

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）](#)

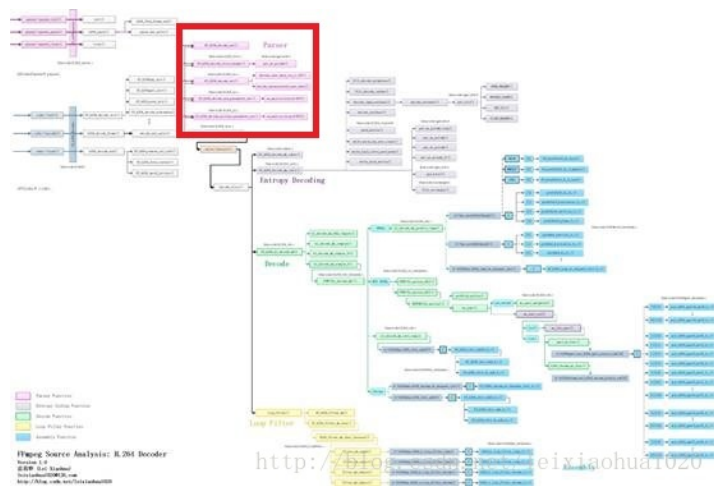
[FFmpeg的H.264解码器源代码简单分析：环路滤波（Loop Filter）部分](#)

=====

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。

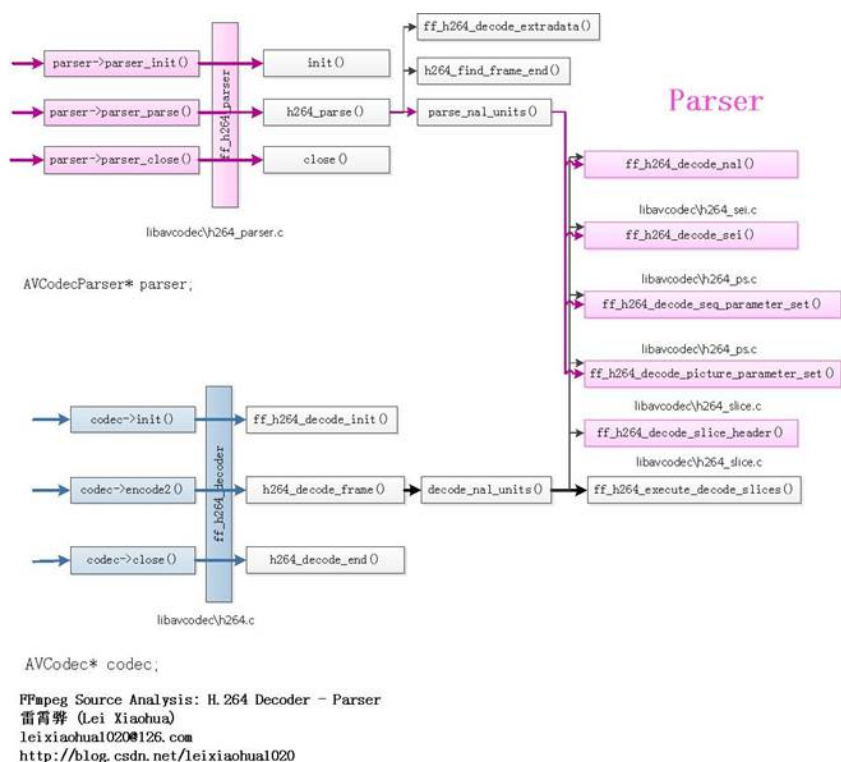
函数调用关系图

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]
1.  AVCodec ff_h264_decoder = {
2.      .name           = "h264",
3.      .long_name      = NULL_IF_CONFIG_SMALL("H.264 / AVC / MPEG-4 AVC / MPEG-4 part 10"),
4.      .type           = AVMEDIA_TYPE_VIDEO,
5.      .id             = AV_CODEC_ID_H264,
6.      .priv_data_size  = sizeof(H264Context),
7.      .init            = ff_h264_decode_init,
8.      .close           = h264_decode_end,
9.      .decode          = h264_decode_frame,
10.     .capabilities     = /*CODEC_CAP_DRAW_HORIZ_BAND |*/ CODEC_CAP_DR1 |
11.                        CODEC_CAP_DELAY | CODEC_CAP_SLICE_THREADS |
12.                        CODEC_CAP_FRAME_THREADS,
13.     .flush            = flush_dpb,
14.     .init_thread_copy = ONLY_IF_THREADS_ENABLED(decode_init_thread_copy),
15.     .update_thread_context = ONLY_IF_THREADS_ENABLED(ff_h264_update_thread_context),
16.     .profiles         = NULL_IF_CONFIG_SMALL(profiles),
17.     .priv_class       = &h264_class,
18. };
```

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]
1.  AVCodecParser ff_h264_parser = {
2.      .codec_ids      = { AV_CODEC_ID_H264 },
3.      .priv_data_size = sizeof(H264Context),
4.      .parser_init     = init,
5.      .parser_parse    = h264_parse,
6.      .parser_close    = close,
7.      .split           = h264_split,
8.  };
```

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]
1.  static av_cold int init(AVCodecParserContext *s)
2.  {
3.      H264Context *h = s->priv_data;
4.      h->thread_context[0] = h;
5.      h->slice_context_count = 1;
6.      ff_h264dsp_init(&h->h264dsp, 8, 1);
7.      return 0;
8.  }
```

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]
1.  static void close(AVCodecParserContext *s)
2.  {
3.      H264Context *h = s->priv_data;
4.      ParseContext *pc = &h->parse_context;
5.
6.      av_freep(&pc->buffer);
7.      ff_h264_free_context(h);
8.  }
```

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。

```

1. //解析H.264码流
2. //输出一个完整的NAL, 存储于poutbuf中
3. static int h264_parse(AVCodecParserContext *s,
4.                     AVCodecContext *avctx,
5.                     const uint8_t **poutbuf, int *poutbuf_size,
6.                     const uint8_t *buf, int buf_size)
7. {
8.     H264Context *h = s->priv_data;
9.     ParseContext *pc = &h->parse_context;
10.    int next;
11.    //如果还没有解析过1帧, 就调用这里解析extradata
12.    if (!h->got_first) {
13.        h->got_first = 1;
14.        if (avctx->extradata_size) {
15.            h->avctx = avctx;
16.            // must be done like in decoder, otherwise opening the parser,
17.            // letting it create extradata and then closing and opening again
18.            // will cause has_b_frames to be always set.
19.            // Note that estimate_timings_from_pts does exactly this.
20.            if (!avctx->has_b_frames)
21.                h->low_delay = 1;
22.            //解析AVCodecContext的extradata
23.            ff_h264_decode_extradata(h, avctx->extradata, avctx->extradata_size);
24.        }
25.    }
26.    //输入的数据是完整的一帧?
27.    //这里通过设置flags的PARSER_FLAG_COMPLETE_FRAMES来确定
28.    if (s->flags & PARSER_FLAG_COMPLETE_FRAMES) {
29.        //和缓存大小一样
30.        next = buf_size;
31.    } else {
32.        //查找帧结尾(帧开始)位置
33.        //以“起始码”为依据(0x0000001或0x00000001)
34.        next = h264_find_frame_end(h, buf, buf_size);
35.        //组帧
36.        if (ff_combine_frame(pc, next, &buf, &buf_size) < 0) {
37.            *poutbuf = NULL;
38.            *poutbuf_size = 0;
39.            return buf_size;
40.        }
41.
42.        if (next < 0 && next != END_NOT_FOUND) {
43.            av_assert1(pc->last_index + next >= 0);
44.            h264_find_frame_end(h, &pc->buffer[pc->last_index + next], -next); // update state
45.        }
46.    }
47.    //解析NALU, 从SPS、PPS、SEI中获得一些基本信息。
48.    //此时buf中存储的是完整的1帧数据
49.    parse_nal_units(s, avctx, buf, buf_size);
50.
51.    if (avctx->framerate.num)
52.        avctx->time_base = av_inv_q(av_mul_q(avctx->framerate, (AVRational){avctx->ticks_per_frame, 1}));
53.    if (h->sei_cpb_removal_delay >= 0) {
54.        s->pts_sync_point = h->sei_buffering_period_present;
55.        s->pts_ref_dts_delta = h->sei_cpb_removal_delay;
56.        s->pts_dts_delta = h->sei_dpb_output_delay;
57.    } else {
58.        s->pts_sync_point = INT_MIN;
59.        s->pts_ref_dts_delta = INT_MIN;
60.        s->pts_dts_delta = INT_MIN;
61.    }
62.
63.    if (s->flags & PARSER_FLAG_ONCE) {
64.        s->flags &= PARSER_FLAG_COMPLETE_FRAMES;
65.    }
66.    //分割后的帧数据输出至poutbuf
67.    *poutbuf = buf;
68.    *poutbuf_size = buf_size;
69.    return next;
70. }

