11-音视频同步基础

《FFmpeg/WebRTC/RTMP音视频流媒体高级开发教程》 – Darren老师：QQ326873713
课程链接：https://ke.qq.com/course/468797?tuin=137bb271

11 音视频同步基础

由于音频和视频的输出不在同一个线程，而且，也不一定会同时解出同一个pts的音频帧和视频帧。更有甚者，编码或封装的时候可能pts还是不连续的，或有个别错误的。因此，在进行音频和视频的播放时，需要对音频和视频的播放速度、播放时刻进行控制，以实现音频和视频保持同步，即所谓的音视频同步。
在ffplay中，音频（audio）和视频（video）有各自的输出线程，其中音频的输出线程是sdl的音频输出回调线程，video的输出线程是程序的主线程。

11.1 音视频同步策略

音视频的同步策略，一般有如下几种：

- **以音频为基准，同步视频到音频**（AV_SYNC_AUDIO_MASTER）
  - 视频慢了则丢掉部分视频帧（视觉->画面跳帧）
  - 视频快了则继续渲染上一帧
- **以视频为基准，同步音频到视频**（AV_SYNC_VIDEO_MASTER）
  - 音频慢了则加快播放速度（或丢掉部分音频帧，丢帧极容易听出来断音）
  - 音频快了则放慢播放速度（或重复上一帧 ）
  - 音频改变播放速度时涉及到重采样

- **以外部时钟为基准，同步音频和视频到外部时钟**（AV_SYNC_EXTERNAL_CLOCK）
  - 前两者的综合，根据外部时钟改变播放速度
- **视频和音频各自输出，即不作同步处理（FREE RUN）**

由于人耳对于声音变化的敏感度比视觉高，因此，一般采样的策略是将视频同步到音频，即对画面进行适当的丢帧或重复以追赶或等待音频。
特殊地，有时候会碰到一些特殊封装（或者有问题的封装），此时就不作同步处理，各自为主时钟，进行播放。

在ffplay中实现了上述前3种的同步策略。由`sync`参数控制：

```
{ "sync", HAS_ARG | OPT_EXPERT, { .func_arg = opt_sync }, "set audio-video sync. type (type=audio/video/ext)", "type" },
```

比如
ffplay source.200kbps.768x320.flv –sync video
设置以video master

11.2 音视频同步概念

在深入代码了解其实现前，需要先简单了解下一些结构体和概念。读者也可以选择在后文阅读中回头查阅本节。

- DTS（Decoding Time Stamp）：即解码时间戳，这个时间戳的意义在于告诉播放器该在什么时候解码这一帧的数据。
- PTS（Presentation Time Stamp）：即显示时间戳，这个时间戳用来告诉播放器该在什么时候显示这一帧的数据。
- timebase 时基：pts的值的真正单位
- ffplay中的pts，ffplay在做音视频同步时使用秒为单位，使用double类型去标识pts，在ffmpeg内部不会用浮点数去标记pts。
- Clock 时钟

当视频流中没有 B 帧时，通常 DTS 和 PTS 的顺序是一致的。但存在B帧的时候两者的顺序就不一致了。

（1）pts是presentation timestamp的缩写，即显示时间戳，用于标记一个帧的呈现时刻，它的单位由timebase决定。timebase的类型是结构体AVRational（用于表示分数）：

```
typedef struct AVRational{
    int num; ///< Numerator
```

```
    int den; ///< Denominator
} AVRational;
```

如`timebase={1, 1000}`表示千分之一秒（毫秒），那么pts=1000，即为pts*1/1000 = 1秒，那么这一帧就需要在第一秒的时候呈现。

将AVRatioal结构转换成double

```
static inline double av_q2d(AVRational a) {
    return a.num / (double) a.den;
}
```

计算时间戳

timestamp(秒) = pts * av_q2d(st->time_base)

计算帧时长

time(秒) = st->duration * av_q2d(st->time_base)

不同时间基之间的转换

int64_t av_rescale_q(int64_t a, AVRational bq, AVRational cq)

在ffplay中，将pts转化为秒，一般做法是：`pts * av_q2d(timebase)`

（2）在做同步的时候，我们需要一个"时钟"的概念，音频、视频、外部时钟都有自己独立的时钟，**各自set各自的时钟，**以谁为基准(master), **其他的则只能get该时钟进行同步，**ffplay定义的结构体是Clock：

```
typedef struct Clock {
    double  pts;            // 时钟基础, 当前帧(待播放)显示时间戳, 播放后, 当前帧变成上一帧
    // 当前pts与当前系统时钟的差值, audio、video对于该值是独立的
    double  pts_drift;      // clock base minus time at which we updated the clock
    // 当前时钟(如视频时钟)最后一次更新时间, 也可称当前时钟时间
    double  last_updated;   // 最后一次更新的系统时钟
    double  speed;          // 时钟速度控制, 用于控制播放速度
    // 播放序列, 所谓播放序列就是一段连续的播放动作, 一个seek操作会启动一段新的播放序列
    int serial;             // clock is based on a packet with this serial
    int paused;             // = 1 说明是暂停状态
```

```
    // 指向packet_serial
    int *queue_serial;      /* pointer to the current packet queue s
erial, used for obsolete clock detection */
} Clock;
```

