FFmpeg源代码简单分析：av_write_frame()

2015年03月11日 16:03:35　阅读数：26385

==

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

===

打算写两篇文章简单分析FFmpeg的写文件用到的3个函数avformat_write_header()，av_write_frame()以及av_write_trailer()。上篇文章已经分析了avformat_write_header()，这篇文章继续分析av_write_frame()。

av_write_frame()用于输出一帧视音频数据，它的声明位于libavformat\avformat.h，如下所示。

[cpp]

```
/**
 * Write a packet to an output media file.
 *
 * This function passes the packet directly to the muxer, without any buffering
 * or reordering. The caller is responsible for correctly interleaving the
 * packets if the format requires it. Callers that want libavformat to handle
 * the interleaving should call av_interleaved_write_frame() instead of this
 * function.
 *
 * @param s media file handle
 * @param pkt The packet containing the data to be written. Note that unlike
 *            av_interleaved_write_frame(), this function does not take
 *            ownership of the packet passed to it (though some muxers may make
 *            an internal reference to the input packet).
 *            <br>
 *            This parameter can be NULL (at any time, not just at the end), in
 *            order to immediately flush data buffered within the muxer, for
 *            muxers that buffer up data internally before writing it to the
 *            output.
 *            <br>
 *            Packet's @ref AVPacket.stream_index "stream_index" field must be
 *            set to the index of the corresponding stream in @ref
 *            AVFormatContext.streams "s->streams". It is very strongly
 *            recommended that timing information (@ref AVPacket.pts "pts", @ref
 *            AVPacket.dts "dts", @ref AVPacket.duration "duration") is set to
 *            correct values.
 * @return < 0 on error, = 0 if OK, 1 if flushed and there is no more data to flush
 *
 * @see av_interleaved_write_frame()
 */
int av_write_frame(AVFormatContext *s, AVPacket *pkt);
```

简单解释一下它的参数的含义：

s：用于输出的AVFormatContext。

pkt：等待输出的AVPacket。

函数正常执行后返回值等于0。

这个函数最典型的例子可以参考：

最简单的基于FFMPEG的视频编码器（YUV编码为H.264）

函数调用关系图

av_write_frame()的调用关系如下图所示。

av_write_frame()

av_write_frame()的定义位于libavformat\mux.c，如下所示。

[cpp]

```cpp
int av_write_frame(AVFormatContext *s, AVPacket *pkt)
{
    int ret;

    ret = check_packet(s, pkt);
    if (ret < 0)
        return ret;
    //Packet为NULL，Flush Encoder
    if (!pkt) {
        if (s->oformat->flags & AVFMT_ALLOW_FLUSH) {
            ret = s->oformat->write_packet(s, NULL);
            if (s->flush_packets && s->pb && s->pb->error >= 0 && s->flags & AVFMT_FLAG_FLUSH_PACKETS)
                avio_flush(s->pb);
            if (ret >= 0 && s->pb && s->pb->error < 0)
                ret = s->pb->error;
            return ret;
        }
        return 1;
    }

    ret = compute_pkt_fields2(s, s->streams[pkt->stream_index], pkt);

    if (ret < 0 && !(s->oformat->flags & AVFMT_NOTIMESTAMPS))
        return ret;
    //写入
    ret = write_packet(s, pkt);
    if (ret >= 0 && s->pb && s->pb->error < 0)
        ret = s->pb->error;

    if (ret >= 0)
        s->streams[pkt->stream_index]->nb_frames++;
    return ret;
}
```

