

## 2-播放器-框架设计 之 ijkplayer框架深入剖析

---

### 一、总体说明

### 二、初始化流程

### 三、核心代码剖析

#### 3.1 数据读取

#### 3.2 音视频解码

#### 3.3 音视频渲染及同步

### 四、事件处理

#### 4.1 消息上报初始化

#### 4.2 消息上报处理

### 五、结束语

ijkplayer源码地址：<https://gitee.com/mirrors/ijkplayer>

原始文档：<https://www.jianshu.com/p/daf0a61cc1e0>

ijkplayer.c

ijkmp\_get\_msg

message\_loop\_n

内部处理和外部处理都在这里。

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、兼容性是否优秀，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：

image

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
- extra – 存放编译ijkplayer所需的依赖源文件，如ffmpeg、openssl等
- 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。

```
1 - (void)viewDidLoad
2 {
3     .....
4     self.player = [[IJKFFMoviePlayerController alloc] initWithContentURL:self.url withOptions:options];
5     .....
6 }
```

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

```
1 - (id)initWithContentURL:(NSURL *)aUrl
2         withOptions:(IJKFFOptions *)options
3 {
4     if (aUrl == nil)
5         return nil;
6     // Detect if URL is file path and return proper string for it
7     NSString *aUrlString = [aUrl isFileURL] ? [aUrl path] : [aUrl
    absoluteString];
8     return [self initWithContentURLString:aUrlString
9             withOptions:options];
10 }
```

```
1 - (id)initWithContentURLString:(NSString *)aUrlString
2         withOptions:(IJKFFOptions *)options
3 {
4     if (aUrlString == nil)
5         return nil;
6     self = [super init];
7     if (self) {
8         .....
9         // init player
10        _mediaPlayer = ijkmp_ios_create(media_player_msg_loop);
11        .....
12    }
13    return self;
14 }
```

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

```
1 IjkMediaPlayer *ijkmp_ios_create(int (*msg_loop)(void*))
2 {
3     IjkMediaPlayer *mp = ijkmp_create(msg_loop);
4     if (!mp)
5         goto fail;
```

```

6     mp->ffplayer->vout = SDL_VoutIos_CreateForGLES2();
7     if (!mp->ffplayer->vout)
8         goto fail;
9     mp->ffplayer->pipeline = ffpipeline_create_from_ios(mp->ffplayer);
10    if (!mp->ffplayer->pipeline)
11        goto fail;
12    return mp;
13 fail:
14    ijkmp_dec_ref_p(&mp);
15    return NULL;
16 }

```

在该方法中主要完成了三个动作：

#### 1. 创建IJKMediaPlayer对象

```

1 IjkMediaPlayer *ijkmp_create(int (*msg_loop)(void*))
2 {
3     IjkMediaPlayer *mp = (IjkMediaPlayer *) malloc(sizeof(IjkMediaPlayer));
4     .....
5     mp->ffplayer = ffp_create();
6     .....
7     mp->msg_loop = msg_loop;
8     .....
9     return mp;
10 }

```

1. 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

```

1 SDL_Vout *SDL_VoutIos_CreateForGLES2()
2 {
3     SDL_Vout *vout = SDL_Vout_CreateInternal(sizeof(SDL_Vout_Opaque));
4     if (!vout)
5         return NULL;
6     SDL_Vout_Opaque *opaque = vout->opaque;
7     opaque->gl_view = nil;
8     vout->create_overlay = vout_create_overlay;
9     vout->free_l = vout_free_l;

```

```

10     vout->display_overlay = vout_display_overlay;
11     return vout;
12 }

```

1. 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

```

1 IJKFF_Pipeline *ffpipeline_create_from_ios(FFPlayer *ffp)
2 {
3     IJKFF_Pipeline *pipeline = ffpipeline_alloc(&g_pipeline_class, sizeof(IJKFF_Pipeline_Opaque));
4     if (!pipeline)
5         return pipeline;
6     IJKFF_Pipeline_Opaque *opaque = pipeline->opaque;
7     opaque->ffp = ffp;
8     pipeline->func_destroy = func_destroy;
9     pipeline->func_open_video_decoder = func_open_video_decoder;
10    pipeline->func_open_audio_output = func_open_audio_output;
11    return pipeline;
12 }

