FFMPEG 编码

目录

[一 基本组成 3](#_Toc7510970)

[二 编码指令 3](#_Toc7510971)

[2.1 视频编码 3](#_Toc7510972)

[2.1.1 X264参数 3](#_Toc7510973)

[2.1.2 编码 4](#_Toc7510974)

[2.1.3 转码 5](#_Toc7510975)

[2.1.4 提取视频 5](#_Toc7510976)

[2.1.5 设置preset 5](#_Toc7510977)

[2.1.6 设置tune 6](#_Toc7510978)

[2.1.7 设置profile 7](#_Toc7510979)

[2.1.8 X264内部参数设置 7](#_Toc7510980)

[2.1.9 码率控制 7](#_Toc7510981)

[2.1.10 Intel硬编 8](#_Toc7510982)

[三 数据结构 9](#_Toc7510983)

[3.1 像素格式 9](#_Toc7510984)

[3.2 层次结构 10](#_Toc7510985)

[3.3 通用结构 11](#_Toc7510986)

[3.3.1 AVFrame 12](#_Toc7510987)

[3.3.2 AVPacket 12](#_Toc7510988)

[3.3.3 AVFormatContext 12](#_Toc7510989)

[3.3.4 AVIOContext 13](#_Toc7510990)

[3.3.5 AVStream 13](#_Toc7510991)

[3.3.6 AVCodecContext 14](#_Toc7510992)

[3.3.7 AVCodec 14](#_Toc7510993)

[3.4 硬编结构 15](#_Toc7510994)

[3.4.1硬编码结构图 15](#_Toc7510995)

[3.4.2 AVHWDeviceContext 15](#_Toc7510996)

[3.4.3 AVHWFramesContext 15](#_Toc7510997)

[四 编码API 16](#_Toc7510998)

[4.1 封装API 16](#_Toc7510999)

[4.2 编码API 17](#_Toc7511000)

[4.3 硬编API 18](#_Toc7511001)

[五 编码流程 21](#_Toc7511002)

[5.1 软编码流程 21](#_Toc7511003)

[5.2 硬编码流程 22](#_Toc7511004)

[5.3 编码设置 24](#_Toc7511005)

[5.3.1 常用参数设置 24](#_Toc7511006)

[5.3.2 profile/preset设置 25](#_Toc7511007)

[5.3.3 编码延时 25](#_Toc7511008)

[5.3.4 并行编码 26](#_Toc7511009)

[5.3.5 关键帧设置 27](#_Toc7511010)

[六 编码实例 27](#_Toc7511011)

[6.1 数据结构 27](#_Toc7511012)

[6.2 接口 28](#_Toc7511013)

[6.3 实例 29](#_Toc7511014)

# 一 基本组成

FFmpeg框架基本由AVFormat，AVCodec，AVutil，AVDevice，Swscale，Swresample，AVFilter等组成。

* AVFormat

AVFormat是FFmpeg的封装模块，实现了绝大多数多媒体封装格式，包括封装及解封装，比如TS，MP4，FLV等媒体封装格式，以及HLS，RTMP，RTP等网络协议封装格式，此处file也使当着协议处理的。

* AVCodec

AVCodec是FFmpeg的编解码模块，支持自带的MPEG4/aac等编解码方式外，还支持第三方编解码库，比如支持AVC编码（需libx264库），HEVC编码（需libx265库），AAC编码（需libfdk-aac库-音频编码质量最优）。

* AVDevice

AVDevice是FFmpeg与多媒体设备交互的模块，使用这个库可以读取电脑多媒体设备的数据，或者输出数据到指定的多媒体设备上。比如对于windows系统，采用gdigrab进行录屏，或采用dshow采集摄像头，Linux系统采用V4l2采集摄像头。

* Swscale

Swscale是FFmpeg的视频图像缩放模块，可以进行图像的缩放（1080P—720P）及像素格式的转换（YUV420P---RGB）。

* Swresample

Swresample是FFmpeg的音频重采样模块，对音频进行重采样（48K—44.1k）。

* AVFilter

AVFilter是FFmpeg的滤镜模块，提供了对音视频的滤镜处理框架，比如对视频加水印，宿放以及画中画，对音频混音等滤镜。

# 二 编码指令

## 2.1 视频编码

### 2.1.1 X264参数

可以通过ffmpeg –h encoder=libx264查看具体的参数及介绍。

ffmpeg –h encoder=libx264输出如下：

Encoder libx264 [libx264 H.264 / AVC / MPEG-4 AVC / MPEG-4 part 10]:

General capabilities: delay threads

Threading capabilities: auto

Supported pixel formats: yuv420p yuvj420p yuv422p yuvj422p yuv444p yuvj444p nv12 nv16 nv21 yuv420p10le yuv422p10le yuv444p10le nv20le

libx264 AVOptions:

-preset <s> Set the encoding preset (cf. x264 --fullhelp) (default "medium")

-tune <s> Tune the encoding params (cf. x264 --fullhelp)

-profile <s> Set profile restrictions (cf. x264 --fullhelp)

-x264opts <s> x264 options

-crf <f> Select the quality for constant quality mode (default -1)

-qp <i> Constant quantization parameter rate control method (default -1)

-rc-lookahead <i> Number of frames to look ahead for frametype and ratecontrol (default -1)

-8x8dct <b> High profile 8x8 transform. (default auto)

-mbtree <b> Use macroblock tree ratecontrol. (default auto)

-deblock <s> Loop filter parameters, in <alpha:beta> form.

