**ijkplayer native播放分析**

Java层ijkplayer启动调用native ijkplayer的流程步骤：

1、创建ijkplayer

2、设置listener、option等参数

3、setDataSource

4、prepareAsync

5、start

**1、创建ijkplayer**

Java层ijkplayer构造函数执行流程：

1.1、加载ijkffmpeg、ijksdl、ijkplayer三个动态库

1.2、native\_init

1.3、native\_setup

**1.1、ijkplayer\_jni.c JNI\_OnLoad**

**1.1.1、得到jclass**

得到JNIEnv\* env、jclass指向java类"tv/danmaku/ijk/media/player/IjkMediaPlayer"

RegisterNatives映射本地方法与java方法。

**1.1.2、RegisterNatives**

(\*env)->RegisterNatives(env, g\_clazz.clazz, g\_methods, NELEM(g\_methods) )

映射native方法与java方法

**1.1.3、ijkmp\_global\_init**

ijkplayer.c ijkmp\_global\_init

Ff\_ffplay.c ffp\_global\_init

ffmpeg初始化 ：

Allcodecs.c avcodec\_register\_all注册encoder、decoder、parser。

Allformats.c av\_register\_all注册muxer、demuxer。

ijkav\_register\_all注册protocol，注意ijkmediadatasource protocol。

Utils.c avformat\_network\_init打开网络流播放

av\_lockmgr\_register av\_log\_set\_callback设置log输出函数

av\_init\_packet初始化flush\_pkt

**1.1.4、ijkmp\_global\_set\_inject\_callback(inject\_callback)**

ijkplayer.c ijkmp\_global\_set\_inject\_callback

Ff\_ffplay.c ffp\_global\_set\_inject\_callback

设置ff\_ffplayer中的全局变量s\_inject\_callback，回调函数：从ff\_ffplayer回调jni代码中的inject\_callback，内部调用J4AC\_Bundle\_\_Bundle\_\_catchAll等，是ijkplayer在jni中调用java类及java方法的一种抽象封装方法，可以在native c语言文件中直接调用J4AC\*类接口间接操作java对象和方法。这里创建了一个bundle，之后调用J4AC\_IjkMediaPlayer\_\_onNativeInvoke，回调IjkMediaPlayer的onNativeInvoke方法，传递的消息内容封装在一个bundle对象中。

**1.1.5、FFmpegApi\_global\_init(env)**

ffmpeg\_api\_jni.c IJK\_FIND\_JAVA\_CLASS找到java类jclass "tv/danmaku/ijk/media/player/ffmpeg/FFmpegApi"

映射java中的本地方法av\_base64\_encode到jni方法FFmpegApi\_av\_base64\_encode，java中可调用native的av\_base64\_encode。

**1.1.6、FFmpegMediaMetadataRetriever\_global\_init**

Java 类"com/xmly/media/FFmpegMediaMetadataRetriever"

nativeMethods：setDataSource、extractMetadata、extractMetadataFromChapter、native\_getMetadata等 。

**1.2、ijksdl\_android\_jni.c JNI\_OnLoad**

j4a\_allclasses.c

J4A\_LoadAll\_\_catchAll

{

J4A\_LOAD\_CLASS(android\_os\_Build);

J4A\_LOAD\_CLASS(java\_nio\_Buffer);

J4A\_LOAD\_CLASS(java\_nio\_ByteBuffer);

J4A\_LOAD\_CLASS(java\_util\_ArrayList);

J4A\_LOAD\_CLASS(android\_media\_AudioTrack);

J4A\_LOAD\_CLASS(android\_media\_MediaCodec);

J4A\_LOAD\_CLASS(android\_media\_MediaFormat);

J4A\_LOAD\_CLASS(android\_media\_PlaybackParams);

J4A\_LOAD\_CLASS(android\_os\_Build);

J4A\_LOAD\_CLASS(android\_os\_Bundle)

J4A\_LOAD\_CLASS(tv\_danmaku\_ijk\_media\_player\_misc\_IMediaDataSource)

J4A\_LOAD\_CLASS(tv\_danmaku\_ijk\_media\_player\_IjkMediaPlayer)

}

j4a\_base.h

#define J4A\_LOAD\_CLASS(class\_\_) \

do { \

ret = J4A\_loadClass\_\_J4AC\_##class\_\_(env); \

if (ret) \

goto fail; \

} while (0)

宏定义，调用J4A\_loadClass\_\_J4AC\_##class\_\_(env)

**1.2.1、android\_os\_Build**

build.c

J4A\_loadClass\_\_J4AC\_android\_os\_Build

通过J4A\_FindClass\_\_asGlobalRef\_\_catchAll获取java类"android/os/Build"的jclass，并创建一个全局的引用，代码如下：

{

J4A\_FindClass\_\_catchAll

{

jclass clazz = (\*env)->FindClass(env, class\_sign)

}

J4A\_NewGlobalRef\_\_catchAll

{

jclass obj\_global = (\*env)->NewGlobalRef(env, obj);全局引用

}

}

**1.2.2、android\_media\_AudioTrack**

AudioTrack.c

J4A\_loadClass\_\_J4AC\_android\_media\_AudioTrack

{

J4A\_FindClass\_\_asGlobalRef\_\_catchAll找到java类"android/media/AudioTrack"并创建全局引用jclass对象

J4A\_GetMethodID\_\_catchAll得到java类中方法ID：jmethodID

}

总结：J4A封装了从natvie到java的jni方法，封装的java类有：AudioTrack、MediaCodec、MediaFormat等；自定义类有： IMediaDataSource、 IjkMediaPlayer等。

**1.3、native\_init**

啥也没干，打印log。

**1.4、native\_setup**

static void IjkMediaPlayer\_native\_setup(JNIEnv \*env, jobject thiz, jobject weak\_this)

{

1.4.1、IjkMediaPlayer \*mp = ijkmp\_android\_create(message\_loop);

1.4.2、jni\_set\_media\_player(env, thiz, mp);

1.4.3、ijkmp\_set\_weak\_thiz(mp, (\*env)->NewGlobalRef(env, weak\_this));

1.4.4、ijkmp\_set\_inject\_opaque(mp, ijkmp\_get\_weak\_thiz(mp));

1.4.5、ijkmp\_android\_set\_mediacodec\_select\_callback(mp, mediacodec\_select\_callback, ijkmp\_get\_weak\_thiz(mp));

LABEL\_RETURN:

ijkmp\_dec\_ref\_p(&mp);

}

Setup主要创建了native IjkMediaPlayer对象并初始化、MediaCodec的选择等。

**1.4.1、native ijkplayer初始化**

**ijkmp\_android\_create**

IjkMediaPlayer \*mp = ijkmp\_android\_create(message\_loop)

ijkmp\_create(msg\_loop)

