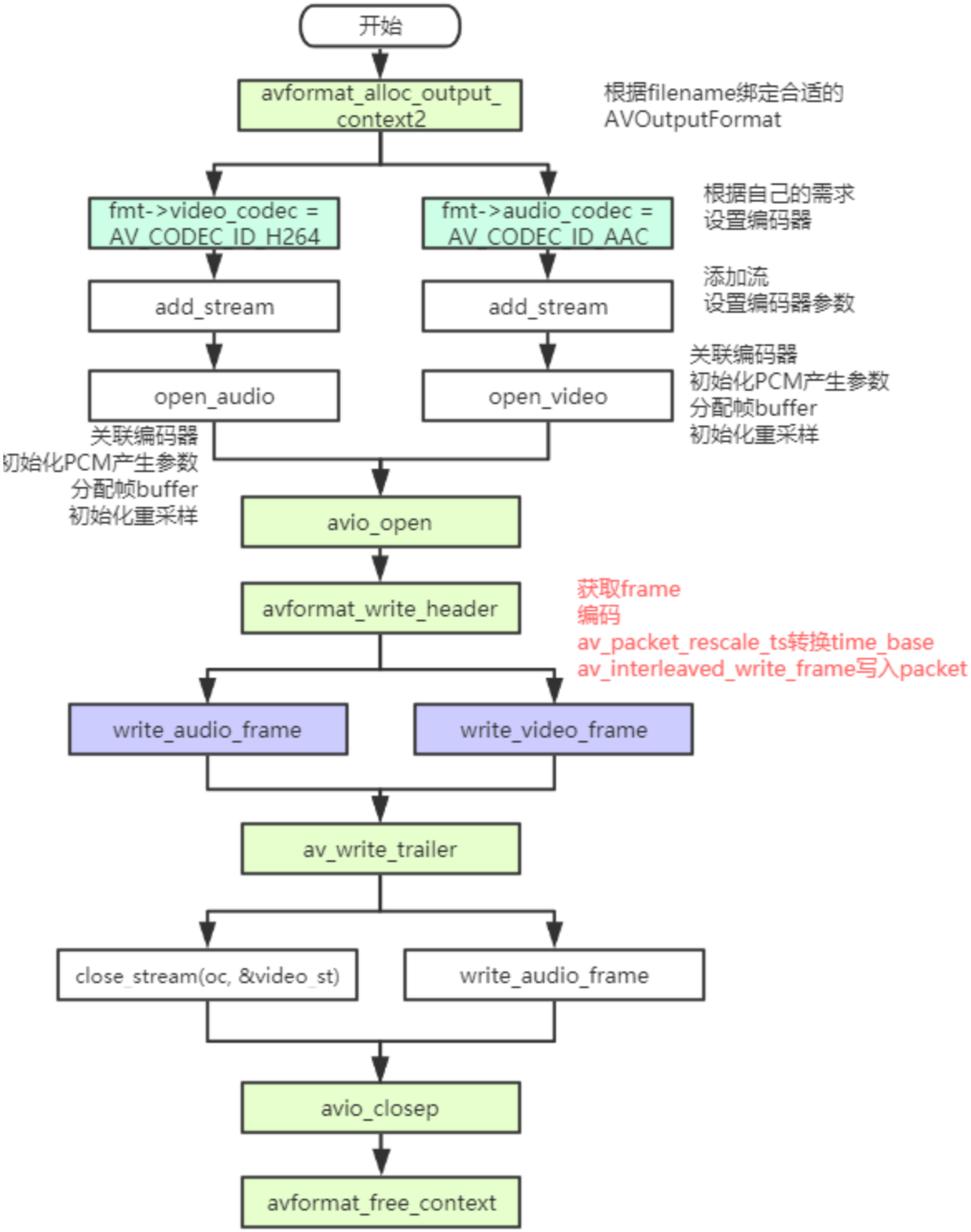
音视频FLV合成实战

FFmpeg合成流程

示例本程序会生成一个合成的音频和视频流，并将它们编码和封装输出到输出文件，输出格式是根据文件扩展名自动猜测的。
示例的流程图如下所示。



ffmpeg 的 Mux 主要分为 三步操作：

- **avformat_write_header** ： 写文件头
- **av_write_frame/av_interleaved_write_frame**： 写packet
- **av_write_trailer** ： 写文件尾

avcodec_parameters_from_context：将AVCodecContext结构体中码流参数拷贝到AVCodecParameters结构体中，和avcodec_parameters_to_context刚好相反。

FFmpeg函数：avformat_write_header

```
int avformat_write_header(AVFormatContext *s, AVDictionary **options
)
{
    int ret = 0;
    int already_initialized = s->internal->initialized;
    int streams_already_initialized = s->internal->streams_initializ
ed;

    if (!already_initialized)
        if ((ret = avformat_init_output(s, options)) < 0)
            return ret;

    if (!(s->oformat->flags & AVFMT_NOFILE) && s->pb)
        avio_write_marker(s->pb, AV_NOPTS_VALUE, AVIO_DATA_MARKER_HE
ADER);
    if (s->oformat->write_header) {
        ret = s->oformat->write_header(s);
        if (ret >= 0 && s->pb && s->pb->error < 0)
            ret = s->pb->error;
        if (ret < 0)
            goto fail;
        flush_if_needed(s);
    }
    if (!(s->oformat->flags & AVFMT_NOFILE) && s->pb)
        avio_write_marker(s->pb, AV_NOPTS_VALUE, AVIO_DATA_MARKER_UN
KNOWN);

    if (!s->internal->streams_initialized) {
        if ((ret = init_pts(s)) < 0)
            goto fail;
    }

```

```
    return streams_already_initialized;

fail:
    if (s->oformat->deinit)
        s->oformat->deinit(s);
    return ret;
}
```

最终调用到复用器的 write_header，比如

```
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,
    .init           = flv_init,
    .write_header   = flv_write_header, // 写头部
    .write_packet   = flv_write_packet,
    .write_trailer  = flv_write_trailer,    // 写文件尾部
    .check_bitstream= flv_check_bitstream,
    .codec_tag      = (const AVCodecTag* const []) {
                          flv_video_codec_ids, flv_audio_codec_ids,
0
                      },
    .flags          = AVFMT_GLOBALHEADER | AVFMT_VARIABLE_FPS |
                      AVFMT_TS_NONSTRICT,
    .priv_class     = &flv_muxer_class,
};
```

FFmpeg结构体：avformat_alloc_output_context2

函数在libavformat.h里面的定义

```
/**
 * Allocate an AVFormatContext for an output format.
 * avformat_free_context() can be used to free the context and
 * everything allocated by the framework within it.
 *
 * @param *ctx is set to the created format context, or to NULL in
 * case of failure
 * @param oformat format to use for allocating the context, if NULL
 * format_name and filename are used instead
 * @param format_name the name of output format to use for allocating the
 * context, if NULL filename is used instead
 * @param filename the name of the filename to use for allocating the
 * context, may be NULL
 * @return >= 0 in case of success, a negative AVERROR code in case of
 * failure
 */
int avformat_alloc_output_context2(AVFormatContext **ctx, ff_const59 AVOutputFormat *oformat,
                                   const char *format_name, const char *filename);
```

