FFMPEG 解码

目录

[一 解码指令 2](#_Toc7511727)

[1.1 软解 2](#_Toc7511728)

[1.2 硬解 2](#_Toc7511729)

[1.3 视频采集 3](#_Toc7511730)

[1.3.1 摄像头采集 3](#_Toc7511731)

[1.3.2 屏幕录制 3](#_Toc7511732)

[二 解码结构 4](#_Toc7511733)

[2.1 解码API 4](#_Toc7511734)

[2.2 读取流API 7](#_Toc7511735)

[2.3 数据结构 7](#_Toc7511736)

[2.4 解码流程 8](#_Toc7511737)

[2.4.1 读取流 9](#_Toc7511738)

[2.4.2 原始数据 9](#_Toc7511739)

[2.4.3 软解码流程 10](#_Toc7511740)

[2.4.4 硬解码流程 11](#_Toc7511741)

[三 解码实例 13](#_Toc7511742)

[3.1 数据结构 14](#_Toc7511743)

[3.2 接口 14](#_Toc7511744)

[3.3 实例 16](#_Toc7511745)

# 一 解码指令

## 1.1 软解

* yuv420P

ffmpeg -i 0001.mp4 -s 1920x1080 -pix\_fmt yuv420p -y syuv420p.yuv

参数说明：

s: 分辨率（w\*h）

pix\_fmt: 像素格式yuv420p

* nv12

ffmpeg -i 0001.mp4 -s 1920x1080 -pix\_fmt nv12 -y snv12.yuv

参数说明：

pix\_fmt: 像素格式nv12

## 1.2 硬解

* yuv420P

ffmpeg -hwaccel dxva2 -i 0001.mp4 -s 1920x1080 -pix\_fmt yuv420p -y hyuv420p.yuv

参数说明：

Hwaccel: 对于h264支持DXVA2/NVDEC等

对于Intel，可以采用h264\_qsv解码：

ffmpeg -i 0001.mp4 -vcodec h264\_qsv -pix\_fmt yuv420p -y hnv12.yuv

* nv12

ffmpeg -hwaccel dxva2 -i 0001.mp4 -s 1920x1080 -pix\_fmt nv12 -y hnv12.yuv

对于Intel，可以采用h264\_qsv解码：

ffmpeg -i 0001.mp4 -vcodec h264\_qsv -y hnv12.yuv

## 1.3 视频采集

### 1.3.1 摄像头采集

* 采集设备列表

ffmpeg -list\_devices true -f dshow -i dummy

* 采集摄像头

ffmpeg -f dshow -i video="Integrated Camera" -vcodec libx264 –y cap.mp4

ffmpeg -f dshow -i video="Integrated Camera" -vcodec libx264 –s 640x480 -preset:v fast -tune:v zerolatency -f h264 udp://233.233.233.223:66668

### 1.3.2 屏幕录制

在Windows平台下，有两种录屏方式，一种使用dshow录屏但需注册screen-capture-recorder；一种是gdigrab.

* Dshow:

ffmpeg -f dshow -i video="screen-capture-recorder" -r 25 -vcodec libx264 -preset:v fast -tune:v zerolatency dshow.mp4

* gdigrab

gdigrab是基于GDI的抓屏设备，用于抓取屏幕的区域。通过设定不同的输入情况，支持两种方式的屏幕抓取：

（1）“desktop”：全屏抓取。

（2）“title={窗口名称}”：抓取屏幕中某一个窗口。

|  |  |
| --- | --- |
| 参数 | 说明 |
| framerate | 视频帧率 |
| Video\_size | 视频大小 |
| Offset\_x | X坐标 |
| Offset\_y | Y坐标 |

ffmpeg -f gdigrab -i desktop out.mp4

ffmpeg -f gdigrab -framerate 25 -offset\_x 10 -offset\_y 20 -video\_size 640x480 -i desktop out.mp4

ffmpeg -re -f gdigrab -i desktop -vcodec libx264 -s 1920\*1080 -r 60 -g 3000 -pix\_fmt yuv420p -tune:v zerolatency -f rtp -y rtp://127.0.0.1:56782> rtp.sdp

# 二 解码结构

## 2.1 解码API

* avcodec\_find\_decoder\_by\_name

AVCodec \*avcodec\_find\_decoder\_by\_name(const char \*name);