（IjkMediaPlayer \*）mp->msg\_loop = msg\_loop;

赋值native IjkMediaPlayer的消息回调函数msg\_loop。

**1.4.1.1、message\_loop**

把native IjkMediaPlayer的通知消息抛给java IjkMediaPlayer对象。

ijkmp\_get\_weak\_thiz从本地对象中取出java对象指针jobject

message\_loop\_n

While（1）循环执行：是一个线程

ijkmp\_get\_msg

msg\_queue\_get

从IjkMediaPlayer→ffplayer→msg\_queue中读取消息通知，没有则SDL\_CondWait。例：如果读出来是FFP\_MSG\_PREPARED会调用ijkmp\_change\_state\_l最后通过msg\_queue\_put\_private添加一个状态变化的消息FFP\_MSG\_PLAYBACK\_STATE\_CHANGED到ffplayer->msg\_queue。另外有一些变量、状态设置。

得到msg之后返回message\_loop\_n执行：

case FFP\_MSG\_PREPARED执行post\_event(MEDIA\_PREPARED)，回调IjkMediaPlayer的postEventFromNative通过J4AC\_IjkMediaPlayer\_\_postEventFromNative。

case FFP\_MSG\_PLAYBACK\_STATE\_CHANGED: break;收到状态变化消息时不作任何处理。

**1.4.1.2、ijkmp\_create**

IjkMediaPlayer \*mp = (IjkMediaPlayer \*) mallocz(sizeof(IjkMediaPlayer))

内存申请，创建一个native IjkMediaPlayer对象。

mp->ffplayer = ffp\_create();创建一个FFPlayer对象ffp

msg\_queue\_init(&ffp→msg\_queue); ffp中消息队列 msg\_queue初始化

ffp->av\_class = &ffp\_context\_class; FFPlayer的AvClass，包含AVOption。

ffp->meta = ijkmeta\_create(); 创建IjkMediaMeta

av\_opt\_set\_defaults(ffp);AVOption初始化

mp->msg\_loop = msg\_loop;赋值通知消息回调函数。

创建完成native IjkMediaPlayer对象

**1.4.1.3、SDL\_VoutAndroid\_CreateForAndroidSurface**

SDL\_VoutAndroid\_CreateForANativeWindow

SDL\_Vout \*vout = SDL\_Vout\_CreateInternal创建SDL\_Vout，video输出。

主要成员有SDL\_Vout\_Opaque、方法create\_overlay、方法display\_overlay。

SDL\_Vout\_Opaque \*opaque = vout->opaque;

opaque->egl = IJK\_EGL\_create();

vout->opaque\_class = &g\_nativewindow\_class; "ANativeWindow\_Vout"

vout->create\_overlay = func\_create\_overlay;

vout->free\_l = func\_free\_l;

vout->display\_overlay = func\_display\_overlay;

A、SDL\_Vout\_Opaque

主要成员：ANativeWindow（Android渲染器）、SDL\_AMediaCodec（java MediaCodec）、IJK\_EGL（OpenGL）

SDL\_AMediaCodec分析：

{

SDL\_AmediaCodec\_Opaque

{

jobject android\_media\_codec; java MediaCodec对象

}

(\*func\_start)

(\*func\_stop)

(\*func\_flush)

(\*func\_writeInputData)

(\*func\_dequeueInputBuffer)

(\*func\_queueInputBuffer)

(\*func\_dequeueOutputBuffer)

(\*func\_getOutputFormat)

(\*func\_releaseOutputBuffer)

以上函数指针通过J4A一一对应java MediaCodec方法。

}

总结：SDL\_AMediaCodec是对java mediacodec类的封装，类似代理。

IJK\_EGL分析：

{

IJK\_EGL\_Opaque \*opaque;

{

IJK\_GLES2\_Renderer\_Opaque没有实现

(\*func\_use)

(\*func\_getBufferWidth)

(\*func\_uploadTexture)

(\*func\_destroy)

}

EGLNativeWindowType window;//不同的平台定义不同，android平台定义是ANativeWindow

EGLDisplay display; 空

EGLSurface surface; 空

EGLContext context; 空

EGLint width;

EGLint height;

}

总结：封装ANativeWindow，方法 func\_use没看到具体实现？？

B、SDL\_VoutOverlay

主要成员：SDL\_ VoutOverlay\_Opaque、方法func\_fill\_frame

struct SDL\_VoutOverlay {

Uint32 format; /\*\*< Read-only \*/ 输出帧pix format

Uint16 \*pitches; /\*\*< in bytes, Read-only \*/像素数据

Uint8 \*\*pixels; /\*\*< Read-write \*/像素数据

SDL\_VoutOverlay\_Opaque \*opaque;

int (\*func\_fill\_frame)(SDL\_VoutOverlay \*overlay, const AVFrame \*frame);把frame数据填充到overlay的pixels中。

};

SDL\_ VoutOverlay\_Opaque的定义有两个，硬解mediacodec的定义如下：

{

SDL\_Vout \*vout;

SDL\_AMediaCodec \*acodec; java MediaCodec类对象

SDL\_AMediaCodecBufferProxy \*buffer\_proxy;

Uint16 pitches[AV\_NUM\_DATA\_POINTERS]; 像素数据相关

Uint8 \*pixels[AV\_NUM\_DATA\_POINTERS]; 像素数据相关

}

SDL\_ VoutOverlay\_Opaque的定义有两个，软解ffmpeg 的定义如下：

{

AVFrame \*managed\_frame; 软解管理帧，和swscale有关

AVBufferRef \*frame\_buffer;

int planes;

AVFrame \*linked\_frame;

Uint16 pitches[AV\_NUM\_DATA\_POINTERS];像素数据相关

Uint8 \*pixels[AV\_NUM\_DATA\_POINTERS]; 像素数据相关

int no\_neon\_warned;

struct SwsContext \*img\_convert\_ctx; 软解解码后数据二次处理，转成目标输出格式，例如YUV420P等。

}

D、Overlay

vout->create\_overlay = func\_create\_overlay;

vout->display\_overlay = func\_display\_overlay;

根据不同的解码器，实现不同，在后面prepare执行时再做具体分析。

**1.4.1.4、ffpipeline\_create\_from\_android**

创建IJKFF\_Pipeline

pipeline->opaque\_class = opaque\_class; "ffpipeline\_android\_media"

pipeline->func\_destroy = func\_destroy;

pipeline->func\_open\_video\_decoder = func\_open\_video\_decoder;

pipeline->func\_open\_audio\_output = func\_open\_audio\_output;

IJKFF\_Pipeline结构体分析：

{

SDL\_Class \*opaque\_class;

IJKFF\_Pipeline\_Opaque \*opaque;

(\*func\_destroy)

(\*func\_open\_video\_decoder)

(\*func\_open\_audio\_output)

};