函数参数的介绍：

- ctx:需要创建的context，返回NULL表示失败。
- oformat:指定对应的AVOutputFormat，如果不指定，可以通过后面format_name、filename两个参数进行指定，让ffmpeg自己推断。
- format_name: 指定音视频的格式，比如“flv”，“mpeg”等，如果设置为NULL，则由filename进行指定，让ffmpeg自己推断。
- filename: 指定音视频文件的路径，如果oformat、format_name为NULL，则ffmpeg内部根据filename后缀名选择合适的复用器，比如xxx.flv则使用flv复用器。

```
int avformat_alloc_output_context2(AVFormatContext **avctx, ff_const59 AVOutputFormat *oformat,
                                   const char *format, const char *filename)
{
    AVFormatContext *s = avformat_alloc_context();
    int ret = 0;

    *avctx = NULL;
    if (!s)
        goto nomem;

    if (!oformat) {  // oformat为NULL
        if (format) {
            oformat = av_guess_format(format, NULL, NULL); //根据提供的格式进行查找 format
            if (!oformat) {
                av_log(s, AV_LOG_ERROR, "Requested output format '%s' is not a suitable output format\n", format);
                ret = AVERROR(EINVAL);
                goto error;
            }
        } else {    // oformat和format都为NULL
            oformat = av_guess_format(NULL, filename, NULL); // 根据文件名后缀进行查找
            if (!oformat) {
                ret = AVERROR(EINVAL);
                av_log(s, AV_LOG_ERROR, "Unable to find a suitable output format for '%s'\n",
                       filename);
                goto error;
            }
        }
    }

    s->oformat = oformat;
    if (s->oformat->priv_data_size > 0) {
        s->priv_data = av_mallocz(s->oformat->priv_data_size);
        if (!s->priv_data)
```

```
            goto nomem;
        if (s->oformat->priv_class) {
            *(const AVClass**)s->priv_data= s->oformat->priv_class;
            av_opt_set_defaults(s->priv_data);
        }
    } else
        s->priv_data = NULL;

    if (filename) {
#if FF_API_FORMAT_FILENAME
FF_DISABLE_DEPRECATION_WARNINGS
        av_strlcpy(s->filename, filename, sizeof(s->filename));
FF_ENABLE_DEPRECATION_WARNINGS
#endif
        if (!(s->url = av_strdup(filename)))
            goto nomem;

    }
    *avctx = s;
    return 0;
nomem:
    av_log(s, AV_LOG_ERROR, "Out of memory\n");
    ret = AVERROR(ENOMEM);
error:
    avformat_free_context(s);
    return ret;
}
```

可以看出，里面最主要的就两个函数，avformat_alloc_context和av_guess_format，一个是申请内存分配上下文，一个是通过后面两个参数获取AVOutputFormat。

出av_guess_format这个函数会通过filename和short_name来和所有的编码器进行比对，找出最接近的编码器然后返回。

```
ff_const59 AVOutputFormat *av_guess_format(const char *short_name, c
onst char *filename,
                                const char *mime_type)
{
```

```
    const AVOutputFormat *fmt = NULL;
    AVOutputFormat *fmt_found = NULL;
    void *i = 0;
    int score_max, score;

    /* specific test for image sequences */
#if CONFIG_IMAGE2_MUXER
    if (!short_name && filename &&
        av_filename_number_test(filename) &&
        ff_guess_image2_codec(filename) != AV_CODEC_ID_NONE) {
        return av_guess_format("image2", NULL, NULL);
    }
#endif
    /* Find the proper file type. */
    score_max = 0;
    while ((fmt = av_muxer_iterate(&i))) {
        score = 0;
        if (fmt->name && short_name && av_match_name(short_name, fmt
->name)) // fmt->name比如ff_flv_muxer的为"flv"
            score += 100;  // 匹配了name 最高规格
        if (fmt->mime_type && mime_type && !strcmp(fmt->mime_type, m
ime_type)) // ff_flv_muxer的为 "video/x-flv"
            score += 10; // 匹配mime_type
        if (filename && fmt->extensions &&
            av_match_ext(filename, fmt->extensions)) { //ff_flv_muxe
r的为  "flv"
            score += 5; // 匹配
        }
        if (score > score_max) {
            score_max = score;   // 更新最匹配的分值
            fmt_found = (AVOutputFormat*)fmt;
        }
    }
    return fmt_found;
}
```

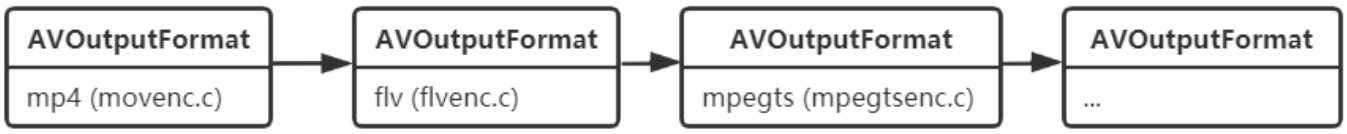
FFmpeg结构体：AVOutputFormat

1.描述

AVOutpufFormat表示输出文件容器格式，AVOutputFormat 结构主要包含的信息有：封装名称描述，编码格式信息(video/audio 默认编码格式，支持的编码格式列表)，一些对封装的操作函数(write_header,write_packet,write_tailer等)。

