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：

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\avcodec.h，如下所示。

[cpp]

```cpp
/**
 * Encode a frame of video.
 *
 * Takes input raw video data from frame and writes the next output packet, if
 * available, to avpkt. The output packet does not necessarily contain data for
 * the most recent frame, as encoders can delay and reorder input frames
 * internally as needed.
 *
 * @param avctx     codec context
 * @param avpkt     output AVPacket.
 *                  The user can supply an output buffer by setting
 *                  avpkt->data and avpkt->size prior to calling the
 *                  function, but if the size of the user-provided data is not
 *                  large enough, encoding will fail. All other AVPacket fields
 *                  will be reset by the encoder using av_init_packet(). If
 *                  avpkt->data is NULL, the encoder will allocate it.
 *                  The encoder will set avpkt->size to the size of the
 *                  output packet. The returned data (if any) belongs to the
 *                  caller, he is responsible for freeing it.
 *
 *                  If this function fails or produces no output, avpkt will be
 *                  freed using av_free_packet() (i.e. avpkt->destruct will be
 *                  called to free the user supplied buffer).
 * @param[in] frame AVFrame containing the raw video data to be encoded.
 *                  May be NULL when flushing an encoder that has the
 *                  CODEC_CAP_DELAY capability set.
 * @param[out] got_packet_ptr This field is set to 1 by libavcodec if the
 *                            output packet is non-empty, and to 0 if it is
 *                            empty. If the function returns an error, the
 *                            packet can be assumed to be invalid, and the
 *                            value of got_packet_ptr is undefined and should
 *                            not be used.
 * @return          0 on success, negative error code on failure
 */
int avcodec_encode_video2(AVCodecContext *avctx, AVPacket *avpkt,
                          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。

[cpp]

```cpp
int attribute_align_arg avcodec_encode_video2(AVCodecContext *avctx,
                                              AVPacket *avpkt,
                                              const AVFrame *frame,
                                              int *got_packet_ptr)
{
    int ret;
    AVPacket user_pkt = *avpkt;
    int needs_realloc = !user_pkt.data;

    *got_packet_ptr = 0;

    if(CONFIG_FRAME_THREAD_ENCODER &&
       avctx->internal->frame_thread_encoder && (avctx->active_thread_type&FF_THREAD_FRAME))
        return ff_thread_video_encode_frame(avctx, avpkt, frame, got_packet_ptr);

    if ((avctx->flags&CODEC_FLAG_PASS1) && avctx->stats_out)
        avctx->stats_out[0] = '\0';

    if (!(avctx->codec->capabilities & CODEC_CAP_DELAY) && !frame) {
        av_free_packet(avpkt);
        av_init_packet(avpkt);
        avpkt->size = 0;
        return 0;
    }
    //检查输入
    if (av_image_check_size(avctx->width, avctx->height, 0, avctx))
        return AVERROR(EINVAL);

    av_assert0(avctx->codec->encode2);
    //编码
    ret = avctx->codec->encode2(avctx, avpkt, frame, got_packet_ptr);
    av_assert0(ret <= 0);

    if (avpkt->data && avpkt->data == avctx->internal->byte_buffer) {
        needs_realloc = 0;
        if (user_pkt.data) {
            if (user_pkt.size >= avpkt->size) {
                memcpy(user_pkt.data, avpkt->data, avpkt->size);
            } else {
                av_log(avctx, AV_LOG_ERROR, "Provided packet is too small, needs to be %d\n", avpkt->size);
                avpkt->size = user_pkt.size;
                ret = -1;
            }
            avpkt->buf      = user_pkt.buf;
            avpkt->data     = user_pkt.data;
#if FF_API_DESTRUCT_PACKET
FF_DISABLE_DEPRECATION_WARNINGS
            avpkt->destruct = user_pkt.destruct;
FF_ENABLE_DEPRECATION_WARNINGS
#endif
        } else {
            if (av_dup_packet(avpkt) < 0) {
                ret = AVERROR(ENOMEM);
            }
        }
    }