从源代码可以看出，av_write_frame()主要完成了以下几步工作：
（1）调用check_packet()做一些简单的检测
（2）调用compute_pkt_fields2()设置AVPacket的一些属性值
（3）调用write_packet()写入数据
下面分别看一下这几个函数功能。

check_packet()

check_packet()定义位于libavformat\mux.c，如下所示。

[cpp]

```cpp
static int check_packet(AVFormatContext *s, AVPacket *pkt)
{
    if (!pkt)
        return 0;

    if (pkt->stream_index < 0 || pkt->stream_index >= s->nb_streams) {
        av_log(s, AV_LOG_ERROR, "Invalid packet stream index: %d\n",
               pkt->stream_index);
        return AVERROR(EINVAL);
    }

    if (s->streams[pkt->stream_index]->codec->codec_type == AVMEDIA_TYPE_ATTACHMENT) {
        av_log(s, AV_LOG_ERROR, "Received a packet for an attachment stream.\n");
        return AVERROR(EINVAL);
    }

    return 0;
}
```

从代码中可以看出，check_packet()的功能比较简单：首先检查一下输入的AVPacket是否为空，如果为空，则是直接返回；然后检查一下AVPacket的stream_index（标记了该AVPacket所属的AVStream）设置是否正常，如果为负数或者大于AVStream的个数，则返回错误信息；最后检查AVPacket所属的AVStream是否属于attachment stream，这个地方没见过，目前还没有研究。

compute_pkt_fields2()

compute_pkt_fields2()函数的定义位于libavformat\mux.c，如下所示。

[cpp]

```cpp
//FIXME merge with compute_pkt_fields
static int compute_pkt_fields2(AVFormatContext *s, AVStream *st, AVPacket *pkt)
{
    int delay = FFMAX(st->codec->has_b_frames, st->codec->max_b_frames > 0);
    int num, den, i;
    int frame_size;

    av_dlog(s, "compute_pkt_fields2: pts:%s dts:%s cur_dts:%s b:%d size:%d st:%d\n",
            av_ts2str(pkt->pts), av_ts2str(pkt->dts), av_ts2str(st->cur_dts), delay, pkt->size, pkt->stream_index);

    if (pkt->duration < 0 && st->codec->codec_type != AVMEDIA_TYPE_SUBTITLE) {
        av_log(s, AV_LOG_WARNING, "Packet with invalid duration %d in stream %d\n",
               pkt->duration, pkt->stream_index);
        pkt->duration = 0;
    }

    /* duration field */
    if (pkt->duration == 0) {
        ff_compute_frame_duration(s, &num, &den, st, NULL, pkt);
        if (den && num) {
            pkt->duration = av_rescale(1, num * (int64_t)st->time_base.den * st->codec->ticks_per_frame, den * (int64_t)st->time_base.num);
        }
    }

    if (pkt->pts == AV_NOPTS_VALUE && pkt->dts != AV_NOPTS_VALUE && delay == 0)
        pkt->pts = pkt->dts;

    //XXX/FIXME this is a temporary hack until all encoders output pts
    if ((pkt->pts == 0 || pkt->pts == AV_NOPTS_VALUE) && pkt->dts == AV_NOPTS_VALUE && !delay) {
        static int warned;
        if (!warned) {
            av_log(s, AV_LOG_WARNING, "Encoder did not produce proper pts, making some up.\n");
            warned = 1;
        }
        pkt->dts =
//        pkt->pts= st->cur_dts;
            pkt->pts = st->pts.val;
    }

    //calculate dts from pts
    if (pkt->pts != AV_NOPTS_VALUE && pkt->dts == AV_NOPTS_VALUE && delay <= MAX_REORDER_DELAY) {
        st->pts_buffer[0] = pkt->pts;
        for (i = 1; i < delay + 1 && st->pts_buffer[i] == AV_NOPTS_VALUE; i++)
            st->pts_buffer[i] = pkt->pts + (i - delay - 1) * pkt->duration;
        for (i = 0; i<delay && st->pts_buffer[i] > st->pts_buffer[i + 1]; i++)
            FFSWAP(int64_t, st->pts_buffer[i], st->pts_buffer[i + 1]);

        pkt->dts = st->pts_buffer[0];
    }

    if (st->cur_dts && st->cur_dts != AV_NOPTS_VALUE &&
        ((!(s->oformat->flags & AVFMT_TS_NONSTRICT) &&
          st->cur_dts >= pkt->dts) || st->cur_dts > pkt->dts)) {
        av_log(s, AV_LOG_ERROR,
               "Application provided invalid, non monotonically increasing dts to muxer in stream %d: %s >= %s\n",
               st->index, av_ts2str(st->cur_dts), av_ts2str(pkt->dts));
        return AVERROR(EINVAL);
    }
    if (pkt->dts != AV_NOPTS_VALUE && pkt->pts != AV_NOPTS_VALUE && pkt->pts < pkt->dts) {
        av_log(s, AV_LOG_ERROR,
               "pts (%s) < dts (%s) in stream %d\n",
               av_ts2str(pkt->pts), av_ts2str(pkt->dts),
               st->index);
        return AVERROR(EINVAL);
    }

    av_dlog(s, "av_write_frame: pts2:%s dts2:%s\n",
            av_ts2str(pkt->pts), av_ts2str(pkt->dts));
    st->cur_dts = pkt->dts;
    st->pts.val = pkt->dts;

    /* update pts */
    switch (st->codec->codec_type) {
    case AVMEDIA_TYPE_AUDIO:
        frame_size = (pkt->flags & AV_PKT_FLAG_UNCODED_FRAME) ?
                     ((AVFrame *)pkt->data)->nb_samples :
                     av_get_audio_frame_duration(st->codec, pkt->size);

        /* HACK/FIXME, we skip the initial 0 size packets as they are most
         * likely equal to the encoder delay, but it would be better if we
         * had the real timestamps from the encoder */
        if (frame_size >= 0 && (pkt->size || st->pts.num != st->pts.den >> 1 || st->pts.val)) {
            frac_add(&st->pts, (int64_t)st->time_base.den * frame_size);
        }
        break;
    case AVMEDIA_TYPE_VIDEO:
        frac_add(&st->pts, (int64_t)st->time_base.den * st->codec->time_base.num);
        break;
    }
    return 0;
}
```