IJKFF\_Pipeline\_Opaque分析：

{

FFPlayer \*ffp; 存FFPlayer对象

jobject jsurface; java surface对象

(\*mediacodec\_select\_callback) 选择mediacodec，回调函数，到java

SDL\_Vout \*weak\_vout;

}

**1.4.1.5、ffpipeline\_set\_vout**

IJKFF\_Pipeline\_Opaque \*opaque = pipeline->opaque;

opaque->weak\_vout = vout;

SDL\_VoutAndroid\_CreateForAndroidSurface创建的SDL\_Vout保存在pipeline->opaque中。方便解码后的数据输出到SDL\_Vout。

**1.4.2、jni\_set\_media\_player**

把上一步创建的mp设置到java IjkMediaPlayer类的mNativeMediaPlayer中，并把older之前的释放。

**1.4.3、ijkmp\_set\_weak\_thiz**

Native IjkMediaPlayer mp->weak\_thiz = weak\_thiz保存一个java IjkMediaPlayer对象的引用。

**1.4.4、ijkmp\_set\_inject\_opaque**

在FFPlayer中保存java IjkMediaPlayer对象，赋值给inject\_opaque，app\_ctx是AVApplicationContext，成员opaque存FFPlayer对象，并把app\_ctx保存在ijkapplication option中

AVApplicationContext func\_on\_app\_event设置为app\_func\_event，功能：回调java IjkMediaPlayer onNativeInvoke

**1.4.5、ijkmp\_android\_set\_mediacodec\_select\_callback**

ffpipeline\_set\_mediacodec\_select\_callback

设置pipeline mediacodec\_select\_callback

pipeline->opaque->mediacodec\_select\_callback = callback;

pipeline->opaque->mediacodec\_select\_callback\_opaque = opaque;

回调java IjkMediaPlayer的mediacodec\_select\_callback，需要java IjkMediaPlayer对象引用和回调方法。

**2、配置选项**

Listener在java player中调用，native代码只通过msg loop抛消息给java player。

分析下setOption的关键变量：

ijkMediaPlayer.setOption(IjkMediaPlayer.OPT\_CATEGORY\_PLAYER, "mediacodec

ijkMediaPlayer.setOption(IjkMediaPlayer.OPT\_CATEGORY\_PLAYER,"overlay-format

以上变量关系到解码器的选择、渲染输出帧的pix format。

**2.1、ffp->av\_class**

先分析下FFPlayer的option定义：

FFPlayer创建时，ffp->av\_class = &ffp\_context\_class;

const AVClass ffp\_context\_class = {

.class\_name = "FFPlayer",

.item\_name = ffp\_context\_to\_name,

.option = ffp\_context\_options,

.version = LIBAVUTIL\_VERSION\_INT,

.child\_next = ffp\_context\_child\_next,

.child\_class\_next = ffp\_context\_child\_class\_next,

};

Option ffp\_context\_options定义如下，取一部分分析：

static const AVOption ffp\_context\_options[] = {

// original options in ffplay.c

{ "overlay-format", "fourcc of overlay format",

OPTION\_OFFSET(overlay\_format), OPTION\_INT(SDL\_FCC\_RV32, INT\_MIN, INT\_MAX),

.unit = "overlay-format" },

// Android only options

{ "mediacodec", "MediaCodec: enable H264 (deprecated by 'mediacodec-avc')",

OPTION\_OFFSET(mediacodec\_avc), OPTION\_INT(0, 0, 1) },

{ "opensles", "OpenSL ES: enable",

OPTION\_OFFSET(opensles), OPTION\_INT(0, 0, 1) },

{ NULL } 遍历结束标志

}

**2.2、setOption mediacodec**

setOption到ffp\_set\_option

AVDictionary \*\*dict = ffp\_get\_opt\_dict(ffp, opt\_category);

switch (opt\_category)

{

case FFP\_OPT\_CATEGORY\_FORMAT: return &ffp->format\_opts;

case FFP\_OPT\_CATEGORY\_CODEC: return &ffp->codec\_opts;

case FFP\_OPT\_CATEGORY\_SWS: return &ffp->sws\_dict;

case FFP\_OPT\_CATEGORY\_PLAYER: return &ffp->player\_opts;

case FFP\_OPT\_CATEGORY\_SWR: return &ffp->swr\_opts;

}

取出ffp->player\_opts，以overlay-format、mediacodec、opensles为例

av\_dict\_set(dict, name, value, 0);

添加key-value键值对：overlay-format及其value值、mediacodec及其value值、opensles及其value值。

在prepare时，av\_opt\_set\_dict(ffp, &ffp->player\_opts)

av\_opt\_set\_dict2

While

{

av\_dict\_get获取player\_opts中的值AVDictionaryEntry，里面有前面设置的key-value

av\_opt\_set把key-value写到FFPlayer ffp中与key同名的变量中。

}

av\_opt\_set，找到 ffp中对应AVDictionaryEntry key的AVOption（根据name），并把value值赋给AVOption，看AVOption实现：OPTION\_OFFSET(overlay\_format)，这里是取 overlay\_format在FFPlayer结构体中的偏移量，加上指针ffp得到overlay\_format的地址，并给它赋值，既给FFPlayer overlay\_format赋值。同样道理给mediacodec-avc赋值。

比如我们分别在setting中给mediacodec-avc赋值1、overlay\_format为yv12。

**2.3、setSurface、setDisplay**

IjkMediaPlayer\_setVideoSurface

ijkmp\_android\_set\_surface

ijkmp\_android\_set\_surface\_l

2.3.1、SDL\_VoutAndroid\_SetAndroidSurface

native\_window = ANativeWindow\_fromSurface(env, android\_surface);得到ANativeWindow

SDL\_VoutAndroid\_SetNativeWindow\_l

ANativeWindow\_acquire

opaque->native\_window = native\_window

根据surface得到ANativeWindow，赋值给vout

2.3.2、ffpipeline\_set\_surface

SDL\_VoutAndroid\_setAMediaCodec(opaque->weak\_vout, NULL);

SDL\_VoutAndroid\_setAMediaCodec\_l

把Vout中AMediaCodec清空。

opaque->jsurface = (\*env)->NewGlobalRef(env, surface)

得到一个jobject surface存到Vout中。

**启动播放器**

**3、SetDataSource**

IjkMediaPlayer\_setDataSourceCallback

**3.1、jni\_set\_media\_data\_source**

nativeMediaDataSource = jni\_set\_media\_data\_source(env, thiz, callback);

把MediaDataSource（其实也是个java对象）赋值给java player中的mNativeMediaDataSource

snprintf(uri, sizeof(uri), "ijkmediadatasource:%"PRId64"#xmly#", nativeMediaDataSource);

Header，加上协议头ijkmediadatasource:1049738#xmly#，里面含有java MediaDataSource的对象地址1049738。