-nal-hrd <i> Signal HRD information (requires vbv-bufsize; cbr not allowed in .mp4) (default -1)

-forced-idr <b> If forcing keyframes, force them as IDR frames. (default false)

-sc\_threshold <i> Scene change threshold (default -1)

-x264-params <s> Override the x264 configuration using a :-separated list of key=value parameters

下面给出常用参数介绍：

preset： 编码器预设参数，默认值为medium

tune： 调优编码参数，比如zerolatency

profile： 编码档次的设置，比如base,main,high

level：编码层级的设置，针对相关编码参数，比如分辨率

crf：恒定质量模式

qp：恒定量化参数模式

forced-IDR：强制设置关键帧为IDR

x264-params：设置X264专有参数，比如sliced-threads，annexb

sc\_threshold：场景切换（从一个画面突然变到另外一个画面时）阈值决定是否插入关键帧，如为0就不插入关键帧（HLS）。

nal-hrd：HRD（Hypothetical Reference Decoder 假设参考解码器），有两种可选模式：CBR和VBR（需同时设置FFmpeg参数b:v， maxrate， minrate及bufsize）。

### 2.1.2 编码

ffmpeg -f rawvideo -s:v 640x480 -pix\_fmt yuv420p -i 0001\_640\_480\_I420.yuv -vcodec libx264 -s 1920x1080 -y swtest.mp4

参数说明：

pix\_fmt：像素格式

i：输入文件，文件名或流地址URL

vcodec/c:v ：视频编码方式：copy/libx264

s: 分辨率

y： 覆盖输出文件

### 2.1.3 转码

* 采用libx264编码

ffmpeg -i 0004.mp4 -vcodec libx264 -s 640\*360 -g 60 -r 30 -b:v 2000k -movflags faststart -pix\_fmt yuv420p -y sw.mp4

参数说明：

s： 帧的大小，分辨率（w\*h）

g：图像组大小（两个关键帧间的帧数）

r: 帧率fps

b:v 比特率

pix\_fmt 输出的像素格式

-movflags faststart 适合流播放，将moovbox置于mdatbox之前。

### 2.1.4 提取视频

ffmpeg -i 0004.mp4 -c:v copy -an -y output.h264

参数说明：

an: 取消音频

### 2.1.5 设置preset

设置预设参数，主要是为了均衡图像清晰度与转码速度，默认为medium。

* 参数说明：

可以采用x264 --fullhelp查看其详细说明。

--preset <s> Use a preset to select encoding settings [medium]

取值如下：

- ultrafast:

--no-8x8dct --aq-mode 0 --b-adapt 0 --bframes 0 --no-cabac --no-deblock

--no-mbtree --me dia --no-mixed-refs --partitions none --rc-lookahead 0 --ref 1

--scenecut 0 --subme 0 --trellis 0 --no-weightb --weightp 0

- superfast:

--no-mbtree --me dia --no-mixed-refs --partitions i8x8,i4x4 --rc-lookahead 0

--ref 1 --subme 1 --trellis 0 --weightp 1

- veryfast:

--no-mixed-refs --rc-lookahead 10 --ref 1 --subme 2 --trellis 0 --weightp 1

- faster:

--no-mixed-refs --rc-lookahead 20 --ref 2 --subme 4 --weightp 1

- fast:

--rc-lookahead 30 --ref 2 --subme 6 --weightp 1

- medium:

--rc-lookahead 40 --ref 3 –bframes 3

- slow:

--direct auto --rc-lookahead 50 --ref 5 --subme 8 --trellis 2

- slower:

--b-adapt 2 --direct auto --me umh --partitions all --rc-lookahead 60

--ref 8 --subme 9 --trellis 2

- veryslow:

--b-adapt 2 --bframes 8 --direct auto --me umh --merange 24 --partitions all

--ref 16 --subme 10 --trellis 2 --rc-lookahead 60

- placebo:

--bframes 16 --b-adapt 2 --direct auto --slow-firstpass --no-fast-pskip

--me tesa --merange 24 --partitions all --rc-lookahead 60 --ref 16 --subme 11

--trellis 2

* 设置preset为ultrafast

ffmpeg -i swVid.mp4 -vcodec libx264 -preset ultrafast -b:v 2000k swVidUltrafast.mp4

输出：

frame= 59 fps=0.0 q=-1.0 Lsize= 625kB time=00:00:02.32 bitrate=2208.5kbits/s speed=23.7x

编码速度为23.7，但画质差。

### 2.1.6 设置tune

会提升编码效率，降低编码导致的延时，特别是对于直播推流的场景。

* 参数

可以采用x264 --fullhelp查看其详细说明。

--tune <string> Tune the settings for a particular type of source or situation

- zerolatency:

--bframes 0 --force-cfr --no-mbtree --sync-lookahead 0 --sliced-threads

--rc-lookahead 0

* 编码命令：

ffmpeg -i swVid.mp4 -vcodec libx264 -tune zerolatency -b:v 2000k sw.mp4

### 2.1.7 设置profile

X264的常用profile:baseline，main，high

BP: 支持： I，P slice；多参考帧；CAVLC；8bit采样；

不支持： B slice；CABAC；8x8与4x4转换

MP: 支持： B slice；CABAC；

不支持： 8x8与4x4转换

HP: 支持： 8x8与4x4转换

ffmpeg -i swVid.mp4 -vcodec libx264 -profile:v high -y sw.mp4

### 2.1.8 X264内部参数设置

* 设置B帧数

ffmpeg -i swVid.mp4 -vcodec libx264 -x264-params “bframes=5” -y sw.mp4

* 采用annexb方式

ffmpeg -i swVid.mp4 -vcodec libx264 -x264-params “annexb=1” -y sw.mp4

### 2.1.9 码率控制

码率控制：ABR, CQP, CRF. 缺省方法是CRF。这三种方式的优先级是ABR > CQP > CRF

CQP不推荐使用，CRF关注视频质量，而用于文件大小或码率不太紧要的场景，ABR用于目标码率受限的场景。

* CRF命令：

ffmpeg -i swVid.mp4 -vcodec libx264 -crf 22 -y sw.mp4

* ABR命令：

ffmpeg -i swVid.mp4 -vcodec libx264 -b:v 2000k -y sw.mp4

与ABR向对应的码控有CBR,VBR

CBR：保持在恒定目标比特率，有利于流式播放。

VBR：为得到稳定的视频质量，为简单场景分配较大的QP，为复杂的场景分配较小的QP，但输出码流大小不可控。

* CBR命令：

ffmpeg -i swVid.mp4 -vcodec libx264 -b:v 2000k -x264-params "vbv-maxrate=2000:vbv-bufsize=2000" -y sw.mp4

ffmpeg -i swVid.mp4 -vcodec libx264 -b:v 2000k –qmin 20 –qmax 40 -maxrate 2000k -minrate 2000k -bufsize 2000k -y sw.mp4

参数说明：

Qmin:最小视频量化尺度[0-51]

Qmax：最大视频量化尺度[0-51]

Maxrate：最大视频码率

Minrate：最小视频码率

Bufsize：码率控制缓冲区

### 2.1.10 Intel硬编

* 硬编码支持的平台信息：

|  |  |  |
| --- | --- | --- |
| Platform Name | Graphics | Adds support for... |
| Ironlake | gen5 | MPEG-2, H.264 decode. |
| Sandy Bridge | gen6 | VC-1 decode; H.264 encode. |
| Ivy Bridge | gen7 | JPEG decode; MPEG-2 encode. |
| Bay Trail | gen7 | - |
| Haswell | gen7.5 | - |
| Broadwell | gen8 | VP8 decode. |
| Braswell | gen8 | H.265 decode; JPEG, VP8 encode. |
| Skylake | gen9 | H.265 encode. |
| Apollo Lake | gen9 | VP9, H.265 Main10 decode. |
| Kaby Lake | gen9.5 | VP9 profile 2 decode; VP9, H.265 Main10 encode. |
| Coffee Lake | gen9.5 | - |
| Gemini Lake | gen9.5 | - |
| Cannonlake | gen10 | - |

参考：<http://trac.ffmpeg.org/wiki/Hardware/QuickSync>

* 检查QSV编解码列表

ffmpeg -encoders | grep qsv

ffmpeg -decoders | grep qsv

* h264\_qsv支持的属性

ffmpeg -h encoder=h264\_qsv

* ffmpeg支持的加速器

命令：ffmpeg -hwaccels

输出：

Hardware acceleration methods:

cuda

dxva2

qsv

d3d11va

qsv

cuvid

* H264\_qsv

ffmpeg -s 640x480 -pix\_fmt yuv420p -i 0001\_640\_480\_I420.yuv -vcodec h264\_qsv -r 60 -s 1920x1080 -y r60\_1080p.h264

* Hevc\_qsv

ffmpeg -s 640x480 -pix\_fmt yuv420p -i 0001\_640\_480\_I420.yuv -vcodec hevc\_qsv -r 60 -s 1920x1080 -y r60\_1080p.h265

