

# ffplay播放器-12-14音视频同步

## 12 以音频为基准

音频主流程

视频主流程

delay的计算

## 13 以视频为基准

视频主流程

音频主流程

synchronize\_audio

swr\_set\_compensation

## 14 以外部时钟为基准

回顾

分析

《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>

## 12 以音频为基准

### 音频主流程

ffplay默认也是采用的这种同步策略。

此时音频的时钟设置在sdl\_audio\_callback:

```
1 audio_callback_time = av_gettime_relative();
2
3 .....
4
5 /* Let's assume the audio driver that is used by SDL has two period
   s. */
6     if (!isnan(is->audio_clock)) {
7         set_clock_at(&is->audclk, is->audio_clock -
8                     (double)(2 * is->audio_hw_buf_size + is->audio_
```

```

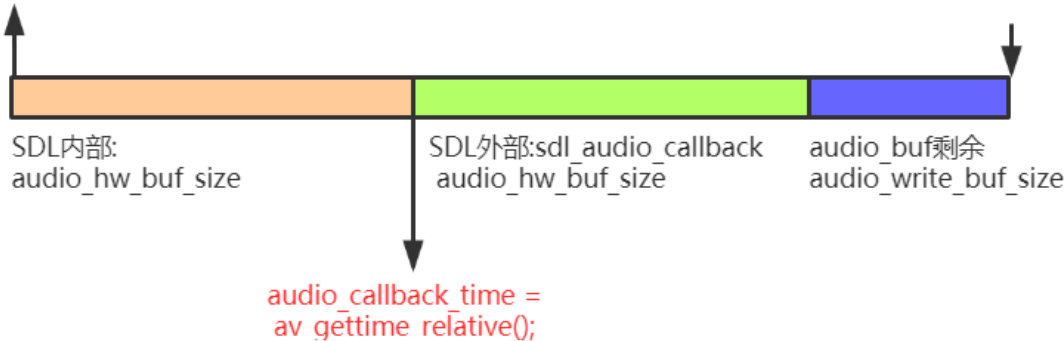
write_buf_size)
9                                     / is->audio_tgt.bytes_per_sec,
10                                 is->audio_clock_serial,
11                                 audio_callback_time / 1000000.0);
12     sync_clock_to_slave(&is->extclk, &is->audclk);
13 }

```

## 音频时钟的维护

实质在播放的数据 pts

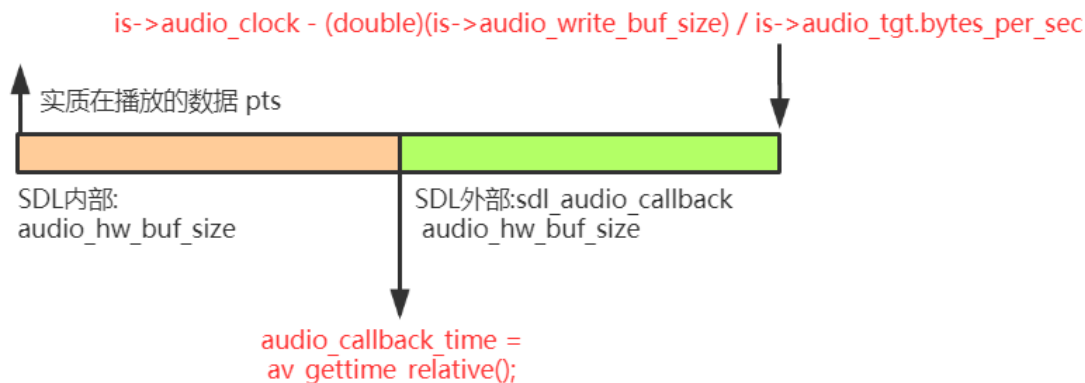
audio\_clock (pts)



我们先来is->audio\_clock是在audio\_decode\_frame赋值：

$is->audio\_clock = af->pts + (double) af->frame->nb\_samples / af->frame->sample\_rate;$

从这里可以看出来，这里的时间戳是audio\_buf结束位置的时间戳，而不是audio\_buf起始位置的时间戳，所以当audio\_buf有剩余时（剩余的长度记录在audio\_write\_buf\_size），那实际数据的pts就变成is->audio\_clock - (double)(is->audio\_write\_buf\_size) / is->audio\_tgt.bytes\_per\_sec，即是



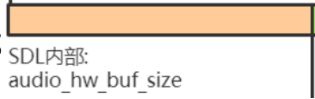
再考虑到，实质上audio\_hw\_buf\_size\*2这些数据实际都没有播放出去，所以就有

$is->audio\_clock - (double)(2 * is->audio\_hw\_buf\_size + is->audio\_write\_buf\_size) / is->audio\_tgt.bytes\_per\_sec。$

再加上我们在SDL回调进行填充时，实际上是有开始被播



放，所以我们这里采用的相对时间是，刚回调产生的，就是内部在播放的时候，那



相对时间实际也在走。

```
static void sdl_audio_callback(void *opaque, Uint8 *stream, int len)
{
    VideoState *is = opaque;
    int audio_size, len1;

    audio_callback_time = av_gettime_relative();
}
```