```

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]
1. //解析extradata
2. //最常见的就是解析AVCodecContext的extradata。其中extradata实际上存储的就是SPS、PPS
3. int ff_h264_decode_extradata(H264Context *h, const uint8_t *buf, int size)
4. {
5.     AVCodecContext *avctx = h->avctx;
6.     int ret;
7.
8.     if (!buf || size <= 0)
9.         return -1;
10.
11.     if (buf[0] == 1) {
12.         int i, cnt, nalsize;
13.         const unsigned char *p = buf;
14.
15.         //AVC1 描述:H.264 bitstream without start codes.是不带起始码0x00000001的。MKV/MOV/FLV中的H.264属于这种类型
16.         //H264 描述:H.264 bitstream with start codes.是带有起始码0x00000001的。MPEGTS中的H.264，或者H.264裸流属于这种类型
17.         h->is_avc = 1;
18.         //数据量太小
19.         //随意测了一个视频
20.         //SPS: 30 Byte
21.         //PPS: 6 Byte
22.         if (size < 7) {
23.             av_log(avctx, AV_LOG_ERROR,
24.                 "avcC %d too short\n", size);
25.             return AVERROR_INVALIDDATA;
26.         }
27.         /* sps and pps in the avcC always have length coded with 2 bytes,
28.          * so put a fake nal_length_size = 2 while parsing them */
29.         h->nal_length_size = 2;
30.         // Decode sps from avcC
31.         //解码SPS
32.         cnt = *(p + 5) & 0x1f; // Number of sps
33.         p += 6;
34.         for (i = 0; i < cnt; i++) {
35.             nalsize = AV_RB16(p) + 2;
36.             if(nalsize > size - (p-buf))
37.                 return AVERROR_INVALIDDATA;
38.             //解析
39.             ret = decode_nal_units(h, p, nalsize, 1);
40.             if (ret < 0) {
41.                 av_log(avctx, AV_LOG_ERROR,
42.                     "Decoding sps %d from avcC failed\n", i);
43.                 return ret;
44.             }
45.             p += nalsize;
46.         }
47.         // Decode pps from avcC
48.         //解码PPS
49.         cnt = *(p++); // Number of pps
50.         for (i = 0; i < cnt; i++) {
51.             nalsize = AV_RB16(p) + 2;
52.             if(nalsize > size - (p-buf))
53.                 return AVERROR_INVALIDDATA;
54.             ret = decode_nal_units(h, p, nalsize, 1);
55.             if (ret < 0) {
56.                 av_log(avctx, AV_LOG_ERROR,
57.                     "Decoding pps %d from avcC failed\n", i);
58.                 return ret;
59.             }
60.             p += nalsize;
61.         }
62.         // Store right nal length size that will be used to parse all other nals
63.         h->nal_length_size = (buf[4] & 0x03) + 1;
64.     } else {
65.         h->is_avc = 0;
66.         //解析
67.         ret = decode_nal_units(h, buf, size, 1);
68.         if (ret < 0)
69.             return ret;
70.     }
71.     return size;
72. }
```

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]
1. //查找帧结尾（帧开始）位置
```

```

2. //
3. //几种状态state:
4. //2 - 找到1个0
5. //1 - 找到2个0
6. //0 - 找到大于等于3个0
7. //4 - 找到2个0和1个1, 即001 (即找到了起始码)
8. //5 - 找到至少3个0和1个1, 即0001等等 (即找到了起始码)
9. //7 - 初始化状态
10. //>=8 - 找到2个Slice Header
11. //
12. //关于起始码startcode的两种形式: 3字节的0x000001和4字节的0x00000001
13. //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,
14. //包含这些slice的nal使用3字节起始码。其余场合都是4字节的。
15. //
16. static int h264_find_frame_end(H264Context *h, const uint8_t *buf,
17.                               int buf_size)
18. {
19.     int i, j;
20.     uint32_t state;
21.     ParseContext *pc = &h->parse_context;
22.     int next_avc= h->is_avc ? 0 : buf_size;
23.
24.     // mb_addr= pc->mb_addr - 1;
25.     state = pc->state;
26.     if (state > 13)
27.         state = 7;
28.
29.     if (h->is_avc && !h->nal_length_size)
30.         av_log(h->avctx, AV_LOG_ERROR, "AVC-parser: nal length size invalid\n");
31.     //
32.     //每次循环前进1个字节, 读取该字节的值
33.     //根据此前的状态state做不同的处理
34.     //state取值为4,5代表找到了起始码
35.     //类似于一个状态机, 简单画一下状态转移图:
36.     //
37.     //           +-----+
38.     //           |       |
39.     //           v       |
40.     // 7--(0)-->2--(0)-->1--(0)-->0--(0)-+
41.     // ^      |      |      |
42.     // |      (1)    (1)    (1)
43.     // |      |      |      |
44.     // +-----+      v      v
45.     //           4      5
46.     //
47.     for (i = 0; i < buf_size; i++) {
48.         //超过了
49.         if (i >= next_avc) {
50.             int nalsize = 0;
51.             i = next_avc;
52.             for (j = 0; j < h->nal_length_size; j++)
53.                 nalsize = (nalsize << 8) | buf[i++];
54.             if (nalsize <= 0 || nalsize > buf_size - i) {
55.                 av_log(h->avctx, AV_LOG_ERROR, "AVC-parser: nal size %d remaining %d\n", nalsize, buf_size - i);
56.                 return buf_size;
57.             }
58.             next_avc = i + nalsize;
59.             state = 5;
60.         }
61.         //初始state为7
62.         if (state == 7) {
63.             //查找startcode的候选者?
64.             //从一段内存中查找取值为0的元素的位置并返回
65.             //增加i取值
66.             i += h->h264dsp.startcode_find_candidate(buf + i, next_avc - i);
67.             //因为找到1个0, 状态转换为2
68.             if (i < next_avc)
69.                 state = 2;
70.         } 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)。
71.             if (buf[i] == 1) //发现了一个1
72.                 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
73.             else if (buf[i])
74.                 state = 7; //恢复初始
75.             else //发现了一个0
76.                 state >= 1; // 2->1, 1->0, 0->0
77.         } else if (state <= 5) {
78.             //状态4代表找到了001, 状态5代表找到了0001
79.             //获取NALU类型
80.             //NALU Header (1Byte) 的后5bit
81.             int nalu_type = buf[i] & 0x1F;
82.
83.             if (nalu_type == NAL_SEI || nalu_type == NAL_SPS ||
84.                 nalu_type == NAL_PPS || nalu_type == NAL_AUD) {
85.                 //SPS, PPS, SEI类型的NALU
86.                 if (pc->frame_start_found) { //如果之前已找到了帧头
87.                     i++;
88.                     goto found;
89.                 }
90.             } else if (nalu_type == NAL_SLICE || nalu_type == NAL_DPA ||
91.                 nalu_type == NAL_IDR_SLICE) {
92.                 //表示有slice header的NALU
93.                 //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
94.                 //
95.                 //
96.                 //
97.                 //
98.                 //
99.                 //
100.                //
101.                //
102.                //
103.                //
104.                //
105.                //
106.                //
107.                //
108.                //
109.                //
110.                //
111.                //
112.                //
113.                //
114.                //
115.                //
116.                //
117.                //
118.                //
119.                //
120.                //
121.                //
122.                //
123.                //
124.                //
125.                //
126.                //
127.                //
128.                //
129.                //
130.                //
131.                //
132.                //
133.                //
134.                //
135.                //
136.                //
137.                //
138.                //
139.                //
140.                //
141.                //
142.                //
143.                //
144.                //
145.                //
146.                //
147.                //
148.                //
149.                //
150.                //
151.                //
152.                //
153.                //
154.                //
155.                //
156.                //
157.                //
158.                //
159.                //
160.                //
161.                //
162.                //
163.                //
164.                //
165.                //
166.                //
167.                //
168.                //
169.                //
170.                //
171.                //
172.                //
173.                //
174.                //
175.                //
176.                //
177.                //
178.                //
179.                //
180.                //
181.                //
182.                //
183.                //
184.                //
185.                //
186.                //
187.                //
188.                //
189.                //
190.                //
191.                //
192.                //
193.                //
194.                //
195.                //
196.                //
197.                //
198.                //
199.                //
200.                //
201.                //
202.                //
203.                //
204.                //
205.                //
206.                //
207.                //
208.                //
209.                //
210.                //
211.                //
212.                //
213.                //
214.                //
215.                //
216.                //
217.                //
218.                //
219.                //
220.                //
221.                //
222.                //
223.                //
224.                //
225.                //
226.                //
227.                //
228.                //
229.                //
230.                //
231.                //
232.                //
233.                //
234.                //
235.                //
236.                //
237.                //
238.                //
239.                //
240.                //
241.                //
242.                //
243.                //
244.                //
245.                //
246.                //
247.                //
248.                //
249.                //
250.                //
251.                //
252.                //
253.                //
254.                //
255.                //
256.                //
257.                //
258.                //
259.                //
260.                //
261.                //
262.                //
263.                //
264.                //
265.                //
266.                //
267.                //
268.                //
269.                //
270.                //
271.                //
272.                //
273.                //
274.                //
275.                //
276.                //
277.                //
278.                //
279.                //
280.                //
281.                //
282.                //
283.                //
284.                //
285.                //
286.                //
287.                //
288.                //
289.                //
290.                //
291.                //
292.                //
293.                //
294.                //
295.                //
296.                //
297.                //
298.                //
299.                //
300.                //
301.                //
302.                //
303.                //
304.                //
305.                //
306.                //
307.                //
308.                //
309.                //
310.                //
311.                //
312.                //
313.                //
314.                //
315.                //
316.                //
317.                //
318.                //
319.                //
320.                //
321.                //
322.                //
323.                //
324.                //
325.                //
326.                //
327.                //
328.                //
329.                //
330.                //
331.                //
332.                //
333.                //
334.                //
335.                //
336.                //
337.                //
338.                //
339.                //
340.                //
341.                //
342.                //
343.                //
344.                //
345.                //
346.                //
347.                //
348.                //
349.                //
350.                //
351.                //
352.                //
353.                //
354.                //
355.                //
356.                //
357.                //
358.                //
359.                //
360.                //
361.                //
362.                //
363.                //
364.                //
365.                //
366.                //
367.                //
368.                //
369.                //
370.                //
371.                //
372.                //
373.                //
374.                //
375.                //
376.                //
377.                //
378.                //
379.                //
380.                //
381.                //
382.                //
383.                //
384.                //
385.                //
386.                //
387.                //
388.                //
389.                //
390.                //
391.                //
392.                //
393.                //
394.                //
395.                //
396.                //
397.                //
398.                //
399.                //
400.                //
401.                //
402.                //
403.                //
404.                //
405.                //
406.                //
407.                //
408.                //
409.                //
410.                //
411.                //
412.                //
413.                //
414.                //
415.                //
416.                //
417.                //
418.                //
419.                //
420.                //
421.                //
422.                //
423.                //
424.                //
425.                //
426.                //
427.                //
428.                //
429.                //
430.                //
431.                //
432.                //
433.                //
434.                //
435.                //
436.                //
437.                //
438.                //
439.                //
440.                //
441.                //
442.                //
443.                //
444.                //
445.                //
446.                //
447.                //
448.                //
449.                //
450.                //
451.                //
452.                //
453.                //
454.                //
455.                //
456.                //
457.                //
458.                //
459.                //
460.                //
461.                //
462.                //
463.                //
464.                //
465.                //
466.                //
467.                //
468.                //
469.                //
470.                //
471.                //
472.                //
473.                //
474.                //
475.                //
476.                //
477.                //
478.                //
479.                //
480.                //
481.                //
482.                //
483.                //
484.                //
485.                //
486.                //
487.                //
488.                //
489.                //
490.                //
491.                //
492.                //
493.                //
494.                //
495.                //
496.                //
497.                //
498.                //
499.                //
500.                //
501.                //
502.                //
503.                //
504.                //
505.                //
506.                //
507.                //
508.                //
509.                //
510.                //
511.                //
512.                //
513.                //
514.                //
515.                //
516.                //
517.                //
518.                //
519.                //
520.                //
521.                //
522.                //
523.                //
524.                //
525.                //
526.                //
527.                //
528.                //
529.                //
530.                //
531.                //
532.                //
533.                //
534.                //
535.                //
536.                //
537.                //
538.                //
539.                //
540.                //
541.                //
542.                //
543.                //
544.                //
545.                //
546.                //
547.                //
548.                //
549.                //
550.                //
551.                //
552.                //
553.                //
554.                //
555.                //
556.                //
557.                //
558.                //
559.                //
560.                //
561.                //
562.                //
563.                //
564.                //
565.                //
566.                //
567.                //
568.                //
569.                //
570.                //
571.                //
572.                //
573.                //
574.                //
575.                //
576.                //
577.                //
578.                //
579.                //
580.                //
581.                //
582.                //
583.                //
584.                //
585.                //
586.                //
587.                //
588.                //
589.                //
590.                //
591.                //
592.                //
593.                //
594.                //
595.                //
596.                //
597.                //
598.                //
599.                //
600.                //
601.                //
602.                //
603.                //
604.                //
605.                //
606.                //
607.                //
608.                //
609.                //
610.                //
611.                //
612.                //
613.                //
614.                //
615.                //
616.                //
617.                //
618.                //
619.                //
620.                //
621.                //
622.                //
623.                //
624.                //
625.                //
626.                //
627.                //
628.                //
629.                //
630.                //
631.                //
632.                //
633.                //
634.                //
635.                //
636.                //
637.                //
638.                //
639.                //
640.                //
641.                //
642.                //
643.                //
644.                //
645.                //
646.                //
647.                //
648.                //
649.                //
650.                //
651.                //
652.                //
653.                //
654.                //
655.                //
656.                //
657.                //
658.                //
659.                //
660.                //
661.                //
662.                //
663.                //
664.                //
665.                //
666.                //
667.                //
668.                //
669.                //
670.                //
671.                //
672.                //
673.                //
674.                //
675.                //
676.                //
677.                //
678.                //
679.                //
680.                //
681.                //
682.                //
683.                //
684.                //
685.                //
686.                //
687.                //
688.                //
689.                //
690.                //
691.                //
692.                //
693.                //
694.                //
695.                //
696.                //
697.                //
698.                //
699.                //
700.                //
701.                //
702.                //
703.                //
704.                //
705.                //
706.                //
707.                //
708.                //
709.                //
710.                //
711.                //
712.                //
713.                //
714.                //
715.                //
716.                //
717.                //
718.                //
719.                //
720.                //
721.                //
722.                //
723.                //
724.                //
725.                //
726.                //
727.                //
728.                //
729.                //
730.                //
731.                //
732.                //
733.                //
734.                //
735.                //
736.                //
737.                //
738.                //
739.                //
740.                //
741.                //
742.                //
743.                //
744.                //
745.                //
746.                //
747.                //
748.                //
749.                //
750.                //
751.                //
752.                //
753.                //
754.                //
755.                //
756.                //
757.                //
758.                //
759.                //
760.                //
761.                //
762.                //
763.                //
764.                //
765.                //
766.                //
767.                //
768.                //
769.                //
770.                //
771.                //
772.                //
773.                //
774.                //
775.                //
776.                //
777.                //
778.                //
779.                //
780.                //
781.                //
782.                //
783.                //
784.                //
785.                //
786.                //
787.                //
788.                //
789.                //
790.                //
791.                //
792.                //
793.                //
794.                //
795.                //
796.                //
797.                //
798.                //
799.                //
800.                //
801.                //
802.                //
803.                //
804.                //
805.                //
806.                //
807.                //
808.                //
809.                //
810.                //
811.                //
812.                //
813.                //
814.                //
815.                //
816.                //
817.                //
818.                //
819.                //
820.                //
821.                //
822.                //
823.                //
824.                //
825.                //
826.                //
827.                //
828.                //
829.                //
830.                //
831.                //
832.                //
833.                //
834.                //
835.                //
836.                //
837.                //
838.                //
839.                //
840.                //
841.                //
842.                //
843.                //
844.                //
845.                //
846.                //
847.                //
848.                //
849.                //
850.                //
851.                //
852.                //
853.                //
854.                //
855.                //
856.                //
857.                //
858.                //
859.                //
860.                //
861.                //
862.                //
863.                //
864.                //
865.                //
866.                //
867.                //
868.                //
869.                //
870.                //
871.                //
872.                //
873.                //
874.                //
875.                //
876.                //
877.                //
878.                //
879.                //
880.                //
881.                //
882.                //
883.                //
884.                //
885.                //
886.                //
887.                //
888.                //
889.                //
890.                //
891.                //
892.                //
893.                //
894.                //
895.                //
896.                //
897.                //
898.                //
899.                //
900.                //
901.                //
902.                //
903.                //
904.                //
905.                //
906.                //
907.                //
908.                //
909.                //
910.                //
911.                //
912.                //
913.                //
914.                //
915.                //
916.                //
917.                //
918.                //
919.                //
920.                //
921.                //
922.                //
923.                //
924.                //
925.                //
926.                //
927.                //
928.                //
929.                //
930.                //
931.                //
932.                //
933.                //
934.                //
935.                //
936.                //
937.                //
938.                //
939.                //
940.                //
941.                //
942.                //
943.                //
944.                //
945.                //
946.                //
947.                //
948.                //
949.                //
950.                //
951.                //
952.                //
953.                //
954.                //
955.                //
956.                //
957.                //
958.                //
959.                //
960.                //
961.                //
962.                //
963.                //
964.                //
965.                //
966.                //
967.                //
968.                //
969.                //
970.                //
971.                //
972.                //
973.                //
974.                //
975.                //
976.                //
977.                //
978.                //
979.                //
980.                //
981.                //
982.                //
983.                //
984.                //
985.                //
986.                //
987.                //
988.                //
989.                //
990.                //
991.                //
992.                //
993.                //
994.                //
995.                //
996.                //
997.                //
998.                //
999.                //
1000.               //