# 三 数据结构

## 3.1 像素格式

YUV： “Y”表示亮度（Luma），也就是灰度值；而“U”和“V” 表示色度（Chroma），描述色彩及饱和度，用于指定像素的颜色。

YUV格式有两大类：planar和packed。

Planar：先存储所有像素点Y，然后存储所有像素点U，最后所有像素点V。

Packed：每个像素点的Y,U,V连续交错存储。

YUV存储格式与采样的方式相关，主流的采样方式有三种，YUV4:4:4，YUV4:2:2，YUV4:2:0

YUV 4:4:4采样，每一个Y对应一组UV分量。

YUV 4:2:2采样，每两个Y共用一组UV分量。

YUV 4:2:0采样，每四个Y共用一组UV分量。

常用像素格式：

YUV420P：Y，U，V三个分量都是平面格式，分为I420和YV12。

I420: YYYYYYYY UU VV

YV12： YYYYYYYY VV UU

YUV420： YUV YUV YUV连续交错存储

YUV420SP：Y分量平面格式，UV打包格式, 分为NV12和NV21。

NV12： YYYYYYYY UVUV

NV21： YYYYYYYY VUVU

## 3.2 层次结构

FFmpeg编解码层次：协议层，封装层，编码层，像素层

编码层次如下图：



各层次对应的关键的结构体分成以下几类：

* 音视频编解码数据封装结构：

AVPacket：压缩后的数据（视频：H264，音频：aac）；

AVFrame：存储原始数据（即非压缩数据，比如视频：YUV，RGB；音频：PCM）。

* 协议层（http,rtsp,rtmp,file）

AVIOContext：实现输入输出数据带缓冲的读写操作；

URLContext： 管理协议操作对象及具体协议数据结构；

URLProtocol：协议操作对象，管理数据读写，每种协议对应一个URLProtocol结构。

* 封装层（flv ,mp4,ts）

AVFormatContext：主要存储音视频封装格式中包含的信息；

AVInputFormat： 存储输入音视频使用的封装格式；

AVOutputFormat：存储输出音视频使用的封装格式。

* 编码层（avc,hevc,aac,mp3）

AVStream：存储一个视频/音频流的相关数据，并对应一个AVCodecContext；

AVCodecContext：存储视频/音频流使用编解码方式的相关数据，每个AVCodecContext中对应一个AVCodec；

AVCodec：包含该视频/音频对应的编解码器，每种编解码器都对应一个AVCodec结构。

* 像素层（yuv , rgb）

视频：一般是存一帧；

音频：可能存好几帧。

## 通用结构

数据结构总关系图：



### 3.3.1 AVFrame

uint8\_t \*data[AV\_NUM\_DATA\_POINTERS]：音视频原始数据（视频：YUV/RGB，音频：PCM）

linesize[AV\_NUM\_DATA\_POINTERS]：data中“一行”数据的大小，一般大于等于图像的宽。

packed格式的数据（RGB24）：

Data[0]: RGB

Linesize[0]: 1280 \* 3

planar格式的数据（720P YUV420P）：

Data[0]: Y data[1]: U data[1]: v

Linesize[0]: 1280 Linesize[1]: 640 Linesize[2]: 640

int width, height：视频帧宽和高（1920x1080,1280x720）

int format：原始数据类型（YUV420P/RGB24）

int key\_frame：设置关键帧

enum AVPictureType pict\_type：帧类型（I,B,P）

AVRational sample\_aspect\_ratio：宽高比（16:9，4:3）

int64\_t pts：显示时间戳，编码时可以通过AVCodecContext->frame\_number设置。

### AVPacket

uint8\_t \*data：压缩编码数据。比如对于AVC，就是一个nal（pps或sps或一帧图像）.

size：data的大小

int64\_t pts：显示时间戳 (基于AVStream->time\_base，编码后通过av\_packet\_rescale\_ts从基于AVcodecContext->timebase转为基于AVStream->time\_base)

int64\_t dts：解码时间戳

stream\_index：标识该AVPacket所属的视频/音频流的索引。

flags：通过AV\_PKT\_FLAG\_KEY标识是否为关键帧。

### 3.3.3 AVFormatContext

struct AVInputFormat \*iformat：输入数据的封装格式

struct AVOutputFormat \*oformat：输出数据的封装格式

AVIOContext \*pb：输入数据的缓存

unsigned int nb\_streams：音视频流的个数

AVStream \*\*streams：音视频流

char \*url：输入输出文件名

int64\_t duration：时长（单位：微秒us）

int bit\_rate：比特率（单位：bps，kbps= bps/1000）

AVDictionary \*metadata：元数据（比如作者，版权，创建时间等）

### 3.3.4 AVIOContext



buffer：数据缓存开始位置

buffer\_size：数据缓存大小（默认为32768）

buf\_ptr：当前读取到的缓存位置

buf\_end：缓存结束的位置

opaque：URLContext结构体

Read\_packet:可以自定义读取内存（通过avio\_alloc\_context设置），或在avformat\_open\_input（format探测，获取协议操作对象）解析文件时通过Avformatcontext->io\_open调ffio\_open\_whitelist函数通过avio\_alloc\_context函数设置为io\_read\_packet（是对协议操作对象的读操作url\_read的封装）。

### 3.3.5 AVStream



int index： 在AVFormatContext中的stream索引

AVCodecContext \*codec：用AVCodecParameters代替

AVRational time\_base：帧时间戳的基本时间单位（以秒为单位），通过该值可以把PTS，DTS转化为真正的时间。

int64\_t start\_time：解码时流第一帧PTS时间

int64\_t duration：解码时流的时间长度

int64\_t nb\_frames：此流中已知的帧数或0。

AVDictionary \*metadata：元数据信息（标题/作者/创建时间）。

AVCodecParameters \*codecpar：此流相关的编解码参数，解封装时由libavformat在流创建时填充或在avformat\_find\_stream\_info（）赋值。封装时在avformat\_write\_header（）之前可以由avcodec\_parameters\_from\_context填充。

AVCodecParserContext \*parser：比如H264解析AVCodecContext->extradata中的SPS/PPS，以及组装及解析NALU单元。