从代码中可以看出，compute_pkt_fields2()主要有两方面的功能：一方面用于计算AVPacket的duration， dts等信息；另一方面用于检查pts、dts这些参数的合理性（例如PTS是否一定大于DTS）。具体的代码还没有细看，以后有时间再进行分析。

AVOutputFormat->write_packet()

write_packet()函数的定义位于libavformat\mux.c，如下所示。

[cpp]

```cpp
/**
 * Make timestamps non negative, move side data from payload to internal struct, call muxer, and restore
 * sidedata.
 *
 * FIXME: this function should NEVER get undefined pts/dts beside when the
 * AVFMT_NOTIMESTAMPS is set.
 * Those additional safety checks should be dropped once the correct checks
 * are set in the callers.
 */
static int write_packet(AVFormatContext *s, AVPacket *pkt)
{
    int ret, did_split;

    if (s->output_ts_offset) {
        AVStream *st = s->streams[pkt->stream_index];
        int64_t offset = av_rescale_q(s->output_ts_offset, AV_TIME_BASE_Q, st->time_base);

        if (pkt->dts != AV_NOPTS_VALUE)
            pkt->dts += offset;
        if (pkt->pts != AV_NOPTS_VALUE)
            pkt->pts += offset;
    }

    if (s->avoid_negative_ts > 0) {
        AVStream *st = s->streams[pkt->stream_index];
        int64_t offset = st->mux_ts_offset;

        if (s->offset == AV_NOPTS_VALUE && pkt->dts != AV_NOPTS_VALUE &&
            (pkt->dts < 0 || s->avoid_negative_ts == AVFMT_AVOID_NEG_TS_MAKE_ZERO)) {
            s->offset = -pkt->dts;
            s->offset_timebase = st->time_base;
        }

        if (s->offset != AV_NOPTS_VALUE && !offset) {
            offset = st->mux_ts_offset =
                av_rescale_q_rnd(s->offset,
                                 s->offset_timebase,
                                 st->time_base,
                                 AV_ROUND_UP);
        }

        if (pkt->dts != AV_NOPTS_VALUE)
            pkt->dts += offset;
        if (pkt->pts != AV_NOPTS_VALUE)
            pkt->pts += offset;

        av_assert2(pkt->dts == AV_NOPTS_VALUE || pkt->dts >= 0 || s->max_interleave_delta > 0);
        if (pkt->dts != AV_NOPTS_VALUE && pkt->dts < 0) {
            av_log(s, AV_LOG_WARNING,
                   "Packets poorly interleaved, failed to avoid negative "
                   "timestamp %s in stream %d.\n"
                   "Try -max_interleave_delta 0 as a possible workaround.\n",
                   av_ts2str(pkt->dts),
                   pkt->stream_index
            );
        }
    }

    did_split = av_packet_split_side_data(pkt);
    if ((pkt->flags & AV_PKT_FLAG_UNCODED_FRAME)) {
        AVFrame *frame = (AVFrame *)pkt->data;
        av_assert0(pkt->size == UNCODED_FRAME_PACKET_SIZE);
        ret = s->oformat->write_uncoded_frame(s, pkt->stream_index, &frame, 0);
        av_frame_free(&frame);
    } else {
        //写入
        ret = s->oformat->write_packet(s, pkt);
    }

    if (s->flush_packets && s->pb && ret >= 0 && s->flags & AVFMT_FLAG_FLUSH_PACKETS)
        avio_flush(s->pb);

    if (did_split)
        av_packet_merge_side_data(pkt);

    return ret;
}
```

