迄今为止最全面的介绍 FFmpeg/Libav 开源库丛书

多媒体编程开发之
FFmpeg 基础库

多媒体开发核心工具

（第一版 试读修订）

刘诗萌 著

2013/11/13

邮件地址：109117198@qq.com
联系电话：13520647302

2017 年 5 月 4 日百度文库群共享可以公开下载

FFmpeg 技术交流 QQ 群： 28785698 交流论坛：bbs.chinaffmpeg.com

引言

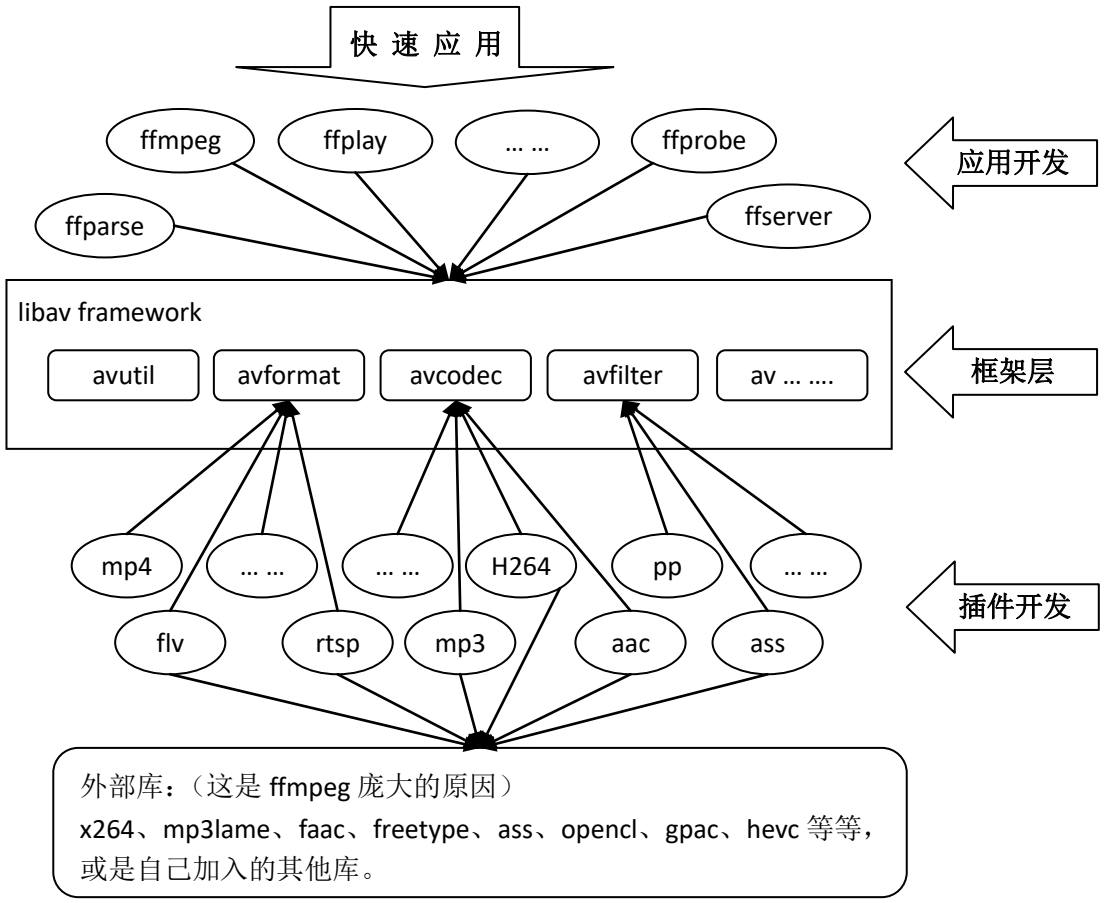
关于 FFmpeg 是多媒体程序核心开源库，涉及广播电视行业、视频监控行业、互联网视频行业等，涉及流媒体、音视频编解码、容器、格式转换、以及后期处理等技术。

本书旨在让更多的人深入理解 FFmpeg 框架库，通过使用和学习它应用到各行业的软件开发当中，快速、安全的构建自己的多媒体软件程序。

FFmpeg 可以是你学习多媒体程序开发的入口点，由此扩展到流媒体、编解码、播放器、媒体服务器端等等专业项目当中，是学习多媒体的最好的入门程序，同时也是多媒体程序的底层基础工具库。

如果你熟悉 FFmpeg 可以快速开发一个播放器、转码器、以及流媒体服务器等应用程序。

FFmpeg 之所以是很好的多媒体开发的入门程序，因为它的开发人员也都慢慢成为其他开源工程的核心工作者，例如从 FFmpeg 开发者中分离到 x264、lamemp3、live555、aac、hevc、librtmp、sip、mplay、vlc、gpac、flv、preprocessing、postprocessing 等等。



根据上图 FFmpeg 开发框架层次图就会明白，至于学习 FFmpeg 可以不必全部了解，主要学习涉及自己的部分即可。

快速应用是用户通过命令行等参数启动 FFmpeg 库默认应用程序，可以实现转码、播放器、媒体信息、流服务器功能；而应用开发是开发人员通过应用层函数，实现各种应用功能 mplayer、vlc、ffdshow 等等都是实现样例；框架层修改一般有 FFmpeg 开发人员更新修改；插件开发一般可以由开发人员自己定制。

至于学习方法，作者没法不让你走偏路，可以让你看清每条路，给你更多的选择，做好自己的选择，坚持下去。

对于学习的态度，我认为，不要为了学习而学习，也不要为了工作而工作，没用心往往是事倍功半；子曰:“知之者不如好之者，好知者不如乐知者。” 其实我们不需要太多兴趣，只要一点点就行，只要能从“知之者”身份变为“好之者”，工作上就搓搓有余了，正所谓干一行爱一行嘛。

其中了解行业发展也很重要，这个紧密关系到自己的未来、以及就业和发展方向；而特别是想成为“伟大的科技公司”或是“技术大牛”的，就要有更敏锐的眼光、更深刻的洞悉力和更长远战略发展，正所谓知己知彼百战百胜，了解他人学习他人的优点，且不要“闭关锁国，闭门造车”，因为了解不需要太多机会和代价，只需要用心，真的想去了解。

关于多媒体技术行业是一个长期的投入，新的概念、新的标准、新的需求每时每刻都在更新，为了让自己的播放器、流媒体服务器、转码工具等等能正常运行，是需要有开发人员进行不断的维护更新、添加功能、版本迭代的。

关于本书的代码风格，本书粘贴的很多代码都是通过使用 VS 粘贴到 Word 上的，为了保持风格一致性，一些代码是从别的 IDE 上先粘贴到 VS 上后粘贴到 Word 上编辑的。

看标准文档和书一定要先找到重点，学习不是为了考试，而是为了实践，编写出更好的程序，而标准文档对于大多数开发人员来说更像是字典类工具，当然需要先了解基本的东西，理解铭记于心，更细节的不常用的东西可以通过工具来弥补。

关于书的版本迭代问题，我并没有打算一次性把书写的大而全、专而精，我想通过软件开发一样，从不断的学习、努力当中不断完善，这个版本开发编写也是不断积累迭代出来的。

理性的追求完美，坚持中得到改变。

-------- lsm--------

目录

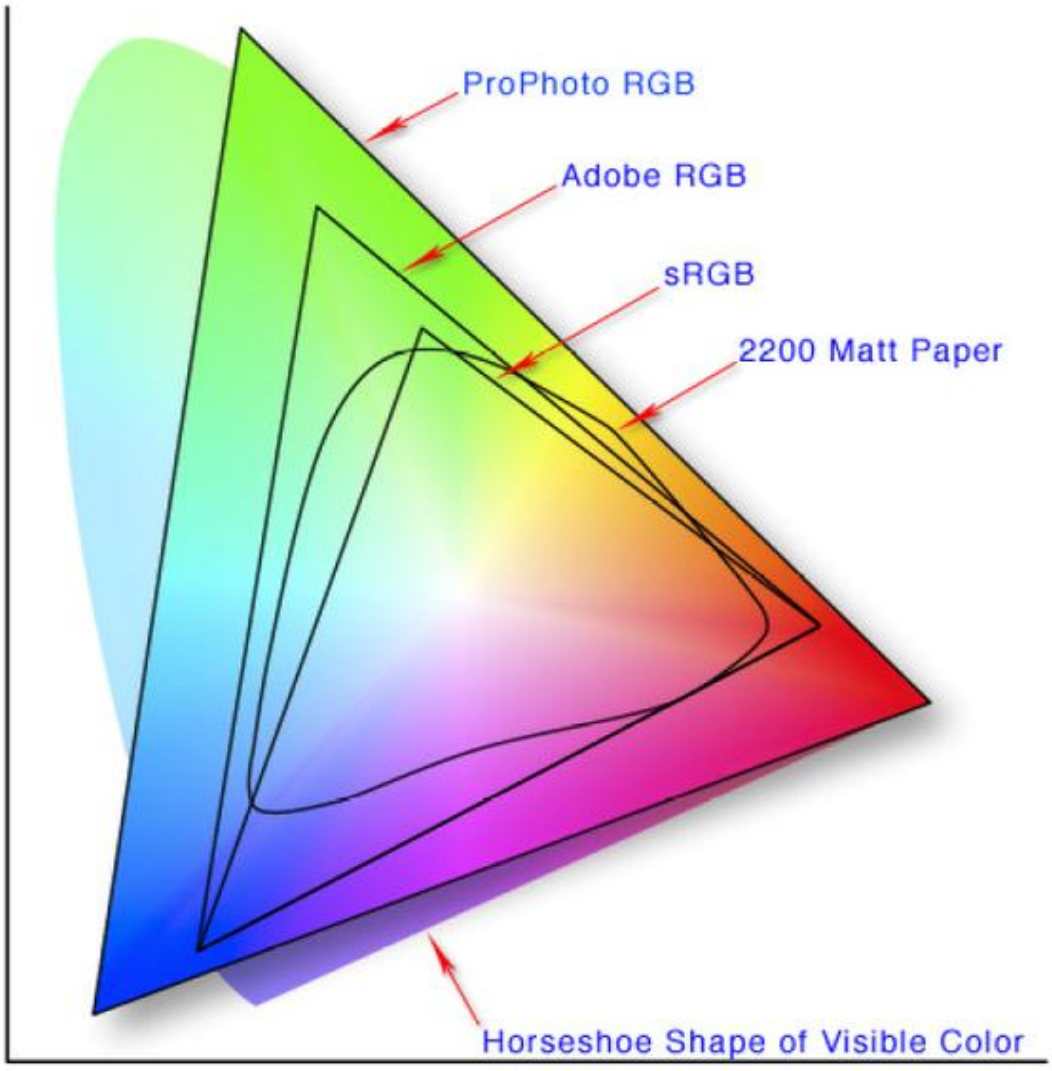
第一篇 多媒体概念介绍

1. 视频原始数据格式

1 色彩空间

一种颜色的表式模型是用一组数值分量表示的，通常是由一个、三个或四个颜色值分量组成（例如：RGB 和 CMYK 色彩空间），为了方便于人的直观理解，利用坐标系的形式表现出来，分量组数就变成了维度，由一维、三维或是四维坐标分量分别表示各个颜色的数值分量，它是一种抽象的数学模型。

如下图是一些常用色彩空间的表示范围：



1) Gray 灰度模式

从小我们就看过黑白电视机，它就是 Gray 灰度模式，8bit 位深的一维分量表示，其数值的取值范围为 0~255 表示明暗度，0 是最暗即是黑色，255 代表最亮也就是白色。

如果一个 320x240 图像大小的 Gray 图像，每一个像素由 8bit 位深的数值表示，那么一个灰度图像最少所占存储空间大小为 320x240x8/8 = 76800 字节。

如下图是 Gray 色彩表示范围：

2) YUV 色彩空间

YUV 色彩空间（也叫 YCrCb），主要用于优化彩色视频信号的传输，使其向后相容老式黑白电视而提出的。

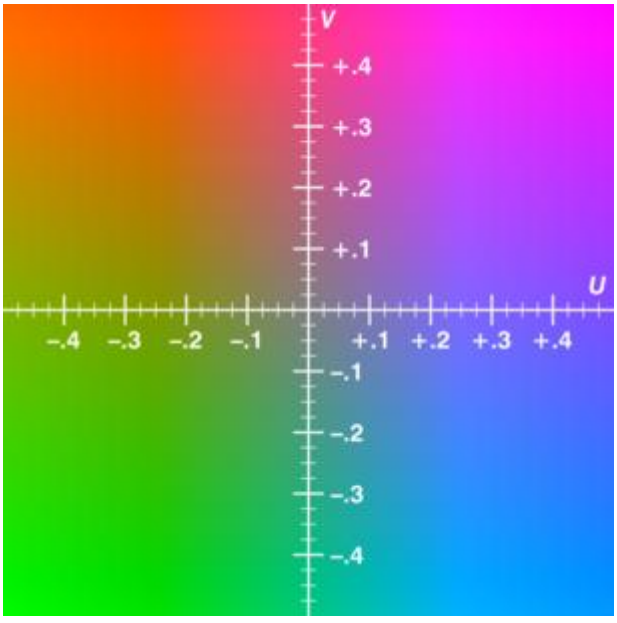
其中 Y 分量就是代表明亮度（Luma），也就相当于 Gray 灰度，一般也是由 8bit（1 字节）位深的一维分量表示。

注意：这里黑白电视的灰度显示图像就是 YCrCb 中 Y 的显示图像，但是绝对不是 YCbCr 或是 RGB 计算出来灰度图像，这个灰度图是用于图像处理，它描画的黑白图像显示纹理更加细致，当然这个通过一定计算获得的，需要耗费一定计算工作量。

uv 分量共同组成的色度，“U”和“V”分别表示色度和饱和度（Chrominance、Chroma），一般也是由 8bit（1 字节）位深分别表示。所以单独一个像素的 YUV 数值表示需要 3*8/8 = 3 字节大小。

注意：如果 Y 值是最暗即为黑色，相当于没有光的情况；如果 Y 值是最亮即为白色，相当于全部的光都照射进来一样。

如下图是 YUV 中 UV 分量数值分布的平面图，其中 Y 分量值为 0.5：



由于人眼的视觉特性，即是人眼对于明亮度分辨能力要敏感于对色度的分辨能力，所以就可以降低色度的有效数据量从而降低了原始数据总量。

常用的 YUV 彩色格式为 YUV4:2:0、YUV4:2:2、YUV4:4:4、YUV4:1:1，下面还以 320x240 的图像大小为例，来分别详细介绍各个采样格式区别。

i. YUV4:2:0 格式

表示 4 比 2 比 0 的 YUV 取样，水平每 2 个像素和垂直每 2 个像素（即 2x2 的 4 个像素）中 Y 取样 4 个，U 取样 1 个，V 取样 1 个。

所以每 2x2 个像素 Y 占有 4 个字节，U 占有 1 个字节，V 占有 1 个字节，平均 YUV420 每个像素所占(4+1+1)*8bit/4pix = 12bpp。

那 320x240 的图像大小，所占用总字节数为 320x240x12bit/8bit = 115200 字节。

ii. YUV4:1:1 格式

表示 4 比 1 比 1 的 YUV 取样，水平每 4 个像素（即 4x1 的 4 个像素）中 Y 取样 4 个，U 取样 1 个，V 取样 1 个。

所以每 4x1 个像素 Y 占有 4 个字节，U 占有 1 个字节，V 占有 1 个字节，平均 YUV411 每个像素所占(4+1+1)*8bit/4pix = 12bpp。

那 320x240 的图像大小，所占用总字节数为 320x240x12bit/8bit = 115200 字节。

iii. YUV4:2:2 格式

表示 4 比 2 比 2 的 YUV 取样，水平每 2 个像素（即 2x1 的 2 个像素）中 Y 取样 2 个，U 取样 1 个，V 取样 1 个。

所以每 2x1 个像素 Y 占有 2 个字节，U 占有 1 个字节，V 占有 1 个字节，平均 YUV422 每个像素所占(2+1+1)*8bit/2pix = 16bpp。

那 320x240 的图像大小，所占用总字节数为 320x240x16bit/8bit = 153600 字节。

iv. YUV4:4:4 格式

表示 4 比 4 比 4 的 YUV 取样，水平每 1 个像素（即 1x1 的 1 个像素）中 Y 取样 1 个，U 取样 1 个，V 取样 1 个。

所以每 1x1 个像素 Y 占有 1 个字节，U 占有 1 个字节，V 占有 1 个字节，平均 YUV444 每个像素所占(1+1+1)*8bit/1pix = 24bpp。

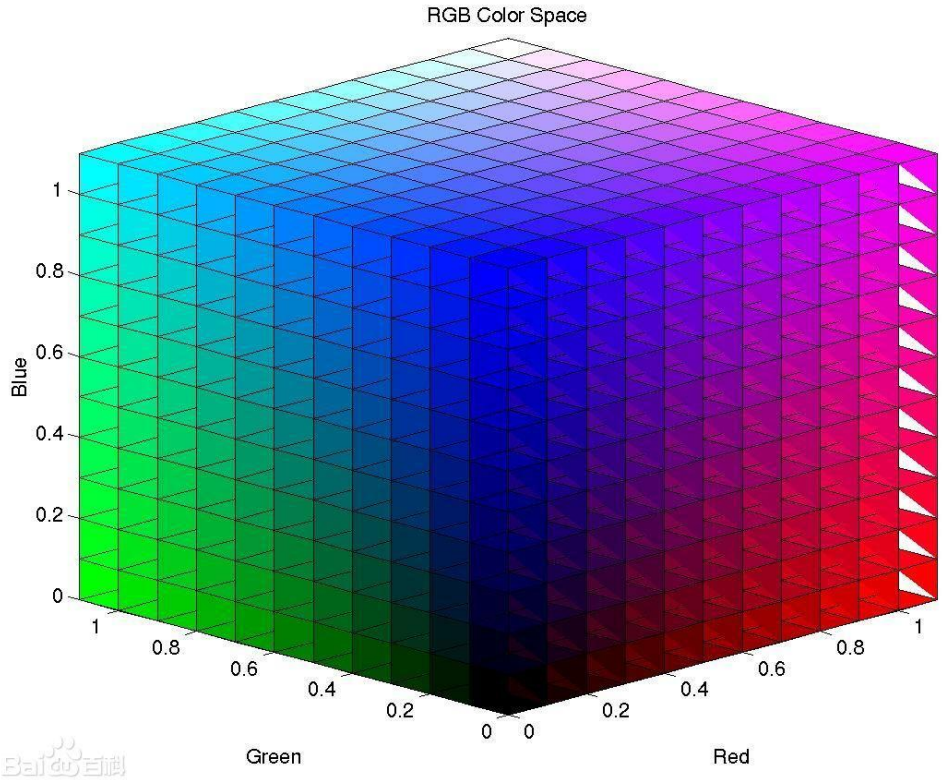
那 320x240 的图像大小，所占用总字节数为 320x240x16bit/8bit = 230400 字节。

3) RGB 色彩空间

RGB 色彩（也叫做三原色光），RGB 分别表示为红（Red）、绿（Green）、蓝（Blue），三原色光以不同的比例相加，以产生多种多样的色光。三原色的原理不是出于物理原因，而是由于人眼的生理原因造成的。

下图是三原色光投影效果图：

下面是 RGB 色彩分布直方图：



i. RGB24 格式，默认 RGB 格式，每个像素占用 24bit 即为 3 个字节，R、G、B 各所占 1 个字节。

ii. RGB32 格式，每个像素占用 32bit 即为 4 个字节，R、G、B 各所占 1 个字节，多余一个字节为垃圾数据。

iii. RGBA 格式，每个像素占用 32bit 即为 4 个字节，R、G、B、A 各所占 1 个字节，A 表示透明度 Alpha。

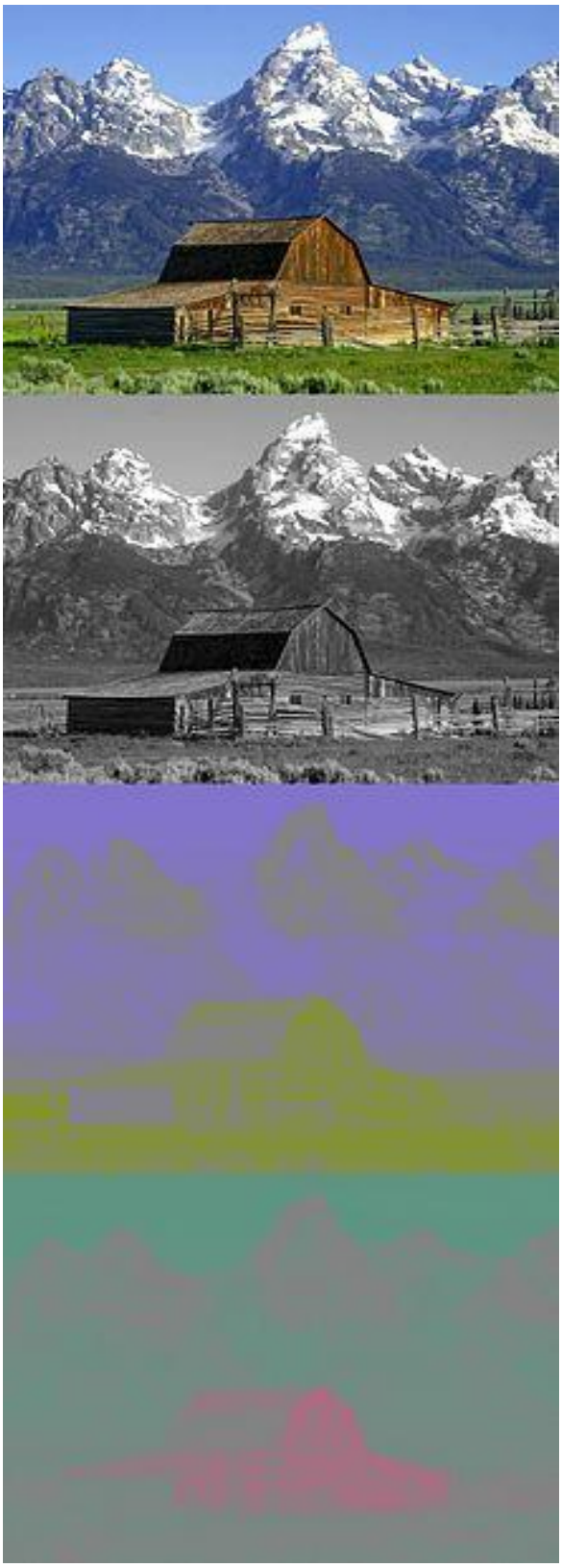
4) YCC 色彩空间

YCC 色彩空间（也叫 YCbCr、YPbPr），YCbCr 通用用于视频处理，多数是通过 RGB 计算获得。

Y 仍然代表的是亮度，在同一个复杂场景的画面中，YUV 和 YCC 色彩空间下的 Y 平面的显示对比，YCC 的 Y 能显示更多的画面细节，所以才用于图像处理（例如：物体识别、行为分析等）。

而 Cb 和 Cr 分别表示为蓝色和红色的浓度偏移量，通过左右拉伸表示其他颜色，也有 YUV420、YUV411、YUV422、YUV444 之分。

下图是 YCbCr 组合和分别显示图：



5) 扫描方式

YUV 图像的扫描方式分为逐行扫描（Progressive）和隔行扫描（Interlaced）两种，都分为存储、传输和显示三个部分，概念相通而不相同。

逐行扫描很简单，现在的液晶显示器都是逐行扫描，存储、传输和显示都是一行一行的处理，没有什么特别之处（在 ffmpeg 中 YUV420P，中 P 代表逐行扫描的意思）。

隔行扫描用于是旧式电视和 CRT 显示器中。首先 YUV 隔行存储，YUV 作为 3 个独立平面来处理，每个平面中每隔一行抽出，由奇数行组成顶场（或上场），有偶数行组成底场（下场）分开存储和传输，显示例如 CRT 显示器分 2 次显示上下两场，60Hz 的显示器其最大 FPS 是 30，每幅画面扫描 2 次，60 分之一秒内一般人眼很难感觉到亮度的衰减。

6) 位深

正常色彩空间默认深度为 8bit，也有 9bit、10bit、12bit 和 16bit，最常用的是 8bit，即一个字节代表一个像素值，因为一般人眼仅能识别出 265 位的色彩梯度，足够还原其真实感，8bit 位深也有另外一个称呼，一般叫 256 位真彩色（暗指 RGB 色彩空间的 8bit 位深）。

9bit、10bit 常用于编解码处理，因为现在计算机体系结构计算单元、存储单元最小都是整字节对齐，所以在程序开发时候 9bit、10bit、12bit、16bit 每一个像素都会占用 2 个字节空间，16bit 也被称为 65536 位真彩色（也暗指 RGB 色彩空间的位深）。

2. 音频原始数据格式

原始音频数据格式，没有压缩的 PCM 就是 WAV 原始波形数据。
原始音视数据格式参数：

1) 采样频率

在数字音频领域，常用的采样率有：

采样频率	常用用途
8,000 Hz	电话所用采样率, 对于人的说话已经足够
11,025 Hz - 22,050 Hz	无线电广播所用采样率
32,000 Hz	miniDV 数码视频 camcorder、DAT (LP mode) 所用采样率
44,100 Hz	音频 CD, 也常用于 MPEG-1 音频（VCD, SVCD, MP3）所用采样率
47,250 Hz	Nippon Columbia (Denon)开发的世界上第一个商用 PCM 录音机所用采样率
48,000 Hz	miniDV、数字电视、DVD、DAT、电影和专业音频所用的数字声音所用采样率
50,000 Hz	二十世纪七十年代后期出现的 3M 和 Soundstream 开发的第一款商用数字录音机所用采样率
50,400 Hz	三菱 X-80 数字录音机所用所用采样率
96,000 或者 192,000 Hz	DVD-Audio、一些 LPCM DVD 音轨、BD-ROM（蓝光盘）音轨、和 HD-DVD（高清晰度 DVD）音轨所用所用采样率
2.8224 MHz	SACD、索尼 和 飞利浦 联合开发的称为 Direct Stream Digital 的 1 位 sigma-delta modulation 过程所用采样率

2) 采样深度

下表列举出常用的采样深度类型：

单位	字节数	有无符号
Char	单字节	有符号
Unsigned char	单字节	无符号
Short	双字节	有符号
Unsigned short	双字节	无符号
Float	四字节	浮点数
Int	四字节	有符号
Unsigned int	四字节	无符号
Double	八字节	双精度

除了综上所述，再加上字节序，大端或是小端字节序就是最完整的的采样深度格式。

3) 声道数和声道布局

下表列举出常用的声道类型：

声道名称	声道数	声道布局
单声道	1	正前方居中
立体声（双声道）	2	左右声道，有关资料离正前方各 60°，这个要看个人喜好。
2.1 声道	3	左右声道+低音
3.0 声道	3	左右声道+正后方居中
环绕 3.0	3	左右声道+正前方居中
环绕 3.1	4	环绕 3.0+低音
环绕 4.0	4	左右声道+正前方居中+正后方居中
环绕 4.1	5	环绕 4.0+低音
环绕 2-2	4	立体声+正左边+正右面
环绕 4	4	立体声+左右后面，左右各离正左右面 20°夹角
环绕 5.0	5	环绕 3.0+正左边+正右面
环绕 5.1	6	环绕 5.0 + 低音
环绕 5.0 后	5	环绕 4+正前方居中
环绕 5.1 后	6	环绕 5.0 后+低音
环绕 6.0	6	环绕 5.0 + 正后方居中
环绕 6.0 前	7	……
……	……	……

4) ADPCM

ADPCM 是自适应差分脉冲编码调制的简称，它意义和 YUV 格式一样，在原始音频 PCM 的基础上做可以接受的数据量降低。其原理是音乐和语音都有自己的最大和最小频率，所以在两个采样点之间的部分数据量是可以丢失的，因为不会太影响音质，也是 PCM 类编码的原理所在。

3. 字幕原始数据格式

字幕的原始数据主要分为 2 种，图形图像数据和文字数据，例如 PSG 就是图形图像数据，srt 就是文字数据。图形图像数据需要渲染贴图，文字数据需要先从字体文件中渲染为图形图像数据再渲染贴图在视频源上达到效果。

文字数据例如：我的字幕，我精彩。

图像数据：8bit 位深，宽高是 32*32 的内存数据，来表示一个文字的黑白图像。

4. 采集录制和播放渲染

音视频的采集录制和播放渲染，最终交给或是获取的都是原始音视频数据，视频例如 yuv 和 rgb 数据，音频 PCM 字节流。

下面简单介绍一下常用采集录制和播放渲染用到的模拟或是数字接口，他们都有自己音视频接口标准规范和协议（有的协议类似网络链路层协议，这些接口协议和 FFmpeg 无关）。

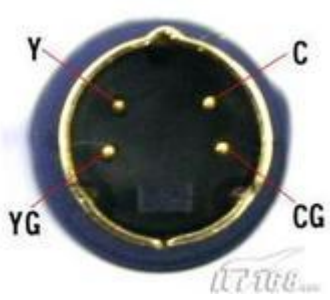
简要介绍些音视频接口：

1) 音频单独接口常用的有：3.5mm 接口、RCA 接口（莲花头）、Jack Cannon（卡农头），下图从左往右依次排列。

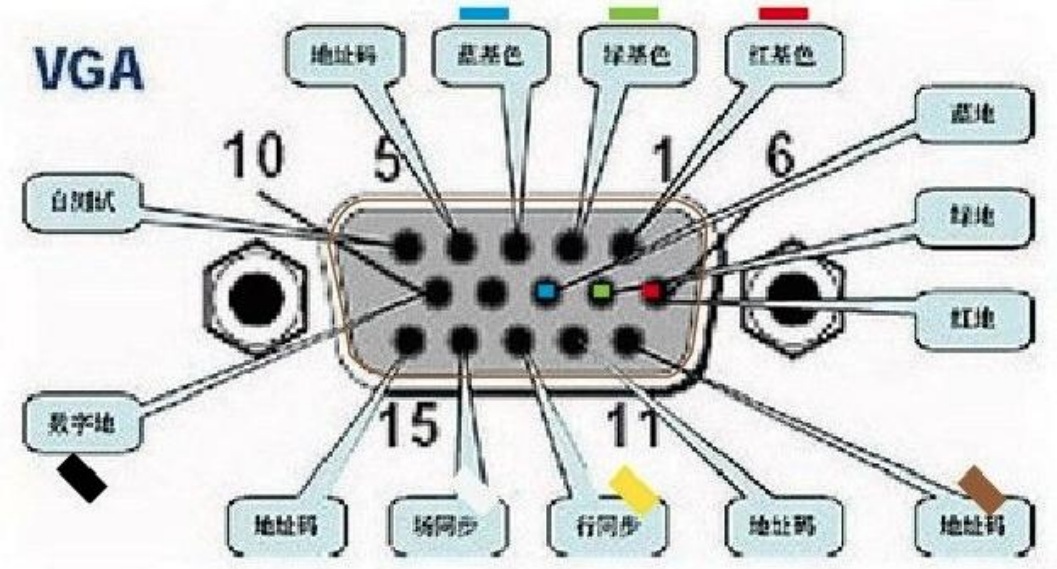

2) AV 接口：白线 L 音频，红线 R 音频，黄线 V 视频，yuv 混合传输。下图从左到右分别是 AV 接口和插槽图。


3) S 端子：只是视频传输线，常规 4 口分别为，1- GND 地线 (Y)、2- GND 地线 (C)、3- Y 亮度 (Luminance)、4- C 色度 (Chrominance)。下图从左到右分别为 S 端子接口、S 端子 7 针插槽、S 端子 4 针插头结构图。





4) VGA 接口：又称 D-Sub 接口，是模拟接口类，分解为 R、G、B 三原色和 HV 行场信号进行传输显示。下图从左到右、从上到下依次为 VGA 接口图、VGA 插槽图、VGA 接口 15 引脚逻辑图。

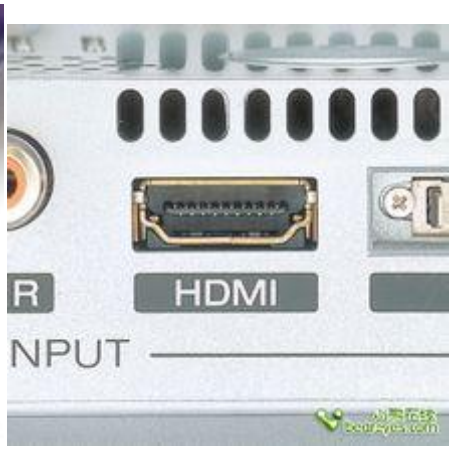


VGA管脚定义及VGA线焊接方式

引脚号	对应信号	对应焊接
1	红基色	red 红线的芯线
2	绿基色	green 绿线的芯线
3	蓝基色	blue 蓝线的芯线
4	地址码	ID Bit
5	自测试	
6	红地	红线的屏蔽线
7	绿地	绿线的屏蔽线
8	蓝地	蓝线的屏蔽线
9	保留	
10	数字地	黑线
11	地址码	棕线
12	地址码	
13	行同步	黄线
14	场同步	白线
15	地址码	
	外层屏蔽	D15 端壳压接

5) DIV 接口：分为 3 种类型 DVI-A（12+5）模拟信号、DVI-D（18+1 和 24+1）数字信号、DVI-I（18+5 和 24+5）兼容数字和模拟接口。下图是 DVI-D 数字视频接口图。

6) HDMI 接口：全称高清晰度多媒体接口（High Definition Multimedia Interface）是音视频数字接口，最大传输速率 165Mpix/second，足够支持 1080p 80FPS 的数据传输，最高传输速度为 5Gbps。下图从左到右依次为 HDMI 接口和 HDMI 插槽图。



7) DP 接口：全称 DisplayPort 接口，一种高清晰音视频流的传输接口，速度更快。下图为 DP 接口和插槽。

上面简述的都是硬件接口，还有音视频采集卡、显卡显示器、摄像头、投影仪、麦克、耳机等等具体硬件设备不在介绍。而他们才是最终的工作设备，这些设备最终要应用在程序中是需要硬件驱动程序，我们开发的应用程序一般不会直接调用硬件驱动程序，而是通过一个第三方的库或是平台来调用。

各个操作系统平台，采集和播放库各不相同，下面主要是介绍一些常用的库或是 API。

Windows 平台：

1. DShow 框架，这是微软自己的一套多媒体框架，filter 架构，支持音视频采集和播放。
2. WaveIO API 是用于音频采集和播放的 API。
3. GDI/GDI+ 是画图接口程序，也可以用来做视频渲染。
4. vfw 框架，这是微软自己的一套多媒体框架，支持音视频采集和播放，基本已经被淘汰。

Linux 平台：

1. OSS （Open Sound System）和 ALSA（Advanced Linux Sound Architecture）音频采集和播放。
2. fbdev （The Linux framebuffer）视频播放和采集，是一个独立图形硬件抽象层。
3. Bktr 与 X11 视频采集和渲染。

Android 平台：

1. 默认的 API 采集和播放 Camera、MediaRecorder、MediaPlayer、SoundPool 等。
2. 默认的播放器， VideoView 和 MediaController 系统的 API 控制播放。
3. SurfaceFlinger 视频画图渲染。
4. OpenGL ES（OpenGL for Embedded Systems）图形处理和播放。

Mac OS/iOS 平台：

1. Media Player、 HTTP Live Streaming、AV Foundation 组成媒体框架库。
2. QuickTIme 播放器，提供 SDK 使用。

除了以上各个系统平台自带的功能，还有第三方的渲染库，大多数都支持跨平台，例如 SDL，QT，opengl、opencl、openal 也可以做相关工作。

也顺道介绍下常用的开源播放器：MPlayer 播放器，VLC 播放器，射手播放器，也包含 FFmpeg 库自带的 ffplay 简易播放器。

其他 Android、Mac 平台都有各自的播放器，Flash 属于跨平台播放器。

5. 编解码器

编解码的主要作用就是压缩原始音视频数据方便存储和传输。下面主要介绍编解码器的一些行为和属性，这也是 FFmpeg 所抽象出来编解码器重要的属性和行为。

1) 输入和输出数据的单元

视频部分：输入的是 yuv/rgb 原始未压缩的视频图片，输出是字节流，有自己的字节流格式，也有头信息，头信息的作用是为了能正确解码。

例如：H264 视频编码，输入的画面是一帧逐行画面或是隔行的一场，输出的字节流 sps、pps、sei 组成的头信息，以及后续数据信息，x264 编码数据输出每一帧起始符号是 0x00 0x00 0x00 0x01，表示完整的一帧数据 slice，可以由多个 nalu 来组成。解码每一包一般都有相互参考，所以不可以随意丢包，否则解码会花，解码过程就是它的反序处理过程。

音频部分：输入是 PCM 字节流，编码单元根据编码器算法而决定，音频几乎所有的音频编码分析使用了傅里叶变换进行分析，所以一般输入单元为 2 的 N 次幂大小，输出的字节流单元不定，也需要有头，这个头信息一般跟在每一帧的前面，主要为解码算法使用。解码输出就是音频 PCM 流，一般包之间没有参考可以丢掉完整一包，播放时即缺少一包的时间，解码过程就是它的反序处理过程。

2) 延时性和数据缓存

视频部分：由于有的视频编码器需要根据时间进行动态图像的前后预测，所以视频 B 帧就会有延时输出的情况，同理解码也会出现相同的情况，所以视频编解码器就会有缓存帧的情况发生。

音频部分：一般不会有缓存，音频编码器都是在一定采样范围内进行分析编码，频谱的数值分析也都是在这一次采样块内完成，没有下一次采样块需要这次和下下次采样块的数值分析，也无前后预测，所以音频编解码没有延时和缓存。

3) 输出的字节流头

音视频编码器输出的字节流，有的都会有头信息，头信息的作用是编码器和解码器之间的约定协议，没有则不能解码。

6. 容器和协议

音视频流：将一帧帧的视频图片按照播放时间组合起来就是视频流。将一个个音频采样数据按照播放顺序组合起来就是音频流。

3D 视频流：现在去电影院可以看到 3D 立体电影，实际是人眼的双视觉效果，人分为左右眼同一时间左右眼看到的是 2 幅图片，由于人眼的视觉角度，这 2 幅图片所呈现的效果不同，也可以看做是 2 条不同的视频流。但是在实际的数据流传输、存储、处理的时候，其仍然是一条视频流，主要原因左右视觉的画面有很大部分都是相同或是有很大相同性，方便于视频压缩、处理和计算。

对于**容器**的理解好比是人的衣服，是装东西的盒子。例如：avi、flv、mp4、rmvb、wmv 这些文件格式都是容器，如果说容器是盒子，那盒子里装的东西就是音视频流。有的容器不仅可以放音视频流，也可以放字幕流、2 进制数据流、附件信息等等。

对于**协议**的理解就像交通工具，是运输盒子的传送带，它的作用就是将盒子从一个地方搬到另一个地方去的工具。例如：文件 io、tcp、udp、http、rtmp、pipe 这些都属于协议。

点播和直播：点播就是视频内容已经封装到容器当中，通过协议将这个容器从 A 地点搬运到 B 地点。直播就是 A 地点的人不断的从容器的头部往里面插入数据流，通过协议而在 B 点的人不断的从容器的另一头不断的向外拿数据流。

所以可以看出，直播情况下的容器和协议的耦合性要远远大于点播情况，由于直播情况的存在，所以容器和协议的耦合性是必然存在的，有时甚至会是同一个东西。

当然协议和容器之间也是由规格尺寸的，在特定情况下有的容器不支持这种协议传输，例如：mp4 容器在直播情况下无法通过 pipe 进行协议传输。

容器有意义的基础元素动作行为就是三种：封装、解析、Seek。而 Seek 动作的意义是：在播放视频的时候，拖动进度条到指定时间播放，就是容器 Seek 到指定时间处，开始做容器解析，将解析出的数据进行解码和渲染。

7. 常用概念介绍

Format：字面意思是格式（实际也有这个意思），在多媒体中翻译为容器（Container），例如：mp4、avi、flv、rmvb、rtsp 等都是容器。

Muxer：容器封装器，Muxing 一般指容器封装过程。

Demuxer：容器解析器，Demuxing 一般指容器封装过程。

Protocol：协议，在流媒体中协议和容器是相辅助相成的，例如：rtmp、rtp、http、pipe、file 等都是协议。

Stream：数据流，是容器中的一种逻辑抽象，一帧帧视频图片组成了视频流，一帧帧音频采样数据组成了音频流，也有字幕流、附加信息流、附件流等。

Encoder：编码器，是一种将信息由一种特定格式（或编码）转换为其他特定格式（或编码）的传感器、软件或是算法，转换的目的可能是由于标准化、速度、保密性、保安或是为了压缩数据。Encoding 一般指编码过程。

Decoder：解码器，是一种做反向操作撤销编码的传感器、软件或是算法，使其得到原有的信息数据。Decoding 一般指解码过程。

Codec：编解码器，指的是一个能够对一个信号或者一个数据流进行编解码操作的设备或者程序。

Packet：数据包，一般暗指经过编码后输出的或解码前输入的数据包信息。

Frame：数据帧，一般暗指未经过编码前或是解码后的数据帧信息。

Capture：采集，一般指音视频数据采集。

Render：渲染，一般指音视频数据渲染播放。

PTZ：全称是 Pan Tilt Zoom，翻译为云台，云台监控摄像头支撑设计，固定和电动两种。

FPS：帧率，英文 Frames Per Second 表示每秒钟的帧数。

BitRate：比特率，表示每秒钟的比特数，分为两个单位 bps（bits per second）每秒钟传输的比特数和 Bps（Byte per second）每秒钟传输的字节数。
换算 1Bps = 8bps，1KBps = 1024Bps，1MBps = 1024KBps，1Kbps = 1024bps，1Mbps = 1024Kbps。

pts、dts：pts 是 显示时间 或是 显示时序，dts 是 编码时间 或是 解码时序；时序可以看

做是从 0 开始的累加数值；时间可以是从 0 秒开始的相对时间，也可以是类似日期的绝对时间（FFmpeg 库的时间管理也是如此）。

Duration：时长，例如视频：一帧需要显示的时间。

Seek：在多媒体中翻译为检索最为合适，例如：向前检索、不支持检索、检索表、检索到视频第一帧等等。

PCR：全称 Program Clock Reference 节目参考时间，一般是 TS 流中插入的本地时间值即时钟数值，是在直播中做时间同步的。

sps、pps：全称 seq parameter set 和 pic parameter set，是 H264 编码字节流重要信息（俗称头信息），包含编码参数序列信息、编解码算法调整参数、编码图像信息，必须拥有才能解码。

vps、sps、pps：全称 video parameter set、seq parameter set、pic parameter set，是 H265 编码字节流重要信息（俗称头信息），主要说下 vps 主要设计目的代替 h264 时代的重要的 sei 辅助信息，解决在标准下重要 sei 可以丢失的尴尬局面，主要包括 HRD、LayersInfo、BitRateInfo、MultiviewInfo 等。

h264_mp4、h264_annexb：这是 h264 码流两种存储格式，h264_mp4 是在 ISO/IEC 14496-15 有所定义的格式，h264_annexb 是 h264 标准定义字节流格式，主要功能区别是 h264_mp4 用在可靠的数据传输当中，高效、低容错性，而 h264_annexb 可以用在字节流传输当中，低效、高容错性。

adts、asc：全称 Audio Data Transport Stream 和 Audio Specific Config，在 MPEG2 和 MPEG4 中有所定义，主要用于 AAC 音频编解码的字节流包头信息，保存的是具体编码参数和音频数据格式信息。

I、P、B 帧：是 H264/MPEG 编码帧类型，全称 Intra、Predicted、Bi-dir predicted，也有 S 帧类型是 MPEG4 编码器，SI、SP、SB 帧中的 S 表示 Switching。

IDR 帧：是刷新帧，属于 I 帧，IDR 帧等于 sps + pps + flush + I 帧。

Flush：清空缓冲数据操作，只要是有缓冲数据的都应该有次操作，例如：编解码器、容器操作、协议处理等等。

slice：编解码中，视频帧数据单元，可以是逐行的一幅或是隔行的一场数据。

VFR：全称是 Variable Frame Rate 翻译为动态帧率，每秒钟的帧数动态改变的，每一帧的时长也可能是动态改变的。

CFR：全称是 Constants Frame Rate 翻译为固定帧率，每秒钟的帧数固定不变的。

VBR、CBR、ABR：全称是 Variable Bit Rate、Constants Bit Rate、Adaptive Bit Rate；翻译为动态码率、固定码率、自适应码率；他们属于码控方式，不会影响到编码器的标准算法，但会需要码控算法才能实现，音视频编码都适用。

CBR：每一次编码输入的音频固定采样率、或视频固定帧数下，每一次编码输出比特数基本固定，每一次值基本相同。

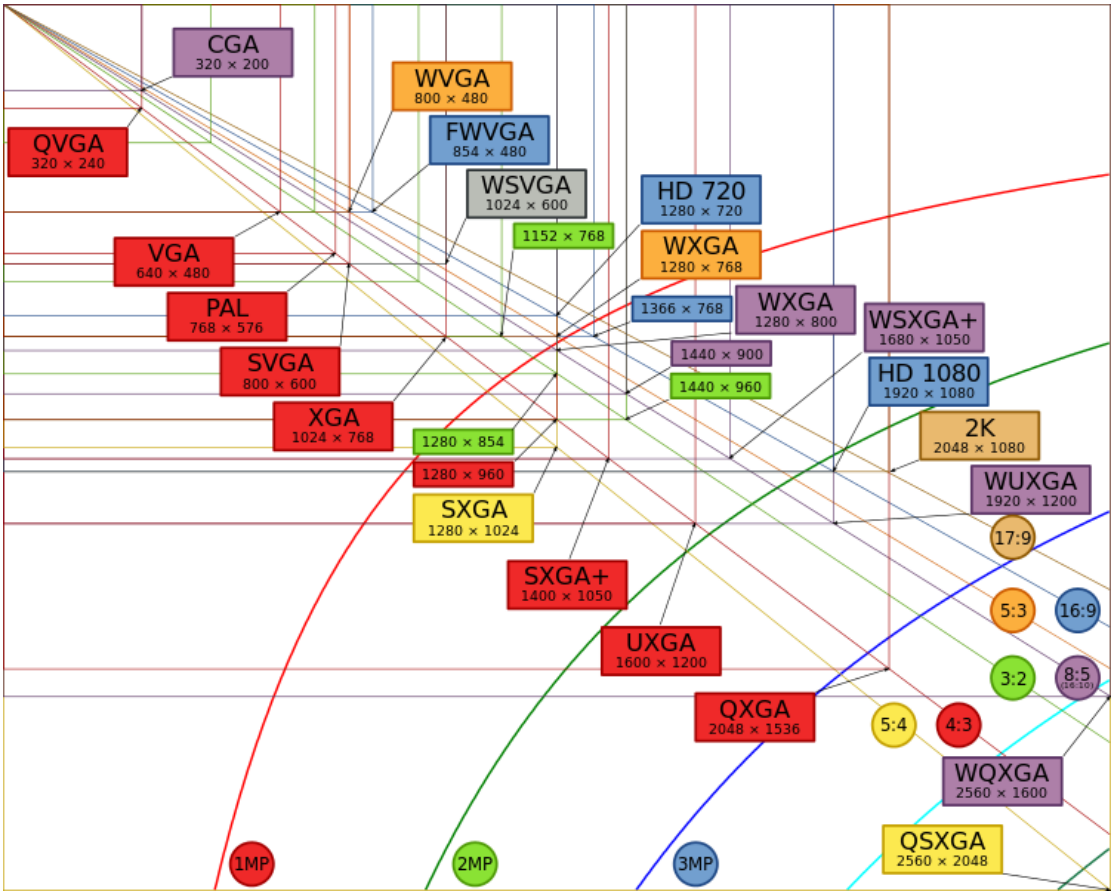
ABR：每秒编码输入的音频固定采样率、或视频固定帧数下，每秒编码输出比特数基本固定，每秒值基本相同。

VBR：表示每秒、或音频固定采样数、或视频固定帧数下，每一次编码输出的总体比特数不可控，主要考虑音视频质量的一种处理方法。

下表是国内常用的行业标准（涉及广电、监控、TV、互联网）：

宽 x 高	宽高比	像素（Pixel）	其他描述
480x270	16:9	12,9600	
512x288	16:9	14,7456	【288p-15FPS】
640x360	16:9	23,0400	
1280x720	16:9	92,1600	【720p-60/45Hz-D4】 【720p-25FPS】
1920x1080	16:9	207,3600	【1080i-60/33.75Hz-D3】 【1080p-60/67.5Hz-D5】 【1080p-25/30FPS】
3840x2160	16:9	829,4400	【4K Ultra HD】
7680x4320	16:9	3317,7600	【8K UHD】
480x320	3:2	15,3600	
540x360	3:2	19,4400	
720x480	3:2	34,5600	【480i-30/15.25-D1】 【480p-30/31.5Hz-D2】
320x240	4:3	7,6800	【QVGA】
480x360	4:3	17,2800	
640x480	4:3	30,7200	【VGA】
768x576	4:3	44,2368	【PAL-25fps-Interlaced】
800x600	4:3	48,0000	【SVGA】
1024x768	4:3	78,6432	【XGA】
1400x1050	4:3	147,0000	【SXGA+】
1440x1080	4:3	155,5200	
176x144	11:9	2,5344	【QCIF】
352x288	11:9	10,1376	【CIF】
704x576	11:9	40,5504	【D1】

下图是国际 TV 行业标准：



略讲 H264 历史：

为什么要在这里说一下 H264 历史呢？不仅是因为一些多媒体行业“标准”发展的历史很有趣，主要是以史为鉴可以对事物看的更加通透，了解的多了才能客观的给出评价，或是一个属于自己的认识。

H.264/AVC 虽说本质是一个，但还是侧重点不同。

H264 是 ITU-T 国际电信联盟远程通信标准化组织(ITU-T for ITU Telecommunication Standardization Sector)的 VCEG（视频编码专家组）所定义的标准名称。

AVC 是 ISO/IEC 即 国际标准化组织（ISO）和国际电工委员会（IEC）依据 MPEG（活动图像编码专家组）所定义的 MPEG – AVC/MPEG – Part 10（简称 AVC）标准名称。

这俩个哥们就像孪生兄弟，最近俩哥们在搞 H.265/HEVC，仿佛历史的巨轮又在重复着昨天的故事，让我大开眼界的是原来标准制定组织也能搞竞争？当然这是一个玩笑话。

