# 环境搭建

需要完整安装ffmpeg

/home/lizhiyong/rtmp/xiongzhiliang

有完整的FFMPEG安装指南，和官方教程相比其他都是浮云.

https://trac.ffmpeg.org/wiki/CompilationGuide/Ubuntu

## 下载源码包

libfaac faac格式的编解码包

libmp3lame mp3格式编解码包

libopencore-amrwb libopencore-amrnb amr格式编解码包

libx264 x264格式编解码包

libvorbis ogg格式编解码包

libxvid xvid格式编解码包

安装

## 安装

0001.官方说明：

http://ffmpeg.org/general.html

External libraries

x265

FFmpeg can make use of the x265 library for HEVC encoding.

Go to http://x265.org/developers.html and follow the instructions for installing the library. Then pass --enable-libx265 to configure to enable it.

x265 is under the GNU Public License Version 2 or later (see http://www.gnu.org/licenses/old-licenses/gpl-2.0.html for details), you must upgrade FFmpeg’s license to GPL in order to use it.

0002.安装x264

$git clone git://git.videolan.org/x264.git

$cd x264

$./configure --enable-static --disable-opencl --disable-avs --disable-cli --disable-ffms --disable-gpac --disable-lavf --disable-swscale

$make

$make install

0003.安装x265

https://bitbucket.org/multicoreware/x265/wiki/Home

# ubuntu packages:

$ sudo apt-get install mercurial cmake cmake-curses-gui build-essential yasm

# Note: if the packaged yasm is older than 1.2, you must download yasm-1.2 and build it

$ hg clone https://bitbucket.org/multicoreware/x265

$ cd x265/build/linux

$ ./make-Makefiles.bash

# 这里将 LOG\_CU\_STATISTICS　设置为ON，然后，按下“c”，实现configure，按下“q”退出

$ make

$ make install

0004.下载ffmpeg

从 http://ffmpeg.org/download.html 下载ffmpeg 2.1版本源码

$./configure --enable-libx264 --enable-libx265 --enable-gpl

$make

$make install

# 常用的命令

https://ffmpeg.org/documentation.html

## 分离视频音频流

-vn和-an分别是屏蔽视频流和屏蔽音频流，分别对源文件处理一次即可得到分离的音频和视频。

ffmpeg -i input\_file -vcodec copy -an output\_file\_video　　//分离视频流

ffmpeg -i input\_file -acodec copy -vn output\_file\_audio　　//分离音频流

**范例 2-1 分离音频，分离视频**

ffmpeg -i source.200kbps.768x320.flv -acodec copy -vn audio.aac

ffmpeg -i source.200kbps.768x320.flv -vcodec copy -an video.mp4

解到容器里

测试分离的文件

ffplay -stats -f h264

ffplay -stats -f h264

## 视频解复用

ffmpeg -i test.mp4 -vcodec copy -an -f m4v test.264

ffmpeg -i test.avi -vcodec copy -an -f m4v test.264

范例2-2 视频解复用

ffmpeg -i source.200kbps.768x320.flv -f h264 video.h264

## 视频转码

ffmpeg -i test.mp4 -vcodec h264 -s 352\*278 -an -f m4v test.264 //转码为码流原始文件

ffmpeg -i test.mp4 -vcodec h264 -bf 0 -g 25 -s 352\*278 -an -f m4v test.264 //转码为码流原始文件

ffmpeg -i test.avi -vcodec mpeg4 -vtag xvid -qsame test\_xvid.avi //转码为封装文件

//-bf B帧数目控制，-g 关键帧间隔控制，-s 分辨率控制

## 视频封装

ffmpeg -i video\_file -i audio\_file -vcodec copy -acodec copy output\_file

范例

ffmpeg -i source.200kbps.768x320.flv -vcodec copy -acodec copy video.mp4

ffmpeg -i source.200kbps.768x320.flv -vcodec copy -acodec copy video.mkv

当转为ts和avi格式时报错

ffmpeg -i source.200kbps.768x320.flv -vcodec copy -acodec copy video.ts

报错

[mpegts @ 0xc724e0] H.264 bitstream malformed, no startcode found, use the video bitstream filter 'h264\_mp4toannexb' to fix it ('-bsf:v h264\_mp4toannexb' option with ffmpeg)

av\_interleaved\_write\_frame(): Invalid data found when processing input

改为

ffmpeg -i source.200kbps.768x320.flv -vcodec copy -acodec copy -bsf video.ts

报错

At least one output file must be specified

## 视频剪切

ffmpeg -i test.avi -r 1 -f image2 image-%3d.jpeg //提取图片

ffmpeg -ss 0:1:30 -t 0:0:20 -i input.avi -vcodec copy -acodec copy output.avi //剪切视频

//-r 提取图像的频率，-ss 开始时间，-t 持续时间

6.视频录制

ffmpeg -i rtsp://192.168.3.205:5555/test -vcodec copy out.avi

7.YUV序列播放

ffplay -f rawvideo -video\_size 1920x1080 input.yuv

8.YUV序列转AVI

ffmpeg -s w\*h -pix\_fmt yuv420p -i input.yuv -vcodec mpeg4 output.avi

1、ffmpeg使用语法

命令格式：

ffmpeg -i [输入文件名] [参数选项] -f [格式] [输出文件]

ffmpeg [[options][`-i' input\_file]]... {[options] output\_file}...

1、参数选项：

(1) -an: 去掉音频

(2) -acodec: 音频选项， 一般后面加copy表示拷贝

(3) -vcodec:视频选项，一般后面加copy表示拷贝

2、格式：

(1) h264: 表示输出的是h264的视频裸流

(2) mp4: 表示输出的是mp4的视频

(3)mpegts: 表示ts视频流

如果没有输入文件，那么视音频捕捉（只在Linux下有效，因为Linux下把音视频设备当作文件句柄来处理）就会起作用。作为通用的规则，选项一般用于下一个特定的文件。如果你给 -b 64选项，改选会设置下一个视频速率。对于原始输入文件，格式选项可能是需要的。缺省情况下，ffmpeg试图尽可能的无损转换，采用与输入同样的音频视频参数来输出。（by ternence.hsu）

2、视频转换

H264视频转ts视频流

ffmpeg -i test.h264 -vcodec copy -f mpegts test.ts

H264视频转mp4

ffmpeg -i test.h264 -vcodec copy -f mp4 test.mp4

ts视频转mp4

ffmpeg -i test.ts -acodec copy -vcodec copy -f mp4 test.mp4

mp4视频转flv

ffmpeg -i test.mp4 -acodec copy -vcodec copy -f flv test.flv

转换文件为3GP格式

ffmpeg -y -i test.mpeg -bitexact -vcodec h263 -b 128 -r 15 -s 176x144 -acodec aac -ac 2 -ar 22500 -ab 24 -f 3gp test.3gp

转换文件为3GP格式 v2

ffmpeg -y -i test.wmv -ac 1 -acodec libamr\_nb -ar 8000 -ab 12200 -s 176x144 -b 128 -r 15 test.3gp

使用 ffmpeg 编码得到高质量的视频

ffmpeg.exe -i "D:\Video\Fearless\Fearless.avi" -target film-dvd -s 720x352 -padtop 64 -padbottom 64 -maxrate 7350000 -b 3700000 -sc\_threshold 1000000000 -trellis -cgop -g 12 -bf 2 -qblur 0.3 -qcomp 0.7 -me full -dc 10 -mbd 2 -aspect 16:9 -pass 2 -passlogfile "D:\Video\ffmpegencode" -an -f mpeg2video "D:\Fearless.m2v"

转换指定格式文件到FLV格式

ffmpeg.exe -i test.mp3 -ab 56 -ar 22050 -b 500 -r 15 -s 320x240 f:\test.flv

ffmpeg.exe -i test.wmv -ab 56 -ar 22050 -b 500 -r 15 -s 320x240 f:\test.flv

转码解密的VOB

ffmpeg -i snatch\_1.vob -f avi -vcodec mpeg4 -b 800 -g 300 -bf 2 -acodec mp3 -ab 128 snatch.avi

（上面的命令行将vob的文件转化成avi文件，mpeg4的视频和mp3的音频。注意命令中使用了B帧，所以mpeg4流是divx5兼容的。GOP大小是300意味着29.97帧频下每10秒就有INTRA帧。该映射在音频语言的DVD转码时候尤其有用，同时编码到几种格式并且在输入流和输出流之间建立映射）

转换文件为3GP格式

ffmpeg -i test.avi -y -b 20 -s sqcif -r 10 -acodec amr\_wb -ab 23.85 -ac 1 -ar 16000 test.3gp

（如果要转换为3GP格式，则ffmpeg在编译时必须加上-enable-amr\_nb -enable-amr\_wb，详细内容可参考：转换视频为3GPP格式）

转换文件为MP4格式（支持iPhone/iTouch）

ffmpeg -y -i input.wmv -f mp4 -async 1-s 480x320 -acodec libfaac -vcodec libxvid -qscale 7 -dts\_delta\_threshold 1 output.mp4

ffmpeg -y -i source\_video.avi input -acodec libfaac -ab 128000 -vcodec mpeg4 -b 1200000 -mbd 2 -flags +4mv+trell -aic 2 -cmp 2 -subcmp 2 -s 320x180 -title X final\_video.mp4

将一段音频与一段视频混合

ffmpeg -i son.wav -i video\_origine.avi video\_finale.mpg

将一段视频转换为DVD格式

ffmpeg -i source\_video.avi -target pal-dvd -ps 2000000000 -aspect 16:9 finale\_video.mpeg

（target pal-dvd : Output format ps 2000000000 maximum size for the output file, in bits (here, 2 Gb) aspect 16:9 : Widescreen）

转换一段视频为DivX格式

ffmpeg -i video\_origine.avi -s 320x240 -vcodec msmpeg4v2 video\_finale.avi

Turn X images to a video sequence

ffmpeg -f image2 -i image%d.jpg video.mpg

（This command will transform all the images from the current directory (named image1.jpg, image2.jpg, etc...) to a video file named video.mpg.）

Turn a video to X images

ffmpeg -i video.mpg image%d.jpg

（This command will generate the files named image1.jpg, image2.jpg, ... ；The following image formats are also availables : PGM, PPM, PAM, PGMYUV, JPEG, GIF, PNG, TIFF, SGI.）

使用ffmpeg录像屏幕(仅限Linux平台)

ffmpeg -vcodec mpeg4 -b 1000 -r 10 -g 300 -vd x11:0,0 -s 1024x768 ~/test.avi

（-vd x11:0,0 指录制所使用的偏移为 x=0 和 y=0，-s 1024×768 指录制视频的大小为 1024×768。录制的视频文件为 test.avi，将保存到用户主目录中；如果你只想录制一个应用程序窗口或者桌面上的一个固定区域，那么可以指定偏移位置和区域大小。使用xwininfo -frame命令可以完成查找上述参数。）

重新调整视频尺寸大小(仅限Linux平台)

ffmpeg -vcodec mpeg4 -b 1000 -r 10 -g 300 -i ~/test.avi -s 800×600 ~/test-800-600.avi

把摄像头的实时视频录制下来，存储为文件(仅限Linux平台)

ffmpeg -f video4linux -s 320\*240 -r 10 -i /dev/video0 test.asf

使用ffmpeg压制H.264视频

ffmpeg -threads 4 -i INPUT -r 29.97 -vcodec libx264 -s 480x272 -flags +loop -cmp chroma -deblockalpha 0 -deblockbeta 0 -crf 24 -bt 256k -refs 1 -coder 0 -me umh -me\_range 16 -subq 5 -partitions parti4x4+parti8x8+partp8x8 -g 250 -keyint\_min 25 -level 30 -qmin 10 -qmax 51 -trellis 2 -sc\_threshold 40 -i\_qfactor 0.71 -acodec libfaac -ab 128k -ar 48000 -ac 2 OUTPUT

（使用该指令可以压缩出比较清晰，而且文件转小的H.264视频文件）

3、网络推送

udp视频流的推送

ffmpeg -re -i 1.ts -c copy -f mpegts udp://192.168.0.106:1234

4、视频拼接

裸码流的拼接，先拼接裸码流，再做容器的封装

ffmpeg -i "concat:test1.h264|test2.h264" -vcodec copy -f h264 out12.h264

5、图像相关

截取一张352x240尺寸大小的，格式为jpg的图片

ffmpeg -i test.asf -y -f image2 -t 0.001 -s 352x240 a.jpg

把视频的前30帧转换成一个Animated Gif

ffmpeg -i test.asf -vframes 30 -y -f gif a.gif

截取指定时间的缩微图,-ss后跟的时间单位为秒

ffmpeg -i test.avi -y -f image2 -ss 8 -t 0.001 -s 350x240 test.jpg

6、音频处理

转换wav到mp2格式

ffmpeg -i /tmp/a.wav -ab 64 /tmp/a.mp2 -ab 128 /tmp/b.mp2 -map 0:0 -map 0:0

（上面的命令行转换一个64Kbits 的a.wav到128kbits的a.mp2 ‘-map file:index’在输出流的顺序上定义了哪一路输入流是用于每一个输出流的。）

7、切割ts分片

ffmpeg -i input.mp4 -c:v libx264 -c:a aac -strict -2 -f hls -hls\_list\_size 6 -hls\_time 5 output1.m3u8

## 播放

## 转码

## 添加水印

# 编程范例

演示范例的开发原则，以总分总的方式进行，先演示一个简单播放器的开发；然后演示音视频的编解码，音频的重采样（音效处理），视频画面处理（尺寸改变、画面改变）；最后使用开发一个完整的播放器。

1. 简单播放器
2. 解封装音视频分离
3. 解码H264，并存储为yuv格式和rgb格式
4. 解码AAC，并存储为PCM
5. Yuv编码为H264
6. PCM编码为AAC
7. 将H264和AAC封装为FLV格式
8. 将YUV和PCM编码后封装为FLV格式
9. 将FLV格式文件转封装为MP4/TS/MKV
10. 视频改变缩放比例,并保存为原始格式
11. 音频图像处理（调整亮度、对比度等）
12. 音频重采样
13. 视频转码H264转MP4
14. 视频转码MP4转码H264
15. 音频MP3转码AAC
16. 音频AAC转码MP3
17. 增加水印
18. 为视频添加/更换背景音乐
19. 两段视频做拼接，不同封装不同的解码器，输出指定为FLV封装，H264+AAC编码
20. 实现画中画效果并重新编码
21. 画面缩放裁剪
22. 自定义filter
23. Ffmpeg.c分析
24. Ffplay.c分析
25. Ffprobe.c分析

## 简单播放器

源码：3-1-palyer

视频显示独立，音频播放独立。方便后续的范例使用。

**基础知识**

PTS

DTS

PCM

YUV

RGB

播放器基本架构





### 框架分析

FFmpeg主要数据架构

AVFormatContext

AVInputFormat

AVOutputFormat

AVCodecContext

AVCodec

AVStream

AVPacket

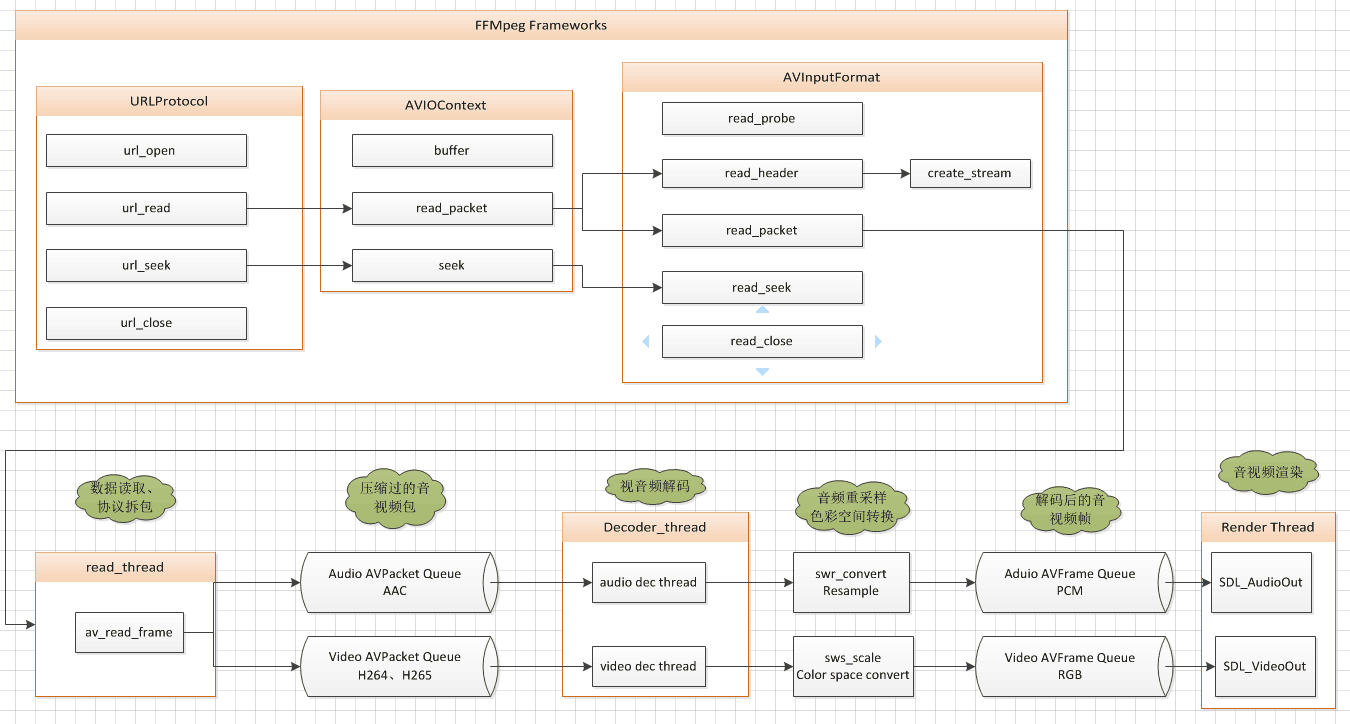
AVFrame

#### 背景

随着游戏娱乐等直播业务的增长，在移动端观看直播的需求也日益迫切。但是移动端原生的播放器对各种直播流的支持却不是很好。Android 原生的 MediaPlayer 不支持 flv、hls 直播流，iOS 只支持标准的 HLS 流。本文介绍一种基于 ffplay 框架下的跨平台播放器的实现，且兼顾硬解码的实现。

#### 播放器原理

直观的讲，我们播放一个媒体文件一般需要5个基本模块，按层级顺序：文件读取模块（Source）、解复用模块（Demuxer）、视频频解码模块（Decoder）、色彩空间转换模块（Color Space Converter）、音视频渲染模块（Render）。数据的流向如下图所示，其中 ffmpeg 框架包含了文件读取、音视频解复用的模块。



文件读取模块（Source）的作用是为下级解复用模块（Demuxer）以包的形式源源不断的提供数据流,对于下一级的Demuxer来说，本地文件和网络数据是一样的。在ffmpeg框架中，文件读取模块可分为3层：

协议层： pipe，tcp，udp，http等这些具体的本地文件或网络协议

抽象层：URLContext结构来统一表示底层具体的本地文件或网络协议

接口层用：AVIOContext结构来扩展URLProtocol结构成内部有缓冲机制的广泛意义上的文件，并且仅仅由最上层用AVIOContext对模块外提供服务，实现读媒体文件功能。

解复用模块（Demuxer）：的作用是识别文件类型，媒体类型，分离出音频、视频、字幕原始数据流，打上时戳信息后传给下级的视频频解码模块（Decoder）。可以简单的分为两层，底层是 AVIContext，TCPContext，UDPContext 等等这些具体媒体的解复用结构和相关的基础程序，上层是 AVInputFormat 结构和相关的程序。上下层之间由 AVInputFormat 相对应的 AVFormatContext 结构的 priv\_data 字段关联 AVIContext 或 TCPContext 或 UDPContext 等等具体的文件格式。AVInputFormat 和具体的音视频编码算法格式由 AVFormatContext 结构的 streams 字段关联媒体格式，streams 相当于 Demuxer 的 output pin，解复用模块分离音视频裸数据通过 streams 传递给下级音视频解码器。

视频频解码模块（Decoder）的作用就是解码数据包，并且把同步时钟信息传递下去。

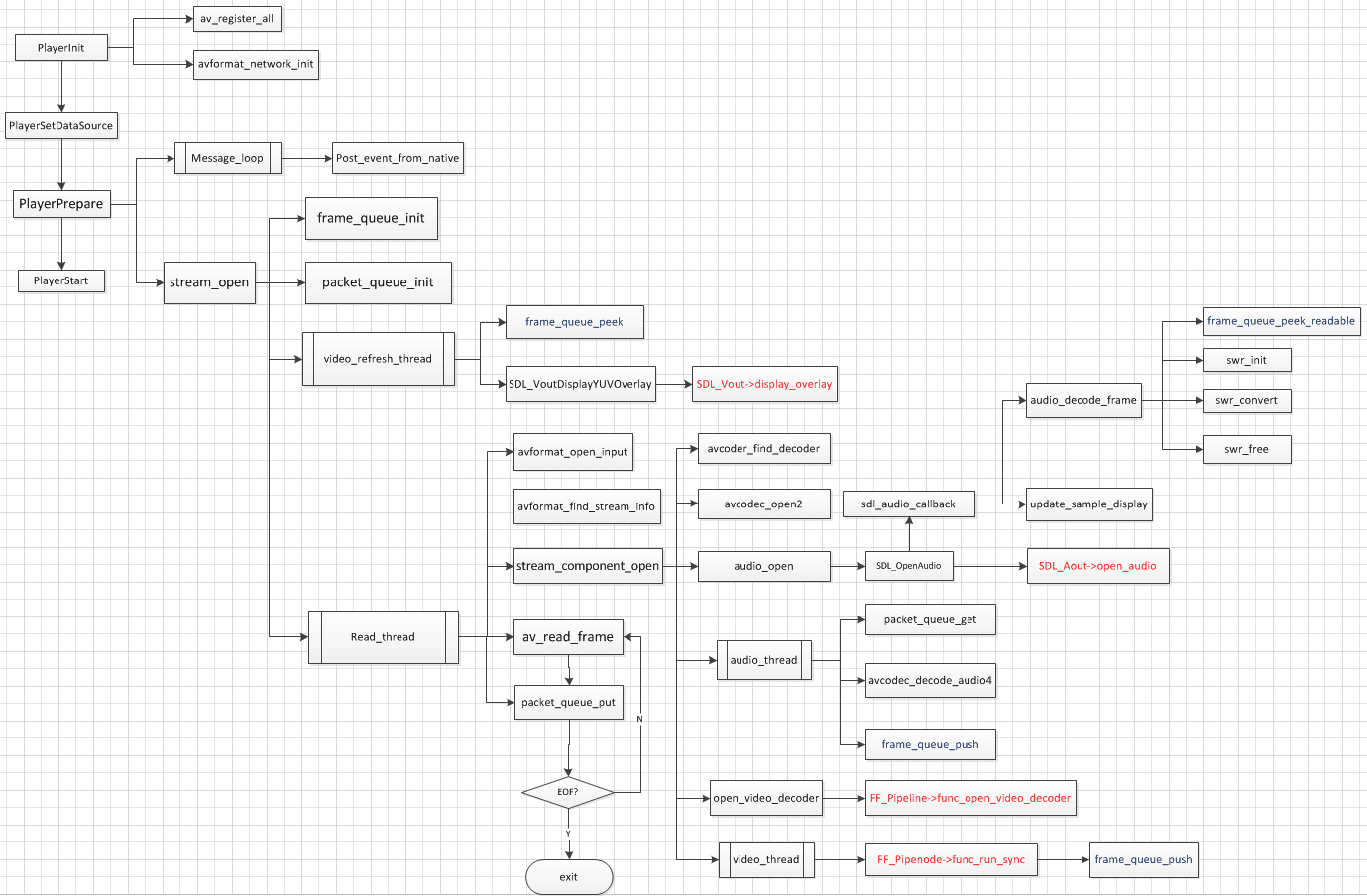
色彩空间转换模块（Color Space Converter）颜色空间转换过滤器的作用是把视频解码器解码出来的数据转换成当前显示系统支持的颜色格式

音视频渲染模块（Render）的作用就是在适当的时间渲染相应的媒体，对视频媒体就是直接显示图像，对音频就是播放声音

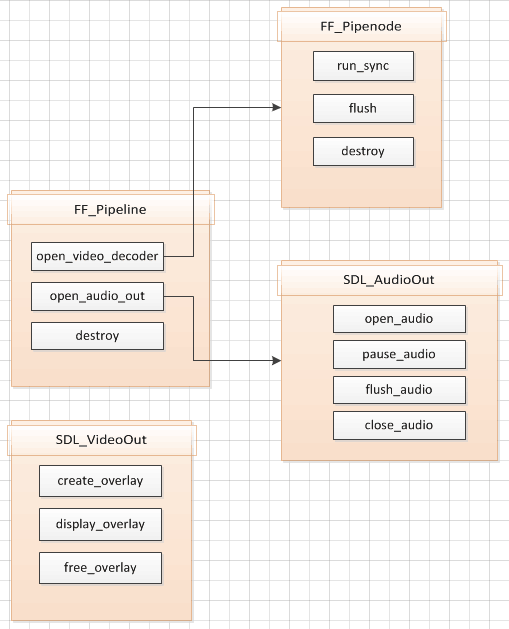
#### 跨平台实现

在播放器得5个模块中文件读取模块（Source）、解复用模块（Demuxer）和色彩空间转换模块（Color Space Converter）这三个模块都可以用 ffmpeg 的框架进行实现，而f fmpeg 本身就是跨平台的。因此，实现跨平台的播放器的就需要抽象一层平台无关的音视频解码、渲染接口。Android、iOS、Window 等平台只需要实现各自平台的渲染、硬件解码（如果支持的话）就可以构建一个标准的基于 ffmpeg 的播放器了。

下图是基于ffplay的基本播放流程图：



图中红色部分是需要抽象的接口的，结构如下：



其中 FF\_Pipenode.run\_sync 视频解码线程，默认有 libavcodec 的软解码实现，其他平台可以增加自己的硬解码实现。SDL\_VideoOut 为视频渲染抽象层，这里 overlay 可以是 Android的 NativeWindow，或者是 OpenGL 的 Texture。SDL\_AudioOut 是音频播放抽象层，可以直接操作声卡驱动，SDL2.0 里就支持 ALSA、OSS 接口，当然也可以用 Android、iOS SDK 中的音频 API 实现。

这里顺便提下，随着 Android、iOS 平台的普及，ffmpeg 版本的也逐步支持了 Android、iOS 的硬件解码器，如f fmpeg 在很早之前就支持了 libstagefright，最新的 ffmpeg2.8 也已经支持了 iOS 的硬件解码库 VideoToolBox。从下面重点介绍下视频硬解码以及音视频渲染模块在移动平台上的实现。

### 代码分析

### 结果验证

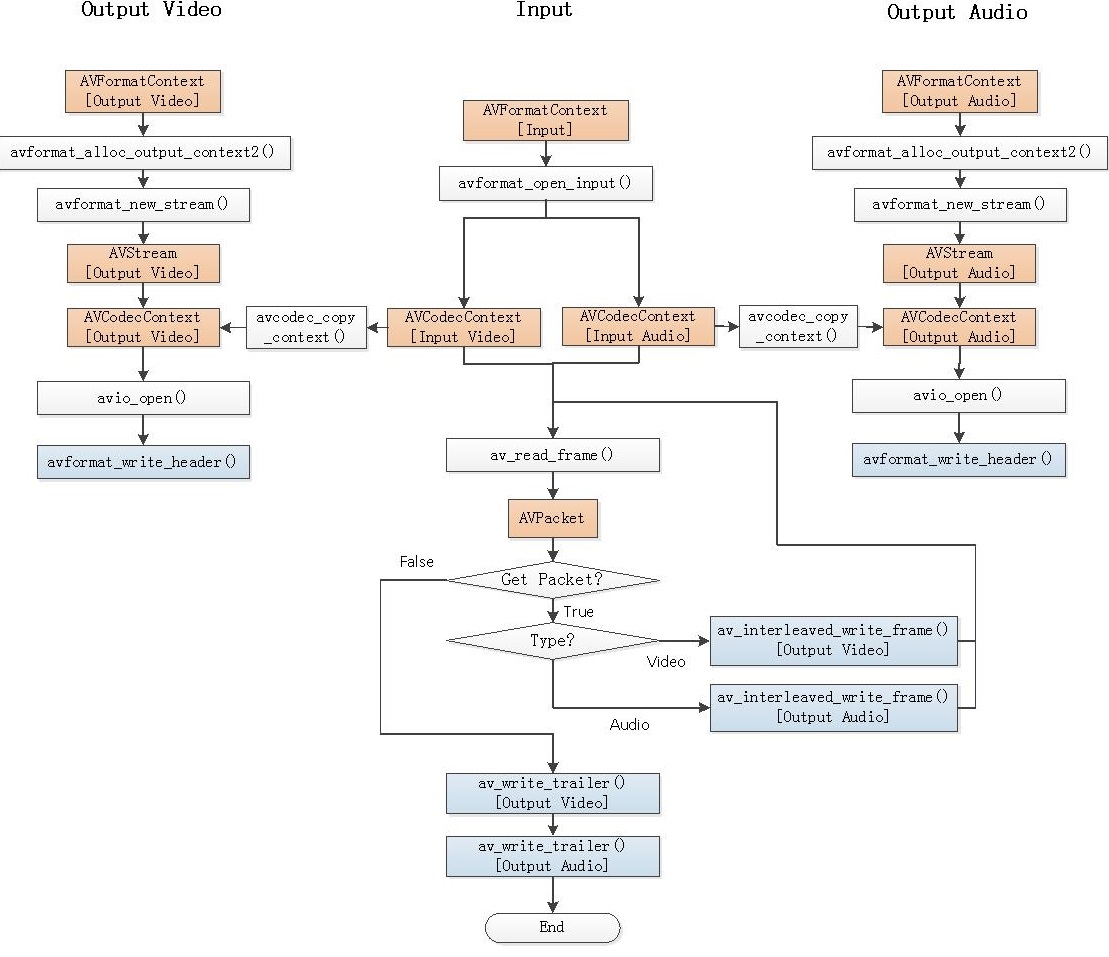
### 参考文档

[1] 基于 ffmpeg 的跨平台播放器实现 https://www.cnblogs.com/stnlcd/p/7261879.html

## 解封装音视频分离

源码：3-2-demuxer

### 框架



雷大神

### 代码分析

### 验证分离后的文件

使用ffplay进行播放

播放H264文件

ffplay source.200kbps.768x320.h264

播放AAC文件

ffprobe source.200kbps.768x320.aac

### 参考

1. FFmpeg解封装音视频分离 <https://www.cnblogs.com/ht-beyond/p/5225664.html.>

## 解码H264，并存储为yuv格式和rgb格式

源码：3-3-decode\_h264

将3-2分离出来的文件source.200kbps.768x320.h264作为操作对象。

本文存储yuv和rgb的时候，都对图像做了缩放的操作。

### 框架

### 代码分析

Yuv和rgb的数据存储格式存在不同。

### 结果验证

使用ffplay进行播放

ffplay -f rawvideo -video\_size 768x320 768x320.yuv

ffplay -f rawvideo -pixel\_format rgb24 -video\_size 1920x1080 e:\workspace\encoder\output\test\_cff.rgb