J4AC\_IMediaDataSource\_\_getUri\_\_asCBuffer，调用java mediadatasource的geturi并叠加上面的header，得到最终的uri，如下：

ijkmediadatasource:1049738#xmly#/storage/emulated/legacy/DCIM/Camera/peiyinxiu/1523618724408.mp4

ijkmp\_set\_data\_source\_l

mp->data\_source = strdup(url);赋值给native ijkplayer的data\_source。作为open时的filename

**3.1.1、ijkimp\_ff\_ijkmediadatasource\_protocol查找**

Ffmpeg内部打开文件或流时，根据filename查找protocol。

ijkmediadatasource:1049738#xmly#/storage/emulated/legacy/DCIM/Camera/peiyinxiu/1523618724408.mp4

1049738是java的jobject nativeMediaDataSource 指针地址，从ffmpeg回调时使用。

Ijkmediadatasource是协议头，在ffmpeg内部根据name会找到ijkimp\_ff\_ijkmediadatasource\_protocol，定义如下：

ijkmediadatasource.c

URLProtocol ijkimp\_ff\_ijkmediadatasource\_protocol = {

.name = "ijkmediadatasource",

.url\_open2 = ijkmds\_open,

.url\_read = ijkmds\_read,

.url\_seek = ijkmds\_seek,

.url\_close = ijkmds\_close,

.priv\_data\_size = sizeof(Context),

.priv\_data\_class = &ijkmediadatasource\_context\_class,

.flags = URL\_PROTOCOL\_FLAG\_NETWORK,

.default\_whitelist = "ijkmediadatasource,http,https,tcp"

};

ijkmds\_open等函数，根据jobject 1049738调用java IMediaDataSource对象对应的open方法；read、seek同理。

**4、prepareAsync**

prepareAsync内容较多，硬解时启动7个线程，软解时启动6个线程。

ijkmp\_prepare\_async

**4.1、第一个线程ff\_msg\_loop**

**4.1.1、msg\_queue\_start**

q->abort\_request = 0;启动

AVMessage msg;

msg\_init\_msg(&msg);

msg.what = FFP\_MSG\_FLUSH;

msg\_queue\_put\_private(q, &msg);

向mp->ffplayer->msg\_queue加入消息FFP\_MSG\_FLUSH

04-26 11:31:35.861 6474 7398 D IJKMEDIA: FFP\_MSG\_FLUSH:

**4.1.2、启动消息循环ijkmp\_msg\_loop**

第一个线程：

SDL\_CreateThreadEx启动第一个线程"ff\_msg\_loop"，执行ijkmp\_msg\_loop：mp->msg\_loop(arg)，根据前面介绍这里启动callback函数：jni代码中的message\_loop；有消息时读取并处理，无消息时wait。

前面有向消息队列导入FFP\_MSG\_FLUSH消息，线程启动后Log输出：

04-26 11:31:35.861 6474 7398 D IJKMEDIA: FFP\_MSG\_FLUSH

**4.2、ijkmp\_prepare\_async\_l**

mp->data\_source作为file\_name传入，文件路径、url。

av\_opt\_set\_dict(ffp, &ffp->player\_opts);前面有介绍，这里配置ffp的AVOption，把java中setoption所设置的属性写入FFPlayer。

ffp->aout = ffpipeline\_open\_audio\_output(ffp->pipeline, ffp);创建aout

**4.2.1、ffpipeline\_open\_audio\_output**

pipeline->func\_open\_audio\_output

不同的平台有不同的实现，这里以android为例，走ffpipeline\_android.c中实现。

if (ffp->opensles) {

aout = SDL\_AoutAndroid\_CreateForOpenSLES();

} else {

aout = SDL\_AoutAndroid\_CreateForAudioTrack();

}

if (aout)

SDL\_AoutSetStereoVolume(aout, pipeline->opaque->left\_volume, pipeline->opaque->right\_volume);

ffp->opensles根据app setting选择，以opensles为0举例：

SDL\_AoutAndroid\_CreateForAudioTrack 创建SDL\_Aout

SDL\_Aout重要成员：

{

SDL\_Aout\_Opaque \*opaque

{

SDL\_AudioSpec spec;

SDL\_Android\_AudioTrack\* atrack; java AudioTrack

}

aout->opaque\_class = &g\_audiotrack\_class; name："AudioTrack"

aout->open\_audio = aout\_open\_audio;

aout->close\_audio = aout\_close\_audio;

};

SDL\_AoutSetStereoVolume

aout->set\_volume(aout, left\_volume, right\_volume);

opaque->left\_volume = left\_volume;

opaque->right\_volume = right\_volume;

给SDL\_Aout\_Opaque赋值，后面AudioTrack创建时需要。

**4.2.2、stream\_open**

VideoState \*is = stream\_open(ffp, file\_name, NULL);

stream\_open中启动的线程较多，后面详细分析。创建VideoState。

ffp->is = is; 保存在FFPLayer中

ffp->input\_filename = av\_strdup(file\_name);把data\_source存在FFPLayer

**4.2.2.1、创建VideoState**

is->filename = av\_strdup(filename);

is->iformat = iformat;这里时NULL

**4.2.2.2、frame\_queue\_init**

video和audio等流的输出帧VideoState中的pictq、sampq（FrameQueue）队列初始化。

f->pktq = pktq; 把VideoState中的PacketQueue audioq、videoq分别赋值在FrameQueue中

f->keep\_last = !!keep\_last

if (!(f->queue[i].frame = av\_frame\_alloc()))给FrameQueue中的Frame队列申请内存。

**4.2.2.3、packet\_queue\_init**

video和audio等流解码前数据包队列PacketQueue audioq、videoq初始化

q->abort\_request = 1;初始状态是abort

**4.2.2.4、init\_clock**

每一个流对应一个clock，显示和同步使用，vidclk、audclk、extclk三个。

c->queue\_serial = videoq.serial、audioq.serial，不同的队列号

set\_clock(c, NAN, -1);clock初始化。

is->av\_sync\_type = ffp->av\_sync\_type;同步基准，前面有看到，以audio的时间为准。

ffp->is = is;FFPlayer中存VideoState

is->pause\_req = !ffp->start\_on\_prepared; 这个值在app setting时是0，所以pause\_req是1。Prepare后处于暂停状态。

接下来还是stream\_open中的流程：

**4.3、第二个线程ff\_vout**

Log：04-26 11:31:35.861 6474 7399 I IJKMEDIA: SDL\_RunThread: [7399] ff\_vout

SDL\_CreateThreadEx启动video渲染线程：video\_refresh\_thread "ff\_vout"

while循环执行video\_refresh(ffp, &remaining\_time)