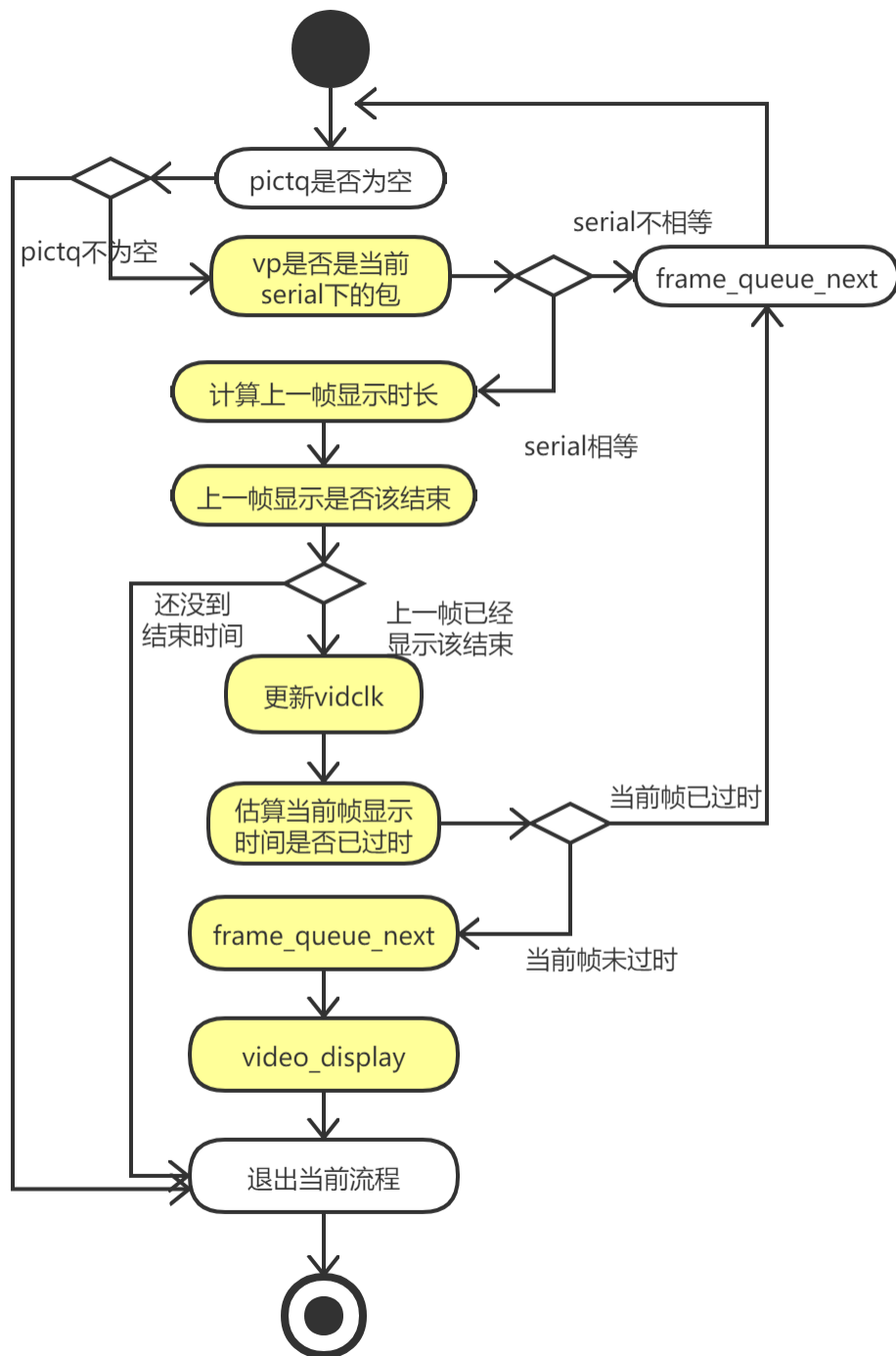
最终

```
set_clock_at(&is->audclk, is->audio_clock - (double)(2 * is->audio_hw_buf_size + is->
audio_write_buf_size) / is->audio_tgt.bytes_per_sec, is->audio_clock_serial,
audio_callback_time / 1000000.0);
```

## 视频主流程

ffplay中将视频同步到音频的主要方案是，如果视频播放过快，则重复播放上一帧，以等待音频；如果视频播放过慢，则丢帧追赶音频。

这一部分的逻辑实现在视频输出函数 `video_refresh` 中，分析代码前，我们先来回顾下这个函数的流程图：



在这个流程中，“计算上一帧显示时长”这一步骤至关重要。先来看下代码：

```

1 static void video_refresh(void *opaque, double *remaining_time)
2 {
3     ...
4     /* compute nominal last_duration */
5     //lastvp上一帧, vp当前帧 , nextvp下一帧