播放RBG文件

ffplay -f rawvideo -pixel\_format rgb24 -video\_size 240x135 240x135.rgb

播放YUV文件

ffplay -f rawvideo -pixel\_format yuv420p -video\_size 320x180 320x180.yuv

### 参考

1. 将h264编码的视频流保存为图片http://blog.csdn.net/oldmtn/article/details/46742555

## 解码AAC，并存储为PCM

范例 3-4-decode\_aac

解码aac，使用3-2分离出来的aac文件

解码后的文件保存为

### 结果验证

{ AV\_SAMPLE\_FMT\_U8, "u8", "u8" },

{ AV\_SAMPLE\_FMT\_S16, "s16be", "s16le" },

{ AV\_SAMPLE\_FMT\_S32, "s32be", "s32le" },

{ AV\_SAMPLE\_FMT\_FLT, "f32be", "f32le" },

{ AV\_SAMPLE\_FMT\_DBL, "f64be", "f64le" },

ffplay -ar 44100 -channels 2 -f s16le -i xxx.pcm

**播放f32lex2.pcm**

ffplay -ar 44100 -channels 2 -f f32le -i f32lex2.pcm

**播放s16lex2.pcm**

ffplay -ar 44100 -channels 2 -f s16le s16lex2.pcm

ffplay -ar 96000 -channels 2 -f s16le s16lex2.pcm

## Yuv编码为H264

源码：3-5-encoder\_yuv\_to\_h264

框架分析

源码分析

### 结果验证

width=352,height=288

Codec not found

H264编码器打不开，是因为编码器没有移植。

### 参考文档

## PCM编码为AAC

范例 3-6-encode\_aac

使用范例3-4-decode\_aac产生的PCM数据作为实验，分不同的PCM编码进行编码，同时编码两个文件。

### 框架

pcm是最原始的音频编码格式，这种编码是无损的。同时意味着存储这种数据的文件将会很庞大，因此必须进行压缩。pcm是音频的编码格式，它不是文件的封装格式，上一节我们录制的声音存储在一个.pcm为后缀的文件中，这只是我们愿意这么做而已，你完全可以不这么做，这没有关系。

aac既是一种文件的封装格式，又是音频的编码格式。一aac为封装格式的文件，以.aac为后缀。aac封装格式一般内部的音频数据编码格式也为aac。

音频编码和视频编码的流程基本一致，而视频编码我们在前面已经做过了。因此，关于程序的流程就没有太多需要废话的了。下面介绍几个音频相关的参数，这几个参数是编码器进行编码所必需的。

我们总共需要设置四个参数即可：

1.sample\_rate

codecContext->sample\_rate = frame->sample\_rate;

sample\_rate指的是采样率。也就是我们一秒钟采集多少次声音样本。

2.frame->channels

codecContext->channels = frame->channels;

frame->channels之的是通道的数目。音频一般有双通道或者单通道之分，一般都是双通道吧，我们的程序里面也是设置为双通道的。也就是frame->channels=2.

3.frame->format

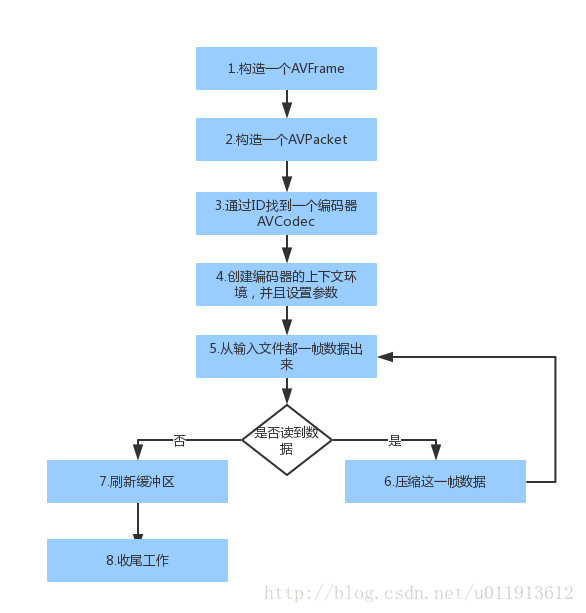
codecContext->sample\_fmt = frame->format;

frame->format指的是样本的格式。一个音频的样本一般用两个字节来描述，分为大小端。我们的程序中使用的是16bit的小端格式。

4.channel\_layout

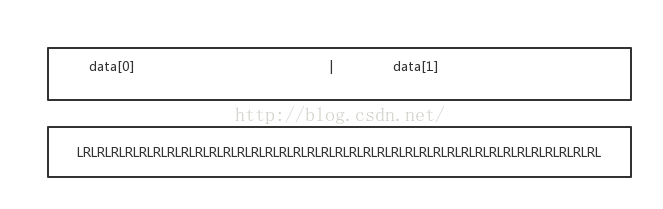
codecContext->channel\_layout = AV\_CH\_LAYOUT\_STEREO;

channel\_layout 用来设置输出通道布局。这个参数不太理解！！！

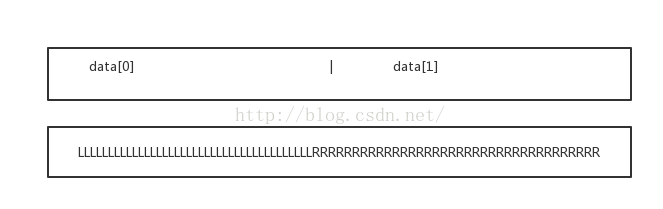


由于AVframe结构体data指针数组不同指针代表指向不同声道的数据，所以产生错误。

上述代码，data指向情况：



而FFmpeg编码PCM为AAC时，需要的是：



所以，我要让到读取一帧时，刚好让data[0]指向一个声道的数据，而data[1]指向另一个声道的数据。

### 源码分析

### 结果验证

编码aac文件

./3-6-encode\_aac ../f32lex2\_44100.pcm test.aac

使用平台的播放播放test.aac即可。

### 参考

1. FFmpeg(2016)PCM编码AAC <http://blog.csdn.net/jammg/article/details/52684894>
2. 将H264和AAC封装为FLV格式

## 将H264和AAC封装为FLV格式

源码：3-7-h264\_aac\_muxer

### 框架

FFmpeg官方文档对aac\_adtstoasc的说明：



1）将AAC编码器编码后的原始码流（ADTS头 + ES流）封装为MP4或者FLV或者MOV等格式时，需要先将ADTS头转换为MPEG-4 AudioSpecficConfig （将音频相关编解码参数提取出来），并将原始码流中的ADTS头去掉（只剩下ES流）。

2）相反，从MP4或者FLV或者MOV等格式文件中解封装出AAC码流（只有ES流）时，需要在解析出的AAC码流前添加ADTS头（含音频相关编解码参数）。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ADTS AAC | | | | | | |
| ADTS\_header | AAC ES | ADTS\_header | AAC ES | ... | ADTS\_header | AAC ES |

MP4 container requires AV\_CODEC\_FLAG\_GLOBAL\_HEADER, which means all stream should contains stream data only, and other data is provided by setting AVCodecContext.extradata. Because MP4 has its own way of transporting meta information (here transport info), writing that transport prefix before each frame will make the data unreadable.

//ofmt\_ctx is AVFormatContext//enc\_ctx is the AVCodecContext of the current stream if (ofmt\_ctx->oformat->flags & AVFMT\_GLOBALHEADER)

enc\_ctx->flags |= CODEC\_FLAG\_GLOBAL\_HEADER;

Without them the encoder may add the metadatas in the data which is sent to the container. For AACit is the ADTS header, for H264 it is SPS and PPS data.

### 代码分析

### 结果验证

过程错误

Malformed AAC bitstream detected: use the audio bitstream filter 'aac\_adtstoasc' to fix it

不同的封装格式不同。

./3-7-h264\_aac\_muxer output.mkv

./3-7-h264\_aac\_muxer output-mp3.flv

./3-7-h264\_aac\_muxer output-mp3.mp4

测试了AAC+H264、MP3+H264封装成ts/flv/mp4/mkv格式没有问题。

为后续的音乐替换背景打下基础。

### 参考

1. ffmpegj将h264与aac封装成mkv/ts或者h264与mp3封装成mp4/flv/avi测试代码http://blog.csdn.net/chenyefei/article/details/52329988
2. aac\_adtstoasc bitstream filter http://blog.csdn.net/liuyl2016/article/details/53080733

## 将YUV和PCM编码后封装为FLV格式（暂未处理）

## 将FLV格式文件转封装为MP4/TS/MKV

源码：3-9-remuxer

不同封装直接的转换

### 框架分析

### 源码分析

[mpegts @ 0x22761a0] H.264 bitstream malformed, no startcode found, use the h264\_mp4toannexb bitstream filter (-bsf h264\_mp4toannexb)

av\_interleaved\_write\_frame(): Invalid argument

主要是因为使用了mp4中的h264编码，而h264有两种封装：

一种是annexb模式，传统模式，有startcode，SPS和PPS是在ES中；另一种是mp4模式，一般mp4、mkv、avi会没有startcode，SPS和PPS以及其它信息被封装在container中，每一个frame前面是这个frame的长度，很多解码器只支持annexb这种模式，因此需要将mp4做转换；在ffmpeg中用h264\_mp4toannexb\_filter可以做转换；所以需要使用-bsf h264\_mp4toannexb来进行转换；

1. 初始化filterav\_bitstream\_filter\_init

2. 使用filter转换av\_bitstream\_filter\_filter

3. 关闭filter av\_bitstream\_filter\_close

if (pkt.stream\_index == 0)

{

int a = av\_bitstream\_filter\_filter(h264bsfc, in\_stream->codec, NULL, &pkt.data, &pkt.size,

pkt.data, pkt.size, pkt.flags & AV\_PKT\_FLAG\_KEY);

}

### 结果验证

./3-9-remuxer ../input-mp3.flv output.mkv

错误的情况

./3-9-remuxer ../input-mp3.flv output.avi

[avi @ 0x2153700] H.264 bitstream malformed, no startcode found, use the video bitstream filter 'h264\_mp4toannexb' to fix it ('-bsf:v h264\_mp4toannexb' option with ffmpeg)

### 参考文档

[1]http://blog.chinaunix.net/uid-11344913-id-4432752.html

## 视频改变缩放比例,并保存为原始格式(暂不处理)

源码：3-8-scaling\_video

框架分析

源码分析

结果验证

参考文档

## 音频图像处理（调整亮度、对比度等）（暂不处理）

框架分析

源码分析

结果验证

参考文档

## 音频重采样

源码：3-12-resampling\_audio

也可以参考范例3-4-decode\_aac的处理

框架分析

源码分析

结果验证

参考文档

1. ffmpeg重采样resample包含AVAudioFifo(三) <http://blog.csdn.net/zhuweigangzwg/article/details/72624857>

框架分析

源码分析

### 结果验证

参考文档

## (14)视频转码H264转MP4

## (15)视频转码MP4转码H264

## 音频MP3转码AAC

先解码MP3，然后编码为AAC。

因为MP3的帧长和AAC的帧长不匹配。

## 音频AAC转码MP3（暂不处理）

Unknown encoder 'mp3'

后续的都需要H264编码支持，此时应该把前面的实验补充，并将代码风格统一。

(18)增加水印

(19)为视频添加/更换背景音乐

(20)两段视频做拼接，不同封装不同的解码器，输出指定为FLV封装，H264+AAC编码

(21)实现画中画效果并重新编码。

## 实现视频缩放，裁剪，水印等

FFMPEG官网给出了FFMPEG 滤镜使用的实例，它是将视频中的像素点替换成字符，然后从终端输出。我在该实例的基础上稍微的做了修改，使它能够保存滤镜处理过后的文件。在上代码之前先明白几个概念：

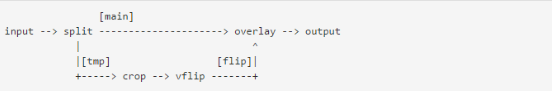
**Filter**:代表单个filter

**FilterPad**:代表一个filter的输入或输出端口，每个filter都可以有多个输入和多个输出，只有输出pad的filter称为source,只有输入pad的filter称为sink

**FilterLink**：若一个filter的输出pad和另一个filter的输入pad名字相同，即认为两个filter之间建立了link

**FilterChain**:代表一串相互连接的filters，除了source和sink外，要求每个filter的输入输出pad都有对应的输出和输入pad

经典示例：



图中的一系列操作共使用了四个filter，分别是

**splite**：将输入的流进行分裂复制，分两路输出。

**crop**：根据给定的参数，对视频进行裁剪

**vflip**：根据给定参数，对视频进行翻转等操作

**overlay**：将一路输入覆盖到另一路之上，合并输出为一路视频

下面上代码：

****[objc]**** [view plain](http://blog.csdn.net/li_wen01/article/details/62442162" \o "view plain" \t "http://blog.csdn.net/sunzhenjie9108/article/details/_blank) [copy](http://blog.csdn.net/li_wen01/article/details/62442162" \o "copy" \t "http://blog.csdn.net/sunzhenjie9108/article/details/_blank)

1. /\*=============================================================================
2. #     FileName: filter\_video.c
3. #         Desc: an example of ffmpeg fileter
4. #       Author: licaibiao
5. #   LastChange: 2017-03-16
6. =============================================================================\*/
7. #define \_XOPEN\_SOURCE 600 /\* for usleep \*/
8. #include <unistd.h>
10. #include "avcodec.h"
11. #include "avformat.h"
12. #include "avfiltergraph.h"
13. #include "avcodec.h"
14. #include "buffersink.h"
15. #include "buffersrc.h"
16. #include "opt.h"
18. #define SAVE\_FILE
20. **const** **charchar** \*filter\_descr = "scale=iw\*2:ih\*2";
21. **static** **AVFormatContext** \*fmt\_ctx;
22. **static** **AVCodecContext** \*dec\_ctx;
23. **AVFilterContext** \*buffersink\_ctx;
24. **AVFilterContext** \*buffersrc\_ctx;
25. **AVFilterGraph** \*filter\_graph;
26. **static** **int** video\_stream\_index = -1;
27. **static** int64\_t last\_pts = AV\_NOPTS\_VALUE;
29. **static** **int** open\_input\_file(**const** **charchar** \*filename)
30. {
31. **int** ret;
32. **AVCodec** \*dec;
34. **if** ((ret = avformat\_open\_input(&fmt\_ctx, filename, **NULL**, **NULL**)) < 0) {
35. av\_log(**NULL**, AV\_LOG\_ERROR, "Cannot open input file\n");
36. **return** ret;
37. }
39. **if** ((ret = avformat\_find\_stream\_info(fmt\_ctx, **NULL**)) < 0) {
40. av\_log(**NULL**, AV\_LOG\_ERROR, "Cannot find stream information\n");
41. **return** ret;
42. }
44. /\* select the video stream  判断流是否正常 \*/
45. ret = av\_find\_best\_stream(fmt\_ctx, AVMEDIA\_TYPE\_VIDEO, -1, -1, &dec, 0);
46. **if** (ret < 0) {
47. av\_log(**NULL**, AV\_LOG\_ERROR, "Cannot find a video stream in the input file\n");
48. **return** ret;
49. }
50. video\_stream\_index = ret;
51. dec\_ctx = fmt\_ctx->streams[video\_stream\_index]->codec;
52. av\_opt\_set\_int(dec\_ctx, "refcounted\_frames", 1, 0); /\* refcounted\_frames 帧引用计数 \*/
54. /\* init the video decoder \*/
55. **if** ((ret = avcodec\_open2(dec\_ctx, dec, **NULL**)) < 0) {
56. av\_log(**NULL**, AV\_LOG\_ERROR, "Cannot open video decoder\n");
57. **return** ret;
58. }
60. **return** 0;
61. }
63. **static** **int** init\_filters(**const** **charchar** \*filters\_descr)
64. {
65. **char** args[512];
66. **int** ret = 0;
67. **AVFilter** \*buffersrc  = avfilter\_get\_by\_name("buffer");     /\* 输入buffer filter \*/
68. **AVFilter** \*buffersink = avfilter\_get\_by\_name("buffersink"); /\* 输出buffer filter \*/
69. **AVFilterInOut** \*outputs = avfilter\_inout\_alloc();
70. **AVFilterInOut** \*inputs  = avfilter\_inout\_alloc();
71. AVRational time\_base = fmt\_ctx->streams[video\_stream\_index]->time\_base;   /\* 时间基数 \*/
73. #ifndef SAVE\_FILE
74. **enum** AVPixelFormat pix\_fmts[] = { AV\_PIX\_FMT\_GRAY8, AV\_PIX\_FMT\_NONE };
75. #else
76. **enum** AVPixelFormat pix\_fmts[] = { AV\_PIX\_FMT\_YUV420P, AV\_PIX\_FMT\_NONE };
77. #endif
79. filter\_graph = avfilter\_graph\_alloc();                     /\* 创建graph  \*/
80. **if** (!outputs || !inputs || !filter\_graph) {
81. ret = AVERROR(ENOMEM);
82. **goto** end;
83. }
85. /\* buffer video source: the decoded frames from the decoder will be inserted here. \*/
86. snprintf(args, **sizeof**(args),
87. "video\_size=%dx%d:pix\_fmt=%d:time\_base=%d/%d:pixel\_aspect=%d/%d",
88. dec\_ctx->width, dec\_ctx->height, dec\_ctx->pix\_fmt,
89. time\_base.num, time\_base.den,
90. dec\_ctx->sample\_aspect\_ratio.num, dec\_ctx->sample\_aspect\_ratio.den);
92. /\* 创建并向FilterGraph中添加一个Filter \*/
93. ret = avfilter\_graph\_create\_filter(&buffersrc\_ctx, buffersrc, "in",
94. args, **NULL**, filter\_graph);
95. **if** (ret < 0) {
96. av\_log(**NULL**, AV\_LOG\_ERROR, "Cannot create buffer source\n");
97. **goto** end;
98. }
100. /\* buffer video sink: to terminate the filter chain. \*/
101. ret = avfilter\_graph\_create\_filter(&buffersink\_ctx, buffersink, "out",
102. **NULL**, **NULL**, filter\_graph);
103. **if** (ret < 0) {
104. av\_log(**NULL**, AV\_LOG\_ERROR, "Cannot create buffer sink\n");
105. **goto** end;
106. }
108. /\* Set a binary option to an integer list. \*/
109. ret = av\_opt\_set\_int\_list(buffersink\_ctx, "pix\_fmts", pix\_fmts,
110. AV\_PIX\_FMT\_NONE, AV\_OPT\_SEARCH\_CHILDREN);
111. **if** (ret < 0) {
112. av\_log(**NULL**, AV\_LOG\_ERROR, "Cannot set output pixel format\n");
113. **goto** end;
114. }
116. /\*
117. \* Set the endpoints for the filter graph. The filter\_graph will
118. \* be linked to the graph described by filters\_descr.
119. \*/
121. /\*
122. \* The buffer source output must be connected to the input pad of
123. \* the first filter described by filters\_descr; since the first
124. \* filter input label is not specified, it is set to "in" by
125. \* default.
126. \*/
127. outputs->name       = av\_strdup("in");
128. outputs->filter\_ctx = buffersrc\_ctx;
129. outputs->pad\_idx    = 0;
130. outputs->next       = **NULL**;
132. /\*
133. \* The buffer sink input must be connected to the output pad of
134. \* the last filter described by filters\_descr; since the last
135. \* filter output label is not specified, it is set to "out" by
136. \* default.
137. \*/
138. inputs->name       = av\_strdup("out");
139. inputs->filter\_ctx = buffersink\_ctx;
140. inputs->pad\_idx    = 0;
141. inputs->next       = **NULL**;
143. /\* Add a graph described by a string to a graph \*/
144. **if** ((ret = avfilter\_graph\_parse\_ptr(filter\_graph, filters\_descr,
145. &inputs, &outputs, **NULL**)) < 0)
146. **goto** end;
148. /\* Check validity and configure all the links and formats in the graph \*/
149. **if** ((ret = avfilter\_graph\_config(filter\_graph, **NULL**)) < 0)
150. **goto** end;
152. end:
153. avfilter\_inout\_free(&inputs);
154. avfilter\_inout\_free(&outputs);
156. **return** ret;
157. }
159. #ifndef SAVE\_FILE
160. **static** **void** display\_frame(**const** **AVFrame** \*frame, AVRational time\_base)
161. {
162. **int** x, y;
163. **uint8\_t** \*p0, \*p;
164. int64\_t delay;
166. **if** (frame->pts != AV\_NOPTS\_VALUE) {
167. **if** (last\_pts != AV\_NOPTS\_VALUE) {
168. /\* sleep roughly the right amount of time;
169. \* usleep is in microseconds, just like AV\_TIME\_BASE. \*/
170. /\* 计算 pts 是用来把时间戳从一个时基调整到另外一个时基时候用的函数 \*/
171. delay = av\_rescale\_q(frame->pts - last\_pts,
172. time\_base, AV\_TIME\_BASE\_Q);
173. **if** (delay > 0 && delay < 1000000)
174. usleep(delay);
175. }
176. last\_pts = frame->pts;
177. }
179. /\* Trivial ASCII grayscale display. \*/
180. p0 = frame->data[0];
181. puts("\033c");
182. **for** (y = 0; y < frame->height; y++) {
183. p = p0;
184. **for** (x = 0; x < frame->width; x++)
185. putchar(" .-+#"[\*(p++) / 52]);
186. putchar('\n');
187. p0 += frame->linesize[0];
188. }
189. fflush(stdout);
190. }
191. #else
192. **FILEFILE** \* file\_fd;
193. **static** **void** write\_frame(**const** **AVFrame** \*frame)
194. {
195. **static** **int** printf\_flag = 0;
196. **if**(!printf\_flag){
197. printf\_flag = 1;
198. printf("frame widht=%d,frame height=%d\n",frame->width,frame->height);
200. **if**(frame->format==AV\_PIX\_FMT\_YUV420P){
201. printf("format is yuv420p\n");
202. }
203. **else**{
204. printf("formet is = %d \n",frame->format);
205. }
207. }
209. fwrite(frame->data[0],1,frame->width\*frame->height,file\_fd);
210. fwrite(frame->data[1],1,frame->width/2\*frame->height/2,file\_fd);
211. fwrite(frame->data[2],1,frame->width/2\*frame->height/2,file\_fd);
212. }
214. #endif
216. **int** main(**int** argc, **charchar** \*\*argv)
217. {
218. **int** ret;
219. AVPacket packet;
220. **AVFrame** \*frame = av\_frame\_alloc();
221. **AVFrame** \*filt\_frame = av\_frame\_alloc();
222. **int** got\_frame;
224. #ifdef SAVE\_FILE
225. file\_fd = fopen("test.yuv","wb+");
226. #endif
228. **if** (!frame || !filt\_frame) {
229. perror("Could not allocate frame");
230. exit(1);
231. }
232. **if** (argc != 2) {
233. fprintf(stderr, "Usage: %s file\n", argv[0]);
234. exit(1);
235. }
237. av\_register\_all();
238. avfilter\_register\_all();
240. **if** ((ret = open\_input\_file(argv[1])) < 0)
241. **goto** end;
242. **if** ((ret = init\_filters(filter\_descr)) < 0)
243. **goto** end;
245. /\* read all packets \*/
246. **while** (1) {
247. **if** ((ret = av\_read\_frame(fmt\_ctx, &packet)) < 0)
248. **break**;
250. **if** (packet.stream\_index == video\_stream\_index) {
251. got\_frame = 0;
252. ret = avcodec\_decode\_video2(dec\_ctx, frame, &got\_frame, &packet);
253. **if** (ret < 0) {
254. av\_log(**NULL**, AV\_LOG\_ERROR, "Error decoding video\n");
255. **break**;
256. }
258. **if** (got\_frame) {
259. frame->pts = av\_frame\_get\_best\_effort\_timestamp(frame);    /\* pts: Presentation Time Stamp \*/
261. /\* push the decoded frame into the filtergraph \*/
262. **if** (av\_buffersrc\_add\_frame\_flags(buffersrc\_ctx, frame, AV\_BUFFERSRC\_FLAG\_KEEP\_REF) < 0) {
263. av\_log(**NULL**, AV\_LOG\_ERROR, "Error while feeding the filtergraph\n");
264. **break**;
265. }
267. /\* pull filtered frames from the filtergraph \*/
268. **while** (1) {
269. ret = av\_buffersink\_get\_frame(buffersink\_ctx, filt\_frame);
270. **if** (ret == AVERROR(EAGAIN) || ret == AVERROR\_EOF)
271. **break**;
272. **if** (ret < 0)
273. **goto** end;
274. #ifndef SAVE\_FILE
275. display\_frame(filt\_frame, buffersink\_ctx->inputs[0]->time\_base);
276. #else
277. write\_frame(filt\_frame);
278. #endif
279. av\_frame\_unref(filt\_frame);
280. }
281. /\* Unreference all the buffers referenced by frame and reset the frame fields. \*/
282. av\_frame\_unref(frame);
283. }
284. }
285. av\_packet\_unref(&packet);
286. }
287. end:
288. avfilter\_graph\_free(&filter\_graph);
289. avcodec\_close(dec\_ctx);
290. avformat\_close\_input(&fmt\_ctx);
291. av\_frame\_free(&frame);
292. av\_frame\_free(&filt\_frame);
294. **if** (ret < 0 && ret != AVERROR\_EOF) {
295. fprintf(stderr, "Error occurred: %s\n", av\_err2str(ret));
296. exit(1);
297. }
298. #ifdef SAVE\_FILE
299. fclose(file\_fd);
300. #endif
301. exit(0);
302. }

该工程中，我的Makefile文件如下：

****[objc]**** [view plain](http://blog.csdn.net/li_wen01/article/details/62442162" \o "view plain" \t "http://blog.csdn.net/sunzhenjie9108/article/details/_blank) [copy](http://blog.csdn.net/li_wen01/article/details/62442162" \o "copy" \t "http://blog.csdn.net/sunzhenjie9108/article/details/_blank)

1. OUT\_APP      = test
2. INCLUDE\_PATH = /usr/local/include/
3. INCLUDE = -I$(INCLUDE\_PATH)libavutil/ -I$(INCLUDE\_PATH)libavdevice/ \
4. -I$(INCLUDE\_PATH)libavcodec/ -I$(INCLUDE\_PATH)libswresample \
5. -I$(INCLUDE\_PATH)libavfilter/ -I$(INCLUDE\_PATH)libavformat \
6. -I$(INCLUDE\_PATH)libswscale/
8. FFMPEG\_LIBS = -lavformat -lavutil -lavdevice -lavcodec -lswresample -lavfilter -lswscale
9. SDL\_LIBS    =
10. LIBS        = $(FFMPEG\_LIBS)$(SDL\_LIBS)
12. COMPILE\_OPTS = $(INCLUDE)
13. C            = c
14. OBJ          = o
15. C\_COMPILER   = cc
16. C\_FLAGS      = $(COMPILE\_OPTS) $(CPPFLAGS) $(CFLAGS)
18. LINK         = cc -o
19. LINK\_OPTS    = -lz -lm  -lpthread
20. LINK\_OBJ     = test.o
22. .$(C).$(OBJ):
23. $(C\_COMPILER) -c $(C\_FLAGS) $<

26. $(OUT\_APP): $(LINK\_OBJ)
27. $(LINK)$@  $(LINK\_OBJ)  $(LIBS) $(LINK\_OPTS)
29. clean:
30. -rm -**rf** \*.$(OBJ) $(OUT\_APP) **core** \*.core \*~ \*yuv

运行结果如下：

****[objc]**** [view plain](http://blog.csdn.net/li_wen01/article/details/62442162" \o "view plain" \t "http://blog.csdn.net/sunzhenjie9108/article/details/_blank) [copy](http://blog.csdn.net/li_wen01/article/details/62442162" \o "copy" \t "http://blog.csdn.net/sunzhenjie9108/article/details/_blank)

1. licaibiao**@ubuntu**:~/test/FFMPEG/filter$ ls
2. Makefile  school.flv  test  test.c  test.o
3. licaibiao**@ubuntu**:~/test/FFMPEG/filter$ ./test school.flv
4. [flv @ 0x12c16c0] video stream discovered after head already parsed
5. [flv @ 0x12c16c0] audio stream discovered after head already parsed
6. frame widht=1024,frame height=576
7. format is yuv420p
8. licaibiao**@ubuntu**:~/test/FFMPEG/filter$ ls
9. Makefile  school.flv  test  test.c  test.o  test.yuv

在这里，我打印出来了输出视频的格式和图片的长和宽，该实例生成的是一个YUV420 格式的视频，使用YUV播放器播放视频的时候，需要设置正确的视频长度和宽度。在代码中通过设置enum AVPixelFormat pix\_fmts[] = { AV\_PIX\_FMT\_YUV420P, AV\_PIX\_FMT\_NONE };来设置输出格式。

过滤器的参数设置是通过const char \*filter\_descr = "scale=iw\*2:ih\*2"; 来设置。它表示将视频的长和框都拉伸到原来的两倍。具体的filter参数可以通过命令：ffmpeg -filters 来查询。结果如下：

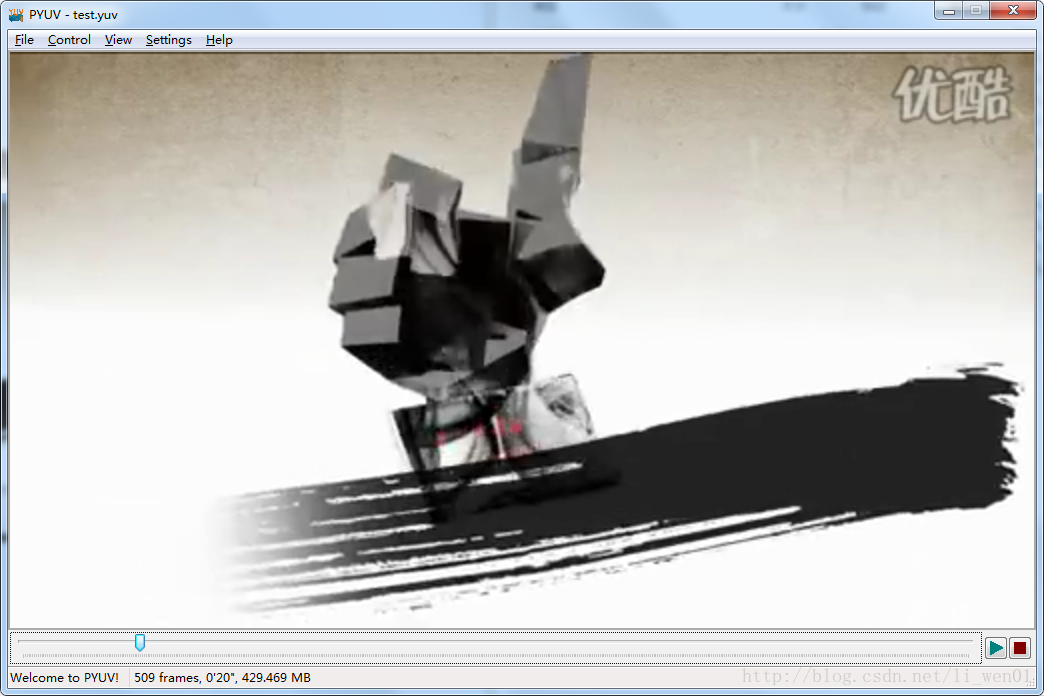
****[objc]**** [view plain](http://blog.csdn.net/li_wen01/article/details/62442162" \o "view plain" \t "http://blog.csdn.net/sunzhenjie9108/article/details/_blank) [copy](http://blog.csdn.net/li_wen01/article/details/62442162" \o "copy" \t "http://blog.csdn.net/sunzhenjie9108/article/details/_blank)

1. Filters:
2. T.. = Timeline support
3. .S. = Slice threading
4. ..C = Command support
5. A = Audio input/output
6. V = Video input/output
7. N = Dynamic number and/or type of input/output
8. | = Source or sink filter
9. ... abench            A->A       Benchmark part of a filtergraph.
10. ... acompressor       A->A       Audio compressor.
11. ... acrossfade        AA->A      Cross fade two input audio streams.
12. ... acrusher          A->A       Reduce audio bit resolution.
13. .............................................................................

