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

2015年04月12日 00:37:12 阅读数：38368

===

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）

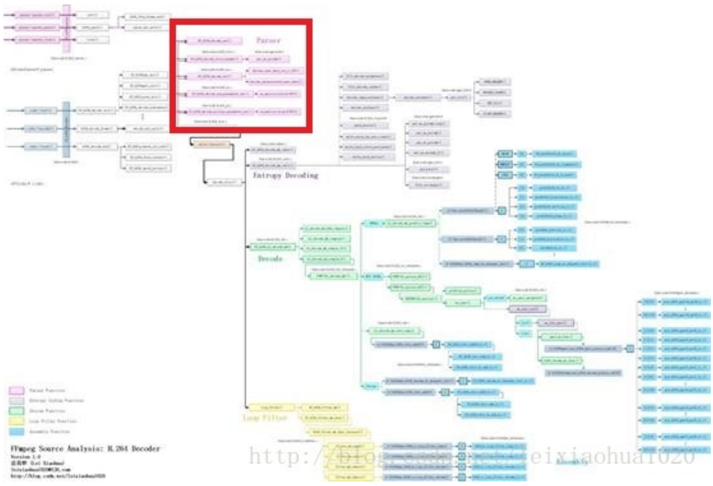
FFmpeg的H.264解码器源代码简单分析：环路滤波（Loop Filter）部分

===

本文继续分析FFmpeg中libavcodec的H.264解码器（H.264 Decoder）。上篇文章概述了FFmpeg中H.264解码器的结构；从这篇文章开始，具体研究H.264解码器的源代码。本文分析H.264解码器中解析器（Parser）部分的源代码。这部分的代码用于分割H.264的NALU，并且解析SPS、PPS、SEI等信息。解析H.264码流（对应AVCodecParser结构体中的函数）和解码H.264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。

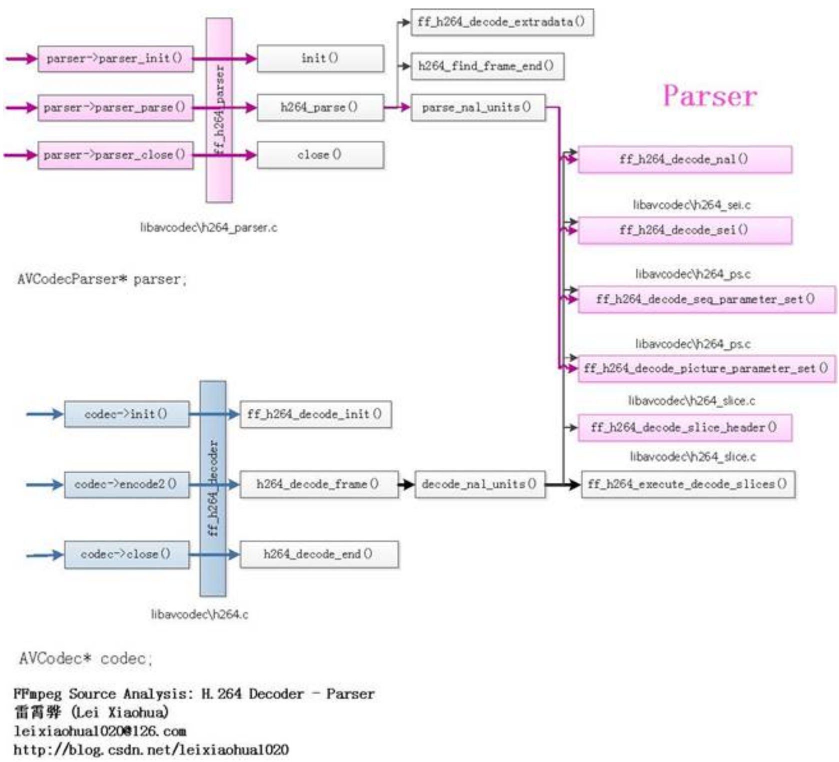
函数调用关系图

解析器（Parser）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。



单击查看更清晰的图片

从图中可以看出，H.264的解析器（Parser）在解析数据的时候调用h264_parse()，h264_parse()调用了parse_nal_units()，parse_nal_units()则调用了一系列解析特定NALU的函数。H.264的解码器（Decoder）在解码数据的时候调用h264_decode_frame()，h264_decode_frame()调用了decode_nal_units()，decode_nal_units()也同样调用了一系列解析不同NALU的函数。

图中简单列举了几个解析特定NALU的函数：

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

H.264解码器与H.264解析器最主要的不同的地方在于它调用了ff_h264_execute_decode_slices()函数进行了解码工作。这篇文章只分析H.264解析器的源代码，至于H.264解码器的源代码，则在后面几篇文章中再进行分析。

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。
有关H.264解码器这方面的源代码在以后的文章中再进行详细的分析。在这里我们只需要知道h264_decode_frame()内部调用了decode_nal_units()，而decode_nal_units()调用了和H.264解析器（Parser）有关的源代码就可以了。

ff_h264_parser

ff_h264_parser是FFmpeg的H.264解析器对应的AVCodecParser结构体。它的定义位于libavcodec\h264_parser.c，如下所示。

```cpp
AVCodecParser ff_h264_parser = {
    .codec_ids      = { AV_CODEC_ID_H264 },
    .priv_data_size = sizeof(H264Context),
    .parser_init    = init,
    .parser_parse   = h264_parse,
    .parser_close   = close,
    .split          = h264_split,
};
```

从ff_h264_parser的定义可以看出：AVCodecParser初始化的函数指针parser_init()指向init()函数；解析数据的函数指针parser_parse()指向h264_parse()函数；销毁的函数指针parser_close()指向close()函数。下面分别看看这些函数。

init() [对应于AVCodecParser-> parser_init()]

ff_h264_parser结构体中AVCodecParser的parser_init()指向init()函数。该函数完成了AVCodecParser的初始化工作。函数的定义很简单，如下所示。

```cpp
static av_cold int init(AVCodecParserContext *s)
{
    H264Context *h = s->priv_data;
    h->thread_context[0]    = h;
    h->slice_context_count = 1;
    ff_h264dsp_init(&h->h264dsp, 8, 1);
    return 0;
}
```

close() [对应于AVCodecParser-> parser_close()]

ff_h264_parser结构体中AVCodecParser的parser_close()指向close()函数。该函数完成了AVCodecParser的关闭工作。函数的定义也比较简单，如下所示。

```cpp
static void close(AVCodecParserContext *s)
{
    H264Context *h    = s->priv_data;
    ParseContext *pc = &h->parse_context;

    av_freep(&pc->buffer);
    ff_h264_free_context(h);
}
```

h264_parse() [对应于AVCodecParser-> parser_parse()]

ff_h264_parser结构体中AVCodecParser的parser_parse()指向h264_parse()函数。该函数完成了AVCodecParser的解析工作（在这里就是H.264码流的解析工作）。h264_parse()的定义位于libavcodec\h264_parser.c，如下所示。

```cpp
//解析H.264码流
//输出一个完整的NAL，存储于poutbuf中
static int h264_parse(AVCodecParserContext *s,
                      AVCodecContext *avctx,
                      const uint8_t **poutbuf, int *poutbuf_size,
                      const uint8_t *buf, int buf_size)
{
    H264Context *h   = s->priv_data;
    ParseContext *pc = &h->parse_context;
    int next;
    //如果还没有解析过1帧，就调用这里解析extradata
    if (!h->got_first) {
        h->got_first = 1;
        if (avctx->extradata_size) {
            h->avctx = avctx;
            // must be done like in decoder, otherwise opening the parser,
            // letting it create extradata and then closing and opening again
            // will cause has_b_frames to be always set.
            // Note that estimate_timings_from_pts does exactly this.
            if (!avctx->has_b_frames)
                h->low_delay = 1;
            //解析AVCodecContext的extradata
            ff_h264_decode_extradata(h, avctx->extradata, avctx->extradata_size);
        }
    }
    //输入的数据是完整的一帧？
    //这里通过设置flags的PARSER_FLAG_COMPLETE_FRAMES来确定
    if (s->flags & PARSER_FLAG_COMPLETE_FRAMES) {
        //和缓存大小一样
        next = buf_size;
    } else {
        //查找帧结尾（帧开始）位置
        //以“起始码”为依据（0x000001或0x00000001）
        next = h264_find_frame_end(h, buf, buf_size);
        //组帧
        if (ff_combine_frame(pc, next, &buf, &buf_size) < 0) {
            *poutbuf      = NULL;
            *poutbuf_size = 0;
            return buf_size;
        }

        if (next < 0 && next != END_NOT_FOUND) {
            av_assert1(pc->last_index + next >= 0);
            h264_find_frame_end(h, &pc->buffer[pc->last_index + next], -next); // update state
        }
    }
    //解析NALU，从SPS、PPS、SEI等中获得一些基本信息。
    //此时buf中存储的是完整的1帧数据
    parse_nal_units(s, avctx, buf, buf_size);

    if (avctx->framerate.num)
        avctx->time_base = av_inv_q(av_mul_q(avctx->framerate, (AVRational){avctx->ticks_per_frame, 1}));
    if (h->sei_cpb_removal_delay >= 0) {
        s->dts_sync_point    = h->sei_buffering_period_present;
        s->dts_ref_dts_delta = h->sei_cpb_removal_delay;
        s->pts_dts_delta     = h->sei_dpb_output_delay;
    } else {
        s->dts_sync_point    = INT_MIN;
        s->dts_ref_dts_delta = INT_MIN;
        s->pts_dts_delta     = INT_MIN;
    }

    if (s->flags & PARSER_FLAG_ONCE) {
        s->flags &= PARSER_FLAG_COMPLETE_FRAMES;
    }
    //分割后的帧数据输出至poutbuf
    *poutbuf      = buf;
    *poutbuf_size = buf_size;
    return next;
}
```