从编解码器的角度看 ITU-T 和 ISO/IEC 侧重点不同，当然他们所处的行业也不同，至于怎么不同需要自己揣摩，当然他们的基本标准是相同的。

ITU-T 制定了 H.120、H.261、H.262、H.263、H.264、H.265 这些视频标准，还制定 G723、G726、G729 这些音频标准。

ISO/IEC 制定了 MJPEG、JPEG2、MPEG-2、MPEG-4、AVC、HEVC 这些视频标准，还制定了 MP3、AAC、HE-AAC 这些音频标准，当然还有 mp2、mp3、mp4、ts、jpg 容器标准。

关于 H264/AVC 和 H265/HEVC 的标准制定都是一样的，只是 MPEG 工作组还定义了 ES 码流存放在容器或是协议中时的具体格式，就是涉及他们家的那些事，而 VCEG 那输出的就是裸流格式了。

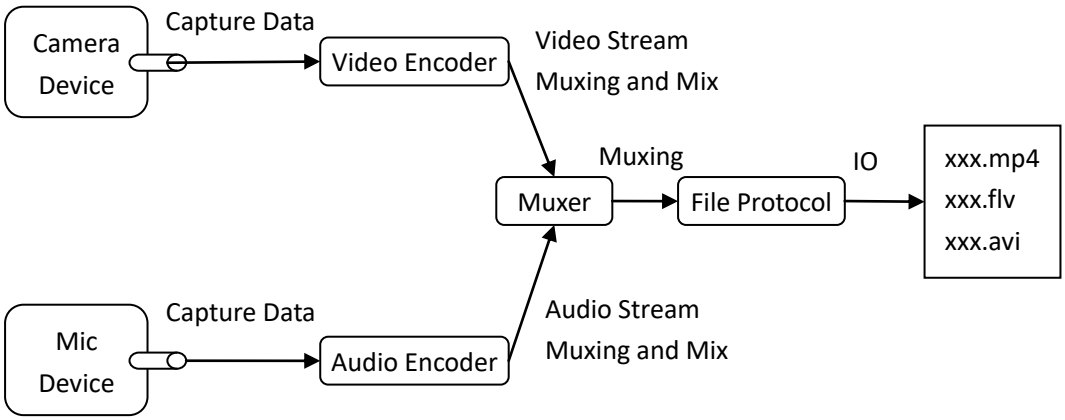
当然标准制定不是“拍脑袋用屁股做决定”的事情，都是经过实践得出来的结论，例如 GPAC、HM 等，都属于科研项目也就是开源项目，开发者通过不断实践运行的结果，来制定优化标准细则，也就是“先有鸡还是先有蛋”的问题。所以可以看出来，是先有设计和讨论，后有的代码以及修改，最终提出完善行业标准的。

8. 流媒体数据流程讲解

这个小节主要讲解一些流媒体程序的数据流程图，从总体上了解流媒体程序的不变原则，在宏观上为初学者指点迷津。

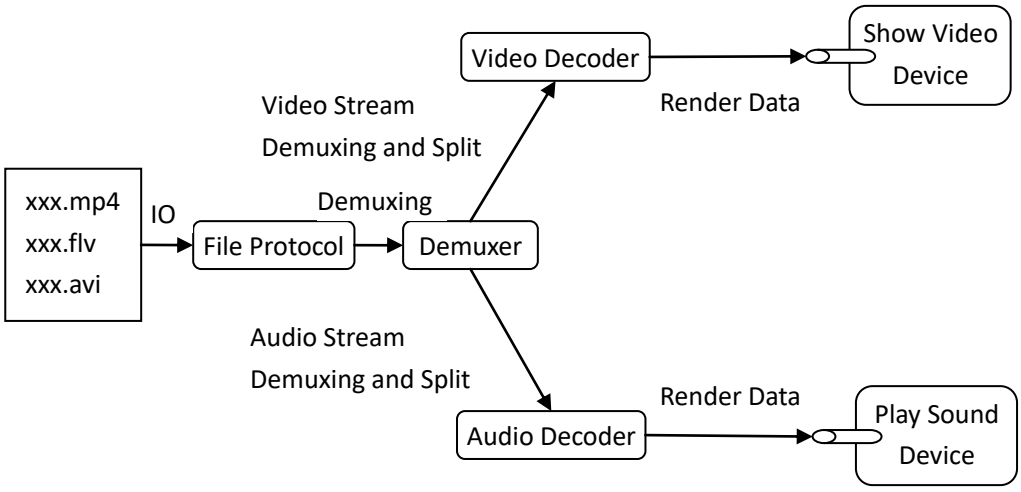
下图是音视频数据采集保存媒体文件的流媒体数据流程图：

流媒体数据流程图(copyright lsm)



下图是播放器播放媒体文件的流媒体数据流程图：

流媒体数据流程图(copyright lsm)

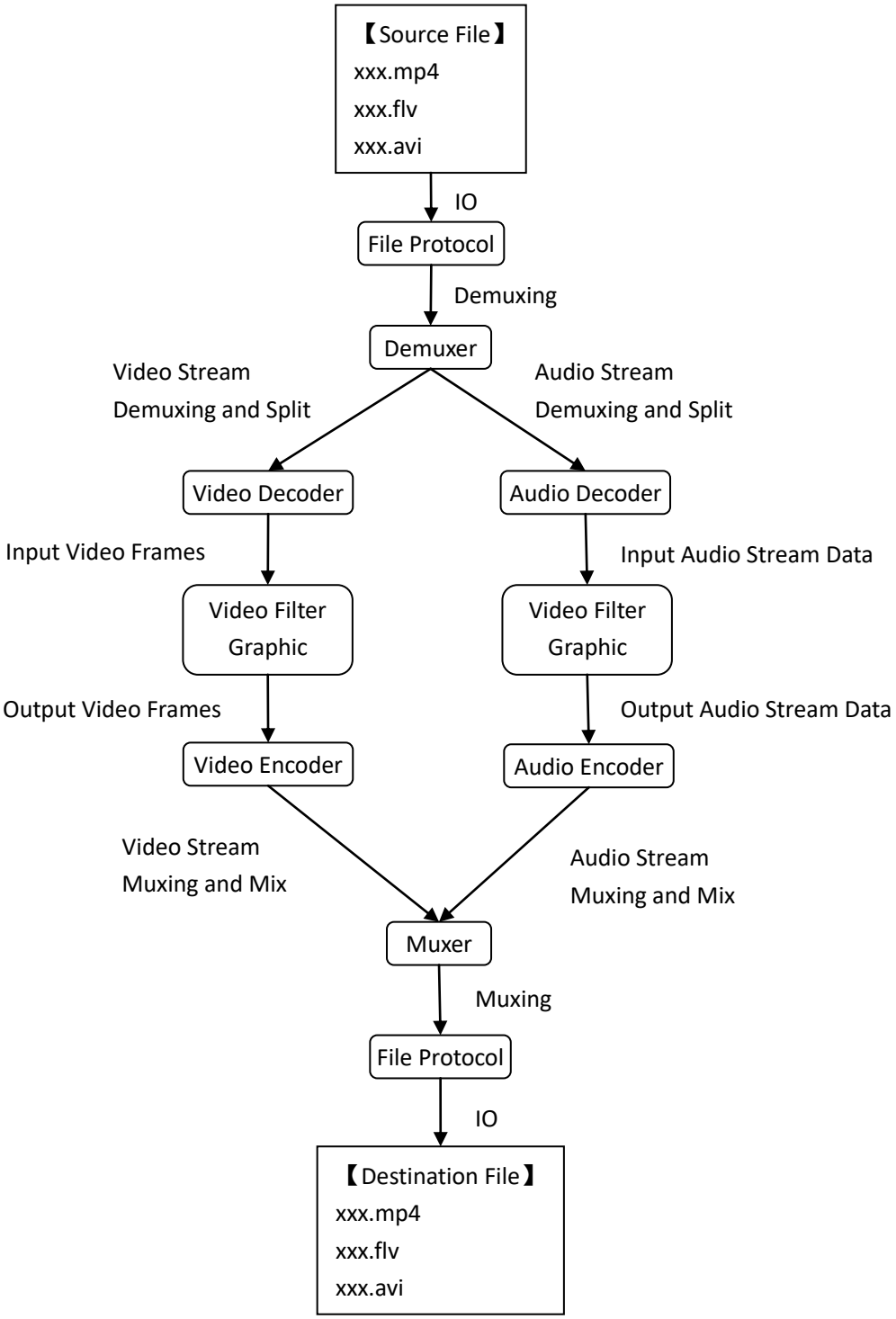


从上面 2 个图可以看出采集和播放的数据流程的整个经过，以及最终的数据流向。通过

这两个图可以很容易的得到转码器的数据流程图，就是将采集和渲染两个步骤除去，把剩下的连接起来，在转码的时候可以加入 AVFilter Graphic 进行音视频数据的处理。

下图是转码器的流媒体数据流程图：

流媒体数据流程图(copyright lsm)

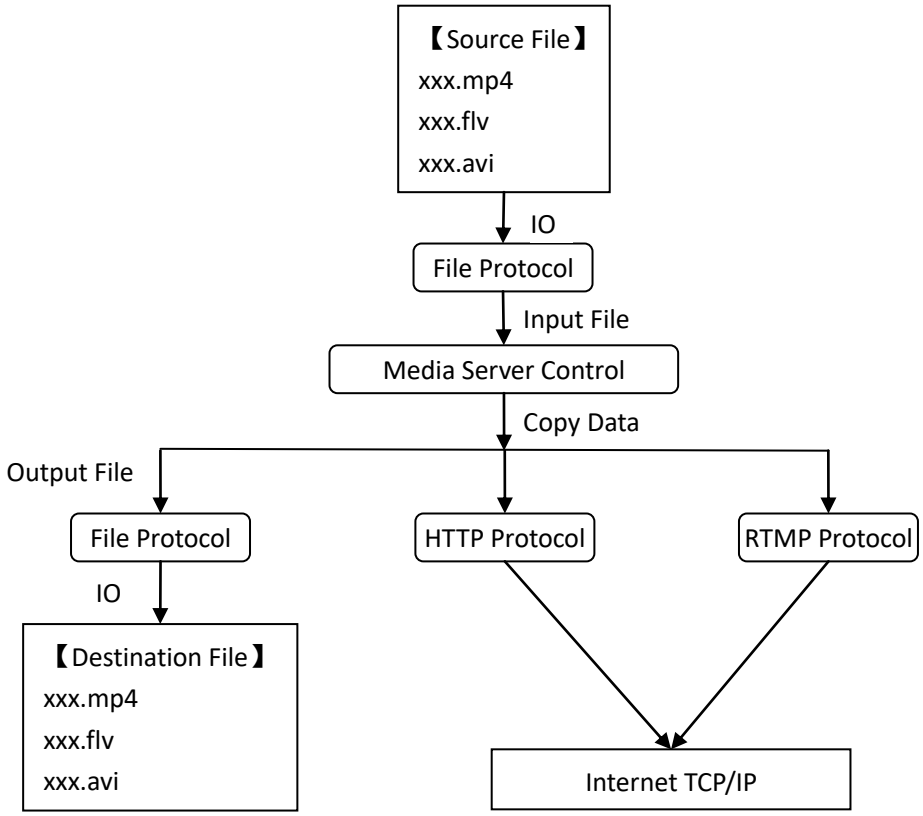


其实在采集和播放流程中也可以随意加入 Filter Graphic 进行音视频数据的处理，Filter Graphic 主要针对的是音视频原始数据进行，还有 BitFilter Graphic 主要针对音视频编码后字节流数据或字节流数据包进行的处理。

根据上面采集、播放、转码、音视频处理后也不难猜出流媒体服务器程序的数据流程图，一般情况下流媒体服务器一般不会做编解码操作，最多的操作是容器转换，有的甚至容器转换都不需要只是修改下协议而已，流媒体服务器强调的是高并发和高效利用率，在低 CPU 利用率下将磁盘、网卡等设备的使用率充分发挥。

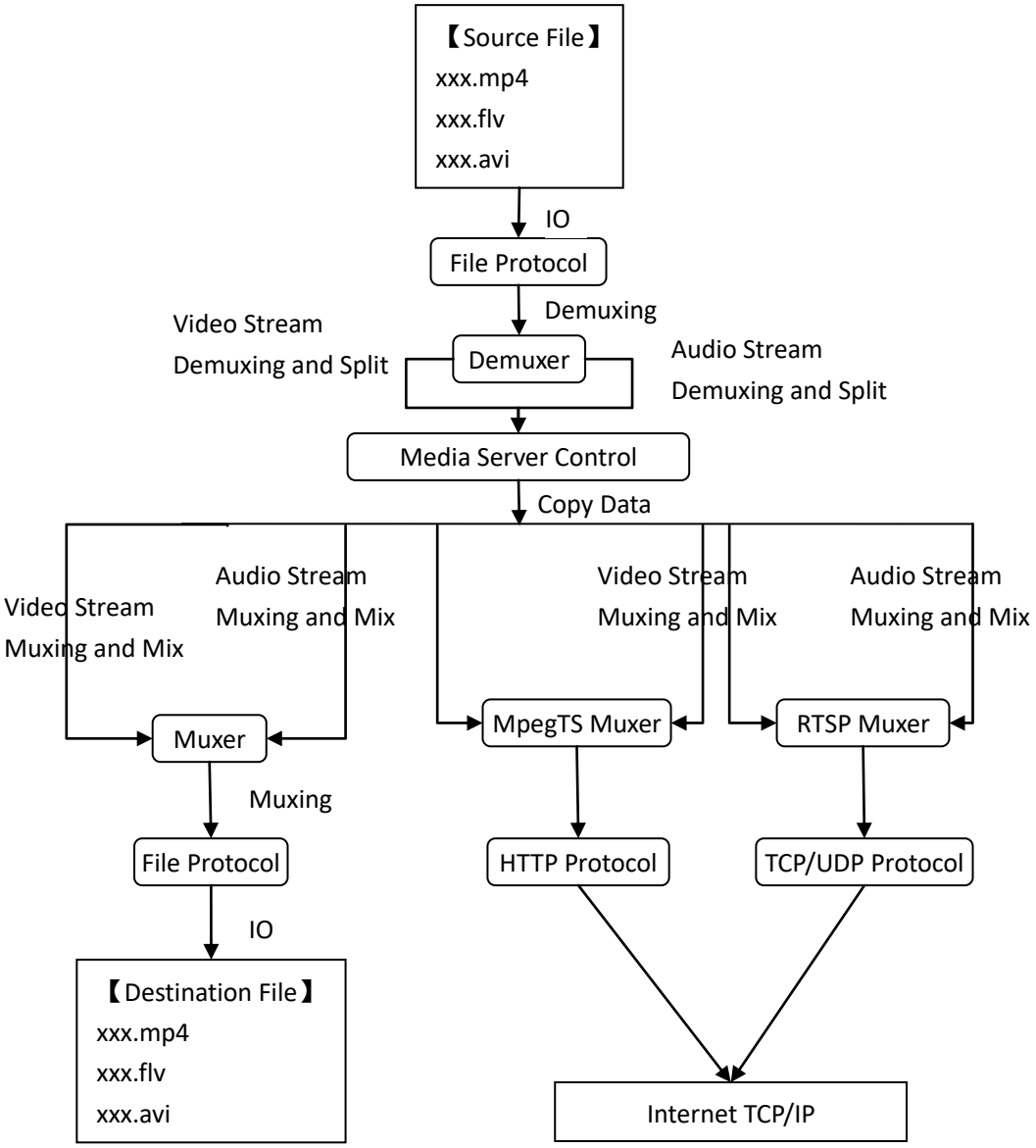
下图是支持协议转换媒体服务器的流媒体数据流程图：

流媒体数据流程图(copyright lsm)



下图是支持容器转换媒体服务器的流媒体数据流程图：

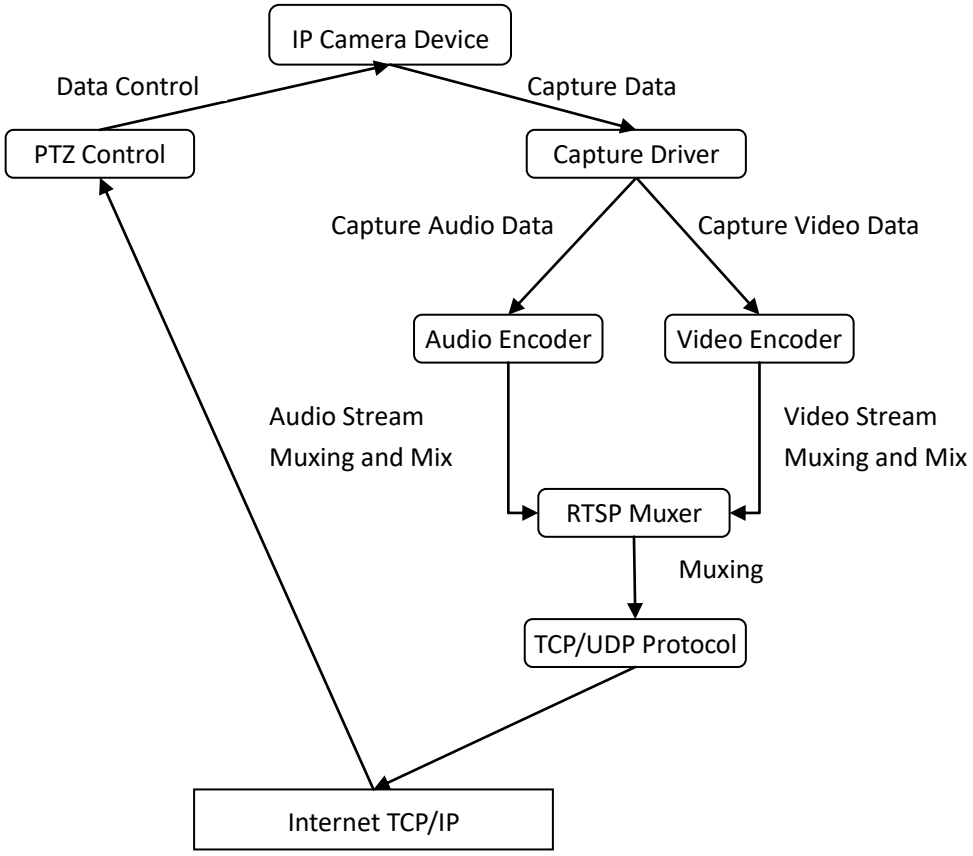
流媒体数据流程图(copyright lsm)



关于数据采集、播放器、转码器、流媒体服务器的大致数据流程都已经讲过了，下面是监控摄像头到 DVR、DVS、播放器的数据流程图解。

下图是监控摄像头的流媒体数据流程图：

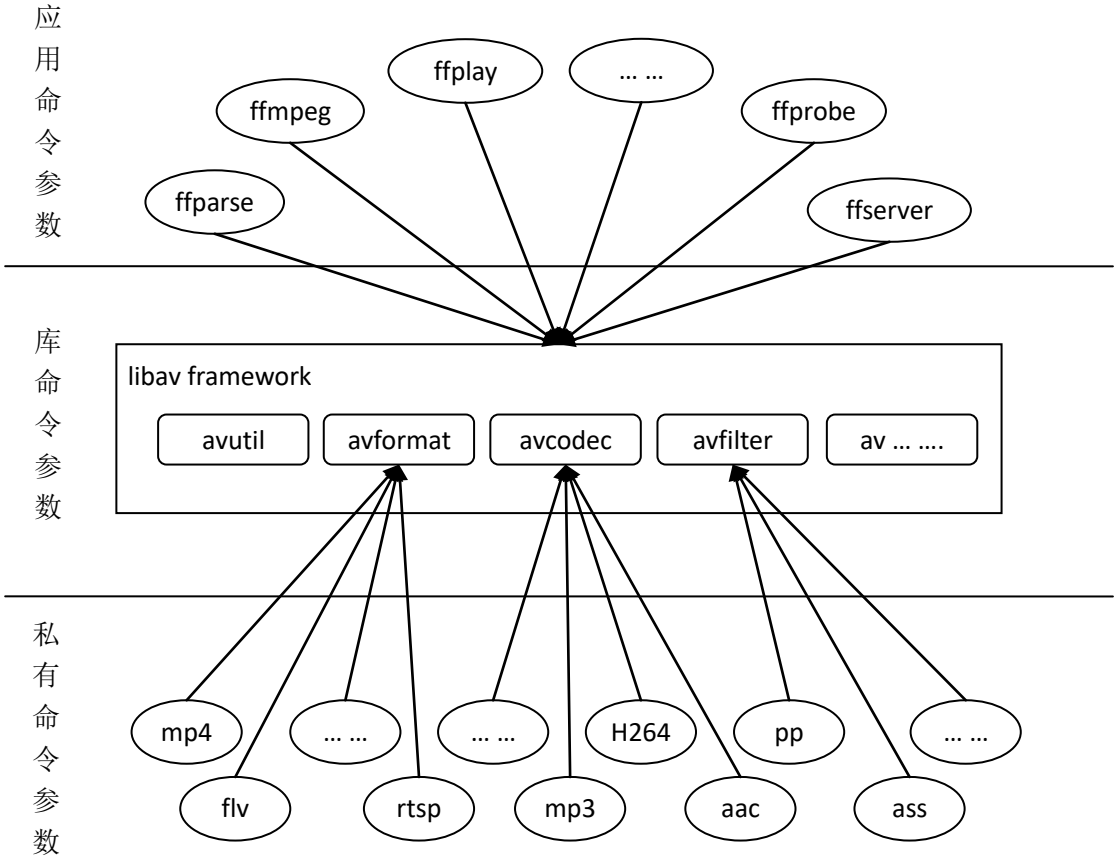
流媒体数据流程图(copyright lsm)



第二篇 快速应用篇

这一篇其实是作者最后所写的，为了能和读者尽快见面，所以仅挑选一些常用精简的命令作为介绍，其核心工具包含 ffplay 播放器、ffmpeg 转码器、ffserver 流媒体服务器、ffprobe 媒体信息查询，ffparse 作者编译的库查询工具。

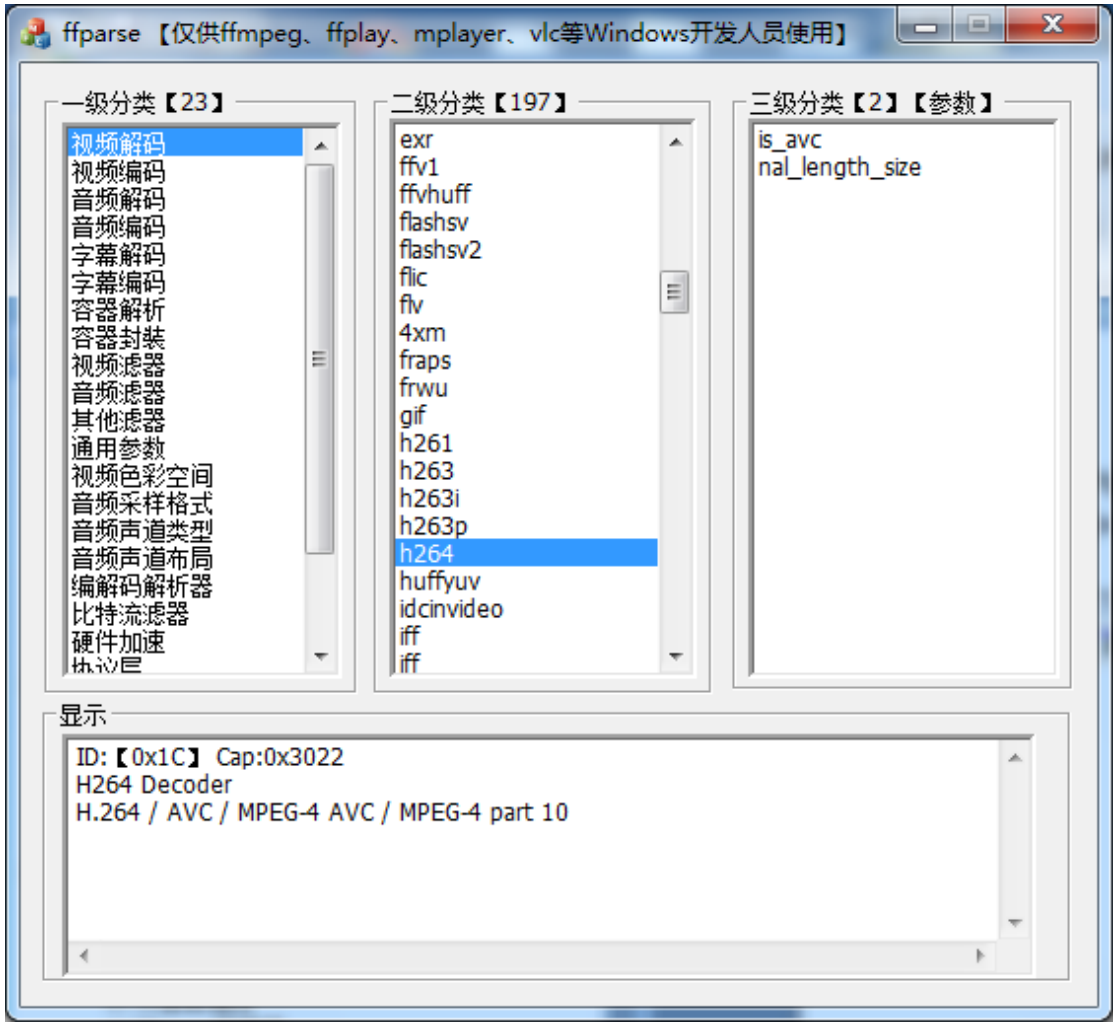
关于 ffmpeg、ffplay、ffserver、ffprobe 的命令行参数，按照软件的层次不同主要分为三类，第一类是 ffmpeg、ffplay、ffserver、ffprobe 内部实现的**应用命令参数**，第二类是库实现的**库命令参数**，第三类是私有对象实现的**私有命令参数**，如下图所示。



1. ffparse

ffparse 工具是作者 StrongFFmpeg 项目中一个小的开源工具，下载地址可以通过 Baidu、Google 等搜索引擎找到,它是一个在 Windows 下带界面的帮助工具。

下图是 ffparse 的主界面展示图：

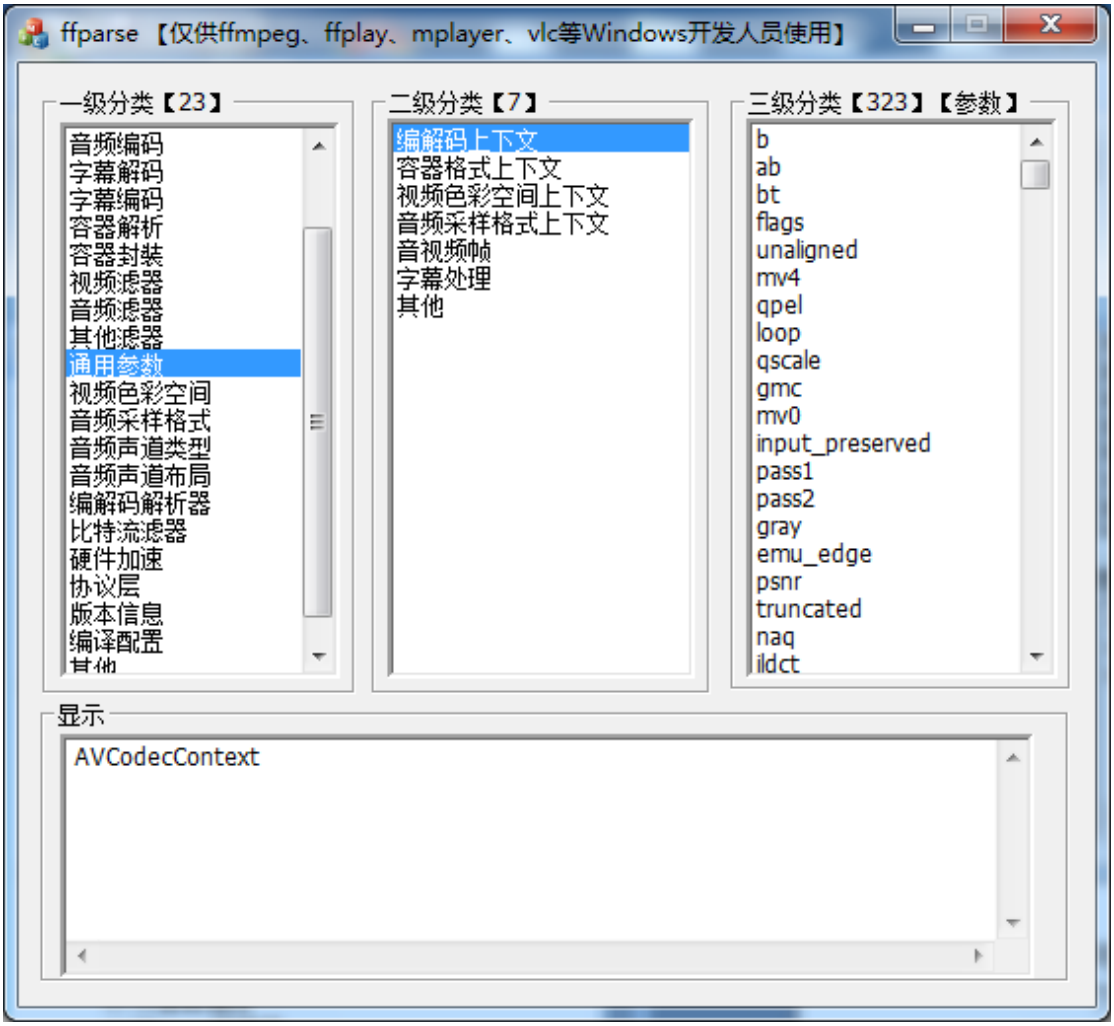


ffparse 界面显示主要分为三级索引格式，一级分类主要是通过调用库可以直接使用到的各个功能名称类型，二级分类是对一级分类的展开主要介绍各个相同功能类型下的对象成员，三级分类是在对象成员的基础上展开其参数信息、类型支持信息等。

可以通过 ffparse 查找到库命令参数和私有命令参数。

私有命令参数：主要通过 ffparse 一级分类除了【通过参数】以外找到的所有三级分类的参数都是私有命令参数。

下图是 ffparse 的库命令参数：



库命令参数是由 FFmpeg 库的框架实现，就是 ffparse 一级分类中的【通用参数】，二级分类是具体的通用参数的类型，三级就是对应类型的参数说明。

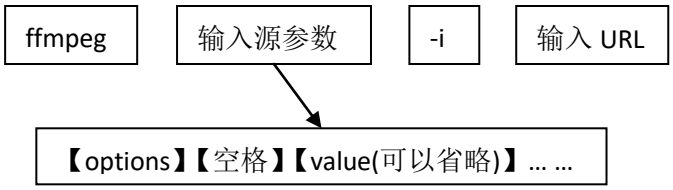
2. ffplay

ffplay 是一个基于 FFmpeg 库的精简播放器程序，渲染使用到了 SDL（SimpleDirectMediaLayer）开源库，通过命令行启动播放，这里主要以命令行语法讲解为主。ffplay 仅是一个精简的播放器程序 Demo，复杂的可以参考：MPlayer、VLC media Player 等开源项目。

帮助命令：列出全部命令信息，比较多挑重点看。

- **ffplay -h**

ffplay 语法格式：**【命令+命令值（可选）紧邻成对出现】**
ffplay【空格】【options(命令)】【空格】【value(命令值，如果命令有参数值，否则可以省略)】



输入源：-i 后面紧跟 URL，这个命令就可以实现视频的播放功能。

- **ffplay -i "目录\测试片源.mp4"**
- **ffplay -i "http://192.168.1.14:8012/test.mp4"**
- **ffplay -i "rtsp:// 192.168.1.14:8013/test"**

ffplay 的应用命令参数都保存在 cmdutils_common_opts.h 和 ffplay.c 中，其中头文件中的是 ffmpeg、ffprobe、ffplay、ffserver 的通用命令。

输入源描述：要放到-i 命令的前面表示

- **ffplay -f mpegts -x 320 -y 240 -i "http://192.168.1.14:8012/test"**

强制输入容器使用 mpegts，强制使用 320x240 尺寸显示。

3. ffmpeg

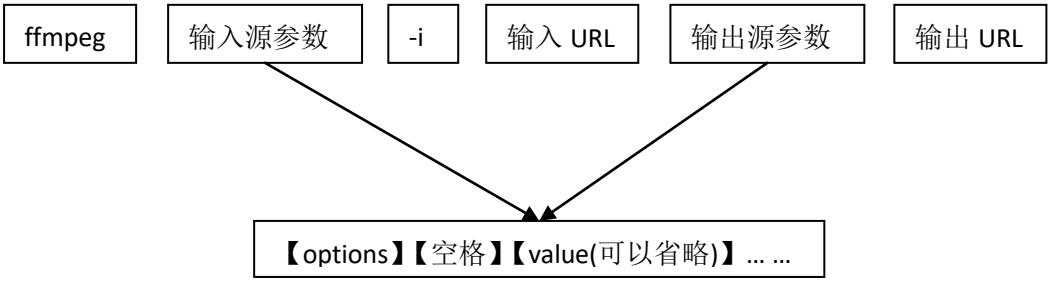
ffmpeg 是一个基于 FFmpeg 库的经典转码程序，几乎被所有的知名系统、软件工具、服务器后台等静默使用，通过命令行启动转码，这里主要以命令行语法讲解为主。

精简帮助命令：
- ffmpeg -h

更多帮助命令：
- ffmpeg -h long

完整帮助命令：
- ffmpeg -h full

ffmpeg 的语法格式：**【命令+命令值（可选）紧邻成对出现】**
是在原有 ffplay 格式的基础上进行添加的。



案例 1 解析：
ffmpeg -f mpegts -i "http://AVTestFile/AVNormal/52" -vcodec x264Encoder -r 15 -b:v 256000 -vf scale=800:600 -an -copyts -y "51.avi"

输入源使用 mpegts 容器 http 协议的 URL
-vcodec x264Encoder 使用 x264Encoder 视频编码器
-r 15 视频 15 帧
-b:v 256000 视频编码 265Kbps 码率
-vf scale=800:600 使用视频滤器 scale 进行缩放到 800x600 尺寸
-an 禁用音频
-copyts 时间戳拷贝
-y 覆盖输出文件
输出 URL 默认使用 file 协议

案例 2 解析：

```
ffmpeg -i "http://AVTestFile/AVNormal/52" -vn -acodec libvo_aacenc -strict
experimental -ar 44100 -ac 2 -b:a 62000 -sample_fmt s16 -copyts -y
"D:/AVTestFile/OutPut/lsm_sample.mp4"
```

输入源使用 http 协议的 URL 容器格式自适应
-vn 禁用视频
-acodec libvo_aacenc 使用 libvo_aacenc 音频编码器
-strict experimental 表示此编码器是实验版本，只有这样才可以正常使用
-ar 44100 采样率使用 44100
-ac 2 使用双声道
-b:a 62000 音频码率 62Kbps
-sample_fmt s16 采样深度使用 signed 16bit 格式
-copyts 时间戳拷贝
-y 覆盖输出文件
输出 URL 默认使用 file 协议

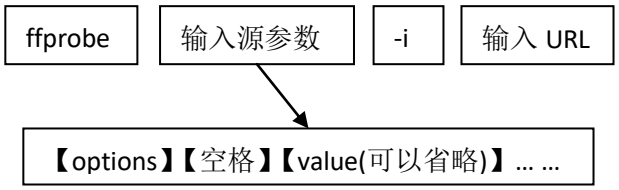
4. ffprobe

ffprobe 是一个基于 FFmpeg 库的媒体信息分析工具，更加专业的 MediaInfo 项目可以对容器格式属性进行专业分析，而 ffprobe 可以对数据包进行数据分析，并且可以对解码进行全面的实际分析。

帮助命令：列出全部命令信息，比较多挑重点看。

- ffprobe -h

ffprobe 语法格式：**【命令+命令值（可选）紧邻成对出现】**



案例 1 解析：

ffprobe -i "http://AVTestFile/AVNormal/52"

自动分析 URL 文件

5. ffserver

ffserver 是一个基于 FFmpeg 库的精简流媒体服务器，而更加专业的 live555 项目有更高的兼容性和支持更多的输出数据格式，而 ffserver 在多用户支持方面仅是个 Demo 级应用程序。

ffserver 默认读取/etc/ffserver.conf 文件进行初始化配置解析，主要有 IP 地址、最大连接数、RTSP 和 HTTP 端口号、ffm 文件、源配置信息等等。此配置文件在源码 doc 文件夹下，有案例文件 ffserver.conf，有详尽的说明。ffserver 通过读取配置文件来确定输入的源文件 ffm 的 URL 或 HTTP 等。

ffserver 需要和 ffmpeg 程序放在一起，通过调用 ffmpeg 程序进行输入源处理、采集、转码等，动态输出 ffm 文件、HTTP、 Pipe 等为 ffserver 提供数据源信息，ffserver 也可以使用生成好的文件进行工作。ffm 文件的生成需要自己通过命令行启动 ffmpeg 生成。

以上就是 ffserver 的一个简单的工作流程。

案例解析：

- ffserver -f doc/ffserver.conf

启动 ffserver，指定 ffserver 的.conf 配置文件路径。

第三篇 应用开发篇

1. FFmpeg 库编译和入门介绍

1) 下载源码

登陆 http://www.ffmpeg.org/download.html 网址下载您需要的对应平台的发布版本的代码，也可以登录 http://www.libav.org/download.html 网址下载您需要的对应平台的稳定版本的代码。

FFmpeg 和 Libav 代码基本相同，没有太多出入，即使在某一个时期略有不同，很快在后续版本更新时，相互之间有代码移植的工作，所有从时间轴总体趋势上最终代码还是保持着一致性，可能未来的接口版本略有不同。

FFmpeg 开源库的最大价值是它的各个框架的抽象接口，它可以把任意的协议、容器格式、编解码器、包括不断完善的过滤器等融入到自己的框架当中。

FFmpeg 框架库完全是由 C 语言和汇编语言编写的，每一个模块的框架设计上不仅花费了开发人员的大量心血，也是在不断版本升级中用时间磨练和总结出来的，同时也是面向对象的。在 FFmpeg 的调用者看来，avcodec、avformat、avfilter 等这些框架结构都是独立的，只有内部实现有关联性，由于其每个模块的独立性，比起来 DShow、avs 等 filter 类框架有很好的灵活性，大大方便了调用开发者，调用开发者可以任意选择使用某一个功能，仅通过调用 3,4 函数就可以完成。

如果 FFmpeg 开源库仅使用自己实现的协议、容器格式、编解码器、过滤器等，它就不能算的上是“多媒体开发核心工具”，它还集成了 20 到 30 开源或非开源的多媒体相关库，例如：MP3Lame、x264、faac、aacplus、xvid、freetype2 等等。也可支持对 DShow、Vaapi、avs、opencl 等等库的接入支持。

回到下载代码主题：

1. 下载 SDL 库为编译使用 ffplay 做准备，SDL 库是给 ffplay 播放器做渲染使用的，选择您对应的平台版本。
   下载地址：【http://www.libsdl.org/download-1.2.php】。

2. 下载 yasm 编译器，为编译 ffmpeg 的.asm 文件所使用，选择您对应的平台版本。
   下载地址：【http://yasm.tortall.net/Download.html】。

3. 下载 zlib 库，是数据压缩解压库。选择您对应的平台版本。
   下载地址：【http://zlib.net/】。

4. 下载 pthread 库，扩平台多线程基础管理库，选择您对应的平台版本。
   下载地址：【https://computing.llnl.gov/tutorials/pthreads/】。

5. 如果您是 Linux 平台开发，建议下载较新版本的 gcc 编译器 4.6 或是 4.7 版本。
   下载地址：【http://gcc.gnu.org/】。
   如果您是 Linux on Windows 开发建议使用 Cygwin 环境，也建议更新 gcc 编译器，cygwin 有对应的 gcc 更新。
   下载地址：【http://www.cygwin.com/】。

6. 如果您是 Android 平台开发，下载 NDK 平台开发工具。NDK on Linux 或是 cygwin + NDK on Windows。
   下载地址：【http://developer.android.com/tools/sdk/ndk/index.html】。

7. 如果您是 Mac 平台开发，xcode 已经集成，也可以自己更新 install gcc on mac os；如果是 ios 系统只能用 xcode 集成的编译。

8. 如果您是 Windows 平台，需要使用 C99 及其以上标准的编译，由于 ms 的 cl 编译不支持，所以交叉编译环境 msys 或 cygwin，也需要 MinGW 中间件支持交叉编译，编译输出 windows 二进制，MinGW 环境已经集成了 msys。
   下载地址：【http://www.mingw.org/】。
   如果需要更强的兼容性，也需要 Microsoft Visual Studio 的集成开发环境，详细用途在以后再做详尽说明。
   下载地址：【http://www.microsoft.com/visualstudio/eng/】。

9. 如果您想使用 intel 编译器，FFmpeg 库代码和配置都有支持。intel 编译器支持 windows、linux、mac 平台版本 intel XE 2013，intel 编译器是收费的。
   下载地址：【http://software.intel.com/en-us/c-compilers】。

10. 同时官方也提供已经编译好的各个平台的 bin 文件以及对应的源代码，方便二次开发人员的直接使用。
    下载地址：【http://www.ffmpeg.org/download.html】。

2) 配置编译环境

配置编译环境在这里就不做详尽的介绍了，由于环境太多不能一一给出。每一个环境的配置可能在网上搜索到，建议可以先百度，看一些中文的博客或论坛，因为搜索出来的基本都是中文易于新手上路，最后用 google 吧，在搜索的时候建议使用.com、.ca 顶级域名站点，实在不行就用.hk 吧。

关于网上配置文件，建议能使用最新稳定版本就使用最新的稳定版本，文章也尽量看最新的介绍，因为如果看到的是老文章，可能会走弯路，浪费时间。

FFmpeg 编译环境校验，可以通过命令，一一对各个环境进行检验，查看是否存在以及版本是否正确。

命令例如：gcc –version、yasm –version、link –version、icl –version 等等。

3) 外部库编译

在编译外部库部分也不做详细介绍，还是建议读者通过搜索的方法，针对某一个平台的某一个库的编译和配置进行详细的了解。

在除了 SDL、pthread、zlib 等基础性库外，这里还详细介绍些需要编译 FFmpeg 使用的其他外部功能库，有些可能涉及到版权，请谨慎使用。

常用介绍：

1. x264 库，主要功能用于 AVC/H264/ MPEG-4 part 10 视频编码器。
   下载地址：【http://www.videolan.org/developers/x264.html】。

2. Webm 库，主要功能用于 Google Vpx（vp3、vp5、vp6、vp8、Futrue vp9） 视频编码器、容器格式、webp 压缩。
   下载地址：【http://www.webmproject.org/】。

3. XAVS 库，主要功能用于中国音视频标准 AVS 编解码器。
   下载地址：【http://xavs.sourceforge.net/】。

4. Xvid 库，主要功能用于 MPEG-4 part 2 视频编码器。
   下载地址：【http://www.xvid.org/】。

5. Theora 库，主要功能用于 MPEG-4/DivX 视频编码器。
   下载地址：【http://theora.org/】。

6. Openjpeg 库，主要功能用于 jpeg 2000 图像编码器。
   下载地址：【http://www.openjpeg.org/】。

7. Schroedinger 库，主要功能用于 dirac 视频编码器。
   下载地址：【http://diracvideo.org/】。

8. utvideo 库，主要功能用于视频编解码器组件套，vfw 版本。
   下载地址：【http://www.free-codecs.com/download/ut_video_codec_suite.htm】。

9. lame 库，主要功能用于 MP3 音频编码器。
   下载地址：【http://lame.sourceforge.net/download.php】。

10. TwoLAME 库，主要功能用于 MP2 音频编码器。
    下载地址：【http://twolame.org/】。

11. faac 库，主要功能用于 AAC 音频编码器。
    下载地址：【http://www.audiocoding.com/faac.html】。

12. fdk-aac 库，主要功能用于 AAC 音频编码器。
    下载地址：【https://github.com/mstorsjo/fdk-aac】。

13. vo-aacenc 库，主要用于功能用于 AAC 音频编码器。
    下载地址：【https://github.com/mstorsjo/vo-aacenc】。

14. opencore-amr 库，主要功能用于 amr 音频编解码器。
    下载地址：【http://sourceforge.net/projects/opencore-amr/】。

15. vo-amrwbenc 库，主要功能用于 amr 音频编码器。
    下载地址：【https://github.com/mstorsjo/vo-amrwbenc】。

16. Speex 库，主要功能用于音频编码器。
    下载地址：【http://speex.org/】。

17. celt 库，主要功能用于 celt 音频编码器。
    下载地址：【http://celt-codec.org/】。

18. opus 库，主要功能用于 celt 音频编码器。
    下载地址：【http://opus-codec.org/】。

19. iLBC 库，主要功能用于音频编码器。
    下载地址：【http://www.ilbcfreeware.org/】。

20. Vorbis 库,主要功能用于音频编码器。
    下载地址：【http://vorbis.com/】。

21. gsm 库，主要功能用于 gsm 音频编码器。
    下载地址：【http://packages.debian.org/source/squeeze/libgsm】。

22. libass 库，主要功能用于字幕文件解析、封装、编解码。
    下载地址：【http://code.google.com/p/libass/】。

23. GnuTLS 库，主要功能用于协议通信。
    下载地址：【http://gnutls.org/】。

24. RTMPDump 库，主要功能用于 rtmp、rtmpt、rtmpe、rtmpte、rtmps 协议通信。
    下载地址：【http://rtmpdump.mplayerhq.hu/】。

25. libbluray 库，主要功能用于蓝光视频解析、封装、播放器。
    下载地址：【http://www.videolan.org/developers/libbluray.html】。

26. Frei0r 库，主要功能用于视频后期处理、特效、滤镜等。

下载地址：【http://files.dyne.org/frei0r/】。

27. Soxr 库，主要功能用于音频重采样和格式转换。
    下载地址：【http://sourceforge.net/projects/soxr/】。

28. Fontconfig 库，主要功能用于字体文件解码和文字渲染。
    下载地址：【http://freedesktop.org/wiki/Software/fontconfig】。

29. FreeType 库，主要功能用于字体文件解码和文字渲染。
    下载地址：【http://freetype.sourceforge.net/】。

30. libcaca 库，主要功能用于图形处理。
    下载地址：【http://caca.zoy.org/wiki/libcaca】。

其中还没有包含硬件程序库，例如 Stagefright、X11、DShow、vfw、vdpau、vaapi、dxva2、vda 等等。

将编译好的库要 make install 到相应文件夹，以便 FFmpeg 库编译的时候可以顺利找到。

4) FFmpeg 库编译

1. configure 设置编译配置，可以先通过 configure –help 查看默认和自定义的命令参数及其意义。

在看下面参数前，建议您先运行./configure –help 将其列出的所有参数详细查看清楚后再来看以下内容。

必须要知道和了解的编译的命令参数：

i. 可以通过 Program options 来禁用是否输出 ffmpeg、ffplay、ffprobe、ffserver 等可以执行程序，如果您是直接调用库函数可以禁用。或是先编译出 ffmpeg 的库，再通过--enable-shared 指定目录的方式后编译输出可执行程序也可以。
ii. 可以通过 External library support 给出的参数，指定加入哪个扩展库。
iii. 编译线程库可以选择 w32threads、os2threads、pthread、以及默认，根据不同平台一定要选择适合自己的线程库，不然会出现性能或是其它问题。
iv. 注意 Advanced options 中的命令，例如--cross-prefix

2. make 命令编译，以及 make install 命令要的头文件、库文件和二进制文件复制到默认或是制定用户目录下。

编译需要较长时间请耐心等候，如果一个完整库的编译时间大概需要半小时多，这个还要看机器的性能。如果您想从头到尾，学习编译 FFmpeg 库加上，寻找文章的时间，从零开始配置环境的时间，以及各个第三方库的配置编译时间，最终编译出 FFmpeg 完整库的大概时间在需要 1-3 周不等的时间，这个要看熟练度了。很多外部库的编译时间是不好预期的。

在 Linux、Cygwin 或是其它环境编译，Linux 平台输出较为简洁，毕竟 FFmpeg 基础就是在这个平台开发的。其次是 Android 和 Mac 平台的输出，ios 和 windows 平台的输出较为繁琐。

当然编译很重要，但和真正有意义的代码来讲还是很简单的。对于新手或是初学者来说，FFmpeg 的编译工作是必学的，可以让你了解各个开发和运行平台及其 SDK 环境。

5) 输出测试

FFmpeg 库编译完成后，建议调用 FFmpeg 和 FFplay 进行简单的测试。有的朋友问我：“我是开发 XXX 的，要是使用 FFmpeg 用哪个版本比较好?”。我的回答是：“你先开发，FFmpeg 同一个版本分支的接口不会有功能行的变化，基本不用修改你的代码就可以升级到新的版本，所以 FFmpeg 到到底使用哪个版本需要您自己做测试，可以放到后期产品测试当中进行.”。我说此话的原因，是因为 FFmpeg 库比较庞大不知道您都使用到了哪几个模块，所以最好做自己的产品测试。

6) 编译调试

看码千百遍不如上手调一遍。

Visual Studio 和 GDB 的调试器原理都是一样的，底层都是依赖于操作系统和 CPU 处理器的支持。

例如下面是一个断点调试过程调试器、操作系统和 CPU 配合工作流程：

你调试时在代码的某一行设定一个断点，实际是由调试器访问被调试的进程，找到其内存空间的代码段的对应源码行的代码块的第一个汇编指令，修改其汇编指令的第一个字节，改为 int 3 汇编指令，并保存原先这第一个字节的数据。

在程序运行时执行到此处后，就会引发一个 CPU 指令异常触发中断，CPU 会运行到操作系统的异常处理表，运行操作系统异常处理函数，经过操作系统的一系列执行和过滤，发现是 int 3 软中断指令，找到出现异常所属的进程，如果此进程出现的异常“无人”处理，就会将此进程异常交给系统处理，就是传说中的 Crash。如果此进程出现的异常有注册了的异常处理对象，即是调试器进程，下面就会进入调试阶段。

调试器进程就会收到，由操作系统通过 API 回调的调试消息，进入调试器的异常处理函数（这是系统 API），其中包含具体参数，异常的位置、类型、进程信息等。被调试进程会处于调试状态，调试器根据提前编译时输出的符号位置信息和源码进行分析，通过读写被调试器进程内存，计算出调试进程当前栈、堆、变量等具体值显示出来。

当用户调试操作完成后，准备开始继续运行，调试器就会将原先保存的第一个字节的数据恢复到代码内存，吃掉异常消息继续运行。如果还要保持下次还要运行到此断点处中断，就还要重新修改内存，等等上述动作的重复操作。