```

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

```

1 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。

```

1 static VideoState *stream_open(FFPlayer *ffp, const char *filename, AVInputFormat *iformat)
2 {

```

```

3      .....
4      /* start video display */
5      if (frame_queue_init(&is->pictq, &is->videoq, ffp->pictq_size, 1) < 0)
6          goto fail;
7      if (frame_queue_init(&is->sampq, &is->audioq, SAMPLE_QUEUE_SIZE, 1) < 0)
8          goto fail;
9      if (packet_queue_init(&is->videoq) < 0 ||
10         packet_queue_init(&is->audioq) < 0 )
11         goto fail;
12      .....
13
14      is->video_refresh_tid = SDL_CreateThreadEx(&is->_video_refresh_tid, video_refresh_thread, ffp, "ff_vout");
15
16      .....
17
18      is->read_tid = SDL_CreateThreadEx(&is->_read_tid, read_thread, ffp, "ff_read");
19
20      .....
21 }

```

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

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

1. 设置中断函数，如果出错或者退出，就可以立刻退出

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

1. 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

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

1. 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

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

1. 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。

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

1. 读取媒体数据，得到的是音视频分离的解码前数据

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

1. 将音视频数据分别送入相应的queue中

```
1 if (pkt->stream_index == is->audio_stream && pkt_in_play_range) {
2     packet_queue_put(&is->audioq, pkt);
3 } else if (pkt->stream_index == is->video_stream && pkt_in_play_range && !(is->video_st && (is->video_st->disposition & AV_DISPOSITION_ATTACHED_PIC))) {
4     packet_queue_put(&is->videoq, pkt);
5     .....
6 } else {
7     av_packet_unref(pkt);
8 }
```

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：

```

1 static int stream_component_open(FFPlayer *ffp, int stream_index)
2 {
3     .....
4     codec = avcodec_find_decoder(avctx->codec_id);
5     .....
6     if ((ret = avcodec_open2(avctx, codec, &opts)) < 0) {
7         goto fail;
8     }
9     .....
10    case AVMEDIA_TYPE_VIDEO:
11        .....
12        decoder_init(&is->viddec, avctx, &is->videoq, is->continuu
e_read_thread);
13        ffp->node_vdec = ffpipeline_open_video_decoder(ffp->pipel
ine, ffp);
14        if (!ffp->node_vdec)
15            goto fail;
16        if ((ret = decoder_start(&is->viddec, video_thread, ffp,
"ff_video_dec")) < 0)
17            goto out;
18        .....
19 }

```

首先会打开ffmpeg的解码器，然后通过 `ffpipeline_open_video_decoder` 创建 `IJKFF_Pipenode`。第二章中有介绍，在创建 `IJKMediaPlayer` 对象时，通过 `ffpipeline_create_from_ios` 创建了 pipeline，则

```

1 IJKFF_Pipenode* ffpipeline_open_video_decoder(IJKFF_Pipeline *pipe
line, FFPlayer *ffp)
2 {
3     return pipeline->func_open_video_decoder(pipeline, ffp);
4 }

```

`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：

```

1 static IJKFF_Pipenode *func_open_video_decoder(IJKFF_Pipeline *pi
peline, FFPlayer *ffp)

```



```

2 {
3     IJKFF_Pipenode* node = NULL;
4     IJKFF_Pipeline_Opaque *opaque = pipeline->opaque;
5     if (ffp->videotoolbox) {
6         node = ffpipenode_create_video_decoder_from_ios_videotoolb
ox(ffp);
7         if (!node)
8             ALOGE("vtb fail!!! switch to ffmpeg decode!!!! \n");
9     }
10    if (node == NULL) {
11        node = ffpipenode_create_video_decoder_from_ffplay(ffp);
12        ffp->stat.vdec_type = FFP_PROPV_DECODER_AVCODEC;
13        opaque->is_videotoolbox_open = false;
14    } else {
15        ffp->stat.vdec_type = FFP_PROPV_DECODER_VIDEOTOOBOX;
16        opaque->is_videotoolbox_open = true;
17    }
18    ffp_notify_msg2(ffp, FFP_MSG_VIDEO_DECODER_OPEN, opaque->is_
videotoolbox_open);
19    return node;
20 }

```

如果配置了ffp->videotoolbox，会优先去尝试打开硬件解码器，