write_packet()函数最关键的地方就是调用了AVOutputFormat中写入数据的方法。如果AVPacket中的flag标记中包含AV_PKT_FLAG_UNCODED_FRAME，就会调用AVOutputFormat的write_uncoded_frame()函数；如果不包含那个标记，就会调用write_packet()函数。write_packet()实际上是一个函数指针，指向特定的AVOutputFormat中的实现函数。例如，我们看一下FLV对应的AVOutputFormat，位于libavformat\flvenc.c，如下所示。

[cpp]

```cpp
AVOutputFormat ff_flv_muxer = {
    .name           = "flv",
    .long_name      = NULL_IF_CONFIG_SMALL("FLV (Flash Video)"),
    .mime_type      = "video/x-flv",
    .extensions     = "flv",
    .priv_data_size = sizeof(FLVContext),
    .audio_codec    = CONFIG_LIBMP3LAME ? AV_CODEC_ID_MP3 : AV_CODEC_ID_ADPCM_SWF,
    .video_codec    = AV_CODEC_ID_FLV1,
    .write_header   = flv_write_header,
    .write_packet   = flv_write_packet,
    .write_trailer  = flv_write_trailer,
    .codec_tag      = (const AVCodecTag* const []) {
                          flv_video_codec_ids, flv_audio_codec_ids, 0
                      },
    .flags          = AVFMT_GLOBALHEADER | AVFMT_VARIABLE_FPS |
                      AVFMT_TS_NONSTRICT,
};
```

从ff_flv_muxer的定义可以看出，write_packet()指向的是flv_write_packet()函数。在看flv_write_packet()函数的定义之前，我们先回顾一下FLV封装格式的结构。

FLV封装格式

FLV封装格式如下图所示。

PS:原图是网上找的,感觉画的很清晰,比官方的Video File Format Specification更加通俗易懂。但是图中有一个错误,就是TagHeader中的StreamID字段的长度写错了（查看了一下官方标准，应该是3字节，现在已经改过来了）。

从FLV的封装格式结构可以看出，它的文件数据是一个一个的Tag连接起来的，中间间隔包含着Previous Tag Size。因此，flv_write_packet()函数的任务就是写入一个Tag和Previous Tag Size。下面简单记录一下Tag Data的格式。Tag Data根据Tag的Type不同而不同：可以分为音频Tag Data，视频Tag Data以及Script Tag Data。下面简述一下音频Tag Data和视频Tag Data。

