

## 原 FFMpeg源代码简单分析：avcodec\_encode\_video()

2015年03月11日 22:26:14 阅读数：20371

=====

FFmpeg的库函数源代码分析文章列表：

### 【架构图】

[FFmpeg 源代码结构图 - 解码](#)

[FFmpeg 源代码结构图 - 编码](#)

### 【通用】

[FFmpeg 源代码简单分析：av\\_register\\_all\(\)](#)

[FFmpeg 源代码简单分析：avcodec\\_register\\_all\(\)](#)

[FFmpeg 源代码简单分析：内存的分配和释放（av\\_malloc\(\)、av\\_free\(\)等）](#)

[FFmpeg 源代码简单分析：常见结构体的初始化和销毁（AVFormatContext，AVFrame等）](#)

[FFmpeg 源代码简单分析：avio\\_open2\(\)](#)

[FFmpeg 源代码简单分析：av\\_find\\_decoder\(\)和av\\_find\\_encoder\(\)](#)

[FFmpeg 源代码简单分析：avcodec\\_open2\(\)](#)

[FFmpeg 源代码简单分析：avcodec\\_close\(\)](#)

### 【解码】

[图解 FFMPEG 打开媒体的函数 avformat\\_open\\_input](#)

[FFmpeg 源代码简单分析：avformat\\_open\\_input\(\)](#)

[FFmpeg 源代码简单分析：avformat\\_find\\_stream\\_info\(\)](#)

[FFmpeg 源代码简单分析：av\\_read\\_frame\(\)](#)

[FFmpeg 源代码简单分析：avcodec\\_decode\\_video2\(\)](#)

[FFmpeg 源代码简单分析：avformat\\_close\\_input\(\)](#)

### 【编码】

[FFmpeg 源代码简单分析：avformat\\_alloc\\_output\\_context2\(\)](#)

[FFmpeg 源代码简单分析：avformat\\_write\\_header\(\)](#)

[FFmpeg 源代码简单分析：avcodec\\_encode\\_video\(\)](#)

[FFmpeg 源代码简单分析：av\\_write\\_frame\(\)](#)

[FFmpeg 源代码简单分析：av\\_write\\_trailer\(\)](#)

### 【其它】

[FFmpeg 源代码简单分析：日志输出系统（av\\_log\(\)等）](#)

[FFmpeg 源代码简单分析：结构体成员管理系统 -AVClass](#)

[FFmpeg 源代码简单分析：结构体成员管理系统 -AVOption](#)

[FFmpeg 源代码简单分析：libswscale 的 sws\\_getContext\(\)](#)

[FFmpeg 源代码简单分析：libswscale 的 sws\\_scale\(\)](#)

[FFmpeg 源代码简单分析：libavdevice 的 avdevice\\_register\\_all\(\)](#)

[FFmpeg 源代码简单分析：libavdevice 的 gdigrab](#)

### 【脚本】

FFmpeg 源代码简单分析：makefile

FFmpeg 源代码简单分析：configure

【H.264】

FFmpeg 的 H.264 解码器源代码简单分析：概述

本文简单分析FFmpeg的avcodec\_encode\_video2()函数。该函数用于编码一帧视频数据。avcodec\_encode\_video2()函数的声明位于libavcodec\lavcodec.h，如下所示。

```
[cpp]
1.  /**
2.   * Encode a frame of video.
3.   *
4.   * Takes input raw video data from frame and writes the next output packet, if
5.   * available, to avpkt. The output packet does not necessarily contain data for
6.   * the most recent frame, as encoders can delay and reorder input frames
7.   * internally as needed.
8.   *
9.   * @param avctx    codec context
10.  * @param avpkt     output AVPacket.
11.  *                 The user can supply an output buffer by setting
12.  *                 avpkt->data and avpkt->size prior to calling the
13.  *                 function, but if the size of the user-provided data is not
14.  *                 large enough, encoding will fail. All other AVPacket fields
15.  *                 will be reset by the encoder using av_init_packet(). If
16.  *                 avpkt->data is NULL, the encoder will allocate it.
17.  *                 The encoder will set avpkt->size to the size of the
18.  *                 output packet. The returned data (if any) belongs to the
19.  *                 caller, he is responsible for freeing it.
20.  *
21.  *                 If this function fails or produces no output, avpkt will be
22.  *                 freed using av_free_packet() (i.e. avpkt->destruct will be
23.  *                 called to free the user supplied buffer).
24.  * @param[in] frame AVFrame containing the raw video data to be encoded.
25.  *                 May be NULL when flushing an encoder that has the
26.  *                 CODEC_CAP_DELAY capability set.
27.  * @param[out] got_packet_ptr This field is set to 1 by libavcodec if the
28.  *                 output packet is non-empty, and to 0 if it is
29.  *                 empty. If the function returns an error, the
30.  *                 packet can be assumed to be invalid, and the
31.  *                 value of got_packet_ptr is undefined and should
32.  *                 not be used.
33.  * @return          0 on success, negative error code on failure
34.  */
35. int avcodec_encode_video2(AVCodecContext *avctx, AVPacket *avpkt,
36.                           const AVFrame *frame, int *got_packet_ptr);
```