这个时钟的工作原理是这样的：

1. 需要不断“对时”。对时的方法`set_clock_at(Clock *c, double pts, int serial, double time)`，需要用pts、serial、time（系统时间）进行对时。
2. 获取的时间是一个估算值。估算是通过对时时记录的pts_drift估算的。**pts_drift是最精华的设计，一定要理解。**

可以看这个图来帮助理解：

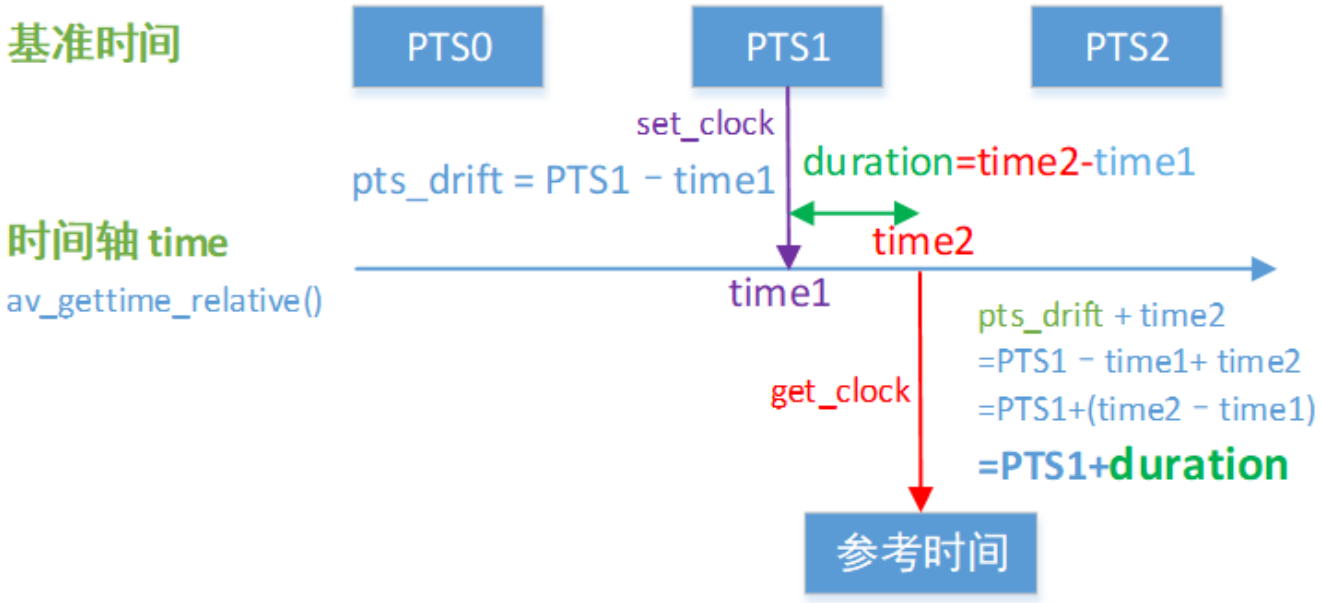
图中央是一个时间轴（time是一直在按时间递增），从左往右看。首先我们调用`set_clock`进行一次对时，假设这时的`pts`是落后时间`time`的，那么计算`pts_drift = pts - time`，计算出pts和time的相对差值。

接着，过了一会儿，且在下次对时前，通过`get_clock`来查询时间，因为set_clock时的`pts`已经过时，不能直接拿set_clock时的pts当做这个时钟的时间。不过我们前面计算过`pts_drift`，也就是`pts`和`time`的差值，所以我们可以通过当前时刻的时间来估算当前时刻的pts：`pts = time + pts_drift`。

> 一般time会取CLOCK_MONOTONIC（单调递增的时钟），即系统开机到现在的时间.

ffplay使用ffmpeg提供的av_gettime_relative()函数。

再来一个图



11.3 FFmpeg中的时间单位

AV_TIME_BASE

- 定义#define AV_TIME_BASE 1 000 000
- ffmpeg中的内部计时单位（时间基）

AV_TIME_BASE_Q

- 定义#define AV_TIME_BASE_Q (AVRational){1, AV_TIME_BASE}
- ffmpeg内部时间基的分数表示，实际上它是AV_TIME_BASE的倒数