ffmpeg支持各种各样的输出文件格式，MP4，FLV，3GP等等。而 AVOutputFormat 结构体则保存了这些格式的信息和一些常规设置。

每一种封装对应一个 AVOutputFormat 结构，ffmpeg将AVOutputFormat 按照链表存储：



2.结构体定义

```
/**
 * @addtogroup lavf_encoding
 * @{
 */
typedef struct AVOutputFormat {
    const char *name;
    /**
     * Descriptive name for the format, meant to be more human-read
able
     * than name. You should use the NULL_IF_CONFIG_SMALL() macro
     * to define it.
     */
    const char *long_name;
    const char *mime_type;
    const char *extensions; /**< comma-separated filename extension
s */
    /* output support */
    enum AVCodecID audio_codec;    /**< default audio codec */
    enum AVCodecID video_codec;    /**< default video codec */
    enum AVCodecID subtitle_codec; /**< default subtitle codec */
    /**
```

```
20      * can use flags: AVFMT_NOFILE, AVFMT_NEEDNUMBER,
21      * AVFMT_GLOBALHEADER, AVFMT_NOTIMESTAMPS, AVFMT_VARIABLE_FPS,
22      * AVFMT_NODIMENSIONS, AVFMT_NOSTREAMS, AVFMT_ALLOW_FLUSH,
23      * AVFMT_TS_NONSTRICT, AVFMT_TS_NEGATIVE
24      */
25     int flags;
26
27     /**
28      * List of supported codec_id-codec_tag pairs, ordered by "bett
   er
29      * choice first". The arrays are all terminated by AV_CODEC_ID_
   NONE.
30      */
31     const struct AVCodecTag * const *codec_tag;
32
33
34     const AVClass *priv_class; ///< AVClass for the private context
35
36     /*****************************************************************
   ***
37      * No fields below this line are part of the public API. They
38      * may not be used outside of libavformat and can be changed an
   d
39      * removed at will.
40      * New public fields should be added right above.
41      *****************************************************************
   ***
42      */
43     /**
44      * The ff_const59 define is not part of the public API and will
45      * be removed without further warning.
46      */
47 #if FF_API_AVIOFORMAT
48 #define ff_const59
49 #else
50 #define ff_const59 const
51 #endif
52     ff_const59 struct AVOutputFormat *next;
53     /**
54      * size of private data so that it can be allocated in the wrap
```

```
per
55       */
56      int priv_data_size;
57
58      int (*write_header)(struct AVFormatContext *);
59      /**
60       * Write a packet. If AVFMT_ALLOW_FLUSH is set in flags,
61       * pkt can be NULL in order to flush data buffered in the muxe
    r.
62       * When flushing, return 0 if there still is more data to flus
    h,
63       * or 1 if everything was flushed and there is no more buffered
64       * data.
65       */
66      int (*write_packet)(struct AVFormatContext *, AVPacket *pkt);
67      int (*write_trailer)(struct AVFormatContext *);
68      /**
69       * Currently only used to set pixel format if not YUV420P.
70       */
71      int (*interleave_packet)(struct AVFormatContext *, AVPacket *ou
    t,
72                               AVPacket *in, int flush);
73      /**
74       * Test if the given codec can be stored in this container.
75       *
76       * @return 1 if the codec is supported, 0 if it is not.
77       *         A negative number if unknown.
78       *         MKTAG('A', 'P', 'I', 'C') if the codec is only suppo
    rted as AV_DISPOSITION_ATTACHED_PIC
79       */
80      int (*query_codec)(enum AVCodecID id, int std_compliance);
81
82      void (*get_output_timestamp)(struct AVFormatContext *s, int str
    eam,
83                                   int64_t *dts, int64_t *wall);
84      /**
85       * Allows sending messages from application to device.
86       */
87      int (*control_message)(struct AVFormatContext *s, int type,
88                             void *data, size_t data_size);
```

```
89
90      /**
91       * Write an uncoded AVFrame.
92       *
93       * See av_write_uncoded_frame() for details.
94       *
95       * The library will free *frame afterwards, but the muxer can p
   revent it
96       * by setting the pointer to NULL.
97       */
98      int (*write_uncoded_frame)(struct AVFormatContext *, int stream
   _index,
99                                 AVFrame **frame, unsigned flags);
100     /**
101      * Returns device list with it properties.
102      * @see avdevice_list_devices() for more details.
103      */
104     int (*get_device_list)(struct AVFormatContext *s, struct AVDevi
   ceInfoList *device_list);
105     /**
106      * Initialize device capabilities submodule.
107      * @see avdevice_capabilities_create() for more details.
108      */
109     int (*create_device_capabilities)(struct AVFormatContext *s, st
   ruct AVDeviceCapabilitiesQuery *caps);
110     /**
111      * Free device capabilities submodule.
112      * @see avdevice_capabilities_free() for more details.
113      */
114     int (*free_device_capabilities)(struct AVFormatContext *s, stru
   ct AVDeviceCapabilitiesQuery *caps);
115     enum AVCodecID data_codec; /**< default data codec */
116     /**
117      * Initialize format. May allocate data here, and set any AVFor
   matContext or
118      * AVStream parameters that need to be set before packets are s
   ent.
119      * This method must not write output.
120      *
121      * Return 0 if streams were fully configured, 1 if not, negativ
```

```
    e AVERROR on failure
122      *
123      * Any allocations made here must be freed in deinit().
124      */
125     int (*init)(struct AVFormatContext *);
126     /**
127      * Deinitialize format. If present, this is called whenever the
    muxer is being
128      * destroyed, regardless of whether or not the header has been
     written.
129      *
130      * If a trailer is being written, this is called after write_tr
    ailer().
131      *
132      * This is called if init() fails as well.
133      */
134     void (*deinit)(struct AVFormatContext *);
135     /**
136      * Set up any necessary bitstream filtering and extract any ext
    ra data needed
137      * for the global header.
138      * Return 0 if more packets from this stream must be checked; 1
    if not.
139      */
140     int (*check_bitstream)(struct AVFormatContext *, const AVPacket
    *pkt);
141 } AVOutputFormat;
```

3.常见变量及其作用

const char *name; // 复用器名称
const char *long_name;//格式的描述性名称，易于阅读。
enum AVCodecID audio_codec; //默认的音频编解码器
enum AVCodecID video_codec; //默认的视频编解码器
enum AVCodecID subtitle_codec; //默认的字幕编解码器

大部分复用器都有默认的编码器，所以大家如果要调整编码器类型则需要自己手动指定。

比如AVOutputFormat ff_flv_muxer

```
AVOutputFormat ff_flv_muxer = {
    .name           = "flv",
    .audio_codec    = CONFIG_LIBMP3LAME ? AV_CODEC_ID_MP3 :
AV_CODEC_ID_ADPCM_SWF, // 默认了MP3
    .video_codec    = AV_CODEC_ID_FLV1,
   ....
};

AVOutputFormat ff_mpegts_muxer = {
    .name           = "mpegts",
    .extensions     = "ts,m2t,m2ts,mts",
    .audio_codec    = AV_CODEC_ID_MP2,
    .video_codec    = AV_CODEC_ID_MPEG2VIDEO,
    ....
};
```

int (*write_header)(struct AVFormatContext *);
int (*write_packet)(struct AVFormatContext *, AVPacket *pkt);//写一个数据包。 如果在标志中设置AVFMT_ALLOW_FLUSH，则pkt可以为NULL。
int (*write_trailer)(struct AVFormatContext *);
int (*interleave_packet)(struct AVFormatContext *, AVPacket *out, AVPacket *in, int flush);
int (*control_message)(struct AVFormatContext *s, int type, void *data, size_t data_size);//允许从应用程序向设备发送消息。
int (*write_uncoded_frame)(struct AVFormatContext *, int stream_index, AVFrame **frame, unsigned flags);//写一个未编码的AVFrame。
int (*init)(struct AVFormatContext *);//初始化格式。 可以在此处分配数据，并设置在发送数据包之前需要设置的任何AVFormatContext或AVStream参数。
void (*deinit)(struct AVFormatContext *);//取消初始化格式。
int (*check_bitstream)(struct AVFormatContext *, const AVPacket *pkt);//设置任何必要的比特流过滤，并提取全局头部所需的任何额外数据。

