原 ffmpeg 源代码简单分析 ： avcodec_decode_video2()

2013年10月13日 16:07:23 阅读数：54723

==========

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

==========

ffmpeg中的avcodec_decode_video2()的作用是解码一帧视频数据。输入一个压缩编码的结构体AVPacket，输出一个解码后的结构体AVFrame。该函数的声明位于libavcodec\avcodec.h，如下所示。

```cpp
/**
 * Decode the video frame of size avpkt->size from avpkt->data into picture.
 * Some decoders may support multiple frames in a single AVPacket, such
 * decoders would then just decode the first frame.
 *
 * @warning The input buffer must be FF_INPUT_BUFFER_PADDING_SIZE larger than
 * the actual read bytes because some optimized bitstream readers read 32 or 64
 * bits at once and could read over the end.
 *
 * @warning The end of the input buffer buf should be set to 0 to ensure that
 * no overreading happens for damaged MPEG streams.
 *
 * @note Codecs which have the CODEC_CAP_DELAY capability set have a delay
 * between input and output, these need to be fed with avpkt->data=NULL,
 * avpkt->size=0 at the end to return the remaining frames.
 *
 * @param avctx the codec context
 * @param[out] picture The AVFrame in which the decoded video frame will be stored.
 *             Use av_frame_alloc() to get an AVFrame. The codec will
 *             allocate memory for the actual bitmap by calling the
 *             AVCodecContext.get_buffer2() callback.
 *             When AVCodecContext.refcounted_frames is set to 1, the frame is
 *             reference counted and the returned reference belongs to the
 *             caller. The caller must release the frame using av_frame_unref()
 *             when the frame is no longer needed. The caller may safely write
 *             to the frame if av_frame_is_writable() returns 1.
 *             When AVCodecContext.refcounted_frames is set to 0, the returned
 *             reference belongs to the decoder and is valid only until the
 *             next call to this function or until closing or flushing the
 *             decoder. The caller may not write to it.
 *
 * @param[in] avpkt The input AVPacket containing the input buffer.
 *            You can create such packet with av_init_packet() and by then setting
 *            data and size, some decoders might in addition need other fields like
 *            flags&AV_PKT_FLAG_KEY. All decoders are designed to use the least
 *            fields possible.
 * @param[in,out] got_picture_ptr Zero if no frame could be decompressed, otherwise, it is nonzero.
 * @return On error a negative value is returned, otherwise the number of bytes
 * used or zero if no frame could be decompressed.
 */
int avcodec_decode_video2(AVCodecContext *avctx, AVFrame *picture,
                          int *got_picture_ptr,
                          const AVPacket *avpkt);
```

