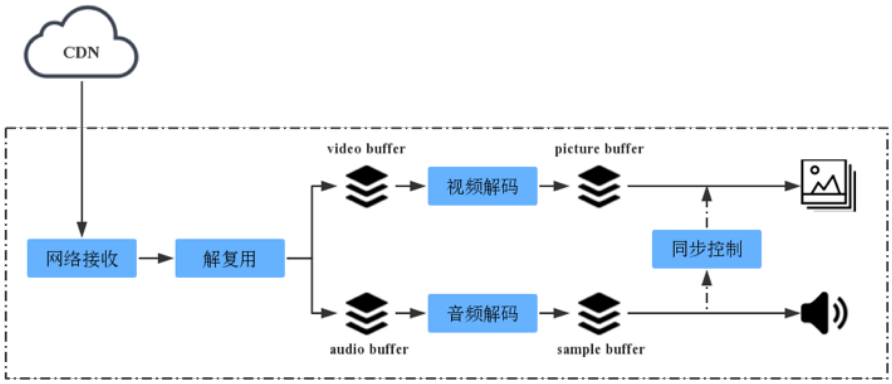
ijkplayer框架深入剖析

金山视频云 关注 IP属地: 浙江
8 2017.01.23 11:34:34 字数 3,938 阅读

随着互联网技术的飞速发展，移动端播放视频的需求如日中天，由此也催生了一批开源/闭源的播放器，但是无论这个播放器功能是否强大、[illegible]是否优秀，它的基本模块通常都是由以下部分组成：事务处理、数据的接收和解复用、音视频解码以及渲染，其基本框架如下图所示：



播放器基本框图.png

针对各种铺天盖地的播放器项目，我们选取了比较出众的ijkplayer进行源码剖析。它是一个基于FFPlay的轻量级Android/iOS视频播放器，实现了跨平台的功能，API易于集成；编译配置可裁剪，方便控制安装包大小。

本文基于k0.7.6版本的ijkplayer，重点分析其C语言实现的核心代码，涉及到不同平台下的封装接口或处理方式时，均以iOS平台为例，Android平台大同小异，请大家自行查阅研究。

一、总体说明

打开ijkplayer，可看到其主要目录结构如下：

> **tool** - 初始化项目工程脚本
>
> **config** - 编译ffmpeg使用的配置文件

> **ijkmedia** - 核心代码
>
> > **ijkplayer** - 播放器数据下载及解码相关
> >
> > **ijksdl** - 音视频数据渲染相关
>
> **ios** - iOS平台上的上层接口封装以及平台相关方法
>
> **android** - android平台上的上层接口封装以及平台相关方法

在功能的具体实现上，iOS和Android平台的差异主要表现在视频硬件解码以及音视频渲染方面，两者实现的载体区别如下表所示：

Platform	Hardware Codec	Video Render	Audio Output
iOS	VideoToolBox	OpenGL ES	AudioQueue
Android	MediaCodec	OpenGL ES、MediaCodec	OpenSL ES、AudioTrack

二、初始化流程

初始化完成的主要工作就是创建播放器对象，打开ijkplayer/ios/IJKMediaDemo/IJKMediaDemo.xcodeproj工程，可看到IJKMoviePlayerViewController类中viewDidLoad方法中创建了IJKFFMoviePlayerController对象，即iOS平台上的播放器对象。

```
- (void)viewDidLoad
{
    ......
    self.player = [[IJKFFMoviePlayerController alloc] initWithContentURL:self.url with
    ......
}
```

查看ijkplayer/ios/IJKMediaPlayer/IJKMediaPlayer/IJKFFMoviePlayerController.m文件，其初始化方法具体实现如下：

```
- (id)initWithContentURL:(NSURL *)aUrl
             withOptions:(IJKFFOptions *)options
{
    if (aUrl == nil)
        return nil;

    // Detect if URL is file path and return proper string for it
    NSString *aUrlString = [aUrl isFileURL] ? [aUrl path] : [aUrl absoluteString];

    return [self initWithContentURLString:aUrlString
                              withOptions:options];
}
```

```
- (id)initWithContentURLString:(NSString *)aUrlString
                   withOptions:(IJKFFOptions *)options
{
    if (aUrlString == nil)
        return nil;

    self = [super init];
    if (self) {
```

```
        ......
        // init player
        _mediaPlayer = ijkmp_ios_create(media_player_msg_loop);
        ......
    }
    return self;
}
```

可发现在此创建了IjkMediaPlayer结构体实例_mediaPlayer：

```
IjkMediaPlayer *ijkmp_ios_create(int (*msg_loop)(void*))
{
    IjkMediaPlayer *mp = ijkmp_create(msg_loop);
    if (!mp)
        goto fail;

    mp->ffplayer->vout = SDL_VoutIos_CreateForGLES2();
    if (!mp->ffplayer->vout)
        goto fail;

    mp->ffplayer->pipeline = ffpipeline_create_from_ios(mp->ffplayer);
    if (!mp->ffplayer->pipeline)
        goto fail;

    return mp;

fail:
    ijkmp_dec_ref_p(&mp);
    return NULL;
}
```

在该方法中主要完成了三个动作：

1. 创建IJKMediaPlayer对象

```
IjkMediaPlayer *ijkmp_create(int (*msg_loop)(void*))
{
    IjkMediaPlayer *mp = (IjkMediaPlayer *) mallocz(sizeof(IjkMediaPlayer));
    ......
    mp->ffplayer = ffp_create();
    ......
    mp->msg_loop = msg_loop;
    ......
    return mp;
}
```