FFmpeg函数：avformat_new_stream

AVStream 即是流通道。例如我们将 H264 和 AAC 码流存储为MP4文件的时候，就需要在 MP4文件中增加两个流通道，一个存储Video：H264，一个存储Audio：AAC。（假设H264和AAC只包含单个流通道）。

```
/**
* Add a new stream to a media file.
*
* When demuxing, it is called by the demuxer in read_header(). If the
* flag AVFMTCTX_NOHEADER is set in s.ctx_flags, then it may also
* be called in read_packet().
*
* When muxing, should be called by the user before avformat_write_header().
*
* User is required to call avcodec_close() and avformat_free_context() to
* clean up the allocation by avformat_new_stream().
*
* @param s media file handle
* @param c If non-NULL, the AVCodecContext corresponding to the new stream
* will be initialized to use this codec. This is needed for e.g. codec-specific
* defaults to be set, so codec should be provided if it is known.
*
* @return newly created stream or NULL on error.
*/
AVStream *avformat_new_stream(AVFormatContext *s, const AVCodec *c);
```

avformat_new_stream 在 AVFormatContext 中创建 Stream 通道。

关联的结构体

AVFormatContext ：

unsigned int nb_streams;　记录stream通道数目。

AVStream **streams;　存储stream通道。

AVStream ：

int index;　在AVFormatContext 中所处的通道索引

avformat_new_stream之后便在 AVFormatContext 里增加了 AVStream 通道（相关的index已经被设置了）。之后，我们就可以自行设置 AVStream 的一些参数信息。例如：codec_id , format ,bit_rate ,width , height

FFmpeg函数：av_interleaved_write_frame

函数原型：int av_interleaved_write_frame(AVFormatContext *s, AVPacket *pkt);

说明：将数据包写入输出媒体文件，并确保正确的交织（保持packet dts的增长性）。
该函数会在内部根据需要缓存packet，以确保输出文件中的packet按dts递增的顺序正确交织。如果自己进行交织则应调用av_write_frame()。

参数：

s	媒体文件句柄
pkt	要写入的packet。 如果packet使用引用参考计数的内存方式，则此函数将获取此引用权(可以理解为move了reference)，并在内部在合适的时候进行释放。此函数返回后，调用者不得通过此引用访问数据。如果packet没有引用计数，libavformat将进行复制。 此参数可以为NULL（在任何时候，不仅在结尾），以刷新交织队列。 Packet的stream_index字段必须设置为s-> streams中相应流的索引。 时间戳记（pts，dts）必须设置为stream's timebase中的正确值（除非输出格式用AVFMT_NOTIMESTAMPS标志标记，然后可以将其设置为AV_NOPTS_VALUE）。 同一stream后续packet的dts必须严格递增（除非输出格式用AVFMT_TS_NONSTRICT标记，则它们只必须不减少）。duration也应设置(如果已知)。

返回值：成功时为0，错误时为负AVERROR。即使此函数调用失败，Libavformat仍将始终释放该packet。

FFmpeg函数：av_compare_ts

```
/**
* Compare two timestamps each in its own time base.
*
* @return One of the following values:
*         - -1 if `ts_a` is before `ts_b`
*         - 1 if `ts_a` is after `ts_b`
*         - 0 if they represent the same position
*
* @warning
* The result of the function is undefined if one of the timestamps is outside
* the `int64_t` range when represented in the other's timebase.
*/
int av_compare_ts(int64_t ts_a, AVRational tb_a, int64_t ts_b, AVRational tb_b);
```

返回值：

- -1 ts_a 在ts_b之前
- 1 ts_a 在ts_b之后
- 0 ts_a 在ts_b同一位置

用伪代码：return ts_a == ts_b ? 0 : ts_a < ts_b ? -1 : 1

MediaInfo分析文件写入

这里只是分析avformat_write_header和av_write_trailer的作用。

flv

只写**avformat_write_header**

000 File Header (9 bytes)
000 FLV header (9 bytes)
000 Signature: FLV
003 Version: 1 (0x01)
004 Flags: 5 (0x05)
005 Video: Yes
005 Audio: Yes
005 Size: 9 (0x00000009)
009 ----------
009 --- FLV, accepted ---
009 ----------
009 Meta - onMetaData - 12 elements (288 bytes)
009 Header (15 bytes)
009 PreviousTagSize: 0 (0x00000000)
00D Type: 18 (0x12)
00E BodyLength: 273 (0x000111)
011 Timestamp_Base: 0 (0x000000)
014 Timestamp_Extended: 0 (0x00)
015 StreamID: 0 (0x000000)
018 Type: 2 (0x02) - SCRIPTDATASTRING
019 Value_Size: 10 (0x000A)
01B Value: onMetaData
025 Type: 8 (0x08) - SCRIPTDATAVARIABLE[ECMAArrayLength]
026 ECMAArrayLength: 12 (0x0000000C)
02A duration (19 bytes)
02A StringLength: 8 (0x0008)
02C StringData: duration
034 Type: 0 (0x00) - DOUBLE
035 Value: 0.000
03D width - 352 (16 bytes)
03D StringLength: 5 (0x0005)
03F StringData: width

044 Type: 0 (0x00) – DOUBLE
045 Value: 352.000
04D height – 288 (17 bytes)
04D StringLength: 6 (0x0006)
04F StringData: height
055 Type: 0 (0x00) – DOUBLE
056 Value: 288.000
05E videodatarate – 390625 (24 bytes)
05E StringLength: 13 (0x000D)
060 StringData: videodatarate
06D Type: 0 (0x00) – DOUBLE
06E Value: 390.625
076 videocodecid – 2 (23 bytes)
076 StringLength: 12 (0x000C)
078 StringData: videocodecid
084 Type: 0 (0x00) – DOUBLE
085 Value: 2.000
08D audiodatarate – 62500 (24 bytes)
08D StringLength: 13 (0x000D)
08F StringData: audiodatarate
09C Type: 0 (0x00) – DOUBLE
09D Value: 62.500
0A5 audiosamplerate – 44100 (26 bytes)
0A5 StringLength: 15 (0x000F)
0A7 StringData: audiosamplerate
0B6 Type: 0 (0x00) – DOUBLE
0B7 Value: 44100.000
0BF audiosamplesize – 16 (26 bytes)
0BF StringLength: 15 (0x000F)
0C1 StringData: audiosamplesize
0D0 Type: 0 (0x00) – DOUBLE
0D1 Value: 16.000
0D9 stereo – 1 (0x1) (10 bytes)
0D9 StringLength: 6 (0x0006)
0DB StringData: stereo
0E1 Type: 1 (0x01) – UI8
0E2 Value: 1 (0x01)
0E3 audiocodecid – 2 (23 bytes)
0E3 StringLength: 12 (0x000C)
0E5 StringData: audiocodecid

```
0F1    Type:                          0 (0x00) - DOUBLE
0F2    Value:                         2.000
0FA  encoder - Lavf58.29.100 (25 bytes)
0FA    StringLength:                  7 (0x0007)
0FC    StringData:                    encoder
103    Type:                          2 (0x02) - SCRIPTDATASTRING
104    Value_Size:                    13 (0x000D)
106    Value:                         Lavf58.29.100
113  filesize (19 bytes)
113    StringLength:                  8 (0x0008)
115    StringData:                    filesize
11D    Type:                          0 (0x00) - DOUBLE
11E    Value:                         0.000
129 End Of File (4 bytes)
129  Header (4 bytes)
129    PreviousTagSize:               284 (0x0000011C)
12D ------------------------
12D ---   FLV, filling   ---
12D ------------------------
12D -------------------------
12D ---   FLV, finished   ---
12D -------------------------
```

