

# 《FFmpeg 基础库编程开发》

# 目录

第一章 多媒体概念介绍 .....	6
1.1 视频格式 .....	6
1.1.1 常见格式 .....	6
1.2 音频格式 .....	8
1.2.1 常见格式 .....	9
1.2.2 比较 .....	14
1.3 字幕格式 .....	14
1.3.1 外挂字幕与内嵌字幕的阐述 .....	14
1.3.2 外挂字幕视频与内嵌字幕视频的画面比较 .....	15
1.3.3 外挂字幕的三种格式 .....	15
1.4 采集录制和播放渲染 .....	15
1.4.1 视频采集 .....	15
1.4.2 视频录制 .....	16
1.4.3 视频渲染 .....	16
1.5 编解码器 .....	18
1.6 容器和协议 .....	18
1.6.1 容器格式和编码格式 .....	18
1.6.2 协议 .....	24
1.6.2.1 视频协议 .....	25
1.6.2.2 音频协议 .....	25
1.6.2.3 上层通讯协议 .....	25
1.7 常用概念介绍 .....	26
1.7.1 硬解 .....	26
1.7.2 IBP 帧 .....	26
1.7.3 DTS 和 PTS .....	30
1.7.4 分辨率 .....	30
1.7.5 码率 .....	30
1.7.6 帧率 .....	30
1.7.7 RGB 和 YUV .....	30
1.7.8 实时和非实时 .....	30
1.7.9 复合视频和 s-video .....	31
1.7.10 硬件加速 .....	31
1.7.11 FFmpeg Device .....	31
第二章 FFmpeg 框架 .....	32
2.1 FFmpeg 概述 .....	32
2.1.1 简介 .....	32
2.1.2 功能 .....	32
2.1.3 模块组成 .....	33
2.1.4 命令集 .....	33
2.2 媒体播放器三大底层框架 .....	35
第三章 编译及简单应用 .....	39
3.1 FFmpeg 库编译和入门介绍 41 .....	39
3.2 流媒体数据流程讲解 .....	40

## 《FFmpeg 基础库编程开发》

3.3 简单应用 .....	42
3.4 SDL ( Simple Direct Layer) .....	45
3.4.1 SDL 显示视频.....	45
3.4.2 SDL 显示音频.....	46
3.5 ffmpeg 程序的使用 (ffmpeg.exe, ffplay.exe, ffprobe.exe) .....	46
3.5.1 ffmpeg.exe .....	46
3.5.2 ffplay.exe .....	46
3.5.3 ffprobe.exe.....	46
第四章 数据结构 .....	50
4.1 AVCodec 结构体.....	51
4.2 AVCodecContext 结构体.....	52
4.3 AVInputFormat 结构体.....	53
4.4 AVFormatContext 结构体.....	62
4.5 MovContext 结构体.....	63
4.6 URLProtocol 结构体.....	63
4.7 URLContext 结构体 .....	64
4.8 AVIOContext 结构体(老版本为: ByteIOContext) .....	64
4.9 AVStream 结构体.....	65
4.10 MOVStreamContext 结构体.....	66
4.11 AVPacket 结构体 .....	67
4.12 AVPacketList 结构体 .....	67
4.13 AVFrame 结构体 .....	53
第五章 重要模块 .....	68
5.1 libavutil 公共模块.....	68
1 文件列表 .....	68
2 common.h 文件 .....	68
3 bswap.h 文件 .....	70
4 rational.h 文件 .....	71
5 mathematics.h 文件 .....	71
6 avutil.h 文件 .....	72
5.2 libavcodec 编解码模块.....	73
1 文件列表 .....	73
2 avcodec.h 文件 .....	74
3 allcodec.c 文件 .....	78
4 dsutil.h 文件 .....	79
5 dsutil.c 文件 .....	79
6 utils_codec.c 文件 .....	80
7 imgconvert_template.h 文件 .....	90
8 imgconvert.c 文件 .....	110
9 msrle.c 文件 .....	152
10 turespeech_data.h 文件 .....	159
11 turespeech.c 文件 .....	162
5.3 libavformat 容器模块 .....	171
1 文件列表 .....	171
2 avformat.h 文件.....	172

# 《FFmpeg 基础库编程开发》

3 allformat.c 文件.....	177
4 cutils.c 文件.....	178
5 file.c 文件.....	179
6 avio.h 文件.....	182
7 avio.c 文件.....	184
8 aviobuf.c 文件.....	188
9 utils_format.c 文件.....	197
10 avidec.c 文件.....	208
5.4 libswscale 视频色彩空间转换.....	230
5.5 libswresample 音频重采样.....	230
5.6 libavfilter 音视频滤器.....	230
5.7 libavdevice 设备输入和输出容器.....	230
5.8 libpostproc 视频后期处理.....	230
第六章 播放器.....	230
6.1 视频播放器.....	230
6.1.1 ffmpeg 库的配置.....	230
6.1.2 一个简单的视频播放器.....	231
6.2 音频播放器.....	234
6.3 一个完整的播放器--ffplay.....	240
6.3.1 ffplay 流程图.....	240
6.3.2 ffplay 源码剖析.....	240
第七章 应用开发.....	262
7.1 ffmpeg 库的使用：编码.....	262
第八章 关键函数介绍.....	267
8.1 avformat_open_input.....	267
8.2 avcodec_register_all() .....	268
8.3 av_read_frame() .....	269
8.4 avcodec_decode_video2() .....	270
8.5 transcode_init().....	270
8.6 transcode().....	280
第九章 ffmpeg 相关工程.....	288
9.1 ffdshow .....	288
ffdshow 源代码分析 1 : 整体结构 .....	288
ffdshow 源代码分析 2: 位图覆盖滤镜（对话框部分 Dialog） .....	290
ffdshow 源代码分析 3: 位图覆盖滤镜（设置部分 Settings） .....	297
ffdshow 源代码分析 4: 位图覆盖滤镜（滤镜部分 Filter） .....	301
ffdshow 源代码分析 5: 位图覆盖滤镜（总结） .....	306
ffdshow 源代码分析 6: 对解码器的 dll 的封装（libavcodec） .....	306
ffdshow 源代码分析 8: 视频解码器类（TvideoCodecDec） .....	328
ffdshow 源代码分析 9: 编解码器有关类的总结 .....	335
9.2 LAV filters .....	340
LAV Filter 源代码分析 1: 总体结构 .....	340
LAV Filter 源代码分析 2: LAV Splitter.....	341
LAV Filter 源代码分析 3: LAV Video (1) .....	364
LAV Filter 源代码分析 4: LAV Video (2) .....	382

## 《FFmpeg 基础库编程开发》

9.3 MPlayer .....	408
9.3.1 Mplayer 支持的格式.....	408
9.3.2 Mplayer 中头文件的功能分析.....	408
9.3.3 MPlayer.main 主流程简要说明.....	408
9.3.4 Mplayer 源码分析.....	409
第十章 开发实例.....	416
第十一章 mp4 文件封装协议分析.....	416
11.1 概述 .....	416
11.2 mp4 的物理结构 .....	416
11.3 数据的组织结构 .....	417
11.4 mp4 的时间结构 .....	417
11.5 文件结构分析 .....	418
11.5.1 File Type Box (ftyp) .....	418
11.5.2 Movie Box (moov) .....	418
第十二章 flv 文件格式分析.....	437
12.1 概述 .....	437
12.2 文件总体结构 .....	437
12.3 文件结构分析 .....	438
12.3.1 flv 文件头的结构.....	438
12.3.2 body 主体结构 .....	439
附录 A: 常见问题 .....	444
1 ffmpeg 从内存中读取数据.....	444
2 MFC 中使用 SDL 播放音频没有声音的解决方法 .....	444
附录 B: 经典代码示例 .....	445
附录 c: ffmpeg 参数中文详细解释 .....	456
附录 D: ffplay 的快捷键以及选项.....	458
附录 E: ffmpeg 处理 rtmp 流媒体.....	459

# 第一章 多媒体概念介绍

## 1.1 视频格式

视频格式可以分为适合本地播放的本地影像视频和适合在网络中播放的网络流媒体影像视频两大类。尽管后者在播放的稳定性和播放画面质量上可能没有前者优秀，但网络流媒体影像视频的广泛传播性使之正被广泛应用于视频点播、网络演示、远程教育、网络视频广告等等互联网信息服务领域。

注：原始的视频数据可以理解为通过摄像头等驱动获取的没有经过编码的数据，市面上 usb 摄像头输出格式常见的有：RGB24、YUV2、YV2（这些都是没有编码的原始数据），MJPEG（经过编码的数据）。摄像头捕捉的数据也是可以设置的，比如 windows 下用 cap 来设置。

### 1.1.1 常见格式

#### MPEG/MPG/DAT

MPEG(运动图像专家组)是 Motion Picture Experts Group 的缩写。这类格式包括了 MPEG-1,MPEG-2 和 MPEG-4 在内的多种视频格式。MPEG-1 相信是大家接触得最多的了，因为其正在被广泛地应用在 VCD 的制作和一些视频片段下载的网络应用上面，大部分的 VCD 都是用 MPEG1 格式压缩的(刻录软件自动将 MPEG1 转换为 DAT 格式)，使用 MPEG-1 的压缩算法，可以把一部 120 分钟长的电影压缩到 1.2 GB 左右大小。MPEG-2 则是应用在 DVD 的制作，同时在一些 HDTV (高清晰电视广播) 和一些高要求视频编辑、处理上面也有相当多的应用。使用 MPEG-2 的压缩算法压缩一部 120 分钟长的电影可以压缩到 5-8 GB 的大小 (MPEG2 的图像质量是 MPEG-1 无法比拟的)。MPEG 系列标准已成为国际上影响最大的多媒体技术标准，其中 MPEG-1 和 MPEG-2 是采用相同原理为基础的预测编码、变换编码、熵编码及运动补偿等第一代数据压缩编码技术；MPEG-4 (ISO/IEC 14496) 则是基于第二代压缩编码技术制定的国际标准，它以视听媒体对象为基本单元，采用基于内容的压缩编码，以实现数字视音频、图形合成应用及交互式多媒体的集成。MPEG 系列标准对 VCD、DVD 等视听消费电子及数字电视和高清晰度电视 (DTV&&HDTV) 、多媒体通信等信息产业的发展产生了巨大而深远的影响。



#### AVI

AVI，音频视频交错(Audio Video Interleaved)的英文缩写。AVI 这个由微软公司发表的视频格式，在视频领域可以说是最悠久的格式之一。AVI 格式调用方便、图像质量好，压缩标准可任意选择，是应用最广泛、也是应用时间最长的格式之一。

## MOV

使用过 Mac 机的朋友应该多少接触过 QuickTime。QuickTime 原本是 Apple 公司用于 Mac 计算机上的一种图像视频处理软件。Quick-Time 提供了两种标准图像和数字视频格式，即可以支持静态的\*.PIC 和\*.JPG 图像格式，动态的基于 Indeo 压缩法的\*.MOV 和基于 MPEG 压缩法的\*.MPG 视频格式。

## ASF

ASF(Advanced Streaming format 高级流格式)。ASF 是 MICROSOFT 为了和的 Real player 竞争而发展出来的一种可以直接在网上观看视频节目的文件压缩格式。ASF 使用了 MPEG4 的压缩算法，压缩率和图像的质量都很不错。因为 ASF 是以一个可以在网上即时观赏的视频“流”格式存在的，所以它的图像质量比 VCD 差一点点并不出奇，但比同是视频“流”格式的 RAM 格式要好。

## WMV

一种独立于编码方式的在 Internet 上实时传播多媒体的技术标准，Microsoft 公司希望用其取代 QuickTime 之类的技术标准以及 WAV、AVI 之类的文件扩展名。WMV 的主要优点在于：可扩充的媒体类型、本地或网络回放、可伸缩的媒体类型、流的优先级化、多语言支持、扩展性等。

## NAVI

如果发现原来的播放软件突然打不开此类格式的 AVI 文件，那你就考虑是不是碰到了 n AVI。n AVI 是 New AVI 的缩写，是一个名为 Shadow Realm 的地下组织发展起来的一种新视频格式。它是由 Microsoft ASF 压缩算法的修改而来的（并不是想象中的 AVI），视频格式追求的无非是压缩率和图像质量，所以 NAVI 为了追求这个目标，改善了原始的 ASF 格式的一些不足，让 NAVI 可以拥有更高的帧率。可以这样说，NAVI 是一种去掉视频流特性的改良型 ASF 格式。

## 3GP

3GP 是一种 3G 流媒体的视频编码格式，主要是为了配合 3G 网络的高传输速度而开发的，也是目前手机中最为常见的一种视频格式。

简单的说，该格式是“第三代合作伙伴项目”(3GPP)制定的一种多媒体标准，使用户能使用手机享受高质量的视频、音频等多媒体内容。其核心由包括高级音频编码(AAC)、自适应多速率 (AMR) 和 MPEG-4 和 H.263 视频编码解码器等组成，目前大部分支持视频拍摄的手机都支持 3GPP 格式的视频播放。其特点是网速占用较少，但画质较差。

## REAL VIDEO

REAL VIDEO (RA、RAM) 格式由一开始就是定位在视频流应用方面的，也可以说是视频流技术的始创者。它可以在用 56K MODEM 拨号上网的条件实现不间断的视频播放，当然，其图像质量和 MPEG2、DIVX 等比是不敢恭维的啦。毕竟要实现在网上传输不间断的视频是需要很大的频宽的，这方面是 ASF 的有力竞争者。

## MKV

一种后缀为 MKV 的视频文件频频出现在网络上，它可在在一个文件中集成多条不同类型的音轨和字幕轨，而且其视频编码的自由度也非常大，可以是常见的 DivX、XviD、3IVX，甚至可以是 RealVideo、QuickTime、WMV 这类流式视频。实际上，它是一种全称为 Matroska 的新型多媒体封装格式，这种先进的、开放的封装格式已经给我们展示出非常好的应用前景。

## FLV

FLV 是 FLASH VIDEO 的简称，FLV 流媒体格式是一种新的视频格式。由于它形成的文件极小、加载速度极快，使得网络观看视频文件成为可能，它的出现有效地解决了视频文件导入 Flash 后，使导出的 SWF 文件体积庞大，不能在网络上很好的使用等缺点。

## F4V

作为一种更小更清晰，更利于在网络传播的格式，F4V 已经逐渐取代了传统 FLV，也已经被大多数主流播放器兼容播放，而不需要通过转换等复杂的方式。F4V 是 Adobe 公司为了迎接高清时代而推出继 FLV 格式后的支持 H.264 的 F4V 流媒体格式。它和 FLV 主要的区别在于，FLV 格式采用的是 H263 编码，而 F4V 则支持 H.264 编码的高清晰视频，码率最高可达 50Mbps。也就是说 F4V 和 FLV 在同等体积的前提下，能够实现更高的分辨率，并支持更高比特率，就是我们所说的更清晰更流畅。另外，很多主流媒体网站上下载的 F4V 文件后缀却为 FLV，这是 F4V 格式的另一个特点，属正常现象，观看时可明显感觉到这种实为 F4V 的 FLV 有明显更高的清晰度和流畅度。

## RMVB

RMVB 的前身为 RM 格式，它们是 Real Networks 公司所制定的音频视频压缩规范，根据不同的网络传输速率，而制定出不同的压缩比率，从而实现在低速率的网络上进行影像数据实时传送和播放，具有体积小，画质也还不错的特点。

早期的 RM 格式为了能够实现在有限带宽的情况下，进行视频在线播放而被研发出来，并一度红遍整个互联网。而为了实现更优化的体积与画面质量，Real Networks 公司不久又在 RM 的基础上，推出了可变比特率编码的 RMVB 格式。RMVB 的诞生，打破了原先 RM 格式那种平均压缩采样的方式，在保证平均压缩比的基础上，采用浮动比特率编码的方式，将较高的比特率用于复杂的动态画面（如歌舞、飞车、战争等），而在静态画面中则灵活地转为较低的采样率，从而合理地利用了比特率资源，使 RMVB 最大限度地压缩了影片的大小，最终拥有了近乎完美的接近于 DVD 品质的视听效果。我们可以做个简单对比，一般而言一部 120 分钟的 dvd 体积为 4GB，而 rmvb 格式来压缩，仅 400MB 左右，而且清晰度流畅度并不比原 DVD 差太远。

人们为了缩短视频文件在网络进行传播的下载时间，为了节约用户电脑硬盘宝贵的空间容量，已越来越多的视频被压制成了 RMVB 格式，并广为流传。到如今，可能每一位电脑使用者（或许就包括正在阅读这篇文章的您）电脑中的视频文件，超过 80% 都会是 RMVB 格式。

RMVB 由于本身的优势，成为目前 PC 中最广泛存在的视频格式，但在 MP4 播放器中，RMVB 格式却长期得不到重视。MP4 发展的整整七个年头里，虽然早就可以做到完美支持 AVI 格式，但却久久未有能够完全兼容 RMVB 格式的机型诞生。对于 MP4，尤其是容量小价格便宜的闪存 MP4 而言，怎样的视频格式才将会是其未来的主流呢？我们不妨来探讨一番。

## WebM

由 Google 提出，是一个开放、免费的媒体文件格式。WebM 影片格式其实是以 Matroska（即 MKV）容器格式为基础开发的新容器格式，里面包括了 VP8 影片轨和 Ogg Vorbis 音轨，其中 Google 将其拥有的 VP8 视频编码技术以类似 BSD 授权开源，Ogg Vorbis 本来就是开放格式。WebM 标准的网络视频更加偏向于开源并且是基于 HTML5 标准的，WebM 项目旨在为对每个人都开放的网络开发高质量、开放的视频格式，其重点是解决视频服务这一核心的网络用户体验。Google 说 WebM 的格式相当有效率，应该可以在 netbook、tablet、手持式装置等上面顺畅地使用。

Ogg Vorbis 本来就是开放格式，大家应该都知道，至于 VP8 则是 Google 当年买下一间叫 On2 的公司的时候，取得的 Video Codec，Google 也把这个 Codec 以类似 BSD 授权放出来，因此 WebM 应该是不会有 H.264 的那些潜在的专利问题。

Youtube 也会支持 WebM 的播放。来自产业界的有 Adobe -- Flash Player 将会支持 WebM 格式的播放 -- AMD、ARM、Broadcom、Freescale、NVIDIA、Qualcomm、TI 等。谁不在上头？Intel。在 Browser 方面，Chrome 不要说，Firefox、Opera 都已经表态将会支持这个新格式。微软 IE9 的支持就没这么直接，出厂时仅会支持 H.264 影片的播放，但如果你另外下载并安装了 VP8，那当然你也可以播放 HTML / VP8 的影片。要推动一个新格式进入主流，甚至成为龙头老大，是非常不容易的。但 WebM 和 VP8 的推动者是 Google，而且是在 H.264 正因为其非开放性而备受质疑的时候，或许 WebM 真有机会迅速地站稳脚跟，一举成为新一代的影片通用格式呢！

## 1.2 音频格式

音频格式是指要在计算机内播放或是处理音频文件，也就是要对声音文件进行数、模转换，这个过程同样由采样和量化构成，人耳所能听到的声音，最低的频率是从 20Hz 起一直到最高频率 20KHZ，20KHz 以上人耳是听不到的，因此音频文件格式的最大带宽是 20KHZ，故而采样速率需要介于 40~50KHZ 之间，而且对每个样本需要更多的量化比特数。音频数字化的标准是每个样本 16 位-96dB 的信噪比，采用线性脉冲编码调制 PCM，每一量化步长都具有相等的长度。在音频文件的制作中，正是采用这一标准。

## 1.2.1 常见格式

常见的音频格式有：CD 格式、WAVE (\*.WAV)、AIFF、AU、MP3、MIDI、WMA、RealAudio、VQF、OggVorbis、AAC、APE。

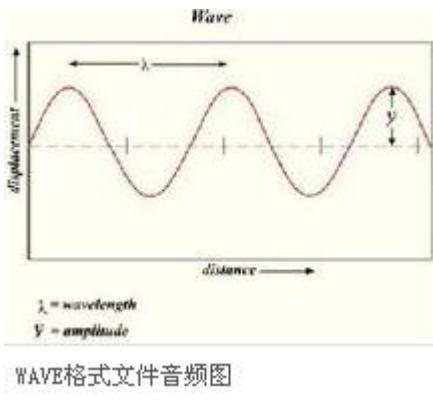
### CD



cd光盘用于储存cd格式文件

CD 格式的音质是比较高的音频格式。因此要讲音频格式，CD 自然是打头阵的先锋。在大多数播放软件的“打开文件类型”中，都可以看到\*.cda 格式，这就是 CD 音轨了。标准 CD 格式也就是 44.1K 的采样频率，速率 88K/秒，16 位量化位数，因为 CD 音轨可以说是近似无损的，因此它的声音基本上是忠于原声的，因此如果你是一个音响发烧友的话，CD 是你的首选。它会让你感受到天籁之音。CD 光盘可以在 CD 唱机中播放，也能用电脑里的各种播放软件来重放。一个 CD 音频文件是一个\*.cda 文件，这只是一个索引信息，并不是真正的包含声音信息，所以不论 CD 音乐的长短，在电脑上看到的“\*.cda 文件”都是 44 字节长。注意：不能直接的复制 CD 格式的\*.cda 文件到硬盘上播放，需要使用象 EAC 这样的抓音轨软件把 CD 格式的文件转换成 WAV，这个转换过程如果光盘驱动器质量过关而且 EAC 的参数设置得当的话，可以说是基本上无损抓音频。推荐大家使用这种方法。

### WAVE

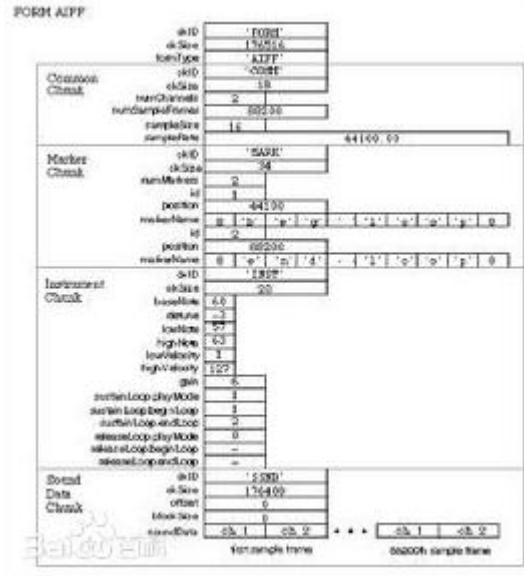


WAVE格式文件音频图

WAVE (\*.WAV) 是微软公司开发的一种声音文件格式，它符合 PIFFResource Interchange File Format 文件规范，用于保存 WINDOWS 平台的音频信息资源，被 WINDOWS 平台及其应用程序所支持。“\*.WAV” 格式支持 MSADPCM、CCITT A LAW 等多种压缩算法，支持多种音频位数、采样频率和声道，标准格式的 WAV 文件和 CD 格式一样，也是 44.1K 的采样频率，速率 88K/秒，16 位量化位数，看到了吧，WAV 格式的声音文件质量和 CD 相差无几，也是目前 PC 机上广为流行的声音文件格式，几乎所有的音频编辑软件都“认识” WAV 格式。

### AIFF

AIFF (Audio Interchange File Format) 格式和 AU 格式，它们都和 WAV 非常相像，在大多数的音频编辑软件中也都支持它们这几种常见的音乐格式。

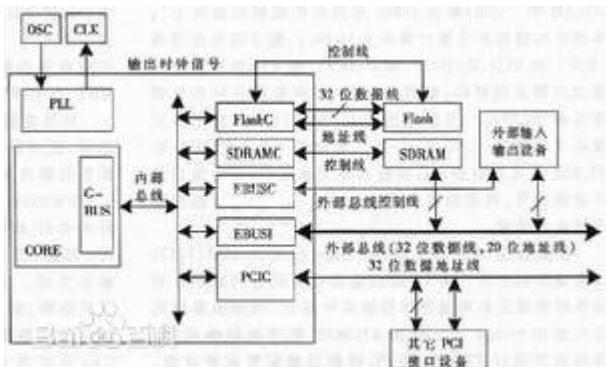


AIFF 是音频交换文件格式的英文缩写。是 APPLE 公司开发的一种音频文件格式，被 MACINTOSH 平台及其应用程序所支持，NETSCAPE 浏览器中 LIVEAUDIO 也支持 AIFF 格式。所以大家都不常见。AIFF 是 Apple 苹果电脑上面的标准音频格式，属于 QuickTime 技术的一部分。这一格式的特点就是格式本身与数据的意义无关，因此受到了 Microsoft 的青睐，并据此搞出来 WAV 格式。AIFF 虽然是一种很优秀的文件格式，但由于它是苹果电脑上的格式，因此在 PC 平台上并没有得到很大的流行。不过由于 Apple 电脑多用于多媒体制作出版行业，因此几乎所有的音频编辑软件和播放软件都或多或少地支持 AIFF 格式。只要苹果电脑还在，AIFF 就始终还占有一席之地。由于 AIFF 的包容特性，所以它支持许多压缩技术。

## AU

AUDIO 文件是 SUN 公司推出的一种数字音频格式。AU 文件原先是 UNIX 操作系统下的数字声音文件。由于早期 INTERNET 上的 WEB 服务器主要是基于 UNIX 的，所以，AU 格式的文件在如今的 INTERNET 中也是常用的声音文件格式。

## MPEG



MPEG 是动态图象专家组的英文缩写。这个专家组始建于 1988 年，专门负责为 CD 建立视频和音频压缩标准。MPEG 音频文件指的是 MPEG 标准中的声音部分即 MPEG 音频层。目前 INTERNET 上的音乐格式以 MP3 最为常见。虽然它是一种有损压缩，但是它的最大优势是以极小的声音失真换来了较高的压缩比。MPEG 含有格式包括：MPEG-1、MPEG-2、MPEG-Layer3、MPEG-4

## MP3

MP3 格式诞生于八十年代的德国，所谓的 MP3 也就是指的是 MPEG 标准中的音频部分，也就是 MPEG 音频层。根据压缩质量和编码处理的不同分为 3 层，分别对应 “\*.mp1” / “\*.mp2” / “\*.mp3” 这 3 种声音文件。需要提醒大家注意的地方是：MPEG 音频文件的压缩是一种有损压缩，MPEG3 音频编码具有 10: 1~12: 1 的高压缩率，同时基本保持低音频部分不失真，但是牺牲了声音文件中 12KHz 到 16KHz 高音频这部分的质量来换取文件的尺寸，相

## 《FFmpeg 基础库编程开发》

同长度的音乐文件，用\*.mp3 格式来储存，一般只有\*.wav 文件的 1/10，而音质要次于 CD 格式或 WAV 格式的声音文件。由于其文件尺寸小，音质好；所以在它问世之初还没有什么别的音频格式可以与之匹敌，因而为\*.mp3 格式的发展提供了良好的条件。直到现在，这种格式还是风靡一时，作为主流音频格式的地位难以被撼动。但是树大招风，MP3 音乐的版权问题也一直是找不到办法解决，因为 MP3 没有版权保护技术，说白了也就是谁都可以用。

MP3 格式压缩音乐的采样频率有很多种，可以用 64Kbps 或更低的采样频率节省空间，也可以用 320Kbps 的标准达到极高的音质。用装有 Fraunhofer IIS Mpeg Lyaer3 的 MP3 编码器(现在效果最好的编码器)MusicMatch Jukebox 6.0 在 128Kbps 的频率下编码一首 3 分钟的歌曲，得到 2.82MB 的 MP3 文件。采用缺省的 CBR (固定采样频率) 技术可以以固定的频率采样一首歌曲，而 VBR (可变采样频率) 则可以在音乐“忙”的时候加大采样的频率获取更高的音质，不过产生的 MP3 文件可能在某些播放器上无法播放。把 VBR 的级别设定成为与前面的 CBR 文件的音质基本一样，生成的 VBR MP3 文件为 2.9MB。

MP3 是到 2008 年止使用用户最多的有损压缩数字音频格式了。它的全称是 MPEG(MPEG: MovingPictureExpertsGroup)AudioLayer-3，刚出现时它的编码技术并不完善，它更像一个编码标准框架，留待人们去完善。早期的 MP3 编码采用的是固定编码率的方式 (CBR)，看到的 128KBPS，就是代表它是以 128KBPS 固定数据速率编码——你可以提高这个编码率，最高可以到 320KBPS，音质会更好，自然，文件的体积会相应增大。

因为 MP3 的编码方式是开放的，可以在这个标准框架的基础上自己选择不同的声学原理进行压缩处理，所以，很快由 Xing 公司推出可变编码率的压缩方式 (VBR)。它的原理就是利用将一首歌的复杂部分用高 bitrate 编码，简单部分用低 bitrate 编码，通过这种方式，进一步取得质量和体积的统一。当然，早期的 Xing 编码器的 VBR 算法很差，音质与 CBR (固定码率) 相去甚远。但是，这种算法指明了一种方向，其他开发者纷纷推出自己的 VBR 算法，使得效果一直在改进。目前公认比较好的首推 LAME，它完美地实现了 VBR 算法，而且它是完全免费的软件，并且由爱好者组成的开发团队一直在不断的发展完善。

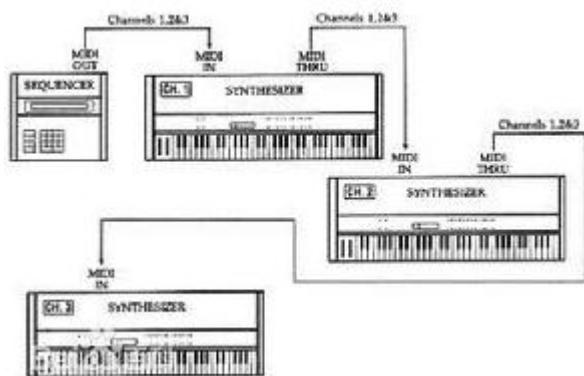
而在 VBR 的基础上，LAME 更加发展出 ABR 算法。ABR (AverageBitrate) 平均比特率，是 VBR 的一种插值参数。LAME 针对 CBR 不佳的文件体积比和 VBR 生成文件大小不定的特点独创了这种编码模式。ABR 在指定的文件大小内，以每 50 帧 (30 帧约 1 秒) 为一段，低频和不敏感频率使用相对低的流量，高频和大动态表现时使用高流量，可以做为 VBR 和 CBR 的一种折衷选择。

MP3 问世不久，就凭这较高的压缩比 12:1 和较好的音质创造了一个全新的音乐领域，然而 MP3 的开放性却最终不可避免的导致了版权之争，在这样的背景之下，文件更小，音质更佳，同时还能有效保护版权的 MP4 就应运而生了。MP3 和 MP4 之间其实并没有必然的联系，首先 MP3 是一种音频压缩的国际技术标准，而 MP4 却是一个商标的名称。

## MPEG-4

MPEG-4 标准是由国际运动图像专家组于 2000 年 10 月公布的一种面向多媒体应用的视频压缩标准。它采用了基于对象的压缩编码技术，在编码前首先对视频序列进行分析，从原始图像中分割出各个视频对象，然后再分别对每个视频对象的形状信息、运动信息、纹理信息单独编码，并通过比 MPEG-2 更优的运动预测和运动补偿来去除连续帧之间的时间冗余。其核心是基于内容的尺度可变性(Content-basedscalability)，可以对图像中各个对象分配优先级，对比较重要的对象用高的空间和时间分辨率表示，对不甚重要的对象(如监控系统的背景)以较低的分辨率表示，甚至不显示。因此它具有自适应调配资源能力，可以实现高质量低速率的图像通信和视频传输。MPEG-4 以其高质量、低传输速率等优点已经被广泛应用到网络多媒体、视频会议和多媒体监控等图像传输系统中。中国内外大部分成熟的 MPEG-4 应用均为基于 PC 层面的客户端和服务器模式，应用在嵌入式系统上的并不多，且多数嵌入式 MPEG-4 解码系统大多使用商业的嵌入式操作系统，如 WindowsCE、VxWorks 等，成本高、灵活性差。如以嵌入式 Linux 作为操作系统不仅开发方便，且可以节约成本，并可以根据实际情况进行裁减，占用资源少、灵活性强，网络性能好，适用范围更广。

## MIDI



MIDI (Musical Instrument Digital Interface) 格式被经常玩音乐的人使用，MIDI 允许数字合成器和其他设备交换数据。MID 文件格式由 MIDI 继承而来。MID 文件并不是一段录制好的声音，而是记录声音的信息，然后在告诉声卡如何再现音乐的一组指令。这样一个 MIDI 文件每存 1 分钟的音乐只用大约 5~10KB。MID 文件主要用于原始乐器作品，流行歌曲的业余表演，游戏音轨以及电子贺卡等。\*.mid 文件重放的效果完全依赖声卡的档次。\*.mid 格式最大的用处是在电脑作曲领域。\*.mid 文件可以用作曲软件写出，也可以通过声卡的 MIDI 口把外接音序器演奏的乐曲输入电脑里，制成\*.mid 文件。

### WMA

WMA (Windows Media Audio) 格式是来自于微软的重量级选手，后台强硬，音质要强于 MP3 格式，更远胜于 RA 格式，它和日本 YAMAHA 公司开发的 VQF 格式一样，是以减少数据流量但保持音质的方法来达到比 MP3 压缩率更高的目的，WMA 的压缩率一般都可以达到 1: 18 左右，WMA 的另一个优点是内容提供商可以通过 DRM (Digital Rights Management) 方案如 Windows Media Rights Manager 7 加入防拷贝保护。这种内置了版权保护技术可以限制播放时间和播放次数甚至于播放的机器等等，这对被盗版搅得焦头乱额的音乐公司来说可是一个福音，另外 WMA 还支持音频流(Stream)技术，适合在网络上在线播放，作为微软抢占网络音乐的开路先锋可以说是技术领先、风头强劲，更方便的是不用象 MP3 那样需要安装额外的播放器，而 Windows 操作系统和 Windows Media Player 的无缝捆绑让你只要安装了 windows 操作系统就可以直接播放 WMA 音乐，新版本的 Windows Media Player7.0 更是增加了直接把 CD 光盘转换为 WMA 声音格式的功能，在新出品的操作系统 Windows XP 中，WMA 是默认的编码格式，大家知道 Netscape 的遭遇，现在“狼”又来了。WMA 这种格式在录制时可以对音质进行调节。同一格式，音质好的可与 CD 媲美，压缩率较高的可用于网络广播。虽然现在网络上还不是很流行，但是在微软的大规模推广下已经是得到了越来越多站点的承认和大力支持，在网络音乐领域中直逼\*.mp3，在网络广播方面，也正在瓜分 Real 打下的天下。因此，几乎所有的音频格式都感受到了 WMA 格式的压力。



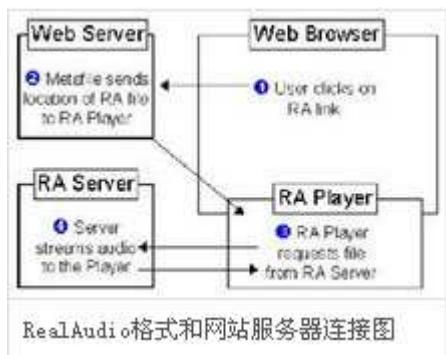
微软官方宣布的资料中称 WMA 格式的可保护性极强，甚至可以限定播放机器、播放时间及播放次数，具有相当的版权保护能力。应该说，WMA 的推出，就是针对 MP3 没有版权限制的缺点而来——普通用户可能很欢迎这种格式，但作为版权拥有者的唱片公司来说，它们更喜欢难以复制拷贝的音乐压缩技术，而微软的 WMA 则照顾到了这些唱片公司的需求。

除了版权保护外，WMA 还在压缩比上进行了深化，它的目标是在相同音质条件下文件体积可以变的更小（当然，只在 MP3 低于 192KBPS 码率的情况下有效，实际上当采用 LAME 算法压缩 MP3 格式时，高于 192KBPS 时普遍的反映是 MP3 的音质要好于 WMA）。

### RealAudio

RealAudio 主要适用于在网络上的在线音乐欣赏，现在大多数的用户仍然在使用 56Kbps 或更低速率的 Modem，

所以典型的回放并非最好的音质。有的下载站点会提示你根据你的 Modem 速率选择最佳的 Real 文件。real 的文件格式主要有这么几种：有 RA (RealAudio)、RM (RealMedia, RealAudio G2)、RMX (RealAudio Secured)，还有更多。这些格式的特点是可以随网络带宽的不同而改变声音的质量，在保证大多数人听到流畅声音的前提下，令带宽较富裕的听众获得较好的音质。



近来随着网络带宽的普遍改善，Real 公司正推出用于网络广播的、达到 CD 音质的格式。如果你的 RealPlayer 软件不能处理这种格式，它就会提醒你下载一个免费的升级包。许多音乐网站 提供了歌曲的 Real 格式的试听版本。现在最新的版本是 RealPlayer 9.0，第 39 期《电脑报》也对 RealPlayer 9.0 作了详细的介绍，这里不再赘述。

### VQF

雅马哈公司另一种格式是\*.vqf，它的核心是减少数据流量但保持音质的方法来达到更高的压缩比，VQF 的音频压缩率比标准的 MPEG 音频压缩率高出近一倍，可以达到 18:1 左右甚至更高。也就是说把一首 4 分钟的歌曲 (WAV 文件) 压成 MP3，大约需要 4MB 左右的硬盘空间，而同一首歌曲，如果使用 VQF 音频压缩技术的话，那只需要 2MB 左右的硬盘空间。因此，在音频压缩率方面，MP3 和 RA 都不是 VQF 的对手。相同情况下压缩后 VQF 的文件体积比 MP3 小 30%~50%，更便于网上传播，同时音质极佳，接近 CD 音质(16 位 44.1kHz 立体声)。可以说技术上也是很先进的，但是由于宣传不力，这种格式难有用武之地。\*.vqf 可以用雅马哈的播放器播放。同时雅马哈也提供从\*.wav 文件转换到\*.vqf 文件的软件。此文件缺少特点外加缺乏宣传。

当 VQF 以 44KHz、80kbit/s 的音频采样率压缩音乐时，它的音质优于 44KHz、128kbit/s 的 MP3，当 VQF 以 44KHz、96kbit/s 的频率压缩时，它的音质几乎等于 44KHz、256kbit/s 的 MP3。经 SoundVQ 压缩后的音频文件在进行回放效果试听时，几乎没有能听出它与原音频文件的差异。

### VQF 音频文件个格式

播放 VQF 对计算机的配置要求仅为奔腾 75 或更高，当然如果您用奔腾 100 或以上的机器，VQF 能够运行得更加出色。实际上，播放 VQF 对 CPU 的要求仅比 Mp3 高 5~10%左右。

VQF 即 TwinVQ 技术虽然是由 NTT 和 YAMAHA 开发的，但它们的应用软件都是免费的。只是 NTT 和 YAMAHA 并没有公布 VQF 的源代码。

### OggVorbis

OggVorbis 是一种新的音频压缩格式，类似于 MP3 等现有的音乐格式。但有一点不同的是，它是完全免费、开放和没有专利限制的。Vorbis 是这种音频压缩机制的名字，而 Ogg 则是一个计划的名字，该计划意图设计一个完全开放性的多媒体系统。目前该计划只实现了 OggVorbis 这一部分。

OggVorbis 文件的扩展名是\*.OGG。这种文件的设计格式是非常先进的。这种文件格式可以不断地进行大小和音质的改良，而不影响旧有的编码器或播放器。

VORBIS 采用有损压缩，但通过使用更加先进的声学模型去减少损失，因此，同样位速率(BitRate)编码的 OGG 与 MP3 相比听起来更好一些。另外，还有一个原因，MP3 格式是受专利保护的。如果你想使用 MP3 格式发布自己的作品，则需要付给 Fraunhofer (发明 MP3 的公司) 专利使用费。而 VORBIS 就完全没有这个问题。

对于乐迷来说，使用 OGG 文件的显著好处是可以用更小的文件获得优越的声音质量。而且，由于 OGG 是完全开放和免费的，制作 OGG 文件将不受任何专利限制，可望可以获得大量的编码器和播放器。这也是为何现在 MP3 编码器如此少而且大多是商业软件的原因，因为 Fraunhofer 要收取专利使用费。Vorbis 使用了与 MP3 相比完全不同的数学原理，因此在压缩音乐时受到的挑战也不同。同样位速率编码的 Vorbis 和 MP3 文件具有同等的声音质量。

Vorbis 具有一个设计良好、灵活的注释，避免了象 MP3 文件的 ID3 标记那样烦琐的操作；Vorbis 还具有位速率缩放：可以不用重新编码便可调节文件的位速率。Vorbis 文件可以被分成小块并以样本粒度进行编辑；Vorbis 支持多通道；Vorbis 文件可以以逻辑方式相连接等。

## AMR

AMR 全称 Adaptive Multi-Rate，自适应多速率编码，主要用于移动设备的音频，压缩比比较大，但相对其他的压缩格式质量比较差，由于多用于人声，通话，效果还是很不错的。

### 分类

1. AMR：又称为 AMR-NB，相对于下面的 WB 而言，语音带宽范围：300—3400Hz，8KHz 抽样
2. AMR-WB:AMR WideBand，

语音带宽范围： 50—7000Hz 16KHz 抽样

“AMR-WB”全称为“Adaptive Multi-rate - Wideband”，即“自适应多速率宽带编码”，采样频率为 16kHz，是一种同时被国际标准化组织 ITU-T 和 3GPP 采用的宽带语音编码标准，也称为 G722.2 标准。AMR-WB 提供语音带宽范围达到 50~7000Hz，用户可主观感受到话音比以前更加自然、舒适和易于分辨。

与之作比较，现在 GSM 用的 EFR(Enhanced Full Rate，增强型全速率编码)采样频率为 8kHz，语音带宽为 200~3400Hz。

AMR-WB 应用于窄带 GSM(全速信道 16k，GMSK)的优势在于其可采用从 6.6kb/s, 8.85kb/s 和 12.65kb/s 三种编码，当网络繁忙时 C/I 恶化，编码器可以自动调整编码模式，从而增强 QoS。在这种应用中，AMR-WB 抗扰度优于 AMR-NB。

AMR-WB 应用于 EDGE、3G 可充分体现其优势。足够的传输带宽保证 AMR-WB 可采用从 6.6kb/s 到 23.85kb/s 共九种编码，语音质量超越 PSTN 固定电话。

## 1.2.2 比较

作为数字音乐文件格式的标准，WAV 格式容量过大，因而使用起来很不方便。因此，一般情况下我们把它压缩为 MP3 或 WMA 格式。压缩方法有无损压缩，有损压缩，以及混成压缩。MPEG,JPEG 就属于混成压缩，如果把压缩的数据还原回去，数据其实是不一样的。当然，人耳是无法分辨的。因此，如果把 MP3，OGG 格式从压缩的状态还原回去的话，就会产生损失。然而，APE 格式即使还原，也能毫无损失地保留原有音质。所以，APE 可以无损失高音质地压缩和还原。在完全保持音质的前提下，APE 的压缩容量有了适当的减小。拿一个最为常见的 38MBWAV 文件为例，压缩为 APE 格式后为 25MB 左右，比开始足足少了 13MB。而且 MP3 容量越来越大的今天，25M 的歌曲已经算不上什么庞然大物了。以 1GB 的 mp3 来说可以放入 4 张 CD，那就是 40 多首歌曲，已经足够了！

MP3 支持格式有 MP3 和 WMA。MP3 由于是有损压缩，因此讲求采样率，一般是 44.1KHZ。另外，还有比特率，即数据流，一般为 8---320KBPS。在 MP3 编码时，还看看它是否支持可变比特率（VBR），现在出的 MP3 机大部分都支持，这样可以减小有效文件的体积。WMA 则是微软力推的一种音频格式，相对来说要比 MP3 体积更小。

## 1.3 字幕格式

### 1.3.1 外挂字幕与内嵌字幕的阐述

外挂字幕：是视频文件和字幕文件分离，在播放的时候要导入字幕文件。比如 DVD 就会自动导入字幕。外挂字幕的好处是：可以导入自己国家的语言。

内嵌字幕：视频文件和字幕文件已经集成到了一起，没有办法改变和去掉了。

## 1.3.2 外挂字幕视频与内嵌字幕视频的画面比较

外挂字幕相对于内嵌字幕来说对视频的质量损害就会小很多，外挂的意思就是在视频之外单独运行的一种字幕文件，对视频本身的分辨率损害很小甚至为零。而内嵌的字面意思就是将视频连带外挂字幕用专有的录制软件重新将视频录制一遍，成为一个新的视频；这种方法虽然解决了视频体积过大和播放器不兼容等问题，但是在重新录制视频过程当中会无意识的损害原视频本身的码率，使重新录制出来的视频分辨率大大不如原视频，所以在选择外挂与内嵌字幕时需结合自身情况考虑视频需要进行选择。

## 1.3.3 外挂字幕的三种格式

- 1、srt 格式：这是最好的，体积小，用记事本可以打开编辑。
- 2、sub+idx：这种是图形字幕，只能用字幕转换软件；体积较大。
- 3、ass 字幕：网上比较少，比 srt 多一些特效。

外挂字幕的一些基本注意事项：

使用外挂字幕的时候，要保证字幕文件和视频文件放置在同一个文件夹下，并且保证两者的文件名相同，但是不要修改后缀和标识（常见的标识有 chs、GB，cht，Big5，eng 五种；其中 chs 和 GB 表示简体中文，cht 和 Big5 表示繁体中文，eng 表示英文）：

例如：

视频的文件名为：越狱（13）.avi

外挂字幕的文件名就应为：越狱（13）.chs.srt

当然，能在视频中显示字幕的前提是你的电脑里安装有字幕插件。否则建议安装能够完美解码的万能播放器。

## 1.4 采集录制和播放渲染

### 1.4.1 视频采集

视频采集（Video Capture）把模拟视频转换成数字视频，并按数字视频文件的格式保存下来。所谓视频采集就是将模拟摄像机、录像机、LD 视盘机、电视机输出的视频信号，通过专用的模拟、数字转换设备，转换为二进制数字信息的过程。在视频采集工作中，视频采集卡是主要设备，它分为专业和家用两个级别。专业级视频采集卡不仅可以进行视频采集，并且还可以实现硬件级的视频压缩和视频编辑。家用级的视频采集卡只能做到视频采集和初步的硬件级压缩，而更为“低端”的电视卡，虽可进行视频的采集，但它通常都省却了硬件级的视频压缩功能。

#### 视频保存格式

影片拍好了，可以直接放在 DV 带上保存，以后就用 DV 机回放，也可以采集到计算机里，编辑后回录到 DV 带上，还可以采集到计算机里，直接把 DVAVI 文件刻到 CDR 上去保存，也可以压缩成 MPG，刻成 VCD 或者 SVCD，DVD 和 CD 保存。MPG 是有损压缩，不管是压缩成什么格式，对画质都有损失，但是刻 MPG 盘保存还是最常用的方式。



DV 影片的回放在电视机上的表现远强于在 CRT 上的表现，尽管 CRT 的分辨率要高得多，主要是因为电视的设计就是为了显示动态画面，所以在亮度、色彩鲜艳上都比显示静态为主的 CRT 要好，而普通电视的显示分辨率只有 320 线，那么 DV 的高达  $720 \times 576$  的分辨率根本用不着，不管是 VCD 的  $352 \times 288$  还是 SVCD 的  $480 \times 576$  都够了，所以尽管压缩成 MPG 画质有损失，但是在电视上基本是看不出来的。在电脑上看，SVCD 的分辨率也足够清晰了。**保存格式的优劣性**

DV 带的保存是个问题，毕竟是磁带，DV 带还用得时间不长，但是以前的录音机磁带时间长了粘连和发霉大家估计都见过的。而 CDR 光盘蓝盘、绿盘在一般情况下不磨损光盘一般是保存 30~50 年，金盘号称能保存 100 年，虽然光盘也有发霉的可能，但是毕竟好得多。

播放的方便性上，也是光盘强，DV 带就得把 DV 机搬出来，还只能在电视上看，对磁头也是个磨损，倒带也很麻烦，而 VCD、SVCD 光盘方便。

## 1.4.2 视频录制

## 1.4.3 视频渲染

渲染，英文为 Render，也有的把它称为着色，但我更习惯把 Shade 称为着色，把 Render 称为渲染。因为 Render 和 Shade 这两个词在三维软件中是截然不同的两个概念，虽然它们的功能很相似，但却有不同。Shade 是一种显示方案，一般出现在三维软件的主要窗口中，和三维模型的线框图一样起到辅助观察模型的作用。很明显，着色模式比线框模式更容易让我们理解模型的结构，但它只是简单的显示而已，数字图像中把它称为明暗着色法。在像 Maya 这样的高级三维软件中，还可以用 Shade 显示出简单的灯光效果、阴影效果和表面纹理效果，当然，高质量的着色效果是需要专业三维图形显示卡来支持的，它可以加速和优化三维图形的显示。但无论怎样优化，它都无法把显示出来的三维图形变成高质量的图像，这是因为 Shade 采用的是一种实时显示技术，硬件的速度限制它无法实时地反馈出场景中的反射、折射等光线追踪效果。而现实工作中我们往往要把模型或者场景输出成图像文件、视频信号或者电影胶片，这就必须经过 Render 程序。

Shade 窗口，提供了非常直观、实时的表面基本着色效果，根据硬件的能力，还能显示出纹理贴图、光源影响甚至阴影效果，但这一切都是粗糙的，特别是在没有硬件支持的情况下，它的显示甚至会是无理无序的。Render 效果就不同了，它是基于一套完整的程序计算出来的，硬件对它的影响只是一个速度问题，而不会改变渲染的结果，影响结果的是看它是基于什么程序渲染的，比如是光影追踪还是光能传递。

### 渲染过程

首先，必须定位三维场景中的摄像机，这和真实的摄影是一样的。一般来说，三维软件已经提供了四个默认的摄像机，那就是软件中四个主要的窗口，分为顶视图、正视图、侧视图和透视图。我们大多数时候渲染的是透视图而不是其它视图，透视图的摄像机基本遵循真实摄像机的原理，所以我们看到的结果才会和真实的三维世界一样，具备立体感。接下来，为了体现空间感，渲染程序要做一些“特殊”的工作，就是决定哪些物体在前面、哪些物体在后面和哪些物体被遮挡等。空间感仅通过物体的遮挡关系是不能完美再现的，很多初学三维的人只注意立体感的塑

## 《FFmpeg 基础库编程开发》

造而忽略了空间感。要知道空间感和光源的衰减、环境雾、景深效果都是有着密切联系的。

渲染程序通过摄像机获取了需要渲染的范围之后，就要计算光源对物体的影响，这和真实世界的情况又是一样的。许多三维软件都有默认的光源，否则，我们是看不到透视图中的着色效果的，更不要说渲染了。因此，渲染程序就是要计算我们在场景中添加的每一个光源对物体的影响。和真实世界中光源不同的是，渲染程序往往要计算大量的辅助光源。在场景中，有的光源会照射所有的物体，而有的光源只照射某个物体，这样使得原本简单的事情又变得复杂起来。在这之后，还要是使用深度贴图阴影还是使用光线追踪阴影？这往往取决于在场景中是否使用了透明材质的物体计算光源投射出来的阴影。另外，使用了面积光源之后，渲染程序还要计算一种特殊的阴影——软阴影（只能使用光线追踪），场景中的光源如果使用了光源特效，渲染程序还将花费更多的系统资源来计算特效的结果，特别是体积光，也称为灯光雾，它会占用大量的系统资源，使用的时候一定要注意。

在这之后，渲染程序还要根据物体的材质来计算物体表面的颜色，材质的类型不同，属性不同，纹理不同都会产生各种不同的效果。而且，这个结果不是独立存在的，它必须和前面所说的光源结合起来。如果场景中有粒子系统，比如火焰、烟雾等，渲染程序都要加以“考虑”。

### 数字影片的后期处理

对录制完成的数字影片进行了剪接、加效果、加字幕、音乐等后期制作，当生成影片时需要将后加入的素材融合到影片中并压缩成为影片最终格式。这个一般都是这样，只是因环境的不同而不同。

#### 渲染滤镜

“**渲染**”滤镜在图像中创建云彩图案、折射图案和模拟的光反射。也可在 3D 空间中操纵对象，并从灰度文件创建纹理填充以产生类似 3D 的光照效果。

##### 1、分层云彩

使用随机生成的介于前景色与背景色之间的值，生成云彩图案。此滤镜将云彩数据和现有的像素混合，其方式与“差值”模式混合颜色的方式相同。第一次选取此滤镜时，图像的某些部分被反相为云彩图案。应用此滤镜几次之后，会创建出与大理石的纹理相似的凸缘与叶脉图案。

##### 2、光照效果

使您可以通过改变 17 种光照样式、3 种光照类型和 4 套光照属性，在 RGB 图像上产生无数种光照效果。还可以使用灰度文件的纹理（称为凹凸图）产生类似 3D 的效果，并存储您自己的样式以在其它图像中使用。

##### 3、镜头光晕

模拟亮光照射到像机镜头所产生的折射。通过点按图像缩览图的任一位置或拖移其十字线，指定光晕中心的位置。

##### 4、纹理填充

用灰度文件或其中的一部分填充选区。若要将纹理添加到文档或选区，请打开要用作纹理填充的灰度文档。并将它装入要进行纹理填充的图像的某一通道中（新建），执行完效果后，可以看到灰度图浮凸在该图像中的效果。

##### 5、云彩

使用介于前景色与背景色之间的随机值，生成柔和的云彩图案。若要生成色彩较为分明的云彩图案，请按住 Alt 键并选取“滤镜/渲染/云彩”命令。

#### 【Proe 中的渲染】

Pro / E 提供了制作高质量图像的渲染工具，能使零件或装配的显现近乎于照片。使用 Pro/E 的渲染功能，给予各零件色彩及相应的透明度，可是所设计的产品立体分明，更具视觉效果。而不必通过产生样机或实物模型来比较外观。特别是值入了 CDRS2001 里的高级渲染功能 Photolux，增加渲染的特殊效果而设的指令，可以做出雾效和透镜闪光等效果。可以将产品模型置于特定的环境，比如室内，你可以在此设置地板、四壁和天花板的背景，可对背景进行预览、尺寸和位置的调整；可以在特征或某个表面上设置材质，定义表面颜色、透明度、粗糙度和纹理等；另外，运用贴图功能在产品和包装上生成和附加常规的标记和图案，指定每个图案的大小、位置和透明度；指定光线类型颜色和强度，方便地选择和控制阴影的形式。

## 1.5 编解码器

编解码器（codec）指的是一个能够对一个信号或者一个数据流进行变换的设备或者程序。这里指的变换既包括将信号或者数据流进行编码（通常是为了传输、存储或者加密）或者提取得一个编码流的操作，也包括为了观察或者处理从这个编码流中恢复适合观察或操作的形式的操作。编解码器经常用在视频会议和流媒体等应用中，通常主要还是用在广电行业，作前端应用。

经过编码的音频或者视频原始码流经常被叫做“Essence”（有译作“本体”，“精”），以区别于之后加入码流的元信息和其它用以帮助访问码流和增强码流鲁棒性的数据。

大多数编解码器是有损的，目的是为了得到更大的压缩比和更小的文件大小。当然也有无损的编解码器，但是通常没有必要为了一些几乎注意不到的质量损失而大大增加编码后文件的大小。除非该编码的结果还将在以后进行下一步的处理，此时连续的有损编码通常会带来较大的质量损失。

很多多媒体数据流需要同时包含音频数据和视频数据，这时通常会加入一些用于音频和视频数据同步的元数据。这三种数据流可能会被不同的程序，进程或者硬件处理，但是当它们传输或者存储的时候，这三种数据通常是被封装在一起的。通常这种封装是通过视频文件格式来实现的，例如常见的\*.mpg, \*.avi, \*.mov, \*.mp4, \*.rm, \*.ogg or \*.tta. 这些格式中有些只能使用某些编解码器，而更多可以以容器的方式使用各种编解码器。

编解码器对应的英文“codec”（coder 和 decoder 简化而成的合成词语）和 decode 通常指软件，当特指硬件的时候，通常使用“endec”这个单词。

硬件编解码器有标清编解码器和高清编解码器。所谓标清，英文为“Standard Definition”，是物理分辨率在 720p 以下的一种视频格式。720p 是指视频的垂直分辨率为 720 线逐行扫描。具体的说，是指分辨率在 400 线左右的 VCD、DVD、电视节目等“标清”视频格式，即标准清晰度。而物理分辨率达到 720p 以上则称作为高清，（英文表述 High Definition）简称 HD。关于高清的标准，国际上公认的有两条：视频垂直分辨率超过 720p 或 1080i；视频宽纵比为 16: 9。

## 1.6 容器和协议

### 1.6.1 容器格式和编码格式

#### 1.6.1.1 简介

音频视频编码及文件格式（容器）是一个很庞大的知识领域，完整的说清楚，那就需要些写成一本教材了。这里先就几个简单的概念问题作以介绍：

首先要分清楚媒体文件和编码的区别：

文件是既包括视频又包括音频、甚至还带有脚本的一个集合，也可以叫容器；

文件当中的视频和音频的压缩算法才是具体的编码。

也就是说一个.avi 文件，当中的视频可能是编码 a，也可能是编码 b，音频可能是编码 5，也可能是编码 6，具体的用那种编码的解码器，则由播放器按照 avi 文件格式读取信息去调用了。

音频视频编码方案有很多，用百家争鸣形容不算过分，目前常见的音频视频编码有以下几类：

◆ **MPEG 系列：**（由 ISO[国际标准组织机构]下属的 MPEG[运动图象专家组]开发）

视频编码方面主要是 Mpeg1（vcd 用的就是它）、Mpeg2（DVD 使用）、Mpeg4（现在的 DVDRIP 使用的都是它的变种，如：divx, xvid 等）、Mpeg4 AVC（现在正热门）；

音频编码方面主要是 MPEG Audio Layer 1/2、MPEG Audio Layer 3（大名鼎鼎的 mp3）、MPEG-2 AAC 、MPEG-4

## 《FFmpeg 基础库编程开发》

AAC 等等。注意：DVD 音频没有采用 Mpeg 的

◆ H.26X 系列：（由 ITU[国际电传视讯联盟]主导，侧重网络传输，注意：只是视频编码）

包括 H261、H262、H263、H263+、H263++、H264（就是 MPEG4 AVC-合作的结晶）

◆ 微软 windows media 系列：（公司牛，能自己定标准啊...）

视频编码有 Mpeg-4 v1/v2/v3（基于 MPEG4，DIVX3 的来源，呵呵）、Windows Media Video 7/8/9/10

音频编码有 Windows Media audio v1/v2/7/8/9

◆ Real Media 系列：（注意，这里说的 Real 的编码，可不是 rm、rmvb 文件，呵呵）

视频编码有 RealVideo G2（早期）、RealVideo 8/9/10

音频编码有 RealAudio cook/sipro（早期）、RealAudio AAC/AACPlus 等

◆ QuickTime 系列：（是一个平台，有很多编码器）

视频编码有 Sorenson Video 3（用于 QT5，成标准了）、Apple MPEG-4、Apple H.264

音频编码有 QDesign Music 2、Apple MPEG-4 AAC（这个不错）

其它，如：Ogg、On2-vpx、flash video：不详述啦。

特殊说明的，是 DVD 这种媒介的音频编码，采用了相对独立的几种，就列 2 个常见的吧：AC3（杜比公司开发）、

DTS 文件格式（容器）：

◆ AVI

音视频交互存储，最常见的音频视频容器。支持的视频音频编码也是最多的。

◆ MPG

◆ MPEG 编码采用的音频视频容器，具有流的特性。里面又分为 PS，TS 等，PS 主要用于 DVD 存储，TS 主要用于 HDTV。

◆ VOB

DVD 采用的音频视频容器格式（即视频 MPEG-2，音频用 AC3 或者 DTS），支持多视频多音轨多字幕章节等。

◆ MP4

MPEG-4 编码采用的音频视频容器，基于 QuickTime MOV 开发，具有许多先进特性。

◆ 3GP

3GPP 视频采用的格式，主要用于流媒体传送。

◆ ASF

Windows Media 采用的音频视频容器，能够用于流传送，还能包容脚本等。

◆ RM

RealMedia 采用的音频视频容器，用于流传送。

注意：RMVB，是视频编码部分采用可变码率压缩的文件格式（容器）

◆ MOV

QuickTime 的音频视频容器，恐怕也是现今最强大的容器，甚至支持虚拟现实技术，Java 等，它的变种 MP4,3GP 都没有这么厉害。

◆ MKV

MKV 它能把 Windows Media Video，RealVideo，MPEG-4 等视频音频融为一个文件，而且支持多音轨，支持章节字幕等。

◆ WAV

一种音频容器（注意：只是音频），大家常说的 WAV 就是没有压缩的 PCM 编码，其实 WAV 里面还可以包括 MP3 等其他 ACM 压缩编码。

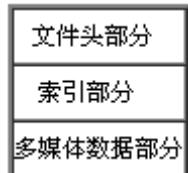
◆ MP3

如前所述，不用多说了吧？就是 MPEG Audio Layer 3（Mpeg 1 的音频编码的一种）

文件转换（实际上也是编码转换）

### 1.6.1.2 多媒体容器文件格式

多媒体容器文件格式一般都包括文件头部分、索引部分和多媒体数据部分（如图 1 所示）。



文件头部分

索引部分

多媒体数据部分文件头部分说明了多媒体数据符合的压缩标准及规范信息，多媒体数据符合的规范信息可以包括视频的分辨率、帧率，音频的采样率等。

索引部分：由于多媒体数据通常会被分成若干块，各块数据之间也可能是不连续存储的，因此需要在索引部分建立多媒体数据的存储位置索引（如图 2 所示），其详细显示了视频数据存储位置索引，用来记录相应数据块的存储位置的偏移量，由于各数据块的大小可能不同，因此也可能需要在索引部分建立各种多媒体数据块的尺寸大小索引，用来记录相应数据块的尺寸大小。此外在索引部分还建立了其他索引，比如音视频同步索引等等。PC 上播放这些多媒体容器文件时，一般是将索引一次性的全部放到内存中，然后在播放中根据操作（快进、快退等）来通过数据索引得到所需的数据。这个貌似和项目里面的视频信息文件的作用类似~~~

多媒体数据部分就是经过压缩的多媒体数据，包括视频数据、音频数据、文本数据及其他多媒体数据。

视频数据存储位置索引	音频数据存储位置索引
offset1	offset1
offset2	offset2
.....	.....
offsetN	offsetN

### 1.6.1.3 音频编解码格式

a) 音频编解码格式

\*MPEG Audio Layer 1/2

\*MPEG Audio Layer 3(MP3)

\*MPEG2 AAC

\*MPEG4 AAC

\*Windows Media audio v1/v2/7/8/9

\*RealAudio cook/sipro(real media array)

\*RealAudio AAC/AACPlus(real media series)

\*QDesign Music 2(apple series)

是 QDesign 公司开发的用于高保真高压缩率的编码方式，类似于 MP3，不过比 MP3 要先进。支持流式播放。

\*Apple MPEG-4 AAC(apple series)

## 《FFmpeg 基础库编程开发》

- \*ogg(ogg vorbis 音频)
- \*AC3(DVD 专用音频编码)
- \*DTS(DVD 专用音频编码)
- \*APE(monkey's 音频)
- \*AU(sun 格式)
- \*FLAC(fress lossless 音频)
- \*M4A(mpeg-4 音频) (苹果改用的名字，可以改成.mp4)
- \*MP2(mpeg audio layer2 音频)

\*WMA

b) 视频编解码格式

- \*MPEG1(VCD)
- \*MPEG2(DVD)
- \*MPEG4(divx,xvid)
- \*MPEG4 AVC/h.264
- \*h.261
- \*h.262
- \*h.263
- \*h.263+
- \*h.263++
- \*MPEG-4 v1/v2/v3(微软 windows media 系列)

\*Windows Media Video 7/8/9/10

\*Sorenson Video 3 (用于 QT5, 成标准了) (apple series)

\*RealVideo G2(real media series)

\*RealVideo 8/9/10(real media series)

\*Apple MPEG-4(apple series)

\*Apple H.264(apple series)

\*flash video

c) 音视频文件格式

首先要分清楚媒体文件和编码的区别：文件是既包括视频又包括音频、甚至还带有脚本的一个集合，也可以叫容器；文件当中的视频和音频的压缩算法才是具体的编码。

\*AVI

音视频交互存储，最常见的音频视频容器。支持的视频音频编码也是最多的

\*MPG

MPEG 编码采用的音频视频容器，具有流的特性。里面又分为 PS, TS 等，PS 主要用于 DVD 存储，TS 主要用于 HDTV。

\*VOB

DVD 采用的音频视频容器格式（即视频 MPEG-2，音频用 AC3 或者 DTS），支持多视频多音轨多字幕章节等。

\*MP4

MPEG-4 编码采用的音频视频容器，基于 QuickTime MOV 开发，具有许多先进特性。

\*3GP

3GPP 视频采用的格式，主要用于流媒体传送。

\*ASF

Windows Media 采用的音频视频容器，能够用于流传送，还能包容脚本等。

\*RM

RealMedia 采用的音频视频容器，用于流传送。

\*MOV

QuickTime 的音频视频容器,恐怕也是现今最强大的容器,甚至支持虚拟现实技术,Java 等,它的变种 MP4,3GP 都没有这么厉害。

\*MKV

MKV 它能把 Windows Media Video, RealVideo, MPEG-4 等视频音频融为一个文件,而且支持多音轨,支持章节字幕等。

\*WAV

一种音频容器(注意:只是音频),大家常说的 WAV 就是没有压缩的 PCM 编码,其实 WAV 里面还可以包括 MP3 等其他 ACM 压缩编码。

d) 音视频技术

VCD

DVD

DVD 目录是如何工作的

Audio CD

\*标准 CD 格式也就是 44.1K 的采样频率,速率 88K/秒,16 位量化位数

\* \*.cda 格式,这就是 CD 音轨了,一个 CD 音频文件是一个 \*.cda 文件,这只是一个索引信息,并不是真正的包含声音信息,所以不论 CD 音乐的长短,在电脑上看到的“\*.cda 文件”都是 44 字节长

MP3

\*MPEG 音频文件的压缩是一种有损压缩,MPEG3 音频编码具有 10: 1~12: 1 的高压缩率,同时基本保持低音频部分不失真,但是牺牲了声音文件中 12KHz 到 16KHz 高音频这部分的质量来换取文件的尺寸,相同长度的音乐文件,用 \*.mp3 格式来储存,一般只有 \*.wav 文件的 1/10,而音质要次于 CD 格式或 WAV 格式的声音文件

\*MP3 格式压缩音乐的采样频率有很多种,可以用 64Kbps 或更低的采样频率节省空间,也可以用 320Kbps 的标准达到极高的音质

\*每分钟音乐的 MP3 格式只有 1MB 左右大小

MIDI:

经常玩音乐的人应该常听到 MIDI (Musical Instrument Digital Interface) 这个词,MIDI 允许数字合成器和其他设备交换数据。MID 文件格式由 MIDI 继承而来。MID 文件并不是一段录制好的声音,而是记录声音的信息,然后在告诉声卡如何再现音乐的一组指令。这样一个 MIDI 文件每存 1 分钟的音乐只用大约 5~10KB。今天,MID 文件主要用于原始乐器作品,流行歌曲的业余表演,游戏音轨以及电子贺卡等。\*.mid 文件重放的效果完全依赖声卡的档次。\*.mid 格式的最大用处是在电脑作曲领域。\*.mid 文件可以用作曲软件写出,也可以通过声卡的 MIDI 口把外接音序器演奏的乐曲输入电脑里,制成 \*.mid 文件。

WMA:

\*WMA 的压缩率一般都可以达到 1: 18 左右,WMA 的另一个优点是内容提供商可以通过 DRM (Digital Rights Management) 方案如 Windows Media Rights Manager 7 加入防拷贝保护。这种内置了版权保护技术可以限制播放时间和播放次数甚至于播放的机器等等,这对被盗版搅得焦头乱额的音乐公司来说可是一个福音,另外 WMA 还支持音频流(Stream)技术,适合在网络上在线播放

\* WMA 这种格式在录制时可以对音质进行调节。同一格式,音质好的可与 CD 媲美,压缩率较高的可用于网络广播

e) 以文件名标识识别音频编码格式

\*.aac

音频编码: aac

\*.ac3

音频编码: ac3

\*.ape

\*.au

## 《FFmpeg 基础库编程开发》

音频编码: pcm\_s16be

\*.m4a

音频编码: mpeg4 aac

\*.mp2

\*.mp3

\*.ogg

音频编码: vorbis

\*.wav

音频编码: pcm\_s16le

\*.flav

\*.wma

音频编码: wma7x

以文件名标识识别音视频编码格式

1. \*.MP4 (MP4 MPEG-4 视频)

视频编码: mpeg4

音频编码: mpeg4 aac

2 . \*.3gp (3GPP 第三代合作项目)

视频编码: mpeg4

音频编码: amr\_nb((mono, 8000 Hz, Sample Depth 16 morsel, bitrate 12 kbps)

3 . \*.3g2 (3GPP 第三代合作项目 2)

视频编码: mpeg4

音频编码: mpeg4 aac

4. \*.ASF (ASF 高级流格式)

视频编码: msmpeg4

音频编码: mp3

5. \*.avi (AVI 音视频交错格式)

视频编码: mpeg4

音频编码: pcm\_s16le

6. \*.avi (divx 影片)

视频编码: mpeg4

音频编码: mp3

7. \*.avi (xvid 视频)

视频编码: Xvid

音频编码: mp3

8. \*.vob (DVD)

视频编码: mpeg2 video

音频编码: ac3

9. \*.flv (flash 视频格式)

视频编码:

音频编码: mp3

10. \*.mp4 (iPod 320\*240 MPEG-4 视频格式)

视频编码: mpeg4

音频编码: mpeg4 aac

11. \*.mp4(iPod video2 640\*480 MPEG-4 视频格式)

视频编码: mpeg4

音频编码: mpeg4 aac

12. \*.mov (MOV 苹果 quicktime 格式)

视频编码: mpeg4\_qt

音频编码: mpeg4 aac\_qt

13. \*.mpg (mpeg1 影片)

视频编码: mpeg1 video

音频编码: mp2

14. \*.mpg (mpeg2 影片)

视频编码: mpeg2 video

音频编码: mp2

15. \*.mp4 (mpeg4 avc 视频格式)

视频编码: h.264

音频编码: mpeg4 aac

16. \*.mp4 (PSP mpeg4 影片)

视频编码: Xvid

音频编码: mpeg4 aac

17. \*.mp4 (PSP AVC 视频格式)

视频编码: h.264

音频编码: mpeg4 aac

18. \*.rm (RM realvideo)

视频编码: rv10

音频编码: ac3

19. \*.mpg (超级 VCD)

视频编码: mpeg2 video

音频编码: mp2

20. \*.swf (SWF 格式)

视频编码:

音频编码: mp3

21. \*.mpg (video CD 格式)

视频编码: mpeg1 video

音频编码: mp2

22. \*.vob (mpeg2 ps 格式)

视频编码: mpeg2 video

音频编码: ac3

23. \*.wmv (windows 视频格式)

视频编码: wmv3x

音频编码: wma7x

## 1.6.2 协议

两大标准制定组织

这里的标准，主要指的是音视频压缩标准。两大组织分别是国际标准化组织（iso）和国际电信联盟（itu），相信 it 行业的从业者没听说过这两个行业的人很少。

在音视频压缩标准方面，mpeg 系列的协议是 iso 制定的标准，而 h 系列的协议则是 itu 制定的标准。

### 1.6.2.1 视频协议

目前主要的视频压缩协议有：h.261、h.263、h.264 和 mpeg-1、mpeg-2 和 mpeg-4。第一个视频压缩标准是 h.261，它的算法现在来看，非常简单，但是，它的很多视频压缩的思想，一直影响到现在最新的压缩标准 h.264。h.264 单看名字，感觉是 itu 组织制定的，其实它还有一个名字叫 mpeg-4 part 10，翻译过来叫 mpeg-4 第十部分，这是因为 h.264 是 iso 和 itu 组织共同制定的，版权共享。其实，一直以来，h 系列的标准制定者和 mpeg 系列的标准制定者基本上就是同一群人，而且，这两个系列的算法思想基本上都差不多，唯一有一点不同的协议是 mpeg-4，它在它的高级 profile 中提出了小波变换等算法来实现视频压缩，从实际发展看，个人感觉不是很成功，采用小波变换的商用 codec 很少，这可能和这些算法的达不到实时性有关系。

从应用的角度看，mpeg 系列在消费类应用更广些，大家也更熟悉些，我们熟悉的 vcd 格式视频主要是 mpeg-1，dvd 的视频则是 mpeg-2，早期大家看的电影在电脑上存盘文件格式都是 \*.mpg，基本上也都是 mpeg 做的压缩了。在行业上，国内的监控行业，也是从 mpeg-1 到 mpeg-2，到前两三年的 mpeg-4，再到最近的 h.264。而 h 系列的标准，用得最多的是视频会议，从 h.261 到 h.263，再到 h.263+、h.263++ 等，再到底现在的 h.264。

从技术角度说，h 系列的协议对网络的支持更好些，这点 mpeg 系列要差一些，但是，mpeg 它每一代都比 h 系列同一代的协议要出得晚些，算法也相对更先进些，因此，它用来做存储协议是很合适的，这也就是为什么普通消费类产品用户很少了解到 h 系列协议的原因。

h.264 是两大组织最新的算法成果，它在算法层面应该说是非常先进了，有人评价，h.264 是视频压缩技术的一个里程碑，在可预见的 5 到 10 年内，出现新的视频压缩协议可能性很小，除非压缩理论有重大突破。

中国也有自己的视频压缩协议，叫做 avs，搞了好多年了，不过搞得不是很好。从市场分析，消费类电子、视频会议和流媒体行业，现在要再想进去可能很困难了。不过最近听说 avs 又有点火起来了，有消息称，iptv 指定要支持 avs，这可能是它的最后机会了吧。

除了上面说的协议，还有很多公司也有自己的压缩算法，不过基本上都是不公开的了，他们这些算法也都非常好，不过和开发人员关系倒不是很大了，典型的是微软的 wmv、realplay 公司的 rm 和 rmvb 等，他们的使用者也很多，而且他们都偏向流媒体应用。

### 1.6.2.2 音频协议.

音频协议也分两大类，itu 组织的主要是用于视频会议的 g 系列协议，包括 g.711、g.722、g.723、g.726、g.728、g.729 等。这些协议主要有两大特点，第一是比较关注语音压缩，毕竟开会主要是要听人讲话；对音乐的压缩效果可能就不是太好了；第二是压缩率都比较大，码率都比较低，典型的 g.723 支持 5.9k/s 这样的码率，而且语音音质还很不错。iso 的音频可能更为人熟知一些，最流行的就是 mp3，它的全称是 mpeg-1 audio layer 3，意思是 mpeg-1 的音频第三层；另外，最新的音频算法被称为 aac（也称为 mp4），它定义在 mpeg-2 或 mpeg-4 的音频部分。他们的特点是音质好，支持多声道，高采样精度和采样频率，尤其对音乐的压缩效果比 g 系列要好太多。当然，这也是因为它们的应用领域侧重点不同造成的。

同样的，很多大公司也有自己的语音压缩标准，效果也非常好。不过都是他们自己的知识产权和算法，通用市场用的还是少。

### 1.6.2.3 上层通讯协议

在视频会议系统中，目前最流行的有 h.323 和 sip 协议，在流媒体应用中，isma rtsp 应用得比较多，它属于开源项目，而很多流媒体产商有自己的流媒体传输协议，比如微软的 mms 等。

h.323 主要用于视频会议，被称为协议簇，我们前面提到的 h 系列视频压缩协议和 g 系列音频压缩协议都属于它的子协议。除了音视频编解码器外；它还定义了各种数据应用，包括 t.120、t.84、t.434 等；另外还包括 h.245

控制信道、h.225.0 呼叫信令信道以及 ras 信道。详细的 h.323 的知识，这里就不深入介绍了。

sip 是由 ietf 提出来的一个应用控制（信令）协议。正如名字所隐含的--用于发起会话。它可用来创建、修改以及终结多个参与者参加的多媒體会话进程。参与会话的成员可以通过组播方式、单播连网或者两者结合的形式进行通信。

**h.323 和 sip 分别是通信领域与因特网两大阵营推出的建议。** h.323 企图把 ip 电话当作是众所周知的传统电话，只是传输方式发生了改变，由电路交换变成了分组交换。而 sip 协议侧重于将 ip 电话作为因特网上一个应用，较其实应用（如 ftp, e-mail 等）增加了信令和 qos 的要求，它们支持的业务基本相同，也都利用 rtp 作为媒体传输的协议。但 h.323 是一个相对复杂的协议。

rtsp 主要用于流媒体传输，它的英文全称是 real time streaming protocol。典型的应用就是网络电视的应用，由客户向服务器进行点播，如果在监控行业应用的话，建议当用户进行远程回放录像时，可采用 rtsp 协议。

## 1.7 常用概念介绍

### 1.7.1 硬解

硬件解码：

视频解码分为软解和硬解。

所谓“软解”就是通过软件让 CPU 进行视频解码处理；而“硬解”是指不依赖于 CPU，通过专用的设备（子卡）单独完成视频解码，比如曾经的 VCD/DVD 解压卡、视频压缩卡都被冠以“硬解”的称号。现在实现高清硬解不需要额外的子卡，也不需要额外的投入，因为硬解码模块被整合在了 GPU 内部，而目前主流的显卡（包括整合显卡）都能支持硬解码。

“硬解”其实更需要软件的支持，只是基本不需要 CPU 参与运算，从而为系统节约了很多资源开销。通过降低 CPU 占用率，可以给用户带来很多实惠：

● GPU 硬解码高清视频的优势：

1. 不需要太好的 CPU，单核足矣，CPU 方面节约不少资金；
2. 硬解码基本相当于免费附送，不到 500 元的整合主板都能完美支持；
3. 硬解码让 CPU 占用率超低，系统有能力在看 HDTV 的同时进行多任务操作；
4. CPU 需要倾尽全力才能解码 HDTV，而 GPU 只需动用 0.1 亿晶体管的解码模块就能完成任务，功耗控制更好；