在上面的代码中，我们设置的是将图片拉升到原来图像的两倍，其显示效果如下，可能是截图的问题，这里看好像没有拉伸到两倍。



原图



拉伸后

在上面的代码中，我们设置的是：

const char \*filter\_descr = "scale=iw\*2:ih\*2";   iw 表示输入视频的宽，ih表示输入视频的高。可以任意比例的缩放视频。这里\*2 表示放大两倍,如果是/2表示缩小两倍。

视频缩放还可以直接设置：

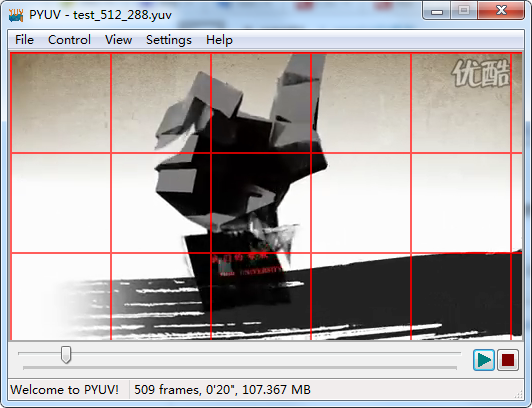
const char \*filter\_descr = "scale=320:240"; 设置视频输出宽为320，高位240，当然也是可以随意的设置其他的参数。

视频的裁剪可以设置为：

 const char \*filter\_descr = "crop=320:240:0:0";   具体含义是 crop=width:height:x:y，其中 width 和 height 表示裁剪后的尺寸，x:y 表示裁剪区域的左上角坐标。

视频添加一个网格水印可以设置为：

const char \*filter\_descr = "drawgrid=width=100:height=100:thickness=2:color=red@0.5";    具体含义是 width 和 height 表示添加网格的宽和高，thickness表示网格的线宽，color表示颜色 。其效果如下：



更多filter参数的使用，可以直接参考ffmpeg 的官方文档：[http://www.ffmpeg.org/ffmpeg-filters.html](http://www.ffmpeg.org/ffmpeg-filters.html" \t "http://blog.csdn.net/sunzhenjie9108/article/details/_blank)

## 自定义filter

ffmpeg filter

### 参考文档

ffmpeg filter过滤器 基础实例及全面解析

## 转行格式

一种是annexb模式，传统模式，有startcode，SPS和PPS是在ES中；另一种是mp4模式，一般mp4、mkv、avi会没有startcode，SPS和PPS以及其它信息被封装在container中，每一个frame前面是这个frame的长度，很多解码器只支持annexb这种模式，因此需要将mp4做转换；在ffmpeg中用h264\_mp4toannexb\_filter可以做转换；所以需要使用-bsf h264\_mp4toannexb来进行转换；

# 播放器设计

## 框架

## 所涉及的技术

# Ffmpeg.c分析

## 前沿

接触ffmpeg2年多，系统使用大概半年左右。ffmpeg命令行方式作为“本地文件”处理足够，但要更加精准的控制还是需要借用SDK的方式，好在ffmpeg的example中提供了大量的简单例子。好了废话不多说，开始干活。

ffmpeg版本3.2

## 主体结构分析

简单记录讲解比较重要的函数

main：主函数入口

|--\*\_register\_all:注册各种模块(ffmpeg驱动模块化思维的最佳体现);

|--avformat\_network\_init：初始化网络环境

|--ffmpeg\_parse\_options:<1>解析并设置命令行的输入参数;<2>打开输入输出“文件”;<3>初始化complex filters;

|--transcode：转码/流重封装入口

|--transcode\_init：初始化输入输出的codec;

|--init\_input\_stream

|--avcodec\_open2：打开解码器

|--init\_output\_stream

|--1.0.编码

|--1.1.init\_output\_stream\_encode:设置编码参数的函数，该函数具有很强的参考意义

|--1.2.avcodec\_open2：打开编码器

|--1.3.avcodec\_parameters\_from\_context

|--1.4.avcodec\_copy\_context

|--2.0.重封装

|--2.1.init\_output\_stream\_streamcopy：重封装函数

|--2.2.avcodec\_parameters\_to\_context

|--init\_output\_bsfs：初始化bsfs（bsfs相关函数接口有新的变化，可以参考更新

|--transcode\_step

|--configure\_filtergraph:初始化simple filters，包括构建graph

|--transcode\_from\_filter:

|--avfilter\_graph\_request\_oldest

|--reap\_filters(0)

|--av\_buffersink\_get\_frame\_flags (AV\_BUFFERSINK\_FLAG\_NO\_REQUEST)

|--do\_video\_out

|--process\_input：转码重要步骤

|--get\_input\_packet：获取解码帧数据

|--av\_read\_frame：读取一帧数据

|--process\_input\_packet

|--decode\_video

|--decode:(avcodec\_send\_packet/avcodec\_receive\_frame)：解码关键函数，新的接口

|--av\_frame\_get\_best\_effort\_timestamp

|--send\_frame\_to\_filters：将解码帧送往graph入口

|--ifilter\_send\_frame

|--reap\_filters(1)

|--av\_buffersrc\_add\_frame\_flags (AV\_BUFFERSRC\_FLAG\_PUSH)

|--reap\_filters(0)：获取帧并封装输出

HAVE\_PTHREADS：将为对读输入流单独创建线程，读取的帧数据放入队列;当解码需要数据时，再从队列中取出数据;

## reap\_filters分析

reap\_filters

|--av\_buffersink\_get\_frame\_flags (AV\_BUFFERSINK\_FLAG\_NO\_REQUEST)：从graph输出口获取filter数据

|--do\_video\_out

|--编码:(avcodec\_send\_frame/avcodec\_receive\_packet) ：编码帧

|--av\_packet\_rescale\_ts(&pkt, enc->time\_base, ost->mux\_timebase);

|--output\_packet

|--bitstream filters:(av\_bsf\_send\_packet/av\_bsf\_receive\_packet)：bsfs处理

|--write\_packet：封装发送

|--av\_packet\_rescale\_ts(pkt, ost->mux\_timebase, ost->st->time\_base);

|--av\_interleaved\_write\_frame(s, pkt);

## 关于ffmpeg例子的一些坑

1.编码后的fps和tbc，总是源的两倍？

2.输出为mpegts时，且设置muxrate，阻塞？

3.当流出现丢包时，会卡住？

4.多线程使用，出现段错误？

5.当不使用muxrate时，pat/pmt/pcr间隔严重失准？

6.如何偏移pts和dts?

## ffmpeg.c函数结构简单分析（画图）

原创 2014年10月04日 00:12:40 标签：ffmpeg /函数 /分析 25233

前一阵子研究转码的时候看了FFmpeg的源代码。由于ffmpeg.c的代码相对比较长，而且其中有相当一部分是AVFilter有关的代码（这一部分一直不太熟），因此之前学习FFmpeg的时候一直也没有好好看一下其源代码。最近正好看了看AVFilter的知识，顺便就看了下FFmpeg的源代码，在这里画图理一下它的结构。目前好多地方还没有弄明白，等到以后慢慢完善了。

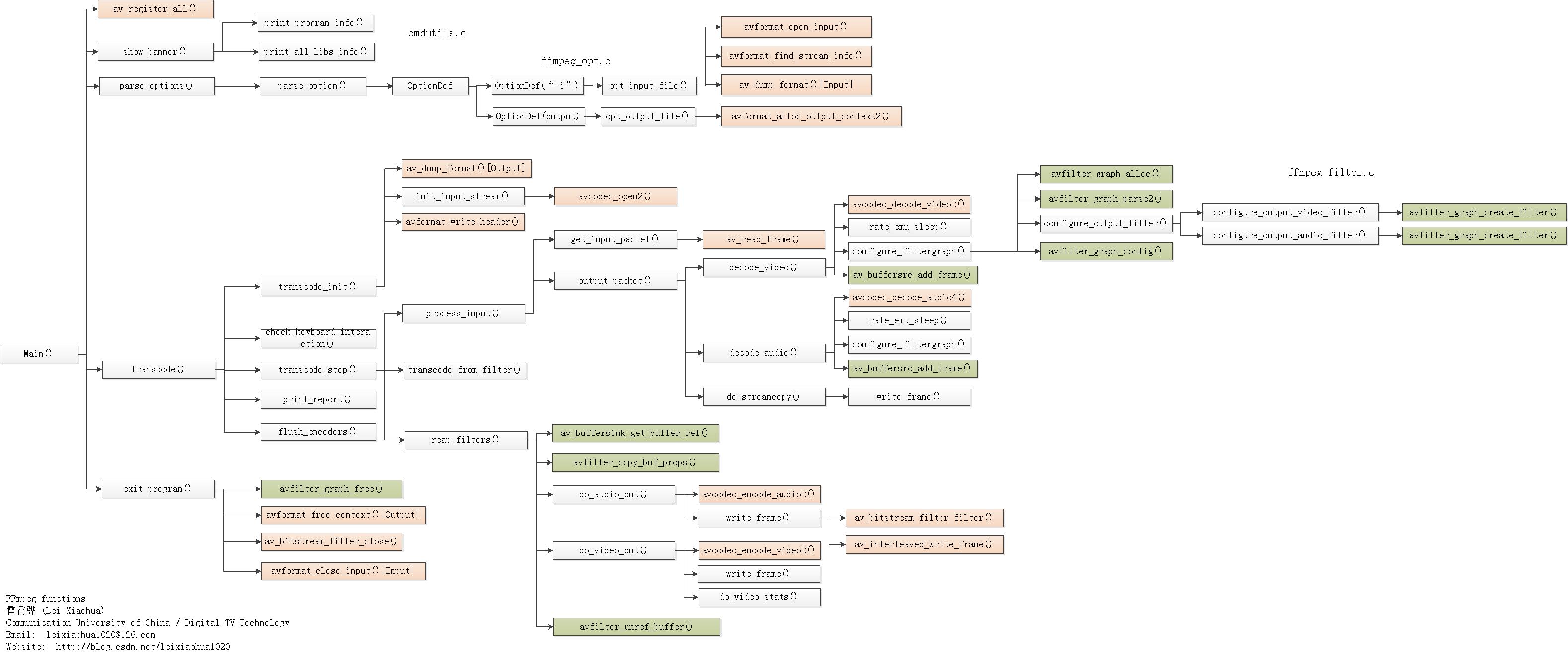
先说明一下自己画的结构图的规则：图中仅画出了比较重要的函数之间的调用关系。粉红色的函数是FFmpeg编解码类库（libavcodec，libavformat等）的API。绿色的函数是FFmpeg的libavfilter的API。其他不算很重要的函数就不再列出了。

PS：有一部分代码可能和ffmpeg.c有一些出入。因为本文使用的ffmpeg.c的代码是移植到VC之后的代码。

在看ffmpeg.c的代码之前，最好先看一下简单的代码了解FFmpeg解码，编码的关键API：

### 函数调用结构图

FFmpeg的总体函数调用结构图如下图所示



下文将会对主要函数分别解析。

main()

### main()是FFmpeg的主函数

调用了如下函数

av\_register\_all()：注册所有编码器和解码器。

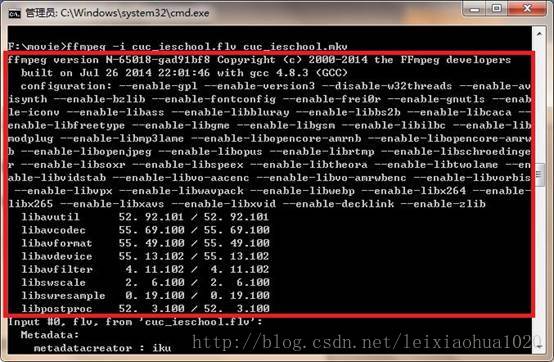
show\_banner()：打印输出FFmpeg版本信息（编译时间，编译选项，类库信息等）。

parse\_options()：解析输入的命令。

transcode()：转码。

exit\_progam()：退出和清理。

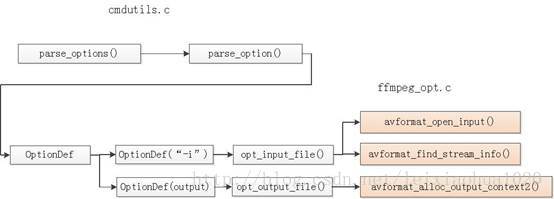
下图红框中的内容即为show\_banner()的输出结果。



### parse\_options()

parse\_options()解析全部输入选项。即将输入命令“ffmpeg -i xxx.mpg -vcodec libx264 yyy.mkv”中的“-i”，“-vcodec”这样的命令解析出来。其函数调用结构如下图所示。

注：定义位于cmdutils.c中。



调用了如下函数：

parse\_option()：解析一个输入选项。具体的解析步骤不再赘述。parse\_options()会循环调用parse\_option()直到所有选项解析完毕。FFmpeg的每一个选项信息存储在一个OptionDef结构体中。定义如下：

typedef struct OptionDef {

const char \*name;

int flags;

#define HAS\_ARG 0x0001

#define OPT\_BOOL 0x0002

#define OPT\_EXPERT 0x0004

#define OPT\_STRING 0x0008

#define OPT\_VIDEO 0x0010

#define OPT\_AUDIO 0x0020

#define OPT\_INT 0x0080

#define OPT\_FLOAT 0x0100

#define OPT\_SUBTITLE 0x0200

#define OPT\_INT64 0x0400

#define OPT\_EXIT 0x0800

#define OPT\_DATA 0x1000

#define OPT\_PERFILE 0x2000 /\* the option is per-file (currently ffmpeg-only).

implied by OPT\_OFFSET or OPT\_SPEC \*/

#define OPT\_OFFSET 0x4000 /\* option is specified as an offset in a passed optctx \*/

#define OPT\_SPEC 0x8000 /\* option is to be stored in an array of SpecifierOpt.

Implies OPT\_OFFSET. Next element after the offset is

an int containing element count in the array. \*/

#define OPT\_TIME 0x10000

#define OPT\_DOUBLE 0x20000

union {

void \*dst\_ptr;

int (\*func\_arg)(void \*, const char \*, const char \*);

size\_t off;

} u;

const char \*help;

const char \*argname;

} OptionDef;

其中的重要字段：

name：用于存储选项的名称。例如“i”，“f”，“codec”等等。

flags：存储选项值的类型。例如：HAS\_ARG（包含选项值），OPT\_STRING（选项值为字符串类型），OPT\_TIME（选项值为时间类型。

u：存储该选项的处理函数。

help：选项的说明信息。

FFmpeg使用一个名称为options，类型为OptionDef的数组存储所有的选项。有一部分通用选项存储在cmdutils\_common\_opts.h中。cmdutils\_common\_opts.h内容如下：

{ "L" , OPT\_EXIT, {(void\*)show\_license}, "show license" },

{ "h" , OPT\_EXIT, {(void\*) show\_help}, "show help", "topic" },

{ "?" , OPT\_EXIT, {(void\*)show\_help}, "show help", "topic" },

{ "help" , OPT\_EXIT, {(void\*)show\_help}, "show help", "topic" },

{ "-help" , OPT\_EXIT, {(void\*)show\_help}, "show help", "topic" },

{ "version" , OPT\_EXIT, {(void\*)show\_version}, "show version" },

{ "formats" , OPT\_EXIT, {(void\*)show\_formats }, "show available formats" },

{ "codecs" , OPT\_EXIT, {(void\*)show\_codecs }, "show available codecs" },

{ "decoders" , OPT\_EXIT, {(void\*)show\_decoders }, "show available decoders" },

{ "encoders" , OPT\_EXIT, {(void\*)show\_encoders }, "show available encoders" },

{ "bsfs" , OPT\_EXIT, {(void\*)show\_bsfs }, "show available bit stream filters" },

{ "protocols" , OPT\_EXIT, {(void\*)show\_protocols}, "show available protocols" },

{ "filters" , OPT\_EXIT, {(void\*)show\_filters }, "show available filters" },

{ "pix\_fmts" , OPT\_EXIT, {(void\*)show\_pix\_fmts }, "show available pixel formats" },

{ "layouts" , OPT\_EXIT, {(void\*)show\_layouts }, "show standard channel layouts" },

{ "sample\_fmts", OPT\_EXIT, {(void\*)show\_sample\_fmts }, "show available audio sample formats" },

{ "loglevel" , HAS\_ARG, {(void\*)opt\_loglevel}, "set libav\* logging level", "loglevel" },

{ "v", HAS\_ARG, {(void\*)opt\_loglevel}, "set libav\* logging level", "loglevel" },

{ "debug" , HAS\_ARG, {(void\*)opt\_codec\_debug}, "set debug flags", "flags" },

{ "fdebug" , HAS\_ARG, {(void\*)opt\_codec\_debug}, "set debug flags", "flags" },

{ "report" , 0, {(void\*)opt\_report}, "generate a report" },

{ "max\_alloc" , HAS\_ARG, {(void\*) opt\_max\_alloc}, "set maximum size of a single allocated block", "bytes" },

{ "cpuflags" , HAS\_ARG | OPT\_EXPERT, {(void\*) opt\_cpuflags}, "force specific cpu flags", "flags" },

options数组的定义位于ffmpeg\_opt.c中：

const OptionDef options[] = {

/\* main options \*/

#include "cmdutils\_common\_opts.h"//包含了cmdutils\_common\_opts.h中的选项

{ "f", HAS\_ARG | OPT\_STRING | OPT\_OFFSET, { (void\*)OFFSET(format) },

"force format", "fmt" },

{ "i", HAS\_ARG | OPT\_PERFILE, { (void\*) opt\_input\_file },

"input file name", "filename" },

{ "y", OPT\_BOOL, { &file\_overwrite },

"overwrite output files" },

{ "n", OPT\_BOOL, { &no\_file\_overwrite },

"do not overwrite output files" },

{ "c", HAS\_ARG | OPT\_STRING | OPT\_SPEC,{ (void\*) OFFSET(codec\_names) },

"codec name", "codec" },

{ "codec", HAS\_ARG | OPT\_STRING | OPT\_SPEC,{(void\*) OFFSET(codec\_names) },

"codec name", "codec" },

{ "pre", HAS\_ARG | OPT\_STRING | OPT\_SPEC,{ (void\*) OFFSET(presets) },

"preset name", "preset" },

{ "map", HAS\_ARG | OPT\_EXPERT | OPT\_PERFILE, { (void\*) opt\_map },

"set input stream mapping",

"[-]input\_file\_id[:stream\_specifier][,sync\_file\_id[:stream\_specifier]]" },

{ "map\_channel", HAS\_ARG | OPT\_EXPERT | OPT\_PERFILE, {(void\*)opt\_map\_channel },

"map an audio channel from one stream to another", "file.stream.channel[:syncfile.syncstream]" },

{ "map\_metadata", HAS\_ARG | OPT\_STRING | OPT\_SPEC,{ (void\*)OFFSET(metadata\_map) },

"set metadata information of outfile from infile",

"outfile[,metadata]:infile[,metadata]" },

{ "map\_chapters", HAS\_ARG | OPT\_INT | OPT\_EXPERT | OPT\_OFFSET, { (void\*) OFFSET(chapters\_input\_file) },

"set chapters mapping", "input\_file\_index" },

{ "t", HAS\_ARG | OPT\_TIME | OPT\_OFFSET,{(void\*) OFFSET(recording\_time) },

"record or transcode \"duration\" seconds of audio/video",

"duration" },

{ "fs",HAS\_ARG | OPT\_INT64 | OPT\_OFFSET, { (void\*) OFFSET(limit\_filesize) },

"set the limit file size in bytes", "limit\_size" },

{ "ss",HAS\_ARG | OPT\_TIME | OPT\_OFFSET,{ (void\*) OFFSET(start\_time) },

"set the start time offset", "time\_off" },

…//选项太多，不一一列出

};

在这里，例举一个选项的OptionDef结构体：输入

[cpp] view plain copy

{ "i",HAS\_ARG | OPT\_PERFILE, { (void\*) opt\_input\_file }, "input file name", "filename" }

在这个结构体中，可以看出选项的名称为“i”，选项包含选项值（HAS\_ARG），选项的处理函数是opt\_input\_file()，选项的说明是“input file name”。下面可以详细看一下选项的处理函数opt\_input\_file()。该函数的定义位于ffmpeg\_opt.c文件中。可以看出，调用了avformat\_alloc\_context()初始化了AVFormatContext结构体，调用了avformat\_open\_input()函数打开了“-i”选项指定的文件。此外，调用了avformat\_find\_stream\_info()等完成了一些初始化操作。此外，调用了av\_dump\_format()打印输出输入文件信息。

[cpp] view plain copy

static int opt\_input\_file(void \*optctx, const char \*opt, const char \*filename)

{

//略…

/\* open the input file with generic avformat function \*/

err = avformat\_open\_input(&ic, filename, file\_iformat, &format\_opts);

if (err < 0) {

print\_error(filename, err);

exit(1);

}

//略…

/\* Set AVCodecContext options for avformat\_find\_stream\_info \*/

opts = setup\_find\_stream\_info\_opts(ic, codec\_opts);

orig\_nb\_streams = ic->nb\_streams;

/\* If not enough info to get the stream parameters, we decode the

first frames to get it. (used in mpeg case for example) \*/

ret = avformat\_find\_stream\_info(ic, opts);

if (ret < 0) {

av\_log(NULL, AV\_LOG\_FATAL, "%s: could not find codec parameters\n", filename);

avformat\_close\_input(&ic);

exit(1);

}

//略…

/\* dump the file content \*/

av\_dump\_format(ic, nb\_input\_files, filename, 0);

//略…

return 0;

}

再例举一个输出文件处理函数opt\_output\_file()。这里需要注意，输出文件的处理并不包含在OptionDef类型的数组options中。因为FFmpeg中指定输出文件时并不包含选项名称，这是一个比较特殊的地方。一般的选项格式是“-名称 值”，例如指定输入文件的时候，选项格式是“-i xxx.flv”。而指定输出文件的时候，直接指定“值”即可，这是新手可能容易搞混的地方。

例如，最简单的转码命令如下（输出文件前面不包含选项）：

[plain] view plain copy

ffmpeg -i xxx.mpg xxx.mkv

而不是

[plain] view plain copy

ffmpeg -i xxx.mpeg -o xxx.mkv

下面简单看一下opt\_output\_file()函数的定义。该函数的定义同样位于ffmpeg\_opt.c文件中。这个函数的定义特别长，完成了输出视频的初始化工作。在这里就不列出代码了。该函数首先调用avformat\_alloc\_output\_context2()初始化AVFormatContext结构体。而后根据媒体类型的不同，分别调用new\_video\_stream()，new\_audio\_stream()，new\_subtitle\_stream()等创建不同的AVStream。实际上上述的几个创建AVStream的函数调用了new\_output\_stream()。而new\_output\_stream()又调用了FFmpeg类库的API函数avformat\_new\_stream()。

[cpp] view plain copy

void opt\_output\_file(void \*optctx, const char \*filename)

{

//略…

err = avformat\_alloc\_output\_context2(&oc, NULL, o->format, filename);

if (!oc) {

print\_error(filename, err);

exit(1);

}

//略…

new\_video\_stream();

…

new\_audio\_stream();

…

new\_subtitle\_stream ();

//略…

}

transcode()

transcode()的功能是转码。其函数调用结构如下图所示。

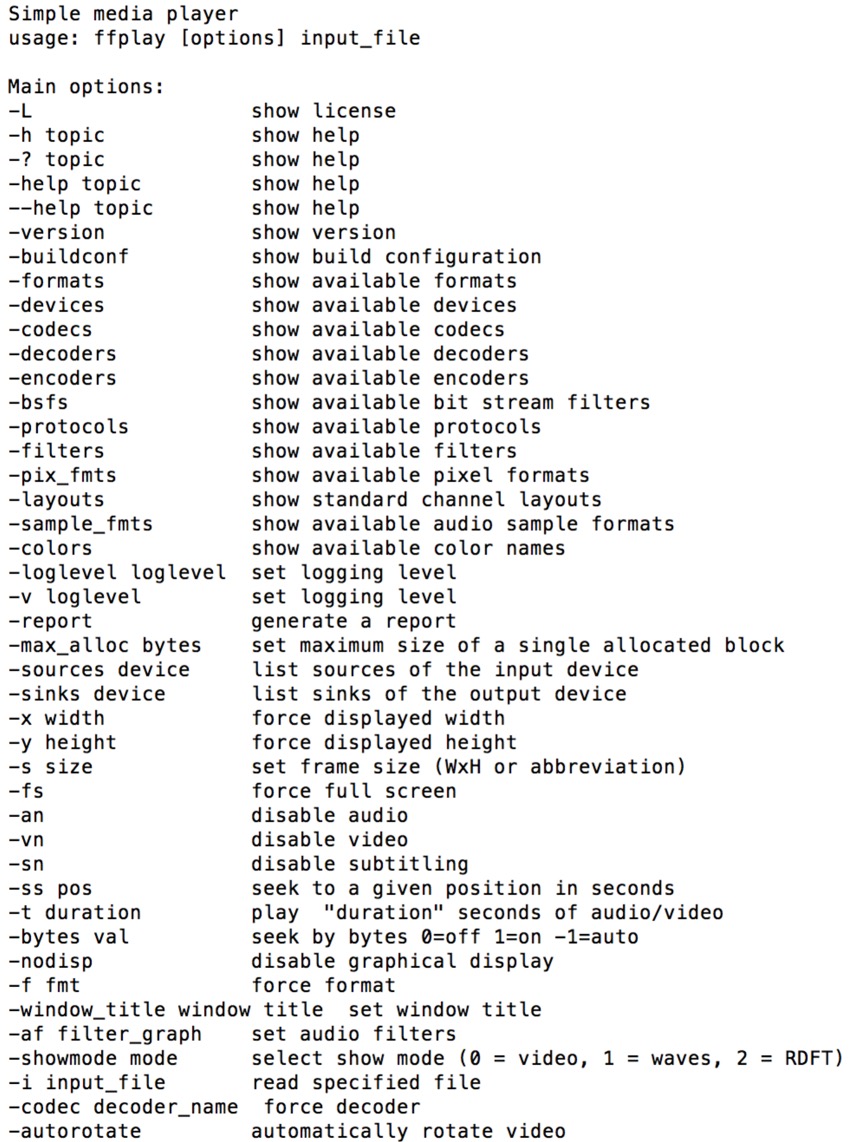
## 参考文档

1. ffmpeg.c函数结构简单分析（画图） <http://blog.csdn.net/leixiaohua1020/article/details/39760711>
2. [FFMPEG-代码分析]ffmpeg.c http://blog.csdn.net/weixin\_35804181/article/details/70215118

