

## FFmpeg的HEVC解码器源代码简单分析：解码器主干部分

2015年06月10日 16:23:08 阅读数：10723

HEVC源代码分析文章列表：

【解码 -libavcodec HEVC 解码器】

FFmpeg的HEVC解码器源代码简单分析：概述

FFmpeg的HEVC解码器源代码简单分析：解析器（Parser）部分

FFmpeg的HEVC解码器源代码简单分析：解码器主干部分

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

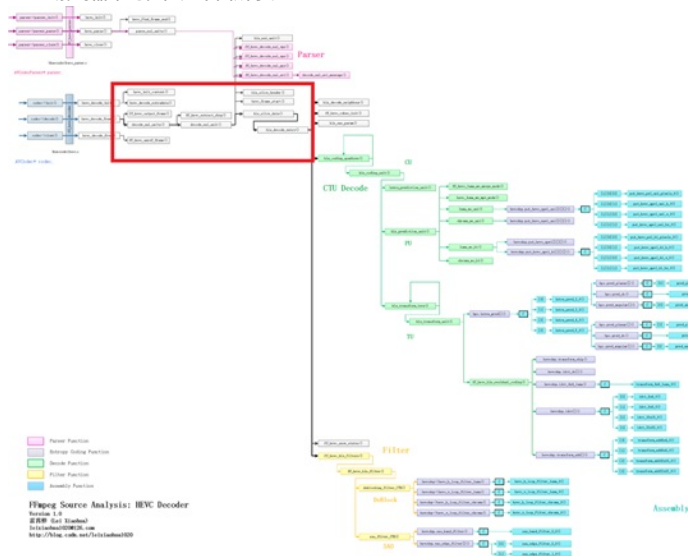
FFmpeg的HEVC解码器源代码简单分析：CTU解码（CTU Decode）部分-TU

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

本文分析FFmpeg的libavcodec中的HEVC解码器的主干部分。“主干部分”是相对于“CTU解码”、“环路滤波”这些细节部分而言的。它包含了HEVC解码器直到hls\_decode\_entry()前面的函数调用关系（hls\_decode\_entry()后面就是HEVC解码器的细节部分，主要包含了“CTU解码”、“环路滤波”2个部分）。

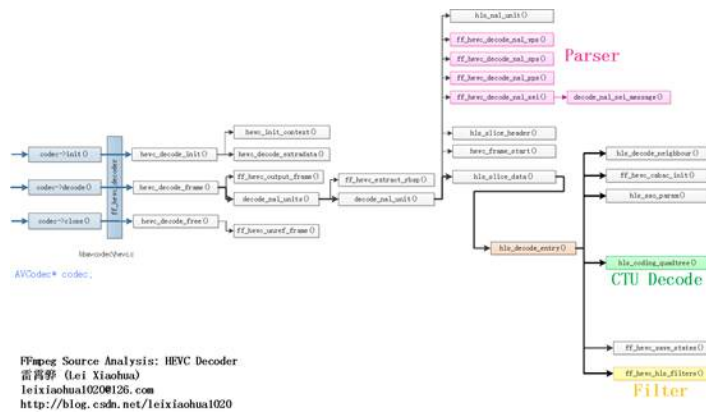
## 函数调用关系图

FFmpeg HEVC解码器主干部分在整个HEVC解码器中的位置如下图所示。



[单击查看更清晰的大图](#)

HEVC解码器主干部分的源代码的调用关系如下图所示。



[单击查看更清晰的大图](#)

从图中可以看出，HEVC解码器初始化函数是hevc\_decode\_init()，解码函数是hevc\_decode\_frame()，关闭函数是hevc\_decode\_free()。其中hevc\_decode\_frame()调用了decode\_nal\_units()进行一帧NALU的解码，decode\_nal\_units()又调用了decode\_nal\_unit()进行一个NALU的解码。

decode\_nal\_unit()一方面调用解析函数ff\_hevc\_decode\_nal\_vps(), ff\_hevc\_decode\_nal\_sps(), ff\_hevc\_decode\_nal\_pps()等对VPS、SPS、PPS进行解析；另一方面调用了hls\_slice\_header()和hls\_slice\_data()对Slice数据进行解码。

hls\_slice\_data()中调用了hls\_decode\_entry()，在其中完成了Slice Data解码的流程。该流程包含了CU、PU、TU解码，环路滤波、SAO滤波等环节。