通过`ffp_create`方法创建了FFPlayer对象，并设置消息处理函数。

2. 创建图像渲染对象SDL_Vout

```
SDL_Vout *SDL_VoutIos_CreateForGLES2()
{
    SDL_Vout *vout = SDL_Vout_CreateInternal(sizeof(SDL_Vout_Opaque));
    if (!vout)
        return NULL;

    SDL_Vout_Opaque *opaque = vout->opaque;
    opaque->gl_view = nil;
    vout->create_overlay = vout_create_overlay;
    vout->free_l = vout_free_l;
    vout->display_overlay = vout_display_overlay;

    return vout;
}
```

3. 创建平台相关的IJKFF_Pipeline对象，包括视频解码以及音频输出部分

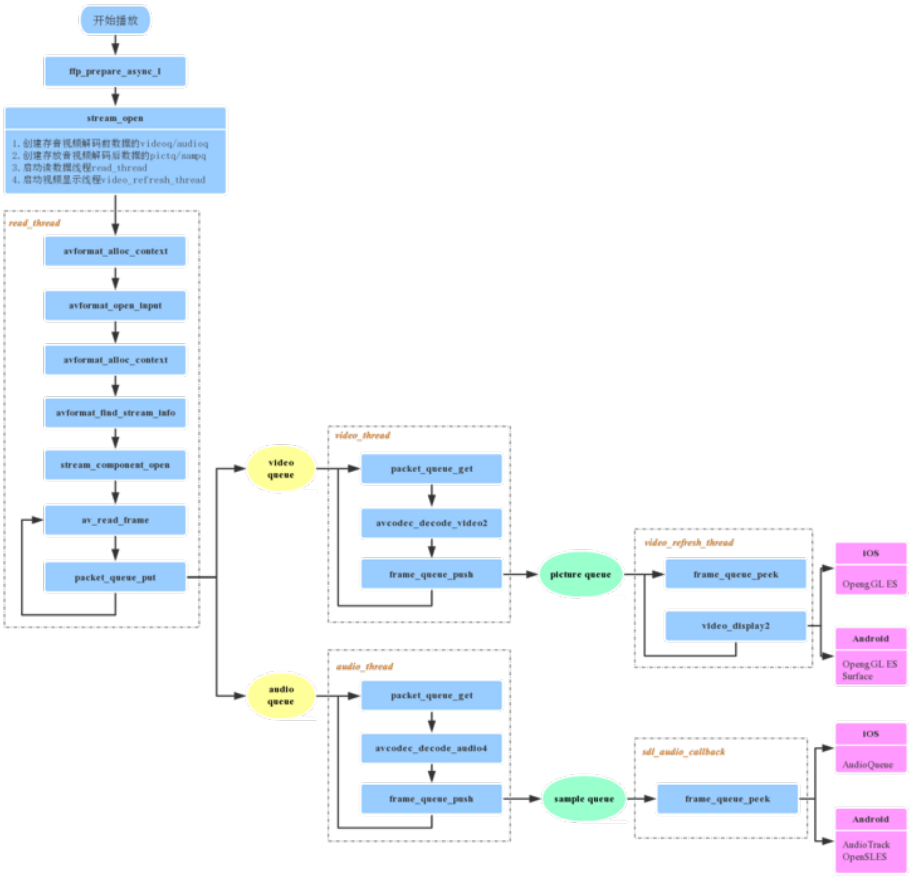
```
IJKFF_Pipeline *ffpipeline_create_from_ios(FFPlayer *ffp)
{
    IJKFF_Pipeline *pipeline = ffpipeline_alloc(&g_pipeline_class, sizeof(IJKFF_Pipe
    if (!pipeline)
        return pipeline;

    IJKFF_Pipeline_Opaque *opaque     = pipeline->opaque;
    opaque->ffp                       = ffp;
    pipeline->func_destroy            = func_destroy;
    pipeline->func_open_video_decoder = func_open_video_decoder;
    pipeline->func_open_audio_output  = func_open_audio_output;

    return pipeline;
}
```

至此已经完成了ijkplayer播放器初始化的相关流程，简单来说，就是创建播放器对象，完成音视频解码、渲染的准备工作。在下一章节中，会重点介绍播放的核心代码。

三、核心代码剖析

ijkplayer实际上是基于ffplay.c实现的，本章节将以该文件为主线，从数据接收、音视频解码、音视频渲染及同步这三大方面进行讲解，要求读者有基本的ffmpeg知识。

ffplay.c中主要的代码调用流程如下图所示：



ffplay代码调用流程图.png

当外部调用`prepareToPlay`启动播放后，ijkplayer内部最终会调用到ffplay.c中的

```
int ffp_prepare_async_l(FFPlayer *ffp, const char *file_name)
```