Audio Tag Data

Audio Tag在官方标准中定义如下。

Audio Tag开始的第1个字节包含了音频数据的参数信息，从第2个字节开始为音频流数据。
第1个字节的前4位的数值表示了音频数据格式：

0 = Linear PCM, platform endian
1 = ADPCM
2 = MP3
3 = Linear PCM, little endian
4 = Nellymoser 16-kHz mono
5 = Nellymoser 8-kHz mono
6 = Nellymoser
7 = G.711 A-law logarithmic PCM
8 = G.711 mu-law logarithmic PCM
9 = reserved
10 = AAC
14 = MP3 8-Khz
15 = Device-specific sound

第1个字节的第5-6位的数值表示采样率：0 = 5.5kHz，1 = 11KHz，2 = 22 kHz，3 = 44 kHz。
第1个字节的第7位表示采样精度：0 = 8bits，1 = 16bits。
第1个字节的第8位表示音频类型：0 = sndMono，1 = sndStereo。
其中，当音频编码为AAC的时候，第一个字节后面存储的是AACAUDIODATA，格式如下所示。

Video Tag Data

Video Tag在官方标准中的定义如下。

Video Tag也用开始的第1个字节包含视频数据的参数信息，从第2个字节为视频流数据。
第1个字节的前4位的数值表示帧类型（FrameType）：

1: keyframe (for AVC, a seekableframe)（关键帧）
2: inter frame (for AVC, a nonseekableframe)
3: disposable inter frame (H.263only)
4: generated keyframe (reservedfor server use only)

5: video info/command frame

第1个字节的后4位的数值表示视频编码ID（CodecID）：

1: JPEG (currently unused)

2: Sorenson H.263

3: Screen video

4: On2 VP6

5: On2 VP6 with alpha channel

6: Screen video version 2

7: AVC

其中，当音频编码为AVC（H.264）的时候，第一个字节后面存储的是AVCVIDEOPACKET，格式如下所示。

flv_write_packet()

下面我们看一下FLV格式中write_packet()对应的实现函数flv_write_packet()的定义，位于libavformat\flvenc.c，如下所示。