查看源代码之后发现，这个函数竟然十分的简单，源代码位于libavcodec\utils.c，如下所示：

```cpp
int attribute_align_arg avcodec_decode_video2(AVCodecContext *avctx, AVFrame *picture,
                                              int *got_picture_ptr,
                                              const AVPacket *avpkt)
{
    AVCodecInternal *avci = avctx->internal;
    int ret;
    // copy to ensure we do not change avpkt
    AVPacket tmp = *avpkt;

    if (!avctx->codec)
        return AVERROR(EINVAL);
    //检查是不是视频（非音频）
    if (avctx->codec->type != AVMEDIA_TYPE_VIDEO) {
        av_log(avctx, AV_LOG_ERROR, "Invalid media type for video\n");
        return AVERROR(EINVAL);
    }

    *got_picture_ptr = 0;
    //检查宽、高设置是否正确
    if ((avctx->coded_width || avctx->coded_height) && av_image_check_size(avctx->coded_width, avctx->coded_height, 0, avctx))
        return AVERROR(EINVAL);
```

```
    av_frame_unref(picture);

    if ((avctx->codec->capabilities & CODEC_CAP_DELAY) || avpkt->size || (avctx->active_thread_type & FF_THREAD_FRAME)) {
        int did_split = av_packet_split_side_data(&tmp);
        ret = apply_param_change(avctx, &tmp);
        if (ret < 0) {
            av_log(avctx, AV_LOG_ERROR, "Error applying parameter changes.\n");
            if (avctx->err_recognition & AV_EF_EXPLODE)
                goto fail;
        }

        avctx->internal->pkt = &tmp;
        if (HAVE_THREADS && avctx->active_thread_type & FF_THREAD_FRAME)
            ret = ff_thread_decode_frame(avctx, picture, got_picture_ptr,
                                         &tmp);
        else {
            //最关键的解码函数
            ret = avctx->codec->decode(avctx, picture, got_picture_ptr,
                                       &tmp);
            //设置pkt_dts字段的值
            picture->pkt_dts = avpkt->dts;

            if(!avctx->has_b_frames){
                av_frame_set_pkt_pos(picture, avpkt->pos);
            }
            //FIXME these should be under if(!avctx->has_b_frames)
            /* get_buffer is supposed to set frame parameters */
            if (!(avctx->codec->capabilities & CODEC_CAP_DR1)) {
                //对一些字段进行赋值
                if (!picture->sample_aspect_ratio.num)    picture->sample_aspect_ratio = avctx->sample_aspect_ratio;
                if (!picture->width)                      picture->width               = avctx->width;
                if (!picture->height)                     picture->height              = avctx->height;
                if (picture->format == AV_PIX_FMT_NONE)   picture->format              = avctx->pix_fmt;
            }
        }
        add_metadata_from_side_data(avctx, picture);

fail:
        emms_c(); //needed to avoid an emms_c() call before every return;

        avctx->internal->pkt = NULL;
        if (did_split) {
            av_packet_free_side_data(&tmp);
            if(ret == tmp.size)
                ret = avpkt->size;
        }

        if (*got_picture_ptr) {
            if (!avctx->refcounted_frames) {
                int err = unrefcount_frame(avci, picture);
                if (err < 0)
                    return err;
            }

            avctx->frame_number++;
            av_frame_set_best_effort_timestamp(picture,
                                               guess_correct_pts(avctx,
                                                                 picture->pkt_pts,
                                                                 picture->pkt_dts));
        } else
            av_frame_unref(picture);
    } else
        ret = 0;

    /* many decoders assign whole AVFrames, thus overwriting extended_data;
     * make sure it's set correctly */
    av_assert0(!picture->extended_data || picture->extended_data == picture->data);

#if FF_API_AVCTX_TIMEBASE
    if (avctx->framerate.num > 0 && avctx->framerate.den > 0)
        avctx->time_base = av_inv_q(av_mul_q(avctx->framerate, (AVRational){avctx->ticks_per_frame, 1}));
#endif

    return ret;
}
```

从代码中可以看出，avcodec_decode_video2()主要做了以下几个方面的工作：

（1）对输入的字段进行了一系列的检查工作：例如宽高是否正确，输入是否为视频等等。

（2）通过ret = avctx->codec->decode(avctx, picture, got_picture_ptr,&tmp)这句代码，调用了相应AVCodec的decode()函数，完成了解码操作。

（3）对得到的AVFrame的一些字段进行了赋值，例如宽高、像素格式等等。

其中第二部是关键的一步，它调用了AVCodec的decode()方法完成了解码。AVCodec的decode()方法是一个函数指针，指向了具体解码器的解码函数。在这里我们以H.264解码器为例，看一下解码的实现过程。H.264解码器对应的AVCodec的定义位于libavcodec\h264.c，如下所示。

```cpp
AVCodec ff_h264_decoder = {
    .name                  = "h264",
    .long_name             = NULL_IF_CONFIG_SMALL("H.264 / AVC / MPEG-4 AVC / MPEG-4 part 10"),
    .type                  = AVMEDIA_TYPE_VIDEO,
    .id                    = AV_CODEC_ID_H264,
    .priv_data_size        = sizeof(H264Context),
    .init                  = ff_h264_decode_init,
    .close                 = h264_decode_end,
    .decode                = h264_decode_frame,
    .capabilities          = /*CODEC_CAP_DRAW_HORIZ_BAND |*/ CODEC_CAP_DR1 |
                             CODEC_CAP_DELAY | CODEC_CAP_SLICE_THREADS |
                             CODEC_CAP_FRAME_THREADS,
    .flush                 = flush_dpb,
    .init_thread_copy      = ONLY_IF_THREADS_ENABLED(decode_init_thread_copy),
    .update_thread_context = ONLY_IF_THREADS_ENABLED(ff_h264_update_thread_context),
    .profiles              = NULL_IF_CONFIG_SMALL(profiles),
    .priv_class            = &h264_class,
};
```

