->num_tile_columns + i] = pps->row_bd[j] * sps->ctb_width + pps->col_bd[i];

    log2_diff_ctb_min_tb_size = sps->log2_ctb_size - sps->log2_min_tb_size;
    pps->min_tb_addr_zs = &pps->min_tb_addr_zs_tab[1*(sps->tb_mask+2)+1];
    for (y = 0; y < sps->tb_mask+2; y++) {
        pps->min_tb_addr_zs_tab[y*(sps->tb_mask+2)] = -1;
        pps->min_tb_addr_zs_tab[y]    = -1;
    }
    for (y = 0; y < sps->tb_mask+1; y++) {
        for (x = 0; x < sps->tb_mask+1; x++) {
            int tb_x        = x >> log2_diff_ctb_min_tb_size;
            int tb_y        = y >> log2_diff_ctb_min_tb_size;
            int ctb_addr_rs = sps->ctb_width * tb_y + tb_x;
            int val         = pps->ctb_addr_rs_to_ts[ctb_addr_rs] <<
                              (log2_diff_ctb_min_tb_size * 2);
            for (i = 0; i < log2_diff_ctb_min_tb_size; i++) {
                int m = 1 << i;
                val += (m & x ? m * m : 0) + (m & y ? 2 * m * m : 0);
            }
            pps->min_tb_addr_zs[y * (sps->tb_mask+2) + x] = val;
        }
    }

    if (get_bits_left(gb) < 0) {
        av_log(s->avctx, AV_LOG_ERROR,
               "Overread PPS by %d bits\n", -get_bits_left(gb));
        goto err;
    }

    av_buffer_unref(&s->pps_list[pps_id]);
    s->pps_list[pps_id] = pps_buf;

    return 0;

err:
    av_buffer_unref(&pps_buf);
    return ret;
}
```

与解析SPS类似，解析PPS源代码并不是很有“技术含量”。只要参考ITU-T的《H.264标准》就可以理解了，不再做过多详细的分析。

ff_hevc_decode_nal_sei()

ff_hevc_decode_nal_sei()用于解析HEVC码流中的SEI。该函数的定义位于libavcodec\hevc_sei.c，如下所示。

```cpp
//解析SEI
int ff_hevc_decode_nal_sei(HEVCContext *s)
{
    int ret;

    do {
        //解析SEI信息
        ret = decode_nal_sei_message(s);
        if (ret < 0)
            return(AVERROR(ENOMEM));
    } while (more_rbsp_data(&s->HEVClc->gb));
    return 1;
}
```

从源代码可以看出，ff_hevc_decode_nal_sei()在一个do while循环中调用了另外一个函数decode_nal_sei_message()解析SEI信息。

decode_nal_sei_message()

decode_nal_sei_message()用于解析SEI信息，它的定义如下。

```cpp
//解析SEI信息
static int decode_nal_sei_message(HEVCContext *s)
{
    GetBitContext *gb = &s->HEVClc->gb;

    int payload_type = 0;
    int payload_size = 0;
    int byte = 0xFF;
    av_log(s->avctx, AV_LOG_DEBUG, "Decoding SEI\n");

    while (byte == 0xFF) {
        byte          = get_bits(gb, 8);
        payload_type += byte;
    }
    byte = 0xFF;
    while (byte == 0xFF) {
        byte          = get_bits(gb, 8);
        payload_size += byte;
    }
    if (s->nal_unit_type == NAL_SEI_PREFIX) {
        if (payload_type == 256 /*&& s->decode_checksum_sei*/) {
            decode_nal_sei_decoded_picture_hash(s);
        } else if (payload_type == 45) {
            decode_nal_sei_frame_packing_arrangement(s);
        } else if (payload_type == 47) {
            decode_nal_sei_display_orientation(s);
        } else if (payload_type == 1){
            int ret = decode_pic_timing(s);
            av_log(s->avctx, AV_LOG_DEBUG, "Skipped PREFIX SEI %d\n", payload_type);
            skip_bits(gb, 8 * payload_size);
            return ret;
        } else if (payload_type == 129){
            active_parameter_sets(s);
            av_log(s->avctx, AV_LOG_DEBUG, "Skipped PREFIX SEI %d\n", payload_type);
        } else {
            av_log(s->avctx, AV_LOG_DEBUG, "Skipped PREFIX SEI %d\n", payload_type);
            skip_bits(gb, 8*payload_size);
        }
    } else { /* nal_unit_type == NAL_SEI_SUFFIX */
        if (payload_type == 132 /* && s->decode_checksum_sei */)
            decode_nal_sei_decoded_picture_hash(s);
        else {
            av_log(s->avctx, AV_LOG_DEBUG, "Skipped SUFFIX SEI %d\n", payload_type);
            skip_bits(gb, 8 * payload_size);
        }
    }
    return 1;
}
```

从源代码可以看出，decode_nal_sei_message()根据不同的payload_type调用不同的函数进行处理，例如调用decode_nal_sei_decoded_picture_hash()，decode_nal_sei_frame_packing_arrangement()，decode_nal_sei_display_orientation()等等。

雷霄骅
leixiaohua1020@126.com
http://blog.csdn.net/leixiaohua1020

文章标签： FFmpeg 解析 VPS SPS PPS

个人分类： FFMPEG

所属专栏： FFmpeg