[cpp]

```cpp
static int flv_write_packet(AVFormatContext *s, AVPacket *pkt)
{
    AVIOContext *pb      = s->pb;
    AVCodecContext *enc  = s->streams[pkt->stream_index]->codec;
    FLVContext *flv      = s->priv_data;
    FLVStreamContext *sc = s->streams[pkt->stream_index]->priv_data;
    unsigned ts;
    int size = pkt->size;
    uint8_t *data = NULL;
    int flags = -1, flags_size, ret;

    if (enc->codec_id == AV_CODEC_ID_VP6F || enc->codec_id == AV_CODEC_ID_VP6A ||
        enc->codec_id == AV_CODEC_ID_VP6  || enc->codec_id == AV_CODEC_ID_AAC)
        flags_size = 2;
    else if (enc->codec_id == AV_CODEC_ID_H264 || enc->codec_id == AV_CODEC_ID_MPEG4)
        flags_size = 5;
    else
        flags_size = 1;

    if (flv->delay == AV_NOPTS_VALUE)
        flv->delay = -pkt->dts;

    if (pkt->dts < -flv->delay) {
        av_log(s, AV_LOG_WARNING,
               "Packets are not in the proper order with respect to DTS\n");
        return AVERROR(EINVAL);
    }

    ts = pkt->dts + flv->delay; // add delay to force positive dts

    if (s->event_flags & AVSTREAM_EVENT_FLAG_METADATA_UPDATED) {
        write_metadata(s, ts);
        s->event_flags &= ~AVSTREAM_EVENT_FLAG_METADATA_UPDATED;
    }
    //Tag Header
    switch (enc->codec_type) {
    case AVMEDIA_TYPE_VIDEO:
        //Type
        avio_w8(pb, FLV_TAG_TYPE_VIDEO);

        flags = enc->codec_tag;
        if (flags == 0) {
            av_log(s, AV_LOG_ERROR,
                   "Video codec '%s' is not compatible with FLV\n",
                   avcodec_get_name(enc->codec_id));
            return AVERROR(EINVAL);
        }
        //Key Frame?
        flags |= pkt->flags & AV_PKT_FLAG_KEY ? FLV_FRAME_KEY : FLV_FRAME_INTER;
        break;
    case AVMEDIA_TYPE_AUDIO:

        flags = get_audio_flags(s, enc);

        av_assert0(size);
        //Type
        avio_w8(pb, FLV_TAG_TYPE_AUDIO);
        break;
    case AVMEDIA_TYPE_DATA:
        //Type
        avio_w8(pb, FLV_TAG_TYPE_META);
        break;
    default:
        return AVERROR(EINVAL);
    }

    if (enc->codec_id == AV_CODEC_ID_H264 || enc->codec_id == AV_CODEC_ID_MPEG4) {
        /* check if extradata looks like mp4 formated */
        if (enc->extradata_size > 0 && *(uint8_t*)enc->extradata != 1)
            if ((ret = ff_avc_parse_nal_units_buf(pkt->data, &data, &size)) < 0)
                return ret;
    } else if (enc->codec_id == AV_CODEC_ID_AAC && pkt->size > 2 &&
               (AV_RB16(pkt->data) & 0xfff0) == 0xfff0) {
        if (!s->streams[pkt->stream_index]->nb_frames) {
        av_log(s, AV_LOG_ERROR, "Malformed AAC bitstream detected: "
               "use the audio bitstream filter 'aac_adtstoasc' to fix it "
               "('-bsf:a aac_adtstoasc' option with ffmpeg)\n");
        return AVERROR_INVALIDDATA;
        }
        av_log(s, AV_LOG_WARNING, "aac bitstream error\n");
    }

    /* check Speex packet duration */
    if (enc->codec_id == AV_CODEC_ID_SPEEX && ts - sc->last_ts > 160)
        av_log(s, AV_LOG_WARNING, "Warning: Speex stream has more than "
                                  "8 frames per packet. Adobe Flash "
                                  "Player cannot handle this!\n");

    if (sc->last_ts < ts)
        sc->last_ts = ts;

    if (size + flags_size >= 1<<24) {
        av_log(s, AV_LOG_ERROR, "Too large packet with size %u >= %u\n",
               size + flags_size, 1<<24);
        return AVERROR(EINVAL);
    }
    //Tag Header - Datasize
    avio_wb24(pb, size + flags_size);
    //Tag Header - Timestamp
    avio_wb24(pb, ts & 0xFFFFFF);
    avio_w8(pb, (ts >> 24) & 0x7F); // timestamps are 32 bits _signed_
    //StreamID
    avio_wb24(pb, flv->reserved);

    if (enc->codec_type == AVMEDIA_TYPE_DATA) {
        int data_size;
        int64_t metadata_size_pos = avio_tell(pb);
        if (enc->codec_id == AV_CODEC_ID_TEXT) {
            // legacy FFmpeg magic?
            avio_w8(pb, AMF_DATA_TYPE_STRING);
            put_amf_string(pb, "onTextData");
            avio_w8(pb, AMF_DATA_TYPE_MIXEDARRAY);
            avio_wb32(pb, 2);
            put_amf_string(pb, "type");
            avio_w8(pb, AMF_DATA_TYPE_STRING);
            put_amf_string(pb, "Text");
            put_amf_string(pb, "text");
            avio_w8(pb, AMF_DATA_TYPE_STRING);
            put_amf_string(pb, pkt->data);
            put_amf_string(pb, "");
            avio_w8(pb, AMF_END_OF_OBJECT);
        } else {
            // just pass the metadata through
            avio_write(pb, data ? data : pkt->data, size);
        }
        /* write total size of tag */
        data_size = avio_tell(pb) - metadata_size_pos;
        avio_seek(pb, metadata_size_pos - 10, SEEK_SET);
        avio_wb24(pb, data_size);
        avio_seek(pb, data_size + 10 - 3, SEEK_CUR);
        avio_wb32(pb, data_size + 11);
    } else {
        av_assert1(flags>=0);
        //First Byte of Tag Data
        avio_w8(pb,flags);
        if (enc->codec_id == AV_CODEC_ID_VP6)
            avio_w8(pb,0);
        if (enc->codec_id == AV_CODEC_ID_VP6F || enc->codec_id == AV_CODEC_ID_VP6A) {
            if (enc->extradata_size)
                avio_w8(pb, enc->extradata[0]);
            else
                avio_w8(pb, ((FFALIGN(enc->width,  16) - enc->width) << 4) |
                             (FFALIGN(enc->height, 16) - enc->height));
        } else if (enc->codec_id == AV_CODEC_ID_AAC)
            avio_w8(pb, 1); // AAC raw
        else if (enc->codec_id == AV_CODEC_ID_H264 || enc->codec_id == AV_CODEC_ID_MPEG4) {
            //AVCVIDEOPACKET-AVCPacketType
            avio_w8(pb, 1); // AVC NALU
            //AVCVIDEOPACKET-CompositionTime
            avio_wb24(pb, pkt->pts - pkt->dts);
        }
        //Data
        avio_write(pb, data ? data : pkt->data, size);

        avio_wb32(pb, size + flags_size + 11); // previous tag size
        flv->duration = FFMAX(flv->duration,
                              pkt->pts + flv->delay + pkt->duration);
    }
```

```cpp

    av_free(data);

    return pb->error;
}
```