所以通过以上调试原理讲解完成后，你也可以自己写一个调试器（当然你要是有精力的话），讲这段文章的重点不是写一个调试器，也是告诉你调试的本质是一样的。

所以无论使用 VS 还是 GDB 调试都需要最主要的一样东西就是链接符号表（或叫调试符号表），这是调试器和编译器共同定义的文件格式，VS 就是.pdb 文件，GDB 就是.debug 文件（gcc 也可以将其保存在程序里），是源代码和 bin 之间的链接纽带，通常要比程序本身大很多，它主要保存的内容有进程堆和栈管理信息、所有函数的相对地址偏移、函数名称、参数信息、以及对应源代码在磁盘中的位置，源码文件名称、行号及其他信息等等。

所以调试必备：调试连接文件、bin 程序、源码、调试器。

调试文件是调试器和编译器共同认识协同工作的格式文件，下面就要说说编译器，看看编译器是怎么生成调试文件的，在这个之前先要看看编译过程，顺道也说说交叉编译。

编译过程最终要输出的是 bin 程序文件，程序文件是给操作系统引导最终由 CPU 进行指

令执行的。由于程序需要在操作系统上运行，就必须由操作系统管理，程序的执行也要由操作系统引导，由于操作系统的不同，他们所定义的执行文件格式和意义都不相同，一个 bin 程序也无法在不同操作系统下引导执行，并且一个程序运行都需要依赖不同的其他程序模块就更加深了平台内的关联性，所以一般支持多平台的源代码库，在编译的时候都要指定“目标平台”来生成不同的格式文件。这个只是系统关联和引导，真正执行不是操作系统而是 CPU，所以即使在不同平台最终生成的二进制的指令都是相同的，因为指令是由 CPU 执行的，理论上你手写机器码都可以执行，所以编译的时候就有“目标平台-目标处理器”。

当你编译的机器平台（即操作系统）也叫做“编译平台”，不同于目标平台时就是“交叉编译”了，交叉编译可以通过很多手段完成，例如一些交叉编译环境和中间件库：MinGW、NDK、Cygwin 等。

那关于 FFmpeg 库的编译和调试，下面举例主要是在 Windows 下编译运行调试。Windows 下的可执行程序.exe 或.dll 都是 Windows 自己定义的 PE 文件格式，所以最终生成文件的程序就只好使用 Widnows 自己的 link 连接器。庆幸的是 gcc 编译、icl Intel 的编译器以及 cl 微软自己的编译器所生成的.o、.obj 是基本相同的格式，保存的是 CPU 纯指令组成的各个函数的 CPU 指令块、以及数据段等。只需要使用微软自己的 link 连接器，组装成.dll 或是.exe 即可，只要在 link 的时候才会生成.pdb 文件，所以完全可以调试。

建议使用 intel 的编译器，intel 编译器和 VS 环境结合的很好，但是仍然需要修改代码，因为源码中调用了 Window 没有或是参数不一致的函数。如果你不想修改代码需要引入 MinGW 中间件的库，就是将 linux 函数对应实现在 windows 系统上，一般使用 ggc+MinGW+msys 配合使用。

以上关于编译、运行和调试的流程，都是从宏观上总体的讲解，如果你对这部分很感兴趣，想了解细节，可以自己查找相关资料学习，不要求快要追求学习质量，了解本质。

2. 各个模块框架介绍

FFmpeg/Libav 库共分为以下几个模块，每一个模块基本都依赖于 libavutil 公共模块，并且每一个模块都是一个独立的框架，FFmpeg 库就是将这些离散的功能框架组建起来而已。简单介绍下各个模块之间的依赖关系，基本所有模块都依赖于 libavutil 模块，它属于通用模块，而只有 libavformat 和 libavcodec 是强耦合，其他总体以离散为主。

1) libavutil 公共模块

util 代表通用的意思，其主要包含：

1) 基础和通用数据结构（类）定义和管理。
2) 媒体数据的各种通用格式，覆盖面广，内容庞大，建议仅了解涉及自己部分的媒体数据结构即可。其主要涉及音视频裸数据的各种信息，包括存储、传输、复制、释放。FFmpeg 时间处理和计算转换。
3) CRC 数据校验、MD5 算法、AES 加解密、Blowfish 加解密、DES 加解密、RC4 加解密、SHA-1/2 算法、DFT/FFT/FFTW、LLS 模型、PCA 分析、PSNR 计算、随机与排序、数据结构排序和管理、其他算术计算转换和算法等。
4) 字典参数管理，错误信息管理，DSP 抽象层管理。
5) 文件 IO 和网络 IO，MAC 信息，CPU 特性和检查。
6) 编译和运行系统兼容，64 位兼容控制管理，版本兼容控制管理。
7) 内存管理、打印和日志管理，字符串处理、字节序处理、Debug、Assert 等等。

综上所述不难看出，libavutil 是各个框架的基础，本身没有多些有意义的功能，主要以数据结构定义，通用接口和算法为主。libavutil 目录下对应 x86、arm、avr32、bfin、mips、ppc、sh4、tomi 等都是对应平台指令优化代码。

下表是 libavutil 模块常用数据结构简介：

结构体名称	类型	意义
AVClass	功能对象，固定分配，外部可见	是 FFmpeg 库的基础类，其主要提供：名称、元素类型、扩展信息、参数信息、版本、日志等级、其他扩展等。
AVClassCategory	枚举	表示类的类型
AVOption	功能对象，固定分配，外部可见	参数命令。属于 AVClass 的成员变量，表示这个类有哪些私有参数；是数组结构；参数之间可以归类、有最大值和最小值限制、有默认值、有类型和说明信息等。
AVOptionType	枚举	表示参数类型
AVPixelFormat	枚举	图像的像素格式
AVMediaType	枚举	媒体类型，包括：视频、音频、数据、字幕、附件等。
AVRational	typedef struct AVRational{	分数值，num 为分子，den 为分母。

	int num; int den; } AVRational;	常用于表示帧率、采样率、时间单位等。

2) libavcodec 编解码模块

libavcodec 主要有 AVCodec 音视频以及字幕编解码、AVParse 内部解析器、AVBitStreamFilter 音视频比特滤器、编解码需要到的一些算法、AVFrame 音视频数据帧、AVSubtitle 字幕数据包、AVPicture 视频图像、AVPacket 数据包等这几块也包含其中。

列举一些常用编解码器，所有 AVcodec 个数将近 400 个，而且还在不断增加中：
V is Video，A is Audio，S is Subtitle。

FFmpeg 标示名称	类型	说明
h264	V-Decoder/Encoder	AVC/H264 解码器，编码器使用的是 x264 库,也支持 h263、h261。
vp8	V-Decoder/Encoder	Google VP8 视频解码器，编码器使用的是 libvpx google vp8 库，也支持 vp3、vp5、vp6 等。
mpeg4	V-Decoder/Encoder	MPEG-4 part 2 编解码器，编码器也可以使用 libxvid 库，也支持 mpeg1video、mpeg2video、mpegvideo、msmpeg4v1、msmpeg4v2、msmpeg4。
vc1	V-Decoder/Encoder	SMPTE VC-1 编解码器,也支持 vc1image(Windows Media Video 9 Image v2）、硬件加速。
wmv2	V-Decoder/Encoder	Windows Media Video 8 编解码器，也支持 wmv1、wmv3 解码、wmv3image 解码(Windows Media Video 9 Image)。
rv20	V-Decoder/Encoder	RealVideo 2.0 编解码器，也支持 rv10、rv30 解码、rv40 解码。
avs	V-Decoder/Encoder	中国 avs 视频编解码器，编码器使用 libxavs/avs 库。
zlib	V-Decoder/Encoder	LCL (LossLess Codec Library) ZLIB 编解码器。
bmp	V-Decoder/Encoder	原始 RGB 数据，即是 Bmp 文件数据内容。
j2k	V-Decoder/Encoder	JPEG 2000 编解码器，编码器也可以使用 openjpeg 库，也支持 jpegls、mjpeg 等。
png	V-Decoder/Encoder	PNG (Portable Network Graphics) image 编解码器。
gif	V-Decoder/Encoder	Graphics Interchange Format 编解码器。
tiff	V-Decoder/Encoder	TIFF image 编解码器。
pgm	V-Decoder/Encoder	PGM (Portable GrayMap) image 编解码器,也支持 pgmyuv。
xbm	V-Decoder/Encoder	XBM (X BitMap) image 编解码器
rawvideo	V-Decoder/Encoder	原始 yuv 数据，FFmpeg 默认的 YUV420p 像素格式；即是 YV12、UYVY、YUYV 格式。也支持 yuv4、y41p。
mp3	A-Decoder/Encoder	MP3 (MPEG audio layer 3)解码器，编码器使用的是 mp3lame 库，也支持 mp1、mp2、wolame。
aac	A-Decoder/Encoder	AAC (Advanced Audio Coding)编解码器，编码器也可以使用 vo_aacenc、aacplus、faac 库。
ac3	A-Decoder/Encoder	ATSC A/52A (AC-3)编解码器
g726	A-Decoder/Encoder	G.726 ADPCM 编解码，也支持 g722、g723、g729 解码。
pcm_alaw	A-Decoder/Encoder	PCM A-law / G.711 A-law 编解码器

pcm_mulaw		PCM mu-law / G.711 mu-law 编解码器
speex	A-Decoder/Encoder	speech 编解码器，使用的是 speex 库。
amr_nb amr_wb	A-Decoder/Encoder	AMR-NB (Adaptive Multi-Rate NarrowBand)编解码器，使用 opencore 编码库；AMR-WB (Adaptive Multi-Rate WideBand)编解码器，使用 vo_amrwbenc 编码库。
real_144	A-Decoder/Encoder	RealAudio 1.0 (14.4K)编解码器，也支持 real_288 解码。
wmav2	A-Decoder/Encoder	Windows Media Audio 2 编解码器，也支持 wmav1、wmalossless 解码、wmavoice 解码。
gsm	A-Decoder/Encoder	GSM audio / voice 编解码器，也支持 gsm_ms。
vorbis	A-Decoder/Encoder	Vorbis 编解码器
srt	S-Decoder/Encoder	SubRip subtitle with embedded timing 字幕编解码器
ass	S-Decoder/Encoder	SSA (SubStation Alpha) subtitle 字幕编解码器
dvdsub dvbsub	S-Decoder/Encoder	DVD subtitles 字幕编解码器 DVB subtitles 字幕编解码器
subrip	S-Decoder/Encoder	SubRip subtitle 编解码器
pgssub	S-Decoder	PGS subtitle 解码器，HDMV Presentation Graphic Stream subtitles。

AVParse 解析器的作用是：输入容器向解码器之间的数据传递和转换，例如在 MP4 容器中取出 h264 的 sps、pps 信息交给解码器来处理。

AVBitStreamFilter 比特流滤器的作用是：编码器向输出容器之间的数据传递和转码，例如 MP4 容器输出中，需要将编码器输出的 sps、pps 给 MP4 容器来处理保存，并且要重组 ES 数据包，把起始符变为包大小。

3) libavformat 容器模块

libavformat 主要有 URLProtocol 协议、AVInputFormat 输入容器、AVOutputFormat 输出容器、AVStream 数据流（包括音视频流、字幕流、二进制流等等）、AVProgram 节目信息和 AVChapter 章节信息等。

关于 FFmpeg 的容器和协议的使用，容器的使用是通过函数指定选择的，例如指定名称“mpegts”就会指定使用 TS 容器格式。而协议的选用是通过传入的 URL 路径的协议头选择的，例如输入路径“http://www.strongffplugin.com/abc.mp4”程序会自动检索 URL，将 http 协议头提取出，选择对应的协议。

下面是在 FFmpeg 近 300 个输入输出容器中，列举一些常用的输入和输出容器：

FFmpeg 标示名称	类型	说明
rtp	Input/Output	RTP 容器
rtsp	Input/Output	RTSP 容器
rawvideo	Input/Output	原始视频 yuv 容器，对应使用 rawvideo 的编解码器
mp3	Input/Output	MPEG audio layer 3 容器
mp4	Input/Output	MPEG-4 Part 14 容器
rm	Input/Output	RealMedia 容器
mpeg	Input/Output	MPEG-1 Systems / MPEG program stream 容器
flv	Input/Output	Flash Video 容器
h264	Input/Output	H264 裸流容器，对应使用 H264 编解码器
hls	Input/Output	Apple HTTP Live Streaming 容器
webm	Output	WebM 输出容器
avs	Input	AviSynth 输入容器
image2	Input/Output	bmp,dpx,jls,jpeg,jpg,ljpg,pam,pbm,pcx,pgm,pgmyuv,png,ppm,sgi,tga,tif,tiff,jp2,j2c,xwd,sun,ras,rs,im1,im8,im24,sunras,xbm,xface 容器

下面是在 FFmpeg 近 20 来个协议中，列举一些常用的协议：

FFmpeg 标示名称	说明
file	文件 IO 处理协议
http	HTTP 协议
httpproxy	HTTP 代理协议
https	HTTPS 协议
pipe	管道协议
rtp	RTP 协议
tcp	TCP 协议
udp	UDP 协议

rtmp	RTMP 协议

4) libswscale 视频色彩空间转换

这个模块的主要功能就是色彩空间转换，而且都是经过汇编优化的，计算速度上可以算是 CPU 上的最好的了。

FourCC 标准网站：http://www.fourcc.org/

FFmpeg 支持色彩空间类型很多，下面是其内部定义 AVPixelFormat 枚举值，P 一般表示逐行，A 一般表示 alpha 透明度：

FFmpeg 标示名称	说明	参考标准	平均每像素比特数
AV_PIX_FMT_GRAY8	planar Y	GREY	8bpp
AV_PIX_FMT_YUV420P	planar YUV 4:2:0 Progressive (1 Cr & Cb sample per 2x2 Y samples)	YV12、IYUV、I420	12bpp
AV_PIX_FMT_YUV422P	planar YUV 4:2:2 Progressive (1 Cr & Cb sample per 2x1 Y samples)	YV16、YUY2、UYVY	16bpp
AV_PIX_FMT_YUV444P	planar YUV 4:4:4 Progressive (1 Cr & Cb sample per 1x1 Y samples)	IYU2	24bpp
AV_PIX_FMT_YUVJ420P	planar YUV 4:2:0, full scale (JPEG), deprecated in favor of PIX_FMT_YUV420P and setting color_range		12bpp
AV_PIX_FMT_NV12	planar YUV 4:2:2 ， 1 plane for Y and 1 plane for the UV components	NV12	12bpp
AV_PIX_FMT_YUVA420P	planar YUV 4:2:0 Progressive (1 Cr & Cb sample per 2x2 Y & A samples)		20bpp
AV_PIX_FMT_YUVA444P	planar YUV 4:4:4 Progressive (1 Cr & Cb sample per 1x1 Y & A samples)	AYUV	32bpp
AV_PIX_FMT_RGB24	packed RGB 8:8:8, RGBRGB...	BI_RGB、RGB24	24bpp
AV_PIX_FMT_BGR24	packed RGB 8:8:8, BGRBGR...	BI_RGB、BGR24	24bpp
AV_PIX_FMT_RGB0	packed RGB 8:8:8, RGB0RGB0...	BI_RGB、RGB32	32bpp
AV_PIX_FMT_RGBA	packed RGBA 8:8:8:8, RGBARGBA...	RGBA	32bpp

YUV 与 RGB 的转换公式：

$$
\begin{aligned}
Y &= 0.299 * R + 0.587 * G + 0.114 * B \\
U &= 0.436 * (B - Y)/(1 - 0.114) + 128 \\
V &= 0.615 * (R - Y)/(1 - 0.299) + 128
\end{aligned}
$$

YCC 与 RGB 的转换公式：

$$Y' = Kr * R' + (1 - Kr - Kb) * G' + Kb * B'$$

$$Pb = \frac{1}{2} * \frac{B' - Y'}{1 - Kb}$$

$$Pr = \frac{1}{2} * \frac{R' - Y'}{1 - Kr}$$

5) libswresample 音频重采样

FFmpeg 使用的是 libswresample 模块，而 libavresample 是 libav 里面的，FFmpeg 只是拿来“借鉴一下”也顺道使用，按以后的说法，会完全移植过来（前提人家没啥更新的了）再把 libavresample 移除掉，本来 FFmpeg 和 libav 就是一家所以这个也无需多虑。

主要功能是原始音频数据格式转换（简称重采样），可以修改采样精度（采样深度）、采样频率和声道相关信息。

采样精度的转换很简单，可以通过汇编指令完成大数据的流化处理。

至于采样音频的转码，算法也并不复杂，只要能深刻理解采样音频的意义就会明白。

最难的是声道信息的转换，涉及声道布局（即杜比音效），也是 FFmpeg 一直努力学习修改的地方，这方面老字号的还是 DOLBY，FFmpeg 对于专业来说只能算是勉强胜任，但 FFmpeg 对多声道转双声道的处理转换已经做的很好了，在低音和多声道等等方面还是很难达到优秀的地步，但毕竟只是拿它来转换工作也就足够了。

下表是 FFmpeg 的音频重采样的参数结构简介：

结构体名称	类型	意义
in_sample_fmt	AVSampleFormat（枚举）	采样格式（采样深度）
in_sample_rate	int	采样频率
in_channel_layout	uint64_t（宏值）	声道布局
in_channels	int	声道数

6) libavfilter 音视频滤器

其功能主要包括音视频滤器，这个也是 FFmpeg/Libav 正在完善的地方，逐步将 vlc、mplayer、mpc 等播放器的视频处理滤镜拿进 FFmpeg/Libav 中，再提供给它们使用。现在 avfilter 加起来有近 150 个 avfilter 左右。

下表是视频滤器 AVFilter 常用对象简介：

FFmpeg 名称	功能意义
blackdetect blackframe	黑场检测
crop	图像剪切
overlay	在顶部叠加视频源
delogo removelogo	消除 LOGO 图标模糊化 消除 LOGO 图标，根据一个 LOGO 输入
drawbox/drawgrid	绘制矩形框
drawtext	绘制文字，现根据 freetype2 实现
hue	色度和饱和度调整
swapuv	将 U 和 V 调换
curves	调整色彩曲线
format	色彩空间转换
scale	图像缩放和色彩空间转换
fps	强制设置帧率
unsharp	锐化
Framestep decimate	抽帧
hflip vflip	图像水平翻转 图像垂直翻转
stereo3d	转换为 3D 立体视觉视图
hqdn3d	高质量 3D 降噪
noise	添加噪声
idet	隔行扫描检测
yadif	消除隔行扫描
il interlace interleave kerndeint	隔行扫描和逐行扫描之间的转换，交织转换
field fieldmatch fieldmatch	隔行扫描，场处理相关
boxblur smartblur	淡入淡出
showwaves	分析音频，显示波形
showspectrum	分析音频，显示频谱

split	分路器
mp	引入 MPlayer 滤器
ass	引入 ass 字幕处理
frei0r frei0r_src	引入 frei0r 库滤器
pp	使用 libpostproc 滤器

下表是音频滤器 AVFilter 常用对象简介：

FFmpeg 名称	功能意义
afade	音频淡入淡出效果滤器
aformat	音频格式转换滤器，包括采样格式、采样频率、声道布局及声道数
amergae/join	将多个音频流合成一个音频流，增加声道数
amix	混音器
aresample	重采样（改变采样频率）
volume	调整音量
asetpts	设置 PTS，根据数据量设置时间戳
asetrate	不修改数据，设置采样频率参数
asplit	分路器
bass	低频消除（剪除低音部分）
biquad	IIR 滤波
ebur128 highpass lowpass	EBU R128 归一 loudness 滤波器 滤波器
equalizer	均衡器
silencedetect	静音检测
sine	输出正弦音频信号

7) libavdevice 设备输入和输出容器

其功能主要涉及和硬件相关的东西，包括音视频输入和输出，最终以 AVInputFormat 和 AVOutputFormat 输入和输出容器的形式，接入到 FFmpeg 库。

下表是 libavdevice 模块常用数据结构简介：

结构体名称	类型	管理	位置	意义
AVInputFormat	功能对象	固定	内部实现	输入容器
AVOutputFormat	功能对象	固定	外部接口	输出容器

8) libpostproc 视频后期处理

后期处理，主要是优化解码后的视频画面，再编码后无效。根据 PPFilter 唯一标示，其功能主要包括去宏块滤波、去隔行、消噪和校对。

下表是 libpostproc 模块常用数据结构简介：

结构体名称	类型	管理	位置	意义
PPFilter	模板参数	固定	内部实现	Postprocessing filter
PPMode	功能对象	固定	外部接口	Postprocessing mode
PPContext	数据对象	动态	外部接口	Postprocess context

9) ffplay 播放和 ffmpeg 转码

ffplay 是通过调用库接口实现的可执行播放器程序，ffmpeg 是通过调用库接口实现的可执行转码程序。他们都是使用相同的 libavutil、libavcodec、libavformat、libavswscale、libavfilter 等库模块实现的。

FFmpeg 可执行程序的总体转码流程：【输入 URL】to【协议】to【输入容器】to【可选项 - AVParse】to【解码器】to【原始数据】to【音视频、字幕滤器】to【原始数据】to【编码器】to【可选项 - 比特流滤器】to【输出容器】to【协议】to【输出 URL】。

FFplay 可执行程序的总体模仿流程：【输入 URL】to【协议】to【输入容器】to【可选项 - AVParse】to【解码器】to【原始数据】to【可选项 - 后期处理】to【音视频、字幕滤器】to【渲染播放】。

3. 基础设施

基础管理功能主要包括：库的初始化和释放、内存管理、日志管理、错误管理、线程和版本管理等。主要功能都属于公共模块使用范畴，重点了解一些基础数据结构，对于以后的学习和使用其方法都是相同或是相通。

从此以后的文章和代码关联性很大，所以将采用编写代码的风格，将文本以代码注释的形式，结合代码进行有效讲解。

1) 库管理

各个模块库的初始化：

```
av_register_all();          ///< 注册核心功能，包括编解码器、容器和协议。
avcodec_register_all();     ///< 注册编解码器，静态初始化编解码器数据信息。
avdevice_register_all();    ///< 注册设备输入和输出容器。
avfilter_register_all();    ///< 注册音视频滤器。
avformat_network_init();    ///< 初始化 Socket 库，以及涉及网络协议库信息。
```

各个模块库的释放：

```
avformat_network_deinit();  ///< 释放 Socket 库，以及涉及网络协议库信息。
```

2) 内存管理

void *av_malloc(size_t size);
分配一块字节数为 size 的内存区域，默认是从堆上，默认是按字节对齐，字节对齐的大小和 CPU 有密切方便汇编指令处理。

void *av_mallocz(size_t size);
由 av_malloc 分配，再通过 memset(,0,)将刚申请到的内存清零。

void *av_calloc(size_t nmemb, size_t size);
由 av_mallocz 分配，分配的大小是 nmemb 和 size 的乘积，即 nmemb 内存块数，size 每个内存块的大小。

void av_free(void *ptr);
释放由 av_malloc 系列所申请的指针，释放内存空间。
示例：void * pmem = av_malloc(512); av_freep(pmem);

void av_freep(void *arg);
由 av_free 释放，传入参数为指针的指针，释放完内存后将指针置空。
示例：void * pmem = av_malloc(512); av_freep(&pmem);

char *av_strdup(const char *s);
根据 s 字符串的长度，自动申请内存并且将字符串内容赋值到其中，返回新的字符串地址，返回值由 av_free 所释放。

void av_max_alloc(size_t max);
修改 av_malloc 的最大分配内存空间大小，默认大小是 INT_MAX。

void av_memcpy_backptr(uint8_t *dst, int back, int cnt);
重复做 memcpy 工作；dst 起始目的内存，连续空间后面的内存地址为 src 原始内存；back 为 src 原始内存空间大小；cnt 为要做 memcpy 的内存块个数。

3) 日志管理

void av_log(void *avcl, int level, const char *fmt, ...);
void av_vlog(void *avcl, int level, const char *fmt, va_list);
【功能】：输出日志信息；av_vlog 只是使用了 va_list 参数进行格式化。
【avcl】：为所在模块句柄，是 AVClass 的指针的指针，根据 AVClass 具体信息进行格式输出，可以做功能区分，功能模块地址打印等等。
【level】：日志级别，主要分为 4 类：

```
#define AV_LOG_ERROR    16          ///< 错误
#define AV_LOG_WARNING  24          ///< 警告
#define AV_LOG_INFO     32          ///< 信息
#define AV_LOG_DEBUG    48          ///< 调试
```

级别从高到低依次递减，数值依次递增，还有一些细化级别具体查看 Log.h 文件，数值也可以自己定义只要在这个范围内，可以自己加。
【fmt】：输入格式化字符串。

int av_log_get_level(void);
void av_log_set_level(int);
【功能】：获取和设置打印级别，默认是 AV_LOG_INFO 级别，只有高于此级别（即小于此数值），才会打印输出。

av_log_set_flags(AV_LOG_SKIP_REPEATED);
【功能】：设置 LOG 输出跳过相同的消息内容的日志输出。

void av_log_set_callback(void (*)(void*, int, const char*, va_list));
void av_log_default_callback(void* ptr, int level, const char* fmt, va_list vl);
【功能】：日志的回调管理，可以注册全局唯一回调函数进行日志输出管理。
【样例】：
第一步：设置回调函数

```
av_log_set_callback(ffmpeg_log_callback);   ///< 设置 LOG 回调函数
```

第二步：回调函数中添加处理的代码。
【功能】：不是根据日志消息内容，根据日志消息输出模块，通知同一个模块允许输出消息的次数。

```
#define ALLOW_CONTINUOUS_SAME_ADDR_NUMBERS 8
/* 同模块地址，最大允许连续输出的 LOG 数 */
char buffer_log[1024*1024] = {0};       ///< 缓冲区
void * log_addr = NULL;                 ///< 控制变量
int allow_log_number = 0;               ///< 控制变量
static void ffmpeg_log_callback(void *ptr, int level, const char *fmt, va_list vl)
{
    if(level <= AV_LOG_WARNING)     ///< 只处理大于警告级别的日志消息
    {
```

```
        if(ptr)
        {
            /* 控制同模块地址，最大允许连续输出的LOG数 */
            if(log_addr == ptr)
            {
                if(allow_log_number>ALLOW_CONTINUOUS_SAME_ADDR_NUMBERS)
                {return;}
                allow_log_number++;
            }
            else
            {log_addr = ptr;allow_log_number = 0;}
        }
        int prifix = 1;

        /* 调用FFmpeg库函数，格式化字符串 */
        av_log_format_line(ptr,level,fmt,vl,buffer_log,sizeof(buffer_log),&prif
    ix);
        if(level <= AV_LOG_ERROR)
        {writeError(buffer_log);}   ///< 其他日志系统对接输出接口
        else
        {writeWarning(buffer_log);} ///< 其他日志系统对接输出接口
    }

    av_log_default_callback(ptr,level,fmt,vl); ///< 回调FFmpeg库输出函数
}
```

4) 错误管理

关于 FFmpeg 错误管理是在 C 运行时库（C Run-Time Library）的基础上扩展的，根据函数的返回值 int 进行判断区分的，成功一半返回值是大于等于 0（>=0），错误的返回值为负数，错误值继承 C 运行时库的错误值，扩展了自己的错误值定义在 libavcodec/error.h（较老版本位置）或是 libavutil/error.h（较新版本位置）头文件当中。

FFmpeg 实现了一些函数翻译返回值的信息例如：

```
int av_strerror(int errnum, char *errbuf, size_t errbuf_size);
```

是将 int 类型的返回值翻译成字符串，errnum 是输入的错误值，errbuf 输出的字符串缓冲区的地址，errbuf_size 是输出的字符串缓冲区的大小，一般字符串缓冲区最大是 64 字节由宏定义。

5) 线程管理

关于 FFmpeg 多线程管理，现阶段主要涉及编解码和网络通信部分，并不涉及到别的模块，并没有一个统一的模块做多线程池复杂调度管理，而是简单的通过一些宏进行代码划分选择而已。

现阶段关于多线程主要是 3 种实现：pthread 跨平台库、windows 的 threads API 和 linux/unix 的 os2 threads API。主要涉及线程管理的创建、挂起、停止、同步。

在预编译的时候通过-pthreads、-w32threads、-os2threads 相关命令传输选择使用对应的多线程库。代码中通过宏 HAVE_THREADS 标示是否使用了多线程库，然后依次由宏 HAVE_PTHREADS、HAVE_W32THREADS 和 HAVE_OS2THREADS 进行区分。

由于是通过宏，所以最终接口名称都归约到 pthread 接口上。

下面简单列举几个常用接口：

```
1.  pthread_create              ///< 创建线程
2.  pthread_join                ///< 结束线程，等待正常退出
3.  pthread_mutex_init          ///< 创建互斥锁
4.  pthread_mutex_destroy       ///< 销毁互斥锁
5.  pthread_mutex_lock          ///< 互斥加锁
6.  pthread_mutex_lock          ///< 互斥解锁
7.  pthread_cond_init           ///< 创建信号量
8.  pthread_cond_destroy        ///< 销毁信号量
9.  pthread_cond_signal         ///< 激发一个信号量
10. pthread_cond_broadcast      ///< 激发所有信号量
11. pthread_cond_wait           ///< 等待信号量激发
```

6) 版本管理

登陆到 http://www.ffmpeg.org/download.html 官方网站中，可以看有 FFmpeg2.1、FFmpeg2.0.2、FFmpeg1.2.4 等等，它们都属于 FFmpeg 的版本分支，如若更新都有相同功能的代码，只是接口不同而已。

每个模块也都有自己的版本，版本数值通过宏定义到各个模块的 Version.h 头文件当中，版本号通过主版本号、小版本号和微小版本号组成。也都有相应模块的函数查询，例如通用模块的 avutil_version 函数获取版本信息，avutil_configuration 获取 configuration 预编译配置字串，avutil_license 获取授权字串等信息。

如下是列举了 FFmpeg2.1 版本的模块版本信息：

```
libavutil      52. 48.100
libavcodec     55. 39.100
libavformat    54. 63.104
libavdevice    54.  3.103
libavfilter     3. 42.103
libswscale      2.  2.100
libswresample   0. 17.102
libpostproc    52.  2.100
```

如果您是项目经理、或是 TeamLeader 此小结必须看，如果您把 FFmpeg 用到了自己的项目当中版本管理是最头疼的一个问题。

列出以下几个原则：

1. 同一个版本分支的升级版本，各个模块结构不会变，所以可以完整性升级。
2. 如果您以前是 FFmpeg-1.2 版本现在想升级到 FFmpeg-2.1 版本，这时候需要注意模板的版本信息，如果同模块的主版本一样，说明此模块接口未变；那么不一样这时候，可能需要您修改相应模块接口的调用代码，才使用新版本的模块。

4. 编解码应用开发

关于 FFmpeg 的编解码器都属于 libavcodec 模块，下面开始介绍编解码模块常用数据结构，将会详细说明每个结构的具体内容。

下表是编解码模块常用数据结构介绍：

结构体名称	类型	意义
AVCodec	功能对象，固定分配，外部可见	编解码器
AVHWAccel	功能对象，固定分配，外部可见	编解码硬件加速器
AVCodecContext	数据对象，动态管理，外部可见	编解码核心类
AVCodecID	枚举	编解码器类型 ID 值

下面只列举了 AVCodecContext 一些常用的、核心成员变量：

成员声明名称	成员声明类型	意义
av_class	const AVClass *	继承类。
codec_type	enum AVMediaType	媒体类型，视频、音频、字幕、数据、附件等。
codec	const struct AVCodec *	编解码器功能对象
hwaccel	struct AVHWAccel *	硬件加速器功能对象
codec_tag 和 stream_codec_tag	unsigned int	FourCC、mpeg、ITU-R 等标准 ID 值，stream_codec_tag 为容器层参考数据，在 FFmpeg 中不做为唯一参考 http://www.fourcc.org/。
priv_data	void *	编解码器私有数据，插件开发使用。
internal	struct AVCodecInternal *	编解码器内部数据，框架管理使用。
opaque	void *	编解码器自定义数据，应用开发调用者使用。
hwaccel_context	void *	硬件加速器数据对象上下文
extradata extradata_size	uint8_t * int	头信息和头信息大小，sps+pps、adts、haffuman table 等信息。
subtitle_header subtitle_header_size	uint8_t * int	字幕编解码扩展头信息
pix_fmt	enum AVPixelFormat	图像像素格式
sample_fmt	enum AVSampleFormat	音频采样格式（采样深度类型）
time_base ticks_per_frame	AVRational int	音频采样率，视频帧率。 1 表示 time_base 是帧率，2 表示 time_base 是场率

width，height coded_width ， coded_height	int	图像宽高 编解码图像宽高
sample_rate	int	音频每秒采样频率
channels	int	音频声道数
bits_per_raw_sample	int	原始音频采样点平均比特数，原始视频像素点平均比特数
bits_per_coded_sample	int	编解码音频采样点平均比特数，编解码视频像素点平均比特数
sample_aspect_ratio	AVRational	视频编码的宽高采样比 或是 音频编码的采样比
channel_layout	uint64_t	音频声道布局
frame_size	int	音频编解码每个声道的采样数
block_align	int	音频编解码每个数据包字节对齐大小
cutoff	int	音频截止带宽
frame_number	int	统计编解码帧个数
frame_bits	int	统计编解码比特数
field_order	enum AVFieldOrder	图像扫描方式
bit_rate	int	平均比特率，单位 bps； bit_rate_tolerance 码率的波动参考差值。
draw_horiz_band	回调函数地址	解码器插件开发，多线程使用，提高缓存使用率，可以不调用。
get_buffer2	回调函数地址	解码器应用开发，推模式使用，需要配合 CODEC_CAP_DR1 表示，降低由于数据缓冲所造成的帧延时，可以不调用，如若不使用就是拉模式。
get_format	回调函数地址	协商图像像素格式
execute/ execute2	回调函数地址	解码器应用开发，多线程编解码执行
refcounted_frames	int	音视频解码内存帧个数，视频最少为 1
thread_count	int	编解码器多线程个数
Profile level slice_count/slices max_b_frames has_b_frames delay refs gop_size keyint_min rc_max_rate	int/float	音视频编码模板 音视频编码级别 一帧的 slice 个数，并行编解码 最大连续 B 帧数 常表示是否有 B 帧，以及 B 帧数 延时帧数，有 B 帧情况 最大参考帧数 IDR-GOP 大小，即最大 I 帧间隔，多少个 I 帧之间有一个 IDR 帧 I-GOP 大小，即最小 I 帧间隔，多少个参考帧之间有一个关键帧 最大码率

rc_min_rate me_method i_quant_factor i_quant_offset b_quant_factor b_quant_offset qmin qmax max_qdiff noise_reduction scenechange_threshold rc_buffer_size rc_initial_cplx rc_min_vbv_overflow_use rc_max_available_vbv_use rc_initial_buffer_occupancy lmin lmax global_quality compression_level debug		最小码率 视频编码的运动估计算法 I 和 P 帧之间量化因子 I 和 P 帧之间量化偏移 IP 和 B 帧之间量化因子 IP 和 B 帧之间量化偏移 质量最小量化值 质量最大量化值 两帧间最大量化差值 编解码消噪强度 场景变化的检测值 编解码预设比特缓冲区大小 模式 1 码控复杂度 最小 vbv 溢出数 最大 vbv 可用数 编解码初始化比特缓冲区大小 质量最小量化值（拉格朗日乘积） 质量最大量化值（拉格朗日乘积） 全局质量，MPEG 类编码 压缩级别，音频、PNG 编码 调试
flags flags2	int	**【这部分涉内容较多，仅挑出常用的一些标示讲解，会很难在遇到问题时在返回仔细查看】** **CODEC_FLAG_* 和 CODEC_FLAG2_* 编解码器标示，可以输入也是属性值，非常重要的值。** 例如： CODEC_FLAG_PASS1 模式 1 默认，同步拉数据。 CODEC_FLAG_PASS2 模式 2，回调推数据。 CODEC_FLAG_PSNR 编码时 PSNR 计算。 CODEC_FLAG_LOW_DELAY 编码时强制延时输出，也就类似于解码时无 B 帧，强制逐输出，后果自负。 CODEC_FLAG_LOOP_FILTER 编码指定 deblock loop filter 功能。 CODEC_FLAG_CLOSED_GOP 是否 open gop 参考。 CODEC_FLAG_INTERLACED_ME 在 MPEG 编码做隔行运动估计。 CODEC_FLAG2_FAST 可以使用非规范技术做解码快速处理。

		CODEC_FLAG_GLOBAL_HEADER 编解码时使用编解码器里全局唯一的头，相当于替换码流里面的头。 CODEC_FLAG2_LOCAL_HEADER 编解码时使用本地码流里面的头，相当于实时替换了全局头。 CODEC_FLAG_TRUNCATED 输入数据的边界是任意字节位置，需要逐字节比特流格式过滤缓冲。 CODEC_FLAG2_CHUNKS 输入数据的边界是包或是单元位置，可以避免逐字节比特流格式过滤。 CODEC_FLAG2_IGNORE_CROP 丢弃 sps 信息。 CODEC_FLAG2_STRICT_GOP 设置、严格执行 gop 的大小。 CODEC_FLAG2_SHOW_ALL 正常解码直到第一个关键帧才输出，这个表示即使第一个帧不是关键帧也照样输出。

AVFrame 用于存储，编码前的原始音视频数据，一帧只能存放视频或是音频。
下面只列举了 AVFrame 一些常用的、核心成员变量：

成员声明名称	成员声明类型	意义
width, height pict_type	int enum AVPictureType	视频的宽高 图像解码帧类型 I、P、B、S 或场
nb_samples channels	int int	音频的采样率 音频声道数
format	int	视频像素格式 或 音频采样格式
key_frame	int	0 不是关键帧 ，1 是关键帧
sample_aspect_ratio	AVRational	视频编码的宽高采样比 音频编码的采样数
pts	int64_t	编解码使用的 pts 值，也是播放显示默认使用的 pts。
pkt_pts	int64_t	编码后，给容器层输出的 pts 值 解码后，从容器层获取的 pts 值
pkt_dts	int64_t	编码后，给容器层输出的 dts 值 解码后，从解码器获取的 dts 值
pkt_duration	int64_t	帧时长
pkt_pos pkt_size	int64_t int	所在容器层的物理位置 容器层包大小
coded_picture_number	int	编解码图像的序号数
display_picture_number	int	显示图像的序号数
quality	int	质量 1 到 32767 之间
opaque	void *	应用开发用户传输的私有数据

metadata	AVDictionary *	允许传递的媒体信息，用于应用开发
error	uint64_t [AV_NUM_DATA_POINTERS];	在编解码哪个平面时出错
#define AV_NUM_DATA_POINTERS 8		
data linesize	uint8_t * [AV_NUM_DATA_POINTERS] int [AV_NUM_DATA_POINTERS]	音频数据： 正常仅使用 data[0]保存数据,而 linesize[0]是对应的数据大小。 视频数据： data 和 linesize 有 8 个数组指针，默认是分别保存平面数据。 linesize 是对齐行大小，在其他库中也用 Stride、Pitch 来表示 例如： AV_PIX_FMT_YUV420P 格式 data[0]存放的是 Y 平面 data[1]存放的是 U 平面 data[2]存放的是 V 平面 AV_PIX_FMT_RGB24 格式 data[0]存放的是 RGB 混合平面 AV_PIX_FMT_NV12 格式 data[0]存放的是 Y 平面 data[1]存放的是 UV 混合平面 linesize 就是对应平面的，行（宽）像素对齐后的内存大小。
extended_data	uint8_t **	是 data 总的数据指针
extended_buf nb_extended_buf	AVBufferRef ** int	音视频引用数据，常用于滤器处理

AVPacket 用于存储，编码后的输出字节流数据，也是容器封装以及解析的数据包。
下面只列举了 AVPacket 一些常用的、核心成员变量：

成员声明名称	成员声明类型	意义
data size	uint8_t * int	数据包地址指针和数据包大小
buf	AVBufferRef *	数据引用，常用于解析器和比特流过滤。
stream_index	int	此数据包所属 AVStream 的流 ID 值
dts	int64_t	容器层 dts 值
pts	int64_t	容器层 pts 值
flags	int	帧标示，AV_PKT_FLAG 表示是关键帧
side_data	struct {	容器使用，常用语容器插件开发，比特

side_data_elems	uint8_t *data; int size; enum AVPacketSideDataType type; } *;	流滤器，解析器等。

在进行编解码之前需要调用avcodec_register_all进行编解码器的注册，编解码器默认是拉模式数据。

AVPacket保存的是字节流数据包属于容器层，保存的是编码后或解码前的数据。AVFrame保存的是原始数据帧属于编解码层，保存的是解码后或编码前的数据。在FFmpeg中AVFame就是帧，但在编码器开发时候，由于有逐行扫描和隔行扫描之分，在隔行扫描的时候一帧数据分为两场，低场和顶场，一场数据或是逐行一帧数据都看成一样的输入单元，在编码器术语中都称为帧（此帧非彼帧也）。

1) 编码流程（Encoding）

第一步：找到编码器

根据 ID 值找一个最先注册编解码器
```
AVCodec *codec = avcodec_find_encoder(AV_CODEC_ID_H264);
```

根据名称找一个最先注册编解码器
```
AVCodec *codec = avcodec_find_encoder_by_name("libx264");
```

第二步：为编码器初始化参数

```
AVCodecContext * c = avcodec_alloc_context3(codec);     ///< 申请编解码器上下文
```

视频编码器设置视频参数
```
c->width = 352;                                  ///< 宽
c->height = 288;                                 ///< 高
c->bit_rate = 400000;                            ///< 码率
c->time_base= (AVRational){1,25};                ///< 帧率
c->pix_fmt = AV_PIX_FMT_YUV420P;                 ///< 像素格式
av_opt_set(c->priv_data, "preset", "slow", 0);      ///< 编码器私有参数设置
av_opt_set(c->priv_data, "profile", "main", 0);
av_opt_set(c->priv_data, "level", "2.0", 0);
```

音频编码器设置音频参数
```
c->sample_fmt = AV_SAMPLE_FMT_S16;               ///< 音频采样类型（即采样深度）
c->sample_rate = 44100;                          ///< 采样率
c->channels = 2;                                 ///< 声道数
c->channel_layout = AV_CH_LAYOUT_STEREO;         ///< 声道布局
c->bit_rate = 64000;                             ///< 码率
```

音视频参数必须设置，否则编码器根本不知道怎么编码。

第三步：打开编码器

```
avcodec_open2(c, codec, NULL);       ///< 打开编码器
```
其中第三个参数是传入的参数命令集，此处可以传入编解码器的私有参数进行赋值；此处会预先编码或是组装出头信息。

第四步：初始化输入帧信息

已经通过第二步添加的编码参数，输入对应格式的原始数据。

```
AVFrame *frame = avcodec_alloc_frame();     ///< 申请可以保存原始数据的音视频帧
```

视频初始化：

```
frame->format = c->pix_fmt;          ///< 像素格式
frame->width  = c->width;            ///< 宽
frame->height = c->height;           ///< 高

/* 为AVFrame分配内存空间 */
av_image_alloc(frame->data,frame->linesize,
        c->width, c->height,
        c->pix_fmt,
        32);                         ///< 按bit对齐数
```

音频初始化：

```
frame->nb_samples     = c->frame_size;          ///< 采样率大小
frame->format         = c->sample_fmt;          ///< 采样格式
frame->channels       = c->channels;            ///< 声道数
frame->channel_layout = c->channel_layout;      ///< 声道布局
frame->linesize[0]    = 原始音频数据一个包的大小;
frame->extended_data = frame->data[0] = av_malloc(frame->linesize[0]);
```

原始音频数据1秒钟字节数 = 声道数 * 采样格式字节数 * 采样频率。
原始音频数据一个包的大小：如果是转码一般就是解码原始文件出来的原始音频大小；如果是自己采集的原始数据，这个值需要参考编码器算法。例如一般使用快速傅里叶变换进行的音频编码器，每个包的数据大小一般就是【2的N次幂采样 * 声道数 * 采样格式字节数】当然其他数据大小可以增加对齐。

第五步：填充原始音视频数据-编码输出字节流

```
AVPacket pkt;                                ///< 保存编码输出流
av_init_packet(&pkt);                        ///< 初始化AVPacket信息

while(true)     ///< 循环的输入帧数据和获取编码输出数据流
{
  /*  填充原始音视频数据到AVFrame中，给编码器读取  */

  音频原始数据填充：
  frame->linesize[0]= 原始音频数据字节流，包大小（主要和采样音频、位深、声道数、
  算法、时长有关）
  frame->data[0] = 数据地址

  视频原始数据填充：
  例如：AV_PIX_FMT_YUV420P 格式的数据填充 。
  frame->data[0] = 数据Y平面的地址
  frame->data[1] = 数据U平面的地址
```

```
  frame->data[2] = 数据 V 平面的地址
  frame->linesize[0] = 数据 Y 平面的行对齐的字节大小
  frame->linesize[1] = 数据 U 平面的行对齐的字节大小
  frame->linesize[2] = 数据 V 平面的行对齐的字节大小
```

行对齐的大小，是数据内存字节对齐，例如：AV_PIX_FMT_YUV420P 格式，图像宽高 721x576，如果 Y 数据行对齐按 4 字节大小对齐，则 linesize[0] = 724，linesize[1] = 362，linesize[1] = 362（注意：一般对齐的字节大小和编码算法有关）。

此处涉及视频数据格式和数据填充方式，具体信息在后面详细介绍，请参考后面小结关于 AVPixFmtDescriptor 数据结构的详细介绍。

```
  /* 编码音视频数据 */
  int ret = 0, got_output = 0;
  ret = avcodec_encode_video2(c, &pkt, frame, &got_output); ///< 视频编码函数
  ret = avcodec_encode_audio2(c, &pkt, frame, &got_output); ///< 音频编码函数
  if (ret < 0) {
      /* 编码失败要停止编解码器，重新初始化编解码器 */
      fprintf(stderr, "Encoder Failed\n");
      break;    ///< 具体看失败原因，有的还可以继续编码
  }
  if(got_output > 0){
      /* 有编码数据的输出，输出结构保存在 pkt 中 */
      av_free_packet(&pkt);     ///< 记得 pkt 使用完要释放
  }
}

///< 推出编码器缓冲的数据流
///< 注意：视频编码因为有 B 帧、参考帧等，编码器会缓存数据，最后需要一一推出
执行一遍编码流程
```

第六步：关闭编码器

```
avcodec_close(c);                          ///< 关闭编解码器
av_free(c);                                ///< 释放编解码器上下文
avcodec_free_frame(&frame);                ///< 释放帧数据
```

2) 解码流程（Decoding）

第一步：找到编码器

根据 ID 值找一个最先注册编解码器
```
AVCodec *codec = avcodec_find_encoder(AV_CODEC_ID_H264);
```

根据名称找一个最先注册编解码器
```
AVCodec *codec = avcodec_find_encoder_by_name("h264");
```

第二步：打开编码器

```
AVCodecContext * c = avcodec_alloc_context3(codec);     ///< 申请编解码器上下文

avcodec_open2(c, codec, NULL);      ///< 打开编码器
```
其中第三个参数是传入的参数命令集，此处可以传入编解码器的私有参数进行赋值；此处一般情况下，会预先解码出头信息准备解码数据。

第三步：填充字节流数据-解码输出原始音视频数据

```
AVPacket pkt;                                   ///< 保存编码输入流
AVFrame * frame;                                ///< 保存输出原始音视频数据
av_init_packet(&pkt);                           ///< 初始化 AVPacket 信息
frame = avcodec_alloc_frame();                  ///< 初始化 AVFrame 信息

int len = 0;
while(true)          ///< 循环的输入码流数据和输出原始音视频帧数据
{
    ///< 填充码流数据
    avpkt.data = 码流数据地址，由 av_malloc 申请出来的;
    avpkt.size = 码流数据大小;

    ///< 视频编码
    len = avcodec_decode_video2(avctx, frame, &got_frame, pkt);
    ///< 音频编码
    len = avcodec_decode_audio4(c, decoded_frame, &got_frame, &avpkt);
    if (len < 0) {
        ///< 转码出现错误，跳出解码流程，也可以多次失败后跳出
        break;
    }
    if (got_frame) {
```

```
        ///< 有解码输出，输出信息在 decoded_frame 中，可以参考编码的原始音视频数据填充
    }

    ///< 注意可以需要继续推流
    if (pkt->data) {
            pkt->size -= len;
            pkt->data += len;
```

如果 len 是解码所用的字节流长度，如果使用长度 len 小于 pkt->size，所有还有部分剩余数据没有推入解码器，这部分数据可以再次推入解码器，进行解码操作。

```
    }
}

///< 推出解码器缓冲的视频帧
avpkt.data = NULL;
avpkt.size = 0;
```

执行一遍解码流程。。。。。。。

第四步：关闭编码器

```
avcodec_close(c);                       ///< 关闭编解码器
av_free(c);                             ///< 释放编解码器上下文
avcodec_free_frame(&frame);             ///< 释放帧数据
```

3) 音视频数据格式讲解（Audio and Video Format）

音频格式讲解：

声明名称	成员声明类型	意义
AVSampleFormat	枚举	音频采样格式（音频采样深度类型） 主要有：1,2,4 字节整数值，float，double，有符号和无符号，大段和小端模式，bit 和 planar。
channel_layout	int64_t	是通过宏进行按位&和|进行判断的。 例如：AV_CH_LAYOUT_MONO 表示单声道中间布局。
channel_layout_map	内部对象	完整描述了支持的不同声道类型。

WAV ：是微软定义的一种容器格式，存储音频数据，也叫存储原始波形文件。
PCM：Pulse-code modulation（中文称脉冲编码调制，即原始音频数据），ADPCM 自适应差分脉冲编码调制，丢失一定可接受的数据。

数据计算：
音频 PCM 的数据大小 = 声道数 * 采样位深 * 采样频率 * 秒
例如：双声道、采样频率 44100、采样格式 float
2 秒数据大小 = 2 * 4 * 44100 * 2 = 705600 字节。

在 AFrame 中，data[0]是数据地址，linesize[0]是数据大小，一般情况下多声道 PCM 都是混合在 data[0]，极个别的情况会把声道展开。

视频格式讲解：

声明名称	成员声明类型	意义
AVPixelFormat	枚举	视频像素格式
AVPixFmtDescriptor	结构体	像素格式描述，主要包括各个像素格式数据在结构体中存储的位置信息，swscale 转换的时候常使用到。
AVComponentDescriptor	结构体	属于 AVPixFmtDescriptor 成员，主要描述像素格式在各个平面的信息。
AVColorPrimaries、AVColorTransferCharacteristic、AVColorSpace	枚举	主要描述像素格式不同标准的定义，主要用于编解码器