```

1 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

```

1 node = ffpipenode_create_video_decoder_from_ffplay(ffp);

```

ffp->videotoolbox需要在起播前通过如下方法配置：

```

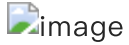
1 ijkmp_set_option_int(_mediaPlayer, IJKMP_OPT_CATEGORY_PLAYER,    "v
ideotoolbox", 1);

```

### 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

```
1 static int video_thread(void *arg)
2 {
3     FFPlayer *ffp = (FFPlayer *)arg;
4     int      ret = 0;
5     if (ffp->node_vdec) {
6         ret = ffpipenode_run_sync(ffp->node_vdec);
7     }
8     return ret;
9 }
```

`ffpipenode_run_sync`中调用的是IJKFF\_Pipenode对象中的`func_run_sync`

```
1 int ffpipenode_run_sync(IJKFF_Pipenode *node)
2 {
3     return node->func_run_sync(node);
4 }
```

`func_run_sync`取决于播放前配置的软硬解，假设为软解，则调用

```
1 static int ffplay_video_thread(void *arg)
2 {
3     FFPlayer *ffp = arg;
4
5     .....
6     for (;;) {
7         ret = get_video_frame(ffp, frame);
8         .....
9         ret = queue_picture(ffp, frame, pts, duration, av_frame_g
10         et_pkt_pos(frame), is->viddec.pkt_serial);
11     }
12     return 0;
13 }
```

```
12 }
```

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

```
1 static int decoder_decode_frame(FFPlayer *ffp, Decoder *d, AVFrame
  e *frame, AVSubtitle *sub) {
2     int got_frame = 0;
3     do {
4         int ret = -1;
5         .....
6         if (!d->packet_pending || d->queue->serial != d->pkt_seri
al){
7             AVPacket pkt;
8             do {
9                 .....
10                if (packet_queue_get_or_buffering(ffp, d->queue,
&pkt, &d->pkt_serial, &d->finished) < 0)
11                    return -1;
12                .....
13            } while (pkt.data == flush_pkt.data || d->queue->seri
al != d->pkt_serial);
14            .....
15        }
16        switch (d->avctx->codec_type) {
17            case AVMEDIA_TYPE_VIDEO: {
18                ret = avcodec_decode_video2(d->avctx, frame, &got
_frame, &d->pkt_temp);
19                .....
20            }
21            break;
22        }
23        .....
24    } while (!got_frame && !d->finished);
25    return got_frame;
26 }
```

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_1`方法中被创建：

```
1 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_output`，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`

```
1 static SDL_Aout *func_open_audio_output(IJKFF_Pipeline *pipeline,
    FFPlayer *ffp)
2 {
3     return SDL_AoutIos_CreateForAudioUnit();
4 }
```

`SDL_AoutIos_CreateForAudioUnit`定义如下，主要完成的是创建SDL\_Aout对象

```
1 SDL_Aout *SDL_AoutIos_CreateForAudioUnit()
2 {
3     SDL_Aout *aout = SDL_Aout_CreateInternal(sizeof(SDL_Aout_Opaque));
4     if (!aout)
5         return NULL;
6     // SDL_Aout_Opaque *opaque = aout->opaque;
7     aout->free_l = aout_free_l;
8     aout->open_audio = aout_open_audio;
9     aout->pause_audio = aout_pause_audio;
10    aout->flush_audio = aout_flush_audio;
11    aout->close_audio = aout_close_audio;
12    aout->func_set_playback_rate = aout_set_playback_rate;
13    aout->func_set_playback_volume = aout_set_playback_volume;
14    aout->func_get_latency_seconds = aout_get_latency_seconds;
15    aout->func_get_audio_persecond_callbacks = aout_get_persecond
        _callbacks;
16    return aout;
17 }
```

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。