根据remaining\_time设定刷新频率，视频帧的刷新频率。remaining\_time默认10ms，正常由video\_refresh计算得到。

**4.3.1、显示帧的环形buffer：FrameQueue**

FrameQueue

{

Frame queue[FRAME\_QUEUE\_SIZE]; 帧数组buffer最大16个

int rindex; 正在显示的帧的下标

int windex; 帧入队地址下标，队列中最后一个有效帧的下标+1是windex

int size; 队列中有效帧的个数，包括正在显示的帧。待显示帧个数时size-1个

int max\_size;

int keep\_last;值是1，保持当前显示帧在队列中

int rindex\_shown; 初始值为0，如果keep\_last为1，第一帧显示后变为1，表明显示帧个数

PacketQueue \*pktq;指向videostate的PacketQueue

}

**4.3.2、第一帧显示**

frame\_queue\_next

if (f->keep\_last && !f->rindex\_shown) {

f->rindex\_shown = 1;

return;

}

这里显示第一帧时，把rindex\_shown = 1后返回。没有改变FrameQueue中的rindex、size值，并且不执行释放frame\_queue\_unref\_item动作。

**4.3.3、正常显示**

frame\_queue\_nb\_remaining

return f->size - f->rindex\_shown;返回待显示帧个数，不包括当前显示帧。

frame\_queue\_peek\_last

return &f->queue[f->rindex]获取当前显示帧

frame\_queue\_peek

return &f->queue[(f->rindex + f->rindex\_shown) % f->max\_size];获取将要显示的帧，正在显示帧的下一帧。

if (vp->serial != is->videoq.serial) {

frame\_queue\_next(&is->pictq); 如果取出的下一个帧不是video packet，再读取一帧。

goto retry; 返回

}

last\_duration = vp\_duration(is, lastvp, vp);计算下一帧和当前帧的显示时间差

delay = compute\_target\_delay(ffp, last\_duration, is);

计算将要显示的帧和当前时间基线之间的delay（比如audio时间）时间。

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;

}

更新frame\_timer，如果当前时间小于下一帧将要显示的时间，那么更新remaining\_time的值，确保remaining\_time是两个相邻帧之间时间delay的最小值，保证渲染线程的刷新率能跟上视频的帧率。

is->frame\_timer += delay;

if (delay > 0 && time - is->frame\_timer > AV\_SYNC\_THRESHOLD\_MAX)

is->frame\_timer = time;

更新is->frame\_timer为下一帧的显示时间。如果下一帧显示的时间小于当前time太多（阈值#define AV\_SYNC\_THRESHOLD\_MAX 0.1），更新is->frame\_timer为time，和后面的丢帧策略有关系。

update\_video\_pts更新video clock时间，如果以audio时间为准，在audio数据缺失或播放完之后，把audio clock设置为video clock ，确保视频可以播放。

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(is) != AV\_SYNC\_VIDEO\_MASTER)) && time > is->frame\_timer + duration) {

frame\_queue\_next(&is->pictq);

goto retry;

}

}

frame\_queue\_peek\_next取出第二个要显示的帧，计算时间差duration ，如果当前时间大于第二帧显示时间，说明视频显示速度明显慢于音频，frame\_queue\_next释放下一帧，丢帧，直接跳过将要显示的帧。

frame\_queue\_next(&is->pictq);

static void frame\_queue\_next(FrameQueue \*f)

{ 第一次进来会把rindex\_shown设为1后直接返回，保留显示帧。

if (f->keep\_last && !f->rindex\_shown) {

f->rindex\_shown = 1;

return;

}

frame\_queue\_unref\_item(&f->queue[f->rindex]);释放当前显示帧

if (++f->rindex == f->max\_size) 显示帧下标加1

f->rindex = 0; 循环，从头开始。

SDL\_LockMutex(f->mutex);

f->size--; 待显示帧数量减1

SDL\_CondSignal(f->cond);

SDL\_UnlockMutex(f->mutex);

}

is->force\_refresh = 1; 强制刷新标志

当前显示帧数据出队列，更新rindex、size、rindex\_shown等。video\_display2 正常显示。

video\_image\_display2

frame\_queue\_peek\_last取出下一帧，前面rindex已经加1指向下一帧。

SDL\_VoutDisplayYUVOverlay vout->display\_overlay显示，后面分析。

if (!ffp->first\_video\_frame\_rendered) {

ffp->first\_video\_frame\_rendered = 1;

ffp\_notify\_msg1(ffp, FFP\_MSG\_VIDEO\_RENDERING\_START);

}

第一帧渲染通知，通知app开始显示。

**4.3.4、display\_overlay**

android的显示只有nativewindow，ijksdl\_vout\_android\_nativewindow.c

vout->display\_overlay = func\_display\_overlay;

func\_display\_overlay\_l

从vout取出opaque，取出ANativeWindow

switch(overlay->format)根据overlay->format选择显示方式，后面会讲format怎么设置下来的，是在app setting以及解码时赋值。

switch(overlay->format) {

case SDL\_FCC\_\_AMC: {

IJK\_EGL\_terminate(opaque->egl);

return SDL\_VoutOverlayAMediaCodec\_releaseFrame\_l(overlay, NULL, true);

case SDL\_FCC\_RV24:

case SDL\_FCC\_I420:

case SDL\_FCC\_I444P10LE: {

if (opaque->egl)

return IJK\_EGL\_display(opaque->egl, native\_window, overlay);

break;

case SDL\_FCC\_YV12:

case SDL\_FCC\_RV16:

case SDL\_FCC\_RV32: {

if (vout->overlay\_format == SDL\_FCC\_\_GLES2 && opaque->egl)

return IJK\_EGL\_display(opaque->egl, native\_window, overlay);

break;

IJK\_EGL\_terminate(opaque->egl);

return SDL\_Android\_NativeWindow\_display\_l(native\_window, overlay);