在 AFrame 中一般情况，YUV 和 RGB 平面根据 AVPixelFormat 格式定义分别保存在 data 和 linesize 数组中，data 是平面地址，linesize 是平面行对齐的大小。

4) 高级编解码流程讲解（Advanced codecs Process）

在讲解高级编解码流程前先要说一说多媒体框架的流程分类，无论是转码、播放器还是媒体服务器应用程序，还是协议、容器、编解码器、录制、渲染和合成处理等都组成一个链路，这个链路的数据流程分为 2 类：推模式和拉模式。

最典型的框架就是 DShow 的 filter 框架，其实 ffmpeg、vlc、mplayer、live555、avs、dshow 等多媒体框架，无论应用到播放器、转码还是服务器，都可以看成是 filter graphic 或是 filter link。其思想是将采集、录制、输入输出容器、编解码器、音视频处理等等单元都看做是 filter，其播放器、转码器或是流服务器等应用都是由各种各样 filter 组成的链路（filter graphic 或 filter link），filter 和 filter 之间的数据传递通过统一的接口进行，这个接口称为 pin，而由 filter 组成的链路根据其启动线程的动力管理，又分为推模式或拉模式链路，而每个 filter 本身也分为推模式和拉模式 2 种接口。

综上所述我们理解了 filter link、filter graphic、filter、pin、推模式和拉模式，还有一个需要理解的是，启动发动力，它可以是一个线程也可以是多个线程，像 DShow 这种框架在使用时，其实启动动力线程库以及管理已经由框架帮我们写完了，而 FFmpeg 库没有，可以我们自己来实现动力线程（Power threads），在 ffmpeg.exe 和 ffplay.exe 的代码实现中，这根动力线程就是主线程。

下面来讲解整个 filter 链路的推模式和拉模式：

1. 推模式

转码流程一般属于推模式。例如 ffmpeg.exe 转码流程也是推模式，这个主要是看启动线程控制的数据流程，从数据源读取解析得到一包数据流，将这包数据推入解码器，再将解码器输出的数据推入 AVFilter，再将 AVFilter 输出的数据推入编码器，再将编码输出的数据流推入输出容器，整个链路就形成的推模式，特点是不考虑最后结尾 filter 的速率，只关心开头 filter 输入速率，将一包包数据压入链路当中。

2. 拉模式

播放流程一般属于拉模式，因为最后播放视频和音频需要和时间密切相关，渲染如果采样拉模式更加合理。例如 ffplay.exe 的播放流程就是属于拉模式，拉模式可以在结尾 filter 控制输出的数据帧的速率，渲染处理的线程定时去缓冲队列去取数据，如果没有数据就会一层一层的向上要数据。

以上是从 filter 链路总体上数据流程来看是推模式还是拉模式，实际每个 filter 也有自己的推模式和拉模式，因为编解码器、输入输出容器、或是录制和播放都有可能缓冲数据，所以有缓冲数据功能 filter 就会有推和拉 2 种模式。

而 FFmpeg 在实际开发中编解码器推拉模式：

1. 拉模式

编解码器的拉模式，就是本章前 2 小结所将的流程。

编码器推入数据帧，通过参数返回立即检查是否有数据包输出，然后循环推入数据帧，同时检查数据包输出信息，直到没有输入数据帧，这时需要推入空的数据帧，将编码器内剩余的缓冲数据编码输出数据包。

解码器也是一样，推入数据包，通过参数返回立即检查是否数据帧输出，然后循环推入数据包，同时检查数据帧输出信息，直到没有输入数据包，这时需要推入空的数据包，将解码器内剩余的缓冲数据解码输出数据帧。

2. 推模式

编解码器的推模式是通过回调函数来实现的，下面先讲解具体流程，然后再针对 FFmpeg 做细致的 API 解读，不过推模式在需求上慢慢会被拉模式所取代。

编码器循环推入数据帧，如果有编码完的数据包输出，通过提前注册好的回调函数来处理编码输出的数据包。

解码器循环推入数据包，如果有解码完的数据帧输出，通过提前注册好的回调函数来处理解码输出的数据帧。

解码器推模式：

下面是 AVCodecContext 的 2 个回调函数原型

```
///< 回调函数解码数据帧
int (*get_buffer2)(struct AVCodecContext *s, AVFrame *frame, int flags);
///< 回调函数解码数据场单位
void (*draw_horiz_band)(struct AVCodecContext *s,const AVFrame *src, int
offset[AV_NUM_DATA_POINTERS],nt y, int type, int height);

AVCodecContext * c = avcodec_alloc_context3(codec);     ///< 申请编解码器上下文
c.get_buffer2 = 自定义回调函数地址

/*
如果注册了回调函数，建议不设置这个。
if(codec->capabilities&CODEC_CAP_TRUNCATED)
    c->flags|= CODEC_FLAG_TRUNCATED; /* we do not send complete frames */
*/
avcodec_open2(c, codec, NULL);      ///< 打开编码器

///< 开始解码操作… …
```

5. 容器和协议应用开发

关于 FFmpeg 的输入输出容器和 IO 协议都属于 libavformat 模块，下面开始介绍容器和协议的概念、关系组成、核心数据结构、功能等。

根据前部分的知识回忆下，容器和协议的概念和关系，容器只是空壳、是衣服、是数据存放的格式，如果容器是盒子，那么协议就是传送带、是 IO 传输层。

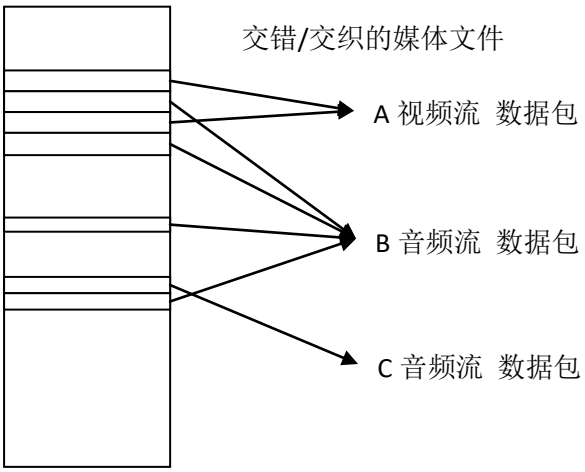
在这个基础上我们开始说下容器的组成，容器中实质存储的是数据流，如果容器中没有数据流就只是空壳容器。

数据流是一种抽象概念，是由一个个数据包组成的，存放在容器当中，视频有视频的数据流，音频有音频的数据流，他们之间是相互独立的。

数据流之间在内容上没有任何关联，他们之间都是相互独立的。

数据流之间在空间上它们同处在一个容器当中，由于数据流由数据包组成，由于一个容器存在空间和时间上存在顺序性，所以多条数据流存在容器当中，各个数据流的数据包的存放顺序就分为交错和非交错。交错在物理顺序上，是第一个可能是视频 A 的流数据包，第二个包可能是音频 C 的流数据包等等交叉出现。非交错是在物理顺序上各个流数据包存放没有交叉出现。

下图是数据包交错的媒体文件的示意图：



数据流之间在时间上，理论上是相互独立的，但是涉及音视频同步时需要相互参考，下面介绍播放器时详细说明。

播放器主要流程解说：

首先播放器工作是容器分析，从容器中读取数据流，容器无论是文件或网络流，都存在着时间和空间的顺序，播放器程序先按照物理顺序依次读取字节流数据，通过数据分析得到容器的具体信息，例如：容器的时长，有多少个数据流，每个数据流的类型，数据流的时长，数据流的时间单位，数据流的包的个数，数据流每个包的位置偏移、包大小，数据流需要编解码器的标示，视频数据流图像宽高、宽高比、帧率等，音频数据流采样频率、采样格式、声道布局等，其他数据流信息等等。这些在容器上的信息，就像贴在衣服上的明信片，标示衣服里面都放着什么东西，但是也会出现明信片标示和实际存放的数据不符，这样在后续的处理当中可能会出现意想不到的问题，这时候就是考验播放器的兼容性、健壮性的时候，如果程序写得越合理，当然就会是越好了。

其次播放器的工作是分包（分流），因为读取有一个物理顺序，所以先读取的是一个个数据包，每个数据包上都有标示，指出这个数据包属于哪个流，这时播放器会将这个包分给对应的流进行处理。

然后播放器的工作是解码，每个数据流对应着一个解码器，每一个数据流都一个缓冲队列（主要用于存放解码后的数据），播放器会按照先后顺序把属于自己数据流的数据包依次压入解码器当中，就像传送带一样，数据包不断的压入解码器进行解码，解码器通过解析解码等工作，将解码出来的数据帧（前面介绍的原始数据）投递给数据流的缓冲队列当中，由于解码器的算法等原因，解码器会缓冲帧数据并且播放时间在几个参考帧之间会出现 pts 时间先后顺序颠倒等问题，所以解码器在放入缓冲队列的时候需要对队列的数据进行管理或是重排序等，当然这个工作也可以拖后到渲染部分。

最后播放器的工作是渲染，是根据硬件适配程序接口将解码后的数据填充到其缓冲区中渲染播放。

音视频同步问题，前面已经讲解了，音频和视频数据流是完全独立的，但是视频播放的时候，画面和声音要保持一致，这个就是根据时间戳和时长进行控制的，数据包按顺序进入缓冲队列后，别分再根据时间戳和时长将数据按照时间的参数填充给渲染，这就完成了音视频同步播放的效果。

渲染双缓冲区问题，这个需要硬件适配层 API 的支持，不使用渲染双缓冲区，在底层画图或是音频播放的时候会出现间歇性中断的问题，导致画面或是声音不连续。提前给硬件适配层一个渲染缓冲区，让底层再完成这个缓冲区渲染的时候及时使用备用的下一个缓冲区数据进行播放。

根据以上一系列的概念，容器对应的是 AVFormatContext，协议对应的是 AVIOContext，数据流对应的是 AVStream，编解码对应的是 AVCodecContext，数据包对应的是 AVPacket，数据帧对应的是 AVFrame，这些对象数据结构如果在前面章节没有做详细介绍，将在后面做详细讲解。

下表是容器和协议层常用数据结构介绍：

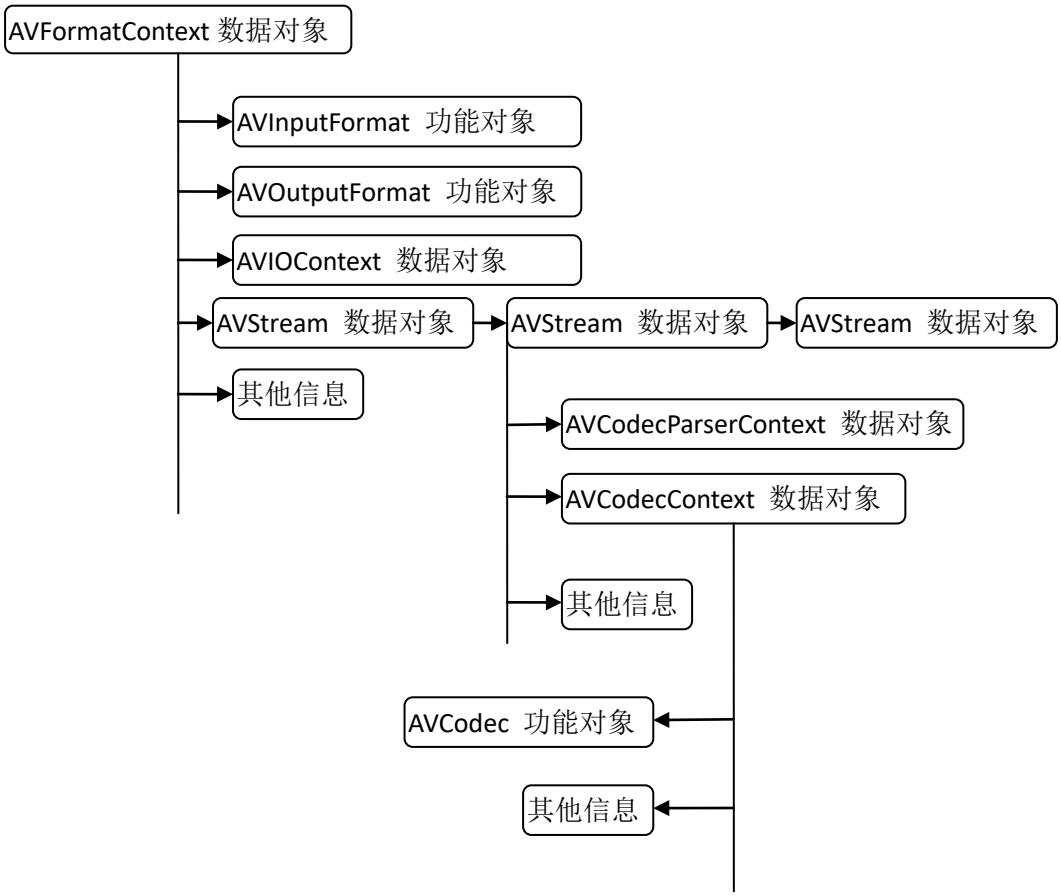
结构体名称	类型	意义
AVFrac	typedef struct AVFrac { int64_t val, num, den;	时间值 val 表示 pts 值

	} AVFrac;	num、den 通过分数的形式表示 pts 的时间单位 （新版本结构，以后修改趋势）
AVCodecTag	typedef struct AVCodecTag { enum AVCodecID id; unsigned int tag; } AVCodecTag;	编解码器对应 ID 结构 id 是 ffmpeg 库使用的内部 ID tag 是标准 ID 值，外部使用

AVDictionary	struct AVDictionaryEntry {
	char *key;
	char *value;
	} AVDictionaryEntry;
	struct AVDictionary {
	int count;
	AVDictionaryEntry *elems;
	};
		数据字典
		count 为字典元素个数
		elems 是字典数据数组
		（多用于表示 mediadata 字典数据）

AVFormatContext	数据对象，动态管理，外部可见	容器层核心类
AVOutputFormat	功能对象，固定分配，外部可见	输出容器
AVInputFormat	功能对象，固定分配，外部可见	输入容器
AVIOContext	数据对象，动态管理，外部可见	协议层核心类
AVStream	数据对象，动态管理，外部可见	容器数据流-音视频数据流
AVProgram	数据对象，动态管理，外部可见	节目-数据流(虽说暂用，但至今)
AVChapter	数据对象，动态管理，外部可见	章-数据流(虽说暂用，但至今)
AVIOInterruptCB	typedef struct AVIOInterruptCB { int (*callback)(void*); void *opaque; } AVIOInterruptCB;	协议层 IO 回调中断

如果从一个文件的结构来看容器，下面准备通过一个图来详细简介其 FFmpeg 组织结构，下面先看图：



AVFormatContext：是最大的数据对象（实为数据结构），这样一个数据对象对应一个实体文件，数据对象是动态管理，输入输出容器共享这个数据对象空间。

AVInputFormat：是容器输入功能对象，用于容器解析操作，其数据结构保存的是执行的函数地址和一些对象常量属性、类型等。

AVOutputFormat：是容器输出功能对象，用于容器封装操作，其数据结构保存的是执行的函数地址和一些对象常量属性、类型等。

AVIOContext：是容器层下协议层的数据对象，用于 IO 操作，其功能对象也包含其中。

AVStream：是容器层内的数据流对象，每一包数据 AVPacket 不会保存在这里，这是保存数据流的信息和数据包的一些分析数据，其中最主要的是包含编解码层对象。

AVCodecContext：是数据流内的数据对象，每一个解码后或是编码前的原始数据帧 AVFrame 不会保存在这里，这里是保存编解码所需的算法数据、参数信息等。

AVCodec：是编解码数据对象内的功能对象，用于编解码操作，其数据结构保存的是执行的函数地址和一些对象常量属性、类型等。

数据包和数据帧的关系：数据包经过解码后得到的是数据帧，数据帧经过编码后得到的是数据包。

编解码器缓冲数据：由于编码算法的因素，一帧数据可以双向（前后两帧）参考，同解码器也需要解码前后帧后，才可以解码出当前帧，这就导致了编解码器会缓冲数据帧或数据包的情况发生。

数据包和数据帧的对应关系：从数据总量角度看一个数据帧对应一个数据包，又因为编解码算法的原因导致编解码器有缓冲数据的情况，同样在编解码时候后推入一帧原始数据或是一包字节流进行编码或解码，可能不会立即输出对应的一包或是一帧数据，但是最后的总量是相同的。

下面只列举了 AVFormatContext 一些常用的、核心成员变量：

成员声明名称	成员声明类型	意义
av_class	const AVClass *	继承类。
iformat	AVInputFormat *	输入格式
oformat	AVOutputFormat *	输出格式
priv_data	void *	容器私有数据，插件开发使用。
pb	AVIOContext *	协议数据对象（协议层）
ctx_flags flags	int	容器标示
nb_streams streams	unsigned int AVStream **	数据流个数 数据流对象数组
nb_programs programs	unsigned int AVProgram **	节目流个数 节目流对象数组
nb_chapters chapters	unsigned int AVChapter **	章节流个数 章节流对象数组
filename	char [1024]	URL 全路径字符串
start_time duration	int64_t	起始时间 时长
bit_rate	int	码率
metadata	AVDictionary *	媒体信息
interrupt_callback	AVIOInterruptCB	IO 层操作回调
packet_buffer packet_buffer_end raw_packet_buffer raw_packet_buffer_end parse_queue parse_queue_end	struct AVPacketList *	数据包链表头和尾
video_codec_id audio_codec_id subtitle_codec_id	AVCodecID	参数设置编解码 ID

下面只列举了 AVStream 一些常用的、核心成员变量：

成员声明名称	成员声明类型	意义
index id	int	数据流-索引值 数据流-ID 值
codec	AVCodecContext *	编解码数据对象
priv_data	void *	数据流私有数据
time_base start_time duration nb_frames	AVRational int64_t	时间单位 起始时间 时长 总帧数
metadata	AVDictionary *	媒体信息
r_frame_rate avg_frame_rate	AVRational	帧率 平均帧率
info	结构体	内部使用为 av_find_stream_info 做分析，保存的是流信息
need_parsing parser	AVStreamParseType AVCodecParserContext	需要 parser 解析器类型 parser 解析器数据对象
pts	typedef struct AVFrac { int64_t val, num, den; } AVFrac;	输出的 pts 值
reference_dts first_dts cur_dts last_IP_pts	int64_t	dts、pts 参考时间参数

1) 数据流和容器与协议（Stream Container and Protocol）

前面已经讲述了数据包、帧、数据流、容器和协议的概念，这部分首要是从实际应用角度出发，把一系列数据串起来，根据做一个完整的流程概念解析的过程。

容器解析流程：

1. URL 解析，根据输入的 URL 全路径解析出协议头，经过检测这样就确定了协议层 IO 对象。例如全路径 pipe:abc，根据:符号取出协议头 pipe 使用对应的 IO 层对象。例子 2 全路径 http://f.youku.com/player/getFlvPath/sid/137143620640976_01/st/mp4/fileid/030020010051BE4B9A579504E9D2A7992D2214-6F84-425B-C2B7-E17792DD216A?K=2430d04903d5334e2411704f&hd=0&ts=504 根据:符号取出协议头 http 使用对应 IO 层对象，那么 D:\lsm\tBook.mp4 这种路径在没有匹配的协议头下，会使用默认协议即 file 协议对象，默认是本地路径处理。

2. 容器适配，根据已经获取出的 IO 层对象缓存预读数据 1Kbyte 到 32Kbyte 之间的数据，通过初选（初选通过 URL 信息、例如：扩展名）然后再大范围的，将缓冲数据送给各个

容器的适配函数进行匹配，通过打分制，第一个 100 分满分的容器将会得到数据，从而成为其解析的容器格式。

3. 容器解头，解头的作用就是获取到容器这件衣服上的所有名片的信息，时长、大小、媒体信息等，也包括数据流的个数，数据流的信息等，其这部分的主要功能是为下面解包和其他动作做准备。

4. 容器解包，包含 Read、Seek、Pause、Play 等动作，这里只是从使用的方面介绍容器解析，最终是通过 read 按照物理顺序依次获取 AVPacket 数据，完成容器格式的解析功能，剥离掉容器的衣服得到一包包的字节流数据。

5. 分包处理，前面已经介绍了数据流的概念，所以依次获取的 AVPacket 可能属于不同的数据流，所以这时候开始写自己的逻辑，怎么处理每条数据流上的数据包。

容器封装流程：

1. 确定容器，找到你使用哪个容器格式进行封装工作，然后确认你的封装容器的信息，填写一些时长等媒体信息数据，然后确认你都有哪几个数据流，每个数据流的容器信息是什么，以及一些提前计算信息和媒体信息等，就好比把容器这件衣服上的名片都准备好一样，这里主要是为下一步做数据的信息准备。

2. 封装头，这里首先是输入一个 URL，根据这个 URL 使用对应的协议层的 IO 对象，根据上一步的准备数据，依照容器格式的规则，将其一些信息提前输出。

3. 容器封包，把不同数据流的填充好的数据包 AVPacket 通过 write 动作，按照物理的依次顺序封装到容器输出位置。

4. 封包完成，这个需要根据具体的容器格式有自己的作用，主要是在所有数据都已经写完的情况下，修改头信息、重写数据、写入文件尾、写入统计数据、写入 SeekTable、写入时长、写入数据大小等等功能。

这里通过了容器封装和容器解析的流程，讲解完一般的通用的处理流程，下面的小节主要是根据上面的流程对应的 FFmpeg 的实现，基本流程不变，在细节上略有出入。

2) 容器封装流程（Muxing）

```
/* 第一步 申请空间 */

AVFormatContext * oc = NULL;                    ///< 容器数据对象
const char * filename = “URL 路径”;    ///< 输出的 URL
AVOutputFormat * fmt = NULL;                    ///< 输出容器功能对象
avformat_alloc_output_context2(&oc, NULL, NULL, filename);  ///< 申请空间


/* 第二步 找到输出容器 */

///< 根据容器的名称找到适合的输出容器
fmt = av_guess_format(“mpeg”, NULL, NULL);

///< 根据 URL 后缀的形式找到适合的输出容器
fmt = av_guess_format(NULL, filename, NULL);


/* 第三步 填充数据流 */

///< 确定使用数据对应的编解码
AVCodec * codec = avcodec_find_encoder(AV_CODEC_ID_H264);

///< 根据 Codec 在指定容器中创建数据流
AVStream * st = avformat_new_stream(oc, codec);

/* 填充编解码和容器的流信息，这里不再重复编码过程的信息数据的添加 */

av_dump_format(oc, 0, filename, 1); ///< 打印容器媒体信息


/* 第四步 写头信息 */

avio_open(&oc->pb, filename, AVIO_FLAG_WRITE);///< 打开协议层 IO
avformat_write_header(oc, NULL); ///< 写入头信息


/* 第四步 封包 */

while(true) ///< 循环依次写入数据包
{
```

```
    AVPacket pkt;
    av_init_packet(&pkt);
    av_new_packet(&pkt, 数据包申请大小));   ///< 申请 AVPacket 空间
    av_interleaved_write_frame(oc, &pkt);   ///< 封包会自动释放 AVPacket
}


/* 第五步 封包完成 */

av_write_trailer(oc); ///< 写尾、回写头、写媒体信息、写 SeekTable 等等
avio_close(oc->pb); ///< 关闭协议层 IO 对象
avformat_free_context(oc);  ///< 释放空间
```

3) 容器解析流程（Demuxing）

```
/* 第一步 打开容器 */

AVFormatContext * oc = NULL;                    ///< 容器数据对象
const char * filename = “URL 路径”;             ///< 输入的 URL
avformat_open_input(&oc, filename, NULL, NULL); ///< 跟 URL 地址打开容器解析

/* 第二步 数据流分析 */

avformat_find_stream_info(oc, NULL);    ///< 主要是知道有多少个数据流和每个数据流的信息等

int st_id = -1;///< 保存找到的流 ID 值

st_i = av_find_best_stream(oc, AVMEDIA_TYPE_VIDEO, -1, -1, NULL, 0); ///< 找到最好的视频流，通过 FFmpeg 的一系列算法，找到最好的默认的音视频流，主要用于播放器等需要程序自行识别选择视频流播放。

st_i = av_find_best_stream(oc, AVMEDIA_TYPE_AUDIO, -1, -1, NULL, 0); ///< 找到最好的音频流，同上。

av_dump_format(oc, 0, filename, 0); ///< 打印容器媒体信息

/* 第三步 分流处理 */

AVPacket pkt;
av_init_packet(&pkt);
pkt.data = NULL;
pkt.size = 0;

while(av_read_frame(oc, &pkt) >= 0) ///< 循环依次读取数据包
{
    if(st_id == pkt.stream_index)
    {
        ///< 读到对应数据的数据包，添加自己的逻辑处理
    }
    av_free_packet(&pkt);
}

/* 第四步 关闭容器 */
```

```
avformat_close_input(&oc); ///< 释放相关资源
```

上面讲解完了一般顺序读取的流程，但是在实际应用当中，会有 Seek 功能，跳到指定的位置进行读取数据包，下面说下 seek 动作函数。

```
int av_seek_frame(AVFormatContext *s, int stream_index, int64_t timestamp,
                  int flags);
```

s 是容器，stream_index 依据数据流 ID 值。
flags 是宏定义的值，可以进行按位操作

```
#define AVSEEK_FLAG_BACKWARD 1 ///< seek backward （表示 Seek 方向）
#define AVSEEK_FLAG_BYTE     2 ///< seeking based on position in bytes
#define AVSEEK_FLAG_ANY      4 ///< seek to any frame, even non-keyframes
#define AVSEEK_FLAG_FRAME    8 ///< seeking based on frame number
```

timestamp 是根据 flags 值精确度，是按时间 Seek 还是按字节 Seek 的值，如果是按字节 Seek 值是字节单位值，如果是按时间这个是 pts 值，时间单位参考容器，若没有则是默认的 AV_TIME_BASE 时间单位。

```
int avformat_seek_file(AVFormatContext *s, int stream_index, int64_t min_ts,
    int64_t ts, int64_t max_ts, int flags);
```

是更精确的范围 Seek 函数，是 av_seek_frame 的底层实现。

```
int av_read_play(AVFormatContext *s);
int av_read_pause(AVFormatContext *s);
```

算是 2 个协议层的接口，主要用于 rtsp 等网络协议和数据流容器操作，开始和暂停功能。

4) 协议与输入和输出（Protocol and IO）

FFmpeg 的 IO 协议层是可以脱离容器单独使用的，可以完成 IO 的基本操作，完全满足其使用者的各种要求，IO 层接口是对底层协议各个模块的抽象、总结出来的，是其 FFmpeg 经典库的一部分。

核心数据结构主要是 AVIOContext 它是 IO 层的句柄，根据句柄进行一系列的操作。IO 层还有另外一个数据结构是 AVIOInterruptCB，从字面上理解的意思就是 IO 层的中断控制块，是 IO 层的中断模块，现主要用于 IO 阻塞中断，未来可以做更复杂的扩展。

FFmpeg 库是纯 C 的代码，凸显了其简单、经典的特色，其 IO 层接口继承了文件系统接口的特点，主要还是方面大多数人使用和理解，节省了学习时间，有经验的可以一看就会，下面也是挑选基本常用接口做一个串联介绍。

FFmpeg 的 IO 协议层在读写的时候都有缓冲区，这个由库完成，默认的读取缓冲区大小是 32Kbyte，他们的作用是增加 IO 操作的工作效率。

1. 找到 IO 协议对象

```
///< 枚举协议对象
const char *avio_enum_protocols(void **opaque, int output);
```

2. 打开 IO 数据流

```
Int ret_error = 0; ///< 检查函数的返回值，如果小于 0 表示失败，返回对应的错误值

///< 检查这个 URL 地址、相应权限是否可用
ret_error = avio_check(“http://www.abc.com/aa”, AVIO_FLAG_READ);

AVIOContext * ph_read = NULL;          ///< 读句柄指针
AVIOContext * ph_write = NULL;         ///< 写句柄指针
char * url = “lsm.mp4”;                ///< URL 路径

ret_error = avio_open2(ph_read,url, AVIO_FLAG_READ,NULL,NULL);   ///< 打开 URL 地址
ret_error = avio_open2(ph_write,url, AVIO_FLAG_WRITE,NULL,NULL);
```

3. 读取数据部分

```
///< 读取一个缓冲区数据
int avio_read(AVIOContext *s, unsigned char *buf, int size);

///< 按 bit 读取，有 l 和 b 的区分，l 是小端字节序（即主机字节序），b 是大端字节序（即为网络字节序）
```

avio_r8、avio_rl16、avio_rb16、avio_rl24、avio_rb24、avio_rl32、avio_rb32、avio_rl64、avio_rb64

4. 写入数据部分

///< 写入字符串，第一个默认 UTF-8，第二个是 UTF-16LE
int **avio_put_str**(AVIOContext *s, const char *str);
int **avio_put_str16le**(AVIOContext *s, const char *str);

///< 写入一个缓冲区数据
void **avio_write**(AVIOContext *s, const unsigned char *buf, int size);

///< 按 bit 写入，也有 l 和 b 的区分。
avio_w8、avio_wl16、avio_wb16、avio_wb24、avio_wl24、avio_wl32、avio_wb32、avio_wl64、avio_wb64

5. 其他操作部分

///< 在读取数据时，向前跳过指定的字节数，底层实现实际会缓存数据
int64_t avio_skip(AVIOContext *s, int64_t offset);

///< Seek 功能
int64_t avio_seek(AVIOContext *s, int64_t offset, int whence);
offset 是偏移
whence 标示符 SEEK_SET 起始位置；SEEK_END 结束位置；SEEK_CUR 当前位置；AVSEEK_SIZE 文件大小。

///< 文件尾检测
int url_feof(AVIOContext *s);

///< 写输出数据 FFmpeg 底层有缓冲区，默认保存在缓冲区中，Flush 动作就是将缓冲区的数据真正输出。
void avio_flush(AVIOContext *s);

///< 暂停功能（协议层）
Int avio_pause(AVIOContext *h, int pause);

///< 按时间 Seek 功能（协议层）
int64_t avio_seek_time(AVIOContext *h, int stream_index,
int64_t timestamp, int flags);

6. 关闭 IO 数据流
int avio_close(AVIOContext *s);

6. 音视频滤器应用开发

FFmpeg 的编解码框架（libavcodec）、容器格式框架（libavformat）、视频格式转换（色彩空间转换）、音频格式转换（音频重采样格式转换）都是经典之作，被无数次应用到各个行业的多媒体开发当中。

将这些拆分为各个子框架的好处，显而易见可以快速开始使用，简约而不简单，精炼而高效，唯一欠缺的是没有一个系统性的整合，即缺少 filter 的思想架构，这就成为了其发展壁垒，当然缺点也就成为了优点，更专业化就成为其发展方向，所以 FFmpeg 的 H264 解码、格式转换等等就是其专业发展的结果，但是随着时间的推移如果没有跟上技术的时代浪潮，优势就会被慢慢削弱，其弊端变的更加尖锐，问题也就接踵而来。

在 FFmpeg 其十多年的发展当中，正因为其缺少这种系统性的架构，FFmpeg 被应用在转码、视频处理、播放器和流媒体等等开发当中处于底层工具的角色，没有得到充分发挥，毕竟 FFmpeg 的追随者众多，是多媒体行业的“泰斗”级开源项目，其开发者也是众多，参与过开发的团队和软硬件公司也众多，随着时间推移其弊端和矛盾增加，也就酿成了那次叛变的主要原因，才有了后来的 libav 项目。

现在的 FFmpeg 和 libav 之间像是同时发展的哥俩，他们之间也都“相互借鉴”，或是直接做 Copy 等工作，所以 FFmpeg 也在不断学习和完善 filter 框架，这是一个长期的过程，用 C 从头写一套 filter 框架还是需要不断完善的，从 0.8 版本以后的 filter 的效率提升就可以看出 filter 的重要性。

其实 libav 是率先做框架优化的，FFmpeg 的 0.8、0.9 版本就是其学习的案例，当时很多人包括现在仍然坚定的支持 FFmpeg 不仅是 Michael Niedermayer 的个人魅力，主要还是 libav 做的还不是特别优秀，随着 FFmpeg 后续版本的升级，FFmpeg 的框架优化已经走到了 libav 的前头，关于 filter 的框架优化慢慢的 2 个项目略显不同，倒是谁优谁略需要时间的验证，最大可能是发展最后都是一样的接口和框架，这个是最好的结果。

关于 filter 框架仍然是不断发展阶段，所以很容易看到其 filter 框架的修改，所以这里作者打算思想讲解为主，接口应用为辅，如果你了解思想，可以在其开源项目中快速学习到应用代码的编写。

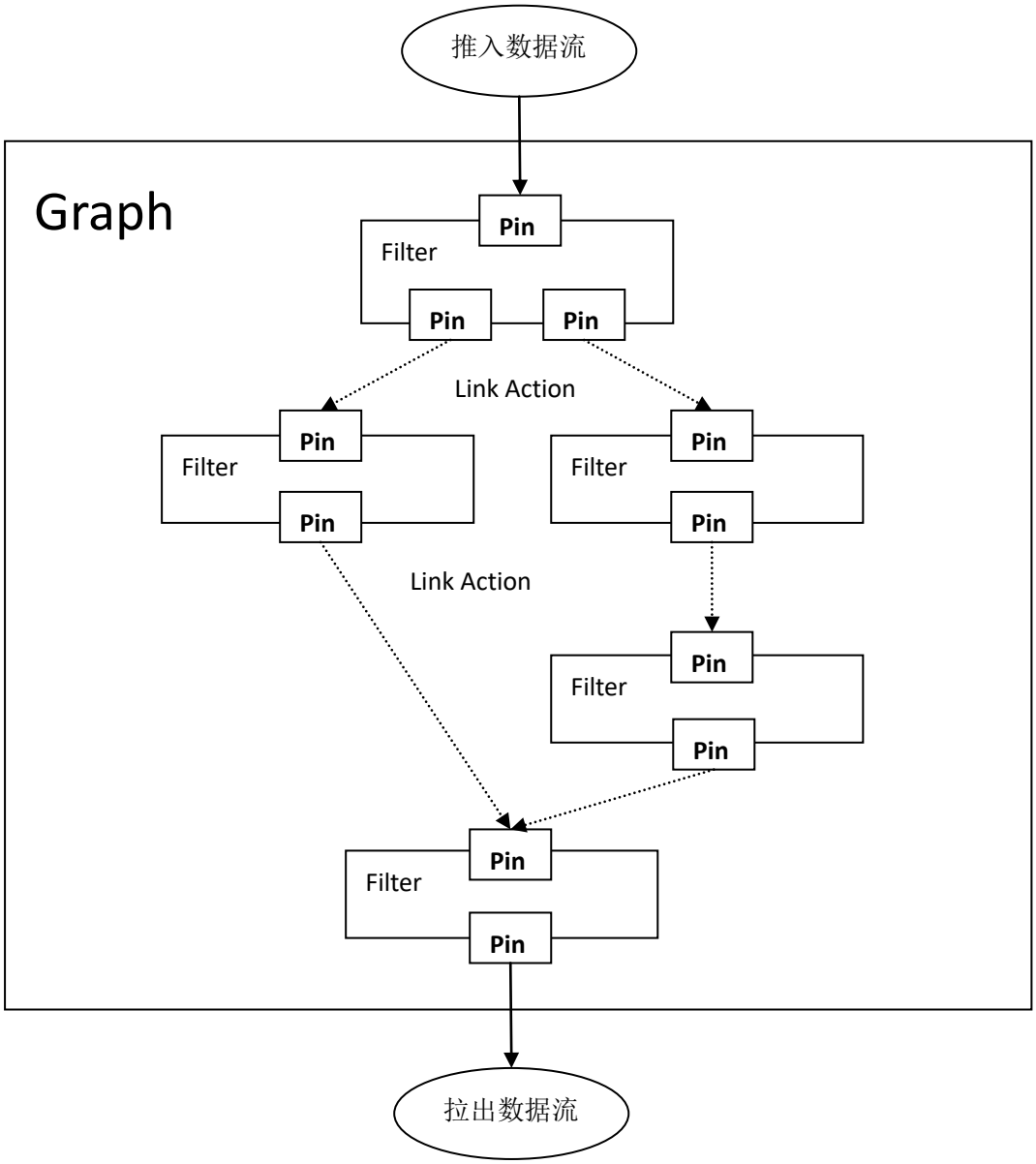
filter 架构思想中第一个概念是 Graph，一般翻译为画布，如果 Graph 看做是桌子的话那 filters 们就是桌子上的“悲剧”。所以先要有 Graph，然后再将 filter 摆在上面，filter 是身上有 pin 接口，pin 的作用是统一数据接口，然后还需要一个 link 的动作，link 的作用是将指定的 2 个 filter 通过其 pin 接口连接起来，这样就形成了一个完整的 filter graph 或是叫 filter link list。

如果只有 filter graph 的存在，它只是一堆参数数据和代码，并不能运行，需要一个动力泵或是动力引擎将整个过程驱动起来，这就像人还缺一颗心脏一样，那人的血液就是 filter graph 的数据流。

驱动引擎、消息泵、驱动器在不同的 filter 框架有不同的实现，这个是和 filter 框架紧密结合的，在 FFmpeg 的 filter graph 的动力源泉是由外部驱动的，因为现阶段 filter 框架实现比较简单，filter 框架也没有实现多线程驱动等复杂功能。

这样 FFmpeg 就把驱动的能力交给了 filter 框架外面来做，通过向 filter graph 的首个 filter 推数据和从 filter graph 的末尾 filter 拉数据从而驱动整个 filter graph 的数据流动。

下图是 filter Graph 总体结构示意图：



下表是 libavfilter 模块常用数据结构简介：

结构体名称	类型	意义
AVFilterGraph	数据对象，动态管理，外部可见	滤器链路图
AVFilter	功能对象，固定分配，外部可见	音视频滤器

AVFilterContext	数据对象，动态管理，外部可见	音视频滤器上下文
AVFilterPad	功能对象，固定分配，外部可见	音视频滤器输入输出接口
AVFilterBuffer	数据对象，动态管理，外部可见	音视频数据帧
AVFilterBufferRefAudioProps	数据结构	音频数据帧，属性信息
AVFilterBufferRefVideoProps	数据结构	视频数据帧，属性信息
AVFilterBufferRef	数据对象，动态管理，外部可见	音视频数据帧引用扩展对象
AVFilterLink	数据对象，动态管理，内部使用	滤器链路（内部 filter 之间）
AVFilterInOut	数据对象，动态管理，外部可见	滤器链路（外部 IO 之间）
AVFilterCommand	数据结构	外部参数传递对象
AVFilterPool	数据结构	音视频数据帧引用池
AVFilterFormats	数据结构	滤器支持类型，视频格式
AVFilterChannelLayouts	数据结构	滤器支持类型，音频格式
AVBufferSinkParams AVABufferSinkParams	内部	音视频参数传递

通过上面介绍 FFmpeg 常用数据结构，不难看出各个数据结构、对象的角色位置，这里基本涵盖了 libavfilter 会用到的所有核心数据结构。

1) 视频滤器应用（Video Filter Graph）

直奔主题，下面开始 FFmpeg 视频滤器使用的流程讲解。

一、 申请资源

```
AVFilterGraph *filter_graph = NULL;
AVFilterInOut *outputs = NULL;
AVFilterInOut *inputs  = NULL;

///< 申请 Graph 画布
filter_graph = avfilter_graph_alloc();
```

二、 构建 Graph

A. 方法一：手动构建

1. 添加 avfilter 到画布

注意：AVFilter 仅是内部的功能对象（即相当于虚基类，保存的是函数代码的地址和属性参数信息），所以要在 AVFilterGraph 数据对象（即相当于实体对象）创建 avfilter 数据对象应该是 AVFilterContext，它是 avfilter 数据对象由 graph 管理。

```
///< 首部 filter 要使用 buffer filter，可以推入数据
AVFilter * header_filter  = avfilter_get_by_name("buffer");

///< 尾部 filter 要使用 buffersink filter，通过内部 sink 实现，可以拉出处理完数据
AVFilter * foot_filter = avfilter_get_by_name("buffersink");

AVFilterContext * header_filter_ctx = NULL;
AVFilterContext * foot_filter_ctx = NULL;

///< 创建首部 filter 对象，添加到 graph 中
ret = avfilter_graph_create_filter(&header_filter_ctx,header_filter,
"in",                                ///< 在 graph 中 filter 链的唯一标示
"video_size=752x424:pix_fmt=0:time_base=1/30000:pixel_aspect=753/752:sws_param=
flags=2:frame_rate=30000/1001",      ///< 创建 filter 参数字符串
NULL, filter_graph);

///< 视频 avfilter sink，尾部 filter 参数
AVBufferSinkParams * buffersink_params = av_buffersink_params_alloc();
```

```
enum AVPixelFormat pix_fmts[] = { AV_PIX_FMT_YUV420P, AV_PIX_FMT_NONE };
buffersink_params->pixel_fmts = pix_fmts;

///< 创建尾部 filter 对象，添加到 graph 中
ret = avfilter_graph_create_filter(&foot_filter_ctx, buffersink,
"out",                    ///< 在 graph 中 filter 链的唯一标示
NULL,
buffersink_params,        ///< 创建 filter 参数字符串
filter_graph);

av_free(buffersink_params);
```

2.Graph 连接

```
///< 申请 Graph 外部输入和输出端接口
inputs  = avfilter_inout_alloc();
outputs = avfilter_inout_alloc();

///< 勾画
outputs->name       = av_strdup("in");
outputs->filter_ctx = header_filter_ctx;
outputs->pad_idx    = 0;
outputs->next       = NULL;

inputs->name       = av_strdup("out");
inputs->filter_ctx = foot_filter_ctx;
inputs->pad_idx    = 0;
inputs->next       = NULL;

///< 构建 graph 链
avfilter_graph_parse(filter_graph,"scale=800:600",&inputs, &outputs, NULL);

///< 检查 Graph 有效性
avfilter_graph_config(filter_graph, NULL);
```

B. 方法二：通过字符串解析

```
///< 通过字符串构建 graph 关系结构
avfilter_graph_parse2(filter_graph,"scale=800:600",&inputs, &outputs);

///< 创建仍需要同上调用
主要是 avfilter_graph_create_filter 系列函数等

///< 检查 Graph 有效性
```

```
avfilter_graph_config(filter_graph, NULL);
```

三、 驱动 Graph

```
///< 从头部压入数据帧
av_buffersrc_add_frame_flags(&header_filter_ctx, frame,
AV_BUFFERSRC_FLAG_KEEP_REF);

///< 从尾部获取处理完的数据帧
while (1) {
int ret = av_buffersink_get_frame(buffersink_ctx, filt_frame);
if (ret < 0)
    break;

///< 使用数据帧引用
av_frame_unref(filt_frame);
}
```

四、 释放资源

```
avfilter_graph_free(&filter_graph);
```

2) 音频滤器应用（Audio Filter Graph）

音频滤器的应用函数和视频滤器的应用函数，及调用过程，基本都是一样的，只是在关键流程上的参数不一样。

区别 1 ：avilter graph 的头尾 filter 不一样。

视频滤器是：buffer 和 buffersink 。
音频滤器是：abuffer 和 abuffersink 。

区别 2：首 filter 输入参数格式。

视频首 filter 是 buffer，其参数包含 video_size、pix_fmt、time_base、pixel_aspect 视频源输入格式相关的信息。
音频首 filter 是 abuffer，其参数包含 time_base、sample_rate、sample_fmt、channel_layout 音频源输入格式相关的信息。

区别 3：尾 filter 输出参数格式。

视频尾 filter 是 buffersink，其参数通过数据结构 AVBufferSinkParams 将视频像素格式传递到 filter 中使用。

音频尾 filter 是 abuffersink，其参数通过 av_opt_set_int_list 函数将 sample_fmts、channel_layouts、sample_rates 这些参数赋值。

7. 音视频原始数据格式转换应用开发

原始音视频数据格式转换一直都是 FFmpeg 的优势，它的优点主要有高精度（使用双精度数据类型）、高效（汇编优化）、高兼容性（支持格式多、标准多）。

其核心主要模块是 libswresample 和 libswscale，现阶段中 libswresample 引用了 libavresample 的一些功能代码，从字面上可以理解 libswresample 中的 resample 是重采样的意思，就是音频原始数据格式转换，从字面上 libswscale 中的 scale 意思是视频原始数据格式转换，其中最有意思的是关于 sw 字符的意思，作者并没有明确表达 sw 的意思，众开发者们一致认为是 SoftWare 的缩写。

先简单介绍一下 libswscale 常用数据结构：

结构体名称	类型	意义
SwsContext	数据对象，动态管理，外部接口	视频原始数据格式转换对象
SwsVector	typedef struct SwsVector { double *coeff; int length; } SwsVector;	数据单平面、单向量平面（buffer 模式，统一 sw 接口工作部分）
SwsFilter	typedef struct SwsFilter { SwsVector *lumH; SwsVector *lumV; SwsVector *chrH; SwsVector *chrV; } SwsFilter;	数据帧、包含所有向量平面（buffer 模式，统一 sw 接口工作部分）

然后再介绍一下 libswresample 常用数据结构：

结构体名称	类型	意义
SwrContext	数据对象，动态管理，外部接口	音频原始数据格式转换对象

1) 视频数据格式转换（Scaling Video）

关于 Swscale 的主要工作是：色彩空间的转换和图像缩放功能。是将源数据的一张图片的数据转换为目的数据的一张图片格式，同时可以做缩放工作，目的数据的内存空间和源数据的内存工作都由使用者管理。

下面是一个将格式 YUV420、宽高 800x600 的一帧数据图像转换为格式 RGB、宽高 640x360 的一帧数据图像过程的具体流程步骤。

第一步：定义源数据和目的数据

```
///< 源数据
AVFrame * pFrame 假定这是解码后的原始数据，也可以是自己的填充

///< 假定填充
pFrame = av_frame_alloc();
pFrame->format  = AV_PIX_FMT_YUV420P;
pFrame->width = 800;
pFrame->height = 600;
pFrame->line 和 pFrame->data 由 AVFrame 结构管理。
内存申请可以使用 av_image_alloc 或是 av_malloc 等。

///< 目的数据
AVPicture dst_pic = {0};
enum AVPixelFormat dst_pix_fmt = AV_PIX_FMT_RGB24;
int dst_w = 640;
int dst_h = 360;
avpicture_alloc(&dst_pic, dst_pix_fmt, dst_w, dst_h);
/* 这里可以使用 av_image_alloc 或 av_malloc 管理内存 */
```

第二步：创建 Scale 对象

```
///< 创建视频格式转换句柄
struct SwsContext *sws_ctx = NULL;
sws_ctx = sws_getContext(       ///< 创建函数
pFrame->width,                  ///< 源图像宽度
pFrame->height,                 ///< 源图像高度
pFrame->format,                 ///< 源图像像素格式
dst_w,                          ///< 目标图像宽度
dst_h,                          ///< 目标图像高度
dst_pix_fmt,                    ///< 目标图像像素格式
```

```
SWS_BILINEAR,                    ///< 标示符
NULL, NULL, NULL);               ///< 其他

/**
 *  标示符说明
 *
 *  SWS_FAST_BILINEAR/SWS_BILINEAR  chroma Bilinear, 快速/高质量,2pix
 *  SWS_BICUBIC                     luma  Bicubic,4pix
 *  SWS_SPLINE                      S-Spline
 *  SWS_BICUBLIN                    chroma-2/luma-2,
 *  SWS_GAUSS                       Gaussian scaler
 *  SWS_SINC                        sinc scaler
 *  SWS_LANCZOS                     lanczos scaler
 *  SWS_AREA                        Area Averaging scaler
 *  SWS_POINT                       Nearest Neighbor / POINT scaler
 *  SWS_X                           实验优化
```

Bilinear （两次线性）
这种方法使用一个原始像素的信息，同时使用原始像素周围相邻的四个像素的信息决定的新像素的颜色。该方法相对简单的线性计算方式完成工作，但也会产生锯齿的效果。

Bicubic （两次立方）
这种方法通过原始像素和它周围 16 个像素的信息计算出新像素的颜色。是在前两种方法基础上的重大改进，原因如下：1.使用了大量的像素信息参加计算 2.使用了较前两钟方法更具先进的计算方法。这种插值方法可以产生高质量的图片，是最经常使用的方法。

S-Spline
这种方法改变了以往大多数的像素颜色的计算方法,像 Bicubic 等插值方法对插值的每一个像素使用了相同的规则(也就是说,无论周围像素是什么样子,都会使用相同的计算规则)。而 S-Spline 分析周围的像素并且依据不同像素的颜色改变计算规则。

```
 */
```

第三步：转换工作

```
///< 转换函数
int ret = sws_scale(sws_ctx, pFrame->data, pFrame->linesize, 0, dst_h, dst_pic.data,
    dst_pic.linesize);
/* 如果成功返回的是目标图像的高度,数据就会存储在 dst_pic 中 */
```

第四步：释放相关资源

```
av_frame_free(&pFrame);
avpicture_free(&dst_pic);
sws_freeContext(sws_ctx);
```

2) 音频数据格式转换（Resampling Audio）

关于 Resampling 的主要工作是：声道布局（声道数）、采样频率、以及采样格式（采样深度）的转换。以字节流的形式输出和输出数据缓冲区，每个缓冲区的大小（即转换时段处理的大小）是由采样频率和算法共同制约的。

下面通过一个样例讲述音频格式转换的过程。

第一步：定义源数据和目的数据

```
///< 源数据
int64_t src_ch_layout = AV_CH_LAYOUT_STEREO;            ///< 声道格式
int src_rate = 48000;                                   ///< 采样频率
enum AVSampleFormat src_sample_fmt = AV_SAMPLE_FMT_DBL; ///< 采样格式
int src_nb_channels = 0;
src_nb_channels = av_get_channel_layout_nb_channels(src_ch_layout);///< 声道数
int src_nb_samples = 1024;      ///< 单个数据包、采样点数
uint8_t **src_data = NULL;      ///< 数据包地址
int src_linesize;               ///< 数据包大小

///< 申请缓冲区
av_samples_alloc_array_and_samples(&src_data, &src_linesize, src_nb_channels,
                                   src_nb_samples, src_sample_fmt, 0);

///< 目的数据
int64_t dst_ch_layout = AV_CH_LAYOUT_SURROUND;              ///< 声道格式
int dst_rate = 44100;                                       ///< 采样频率
enum AVSampleFormat dst_sample_fmt = AV_SAMPLE_FMT_S16;     ///< 采样格式
int dst_nb_channels = 0;
dst_nb_channels = av_get_channel_layout_nb_channels(dst_ch_layout); ///< 声道数

int dst_nb_samples;     ///< 单个数据包，采样点数
dst_nb_samples = av_rescale_rnd(src_nb_samples, dst_rate, src_rate, AV_ROUND_UP);

uint8_t **dst_data = NULL;      ///< 数据包地址
int dst_linesize;               ///< 数据包大小

///< 申请缓冲区
av_samples_alloc_array_and_samples(&dst_data, &dst_linesize, dst_nb_channels,
                                   dst_nb_samples, dst_sample_fmt, 0);
```