```
1 static int audio_open(FFPlayer *opaque, int64_t wanted_channel_la
  yout, int wanted_nb_channels, int wanted_sample_rate, struct Audi
  oParams *audio_hw_params)
2 {
3     FFPlayer *ffp = opaque;
4     VideoState *is = ffp->is;
5     SDL_AudioSpec wanted_spec, spec;
6     .....
7     wanted_nb_channels = av_get_channel_layout_nb_channels(wanted
  _channel_layout);
8     wanted_spec.channels = wanted_nb_channels;
9     wanted_spec.freq = wanted_sample_rate;
10    wanted_spec.format = AUDIO_S16SYS;
11    wanted_spec.silence = 0;
12    wanted_spec.samples = FFMAX(SDL_AUDIO_MIN_BUFFER_SIZE, 2 << a
  v_log2(wanted_spec.freq / SDL_AoutGetAudioPerSecondCallbacks(ffp-
  >aout)));
13    wanted_spec.callback = sdl_audio_callback;
14    wanted_spec.userdata = opaque;
15    while (SDL_AoutOpenAudio(ffp->aout, &wanted_spec, &spec) < 0)
  {
16        .....
17    }
18    .....
19    return spec.size;
20 }
```

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

```
1 int SDL_AoutOpenAudio(SDL_Aout *aout, const SDL_AudioSpec *desire
  d, SDL_AudioSpec *obtained)
2 {
3     if (aout && desired && aout->open_audio)
4         return aout->open_audio(aout, desired, obtained);
5     return -1;
```

```
6 }
```

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:

```
1 static void video_image_display2(FFPlayer *ffp)
2 {
3     VideoState *is = ffp->is;
4     Frame *vp;
5     Frame *sp = NULL;
6     vp = frame_queue_peek_last(&is->pictq);
7     .....
8
9     SDL_VoutDisplayYUVOverlay(ffp->vout, vp->bmp);
10    .....
11 }
```

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

```
1 int SDL_VoutDisplayYUVOverlay(SDL_Vout *vout, SDL_VoutOverlay *overlay)
2 {
3     if (vout && overlay && vout->display_overlay)
4         return vout->display_overlay(vout, overlay);
5     return -1;
6 }
```

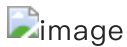
1. `display_overlay`函数指针在前面初始化流程有介绍过, 它在

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

1. 方法中被赋值为`vout_display_overlay`, 该方法就是调用OpenGL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.
9             0));
10         remaining_time = REFRESH_RATE;
11         if (is->show_mode != SHOW_MODE_NONE && (!is->paused || is
12             ->force_refresh))
13             video_refresh(ffp, &remaining_time);
14     }
15     return 0;
16 }
```

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, V
    ideoState *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_SY
            NC_THRESHOLD_MAX, delay));
13         /* -- by bbcallen: replace is->max_frame_duration with AV
            _NOSYNC_THRESHOLD */
14         if (!isnan(diff) && fabs(diff) < AV_NOSYNC_THRESHOLD) {
15             if (diff <= -sync_threshold)
16                 delay = FFMAX(0, delay + diff);
17             else if (diff >= sync_threshold && delay > AV_SYNC_FR
                AMEDUP_THRESHOLD)
18                 delay = delay + diff;
19             else if (diff >= sync_threshold)
20                 delay = 2 * delay;
21         }
22     }
23     .....
24     return delay;
25 }

```



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;
2 if (isnan(is->frame_timer) || time < is->frame_timer)
3     is->frame_timer = time;
4 if (time < is->frame_timer + delay) {
5     *remaining_time = FFMIN(is->frame_timer + delay - time, *remaining_time);
6     goto display;
7 }
```

`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。

```
1 is->frame_timer += delay;
2 if (delay > 0 && time - is->frame_timer > AV_SYNC_THRESHOLD_MAX)
3     is->frame_timer = time;
4 SDL_LockMutex(is->pictq.mutex);
5 if (!isnan(vp->pts))
6     update_video_pts(is, vp->pts, vp->pos, vp->serial);
7 SDL_UnlockMutex(is->pictq.mutex);
8 if (frame_queue_nb_remaining(&is->pictq) > 1) {
9     Frame *nextvp = frame_queue_peek_next(&is->pictq);
10    duration = vp_duration(is, vp, nextvp);
11    if(!is->step && (ffp->framedrop > 0 || (ffp->framedrop && get_master_sync_type(is) != AV_SYNC_VIDEO_MASTER)) && time > is->frame_timer + duration) {
12        frame_queue_next(&is->pictq);
13        goto retry;
```

```

14     }
15 }