方法，该方法是启动播放器的入口函数，在此会设置player选项，打开audio output，最重要的是调用`stream_open`方法。

```
static VideoState *stream_open(FFPlayer *ffp, const char *filename, AVInputFormat *ifo
{
    ......
    /* start video display */
    if (frame_queue_init(&is->pictq, &is->videoq, ffp->pictq_size, 1) < 0)
        goto fail;
    if (frame_queue_init(&is->sampq, &is->audioq, SAMPLE_QUEUE_SIZE, 1) < 0)
        goto fail;

    if (packet_queue_init(&is->videoq) < 0 ||
        packet_queue_init(&is->audioq) < 0 )
        goto fail;

    ......

    is->video_refresh_tid = SDL_CreateThreadEx(&is->_video_refresh_tid, video_refresh_

    ......

    is->read_tid = SDL_CreateThreadEx(&is->_read_tid, read_thread, ffp, "ff_read");

    ......
}
```

从代码中可以看出，stream_open主要做了以下几件事情:

- 创建存放video/audio解码前数据的videoq/audioq
- 创建存放video/audio解码后数据的pictq/sampq
- 创建读数据线程`read_thread`
- 创建视频渲染线程`video_refresh_thread`

说明：subtitle是与video、audio平行的一个stream，ffplay中也支持对它的处理，即创建存放解码前后数据的两个queue，并且当文件中存在subtitle时，还会启动subtitle的解码线程，由于篇幅有限，本文暂时忽略对它的相关介绍。

3.1 数据读取

数据读取的整个过程都是由ffmpeg内部完成的，接收到网络过来的数据后，ffmpeg根据其封装格式，完成了解复用的动作，我们得到的，是音视频分离开的解码前的数据，步骤如下：

1. 创建上下文结构体，这个结构体是最上层的结构体，表示输入上下文

```
ic = avformat_alloc_context();
```

2. 设置中断函数，如果出错或者退出，就可以立刻退出

```
ic->interrupt_callback.callback = decode_interrupt_cb;
ic->interrupt_callback.opaque = is;
```

3. 打开文件，主要是探测协议类型，如果是网络文件则创建网络链接等

```
err = avformat_open_input(&ic, is->filename, is->iformat, &ffp->format_opts);
```

4. 探测媒体类型，可得到当前文件的封装格式，音视频编码参数等信息

```
err = avformat_find_stream_info(ic, opts);
```

5. 打开视频、音频解码器。在此会打开相应解码器，并创建相应的解码线程。

```
stream_component_open(ffp, st_index[AVMEDIA_TYPE_AUDIO]);
```

6. 读取媒体数据，得到的是音视频分离的解码前数据

```
ret = av_read_frame(ic, pkt);
```

7. 将音视频数据分别送入相应的queue中

```
if (pkt->stream_index == is->audio_stream && pkt_in_play_range) {
    packet_queue_put(&is->audioq, pkt);
} else if (pkt->stream_index == is->video_stream && pkt_in_play_range && !(is->vide
    packet_queue_put(&is->videoq, pkt);
    ......
} else {
    av_packet_unref(pkt);
}
```

重复6、7步，即可不断获取待播放的数据。

3.2 音视频解码

ijkplayer在视频解码上支持软解和硬解两种方式，可在起播前配置优先使用的解码方式，播放过程中不可切换。iOS平台上硬解使用VideoToolbox，Android平台上使用MediaCodec。ijkplayer中的音频解码只支持软解，暂不支持硬解。

3.2.1 视频解码方式选择

在打开解码器的方法中：

```
static int stream_component_open(FFPlayer *ffp, int stream_index)
{
    ......
    codec = avcodec_find_decoder(avctx->codec_id);
    ......
    if ((ret = avcodec_open2(avctx, codec, &opts)) < 0) {
        goto fail;
    }
    ......
    case AVMEDIA_TYPE_VIDEO:
        ......
        decoder_init(&is->viddec, avctx, &is->videoq, is->continue_read_thread);
        ffp->node_vdec = ffpipeline_open_video_decoder(ffp->pipeline, ffp);
        if (!ffp->node_vdec)
            goto fail;
        if ((ret = decoder_start(&is->viddec, video_thread, ffp, "ff_video_dec")) < 0)
            goto out;
    ......
}
```

首先会打开ffmpeg的解码器，然后通过`ffpipeline_open_video_decoder`创建IJKFF_Pipenode。

第二章节中有介绍，在创建IJKMediaPlayer对象时，通过`ffpipeline_create_from_ios`创建了pipeline，则

```
IJKFF_Pipenode* ffpipeline_open_video_decoder(IJKFF_Pipeline *pipeline, FFPlayer *ffp)
{
    return pipeline->func_open_video_decoder(pipeline, ffp);
}
```

`func_open_video_decoder`函数指针最后指向的是ffpipeline_ios.c中的`func_open_video_decoder`，其定义如下：

```
static IJKFF_Pipenode *func_open_video_decoder(IJKFF_Pipeline *pipeline, FFPlayer *ffp
{
    IJKFF_Pipenode* node = NULL;
    IJKFF_Pipeline_Opaque *opaque = pipeline->opaque;
    if (ffp->videotoolbox) {
       node = ffpipenode_create_video_decoder_from_ios_videotoolbox(ffp);
       if (!node)
          ALOGE("vtb fail!!! switch to ffmpeg decode!!!! \n");
     }
     if (node == NULL) {
        node = ffpipenode_create_video_decoder_from_ffplay(ffp);
        ffp->stat.vdec_type = FFP_PROPV_DECODER_AVCODEC;
        opaque->is_videotoolbox_open = false;
     } else {
        ffp->stat.vdec_type = FFP_PROPV_DECODER_VIDEOTOOLBOX;
        opaque->is_videotoolbox_open = true;
     }
     ffp_notify_msg2(ffp, FFP_MSG_VIDEO_DECODER_OPEN, opaque->is_videotoolbox_open);
     return node;
}
```