从源代码可以看出，h264_parse()主要完成了以下3步工作：

（1）如果是第一次解析，则首先调用ff_h264_decode_extradata()解析AVCodecContext的extradata（里面实际上存储了H.264的SPS、PPS）。

（2）如果传入的flags 中包含PARSER_FLAG_COMPLETE_FRAMES，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PARSER_FLAG_COMPLETE_FRAMES，则说明传入的不是完整的一帧数据而是任意一段H.264数据，则需要调用h264_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。

（3）调用parse_nal_units()完成了NALU的解析工作。

下面分别看一下这3步中涉及到的函数：ff_h264_decode_extradata()，h264_find_frame_end()，parse_nal_units()。

ff_h264_decode_extradata()

ff_h264_decode_extradata()用于解析AVCodecContext的extradata（里面实际上存储了H.264的SPS、PPS）。ff_h264_decode_extradata()的定义如下所示。

```cpp
//解析extradata
//最常见的就是解析AVCodecContext的extradata。其中extradata实际上存储的就是SPS、PPS
int ff_h264_decode_extradata(H264Context *h, const uint8_t *buf, int size)
{
    AVCodecContext *avctx = h->avctx;
    int ret;

    if (!buf || size <= 0)
        return -1;

    if (buf[0] == 1) {
        int i, cnt, nalsize;
        const unsigned char *p = buf;

        //AVC1 描述:H.264 bitstream without start codes.是不带起始码0×00000001的。MKV/MOV/FLV中的H.264属于这种类型
        //H264 描述:H.264 bitstream with start codes.是带有起始码0×00000001的。MPEGTS中的H.264，或者H.264裸流属于这种类型
        h->is_avc = 1;
        //数据量太小
        //随意测了一个视频
        //SPS: 30 Byte
        //PPS: 6 Byte
        if (size < 7) {
            av_log(avctx, AV_LOG_ERROR,
                   "avcC %d too short\n", size);
            return AVERROR_INVALIDDATA;
        }
        /* sps and pps in the avcC always have length coded with 2 bytes,
         * so put a fake nal_length_size = 2 while parsing them */
        h->nal_length_size = 2;
        // Decode sps from avcC
        //解码SPS
        cnt = *(p + 5) & 0x1f; // Number of sps
        p  += 6;
        for (i = 0; i < cnt; i++) {
            nalsize = AV_RB16(p) + 2;
            if(nalsize > size - (p-buf))
                return AVERROR_INVALIDDATA;
            //解析
            ret = decode_nal_units(h, p, nalsize, 1);
            if (ret < 0) {
                av_log(avctx, AV_LOG_ERROR,
                       "Decoding sps %d from avcC failed\n", i);
                return ret;
            }
            p += nalsize;
        }
        // Decode pps from avcC
        //解码PPS
        cnt = *(p++); // Number of pps
        for (i = 0; i < cnt; i++) {
            nalsize = AV_RB16(p) + 2;
            if(nalsize > size - (p-buf))
                return AVERROR_INVALIDDATA;
            ret = decode_nal_units(h, p, nalsize, 1);
            if (ret < 0) {
                av_log(avctx, AV_LOG_ERROR,
                       "Decoding pps %d from avcC failed\n", i);
                return ret;
            }
            p += nalsize;
        }
        // Store right nal length size that will be used to parse all other nals
        h->nal_length_size = (buf[4] & 0x03) + 1;
    } else {
        h->is_avc = 0;
        //解析
        ret = decode_nal_units(h, buf, size, 1);
        if (ret < 0)
            return ret;
    }
    return size;
}
```

从源代码中可以看出，ff_h264_decode_extradata()调用decode_nal_units()解析SPS、PPS信息。有关decode_nal_units()的源代码在后续文章中再进行分析。

h264_find_frame_end()

h264_find_frame_end()用于查找H.264码流中的“起始码”（start code）。在H.264码流中有两种起始码：0x000001和0x00000001。其中4Byte的长度的起始码最为常见。只有当一个完整的帧被编为多个slice的时候，包含这些slice的NALU才会使用3Byte的起始码。h264_find_frame_end()的定义位于libavcodec\h264_parser.c，如下所示。

```cpp
//查找帧结尾（帧开始）位置
```

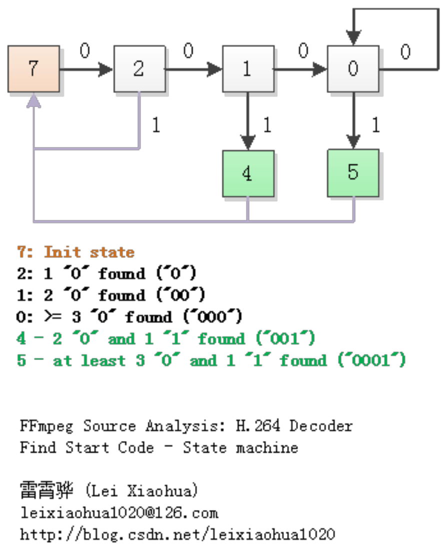
```
//
//几种状态state：
//2 - 找到1个0
//1 - 找到2个0
//0 - 找到大于等于3个0
//4 - 找到2个0和1个1，即001（即找到了起始码）
//5 - 找到至少3个0和1个1，即0001等等（即找到了起始码）
//7 - 初始化状态
//>=8 - 找到2个Slice Header
//
//关于起始码startcode的两种形式：3字节的0x000001和4字节的0x00000001
//3字节的0x000001只有一种场合下使用，就是一个完整的帧被编为多个slice的时候，
//包含这些slice的nalu使用3字节起始码。其余场合都是4字节的。
//
static int h264_find_frame_end(H264Context *h, const uint8_t *buf,
                               int buf_size)
{
    int i, j;
    uint32_t state;
    ParseContext *pc = &h->parse_context;
    int next_avc= h->is_avc ? 0 : buf_size;

//    mb_addr= pc->mb_addr - 1;
    state = pc->state;
    if (state > 13)
        state = 7;

    if (h->is_avc && !h->nal_length_size)
        av_log(h->avctx, AV_LOG_ERROR, "AVC-parser: nal length size invalid\n");
    //
    //每次循环前进1个字节，读取该字节的值
    //根据此前的状态state做不同的处理
    //state取值为4,5代表找到了起始码
    //类似于一个状态机，简单画一下状态转移图：
    //                            +-----+
    //                            |     |
    //                            v     |
    // 7--(0)-->2--(0)-->1--(0)-->0-(0)-+
    // ^        |        |        |
    // |       (1)      (1)      (1)
    // |        |        |        |
    // +--------+        v        v
    //                   4        5
    //
    for (i = 0; i < buf_size; i++) {
        //超过了
        if (i >= next_avc) {
            int nalsize = 0;
            i = next_avc;
            for (j = 0; j < h->nal_length_size; j++)
                nalsize = (nalsize << 8) | buf[i++];
            if (nalsize <= 0 || nalsize > buf_size - i) {
                av_log(h->avctx, AV_LOG_ERROR, "AVC-parser: nal size %d remaining %d\n", nalsize, buf_size - i);
                return buf_size;
            }
            next_avc = i + nalsize;
            state    = 5;
        }
        //初始state为7
        if (state == 7) {
            //查找startcode的候选者？
            //从一段内存中查找取值为0的元素的位置并返回
            //增加i取值
            i += h->h264dsp.startcode_find_candidate(buf + i, next_avc - i);
            //因为找到1个0，状态转换为2
            if (i < next_avc)
                state = 2;
        } else if (state <= 2) {       //找到0时候的state。包括1个0（状态2），2个0（状态1），或者3个及3个以上0（状态0）。
            if (buf[i] == 1)           //发现了一个1
                state ^= 5;            //状态转换关系：2->7, 1->4, 0->5。状态4代表找到了001，状态5代表找到了0001
            else if (buf[i])
                state = 7;             //恢复初始
            else                       //发现了一个0
                state >>= 1;           // 2->1, 1->0, 0->0
        } else if (state <= 5) {
            //状态4代表找到了001，状态5代表找到了0001
            //获取NALU类型
            //NALU Header（1Byte）的后5bit
            int nalu_type = buf[i] & 0x1F;

            if (nalu_type == NAL_SEI || nalu_type == NAL_SPS ||
                nalu_type == NAL_PPS || nalu_type == NAL_AUD) {
                //SPS，PPS，SEI类型的NALU
                if (pc->frame_start_found) {    //如果之前已找到了帧头
                    i++;
                    goto found;
                }
            } else if (nalu_type == NAL_SLICE || nalu_type == NAL_DPA ||
                       nalu_type == NAL_IDR_SLICE) {
                //表示有slice header的NALU
                //大于等于8的状态表示找到了两个帧头，但没有找到帧尾的状态
```

```
                state += 8;
                continue;
            }
            //上述两个条件都不满足，回归初始状态（state取值7）
            state = 7;
        } else {
            h->parse_history[h->parse_history_count++]= buf[i];
            if (h->parse_history_count>5) {
                unsigned int mb, last_mb= h->parse_last_mb;
                GetBitContext gb;

                init_get_bits(&gb, h->parse_history, 8*h->parse_history_count);
                h->parse_history_count=0;
                mb= get_ue_golomb_long(&gb);
                h->parse_last_mb= mb;
                if (pc->frame_start_found) {
                    if (mb <= last_mb)
                        goto found;
                } else
                    pc->frame_start_found = 1;
                state = 7;
            }
        }
    }
    pc->state = state;
    if (h->is_avc)
        return next_avc;
    //没找到
    return END_NOT_FOUND;

found:
    pc->state             = 7;
    pc->frame_start_found = 0;
    if (h->is_avc)
        return next_avc;
    //state=4时候，state & 5=4
    //找到的是001（长度为3），i减小3+1=4，标识帧结尾
    //state=5时候，state & 5=5
    //找到的是0001（长度为4），i减小4+1=5，标识帧结尾
    return i - (state & 5) - 5 * (state > 7);
}
```