avformat_write_header+ av_write_trailer
对于FLV而言没有任何变化。

mp4

avformat_write_header

```
00 File Type (32 bytes)
00  Header (8 bytes)
00    Size:                          32 (0x00000020)
04    Name:                          ftyp
08  MajorBrand:                      isom
0C  MajorBrandVersion:               512 (0x00000200)
10  CompatibleBrand:                 isom
14  CompatibleBrand:                 iso2
18  CompatibleBrand:                 avc1
1C  CompatibleBrand:                 mp41
20 ---------------------------
```

```
20 ---   MPEG-4, accepted   ---
20 ----------------------------
20 Free space (8 bytes)
20  Header (8 bytes)
20   Size:                      8 (0x00000008)
24   Name:                        free
28 Junk (4 bytes)
28  Header (4 bytes)
28   Size:                      0 (0x00000000)
2C Problem (4 bytes)
2C  Header (4 bytes)
2C   Size:                      1835295092 (0x6D646174)
30   Size is wrong:               0 (0x00000000)
30 ----------------------------
30 ---   MPEG-4, filling   ---
30 ----------------------------
30 ----------------------------
30 ---   MPEG-4, finished   ---
30 ----------------------------
```

avformat_write_header+av_write_trailer

```
000 File Type (32 bytes)
000  Header (8 bytes)
000   Size:                     32 (0x00000020)
004   Name:                       ftyp
008  MajorBrand:                    isom
00C  MajorBrandVersion:                512 (0x00000200)
010  CompatibleBrand:               isom
014  CompatibleBrand:               iso2
018  CompatibleBrand:               avc1
01C  CompatibleBrand:               mp41
020 ----------------------------
020 ---   MPEG-4, accepted   ---
020 ----------------------------
020 Free space (8 bytes)
020  Header (8 bytes)
020   Size:                      8 (0x00000008)
024   Name:                        free
028 Data (8 bytes)
```

028 Header (8 bytes)
028 Size: 8 (0x00000008)
02C Name: mdat
030 File header (214 bytes)
030 Header (8 bytes)
030 Size: 214 (0x000000D6)
034 Name: moov
038 Movie header (108 bytes)
038 Header (8 bytes)
038 Size: 108 (0x0000006C)
03C Name: mvhd
040 Version: 0 (0x00)
041 Flags: 0 (0x000000)
044 Creation time: 0 (0x00000000) –
048 Modification time: 0 (0x00000000) –
04C Time scale: 1000 (0x000003E8) – 1000 Hz
050 Duration: 0 (0x00000000) – 0 ms
054 Preferred rate: 65536 (0x00010000) – 1.000
058 Preferred volume: 256 (0x0100) – 1.000
05A Reserved: (10 bytes)
064 Matrix structure (36 bytes)
064 a (width scale): 1.000
068 b (width rotate): 0.000
06C u (width angle): 0.000
070 c (height rotate): 0.000
074 d (height scale): 1.000
078 v (height angle): 0.000
07C x (position left): 0.000
080 y (position top): 0.000
084 w (divider): 1.000
088 Preview time: 0 (0x00000000)
08C Preview duration: 0 (0x00000000)
090 Poster time: 0 (0x00000000)
094 Selection time: 0 (0x00000000)
098 Selection duration: 0 (0x00000000)
09C Current time: 0 (0x00000000)
0A0 Next track ID: 2 (0x00000002)
0A4 User Data (98 bytes)
0A4 Header (8 bytes)
0A4 Size: 98 (0x00000062)

```
0A8    Name:                          udta
0AC   Metadata (90 bytes)
0AC    Header (8 bytes)
0AC     Size:                        90 (0x0000005A)
0B0     Name:                         meta
0B4    Version:                      0 (0x00)
0B5    Flags:                       0 (0x000000)
0B8    Metadata Header (33 bytes)
0B8     Header (8 bytes)
0B8      Size:                      33 (0x00000021)
0BC      Name:                        hdlr
0C0     Version:                      0 (0x00)
0C1     Flags:                     0 (0x000000)
0C4     Type (Quicktime):
0C8     Metadata type:                  mdir
0CC     Manufacturer:                   appl
0D0     Component reserved flags:          0 (0x00000000)
0D4     Component reserved flags mask:     0 (0x00000000)
0D8     Component type name:
0D9    List (45 bytes)
0D9     Header (8 bytes)
0D9      Size:                      45 (0x0000002D)
0DD      Name:                        ilst
0E1     Element (37 bytes)
0E1      Header (8 bytes)
0E1       Size:                     37 (0x00000025)
0E5       Name:                       ? oo
0E9      Data - Encoded_Application (29 bytes)
0E9       Header (8 bytes)
0E9        Size:                    29 (0x0000001D)
0ED        Name:                      data
0F1       Kind:                     1 (0x00000001) - UTF8
0F5       Language:                    0 (0x00000000)
0F9       Value:                    Lavf58.29.100
106 ---------------------------
106 ---   MPEG-4, filling   ---
106 ---------------------------
106 ---------------------------
106 ---   MPEG-4, finished   ---
106 ---------------------------
```

FFmpeg时间戳详解

原文地址：：https://www.cnblogs.com/leisure_chn/p/10584910.html

编者注：相对原文有一定的改动。

1. I帧/P帧/B帧

I帧：I帧(Intra-coded picture，帧内编码帧，常称为关键帧)包含一幅完整的图像信息，属于帧内编码图像，不含运动矢量，在解码时不需要参考其他帧图像。因此在I帧图像处可以切换频道，而不会导致图像丢失或无法解码。I帧图像用于阻止误差的累积和扩散。在闭合式GOP中，每个GOP的第一个帧一定是I帧，且当前GOP的数据不会参考前后GOP的数据。

P帧：P帧(Predictive-coded picture，预测编码图像帧)是帧间编码帧，利用之前的I帧或P帧进行预测编码。

B帧：B帧(Bi-directionally predicted picture，双向预测编码图像帧)是帧间编码帧，利用之前和(或)之后的I帧或P帧进行双向预测编码。B帧不可以作为参考帧。