## ff\_hevc\_decoder

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

```
[cpp]
1. AVCodec ff_hevc_decoder = {
2.     .name = "hevc",
3.     .long_name = NULL_IF_CONFIG_SMALL("HEVC (High Efficiency Video Coding)"),
4.     .type = AVMEDIA_TYPE_VIDEO,
5.     .id = AV_CODEC_ID_HEVC,
6.     .priv_data_size = sizeof(HEVCContext),
7.     .priv_class = &hevc_decoder_class,
8.     .init = hevc_decode_init,
9.     .close = hevc_decode_free,
10.    .decode = hevc_decode_frame,
11.    .flush = hevc_decode_flush,
12.    .update_thread_context = hevc_update_thread_context,
13.    .init_thread_copy = hevc_init_thread_copy,
14.    .capabilities = CODEC_CAP_DR1 | CODEC_CAP_DELAY |
15.        CODEC_CAP_SLICE_THREADS | CODEC_CAP_FRAME_THREADS,
16.    .profiles = NULL_IF_CONFIG_SMALL(profiles),
17.};
```

从源代码可以看出，HEVC解码器初始化函数是hevc\_decode\_init()，解码函数是hevc\_decode\_frame()，关闭函数是hevc\_decode\_free()。

## hevc\_decode\_init()

hevc\_decode\_init()用于初始化HEVC解码器。该函数的定义如下。

```

1. //初始化HEVC解码器
2. static av_cold int hevc_decode_init(AVCodecContext *avctx)
3. {
4.     HEVCContext *s = avctx->priv_data;
5.     int ret;
6.
7.     //初始化CABAC
8.     ff_init_cabac_states();
9.
10.    avctx->internal->allocate_progress = 1;
11.
12.    //为HEVCContext中的变量分配内存空间
13.    ret = hevc_init_context(avctx);
14.    if (ret < 0)
15.        return ret;
16.
17.    s->enable_parallel_tiles = 0;
18.    s->picture_struct = 0;
19.
20.    if(avctx->active_thread_type & FF_THREAD_SLICE)
21.        s->threads_number = avctx->thread_count;
22.    else
23.        s->threads_number = 1;
24.
25.    //如果AVCodecContext中包含extradata, 则解码之
26.    if (avctx->extradata_size > 0 && avctx->extradata) {
27.        ret = hevc_decode_extradata(s);
28.        if (ret < 0) {
29.            hevc_decode_free(avctx);
30.            return ret;
31.        }
32.    }
33.
34.    if((avctx->active_thread_type & FF_THREAD_FRAME) && avctx->thread_count > 1)
35.        s->threads_type = FF_THREAD_FRAME;
36.    else
37.        s->threads_type = FF_THREAD_SLICE;
38.
39.    return 0;
40. }

```

从源代码中可以看出，hevc\_decode\_init()对HEVCContext中的变量做了一些初始化工作。其中调用了函数hevc\_init\_context()用于给HEVCContext中的变量分配内存空间。

## hevc\_init\_context()

hevc\_init\_context()用于给HEVCContext中的变量分配内存空间。该函数的定义如下所示。

[cpp]  

```
1. //为HEVCContext中的变量分配内存空间
2. static av_cold int hevc_init_context(AVCodecContext *avctx)
3. {
4.     HEVCContext *s = avctx->priv_data;
5.     int i;
6.
7.     s->avctx = avctx;
8.
9.     s->HEVCcl = av_mallocz(sizeof(HEVCLocalContext));
10.    if (!s->HEVCcl)
11.        goto fail;
12.    s->HEVCclList[0] = s->HEVCcl;
13.    s->sList[0] = s;
14.
15.    s->cabac_state = av_malloc(HEVC_CONTEXTS);
16.    if (!s->cabac_state)
17.        goto fail;
18.
19.    s->tmp_frame = av_frame_alloc();
20.    if (!s->tmp_frame)
21.        goto fail;
22.
23.    s->output_frame = av_frame_alloc();
24.    if (!s->output_frame)
25.        goto fail;
26.
27.    for (i = 0; i < FF_ARRAY_ELEMS(s->DPB); i++) {
28.        s->DPB[i].frame = av_frame_alloc();
29.        if (!s->DPB[i].frame)
30.            goto fail;
31.        s->DPB[i].tf.f = s->DPB[i].frame;
32.    }
33.
34.    s->max_ra = INT_MAX;
35.
36.    s->md5_ctx = av_md5_alloc();
37.    if (!s->md5_ctx)
38.        goto fail;
39.
40.    ff_bswapdsp_init(&s->bdsp);
41.
42.    s->context_initialized = 1;
43.    s->eos = 0;
44.
45.    return 0;
46.
47. fail:
48.    hevc_decode_free(avctx);
49.    return AVERROR(ENOMEM);
50. }
```