● GPU 硬解码高清视频的劣势：

1. 起步较晚，软件支持度无法与软解相提并论；
2. 面对杂乱无章的视频编码、封装格式，硬解码无法做到全面兼容；
3. 软解拥有大量画面输出补偿及画质增强技术，而硬解这方面做得还远远不够；
4. 硬解码软件设置较为复杂，很多朋友根本不知道该如何正确使用 GPU 硬件解码。

虽然硬解码拥有种种缺点，但依然倍受广大用户追捧，因为低成本和节能环保这两大致命诱惑让人难以抗拒。随着时间的推移，现在硬解码的缺点基本被改进，只是很多人还不懂得如何用好硬解码，本文就通过大量应用案例来释放出硬解码真正的威力！

**解码芯片(又叫解压缩芯片).** 手机播放视频要依赖于解码芯片把画面和声音还原成可以播放的信号，交由显示屏和喇叭(耳机)输出。解码芯片的性能是有局限的，类似于汽车的发动机功率是有极限的。它能够流畅解码的数据，主要受限于以下几个参数和条件。

### 1.7.2 IBP 帧

帧——就是影像动画中最小单位的单幅影像画面，相当于电影胶片上的每一格镜头。而在实际压缩时，会采取各种

算法减少数据的容量，其中 IPB 就是最常见的。

## 1、基本概念

I frame：帧内编码帧 又称 intra picture, I 帧通常是每个 GOP (MPEG 所使用的一种视频压缩技术) 的第一个帧，经过适度地压缩，做为随机访问的参考点，可以当成图象。I 帧可以看成是一个图像经过压缩后的产物。P frame: 前向预测编码帧 又称 predictive-frame，通过充分将低于图像序列中前面已编码帧的时间冗余信息来压缩传输数据量的编码图像，也叫预测帧；

B frame: 双向预测内插编码帧 又称 bi-directional interpolated prediction frame，既考虑与源图像序列前面已编码帧，也顾及源图像序列后面已编码帧之间的时间冗余信息来压缩传输数据量的编码图像，也叫双向预测帧；

PTS: Presentation Time Stamp。PTS 主要用于度量解码后的视频帧什么时候被显示出来

DTS: Decode Time Stamp。DTS 主要是标识读入内存中的 b i t 流在什么时候开始送入解码器中进行解码。

ps:在没有 B 帧存在的情况下 DTS 的顺序和 PTS 的顺序应该是一样的。

## 2、I、B、P 的特点

I 帧特点:

1. 它是一个全帧压缩编码帧。它将全帧图像信息进行 JPEG 压缩编码及传输;
2. 解码时仅用 I 帧的数据就可重构完整图像;
3. I 帧描述了图像背景和运动主体的详情;
4. I 帧不需要参考其他画面而生成;
5. I 帧是 P 帧和 B 帧的参考帧(其质量直接影响到同组中以后各帧的质量);
6. I 帧是帧组 GOP 的基础帧(第一帧),在一组中只有一个 I 帧;
7. I 帧不需要考虑运动矢量;
8. I 帧所占数据的信息量比较大。

P 帧:前向预测编码帧。

P 帧的预测与重构:P 帧是以 I 帧为参考帧,在 I 帧中找出 P 帧“某点”的预测值和运动矢量,取预测差值和运动矢量一起传送。在接收端根据运动矢量从 I 帧中找出 P 帧“某点”的预测值并与差值相加以得到 P 帧“某点”样值,从而可得到完整的 P 帧。

P 帧特点:

1. P 帧是 I 帧后面相隔 1~2 帧的编码帧;
2. P 帧采用运动补偿的方法传送它与前面的 I 或 P 帧的差值及运动矢量(预测误差);
3. 解码时必须将 I 帧中的预测值与预测误差求和后才能重构完整的 P 帧图像;
4. P 帧属于前向预测的帧间编码。它只参考前面最靠近它的 I 帧或 P 帧;
5. P 帧可以是其后面 P 帧的参考帧,也可以是其前后的 B 帧的参考帧;
6. 由于 P 帧是参考帧,它可能造成解码错误的扩散;
7. 由于是差值传送,P 帧的压缩比较高。

B 帧:双向预测内插编码帧。

B 帧的预测与重构

B 帧以前面的 I 或 P 帧和后面的 P 帧为参考帧,“找出”B 帧“某点”的预测值和两个运动矢量,并取预测差值和运动矢量传送。接收端根据运动矢量在两个参考帧中“找出(算出)”预测值并与差值求和,得到 B 帧“某点”样值,从而可得到完整的 B 帧。

B 帧特点

1. B 帧是由前面的 I 或 P 帧和后面的 P 帧来进行预测的;
2. B 帧传送的是它与前面的 I 或 P 帧和后面的 P 帧之间的预测误差及运动矢量;
3. B 帧是双向预测编码帧;
4. B 帧压缩比最高,因为它只反映丙参考帧间运动主体的变化情况,预测比较准确;
5. B 帧不是参考帧,不会造成解码错误的扩散。

注:I、B、P 各帧是根据压缩算法的需要,是人为定义的,它们都是实实在在的物理帧,至于图像

中的哪一帧是 I 帧,是随机的,一但确定了 I 帧,以后的各帧就严格按規定顺序排列

从上面的解释看,我们知道 I 和 P 的解码算法比较简单,资源占用也比较少,I 只要自己完成就行了,P 呢,也只需要解码器把前一个画面缓存一下,遇到 P 时就使用之前缓存的画面就好了,如果视频流只有 I 和 P,解码器可以不管后面的数据,边读边解码,线性前进,大家很舒服。

但网络上的电影很多都采用了 B 帧,因为 B 帧记录的是前后帧的差别,比 P 帧能节约更多的空间,但这样一来,文件小了,解码器就麻烦了,因为在解码时,不仅要用之前缓存的画面,还要知道下一个 I 或者 P 的画面(也就是说要预读预解码),而且,B 帧不能简单地丢掉,因为 B 帧其实也包含了画面信息,如果简单丢掉,并用之前的画面简单重复,就会造成画面卡(其实就是丢帧了),并且由于网络上的电影为了节约空间,往往使用相当多的 B 帧,B 帧用的多,对不支持 B 帧的播放器就造成更大的困扰,画面也就越卡。

一般平均来说,I 的压缩率是 7(跟 JPG 差不多),P 是 20,B 可以达到 50,可见使用 B 帧能节省大量空间,节省出来的空间可以用来保存多一些 I 帧,这样在相同码率下,可以提供更好的画质。

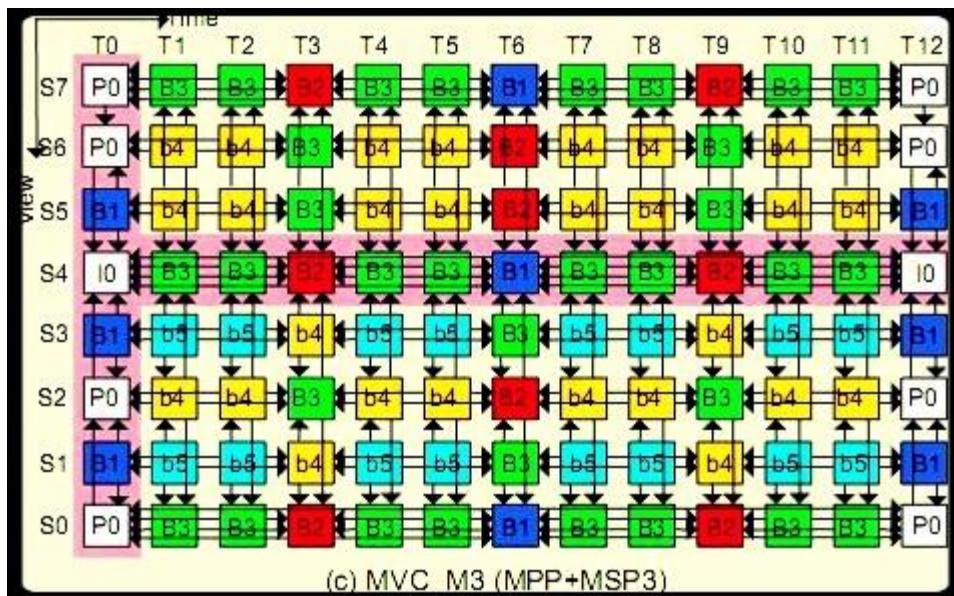
### 3、GOP

GOP: Group of Pictures 画面组

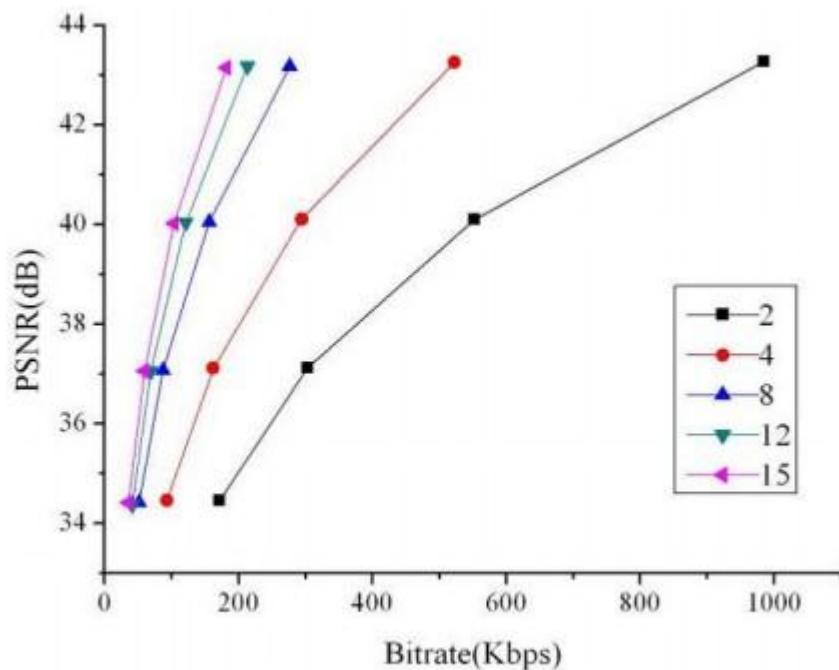
GOP (Group of Pictures) 策略影响编码质量: 所谓 GOP,意思是画面组,一个 GOP 就是一组连续的画面。MPEG 编码将画面(即帧)分为 I、P、B 三种,I 是内部编码帧,P 是前向预测帧,B 是双向内插帧。简单地讲,I 帧是一个完整的画面,而 P 帧和 B 帧记录的是相对于 I 帧的变化。没有 I 帧,P 帧和 B 帧就无法解码,这就是 MPEG 格式难以精确剪辑的原因,也是我们之所以要微调头和尾的原因。MPEG-2 帧结构

MPEG-2 压缩的帧结构有两个参数,一个是 GOP (Group Of Picture) 图像组的长度,一般可按编码方式从 1—15;另一个是 I 帧和 P 帧之间 B 帧的数量,一般是 1—2 个。前者在理论上记录为 N,即多少帧里面出现一次 I 帧;后者描述为多少帧里出现一次 P 帧,记录为 M。

下面举例说明:



在如上图中,GOP (Group of Pictures)长度为 13,S0~S7 表示 8 个视点,T0~T12 为 GOP 的 13 个时刻。每个 GOP 包含帧数为视点数 GOP 长度的乘积。在该图中一个 GOP 中,包含 94 个 B 帧。B 帧占一个 GOP 总帧数的 90.38%。GOP 越长,B 帧所占比例更高,编码的率失真性能越高。下图测试序列 Race1 在不同 GOP 下的率失真性能对比。



Race1 测试序列在不同 GOP 长度的率失真性能对比图

#### 4、mpeg4 视频中 IPB 的判定

mpeg4 的每一帧开头是固定的: 00 00 01 b6, 那么我们如何判断当前帧属于什么帧呢? 在接下来的 2bit, 将会告诉我们答案。注意: 是 2bit, 不是 byte, 下面是各类型帧与 2bit 的对应关系:

00: I Frame

01: P Frame

10: B Frame

为了更好地说明, 我们举几个例子, 以下是 16 进制显示的视频编码:

00 00 01 b6 10 34 78 97 09 87 06 57 87 .....

I 帧

00 00 01 b6 50 78 34 20 cc 66 b3 89 .....

P 帧

00 00 01 b6 96 88 99 06 54 34 78 90 98 .....

B 帧

下面我们来分析一下为什么他们是 I、P、B 帧

$0x10 = 0001\ 0000$

$0x50 = 0101\ 0000$

$0x96 = 1001\ 0100$

大家看红色的 2bit, 再对照开头说的帧与 2bit 的对应关系, 是不是符合了呢?

下面给出一段 c++ 代码供大家参考:

```

1 switch(buf[i] & (BYTE)0xc0)
2 {
3     case 0x00:
4         //I Frame
5         break;
6     case 0x40:
7         //P Frame
8         break;
9     case 0x80:

```

```

10 //B Frame
11 break;
12 default:
13 break; }
```

## 1.7.3 DTS 和 PTS

### 1.7.4 分辨率

这里有 2 个概念， 分别是：

- a. 物理分辨率，即手机屏幕能显示的像素数，用 W x H 个像素表示。常见的手机屏幕分辨率为 320x240(QVGA)，随着大屏幕手机的普及，更高的分辨率也开始出现。例如：480x320(iphone),640x360(nHD, 诺基亚触屏系列常见),640x480(VGA, 多普达系列常见)，甚至高达 852x480(夏普高端手机常见)。
- b. 视频文件的分辨率，这个是指视频画面的实际分辨率，如，320x240, 480x272, 640x480 等等。

一般来说，大部分手机的解码芯片不支持超过其屏幕物理分辨率的视频，部分可以支持超过其屏幕物理分辨率的视频，例如，虽然 iphone 的屏幕物理分辨率为 480x320，但它支持 640x480 的视频，此时播放的画面实际是把原视频缩小的。

### 1.7.5 码率

一般用多少 kbps(千比特/秒)或者 mbps(兆比特/秒)来表示。手机解码芯片所支持的码率一般都在 1Mbps 以下。

### 1.7.6 帧率

(FPS, 帧/秒)，就是视频画面刷新的速度，作为参考，国内电视机一般是 25FPS，电影标准为 24FPS. 手机芯片，最高支持 30FPS，早期型号最大只能 15fps。

### 1.7.7 RGB 和 YUV

RGB 指的是红绿蓝，应用还是很广泛的，比如显示器显示，bmp 文件格式中的像素值等；而 yuv 主要指亮度和两个色差信号，被称为 luminance 和 chrominance 他们的转化关系可以自己去查一下，我们视频里面基本上都是用 yuv 格式。

YUV 文件格式又分很多种，如果算上存储格式，就更多了，比如 yuv444、yuv422、yuv411、yuv420 等等，视频压缩用到的是 420 格式，这是因为人眼对亮度更敏感些，对色度相对要差些。另外要注意几个英文单词的意思，比如：packet、planar、interlace、progressive 等。

### 1.7.8 实时和非实时

实时与非实时 主要用来形容编码器，它含有两个意思，一个是要保证帧率，也就是每秒 25 帧，另一个是“live”的意思，意味着直播，所谓的“实况转播”的“实”。

## 1.7.9 复合视频和 s-video

ntsc 和 pal 彩色视频信号是这样构成的--首先有一个基本的黑白视频信号，然后在每个水平同步脉冲之后，加入一个颜色脉冲和一个亮度信号。因为彩色信号是由多 种数据“叠加”起来的，故称之为“复合视频”。s-video 则是一种信号质量更高的视频接口，它取消了信号叠加的方法，可有效避免一些无谓的质量损失。它的 功能是将rgb 三原色和亮度进行分离处理。

## 1.7.10 硬件加速

VDA/vaspi/DX 等等。

## 1.7.11 FFmpeg Device

硬件方式：CDIO / DC1394 （输入设备）

非扩展硬件：DSHOW（输入设备）、SDL（输出设备）、X11（输入）、VFWCAP（输入）、DV1394（输入）等等。

## 第二章 FFmpeg 框架

### 2.1 FFmpeg 概述

#### 2.1.1 简介

Open-source multimedia library, 遵从 GPL/LGPL 协议, ffmpeg 只是一个商标, 它的所有权属于 ffmpeg.org。

由 Fabrice Bellard (法国著名程序员 Born in 1972) 于 2000 年发起创建的开源项目, 同时也是 TinyCC(1996)、发现最快速计算圆周率算法(1997)、TinyGL(1998)、QEMU(2003)、Jslinux(2011)等等的发起人或作者。

FFmpeg 在 Linux 平台上开发, 但它同样也可以在其它操作系统环境中编译运行, 包括 Windows、Mac OS X 等。这个项目是由 Fabrice Bellard 发起的, 现在由 Michael Niedermayer 主持。

许多 FFmpeg 的开发人员都来自 MPlayer 项目, 而且当前 FFmpeg 也是放在 MPlayer 项目组的服务器上。项目的名称来自 MPEG 视频编码标准, 前面的"FF"代表"Fast Forward"。

FFmpeg 是一套可以用来记录、转换数字音频、视频, 并能将其转化为流的开源计算机程序。它包括了目前领先的音/视频编码库 libavcodec。FFmpeg 是在 Linux 下开发出来的, 但它可以在包括 Windows 在内的大多数操作系统中编译。这个项目是由 Fabrice Bellard 发起的, 现在由 Michael Niedermayer 主持。可以轻易地实现多种视频格式之间的相互转换, 例如可以将摄录下的视频 avi 等转成现在视频网站所采用的 flv 格式。

#### 2.1.2 功能

多媒体视频处理工具 FFmpeg 有非常强大的功能[2]包括视频采集功能、视频格式转换、视频抓图、给视频加水印等。

##### 视频采集功能

ffmpeg 视频采集功能非常强大, 不仅可以采集视频采集卡或 USB 摄像头的图像, 还可以进行屏幕录制, 同时还支持以 RTP 方式将视频流传送给支持 RTSP 的流媒体服务器, 支持直播应用。

##### ffmpeg 在 Linux 下的视频采集

在 Linux 平台上, ffmpeg 对 V4L2 的视频设备提高了很好的支持, 如:

```
/ffmpeg -t 10 -f video4linux2 -s 176*144 -r 8 -i /dev/video0 -vcodec h263 -f rtp rtp://192.168.1.105:5060 > /tmp/ffmpeg.sdp
```

以上命令表示: 采集 10 秒钟视频, 对 video4linux2 视频设备进行采集, 采集 QCIF(176\*144)的视频, 每秒 8 帧, 视频设备为/dev/video0, 视频编码为 h263, 输出格式为 RTP, 后面定义了 IP 地址及端口, 将该码流所对应的 SDP 文件重定向到/tmp/ffmpeg.sdp 中, 将此 SDP 文件上传到流媒体服务器就可以实现直播了。

##### ffmpeg 在 windows 下的视频采集

在 windows 下关于 ffmpeg 视频采集的资料非常少, 但是 ffmpeg 还是支持 windows 下视频采集的。ffmpeg 支持 windows 下 video for windows(VFW)设备的视频采集, 不过 VFW 设备已经过时, 正在被 WDM 的视频设备所取代, 但是 ffmpeg 还没有支持 WDM 的计划, 不过好像有将 WDM 转为 VFW 的工具, 因此 ffmpeg 还是可以在 windows 下进行视频采集的。

##### 视频格式转换功能

ffmpeg 视频转换功能。视频格式转换, 比如可以将多种视频格式转换为 flv 格式, 可不是视频信号转换。

ffmpeg 可以轻易地实现多种视频格式之间的相互转换(wma,rm,avi,mod 等), 例如可以将摄录下的视频 avi 等转成现在视频网站所采用的 flv 格式。

视频截图功能

对于选定的视频，截取指定时间的缩略图。视频抓图，获取静态图和动态图，不提倡抓 gif 文件；因为抓出的 gif 文件大而播放不流畅

给视频加水印功能

使用 ffmpeg 视频添加水印(logo)。

### 2.1.3 模块组成

**libavformat:** 用于各种音视频封装格式的生成和解析，包括获取解码所需信息以生成解码上下文结构和读取音视频帧等功能；音视频的格式解析协议，为 libavcodec 分析码流提供独立的音频或视频码流源。

**libavcodec:** 用于各种类型声音/图像编解码；该库是音视频编解码核心，实现了市面上可见的绝大部分解码器的功能，libavcodec 库被其他各大解码器 ffdshow，Mplayer 等所包含或应用。

**libavdevice :** 硬件采集、加速、显示。操作计算机中常用的音视频捕获或输出设备：ALSA,AUDIO\_BEOS,JACK,OSS,1394, VFW。

**libavfilter:filter** (FileIO、FPS、DrawText) 音视频滤波器的开发，如宽高比 裁剪 格式化 非格式化 伸缩。

**libavutil:** 包含一些公共的工具函数的使用库，包括算数运算 字符操作；

**libavresample:** 音视频封转编解码格式预设等。

**libswscale:** (原始视频格式转换) 用于视频场景比例缩放、色彩映射转换；图像颜色空间或格式转换，如 rgb565 rgb888 等与 yuv420 等之间转换。

**libswresample:** 原始音频格式转码

**libpostproc:** (同步、时间计算的简单算法) 用于后期效果处理；音视频应用的后处理，如图像的去块效应。

**ffmpeg:** 该项目提供的一个工具，可用于格式转换、解码或电视卡即时编码等；

**ffserver:** 一个 HTTP 多媒体即时广播串流服务器；

**ffplay:** 是一个简单的播放器，使用 ffmpeg 库解析和解码，通过 SDL 显示；

### 2.1.4 命令集

ffmpeg 命令集举例

1. 获取视频的信息

ffmpeg -i video.avi

2. 将图片序列合成视频

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

上面的命令会把当前目录下的图片（名字如：image1.jpg, image2.jpg, 等...）合并成 video.mpg

3. 将视频分解成图片序列

ffmpeg -i video.mpg image%d.jpg

上面的命令会生成 image1.jpg, image2.jpg, ...

支持的图片格式有：PGM, PPM, PAM, PGMYUV, JPEG, GIF, PNG, TIFF, SGI

4. 为视频重新编码以适合在 iPod/iPhone 上播放

ffmpeg -i source\_video.avi input -acodec aac -ab 128kb -vcodec mpeg4 -b 1200kb -mbd 2 -flags +4mv+trell -aic 2 -cmp 2 -subcmp 2 -s 320x180 -title X final\_video.mp4

说明：

\* 源视频：source\_video.avi

\* 音频编码：aac

## 《FFmpeg 基础库编程开发》

- \* 音频位率: 128kb/s
- \* 视频编码: mpeg4
- \* 视频位率: 1200kb/s
- \* 视频尺寸: 320 X 180
- \* 生成的视频: final\_video.mp4

5.为视频重新编码以适合在 PSP 上播放

```
ffmpeg -i source_video.avi -b 300 -s 320x240 -vcodec xvid -ab 32 -ar 24000 -acodec aac final_video.mp4
```

说明:

- \* 源视频: source\_video.avi
- \* 音频编码: aac
- \* 音频位率: 32kb/s
- \* 视频编码: xvid
- \* 视频位率: 1200kb/s
- \* 视频尺寸: 320 X 180
- \* 生成的视频: final\_video.mp4

6.从视频抽出声音.并存为 Mp3

```
ffmpeg -i source_video.avi -vn -ar 44100 -ac 2 -ab 192 -f mp3 sound.mp3
```

说明:

- \* 源视频: source\_video.avi
- \* 音频位率: 192kb/s
- \* 输出格式: mp3
- \* 生成的声音: sound.mp3

7.将 wav 文件转成 Mp3

```
ffmpeg -i son_origine.avi -vn -ar 44100 -ac 2 -ab 192 -f mp3 son_final.mp3
```

8.将.avi 视频转成.mpg

```
ffmpeg -i video_origine.avi video_finale.mpg
```

9.将.mpg 转成.avi

```
ffmpeg -i video_origine.mpg video_finale.avi
```

10.将.avi 转成 gif 动画 (未压缩)

```
ffmpeg -i video_origine.avi gif_anime.gif
```

11.合成视频和音频

```
ffmpeg -i son.wav -i video_origine.avi video_finale.mpg
```

12.将.avi 转成.flv

```
ffmpeg -i video_origine.avi -ab 56 -ar 44100 -b 200 -r 15 -s 320x240 -f flv video_finale.flv
```

13.将.avi 转成 dv

```
ffmpeg -i video_origine.avi -s pal -r pal -aspect 4:3 -ar 48000 -ac 2 video_finale.dv
```

或者:

```
ffmpeg -i video_origine.avi -target pal-dv video_finale.dv
```

14.将.avi 压缩成 divx

```
ffmpeg -i video_origine.avi -s 320x240 -vcodec msmpeg4v2 video_finale.avi
```

15.将 Ogg Theora 压缩成 Mpeg dvd

```
ffmpeg -i film_sortie_cinelerra.ogm -s 720x576 -vcodec mpeg2video -acodec mp3 film_terminate.mpg
```

16.将.avi 压缩成 SVCD mpeg2

NTSC 格式:

```
ffmpeg -i video_origine.avi -target ntsc-svcd video_finale.mpg
```

PAL 格式:

```
ffmpeg -i video_origine.avi -target pal-svcd video_finale.mpg
```

17. 将.avi 压缩成 VCD mpeg2

NTSC 格式:

```
ffmpeg -i video_origine.avi -target ntsc-svcd video_finale.mpg
```

PAL 格式:

```
ffmpeg -i video_origine.avi -target pal-vcd video_finale.mpg
```

18. 多通道编码

```
ffmpeg -i fichierentree -pass 2 -passlogfile ffmpeg2pass fichiersortie-2
```

19. 从 flv 提取 mp3

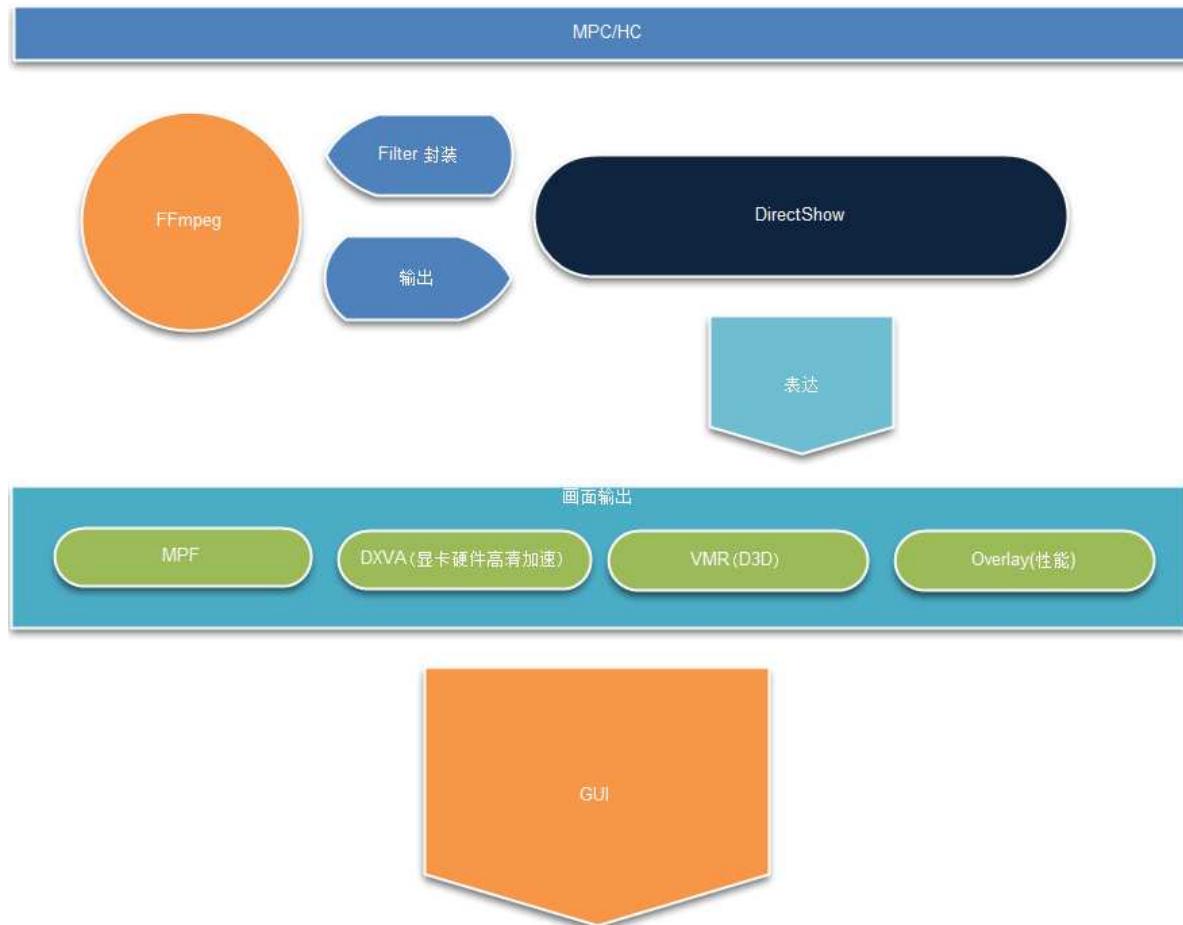
```
ffmpeg -i source.flv -ab 128k dest.mp3
```

## 2.2 媒体播放器三大底层框架

媒体播放工具，这里主要指视频播放，因为要面临庞大的兼容性和纷繁复杂的算法，从架构上看，能脱颖而出的体系屈指可数。大体来说业界主要有 3 大架构：MPC、MPlayer 和 VLC。这 3 大架构及其衍生品占领了 90% 的市场，凡是用户能看到的免费媒体播放软件，无一不是源自这 3 大架构。

### MPC/HC 架构

MPC（Media Player Classic）和它的后续者 MPC-HC 应该并列而说。MPC 基于 DirectShow 架构，是 Windows 系统下元祖级别的播放器。包括 KMP 之流最早也就是抄来 MPC 的代码再换个界面。MPCHC 则在 MPC 的原作者 Gabest 渐渐退出开发后的继承者，MPCHC 有很多创新特性，包括开始融入 ffmpeg 和支持更多 DirectX 特性和 DXVA 等等。



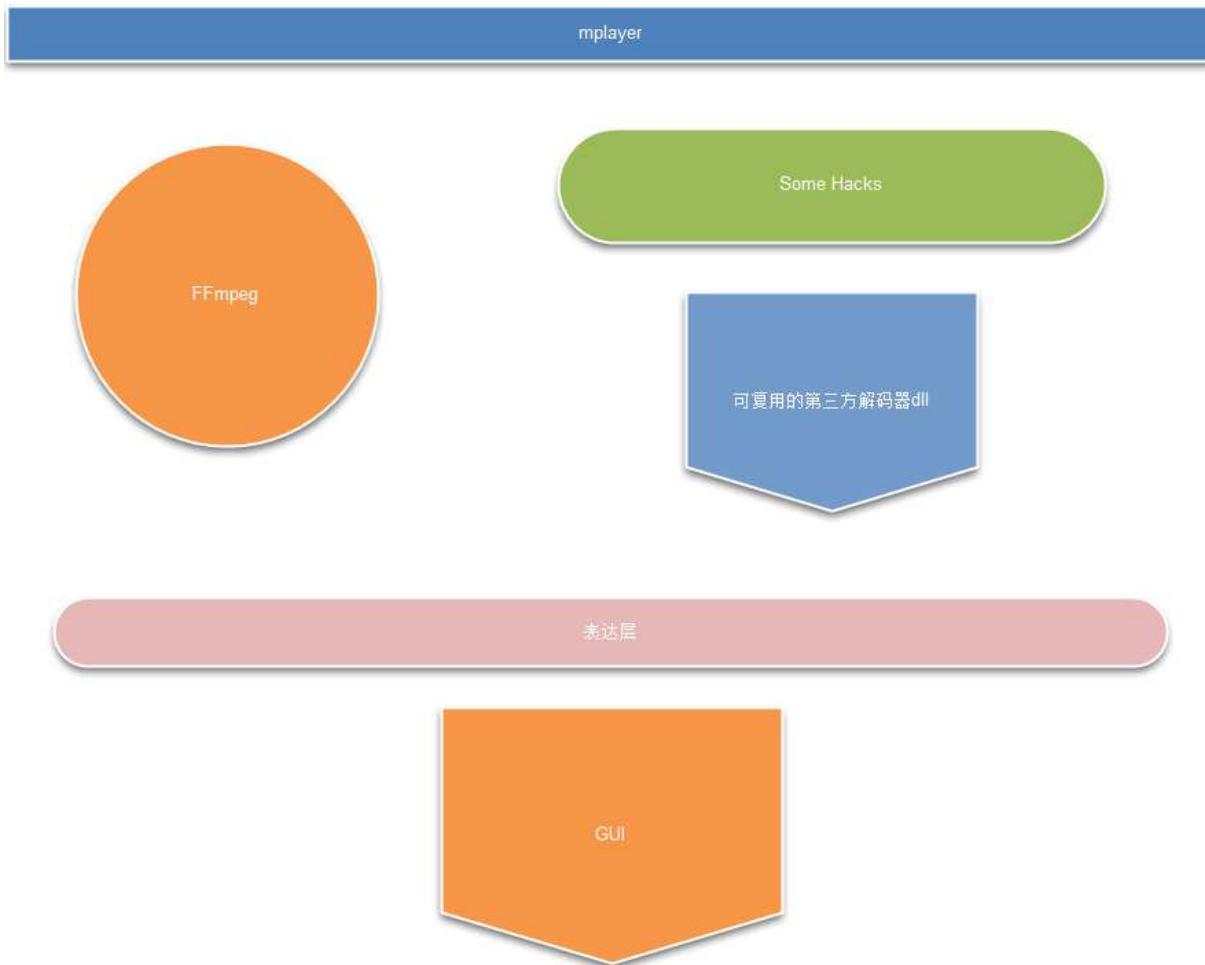
**优点：**更直接的支持 DXVA，对一些稀奇古怪的 Windows 平台上的格式可以通过调用第三方的 Filter 组件等，拥有更好的兼容性

**缺点：**有人说 DirectShow 是 Windows 中最难掌握的 SDK，开发复杂；DirectShow 允许第三方封装的特点也让兼容性和稳定性问题复杂化；第三方 Filter 出现异常时非常难以分析处理，更难以复用；

射手播放器的架构主要来自 MPC-HC，但更多的融合了 FFmpeg 的优势，对 DirectShow Filter 进行了多处改写，大大加强了对 ffmpeg 的利用，提高了解码稳定性，同时扩展了解码能力和兼容性。

### mplayer 架构

如果说 MPC 是 Windows 上的元祖，那么 mplayer 就是 linux 上媒体播放的元祖了。mplayer 使用 ffmpeg 作为解码核心，也是与 ffmpeg 结合最紧密的项目，ffmpeg 的代码就是由 mplayer 来 host，开发者群也有非常大的交集。借助 linux 开发/使用者的强大实力，mplayer 建立了要比 DirectShow 稳定的多的工作流程。超越 ffmpeg 本身的功能外，后来又通过反向工程使之可以调用 Windows 上的 DirectShow Filter DLL，让 mplayer 架构越来越吸引人，成为兼具稳定性和性能的优秀作品。

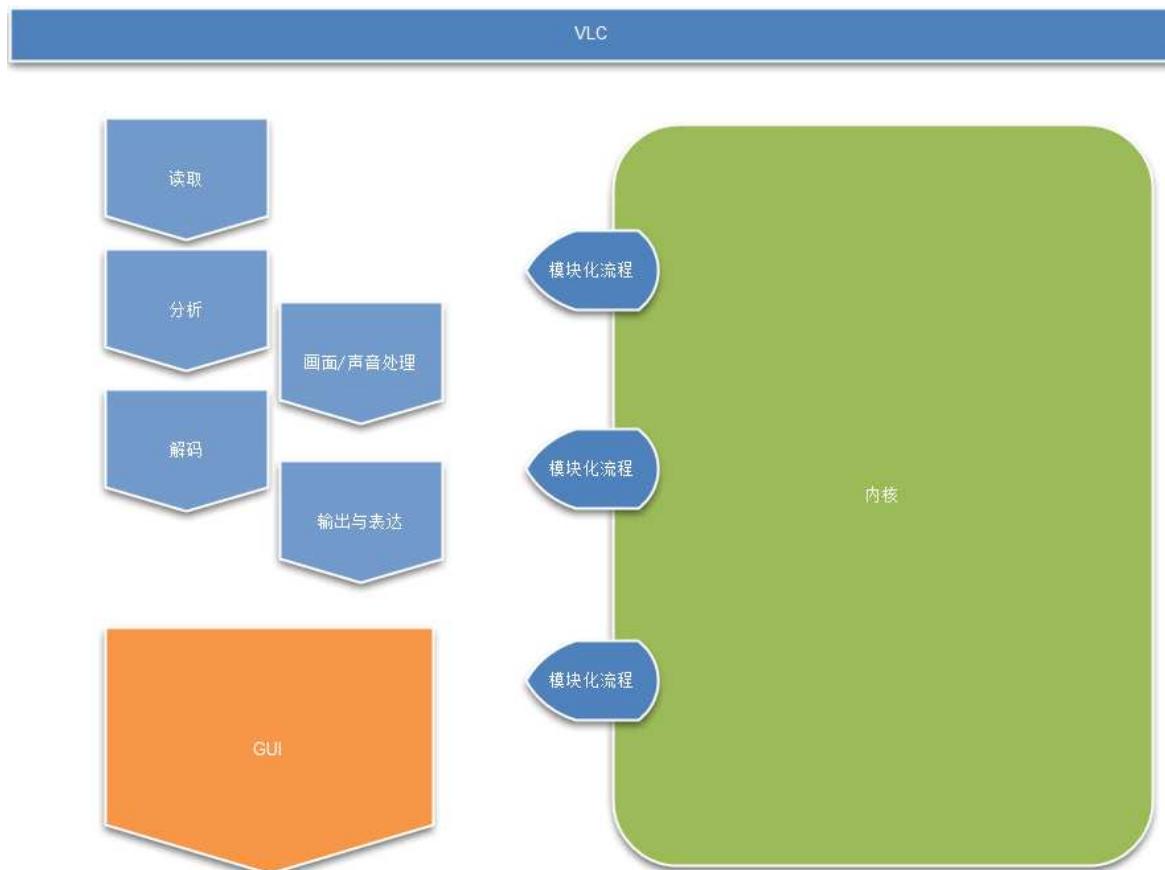


优点：稳定，兼容性也可以说相当不错

缺点：代码结构不清晰；纯 C 语言开发，难于阅读；显卡硬件加速还需要越过更多障碍

### VLC 架构

VLC 是个后起之秀，开发速度的进展可以说是一只奇葩。虽然同样基于 ffmpeg，但可能是相对于“左三年右三年缝缝补补又三年”的 mplayer 架构来说，VLC 的架构在设计之初就很好的考虑到模块化开发，所以使它更吸引年轻的开发人员。成为近年发展非常快的架构。

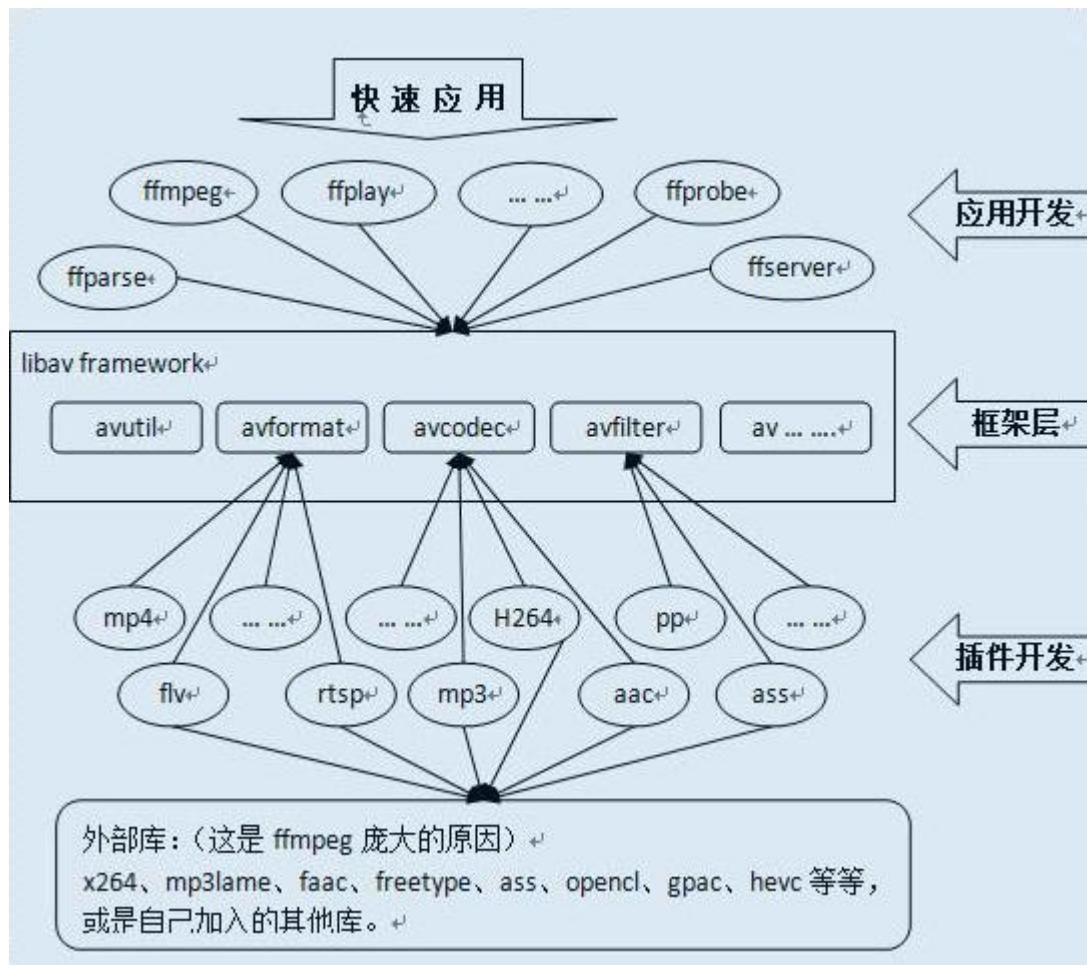


优点：稳定，兼容性也可以说相当不错

缺点：纯 C 语言开发，难于阅读；硬件加速略有障碍

很多人都会发现，3 大架构中都可以看到 ffmpeg 的名字。说起 ffmpeg，那真是“One Ring to rule them all, One Ring to find them, One Ring to bring them all”(翻译：至尊戒驭众戒，至尊戒寻众戒，至尊戒引众戒。出自《魔戒》)。在 #ffmpeg 有人和我说过，想不用 ffmpeg 去写媒体播放器，就像是造汽车而不用车轮。但是 ffmpeg 本身仅作为命令行工具或类库（常见的如 libavcodec）出现。终端用户很少能直接接触到 ffmpeg，所以知名度也较小。

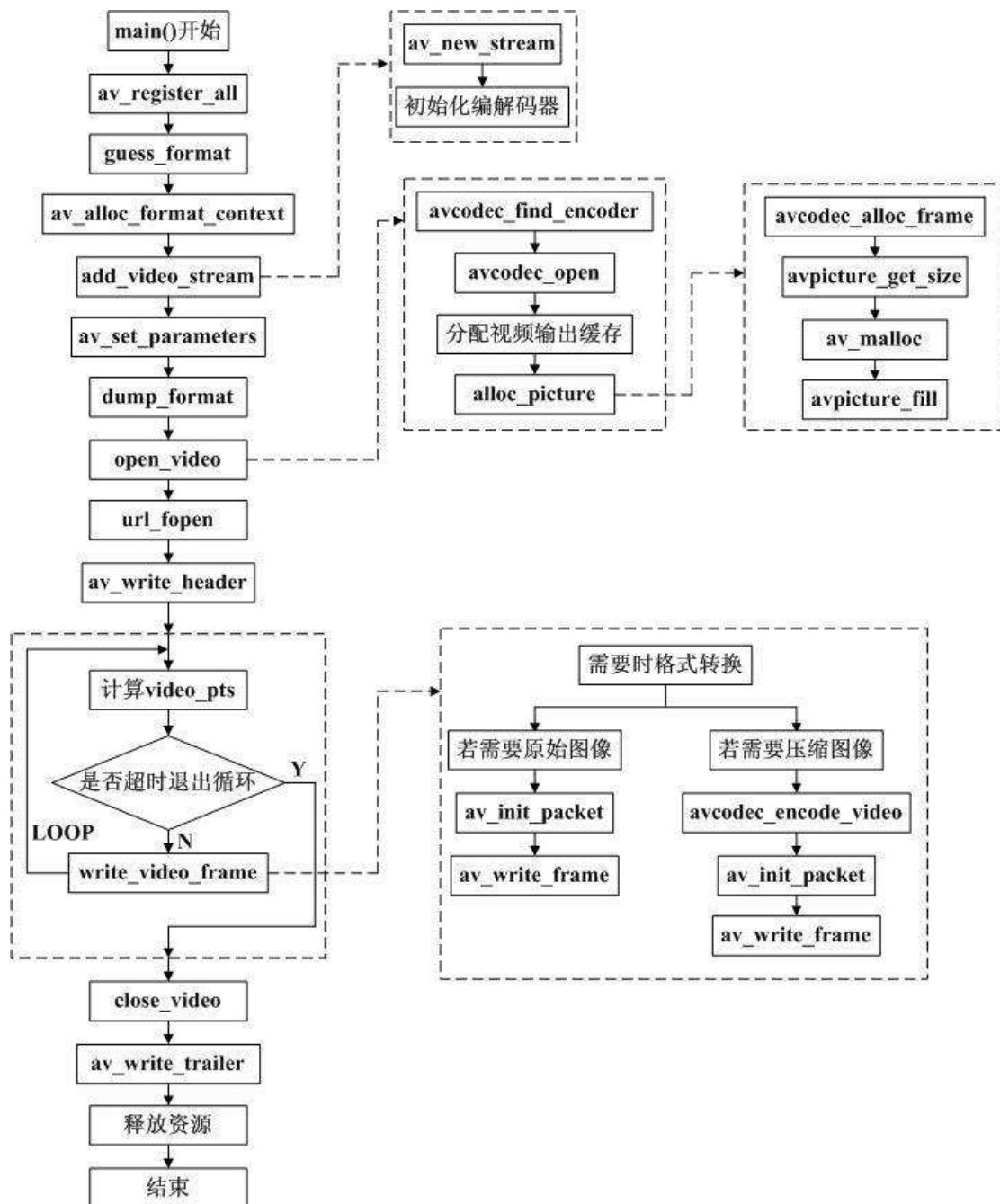
# 第三章 编译及简单应用



## 3.1 FFmpeg 库编译和入门介绍

编译过程详见专门文档。

## 3.2 流媒体数据流程讲解



FFMpeg 的 output\_example.c 例子分析

该例子讲了如何输出一个 libavformat 库所支持格式的媒体文件。

(1) `av_register_all()`, 初始化 libavcodec 库, 并注册所有的编解码器和格式。

(2) `guess_format()`, 根据文件名来获取输出文件格式, 默认为 mpeg。

(3) `av_alloc_format_context()`分配输出媒体内容。

`ov->ofmt = fm;`

`sprintf(oc->filename, sizeof(oc->filename), "%s", filename);`

(4) `add_video_stream()`使用默认格式的编解码器来增加一个视频流, 并初始化编解码器。

(4.1) `av_new_stream()`增加一个新的流到一个媒体文件。

(4.2) 初始化编解码器:

```
c = s t->codec;
c->codec_id = codec_id;
c->codec_type = CODEC_TYPE_VIDEO;
c->bit_rate = 400000;
c->width = 352;
c->height = 288;
c->tim e_base.den = STREAM_FRAME_RATE; //每秒 25 副图像
c->tim e_base.num = 1;
c->gop_size = 12;
c->pix_fmt = STREAM_PIX_FMT; //默认格式为 PIX_FMT_YUV420P
.....
```

(5) av\_set\_parameters()设置输出参数, 即使没有参数, 该函数也必须被调用。

(6) dump\_format()输出格式信息, 用于调试。

(7) open\_video()打开视频编解码器并分配必要的编码缓存。

(7.1) avcodec\_find\_encoder()寻找 c->codec\_id 指定的视频编码器。

(7.2) avcodec\_open()打开编码器。

(7.3) 分配视频输出缓存:

```
video_outbuf_size = 200000;
video_outbuf = av_malloc(video_outbuf_size);
```

(7.4) picture = alloc\_picture()分配原始图像。

(7.4.1) avcodec\_alloc\_frame()分配一个 AVFrame 并设置默认值。

(7.4.2) size = avpicture\_get\_size()计算对于给定的图片格式以及宽和高, 所需占用多少 内存。

(7.4.3) picture\_buf = av\_malloc(size)分配所需内存。

(7.4.4) avpicture\_fill()填充 AVPicture 的域。

(7.5) 可选。如果输出格式不是 YUV420P, 那么临时的 YUV420P 格式的图像也是需要的, 由此再转换为我们所需的格式, 因此需要为临时的 YUV420P 图像分配缓存:

```
tmp_picture = alloc_picture()
```

说明: tmp\_picture, picture, video\_outbuf。如果输出格式为 YUV420P, 则直接通过 avcodec\_encode\_video()函数将 picture 缓存中的原始图像编码保存到 video\_outbuf 缓存中; 如果输出格式不是 YUV420P, 则需要先通过 sws\_scale()函数, 将 YUV420P 格式转换为目标格式, 此时 tmp\_picture 缓存存放的是 YUV420P 格式的图像, 而 picture 缓存为转换为目标格式后保存的图像, 进而再将 picture 缓存中的图像编码保存到 video\_outbuf 缓存中。

(8) url\_fopen()打开输出文件, 如果需要的话。

(9) av\_write\_header()写流动头部。

(10) LOOP 循环{

计算当前视频时间 video\_pts 是否超时退出循环? write\_video\_frame()视频编码  
}

(10.1) write\_video\_frame()

如果图片不是 YUV420P, 则需要用 sws\_scale()函数先进行格式转换。若需要原始图像: av\_init\_packet()初始化一个包的选项域。

av\_write\_frame()向输出媒体文件写一个包, 该包会包含一个视频帧。若需要编码图像: avcodec\_encode\_video()编码一视频帧。

av\_init\_packet()

av\_write\_frame()

(11) close\_video()关闭每个编解码器。

(12) av\_write\_trailer()写流的尾部。

(13) 释放资源

av\_free()释放 AVForm atContext 下的 AVS tream ->AVCodecContext 和 AVStream :

```
for( i = 0; i < oc->nb_s treams ; i++ ){
    av_free( &oc->s treams [i]->codec );
    av_free( &oc->s treams [i] );
}
```

url\_fclose()关闭输出文件。

av\_free()释放 AVForm atContext。

### 3.3 简单应用

PS:此处举 tutorial 的例子是为了更好的引出一个循序渐进的例程。条件适当的话添加 output\_example.c 实例并进行说明。

FFmpeg tutorial 对初级的掌握以及使用 ffmpeg 有重要指导作用，但是里面的一些函数没有实时更新了，tutorial01~08 是一个播放器开发的由浅入深的过程，下面介绍 tutorial01 (tutorial 02~08 详见附录) 使用源码：

```
/****************************************************************************
 * tutorial1 制作屏幕录像
 * 执行后，将视频文件按照一定的格式保存为.ppm文件 */
#include "libavcodec/avcodec.h"
#include "libavformat/avformat.h"
#include "libswscale/swscale.h"
#include <windows.h>
#include <stdio.h>
void SaveFrame(AVFrame *pFrame, int width, int height, int iFrame) {
    FILE *pFile;
    char szFilename[32];
    int y;
    // Open file
    sprintf(szFilename, "frame%d.ppm", iFrame);
    pFile=fopen(szFilename, "wb");
    if(pFile==NULL)
        return;
    // Write header
    fprintf(pFile, "P6\n%d %d\n255\n", width, height);
    // Write pixel data
    for(y=0; y<height; y++)
        fwrite(pFrame->data[0]+y*pFrame->linesize[0], 1, width*3, pFile);
    // Close file
    fclose(pFile);
}
```

## 《FFmpeg 基础库编程开发》

```
int main(int argc, char *argv[])
{
    AVFormatContext *pFormatCtx;
    int i, videoStream;
    AVCodecContext *pCodecCtx;
    AVCodec *pCodec;
    AVFrame *pFrame;
    AVFrame *pFrameRGB;
    AVPacket packet;
    int frameFinished;
    int numBytes;
    uint8_t *buffer;

    if (argc < 2)
    {
        printf("please provide a movie file\n");
        return -1;
    }
    //register all formats and codes
    av_register_all();
    //support network stream input
    avformat_network_init();
    pFormatCtx = avformat_alloc_context();

    //Open the media file and read the header
    if(avformat_open_input(&pFormatCtx, argv[1], NULL, NULL) != 0)
    {
        printf("couldn't open file \n");
        return -1;
    }

    //retrieve stream information
    if (av_find_stream_info(pFormatCtx) < 0 )
        return -1;
    //dump information about file into standard error
    av_dump_format(pFormatCtx, -1, argv[1], 0 );
    // Find the first video stream
    videoStream = -1;
    for (int i = 0 ;i < pFormatCtx->nb_streams; i++ )
        if (pFormatCtx->streams[i]->codec->codec_type == AVMEDIA_TYPE_VIDEO)
    {
        videoStream = i ;
        break;
    }
```

```

if (videoStream == -1)
    return -1;
//get a pointer to the codec context for the video stream
pCodecCtx = pFormatCtx->streams[videoStream]->codec;
pCodec = avcodec_find_decoder(pCodecCtx->codec_id);
if (pCodec == NULL)
{
    fprintf(stderr, "unsupported codec \n");
    return -1;
}
//open codec
if(avcodec_open2(pCodecCtx, pCodec, NULL) < 0 )
    return -1;
//allocate video frame
pFrame = avcodec_alloc_frame();
if(NULL == pFrame )
    return -1;
//allocate an avframe structure
pFrameRGB = avcodec_alloc_frame();
if (pFrameRGB == NULL)
    return -1;
//determine required buffer size and allocate buffer
numBytes = avpicture_get_size(PIX_FMT_RGB24, pCodecCtx->width, pCodecCtx->height);
//buffer = (uint8_t *)av_malloc_attrib(numBytes * sizeof(uint8_t));
buffer = (uint8_t *)av_malloc(numBytes * sizeof(uint8_t));
avpicture_fill((AVPicture *)pFrameRGB, buffer, PIX_FMT_RGB24, pCodecCtx->width, pCodecCtx->height);

i = 0;
while (av_read_frame(pFormatCtx, &packet)>=0)
{
    if (packet.stream_index == videoStream)
    {
        avcodec_decode_video2(pCodecCtx, pFrame, &frameFinished, &packet);
        if( frameFinished )
        {
            struct SwsContext *img_convert_ctx = NULL;
            img_convert_ctx =sws_getCachedContext(img_convert_ctx, pCodecCtx->width, pCodecCtx->height, \
                pCodecCtx->pix_fmt, pCodecCtx->width, pCodecCtx->height, PIX_FMT_RGB24, SWS_BICUBIC, NULL,
                NULL, NULL);

            if( !img_convert_ctx )
            {
                fprintf(stderr, "Cannot initialize sws conversion context\n");
                exit(1);

```

```

    《FFmpeg 基础库编程开发》

}

/*
int sws_scale(struct SwsContext *c, const uint8_t *const srcSlice[],
const int srcStride[], int srcSliceY, int srcSliceH,
uint8_t *const dst[], const int dstStride[]);
*/
sws_scale(img_convert_ctx, pFrame->data, \
          pFrame->linesize, 0, pCodecCtx->height, pFrameRGB->data, pFrameRGB->linesize);
if( i++ < 50 )
    SaveFrame(pFrameRGB, pCodecCtx->width, pCodecCtx->height, i);
}
}

av_free_packet(&packet);
}

//free the RGB image
av_free(buffer);
av_free(pFrameRGB);
av_free(pFrame);
avcodec_close(pCodecCtx);
av_close_input_file(pFormatCtx);
}

```

## 3.4 SDL ( Simple Direct Layer )

它是一个出色的多媒体库，适用于 PC 平台，并且已经应用在许多工 程中，它是如此的优秀，甚至已移植到某些手机平台上。它的官方网站是 <http://www.libsdl.org/>，在这个网站上可以下载 SDL 库的源代码自己编译库，也可以直接下载预编译库。

### 3.4.1 SDL 显示视频

SDL 显示视频图像函数调用序列如下，忽略掉错误处理：

1): 初始化 SDL 库，

`SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER)`

2): 创建显示表面，

`SDL_Surface *screen = SDL_SetVideoMode(width, height, 0, 0)`

3): 创建 Overlay 表面，

`SDL_Overlay *bmp = SDL_CreateYUVOverlay(width, height, SDL_YV12_OVERLAY, screen)`

4): 取得独占权和 Overlay 表面首地址， `SDL_LockYUVOverlay(bmp);`

5): 填充视频数据，纹理数据

6): 释放独占权， `SDL_UnlockYUVOverlay(bmp);`

7): 刷新视频， `SDL_DisplayYUVOverlay(bmp, &rect);`

### 3.4.2 SDL 显示音频

SDL 播放音频采用回调函数的方式来保证音频的连续性，在设置音频输出参数，向系统注册回调函数后，每次写入的音频数据播放完，系统自动调用注册的回调函数，通常在此回调函数中继续往系统写入音频数据。

SDL 播放音频函数调用序列，忽略掉错误处理：

1): 初始化 `SDL_AudioSpec` 结构，此结构包括音频参数和回调函数，比如 `SDL_AudioSpec wanted_spec;`

`wanted_spec userdata = is; wanted_spec.channels = 2; wanted_spec.callback = sdl_audio_callback;`

2).....

3): 打开音频设备 `SDL_OpenAudio(&wanted_spec, &spec);`

3) 激活 音频设备开始工作 `SDL_PauseAudio(0);`

4) 在音频回调函数中写入音频数据，示意代码如下

```
void sdl_audio_callback(void *opaque, Uint8 *stream, int len)
{
    memcpy(stream, (uint8_t*)audio_buf, len);
}
```

5): 播放完后关闭音频 `SDL_CloseAudio();`

## 3.5 ffmpeg 程序的使用

### 3.5.1 ffmpeg.exe

ffmpeg 是用于转码的应用程序。

一个简单的转码命令可以这样写：

将 `input.avi` 转码成 `output.ts`，并设置视频的码率为 `640kbps`

`#ffmpeg -i input.avi -b:v 640k output.ts`

具体的使用方法可以参考： `ffmpeg` 参数中文详细解释

详细的使用说明（英文）：<http://ffmpeg.org/ffmpeg.html>

### 3.5.2 ffplay.exe

ffplay 是用于播放的应用程序。

一个简单的播放命令可以这样写：

播放 `test.avi`

`#ffplay test.avi`

具体的使用方法可以参考： `ffplay` 的快捷键以及选项

详细的使用说明（英文）：<http://ffmpeg.org/ffplay.html>

### 3.5.3 ffprobe.exe

ffprobe 是用于查看文件格式的应用程序。

这个就不多介绍了。

## 《FFmpeg 基础库编程开发》

详细的使用说明（英文）：<http://ffmpeg.org/ffprobe.html>

## 第四章 多媒体处理基本流程

本文将从介绍一些基本的多媒体只是，主要是为研读 ffmpeg 源代码做准备，比如一些编解码部分，只有真正了解了多媒体处理的基本流程，研读 ffmpeg 源代码才能事半功倍。

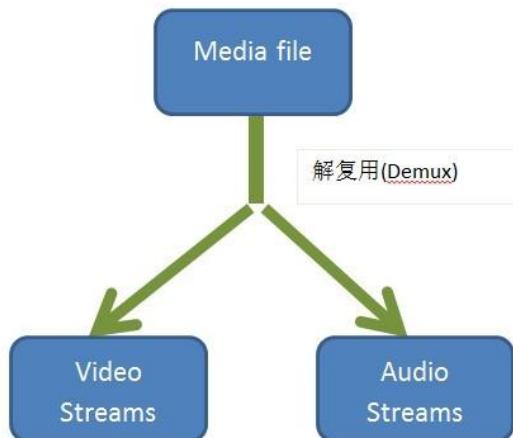
下面分析一下多媒体中最基本最核心的视频解码过程，平常我们从网上下载一部电影或者一首歌曲，那么相应的多媒体播放器为我们做好了一切工作，我们只用欣赏就 ok 了。目前几乎所有的主流多媒体播放器都是基于开源多媒体框架 ffmpeg 来做的，可见 ffmpeg 的强大。下面是对一个媒体文件进行解码的主要流程：



解码流程图

### 1. 解复用(Demux)

当我们打开一个多媒体文件之后，第一步就是解复用，称之为 Demux。为什么需要这一步，这一步究竟是做什么的？我们知道在一个多媒体文件中，既包括音频也包括视频，而且音频和视频都是分开进行压缩的，因为音频和视频的压缩算法不一样，既然压缩算法不一样，那么肯定解码也不一样，所以需要对音频和视频分别进行解码。虽然音频和视频是分开进行压缩的，但是为了传输过程的方便，将压缩过的音频和视频捆绑在一起进行传输。所以我们解码的第一步就是将这些绑在一起的音频流和视频流分开来，也就是传说中的解复用，所以一句话，解复用这一步就是将文件中捆绑在一起的音频流和视频流分开来以方便后面分别对它们进行解码，下面是 Demux 之后的效果。

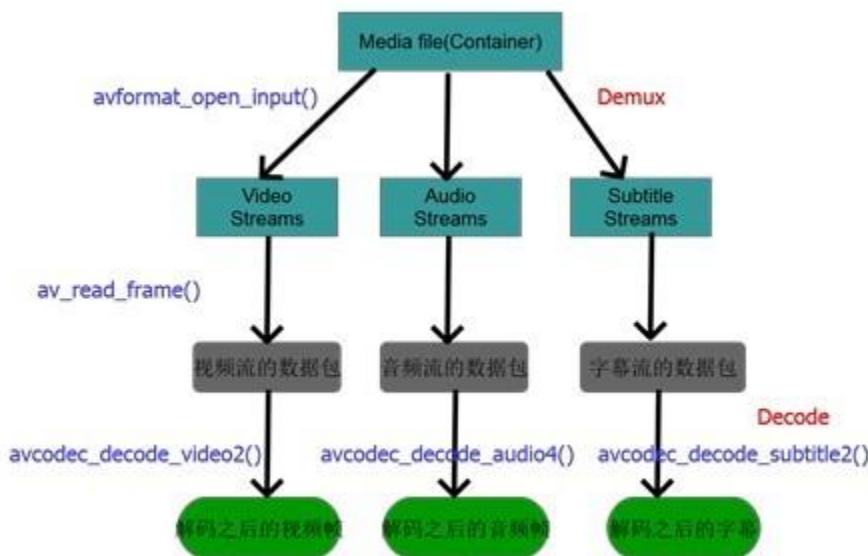


## 2. 解码(Decode)

这一步不用多说，一个多媒体文件肯定是经过某种或几种格式的压缩的，也就是通常所说的视频和音频编码，编码是为了减少数据量，否则的话对我们的存储设备是一个挑战，如果是流媒体的话对网络带宽也是一个几乎不可能完成的任务。所以我们必须对媒体信息进行尽可能的压缩。

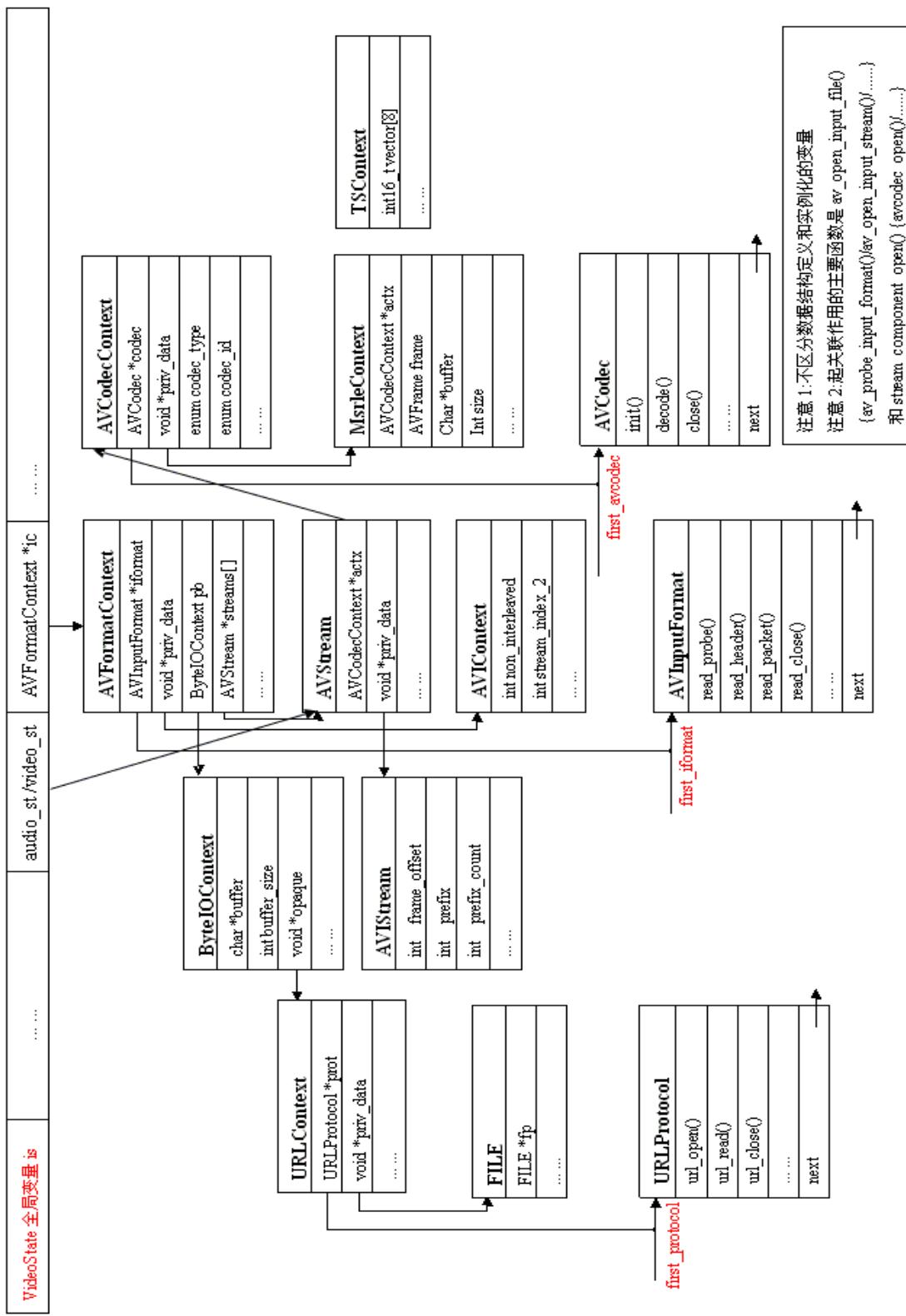
## 3. FFmpeg 中解码流程对应的 API 函数

了解了上面的一个媒体文件从打开到解码的流程，就可以很轻松的阅读 ffmpeg 代码，ffmpeg 的框架也基本是按照这个流程来的，但不是每个流程对应一个 API，下面这副图是我分析 ffmpeg 并根据自己的理解得到的 ffmpeg 解码流程对应的 API，我想这幅图应该对理解 ffmpeg 和编解码有一些帮助。



FFmpeg 中 Demux 这一步是通过 `avformat_open_input()`这个 api 来做的，这个 api 读出文件的头部信息，并做 demux，在此之后我们就可以读取媒体文件中的音频和视频流，然后通过 `av_read_frame()`从音频和视频流中读取出基本数据流 packet，然后将 packet 送到 `avcodec_decode_video2()`和相对应的 api 进行解码。

# 第四章 数据结构



注: ByteIOContext → AVIOContext

ffmpeg 定义的数据结构很有特色:

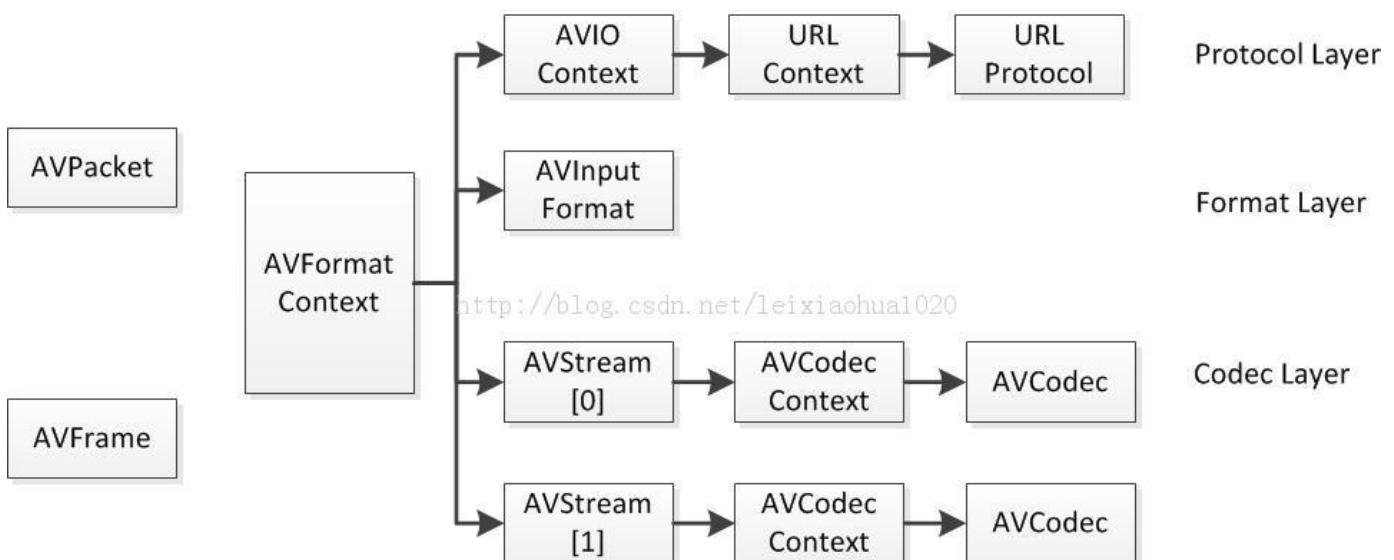
有一些是动态与静态的关系, 比如, URLProtocol 和 URLContext, AVInputFormat 和 AVFormatContext, AVCodec 和 AVCodecContext。从前面播放器的一般原理我们可知, 播放器内部要实现的几大功能是, 读文件, 识别格式, 音视频解码, 音视频渲染(这也是接口调用流程)。其中音视频渲染由 SDL 实现, 我们不讨论。ffplay 把其他的每个大功能抽象成一个相当于 C++ 中 COM 接口的数据结构, 着重于功能函数, 同时这些功能函数指针在编译的时候就能静态确定。每一个大功能都要支持多种类型的广义数据, ffplay 把这多种类型的广义数据的共同的部分抽象成对应的 Context 结构, 这些对应的 context 结构着重于动态性, 其核心成员只能在程序运行时动态确定其值。并且 COM 接口类的数据结构在程序运行时有很多很多实例, 而相应的 Context 类只有一个实例, 这里同时体现了数据结构的划分原则, 如果有一对多的关系就要分开定义。

有一些是指针表述的排他性包含关系(因为程序运行时同一类型的多种数据只支持一种, 所以就有排他性)。比如, AVCodecContext 用 priv\_data 包含 MsrleContext 或 TSContext, AVFormatContext 用 priv\_data 包含 AVIContext 或其他类 Context, AVStream 用 priv\_data 包含 AVIStream 或其他类 Stream。由前面数据结构的动态与静态关系可知, ffplay 把多种类型的广义数据的共同部分抽象成 context 结构, 那么广义数据的各个特征不同部分就抽象成各种具体类型的 context, 然后用 priv\_data 字段表述的指针排他性的关联起来。由于瘦身后的 ffplay 只支持有限类型, 所以 AVFormatContext 只能关联包含 AVIContext, AVStream 只能关联包含 AVIStream。

有一些是扩展包含关系, 比如, ByteIOContext 包含 URLContext, 就是在应用层把没有缓存的 URLContext 扩展有缓冲区的广义文件 ByteIOContext, 改善程序 IO 性能。

有一些是直接包含关系, 比如, AVFrame 包含 AVPicture, 这两个结构共有的字段, 其定义类型、大小、顺序都一模一样, 除了更准确的描述各自的意义便于阅读理解维护代码外, 还可以方便的把 AVFrame 大结构强制转换成 AVPicture 小结构。

我们先来重点分析 AVCodec/AVCodecContext/MsrleContext 这几个数据结构, 这几个数据结构定义了编解码器的核心架构, 相当于 Directshow 中的各种音视频解码器 decoder。



解协议 (http, rtsp, rtmp, mms) → 解封装 (flv, avi, rmvb, mp4) → 解码 h264, mpeg2, aac, mp3) → 存数据

## 4.1 AVCodec 结构体

```

typedef struct AVCodec
{
    // 标示 Codec 的名字, 比如, "h264" "h263" 等。
}

```

```

const char *name;
// 标示 Codec 的类型，有 video , audio 等类型。
enum CodecType type;
// 标示 Codec 的 ID，有 CODEC_ID_H264 等。
enum CodecID id;
// 标示具体的 Codec 对应的 Context 的 size,如: H264Context。
int priv_data_size;
// 以下标示 Codec 对外提供的操作,每一种解码器都会实现这些操作。
int(*init)(AVCodecContext*);
int(*encode)(AVCodecContext *, uint8_t *buf, int buf_size, void *data);
int(*close)(AVCodecContext*);
int(*decode)(AVCodecContext *, void *outdata, int *outdata_size, uint8_t *buf, int buf_size);
struct AVCodec *next;
}AVCodec;
H264 的主要结构的初始化如下:
AVCodec ff_h264_decoder = {
    "h264",
    AVMEDIA_TYPE_VIDEO,
    CODEC_ID_H264,
    sizeof(H264Context),
    ff_h264_decode_init,
    NULL,
    ff_h264_decode_end,
    decode_frame
}
说明:

```

AVCodec 是类似 COM 接口的数据结构，表示音视频编解码器，着重于功能函数，一种媒体类型对应一个 AVCodec 结构，在程序运行时有多个实例。next 变量用于把所有支持的编解码器连接成链表，便于遍历查找；id 确定了唯一编解码器；priv\_data\_size 表示具体 Codec 对应的 Context 结构大小，比如 MsrleContext 或 TSContext，这些具体的结构定义散落于各个.c 文件中，为避免太多的 if else 类语句判断类型再计算大小，这里就直接指明大小，因为这是一个编译时静态确定的字段，所以放在 AVCodec 而不是 AVCodecContext 中。

## 4.2 AVCodecContext 结构体

```

typedef struct AVCodecContext
{
    int bit_rate;
    int frame_number;
    // 扩展数据，如 mov 格式中 audio trak 中 aac 格式中 esds 的附加解码信息。
    unsigned char *extradata;
    // 扩展数据的 size
    int extradata_size;
    // 视频的原始的宽度与高度
    int width, height; // 此逻辑段仅针对视频
}

```

```

//视频一帧图像的格式，如 YUV420
enum PixelFormat pix_fmt;
//音频的采样率
int sample_rate;
//音频的声道的数目
int channels;
int bits_per_sample;
int block_align;
// 指向相应的解码器，如：ff_h264_decoder
struct AVCodec *codec;
//指向具体相应的解码器的 context，如 H264Context
void *priv_data;
//公共操作函数
int(*get_buffer)(struct AVCodecContext *c, AVFrame *pic);
void(*release_buffer)(struct AVCodecContext *c, AVFrame *pic);
int(*reget_buffer)(struct AVCodecContext *c, AVFrame *pic);
}AVCodecContext;
说明：

```

AVCodecContext 结构表示程序运行的当前 Codec 使用的上下文，着重于所有 Codec 共有的属性(并且是在程序运行时才能确定其值)和关联其他结构的字段。extradata 和 extradata\_size 两个字段表述了相应 Codec 使用的私有数据； codec 字段关联相应的编解码器； priv\_data 字段关联各个具体编解码器独有的属性 context，和 AVCodec 结构中的 priv\_data\_size 配对使用。

## 4.13 AVFrame 结构体

```

typedef struct AVFrame {
#define AV_NUM_DATA_POINTERS 8
    uint8_t *data[AV_NUM_DATA_POINTERS];
    int linesize[AV_NUM_DATA_POINTERS];
    uint8_t **extended_data;
    /**宽高 */
    int width, height;
    int nb_samples;
    int format;
    /**是否是关键帧*/
    int key_frame;
    /**帧类型 (I,B,P) */
    enum AVPictureType pict_type;
    uint8_t *base[AV_NUM_DATA_POINTERS];
    AVRational sample_aspect_ratio;
    int64_t pts;
    int64_t pkt_pts;
    int64_t pkt_dts;
    int coded_picture_number;
}

```

```

int display_picture_number;
int quality;
int reference;
/**QP 表*/
int8_t *qscale_table;

int qstride;
int qscale_type;
/**跳过宏块表 */
uint8_t *mbskip_table;
/**运动矢量表*/
int16_t (*motion_val[2])[2];
/**宏块类型表 */
uint32_t *mb_type;
/**DCT 系数 */
short *dct_coeff;
/**参考帧列表 */
int8_t *ref_index[2];
void *opaque;
uint64_t error[AV_NUM_DATA_POINTERS];
int type;
int repeat_pict;
int interlaced_frame;
int top_field_first;
int palette_has_changed;
int buffer_hints;
AVPanScan *pan_scan;
int64_t reordered_opaque;
void *hwaccel_picture_private;
struct AVCodecContext *owner;
void *thread_opaque;
/***
 * log2 of the size of the block which a single vector in motion_val represents:
 * (4->16x16, 3->8x8, 2-> 4x4, 1-> 2x2)
 * - encoding: unused
 * - decoding: Set by libavcodec.
 */
uint8_t motion_subsample_log2;
/** (音频) 采样率 */
int sample_rate;
uint64_t channel_layout;
int64_t best_effort_timestamp;
int64_t pkt_pos;
int64_t pkt_duration;
AVDictionary *metadata;

```

```

int decode_error_flags;
#define FF_DECODE_ERROR_INVALID_BITSTREAM    1
#define FF_DECODE_ERROR_MISSING_REFERENCE    2
int64_t channels;
} AVFrame;

```

AVFrame 结构体一般用于存储原始数据（即非压缩数据，例如对视频来说是 YUV, RGB，对音频来说是 PCM），此外还包含了一些相关的信息。比如说，解码的时候存储了宏块类型表，QP 表，运动矢量表等数据。编码的时候也存储了相关的数据。因此在使用 FFMPEG 进行码流分析的时候，AVFrame 是一个很重要的结构体。

下面看几个主要变量的作用（在这里考虑解码的情况）：

```

uint8_t *data[AV_NUM_DATA_POINTERS]: 解码后原始数据（对视频来说是 YUV, RGB，对音频来说是 PCM）
int linesize[AV_NUM_DATA_POINTERS]: data 的大小
int width, height: 视频帧宽和高（1920x1080, 1280x720...）
int nb_samples: 音频的一个 AVFrame 中可能包含多个音频帧，在此标记包含了几个
int format: 解码后原始数据类型（YUV420, YUV422, RGB24...）
int key_frame: 是否是关键帧
enum AVPictureType pict_type: 帧类型（I,B,P...）
AVRational sample_aspect_ratio: 宽高比（16:9, 4:3...）
int64_t pts: 显示时间戳
int coded_picture_number: 编码帧序号
int display_picture_number: 显示帧序号
int8_t *qscale_table: QP 表
uint8_t *mbskip_table: 跳过宏块表
int16_t (*motion_val[2])[2]: 运动矢量表
uint32_t *mb_type: 宏块类型表
short *dct_coeff: DCT 系数，这个没有提取过
int8_t *ref_index[2]: 运动估计参考帧列表（貌似 H.264 这种比较新的标准才会涉及到多参考帧）
int interlaced_frame: 是否是隔行扫描
uint8_t motion_subsample_log2: 一个宏块中的运动矢量采样个数，取 log 的
其他的变量不再一一列举，源代码中都有详细的说明。在这里重点分析一下几个需要一定的理解的变量：

```

## 1.data[]

对于 packed 格式的数据（例如 RGB24），会存到 data[0]里面。

对于 planar 格式的数据（例如 YUV420P），则会分开成 data[0], data[1], data[2]...（YUV420P 中 data[0]存 Y, data[1]存 U, data[2]存 V）

具体参见：FFMPEG 实现 YUV, RGB 各种图像原始数据之间的转换（swscale）

## 2.pict\_type

包含以下类型：

py

```

enum AVPictureType {
    AV_PICTURE_TYPE_NONE = 0, ///< Undefined
    AV_PICTURE_TYPE_I,      ///< Intra
    AV_PICTURE_TYPE_P,      ///< Predicted
    AV_PICTURE_TYPE_B,      ///< Bi-dir predicted
    AV_PICTURE_TYPE_S,      ///< S(GMC)-VOP MPEG4
    AV_PICTURE_TYPE_SI,     ///< Switching Intra
    AV_PICTURE_TYPE_SP,     ///< Switching Predicted
}

```

```

AV_PICTURE_TYPE BI,    ///

```

```

typedef struct AVRational{
    int num; ///

```

#### 4.qscale\_table

QP 表指向一块内存，里面存储的是每个宏块的 QP 值。宏块的标号是从左往右，一行一行的来的。每个宏块对应 1 个 QP。

qscale\_table[0]就是第 1 行第 1 列宏块的 QP 值； qscale\_table[1]就是第 1 行第 2 列宏块的 QP 值； qscale\_table[2]就是第 1 行第 3 列宏块的 QP 值。以此类推...