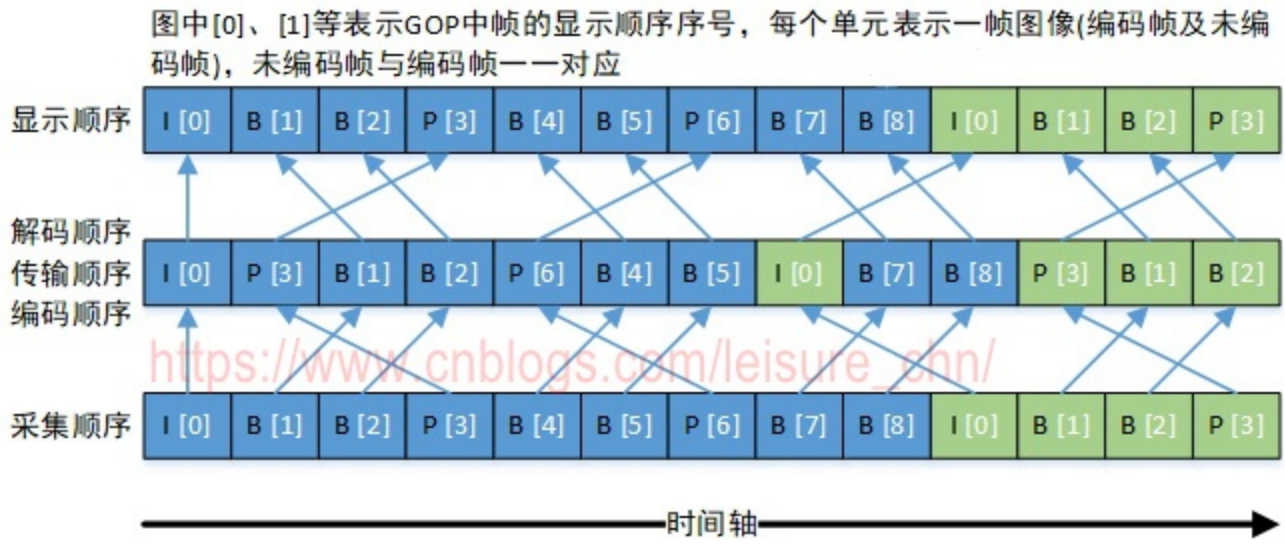
B帧具有更高的压缩率，但需要更多的缓冲时间以及更高的CPU占用率，因此B帧适合本地存储以及视频点播，而不适用对实时性要求较高的直播系统。

2. DTS和PTS

DTS(Decoding Time Stamp，解码时间戳)，表示压缩帧的解码时间。

PTS(Presentation Time Stamp，显示时间戳)，表示将压缩帧解码后得到的原始帧的显示时间。

音频中DTS和PTS是相同的。视频中由于B帧需要双向预测，B帧依赖于其前和其后的帧，因此含B帧的视频解码顺序与显示顺序不同，即DTS与PTS不同。当然，不含B帧的视频，其DTS和PTS是相同的。下图以一个开放式GOP示意图为例，说明视频流的解码顺序和显示顺序



采集顺序指图像传感器采集原始信号得到图像帧的顺序。

编码顺序指编码器编码后图像帧的顺序。存储到磁盘的本地视频文件中图像帧的顺序与编码顺序相同。

传输顺序指编码后的流在网络中传输过程中图像帧的顺序。

解码顺序指解码器解码图像帧的顺序。

显示顺序指图像帧在显示器上显示的顺序。
采集顺序与显示顺序相同。编码顺序、传输顺序和解码顺序相同。
以图中“B[1]”帧为例进行说明，“B[1]”帧解码时需要参考“I[0]”帧和“P[3]”帧，因此“P[3]”帧必须比“B[1]”帧先解码。这就导致了解码顺序和显示顺序的不一致，后显示的帧需要先解码。

3. FFmpeg中的时间基与时间戳

3.1 时间基与时间戳的概念

在FFmpeg中，时间基(time_base)是时间戳(timestamp)的单位，时间戳值乘以时间基，可以得到实际的时刻值(以秒等为单位)。例如，如果一个视频帧的dts是40，pts是160，其time_base是1/1000秒，那么可以计算出此视频帧的解码时刻是40毫秒(40/1000)，显示时刻是160毫秒(160/1000)。FFmpeg中时间戳(pts/dts)的类型是int64_t类型，把一个time_base看作一个时钟脉冲，则可把dts/pts看作时钟脉冲的计数。

3.2 三种时间基tbr、tbn和tbc

不同的封装格式具有不同的时间基。在FFmpeg处理音视频过程中的不同阶段，也会采用不同的时间基。
FFmepg中有三种时间基，命令行中tbr、tbn和tbc的打印值就是这三种时间基的倒数：
tbn：对应容器中的时间基。值是AVStream.time_base的倒数
tbc：对应编解码器中的时间基。值是AVCodecContext.time_base的倒数
tbr：从视频流中猜算得到，可能是帧率或场率(帧率的2倍)
测试文件下载(右键另存为)：tnmil3.flv
使用ffprobe探测媒体文件格式，如下：

```
think@opensuse> ffprobe tnmil3.flv
ffprobe version 4.1 Copyright (c) 2007-2018 the FFmpeg developers
Input #0, flv, from 'tnmil3.flv':
  Metadata:
    encoder         : Lavf58.20.100
  Duration: 00:00:03.60, start: 0.017000, bitrate: 513 kb/s
    Stream #0:0: Video: h264 (High), yuv420p(progressive), 784x480, 2
 5 fps, 25 tbr, 1k tbn, 50 tbc
    Stream #0:1: Audio: aac (LC), 44100 Hz, stereo, fltp, 128 kb/s
```

关于tbr、tbn和tbc的说明，原文如下，来自FFmpeg邮件列表：
There are three different time bases for time stamps in FFmpeg. The values printed are actually reciprocals of these, i.e. 1/tbr, 1/tbn and 1/tbc.

- tbn is the time base in AVStream that has come from the container, I think. It is used for all AVStream time stamps.
- tbc is the time base in AVCodecContext for the codec used for a particular stream. It is used for all AVCodecContext and related time stamps.

- tbr is guessed from the video stream and is the value users want to see when they look for the video frame rate, except sometimes it is twice what one would expect because of field rate versus frame rate.