该函数每个参数的含义在注释里面已经写的很清楚了，在这里用中文简述一下：

avctx：编码器的AVCodecContext。

avpkt：编码输出的AVPacket。

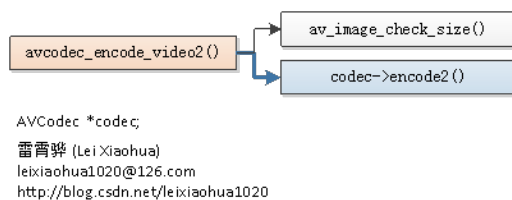
frame：编码输入的AVFrame。

got\_packet\_ptr：成功编码一个AVPacket的时候设置为1。

函数返回0代表编码成功。

## 函数调用关系图

函数的调用关系如下图所示。



## avcodec\_encode\_video2()

avcodec\_encode\_video2()的定义位于libavcodec\utils.c，如下所示。

```



1. int attribute_align_arg avcodec_encode_video2(AVCodecContext *avctx,
2.                                               AVPacket *avpkt,
3.                                               const AVFrame *frame,
4.                                               int *got_packet_ptr)
5. {
6.     int ret;
7.     AVPacket user_pkt = *avpkt;
8.     int needs_realloc = !user_pkt.data;
9.
10.    *got_packet_ptr = 0;
11.
12.    if(CONFIG_FRAME_THREAD_ENCODER &&
13.        avctx->internal->frame_thread_encoder && (avctx->active_thread_type&FF_THREAD_FRAME))
14.        return ff_thread_video_encode_frame(avctx, avpkt, frame, got_packet_ptr);
15.
16.    if ((avctx->flags&CODEC_FLAG_PASS1) && avctx->stats_out)
17.        avctx->stats_out[0] = '\0';
18.
19.    if (!(avctx->codec->capabilities & CODEC_CAP_DELAY) && !frame) {
20.        av_free_packet(avpkt);
21.        av_init_packet(avpkt);
22.        avpkt->size = 0;
23.        return 0;
24.    }
25.    //检查输入
26.    if (av_image_check_size(avctx->width, avctx->height, 0, avctx))
27.        return AVERROR(EINVAL);
28.
29.    av_assert0(avctx->codec->encode2);
30.    //编码
31.    ret = avctx->codec->encode2(avctx, avpkt, frame, got_packet_ptr);
32.    av_assert0(ret <= 0);
33.
34.    if (avpkt->data && avpkt->data == avctx->internal->byte_buffer) {
35.        needs_realloc = 0;
36.        if (user_pkt.data) {
37.            if (user_pkt.size >= avpkt->size) {
38.                memcpy(user_pkt.data, avpkt->data, avpkt->size);
39.            } else {
40.                av_log(avctx, AV_LOG_ERROR, "Provided packet is too small, needs to be %d\n", avpkt->size);
41.                avpkt->size = user_pkt.size;
42.                ret = -1;
43.            }
44.            avpkt->buf = user_pkt.buf;
45.            avpkt->data = user_pkt.data;
46.            #if FF_API_DESTRUCT_PACKET
47.            FF_DISABLE_DEPRECATION_WARNINGS
48.            avpkt->destruct = user_pkt.destruct;
49.            FF_ENABLE_DEPRECATION_WARNINGS
50.            #endif
51.        } else {
52.            if (av_dup_packet(avpkt) < 0) {
53.                ret = AVERROR(ENOMEM);
54.            }
55.        }
56.    }
57.
58.    if (!ret) {
59.        if (!*got_packet_ptr)
60.            avpkt->size = 0;
61.        else if (!(avctx->codec->capabilities & CODEC_CAP_DELAY))
62.            avpkt->pts = avpkt->dts = frame->pts;
63.
64.        if (needs_realloc && avpkt->data) {
65.            ret = av_buffer_realloc(&avpkt->buf, avpkt->size + FF_INPUT_BUFFER_PADDING_SIZE);
66.            if (ret >= 0)
67.                avpkt->data = avpkt->buf->data;
68.        }
69.
70.        avctx->frame_number++;
71.    }
72.
73.    if (ret < 0 || !*got_packet_ptr)
74.        av_free_packet(avpkt);
75.    else
76.        av_packet_merge_side_data(avpkt);
77.
78.    emms_c();
79.    return ret;
80. }