从源代码可以看出，h264_find_frame_end()使用了一种类似于状态机的方式查找起始码。函数中的for()循环每执行一遍，状态机的状态就会改变一次。该状态机主要包含以下几种状态：

7 - 初始化状态
2 - 找到1个0
1 - 找到2个0
0 - 找到大于等于3个0
4 - 找到2个0和1个1，即001（即找到了起始码）
5 - 找到至少3个0和1个1，即0001等等（即找到了起始码）
>=8 - 找到2个Slice Header

这些状态之间的状态转移图如下所示。图中粉红色代表初始状态，绿色代表找到“起始码”的状态。



如图所示，h264_find_frame_end()初始化时候位于状态“7”；当找到1个“0”之后，状态从“7”变为“2”；在状态“2”下，如果再次找到1个“0”，则状态变为“1”；在状态“1”下，如果找到“1”，则状态变换为“4”，表明找到了“0x000001”起始码；在状态“1”下，如果找到“0”，则状态变换为“0”；在状态“0”下，如果找到“1”，则状态变换为“5”，表明找到了“0x000001”起始码。

startcode_find_candidate()

其中，在查找数据中第1个"0"的时候，使用了H264DSPContext结构体中的startcode_find_candidate()函数。startcode_find_candidate()除了包含C语言版本的函数外，还包含了ARMV6等平台下经过汇编优化的函数（估计效率会比C语言版本函数高一些）。C语言版本的函数ff_startcode_find_candidate_c()的定义很简单，位于libavcodec\startcode.c，如下所示。

[cpp]

```
int ff_startcode_find_candidate_c(const uint8_t *buf, int size)
{
    int i = 0;
    for (; i < size; i++)
        if (!buf[i])
            break;
    return i;
}
```

parse_nal_units()

parse_nal_units()用于解析NALU，从SPS、PPS、SEI等中获得一些基本信息。在该函数中，根据NALU的不同，分别调用不同的函数进行具体的处理。parse_nal_units()的定义位于libavcodec\h264_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，从SPS、PPS、SEI等中获得一些基本信息。
static inline int parse_nal_units(AVCodecParserContext *s,
                                  AVCodecContext *avctx,
                                  const uint8_t * const buf, int buf_size)
{
    H264Context *h         = s->priv_data;
    int buf_index, next_avc;
    unsigned int pps_id;
    unsigned int slice_type;
    int state = -1, got_reset = 0;
    const uint8_t *ptr;
    int q264 = buf_size >=4 && !memcmp("Q264", buf, 4);
    int field_poc[2];

    /* 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;
    ff_h264_reset_sei(h);
    h->sei_fpa.frame_packing_arrangement_cancel_flag = -1;

    if (!buf_size)
        return 0;

    buf_index     = 0;
    next_avc      = h->is_avc ? 0 : buf_size;
    for (;;) {
        int src_length, dst_length, consumed, nalsize = 0;

        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;
        }
        src_length = next_avc - buf_index;
        //NALU Header (1 Byte)
        state = buf[buf_index];
        switch (state & 0x1f) {
        case NAL_SLICE:
        case NAL_IDR_SLICE:
            // Do not walk the whole buffer just to decode slice header
            if ((state & 0x1f) == NAL_IDR_SLICE || ((state >> 5) & 0x3) == 0) {
                /* IDR or disposable slice
                 * No need to decode many bytes because MMCOs shall not be present. */
                if (src_length > 60)
```

```c
                if (src_length > 60)
                    src_length = 60;
            } else {
                /* To decode up to MMCOs */
                if (src_length > 1000)
                    src_length = 1000;
            }
            break;
        }
        //解析NAL Header，获得nal_unit_type等信息
        ptr = ff_h264_decode_nal(h, buf + buf_index, &dst_length,
                                 &consumed, src_length);
        if (!ptr || dst_length < 0)
            break;

        buf_index += consumed;
        //初始化GetBitContext
        //H264Context->gb
        //后面的解析都是从这里获取数据
        init_get_bits(&h->gb, ptr, 8 * dst_length);
        switch (h->nal_unit_type) {
        case NAL_SPS:
            //解析SPS
            ff_h264_decode_seq_parameter_set(h);
            break;
        case NAL_PPS:
            //解析PPS
            ff_h264_decode_picture_parameter_set(h, h->gb.size_in_bits);
            break;
        case NAL_SEI:
            //解析SEI
            ff_h264_decode_sei(h);
            break;
        case NAL_IDR_SLICE:
            //如果是IDR Slice
            //赋值AVCodecParserContext的key_frame为1
            s->key_frame = 1;

            h->prev_frame_num        = 0;
            h->prev_frame_num_offset = 0;
            h->prev_poc_msb          =
            h->prev_poc_lsb          = 0;
        /* fall through */
        case NAL_SLICE:
            //获取Slice的一些信息
            //跳过first_mb_in_slice这一字段
            get_ue_golomb_long(&h->gb);  // skip first_mb_in_slice
            //获取帧类型（I,B,P）
            slice_type   = get_ue_golomb_31(&h->gb);
            //赋值到AVCodecParserContext的pict_type（外部可以访问到）
            s->pict_type = golomb_to_pict_type[slice_type % 5];
            //关键帧
            if (h->sei_recovery_frame_cnt >= 0) {
                /* key frame, since recovery_frame_cnt is set */
                //赋值AVCodecParserContext的key_frame为1
                s->key_frame = 1;
            }
            //获取 PPS ID
            pps_id = get_ue_golomb(&h->gb);
            if (pps_id >= MAX_PPS_COUNT) {
                av_log(h->avctx, AV_LOG_ERROR,
                       "pps_id %u out of range\n", pps_id);
                return -1;
            }
            if (!h->pps_buffers[pps_id]) {
                av_log(h->avctx, AV_LOG_ERROR,
                       "non-existing PPS %u referenced\n", pps_id);
                return -1;
            }
            h->pps = *h->pps_buffers[pps_id];
            if (!h->sps_buffers[h->pps.sps_id]) {
                av_log(h->avctx, AV_LOG_ERROR,
                       "non-existing SPS %u referenced\n", h->pps.sps_id);
                return -1;
            }
            h->sps       = *h->sps_buffers[h->pps.sps_id];
            h->frame_num = get_bits(&h->gb, h->sps.log2_max_frame_num);

            if(h->sps.ref_frame_count <= 1 && h->pps.ref_count[0] <= 1 && s->pict_type == AV_PICTURE_TYPE_I)
                s->key_frame = 1;
            //获得“型”和“级”
            //赋值到AVCodecContext的profile和level
            avctx->profile = ff_h264_get_profile(&h->sps);
            avctx->level   = h->sps.level_idc;

            if (h->sps.frame_mbs_only_flag) {
                h->picture_structure = PICT_FRAME;
            } else {
                if (get_bits1(&h->gb)) { // field_pic_flag
                    h->picture_structure = PICT_TOP_FIELD + get_bits1(&h->gb); // bottom_field_flag
                } else {
                    h->picture_structure = PICT_FRAME;
```