宏块的个数用下式计算：

注：宏块大小是 16x16 的。

每行宏块数：

```
int mb_stride = pCodecCtx->width/16+1
```

宏块的总数：

```
int mb_sum = ((pCodecCtx->height+15)>>4)*(pCodecCtx->width/16+1)
```

#### 5.motion\_subsample\_log2

1 个运动矢量所能代表的画面大小（用宽或者高表示，单位是像素），注意，这里取了 log2。

代码注释中给出以下数据：

4->16x16, 3->8x8, 2-> 4x4, 1-> 2x2

即 1 个运动矢量代表 16x16 的画面的时候，该值取 4； 1 个运动矢量代表 8x8 的画面的时候，该值取 3...以此类推

#### 6.motion\_val

运动矢量表存储了一帧视频中的所有运动矢量。

该值的存储方式比较特别：

```
int16_t (*motion_val[2])[2];
```

为了弄清楚该值究竟是怎么存的，花了我好一阵子功夫...

注释中给了一段代码：

```

int mv_sample_log2= 4 - motion_subsample_log2;
int mb_width= (width+15)>>4;
int mv_stride= (mb_width << mv_sample_log2) + 1;
motion_val[direction][x + y*mv_stride][0->mv_x, 1->mv_y];

```

大概知道了该数据的结构：

1.首先分为两个列表 L0 和 L1

2.每个列表（L0 或 L1）存储了一系列的 MV（每个 MV 对应一个画面，大小由 motion\_subsample\_log2 决定）

3.每个 MV 分为横坐标和纵坐标（x,y）

注意，在 FFMPEG 中 MV 和 MB 在存储的结构上是没有什么关联的，第 1 个 MV 是屏幕上左上角画面的 MV（画面的大小取决于 motion\_subsample\_log2），第 2 个 MV 是屏幕上第 1 行第 2 列的画面的 MV，以此类推。因此在一个宏块（16x16）的运动矢量很有可能如下图所示（line 代表一行运动矢量的个数）：

//例如 8x8 划分的运动矢量与宏块的关系：

```

//-----
//|      |      |
//|mv[x] |mv[x+1] |
//-----
//|      |      |
//|mv[x+line]|mv[x+line+1]|
//-----

```

## 7.mb\_type

宏块类型表存储了一帧视频中的所有宏块的类型。其存储方式和 QP 表差不多。只不过其是 uint32 类型的，而 QP 表是 uint8 类型的。每个宏块对应一个宏块类型变量。

宏块类型如下定义所示：

```

//The following defines may change, don't expect compatibility if you use them.
#define MB_TYPE_INTRA4x4    0x0001
#define MB_TYPE_INTRA16x16 0x0002 //FIXME H.264-specific
#define MB_TYPE_INTRA_PCM   0x0004 //FIXME H.264-specific
#define MB_TYPE_16x16        0x0008
#define MB_TYPE_16x8         0x0010
#define MB_TYPE_8x16         0x0020
#define MB_TYPE_8x8          0x0040
#define MB_TYPE_INTERLACED 0x0080
#define MB_TYPE_DIRECT2     0x0100 //FIXME
#define MB_TYPE_ACPRED      0x0200
#define MB_TYPE_GMC         0x0400
#define MB_TYPE_SKIP        0x0800
#define MB_TYPE_P0L0        0x1000
#define MB_TYPE_P1L0        0x2000
#define MB_TYPE_P0L1        0x4000
#define MB_TYPE_P1L1        0x8000
#define MB_TYPE_L0           (MB_TYPE_P0L0 | MB_TYPE_P1L0)
#define MB_TYPE_L1           (MB_TYPE_P0L1 | MB_TYPE_P1L1)
#define MB_TYPE_L0L1         (MB_TYPE_L0 | MB_TYPE_L1)
#define MB_TYPE_QUANT        0x00010000
#define MB_TYPE_CBP          0x00020000

//Note bits 24-31 are reserved for codec specific use (h264 ref0, mpeg1 0mv, ...)

```

一个宏块如果包含上述定义中的一种或两种类型，则其对应的宏块变量的对应位会被置 1。

注：一个宏块可以包含好几种类型，但是有些类型是不能重复包含的，比如说一个宏块不可能既是 16x16 又是 8x8。

## 8.ref\_index

运动估计参考帧列表存储了一帧视频中所有宏块的参考帧索引。这个列表其实在比较早的压缩编码标准中是没有什么用的。只有像 H.264 这样的编码标准才有多参考帧的概念。但是这个字段目前我还没有研究透。只是知道每个宏块包含有 4 个该值，该值反映的是参考帧的索引。以后有机会再进行细研究吧。

在这里展示一下自己做的码流分析软件的运行结果。将上文介绍的几个列表图像化显示了出来（在这里是使用 MFC 的绘图函数画出来的）

视频帧：



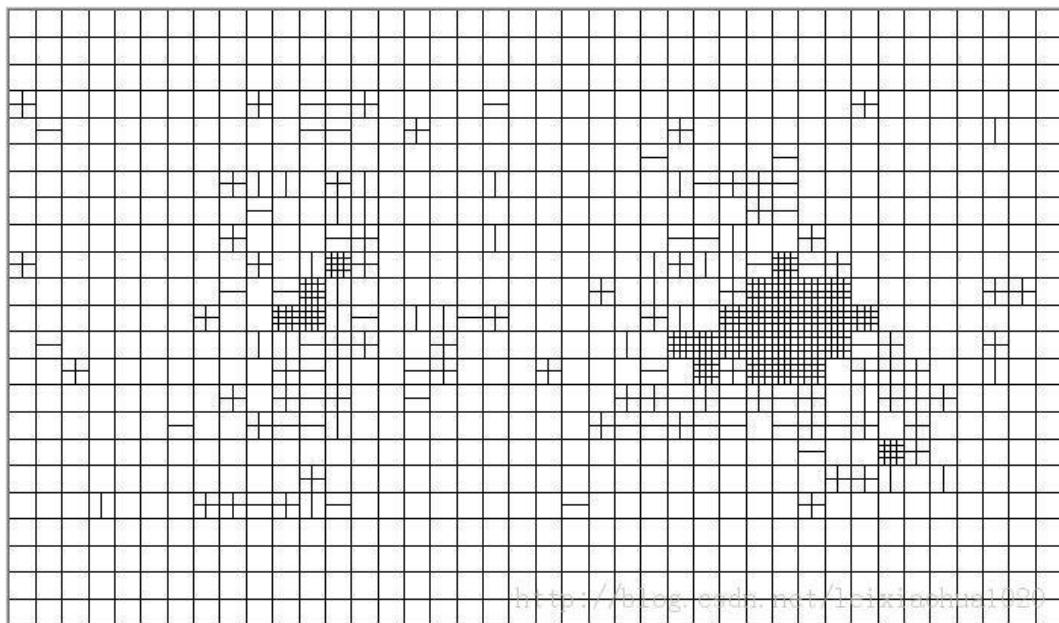
QP 参数提取的结果：

26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26														
26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26													
26	26	26	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19													
17	15	15	15	15	15	13	13	13	16	16	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14													
12	12	12	12	12	12	12	12	17	12	12	12	16	16	16	16	16	16	16	16	16	14	14	17	17	17	17	17	17													
16	16	16	16	16	16	16	16	16	18	18	18	15	15	15	15	15	15	15	15	15	18	14	14	14	18	18	18	18													
16	16	16	16	16	16	16	16	16	18	22	25	21	23	23	15	15	15	15	15	15	18	18	18	18	20	20	20	20													
20	20	17	17	17	17	17	17	15	17	20	22	22	19	19	19	19	19	19	19	19	19	24	19	21	21	24	19	15	15												
19	19	19	19	17	15	15	13	15	15	24	24	24	24	24	24	24	24	24	24	24	21	25	17	20	20	25	17	17	17												
17	17	17	17	17	17	17	22	22	22	24	24	24	26	23	20	20	20	20	20	20	23	23	23	19	22	19	24	22	13	13											
13	13	13	13	13	13	17	17	20	26	23	25	23	28	28	21	19	19	19	19	19	21	19	19	25	22	24	22	22	24	24	20	20	11	11							
11	11	11	11	18	18	18	18	21	25	25	25	28	28	28	21	19	16	20	20	20	20	22	25	25	25	25	25	25	25	25	22	22	27	27	27	27	19				
15	18	12	12	12	12	21	21	26	26	24	24	28	26	24	21	21	19	19	19	19	19	22	22	22	27	27	27	25	22	24	24	24	24	24	24	24					
24	18	9	16	16	18	18	18	21	21	28	25	25	19	19	19	19	19	19	19	19	19	19	24	24	24	22	20	20	25	22	24	21	21	21	16	16	16				
16	18	8	8	8	18	18	18	18	18	18	20	27	23	23	23	19	19	19	19	19	19	23	21	21	24	22	22	25	25	27	23	23	19	19	19	19	19	19			
19	19	13	13	13	13	13	13	13	13	19	19	23	26	26	26	21	21	21	21	21	21	21	23	23	23	25	27	24	24	24	26	24	24	24	18	18	18	18			
18	18	18	18	18	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	17	17	17	17			
19	19	19	21	21	21	21	21	21	21	23	25	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	18	15	15	15	15	
15	15	15	15	15	15	17	20	20	20	20	20	24	24	24	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	15	15	17	30	28	25	25	25	21	17	17	17
17	17	17	17	17	17	17	17	17	17	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	20	20	20	20	20	
20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20		
29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	
29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29		

美化过的（加上了颜色）：

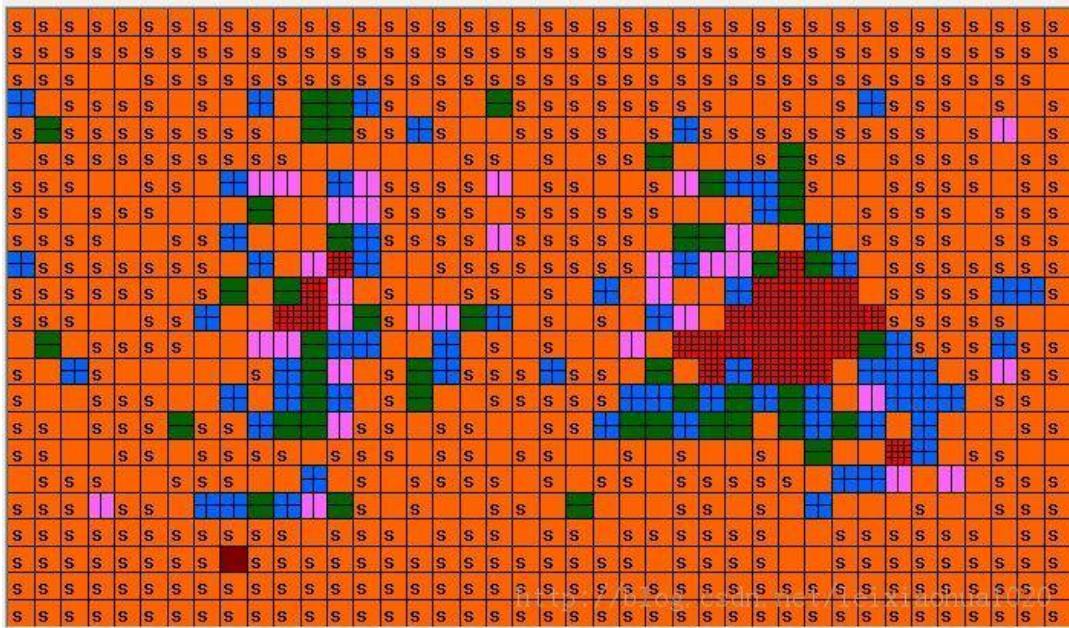
26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26		
26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26		
26	26	26	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19		
17	15	15	15	15	15	15	13	13	13	16	16	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	12	
12	12	12	12	12	12	12	12	12	12	17	12	12	12	16	16	16	16	16	16	14	14	17	17	17	17	17	11	
16	16	16	16	16	16	16	16	16	16	16	16	16	16	18	18	18	15	15	15	15	15	18	14	14	14	14		
16	16	16	16	16	16	16	18	22	25	21	23	23	15	15	15	15	15	15	18	18	18	18	18	20	20	20		
20	20	17	17	17	17	17	17	17	17	17	20	22	22	19	19	19	19	19	19	24	19	21	21	24	19	15	15	
19	19	19	19	17	15	15	13	13	15	15	24	24	24	24	24	24	24	24	24	24	21	25	17	20	20	20	19	
17	17	17	17	17	17	17	17	17	17	17	22	22	24	24	24	26	23	20	20	20	20	20	23	23	19	22	19	
13	13	13	13	13	13	13	17	17	20	26	23	25	23	28	28	21	19	19	19	19	19	21	19	19	25	22	24	
11	11	11	11	18	18	18	18	18	21	25	25	25	28	28	28	21	19	16	20	20	20	20	22	25	25	25	22	
15	18	12	12	12	12	12	21	21	26	26	24	24	28	26	24	21	21	19	19	19	19	22	22	22	27	27	19	
24	18	9	16	16	18	18	18	18	21	21	28	25	25	19	19	19	19	19	19	19	19	24	24	24	24	24	24	
16	18	8	8	8	18	18	18	18	18	20	27	23	23	23	19	19	19	19	19	19	23	21	21	24	22	22	25	25
16	18	8	8	8	18	18	18	18	18	20	27	23	23	23	19	19	19	19	19	19	23	21	21	21	16	16	16	16
19	19	13	13	13	13	13	13	13	19	19	23	26	26	26	21	21	21	21	21	21	21	23	23	25	27	24	24	18
18	18	18	18	18	18	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	26	26	17	17
19	19	19	19	21	21	21	21	21	21	21	23	25	25	22	22	22	22	22	22	22	22	22	22	22	22	22	18	15
15	15	15	15	15	15	15	17	20	20	20	20	24	24	24	20	20	20	20	20	20	20	20	20	20	20	20	17	17
17	17	17	17	17	17	17	17	17	17	17	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	27	19
20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29
29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29

宏块类型参数提取的结果:

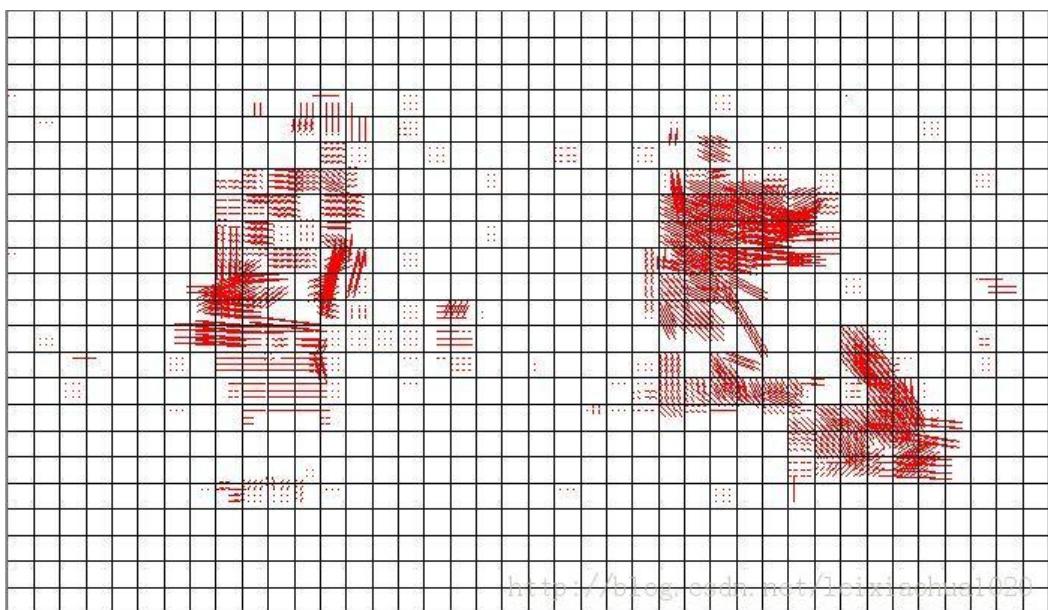


美化过的（加上了颜色，更清晰一些，s 代表 skip 宏块):

## 《FFmpeg 基础库编程开发》



运动矢量参数提取的结果（在这里是 List0）：



运动估计参考帧参数提取的结果：



## 4.3 AVInputFormat 结构体

```

typedef struct AVInputFormat
{
    // 标示 format 的名字, 比如, "mov" "mp4" 等。
    const char *name;
    // 标示具体的 format 对应的 Context 的 size, 如: MovContext。
    int priv_data_size;
    // 具体的操作函数
    int(*read_probe)(AVProbeData*);
    int(*read_header)(struct AVFormatContext *,AVFormatParameters *ap);
    int(*read_packet)(struct AVFormatContext *, AVPacket *pkt);
    int(*read_close)(struct AVFormatContext* );
    struct AVInputFormat *next;
} AVInputFormat;
Mov 或 mp4 的主要结构的初始化如下:
AVInputFormat ff_mov_demuxer = {
    "mov,mp4,m4a,3gp,3g2,mj2",
    NULL_IF_CONFIG_SMALL("QuickTime/MPEG-4/Motion JPEG 2000           format"),
    sizeof(MOVContext),
    mov_probe,
    mov_read_header,
    mov_read_packet,
    mov_read_close,
    mov_read_seek,
}
说明:
AVInputFormat 是类似 COM 接口的数据结构, 表示输入文件容器格式, 着重于功能函数, 一种文件容器格式对应一个 AVInputFormat 结构, 在程序运行时有多个实例。next 变量用于把所有支持的输入文件容器格式连接成链表, 便于遍历查找。priv_data_size 标示具体的文件容器格式对应的 Context 的大小, 在本例中是 MovContext, 这些具体的结够定义散落于各个.c 文件中。

```

## 4.4 AVFormatContext 结构体

```

typedef struct AVFormatContext
{
    // 指向 AVInputFormat, 如对于 mp4 或 mov 为 ff_mov_demuxer
    struct AVInputFormat *iformat;
    // 指向具体的格式对应的 Context, 如: MovContext。
    void *priv_data;
    // 指向数据读取统一接口 context
    ByteIOContext pb;
}

```

```
//流的数目
int nb_streams;
//至少 2 个指针元素分别指向 video stream 和 audio stream
AVStream *streams[MAX_STREAMS];
} AVFormatContext;
```

说明：

AVFormatContext 结构表示程序运行的当前文件容器格式使用的上下文，着重于所有文件容器共有的属性(并且是在程序运行时才能确定其值)和关联其他结构的字段。iformat 字段关联相应的文件容器格式；pb 关联广义的输入文件；streams 关联音视频流；priv\_data 字段关联各个具体文件容器独有的属性上下文，和 priv\_data\_size 配对使用。

## 4.5 MovContext 结构体

```
typedef struct MovContext
{
    //临时持有 AVFormatContext 的指针
    AVFormatContext *fc;

    //时间缩放因子
    int time_scale;
    //视频的时长
    int64_t duration;
    //拆包时是否发现“moov”头
    int found_moov;
    //拆包时是否发现“mdat”头
    int found_mdat;

    int isom;
    MOVFragment fragment;
    MOVTrackExt *trex_data;
    unsigned trex_count;
    int itunes_metadata; // < metadata are itunes style
    int chapter_track;
} MovContext;
```

说明：

MovContext 定义了 mp4 中流的一些属性。

## 4.6 URLProtocol 结构体

```
typedef struct URLProtocol
{
    const char *name;
    //用的统一的模板函数
    int(*url_open)(URLContext *h, const char *filename, int flags);
    int(*url_read)(URLContext *h, unsigned char *buf, int size);
```

```

int(*url_write)(URLContext *h, unsigned char *buf, int size);
offset_t(*url_seek)(URLContext *h, offset_t pos, int whence);
int(*url_close)(URLContext *h);
struct URLProtocol *next;
} URLProtocol;ffurl_connect
file 的主要结构的初始化如下:
URLProtocol ff_file_protocol =
{
    .name          = "file",
    .url_open      = file_open,
    .url_read      = file_read,
    .url_write     = file_write,
    .url_seek      = file_seek,
    .url_close     = file_close,
    .url_get_file_handle = file_get_handle,
    .url_check     = file_check,
}

```

说明:

URLProtocol 是类似 COM 接口的数据结构，表示广义的输入文件，着重于功能函数，一种广义的输入文件对应一个 URLProtocol 结构，比如 file, pipe, tcp 等等，定义了对 file tcp 等方式的通用模板函数。next 变量用于把所有支持的广义的输入文件连接成链表，便于遍历查找。

## 4.7 URLContext 结构体

```

typedef struct URLContext
{
    //指向相应的协议(协议为从初始化链表中注册的),如 ff_file_protocol
    struct URLProtocol *prot;
    int flags;
    int max_packet_size;
    //相应通信方式的句柄，对于文件为 fd 句柄，对于网络为 socket 句柄等
    void *priv_data;
    //文件的名字，不区分本地和网络
    char *filename;
} URLContext

```

说明:

URLContext 结构表示程序运行的当前广义输入文件使用的 context，着重于所有广义输入文件共有的属性(并且是在程序运行时才能确定其值)和关联其他结构的字段。prot 字段关联相应的广义输入文件；priv\_data 字段关联各个具体广义输入文件的句柄。

## 4.8 AVIOContext 结构体(老版本为: ByteIOContext)

```

typedef struct ByteIOContext
{

```

```

//数据缓冲区
unsigned char *buffer;

//数据缓冲 size
int buffer_size;
//数据读取标记指针
unsigned char *buf_ptr, *buf_end;
//该指针指向相应的 URLContext, 关联 URLContext
void *opaque;
int (*read_packet)(void *opaque, uint8_t *buf, int buf_size);
int (*write_packet)(void *opaque, uint8_t *buf, int buf_size);
offset_t(*seek)(void *opaque, offset_t offset, int whence);
//当前 buffer 在文件中的位置
offset_t pos;
//表示要进行 seek, 冲刷数据
int must_flush;
//是否到达了文件末尾
int eof_reached; // true if eof reached
int write_flag;
int max_packet_size;
int error; // contains the error code or 0 if no error happened
} ByteIOContext;
说明:
ByteIOContext 结构扩展 URLProtocol 结构成内部有缓冲机制的广泛意义上的文件, 改善广义输入文件的 IO 性能。
由其数据结构定义的字段可知, 主要是缓冲区相关字段, 标记字段, 和一个关联字段 opaque 来完成广义文件读写操作。opaque 关联字段用于关联 URLContext 结构, 间接关联并扩展 URLProtocol 结构。

```

## 4.9 AVStream 结构体

```

typedef struct AVStream
{
//指向解码器 context, 用于关联解码器
AVCodecContext *actx;
//codec 解析器, 每一种编码器在进行压缩时都会对实际负载数据进行封装, 加//入头信息, 如 h264, 需要解析 nal
单元, 关联通过 avav_find_stream_info()
    struct AVCodecParserContext *parser;
//指向解复用的流的 context, 比如 mp4 的 MovStreamcontext
    void *priv_data;
    AVRational time_base;
//用于 seek 时使用, 用于快速索引关键帧, 如 flv 的 keyframes 索引表和 mp4 的 I
//帧的索引表都存于此, 很重要
    AVIndexEntry *index_entries;
//index_entries 的元素的个数
    int nb_index_entries;
}

```

```
int index_entries_allocated_size;
double frame_last_delay;
} AVStream;
```

说明：

AVStream 结构表示当前媒体流的上下文，着重于所有媒体流共有的属性(并且是在程序运行时才能确定其值)和关联其他结构的字段。actx 字段关联当前音视频媒体使用的编解码器的 context; priv\_data 字段关联解析各个具体媒体流解复用拆包用的 context; 还有关键帧的索引表也存于此。

## 4.10 MOVStreamContext 结构体

```
typedef struct MOVStreamContext {
    //流的索引,0 或者 1
    int ffindex;
    //临时变量，保存下一个 chunk 块的编号
    int next_chunk;
    //chunk 的个数(在 mp4 的文件格式中,从 stco 中取值肯定为 chunk 的总数)
    unsigned int chunk_count;
    //chunk 在文件中的偏移量数组(每个 chunk 中的 sample 在文件中的物理存储 //是连续 的),用于保存 scto 表
    int64_t *chunk_offsets;
    //stts 的元素的个数
    unsigned int stts_count;
    //stts 时间数据表
    MOVStts *stts_data;
    //ctts(用于在有 B 帧混合时进行纠正时间戳)的元素的个数
    unsigned int ctts_count;
    //ctts 数据表
    MOVStts *ctts_data;
    //stsc(空间分布表)的元素的个数
    unsigned int stsc_count;
    //stsc 数据表
    MOVStsc *stsc_data;

    //临时变量，记录当前使用的 ctts 表的索引
    int ctts_index;
    //记录当前的 ctts 元素作用的 sample 的索引
    int ctts_sample;

    //stsz 表中可能 smaple 的 size 相同，如果相同使用该值
    unsigned int sample_size;
    //stsz 中元素的个数
    unsigned int sample_count;//sample 的个数
    //stsz 数据表，记录每个 sample 的 size，如果 sample_size=0，该表才不会 //空
    int *sample_sizes;
    //stss(关键帧索引表)中元素的个数
```

```
unsigned int keyframe_count;
//关键帧数据表
int *keyframes;

//dref 的元素的个数，一般为 1
unsigned drefs_count;
//dref 数据表
MOVDrRef *drefs;
    //tkhd 宽度
int width;

//tkhd 高度
int height;
} MOVStreamContext;
说明：
MOVStreamContext 结构用于保存从 mov 或 mp4 中进行拆包解复用从头部得到的信息。
```

## 4.11 AVPacket 结构体

```
typedef struct AVPacket
{
    //显示时间戳
    int64_t pts;
    //解码时间戳
    int64_t dts;
    //记录在文件或网络中的流中的字节的位置
    int64_t pos;
    //实际数据指针
    uint8_t *data;
    //实际的数据的大小
    int size;
    //该 packet 所属的流的索引，一般为 0 或者 1
    int stream_index;
    int flags;
    //析构函数
    void(*destruct)(struct AVPacket*);
} AVPacket;
说明：
AVPacket 代表音视频数据帧，固有的属性是一些标记，时钟信息，和压缩数据首地址，大小等信息。
```

## 4.12 AVPacketList 结构体

```
typedef struct AVPacketList
{
```

```

AVPacket pkt;
struct AVPacketList *next;
} AVPacketList;

```

说明：AVPacketList 把音视频 AVPacket 组成一个小链表。

## 第五章 重要模块

介绍几个常用模块及其函数实现，有贴代码的嫌疑。（下面分析的代码是较老版本的，新版本部分已经不适用了，但是具有一定参考价值，初期熟悉 api 的时候可以不理会具体的代码实现）

ps：下列文件列表中的大小均为裁剪后的大小，非源码中实际代码带大小。

### 5.1 libavutil 公共模块

#### 1 文件列表

文件类型	文件名	大小(bytes)
h	common.h	1515
h	bswap.h	489
h	rational.h	257
h	mathematics.h	153
h	avutil.h	1978

#### 2 common.h 文件

##### 2.1 功能描述

ffplay 使用的工具类数据类型定义，宏定义和两个简单的内联函数，基本上是自注释的。

##### 2.2 文件注释

```

1
2  #ifndef COMMON_H
3  #define COMMON_H
4
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <string.h>
8  #include <ctype.h>
9
10 #if defined(WIN32) && !defined( MINGW32 ) && !defined( CYGWIN )
11 #define CONFIG_WIN32
12 #endif
13

```

内联函数的关键字在 linux gcc 和 windows vc 中的定义是不同的，gcc 是 in line，vc 是 \_\_in line。因为代码是从 linux 下移植过来的，在这里做一个宏定义修改相对简单。

```
14 #ifdef CONFIG_WIN32
15 #define inline inline
16 #endif
17
简单的数据类型定义，linux gcc 和 windows vc 编译器有稍许不同，用宏开关 CONFIG_WIN32 来屏蔽 64
位整数类型的差别。
18 typedef signed char int8_t;
19 typedef signed short int16_t;
20 typedef signed int int32_t;
21 typedef unsigned char uint8_t;
22 typedef unsigned short uint16_t;
23 typedef unsigned int uint32_t;
24
25 #ifdef CONFIG_WIN32
26 typedef signed int64 int64_t;
27 typedef unsigned int64 uint64_t;
28 #else
29 typedef signed long long int64_t;
30 typedef unsigned long long uint64_t;
31 #endif
32
```

64 位整数的定义语法，linux gcc 和 windows vc 编译器有稍许不同，用宏开关 CONFIG\_WIN32 来屏蔽 64  
位整数定义的差别。

Linux 用 LL/ULL 来表示 64 位整数，VC 用 i64 来表示 64 位整数。

## 是连接符，把##前后的两个字符串连接成一个字符串。

```
33 #ifdef CONFIG_WIN32
34 #define int64_t_C(c) (c ## i64)
35 #define uint64_t_C(c) (c ## i64)
36 #else
37 #define int64_t_C(c) (c ## LL)
38 #define uint64_t_C(c) (c ## ULL)
39 #endif
40
```

定义最大的 64 位整数。

```
41 #ifndef INT64_MAX
42 #define INT64_MAX int64_t_C(9223372036854775807)
43 #endif
44
```

大小写敏感的字符串比较函数。在 ffplay 中只关心是否相等，不关心谁大谁小。

```
45 static int strcasecmp(char *s1, const char *s2)
46 {
47     while (toupper((unsigned char) *s1) == toupper((unsigned char) *s2++))
```

```

48 if (*s1++ == '\0')
49 return 0;
50
51 return (toupper((unsigned char) *s1) - toupper((unsigned char) *--s2));
52 }
53
限幅函数，这个函数使用简单的比较逻辑来实现，比较语句多，容易中断 CPU 的指令流水线，导致性能低下。如果变量 a 的取值范围比较小，可以用常规的空间换时间的查表方法来优化。
54 static inline int clip(int a, int amin, int amax)
55 {
56 if (a < amin)
57 return amin;
58 else if (a > amax)
59 return amax;
60 else
61 return a;
62 }
63
64 #endif

```

## 3 bswap.h 文件

### 3.1 功能描述

short 和 int 整数类型字节顺序交换，通常和 CPU 大端或小端有关。  
 对 int 型整数，小端 CPU 低地址内存存低位字节，高地址内存存高位字节。对 int 型整数，大端 CPU 低地址内存存高位字节，高地址内存存低位字节。  
 常见的 CPU 中，Intel X86 序列及其兼容序列只能是小端，Motorola 68 序列只能是大端，ARM 大端小端都 支持，但默认小端。

### 3.2 文件注释

```

1 #ifndef BSWAP_H
2 #define BSWAP_H
3
Int 16 位短整数字节交换，简单的移位再或运算。
4 static inline uint16_t bswap_16(uint16_t x)
5 {
6     return (x >> 8) | (x << 8);
7 }
8
Int 32 位长整数字节交换，看遍所有的开源代码，这个代码是最简洁的 C 代码，并且和上面 16 位短整数字节交换一脉相承。
9 static inline uint32_t bswap_32(uint32_t x)
10 {
11     x = ((x << 8) & 0xFF00FF00) | ((x >> 8) & 0x00FF00FF);
12     return (x >> 16) | (x << 16);

```

```

13 }
14
15 // be2me ... BigEndian to MachineEndian
16 // le2me ... LittleEndian to MachineEndian
17
18 #define be2me_16(x) bswap_16(x)
19 #define be2me_32(x) bswap_32(x)
20 #define le2me_16(x) (x)
21 #define le2me_32(x) (x)
22
23 #endif

```

---

## 4 rational.h 文件

### 4.1 功能描述

用两整数精确表示分数。常规的可以用一个 float 或 double 型数来表示分数，但不是精确表示，在需要相对比较精确计算的时候，为避免非精确表示带来的计算误差，采用两整数来精确表示。

### 4.2 文件注释

```
1 #ifndef RATIONAL_H
```

```
2 #define RATIONAL_H
```

```
3
```

用分数最原始的分子和分母的定义来表示，用分子和分母的组合来表示分数。

```
4 typedef struct AVRational
```

```
5 {
```

```
6     int num; // numerator // 分子
```

```
7     int den; // denominator // 分母
```

```
8 } AVRational;
```

```
9
```

用 float 或 double 表示分数值，强制类型转换后，简单的除法运算。

```
10 static inline double av_q2d(AVRational a)
```

```
11 {
```

```
12     return a.num / (double)a.den;
```

```
13 }
```

```
1415 #endif
```

---

## 5 mathematics.h 文件

### 5.1 功能描述

数学上的缩放运算。为避免计算误差，缩放因子用两整数表示做精确的整数运算。为防止计算溢出，强制转换为 int 64 位整数后计算。

此处做了一些简化，运算精度会降低，但普通的人很难感知到计算误差。

### 5.2 文件注释

```
1 #ifndef MATHEMATICS_H
```

```
2 #define MATHEMATICS_H
```

3

~~数学上的缩放运算，此处简化了很多，虽然计算结果有稍许误差，但不影响播放效果。~~

```
4 static inline int64_t av_rescale(int64_t a, int64_t b, int64_t c)
5 {
6     return a * b / c;
7 }
9 #endif
```

## 6 avutil.h 文件

### 6.1 功能描述

ffplay 基础工具库使用的一些常数和宏的定义。

### 6.2 文件注释

```
1 #ifndef AVUTIL_H
2 #define AVUTIL_H
3
4 #ifdef __cplusplus
5 extern "C" {
6 #endif
7
8 #define AV_STRINGIFY(s) AV_TOSTRING(s)
9 #define AV_TOSTRING(s) #s
10
11 #define LIBAVUTIL_VERSION_INT ((49<<16)+(0<<8)+0)
12 #define LIBAVUTIL_VERSION 49.0.0
13 #define LIBAVUTIL_BUILD LIBAVUTIL_VERSION_INT
14
15 #define LIBAVUTIL_IDENT "Lavu" AV_STRINGIFY(LIBAVUTIL_VERSION)
16
17 #include "common.h"
18 #include "mathematics.h"
19 #include "rational.h"
```

~~像素格式的宏定义，便于代码编写和维护。把一些常数定义成有意义的宏是一个值得鼓励的好习惯。~~

```
21 enum PixelFormat
22 {
23     PIX_FMT_NONE=-1,
24     PIX_FMT_YUV420P, // Planar YUV 4:2:0 (1 Cr & Cb sample per 2x2 Y samples)
25     PIX_FMT_YUV422, // Packed pixel, Y0 Cb Y1 Cr
26     PIX_FMT_RGB24, // Packed pixel, 3 bytes per pixel, RGBRGB...
27     PIX_FMT_BGR24, // Packed pixel, 3 bytes per pixel, BGRBGR...
28     PIX_FMT_YUV422P, // Planar YUV 4:2:2 (1 Cr & Cb sample per 2x1 Y samples)
29     PIX_FMT_YUV444P, // Planar YUV 4:4:4 (1 Cr & Cb sample per 1x1 Y samples)
```

---

```

30 PIX_FMT_RGBA32, // Packed pixel, 4 bytes per pixel, BGRABGRA..., stored in cpu endianness
31 PIX_FMT_YUV410P, // Planar YUV 4:1:0 (1 Cr & Cb sample per 4x4 Y samples)
32 PIX_FMT_YUV411P, // Planar YUV 4:1:1 (1 Cr & Cb sample per 4x1 Y samples)
33 PIX_FMT_RGB565, // always stored in cpu endianness
34 PIX_FMT_RGB555, // always stored in cpu endianness, most significant bit to 1
35 PIX_FMT_GRAY8,
36 PIX_FMT_MONOWHITE, // 0 is white
37 PIX_FMT_MONOBLACK, // 0 is black
38 PIX_FMT_PAL8, // 8 bit with RGBA palette
39 PIX_FMT_YUVJ420P, // Planar YUV 4:2:0 full scale (jpeg)
40 PIX_FMT_YUVJ422P, // Planar YUV 4:2:2 full scale (jpeg)
41 PIX_FMT_YUVJ444P, // Planar YUV 4:4:4 full scale (jpeg)
42 PIX_FMT_XVMC_MPEG2_MC, // XVideo Motion Acceleration via common packet passing(xvmc_render.h)
43 PIX_FMT_XVMC_MPEG2_IDCT,
44 PIX_FMT_UYVY422, // Packed pixel, Cb Y0 Cr Y1
45 PIX_FMT_UYVY411, // Packed pixel, Cb Y0 Y1 Cr Y2 Y3
46 PIX_FMT_NB,
47 };
48
49 #ifdef cplusplus
50 }
51 #endif
52
53 #endif

```

## 5.2 libavcodec 编解码模块

### 1 文件列表

文件类型	文件名	大小(bytes)
	avcodec.h	4943
	allcodecs.c	310
	dsputil.h	163
	dsputil.c	350
	imgconvert_template.h	22311
	imgconvert.c	47834
	msrle.c	8387
	turespeech_data.h	4584
	turespeech.c	9622
	utils_codec.c	8973

## 2 avcodec.h 文件

### 2.1 功能描述

定义编解码器库使用的宏、数据结构和函数，通常这些宏、数据结构和函数在此模块内相对全局有效。

### 2.2 文件注释

```

1 #ifndef AVCODEC_H
2 #define AVCODEC_H
3
4 #ifdef cplusplus
5 extern "C"
6 {
7 #endif
8
9 #include "../libavutil/avutil.h"
10 #include <sys/types.h> // size_t
11

```

和版本信息有关的几个宏定义

```

12 #define FFMPEG_VERSION_INT 0x000409
13 #define FFMPEG_VERSION"CVS"
14
15 #define AV_STRINGIFY(s) AV_TOSTRING(s)
16 #define AV_TOSTRING(s) #s
17
18 #define LIBAVCODEC_VERSION_INT ((51<<16)+(8<<8)+0)
19 #define LIBAVCODEC_VERSION 51.8.0
20 #define LIBAVCODEC_BUILD LIBAVCODEC_VERSION_INT
21
22 #define LIBAVCODEC_IDENT "Lavc" AV_STRINGIFY(LIBAVCODEC_VERSION)
23
24 #define AV_NOPTS_VALUE int64_t_C(0x8000000000000000)
25 #define AV_TIME_BASE 1000000
26

```

Codec ID 宏定义，瘦身后的 ffplay 只支持这两种 codec，其他的都删掉了。

```

27 enum CodecID
28 {
29 CODEC_ID_TRUESPEECH,
30 CODEC_ID_MSRLE,
31 CODEC_ID_NONE
32 };
33

```

Codec 类型定义，瘦身后的 ffplay 只支持视频和音频。

```

34 enum CodecType
35 {
36 CODEC_TYPE_UNKNOWN = -1,

```

```

37 CODEC_TYPE_VIDEO,
38 CODEC_TYPE_AUDIO,
39 CODEC_TYPE_DATA
40 };
41
42 #define AVCODEC_MAX_AUDIO_FRAME_SIZE 192000 // 1 second of 48khz 32bit audio
43
44 #define FF_INPUT_BUFFER_PADDING_SIZE 8
45

AVPicture 和 AVFrame 主要表示解码过程中的使用缓存，通常帧缓存是 YUV 格式，输出格式有 YUV
也有 RGB 格式，所以定义了 4 个 data 指针来表示分量。
46 typedef struct AVPicture
47 {
48     uint8_t *data[4];
49     int linesize[4];
50 } AVPicture;
51

52 typedef struct AVFrame
53 {
54     uint8_t *data[4]; // 有多重意义，其一用 NULL 来判断是否被占用
55     int linesize[4];
56     uint8_t *base[4]; // 有多重意义，其一用 NULL 来判断是否分配内存
57 } AVFrame;
58

程序运行时当前 Codec 使用的上下文，着重于所有 Codec 共有的属性(并且是在程序运行时才能确定其
值)，codec 和 priv_data 关联其他结构的字段，便于在数据结构间跳转。
59 typedef struct AVCodecContext
60 {
61     int bit_rate;
62     int frame_number; // audio or video frame number
63
64     unsigned char *extradata; // codec 的私有数据,对 Audio 是 WAVEFORMATEX 扩展结构。
65     int extradata_size; // 对 Video 是 BITMAPINFOHEADER 扩展结构
66
67     int width, height; // video only
68
69     enum PixelFormat pix_fmt; // 输出像素格式/视频图像格式
70
71     int sample_rate; // samples per sec // audio only
72     int channels;
73     int bits_per_sample;
74     int block_align;
75
76     struct AVCodec *codec; // 指向 Codec 的指针,
77     void *priv_data; // 具体解码器属性，在本例中指向 MsrleContext 或 TSContext

```

```

78
79 enum CodecType codec_type;// see CODEC_TYPE_xxx
80 enum CodecID codec_id; // see CODEC_ID_xxx
81
82 int(*get_buffer)(struct AVCodecContext *c, AVFrame *pic);
83 void(*release_buffer)(struct AVCodecContext *c, AVFrame *pic);
84 int(*reget_buffer)(struct AVCodecContext *c, AVFrame *pic);
85
86 int internal_buffer_count;
87 void *internal_buffer;
88
89 struct AVPaletteControl *palctrl;
90 }AVCodecContext;
91
类似 COM 接口的数据结构，表示音视频编解码器，着重于功能函数，一种媒体类型对应一个 AVCodec
结构，在程序运行时有多个实例串联成链表便于查找。
92 typedef struct AVCodec
93 {
94 const char *name; // 便于阅读的友好字符串，表征编解码器名称，比如"msrle","truespeech"
95 enum CodecType type; // 编解码器类型，有效取值为 CODEC_TYPE_VIDEO 或 CODEC_TYPE_AUDIO
96 enum CodecID id; // 编解码器 ID 值,
97 int priv_data_size; // 具体编解码属性结构的大小，取代很多的 if-else 语句
98 int(*init)(AVCodecContext*);
99 int(*encode)(AVCodecContext *, uint8_t *buf, int buf_size, void *data);
100 int(*close)(AVCodecContext*);
101 int(*decode)(AVCodecContext *, void *outdata, int *outdata_size, uint8_t *buf, int buf_size);
102 int capabilities;
103
104 struct AVCodec *next; // 把所有的编解码器串联成链表便于查找
105 }AVCodec;
106
调色板大小和大小宏定义，每个调色板四字节(R,G,B,α)。有很多的视频图像颜色种类比较少，用索引 间接表示每
个像素的颜色值，就可以用调色板和索引值实现简单的大约的 4:1 压缩比。
107 #define AVPALETTE_SIZE 1024
108 #define AVPALETTE_COUNT 256
109
调色板数据结构定义，保存调色板数据。
110 typedef struct AVPaletteControl
111 {
112 // demuxer sets this to 1 to indicate the palette has changed; decoder resets to 0
113 int palette_changed;
114
115 /* 4-byte ARGB palette entries, stored in native byte order; note that
116 * the individual palette components should be on a 8-bit scale; if
117 * the palette data comes from a IBM VGA native format, the component

```

```

118 * data is probably 6 bits in size and needs to be scaled */
119 unsigned int palette[AVPALETTE_COUNT];
120
121 } AVPaletteControl;
122
编解码库使用的函数声明。
123 int avpicture_alloc(AVPicture *picture, int pix_fmt, int width, int height);
124
125 void avpicture_free(AVPicture *picture);
126
127 int avpicture_fill(AVPicture *picture, uint8_t *ptr, int pix_fmt, int width, int height);
128 int avpicture_get_size(int pix_fmt, int width, int height);
129 void avcodec_get_chroma_sub_sample(int pix_fmt, int *h_shift, int *v_shift);
130
131 int img_convert(AVPicture *dst, int dst_pix_fmt, const AVPicture *src, int pix_fmt,
132 int width, int height);
133
134 void avcodec_init(void);
135
136 void register_avcodec(Codec *format);
137 Codec *avcodec_find_decoder(enum CodecID id);
138
139 CodecContext *avcodec_alloc_context(void);
140
141 int avcodec_default_get_buffer(CodecContext *s, AVFrame *pic);
142 void avcodec_default_release_buffer(CodecContext *s, AVFrame *pic);
143 int avcodec_default_reget_buffer(CodecContext *s, AVFrame *pic);
144 void avcodec_align_dimensions(CodecContext *s, int *width, int *height);
145 int avcodec_check_dimensions(void *av_log_ctx, unsigned int w, unsigned int h);
146
147 int avcodec_open(CodecContext *avctx, Codec *codec);
148
149 int avcodec_decode_audio(CodecContext *avctx, int16_t *samples, int *frame_size_ptr,
150 uint8_t *buf, int buf_size);
151 int avcodec_decode_video(CodecContext *avctx, AVFrame *picture, int *got_picture_ptr,
152 uint8_t *buf, int buf_size);
153
154 int avcodec_close(CodecContext *avctx);
155
156 void avcodec_register_all(void);
157
158 void avcodec_default_free_buffers(CodecContext *s);
159
160 void *av_malloc(unsigned int size);
161 void *av_mallocz(unsigned int size);

```

---

```

162 void *av_realloc(void *ptr, unsigned int size);
163 void av_free(void *ptr);
164 void av_freep(void *ptr);
165 void *av_fast_realloc(void *ptr, unsigned int *size, unsigned int min_size);
166
167 void img_copy(APicture *dst, const APicture *src, int pix_fmt, int width, int height);
168
169 #ifdef cplusplus
170 }
171
172 #endif
173
174 #endif

```

---

### 3 allcodec.c 文件

#### 3.1 功能描述

简单的注册/初始化函数，把编解码器用相应的链表串起来便于查找识别。

#### 3.2 文件注释

```

1 #include "avcodec.h"
2
3 extern AVCodec truespeech_decoder;
4 extern AVCodec msrle_decoder;
5
6 void avcodec_register_all(void)
7 {

```

8 到 13 行，inited 变量声明成 static，做一下比较是为了避免此函数多次调用。

编程基本原则之一，初始化函数只调用一次，不能随意多次调用。

```

8 static int inited = 0;
9
10 if (inited != 0)
11 return ;
12
13 inited = 1;
14

```

把 msrle\_decoder 解码器串接到解码器链表，链表头指针是 first\_avcodec。

```
15 register_avcodec(&msrle_decoder);
```

16

把 truespeech\_decoder 解码器串接到解码器链表，链表头指针是 first\_avcodec。

```
17 register_avcodec(&truespeech_decoder);
```

---

```
18 }
```

## 4 dsputil.h 文件

### 4.1 功能描述

定义 dsp 优化限幅运算使用的查找表及其初始化函数。

### 4.2 文件注释

```

1  #ifndef DSPUTIL_H
2  #define DSPUTIL_H
3
4  #define MAX_NEG_CROP 1024
5
6  extern uint8_t cropTbl[256+2 * MAX_NEG_CROP];
7
8  void dsputil_static_init(void);
9
10 #endif

```

---

## 5 dsputil.c 文件

### 5.1 功能描述

定义 dsp 优化限幅运算使用的查找表，实现其初始化函数。

### 5.2 文件注释

```

1  #include "avcodec.h"
2  #include "dsputil.h"
3
4  uint8_t cropTbl[256+2 * MAX_NEG_CROP] = {0, };
5
6  void dsputil_static_init(void)
7  {
8      int i;
9
10     for (i = 0; i < 256; i++)
11         cropTbl[i + MAX_NEG_CROP] = i;
12
13     for (i = 0; i < MAX_NEG_CROP; i++)
14     {
15         cropTbl[i] = 0;
16         cropTbl[i + MAX_NEG_CROP + 256] = 255;
17     }
18 }

```

---

初始化限幅运算查找表，最后的结果是：前 MAX\_NEG\_CROP 个数组项为 0，接着的 256 个数组项分别为 0 到 255，后面 MAX\_NEG\_CROP 个数组项为 255。用查表代替比较实现限幅运算。

## 6 utils\_codec.c 文件

### 6.1 功能描述

编解码库使用的帮助和工具函数，

### 6.2 文件注释

```

1 #include "avcodec.h"
2 #include "dsputil.h"
3
4
5 #include <assert.h>
6 #include "avcodec.h"
7 #include "dsputil.h"
8
9
10 #define EDGE_WIDTH 16
11 #define STRIDE_ALIGN 16
12 #define INT_MAX 2147483647
13
14 #define FFMAX(a,b) ((a) > (b) ? (a) : (b))

15
16 void *av_malloc(unsigned int size)
17 {
18     void *ptr;
19
20     if (size > INT_MAX)
21         return NULL;
22     ptr = malloc(size);
23
24     return ptr;
25 }
26
27
28 void *av_realloc(void *ptr, unsigned int size)
29 {
30     if (size > INT_MAX)
31         return NULL;
32     ptr = realloc(ptr, size);
33
34     if (!ptr)
35         return NULL;
36
37     return ptr;
38 }

39
40 void av_free(void *ptr)
41 {
42     if (ptr)
43         free(ptr);
44 }
```

内存动态分配函数，做一下简单参数校验后调用系统函数

内存动态重分配函数，做一下简单参数校验后调用系统函数

内存动态释放函数，做一下简单参数校验后调用系统函数

```

34 free(ptr);
35 }
36

```

内存动态分配函数，复用 av\_malloc() 函数，再把分配的内存清 0.

```

37 void *av_mallocz(unsigned int size)

```

```

38 {
39 void *ptr;
40
41 ptr = av_malloc(size);
42 if (!ptr)
43 return NULL;
44
45 memset(ptr, 0, size);
46 return ptr;
47 }
48

```

快速内存动态分配函数，预分配一些内存来避免多次调用系统函数达到快速的目的。

```

49 void *av_fast_realloc(void *ptr, unsigned int *size, unsigned int min_size)

```

```

50 {
51 if (min_size < *size)
52 return ptr;
53
54 *size = FFMAX(17 *min_size / 16+32, min_size);
55
56 return av_realloc(ptr, *size);
57 }
58

```

动态内存释放函数，注意传入的变量的类型。

```

59 void av_free(void *arg)

```

```

60 {
61 void **ptr = (void **)arg;
62 av_free(*ptr);
63 *ptr = NULL;
64 }
65

```

```

66 AVCodec *first_avcodec = NULL;

```

把编解码器串联成一个链表，便于查找。

```

68 void register_avcodec(AVCodec *format)

```

```

69 {
70 AVCodec **p;
71 p = &first_avcodec;
72 while (*p != NULL)
73 p = &(*p)->next;
74 *p = format;

```

```

75 format->next = NULL;
76 }
77
78 编解码库内部使用的缓存区，因为视频图像有 RGB 或 YUV 分量格式，所以每个数组有四个分量。
79 typedef struct InternalBuffer
80 {
81     uint8_t *base[4];
82     uint8_t *data[4];
83     int linesize[4];
84 } InternalBuffer;
85
86 #define INTERNAL_BUFFER_SIZE 32
87
88 #define ALIGN(x, a) (((x)+(a)-1)&~((a)-1))
89 计算各种图像格式要求的图像长宽的字节对齐数，是 1 个还是 2 个，4 个，8 个，16 个字节对齐。
90 void avcodec_align_dimensions(AVCodecContext *s, int *width, int *height)
91 {
92     int w_align = 1;
93     int h_align = 1;
94     switch (s->pix_fmt)
95     {
96         case PIX_FMT_YUV420P:
97         case PIX_FMT_YUV422:
98         case PIX_FMT_UYVY422:
99         case PIX_FMT_YUV422P:
100        case PIX_FMT_YUV444P:
101        case PIX_FMT_GRAY8:
102        case PIX_FMT_YUVJ420P:
103        case PIX_FMT_YUVJ422P:
104        case PIX_FMT_YUVJ444P: //FIXME check for non mpeg style codecs and use less alignment
105        w_align = 16;
106        h_align = 16;
107        break;
108        case PIX_FMT_YUV411P:
109        case PIX_FMT_UYVY411:
110        w_align = 32;
111        h_align = 8;
112        break;
113        case PIX_FMT_YUV410P:
114        case PIX_FMT_RGB555:
115        case PIX_FMT_PAL8:
116        break;

```

```

117 case PIX_FMT_BGR24:
118     break;
119 default:
120     w_align = 1;
121     h_align = 1;
122     break;
123 }
124
125 *width = ALIGN(*width, w_align);
126 *height = ALIGN(*height, h_align);
127 }
128
校验视频图像的长宽是否合法。
129 int avcodec_check_dimensions(void *av_log_ctx, unsigned int w, unsigned int h)
130 {
131     if ((int)w > 0 && (int)h > 0 && (w + 128)*(uint64_t)(h + 128) < INT_MAX / 4)
132         return 0;
133
134     return -1;
135 }
136
每次取 internal_buffer_count 数据项，用 base[0] 来判断是否已分配内存，用 data[0] 来判断是否已被占用。base[] 和 data[] 有多重意义。
在 avcodec_alloc_context 中已把 internal_buffer 各项清 0，所以可以用 base[0] 来判断。
137 int avcodec_default_get_buffer(AVCodecContext *s, AVFrame *pic)
138 {
139     int i;
140     int w = s->width;
141     int h = s->height;
142     int align_off;
143     InternalBuffer *buf;
144
145     assert(pic->data[0] == NULL);
146     assert(INTERNAL_BUFFER_SIZE > s->internal_buffer_count);
147
校验视频图像的长宽是否合法。
148     if (avcodec_check_dimensions(s, w, h))
149         return -1;
150
如果没有分配内存，就分配动态内存并清 0。
151     if (s->internal_buffer == NULL)
152         s->internal_buffer = av_mallocz(INTERNAL_BUFFER_SIZE * sizeof(InternalBuffer));
153
取缓存中的第一个没有占用内存。
154     buf = &((InternalBuffer*)s->internal_buffer)[s->internal_buffer_count];

```

155

```
156 if (buf->base[0])
157 { /* 如果内存已分配就跳过 */ }
```

158 else

159 {

如果没有分配内存就按照图像格式要求分配内存，并设置一些标记和计算一些参数值。

160 int h\_chroma\_shift, v\_chroma\_shift;

161 int pixel\_size, size[3];

162

163 AVPicture picture;

164

计算 CbCr 色度分量长宽的与 Y 亮度分量长宽的比，最后用移位实现。

165 avcodec\_get\_chroma\_sub\_sample(s-&gt;pix\_fmt, &amp;h\_chroma\_shift, &amp;v\_chroma\_shift);

166

规整长宽满足特定图像像素格式的要求。

167 avcodec\_align\_dimensions(s, &amp;w, &amp;h);

168

把长宽放大一些，比如在 mpeg4 视频中编码算法中的运动估计要把原始图像做扩展来满足不受限制运动矢量的要求(运动矢量可以超出原始图像边界)。

169 w+=EDGE\_WIDTH\*2;

170 h+=EDGE\_WIDTH\*2;

171

计算特定格式的图像参数，包括各分量的大小，单行长度(linesize/stride) 等等。

172 avpicture\_fill(&amp;picture, NULL, s-&gt;pix\_fmt, w, h);

173 pixel\_size = picture.linesize[0] \* 8 / w;

174 assert(pixel\_size &gt;= 1);

175

176 if (pixel\_size == 3 \*8)

177 w = ALIGN(w, STRIDE\_ALIGN &lt;&lt; h\_chroma\_shift);

178 else

179 w = ALIGN(pixel\_size \*w, STRIDE\_ALIGN &lt;&lt; (h\_chroma\_shift + 3)) / pixel\_size;

180

181 size[1] = avpicture\_fill(&amp;picture, NULL, s-&gt;pix\_fmt, w, h);

182 size[0] = picture.linesize[0] \*h;

183 size[1] -= size[0];

184 if (picture.data[2])

185 size[1] = size[2] = size[1] / 2;

186 else

187 size[2] = 0;

188

注意 base[] 和 data[] 数组还有作为标记的用途，free() 时的非 NULL 判断，这里要清 0。

189 memset(buf-&gt;base, 0, sizeof(buf-&gt;base));

190 memset(buf-&gt;data, 0, sizeof(buf-&gt;data));

191

192 for (i = 0; i &lt; 3 &amp;&amp; size[i]; i++)

```

193 {
194 const int h_shift = i == 0 ? 0 : h_chroma_shift;
195 const int v_shift = i == 0 ? 0 : v_chroma_shift;
196
197 buf->linesize[i] = picture.linesize[i];
198
实质性分配内存，并且在 202 行把内存清 0。
199 buf->base[i] = av_malloc(size[i] + 16); //FIXME 16
200 if (buf->base[i] == NULL)
201 return -1;
202 memset(buf->base[i], 128, size[i]);
203
内存对齐计算。
204 align_off=ALIGN((buf->linesize[i]*EDGE_WIDTH>>v_shift)+(EDGE_WIDTH>>h_shift),STRIDE_ALIGN);
205
206 if ((s->pix_fmt == PIX_FMT_PAL8) || !size[2])
207 buf->data[i] = buf->base[i];
208 else
209 buf->data[i] = buf->base[i] + align_off;
210 }
211
212
213 for (i = 0; i < 4; i++)
214 {
把分配的内存参数赋值到 pic 指向的结构中，传递出去。


---


215     pic->base[i] = buf->base[i];
216     pic->data[i] = buf->data[i];
217     pic->linesize[i] = buf->linesize[i];
218 }
内存数组计数+1，注意释放时的操作，保证计数对应的内存数组是空闲的。
219 s->internal_buffer_count++;
220
221 return 0;
222 }
223
释放占用的内存数组项。保证从 0 到 internal_buffer_count-1 数据项为有效数据，其他是空闲数据项
224 void avcodec_default_release_buffer(CodecContext *s, AVFrame *pic)
225 {
226 int i;
227 InternalBuffer *buf, *last, temp;
228
简单的参数校验，内存必须是已经分配过。
229 assert(s->internal_buffer_count);
230

```

```

231 buf = NULL;
232 for (i = 0; i < s->internal_buffer_count; i++)
233 {
    遍历内存数组，查找对应 pic 的内存数组项，以 data[0] 内存地址为比较判别标记。
234 buf = &((InternalBuffer*)s->internal_buffer)[i]; //just 3-5 checks so is not worth to optimize
235 if (buf->data[0] == pic->data[0])
236 break;
237 }
238 assert(i < s->internal_buffer_count);
    内存数组计数-1，删除最后一项。
239 s->internal_buffer_count--;
240 last = &((InternalBuffer*)s->internal_buffer)[s->internal_buffer_count];
241
    把将要空闲的数组项和数组最后一项交换，保证 internal_buffer_count 计算正确无误。注意这里并
    没有内存释放的动作，便于下次复用已分配的内存。
242 temp = *buf;
243 *buf = *last;
244 *last = temp;
245
246 for (i = 0; i < 3; i++)
247 {
    把 data[i] 置空，指示本块内存没有被占用，实际分配的首地址保持在 base[] 中。
    整个程序最多分配 INTERNAL_BUFFER_SIZE 次 avframe，其他次循环使用。
248 pic->data[i] = NULL;
249 }
250 }
251
    重新获得缓存。

```

```

252 int avcodec_default_reget_buffer(AVCodecContext *s, AVFrame *pic)
253 {
254 if (pic->data[0] == NULL) // If no picture return a new buffer
255 {
256 return s->get_buffer(s, pic);
257 }
258
259 return 0;
260 }
261
    释放内存数组项占用的内存。
262 void avcodec_default_free_buffers(AVCodecContext *s)
263 {
264 int i, j;
265
266 if (s->internal_buffer == NULL)

```

```

267 return ;
268
269 for (i = 0; i < INTERNAL_BUFFER_SIZE; i++)
270 {
271 InternalBuffer *buf = &((InternalBuffer*)s->internal_buffer)[i];
272 for (j = 0; j < 4; j++)
273 {
av_free()函数调用的 av_free()函数做了非 NULL 判断，并且分配时已置 NULL，所以内循环可以到 4，  

外循环可以到 INTERNAL_BUFFER_SIZE。
274 av_free(&buf->base[j]);
275 buf->data[j] = NULL;
276 }
277 }
278 av_free(&s->internal_buffer);
279
280 s->internal_buffer_count = 0;
281 }
282

```

分配编解码器上下文占用的内存，清 0 后部分参数赋初值。

---

```
283 AVCodecContext *avcodec_alloc_context(void)
```

```
284 {
285 AVCodecContext *s = av_malloc(sizeof(AVCodecContext));
286
```

```
287 if (s == NULL)
288 return NULL;
```

```
289
注意这里的清 0。
```

```
290 memset(s, 0, sizeof(AVCodecContext));
291
292 s->get_buffer = avcodec_default_get_buffer;
293 s->release_buffer = avcodec_default_release_buffer;
294
295 s->pix_fmt = PIX_FMT_NONE;
296
297 s->palctrl = NULL;
298 s->reget_buffer = avcodec_default_reget_buffer;
299
```

```
300 return s;
301 }
```

打开编解码器，分配具体编解码器使用的上下文，简单变量赋初值，调用初始化函数初始化编解码器

```
303 int avcodec_open(AVCodecContext *avctx, AVCodec *codec)
```

```
304 {
```

```
305 int ret = -1;
```

```
306
```

```

307 if (avctx->codec)
308 goto end;
309
310 if (codec->priv_data_size > 0)
311 {
    这里体现了 priv_data_size 参数的重大作用，如果没有这个参数，就要用 codec 结构的名字比较确定具体编解码器使用的上下文结构大小，超级长的 if-else 语句。
312 avctx->priv_data = av_mallocz(codec->priv_data_size);
313 if (!avctx->priv_data)
314 goto end;
315 }
316 else
317 {
318 avctx->priv_data = NULL;
319 }
320
321 avctx->codec = codec;
322 avctx->codec_id = codec->id;
323 avctx->frame_number = 0;
324 ret = avctx->codec->init(avctx);
325 if (ret < 0)
326 {
327 av_freep(&avctx->priv_data);
328 avctx->codec = NULL;
329 goto end;
330 }
331 ret = 0;
332 end:
333 return ret;
334 }
335
    视频解码，简单的跳转
336 int avcodec_decode_video(AVCodecContext *avctx, AVFrame *picture, int *got_picture_ptr,
337 uint8_t *buf, int buf_size)
338 {
339 int ret;
340
341 *got_picture_ptr = 0;
342
343 if (buf_size)
344 {
345 ret = avctx->codec->decode(avctx, picture, got_picture_ptr, buf, buf_size);
346
347 if (*got_picture_ptr)
348 avctx->frame_number++;

```

```

349 }
350 else
351 ret = 0;
352
353 return ret;
354 }
355
356 int avcodec_decode_audio(AVCodecContext *avctx, int16_t *samples, int *frame_size_ptr,
357 uint8_t *buf, int buf_size)
358 {
359 int ret;
360
361 *frame_size_ptr = 0;
362 if (buf_size)
363 {
364 ret = avctx->codec->decode(avctx, samples, frame_size_ptr, buf, buf_size);
365 avctx->frame_number++;
366 }
367 else
368 ret = 0;
369 return ret;
370 }
371
372 int avcodec_close(AVCodecContext *avctx)
373 {
374 if (avctx->codec->close)
375 avctx->codec->close(avctx);
376 avcodec_default_free_buffers(avctx);
377 av_freep(&avctx->priv_data);
378 avctx->codec = NULL;
379 return 0;
380 }
381
382 AVCodec *avcodec_find_decoder(enum CodecID id)
383 {
384 AVCodec *p;
385 p = first_avcodec;
386 while (p)
387 {
388 if (p->decode != NULL && p->id == id)
389 return p;

```

```

390 p = p->next;


---


391 }
392 return NULL;
393 }
394

```

初始化编解码库，在本例中仅初始化限幅数组/查找表。

```

395 void avcodec_init(void)
396 {
397 static int inited = 0;
398
399 if (inited != 0)
400 return ;
401 inited = 1;
402
403 dsputil_static_init();
404 }

```

---

## 7 imgconvert\_template.h 文件

### 7.1 功能描述

定义并实现图像颜色空间转换使用的函数和宏，此文件请各位自己仔细分析。

### 7.2 文件注释

```

1 #ifndef RGB_OUT
2 #define RGB_OUT(d, r, g, b) RGBA_OUT(d, r, g, b, 0xff)
3 #endif
4
5 #pragma warning (disable:4305 4244)
6

```

此文件请各位读者自行分析，都是些颜色空间转换函数。

```

7 static void glue(yuv420p_to_, RGB_NAME)(AVPicture *dst, const AVPicture *src, int width, int height)
8 {
9     const uint8_t *y1_ptr, *y2_ptr, *cb_ptr, *cr_ptr;
10    uint8_t *d, *d1, *d2;
11    int w, y, cb, cr, r_add, g_add, b_add, width2;
12    uint8_t *cm = cropTbl + MAX_NEG_CROP;
13    unsigned int r, g, b;
14
15    d = dst->data[0];
16    y1_ptr = src->data[0];
17    cb_ptr = src->data[1];
18    cr_ptr = src->data[2];
19    width2 = (width + 1) >> 1;
20
21    for (; height >= 2; height -= 2)

```

```

22 {
23 d1 = d;
24 d2 = d + dst->linesize[0];
25 y2_ptr = y1_ptr + src->linesize[0];
26 for (w = width; w >= 2; w -= 2)
27 {
28 YUV_TO_RGB1_CCIR(cb_ptr[0], cr_ptr[0]);
29
30 YUV_TO_RGB2_CCIR(r, g, b, y1_ptr[0]); /* output 4 pixels */
31 RGB_OUT(d1, r, g, b);
32


---


33 YUV_TO_RGB2_CCIR(r, g, b, y1_ptr[1]);
34 RGB_OUT(d1 + BPP, r, g, b);
35
36 YUV_TO_RGB2_CCIR(r, g, b, y2_ptr[0]);
37 RGB_OUT(d2, r, g, b);
38
39 YUV_TO_RGB2_CCIR(r, g, b, y2_ptr[1]);
40 RGB_OUT(d2 + BPP, r, g, b);
41
42 d1 += 2 * BPP;
43 d2 += 2 * BPP;
44
45 y1_ptr += 2;
46 y2_ptr += 2;
47 cb_ptr++;
48 cr_ptr++;
49 }
50
51 if (w) /* handle odd width */
52 {
53 YUV_TO_RGB1_CCIR(cb_ptr[0], cr_ptr[0]);
54 YUV_TO_RGB2_CCIR(r, g, b, y1_ptr[0]);
55 RGB_OUT(d1, r, g, b);
56
57 YUV_TO_RGB2_CCIR(r, g, b, y2_ptr[0]);
58 RGB_OUT(d2, r, g, b);
59 d1 += BPP;
60 d2 += BPP;
61 y1_ptr++;
62 y2_ptr++;
63 cb_ptr++;
64 cr_ptr++;
65 }
66 d += 2 * dst->linesize[0];

```

```

67  y1_ptr += 2 * src->linesize[0] - width;
68  cb_ptr += src->linesize[1] - width2;
69  cr_ptr += src->linesize[2] - width2;
70  }
71
72  if (height)    /* handle odd height */
73  {


---


74  d1 = d;
75  for (w = width; w >= 2; w -= 2)
76  {
77  YUV_TO_RGB1_CCIR(cb_ptr[0], cr_ptr[0]);
78
79  YUV_TO_RGB2_CCIR(r, g, b, y1_ptr[0]); /* output 2 pixels */
80  RGB_OUT(d1, r, g, b);
81
82  YUV_TO_RGB2_CCIR(r, g, b, y1_ptr[1]);
83  RGB_OUT(d1 + BPP, r, g, b);
84
85  d1 += 2 * BPP;
86
87  y1_ptr += 2;
88  cb_ptr++;
89  cr_ptr++;
90  }
91
92  if (w)    /* handle width */
93  {
94  YUV_TO_RGB1_CCIR(cb_ptr[0], cr_ptr[0]);
95
96  YUV_TO_RGB2_CCIR(r, g, b, y1_ptr[0]); /* output 2 pixels */
97  RGB_OUT(d1, r, g, b);
98  d1 += BPP;
99
100 y1_ptr++;
101 cb_ptr++;
102 cr_ptr++;
103 }
104 }
105 }
106
107 static void glue(yuvj420p_to_, RGB_NAME)(AVPicture *dst, const AVPicture *src, int width, int height)
108 {
109 const uint8_t *y1_ptr, *y2_ptr, *cb_ptr, *cr_ptr;
110 uint8_t *d, *d1, *d2;
111 int w, y, cb, cr, r_add, g_add, b_add, width2;

```

```

112 uint8_t *cm = cropTbl + MAX_NEG_CROP;
113 unsigned int r, g, b;
114


---


115 d = dst->data[0];
116 y1_ptr = src->data[0];
117 cb_ptr = src->data[1];
118 cr_ptr = src->data[2];
119 width2 = (width + 1) >> 1;
120
121 for (; height >= 2; height -= 2)
122 {
123 d1 = d;
124 d2 = d + dst->linesize[0];
125 y2_ptr = y1_ptr + src->linesize[0];
126 for (w = width; w >= 2; w -= 2)
127 {
128 YUV_TO_RGB1(cb_ptr[0], cr_ptr[0]);
129
130 YUV_TO_RGB2(r, g, b, y1_ptr[0]); /* output 4 pixels */
131 RGB_OUT(d1, r, g, b);
132
133 YUV_TO_RGB2(r, g, b, y1_ptr[1]);
134 RGB_OUT(d1 + BPP, r, g, b);
135
136 YUV_TO_RGB2(r, g, b, y2_ptr[0]);
137 RGB_OUT(d2, r, g, b);
138
139 YUV_TO_RGB2(r, g, b, y2_ptr[1]);
140 RGB_OUT(d2 + BPP, r, g, b);
141
142 d1 += 2 * BPP;
143 d2 += 2 * BPP;
144
145 y1_ptr += 2;
146 y2_ptr += 2;
147 cb_ptr++;
148 cr_ptr++;
149 }
150
151 if (w) /* handle odd width */
152 {
153 YUV_TO_RGB1(cb_ptr[0], cr_ptr[0]);
154 YUV_TO_RGB2(r, g, b, y1_ptr[0]);
155 RGB_OUT(d1, r, g, b);


---


156

```

```

157 YUV_TO_RGB2(r, g, b, y2_ptr[0]);
158 RGB_OUT(d2, r, g, b);
159 d1 += BPP;
160 d2 += BPP;
161 y1_ptr++;
162 y2_ptr++;
163 cb_ptr++;
164 cr_ptr++;
165 }
166 d += 2 * dst->linesize[0];
167 y1_ptr += 2 * src->linesize[0] - width;
168 cb_ptr += src->linesize[1] - width2;
169 cr_ptr += src->linesize[2] - width2;
170 }
171
172 if (height) /* handle odd height */
173 {
174 d1 = d;
175 for (w = width; w >= 2; w -= 2)
176 {
177 YUV_TO_RGB1(cb_ptr[0], cr_ptr[0]);
178
179 YUV_TO_RGB2(r, g, b, y1_ptr[0]); /* output 2 pixels */
180 RGB_OUT(d1, r, g, b);
181
182 YUV_TO_RGB2(r, g, b, y1_ptr[1]);
183 RGB_OUT(d1 + BPP, r, g, b);
184
185 d1 += 2 * BPP;
186
187 y1_ptr += 2;
188 cb_ptr++;
189 cr_ptr++;
190 }
191
192 if (w) /* handle width */
193 {
194 YUV_TO_RGB1(cb_ptr[0], cr_ptr[0]);
195
196 YUV_TO_RGB2(r, g, b, y1_ptr[0]); /* output 2 pixels */


---


197 RGB_OUT(d1, r, g, b);
198 d1 += BPP;
199
200 y1_ptr++;
201 cb_ptr++;

```

```

202 cr_ptr++;
203 }
204 }
205 }
206
207 static void glue(RGB_NAME, _to_yuv420p)(AVPicture *dst, const AVPicture *src, int width, int height)
208 {
209 int wrap, wrap3, width2;
210 int r, g, b, r1, g1, b1, w;
211 uint8_t *lum, *cb, *cr;
212 const uint8_t *p;
213
214 lum = dst->data[0];
215 cb = dst->data[1];
216 cr = dst->data[2];
217
218 width2 = (width + 1) >> 1;
219 wrap = dst->linesize[0];
220 wrap3 = src->linesize[0];
221 p = src->data[0];
222 for (; height >= 2; height -= 2)
223 {
224 for (w = width; w >= 2; w -= 2)
225 {
226 RGB_IN(r, g, b, p);
227 r1 = r;
228 g1 = g;
229 b1 = b;
230 lum[0] = RGB_TO_Y_CCIR(r, g, b);
231
232 RGB_IN(r, g, b, p + BPP);
233 r1 += r;
234 g1 += g;
235 b1 += b;
236 lum[1] = RGB_TO_Y_CCIR(r, g, b);
237 p += wrap3;


---


238 lum += wrap;
239
240 RGB_IN(r, g, b, p);
241 r1 += r;
242 g1 += g;
243 b1 += b;
244 lum[0] = RGB_TO_Y_CCIR(r, g, b);
245
246 RGB_IN(r, g, b, p + BPP);

```

```

247 r1 += r;
248 g1 += g;
249 b1 += b;
250 lum[1] = RGB_TO_Y_CCIR(r, g, b);
251
252 cb[0] = RGB_TO_U_CCIR(r1, g1, b1, 2);
253 cr[0] = RGB_TO_V_CCIR(r1, g1, b1, 2);
254
255 cb++;
256 cr++;
257 p += - wrap3 + 2 * BPP;
258 lum += - wrap + 2;
259 }
260 if (w)
261 {
262 RGB_IN(r, g, b, p);
263 r1 = r;
264 g1 = g;
265 b1 = b;
266 lum[0] = RGB_TO_Y_CCIR(r, g, b);
267 p += wrap3;
268 lum += wrap;
269 RGB_IN(r, g, b, p);
270 r1 += r;
271 g1 += g;
272 b1 += b;
273 lum[0] = RGB_TO_Y_CCIR(r, g, b);
274 cb[0] = RGB_TO_U_CCIR(r1, g1, b1, 1);
275 cr[0] = RGB_TO_V_CCIR(r1, g1, b1, 1);
276 cb++;
277 cr++;
278 p += - wrap3 + BPP;


---


279 lum += - wrap + 1;
280 }
281 p += wrap3 + (wrap3 - width * BPP);
282 lum += wrap + (wrap - width);
283 cb += dst->linesize[1] - width2;
284 cr += dst->linesize[2] - width2;
285 }
286
287 if (height) /* handle odd height */
288 {
289 for (w = width; w >= 2; w -= 2)
290 {
291 RGB_IN(r, g, b, p);