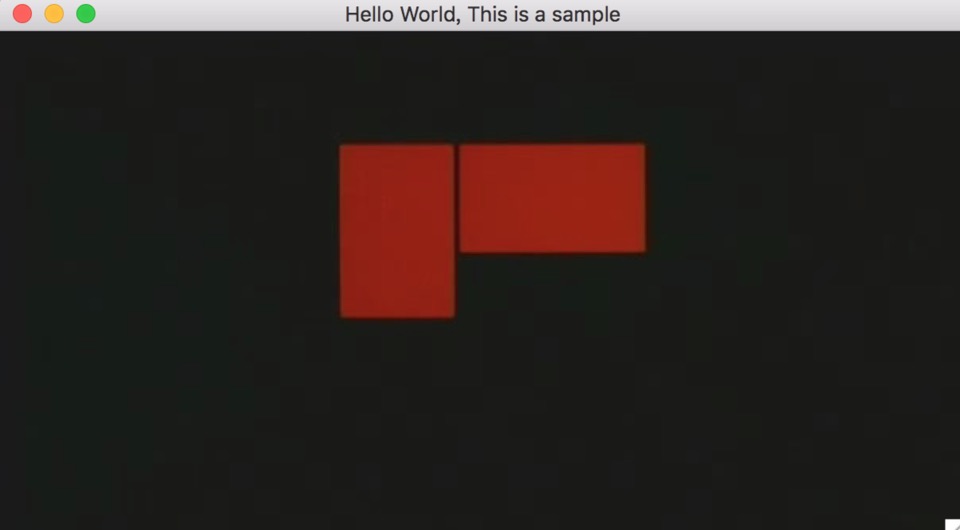
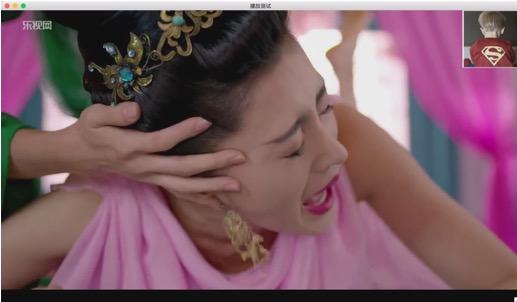
# 参考

## FFmpeg使用手册 - ffplay 的常用命令

在编译FFmpeg源代码时，如果系统中包含了SDL-1.2版本时，会默认将ffplay编译生成出来，如果不包含SDL-1.2或者版本不是SDL-1.2时，无法生成ffplay文件，所以，生成如果想使用ffplay进行流媒体播放测试，是需要安装SDL-1.2的。  
  
通常使用ffplay作为播放器，其实ffplay不但可以做播放器，同样可以作为很多图像化音视频数据的分析根据，通过ffplay可以看到视频图像的运动估计方向，音频数据的波形等，在本节将会有更多的参数进行介绍并举例。

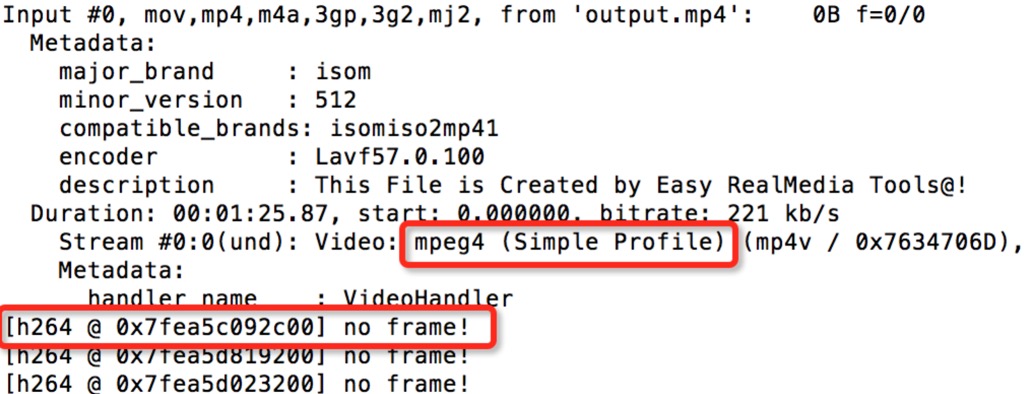
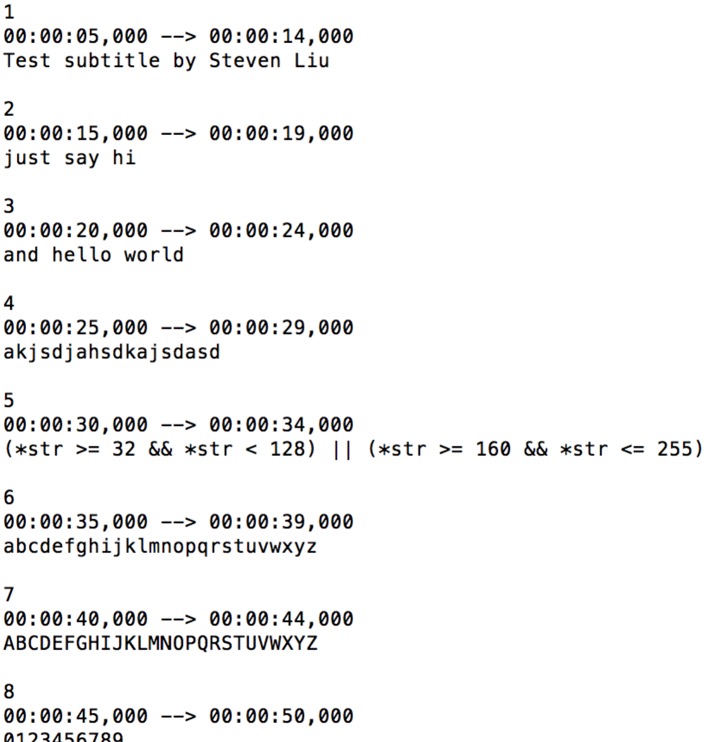
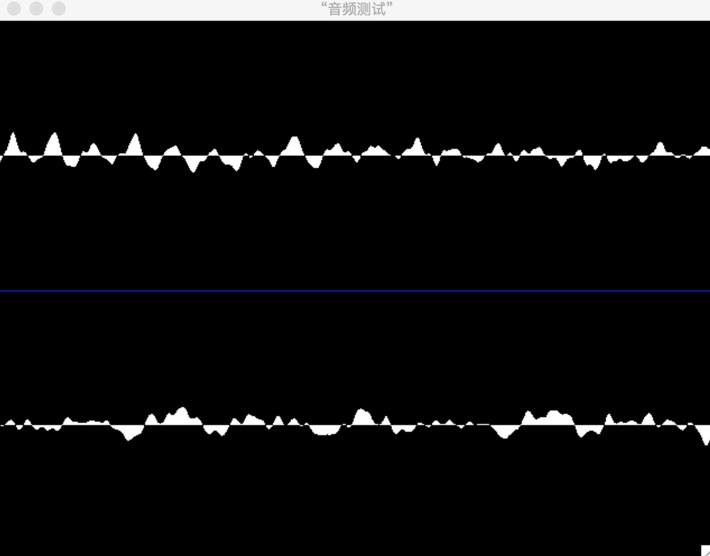
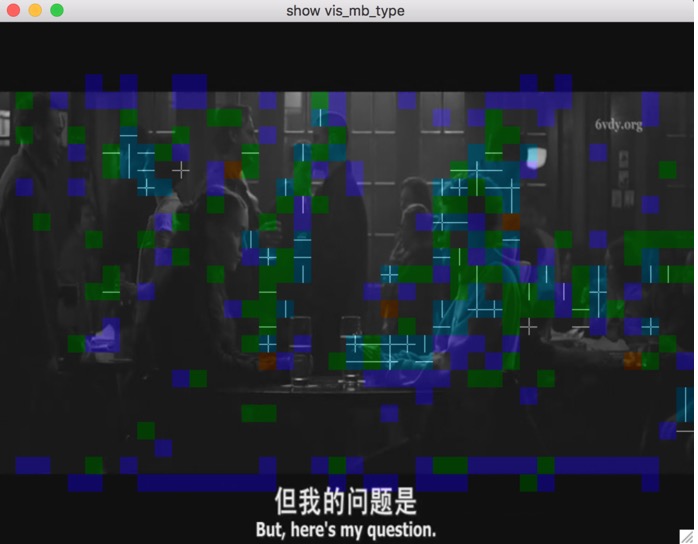
6.1 ffplay 常用参数  
ffplay不仅仅是播放器，同时也是测试ffmpeg的codec引擎，format引擎，以及filter引擎的工具，并且也可以做可视化的媒体参数分析，可以通过ffplay --help 进行查看：  
  
如上图，大多数是前面已经介绍过的参数，还有些是未介绍过的

|  |  |
| --- | --- |
| 参数 | 说明 |
| X | 强制设置视频显示窗口的宽度 |
| y | 强制设置视频显示窗口的高度 |
| S | 设置视频显示的宽高 |
| fs | 强制全屏显示 |
| an | 屏蔽音频 |
| vn | 屏蔽视频 |
| Sn | 屏蔽字幕 |
| ss | 根据设置的秒进行定位拖动 |
| t | 设置播放视频/音频长度 |
| Bytes | 设置定位拖动的策略，0为不可拖动，1为可拖动，-1为自动 |
| Nodisp | 关闭图形化显示窗口 |
| f | 强制使用设置的格式进行解析 |
| window\_title | 设置显示窗口的标题 |
| af | 设置音频的滤镜 |
| Codec | 强制使用设置的codec进行解码 |
| autorotate | 自动旋转视频 |

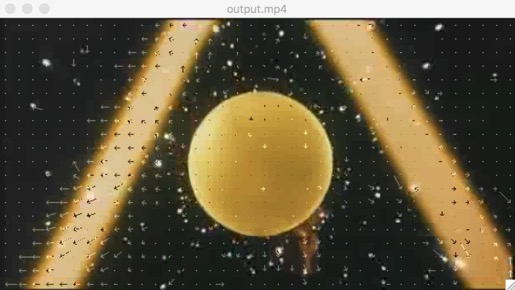
常见参数可以手动进行尝试：  
如果希望从视频的第30秒开始播放，播放10秒钟的文件，则可以使用如下命令  
#ffplay -ss 30 -t 10 input.mp4  
如果希望视频播放时播放器的窗口显示标题为自定义标题，则使用如下命令  
ffplay -window\_title "Hello World, This is a sample" output.mp4  
显示窗口如下图：  
  
例如使用ffplay打开网络直播流，可以使用命令  
# ffplay -window\_title "播放测试"  rtmp://up.v.test.com/live/stream  
  
如图播放的视频为实时网络直播视频流。  
  
  
根据上图可以看到播放器播放的窗口标题已经显示为自定义设置的内容。  
基本参数介绍完毕，下面进一步介绍ffplay的高级一些的参数。

6.2 ffplay高级参数  
通过使用ffplay –help参数看到帮助信息比较多，其中包含了高级参数介绍，下面详细介绍一下

|  |  |
| --- | --- |
| 参数 | 说明 |
| ast | 设置将要播放的音频流 |
| vst | 设置将要播放的视频流 |
| sst | 设置将要播放的字幕流 |
| Stats | 输出多媒体播放状态 |
| Fast | 非标准化规范的多媒体兼容优化 |
| sync | 音视频同步设置可设置根据音频视频进行参考，视频时间参考，或者外部扩展时间进行参考 |
| autoexit | 多媒体播放完毕自动退出ffplay，ffplay默认播放完毕不退出播放器 |
| exitonkeydown | 当有按键按下事件产生时退出ffplay |
| exitonmousedown | 当有鼠标按键事件产生时退出ffplay |
| loop | 设置多媒体文件循环播放次数 |
| framedrop | 当CPU资源占用过高时，自动丢帧 |
| infbuf | 设置无极限的播放器buffer，这个选项常见于实时流媒体播放场景 |
| vf | 视频滤镜设置 |
| acodec | 强制使用设置的音频解码器 |
| vcodec | 强制使用设置的视频解码器 |
| scodec | 强制使用设置的字幕解码器 |

下面根据这些参数与前面介绍过的一些参数进行组合：  
例如从20秒播放一个视频，播放时长为10秒钟，播放完成后自动退出ffplay，播放器的窗口标题为”Hello World”，为了确认播放时长正确，可以通过系统命令time查看命令运行时长  
#time ffplay -window\_title "Hello World" -ss 20 -t 10 -autoexit output.mp4  
该命令执行完毕之后输出如下：  
real 0m10.783s  
user 0m8.401s  
sys 0m0.915s  
  
例如强制使用H264解码器解码mpeg4的视频，将会报错：  
# ffplay -vcodec h264 output.mp4  
  
使用h264的解码器解码mpeg4时会得到no frame的错误，视频也解析不出来。  
前面举过的例子中，看到比较多的是单节目的流，下面举一个多节目的流，常见于\*\*行业的视频：  
  
当视频流中出现多个Program时，播放Program与常规的播放方式则有所不同，需要指定对应的流，可以通过vst、ast、sst参数指定，例如希望播放Program 13中的音视频流，则通过如下命令行指定：  
#ffplay -vst 4 -ast 5 ~/Movies/movie/ChinaTV-11.ts  
播放效果如图：  
  
通过Program 13中的信息可以看到该流名称为service\_name对应的值是CCTV 9，而指定音视频流播放之后播放出来的图像也能够与之对应。  
如果使用ffplay播放视频时希望加载字幕文件，则可以通过加载ASS或者SRT字幕文件来解决，下面举一个加载SRT字幕的例子，首先编辑SRT字幕文件，内容如下：  
  
然后通过filter将字幕文件加载如播放数据中，使用命令  
# ffplay -window\_title "Test Movie" -vf "subtitles=input.srt" output.mp4  
通过这条命令可以看到播放的效果如下  
  
  
  
  
  
  
6.3 ffplay的数据可视化分析应用  
使用ffplay除了可以播放视频流媒体文件之外，还可以作为可视化的视频流媒体分析工具，例如当播放音频文件时，不确定文件的声音是否正常，噪声数据等分析，可以直接使用ffplay播放音频文件，播放的时候将会把解码后的音频数据以音频波形显示出来：  
#ffplay -showmode 1 output.mp3  
  
从图中可以看到，音频的播放时的波形可以通过振幅显示出来，可以用来看到音频的播放情况。  
例如当播放视频时体验解码器是如何解码每个宏块的，可以使用命令  
# ffplay -debug vis\_mb\_type -window\_title "show vis\_mb\_type" -ss 20 -t 10 -autoexit output.mp4  
显示窗口内容  
  
在输出的视频信息中，可以看到不同颜色的方块，下面来说明一下这些颜色代表着什么信息：

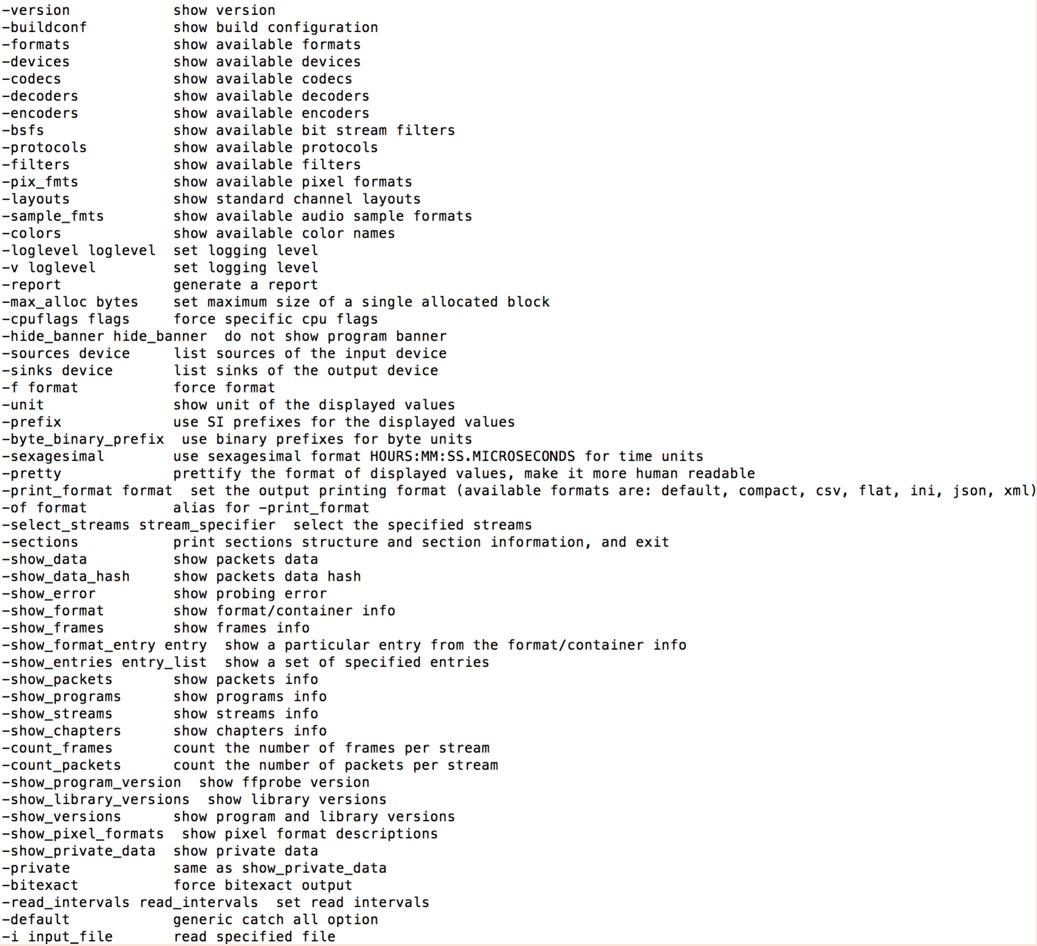
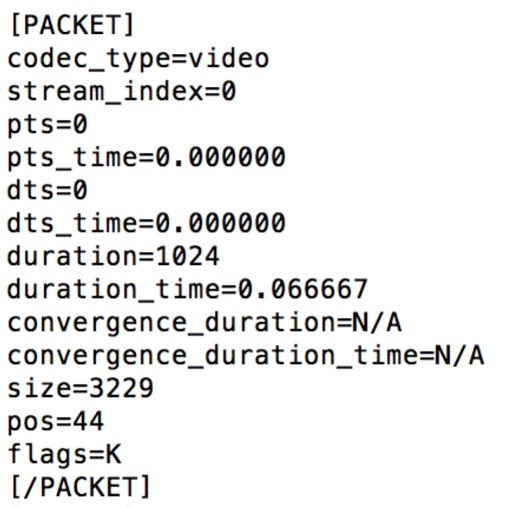
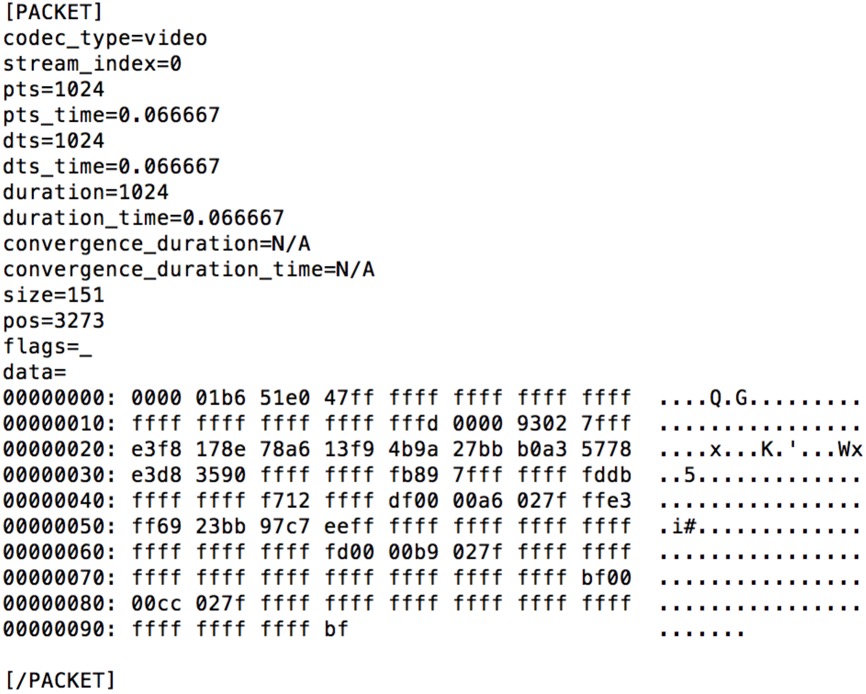
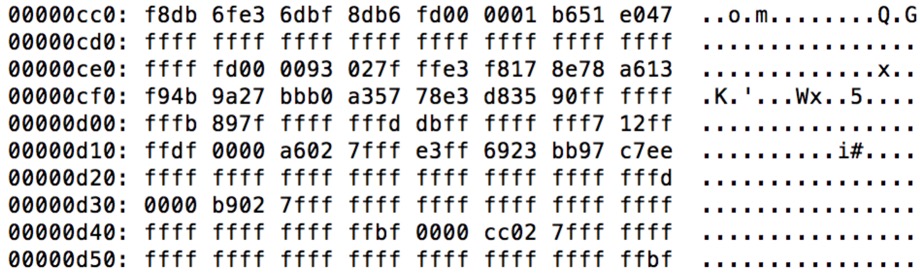
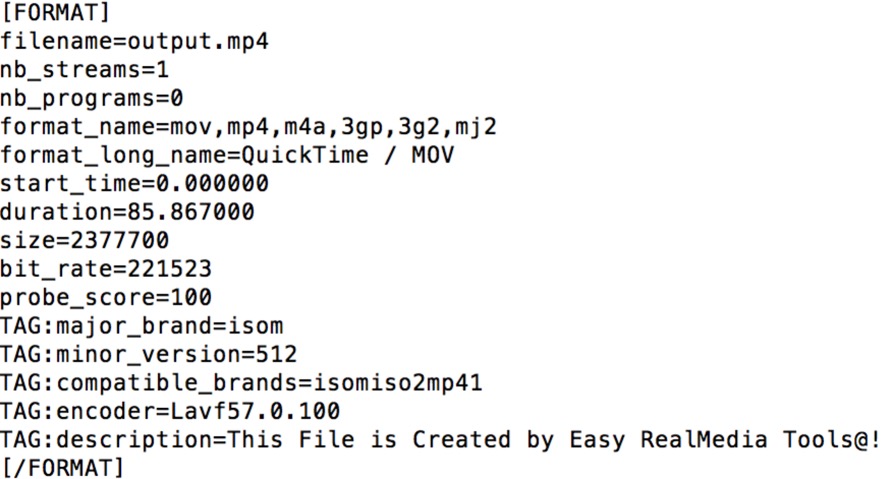
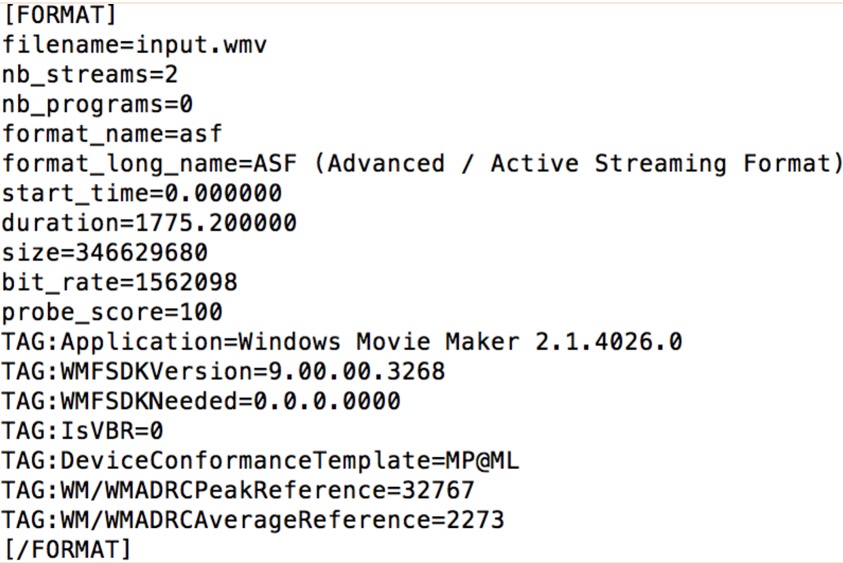
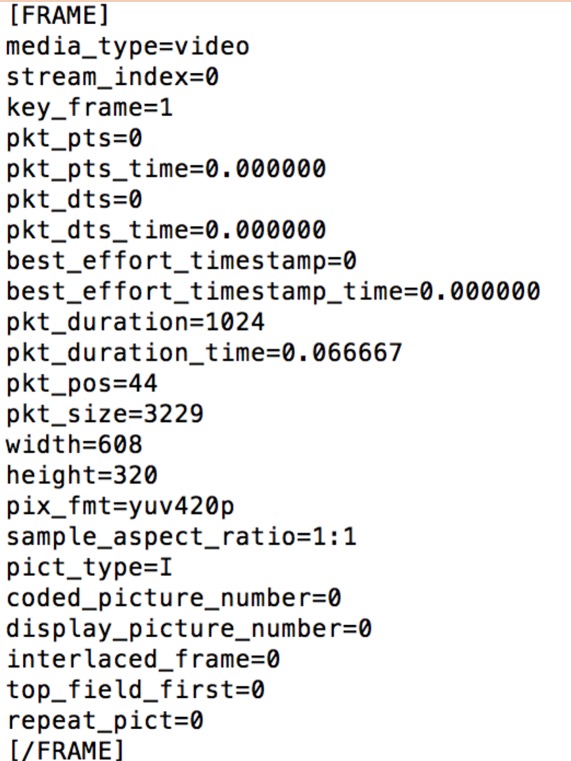
|  |  |  |
| --- | --- | --- |
| 颜色 | 宏块类型条件 | 说明 |
| IMG_266 | IS\_PCM (MB\_TYPE\_INTRA\_PCM) | 无损(原始采样不包含预测信息) |
| IMG_267 | (IS\_INTRA && IS\_ACPRED) || IS\_INTRA16x16 | 16x16帧内预测 |
| IMG_268 | IS\_INTRA4x4 | 4x4帧内预测 |
| IMG_269 | IS\_DIRECT | 无运动向量处理（B帧分片） |
| IMG_270 | IS\_GMC && IS\_SKIP | 16x16跳宏块(P或B帧分片) |
| IMG_271 | IS\_GMC | 全局运动补偿(与H264无关) |
| IMG_272 | !USES\_LIST(1) | 参考过去的信息(P或B帧分片) |
| IMG_273 | !USES\_LIST(0) | 参考未来的信息(B帧分片) |
| IMG_274 | USES\_LIST(0) && USES\_LIST(1) | 参考过去和未来信息(B帧分片) |

例如通过ffplay查看B帧预测与P帧预测信息，希望将信息在窗口中显示出来，使用命令  
# ffplay -vismv pf output.mp4  
显示效果如图  
  
通过图中的箭头可以看到p帧运动估计的信息  
而vismv参数则是用来显示图像解码时的运动向量信息的，参数可以设置三种类型的运动向量信息显示

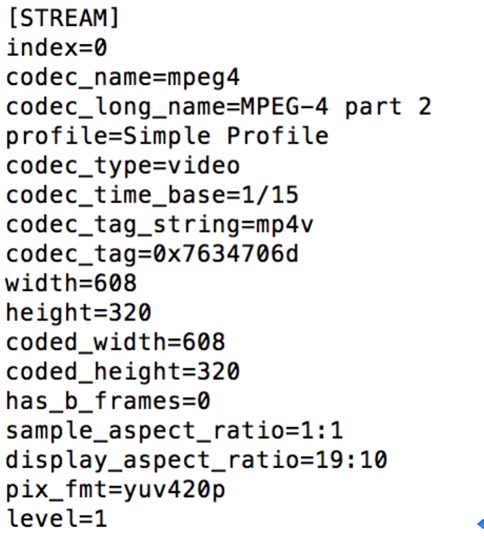
|  |  |
| --- | --- |
| 参数 | 说明 |
| pf | P帧向前运动估计显示 |
| bf | B帧向前运动估计显示 |
| bb | B帧向后运动估计显示 |

这个vismv参数将会在未来被替换掉，未来更多的是使用codecview这个filter来进行设置，如上面图像，也可以通过下面这条命令完成  
# ffplay -flags2 +export\_mvs -ss 40 ~/Movies/objectC/facebook.mp4 -vf codecview=mv=pf+bf+bb