}

**4.3.4.1、case SDL\_FCC\_\_AMC 硬解**

说明是MediaCodec解码：

SDL\_VoutOverlayAMediaCodec\_releaseFrame\_l

SDL\_AmediaCodec\_releaseOutputBuffer

func\_releaseOutputBuffer

硬解MediaCodec时没有真正的显示动作，只是控制MediaCodec的输出帧释放，实现音视频同步。

**4.3.4.2、其它（软解）**

IJK\_EGL\_display：OpenGLES显示

Ffmpeg：SDL\_Android\_NativeWindow\_display\_l

native\_window\_get\_desc

AnativeWindow\_setBuffersGeometry

AnativeWindow\_lock

voutDesc render

AnativeWindow\_unlockAndPost

软解显示时，直接操作ANativeWindow，graphic渲染。

**4.4、第三个线程ff\_read**

Log：04-26 11:31:35.861 6474 7400 I IJKMEDIA: SDL\_RunThread: [7400] ff\_read

read\_thread启动数据读取。ffmpeg操作：

avformat\_alloc\_context

avformat\_open\_input

avformat\_find\_stream\_info

avformat\_seek\_file

av\_find\_best\_stream 找到最佳的audio和video流

**4.4.1、stream\_component\_open audio**

Ffmpeg操作：

avcodec\_alloc\_context3

stream\_component\_open

avcodec\_find\_decoder

avcodec\_open2

**4.4.1.1、AudioTrack创建**

audio\_open

wanted\_spec.callback = sdl\_audio\_callback

SDL\_AoutOpenAudio调用SDL\_AoutOpenAudio aout→open\_audio

Android设备的实现：ijksdl\_aout\_android\_audiotrack.c的aout\_open\_audio。

aout\_open\_audio\_n

SDL\_Android\_AudioTrack\_new\_from\_sdl\_spec

SDL\_Android\_AudioTrack\_new\_from\_spec

配置audiotrack创建前所需要的参数，创建SDL\_Android\_AudioTrack \*atrack。

J4AC\_AudioTrack\_\_getMinBufferSize

J4AC\_AudioTrack\_\_AudioTrack\_\_asGlobalRef\_\_catchAll创建一个java AudioTrack对象并赋值在atrack->thiz中。

J4AC\_AudioTrack\_\_setStereoVolume\_\_catchAll初始化音量

上面创建了一个SDL\_Android\_AudioTrack对象，赋值给SDL\_Aout\_Opaque，封装java AudioTrack操作。

SDL\_Android\_AudioTrack\_get\_min\_buffer\_size，malloc(opaque→buffer\_size);申请一块内存，存放需要写给AudioTrack的音频数据。

SDL\_Android\_AudioTrack\_getAudioSessionId得到audio sessionid

SDL\_AoutSetDefaultLatencySeconds {func\_set\_default\_latency\_seconds}，没有实现。

SDL\_CreateThreadEx(&opaque->\_audio\_tid, aout\_thread, aout, "ff\_aout\_android")启动aout\_thread线程，audio输出线程，转4.5分析。

decoder\_init、decoder\_start 启动audio\_thread线程，audio解码线程， 转4.6分析。

SDL\_AoutPauseAudio aout->pause\_audio暂停标志配置为0，

**4.4.2、stream\_component\_open video**

Ffmpeg操作：

avcodec\_alloc\_context3

stream\_component\_open

avcodec\_find\_decoder

avcodec\_open2

decoder\_init

**4.4.2.1、创建video decoder**

ffpipeline\_open\_video\_decoder

func\_open\_video\_decoder

区分不同的平台，android走ffpipeline\_android.c

if (ffp->mediacodec\_all\_videos || ffp->mediacodec\_avc || ffp->mediacodec\_hevc || ffp->mediacodec\_mpeg2) {

node = ffpipenode\_create\_video\_decoder\_from\_android\_mediacodec(ffp, pipeline, opaque->weak\_vout);

if (!node) {

ALOGE("config hw video decoder but create failed, platform unsupport or have some problem.\n");

}

} else {

ALOGD("config use sf video decoder.\n");

}

if (!node) {

node = ffpipenode\_create\_video\_decoder\_from\_ffplay(ffp);

}

根据mediacodec\_avc 等options选择是否走MediaCodec解码器，前面有讲，在app中设置mediacodec，最终会配置给ffp的mediacodec\_avc。

**4.4.2.2、软解decoder**

ffpipenode\_create\_video\_decoder\_from\_ffplay，实现在ffpipenode\_ffplay\_vdec.c中

创建一个IJKFF\_Pipenode \*node，pipenode对应一个解码器。

node->func\_destroy = func\_destroy;

node->func\_run\_sync = func\_run\_sync;

ffp->stat.vdec\_type = FFP\_PROPV\_DECODER\_AVCODEC;vdec类型为1，表示ffplay。

**4.4.2.3、硬解decoder**

ffpipenode\_create\_video\_decoder\_from\_android\_mediacodec

创建IJKFF\_Pipenode \*node，一个pipenode，解码器。

node->func\_destroy = func\_destroy;

node->func\_run\_sync = func\_run\_sync;

node->func\_flush = func\_flush;

A、recreate\_format\_l创建MediaFormat

J4AC\_MediaFormat\_\_createVideoFormat\_\_withCString\_\_asGlobalRef\_\_catchAll调用java MediaFormat的方法createVideoFormat创建一个format：jobject android\_media\_format

SDL\_AMediaFormat\_CreateInternal创建SDL\_AMediaFormat \*aformat

setup\_aformat

SDL\_AMediaFormat\_Opaque \*opaque = aformat->opaque;

opaque->android\_media\_format = android\_media\_format;

保存java对象在SDL\_AMediaFormat\_Opaque。

ffp\_get\_video\_rotate\_degrees设置视频旋转角度。并通知 ffp\_notify\_msg2(ffp, FFP\_MSG\_VIDEO\_ROTATION\_CHANGED, rotate\_degrees)。

B、ffpipeline\_select\_mediacodec\_l

mediacodec\_select\_callback

J4AC\_IjkMediaPlayer\_\_onSelectCodec\_\_withCString\_\_asCBuffer

回调java IjkMediaPlayer中的onSelectCodec，IjkMediaPlayer根据mimeType得到打分最高的omx插件。

C、ffpipeline\_get\_surface\_as\_global\_ref

从IJKFF\_Pipeline\_Opaque中获得setSurface时保存的java surface对象，并创建一个新的引用jobject jsurface，传给MediaCodec。

D、reconfigure\_codec\_l

create\_codec\_l

SDL\_AmediaCodecJava\_createByCodecName

J4AC\_MediaCodec\_\_createByCodecName\_\_withCString创建一个java MediaCodec android\_media\_codec，并新建全局引用global\_android\_media\_codec

SDL\_AmediaCodecJava\_init创建SDL\_AMediaCodec acodec

SDL\_AMediaCodec\_Opaque android\_media\_codec中保存java MediaCodec对象

acodec->opaque\_class = &g\_amediacodec\_class "AMediaCodecJava"

SDL\_AMediaCodec函数指针赋值：指向java mediacodec方法

acodec->func\_dequeueInputBuffer = SDL\_AMediaCodecJava\_dequeueInputBuffer;

acodec->func\_queueInputBuffer = SDL\_AMediaCodecJava\_queueInputBuffer;

SDL\_AmediaCodec\_configure\_surface func\_configure\_surface配置surface给MediaCodec