```

```

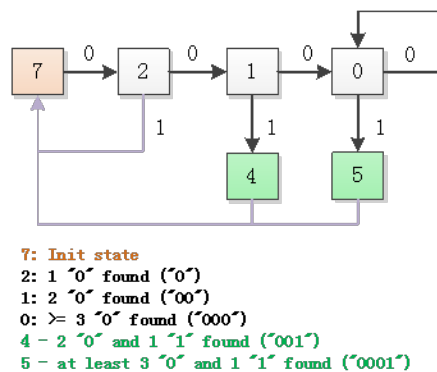
93.         state += 8;
94.         continue;
95.     }
96.     //上述两个条件都不满足, 回归初始状态 (state取值7)
97.     state = 7;
98. } else {
99.     h->parse_history[h->parse_history_count++] = buf[i];
100.    if (h->parse_history_count > 5) {
101.        unsigned int mb, last_mb = h->parse_last_mb;
102.        GetBitContext gb;
103.
104.        init_get_bits(&gb, h->parse_history, 8*h->parse_history_count);
105.        h->parse_history_count = 0;
106.        mb = get_ue_golomb_long(&gb);
107.        h->parse_last_mb = mb;
108.        if (pc->frame_start_found) {
109.            if (mb <= last_mb)
110.                goto found;
111.        } else
112.            pc->frame_start_found = 1;
113.        state = 7;
114.    }
115. }
116. }
117. pc->state = state;
118. if (h->is_avc)
119.     return next_avc;
120. //没找到
121. return END_NOT_FOUND;
122.
123. found:
124.     pc->state = 7;
125.     pc->frame_start_found = 0;
126.     if (h->is_avc)
127.         return next_avc;
128.     //state=4时候, state & 5=4
129.     //找到的是001 (长度为3), i减小3+1=4, 标识帧结尾
130.     //state=5时候, state & 5=5
131.     //找到的是0001 (长度为4), i减小4+1=5, 标识帧结尾
132.     return i - (state & 5) - 5 * (state > 7);
133. }

```

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。



FFmpeg Source Analysis: H.264 Decoder
Find Start Code - State machine

雷霄骅 (Lei Xiaohua)
leixiaohua1020@126.com
<http://blog.csdn.net/leixiaohua1020>

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]
1. int ff_startcode_find_candidate_c(const uint8_t *buf, int size)
2. {
3.     int i = 0;
4.     for (; i < size; i++)
5.         if (!buf[i])
6.             break;
7.     return i;
8. }
```

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_unit_s()的定义位于libavcodec\h264_parser.c，如下所示。