```

```

292 r1 = r;
293 g1 = g;
294 b1 = b;
295 lum[0] = RGB_TO_Y_CCIR(r, g, b);
296
297 RGB_IN(r, g, b, p + BPP);
298 r1 += r;
299 g1 += g;
300 b1 += b;
301 lum[1] = RGB_TO_Y_CCIR(r, g, b);
302 cb[0] = RGB_TO_U_CCIR(r1, g1, b1, 1);
303 cr[0] = RGB_TO_V_CCIR(r1, g1, b1, 1);
304 cb++;
305 cr++;
306 p += 2 * BPP;
307 lum += 2;
308 }
309 if (w)
310 {
311 RGB_IN(r, g, b, p);
312 lum[0] = RGB_TO_Y_CCIR(r, g, b);
313 cb[0] = RGB_TO_U_CCIR(r, g, b, 0);
314 cr[0] = RGB_TO_V_CCIR(r, g, b, 0);
315 }
316 }
317 }
318
319 static void glue(RGB_NAME, _to_gray)(AVPicture *dst, const AVPicture *src, int width, int height)


---


320 {
321 const unsigned char *p;
322 unsigned char *q;
323 int r, g, b, dst_wrap, src_wrap;
324 int x, y;
325
326 p = src->data[0];
327 src_wrap = src->linesize[0] - BPP * width;
328
329 q = dst->data[0];
330 dst_wrap = dst->linesize[0] - width;
331
332 for (y = 0; y < height; y++)
333 {
334 for (x = 0; x < width; x++)
335 {
336 RGB_IN(r, g, b, p);

```

```

337 q[0] = RGB_TO_Y(r, g, b);
338 q++;
339 p += BPP;
340 }
341 p += src_wrap;
342 q += dst_wrap;
343 }
344 }
345
346 static void glue(gray_to_, RGB_NAME)(AVPicture *dst, const AVPicture *src, int width, int height)
347 {
348 const unsigned char *p;
349 unsigned char *q;
350 int r, dst_wrap, src_wrap;
351 int x, y;
352
353 p = src->data[0];
354 src_wrap = src->linesize[0] - width;
355
356 q = dst->data[0];
357 dst_wrap = dst->linesize[0] - BPP * width;
358
359 for (y = 0; y < height; y++)
360 {


---


361 for (x = 0; x < width; x++)
362 {
363 r = p[0];
364 RGB_OUT(q, r, r, r);
365 q += BPP;
366 p++;
367 }
368 p += src_wrap;
369 q += dst_wrap;
370 }
371 }
372
373 static void glue(pal8_to_, RGB_NAME)(AVPicture *dst, const AVPicture *src, int width, int height)
374 {
375 const unsigned char *p;
376 unsigned char *q;
377 int r, g, b, dst_wrap, src_wrap;
378 int x, y;
379 uint32_t v;
380 const uint32_t *palette;
381

```

```

382 p = src->data[0];
383 src_wrap = src->linesize[0] - width;
384 palette = (uint32_t*)src->data[1];
385
386 q = dst->data[0];
387 dst_wrap = dst->linesize[0] - BPP * width;
388
389 for (y = 0; y < height; y++)
390 {
391 for (x = 0; x < width; x++)
392 {
393 v = palette[p[0]];
394 r = (v >> 16) &0xff;
395 g = (v >> 8) &0xff;
396 b = (v) &0xff;
397 #ifdef RGBA_OUT
398 {
399 int a;
400 a = (v >> 24) &0xff;
401 RGBA_OUT(q, r, g, b, a);
402 }
403 #else
404 RGB_OUT(q, r, g, b);
405 #endif
406 q += BPP;
407 p++;
408 }
409 p += src_wrap;
410 q += dst_wrap;
411 }
412 }
413
414 #if !defined(FMT_RGBA32) && defined(RGBA_OUT)
415 /* alpha support */
416
417 static void glue(rgb32_to_, RGB_NAME)(AVPicture *dst, const AVPicture *src, int width, int height)
418 {
419 const uint8_t *s;
420 uint8_t *d;
421 int src_wrap, dst_wrap, j, y;
422 unsigned int v, r, g, b, a;
423
424 s = src->data[0];
425 src_wrap = src->linesize[0] - width * 4;
426

```

```

427 d = dst->data[0];
428 dst_wrap = dst->linesize[0] - width * BPP;
429
430 for (y = 0; y < height; y++)
431 {
432 for (j = 0; j < width; j++)
433 {
434 v = ((const uint32_t*)(s))[0];
435 a = (v >> 24) &0xff;
436 r = (v >> 16) &0xff;
437 g = (v >> 8) &0xff;
438 b = v &0xff;
439 RGBA_OUT(d, r, g, b, a);
440 s += 4;
441 d += BPP;
442 }
443 s += src_wrap;
444 d += dst_wrap;
445 }
446 }
447
448 static void glue(RGB_NAME, _to_rgba32)(AVPicture *dst, const AVPicture *src, int width, int height)
449 {
450 const uint8_t *s;
451 uint8_t *d;
452 int src_wrap, dst_wrap, j, y;
453 unsigned int r, g, b, a;
454
455 s = src->data[0];
456 src_wrap = src->linesize[0] - width * BPP;
457
458 d = dst->data[0];
459 dst_wrap = dst->linesize[0] - width * 4;
460
461 for (y = 0; y < height; y++)
462 {
463 for (j = 0; j < width; j++)
464 {
465 RGBA_IN(r, g, b, a, s);
466 ((uint32_t*)(d))[0] = (a << 24) | (r << 16) | (g << 8) | b;
467 d += 4;
468 s += BPP;
469 }
470 s += src_wrap;
471 d += dst_wrap;

```

```

472 }
473 }
474
475 #endif /* !defined(FMT_RGBA32) && defined(RGBA_IN) */
476
477 #ifndef FMT_RGB24
478
479 static void glue(rgb24_to_, RGB_NAME)(AVPicture *dst, const AVPicture *src, int width, int height)
480 {
481     const uint8_t *s;
482     uint8_t *d;
483     int src_wrap, dst_wrap, j, y;
484     unsigned int r, g, b;
485
486     s = src->data[0];
487     src_wrap = src->linesize[0] - width * 3;
488
489     d = dst->data[0];
490     dst_wrap = dst->linesize[0] - width * BPP;
491
492     for (y = 0; y < height; y++)
493     {
494         for (j = 0; j < width; j++)
495         {
496             r = s[0];
497             g = s[1];
498             b = s[2];
499             RGB_OUT(d, r, g, b);
500             s += 3;
501             d += BPP;
502         }
503         s += src_wrap;
504         d += dst_wrap;
505     }
506 }
507
508 static void glue(RGB_NAME, _to_rgb24)(AVPicture *dst, const AVPicture *src, int width, int height)
509 {
510     const uint8_t *s;
511     uint8_t *d;
512     int src_wrap, dst_wrap, j, y;
513     unsigned int r, g, b;
514
515     s = src->data[0];
516     src_wrap = src->linesize[0] - width * BPP;

```

517

```

518 d = dst->data[0];
519 dst_wrap = dst->linesize[0] - width * 3;
520
521 for (y = 0; y < height; y++)
522 {
523 for (j = 0; j < width; j++)
524 {
525 RGB_IN(r, g, b, s)d[0] = r;
```

---

```

526         d[1] = g;
527         d[2] = b;
528         d += 3;
529         s += BPP;
530     }
```

531 s += src\_wrap;

532 d += dst\_wrap;

533 }

534 }

535

536 #endif /\* !FMT\_RGB24 \*/

537

538 #ifdef FMT\_RGB24

539

540 static void yuv444p\_to\_rgb24(AVPicture \*dst, const AVPicture \*src, int width, int height)

541 {

```

542 const uint8_t *y1_ptr, *cb_ptr, *cr_ptr;
543 uint8_t *d, *d1;
544 int w, y, cb, cr, r_add, g_add, b_add;
545 uint8_t *cm = cropTbl + MAX_NEG_CROP;
546 unsigned int r, g, b;
```

547

548 d = dst-&gt;data[0];

549 y1\_ptr = src-&gt;data[0];

550 cb\_ptr = src-&gt;data[1];

551 cr\_ptr = src-&gt;data[2];

552 for (; height &gt; 0; height--)

553 {

554 d1 = d;

555 for (w = width; w &gt; 0; w--)

556 {

557 YUV\_TO\_RGB1\_CCIR(cb\_ptr[0], cr\_ptr[0]);

558

559 YUV\_TO\_RGB2\_CCIR(r, g, b, y1\_ptr[0]);

560 RGB\_OUT(d1, r, g, b);

```

561 d1 += BPP;
562
563 y1_ptr++;
564 cb_ptr++;
565 cr_ptr++;


---


566 }
567 d += dst->linesize[0];
568 y1_ptr += src->linesize[0] - width;
569 cb_ptr += src->linesize[1] - width;
570 cr_ptr += src->linesize[2] - width;
571 }
572 }
573
574 static void yuvj444p_to_rgb24(AVPicture *dst, const AVPicture *src, int width, int height)
575 {
576 const uint8_t *y1_ptr, *cb_ptr, *cr_ptr;
577 uint8_t *d, *d1;
578 int w, y, cb, cr, r_add, g_add, b_add;
579 uint8_t *cm = cropTbl + MAX_NEG_CROP;
580 unsigned int r, g, b;
581
582 d = dst->data[0];
583 y1_ptr = src->data[0];
584 cb_ptr = src->data[1];
585 cr_ptr = src->data[2];
586 for (; height > 0; height--)
587 {
588 d1 = d;
589 for (w = width; w > 0; w--)
590 {
591 YUV_TO_RGB1(cb_ptr[0], cr_ptr[0]);
592
593 YUV_TO_RGB2(r, g, b, y1_ptr[0]);
594 RGB_OUT(d1, r, g, b);
595 d1 += BPP;
596
597 y1_ptr++;
598 cb_ptr++;
599 cr_ptr++;
600 }
601 d += dst->linesize[0];
602 y1_ptr += src->linesize[0] - width;
603 cb_ptr += src->linesize[1] - width;
604 cr_ptr += src->linesize[2] - width;
605 }

```

```

606 }


---


607
608 static void rgb24_to_yuv444p(AVPicture *dst, const AVPicture *src, int width, int height)
609 {
610     int src_wrap, x, y;
611     int r, g, b;
612     uint8_t *lum, *cb, *cr;
613     const uint8_t *p;
614
615     lum = dst->data[0];
616     cb = dst->data[1];
617     cr = dst->data[2];
618
619     src_wrap = src->linesize[0] - width * BPP;
620     p = src->data[0];
621
622     for (y = 0; y < height; y++)
623     {
624         for (x = 0; x < width; x++)
625         {
626             RGB_IN(r, g, b, p);
627             lum[0] = RGB_TO_Y_CCIR(r, g, b);
628             cb[0] = RGB_TO_U_CCIR(r, g, b, 0);
629             cr[0] = RGB_TO_V_CCIR(r, g, b, 0);
630             p += BPP;
631             cb++;
632             cr++;
633             lum++;
634         }
635         p += src_wrap;
636         lum += dst->linesize[0] - width;
637         cb += dst->linesize[1] - width;
638         cr += dst->linesize[2] - width;
639     }
640 }
641
642 static void rgb24_to_yuvj420p(AVPicture *dst, const AVPicture *src, int width, int height)
643 {
644     int wrap, wrap3, width2;
645     int r, g, b, r1, g1, b1, w;
646     uint8_t *lum, *cb, *cr;
647     const uint8_t *p;


---


648
649     lum = dst->data[0];
650     cb = dst->data[1];

```

```

651 cr = dst->data[2];
652
653 width2 = (width + 1) >> 1;
654 wrap = dst->linesize[0];
655 wrap3 = src->linesize[0];
656 p = src->data[0];
657 for (; height >= 2; height -= 2)
658 {
659 for (w = width; w >= 2; w -= 2)
660 {
661 RGB_IN(r, g, b, p);
662 r1 = r;
663 g1 = g;
664 b1 = b;
665 lum[0] = RGB_TO_Y(r, g, b);
666
667 RGB_IN(r, g, b, p + BPP);
668 r1 += r;
669 g1 += g;
670 b1 += b;
671 lum[1] = RGB_TO_Y(r, g, b);
672 p += wrap3;
673 lum += wrap;
674
675 RGB_IN(r, g, b, p);
676 r1 += r;
677 g1 += g;
678 b1 += b;
679 lum[0] = RGB_TO_Y(r, g, b);
680
681 RGB_IN(r, g, b, p + BPP);
682 r1 += r;
683 g1 += g;
684 b1 += b;
685 lum[1] = RGB_TO_Y(r, g, b);
686
687 cb[0] = RGB_TO_U(r1, g1, b1, 2);
688 cr[0] = RGB_TO_V(r1, g1, b1, 2);
689

```

---

```

690         cb++;
691         cr++;
692         p += -wrap3 + 2 * BPP;
693         lum += -wrap + 2;
694     }

```

```

695 if (w)
696 {
697 RGB_IN(r, g, b, p);
698 r1 = r;
699 g1 = g;
700 b1 = b;
701 lum[0] = RGB_TO_Y(r, g, b);
702 p += wrap3;
703 lum += wrap;
704 RGB_IN(r, g, b, p);
705 r1 += r;
706 g1 += g;
707 b1 += b;
708 lum[0] = RGB_TO_Y(r, g, b);
709 cb[0] = RGB_TO_U(r1, g1, b1, 1);
710 cr[0] = RGB_TO_V(r1, g1, b1, 1);
711 cb++;
712 cr++;
713 p += -wrap3 + BPP;
714 lum += -wrap + 1;
715 }
716 p += wrap3 + (wrap3 - width * BPP);
717 lum += wrap + (wrap - width);
718 cb += dst->linesize[1] - width2;
719 cr += dst->linesize[2] - width2;
720 }
721
722 if (height) /* handle odd height */
723 {
724 for (w = width; w >= 2; w -= 2)
725 {
726 RGB_IN(r, g, b, p);
727 r1 = r;
728 g1 = g;
729 b1 = b;
730 lum[0] = RGB_TO_Y(r, g, b);


---


731
732 RGB_IN(r, g, b, p + BPP);
733 r1 += r;
734 g1 += g;
735 b1 += b;
736 lum[1] = RGB_TO_Y(r, g, b);
737 cb[0] = RGB_TO_U(r1, g1, b1, 1);
738 cr[0] = RGB_TO_V(r1, g1, b1, 1);
739 cb++;

```

```

740 cr++;
741 p += 2 * BPP;
742 lum += 2;
743 }
744 if (w)
745 {
746 RGB_IN(r, g, b, p);
747 lum[0] = RGB_TO_Y(r, g, b);
748 cb[0] = RGB_TO_U(r, g, b, 0);
749 cr[0] = RGB_TO_V(r, g, b, 0);
750 }
751 }
752 }
753
754 static void rgb24_to_yuvj444p(AVPicture *dst, const AVPicture *src, int width, int height)
755 {
756 int src_wrap, x, y;
757 int r, g, b;
758 uint8_t *lum, *cb, *cr;
759 const uint8_t *p;
760
761 lum = dst->data[0];
762 cb = dst->data[1];
763 cr = dst->data[2];
764
765 src_wrap = src->linesize[0] - width * BPP;
766 p = src->data[0];
767 for (y = 0; y < height; y++)
768 {
769 for (x = 0; x < width; x++)
770 {


---


771 RGB_IN(r, g, b, p);
772 lum[0] = RGB_TO_Y(r, g, b);
773 cb[0] = RGB_TO_U(r, g, b, 0);
774 cr[0] = RGB_TO_V(r, g, b, 0);
775 p += BPP;
776 cb++;
777 cr++;
778 lum++;
779 }
780 p += src_wrap;
781 lum += dst->linesize[0] - width;
782 cb += dst->linesize[1] - width;
783 cr += dst->linesize[2] - width;
784 }

```

```

785 }
786
787 #endif /* FMT_RGB24 */
788
789 #if defined(FMT_RGB24) || defined(FMT_RGBA32)
790
791 static void glue(RGB_NAME, _to_pal8)(AVPicture *dst, const AVPicture *src, int width, int height)
792 {
793     const unsigned char *p;
794     unsigned char *q;
795     int dst_wrap, src_wrap;
796     int x, y, has_alpha;
797     unsigned int r, g, b;
798
799     p = src->data[0];
800     src_wrap = src->linesize[0] - BPP * width;
801
802     q = dst->data[0];
803     dst_wrap = dst->linesize[0] - width;
804     has_alpha = 0;
805
806     for (y = 0; y < height; y++)
807     {
808         for (x = 0; x < width; x++)
809         {
810 #ifdef RGBA_IN
811         {


---


812             unsigned int a;
813             RGBA_IN(r, g, b, a, p);
814
815             if (a < 0x80) /* crude approximation for alpha ! */
816             {
817                 has_alpha = 1;
818                 q[0] = TRANSP_INDEX;
819             }
820             else
821             {
822                 q[0] = gif_clut_index(r, g, b);
823             }
824         }
825 #else
826             RGB_IN(r, g, b, p);
827             q[0] = gif_clut_index(r, g, b);
828 #endif
829         q++;

```

```

830 p += BPP;
831 }
832 p += src_wrap;
833 q += dst_wrap;
834 }
835
836 build_rgb_palette(dst->data[1], has_alpha);
837 }
838
839 #endif /* defined(FMT_RGB24) || defined(FMT_RGBA32) */
840
841 #ifdef RGBA_IN
842
843 #define FF_ALPHA_TRANSP 0x0001 /* image has some totally transparent pixels */
844 #define FF_ALPHA_SEMI_TRANSP 0x0002 /* image has some transparent pixels */
845
846 static int glue(get_alpha_info_, RGB_NAME)(const AVPicture *src, int width, int height)
847 {
848 const unsigned char *p;
849 int src_wrap, ret, x, y;
850 unsigned int r, g, b, a;
851
852 p = src->data[0];


---


853 src_wrap = src->linesize[0] - BPP * width;
854 ret = 0;
855 for (y = 0; y < height; y++)
856 {
857 for (x = 0; x < width; x++)
858 {
859 RGBA_IN(r, g, b, a, p);
860 if (a == 0x00)
861 {
862 ret |= FF_ALPHA_TRANSP;
863 }
864 else if (a != 0xff)
865 {
866 ret |= FF_ALPHA_SEMI_TRANSP;
867 }
868 p += BPP;
869 }
870 p += src_wrap;
871 }
872 return ret;
873 }
874

```

```

875 #endif /* RGBA_IN */
876
877 #undef RGB_IN
878 #undef RGBA_IN
879 #undef RGB_OUT
880 #undef RGBA_OUT
881 #undef BPP
882 #undef RGB_NAME
883 #undef FMT_RGB24


---


884 #undef FMT_RGBA32

```

## 8 imgconvert.c 文件

### 8.1 功能描述

定义并实现图像颜色空间转换使用的函数和宏，此文件大部分请各位自己仔细分析。

### 8.2 文件注释

```

1  #include "avcodec.h"
2  #include "dsputil.h"
3
4  #define xglue(x, y) x ## y
5  #define glue(x, y) xglue(x, y)
6
7  #define FF_COLOR_RGB  0 // RGB color space
8  #define FF_COLOR_GRAY 1 // gray color space
9  #define FF_COLOR_YUV   2 // YUV color space. 16 <= Y <= 235, 16 <= U, V <= 240
10 #define FF_COLOR_YUV_JPEG 3 // YUV color space. 0 <= Y <= 255, 0 <= U, V <= 255
11
12 #define FF_PIXEL_PLANAR    0 // each channel has one component in AVPicture
13 #define FF_PIXEL_PACKED    1 // only one components containing all the channels
14 #define FF_PIXEL_PALETTE   2 // one components containing indexes for a palette
15

```

15  
定义视频图像格式信息类型。

```

16  typedef struct PixFmtInfo
17  {
18      const char *name;
19      uint8_t nb_channels; // number of channels (including alpha)
20      uint8_t color_type; // color type (see FF_COLOR_xxx constants)
21      uint8_t pixel_type; // pixel storage type (see FF_PIXEL_xxx constants)
22      uint8_t is_alpha; // true if alpha can be specified
23      uint8_t x_chroma_shift; // X chroma subsampling factor is 2 ^ shift
24      uint8_t y_chroma_shift; // Y chroma subsampling factor is 2 ^ shift
25      uint8_t depth; // bit depth of the color components
26  } PixFmtInfo;
27

```

定义支持的视频图像格式信息。

```
28 // this table gives more information about formats
29 static PixFmtInfo pix_fmt_info[PIX_FMT_NB] =
30 {
31 { "yuv420p", 3, FF_COLOR_YUV, FF_PIXEL_PLANAR, 0, 1, 1, 8},
32 { "yuv422", 1, FF_COLOR_YUV,
33 { "rgb24", 3, FF_COLOR_RGB,
34 { "bgr24", 3, FF_COLOR_RGB,
35 { "yuv422p", 3, FF_COLOR_YUV,
36 { "yuv444p", 3, FF_COLOR_YUV,
37 { "rgba32", 4, FF_COLOR_RGB,
38 { "yuv410p", 3, FF_COLOR_YUV,
39 { "yuv411p", 3, FF_COLOR_YUV,
40 { "rgb565", 3, FF_COLOR_RGB,
41 { "rgb555", 4, FF_COLOR_RGB,
```

## 《FFmpeg 基础库编程开发》

```
FF_PIXEL_PACKED, 0, 1, 0, 8}, FF_PIXEL_PACKED, 0, 0, 0, 8}, FF_PIXEL_PACKED, 0, 0, 0, 8},
FF_PIXEL_PLANAR, 0, 1, 0, 8}, FF_PIXEL_PLANAR, 0, 0, 0, 8}, FF_PIXEL_PACKED, 1, 0, 0, 8},
FF_PIXEL_PLANAR, 0, 2, 2, 8}, FF_PIXEL_PLANAR, 0, 2, 0, 8}, FF_PIXEL_PACKED, 0, 0, 0, 5},
FF_PIXEL_PACKED, 1, 0, 0, 5},
42 { "gray", 1, FF_COLOR_GRAY, FF_PIXEL_PLANAR, 0, 0, 0, 8},
43 { "monow", 1, FF_COLOR_GRAY, FF_PIXEL_PLANAR, 0, 0, 0, 1},
44 { "monob", 1, FF_COLOR_GRAY, FF_PIXEL_PLANAR, 0, 0, 0, 1},
45 { "pal8", 4, FF_COLOR_RGB, FF_PIXEL_PALETTE, 1, 0, 0, 8},
46 { "yuvj420p", 3, FF_COLOR_YUV_JPEG, FF_PIXEL_PLANAR, 0, 1, 1, 8},
47 { "yuvj422p", 3, FF_COLOR_YUV_JPEG, FF_PIXEL_PLANAR, 0, 1, 0, 8},
48 { "yuvj444p", 3, FF_COLOR_YUV_JPEG, FF_PIXEL_PLANAR, 0, 0, 0, 8},
49 { "xvmcmc", },
50 { "xvmcidct", },
51 { "uyvy422", 1, FF_COLOR_YUV, FF_PIXEL_PACKED, 0, 1, 0, 8},
52 { "uyvy411", 1, FF_COLOR_YUV, FF_PIXEL_PACKED, 0, 2, 0, 8},
53 };
54 }
```

读取视频图像格式信息中色度相对亮度采样比例(用移位的位数表示)。

```
55 void avcodec_get_chroma_sub_sample(int pix_fmt, int *h_shift, int *v_shift)
56 {
57 *h_shift = pix_fmt_info[pix_fmt].x_chroma_shift;
58 *v_shift = pix_fmt_info[pix_fmt].y_chroma_shift;
59 }
60 }
```

填充各种视频图像格式对应的 AVPicture 结构字段，返回图像大小。

```
61 // Picture field are filled with 'ptr' addresses. Also return size
62 int avpicture_fill(APicture *picture, uint8_t *ptr, int pix_fmt, int width, int height)
63 {
64 int size, w2, h2, size2;
65 PixFmtInfo *pinfo;
```

图像像素大小规整，比如 YUV420P 宽度和高度必须是 2 的整数倍，如果不符，程序自动填充补足。

```

67 if (avcodec_check_dimensions(NULL, width, height))
68 goto fail;
69
70 pinfo = &pix_fmt_info[pix_fmt];
71 size = width * height;
72 switch (pix_fmt)
73 {
按照图像格式，分别计算 AVPicture 结构字段的值。
74 case PIX_FMT_YUV420P:
75 case PIX_FMT_YUV422P:
76 case PIX_FMT_YUV444P:
77 case PIX_FMT_YUV410P:
78 case PIX_FMT_YUV411P:
79 case PIX_FMT_YUVJ420P:
80 case PIX_FMT_YUVJ422P:
81 case PIX_FMT_YUVJ444P:
82 w2 = (width + (1 << pinfo->x_chroma_shift) - 1) >> pinfo->x_chroma_shift;
83 h2 = (height + (1 << pinfo->y_chroma_shift) - 1) >> pinfo->y_chroma_shift;
84 size2 = w2 * h2;
85 picture->data[0] = ptr;
86 picture->data[1] = picture->data[0] + size;
87 picture->data[2] = picture->data[1] + size2;
88 picture->linesize[0] = width;
89 picture->linesize[1] = w2;
90 picture->linesize[2] = w2;
91 return size + 2 * size2;
92 case PIX_FMT_RGB24:
93 case PIX_FMT_BGR24:
94 picture->data[0] = ptr;
95 picture->data[1] = NULL;
96 picture->data[2] = NULL;
97 picture->linesize[0] = width * 3;
98 return size * 3;
99 case PIX_FMT_RGBA32:
100 picture->data[0] = ptr;
101 picture->data[1] = NULL;
102 picture->data[2] = NULL;
103 picture->linesize[0] = width * 4;
104 return size * 4;
105 case PIX_FMT_RGB555:

```

```
106 case PIX_FMT_RGB565:  
107 case PIX_FMT_YUV422:  
108 picture->data[0] = ptr;  
109 picture->data[1] = NULL;  
110 picture->data[2] = NULL;  
111 picture->linesize[0] = width * 2;  
112 return size *2;  
113 case PIX_FMT_UYVY422:  
114 picture->data[0] = ptr;  
115 picture->data[1] = NULL;  
116 picture->data[2] = NULL;  
117 picture->linesize[0] = width * 2;  
118 return size *2;  
119 case PIX_FMT_UYVY411:  
120 picture->data[0] = ptr;  
121 picture->data[1] = NULL;  
122 picture->data[2] = NULL;  
123 picture->linesize[0] = width + width / 2;  
124 return size + size / 2;  
125 case PIX_FMT_GRAY8:  
126 picture->data[0] = ptr;  
127 picture->data[1] = NULL;  
128 picture->data[2] = NULL;  
129 picture->linesize[0] = width;  
130 return size;  
131 case PIX_FMT_MONOWHITE:  
132 case PIX_FMT_MONOBLACK:  
133 picture->data[0] = ptr;  
134 picture->data[1] = NULL;  
135 picture->data[2] = NULL;  
136 picture->linesize[0] = (width + 7) >> 3;  
137 return picture->linesize[0] *height;  
138 case PIX_FMT_PAL8:  
139 size2 = (size + 3) &~3;  
140 picture->data[0] = ptr;  
141 picture->data[1] = ptr + size2; // palette is stored here as 256 32 bit words  
142 picture->data[2] = NULL;  
143 picture->linesize[0] = width;  
144 picture->linesize[1] = 4;  
145 return size2 + 256 * 4;  
146 default:
```

```

147 fail:
148 picture->data[0] = NULL;
149 picture->data[1] = NULL;
150 picture->data[2] = NULL;
151 picture->data[3] = NULL;
152 return -1;
153 }
154 }
```

155

传入像素格式，图像长宽，计算图像大小。程序简单的复用 avpicture\_fill()函数的返回值。

```

156 int avpicture_get_size(int pix_fmt, int width, int height)
157 {
158 AVPicture dummy_pict;
159 return avpicture_fill(&dummy_pict, NULL, pix_fmt, width, height);
160 }
```

161

初始化 AVPicture 结构。输入像素格式和长宽，计算图像大小，分配图像缓存，填充 AVPicture 结构。

```

162 int avpicture_alloc(APicture *picture, int pix_fmt, int width, int height)
163 {
```

164 unsigned int size;

165 void \*ptr;

166

调用函数计算图像大小。

```
167 size = avpicture_get_size(pix_fmt, width, height);
```

168 if (size < 0)

169 goto fail;

调用函数分配图像缓存。

```
170 ptr = av_malloc(size);
```

171 if (!ptr)

172 goto fail;

填充 AVPicture 结构。

```
173 avpicture_fill(picture, ptr, pix_fmt, width, height);
```

174 return 0;

175 fail:

```
176 memset(picture, 0, sizeof(APicture));
```

177 return -1;

178 }

179

释放 AVPicture 分配的内存，因为内存首地址在 picture->data[0]中，所以可以简单的释放。

```
180 void avpicture_free(APicture *picture)
```

181 {

```
182 av_free(picture->data[0]);
```

183 }

184

计算各种图像格式平均每个像素占用的 bit 位数。

```

185 static int avg_bits_per_pixel(int pix_fmt)
186 {
187     int bits;
188     const PixFmtInfo *pf;
189
190     pf = &pix_fmt_info[pix_fmt];
191     switch (pf->pixel_type)
192     {
193         case FF_PIXEL_PACKED:
194             switch (pix_fmt)
195             {
196                 case PIX_FMT_YUV422:
197                 case PIX_FMT_UYVY422:
198                 case PIX_FMT_RGB565:
199                 case PIX_FMT_RGB555:
200                     bits = 16;
201                     break;
202                 case PIX_FMT_UYVY411:
203                     bits = 12;
204                     break;
205                 default:
206                     bits = pf->depth * pf->nb_channels;
207                     break;
208             }
209             break;
210         case FF_PIXEL_PLANAR:
211             if (pf->x_chroma_shift == 0 && pf->y_chroma_shift == 0)
212             {
213                 bits = pf->depth * pf->nb_channels;
214             }
215             else
216             {
217                 bits = pf->depth + ((2 * pf->depth) >> (pf->x_chroma_shift + pf->y_chroma_shift));
218             }
219             break;
220         case FF_PIXEL_PALETTE:
221             bits = 8;
222             break;
223         default:
224             bits = - 1;
225             break;

```

```

226 }
227 return bits;
228 }
229
230 /////////////////
231
231 图像数据平面拷贝，由于宽度可能有差别，只能一行一行的拷贝。
232 void ff_img_copy_plane(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
233 {
234 if ((!dst) || (!src))
235 return ;
236 for (; height > 0; height--)
237 {
238 memcpy(dst, src, width);
239 dst += dst_wrap;
240 src += src_wrap;
241 }
242 }
243
243 各种图像格式的图像数据拷贝。
244 void img_copy(AVPicture *dst, const AVPicture *src, int pix_fmt, int width, int height)
245 {
246 int bwidth, bits, i;
247 PixFmtInfo *pf = &pix_fmt_info[pix_fmt];
248
249 pf = &pix_fmt_info[pix_fmt];
250 switch (pf->pixel_type)
251 {
252 case FF_PIXEL_PACKED:
253 switch (pix_fmt)
254 {
255 case PIX_FMT_YUV422:
256 case PIX_FMT_UYVY422:
257 case PIX_FMT_RGB565:
258 case PIX_FMT_RGB555:
259 bits = 16;
260 break;
261 case PIX_FMT_UYVY411:
262 bits = 12;
263 break;
264 default:
265 bits = pf->depth * pf->nb_channels;
266 break;

```

```

267 }
268 bwidth = (width *bits + 7) >> 3;
269 ff_img_copy_plane(dst->data[0], dst->linesize[0], src->data[0], src->linesize[0], bwidth, height);
270 break;
271 case FF_PIXEL_PLANAR:
272 for (i = 0; i < pf->nb_channels; i++)
273 {
274 int w, h;
275 w = width;
276 h = height;
277 if (i == 1 || i == 2)
278 {
279 w >>= pf->x_chroma_shift;
280 h >>= pf->y_chroma_shift;
281 }
282 bwidth = (w *pf->depth + 7) >> 3;
283 ff_img_copy_plane(dst->data[i], dst->linesize[i], src->data[i], src->linesize[i], bwidth, h);
284 }
285 break;
286 case FF_PIXEL_PALETTE:
287 ff_img_copy_plane(dst->data[0], dst->linesize[0], src->data[0], src->linesize[0], width, height);
288 // copy the palette
289 ff_img_copy_plane(dst->data[1], dst->linesize[1], src->data[1], src->linesize[1], 4,
256);
290 break;
291 }
292 }
293

```

本文件的后面部分请各位自行仔细分析。

```

294 static void yuv422_to_yuv420p(AVPicture *dst, const AVPicture *src, int width, int height)
295 {
296 const uint8_t *p, *p1;
297 uint8_t *lum, *cr, *cb, *lum1, *cr1, *cb1;
298 int w;
299
300 p1 = src->data[0];
301 lum1 = dst->data[0];
302 cb1 = dst->data[1];
303 cr1 = dst->data[2];
304
305 for (; height >= 1; height -= 2)

```

```
306 {
307 p = p1;
308 lum = lum1;
309 cb = cb1;
310 cr = cr1;
311 for (w = width; w >= 2; w -= 2)
312 {
313 lum[0] = p[0];
314 cb[0] = p[1];
315 lum[1] = p[2];
316 cr[0] = p[3];
317 p += 4;
318 lum += 2;
319 cb++;
320 cr++;
321 }
322 if (w)
323 {
324 lum[0] = p[0];
325 cb[0] = p[1];
326 cr[0] = p[3];
327 cb++;


---


328 cr++;
329 }
330 p1 += src->linesize[0];
331 lum1 += dst->linesize[0];
332 if (height > 1)
333 {
334 p = p1;
335 lum = lum1;
336 for (w = width; w >= 2; w -= 2)
337 {
338 lum[0] = p[0];
339 lum[1] = p[2];
340 p += 4;
341 lum += 2;
342 }
343 if (w)
344 {
345 lum[0] = p[0];
346 }
347 p1 += src->linesize[0];
348 lum1 += dst->linesize[0];
```

```

349 }
350 cb1 += dst->linesize[1];
351 cr1 += dst->linesize[2];
352 }
353 }
354
355 static void uyvy422_to_yuv420p(AVPicture *dst, const AVPicture *src, int width, int height)
356 {
357 const uint8_t *p,    *p1;
358 uint8_t *lum,   *cr,   *cb, *lum1,   *cr1,     *cb1;
359 int w;
360
361 p1 = src->data[0];
362
363 lum1 = dst->data[0];
364 cb1 = dst->data[1];
365 cr1 = dst->data[2];
366
367 for (; height >= 1; height -= 2)
368 {


---


369 p = p1;
370 lum = lum1;
371 cb = cb1;
372 cr = cr1;
373 for (w = width; w >= 2; w -= 2)
374 {
375 lum[0] = p[1];
376 cb[0] = p[0];
377 lum[1] = p[3];
378 cr[0] = p[2];
379 p += 4;
380 lum += 2;
381 cb++;
382 cr++;
383 }
384 if (w)
385 {
386 lum[0] = p[1];
387 cb[0] = p[0];
388 cr[0] = p[2];
389 cb++;
390 cr++;
391 }

```

```

392 p1 += src->linesize[0];
393 lum1 += dst->linesize[0];
394 if (height > 1)
395 {
396 p = p1;
397 lum = lum1;
398 for (w = width; w >= 2; w -= 2)
399 {
400 lum[0] = p[1];
401 lum[1] = p[3];
402 p += 4;
403 lum += 2;
404 }
405 if (w)
406 {
407 lum[0] = p[1];
408 }
409 p1 += src->linesize[0];


---


410 lum1 += dst->linesize[0];
411 }
412 cb1 += dst->linesize[1];
413 cr1 += dst->linesize[2];
414 }
415 }
416
417 static void uyvy422_to_yuv422p(AVPicture *dst, const AVPicture *src, int width, int height)
418 {
419 const uint8_t *p,    *p1;
420 uint8_t *lum,   *cr,   *cb, *lum1,   *cr1,     *cb1;
421 int w;
422
423 p1 = src->data[0];
424 lum1 = dst->data[0];
425 cb1 = dst->data[1];
426 cr1 = dst->data[2];
427 for (; height > 0; height--)
428 {
429 p = p1;
430 lum = lum1;
431 cb = cb1;
432 cr = cr1;
433 for (w = width; w >= 2; w -= 2)
434 {

```

```

435 lum[0] = p[1];
436 cb[0] = p[0];
437 lum[1] = p[3];
438 cr[0] = p[2];
439 p += 4;
440 lum += 2;
441 cb++;
442 cr++;
443 }
444 p1 += src->linesize[0];
445 lum1 += dst->linesize[0];
446 cb1 += dst->linesize[1];
447 cr1 += dst->linesize[2];
448 }
449 }
450


---


451 static void yuv422_to_yuv422p(AVPicture *dst, const AVPicture *src, int width, int height)
452 {
453 const uint8_t *p,    *p1;
454 uint8_t *lum,   *cr,   *cb, *lum1,   *cr1,     *cb1;
455 int w;
456
457 p1 = src->data[0];
458 lum1 = dst->data[0];
459 cb1 = dst->data[1];
460 cr1 = dst->data[2];
461 for (; height > 0; height--)
462 {
463 p = p1;
464 lum = lum1;
465 cb = cb1;
466 cr = cr1;
467 for (w = width; w >= 2; w -= 2)
468 {
469 lum[0] = p[0];
470 cb[0] = p[1];
471 lum[1] = p[2];
472 cr[0] = p[3];
473 p += 4;
474 lum += 2;
475 cb++;
476 cr++;
477 }

```

```

478 p1 += src->linesize[0];
479 lum1 += dst->linesize[0];
480 cb1 += dst->linesize[1];
481 cr1 += dst->linesize[2];
482 }
483 }
484
485 static void yuv422p_to_yuv422(APicture *dst, const APicture *src, int width, int height)
486 {
487     uint8_t *p,    *p1;
488     const uint8_t *lum, *cr, *cb, *lum1,   *cr1,
489             *cb1;
490     int w;
491     p1 = dst->data[0];
492     lum1 = src->data[0];


---


493     cb1 = src->data[1];
494     cr1 = src->data[2];
495     for (; height > 0; height--)
496     {
497         p = p1;
498         lum = lum1;
499         cb = cb1;
500         cr = cr1;
501         for (w = width; w >= 2; w -= 2)
502         {
503             p[0] = lum[0];
504             p[1] = cb[0];
505             p[2] = lum[1];
506             p[3] = cr[0];
507             p += 4;
508             lum += 2;
509             cb++;
510             cr++;
511         }
512         p1 += dst->linesize[0];
513         lum1 += src->linesize[0];
514         cb1 += src->linesize[1];
515         cr1 += src->linesize[2];
516     }
517 }
518

```

## 《FFmpeg 基础库编程开发》

```
519 static void yuv422p_to_uyvy422(AVPicture *dst, const AVPicture *src, int width, int height)
520 {
521     uint8_t *p,    *p1;
522     const uint8_t *lum, *cr, *cb, *lum1,   *cr1,      *cb1;
523     int w;
524
525     p1 = dst->data[0];
526     lum1 = src->data[0];
527     cb1 = src->data[1];
528     cr1 = src->data[2];
529     for (; height > 0; height--)
530     {
531         p = p1;
532         lum = lum1;


---


533         cb = cb1;
534         cr = cr1;
535         for (w = width; w >= 2; w -= 2)
536         {
537             p[1] = lum[0];
538             p[0] = cb[0];
539             p[3] = lum[1];
540             p[2] = cr[0];
541             p += 4;
542             lum += 2;
543             cb++;
544             cr++;
545         }
546         p1 += dst->linesize[0];
547         lum1 += src->linesize[0];
548         cb1 += src->linesize[1];
549         cr1 += src->linesize[2];
550     }
551 }
552
553 static void uyvy411_to_yuv411p(AVPicture *dst, const AVPicture *src, int width, int height)
554 {
555     const uint8_t *p,    *p1;
556     uint8_t *lum, *cr, *cb, *lum1,   *cr1,      *cb1;
557     int w;
558
559     p1 = src->data[0];
560     lum1 = dst->data[0];
561     cb1 = dst->data[1];
```

```

562 cr1 = dst->data[2];
563 for (; height > 0; height--)
564 {
565 p = p1;
566 lum = lum1;
567 cb = cb1;
568 cr = cr1;
569 for (w = width; w >= 4; w -= 4)
570 {
571 cb[0] = p[0];
572 lum[0] = p[1];
573 lum[1] = p[2];


---


574 cr[0] = p[3];
575 lum[2] = p[4];
576 lum[3] = p[5];
577 p += 6;
578 lum += 4;
579 cb++;
580 cr++;
581 }
582 p1 += src->linesize[0];
583 lum1 += dst->linesize[0];
584 cb1 += dst->linesize[1];
585 cr1 += dst->linesize[2];
586 }
587 }
588
589 static void yuv420p_to_yuv422(APVPicture *dst, const APVPicture *src, int width, int height)
590 {
591 int w, h;
592 uint8_t *line1, *line2,    *linesrc = dst->data[0];
593 uint8_t *lum1, *lum2,    *lumsrc = src->data[0];
594 uint8_t *cb1,   *cb2 = src->data[1];
595 uint8_t *cr1,   *cr2 = src->data[2];
596
597 for (h = height / 2; h--;}
598 {
599 line1 = linesrc;
600 line2 = linesrc + dst->linesize[0];
601
602 lum1 = lumsrc;
603 lum2 = lumsrc + src->linesize[0];
604

```

```

605 cb1 = cb2;
606 cr1 = cr2;
607
608 for (w = width / 2; w--;) {
609 {
610 *line1++ = *lum1++;
611 *line2++ = *lum2++;
612 *line1++ = *line2++ = *cb1++;
613 *line1++ = *lum1++;
614 *line2++ = *lum2++;


---


615 *line1++ = *line2++ = *cr1++;
616 }
617
618 linesrc += dst->linesize[0] *2;
619 lumsrc += src->linesize[0] *2;
620 cb2 += src->linesize[1];
621 cr2 += src->linesize[2];
622 }
623 }
624
625 static void yuv420p_to_uvy422(AVPicture *dst, const AVPicture *src, int width, int height)
626 {
627 int w, h;
628 uint8_t *line1, *line2, *linesrc = dst->data[0];
629 uint8_t *lum1, *lum2, *lumsrc = src->data[0];
630 uint8_t *cb1, *cb2 = src->data[1];
631 uint8_t *cr1, *cr2 = src->data[2];
632
633 for (h = height / 2; h--;) {
634 {
635 line1 = linesrc;
636 line2 = linesrc + dst->linesize[0];
637
638 lum1 = lumsrc;
639 lum2 = lumsrc + src->linesize[0];
640
641 cb1 = cb2;
642 cr1 = cr2;
643
644 for (w = width / 2; w--;) {
645 {
646 *line1++ = *line2++ = *cb1++;
647 *line1++ = *lum1++;

```

```

648 *line2++ = *lum2++;
649 *line1++ = *line2++ = *cr1++;
650 *line1++ = *lum1++;
651 *line2++ = *lum2++;
652 }
653
654 linesrc += dst->linesize[0] *2;
655 lumsrc += src->linesize[0] *2;


---


656 cb2 += src->linesize[1];
657 cr2 += src->linesize[2];
658 }
659 }
660
661 #define SCALEBITS 10
662 #define ONE_HALF (1 << (SCALEBITS - 1))
663 #define FIX(x) ((int) ((x) * (1<<SCALEBITS) + 0.5))
664
665 #define YUV_TO_RGB1_CCIR(cb1, cr1) \
666 { \
667 cb = (cb1) - 128; \
668 cr = (cr1) - 128; \
669 r_add = FIX(1.40200*255.0/224.0) * cr + ONE_HALF; \
670 g_add = - FIX(0.34414*255.0/224.0) * cb - FIX(0.71414*255.0/224.0) * cr + \
671 ONE_HALF; \
672 b_add = FIX(1.77200*255.0/224.0) * cb + ONE_HALF; \
673 }
674
675 #define YUV_TO_RGB2_CCIR(r, g, b, y1) \
676 { \
677 y = ((y1) - 16) * FIX(255.0/219.0); \
678 r = cm[(y + r_add) >> SCALEBITS]; \
679 g = cm[(y + g_add) >> SCALEBITS]; \
680 b = cm[(y + b_add) >> SCALEBITS]; \
681 }
682
683 #define YUV_TO_RGB1(cb1, cr1) \
684 { \
685 cb = (cb1) - 128; \
686 cr = (cr1) - 128; \
687 r_add = FIX(1.40200) * cr + ONE_HALF; \
688 g_add = - FIX(0.34414) * cb - FIX(0.71414) * cr + ONE_HALF; \
689 b_add = FIX(1.77200) * cb + ONE_HALF; \
690 }

```

```

691
692 #define YUV_TO_RGB2(r, g, b, y1) \
693 { \
694     y = (y1) << SCALEBITS; \
695     r = cm[(y + r_add) >> SCALEBITS]; \
696     g = cm[(y + g_add) >> SCALEBITS]; \
697     b = cm[(y + b_add) >> SCALEBITS]; \
698 }
699
700 #define Y_CCIR_TO_JPEG(y) \
701     cm[((y) * FIX(255.0/219.0) + (ONE_HALF - 16 * FIX(255.0/219.0))) >> SCALEBITS]
702
703 #define Y_JPEG_TO_CCIR(y) \
704     (((y) * FIX(219.0/255.0) + (ONE_HALF + (16 << SCALEBITS))) >> SCALEBITS)
705
706 #define C_CCIR_TO_JPEG(y) \
707     cm[((((y) - 128) * FIX(127.0/112.0) + (ONE_HALF + (128 << SCALEBITS))) >> SCALEBITS]
708
709 /* NOTE: the clamp is really necessary! */
710 static inline int C_JPEG_TO_CCIR(int y)
711 {
712     y = (((y - 128) * FIX(112.0 / 127.0) + (ONE_HALF + (128 << SCALEBITS))) >> SCALEBITS);
713     if (y < 16)
714         y = 16;
715     return y;
716 }
717
718 #define RGB_TO_Y(r, g, b) \
719     ((FIX(0.29900) * (r) + FIX(0.58700) * (g) + \
720     FIX(0.11400) * (b) + ONE_HALF) >> SCALEBITS)
721
722 #define RGB_TO_U(r1, g1, b1, shift) \
723     (((- FIX(0.16874) * r1 - FIX(0.33126) * g1 +      \
724     FIX(0.50000) * b1 + (ONE_HALF << shift) - 1) >> (SCALEBITS + shift)) + 128)
725
726 #define RGB_TO_V(r1, g1, b1, shift) \
727     (((FIX(0.50000) * r1 - FIX(0.41869) * g1 -      \
728     FIX(0.08131) * b1 + (ONE_HALF << shift) - 1) >> (SCALEBITS + shift)) + 128)
729
730 #define RGB_TO_Y_CCIR(r, g, b) \
731     ((FIX(0.29900*219.0/255.0) * (r) + FIX(0.58700*219.0/255.0) * (g) + \
732     FIX(0.11400*219.0/255.0) * (b) + (ONE_HALF + (16 << SCALEBITS))) >> SCALEBITS)
733

```

## 《FFmpeg 基础库编程开发》

```
734 #define RGB_TO_U_CCIR(r1, g1, b1, shift) \
735 (((- FIX(0.16874*224.0/255.0) * r1 - FIX(0.33126*224.0/255.0) * g1 +  \
736 FIX(0.50000*224.0/255.0) * b1 + (ONE_HALF << shift) - 1) >> (SCALEBITS + shift)) + 128)
737


---


738 #define RGB_TO_V_CCIR(r1, g1, b1, shift) \
739 (((FIX(0.50000*224.0/255.0) * r1 - FIX(0.41869*224.0/255.0) * g1 - \
740 FIX(0.08131*224.0/255.0) * b1 + (ONE_HALF << shift) - 1) >> (SCALEBITS + shift)) + 128)
741
742 static uint8_t y_ccir_to_jpeg[256];
743 static uint8_t y_jpeg_to_ccir[256];
744 static uint8_t c_ccir_to_jpeg[256];
745 static uint8_t c_jpeg_to_ccir[256];
746
747 /* apply to each pixel the given table */
748 static void img_apply_table(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap,
749 int width, int height, const uint8_t *table1)
750 {
751 int n;
752 const uint8_t *s;
753 uint8_t *d;
754 const uint8_t *table;
755
756 table = table1;
757 for (; height > 0; height--)
758 {
759 s = src;
760 d = dst;
761 n = width;
762 while (n >= 4)
763 {
764 d[0] = table[s[0]];
765 d[1] = table[s[1]];
766 d[2] = table[s[2]];
767 d[3] = table[s[3]];
768 d += 4;
769 s += 4;
770 n -= 4;
771 }
772 while (n > 0)
773 {
774 d[0] = table[s[0]];
775 d++;
776 s++;
777 }
```

```

777 n--;
778 }


---


779 dst += dst_wrap;
780 src += src_wrap;
781 }
782 }
783
784 /* XXX: use generic filter ? */
785 /* XXX: in most cases, the sampling position is incorrect */
786
787 /* 4x1 -> 1x1 */
788 static void shrink41(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
789 {
790 int w;
791 const uint8_t *s;
792 uint8_t *d;
793
794 for (; height > 0; height--)
795 {
796 s = src;
797 d = dst;
798 for (w = width; w > 0; w--)
799 {
800 d[0] = (s[0] + s[1] + s[2] + s[3] + 2) >> 2;
801 s += 4;
802 d++;
803 }
804 src += src_wrap;
805 dst += dst_wrap;
806 }
807 }
808
809 /* 2x1 -> 1x1 */
810 static void shrink21(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
811 {
812 int w;
813 const uint8_t *s;
814 uint8_t *d;
815
816 for (; height > 0; height--)
817 {
818 s = src;
819 d = dst;

```

```
820 for (w = width; w > 0; w--)  
821 {  
822 d[0] = (s[0] + s[1]) >> 1;  
823 s += 2;  
824 d++;  
825 }  
826 src += src_wrap;  
827 dst += dst_wrap;  
828 }  
829 }  
830  
831 /* 1x2 -> 1x1 */  
832 static void shrink12(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)  
833 {  
834 int w;  
835 uint8_t *d;  
836 const uint8_t *s1, *s2;  
837  
838 for (; height > 0; height--)  
839 {  
840 s1 = src;  
841 s2 = s1 + src_wrap;  
842 d = dst;  
843 for (w = width; w >= 4; w -= 4)  
844 {  
845 d[0] = (s1[0] + s2[0]) >> 1;  
846 d[1] = (s1[1] + s2[1]) >> 1;  
847 d[2] = (s1[2] + s2[2]) >> 1;  
848 d[3] = (s1[3] + s2[3]) >> 1;  
849 s1 += 4;  
850 s2 += 4;  
851 d += 4;  
852 }  
853 for (; w > 0; w--)  
854 {  
855 d[0] = (s1[0] + s2[0]) >> 1;  
856 s1++;  
857 s2++;  
858 d++;  
859 }  
860 src += 2 * src_wrap;  
861 dst += dst_wrap;  
862 }
```

```

863 }
864
865 /* 2x2 -> 1x1 */
866 void ff_shrink22(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
867 {
868     int w;
869     const uint8_t *s1, *s2;
870     uint8_t *d;
871
872     for (; height > 0; height--)
873     {
874         s1 = src;
875         s2 = s1 + src_wrap;
876         d = dst;
877         for (w = width; w >= 4; w -= 4)
878         {
879             d[0] = (s1[0] + s1[1] + s2[0] + s2[1] + 2) >> 2;
880             d[1] = (s1[2] + s1[3] + s2[2] + s2[3] + 2) >> 2;
881             d[2] = (s1[4] + s1[5] + s2[4] + s2[5] + 2) >> 2;
882             d[3] = (s1[6] + s1[7] + s2[6] + s2[7] + 2) >> 2;
883             s1 += 8;
884             s2 += 8;
885             d += 4;
886         }
887         for (; w > 0; w--)
888         {
889             d[0] = (s1[0] + s1[1] + s2[0] + s2[1] + 2) >> 2;
890             s1 += 2;
891             s2 += 2;
892             d++;
893         }
894         src += 2 * src_wrap;
895         dst += dst_wrap;
896     }
897 }
898
899 /* 4x4 -> 1x1 */
900 void ff_shrink44(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
901 {
902     int w;
903     const uint8_t *s1, *s2, *s3, *s4;
904     uint8_t *d;
905

```

---

```

906 for (; height > 0; height--)
907 {
908 s1 = src;
909 s2 = s1 + src_wrap;
910 s3 = s2 + src_wrap;
911 s4 = s3 + src_wrap;
912 d = dst;
913 for (w = width; w > 0; w--)
914 {
915 d[0] = (s1[0] + s1[1] + s1[2] + s1[3] + s2[0] + s2[1] + s2[2] + s2[3] +
916 s3[0] + s3[1] + s3[2] + s3[3] + s4[0] + s4[1] + s4[2] + s4[3] + 8) >> 4;
917 s1 += 4;
918 s2 += 4;
919 s3 += 4;
920 s4 += 4;
921 d++;
922 }
923 src += 4 * src_wrap;
924 dst += dst_wrap;
925 }
926 }
927
928 static void grow21_line(uint8_t *dst, const uint8_t *src, int width)
929 {
930 int w;
931 const uint8_t *s1;
932 uint8_t *d;
933
934 s1 = src;
935 d = dst;
936 for (w = width; w >= 4; w -= 4)
937 {
938 d[1] = d[0] = s1[0];
939 d[3] = d[2] = s1[1];
940 s1 += 2;
941 d += 4;
942 }


---


943 for (; w >= 2; w -= 2)
944 {
945 d[1] = d[0] = s1[0];
946 s1++;
947 d += 2;
948 }

```

## 《FFmpeg 基础库编程开发》

```
949 /* only needed if width is not a multiple of two */
950 /* XXX: veryfy that */
951 if (w)
952 {
953 d[0] = s1[0];
954 }
955 }
956
957 static void grow41_line(uint8_t *dst, const uint8_t *src, int width)
958 {
959 int w, v;
960 const uint8_t *s1;
961 uint8_t *d;
962
963 s1 = src;
964 d = dst;
965 for (w = width; w >= 4; w -= 4)
966 {
967 v = s1[0];
968 d[0] = v;
969 d[1] = v;
970 d[2] = v;
971 d[3] = v;
972 s1++;
973 d += 4;
974 }
975 }
976
977 /* 1x1 -> 2x1 */
978 static void grow21(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
979 {
980 for (; height > 0; height--)
981 {
982 grow21_line(dst, src, width);
983 src += src_wrap;
984 dst += dst_wrap;
985 }
986 }
987
988 /* 1x1 -> 2x2 */
989 static void grow22(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
990 {
991 for (; height > 0; height--)
```

```

992 {
993 grow21_line(dst, src, width);
994 if (height % 2)
995 src += src_wrap;
996 dst += dst_wrap;
997 }
998 }
999

1000 /* 1x1 -> 4x1 */
1001 static void grow41(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
1002 {
1003     for (; height > 0; height--)
1004     {
1005         grow41_line(dst, src, width);
1006         src += src_wrap;
1007         dst += dst_wrap;
1008     }
1009 }
1010

1011 /* 1x1 -> 4x4 */
1012 static void grow44(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
1013 {
1014     for (; height > 0; height--)
1015     {
1016         grow41_line(dst, src, width);
1017         if ((height &3) == 1)
1018             src += src_wrap;
1019             dst += dst_wrap;
1020     }
1021 }
1022

1023 /* 1x2 -> 2x1 */
1024 static void conv411(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
1025 {
1026     int w, c;
1027     const uint8_t *s1, *s2;
1028     uint8_t *d;
1029
1030     width >>= 1;
1031
1032     for (; height > 0; height--)
1033     {
1034         s1 = src;

```

---

```

1035     s2 = src + src_wrap;
1036     d = dst;
1037     for (w = width; w > 0; w--)
1038     {
1039         c = (s1[0] + s2[0]) >> 1;
1040         d[0] = c;
1041         d[1] = c;
1042         s1++;
1043         s2++;
1044         d += 2;
1045     }
1046     src += src_wrap * 2;
1047     dst += dst_wrap;
1048 }
1049 }
1050
1051 /* XXX: add jpeg quantize code */
1052
1053 #define TRANSP_INDEX (6*6*6)
1054
1055 /* this is maybe slow, but allows for extensions */
1056 static inline unsigned char gif_clut_index(uint8_t r, uint8_t g, uint8_t b)
1057 {
1058     return (((r) / 47) % 6) *6 * 6+(((g) / 47) % 6) *6+(((b) / 47) % 6));
1059 }
1060
1061 static void build_rgb_palette(uint8_t *palette, int has_alpha)
1062 {
1063     uint32_t *pal;
1064     static const uint8_t pal_value[6] = {0x00, 0x33, 0x66, 0x99, 0xcc, 0xff };
1065     int i, r, g, b;
1066
1067     pal = (uint32_t*)palette;
1068     i = 0;
1069     for (r = 0; r < 6; r++)
1070     {
1071         for (g = 0; g < 6; g++)
1072         {
1073             for (b = 0; b < 6; b++)
1074             {
1075                 pal[i++] = (0xff << 24) | (pal_value[r] << 16) | (pal_value[g] << 8) | pal_value[b];
1076             }
1077         }

```

```

1078     }
1079     if (has_alpha)
1080         pal[i++] = 0;
1081     while (i < 256)
1082         pal[i++] = 0xff000000;
1083 }
1084
1085 /* copy bit n to bits 0 ... n - 1 */
1086 static inline unsigned int bitcopy_n(unsigned int a, int n)
1087 {
1088     int mask;
1089     mask = (1 << n) - 1;
1090     return (a & (0xff & ~mask)) | (( - ((a >> n) & 1)) & mask);
1091 }
1092
1093 /* rgb555 handling */
1094
1095 #define RGB_NAME rgb555
1096
1097 #define RGB_IN(r, g, b, s)\
1098 {\
1099     unsigned int v = ((const uint16_t *)(s))[0];\
1100 r = bitcopy_n(v >> (10 - 3), 3);\
1101 g = bitcopy_n(v >> (5 - 3), 3);\
1102 b = bitcopy_n(v << 3, 3);\
1103 }
1104
1105 #define RGBA_IN(r, g, b, a, s)\\
1106 {\
1107 unsigned int v = ((const uint16_t *)(s))[0];\
1108 r = bitcopy_n(v >> (10 - 3), 3);\
1109 g = bitcopy_n(v >> (5 - 3), 3);\
1110 b = bitcopy_n(v << 3, 3);\
1111 a = (- (v >> 15)) & 0xff; \
1112 }
1113
1114 #define RGBA_OUT(d, r, g, b, a)\\
1115 {\
1116 ((uint16_t *)(d))[0] = ((r >> 3) << 10) | ((g >> 3) << 5) | (b >> 3) | \
1117 ((a << 8) & 0x8000);\
1118 }
1119
1120 #define BPP 2

```

```

1121
1122 #include "imgconvert_template.h"
1123
1124 /* rgb565 handling */
1125
1126 #define RGB_NAME rgb565
1127
1128 #define RGB_IN(r, g, b, s) \
1129 { \
1130     unsigned int v = ((const uint16_t *)s)[0]; \
1131     r = bitcopy_n(v >> (11 - 3), 3); \
1132     g = bitcopy_n(v >> (5 - 2), 2); \
1133     b = bitcopy_n(v << 3, 3); \
1134 }
1135
1136 #define RGB_OUT(d, r, g, b) \
1137 { \
1138     ((uint16_t *)d)[0] = ((r >> 3) << 11) | ((g >> 2) << 5) | (b >> 3); \
1139 }
1140
1141 #define BPP 2
1142
1143 #include "imgconvert_template.h"
1144
1145 /* bgr24 handling */
1146
1147 #define RGB_NAME bgr24


---


1148
1149 #define RGB_IN(r, g, b, s) \
1150 { \
1151     b = (s)[0]; \
1152     g = (s)[1]; \
1153     r = (s)[2]; \
1154 }
1155
1156 #define RGB_OUT(d, r, g, b) \
1157 { \
1158     (d)[0] = b; \
1159     (d)[1] = g; \
1160     (d)[2] = r; \
1161 }
1162
1163 #define BPP 3

```

```

1164
1165 #include "imgconvert_template.h"
1166
1167 #undef RGB_IN
1168 #undef RGB_OUT
1169 #undef BPP
1170
1171 /* rgb24 handling */
1172
1173 #define RGB_NAME rgb24
1174 #define FMT_RGB24
1175
1176 #define RGB_IN(r, g, b, s) \
1177 { \
1178 r = (s)[0]; \
1179 g = (s)[1]; \
1180 b = (s)[2]; \
1181 }
1182
1183 #define RGB_OUT(d, r, g, b) \
1184 { \
1185 (d)[0] = r; \
1186 (d)[1] = g; \
1187 (d)[2] = b; \
1188 }


---


1189
1190 #define BPP 3
1191
1192 #include "imgconvert_template.h"
1193
1194 /* rgba32 handling */
1195
1196 #define RGB_NAME rgba32
1197 #define FMT_RGBA32
1198
1199 #define RGB_IN(r, g, b, s) \
1200 { \
1201     unsigned int v = ((const uint32_t *) (s))[0]; \
1202     r = (v >> 16) & 0xff; \
1203     g = (v >> 8) & 0xff; \
1204     b = v & 0xff; \
1205 }
1206

```

```

1207 #define RGBA_IN(r, g, b, a, s) \
1208 { \
1209     unsigned int v = ((const uint32_t *)(s))[0]; \
1210     a = (v >> 24) & 0xff; \
1211     r = (v >> 16) & 0xff; \
1212     g = (v >> 8) & 0xff; \
1213     b = v & 0xff; \
1214 } \
1215 \
1216 #define RGBA_OUT(d, r, g, b, a) \
1217 { \
1218     ((uint32_t *)(d))[0] = (a << 24) | (r << 16) | (g << 8) | b; \
1219 } \
1220 \
1221 #define BPP 4 \
1222 \
1223 #include "imgconvert_template.h" \
1224 \
1225 static void mono_to_gray(AVPicture *dst, const AVPicture *src, int width, int height, int xor_mask) \
1226 { \
1227     const unsigned char *p; \
1228     unsigned char *q; \
1229     int v, dst_wrap, src_wrap; \
1230     int y, w; \
1231 \
1232     p = src->data[0]; \
1233     src_wrap = src->linesize[0] - ((width + 7) >> 3); \
1234 \
1235     q = dst->data[0]; \
1236     dst_wrap = dst->linesize[0] - width; \
1237     for (y = 0; y < height; y++) \
1238     { \
1239         w = width; \
1240         while (w >= 8) \
1241         { \
1242             v = *p++ ^ xor_mask; \
1243             q[0] = - (v >> 7); \
1244             q[1] = - ((v >> 6) & 1); \
1245             q[2] = - ((v >> 5) & 1); \
1246             q[3] = - ((v >> 4) & 1); \
1247             q[4] = - ((v >> 3) & 1); \
1248             q[5] = - ((v >> 2) & 1); \
1249             q[6] = - ((v >> 1) & 1); \

```

```

1250     q[7] = -((v >> 0) & 1);
1251     w -= 8;
1252     q += 8;
1253 }
1254 if (w > 0)
1255 {
1256     v = *p++ ^ xor_mask;
1257     do
1258     {
1259         q[0] = -((v >> 7) & 1);
1260         q++;
1261         v <<= 1;
1262     }
1263     while (--w);
1264 }
1265 p += src_wrap;
1266 q += dst_wrap;
1267 }
1268 }
1269
1270 static void monowhite_to_gray(APPicture *dst, const AVPicture *src, int width, int height)
1271 {
1272     mono_to_gray(dst, src, width, height, 0xff);
1273 }
1274
1275 static void monoblack_to_gray(APPicture *dst, const AVPicture *src, int width, int height)
1276 {
1277     mono_to_gray(dst, src, width, height, 0x00);
1278 }
1279
1280 static void gray_to_mono(APPicture *dst, const AVPicture *src, int width, int height, int xor_mask)
1281 {
1282     int n;
1283     const uint8_t *s;
1284     uint8_t *d;
1285     int j, b, v, n1, src_wrap, dst_wrap, y;
1286
1287     s = src->data[0];
1288     src_wrap = src->linesize[0] - width;
1289
1290     d = dst->data[0];
1291     dst_wrap = dst->linesize[0] - ((width + 7) >> 3);
1292 }
```

```

1293     for (y = 0; y < height; y++)
1294     {
1295         n = width;
1296         while (n >= 8)
1297         {
1298             v = 0;
1299             for (j = 0; j < 8; j++)
1300             {
1301                 b = s[0];
1302                 s++;
1303                 v = (v << 1) | (b >> 7);
1304             }
1305             d[0] = v ^ xor_mask;
1306             d++;
1307             n -= 8;
1308         }
1309         if (n > 0)
1310         {
1311             n1 = n;
1312             v = 0;

```

```

1313         while (n > 0)
1314         {
1315             b = s[0];
1316             s++;
1317             v = (v << 1) | (b >> 7);
1318             n--;
1319         }
1320         d[0] = (v << (8-(n1 &7))) ^ xor_mask;
1321         d++;
1322     }
1323     s += src_wrap;
1324     d += dst_wrap;
1325 }
1326 }
1327
1328 static void gray_to_monowhite(AVPicture *dst, const AVPicture *src, int width, int height)
1329 {
1330     gray_to_mono(dst, src, width, height, 0xff);
1331 }
1332
1333 static void gray_to_monoblack(AVPicture *dst, const AVPicture *src, int width, int height)

```

```

1334 {
1335     gray_to_mono(dst, src, width, height, 0x00);
1336 }
1337
1338 typedef struct ConvertEntry
1339 {
1340     void(*convert)(AVPicture *dst, const AVPicture *src, int width, int height);
1341 } ConvertEntry;
1342
1343 /* Add each new conversion function in this table. In order to be able
1344 to convert from any format to any format, the following constraints must be satisfied:
1345
1346     - all FF_COLOR_RGB formats must convert to and from PIX_FMT_RGB24
1347
1348     - all FF_COLOR_GRAY formats must convert to and from PIX_FMT_GRAY8
1349
1350     - all FF_COLOR_RGB formats with alpha must convert to and from PIX_FMT_RGBA32
1351
1352     - PIX_FMT_YUV444P and PIX_FMT_YUVJ444P must convert to and from PIX_FMT_RGB24.
1353
1354     - PIX_FMT_422 must convert to and from PIX_FMT_422P.
1355
1356     The other conversion functions are just optimisations for common cases.
1357 */
1358
1359 static ConvertEntry convert_table[PIX_FMT_NB][PIX_FMT_NB];
1360
1361 static void img_convert_init(void)
1362 {
1363     int i;
1364     uint8_t *cm = cropTbl + MAX_NEG_CROP;
1365
1366     for (i = 0; i < 256; i++)
1367     {
1368         y_ccir_to_jpeg[i] = Y_CCIR_TO_JPEG(i);
1369         y_jpeg_to_ccir[i] = Y_JPEG_TO_CCIR(i);
1370         c_ccir_to_jpeg[i] = C_CCIR_TO_JPEG(i);
1371         c_jpeg_to_ccir[i] = C_JPEG_TO_CCIR(i);
1372     }
1373
1374     convert_table[PIX_FMT_YUV420P][PIX_FMT_YUV422].convert = yuv420p_to_yuv422;
1375     convert_table[PIX_FMT_YUV420P][PIX_FMT_YUV422].convert = yuv420p_to_yuv422;
1376     convert_table[PIX_FMT_YUV420P][PIX_FMT_RGB555].convert = yuv420p_to_rgb555;

```

## 《FFmpeg 基础库编程开发》

```
1377 convert_table[PIX_FMT_YUV420P][PIX_FMT_RGB565].convert = yuv420p_to_rgb565;
1378 convert_table[PIX_FMT_YUV420P][PIX_FMT_BGR24].convert = yuv420p_to_bgr24;
1379 convert_table[PIX_FMT_YUV420P][PIX_FMT_RGB24].convert = yuv420p_to_rgb24;
1380 convert_table[PIX_FMT_YUV420P][PIX_FMT_RGBA32].convert = yuv420p_to_rgba32;
1381 convert_table[PIX_FMT_YUV420P][PIX_FMT_UYVY422].convert = yuv420p_to_uyvy422;
1382
1383 convert_table[PIX_FMT_YUV422P][PIX_FMT_YUV422].convert = yuv422p_to_yuv422;
1384 convert_table[PIX_FMT_YUV422P][PIX_FMT_UYVY422].convert = yuv422p_to_uyvy422;
1385
1386 convert_table[PIX_FMT_YUV444P][PIX_FMT_RGB24].convert = yuv444p_to_rgb24;
1387
1388 convert_table[PIX_FMT_YUVJ420P][PIX_FMT_RGB555].convert = yuvj420p_to_rgb555;
1389 convert_table[PIX_FMT_YUVJ420P][PIX_FMT_RGB565].convert = yuvj420p_to_rgb565;
1390 convert_table[PIX_FMT_YUVJ420P][PIX_FMT_BGR24].convert = yuvj420p_to_bgr24;
1391 convert_table[PIX_FMT_YUVJ420P][PIX_FMT_RGB24].convert = yuvj420p_to_rgb24;
1392 convert_table[PIX_FMT_YUVJ420P][PIX_FMT_RGBA32].convert = yuvj420p_to_rgba32;
1393
1394 convert_table[PIX_FMT_YUVJ444P][PIX_FMT_RGB24].convert = yuvj444p_to_rgb24;
1395
1396 convert_table[PIX_FMT_YUV422][PIX_FMT_YUV420P].convert = yuv422_to_yuv420p;
1397 convert_table[PIX_FMT_YUV422][PIX_FMT_YUV422P].convert = yuv422_to_yuv422p;
1398
1399 convert_table[PIX_FMT_UYVY422][PIX_FMT_YUV420P].convert = uyvy422_to_yuv420p;
1400 convert_table[PIX_FMT_UYVY422][PIX_FMT_YUV422P].convert = uyvy422_to_yuv422p;
1401
1402 convert_table[PIX_FMT_RGB24][PIX_FMT_YUV420P].convert = rgb24_to_yuv420p;
1403 convert_table[PIX_FMT_RGB24][PIX_FMT_RGB565].convert = rgb24_to_rgb565;
1404 convert_table[PIX_FMT_RGB24][PIX_FMT_RGB555].convert = rgb24_to_rgb555;
1405 convert_table[PIX_FMT_RGB24][PIX_FMT_RGBA32].convert = rgb24_to_rgba32;
1406 convert_table[PIX_FMT_RGB24][PIX_FMT_BGR24].convert = rgb24_to_bgr24;
1407 convert_table[PIX_FMT_RGB24][PIX_FMT_GRAY8].convert = rgb24_to_gray;
1408 convert_table[PIX_FMT_RGB24][PIX_FMT_PAL8].convert = rgb24_to_pal8;
1409 convert_table[PIX_FMT_RGB24][PIX_FMT_YUV444P].convert = rgb24_to_yuv444p;
1410 convert_table[PIX_FMT_RGB24][PIX_FMT_YUVJ420P].convert = rgb24_to_yuvj420p;
1411 convert_table[PIX_FMT_RGB24][PIX_FMT_YUVJ444P].convert = rgb24_to_yuvj444p;
1412
1413 convert_table[PIX_FMT_RGBA32][PIX_FMT_RGB24].convert = rgba32_to_rgb24;
1414 convert_table[PIX_FMT_RGBA32][PIX_FMT_RGB555].convert = rgba32_to_rgb555;
1415 convert_table[PIX_FMT_RGBA32][PIX_FMT_PAL8].convert = rgba32_to_pal8;
1416 convert_table[PIX_FMT_RGBA32][PIX_FMT_YUV420P].convert = rgba32_to_yuv420p;
1417 convert_table[PIX_FMT_RGBA32][PIX_FMT_GRAY8].convert = rgba32_to_gray;
1418
1419 convert_table[PIX_FMT_BGR24][PIX_FMT_RGB24].convert = bgr24_to_rgb24;
```

## 《FFmpeg 基础库编程开发》

```
1420 convert_table[PIX_FMT_BGR24][PIX_FMT_YUV420P].convert = bgr24_to_yuv420p;
1421 convert_table[PIX_FMT_BGR24][PIX_FMT_GRAY8].convert = bgr24_to_gray;
1422
1423 convert_table[PIX_FMT_RGB555][PIX_FMT_RGB24].convert = rgb555_to_rgb24;
1424 convert_table[PIX_FMT_RGB555][PIX_FMT_RGBA32].convert = rgb555_to_rgba32;
1425 convert_table[PIX_FMT_RGB555][PIX_FMT_YUV420P].convert = rgb555_to_yuv420p;
1426 convert_table[PIX_FMT_RGB555][PIX_FMT_GRAY8].convert = rgb555_to_gray;
1427
1428 convert_table[PIX_FMT_RGB565][PIX_FMT_RGB24].convert = rgb565_to_rgb24;
1429 convert_table[PIX_FMT_RGB565][PIX_FMT_YUV420P].convert = rgb565_to_yuv420p;
1430 convert_table[PIX_FMT_RGB565][PIX_FMT_GRAY8].convert = rgb565_to_gray;
1431
1432 convert_table[PIX_FMT_GRAY8][PIX_FMT_RGB555].convert = gray_to_rgb555;
1433 convert_table[PIX_FMT_GRAY8][PIX_FMT_RGB565].convert = gray_to_rgb565;
1434 convert_table[PIX_FMT_GRAY8][PIX_FMT_RGB24].convert = gray_to_rgb24;


---


1435 convert_table[PIX_FMT_GRAY8][PIX_FMT_BGR24].convert = gray_to_bgr24;
1436 convert_table[PIX_FMT_GRAY8][PIX_FMT_RGBA32].convert = gray_to_rgba32;
1437 convert_table[PIX_FMT_GRAY8][PIX_FMT_MONOWHITE].convert = gray_to_monowhite;
1438 convert_table[PIX_FMT_GRAY8][PIX_FMT_MONOBLACK].convert = gray_to_monoblack;
1439
1440 convert_table[PIX_FMT_MONOWHITE][PIX_FMT_GRAY8].convert = monowhite_to_gray;
1441
1442 convert_table[PIX_FMT_MONOBLACK][PIX_FMT_GRAY8].convert = monoblack_to_gray;
1443
1444 convert_table[PIX_FMT_PAL8][PIX_FMT_RGB555].convert = pal8_to_rgb555;
1445 convert_table[PIX_FMT_PAL8][PIX_FMT_RGB565].convert = pal8_to_rgb565;
1446 convert_table[PIX_FMT_PAL8][PIX_FMT_BGR24].convert = pal8_to_bgr24;
1447 convert_table[PIX_FMT_PAL8][PIX_FMT_RGB24].convert = pal8_to_rgb24;
1448 convert_table[PIX_FMT_PAL8][PIX_FMT_RGBA32].convert = pal8_to_rgba32;
1449
1450 convert_table[PIX_FMT_UYVY411][PIX_FMT_YUV411P].convert = uyvy411_to_yuv411p;
1451 }
1452
1453 static inline int is_yuv_planar(PixFmtInfo *ps)
1454 {
1455     return (ps->color_type == FF_COLOR_YUV || ps->color_type == FF_COLOR_YUV_JPEG)
1456     && ps->pixel_type == FF_PIXEL_PLANAR;
1457 }
1458
1459 int img_convert(APicture *dst, int dst_pix_fmt, const APicture *src, int src_pix_fmt,
1460     int src_width, int src_height)
1461 {
1462     static int initied;
```

## 《FFmpeg 基础库编程开发》

```
1463 int i, ret, dst_width, dst_height, int_pix_fmt;
1464 PixFmtInfo *src_pix, *dst_pix;
1465 ConvertEntry *ce;
1466 AVPicture tmp1, *tmp = &tmp1;
1467
1468 if (src_pix_fmt < 0 || src_pix_fmt >= PIX_FMT_NB || dst_pix_fmt < 0 || dst_pix_fmt >= PIX_FMT_NB)
1469 return -1;
1470
1471 if (src_width <= 0 || src_height <= 0)
1472 return 0;
1473
1474 if (!inited)
1475 {
1476     inited = 1;
1477     img_convert_init();
1478 }
1479
1480 dst_width = src_width;
1481 dst_height = src_height;
1482
1483 dst_pix = &pix_fmt_info[dst_pix_fmt];
1484 src_pix = &pix_fmt_info[src_pix_fmt];
1485
1486 if (src_pix_fmt == dst_pix_fmt) // no conversion needed: just copy
1487 {
1488     img_copy(dst, src, dst_pix_fmt, dst_width, dst_height);
1489     return 0;
1490 }
1491
1492 ce = &convert_table[src_pix_fmt][dst_pix_fmt];
1493 if (ce->convert)
1494 {
1495     ce->convert(dst, src, dst_width, dst_height); // specific conversion routine
1496     return 0;
1497 }
1498
1499 if (is_yuv_planar(dst_pix) && src_pix_fmt == PIX_FMT_GRAY8) // gray to YUV
1500 {
1501     int w, h, y;
1502     uint8_t *d;
1503
1504     if (dst_pix->color_type == FF_COLOR_YUV_JPEG)
1505     {
```

```

1506 ff_img_copy_plane(dst->data[0], dst->linesize[0], src->data[0], src->linesize[0] ,
1507 dst_width, dst_height);
1508 }
1509 else
1510 {
1511 img_apply_table(dst->data[0], dst->linesize[0], src->data[0], src->linesize[0],
1512 dst_width, dst_height, y_jpeg_to_ccir);
1513 }
1514
1515 w = dst_width; // fill U and V with 128
1516 h = dst_height;


---


1517 w >>= dst_pix->x_chroma_shift;
1518 h >>= dst_pix->y_chroma_shift;
1519 for (i = 1; i <= 2; i++)
1520 {
1521 d = dst->data[i];
1522 for (y = 0; y < h; y++)
1523 {
1524 memset(d, 128, w);
1525 d += dst->linesize[i];
1526 }
1527 }
1528 return 0;
1529 }
1530
1531 if (is_yuv_planar(src_pix) && dst_pix_fmt == PIX_FMT_GRAY8) // YUV to gray
1532 {
1533 if (src_pix->color_type == FF_COLOR_YUV_JPEG)
1534 {
1535 ff_img_copy_plane(dst->data[0], dst->linesize[0], src->data[0], src->linesize[0] ,
1536 dst_width, dst_height);
1537 }
1538 else
1539 {
1540 img_apply_table(dst->data[0], dst->linesize[0], src->data[0], src->linesize[0],
1541 dst_width, dst_height, y_ccir_to_jpeg);
1542 }
1543 return 0;
1544 }
1545
1546 if (is_yuv_planar(dst_pix) && is_yuv_planar(src_pix)) // YUV to YUV planar
1547 {
1548 int x_shift, y_shift, w, h, xy_shift;