```

从函数的定义可以看出，avcodec\_encode\_video2()首先调用了av\_image\_check\_size()检查设置的宽高参数是否合理，然后调用了AVCodec的encode2()调用具体的解码器。

## av\_image\_check\_size()



av\_image\_check\_size()是一个很简单函数，用于检查图像宽高是否正常，它的定义如下所示。

```
[cpp]    
1. int av_image_check_size(unsigned int w, unsigned int h, int log_offset, void *log_ctx)  
2. {  
3.     ImgUtils imgutils = { &imgutils_class, log_offset, log_ctx };  
4.  
5.     if ((int)w>0 && (int)h>0 && (w+128)*(uint64_t)(h+128) < INT_MAX/8)  
6.         return 0;  
7.  
8.     av_log(&imgutils, AV_LOG_ERROR, "Picture size %ux%u is invalid\n", w, h);  
9.     return AVERROR(EINVAL);  
10. }
```

从代码中可以看出，av\_image\_check\_size()主要是要求图像宽高必须为正数，而且取值不能太大。

## AVCodec->encode2()



AVCodec的encode2()是一个函数指针，指向特定编码器的编码函数。在这里我们以libx264为例，看一下它对应的AVCodec的结构体的定义，如下所示。

```
[cpp]    
1. AVCodec ff_libx264_encoder = {  
2.     .name           = "libx264",  
3.     .long_name      = NULL_IF_CONFIG_SMALL("libx264 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(X264Context),  
7.     .init           = X264_init,  
8.     .encode2        = X264_frame,  
9.     .close          = X264_close,  
10.    .capabilities    = CODEC_CAP_DELAY | CODEC_CAP_AUTO_THREADS,  
11.    .priv_class      = &x264_class,  
12.    .defaults        = x264_defaults,  
13.    .init_static_data = X264_init_static,  
14. };
```