```c
        }
    }

    if (h->nal_unit_type == NAL_IDR_SLICE)
        get_ue_golomb(&h->gb); /* idr_pic_id */
    if (h->sps.poc_type == 0) {
        h->poc_lsb = get_bits(&h->gb, h->sps.log2_max_poc_lsb);

        if (h->pps.pic_order_present == 1 &&
            h->picture_structure == PICT_FRAME)
            h->delta_poc_bottom = get_se_golomb(&h->gb);
    }

    if (h->sps.poc_type == 1 &&
        !h->sps.delta_pic_order_always_zero_flag) {
        h->delta_poc[0] = get_se_golomb(&h->gb);

        if (h->pps.pic_order_present == 1 &&
            h->picture_structure == PICT_FRAME)
            h->delta_poc[1] = get_se_golomb(&h->gb);
    }

    /* Decode POC of this picture.
     * The prev_ values needed for decoding POC of the next picture are not set here. */
    field_poc[0] = field_poc[1] = INT_MAX;
    ff_init_poc(h, field_poc, &s->output_picture_number);

    /* Continue parsing to check if MMCO_RESET is present.
     * FIXME: MMCO_RESET could appear in non-first slice.
     *        Maybe, we should parse all undisposable non-IDR slice of this
     *        picture until encountering MMCO_RESET in a slice of it. */
    if (h->nal_ref_idc && h->nal_unit_type != NAL_IDR_SLICE) {
        got_reset = scan_mmco_reset(s);
        if (got_reset < 0)
            return got_reset;
    }

    /* Set up the prev_ values for decoding POC of the next picture. */
    h->prev_frame_num        = got_reset ? 0 : h->frame_num;
    h->prev_frame_num_offset = got_reset ? 0 : h->frame_num_offset;
    if (h->nal_ref_idc != 0) {
        if (!got_reset) {
            h->prev_poc_msb = h->poc_msb;
            h->prev_poc_lsb = h->poc_lsb;
        } else {
            h->prev_poc_msb = 0;
            h->prev_poc_lsb =
                h->picture_structure == PICT_BOTTOM_FIELD ? 0 : field_poc[0];
        }
    }
    //包含“场”概念的时候，先不管
    if (h->sps.pic_struct_present_flag) {
        switch (h->sei_pic_struct) {
        case SEI_PIC_STRUCT_TOP_FIELD:
        case SEI_PIC_STRUCT_BOTTOM_FIELD:
            s->repeat_pict = 0;
            break;
        case SEI_PIC_STRUCT_FRAME:
        case SEI_PIC_STRUCT_TOP_BOTTOM:
        case SEI_PIC_STRUCT_BOTTOM_TOP:
            s->repeat_pict = 1;
            break;
        case SEI_PIC_STRUCT_TOP_BOTTOM_TOP:
        case SEI_PIC_STRUCT_BOTTOM_TOP_BOTTOM:
            s->repeat_pict = 2;
            break;
        case SEI_PIC_STRUCT_FRAME_DOUBLING:
            s->repeat_pict = 3;
            break;
        case SEI_PIC_STRUCT_FRAME_TRIPLING:
            s->repeat_pict = 5;
            break;
        default:
            s->repeat_pict = h->picture_structure == PICT_FRAME ? 1 : 0;
            break;
        }
    } else {
        s->repeat_pict = h->picture_structure == PICT_FRAME ? 1 : 0;
    }

    if (h->picture_structure == PICT_FRAME) {
        s->picture_structure = AV_PICTURE_STRUCTURE_FRAME;
        if (h->sps.pic_struct_present_flag) {
            switch (h->sei_pic_struct) {
            case SEI_PIC_STRUCT_TOP_BOTTOM:
            case SEI_PIC_STRUCT_TOP_BOTTOM_TOP:
                s->field_order = AV_FIELD_TT;
                break;
            case SEI_PIC_STRUCT_BOTTOM_TOP:
            case SEI_PIC_STRUCT_BOTTOM_TOP_BOTTOM:
                s->field_order = AV_FIELD_BB;
```

```
                        break;
                    default:
                        s->field_order = AV_FIELD_PROGRESSIVE;
                        break;
                    }
                } else {
                    if (field_poc[0] < field_poc[1])
                        s->field_order = AV_FIELD_TT;
                    else if (field_poc[0] > field_poc[1])
                        s->field_order = AV_FIELD_BB;
                    else
                        s->field_order = AV_FIELD_PROGRESSIVE;
                }
            } else {
                if (h->picture_structure == PICT_TOP_FIELD)
                    s->picture_structure = AV_PICTURE_STRUCTURE_TOP_FIELD;
                else
                    s->picture_structure = AV_PICTURE_STRUCTURE_BOTTOM_FIELD;
                s->field_order = AV_FIELD_UNKNOWN;
            }

            return 0; /* no need to evaluate the rest */
        }
    }
    if (q264)
        return 0;
    /* didn't find a picture! */
    av_log(h->avctx, AV_LOG_ERROR, "missing picture in access unit with size %d\n", buf_size);
    return -1;
}
```

从源代码可以看出，parse_nal_units()主要做了以下几步处理：

（1）对于所有的NALU，都调用ff_h264_decode_nal解析NALU的Header，得到nal_unit_type等信息

（2）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SPS的时候调用ff_h264_decode_seq_parameter_set()

b)解析PPS的时候调用ff_h264_decode_picture_parameter_set()

c)解析SEI的时候调用ff_h264_decode_sei()

d)解析IDR Slice / Slice的时候，获取slice_type等一些信息。

ff_h264_decode_nal()

ff_h264_decode_nal()用于解析NAL Header，获得nal_unit_type等信息。该函数的定义位于libavcodec\h264.c，如下所示。

[cpp]

```
//解析NAL Header，获得nal_unit_type等信息
const uint8_t *ff_h264_decode_nal(H264Context *h, const uint8_t *src,
                                  int *dst_length, int *consumed, int length)
{
    int i, si, di;
    uint8_t *dst;
    int bufidx;

    // src[0]&0x80; // forbidden bit
    //
    // 1 byte NALU头
    // forbidden_zero_bit: 1bit
    // nal_ref_idc: 2bit
    // nal_unit_type: 5bit
    // nal_ref_idc指示NAL的优先级，取值0-3，值越高，代表NAL越重要
    h->nal_ref_idc   = src[0] >> 5;
    // nal_unit_type指示NAL的类型
    h->nal_unit_type = src[0] & 0x1F;
    //后移1Byte
    src++;
    //未处理数据长度减1
    length--;

    //起始码:0x000001
    //保留:0x000002
    //防止竞争:0x000003
    //既表示NALU的开始，又表示NALU的结束
    //STARTCODE_TEST这个宏在后面用到
    //得到length
    //length是指当前NALU单元长度，这里不包括nalu头信息长度（即1个字节）
#define STARTCODE_TEST                                                  \
    if (i + 2 < length && src[i + 1] == 0 && src[i + 2] <= 3) {         \
        if (src[i + 2] != 3 && src[i + 2] != 0) {                       \
            /* 取值为1或者2（保留用），为起始码。startcode, so we must be past the end */\
            length = i;                                                 \
        }                                                               \
        break;                                                          \
    }

#if HAVE_FAST_UNALIGNED
#define FIND_FIRST_ZERO                                                 \
```

```
#define FIND_FIRST_ZERO                                          \
    if (i > 0 && !src[i])                                        \
        i--;                                                     \
    while (src[i])                                               \
        i++

#if HAVE_FAST_64BIT
    for (i = 0; i + 1 < length; i += 9) {
        if (!((~AV_RN64A(src + i) &
               (AV_RN64A(src + i) - 0x0100010001000101ULL)) &
              0x8000800080008080ULL))
            continue;
        FIND_FIRST_ZERO;
        STARTCODE_TEST;
        i -= 7;
    }
#else
    for (i = 0; i + 1 < length; i += 5) {
        if (!((~AV_RN32A(src + i) &
               (AV_RN32A(src + i) - 0x01000101U)) &
              0x80008080U))
            continue;
        FIND_FIRST_ZERO;
        STARTCODE_TEST;
        i -= 3;
    }
#endif
#else
    for (i = 0; i + 1 < length; i += 2) {
        if (src[i])
            continue;
        if (i > 0 && src[i - 1] == 0)
            i--;
        //起始码检测
        STARTCODE_TEST;
    }
#endif