如果配置了ffp->videotoolbox，会优先去尝试打开硬件解码器，

```
node = ffpipenode_create_video_decoder_from_ios_videotoolbox(ffp);
```

如果硬件解码器打开失败，则会自动切换至软解

```
node = ffpipenode_create_video_decoder_from_ffplay(ffp);
```

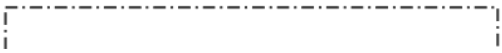
ffp->videotoolbox需要在起播前通过如下方法配置：

```
ijkmp_set_option_int(_mediaPlayer, IJKMP_OPT_CATEGORY_PLAYER,   "videotoolbox", 1);
```

3.2.2 音视频解码

video的解码线程为`video_thread`，audio的解码线程为`audio_thread`。

不管视频解码还是音频解码，其基本流程都是从解码前的数据缓冲区中取出一帧数据进行解码，完成后放入相应的解码后的数据缓冲区，如下图所示：



音视频解码示意图.png

本文以video的软解流程为例进行分析，audio的流程可对照研究。

视频解码线程

```
static int video_thread(void *arg)
{
    FFPlayer *ffp = (FFPlayer *)arg;
    int       ret = 0;

    if (ffp->node_vdec) {
        ret = ffpipenode_run_sync(ffp->node_vdec);
    }
    return ret;
}
```

`ffpipenode_run_sync`中调用的是IJKFF_Pipenode对象中的`func_run_sync`

```
int ffpipenode_run_sync(IJKFF_Pipenode *node)
{
    return node->func_run_sync(node);
}
```

`func_run_sync`取决于播放前配置的软硬解，假设为软解，则调用

```
static int ffplay_video_thread(void *arg)
{
    FFPlayer *ffp = arg;

    ......

    for (;;) {
        ret = get_video_frame(ffp, frame);
        ......
        ret = queue_picture(ffp, frame, pts, duration, av_frame_get_pkt_pos(frame), is
    }
    return 0;
}
```

`get_video_frame中`调用了`decoder_decode_frame`，其定义如下：

```
static int decoder_decode_frame(FFPlayer *ffp, Decoder *d, AVFrame *frame, AVSubtitle
    int got_frame = 0;

    do {
        int ret = -1;
        ......
        if (!d->packet_pending || d->queue->serial != d->pkt_serial){
            AVPacket pkt;
            do {
                ......
                if (packet_queue_get_or_buffering(ffp, d->queue, &pkt, &d->pkt_serial,
                    return -1;
                ......
            } while (pkt.data == flush_pkt.data || d->queue->serial != d->pkt_serial);
            ......
        }

        switch (d->avctx->codec_type) {
            case AVMEDIA_TYPE_VIDEO: {
```

```
                ret = avcodec_decode_video2(d->avctx, frame, &got_frame, &d->pkt_temp)
                ......
               }
                break;
        }
        ......
    } while (!got_frame && !d->finished);

    return got_frame;
}
```

该方法中从解码前的video queue中取出一帧数据，送入decoder进行解码，解码后的数据在`ffplay_video_thread`中送入pictq。

3.3 音视频渲染及同步

3.3.1 音频输出

ijkplayer中Android平台使用OpenSL ES或AudioTrack输出音频，iOS平台使用AudioQueue输出音频。

audio output节点，在`ffp_prepare_async_l`方法中被创建：

```
ffp->aout = ffpipeline_open_audio_output(ffp->pipeline, ffp);
```

`ffpipeline_open_audio_output`方法实际上调用的是IJKFF_Pipeline对象的函数指针`func_open_audio_utput`，该函数指针在初始化中的`ijkmp_ios_create`方法中被赋值，最后指向的是`func_open_audio_output`

```
static SDL_Aout *func_open_audio_output(IJKFF_Pipeline *pipeline, FFPlayer *ffp)
{
    return SDL_AoutIos_CreateForAudioUnit();
}
```

`SDL_AoutIos_CreateForAudioUnit`定义如下，主要完成的是创建SDL_Aout对象

```
SDL_Aout *SDL_AoutIos_CreateForAudioUnit()
{
    SDL_Aout *aout = SDL_Aout_CreateInternal(sizeof(SDL_Aout_Opaque));
    if (!aout)
        return NULL;

    // SDL_Aout_Opaque *opaque = aout->opaque;

    aout->free_l = aout_free_l;
    aout->open_audio  = aout_open_audio;
    aout->pause_audio = aout_pause_audio;
    aout->flush_audio = aout_flush_audio;
    aout->close_audio = aout_close_audio;

    aout->func_set_playback_rate = aout_set_playback_rate;
    aout->func_set_playback_volume = aout_set_playback_volume;
    aout->func_get_latency_seconds = auout_get_latency_seconds;
    aout->func_get_audio_persecond_callbacks = aout_get_persecond_callbacks;
    return aout;
}
```