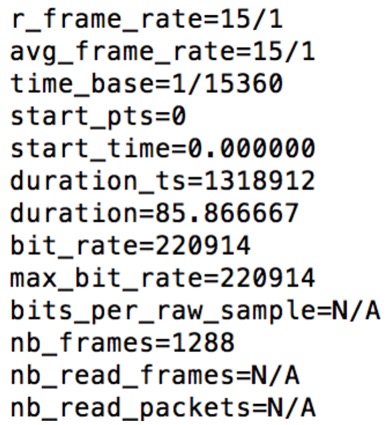
## [FFmpeg使用手册 - ffprobe 的常用命令](http://blog.chinaunix.net/uid-11344913-id-5750194.html)

在FFMpeg套件中，出了ffmpeg还有ffprobe，ffprobe主要用来查看多媒体文件的信息，下面看一下ffprobe中常见的基本命令.  
5.1 ffprobe 常用参数  
ffprobe常用参数比较多，可以用过ffprobe --help来查看详细的信息：  
  
5.2 下面重点聚几个例子：  
5.2.1 使用show\_packets查看多媒体报信息：  
  
show\_packets查看的多媒体包信息使用PACKET标签所包括起来，其中包含的信息主要如下:  
codec\_type: 多媒体类型，例如视频包，音频包等；  
stream\_index:  多媒体的stream索引；  
pts: 多媒体的显示时间值  
pts\_time: 根据不同格式计算过后的多媒体的显示时间  
dts: 多媒体解码时间值  
dts\_time: 根据不同格式计算过后的多媒体解码时间  
duration: 多媒体包占用的时间值  
duration\_time: 根据不同格式计算过后的多媒体包占用的时间值  
size: 多媒体包的大小  
pos: 多媒体包所在的文件偏移位置  
flags: 多媒体包标记，关键包与非关键包的标记  
  
packets包中除了以上的字段和信息外，还可以通过show\_data与show\_packets组合来显示包中的具体的数据：  
  
图中看到了多媒体包中包含的数据，初始信息为0000 01b6 51e0 47ff，那么我们可以根据图中的pos，也就是文件偏移位置查看，根据图中，pos的值为3273，通过转换为16进制，位置为0x00000cc9，与data的数据是可以对应的：  
  
通过ffprobe读取packets来进行对应的数据分析，使用show\_packets与show\_data配合可以更加精确的分析。  
5.2.2 除了packets与data外，ffprobe还可以分析多媒体的封装格式，通过show\_format参数可以查看多媒体的封装格式，封装可是使用FORMAT标签括起来显示：  
  
通过读取format信息，可以看到这个视频文件只有一个流通道，起始时间是0.000000，长度为85.867000，文件大小为2377700字节，码率为221523字节每秒，这个文件有可能是一个mov、有可能是mp4、有可能是m4a、有可能是3gp、有可能是3g2、也有可能是mj2，之所以ffprobe会这么输出，是因为这几种封装格式在ffmpeg中所识别的标签基本相同，所以才会这么多种显示方式，而其他种封装格式不一定是这样的，下面我们再看一个wmv的封装格式  
  
这个input.wmv文件中包含两个流通道，文件封装格式为asf。  
5.2.3 通过ffprobe的show\_frame的参数可以查看视频文件中的帧信息，输出的帧信息使用FRAME标签括起来  
  
通过-show\_frame参数查看到如图的信息，能够看到每一帧的信息，下面介绍一下重要的信息

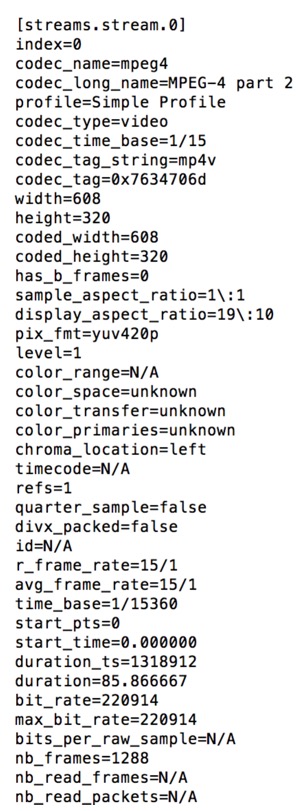
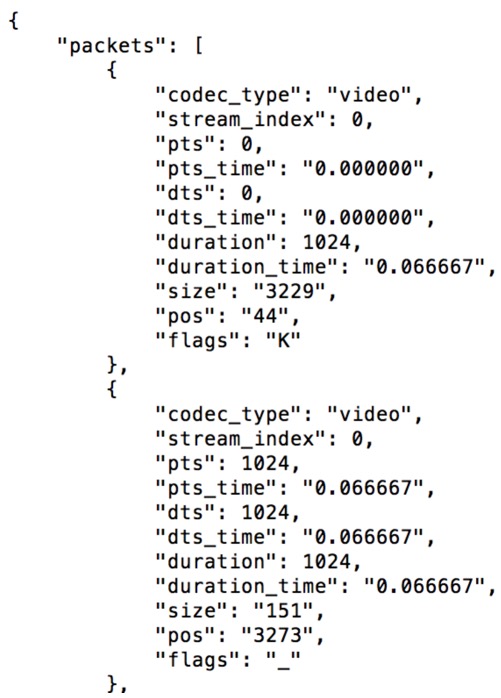
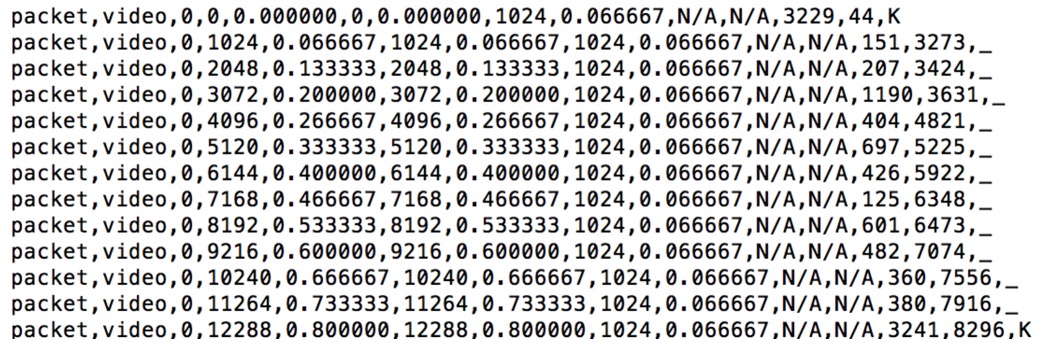
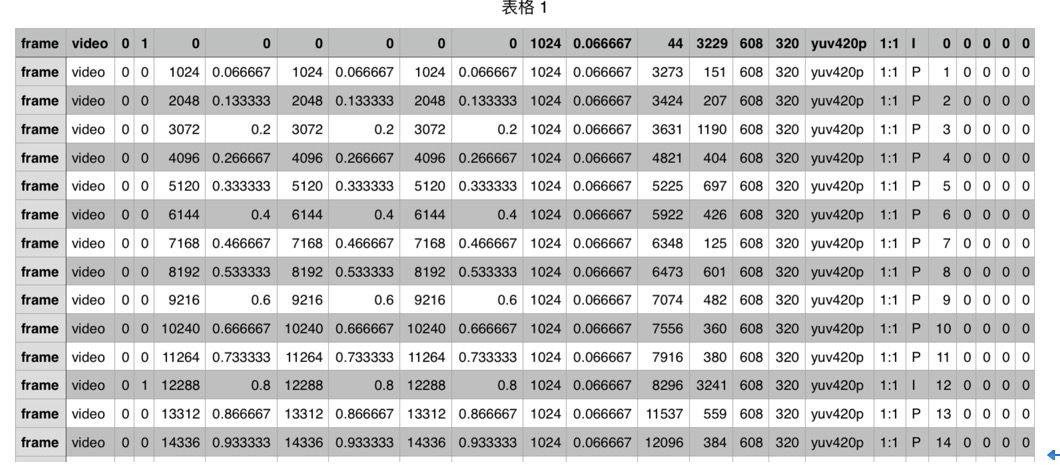
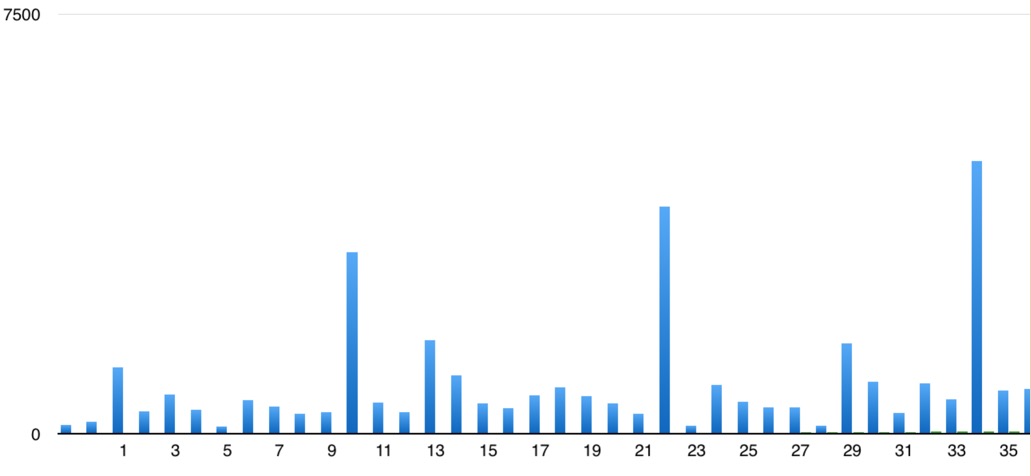
|  |  |  |
| --- | --- | --- |
| 属性 | 说明 | 值 |
| media\_type | 帧的类型(视频、音频、字幕等) | video |
| stream\_index | 帧所在的索引区域 | 0 |
| Key\_frame | 是否是关键帧 | 1 |
| Pkt\_pts | Frame包的pts | 0 |
| Pkt\_pts\_time | Frame包的pts的时间显示 | 0.000000 |
| Pkt\_dts | Frame包的dts | 0 |
| Pkt\_dts\_time | Frame包的dts的时间显示 | 0.000000 |
| Pkt\_duration | Frame包的时长 | 1024 |
| Pkt\_duration\_tine | Frame包的时长时间显示 | 0.066667 |
| Pkt\_pos | Frame包所在文件的偏移位置 | 44 |
| Width | 帧显示的宽度 | 608 |
| height | 帧显示的高度 | 320 |
| Pix\_fmt | 帧的图像色彩格式 | Yuv420p |
| Pict\_type | 帧类型 | I |

在windows下常用的Elecard Stream Eye打开查看mp4时，会看到很直观的帧类型显示，每一帧的判断，用ffprobe的pict\_type同样可以看到，每一帧的大小，也同样可以通过ffprobe的pkt\_size看到，pict\_type可以或得到视频的帧是I帧、P帧或者B帧。  
通过-show\_streams参数可以查看到多媒体文件中的流信息，流的信息使用STREAMS标签括起来：  
  
如图所示，可以看到流的信息：

|  |  |  |
| --- | --- | --- |
| 属性 | 说明 | 值 |
| Index | 流所在的索引区域 | 0 |
| Codec\_name | 编码名 | Mpeg4 |
| Codec\_long\_name | 编码全名 | MPEG-4 part 2 |
| profile | 编码的profile | Simple Profile |
| level | 编码的level | 1 |
| Has\_b\_frames | 包含B帧信息 | 0 |
| Codec\_tyoe | 编码类型 | Video |
| Codec\_time\_base | 编码的时间戳计算基础单位 | 1/15 |
| Pix\_fmt | 图像显示图像色彩格式 | Yuv420p |
| Coded\_width | 图像的宽度 | 608 |
| Coded\_height | 图像的高度 | 320 |
| Codec\_tag\_string | 编码的标签数据 | Mp4v |

除了以上这些信息，还有更多信息  
  
下面再介绍一下图中这些信息

|  |  |  |
| --- | --- | --- |
| 属性 | 说明 | 值 |
| R\_frame\_rate | 实际帧率 | 15/1 |
| Avg\_frame\_rate | 平均帧率 | 15/1 |
| Time\_base | 时间基数(用来做timestamp计算) | 1/15360 |
| Bit\_rate | 码率 | 220914 |
| Max\_bit\_rate | 最大码率 | 220914 |
| Nb\_frames | 帧数 | 1288 |

5.3 ffprobe使用前面的参数可以获得到对应的key-value，但是阅读起来因习惯不同所以有的人认为方便，有的人认为不方便，这样就需要用到ffprobe的print\_format参数来进行相应的格式输出，下面举几种输出的例子：  
5.3.1 XML格式输出：  
  
5.3.2 ini格式输出  
  
5.3.3 flat格式输出  
  
5.3.4 json格式输出  
  
5.3.5 csv格式输出  
  
通过各种格式输出，可以使用对应的绘图方式绘制出可视化的图形：  
csv格式输出后使用excel打开表格形式  
  
将表格中的数据以图形方式绘制出来  
  
图形绘制出来的时候，可以看到对应的图形与Elecard Stream Eye基本相同。

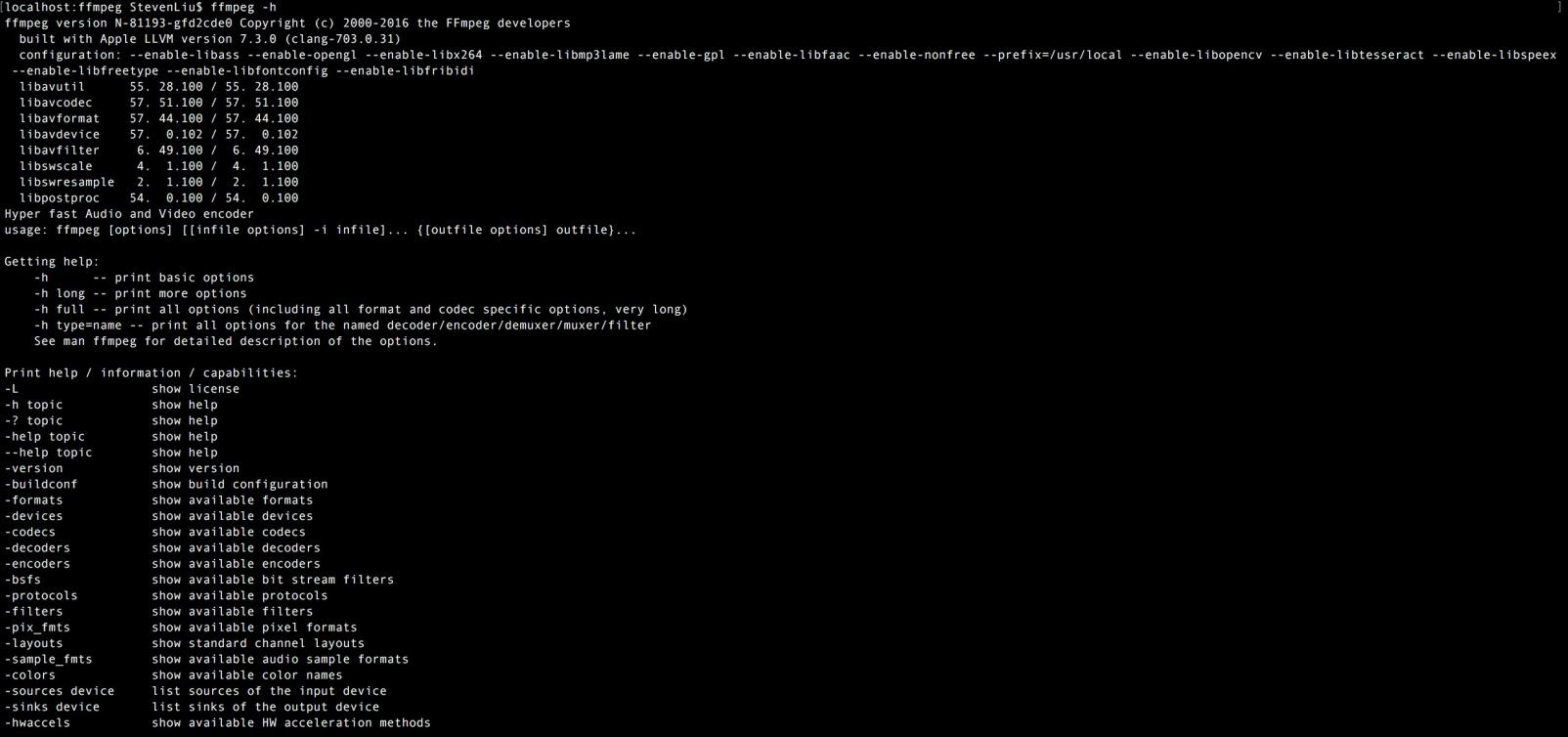
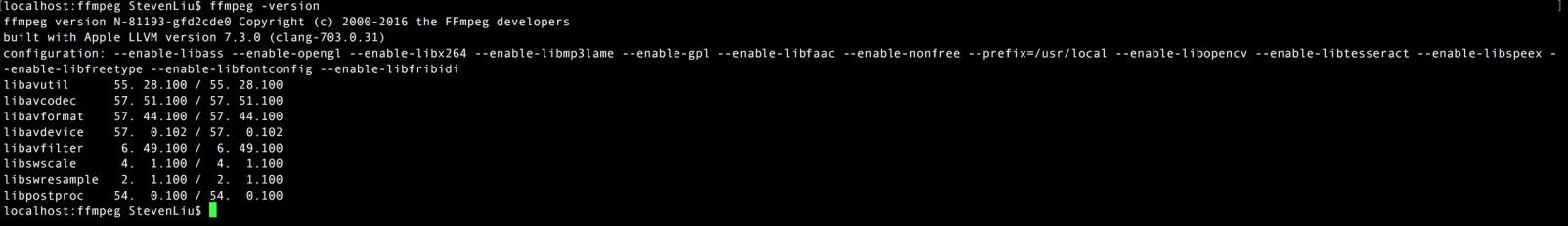
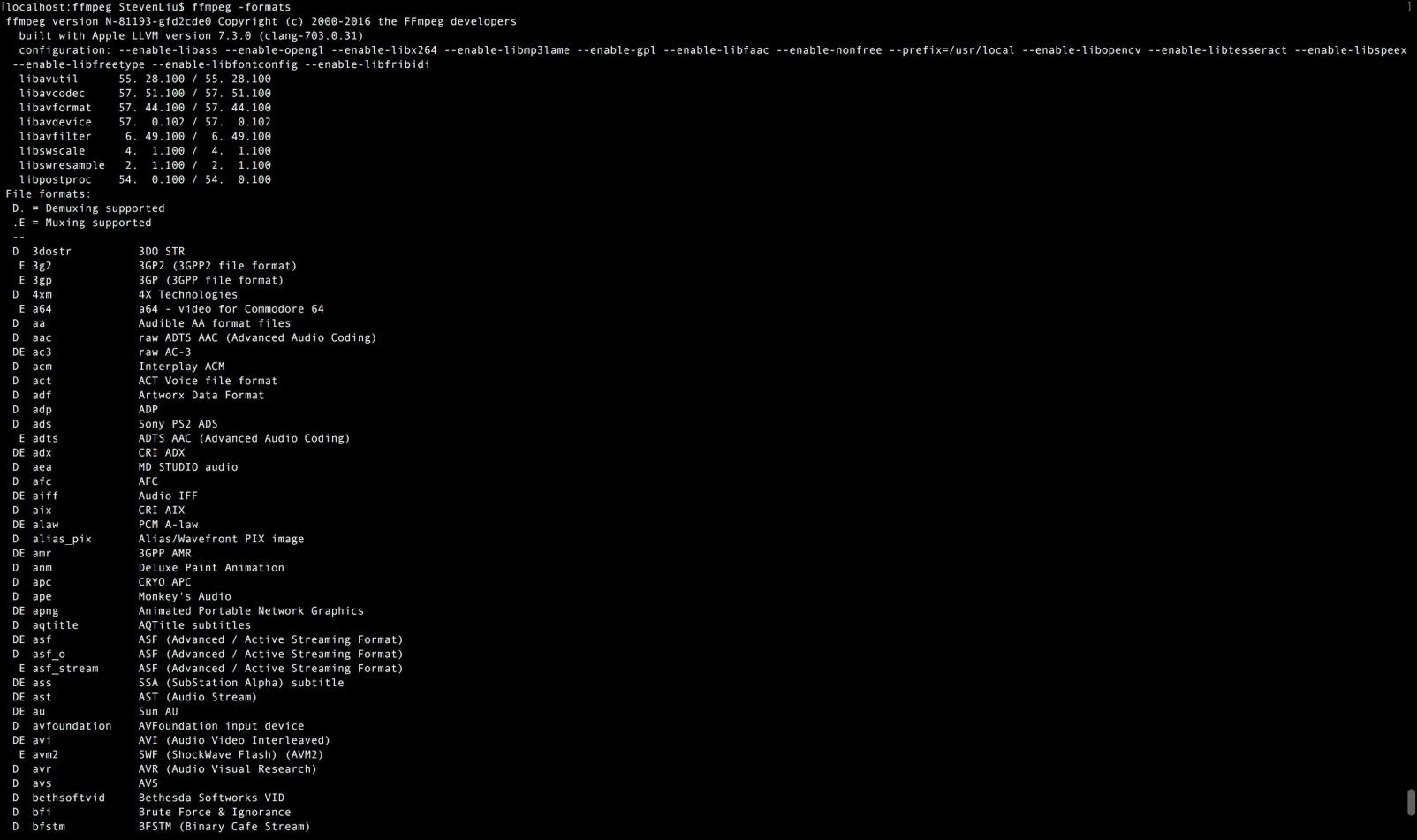
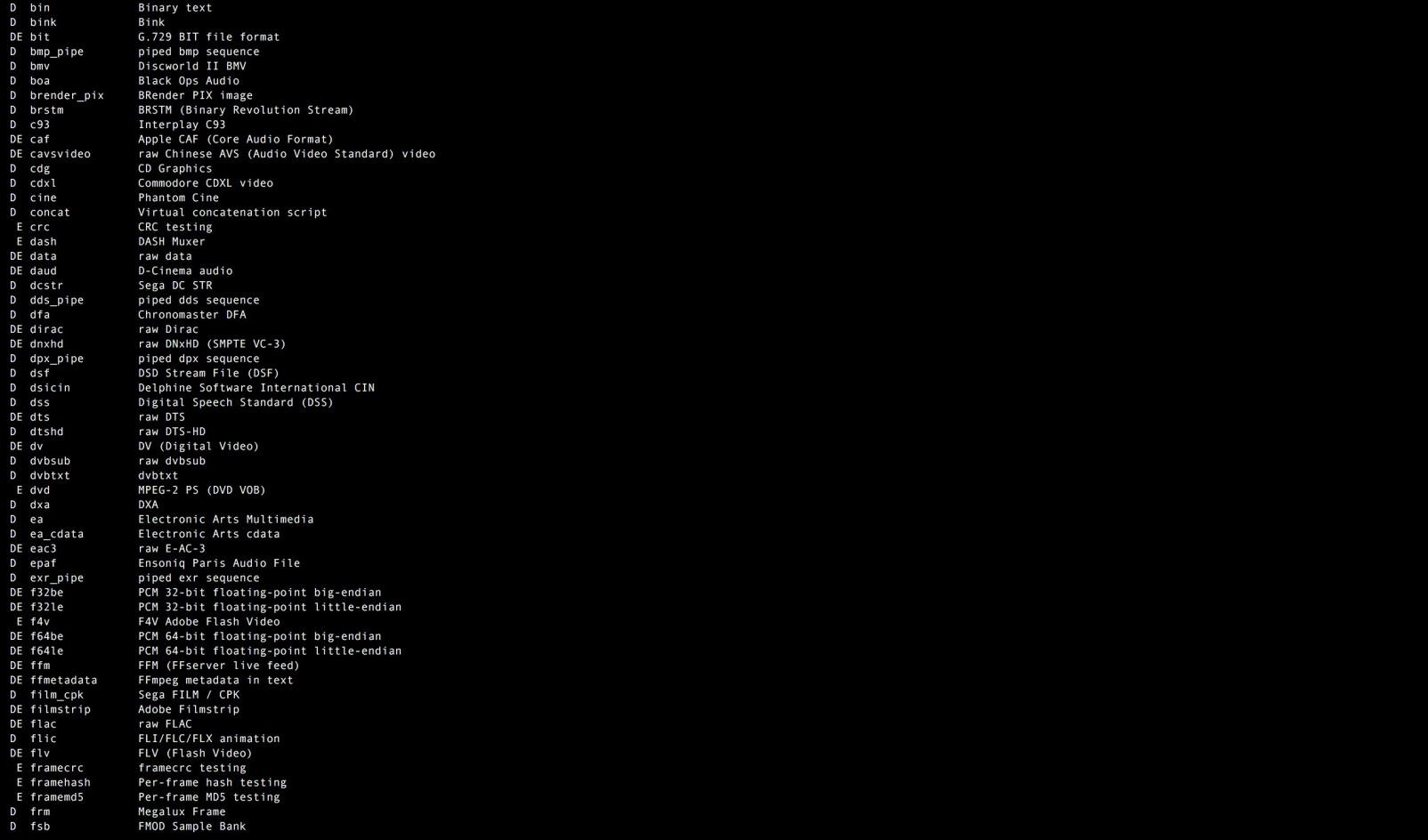
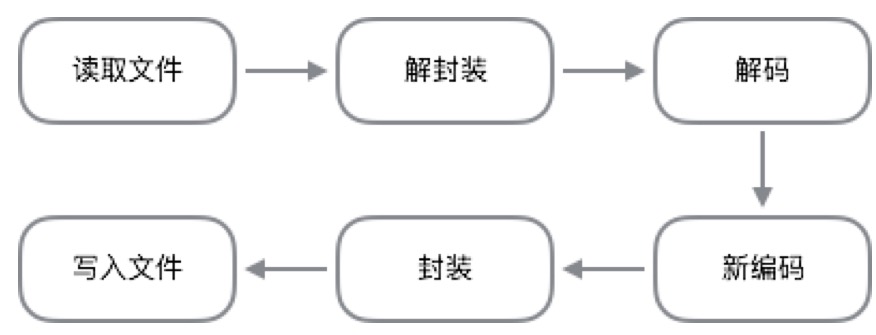
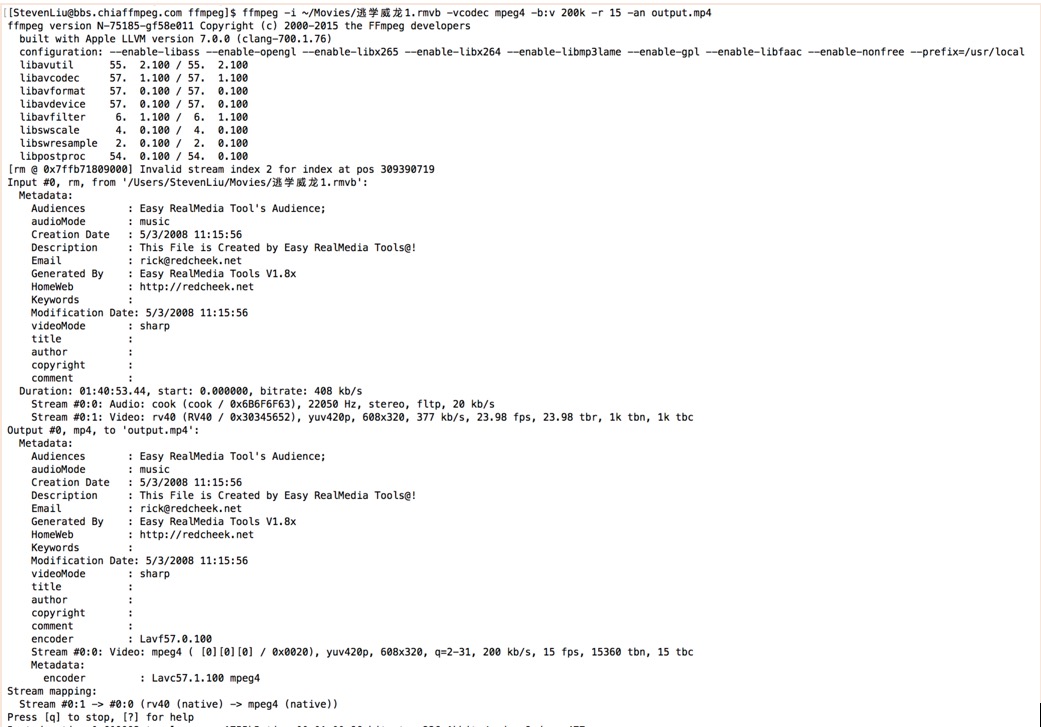
5.4 如果只查看音频流或视频流，使用select\_streams参数即可，例如只查看视频流的frames信息：

ffprobe -show\_frames -select\_streams v -of xml input.mp4

命令行执行后可以看到输出的信息如下，全部为视频的frames信息：



## [FFmpeg使用手册 - FFmpeg 的常用命令](http://blog.chinaunix.net/uid-11344913-id-5750173.html)

ffmpeg在做音视频编解码时非常方便，所以很多场景下转码使用的是ffmpeg，铜鼓通过ffmpeg –help命令操作可以看到ffmpeg常见的命令大概分为六部分：  
1. ffmpeg信息查询部分  
2. 公共做操参数部分  
3. 文件主要操作参数部分  
4. 视频操作参数部分  
5. 音频操作参数部分  
6. 字幕操作参数部分  
ffmpeg信息查询部分主要参数：  
  