### AVCodecContext

enum AVMediaType codec\_type：编解码器类型（视频/音频）

struct AVCodec  \*codec：编解码器AVCodec（H.264）

codec\_id：AV\_CODEC\_ID\_H264

int bit\_rate/rc\_max\_rate/rc\_min\_rate：比特率

uint8\_t \*extradata：附加信息（H264存储SPS，PPS等）

AVRational time\_base：1/帧率

AVRational framerate：帧率

int width, height：视频的宽和高

int qmin,qmax: 量化系数

int refs：参考帧的个数

int gop\_size： group of picture，一个序列

enum AVPixelFormat pix\_fmt：像素格式，比如AV\_PIX\_FMT\_YUV420P表示4:2:0的YUV图像

int profile：档次

int level：级

int sample\_rate: 音频采样率

int channels：音频声道数

enum AVSampleFormat sample\_fmt：音频采样格式

int frame\_size：音频帧中每个声道的采样数，aac-1024

int frame\_number：帧计数器。解码时从解码器返回的帧的总数，编码时到目前为止传递给编码器的帧的总数

uint64\_t channel\_layout：音频声道布局

### 3.3.7 AVCodec

const char \*name：编解码器的名字

enum AVMediaType type：指明类型(视频/音频/字幕)

enum AVCodecID id：比如AV\_CODEC\_ID\_H264

const enum AVPixelFormat \*pix\_fmts：视频支持的像素格式

const enum AVSampleFormat \*sample\_fmts：音频支持的采样格式

int priv\_data\_size：私有数据的大小（sizeof(X264Context)）

encode2：编码操作API

decode：解码操作API

const struct AVCodecHWConfigInternal \*\*hw\_configs：解码时硬件加速配置

## 3.4 硬编结构

### 3.4.1硬编码结构图



注意：

在acodec\_open2打开编码器前，需创建AVHWDeviceContext及AVHWFramesContext结构，并将AVHWFramesContext赋值给AVCodecContext的成员hw\_frames\_ctx。

### 3.4.2 AVHWDeviceContext

AVHWDeviceInternal \*internal;硬件设备内部数据结构（含有HWContextType）

enum AVHWDeviceType type;硬件设备类型，比如AV\_HWDEVICE\_TYPE\_QSV

void \*hwctx;内部硬件设备数据结构，比如AVQSVDeviceContext-mfxsession

### 3.4.3 AVHWFramesContext

AVHWFramesInternal \*internal; 硬件帧内部数据结构（含有HWContextType）

AVBufferRef \*device\_ref; 硬件设备结构引用

AVHWDeviceContext \*device\_ctx;关联的硬件设备结构

void \*hwctx; 比如AVQSVFramesContext

enum AVPixelFormat format;硬件加速像素AV\_PIX\_FMT\_QSV

enum AVPixelFormat sw\_format; 帧像素AV\_PIX\_FMT\_NV12

AVBufferPool \*pool;预分配帧的缓存池

# 四 编码API

## 4.1 封装API

* avformat\_alloc\_output\_context2

int avformat\_alloc\_output\_context2(AVFormatContext \*\*avctx, AVOutputFormat \*oformat, const char \*format, const char \*filename)

功能：

初始化一个用于输出的AVFormatContext结构体

参数：

avctx：成功后创建的AVFormatContext结构体。

oformat： 用于确定输出格式,如果为空，可以设定后两个参数（format或者filename）由FFmpeg探测输出格式。

format：指定输出格式的名称，比如：flv。

filename：指定输出文件的名称。

* avformat\_new\_stream

AVStream \*avformat\_new\_stream(AVFormatContext \*s, const AVCodec \*c)

功能：

创建音视频流的通道

* avio\_open

int avio\_open(AVIOContext \*\*s, const char \*filename, int flags)

功能：

打开输入输出文件

参数：

s：成功之后创建的AVIOContext结构体。

url：输入输出协议的地址。

flags：打开地址的方式，取值如下。

AVIO\_FLAG\_READ：只读。

AVIO\_FLAG\_WRITE：只写。

AVIO\_FLAG\_READ\_WRITE：读写。

## 4.2 编码API

* avcodec\_find\_encoder\_by\_name

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

通过编码器名查找AVCodec，比如libx264查找ff\_libx264\_encoder

AVCodec ff\_libx264\_encoder = {

.name = "libx264",

.long\_name = NULL\_IF\_CONFIG\_SMALL("libx264 H.264 / AVC / MPEG-4 AVC / MPEG-4 part 10"),

.type = AVMEDIA\_TYPE\_VIDEO,

.id = AV\_CODEC\_ID\_H264,

.priv\_data\_size = sizeof(X264Context),

.init = X264\_init,

.encode2 = X264\_frame,

.close = X264\_close,

.capabilities = AV\_CODEC\_CAP\_DELAY | AV\_CODEC\_CAP\_AUTO\_THREADS,

.priv\_class = &x264\_class,

.defaults = x264\_defaults,

.init\_static\_data = X264\_init\_static,

.caps\_internal = FF\_CODEC\_CAP\_INIT\_THREADSAFE |

FF\_CODEC\_CAP\_INIT\_CLEANUP,

.wrapper\_name = "libx264",

};

* avcodec\_alloc\_context3

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

创建及默认配置AVCodecContext

* avcodec\_open2

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

通过AVCodec初始化AVCodecContext

* avcodec\_send\_frame

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

发送音视频原始数据给编码器编码

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frame未被接收，需要通过avcodec\_receive\_packet输出一个或多个的packet后才能重新输入当前frame。