```

## 《FFmpeg 基础库编程开发》

```
1549 void(*resize_func)(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap,
1550 int width, int height);
1551
1552 // compute chroma size of the smallest dimensions
1553 w = dst_width;
1554 h = dst_height;
1555 if (dst_pix->x_chroma_shift >= src_pix->x_chroma_shift)
1556 w >>= dst_pix->x_chroma_shift;
1557 else
1558 w >>= src_pix->x_chroma_shift;
1559 if (dst_pix->y_chroma_shift >= src_pix->y_chroma_shift)
1560 h >>= dst_pix->y_chroma_shift;
1561 else
1562 h >>= src_pix->y_chroma_shift;
1563
1564 x_shift = (dst_pix->x_chroma_shift - src_pix->x_chroma_shift);
1565 y_shift = (dst_pix->y_chroma_shift - src_pix->y_chroma_shift);
1566 xy_shift = ((x_shift &0xf) << 4) | (y_shift &0xf);
1567
1568 // there must be filters for conversion at least from and to YUV444 format
1569 switch (xy_shift)
1570 {
1571 case 0x00:
1572 resize_func = ff_img_copy_plane;
1573 break;
1574 case 0x10:
1575 resize_func = shrink21;
1576 break;
1577 case 0x20:
1578 resize_func = shrink41;
1579 break;
1580 case 0x01:
1581 resize_func = shrink12;
1582 break;
1583 case 0x11:
1584 resize_func = ff_shrink22;
1585 break;
1586 case 0x22:
1587 resize_func = ff_shrink44;
1588 break;
1589 case 0xf0:
1590 resize_func = grow21;
1591 break;
```

```

1592 case 0xe0:
1593     resize_func = grow41;
1594     break;
1595 case 0xff:
1596     resize_func = grow22;
1597     break;
1598 case 0xee:
1599     resize_func = grow44;
1600     break;
1601 case 0xf1:
1602     resize_func = conv411;
1603     break;
1604 default:
1605     goto no_chroma_filter; // currently not handled
1606 }
1607
1608 ff_img_copy_plane(dst->data[0], dst->linesize[0], src->data[0], src->linesize[0],
1609 dst_width, dst_height);
1610
1611 for (i = 1; i <= 2; i++)
1612     resize_func(dst->data[i], dst->linesize[i], src->data[i], src->linesize[i],
1613     dst_width >> dst_pix->x_chroma_shift, dst_height >> dst_pix->y_chroma_shift);
1614
1615 // if yuv color space conversion is needed, we do it here on the destination image
1616 if (dst_pix->color_type != src_pix->color_type)
1617 {
1618     const uint8_t *y_table, *c_table;
1619     if (dst_pix->color_type == FF_COLOR_YUV)
1620     {
1621         y_table = y_jpeg_to_ccir;
1622         c_table = c_jpeg_to_ccir;
1623     }
1624     else
1625     {
1626         y_table = y_ccir_to_jpeg;
1627         c_table = c_ccir_to_jpeg;
1628     }
1629
1630     img_apply_table(dst->data[0], dst->linesize[0], dst->data[0], dst->linesize[0],
1631     dst_width, dst_height, y_table);
1632
1633     for (i = 1; i <= 2; i++)
1634         img_apply_table(dst->data[i], dst->linesize[i], dst->data[i], dst->linesize[i],

```

## 《FFmpeg 基础库编程开发》

```
1635     dst_width >> dst_pix->x_chroma_shift, dst_height >> dst_pix->y_chroma_shift, c_table);  
1636 }  
1637 return 0;  
1638 }  
1639

---

  
1640 no_chroma_filter: // try to use an intermediate format  
1641  
1642 if (src_pix_fmt == PIX_FMT_YUV422 || dst_pix_fmt == PIX_FMT_YUV422)  
1643 {  
1644     int_pix_fmt = PIX_FMT_YUV422P; // specific case: convert to YUV422P first  
1645 }  
1646 else if (src_pix_fmt == PIX_FMT_UYVY422 || dst_pix_fmt == PIX_FMT_UYVY422)  
1647 {  
1648     int_pix_fmt = PIX_FMT_YUV422P; // specific case: convert to YUV422P first  
1649 }  
1650 else if (src_pix_fmt == PIX_FMT_UYVY411 || dst_pix_fmt == PIX_FMT_UYVY411)  
1651 {  
1652     int_pix_fmt = PIX_FMT_YUV411P; // specific case: convert to YUV411P first  
1653 }  
1654 else if ((src_pix->color_type == FF_COLOR_GRAY && src_pix_fmt != PIX_FMT_GRAY8)  
1655 || (dst_pix->color_type == FF_COLOR_GRAY && dst_pix_fmt != PIX_FMT_GRAY8))  
1656 {  
1657     int_pix_fmt = PIX_FMT_GRAY8;// gray8 is the normalized format  
1658 }  
1659 else if ((is_yuv_planar(src_pix) && src_pix_fmt != PIX_FMT_YUV444P  
1660 && src_pix_fmt != PIX_FMT_YUVJ444P))  
1661 {  
1662     if (src_pix->color_type == FF_COLOR_YUV_JPEG) // yuv444 is the normalized format  
1663     int_pix_fmt = PIX_FMT_YUVJ444P;  
1664     else  
1665     int_pix_fmt = PIX_FMT_YUV444P;  
1666 }  
1667 else if ((is_yuv_planar(dst_pix) && dst_pix_fmt != PIX_FMT_YUV444P  
1668 && dst_pix_fmt != PIX_FMT_YUVJ444P))  
1669 {  
1670     if (dst_pix->color_type == FF_COLOR_YUV_JPEG) // yuv444 is the normalized format  
1671     int_pix_fmt = PIX_FMT_YUVJ444P;  
1672     else  
1673     int_pix_fmt = PIX_FMT_YUV444P;  
1674 }  
1675 }
```

```
1678     else // the two formats are rgb or gray8 or yuv[j]444p
1679     {
1680         if (src_pix->is_alpha && dst_pix->is_alpha)
1681             int_pix_fmt = PIX_FMT_RGBA32;
1682         else
1683             int_pix_fmt = PIX_FMT_RGB24;
1684     }
1685
1686     if (avpicture_alloc(tmp, int_pix_fmt, dst_width, dst_height) < 0)
1687         return -1;
1688
1689     ret = -1;
1690
1691     if (img_convert(tmp, int_pix_fmt, src, src_pix_fmt, src_width, src_height) < 0)
1692         goto fail1;
1693
1694     if (img_convert(dst, dst_pix_fmt, tmp, int_pix_fmt, dst_width, dst_height) < 0)
1695         goto fail1;
1696     ret = 0;
1697
1698 fail1:
1699     avpicture_free(tmp);
1700     return ret;
1701 }
1702
1703 #undef FIX
```

## 9 msrle.c 文件

### 9.1 功能描述

此文件实现微软行程长度压缩算法解码器，此文件请各位参考压缩算法自己仔细分析。

### 9.2 文件注释

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #include "../libavutil/common.h"
6 #include "avcodec.h"
7 #include "dsputil.h"
8
9 #define FF_BUFFER_HINTS_VALID 0x01 // Buffer hints value is meaningful (if 0 ignore)
10 #define FF_BUFFER_HINTS_READABLE 0x02 // Codec will read from buffer
11 #define FF_BUFFER_HINTS_PRESERVE 0x04 // User must not alter buffer content
12 #define FF_BUFFER_HINTS_REUSABLE 0x08 // Codec will reuse the buffer (update)
13

```

此文件请各位参考压缩算法自己仔细分析。

```

14 typedef struct MsrleContext
15 {
16     AVCodecContext *avctx;
17     AVFrame frame;
18
19     unsigned char *buf;
20     int size;
21
22 } MsrleContext;
23
24 #define FETCH_NEXT_STREAM_BYTE() \
25 if (stream_ptr >= s->size) \
26 { \
27     return; \
28 } \
29 stream_byte = s->buf[stream_ptr++];
30
31 static void msrle_decode_pal4(MsrleContext *s)
32 {
33     int stream_ptr = 0;
34     unsigned char rle_code;
35     unsigned char extra_byte, odd_pixel;

```

```
36 unsigned char stream_byte;
37 int pixel_ptr = 0;
38 int row_dec = s->frame.linesize[0];
39 int row_ptr = (s->avctx->height - 1) *row_dec;
40 int frame_size = row_dec * s->avctx->height;
41 int i;
42
43 // make the palette available
44 memcpy(s->frame.data[1], s->avctx->palctrl->palette, AVPALETTE_SIZE);
45 if (s->avctx->palctrl->palette_changed)
46 {
47 // s->frame.palette_has_changed = 1;
48 s->avctx->palctrl->palette_changed = 0;
49 }
50
51 while (row_ptr >= 0)
52 {
53 FETCH_NEXT_STREAM_BYTE();
54 rle_code = stream_byte;
55 if (rle_code == 0)
56 {
57 // fetch the next byte to see how to handle escape code
58 FETCH_NEXT_STREAM_BYTE();
59 if (stream_byte == 0)
60 {
61 // line is done, goto the next one
62 row_ptr -= row_dec;
63 pixel_ptr = 0;
64 }
65 else if (stream_byte == 1)
66 {
67 // decode is done
68 return ;
69 }
70 else if (stream_byte == 2)
71 {
72 // reposition frame decode coordinates
73 FETCH_NEXT_STREAM_BYTE();
74 pixel_ptr += stream_byte;
75 FETCH_NEXT_STREAM_BYTE();
76 row_ptr -= stream_byte * row_dec;
77 }
78 else
```

```

79  {
80  // copy pixels from encoded stream
81  odd_pixel = stream_byte &1;
82  rle_code = (stream_byte + 1) / 2;
83  extra_byte = rle_code &0x01;
84  if ((row_ptr + pixel_ptr + stream_byte > frame_size) || (row_ptr < 0))
85  {
86  return ;
87  }
88
89  for (i = 0; i < rle_code; i++)
90  {
91  if (pixel_ptr >= s->avctx->width)
92  break;
93  FETCH_NEXT_STREAM_BYTE();
94  s->frame.data[0][row_ptr + pixel_ptr] = stream_byte >> 4;
95  pixel_ptr++;
96  if (i + 1 == rle_code && odd_pixel)
97  break;
98  if (pixel_ptr >= s->avctx->width)
99  break;
100 s->frame.data[0][row_ptr + pixel_ptr] = stream_byte &0x0F;
101 pixel_ptr++;
102 }
103
104 // if the RLE code is odd, skip a byte in the stream
105 if (extra_byte)
106 stream_ptr++;
107 }
108 }
109 else
110 {
111 // decode a run of data
112 if ((row_ptr + pixel_ptr + stream_byte > frame_size) || (row_ptr < 0))
113 {
114 return ;


---


115 }
116 FETCH_NEXT_STREAM_BYTE();
117 for (i = 0; i < rle_code; i++)
118 {
119 if (pixel_ptr >= s->avctx->width)
120 break;
121 if ((i &1) == 0)

```

## 《FFmpeg 基础库编程开发》

```
122 s->frame.data[0][row_ptr + pixel_ptr] = stream_byte >> 4;
123 else
124 s->frame.data[0][row_ptr + pixel_ptr] = stream_byte &0x0F;
125 pixel_ptr++;
126 }
127 }
128 }
129
130 // one last sanity check on the way out
131 if (stream_ptr < s->size)
132 {
133 // error
134 }
135 }
136
137 static void msrle_decode_pal8(MsrleContext *s)
138 {
139 int stream_ptr = 0;
140 unsigned char rle_code;
141 unsigned char extra_byte;
142 unsigned char stream_byte;
143 int pixel_ptr = 0;
144 int row_dec = s->frame.linesize[0];
145 int row_ptr = (s->avctx->height - 1) *row_dec;
146 int frame_size = row_dec * s->avctx->height;
147
148 // make the palette available
149 memcpy(s->frame.data[1], s->avctx->palctrl->palette, AVPALETTE_SIZE);
150 if (s->avctx->palctrl->palette_changed)
151 {
152 // s->frame.palette_has_changed = 1;
153 s->avctx->palctrl->palette_changed = 0;
154 }
155
156 while (row_ptr >= 0)
```

---

```
157 {
158     FETCH_NEXT_STREAM_BYTE();
159     rle_code = stream_byte;
160 if (rle_code == 0)
161 {
162 // fetch the next byte to see how to handle escape code
163 FETCH_NEXT_STREAM_BYTE();
```

```

164 if (stream_byte == 0)
165 {
166 // line is done, goto the next one
167 row_ptr -= row_dec;
168 pixel_ptr = 0;
169 }
170 else if (stream_byte == 1)
171 {
172 // decode is done
173 return ;
174 }
175 else if (stream_byte == 2)
176 {
177 // reposition frame decode coordinates
178 FETCH_NEXT_STREAM_BYTE();
179 pixel_ptr += stream_byte;
180 FETCH_NEXT_STREAM_BYTE();
181 row_ptr -= stream_byte * row_dec;
182 }
183 else
184 {
185 // copy pixels from encoded stream
186 if ((row_ptr + pixel_ptr + stream_byte > frame_size) || (row_ptr < 0))
187 {
188 return ;
189 }
190
191 rle_code = stream_byte;
192 extra_byte = stream_byte &0x01;
193 if (stream_ptr + rle_code + extra_byte > s->size)
194 {
195 return ;
196 }


---


197 while (rle_code--)
198 {
199
200 FETCH_NEXT_STREAM_BYTE();
201 s->frame.data[0][row_ptr + pixel_ptr] = stream_byte;
202 pixel_ptr++;
203 }
204
205 // if the RLE code is odd, skip a byte in the stream
206 if (extra_byte)

```

```

207 stream_ptr++;
208 }
209 }
210 else
211 {
212 // decode a run of data
213 if ((row_ptr + pixel_ptr + stream_byte > frame_size) || (row_ptr < 0))
214 {
215 return ;
216 }
217
218 FETCH_NEXT_STREAM_BYTE();
219
220 while (rle_code--)
221 {
222 s->frame.data[0][row_ptr + pixel_ptr] = stream_byte;
223 pixel_ptr++;
224 }
225 }
226 }
227
228 // one last sanity check on the way out
229 if (stream_ptr < s->size)
230 {
231 // error
232 }
233 }
234
235 static int msrle_decode_init(AVCodecContext *avctx)
236 {
237 MsrleContext *s = (MsrleContext*)avctx->priv_data;


---


238
239 s->avctx = avctx;
240
241 avctx->pix_fmt = PIX_FMT_PAL8;
242
243 s->frame.data[0] = NULL;
244
245 return 0;
246 }
247
248 static int msrle_decode_frame(AVCodecContext *avctx, void *data, int *data_size, uint8_t *buf, int buf_size)
249 {

```

## 《FFmpeg 基础库编程开发》

```
250 MsrleContext *s = (MsrleContext*)avctx->priv_data;
251
252 s->buf = buf;
253 s->size = buf_size;
254
255 if (avctx->reget_buffer(avctx, &s->frame))
256     return -1;
257
258 switch (avctx->bits_per_sample)
259 {
260     case 8:
261         msrle_decode_pal8(s);
262         break;
263     case 4:
264         msrle_decode_pal4(s);
265         break;
266     default:
267         break;
268 }
269
270 *data_size = sizeof(AVFrame);
271 *(AVFrame**)data = s->frame;
272
273 // report that the buffer was completely consumed
274 return buf_size;
275 }
276
277 static int msrle_decode_end(AVCodecContext *avctx)
278 {
279     MsrleContext *s = (MsrleContext*)avctx->priv_data;
280
281 // release the last frame
282 if (s->frame.data[0])
283     avctx->release_buffer(avctx, &s->frame);
284
285 return 0;
286 }
287
288 AVCodec msrle_decoder =
289 {
290     "msrle",
291     CODEC_TYPE_VIDEO,
292     CODEC_ID_MSRLE,
```

```

293 sizeof(MsrleContext),
294 msrle_decode_init,
295 NULL,
296 msrle_decode_end,
297 msrle_decode_frame


---


298 };

```

## 10 turespeech\_data.h 文件

### 10.1 功能描述

此文件定义 true speed 音频解码器使用的常数，此文件请各位参考 TrueSpeed 压缩算法自己仔细分析。

### 10.2 文件注释

```

1 #ifndef TRUE SPEECH _ DATA
2 #define TRUE SPEECH _ DATA
3
4 #pragma warning(disable:4305)
5

```

此文件请各位参考 TrueSpeed 压缩算法自己仔细分析。

```

6 /* codebooks fo expanding input filter */
7 static const int16_t ts_cb_0[32] =
8 {
9 0x8240, 0x8364, 0x84CE, 0x865D, 0x8805, 0x89DE, 0x8BD7, 0x8DF4,
10 0x9051, 0x92E2, 0x95DE, 0x990F, 0x9C81, 0xA079, 0xA54C, 0AAD2,
11 0xB18A, 0xB90A, 0xC124, 0xC9CC, 0xD339, 0xDDD3, 0xE9D6, 0xF893,
12 0x096F, 0x1ACA, 0x29EC, 0x381F, 0x45F9, 0x546A, 0x63C3, 0x73B5,
13 };
14
15 static const int16_t ts_cb_1[32] =
16 {
17 0x9F65, 0xB56B, 0xC583, 0xD371, 0xE018, 0xEBB4, 0xF61C, 0xFF59,
18 0x085B, 0x1106, 0x1952, 0x214A, 0x28C9, 0x2FF8, 0x36E6, 0x3D92,
19 0x43DF, 0x49BB, 0x4F46, 0x5467, 0x5930, 0x5DA3, 0x61EC, 0x65F9,
20 0x69D4, 0x6D5A, 0x709E, 0x73AD, 0x766B, 0x78F0, 0x7B5A, 0x7DA5,
21 };
22
23 static const int16_t ts_cb_2[16] =
24 {
25 0x96F8, 0xA3B4, 0xAF45, 0xBA53, 0xC4B1, 0xCECC, 0xD86F, 0xE21E,
26 0xEBF3, 0xF640, 0x00F7, 0x0C20, 0x1881, 0x269A, 0x376B, 0x4D60,
27 };
28
29 static const int16_t ts_cb_3[16] =

```

```

30  {
31  0xC654, 0xDEF2, 0xEFAA, 0xFD94, 0x096A, 0x143F, 0x1E7B, 0x282C,
32  0x3176, 0x3A89, 0x439F, 0x4CA2, 0x557F, 0x5E50, 0x6718, 0x6F8D,
33  };
34
35  static const int16_t ts_cb_4[16] =
36  {
37  0xABE7, 0xBBA8, 0xC81C, 0xD326, 0xDD0E, 0xE5D4, 0xEE22, 0xF618,
38  0xFE28, 0x064F, 0x0EB7, 0x17B8, 0x21AA, 0x2D8B, 0x3BA2, 0x4DF9,
39  };
40
41  static const int16_t ts_cb_5[8] = { 0xD51B, 0xF12E, 0x042E, 0x13C7, 0x2260, 0x311B, 0x40DE, 0x5385,};
42
43  static const int16_t ts_cb_6[8] = { 0xB550, 0xC825, 0xD980, 0xE997, 0xF883, 0x0752, 0x1811, 0x2E18,};
44
45  static const int16_t ts_cb_7[8] = { 0xCEF0, 0xE4F9, 0xF6BB, 0x0646, 0x14F5, 0x23FF, 0x356F, 0x4A8D,};
46
47  static const int16_t *ts_codebook[8] = {ts_cb_0, ts_cb_1, ts_cb_2, ts_cb_3,
48  ts_cb_4, ts_cb_5, ts_cb_6, ts_cb_7};
49 /* table used for decoding pulse positions */
50  static const int16_t ts_140[120] =
51  {
52  0xE46, 0xCCC, 0xB6D, 0xA28, 0x08FC, 0x07E8, 0x06EB, 0x0604,
53  0x0532, 0x0474, 0x03C9, 0x0330, 0x02A8, 0x0230, 0x01C7, 0x016C,
54  0x011E, 0x00DC, 0x00A5, 0x0078, 0x0054, 0x0038, 0x0023, 0x0014,
55  0x000A, 0x0004, 0x0001, 0x0000, 0x0000, 0x0000,
56
57  0x0196, 0x017A, 0x015F, 0x0145, 0x012C, 0x0114, 0x00FD, 0x00E7,
58  0x00D2, 0x00BE, 0x00AB, 0x0099, 0x0088, 0x0078, 0x0069, 0x005B,
59  0x004E, 0x0042, 0x0037, 0x002D, 0x0024, 0x001C, 0x0015, 0x000F,
60  0x000A, 0x0006, 0x0003, 0x0001, 0x0000, 0x0000,
61
62  0x001D, 0x001C, 0x001B, 0x001A, 0x0019, 0x0018, 0x0017, 0x0016,
63  0x0015, 0x0014, 0x0013, 0x0012, 0x0011, 0x0010, 0x000F, 0x000E,
64  0x000D, 0x000C, 0x000B, 0x000A, 0x0009, 0x0008, 0x0007, 0x0006,
65  0x0005, 0x0004, 0x0003, 0x0002, 0x0001, 0x0000,
66
67  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001,
68  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001,
69  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001,
70  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
71  };
72

```

```
73 /* filter for correlated input filter */
74 static const int16_t ts_230[8] = { 0x7F3B, 0x7E78, 0x7DB6, 0x7CF5, 0x7C35, 0x7B76, 0x7AB8, 0x79FC };
75
76 /* two-point filters table */
77 static const int16_t ts_240[25 * 2] =
78 {
79     0xED2F, 0x5239,
80     0x54F1, 0xE4A9,
81     0x2620, 0xEE3E,
82     0x09D6, 0x2C40,
83     0xEF5, 0x2BE0,
84
85     0x3FE1, 0x3339,
86     0x442F, 0xE6FE,
87     0x4458, 0xF9DF,
88     0xF231, 0x43DB,
89     0x3DB0, 0xF705,
90
91     0x4F7B, 0xFEFB,
92     0x26AD, 0x0CDC,
93     0x33C2, 0x0739,
94     0x12BE, 0x43A2,
95     0x1BDF, 0x1F3E,
96
97     0x0211, 0x0796,
98     0x2AEB, 0x163F,
99     0x050D, 0x3A38,
100    0x0D1E, 0x0D78,
101    0x150F, 0x3346,
102
103    0x38A4, 0x0B7D,
104    0x2D5D, 0x1FDF,
105    0x19B7, 0x2822,
106    0x0D99, 0x1F12,
107    0x194C, 0x0CE6
108 };
109
110 /* possible pulse values */
111 static const int16_t ts_562[64] =
112 {
113     0x0002, 0x0006, 0xFFFFE, 0xFFFFA,
114     0x0004, 0x000C, 0xFFFFC, 0xFFFF4,
115     0x0006, 0x0012, 0xFFFFA, 0xFFFFE,
```

```

116 0x000A, 0x001E, 0xFFFF6, 0xFFE2,
117 0x0010, 0x0030, 0xFFFF0, 0xFFD0,
118 0x0019, 0x004B, 0xFFE7, 0xFFB5,
119 0x0028, 0x0078, 0xFFD8, 0xFF88,
120 0x0040, 0x00C0, 0xFFC0, 0xFF40,
121 0x0065, 0x012F, 0xFF9B, 0xFED1,
122 0x00A1, 0x01E3, 0xFF5F, 0xFE1D,
123 0x0100, 0x0300, 0xFF00, 0xFD00,
124 0x0196, 0x04C2, 0xFE6A, 0xFB3E,
125 0x0285, 0x078F, 0xFD7B, 0xF871,
126 0x0400, 0x0C00, 0xFC00, 0xF400,
127 0x0659, 0x130B, 0xF9A7, 0xECF5,
128 0xA14, 0x1E3C, 0xF5EC, 0xE1C4
129 };
130
131 /* filters used in final output calculations */
132 static const int16_t ts_5E2[8] = { 0x4666, 0x26B8, 0x154C, 0x0BB6, 0x0671, 0x038B, 0x01F3, 0x0112 };
133
134 static const int16_t ts_5F2[8] = { 0x6000, 0x4800, 0x3600, 0x2880, 0x1E60, 0x16C8, 0x1116, 0x0CD1 };
135
136 #endif

```

---

## 11 turespeech.c 文件

### 11.1 功能描述

此文件实现 true speed 音频解码器，此文件请各位参考压缩算法自己仔细分析。

### 11.2 文件注释

```

1 #include "avcodec.h"
2
3 #include "truespeech_data.h"
4
5 // TrueSpeech decoder context
6
7 typedef struct TSContext
8 {
9     // input data
10    int16_t vector[8]; // input vector: 5/5/4/4/4/3/3/3
11    int offset1[2]; // 8-bit value, used in one copying offset
12    int offset2[4]; // 7-bit value, encodes offsets for copying and for two-point filter
13    int pulseoff[4]; // 4-bit offset of pulse values block
14    int pulsepos[4]; // 27-bit variable, encodes 7 pulse positions

```

```

15 int pulseval[4]; // 7x2-bit pulse values
16 int flag; // 1-bit flag, shows how to choose filters
17 // temporary data
18 int filtbuf[146]; // some big vector used for storing filters
19 int prevfilt[8]; // filter from previous frame
20 int16_t tmp1[8]; // coefficients for adding to out
21 int16_t tmp2[8]; // coefficients for adding to out
22 int16_t tmp3[8]; // coefficients for adding to out
23 int16_t cvector[8]; // correlated input vector
24 int filtval;// gain value for one function
25 int16_t newvec[60]; // tmp vector
26 int16_t filters[32]; // filters for every subframe
27 } TSContext;
28
29 #if !defined(LE_32)
30 #define LE_32(x) (((uint8_t*)(x))[3] << 24)|((uint8_t*)(x))[2] << 16)|\
31 ((uint8_t*)(x))[1] << 8)|((uint8_t*)(x))[0])
32 #endif
33
34 static int truespeech_decode_init(AVCodecContext *avctx)
35 {
36     return 0;
37 }
38
39 static void truespeech_read_frame(TSContext *dec, uint8_t *input)
40 {
41     uint32_t t;
42
43     t = LE_32(input); // first dword
44     input += 4;
45
46     dec->flag = t &1;
47
48     dec->vector[0] = ts_codebook[0][(t >> 1) &0x1F];
49     dec->vector[1] = ts_codebook[1][(t >> 6) &0x1F];
50     dec->vector[2] = ts_codebook[2][(t >> 11) &0xF];
51     dec->vector[3] = ts_codebook[3][(t >> 15) &0xF];
52     dec->vector[4] = ts_codebook[4][(t >> 19) &0xF];
53     dec->vector[5] = ts_codebook[5][(t >> 23) &0x7];
54     dec->vector[6] = ts_codebook[6][(t >> 26) &0x7];
55     dec->vector[7] = ts_codebook[7][(t >> 29) &0x7];
56
57

```

```

58 t = LE_32(input); // second dword
59 input += 4;
60
61 dec->offset2[0] = (t >> 0) &0x7F;
62 dec->offset2[1] = (t >> 7) &0x7F;
63 dec->offset2[2] = (t >> 14) &0x7F;
64 dec->offset2[3] = (t >> 21) &0x7F;
65
66 dec->offset1[0] = ((t >> 28) &0xF) << 4;
67
68
69 t = LE_32(input); // third dword
70 input += 4;
71
72 dec->pulseval[0] = (t >> 0) &0x3FFF;
73 dec->pulseval[1] = (t >> 14) &0x3FFF;


---


74
75 dec->offset1[1] = (t >> 28) &0x0F;
76
77
78 t = LE_32(input); // fourth dword
79 input += 4;
80
81 dec->pulseval[2] = (t >> 0) &0x3FFF;
82 dec->pulseval[3] = (t >> 14) &0x3FFF;
83
84 dec->offset1[1] |= ((t >> 28) &0x0F) << 4;
85
86
87 t = LE_32(input); // fifth dword
88 input += 4;
89
90 dec->pulsepos[0] = (t >> 4) &0x7FFFFFF;
91
92 dec->pulseoff[0] = (t >> 0) &0xF;
93
94 dec->offset1[0] |= (t >> 31) &1;
95
96
97 t = LE_32(input); // sixth dword
98 input += 4;
99
100 dec->pulsepos[1] = (t >> 4) &0x7FFFFFF;

```

```

101
102 dec->pulseoff[1] = (t >> 0) &0xF;
103
104 dec->offset1[0] |= ((t >> 31) &1) << 1;
105
106
107 t = LE_32(input); // seventh dword
108 input += 4;
109
110 dec->pulsepos[2] = (t >> 4) &0x7FFFFFFF;
111
112 dec->pulseoff[2] = (t >> 0) &0xF;
113
114 dec->offset1[0] |= ((t >> 31) &1) << 2;


---


115
116
117 t = LE_32(input); // eighth dword
118 input += 4;
119
120 dec->pulsepos[3] = (t >> 4) &0x7FFFFFFF;
121
122 dec->pulseoff[3] = (t >> 0) &0xF;
123
124 dec->offset1[0] |= ((t >> 31) &1) << 3;
125 }
126
127 static void truespeech_correlate_filter(TSContext *dec)
128 {
129 int16_t tmp[8];
130 int i, j;
131
132 for (i = 0; i < 8; i++)
133 {
134 if (i > 0)
135 {
136 memcpy(tmp, dec->cvector, i *2);
137 for (j = 0; j < i; j++)
138 dec->cvector[j] =((tmp[i-j-1]*dec->vector[i])+(dec->cvector[j]<< 15)+0x4000)>>15;
139 }
140 dec->cvector[i] = (8-dec->vector[i]) >> 3;
141 }
142
143 for (i = 0; i < 8; i++)

```

```

144 dec->cvector[i] = (dec->cvector[i] *ts_230[i]) >> 15;
145
146 dec->filtval = dec->vector[0];
147 }
148
149 static void truespeech_filters_merge(TSContext *dec)
150 {
151 int i;
152
153 if (!dec->flag)
154 {
155     for (i = 0; i < 8; i++)
156     {


---


157     dec->filters[i + 0] = dec->prevfilt[i];
158     dec->filters[i + 8] = dec->prevfilt[i];
159 }
160 }
161 else
162 {
163     for (i = 0; i < 8; i++)
164     {
165     dec->filters[i + 0] = (dec->cvector[i] *21846+dec->prevfilt[i] *10923+16384) >>
15;
166     dec->filters[i + 8] = (dec->cvector[i] *10923+dec->prevfilt[i] *21846+16384) >>
15;
167 }
168 }
169 for (i = 0; i < 8; i++)
170 {
171     dec->filters[i + 16] = dec->cvector[i];
172     dec->filters[i + 24] = dec->cvector[i];
173 }
174 }
175
176 static void truespeech_apply_twopoint_filter(TSContext *dec, int quart)
177 {
178     int16_t tmp[146+60],    *ptr0,    *ptr1,    *filter;
179     int i, t, off;
180
181     t = dec->offset2[quart];
182     if (t == 127)

```

```

183 {
184 memset(dec->newvec, 0, 60 *2);
185 return ;
186 }
187
188 for (i = 0; i < 146; i++)
189 tmp[i] = dec->filtbuf[i];
190
191 off = (t / 25) + dec->offset1[quart >> 1] + 18;
192 ptr0 = tmp + 145-off;
193 ptr1 = tmp + 146;
194 filter = (int16_t*)ts_240 + (t % 25) *2;


---


195 for (i = 0; i < 60; i++)
196 {
197 t = (ptr0[0] *filter[0] + ptr0[1] *filter[1] + 0x2000) >> 14;
198 ptr0++;
199 dec->newvec[i] = t;
200 ptr1[i] = t;
201 }
202 }
203
204 static void truespeech_place_pulses(TSContext *dec, int16_t *out, int quart)
205 {
206 int16_t tmp[7];
207 int i, j, t;
208 int16_t *ptr1, *ptr2;
209 int coef;
210
211 memset(out, 0, 60 *2);
212 for (i = 0; i < 7; i++)
213 {
214 t = dec->pulseval[quart] &3;
215 dec->pulseval[quart] >>= 2;
216 tmp[6-i] = ts_562[dec->pulseoff[quart] *4+t];
217 }
218
219 coef = dec->pulsepos[quart] >> 15;
220 ptr1 = (int16_t*)ts_140 + 30;
221 ptr2 = tmp;
222 for (i = 0, j = 3; (i < 30) && (j > 0); i++)
223 {
224 t = *ptr1++;
225 if (coef >= t)

```

```

226 coef -= t;
227 else
228 {
229     out[i] = *ptr2++;
230     ptr1 += 30;
231     j--;
232 }
233 }
234 coef = dec->pulsepos[quart] &0x7FFF;
235 ptr1 = (int16_t*)ts_140;
236 for (i = 30, j = 4; (i < 60) && (j > 0); i++)


---


237 {
238     t = *ptr1++;
239     if (coef >= t)
240         coef -= t;
241     else
242     {
243         out[i] = *ptr2++;
244         ptr1 += 30;
245         j--;
246     }
247 }
248 }
249
250 static void truespeech_update_filters(TSContext *dec, int16_t *out, int quart)
251 {
252     int i;
253
254     for (i = 0; i < 86; i++)
255         dec->filtbuf[i] = dec->filtbuf[i + 60];
256
257     for (i = 0; i < 60; i++)
258     {
259         dec->filtbuf[i + 86] = out[i] + dec->newvec[i] - (dec->newvec[i] >> 3);
260         out[i] += dec->newvec[i];
261     }
262 }
263
264 static void truespeech_synth(TSContext *dec, int16_t *out, int quart)
265 {
266     int i, k;
267     int t[8];

```

```

268 int16_t *ptr0, *ptr1;
269
270 ptr0 = dec->tmp1;
271 ptr1 = dec->filters + quart * 8;
272 for (i = 0; i < 60; i++)
273 {
274     int sum = 0;
275     for (k = 0; k < 8; k++)
276         sum += ptr0[k] *ptr1[k];
277     sum = (sum + (out[i] << 12) + 0x800) >> 12;

```

---

```

278     out[i] = clip(sum, - 0x7FFE, 0x7FFE);
279     for (k = 7; k > 0; k--)
280         ptr0[k] = ptr0[k - 1];
281     ptr0[0] = out[i];
282 }
283
284 for (i = 0; i < 8; i++)
285 t[i] = (ts_5E2[i] *ptr1[i]) >> 15;
286
287 ptr0 = dec->tmp2;
288 for (i = 0; i < 60; i++)
289 {
290     int sum = 0;
291     for (k = 0; k < 8; k++)
292         sum += ptr0[k] *t[k];
293     for (k = 7; k > 0; k--)
294         ptr0[k] = ptr0[k - 1];
295     ptr0[0] = out[i];
296     out[i] = ((out[i] << 12) - sum) >> 12;
297 }
298
299 for (i = 0; i < 8; i++)
300 t[i] = (ts_5F2[i] *ptr1[i]) >> 15;
301
302 ptr0 = dec->tmp3;
303 for (i = 0; i < 60; i++)
304 {
305     int sum = out[i] << 12;
306     for (k = 0; k < 8; k++)
307         sum += ptr0[k] *t[k];
308     for (k = 7; k > 0; k--)
309         ptr0[k] = ptr0[k - 1];

```

```

310 ptr0[0] = clip((sum + 0x800) >> 12, - 0x7FFE, 0x7FE);
311
312 sum = ((ptr0[1]*(dec->filtval - (dec->filtval >> 2))) >> 4) + sum;
313 sum = sum - (sum >> 3);
314 out[i] = clip((sum + 0x800) >> 12, - 0x7FFE, 0x7FE);
315 }
316 }
317


---


318 static void truespeech_save_prevvec(TSContext *c)
319 {
320 int i;
321
322 for (i = 0; i < 8; i++)
323 c->prevfilt[i] = c->cvector[i];
324 }
325
326 int truespeech_decode_frame(AVCodecContext *avctx, void *data, int *data_size, uint8_t *buf, int buf_size)
327 {
328 TSContext *c = avctx->priv_data;
329
330 int i;
331 short *samples = data;
332 int consumed = 0;
333 int16_t out_buf[240];
334
335 if (!buf_size)
336 return 0;
337
338 while (consumed < buf_size)
339 {
340 truespeech_read_frame(c, buf + consumed);
341 consumed += 32;
342
343 truespeech_correlate_filter(c);
344 truespeech_filters_merge(c);
345
346 memset(out_buf, 0, 240 *2);
347 for (i = 0; i < 4; i++)
348 {
349 truespeech_apply_twopoint_filter(c, i);
350 truespeech_place_pulses(c, out_buf + i * 60, i);
351 truespeech_update_filters(c, out_buf + i * 60, i);
352 truespeech_synth(c, out_buf + i * 60, i);

```

```

353 }
354
355 truespeech_save_prevvec(c);
356
357 for (i = 0; i < 240; i++) // finally output decoded frame
358 *samples++ = out_buf[i];


---


359
360 }
361
362 *data_size = consumed * 15;
363
364 return buf_size;
365 }
366
367 AVCCodec truespeech_decoder =
368 {
369 "truespeech",
370 CODEC_TYPE_AUDIO,
371 CODEC_ID_TRUESPEECH,
372 sizeof(TSContext),
373 truespeech_decode_init,
374 NULL,
375 NULL,
376 truespeech_decode_frame,
377 };

```

## 5.3 libavformat 容器模块

### 1 文件列表

文件类型	文件名	大小(bytes)
h	avformat.h	5352
c	allformats.c	299
c	cutils.c	606
c	file.c	1504
h	avio.h	3103
c	avio.c	2286
c	aviobuf.c	6887
c	utils_format.c	7662
c	avidec.c	21713

## 2 avformat.h 文件

### 2.1 功能描述

定义识别文件格式和媒体类型库使用的宏、数据结构和函数，通常这些宏、数据结构和函数在此模块内相对全局有效。

### 2.2 文件注释

```

1  #ifndef AVFORMAT_H
2  #define AVFORMAT_H
3
4  #ifdef cplusplus
5  extern "C"
6  {
7  #endif
8
9  #define LIBAVFORMAT_VERSION_INT ((50<<16)+(4<<8)+0)
10 #define LIBAVFORMAT_VERSION 50.4.0
11 #define LIBAVFORMAT_BUILD LIBAVFORMAT_VERSION_INT
12
13 #define LIBAVFORMAT_IDENT "Lavf" AV_STRINGIFY(LIBAVFORMAT_VERSION)
14
15 #include "../libavcodec/avcodec.h"
16 #include "avio.h"
17

```

### 一些简单的宏定义

```

18 #define AVERROR_UNKNOWN (-1) // unknown error
19 #define AVERROR_IO (-2) // i/o error
20 #define AVERROR_NUMEXPECTED (-3) // number syntax expected in filename
21 #define AVERROR_INVALIDDATA (-4) // invalid data found
22 #define AVERROR_NOMEM (-5) // not enough memory
23 #define AVERROR_NOFMT (-6) // unknown format
24 #define AVERROR_NOTSUPP (-7) // operation not supported
25
26 #define AVSEEK_FLAG_BACKWARD 1 // seek backward
27 #define AVSEEK_FLAG_BYTE 2 // seeking based on position in bytes
28 #define AVSEEK_FLAG_ANY 4 // seek to any frame, even non keyframes
29
30 #define AVFMT_NOFILE 0x0001 // no file should be opened
31
32 #define PKT_FLAG_KEY 0x0001
33
34 #define AVINDEX_KEYFRAME 0x0001
35

```

```

36 #define AVPROBE_SCORE_MAX    100
37
38 #define MAX_STREAMS 20
39
40 typedef struct AVPacket {
41     {
42         int64_t pts; // presentation time stamp in time_base units // 表示时间，对视频是显示时间
43         int64_t dts; // decompression time stamp in time_base units// 解码时间，这个不是很重要
44         int64_t pos; // byte position in stream, -1 if unknown
45         uint8_t *data; // 实际保存音视频数据缓存的首地址
46         int size; // 实际保存音视频数据缓存的大小
47         int stream_index; // 当前音视频数据包对应的流索引，在本例中用于区别音频还是视频。
48         int flags; // 数据包的一些标记，比如是否是关键帧等。
49         void(*destruct)(struct AVPacket*); // 析构函数指针
50     } AVPacket;
51
52     typedef struct AVPacketList {
53         {
54             AVPacket pkt;
55             struct AVPacketList *next; // 用于把各个 AVPacketList 串联起来。
56         } AVPacketList;
57
58     static inline void av_destruct_packet(AVPacket *pkt)
59     {
60         av_free(pkt->data);
61         pkt->data = NULL;
62         pkt->size = 0;
63     }
64
65     static inline void av_free_packet(AVPacket *pkt)
66     {
67         if (pkt && pkt->destruct)
68             pkt->destruct(pkt);
69     }
70

```

音视频数据包定义，在瘦身后的 ffplay 中，每一个包是一个完整的数据帧。注意保存音视频数据包的内存是 malloc 出来的，用完后应及时用 free 归还给系统。

音视频数据包链表定义，注意每一个 AVPacketList 仅含有一个 AVPacket，和传统的很多很多节点的 list 不同，不要被 list 名字迷惑。

释放掉音视频数据包占用的内存，把首地址置空是一个很好的习惯。

判断一些指针，中转一下，释放掉音视频数据包占用的内存。

读文件往数据包中填数据，注意程序跑到这里时，文件偏移量已确定，要读数据的大小也确定，但是数据包的缓存没有分配。分配好内存后，要初始化包的一些变量。

## 《FFmpeg 基础库编程开发》

```
71 static inline int av_get_packet(ByteIOContext *s, AVPacket *pkt, int size)
72 {
73     int ret;
74     unsigned char *data;
75     if ((unsigned)size > (unsigned)size + FF_INPUT_BUFFER_PADDING_SIZE)
76         return AVERROR_NOMEM;
```

77 分配数据包缓存

```
78     data = av_malloc(size + FF_INPUT_BUFFER_PADDING_SIZE);
79     if (!data)
80         return AVERROR_NOMEM;
```

81

把数据包中 pad 部分清 0，这是一个很好的习惯。缓存清 0 不管在什么情况下都是好习惯。

```
82     memset(data + size, 0, FF_INPUT_BUFFER_PADDING_SIZE);
```

83

设置 AVPacket 其他的成员变量，能确定的就赋确定值，不能确定的赋初值。

```
84     pkt->pts = AV_NOPTS_VALUE;
85     pkt->dts = AV_NOPTS_VALUE;
86     pkt->pos = -1;
87     pkt->flags = 0;
88     pkt->stream_index = 0;
89     pkt->data = data;
90     pkt->size = size;
91     pkt->destruct = av_destruct_packet;
92
```

```
93     pkt->pos = url_ftell(s);
```

94

实际读广义文件填充数据包，如果读文件错误时通常是到了末尾，要归还刚刚 malloc 出来的内存。

```
95     ret = url_fread(s, pkt->data, size);
96     if (ret <= 0)
97         av_free_packet(pkt);
98     else
99         pkt->size = ret;
100
```

```
101     return ret;
```

```
102 }
```

103

为识别文件格式，要读一部分文件头数据来分析匹配 ffplay 支持的文件格式文件特征。于是 AVProbeData 结构就定义了文件名，首地址和大小。此处的读独立于其他文件操作。

```
104 typedef struct AVProbeData
```

```
105 {
```

```
106     const char *filename;
```

```
107     unsigned char *buf;
```

```

108 int buf_size;
109 } AVProbeData;
110
文件索引结构, flags 和 size 位定义是为了节省内存。
111 typedef struct AVIndexEntry
112 {
113     int64_t pos;
114     int64_t timestamp;
115     int flags: 2;
116     int size: 30;
117 } AVIndexEntry;
118
AVStream 抽象的表示一个媒体流, 定义了所有媒体一些通用的属性。
119 typedef struct AVStream
120 {
121     AVCCodecContext *actx;// 关联到解码器//codec context,change from AVCCodecContext *codec;
122
123     void *priv_data; // 在本例中, 关联到 AVIStream
124
125     AVRational time_base; // 由 av_set_pts_info()函数初始化
126
127     AVIndexEntry *index_entries; // only used if the format does not support seeking natively
128     int nb_index_entries;
129     int index_entries_allocated_size;
130
131     double frame_last_delay; // 帧最后延迟
132 } AVStream;
133
AVFormatParameters 结构在瘦身后的 ffplay 中没有实际意义, 为保证函数接口不变, 没有删除。
134 typedef struct AVFormatParameters
135 {
136     int dbg; //only for debug
137 } AVFormatParameters;
138
AVInputFormat 定义输入文件容器格式, 着重于功能函数, 一种文件容器格式对应一个 AVInputFormat
结构, 在程序运行时有多个实例, 但瘦身后 ffplay 仅一个实例。
139 typedef struct AVInputFormat
140 {
141     const char *name; // 文件容器格式名, 用于人性化阅读, 维护代码
142
143     int priv_data_size; // 程序运行时, 文件容器格式对应的上下文结构大小, 便于内存分配。
144
145     int(*read_probe)(AVProbeData*); // 功能性函数

```

146

```

147 int(*read_header)(struct AVFormatContext *, AVFormatParameters *ap);
148
149 int(*read_packet)(struct AVFormatContext *, AVPacket *pkt);
150
151 int(*read_close)(struct AVFormatContext *);
152
153 const char *extensions; // 此种文件容器格式对应的文件扩展名，识别文件格式的最后办法。
154
155 struct AVInputFormat *next; // 用于把 ffplay 支持的所有文件容器格式链成一个链表。
156
157 } AVInputFormat;
158

```

AVFormatContext 结构表示程序运行的当前文件容器格式使用的上下文，着重于所有文件容器共有的属性，程序运行后仅一个实例。

```

159 typedef struct AVFormatContext // format I/O context
160 {
161     struct AVInputFormat *iformat; // 关联程序运行时，实际的文件容器格式指针。
162
163     void *priv_data; // 关联具体文件容器格式上下文的指针，在本例中是 AVIContext
164
165     ByteIOContext pb; // 关联广义输入文件
166
167     int nb_streams; // 广义输入文件中媒体流计数
168
169     AVStream *streams[MAX_STREAMS]; // 关联广义输入文件中的媒体流
170
171 } AVFormatContext;
172

```

相关函数说明参考相应的 c 实现文件。

```

173 int avidec_init(void);
174
175 void av_register_input_format(AVInputFormat *format);
176
177 void av_register_all(void);
178
179 AVInputFormat *av_probe_input_format(AVProbeData *pd, int is_opened);
180 int match_ext(const char *filename, const char *extensions);
181
182 int av_open_input_stream(AVFormatContext **ic_ptr, ByteIOContext *pb, const char *filename,
183                         AVInputFormat *fmt, AVFormatParameters *ap);
184
185 int av_open_input_file(AVFormatContext **ic_ptr, const char *filename, AVInputFormat *fmt,

```

```

186 int buf_size, AVFormatParameters *ap);
187
188 int av_read_frame(AVFormatContext *s, AVPacket *pkt);
189 int av_read_packet(AVFormatContext *s, AVPacket *pkt);
190 void av_close_input_file(AVFormatContext *s);
191 AVStream *av_new_stream(AVFormatContext *s, int id);
192 void av_set_pts_info(AVStream *s, int pts_wrap_bits, int pts_num, int pts_den);
193
194 int av_index_search_timestamp(AVStream *st, int64_t timestamp, int flags);
195 int av_add_index_entry(AVStream *st, int64_t pos, int64_t timestamp, int size, int distance, int flags);
196
197 int strstart(const char *str, const char *val, const char **ptr);
198 void pstrcpy(char *buf, int buf_size, const char *str);
199
200 #ifdef cplusplus
201 }
202
203 #endif
204
205 #endif

```

---

## 3 allformat.c 文件

### 3.1 功能描述

简单的注册/初始化函数，把相应的协议，文件格式，解码器等用相应的链表串起来便于查找。

### 3.2 文件注释

```

1 #include "avformat.h"
2
3 extern URLProtocol file_protocol;
4
5 void av_register_all(void)
6 {
7 到 11 行, initied 变量声明成 static , 做一下比较是为了避免此函数多次调用。

```

编程基本原则之一，初始化函数只调用一次，不能随意多次调用。

```

7 static int initied = 0;
8
9 if (initied != 0)
10 return ;
11 initied = 1;
12

```

ffplay 把 CPU 当做一个广义的 DSP。有些计算可以用 CPU 自带的加速指令来优化，ffplay 把这类函数独立出来放到 dsutil.h 和 dsutil.c 文件中，用函数指针的方法映射到各个 CPU 具体的加速优化实现函数，此处

初始化这些函数指针。

13 avcodec\_init();

14

把所有的解码器用链表的方式都串连起来，链表头指针是 first\_avcodec。

15 avcodec\_register\_all();

16

把所有的输入文件格式用链表的方式都串连起来，链表头指针是 first\_iformat。

17 avidec\_init();

18

把所有的输入协议用链表的方式都串连起来，比如 tcp/udp/file 等，链表头指针是 first\_protocol。

19 register\_protocol(&file\_protocol);

20 }

## 4 cutils.c 文件

### 4.1 功能描述

ffplay 文件格式分析模块使用的两个工具类函数，都是对字符串的操作。

### 4.2 文件注释

1 #include "avformat.h"

2

strstart 实际的功能就是在 str 字符串中搜索 val 字符串指示的头，并且去掉头后用\*ptr 返回。

在本例中，在播本地文件时，在命令行输入时可能会在文件路径名前加前缀"file:"，为调用系统的 open 函数，需要把这几个前导字符去掉，仅仅传入完整有效的文件路径名。和 rtsp://等网络协议相对应，播本地文件时应加 file:前缀。

3 int strstart(const char \*str, const char \*val, const char \*\*ptr)

4 {

5 const char \*p, \*q;

6 p = str;

7 q = val;

8 while (\*q != '\0')

9 {

10 if (\*p != \*q)

11 return 0;

12 p++;

13 q++;

14 }

15 if (ptr)

16 \*ptr = p;

17 return 1;

18 }

19

字符串拷贝函数，拷贝的字符数由 buf\_size 指定，更安全的字符串拷贝操作。

传统的 `strcpy()` 函数是拷贝一个完整的字符串，如果目标字符串缓冲区小于源字符串长度，那么就会发生缓冲区溢出导致错误，并且这种错误很难发现。

```

20 void pstrcpy(char *buf, int buf_size, const char *str)
21 {
22     int c;
23     char *q = buf;
24
25     if (buf_size <= 0)
26         return ;
27
28     for (;;)
29     {
30         c = *str++;
31         if (c == 0 || q >= buf + buf_size - 1)
32             break;
33         *q++ = c;
34     }
35     *q = '\0';
36 }
```

---

## 5 file.c 文件

### 5.1 功能描述

`ffplay` 把 `file` 当做类似于 `rtsp`, `rtp`, `tcp` 等协议的一种协议，用 `file:` 前缀标示 `file` 协议。`URLContext` 结构抽象统一表示这些广义上的协议，对外提供统一的抽象接口。各具体的广义协议实现文件实现 `URLContext` 接口。此文件实现了 `file` 广义协议的 `URLContext` 接口。

### 5.2 文件注释

```

1 #include "../berrno.h"
2
3 #include "avformat.h"
4 #include <fcntl.h>
5
6 #ifndef CONFIG_WIN32
7 #include <unistd.h>
8 #include <sys/ioctl.h>
9 #include <sys/time.h>
10 #else
11 #include <io.h>
12 #define open(fname,oflag,pmode) _open(fname,oflag,pmode)
13 #endif
14
```

打开本地媒体文件，把本地文件句柄作为广义文件句柄存放在 `priv_data` 中。

```
15 static int file_open(URLContext *h, const char *filename, int flags)
```

```
16 {
```

```
17     int access;
```

```
18     int fd;
```

```
19 }
```

规整本地路径文件名，去掉前面可能的“file:”字符串。ffplay 把本地文件看做广义 URL 协议。

```
20     strstart(filename, "file:", &filename);
```

```
21 }
```

设置本地文件存取属性。

```
22     if (flags & URL_RDWR)
```

```
23         access = O_CREAT | O_TRUNC | O_RDWR;
```

```
24     else if (flags & URL_WRONLY)
```

```
25         access = O_CREAT | O_TRUNC | O_WRONLY;
```

```
26     else
```

```

27 access = O_RDONLY;
28 #if defined(CONFIG_WIN32) || defined(CONFIG_OS2) || defined(CYGWIN)
29 access |= O_BINARY;
30#endif

```

调用 open() 打开本地文件，并把本地文件句柄作为广义的 URL 句柄存放在 priv\_data 变量中。

```

31     fd = open(filename, access, 0666);
32     if (fd < 0)
33         return -ENOENT;
34     h->priv_data = (void*)(size_t)fd;
35     return 0;
36 }
37

```

转换广义 URL 句柄为本地文件句柄，调用 read() 函数读本地文件。

```

38 static int file_read(URLContext *h, unsigned char *buf, int size)
39 {
40     int fd = (size_t)h->priv_data;
41     return read(fd, buf, size);
42 }
43

```

转换广义 URL 句柄为本地文件句柄，调用 write() 函数写本地文件，本播放器没实际使用此函数。

```

44 static int file_write(URLContext *h, unsigned char *buf, int size)
45 {
46     int fd = (size_t)h->priv_data;
47     return write(fd, buf, size);
48 }
49

```

转换广义 URL 句柄为本地文件句柄，调用 lseek() 函数设置本地文件读指针。

```

50 static offset_t file_seek(URLContext *h, offset_t pos, int whence)
51 {
52     int fd = (size_t)h->priv_data;
53     return lseek(fd, pos, whence);
54 }
55

```

转换广义 URL 句柄为本地文件句柄，调用 close() 函数关闭本地文件。

```

56 static int file_close(URLContext *h)
57 {
58     int fd = (size_t)h->priv_data;
59     return close(fd);
60 }
61

```

用 file 协议相应函数初始化 URLProtocol 结构。

```

62 URLProtocol file_protocol =
63 {
64 "file",
65 file_open,
66 file_read,
67 file_write,
68 file_seek,
69 file_close,
70 };

```

---

## 6 avio.h 文件

### 6.1 功能描述

文件读写模块定义的数据结构和函数声明，ffplay 把这些全部放到这个.h 文件中。

### 6.2 文件注释

```

1 #ifndef AVIO_H
2 #define AVIO_H
3
4 #define URL_EOF (-1)
5
6 typedef int64_t offset_t;
7

```

简单的文件存取宏定义

```

8 #define URL_RDONLY 0
9 #define URL_WRONLY 1
10 #define URL_RDWR 2
11

```

URLContext 结构表示程序运行的当前广义文件协议使用的上下文，着重于所有广义文件协议共有的 属性(并且是在程序运行时才能确定其值)和关联其他结构的字段。

```

12 typedef struct URLContext
13 {
14     struct URLProtocol *prot; // 关联相应的广义输入文件协议。
15     int flags;
16     int max_packet_size;    // 如果非 0，表示最大包大小，用于分配足够的缓存。
17     void *priv_data;       // 在本例中，关联一个文件句柄
18     char filename[1];      // 在本例中，存取本地文件名，filename 仅指示本地文件名首地址。
19 } URLContext;
20

```

URLProtocol 定义广义的文件协议，着重于功能函数，一种广义的文件协议对应一个 URLProtocol 结构，本例删掉了 pipe, udp, tcp 等输入协议，仅保留一个 file 协议。

```

21 typedef struct URLProtocol
22 {

```

```

23 const char *name; // 协议文件名，便于人性化阅读理解。
24 int(*url_open)(URLContext *h, const char *filename, int flags);
25 int(*url_read)(URLContext *h, unsigned char *buf, int size);
26 int(*url_write)(URLContext *h, unsigned char *buf, int size);
27 offset_t(*url_seek)(URLContext *h, offset_t pos, int whence);
28 int(*url_close)(URLContext *h);
29 struct URLProtocol *next; // 把所有支持的输入协议串链起来，便于遍历查找。
30 } URLProtocol;
31

```

ByteIOContext 结构扩展 URLProtocol 结构成内部有缓冲机制的广泛意义上的文件，改善广义输入文件的 IO 性能。 主要变量间的逻辑位置关系简单示意如下：



```

32 typedef struct ByteIOContext
33 {
    注1：buffer和媒体文件的逻辑示意图，用于缓存的管理，各变量的理解和计算
34     unsigned char *buffer; // 缓存首地址
35     int buffer_size; // 缓存大小
36     灰红表示缓存已使用数据
37     高红表示缓存未使用数据
38     高灰表示文件未读数据
39     int(*read_packet)(void *opaque, uint8_t *buf, int buf_size);
40     int(*write_packet)(void *opaque, uint8_t *buf, int buf_size);
41     offset_t(*seek)(void *opaque, offset_t offset, int whence);
42     offset_t pos; // position in the file of the current buffer
43     int must_flush; // true if the next seek should flush
44     int eof_reached; // true if eof reached
45     int write_flag; // true if open for writing
46     int max_packet_size; // 如果非 0，表示最大数据帧大小，用于分配足够的缓存。
47     int error; // contains the error code or 0 if no error happened
48 } ByteIOContext;

```

相关函数说明参考相应的 c 实现文件。

```

49 int url_open(URLContext **h, const char *filename, int flags);
50 int url_read(URLContext *h, unsigned char *buf, int size);
51 int url_write(URLContext *h, unsigned char *buf, int size);
52 offset_t url_seek(URLContext *h, offset_t pos, int whence);
53 int url_close(URLContext *h);
54 int url_get_max_packet_size(URLContext *h);
55
56 int register_protocol(URLProtocol *protocol);
57

```

```

58 int init_put_byte(ByteIOContext *s,
59 unsigned char *buffer,
60 int buffer_size,
61 int write_flag,
62 void *opaque,
63 int(*read_buf)(void *opaque, uint8_t *buf, int buf_size),
64 int(*write_buf)(void *opaque, uint8_t *buf, int buf_size),
65 offset_t(*seek)(void *opaque, offset_t offset, int whence));
66
67 offset_t url_fseek(ByteIOContext *s, offset_t offset, int whence);
68 void url_fskip(ByteIOContext *s, offset_t offset);
69 offset_t url_ftell(ByteIOContext *s);
70 offset_t url_fsize(ByteIOContext *s);
71 int url_feof(ByteIOContext *s);
72 int url_ferror(ByteIOContext *s);
73
74 int url_fread(ByteIOContext *s, unsigned char *buf, int size); // get_buffer
75 int get_byte(ByteIOContext *s);
76 unsigned int get_le32(ByteIOContext *s);
77 unsigned int get_le16(ByteIOContext *s);
78
79 int url_setbufsize(ByteIOContext *s, int buf_size);
80 int url_fopen(ByteIOContext *s, const char *filename, int flags);
81 int url_fclose(ByteIOContext *s);
82
83 int url_open_buf(ByteIOContext *s, uint8_t *buf, int buf_size, int flags);
84 int url_close_buf(ByteIOContext *s);
85
86 #endif

```

---

## 7 avio.c 文件

### 7.1 功能描述

此文件实现了 URLProtocol 抽象层广义文件操作函数，由于 URLProtocol 是底层其他具体文件 (file, pipe 等) 的简单封装，这一层只是一个中转站，大部分函数都是简单中转到底层的具体实现函数。

### 7.2 文件注释

```

1 #include "../berrror.h"
2 #include "avformat.h"
3
4 URLProtocol *first_protocol = NULL;
5

```

ffplay 抽象底层的 file , pipe 等为 URLProtocol，然后把这些 URLProtocol 串联起来做成链表，便于查找。

## 《FFmpeg 基础库编程开发》

register\_protocol 实际就是串联的各个 URLProtocol，全局表头为 first\_protocol。

```
6 int register_protocol(URLProtocol *protocol)
```

```
7 {
```

```
8     URLProtocol **p;
```

```
9     p = &first_protocol;
```

移动指针到 URLProtocol 链表末尾。

```
10    while (*p != NULL)
```

```
11        p = &(*p)->next;
```

在 URLProtocol 链表末尾直接挂接当前的 URLProtocol 指针。

```
12    *p = protocol;
```

```
13    protocol->next = NULL;
```

```
14    return 0;
```

```
15 }
```

```
16
```

打开广义输入文件。此函数主要有三部分逻辑，首先从文件路径名中分离出协议字符串到 proto\_str 字符数组中，接着遍历 URLProtocol 链表查找匹配 proto\_str 字符数组中的字符串来确定使用的协议，最后调用相应的文件协议的打开函数打开输入文件。

```
17 int url_open(URLContext **puc, const char *filename, int flags)
```

```
18 {
```

```
19     URLContext *uc;
```

```
20     URLProtocol *up;
```

```
21     const char *p;
```

```
22     char proto_str[128],*q;
```

```
23     int err;
```

```
24
```

以冒号和结束符作为边界从文件名中分离出的协议字符串到 proto\_str 字符数组。由于协议只能是字符，所以在边界前识别到非字符就断定是 file。

```
25     p = filename;
```

```
26     q = proto_str;
```

```
27     while (*p != '\0' &&      *p != ':')
```

```
28 {
```

```
29     if (!isalpha(*p)) // protocols can only contain alphabetic chars
```

```
30     goto file_proto;
```

```
31     if ((q - proto_str) < sizeof(proto_str) - 1)
```

```
32     *q++ = *p;
```

```
33     p++;
```

```
34 }
```

```
35
```

如果协议字符串只有一个字符，我们就认为是 windows 下的逻辑盘符，断定是 file。

```
36     if (*p == '\0' || (q - proto_str) <= 1)
```

```
37 {
```

```
38     file_proto:
```

```
39     strcpy(proto_str, "file");
```

```

40 }
41 else
42 {
43     *q = '\0';
44 }
45

```

遍历 URLProtocol 链表匹配使用的协议，如果没有找到就返回错误码。

```

46     up = first_protocol;
47     while (up != NULL)
48     {
49         if (!strcmp(proto_str, up->name))
50             goto found;
51         up = up->next;
52     }
53     err = -ENOENT;
54     goto fail;
55     found:

```

如果找到就分配 URLContext 结构内存，特别注意内存大小要加上文件名长度，文件名字符串结束标记 0 也要预先分配 1 个字节内存，这 1 个字节就是 URLContext 结构中的 char filename[1]。

```

56     uc = av_malloc(sizeof(URLContext) + strlen(filename));
57     if (!uc)
58     {
59         err = -ENOMEM;
60         goto fail;
61     }

```

strcpy 函数会自动在 filename 字符数组后面补 0 作为字符串结束标记，所以不用特别赋值为 0。

```

62     strcpy(uc->filename, filename);
63     uc->prot = up;
64     uc->flags = flags;
65     uc->max_packet_size = 0; // default: stream file

```

接着调用相应协议的文件打开函数实质打开文件。如果文件打开错误，就需要释放 malloc 出来的内存，并返回错误码。

```

66     err = up->url_open(uc, filename, flags);
67     if (err < 0)
68     {
69         av_free(uc); // 打开失败，释放刚刚分配的内存。
70         *puc = NULL;
71     }
72     *puc = uc;
73     return 0;
74 fail:
75     *puc = NULL;

```

```

77  return err;
78 }
79
简单的中转读操作到底层协议的读函数，完成读操作。
80 int url_read(URLContext *h, unsigned char *buf, int size)
81 {
82     int ret;
83     if (h->flags & URL_WRONLY)
84         return AVERROR_IO;
85     ret = h->prot->url_read(h, buf, size);
86     return ret;
87 }
88
简单的中转 seek 操作到底层协议的 seek 函数，完成 seek 操作。
89 offset_t url_seek(URLContext *h, offset_t pos, int whence)
90 {
91     offset_t ret;
92
93     if (!h->prot->url_seek)
94         return -EPIPE;
95     ret = h->prot->url_seek(h, pos, whence);
96     return ret;
97 }
98
简单的中转关闭操作到底层协议的关闭函数，完成关闭操作，并释放放在 url_open() 函数中 malloc 出
来的内存。
99 int url_close(URLContext *h)
100 {
101     int ret;
102
103     ret = h->prot->url_close(h);
104     av_free(h);
105     return ret;
106 }
107
取最大数据包大小，如果非 0，必须是实质有效的。
108 int url_get_max_packet_size(URLContext *h)
109 {
110     return h->max_packet_size;
111 }

```

## 8 aviobuf.c 文件

### 8.1 功能描述

有缓存的广义文件 ByteIOContext 相关的文件操作，比如 open, read, close, seek 等等。

### 8.2 文件注释

```

1 #include "../berrno.h"
2 #include "avformat.h"
3 #include "avio.h"
4 #include <stdarg.h>
5
6 #define IO_BUFFER_SIZE 32768
7

```

初始化广义文件 ByteIOContext 结构，一些简单的赋值操作。

```

8 int init_put_byte(ByteIOContext *s,
9         unsigned char *buffer,
10        int buffer_size,
11        int write_flag,
12        void *opaque,
13        int(*read_buf)(void *opaque, uint8_t *buf, int buf_size),
14        int(*write_buf)(void *opaque, uint8_t *buf, int buf_size),
15        offset_t(*seek)(void *opaque, offset_t offset, int whence))
16 {
17     s->buffer = buffer;
18     s->buffer_size = buffer_size;
19     s->buf_ptr = buffer;
20     s->write_flag = write_flag;
21     if (!s->write_flag)
22         s->buf_end = buffer; // 初始情况下，缓存中没有有效数据，所以 buf_end 指向缓存首地址。
23     else
24         s->buf_end = buffer + buffer_size;
25     s->opaque = opaque;
26     s->write_buf = write_buf;
27     s->read_buf = read_buf;
28     s->seek = seek;
29     s->pos = 0;
30     s->must_flush = 0;
31     s->eof_reached = 0;
32     s->error = 0;
33
34     s->max_packet_size = 0;
35     return 0;
36 }

```

36

广义文件 ByteIOContext 的 seek 操作。

输入变量: s 为广义文件句柄, offset 为偏移量, whence 为定位方式。输出变量: 相对广义文件开始的偏移量。

37 offset\_t url\_fseek(ByteIOContext \*s, offset\_t offset, int whence)

38 {

39 offset\_t offset1;

40

只支持 SEEK\_CUR 和 SEEK\_SET 定位方式, 不支持 SEEK\_END 方式。

SEEK\_CUR:从文件当前读写位置为基准偏移 offset 字节。 SEEK\_SET:从文件开始位置偏移 offset 字节。

41 if (whence != SEEK\_CUR && whence != SEEK\_SET)

42 return -EINVAL;

43

ffplay 把 SEEK\_CUR 和 SEEK\_SET 统一成 SEEK\_SET 方式处理, 所以如果是 SEEK\_CUR 方式就要转换成 SEEK\_SET 的偏移量。

offset1 = s->pos - (s->buf\_end - s->buffer) + (s->buf\_ptr - s->buffer) 算式关系请参照 3.6

节的示意图, 表示广义文件的当前实际偏移。

44 if (whence == SEEK\_CUR)

45 {

46 offset1 = s->pos - (s->buf\_end - s->buffer) + (s->buf\_ptr - s->buffer);

47 if (offset == 0)

48 return offset1; // 如果偏移量为 0, 返回实际偏移位置。

计算绝对偏移量, 赋值给 offset。

49 offset += offset1; // 加上实际偏移量, 统一成相对广义文件开始的绝对偏移量

50 }

计算绝对偏移量相对当前缓存的偏移量, 赋值给 offset1。

51 offset1 = offset - (s->pos - (s->buf\_end - s->buffer));

判断绝对偏移量是否在当前缓存中, 如果在当前缓存中, 就简单的修改 buf\_ptr 指针。

52 if (offset1 >= 0 && offset1 <= (s->buf\_end - s->buffer))

53 {

54 s->buf\_ptr = s->buffer + offset1; // can do the seek inside the buffer

55 }

56 else

57 {

判断当前广义文件是否可以 seek, 如果不能 seek 就返回错误。

58 if (!s->seek)

59 return -EPIPE;

调用底层具体的文件系统的 seek 函数完成实际的 seek 操作, 此时缓存需重新初始化, buf\_end 重新指向缓存首地址, 并修改 pos 变量为广义文件当前实际偏移量。

60 s->buf\_ptr = s->buffer;

61 s->buf\_end = s->buffer;

62 if (s->seek(s->opaque, offset, SEEK\_SET) == (offset\_t) - EPIPE)

63 return -EPIPE;

```
64 s->pos = offset;
65 }
66 s->eof_reached = 0;
67
68 return offset;
69 }
70
71 void url_fskip(ByteIOContext *s, offset_t offset)
72 {
73 url_fseek(s, offset, SEEK_CUR);
74 }
75
76 offset_t url_ftell(ByteIOContext *s)
77 {
78 return url_fseek(s, 0, SEEK_CUR);
79 }
80
81 offset_t url_fsize(ByteIOContext *s)
82 {
```

返回广义文件当前的实际偏移量。

广义文件 ByteIOContext 的当前实际偏移量再偏移 offset 字节，调用 url\_fseek 实现。

返回广义文件 ByteIOContext 的当前实际偏移量。

返回广义文件 ByteIOContext 的大小。

```

83 offset_t size;
84
判断当前广义文件 ByteIOContext 是否能 seek, 如果不能就返回错误
85 if (!s->seek)
86     return -EPIPE;
调用底层的 seek 函数取得文件大小。
87 size = s->seek(s->opaque, -1, SEEK_END) + 1;
注意 seek 操作改变了读指针, 所以要重新 seek 到当前读指针位置。
88 s->seek(s->opaque, s->pos, SEEK_SET);
89 return size;
90 }
91
判断当前广义文件 ByteIOContext 是否到末尾
92 int url_feof(ByteIOContext *s)
93 {
94     return s->eof_reached;
95 }
96
返回当前广义文件 ByteIOContext 操作错误码
97 int url_ferror(ByteIOContext *s)
98 {
99     return s->error;
100 }
101
102 // Input stream
103
填充广义文件 ByteIOContext 内部的数据缓存区。
104 static void fill_buffer(ByteIOContext *s)
105 {
106     int len;
107
如果到了广义文件 ByteIOContext 末尾就直接返回。
108     if (s->eof_reached)
109         return;
110
调用底层文件系统的读函数实际读数据填到缓存, 注意这里经过了好几次跳转才到底层读函数。首先
跳转的 url_read_buf() 函数, 再跳转到 url_read(), 再跳转到实际文件协议的读函数完成读操作。
111     len = s->read_buf(s->opaque, s->buffer, s->buffer_size); // url_read_buf //
112     if (len <= 0)
113     {
如果是到达文件末尾就不要改 buffer 参数, 这样不用重新读数据就可以做 seek back 操作。
114     s->eof_reached = 1;
115

```

设置错误码，便于分析定位。

```
116 if (len < 0)
117 s->error = len;
118 }
119 else
120 {
```

如果正确读取，修改一下基本参数。参加 3.6 节中的示意图。

---

```
121     s->pos += len;
122     s->buf_ptr = s->buffer;
123     s->buf_end = s->buffer + len;
124 }
125 }
```

126

从广义文件 ByteIOContext 中读取一个字节。

```
127 int get_byte(ByteIOContext *s)
128 {
```

129 if (s->buf\_ptr < s->buf\_end)

130 {

如果广义文件 ByteIOContext 内部缓存有数据，就修改读指针，返回读取的数据。

```
131 return *s->buf_ptr++;
132 }
133 else
134 {
```

如果广义文件 ByteIOContext 内部缓存没有数据，就先填充内部缓存。

135 fill\_buffer(s);

如果广义文件 ByteIOContext 内部缓存有数据，就修改读指针，返回读取的数据。如果没有数据就是到了文件末尾，返回 0。

NOTE: return 0 if EOF, so you cannot use it if EOF handling is necessary

```
136 if (s->buf_ptr < s->buf_end)
137 return *s->buf_ptr++;
138 else
139 return 0;
140 }
```

141 }

142

从广义文件 ByteIOContext 中以小端方式读取两个字节,实现代码充分复用 get\_byte()函数。

```
143 unsigned int get_le16(ByteIOContext *s)
144 {
145     unsigned int val;
146     val = get_byte(s);
147     val |= get_byte(s) << 8;
```

```
148 return val;
149 }
150
```

从广义文件 ByteIOContext 中以小端方式读取四个字节,实现代码充分复用 get\_le16()函数。

```
151 unsigned int get_le32(ByteIOContext *s)
```

```
152 {
```

```
153 unsigned int val;
```

```
154 val = get_le16(s);
```

```
155 val |= get_le16(s) << 16;
```

```
156 return val;
```

```
157 }
```

```
158
```

```
159 #define url_write_buf NULL
```

```
160
```

简单中转读操作函数。

```
161 static int url_read_buf(void *opaque, uint8_t *buf, int buf_size)
```

```
162 {
```

```
163 URLContext *h = opaque;
```

```
164 return url_read(h, buf, buf_size);
```

```
165 }
```

```
166
```

简单中转 seek 操作函数。

```
167 static offset_t url_seek_buf(void *opaque, offset_t offset, int whence)
```

```
168 {
```

```
169 URLContext *h = opaque;
```

```
170 return url_seek(h, offset, whence);
```

```
171 }
```

```
172
```

设置并分配广义文件 ByteIOContext 内部缓存的大小。更多的应用在修改内部缓存大小场合。

```
173 int url_setbufsize(ByteIOContext *s, int buf_size) // must be called before any I/O
```

```
174 {
```

```
175 uint8_t *buffer;
```

分配广义文件 ByteIOContext 内部缓存。

```
176 buffer = av_malloc(buf_size);
```

```
177 if (!buffer)
```

```
178 return - ENOMEM;
```

```
179
```

释放掉原来广义文件 ByteIOContext 的内部缓存, 这是一个保险的操作。

```
180 av_free(s->buffer);
```

设置广义文件 ByteIOContext 内部缓存相关参数。

```
181 s->buffer = buffer;
```

```
182 s->buffer_size = buf_size;
```

```
183 s->buf_ptr = buffer;
```

```

184 if (!s->write_flag)
185 s->buf_end = buffer; // 因为此时只是分配了内存,并没有读入数据,所以 buf_end 指向首地址
186 else
187 s->buf_end = buffer + buf_size;
188 return 0;
189 }
190

```

打开广义文件 ByteIOContext

```
191 int url_fopen(ByteIOContext *s, const char *filename, int flags)
```

```
192 {
```

```
193 URLContext *h;
```

```
194 uint8_t *buffer;
```

```
195 int buffer_size, max_packet_size;
```

```
196 int err;
```

```
197

```

调用底层文件系统的 open 函数实质性打开文件

```
198 err = url_open(&h, filename, flags);
```

```
199 if (err < 0)
```

```
200 return err;
```

```
201

```

读取底层文件系统支持的最大包大小。如果非 0, 则设置为内部缓存的大小; 否则内部缓存设置为默认大小 IO\_BUFFER\_SIZE(32768 字节)。

```
202 max_packet_size = url_get_max_packet_size(h);
```

```
203 if (max_packet_size)
```

```
204 {

```

```
205 buffer_size = max_packet_size; // no need to bufferize more than one packet
```

```
206 }

```

```
207 else

```

```
208 {

```

```
209 buffer_size = IO_BUFFER_SIZE;
```

```
210 }

```

```
211

```

分配广义文件 ByteIOContext 内部缓存, 如果错误就关闭文件返回错误码。

```
212 buffer = av_malloc(buffer_size);
```

```
213 if (!buffer)

```

```
214 {

```

```
215 url_close(h);

```

```
216 return - ENOMEM;

```

```
217 }

```

```
218

```

初始化广义文件 ByteIOContext 数据结构, 如果错误就关闭文件, 释放内部缓存, 返回错误码

```
219 if (init_put_byte(s,
```

```

220     buffer,
221     buffer_size,
222     (h->flags & URL_WRONLY || h->flags & URL_RDWR),
223     h,
224     url_read_buf,
225     url_write_buf,
226     url_seek_buf) < 0)
227 {

```

```

228 url_close(h);
229 av_free(buffer);
230 return AERROR_IO;
231 }
232
保存最大包大小。
233 s->max_packet_size = max_packet_size;
234
235 return 0;
236 }
237

```

关闭广义文件 ByteIOContext，首先释放掉内部使用的缓存，再把自己的字段置 0，最后转入底层文件系统的关闭函数实质性关闭文件。

```

238 int url_fclose(ByteIOContext *s)
239 {
240     URLContext *h = s->opaque;
241
242     av_free(s->buffer);
243     memset(s, 0, sizeof(ByteIOContext));
244     return url_close(h);
245 }
246

```

广义文件 ByteIOContext 读操作，注意此函数从 get\_buffer 改名而来，更贴切函数功能，也为了完备广义文件操作函数集。

```
247 int url_fread(ByteIOContext *s, unsigned char *buf, int size) // get_buffer
```

```

248 {
249     int len, size1;
250

```

考虑到 size 可能比缓存中的数据大得多，此时就多次读缓存，所以用 size1 保存要读取的总字节数，size 意义变更为还需要读取的字节数。

```
251     size1 = size;
```

如果还需要读的字节数大于 0，就进入循环继续读。

```
252     while (size1 > 0)
```

253 {  
计算当次循环应该读取的字节数 len , 首先设置 len 为内部缓存数据长度, 再和需要读的字节数 size 比, 有条件修正 len 的值。

254 len = s->buf\_end - s->buf\_ptr;

255 if (len > size)

256 len = size;

257 if (len == 0)

258 {

如果内部缓存没有数据。

259 if (size > s->buffer\_size) // 读操作是否绕过内部缓存的判别条件

260 {

如果要读取的数据量比内部缓存数据量大, 就调用底层函数读取数据绕过内部缓存直接到目标缓存。

261 len = s->read\_buf(s->opaque, buf, size);

262 if (len <= 0)

263 {

如果底层文件系统读错误, 设置文件末尾标记和错误码, 跳出循环, 返回实际读到的字节数。

264 s->eof\_reached = 1;

265 if (len < 0)

266 s->error = len;

267 break;

268 }

269 else

270 {

如果底层文件系统正确读, 修改相关参数, 进入下一轮循环。特别注意此处读文件绕过了内部缓存。

---

271                   s->pos += len;

272                   size -= len;

273                   buf += len; // 因为绕过了内部缓存, 特别注意此处的修改

274                   s->buf\_ptr = s->buffer;

275                   s->buf\_end = s->buffer /\* +len \*/; //因为绕过了内部缓存,特别注意此处

276        }

277        }

278   else

279   {

如果要读取的数据量比内部缓存数据量小, 就调用底层函数读取数据到内部缓存, 判断读成果否。

280 fill\_buffer(s),

281 len = s->buf\_end - s->buf\_ptr;

---

282                   if (len == 0)

283                   break;

284        }

285   }