第二步：创建 Resample 对象

其中 src_data 就是输入的音频数据的存放地址，可以填充数据也可以是解码后数据，就是对应的数据源信息。

```
///< 创建音频格式转换句柄
struct SwrContext *swr_ctx = NULL;
swr_ctx = swr_alloc();

///< 设置输入格式
av_opt_set_int(swr_ctx, "in_channel_layout",    src_ch_layout, 0);
av_opt_set_int(swr_ctx, "in_sample_rate",       src_rate, 0);
av_opt_set_sample_fmt(swr_ctx, "in_sample_fmt", src_sample_fmt, 0);

///< 设置输出格式
av_opt_set_int(swr_ctx, "out_channel_layout",    dst_ch_layout, 0);
av_opt_set_int(swr_ctx, "out_sample_rate",       dst_rate, 0);
av_opt_set_sample_fmt(swr_ctx, "out_sample_fmt", dst_sample_fmt, 0);

int ret = swr_init(swr_ctx);///< 正常返回值>=0,小于 0 为负数。
```

第三步：转换工作

在进行转换的时候由于采样频率不一致，长时间转换，从字节流到时间上会出现不一致的问题，这时候需要修改字节流数据，来保持时间上的一致性，这个很好理解，可以参考 example 中的例子。

```
///< 转换函数
ret = swr_convert(swr_ctx, dst_data, dst_nb_samples, (const uint8_t **)src_data,
    src_nb_samples);
/* 成功返回值为目标数据，单个数据包，采样点数；失败为负数 */
```

第四步：释放相关资源

```
if (src_data)
    av_freep(&src_data[0]);
av_freep(&src_data);

if (dst_data)
    av_freep(&dst_data[0]);
av_freep(&dst_data);

swr_free(&swr_ctx);
```

8. 解析器和比特流滤器应用开发

解析器的叫法是根据字面意思翻译的，实际按照功能应该叫解码解析器；而比特流滤器也是根据字面意思进行的翻译，按照功能应该叫封装/编码比特流滤器。

这里读者先要回顾一下前面章节讲到的 ffmpeg 转码流程，就可以看到解析器是将容器解析器和解码器关联起来的纽带，而比特流滤器是将编码器和容器封装器关联起来的纽带；它俩都是针对的是数据格式，并没有修改实际的数据内容。

例如 MP4 到 H264 的解析器：

在解封装 MP4 容器格式时，开发者开发容器格式时默认遵循保持原有数据格式的规则，所以根据 ISO-IEC-14496 定义的文档在解封装后，得到的 sps、pps 和数据包格式非字节流序，经过 parse 解析后转为 00 00 00 01 字节流序或是自己定义的格式传给解码器解码。同时 Parse 解析器还充当预处理的过程,例如在直接解析码流的时候这时候需要分析出码流的具体信息，方便框架为解码器准备数据和资源。

例如 H264 到 MP4 的比特流滤器：

就是将编码后输出的字节流序，转换为符合标准的数据格式，存放到容器里面，可以将自己填充的数据，在封装前转换填充到容器中，算是解析器的逆过程。

解析器对应的核心实现是 AVCodecParser，比特流滤器对应的核心实现是 AVBitStream Filter。

根据以上功能行介绍，大家可以分析出，解析器和比特流滤器主要还是在 ffmpeg 转码时起到很大的作用，特别是比特流滤器，后面介绍其在实际开发中的应用。在介绍开发应用前需要重点说明一下,在解码器内部实现有的可以强制自动调用解析器为其进行数据的预处理工作，方便解码。

解析器应用调用：

根据上面的介绍解析器的功能，所以解析器只是在容器解析后得到 AVPactet 后对其中的数据进行格式处理，和再 open 的时候对 AVCodecContext 中的数据进行格式处理。

```
///< 仅需要调用这个函数即可
int av_parser_change(
AVCodecParserContext *s,                    ///< 使用 avctx 中的解析器
AVCodecContext *avctx,                      ///< 解码器 contex
uint8_t **poutbuf, int *poutbuf_size,       ///< 目的数据
const uint8_t *buf, int buf_size,           ///< 源数据
int keyframe);                              ///< 是否是关键帧,容器层数据属性
```

解析器库调用：

这里面的库使用，主要针对在进行解码器插件开发时候需要调用的流程，解码器开发会在后续的章节中详细介绍。

第一步：创建解析器

```
AVCodecParserContext * myparser = av_parser_init(AV_CODEC_ID_H264);
```

第二步：解析处理

```
while(in_len){
len = av_parser_parse2(myparser, AVCodecContext, &data, &size,
                                 in_data, in_len,
                                 pts, dts, pos);
in_data += len;
in_len  -= len;

if(size)
decode_frame(data, size);///< 解码过程
}
```

第三步：释放关闭解析器

```
av_parser_close(myparser);
```

比特流滤器部分：

在介绍比特流滤器开发应用前需要重点说明一下,在一些解码器以及容器封装时会调用到比特流滤器，它们的调用流程和函数和自己调用的流程函数一样，下面主要介绍比特流滤器在实际开发中的应用。

第一步：创建比特流滤器

```
AVBitStreamFilterContext * bsfc = av_bitstream_filter_init("h264_mp4toannexb");
```

第二步：过滤处理

```
///< 过滤函数，返回值是处理的数据大小
int len = av_bitstream_filter_filter(bsfc,
AVCodecContext *avctx,                      ///< 编码器 contex
const char *args,                           ///< 自定义参数
uint8_t **poutbuf, int *poutbuf_size,       ///< 目的数据
```

```
const uint8_t *buf, int buf_size,          ///< 源数据
int keyframe);                             ///< 关键帧
```

第三步：释放关闭比特流滤器

```
av_bitstream_filter_close(bsfc);
```

第四篇 插件开发篇

插件开发属于 FFmpeg 高级编程阶段，进入插件开发其主要难点还是在插件的核心业务和框架的支持。假设你已经了解自己的业务，现在准备基于自己的业务开发一个插件，下面就是主要介绍框架如何支持插件工作的。

开发插件类型几乎涵盖所有功能，唯独没有覆盖到视频像素和音频采样的格式转换，如果想对 scale 和 resample 进行功能性新增，需要修改和编译库，并没有接口提供外部注册。

其插件的核心类型如下：音视频编解码插件、字幕编解码插件、解析器和比特流滤器插件、容器格式输入输出插件、IO 协议的输入输出插件、以及音视频滤器插件。应用开发结合插件开发几乎涵盖了大部分多媒体开发的需求，而且 FFmpeg 是大多数多媒体程序的基础库，也可以将很多开源或自己的外部库引入其中。

其所对应的核心数据结构如下：AVCodec、AVCodecParse、AVBitStreamFilter、AVInputFormat、AVOutputFormat、URLProtocol、AVFilter。插件开发就是将这些定义好的对象注册的库中，供给应用开发调用使用。

1. 编解码器插件开发

开始进入高级开发讲解，这一节主要讲解 AVCodec 插件开发的结构和注意事项。在编解码模块还有 AVHWAccel 硬件加速器插件、AVCodecParser 解析器插件、AVBitStreamFilter 比特流滤器插件。

可以在调用完 avcodec_register_all 注册完全部默认组件后，在调用如下函数注册自己想要的插件。

```
///< 注册编解码器
void avcodec_register(AVCodec *codec);

///< 注册硬件加速器
void av_register_hwaccel(AVHWAccel *hwaccel);

///< 注册解析器
void av_register_codec_parser(AVCodecParser *parser);

///< 注册比特流滤器
void av_register_bitstream_filter(AVBitStreamFilter *bsf);
```

由此可见编码器的注册就是将 AVCodec 对象添加到库中,下面是 AVCodec 中的一个重要成员

CODEC_CAP_DRAW_HORIZ_BAND 解码器使用 draw_horiz_band 回调

CODEC_CAP_DR1 解码器使用 get_buffer 否则数据帧由 avcodec_default_get_buffer 进行分配

CODEC_CAP_TRUNCATED 解码器支持截断

CODEC_CAP_HWACCEL 解码器支持硬件加速

CODEC_CAP_DELAY 表示有延时帧，编解码算法数据帧有前后预测的功能例如 H264，会有 B 帧出现，所以编解码器都会缓冲数据帧。

CODEC_CAP_SMALL_LAST_FRAME 音频编解码可以填充数据样本，压出数据

CODEC_CAP_HWACCEL_VDPAU 支持硬件 VDPAU

CODEC_CAP_SUBFRAMES 总量上不是一帧对应一包的编解码流程,需要内部实现分包分帧，然后进行编解码器处理

CODEC_CAP_EXPERIMENTAL 实验加入版本标示

CODEC_CAP_CHANNEL_CONF 在 context 需要配置声道格式、采样率、采样格式信息

CODEC_CAP_NEG_LINESIZES 负数 linesize

CODEC_CAP_FRAME_THREADS frame 多线程

CODEC_CAP_SLICE_THREADS slice 多线程

CODEC_CAP_AUTO_THREADS 自动多线程

CODEC_CAP_PARAM_CHANGE 支持参数修改

CODEC_CAP_VARIABLE_FRAME_SIZE　音频编码不等数量采样点的输入
CODEC_CAP_INTRA_ONLY　单帧内参考的编解码器
CODEC_CAP_LOSSLESS　无损压缩的编解码器

1) 视频编码器插件（Video Encoder）

这部分将很据自己注册一个 HEVC 编码器的例子进行讲解，这里不会讲解 hevc 具体的一些编码实现，抛开 hevc 内部实现主要开始讲 FFmpeg 编解码插件开发。

第一步：核心定义

```
///< 宏定义
#define X265_ENCODER_NAME                    "x265Encoder"        //h265视频编码器

///< 核心句柄
AVCodec ff_h265Encoder_codec = {
X265_ENCODER_NAME,              ///< 唯一标示名称（可以通过 avcodec_find_encoder_by_name 函数调用，找到此编码器）
X265_ENCODER_NAME" is HEVC/H265Video Experiment Encoder ",      ///< 显示插件名称
AVMEDIA_TYPE_VIDEO,             ///< 编解码器类型为视频
AV_CODEC_ID_H265,               ///< 自定义ID值（可以通过 avcodec_find_encoder 找到）
CODEC_CAP_DELAY | CODEC_CAP_AUTO_THREADS,     ///< 编解码器属性标示符
/*
    CODEC_CAP_DELAY 表示有延时帧，编解码算法数据帧有前后预测的功能例如H264，会有B帧出现，所以编解码器都会缓冲数据帧。
    CODEC_CAP_AUTO_THREADS 表示编解码器插件内部实践多线程。
*/
NULL,                           ///< 支持帧率
pix_fmts_8bit_h265Encoder,      ///< 支持像素格式
NULL,                           ///< 支持采样率
NULL,                           ///< 支持采样格式
NULL,                           ///< 支持声道格式
0,                              ///< 支持解码器支持最低分辨率的最大窗口值
&h265_encoder_class,            ///< priv_class接口-自定义类
NULL,                           ///< 支持profile数组
sizeof(H265EncoderContext),     ///< priv_data_size接口-自定义结构大小
NULL,                           ///< 框架内部使用
NULL,                           ///< init_thread_copy接口-多线程初始化
NULL,                           ///< update_thread_context接口-多线程调用
h265_encoder_defaults,          ///< defaults接口-给AVClass中的option和context设置默认值
NULL,                           ///< init_static_data接口-在注册编解码器时调用
H265_encoder_init,              ///< init接口-打开编解码器时调用
NULL,                           ///< encode接口-编码函数
H265_encoder_frame,             ///< encode2接口-编码函数
NULL,                           ///< decode接口-解码函数
```

```
H265_encoder_close,                    ///< close接口-关闭编解码器时调用
NULL,                                  ///< flush接口-可以自定清空编解码器内部缓存数据
};
```

这里开始依次讲解上面没有详细介绍的了数据结构，而回调函数将在后面的流程中结合调用过程进行依次的讲解。

```
///< 定义编码器支持的像素格式
static const enum PixelFormat pix_fmts_8bit_h265Encoder[] = {
    AV_PIX_FMT_YUV420P,
    AV_PIX_FMT_NONE
};

///< 内部实现核心句柄的定义
typedef struct H265EncoderContext {
    AVClass *       pclass;              ///< 必须有，必须是第一的位置，是编码器的自定义类
    HMEncTop *    pHevcEncTop;          ///< encoder class 是Hevc编码的内部实现
    int nal_hrd;
    char * cfg_file_name;
    char * x265opts;
    char * profile;
    char * level;
} H265EncoderContext;

///< 定义编码器的私有参数 – 参数命令针对典型进行展开讲解
#define OFFSET_H265_ENCODER(x) offsetof(H265EncoderContext, x)
#define VE AV_OPT_FLAG_VIDEO_PARAM | AV_OPT_FLAG_ENCODING_PARAM
static AVOption h265Encoder_options[] = {
    { "cfg_file_name", "cfg_file_name is only filename.(dir is hevc_cfg)",
OFFSET_H265_ENCODER(cfg_file_name),          AV_OPT_TYPE_STRING,      {0 }, 0, 0, VE,
NULL },
    { "x265opts", "x265 private options", OFFSET_H265_ENCODER(x265opts),
AV_OPT_TYPE_STRING, {NULL}, 0, 0, VE},
    { "profile",          "value:baseline、main、main10、high", OFFSET_H265_ENCODER(profile),
AV_OPT_TYPE_STRING, { 0 }, 0, 0, VE},
    { "level",  "value:1.0、2.0、2.1、3.0、3.1、4.0、4.1、5.0、5.1、5.2、6.0、6.1、6.2",
OFFSET_H265_ENCODER(level), AV_OPT_TYPE_STRING, {NULL}, 0, 0, VE},
{ ///< nal-hrd 命令参数
"nal-hrd",       ///< 参数名称
"Default is ABR , Signal HRD information (requires vbv-bufsize; cbr not allowed in .mp4)", ///< 参数说明
OFFSET_H265_ENCODER(nal_hrd),          ///< 参数变量所在结构的偏移
AV_OPT_TYPE_INT,                       ///< 参数类型
{1 },                                  ///< 参数默认值
 -1,                                   ///< 参数最小值边界
```

```
INT_MAX,                                    ///< 参数最大值边界
VE,                                 ///< 参数最大值边界
"nal-hrd"                           ///< 逻辑族，名字必须和主键一样，后面的参数abr、vbr、cbr都是同族的数据值
 },
    { "abr",            "adaptive and dynamic bit rate mode and set bitrate number", 0, AV_OPT_TYPE_CONST, {1},  INT_MIN, INT_MAX, VE, "nal-hrd" },
    { "vbr",            "quality first , dynamic bit rate mode , not recommended to set the bit rate, is qp Anti-calculation", 0, AV_OPT_TYPE_CONST, {2},  INT_MIN, INT_MAX, VE, "nal-hrd" },
    { "cbr",            "bitrate is first , stable bit rate mode and must be set bitrate number", 0, AV_OPT_TYPE_CONST, {3},  INT_MIN, INT_MAX, VE, "nal-hrd" },

};

///< 定义编码器的自定义类
static const AVClass h265_encoder_class = {
    X265_ENCODER_NAME,      ///< 类名称
    av_default_item_name,       ///< 使用系统函数，对AVClass进行默认显示
    h265Encoder_options,        ///< 私有参数命令集合
    LIBAVUTIL_VERSION_INT,  ///< 版本信息
};

///< 默认的变量赋值，主要针对 option 和 context 值进行赋初值
static const AVCodecDefault h265_encoder_defaults[] = {
    { "b",                  "0" },
    { NULL },
};
```

下面将根据视频编码的具体流程，依次介绍调用插件被调用的具体流程以及需要实现的响应功能。

第二步：插件注册

```
///< 插件注册
if(NULL == avcodec_find_encoder_by_name(ff_h265Encoder_codec.name))
    {avcodec_register(&ff_h265Encoder_codec);}
```

在插件注册的同时会触发，插件 init_static_data 接口，在注册插件时进行全局静态数据的初始化工作，由于这里没有注册所有不会触发。

第三步：打开编码器

在调用avcodec_open2进行打开指定codec的时候会自动回调插件的init接口，进行插件的初始化的操作，不同的编解码器、容器等其init都会加入一些自定义的逻辑加在其中，可以理解为解码头、编码头、解析文件头、封装文件头等。

下面H265_encoder_init的主要作用是初始化编码器，需要传入一些参数和值。

```
///< 编码器初始化函数
static av_cold int H265_encoder_init(AVCodecContext *avctx)
{
    // 获取私有句柄
    H265EncoderContext *h5 = (H265EncoderContext *)avctx->priv_data;
    try{
        // 构建内部编码器对象
        h5->pHevcEncTop = new TAppEncTop;

        // 获取当前模块的URL文件夹路径保存在current_dir中
        char current_dir[MAX_PATH+40] = {0};
        if(::GetModuleFileNameA(NULL, current_dir, sizeof(current_dir)) > 0)
        {
            int index_temp = 0;
            for(int i = 0;i<MAX_PATH;i++)
            {
                if(current_dir[i] == '\\' || current_dir[i] == '/')
                {index_temp = i;}
            }
            current_dir[index_temp] = '\0';
        }
        else
        {GetCurrentDirectoryA(sizeof(current_dir),current_dir);}

        // 查找使用默认的预设文件（HEVC现编码器需要）
        std::string cfg_url;
        cfg_url = current_dir;
        cfg_url += "/hevc_cfg/";    // 默认存放在当前模块下hevc_cfg文件夹
        if(h5->cfg_file_name)
        {cfg_url += h5->cfg_file_name;}
        else
        {cfg_url += "encoder_intra_main.cfg";}
        FILE * pcfg = fopen(cfg_url.c_str(),"rb");
        if(pcfg){fclose(pcfg);pcfg=NULL;}
        else
        {
            // 如果没有预设文件，则编码器初始化失败
            av_log(avctx,AV_LOG_ERROR,"hevc_cfg file not exsit %s .\n",cfg_url.c_str());
```

```
            return -1;
        }

        // 检查必要参数的有效性
        if(avctx->time_base.den && avctx->time_base.num &&
            avctx->width>0 && avctx->height>0 && cfg_url.length() > 0)
        {
            // 设置编码器参数
            if(!h5->pHevcEncTop->setParam(cfg_url.c_str(),
                (float) (avctx->time_base.den/avctx->time_base.num),
                avctx->width, avctx->height,
                avctx->bit_rate, h5-> profile, h5-> level, h5->x265opts)
            {
                av_log(avctx,AV_LOG_ERROR,"check param is failed!\n");
                return -1;
            }

            // 创建编码器
            h5->pHevcEncTop->create();

/* 这里省略了一些操作，需要将vps、sps、pps解码需要的必须数据存放到 extradata 当中 */

        }
        else
        {
            av_log(avctx,AV_LOG_ERROR,"check param is failed!\n");
            return -1;
        }
    }
    catch (df::program_options_lite::ParseFailure & e)
    {return -3;}
    catch(...)
    {return -2;}

    av_log(avctx,AV_LOG_INFO,"这是2013年3月最新的HECV HM10标准，一个实验室版本的H265单线程编码器；HEVC仍处于实验室阶段，现商用多数是基于HM8.0或更早HM5.0、HM4.0版本；HM现在仍然处在添加功能参数、结构调整、质量和码流优化阶段。/n");

    return 0;
}
```

第四步：视频编码调用

在调用avcodec_encode_video2进行编码指定codec的时候会自动回调插件的encode2接口，而老版本的avcodec_encode_video函数会对应调用encode接口。

```
/**
 *   编码函数（下面是一个实现的简要版本）
 *
 *   @param  ctx              句柄
 *   @param  pkt              输出码流数据包
 *   @param  frame            输入视频数据帧
 *   @param  got_packet       如果有码流输出则值为1，否则为0
 *
 *   @return                  大于等0表示成功，负数则错误
 */
static int H265_encoder_frame(AVCodecContext *ctx, AVPacket *pkt, const AVFrame *frame,
                              int *got_packet)
{
    // 获取私有句柄
    H265EncoderContext *h5 = (H265EncoderContext *)ctx->priv_data;

    // 有数据输入
    if(frame && frame->data[0] && frame->linesize[0] && frame->height)
    {
        int ret_value = 0;

        // 调用内部实现的编码函数将输出的Nalu码流存储在outputAccessUnits之中
        Int iNumEncoded = 0;
        list<AccessUnit> outputAccessUnits;
        ret_value = h5->pHevcEncTop->encode(frame->data[0],frame->data[1],frame->data[2],frame->
linesize[0],frame->linesize[1],frame->linesize[2],ctx->width,ctx->height,iNumEncoded,outputAccessUnits);

        // 如果有输出码流
        if(iNumEncoded > 0)
        {
            // 下面是编码成功的处理
            if(ret_value >= 0)
            {
                // 首先计算输出码流的大小
                static const Char start_code_prefix[] = {0,0,0,1};
                int all_nalu_size = 0;/* size of annexB unit in bytes */
                list<AccessUnit>::const_iterator iterBitstream = outputAccessUnits.begin();
                for(int i = 0; i< outputAccessUnits.size(); i++)
                {
                    const AccessUnit& au = *(iterBitstream++);
                    for (AccessUnit::const_iterator it = au.begin(); it != au.end(); it++)
```

```
                {
                    const NALUnitEBSP& nalu = **it;
                    if (it == au.begin() || nalu.m_nalUnitType == NAL_UNIT_SPS || nalu.
m_nalUnitType == NAL_UNIT_PPS)
                    {
                    /* From AVC, When any of the following conditions are fulfilled, the
                     * zero_byte syntax element shall be present:
                     *  - the nal_unit_type within the nal_unit() is equal to 7 (sequence
                     *    parameter set) or 8 (picture parameter set),
                     *  - the byte stream NAL unit syntax structure contains the first NAL
                     *    unit of an access unit in decoding order, as specified by subclause
                     *    7.4.1.2.3.
                     */
                        all_nalu_size += sizeof(start_code_prefix);
                    }
                    else
                    {
                        all_nalu_size += sizeof(start_code_prefix)-1;
                    }
                    all_nalu_size += UInt(nalu.m_nalUnitData.str().size());
                }
            }

            // 分配输出码流存放地址的内存，并且赋值
            if ((ret_value = ff_alloc_packet2(ctx, pkt, all_nalu_size)) >= 0)
            {
                int data_offset = 0;
                *got_packet = 1;
                list<AccessUnit>::const_iterator iterBitstream = outputAccessUnits.begin();
                for(int i = 0; i< outputAccessUnits.size(); i++)
                {
                    const AccessUnit& au = *(iterBitstream++);
                    for (AccessUnit::const_iterator it = au.begin(); it != au.end(); it++)
                    {
                        const NALUnitEBSP& nalu = **it;
                        if (it == au.begin() || nalu.m_nalUnitType == NAL_UNIT_SPS ||
 nalu.m_nalUnitType == NAL_UNIT_PPS)
                        {
                memcpy(pkt->data + data_offset,start_code_prefix,sizeof(start_code_prefix));
                            data_offset += sizeof(start_code_prefix);
                        }
                        else
                        {
            memcpy(pkt->data + data_offset,&start_code_prefix[1],sizeof(start_code_prefix)-1);
```

```
                                    data_offset += sizeof(start_code_prefix)-1;
                                }

                                if(!pkt->flags && (nalu.m_nalUnitType == NAL_UNIT_SPS || na
lu.m_nalUnitType == NAL_UNIT_PPS || nalu.m_nalUnitType == NAL_UNIT_CODED_SLICE_IDR))
                                {pkt->flags = 1;}

                                int tdatalen = UInt(nalu.m_nalUnitData.str().size());
                                for(int i = 0; i<tdatalen ;i++)
                                {
                                    *(pkt->data + data_offset) = (nalu.m_nalUnitData.str())[i];
                                    data_offset++;
                                }
                            }
                        }
                    }
                }
                outputAccessUnits.clear();
            }
            return ret_value;
        }

        return 0;
}
```

第五步：关闭编码器

在调用avcodec_close进行关闭编码的时候会自动回调插件的close接口，去释放插件所申请的相关资源。

```
///< 关闭编码器函数处理
static av_cold int H265_encoder_close(AVCodecContext *avctx)
{

    // 获取私有句柄
    H265EncoderContext *h5 = (H265EncoderContext *)avctx->priv_data;

    // 关闭解码器  释放相关资源
    if(h5->pHevcEncTop)
    {
        h5->pHevcEncTop->destroy();
```

```
            delete h5->pHevcEncTop;
            h5->pHevcEncTop=NULL;
        }

        return 0;
}
```

2) 音频编码器插件（Audio Encoder）

这部分将很据自己注册一个 Faac 音频编码器的例子进行讲解，这里不会讲解 Faac 具体的一些编码实现，可以看到怎么调用 Faac 编码库，因为 Faac 涉及版权问题，所以这部分讲解只起到教程的意义。

第一步：核心定义

```
///< 宏定义
#define LIBFAAC_ENCODER_NAME    "libfaac"   //faac音频编码器
#define FAAC_DELAY_SAMPLES      1024        //libfaac 1024个 采样点的编码延时

///< 核心句柄
AVCodec ff_libfaac_encoder = {
LIBFAAC_ENCODER_NAME,                       ///< 唯一标示名称（可以通过
avcodec_find_encoder_by_name 函数调用，找到此编码器）
LIBFAAC_ENCODER_NAME" AAC (Advanced Audio Codec)",    ///< 显示插件名称
AVMEDIA_TYPE_AUDIO,         ///< 编解码器类型为音频
CODEC_ID_AAC,               ///< 自定义ID值（可以通过 avcodec_find_encoder 找到）
CODEC_CAP_SMALL_LAST_FRAME | CODEC_CAP_DELAY,    ///< 编解码器属性标示符
/*
    CODEC_CAP_SMALL_LAST_FRAME 音频编解码可以填充数据样本，压出数据。
    CODEC_CAP_DELAY 表示有延时帧，编解码算法数据帧有前后预测的功能例如H264，会有B帧出现，所以编解码器都会缓冲数据帧。
*/
NULL,                       ///< 支持帧率
0,                          ///< 支持像素格式
NULL,                       ///< 支持采样率
enum_ff_libfaac_encoder_fmts,   ///< 支持采样格式
faac_channel_layouts,       ///< 支持声道格式
0,                          ///< 支持解码器支持最低分辨率的最大窗口值
NULL,                       ///< priv_class接口-自定义类
ff_libfaac_profiles,        ///< 支持profile数组
sizeof(FaacAudioContext),   ///< priv_data_size接口-自定义结构大小
NULL,                       ///< 框架内部使用
NULL,                       ///< init_thread_copy接口-多线程初始化
NULL,                       ///< update_thread_context接口-多线程调用
NULL,                       ///< defaults接口-给AVClass中的option和context设置默认值
NULL,                       ///< init_static_data接口-在注册编解码器时调用
Faac_encode_init,           ///< init接口-打开编解码器时调用
NULL,                       ///< encode接口-编码函数
```

```
Faac_encode_frame,                  ///< encode2接口-编码函数
NULL,                               ///< decode接口-解码函数
Faac_encode_close,                  ///< close接口-关闭编解码器时调用
NULL,                               ///< flush接口-可以自定清空编解码器内部缓存数据
};

///< 定义编码器支持的采样格式
const enum AVSampleFormat enum_ff_libfaac_encoder_fmts[] = { AV_SAMPLE_FMT_S16,
                                                             AV_SAMPLE_FMT_NONE };

///< 定义编码器支持的声道格式
static const uint64_t faac_channel_layouts[] = {
    AV_CH_LAYOUT_MONO,
    AV_CH_LAYOUT_STEREO,
    AV_CH_LAYOUT_SURROUND,
    AV_CH_LAYOUT_4POINT0,
    AV_CH_LAYOUT_5POINT0_BACK,
    AV_CH_LAYOUT_5POINT1_BACK,
    0
};

///< 定义编码器支持的 Profile
static const AVProfile ff_libfaac_profiles[] = {
    { FF_PROFILE_AAC_MAIN, "Main" },
    { FF_PROFILE_AAC_LOW,  "LC"   },
    { FF_PROFILE_AAC_SSR,  "SSR"  },
    { FF_PROFILE_AAC_LTP,  "LTP"  },
    { FF_PROFILE_UNKNOWN },
};

///< 内部实现核心句柄的定义
typedef struct FaacAudioContext {
// 注意这里第一位置对象，并没有定义AVClass，因为AVCodec对象注册时候也没有AVClass
    faacEncHandle faac_handle;
    AudioFrameQueue afq;
} FaacAudioContext;
```

第二步：插件注册

```
///< 插件注册
if(NULL == avcodec_find_encoder_by_name(ff_libfaac_encoder.name))
    {avcodec_register(&ff_libfaac_encoder);}
```

在插件注册的同时会触发，插件 init_static_data 接口，在注册插件时进行全局静态数据的初始化工作，由于这里没有注册所有不会触发。

第三步：打开编码器

在调用avcodec_open2进行打开指定codec的时候会自动回调插件的init接口，进行插件的初始化的操作，不同的编解码器、容器等其init都会加入一些自定义的逻辑加在其中，可以理解为解码头、编码头、解析文件头、封装文件头等。

下面Faac_encode_init的主要作用是初始化编码器，需要传入一些参数和值。

```
///< 编码器初始化函数
static av_cold int Faac_encode_init(AVCodecContext *avctx)
{
    // 获取私有句柄
    FaacAudioContext *s = (FaacAudioContext *)avctx->priv_data;
    faacEncConfigurationPtr faac_cfg;
    unsigned long samples_input, max_bytes_output;
    int ret;

    // 检查声道数
    if (avctx->channels < 1 || avctx->channels > 6) {
        av_log(avctx, AV_LOG_ERROR, "encoding %d channel(s) is not allowed\n", avctx->channels);
        ret = AVERROR(EINVAL);
        goto error;
    }

    // 打开faac句柄
    s->faac_handle = faacEncOpen(avctx->sample_rate,
                                 avctx->channels,
                                 &samples_input, &max_bytes_output);
    if (!s->faac_handle) {
        av_log(avctx, AV_LOG_ERROR, "error in faacEncOpen()\n");
        ret = AVERROR_UNKNOWN;
        goto error;
    }

    // 检查Faac版本
    faac_cfg = faacEncGetCurrentConfiguration(s->faac_handle);
    if (faac_cfg->version != FAAC_CFG_VERSION) {
        av_log(avctx, AV_LOG_ERROR, "wrong libfaac version (compiled for: %d, using %d)\n",
FAAC_CFG_VERSION, faac_cfg->version);
        ret = AVERROR(EINVAL);
```

```
            goto error;
        }

        //设置编码参数
        switch(avctx->profile) {
            case FF_PROFILE_AAC_MAIN:
                faac_cfg->aacObjectType = MAIN;
                break;
            case FF_PROFILE_UNKNOWN:
            case FF_PROFILE_AAC_LOW:
                faac_cfg->aacObjectType = LOW;
                break;
            case FF_PROFILE_AAC_SSR:
                faac_cfg->aacObjectType = SSR;
                break;
            case FF_PROFILE_AAC_LTP:
                faac_cfg->aacObjectType = LTP;
                break;
            default:
                av_log(avctx, AV_LOG_ERROR, "invalid AAC profile\n");
                ret = AVERROR(EINVAL);
                goto error;
        }
        faac_cfg->mpegVersion = MPEG4;
        faac_cfg->useTns = 0;
        faac_cfg->allowMidside = 1;
        faac_cfg->bitRate = avctx->bit_rate / avctx->channels;
        faac_cfg->bandWidth = avctx->cutoff;
        if(avctx->flags & CODEC_FLAG_QSCALE) {
            faac_cfg->bitRate = 0;
            faac_cfg->quantqual = avctx->global_quality / FF_QP2LAMBDA;
        }
        faac_cfg->outputFormat = 1;
        faac_cfg->inputFormat = FAAC_INPUT_16BIT;
        if (avctx->channels > 2)
            memcpy(faac_cfg->channel_map, channel_maps[avctx->channels-3],
                   avctx->channels * sizeof(int));
        avctx->frame_size = samples_input / avctx->channels;

#if FF_API_OLD_ENCODE_AUDIO
        avctx->coded_frame= avcodec_alloc_frame();
        if (!avctx->coded_frame) {
            ret = AVERROR(ENOMEM);
            goto error;
```

```
    }
#endif

    /* Set decoder specific info */
    avctx->extradata_size = 0;
    if (avctx->flags & CODEC_FLAG_GLOBAL_HEADER) {

        unsigned char *buffer = NULL;
        unsigned long decoder_specific_info_size;

        if (!faacEncGetDecoderSpecificInfo(s->faac_handle, &buffer,
                                           &decoder_specific_info_size)) {
            avctx->extradata = (uint8_t *)av_malloc(decoder_specific_info_size +
FF_INPUT_BUFFER_PADDING_SIZE);
            if (!avctx->extradata) {
                ret = AVERROR(ENOMEM);
                goto error;
            }
            avctx->extradata_size = decoder_specific_info_size;
            memcpy(avctx->extradata, buffer, avctx->extradata_size);
            faac_cfg->outputFormat = 0;
        }
#undef free
        free(buffer);
#define free please_use_av_free
    }

    // 设置编码参数
    if (!faacEncSetConfiguration(s->faac_handle, faac_cfg)) {
        av_log(avctx, AV_LOG_ERROR, "libfaac doesn't support this output format!\n");
        ret = AVERROR(EINVAL);
        goto error;
    }

    avctx->delay = FAAC_DELAY_SAMPLES;
    ff_af_queue_init(avctx, &s->afq);

    return 0;
error:
    Faac_encode_close(avctx);
    return ret;
}
```

第四步：音频编码处理

在调用avcodec_decode_audio4进行编码指定codec的时候会自动回调插件的encode2接口，而老版本的avcodec_decode_audio3函数会对应调用encode接口。

```
/**
 *   编码函数
 *
 *   @param  ctx            句柄
 *   @param  pkt            输出码流数据包
 *   @param  frame          输入视频数据帧
 *   @param  got_packet     如果有码流输出则值为1，否则为0
 *
 *   @return                大于等0表示成功，负数则错误
 */
static int Faac_encode_frame(AVCodecContext *avctx, AVPacket *avpkt,
                             const AVFrame *frame, int *got_packet_ptr)
{
    // 获取私有句柄
    FaacAudioContext *s = (FaacAudioContext *)avctx->priv_data;
    int bytes_written, ret;
    int num_samples    = frame ? frame->nb_samples : 0;
    void *samples      = frame ? frame->data[0]    : NULL;

    // 分配数据包缓冲区
    if ((ret = ff_alloc_packet2(avctx, avpkt, (7 + 768) * avctx->channels))) {
        av_log(avctx, AV_LOG_ERROR, "Error getting output packet\n");
        return ret;
    }

    // Faac编码
    bytes_written = faacEncEncode(s->faac_handle, (int32_t *)samples,
                                  num_samples * avctx->channels,
                                  avpkt->data, avpkt->size);
    if (bytes_written < 0) {
        av_log(avctx, AV_LOG_ERROR, "faacEncEncode() error\n");
        return bytes_written;
    }

    // 将当前数据帧加入到队列中保存
    if (frame) {
        if ((ret = ff_af_queue_add(&s->afq, frame) < 0))
            return ret;
    }
```

```
    if (!bytes_written)
        return 0;

    // 获取下一帧的pts和时长
    ff_af_queue_remove(&s->afq, avctx->frame_size, &avpkt->pts,
                       &avpkt->duration);

    avpkt->size = bytes_written;
    *got_packet_ptr = 1;
    return 0;
}
```

第五步：关闭编码器

在调用avcodec_close进行关闭编码的时候会自动回调插件的close接口，去释放插件所申请的相关资源。

```
///< 关闭编码器函数处理
static av_cold int Faac_encode_close(AVCodecContext *avctx)
{
    // 获取私有句柄
    FaacAudioContext *s = (FaacAudioContext *)avctx->priv_data;

    // 释放资源
#if FF_API_OLD_ENCODE_AUDIO
    av_freep(&avctx->coded_frame);
#endif
    av_freep(&avctx->extradata);
    ff_af_queue_close(&s->afq);

    // 关闭Faac编码器
    if (s->faac_handle)
        faacEncClose(s->faac_handle);

    return 0;
}
```

3) 视频解码器插件（Video Decoder）

关于视频解码的需求我们可能会替换或是编写 h264、wmv、vp9、hevc 等，编写他们的插件难度各不相同，我们挑选 H265 解码来进行案例讲解。

而编写 H264 解码插件是最难的，主要因为 H264 解码是 ffmpeg 库内部开发的，由于是自家继承的原因，它和框架有很多高耦合的东西，如果要完全替换掉需要知道各种各样杂七杂八的知识，不然说不定在某种组合使用下，就会出现意想不到的问题。

第一步：核心定义

```
///< 宏定义
#define H265_DECODER_NAME    "h265"      //faac音频编码器

///< 核心句柄
AVCodec ff_h265_decoder = {
H265_DECODER_NAME,                      ///< 唯一标示名称（可以通过avcodec_find_encoder_by_name 函数调用，找到此编码器）
H265_DECODER_NAME" HEVC/H265 base on HM10.0 Video Decoder（Experiment Codec）",   ///< 显示插件名称
AVMEDIA_TYPE_VIDEO,             ///< 编解码器类型为视频
AV_CODEC_ID_H265,                       ///< 自定义ID值（可以通过 avcodec_find_encoder 找到）
CODEC_CAP_DR1 |CODEC_CAP_DELAY,     ///< 编解码器属性标示符
/*
    CODEC_CAP_DR1 码器使用 get_buffer 进行数据管理，否则数据帧由 avcodec_default_get_buffer 进行分配。
    CODEC_CAP_DELAY 表示有延时帧，编解码算法数据帧有前后预测的功能例如H264，会有B帧出现，所以编解码器都会缓冲数据帧。
*/
NULL,                               ///< 支持帧率
0,                                  ///< 支持像素格式
NULL,                               ///< 支持采样率
NULL,                               ///< 支持采样格式
NULL,                               ///< 支持声道格式
0,                                  ///< 支持解码器支持最低分辨率的最大窗口值
&ff_h265Decoder_class,              ///< priv_class接口-自定义类
NULL,                               ///< 支持profile数组
sizeof(h265DecoderContext),         ///< priv_data_size接口-自定义结构大小
NULL,                               ///< 框架内部使用
NULL,                               ///< init_thread_copy接口-多线程初始化
NULL,                               ///< update_thread_context接口-多线程调用
```

```
NULL,                           ///<  defaults接口-给AVClass中的option和context设置默认值
NULL,                           ///<  init_static_data接口-在注册编解码器时调用
ff_h265_decoder_init,           ///< init接口-打开编解码器时调用
NULL,                           ///<  encode接口-编码函数
NULL,                           ///<  encode2接口-编码函数
ff_h265_decoder_frame,          ///< decode接口-解码函数
ff_h265_decoder_close,          ///< close接口-关闭编解码器时调用
NULL,                           ///<  flush接口-可以自定清空编解码器内部缓存数据
};

///<  内部实现核心句柄的定义
typedef struct h265DecoderContext {
    AVClass * pclass;           ///< FFmpeg系统默认类对象
    AVFrame picture;
    uint64_t coded_frame_number;
    H265DecTop * pHevcDecTop;
    int nal_length_size;        ///< Number of bytes used for nal length (1, 2 or 4)
}h265DecoderContext;

///<  定义参数命令
#define OFFSET_H265_DECODER(x) offsetof(h265DecoderContext, x)
static const AVOption ff_h265_decoder_options[] = {\
    {\
    "nal_length_size",                  ///<  命令名称
    "nal_length_size helper",           ///<  显示帮助
    OFFSET_H265_DECODER(nal_length_size),   ///<  参数对象在内部句柄的内存偏移
     FF_OPT_TYPE_INT,                   ///<  参数类型
    {0},                                ///<  参数默认值
    0,                                  ///<  参数最小值
    4,                                  ///<  参数最大值
    0},
    {NULL}
};

///<  定义h264Decoder类对象
static const AVClass ff_h265Decoder_class = {
    H265_DECODER_NAME,                  ///<  类名称
    av_default_item_name,               ///<  默认处理函数
    ff_h265_decoder_options,            ///<  参数集
    LIBAVUTIL_VERSION_INT,              ///<  版本
};
```

第二步：插件注册

```
///< 插件注册
if(NULL == avcodec_find_encoder_by_name(ff_h265_decoder.name))
    {avcodec_register(&ff_h265_decoder);}
```

在插件注册的同时会触发，插件 init_static_data 接口，在注册插件时进行全局静态数据的初始化工作，由于这里没有注册所有不会触发。

第三步：打开解码器

在调用avcodec_open2进行打开指定codec的时候会自动回调插件的init接口，进行插件的初始化的操作，不同的编解码器、容器等其init都会加入一些自定义的逻辑加在其中，可以理解为解码头、编码头、解析文件头、封装文件头等。

下面ff_h265_decoder_init的主要作用是初始化解码器，需要传入一些参数和值，这里也可以调用对应的AVParase。

```
///< 解码器初始化函数
static av_cold int ff_h265_decoder_init(AVCodecContext *avctx)
{
    // 获取私有句柄
    h265DecoderContext * h5 = (h265DecoderContext *)avctx->priv_data;
    h5->pHevcDecTop = NULL;
    int ret = 0;

    // 初始化AVFrame
    avcodec_get_frame_defaults(&h5->picture);

    try
    {
        // 创建编码器对象
        h5->pHevcDecTop = new TAppDecTop;

        // 设置解码器参数，这里可以将vps、sps、pps先行传入
        if(!h5->pHevcDecTop->parseCfg(NULL, NULL ))
        {
            h5->pHevcDecTop->destroy();
            delete h5->pHevcDecTop;
            h5->pHevcDecTop = NULL;
            return -3;
        }

        // 创建解码器
        h5->pHevcDecTop->create();
```

```
        // 设置解码帧数据
        avctx->coded_frame = &h5->picture;
        h5->coded_frame_number = 0;
        avctx->pix_fmt = AV_PIX_FMT_YUV420P;    ///< 实际开发不应该写死
/*
 *      一般这里可以做vps、pps、sps预分析工作，来进行资源的预申请。
 */
    }
    catch(...)
    {h5->pHevcDecTop = NULL;return -2;}
    return ret;
}
```

第四步：解码处理

在调用avcodec_decode_video2进行解码指定codec的时候会自动回调插件的decode接口实现解码功能。

```
/**
 *  解码函数
 *
 *  @param  avctx           句柄
 *  @param  data            输出视频AVFrame数据帧
 *  @param  data_size       是否有数据帧输出，如果有值为sizeof(AVFrame)大小
 *  @param  avpkt           输入码流的数据包
 *
 *  @return                 大于等0表示成功，负数则错误
 */
static int ff_h265_decoder_frame(AVCodecContext *avctx, void *data,int *data_size, AVPacket *avpkt)
{
    // 获取私有句柄
    h265DecoderContext * h5 = (h265DecoderContext *)avctx->priv_data;
    if(h5->pHevcDecTop == NULL)
    {return -1;}

    // 输出AVFrame是由avctx->coded_frame分配的
    AVFrame * frame_yuv = (AVFrame *)data;
    int ret = 0;
    try
    {
        // 解码函数，如果无输入数据avpkt->data[0]为空
        ret = h5->pHevcDecTop->decode(avpkt,frame_yuv,data_size);
```

```
            if(ret >=0 )
            {ret = avpkt->size;}
            if(data_size && *data_size)
            {
                avctx->width = frame_yuv->width;
                avctx->height = frame_yuv->height;
                ///<  暂计算解决办法，先阶段解码器无缓冲pts队列功能
                frame_yuv->pts = h5->coded_frame_number;
                frame_yuv->key_frame = 1;
                h5->coded_frame_number++;
            }

            return ret;
        }
        catch(...)
        {return -2;}
        return ret;
}
```

第五步：关闭解码器

在调用avcodec_close进行关闭编码的时候会自动回调插件的close接口，去释放插件所申请的相关资源。

```
///<  关闭解码器函数处理
static av_cold int ff_h265_decoder_close(AVCodecContext *avctx)
{
    //  获取私有句柄
    h265DecoderContext * h5 = (h265DecoderContext *)avctx->priv_data;

    //  释放内部解码器实现
    if(h5->pHevcDecTop)
    {
        h5->pHevcDecTop->destroy();
        delete h5->pHevcDecTop;
        h5->pHevcDecTop = 0;
    }

    //  释放缓冲帧
    if(h5->picture.data[0])
    {av_free(h5->picture.data[0]);h5->picture.data[0] = 0;}

    return 0;
```

```
}
```

4) 音频解码器插件（Audio Decoder）

关于音频解码器插件开发的案例讲解将以 g726 解码器进行讲解，而 g726 解码 codec 属于 ADPCM 类型简单便于理解。

第一步：核心定义

```
///< 宏定义
#define G726_DECODER_NAME   "g726"     //g726音频解码器

///< 核心句柄
AVCodec ff_g726_decoder = {
G726_DECODER_NAME,                      ///< 唯一标示名称（可以通过avcodec_find_encoder_by_name 函数调用，找到此编码器）
G726_DECODER_NAME " ADPCM（Experiment Codec）",  ///< 显示插件名称
AVMEDIA_TYPE_AUDIO,         ///< 编解码器类型为音频
AV_CODEC_ID_ADPCM_G726,     ///< 自定义ID值（可以通过 avcodec_find_encoder 找到）
CODEC_CAP_DR1,              ///< 编解码器属性标示符
/*
    CODEC_CAP_DR1 码器使用 get_buffer 进行数据管理，否则数据帧由 avcodec_default_get_buffer 进行分配。
*/
NULL,                       ///< 支持帧率
0,                          ///< 支持像素格式
NULL,                       ///< 支持采样率
NULL,                       ///< 支持采样格式
NULL,                       ///< 支持声道格式
0,                          ///< 支持解码器支持最低分辨率的最大窗口值
NULL,                       ///< priv_class接口-自定义类
NULL,                       ///< 支持profile数组
sizeof(G726Context),        ///< priv_data_size接口-自定义结构大小
NULL,                       ///< 框架内部使用
NULL,                       ///< init_thread_copy接口-多线程初始化
NULL,                       ///< update_thread_context接口-多线程调用
NULL,                       ///< defaults接口-给AVClass中的option和context设置默认值
NULL,                       ///< init_static_data接口-在注册编解码器时调用
g726_decode_init,           ///< init接口-打开编解码器时调用
NULL,                       ///< encode接口-编码函数
NULL,                       ///< encode2接口-编码函数
g726_decode_frame,          ///< decode接口-解码函数
NULL,                       ///< close接口-关闭编解码器时调用
```

```
g726_decode_flush,                  ///<  flush接口-可以自定清空编解码器内部缓存数据
};

///<  编解码置换表
typedef struct G726Tables {
    const int* quant;               /**< quantization table */
    const int16_t* iquant;          /**< inverse quantization table */
    const int16_t* W;               /**< special table #1 ;-) */
    const uint8_t* F;               /**< special table #2 */
} G726Tables;

///<  内部实现核心句柄的定义
typedef struct G726Context {
    AVClass * pclass;       ///< FFmpeg系统默认类对象
    G726Tables tbls;        /**< static tables needed for computation */

    Float11 sr[2];          /**< prev. reconstructed samples */
    Float11 dq[6];          /**< prev. difference */
    int a[2];               /**< second order predictor coeffs */
    int b[6];               /**< sixth order predictor coeffs */
    int pk[2];              /**< signs of prev. 2 sez + dq */

    int ap;                 /**< scale factor control */
    int yu;                 /**< fast scale factor */
    int yl;                 /**< slow scale factor */
    int dms;                /**< short average magnitude of F[i] */
    int dml;                /**< long average magnitude of F[i] */
    int td;                 /**< tone detect */

    int se;                 /**< estimated signal for the next iteration */
    int sez;                /**< estimated second order prediction */
    int y;                  /**< quantizer scaling factor for the next iteration */
    int code_size;
} G726Context;

///<  下面是算法参数、置换表
static const int quant_tbl16[] = { 260, INT_MAX }; /**< 16kbit/s 2bits per sample */
static const int16_t iquant_tbl16[] = { 116, 365, 365, 116 };
static const int16_t W_tbl16[] = { -22, 439, 439, -22 };
static const uint8_t F_tbl16[] = { 0, 7, 7, 0 };
static const int quant_tbl24[] = {    7, 217, 330, INT_MAX };    /**< 24kbit/s 3bits per sample */
static const int16_t iquant_tbl24[] = { INT16_MIN, 135, 273, 373, 373, 273, 135, INT16_MIN };
static const int16_t W_tbl24[] = { -4,    30, 137, 582, 582, 137,    30, -4 };
static const uint8_t F_tbl24[] = { 0, 1, 2, 7, 7, 2, 1, 0 };
```

```
/**< 32kbit/s 4bits per sample */
static const int quant_tbl32[] = { -125,    79, 177, 245, 299, 348, 399, INT_MAX };
static const int16_t iquant_tbl32[] = { INT16_MIN,      4, 135, 213, 273, 323, 373, 425,
                            425, 373, 323, 273, 213, 135,      4, INT16_MIN };
static const int16_t W_tbl32[] = { -12,    18,    41,    64, 112, 198, 355, 1122,
                    1122, 355, 198, 112,    64,    41,    18, -12};
static const uint8_t F_tbl32[] = { 0, 0, 0, 1, 1, 1, 3, 7, 7, 3, 1, 1, 1, 0, 0, 0 };
/**< 40kbit/s 5bits per sample */
static const int quant_tbl40[] = { -122, -16,    67, 138, 197, 249, 297, 338,
                        377, 412, 444, 474, 501, 527, 552, INT_MAX };
static const int16_t iquant_tbl40[] = { INT16_MIN, -66,    28, 104, 169, 224, 274, 318,
                            358, 395, 429, 459, 488, 514, 539, 566,
                            566, 539, 514, 488, 459, 429, 395, 358,
                            318, 274, 224, 169, 104, 28, -66, INT16_MIN };
static const int16_t W_tbl40[] = {14, 14, 24, 39, 40, 41, 58, 100,
                        141, 179, 219, 280, 358, 440, 529, 696,
                        696, 529, 440, 358, 280, 219, 179, 141,
                        100, 58, 41, 40, 39, 24, 14, 14 };
static const uint8_t F_tbl40[] = { 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 3, 4, 5, 6, 6,
                    6, 6, 5, 4, 3, 2, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0 };
static const G726Tables G726Tables_pool[] = {{ quant_tbl16, iquant_tbl16, W_tbl16, F_tbl16 },
                { quant_tbl24, iquant_tbl24, W_tbl24, F_tbl24 },
                { quant_tbl32, iquant_tbl32, W_tbl32, F_tbl32 },
                { quant_tbl40, iquant_tbl40, W_tbl40, F_tbl40 }};
```