AVERROR\_EOF:

当输入为NULL时才会触发该状态，用于通知编码器输入已结束。

* avcodec\_receive\_packet

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

从编码器读取压缩数据

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，还需继续向编码器发送帧；

AVERROR\_EOF:

表示没有数据输出，编码结束。

## 4.3 硬编API

* av\_hwdevice\_ctx\_create

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

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

依据AVHWDeviceType （AV\_HWDEVICE\_TYPE\_QSV）创建AVHWDeviceContext（存储了结构HWContextType -- ff\_hwcontext\_type\_qsv），对于windows而言此处硬件加速器采用的是DXVA2

const HWContextType ff\_hwcontext\_type\_qsv = {

.type = AV\_HWDEVICE\_TYPE\_QSV,

.name = "QSV",

.device\_hwctx\_size = sizeof(AVQSVDeviceContext),

.device\_priv\_size = sizeof(QSVDeviceContext),

.frames\_hwctx\_size = sizeof(AVQSVFramesContext),

.frames\_priv\_size = sizeof(QSVFramesContext),

.device\_create = qsv\_device\_create,

.device\_derive = qsv\_device\_derive,

.device\_init = qsv\_device\_init,

.frames\_get\_constraints = qsv\_frames\_get\_constraints,

.frames\_init = qsv\_frames\_init,

.frames\_uninit = qsv\_frames\_uninit,

.frames\_get\_buffer = qsv\_get\_buffer,

.transfer\_get\_formats = qsv\_transfer\_get\_formats,

.transfer\_data\_to = qsv\_transfer\_data\_to,

.transfer\_data\_from = qsv\_transfer\_data\_from,

.map\_to = qsv\_map\_to,

.map\_from = qsv\_map\_from,

.frames\_derive\_to = qsv\_frames\_derive\_to,

.frames\_derive\_from = qsv\_frames\_derive\_from,

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

};

* avcodec\_find\_encoder\_by\_name

通过编码器名h264\_qsv查找AVCodec === ff\_h264\_qsv\_encoder

AVCodec ff\_h264\_qsv\_encoder = {

.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(QSVH264EncContext),

.type = AVMEDIA\_TYPE\_VIDEO,

.id = AV\_CODEC\_ID\_H264,

.init = qsv\_enc\_init,

.encode2 = qsv\_enc\_frame,

.close = qsv\_enc\_close,

.capabilities = AV\_CODEC\_CAP\_DELAY | AV\_CODEC\_CAP\_HYBRID,

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

AV\_PIX\_FMT\_P010,

AV\_PIX\_FMT\_QSV,

AV\_PIX\_FMT\_NONE },

.priv\_class = &class,

.defaults = qsv\_enc\_defaults,

.caps\_internal = FF\_CODEC\_CAP\_INIT\_CLEANUP,

.wrapper\_name = "qsv",

};

* av\_hwframe\_ctx\_alloc

AVBufferRef \*av\_hwframe\_ctx\_alloc(AVBufferRef \*device\_ctx);

对于给定的AVHWDeviceContext创建AVHWFramesContext。

* av\_hwframe\_ctx\_init

int av\_hwframe\_ctx\_init(AVBufferRef \*ref)

初始化AVHWFramesContext，对于qsv调用qsv\_frames\_init-分配显存surface，同时会分配同等数量（帧缓冲池）的frame的buffer（在编码时，通过av\_hwframe\_get\_buffer为AVFrame获取）

* av\_hwframe\_transfer\_data

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

对于编码，将系统内存数据拷贝到显存上并且进行VPP。比如，对于qsv调用qsv\_transfer\_data\_to

* av\_hwframe\_get\_buffer

int av\_hwframe\_get\_buffer(AVBufferRef \*hwframe\_ref, AVFrame \*frame, int flags)

从帧缓存池中获取未使用的buffer指配给frame的buffer

# 五 编码流程

## 5.1 软编码流程



上图虚线框部分为视频输出封装。

编码主要步骤如下：

1）查找编码器

可以通过接口AVCodec \*avcodec\_find\_encoder(enum AVCodecID id)设置，比如查找H264编码，设置id= AV\_CODEC\_ID\_H264即可找到H264编码器；

也可以通过AVCodec \*avcodec\_find\_encoder\_by\_name(const char \*name)设置，比如查找H264编码，设置name=libx264即可找到H264编码器。

2）设置编码器

首先，需申请编码器AVCodecContext，根据AVCodec信息通过接口AVCodecContext \*avcodec\_alloc\_context3(const AVCodec \*codec)创建AVCodecContext，同时将AVCodec赋值给成员codec；

设置编码器参数，比如比特率（码率），帧率，分辨率，像素格式等

3）打开编码器

参数设置之后，可通过接口int avcodec\_open2(AVCodecContext \*avctx, const AVCodec \*codec, AVDictionary \*\*options)打开编码器。