回到ffplay.c中，如果发现待播放的文件中含有音频，那么在调用`stream_component_open`打开解码器时，该方法里面也调用`audio_open`打开了audio output设备。

```
static int audio_open(FFPlayer *opaque, int64_t wanted_channel_layout, int wanted_nb_c
```

```
{
    FFPlayer *ffp = opaque;
    VideoState *is = ffp->is;
    SDL_AudioSpec wanted_spec, spec;
    ......
    wanted_nb_channels = av_get_channel_layout_nb_channels(wanted_channel_layout);
    wanted_spec.channels = wanted_nb_channels;
    wanted_spec.freq = wanted_sample_rate;
    wanted_spec.format = AUDIO_S16SYS;
    wanted_spec.silence = 0;
    wanted_spec.samples = FFMAX(SDL_AUDIO_MIN_BUFFER_SIZE, 2 << av_log2(wanted_spec.fr
    wanted_spec.callback = sdl_audio_callback;
    wanted_spec.userdata = opaque;
    while (SDL_AoutOpenAudio(ffp->aout, &wanted_spec, &spec) < 0) {
        .....
    }
    ......
    return spec.size;
}
```

在`audio_open`中配置了音频输出的相关参数`SDL_AudioSpec`，并通过

```
int SDL_AoutOpenAudio(SDL_Aout *aout, const SDL_AudioSpec *desired, SDL_AudioSpec *obt
{
    if (aout && desired && aout->open_audio)
        return aout->open_audio(aout, desired, obtained);

    return -1;
}
```

设置给了Audio Output, iOS平台上即为AudioQueue。

AudioQueue模块在工作过程中，通过不断的callback来获取pcm数据进行播放。

有关AudioQueue的具体内容此处不再介绍。

3.3.2 视频渲染

iOS平台上采用OpenGL渲染解码后的YUV图像，渲染线程为`video_refresh_thread`，最后渲染图像的方法为`video_image_display2`，定义如下：

```
static void video_image_display2(FFPlayer *ffp)
{
    VideoState *is = ffp->is;
    Frame *vp;
    Frame *sp = NULL;

    vp = frame_queue_peek_last(&is->pictq);
    ......

    SDL_VoutDisplayYUVOverlay(ffp->vout, vp->bmp);
    ......
}
```

从代码实现上可以看出，该线程的主要工作为：

1. 调用`frame_queue_peek_last`从pictq中读取当前需要显示视频帧

2. 调用`SDL_VoutDisplayYUVOverlay`进行绘制

```
int SDL_VoutDisplayYUVOverlay(SDL_Vout *vout, SDL_VoutOverlay   *overlay)
{
    if (vout && overlay && vout->display_overlay)
        return vout->display_overlay(vout, overlay);
```

```
4
5       return -1;
6   }
7
```

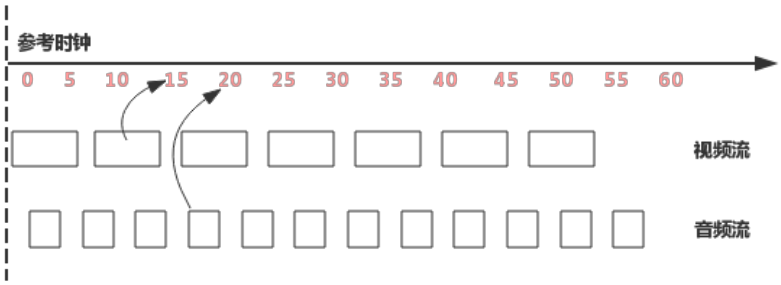
display_overlay函数指针在前面初始化流程有介绍过，它在

```
1   SDL_Vout *SDL_VoutIos_CreateForGLES2()
```

方法中被赋值为`vout_display_overlay`，该方法就是调用OpengGL绘制图像。

3.4.3 音视频同步

对于播放器来说，音视频同步是一个关键点，同时也是一个难点，同步效果的好坏，直接决定着播放器的质量。通常音视频同步的解决方案就是选择一个参考时钟，播放时读取音视频帧上的时间戳，同时参考当前时钟参考时钟上的时间来安排播放。如下图所示：



音视频同步示意图.png

如果音视频帧的播放时间大于当前参考时钟上的时间，则不急于播放该帧，直到参考时钟达到该帧的时间戳；如果音视频帧的时间戳小于当前参考时钟上的时间，则需要“尽快”播放该帧或丢弃，以便播放进度追上参考时钟。

参考时钟的选择也有多种方式：

- 选取视频时间戳作为参考时钟源
- 选取音频时间戳作为参考时钟源
- 选取外部时间作为参考时钟源

考虑人对视频、和音频的敏感度，在存在音频的情况下，优先选择音频作为主时钟源。

ijkplayer在默认情况下也是使用音频作为参考时钟源，处理同步的过程主要在视频渲染`video_refresh_thread`的线程中：

```
1   static int video_refresh_thread(void *arg)
2   {
3       FFPlayer *ffp = arg;
4       VideoState *is = ffp->is;
5       double remaining_time = 0.0;
6       while (!is->abort_request) {
7           if (remaining_time > 0.0)
8               av_usleep((int)(int64_t)(remaining_time * 1000000.0));
9           remaining_time = REFRESH_RATE;
10          if (is->show_mode != SHOW_MODE_NONE && (!is->paused || is->force_refresh))
11              video_refresh(ffp, &remaining_time);
12      }
```

```
13
14      return 0;
15  }
```