  
其中通过ffmpeg --help查看到的help信息是基础的信息，如果想获得高级参数部分，可以通过使用ffmpeg --help long参数来查看，如果希望获得全部的帮助信息，可以同过使用ffmpeg --help full参数来获得。  
通过-L参数，可以看到ffmpeg目前所支持的license协议，通过-version查看ffmpeg的版本，这里所查看的版本，是详细的版本，包含libavformat、libavcodec、libavutil、libavfilter、libswscale、libswresample的版本，如图：  
  
有时候使用ffmpeg转码，发现无法解析视频文件，或者无法生成视频文件，报一些不支持生成对应的视频文件的错误，这个时候就需要查看当前使用的ffmpeg是否支对应的视频文件格式，需要使用ffmpeg -formats来查看：  
  
  
  
bbs.chinaffmpeg.com的孙悟空提示，这个列表比较长，还有一些信息，可以自己执行命令查看  
根据上图可以看到，在输出的内容中，分为三部分：  
第一列是多媒体文件格式的Demuxing支持与Muxing支持；  
第二列是多媒体文件格式；  
第三列是文件格式的详细说明。  
下面详细介绍ffmpeg的封装转换。  
  
4.1 ffmpeg 的封装转换  
FFmpeg的封装转换部分，主要是通过libavformat这部分来完成的功能，通过libavformat库进行mux和demux操作，多媒体文件的格式有很多种，但是还是有好多demux与mux的操作参数是公用的，下面来详细介绍一下这些公用的参数：  
通过查看ffmpeg的help full参数，找到AVFormatContext参数部分，在这个参数下面的所有的参数均为封装转换可使用的参数：  
avioflags： format的缓冲设置，默认为0，就是有缓冲：  
direct： 无缓冲状态；  
probesize： 获得文件内容的大小；  
formatprobesiz： 设置一个获得文件内容的大小来解析文件的format；  
fflags: 设置读取或者写出的格式的形式的标签，分为多种方式：flush\_packets，ignidx，genpts，nofillin，noparse，igndts，discardcorrupt，sortdts，keepside，fastseek，latm，nobuffer，bitexact，下面针对这些format读写方式进行一一详解；  
flush\_packets：  立即将packets数据刷新入文件中；  
genpts: 输出是产生pts;  
nofillin: 不填写可以精确计算缺失的值;  
igndts: 忽略dts；  
discardcorrupt： 丢弃损坏的帧；  
sortdts： 尝试以dts的顺序为准输出；  
keepside： 不合并数据；  
fastseek： 快速seek，但是不够精确；  
latm： 设置RTP MP4\_LATM 生效；  
nobuffer： 直接读取或者写出，不存buffer；  
bitexact： 不写入随机或者不稳定的数据；  
seek2any： 支持随意seek，这个seek可以不以keyframe为参考；  
analyzeduration： 指定解析媒体所需要花销的时间，这里设置的值越高，解析的越准确，如果在直播中为了降低延迟，这个可以设置得低一些  
codec\_whitelist: 设置可以解析的codec的白名单；  
format\_whitelist： 设置可以解析的format的白名单；  
  
  
4.2 ffmpeg 的编转码  
FFmpeg的编解码部分主要是通过libavcodec这个库来完成的功能，通过libavcodec库进行encode与decode操作，多媒体的编码格式种类很多，但是还是有好多基本的操作参数是共同可以设置的，下面来详细介绍一下这些公用的参数：  
通过查看ffmpeg的help full参数可以看到AVCodecContext AVOptions，在这个选项下面的所有参数均为编解码可以使用的参数：  
b: 设置音频与视频码率，可以认为是音视频加起来的码率，默认为200k；  
使用这个参数可以根据b:v设置视频码率，b:a设置音频码率；  
ab： 设置音频的码率，默认是128k；  
g: 设置视频GOP大小，关键帧间隔，默认是12帧一个gop  
ar: 设置音频采样率，默认为0  
ac： 设置音频通道数，默认为0  
bf: 设置非B帧之间的B帧个数，默认为0  
maxrate：最大码率设置，与bufsize一同使用即可，默认为0  
minrate： 最小码率设置，配合maxrate与bufsize可以设置为CBR模式，平时基本没用，默认为0  
bufsize：设置控制码率的buffer的大小，默认为0  
keyint\_min: 设置最小IDR帧间隔，默认为25  
sc\_threshold: 设置场景切换支持，默认为0  
me\_threshold: 设置运动估计阀值，默认为0  
mb\_threshold: 设置宏块阀值，默认为0  
profile: 设置音视频的profile，默认为-99  
level: 设置音视频的level，默认为-99  
timecode\_frame\_start: 设置GOP帧的开始时间，需要在non-drop-frame默认情况下使用  
channel\_layout: 设置音频通道的布局格式  
threads: 设置编解码等工作的线程数  
  
4.3 ffmpeg 的基本编转码原理  
ffmpeg工具主要用途为编码、解码、转码，常用ffmpeg做的为转码操作，使用ffmpeg转码主要原理为：  
  
 通过前两节介绍的参数，可以设置转码的相关参数，如果转码操作则涉及到封装的改变，从而可以通过设置AVCodec与AVFormat的操作参数进行封装与编码的改变，下面举一个例子：  
  
从图中可以看到，上图参数中使用了前面介绍过的参数：  
1. 转封装格式从rmvb格式转为mp4；  
2. 视频编码从rv40转为mpeg4；  
3. 视频码率从原来的377 kb/s 转为 200 kb/s;  
4. 视频帧率从原来的 23.98 fps转为 15 fps  
可以分析出，这个例子的流程与前面提到的流程相同，首先解封装，需要解的封装为rmvb，然后解码，其中视频编码为rv40，音频编码为cook；然后解码后的视频编码为mpeg4，然后封装为一个没有音频的mp4文件。

## FFmpeg\_Tutorial

FFmpeg工具和sdk库的使用demo

### 使用FFmpeg命令行工具和批处理脚本进行简单的音视频文件编辑

1、基本介绍

对于每一个从事音视频技术开发的工程师，想必没有一个人对FFmpeg这个名称感到陌生。FFmpeg是一套非常知名的音视频处理的开源工具，它包含了开发完成的工具软件、封装好的函数库以及源代码供我们按需使用。FFmpeg提供了非常强大的功能，可以完成音视频的编码、解码、转码、视频采集、后处理（抓图、水印、封装/解封装、格式转换等），还有流媒体服务等诸多功能，可以说涵盖了音视频开发中绝大多数的领域。原生的FFmpeg是在Linux环境下开发的，但是通过各种方法（比如交叉编译等）可以使它运行在多种平台环境上，具有比较好的可移植性。

FFmpeg项目的官方网址为：https://ffmpeg.org/。在它的官网上我们可以找到许多非常有用的内容，如项目的简介、版本更新日志、库和源代码的地址、使用文档等信息。官方的使用文档是我们在开发时必不可少的信息来源，其重要性不言而喻。除了官方网站以外，我们下载的FFmpeg的程序包中也有使用参考文档的离线版本。

2、FFmpeg组成

构成FFmpeg主要有三个部分，第一部分是四个作用不同的工具软件，分别是：ffmpeg.exe，ffplay.exe，ffserver.exe和ffprobe.exe。

ffmpeg.exe：音视频转码、转换器

ffplay.exe：简单的音视频播放器

ffserver.exe：流媒体服务器

ffprobe.exe：简单的多媒体码流分析器

第二部分是可以供开发者使用的SDK，为各个不同平台编译完成的库。如果说上面的四个工具软件都是完整成品形式的玩具，那么这些库就相当于乐高积木一样，我们可以根据自己的需求使用这些库开发自己的应用程序。这些库有：

libavcodec：包含音视频编码器和解码器

libavutil：包含多媒体应用常用的简化编程的工具，如随机数生成器、数据结构、数学函数等功能

libavformat：包含多种多媒体容器格式的封装、解封装工具

libavfilter：包含多媒体处理常用的滤镜功能

libavdevice：用于音视频数据采集和渲染等功能的设备相关

libswscale：用于图像缩放和色彩空间和像素格式转换功能

libswresample：用于音频重采样和格式转换等功能

第三部分是整个工程的源代码，无论是编译出来的可执行程序还是SDK，都是由这些源代码编译出来的。FFmpeg的源代码由C语言实现，主要在Linux平台上进行开发。FFmpeg不是一个孤立的工程，它还存在多个依赖的第三方工程来增强它自身的功能。在当前这一系列的博文/视频中，我们暂时不会涉及太多源代码相关的内容，主要以FFmpeg的工具和SDK的调用为主。到下一系列我们将专门研究如何编译源代码并根据源代码来进行二次开发。

3、FFMpeg工具的下载和使用

(1)FFmpeg工具的下载：

在官网上我们可以找到"Download"页面，该页上可以下载FFmpeg的工具、库和源代码等。在选择"Windows Packages"下的Windows Builds后，会跳转到Windows版本的下载页面：

在下载页面上，我们可以看到，对于32位和64位版本，分别提供了三种不同的模式：static、shared和dev

static: 该版本提供了静态版本的FFmpeg工具，将依赖的库生成在了最终的可执行文件中；作为工具而言此版本就可以满足我们的需求；

share: 该版本的工具包括可执行文件和dll，程序运行过程必须依赖于提供的dll文件；

dev: 提供了库的头文件和dll的引导库；

(2)ffplay.exe的使用

ffplay是一个极为简单的音视频媒体播放器。ffplay.exe使用了ffmpeg库和SDL库开发成的，可以用作FFmpeg API的测试工具。 ffplay的使用方法，最简单的是直接按照默认格式播放某一个音视频文件或流：

ffplay.exe -i ../video/IMG\_0886.MOV

除此之外，ffplay还支持传入各种参数来控制播放行为。比较常用的参数有：

-i input\_file：输入文件名

-x width -y height：控制播放窗口的宽高

-t duration：控制播放的时长

-window\_title title：播放窗口的标题，默认为输入文件名

-showmode mode：设置显示模式，0:显示视频;1:显示音频波形；2：显示音频频谱

-autoexit：设置视频播放完成后自动退出

其他参数可以参考官网的文档：https://www.ffmpeg.org/ffplay.html或下载包里的文档

(3)ffprobe的使用

ffprobe可以提供简单的音视频文件分析功能。最简单的方法同ffplay类似：

ffprobe.exe -i ../video/IMG\_0886.MOV

分析完成后，ffprobe会显示音视频文件中包含的每个码流的信息，包括编码格式、像素分辨率、码率、帧率等信息：

ffprobe

(4)ffmpeg的使用

ffmpeg.exe可谓是整个工程的核心所在，它的主要功能是完成音视频各种各样的转换操作。 视频转码：ffmpeg.exe可以将视频文件由原格式转换为其他格式，如从avi转为mp4等：

ffmpeg -i ../video/IMG\_0886.MOV ../video/output\_mpeg4\_mp3.avi

这里，ffmpeg默认将视频编码格式选择为mpeg4，音频转码格式为mp3。如果我们希望保留原始编码，需要增加参数-c copy，表明不做任何转码操作：

ffmpeg -i ../video/IMG\_0886.MOV -c copy ../video/output\_copy.avi

如果我们希望将视频转换为其他编码格式，则需要在参数中指定目标格式-c:v libx265或-vcodec libx265。ffmpeg支持的所有编码器格式可以通过以下命令查看：

ffmpeg.exe -encoders

实际操作：

ffmpeg -i ../video/IMG\_0886.MOV -c:v mjpeg ../video/output\_mjpeg.avi

视频解封装：ffmpeg可以将视频中的音频和视频流分别提取出来。需要在命令行中添加参数-an和-vn，分别表示屏蔽音频和视频流：

@REM 提取视频流

ffmpeg -i ../video/IMG\_0886.MOV -c:v copy -an ../video/IMG\_0886\_v.MOV

@REM 提取音频流

ffmpeg -i ../video/IMG\_0886.MOV -c:a copy -vn ../video/IMG\_0886\_a.aac

视频截取：使用ffmpeg命令并指定参数-ss和-t，分别表示截取开始时刻和截取时长

@REM 视频截取

ffmpeg -ss 5 -t 5 -i ../video/IMG\_0886.MOV -c copy ../video/IMG\_0886\_cut.MOV

二、调用FFmpeg SDK对YUV视频序列进行编码

视频由像素格式编码为码流格式是FFMpeg的一项基本功能。通常，视频编码器的输入视频通常为原始的图像像素值，输出格式为符合某种格式规定的二进制码流。

1、FFMpeg进行视频编码所需要的结构：

AVCodec：AVCodec结构保存了一个编解码器的实例，实现实际的编码功能。通常我们在程序中定义一个指向AVCodec结构的指针指向该实例。

AVCodecContext：AVCodecContext表示AVCodec所代表的上下文信息，保存了AVCodec所需要的一些参数。对于实现编码功能，我们可以在这个结构中设置我们指定的编码参数。通常也是定义一个指针指向AVCodecContext。

AVFrame：AVFrame结构保存编码之前的像素数据，并作为编码器的输入数据。其在程序中也是一个指针的形式。

AVPacket：AVPacket表示码流包结构，包含编码之后的码流数据。该结构可以不定义指针，以一个对象的形式定义。

在我们的程序中，我们将这些结构整合在了一个结构体中：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Struct: CodecCtx

Description: FFMpeg编解码器上下文

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typedef struct

{

AVCodec \*codec; //指向编解码器实例

AVFrame \*frame; //保存解码之后/编码之前的像素数据

AVCodecContext \*c; //编解码器上下文，保存编解码器的一些参数设置

AVPacket pkt; //码流包结构，包含编码码流数据

} CodecCtx;

2、FFMpeg编码的主要步骤：

(1)、输入编码参数

这一步我们可以设置一个专门的配置文件，并将参数按照某个事写入这个配置文件中，再在程序中解析这个配置文件获得编码的参数。如果参数不多的话，我们可以直接使用命令行将编码参数传入即可。

(2)、按照要求初始化需要的FFMpeg结构

首先，所有涉及到编解码的的功能，都必须要注册音视频编解码器之后才能使用。注册编解码调用下面的函数：

avcodec\_register\_all();

编解码器注册完成之后，根据指定的CODEC\_ID查找指定的codec实例。CODEC\_ID通常指定了编解码器的格式，在这里我们使用当前应用最为广泛的H.264格式为例。查找codec调用的函数为avcodec\_find\_encoder，其声明格式为：

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

该函数的输入参数为一个AVCodecID的枚举类型，返回值为一个指向AVCodec结构的指针，用于接收找到的编解码器实例。如果没有找到，那么该函数会返回一个空指针。调用方法如下：

/\* find the mpeg1 video encoder \*/

ctx.codec = avcodec\_find\_encoder(AV\_CODEC\_ID\_H264); //根据CODEC\_ID查找编解码器对象实例的指针

if (!ctx.codec)

{

fprintf(stderr, "Codec not found\n");

return false;

}

AVCodec查找成功后，下一步是分配AVCodecContext实例。分配AVCodecContext实例需要我们前面查找到的AVCodec作为参数，调用的是avcodec\_alloc\_context3函数。其声明方式为：

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

其特点同avcodec\_find\_encoder类似，返回一个指向AVCodecContext实例的指针。如果分配失败，会返回一个空指针。调用方式为：

ctx.c = avcodec\_alloc\_context3(ctx.codec); //分配AVCodecContext实例

if (!ctx.c)

{

fprintf(stderr, "Could not allocate video codec context\n");

return false;

}

需注意，在分配成功之后，应将编码的参数设置赋值给AVCodecContext的成员。

现在，AVCodec、AVCodecContext的指针都已经分配好，然后以这两个对象的指针作为参数打开编码器对象。调用的函数为avcodec\_open2，声明方式为：

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

该函数的前两个参数是我们刚刚建立的两个对象，第三个参数为一个字典类型对象，用于保存函数执行过程总未能识别的AVCodecContext和另外一些私有设置选项。函数的返回值表示编码器是否打开成功，若成功返回0，失败返回一个负数。调用方式为：

if (avcodec\_open2(ctx.c, ctx.codec, NULL) < 0) //根据编码器上下文打开编码器

{

fprintf(stderr, "Could not open codec\n");

exit(1);

}

然后，我们需要处理AVFrame对象。AVFrame表示视频原始像素数据的一个容器，处理该类型数据需要两个步骤，其一是分配AVFrame对象，其二是分配实际的像素数据的存储空间。分配对象空间类似于new操作符一样，只是需要调用函数av\_frame\_alloc。如果失败，那么函数返回一个空指针。AVFrame对象分配成功后，需要设置图像的分辨率和像素格式等。实际调用过程如下：

ctx.frame = av\_frame\_alloc(); //分配AVFrame对象

if (!ctx.frame)

{

fprintf(stderr, "Could not allocate video frame\n");

return false;

}

ctx.frame->format = ctx.c->pix\_fmt;

ctx.frame->width = ctx.c->width;

ctx.frame->height = ctx.c->height;

分配像素的存储空间需要调用av\_image\_alloc函数，其声明方式为：

int av\_image\_alloc(uint8\_t \*pointers[4], int linesizes[4], int w, int h, enum AVPixelFormat pix\_fmt, int align);

该函数的四个参数分别表示AVFrame结构中的缓存指针、各个颜色分量的宽度、图像分辨率（宽、高）、像素格式和内存对其的大小。该函数会返回分配的内存的大小，如果失败则返回一个负值。具体调用方式如：

ret = av\_image\_alloc(ctx.frame->data, ctx.frame->linesize, ctx.c->width, ctx.c->height, ctx.c->pix\_fmt, 32);

if (ret < 0)

{

fprintf(stderr, "Could not allocate raw picture buffer\n");

return false;

}

(3)、编码循环体

到此为止，我们的准备工作已经大致完成，下面开始执行实际编码的循环过程。用伪代码大致表示编码的流程为：

while (numCoded < maxNumToCode)

{

read\_yuv\_data();

encode\_video\_frame();

write\_out\_h264();

}

其中，read\_yuv\_data部分直接使用fread语句读取即可，只需要知道的是，三个颜色分量Y/U/V的地址分别为AVframe::data[0]、AVframe::data[1]和AVframe::data[2]，图像的宽度分别为AVframe::linesize[0]、AVframe::linesize[1]和AVframe::linesize[2]。需要注意的是，linesize中的值通常指的是stride而不是width，也就是说，像素保存区可能是带有一定宽度的无效边区的，在读取数据时需注意。

编码前另外需要完成的操作时初始化AVPacket对象。该对象保存了编码之后的码流数据。对其进行初始化的操作非常简单，只需要调用av\_init\_packet并传入AVPacket对象的指针。随后将AVPacket::data设为NULL，AVPacket::size赋值0.

成功将原始的YUV像素值保存到了AVframe结构中之后，便可以调用avcodec\_encode\_video2函数进行实际的编码操作。该函数可谓是整个工程的核心所在，其声明方式为：

int avcodec\_encode\_video2(AVCodecContext \*avctx, AVPacket \*avpkt, const AVFrame \*frame, int \*got\_packet\_ptr);

其参数和返回值的意义：

avctx: AVCodecContext结构，指定了编码的一些参数；

avpkt: AVPacket对象的指针，用于保存输出码流；

frame：AVframe结构，用于传入原始的像素数据；

got\_packet\_ptr：输出参数，用于标识AVPacket中是否已经有了完整的一帧；

返回值：编码是否成功。成功返回0，失败则返回负的错误码

通过输出参数\*got\_packet\_ptr，我们可以判断是否应有一帧完整的码流数据包输出，如果是，那么可以将AVpacket中的码流数据输出出来，其地址为AVPacket::data，大小为AVPacket::size。具体调用方式如下：

/\* encode the image \*/

ret = avcodec\_encode\_video2(ctx.c, &(ctx.pkt), ctx.frame, &got\_output); //将AVFrame中的像素信息编码为AVPacket中的码流

if (ret < 0)

{

fprintf(stderr, "Error encoding frame\n");

exit(1);

}

if (got\_output)

{

//获得一个完整的编码帧

printf("Write frame %3d (size=%5d)\n", frameIdx, ctx.pkt.size);

fwrite(ctx.pkt.data, 1, ctx.pkt.size, io\_param.pFout);

av\_packet\_unref(&(ctx.pkt));

}

因此，一个完整的编码循环提就可以使用下面的代码实现：

/\* encode 1 second of video \*/

for (frameIdx = 0; frameIdx < io\_param.nTotalFrames; frameIdx++)

{

av\_init\_packet(&(ctx.pkt)); //初始化AVPacket实例

ctx.pkt.data = NULL; // packet data will be allocated by the encoder

ctx.pkt.size = 0;

fflush(stdout);

Read\_yuv\_data(ctx, io\_param, 0); //Y分量

Read\_yuv\_data(ctx, io\_param, 1); //U分量

Read\_yuv\_data(ctx, io\_param, 2); //V分量

ctx.frame->pts = frameIdx;

/\* encode the image \*/

ret = avcodec\_encode\_video2(ctx.c, &(ctx.pkt), ctx.frame, &got\_output); //将AVFrame中的像素信息编码为AVPacket中的码流

if (ret < 0)

{

fprintf(stderr, "Error encoding frame\n");

exit(1);

}

if (got\_output)

{

//获得一个完整的编码帧

printf("Write frame %3d (size=%5d)\n", frameIdx, ctx.pkt.size);

fwrite(ctx.pkt.data, 1, ctx.pkt.size, io\_param.pFout);

av\_packet\_unref(&(ctx.pkt));

}

} //for (frameIdx = 0; frameIdx < io\_param.nTotalFrames; frameIdx++)

(4)、收尾处理

如果我们就此结束编码器的整个运行过程，我们会发现，编码完成之后的码流对比原来的数据少了一帧。这是因为我们是根据读取原始像素数据结束来判断循环结束的，这样最后一帧还保留在编码器中尚未输出。所以在关闭整个解码过程之前，我们必须继续执行编码的操作，直到将最后一帧输出为止。执行这项操作依然调用avcodec\_encode\_video2函数，只是表示AVFrame的参数设为NULL即可：

/\* get the delayed frames \*/

for (got\_output = 1; got\_output; frameIdx++)

{

fflush(stdout);

ret = avcodec\_encode\_video2(ctx.c, &(ctx.pkt), NULL, &got\_output); //输出编码器中剩余的码流

if (ret < 0)

{

fprintf(stderr, "Error encoding frame\n");

exit(1);

}

if (got\_output)

{

printf("Write frame %3d (size=%5d)\n", frameIdx, ctx.pkt.size);

fwrite(ctx.pkt.data, 1, ctx.pkt.size, io\_param.pFout);

av\_packet\_unref(&(ctx.pkt));

}

} //for (got\_output = 1; got\_output; frameIdx++)

此后，我们就可以按计划关闭编码器的各个组件，结束整个编码的流程。编码器组件的释放流程可类比建立流程，需要关闭AVCocec、释放AVCodecContext、释放AVFrame中的图像缓存和对象本身：

avcodec\_close(ctx.c);

av\_free(ctx.c);

av\_freep(&(ctx.frame->data[0]));

av\_frame\_free(&(ctx.frame));

3、总结

使用FFMpeg进行视频编码的主要流程如：

首先解析、处理输入参数，如编码器的参数、图像的参数、输入输出文件；

建立整个FFMpeg编码器的各种组件工具，顺序依次为：avcodec\_register\_all -> avcodec\_find\_encoder -> avcodec\_alloc\_context3 -> avcodec\_open2 -> av\_frame\_alloc -> av\_image\_alloc;

编码循环：av\_init\_packet -> avcodec\_encode\_video2(两次) -> av\_packet\_unref

关闭编码器组件：avcodec\_close，av\_free，av\_freep，av\_frame\_free

三、调用FFmpeg SDK对H.264格式的视频压缩码流进行解码

经过了上篇调用FFMpeg SDK对视频进行编码的过程之后，我们可以比较容易地理解本篇的内容，即上一篇的逆过程——将H.264格式的裸码流解码为像素格式的图像信息。

1、FFMpeg视频解码器所包含的结构

同FFMpeg编码器类似，FFMpeg解码器也需要编码时的各种结构，除此之外，解码器还需要另一个结构——编解码解析器——用于从码流中截取出一帧完整的码流数据单元。因此我们定义一个编解码上下文结构为：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Struct: CodecCtx

Description: FFMpeg编解码器上下文

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typedef struct

{

AVCodec \*pCodec; //编解码器实例指针

AVCodecContext \*pCodecContext; //编解码器上下文，指定了编解码的参数

AVCodecParserContext \*pCodecParserCtx; //编解码解析器，从码流中截取完整的一个NAL Unit数据

AVFrame \*frame; //封装图像对象指针

AVPacket pkt; //封装码流对象实例

} CodecCtx;

2、FFMpeg进行解码操作的主要步骤

(1). 参数传递和解析

同编码器类似，解码器也需要传递参数。不过相比编码器，解码器在运行时所需要的大部分信息都包含在输入码流中，因此输入参数一般只需要指定一个待解码的视频码流文件即可

(2). 按照要求初始化需要的FFMpeg结构

首先，所有涉及到编解码的的功能，都必须要注册音视频编解码器之后才能使用。注册编解码调用下面的函数：

avcodec\_register\_all();

编解码器注册完成之后，根据指定的CODEC\_ID查找指定的codec实例。CODEC\_ID通常指定了编解码器的格式，在这里我们使用当前应用最为广泛的H.264格式为例。查找codec调用的函数为avcodec\_find\_encoder，其声明格式为：

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

该函数的输入参数为一个AVCodecID的枚举类型，返回值为一个指向AVCodec结构的指针，用于接收找到的编解码器实例。如果没有找到，那么该函数会返回一个空指针。调用方法如下：

/\* find the mpeg1 video encoder \*/

ctx.codec = avcodec\_find\_encoder(AV\_CODEC\_ID\_H264); //根据CODEC\_ID查找编解码器对象实例的指针

if (!ctx.codec)

{

fprintf(stderr, "Codec not found\n");

return false;

}

AVCodec查找成功后，下一步是分配AVCodecContext实例。分配AVCodecContext实例需要我们前面查找到的AVCodec作为参数，调用的是avcodec\_alloc\_context3函数。其声明方式为：

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

其特点同avcodec\_find\_encoder类似，返回一个指向AVCodecContext实例的指针。如果分配失败，会返回一个空指针。调用方式为：

ctx.c = avcodec\_alloc\_context3(ctx.codec); //分配AVCodecContext实例

if (!ctx.c)

{

fprintf(stderr, "Could not allocate video codec context\n");

return false;

}

我们应该记得，在FFMpeg视频编码的实现中，AVCodecContext对象分配完成后，下一步实在该对象中设置编码的参数。而在解码器的实现中，基本不需要额外设置参数信息，因此这个对象更多地作为输出参数接收数据。因此对象分配完成后，不需要进一步的初始化操作。

解码器与编码器实现中不同的一点在于，解码器的实现中需要额外的一个AVCodecParserContext结构，用于从码流中截取一个完整的NAL单元。因此我们需要分配一个AVCodecParserContext类型的对象，使用函数av\_parser\_init，声明为：

AVCodecParserContext \*av\_parser\_init(int codec\_id);

调用方式为：

ctx.pCodecParserCtx = av\_parser\_init(AV\_CODEC\_ID\_H264);

if (!ctx.pCodecParserCtx)

{

printf("Could not allocate video parser context\n");

return false;

}

随后，打开AVCodec对象，然后分配AVFrame对象：

//打开AVCodec对象

if (avcodec\_open2(ctx.pCodecContext, ctx.pCodec, NULL) < 0)

{

fprintf(stderr, "Could not open codec\n");

return false;

}

//分配AVFrame对象

ctx.frame = av\_frame\_alloc();

if (!ctx.frame)

{

fprintf(stderr, "Could not allocate video frame\n");

return false;

}

(3)、解码循环体

完成必须的codec组件的建立和初始化之后，开始进入正式的解码循环过程。解码循环通常按照以下几个步骤实现：

首先按照某个指定的长度读取一段码流保存到缓存区中。

由于H.264中一个包的长度是不定的，我们读取一段固定长度的码流通常不可能刚好读出一个包的长度。所以我们就需要使用AVCodecParserContext结构对我们读出的码流信息进行解析，直到取出一个完整的H.264包。对码流解析的函数为av\_parser\_parse2，声明方式如：

int av\_parser\_parse2(AVCodecParserContext \*s,

AVCodecContext \*avctx,

uint8\_t \*\*poutbuf, int \*poutbuf\_size,

const uint8\_t \*buf, int buf\_size,

int64\_t pts, int64\_t dts,

int64\_t pos);

这个函数的各个参数的意义：

AVCodecParserContext \*s：初始化过的AVCodecParserContext对象，决定了码流该以怎样的标准进行解析；

\*AVCodecContext avctx：预先定义好的AVCodecContext对象；

uint8\_t \*\*poutbuf：AVPacket::data的地址，保存解析完成的包数据；

int \*poutbuf\_size：AVPacket的实际数据长度；如果没解析出完整的一个包，这个值为0；

const uint8\_t \*buf, int buf\_size：输入参数，缓存的地址和长度；

int64\_t pts, int64\_t dts：显示和解码的时间戳；

nt64\_t pos ：码流中的位置；

返回值为解析所使用的比特位的长度；

具体的调用方式为：

len = av\_parser\_parse2(ctx.pCodecParserCtx, ctx.pCodecContext,

&(ctx.pkt.data), &(ctx.pkt.size),

pDataPtr, uDataSize,

AV\_NOPTS\_VALUE, AV\_NOPTS\_VALUE, AV\_NOPTS\_VALUE);

如果参数poutbuf\_size的值为0，那么应继续解析缓存中剩余的码流；如果缓存中的数据全部解析后依然未能找到一个完整的包，那么继续从输入文件中读取数据到缓存，继续解析操作，直到pkt.size不为0为止。