```
[cpp]
1. /**
2.  * Parse NAL units of found picture and decode some basic information.
3.  *
4.  * @param s parser context.
5.  * @param avctx codec context.
6.  * @param buf buffer with field/frame data.
7.  * @param buf_size size of the buffer.
8.  */
9. //解析NALU，从SPS、PPS、SEI等中获得一些基本信息。
10. static inline int parse_nal_units(AVCodecParserContext *s,
11.                                  AVCodecContext *avctx,
12.                                  const uint8_t * const buf, int buf_size)
13. {
14.     H264Context *h = s->priv_data;
15.     int buf_index, next_avc;
16.     unsigned int pps_id;
17.     unsigned int slice_type;
18.     int state = -1, got_reset = 0;
19.     const uint8_t *ptr;
20.     int q264 = buf_size >= 4 && !memcmp("Q264", buf, 4);
21.     int field_poc[2];
22.
23.     /* set some sane default values */
24.     s->pict_type = AV_PICTURE_TYPE_I;
25.     s->key_frame = 0;
26.     s->picture_structure = AV_PICTURE_STRUCTURE_UNKNOWN;
27.
28.     h->avctx = avctx;
29.     ff_h264_reset_sei(h);
30.     h->sei_fpa.frame_packing_arrangement_cancel_flag = -1;
31.
32.     if (!buf_size)
33.         return 0;
34.
35.     buf_index = 0;
36.     next_avc = h->is_avc ? 0 : buf_size;
37.     for (;;) {
38.         int src_length, dst_length, consumed, nalsize = 0;
39.
40.         if (buf_index >= next_avc) {
41.             nalsize = get_avc_nalsize(h, buf, buf_size, &buf_index);
42.             if (nalsize < 0)
43.                 break;
44.             next_avc = buf_index + nalsize;
45.         } else {
46.             buf_index = find_start_code(buf, buf_size, buf_index, next_avc);
47.             if (buf_index >= buf_size)
48.                 break;
49.             if (buf_index >= next_avc)
50.                 continue;
51.         }
52.         src_length = next_avc - buf_index;
53.         //NALU Header (1 Byte)
54.         state = buf[buf_index];
55.         switch (state & 0x1f) {
56.             case NAL_SLICE:
57.             case NAL_IDR_SLICE:
58.                 // Do not walk the whole buffer just to decode slice header
59.                 if ((state & 0x1f) == NAL_IDR_SLICE || ((state >> 5) & 0x3) == 0) {
60.                     /* IDR or disposable slice
61.                      * No need to decode many bytes because MMC0s shall not be present. */
62.                     if (src_length > 60)
```



```

62.         if (src_length < 60)
63.             src_length = 60;
64.     } else {
65.         /* To decode up to MMC0s */
66.         if (src_length > 1000)
67.             src_length = 1000;
68.     }
69.     break;
70. }
71. //解析NAL Header, 获得nal_unit_type等信息
72. ptr = ff_h264_decode_nal(h, buf + buf_index, &dst_length,
73.                          &consumed, src_length);
74. if (!ptr || dst_length < 0)
75.     break;
76.
77. buf_index += consumed;
78. //初始化GetBitContext
79. //H264Context->gb
80. //后面的解析都是从这里获取数据
81. init_get_bits(&h->gb, ptr, 8 * dst_length);
82. switch (h->nal_unit_type) {
83. case NAL_SPS:
84.     //解析SPS
85.     ff_h264_decode_seq_parameter_set(h);
86.     break;
87. case NAL_PPS:
88.     //解析PPS
89.     ff_h264_decode_picture_parameter_set(h, h->gb.size_in_bits);
90.     break;
91. case NAL_SEI:
92.     //解析SEI
93.     ff_h264_decode_sei(h);
94.     break;
95. case NAL_IDR_SLICE:
96.     //如果是IDR Slice
97.     //赋值AVCodecParserContext的key_frame为1
98.     s->key_frame = 1;
99.
100.     h->prev_frame_num = 0;
101.     h->prev_frame_num_offset = 0;
102.     h->prev_poc_msb = 0;
103.     h->prev_poc_lsb = 0;
104. /* fall through */
105. case NAL_SLICE:
106.     //获取SLICE的一些信息
107.     //跳过first_mb_in_slice这一字段
108.     get_ue_golomb_long(&h->gb); // skip first_mb_in_slice
109.     //获取帧类型 (I,B,P)
110.     slice_type = get_ue_golomb_31(&h->gb);
111.     //赋值到AVCodecParserContext的pict_type (外部可以访问到)
112.     s->pict_type = golomb_to_pict_type[slice_type % 5];
113.     //关键帧
114.     if (h->sei_recovery_frame_cnt >= 0) {
115.         /* key frame, since recovery_frame_cnt is set */
116.         //赋值AVCodecParserContext的key_frame为1
117.         s->key_frame = 1;
118.     }
119.     //获取 PPS ID
120.     pps_id = get_ue_golomb(&h->gb);
121.     if (pps_id >= MAX_PPS_COUNT) {
122.         av_log(h->avctx, AV_LOG_ERROR,
123.                "pps_id %u out of range\n", pps_id);
124.         return -1;
125.     }
126.     if (!h->pps_buffers[pps_id]) {
127.         av_log(h->avctx, AV_LOG_ERROR,
128.                "non-existing PPS %u referenced\n", pps_id);
129.         return -1;
130.     }
131.     h->pps = *h->pps_buffers[pps_id];
132.     if (!h->sps_buffers[h->pps.sps_id]) {
133.         av_log(h->avctx, AV_LOG_ERROR,
134.                "non-existing SPS %u referenced\n", h->pps.sps_id);
135.         return -1;
136.     }
137.     h->sps = *h->sps_buffers[h->pps.sps_id];
138.     h->frame_num = get_bits(&h->gb, h->sps.log2_max_frame_num);
139.
140.     if (h->sps.ref_frame_count <= 1 && h->pps.ref_count[0] <= 1 && s->pict_type == AV_PICTURE_TYPE_I)
141.         s->key_frame = 1;
142.     //获得“型”和“级”
143.     //赋值到AVCodecContext的profile和level
144.     avctx->profile = ff_h264_get_profile(&h->sps);
145.     avctx->level = h->sps.level_idc;
146.
147.     if (h->sps.frame_mbs_only_flag) {
148.         h->picture_structure = PICT_FRAME;
149.     } else {
150.         if (get_bits1(&h->gb)) { // field_pic_flag
151.             h->picture_structure = PICT_TOP_FIELD + get_bits1(&h->gb); // bottom_field_flag
152.         } else {
153.             h->picture_structure = PICT_FRAME;

```

```

154.     }
155. }
156.
157. if (h->nal_unit_type == NAL_IDR_SLICE)
158.     get_ue_golomb(&h->gb); /* idr_pic_id */
159. if (h->sps.poc_type == 0) {
160.     h->poc_lsb = get_bits(&h->gb, h->sps.log2_max_poc_lsb);
161.
162.     if (h->pps.pic_order_present == 1 &&
163.         h->picture_structure == PICT_FRAME)
164.         h->delta_poc_bottom = get_se_golomb(&h->gb);
165. }
166.
167. if (h->sps.poc_type == 1 &&
168.     !h->sps.delta_pic_order_always_zero_flag) {
169.     h->delta_poc[0] = get_se_golomb(&h->gb);
170.
171.     if (h->pps.pic_order_present == 1 &&
172.         h->picture_structure == PICT_FRAME)
173.         h->delta_poc[1] = get_se_golomb(&h->gb);
174. }
175.
176. /* Decode POC of this picture.
177.  * The prev_ values needed for decoding POC of the next picture are not set here. */
178. field_poc[0] = field_poc[1] = INT_MAX;
179. ff_init_poc(h, field_poc, &s->output_picture_number);
180.
181. /* Continue parsing to check if MMCO RESET is present.
182.  * FIXME: MMCO_RESET could appear in non-first slice.
183.  * Maybe, we should parse all undisposable non-IDR slice of this
184.  * picture until encountering MMCO_RESET in a slice of it. */
185. if (h->nal_ref_idc && h->nal_unit_type != NAL_IDR_SLICE) {
186.     got_reset = scan_mmco_reset(s);
187.     if (got_reset < 0)
188.         return got_reset;
189. }
190.
191. /* Set up the prev_ values for decoding POC of the next picture. */
192. h->prev_frame_num = got_reset ? 0 : h->frame_num;
193. h->prev_frame_num_offset = got_reset ? 0 : h->frame_num_offset;
194. if (h->nal_ref_idc != 0) {
195.     if (!got_reset) {
196.         h->prev_poc_msb = h->poc_msb;
197.         h->prev_poc_lsb = h->poc_lsb;
198.     } else {
199.         h->prev_poc_msb = 0;
200.         h->prev_poc_lsb =
201.             h->picture_structure == PICT_BOTTOM_FIELD ? 0 : field_poc[0];
202.     }
203. }
204. //包含“场”概念的时候，先不管
205. if (h->sps.pic_struct_present_flag) {
206.     switch (h->sei_pic_struct) {
207.     case SEI_PIC_STRUCT_TOP_FIELD:
208.     case SEI_PIC_STRUCT_BOTTOM_FIELD:
209.         s->repeat_pict = 0;
210.         break;
211.     case SEI_PIC_STRUCT_FRAME:
212.     case SEI_PIC_STRUCT_TOP_BOTTOM:
213.     case SEI_PIC_STRUCT_BOTTOM_TOP:
214.         s->repeat_pict = 1;
215.         break;
216.     case SEI_PIC_STRUCT_TOP_BOTTOM_TOP:
217.     case SEI_PIC_STRUCT_BOTTOM_TOP_BOTTOM:
218.         s->repeat_pict = 2;
219.         break;
220.     case SEI_PIC_STRUCT_FRAME_DOUBLING:
221.         s->repeat_pict = 3;
222.         break;
223.     case SEI_PIC_STRUCT_FRAME_TRIPLING:
224.         s->repeat_pict = 5;
225.         break;
226.     default:
227.         s->repeat_pict = h->picture_structure == PICT_FRAME ? 1 : 0;
228.         break;
229.     }
230. } else {
231.     s->repeat_pict = h->picture_structure == PICT_FRAME ? 1 : 0;
232. }
233.
234. if (h->picture_structure == PICT_FRAME) {
235.     s->picture_structure = AV_PICTURE_STRUCTURE_FRAME;
236.     if (h->sps.pic_struct_present_flag) {
237.         switch (h->sei_pic_struct) {
238.         case SEI_PIC_STRUCT_TOP_BOTTOM:
239.         case SEI_PIC_STRUCT_TOP_BOTTOM_TOP:
240.             s->field_order = AV_FIELD_TT;
241.             break;
242.         case SEI_PIC_STRUCT_BOTTOM_TOP:
243.         case SEI_PIC_STRUCT_BOTTOM_TOP_BOTTOM:
244.             s->field_order = AV_FIELD_BB;

```