Options可以设置编码器的相关属性，或通过av\_opt\_set接口对AVCodecContext的成员priv\_data设置（必须在打开编码器前）。

4）视频编码

编码前，需申请AVFrame（原始视频帧存储空间）及AVPacket（编码后视频存储空间）

AVFrame：通过AVFrame \*av\_frame\_alloc(void)申请，而存储区需通过int av\_frame\_get\_buffer(AVFrame \*frame, int align)（申请前需指定分辨率及像素格式，align一般取32bit对齐）申请，最后通过void av\_frame\_free(AVFrame \*\*frame)释放。

AVPacket：通过AVPacket \*av\_packet\_alloc(void)申请，最后通过void av\_packet\_free(AVPacket \*\*pkt)释放。

编码通过int avcodec\_send\_frame(AVCodecContext \*avctx, const AVFrame \*frame) 将AVFrame发送到编码器，然后通过int avcodec\_receive\_packet(AVCodecContext \*avctx, AVPacket \*avpkt)将编码后的数据读到AVPacket（成员data,size）

## 5.2 硬编码流程



上图虚线框部分为视频硬编码部分。

硬编码主要步骤如下：

1）查找编码器

通过AVCodec \*avcodec\_find\_encoder\_by\_name(const char \*name)设置，比如查找QSV H264编码，设置name=h264\_qsv即可找到QSV H264编码器。

2）设置编码器

创建编码器与软编码一样，只是私有参数设置不同。

3）设置硬编码设备结构

通过接口av\_hwdevice\_ctx\_create创建硬编码设备结构，比如，AV\_HWDEVICE\_TYPE\_QSV创建QSV对应的结构，并获取内部处理接口HWContextType ff\_hwcontext\_type\_qsv（创建及初始化设备，分配帧缓存池等）

4）创建硬编帧结构及帧缓存池

通过av\_hwframe\_ctx\_alloc创建硬编帧结构，并设置相关参数（分辨率，硬编像素-qsv，帧像素-nv12，以及缓冲池大小）;

通过av\_hwframe\_ctx\_init分配帧缓存池。

同时，将硬编帧结构引用给AVCodecContex的成员hw\_frames\_ctx

5）打开编码器

与软编码一样。

6）硬编帧

通过av\_hwframe\_get\_buffer从帧缓存池中获取硬编帧的缓存；

通过av\_hwframe\_transfer\_data将系统内存的帧转换为显存硬编码帧。

7）视频编码

操作过程与软编码类似，只是需传入硬编帧，及采用ff\_h264\_qsv\_encoder结构编码。

## 5.3 编码设置

### 5.3.1 常用参数设置

码率，分辨率，帧率等常用参数设置如下：

首先通过libx264查找AVCodec ff\_libx264\_encoder

m\_ptVidEncCodec = avcodec\_find\_encoder\_by\_name(pchCodecName);

然后分配包含ff\_libx264\_encoder的AVCodecContext

m\_ptVidEncCodecCtx = avcodec\_alloc\_context3(m\_ptVidEncCodec);

最后给X264编码器设置参数

码率设置：

m\_ptVidEncCodecCtx->bit\_rate = 2048\*1000;

m\_ptVidEncCodecCtx->rc\_max\_rate = 2048\*1000;

m\_ptVidEncCodecCtx->rc\_min\_rate = 2048\*1000;

m\_ptVidEncCodecCtx->rc\_buffer\_size = m\_ptVidEncCodecCtx->rc\_max\_rate;

量化参数设置：

m\_ptVidEncCodecCtx->qmax = 18;

m\_ptVidEncCodecCtx->qmin = 40;

分辨率设置：

m\_ptVidEncCodecCtx->width = 1280;

m\_ptVidEncCodecCtx->height = 720;

时基设置

m\_ptVidEncCodecCtx->time\_base.num = 1;

m\_ptVidEncCodecCtx->time\_base.den = 25;

帧率设置：

m\_ptVidEncCodecCtx->framerate.num = 25;

m\_ptVidEncCodecCtx->framerate.den = 1;

GOP设置：

m\_ptVidEncCodecCtx->gop\_size = 50;

像素格式设置，YUV420P：

m\_ptVidEncCodecCtx->pix\_fmt = AV\_PIX\_FMT\_YUV420P;

### 5.3.2 profile/preset设置

必须在打开编码器前设置这些属性。

Profile: base

av\_opt\_set(m\_ptVidEncCodecCtx->priv\_data, "profile", "baseline", 0);

preset: veryfast

av\_opt\_set(m\_ptVidEncCodecCtx->priv\_data, "preset", “veryfast”, 0);

### 5.3.3 编码延时

X264默认编码延时：

param->i\_sync\_lookahead + // 前向预读帧数

max ( param->i\_bframe, + // B帧数量 3

param->rc.i\_lookahead) + // 码率控制前向预读帧数 40

param->i\_threads - 1. // 并行编码帧数 4

必须在打开编码器前设置此属性用于消除编码延时。

av\_opt\_set(m\_ptVidEncCodecCtx->priv\_data, "tune", "zerolatency", 0);