第二步：插件注册

```
///<  插件注册
if(NULL == avcodec_find_encoder_by_name(ff_g726_decoder.name))
    {avcodec_register(&ff_g726_decoder);}
```

在插件注册的同时会触发，插件 init_static_data 接口，在注册插件时进行全局静态数据的初始化工作，由于这里没有注册所有不会触发。

第三步：打开解码器

在调用avcodec_open2进行打开指定codec的时候会自动回调插件的init接口，进行插件的初始化的操作，不同的编解码器、容器等其init都会加入一些自定义的逻辑加在其中，可以理解为解码头、编码头、解析文件头、封装文件头等。

```
///< 解码器初始化函数 - 采样音频、采样格式、声道格式一般是保存在容器层各个格式中，码流默认是adpcm数据
static av_cold int g726_encode_init(AVCodecContext *avctx)
{
    // 获取私有句柄
    G726Context* c = avctx->priv_data;

    // 采样率大于8kHz
    if (avctx->strict_std_compliance > FF_COMPLIANCE_UNOFFICIAL &&
        avctx->sample_rate != 8000) {
        av_log(avctx, AV_LOG_ERROR, "Sample rates other than 8kHz are not "
                "allowed when the compliance level is higher than unofficial. "
                "Resample or reduce the compliance level.\n");
        return AVERROR(EINVAL);
    }
    av_assert0(avctx->sample_rate > 0);

    // 仅支持单声道
    if(avctx->channels != 1){
        av_log(avctx, AV_LOG_ERROR, "Only mono is supported\n");
        return AVERROR(EINVAL);
    }

    // 估测采样位深
    if (avctx->bit_rate)
        c->code_size = (avctx->bit_rate + avctx->sample_rate/2) / avctx->sample_rate;

    c->code_size = av_clip(c->code_size, 2, 5);
    avctx->bit_rate = c->code_size * avctx->sample_rate;
    avctx->bits_per_coded_sample = c->code_size;

    ///< 初始化数据
    g726_reset(c);

    /* select a frame size that will end on a byte boundary and have a size of
       approximately 1024 bytes */
    avctx->frame_size = ((int[]){ 4096, 2736, 2048, 1640 })[c->code_size - 2];

    return 0;
}

///< 重置编解码缓冲池
static av_cold int g726_reset(G726Context *c)
{
```

```
    int i;

    c->tbls = G726Tables_pool[c->code_size - 2];
    for (i=0; i<2; i++) {
        c->sr[i].mant = 1<<5;
        c->pk[i] = 1;
    }
    for (i=0; i<6; i++) {
        c->dq[i].mant = 1<<5;
    }
    c->yu = 544;
    c->yl = 34816;

    c->y = 544;

    return 0;
}
```

第四步：解码处理

在调用avcodec_decode_audio4进行解码指定codec的时候会自动回调插件的decode接口实现解码功能。

```
/**
 *  解码函数
 *
 *  @param  avctx           句柄
 *  @param  data            输出音频AVFrame数据帧
 *  @param  got_frame_ptr   是否有数据帧输出，如果有值为1真值即可
 *  @param  avpkt           输入码流的数据包
 *
 *  @return                 大于等0表示成功，负数则错误
 */
///<  解码器函数
static int g726_decode_frame(AVCodecContext *avctx, void *data,
                             int *got_frame_ptr, AVPacket *avpkt)
{
    // 获取私有句柄
    G726Context *c = avctx->priv_data;
    AVFrame *frame      = data;
    const uint8_t *buf = avpkt->data;
    int buf_size = avpkt->size;
```

```
    int16_t *samples;
    GetBitContext gb;
    int out_samples, ret;

    // 输出采样点数
    out_samples = buf_size * 8 / c->code_size;

    /* get output buffer */
    frame->nb_samples = out_samples;
    if ((ret = ff_get_buffer(avctx, frame, 0)) < 0)
        return ret;
    samples = (int16_t *)frame->data[0];

    init_get_bits(&gb, buf, buf_size * 8);

    // 解码
    while (out_samples--)
        *samples++ = g726_decode(c, get_bits(&gb, c->code_size));

    if (get_bits_left(&gb) > 0)
        av_log(avctx, AV_LOG_ERROR, "Frame invalidly split, missing parser?\n");

    // 有数据输出
    *got_frame_ptr = 1;

    return buf_size;
}

///< g726解码
static int16_t g726_decode(G726Context* c, int I)
{
    int dq, re_signal, pk0, fa1, i, tr, ylint, ylfrac, thr2, al, dq0;
    Float11 f;
    int I_sig= I >> (c->code_size - 1);

    dq = inverse_quant(c, I);

    /* Transition detect */
    ylint = (c->yl >> 15);
    ylfrac = (c->yl >> 10) & 0x1f;
    thr2 = (ylint > 9) ? 0x1f << 10 : (0x20 + ylfrac) << ylint;
    tr= (c->td == 1 && dq > ((3*thr2)>>2));

    if (I_sig)    /* get the sign */
```

```
        dq = -dq;
re_signal = c->se + dq;

/* Update second order predictor coefficient A2 and A1 */
pk0 = (c->sez + dq) ? sgn(c->sez + dq) : 0;
dq0 = dq ? sgn(dq) : 0;
if (tr) {
        c->a[0] = 0;
        c->a[1] = 0;
        for (i=0; i<6; i++)
                c->b[i] = 0;
} else {
        /* This is a bit crazy, but it really is +255 not +256 */
        fa1 = av_clip((-c->a[0]*c->pk[0]*pk0)>>5, -256, 255);

        c->a[1] += 128*pk0*c->pk[1] + fa1 - (c->a[1]>>7);
        c->a[1] = av_clip(c->a[1], -12288, 12288);
        c->a[0] += 64*3*pk0*c->pk[0] - (c->a[0] >> 8);
        c->a[0] = av_clip(c->a[0], -(15360 - c->a[1]), 15360 - c->a[1]);

        for (i=0; i<6; i++)
                c->b[i] += 128*dq0*sgn(-c->dq[i].sign) - (c->b[i]>>8);
}

/* Update Dq and Sr and Pk */
c->pk[1] = c->pk[0];
c->pk[0] = pk0 ? pk0 : 1;
c->sr[1] = c->sr[0];
i2f(re_signal, &c->sr[0]);
for (i=5; i>0; i--)
        c->dq[i] = c->dq[i-1];
i2f(dq, &c->dq[0]);
c->dq[0].sign = I_sig; /* Isn't it crazy ?!?! */

c->td = c->a[1] < -11776;

/* Update Ap */
c->dms += (c->tbls.F[I]<<4) + ((- c->dms) >> 5);
c->dml += (c->tbls.F[I]<<4) + ((- c->dml) >> 7);
if (tr)
        c->ap = 256;
else {
        c->ap += (-c->ap) >> 4;
        if (c->y <= 1535 || c->td || abs((c->dms << 2) - c->dml) >= (c->dml >> 3))
```

```
            c->ap += 0x20;
    }

    /* Update Yu and Yl */
    c->yu = av_clip(c->y + c->tbls.W[I] + ((-c->y)>>5), 544, 5120);
    c->yl += c->yu + ((-c->yl)>>6);

    /* Next iteration for Y */
    al = (c->ap >= 256) ? 1<<6 : c->ap >> 2;
    c->y = (c->yl + (c->yu - (c->yl>>6))*al) >> 6;

    /* Next iteration for SE and SEZ */
    c->se = 0;
    for (i=0; i<6; i++)
        c->se += mult(i2f(c->b[i] >> 2, &f), &c->dq[i]);
    c->sez = c->se >> 1;
    for (i=0; i<2; i++)
        c->se += mult(i2f(c->a[i] >> 2, &f), &c->sr[i]);
    c->se >>= 1;

    return av_clip(re_signal << 2, -0xffff, 0xffff);
}
```

第五步：关闭解码器

在调用avcodec_close进行关闭编码的时候会自动回调插件的close接口，去释放插件所申请的相关资源，由于此处没有实现close接口，所以不会调用响应函数。

而在AVCodec ff_g726_decoder实现了一个flush接口，在调用avcodec_flush_buffers进行清空编解码器缓存数据时调用。

Flush的清空缓存数据的功能，可以理解为被清空缓存数据的编解码器和打开编解码器后初始化参数后的状态是一致的。如果是H264视频解码器，可以这么了解 IDR == sps、pps + flush + I Frame。

```
///< Flush 清空功能
static void g726_decode_flush(AVCodecContext *avctx)
{
    G726Context *c = avctx->priv_data;
    g726_reset(c);
}
```

2. 容器插件开发

关于编解码插件的开发在前面章节已经介绍完毕了,通过实例插件可以了解到其核心业务都是其他库实现，因为其算法的复杂性仅是将内部实现和外部应用进行对接。而对于容器插件的开发，实例将把部分容器业务的实现贴出来。

要学习了解容器仅仅知道标准，拿着标准文档是不够的！最重要的是要了解，容器设计者的设计思想，为什么要这么设计？这个更重要。

如果你要了解一个容器只是手里拿着标准文档去学习，这个方法很笨也浪费时间，学习方法很重要，首先可以先去网上找一下相关的 Blog 文章之类，大概了解流程，而在实际开发的时候遇到以前没有想到的问题时候这才需要标准文档来引路,所以标准文档在开发过程只是一个工具。

我建议在阅读本章节前要先要了解最少 4 到 5 中常用不同类型的容器格式,因为本章节将要介绍 StrongFFplugin 作者自己定义的 avfs 部分容器格式,读者从这里应该主要学习的是设计思想，作者认为介绍 avfs 容器格式标准并无太大意思，通过代码而传播思想，可以让所有开发人员终身受益。

在介绍自定义 avfs 容器格式前，作者简要介绍一下 ts、mp4、flv、avi、asf、rtsp 等主要设计思想和思维区别,在阅读一下文章前,建议读者先去简要了解下对应的格式标准。

AVI（Audio Video Interleaved）容器格式设计要考虑当时计算机行业软硬件的发展，avi 格式是 92 年由 Microsoft 公司推出,它为了解决音视频同步的保存在当时存储介质中，并且可以流畅解析音视频数据包而设计的。格式主要分为文件头、数据块和索引块三个部分，每个数据块都有数据头，也会生成索引表。

MP4（MPEG-4 Part 14）容器的标准版 03 年推出的，是在 MPEG-2（94 年推出）和比较 AVI 格式的一个升级版本。强调低冗余、高密度数据集中，意思是同样的编码后的字节流有意义的、有组织的存放在 MP4 文件格式中生成的物理文件大小要小于其他文件格式。在设计 MP4 时把数据库中的思想引入到媒体文件格式的设计当中。

例如：
```
stts-(decoding) time-to-sample
stsc-sample-to-chunk, partial data-offset information
stsz-sample sizes (framing)
stco-chunk offset, partial data-offset information
```

这些 MP4 的 box 表现出数据库中数据表的性质，而 mdat 是 MP4 容器格式的数据区，可以看做是一锅粥的数据段，有了这些 box 可以快速的索引到你想要的数据包。当然在设计上为了得到低冗余和高效索引的特点，如果遵循标准 MP4 失去支持直播的功能。

ASF（Advanced Streaming Format）是 Microsoft 为 win98 开发的流化的多媒体文件格式，被设计在 Internet 上进行传播的容器格式，可以用于 28.8Kbps 到 3Mbps 之间合适的速率的 VOD 点播和直播。

TS（MPEG Transport Stream）格式最早 95 年提出完整标准规格，主要用于电视广播行业，理论上可以适用于任何流化协议层支持，顾名思义它主要是考虑到网络传输，由于其格式特点有很大的数据冗余，它的可扩展行极强，也有一个本地化版本 MPEG-PS（MPEG Program Stream）容器。所有的流化的容器格式对码率的要求都很高，也是因为这样设计才会支持更为复杂的操作需求。当然它有优势就会凸显它的劣势，它的高扩张、高流化的特点导致其高冗余和非常低效的索引功能。

FLV（Flash Video）是 Adobe 公司设计的，最早可能追溯到 2000 年左右在 Flash Player 4/5 时代提出的。容器格式发展到今天，如果 MP4 和 TS 容器格式是两个极端方向，那么 flv 发展到今天，它就是被公认为的一个中立版本容器格式。只需要简单处理就可以用于直播，只是扩展性和 TS 差很远、冗余度比 TS 略低，支持的较为简单。而用于 VOD 点播或是本地播放的 flv，冗余度要远大于 MP4、检索数据的效率上要高于 TS 很多等特点。

RMVB（RealMedia Variable BitRate）是 RealNetworks 公司设计的，在 2000 年左右跟随 RealPlayer-6/9 时代推出的，RM 是固定码率版本，也可以用于直播流化，它还有很多不同的内部扩展版本。

以上容器介绍都是挑选一些格式公开和有特点特色的容器格式设计思想的进行讲解，也给使用者指明一条如何选择自己需要的容器格式（注意：以上给出的参考时间仅是提出的时间，有些容器仍然在持续更新和完善数据标准中）。

【AVFS 容器格式总体设计】

下面是 AVFS 格式定义的第一版的总体结构示意流程：

```
AVFastSearching Format
字节流总体结构：
--------------------
文件头标示
avfs_format_header_marked[8] = {0x00,0x00,0x00, 'a','v','f','s',version}
--------------------
文件头信息
file header(sytem) info  = struct avfs_system_info
--------------------
数据流包头
Linked list Layer (element is Packet) = struct avfs_list_layer_header
--------------------
数据流包内容 到 帧
Packet0 --> Stream Layer    --> stream type 0 : System
                                -->frame 1:
                                    -->seek_table();
Packet1 --> Stream Layer    --> stream type 0 : System
                                -->frame 0:
                                    -->avfs_get_system_info();
```

```
--> stream type 1 : Video
    -->frame 0:
        --> decode_video();
    -->frame 1:
        --> decode_video();
    -->frame 2:
        --> decode_video();
    -->frame 3:
    -->frame 4:
--> stream type 2 : Audio
    -->frame 0:
        --> decode_audio();
    -->frame 1:
        --> decode_audio();
    -->frame 2:
        --> decode_audio();
    -->frame 3:
        --> decode_audio();
    -->frame 4:
--> stream type 3 : Subtitle
    -->frame 0:
        --> decode_subtitle();
    -->frame 1:
    -->frame 2:
```

Packet2
Packet3
Packet4

注意：
DTS 编解码时序/时间 0,1,2,3,4,5,6,7,8,9,.....MAX
PTS 显示时间戳 单位是千万分之一秒 == 1/10,000,000 second

以上是以 AVFS 容器格式为案例的一个总体结构设计图，不难看出它和其他的容器格式并没有太多区别，因为容器格式发展到今天，以作者的智商也不会有什么本质的改变了。

【AVFS 容器格式详细设计】

下面是 AVFS 格式定义的第一版的具体详细的核心数据结构的定义，基本可以看出其设计思想，容器支持 VOD 点播、支持直播、支持高效 Seek、支持版权保护、支持关键数据加密、支持字幕、支持附带任意数据信息等，扩展性也很强。

具体的容器格式的封装和解析过程将在后面的具体容器插件开发中实现，详细的从封装和解析两个过程讲述容器插件开发流程，其中会涉及协议部分知识，不会的可以参考后面的章节介绍信息。

```
#define AVFS_VERSION 0x01        //版本号

/* format header marked （格式头标示） */
static uint8_t avfs_format_header_marked[8] = {0x00,0x00,0x00,'a','v','f','s',AVFS_VERSION};


//数据加密算法默认 R-Key
const static uint32_t avfs_packet_key = 0xA55AA55A;

/**
 *    System Header
 */
typedef struct avfs_system_info
{
    /**
     *    Seek Table （查询表）
     */
    struct avfs_stream_key_table
    {
        uint64_t pts_1;
        uint64_t key_file_offset_2;
    };
    struct avfs_system_key_table_stream_info
    {
        struct avfs_stream_id stream_id_1;                ///<32bit
        uint32_t stream_packet_number_2;
        std::vector<struct avfs_stream_key_table> stream_packets_3;

        uint64_t pre_stream_key_packet_pts;
    };
    uint16_t avfs_system_key_table_stream_number1;
    std::map<int , struct avfs_system_key_table_stream_info *> * avfs_system_key_table_streams2;

    /**
     *    属性信息
     */
    struct avfs_system_property_info
    {
        uint64_t info1_start_time;
        uint64_t info2_duration;
```

```
        uint64_t info3_framenumber;
        uint64_t info4_filesize;
        uint64_t info5_frame_seektable_file_pos;        ///<  在整个容器中的位置偏移(从文件头开始)
        uint64_t info6_other_data_file_pos;             ///<  在整个容器中的位置偏移(从文件头开始)
    }property_info1;

    /**
     *    属性媒体信息
     */
    std::map<char *,char *>    * property_media_data2;          //uint16_t is number

    /**
     *    流信息
     */
    struct avfs_stream_info
    {
        //stream id
        struct avfs_stream_id stream_id_1;

        //Video
        struct avfs_stream_info_video_param
        {
            uint16_t width_1;                   ///< width
            uint16_t height_2;                  ///< height
            uint32_t fps_numerator_3;           ///< fps numerator
            uint32_t fps_denominator_4;         ///< fps denominator
            uint8_t color_space_5;
        }video_param_2;

        //Audio
        struct avfs_stream_info_audio_param
        {
            uint16_t a_sample_rate_1;           ///< sample rate (100Hz)
            uint8_t a_sample_format_2;          ///< sample format
            uint8_t a_channel_layout_3;         ///< channel number / channel layout
        }audio_param_2;

        //Subtitle
        struct avfs_stream_info_subtitle_param
        {
            uint64_t s_framenumber_1;
        }subtitle_param_2;
```

```
        //General
        struct avfs_stream_info_general_param
        {
            uint32_t bit_rate_1;
            uint16_t extra_data_size_2;
            void * extra_data_3;
            std::map<char *,char *>    * media_data_4;        //uint16_t is number
        }general_param_3;

        /**
         *    其他参数
         */
        uint64_t this_stream_start_pos_offset;      ///<  本流开始位置相对于文件头的位置偏移
        int AVStream_index;                         ///< for AVStream
        int AVStream_id;                            ///< for AVStream
        uint32_t stream_total_packet_number;        ///< current stream total frame number
        uint64_t pre_packet_file_pos;               ///< current stream pre packet position
        uint64_t pre_packet_pts;                    ///< current stream pre packet pts
        uint64_t pre_packet_dts;                    ///< current stream pre packet dts
        uint64_t pre_key_packet_file_pos;           ///< current stream pre key packet position
        uint64_t pre_key_packet_pts;                ///< current stream pre key packet pts
        struct avfs_system_key_table_stream_info * seektable;
    };
    uint16_t info_stream_number3;
    struct avfs_stream_info * info_streams4;

    /**
     *    内部参数
     */
    uint64_t system_info_stream_start_pos_offset;   ///<  头信息中流信息位置相对于文件头的位置偏
移
    uint64_t stream_data_start_pos_offset;              ///<  开始数据位置
    uint64_t pre_packet_file_pos;                       ///< any stream pre packet position
    uint64_t pre_packet_pts;                            ///< any stream pre packet pts
    uint64_t pre_key_packet_file_pos;                   ///< only video stream pre key packet position
    uint64_t pre_key_packet_pts;                        ///< only video stream pre key packet pts
}AVFS_SYSTEM_INFO;

/**
 *    Packet Header
 */
typedef struct avfs_list_layer_header{
```

```
/**
 *    关键帧类型
 *    0-NoKeyFrame,1=KeyFrame
 *    1bit
 */
uint8_t keyframe : 1;

/**
 *    是否加密
 *    0-NoEncryption,1-Encryption
 *    1bit
 */
uint8_t encryp_option : 2;

/**
 *    index packet or pts marked
 *    0-packet_index_or_clock_dts          使用包序号从0累加序号
 *    1-packet_index_or_clock_dts          使用时间戳
 *    1bit
 */
uint8_t packet_index_or_clock_dts_flag : 1;

/**
 *    previous packet offset (from current packet header)
 *    0 - is any previous packet
 *    1 - is the same stream and is key packet
 */
uint8_t pre_packet_offset_flag : 1;

/**
 *    next packet offset (from current packet foot)
 *    0 - is any next packet
 *    1 - is the same stream and is key packet
 */
uint8_t next_packet_offset_flag : 1;

/**
 *    Extended information
 *    6bit
 *    (本地字节顺序、低3位)
 */
uint8_t ext_info : 2;

struct avfs_stream_id stream_id;              ///<32bit    STREAM ID
```

```
    uint32_t current_packet_size;          ///<32bit 当前包的大小（not include packet header）
    uint64_t packet_clock_pts;             ///<64bit  当前包的PTS时间戳
    uint64_t packet_index_or_clock_dts;    ///<64bit   当前包的序号 或是 DTS时间戳
    uint64_t pre_key_packet_pos;           ///<64bit   is the position from current packet
header
    uint64_t next_key_packet_pos;          ///<64bit   is the position from current packet foot

}AVFS_LIST_LAYER_HEADER;

/**
 *  Stream ID
 */
typedef struct avfs_stream_id
{
    uint16_t s_index;
    uint16_t codec_id;                     ///< enum enAVFSStreamCodecID
}AVFS_STREAM_ID;

/**
 *  Codec ID
 */
enum enAVFSStreamCodecID
{
    enAVFSCI_System = 0x0000,              ///<  系统
    enAVFSCI_System_SEEK_TABLE,            ///< SeekTable

    enAVFSCI_VIDEO = 0x1000,               ///<  视频
    enAVFSCI_H264,
    enAVFSCI_H265,
    enAVFSCI_VP8,
    enAVFSCI_VP9,
    enAVFSCI_MPEG4,
    enAVFSCI_H263,
    enAVFSCI_RV10,

    enAVFSCI_MJPEG = 0x1201,
    enAVFSCI_LJPEG,
    enAVFSCI_JPEG2000,
    enAVFSCI_PNG,
    enAVFSCI_BMP,
    enAVFSCI_TIFF,

    enAVFSCI_RAWVIDEO = 0x1301,
```

```
    enAVFSCI_AUDIO = 0x4000,                ///<  音频
    enAVFSCI_MP2,
    enAVFSCI_MP3,
    enAVFSCI_AAC,
    enAVFSCI_AC3,
    enAVFSCI_DTS,
    enAVFSCI_ADPCM_G726,
    enAVFSCI_AMR_NB,
    enAVFSCI_AMR_WB,

    enAVFSCI_SUBTITLE = 0x8000,             ///<  字幕
    enAVFSCI_DVD_SUBTITLE,

    enAVFSCI_OTHERS = 0xF000,               ///<  其他流
    enAVFSCI_DATA_BIN,

    enAVFSCI_None = 0xFFFF,
};

/**
 *    Color Space
 */
enum enAVFSColorSpace
{
    enAVFSCP_YV12 = 1,          ///< 8 bit Y plane followed by 8 bit 2x2 subsampled V and U planes.
    enAVFSCP_NV12,              ///< 8-bit Y plane followed by an interleaved U/V plane with 2x2
subsampling
    enAVFSCP_YV16,              ///< 8 bit Y plane followed by 8 bit 2x1 subsampled V and U planes.
    enAVFSCP_Y41P,              ///< YUV 4:1:1 (Y sample at every pixel, U and V sampled at every
fourth pixel horizontally on each line). A macropixel contains 8 pixels in 3 u_int32s.
    enAVFSCP_YUV444,            ///< planar YUV 4:4:4, 24bpp, (1 Cr & Cb sample per 1x1 Y samples)
    enAVFSCP_AYUV,              ///< This is a 4:4:4 YUV format with 8 bit samples for each
component along with an 8 bit alpha blend value per pixel. Component ordering is A Y U V (as the name
suggests).
    enAVFSCP_RGB24,             ///< packed RGB 8:8:8, 24bpp, RGBRGB...
    enAVFSCP_RGBA,              ///< packed RGBA 8:8:8:8, 32bpp, RGBARGBA...
    enAVFSCP_YUV9,              ///< planar YUV 4:1:0,    9bpp, (1 Cr & Cb sample per 4x4 Y
samples)
    enAVFSCP_GRAY,              ///< Y - 8bpp
    enAVFSCP_None,
};

/**
 *    Sample Format
```

```
 */
enum enAVFSSampleFormat
{
	enAVFSSF_BYTE = 1,		///< signed/unsigned char,8bit
	enAVFSSF_SHORT,			///< signed/unsigned short,16bits
	enAVFSSF_INT,			///< signed/unsigned int,32bits
	enAVFSSF_FLOAT,			///< float,32bits
	enAVFSSF_DOUBLE,		///< double,64bits
	enAVFSSF_None,
};

/**
 *	Channel Layout
 */
enum enAVFSChannelLayout
{
	enAVFSCL_MONO = 1,				///< middle location,1 channel number
	enAVFSCL_STEREO,				///< left right 120/180 degree,2 channel number
	enAVFSCL_5SURROUND_STEREO = 5, ///< centre、left and right front 30 degree、left and right surround 110 degree,5 channel number
	enAVFSCL_6SURROUND_STEREO = 6, ///< 5.1 surround stereo、bass,6 channel number
	enAVFSCL_None = 0xFF,
};

/**
 *	extern data
 */
typedef struct avfs_property_media_extra_data
{
	uint16_t extra_data_number_1;		//有多少个
	uint32_t extra_data_type_2;			//类型
	uint32_t extra_data_size_3;
}AVFS_PM_EXTRA_DATA;

#define AVFS_PM_EXTRA_DATA_TYPE_MEDIAGREATWALL		1	//Media Greet Wall Project
```

1) 容器封装器插件（Muxer）

AVOutputFormat 是容器封装对象（容器输出）的定义，是将已存在的数据包、数据流保存在一个实体的容器格式中，它和容器解析都使用 AVFormatContext 作为自己的数据上下文存储对象。

这里需要再温习一遍 FFmpeg 的容器层数据的结构：

```
AVFormatContext
    AVInputFormat  *iformat;
    AVOutputFormat  *oformat;
    AVStream  **streams
                ----------  AVStream
                            --------AVPacket  1
                            --------AVPacket  2
                            --------AVPacket  …  …
                ----------  AVStream
                            --------AVPacket  1
                            --------AVPacket  2
                            --------AVPacket  …  …
```

第一步：定义插件核心句柄

```
#define  AVFS_FORMAT_NAME                  "avfs"            //avfs文件容器

///< 封装容器插件
AVOutputFormat ff_avfs_muxer = {
AVFS_FORMAT_NAME,              ///< 唯一名称表示符，可以通过avformat_alloc_output_context2进行指定容器封装插件的名称
AVFS_FORMAT_NAME" is av-fast-searching-format.",    ///< 完整信息说明
AVFS_FORMAT_NAME,              ///< 简要类型说明符
AVFS_FORMAT_NAME,              ///< 扩展名，可以在调用avformat_alloc_output_context2的时候，通过URL进行自动适配，例如：” asf,wmv,wma”
AV_CODEC_ID_AAC,               ///< 封装参考的默认音频编解码器ID
AV_CODEC_ID_H264,              ///< 封装参考的默认视频编解码器ID
AV_CODEC_ID_NONE,              ///< 封装参考的默认字幕编解码器ID
AVFMT_VARIABLE_FPS | AVFMT_ALLOW_FLUSH | AVFMT_GLOBALHEADER | AVFMT_RAWPICTURE | AVFMT_SHOW_IDS | AVFMT_NO_BYTE_SEEK | AVFMT_SEEK_TO_PTS,
///< 容器封装插件属性标示符
/**
 *    AVFMT_VARIABLE_FPS          支持VFR动态帧率结构封装操作
 *    AVFMT_ALLOW_FLUSH           支持Flush清空容器缓冲操作
 *    AVFMT_GLOBALHEADER          容器格式支持全局头信息保存，例如可以将sps/pps、adts、huffman table、palette等全局音视频头信息保存在容器层中，非字节流数据中。
 *    AVFMT_RAWPICTURE            支持没有编码的数据存放
 *    AVFMT_SHOW_IDS              使用Stream ID进行标示的唯一管理，支持index id
 *    AVFMT_NO_BYTE_SEEK          不支持字节级索引
 *    AVFMT_SEEK_TO_PTS           基于pts值进行索引

其他关键标示：
AVFMT_NOFILE                  不支持保存文件，仅是内存数据流容器格式
AVFMT_NEEDNUMBER              分片容器必须，支持自动分片输出，根据URL中的’%d’格式化字符串进行自动生成分片输出的URL。
AVFMT_NOTIMESTAMPS            不支持时间戳，多数是字节流格式、ES裸流等。
AVFMT_GENERIC_INDEX           一般表示ES裸流格式容器或是原始数据容器
AVFMT_TS_DISCONT              允许时间戳的不连续性，多数是流混合模式
AVFMT_NODIMENSIONS            容器属性不需要宽、高
AVFMT_NOSTREAMS               不支持AVStream，相当于只用名片无数据
AVFMT_NOBINSEARCH             不支持向后字节级索引
AVFMT_NOGENSEARCH             不支持向后索引
AVFMT_TS_NEGATIVE             允许pts值小于0

 */
ff_avfs_codectags_muxer,                  ///< 支持编码器ID数组
```

```
&ff_avfs_muxer_class,                   ///< AVFS容器封装类
NULL,                                   ///< 框架内部使用
sizeof(AVFS_MUXER_CONTEXT),             ///< 核心私有句柄结构体大小
ff_avfs_muxer_write_header,             ///< write_header接口，封装文件头
ff_avfs_muxer_write_packet,             ///< write_packet接口，封装数据包
ff_avfs_muxer_write_trailer,            ///< write_trailer接口，封装文件尾、回写头、完成格式处理
NULL,           ///< interleave_packet老接口，交错处理
NULL,           ///< query_codec老接口，用于查询默认编解码ID
NULL            ///< get_output_timestamp老接口，用于获取输出时间
};

///< 编解码支持
const AVCodecTag ff_avfs_codec_tags[] = {

    /* video codec */
    { AV_CODEC_ID_H264              , enAVFSCI_H264 },
    { AV_CODEC_ID_H265              , enAVFSCI_H265 },
    { AV_CODEC_ID_VP8               , enAVFSCI_VP8 },
    { AV_CODEC_ID_VP9               , enAVFSCI_VP9 },
    { AV_CODEC_ID_MPEG4                 , enAVFSCI_MPEG4 },
    { AV_CODEC_ID_H263              , enAVFSCI_H263 },
    { AV_CODEC_ID_RV10              , enAVFSCI_RV10 },

    { AV_CODEC_ID_MJPEG                 , enAVFSCI_MJPEG },
    { AV_CODEC_ID_LJPEG             , enAVFSCI_LJPEG },
    { AV_CODEC_ID_JPEG2000          , enAVFSCI_JPEG2000 },
    { AV_CODEC_ID_PNG               , enAVFSCI_PNG },
    { AV_CODEC_ID_BMP               , enAVFSCI_BMP },
    { AV_CODEC_ID_TIFF              , enAVFSCI_TIFF },

    { AV_CODEC_ID_RAWVIDEO              , enAVFSCI_RAWVIDEO },

    /* audio codec */
    { AV_CODEC_ID_MP2               , enAVFSCI_MP2 },
    { AV_CODEC_ID_MP3               , enAVFSCI_MP3 },
    { AV_CODEC_ID_AAC               , enAVFSCI_AAC },
    { AV_CODEC_ID_AC3               , enAVFSCI_AC3 },
    { AV_CODEC_ID_DTS               , enAVFSCI_DTS },
    { AV_CODEC_ID_ADPCM_G726    , enAVFSCI_ADPCM_G726 },
    { AV_CODEC_ID_AMR_NB            , enAVFSCI_AMR_NB },
    { AV_CODEC_ID_AMR_WB            , enAVFSCI_AMR_WB },

    /* subtitle codec */
    { AV_CODEC_ID_DVD_SUBTITLE      , enAVFSCI_DVD_SUBTITLE },
```

```
    /* other codec */
    { AV_CODEC_ID_NONE              , 0 },
};
const AVCodecTag * ff_avfs_codectags_muxer[] = {ff_avfs_codec_tags,0};

///<  核心句柄数据结构
typedef struct AVFastSearching_MuxerContext{
    const AVClass * pclass;                         ///< FFmpeg系统默认
    int32_t save_through_media_data;                ///<  是否保存添加的媒体信息
    int32_t save_avfs_media_data;                   ///<  是否保存自己生成的媒体信息
    int64_t seektable_keypacket_min_interval_pts;   ///<  生成SeekTable关键帧时间间隔
    int32_t media_encryption_mode;
    AVFS_SYSTEM_INFO this_system_info;              ///<  核心数据成员
}AVFS_MUXER_CONTEXT;

///<  命令参数
#define AVFS_MUXER_OFFSET(x) offsetof(AVFastSearching_MuxerContext, x)
#define E AV_OPT_FLAG_ENCODING_PARAM
static const AVOption ff_avfs_muxer_options[] = {
    { "save_through_media_data", "0 is not save through media data,others is save",
AVFS_MUXER_OFFSET(save_through_media_data), AV_OPT_TYPE_INT, {1}, 0, INT_MAX, E },
    { "save_avfs_media_data", "0 is not save avfs media data,others is save",
AVFS_MUXER_OFFSET(save_avfs_media_data), AV_OPT_TYPE_INT, {1}, 0, INT_MAX, E },
    { "seektable_keypacket_min_interval_pts", "create seek table, the same stream key packet interval
pts(1/10000000 second).", AVFS_MUXER_OFFSET(seektable_keypacket_min_interval_pts),
AV_OPT_TYPE_INT64, {20000000}, 0, INT64_MAX, E },
    { NULL },
};

///< AVFS容器封装插件类定义
static const AVClass ff_avfs_muxer_class = {
    AVFS_FORMAT_NAME"_enc",             ///<  类名称
    av_default_item_name,               ///<  默认处理函数
    ff_avfs_muxer_options,              ///<  参数
    LIBAVUTIL_VERSION_INT,              ///<  版本
};
```

第二步：注册插件

```
///< 插件注册函数
bool static_register_avfs_muxer(){
bool isfind = false;
AVOutputFormat * muxer = NULL;
while(muxer = av_oformat_next(muxer))
{
    if(muxer->name && !strcmp(muxer->name,ff_avfs_muxer.name))
    {
    // 检查插件是否已经存在
    av_log(NULL,AV_LOG_INFO,"the %s format is already registered.\n",ff_avfs_muxer.name);
    isfind = true;
    }
}
if(isfind == false)
{
    // 未存在，注册插件
    av_register_output_format(&ff_avfs_muxer);
    isfind = true;
    av_log(NULL,AV_LOG_INFO,"register %s muxer.\n",ff_avfs_muxer.name);
}
return isfind;
}
```

第三步：封装容器头

根据前面章节关于封装流程的讲解已经知道了，封装一个文件的具体流程和函数调用。先调用avformat_alloc_output_context2以及avformat_new_stream等函数并不会触发write_header接口。因为前面的都是准备操作，只有调用avformat_write_header进行文件头封装的时候会触发。

```
///< 封装文件头
static int ff_avfs_muxer_write_header(AVFormatContext *s)
{
    // 获取核心句柄
    struct AVFastSearching_MuxerContext * h = (struct AVFastSearching_MuxerContext *)s->priv_data;
    if(h == NULL || s->pb == NULL)
    {return -1;}
    memset(&h->this_system_info,0,sizeof(h->this_system_info));
    av_log(s,AV_LOG_INFO,"version= %d packet header size = %d
Byte.\n",AVFS_VERSION,sizeof(struct avfs_list_layer_header));

    //save system info
    h->this_system_info.property_info1.info1_start_time = 0;
    h->this_system_info.property_info1.info2_duration = 0xFFFFFFFFFFFFFFFF;
    h->this_system_info.property_info1.info3_framenumber = 0;
    h->this_system_info.property_info1.info4_filesize = 0;
    h->this_system_info.property_info1.info5_frame_seektable_file_pos = 0;
    h->this_system_info.property_info1.info6_other_data_file_pos = 0;
    if(s->metadata)
    {
        if(h->this_system_info.property_media_data2 == NULL)
        {h->this_system_info.property_media_data2 = new std::map<char *,char *>;}
        if(h->save_through_media_data)
        {
            for(int i = 0; i < s->metadata->count; i++)
            {
                if(s->metadata->elems[i].key && s->metadata->elems[i].value
                    && strlen(s->metadata->elems[i].key) > 0 &&
strlen(s->metadata->elems[i].value))
                {
                    h->this_system_info.property_media_data2->insert(std::map<char *, char
*>::value_type(
                        strdup(s->metadata->elems[i].key),
                        strdup(s->metadata->elems[i].value)));
                }
            }
```

```
            }
        }

        //save streams number
        int stream_numbers = 0;
        if(s->nb_streams > 0)
        {
            //save streams info header first stream id is from 1
            h->this_system_info.info_streams4 = new
AVFS_SYSTEM_INFO::avfs_stream_info[s->nb_streams];

        memset(h->this_system_info.info_streams4,0,sizeof(AVFS_SYSTEM_INFO::avfs_stream_info)*s->n
b_streams);
            for(int i = 0;i < s->nb_streams;i++)
            {
                AVStream * stream = s->streams[i];
                AVFS_SYSTEM_INFO::avfs_stream_info * avfs_stream =
h->this_system_info.info_streams4 + i;
                if(stream && avfs_stream && stream->codec &&
                    stream->codec->codec_type > AVMEDIA_TYPE_UNKNOWN &&
                    stream->codec->codec_type < AVMEDIA_TYPE_NB)
                {
                    switch(stream->codec->codec_type)
                    {
                    case AVMEDIA_TYPE_VIDEO:
                        {
                            //Video codec
                            avfs_stream->stream_id_1.s_index = i+1;
                            switch(stream->codec->codec_id)
                            {
                            case AV_CODEC_ID_H264:
                                {avfs_stream->stream_id_1.codec_id = enAVFSCI_H264;}break;
                            case AV_CODEC_ID_H265:
                                {avfs_stream->stream_id_1.codec_id = enAVFSCI_H265;}break;
                            case AV_CODEC_ID_VP8:
                                {avfs_stream->stream_id_1.codec_id = enAVFSCI_VP8;}break;
                            case AV_CODEC_ID_VP9:
                                {avfs_stream->stream_id_1.codec_id = enAVFSCI_VP9;}break;
                            case AV_CODEC_ID_MPEG4:
                                {avfs_stream->stream_id_1.codec_id = enAVFSCI_MPEG4;}break;
                            case AV_CODEC_ID_H263:
                                {avfs_stream->stream_id_1.codec_id = enAVFSCI_H263;}break;
                            case AV_CODEC_ID_RV10:
                                {avfs_stream->stream_id_1.codec_id = enAVFSCI_RV10;}break;
```

```
                case AV_CODEC_ID_MJPEG:
                    {avfs_stream->stream_id_1.codec_id = enAVFSCI_MJPEG;}break;
                case AV_CODEC_ID_LJPEG:
                    {avfs_stream->stream_id_1.codec_id = enAVFSCI_LJPEG;}break;
                case AV_CODEC_ID_JPEG2000:
                    {avfs_stream->stream_id_1.codec_id =
enAVFSCI_JPEG2000;}break;
                case AV_CODEC_ID_PNG:
                    {avfs_stream->stream_id_1.codec_id = enAVFSCI_PNG;}break;
                case AV_CODEC_ID_BMP:
                    {avfs_stream->stream_id_1.codec_id = enAVFSCI_BMP;}break;
                case AV_CODEC_ID_TIFF:
                    {avfs_stream->stream_id_1.codec_id = enAVFSCI_TIFF;}break;
                case AV_CODEC_ID_RAWVIDEO:
                    {avfs_stream->stream_id_1.codec_id =
enAVFSCI_RAWVIDEO;}break;
                default:
                    {
                        av_log(s,AV_LOG_WARNING,"video not support
codec_id=%d codec_name=%s.\n",stream->codec->codec_id,
                                stream->codec->codec?stream->codec->codec->name:"");
                        return AVERROR_INVALIDDATA;
                    }
                }

                //picture size
                avfs_stream->video_param_2.width_1 = stream->codec->width;
                avfs_stream->video_param_2.height_2 = stream->codec->height;

                //fps
                avfs_stream->video_param_2.fps_numerator_3 =
stream->codec->time_base.num;
                avfs_stream->video_param_2.fps_denominator_4 =
stream->codec->time_base.den;

                //color space
                if(avfs_stream->general_param_3.media_data_4 == NULL)
                {avfs_stream->general_param_3.media_data_4 = new std::map<char *,char
*>;}
                switch(stream->codec->pix_fmt)
                {
                case AV_PIX_FMT_YUV420P:
                    {
                    if(h->save_avfs_media_data)
```

```
                                avfs_stream->general_param_3.media_data_4->insert(std::map<char
*, char *>::value_type(
                                    strdup(AVFS_METADATA_KEY_COLOR_SPACE),
                                    strdup("YUV 4:2:0")));
                                avfs_stream->video_param_2.color_space_5 =
enAVFSColorSpace::enAVFSCP_YV12;
                                }break;
                            case AV_PIX_FMT_YUYV422:
                            case AV_PIX_FMT_YUV422P:
                                {
                                if(h->save_avfs_media_data)
                                avfs_stream->general_param_3.media_data_4->insert(std::map<char
*, char *>::value_type(
                                    strdup(AVFS_METADATA_KEY_COLOR_SPACE),
                                    strdup("YUV 4:2:2")));
                                avfs_stream->video_param_2.color_space_5 =
enAVFSColorSpace::enAVFSCP_YV16;
                                }break;
                            case AV_PIX_FMT_YUV444P:
                                {
                                if(h->save_avfs_media_data)
                                avfs_stream->general_param_3.media_data_4->insert(std::map<char
*, char *>::value_type(
                                    strdup(AVFS_METADATA_KEY_COLOR_SPACE),
                                    strdup("YUV 4:4:4")));
                                avfs_stream->video_param_2.color_space_5 =
enAVFSColorSpace::enAVFSCP_YUV444;
                                }break;
                            case AV_PIX_FMT_YUV410P:
                                {
                                if(h->save_avfs_media_data)
                                avfs_stream->general_param_3.media_data_4->insert(std::map<char
*, char *>::value_type(
                                    strdup(AVFS_METADATA_KEY_COLOR_SPACE),
                                    strdup("YUV 4:1:0")));
                                avfs_stream->video_param_2.color_space_5 =
enAVFSColorSpace::enAVFSCP_YUV9;
                                }break;
                            case AV_PIX_FMT_YUV411P:
                                {
                                if(h->save_avfs_media_data)
                                avfs_stream->general_param_3.media_data_4->insert(std::map<char
*, char *>::value_type(
                                    strdup(AVFS_METADATA_KEY_COLOR_SPACE),
```

```
                                        strdup("YUV 4:1:1")));
                                    avfs_stream->video_param_2.color_space_5 =
enAVFSColorSpace::enAVFSCP_Y41P;
                                    }break;
                                case AV_PIX_FMT_GRAY8:
                                    {
                                    if(h->save_avfs_media_data)
                                    avfs_stream->general_param_3.media_data_4->insert(std::map<char
*, char *>::value_type(
                                        strdup(AVFS_METADATA_KEY_COLOR_SPACE),
                                        strdup("GRAY 8Bit")));
                                    avfs_stream->video_param_2.color_space_5 =
enAVFSColorSpace::enAVFSCP_GRAY;
                                    }break;
                                case AV_PIX_FMT_RGB24:
                                    {
                                    if(h->save_avfs_media_data)
                                    avfs_stream->general_param_3.media_data_4->insert(std::map<char
*, char *>::value_type(
                                        strdup(AVFS_METADATA_KEY_COLOR_SPACE),
                                        strdup("RGB 24Bit")));
                                    avfs_stream->video_param_2.color_space_5 =
enAVFSColorSpace::enAVFSCP_RGB24;
                                    }break;
                                case AV_PIX_FMT_ARGB:
                                    {
                                    if(h->save_avfs_media_data)
                                    avfs_stream->general_param_3.media_data_4->insert(std::map<char
*, char *>::value_type(
                                        strdup(AVFS_METADATA_KEY_COLOR_SPACE),
                                        strdup("RGBA 32Bit")));
                                    avfs_stream->video_param_2.color_space_5 =
enAVFSColorSpace::enAVFSCP_RGBA;
                                    }break;
                                }
                            }break;
                        case AVMEDIA_TYPE_AUDIO:
                            {
                                //Audio codec
                                avfs_stream->stream_id_1.s_index = i+1;
                                switch(stream->codec->codec_id)
                                {
                                case AV_CODEC_ID_MP2:
                                    {avfs_stream->stream_id_1.codec_id = enAVFSCI_MP2;}break;
```

```
                                case AV_CODEC_ID_MP3:
                                    {avfs_stream->stream_id_1.codec_id = enAVFSCI_MP3;}break;
                                case AV_CODEC_ID_AAC:
                                    {avfs_stream->stream_id_1.codec_id = enAVFSCI_AAC;}break;
                                case AV_CODEC_ID_AC3:
                                    {avfs_stream->stream_id_1.codec_id = enAVFSCI_AC3;}break;
                                case AV_CODEC_ID_DTS:
                                    {avfs_stream->stream_id_1.codec_id = enAVFSCI_DTS;}break;
                                case AV_CODEC_ID_ADPCM_G726:
                                    {avfs_stream->stream_id_1.codec_id =
enAVFSCI_ADPCM_G726;}break;
                                case AV_CODEC_ID_AMR_NB:
                                    {avfs_stream->stream_id_1.codec_id =
enAVFSCI_AMR_NB;}break;
                                case AV_CODEC_ID_AMR_WB:
                                    {avfs_stream->stream_id_1.codec_id =
enAVFSCI_AMR_WB;}break;
                                default:
                                    {
                                        av_log(s,AV_LOG_WARNING,"audio not support
codec_id=%d codec_name=%s.\n",stream->codec->codec_id,
                                                stream->codec->codec?stream->codec->codec->name:"");
                                        return AVERROR_INVALIDDATA;
                                    }break;
                                }

                                //Audio Param
                                if(avfs_stream->general_param_3.media_data_4 == NULL)
                                {avfs_stream->general_param_3.media_data_4 = new std::map<char *,char
*>;}
                                avfs_stream->audio_param_2.a_sample_rate_1 =
stream->codec->sample_rate/100;
                                switch(stream->codec->sample_fmt)
                                {
                                case AV_SAMPLE_FMT_U8:
                                case AV_SAMPLE_FMT_U8P:
                                    {
                                        avfs_stream->audio_param_2.a_sample_format_2 =
enAVFSSampleFormat::enAVFSSF_BYTE;
                                    }break;
                                case AV_SAMPLE_FMT_S16:
                                case AV_SAMPLE_FMT_S16P:
                                    {
                                        avfs_stream->audio_param_2.a_sample_format_2 =
```

```
enAVFSSampleFormat::enAVFSSF_SHORT;
                    }break;
                case AV_SAMPLE_FMT_S32:
                case AV_SAMPLE_FMT_S32P:
                    {
                        avfs_stream->audio_param_2.a_sample_format_2 =
enAVFSSampleFormat::enAVFSSF_INT;
                    }break;
                case AV_SAMPLE_FMT_FLT:
                case AV_SAMPLE_FMT_FLTP:
                    {
                        avfs_stream->audio_param_2.a_sample_format_2 =
enAVFSSampleFormat::enAVFSSF_FLOAT;
                    }break;
                case AV_SAMPLE_FMT_DBL:
                case AV_SAMPLE_FMT_DBLP:
                    {
                        avfs_stream->audio_param_2.a_sample_format_2 =
enAVFSSampleFormat::enAVFSSF_DOUBLE;
                    }break;
                }
                if(stream->codec->channels == 1 && stream->codec->channel_layout ==
AV_CH_LAYOUT_MONO)
                {
                    avfs_stream->audio_param_2.a_channel_layout_3 =
enAVFSChannelLayout::enAVFSCL_MONO;
                    if(h->save_avfs_media_data)
                    avfs_stream->general_param_3.media_data_4->insert(std::map<char
*, char *>::value_type(
                        strdup(AVFS_METADATA_KEY_CHANNEL_LAYOUT),
                        strdup("Mono")));
                }
                else if(stream->codec->channels == 2 && stream->codec->channel_layout
== AV_CH_LAYOUT_STEREO)
                {
                    avfs_stream->audio_param_2.a_channel_layout_3 =
enAVFSChannelLayout::enAVFSCL_STEREO;
                    if(h->save_avfs_media_data)
                    avfs_stream->general_param_3.media_data_4->insert(std::map<char
*, char *>::value_type(
                        strdup(AVFS_METADATA_KEY_CHANNEL_LAYOUT),
                        strdup("Stereo")));
                }
                else if(stream->codec->channels == 5 && stream->codec->channel_layout
```

```
== AV_CH_LAYOUT_5POINT0)
                                {
                                    avfs_stream->audio_param_2.a_channel_layout_3 =
enAVFSChannelLayout::enAVFSCL_5SURROUND_STEREO;
                                    if(h->save_avfs_media_data)
                                    avfs_stream->general_param_3.media_data_4->insert(std::map<char
*, char *>::value_type(
                                        strdup(AVFS_METADATA_KEY_CHANNEL_LAYOUT),
                                        strdup("Surround Stereo")));
                                }
                                else if(stream->codec->channels == 6 && stream->codec->channel_layout
== AV_CH_LAYOUT_5POINT1)
                                {
                                    avfs_stream->audio_param_2.a_channel_layout_3 =
enAVFSChannelLayout::enAVFSCL_6SURROUND_STEREO;
                                    if(h->save_avfs_media_data)
                                    avfs_stream->general_param_3.media_data_4->insert(std::map<char
*, char *>::value_type(
                                        strdup(AVFS_METADATA_KEY_CHANNEL_LAYOUT),
                                        strdup("5.1 Surround Stereo")));
                                }
                                else
                                {
                                    avfs_stream->audio_param_2.a_channel_layout_3 =
enAVFSChannelLayout::enAVFSCL_None;
                                    char str_channel_numbers[50] = {0};
                                    sprintf(str_channel_numbers,"%d
channel",stream->codec->channels);
                                    if(h->save_avfs_media_data)
                                    avfs_stream->general_param_3.media_data_4->insert(std::map<char
*, char *>::value_type(
                                        strdup(AVFS_METADATA_KEY_CHANNEL_LAYOUT),
                                        strdup(str_channel_numbers)));
                                }
                            }break;
                        case AVMEDIA_TYPE_SUBTITLE:
                            {
                                //Subtitle codec
                                avfs_stream->stream_id_1.s_index = i+1;
                                switch(stream->codec->codec_id)
                                {
                                case AV_CODEC_ID_DVD_SUBTITLE:
                                    {avfs_stream->stream_id_1.codec_id =
enAVFSCI_DVD_SUBTITLE;}break;
```

```
                default:
                    {
                        av_log(s,AV_LOG_WARNING,"subtitle not support codec_id=%d codec_name=%s.\n",stream->codec->codec_id,
                            stream->codec->codec?stream->codec->codec->name:"");
                        return AVERROR_INVALIDDATA;
                    }break;
                }
            }break;
        case AVMEDIA_TYPE_DATA:             ///< Opaque data information usually continuous
        case AVMEDIA_TYPE_ATTACHMENT:       ///< Opaque data information usually sparse
            {
                //Other codec
                avfs_stream->stream_id_1.s_index = i+1;
                avfs_stream->stream_id_1.codec_id = enAVFSCI_OTHERS;
            }break;
        }

        //general data
        if(stream->codec->bit_rate > 0)
        {avfs_stream->general_param_3.bit_rate_1 = stream->codec->bit_rate;}
        if(avfs_stream->general_param_3.media_data_4 == NULL)
        {avfs_stream->general_param_3.media_data_4 = new std::map<char *,char *>;}
        if(stream->codec->codec->long_name && h->save_avfs_media_data)
        {
            avfs_stream->general_param_3.media_data_4->insert(std::map<char *, char *>::value_type(
                strdup(AVFS_METADATA_KEY_CODEC_NAME),
                strdup(stream->codec->codec->long_name)));
        }
        else if(stream->codec->codec->name && h->save_avfs_media_data)
        {
            avfs_stream->general_param_3.media_data_4->insert(std::map<char *, char *>::value_type(
                strdup(AVFS_METADATA_KEY_CODEC_NAME),
                strdup(stream->codec->codec->name)));
        }
        avfs_stream->general_param_3.extra_data_size_2 = stream->codec->extradata_size;
        if(avfs_stream->general_param_3.extra_data_size_2 > 0)
        {
            avfs_stream->general_param_3.extra_data_3 = new uint8_t[avfs_stream->general_param_3.extra_data_size_2];
```

```
        memcpy(avfs_stream->general_param_3.extra_data_3,stream->codec->extradata,avfs_stream->general
_param_3.extra_data_size_2);
                        }
                        if(stream->metadata)
                        {
                            if(avfs_stream->general_param_3.media_data_4 == NULL)
                            {avfs_stream->general_param_3.media_data_4 = new std::map<char *,char *>;}
                            if(h->save_through_media_data)
                            {
                                for(int j = 0;j < stream->metadata->count; j++)
                                {
                                    if(stream->metadata->elems[j].key &&
stream->metadata->elems[j].value &&
                                            strlen(stream->metadata->elems[j].key) > 0 &&
strlen(stream->metadata->elems[j].value) > 0)
                                        {

        avfs_stream->general_param_3.media_data_4->insert(std::map<char *, char *>::value_type(
                                            strdup(stream->metadata->elems[j].key),
                                            strdup(stream->metadata->elems[j].value)));
                                        }
                                }
                            }
                        }
                        avfs_stream->AVStream_index = stream->index;
                        avfs_stream->AVStream_id = stream->id;
                        stream_numbers++;
                    }
                }
        }

        //write file marked
        avio_write(s->pb,avfs_format_header_marked,sizeof(avfs_format_header_marked));

        //write property info
        avio_write(s->pb,(unsigned char
*)&h->this_system_info.property_info1,sizeof(h->this_system_info.property_info1));

        //save file size
        ADD_FILE_SIZE(sizeof(avfs_format_header_marked));
        ADD_FILE_SIZE(sizeof(h->this_system_info.property_info1));

        //write property media data
```

```
        uint16_t property_media_data_numbers = 0;
        if(h->this_system_info.property_media_data2)
        {property_media_data_numbers = h->this_system_info.property_media_data2->size();}
        avio_write(s->pb,(unsigned char
*)&property_media_data_numbers,sizeof(property_media_data_numbers));
        ADD_FILE_SIZE(sizeof(property_media_data_numbers));
        std::map<char *,char *> ::iterator iter_pi;
        iter_pi = h->this_system_info.property_media_data2->begin();
        for(int i = 0;((iter_pi != h->this_system_info.property_media_data2->end()) && i<
property_media_data_numbers);i++,iter_pi++)
        {
            uint16_t key_size = 0,value_size = 0;
            if(strlen(iter_pi->first) > 0)
            {key_size = strlen(iter_pi->first);}
            if(strlen(iter_pi->second) > 0)
            {value_size = strlen(iter_pi->second);}
            avio_write(s->pb,(unsigned char *)&key_size,sizeof(key_size));
            ADD_FILE_SIZE(sizeof(key_size));
            if(key_size)
            {
                avio_write(s->pb,(const unsigned char *)iter_pi->first,key_size);
                ADD_FILE_SIZE(key_size);
            }
            avio_write(s->pb,(unsigned char *)&value_size,sizeof(value_size));
            ADD_FILE_SIZE(sizeof(value_size));
            if(value_size)
            {
                avio_write(s->pb,(const unsigned char *)iter_pi->second,value_size);
                ADD_FILE_SIZE(value_size);
            }
        }

        //save system info stream start position offset
        h->this_system_info.system_info_stream_start_pos_offset =
h->this_system_info.property_info1.info4_filesize;

        //write streams number
        h->this_system_info.info_stream_number3 = stream_numbers;
        avio_write(s->pb,(unsigned char
*)&h->this_system_info.info_stream_number3,sizeof(h->this_system_info.info_stream_number3));
        ADD_FILE_SIZE(sizeof(h->this_system_info.info_stream_number3));

        //write streams info header
        for(int i = 0;i < stream_numbers;i++)
```

```
    {
        //stream
        AVFS_SYSTEM_INFO::avfs_stream_info * avfs_stream = h->this_system_info.info_streams4 +
i;

        //save stream offset
        avfs_stream->this_stream_start_pos_offset = h->this_system_info.property_info1.info4_filesize;

        //write stream id
        avio_write(s->pb,(unsigned char
*)&avfs_stream->stream_id_1,sizeof(avfs_stream->stream_id_1));
        ADD_FILE_SIZE(sizeof(avfs_stream->stream_id_1));

        //write stream param
        if(avfs_stream->stream_id_1.codec_id >= enAVFSCI_VIDEO &&
            avfs_stream->stream_id_1.codec_id < enAVFSCI_AUDIO)
        {
            //write video stream param
            avio_write(s->pb,(unsigned char
*)&avfs_stream->video_param_2,sizeof(avfs_stream->video_param_2));
            ADD_FILE_SIZE(sizeof(avfs_stream->video_param_2));
        }
        else if(avfs_stream->stream_id_1.codec_id >= enAVFSCI_AUDIO &&
            avfs_stream->stream_id_1.codec_id < enAVFSCI_SUBTITLE)
        {
            //write audio stream param
            avio_write(s->pb,(unsigned char
*)&avfs_stream->audio_param_2,sizeof(avfs_stream->audio_param_2));
            ADD_FILE_SIZE(sizeof(avfs_stream->audio_param_2));
        }
        else if(avfs_stream->stream_id_1.codec_id >= enAVFSCI_SUBTITLE &&
            avfs_stream->stream_id_1.codec_id < enAVFSCI_OTHERS)
        {
            //write subtitle stream param
            avio_write(s->pb,(unsigned char
*)&avfs_stream->subtitle_param_2,sizeof(avfs_stream->subtitle_param_2));
            ADD_FILE_SIZE(sizeof(avfs_stream->subtitle_param_2));
        }
        else
        {av_log(s,AV_LOG_WARNING,"this is unknow stream[index=%d] to
save.\n",avfs_stream->stream_id_1.s_index);}

        //write stream general param
        avio_write(s->pb,(unsigned char
```

```
*)&avfs_stream->general_param_3.bit_rate_1,sizeof(avfs_stream->general_param_3.bit_rate_1));
            ADD_FILE_SIZE(sizeof(avfs_stream->general_param_3.bit_rate_1));
            avio_write(s->pb,(unsigned char
*)&avfs_stream->general_param_3.extra_data_size_2,sizeof(avfs_stream->general_param_3.extra_data_siz
e_2));
            ADD_FILE_SIZE(sizeof(avfs_stream->general_param_3.extra_data_size_2));
            if(avfs_stream->general_param_3.extra_data_size_2 > 0)
            {
                avio_write(s->pb,(unsigned char
*)avfs_stream->general_param_3.extra_data_3,avfs_stream->general_param_3.extra_data_size_2);
                ADD_FILE_SIZE(avfs_stream->general_param_3.extra_data_size_2);
            }
            uint16_t stream_media_data_numbers = 0;
            if(avfs_stream->general_param_3.media_data_4)
            {stream_media_data_numbers = avfs_stream->general_param_3.media_data_4->size();}
            avio_write(s->pb,(unsigned char
*)&stream_media_data_numbers,sizeof(stream_media_data_numbers));
            ADD_FILE_SIZE(stream_media_data_numbers);
            if(stream_media_data_numbers > 0)
            {
                std::map<char *,char *> ::iterator iter_pi;
                iter_pi = avfs_stream->general_param_3.media_data_4->begin();
                for(int i = 0;((iter_pi != avfs_stream->general_param_3.media_data_4->end()) && i<
stream_media_data_numbers);i++,iter_pi++)
                {
                    uint16_t key_size = 0,value_size = 0;
                    if(strlen(iter_pi->first) > 0)
                    {key_size = strlen(iter_pi->first);}
                    if(strlen(iter_pi->second) > 0)
                    {value_size = strlen(iter_pi->second);}
                    avio_write(s->pb,(unsigned char *)&key_size,sizeof(key_size));
                    ADD_FILE_SIZE(sizeof(key_size));
                    if(key_size)
                    {
                        avio_write(s->pb,(const unsigned char *)iter_pi->first,key_size);
                        ADD_FILE_SIZE(key_size);
                    }
                    avio_write(s->pb,(unsigned char *)&value_size,sizeof(value_size));
                    ADD_FILE_SIZE(sizeof(value_size));
                    if(value_size)
                    {
                        avio_write(s->pb,(const unsigned char *)iter_pi->second,value_size);
                        ADD_FILE_SIZE(value_size);
                    }
```

```
                }
            }
        }

        //Save Seek Table
        if(h->this_system_info.info_stream_number3 > 0)
        {
            h->this_system_info.avfs_system_key_table_stream_number1 = 0;
            h->this_system_info.avfs_system_key_table_streams2 = new std::map<int ,
AVFS_SYSTEM_INFO::avfs_system_key_table_stream_info *>;
            for(int i = 0;i < h->this_system_info.info_stream_number3;i++)
            {
                AVFS_SYSTEM_INFO::avfs_stream_info * avfs_stream =
h->this_system_info.info_streams4 + i;
                //Only Save Video
                if(avfs_stream->stream_id_1.codec_id >= enAVFSCI_VIDEO &&
avfs_stream->stream_id_1.codec_id < enAVFSCI_AUDIO)
                {
                    AVFS_SYSTEM_INFO::avfs_system_key_table_stream_info * pElement = new
AVFS_SYSTEM_INFO::avfs_system_key_table_stream_info;
                    pElement->stream_id_1 = avfs_stream->stream_id_1;
                    pElement->stream_packet_number_2 = 0;
                    pElement->pre_stream_key_packet_pts = 0;
                    h->this_system_info.avfs_system_key_table_streams2->insert(std::map<int,
AVFS_SYSTEM_INFO::avfs_system_key_table_stream_info *>::value_type(
                        avfs_stream->AVStream_index,
                        pElement));
                    avfs_stream->seektable = pElement;
                    h->this_system_info.avfs_system_key_table_stream_number1++;
                }
            }
        }

        //save system info size
        h->this_system_info.stream_data_start_pos_offset =
h->this_system_info.property_info1.info4_filesize;

        //flush
        avio_flush(s->pb);
        return 0;
}
```