## hevc\_decode\_free()

hevc\_decode\_free()用于关闭HEVC解码器。该函数的定义如下所示。

```

1. //关闭HEVC解码器
2. static av_cold int hevc_decode_free(AVCodecContext *avctx)
3. {
4.     HEVCContext      *s = avctx->priv_data;
5.     int i;
6.
7.     pic_arrays_free(s);
8.
9.     av_freep(&s->md5_ctx);
10.
11.     for(i=0; i < s->nals_allocated; i++) {
12.         av_freep(&s->skipped_bytes_pos_nal[i]);
13.     }
14.     av_freep(&s->skipped_bytes_pos_size_nal);
15.     av_freep(&s->skipped_bytes_nal);
16.     av_freep(&s->skipped_bytes_pos_nal);
17.
18.     av_freep(&s->cabac_state);
19.
20.     av_frame_free(&s->tmp_frame);
21.     av_frame_free(&s->output_frame);
22.
23.     for (i = 0; i < FF_ARRAY_ELEMS(s->DPB); i++) {
24.         ff_hevc_unref_frame(s, &s->DPB[i], ~0);
25.         av_frame_free(&s->DPB[i].frame);
26.     }
27.
28.     for (i = 0; i < FF_ARRAY_ELEMS(s->vps_list); i++)
29.         av_buffer_unref(&s->vps_list[i]);
30.     for (i = 0; i < FF_ARRAY_ELEMS(s->sps_list); i++)
31.         av_buffer_unref(&s->sps_list[i]);
32.     for (i = 0; i < FF_ARRAY_ELEMS(s->pps_list); i++)
33.         av_buffer_unref(&s->pps_list[i]);
34.     s->sps = NULL;
35.     s->pps = NULL;
36.     s->vps = NULL;
37.
38.     av_buffer_unref(&s->current_sps);
39.
40.     av_freep(&s->sh.entry_point_offset);
41.     av_freep(&s->sh.offset);
42.     av_freep(&s->sh.size);
43.
44.     for (i = 1; i < s->threads_number; i++) {
45.         HEVCLocalContext *lc = s->HEVCList[i];
46.         if (lc) {
47.             av_freep(&s->HEVCList[i]);
48.             av_freep(&s->sList[i]);
49.         }
50.     }
51.     if (s->HEVCList == s->HEVCList[0])
52.         s->HEVCList = NULL;
53.     av_freep(&s->HEVCList[0]);
54.
55.     for (i = 0; i < s->nals_allocated; i++)
56.         av_freep(&s->nals[i].rbsp_buffer);
57.     av_freep(&s->nals);
58.     s->nals_allocated = 0;
59.
60.     return 0;
61. }

```

从源代码可以看出，hevc\_decode\_free()释放了HEVCContext中的内存。

## hevc\_decode\_frame()

hevc\_decode\_frame()是HEVC解码器中最关键的函数，用于解码一帧数据。该函数的定义如下所示。

```

1.  /*
2.  * 解码一帧数据
3.  *
4.  * 注释：雷霄骅
5.  * leixiaohua1020@126.com
6.  * http://blog.csdn.net/leixiaohua1020
7.  *
8.  */
9.  static int hevc_decode_frame(AVCodecContext *avctx, void *data, int *got_output,
10.                             AVPacket *avpkt)
11.  {
12.      int ret;
13.      HEVCContext *s = avctx->priv_data;
14.      //没有输入码流的时候，输出解码器中剩余数据
15.      //对应“Flush Decoder”功能
16.      if (!avpkt->size) {
17.          //第3个参数flush取值为1
18.          ret = ff_hevc_output_frame(s, data, 1);
19.          if (ret < 0)
20.              return ret;
21.
22.          *got_output = ret;
23.          return 0;
24.      }
25.
26.      s->ref = NULL;
27.      //解码一帧数据
28.      ret = decode_nal_units(s, avpkt->data, avpkt->size);
29.      if (ret < 0)
30.          return ret;
31.
32.      /* verify the SEI checksum */
33.      if (avctx->err_recognition & AV_EF_CRC_CHECK && s->is_decoded &&
34.          s->is_md5) {
35.          ret = verify_md5(s, s->ref->frame);
36.          if (ret < 0 && avctx->err_recognition & AV_EF_EXPLODE) {
37.              ff_hevc_unref_frame(s, s->ref, ~0);
38.              return ret;
39.          }
40.      }
41.      s->is_md5 = 0;
42.
43.      if (s->is_decoded) {
44.          av_log(avctx, AV_LOG_DEBUG, "Decoded frame with POC %d.\n", s->poc);
45.          s->is_decoded = 0;
46.      }
47.
48.      if (s->output_frame->buf[0]) {
49.          //输出解码后数据
50.          av_frame_move_ref(data, s->output_frame);
51.          *got_output = 1;
52.      }
53.
54.      return avpkt->size;
55.  }