可以将i\_sync\_lookahead，i\_bframe， i\_lookahead置零，但b\_sliced\_threads=1。

### 5.3.4 并行编码

x264并行编码有slice并行编码和frame并行编码。

slice并行编码：把一帧划分为多个slice，每个slice是相互独立的，是一种非延时性的并行模式，多slice会稍微降低编码性能。

frame并行编码：需同时开启多帧编码，x264在N个frame并行的时候需要集齐N帧再开始一起编码，所以是一种延时性的并行模式。

对于对编码时间敏感的场景，不能采用frame并行，只能采用slice并行或者不做并行编码。

X264一个nalu单元一个slice或一个frame

设置如下：

* 帧编码：

未设置zerolatency，是进行帧frame并行编码，会有上一节的延时。

如果设置zerolatency,且i\_threads>1,并设sliced-threads=0，会进行帧并行编码，但会有i\_threads – 1帧数延时；

av\_opt\_set(m\_ptVidEncCodecCtx->priv\_data, "x264-params", "sliced-threads=0", 0);

* 片编码

如果设置zerolatency,且i\_threads>1,会进行多片slice并行编码，片数为threads；

av\_opt\_set(m\_ptVidEncCodecCtx->priv\_data, "tune", "zerolatency", 0);

设置后只有sliced\_threads为1，i\_thraeds不变，其余为0；

* 单片编码

如果设置zerolatency,并设i\_threads=1和sliced-threads=0，会进行单片slice编码(帧frame编码)，没有延时，只有单线程编码。

av\_opt\_set(m\_ptVidEncCodecCtx->priv\_data, "x264-params", "sliced-threads=0", 0);

av\_opt\_set(m\_ptVidEncCodecCtx->priv\_data, "x264-params", "threads=1", 0);

### 5.3.5 关键帧设置

编码前设置AVFrame \*frm

frm->pict\_type = FF\_I\_TYPE;

frm->key\_frame = 1;

# 六 编码实例

## 6.1 数据结构

数据结构关系图：



由两部分组成：

一部分：数据封装，CFFmpegVideoFormatWrapper

一部分：数据编码，CFFVideoEncoderObj

CFFVideoEncoderObj类主要实现：

初始化视频编码参数；

编码一帧视频帧操作；

刷新一帧视频帧操作；

设置视频关键帧。

## 6.2 接口

* 编码句柄

void\* FFVideoEnc\_CreateInstance ()

功能：

返回编码句柄

* 销毁编码句柄

s16 FFVideoEnc\_DestoryInstance (void\* pHandle)

功能：

销毁编码句柄释放资源

参数：

编码器句柄

* 初始化编码器

s16 FFVideoEnc\_InitEncoder(void\* pHandle, em\_FFVideoEncoderType emType, const TFFVideoEncParam \*ptVdioEncParam);

功能：

利用参数初始化编码器以及打开编码器实例。

参数：

ptVdioEncParam：指定编码类型（H264/H265）以及分辨率，码率，帧率等信息；

emType： 指定编码方式（软编码/Intel编码/AMD编码/NVIDIA编码方式）

返回值：

= 0 ： 表示成功

< 0 ：表示初始化失败

* 销毁编码器

void FFVideoEnc\_CloseEncoder(void\* pHandle);

功能：

释放编码器的相关资源

* 编码一帧视频

s16 FFVideoEnc\_EncodeFrame(void\* pHandle, u8 \*pbyOutData, u32 &dwOutLen, BOOL32 &bKeyFlag, const TFFVideoYUVFrame \*pTVidRawData);

功能：

转换输入数据为内部数据，并进行视频编码压缩。

参数：

pbyOutData：存储一帧视频；

dwOutLen：一帧视频大小，为0时，表示没有产生一帧数据；

bKeyFlag ： 此输出帧为关键帧；

pTVidRawData：输入的原始数据封装。

返回值：

=2 ： 表示有编码数据；

=1 ： 表示编码结束；

= 0 ： 表示没有编码数据；

< 0 ： 表示编码失败。

* 刷新编码器

对于有延时的编码而言，停编码后编码器中可能会缓存几帧数据缓存。

s16 FFVideoEnc\_FlushFrame(void\* pHandle, u8 \*pbyOutData, u32 &dwOutLen, BOOL32 &bKeyFlag);

功能：

取出缓存数据。

参数：

pbyOutData：存储一帧视频；

dwOutLen：一帧视频大小，为0时，表示没有产生一帧数据；

bKeyFlag ： 此输出帧为关键帧；

返回值：

=2 ： 表示有编码数据；

=1 ： 表示编码结束；

= 0 ： 表示没有编码数据；

< 0 ： 表示编码失败。

* 请求编码一个关键帧

s16 FFVideoEnc\_RequestIDR(void\* pHandle);

## 6.3 实例

* 软编码

程序命令格式：SWFFmpeg.exe <width> <height> <input file>

<input file> : 输入原始视频数据，比如yuv

<width>: 原始视频数据的宽

<height>: 原始视频数据的高

SWFFmpeg.exe 640 480 E:\code\project\ffmpegSdk\video\0001\_640\_480\_I420.yuv

* 硬编码

程序命令格式：HWFFmpeg.exe <width> <height> <input\_pixel> [<input file>]

<input\_pixel>： 输入文件的像素格式，0-yuv420p,1-nv12

<input file> : 输入原始视频数据，比如yuv

<width>: 原始视频数据的宽

<height>: 原始视频数据的高

HWFFmpeg.exe 640 480 0 E:\code\project\ffmpegSdk\video\0001\_640\_480\_I420.yuv