3.3 内部时间基AV_TIME_BASE

除以上三种时间基外，FFmpeg还有一个内部时间基AV_TIME_BASE(以及分数形式的AV_TIME_BASE_Q)

```
// Internal time base represented as integer
#define AV_TIME_BASE            1000000    //微妙
// Internal time base represented as fractional value
#define AV_TIME_BASE_Q          (AVRational){1, AV_TIME_BASE}
```

AV_TIME_BASE及AV_TIME_BASE_Q用于FFmpeg内部函数处理，使用此时间基计算得到时间值表示的是微秒。

3.4 时间值形式转换

av_q2d()将时间从AVRational形式转换为double形式。AVRational是分数类型，double是双精度浮点数类型，转换的结果单位是秒。转换前后的值基于同一时间基，仅仅是数值的表现形式不同而已。

av_q2d()实现如下：

```
/**
 * Convert an AVRational to a `double`.
 * @param a AVRational to convert
 * @return `a` in floating-point form
 * @see av_d2q()
 */
static inline double av_q2d(AVRational a){
    return a.num / (double) a.den;
}
```

av_q2d()使用方法如下：

```
AVStream stream;
AVPacket packet;
packet播放时刻值：timestamp(单位秒) = packet.pts * av_q2d(stream.time_base);
packet播放时长值：duration(单位秒) = packet.duration * av_q2d(stream.time_base);
```

3.5 时间基转换函数

av_rescale_q

av_rescale_q()用于不同时间基的转换，用于将时间值从一种时间基转换为另一种时间基。
将a数值由 bq时间基转成 cq的时间基，通过返回结果获取以cq时间基表示的新数值。

```
/**
 * Rescale a 64-bit integer by 2 rational numbers.
 *
 * The operation is mathematically equivalent to `a × bq / cq`.
 *
 * This function is equivalent to av_rescale_q_rnd() with #AV_ROUND_
NEAR_INF.
 *
 * @see av_rescale(), av_rescale_rnd(), av_rescale_q_rnd()
 */
int64_t av_rescale_q(int64_t a, AVRational bq, AVRational cq) av_con
st;
```

av_rescale_rnd

int64_t av_rescale_rnd(int64_t a, int64_t b, int64_t c, enum AVRounding rnd);
它的作用是计算 "a * b / c" 的值并分五种方式来取整 （具体的用途用哪个不用那么纠结，基本上开发的时候不）.

- AV_ROUND_ZERO = 0, // Round toward zero. 趋近于0， round(2.5) 为 2, 而 round(-2.5) 为 -2
- AV_ROUND_INF = 1, // Round away from zero. 趋远于0 round(3.5)=4, round(-3.5)=-4
- AV_ROUND_DOWN = 2, // Round toward -infinity.向负无穷大方向 [-2.9, -1.2, 2.4, 5.6, 7.0, 2.4] -> [-3, -2, 2, 5, 7, 2]
- AV_ROUND_UP = 3, // Round toward +infinity. 向正无穷大方向[-2.9, -1.2, 2.4, 5.6, 7.0, 2.4] -> [-2, -1, 3, 6, 7, 3]
- AV_ROUND_NEAR_INF = 5, // Round to nearest and halfway cases away from zero. // 四舍五入,小于0.5取值趋向0,大于0.5取值趋远于0

av_packet_rescale_ts()用于将AVPacket中各种时间值从一种时间基转换为另一种时间基。

```
/**
 * Convert valid timing fields (timestamps / durations) in a packet
 from one
```

```
 * timebase to another. Timestamps with unknown values (AV_NOPTS_VALUE) will be
 * ignored.
 *
 * @param pkt packet on which the conversion will be performed
 * @param tb_src source timebase, in which the timing fields in pkt are
 *               expressed
 * @param tb_dst destination timebase, to which the timing fields will be
 *               converted
 */
void av_packet_rescale_ts(AVPacket *pkt, AVRational tb_src, AVRational tb_dst);
```

3.6 转封装过程中的时间基转换

容器中的时间基(AVStream.time_base，3.2节中的tbn)定义如下：

```
typedef struct AVStream {
    ......
    /**
     * This is the fundamental unit of time (in seconds) in terms
     * of which frame timestamps are represented.
     *
     * decoding: set by libavformat
     * encoding: May be set by the caller before avformat_write_header() to
     *           provide a hint to the muxer about the desired timebase. In
     *           avformat_write_header(), the muxer will overwrite this field
     *           with the timebase that will actually be used for the timestamps
     *           written into the file (which may or may not be related to the
     *           user-provided one, depending on the format).
     */
    AVRational time_base;
    ......
```

```
17 }
```

AVStream.time_base是AVPacket中pts和dts的时间单位，输入流与输出流中time_base按如下方式确定：

- 对于输入流：打开输入文件后，调用avformat_find_stream_info()可获取到每个流中的time_base
- 对于输出流：打开输出文件后，调用avformat_write_header()可根据输出文件封装格式确定每个流的time_base并写入输出文件中

不同封装格式具有不同的时间基，在转封装(将一种封装格式转换为另一种封装格式)过程中，时间基转换相关代码如下：

```
1 av_read_frame(ifmt_ctx, &pkt);
2 pkt.pts = av_rescale_q_rnd(pkt.pts, in_stream->time_base, out_stream-
  >time_base,
3                            AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX);
4 pkt.dts = av_rescale_q_rnd(pkt.dts, in_stream->time_base, out_stream-
  >time_base,
5                            AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX);
6 pkt.duration = av_rescale_q(pkt.duration, in_stream->time_base, out_s
  tream->time_base);
```

下面的代码具有和上面代码相同的效果：

```
1 // 从输入文件中读取packet
2 av_read_frame(ifmt_ctx, &pkt);
3 // 将packet中的各时间值从输入流封装格式时间基转换到输出流封装格式时间基
4 av_packet_rescale_ts(&pkt, in_stream->time_base, out_stream->time_bas
  e);
```

这里流里的时间基`in_stream->time_base`和`out_stream->time_base`，是容器中的时间基，就是3.2节中的tbn。

例如，flv封装格式的time_base为{1,1000}，ts封装格式的time_base为{1,90000}

我们编写程序将flv封装格式转换为ts封装格式，抓取原文件(flv)的前四帧显示时间戳：

```
1 think@opensuse> ffprobe -show_frames -select_streams v tnmil3.flv |
  grep pkt_pts
2 ffprobe version 4.1 Copyright (c) 2007-2018 the FFmpeg developers
3 Input #0, flv, from 'tnmil3.flv':
4   Metadata:
5     encoder         : Lavf58.20.100
6   Duration: 00:00:03.60, start: 0.017000, bitrate: 513 kb/s
7     Stream #0:0: Video: h264 (High), yuv420p(progressive), 784x480,
```

```
   25 fps, 25 tbr, 1k tbn, 50 tbc
    Stream #0:1: Audio: aac (LC), 44100 Hz, stereo, fltp, 128 kb/s
pkt_pts=80
pkt_pts_time=0.080000
pkt_pts=120
pkt_pts_time=0.120000
pkt_pts=160
pkt_pts_time=0.160000
pkt_pts=200
pkt_pts_time=0.200000
```