```

```

6      //last_duration 计算上一帧应显示的时长
7      last_duration = vp_duration(is, lastvp, vp);
8
9      // 经过compute_target_delay方法，计算出待显示帧vp需要等待的时间
10     // 如果以video同步，则delay直接等于last_duration。
11     // 如果以audio或外部时钟同步，则需要比对主时钟调整待显示帧vp要等待的时间。
12     delay = compute_target_delay(last_duration, is);
13
14     time= av_gettime_relative()/1000000.0;
15     // is->frame_timer 实际上就是上一帧lastvp的播放时间，
16     // is->frame_timer + delay 是待显示帧vp该播放的时间
17     if (time < is->frame_timer + delay) { //判断是否继续显示上一帧
18         // 当前系统时刻还未到达上一帧的结束时刻，那么还应该继续显示上一帧。
19         // 计算出最小等待时间
20         *remaining_time = FFMIN(is->frame_timer + delay - time, *remaining_time);
21         goto display;
22     }
23
24     // 走到这一步，说明已经到了或过了该显示的时间，待显示帧vp的状态变更为当前要显示的帧
25
26     is->frame_timer += delay; // 更新当前帧播放的时间
27     if (delay > 0 && time - is->frame_timer > AV_SYNC_THRESHOLD_MAX)
28     {
29         is->frame_timer = time; //如果和系统时间差距太大，就纠正为系统时间
30     }
31     SDL_LockMutex(is->pictq.mutex);
32     if (!isnan(vp->pts))
33         update_video_pts(is, vp->pts, vp->pos, vp->serial); // 更新video时钟
34     SDL_UnlockMutex(is->pictq.mutex);
35     //丢帧逻辑
36     if (frame_queue_nb_remaining(&is->pictq) > 1) { //有nextvp才会检测是否该丢帧
37         Frame *nextvp = frame_queue_peek_next(&is->pictq);
38         duration = vp_duration(is, vp, nextvp);
39         if (!is->step // 非逐帧模式才检测是否需要丢帧 is->step==1 为逐帧播放
40             && (framedrop>0 || // cpu解帧过慢

```

```

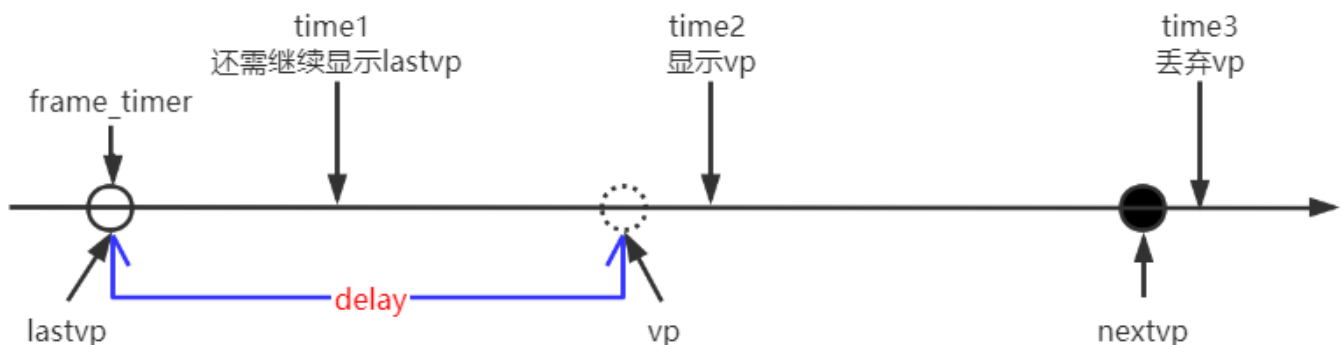
40      (framedrop && get_master_sync_type(is) != AV_SYNC_VID
EO_MASTER)) // 非视频同步方式
41      && time > is->frame_timer + duration // 确实落后了一帧数据
42      ) {
43          printf("%s(%d) dif:%lfs, drop frame\n", __FUNCTION__, __
LINE__,
44                  (is->frame_timer + duration) - time);
45          is->frame_drops_late++; // 统计丢帧情况
46          frame_queue_next(&is->pictq); // 这里实现真正的丢帧
47          //(这里不能直接while丢帧, 因为很可能audio clock重新对时了, 这样de
lay值需要重新计算)
48          goto retry; //回到函数开始位置, 继续重试
49      }
50  }
51  ...
52 }

```

这段代码的逻辑在上述流程图中有包含。主要思路就是一开始提到的：

- 如果视频播放过快，则重复播放上一帧，以等待音频；
- 如果视频播放过慢，则丢帧追赶音频。实现的方式是，参考audio clock，计算上一帧（在屏幕上的那个画面）还应显示多久（含帧本身时长），然后与系统时刻对比，是否该显示下一帧了。

这里与系统时刻的对比，引入了另一个概念——frame\_timer。可以理解为帧显示时刻，如更新前，是上一帧lastvp的显示时刻；对于更新后（`is->frame_timer += delay`），则为当前帧vp显示时刻。上一帧显示时刻加上delay（还应显示多久（含帧本身时长））即为上一帧应结束显示的時刻。具体原理看如下示意图：



这里给出了3种情况的示意图：

- time1: 系统时刻小于lastvp结束显示的时刻 (frame\_timer+dealy) , 即虚线圆圈位置。此时应该继续显示lastvp
- time2: 系统时刻大于lastvp的结束显示时刻, 但小于vp的结束显示时刻 (vp的显示时间开始于虚线圆圈, 结束于黑色圆圈) 。此时既不重复显示lastvp, 也不丢弃vp, 即应显示vp
- time3: 系统时刻大于vp结束显示时刻 (黑色圆圈位置, 也是nextvp预计的开始显示时刻) 。此时应该丢弃vp。