从ff_h264_decoder的定义可以看出，decode()指向了h264_decode_frame()函数。继续看一下h264_decode_frame()函数的定义，如下所示。

```cpp
static int h264_decode_frame(AVCodecContext *avctx, void *data,
                             int *got_frame, AVPacket *avpkt)
{
    const uint8_t *buf = avpkt->data;
    int buf_size       = avpkt->size;
    H264Context *h     = avctx->priv_data;
    AVFrame *pict      = data;
    int buf_index      = 0;
    H264Picture *out;
    int i, out_idx;
    int ret;

    h->flags = avctx->flags;
    /* reset data partitioning here, to ensure GetBitContexts from previous
     * packets do not get used. */
    h->data_partitioning = 0;

    /* end of stream, output what is still in the buffers */
    if (buf_size == 0) {
 out:

        h->cur_pic_ptr = NULL;
        h->first_field = 0;

        // FIXME factorize this with the output code below
        out     = h->delayed_pic[0];
        out_idx = 0;
        for (i = 1;
             h->delayed_pic[i] &&
             !h->delayed_pic[i]->f.key_frame &&
             !h->delayed_pic[i]->mmco_reset;
             i++)
            if (h->delayed_pic[i]->poc < out->poc) {
                out     = h->delayed_pic[i];
                out_idx = i;
            }

        for (i = out_idx; h->delayed_pic[i]; i++)
            h->delayed_pic[i] = h->delayed_pic[i + 1];

        if (out) {
            out->reference &= ~DELAYED_PIC_REF;
            ret = output_frame(h, pict, out);
            if (ret < 0)
                return ret;
            *got_frame = 1;
        }

        return buf_index;
    }
    if (h->is_avc && av_packet_get_side_data(avpkt, AV_PKT_DATA_NEW_EXTRADATA, NULL)) {
        int side_size;
        uint8_t *side = av_packet_get_side_data(avpkt, AV_PKT_DATA_NEW_EXTRADATA, &side_size);
        if (is_extra(side, side_size))
            ff_h264_decode_extradata(h, side, side_size);
    }
    if(h->is_avc && buf_size >= 9 && buf[0]==1 && buf[2]==0 && (buf[4]&0xFC)==0xFC && (buf[5]&0x1F) && buf[8]==0x67){
        if (is_extra(buf, buf_size))
            return ff_h264_decode_extradata(h, buf, buf_size);
    }
    //H.264解码
    buf_index = decode_nal_units(h, buf, buf_size, 0);
    if (buf_index < 0)
        return AVERROR_INVALIDDATA;

    if (!h->cur_pic_ptr && h->nal_unit_type == NAL_END_SEQUENCE) {
```

```
    if (!h->cur_pic_ptr && h->nal_unit_type == NAL_END_SEQUENCE) {
        av_assert0(buf_index <= buf_size);
        goto out;
    }

    if (!(avctx->flags2 & CODEC_FLAG2_CHUNKS) && !h->cur_pic_ptr) {
        if (avctx->skip_frame >= AVDISCARD_NONREF ||
            buf_size >= 4 && !memcmp("Q264", buf, 4))
            return buf_size;
        av_log(avctx, AV_LOG_ERROR, "no frame!\n");
        return AVERROR_INVALIDDATA;
    }

    if (!(avctx->flags2 & CODEC_FLAG2_CHUNKS) ||
        (h->mb_y >= h->mb_height && h->mb_height)) {
        if (avctx->flags2 & CODEC_FLAG2_CHUNKS)
            decode_postinit(h, 1);

        ff_h264_field_end(h, 0);

        /* Wait for second field. */
        *got_frame = 0;
        if (h->next_output_pic && (
                                   h->next_output_pic->recovered)) {
            if (!h->next_output_pic->recovered)
                h->next_output_pic->f.flags |= AV_FRAME_FLAG_CORRUPT;

            ret = output_frame(h, pict, h->next_output_pic);
            if (ret < 0)
                return ret;
            *got_frame = 1;
            if (CONFIG_MPEGVIDEO) {
                ff_print_debug_info2(h->avctx, pict, h->er.mbskip_table,
                                     h->next_output_pic->mb_type,
                                     h->next_output_pic->qscale_table,
                                     h->next_output_pic->motion_val,
                                     &h->low_delay,
                                     h->mb_width, h->mb_height, h->mb_stride, 1);
            }
        }
    }

    assert(pict->buf[0] || !*got_frame);

    return get_consumed_bytes(buf_index, buf_size);
}
```

从h264_decode_frame()的定义可以看出，它调用了decode_nal_units()完成了具体的H.264解码工作。有关H.264解码就不在详细分析了。

文章标签： ffmpeg 源代码简单 avcodec_decode_video

个人分类： FFMPEG

所属专栏： 开源多媒体项目源代码分析 FFmpeg