功能：通过解码器名查找AVCodec；

参数：

Name: 解码器名，比如h264/h264\_qsv

AVCodec \*avcodec\_find\_decoder(enum AVCodecID id);

* 比如h264查找ff\_h264\_decoder：

AVCodec ff\_h264\_decoder = {

.name = "h264",

.long\_name = NULL\_IF\_CONFIG\_SMALL("H.264/AVC/MPEG-4 AVC/MPEG-4 part 10"),

.type = AVMEDIA\_TYPE\_VIDEO,

.id = AV\_CODEC\_ID\_H264,

.priv\_data\_size = sizeof(H264Context),

.init = h264\_decode\_init,

.close = h264\_decode\_end,

.decode = h264\_decode\_frame,

.capabilities = /\*AV\_CODEC\_CAP\_DRAW\_HORIZ\_BAND |\*/ AV\_CODEC\_CAP\_DR1 |

AV\_CODEC\_CAP\_DELAY | AV\_CODEC\_CAP\_SLICE\_THREADS |

AV\_CODEC\_CAP\_FRAME\_THREADS,

.hw\_configs = (const AVCodecHWConfigInternal\*[]) {

#if CONFIG\_H264\_DXVA2\_HWACCEL

HWACCEL\_DXVA2(h264),

#endif

#if CONFIG\_H264\_D3D11VA\_HWACCEL

HWACCEL\_D3D11VA(h264),

#endif

#if CONFIG\_H264\_D3D11VA2\_HWACCEL

HWACCEL\_D3D11VA2(h264),

#endif

#if CONFIG\_H264\_NVDEC\_HWACCEL

HWACCEL\_NVDEC(h264),

#endif

#if CONFIG\_H264\_VAAPI\_HWACCEL

HWACCEL\_VAAPI(h264),

#endif

#if CONFIG\_H264\_VDPAU\_HWACCEL

HWACCEL\_VDPAU(h264),

#endif

#if CONFIG\_H264\_VIDEOTOOLBOX\_HWACCEL

HWACCEL\_VIDEOTOOLBOX(h264),

#endif

NULL

},

.caps\_internal = FF\_CODEC\_CAP\_INIT\_THREADSAFE|FF\_CODEC\_CAP\_EXPORTS\_CROPPING,

.flush = flush\_dpb,

.init\_thread\_copy = ONLY\_IF\_THREADS\_ENABLED(decode\_init\_thread\_copy),

.update\_thread\_context= ONLY\_IF\_THREADS\_ENABLED(ff\_h264\_update\_thread\_context),

.profiles = NULL\_IF\_CONFIG\_SMALL(ff\_h264\_profiles),

.priv\_class = &h264\_class,

};

* 比如h264\_qsv查找ff\_h264\_qsv\_decoder：

AVCodec ff\_h264\_qsv\_decoder = {

.name = "h264\_qsv",

.long\_name = NULL\_IF\_CONFIG\_SMALL("H.264 / AVC / MPEG-4 AVC / MPEG-4 part 10 (Intel Quick Sync Video acceleration)"),

.priv\_data\_size = sizeof(QSVH2645Context),

.type = AVMEDIA\_TYPE\_VIDEO,

.id = AV\_CODEC\_ID\_H264,

.init = qsv\_decode\_init,

.decode = qsv\_decode\_frame,

.flush = qsv\_decode\_flush,

.close = qsv\_decode\_close,

.capabilities = AV\_CODEC\_CAP\_DELAY | AV\_CODEC\_CAP\_DR1 | AV\_CODEC\_CAP\_AVOID\_PROBING | AV\_CODEC\_CAP\_HYBRID,

.priv\_class = &class,

.pix\_fmts = (const enum AVPixelFormat[]){ AV\_PIX\_FMT\_NV12,

AV\_PIX\_FMT\_P010,

AV\_PIX\_FMT\_QSV,

AV\_PIX\_FMT\_NONE },

.hw\_configs = ff\_qsv\_hw\_configs,

.bsfs = "h264\_mp4toannexb",

.wrapper\_name = "qsv",

};

* avcodec\_alloc\_context3

AVCodecContext \*avcodec\_alloc\_context3(const AVCodec \*codec)

功能：创建解码结构AVCodecContext

参数：

Codec: 具体编解码Codec

* avcodec\_open2

int avcodec\_open2(AVCodecContext \*avctx, const AVCodec \*codec, AVDictionary \*\*options);