## delay的计算

那么接下来就要看最关键的lastvp的显示时长delay (不是很好理解, 要反复体会) 是如何计算的。

这在函数compute\_target\_delay中实现:

```

1 static double compute_target_delay(double delay, VideoState *is)
2 {
3     double sync_threshold, diff = 0;
4     /* update delay to follow master synchronisation source */
5     if (get_master_sync_type(is) != AV_SYNC_VIDEO_MASTER) {
6         /* if video is slave, we try to correct big delays by
7            duplicating or deleting a frame */
8         diff = get_clock(&is->vidclk) - get_master_clock(is);
9         /* skip or repeat frame. We take into account the
10            delay to compute the threshold. I still don't know
11            if it is the best guess */
12         sync_threshold = FFMAX(AV_SYNC_THRESHOLD_MIN, FFMIN(AV_SYNC_
13 THRESHOLD_MAX, delay));
14         if (!isnan(diff) && fabs(diff) < is->max_frame_duration) {
15             if (diff <= -sync_threshold)
16                 delay = FFMAX(0, delay + diff);
17             else if (diff >= sync_threshold && delay > AV_SYNC_FRAME
18 DUP_THRESHOLD)
19                 delay = delay + diff;
20             else if (diff >= sync_threshold)
21                 delay = 2 * delay;
22         }
23     }
24     av_log(NULL, AV_LOG_TRACE, "video: delay=%0.3f A-V=%f\n",
25         delay, -diff);
26     return delay;
27 }

```

compute\_target\_delay 返回的值越大, 画面越慢 (上一帧持续的时间就越长了)。

这段代码中最难理解的是sync\_threshold, sync\_threshold值范围:

$\text{FFMAX}(\text{AV\_SYNC\_THRESHOLD\_MIN}, \text{FFMIN}(\text{AV\_SYNC\_THRESHOLD\_MAX}, \text{delay}))$ , 其中delay为传入的上一帧播放需要持续的时间(本质是帧持续时间 frame duration), 即是分以下3种情况:

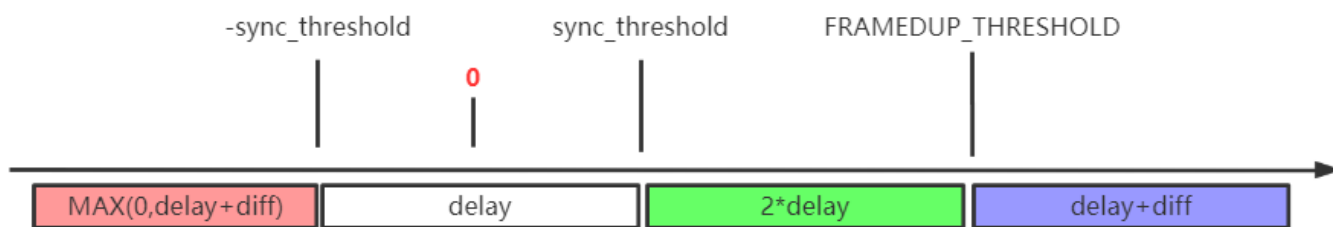
1.  $\text{delay} > \text{AV\_SYNC\_THRESHOLD\_MAX} = 0.1\text{秒}$ , 则 $\text{sync\_threshold} = 0.1\text{秒}$
2.  $\text{delay} < \text{AV\_SYNC\_THRESHOLD\_MIN} = 0.04\text{秒}$ , 则 $\text{sync\_threshold} = 0.04\text{秒}$
3.  $\text{AV\_SYNC\_THRESHOLD\_MIN} = 0.04\text{秒} \leq \text{delay} \leq \text{AV\_SYNC\_THRESHOLD\_MAX} = 0.1\text{秒}$ , 则 $\text{sync\_threshold}$ 为delay本身。

从这里分析也可以看出来, sync\_threshold 最大值为0.1秒, 最小值为0.04秒。这里说明一个说明问题呢?

- 同步精度最好的范围是:  $-0.04\text{秒} \sim +0.04\text{秒}$ ;
- 同步精度最差的范围是:  $-0.1\text{秒} \sim +0.1\text{秒}$ ;

和具体视频的帧率有关系, delay帧间隔 (frame duration) 落在0.04~0.1秒时, 则同步精度为正负1帧。

画个图帮助理解:



图中:

- 坐标轴是diff值大小, diff为0表示video clock与audio clock完全相同, 完美同步。
- 坐标轴下方色块, 表示要返回的值, 色块值的delay指传入参数, 结合上一节代码, 即lastvp的显示时长 (frame duration) 。

从图上可以看出来sync\_threshold是建立一块区域, 在这块区域内无需调整lastvp的显示时长, 直接返回delay即可。也就是在这块区域内认为是准同步的 (sync\_threshold也是最大允许同步误差)。

同步判断结果:

- $\text{diff} \leq -\text{sync\_threshold}$ : 如果小于 $-\text{sync\_threshold}$ , 那就是视频播放较慢, 需要适当丢帧。具体是返回一个最大为0的值。根据前面frame\_timer的图, 至少应更新画面为vp。
- $\text{diff} \geq \text{sync\_threshold} \ \&\& \ \text{delay} > \text{AV\_SYNC\_FRAMEDUP\_THRESHOLD}$ : 如果不仅大于 $\text{sync\_threshold}$ , 而且超过了 $\text{AV\_SYNC\_FRAMEDUP\_THRESHOLD}$ , 那么返回 $\text{delay} + \text{diff}$ , 由具体diff决定还要显示多久 (这里不是很明白代码意图, 按我理解, 统一处理为返回 $2 * \text{delay}$ , 或者 $\text{delay} + \text{diff}$ 即可, 没有区分的必要)