```

从源代码可以看出，hevc\_decode\_frame()根据输入的AVPacket的data是否为NULL分成两个情况：

- (1) AVPacket的data为NULL的时候，代表没有输入码流，这时候直接调用ff\_hevc\_output\_frame()输出解码器中缓存的帧。
- (2) AVPacket的data不为NULL的时候，调用decode\_nal\_units()解码输入的一帧数据的NALU。

下面看一下一帧NALU的解码函数decode\_nal\_units()。

## decode\_nal\_units()

decode\_nal\_units()用于解码一帧NALU。该函数的定义如下所示。

```

1.  //解码一帧数据
2.  static int decode_nal_units(HEVCContext *s, const uint8_t *buf, int length)
3.  {
4.      int i, consumed, ret = 0;
5.
6.      s->ref = NULL;
7.      s->last_eos = s->eos;
8.      s->eos = 0;
9.
10.     /* split the input packet into NAL units, so we know the upper bound on the
11.      * number of slices in the frame */
12.     s->nb_nals = 0;
13.     while (length >= 4) {
14.         HEVCNAL *nal;
15.         int extract_length = 0;
16.
17.         if (s->is_nalff) {
18.             int i;
19.             for (i = 0; i < s->nal_length_size; i++)

```

```

20.         extract_length = (extract_length <= 8) | buf[i];
21.         buf += s->nal_length_size;
22.         length -= s->nal_length_size;
23.
24.         if (extract_length > length) {
25.             av_log(s->avctx, AV_LOG_ERROR, "Invalid NAL unit size.\n");
26.             ret = AERROR_INVALIDDATA;
27.             goto fail;
28.         }
29.     } else {
30.         /* search start code */
31.         //查找起始码0x000001
32.         while (buf[0] != 0 || buf[1] != 0 || buf[2] != 1) {
33.             ++buf;
34.             --length;
35.             if (length < 4) {
36.                 av_log(s->avctx, AV_LOG_ERROR, "No start code is found.\n");
37.                 ret = AERROR_INVALIDDATA;
38.                 goto fail;
39.             }
40.         }
41.         //找到后, 跳过起始码 (3Byte)
42.         buf += 3;
43.         length -= 3;
44.     }
45.
46.     if (!s->is_nalff)
47.         extract_length = length;
48.
49.     if (s->nals_allocated < s->nb_nals + 1) {
50.         int new_size = s->nals_allocated + 1;
51.         HEVCNAL *tmp = av_realloc_array(s->nals, new_size, sizeof(*tmp));
52.         if (!tmp) {
53.             ret = AERROR_ENOMEM;
54.             goto fail;
55.         }
56.         s->nals = tmp;
57.         memset(s->nals + s->nals_allocated, 0,
58.                (new_size - s->nals_allocated) * sizeof(*tmp));
59.         av_realloc_array(&s->skipped_bytes_nal, new_size, sizeof(*s->skipped_bytes_nal));
60.         av_realloc_array(&s->skipped_bytes_pos_size_nal, new_size, sizeof(*s->skipped_bytes_pos_size_nal));
61.         av_realloc_array(&s->skipped_bytes_pos_nal, new_size, sizeof(*s->skipped_bytes_pos_nal));
62.         s->skipped_bytes_pos_size_nal[s->nals_allocated] = 1024; // initial buffer size
63.         s->skipped_bytes_pos_nal[s->nals_allocated] = av_malloc_array(s->skipped_bytes_pos_size_nal[s->nals_allocated], sizeof(*
s->skipped_bytes_pos));
64.         s->nals_allocated = new_size;
65.     }
66.     s->skipped_bytes_pos_size = s->skipped_bytes_pos_size_nal[s->nb_nals];
67.     s->skipped_bytes_pos = s->skipped_bytes_pos_nal[s->nb_nals];
68.     nal = &s->nals[s->nb_nals];
69.
70.     consumed = ff_hevc_extract_rbsp(s, buf, extract_length, nal);
71.
72.     s->skipped_bytes_nal[s->nb_nals] = s->skipped_bytes;
73.     s->skipped_bytes_pos_size_nal[s->nb_nals] = s->skipped_bytes_pos_size;
74.     s->skipped_bytes_pos_nal[s->nb_nals++] = s->skipped_bytes_pos;
75.
76.
77.     if (consumed < 0) {
78.         ret = consumed;
79.         goto fail;
80.     }
81.
82.     ret = init_get_bits8(&s->HEVCcl->gb, nal->data, nal->size);
83.     if (ret < 0)
84.         goto fail;
85.     hls_nal_unit(s);
86.
87.     if (s->nal_unit_type == NAL_EOB_NUT ||
88.         s->nal_unit_type == NAL_EOS_NUT)
89.         s->eos = 1;
90.
91.     buf += consumed;
92.     length -= consumed;
93. }
94.
95. /* parse the NAL units */
96. for (i = 0; i < s->nb_nals; i++) {
97.     int ret;
98.     s->skipped_bytes = s->skipped_bytes_nal[i];
99.     s->skipped_bytes_pos = s->skipped_bytes_pos_nal[i];
100.    //解码NALU
101.    ret = decode_nal_unit(s, s->nals[i].data, s->nals[i].size);
102.    if (ret < 0) {
103.        av_log(s->avctx, AV_LOG_WARNING,
104.               "Error parsing NAL unit #%.d.\n", i);
105.        goto fail;
106.    }
107. }
108.
109. fail:

```

```

110.     if (s->ref && s->threads_type == FF_THREAD_FRAME)
111.         ff_thread_report_progress(&s->ref->tf, INT_MAX, 0);
112.
113.     return ret;
114. }

```

从源代码可以看出，decode\_nal\_units()中又调用了另一个函数decode\_nal\_unit()，两者的名字只相差一个“s”。由此可以看出decode\_nal\_unit()作用是解码一个NALU。

## decode\_nal\_unit()

decode\_nal\_unit()用于解码一个NALU。该函数的定义如下所示。

```

[cpp]
1. //解码一个NALU
2. static int decode_nal_unit(HEVCContext *s, const uint8_t *nal, int length)
3. {
4.     HEVCLocalContext *lc = s->HEVCLc;
5.     GetBitContext *gb = &lc->gb;
6.     int ctb_addr_ts, ret;
7.
8.     ret = init_get_bits8(gb, nal, length);
9.     if (ret < 0)
10.        return ret;
11.
12.     ret = hls_nal_unit(s);
13.     if (ret < 0) {
14.         av_log(s->avctx, AV_LOG_ERROR, "Invalid NAL unit %d, skipping.\n",
15.             s->nal_unit_type);
16.         goto fail;
17.     } else if (!ret)
18.        return 0;
19.
20.     switch (s->nal_unit_type) {
21.     case NAL_VPS:
22.         //解析VPS
23.         ret = ff_hevc_decode_nal_vps(s);
24.         if (ret < 0)
25.             goto fail;
26.         break;
27.     case NAL_SPS:
28.         //解析SPS
29.         ret = ff_hevc_decode_nal_sps(s);
30.         if (ret < 0)
31.             goto fail;
32.         break;
33.     case NAL_PPS:
34.         //解析PPS
35.         ret = ff_hevc_decode_nal_pps(s);
36.         if (ret < 0)
37.             goto fail;
38.         break;
39.     case NAL_SEI_PREFIX:
40.     case NAL_SEI_SUFFIX:
41.         //解析SEI
42.         ret = ff_hevc_decode_nal_sei(s);
43.         if (ret < 0)
44.             goto fail;
45.         break;
46.     case NAL_TRAIL_R:
47.     case NAL_TRAIL_N:
48.     case NAL_TSA_N:
49.     case NAL_TSA_R:
50.     case NAL_STSA_N:
51.     case NAL_STSA_R:
52.     case NAL_BLA_W_LP:
53.     case NAL_BLA_W_RADL:
54.     case NAL_BLA_N_LP:
55.     case NAL_IDR_W_RADL:
56.     case NAL_IDR_N_LP:
57.     case NAL_CRA_NUT:
58.     case NAL_RADL_N:
59.     case NAL_RADL_R:
60.     case NAL_RASL_N:
61.     case NAL_RASL_R:
62.         //解析Slice
63.         //解析Slice Header
64.         ret = hls_slice_header(s);
65.         if (ret < 0)
66.             return ret;
67.
68.         if (s->max_ra == INT_MAX) {
69.             if (s->nal_unit_type == NAL_CRA_NUT || IS_BLA(s)) {
70.                 s->max_ra = s->poc;
71.             } else {
72.                 if (IS_IDR(s))
73.                     s->max_ra = INT_MIN;
74.             }
75.         }