功能：打开AVCodecContext对应的解码器

参数：

Avctx：avcodec\_alloc\_context3分配

Codec：具体编解码Codec

Options：编解码属性

* avcodec\_send\_packet

int avcodec\_send\_packet(AVCodecContext \*avctx, const AVPacket \*avpkt);

功能：发送音视频数据给解码器

返回值：

= 0 ： 表示成功

< 0 ：除了下面两种错误状态，其余表示失败。

AVERROR(EAGAIN)：

表示当前状态下输入的packet未被接收，需要通过avcodec\_receive\_frame输出一个或多个的frame后才能重新输入当前packet。

AVERROR\_EOF:

当输入为NULL时才会触发该状态，用于通知解码器输入已结束。

* avcodec\_receive\_frame

int avcodec\_receive\_frame(AVCodecContext \*avctx, AVFrame \*frame);

功能：从解码器读取非压缩数据

返回值：

= 0 ： 表示成功

< 0 ： 除了下面两种错误状态，其余表示失败。

AVERROR(EAGAIN)：

表示当前状态下没有输出，还需继续向解码器发送数据；

AVERROR\_EOF:

表示没有数据输出，解码结束。

* avcodec\_get\_hw\_config

const AVCodecHWConfig \*avcodec\_get\_hw\_config(const AVCodec \*codec, int index);

功能：从AVCodec中获取对应硬配置

* av\_hwdevice\_ctx\_create

int av\_hwdevice\_ctx\_create(AVBufferRef \*\*device\_ctx, enum AVHWDeviceType type,

const char \*device, AVDictionary \*opts, int flags);

功能：通过AVHWDeviceType创建对应的AVHWDeviceContext

* av\_hwframe\_transfer\_data

int av\_hwframe\_transfer\_data(AVFrame \*dst, const AVFrame \*src, int flags);

功能：对于解码，从显存上提取数据为系统内存的AVFrame数据

## 2.2 读取流API

* avformat\_open\_input

int avformat\_open\_input(AVFormatContext \*\*ps, const char \*filename,

AVInputFormat \*fmt, AVDictionary \*\*options)

功能：

依据文件名后缀或文件内容探测AVInputFormt

参数：

AVFormatContext：如果为空，会在内部通过avformat\_alloc\_context分配；

Filename： 本地文件/流媒体URL/NULL。

AVInputFormat：av\_guess\_format函数通过格式名获取

AVDictionary ： 一般为空

* avformat\_find\_stream\_info

int avformat\_find\_stream\_info(AVFormatContext \*ic, AVDictionary \*\*options)

功能：

主要给每个媒体流（音频/视频）的AVStream结构体赋值（通过AVInputFormat的read\_packet函数读取一段数据进行解封装或解码）。

## 2.3 数据结构

硬解码的数据结构

* 此处通过专库进行解码：



注意：

在acodec\_open2打开解码器前，需要通过AVCodecContext的回调函数get\_format设置AVHWFramesContext，并将其赋给AVCodecContext成员hw\_frames\_ctx。

比如Intel芯片的h264\_qsv解码器,通过调用libmfx进行解码

* 此处通过AVHWAccel进行解码:



注意：

在acodec\_open2打开解码器前，通过av\_hwdevice\_ctx\_create创建AVHWDeviceContext后，需将其赋给AVCodecContext成员hw\_device\_ctx。

比如通过windows自带的DXVA2解码

## 2.4 解码流程

首先，介绍数据获取的几种方法（文件/内存/设备）。

### 2.4.1 读取流

s

* 从文件读取数据：

上图就为从文件/网络流读取数据流程。

* 从内存读取数据：

在avformat\_open\_input函数调用前，需通过avio\_alloc\_context分配AVIOContext（赋值给AVFormatContext的成员pb），并设置回调函数read\_packet及参数opaque(存储数据的地址)。

* 从设备读取数据

在avformat\_open\_input函数调用前，通过av\_find\_input\_format获取注册的AVInputFormat结构（比如：ff\_gdigrab\_demuxer），然后将其传给avformat\_open\_input函数.