从ff\_libx264\_encoder的定义可以看出，encode2()函数指向的是X264\_frame()函数。

## X264\_frame()

X264\_frame()函数的定义位于libavcodec/libx264.c，如下所示。

```
[cpp]    
1. static int X264_frame(AVCodecContext *ctx, AVPacket *pkt, const AVFrame *frame,  
2.     int *got_packet)  
3. {  
4.     X264Context *x4 = ctx->priv_data;  
5.     x264_nal_t *nal;  
6.     int nnal, i, ret;  
7.     x264_picture_t pic_out = {0};  
8.     AVFrameSideData *side_data;  
9.  
10.    x264_picture_init(&x4->pic );  
11.    x4->pic.img.i_csp = x4->params.i_csp;  
12.    if (x264_bit_depth > 8)  
13.        x4->pic.img.i_csp |= X264_CSP_HIGH_DEPTH;  
14.    x4->pic.img.i_plane = avfmt2_num_planes(ctx->pix_fmt);  
15.  
16.    if (frame) {  
17.        for (i = 0; i < x4->pic.img.i_plane; i++) {  
18.            x4->pic.img.plane[i] = frame->data[i];  
19.            x4->pic.img.i_stride[i] = frame->linesize[i];  
20.        }  
21.  
22.        x4->pic.i_pts = frame->pts;  
23.        x4->pic.i_type =  
24.            frame->pict_type == AV_PICTURE_TYPE_I ? X264_TYPE_KEYFRAME :  
25.            frame->pict_type == AV_PICTURE_TYPE_P ? X264_TYPE_P :  
26.            frame->pict_type == AV_PICTURE_TYPE_B ? X264_TYPE_B :  
27.            X264_TYPE_AUTO;  
28.  
29.        if (x4->avcintra_class < 0) {  
30.            if (x4->params.b_interlaced && x4->params.b_tff != frame->top_field_first) {  
31.                x4->params.b_tff = frame->top_field_first;  
32.                x264_encoder_reconfig(x4->enc, &x4->params);  
33.            }  
34.            if (x4->params.vui.i_sar_height != ctx->sample_aspect_ratio.den ||
```

```

35.     x4->params.vui.i_sar_width != ctx->sample_aspect_ratio.num) {
36.         x4->params.vui.i_sar_height = ctx->sample_aspect_ratio.den;
37.         x4->params.vui.i_sar_width = ctx->sample_aspect_ratio.num;
38.         x264_encoder_reconfig(x4->enc, &x4->params);
39.     }
40.
41.     if (x4->params.rc.i_vbv_buffer_size != ctx->rc_buffer_size / 1000 ||
42.         x4->params.rc.i_vbv_max_bitrate != ctx->rc_max_rate / 1000) {
43.         x4->params.rc.i_vbv_buffer_size = ctx->rc_buffer_size / 1000;
44.         x4->params.rc.i_vbv_max_bitrate = ctx->rc_max_rate / 1000;
45.         x264_encoder_reconfig(x4->enc, &x4->params);
46.     }
47.
48.     if (x4->params.rc.i_rc_method == X264_RC_ABR &&
49.         x4->params.rc.i_bitrate != ctx->bit_rate / 1000) {
50.         x4->params.rc.i_bitrate = ctx->bit_rate / 1000;
51.         x264_encoder_reconfig(x4->enc, &x4->params);
52.     }
53.
54.     if (x4->crf >= 0 &&
55.         x4->params.rc.i_rc_method == X264_RC_CRF &&
56.         x4->params.rc.f_rf_constant != x4->crf) {
57.         x4->params.rc.f_rf_constant = x4->crf;
58.         x264_encoder_reconfig(x4->enc, &x4->params);
59.     }
60.
61.     if (x4->params.rc.i_rc_method == X264_RC_CQP &&
62.         x4->cqp >= 0 &&
63.         x4->params.rc.i_qp_constant != x4->cqp) {
64.         x4->params.rc.i_qp_constant = x4->cqp;
65.         x264_encoder_reconfig(x4->enc, &x4->params);
66.     }
67.
68.     if (x4->crf_max >= 0 &&
69.         x4->params.rc.f_rf_constant_max != x4->crf_max) {
70.         x4->params.rc.f_rf_constant_max = x4->crf_max;
71.         x264_encoder_reconfig(x4->enc, &x4->params);
72.     }
73. }
74.
75. side_data = av_frame_get_side_data(frame, AV_FRAME_DATA_STEREO3D);
76. if (side_data) {
77.     AVStereo3D *stereo = (AVStereo3D *)side_data->data;
78.     int fpa_type;
79.
80.     switch (stereo->type) {
81.     case AV_STEREO3D_CHECKERBOARD:
82.         fpa_type = 0;
83.         break;
84.     case AV_STEREO3D_COLUMNS:
85.         fpa_type = 1;
86.         break;
87.     case AV_STEREO3D_LINES:
88.         fpa_type = 2;
89.         break;
90.     case AV_STEREO3D_SIDEBYSIDE:
91.         fpa_type = 3;
92.         break;
93.     case AV_STEREO3D_TOPBOTTOM:
94.         fpa_type = 4;
95.         break;
96.     case AV_STEREO3D_FRAMESEQUENCE:
97.         fpa_type = 5;
98.         break;
99.     default:
100.        fpa_type = -1;
101.        break;
102.    }
103.
104.    if (fpa_type != x4->params.i_frame_packing) {
105.        x4->params.i_frame_packing = fpa_type;
106.        x264_encoder_reconfig(x4->enc, &x4->params);
107.    }
108. }
109. }
110. do {
111.     if (x264_encoder_encode(x4->enc, &nal, &nnal, frame? &x4->pic: NULL, &pic_out) < 0)
112.         return -1;
113.
114.     ret = encode_nals(ctx, pkt, nal, nnal);
115.     if (ret < 0)
116.         return -1;
117. } while (!ret && !frame && x264_encoder_delayed_frames(x4->enc));
118.
119. pkt->pts = pic_out.i_pts;
120. pkt->dts = pic_out.i_dts;
121.
122. switch (pic_out.i_type) {
123. case X264_TYPE_IDR:
124. case X264_TYPE_I:
125.     ctx->coded_frame->pict_type = AV_PICTURE_TYPE_I;

```

```
126.         break;
127.     case X264_TYPE_P:
128.         ctx->coded_frame->pict_type = AV_PICTURE_TYPE_P;
129.         break;
130.     case X264_TYPE_B:
131.     case X264_TYPE_BREF:
132.         ctx->coded_frame->pict_type = AV_PICTURE_TYPE_B;
133.         break;
134.     }
135.
136.     pkt->flags |= AV_PKT_FLAG_KEY*pic_out.b_keyframe;
137.     if (ret)
138.         ctx->coded_frame->quality = (pic_out.i_qpplus1 - 1) * FF_QP2LAMBDA;
139.
140.     *got_packet = ret;
141.     return 0;
142. }
```

有关X264编码的代码在以后分析X264的时候再进行详细分析。在这里我们可以简单看出该函数中有一个do while循环，其中调用了x264\_encoder\_encode()完成了编码的工作。

## 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/44206485>

文章标签：

FFmpeg

源代码

AVCodec

编码

个人分类：

FFMPEG

所属专栏：

FFmpeg

此PDF由spygg生成,请尊重原作者版权!!!

我的邮箱:liushidc@163.com