我们通过源代码简单梳理一下flv_write_packet()在写入H.264/AAC时候的流程：
（1）写入Tag Header的Type，如果是视频，代码如下：

```cpp
avio_w8(pb, FLV_TAG_TYPE_VIDEO);
```

如果是音频，代码如下：

```cpp
avio_w8(pb, FLV_TAG_TYPE_AUDIO);
```

（2）写入Tag Header的Datasize，Timestamp和StreamID（至此完成Tag Header）：

```cpp
//Tag Header - Datasize
avio_wb24(pb, size + flags_size);
//Tag Header - Timestamp
avio_wb24(pb, ts & 0xFFFFFF);
avio_w8(pb, (ts >> 24) & 0x7F); // timestamps are 32 bits _signed_
//StreamID
avio_wb24(pb, flv->reserved);
```

（3）写入Tag Data的第一字节（其中flag已经在前面的代码中设置完毕）：

```cpp
//First Byte of Tag Data
avio_w8(pb,flags);
```

（4）如果编码格式VP6作相应的处理（不研究）；编码格式为AAC，写入AACAUDIODATA；编码格式为H.264，写入AVCVIDEOPACKET：

```cpp
if (enc->codec_id == AV_CODEC_ID_VP6F || enc->codec_id == AV_CODEC_ID_VP6A) {
    if (enc->extradata_size)
        avio_w8(pb, enc->extradata[0]);
    else
        avio_w8(pb, ((FFALIGN(enc->width,  16) - enc->width) << 4) |
                     (FFALIGN(enc->height, 16) - enc->height));
} else if (enc->codec_id == AV_CODEC_ID_AAC)
    avio_w8(pb, 1); // AAC raw
else if (enc->codec_id == AV_CODEC_ID_H264 || enc->codec_id == AV_CODEC_ID_MPEG4) {
    //AVCVIDEOPACKET-AVCPacketType
    avio_w8(pb, 1); // AVC NALU
    //AVCVIDEOPACKET-CompositionTime
    avio_wb24(pb, pkt->pts - pkt->dts);
}
```

（5）写入数据：

```cpp
//Data
avio_write(pb, data ? data : pkt->data, size);
```

（6）
写入previous tag size：

```cpp
avio_wb32(pb, size + flags_size + 11); // previous tag size
```

至此，flv_write_packet()就完成了一个Tag的写入。

雷霄骅
leixiaohua1020@126.com
http://blog.csdn.net/leixiaohua1020

文章标签： ffmpeg AVPacket FLV 输出 源代码
个人分类： FFMPEG
所属专栏： FFmpeg