注意：

使用前，需调用avdevice\_register\_all进行设备注册（比如windows系统录屏：gdigrab-ff\_gdigrab\_demuxer）

### 2.4.2 原始数据



注意：

此处是原始(h264)数据，不是封装（MP4/flv）的数据.

av\_parser\_init通过codec\_id得到AVCodecParserContext(比如h264: ff\_h264\_parser)

AVCodecParser ff\_h264\_parser = {

.codec\_ids = { AV\_CODEC\_ID\_H264 },

.priv\_data\_size = sizeof(H264ParseContext),

.parser\_init = init,

.parser\_parse = h264\_parse,

.parser\_close = h264\_close,

.split = h264\_split,

};

### 2.4.3 软解码流程



### 2.4.4 硬解码流程

* 专用解码库



上述虚线框内，为硬解码关键点。

* AVHWAccel加速器



# 三 解码实例

## 3.1 数据结构

数据结构关系图：



主要分为两部分：

一部分：解析数据CFFVideoDataObj；

一部分：解码CFFVideoDecoderObj

* CFFVideoDataObj类主要实现：

文件/网络流数据解封装；

原始数据解封装；

设备数据解封装；

* CFFVideoDecoderObj类主要实现：

软解码；

设备数据解码；

专有芯片硬解码

解码加速器解码。

## 3.2 接口

* 解码句柄

void\* FFVideoDec\_CreateInstance()

功能：

返回解码句柄

* 销毁解码句柄

s16 FFVideoDec\_DestoryInstance(void\* pHandle)

功能：

销毁解码句柄释放资源

参数：

解码器句柄

* 初始化解码器

s16 FFVideoDec\_InitDecoder(void\* pHandle, const TFFVideoDecoderParam \*ptVidDecParam)

功能：

利用参数初始化解码器以及打开解码器实例。

参数：

ptVidDecParam：解码参数（文件名，像素格式等）；

返回值：

= 0 ： 表示成功

< 0 ：表示初始化失败

* 销毁解码器

void FFVideoDec\_CloseDecoder(void\* pHandle);

功能：

释放解码器的相关资源

* 解码一帧视频

s16 FFVideoDec\_DecodeFrame(void\* pHandle, u8 \*pbyInData, u32 &dwInLen, u8 \*pbyOutData, u32 &dwOutLen);

功能：

将输入的原始数据或文件流进行视频解码。

参数：

pbyInData:传入原始数据，否则为NULL（读取文件/设备时）

dwInLen: 原始数据字节数

pbyOutData：存储一帧解码视频；

dwOutLen：一帧视频大小，为0时，表示没有产生一帧数据。

返回值：

=2 ： 表示有解码数据；

=1 ： 表示解码结束；

= 0 ： 表示没有解码数据；

< 0 ： 表示解码失败。

* 刷新解码器

s16 FFVideoDec\_FlushFrame(void\* pHandle, u8 \*pbyOutData, u32 &dwOutLen);

功能：

取出解码缓存视频帧。

参数：

pbyOutData：存储一帧解码视频；

dwOutLen：一帧视频大小，为0时，表示没有产生一帧数据。

返回值：

=2 ： 表示有解码数据；

=1 ： 表示解码结束；

= 0 ： 表示没有解码数据；

< 0 ： 表示解码失败。

## 3.3 实例

解码命令格式：

SWDecFFmpeg.exe <dataType> <devType> <width> <height> <codec> <input file>

<dataType> : 0 : 从文件读取数据

1 : 直接传入原始数据，比如h264数据

2 : 从设备读取数据，比如录屏

<devType> :0 : 软设备

1 : dxva2加速器

2 : qsv解码器

<width> : 输出图像宽

<height> : 输出图像高

<codec> ： 解码器名称，比如h264/h264\_qsv

<input file> : 输入文件名

* 软解码

SWDecFFmpeg.exe 0 0 640 480 h264 E:\code\project\ffmpegSdk\video\0001.mp4

SWDecFFmpeg.exe 1 0 640 480 h264

E:\code\project\ffmpegSdk\video\decRaw.h264

* 加速器

SWDecFFmpeg.exe 0 1 640 480 h264 E:\code\project\ffmpegSdk\video\0001.mp4

SWDecFFmpeg.exe 1 1 640 480 h264

E:\code\project\ffmpegSdk\video\decRaw.h264

* 专有硬解码

SWDecFFmpeg.exe 0 2 640 480 h264\_qsv

E:\code\project\ffmpegSdk\video\0001.mp4

SWDecFFmpeg.exe 1 2 640 480 h264\_qsv

E:\code\project\ffmpegSdk\video\decRaw.h264