    // use second escape buffer for inter data
    bufidx = h->nal_unit_type == NAL_DPC ? 1 : 0;

    av_fast_padded_malloc(&h->rbsp_buffer[bufidx], &h->rbsp_buffer_size[bufidx], length+MAX_MBPAIR_SIZE);
    dst = h->rbsp_buffer[bufidx];

    if (!dst)
        return NULL;

    if(i>=length-1){ //no escaped 0
        *dst_length= length;
        *consumed= length+1; //+1 for the header
        if(h->avctx->flags2 & CODEC_FLAG2_FAST){
            return src;
        }else{
            memcpy(dst, src, length);
            return dst;
        }
    }

    memcpy(dst, src, i);
    si = di = i;
    while (si + 2 < length) {
        // remove escapes (very rare 1:2^22)
        if (src[si + 2] > 3) {
            dst[di++] = src[si++];
            dst[di++] = src[si++];
        } else if (src[si] == 0 && src[si + 1] == 0 && src[si + 2] != 0) {
            if (src[si + 2] == 3) { // escape
                dst[di++]  = 0;
                dst[di++]  = 0;
                si        += 3;
                continue;
            } else // next start code
                goto nsc;
        }

        dst[di++] = src[si++];
    }
    while (si < length)
        dst[di++] = src[si++];

nsc:
    memset(dst + di, 0, FF_INPUT_BUFFER_PADDING_SIZE);

    *dst_length = di;
    *consumed   = si + 1; // +1 for the header
    /* FIXME store exact number of bits in the getbitcontext
     * (it is needed for decoding) */
    return dst;
}
```

从源代码可以看出，ff_h264_decode_nal()首先从NALU Header（NALU第1个字节）中解析出了nal_ref_idc，nal_unit_type字段的值。然后函数进入了一个for()循环进行起始码检测。

起始码检测这里稍微有点复杂，其中包含了一个STARTCODE_TEST的宏。这个宏用于做具体的起始码的判断。这部分的代码还没有细看，以后有时间再进行补充。

ff_h264_decode_seq_parameter_set()

ff_h264_decode_seq_parameter_set()用于解析H.264码流中的SPS。该函数的定义位于libavcodec\h264_ps.c，如下所示。

```cpp
//解码SPS
int ff_h264_decode_seq_parameter_set(H264Context *h)
{
    int profile_idc, level_idc, constraint_set_flags = 0;
    unsigned int sps_id;
    int i, log2_max_frame_num_minus4;

    SPS *sps;
    //profile型，8bit
    //注意get_bits()
    profile_idc           = get_bits(&h->gb, 8);
    constraint_set_flags |= get_bits1(&h->gb) << 0;   // constraint_set0_flag
    constraint_set_flags |= get_bits1(&h->gb) << 1;   // constraint_set1_flag
    constraint_set_flags |= get_bits1(&h->gb) << 2;   // constraint_set2_flag
    constraint_set_flags |= get_bits1(&h->gb) << 3;   // constraint_set3_flag
    constraint_set_flags |= get_bits1(&h->gb) << 4;   // constraint_set4_flag
    constraint_set_flags |= get_bits1(&h->gb) << 5;   // constraint_set5_flag
    skip_bits(&h->gb, 2);                             // reserved_zero_2bits
    //level级，8bit
    level_idc = get_bits(&h->gb, 8);
    //该SPS的ID号，该ID号将被picture引用
    //注意：get_ue_golomb()
    sps_id    = get_ue_golomb_31(&h->gb);

    if (sps_id >= MAX_SPS_COUNT) {
        av_log(h->avctx, AV_LOG_ERROR, "sps_id %u out of range\n", sps_id);
        return AVERROR_INVALIDDATA;
    }
    //赋值给这个结构体
    sps = av_mallocz(sizeof(SPS));
    if (!sps)
        return AVERROR(ENOMEM);
    //赋值
    sps->sps_id               = sps_id;
    sps->time_offset_length   = 24;
    sps->profile_idc          = profile_idc;
    sps->constraint_set_flags = constraint_set_flags;
    sps->level_idc            = level_idc;
    sps->full_range           = -1;

    memset(sps->scaling_matrix4, 16, sizeof(sps->scaling_matrix4));
    memset(sps->scaling_matrix8, 16, sizeof(sps->scaling_matrix8));
    sps->scaling_matrix_present = 0;
    sps->colorspace = 2; //AVCOL_SPC_UNSPECIFIED
    //Profile对应关系
    if (sps->profile_idc == 100 ||  // High profile
        sps->profile_idc == 110 ||  // High10 profile
        sps->profile_idc == 122 ||  // High422 profile
        sps->profile_idc == 244 ||  // High444 Predictive profile
        sps->profile_idc ==  44 ||  // Cavlc444 profile
        sps->profile_idc ==  83 ||  // Scalable Constrained High profile (SVC)
        sps->profile_idc ==  86 ||  // Scalable High Intra profile (SVC)
        sps->profile_idc == 118 ||  // Stereo High profile (MVC)
        sps->profile_idc == 128 ||  // Multiview High profile (MVC)
        sps->profile_idc == 138 ||  // Multiview Depth High profile (MVCD)
        sps->profile_idc == 144) {  // old High444 profile

        //色度取样
        //0代表单色
        //1代表4:2:0
        //2代表4:2:2
        //3代表4:4:4
        sps->chroma_format_idc = get_ue_golomb_31(&h->gb);
        if (sps->chroma_format_idc > 3U) {
            avpriv_request_sample(h->avctx, "chroma_format_idc %u",
                                  sps->chroma_format_idc);
            goto fail;
        } else if (sps->chroma_format_idc == 3) {
            sps->residual_color_transform_flag = get_bits1(&h->gb);
            if (sps->residual_color_transform_flag) {
                av_log(h->avctx, AV_LOG_ERROR, "separate color planes are not supported\n");
                goto fail;
            }
        }
        //bit_depth_luma_minus8
```

```
        //bit_depth_luma_minus8
        //加8之后为亮度颜色深度
        //该值取值范围应该在0到4之间。即颜色深度支持0-12bit
        sps->bit_depth_luma   = get_ue_golomb(&h->gb) + 8;
        //加8之后为色度颜色深度
        sps->bit_depth_chroma = get_ue_golomb(&h->gb) + 8;
        if (sps->bit_depth_chroma != sps->bit_depth_luma) {
            avpriv_request_sample(h->avctx,
                                  "Different chroma and luma bit depth");
            goto fail;
        }
        if (sps->bit_depth_luma > 14U || sps->bit_depth_chroma > 14U) {
            av_log(h->avctx, AV_LOG_ERROR, "illegal bit depth value (%d, %d)\n",
                   sps->bit_depth_luma, sps->bit_depth_chroma);
            goto fail;
        }
        sps->transform_bypass = get_bits1(&h->gb);
        decode_scaling_matrices(h, sps, NULL, 1,
                                sps->scaling_matrix4, sps->scaling_matrix8);
    } else {
        //默认
        sps->chroma_format_idc = 1;
        sps->bit_depth_luma    = 8;
        sps->bit_depth_chroma  = 8;
    }
    //log2_max_frame_num_minus4为另一个句法元素frame_num服务
    //fram_num的解码函数是ue（v），函数中的v 在这里指定：
    //     v = log2_max_frame_num_minus4 + 4
    //从另一个角度看，这个句法元素同时也指明了frame_num 的所能达到的最大值：
    //     MaxFrameNum = 2^( log2_max_frame_num_minus4 + 4 )
    log2_max_frame_num_minus4 = get_ue_golomb(&h->gb);
    if (log2_max_frame_num_minus4 < MIN_LOG2_MAX_FRAME_NUM - 4 ||
        log2_max_frame_num_minus4 > MAX_LOG2_MAX_FRAME_NUM - 4) {
        av_log(h->avctx, AV_LOG_ERROR,
               "log2_max_frame_num_minus4 out of range (0-12): %d\n",
               log2_max_frame_num_minus4);
        goto fail;
    }
    sps->log2_max_frame_num = log2_max_frame_num_minus4 + 4;
    //pic_order_cnt_type 指明了poc (picture order count) 的编码方法
    //poc标识图像的播放顺序。
    //由于H.264使用了B帧预测，使得图像的解码顺序并不一定等于播放顺序，但它们之间存在一定的映射关系
    //poc 可以由frame-num 通过映射关系计算得来，也可以索性由编码器显式地传送。
    //H.264 中一共定义了三种poc 的编码方法
    sps->poc_type = get_ue_golomb_31(&h->gb);
    //3种poc的编码方法
    if (sps->poc_type == 0) { // FIXME #define
        unsigned t = get_ue_golomb(&h->gb);
        if (t>12) {
            av_log(h->avctx, AV_LOG_ERROR, "log2_max_poc_lsb (%d) is out of range\n", t);
            goto fail;
        }
        sps->log2_max_poc_lsb = t + 4;
    } else if (sps->poc_type == 1) { // FIXME #define
        sps->delta_pic_order_always_zero_flag = get_bits1(&h->gb);
        sps->offset_for_non_ref_pic           = get_se_golomb(&h->gb);
        sps->offset_for_top_to_bottom_field   = get_se_golomb(&h->gb);
        sps->poc_cycle_length                 = get_ue_golomb(&h->gb);