- 此逻辑帧间隔 $\text{delay} > \text{AV\_SYNC\_FRAMEDUP\_THRESHOLD} = 0.1$ 秒，此时 $\text{sync\_threshold} = 0.1$ 秒，那 $\text{delay} + \text{diff} > 0.1 + \text{diff} \geq 0.1 + 0.1 = 0.2$ 秒。
- $\text{diff} \geq \text{sync\_threshold}$ ：如果大于 $\text{sync\_threshold}$ ，那么视频播放太快，需要适当重复显示 $\text{lastvp}$ 。具体是返回 $2 * \text{delay}$ ，也就是2倍的 $\text{lastvp}$ 显示时长，也就是让 $\text{lastvp}$ 再显示一帧。
  - 此逻辑一定是  $\text{delay} \leq 0.1$ 秒， $2 * \text{delay} \leq 0.2$ 秒
- $-\text{sync\_threshold} < \text{diff} < +\text{sync\_threshold}$ ：允许误差内，按 $\text{frame duration}$ 去显示视频，即返回 $\text{delay}$ 。

比如写这段代码的作也表示：I still don't know if it is the best guess

至此，基本上分析完了视频同步音频的过程，简单总结下：

- 基本策略是：如果视频播放过快，则重复播放上一帧，以等待音频；如果视频播放过慢，则丢帧追赶音频。
- 这一策略的实现方式是：引入 $\text{frame\_timer}$ 概念，标记帧的显示时刻和应结束显示的时刻，再与系统时刻对比，决定重复还是丢帧。
- $\text{lastvp}$ 的应结束显示的时刻，除了考虑这一帧本身的显示时长，还应考虑了 $\text{video clock}$ 与 $\text{audio clock}$ 的差值。
- 并不是每时每刻都在同步，而是有一个“准同步”的差值区域。

## 13 以视频为基准

媒体流里面只有视频成分，这个时候才会用以视频为基准。

在“视频同步音频”的策略中，我们是通过丢帧或重复显示的方法来达到追赶或等待音频时钟的目的，但在“音频同步视频”时，却不能这样简单处理。

在音频输出时，最小单位是“样本”。音频一般以数字采样值保存，一般常用的采样频率有44.1K，48K等，也就是每秒钟有44100或48000个样本。视频输出中与“样本”概念最为接近的画面帧，如一个24fps(frame per second)的视频，一秒钟有24个画面输出，这里的一个画面和音频中的一个样本是等效的。可以想见，如果对音频使用一样的丢帧（丢样本）和重复显示方案，是不科学的。（音频的连续性远高于视频，通过重复几百个样本或者丢弃几百个样本来达到同步，会在听觉有很明显的不连贯）

音频本质上来讲：就是做重采样补偿，音频慢了，重采样后的样本就比正常的减少，以赶紧播放下一帧；音频快了，重采样后的样本就比正常的增加，从而播放慢一些。

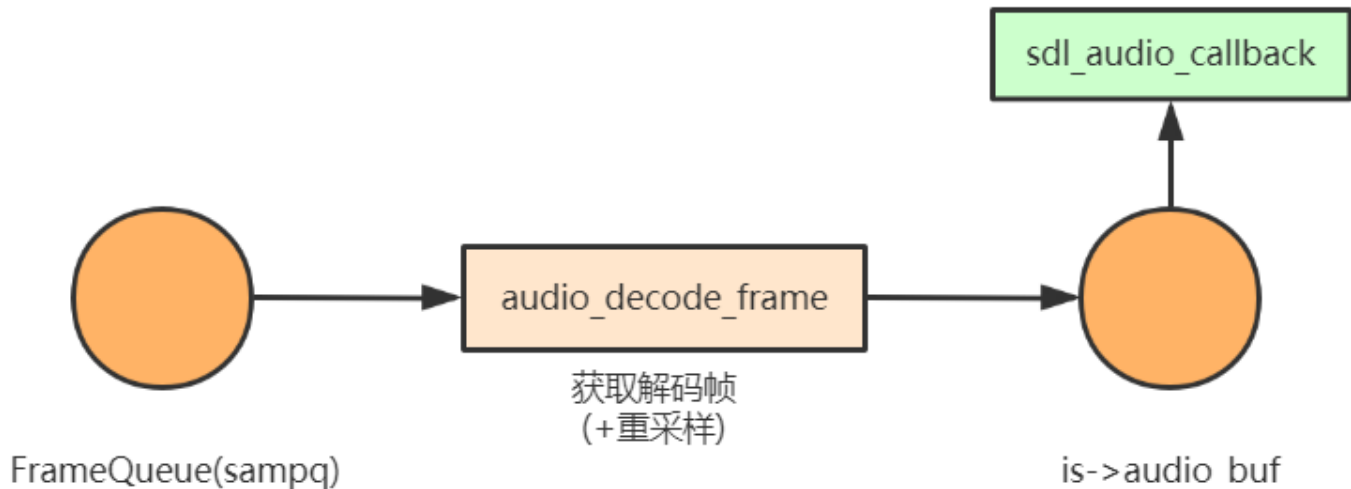
## 视频主流程

video\_refresh()-> update\_video\_pts() 按照着视频帧间隔去播放，并实时地重新矫正video时钟。  
重点主要在audio的播放。