    if (!ret) {
        if (!*got_packet_ptr)
            avpkt->size = 0;
        else if (!(avctx->codec->capabilities & CODEC_CAP_DELAY))
            avpkt->pts = avpkt->dts = frame->pts;

        if (needs_realloc && avpkt->data) {
            ret = av_buffer_realloc(&avpkt->buf, avpkt->size + FF_INPUT_BUFFER_PADDING_SIZE);
            if (ret >= 0)
                avpkt->data = avpkt->buf->data;
        }

        avctx->frame_number++;
    }

    if (ret < 0 || !*got_packet_ptr)
        av_free_packet(avpkt);
    else
        av_packet_merge_side_data(avpkt);

    emms_c();
    return ret;
}
```

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()是一个很简单的函数，用于检查图像宽高是否正常，它的定义如下所示。

```cpp
int av_image_check_size(unsigned int w, unsigned int h, int log_offset, void *log_ctx)
{
    ImgUtils imgutils = { &imgutils_class, log_offset, log_ctx };

    if ((int)w>0 && (int)h>0 && (w+128)*(uint64_t)(h+128) < INT_MAX/8)
        return 0;

    av_log(&imgutils, AV_LOG_ERROR, "Picture size %ux%u is invalid\n", w, h);
    return AVERROR(EINVAL);
}
```

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
AVCodec ff_libx264_encoder = {
    .name             = "libx264",
    .long_name        = NULL_IF_CONFIG_SMALL("libx264 H.264 / AVC / MPEG-4 AVC / MPEG-4 part 10"),
    .type             = AVMEDIA_TYPE_VIDEO,
    .id               = AV_CODEC_ID_H264,
    .priv_data_size   = sizeof(X264Context),
    .init             = X264_init,
    .encode2          = X264_frame,
    .close            = X264_close,
    .capabilities     = CODEC_CAP_DELAY | CODEC_CAP_AUTO_THREADS,
    .priv_class       = &x264_class,
    .defaults         = x264_defaults,
    .init_static_data = X264_init_static,
};
```

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
static int X264_frame(AVCodecContext *ctx, AVPacket *pkt, const AVFrame *frame,
                      int *got_packet)
{
    X264Context *x4 = ctx->priv_data;
    x264_nal_t *nal;
    int nnal, i, ret;
    x264_picture_t pic_out = {0};
    AVFrameSideData *side_data;

    x264_picture_init( &x4->pic );
    x4->pic.img.i_csp   = x4->params.i_csp;
    if (x264_bit_depth > 8)
        x4->pic.img.i_csp |= X264_CSP_HIGH_DEPTH;
    x4->pic.img.i_plane = avfmt2_num_planes(ctx->pix_fmt);

    if (frame) {
        for (i = 0; i < x4->pic.img.i_plane; i++) {
            x4->pic.img.plane[i]    = frame->data[i];
            x4->pic.img.i_stride[i] = frame->linesize[i];
        }

        x4->pic.i_pts  = frame->pts;
        x4->pic.i_type =
            frame->pict_type == AV_PICTURE_TYPE_I ? X264_TYPE_KEYFRAME :
            frame->pict_type == AV_PICTURE_TYPE_P ? X264_TYPE_P :
            frame->pict_type == AV_PICTURE_TYPE_B ? X264_TYPE_B :
                                            X264_TYPE_AUTO;

        if (x4->avcintra_class < 0) {
        if (x4->params.b_interlaced && x4->params.b_tff != frame->top_field_first) {
            x4->params.b_tff = frame->top_field_first;
            x264_encoder_reconfig(x4->enc, &x4->params);
        }
        if (x4->params.vui.i_sar_height != ctx->sample_aspect_ratio.den ||
```