```

245.         break;
246.     default:
247.         s->field_order = AV_FIELD_PROGRESSIVE;
248.         break;
249.     }
250. } else {
251.     if (field_poc[0] < field_poc[1])
252.         s->field_order = AV_FIELD_TT;
253.     else if (field_poc[0] > field_poc[1])
254.         s->field_order = AV_FIELD_BB;
255.     else
256.         s->field_order = AV_FIELD_PROGRESSIVE;
257. }
258. } else {
259.     if (h->picture_structure == PICT_TOP_FIELD)
260.         s->picture_structure = AV_PICTURE_STRUCTURE_TOP_FIELD;
261.     else
262.         s->picture_structure = AV_PICTURE_STRUCTURE_BOTTOM_FIELD;
263.     s->field_order = AV_FIELD_UNKNOWN;
264. }
265.
266.     return 0; /* no need to evaluate the rest */
267. }
268. }
269. if (q264)
270.     return 0;
271. /* didn't find a picture! */
272. av_log(h->avctx, AV_LOG_ERROR, "missing picture in access unit with size %d\n", buf_size);
273. return -1;
274. }

```

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。

```

1. //解析NAL Header, 获得nal_unit_type等信息
2. const uint8_t *ff_h264_decode_nal(H264Context *h, const uint8_t *src,
3.                                   int *dst_length, int *consumed, int length)
4. {
5.     int i, si, di;
6.     uint8_t *dst;
7.     int bufidx;
8.
9.     // src[0]&0x80; // forbidden bit
10.    //
11.    // 1 byte NALU头
12.    // forbidden_zero_bit: 1bit
13.    // nal_ref_idc: 2bit
14.    // nal_unit_type: 5bit
15.    // nal_ref_idc指示NAL的优先级，取值0-3，值越高，代表NAL越重要
16.    h->nal_ref_idc = src[0] >> 5;
17.    // nal_unit_type指示NAL的类型
18.    h->nal_unit_type = src[0] & 0x1F;
19.    //后移1Byte
20.    src++;
21.    //未处理数据长度减1
22.    length--;
23.
24.    //起始码:0x000001
25.    //保留:0x000002
26.    //防止竞争:0x000003
27.    //既表示NALU的开始，又表示NALU的结束
28.    //STARTCODE_TEST这个宏在后面用到
29.    //得到length
30.    //length是指当前NALU单元长度，这里不包括nal_u头信息长度（即1个字节）
31.    #define STARTCODE_TEST \
32.    if (i + 2 < length && src[i + 1] == 0 && src[i + 2] <= 3) { \
33.        if (src[i + 2] != 3 && src[i + 2] != 0) { \
34.            /* 取值为1或者2（保留用），为起始码。startcode, so we must be past the end */\
35.            length = i; \
36.        } \
37.        break; \
38.    }
39.
40.    #if HAVE_FAST_UNALIGNED
41.    #define FIND_FIRST_ZERO \

```

```

42.     if (i > 0 && !src[i])
43.         i--;
44.     while (src[i])
45.         i++;
46.
47. #if HAVE_FAST_64BIT
48.     for (i = 0; i + 1 < length; i += 9) {
49.         if (!((~AV_RN64A(src + i) &
50.             (AV_RN64A(src + i) - 0x0100010001000101ULL)) &
51.             0x8000800080008000ULL))
52.             continue;
53.         FIND_FIRST_ZERO;
54.         STARTCODE_TEST;
55.         i -= 7;
56.     }
57. #else
58.     for (i = 0; i + 1 < length; i += 5) {
59.         if (!((~AV_RN32A(src + i) &
60.             (AV_RN32A(src + i) - 0x01000101U)) &
61.             0x80008000U))
62.             continue;
63.         FIND_FIRST_ZERO;
64.         STARTCODE_TEST;
65.         i -= 3;
66.     }
67. #endif
68. #else
69.     for (i = 0; i + 1 < length; i += 2) {
70.         if (src[i])
71.             continue;
72.         if (i > 0 && src[i - 1] == 0)
73.             i--;
74.         //起始码检测
75.         STARTCODE_TEST;
76.     }
77. #endif
78.
79.     // use second escape buffer for inter data
80.     bufidx = h->nal_unit_type == NAL_DPC ? 1 : 0;
81.
82.     av_fast_padded_malloc(&h->rbsp_buffer[bufidx], &h->rbsp_buffer_size[bufidx], length+MAX_MBPAIR_SIZE);
83.     dst = h->rbsp_buffer[bufidx];
84.
85.     if (!dst)
86.         return NULL;
87.
88.     if(i>=length-1){ //no escaped 0
89.         *dst_length= length;
90.         *consumed= length+1; //+1 for the header
91.         if(h->avctx->flags2 & CODEC_FLAG2_FAST){
92.             return src;
93.         }else{
94.             memcpy(dst, src, length);
95.             return dst;
96.         }
97.     }
98.
99.     memcpy(dst, src, i);
100.    si = di = i;
101.    while (si + 2 < length) {
102.        // remove escapes (very rare 1:2^22)
103.        if (src[si + 2] > 3) {
104.            dst[di++] = src[si++];
105.            dst[di++] = src[si++];
106.        } else if (src[si] == 0 && src[si + 1] == 0 && src[si + 2] != 0) {
107.            if (src[si + 2] == 3) { // escape
108.                dst[di++] = 0;
109.                dst[di++] = 0;
110.                si += 3;
111.                continue;
112.            } else // next start code
113.                goto nsc;
114.        }
115.
116.        dst[di++] = src[si++];
117.    }
118.    while (si < length)
119.        dst[di++] = src[si++];
120.
121. nsc:
122.    memset(dst + di, 0, FF_INPUT_BUFFER_PADDING_SIZE);
123.
124.    *dst_length = di;
125.    *consumed = si + 1; // +1 for the header
126.    /* FIXME store exact number of bits in the getbitcontext
127.     * (it is needed for decoding) */
128.    return dst;
129. }

```

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]
1. //解码SPS
2. int ff_h264_decode_seq_parameter_set(H264Context *h)
3. {
4.     int profile_idc, level_idc, constraint_set_flags = 0;
5.     unsigned int sps_id;
6.     int i, log2_max_frame_num_minus4;
7.
8.     SPS *sps;
9.     //profile型, 8bit
10.    //注意get_bits()
11.    profile_idc = get_bits(&h->gb, 8);
12.    constraint_set_flags |= get_bits1(&h->gb) << 0; // constraint_set0_flag
13.    constraint_set_flags |= get_bits1(&h->gb) << 1; // constraint_set1_flag
14.    constraint_set_flags |= get_bits1(&h->gb) << 2; // constraint_set2_flag
15.    constraint_set_flags |= get_bits1(&h->gb) << 3; // constraint_set3_flag
16.    constraint_set_flags |= get_bits1(&h->gb) << 4; // constraint_set4_flag
17.    constraint_set_flags |= get_bits1(&h->gb) << 5; // constraint_set5_flag
18.    skip_bits(&h->gb, 2); // reserved_zero_2bits
19.    //level级, 8bit
20.    level_idc = get_bits(&h->gb, 8);
21.    //该SPS的ID号, 该ID号将被picture引用
22.    //注意: get_ue_golomb()
23.    sps_id = get_ue_golomb_31(&h->gb);
24.
25.    if (sps_id >= MAX_SPS_COUNT) {
26.        av_log(h->avctx, AV_LOG_ERROR, "sps_id %u out of range\n", sps_id);
27.        return AVERROR_INVALIDDATA;
28.    }
29.    //赋值给这个结构体
30.    sps = av_mallocz(sizeof(SPS));
31.    if (!sps)
32.        return AVERROR(ENOMEM);
33.    //赋值
34.    sps->sps_id = sps_id;
35.    sps->time_offset_length = 24;
36.    sps->profile_idc = profile_idc;
37.    sps->constraint_set_flags = constraint_set_flags;
38.    sps->level_idc = level_idc;
39.    sps->full_range = -1;
40.
41.    memset(sps->scaling_matrix4, 16, sizeof(sps->scaling_matrix4));
42.    memset(sps->scaling_matrix8, 16, sizeof(sps->scaling_matrix8));
43.    sps->scaling_matrix_present = 0;
44.    sps->colourspace = 2; //AVCOL_SPC_UNSPECIFIED
45.    //Profile对应关系
46.    if (sps->profile_idc == 100 || // High profile
47.        sps->profile_idc == 110 || // High10 profile
48.        sps->profile_idc == 122 || // High422 profile
49.        sps->profile_idc == 244 || // High444 Predictive profile
50.        sps->profile_idc == 44 || // Cavlc444 profile
51.        sps->profile_idc == 83 || // Scalable Constrained High profile (SVC)
52.        sps->profile_idc == 86 || // Scalable High Intra profile (SVC)
53.        sps->profile_idc == 118 || // Stereo High profile (MVC)
54.        sps->profile_idc == 128 || // Multiview High profile (MVC)
55.        sps->profile_idc == 138 || // Multiview Depth High profile (MVCD)
56.        sps->profile_idc == 144) { // old High444 profile
57.
58.        //色度取样
59.        //0代表单色
60.        //1代表4:2:0
61.        //2代表4:2:2
62.        //3代表4:4:4
63.        sps->chroma_format_idc = get_ue_golomb_31(&h->gb);
64.        if (sps->chroma_format_idc > 3U) {
65.            avpriv_request_sample(h->avctx, "chroma_format_idc %u",
66.                                   sps->chroma_format_idc);
67.            goto fail;
68.        } else if (sps->chroma_format_idc == 3) {
69.            sps->residual_color_transform_flag = get_bits1(&h->gb);
70.            if (sps->residual_color_transform_flag) {
71.                av_log(h->avctx, AV_LOG_ERROR, "separate color planes are not supported\n");
72.                goto fail;
73.            }
74.        }
75.        //bit depth luma minus8
```

```

76. //加8之后为亮度颜色深度
77. //该值取值范围应该在0到4之间。即颜色深度支持0-12bit
78. sps->bit_depth_luma = get_ue_golomb(&h->gb) + 8;
79. //加8之后为色度颜色深度
80. sps->bit_depth_chroma = get_ue_golomb(&h->gb) + 8;
81. if (sps->bit_depth_chroma != sps->bit_depth_luma) {
82.     avpriv_request_sample(h->avctx,
83.         "Different chroma and luma bit depth");
84.     goto fail;
85. }
86. if (sps->bit_depth_luma > 14U || sps->bit_depth_chroma > 14U) {
87.     av_log(h->avctx, AV_LOG_ERROR, "illegal bit depth value (%d, %d)\n",
88.         sps->bit_depth_luma, sps->bit_depth_chroma);
89.     goto fail;
90. }
91. sps->transform_bypass = get_bits1(&h->gb);
92. decode_scaling_matrices(h, sps, NULL, 1,
93.     sps->scaling_matrix4, sps->scaling_matrix8);
94. } else {
95.     //默认
96.     sps->chroma_format_idc = 1;
97.     sps->bit_depth_luma = 8;
98.     sps->bit_depth_chroma = 8;
99. }
100. //log2_max_frame_num_minus4为另一个句法元素frame_num服务
101. //frame_num的解码函数是ue(v)，函数中的v在这里指定：
102. // v = log2_max_frame_num_minus4 + 4
103. //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：
104. // MaxFrameNum = 2^( log2_max_frame_num_minus4 + 4 )
105. log2_max_frame_num_minus4 = get_ue_golomb(&h->gb);
106. if (log2_max_frame_num_minus4 < MIN_LOG2_MAX_FRAME_NUM - 4 ||
107.     log2_max_frame_num_minus4 > MAX_LOG2_MAX_FRAME_NUM - 4) {
108.     av_log(h->avctx, AV_LOG_ERROR,
109.         "log2_max_frame_num_minus4 out of range (0-12): %d\n",
110.         log2_max_frame_num_minus4);
111.     goto fail;
112. }
113. sps->log2_max_frame_num = log2_max_frame_num_minus4 + 4;
114. //pic_order_cnt_type 指明了poc (picture order count) 的编码方法
115. //poc标识图像的播放顺序。
116. //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
117. //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。
118. //H.264 中一共定义了三种poc 的编码方法
119. sps->poc_type = get_ue_golomb_31(&h->gb);
120. //3种poc的编码方法
121. if (sps->poc_type == 0) { // FIXME #define
122.     unsigned t = get_ue_golomb(&h->gb);
123.     if (t>12) {
124.         av_log(h->avctx, AV_LOG_ERROR, "log2_max_poc_lsb (%d) is out of range\n", t);
125.         goto fail;
126.     }
127.     sps->log2_max_poc_lsb = t + 4;
128. } else if (sps->poc_type == 1) { // FIXME #define
129.     sps->delta_pic_order_always_zero_flag = get_bits1(&h->gb);
130.     sps->offset_for_non_ref_pic = get_se_golomb(&h->gb);
131.     sps->offset_for_top_to_bottom_field = get_se_golomb(&h->gb);
132.     sps->poc_cycle_length = get_ue_golomb(&h->gb);
133.
134.     if ((unsigned)sps->poc_cycle_length >=
135.         FF_ARRAY_ELEMS(sps->offset_for_ref_frame)) {
136.         av_log(h->avctx, AV_LOG_ERROR,
137.             "poc_cycle_length overflow %d\n", sps->poc_cycle_length);
138.         goto fail;
139.     }
140.
141.     for (i = 0; i < sps->poc_cycle_length; i++)
142.         sps->offset_for_ref_frame[i] = get_se_golomb(&h->gb);
143. } else if (sps->poc_type != 2) {
144.     av_log(h->avctx, AV_LOG_ERROR, "illegal POC type %d\n", sps->poc_type);
145.     goto fail;
146. }
147. //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，
148. //H.264 规定最多可用16 个参考帧，因此最大值为16。
149. sps->ref_frame_count = get_ue_golomb_31(&h->gb);
150. if (h->avctx->codec_tag == MKTAG('S', 'M', 'V', '2'))
151.     sps->ref_frame_count = FFMAX(2, sps->ref_frame_count);
152. if (sps->ref_frame_count > H264_MAX_PICTURE_COUNT - 2 ||
153.     sps->ref_frame_count > 16U) {
154.     av_log(h->avctx, AV_LOG_ERROR,
155.         "too many reference frames %d\n", sps->ref_frame_count);
156.     goto fail;
157. }
158. sps->gaps_in_frame_num_allowed_flag = get_bits1(&h->gb);
159. //加1后为图像宽（以宏块为单位）
160. //以像素为单位图像宽度（亮度）：width=mb_width*16
161. sps->mb_width = get_ue_golomb(&h->gb) + 1;
162. //加1后为图像高（以宏块为单位）
163. //以像素为单位图像高度（亮度）：height=mb_height*16
164. sps->mb_height = get_ue_golomb(&h->gb) + 1;
165. //检查一下
166. if ((unsigned)sps->mb_width >= INT_MAX / 16 ||