```

286     else
287     {
如果内部缓存有数据，就拷贝 len 长度的数据到缓存区，并修改相关参数，进入下一个循环的条件判断。
288     memcpy(buf, s->buf_ptr, len);
289     buf += len;
290     s->buf_ptr += len;
291     size -= len;
292 }
293 }
返回实际读取的字节数。
294 return size1 - size;
295 }

```

## 9 utils\_format.c 文件

### 9.1 功能描述

识别文件格式和媒体格式部分使用的一些工具类函数。

### 9.2 文件注释

```

1 #include "../berrno.h"
2 #include "avformat.h"
3 #include <assert.h>
4
5 #define UINT_MAX    (0xffffffff)
6
7 #define PROBE_BUF_MIN 2048
8 #define PROBE_BUF_MAX 131072
9
10 AVInputFormat *first_iformat = NULL;
11

```

注册文件容器格式。ffplay 把所有支持的文件容器格式用链表串联起来，表头是 first\_iformat。

```

12 void av_register_input_format(AVInputFormat *format)
13 {
14     AVInputFormat **p;
15     p = &first_iformat;

```

循环移动节点指针到最后一个文件容器格式。

```

16     while (*p != NULL)
17         p = &(*p)->next;

```

直接挂接要注册的文件容器格式。

```

18     *p = format;
19     format->next = NULL;
20 }
21

```

比较文件的扩展名来识别文件类型。

```
22 int match_ext(const char *filename, const char *extensions)
```

```
23 {
```

```
24     const char *ext,    *p;
```

```
25     char ext1[32], *q;
```

```
26
```

如果输入文件为空就直接返回。

```
27     if (!filename)
```

```
28         return 0;
```

```
29
```

用'.'号作为扩展名分割符，在文件名中找扩展名分割符。

```
30     ext = strrchr(filename, '.');
```

```
31     if (ext)
```

```
32     {
```

```
33         ext++;

```

```
34         p = extensions;
```

```
35         for (;;) {
```

```
36             {
```

文件名中可能有多个标点符号，取两个标点符号间或一个标点和一个结束符间的字符串和扩展名比较 来判断文件类型，所以可能要多次比较，所以这里有一个循环。

```
37             q = ext1;
```

定位下一个标点符号或字符串结束符，把这之间的字符串拷贝到扩展名字符数组中。

```
38     while (*p != '\0' &&      *p != '.' && q - ext1 < sizeof(ext1) - 1)
```

```
39         *q++ = *p++;
```

添加扩展名字符串结束标记 0。

```
40         *q = '\0';
```

比较识别的扩展名是否后给定的扩展名相同，如果相同就返回 1，否则继续。

```
41     if (!strcasecmp(ext1, ext))
```

```
42         return 1;
```

判断是否到了文件名末尾，如果是就返回，否则进入下一个循环

```
43     if (*p == '\0')
```

```
44         break;
```

```
45     p++;

```

```
46 }
```

```
47 }
```

如果在前面的循环中没有匹配到扩展名，就是不识别的文件类型，返回 0

```
48     return 0;
```

```
49 }
```

```
50
```

探测输入的文件容器格式，返回识别出来的文件格式。如果没有识别出来，就返回初始值 NULL。

```
51     AVInputFormat *av_probe_input_format(AVProbeData *pd, int is_opened)
```

```
52 {
```

```
53 AVInputFormat *fmt1, *fmt;
```

```
54 int score, score_max;
```

```
55
```

```
56 fmt = NULL;
```

score, score\_max 可以理解识别文件容器格式的正确级别。文件容器格式识别结果，如果完全正确可以设定为 100，如果可能正确可以设定为 50，没识别出来设定为 0。识别方法不同导致等级不同。

```
57 score_max = 0;
```

```
58 for (fmt1 = first_iformat; fmt1 != NULL; fmt1 = fmt1->next)
```

```
59 {
```

```
60 if (!is_opened)
```

```
61 continue;
```

```
62
```

```
63 score = 0;
```

```
64 if (fmt1->read_probe)
```

```
65 {
```

读取文件头，判断文件头的内容来识别文件容器格式，这种识别方法非常可靠，设定 score 为 100。

```
66 score = fmt1->read_probe(pd);
```

```
67 }
```

```
68 else if (fmt1->extensions)
```

```
69 {
```

通过文件扩展名来识别文件容器格式，因为文件扩展名任何人都可以改，如果改变扩展名，这种方法就错误，如果不改变扩展名，这种识别方法有点可靠，综合等级为 50。

```
70     if (match_ext(pd->filename, fmt1->extensions))
```

```
71         score = 50;
```

```
72     }
```

如果识别出来的等级大于最大要求的等级，就认为正确识别，相关参数赋值后，进下一个循环，最后 返回最高级别的文件容器格式。

```
73 if (score > score_max)
```

```
74 {
```

```
75     score_max = score;
```

```
76     fmt = fmt1;
```

```
77 }
```

```
78 }
```

返回文件容器格式，如果没有识别出来，返回的是初始值 NULL。

```
79 return fmt;
```

```
80 }
```

```
81
```

打开输入流，其中 AVFormatParameters \*ap 参数在瘦身后的 ffplay 中没有用到，保留为了不改变接口。

```
82 int av_open_input_stream(AVFormatContext **ic_ptr, ByteIOContext *pb, const char *filename,
```

```
83 AVInputFormat *fmt, AVFormatParameters *ap)
```

```
84 {
```

```

85 int err;
86 AVFormatContext *ic;
87 AVFormatParameters default_ap;
88
89 if (!ap)
90 {
91 ap = &default_ap;
92 memset(ap, 0, sizeof(default_ap));
93 }
94
95 ic = av_mallocz(sizeof(AVFormatContext));
96 if (!ic)
97 {
98 err = AERROR_NOMEM;
99 goto fail;
100 }
101 ic->iformat = fmt;
102 if (pb)
103 ic->pb = *pb;
104
105 if (fmt->priv_data_size > 0)
106 {
107分配 priv_data 指向的内存。
108 ic->priv_data = av_mallocz(fmt->priv_data_size);
109 if (lic->priv_data)
110 {
111 err = AERROR_NOMEM;
112 goto fail;
113 }
114 else
115 {
116 ic->priv_data = NULL;
117 }
118
119读取文件头，识别媒体流格式。
120 err = ic->iformat->read_header(ic, ap);
121 if (err < 0)
122 goto fail;

```

122

123 \*ic\_ptr = ic;

124 return 0;

125

简单常规的错误处理。

126 fail:

127 if (ic)

128 av\_freep(&amp;ic-&gt;priv\_data);

129

130 av\_free(ic);

131 \*ic\_ptr = NULL;

132 return err;

133 }

134

打开输入文件，并识别文件格式，然后调用函数识别媒体流格式。

135 int av\_open\_input\_file(AVFormatContext \*\*ic\_ptr, const char \*filename, AVInputFormat \*fmt,

136 int buf\_size, AVFormatParameters \*ap)

137 {

138 int err, must\_open\_file, file\_opened, probe\_size;

139 AVProbeData probe\_data, \*pd = &amp;probe\_data;

140 ByteIOContext pb1, \*pb = &amp;pb1;

141

142 file\_opened = 0;

143 pd-&gt;filename = "";

144 if (filename)

145 pd-&gt;filename = filename;

146 pd-&gt;buf = NULL;

147 pd-&gt;buf\_size = 0;

148

149 must\_open\_file = 1;

150

151 if (!fmt || must\_open\_file)

152 {

打开输入文件，关联 ByteIOContext，经过跳转几次后才实质调用文件系统 open() 函数实质打开文件。

153 if (url\_fopen(pb, filename, URL\_RDONLY) &lt; 0)

154 {

155 err = AVERROR\_IO;

156 goto fail;

157 }

158 file\_opened = 1;

如果程序指定 ByteIOContext 内部使用的缓存大小，就重新设置内部缓存大小。通常不指定大小。

159 if (buf\_size &gt; 0)

160 url\_setbufsize(pb, buf\_size);

161

先读 PROBE\_BUF\_MIN(2048)字节文件开始数据识别文件格式，如果不能识别文件格式，就把识别文件缓存以 2 倍的增长扩大再读文件开始数据识别，直到识别出文件格式或者超过 131072 字节缓存。

162 for (probe\_size = PROBE\_BUF\_MIN; probe\_size <= PROBE\_BUF\_MAX && !fmt; probe\_size <= 1)

163 {

重新分配缓存，重新读文件开始数据。

164 pd->buf = av\_realloc(pd->buf, probe\_size);

165 pd->buf\_size = url\_fread(pb, pd->buf, probe\_size);

把文件读指针 seek 到文件开始处，便于下一次读。

166 if (url\_fseek(pb, 0, SEEK\_SET) == (offset\_t) - EPIPE)

167 {

如果 seek 错误，关闭文件，再重新打开。

168 url\_fclose(pb);

169 if (url\_fopen(pb, filename, URL\_RDONLY) < 0)

```

170 {
171     file_opened = 0;
172     err = AERROR_IO;
173     goto fail;
174 }
175 }
```

重新识别文件格式，因为一次比一次数据多，数据少的时候可能识别不出，数据多了可能就可以了。

```

177     fmt = av_probe_input_format(pd, 1);
178 }
```

```

179     av_freep(&pd->buf);
180 }
```

181

```

182     if (!fmt)
```

```

183 {
```

```

184     err = AERROR_NOFMT;
185     goto fail;
186 }
```

187

识别出文件格式后，调用函数识别流 av\_open\_input\_stream 格式。

```

188     err = av_open_input_stream(ic_ptr, pb, filename, fmt, ap);
```

```

189     if (err)
```

```

190     goto fail;
```

```

191     return 0;
```

192

```

193 fail:
```

简单的异常错误处理。

```

194     av_freep(&pd->buf);
```

```

195     if (file_opened)
```

```

196         url_fclose(pb);
```

```

197     *ic_ptr = NULL;
```

```

198     return err;
```

```

199 }
```

200

一次读取一个数据包，在瘦身后的 ffplay 中，一次读取一个完整的数据帧，数据包。

```

201     int av_read_packet(AVFormatContext *s, AVPacket *pkt)
```

```

202 {
```

```

203     return s->i_format->read_packet(s, pkt);
```

```

204 }
```

205

添加索引到索引表。有些媒体文件为便于 seek，有音视频数据帧有索引，ffplay 把这些索引以时间排序放到一个数据中。返回值添加项的索引。

## 《FFmpeg 基础库编程开发》

```
206 int av_add_index_entry(AVStream *st, int64_t pos, int64_t timestamp, int size, int distance, int flags)
```

```
207 {
```

```
208 AVIndexEntry *entries, *ie;
```

```
209 int index;
```

```
210
```

索引项越界判断，如果占有内存达到 `UINT_MAX` 时，返回。

```
211 if ((unsigned)st->nb_index_entries + 1 >= UINT_MAX / sizeof(AVIndexEntry))
```

```
212 return -1;
```

```
213
```

重新分配索引内存。注意 `av_fast_realloc()` 函数并不是每次调用就一定会重新分配内存，那样效率就太低了。

```
214 entries = av_fast_realloc(st->index_entries, &st->index_entries_allocated_size,
```

```
215 (st->nb_index_entries + 1) * sizeof(AVIndexEntry));
```

```
216 if (!entries)
```

```
217 return -1;
```

```
218
```

保持重新分配内存后，索引的首地址。

```
219 st->index_entries = entries;
```

```
220
```

以时间为顺序查找当前索引应该插在索引表的位置。

```
221 index = av_index_search_timestamp(st, timestamp, AVSEEK_FLAG_ANY);
```

```
222
```

```
223 if (index < 0)
```

```
224 {
```

续补，既接着最后一个插入，索引计算加 1，取得索引项指针，便于后面赋值操作。

```
225 index = st->nb_index_entries++;
```

```
226 ie = &entries[index];
```

```
227 assert(index == 0 || ie[-1].timestamp < timestamp);
```

```
228 }
```

```
229 else
```

```
230 {
```

中插，既插入索引表的中间，取得索引项指针，便于后面赋值操作。

```
231 ie = &entries[index];
```

```
232 if (ie->timestamp != timestamp)
```

```
233 {
```

```
234 if (ie->timestamp <= timestamp)
```

```
235 return -1;
```

```
236
```

把索引项后面的项全部后移一项，空出当前索引项。

```
237 memmove(entries + index + 1, entries + index,
```

```
238 sizeof(AVIndexEntry)*(st->nb_index_entries - index));
```

```
239
```

索引项计数加 1。

```
240 st->nb_index_entries++;
```

```
241 }
```

```
242 }
```

```
243
```

修改索引项参数，完成排序添加。

```
244 ie->pos = pos;
```

```
245 ie->timestamp = timestamp;
```

```
246 ie->size = size;
```

```
247 ie->flags = flags;
```

```
248
```

返回索引。

```
249 return index;
```

```
250 }
```

```
251
```

以时间为关键字查找当前索引应排在索引表中的位置。

```
252 int av_index_search_timestamp(AVStream *st, int64_t wanted_timestamp, int flags)
```

```
253 {
```

```
254 AVIndexEntry *entries = st->index_entries;
```

```
255 int nb_entries = st->nb_index_entries;
```

```
256 int a, b, m;
```

```
257 int64_t timestamp;
```

```
258
```

```
259 a = -1;
```

```
260 b = nb_entries;
```

```
261
```

以时间为关键字折半查找位置，请仔细理解。

```
262 while (b - a > 1)
```

```
263 {
```

```
264 m = (a + b) >> 1;
```

```
265 timestamp = entries[m].timestamp;
```

```
266 if (timestamp >= wanted_timestamp)
```

```
267 b = m;
```

```
268 if (timestamp <= wanted_timestamp)
```

```
269 a = m;
```

```
270 }
```

```
271
```

```
272 m = (flags & AVSEEK_FLAG_BACKWARD) ? a : b;
```

```
273
```

```
274 if (!(flags & AVSEEK_FLAG_ANY))
```

```
275 {
```

Seek 时，找关键帧，从关键帧开始解码，注意有些帧解码但不显示。

```
276 while (m >= 0 && m < nb_entries && !(entries[m].flags & AVINDEX_KEYFRAME))
```

```
277 {
```

```
278 m += (flags & AVSEEK_FLAG_BACKWARD) ? -1 : 1;  
279 }  
280 }  
281  
282 if (m == nb_entries)  
283 return -1;
```

284  
返回找到的位置。

```
285 return m;  
286 }
```

287  
关闭输入媒体文件，一大堆的关闭释放操作。

```
288 void av_close_input_file(AVFormatContext *s)
```

```
289 {  
290 int i;  
291 AVStream *st;  
292  
293 if (s->iformat->read_close)  
294 s->iformat->read_close(s);  
295  
296 for (i = 0; i < s->nb_streams; i++)
```

```
297 {  
298 st = s->streams[i];  
299 av_free(st->index_entries);  
300 av_free(st->actx);  
301 av_free(st);  
302 }
```

303

```
304 url_fclose(&s->pb);  
305  
306 av_freep(&s->priv_data);  
307 av_free(s);
```

```
308 }  
309  
new 一个新的媒体流，返回 AVStream 指针
```

```
310 AVStream *av_new_stream(AVFormatContext *s, int id)
```

```
311 {  
312 AVStream *st;
```

313  
判断媒体流的数目是否超限，如果超过就丢弃当前流返回 NULL。

```
314 if (s->nb_streams >= MAX_STREAMS)  
315 return NULL;
```

```
316
```

分配一块 AVStream 内存。

317 st = av\_mallocz(sizeof(AVStream));

318 if (!st)

319 return NULL;

320

通过 avcodec\_alloc\_context 分配一块 AVFormatContext 内存，并关联到 AVStream。

321 st->actx = avcodec\_alloc\_context();

322

关联 AVFormatContext 和 AVStream。

323 s->streams[s->nb\_streams++] = st;

324 return st;

325 }

326

设置计算 pts 时钟的相关参数。

327 void av\_set\_pts\_info(AVStream \*s, int pts\_wrap\_bits, int pts\_num, int pts\_den)

328 {

329 s->time\_base.num = pts\_num;

330 s->time\_base.den = pts\_den;

331 }

## 10 avidec.c 文件

### 10.1 功能描述

AVI 文件解析的相关函数，注意有些地方有些技巧性代码。

注意 1：AVI 文件容器媒体数据有两种存放方式，非交织存放和交织存放。交织存放就是音视频数据以帧为最小连续单位，相互间隔存放，这样音视频帧互相交织在一起，并且存放的间隔没有特别规定；非交织存放就是把单一媒体的所有数据帧连续存放在一起，非交织存放的 avi 文件很少。

注意 2：AVI 文件索引结构 AVIINDEXENTRY 中的 dwChunkOffset 字段指示的偏移有的是相对文件开始字节的偏移，有的事相对文件数据块 chunk 的偏移。

注意 3：附带的 avi 测试文件是交织存放的。

### 10.2 文件注释

```

1 #include "avformat.h"
2
3 #include <assert.h>
4

```

几个简单的宏定义。

```

5 #define AVIIF_INDEX 0x10
6
7 #define AVIF_HASINDEX 0x00000010 // Index at end of file?
8 #define AVIF_MUSTUSEINDEX 0x00000020
9
10 #define INT_MAX 2147483647
11
12 #define MKTAG(a,b,c,d) (a | (b << 8) | (c << 16) | (d << 24))
13
14 #define FFMIN(a,b) ((a) > (b) ? (b) : (a))
15 #define FFMAX(a,b) ((a) > (b) ? (a) : (b))
16

```

```

17 static int avi_load_index(AVFormatContext *s);
18 static int guess_ni_flag(AVFormatContext *s);
19

```

AVI 文件中的流参数定义，和 AVStream 数据结构协作。

```

20 typedef struct AVIStream
21 {
22     int64_t frame_offset; // 帧偏移，视频用帧计数，音频用字节计数，用于计算 pts 表示时间
23     int remaining; // 表示需要读的数据大小，初值是帧裸数组大小，全部读完后为 0。
24     int packet_size; // 包大小，非交织和帧裸数据大小相同，交织比帧裸数据大小大 8 字节。
25
26     int scale;
27     int rate;
28     int sample_size; // size of one sample (or packet) (in the rate/scale sense) in bytes

```

```

29
30 int64_t cum_len; // temporary storage (used during seek)
31
32 int prefix;// normally 'd'<<8 + 'c' or 'w'<<8 + 'b'
33 int prefix_count;
34 } AVIStream;
35

```

AVI 文件中的文件格式参数相关定义，和 AVFormatContext 协作。

```

36 typedef struct
37 {
38     int64_t riff_end;    // RIFF 块大小
39     int64_t movi_list;  // 媒体数据块开始字节相对文件开始字节的偏移
40     int64_t movi_end;   // 媒体数据块结束字节相对文件开始字节的偏移
41     int non_interleaved;// 指示是否是非交织 AVI
42     int stream_index_2; // 为了和 AVPacket 中的 stream_index 相区别加一个后缀。
// 指示当前应该读取的流的索引。初值为-1，表示没有确定应该读的流。
// 实际表示 AVFormatContext 结构中 AVStream *streams[] 数组中的索引。
43 } AVIContext;
44

```

CodecTag 数据结构，用于关联具体媒体格式的 ID 和 Tag 标签。

```

45 typedef struct CodecTag
46 {
47     int id;    // ID 号码
48     unsigned int tag; // 标签
49 } CodecTag;
50

```

瘦身后的 ffplay 支持的一些视频媒体 ID 和 Tag 标签数组。

```

51 const CodecTag codec bmp_tags[] =
52 {
53     {CODEC_ID_MSRLE, MKTAG('m', 'r', 'l', 'e')},
54     {CODEC_ID_MSRLE, MKTAG(0x1, 0x0, 0x0, 0x0)},
55     {CODEC_ID_NONE, 0},
56 };
57

```

瘦身后的 ffplay 支持的一些音频媒体 ID 和 Tag 标签数组。

```

58 const CodecTag codec wav_tags[] =
59 {
60     {CODEC_ID_TRUESPEECH, 0x22},
61     {0, 0},
62 };
63

```

以媒体 tag 标签为关键字，查找 codec bmp\_tags 或 codec wav\_tags 数组，返回媒体 ID。

```

64 enum CodecID codec_get_id(const CodecTag *tags, unsigned int tag)

```

```
65 {
66 while (tags->id != CODEC_ID_NONE)
67 {
```

比较 Tag 关键字，相等时返回对应媒体 ID。

```
68 if (toupper((tag >> 0) & 0xFF) == toupper((tags->tag >> 0) & 0xFF)
69 && toupper((tag >> 8) & 0xFF) == toupper((tags->tag >> 8) & 0xFF)
70 && toupper((tag >> 16) & 0xFF) == toupper((tags->tag >> 16) & 0xFF)
71 && toupper((tag >> 24) & 0xFF) == toupper((tags->tag >> 24) & 0xFF))
72 return tags->id;
```

```
73 }
```

比较 Tag 关键字，不等移到数组的下一项。

```
74 tags++;
75 }
```

所有关键字都不匹配，返回 CODEC\_ID\_NONE。

```
76 return CODEC_ID_NONE;
77 }
```

```
78 }
```

校验 AVI 文件，读取 AVI 文件媒体数据块的偏移大小信息，和 avi\_probe()函数部分相同。

```
79 static int get_riff(AVIContext *avi, ByteIOContext *pb)
80 {
81 uint32_t tag;
82 tag = get_le32(pb);
83 }
```

校验 AVI 文件开始关键字串"RIFF"。

```
84 if (tag != MKTAG('R', 'I', 'F', 'F'))
85 return -1;
86
87 avi->riff_end = get_le32(pb); // RIFF chunk size
88 avi->riff_end += url_ftell(pb); // RIFF chunk end
89 tag = get_le32(pb);
90 }
```

校验 AVI 文件关键字串"AVI"或"AVIX"。

```
91 if (tag != MKTAG('A', 'V', 'I', ' ') && tag != MKTAG('A', 'V', 'T', 'X'))
92 return -1;
93 }
```

如果通过 AVI 文件关键字串"RIFF"和"AVI "或"AVIX"校验，就认为是 AVI 文件，这种方式非常可靠。

```
94 return 0;
95 }
96 }
```

排序建立 AVI 索引表，函数名为 clean\_index, 不准确，功能以具体的实现代码为准。

```
97 static void clean_index(AVFormatContext *s)
98 {
99 int i, j;
```

```
100
101 for (i = 0; i < s->nb_streams; i++)
102 {
```

对每个流都建一个独立的索引表。

```
103 AVStream *st = s->streams[i];
104 AVIStream *ast = st->priv_data;
105 int n = st->nb_index_entries;
106 int max = ast->sample_size;
107 int64_t pos, size, ts;
```

```
108
```

如果索引表项大于 1，则认为索引表已建好，不再排序重建。如果 sample\_size 为 0，则没办法重建。

```
109 if (n != 1 || ast->sample_size == 0)
```

```
110 continue;
```

```
111
```

此种情况多半是用在非交织存储的 avi 音频流。不管交织还是非交织存储，视频流通常都有索引。

防止包太小需要太多的索引项占有大量内存，设定最小帧 size 阈值为 1024。比如有些音频流，最小解码帧只十多个字节，如果文件比较大则在索引上耗费太多内存。

```
112 while (max < 1024)
```

```
113 max += max;
```

```
114
```

取位置，大小，时钟等基本参数。

```
115 pos = st->index_entries[0].pos;
116 size = st->index_entries[0].size;
117 ts = st->index_entries[0].timestamp;
```

```
118
```

```
119 for (j = 0; j < size; j += max)
```

```
120 {
```

以 max 指定的字节打包成帧，添加到索引表。

```
121 av_add_index_entry(st, pos + j, ts + j / ast->sample_size, FFMIN(max, size - j), 0, AVINDEX_KEYFRAME);
```

```
122 }
```

```
123 }
```

```
124 }
```

```
125
```

读取 AVI 文件头，读取 AVI 文件索引，并识别具体的媒体格式，关联一些数据结构。

```
126 static int avi_read_header(AVFormatContext *s, AVFormatParameters *ap)
```

```
127 {
```

```
128 AVIContext *avi = s->priv_data;
```

```
129 ByteIOContext *pb = &s->pb;
```

```
130 uint32_t tag, tag1, handler;
```

```
131 int codec_type, stream_index, frame_period, bit_rate;
```

```
132 unsigned int size, nb_frames;
```

```
133 int i, n;
```

```
134 AVStream *st;
```

135 AVIStream \*ast;

136

当前应该读取的流的索引赋初值为-1，表示没有确定应该读的流。

137 avi->stream\_index\_2 = -1;

138

校验 AVI 文件，读取 AVI 文件媒体数据块的偏移大小信息。

139 if (get\_riff(avi, pb) < 0)

140 return -1;

141

简单变量符初值。

142 stream\_index = -1; // first list tag

143 codec\_type = -1;

144 frame\_period = 0;

145

146 for (;;) {

147 }

AVI 文件的基本结构是块，一个文件有多个块，并且块还可以内嵌，在这里循环读文件头中的块。

148 if (url\_feof(pb))

149 goto fail;

150

读取每个块的标签和大小。

151 tag = get\_le32(pb);

152 size = get\_le32(pb);

153

154 switch (tag)

155 {

156 case MKTAG('L', 'T', 'S', 'T'): // ignored, except when start of video packets

157 tag1 = get\_le32(pb);

158 if (tag1 == MKTAG('m', 'o', 'v', 'i'))

159 {

读取 movi 媒体数据块的偏移和大小。

160 avi->movi\_list = url\_ftell(pb) - 4;

161 if (size)

162 avi->movi\_end = avi->movi\_list + size;

163 else

164 avi->movi\_end = url\_fsize(pb);

165

AVI 文件头后面是 movi 媒体数据块，所以到了 movi 块，文件头肯定读完，需要跳出循环。

166 goto end\_of\_header; //

167 }

168 break;

169 case MKTAG('a', 'v', 'i', 'h'): // avi header, using frame\_period is bad idea

170 frame\_period = get\_le32(pb);

```

171 bit_rate = get_le32(pb) *8;
172 get_le32(pb);
读取 non_interleaved 的初值。
173 avi->non_interleaved |= get_le32(pb) & AVIF_MUSTUSEINDEX;
174

```

```

175 url_fskip(pb, 2 *4);
176 n = get_le32(pb);
177 for (i = 0; i < n; i++)
178 {

```

读取流数目 n 后，分配 AVStream 和 AVIStream 数据结构，在 187 行把它们关联起来。

~~特别注意 av\_new\_stream() 函数关联 AVFormatContext 和 AVStream 结构，分配关联 AVCodecContext 结构~~

```

179 AVIStream *ast;
180 st = av_new_stream(s, i);
181 if (!st)
182 goto fail;
183
184 ast = av_mallocz(sizeof(AVIStream));
185 if (!ast)
186 goto fail;
187 st->priv_data = ast;
188
189 st->actx->bit_rate = bit_rate;
190 }
191 url_fskip(pb, size - 7 * 4);
192 break;

```

```

193 case MKTAG('s', 't', 'r', 'h'): // stream header

```

~~指示当前流在 AVFormatContext 结构中 AVStream \*streams[MAX\_STREAMS] 数组中的索引。~~

```

194 stream_index++;

```

从 strh 块读取所有流共有一些信息，跳过有些不用的字段，填写需要的字段。

```

195 tag1 = get_le32(pb);
196 handler = get_le32(pb);
197
198 if (stream_index >= s->nb_streams)
199 {

```

出现这种情况通常代表媒体文件数据有错，ffplay 简单的跳过。

---

```

200         url_fskip(pb, size - 8);
201         break;
202     }

```

```

203 st = s->streams[stream_index];
204 ast = st->priv_data;
205
206 get_le32(pb); // flags
207 get_le16(pb); // priority
208 get_le16(pb); // language
209 get_le32(pb); // initial frame
210 ast->scale = get_le32(pb);
211 ast->rate = get_le32(pb);
212 if (ast->scale && ast->rate)
213 {}
214 else if (frame_period)
215 {
216 ast->rate = 1000000;
217 ast->scale = frame_period;
218 }
219 else
220 {
221 ast->rate = 25;
222 ast->scale = 1;
223 }

```

设置当前流的时间信息，用于计算 pts 表示时间，进而同步。

```

224 av_set_pts_info(st, 64, ast->scale, ast->rate);
225
226 ast->cum_len = get_le32(pb); // start
227 nb_frames = get_le32(pb);
228
229 get_le32(pb); // buffer size
230 get_le32(pb); // quality
231 ast->sample_size = get_le32(pb); // sample sszie
232
233 switch (tag1)
234 {

```

```
235 case MKTAG('v', 'i', 'd', 's'): codec_type = CODEC_TYPE_VIDEO;
```

特别注意视频流的每一帧大小不同，所以 sample\_size 设置为 0；对比音频流每一帧大小固定的情况。

```

236 ast->sample_size = 0;
237 break;
238 case MKTAG('a', 'u', 'd', 's'): codec_type = CODEC_TYPE_AUDIO;
239 break;

```

```

240 case MKTAG('t', 'x', 't', 's'): //FIXME
241 codec_type = CODEC_TYPE_DATA; //CODEC_TYPE_SUB ? FIXME
242 break;
243 case MKTAG('p', 'a', 'd', 's'): codec_type = CODEC_TYPE_UNKNOWN;

```

## 《FFmpeg 基础库编程开发》

如果是填充流，stream\_index 减 1 就实现了简单的丢弃，不计入流数目总数。

```
244 stream_index--;
245 break;
246 default:
247 goto fail;
248 }
```

```
249 ast->frame_offset = ast->cum_len * FFMAX(ast->sample_size, 1);
250 url_fskip(pb, size - 12 * 4);
251 break;
```

```
252 case MKTAG('s', 't', 'r', 'f'): // stream header
```

从 strf 块读取流中编解码器的一些信息，跳过有些不用的字段，填写需要的字段。

注意有些编解码器需要的附加信息从此块中读出，保持至 extradata 并最终传给相应的编解码器。

```
253 if (stream_index >= s->nb_streams)
```

```
254 {
255 url_fskip(pb, size);
256 }
```

```
257 else
```

```
258 {
259 st = s->streams[stream_index];
260 switch (codec_type)
261 {
```

```
262 case CODEC_TYPE_VIDEO: // BITMAPINFOHEADER
```

```
263 get_le32(pb); // size
```

```
264 st->actx->width = get_le32(pb);
```

```
265 st->actx->height = get_le32(pb);
```

```
266 get_le16(pb); // panes
```

```
267 st->actx->bits_per_sample = get_le16(pb); // depth
```

```
268 tag1 = get_le32(pb);
```

```
269 get_le32(pb); // ImageSize
```

```
270 get_le32(pb); // XPelsPerMeter
```

```
271 get_le32(pb); // YPelsPerMeter
```

```
272 get_le32(pb); // ClrUsed
```

```
273 get_le32(pb); // ClrImportant
```

```
274
```

```
275 if (size > 10 * 4 && size < (1 << 30))
```

```
276 {
```

对视频，extradata 通常是保存的是 BITMAPINFO

```
277 st->actx->extradata_size = size - 10 * 4;
```

```
278 st->actx->extradata = av_malloc(st->actx->extradata_size +
```

```
279 FF_INPUT_BUFFER_PADDING_SIZE);
```

```
280 url_fread(pb, st->actx->extradata, st->actx->extradata_size);
```

```
281 }
```

```
282
```

```

283 if (st->actx->extradata_size &1)
284     get_byte(pb);
285
286 /* Extract palette from extradata if bpp <= 8 */
287 /* This code assumes that extradata contains only palette */
288 /* This is true for all paletted codecs implemented in ffmpeg */
289 if (st->actx->extradata_size && (st->actx->bits_per_sample <= 8))
290 {
291     int min = FFMIN(st->actx->extradata_size, AVPALETTE_SIZE);
292
293     st->actx->palcctrl = av_mallocz(sizeof(AVPaletteControl));
294     memcpy(st->actx->palcctrl->palette, st->actx->extradata, min);
295     st->actx->palcctrl->palette_changed = 1;
296 }
297
298 st->actx->codec_type = CODEC_TYPE_VIDEO;
299 st->actx->codec_id = codec_get_id(codec_bmp_tags, tag1);
300
301 st->frame_last_delay = 1.0 * ast->scale / ast->rate;
302
303 break;
304 case CODEC_TYPE_AUDIO:
305 {
306     AVCodecContext *actx = st->actx;
307
308     int id = get_le16(pb);
309     actx->codec_type = CODEC_TYPE_AUDIO;
310     actx->channels = get_le16(pb);
311     actx->sample_rate = get_le32(pb);
312     actx->bit_rate = get_le32(pb) *8;
313     actx->block_align = get_le16(pb);
314     if (size == 14) // We're dealing with plain vanilla WAVEFORMAT

```

```

315     actx->bits_per_sample = 8;
316     else
317         actx->bits_per_sample = get_le16(pb);
318     actx->codec_id = codec_get_id(codec_wav_tags, id);
319
320     if (size > 16)
321     {
322         actx->extradata_size = get_le16(pb);
323         if (actx->extradata_size > 0)

```

324 {

对音频, extradata 通常是保存的是 WAVEFORMATEX

325 if (actx-&gt;extradata\_size &gt; size - 18)

326 actx-&gt;extradata\_size = size - 18;

327 actx-&gt;extradata = av\_mallocz(actx-&gt;extradata\_size +

328 FF\_INPUT\_BUFFER\_PADDING\_SIZE);

329 url\_fread(pb, actx-&gt;extradata, actx-&gt;extradata\_size);

330 }

331 else

332 {

333 actx-&gt;extradata\_size = 0;

334 }

335

336 // It is possible for the chunk to contain garbage at the end

337 if (size - actx-&gt;extradata\_size - 18 &gt; 0)

338 url\_fskip(pb, size - actx-&gt;extradata\_size - 18);

339 }

340 }

341

342 if (size % 2) // 2-aligned (fix for Stargate SG-1 - 3x18 - Shades of

Grey.avi)

343 url\_fskip(pb, 1);

344

345 break;

346 default:

对其他流类型, ffplay 简单的设置为 data 流。常规的是音频流和视频流, 其他的少见。

347 st-&gt;actx-&gt;codec\_type = CODEC\_TYPE\_DATA;

348 st-&gt;actx-&gt;codec\_id = CODEC\_ID\_NONE;

349 url\_fskip(pb, size);

350 break;

351 }

352 }

353 break;

354 default: // skip tag

对其他不识别的块 chunk, 跳过。

355 size += (size &amp;1);

356 url\_fskip(pb, size);

357 break;

358 }

359 }

360

361 end\_of\_header:

362 if (stream\_index != s-&gt;nb\_streams - 1)

```

362 {
364 fail:
校验流的数目，如果有误，释放相关资源，返回-1 错误。
365 for (i = 0; i < s->nb_streams; i++)
366 {
367 av_freep(&s->streams[i]->actx->extradata);
368 av_freep(&s->streams[i]);
369 }
370 return -1;
371 }
372
加载 AVI 文件索引。
373 avi_load_index(s);
374
判别是否是非交织 avi。
375 avi->non_interleaved |= guess_ni_flag(s);
376 if (avi->non_interleaved)
对那些非交织存储的媒体流，人工的补上索引，便于读取操作。
377 clean_index(s);
378
379 return 0;
380 }
381
avi 文件可以简单认为音视频媒体数据时间基相同，因此音视频数据需要同步读取，同步解码，播放才能同步。
交织存储的 avi 文件，临近存储的音视频帧解码时间表示时间相近，微小的解码时间表示时间差别可以用帧缓存队列抵消，所以可以简单的按照文件顺序读取媒体数据。
非交织存储的 avi 文件，视频和音频这两种媒体数据相隔甚远，小缓存简单的顺序读文件时，不能同时读到音频和视频数据，最后导致不同步，ffplay 采取按最近时间点来决定读音频还是视频数据。
382 int avi_read_packet(AVFormatContext *s, AVPacket *pkt)
383 {
384 AVIContext *avi = s->priv_data;
385 ByteIOContext *pb = &s->pb;
386 int n, d[8], size;
387 offset_t i, sync;
388
389 if (avi->non_interleaved)
390 {
如果是非交织 AVI，用最近时间点来决定读取视频还是音频数据。
391 int best_stream_index = 0;
392 AVStream *best_st = NULL;
393 AVIStream *best_ast;
394 int64_t best_ts = INT64_MAX;

```

```

395 int i;
396
397 for (i = 0; i < s->nb_streams; i++)
398 {
遍历所有媒体流，按照已经播放的流数据，计算下一个最近的时间点。
399 AVStream *st = s->streams[i];
400 AVIStream *ast = st->priv_data;
401 int64_t ts = ast->frame_offset;
402
把帧偏移换算成帧数。
403 if (ast->sample_size)
404 ts /= ast->sample_size;
405
把帧数换算成 pts 表示时间。
406 ts = av_rescale(ts, AV_TIME_BASE *(int64_t)st->time_base.num, st->time_base.den);
407
取最小的时间点对应的时间，流指针，流索引作为要读取的最佳 (读取)流参数。
408 if (ts < best_ts) // 每次读取时间点(ast->frame_offset)最近的包
409 {
410 best_ts = ts;
411 best_st = st;
412 best_stream_index = i;
413 }
414 }

保存最佳流对应的 AVIStream，便于 432 行赋值并传递参数 packet_size 和 remaining。
415 best_ast = best_st->priv_data;
换算最小的时间点，查找索引表取出对应的索引。在缓存足够大，一次性完整读取帧数据时，此时
best_ast->remaining 参数为 0。
416 best_ts = av_rescale(best_ts,best_st->time_base.den,AV_TIME_BASE *(int64_t)best_st->time_base.num);
417 if (best_ast->remaining)
418 i = av_index_search_timestamp(best_st, best_ts, AVSEEK_FLAG_ANY | AVSEEK_FLAG_BACKWARD);
419 else
420 i = av_index_search_timestamp(best_st, best_ts, AVSEEK_FLAG_ANY);
421
422 if (i >= 0)
423 {
找到最佳索引，取出其他参数，在 426 行 seek 到相应位置，在 430 行保存最佳流索引，在 432 行保存
并传递要读取的数据大小(通过最佳流索引找到最佳流，再找到对应 AVIStream 结构，再找到数据大小)。
424 int64_t pos = best_st->index_entries[i].pos;
425 pos += best_ast->packet_size - best_ast->remaining;
426 url_fseek(&s->pb, pos + 8, SEEK_SET);
427
428 assert(best_ast->remaining <= best_ast->packet_size);

```

429

```

430 avi->stream_index_2 = best_stream_index;
431 if (!best_ast->remaining)
432 best_ast->packet_size = best_ast->remaining = best_st->index_entries[i].size;
433 }
434 }
435
436 resync:
```

437

```
438 if (avi->stream_index_2 >= 0)
```

```
439 {
```

如果找到最佳流索引，以此为根参数，取出其他参数和读取媒体数据。

```

440 AVStream *st = s->streams[avi->stream_index_2];
441 AVIStream *ast = st->priv_data;
442 int size;
443
444 if (ast->sample_size <= 1) // minorityreport.AVI block_align=1024 sample_size=1 IMA- ADPCM
445 size = INT_MAX;
446 else if (ast->sample_size < 32)
447 size = 64 * ast->sample_size;
448 else
449 size = ast->sample_size;
```

450

在缓存足够大，一次全部读取一帧媒体数据的情况下，451 行判断不成立，size 等于 ast->sample\_size

```
451 if (size > ast->remaining)
```

```
452 size = ast->remaining;
```

453

调用 av\_get\_packet() 函数实际读取媒体数据到 pkt 包中。

```
454 av_get_packet(pb, pkt, size);
```

455

修改媒体流的一些其他参数。

```
456 pkt->dts = ast->frame_offset;
```

457

```
458 if (ast->sample_size)
```

```
459 pkt->dts /= ast->sample_size;
```

460

```
461 pkt->stream_index = avi->stream_index_2;
```

462

在简单情况顺序播放时，463 行到 487 行没有什么实际意义。

```
463 if (st->actx->codec_type == CODEC_TYPE_VIDEO)
```

```
464 {
```

```
465 if (st->index_entries)
```

```
466 {
```

```

467 AVIndexEntry *e;
468 int index;
469
470 index = av_index_search_timestamp(st, pkt->dts, 0);
471 e = &st->index_entries[index];
472
473 if (index >= 0 && e->timestamp == ast->frame_offset)
474 {
475 if (e->flags & AVINDEX_KEYFRAME)
476 pkt->flags |= PKT_FLAG_KEY;
477 }
478 }
479 else
480 {

```

如果没有索引，较好的办法是把所有帧都设为关键帧。

```

481             pkt->flags |= PKT_FLAG_KEY;
482     }
483 }
484 else
485 {
486     pkt->flags |= PKT_FLAG_KEY;
487 }
488

```

修改帧偏移。

```

489 if (ast->sample_size)
490     ast->frame_offset += pkt->size;
491 else
492     ast->frame_offset++;
493

```

494 ast->remaining -= size;

495 if (!ast->remaining)

496 {

缓存足够大时，程序一定跑到这里，复位标志性参数。

```

497     avi->stream_index_2 = -1;
498     ast->packet_size = 0;
499     if (size &1)
500     {
501         get_byte(pb);
502         size++;
503     }

```

```

504 }
505
506 return size;
507 }
508
509 memset(d, -1, sizeof(int) *8);
把数组 d[8]清为-1, 为了在下面的流标记查找时不会出错。
510 for (i = sync = url_ftell(pb); !url_eof(pb); i++)
511 {
交织 avi 时顺序读取文件, 媒体数据。
512 int j;
513
514 if (i >= avi->movi_end)
515 break;
516
首先要找到流标记, 比如 00db,00dc,01w b 等。在 32bit CPU 上为存取数据方便, 把 avi 文件中的帧标记
和帧大小共 8 个字节对应赋值到 int 型数组 d[8]中, 这样每次是整数操作。
517 for (j = 0; j < 7; j++)
518 d[j] = d[j + 1];
519
520 行把整型缓存前移一个单位。520 行从文件中读一个字节补充到整型缓存, 计算包大小和流索引。
521 d[7] = get_byte(pb);
522
523 size = d[4] + (d[5] << 8) + (d[6] << 16) + (d[7] << 24);
524 if (d[2] >= '0' && d[2] <= '9' && d[3] >= '0' && d[3] <= '9')
525 {
526 n = (d[2] - '0') *10+(d[3] - '0');
527 }
528 else
529 {
530 n = 100; //invalid stream id
531 }
532
校验 size 大小, 如果偏移位置加 size 超过数据块大小就不是有效的流标记。
校验流索引, 如果<0 就不是有效的流标记。流索引从 0 开始计数, 媒体文件通常不超过 10 个流。
533 if (i + size > avi->movi_end || d[0] < 0)
534 continue;
535
536 行到 541 行代码处理诸如 junk 等需要跳过的块。
536 if ((d[0] == 'i' && d[1] == 'x' && n < s->nb_streams)
537 || (d[0] == 'J' && d[1] == 'U' && d[2] == 'N' && d[3] == 'K'))

```

```

538 {
539 url_fskip(pb, size);
540 goto resync;
541 }
542

```

计算流索引号 n。

```
543 if (d[0] >= '0' && d[0] <= '9' && d[1] >= '0' && d[1] <= '9')
```

```
544 {

```

```
545 n = (d[0] - '0') * 10 + (d[1] - '0');
```

```
546 }

```

```
547 else

```

```
548 {

```

```
549 n = 100; //invalid stream id

```

```
550 }

```

```
551

```

```
552 //parse ##dc##wb

```

```
553 if (n < s->nb_streams)

```

```
554 {

```

如果流索引号 n 比流总数小，认为有效。(我个人认为这个校验不太严格。)

```
555 AVStream *st;

```

```
556 AVIStream *ast;

```

```
557 st = s->streams[n];

```

```
558 ast = st->priv_data;

```

```
559

```

```
560 if (((ast->prefix_count < 5 || sync + 9 > i) && d[2] < 128 && d[3] < 128)

```

```
561 || d[2] * 256 + d[3] == ast->prefix)

```

```
562 {

```

if(d[2]\*256+d[3]==ast->prefix)为真表示 "db","dc","wb"等字串匹配，找到正确帧标记。

判断 d[2]<128 && d[3]<128 是因为 'd','b','c','w'等字符的 ascii 码小于 128。

判断 ast->prefix\_count<5 || sync + 9 > i，是判断单一媒体的 5 帧内或找帧标记超过 9 个字节。

563 行到 569 行是单一媒体帧边界初次识别成功和以后识别成功的简单处理，计数自增或保存标记。

```
563 if (d[2] * 256 + d[3] == ast->prefix)

```

```
564 ast->prefix_count++;

```

```
565 else

```

```
566 {

```

```
567 ast->prefix = d[2] * 256 + d[3];

```

```
568 ast->prefix_count = 0;

```

```
569 }

```

```
570

```

找到相应的流索引后，保存相关参数，跳转到实质性读媒体程序。

```
571 avi->stream_index_2 = n;

```

```
572 ast->packet_size = size + 8;

```

```
573 ast->remaining = size;

```

```

574 goto resync;
575 }
576 }
577 // palette changed chunk
578 if (d[0] >= '0' && d[0] <= '9' && d[1] >= '0' && d[1] <= '9'
579 && (d[2] == 'p' && d[3] == 'c') && n < s->nb_streams && i + size <= avi->movi_end)
580 {

```

处理调色板改变块数据，读取调色板数据到编解码器上下文的调色板数组中。

```

581 AVStream *st;
582 int first, clr, flags, k, p;
583
584 st = s->streams[n];
585
586 first = get_byte(pb);
587 clr = get_byte(pb);
588 if (!clr) // all 256 colors used
589 clr = 256;
590 flags = get_le16(pb);
591 p = 4;
592 for (k = first; k < clr + first; k++)
593 {

```

---

```

594 int r, g, b;
595 r = get_byte(pb);
596 g = get_byte(pb);
597 b = get_byte(pb);
598 get_byte(pb);
599 st->actx->palctrl->palette[k] = b + (g << 8) + (r << 16);
600 }

```

```

601 st->actx->palctrl->palette_changed = 1;
602 goto resync;
603 }
604 }
605
606 return -1;
607 }
```

实质读取 AVI 文件的索引。

```

609 static int avi_read_idx1(AVFormatContext *s, int size)
610 {
611 AVICContext *avi = s->priv_data;
612 ByteIOContext *pb = &s->pb;
613 int nb_index_entries, i;
614 AVStream *st;
```

```

615 AVIStream *ast;
616 unsigned int index, tag, flags, pos, len;
617 unsigned last_pos = -1;
618

```

619 nb\_index\_entries = size / 16;  
如果没有索引块 chunk，直接返回。

```

620 if (nb_index_entries <= 0)
621 return -1;
622

```

遍历整个索引项。

```

623 for (i = 0; i < nb_index_entries; i++)
624 {
625 tag = get_le32(pb);
626 flags = get_le32(pb);
627 pos = get_le32(pb);
628 len = get_le32(pb);
629

```

如果第一个索引指示的偏移量大于数据块的偏移量，则索引指示的偏移量是相对文件开始字节的偏移量。索引加载到内存后，如果是相对数据块的偏移量就要换算成相对于文件开始字节的偏移量，便于 seek 操作。在 631 行和 633 行统一处理这两个情况。

```
630 if (i == 0 && pos > avi->movi_list)
```

```
631 avi->movi_list = 0;
```

```
632

```

```
633 pos += avi->movi_list;
```

```
634

```

计算流 ID，如索引块中的 00dc, 01w b 等关键字表示的流 ID 分别为数字 0 和 1。

```
635 index = ((tag & 0xff) - '0') * 10;
```

```
636 index += ((tag >> 8) & 0xff) - '0';
```

```
637 if (index >= s->nb_streams)
```

```
638 continue;
```

```
639

```

```
640 st = s->streams[index];
```

```
641 ast = st->priv_data;
```

```
642

```

```
643 if (last_pos == pos)
```

```
644 avi->non_interleaved = 1;
```

```
645 else

```

```
646 av_add_index_entry(st, pos, ast->cum_len, len, 0, (flags&AVIIF_INDEX)?AVINDEX_KEYFRAME:0);
```

```
647

```

```
648 if (ast->sample_size)
```

```
649 ast->cum_len += len / ast->sample_size;
```

```
650 else

```

```
651 ast->cum_len++;
```

```

652 last_pos = pos;
653 }
654 return 0;
655 }
656

```

判断是否是非交织存放媒体数据，其中 ni 是 non\_interleaved 的缩写，非交织的意思。如果是非交织存放返回 1，交织存放返回 0。

非交织存放的 avi 文件，如果有多个媒体流，肯定有某个流的开始字节文件偏移量大于其他某个流的 末尾字节的文件偏移量。程序利用这个来判断是否是非交织存放，否则认定为交织存放。

```
657 static int guess_ni_flag(AVFormatContext *s)
```

```
658 {
```

```
659 int i;
```

```
660 int64_t last_start = 0;
```

```
661 int64_t first_end = INT64_MAX;
```

```
662

```

遍历 AVI 文件中所有的索引，取流开始偏移量的最大值和末尾偏移量的最小值判断。

```
663 for (i = 0; i < s->nb_streams; i++)
```

```
664 {

```

```
665 AVStream *st = s->streams[i];
```

```
666 int n = st->nb_index_entries;
```

```
667

```

如果某个流没有 index 项，认为这个流没有数据，这个流忽略不计。

```
668 if (n <= 0)
```

```
669 continue;
```

```
670

```

遍历 AVI 文件中所有的索引，取流开始偏移量的最大值。

```
671 if (st->index_entries[0].pos > last_start)
```

```
672 last_start = st->index_entries[0].pos;
```

```
673

```

遍历 AVI 文件中所有的索引，取流末尾偏移量的最小值。

```
674 if (st->index_entries[n - 1].pos < first_end)
```

```
675 first_end = st->index_entries[n - 1].pos;
```

```
676 }

```

如果某个流的开始最大值大于某个流的末尾最小值，认为是非交织存储，否则是交织存储。

```
677 return last_start > first_end;
```

```
678 }

```

```
679

```

加载 AVI 文件索引块 chunk，特别注意在 avi\_read\_idx1()函数调用的 av\_add\_index\_entry()函数是分媒体类型按照时间顺序重新排序的。

```
680 static int avi_load_index(AVFormatContext *s)
```

```
681 {

```

```
682 AVIContext *avi = s->priv_data;
```

```
683 ByteIOContext *pb = &s->pb;
```

## 《FFmpeg 基础库编程开发》

```
684 uint32_t tag, size;  
685 offset_t pos = url_ftell(pb);  
686  
687 url_fseek(pb, avi->movi_end, SEEK_SET);
```

```
688
689 for (;;)
690 {
691 if (url_feof(pb))
692 break;
693 tag = get_le32(pb);
694 size = get_le32(pb);
695
696 switch (tag)
697 {
698 case MKTAG('i', 'd', 'x', '1'):
699 if (avi_read_idx1(s, size) < 0)
700 goto skip;
701 else
702 goto the_end;
703 break;
704 default:
705 skip:
706 size += (size &1);
707 url_fskip(pb, size);
708 break;
709 }
710 }
711 the_end:
712 url_fseek(pb, pos, SEEK_SET);
713 return 0;
714 }
```

715  
关闭 AVI 文件，释放内存和其他相关资源。

```
716 static int avi_read_close(AVFormatContext *s)
717 {
718 int i;
719 AVIContext *avi = s->priv_data;
720
721 for (i = 0; i < s->nb_streams; i++)
722 {
723 AVStream *st = s->streams[i];
724 AVISStream *ast = st->priv_data;
725 av_free(ast);
726 av_free(st->actx->extradata);
```

```

727 av_free(st->actx->palctrl);
728 }
729
730 return 0;
731 }
732

```

AVI 文件判断，取 AVI 文件的关键字串"RIFF"和"AVI"判断，和 get\_riff()函数部分相同。

```

733 static int avi_probe(AVProbeData *p)
734 {
735     if (p->buf_size <= 32) // check file header
736         return 0;
737     if (p->buf[0] == 'R' && p->buf[1] == 'I' && p->buf[2] == 'F' && p->buf[3] == 'F'
738         && p->buf[8] == 'A' && p->buf[9] == 'V' && p->buf[10] == 'I' && p->buf[11] == ' ')
739         return AVPROBE_SCORE_MAX;
740     else
741         return 0;
742 }
743

```

初始化 AVI 文件格式 AVInputFormat 结构，直接的赋值操作。

```

744 AVInputFormat avi_iformat =
745 {
746     "avi",
747     sizeof(AVIContext),
748     avi_probe,
749     avi_read_header,
750     avi_read_packet,
751     avi_read_close,
752 };
753

```

注册 avi 文件格式，ffplay 把所有支持的文件格式用链表串联起来，表头是 first\_iformat，便于查找。

```

754 int avidec_init(void)
755 {
756     av_register_input_format(&avi_iformat);
757     return 0;
758 }
759

```

## 5.4 libswscale 视频色彩空间转换

## 5.5 libswresample 音频重采样

## 5.6 libavfilter 音视频滤器

## 5.7 libavdevice 设备输入和输出容器

## 5.8 libpostproc 视频后期处理

# 第六章 播放器

## 6.1 视频播放器

### 6.1.1 ffmpeg 库的配置

从 <http://ffmpeg.zeranoe.com/builds/> 网站上

1. 下载 Dev 版本，里面包含了 ffmpeg 的 xxx.h 头文件以及 xxx.lib 库文件；
2. 下载 Shared 版本，里面包含了 ffmpeg 的 dll 文件；
3. 将这两部分文件拷贝到 VC 工程下面就可以了。

#### FFMPEG 库移植到 VC 需要的步骤：

在 VC 下使用 FFMPEG 编译好的库，不仅仅是把.h, .lib, .dll 拷贝到工程中就行了，还需要做以下几步。（此方法适用于自己使用 MingW 编译的库，也同样适用于从网上下载的编译好的库，例如 <http://ffmpeg.zeranoe.com/builds/>）。

- (1) 像其他额外库一样，设置 VC 的 Include 路径为你 c:\msys\local\include，设置 VCLib 路径为次 c:\msys\local\bin，增加操作系统的一个 Path c:\msys\local\bin（这一步好像不是必须的）。
- (2) 将 mingw 安装目录下的 include 的 inttypes.h, stdint.h, \_mingw.h 三个文件拷到你 ffmpeg 库的目录下的 include
- (3) 在 \_mingw.h 文件的结尾处(在 #endif 一行之前)添加了一行：

```
#define __restrict__
```

- (4) 把所有 long long 改成了 \_\_int64，如果是直接在 vs2008 下编译，则这个修改应该是不需要的(这步我没有遇到)
- (5)

```
#ifdef __cplusplus
#include <stdio.h>
#include <stdlib.h>
```

```
#include "string.h"
#include "SDL/SDL.h"
//#include "windows.h"
extern "C"
{
    #include "ffmpeg/avutil.h"
    #include "ffmpeg/avcodec.h"
    #include "ffmpeg/avformat.h"
}
#endif

#pragma comment(lib,"avutil.lib")
#pragma comment(lib,"avcodec.lib")
#pragma comment(lib,"avformat.lib")

(6) 如果遇到 error C3861: 'UINT64_C': identifier not found
在 common.h 里加入定义如下:
#ifndef INT64_C
#define INT64_C(c) (c ## LL)
#define UINT64_C(c) (c ## ULL)
#endif
```

## 6.1.2 一个简单的视频播放器

该播放器虽然简单，但是几乎包含了使用 FFMPEG 播放一个视频所有必备的 API，并且使用 SDL 显示解码出来的视频。

并且支持流媒体等多种视频输入，出于简单考虑，没有音频部分，同时视频播放采用直接延时 40ms 的方式  
平台使用 VC2010

使用了最新的 FFMPEG 类库

```
int _tmain(int argc, _TCHAR* argv[])
{
    AVFormatContext *pFormatCtx;
    int             i, videoindex;
    AVCodecContext  *pCodecCtx;
    AVCodec         *pCodec;
    char            filepath[]="nwn.mp4";
    av_register_all();
    avformat_network_init();
    pFormatCtx = avformat_alloc_context();
    if(avformat_open_input(&pFormatCtx,filepath,NULL,NULL)!=0){
        printf("无法打开文件\n");
        return -1;
    }
```

## 《FFmpeg 基础库编程开发》

```
if(av_find_stream_info(pFormatCtx)<0)
{
    printf("Couldn't find stream information.\n");
    return -1;
}
videoindex=-1;
for(i=0; i<pFormatCtx->nb_streams; i++)
{
    if(pFormatCtx->streams[i]->codec->codec_type==AVMEDIA_TYPE_VIDEO)
    {
        videoindex=i;
        break;
    }
}
if(videoindex==-1)
{
    printf("Didn't find a video stream.\n");
    return -1;
}
pCodecCtx=pFormatCtx->streams[videoindex]->codec;
pCodec=avcodec_find_decoder(pCodecCtx->codec_id);
if(pCodec==NULL)
{
    printf("Codec not found.\n");
    return -1;
}
if(avcodec_open(pCodecCtx, pCodec)<0)
{
    printf("Could not open codec.\n");
    return -1;
}
AVFrame *pFrame,*pFrameYUV;
pFrame=avcodec_alloc_frame();
pFrameYUV=avcodec_alloc_frame();
uint8_t *out_buffer;
out_buffer=new uint8_t[avpicture_get_size(PIX_FMT_YUV420P, pCodecCtx->width, pCodecCtx->height)];
avpicture_fill((AVPicture *)pFrameYUV, out_buffer, PIX_FMT_YUV420P, pCodecCtx->width,
pCodecCtx->height);
//-----SDL-----
if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER)) {
    printf( "Could not initialize SDL - %s\n", SDL_GetError());
    exit(1);
}
SDL_Surface *screen;
screen = SDL_SetVideoMode(pCodecCtx->width, pCodecCtx->height, 0, 0);
```

《FFmpeg 基础库编程开发》

```

if(!screen) {   printf("SDL: could not set video mode - exiting\n");
exit(1);
}
SDL_Overlay *bmp;
bmp = SDL_CreateYUVOverlay(pCodecCtx->width, pCodecCtx->height,SDL_YV12_OVERLAY, screen);
SDL_Rect rect;
//-----
int ret, got_picture;
static struct SwsContext *img_convert_ctx;
int y_size = pCodecCtx->width * pCodecCtx->height;
AVPacket *packet=(AVPacket *)malloc(sizeof(AVPacket));
av_new_packet(packet, y_size);
//输出一下信息-----
printf("文件信息-----\n");
av_dump_format(pFormatCtx,0,filepath,0);
printf("-----\n");
//-----
while(av_read_frame(pFormatCtx, packet)>=0)
{
    if(packet->stream_index==videoindex)
    {
        ret = avcodec_decode_video2(pCodecCtx, pFrame, &got_picture, packet);
        if(ret < 0)
        {
            printf("解码错误\n");
            return -1;
        }
        if(got_picture)
        {
            img_convert_ctx = sws_getContext(pCodecCtx->width, pCodecCtx->height, pCodecCtx->pix_fmt,
pCodecCtx->width, pCodecCtx->height, PIX_FMT_YUV420P, SWS_BICUBIC, NULL, NULL, NULL);
            sws_scale(img_convert_ctx, (const uint8_t* const*)pFrame->data, pFrame->linesize, 0,
pCodecCtx->height, pFrameYUV->data, pFrameYUV->linesize);
            SDL_LockYUVOverlay(bmp);
            bmp->pixels[0]=pFrameYUV->data[0];
            bmp->pixels[2]=pFrameYUV->data[1];
            bmp->pixels[1]=pFrameYUV->data[2];
            bmp->pitches[0]=pFrameYUV->linesize[0];
            bmp->pitches[2]=pFrameYUV->linesize[1];
            bmp->pitches[1]=pFrameYUV->linesize[2];
            SDL_UnlockYUVOverlay(bmp);
            rect.x = 0;
            rect.y = 0;
        }
    }
}

```

《FFmpeg 基础库编程开发》

```

rect.w = pCodecCtx->width;
rect.h = pCodecCtx->height;
SDL_DisplayYUVOverlay(bmp, &rect);
//延时 40ms
SDL_Delay(40);
}
}
av_free_packet(packet);
}
delete[] out_buffer;
av_free(pFrameYUV);
avcodec_close(pCodecCtx);
avformat_close_input(&pFormatCtx);
return 0;
}

```

## 6.2 音频播放器

注意：

- 1.程序输出的解码后 PCM 音频数据可以使用 Audition 打开播放
- 2.m4a,aac 文件可以直接播放。mp3 文件需要调整 SDL 音频帧大小为 4608（默认是 4096），否则播放会不流畅
- 3.也可以播放视频中的音频

```

#include <stdlib.h>
#include <string.h>
extern "C"
{
#include "libavcodec/avcodec.h"
#include "libavformat/avformat.h"
//SDL
#include "sdl/SDL.h"
#include "sdl/SDL_thread.h"
};
#include "decoder.h"
//#include "wave.h"
#define _WAVE_
//全局变量-----
static Uint8 *audio_chunk;
static Uint32 audio_len;
static Uint8 *audio_pos;
//-----
/* The audio function callback takes the following parameters:
stream: A pointer to the audio buffer to be filled

```

## 《FFmpeg 基础库编程开发》

len: The length (in bytes) of the audio buffer (这是固定的 4096? )

回调函数

注意: mp3 为什么播放不顺畅?

len=4096;audio\_len=4608;两个相差 512! 为了这 512, 还得再调用一次回调函数。。。

m4a,aac 就不存在此问题(都是 4096)!

\*/

```
void fill_audio(void *udata,Uint8 *stream,int len){  
    /* Only play if we have data left */  
    if(audio_len==0)  
        return;  
    /* Mix as much data as possible */  
    len=(len>audio_len?audio_len:len);  
    SDL_MixAudio(stream,audio_pos,len,SDL_MIX_MAXVOLUME);  
    audio_pos += len;  
    audio_len -= len;  
}
```

//-----

```
int decode_audio(char* no_use)  
{  
    AVFormatContext *pFormatCtx;  
    int i, audioStream;  
    AVCodecContext *pCodecCtx;  
    AVCodec *pCodec;  
    char url[300]={0};  
    strcpy(url,no_use);  
    //Register all available file formats and codecs  
    av_register_all();  
    //支持网络流输入  
    avformat_network_init();  
    //初始化  
    pFormatCtx = avformat_alloc_context();  
    //有参数 avdic  
    //if(avformat_open_input(&pFormatCtx,url,NULL,&avdic)!=0){  
    if(avformat_open_input(&pFormatCtx,url,NULL,NULL)!=0){  
        printf("Couldn't open file.\n");  
        return -1;  
    }  
  
    // Retrieve stream information  
    if(av_find_stream_info(pFormatCtx)<0)  
    {  
        printf("Couldn't find stream information.\n");  
    }
```

```

    return -1;
}

// Dump valid information onto standard error
av_dump_format(pFormatCtx, 0, url, false);

// Find the first audio stream
audioStream=-1;
for(i=0; i < pFormatCtx->nb_streams; i++)
    //原为 codec_type==CODEC_TYPE_AUDIO
    if(pFormatCtx->streams[i]->codec->codec_type==AVMEDIA_TYPE_AUDIO)
    {
        audioStream=i;
        break;
    }
if(audioStream== -1)
{
    printf("Didn't find a audio stream.\n");
    return -1;
}

// Get a pointer to the codec context for the audio stream
pCodecCtx=pFormatCtx->streams[audioStream]->codec;
// Find the decoder for the audio stream
pCodec=avcodec_find_decoder(pCodecCtx->codec_id);
if(pCodec==NULL)
{
    printf("Codec not found.\n");
    return -1;
}

// Open codec
if(avcodec_open(pCodecCtx, pCodec)<0)
{
    printf("Could not open codec.\n");
    return -1;
}
***** For output file *****/
FILE *pFile;

#ifndef _WAVE_
pFile=fopen("output.wav", "wb");
fseek(pFile, 44, SEEK_SET); //预留文件头的位置
#else
pFile=fopen("output.pcm", "wb");
#endif

// Open the time stamp file
FILE *pTSFile;

```

## 《FFmpeg 基础库编程开发》

```
pTSFile=fopen("audio_time_stamp.txt", "wb");
if(pTSFile==NULL)
{
    printf("Could not open output file.\n");
    return -1;
}
fprintf(pTSFile, "Time Base: %d/%d\n", pCodecCtx->time_base.num, pCodecCtx->time_base.den);
/** Write audio into file *****/
//把结构体改为指针
AVPacket *packet=(AVPacket *)malloc(sizeof(AVPacket));
av_init_packet(packet);
//音频和视频解码更加统一！
//新加
AVFrame *pFrame;
pFrame=avcodec_alloc_frame();
//-----SDL-----
//初始化
if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER)) {
    printf( "Could not initialize SDL - %s\n", SDL_GetError());
    exit(1);
}
//结构体，包含 PCM 数据的相关信息
SDL_AudioSpec wanted_spec;
wanted_spec.freq = pCodecCtx->sample_rate;
wanted_spec.format = AUDIO_S16SYS;
wanted_spec.channels = pCodecCtx->channels;
wanted_spec.silence = 0;
wanted_spec.samples = 1024; //播放 AAC, M4a, 缓冲区的大小
//wanted_spec.samples = 1152; //播放 MP3, WMA 时候用
wanted_spec.callback = fill_audio;
wanted_spec userdata = pCodecCtx;
if (SDL_OpenAudio(&wanted_spec, NULL)<0)//步骤（2）打开音频设备
{
    printf("can't open audio.\n");
    return 0;
}
//-----
printf("比特率 %3d\n", pFormatCtx->bit_rate);
printf("解码器名称 %s\n", pCodecCtx->codec->long_name);
printf("time_base %d \n", pCodecCtx->time_base);
printf("声道数 %d \n", pCodecCtx->channels);
printf("sample per second %d \n", pCodecCtx->sample_rate);
//新版不再需要
```

《FFmpeg 基础库编程开发》

```

// short decompressed_audio_buf[(AVCODEC_MAX_AUDIO_FRAME_SIZE * 3) / 2];
// int decompressed_audio_buf_size;
uint32_t ret,len = 0;
int got_picture;
int index = 0;
while(av_read_frame(pFormatCtx, packet)>=0)
{
    if(packet->stream_index==audioStream)
    {
        //decompressed_audio_buf_size = (AVCODEC_MAX_AUDIO_FRAME_SIZE * 3) / 2;
        //原为 avcodec_decode_audio2
        //ret = avcodec_decode_audio4( pCodecCtx, decompressed_audio_buf,
        //    &decompressed_audio_buf_size, packet.data, packet.size );
        //改为
        ret = avcodec_decode_audio4( pCodecCtx, pFrame,
            &got_picture, packet);
        if ( ret < 0 ) // if error len = -1
        {
            printf("Error in decoding audio frame.\n");
            exit(0);
        }
        if ( got_picture > 0 )
        {
#if 1
            printf("index %3d\n", index);
            printf("pts %5d\n", packet->pts);
            printf("dts %5d\n", packet->dts);
            printf("packet_size %5d\n", packet->size);

            //printf("test %s\n", rtmp->m_inChunkSize);
#endif
        }
        //直接写入
        //注意：数据是 data 【0】，长度是 linesize 【0】
#if 1
        fwrite(pFrame->data[0], 1, pFrame->linesize[0], pFile);
        //fwrite(pFrame, 1, got_picture, pFile);
        //len+=got_picture;
        index++;
        //fprintf(pTSFile, "%4d,%5d,%8d\n", index, decompressed_audio_buf_size, packet.pts);
#endif
    }
    #if 1
    //-----

```

《FFmpeg 基础库编程开发》

```

//printf("begin....\n");
//设置音频数据缓冲,PCM 数据
audio_chunk = (UInt8*) pFrame->data[0];
//设置音频数据长度
audio_len = pFrame->linesize[0];
//audio_len = 4096;
//播放 mp3 的时候改为 audio_len = 4096
//则会比较流畅，但是声音会变调！MP3 一帧长度 4608
//使用一次回调函数（4096 字节缓冲）播放不完，所以还要使用一次回调函数，导致播放缓慢...
//设置初始播放位置
audio_pos = audio_chunk;
//回放音频数据
SDL_PauseAudio(0);
//printf("don't close, audio playing...\n");
while(audio_len>0)//等待直到音频数据播放完毕!
    SDL_Delay(1);
//-----
#endif
}

// Free the packet that was allocated by av_read_frame
//已改
av_free_packet(packet);
}

//printf("The length of PCM data is %d bytes.\n", len);
#endif _WAVE_
fseek(pFile, 0, SEEK_SET);
struct WAVE_HEADER wh;
memcpy(wh.header.RiffID, "RIFF", 4);
wh.header.RiffSize = 36 + len;
memcpy(wh.header.RiffFormat, "WAVE", 4);
memcpy(wh.format.FmtID, "fmt ", 4);
wh.format.FmtSize = 16;
wh.format.wavFormat.FormatTag = 1;
wh.format.wavFormat.Channels = pCodecCtx->channels;
wh.format.wavFormat.SamplesRate = pCodecCtx->sample_rate;
wh.format.wavFormat.BitsPerSample = 16;
calformat(wh.format.wavFormat); //Calculate AvgBytesRate and BlockAlign
memcpy(wh.data.DataID, "data", 4);
wh.data.DataSize = len;
fwrite(&wh, 1, sizeof(wh), pFile);
#endif
SDL_CloseAudio();//关闭音频设备
// Close file

```

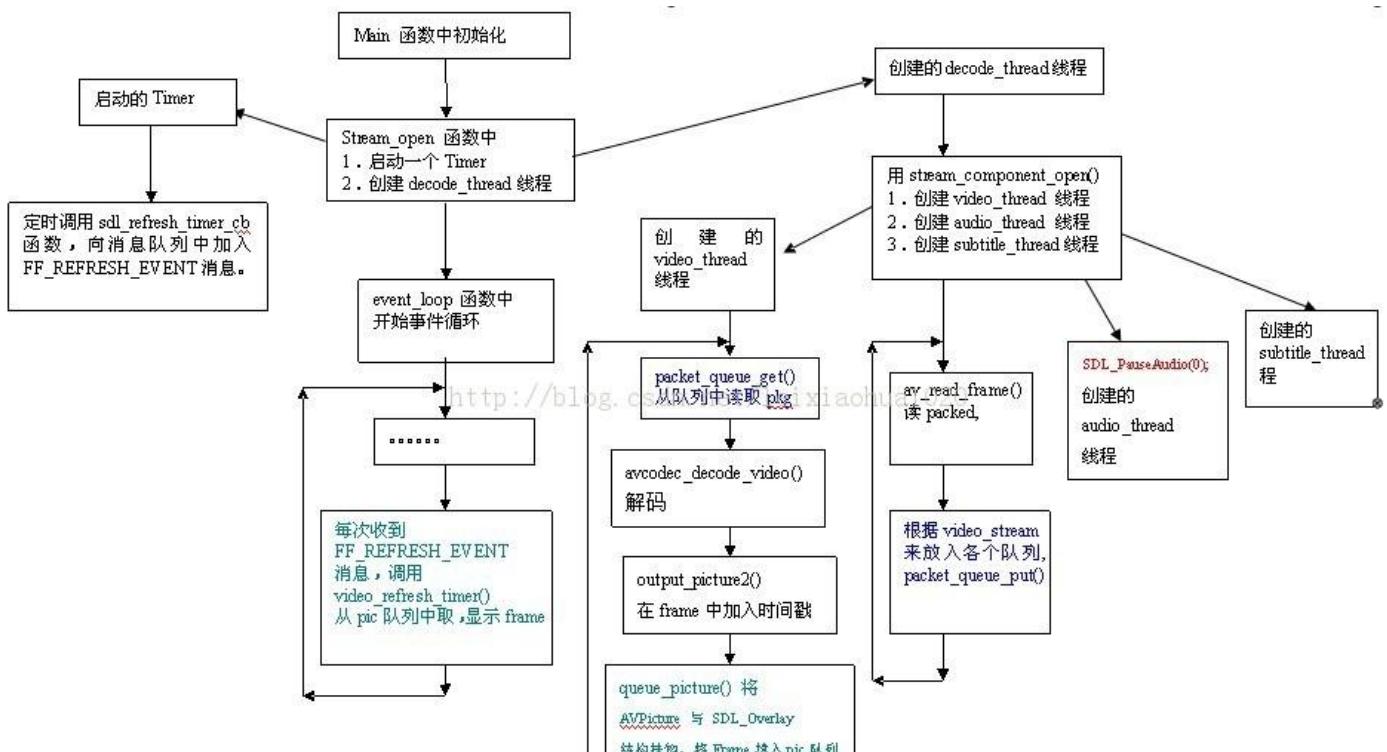
```

fclose(pFile);
// Close the codec
avcodec_close(pCodecCtx);
// Close the video file
av_close_input_file(pFormatCtx);
return 0;
}

```

## 6.3 一个完整的播放器--ffplay

### 6.3.1 ffplay 流程图



### 6.3.2 ffplay 源码剖析

#### ffplay.c 文件

##### 1 功能描述

主控文件，初始化运行环境，把各个数据结构和功能函数有机组织起来，协调数据流和功能函数，响应用户操作，启动并控制程序运行。

##### 2 文件注释

```

1 #include "./libavformat/avformat.h"
2

```

```

3 #if defined(CONFIG_WIN32)
4 #include <sys/types.h>
5 #include <sys/timeb.h>
6 #include <windows.h>
7 #else
8 #include <fcntl.h>
9 #include <sys/time.h>
10 #endif
11
12 #include <time.h>
13
14 #include <math.h>
15 #include <SDL.h>
16 #include <SDL_thread.h>
17

SDL 里面定义了 main 函数，所以在这里取消 sdl 中的 main 定义，避免重复定义。
18 #ifdef CONFIG_WIN32
19 #undef main // We don't want SDL to override our main()
20 #endif
21

导入 SDL 库。
22 #pragma comment(lib, "SDL.lib")
23

简单的几个常数定义。
24 #define FF_QUIT_EVENT (SDL_USEREVENT + 2)
25
26 #define MAX_VIDEOQ_SIZE (5 * 256 * 1024)
27 #define MAX_AUDIOQ_SIZE (5 * 16 * 1024)
28

```

```

29 #define VIDEO_PICTURE_QUEUE_SIZE 1
30
音视频数据包/数据帧队列数据结构定义，几个数据成员一看就明白，不详述。
31 typedef struct PacketQueue
32 {
33     AVPacketList *first_pkt, *last_pkt;
34     int size;
35     int abort_request;
36     SDL_mutex *mutex;
37     SDL_cond *cond;
38 } PacketQueue;
39

视频图像数据结构定义，几个数据成员一看就明白，不详述。
40 typedef struct VideoPicture
41 {
42     SDL_Overlay *bmp;
43     int width, height; // source height & width
44 } VideoPicture;
45

总控数据结构，把其他核心数据结构整合在一起，起一个中转的作用，便于在各个子结构之间跳转。
46 typedef struct VideoState
47 {
48     SDL_Thread *parse_tid;// Demux 解复用线程指针
49     SDL_Thread *video_tid;    // video 解码线程指针
50
51     int abort_request; // 异常退出请求标记
52
53     AVFormatContext *ic; // 输入文件格式上下文指针，和 iformat 配套使用
54
55     int audio_stream; // 音频流索引，表示 AVFormatContext 中 AVStream *streams[] 数组索引
56     int video_stream; // 视频流索引，表示 AVFormatContext 中 AVStream *streams[] 数组索引
57
58     AVStream *audio_st; // 音频流指针
59     AVStream *video_st; // 视频流指针
60
61     PacketQueue audioq; // 音频数据帧/数据包队列
62     PacketQueue videoq; // 视频数据帧/数据包队列

```

```

63
64 VideoPicture pictq[VIDEO_PICTURE_QUEUE_SIZE]; // 解码后视频图像队列数组
65 double frame_last_delay; // 视频帧延迟, 可简单认为是显示间隔时间
66
67 uint8_t audio_buf[(AVCODEC_MAX_AUDIO_FRAME_SIZE *3) / 2]; // 输出音频缓存
68 unsigned int audio_buf_size; // 解码后音频数据大小
69 int audio_buf_index; // 已输出音频数据大小
70 AVPacket audio_pkt; // 如果一个音频包中有多个帧, 用于保存中间状态
71 uint8_t *audio_pkt_data; // 音频包数据首地址, 配合 audio_pkt 保存中间状态
72 int audio_pkt_size; // 音频包数据大小, 配合 audio_pkt 保存中间状态
73
74 SDL_mutex *video_decoder_mutex; // 视频数据包队列同步操作而定义的互斥量指针
75 SDL_mutex *audio_decoder_mutex; // 音频数据包队列同步操作而定义的互斥量指针
76
77 char filename[240];// 媒体文件名
78
79 } VideoState;
80
81 static AVInputFormat *file_iformat;
82 static const char *input_filename;
83 static VideoState *cur_stream;
84
85 static SDL_Surface *screen;
86
取得当前时间, 以 1/1000000 秒为单位, 为便于在各个平台上移植, 由宏开关控制编译的代码。
87 int64_t av_gettime(void)
88 {
89 #if defined(CONFIG_WINCE)
90 return timeGetTime() *int64_t_C(1000);
91 #elif defined(CONFIG_WIN32)
92 struct _timeb tb;
93 _ftime(&tb);
94 return ((int64_t)tb.time *int64_t_C(1000) + (int64_t)tb.millitm) *int64_t_C(1000);
95 #else
96 struct timeval tv;
97 gettimeofday(&tv, NULL);
98 return (int64_t)tv.tv_sec *1000000+tv.tv_usec;
99#endif
100 }
101
数据帧/数据包生命周期:
1: 在 av_get_packet() 函数中调用 av_malloc() 函数分配内存, 并调用 url_fread() 填充媒体数据。

```

## 《FFmpeg 基础库编程开发》

2: 如果是视频包调用 `packet_queue_put()` 进 `is->videoq` 队列, 如果是音频包进 `is->audioq` 队列, 如果是其他包, 就调用 `av_free_packet()` 函数直接释放内存。

3: 进入队列的包, 用 `packet_queue_get()` 取出队列, 用 `av_free_packet()` 释放内存。

### 数据帧/数据包队列、链表、包关系示意图