```

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。

```

1 {
2     frame_queue_next(&is->pictq);
3     is->force_refresh = 1;
4     SDL_LockMutex(ffp->is->play_mutex);
5
6     .....
7
8 display:
9     /* display picture */
10    if (!ffp->display_disable && is->force_refresh && is->show_mode == SHOW_MODE_VIDEO && is->pictq.rindex_shown)
11        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：

```

1 #define FFP_MSG_FLUSH                0
2 #define FFP_MSG_ERROR                100    /* arg1 = error */
3 #define FFP_MSG_PREPARED              200
4 #define FFP_MSG_COMPLETED             300
5 #define FFP_MSG_VIDEO0_SIZE_CHANGED  400    /* arg1 = width, arg2 = height */
6 #define FFP_MSG_SAR_CHANGED           401    /* arg1 = sar.num, arg2 = sar.den */
7 #define FFP_MSG_VIDEO0_RENDERING_START 402
8 #define FFP_MSG_AUDIO0_RENDERING_START 403

```

```

9 #define FFP_MSG_VIDEO_ROTATION_CHANGED      404      /* arg1 = degree */
10 #define FFP_MSG_BUFFERING_START             500
11 #define FFP_MSG_BUFFERING_END               501
12 #define FFP_MSG_BUFFERING_UPDATE            502      /* arg1 = buffering head position in time, arg2 = minimum percent in time or bytes */
13 #define FFP_MSG_BUFFERING_BYTES_UPDATE      503      /* arg1 = cached data in bytes, arg2 = high water mark */
14 #define FFP_MSG_BUFFERING_TIME_UPDATE       504      /* arg1 = cached duration in milliseconds, arg2 = high water mark */
15 #define FFP_MSG_SEEK_COMPLETE               600      /* arg1 = seek position, arg2 = error */
16 #define FFP_MSG_PLAYBACK_STATE_CHANGED      700
17 #define FFP_MSG_TIMED_TEXT                  800
18 #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:

```

1 - (id)initWithContentURLString:(NSString *)aUrlString
2         withOptions:(IJKFFOptions *)options
3 {
4     .....
5     // init player
6     _mediaPlayer = ijkmp_ios_create(media_player_msg_loop);
7     .....
8 }

```

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

```

1 IjkMediaPlayer *ijkmp_create(int (*msg_loop)(void*))
2 {
3     .....
4     mp->msg_loop = msg_loop;
5     .....

```

```
6 }
```

开始播放时，会启动一个消息线程，

```
1 static int ijkmpp_prepare_async_l(IjkMediaPlayer *mp)
2 {
3     .....
4     mp->msg_thread = SDL_CreateThreadEx(&mp->_msg_thread, ijkmpp_ms
      g_loop, mp, "ff_msg_loop");
5     .....
6 }
```

`ijkmpp_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：

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

`ffp_notify_msg1`方法实现如下：

```
1 inline static void ffp_notify_msg1(FFPlayer *ffp, int what) {
2     msg_queue_put_simple3(&ffp->msg_queue, what, 0, 0);
3 }
```

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

```
1 inline static void msg_queue_put_simple3(MessageQueue *q, int wha
  t, int arg1, int arg2)
2 {
3     AVMessage msg;
4     msg_init_msg(&msg);
```

```

5     msg.what = what;
6     msg.arg1 = arg1;
7     msg.arg2 = arg2;
8     msg_queue_put(q, &msg);
9 }

```

继续跟踪代码，可以发现最后在

```

1 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：

```

1 int media_player_msg_loop(void* arg)
2 {
3     @autoreleasepool {
4         IjkMediaPlayer *mp = (IjkMediaPlayer*)arg;
5         __weak IJKFFMoviePlayerController *ffpController = ffplay
erRetain(ijkmp_set_weak_thiz(mp, NULL));
6         while (ffpController) {
7             @autoreleasepool {
8                 IJKFFMoviePlayerMessage *msg = [ffpController obt
ainMessage];
9                 if (!msg)
10                     break;
11                 int retval = ijkmp_get_msg(mp, &msg->_msg, 1);
12                 if (retval < 0)
13                     break;
14                 // block-get should never return 0
15                 assert(retval > 0);
16                 [ffpController performSelectorOnMainThread:@selec
tor(postEvent:) withObject:msg waitUntilDone:NO];
17             }
18         }
19         // retained in prepare_async, before SDL_CreateThreadEx
20         ijkmp_dec_ref_p(&mp);
21         return 0;

```