再抓取转换的文件(ts)的前四帧显示时间戳：

```
think@opensuse> ffprobe -show_frames -select_streams v tnmil3.ts | grep pkt_pts
ffprobe version 4.1 Copyright (c) 2007-2018 the FFmpeg developers
Input #0, mpegts, from 'tnmil3.ts':
  Duration: 00:00:03.58, start: 0.017000, bitrate: 619 kb/s
  Program 1
    Metadata:
      service_name    : Service01
      service_provider: FFmpeg
    Stream #0:0[0x100]: Video: h264 (High) ([27][0][0][0] / 0x001B), yuv420p(progressive), 784x480, 25 fps, 25 tbr, 90k tbn, 50 tbc
    Stream #0:1[0x101]: Audio: aac (LC) ([15][0][0][0] / 0x000F), 44100 Hz, stereo, fltp, 127 kb/s
pkt_pts=7200
pkt_pts_time=0.080000
pkt_pts=10800
pkt_pts_time=0.120000
pkt_pts=14400
pkt_pts_time=0.160000
pkt_pts=18000
pkt_pts_time=0.200000
```

可以发现，对于同一个视频帧，它们时间基(tbn)不同因此时间戳(pkt_pts)也不同，但是计算出来的时刻值(pkt_pts_time)是相同的。

看第一帧的时间戳，计算关系：80×{1,1000} == 7200×{1,90000} == 0.080000

3.7 转码过程中的时间基转换

编解码器中的时间基(AVCodecContext.time_base，3.2节中的tbc)定义如下：

```
typedef struct AVCodecContext {
    ......

    /**
     * This is the fundamental unit of time (in seconds) in terms
     * of which frame timestamps are represented. For fixed-fps cont
ent,
     * timebase should be 1/framerate and timestamp increments shoul
d be
     * identically 1.
     * This often, but not always is the inverse of the frame rate o
r field rate
     * for video. 1/time_base is not the average frame rate if the f
rame rate is not
     * constant.
     *
     * Like containers, elementary streams also can store timestamps
, 1/time_base
     * is the unit in which these timestamps are specified.
     * As example of such codec time base see ISO/IEC 14496-2:2001(E
)
     * vop_time_increment_resolution and fixed_vop_rate
     * (fixed_vop_rate == 0 implies that it is different from the fr
amerate)
     *
     * - encoding: MUST be set by user.
     * - decoding: the use of this field for decoding is deprecated.
     *             Use framerate instead.
     */
    AVRational time_base;

    ......
}
```

上述注释指出，AVCodecContext.time_base**是帧率(视频帧)的倒数**，每帧时间戳递增1，那么tbc就等于帧率。编码过程中，应由用户设置好此参数。解码过程中，此参数已过时，建议直接使用帧率倒数用作时间基。

这里有一个问题：按照此处注释说明，帧率为25的视频流，tbc理应为25，但实际值却为50，不知作何解释？是否tbc已经过时，不具参考意义？

根据注释中的建议，实际使用时，在视频解码过程中，我们不使用AVCodecContext.time_base，而用帧率倒数作时间基，在视频编码过程中，我们将AVCodecContext.time_base设置为帧率的倒数。

3.7.1 视频流

视频按帧播放，所以解码后的原始视频帧时间基为 1/framerate。

视频解码过程中的时间基转换处理（darren注：该段没有参考意义，packet的pts到底什么，要看实际的情况，从av_read_frame读取的packet，是以AVSteam->time_base，送给解码器之前没有必要转成AVcodecContext->time_base， 需要注意的是avcodec_receive_frame后以AVSteam->time_base为单位即可。**）**：

```
AVFormatContext *ifmt_ctx;
AVStream *in_stream;
AVCodecContext *dec_ctx;
AVPacket packet;
AVFrame *frame;
// 从输入文件中读取编码帧
av_read_frame(ifmt_ctx, &packet);
// 时间基转换
int raw_video_time_base = av_inv_q(dec_ctx->framerate);
av_packet_rescale_ts(packet, in_stream->time_base, raw_video_time_base);
// 解码
avcodec_send_packet(dec_ctx, packet)
avcodec_receive_frame(dec_ctx, frame);
```

视频编码过程中的时间基转换处理（darren：编码的时候frame如果以AVstream为time_base送编码器，则avcodec_receive_packet读取的时候也是以转成AVSteam->time_base，**本质来讲就是具体情况具体分析，没必要硬套流程**）：

```
AVFormatContext *ofmt_ctx;
AVStream *out_stream;
AVCodecContext *dec_ctx;
AVCodecContext *enc_ctx;
AVPacket packet;
AVFrame *frame;
// 编码
avcodec_send_frame(enc_ctx, frame);
avcodec_receive_packet(enc_ctx, packet);
// 时间基转换
packet.stream_index = out_stream_idx;
enc_ctx->time_base = av_inv_q(dec_ctx->framerate);
```

```
av_packet_rescale_ts(&opacket, enc_ctx->time_base, out_stream->time_base);
// 将编码帧写入输出媒体文件
av_interleaved_write_frame(o_fmt_ctx, &packet);
```

3.7.2 音频流

darren：对于音频流也是类似的，**本质来讲就是具体情况具体分析，没必要硬套流程，比如ffplay 解码播放时就是AVSteam的time_base为基准的packet进入到编码器，然后出来的frame再用AVSteam的time_base讲对应的pts转成秒，** **但是要注意的是ffplay做了一个比较隐秘的设置：**avctx->pkt_timebase = ic->streams[stream_index]->time_base; 即是对应的codeccontext自己对pkt_timebase设置和AVStream一样的time_base。

音频按采样点播放，所以解码后的原始音频帧时间基为 1/sample_rate
音频解码过程中的时间基转换处理：

```
AVFormatContext *ifmt_ctx;
AVStream *in_stream;
AVCodecContext *dec_ctx;
AVPacket packet;
AVFrame *frame;
// 从输入文件中读取编码帧
av_read_frame(ifmt_ctx, &packet);
// 时间基转换
int raw_audio_time_base = av_inv_q(dec_ctx->sample_rate);
av_packet_rescale_ts(packet, in_stream->time_base, raw_audio_time_base);
// 解码
avcodec_send_packet(dec_ctx, packet)
avcodec_receive_frame(dec_ctx, frame);
```

音频编码过程中的时间基转换处理：

```
AVFormatContext *ofmt_ctx;
AVStream *out_stream;
AVCodecContext *dec_ctx;
AVCodecContext *enc_ctx;
AVPacket packet;
AVFrame *frame;
// 编码
avcodec_send_frame(enc_ctx, frame);
```

```
avcodec_receive_packet(enc_ctx, packet);
// 时间基转换
packet.stream_index = out_stream_idx;
enc_ctx->time_base = av_inv_q(dec_ctx->sample_rate);
av_packet_rescale_ts(&opacket, enc_ctx->time_base, out_stream->time_base);
// 将编码帧写入输出媒体文件
av_interleaved_write_frame(o_fmt_ctx, &packet);
```