在最终解析出一个完整的包之后，我们就可以调用解码API进行实际的解码过程了。解码过程调用的函数为avcodec\_decode\_video2，该函数的声明为：

int avcodec\_decode\_video2(AVCodecContext \*avctx, AVFrame \*picture,

int \*got\_picture\_ptr,

const AVPacket \*avpkt);

这个函数与前篇所遇到的编码函数avcodec\_encode\_video2有些类似，只是参数的顺序略有不同，解码函数的输入输出参数与编码函数相比交换了位置。该函数各个参数的意义：

AVCodecContext \*avctx：编解码器上下文对象，在打开编解码器时生成；

AVFrame \*picture: 保存解码完成后的像素数据；我们只需要分配对象的空间，像素的空间codec会为我们分配好；

int \*got\_picture\_ptr: 标识位，如果为1，那么说明已经有一帧完整的像素帧可以输出了

const AVPacket \*avpkt: 前面解析好的码流包；

实际调用的方法为：

int ret = avcodec\_decode\_video2(ctx.pCodecContext, ctx.frame, &got\_picture, &(ctx.pkt));

if (ret < 0)

{

printf("Decode Error.\n");

return ret;

}

if (got\_picture)

{

//获得一帧完整的图像，写出到输出文件

write\_out\_yuv\_frame(ctx, inputoutput);

printf("Succeed to decode 1 frame!\n");

}

最后，同编码器一样，解码过程的最后一帧可能也存在延迟。处理最后这一帧的方法也跟解码器类似：将AVPacket::data设为NULL，AVPacket::size设为0，然后在调用avcodec\_encode\_video2完成最后的解码过程：

ctx.pkt.data = NULL;

ctx.pkt.size = 0;

while(1)

{

//将编码器中剩余的数据继续输出完

int ret = avcodec\_decode\_video2(ctx.pCodecContext, ctx.frame, &got\_picture, &(ctx.pkt));

if (ret < 0)

{

printf("Decode Error.\n");

return ret;

}

if (got\_picture)

{

write\_out\_yuv\_frame(ctx, inputoutput);

printf("Flush Decoder: Succeed to decode 1 frame!\n");

}

else

{

break;

}

} //while(1)

####(4). 收尾工作 收尾工作主要包括关闭输入输出文件、关闭FFMpeg解码器各个组件。其中关闭解码器组件需要：

avcodec\_close(ctx.pCodecContext);

av\_free(ctx.pCodecContext);

av\_frame\_free(&(ctx.frame));

3、总结

解码器的流程与编码器类似，只是中间需要加入一个解析的过程。整个流程大致为：

1.读取码流数据 -> 2.解析数据，是否尚未解析出一个包就已经用完？是返回1，否继续 -> 3.解析出一个包？是则继续，否则返回上一步继续解析 -> 4.调用avcodec\_decode\_video2进行解码 -> 5.是否解码出一帧完整的图像？是则继续，否则返回上一步继续解码 -> 6.写出图像数据 -> 返回步骤2继续解析。

四、调用FFmpeg SDK解析封装格式的视频为音频流和视频流

我们平常最常用的音视频文件通常不是单独的音频信号和视频信号，而是一个整体的文件。这个文件会在其中包含音频流和视频流，并通过某种方式进行同步播放。通常，文件的音频和视频通过某种标准格式进行复用，生成某种封装格式，而封装的标志就是文件的扩展名，常用的有mp4/avi/flv/mkv等。

从底层考虑，我们可以使用的只有视频解码器、音频解码器，或者再加上一些附加的字幕解码等额外信息，却不存在所谓的mp4解码器或者avi解码器。所以，为了可以正确播放视频文件，必须将封装格式的视频文件分离出视频和音频信息分别进行解码和播放。

事实上，无论是mp4还是avi等文件格式，都有不同的标准格式，对于不同的格式并没有一种通用的解析方法。因此，FFMpeg专门定义了一个库来处理设计文件封装格式的功能，即libavformat。涉及文件的封装、解封装的问题，都可以通过调用libavformat的API实现。这里我们实现一个demo来处理音视频文件的解复用与解码的功能。

1. FFMpeg解复用-解码器所包含的结构

这一过程实际上包括了封装文件的解复用和音频/视频解码两个步骤，因此需要定义的结构体大致包括用于解码和解封装的部分。我们定义下面这样的一个结构体实现这个功能：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Struct: DemuxingVideoAudioContex

Description: 保存解复用器和解码器的上下文组件

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typedef struct

{

AVFormatContext \*fmt\_ctx;

AVCodecContext \*video\_dec\_ctx, \*audio\_dec\_ctx;

AVStream \*video\_stream, \*audio\_stream;

AVFrame \*frame;

AVPacket pkt;

int video\_stream\_idx, audio\_stream\_idx;

int width, height;

uint8\_t \*video\_dst\_data[4];

int video\_dst\_linesize[4];

int video\_dst\_bufsize;

enum AVPixelFormat pix\_fmt;

} DemuxingVideoAudioContex;

这个结构体中的大部分数据类型我们在前面做编码/解码等功能时已经见到过，另外几个是涉及到视频文件的复用的，其中有：

AVFormatContext：用于处理音视频封装格式的上下文信息。

AVStream：表示音频或者视频流的结构。

AVPixelFormat：枚举类型，表示图像像素的格式，最常用的是AV\_PIX\_FMT\_YUV420P

2、FFMpeg解复用-解码的过程

(1)、相关结构的初始化

与使用FFMpeg进行其他操作一样，首先需注册FFMpeg组件：

av\_register\_all();

随后，我们需要打开待处理的音视频文件。然而在此我们不使用打开文件的fopen函数，而是使用avformat\_open\_input函数。该函数不但会打开输入文件，而且可以根据输入文件读取相应的格式信息。该函数的声明如下：

int avformat\_open\_input(AVFormatContext \*\*ps, const char \*url, AVInputFormat \*fmt, AVDictionary \*\*options);

该函数的各个参数的作用为：

ps：根据输入文件接收与格式相关的句柄信息；可以指向NULL，那么AVFormatContext类型的实例将由该函数进行分配。

url：视频url或者文件路径；

fmt：强制输入格式，可设置为NULL以自动检测；

options：保存文件格式无法识别的信息；

返回值：成功返回0，失败则返回负的错误码；

该函数的调用方式为：

if (avformat\_open\_input(&(va\_ctx.fmt\_ctx), files.src\_filename, NULL, NULL) < 0)

{

fprintf(stderr, "Could not open source file %s\n", files.src\_filename);

return -1;

}

打开文件后，调用avformat\_find\_stream\_info函数获取文件中的流信息。该函数的声明为：

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

该函数的第一个参数即前面的文件句柄，第二个参数也是用于保存无法识别的信息的AVDictionary的结构，通常可设为NULL。调用方式如：

/\* retrieve stream information \*/

if (avformat\_find\_stream\_info(va\_ctx.fmt\_ctx, NULL) < 0)

{

fprintf(stderr, "Could not find stream information\n");

return -1;

}

获取文件中的流信息后，下一步则是获取文件中的音频和视频流，并准备对音频和视频信息进行解码。获取文件中的流使用av\_find\_best\_stream函数，其声明如：

int av\_find\_best\_stream(AVFormatContext \*ic,

enum AVMediaType type,

int wanted\_stream\_nb,

int related\_stream,

AVCodec \*\*decoder\_ret,

int flags);

其中各个参数的意义：

ic：视频文件句柄；

type：表示数据的类型，常用的有AVMEDIA\_TYPE\_VIDEO表示视频，AVMEDIA\_TYPE\_AUDIO表示音频等；

wanted\_stream\_nb：我们期望获取到的数据流的数量，设置为-1使用自动获取；

related\_stream：获取相关的音视频流，如果没有则设为-1；

decoder\_ret：返回这一路数据流的解码器；

flags：未定义；

返回值：函数执行成功返回流的数量，失败则返回负的错误码；

在函数执行成功后，便可调用avcodec\_find\_decoder和avcodec\_open2打开解码器准备解码音视频流。该部分的代码实现如：

static int open\_codec\_context(IOFileName &files, DemuxingVideoAudioContex &va\_ctx, enum AVMediaType type)

{

int ret, stream\_index;

AVStream \*st;

AVCodecContext \*dec\_ctx = NULL;

AVCodec \*dec = NULL;

AVDictionary \*opts = NULL;

ret = av\_find\_best\_stream(va\_ctx.fmt\_ctx, type, -1, -1, NULL, 0);

if (ret < 0)

{

fprintf(stderr, "Could not find %s stream in input file '%s'\n", av\_get\_media\_type\_string(type), files.src\_filename);

return ret;

}

else

{

stream\_index = ret;

st = va\_ctx.fmt\_ctx->streams[stream\_index];

/\* find decoder for the stream \*/

dec\_ctx = st->codec;

dec = avcodec\_find\_decoder(dec\_ctx->codec\_id);

if (!dec)

{

fprintf(stderr, "Failed to find %s codec\n", av\_get\_media\_type\_string(type));

return AVERROR(EINVAL);

}

/\* Init the decoders, with or without reference counting \*/

av\_dict\_set(&opts, "refcounted\_frames", files.refcount ? "1" : "0", 0);

if ((ret = avcodec\_open2(dec\_ctx, dec, &opts)) < 0)

{

fprintf(stderr, "Failed to open %s codec\n", av\_get\_media\_type\_string(type));

return ret;

}

switch (type)

{

case AVMEDIA\_TYPE\_VIDEO:

va\_ctx.video\_stream\_idx = stream\_index;

va\_ctx.video\_stream = va\_ctx.fmt\_ctx->streams[stream\_index];

va\_ctx.video\_dec\_ctx = va\_ctx.video\_stream->codec;

break;

case AVMEDIA\_TYPE\_AUDIO:

va\_ctx.audio\_stream\_idx = stream\_index;

va\_ctx.audio\_stream = va\_ctx.fmt\_ctx->streams[stream\_index];

va\_ctx.audio\_dec\_ctx = va\_ctx.audio\_stream->codec;

break;

default:

fprintf(stderr, "Error: unsupported MediaType: %s\n", av\_get\_media\_type\_string(type));

return -1;

}

}

return 0;

}

整体初始化的函数代码为：

int InitDemuxContext(IOFileName &files, DemuxingVideoAudioContex &va\_ctx)

{

int ret = 0, width, height;

/\* register all formats and codecs \*/

av\_register\_all();

/\* open input file, and allocate format context \*/

if (avformat\_open\_input(&(va\_ctx.fmt\_ctx), files.src\_filename, NULL, NULL) < 0)

{

fprintf(stderr, "Could not open source file %s\n", files.src\_filename);

return -1;

}

/\* retrieve stream information \*/

if (avformat\_find\_stream\_info(va\_ctx.fmt\_ctx, NULL) < 0)

{

fprintf(stderr, "Could not find stream information\n");

return -1;

}

if (open\_codec\_context(files, va\_ctx, AVMEDIA\_TYPE\_VIDEO) >= 0)

{

files.video\_dst\_file = fopen(files.video\_dst\_filename, "wb");

if (!files.video\_dst\_file)

{

fprintf(stderr, "Could not open destination file %s\n", files.video\_dst\_filename);

return -1;

}

/\* allocate image where the decoded image will be put \*/

va\_ctx.width = va\_ctx.video\_dec\_ctx->width;

va\_ctx.height = va\_ctx.video\_dec\_ctx->height;

va\_ctx.pix\_fmt = va\_ctx.video\_dec\_ctx->pix\_fmt;

ret = av\_image\_alloc(va\_ctx.video\_dst\_data, va\_ctx.video\_dst\_linesize, va\_ctx.width, va\_ctx.height, va\_ctx.pix\_fmt, 1);

if (ret < 0)

{

fprintf(stderr, "Could not allocate raw video buffer\n");

return -1;

}

va\_ctx.video\_dst\_bufsize = ret;

}

if (open\_codec\_context(files, va\_ctx, AVMEDIA\_TYPE\_AUDIO) >= 0)

{

files.audio\_dst\_file = fopen(files.audio\_dst\_filename, "wb");

if (!files.audio\_dst\_file)

{

fprintf(stderr, "Could not open destination file %s\n", files.audio\_dst\_filename);

return -1;

}

}

if (va\_ctx.video\_stream)

{

printf("Demuxing video from file '%s' into '%s'\n", files.src\_filename, files.video\_dst\_filename);

}

if (va\_ctx.audio\_stream)

{

printf("Demuxing audio from file '%s' into '%s'\n", files.src\_filename, files.audio\_dst\_filename);

}

/\* dump input information to stderr \*/

av\_dump\_format(va\_ctx.fmt\_ctx, 0, files.src\_filename, 0);

if (!va\_ctx.audio\_stream && !va\_ctx.video\_stream)

{

fprintf(stderr, "Could not find audio or video stream in the input, aborting\n");

return -1;

}

return 0;

}

随后要做的，是分配AVFrame和初始化AVPacket对象：

va\_ctx.frame = av\_frame\_alloc(); //分配AVFrame结构对象

if (!va\_ctx.frame)

{

fprintf(stderr, "Could not allocate frame\n");

ret = AVERROR(ENOMEM);

goto end;

}

/\* initialize packet, set data to NULL, let the demuxer fill it \*/

av\_init\_packet(&va\_ctx.pkt); //初始化AVPacket对象

va\_ctx.pkt.data = NULL;

va\_ctx.pkt.size = 0;

(2)、循环解析视频文件的包数据

解析视频文件的循环代码段为：

/\* read frames from the file \*/

while (av\_read\_frame(va\_ctx.fmt\_ctx, &va\_ctx.pkt) >= 0) //从输入程序中读取一个包的数据

{

AVPacket orig\_pkt = va\_ctx.pkt;

do

{

ret = Decode\_packet(files, va\_ctx, &got\_frame, 0); //解码这个包

if (ret < 0)

break;

va\_ctx.pkt.data += ret;

va\_ctx.pkt.size -= ret;

} while (va\_ctx.pkt.size > 0);

av\_packet\_unref(&orig\_pkt);

}

这部分代码逻辑上非常简单，首先调用av\_read\_frame函数，从文件中读取一个packet的数据，并实现了一个Decode\_packet对这个packet进行解码。Decode\_packet函数的实现如下：

int Decode\_packet(IOFileName &files, DemuxingVideoAudioContex &va\_ctx, int \*got\_frame, int cached)

{

int ret = 0;

int decoded = va\_ctx.pkt.size;

static int video\_frame\_count = 0;

static int audio\_frame\_count = 0;

\*got\_frame = 0;

if (va\_ctx.pkt.stream\_index == va\_ctx.video\_stream\_idx)

{

/\* decode video frame \*/

ret = avcodec\_decode\_video2(va\_ctx.video\_dec\_ctx, va\_ctx.frame, got\_frame, &va\_ctx.pkt);

if (ret < 0)

{

printf("Error decoding video frame (%d)\n", ret);

return ret;

}

if (\*got\_frame)

{

if (va\_ctx.frame->width != va\_ctx.width || va\_ctx.frame->height != va\_ctx.height ||

va\_ctx.frame->format != va\_ctx.pix\_fmt)

{

/\* To handle this change, one could call av\_image\_alloc again and

\* decode the following frames into another rawvideo file. \*/

printf("Error: Width, height and pixel format have to be "

"constant in a rawvideo file, but the width, height or "

"pixel format of the input video changed:\n"

"old: width = %d, height = %d, format = %s\n"

"new: width = %d, height = %d, format = %s\n",

va\_ctx.width, va\_ctx.height, av\_get\_pix\_fmt\_name((AVPixelFormat)(va\_ctx.pix\_fmt)),

va\_ctx.frame->width, va\_ctx.frame->height,

av\_get\_pix\_fmt\_name((AVPixelFormat)va\_ctx.frame->format));

return -1;

}

printf("video\_frame%s n:%d coded\_n:%d pts:%s\n", cached ? "(cached)" : "", video\_frame\_count++, va\_ctx.frame->coded\_picture\_number, va\_ctx.frame->pts);

/\* copy decoded frame to destination buffer:

\* this is required since rawvideo expects non aligned data \*/

av\_image\_copy(va\_ctx.video\_dst\_data, va\_ctx.video\_dst\_linesize,

(const uint8\_t \*\*)(va\_ctx.frame->data), va\_ctx.frame->linesize,

va\_ctx.pix\_fmt, va\_ctx.width, va\_ctx.height);

/\* write to rawvideo file \*/

fwrite(va\_ctx.video\_dst\_data[0], 1, va\_ctx.video\_dst\_bufsize, files.video\_dst\_file);

}

}

else if (va\_ctx.pkt.stream\_index == va\_ctx.audio\_stream\_idx)

{

/\* decode audio frame \*/

ret = avcodec\_decode\_audio4(va\_ctx.audio\_dec\_ctx, va\_ctx.frame, got\_frame, &va\_ctx.pkt);

if (ret < 0)

{

printf("Error decoding audio frame (%s)\n", ret);

return ret;

}

/\* Some audio decoders decode only part of the packet, and have to be

\* called again with the remainder of the packet data.

\* Sample: fate-suite/lossless-audio/luckynight-partial.shn

\* Also, some decoders might over-read the packet. \*/

decoded = FFMIN(ret, va\_ctx.pkt.size);

if (\*got\_frame)

{

size\_t unpadded\_linesize = va\_ctx.frame->nb\_samples \* av\_get\_bytes\_per\_sample((AVSampleFormat)va\_ctx.frame->format);

printf("audio\_frame%s n:%d nb\_samples:%d pts:%s\n",

cached ? "(cached)" : "",

audio\_frame\_count++, va\_ctx.frame->nb\_samples,

va\_ctx.frame->pts);

/\* Write the raw audio data samples of the first plane. This works

\* fine for packed formats (e.g. AV\_SAMPLE\_FMT\_S16). However,

\* most audio decoders output planar audio, which uses a separate

\* plane of audio samples for each channel (e.g. AV\_SAMPLE\_FMT\_S16P).

\* In other words, this code will write only the first audio channel

\* in these cases.

\* You should use libswresample or libavfilter to convert the frame

\* to packed data. \*/

fwrite(va\_ctx.frame->extended\_data[0], 1, unpadded\_linesize, files.audio\_dst\_file);

}

}

/\* If we use frame reference counting, we own the data and need

\* to de-reference it when we don't use it anymore \*/

if (\*got\_frame && files.refcount)

av\_frame\_unref(va\_ctx.frame);

return decoded;

}

在该函数中，首先对读取到的packet中的stream\_index分别于先前获取的音频和视频的stream\_index进行对比来确定是音频还是视频流。而后分别调用相应的解码函数进行解码，以视频流为例，判断当前stream为视频流后，调用avcodec\_decode\_video2函数将流数据解码为像素数据，并在获取完整的一帧之后，将其写出到输出文件中。

3、总结

相对于前文讲述过的解码H.264格式裸码流，解封装+解码过程看似多了一个步骤，然而在实现起来实际上并无过多差别。这主要是由于FFMpeg中的多个API已经很好地实现了封装文件的解析和读取过程，如打开文件我们使用avformat\_open\_input代替fopen，读取数据包使用av\_read\_frame代替fread，其他方面只需要多一步判断封装文件中数据流的类型即可，剩余部分与裸码流的解码并无太多差别。

##五、调用FFMpeg SDK封装音频和视频为视频文件

音频和视频的封装过程为解封装的逆过程，即将独立的音频数据和视频数据按照容器文件所规定的格式封装为一个完整的视频文件的过程。对于大多数消费者来说，视频封装的容器是大家最为熟悉的，因为它直接体现在了我们使用的音视频文件扩展名上，比较常见的有mp4、avi、mkv、flv等等。

在进行音频和视频封装时，我们将实际操作一系列音频或视频流数据的生成和写入。所谓流，指的是一系列相关联的包的集合，这些包一般同属于一组按照时间先后顺序进行解码/渲染等处理的数据。在一个比较典型的视频文件中，我们通常至少会包含一个视频流和一个音频流。

在FFMpeg中，表示音频流或视频流有一个专门的结构，即"AVStream"实现。该结构主要对音频和视频数据的处理进行管理和控制。另外，"AVFormatContext"结构也是必须的，因为它包含了控制输入和输出的信息。

音频和视频数据封装为视频文件的主要步骤为：

1. 相关数据结构的准备

首先，根据输出文件的格式获取AVFormatContext结构，获取AVFormatContext结构使用函数avformat\_alloc\_output\_context2实现。该函数的声明为：

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

其中：

ctx：输出到AVFormatContext结构的指针，如果函数失败则返回给该指针为NULL；

oformat：指定输出的AVOutputFormat类型，如果设为NULL则使用format\_name和filename生成；

format\_name：输出格式的名称，如果设为NULL则使用filename默认格式；

filename：目标文件名，如果不使用，可以设为NULL；

分配AVFormatContext成功后，我们需要添加希望封装的数据流，一般是一路视频流+一路音频流（可能还有其他音频流和字幕流等）。添加流首先需要查找流所包含的媒体的编码器，这需要传入codec\_id后使用avcodec\_find\_encoder函数实现，将查找到的编码器保存在AVCodec指针中。

之后，调用avformat\_new\_stream函数向AVFormatContext结构中所代表的媒体文件中添加数据流。该函数的声明如下：

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

其中各个参数的含义：

s：AVFormatContext结构，表示要封装生成的视频文件；

c：上一步根据codec\_id产生的编码器指针；

返回值：指向生成的stream对象的指针；如果失败则返回NULL指针。

此时，一个新的AVStream便已经加入到输出文件中，下面就可以设置stream的id和codec等参数。AVStream::codec是一个AVCodecContext类型的指针变量成员，设置其中的值可以对编码进行配置。整个添加stream的例子如：

/\* Add an output stream. \*/

static void add\_stream(OutputStream \*ost, AVFormatContext \*oc, AVCodec \*\*codec, enum AVCodecID codec\_id)

{

AVCodecContext \*c;

int i;

/\* find the encoder \*/

\*codec = avcodec\_find\_encoder(codec\_id);

if (!(\*codec))

{

fprintf(stderr, "Could not find encoder for '%s'\n", avcodec\_get\_name(codec\_id));

exit(1);

}

ost->st = avformat\_new\_stream(oc, \*codec);

if (!ost->st)

{

fprintf(stderr, "Could not allocate stream\n");

exit(1);

}

ost->st->id = oc->nb\_streams - 1;

c = ost->st->codec;

switch ((\*codec)->type)

{

case AVMEDIA\_TYPE\_AUDIO:

c->sample\_fmt = (\*codec)->sample\_fmts ? (\*codec)->sample\_fmts[0] : AV\_SAMPLE\_FMT\_FLTP;

c->bit\_rate = 64000;

c->sample\_rate = 44100;

if ((\*codec)->supported\_samplerates)

{

c->sample\_rate = (\*codec)->supported\_samplerates[0];

for (i = 0; (\*codec)->supported\_samplerates[i]; i++)

{

if ((\*codec)->supported\_samplerates[i] == 44100)

c->sample\_rate = 44100;

}

}

c->channels = av\_get\_channel\_layout\_nb\_channels(c->channel\_layout);

c->channel\_layout = AV\_CH\_LAYOUT\_STEREO;

if ((\*codec)->channel\_layouts)

{

c->channel\_layout = (\*codec)->channel\_layouts[0];

for (i = 0; (\*codec)->channel\_layouts[i]; i++)

{

if ((\*codec)->channel\_layouts[i] == AV\_CH\_LAYOUT\_STEREO)

c->channel\_layout = AV\_CH\_LAYOUT\_STEREO;

}

}

c->channels = av\_get\_channel\_layout\_nb\_channels(c->channel\_layout);

{

AVRational r = { 1, c->sample\_rate };

ost->st->time\_base = r;

}

break;

case AVMEDIA\_TYPE\_VIDEO:

c->codec\_id = codec\_id;

c->bit\_rate = 400000;

/\* Resolution must be a multiple of two. \*/

c->width = 352;

c->height = 288;

/\* timebase: This is the fundamental unit of time (in seconds) in terms

\* of which frame timestamps are represented. For fixed-fps content,

\* timebase should be 1/framerate and timestamp increments should be

\* identical to 1. \*/

{

AVRational r = { 1, STREAM\_FRAME\_RATE };

ost->st->time\_base = r;

}

c->time\_base = ost->st->time\_base;

c->gop\_size = 12; /\* emit one intra frame every twelve frames at most \*/

c->pix\_fmt = AV\_PIX\_FMT\_YUV420P;

if (c->codec\_id == AV\_CODEC\_ID\_MPEG2VIDEO)

{

/\* just for testing, we also add B frames \*/

c->max\_b\_frames = 2;

}

if (c->codec\_id == AV\_CODEC\_ID\_MPEG1VIDEO)

{

/\* Needed to avoid using macroblocks in which some coeffs overflow.

\* This does not happen with normal video, it just happens here as

\* the motion of the chroma plane does not match the luma plane. \*/

c->mb\_decision = 2;

}

break;

default:

break;

}

/\* Some formats want stream headers to be separate. \*/

if (oc->oformat->flags & AVFMT\_GLOBALHEADER)

c->flags |= AV\_CODEC\_FLAG\_GLOBAL\_HEADER;

}

2. 打开音视频

打开音视频主要涉及到打开编码音视频数据所需要的编码器，以及分配相应的frame对象。其中打开编码器如之前一样，调用avcodec\_open函数，分配frame对象调用av\_frame\_alloc以及av\_frame\_get\_buffer。分配frame对象的实现如下：

static AVFrame \*alloc\_picture(enum AVPixelFormat pix\_fmt, int width, int height)

{

AVFrame \*picture;

int ret;

picture = av\_frame\_alloc();

if (!picture)

{

return NULL;

}

picture->format = pix\_fmt;

picture->width = width;

picture->height = height;

/\* allocate the buffers for the frame data \*/

ret = av\_frame\_get\_buffer(picture, 32);

if (ret < 0)

{

fprintf(stderr, "Could not allocate frame data.\n");

exit(1);

}

return picture;

}

而上层打开音视频部分的实现如：

void Open\_video(AVFormatContext \*oc, AVCodec \*codec, OutputStream \*ost, AVDictionary \*opt\_arg, IOParam &io)

{

int ret;

AVCodecContext \*c = ost->st->codec;

AVDictionary \*opt = NULL;

av\_dict\_copy(&opt, opt\_arg, 0);

/\* open the codec \*/

ret = avcodec\_open2(c, codec, &opt);

av\_dict\_free(&opt);

if (ret < 0)

{

fprintf(stderr, "Could not open video codec: %d\n", ret);

exit(1);

}

/\* allocate and init a re-usable frame \*/

ost->frame = alloc\_picture(c->pix\_fmt, c->width, c->height);

if (!ost->frame)

{

fprintf(stderr, "Could not allocate video frame\n");

exit(1);

}

/\* If the output format is not YUV420P, then a temporary YUV420P

\* picture is needed too. It is then converted to the required

\* output format. \*/

ost->tmp\_frame = NULL;

if (c->pix\_fmt != AV\_PIX\_FMT\_YUV420P)

{

ost->tmp\_frame = alloc\_picture(AV\_PIX\_FMT\_YUV420P, c->width, c->height);

if (!ost->tmp\_frame)

{

fprintf(stderr, "Could not allocate temporary picture\n");

exit(1);

}

}

//打开输入YUV文件

fopen\_s(&g\_inputYUVFile, io.input\_file\_name, "rb+");

if (g\_inputYUVFile == NULL)

{

fprintf(stderr, "Open input yuv file failed.\n");

exit(1);

}

}

3. 打开输出文件并写入文件头

如果判断需要写出文件的话，则需要打开输出文件。在这里，我们可以不再定义输出文件指针，并使用fopen打开，而是直接使用FFMpeg的API——avio\_open来实现输出文件的打开功能。该函数的声明如下：

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

该函数的输入参数为：

s：输出参数，返回一个AVIOContext；如果打开失败则返回NULL；

url：输出的url或者文件的完整路径；

flags：控制文件打开方式，如读方式、写方式和读写方式；

实际的代码实现方式如下：

/\* open the output file, if needed \*/

if (!(fmt->flags & AVFMT\_NOFILE))

{

ret = avio\_open(&oc->pb, io.output\_file\_name, AVIO\_FLAG\_WRITE);

if (ret < 0)

{

fprintf(stderr, "Could not open '%s': %d\n", io.output\_file\_name, ret);

return 1;

}

}

写入文件头操作是生成视频文件中极为重要的一步，而实现过程却非常简单，只需要通过函数avformat\_write\_header即可，该函数的声明为：

int avformat\_write\_header(AVFormatContext \*s, AVDictionary \*\*options);

其输入参数实际上重要的只有第一个，即标记输出文件的句柄对象指针；options用于保存无法识别的设置项，可以传入一个空指针。其返回值表示写文件头成功与否，成功则返回0，失败则返回负的错误码。

实现方式如：

/\* Write the stream header, if any. \*/

ret = avformat\_write\_header(oc, &opt);

if (ret < 0)

{

fprintf(stderr, "Error occurred when opening output file: %d\n",ret);

return 1;

}

###4. 编码和封装循环

以视频流为例。编解码循环的过程实际上可以封装在一个函数Write\_video\_frame中。该函数从逻辑上可以分为3个部分：获取原始视频信号、视频编码、写入输出文件。

(1) 读取原始视频数据

这一部分主要实现根据时长判断是否需要继续进行处理、读取视频到AVFrame和设置pts。其中时长判断部分根据pts和AVCodecContext的time\_base判断。实现如下：

AVCodecContext \*c = ost->st->codec;

/\* check if we want to generate more frames \*/

{

AVRational r = { 1, 1 };

if (av\_compare\_ts(ost->next\_pts, ost->st->codec->time\_base, STREAM\_DURATION, r) >= 0)

{

return NULL;