        if ((unsigned)sps->poc_cycle_length >=
            FF_ARRAY_ELEMS(sps->offset_for_ref_frame)) {
            av_log(h->avctx, AV_LOG_ERROR,
                   "poc_cycle_length overflow %d\n", sps->poc_cycle_length);
            goto fail;
        }

        for (i = 0; i < sps->poc_cycle_length; i++)
            sps->offset_for_ref_frame[i] = get_se_golomb(&h->gb);
    } else if (sps->poc_type != 2) {
        av_log(h->avctx, AV_LOG_ERROR, "illegal POC type %d\n", sps->poc_type);
        goto fail;
    }
    //num_ref_frames 指定参考帧队列可能达到的最大长度，解码器依照这个句法元素的值开辟存储区，这个存储区用于存放已解码的参考帧，
    //H.264 规定最多可用16 个参考帧，因此最大值为16。
    sps->ref_frame_count = get_ue_golomb_31(&h->gb);
    if (h->avctx->codec_tag == MKTAG('S', 'M', 'V', '2'))
        sps->ref_frame_count = FFMAX(2, sps->ref_frame_count);
    if (sps->ref_frame_count > H264_MAX_PICTURE_COUNT - 2 ||
        sps->ref_frame_count > 16U) {
        av_log(h->avctx, AV_LOG_ERROR,
               "too many reference frames %d\n", sps->ref_frame_count);
        goto fail;
    }
    sps->gaps_in_frame_num_allowed_flag = get_bits1(&h->gb);
    //加1后为图像宽（以宏块为单位）
    //以像素为单位图像宽度（亮度）：width=mb_width*16
    sps->mb_width                       = get_ue_golomb(&h->gb) + 1;
    //加1后为图像高（以宏块为单位）
    //以像素为单位图像高度（亮度）：height=mb_height*16
    sps->mb_height                        = get_ue_golomb(&h->gb) + 1;
    //检查一下
    if ((unsigned)sps->mb_width  >= INT_MAX / 16 ||
```

```
        (unsigned)sps->mb_height >= INT_MAX / 16 ||
        av_image_check_size(16 * sps->mb_width,
                            16 * sps->mb_height, 0, h->avctx)) {
        av_log(h->avctx, AV_LOG_ERROR, "mb_width/height overflow\n");
        goto fail;
    }

    sps->frame_mbs_only_flag = get_bits1(&h->gb);
    if (!sps->frame_mbs_only_flag)
        sps->mb_aff = get_bits1(&h->gb);
    else
        sps->mb_aff = 0;

    sps->direct_8x8_inference_flag = get_bits1(&h->gb);

#ifndef ALLOW_INTERLACE
    if (sps->mb_aff)
        av_log(h->avctx, AV_LOG_ERROR,
               "MBAFF support not included; enable it at compile-time.\n");
#endif
    //裁剪输出，没研究过
    sps->crop = get_bits1(&h->gb);
    if (sps->crop) {
        int crop_left   = get_ue_golomb(&h->gb);
        int crop_right  = get_ue_golomb(&h->gb);
        int crop_top    = get_ue_golomb(&h->gb);
        int crop_bottom = get_ue_golomb(&h->gb);
        int width  = 16 * sps->mb_width;
        int height = 16 * sps->mb_height * (2 - sps->frame_mbs_only_flag);

        if (h->avctx->flags2 & CODEC_FLAG2_IGNORE_CROP) {
            av_log(h->avctx, AV_LOG_DEBUG, "discarding sps cropping, original "
                                           "values are l:%d r:%d t:%d b:%d\n",
                   crop_left, crop_right, crop_top, crop_bottom);

            sps->crop_left   =
            sps->crop_right  =
            sps->crop_top    =
            sps->crop_bottom = 0;
        } else {
            int vsub   = (sps->chroma_format_idc == 1) ? 1 : 0;
            int hsub   = (sps->chroma_format_idc == 1 ||
                          sps->chroma_format_idc == 2) ? 1 : 0;
            int step_x = 1 << hsub;
            int step_y = (2 - sps->frame_mbs_only_flag) << vsub;

            if (crop_left & (0x1F >> (sps->bit_depth_luma > 8)) &&
                !(h->avctx->flags & CODEC_FLAG_UNALIGNED)) {
                crop_left &= ~(0x1F >> (sps->bit_depth_luma > 8));
                av_log(h->avctx, AV_LOG_WARNING,
                       "Reducing left cropping to %d "
                       "chroma samples to preserve alignment.\n",
                       crop_left);
            }

            if (crop_left  > (unsigned)INT_MAX / 4 / step_x ||
                crop_right > (unsigned)INT_MAX / 4 / step_x ||
                crop_top   > (unsigned)INT_MAX / 4 / step_y ||
                crop_bottom> (unsigned)INT_MAX / 4 / step_y ||
                (crop_left + crop_right ) * step_x >= width ||
                (crop_top  + crop_bottom) * step_y >= height
            ) {
                av_log(h->avctx, AV_LOG_ERROR, "crop values invalid %d %d %d %d / %d %d\n", crop_left, crop_right, crop_top, crop_bottom, width, height);
                goto fail;
            }

            sps->crop_left   = crop_left   * step_x;
            sps->crop_right  = crop_right  * step_x;
            sps->crop_top    = crop_top    * step_y;
            sps->crop_bottom = crop_bottom * step_y;
        }
    } else {
        sps->crop_left   =
        sps->crop_right  =
        sps->crop_top    =
        sps->crop_bottom =
        sps->crop        = 0;
    }

    sps->vui_parameters_present_flag = get_bits1(&h->gb);
    if (sps->vui_parameters_present_flag) {
        int ret = decode_vui_parameters(h, sps);
        if (ret < 0)
            goto fail;
    }

    if (!sps->sar.den)
        sps->sar.den = 1;
    //Debug的时候可以输出一些信息
    if (h->avctx->debug & FF_DEBUG_PICT_INFO) {
```

```
        static const char csp[4][5] = { "Gray", "420", "422", "444" };
        av_log(h->avctx, AV_LOG_DEBUG,
               "sps:%u profile:%d/%d poc:%d ref:%d %dx%d %s %s crop:%u/%u/%u/%u %s %s %"PRId32"/%"PRId32" b%d reo:%d\n",
               sps_id, sps->profile_idc, sps->level_idc,
               sps->poc_type,
               sps->ref_frame_count,
               sps->mb_width, sps->mb_height,
               sps->frame_mbs_only_flag ? "FRM" : (sps->mb_aff ? "MB-AFF" : "PIC-AFF"),
               sps->direct_8x8_inference_flag ? "8B8" : "",
               sps->crop_left, sps->crop_right,
               sps->crop_top, sps->crop_bottom,
               sps->vui_parameters_present_flag ? "VUI" : "",
               csp[sps->chroma_format_idc],
               sps->timing_info_present_flag ? sps->num_units_in_tick : 0,
               sps->timing_info_present_flag ? sps->time_scale : 0,
               sps->bit_depth_luma,
               sps->bitstream_restriction_flag ? sps->num_reorder_frames : -1
               );
    }
    sps->new = 1;

    av_free(h->sps_buffers[sps_id]);
    h->sps_buffers[sps_id] = sps;

    return 0;

fail:
    av_free(sps);
    return -1;
}
```

解析SPS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_h264_decode_picture_parameter_set()

ff_h264_decode_picture_parameter_set()用于解析H.264码流中的PPS。该函数的定义位于libavcodec\h264_ps.c，如下所示。

[cpp]