从上述实现可以看出，该方法中主要循环做两件事情：

1. 休眠等待，`remaining_time`的计算在`video_refresh`中
2. 调用`video_refresh`方法，刷新视频帧

可见同步的重点是在`video_refresh`中，下面着重分析该方法：

```
1     lastvp = frame_queue_peek_last(&is->pictq);
2     vp = frame_queue_peek(&is->pictq);
3      ......
4     /* compute nominal last_duration */
5     last_duration = vp_duration(is, lastvp, vp);
6     delay = compute_target_delay(ffp, last_duration, is);
```

lastvp是上一帧，vp是当前帧，last_duration则是根据当前帧和上一帧的pts，计算出来上一帧的显示时间，经过`compute_target_delay`方法，计算出显示当前帧需要等待的时间。

```
1   static double compute_target_delay(FFPlayer *ffp, double delay, VideoState *is)
2   {
3       double sync_threshold, diff = 0;
4
5       /* update delay to follow master synchronisation source */
6       if (get_master_sync_type(is) != AV_SYNC_VIDEO_MASTER) {
7           /* if video is slave, we try to correct big delays by
8              duplicating or deleting a frame */
9           diff = get_clock(&is->vidclk) - get_master_clock(is);
10
11          /* skip or repeat frame. We take into account the
12             delay to compute the threshold. I still don't know
13             if it is the best guess */
14          sync_threshold = FFMAX(AV_SYNC_THRESHOLD_MIN, FFMIN(AV_SYNC_THRESHOLD_MAX, del
15          /* -- by bbcallen: replace is->max_frame_duration with AV_NOSYNC_THRESHOLD */
16          if (!isnan(diff) && fabs(diff) < AV_NOSYNC_THRESHOLD) {
17              if (diff <= -sync_threshold)
18                  delay = FFMAX(0, delay + diff);
19              else if (diff >= sync_threshold && delay > AV_SYNC_FRAMEDUP_THRESHOLD)
20                  delay = delay + diff;
21              else if (diff >= sync_threshold)
22                  delay = 2 * delay;
23          }
24      }
25
26      .....
27
28      return delay;
29  }
```

在`compute_target_delay`方法中，如果发现当前主时钟源不是video，则计算当前视频时钟与主时钟的差值：

- 如果当前视频帧落后于主时钟源，则需要减小下一帧画面的等待时间；
- 如果视频帧超前，并且该帧的显示时间大于显示更新门槛，则显示下一帧的时间为超前的时间差加上上一帧的显示时间
- 如果视频帧超前，并且上一帧的显示时间小于显示更新门槛，则采取加倍延时的策略。

回到`video_refresh`中

```
1     time= av_gettime_relative()/1000000.0;
```

```
    if (isnan(is->frame_timer) || time < is->frame_timer)
        is->frame_timer = time;
    if (time < is->frame_timer + delay) {
        *remaining_time = FFMIN(is->frame_timer + delay - time, *remaining_time);
        goto display;
    }
```

`frame_timer`实际上就是上一帧的播放时间，而`frame_timer + delay`实际上就是当前这一帧的播放时间，如果系统时间还没有到当前这一帧的播放时间，直接跳转至display，而此时`is->force_refresh`变量为0，不显示当前帧，进入`video_refresh_thread`中下一次循环，并睡眠等待。

```
    is->frame_timer += delay;
    if (delay > 0 && time - is->frame_timer > AV_SYNC_THRESHOLD_MAX)
        is->frame_timer = time;

    SDL_LockMutex(is->pictq.mutex);
    if (!isnan(vp->pts))
            update_video_pts(is, vp->pts, vp->pos, vp->serial);
    SDL_UnlockMutex(is->pictq.mutex);

    if (frame_queue_nb_remaining(&is->pictq) > 1) {
         Frame *nextvp = frame_queue_peek_next(&is->pictq);
         duration = vp_duration(is, vp, nextvp);
         if(!is->step && (ffp->framedrop > 0 || (ffp->framedrop && get_master_sync_type(
             frame_queue_next(&is->pictq);
             goto retry;
         }
    }
```

如果当前这一帧的播放时间已经过了，并且其和当前系统时间的差值超过了`AV_SYNC_THRESHOLD_MAX`，则将当前这一帧的播放时间改为系统时间，并在后续判断是否需要丢帧，其目的是为后面帧的播放时间重新调整frame_timer，如果缓冲区中有更多的数据，并且当前的时间已经大于当前帧的持续显示时间，则丢弃当前帧，尝试显示下一帧。

```
{
    frame_queue_next(&is->pictq);
    is->force_refresh = 1;

    SDL_LockMutex(ffp->is->play_mutex);

     ......

display:
    /* display picture */
    if (!ffp->display_disable && is->force_refresh && is->show_mode == SHOW_MODE_VIDEO
        video_display2(ffp);
```