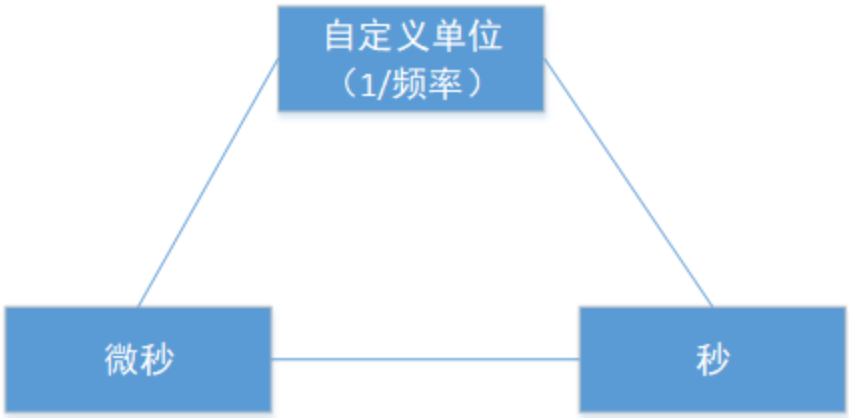
时间基转换公式

- timestamp(ffmpeg内部时间戳) = AV_TIME_BASE * time(秒)
- time(秒) = AV_TIME_BASE_Q * timestamp(ffmpeg内部时间戳)

11.4 音视频时间换算的问题

- 标准时间 秒（seconds）
- 标准时间 微秒（microsecond）
- 自定义时间单位 （a/b 秒）
- 以音频AAC音频帧举例，如果pts以1/采样率为单位，比如44.1khz，则时间单位是1/44100，因此PTS表示：

- 第一帧 PTS1 = 0
- 第二帧 PTS2 = 1024
- 第三帧 PTS2 = 2048
- 播放的时候要将PTS换算成秒的单位，则PTS1 = 0 *1/44100, PTS2 = 1024*1/44100, PTS3 = 2048*1/44100= 0.046439



11.5 不同结构体的time_base/duration分析

ffmpeg存在多个时间基准(time_base)，对应不同的阶段(结构体)，每个time_base具体的值不一样，ffmpeg提供函数在各个time_base中进行切换。

- AVFormatContext
  - duration：整个码流的时长，获取正常时长的时候要除以AV_TIME_BASE,得到的结果单位是秒
- AVStream
  - time_base：单位为秒，比如AAC音频流，他可能是{1,44100}
    TS流，按{1, 90khz}
  - duration：表示该数据流的时长，以AVStream->time_base 为单位

AVStream的time_base是在demuxer或者muxer内设置的，以TS，FLV，MP4为例子：

TS

- avpriv_set_pts_info(st, 33, 1, 90000)　(mpegts.c和mpegtsenc.c)

FLV

- avpriv_set_pts_info(st, 32, 1, 1000)　(flvdec.c)
- avpriv_set_pts_info(s->streams[i], 32, 1, 1000)　(flvenc.c)

MP4

- avpriv_set_pts_info(st, 64, 1, sc->time_scale);　(mov.c)
- avpriv_set_pts_info(st, 64, 1, track->timescale);　(movenc.c)

11.6 不同结构体的PTS/DTS分析

不同结构体下，pts和dts使用哪个time_base来表示?

AVPacket

- pts：以AVStream->time_base为单位
- dts：以AVStream->time_base为单位
- duration：以AVStream->time_base为单位

AVFrame

- pts：以AVStream->time_base为单位
- pkt_pts和pkt_dts：拷贝自AVPacket，同样以AVStream->time_base为单位
- duration：以AVStream->time_base为单位

11.6 ffplay中PTS的转换流程分析

```
typedef struct Frame {
    AVFrame *frame;
    AVSubtitle sub;
    int serial;
    double pts;           /* 时间戳 */
    double duration;      /* 该帧持续时间 */
    int64_t pos;          /* byte position of the frame in the input file */
    int width;
    int height;
    int format;
    AVRational sar;
    int uploaded;
    int flip_v;
} Frame;
```

大家注意到没有，ffplay在重新封装AVFrame的时候自己也有pts和duration两个变量

- 为什么需要这两个变量？而且是浮点型的。
- 这两个变量的单位又是什么？（秒）

Video Frame PTS的获取

PTS校正

frame->pts = frame->best_effort_timestamp;

这里为什么不用AVFrame中的pts来直接计算呢？其实大多数情况下AVFrame的pts和best_effort_timestamp值是一样的

```
/**
 * frame timestamp estimated using various heuristics, in stream time base
 * - encoding: unused
 * - decoding: set by libavcodec, read by user.
 */
int64_t best_effort_timestamp;
```

Audio Frame PTS的获取

ffplay有3次对于Audio的pts进行转换

第一次 将其由AVStrean->time_base转换为（1/采样率）

- frame->pts = av_rescale_q(frame->pts, d->avctx->pkt_timebase, tb);

第二次 将其由（1/采样率）转换为秒

- af->pts = (frame->pts == AV_NOPTS_VALUE) ? NAN : frame->pts * av_q2d(tb);

第三次 根据实际拷贝给sdl的数据长度做调整

- audio_pts = is->audio_clock -
  (double)(2 * is->audio_hw_buf_size + is->audio_write_buf_size) / is->audio_tgt.bytes_per_sec;