```cpp
//解码PPS
int ff_h264_decode_picture_parameter_set(H264Context *h, int bit_length)
{
    //获取PPS ID
    unsigned int pps_id = get_ue_golomb(&h->gb);
    PPS *pps;
    SPS *sps;
    int qp_bd_offset;
    int bits_left;

    if (pps_id >= MAX_PPS_COUNT) {
        av_log(h->avctx, AV_LOG_ERROR, "pps_id %u out of range\n", pps_id);
        return AVERROR_INVALIDDATA;
    }
    //解析后赋值给PPS这个结构体
    pps = av_mallocz(sizeof(PPS));
    if (!pps)
        return AVERROR(ENOMEM);
    //该PPS引用的SPS的ID
    pps->sps_id = get_ue_golomb_31(&h->gb);
    if ((unsigned)pps->sps_id >= MAX_SPS_COUNT ||
        !h->sps_buffers[pps->sps_id]) {
        av_log(h->avctx, AV_LOG_ERROR, "sps_id %u out of range\n", pps->sps_id);
        goto fail;
    }
    sps = h->sps_buffers[pps->sps_id];
    qp_bd_offset = 6 * (sps->bit_depth_luma - 8);
    if (sps->bit_depth_luma > 14) {
        av_log(h->avctx, AV_LOG_ERROR,
               "Invalid luma bit depth=%d\n",
               sps->bit_depth_luma);
        goto fail;
    } else if (sps->bit_depth_luma == 11 || sps->bit_depth_luma == 13) {
        av_log(h->avctx, AV_LOG_ERROR,
               "Unimplemented luma bit depth=%d\n",
               sps->bit_depth_luma);
        goto fail;
    }
    //entropy_coding_mode_flag
    //0表示熵编码使用CAVLC，1表示熵编码使用CABAC
    pps->cabac             = get_bits1(&h->gb);
    pps->pic_order_present = get_bits1(&h->gb);
    pps->slice_group_count = get_ue_golomb(&h->gb) + 1;
    if (pps->slice_group_count > 1) {
        pps->mb_slice_group_map_type = get_ue_golomb(&h->gb);
        av_log(h->avctx, AV_LOG_ERROR, "FMO not supported\n");
        switch (pps->mb_slice_group_map_type) {
        case 0:
#if 0
```

```
    |       for (i = 0; i <= num_slice_groups_minus1; i++)  |    |       |
    |           run_length[i]                               |1   |ue(v) |
#endif
            break;
        case 2:
#if 0
    |       for (i = 0; i < num_slice_groups_minus1; i++) { |    |       |
    |           top_left_mb[i]                              |1   |ue(v) |
    |           bottom_right_mb[i]                          |1   |ue(v) |
    |       }                                               |    |       |
#endif
            break;
        case 3:
        case 4:
        case 5:
#if 0
    |       slice_group_change_direction_flag               |1   |u(1)  |
    |       slice_group_change_rate_minus1                  |1   |ue(v) |
#endif
            break;
        case 6:
#if 0
    |       slice_group_id_cnt_minus1                       |1   |ue(v) |
    |       for (i = 0; i <= slice_group_id_cnt_minus1; i++)|    |       |
    |           slice_group_id[i]                           |1   |u(v)  |
#endif
            break;
        }
    }
    //num_ref_idx_l0_active_minus1 加1后指明目前参考帧队列的长度，即有多少个参考帧
    //读者可能还记得在SPS中有句法元素num_ref_frames 也是跟参考帧队列有关，它们的区
    //别是num_ref_frames 指明参考帧队列的最大值， 解码器用它的值来分配内存空间；
    //num_ref_idx_l0_active_minus1 指明在这个队列中当前实际的、已存在的参考帧数目，这从它的名字
    //“active”中也可以看出来。
    pps->ref_count[0] = get_ue_golomb(&h->gb) + 1;
    pps->ref_count[1] = get_ue_golomb(&h->gb) + 1;
    if (pps->ref_count[0] - 1 > 32 - 1 || pps->ref_count[1] - 1 > 32 - 1) {
        av_log(h->avctx, AV_LOG_ERROR, "reference overflow (pps)\n");
        goto fail;
    }
    //P Slice 是否使用加权预测？
    pps->weighted_pred                        = get_bits1(&h->gb);
    //B Slice 是否使用加权预测？
    pps->weighted_bipred_idc                  = get_bits(&h->gb, 2);
    //QP初始值。读取后需要加26
    pps->init_qp                              = get_se_golomb(&h->gb) + 26 + qp_bd_offset;
    //SP和SI的QP初始值（没怎么见过这两种帧）
    pps->init_qs                              = get_se_golomb(&h->gb) + 26 + qp_bd_offset;
    pps->chroma_qp_index_offset[0]            = get_se_golomb(&h->gb);
    pps->deblocking_filter_parameters_present = get_bits1(&h->gb);
    pps->constrained_intra_pred               = get_bits1(&h->gb);
    pps->redundant_pic_cnt_present            = get_bits1(&h->gb);

    pps->transform_8x8_mode = 0;
    // contents of sps/pps can change even if id doesn't, so reinit
    h->dequant_coeff_pps = -1;
    memcpy(pps->scaling_matrix4, h->sps_buffers[pps->sps_id]->scaling_matrix4,
           sizeof(pps->scaling_matrix4));
    memcpy(pps->scaling_matrix8, h->sps_buffers[pps->sps_id]->scaling_matrix8,
           sizeof(pps->scaling_matrix8));

    bits_left = bit_length - get_bits_count(&h->gb);
    if (bits_left > 0 && more_rbsp_data_in_pps(h, pps)) {
        pps->transform_8x8_mode = get_bits1(&h->gb);
        decode_scaling_matrices(h, h->sps_buffers[pps->sps_id], pps, 0,
                                pps->scaling_matrix4, pps->scaling_matrix8);
        // second_chroma_qp_index_offset
        pps->chroma_qp_index_offset[1] = get_se_golomb(&h->gb);
    } else {
        pps->chroma_qp_index_offset[1] = pps->chroma_qp_index_offset[0];
    }

    build_qp_table(pps, 0, pps->chroma_qp_index_offset[0], sps->bit_depth_luma);
    build_qp_table(pps, 1, pps->chroma_qp_index_offset[1], sps->bit_depth_luma);
    if (pps->chroma_qp_index_offset[0] != pps->chroma_qp_index_offset[1])
        pps->chroma_qp_diff = 1;

    if (h->avctx->debug & FF_DEBUG_PICT_INFO) {
        av_log(h->avctx, AV_LOG_DEBUG,
               "pps:%u sps:%u %s slice_groups:%d ref:%u/%u %s qp:%d/%d/%d/%d %s %s %s %s\n",
               pps_id, pps->sps_id,
               pps->cabac ? "CABAC" : "CAVLC",
               pps->slice_group_count,
               pps->ref_count[0], pps->ref_count[1],
               pps->weighted_pred ? "weighted" : "",
               pps->init_qp, pps->init_qs, pps->chroma_qp_index_offset[0], pps->chroma_qp_index_offset[1],
               pps->deblocking_filter_parameters_present ? "LPAR" : "",
               pps->constrained_intra_pred ? "CONSTR" : "",
               pps->redundant_pic_cnt_present ? "REDU" : "",
               pps->transform_8x8_mode ? "8x8DCT" : "");
    }
```

```
    av_free(h->pps_buffers[pps_id]);
    h->pps_buffers[pps_id] = pps;
    return 0;

fail:
    av_free(pps);
    return -1;
}
```

和解析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，不再做过多详细的分析。

ff_h264_decode_sei()

ff_h264_decode_sei()用于解析H.264码流中的SEI。该函数的定义位于libavcodec\h264_sei.c，如下所示。

[cpp]

```cpp
//SEI补充增强信息单元
int ff_h264_decode_sei(H264Context *h)
{
    while (get_bits_left(&h->gb) > 16 && show_bits(&h->gb, 16)) {
        int type = 0;
        unsigned size = 0;
        unsigned next;
        int ret  = 0;

        do {
            if (get_bits_left(&h->gb) < 8)
                return AVERROR_INVALIDDATA;
            type += show_bits(&h->gb, 8);
        } while (get_bits(&h->gb, 8) == 255);

        do {
            if (get_bits_left(&h->gb) < 8)
                return AVERROR_INVALIDDATA;
            size += show_bits(&h->gb, 8);
        } while (get_bits(&h->gb, 8) == 255);

        if (h->avctx->debug&FF_DEBUG_STARTCODE)
            av_log(h->avctx, AV_LOG_DEBUG, "SEI %d len:%d\n", type, size);

        if (size > get_bits_left(&h->gb) / 8) {
            av_log(h->avctx, AV_LOG_ERROR, "SEI type %d size %d truncated at %d\n",
                   type, 8*size, get_bits_left(&h->gb));
            return AVERROR_INVALIDDATA;
        }
        next = get_bits_count(&h->gb) + 8 * size;

        switch (type) {
        case SEI_TYPE_PIC_TIMING: // Picture timing SEI
            ret = decode_picture_timing(h);
            if (ret < 0)
                return ret;
            break;
        case SEI_TYPE_USER_DATA_ITU_T_T35:
            if (decode_user_data_itu_t_t35(h, size) < 0)
                return -1;
            break;
            //x264的编码参数信息一般都会存储在USER_DATA_UNREGISTERED
            //其他种类的SEI见得很少
        case SEI_TYPE_USER_DATA_UNREGISTERED:
            ret = decode_unregistered_user_data(h, size);
            if (ret < 0)
                return ret;
            break;
        case SEI_TYPE_RECOVERY_POINT:
            ret = decode_recovery_point(h);
            if (ret < 0)
                return ret;
            break;
        case SEI_TYPE_BUFFERING_PERIOD:
            ret = decode_buffering_period(h);
            if (ret < 0)
                return ret;
            break;
        case SEI_TYPE_FRAME_PACKING:
            ret = decode_frame_packing_arrangement(h);
            if (ret < 0)
                return ret;
            break;
        case SEI_TYPE_DISPLAY_ORIENTATION:
            ret = decode_display_orientation(h);
            if (ret < 0)
                return ret;
            break;
        default:
            av_log(h->avctx, AV_LOG_DEBUG, "unknown SEI type %d\n", type);
        }
        skip_bits_long(&h->gb, next - get_bits_count(&h->gb));