```



```

75.     }
76.
77.     if ((s->nal_unit_type == NAL_RASL_R || s->nal_unit_type == NAL_RASL_N) &&
78.         s->poc <= s->max_ra) {
79.         s->is_decoded = 0;
80.         break;
81.     } else {
82.         if (s->nal_unit_type == NAL_RASL_R && s->poc > s->max_ra)
83.             s->max_ra = INT_MIN;
84.     }
85.
86.     if (s->sh.first_slice_in_pic_flag) {
87.         ret = hevc_frame_start(s);
88.         if (ret < 0)
89.             return ret;
90.     } else if (!s->ref) {
91.         av_log(s->avctx, AV_LOG_ERROR, "First slice in a frame missing.\n");
92.         goto fail;
93.     }
94.
95.     if (s->nal_unit_type != s->first_nal_type) {
96.         av_log(s->avctx, AV_LOG_ERROR,
97.             "Non-matching NAL types of the VCL NALUs: %d %d\n",
98.             s->first_nal_type, s->nal_unit_type);
99.         return AVERROR_INVALIDDATA;
100.    }
101.
102.    if (!s->sh.dependent_slice_segment_flag &&
103.        s->sh.slice_type != I_SLICE) {
104.        ret = ff_hevc_slice_rpl(s);
105.        if (ret < 0) {
106.            av_log(s->avctx, AV_LOG_WARNING,
107.                "Error constructing the reference lists for the current slice.\n");
108.            goto fail;
109.        }
110.    }
111.    //解码 Slice Data
112.    if (s->threads_number > 1 && s->sh.num_entry_point_offsets > 0)
113.        ctb_addr_ts = hls_slice_data_wpp(s, nal, length);
114.    else
115.        ctb_addr_ts = hls_slice_data(s);
116.    if (ctb_addr_ts >= (s->sps->ctb_width * s->sps->ctb_height)) {
117.        s->is_decoded = 1;
118.    }
119.
120.    if (ctb_addr_ts < 0) {
121.        ret = ctb_addr_ts;
122.        goto fail;
123.    }
124.    break;
125. case NAL_EOS_NUT:
126. case NAL_EOB_NUT:
127.     s->seq_decode = (s->seq_decode + 1) & 0xff;
128.     s->max_ra = INT_MAX;
129.     break;
130. case NAL_AUD:
131. case NAL_FD_NUT:
132.     break;
133. default:
134.     av_log(s->avctx, AV_LOG_INFO,
135.         "Skipping NAL unit %d\n", s->nal_unit_type);
136. }
137.
138. return 0;
139. fail:
140. if (s->avctx->err_recognition & AV_EF_EXPLODE)
141.     return ret;
142. return 0;
143. }

```

从源代码可以看出，decode\_nal\_unit()根据不同的NALU类型调用了不同的处理函数。这些处理函数可以分为两类——解析函数和解码函数，如下所示。

(1) 解析函数（获取信息）：

- ff\_hevc\_decode\_nal\_vps()：解析VPS。
- ff\_hevc\_decode\_nal\_sps()：解析SPS。
- ff\_hevc\_decode\_nal\_pps()：解析PPS。
- ff\_hevc\_decode\_nal\_sei()：解析SEI。
- hls\_slice\_header()：解析Slice Header。