## 音频主流程

在分析具体的补偿方法的之前，先回顾下音频输出的流程。

在《音频输出》章节我们分析过，音频输出的主要模型是：



在 `audio_buf` 缓冲不足时，`audio_decode_frame` 会从 `FrameQueue` 中取出数据放入 `audio_buf`。  
`audio_decode_frame` 函数有音视频同步相关的控制代码：

```
1 //为了方便阅读，以下代码经过简化，只保留音视频同步相关代码
2 static int audio_decode_frame(VideoState *is)
3 {
4     //1. 根据与video clock的差值，计算应该输出的样本数
5     wanted_nb_samples = synchronize_audio(is, af->frame->nb_samples)
6     ;
7     //2. 判断是否需要重采样：如果要输出的样本数与frame的样本数不相等，也就是需要适
8     当缩减或添加样本了
9     if (wanted_nb_samples != af->frame->nb_samples && !is->swr
10     _ctx){
11         //创建重采样ctx
12         is->swr_ctx = swr_alloc_set_opts(NULL,
13             is->audio_tgt.channel_layout,
14             is->audio_tgt.fmt, is->audi
15             o_tgt.freq,
16             dec_channel_layout,
17             af->frame->format, af->fram
```

```

    e->sample_rate,
14                                     0, NULL);
15     if (!is->swr_ctx || swr_init(is->swr_ctx) < 0) {
16         return -1;
17     }
18 }
19 //3. 重采样, 利用重采样库进行样本的插入或删除
20 if (is->swr_ctx) {
21     uint8_t **out = &is->audio_buf1;
22     int out_count = (int64_t)wanted_nb_samples * is->audio_tgt.f
req / af->frame->sample_rate + 256;
23     if (wanted_nb_samples != af->frame->nb_samples) {
24         if (swr_set_compensation(is->swr_ctx,
25                                 (wanted_nb_samples - af->frame-
>nb_samples) * is->audio_tgt.freq / af->frame->sample_rate,
26                                 wanted_nb_samples * is->audio_t
gt.freq / af->frame->sample_rate) < 0) {
27             return -1;
28         }
29     }
30     len2 = swr_convert(is->swr_ctx, out, out_count, in, af->fram
e->nb_samples);
31     is->audio_buf = is->audio_buf1;
32     resampled_data_size = len2 * is->audio_tgt.channels * av_get
_bytes_per_sample(is->audio_tgt.fmt);
33 }
34 else { //如果重采样ctx没有初始化过, 说明无需做同步(这里不考虑音频源格式设备不
支持的情况)
35     is->audio_buf = af->frame->data[0];
36     resampled_data_size = data_size;
37 }
38 return resampled_data_size;
39 }

```

主要分3个步骤:

1. 根据与video clock的差值, 计算应该输出的样本数。由函数 `synchronize_audio` 完成:
  - a. 音频慢了则样本数减少
  - b. 音频快了则样本数增加
2. 判断是否需要重采样: 如果要输出的样本数与frame的样本数不相等, 也就是需要适当减少或增加样本。

### 3. 重采样——利用重采样库进行样本的插入或删除

可以看到，与视频的处理略有不同，视频的同步控制主要体现在上一帧显示时长的控制，即对 `frame_timer` 的控制；而音频是直接体现在输出样本上的控制。

前面提到如果单纯判断某个时刻应该重复样本或丢弃样本，然后对输出音频进行修改，人耳会很容易感知到这一不连贯，体验不好。

这里的处理方式是利用重采样库进行平滑地样本剔除或添加。即在获知要调整的目标样本数 `wanted_nb_samples` 后，通过 `swr_set_compensation` 和 `swr_convert` 的函数组合完成“重采样”。

需要注意的是，因为增加或删除了样本，样本总数发生了变化，而采样率不变，那么假设原先1s的声音将被以大于1s或小于1s的时长进行播放，这会导致声音整体频率被拉低或拉高。直观感受，就是声音变粗或变尖了。ffplay也考虑到了这点影响，其做法是设定一个最大、最小调整范围，避免大幅度的音调变化。

## synchronize\_audio

在了解了整体流程后，就来看下关键函数：`synchronize_audio`

`synchronize_audio` 负责根据与 video clock 的差值计算出合适的目标样本数，通过样本数控制音频输出速度。

现在让我们看看当 N 组音频采样已经不同步的情况。而这些音频采样不同步的程度也有很大的不同，所以我们要取平均值来衡量每个采样的不同步情况。比如，第一次调用时显示我们不同步了 40ms，下一次是 50ms，等等。但是我们会采取简单的平均计算，因为最近的值比之前的值更重要也更有意义，这时候我们会使用一个小数系数 `audio_diff_cum`，并对不同步的延时求和：`is->audio_diff_cum = diff + is->audio_diff_avg_coef * is->audio_diff_cum`；当我们找到平均差异值时，我们就简单的计算 `avg_diff = is->audio_diff_cum * (1.0 - is->audio_diff_avg_coef)`；。我们代码如下：

```
1 static int synchronize_audio(AudioState *is, int nb_samples)
2 {
3     int wanted_nb_samples = nb_samples;
4     /* if not master, then we try to remove or add samples to correct the clock */
5     if (get_master_sync_type(is) != AV_SYNC_AUDIO_MASTER) {
6         double diff, avg_diff;
7         int min_nb_samples, max_nb_samples;
8         diff = get_clock(&is->audclk) - get_master_clock(is);
9         if (!isnan(diff) && fabs(diff) < AV_NOSYNC_THRESHOLD) {
10             is->audio_diff_cum = diff + is->audio_diff_avg_coef * is->audio_diff_cum;
11             if (is->audio_diff_avg_count < AUDIO_DIFF_AVG_NB) {
```

```

12         /* not enough measures to have a correct estimate */
13         is->audio_diff_avg_count++;
14     } else {
15         /* estimate the A-V difference */
16         avg_diff = is->audio_diff_cum * (1.0 - is->audio_diff_avg_coef);
17         if (fabs(avg_diff) >= is->audio_diff_threshold) {
18             wanted_nb_samples = nb_samples + (int)(diff * is->audio_src.freq);
19             min_nb_samples = ((nb_samples * (100 - SAMPLE_CORRECTION_PERCENT_MAX) / 100));
20             max_nb_samples = ((nb_samples * (100 + SAMPLE_CORRECTION_PERCENT_MAX) / 100));
21             wanted_nb_samples = av_clip(wanted_nb_samples, min_nb_samples, max_nb_samples);
22         }
23         av_log(NULL, AV_LOG_TRACE, "diff=%f adiff=%f sample_diff=%d apts=%0.3f %f\n",
24             diff, avg_diff, wanted_nb_samples - nb_samples,
25             is->audio_clock, is->audio_diff_threshold);
26     }
27 } else {
28     /* too big difference : may be initial PTS errors, so
29     reset A-V filter */
30     is->audio_diff_avg_count = 0;
31     is->audio_diff_cum = 0;
32 }
33 }
34 return wanted_nb_samples;
35 }