第四步：封包处理

关于封包的操作其主要是调用 av_interleaved_write_frame 将编码后的字节流数据包 AVPacket 保存到容器中，注意还有 AVStream 数据流这一逻辑层，框架会调用插件的 write_packet 方法最终实现封包操作。

```
///< AVFS封包函数实现
static int ff_avfs_muxer_write_packet(AVFormatContext *s, AVPacket *pkt)
{
    if(s && s->priv_data && s->pb)
    {
        // 获取到核心句柄对象
        struct AVFastSearching_MuxerContext * h = (struct AVFastSearching_MuxerContext
*)s->priv_data;
        AVIOContext * pb = s->pb;      // 协议层对象
        if(pkt && pkt->stream_index < s->nb_streams &&
            pkt->stream_index < h->this_system_info.info_stream_number3 &&
            pkt->data && pkt->size>0)
        {
            // 数据流分包
            AVStream * stream = s->streams[pkt->stream_index];
            AVFS_SYSTEM_INFO::avfs_stream_info * avfs_stream =
h->this_system_info.info_streams4 + pkt->stream_index;
            if(stream && stream->codec &&
                avfs_stream && avfs_stream->AVStream_id == stream->id &&
avfs_stream->AVStream_index == stream->index)
            {
                //Calculate pts
                double cur_pts = -1,cur_dts = -1;
                if(pkt->pts != AV_NOPTS_VALUE)
                {
                    if(stream->time_base.den && stream->time_base.num)
                    {cur_pts =
(double)pkt->pts/(double)stream->time_base.den*(double)stream->time_base.num;}
                    else
                    {cur_pts = (double)pkt->pts/AV_TIME_BASE;}
                }

                //Calculate dts
                if(pkt->dts != AV_NOPTS_VALUE)
                {
                    if(stream->time_base.den && stream->time_base.num)
                    {cur_dts =
```

```
(double)pkt->dts/(double)stream->time_base.den*(double)stream->time_base.num;}
                    else
                    {cur_dts = (double)pkt->dts/AV_TIME_BASE;}
                }

                //ReCalculate
                if(!(cur_pts >= 0 && cur_pts >= cur_dts && ((uint64_t)cur_dts*10000000) >=
avfs_stream->pre_packet_dts))
                {
                    av_log(s,AV_LOG_WARNING,"the same stream verify packet, index=%d
size=%d ,pre_pts=%lf > pts = %lf pre_dts=%I64d dts=%I64d .\n",

      pkt->stream_index,pkt->size,(double)avfs_stream->pre_packet_pts/10000000,cur_pts,avfs_stream->pr
e_packet_dts,pkt->dts);
                }
                if(cur_dts < 0)
                {cur_dts = (double)avfs_stream->pre_packet_dts/10000000;}
                if(cur_pts < 0)
                {cur_pts = (double)avfs_stream->pre_packet_pts/10000000;}

                //Verify pts、dts
                if(cur_pts >= cur_dts)
                {
                    //Save AVStream start_time
                    if(!avfs_stream->stream_total_packet_number && stream->start_time ==
AV_NOPTS_VALUE && pkt->pts != AV_NOPTS_VALUE)
                    {stream->start_time = pkt->pts;}

                    //Keep dts
                    if(cur_dts >= 0)
                    {avfs_stream->pre_packet_dts = (uint64_t)((uint64_t)cur_dts*10000000);}

                    //write packet
                    ff_avfs_muxer_write_frame(s,
                        &h->this_system_info,
                        avfs_stream,pkt->flags?true:false,
                        false,
                        cur_pts,
                        cur_dts,
                        h->seektable_keypacket_min_interval_pts,
                        pkt->data,
                        pkt->size);
                }
                else
```

```
                {
                    av_log(s,AV_LOG_ERROR,"Not save the same stream packet, index=%d
size=%d ,pre_pts=%lf > pts = %lf pre_dts=%I64d dts=%I64d .\n",

    pkt->stream_index,pkt->size,(double)avfs_stream->pre_packet_pts/10000000,cur_pts,avfs_stream->pr
e_packet_dts,pkt->dts);
                }
            }
            else
            {av_log(s,AV_LOG_WARNING,"Not save the unknow stream packet, index=%d
size=%d.\n",pkt->stream_index,pkt->size);}
        }
        return 0;
    }
    else
    {return AVERROR(28);}
}

///<  具体 AVFS 封包实现函数
void ff_avfs_muxer_write_frame(
    AVFormatContext *s,
    AVFS_SYSTEM_INFO * this_system_info,
    AVFS_SYSTEM_INFO::avfs_stream_info * avfs_stream,
    bool keyframe,
    bool encryption,
    double pts,                                ///<  以秒为单位
    double dts,
    uint64_t seektable_keypacket_min_interval_pts,
    uint8_t * data,
    uint32_t datasize)
{
    //param
    AVFS_LIST_LAYER_HEADER packet_header = {0};
    packet_header.ext_info = 0;
    packet_header.keyframe = keyframe?1:0;
    packet_header.encryp_option = encryption?(0x01):0;
    if(pts >= 0)
    {packet_header.packet_clock_pts = (uint64_t)(pts*10000000);}
    else
    {
        if(avfs_stream->video_param_2.fps_numerator_3 &&
avfs_stream->video_param_2.fps_denominator_4)
        {packet_header.packet_clock_pts =
(uint64_t)((double)avfs_stream->stream_total_packet_number*(10000000/((double)avfs_stream->video_par
```

```
am_2.fps_denominator_4/avfs_stream->video_param_2.fps_numerator_3)));}
            else
            {packet_header.packet_clock_pts = -1;}
        }
        if(dts >= 0)
        {
            packet_header.packet_index_or_clock_dts_flag = 1;
            packet_header.packet_index_or_clock_dts = (uint64_t)(dts*10000000);
        }
        else
        {
            packet_header.packet_index_or_clock_dts_flag = 0;
            packet_header.packet_index_or_clock_dts = avfs_stream->stream_total_packet_number;
        }
        packet_header.stream_id = avfs_stream->stream_id_1;
        packet_header.current_packet_size = datasize;

        //write pre_key_packet save current_key_packet
        if(packet_header.keyframe &&
            avfs_stream->stream_id_1.codec_id >= enAVFSCI_VIDEO &&          //视频
            avfs_stream->stream_id_1.codec_id < enAVFSCI_AUDIO &&
            s->pb->seekable && avfs_stream->pre_key_packet_file_pos > 0 &&
            (this_system_info->property_info1.info4_filesize - avfs_stream->pre_key_packet_file_pos) > 0
&&
            (this_system_info->property_info1.info4_filesize - avfs_stream->pre_key_packet_file_pos) <=
0x7FFFFFFF)
        {
            int64_t next_key_packet_offset =
offsetof(AVFS_LIST_LAYER_HEADER,next_key_packet_pos);
            int64_t seek_offset = avfs_stream->pre_key_packet_file_pos + next_key_packet_offset;
            int64_t seek_u64 = avio_seek(s->pb,seek_offset,SEEK_SET);
            if(seek_u64 == seek_offset)
            {
                avio_wl64(s->pb,this_system_info->property_info1.info4_filesize);
                seek_u64 = avio_seek(s->pb, this_system_info->property_info1.info4_filesize,
SEEK_SET);
                if(this_system_info->property_info1.info4_filesize != seek_u64)
                {av_log(s,AV_LOG_ERROR,"avfs write pre_key_packet save current_key_packet seek
error=%I64d.\n",seek_u64);}
            }
            else
            {avio_seek(s->pb,0,SEEK_END);}
        }
```

```
        //save packet pos and pts info
        avfs_stream->pre_packet_file_pos = this_system_info->pre_packet_file_pos =
this_system_info->property_info1.info4_filesize;
        if(pts >= 0)
        {avfs_stream->pre_packet_pts = this_system_info->pre_packet_pts =
packet_header.packet_clock_pts;}
        else
        {avfs_stream->pre_packet_pts = this_system_info->pre_packet_pts = 0;}
        if(packet_header.keyframe)
        {
            if(avfs_stream->stream_id_1.codec_id >= enAVFSCI_VIDEO &&
avfs_stream->stream_id_1.codec_id < enAVFSCI_AUDIO)
            {
                this_system_info->pre_key_packet_file_pos =
this_system_info->property_info1.info4_filesize;
                if(pts >= 0)
                {this_system_info->pre_key_packet_pts = packet_header.packet_clock_pts;}
                else
                {this_system_info->pre_key_packet_pts = 0;}
            }
            avfs_stream->pre_key_packet_file_pos = this_system_info->property_info1.info4_filesize;
            if(pts >= 0)
            {avfs_stream->pre_key_packet_pts = packet_header.packet_clock_pts;}
            else
            {avfs_stream->pre_key_packet_pts = 0;}
        }

        //save seek table
        if(packet_header.keyframe && pts >= 0 &&
            avfs_stream->stream_id_1.codec_id >= enAVFSCI_VIDEO &&
            avfs_stream->stream_id_1.codec_id < enAVFSCI_AUDIO)
        {
           if(this_system_info->avfs_system_key_table_streams2->count(avfs_stream->AVStream_index)   >
0)
            {
                AVFS_SYSTEM_INFO::avfs_system_key_table_stream_info * stream_seek_table =
(*(this_system_info->avfs_system_key_table_streams2))[avfs_stream->AVStream_index];
                if(stream_seek_table &&
                    (((packet_header.packet_clock_pts -
stream_seek_table->pre_stream_key_packet_pts) >= seektable_keypacket_min_interval_pts) ||
                    (stream_seek_table->pre_stream_key_packet_pts == 0 &&
packet_header.packet_clock_pts == 0)))
                {
                    AVFS_SYSTEM_INFO::avfs_stream_key_table element;
```

```
                    stream_seek_table->pre_stream_key_packet_pts = element.pts_1 =
packet_header.packet_clock_pts;
                    element.key_file_offset_2 = this_system_info->property_info1.info4_filesize;
                    stream_seek_table->stream_packets_3.push_back(element);
                    stream_seek_table->stream_packet_number_2++;
                }
            }
        }

        //current packet - pre offset
        if(avfs_stream->stream_id_1.codec_id >= enAVFSCI_VIDEO &&
avfs_stream->stream_id_1.codec_id < enAVFSCI_AUDIO)
        {
            //Video Stream
            packet_header.pre_packet_offset_flag = 1;
            packet_header.pre_key_packet_pos = avfs_stream->pre_key_packet_file_pos;
        }
        else
        {
            //Others Stream
            packet_header.pre_packet_offset_flag = 0;
            packet_header.pre_key_packet_pos = this_system_info->pre_key_packet_file_pos;
        }

        //write packet header
        avio_write(s->pb,(const unsigned char *)&packet_header,sizeof(packet_header));
        this_system_info->property_info1.info4_filesize += sizeof(packet_header);

        //write packet data (frame)
        avio_write(s->pb,data,datasize);
        this_system_info->property_info1.info4_filesize += datasize;

        //count frame numbers
        avfs_stream->stream_total_packet_number++;
        this_system_info->property_info1.info3_framenumber++;

        //flush avio
        avio_flush(s->pb);
}
```

第五步：容器封装收尾

在容器封装流程中，最后由一个收尾工作调用 av_write_trailer 进行对容器格式写文件格式尾、回写头以及调整等等操作，这时就会调用容器封装插件的 write_trailer 接口实现具体工作，调用完后并没有结束流程操作，还会有 avio_close 和 avformat_free_context 等结束操作和释放资源。

```
///<  容器封装收尾函数
static int ff_avfs_muxer_write_trailer(AVFormatContext *s)
{
    //  获取私有核心对象句柄
    struct AVFastSearching_MuxerContext * h = (struct AVFastSearching_MuxerContext *)s->priv_data;
    AVIOContext * pb = s->pb;
    if(h)
    {
        char * str_release = NULL;
        int c_size = 0;
        if(pb)
        {
            //write the seek table frame
            if(h->this_system_info.avfs_system_key_table_stream_number1 > 0
                && h->this_system_info.avfs_system_key_table_streams2)
            {
                //size - avfs_system_key_table_stream_number1
                uint8_t * buffer = NULL;
                int buffersize = 0;
                buffersize += sizeof(h->this_system_info.avfs_system_key_table_stream_number1);

                //size - avfs_system_key_table_streams2
                c_size = h->this_system_info.avfs_system_key_table_streams2->size();
                if(c_size == h->this_system_info.avfs_system_key_table_stream_number1 &&
                    c_size > 0)
                {
                    //count size
                    std::map<int , AVFS_SYSTEM_INFO::avfs_system_key_table_stream_info
*>::iterator iter_pi;
                    iter_pi = h->this_system_info.avfs_system_key_table_streams2->begin();
                    for(int i = 0;((iter_pi !=
h->this_system_info.avfs_system_key_table_streams2->end()) && i< c_size);i++,iter_pi++)
                    {
                        if(iter_pi->second)
                        {
                            buffersize += sizeof(iter_pi->second->stream_id_1);
```

```
                                buffersize += sizeof(iter_pi->second->stream_packet_number_2);
                                buffersize +=
iter_pi->second->stream_packets_3.size()*sizeof(AVFS_SYSTEM_INFO::avfs_stream_key_table);
                        }
                    }

                    //write frame
                    int total_buffersize = 0;
                    buffer = new uint8_t[buffersize];
                    memcpy(buffer + total_buffersize,(unsigned char
*)&h->this_system_info.avfs_system_key_table_stream_number1,sizeof(h->this_system_info.avfs_system_
key_table_stream_number1));
                    total_buffersize +=
sizeof(h->this_system_info.avfs_system_key_table_stream_number1);
                    iter_pi = h->this_system_info.avfs_system_key_table_streams2->begin();
                    for(int i = 0;((iter_pi !=
h->this_system_info.avfs_system_key_table_streams2->end()) && i< c_size);i++,iter_pi++)
                    {
                        AVFS_SYSTEM_INFO::avfs_system_key_table_stream_info *
avfs_seek_table_stream = iter_pi->second;
                        if(avfs_seek_table_stream)
                        {
                            memcpy(buffer + total_buffersize,(unsigned char
*)&avfs_seek_table_stream->stream_id_1,sizeof(avfs_seek_table_stream->stream_id_1));
                            total_buffersize += sizeof(avfs_seek_table_stream->stream_id_1);
                            memcpy(buffer + total_buffersize,(unsigned char
*)&avfs_seek_table_stream->stream_packet_number_2,sizeof(avfs_seek_table_stream->stream_packet_nu
mber_2));
                            total_buffersize +=
sizeof(avfs_seek_table_stream->stream_packet_number_2);
                            for(int j = 0; j <
avfs_seek_table_stream->stream_packets_3.size();j++)
                            {
                                memcpy(buffer + total_buffersize,(unsigned char
*)&(avfs_seek_table_stream->stream_packets_3[j]),sizeof(AVFS_SYSTEM_INFO::avfs_stream_key_table
));
                                total_buffersize +=
sizeof(AVFS_SYSTEM_INFO::avfs_stream_key_table);
                            }
                        }
                    }
                    if(total_buffersize == buffersize)
                    {
                        //write layer frame header
```

```
                                AVFS_LIST_LAYER_HEADER packet_header = {0};
                                packet_header.ext_info = 0;
                                packet_header.keyframe = 0;
                                packet_header.encryp_option = 0;
                                packet_header.packet_index_or_clock_dts_flag = 0;
                                packet_header.packet_index_or_clock_dts = 0;
                                packet_header.packet_clock_pts = 0;
                                packet_header.stream_id.s_index = 0;
                                packet_header.stream_id.codec_id = enAVFSCI_System_SEEK_TABLE;
                                packet_header.current_packet_size = buffersize;

                                //write layer frame
                                h->this_system_info.property_info1.info5_frame_seektable_file_pos =
h->this_system_info.property_info1.info4_filesize;
                                avio_write(pb,(unsigned char *)&packet_header,sizeof(packet_header));
                                ADD_FILE_SIZE(sizeof(packet_header));
                                avio_write(pb,buffer,buffersize);
                                ADD_FILE_SIZE(buffersize);
                                h->this_system_info.property_info1.info3_framenumber++;
                            }
                            if(buffer){delete buffer;buffer=NULL;}
                        }
                    }
                }

                //rewrite media property
                h->this_system_info.property_info1.info2_duration = h->this_system_info.pre_packet_pts;
                if(pb->seekable)
                {
                    int err = avio_seek(pb,sizeof(avfs_format_header_marked),SEEK_SET);
                    if(err >= 0)
                    {avio_write(s->pb,(unsigned char
*)&h->this_system_info.property_info1,sizeof(h->this_system_info.property_info1));}
                }

                //flush
                avio_flush(s->pb);

                //release avfs system info
                ff_avfs_release_muxer_system_info(h->this_system_info);
            }
            return 0;
}
```

2) 容器解析器插件（Demuxer）

AVInputFormat 是容器封装对象（容器输入）的定义，是将已存在的数据包、数据流保存在一个实体的容器格式中，它和容器解析都使用 AVFormatContext 作为自己的数据上下文存储对象。容器解析和容器的封装流程当然是相互对应紧密结合的，在容器解析和封装的插件代码中没有写出版权保护和加密解密部分的处理以及其他敏感信息等。

容器格式解析比容器格式封装多出一个逆过程叫做容器适配（容器探测），这个主要是在不确定容器格式的具体格式下所要做的工作，这部分工作一般都是封装在框架库的实现当中，而且很多带框架的媒体库的实现机制基本都是一样的。

FFmpeg 库的容器解析适配工作也是如此，通过“**打分制**”进行最终的匹配工作，这个适配过程已经完全封装在 avformat_open_input 等接口实现中，当然也提供外部单独接口 av_probe_input_buffer、av_probe_input_format 等接口函数进行单独适配工作。

打分制的具体内容如下：

1. 满分是 100 分，如果插件给出 100 分表示这个文件内容完全匹配此容器插件的对应格式实现，匹配成功，匹配流程终止。
2. 例如：文件内容匹配为 100 分、文件扩展名匹配为 50 分、文件 URL 或协议分析 25 分等等进行计算。
3. 如果没有满分就会选择一个分数最高的进行匹配，如果匹配失败，表示所有容器都不识别，当然框架库都有优化和机制不会把一个未知内容让所有容器都匹配一遍这个是不愿见到的，这个也要考虑到效率问题。

第一步：定义插件核心句柄

```
///< 宏定义
#define AVFS_FORMAT_NAME                    "avfs"          //avfs文件容器

///<解析容器插件
AVInputFormat ff_avfs_demuxer = {
AVFS_FORMAT_NAME,           ///< 唯一名称表示符，可以通过av_find_input_format进行指定容器封装插件的名称
AVFS_FORMAT_NAME" is av-fast-searching-format.",     ///< 完整信息说明
AVFMT_NO_BYTE_SEEK | AVFMT_GENERIC_INDEX,        ///< 容器解析插件属性标示符
/**
 *  AVFMT_NO_BYTE_SEEK          不支持字节级索引功能
 *  AVFMT_GENERIC_INDEX         流ID值使用标准的索引ID值表示
 */
AVFS_FORMAT_NAME,       ///< 扩展名，可以在调用avformat_open_input的时候，通过URL进行自动适配，例如：” asf,wmv,wma”
NULL,                   ///< 参考的标准字节值数组，是用AVCodecTag结构进行定义
&ff_avfs_demuxer_class,     ///< AVFS容器解析类
NULL,                   ///< 框架内部使用
0,                      ///< 编解码器默认的容器解析的Codec ID
sizeof(AVFS_DEMUXER_CONTEXT),       ///< 核心私有句柄结构体大小
ff_avfs_demuxer_read_probe,     ///< read_probe接口，容器解析器适配
ff_avfs_demuxer_read_header,    ///< read_header接口，容器解析头部分
ff_avfs_demuxer_read_packet,    ///< write_packet接口，容器解析数据包
ff_avfs_demuxer_read_close,     ///< read_close接口，关闭容器解析器
ff_avfs_demuxer_read_seek,      ///< read_seek接口，容器解析索引
NULL,                   ///< read_timestamp接口，从字节流位置读取时间
NULL,                   ///< read_play接口，开始播放
NULL,                   ///< read_pause接口，暂停播放
NULL                    ///< read_seek2接口，根据时间戳范围索引
};

///< 核心句柄数据结构
typedef struct AVFastSearching_DemuxerContext{
    const AVClass * pclass;                             ///< FFmpeg系统默认
     uint64_t current_file_pos;
     AVFS_SYSTEM_INFO this_system_info;                 ///< 核心数据成员;
     AVFS_LIST_LAYER_HEADER current_packet_header;
     int32_t avfs_h264_mpegts_bitstreamfilter;
}AVFS_DEMUXER_CONTEXT;

///< 命令参数
```

```
#define AVFS_DEMUXER_OFFSET(x) offsetof(AVFastSearching_DemuxerContext, x)
#define DE AV_OPT_FLAG_DECODING_PARAM
static const AVOption ff_avfs_demuxer_options[] = {
    { "avfs_h264_mpegts_bitstreamfilter", "0 is default header,ohers is clear extradata", AVFS_DEMUXER_OFFSET(avfs_h264_mpegts_bitstreamfilter), AV_OPT_TYPE_INT, {0}, 0, INT_MAX, DE },
    { NULL },
};

///< AVFS容器解析插件类定义
static const AVClass ff_avfs_demuxer_class = {
    AVFS_FORMAT_NAME"_dec",             ///< 类名称
    av_default_item_name,               ///< 默认处理函数
    ff_avfs_demuxer_options,            ///< 参数
    LIBAVUTIL_VERSION_INT,              ///< 版本
};

///< 宏定义位置计算
#define ADD_FILE_POS(size) h->current_file_pos += size
```

第二步：注册插件

```
///< 插件注册函数
bool static_register_avfs_demuxer(){
bool isfind = false;
AVInputFormat * demuxer = NULL;
while(demuxer = av_iformat_next(demuxer))
{
    // 检查插件是否已经存在
    if(demuxer->name && !strcmp(demuxer->name,ff_avfs_demuxer.name))
    {
        av_log(&this_class,AV_LOG_INFO,"the %s format is already registered.\n",ff_avfs_demuxer.
name);
        isfind = true;
    }
}
if(isfind == false)
{
    // 未存在，注册插件
    av_register_input_format(&ff_avfs_demuxer);
    isfind = true;
    av_log(&this_class,AV_LOG_INFO,"register %s demuxer.\n",ff_avfs_demuxer.name);
}
return isfind;
}
```

第三步：打开输入容器

在打开容器解析器前有一个适配过程，框架会进行IO操作缓存一些数据，实际AVFormatContext对象已经生成，根据一系列的内部操作最终将数据交给AVInputFormat容器解析器的read_probe接口进行数据适配，适配成功才开始真正的数据处理过程。

```
///<  AVFS 容器适配
static int ff_avfs_demuxer_read_probe(AVProbeData *p)
{
    /* check file header */
    if(p->buf_size >= sizeof(avfs_format_header_marked) && p->buf)
    {
        if(!memcmp(p->buf,avfs_format_header_marked,sizeof(avfs_format_header_marked)))
            return AVPROBE_SCORE_MAX;    // 适配成功
    }
    return 0;
}
```

适配成功的容器解析器将会被使用，框架会调用其 read_header 接口进行容器格式的头解析，这些操作都是在 avformat_open_input 接口内实现调用的。

```
///< AVFS容器解析头
static int ff_avfs_demuxer_read_header(AVFormatContext *s)
{
    //init param
    struct AVFastSearching_DemuxerContext * h = (struct AVFastSearching_DemuxerContext
*)s->priv_data;
    int64_t pos = 0;
    int err = 0;

    //skip file marked
    pos = avio_skip(s->pb, sizeof(avfs_format_header_marked));
    ADD_FILE_POS(sizeof(avfs_format_header_marked));
    do
    {
        //read property info
        err = avio_read(s->pb,(unsigned char
*)&h->this_system_info.property_info1,sizeof(h->this_system_info.property_info1));
        if(err < 0){return err;}
        ADD_FILE_POS(sizeof(h->this_system_info.property_info1));
        if(s->pb->seekable &&
            h->this_system_info.property_info1.info4_filesize == 0)
        {
```

```
            int64_t filesize = avio_size(s->pb);
            if(filesize > 0)
            {h->this_system_info.property_info1.info4_filesize = filesize;}
        }
        if(h->this_system_info.property_info1.info2_duration > 0 &&
            h->this_system_info.property_info1.info2_duration != 0xFFFFFFFFFFFFFFFF &&
            h->this_system_info.property_info1.info4_filesize > 0 &&
            h->this_system_info.property_info1.info4_filesize != 0xFFFFFFFFFFFFFFFF)
        {
            s->duration =
(int64_t)((double)h->this_system_info.property_info1.info2_duration/10000000*AV_TIME_BASE);
            s->start_time = h->this_system_info.property_info1.info1_start_time;
            s->bit_rate =
h->this_system_info.property_info1.info4_filesize*8/((double)h->this_system_info.property_info1.info2_du
ration/10000000);
        }
        else{s->start_time = 0;}

        //read property media data
        uint16_t property_media_data_numbers = 0;
        if(h->this_system_info.property_media_data2)
        {h->this_system_info.property_media_data2 = new std::map<char *,char *>;}
        err = avio_read(s->pb,(unsigned char
*)&property_media_data_numbers,sizeof(property_media_data_numbers));
        if(err < 0){return err;}
        ADD_FILE_POS(sizeof(property_media_data_numbers));
        if(property_media_data_numbers > 0)
        {
            h->this_system_info.property_media_data2 = new std::map<char *,char *>;
            if(h->this_system_info.property_media_data2)
            {
                for(int i = 0;i<property_media_data_numbers;i++)
                {
                    uint16_t key_size = 0,value_size = 0;
                    char * key_buffer = "", * value_buffer = "";
                    key_size = avio_rl16(s->pb);
                    ADD_FILE_POS(sizeof(key_size));
                    if(key_size > 0)
                    {
                        key_buffer = (char *)malloc(key_size + 1);
                        if(key_buffer)
                        {
                            avio_read(s->pb,(unsigned char *)key_buffer,key_size);
                            key_buffer[key_size] = 0;
```

```
                    }
                    else
                    {key_buffer = "";}
                    ADD_FILE_POS(key_size);
                }
                value_size = avio_rl16(s->pb);
                ADD_FILE_POS(sizeof(value_size));
                if(value_size > 0)
                {
                    value_buffer = (char *)malloc(value_size + 1);
                    if(value_buffer)
                    {
                        avio_read(s->pb,(unsigned char *)value_buffer,value_size);
                        value_buffer[value_size] = 0;
                    }
                    else
                    {value_buffer = "";}
                    ADD_FILE_POS(value_size);
                }
                if(strlen(key_buffer) > 0)
                {
                    h->this_system_info.property_media_data2->insert(std::map<char *, char
*>::value_type(
                        strdup(key_buffer),
                        strdup(value_buffer)));
                    av_dict_set(&s->metadata, key_buffer, value_buffer, 0);
                }
            }
        }
    }

    //read streams number
    err = avio_read(s->pb,(unsigned char
*)&h->this_system_info.info_stream_number3,sizeof(h->this_system_info.info_stream_number3));
    if(err < 0){return err;}
    ADD_FILE_POS(sizeof(h->this_system_info.info_stream_number3));

    //read streams info header
    if( h->this_system_info.info_stream_number3 > 0)
    {
        h->this_system_info.info_streams4 = new
AVFS_SYSTEM_INFO::avfs_stream_info[h->this_system_info.info_stream_number3];

memset(h->this_system_info.info_streams4,0,sizeof(AVFS_SYSTEM_INFO::avfs_stream_info)*h->t
```

```
his_system_info.info_stream_number3);
                for(int i = 0;i < h->this_system_info.info_stream_number3;i++)
                {
                    //stream
                    AVFS_SYSTEM_INFO::avfs_stream_info * avfs_stream =
h->this_system_info.info_streams4 + i;
                    AVStream * stream = NULL;
                    enum AVCodecID codec_id = AV_CODEC_ID_NONE;

                    //read stream id
                    err = avio_read(s->pb,(unsigned char
*)&avfs_stream->stream_id_1,sizeof(avfs_stream->stream_id_1));
                    if(err < 0){return err;}
                    ADD_FILE_POS(sizeof(avfs_stream->stream_id_1));

                    //save stream id
                    switch(avfs_stream->stream_id_1.codec_id)
                    {
                    case enAVFSCI_H264:
                        {codec_id = AV_CODEC_ID_H264;}break;
                    case enAVFSCI_H265:
                        {codec_id = AV_CODEC_ID_H265;}break;
                    case enAVFSCI_VP8:
                        {codec_id = AV_CODEC_ID_VP8;}break;
                    case enAVFSCI_VP9:
                        {codec_id = AV_CODEC_ID_VP9;}break;
                    case enAVFSCI_MPEG4:
                        {codec_id = AV_CODEC_ID_MPEG4;}break;
                    case enAVFSCI_H263:
                        {codec_id = AV_CODEC_ID_H263;}break;
                    case enAVFSCI_RV10:
                        {codec_id = AV_CODEC_ID_RV10;}break;
                    case enAVFSCI_MJPEG:
                        {codec_id = AV_CODEC_ID_MJPEG;}break;
                    case enAVFSCI_LJPEG:
                        {codec_id = AV_CODEC_ID_LJPEG;}break;
                    case enAVFSCI_JPEG2000:
                        {codec_id = AV_CODEC_ID_JPEG2000;}break;
                    case enAVFSCI_PNG:
                        {codec_id = AV_CODEC_ID_PNG;}break;
                    case enAVFSCI_BMP:
                        {codec_id = AV_CODEC_ID_BMP;}break;
                    case enAVFSCI_TIFF:
                        {codec_id = AV_CODEC_ID_TIFF;}break;
```

```
                case enAVFSCI_RAWVIDEO:
                    {codec_id = AV_CODEC_ID_RAWVIDEO;}break;
                case enAVFSCI_MP2:
                    {codec_id = AV_CODEC_ID_MP2;}break;
                case enAVFSCI_MP3:
                    {codec_id = AV_CODEC_ID_MP3;}break;
                case enAVFSCI_AAC:
                    {codec_id = AV_CODEC_ID_AAC;}break;
                case enAVFSCI_AC3:
                    {codec_id = AV_CODEC_ID_AC3;}break;
                case enAVFSCI_DTS:
                    {codec_id = AV_CODEC_ID_DTS;}break;
                case enAVFSCI_ADPCM_G726:
                    {codec_id = AV_CODEC_ID_ADPCM_G726;}break;
                case enAVFSCI_AMR_NB:
                    {codec_id = AV_CODEC_ID_AMR_NB;}break;
                case enAVFSCI_AMR_WB:
                    {codec_id = AV_CODEC_ID_AMR_WB;}break;
                case enAVFSCI_DVD_SUBTITLE:
                    {codec_id = AV_CODEC_ID_DVD_SUBTITLE;}break;
                }
                if(codec_id == AV_CODEC_ID_NONE)
                {
                    av_log(s,AV_LOG_ERROR,"unknow codec id =
0x%X.\n",avfs_stream->stream_id_1.codec_id);
                    return AVERROR_INVALIDDATA;
                }
                stream = avformat_new_stream(s,NULL);
                stream->id = avfs_stream->stream_id_1.s_index;
                stream->codec->codec_id = codec_id;
                //stream->codec->codec_tag = avfs_stream->stream_id_1.codec_id;
                avpriv_set_pts_info(stream,64,1,10000000);
                stream->priv_data = avfs_stream;
                stream->sample_aspect_ratio.num = stream->codec->sample_aspect_ratio.num = 1;
                stream->sample_aspect_ratio.den = stream->codec->sample_aspect_ratio.den = 1;

                //save stream param
                if(avfs_stream->stream_id_1.codec_id >= enAVFSCI_VIDEO &&
                    avfs_stream->stream_id_1.codec_id < enAVFSCI_AUDIO)
                {
                    stream->codec->codec_type = AVMEDIA_TYPE_VIDEO;
                    //read video stream param
                    err = avio_read(s->pb,(unsigned char
*)&avfs_stream->video_param_2,sizeof(avfs_stream->video_param_2));
```

```
                if(err < 0){return err;}
                ADD_FILE_POS(sizeof(avfs_stream->video_param_2));
                stream->codec->width = avfs_stream->video_param_2.width_1;
                stream->codec->height = avfs_stream->video_param_2.height_2;
                stream->r_frame_rate.num = stream->avg_frame_rate.num =
stream->codec->time_base.num = avfs_stream->video_param_2.fps_numerator_3;
                stream->r_frame_rate.den = stream->avg_frame_rate.den =
stream->codec->time_base.den = avfs_stream->video_param_2.fps_denominator_4;
                switch(avfs_stream->video_param_2.color_space_5)
                {
                case enAVFSCP_YV12:{
                    stream->codec->pix_fmt = AV_PIX_FMT_YUV420P;
                    stream->codec->bits_per_coded_sample = 24;
                                        }break;
                case enAVFSCP_NV12:{
                    stream->codec->pix_fmt = AV_PIX_FMT_NV12;
                    stream->codec->bits_per_coded_sample = 24;
                                        }break;
                case enAVFSCP_YV16:{
                    stream->codec->pix_fmt = AV_PIX_FMT_YUV422P;
                    stream->codec->bits_per_coded_sample = 24;
                                        }break;
                case enAVFSCP_Y41P:{
                    stream->codec->pix_fmt = AV_PIX_FMT_YUV411P;
                    stream->codec->bits_per_coded_sample = 24;
                                        }break;
                case enAVFSCP_YUV444:{
                    stream->codec->pix_fmt = AV_PIX_FMT_YUV444P;
                    stream->codec->bits_per_coded_sample = 24;
                                          }break;
                case enAVFSCP_AYUV:{
                    stream->codec->pix_fmt = AV_PIX_FMT_YUVA444P;
                    stream->codec->bits_per_coded_sample = 32;
                                        }break;
                case enAVFSCP_RGB24:{
                    stream->codec->pix_fmt = AV_PIX_FMT_RGB24;
                    stream->codec->bits_per_coded_sample = 24;
                                          }break;
                case enAVFSCP_RGBA:{
                    stream->codec->pix_fmt = AV_PIX_FMT_RGBA;
                    stream->codec->bits_per_coded_sample = 32;
                                        }break;
                case enAVFSCP_YUV9:{
                    stream->codec->pix_fmt = AV_PIX_FMT_YUV410P;
```

```
                    stream->codec->bits_per_coded_sample = 24;
                                }break;
                case enAVFSCP_GRAY:{
                    stream->codec->pix_fmt = AV_PIX_FMT_GRAY8;
                    stream->codec->bits_per_coded_sample = 8;
                                }break;
                }
            }
            else if(avfs_stream->stream_id_1.codec_id >= enAVFSCI_AUDIO &&
                avfs_stream->stream_id_1.codec_id < enAVFSCI_SUBTITLE)
            {
                stream->codec->codec_type = AVMEDIA_TYPE_AUDIO;
                //read audio stream param
                err = avio_read(s->pb,(unsigned char
*)&avfs_stream->audio_param_2,sizeof(avfs_stream->audio_param_2));
                if(err < 0){return err;}
                ADD_FILE_POS(sizeof(avfs_stream->audio_param_2));
                stream->codec->sample_rate =
avfs_stream->audio_param_2.a_sample_rate_1*100;
                switch(avfs_stream->audio_param_2.a_sample_format_2)
                {
                case enAVFSSF_BYTE:{
                    stream->codec->sample_fmt = AV_SAMPLE_FMT_U8P;
                    stream->codec->bits_per_coded_sample = 8;
                                }break;
                case enAVFSSF_SHORT:{
                    stream->codec->sample_fmt = AV_SAMPLE_FMT_S16P;
                    stream->codec->bits_per_coded_sample = 16;
                                  }break;
                case enAVFSSF_INT:{
                    stream->codec->sample_fmt = AV_SAMPLE_FMT_S32P;
                    stream->codec->bits_per_coded_sample = 32;
                                 }break;
                case enAVFSSF_FLOAT:{
                    stream->codec->sample_fmt = AV_SAMPLE_FMT_FLTP;
                    stream->codec->bits_per_coded_sample = 32;
                                  }break;
                case enAVFSSF_DOUBLE:{
                    stream->codec->sample_fmt = AV_SAMPLE_FMT_DBLP;
                    stream->codec->bits_per_coded_sample = 64;
                                   }break;
                }
                switch(avfs_stream->audio_param_2.a_channel_layout_3)
                {
```

```
                    case enAVFSCL_MONO:{stream->codec->channels =
1;stream->codec->channel_layout = AV_CH_LAYOUT_MONO;}break;
                    case enAVFSCL_STEREO:{stream->codec->channels =
2;stream->codec->channel_layout = AV_CH_LAYOUT_STEREO;}break;
                    case enAVFSCL_5SURROUND_STEREO:{stream->codec->channels =
5;stream->codec->channel_layout = AV_CH_LAYOUT_5POINT0;}break;
                    case enAVFSCL_6SURROUND_STEREO:{stream->codec->channels =
6;stream->codec->channel_layout = AV_CH_LAYOUT_5POINT1;}break;
                    }
                }
                else if(avfs_stream->stream_id_1.codec_id >= enAVFSCI_SUBTITLE &&
                    avfs_stream->stream_id_1.codec_id < enAVFSCI_OTHERS)
                {
                    stream->codec->codec_type = AVMEDIA_TYPE_SUBTITLE;
                    //read subtitle stream param
                    err = avio_read(s->pb,(unsigned char
*)&avfs_stream->subtitle_param_2,sizeof(avfs_stream->subtitle_param_2));
                    if(err < 0){return err;}
                    ADD_FILE_POS(sizeof(avfs_stream->subtitle_param_2));
                }
                else
                {av_log(s,AV_LOG_WARNING,"this is unknow stream[index=%d] to
save.\n",avfs_stream->stream_id_1.s_index);}

                //read stream general param
                err = avio_read(s->pb,(unsigned char
*)&avfs_stream->general_param_3.bit_rate_1,sizeof(avfs_stream->general_param_3.bit_rate_1));
                if(err < 0){return err;}
                ADD_FILE_POS(sizeof(avfs_stream->general_param_3.bit_rate_1));

                //save stream general param
                stream->codec->bit_rate = avfs_stream->general_param_3.bit_rate_1;
                err = avio_read(s->pb,(unsigned char
*)&avfs_stream->general_param_3.extra_data_size_2,sizeof(avfs_stream->general_param_3.extra_data_siz
e_2));
                if(err < 0){return err;}
                ADD_FILE_POS(sizeof(avfs_stream->general_param_3.extra_data_size_2));
                if(avfs_stream->general_param_3.extra_data_size_2 > 0)
                {
                    avfs_stream->general_param_3.extra_data_3 = new
uint8_t[avfs_stream->general_param_3.extra_data_size_2];
                    err = avio_read(s->pb,(unsigned char
*)avfs_stream->general_param_3.extra_data_3,avfs_stream->general_param_3.extra_data_size_2);
                    if(err < 0){return err;}
```

```
                    ADD_FILE_POS(avfs_stream->general_param_3.extra_data_size_2);
                    if(h->avfs_h264_mpegts_bitstreamfilter && stream->codec->codec_id ==
AV_CODEC_ID_H264)
                    {
                        stream->codec->extradata = 0;
                        stream->codec->extradata_size = 0;
                    }
                    else
                    {
                        stream->codec->extradata = (uint8_t
*)av_mallocz(avfs_stream->general_param_3.extra_data_size_2 +
FF_INPUT_BUFFER_PADDING_SIZE);

    memcpy(stream->codec->extradata,avfs_stream->general_param_3.extra_data_3,avfs_stream->general
_param_3.extra_data_size_2);
                        stream->codec->extradata_size =
avfs_stream->general_param_3.extra_data_size_2;
                    }

                }
                uint16_t stream_media_data_numbers = 0;
                err = avio_read(s->pb,(unsigned char
*)&stream_media_data_numbers,sizeof(stream_media_data_numbers));
                if(err < 0){return err;}
                ADD_FILE_POS(sizeof(stream_media_data_numbers));
                if(stream_media_data_numbers > 0)
                {
                    avfs_stream->general_param_3.media_data_4 = new std::map<char *,char *>;
                    for(int i = 0;i < stream_media_data_numbers;i++)
                    {
                        uint16_t key_size = 0,value_size = 0;
                        char * key_buffer = "", * value_buffer = "";
                        key_size = avio_rl16(s->pb);
                        ADD_FILE_POS(sizeof(key_size));
                        if(key_size > 0)
                        {
                            key_buffer = (char *)malloc(key_size + 1);
                            if(key_buffer)
                            {
                                avio_read(s->pb,(unsigned char *)key_buffer,key_size);
                                key_buffer[key_size] = 0;
                            }
                            else
                            {key_buffer = "";}
```

```
                                    ADD_FILE_POS(key_size);
                                }
                                value_size = avio_rl16(s->pb);
                                ADD_FILE_POS(sizeof(value_size));
                                if(value_size > 0)
                                {
                                    value_buffer = (char *)malloc(value_size + 1);
                                    if(value_buffer)
                                    {
                                        avio_read(s->pb,(unsigned char *)value_buffer,value_size);
                                        value_buffer[value_size] = 0;
                                    }
                                    else
                                    {value_buffer = "";}
                                    ADD_FILE_POS(value_size);
                                }
                                if(strlen(key_buffer) > 0)
                                {
                                    avfs_stream->general_param_3.media_data_4->insert(std::map<char
*, char *>::value_type(
                                        strdup(key_buffer),
                                        strdup(value_buffer)));
                                    av_dict_set(&stream->metadata, key_buffer, value_buffer, 0);
                                }
                            }
                        }
                    }
                }
            }while(false);

            //read the seek table frame
            if(h->this_system_info.property_info1.info5_frame_seektable_file_pos > 0)
            {
                uint8_t * seektable_buffer = NULL;
                int seektable_buffer_size = 0;
                if(h->this_system_info.property_info1.info5_frame_seektable_file_pos == h->current_file_pos)
                {
                    err = avio_read(s->pb,(unsigned char
*)&h->current_packet_header,sizeof(h->current_packet_header));
                    if(err < 0){return err;}
                    ADD_FILE_POS(sizeof(h->current_packet_header));
                    if(h->current_packet_header.current_packet_size > 0 &&
                        h->current_packet_header.stream_id.codec_id == enAVFSCI_System_SEEK_TABLE
&&
```

```
                    h->current_packet_header.stream_id.s_index == 0)
                {
                    seektable_buffer = new uint8_t[h->current_packet_header.current_packet_size];
                    seektable_buffer_size = h->current_packet_header.current_packet_size;
                    avio_read(s->pb,seektable_buffer,seektable_buffer_size);
                    if(err < 0){
                        if(seektable_buffer){delete seektable_buffer;seektable_buffer=NULL;}
                        return err;
                    }
                    ADD_FILE_POS(sizeof(h->current_packet_header.current_packet_size));
                    err =
ff_avfs_demuxer_read_seek_table(h->this_system_info,seektable_buffer,seektable_buffer_size);
                    if(err < 0){
                        if(seektable_buffer){delete seektable_buffer;seektable_buffer=NULL;}
                        return err;
                    }
                }
            }
            else if(s->pb && s->pb->seekable)
            {
                err =
avio_seek(s->pb,h->this_system_info.property_info1.info5_frame_seektable_file_pos,SEEK_SET);
                if(err < 0){return err;}

                err = avio_read(s->pb,(unsigned char
*)&h->current_packet_header,sizeof(h->current_packet_header));
                if(err < 0){return err;}
                if(h->current_packet_header.current_packet_size > 0 &&
                    h->current_packet_header.stream_id.codec_id == enAVFSCI_System_SEEK_TABLE
&&
                    h->current_packet_header.stream_id.s_index == 0)
                {
                    seektable_buffer = new uint8_t[h->current_packet_header.current_packet_size];
                    seektable_buffer_size = h->current_packet_header.current_packet_size;
                    avio_read(s->pb,seektable_buffer,seektable_buffer_size);
                    if(err < 0){
                        if(seektable_buffer){delete seektable_buffer;seektable_buffer=NULL;}
                        return err;
                    }
                    err =
ff_avfs_demuxer_read_seek_table(h->this_system_info,seektable_buffer,seektable_buffer_size);
                    if(err < 0){
                        if(seektable_buffer){delete seektable_buffer;seektable_buffer=NULL;}
                        return err;
```

```
                    }
                }

                err = avio_seek(s->pb,h->current_file_pos,SEEK_SET);
                if(err < 0){return err;}
            }
            if(seektable_buffer)
            {delete seektable_buffer;seektable_buffer=NULL;}
        }

        //invalidate data
        for(int i = 0;i<s->nb_streams;i++)
        {
            for(int j = i+1;j<s->nb_streams;j++)
            {
                if(s->streams[i]->id == s->streams[j]->id)
                {
                    av_log(s,AV_LOG_ERROR,"is the same stream id=0x%X.\n",s->streams[i]->id);
                    return AVERROR_INVALIDDATA;
                }
            }
        }
        if(err > 0){err = 0;}
        return err;
}
```