```
22     }  
23 }
```

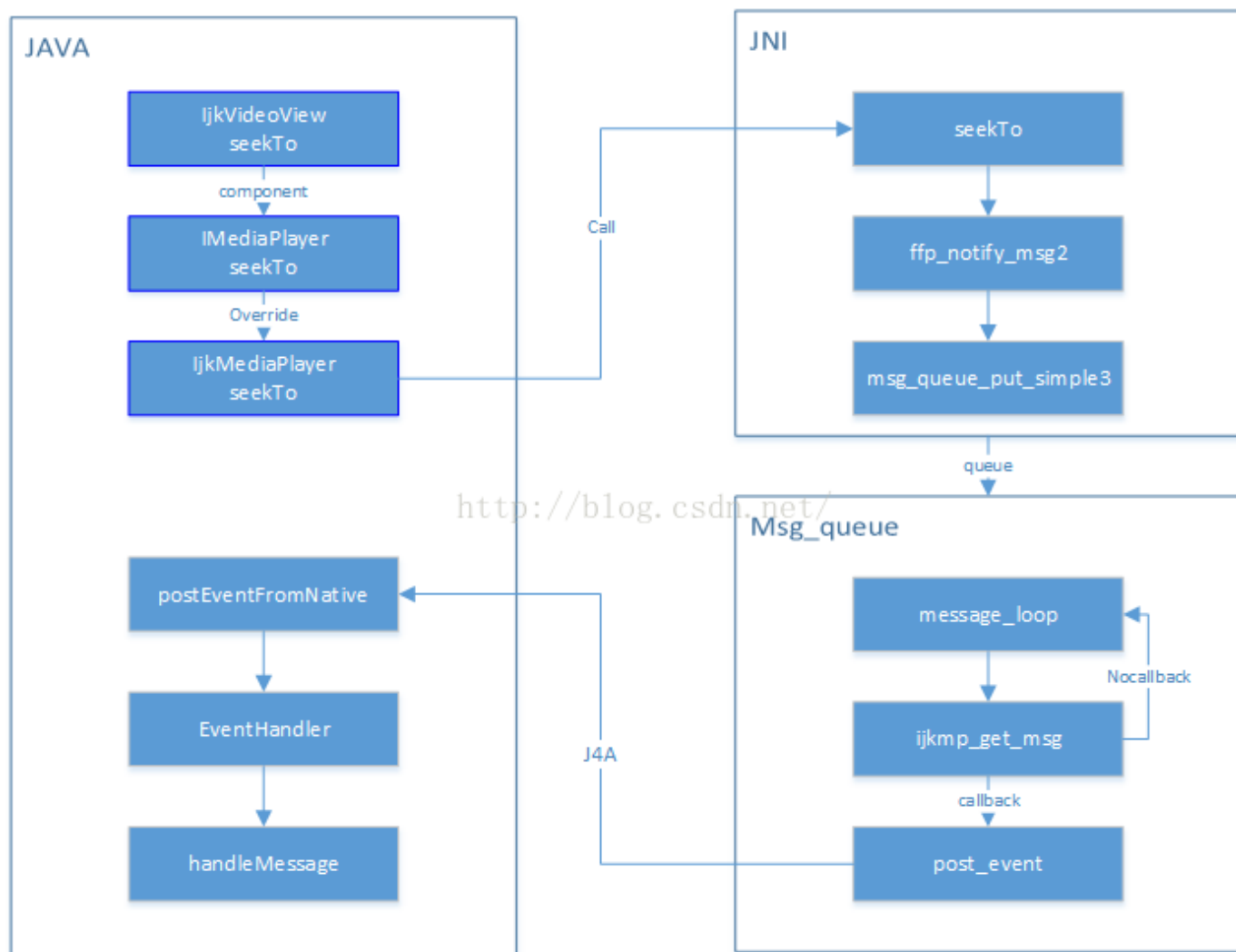
由此可以看出，最后是在该方法中读取消息，并采用notification通知到APP上层。

## 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](https://www.jianshu.com)



作者：金山视频云

链接：<https://www.jianshu.com/p/daf0a61cc1e0>

来源：简书

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。