```

和 `compute_target_delay` 一样，这个函数的源码注释也是ffplay里算多的。

这里首先得先理解一个“神奇的算法”。这里有一组变量 `audio_diff_avg_coef`、

`audio_diff_avg_count`、`audio_diff_cum`、`avg_diff`。我们会发现在开始播放的

AUDIO\_DIFF\_AVG\_NB (20) 个帧内，都是在通过公式 `is->audio_diff_cum = diff + is->`

`>audio_diff_avg_coef * is->audio_diff_cum`；计算累加值 `audio_diff_cum`。按注释的意思

是为了得到一个准确的估计值。接着在后面计算与主时钟的差值时，并不是直接求当前时刻的差值，而

是根据累加值计算一个平均值：`avg_diff = is->audio_diff_cum * (1.0 - is->`

`>audio_diff_avg_coef)`；，然后通过这个均值进行校正。

翻阅了一些资料，这个公式的目的应该是为了让越靠近当前时刻的diff值在平均值中的权重越大，不过还没找到对应的数学公式及含义。

继续看，在计算得到`avg_diff`后，如何确定要输出的样本数：

```
1 wanted_nb_samples = nb_samples + (int)(diff * is->audio_src.freq);
2 min_nb_samples = ((nb_samples * (100 - SAMPLE_CORRECTION_PERCENT_MAX)
  / 100));
3 max_nb_samples = ((nb_samples * (100 + SAMPLE_CORRECTION_PERCENT_MAX)
  / 100));
4 wanted_nb_samples = av_clip(wanted_nb_samples, min_nb_samples, max_nb
  _samples);
```

时间差值乘以采样率可以得到用于补偿的样本数，加之原样本数，即应输出样本数。另外考虑到上一节提到的音频音调变化问题，**这里限制了调节范围在正负10%以内**。

所以如果音视频不同步的差值较大，并不会立即完全同步，最多只调节当前帧样本数的10%，剩余会在下次调节时继续校正。

最后，是与视频同步音频时类似地，有一个准同步的区间，在这个区间内不去做同步校正，其大小是`audio_diff_threshold`：

```
1 is->audio_diff_threshold = (double)(is->audio_hw_buf_size) / is->audi
  o_tgt.bytes_per_sec;
```

即音频输出设备内缓冲的音频时长。

以上，就是音频去同步视频时的主要逻辑。简单总结如下：

1. 音频追赶、等待视频采样的方法是直接调整输出样本数量
2. 调整输出样本时为避免听觉上不连贯的体验，使用了重采样库进行音频的剔除和添加
3. 计算校正后输出的样本数量，使用了一个“神奇的公式”，其意义和含义还有待进一步确认

## swr\_set\_compensation

/\*\*

\* Activate resampling compensation ("soft" compensation). This function is  
\* internally called when needed in `swr_next_pts()`.

\*

\* @param[in,out] s allocated Swr context. If it is not initialized,  
\* or `SWR_FLAG_RESAMPLE` is not set, `swr_init()` is  
\* called with the flag set.

\* @param[in] sample\_delta delta in PTS per sample

\* @param[in] compensation\_distance number of samples to compensate for

```

* @return  >= 0 on success, AERROR error codes if:
*
*         @li @c s is NULL,
*         @li @c compensation_distance is less than 0,
*         @li @c compensation_distance is 0 but sample_delta is not,
*         @li compensation unsupported by resampler, or
*         @li swr_init() fails when called.
*/
int swr_set_compensation(struct SwrContext *s, int sample_delta, int compensation_distance);

```

激活重采样补偿（“软”补偿）。

在swr\_next\_pts（）中需要时，内部调用此函数。

参数：s：分配Swr上下文。如果未初始化，或未设置SWR\_FLAG\_RESAMPLE，则会使用标志集调用swr\_init（）。

sample\_delta：每个样本PTS的delta

compensation\_distance：要补偿的样品数量

返回：> = 0成功，AERROR错误代码如果：

- 1、s为null
- 2、compensation\_distance小于0，
- 3、compensation\_distance是0，但是sample\_delta不是，
- 4、补偿不支持重采样器，或
- 5、调用时，swr\_init（）失败。

## 14 以外部时钟为基准

前面我们分析了音视频同步中的两种策略：视频同步到音频，以及音频同步到视频。接下来要分析的是第三种，音频和视频都同步到外部时钟。

**在seek的时候体验非常差，没有必要选择这种同步方式。**

## 回顾

先回顾下前面两种同步策略。

视频同步到音频主要由函数compute\_target\_delay计算出lastvp应显示时长，并通过frame\_timer对比系统时间控制输出，最后在video\_refresh中更新了video clock(vidclk)。

```

1 static double compute_target_delay(double delay, VideoState *is)
2 {
3     //A. 只要主时钟不是video，就需要作同步校正
4     if (get_master_sync_type(is) != AV_SYNC_VIDEO_MASTER) {
5         diff = get_clock(&is->vidclk) - get_master_clock(is);
6     }