(2) 解码函数（解码得到图像）：

- hls\_slice\_data()：解码Slice Data。

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

## hls\_slice\_data()

`hls_slice_data()`用于解码Slice Data。该函数的定义如下所示。

```
1. //解码Slice Data
2. static int hls_slice_data(HEVCContext *s)
3. {
4.     int arg[2];
5.     int ret[2];
6.
7.     arg[0] = 0;
8.     arg[1] = 1;
9.     //解码入口函数
10.    s->avctx->execute(s->avctx, hls_decode_entry, arg, ret , 1, sizeof(int));
11.    return ret[0];
12. }
```

可以看出该函数的源代码很简单，调用了另一个函数`hls_decode_entry()`。

## hls\_decode\_entry()

hls\_decode\_entry()是Slice Data解码的入口函数。该函数的定义如下所示。

```

1.  /*
2.  * 解码入口函数
3.  *
4.  * 注释：雷霄骅
5.  * leixiaohua1020@126.com
6.  * http://blog.csdn.net/leixiaohua1020
7.  *
8.  */
9. static int hls_decode_entry(AVCodecContext *avctx, void *isFilterThread)
10. {
11.     HEVCContext *s = avctx->priv_data;
12.     //CTB尺寸
13.     int ctb_size = 1 << s->sps->log2_ctb_size;
14.     int more_data = 1;
15.     int x_ctb = 0;
16.     int y_ctb = 0;
17.     int ctb_addr_ts = s->pps->ctb_addr_rs_to_ts[s->sh.slice_ctb_addr_rs];
18.
19.     if (!ctb_addr_ts && s->sh.dependent_slice_segment_flag) {
20.         av_log(s->avctx, AV_LOG_ERROR, "Impossible initial tile.\n");
21.         return AVERROR_INVALIDDATA;
22.     }
23.
24.     if (s->sh.dependent_slice_segment_flag) {
25.         int prev_rs = s->pps->ctb_addr_ts_to_rs[ctb_addr_ts - 1];
26.         if (s->tab_slice_address[prev_rs] != s->sh.slice_addr) {
27.             av_log(s->avctx, AV_LOG_ERROR, "Previous slice segment missing\n");
28.             return AVERROR_INVALIDDATA;
29.         }
30.     }
31.
32.     while (more_data && ctb_addr_ts < s->sps->ctb_size) {
33.         int ctb_addr_rs = s->pps->ctb_addr_ts_to_rs[ctb_addr_ts];
34.         //CTB的位置x和y
35.         x_ctb = (ctb_addr_rs % ((s->sps->width + ctb_size - 1) >> s->sps->log2_ctb_size)) << s->sps->log2_ctb_size;
36.         y_ctb = (ctb_addr_rs / ((s->sps->width + ctb_size - 1) >> s->sps->log2_ctb_size)) << s->sps->log2_ctb_size;
37.         //初始化周围的参数
38.         hls_decode_neighbour(s, x_ctb, y_ctb, ctb_addr_ts);
39.         //初始化CABAC
40.         ff_hevc_cabac_init(s, ctb_addr_ts);
41.         //样点自适应补偿参数
42.         hls_sao_param(s, x_ctb >> s->sps->log2_ctb_size, y_ctb >> s->sps->log2_ctb_size);
43.
44.         s->deblock[ctb_addr_rs].beta_offset = s->sh.beta_offset;
45.         s->deblock[ctb_addr_rs].tc_offset = s->sh.tc_offset;
46.         s->filter_slice_edges[ctb_addr_rs] = s->sh.slice_loop_filter_across_slices_enabled_flag;
47.         /*
48.          * CU示意图
49.          *
50.          * 64x64块
51.          *
52.          * 深度d=0
53.          * split_flag=1时候划分为4个32x32
54.          *
55.          * +-----+-----+-----+-----+-----+-----+-----+-----+
56.          * |
57.          * |
58.          * |
59.          * +
60.          * |
61.          * |
62.          * |
63.          * +
64.          * |