第四步：解析数据包

完成的头和 Codec 分析之后，开始调用 av_read_frame 进行数据包的读取，这时候框架是完全感受不到容器的具体格式的，数据包读取的大小，数据包怎么读，或是怎么索引完全由容器的插件控制，这就会调用 AVInputFormat 容器解析插件的 read_packet 接口完成数据包读取的工作。

```
///< AVFS容器解析数据包读取
static int ff_avfs_demuxer_read_packet(AVFormatContext *s, AVPacket *pkt)
{
	//init param
	struct AVFastSearching_DemuxerContext * h = (struct AVFastSearching_DemuxerContext
*)s->priv_data;
	if(pkt && pkt->priv)
	{pkt->priv = 0;}

	//find packet
	bool find_packet = false;
	int err = 0;
beginread:
	//read packet header
	err = avio_read(s->pb,(unsigned char
*)&h->current_packet_header,sizeof(h->current_packet_header));
	if(err < 0)
	{return err;}	// AVERROR(EAGAIN)	AVERROR_EOF
	ADD_FILE_POS(sizeof(h->current_packet_header));

	//read packet data
	if(h->current_packet_header.current_packet_size > 0)
	{
		for(int i = 0;i<s->nb_streams;i++)
		{
			AVStream * stream = s->streams[i];
			if(stream->id == h->current_packet_header.stream_id.s_index)
			{
				if(stream->priv_data)
				{
					AVFS_SYSTEM_INFO::avfs_stream_info * avfs_stream =
(AVFS_SYSTEM_INFO::avfs_stream_info *)stream->priv_data;
					if(avfs_stream->stream_id_1.s_index ==
h->current_packet_header.stream_id.s_index)
					{
						err =
```

```
av_get_packet(s->pb,pkt,h->current_packet_header.current_packet_size);
                                    if(err < 0)
                                    {return err;}
                                    if(pkt->data)
                                    {
                                        ADD_FILE_POS(h->current_packet_header.current_packet_size);
                                        if(err != h->current_packet_header.current_packet_size)
                                        {
                                            av_log(s,AV_LOG_ERROR,"need more packet size = %d
Byte.\n",h->current_packet_header.current_packet_size);
                                            return AVERROR_INVALIDDATA;
                                        }
                                        pkt->flags = (h->current_packet_header.keyframe?1:0);
                                        pkt->pts = h->current_packet_header.packet_clock_pts;
                                        if(h->current_packet_header.packet_index_or_clock_dts_flag)
                                        {pkt->dts = h->current_packet_header.packet_index_or_clock_dts;}
                                        else
                                        {
                                            if(avfs_stream->video_param_2.fps_numerator_3)
                                            {pkt->dts =
((double)h->current_packet_header.packet_index_or_clock_dts *
avfs_stream->video_param_2.fps_denominator_4 / avfs_stream->video_param_2.fps_numerator_3 *
10000000);}
                                            else
                                            {pkt->dts = 0;}
                                        }
                                        pkt->stream_index = i;
                                        h->this_system_info.pre_packet_pts = avfs_stream->pre_packet_pts =
pkt->pts;
                                        h->this_system_info.pre_packet_file_pos =
avfs_stream->pre_packet_file_pos = h->current_file_pos - sizeof(h->current_packet_header) -
h->current_packet_header.current_packet_size;
                                        if(h->current_packet_header.keyframe)
                                        {
                                            avfs_stream->pre_key_packet_pts =
avfs_stream->pre_packet_pts;
                                            avfs_stream->pre_key_packet_file_pos =
avfs_stream->pre_packet_file_pos;
                                            if(avfs_stream->stream_id_1.codec_id >= enAVFSCI_VIDEO
&& avfs_stream->stream_id_1.codec_id < enAVFSCI_AUDIO)
                                            {
                                                h->this_system_info.pre_key_packet_pts =
avfs_stream->pre_packet_pts;
                                                h->this_system_info.pre_key_packet_file_pos =
```

```
avfs_stream->pre_packet_file_pos;
                                        }
                                    }
                                    if(h->avfs_h264_mpegts_bitstreamfilter &&
stream->codec->codec_id == AV_CODEC_ID_H264)
                                    {
                                        if(pkt->data && pkt->size > 5 &&
                                            (pkt->data[0] != 0x00 || pkt->data[1] != 0x00 ||
pkt->data[2] != 0x00 || pkt->data[3] != 0x01))
                                        {
                                            //memcpy
                                            uint8_t * data = pkt->data;
                                            int size = pkt->size;
                                            pkt->data = (uint8_t *)av_mallocz(pkt->size +
FF_INPUT_BUFFER_PADDING_SIZE + 4 + avfs_stream->general_param_3.extra_data_size_2);

                                            //00 00 00 01
                                            if(pkt->data[0] == 0x00 && pkt->data[1] == 0x00 &&
pkt->data[2] == 0x01)
                                            {
                                                pkt->data[0] = 0x00;
                                                pkt->size = 1;
                                            }
                                            else
                                            {
                                                pkt->data[0] = 0x00;
                                                pkt->data[1] = 0x00;
                                                pkt->data[2] = 0x00;
                                                pkt->data[3] = 0x01;
                                                pkt->size = 4;
                                            }

                                            //first sei
                                            memcpy(&pkt->data[pkt->size],data,size);
                                            pkt->size += size;
                                            av_freep(&data);

                                            //sps pps
                                            if(avfs_stream->general_param_3.extra_data_size_2 > 0
&&
                                                avfs_stream->general_param_3.extra_data_3)
                                            {

    memcpy(&pkt->data[pkt->size],avfs_stream->general_param_3.extra_data_3,avfs_stream->general_pa
```

```
ram_3.extra_data_size_2);
                                                    pkt->size +=
avfs_stream->general_param_3.extra_data_size_2;
                                                }

                                                //bitstreamfilter
                                                h->avfs_h264_mpegts_bitstreamfilter = 0;
                                            }
                                        }
                                    }
                                    find_packet = true;
                                }
                            }
                            break;
                        }
                    }
                }

                if(find_packet == false)
                {
                    av_log(s,AV_LOG_WARNING,"Demuxer avfs packet is 0.\n");
                    goto beginread;
                }
                return 0;
}
```

第五步：容器索引

如果你看一个视频想从第 20 分钟后观看，因为 20 分钟前的视频你已经看过了，所以你在播放器上进度条的工具拖到 20 分钟观看视频，这时候就需要索引功能，跳到指定位置进行观看。

容器的索引功能属于基本功能之一，在 FFmpeg 的接口主要是 av_seek_frame 和 avformat_seek_file 进行实现，这些接口就会调用 AVInputFormat 容器解析插件的 read_seek 接口实现对应的功能。

```
///<  容器解析索引
//下面是 AVFS 容器解析插件实现的索引功能，这里只实现了向前的时间戳的帧级索引。
static int ff_avfs_demuxer_read_seek(AVFormatContext *s, int stream_index,int64_t ts, int flags)
{
    // init param
    struct AVFastSearching_DemuxerContext * h = (struct AVFastSearching_DemuxerContext
*)s->priv_data;
    if(h && stream_index < s->nb_streams && s->pb)
    {
        //init
        int err = 0;
        if(!s->pb->seekable)
        {av_log(s,AV_LOG_ERROR,"ff_avfs_demuxer_read_seek is unable seek action.\n");return
AVERROR_INVALIDDATA;}
        AVStream * stream = s->streams[stream_index];
        AVFS_SYSTEM_INFO::avfs_stream_info * avfs_stream = NULL;
        for(int i = 0;i<h->this_system_info.info_stream_number3;i++)
        {
            if((h->this_system_info.info_streams4 + i)->stream_id_1.s_index == stream->id)
            {avfs_stream = h->this_system_info.info_streams4 + i;break;}
        }

        //Seek Action
        if(stream && avfs_stream)
        {
            uint64_t left_pts = ts,left_pos = h->current_file_pos,right_pts = 0,right_pos = 0;

            //ts  向左面找
            if(flags & AVSEEK_FLAG_BACKWARD)
            {
                if(avfs_stream->seektable && avfs_stream->seektable->stream_packets_3.size() > 0)
                {
                    int c_size = avfs_stream->seektable->stream_packets_3.size();
```

```
                        int index = c_size - 1;
                        for(int i = 0; i< c_size; i++)
                        {
                            if(avfs_stream->seektable->stream_packets_3[i].pts_1 > ts)
                            {index = i;break;}
                        }
                        if(index < 0){index = 0;}

                        err =
avio_seek(s->pb,avfs_stream->seektable->stream_packets_3[index].key_file_offset_2,SEEK_SET);
                        if(err < 0){return err;}

                        h->current_file_pos =
avfs_stream->seektable->stream_packets_3[index].key_file_offset_2;
                    }
                    else
                    {
                        av_log(s,AV_LOG_WARNING,"seek backword frame is not exist seektable");
                        if(h->this_system_info.pre_key_packet_file_pos > 0)
                        {
                            err =
avio_seek(s->pb,h->this_system_info.pre_key_packet_file_pos,SEEK_SET);
                            if(err < 0){return err;}
                            left_pos = h->current_file_pos =
h->this_system_info.pre_key_packet_file_pos;
                        }
                    }
                }
                //ts  向右面找
                else
                {
                    av_log(s,AV_LOG_WARNING,"not support forword key frame");
                }
            }
        }
        return 0;
}
```

第六步：关闭输入容器

容器关闭是调用 avformat_close_input 函数进行文件关闭和释放数据资源的，库也会自动调用 AVInputFormat 的 read_close 接口进行相应的内部资源释放和文件关闭。

```
///< 关闭容器解析器
static int ff_avfs_demuxer_read_close(AVFormatContext *s)
{
    struct AVFastSearching_DemuxerContext * h = (struct AVFastSearching_DemuxerContext
*)s->priv_data;
    if(h)
    {
        //release sttream
        if(s->metadata)
        {av_dict_free(&s->metadata);}
        for(int i = 0;i < s->nb_streams;i++)
        {
            AVStream * stream = s->streams[i];
            stream->priv_data = NULL;
            if(stream && stream->codec)
            {
                if(stream->codec->extradata)
                {av_freep(&stream->codec->extradata);}
                stream->codec->extradata = NULL;
                stream->codec->extradata_size = 0;
            }
            if(stream->metadata)
            {av_dict_free(&stream->metadata);}
        }

        //release avfs system info
        ff_avfs_release_demuxer_system_info(h->this_system_info);
    }
    return 0;
}
```

3. 协议插件开发

通过容器插件的开发介绍已经看到了FFmpeg库的IO层操作函数的使用例子，前面章节也讲到了IO层函数一个大致的调用流程。实际支撑IO层的是一堆协议对象，即【URLProtocol】数据结构，他们是最底层IO操作的实际支持体。

下面一段代码是枚举协议层对象信息的功能，是作者的开源小工具 ffparse 的源码一部分。

```
URLProtocol * pFileControl = NULL;
for(int i = 0;avio_enum_protocols((void **)&pFileControl,1);i++)
{
	if(pFileControl->name)
	{
		m_CSecondListBox.InsertString(numSecond,pFileControl->name);
		vec_SecondIndex.push_back(i);
		numSecond++;
	}
}
```

一些注意事项：

1. 协议对象的注册，是通过调用 ffurl_register_protocol 函数实现。
2. 协议对象的索引（Seek）功能，可以从 IO 层的操作函数上可以看到 avio_seek 接口的协议操作。
3. 一些设计思想是源于 C 运行时库的 File 文件操作函数和 Socket 函数而来。
4. 协议对象的区分是通过 URL 的路径进行区分的。
5. 注意 avio_alloc_context 函数也可以对具体的 IO 对象实现针对性处理。

下面协议的对象以 Windows 版本的 pipe 就行案例分析，这对象也是属于 StrongFFplugin 内部实现的一部分，作者也共享出来其源码。

第一步：协议对象定义

```
///< 宏定义
#define PIPE_PROTOCOL_NAME                    "pipe"                //pipe命名管道协议

///< 协议层对象定义
URLProtocol ff_pipe_protocol = {
PIPE_PROTOCOL_NAME,     ///< 唯一名称表示符，可以通过URL协议头进行区分
pipe_open,              ///< url_open接口，  IO层打开操作（avio_open）
NULL,                   ///< url_open2接口，IO层打开操作（avio_open2）
pipe_read,              ///< url_read接口，IO层读操作
pipe_write,             ///< url_write接口，IO层写操作
```

```
NULL,                   ///< url_seek接口，IO层索引操作
pipe_close,             ///< url_close接口，IO层关闭操作
NULL,                   ///<  框架内部使用
NULL,                   ///< url_read_pause接口，IO层暂停操作
NULL,                   ///< url_read_seek接口，IO层只读索引操作
NULL,                   ///< url_get_file_handle接口，IO层获取底层操作句柄，例如ffserver会使用到
NULL,                   ///< url_get_multi_file_handle接口，IO层获取句柄
NULL,                   ///< url_shutdown接口，IO层Socket关闭操作
sizeof(FileContext),    ///<  核心私有句柄结构体大小
&ff_pipe_class,         ///< pipe协议对象类
NULL,                   ///<  标示符
pipe_check              ///< url_check接口，IO层有效性检查
};

#define WIN_PIPE_BUFFER_SIZE (1024*1024)       // Window管道1Mbyte缓存

///<  核心句柄数据结构
typedef struct PipeContext {
    const AVClass *pclass;          ///< FFmpeg系统默认
     HANDLE hPipe;                  ///< pipe系统句柄
     int open_flags;                ///<  读写标示符
     char * file_name;              ///< URL路径
     char * pipe_addr_ip;           ///< pipe IP地址或主机名
} FileContext;

///<  命令参数
#define PIPE_PROTOCOL_OFFSET(x) offsetof(FileContext, x)
#define E AV_OPT_FLAG_ENCODING_PARAM
static const char * localhost = ".";
static const AVOption ff_pipe_protocol_options[] = {
    { "pipe_addr_ip", "is ip addr string, . is localhost", PIPE_PROTOCOL_OFFSET(pipe_addr_ip),
AV_OPT_TYPE_STRING,    {0},        0, 0, E },
    { NULL },
};

///< pipe协议对象类定义
static const AVClass ff_pipe_class = {
    PIPE_PROTOCOL_NAME" write is Block,read is Immediately", ///<  类名称
    av_default_item_name,               ///<  默认处理函数
    ff_pipe_protocol_options,           ///<  参数
    LIBAVUTIL_VERSION_INT,          ///<  版本
};
```

第二步：0pen 接口实现

```
///<  open 接口实现
static int pipe_open(URLContext *h, const char *filename, int flags)
{
	FileContext * p = (FileContext *)h->priv_data;
	if(filename == NULL){return AVERROR(2);}
	try{
		//字符串组装
		int url_index = 0;
		p->file_name = new char[strlen(filename)+100];
		strcpy(p->file_name,"\\\\");
		url_index += strlen("\\\\");
		if(p->pipe_addr_ip && strlen(p->pipe_addr_ip) > 0 && strlen(p->pipe_addr_ip) < 16)
		{
			strcpy(p->file_name+url_index,p->pipe_addr_ip);
			url_index += strlen(p->pipe_addr_ip);
		}
		else
		{
			strcpy(p->file_name+url_index,".");
			url_index += strlen(".");
		}
		strcpy(p->file_name+url_index,"\\pipe\\");
		url_index += strlen("\\pipe\\");
		strcpy(p->file_name + url_index,filename + strlen("pipe:"));
		url_index = 0;
		p->hPipe = NULL;

		//创建管道
		if(AVIO_FLAG_WRITE & flags)
		{
			//PIPE_ACCESS_DUPLEX		双向模式,服务器进程和客户端进程都可以从管道读取数据和向管道中写入数据。
			//PIPE_ACCESS_INBOUND		服务器端就只能读取数据，而客户端就只能向管道中写入数据。
			//PIPE_ACCESS_OUTBOUND		服务器端就只能写入数据，而客户端就只能从管道中读取数据。
			//写模式 -- 输出文件
			p->hPipe = ::CreateNamedPipeA(p->file_name,

	PIPE_ACCESS_OUTBOUND|FILE_FLAG_WRITE_THROUGH,//PIPE_ACCESS_INBOUND
			PIPE_TYPE_BYTE|PIPE_READMODE_BYTE|PIPE_WAIT,
```

```
                1,//PIPE_UNLIMITED_INSTANCES
                WIN_PIPE_BUFFER_SIZE,WIN_PIPE_BUFFER_SIZE,0,// client time-out
                NULL);                  // default security attribute
                if(INVALID_HANDLE_VALUE == p->hPipe)
                {
                    av_log(h,AV_LOG_ERROR,"CreateNamedPipe [%s] failed
err=0x%X.\n",p->file_name,GetLastError());
                    if(p->file_name){delete p->file_name;p->file_name=NULL;}
                    p->hPipe = NULL;
                    return AVERROR(22);
                }

                //Open Pipe
                av_log(h,AV_LOG_INFO,"Wait ConnectNamedPipe [%s] .\n",p->file_name);
                if(ConnectNamedPipe(p->hPipe, NULL) == FALSE)
                {
                    av_log(h,AV_LOG_ERROR,"ConnectNamedPipe failed
err=0x%X.\n",GetLastError());
                    if(p->file_name){delete p->file_name;p->file_name=NULL;}
                    ::CloseHandle(p->hPipe);p->hPipe = NULL;
                    return AVERROR(22);
                }
            }
            else if(AVIO_FLAG_READ & flags)
            {
                p->hPipe = ::CreateFileA(p->file_name,GENERIC_READ,FILE_SHARE_READ,NULL,

    OPEN_EXISTING,FILE_ATTRIBUTE_READONLY/*FILE_ATTRIBUTE_NORMAL*/,NULL);//|
FILE_SHARE_WRITE
                if(p->hPipe == INVALID_HANDLE_VALUE)
                {
                    av_log(h,AV_LOG_ERROR,"CreateFile %s Failed GetLastError
= %d.\n",p->file_name,GetLastError());
                    if(p->file_name){delete p->file_name;p->file_name=NULL;}
                    ::CloseHandle(p->hPipe);p->hPipe = NULL;
                    return AVERROR(22);
                }
            }
            else
            {return AVERROR(88);}

            //创建管道成功
            p->open_flags = flags;
            av_log(h,AV_LOG_INFO,"ConnectNamedPipe (%s%s) Success [%s] .\n",
```

```
                (p->open_flags & AVIO_FLAG_READ)?"r":"",
                (p->open_flags & AVIO_FLAG_WRITE)?"w":"",
                p->file_name);
    }
    catch(...)
    {return AVERROR(88);}
    return 0;
}
```

第三步：Close 接口实现

```
///<  Close 接口实现
static int pipe_close(URLContext *h)
{
    FileContext * p = (FileContext *)h->priv_data;
    if(p->hPipe)
    {
        if(p->open_flags & AVIO_FLAG_WRITE)
        {::DisconnectNamedPipe(p->hPipe);}
        ::CloseHandle(p->hPipe);
        p->hPipe=NULL;
        p->open_flags = 0;
        if(p->file_name)
        {delete p->file_name;p->file_name=NULL;}
    }
    return 0;
}
```

第四步：Read 接口实现

```
///<  Read 接口实现
static int pipe_read(URLContext *h, unsigned char *buf, int size)
{
    FileContext * p = (FileContext *)h->priv_data;
    if(NULL == p->hPipe){return AVERROR(1);}
    DWORD cbBytesRead = 0;
    BOOL fSuccess = ReadFile(p->hPipe,buf,size,&cbBytesRead,NULL);
    if(fSuccess == false)
    {av_log(h,AV_LOG_ERROR,"fwrite error=0x%X.\n",GetLastError());return AVERROR(22);}
    return (int)cbBytesRead;
}
```

第五步：Write 接口实现

```
///<  Write 接口实现
static int pipe_write(URLContext *h, const unsigned char *buf, int size)
{
      FileContext * p = (FileContext *)h->priv_data;
       if(NULL == p->hPipe){return AVERROR(1);}
       DWORD cbWritten = 0;
       BOOL fSuccess = WriteFile(p->hPipe,buf,size,&cbWritten,NULL);
       if(fSuccess == false || cbWritten != size)
       {av_log(h,AV_LOG_ERROR,"fwrite write=%d written=%d
error=0x%X.\n",size,cbWritten,GetLastError());return AVERROR(22);}
       return (int)cbWritten;
}
```

第六步：Check 接口实现

```
///<  Check 接口实现
static int pipe_check(URLContext *h, int mask)
{
       FileContext * p = (FileContext *)h->priv_data;
       if(NULL == p->hPipe)
       {
              av_log(h,AV_LOG_WARNING,"pipe warning to check state %s",h->filename);
              struct _stat64 st;
              int ret = _stat64(h->filename, &st);
              if (ret < 0)
                     return AVERROR(errno);

              ret |= (st.st_mode&_S_IREAD ? mask&AVIO_FLAG_READ    : 0);
              ret |= (st.st_mode&_S_IWRITE ? mask&AVIO_FLAG_WRITE : 0);

              return ret;
       }
       else
       {return p->open_flags;}
}
```

4. 音视频滤器插件开发

音视频滤器的开发不再把全部代码粘贴出来，而是以主要接口讲解为主,AVFilter 仍属于发展期，而有些机制还在优化当中，AVFilter 的思想和结构设计在应用开发部分已经有所讲解，这里不再重复。

关于 AVFilter 类型是音频滤器还是视频滤器，主要看其 pin 接口（AVFilterPad）的类型和具体的功能意义，例如 showwaves 滤器类型即是音频也是视频，其主要功能是分析音频数据绘画出视频波形显示出来。

关于 AVFilter 输入输出的数量可以是多个，例如 amix 混音滤器，它的主要功能是将多个音频流混合成一个，好像是都在一个场景当中，也有其他的拆分的滤器有一个输入多个输出，或是没有输入也有输出的情况。

关于 Filter 思想不应该局限于单台机器，例如 FFmpeg 转码实现工具也可以看做是 Filter Graphic 中的一个 Filter 处理器，而播放器和流服务器也一样，所有的对媒体中的容器格式和协议就是 pin。

Filter 结构对于性能设计要求很高，这个是可以通过经验弥补的。而主要的缺陷和硬伤是 pin 数据接口的具体定义，接口是实际数据传输的纽带、是沟通的协议，由于理论上 Filter 可以任意功能扩展，所以怎么能让接口适应未知的数据类型和需求是很困难的，这就会出现 Filter 结构设计的弊端，这个是不一定可以通过经验就能解决的问题。截止目前为止作者了解到 DShow、FFM、AVFilter、MPlayer、VLC 等等实用到 Filter 架构的程序都遇到了这个问题，而且出现了问题基本很难修改，因为需要修改的太多，都是通过补丁的形式存在，让开发人员看着有些 Filter 实现确实很是别扭，如若不了解也很难懂。

关于 AVFilter 示例，作者将写一个视频滤器，音频滤器和视频滤器的开发接口都是一样的，原理也相同，就是使用的数据单元不同，作者为了节省纸张，不浪费大家的脑细胞就只写出视频滤器的例子，例子中用到了 opencv 库，下面直接进入主题。

```
///< 宏定义
#define STICK_FIGURE_NAME                    "stickfigure"          //stickfigure视频特效

///< AVFilter 对象
AVFilter avfilter_vf_stickfigure = {
STICK_FIGURE_NAME,              ///< 唯一名称表示符,可以通过avfilter_get_by_name找到指定滤器,也可以通过avfilter_graph_create_filter自定解析组装
STICK_FIGURE_NAME"=q=0:c=0:s=10000:e=20000\" (form 10s to 20s)., ///< 完整信息说明
stickfigure_inputs,                  ///< 输入端处理器AVFilterPad
stickfigure_outputs,                 ///< 输出端处理器AVFilterPad
&stickfigure_class,                  ///< 核心私有句柄结构体大小
AVFILTER_FLAG_SUPPORT_TIMELINE_GENERIC,          ///< AV滤器属性表示符
/**
```

```
 *    AVFILTER_FLAG_DYNAMIC_INPUTS                支持输入源动态个数
 *    AVFILTER_FLAG_DYNAMIC_OUTPUTS               支持输出源动态个数
 *    AVFILTER_FLAG_SUPPORT_TIMELINE_GENERIC      支持时间轴上逐帧处理
 *    AVFILTER_FLAG_SUPPORT_TIMELINE_INTERNAL     支持时间轴上回调帧处理
 *    AVFILTER_FLAG_SUPPORT_TIMELINE              支持时间轴上帧处理
 */
stickfigure_init,                   ///< init接口，插件初始化操作
NULL,                               ///< init_dict接口，支持AVOption参数的插件初始化操作
stickfigure_uninit,                 ///< uninit接口，插件反初始化、销毁释放资源的操作
query_formats,                      ///< query_formats接口，支持格式查询
sizeof(StickFigureContext),         ///<  核心私有句柄结构体大小
NULL,                               ///<  框架内部使用
NULL,                               ///< process_command接口，创建Graphic命令行自定义处理解析
NULL,                               ///< init_opaque接口，可以代替init接口的自定义数据的初始化
};

///<  数据格式支持 – 查询函数
static int query_formats(AVFilterContext *ctx)
{
    static const enum AVPixelFormat pix_fmts[] = {AV_PIX_FMT_YUV420P,AV_PIX_FMT_NONE};

    ff_set_common_formats(ctx, ff_make_format_list(pix_fmts));
    return 0;
}

///<  初始化 AVFilter 处理
static av_cold int stickfigure_init(AVFilterContext *ctx)
{
    //  获取核心私有句柄
    StickFigureContext * context = (StickFigureContext *)ctx->priv;
    context->pclass = &stickfigure_class;
    const char *args = NULL;
    int err = 0;
    context->start_Millisecond = 0;
    context->end_Millisecond = 0;

    ///<  参数处理，数值会自动赋值到context中
    int ret = 0;
    av_opt_set_defaults(context);
    if (ret = av_set_options_string(context, args, "=", ":") < 0) {
        av_log(ctx, AV_LOG_ERROR, "Error parsing options string: '%s'\n", args);
        return ret;
    }
```

```
    av_log(ctx,AV_LOG_INFO,"q=%d c=%d s=%I64d
e=%I64d .\n",context->quality_no_speed,context->color_method,context->start_Millisecond,context->end_
Millisecond);
    return 0;
}

///< 反初始化 AVFilter 处理 （ 释放相关资源 ）
static av_cold void stickfigure_uninit(AVFilterContext *ctx)
{
    // 获取核心私有句柄
    StickFigureContext * context = (StickFigureContext *)ctx->priv;
    context->width = 0;
    context->height = 0;

    // 释放opencv库申请的资源
    if(context->pImageSource)
    {cvReleaseImage(&context->pImageSource);context->pImageSource=NULL;}
    if(context->pImageOutput)
    {cvReleaseImage(&context->pImageOutput);context->pImageOutput=NULL;}
    if(context->pImageQuality)
    {cvReleaseImage(&context->pImageQuality);context->pImageQuality=NULL;}

    // 释放图片资源
    if(context->rgb_pic)
    {avpicture_free(context->rgb_pic);delete context->rgb_pic;context->rgb_pic=NULL;}
}
```

关于 AVFilter 定义的接口，其核心的处理函数是在 AVFilterPad 的输入和输出接口处理函数上，下面是相关的定义。这部分是 FFmpeg 最近主要更新的部分，新版本更简化、效率更高、框架更加合理等等。

```
///< 输入端
const AVFilterPad stickfigure_inputs[] = {
{
"default",                          ///< 接入端名称默认使用default
AVMEDIA_TYPE_VIDEO,                 ///< 输入源类型视频
AV_PERM_WRITE|AV_PERM_READ,         ///< min_perms 老板本借口，操作权限
AV_PERM_PRESERVE,                   ///< rej_perms老板本借口，操作权限
NULL,                               ///< start_frame老板本借口
ff_null_get_video_buffer,           ///< get_video_buffer借口，未来也可以会省略，系统默认数
据处理函数
NULL,                               ///< get_audio_buffer借口
NULL,                               ///< end_frame借口
NULL,                               ///< draw_slice借口
```

```
stickfigure_filter_frame,                ///< filter_frame借口，数据处理函数
NULL,                                    ///< poll_frame借口
NULL,                                    ///< request_frame借口
stickfigure_config_input,                ///< config_props借口，借口数据的配置，数据格式检查
NULL                                     ///< int needs_fifo
},
    {NULL}
};

///< 输出端
const AVFilterPad stickfigure_outputs[] = {
    {
        "default",                              ///< 输出端名称默认使用default
        AVMEDIA_TYPE_VIDEO,                     ///< 输出源类型视频
        NULL,NULL,NULL,NULL,
    },
    {NULL}
};
```

这部分主要功能函数就 2 个 stickfigure_config_input 和 stickfigure_filter_frame，这两个接口即使未来版本更新也不会有太大变化。第一函数的功能是，在处理数据前，初始化时检测输入源的格式是否支持等。第二个函数是数据处理函数，真正的处理每一帧的视频数据，下面分别介绍这两个处理函数。

```
///< 参数处理
static int stickfigure_config_input(AVFilterLink * inlink)
{
    // 获取核心私有句柄
    StickFigureContext * context = (StickFigureContext *)inlink->dst->priv;

    // 申请资源
    if(NULL == context->pImageSource || NULL == context->pImageOutput)
    {
        context->width = inlink->w;
        context->height = inlink->h;
        context->pImageSource =
cvCreateImage(cvSize(context->width,context->height),IPL_DEPTH_8U,1);
        context->pImageOutput =
cvCreateImage(cvGetSize(context->pImageSource),IPL_DEPTH_8U,1);
    }
    if(context->quality_no_speed){
        if(context->rgb_pic == NULL){
            context->rgb_pic = new AVPicture;
            memset(context->rgb_pic,0,sizeof(AVPicture));
```

```
                if(avpicture_alloc(context->rgb_pic,PIX_FMT_BGR24,context->width,context->height)<0)
                {delete context->rgb_pic;context->rgb_pic=NULL;}
            }
            if(context->pImageQuality == NULL){
context->pImageQuality = cvCreateImage(cvSize(context->width,context->height),IPL_DEPTH_8U,3);}
        }
        return 0;
}

///< 滤器帧处理函数
static int stickfigure_filter_frame(AVFilterLink *inlink, AVFrame *picref)
{
        // 获取核心私有句柄
        StickFigureContext *context = (StickFigureContext *)inlink->dst->priv;
        AVFilterContext *ctx = inlink->dst;

        // 获取时间
        int64_t m_pts = picref->pts;
        AVRational time_base = inlink->time_base;
        double time = (double)m_pts/(double)time_base.den*(double)time_base.num;
        m_pts = time*1000;

        // 处理过程
        if(m_pts>=context->start_Millisecond && m_pts<=context->end_Millisecond)
        {
            if(context->height != picref->height || context->width != picref->width)
            {
                //Free
                if(context->pImageSource)
                {cvReleaseImage(&context->pImageSource);context->pImageSource=NULL;}
                if(context->pImageOutput)
                {cvReleaseImage(&context->pImageOutput);context->pImageOutput=NULL;}
                //malloc
                context->width = inlink->w;
                context->height = inlink->h;
                context->pImageSource =
cvCreateImage(cvSize(context->width,context->height),IPL_DEPTH_8U,1);
                context->pImageOutput =
cvCreateImage(cvGetSize(context->pImageSource),IPL_DEPTH_8U,1);
                if(context->quality_no_speed)
                {
                    //Free
                    if(context->rgb_pic)
                    {avpicture_free(context->rgb_pic);delete context->rgb_pic;context->rgb_pic=NULL;}
```

```
                    if(context->pImageQuality)
                    {cvReleaseImage(&context->pImageQuality);context->pImageQuality=NULL;}
                    //malloc
                    if(context->rgb_pic == NULL)
                    {
                        context->rgb_pic = new AVPicture;
                        memset(context->rgb_pic,0,sizeof(AVPicture));

    if(avpicture_alloc(context->rgb_pic,PIX_FMT_BGR24,context->width,context->height)<0)
                            {delete context->rgb_pic;context->rgb_pic=NULL;}
                    }
                    if(context->pImageQuality == NULL)
                    {context->pImageQuality =
cvCreateImage(cvSize(context->width,context->height),IPL_DEPTH_8U,3);}
                }
            }
            if((picref->height == context->height)&&(context->height == inlink->h))
            {
                // 这里只写了yuv420p格式的处理
                switch(picref->format)
                {
                case PIX_FMT_YUV420P:
                    {
                        bool need_speed = true;
                        if(context->quality_no_speed && context->rgb_pic)
                        {
                            struct SwsContext * sws = NULL;
                            sws = sws_getContext(context->width, context->height,
PIX_FMT_YUV420P,
                                    context->width, context->height, PIX_FMT_BGR24,
                                    SWS_BILINEAR, NULL, NULL, NULL);
                            int results = sws_scale(sws, picref->data, picref->linesize,
                                0, context->height, context->rgb_pic->data,
context->rgb_pic->linesize);
                            if(context->height == results)
                            {
                                if(context->rgb_pic->linesize[0] ==
context->pImageQuality->widthStep)

    {memcpy(context->pImageQuality->imageData,context->rgb_pic->data[0],context->pImageQuality->
widthStep*context->height);}
                                else
                                {
                                    for(int i=0;i<picref->height;i++)
```

```
{memcpy(context->pImageQuality->imageData+(i*context->pImageQuality->widthStep),context->rgb_pic->data[0]+(i*context->rgb_pic->linesize[0]),context->width);}
                                        }
                                        cvCvtColor(context->pImageQuality, context->pImageSource, CV_BGR2GRAY);
                                        need_speed = false;
                                    }
                                    sws_freeContext(sws);
                                    sws = NULL;
                                }
                                if(need_speed)
                                {
                                    if(picref->linesize[0] == context->pImageSource->widthStep)

{memcpy(context->pImageSource->imageData,picref->data[0],picref->linesize[0]*context->height);}
                                    else
                                    {
                                        for(int i=0;i<picref->height;i++)

{memcpy(context->pImageSource->imageData+(i*context->pImageSource->widthStep),picref->data[0]+(i*picref->linesize[0]),context->width);}
                                    }
                                }
                                cvCanny(context->pImageSource, context->pImageOutput, 50, 150, 3);

                                switch(context->color_method)
                                {
                                case 0:
                                    {
                                        if(picref->linesize[0] == context->pImageOutput->widthStep)

{memcpy(picref->data[0],context->pImageOutput->imageData,picref->linesize[0]*context->height);}
                                        else
                                        {
                                            for(int i=0;i<picref->height;i++)

{memcpy(picref->data[0]+(i*picref->linesize[0]),context->pImageOutput->imageData+(i*context->pImageOutput->widthStep),context->width);}
                                        }
                                        memset(picref->data[1],255/2,picref->linesize[1]/2*picref->height);
                                        memset(picref->data[2],255/2,picref->linesize[2]/2*picref->height);
                                    }break;
                                case 1:
```

```
                                {
                                    unsigned char * src_data = (unsigned char
*)context->pImageOutput->imageData;
                                    unsigned char * dst_data = (unsigned char *)picref->data[0];
                                    for(int i=0;i<picref->height;i++)
                                    {
                                        for(int j=0;j<context->width;j++)
                                        {
                                            if(*(src_data+(i*context->pImageOutput->widthStep+j)))
                                            {*(dst_data+(i*picref->linesize[0]+j))=0;}
                                        }
                                    }
                                }break;
                            case 2:
                                {
                                    cvFloodFill(context->pImageOutput,cvPoint(0,0),cvScalar(255));
                                    unsigned char * src_data = (unsigned char
*)context->pImageOutput->imageData;
                                    unsigned char * dst_data = (unsigned char *)picref->data[0];
                                    for(int i=0;i<picref->height;i++)
                                    {
                                        for(int j=0;j<context->width;j++)
                                        {
                                            if(*(src_data+(i*context->pImageOutput->widthStep+j)))
                                            {*(dst_data+(i*picref->linesize[0]+j))=0;}
                                        }
                                    }
                                }break;
                            }
                        }break;
                    default:
                        {av_log(ctx,AV_LOG_INFO,"Doesn't support this [%d] color-space
processing.\n",picref->format);}break;
                    }
                }
                else
                {av_log(ctx,AV_LOG_WARNING,"Doesn't process Non-full-picture.\n");}
            }

            return ff_filter_frame(inlink, picref);
}
```

5. 汇编优化和算法应用开发

首先 FFmpeg 库内部实现主要以解码器为主，而主流音视频编码器都是以外部库的形式引用调用，所以 FFmpeg 库汇编优化、多线程处理、算法优化方面都是以解码应用为主，例如 openHevc 开源项目（主要是 H265/HEVC 解码）也是依托 FFmpeg 的算法库进行修改的。

FFmpeg 库的汇编优化与算法的框架设计主要是基于 DSP(Digital Signal Processing）思想和接口定义，将通用算法和计算处理函数抽离出编解码过程,在设计 DSP 计算接口一定要简单、高内聚、低耦合、没上下文关联数据、也可以通过宏定义实现，这主要是通过历史演变而来，它核心定义是 DSPContext 结构。

DSPContext 是 FFmpeg 库的通用算法处理对象，主要包含 DCT（Discrete Cosine Transform 离散余弦变换）、MC（motion compensation 运动补偿）、huffman 编码、IDCT（Inverse Discrete Cosine Transformation 逆离散余弦变换）、向量计算、串行化处理等处理操作。

关于汇编优化支持的芯片，主要包含 Alpha 处理器、arm（Advanced RISC Machines）架构指令、avr32 单片机、mips 架构指令、PowerPC 结构指令、Sparc 架构指令、x86 指令集。

FFmpeg 不仅包含 DSPContext 通用算法对象，也包含一些私有算法对象，主要包含 AVFloatDSPContext、PSDSPContext、SBRDSPContext、AC3DSPContext、HpelDSPContext、BinkDSPContext、VideoDSPContext、CAVSDSPContext、DCADSPContext、DiracDSPContext、FLACDSPContext、VideoDSPContext、H264DSPContext、HpelDSPContext、WMV2DSPContext、IntraX8DSPContext、MLPDSPContext、MPADSPContext、PNGDSPContext、ProresDSPContext、RV34DSPContext、VC1DSPContext、VorbisDSPContext、VP3DSPContext、VP56DSPContext、VP8DSPContext、WMV2DSPContext 等等。

这些私有算法对象也主要包含 FDCT（Fast Discrete Curvelet Transforms 快速离散余弦变换）、Loop Filter、Deblocking Filter 去宏块滤波、bypass transform、向量计算、边缘化处理、MSB（Most Significant Bit 最高加权位）等等。

在不同平台、架构和芯片上进行算法的汇编优化，汇编优化级别、深度也是不同的，例如 x86 架构的芯片其媒体指令集发展从 SSE(Streaming SIMD Extensions)、SSE2、SSSE3、SSE4、SSE5、AVX（Advanced Vector Extensions）到 FMA（Fused Multiply Accumulate）等指令集的发展，armv6、armv7、armv9、armv11 等的发展，都是需要进行汇编优化的方向。

关于性能优化也可以使用硬件加速，FFmpeg 库的硬件加速对象是 AVHWAccel 结构，主要是在码流或是数据帧的标准数据单元处理前做的 Hook 类操作一样。

关于这一小节的内容主要还是放到下一版本书的开发中做详细的介绍。

6. 标准化文档

为什么要在这里介绍标准化文档？主要因为它和我们的工作有密切的关联，了解、学习或是找到一个标准化文档都是工作能力、积累的重要知识点。

在了解标准化文档前，先从宏观看一看标准化组织，涉及到国际或国内标准化组织主要有：ITU、IETF、IOS、IEEE、IEC、ANSI、AVS 工作组等。

ITU（International Telecommunications Union）国际电信联盟 ITU-R 无线电通信部门 ITU-T 电信标准化部门 ITU-D 电信发展部门
IETF（Internet Engineering Task Force）互联网工程任务组 IESG 互联网工程指导小组 IAB 因特网结构委员会 ISOC 国际互联网协会 IANA 互联网数字分配机构
ISO（International Organization For Standardization）国际标准化组织 很多下属组织机构和部门
IEEE（Institute of Electrical and Electronics Engineers）美国电气和电子工程师协会。
IEC（International Electrotechnical Commission）国际电工委员会。
ANSI（American National Standards Institute）美国国家标准学会。
MPEG（Moving Pictures Experts Group）动态图像专家组，和 ISO、IEC 有关联。
VCEG（Video Coding Experts Group）视频编码专家组，属于 ITU-T。
JVT（Joint Video Team）联合视频工作组，是 MPEG 和 VCEG 联合组建的。
AVS 工作组，国内的音视频编解码标准化组织，老外通常叫 Chinese AVS。

标准化文档都是有这些标准化组织撰写和制定，除了这些国际标准化文档，相关行业的**“龙头企业”**也为自己制定了一些文档，**公开自己的容器、协议、或是编解码标准**，这些文档同样很重要，例如：flv、rtmp、mms、apple-http 等等。这些文档需要自己去相关网站、或是内部资料进行整理寻找。

第五篇 其他库简要介绍

首先介绍几个非常有名的媒体技术论坛 Doom9（http://www.doom9.org/）和 Doom10（http://doom10.org/），国内也有中华视频网（http://bbs.chinavideo.org/）等。

1. 多媒体开发常用开源库

库名称	库介绍
OpenCV	图像处理和计算机视觉通用算法开源库，也包含数据格式转换、渲染、采集等等功能。
SDL	简单的多媒体渲染、采集开源库。
WebM	由 Google 资助的多媒体开发的项目，主要包含 libvpx（vp8、vp9 等）库、WebM 音视频文件容器、RTP 传输等（另注：WebP 是 Google 发展出来的图片文件格式）。
WebRTC	由 Google 收购发展而来的网页实时通信的开源解决方案，通过 JavaScript API 或是 HTML5 实现浏览器端音视频应用，同时也是一套流媒体解决方案标准。
GPAC	科研和学术领域的多媒体开源框架，包含独立项目：MP4Client、MP4Box、Osmo4 等。
ffmpeg、libav	是知名的开源多媒体开发基础框架库，包含很多种编解码器、容器以及协议。
GStreamer	是 GNOME 桌面环境下用来构建流媒体应用的多媒体开源框架库。
VideoLAN	非盈利性的开源开发者组织，包含 VLC、VLMC、x264、Multicat、DVBlast 等多个开源项目。
MPlayer	知名的开源媒体播放器项目。
MediaInfo	是分析音视频文件的容器、编解码信息工具的开源项目。
Live555	是支持 RTSP、HTTP 流媒体服务器的开源项目。
Speex	是音频编解码器的开源项目。
LibGSM	是 GSM 音频编解码库的开源项目库。
PJSIP	是一个基于 SIP 的开源多媒体通信框架库。
Xvid	开源的 MPEG-4 视频编解码库。
libZPlay	是多个音频编解码器集成开源库。
RtmpDump	开源的 RTMP 流媒体工具库。
OpenVSS	视频监控软件的视频分析开源框架，似乎 2010 年后无更新维护。
yamdi	Flv 文件开源工具。
frei0r	开源的视频特效处理库。
FreeType	开源的字体文件解码、渲染库。
JWPlayer	基于 Web 的 flash 播放器。
Html5Media	基于 Web 的 HTML5 或 flash 播放器。

libass	开源的字体文件渲染库。
LameMP3	开源的 MP3 音频编码库。
iLBC	现归于 WebRTC 开源音频低码率编解码库。
Ogg Vorbis	开源的音频编解码器、容器库。
Schroedinger	开源的 Dirac 视频编解码器库。

2. 多媒体开发常用 SDK 库

库名称	库介绍
OpenGL、OpenAL	图形图像、音频渲染处理的接口标准。
OpenCL	通用并行多媒体编程的接口标准。
OpenCore	Android 多媒体核心框架库。
VisualOn	媒体播放器支持库。
FAAC 、 FAAD2 、 NeroAAC、aacplus、 libvoaac、fdk-aac、 Opus 等等。	开源或是非开源的 AAC 音视频编解码器。