```
102 static void packet_queue_init(PacketQueue *q) // packet queue handling
103 {
104     memset(q, 0, sizeof(PacketQueue));
105     q->mutex = SDL_CreateMutex();
106     q->cond = SDL_CreateCond();
107 }
108
```

The diagram illustrates the internal structure of a PacketQueue. It shows a central `PacketQueue` object (green box) containing three separate linked lists, each represented by an `AVPacketList` structure (AVPacketList1, AVPacketList2, AVPacketList3). Each `AVPacketList` structure contains a `next` pointer pointing to the next list in the sequence. Within each list, there is an `AVPacket` structure (AVPacket1, AVPacket2, AVPacket3), which contains `AV Data` (音视频数据). The `AVPacket` structure also contains pointers to `pts` and `data`.

刷新队列, 释放掉队列中所有动态分配的内存, 包括音视频裸数据占用的内存和 `AVPacketList` 结构占用的内存, 参考上面示意图。 `AV Data` 音视频数据 `完整帧数据`

```
109 static void packet_queue_flush(PacketQueue *q)
```

```
110 {
```

111 // 注AVPacketList是AVPacket

112 2: 每个AVPacketList都只有一个节点(AVPacket), 这一点和传统的长长的链接

起来的list有根本的不同, 不要混淆。

由于是多线程程序, 需要同步, 所以在遍历队列释放所有动态分配内存前, 加锁。

```
113     SDL_LockMutex(q->mutex);
114     for (pkt = q->first_pkt; pkt != NULL; pkt = pkt1)
115     {
116         pkt1 = pkt->next;
117         av_free_packet(&pkt->pkt); // 释放音视频数据内存
118         av_free(&pkt); // 释放 AVPacketList 结构
119     }
```

```
120     q->last_pkt = NULL;
121     q->first_pkt = NULL;
122     q->size = 0;
```

```
123     SDL_UnlockMutex(q->mutex);
124 }
```

125 释放队列占用所有资源, 首先释放掉所有动态分配的内存, 接着释放申请的互斥量和条件量。

```
126 static void packet_queue_end(PacketQueue *q)
```

```
127 {
```

```

128 packet_queue_flush(q);
129 SDL_DestroyMutex(q->mutex);
130 SDL_DestroyCond(q->cond);
131 }
132
往音视频队列中挂接音视频数据帧/数据包。
133 static int packet_queue_put(PacketQueue *q, AVPacket *pkt)
134 {
135 AVPacketList *pkt1;
136
先分配一个 AVPacketList 结构内存，接着，140 行从 AVPacket 浅复制数据，141 行链表尾置空。
137 pkt1 = av_malloc(sizeof(AVPacketList));
138 if (!pkt1)
139     return -1;
140 pkt1->pkt = *pkt;
141 pkt1->next = NULL;
142
143 SDL_LockMutex(q->mutex);
144
往队列中挂接 AVPacketList，并且在 150 行统计缓存的媒体数据大小。
145 if (!q->last_pkt)
146 q->first_pkt = pkt1;
147 else
148 q->last_pkt->next = pkt1;
149 q->last_pkt = pkt1;
150 q->size += pkt1->pkt.size;
151
设置条件量为有信号状态，如果解码线程因等待而睡眠就及时唤醒。
152 SDL_CondSignal(q->cond);
153
154 SDL_UnlockMutex(q->mutex);
155 return 0;
156 }
157
设置 异常请求退出 状态。
158 static void packet_queue_abort(PacketQueue *q)
159 {
160 SDL_LockMutex(q->mutex);
161
162 q->abort_request = 1; // 请求异常退出
163
164 SDL_CondSignal(q->cond);
165

```

```

166 SDL_UnlockMutex(q->mutex);
167 }
168
从队列中取出一帧/包数据。
169 /* return < 0 if aborted, 0 if no packet and > 0 if packet.      */
170 static int packet_queue_get(PacketQueue *q, AVPacket *pkt, int block)
171 {
172 AVPacketList *pkt1;
173 int ret;
174
175 SDL_LockMutex(q->mutex);
176
177 for(;;)
178 {
如果异常请求退出标记置位，就带错误码返回。
179 if (q->abort_request)
180 {
181 ret = -1;
182 break;
183 }
184
185 pkt1 = q->first_pkt;
186 if (pkt1)
187 {
如果队列中有数据，就取第一个数据包，在 191 行修正缓存的媒体大小，在 192 行浅复制帧/包数据
188 q->first_pkt = pkt1->next;
189 if (!q->first_pkt)
190 q->last_pkt = NULL;
191 q->size -= pkt1->pkt_size;
192 *pkt = pkt1->pkt;
释放掉 AVPacketList 结构，此结构在 packet_queue_put() 函数中动态分配(137 行代码处)。
193 av_free(pkt1);
194 ret = 1;
195 break;
196 }
197 else if (!block)
198 {
如果是非阻塞模式，没数据就直接返回 0。
199 ret = 0;
200 break;
201 }
202 else
203 {

```

## 《FFmpeg 基础库编程开发》

如果是阻塞模式，没数据就进入睡眠状态等待，packet\_queue\_put()中唤醒(152 行代码处)。

204 SDL\_CondWait(q->cond, q->mutex);

205 }

206 }

207 SDL\_UnlockMutex(q->mutex);

208 return ret;

209 }

210

分配 SDL 库需要的 Overlay 显示表面，并设置长宽属性。

211 static void alloc\_picture(void \*opaque)

212 {

213 VideoState \*is = opaque;

214 VideoPicture \*vp;

215

216 vp = &is->pictq[0];

217

218 if (vp->bmp)

219 SDL\_FreeYUVOverlay(vp->bmp);

220

221 vp->bmp = SDL\_CreateYUVOverlay(is->video\_st->actx->width,

222 is->video\_st->actx->height,

223 SDL\_YV12\_OVERLAY,

224 screen);

225

226 vp->width = is->video\_st->actx->width;

227 vp->height = is->video\_st->actx->height;

228 }

229

解码后的视频图像在等待显示间隔时间后，做颜色空间转换，调用 SDL 库显示。简单认为 cpu 耗在前面读文件，解复用，解码的时间为 0，做简单的同步处理逻辑。

230 static int queue\_picture(VideoState \*is, AVFrame \*src\_frame, double pts)

231 {

232 VideoPicture \*vp;

233 int dst\_pix\_fmt;

234 AVPicture pict;

235

236 if (is->videoq.abort\_request)

237 return -1;

238

239 vp = &is->pictq[0];

240

241 /\* if the frame is not skipped, then display it \*/

242 if (vp->bmp)

```

243 {
244     SDL_Rect rect;
245
    等待显示间隔时间，调用 Sleep() 函数简单实现。
246     if (pts)
247         Sleep((int)(is->frame_last_delay *1000));
248
249     /* get a pointer on the bitmap */
250     SDL_LockYUVOverlay(vp->bmp);
251
    设置显示图像的属性。
252     dst_pix_fmt = PIX_FMT_YUV420P;
253     pict.data[0] = vp->bmp->pixels[0];
254     pict.data[1] = vp->bmp->pixels[2];
255     pict.data[2] = vp->bmp->pixels[1];
256
257     pict.linesize[0] = vp->bmp->pitches[0];
258     pict.linesize[1] = vp->bmp->pitches[2];
259     pict.linesize[2] = vp->bmp->pitches[1];
260
    把解码后的颜色空间转换为显示颜色空间。
261     img_convert(&pict,
262     dst_pix_fmt,
263     (AVPicture*)src_frame,
264     is->video_st->actx->pix_fmt,
265     is->video_st->actx->width,
266     is->video_st->actx->height);
267
268     SDL_UnlockYUVOverlay(vp->bmp); /* update the bitmap content */
269
270     rect.x = 0;
271     rect.y = 0;
272     rect.w = is->video_st->actx->width;
273     rect.h = is->video_st->actx->height;
    实质性显示，刷屏操作。
274     SDL_DisplayYUVOverlay(vp->bmp, &rect);
275 }
276
277 }
278
    视频解码线程，主要功能是分配解码帧缓存和 SDL 显示缓存后进入解码循环(从队列中取数据帧，解
    码，计算时钟，显示)，释放视频数据帧/数据包缓存。
279 static int video_thread(void *arg)

```

```

280 {
281 VideoState *is = arg;
282 AVPacket pkt1, *pkt = &pkt1;
283 int len1, got_picture;
284 double pts = 0;
285
分配解码帧缓存
286 AVFrame *frame = av_malloc(sizeof(AVFrame));
287 memset(frame, 0, sizeof(AVFrame));
288
分配SDL显示缓存
289 alloc_picture(is);
290
291 for (;;) {
292
从队列中取数据帧/数据包
293 if (packet_queue_get(&is->videoq, pkt, 1) < 0)
294 break;
295
实质性解码
296 SDL_LockMutex(is->video_decoder_mutex);
297 len1 = avcodec_decode_video(is->video_st->actx, frame, &got_picture, pkt->data, pkt->size);
298 SDL_UnlockMutex(is->video_decoder_mutex);
299
计算同步时钟
300 if (pkt->dts != AV_NOPTS_VALUE)
301 pts = av_q2d(is->video_st->time_base) * pkt->dts;
302
303 if (got_picture)
304 {
判断得到图像，调用显示函数同步显示视频图像。
305         if (queue_picture(is, frame, pts) < 0)
306             goto the_end;
307     }
释放视频数据帧/数据包内存，此数据包内存是在 av_get_packet() 函数中调用 av_malloc() 分配的。
308 av_free_packet(pkt);
309 }
310
311 the_end:
312 av_free(frame);
313 return 0;

```

314 }

315

解码一个音频帧，返回解压的数据大小。特别注意一个音频包可能包含多个音频帧，但一次只解码一个音频帧，所以一包可能要多次才能解码完。程序首先用 while 语句判断包数据是否全部解完，如果没有就解码当前包中的帧，修改状态参数；否则，释放数据包，再从队列中取，记录初始值，再进循环。

316 /\* decode one audio frame and returns its uncompress size \*/

317 static int audio\_decode\_frame(VideoState \*is, uint8\_t \*audio\_buf, double \*pts\_ptr)

318 {

319 AVPacket \*pkt = &amp;is-&gt;audio\_pkt;

320 int len1, data\_size;

321

322 for (;;) {

323 {

324

特别注意，一个音频包可能包含多个音频帧，可能需多次解码，VideoState 用一个 AVPacket 型变量保存多次解码的中间状态。如果多次解码但不是最后一次解码，audio\_decode\_frame 直接进 while 循环。

325 while (is-&gt;audio\_pkt\_size &gt; 0)

326 {

调用解码函数解码，avcodec\_decode\_audio()函数返回解码用掉的字节数。

327 SDL\_LockMutex(is-&gt;audio\_decoder\_mutex);

328 len1 = avcodec\_decode\_audio(is-&gt;audio\_st-&gt;actx, (int16\_t\*)audio\_buf,

329 &amp;data\_size, is-&gt;audio\_pkt\_data, is-&gt;audio\_pkt\_size);

330

331 SDL\_UnlockMutex(is-&gt;audio\_decoder\_mutex);

332 if (len1 &lt; 0)

333 {

334

如果发生错误，跳过当前帧，跳出底层循环。

335 is-&gt;audio\_pkt\_size = 0;

336 break;

337 }

338

修正解码后的音频帧缓存首地址和大小。

339 is-&gt;audio\_pkt\_data += len1;

340 is-&gt;audio\_pkt\_size -= len1;

341 if (data\_size &lt;= 0)

如果没有得到解码后的数据，继续解码。可能有些帧第一次解码时只解一个帧头就返回，此时需要继续解码数据帧。

342 continue;

343

返回解码后的数据大小。

344 return data\_size;

345 }

346

程序到这里，可能是初始时 `audio_pkt` 没有赋值；或者一包已经解码完，此时需要释放包数据内存。

347 /\* free the current packet \*/

348 if (pkt-&gt;data)

349 av\_free\_packet(pkt);

350

读取下一个数据包。

351 /\* read next packet \*/

352 if (packet\_queue\_get(&amp;is-&gt;audioq, pkt, 1) &lt; 0)

353 return -1;

354

初始化数据包首地址和大小，用于一包中包含多个音频帧需多次解码的情况。

355 is-&gt;audio\_pkt\_data = pkt-&gt;data;

356 is-&gt;audio\_pkt\_size = pkt-&gt;size;

357 }

358 }

359

音频输出回调函数，每次音频输出缓存为空时，系统就调用此函数填充音频输出缓存。目前采用比较简单的同步方式，音频按照自己的节拍往前走即可，不需要 `synchronize_audio()` 函数同步处理。

360 /\* prepare a new audio buffer \*/

361 void sdl\_audio\_callback(void \*opaque, Uint8 \*stream, int len)

362 {

363 VideoState \*is = opaque;

364 int audio\_size, len1;

365 double pts = 0;

366

367 while (len &gt; 0)

368 {

369 if (is-&gt;audio\_buf\_index &gt;= is-&gt;audio\_buf\_size)

370 {

如果解码后的数据已全部输出，就进行音频解码，并在 381 行保存解码数据大小，在 383 行读索引置 0。

371 audio\_size = audio\_decode\_frame(is, is-&gt;audio\_buf, &amp;pts);

372 if (audio\_size &lt; 0)

373 {

374 /\* if error, just output silence \*/

375 is-&gt;audio\_buf\_size = 1024;

376 memset(is-&gt;audio\_buf, 0, is-&gt;audio\_buf\_size);

377 }

378 else

379 {

380 // audio\_size = synchronize\_audio(is, (int16\_t\*)is-&gt;audio\_buf, audio\_size, pts);

381 is-&gt;audio\_buf\_size = audio\_size;

382 }

```
383 is->audio_buf_index = 0;
```

```
384 }
```

385 到 391 行，拷贝适当的数据到输出缓存，并修改解码缓存的参数，进下一轮循环。

特别注意：由进下一轮循环可知，程序应填满 SDL 库给出的输出缓存。

```
385 len1 = is->audio_buf_size - is->audio_buf_index;
```

```
386 if (len1 > len)
```

```
387 len1 = len;
```

```
388 memcpy(stream, (uint8_t*)is->audio_buf + is->audio_buf_index, len1);
```

```
389 len = len1;
```

```
390 stream += len1;
```

```
391 is->audio_buf_index += len1;
```

```
392 }
```

```
393 }
```

```
394
```

打开流模块，核心功能是打开相应 codec，启动解码线程(我们把音频回调函数看做一个广义的线程)。

```
395 /* open a given stream. Return 0 if OK */
```

```
396 static int stream_component_open(VideoState *is, int stream_index) // 核心功能 open codec
```

```
397 {
```

```
398 AVFormatContext *ic = is->ic;
```

```
399 AVCodecContext *enc;
```

```
400 AVCodec *codec;
```

```
401 SDL_AudioSpec wanted_spec, spec;
```

```
402
```

```
403 if (stream_index < 0 || stream_index >= ic->nb_streams)
```

```
404 return -1;
```

```
405
```

找到从文件格式分析中得到的解码器上下文指针，便于引用其中的参数。

```
406 enc = ic->streams[stream_index]->actx;
```

```
407
```

```
408 /* prepare audio output */
```

```
409 if (enc->codec_type == CODEC_TYPE_AUDIO)
```

```
410 {
```

初始化音频输出参数，并调用 `SDL_OpenAudio()` 设置到 SDL 库。

```
411 wanted_spec.freq = enc->sample_rate;
```

```
412 wanted_spec.format = AUDIO_S16SYS;
```

```
413 /* hack for AC3. XXX: suppress that */
```

```
414 if (enc->channels > 2)
```

```
415 enc->channels = 2;
```

```
416 wanted_spec.channels = enc->channels;
```

```
417 wanted_spec.silence = 0;
```

```
418 wanted_spec.samples = 1024; //SDL_AUDIO_BUFFER_SIZE;
```

```
419 wanted_spec.callback = sdl_audio_callback; // 此处设定回调函数
```

```
420 wanted_spec.userdata = is;
```

## 《FFmpeg 基础库编程开发》

```
421 if (SDL_OpenAudio(&wanted_spec, &spec) < 0)
```

```
422 {
```

wanted\_spec 是应用程序设定给 SDL 库的音频参数，spec 是 SDL 库返回给应用程序它能支持的音频参数，通常是一致的。如果超过 SDL 支持的参数范围，会返回最相近的参数。

```
423 fprintf(stderr, "SDL_OpenAudio: %s\n", SDL_GetError());
```

```
424 return -1;
```

```
425 }
```

```
426 }
```

```
427
```

依照编解码上下文的 codec\_id，遍历编解码器链表，找到相应功能函数。

```
428 codec = avcodec_find_decoder(enc->codec_id);
```

```
429
```

核心功能之一，打开编解码器，初始化具体编解码器的运行环境。

```
430 if (!codec || avcodec_open(enc, codec) < 0)
```

```
431 return -1;
```

```
432
```

```
433 switch (enc->codec_type)
```

```
434 {
```

```
435 case CODEC_TYPE_AUDIO:
```

在 VideoState 中记录音频流参数。

```
436 is->audio_stream = stream_index;
```

```
437 is->audio_st = ic->streams[stream_index];
```

```
438 is->audio_buf_size = 0;
```

```
439 is->audio_buf_index = 0;
```

```
440
```

初始化音频队列，并在 443 行启动广义的音频解码线程。

```
441 memset(&is->audio_pkt, 0, sizeof(is->audio_pkt));
```

```
442 packet_queue_init(&is->audioq);
```

```
443 SDL_PauseAudio(0);
```

```
444 break;
```

```
445 case CODEC_TYPE_VIDEO:
```

在 VideoState 中记录视频流参数。

```
446 is->video_stream = stream_index;
```

```
447 is->video_st = ic->streams[stream_index];
```

```
448
```

```
449 is->frame_last_delay = is->video_st->frame_last_delay;
```

```
450
```

初始化视频队列，并在 452 行直接启动视频解码线程。

```
451 packet_queue_init(&is->videoq);
```

```
452 is->video_tid = SDL_CreateThread(video_thread, is);
```

```
453 break;
```

```
454 default:
```

```
455 break;
```

```
456 }
```

```
457 return 0;
```

```
458 }
```

```
459
```

关闭流模块，停止解码线程，释放队列资源。

~~通过 packet\_queue\_abort() 函数置 abort\_request 标志位，解码线程判别此标志位并安全退出线程。~~

```
460 static void stream_component_close(VideoState *is, int stream_index)
```

```
461 {
```

```
462 AVFormatContext *ic = is->ic;
```

```
463 AVCCodecContext *enc;
```

```
464
```

简单的流索引参数校验。

```
465 if (stream_index < 0 || stream_index >= ic->nb_streams)
```

```
466 return;
```

找到从文件格式分析中得到的解码器上下文指针，便于引用其中的参数。

```
467 enc = ic->streams[stream_index]->actx;
```

```
468
```

```
469 switch (enc->codec_type)
```

```
470 {
```

停止解码线程，释放队列资源。

```
471 case CODEC_TYPE_AUDIO:
```

```
472 packet_queue_abort(&is->audioq);
```

```
473 SDL_CloseAudio();
```

```
474 packet_queue_end(&is->audioq);
```

```
475 break;
```

```
476 case CODEC_TYPE_VIDEO:
```

```
477 packet_queue_abort(&is->videoq);
```

```
478 SDL_WaitThread(is->video_tid, NULL);
```

```
479 packet_queue_end(&is->videoq);
```

```
480 break;
```

```
481 default:
```

```
482 break;
```

```
483 }
```

```
484
```

释放编解码器上下文资源

```
485 avcodec_close(enc);
```

```
486 }
```

```
487
```

文件解析线程，函数名有点不名副其实。完成三大功能，直接识别文件格式和间接识别媒体格式，

打开具体的编解码器并启动解码线程，分离音视频媒体包并挂接到相应队列。

```
488 static int decode_thread(void *arg)
```

```
489 {
```

```
490 VideoState *is = arg;
```

```
491 AVFormatContext *ic;
```

```
492 int err, i, ret, video_index, audio_index;
```

```
493 AVPacket pkt1, *pkt = &pkt1;
```

```
494 AVFormatParameters params, *ap = &params;
```

```
495
```

```
496 int flags = SDL_HWSURFACE | SDL_ASYNCBLIT | SDL_HWACCEL | SDL_RESIZABLE;
```

```
497
```

498 到 502 行，初始化基本变量指示没有相应的流。

```
498 video_index = -1;
```

```
499 audio_index = -1;
```

```
500
```

```
501 is->video_stream = -1;
```

```
502 is->audio_stream = -1;
```

```
503
```

```
504 memset(ap, 0, sizeof(*ap));
```

```
505
```

调用函数直接识别文件格式，在此函数中再调用其他函数间接识别媒体格式。

```
506 err = av_open_input_file(&ic, is->filename, NULL, 0, ap);
```

```
507 if (err < 0)
```

```
508 {
```

```
509     ret = -1;
```

```
510     goto fail;
```

```
511 }
```

保存文件格式上下文，便于各数据结构间跳转。

```
512 is->ic = ic;
```

```
513
```

```
514 for (i = 0; i < ic->nb_streams; i++)
```

---

```
515 {
```

```
516     AVCodecContext *enc = ic->streams[i]->actx;
```

```
517     switch (enc->codec_type)
```

```
518 {
```

保存音视频流索引，并把显示视频参数设置到 SDL 库。

```
519 case CODEC_TYPE_AUDIO:
```

```
520 if (audio_index < 0)
```

```
521 audio_index = i;
```

```
522 break;
```

```
523 case CODEC_TYPE_VIDEO:
```

```
524 if (video_index < 0)
```

```
525 video_index = i;
```

```
526
```

```
527 screen = SDL_SetVideoMode(enc->width, enc->height, 0, flags);
```

```
528
```

```
529 SDL_WM_SetCaption("FFplay", "FFplay"); // 修改是为了适配视频大小
```

```
530
```

```
531 // schedule_refresh(is, 40);
```

```
532 break;
```

533 default:

534 break;

535 }

536 }

537

如果有音频流，就调用函数打开音频编解码器并启动音频广义解码线程。

538 if (audio\_index >= 0)

539 stream\_component\_open(is, audio\_index);

540

如果有视频流，就调用函数打开视频编解码器并启动视频解码线程。

541 if (video\_index >= 0)

542 stream\_component\_open(is, video\_index);

543

544 if (is->video\_stream < 0 && is->audio\_stream < 0)

545 {

如果既没有音频流，又没有视频流，就设置错误码返回。

546 fprintf(stderr, "%s: could not open codecs\n", is->filename);

547 ret = -1;

548 goto fail;

549 }

550

551 for (;;) {

552 {

553 if (is->abort\_request)

如果异常退出请求置位，就退出文件解析线程。

554 break;

555

556 if (is->audioq.size > MAX\_AUDIOQ\_SIZE || is->videoq.size > MAX\_VIDEOQ\_SIZE || url\_feof(&ic->pb))

557 {

如果队列满，就稍微延时一下。

558 SDL\_Delay(10); // if the queue are full, no need to read more, wait 10 ms

559 continue;

560 }

从媒体文件中完整的读取一包音视频数据。

561 ret = av\_read\_packet(ic, pkt); //av\_read\_frame(ic, pkt);

562 if (ret < 0)

563 {

564 if (url\_ferror(&ic->pb) == 0)

565 {

566           SDL\_Delay(100); // wait for user event

567           continue;

568 }

```

569         else
570             break;
571     }

```

判断包数据的类型，分别挂接到相应队列，如果是不识别的类型，就直接释放丢弃掉。

```

572 if (pkt->stream_index == is->audio_stream)
573 {
574     packet_queue_put(&is->audioq, pkt);
575 }
576 else if (pkt->stream_index == is->video_stream)
577 {
578     packet_queue_put(&is->videoq, pkt);
579 }
580 else
581 {
582     av_free_packet(pkt);
583 }
584 }
585

```

简单的延时，让后面的线程有机会把数据解码显示完。当然丢弃掉最后的一点点数据也可以。

---

```

586 while (!is->abort_request)           // wait until the end
587 {
588     SDL_Delay(100);
589 }
590
591 ret = 0;
592

```

释放掉在本线程中分配的各种资源，体现了谁申请谁释放的程序自封闭性。

```

593 fail:
594 if (is->audio_stream >= 0)
595     stream_component_close(is, is->audio_stream);
596
597 if (is->video_stream >= 0)
598     stream_component_close(is, is->video_stream);
599
600 if (is->ic)
601 {
602     av_close_input_file(is->ic);
603     is->ic = NULL;
604 }
605
606 if (ret != 0)

```

```

607 {
608     SDL_Event event;
609
610     event.type = FF_QUIT_EVENT;
611     event.user.data1 = is;
612     SDL_PushEvent(&event);
613 }
614 return 0;
615 }
616

```

打开流，这个名字也有点名不符实。主要功能是分配全局总控数据结构，初始化相关参数，启动文件解析线程。

```

617 static VideoState *stream_open(const char *filename, AVInputFormat *iformat)
618 {
619     VideoState *is;
620
621     is = av_mallocz(sizeof(VideoState));
622     if (!is)
623         return NULL;
624     strcpy(is->filename, sizeof(is->filename), filename);
625
626     is->audio_decoder_mutex = SDL_CreateMutex();
627     is->video_decoder_mutex = SDL_CreateMutex();
628
629     is->parse_tid = SDL_CreateThread(decode_thread, is);
630     if (!is->parse_tid)
631     {
632         av_free(is);
633         return NULL;
634     }
635     return is;
636 }

```

关闭流，这个名字也有点名不符实。主要功能是释放资源。

```

639 static void stream_close(VideoState *is)
640 {
641     VideoPicture *vp;
642     int i;
643
644     is->abort_request = 1;
645     SDL_WaitThread(is->parse_tid, NULL);
646
647     for (i = 0; i < VIDEO_PICTURE_QUEUE_SIZE; i++)

```

```

648 {
649 vp = &is->pictq[i];
650 if (vp->bmp)
651 {
652 SDL_FreeYUVOverlay(vp->bmp);
653 vp->bmp = NULL;
654 }
655 }
656
657 SDL_DestroyMutex(is->audio_decoder_mutex);
658 SDL_DestroyMutex(is->video_decoder_mutex);
659 }
660
程序退出时调用的函数，关闭释放一些资源。
661 void do_exit(void)
662 {
663 if (cur_stream)
664 {
665 stream_close(cur_stream);
666 cur_stream = NULL;
667 }
668
669 SDL_Quit();
670 exit(0);
671 }
672
SDL 库的消息事件循环。
673 void event_loop(void) // handle an event sent by the GUI
674 {
675 SDL_Event event;
676
677 for (;;)
678 {
679 SDL_WaitEvent(&event);
680 switch (event.type)
681 {
682 case SDL_KEYDOWN:
683 switch (event.key.keysym.sym)
684 {
685 case SDLK_ESCAPE:
686 case SDLK_q:
687 do_exit();
688 break;

```

```
689 default:  
690 break;  
691 }  
692 break;  
693 case SDL_QUIT:  
694 case FF_QUIT_EVENT:  
695 do_exit();  
696 break;  
697 default:  
698 break;  
699 }  
700 }  
701 }  
702 }
```

入口函数，初始化 SDL 库，注册 SDL 消息事件，启动文件解析线程，进入消息循环。

```
703 int main(int argc, char **argv)  
704 {  
705     int flags = SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER;  
706  
707     av_register_all();  
708  
709     input_filename = "d:/yuv/clocktxt_320.avi";  
710  
711     if (SDL_Init(flags))  
712         exit(1);  
713  
714     SDL_EventState(SDL_ACTIVEEVENT, SDL_IGNORE);  
715     SDL_EventState(SDL_MOUSEMOTION, SDL_IGNORE);  
716     SDL_EventState(SDL_SYSWMEVENT, SDL_IGNORE);  
717     SDL_EventState(SDL_USEREVENT, SDL_IGNORE);  
718  
719     cur_stream = stream_open(input_filename, file_iformat);  
720  
721     event_loop();  
722  
723     return 0;  
724 }  
725 }
```

# 第七章 应用开发

## 7.1 ffmpeg 库的使用：编码

### YUV 编码为视频：

搞视频处理的朋友肯定比较熟悉 YUV 视频序列，很多测试库提供的视频数据都是 YUV 视频序列，我们这里就用 YUV 视频序列来做视频。关于 YUV 视频序列，我就不多讲了，可以看书学习，通常的视频序列都是 YUV420 格式的。

步骤也就那几步，添加视频流，打开编码器，开辟相应的内存空间，然后就可以打开 YUV 序列逐帧写入数据了，so easy！记得最后要做好文件的关闭和内存的释放，因为 FFmpeg 是 c 风格的（不知道新版本是否是 c++ 风格的），这些工作都需要自己做好啊。过多的说明是没用的，直接上代码：

这里我补充一下，大多数的视频格式好像只支持 YUV 格式的视频帧 AVFrame，我试图直接把 RGB 的视频序列直接编码到视频这条路好像走不通，都需要把 RGB 的视频帧再转成 YUV 视频帧才行，不知道高手有没有其他高见。

```
#include <stdio.h>
#include <string.h>
```

```
extern "C"
{
#include <libavcodec\avcodec.h>
#include <libavformat\avformat.h>
#include <libswscale\swscale.h>
};

void main(int argc, char ** argv)
{
    AVFormatContext* oc;
    AVOOutputFormat* fmt;
    AVStream* video_st;
    double video_pts;
    uint8_t* video_outbuf;
    uint8_t* picture_buf;
    AVFrame* picture;
//    AVFrame* pictureRGB;
    int size;
    int ret;
    int video_outbuf_size;
```

```
FILE *fin = fopen("akiyo_qcif.yuv", "rb"); //视频源文件
```

```
const char* filename = "test.mpg";
```

## 《FFmpeg 基础库编程开发》

```
// const char* filename;
// filename = argv[1];

av_register_all();

// avcodec_init(); // 初始化 codec 库
// avcodec_register_all(); // 注册编码器

fmt = guess_format(NULL, filename, NULL);
oc = av_alloc_format_context();
oc->oformat = fmt;
snprintf(oc->filename, sizeof(oc->filename), "%s", filename);

video_st = NULL;
if (fmt->video_codec != CODEC_ID_NONE)
{
    AVCodecContext* c;
    video_st = av_new_stream(oc, 0);
    c = video_st->codec;
    c->codec_id = fmt->video_codec;
    c->codec_type = CODEC_TYPE_VIDEO;
    c->bit_rate = 400000;
    c->width = 176;
    c->height = 144;
    c->time_base.num = 1;
    c->time_base.den = 25;
    c->gop_size = 12;
    c->pix_fmt = PIX_FMT_YUV420P;
    if (c->codec_id == CODEC_ID_MPEG2VIDEO)
    {
        c->max_b_frames = 2;
    }
    if (c->codec_id == CODEC_ID_MPEG1VIDEO)
    {
        c->mb_decision = 2;
    }
    if (!strcmp(oc->oformat->name, "mp4") || !strcmp(oc->oformat->name, "mov") || !strcmp(oc->oformat->name,
"3gp"))
    {
        c->flags |= CODEC_FLAG_GLOBAL_HEADER;
    }
}
```

## 《FFmpeg 基础库编程开发》

```
if (av_set_parameters(oc, NULL)<0)
{
    return;
}

dump_format(oc, 0, filename, 1);
if (video_st)
{
    AVCodecContext* c;
    AVCodec* codec;
    c = video_st->codec;
    codec = avcodec_find_encoder(c->codec_id);
    if (!codec)
    {
        return;
    }
    if (avcodec_open(c, codec) < 0)
    {
        return;
    }
    if (!(oc->oformat->flags & AVFMT_RAWPICTURE))
    {
        video_outbuf_size = 200000;
        video_outbuf = (uint8_t*)av_malloc(video_outbuf_size);
    }
    picture = avcodec_alloc_frame();
    size = avpicture_get_size(c->pix_fmt, c->width, c->height);
    picture_buf = (uint8_t*)av_malloc(size);
    if (!picture_buf)
    {
        av_free(picture);
    }
    avpicture_fill((AVPicture*)picture, picture_buf, c->pix_fmt, c->width, c->height);
}

if (!(fmt->flags & AVFMT_NOFILE))
{
    if (url_fopen(&oc->pb, filename, URL_WRONLY) < 0)
    {
        return;
    }
}
av_write_header(oc);
```

```

for (int i=0; i<300; i++)
{
    if (video_st)
    {
        video_pts = (double)(video_st->pts.val * video_st->time_base.num / video_st->time_base.den);
    }
    else
    {
        video_pts = 0.0;
    }
    if (!video_st/* || video_pts >= 5.0*/)
    {
        break;
    }
    AVCodecContext* c;
    c = video_st->codec;
    size = c->width * c->height;

    if (fread(picture_buf, 1, size*3/2, fin) < 0)
    {
        break;
    }

    picture->data[0] = picture_buf; // 亮度
    picture->data[1] = picture_buf+ size; // 色度
    picture->data[2] = picture_buf+ size*5/4; // 色度

    // 如果是 rgb 序列, 可能需要如下代码
//     SwsContext* img_convert_ctx;
//     img_convert_ctx = sws_getContext(c->width, c->height, PIX_FMT_RGB24, c->width, c->height, c->pix_fmt,
//     SWS_BICUBIC, NULL, NULL, NULL);
//     sws_scale(img_convert_ctx, pictureRGB->data, pictureRGB->linesize, 0, c->height, picture->data,
//     picture->linesize);

    if (oc->oformat->flags & AVFMT_RAWPICTURE)
    {
        AVPacket pkt;
        av_init_packet(&pkt);
        pkt.flags |= PKT_FLAG_KEY;
        pkt.stream_index = video_st->index;
        pkt.data = (uint8_t*)picture;
        pkt.size = sizeof(AVPicture);
    }
}

```

## 《FFmpeg 基础库编程开发》

```
ret = av_write_frame(oc, &pkt);
}
else
{
    int out_size = avcodec_encode_video(c, video_outbuf, video_outbuf_size, picture);
    if (out_size > 0)
    {
        AVPacket pkt;
        av_init_packet(&pkt);
        pkt.pts = av_rescale_q(c->coded_frame->pts, c->time_base, video_st->time_base);
        if (c->coded_frame->key_frame)
        {
            pkt.flags |= PKT_FLAG_KEY;
        }
        pkt.stream_index = video_st->index;
        pkt.data = video_outbuf;
        pkt.size = out_size;
        ret = av_write_frame(oc, &pkt);
    }
}

if (video_st)
{
    avcodec_close(video_st->codec);
//    av_free(picture->data[0]);
    av_free(picture);
    av_free(video_outbuf);
//    av_free(picture_buf);
}
av_write_trailer(oc);
for (int i=0; i<oc->nb_streams; i++)
{
    av_freep(&oc->streams[i]->codec);
    av_freep(&oc->streams[i]);
}
if (!(fmt->flags & AVFMT_NOFILE))
{
    url_fclose(oc->pb);
}
av_free(oc);
}
```

# 第八章 关键函数介绍

此章有必要，因为调取的函数都是最新版的（有些函数第五章已经提及，读者可以对比）。

## 8.1 avformat\_open\_input

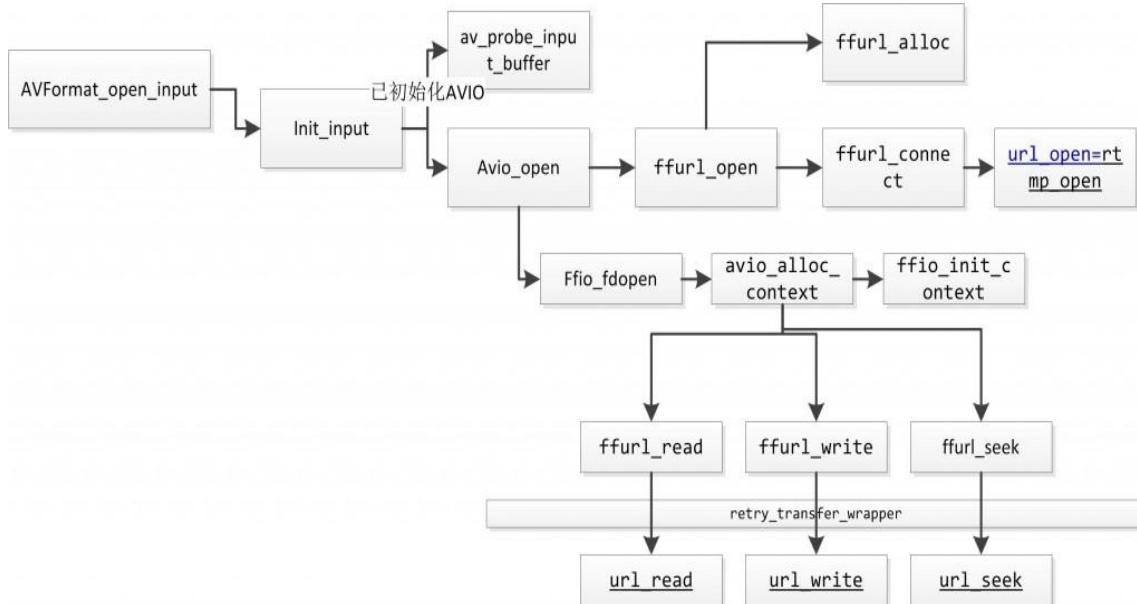
FFMPEG 打开媒体的过程开始于 `avformat_open_input`，因此该函数的重要性不可忽视。  
在该函数中，FFMPEG 完成了：

输入输出结构体 AVIOContext 的初始化；

输入数据的协议（例如 RTMP，或者 file）的识别（通过一套评分机制）：1 判断文件名的后缀 2 读取文件头的数据进行比对；

使用获得最高分的文件协议对应的 URLProtocol，通过函数指针的方式，与 FFMPEG 连接（非专业用词）；  
剩下的就是调用该 URLProtocol 的函数进行 open,read 等操作了

以下是从 eclipse+MinGW 调试 FFMPEG 源代码获得的函数调用关系图



可见最终都调用了 URLProtocol 结构体中的函数指针。

URLProtocol 结构是一大堆函数指针的集合（`avio.h` 文件）参见第四章数据结构

URLProtocol 功能就是完成各种输入协议的读写等操作

但输入协议种类繁多，它是怎样做到“大一统”的呢？

原来，每个具体的输入协议都有自己对应的 URLProtocol。

比如 file 协议（FFMPEG 把文件也当做一种特殊的协议）（`*file.c` 文件）

```

URLProtocol ff_pipe_protocol = {
    .name          = "pipe",
    .url_open      = pipe_open,
    .url_read      = file_read,
    .url_write     = file_write,
  
```

```
.url_get_file_handle = file_get_handle,  
.url_check           = file_check,  
};
```

或者 rtmp 协议（此处使用了 librtmp）(librtmp.c 文件)

```
URLProtocol ff_rtmp_protocol = {  
    .name          = "rtmp",  
    .url_open      = rtmp_open,  
    .url_read      = rtmp_read,  
    .url_write     = rtmp_write,  
    .url_close     = rtmp_close,  
    .url_read_pause = rtmp_read_pause,  
    .url_read_seek  = rtmp_read_seek,  
    .url_get_file_handle = rtmp_get_file_handle,  
    .priv_data_size = sizeof(RTMP),  
    .flags         = URL_PROTOCOL_FLAG_NETWORK,  
};
```

可见它们把各自的函数指针都赋值给了 URLProtocol 结构体的函数指针

因此 avformat\_open\_input 只需调用 url\_open,url\_read 这些函数就可以完成各种具体输入协议的 open,read 等操作了

## 8.2 avcodec\_register\_all()

ffmpeg 注册复用器，编码器等的函数 av\_register\_all()。该函数在所有基于 ffmpeg 的应用程序中几乎都是第一个被调用的。只有调用了该函数，才能使用复用器，编码器等。

可见解复用器注册都是用

REGISTER\_DEMUXER (X,x)

例如：

REGISTER\_DEMUXER (AAC, aac)

可见复用器注册都是用

REGISTER\_MUXER (X,x))

例如：

REGISTER\_MUXER (ADTS, adts)

既有解复用器又有复用器的话，可以用

REGISTER\_MUXDEMUX (X,x));

例如：

REGISTER\_MUXDEMUX (AC3, ac3);

我们来看一下宏的定义，这里以解复用器为例：

```
py  
#define REGISTER_DEMUXER(X,x) {\ \
```

## 《FFmpeg 基础库编程开发》

```
extern AVInputFormat ff_##x##_demuxer; \
if(CONFIG_##X##_DEMUXER) av_register_input_format(&ff_##x##_demuxer); }
```

注意：define 里面的##可能不太常见，它的含义就是拼接两个字符串，比如

```
#define Conn(x,y) x##y
```

那么

```
int n = Conn(123,456); 结果就是 n=123456;
```

我们以 REGISTER\_DEMUXER (AAC, aac)为例，则它等效于

py

```
extern AVInputFormat ff_aac_demuxer;
if(CONFIG_AAC_DEMUXER) av_register_input_format(&ff_aac_demuxer);
```

从上面这段代码我们可以看出，真正注册的函数是 av\_register\_input\_format(&ff\_aac\_demuxer)，那我就看看这个和函数的作用，查看一下 av\_register\_input\_format()的代码：

py

```
void av_register_input_format(AVInputFormat *format)
{
```

```
    AVInputFormat **p;
    p = &first_iformat;
    while (*p != NULL) p = &(*p)->next;
    *p = format;
    format->next = NULL;
```

}

这段代码是比较容易理解的，首先先提一点，first\_iformat 是个什么东东呢？其实它是 Input Format 链表的头部地址，是一个全局静态变量，定义如下：

py

```
/** head of registered input format linked list */
static AVInputFormat *first_iformat = NULL;
```

由此我们可以分析出 av\_register\_input\_format()的含义，一句话概括就是：遍历链表并把当前的 Input Format 加到链表的尾部。

至此 REGISTER\_DEMUXER (X, x)分析完毕。

同理，复用器道理是一样的，只是注册函数改为 av\_register\_output\_format();

既有解复用器又有复用器的话，有一个宏定义：

py

```
#define REGISTER_MUXDEMUX(X,x) REGISTER_MUXER(X,x); REGISTER_DEMUXER(X,x)
```

可见是分别注册了复用器和解复用器。

此外还有网络协议的注册，注册函数为 ffurl\_register\_protocol()，在此不再详述。

整个代码没太多可说的，首先确定是不是已经初始化过了 (initialized)，如果没有，就调用 avcodec\_register\_all()注册编解码器（这个先不分析），然后就是注册，注册，注册...直到完成所有注册。

## 8.3 av\_read\_frame()

ffmpeg 中的 av\_read\_frame()的作用是读取码流中的音频若干帧或者视频一帧。例如，解码视频的时候，每解码一个视频帧，需要先调用 av\_read\_frame()获得一帧视频的压缩数据，然后才能对该数据进行解码（例如 H.264 中一

帧压缩数据通常对应一个 NAL)。

通过 `av_read_packet(***)`, 读取一个包, 需要说明的是此函数必须是包含整数帧的, 不存在半帧的情况, 以 ts 流为例, 是读取一个完整的 PES 包 (一个完整 pes 包包含若干视频或音频 es 包), 读取完毕后, 通过 `av_parser_parse2(***)` 分析出视频一帧 (或音频若干帧), 返回, 下次进入循环的时候, 如果上次的数据没有完全取完, 则 `st = s->cur_st;` 不会是 NULL, 即再此进入 `av_parser_parse2(***)` 流程, 而不是下面的 `av_read_packet (**)` 流程, 这样就保证了, 如果读取一次包含了 N 帧视频数据 (以视频为例), 则调用 `av_read_frame (***)` N 次都不会去读数据, 而是返回第一次读取的数据, 直到全部解析完毕。

## 8.4 avcodec\_decode\_video2()

ffmpeg 中的 `avcodec_decode_video2()` 的作用是解码一帧视频数据。输入一个压缩编码的结构体 AVPacket, 输出一个解码后的结构体 AVFrame。

## 8.5 transcode\_init()

`transcode_init()` 函数是在转换前做准备工作的. 此处看一下它的真面目, 不废话, 看注释吧:

//为转换过程做准备

```
static int transcode_init(OutputFile *output_files,
    int nb_output_files,
    InputFile *input_files,
    int nb_input_files)
{
    int ret = 0, i, j, k;
    AVFormatContext *oc;
    AVCCodecContext *codec, *icodec;
    OutputStream *ost;
    InputStream *ist;
    char error[1024];
    int want_sdp = 1;

    /* init framerate emulation */
    //初始化帧率仿真(转换时是不按帧率来的,但如果要求帧率仿真,就可以做到)
    for (i = 0; i < nb_input_files; i++)
    {
        InputFile *ifile = &input_files[i];
        //如果一个输入文件被要求帧率仿真(指的是即使是转换也像播放那样按照帧率来进行),
        //则为这个文件中所有流记录下开始时间
        if (ifile->rate_emu)
            for (j = 0; j < ifile->nb_streams; j++)
                input_streams[j + ifile->ist_index].start = av_gettime();
    }
}
```

```

/* output stream init */
for (i = 0; i < nb_output_files; i++)
{
    //什么也没做,只是做了个判断而已
    oc = output_files[i].ctx;
    if (!oc->nb_streams && !(oc->oformat->flags & AVFMT_NOSTREAMS))
    {
        av_dump_format(oc, i, oc->filename, 1);
        av_log(NULL, AV_LOG_ERROR,
               "Output file #%"PRIu32" does not contain any stream\n", i);
        return AVERRORE(EINVAL);
    }
}

//轮循所有的输出流,跟据对应的输入流,设置其编解码器的参数
for (i = 0; i < nb_output_streams; i++)
{
    //轮循所有的输出流
    ost = &output_streams[i];
    //输出流对应的 FormatContext
    oc = output_files[ost->file_index].ctx;
    //取得输出流对应的输入流
    ist = &input_streams[ost->source_index];

    //attachment_filename 是不是这样的东西:一个文件,它单独容纳一个输出流?此处不懂
    if (ost->attachment_filename)
        continue;

    codec = ost->st->codec;//输出流的编解码器结构
    icodec = ist->st->codec;//输入流的编解码器结构

    //先把能复制的复制一下
    ost->st->disposition = ist->st->disposition;
    codec->bits_per_raw_sample = icodec->bits_per_raw_sample;
    codec->chroma_sample_location = icodec->chroma_sample_location;

    //如果只是复制一个流(不用解码后再编码),则把输入流的编码参数直接复制给输出流
    //此时是不需要解码也不需要编码的, 所以不需打开解码器和编码器
    if (ost->stream_copy)
    {
        //计算输出流的编解码器的 extradata 的大小,然后分配容纳 extradata 的缓冲
        //然后把输入流的编解码器的 extradata 复制到输出流的编解码器中

```

## 《FFmpeg 基础库编程开发》

```
uint64_t extra_size = (uint64_t) icodec->extradata_size
+ FF_INPUT_BUFFER_PADDING_SIZE;

if (extra_size > INT_MAX) {
    return AVERROR(EINVAL);
}

/* if stream_copy is selected, no need to decode or encode */
codec->codec_id = icodec->codec_id;
codec->codec_type = icodec->codec_type;

if (!codec->codec_tag){
    if (!oc->oformat->codec_tag
        ||av_codec_get_id(oc->oformat->codec_tag,icodec->codec_tag) == codec->codec_id
        ||av_codec_get_tag(oc->oformat->codec_tag,icodec->codec_id) <= 0)
        codec->codec_tag = icodec->codec_tag;
}

codec->bit_rate = icodec->bit_rate;
codec->rc_max_rate = icodec->rc_max_rate;
codec->rc_buffer_size = icodec->rc_buffer_size;
codec->extradata = av_mallocz(extra_size);
if (!codec->extradata){
    return AVERROR(ENOMEM);
}
memcpy(codec->extradata, icodec->extradata, icodec->extradata_size);
codec->extradata_size = icodec->extradata_size;

//重新鼓捣一下 time base(这家伙就是帧率)
codec->time_base = ist->st->time_base;
//如果输出文件是 avi,做一点特殊处理
if (!strcmp(oc->oformat->name, "avi")) {
    if (copy_tb < 0
        && av_q2d(icodec->time_base) * icodec->ticks_per_frame >
            2 * av_q2d(ist->st->time_base)
        && av_q2d(ist->st->time_base) < 1.0 / 500
        || copy_tb == 0)
    {
        codec->time_base = icodec->time_base;
        codec->time_base.num *= icodec->ticks_per_frame;
        codec->time_base.den *= 2;
    }
}
```

《FFmpeg 基础库编程开发》

```

else if (!(oc->oformat->flags & AVFMT_VARIABLE_FPS))
{
    if (copy_tb < 0
        && av_q2d(icodec->time_base) * icodec->ticks_per_frame
            > av_q2d(ist->st->time_base)
        && av_q2d(ist->st->time_base) < 1.0 / 500
        || copy_tb == 0)
    {
        codec->time_base = icodec->time_base;
        codec->time_base.num *= icodec->ticks_per_frame;
    }
}

//再修正一下帧率
av_reduce(&codec->time_base.num, &codec->time_base.den,
          codec->time_base.num, codec->time_base.den, INT_MAX);

//单独复制各不同媒体自己的编码参数
switch (codec->codec_type)
{
case AVMEDIA_TYPE_AUDIO:
    //音频的
    if (audio_volume != 256){
        av_log( NULL,AV_LOG_FATAL,
                "-acodec copy and -vol are incompatible (frames are not decoded)\n");
        exit_program(1);
    }
    codec->channel_layout = icodec->channel_layout;
    codec->sample_rate = icodec->sample_rate;
    codec->channels = icodec->channels;
    codec->frame_size = icodec->frame_size;
    codec->audio_service_type = icodec->audio_service_type;
    codec->block_align = icodec->block_align;
    break;

case AVMEDIA_TYPE_VIDEO:
    //视频的
    codec->pix_fmt = icodec->pix_fmt;
    codec->width = icodec->width;
    codec->height = icodec->height;
    codec->has_b_frames = icodec->has_b_frames;
    if (!codec->sample_aspect_ratio.num){
        codec->sample_aspect_ratio = ost->st->sample_aspect_ratio =
            ist->st->sample_aspect_ratio.num ?ist->st->sample_aspect_ratio :

```

```

ist->st->codec->sample_aspect_ratio.num ?ist->st->codec->sample_aspect_ratio :(AVRational){0, 1};
    }
    ost->st->avg_frame_rate = ist->st->avg_frame_rate;
    break;
case AVMEDIA_TYPE_SUBTITLE:
    //字幕的
    codec->width = icodec->width;
    codec->height = icodec->height;
    break;
case AVMEDIA_TYPE_DATA:
case AVMEDIA_TYPE_ATTACHMENT:
    //??的
    break;
default:
    abort();
}
}

else
{
    //如果不是复制,就麻烦多了

    //获取编码器
    if (!ost->enc)
        ost->enc = avcodec_find_encoder(ost->st->codec->codec_id);

    //因为需要转换,所以既需解码又需编码
    ist->decoding_needed = 1;
    ost->encoding_needed = 1;

    switch(codec->codec_type)
    {
case AVMEDIA_TYPE_AUDIO:
    //鼓捣音频编码器的参数,基本上是把一些不合适的参数替换掉
    ost->fifo = av_fifo_alloc(1024);//音频数据所在的缓冲
    if (!ost->fifo) {
        return AVERROR(ENOMEM);
    }

    //采样率
    if (!codec->sample_rate)
        codec->sample_rate = icodec->sample_rate;
    choose_sample_rate(ost->st, ost->enc);
}
}

```

## 《FFmpeg 基础库编程开发》

```
codec->time_base = (AVRational){1, codec->sample_rate};
```

```
//样点格式
```

```
if (codec->sample_fmt == AV_SAMPLE_FMT_NONE)
```

```
    codec->sample_fmt = icodec->sample_fmt;
```

```
choose_sample_fmt(ost->st, ost->enc);
```

```
//声道
```

```
if (ost->audio_channels_mapped) {
```

```
    /* the requested output channel is set to the number of
```

```
     * -map_channel only if no -ac are specified */
```

```
    if (!codec->channels) {
```

```
        codec->channels = ost->audio_channels_mapped;
```

```
        codec->channel_layout = av_get_default_channel_layout(codec->channels);
```

```
        if (!codec->channel_layout) {
```

```
            av_log(NULL, AV_LOG_FATAL, "Unable to find an appropriate channel layout for requested
```

```
number of channel\n");
```

```
        exit_program(1);
```

```
    }
```

```
}
```

```
/* fill unused channel mapping with -1 (which means a muted
```

```
 * channel in case the number of output channels is bigger
```

```
 * than the number of mapped channel) */
```

```
for (j = ost->audio_channels_mapped; j < FF_ARRAY_ELEMS(ost->audio_channels_map); j++)
```

```
<span>  </span>ost->audio_channels_map[j] = -1;
```

```
}else if (!codec->channels){
```

```
    codec->channels = icodec->channels;
```

```
    codec->channel_layout = icodec->channel_layout;
```

```
}
```

```
if (av_get_channel_layout_nb_channels(codec->channel_layout) != codec->channels)
```

```
    codec->channel_layout = 0;
```

```
//是否需要重采样
```

```
ost->audio_resample = codec->sample_rate != icodec->sample_rate || audio_sync_method > 1;
```

```
ost->audio_resample |= codec->sample_fmt != icodec->sample_fmt ||
```

```
    codec->channel_layout != icodec->channel_layout;
```

```
icodec->request_channels = codec->channels;
```

```
ost->resample_sample_fmt = icodec->sample_fmt;
```

```
ost->resample_sample_rate = icodec->sample_rate;
```

```
ost->resample_channels = icodec->channels;
```

```
break;
```

```
case AVMEDIA_TYPE_VIDEO:
```

```
//鼓捣视频编码器的参数,基本上是把一些不合适的参数替换掉
```

## 《FFmpeg 基础库编程开发》

```
if (codec->pix_fmt == PIX_FMT_NONE)
    codec->pix_fmt = icodec->pix_fmt;
choose_pixel_fmt(ost->st, ost->enc);
if (ost->st->codec->pix_fmt == PIX_FMT_NONE){
    av_log(NULL, AV_LOG_FATAL, "Video pixel format is unknown, stream cannot be encoded\n");
    exit_program(1);
}

//宽高
if (!codec->width || !codec->height){
    codec->width = icodec->width;
    codec->height = icodec->height;
}

//视频是否需要重采样
ost->video_resample = codec->width != icodec->width ||
                      codec->height != icodec->height ||
                      codec->pix_fmt != icodec->pix_fmt;
if (ost->video_resample){
    codec->bits_per_raw_sample= frame_bits_per_raw_sample;
}

ost->resample_height = icodec->height;
ost->resample_width = icodec->width;
ost->resample_pix_fmt = icodec->pix_fmt;

//计算帧率
if (!ost->frame_rate.num)
    ost->frame_rate = ist->st->r_frame_rate.num ?
                      ist->st->r_frame_rate : (AVRational){25,1};
if (ost->enc && ost->enc->supported_framerates && !ost->force_fps)  {
    int idx = av_find_nearest_q_idx(ost->frame_rate,ost->enc->supported_framerates);
    ost->frame_rate = ost->enc->supported_framerates[idx];
}
codec->time_base = (AVRational) {ost->frame_rate.den, ost->frame_rate.num};
if( av_q2d(codec->time_base) < 0.001 &&
    video_sync_method &&
    (video_sync_method==1 ||
     (video_sync_method<0 && !
      (oc->oformat->flags & AVFMT_VARIABLE_FPS))))
{
    av_log(oc, AV_LOG_WARNING, "Frame rate very high for a muxer not efficiently supporting it.\n"
        "Please consider specifying a lower framerate, a different muxer or -vsync 2\n");
}
```

```

    }

<span>  </span>for (j = 0; j < ost->forced_kf_count; j++)
    ost->forced_kf_pts[j] = av_rescale_q(ost->forced_kf_pts[j],
                                         AV_TIME_BASE_Q, codec->time_base);
    break;
case AVMEDIA_TYPE_SUBTITLE:
    break;
default:
    abort();
    break;
}

/* two pass mode */
if (codec->codec_id != CODEC_ID_H264 &&
    (codec->flags & (CODEC_FLAG_PASS1 | CODEC_FLAG_PASS2)))
{
    char logfilename[1024];
    FILE *f;

    snprintf(logfilename, sizeof(logfilename), "%s-%d.log",
             pass_logfilename_prefix ? pass_logfilename_prefix
             : DEFAULT_PASS_LOGFILENAME_PREFIX,
             i);
    if (codec->flags & CODEC_FLAG_PASS2){
        char *logbuffer;
        size_t logbuffer_size;
        if (cmdutils_read_file(logfilename, &logbuffer, &logbuffer_size) < 0){
            av_log(NULL, AV_LOG_FATAL,
                   "Error reading log file '%s' for pass-2 encoding\n",
                   logfilename);
            exit_program(1);
        }
        codec->stats_in = logbuffer;
    }
    if (codec->flags & CODEC_FLAG_PASS1){
        f = fopen(logfilename, "wb");
        if (!f) {
            av_log(NULL, AV_LOG_FATAL, "Cannot write log file '%s' for pass-1 encoding: %s\n",
                   logfilename, strerror(errno));
            exit_program(1);
        }
        ost->logfile = f;
    }
}

```

## 《FFmpeg 基础库编程开发》

```
}

if (codec->codec_type == AVMEDIA_TYPE_VIDEO){
    /* maximum video buffer size is 6-bytes per pixel, plus DPX header size (1664)*/
    //计算编码输出缓冲的大小,计算一个最大值
    int size = codec->width * codec->height;
    bit_buffer_size = FFMAX(bit_buffer_size, 7 * size + 10000);
}

//分配编码后数据所在的缓冲
if (!bit_buffer)
    bit_buffer = av_malloc(bit_buffer_size);
if (!bit_buffer){
    av_log(NULL, AV_LOG_ERROR,
        "Cannot allocate %d bytes output buffer\n",
        bit_buffer_size);
    return AVERRORE(NOMEM);
}

//轮循所有输出流,打开每个输出流的编码器
for (i = 0; i < nb_output_streams; i++)
{
    ost = &output_streams[i];
    if (ost->encoding_needed){
        //当然,只有在需要编码时才打开编码器
        AVCodec *codec = ost->enc;
        AVCodecContext *dec = input_streams[ost->source_index].st->codec;
        if (!codec) {
            snprintf(error, sizeof(error),
                "Encoder (codec %s) not found for output stream #%"PRIu32":%"PRIu32",
                avcodec_get_name(ost->st->codec->codec_id),
                ost->file_index, ost->index);
            ret = AVERRORE(EINVAL);
            goto dump_format;
        }
        if (dec->subtitle_header){
            ost->st->codec->subtitle_header = av_malloc(dec->subtitle_header_size);
            if (!ost->st->codec->subtitle_header){
                ret = AVERRORE(NOMEM);
                goto dump_format;
            }
            memcpy(ost->st->codec->subtitle_header,
                dec->subtitle_header, dec->subtitle_header_size);
        }
    }
}
```

## 《FFmpeg 基础库编程开发》

```
ost->st->codec->subtitle_header_size = dec->subtitle_header_size;
}

//打开啦
if (avcodec_open2(ost->st->codec, codec, &ost->opts) < 0)    {
    snprintf(error, sizeof(error),
        "Error while opening encoder for output stream #%"PRIu32":%"PRIu32" - maybe incorrect parameters such as
bit_rate, rate, width or height",
        ost->file_index, ost->index);
    ret = AVERROR(EINVAL);
    goto dump_format;
}

assert_codec_experimental(ost->st->codec, 1);
assert_avoptions(ost->opts);
if (ost->st->codec->bit_rate && ost->st->codec->bit_rate < 1000)
    av_log(NULL, AV_LOG_WARNING,
        "The bitrate parameter is set too low."
        " It takes bits/s as argument, not kbytes/s\n");
extra_size += ost->st->codec->extradata_size;

if (ost->st->codec->me_threshold)
    input_streams[ost->source_index].st->codec->debug |= FF_DEBUG_MV;
}

//初始化所有的输入流(主要做的就是在需要时打开解码器)
for (i = 0; i < nb_input_streams; i++)
    if ((ret = init_input_stream(i, output_streams, nb_output_streams,
        error, sizeof(error))) < 0)
        goto dump_format;

/* discard unused programs */
for (i = 0; i < nb_input_files; i++){
    InputFile *infile = &input_files[i];
    for (j = 0; j < infile->ctx->nb_programs; j++){
        AVProgram *p = infile->ctx->programs[j];
        int discard = AVDISCARD_ALL;

        for (k = 0; k < p->nb_stream_indexes; k++){
            if (!input_streams[infile->ist_index + p->stream_index[k]].discard){
                discard = AVDISCARD_DEFAULT;
                break;
            }
        }
    }
}
```

```

p->discard = discard;
}

}

//打开所有输出文件，写入媒体文件头
for (i = 0; i < nb_output_files; i++){
    oc = output_files[i].ctx;
    oc->interrupt_callback = int_cb;
    if (avformat_write_header(oc, &output_files[i].opts) < 0){
        snprintf(error, sizeof(error),
                 "Could not write header for output file #%"PRIu32" (incorrect codec parameters ?)",
                 i);
        ret = AVERROREINVAL;
        goto dump_format;
    }
    // assert_avoptions(output_files[i].opts);
    if (strcmp(oc->oformat->name, "rtp")){
        want_sdp = 0;
    }
}

return 0;
}

```

## 8.6 transcode()

直接从主函数进行分析

```

int main(int argc, char **argv)
{
    OptionsContext o = { 0 };
    int64_t ti;

    //与命令行分析有关的结构的初始化,下面不再罗嗦
    reset_options(&o, 0);

    //设置日志级别
    av_log_set_flags(AV_LOG_SKIP_REPEATED);
    parse_loglevel(argc, argv, options);

    if (argc > 1 && !strcmp(argv[1], "-d")) {
        run_as_daemon = 1;

```

《FFmpeg 基础库编程开发》

```

av_log_set_callback(log_callback_null);
argc--;
argv++;
}

//注册组件们
avcodec_register_all();
#if CONFIG_AVDEVICE
    avdevice_register_all();
#endif
#if CONFIG_AVFILTER
    avfilter_register_all();
#endif
av_register_all();
//初始化网络,windows 下需要
avformat_network_init();

show_banner();

term_init();

//分析命令行输入的参数们
parse_options(&o, argc, argv, options, opt_output_file);

//文件的转换就在此函数中发生
if (transcode(output_files, nb_output_files, input_files, nb_input_files) < 0)
    exit_program(1);

exit_program(0);
return 0;
}

下面是 transcode() 函数, 转换就发生在它里面. 不废话, 看注释吧, 应很详细了
static int transcode(
    OutputFile *output_files,//输出文件数组
    int nb_output_files,//输出文件的数量
    InputFile *input_files,//输入文件数组
    int nb_input_files)//输入文件的数量
{
    int ret, i;
    AVFormatContext *is, *os;
    OutputStream *ost;
    InputStream *ist;
    uint8_t *no_packet;
}

```

## 《FFmpeg 基础库编程开发》

```
int no_packet_count = 0;
int64_t timer_start;
int key;

if (!(no_packet = av_mallocz(nb_input_files)))
    exit_program(1);

//设置编码参数,打开所有输出流的编码器,打开所有输入流的解码器,写入所有输出文件的文件头,于是准备好了
ret = transcode_init(output_files, nb_output_files, input_files, nb_input_files);
if (ret < 0)
    goto fail;

if (!using_stdin){
    av_log(NULL, AV_LOG_INFO, "Press [q] to stop, [?] for help\n");
}

timer_start = av_gettime();

//循环,直到收到系统信号才退出
for (; received_sigterm == 0;
{
    int file_index, ist_index;
    AVPacket pkt;
    int64_t ipts_min;
    double opts_min;
    int64_t cur_time = av_gettime();

    ipts_min = INT64_MAX;
    opts_min = 1e100;
    /* if 'q' pressed, exits */
    if (!using_stdin)
    {
        //先查看用户按下了什么键,跟据键做出相应的反应
        static int64_t last_time;
        if (received_nb_signals)
            break;
        /* read_key() returns 0 on EOF */
        if (cur_time - last_time >= 100000 && !run_as_daemon){
            key = read_key();
            last_time = cur_time;
        }else{
    }
}
```

## 《FFmpeg 基础库编程开发》

```
/* select the stream that we must read now by looking at the
   smallest output pts */
//下面这个循环的目的是找一个最小的输出 pts(也就是离当前最近的)的输出流
file_index = -1;
for (i = 0; i < nb_output_streams; i++){
    OutputFile *of;
    int64_t ipts;
    double opts;
    ost = &output_streams[i];//循环每一个输出流
    of = &output_files[ost->file_index];//输出流对应的输出文件
    os = output_files[ost->file_index].ctx;//输出流对应的 FormatContext
    ist = &input_streams[ost->source_index];//输出流对应的输入流

    if (ost->is_past_recording_time || //是否过了录制时间?(可能用户指定了一个录制时间段)
        no_packet[ist->file_index] || //对应的输入流这个时间内没有数据?
        (os->pb && avio_tell(os->pb) >= of->limit_filesize))//是否超出了录制范围(也是用户指定的)
        continue;//是的,符合上面某一条,那么再看下一个输出流吧

    //判断当前输入流所在的文件是否可以使用(我也不很明白)
    opts = ost->st->pts.val * av_q2d(ost->st->time_base);
    ipts = ist->pts;
    if (!input_files[ist->file_index].eof_reached) {
        if (ipts < ipts_min){
            //每找到一个 pts 更小的输入流就记录下来,这样循环完所有的输出流时就找到了
            //pts 最小的输入流,及输入文件的序号
            ipts_min = ipts;
            if (input_sync)
                file_index = ist->file_index;
        }
        if (opts < opts_min){
            opts_min = opts;
            if (!input_sync)
                file_index = ist->file_index;
        }
    }
}

//难道下面这句话的意思是:如果当前的输出流已接收的帧数,超出用户指定的输出最大帧数时,
//则当前输出流所属的输出文件对应的所有输出流,都算超过了录像时间?
if (ost->frame_number >= ost->max_frames){
    int j;
    for (j = 0; j < of->ctx->nb_streams; j++)
        output_streams[of->ost_index + j].is_past_recording_time = 1;
    continue;
}
```

```

        }
    }

/* if none, if is finished */
if (file_index < 0) {
    //如果没有找到合适的输入文件
    if (no_packet_count){
        //如果是因为有的输入文件暂时得不到数据,则还不算是结束
        no_packet_count = 0;
        memset(no_packet, 0, nb_input_files);
        usleep(10000);
        continue;
    }
    //全部转换完成了,跳出大循环
    break;
}

//从找到的输入文件中读出一帧(可能是音频也可能是视频),并放到 fifo 队列中
is = input_files[file_index].ctx;
ret = av_read_frame(is, &pkt);
if (ret == AVERROR(EAGAIN)) {
    //此时发生了暂时没数据的情况
    no_packet[file_index] = 1;
    no_packet_count++;
    continue;
}

//下文判断是否有输入文件到最后了
if (ret < 0){
    input_files[file_index].eof_reached = 1;
    if (opt_shortest)
        break;
    else
        continue;
}

no_packet_count = 0;
memset(no_packet, 0, nb_input_files);

if (do_pkt_dump){
    av_pkt_dump_log2(NULL, AV_LOG_DEBUG, &pkt, do_hex_dump,
                     is->streams[pkt.stream_index]);
}
/* the following test is needed in case new streams appear

```

## 《FFmpeg 基础库编程开发》

```
dynamically in stream : we ignore them */
//如果在输入文件中遇到一个忽然冒出的流,那么我们不鸟它
if (pkt.stream_index >= input_files[file_index].nb_streams)
    goto discard_packet;

//取得当前获得的帧对应的输入流
ist_index = input_files[file_index].ist_index + pkt.stream_index;
ist = &input_streams[ist_index];
if (ist->discard)
    goto discard_packet;

//重新鼓捣一下帧的时间戳
if (pkt.dts != AV_NOPTS_VALUE)
    pkt.dts += av_rescale_q(input_files[ist->file_index].ts_offset,
                           AV_TIME_BASE_Q, ist->st->time_base);
if (pkt.pts != AV_NOPTS_VALUE)
    pkt.pts += av_rescale_q(input_files[ist->file_index].ts_offset,
                           AV_TIME_BASE_Q, ist->st->time_base);

if (pkt.pts != AV_NOPTS_VALUE)
    pkt.pts *= ist->ts_scale;
if (pkt.dts != AV_NOPTS_VALUE)
    pkt.dts *= ist->ts_scale;

if (pkt.dts != AV_NOPTS_VALUE && ist->next_pts != AV_NOPTS_VALUE
    && (is->iformat->flags & AVFMT_TS_DISCONT))
{
    int64_t pkt_dts = av_rescale_q(pkt.dts, ist->st->time_base,
                                   AV_TIME_BASE_Q);
    int64_t delta = pkt_dts - ist->next_pts;
    if ((delta < -1LL * dts_delta_threshold * AV_TIME_BASE
        || (delta > 1LL * dts_delta_threshold * AV_TIME_BASE
            && ist->st->codec->codec_type
            != AVMEDIA_TYPE_SUBTITLE)
        || pkt_dts + 1 < ist->pts) && !copy_ts)
    {
        input_files[ist->file_index].ts_offset -= delta;
        av_log( NULL, AV_LOG_DEBUG,
                "timestamp discontinuity %"PRIId64", new offset= %"PRIId64"\n",
                delta, input_files[ist->file_index].ts_offset);
        pkt.dts -= av_rescale_q(delta, AV_TIME_BASE_Q, ist->st->time_base);
        if (pkt.pts != AV_NOPTS_VALUE)
            pkt.pts -= av_rescale_q(delta, AV_TIME_BASE_Q, ist->st->time_base);
```

```

        }
    }

//把这一帧转换并写入到输出文件中
if (output_packet(ist, output_streams, nb_output_streams, &pkt) < 0){
    av_log(NULL, AV_LOG_ERROR,
           "Error while decoding stream #%"PRIu32":%"PRIu32"\n",
           ist->file_index, ist->st->index);
    if (exit_on_error)
        exit_program(1);
    av_free_packet(&pkt);
    continue;
}

```

discard\_packet:

```
av_free_packet(&pkt);
```

```
/* dump report by using the output first video and audio streams */
print_report(output_files, output_streams, nb_output_streams, 0,
             timer_start, cur_time);
}
```

//文件处理完了,把缓冲中剩余的数据写到输出文件中

```
for (i = 0; i < nb_input_streams; i++){
    ist = &input_streams[i];
    if (ist->decoding_needed){
        output_packet(ist, output_streams, nb_output_streams, NULL);
    }
}
flush_encoders(output_streams, nb_output_streams);
```

term\_exit();

//为输出文件写文件尾(有的不需要).