```c
        if (x4->params.vui.i_sar_height != ctx->sample_aspect_ratio.den ||
            x4->params.vui.i_sar_width  != ctx->sample_aspect_ratio.num) {
            x4->params.vui.i_sar_height = ctx->sample_aspect_ratio.den;
            x4->params.vui.i_sar_width  = ctx->sample_aspect_ratio.num;
            x264_encoder_reconfig(x4->enc, &x4->params);
        }

        if (x4->params.rc.i_vbv_buffer_size != ctx->rc_buffer_size / 1000 ||
            x4->params.rc.i_vbv_max_bitrate != ctx->rc_max_rate    / 1000) {
            x4->params.rc.i_vbv_buffer_size = ctx->rc_buffer_size / 1000;
            x4->params.rc.i_vbv_max_bitrate = ctx->rc_max_rate    / 1000;
            x264_encoder_reconfig(x4->enc, &x4->params);
        }

        if (x4->params.rc.i_rc_method == X264_RC_ABR &&
            x4->params.rc.i_bitrate != ctx->bit_rate / 1000) {
            x4->params.rc.i_bitrate = ctx->bit_rate / 1000;
            x264_encoder_reconfig(x4->enc, &x4->params);
        }

        if (x4->crf >= 0 &&
            x4->params.rc.i_rc_method == X264_RC_CRF &&
            x4->params.rc.f_rf_constant != x4->crf) {
            x4->params.rc.f_rf_constant = x4->crf;
            x264_encoder_reconfig(x4->enc, &x4->params);
        }

        if (x4->params.rc.i_rc_method == X264_RC_CQP &&
            x4->cqp >= 0 &&
            x4->params.rc.i_qp_constant != x4->cqp) {
            x4->params.rc.i_qp_constant = x4->cqp;
            x264_encoder_reconfig(x4->enc, &x4->params);
        }

        if (x4->crf_max >= 0 &&
            x4->params.rc.f_rf_constant_max != x4->crf_max) {
            x4->params.rc.f_rf_constant_max = x4->crf_max;
            x264_encoder_reconfig(x4->enc, &x4->params);
        }
        }

        side_data = av_frame_get_side_data(frame, AV_FRAME_DATA_STEREO3D);
        if (side_data) {
            AVStereo3D *stereo = (AVStereo3D *)side_data->data;
            int fpa_type;

            switch (stereo->type) {
            case AV_STEREO3D_CHECKERBOARD:
                fpa_type = 0;
                break;
            case AV_STEREO3D_COLUMNS:
                fpa_type = 1;
                break;
            case AV_STEREO3D_LINES:
                fpa_type = 2;
                break;
            case AV_STEREO3D_SIDEBYSIDE:
                fpa_type = 3;
                break;
            case AV_STEREO3D_TOPBOTTOM:
                fpa_type = 4;
                break;
            case AV_STEREO3D_FRAMESEQUENCE:
                fpa_type = 5;
                break;
            default:
                fpa_type = -1;
                break;
            }

            if (fpa_type != x4->params.i_frame_packing) {
                x4->params.i_frame_packing = fpa_type;
                x264_encoder_reconfig(x4->enc, &x4->params);
            }
        }
    }
    do {
        if (x264_encoder_encode(x4->enc, &nal, &nnal, frame? &x4->pic: NULL, &pic_out) < 0)
            return -1;

        ret = encode_nals(ctx, pkt, nal, nnal);
        if (ret < 0)
            return -1;
    } while (!ret && !frame && x264_encoder_delayed_frames(x4->enc));

    pkt->pts = pic_out.i_pts;
    pkt->dts = pic_out.i_dts;

    switch (pic_out.i_type) {
    case X264_TYPE_IDR:
    case X264_TYPE_I:
        ctx->coded_frame->pict_type = AV_PICTURE_TYPE_I;
        break;
    case X264_TYPE_P:
        ctx->coded_frame->pict_type = AV_PICTURE_TYPE_P;
        break;
    case X264_TYPE_B:
    case X264_TYPE_BREF:
        ctx->coded_frame->pict_type = AV_PICTURE_TYPE_B;
        break;
    }

    pkt->flags |= AV_PKT_FLAG_KEY*pic_out.b_keyframe;
    if (ret)
        ctx->coded_frame->quality = (pic_out.i_qpplus1 - 1) * FF_QP2LAMBDA;

    *got_packet = ret;
    return 0;
}
```

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