```

```

65.  * |
66.  * |
67.  * + |
68.  * |
69.  * |
70.  * |
71.  * + - - - - - + - - - - - +
72.  * |
73.  * |
74.  * |
75.  * + |
76.  * |
77.  * |
78.  * |
79.  * + |
80.  * |
81.  * |
82.  * |
83.  * + |
84.  * |
85.  * |
86.  * |
87.  * + - - - - - + - - - - - + - - - - - +
88.  *
89.  *
90.  * 32x32 块
91.  * 深度d=1
92.  * split_flag=1时候划分为4个16x16
93.  *
94.  * + - - - - - + - - - - - +
95.  * |
96.  * | |
97.  * |
98.  * + | +
99.  * |
100.  * |
101.  * |
102.  * + - - - - - + - - - - - +
103.  * |
104.  * | |
105.  * |
106.  * + | +
107.  * |
108.  * | |
109.  * |
110.  * + - - - - - + - - - - - +
111.  *
112.  *
113.  * 16x16 块
114.  * 深度d=2
115.  * split_flag=1时候划分为4个8x8
116.  *
117.  * + - - - - - +
118.  * | |
119.  * | |
120.  * |
121.  * + - - + - - +
122.  * | |
123.  * | |
124.  * |
125.  * + - - - - - +
126.  *
127.  *
128.  * 8x8块
129.  * 深度d=3
130.  * split_flag=1时候划分为4个4x4
131.  *
132.  * + - - + - - +
133.  * | | |
134.  * + - - + - - +
135.  * | | |
136.  * + - - + - - +
137.  *
138.  */
139. /*
140.  * 解析四叉树结构，并且解码
141.  *
142.  * hls_coding_quadtree(HEVCContext *s, int x0, int y0, int log2_cb_size, int cb_depth)中：
143.  * s：HEVCContext上下文结构体
144.  * x_ctb：CB位置的x坐标
145.  * y_ctb：CB位置的y坐标
146.  * log2_cb_size：CB大小取log2之后的值
147.  * cb_depth：深度
148.  *
149.  */
150. more_data = hls_coding_quadtree(s, x_ctb, y_ctb, s->sps->log2_ctb_size, 0);
151. if (more_data < 0) {
152.     s->tab_slice_address[ctb_addr_rs] = -1;
153.     return more_data;
154. }
155.

```

```

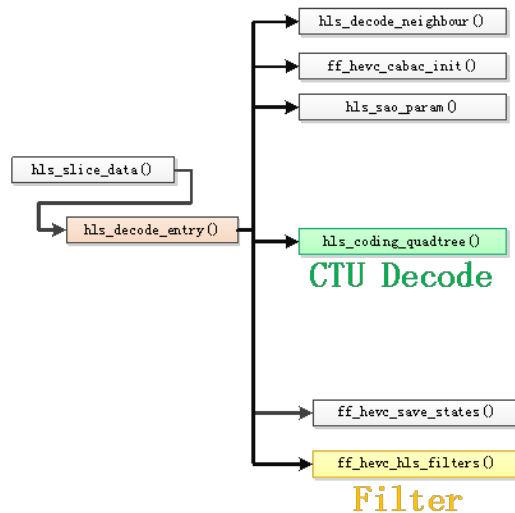
156.
157.     ctb_addr_ts++;
158.     //保存解码信息以供下次使用
159.     ff_hevc_save_states(s, ctb_addr_ts);
160.     //去块效应滤波
161.     ff_hevc_hls_filters(s, x_ctb, y_ctb, ctb_size);
162. }
163.
164. if (x_ctb + ctb_size >= s->sps->width &&
165.     y_ctb + ctb_size >= s->sps->height)
166.     ff_hevc_hls_filter(s, x_ctb, y_ctb, ctb_size);
167.
168. return ctb_addr_ts;
169. }

```

从源代码可以看出，hls\_decode\_entry()以CTB为单位处理输入的视频流。每个CTB的压缩数据经过下面两个基本步骤进行处理：

- (1) 调用hls\_coding\_quadtree()对CTB解码。其中包括了CU、PU、TU的解码。
- (2) 调用ff\_hevc\_hls\_filters()进行滤波。其中包括去块效应滤波和SAO滤波。

hls\_decode\_entry()的函数调用关系如下图所示。后续几篇文章将会对其调用的函数进行分析。



至此，FFmpeg HEVC解码器的主干部分的源代码就分析完毕了。

**雷霄骅**

leixiaohua1020@126.com

<http://blog.csdn.net/leixiaohua1020>

版权声明：本文为博主原创文章，未经博主允许不得转载。 <https://blog.csdn.net/leixiaohua1020/article/details/46412897>

文章标签： FFmpeg CTU libavcodec HEVC 解码

个人分类： FFMPEG

所属专栏： FFMpeg

此PDF由spyyg生成,请尊重原作者版权!!!

我的邮箱:liushidc@163.com