```

```

167.     (unsigned)sps->mb_height >= INT_MAX / 16 ||
168.     av_image_check_size(16 * sps->mb_width,
169.         16 * sps->mb_height, 0, h->avctx)) {
170.     av_log(h->avctx, AV_LOG_ERROR, "mb_width/height overflow\n");
171.     goto fail;
172. }
173.
174. sps->frame_mbs_only_flag = get_bits1(&h->gb);
175. if (!sps->frame_mbs_only_flag)
176.     sps->mb_aff = get_bits1(&h->gb);
177. else
178.     sps->mb_aff = 0;
179.
180. sps->direct_8x8_inference_flag = get_bits1(&h->gb);
181.
182. #ifndef ALLOW_INTERLACE
183.     if (sps->mb_aff)
184.         av_log(h->avctx, AV_LOG_ERROR,
185.             "MBAFF support not included; enable it at compile-time.\n");
186. #endif
187.     //裁剪输出, 没研究过
188.     sps->crop = get_bits1(&h->gb);
189.     if (sps->crop) {
190.         int crop_left  = get_ue_golomb(&h->gb);
191.         int crop_right = get_ue_golomb(&h->gb);
192.         int crop_top   = get_ue_golomb(&h->gb);
193.         int crop_bottom = get_ue_golomb(&h->gb);
194.         int width  = 16 * sps->mb_width;
195.         int height = 16 * sps->mb_height * (2 - sps->frame_mbs_only_flag);
196.
197.         if (h->avctx->flags2 & CODEC_FLAG2_IGNORE_CROP) {
198.             av_log(h->avctx, AV_LOG_DEBUG, "discarding sps cropping, original "
199.                 "values are l:%d r:%d t:%d b:%d\n",
200.                 crop_left, crop_right, crop_top, crop_bottom);
201.
202.             sps->crop_left  =
203.             sps->crop_right =
204.             sps->crop_top   =
205.             sps->crop_bottom = 0;
206.         } else {
207.             int vsub  = (sps->chroma_format_idc == 1) ? 1 : 0;
208.             int hsub  = (sps->chroma_format_idc == 1 ||
209.                 sps->chroma_format_idc == 2) ? 1 : 0;
210.             int step_x = 1 << hsub;
211.             int step_y = (2 - sps->frame_mbs_only_flag) << vsub;
212.
213.             if (crop_left & (0x1F >> (sps->bit_depth_luma > 8)) &&
214.                 !(h->avctx->flags & CODEC_FLAG_UNALIGNED)) {
215.                 crop_left &= ~(0x1F >> (sps->bit_depth_luma > 8));
216.                 av_log(h->avctx, AV_LOG_WARNING,
217.                     "Reducing left cropping to %d "
218.                     "chroma samples to preserve alignment.\n",
219.                     crop_left);
220.             }
221.
222.             if (crop_left > (unsigned)INT_MAX / 4 / step_x ||
223.                 crop_right > (unsigned)INT_MAX / 4 / step_x ||
224.                 crop_top > (unsigned)INT_MAX / 4 / step_y ||
225.                 crop_bottom > (unsigned)INT_MAX / 4 / step_y ||
226.                 (crop_left + crop_right) * step_x >= width ||
227.                 (crop_top + crop_bottom) * step_y >= height
228.             ) {
229.                 av_log(h-
230. >avctx, AV_LOG_ERROR, "crop values invalid %d %d %d %d / %d %d\n", crop_left, crop_right, crop_top, crop_bottom, width, height);
231.                 goto fail;
232.             }
233.
234.             sps->crop_left  = crop_left  * step_x;
235.             sps->crop_right = crop_right * step_x;
236.             sps->crop_top   = crop_top   * step_y;
237.             sps->crop_bottom = crop_bottom * step_y;
238.         }
239.     } else {
240.         sps->crop_left  =
241.         sps->crop_right =
242.         sps->crop_top   =
243.         sps->crop_bottom =
244.         sps->crop       = 0;
245.     }
246.
247.     sps->vui_parameters_present_flag = get_bits1(&h->gb);
248.     if (sps->vui_parameters_present_flag) {
249.         int ret = decode_vui_parameters(h, sps);
250.         if (ret < 0)
251.             goto fail;
252.     }
253.
254.     if (!sps->sar.den)
255.         sps->sar.den = 1;
256.     //Debug的时候可以输出一些信息
257.     if (h->avctx->debug & FF_DEBUG_PICT_INFO) {