```

```

7     return delay;
8 }
9 static void video_refresh(void *opaque, double *remaining_time)
10 {
11     delay = compute_target_delay(last_duration, is);
12     if (time < is->frame_timer + delay) {
13         goto display;
14     }
15     //B. 更新vidclk, 同时更新extclk
16     update_video_pts(is, vp->pts, vp->pos, vp->serial);
17 }
18 static void update_video_pts(VideoState *is, double pts, int64_t pos
    , int serial) {
19     set_clock(&is->vidclk, pts, serial);
20     sync_clock_to_slave(&is->extclk, &is->vidclk);
21 }

```

注意这里的两点：

A. 只要主时钟不是video，就需要作同步校正

B. 更新vidclk，同时更新extclk

再看音频同步到视频。主要由函数 `synchronize_audio` 计算校正后应输出的样本数，然后通过 libswresample 库重采样输出。

```

1 static int synchronize_audio(VideoState *is, int nb_samples)
2 {
3     //C. 只要主时钟不是audio，就需要作同步校正
4     if (get_master_sync_type(is) != AV_SYNC_AUDIO_MASTER) {
5         diff = get_clock(&is->audclk) - get_master_clock(is);
6     }
7     return wanted_nb_samples;
8 }
9 static int audio_decode_frame(VideoState *is)
10 {
11     wanted_nb_samples = synchronize_audio(is, af->frame->nb_samples)
    ;
12     return resampled_data_size;
13 }
14 static void sdl_audio_callback(void *opaque, Uint8 *stream, int len)
15 {
16     audio_size = audio_decode_frame(is);

```



```

17 //D. 更新audclk, 同时更新extclk
18 set_clock_at(&is->audclk, is->audio_clock - (double)(2 * is->audio_hw_buf_size + is->audio_write_buf_size) / is->audio_tgt.bytes_per_sec, is->audio_clock_serial, audio_callback_time / 1000000.0);
19 sync_clock_to_slave(&is->extclk, &is->audclk);
20 }

```

会找到和“视频同步音频”类似的的两点：

C. 只要主时钟不是audio，就需要作同步校正

D. 更新audclk，同时更新extclk

## 分析

我们知道通过sync选项可以选择同步策略，分别可以选择audio/video/ext，选择不同选项的效果是：

- audio：视频同步到音频。上一节中的A被触发，video输出需要作同步，同步的参考(get\_master\_clock)是audclk.
- video：音频同步到视频。上一节中的C被触发，audio输出需要作同步，同步的参考是vidclk。
- ext：视频和音频都同步到外部时钟，上一节中的A和C都被触发，同步的参考是extclk

不论选择的是哪一个选项，B和D始终都有执行。

所以外部时钟为主同步策略是这样的：video输出和audio输出时都需要作校正，校正的方法是参考extclk计算diff值。其余部分参考“视频同步到音频”和“音频同步到视频”这两节的分析即可。

另一个问题是外部时钟(extclk)是如何对时的？在音视频同步基础概念中我们分析过Clock是需要一直对时以保持pts\_drift估算出来的pts不会偏差太远，并且get\_clock的返回值实际是这一Clock对应的流的pts。这两点对于extclk来说都是问题。

答案就在前面的B和D步骤中。

对于audclk和vidclk，都是每次在“显示”时用显示的那一帧的pts去对时set\_clock\_at/set\_clock。顺带地，会执行sync\_clock\_to\_slave(&is->extclk, &is->audclk); // &is->vidclk

```

1 static void sync_clock_to_slave(Clock *c, Clock *slave)
2 {
3     double clock = get_clock(c);
4     double slave_clock = get_clock(slave);
5     if (!isnan(slave_clock) && (isnan(clock) || fabs(clock - slave_clock) > AV_NOSYNC_THRESHOLD))
6         set_clock(c, slave_clock, slave->serial);
7 }

```

sync\_clock\_to\_slave的意思是用从时钟的pts和serial对主时钟对时。

而之所以可以这样做的原因是，在更新audclk和vidclk的时候，音频或视频已经同步到了外部时钟，此时取它们的值来反过来对外部时钟对时可以认为是准确的。

也许你会发现，不对，被兜了一圈！这是一个先有鸡还是先有蛋的问题。既然要把video和audio同步到extclk，我们用的extclk校正video和audio，得到更新后的audclk和vidclk，却又反过来用audclk和vidclk去对时extclk。分明就是蛋要鸡来生，鸡要蛋来敷嘛。

幸运的是，这个问题对于开天辟地，扮演上帝角色的代码而言并不难，ffplay说先有蛋。如果有仔细阅读过 `compute_target_delay` 和 `synchronize_audio`，就会发现进行校正的必要条件之一是 `!isnan(diff)`，也就是diff值是合法数值，这在第一帧的音频或视频显示前是不成立的，也就无需做同步校正。在第一帧视频或音频显示后，此时extclk得到对时，接下来就可以进入正常的同步“循环”了。

至此，同步到外部时钟的同步策略分析完了，简单总结下：

1. 该策略“复用”了前两种策略的代码，代码上几乎等效于前两种策略的叠加
2. extclk的对时依赖于已同步的audio或video的Clock