}

}

读取视频到AVFrame我们定义一个fill\_yuv\_image函数实现：

static void fill\_yuv\_image(AVFrame \*pict, int frame\_index, int width, int height)

{

int x, y, i, ret;

/\* when we pass a frame to the encoder, it may keep a reference to it

\* internally;

\* make sure we do not overwrite it here

\*/

ret = av\_frame\_make\_writable(pict);

if (ret < 0)

{

exit(1);

}

i = frame\_index;

/\* Y \*/

for (y = 0; y < height; y++)

{

ret = fread\_s(&pict->data[0][y \* pict->linesize[0]], pict->linesize[0], 1, width, g\_inputYUVFile);

if (ret != width)

{

printf("Error: Read Y data error.\n");

exit(1);

}

}

/\* U \*/

for (y = 0; y < height / 2; y++)

{

ret = fread\_s(&pict->data[1][y \* pict->linesize[1]], pict->linesize[1], 1, width / 2, g\_inputYUVFile);

if (ret != width / 2)

{

printf("Error: Read U data error.\n");

exit(1);

}

}

/\* V \*/

for (y = 0; y < height / 2; y++)

{

ret = fread\_s(&pict->data[2][y \* pict->linesize[2]], pict->linesize[2], 1, width / 2, g\_inputYUVFile);

if (ret != width / 2)

{

printf("Error: Read V data error.\n");

exit(1);

}

}

}

然后进行pts的设置，很简单，就是上一个frame的pts递增1：

ost->frame->pts = ost->next\_pts++;

整个获取视频信号的实现如：

static AVFrame \*get\_video\_frame(OutputStream \*ost)

{

AVCodecContext \*c = ost->st->codec;

/\* check if we want to generate more frames \*/

{

AVRational r = { 1, 1 };

if (av\_compare\_ts(ost->next\_pts, ost->st->codec->time\_base, STREAM\_DURATION, r) >= 0)

{

return NULL;

}

}

fill\_yuv\_image(ost->frame, ost->next\_pts, c->width, c->height);

ost->frame->pts = ost->next\_pts++;

return ost->frame;

}

(2) 视频编码

视频编码的方式同之前几次使用的方式相同，即调用avcodec\_encode\_video2，实现方法如：

/\* encode the image \*/

ret = avcodec\_encode\_video2(c, &pkt, frame, &got\_packet);

if (ret < 0)

{

fprintf(stderr, "Error encoding video frame: %d\n", ret);

exit(1);

}

(3) 写出编码后的数据到输出视频文件

这部分的实现过程很简单，方式如下：

/\* rescale output packet timestamp values from codec to stream timebase \*/

av\_packet\_rescale\_ts(pkt, \*time\_base, st->time\_base);

pkt->stream\_index = st->index;

/\* Write the compressed frame to the media file. \*/

// log\_packet(fmt\_ctx, pkt);

return av\_interleaved\_write\_frame(fmt\_ctx, pkt);

av\_packet\_rescale\_ts函数的作用为不同time\_base度量之间的转换，在这里起到的作用是将AVCodecContext的time\_base转换为AVStream中的time\_base。av\_interleaved\_write\_frame函数的作用是写出AVPacket到输出文件。该函数的声明为：

int av\_interleaved\_write\_frame(AVFormatContext \*s, AVPacket \*pkt);

该函数的声明也很简单，第一个参数是之前打开并写入文件头的文件句柄，第二个参数是写入文件的packet。返回值为错误码，成功返回0，失败则返回一个负值。

Write\_video\_frame函数的整体实现如：

int Write\_video\_frame(AVFormatContext \*oc, OutputStream \*ost)

{

int ret;

AVCodecContext \*c;

AVFrame \*frame;

int got\_packet = 0;

AVPacket pkt = { 0 };

c = ost->st->codec;

frame = get\_video\_frame(ost);

av\_init\_packet(&pkt);

/\* encode the image \*/

ret = avcodec\_encode\_video2(c, &pkt, frame, &got\_packet);

if (ret < 0)

{

fprintf(stderr, "Error encoding video frame: %d\n", ret);

exit(1);

}

if (got\_packet)

{

ret = write\_frame(oc, &c->time\_base, ost->st, &pkt);

}

else

{

ret = 0;

}

if (ret < 0)

{

fprintf(stderr, "Error while writing video frame: %d\n", ret);

exit(1);

}

return (frame || got\_packet) ? 0 : 1;

}

以上是写入一帧视频数据的方法，写入音频的方法于此大同小异。整个编码封装的循环上层实现如：

while (encode\_video || encode\_audio)

{

/\* select the stream to encode \*/

if (encode\_video && (!encode\_audio || av\_compare\_ts(video\_st.next\_pts, video\_st.st->codec->time\_base, audio\_st.next\_pts, audio\_st.st->codec->time\_base) <= 0))

{

encode\_video = !Write\_video\_frame(oc, &video\_st);

if (encode\_video)

{

printf("Write %d video frame.\n", videoFrameIdx++);

}

else

{

printf("Video ended, exit.\n");

}

}

else

{

encode\_audio = !Write\_audio\_frame(oc, &audio\_st);

if (encode\_audio)

{

printf("Write %d audio frame.\n", audioFrameIdx++);

}

else

{

printf("Audio ended, exit.\n");

}

}

}

###5. 写入文件尾，并进行收尾工作

写入文件尾的数据同写文件头一样简单，只需要调用函数av\_write\_trailer即可实现：

int av\_write\_trailer(AVFormatContext \*s);

该函数只有一个参数即视频文件的句柄，当返回值为0时表示函数执行成功。

整个流程的收尾工作包括关闭文件中的数据流、关闭输出文件和释放AVCodecContext对象。其中关闭数据流的实现方式如：

void Close\_stream(AVFormatContext \*oc, OutputStream \*ost)

{

avcodec\_close(ost->st->codec);

av\_frame\_free(&ost->frame);

av\_frame\_free(&ost->tmp\_frame);

sws\_freeContext(ost->sws\_ctx);

swr\_free(&ost->swr\_ctx);

}

关闭输出文件和释放AVCodecContext对象：

if (!(fmt->flags & AVFMT\_NOFILE))

/\* Close the output file. \*/

avio\_closep(&oc->pb);

/\* free the stream \*/

avformat\_free\_context(oc);

至此，整个处理流程便结束了。正确设置输入的YUV文件就可以获取封装好的音视频文件。

##六、调用FFMpeg SDK实现视频文件的转封装

有时候我们可能会面对这样的一种需求，即我们不需要对视频内的音频或视频信号进行什么实际的操作，只是希望能把文件的封装格式进行转换，例如从avi转换为mp4格式或者flv格式等。实际上，转封装不需要对内部的音视频进行解码，只需要根据从输入文件中获取包含的数据流添加到输出文件中，然后将输入文件中的数据包按照规定格式写入到输出文件中去。

1、解析命令行参数

如同之前的工程一样，我们使用命令行参数传入输入和输出的文件名。为此，我们定义了如下的结构体和函数来实现传入输入输出文件的过程：

typedef struct \_IOFiles

{

const char \*inputName;

const char \*outputName;

} IOFiles;

static bool hello(int argc, char \*\*argv, IOFiles &io\_param)

{

printf("FFMpeg Remuxing Demo.\nCommand format: %s inputfile outputfile\n", argv[0]);

if (argc != 3)

{

printf("Error: command line error, please re-check.\n");

return false;

}

io\_param.inputName = argv[1];

io\_param.outputName = argv[2];

return true;

}

在main函数执行时，调用hello函数解析命令行并保存到IOFiles结构中：

int main(int argc, char \*\*argv)

{

IOFiles io\_param;

if (!hello(argc, argv, io\_param))

{

return -1;

}

//......

}

2、所需要的结构与初始化操作

为了实现视频文件的转封装操作，我们需要以下的结构：

AVOutputFormat \*ofmt = NULL;

AVFormatContext \*ifmt\_ctx = NULL, \*ofmt\_ctx = NULL;

AVPacket pkt;

然后所需要的初始化操作有打开输入视频文件、获取其中的流信息和获取输出文件的句柄：

av\_register\_all();

//按封装格式打开输入视频文件

if ((ret = avformat\_open\_input(&ifmt\_ctx, io\_param.inputName, NULL, NULL)) < 0)

{

printf("Error: Open input file failed.\n");

goto end;

}

//获取输入视频文件中的流信息

if ((ret = avformat\_find\_stream\_info(ifmt\_ctx, NULL)) < 0)

{

printf("Error: Failed to retrieve input stream information.\n");

goto end;

}

av\_dump\_format(ifmt\_ctx, 0, io\_param.inputName, 0);

//按照文件名获取输出文件的句柄

avformat\_alloc\_output\_context2(&ofmt\_ctx, NULL, NULL, io\_param.outputName);

if (!ofmt\_ctx)

{

printf("Error: Could not create output context.\n");

goto end;

}

ofmt = ofmt\_ctx->oformat;

3、 向输出文件中添加Stream并打开输出文件

在我们获取到了输入文件中的流信息后，保持输入流中的codec不变，并以其为依据添加到输出文件中：

for (unsigned int i = 0; i < ifmt\_ctx->nb\_streams ; i++)

{

AVStream \*inStream = ifmt\_ctx->streams[i];

AVStream \*outStream = avformat\_new\_stream(ofmt\_ctx, inStream->codec->codec);

if (!outStream)

{

printf("Error: Could not allocate output stream.\n");

goto end;

}

ret = avcodec\_copy\_context(outStream->codec, inStream->codec);

outStream->codec->codec\_tag = 0;

if (ofmt\_ctx->oformat->flags & AVFMT\_GLOBALHEADER)

{

outStream->codec->flags |= AV\_CODEC\_FLAG\_GLOBAL\_HEADER;

}

}

av\_dump\_format(ofmt\_ctx, 0, io\_param.outputName, 1);

这里调用了函数avcodec\_copy\_context函数，该函数的声明如下：

int avcodec\_copy\_context(AVCodecContext \*dest, const AVCodecContext \*src);

该函数的作用是将src表示的AVCodecContext中的内容拷贝到dest中。

随后，调用avio\_open函数打开输出文件：

av\_dump\_format(ofmt\_ctx, 0, io\_param.outputName, 1);

if (!(ofmt->flags & AVFMT\_NOFILE))

{

ret = avio\_open(&ofmt\_ctx->pb, io\_param.outputName, AVIO\_FLAG\_WRITE);

if (ret < 0)

{

printf("Error: Could not open output file.\n");

goto end;

}

}

4、写入文件的音视频数据

首先向输出文件中写入文件头:

ret = avformat\_write\_header(ofmt\_ctx, NULL);

if (ret < 0)

{

printf("Error: Could not write output file header.\n");

goto end;

}

写入文件的视频和音频包数据，其实就是将音频和视频Packets从输入文件中读出来，正确设置pts和dts等时间量之后，再写入到输出文件中去：

while (1)

{

AVStream \*in\_stream, \*out\_stream;

ret = av\_read\_frame(ifmt\_ctx, &pkt);

if (ret < 0)

break;

in\_stream = ifmt\_ctx->streams[pkt.stream\_index];

out\_stream = ofmt\_ctx->streams[pkt.stream\_index];

/\* copy packet \*/

pkt.pts = av\_rescale\_q\_rnd(pkt.pts, in\_stream->time\_base, out\_stream->time\_base, (AVRounding)(AV\_ROUND\_NEAR\_INF|AV\_ROUND\_PASS\_MINMAX));

pkt.dts = av\_rescale\_q\_rnd(pkt.dts, in\_stream->time\_base, out\_stream->time\_base, (AVRounding)(AV\_ROUND\_NEAR\_INF|AV\_ROUND\_PASS\_MINMAX));

pkt.duration = av\_rescale\_q(pkt.duration, in\_stream->time\_base, out\_stream->time\_base);

pkt.pos = -1;

ret = av\_interleaved\_write\_frame(ofmt\_ctx, &pkt);

if (ret < 0)

{

fprintf(stderr, "Error muxing packet\n");

break;

}

av\_free\_packet(&pkt);

}

最后要做的就是写入文件尾：

av\_write\_trailer(ofmt\_ctx);

5、 收尾工作

写入输出文件完成后，需要对打开的结构进行关闭或释放等操作。主要有关闭输入输出文件、释放输出文件的句柄等：

avformat\_close\_input(&ifmt\_ctx);

/\* close output \*/

if (ofmt\_ctx && !(ofmt->flags & AVFMT\_NOFILE))

avio\_closep(&ofmt\_ctx->pb);

avformat\_free\_context(ofmt\_ctx);

if (ret < 0 && ret != AVERROR\_EOF)

{

fprintf(stderr, "Error failed to write packet to output file.\n");

return 1;

}

##七、 FFMpeg实现视频水印

视频的水印通常指附加在原始视频上的可见或者不可见的，与原始视频无直接关联的标识。通常在有线电视画面上电视台的台标以及视频网站上的logo就是典型的视频水印的应用场景。通常实现视频水印可以通过FFMpeg提供的libavfilter库实现。libavfilter库实际上实现的是视频的滤镜功能，除了水印之外，还可以实现视频帧的灰度化、平滑、翻转、直方图均衡、裁剪等操作。

我们这里实现的视频水印等操作，完全在视频像素域实现，即从一个yuv文件中读取数据到AVFrame结构，对AVFrame结构进行处理后再输出到另一个yuv文件。中间不涉及封装或编码解码等操作。

###1. 解析命令行，获取输入输出文件信息

我们通过与之前类似的方式，在命令行中获取输入、输出文件名，图像宽高。首先定义如下的结构体用于保存配置信息：

typedef struct \_IOFiles

{

const char \*inputFileName; //输入文件名

const char \*outputFileName; //输出文件名

FILE \*iFile; //输入文件指针

FILE \*oFile; //输出文件指针

uint8\_t filterIdx; //Filter索引

unsigned int frameWidth; //图像宽度

unsigned int frameHeight; //图像高度

}IOFiles;

在这个结构体中，filterIdx用于表示当前工程选择哪一种filter，即希望实现哪一种功能。

在进入main函数之后，调用hello函数来解析命令行参数：

static int hello(int argc, char \*\*argv, IOFiles &files)

{

if (argc < 4)

{

printf("usage: %s output\_file input\_file filter\_index\n"

"Filter index:.\n"

"1. Color component\n"

"2. Blur\n"

"3. Horizonal flip\n"

"4. HUE\n"

"5. Crop\n"

"6. Box\n"

"7. Text\n"

"\n", argv[0]);

return -1;

}

files.inputFileName = argv[1];

files.outputFileName = argv[2];

files.frameWidth = atoi(argv[3]);

files.frameHeight = atoi(argv[4]);

files.filterIdx = atoi(argv[5]);

fopen\_s(&files.iFile, files.inputFileName, "rb+");

if (!files.iFile)

{

printf("Error: open input file failed.\n");

return -1;

}

fopen\_s(&files.oFile, files.outputFileName, "wb+");

if (!files.oFile)

{

printf("Error: open output file failed.\n");

return -1;

}

return 0;

}

该函数实现了输入输出文件的文件名获取并打开，并读取filter索引。

###2. Video Filter初始化

在进行初始化之前，必须调用filter的init函数，之后才能针对Video Filter进行各种操作。其声明如下：

void avfilter\_register\_all(void);

为了实现视频水印的功能，所需要的相关结构主要有：

AVFilterContext \*buffersink\_ctx;

AVFilterContext \*buffersrc\_ctx;

AVFilterGraph \*filter\_graph;

其中AVFilterContext用于表示一个filter的实例上下文，AVFilterGraph表示一个video filtering的工作流。Video Filter的初始化实现如以下函数：

//初始化video filter相关的结构

int Init\_video\_filter(const char \*filter\_descr, int width, int height)

{

char args[512];

AVFilter \*buffersrc = avfilter\_get\_by\_name("buffer");

AVFilter \*buffersink = avfilter\_get\_by\_name("buffersink");

AVFilterInOut \*outputs = avfilter\_inout\_alloc();

AVFilterInOut \*inputs = avfilter\_inout\_alloc();

enum AVPixelFormat pix\_fmts[] = { AV\_PIX\_FMT\_YUV420P, AV\_PIX\_FMT\_NONE };

AVBufferSinkParams \*buffersink\_params;

filter\_graph = avfilter\_graph\_alloc();

/\* buffer video source: the decoded frames from the decoder will be inserted here. \*/

snprintf(args, sizeof(args), "video\_size=%dx%d:pix\_fmt=%d:time\_base=%d/%d:pixel\_aspect=%d/%d", width,height,AV\_PIX\_FMT\_YUV420P, 1, 25,1,1);

int ret = avfilter\_graph\_create\_filter(&buffersrc\_ctx, buffersrc, "in", args, NULL, filter\_graph);

if (ret < 0)

{

printf("Error: cannot create buffer source.\n");

return ret;

}

/\* buffer video sink: to terminate the filter chain. \*/

buffersink\_params = av\_buffersink\_params\_alloc();

buffersink\_params->pixel\_fmts = pix\_fmts;

ret = avfilter\_graph\_create\_filter(&buffersink\_ctx, buffersink, "out", NULL, buffersink\_params, filter\_graph);

av\_free(buffersink\_params);

if (ret < 0)

{

printf("Error: cannot create buffer sink\n");

return ret;

}

/\* Endpoints for the filter graph. \*/

outputs->name = av\_strdup("in");

outputs->filter\_ctx = buffersrc\_ctx;

outputs->pad\_idx = 0;

outputs->next = NULL;

inputs->name = av\_strdup("out");

inputs->filter\_ctx = buffersink\_ctx;

inputs->pad\_idx = 0;

inputs->next = NULL;

if ((ret = avfilter\_graph\_parse\_ptr(filter\_graph, filter\_descr, &inputs, &outputs, NULL)) < 0)

{

printf("Error: avfilter\_graph\_parse\_ptr failed.\n");

return ret;

}

if ((ret = avfilter\_graph\_config(filter\_graph, NULL)) < 0)

{

printf("Error: avfilter\_graph\_config");

return ret;

}

return 0;

}

###3. 初始化输入输出AVFrame并分配内存

我们首先声明AVFrame类型的对象和指向像素缓存的指针：

AVFrame \*frame\_in = NULL;

AVFrame \*frame\_out = NULL;

unsigned char \*frame\_buffer\_in = NULL;

unsigned char \*frame\_buffer\_out = NULL;

然后分配AVFrame对象，并分配其中的缓存区：

void Init\_video\_frame\_in\_out(AVFrame \*\*frameIn, AVFrame \*\*frameOut, unsigned char \*\*frame\_buffer\_in, unsigned char \*\*frame\_buffer\_out, int frameWidth, int frameHeight)

{

\*frameIn = av\_frame\_alloc();

\*frame\_buffer\_in = (unsigned char \*)av\_malloc(av\_image\_get\_buffer\_size(AV\_PIX\_FMT\_YUV420P, frameWidth,frameHeight,1));

av\_image\_fill\_arrays((\*frameIn)->data, (\*frameIn)->linesize,\*frame\_buffer\_in, AV\_PIX\_FMT\_YUV420P,frameWidth,frameHeight,1);

\*frameOut = av\_frame\_alloc();

\*frame\_buffer\_out = (unsigned char \*)av\_malloc(av\_image\_get\_buffer\_size(AV\_PIX\_FMT\_YUV420P, frameWidth,frameHeight,1));

av\_image\_fill\_arrays((\*frameOut)->data, (\*frameOut)->linesize,\*frame\_buffer\_out, AV\_PIX\_FMT\_YUV420P,frameWidth,frameHeight,1);

(\*frameIn)->width = frameWidth;

(\*frameIn)->height = frameHeight;

(\*frameIn)->format = AV\_PIX\_FMT\_YUV420P;

}

###4. Video Filtering循环体

这一部分主要包括三大部分：

读取原始的YUV数据到输入的frame；

使用预先定义好的filter\_graph处理输入frame，生成输出frame；

将输出frame中的像素值写入输出yuv文件；

第一部分，读取原始yuv的实现由自定义函数Read\_yuv\_data\_to\_buf实现：

//从输入yuv文件中读取数据到buffer和frame结构

bool Read\_yuv\_data\_to\_buf(unsigned char \*frame\_buffer\_in, const IOFiles &files, AVFrame \*\*frameIn)

{

AVFrame \*pFrameIn = \*frameIn;

int width = files.frameWidth, height = files.frameHeight;

int frameSize = width \* height \* 3 / 2;

if (fread\_s(frame\_buffer\_in, frameSize, 1, frameSize, files.iFile) != frameSize)

{

return false;

}

pFrameIn->data[0] = frame\_buffer\_in;

pFrameIn->data[1] = pFrameIn->data[0] + width \* height;

pFrameIn->data[2] = pFrameIn->data[1] + width \* height / 4;

return true;

}

第二部分实际上分为两部分，即将输入frame送入filter graph，以及从filter graph中取出输出frame。实现方法分别为：

//将待处理的输入frame添加进filter graph

bool Add\_frame\_to\_filter(AVFrame \*frameIn)

{

if (av\_buffersrc\_add\_frame(buffersrc\_ctx, frameIn) < 0)

{

return false;

}

return true;

}

//从filter graph中获取输出frame

int Get\_frame\_from\_filter(AVFrame \*\*frameOut)

{

if (av\_buffersink\_get\_frame(buffersink\_ctx, \*frameOut) < 0)

{

return false;

}

return true;

}

第三部分，写出输出frame到输出yuv文件：

//从输出frame中写出像素数据到输出文件

void Write\_yuv\_to\_outfile(const AVFrame \*frame\_out, IOFiles &files)

{

if(frame\_out->format==AV\_PIX\_FMT\_YUV420P)

{

for(int i=0;i<frame\_out->height;i++)

{

fwrite(frame\_out->data[0]+frame\_out->linesize[0]\*i,1,frame\_out->width,files.oFile);

}

for(int i=0;i<frame\_out->height/2;i++)

{

fwrite(frame\_out->data[1]+frame\_out->linesize[1]\*i,1,frame\_out->width/2,files.oFile);

}

for(int i=0;i<frame\_out->height/2;i++)

{

fwrite(frame\_out->data[2]+frame\_out->linesize[2]\*i,1,frame\_out->width/2,files.oFile);

}

}

}

该部分的综合实现如下：

while (Read\_yuv\_data\_to\_buf(frame\_buffer\_in, files, &frame\_in))

{

//将输入frame添加到filter graph

if (!Add\_frame\_to\_filter(frame\_in))

{

printf("Error while adding frame.\n");

goto end;

}

//从filter graph中获取输出frame

if (!Get\_frame\_from\_filter(&frame\_out))

{

printf("Error while getting frame.\n");

goto end;

}

//将输出frame写出到输出文件

Write\_yuv\_to\_outfile(frame\_out, files);

printf("Process 1 frame!\n");

av\_frame\_unref(frame\_out);

}

###5、 收尾工作

整体实现完成后，需要进行善后的收尾工作有释放输入和输出frame、关闭输入输出文件，以及释放filter graph：

//关闭文件及相关结构

fclose(files.iFile);

fclose(files.oFile);

av\_frame\_free(&frame\_in);

av\_frame\_free(&frame\_out);

avfilter\_graph\_free(&filter\_graph);

八、 FFMpeg实现视频缩放

视频缩放是视频开发中一项最基本的功能。通过对视频的像素数据进行采样或插值，可以将低分辨率的视频转换到更高的分辨率，或者将高分辨率的视频转换为更低的分辨率。通过FFMpeg提供了libswscale库，可以轻松实现视频的分辨率转换功能。除此之外，libswscale库还可以实现颜色空间转换等功能。

通常情况下视频缩放的主要思想是对视频进行解码到像素域后，针对像素域的像素值进行采样或差值操作。这种方式需要在解码端消耗一定时间，但是通用性最好，不需要对码流格式作出任何特殊处理。在FFMpeg中libswscale库也是针对AVFrame结构进行缩放处理。

1. 解析命令行参数

输入输出的数据使用以下结构进行封装：

typedef struct \_IOFiles

{

char \*inputName; //输入文件名

char \*outputName; //输出文件名

char \*inputFrameSize; //输入图像的尺寸

char \*outputFrameSize; //输出图像的尺寸

FILE \*iFile; //输入文件指针

FILE \*oFile; //输出文件指针

} IOFiles;

输入参数解析过程为：

static bool hello(int argc, char \*\*argv, IOFiles &files)

{

printf("FFMpeg Scaling Demo.\nCommand format: %s input\_file input\_frame\_size output\_file output\_frame\_size\n", argv[0]);

if (argc != 5)

{

printf("Error: command line error, please re-check.\n");

return false;

}

files.inputName = argv[1];

files.inputFrameSize = argv[2];

files.outputName = argv[3];

files.outputFrameSize = argv[4];

fopen\_s(&files.iFile, files.inputName, "rb+");

if (!files.iFile)

{

printf("Error: cannot open input file.\n");

return false;

}

fopen\_s(&files.oFile, files.outputName, "wb+");

if (!files.oFile)

{

printf("Error: cannot open output file.\n");

return false;

}

return true;

}

在参数读入完成后，需要从表示视频分辨率的字符串中解析出图像的宽和高两个值。我们在命令行中传入的视频分辨率字符串的格式为“width x height”，例如"720x480"。解析过程需要调用av\_parse\_video\_size函数。声明如下：

int av\_parse\_video\_size(int \*width\_ptr, int \*height\_ptr, const char \*str);

例如，我们传入下面的参数：

int frameWidth, frameHeight;

av\_parse\_video\_size(&frameWidth, &frameHeight, "720x480");

函数将分别把720和480传入frameWidth和frameHeight中。

在获取命令行参数后，调用该函数解析图像分辨率：

int srcWidth, srcHeight, dstWidth, dstHeight;

if (av\_parse\_video\_size(&srcWidth, &srcHeight, files.inputFrameSize))

{

printf("Error: parsing input size failed.\n");

goto end;

}

if (av\_parse\_video\_size(&dstWidth, &dstHeight, files.outputFrameSize))

{

printf("Error: parsing output size failed.\n");

goto end;

}

这样，我们就获得了源和目标图像的宽和高度。

2. 创建SwsContext结构

进行视频的缩放操作离不开libswscale的一个关键的结构，即SwsContext，该结构提供了缩放操作的必要参数。创建该结构需调用函数sws\_getContext。该函数的声明如下：

struct SwsContext \*sws\_getContext(int srcW, int srcH, enum AVPixelFormat srcFormat,

int dstW, int dstH, enum AVPixelFormat dstFormat,

int flags, SwsFilter \*srcFilter,

SwsFilter \*dstFilter, const double \*param);

该函数的前两行参数分别表示输入和输出图像的宽、高、像素格式，参数flags表示采样和差值使用的算法，常用的有SWS\_BILINEAR表示双线性差值等。剩余的不常用参数通常设为NULL。创建该结构的代码如：

//创建SwsContext结构

enum AVPixelFormat src\_pix\_fmt = AV\_PIX\_FMT\_YUV420P;

enum AVPixelFormat dst\_pix\_fmt = AV\_PIX\_FMT\_YUV420P;

struct SwsContext \*sws\_ctx = sws\_getContext(srcWidth, srcHeight, src\_pix\_fmt, dstWidth, dstHeight, dst\_pix\_fmt, SWS\_BILINEAR, NULL,NULL,NULL );

if (!sws\_ctx)

{

printf("Error: parsing output size failed.\n");

goto end;

}

3. 分配像素缓存

视频缩放实际上是在像素域实现，但是实际上我们没有必要真的建立一个个AVFrame对象，我们只需要其像素缓存空间即可，我们定义两个指针数组和两个保存stride的数组，并为其分配内存区域：

//分配input和output

uint8\_t \*src\_data[4], \*dst\_data[4];

int src\_linesize[4], dst\_linesize[4];

if ((ret = av\_image\_alloc(src\_data, src\_linesize, srcWidth, srcHeight, src\_pix\_fmt, 32)) < 0)

{

printf("Error: allocating src image failed.\n");

goto end;

}

if ((ret = av\_image\_alloc(dst\_data, dst\_linesize, dstWidth, dstHeight, dst\_pix\_fmt, 1)) < 0)

{

printf("Error: allocating dst image failed.\n");

goto end;

}

4. 循环处理输入frame

循环处理的代码为：

//从输出frame中写出到输出文件

int dst\_bufsize = ret;

for (int idx = 0; idx < MAX\_FRAME\_NUM; idx++)

{

read\_yuv\_from\_ifile(src\_data, src\_linesize, srcWidth, srcHeight, 0, files);

read\_yuv\_from\_ifile(src\_data, src\_linesize, srcWidth, srcHeight, 1, files);

read\_yuv\_from\_ifile(src\_data, src\_linesize, srcWidth, srcHeight, 2, files);

sws\_scale(sws\_ctx, (const uint8\_t \* const\*)src\_data, src\_linesize, 0, srcHeight, dst\_data, dst\_linesize);

fwrite(dst\_data[0], 1, dst\_bufsize, files.oFile);

}

其核心函数为sws\_scale，其声明为：

int sws\_scale(struct SwsContext \*c, const uint8\_t \*const srcSlice[],

const int srcStride[], int srcSliceY, int srcSliceH,

uint8\_t \*const dst[], const int dstStride[]);

该函数的各个参数比较容易理解，除了第一个是之前创建的SwsContext之外，其他基本上都是源和目标图像的缓存区和大小等。在写完一帧后，调用fwrite将输出的目标图像写入输出yuv文件中。

5. 收尾工作

收尾工作除了释放缓存区和关闭输入输出文件之外，就是需要释放SwsContext结构，需调用函数：sws\_freeContext。实现过程为：

fclose(files.iFile);

fclose(files.oFile);

av\_freep(&src\_data[0]);

av\_freep(&dst\_data[0]);

sws\_freeContext(sws\_ctx);