```

```

257. static const char csp[4][5] = { "Gray", "420", "422", "444" };
258. av_log(h->avctx, AV_LOG_DEBUG,
259.        "sps:%u profile:%d/%d poc:%d ref:%d %dx%d %s %s crop:%u/%u/%u %s %s "PRIid32"/%"PRIid32" b%d reo:%d\n",
260.        sps_id, sps->profile_idc, sps->level_idc,
261.        sps->poc_type,
262.        sps->ref_frame_count,
263.        sps->mb_width, sps->mb_height,
264.        sps->frame_mbs_only_flag ? "FRM" : (sps->mb_aff ? "MB-AFF" : "PIC-AFF"),
265.        sps->direct_8x8_inference_flag ? "8B8" : "",
266.        sps->crop_left, sps->crop_right,
267.        sps->crop_top, sps->crop_bottom,
268.        sps->vui_parameters_present_flag ? "VUI" : "",
269.        csp[sps->chroma_format_idc],
270.        sps->timing_info_present_flag ? sps->num_units_in_tick : 0,
271.        sps->timing_info_present_flag ? sps->time_scale : 0,
272.        sps->bit_depth_luma,
273.        sps->bitstream_restriction_flag ? sps->num_reorder_frames : -1
274.        );
275. }
276. sps->new = 1;
277.
278. av_free(h->sps_buffers[sps_id]);
279. h->sps_buffers[sps_id] = sps;
280.
281. return 0;
282.
283. fail:
284.     av_free(sps);
285.     return -1;
286. }

```

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。

```

1. //解码PPS
2. int ff_h264_decode_picture_parameter_set(H264Context *h, int bit_length)
3. {
4.     //获取PPS ID
5.     unsigned int pps_id = get_ue_golomb(&h->gb);
6.     PPS *pps;
7.     SPS *sps;
8.     int qp_bd_offset;
9.     int bits_left;
10.
11.     if (pps_id >= MAX_PPS_COUNT) {
12.         av_log(h->avctx, AV_LOG_ERROR, "pps_id %u out of range\n", pps_id);
13.         return AVERROR_INVALIDDATA;
14.     }
15.     //解析后赋值给PPS这个结构体
16.     pps = av_mallocz(sizeof(PPS));
17.     if (!pps)
18.         return AVERROR(ENOMEM);
19.     //该PPS引用的SPS的ID
20.     pps->sps_id = get_ue_golomb_31(&h->gb);
21.     if ((unsigned)pps->sps_id >= MAX_SPS_COUNT ||
22.         !h->sps_buffers[pps->sps_id]) {
23.         av_log(h->avctx, AV_LOG_ERROR, "sps_id %u out of range\n", pps->sps_id);
24.         goto fail;
25.     }
26.     sps = h->sps_buffers[pps->sps_id];
27.     qp_bd_offset = 6 * (sps->bit_depth_luma - 8);
28.     if (sps->bit_depth_luma > 14) {
29.         av_log(h->avctx, AV_LOG_ERROR,
30.             "Invalid luma bit depth=%d\n",
31.             sps->bit_depth_luma);
32.         goto fail;
33.     } else if (sps->bit_depth_luma == 11 || sps->bit_depth_luma == 13) {
34.         av_log(h->avctx, AV_LOG_ERROR,
35.             "Unimplemented luma bit depth=%d\n",
36.             sps->bit_depth_luma);
37.         goto fail;
38.     }
39.     //entropy_coding_mode_flag
40.     //0表示熵编码使用CAVLC, 1表示熵编码使用CABAC
41.     pps->cabac = get_bits1(&h->gb);
42.     pps->pic_order_present = get_bits1(&h->gb);
43.     pps->slice_group_count = get_ue_golomb(&h->gb) + 1;
44.     if (pps->slice_group_count > 1) {
45.         pps->mb_slice_group_map_type = get_ue_golomb(&h->gb);
46.         av_log(h->avctx, AV_LOG_ERROR, "FMO not supported\n");
47.         switch (pps->mb_slice_group_map_type) {
48.             case 0:
49.                 #if 0

```



```

50.         |         tor (i = 0; i <= num_slice_groups_minus1; i++) |         |
51.         |         run_length[i]                                |1 |ue(v) |
52.     #endif
53.         break;
54.     case 2:
55.     #if 0
56.         |         for (i = 0; i < num_slice_groups_minus1; i++) { |         |
57.         |         top_left_mb[i]                                |1 |ue(v) |
58.         |         bottom_right_mb[i]                            |1 |ue(v) |
59.         |         }                                             |         |
60.     #endif
61.         break;
62.     case 3:
63.     case 4:
64.     case 5:
65.     #if 0
66.         |         slice_group_change_direction_flag              |1 |u(1) |
67.         |         slice_group_change_rate_minus1                |1 |ue(v) |
68.     #endif
69.         break;
70.     case 6:
71.     #if 0
72.         |         slice_group_id_cnt_minus1                      |1 |ue(v) |
73.         |         for (i = 0; i <= slice_group_id_cnt_minus1; i++) |         |
74.         |         slice_group_id[i]                             |1 |u(v) |
75.     #endif
76.         break;
77.     }
78. }
79. //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
80. //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
81. //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;
82. //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
83. //“active”中也可以看出来。
84. pps->ref_count[0] = get_ue_golomb(&h->gb) + 1;
85. pps->ref_count[1] = get_ue_golomb(&h->gb) + 1;
86. if (pps->ref_count[0] - 1 > 32 - 1 || pps->ref_count[1] - 1 > 32 - 1) {
87.     av_log(h->avctx, AV_LOG_ERROR, "reference overflow (pps)\n");
88.     goto fail;
89. }
90. //P Slice 是否使用加权预测?
91. pps->weighted_pred = get_bits1(&h->gb);
92. //B Slice 是否使用加权预测?
93. pps->weighted_bipred_idc = get_bits(&h->gb, 2);
94. //QP初始值。读取后需要加26
95. pps->init_qp = get_se_golomb(&h->gb) + 26 + qp_bd_offset;
96. //SP和SI的QP初始值 (没怎么见过这两种帧)
97. pps->init_qs = get_se_golomb(&h->gb) + 26 + qp_bd_offset;
98. pps->chroma_qp_index_offset[0] = get_se_golomb(&h->gb);
99. pps->deblocking_filter_parameters_present = get_bits1(&h->gb);
100. pps->constrained_intra_pred = get_bits1(&h->gb);
101. pps->redundant_pic_cnt_present = get_bits1(&h->gb);
102.
103. pps->transform_8x8_mode = 0;
104. // contents of sps/pps can change even if id doesn't, so reinit
105. h->dequant_coeff_pps = -1;
106. memcpy(pps->scaling_matrix4, h->sps_buffers[pps->sps_id]->scaling_matrix4,
107.         sizeof(pps->scaling_matrix4));
108. memcpy(pps->scaling_matrix8, h->sps_buffers[pps->sps_id]->scaling_matrix8,
109.         sizeof(pps->scaling_matrix8));
110.
111. bits_left = bit_length - get_bits_count(&h->gb);
112. if (bits_left > 0 && more_rbsp_data_in_pps(h, pps)) {
113.     pps->transform_8x8_mode = get_bits1(&h->gb);
114.     decode_scaling_matrices(h, h->sps_buffers[pps->sps_id], pps, 0,
115.                             pps->scaling_matrix4, pps->scaling_matrix8);
116.     // second_chroma_qp_index_offset
117.     pps->chroma_qp_index_offset[1] = get_se_golomb(&h->gb);
118. } else {
119.     pps->chroma_qp_index_offset[1] = pps->chroma_qp_index_offset[0];
120. }
121.
122. build_qp_table(pps, 0, pps->chroma_qp_index_offset[0], sps->bit_depth_luma);
123. build_qp_table(pps, 1, pps->chroma_qp_index_offset[1], sps->bit_depth_luma);
124. if (pps->chroma_qp_index_offset[0] != pps->chroma_qp_index_offset[1])
125.     pps->chroma_qp_diff = 1;
126.
127. if (h->avctx->debug & FF_DEBUG_PICT_INFO) {
128.     av_log(h->avctx, AV_LOG_DEBUG,
129.         "pps:%u sps:%u %s slice_groups:%d ref:%u/%u %s qp:%d/%d/%d/%d %s %s %s %s\n",
130.         pps_id, pps->sps_id,
131.         pps->cabac ? "CABAC" : "CAVLC",
132.         pps->slice_group_count,
133.         pps->ref_count[0], pps->ref_count[1],
134.         pps->weighted_pred ? "weighted" : "",
135.         pps->init_qp, pps->init_qs, pps->chroma_qp_index_offset[0], pps->chroma_qp_index_offset[1],
136.         pps->deblocking_filter_parameters_present ? "LPAR" : "",
137.         pps->constrained_intra_pred ? "CONSTR" : "",
138.         pps->redundant_pic_cnt_present ? "REDU" : "",
139.         pps->transform_8x8_mode ? "8x8DCT" : "");
140. }
141.

