（解码）4.avcodec_decode_video2()

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

```
int avcodec_decode_video2(AVCodecContext *avctx, AVFrame *picture,
                         int *got_picture_ptr,
                         const AVPacket *avpkt);
```

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
（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

```
AVCodec ff_h264_decoder = {
    .name                  = "h264",
    .long_name             = NULL_IF_CONFIG_SMALL("H.264 / AVC / MPEG-4 AVC / MPEG-4 par
    .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。

```
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_
        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 && (bu
        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) {
        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。