        // FIXME check bits here
        align_get_bits(&h->gb);
    }

    return 0;
}
```

在《H.264官方标准》中，SEI的种类是非常多的。在ff_h264_decode_sei()中包含以下种类的SEI：

SEI_TYPE_BUFFERING_PERIOD

SEI_TYPE_PIC_TIMING

SEI_TYPE_USER_DATA_ITU_T_T35

SEI_TYPE_USER_DATA_UNREGISTERED

SEI_TYPE_RECOVERY_POINT

SEI_TYPE_FRAME_PACKING

SEI_TYPE_DISPLAY_ORIENTATION

其中的大部分种类的SEI信息我并没有接触过。唯一接触比较多的就是SEI_TYPE_USER_DATA_UNREGISTERED类型的信息了。使用X264编码视频的时候，会自动将配置信息以SEI_TYPE_USER_DATA_UNREGISTERED（用户数据未注册SEI）的形式写入码流。

从ff_h264_decode_sei()的定义可以看出，该函数根据不同的SEI类型调用不同的解析函数。当SEI类型为SEI_TYPE_USER_DATA_UNREGISTERED的时候，就会调用decode_unregistered_user_data()函数。

decode_unregistered_user_data()

decode_unregistered_user_data()的定义如下所示。从代码可以看出该函数只是简单的提取了X264的版本信息。

[cpp]

```cpp
//x264的编码参数信息一般都会存储在USER_DATA_UNREGISTERED
static int decode_unregistered_user_data(H264Context *h, int size)
{
    uint8_t user_data[16 + 256];
    int e, build, i;

    if (size < 16)
        return AVERROR_INVALIDDATA;

    for (i = 0; i < sizeof(user_data) - 1 && i < size; i++)
        user_data[i] = get_bits(&h->gb, 8);
    //user_data内容示例：x264 core 118
    //int sscanf(const char *buffer,const char *format,[argument ]...);
    //sscanf会从buffer里读进数据，依照format的格式将数据写入到argument里。
    user_data[i] = 0;
    e = sscanf(user_data + 16, "x264 - core %d", &build);
    if (e == 1 && build > 0)
        h->x264_build = build;
    if (e == 1 && build == 1 && !strncmp(user_data+16, "x264 - core 0000", 16))
        h->x264_build = 67;

    if (h->avctx->debug & FF_DEBUG_BUGS)
        av_log(h->avctx, AV_LOG_DEBUG, "user data:\"%s\"\n", user_data + 16);

    for (; i < size; i++)
        skip_bits(&h->gb, 8);

    return 0;
}
```

解析Slice Header

对于包含图像压缩编码的Slice，解析器（Parser）并不进行解码处理，而是简单提取一些Slice Header中的信息。该部分的代码并没有写成一个函数，而是直接写到了parse_nal_units()里面，截取出来如下所示。

[cpp]

```cpp
case NAL_IDR_SLICE:
            //如果是IDR Slice
            //赋值AVCodecParserContext的key_frame为1
            s->key_frame = 1;

            h->prev_frame_num        = 0;
            h->prev_frame_num_offset = 0;
            h->prev_poc_msb          =
            h->prev_poc_lsb          = 0;
        /* fall through */
        case NAL_SLICE:
            //获取Slice的一些信息
            //跳过first_mb_in_slice这一字段
            get_ue_golomb_long(&h->gb);  // skip first_mb_in_slice
            //获取帧类型（I,B,P）
            slice_type   = get_ue_golomb_31(&h->gb);
            //赋值到AVCodecParserContext的pict_type（外部可以访问到）
            s->pict_type = golomb_to_pict_type[slice_type % 5];
            //关键帧
            if (h->sei_recovery_frame_cnt >= 0) {
                /* key frame, since recovery_frame_cnt is set */
                //赋值AVCodecParserContext的key_frame为1
                s->key_frame = 1;
            }
            //获取 PPS ID
            pps_id = get_ue_golomb(&h->gb);
            if (pps_id >= MAX_PPS_COUNT) {
                av_log(h->avctx, AV_LOG_ERROR,
                       "pps_id %u out of range\n", pps_id);
                return -1;
            }
            if (!h->pps_buffers[pps_id]) {
                av_log(h->avctx, AV_LOG_ERROR,
                       "non-existing PPS %u referenced\n", pps_id);
```

```
                   "non-existing PPS %u referenced\n", pps_id);
            return -1;
        }
        h->pps = *h->pps_buffers[pps_id];
        if (!h->sps_buffers[h->pps.sps_id]) {
            av_log(h->avctx, AV_LOG_ERROR,
                   "non-existing SPS %u referenced\n", h->pps.sps_id);
            return -1;
        }
        h->sps       = *h->sps_buffers[h->pps.sps_id];
        h->frame_num = get_bits(&h->gb, h->sps.log2_max_frame_num);

        if(h->sps.ref_frame_count <= 1 && h->pps.ref_count[0] <= 1 && s->pict_type == AV_PICTURE_TYPE_I)
            s->key_frame = 1;
        //获得“型”和“级”
        //赋值到AVCodecContext的profile和level
        avctx->profile = ff_h264_get_profile(&h->sps);
        avctx->level   = h->sps.level_idc;

        if (h->sps.frame_mbs_only_flag) {
            h->picture_structure = PICT_FRAME;
        } else {
            if (get_bits1(&h->gb)) { // field_pic_flag
                h->picture_structure = PICT_TOP_FIELD + get_bits1(&h->gb); // bottom_field_flag
            } else {
                h->picture_structure = PICT_FRAME;
            }
        }

        if (h->nal_unit_type == NAL_IDR_SLICE)
            get_ue_golomb(&h->gb); /* idr_pic_id */
        if (h->sps.poc_type == 0) {
            h->poc_lsb = get_bits(&h->gb, h->sps.log2_max_poc_lsb);

            if (h->pps.pic_order_present == 1 &&
                h->picture_structure == PICT_FRAME)
                h->delta_poc_bottom = get_se_golomb(&h->gb);
        }

        if (h->sps.poc_type == 1 &&
            !h->sps.delta_pic_order_always_zero_flag) {
            h->delta_poc[0] = get_se_golomb(&h->gb);

            if (h->pps.pic_order_present == 1 &&
                h->picture_structure == PICT_FRAME)
                h->delta_poc[1] = get_se_golomb(&h->gb);
        }

        /* Decode POC of this picture.
         * The prev_ values needed for decoding POC of the next picture are not set here. */
        field_poc[0] = field_poc[1] = INT_MAX;
        ff_init_poc(h, field_poc, &s->output_picture_number);

        /* Continue parsing to check if MMCO_RESET is present.
         * FIXME: MMCO_RESET could appear in non-first slice.
         *        Maybe, we should parse all undisposable non-IDR slice of this
         *        picture until encountering MMCO_RESET in a slice of it. */
        if (h->nal_ref_idc && h->nal_unit_type != NAL_IDR_SLICE) {
            got_reset = scan_mmco_reset(s);
            if (got_reset < 0)
                return got_reset;
        }

        /* Set up the prev_ values for decoding POC of the next picture. */
        h->prev_frame_num        = got_reset ? 0 : h->frame_num;
        h->prev_frame_num_offset = got_reset ? 0 : h->frame_num_offset;
        if (h->nal_ref_idc != 0) {
            if (!got_reset) {
                h->prev_poc_msb = h->poc_msb;
                h->prev_poc_lsb = h->poc_lsb;
            } else {
                h->prev_poc_msb = 0;
                h->prev_poc_lsb =
                    h->picture_structure == PICT_BOTTOM_FIELD ? 0 : field_poc[0];
            }
        }
```

可以看出该部分代码提取了根据NALU Header、Slice Header中的信息赋值了一些字段，比如说AVCodecParserContext中的key_frame、pict_type，H264Context中的sps、pps、frame_num等等。

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

文章标签： FFmpeg H.264 解码 源代码 NALU

个人分类： FFMPEG

所属专栏： FFmpeg