SDL\_AmediaCodec\_start func\_start

SDL\_VoutAndroid\_setAMediaCodec把新建的SDL\_AmediaCodec保存在vout→opaque→acodec中

**ffp->stat.vdec\_type = FFP\_PROPV\_DECODER\_MEDIACODEC; decoder类型：2**

启动线程"ff\_video\_dec"，video解码线程，转4.7分析。

**4.4.3、回到读取线程read\_thread**

if (!ffp->start\_on\_prepared)根据app setting，这里满足条件，触发pause

toggle\_pause(ffp, 1);

ffp\_notify\_msg1(ffp, FFP\_MSG\_PREPARED); prepare完成通知

ret = av\_read\_frame(ic, pkt);读packet

avio\_feof文件结束判断

packet\_queue\_put\_nullpacket文件结束时填充空数据包给各个流的packet queue

packet\_queue\_put读取出的packet按照codecpar->codec\_type放入各个流的packet queue。

**4.5、第四个线程ff\_aout\_android**

Log：04-26 11:31:35.931 6474 7403 I IJKMEDIA: SDL\_RunThread: [7403] ff\_aout\_android

aout\_thread、aout\_thread\_n

while (!opaque->abort\_request)

{

SDL\_Android\_AudioTrack\_play

SDL\_Android\_AudioTrack\_pause

audio\_cblk(userdata, buffer, copy\_size);

SDL\_Android\_AudioTrack\_flush

SDL\_Android\_AudioTrack\_write

}循环读取数据并write给AudioTrack

SDL\_Android\_AudioTrack\_free 停止播放，释放AudioTrack

**4.5.1、audio\_cblk**

从解码帧队列读取数据，sdl\_audio\_callback

audio\_decode\_frame

frame\_queue\_peek\_readable

frame\_queue\_next

类似vout读取数据，只是帧队列换成了sampq。

synchronize\_audio，如果以audio时间为准，则该函数不执行任何操作。

根据输出audio的format，重采样音频数据swr\_init、swr\_convert。

if (!ffp->first\_audio\_frame\_rendered) {

ffp->first\_audio\_frame\_rendered = 1;

ffp\_notify\_msg1(ffp, FFP\_MSG\_AUDIO\_RENDERING\_START);

}

音频第一帧输出通知。？？？？遗留问题？？？？audio\_cblk待详细分析。

**4.6、第五个线程ff\_audio\_dec**

Log：04-26 11:31:35.931 6474 7404 I IJKMEDIA: SDL\_RunThread: [7404] ff\_audio\_dec

audio\_thread

decoder\_decode\_frame

packet\_queue\_get\_or\_buffering从packet\_queue取packet数据包

avcodec\_decode\_audio4解码输出到frame

frame\_queue\_peek\_writable取出FrameQueue中window指向的Frame，把解码帧写入。

av\_frame\_move\_ref拷贝

frame\_queue\_push windex加1、size加1，FrameQueue中有新的Frame写入。

**4.7、第六个线程ff\_video\_dec**

Log：04-26 11:31:36.242 6474 7420 I IJKMEDIA: SDL\_RunThread: [7420] ff\_video\_dec

video\_thread ffpipenode\_run\_sync node->func\_run\_sync

**4.7.1、硬解 mediacodec**

if (!opaque->acodec) {

return ffp\_video\_thread(ffp);

}

没有创建mediacodec时，走ffmpeg软解流程。

SDL\_CreateThreadEx(&opaque->\_enqueue\_thread, enqueue\_thread\_func, node, "amediacodec\_input\_thread");

创建线程"amediacodec\_input\_thread"，MeidaCodec数据输入，转4.8分析。

while (!q->abort\_request)循环，等待abort命令。

drain\_output\_buffer

drain\_output\_buffer\_l

SDL\_AmediaCodecFake\_dequeueOutputBuffer

**4.7.1.1、amc\_fill\_frame**

frame->format = IJK\_AV\_PIX\_FMT\_\_ANDROID\_MEDIACODEC;代表mediacodec，后面创建overlay时用到。

SDL\_VoutAndroid\_obtainBufferProxy

**4.7.1.2、ffp\_queue\_picture**

queue\_picture，注意是ff\_ffplay.c中的queue\_picture

frame\_queue\_peek\_writable取出可写入的frame地址。

A、第一帧时：执行alloc\_picture

SDL\_VoutSetOverlayFormat 设置vout->overlay\_format，app setting传下来的数据ffp->overlay\_format

SDL\_Vout\_CreateOverlay

static SDL\_VoutOverlay \*func\_create\_overlay\_l(int width, int height, int frame\_format, SDL\_Vout \*vout)

{

switch (frame\_format) { 解码输出帧format，前面可知是mediacodec

case IJK\_AV\_PIX\_FMT\_\_ANDROID\_MEDIACODEC:

return SDL\_VoutAMediaCodec\_CreateOverlay(width, height, vout);

default:

return SDL\_VoutFFmpeg\_CreateOverlay(width, height, frame\_format, vout);

}

}

根据 PIX\_FMT选择不同的Overlay，这里走 SDL\_VoutAMediaCodec\_CreateOverlay

overlay->opaque\_class = &g\_vout\_overlay\_amediacodec\_class; "AndroidMediaCodecVoutOverlay"

overlay->format = SDL\_FCC\_\_AMC;很重要，后面显示时需要

overlay->pitches = opaque->pitches;

overlay->pixels = opaque->pixels;

overlay->func\_fill\_frame = func\_fill\_frame;

SDL\_VoutOverlay和SDL\_VoutOverlay\_Opaque指向同一个像素buffer，创建了一个mediacodec的SDL\_VoutOverlay。

B、SDL\_VoutFillFrameYUVOverlay

func\_fill\_frame填充SDL\_VoutOverlay：

opaque->acodec = SDL\_VoutAndroid\_peekAMediaCodec(opaque->vout);得到SDL\_AMediaCodec对象

opaque->buffer\_proxy = (SDL\_AMediaCodecBufferProxy \*)frame->opaque;代理

overlay->opaque\_class = &g\_vout\_overlay\_amediacodec\_class; "AndroidMediaCodecVoutOverlay"

overlay->format = SDL\_FCC\_\_AMC;

overlay->planes = 1;

overlay->pixels[0] = NULL;

overlay->pixels[1] = NULL;

overlay->pitches[0] = 0;

overlay->pitches[1] = 0;

overlay->is\_private = 1;

Overlay中像素数据为空。像素数据在MediaCodec内部渲染。外部只控制显示时间。

frame\_queue\_push刷新frame\_queue，完成解码数据的更新。

SDL\_VoutAndroid\_releaseBufferProxyP 待分析？这个代理具体作用时是什么？

SDL\_VoutOverlay\_Opaque buffer\_proxy = (SDL\_AMediaCodecBufferProxy \*)frame opaque; AVFrame中的opaque？

SDL\_AMediaCodecBufferProxy

{

int buffer\_id;

int buffer\_index;

int acodec\_serial;

int32\_t offset;

int32\_t size;

int64\_t presentationTimeUs;

uint32\_t flags;