```

```
142.     av_free(h->pps_buffers[pps_id]);
143.     h->pps_buffers[pps_id] = pps;
144.     return 0;
145.
146. fail:
147.     av_free(pps);
148.     return -1;
149. }
```

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。

```

1. //SEI补充增强信息单元
2. int ff_h264_decode_sei(H264Context *h)
3. {
4.     while (get_bits_left(&h->gb) > 16 && show_bits(&h->gb, 16)) {
5.         int type = 0;
6.         unsigned size = 0;
7.         unsigned next;
8.         int ret = 0;
9.
10.        do {
11.            if (get_bits_left(&h->gb) < 8)
12.                return AVERROR_INVALIDDATA;
13.            type += show_bits(&h->gb, 8);
14.        } while (get_bits(&h->gb, 8) == 255);
15.
16.        do {
17.            if (get_bits_left(&h->gb) < 8)
18.                return AVERROR_INVALIDDATA;
19.            size += show_bits(&h->gb, 8);
20.        } while (get_bits(&h->gb, 8) == 255);
21.
22.        if (h->avctx->debug&FF_DEBUG_STARTCODE)
23.            av_log(h->avctx, AV_LOG_DEBUG, "SEI %d len:%d\n", type, size);
24.
25.        if (size > get_bits_left(&h->gb) / 8) {
26.            av_log(h->avctx, AV_LOG_ERROR, "SEI type %d size %d truncated at %d\n",
27.                type, 8*size, get_bits_left(&h->gb));
28.            return AVERROR_INVALIDDATA;
29.        }
30.        next = get_bits_count(&h->gb) + 8 * size;
31.
32.        switch (type) {
33.            case SEI_TYPE_PIC_TIMING: // Picture timing SEI
34.                ret = decode_picture_timing(h);
35.                if (ret < 0)
36.                    return ret;
37.                break;
38.            case SEI_TYPE_USER_DATA_ITU_T_T35:
39.                if (decode_user_data_itu_t_t35(h, size) < 0)
40.                    return -1;
41.                break;
42.            //x264的编码参数信息一般都会存储在USER_DATA_UNREGISTERED
43.            //其他种类的SEI见得很少
44.            case SEI_TYPE_USER_DATA_UNREGISTERED:
45.                ret = decode_unregistered_user_data(h, size);
46.                if (ret < 0)
47.                    return ret;
48.                break;
49.            case SEI_TYPE_RECOVERY_POINT:
50.                ret = decode_recovery_point(h);
51.                if (ret < 0)
52.                    return ret;
53.                break;
54.            case SEI_TYPE_BUFFERING_PERIOD:
55.                ret = decode_buffering_period(h);
56.                if (ret < 0)
57.                    return ret;
58.                break;
59.            case SEI_TYPE_FRAME_PACKING:
60.                ret = decode_frame_packing_arrangement(h);
61.                if (ret < 0)
62.                    return ret;
63.                break;
64.            case SEI_TYPE_DISPLAY_ORIENTATION:
65.                ret = decode_display_orientation(h);
66.                if (ret < 0)
67.                    return ret;
68.                break;
69.            default:
70.                av_log(h->avctx, AV_LOG_DEBUG, "unknown SEI type %d\n", type);
71.        }
72.        skip_bits_long(&h->gb, next - get_bits_count(&h->gb));
73.
74.        // FIXME check bits here
75.        align_get_bits(&h->gb);
76.    }
77.
78.    return 0;
79. }

```

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]    
1. //x264的编码参数信息一般都会存储在USER_DATA_UNREGISTERED  
2. static int decode_unregistered_user_data(H264Context *h, int size)  
3. {  
4.     uint8_t user_data[16 + 256];  
5.     int e, build, i;  
6.  
7.     if (size < 16)  
8.         return AVERROR_INVALIDDATA;  
9.  
10.    for (i = 0; i < sizeof(user_data) - 1 && i < size; i++)  
11.        user_data[i] = get_bits(&h->gb, 8);  
12.    //user_data内容示例: x264 core 118  
13.    //int sscanf(const char *buffer, const char *format, [argument ]...);  
14.    //sscanf会从buffer里读进数据，依照format的格式将数据写入到argument里。  
15.    user_data[i] = 0;  
16.    e = sscanf(user_data + 16, "x264 - core %d", &build);  
17.    if (e == 1 && build > 0)  
18.        h->x264_build = build;  
19.    if (e == 1 && build == 1 && !strncmp(user_data+16, "x264 - core 0000", 16))  
20.        h->x264_build = 67;  
21.  
22.    if (h->avctx->debug & FF_DEBUG_BUGS)  
23.        av_log(h->avctx, AV_LOG_DEBUG, "user data: \"%s\\n\"", user_data + 16);  
24.  
25.    for (; i < size; i++)  
26.        skip_bits(&h->gb, 8);  
27.  
28.    return 0;  
29. }
```

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]    
1. case NAL_IDR_SLICE:  
2.     //如果是IDR Slice  
3.     //赋值AVCodecParserContext的key_frame为1  
4.     s->key_frame = 1;  
5.  
6.     h->prev_frame_num = 0;  
7.     h->prev_frame_num_offset = 0;  
8.     h->prev_poc_msb = 0;  
9.     h->prev_poc_lsb = 0;  
10.    /* fall through */  
11.    case NAL_SLICE:  
12.        //获取Slice的一些信息  
13.        //跳过first_mb_in_slice这一字段  
14.        get_ue_golomb_long(&h->gb); // skip first_mb_in_slice  
15.        //获取帧类型 (I,B,P)  
16.        slice_type = get_ue_golomb_31(&h->gb);  
17.        //赋值到AVCodecParserContext的pict_type (外部可以访问到)  
18.        s->pict_type = golomb_to_pict_type[slice_type % 5];  
19.        //关键帧  
20.        if (h->sei_recovery_frame_cnt >= 0) {  
21.            /* key frame, since recovery_frame_cnt is set */  
22.            //赋值AVCodecParserContext的key_frame为1  
23.            s->key_frame = 1;  
24.        }  
25.        //获取 PPS ID  
26.        pps_id = get_ue_golomb(&h->gb);  
27.        if (pps_id >= MAX_PPS_COUNT) {  
28.            av_log(h->avctx, AV_LOG_ERROR,  
29.                "pps_id %u out of range\\n", pps_id);  
30.            return -1;  
31.        }  
32.        if (!h->pps_buffers[pps_id]) {  
33.            av_log(h->avctx, AV_LOG_ERROR,  
34.                "non-existing PPS %u referenced\\n". pps_id);
```

```

35.         return -1;
36.     }
37.     h->pps = *h->pps_buffers[pps_id];
38.     if (!h->sps_buffers[h->pps.sps_id]) {
39.         av_log(h->avctx, AV_LOG_ERROR,
40.             "non-existing SPS %u referenced\n", h->pps.sps_id);
41.         return -1;
42.     }
43.     h->sps = *h->sps_buffers[h->pps.sps_id];
44.     h->frame_num = get_bits(&h->gb, h->sps.log2_max_frame_num);
45.
46.     if (h->sps.ref_frame_count <= 1 && h->pps.ref_count[0] <= 1 && s->pict_type == AV_PICTURE_TYPE_I)
47.         s->key_frame = 1;
48.     //获得“型”和“级”
49.     //赋值到AVCodecContext的profile和level
50.     avctx->profile = ff_h264_get_profile(&h->sps);
51.     avctx->level = h->sps.level_idc;
52.
53.     if (h->sps.frame_mbs_only_flag) {
54.         h->picture_structure = PICT_FRAME;
55.     } else {
56.         if (get_bits1(&h->gb)) { // field_pic_flag
57.             h->picture_structure = PICT_TOP_FIELD + get_bits1(&h->gb); // bottom_field_flag
58.         } else {
59.             h->picture_structure = PICT_FRAME;
60.         }
61.     }
62.
63.     if (h->nal_unit_type == NAL_IDR_SLICE)
64.         get_ue_golomb(&h->gb); /* idr_pic_id */
65.     if (h->sps.poc_type == 0) {
66.         h->poc_lsb = get_bits(&h->gb, h->sps.log2_max_poc_lsb);
67.
68.         if (h->pps.pic_order_present == 1 &&
69.             h->picture_structure == PICT_FRAME)
70.             h->delta_poc_bottom = get_se_golomb(&h->gb);
71.     }
72.
73.     if (h->sps.poc_type == 1 &&
74.         !h->sps.delta_pic_order_always_zero_flag) {
75.         h->delta_poc[0] = get_se_golomb(&h->gb);
76.
77.         if (h->pps.pic_order_present == 1 &&
78.             h->picture_structure == PICT_FRAME)
79.             h->delta_poc[1] = get_se_golomb(&h->gb);
80.     }
81.
82.     /* Decode POC of this picture.
83.      * The prev_ values needed for decoding POC of the next picture are not set here. */
84.     field_poc[0] = field_poc[1] = INT_MAX;
85.     ff_init_poc(h, field_poc, &s->output_picture_number);
86.
87.     /* Continue parsing to check if MMCO RESET is present.
88.      * FIXME: MMCO_RESET could appear in non-first slice.
89.      * Maybe, we should parse all undisposible non-IDR slice of this
90.      * picture until encountering MMCO_RESET in a slice of it. */
91.     if (h->nal_ref_idc && h->nal_unit_type != NAL_IDR_SLICE) {
92.         got_reset = scan_mmco_reset(s);
93.         if (got_reset < 0)
94.             return got_reset;
95.     }
96.
97.     /* Set up the prev_ values for decoding POC of the next picture. */
98.     h->prev_frame_num = got_reset ? 0 : h->frame_num;
99.     h->prev_frame_num_offset = got_reset ? 0 : h->frame_num_offset;
100.    if (h->nal_ref_idc != 0) {
101.        if (!got_reset) {
102.            h->prev_poc_msb = h->poc_msb;
103.            h->prev_poc_lsb = h->poc_lsb;
104.        } else {
105.            h->prev_poc_msb = 0;
106.            h->prev_poc_lsb =
107.                h->picture_structure == PICT_BOTTOM_FIELD ? 0 : field_poc[0];
108.        }
109.    }

```

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>

版权声明：本文为博主原创文章，未经博主允许不得转载。<https://blog.csdn.net/leixiaohua1020/article/details/45001033>

文章标签：

FFmpeg

H.264

解码

源代码

NALU

个人分类：[FFMPEG](#)

所属专栏：[FFmpeg](#)

此PDF由[spygg](#)生成,请尊重原作者版权!!!

我的邮箱:liushidc@163.com