否则进入正常显示当前帧的流程，调用`video_display2`开始渲染。

四、事件处理

在播放过程中，某些行为的完成或者变化，如prepare完成，开始渲染等，需要以事件形式通知到外部，以便上层作出具体的业务处理。

ijkplayer支持的事件比较多，具体定义在ijkplayer/ijkmedia/ijkplayer/ff_ffmsg.h中：

```
#define FFP_MSG_FLUSH                       0
#define FFP_MSG_ERROR                       100     /* arg1 = error */
#define FFP_MSG_PREPARED                    200
#define FFP_MSG_COMPLETED                   300
#define FFP_MSG_VIDEO_SIZE_CHANGED          400     /* arg1 = width, arg2 = height */
```

```
#define FFP_MSG_SAR_CHANGED                 401     /* arg1 = sar.num, arg2 = sar.den
#define FFP_MSG_VIDEO_RENDERING_START       402
#define FFP_MSG_AUDIO_RENDERING_START       403
#define FFP_MSG_VIDEO_ROTATION_CHANGED      404     /* arg1 = degree */
#define FFP_MSG_BUFFERING_START             500
#define FFP_MSG_BUFFERING_END               501
#define FFP_MSG_BUFFERING_UPDATE            502     /* arg1 = buffering head position
#define FFP_MSG_BUFFERING_BYTES_UPDATE      503     /* arg1 = cached data in bytes,
#define FFP_MSG_BUFFERING_TIME_UPDATE       504     /* arg1 = cached duration in milli
#define FFP_MSG_SEEK_COMPLETE               600     /* arg1 = seek position,
#define FFP_MSG_PLAYBACK_STATE_CHANGED      700
#define FFP_MSG_TIMED_TEXT                  800
#define FFP_MSG_VIDEO_DECODER_OPEN          10001
```

4.1 消息上报初始化

在IJKFFMoviePlayerController的初始化方法中:

```
- (id)initWithContentURLString:(NSString *)aUrlString
                   withOptions:(IJKFFOptions *)options
{
    ......
    // init player
    _mediaPlayer = ijkmp_ios_create(media_player_msg_loop);
    ......
}
```

可以看到在创建播放器时，`media_player_msg_loop`函数地址作为参数传入了`ijkmp_ios_create`，继续跟踪代码，可以发现，该函数地址最终被赋值给了IjkMediaPlayer中的`msg_loop`函数指针

```
IjkMediaPlayer *ijkmp_create(int (*msg_loop)(void*))
{
    ......
    mp->msg_loop = msg_loop;
    ......
}
```

开始播放时，会启动一个消息线程，

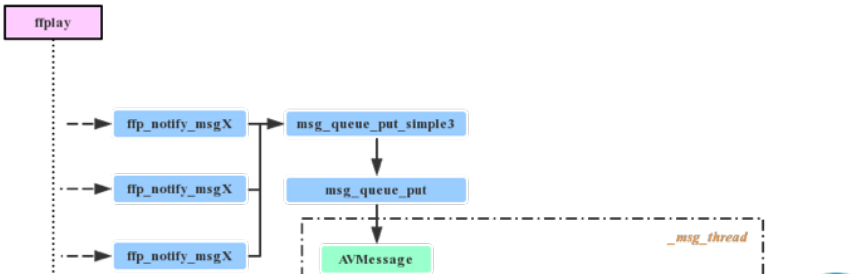
```
static int ijkmp_prepare_async_l(IjkMediaPlayer *mp)
{
    ......
    mp->msg_thread = SDL_CreateThreadEx(&mp->_msg_thread, ijkmp_msg_loop, mp, "ff_msg_
    ......
}
```

`ijkmp_msg_loop`方法中调用的即是`mp->msg_loop`。

至此已经完成了播放消息发送的准备工作。

4.2 消息上报处理

播放器底层上报事件时，实际上就是将待发送的消息放入消息队列，另外有一个线程会不断从队列中取出消息，上报给外部，其代码流程大致如下图所示：



消息上报代码调用流程图.png

我们以prepare完成事件为例，看看代码中事件上报的具体流程。

ffplay.c中上报PREPARED完成时调用：

```
ffp_notify_msg1(ffp, FFP_MSG_PREPARED);
```

`ffp_notify_msg1`方法实现如下：

```
inline static void ffp_notify_msg1(FFPlayer *ffp, int what) {
    msg_queue_put_simple3(&ffp->msg_queue, what, 0, 0);
}
```

`msg_queue_put_simple3`中将事件及其参数封装成了AVMessge对象，

```
inline static void msg_queue_put_simple3(MessageQueue *q, int what, int arg1, int arg2
{
    AVMessage msg;
    msg_init_msg(&msg);
    msg.what = what;
    msg.arg1 = arg1;
    msg.arg2 = arg2;
    msg_queue_put(q, &msg);
}
```

继续跟踪代码，可以发现最后在

```
inline static int msg_queue_put_private(MessageQueue *q, AVMessage *msg)
```

方法中，消息对象被放在了消息队列里。但是哪里读取的队列里的消息呢？在4.1节中，我们有提到在创建播放器时，会传入`media_player_msg_loop`函数地址，最后作为一个单独的线程运行，现在来看一下`media_player_msg_loop`方法的实现：

```
int media_player_msg_loop(void* arg)
{
    @autoreleasepool {
        IjkMediaPlayer *mp = (IjkMediaPlayer*)arg;
        __weak IJKFFMoviePlayerController *ffpController = ffplayerRetain(ijkmp_set_we
        while (ffpController) {
            @autoreleasepool {
                IJKFFMoviePlayerMessage *msg = [ffpController obtainMessage];
                if (!msg)
                    break;

                int retval = ijkmp_get_msg(mp, &msg->_msg, 1);
                if (retval < 0)
                    break;

```

```
                // block-get should never return 0
                assert(retval > 0);
                [ffpController performSelectorOnMainThread:@selector(postEvent:) withO
            }
        }

        // retained in prepare_async, before SDL_CreateThreadEx
        ijkmp_dec_ref_p(&mp);
        return 0;
    }
}
```

由此可以看出，最后是在该方法中读取消息，并采用notification通知到APP上层。

五、结束语

本文只是粗略的分析了*ijkplayer*的关键代码部分，平台相关的解码、渲染以及用户事务处理部分，都没有具体分析到，大家可以参考代码自行分析，也欢迎加QQ群讨论。

转载请注明：
作者金山视频云，首发简书 Jianshu.com

后续我们还会陆续推出其他有关音视频方面的文章，请大家关注。

也欢迎大家使用我们的多媒体SDK：

https://github.com/ksvc

金山云多媒体SDK相关的QQ交流群：

- 视频云技术交流群：574179720
- 视频云iOS技术交流：621137661

深圳植发怎么样




写下你的评论...