} 包含这些数据成员。

**4.7.2、软解 ffmpeg**

实现ffpipenode\_ffplay\_vdec.c

ffp\_video\_thread

ffplay\_video\_thread

get\_video\_frame

decoder\_decode\_frame

packet\_queue\_get

avcodec\_decode\_video2

4.7.2.1、queue\_picture 使用ff\_ffplay.c中的queue\_picture函数

frame\_queue\_peek\_writable读取frame\_queue中windex指向的frame。

第一帧时执行：alloc\_picture

SDL\_VoutSetOverlayFormat(ffp->vout, ffp→overlay\_format);vout->overlay\_format设置为app setting中的值。

SDL\_Vout\_CreateOverlay时还有一个frame\_format，走ffmpeg软解时，没有设置，应该有一个默认值。走MediaCodec解码时，设置为frame->format = IJK\_AV\_PIX\_FMT\_\_ANDROID\_MEDIACODEC;根据这个frame->format决定创建的Overlay类型。

软解走SDL\_VoutFFmpeg\_CreateOverlay

overlay->opaque\_class = &g\_vout\_overlay\_ffmpeg\_class;"FFmpegVoutOverlay"

overlay->func\_fill\_frame = func\_fill\_frame

SDL\_VoutAMediaCodec\_CreateOverlay，则overlay->format = SDL\_FCC\_\_AMC;这个值会影响overlay显示流程

SDL\_VoutFFmpeg\_CreateOverlay，则overlay->format = overlay\_format;

SDL\_VoutSetOverlayFormat时把 ffp->overlay\_format赋值给了 vout overlay\_format，所以这里overlay->format是app setting下来的，如果是SDL\_FCC\_YV12。

则case SDL\_FCC\_YV12:

ff\_format = AV\_PIX\_FMT\_YUV420P;

在创建管理帧时opaque->managed\_frame，managed\_frame->format = ff\_format;这个帧的架构是yuv420p，后面swscale需要用到。

4.7.2.2、opaque->managed\_frame = opaque\_setup\_frame(opaque, ff\_format, buf\_width, buf\_height);创建managed\_frame

overlay\_fill(overlay, opaque->managed\_frame, opaque->planes);

overlay->pixels[i] = frame->data[i];

overlay->pitches[i] = frame->linesize[i];

managed\_frame指向 overlay中的pixels和pitches。

4.7.2.3、SDL\_VoutFillFrameYUVOverlay

AVFrame\* managed\_frame = opaque\_obtain\_managed\_frame\_buffer(opaque);获取管理帧

overlay\_fill(overlay, opaque->managed\_frame, opaque->planes);

overlay->pixels[i] = frame->data[i];

overlay->pitches[i] = frame→linesize[i];

管理帧与 overlay中像素buffer关联，指向同一个。

swscale\_dst\_pic.data[i] = overlay->pixels[i];

swscale\_dst\_pic.linesize[i] = overlay->pitches[i];

overlay与swscale\_dst\_pic关联，指向同一个

ijk\_image\_convert dst\_format

sws\_getCachedContext dst\_format

sws\_scale dst\_format

以上三步得到dst\_format的最终像素数据，把结果输出到swscale\_dst\_pic，即Overlay。overlay、managed\_frame、swscale\_dst\_pic指向同样的地址，所以最后的帧数据存在managed\_frame、frame\_buffer、overlay->pixels中。

frame\_queue\_push刷新frame\_queue，rindex和windex、size更新：队列指针位置更新。

av\_frame\_unref(frame)释放解码后的数据，while循环重新开始解码下一帧数据。

**4.8、第七个线程amediacodec\_input\_thread**

Log：04-26 11:31:36.252 6474 7422 I IJKMEDIA: SDL\_RunThread: [7422] amediacodec\_input\_thread

enqueue\_thread\_func

while (!q->abort\_request) {

ret = feed\_input\_buffer

}

**4.8.1、feed\_input\_buffer**

ffp\_packet\_queue\_get\_or\_buffering从packetQueue获取packet

SDL\_AmediaCodec\_dequeueInputBuffer

SDL\_AmediaCodec\_writeInputData

SDL\_AmediaCodec\_queueInputBuffer

这个线程是往mediacodec inputbuffer送数据的。

总结：启动了7个线程

1、启动线程"ff\_msg\_loop"，执行message\_loop，循环读取消息；

2、启动线程"ff\_read"，读取、分发packet。

3、启动线程"ff\_vout"，video输出

4、启动线程"ff\_aout\_android"，audio输出。

5、启动线程"ff\_audio\_dec" audio解码

6、启动线程"ff\_video\_dec" video解码

7、启动线程"amediacodec\_input\_thread" mediacodec数据读入线程

8、第一次输出视频解码帧时，主动调用SDL\_Vout\_CreateOverlay，pipeline中创建SDL\_Aout，创建VoutOverlay

**5、start**

ijkmp\_start

ijkmp\_start\_l

ffp\_notify\_msg1(mp->ffplayer, FFP\_REQ\_START);

message\_loop\_n

ijkmp\_get\_msg

msg\_queue\_get

case FFP\_REQ\_START:

ffp\_start\_l(mp->ffplayer);

执行toggle\_pause\_l(ffp, pause\_on)

static void toggle\_pause\_l(FFPlayer \*ffp, int pause\_on)

{

VideoState \*is = ffp->is;

is->pause\_req = pause\_on;

ffp->auto\_resume = !pause\_on;

stream\_update\_pause\_l(ffp);

is->step = 0;

}

stream\_toggle\_pause\_l

set\_clock

SDL\_AoutPauseAudio

aout->pause\_audio(aout, pause\_on);

SDL\_Aout\_Opaque \*opaque->pause\_on = pause\_on;

改变audio线程pause状态，VideoState pause状态、以及clock等。

总结：start只是把prepare中创建的线程从休眠中唤醒，真正的运行起来。

**6、软解与硬解流程差异**

软解ffplay和硬解MediaCodec相比，audio处理相同，video 的处理不同。

**6.1、相同：**

1、消息循环meg\_loop

2、ffpipeline\_open\_audio\_output，创建audio输出AudioTrack

3、"ff\_read"线程read\_thread，prepare及数据读取

4、ff\_aout\_android，audio输出线程

5、"ff\_audio\_dec" audio decoder线程，使用ffmpeg解码

**6.2、不同：**

1、video\_refresh\_thread "ff\_vout"线程,流程相同，但是displayOverlay不同，在执行vout->display\_overlay时，有区别：

软解使用ANativeWindow显示视频数据，硬解时通过控制MeidaCodec的输出缓冲帧实现音视频同步。

2、ffpipeline\_open\_video\_decoder

Video decoder不同，创建时根据option选择创建不同的PipelineNode，硬解创建mediacodec解码器、软解创建ffplay（FFMPeg）解码器。

3、video解码线程

视频解码器不同，视频解码启动后的流程也不同。