全部评论 28 只看作者

按时间倒序 按时间正序

木木一直在哭泣 IP属地: 上海
12楼 2020.06.22 00:26
看了你的分享，感觉清楚了不少。
赞 回复

pengxiaochao IP属地: 青海
11楼 2020.04.12 16:44
楼主有意思，说了那么多ijkplayer
赞 回复

深圳植发怎么样




Anxdada IP属地: 湖南
9楼 2020.03.11 19:59
MediaCodec 也有渲染功能么？
赞 回复

噜啦啦ya IP属地: 广东
8楼 2019.05.22 18:43

你好，请问支持转推吗?
我先从其他的地方拉流，在同一个界面，实现直播的功能，直播的数据源不是手机摄像头和麦克风

1 回复

多加辣椒 IP属地: 广东
2019.08.01 14:32

可以的，我也在做这个功能，可以交流一下

回复

噜啦啦ya IP属地: 广东
2019.08.01 19:32

@歪樣 我已经解决这个问题了

回复

多加辣椒 IP属地: 广东
2019.08.02 15:01

@memoryTao 大佬能否分享一下你是如何解决的呢?

回复

添加新评论 | 还有7条评论，查看更多

feng55 IP属地: 安徽
7楼 2018.10.18 14:40

请教下，ios上如何设置跳转到指定位置播放视频呢。我设置了 [options setPlayerOptionIntValue:120 forKey:@"seek_at_start"]但是确没有生效。视频还是从头开始播了

赞 回复

进化中的程序猿 IP属地: 河南
2018.12.17 10:10

兄弟，你这个问题解决了么，怎么从某个时间开始播放?

回复

feng55 IP属地: 安徽
2018.12.17 10:28

@进化中的程序猿 没找到具体的方法，目前用了一个笨方法，在视频加载出来的回掉里面再seek。基本上可以达到你要的效果。

回复

进化中的程序猿 IP属地: 河南
2018.12.17 17:07

@feng55 好的，我看看效果，3q

回复

添加新评论

不会游泳的飞鱼 IP属地: 上海
6楼 2018.06.15 13:50

贵公司KSYMediaPlayer_iOS能实现边下边播吗

赞 回复

超_iOS IP属地: 河南
5楼 2018.05.17 11:28

ijkplayer 和贵公司的KSYMediaPlayer_iOS哪个好啊？我们要二选一😢

赞 回复

winvsmary IP属地: 澳门
2018.05.25 09:57

ijkplayer更原始，KSYMediaPlayer_iOS是定制化后的，根据你的需求选就好。

回复

zyg IP属地: 广东
2018.10.15 13:21

@winvsmary 牛逼 赞

回复

添加新评论

shengshenger IP属地: 重庆
4楼 2018.04.28 10:51

请问iOS版本的怎么实现1.5倍速播放?

1 回复

d3ec87f2182c IP属地: 北京
3楼 2018.01.04 09:58

请问下，播放m3u8，想拿到成功播放出来的每一个ts信息，应该在哪拿呢?

赞 回复

豆宝的老公 IP属地: 西藏
2楼 2017.04.05 17:00

你好，我要怎么把原始的音频数据拿出来？在哪里拿？怎么拿？求指教，谢谢

1 回复

豆宝的老公 IP属地: 西藏
2017.04.13 12:15

@金山视频云 ijkplayer 是可以拿出音视频数据的，并且金山云的播放器最大的缺点是没有集成各种播放模式，例如，全景播放，小行星模式，鱼眼模式，vr模式等

回复

67c5d1142dcc IP属地: 四川
2017.05.27 14:31

老哥 文章写得很不错，我可以转载下吗?

回复

overla5 IP属地: 上海
2017.12.18 11:14

@jeffasd_proc 你好，怎么拿到视频文件的前几个字节进行解密?

回复

添加新评论

被以下专题收入，发现更多相似内容

+收入我的专题 音视频 iOS Study FFMPEG 播放器 音视频播放 Android直播 Android... 展开更多

推荐阅读

更多精彩内容 >

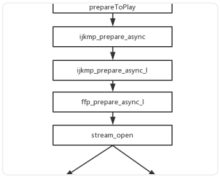
iOS ffmpeg 理解

教程一:视频截图(Tutorial 01: Making Screencaps) 首先我们需要了解视频文件的一些基...

90后的思维 阅读 3,999 评论 0 赞 3

ijkPlayer主流程分析

这是一个跨平台的播放器ijkplayer，iOS上集成看【如何快速的开发一个完整的iOS直播app】(原理篇)。 ...



FindCrt 阅读 6,543 评论 2 赞 45

专注与踏实的企业转型之路

2015年可谓是中国企业的跌宕起伏之年，互联网概念此起彼伏，门口的野蛮人前赴后继，所有人都在互联网思维的泥潭中挣扎...

慕容随风 阅读 370 评论 0 赞 2