```
for (i = 0; i < nb_output_files; i++){
    os = output_files[i].ctx;
    av_write_trailer(os);
}
```

```
/* dump report by using the first video and audio streams */
print_report(output_files, output_streams, nb_output_streams, 1,
             timer_start, av_gettime());
```

## 《FFmpeg 基础库编程开发》

```
//关闭所有的编码器
for (i = 0; i < nb_output_streams; i++){
    ost = &output_streams[i];
    if (ost->encoding_needed){
        av_freep(&ost->st->codec->stats_in);
        avcodec_close(ost->st->codec);
    }
}

#ifndef CONFIG_AVFILTER
avfilter_graph_free(&ost->graph);
#endif

//关闭所有的解码器
for (i = 0; i < nb_input_streams; i++){
    ist = &input_streams[i];
    if (ist->decoding_needed){
        avcodec_close(ist->st->codec);
    }
}

/* finished ! */
ret = 0;

fail: av_freep(&bit_buffer);
av_freep(&no_packet);

if (output_streams) {
    for (i = 0; i < nb_output_streams; i++)  {
        ost = &output_streams[i];
        if (ost)      {
            if (ost->stream_copy)
                av_freep(&ost->st->codec->extradata);
            if (ost->logfile){
                fclose(ost->logfile);
                ost->logfile = NULL;
            }
            av_fifo_free(ost->fifo); /* works even if fifo is not
                                         initialized but set to zero */
            av_freep(&ost->st->codec->subtitle_header);
            av_free(ost->resample_frame.data[0]);
            av_free(ost->forced_kf_pts);
            if (ost->video_resample)
                sws_freeContext(ost->img_resample_ctx);
        }
    }
}
```

《FFmpeg 基础库编程开发》

```

    swr_free(&ost->swr);
    av_dict_free(&ost->opts);
}
}

return ret;
}

```

## 第九章 ffmpeg 相关工程

### 9.1 ffdshow

ffdshow 是基于 ffmpeg 的解码器类库 libavcodec 的 DirectShow Filter。广泛安装在 PC 上

#### ffdshow 源代码分析 1：整体结构

ffdshow 是一个非常强大的 DirectShow 解码器，封装了 ffmpeg，libmpeg2 等解码库。它也提供了丰富的加工处理选项，可以锐化画面，调节画面的亮度等等。不止是视频，FFDShow 现在同样可以解码音频，AC3、MP3 等音频格式都可支持。并且可以外挂 winamp 的 DSP 插件，来改善听觉效果。一个词形容：强大。

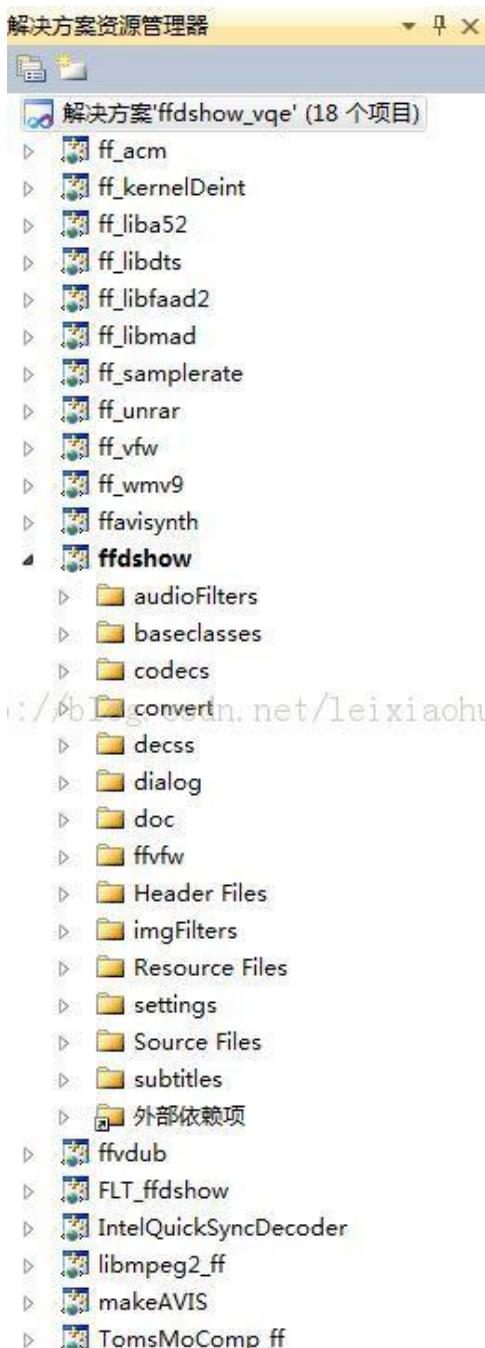
因为项目的要求，需要对 ffdshow 进行二次开发，正好有这个机会，分析研究一下 ffdshow 的源代码。

ffdshow 项目的资源可以从 sourceforge 下载。包括编译好的程序，以及原代码等，下载地址：  
<http://sourceforge.net/projects/ffdshow-tryout/>

注意：sourceforge 上有两个版本的 ffdshow：ffdshow 以及 ffdshow-tryout。其中前一个版本很早之前已经停止开发了，因此我们需要选择后一个（ffdshow-tryout）。

下载源代码的方法不再赘述，下面直接进入正题。源代码下载后，需要进行编译，推荐使用源代码根目录下的 bat 脚本一次性完成所有的资源编译。

编译完成后我们就可以打开源代码根目录里的工程了。我自己的开发环境是 VC2010，打开后工程如下图所示（解决方案的名字被我修改了==）：



由图可见，ffdshow 由一大堆工程组成，乍一看给人一种杂乱无章的感觉，其实大部分工程我们不用去理会，我们重点研究最重要的工程就是那个名字叫“ffdshow”的工程。

下面我介绍几个最重要的文件夹里包含的代码的功能：

**audiofilters:** 音频滤镜都在这里面（例如 EQ，调节高低频等）

**baseclasses:** 微软自带 directshow 的 sdk 里面有，主要是微软为了方便 DirectShow 开发而提供的一些基本的类

**codecs:** 支持的解码器都在这里（例如 libavcodec, libmpeg2 等）

**convert:** 色彩转换的一些功能（没太用过）

**decss:** 解除版权加密的一些功能（没太用过）

**dialog:** 音频视频滤镜的配置页面

**doc:** 文档，不是程序

ffvfw: VFW 相关（目前没太用过）

**Header Files:** 核心代码的头文件

imgfilters: 视频滤镜都在这里（显示 QP/MV，加 LOGO，显示视频信息等）

Resource Files: 资源文件

settings: 音频视频滤镜的配置信息

Source Files: 核心代码的源文件

subtitles: 字幕相关的功能

以上用红色标出的，是我们二次开发中最有可能会涉及到的三个部分。掌握了这三个部分，就可以往 ffdshow 中添加自己写的滤镜（注意：这里说的是视频滤镜，音频的方法是一样的）

黄色背景标出的部分，虽然我们可能不需要做出什么改变，但是为了了解 ffdshow 的架构，我们需要分析其中的代码。

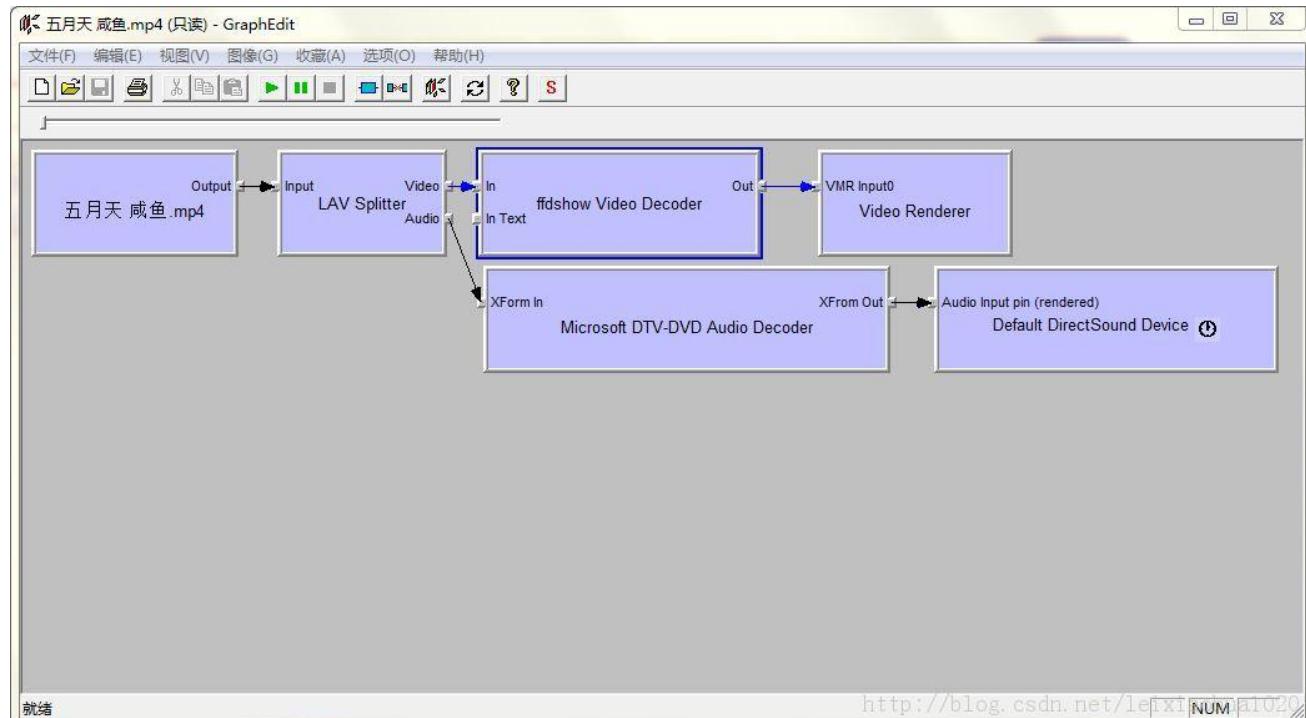
## ffdshow 源代码分析 2: 位图覆盖滤镜（对话框部分 Dialog）

本文我们介绍 ffdshow 的滤镜功能。ffdshow 支持很多种滤镜，可以支持多种视频和音频的后期效果。例如 OSD 滤镜支持在视频的左上角显示视频相关的信息。而可视化滤镜则支持显示视频每一帧的运动矢量以及量化参数。在这里我们介绍一种位图覆盖（Bitmap）滤镜（Filter）。

效果

编译完 ffdshow 之后，在“项目属性->调试->命令”里面将 GraphEdit.exe 所在位置设置为调试程序，例如在这里我设置了《终极解码》里面自带 GraphEdit.exe，路径为“C:\Program Files\Final Codecs\Codecs\GraphEdit.exe”。这样就可以使用 GraphEdit.exe 调试 ffdshow 了。

向 GraphEdit.exe 里面拖入一个文件“五月天 咸鱼.mp4”，结果如下图所示：



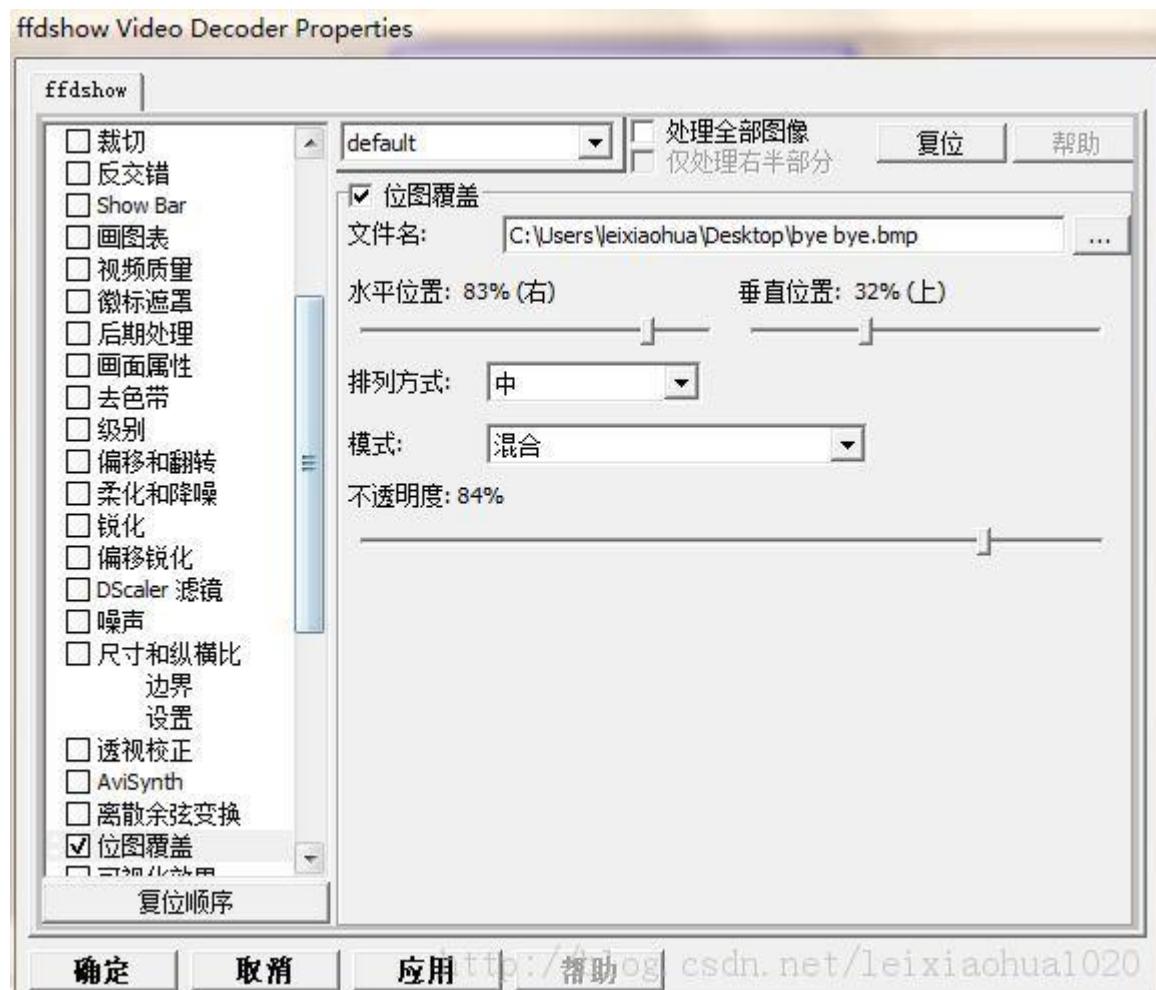
注：有的时候默认的视频解码器可能不是 ffdshow，可能是 CoreAVC 等，可以先删除视频解码器然后添加 ffdshow。点击绿色三角形按钮就可以开始播放视频。

## 《FFmpeg 基础库编程开发》

右键点击 ffdshow 组件，打开属性对话框之后，可以看见右边栏中有很多的滤镜。

勾选“位图覆盖”滤镜，然后选择一张用于覆盖的图片（在这里我选择了一张 bmp 格式的专辑封面）。

注：可以调整位图所在的水平位置，垂直位置，不透明度，并且可以修改位图叠加模式（在这里用混合）。



添加了该滤镜之后，播放窗口的显示内容为：

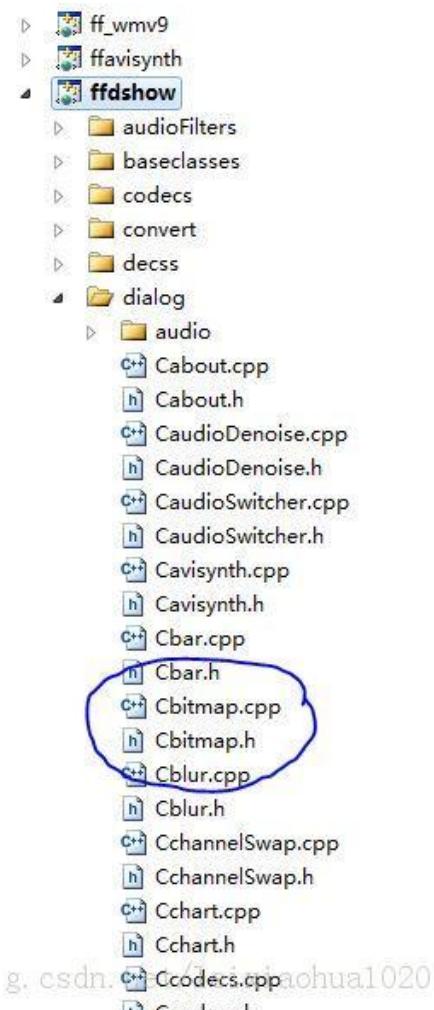


可见在右上角显示出了叠加的位图。

## 源代码分析

### 1.对话框部分

与位图覆盖（Bitmap）滤镜的对话框有关的类位于 dialog 目录下的 Cbitmap.cpp 和 Cbitmap.h 文件中。



先来看看 Cbitmap.h 中类的声明：

需要注意的是，里面类的名字居然叫 TbitmapPage，而没有和头文件名字一致。= =

```
#ifndef _CBITMAPPAGE_H_
#define _CBITMAPPAGE_H_
```

```
#include "TconfPageDecVideo.h"
//Bitmap 配置页面
class TbitmapPage : public TconfPageDecVideo
{
private:
    void pos2dlg(void), opacity2dlg(void);
    //设置文件路径
    void onFlnm(void);
protected:
```

## 《FFmpeg 基础库编程开发》

```
virtual INT_PTR msgProc(UINT uMsg, WPARAM wParam, LPARAM lParam);  
public:  
    //构造函数  
    TbitmapPage(TffdshowPageDec *Iparent, const TfilterIDFF *idff);  
    //初始化  
    virtual void init(void);  
    //配置数据传入到对话框界面  
    virtual void cfg2dlg(void);  
    virtual void translate(void);  
};
```

```
#endif
```

再看看 Cbitmap.cpp 文件吧。关键的代码都已经加上了注释。

```
/*  
 * Copyright (c) 2004-2006 Milan Cutka  
 *  
 * This program is free software; you can redistribute it and/or modify  
 * it under the terms of the GNU General Public License as published by  
 * the Free Software Foundation; either version 2 of the License, or  
 * (at your option) any later version.  
 *  
 * This program is distributed in the hope that it will be useful,  
 * but WITHOUT ANY WARRANTY; without even the implied warranty of  
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the  
 * GNU General Public License for more details.  
 *  
 * You should have received a copy of the GNU General Public License  
 * along with this program; if not, write to the Free Software  
 * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA  
 */
```

```
//Bitmap 配置页面  
#include "stdafx.h"  
#include "TsubtitlesSettings.h"  
#include "TbitmapSettings.h"  
#include "Cbitmap.h"  
//初始化  
void TbitmapPage::init(void)  
{  
    //设置滑动条范围  
    edLimitText(IDC_ED_BITMAP_FLNM, MAX_PATH);  
    tbrSetRange(IDC_TBR_BITMAP_POSX, 0, 100, 10);  
    tbrSetRange(IDC_TBR_BITMAP_POSY, 0, 100, 10);  
    tbrSetRange(IDC_TBR_BITMAP_OPACITY, 0, 256);
```

## 《FFmpeg 基础库编程开发》

```
}

//配置数据传入到对话框界面
void TbitmapPage::cfg2dlg(void)
{
    //各种设置
    //EditControl 设置
    setDlgItemText(m_hwnd, IDC_ED_BITMAP_FLNM, cfgGetStr(IDFF_bitmapFlnm));
    pos2dlg();
    cbxSetCurSel(IDC_CBX_BITMAP_ALIGN, cfgGet(IDFF_bitmapAlign));
    cbxSetCurSel(IDC_CBX_BITMAP_MODE, cfgGet(IDFF_bitmapMode));
    opacity2dlg();
}

//Bitmap 位置信息
void TbitmapPage::pos2dlg(void)
{
    char_t s[260];
    int x;
    //获取
    x = cfgGet(IDFF_bitmapPosx);
    TsubtitlesSettings::getPosHoriz(x, s, this, IDC_LBL_BITMAP_POSX, countof(s));
    setDlgItemText(m_hwnd, IDC_LBL_BITMAP_POSX, s);
    //设置
    tbrSet(IDC_TBR_BITMAP_POSX, x);

    x = cfgGet(IDFF_bitmapPosy);
    TsubtitlesSettings::getPosVert(x, s, this, IDC_LBL_BITMAP_POSY, countof(s));
    setDlgItemText(m_hwnd, IDC_LBL_BITMAP_POSY, s);
    tbrSet(IDC_TBR_BITMAP_POSY, x);
}

void TbitmapPage::opacity2dlg(void)
{
    int o = cfgGet(IDFF_bitmapStrength);
    tbrSet(IDC_TBR_BITMAP_OPACITY, o);
    setText(IDC_LBL_BITMAP_OPACITY, _l("%s %i%%"), _(IDC_LBL_BITMAP_OPACITY), 100 * o / 256);
}

INT_PTR TbitmapPage::msgProc(UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg) {
        case WM_COMMAND:
            switch (LOWORD(wParam)) {
                case IDC_ED_BITMAP_FLNM:
                    if (HIWORD(wParam) == EN_CHANGE && !isSetText) {

```

《FFmpeg 基础库编程开发》

```

char_t flnm[MAX_PATH];
GetDlgItemText(m_hwnd, IDC_ED_BITMAP_FLNM, flnm, MAX_PATH);
cfgSet(IDFF_bitmapFlnm, flnm);
}
return TRUE;
}
break;
}

return TconfPageDecVideo::msgProc(uMsg, wParam, lParam);
}

//设置文件路径
void TbitmapPage::onFlnm(void)
{
    char_t flnm[MAX_PATH];
    cfgGet(IDFF_bitmapFlnm, flnm, MAX_PATH);
    if (dlgGetFile(false, m_hwnd, _(-IDD_BITMAP, _l("Load image file")), _l("All supported
(*.jpg,*.bmp,*.gif,*.png)\0*.bmp;*.jpg;*.jpeg;*.gif;*.png\0Windows Bitmap (*.bmp)\0*.bmp\0JPEG
(*.jpg)\0*.jpg\0Compuserve Graphics Interchange (*.gif)\0*.gif\0Portable Network Graphics (*.png)\0*.png"), _l("bmp"),
flnm, _l("."), 0)) {
        setDlgItemText(m_hwnd, IDC_ED_BITMAP_FLNM, flnm);
        //设置
        cfgSet(IDFF_bitmapFlnm, flnm);
    }
}

void TbitmapPage::translate(void)
{
    TconfPageBase::translate();

    cbxTranslate(IDC_CBX_BITMAP_ALIGN, TsubtitlesSettings::alignments);
    cbxTranslate(IDC_CBX_BITMAP_MODE, TbitmapSettings::modes);
}

//构造函数
TbitmapPage::TbitmapPage(TffdshowPageDec *Iparent, const TfilterIDFF *idff): TconfPageDecVideo(Iparent, idff)
{
    //各种绑定
    resInter = IDC_CHB_BITMAP;
    static const TbindTrackbar<TbitmapPage> htbr[] = {
        IDC_TBR_BITMAP_POSX, IDFF_bitmapPosx, &TbitmapPage::pos2dlg,
        IDC_TBR_BITMAP_POSY, IDFF_bitmapPosy, &TbitmapPage::pos2dlg,
        IDC_TBR_BITMAP_OPACITY, IDFF_bitmapStrength, &TbitmapPage::opacity2dlg,
        0, 0, NULL
    };
}

```

```

bindHtracks(htbr);

static const TbindCombobox<TbitmapPage> cbx[] = {
    IDC_CBX_BITMAP_ALIGN, IDFF_bitmapAlign, BINDCBX_SEL, NULL,
    IDC_CBX_BITMAP_MODE, IDFF_bitmapMode, BINDCBX_SEL, NULL,
    0
};

bindComboboxes(cbx);

static const TbindButton<TbitmapPage> bt[] = {
    IDC_BT_BITMAP_FLNM, &TbitmapPage::onFlnm,
    0, NULL
};

bindButtons(bt);
}

```

看 ffdshow 源代码的时候，开始会比较费劲。为什么？因为它使用了大量自己写的 API 函数，以及自己定义的结构体。这些 API 函数的种类繁多，如果一个一个都看完，估计就精疲力竭了。经过一段时间的学习之后，我发现最方便的方法还是根据函数名字推测其用法。因此我就不深入剖析 ffdshow 的 API 函数了。

以上源代码中包含以下 API（大致按出现先后次序，可能没有例举全，在这里只是举例子）：

```

edLimitText();//限制输入字符串长度
tbrSetRange();//设置滑动条范围
setDlgItemText();//设置组件名称
cbxSetCurSel();//设置下拉框当前选项
cfgGet();//从注册表中读取变量的值
tbrSet();//设置滑动条的值
bindHtracks();//绑定注册表变量和滑动条
bindComboboxes();//绑定注册表变量和下拉框
bindButtons();//绑定函数和按钮

```

从以上函数大致可以看出 tbr\*\*\*()基本上都是操作滑动条的，cbx\*\*\*()基本上都是操作下拉框的，函数基本上可以从名称上理解其的意思。bind\*\*\*()就是绑定注册表变量和控件的。注意 ffdshow 里面有注册表变量这么一个概念。这些变量的值存在系统的注册表里面，不会因为程序结束运行而消失。就目前我的观察来看，绝大部分注册表变量存的是一个整数值。这些注册表变量都以 IDFF\_xxx 的名称预编译定义在 ffdshow\_constants.h 头文件中。与 MFC 控件可以直接与 CString, int 等变量绑定不同，ffdshow 控件只可以和注册表变量绑定。即每次运行的时候都从注册表加载变量的值到界面上。存储的时候把界面上的值存储到注册表中。

注：注册表变量如下所示（截取了一小段）

```

#define IDFF_filterBitmap 1650
#define IDFF_isBitmap 1651
#define IDFF_showBitmap 1652
#define IDFF_orderBitmap 1653
#define IDFF_fullBitmap 1654
#define IDFF_bitmapFlnm 1655
#define IDFF_bitmapPosx 1656
#define IDFF_bitmapPosy 1657
#define IDFF_bitmapPosmode 1658
#define IDFF_bitmapAlign 1659

```

```
#define IDFF_bitmapMode 1660
```

```
#define IDFF_bitmapStrength 1661
```

此外需要注意的是，ffdshow 尽管包含了图形化的属性界面，却没有使用 MFC 类库，因而 MFC 的很多函数都不能使用，对此我还不甚了解为什么要这样，以后有机会要探究探究。

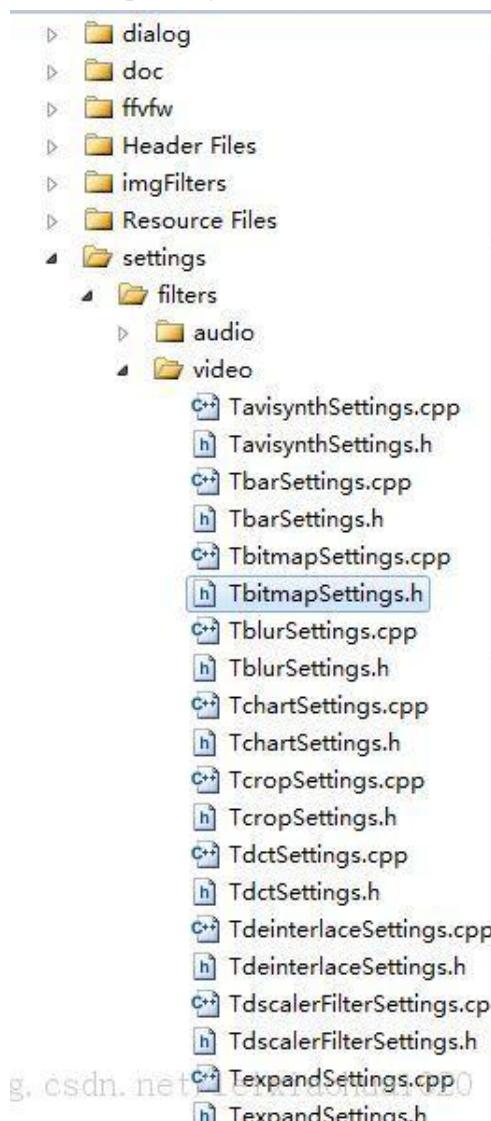
## ffdshow 源代码分析 3：位图覆盖滤镜（设置部分 Settings）

在这里再介绍一下设置部分（Settings），此外还有一个滤镜部分（Filter）。这三个部分就可以组成一个 ffdshow 的滤镜功能了。

### 设置部分（Settings）

在 ffdshow 中滤镜的设置部分（Settings）主要用于存储滤镜运行过程中需要用到的各种变量。一般情况下通过读取注册表变量并赋值给该类当中的变量从而达到操作相应滤镜的功能。

与位图覆盖(Bitmap)滤镜的设置有关的类位于 settings->filters->video 目录下(隐藏的很深啊)的 TbitmapSettings.cpp 和 TbitmapSettings.h 文件中。



先来看看 TbitmapSettings.h

## 《FFmpeg 基础库编程开发》

该类的名字叫 TbitmapSettings，从类的定义我们可以看出，

flnm[]存储了打开的位图的路径

posx, posy 存储了位图在屏幕上显示的位置

mode 存储了显示的方式

等等，所有跟该滤镜（Filter）相关的数据都存储在该类之中。

该类包含一个 TfilterIDFF 类型的结构体 idffs，用于存储该滤镜的一些属性信息（名称，ID，属性对话框 ID 等等）

此外，有两个函数至关重要。createFilters()用于创建滤镜(Filter)。createPages()用于创建滤镜的配置对话框(Dialog)。

```
#ifndef _TBITMAPSETTINGS_H_
```

```
#define _TBITMAPSETTINGS_H_
```

//各个 Filter 预设值

```
#include "TfilterSettings.h"
```

```
#include "Tfont.h"
```

//Bitmap 的配置信息

```
struct TbitmapSettings : TfilterSettingsVideo {
```

private:

```
    static const TfilterIDFF idffs;
```

protected:

```
    virtual const int *getResets(unsigned int pageId);
```

public:

```
    TbitmapSettings(TintStrColl *Icoll = NULL, TfilterIDFFs *filters = NULL);
```

//Bitmap 文件路径

```
    char_t flnm[MAX_PATH];
```

//x,y 坐标，以及坐标的模式

```
    int posx, posy, posmode;
```

```
    int align;
```

//叠加方式

```
    enum {
```

```
        MODE_BLEND = 0,
```

```
        MODE_DARKEN = 1,
```

```
        MODE_LIGHTEN = 2,
```

```
        MODE_ADD = 3,
```

```
        MODE_SOFTLIGHT = 4,
```

```
        MODE_EXCLUSION = 5
```

```
    };
```

```
    int mode;
```

```
    static const char_t *modes[];
```

```
    int strength;
```

//创建 Filter

```
    virtual void createFilters(size_t filtersorder, Tfilters *filters, TfilterQueue &queue) const;
```

//创建属性页面

```
    virtual void createPages(TffdshowPageDec *parent) const;
```

```
    virtual bool getTip(unsigned int pageId, char_t *buf, size_t buflen);
```

```
};
```

```
#endif
```

再来看看 TbitmapSettings.cpp

该类包含了 TbitmapSettings 类中函数方法的具体实现。首先看一下构造函数 TbitmapSettings()。从构造函数中可以看出，绑定了类中的变量和注册表变量，使它们形成一一对应的关系。其他的函数就不再细说了，比较简单，理解起来比较容易。

```
#include "stdafx.h"
#include "TbitmapSettings.h"
#include "TimgFilterBitmap.h"
#include "Cbitmap.h"
#include "TffdshowPageDec.h"
#include "TsubtitlesSettings.h"
//几种叠加方式
const char_t* TbitmapSettings::modes[] = {
    _l("blend"),
    _l("darken"),
    _l("lighten"),
    _l("add"),
    _l("softlight"),
    _l("exclusion"),
    NULL
};

//Filter 属性
const TfilterIDFF TbitmapSettings::idffs = {
    /*name*/      _l("Bitmap overlay"),
    /*id*/        IDFF_filterBitmap,
    /*is*/        IDFF_isBitmap,
    /*order*/     IDFF_orderBitmap,
    /*show*/      IDFF_showBitmap,
    /*full*/     IDFF_fullBitmap,
    /*half*/      0,
    /*dlgId*/    IDD_BITMAP,
};

//构造函数
TbitmapSettings::TbitmapSettings(TintStrColl *Icoll, TfilterIDFFs *filters): TfilterSettingsVideo(sizeof(*this), Icoll, filters,
&idffs)
{
    half = 0;
    memset(flnm, 0, sizeof(flnm));
    //绑定变量
    static const TintOptionT<TbitmapSettings> iopts[] = {
        IDFF_isBitmap      , &TbitmapSettings::is      , 0, 0, _l(""), 1,
        _l("isBitmap"), 0,
```

《FFmpeg 基础库编程开发》

```

IDFF_showBitmap      , &TbitmapSettings::show      , 0, 0, _l(""), 1,
_l("showBitmap"), 1,
IDFF_orderBitmap    , &TbitmapSettings::order     , 1, 1, _l(""), 1,
_l("orderBitmap"), 0,
IDFF_fullBitmap     , &TbitmapSettings::full      , 0, 0, _l(""), 1,
_l("fullBitmap"), 0,
IDFF_bitmapPosx     , &TbitmapSettings::posx      , -4096, 4096, _l(""), 1,
_l("bitmapPosX"), 50,
IDFF_bitmapPosy     , &TbitmapSettings::posy      , -4096, 4096, _l(""), 1,
_l("bitmapPosY"), 50,
IDFF_bitmapPosmode  , &TbitmapSettings::posmode   , 0, 1, _l(""), 1,
_l("bitmapPosMode"), 0,
IDFF_bitmapAlign    , &TbitmapSettings::align     , 0, 3, _l(""), 1,
_l("bitmapAlign"), ALIGN_CENTER,
IDFF_bitmapMode     , &TbitmapSettings::mode      , 0, 5, _l(""), 1,
_l("bitmapMode"), 0,
IDFF_bitmapStrength , &TbitmapSettings::strength  , 0, 256, _l(""), 1,
_l("bitmapStrength"), 128,
0
};

addOptions(iopts);
static const TstrOption sopts[] = {
    IDFF_bitmapFlnm    , (TstrVal)&TbitmapSettings::flnm   , MAX_PATH, 0, _l(""), 1,
    _l("bitmapFlnm"), _l(""),
    0
};

addOptions(sopts);

static const TcreateParamList1 listMode(modes);
setParamList(IDFF_bitmapMode, &listMode);
static const TcreateParamList1 listAlign(TsubtitlesSettings::alignments);
setParamList(IDFF_bitmapAlign, &listAlign);
}

//创建 Filter
void TbitmapSettings::createFilters(size_t filtersorder, Tfilters *filters, TfilterQueue &queue) const
{
    idffOnChange(idffs, filters, queue temporary);
    if (is && show) {
        queueFilter<TimgFilterBitmap>(filtersorder, filters, queue);
    }
}

//创建属性页面
void TbitmapSettings::createPages(TffdshowPageDec *parent) const

```

```

{
    parent->addFilterPage<TbitmapPage>(&idffs);
}

const int* TbitmapSettings::getResets(unsigned int pageId)
{
    static const int idResets[] = {
        IDFF_bitmapPosx,   IDFF_bitmapPosy,   IDFF_bitmapPosmode,   IDFF_bitmapAlign,   IDFF_bitmapMode,
        IDFF_bitmapStrength,
        0
    };
    return idResets;
}

bool TbitmapSettings::getTip(unsigned int pageId, char_t *tipS, size_t len)
{
    if (flnm[0]) {
        tsnprintf_s(tipS, len, _TRUNCATE, _l("%s %s"), modes[mode], flnm);
        tipS[len - 1] = '\0';
    } else {
        tipS[0] = '\0';
    }
    return true;
}

```

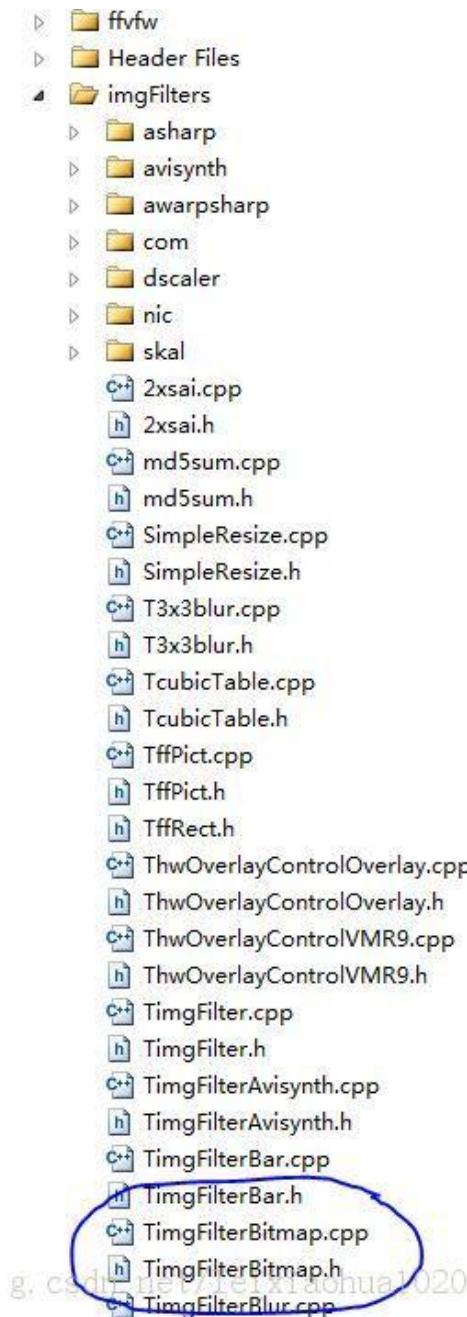
## ffdshow 源代码分析 4：位图覆盖滤镜（滤镜部分 Filter）

### 滤镜部分（Filter）

ffdshow 的滤镜的滤镜部分（怎么感觉名字有点重复 ==，算了先这么叫吧）的功能主要用于完成具体的图像处理功能。具体到位图覆盖滤镜的话，就是用于把图片覆盖到视频上面，他是 ffdshow 滤镜的核心。

与位图覆盖（Bitmap）滤镜的滤镜处理有关的类位于 imgFilters 目录下的 TimgFilterBitmap.h 和 TimgFilterBitmap.cpp 文件中。

## 《FFmpeg 基础库编程开发》



先来看看 TimgFilterBitmap.h

这里要注意一下，该类的名字叫 TimgFilterBitmap。它的声明方式确实比较奇怪：DECLARE\_FILTER(TimgFilterBitmap, public, TimgFilter)

可以看出 DECLARE\_FILTER 是一个宏，具体这个宏的内部代码就先不查看了，否则会感觉很混乱，暂且留下一个小小的谜团。在这里只要知道这是声明了一个滤镜类就可以了。

其实 TimgFilterBitmap 的核心函数不多，就一个，那就是 process()，具体的处理功能都是在这个函数里面实现的。

```
#ifndef _TIMGFILTERBITMAP_H_
#define _TIMGFILTERBITMAP_H_
//叠加一张位图
#include "TimgFilter.h"
#include "Tfont.h"
```

```

struct TffPict;
struct TbitmapSettings;
//特别的声明方式 ==
DECLARE_FILTER(TimgFilterBitmap, public, TimgFilter)
private:
//图像
TffPict *bitmap;
//内存
Tbuffer bitmapbuf;
char_t oldflnm[MAX_PATH];
typedef void (*Tblendplane)(const TcspInfo &cspInfo, const unsigned int dx[3], const unsigned int dy[3], unsigned char
*dst[3], const stride_t dststride[3], const unsigned char *src[3], const stride_t srcstride[3], int strength, int invstrength);
//注意 这个类有一个实例，名字叫 w
class TrenderedSubtitleLineBitmap : public TrenderedSubtitleWordBase
{
public:
    TrenderedSubtitleLineBitmap(void): TrenderedSubtitleWordBase(false) {}
    TffPict *pict;
    const TbitmapSettings *cfg;
    //叠加
    Tblendplane blend;
    //打印
    virtual void print(int startx, int starty /* not used */, unsigned int dx[3], int dy1[3], unsigned char *dstLn[3], const
    stride_t stride[3], const unsigned char *bmp[3], const unsigned char *msk[3], REFERENCE_TIME rtStart =
    REFTIME_INVALID) const;
} w;
TrenderedSubtitleLine l;
//是 TrenderedSubtitleLine 的一个 vector
TrenderedSubtitleLines ls;
int oldmode;
//几种叠加方式
template<class _mm> static void blend(const TcspInfo &cspInfo, const unsigned int dx[3], const unsigned int dy[3],
unsigned char *dst[3], const stride_t dststride[3], const unsigned char *src[3], const stride_t srcstride[3], int strength, int
invstrength);
template<class _mm> static void add(const TcspInfo &cspInfo, const unsigned int dx[3], const unsigned int dy[3], unsigned
char *dst[3], const stride_t dststride[3], const unsigned char *src[3], const stride_t srcstride[3], int strength, int invstrength);
template<class _mm> static void darken(const TcspInfo &cspInfo, const unsigned int dx[3], const unsigned int dy[3],
unsigned char *dst[3], const stride_t dststride[3], const unsigned char *src[3], const stride_t srcstride[3], int strength, int
invstrength);
template<class _mm> static void lighten(const TcspInfo &cspInfo, const unsigned int dx[3], const unsigned int dy[3],
unsigned char *dst[3], const stride_t dststride[3], const unsigned char *src[3], const stride_t srcstride[3], int strength, int
invstrength);

```

## 《FFmpeg 基础库编程开发》

```
template<class _mm> static void softlight(const TcspInfo &cspInfo, const unsigned int dx[3], const unsigned int dy[3],  
unsigned char *dst[3], const stride_t dststride[3], const unsigned char *src[3], const stride_t srcstride[3], int strength, int  
invstrength);  
template<class _mm> static void exclusion(const TcspInfo &cspInfo, const unsigned int dx[3], const unsigned int dy[3],  
unsigned char *dst[3], const stride_t dststride[3], const unsigned char *src[3], const stride_t srcstride[3], int strength, int  
invstrength);  
//获取叠加方式  
template<class _mm> static Tblendplane getBlend(int mode);  
protected:  
virtual bool is(const TffPictBase &pict, const TfilterSettingsVideo *cfg);  
virtual uint64_t getSupportedInputColorspaces(const TfilterSettingsVideo *cfg) const  
{  
    return FF_CSPPS_MASK_YUV_PLANAR;  
}  
public:  
TimgFilterBitmap(IffdshowBase *Idec, Tfilters *Iparent);  
virtual ~TimgFilterBitmap();  
//核心函数（Filter 配置信息队列，图像，配置信息）  
virtual HRESULT process(TfilterQueue::iterator it, TffPict &pict, const TfilterSettingsVideo *cfg0);  
};
```

```
#endif
```

再来看看 TimgFilterBitmap.cpp

这个文件本身代码量是比较大的，只是其他部分我都还没有仔细分析，确实没那没多时间。。。在这里仅简要分析一下最核心的函数 process()。正是这个函数真正实现了滤镜的功能。在这个位图叠加滤镜中，process()实现了位图在视频上面的叠加功能。

//核心函数（Filter 配置信息队列，图像，配置信息）

```
HRESULT TimgFilterBitmap::process(TfilterQueue::iterator it, TffPict &pict, const TfilterSettingsVideo *cfg0)  
{  
    //都有这一句==  
    if (is(pict, cfg0)) {  
        //Bitmap 的配置信息  
        const TbitmapSettings *cfg = (const TbitmapSettings*)cfg0;  
        init(pict, cfg->full, cfg->half);  
        unsigned char *dst[4];  
        bool cspChanged = getCurNext(FF_CSPPS_MASK_YUV_PLANAR, pict, cfg->full, COPYMODE_DEF, dst);  
        //处理  
        if (!bitmap || cspChanged || strcmp(oldflnm, cfg->flnm) != 0) {  
            ff_strncpy(oldflnm, cfg->flnm, countof(oldflnm));  
            if (bitmap) {  
                delete bitmap;  
            }  
            //新建一张图
```

```

//通过 cfg->flnm 路径
//载入 bitmapbuf
bitmap = new TffPict(csp2, cfg->flnm, bitmapbuf, deci);
//3 个颜色分量？
for (int i = 0; i < 3; i++) {
    w.dx[i] = bitmap->rectFull.dx >> bitmap->cspInfo.shiftX[i];
    w.dy[i] = bitmap->rectFull.dy >> bitmap->cspInfo.shiftY[i];
    w.bmp[i] = bitmap->data[i];
    w.bmpmskstride[i] = bitmap->stride[i];
}
w.dxChar = w.dx[0];
w.dyChar = w.dy[0];
}

if (bitmap->rectFull.dx != 0) {
    if (oldmode != cfg->mode)
        if (Tconfig::cpu_flags & FF_CPU_SSE2) {
            //获取叠加方式 (SSE2)
            //在 cfg 的 mode 里
            w.blend = getBlend<Tsse2>(oldmode = cfg->mode);
        } else {
            //获取叠加方式 (MMX)
            w.blend = getBlend<Tmmx>(oldmode = cfg->mode);
        }
    //输出到屏幕上的设置
    TprintPrefs prefs(deci, NULL);
    //各种参数
    prefs.dx = dx2[0];
    prefs.dy = dy2[0];
    prefs.xpos = cfg->posx;
    prefs.ypos = cfg->posy;
    //模式不同的话
    if (cfg->posmode == 1) {
        prefs.xpos *= -1;
        prefs.ypos *= -1;
    }
    prefs.align = cfg->align;
    prefs.linespacing = 100;
    prefs.csp = pict.csp;
    w.pict = &pict;
    w.cfg = cfg;
    //打印，需要用到 TprintPrefs
    ls.print(prefs, dst, stride2);
}

```

```

    }

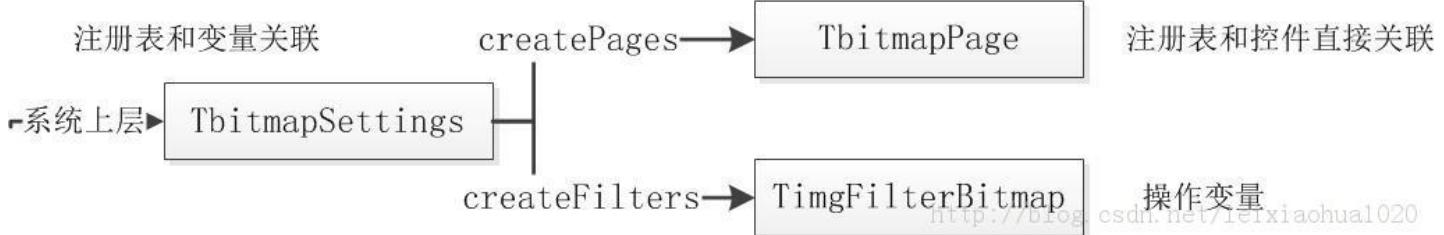
}

//最后都是这一句?
return parent->processSample(++it, pict);
}

```

## ffdshow 源代码分析 5：位图覆盖滤镜（总结）

用一张图总结他们之间的关系：



如图中所示，设置（Settings）部分是直接和系统上层关联的，它包含两个接口函数：createPages()和createFilters()。分别用于创建对话框（Dialog）和滤镜（Filter）。其中在 TbitmapPage 中对话框直接和注册表变量关联。而在 TbitmapSettings 中注册表变量和系统中的变量关联。TimgFilterBitmap 最终读取 TbitmapSettings 中的变量完成相应的操作。

目前来开 TimgFilterBitmap 是不会直接读取 TbitmapPage 类中的值的。

## ffdshow 源代码分析 6：对解码器的 dll 的封装（libavcodec）

ffdshow 封装了多个视音频解码器，比如 libmpeg2, libavcodec, xvid 等等。其中最重要的是 libavcodec，这个是 ffmpeg 提供的解码器，在 ffdshow 中起到了“挑大梁”的作用。本文分析 ffdshow 对解码器的封装方式，就以 libavcodec 为例。

在 ffdshow 中，libavcodec 的被封装在 ffmpeg.dll 文件中，通过加载该 dll 中的函数，就可以使用 libavcodec 的各种方法。

Ffmpeg 对 libavcodec 的封装类的定义位于 codecs->libavcodec->Tlibavcodec.h。实现则位于 codecs->libavcodec->Tlibavcodec.cpp。

先来看一看 Tlibavcodec.h：

```

#ifndef _TLIBAVCODEC_H_
#define _TLIBAVCODEC_H_
//将 FFmpeg 的 Dll 中的方法封装到一个类中，以供使用
#include "../codecs/ffcodecs.h"
#include <dxva.h>
#include "TpostprocSettings.h"
#include "ffImgfmt.h"
#include "libavfilter/vf_yadif.h"
#include "libavfilter/gradfun.h"

```

```
#include "libswscale/swscale.h"

struct AVCodecContext;
struct AVCodec;
struct AVFrame;
struct AVPacket;
struct AVCodecParserContext;
struct SwsContext;
struct SwsParams;
struct PPMode;
struct AVDictionary;

struct Tconfig;
class Tdll;
struct DSPContext;
struct TlibavcodecExt;
//封装 FFMPEG
//里面的函数基本上是 FFMPEG 的 API
struct Tlibavcodec {
private:
    int (*libswscale_sws_scale)(struct SwsContext *context, const uint8_t* const srcSlice[], const int srcStride[],
                               int srcSliceY, int srcSliceH, uint8_t* const dst[], const int dstStride[]);

    //加载 DLL 的类
    Tdll *dll;
    int refcount;
    static int get_buffer(AVCodecContext *c, AVFrame *pic);
    CCritSec csOpenClose;
public:
    Tlibavcodec(const Tconfig *config);
    ~Tlibavcodec();
    static void avlog(AVCodecContext*, int, const char*, va_list);
    static void avlogMsgBox(AVCodecContext*, int, const char*, va_list);
    void AddRef(void) {
        refcount++;
    }
    void Release(void) {
        if (--refcount < 0) {
            delete this;
        }
    }
    static bool getVersion(const Tconfig *config, ffstring &vers, ffstring &license);
    static bool check(const Tconfig *config);
    static int ppCpuCaps(uint64_t csp);
```

## 《FFmpeg 基础库编程开发》

```
static void pp_mode_defaults(PPMode &ppMode);
static int getPPmode(const TpostprocSettings *cfg, int currentq);
static void swsInitParams(SwsParams *params, int resizeMode);
static void swsInitParams(SwsParams *params, int resizeMode, int flags);

bool ok;
AVCodecContext* avcodec_alloc_context(AVCodec *codec, TlibavcodecExt *ext = NULL);

void (*avcodec_register_all)(void);
AVCodecContext* (*avcodec_alloc_context0)(AVCodec *codec);
AVCodec* (*avcodec_find_decoder)(AVCodecID codecId);
AVCodec* (*avcodec_find_encoder)(AVCodecID id);
int (*avcodec_open0)(AVCodecContext *avctx, AVCodec *codec, AVDictionary **options);
int avcodec_open(AVCodecContext *avctx, AVCodec *codec);
AVFrame* (*avcodec_alloc_frame)(void);
int (*avcodec_decode_video2)(AVCodecContext *avctx, AVFrame *picture,
                           int *got_picture_ptr,
                           AVPacket *avpkt);
int (*avcodec_decode_audio3)(AVCodecContext *avctx, int16_t *samples,
                           int *frame_size_ptr,
                           AVPacket *avpkt);
int (*avcodec_encode_video)(AVCodecContext *avctx, uint8_t *buf, int buf_size, const AVFrame *pict);
int (*avcodec_encode_audio)(AVCodecContext *avctx, uint8_t *buf, int buf_size, const short *samples);
void (*avcodec_flush_buffers)(AVCodecContext *avctx);
int (*avcodec_close0)(AVCodecContext *avctx);
int avcodec_close(AVCodecContext *avctx);

void (*av_log_set_callback)(void (*)(AVCodecContext*, int, const char*, va_list));
void* (*av_log_get_callback)(void);
int (*av_log_get_level)(void);
void (*av_log_set_level)(int);

void (*av_set_cpu_flags_mask)(int mask);

int (*avcodec_default_get_buffer)(AVCodecContext *s, AVFrame *pic);
void (*avcodec_default_release_buffer)(AVCodecContext *s, AVFrame *pic);
int (*avcodec_default_reget_buffer)(AVCodecContext *s, AVFrame *pic);
const char* (*avcodec_get_current_idct)(AVCodecContext *avctx);
void (*avcodec_get_encoder_info)(AVCodecContext *avctx, int *xvid_build, int *divx_version, int *divx_build, int
*lavc_build);

void* (*av_mallocz)(size_t size);
void (*av_free)(void *ptr);
```

```

AVCodecParserContext* (*av_parser_init)(int codec_id);
int (*av_parser_parse2)(AVCodecParserContext *s, AVCodecContext *avctx, uint8_t **poutbuf, int *poutbuf_size,
const uint8_t *buf, int buf_size, int64_t pts, int64_t dts, int64_t pos);
void (*av_parser_close)(AVCodecParserContext *s);

void (*av_init_packet)(AVPacket *pkt);
uint8_t* (*av_packet_new_side_data)(AVPacket *pkt, enum AVPacketSideDataType type, int size);

int (*avcodec_h264_search_recovery_point)(AVCodecContext *avctx,
                                         const uint8_t *buf, int buf_size, int *recovery_frame_cnt);

static const char_t *idctNames[], *errorRecognitions[], *errorConcealments[];
struct Tdia_size {
    int size;
    const char_t *descr;
};
static const Tdia_size dia_sizes[];

//libswscale imports
SwsContext* (*sws_getCachedContext)(struct SwsContext *context, int srcW, int srcH, enum PixelFormat srcFormat,
                                    int dstW, int dstH, enum PixelFormat dstFormat, int flags,
                                    SwsFilter *srcFilter, SwsFilter *dstFilter, const double *param,
SwsParams *ffdshow_params);

void (*sws_freeContext)(SwsContext *c);
SwsFilter* (*sws_getDefaultFilter)(float lumaGBlur, float chromaGBlur,
                                    float lumaSharpen, float chromaSharpen,
                                    float chromaHShift, float chromaVShift,
                                    int verbose);

void (*sws_freeFilter)(SwsFilter *filter);
int sws_scale(struct SwsContext *context, const uint8_t* const srcSlice[], const stride_t srcStride[],
              int srcSliceY, int srcSliceH, uint8_t* const dst[], const stride_t dstStride[]);
SwsVector *(*sws_getConstVec)(double c, int length);
SwsVector *(*sws_getGaussianVec)(double variance, double quality);
void (*sws_normalizeVec)(SwsVector *a, double height);
void (*sws_freeVec)(SwsVector *a);
int (*sws_setColorspaceDetails)(struct SwsContext *c, const int inv_table[4],
                               int srcRange, const int table[4], int dstRange,
                               int brightness, int contrast, int saturation);
const int* (*sws_getCoefficients)(int colorspace);

int (*GetCPUCount)(void);

```

```

//libpostproc imports
void (*pp_postprocess)(const uint8_t * src[3], const stride_t srcStride[3],
                      uint8_t * dst[3], const stride_t dstStride[3],
                      int horizontalSize, int verticalSize,
                      const /*QP_STORE_T*/int8_t *QP_store,  int QP_stride,
                      /*pp_mode*/void *mode, /*pp_context*/void *ppContext, int pict_type);

/*pp_context*/
void *(*pp_get_context)(int width, int height, int flags);
void (*pp_free_context)(/*pp_context*/void *ppContext);
void (*ff_simple_idct_mmx)(int16_t *block);

// DXVA imports
int (*av_h264_decode_frame)(struct AVCodecContext* avctx, uint8_t *buf, int buf_size);
int (*av_vc1_decode_frame)(struct AVCodecContext* avctx, uint8_t *buf, int buf_size);

// === H264 functions
int (*FFH264CheckCompatibility)(int nWidth, int nHeight, struct AVCodecContext* pAVCtx, BYTE* pBuffer, UINT
nSize, int nPCIVendor, int nPCIDevice, LARGE_INTEGER VideoDriverVersion);
int (*FFH264DecodeBuffer)(struct AVCodecContext* pAVCtx, BYTE* pBuffer, UINT nSize, int* pFramePOC, int*
pOutPOC, REFERENCE_TIME* pOutrtStart);
HRESULT(*FFH264BuildPicParams)(DXVA_PicParams_H264* pDXVAPicParams, DXVA_Qmatrix_H264*
pDXVAScalingMatrix, int* nFieldType, int* nSliceType, struct AVCodecContext* pAVCtx, int nPCIVendor);

void (*FFH264SetCurrentPicture)(int nIndex, DXVA_PicParams_H264* pDXVAPicParams, struct AVCodecContext*
pAVCtx);
void (*FFH264UpdateRefFramesList)(DXVA_PicParams_H264* pDXVAPicParams, struct AVCodecContext*
pAVCtx);
BOOL (*FFH264IsRefFrameInUse)(int nFrameNum, struct AVCodecContext* pAVCtx);
void (*FF264UpdateRefFrameSliceLong)(DXVA_PicParams_H264* pDXVAPicParams, DXVA_Slice_H264_Long*
pSlice, struct AVCodecContext* pAVCtx);
void (*FFH264SetDxvaSliceLong)(struct AVCodecContext* pAVCtx, void* pSliceLong);

// === VC1 functions
HRESULT(*FFVC1UpdatePictureParam)(DXVA_PictureParameters* pPicParams, struct AVCodecContext* pAVCtx,
int* nFieldType, int* nSliceType, BYTE* pBuffer, UINT nSize, UINT* nFrameSize, BOOL b_SecondField, BOOL*
b_repeat_pict);
int (*FFIIsSkipped)(struct AVCodecContext* pAVCtx);

// === Common functions
char* (*GetFFMpegPictureType)(int nType);
unsigned long(*FFGetMBNumber)(struct AVCodecContext* pAVCtx);

```

```

// yadif
void (*yadif_init)(YADIFContext *yadctx);
void (*yadif_uninit)(YADIFContext *yadctx);
void (*yadif_filter)(YADIFContext *yadctx, uint8_t *dst[3], stride_t dst_stride[3], int width, int height, int parity, int
tff);

// gradfun
int (*gradfunInit)(GradFunContext *ctx, const char *args, void *opaque);
void (*gradfunFilter)(GradFunContext *ctx, uint8_t *dst, uint8_t *src, int width, int height, int dst_linesize, int
src_linesize, int r);
};

#endif

```

从 Tlibavcodec 定义可以看出，里面包含了大量的 ffmpeg 中的 API，占据了很多的篇幅。通过调用这些 API，就可以使用 libavcodec 的各种功能。

在 Tlibavcodec 的定义中，有一个变量：Tdll \*dll，通过该变量，就可以加载 ffmpeg.dll 中的方法。

先来看一下 Tdll 的定义：

```

#ifndef _TDLL_H_
#define _TDLL_H_

#include "Tconfig.h"
//操作 Dll 的类
class Tdll
{
public:
    bool ok;
    Tdll(const char_t *dllName1, const Tconfig *config, bool explicitFullPath = false) {
        char_t name[MAX_PATH], ext[MAX_PATH];
        _splitpath_s(dllName1, NULL, 0, NULL, 0, name, countof(name), ext, countof(ext));
        if (config && !explicitFullPath) {
            char_t dllName2[MAX_PATH]; //installdir+filename+ext
            _makepath_s(dllName2, countof(dllName2), NULL, config->pth, name, ext);
            hdll = LoadLibrary(dllName2);
        } else {
            hdll = NULL;
        }
        if (!hdll) {
            hdll = LoadLibrary(dllName1);
            if (!hdll && !explicitFullPath) {
                if (config) {
                    char_t dllName3[MAX_PATH]; //ffdshow.ax_path+filename+ext
                    _makepath_s(dllName3, countof(dllName3), NULL, config->epth, name, ext);
                    hdll = LoadLibrary(dllName3);
                }
            }
        }
    }
};

```

```

        }
        if (!hdll) {
            char_t dllName0[MAX_PATH]; //only filename+ext - let Windows find it
            _makepath_s(dllName0, countof(dllName0), NULL, NULL, name, ext);
            hdll = LoadLibrary(dllName0);
        }
    }
}

ok = (hdll != NULL);

}

~Tdll() {
    if (hdll) {
        FreeLibrary(hdll);
    }
}

HMODULE hdll;
//封装一下直接加载 Dll 的 GetProcAddress
template<class T> __forceinline void loadFunction(T &fnc, const char *name) {
    fnc = hdll ? (T)GetProcAddress(hdll, name) : NULL;
    ok &= (fnc != NULL);
}

template<class T> __forceinline void loadFunctionByIndex(T &fnc, uint16_t id) {
    uint32_t id32 = uint32_t(id);
    fnc = hdll ?
        (T)GetProcAddress(hdll, (LPCSTR)id32) :
        NULL;
    ok &= (fnc != NULL);
}

//检查 Dll 的状态是否正常
static bool check(const char_t *dllName1, const Tconfig *config) {
    char_t name[MAX_PATH], ext[MAX_PATH];
    _splitpath_s(dllName1, NULL, 0, NULL, 0, name, countof(name), ext, countof(ext));
    if (config) {
        char_t dllName2[MAX_PATH]; //installdir+filename+ext
        _makepath_s(dllName2, countof(dllName2), NULL, config->pth, name, ext);
        if (fileexists(dllName2)) {
            return true;
        }
    }
    if (fileexists(dllName1)) {
        return true;
    }
    if (config) {
}

```

## 《FFmpeg 基础库编程开发》

```
char_t dllName3[MAX_PATH]; //ffdshow.ax_path+filename+ext
_makepath_s(dllName3, MAX_PATH, NULL, config->epth, name, ext);
if (fileexists(dllName3)) {
    return true;
}
}

char_t dllName0[MAX_PATH]; //only filename+ext - let Windows find it
_makepath_s(dllName0, countof(dllName0), NULL, NULL, name, ext);
char_t dir0[MAX_PATH], *dir0flnm;
if (SearchPath(NULL, dllName0, NULL, MAX_PATH, dir0, &dir0flnm)) {
    return true;
}
return false;
}

};

#endif
```

从 Tdll 的定义可以看出，该类的 loadFunction() 函数封装了系统使用 Dll 功能的函数 GetProcAddress()。该类的构造函数 Tdll() 封装了系统加载 Dll 的函数 LoadLibrary()。

此外该类还提供了 check() 用于检查 Dll。

对于 Tdll 的分析先告一段落，现在我们回到 Tlibavcodec，来看看它是如何加载 libavcodec 的函数的。查看一下 Tlibavcodec 的类的实现，位于 codecs->libavcodec->Tlibavcodec.cpp。

该类的实现代码比较长，因此只能选择重要的函数查看一下。首先来看一下构造函数：

```
===== Tlibavcodec =====
//FFMPEG 封装类的构造函数
Tlibavcodec::Tlibavcodec(const Tconfig *config): refcount(0)
{
    //加载 FFMPEG 的 Dll
    dll = new Tdll(_l("ffmpeg.dll"), config);
    //加载各个函数
    dll->loadFunction(avcodec_register_all, "avcodec_register_all");
    dll->loadFunction(avcodec_find_decoder, "avcodec_find_decoder");
    dll->loadFunction(avcodec_open0, "avcodec_open2");
    dll->loadFunction(avcodec_alloc_context0, "avcodec_alloc_context3");
    dll->loadFunction(avcodec_alloc_frame, "avcodec_alloc_frame");
    dll->loadFunction(avcodec_decode_video2, "avcodec_decode_video2");
    dll->loadFunction(avcodec_flush_buffers, "avcodec_flush_buffers");
    dll->loadFunction(avcodec_close0, "avcodec_close");
    dll->loadFunction(av_log_set_callback, "av_log_set_callback");
    dll->loadFunction(av_log_get_callback, "av_log_get_callback");
    dll->loadFunction(av_log_get_level, "av_log_get_level");
    dll->loadFunction(av_log_set_level, "av_log_set_level");
    dll->loadFunction(av_set_cpu_flags_mask, "av_set_cpu_flags_mask");
```

## 《FFmpeg 基础库编程开发》

```
dll->loadFunction(av_mallocz, "av_mallocz");
dll->loadFunction(av_free, "av_free");
dll->loadFunction(avcodec_default_get_buffer, "avcodec_default_get_buffer");
dll->loadFunction(avcodec_default_release_buffer, "avcodec_default_release_buffer");
dll->loadFunction(avcodec_default_reget_buffer, "avcodec_default_reget_buffer");
dll->loadFunction(avcodec_get_current_idct, "avcodec_get_current_idct");
dll->loadFunction(avcodec_get_encoder_info, "avcodec_get_encoder_info");
dll->loadFunction(av_init_packet, "av_init_packet");
dll->loadFunction(av_packet_new_side_data, "av_packet_new_side_data");
dll->loadFunction(avcodec_h264_search_recovery_point, "avcodec_h264_search_recovery_point");

dll->loadFunction(avcodec_decode_audio3, "avcodec_decode_audio3");

dll->loadFunction(avcodec_find_encoder, "avcodec_find_encoder");
dll->loadFunction(avcodec_encode_video, "avcodec_encode_video");
dll->loadFunction(avcodec_encode_audio, "avcodec_encode_audio");

dll->loadFunction(av_parser_init, "av_parser_init");
dll->loadFunction(av_parser_parse2, "av_parser_parse2");
dll->loadFunction(av_parser_close, "av_parser_close");

//libswscale methods
dll->loadFunction(sws_getCachedContext, "sws_getCachedContext");
dll->loadFunction(sws_freeContext, "sws_freeContext");
dll->loadFunction(sws_getDefaultFilter, "sws_getDefaultFilter");
dll->loadFunction(sws_freeFilter, "sws_freeFilter");
dll->loadFunction(libswscale_sws_scale, "sws_scale");

dll->loadFunction(GetCPUCount, "GetCPUCount");
dll->loadFunction(sws_getConstVec, "sws_getConstVec");
dll->loadFunction(sws_getGaussianVec, "sws_getGaussianVec");
dll->loadFunction(sws_normalizeVec, "sws_normalizeVec");
dll->loadFunction(sws_freeVec, "sws_freeVec");
dll->loadFunction(sws_setColorspaceDetails, "swsSetColorspaceDetails");
dll->loadFunction(sws_getCoefficients, "sws_getCoefficients");

//libpostproc methods
dll->loadFunction(pp_postprocess, "pp_postprocess");
dll->loadFunction(pp_get_context, "pp_get_context");
dll->loadFunction(pp_free_context, "pp_free_context");
dll->loadFunction(ff_simple_idct_mmx, "ff_simple_idct_mmx");

//DXVA methods
```

《FFmpeg 基础库编程开发》

```

dll->loadFunction(av_h264_decode_frame, "av_h264_decode_frame");
dll->loadFunction(av_vc1_decode_frame, "av_vc1_decode_frame");

dll->loadFunction(FFH264CheckCompatibility, "FFH264CheckCompatibility");
dll->loadFunction(FFH264DecodeBuffer, "FFH264DecodeBuffer");
dll->loadFunction(FFH264BuildPicParams, "FFH264BuildPicParams");
dll->loadFunction(FFH264SetCurrentPicture, "FFH264SetCurrentPicture");
dll->loadFunction(FFH264UpdateRefFramesList, "FFH264UpdateRefFramesList");
dll->loadFunction(FFH264IsRefFrameInUse, "FFH264IsRefFrameInUse");
dll->loadFunction(FF264UpdateRefFrameSliceLong, "FF264UpdateRefFrameSliceLong");
dll->loadFunction(FFH264SetDxvaSliceLong, "FFH264SetDxvaSliceLong");

dll->loadFunction(FFVC1UpdatePictureParam, "FFVC1UpdatePictureParam");
dll->loadFunction(FFIsSkipped, "FFIsSkipped");

dll->loadFunction(GetFFMpegPictureType, "GetFFMpegPictureType");
dll->loadFunction(FFGetMBNumber, "FFGetMBNumber");

//yadif methods
dll->loadFunction(yadif_init, "yadif_init");
dll->loadFunction(yadif_uninit, "yadif_uninit");
dll->loadFunction(yadif_filter, "yadif_filter");

//gradfun
dll->loadFunction(gradfunInit, "gradfunInit");
dll->loadFunction(gradfunFilter, "gradfunFilter");

ok = dll->ok;
//加载完毕后，进行注册
if (ok) {
    avcodec_register_all();
    av_log_set_callback(avlog);
}
}

该构造函数尽管篇幅比较长，但是还是比较好理解的，主要完成了3步：
1. 创建一个Tdll类的对象，加载“ffmpeg.dll”。
2. 使用loadFunction()加载各种函数。
3. 最后调用avcodec_register_all()注册各种解码器。

```

Tlibavcodec 的析构函数则比较简单：

```

Tlibavcodec::~Tlibavcodec()
{
    delete dll;
}
```

```
}
```

检查 Dll 的函数也比较简单：

```
[cpp] view plaincopy
bool Tlibavcodec::check(const Tconfig *config)
{
    return Tdll::check(_l("ffmpeg.dll"), config);
}
```

此外，可能是出于某些功能的考虑，ffdshow 还自己写了几个函数，但是限于篇幅不能一一介绍，在这里只介绍一个：

获取 libavcodec 版本：

```
bool Tlibavcodec::getVersion(const Tconfig *config, ffstring &vers, ffstring &license)
```

```
{
    Tdll *dl = new Tdll(_l("ffmpeg.dll"), config);

    void (*av_getVersion)(char **version, char **build, char **datetime, const char* *license);
    dl->loadFunction(av_getVersion, "getVersion");
    bool res;
    if (av_getVersion) {
        res = true;
        char *version, *build, *datetime;
        const char *lic;
        av_getVersion(&version, &build, &datetime, &lic);
        vers = (const char_t*)text<char_t>(version) + ffstring(_l(" (")) + (const char_t*)text<char_t>(datetime) + _l(")");
        license = text<char_t>(lic);
    } else {
        res = false;
        vers.clear();
        license.clear();
    }
    delete dl;
    return res;
}
```

ffdshow 源代码分析 7： libavcodec 视频解码器类（TvideoCodecLibavcodec）

在这里我们进一步介绍一下其 libavcodec 解码器类。注意前一篇文章介绍的类 Tlibavcodec 仅仅是对 libavcodec 所在的“ffmpeg.dll”的函数进行封装的类。但 Tlibavcodec 并不是一个解码器类，其没有继承任何类，还不能为 ffdshow 所用。本文介绍的 TvideoCodecLibavcodec 才是 libavcodec 解码器类，其继承了 TvideoCodecDec。

先来看一看 TvideoCodecLibavcodec 的定义吧，位于 codecs-> TvideoCodecLibavcodec.h 中。

```
#ifndef _TVIDEOCODECLEIBAVCODEC_H_
#define _TVIDEOCODECLEIBAVCODEC_H_
```

```
#include "TvideoCodec.h"
#include "ffmpeg/Tlibavcodec.h"
```

## 《FFmpeg 基础库编程开发》

```
#include "ffmpeg/libavcodec/avcodec.h"

#define MAX_THREADS 8 // FIXME: This is defined in mpegvideo.h.

struct Textradata;
class TccDecoder;
//libavcodec 解码器（视频）
struct TlibavcodecExt {
private:
    static int get_buffer(AVCodecContext *s, AVFrame *pic);
    int (*default_get_buffer)(AVCodecContext *s, AVFrame *pic);
    static void release_buffer(AVCodecContext *s, AVFrame *pic);
    void (*default_release_buffer)(AVCodecContext *s, AVFrame *pic);
    static int reget_buffer(AVCodecContext *s, AVFrame *pic);
    int (*default_reget_buffer)(AVCodecContext *s, AVFrame *pic);
    static void handle_user_data0(AVCodecContext *c, const uint8_t *buf, int buf_len);
public:
    virtual ~TlibavcodecExt() {}
    void connectTo(AVCodecContext *ctx, Tlibavcodec *libavcodec);
    virtual void onGetBuffer(AVFrame *pic) {}
    virtual void onRegetBuffer(AVFrame *pic) {}
    virtual void onReleaseBuffer(AVFrame *pic) {}
    virtual void handle_user_data(const uint8_t *buf, int buf_len) {}
};

//libavcodec 解码，不算是 Filter？
class TvideoCodecLibavcodec : public TvideoCodecDec, public TvideoCodecEnc, public TlibavcodecExt
{
    friend class TDXVADecoderVC1;
    friend class TDXVADecoderH264;
protected:
    //各种信息（源自 AVCodecContext 中）
    Tlibavcodec *libavcodec;
    void create(void);
    AVCodec *avcodec;
    mutable char_t codecName[100];
    AVCodecContext *avctx;
    uint32_t palette[AVPALETTE_COUNT];
    int palette_size;
    AVFrame *frame;
    FOURCC fcc;
    FILE *statsfile;
    int cfgcomode;
    int psnr;
```

```

bool isAdaptive;
int threadcount;
bool dont_use_rtStop_from_upper_stream; // and reordering of timestamps is justified.
bool theorart;
bool codecinitd, ownmatrices;
REFERENCE_TIME rtStart, rtStop, avgTimePerFrame, segmentTimeStart;
REFERENCE_TIME prior_in_rtStart, prior_in_rtStop;
REFERENCE_TIME prior_out_rtStart, prior_out_rtStop;

struct {
    REFERENCE_TIME rtStart, rtStop;
    unsigned int srcSize;
} b[MAX_THREADS + 1];
int inPosB;

Textradata *extradata;
bool sendextradata;
unsigned int mb_width, mb_height, mb_count;
static void line(unsigned char *dst, unsigned int _x0, unsigned int _y0, unsigned int _x1, unsigned int _y1, stride_t
strideY);
static void draw_arrow(uint8_t *buf, int sx, int sy, int ex, int ey, stride_t stride, int mulx, int muly, int dstdx, int dstdy);
unsigned char *ffbuf;
unsigned int ffbuflen;
bool wasKey;
virtual void handle_user_data(const uint8_t *buf, int buf_len);
TccDecoder *ccDecoder;
bool autoSkipingLoopFilter;
enum AVDiscard initialSkipLoopFilter;
int got_picture;
bool firstSeek; // firstSeek means start of playback.
bool mpeg2_in_doubt;
bool mpeg2_new_sequence;
bool bReorderBFrame;
//时长 (AVCodecContext 中)
REFERENCE_TIME getDuration();
int isReallyMPEG2(const unsigned char *src, size_t srcLen);

protected:
    virtual LRESULT beginCompress(int cfgcomode, uint64_t csp, const Trect &r);
    virtual bool beginDecompress(TffPictBase &pict, FOURCC infcc, const CMediaType &mt, int sourceFlags);
    virtual HRESULT flushDec(void);
    AVCodecParserContext *parser;

public:
    TvideoCodecLibavcodec(IffdshowBase *Ideci, IdecVideoSink *IsinkD);

```

## 《FFmpeg 基础库编程开发》

```
TvideoCodecLibavcodec(IffdshowBase *IdecI, IencVideoSink *IsinkE);
virtual ~TvideoCodecLibavcodec();
virtual int getType(void) const {
    return IDFF_MOVIE_LAVC;
}
virtual const char_t* getName(void) const;
virtual int caps(void) const {
    return CAPS::VIS_MV | CAPS::VIS_QUANTS;
}

virtual void end(void);

virtual void getCompressColorspaces(TcspS &cspS, unsigned int outDx, unsigned int outDy);
virtual bool supExtradata(void);
//获得 ExtraData (AVCodecContext 中)
virtual bool getExtradata(const void* *ptr, size_t *len);
virtual HRESULT compress(const TffPict &pict, TencFrameParams ¶ms);
virtual HRESULT flushEnc(const TffPict &pict, TencFrameParams ¶ms) {
    return compress(pict, params);
}

virtual HRESULT decompress(const unsigned char *src, size_t srcLen, IMediaSample *pIn);
virtual void onGetBuffer(AVFrame *pic);
virtual bool onSeek(REFERENCE_TIME segmentStart);
virtual bool onDiscontinuity(void);
//画出运动矢量 (AVCodecContext 中)
virtual bool drawMV(unsigned char *dst, unsigned int dx, stride_t stride, unsigned int dy) const;
//编码器信息 (AVCodecContext 中)
virtual void getEncoderInfo(char_t *buf, size_t buflen) const;
virtual const char* get_current_idct(void);
virtual HRESULT BeginFlush();
bool isReorderBFrame() {
    return bReorderBFrame;
};
virtual void reorderBFrames(REFERENCE_TIME& rtStart, REFERENCE_TIME& rtStop);

class Th264RandomAccess
{
    friend class TvideoCodecLibavcodec;
private:
    TvideoCodecLibavcodec* parent;
    int recovery_mode; // 0:OK, 1:searching 2: found, 3:waiting for frame_num decoded, 4:waiting for POC
outputed
```

## 《FFmpeg 基础库编程开发》

```
int recovery_frame_cnt;
int recovery_poc;
int thread_delay;

public:
    Th264RandomAccess(TvideoCodecLibavcodec* Iparent);
    int search(uint8_t* buf, int buf_size);
    void onSeek(void);
    void judgeUsability(int *got_picture_ptr);
} h264RandomAccess;
};

#endif
```

这里有一个类 TlibavcodecExt，我觉得应该是扩展了 Tlibavcodec 的一些功能，在这里我们先不管它，直接看看 TvideoCodecLibavcodec 都包含了什么变量：

Tlibavcodec \*libavcodec：该类封装了 libavcodec 的各种函数，在前一篇文章中已经做过介绍，在此不再重复叙述了。可以认为该变量是 TvideoCodecLibavcodec 类的灵魂，所有 libavcodec 中的函数都是通过该类调用的。

AVCodec \*avcodec：FFMPEG 中的结构体，解码器

AVCodecContext \*avctx：FFMPEG 中的结构体，解码器上下文

AVFrame \*frame FFMPEG 中的结构体，视频帧

mutable char\_t codecName[100]：解码器名称

FOURCC fcc：FourCC

Textradata \*extradata：附加数据

...

再来看一下 TvideoCodecLibavcodec 都包含什么方法：

create()：创建解码器的时候调用

getDuration()：获得时长

getExtradata()：获得附加数据

drawMV()：画运动矢量

getEncoderInfo()：获得编码器信息

此外还包括一些有关解码的方法【这个是最关键的】：

beginDecompress()：解码初始化

decompress()：解码

下面我们来详细看看这些函数的实现吧：

先来看一下 TvideoCodecLibavcodec 的构造函数：

//libavcodec 解码器（视频）

//内容大部分都很熟悉，因为是 FFmpeg 的 API

TvideoCodecLibavcodec::TvideoCodecLibavcodec(IffdshowBase \*IdecI, IdecVideoSink \*IsinkD):

Tcodec(IdecI), TcodecDec(IdecI, IsinkD),

TvideoCodec(IdecI),

TvideoCodecDec(IdecI, IsinkD),

TvideoCodecEnc(IdecI, NULL),

h264RandomAccess(this),

## 《FFmpeg 基础库编程开发》

```
bReorderBFrame(true)
```

```
{  
    create();  
}
```

可见构造函数调用了 Create(), 我们再来看看 Create():

```
void TvideoCodecLibavcodec::create(void)
```

```
{  
    ownmatrices = false;  
    deci->getLibavcodec(&libavcodec);  
    ok = libavcodec ? libavcodec->ok : false;  
    avctx = NULL;  
    avcodec = NULL;  
    frame = NULL;  
    quantBytes = 1;  
    statsfile = NULL;  
    threadcount = 0;  
    codecinitied = false;  
    extradata = NULL;  
    theorart = false;  
    ffbuff = NULL;  
    ffbuflen = 0;  
    codecName[0] = '\0';  
    ccDecoder = NULL;  
    autoSkipingLoopFilter = false;  
    inPosB = 1;  
    firstSeek = true;  
    mpeg2_new_sequence = true;  
    parser = NULL;  
}
```

从 Create() 函数我们可以看出，其完成了各种变量的初始化工作。其中有一行代码：

```
deci->getLibavcodec(&libavcodec);
```

完成了 Tlibavcodec\* libavcodec 的初始化工作。

再来看几个函数。

getDuration(), 用于从 AVCodecContext 中获取时长：

```
REFERENCE_TIME TvideoCodecLibavcodec::getDuration()
```

```
{  
    REFERENCE_TIME duration = REF_SECOND_MULT / 100;  
    if (avctx && avctx->time_base.num && avctx->time_base.den) {  
        duration = REF_SECOND_MULT * avctx->time_base.num / avctx->time_base.den;  
        if (codecId == AV_CODEC_ID_H264) {  
            duration *= 2;  
        }  
    }  
}
```

## 《FFmpeg 基础库编程开发》

```
if (duration == 0) {
    return REF_SECOND_MULT / 100;
}
return duration;
}

getExtradata()用于从 AVCodecContext 中获取附加信息：
bool TvideoCodecLibavcodec::getExtradata(const void* *ptr, size_t *len)
{
    if (!avctx || !len) {
        return false;
    }
    *len = avctx->extradata_size;
    if (ptr) {
        *ptr = avctx->extradata;
    }
    return true;
}

drawMV()用于从 AVFrame 中获取运动矢量信息，并画出来（这个函数用于一个名为“可视化”的滤镜里面，用于显示视频的运动矢量信息）。
//画出运动矢量
bool TvideoCodecLibavcodec::drawMV(unsigned char *dst, unsigned int dstdx, stride_t stride, unsigned int dstdy) const
{
    if (!frame->motion_val || !frame->mb_type || !frame->motion_val[0]) {
        return false;
    }

#define IS_8X8(a) ((a)&MB_TYPE_8x8)
#define IS_16X8(a) ((a)&MB_TYPE_16x8)
#define IS_8X16(a) ((a)&MB_TYPE_8x16)
#define IS_INTERLACED(a) ((a)&MB_TYPE_INTERLACED)
#define USES_LIST(a, list) ((a) & ((MB_TYPE_P0L0|MB_TYPE_P1L0)<<(2*(list))))


    const int shift = 1 + ((frame->play_flags & CODEC_FLAG_QPEL) ? 1 : 0);
    const int mv_sample_log2 = 4 - frame->motion_subsample_log2;
    const int mv_stride = (frame->mb_width << mv_sample_log2) + (avctx->codec_id == AV_CODEC_ID_H264 ? 0 : 1);
    int direction = 0;

    int mulx = (dstdx << 12) / avctx->width;
    int muly = (dstdy << 12) / avctx->height;
    //提取两个方向上的运动矢量信息（根据不同的宏块划分，可以分成几种情况）
    //在 AVCodecContext 的 motion_val 中
    for (int mb_y = 0; mb_y < frame->mb_height; mb_y++) {
        for (int mb_x = 0; mb_x < frame->mb_width; mb_x++) {
```

## 《FFmpeg 基础库编程开发》

```
const int mb_index = mb_x + mb_y * frame->mb_stride;
if (!USES_LIST(frame->mb_type[mb_index], direction)) {
    continue;
}
...此处代码太长，略
}

#define IS_8X8
#define IS_16X8
#define IS_8X16
#define IS_INTERLACED
#define USES_LIST

return true;
}
```

下面来看几个很重要的函数，这几个函数继承自 TvideoCodecDec 类。

beginDecompress()用于解码器的初始化。注：这个函数的代码太长了，因此只选择一点关键的代码。

```
//----- decompression -----
bool TvideoCodecLibavcodec::beginDecompress(TffPictBase &pict, FOURCC fcc, const CMediaType &mt, int sourceFlags)
{
    palette_size = 0;
    prior_out_rtStart = REFTIME_INVALID;
    prior_out_rtStop = 0;
    rtStart = rtStop = REFTIME_INVALID;
    prior_in_rtStart = prior_in_rtStop = REFTIME_INVALID;
    mpeg2_in_doubt = codecId == AV_CODEC_ID_MPEG2VIDEO;

    int using_dxva = 0;

    int numthreads = deci->getParam2(IDFF_numLAVCdecThreads);
    int thread_type = 0;
    if (numthreads > 1 && sup_threads_dec_frame(codecId)) {
        thread_type = FF_THREAD_FRAME;
    } else if (numthreads > 1 && sup_threads_dec_slice(codecId)) {
        thread_type = FF_THREAD_SLICE;
    }

    if (numthreads > 1 && thread_type != 0) {
        threadcount = numthreads;
    } else {
        threadcount = 1;
    }

    if (codecId == CODEC_ID_H264_DXVA) {
```

## 《FFmpeg 基础库编程开发》

```
codecId = AV_CODEC_ID_H264;
using_dxva = 1;
} else if (codecId == CODEC_ID_VC1_DXVA) {
    codecId = AV_CODEC_ID_VC1;
    using_dxva = 1;
}

avcodec = libavcodec->avcodec_find_decoder(codecId);
if (!avcodec) {
    return false;
}
avctx = libavcodec->avcodec_alloc_context(avcodec, this);
avctx->thread_type = thread_type;
avctx->thread_count = threadcount;
avctx->h264_using_dxva = using_dxva;
if (codecId == AV_CODEC_ID_H264) {
    // If we do not set this, first B-frames before the IDR pictures are dropped.
    avctx->has_b_frames = 1;
}

frame = libavcodec->avcodec_alloc_frame();
avctx->width = pict.rectFull.dx;
avctx->height = pict.rectFull.dy;
intra_matrix = avctx->intra_matrix = (uint16_t*)calloc(sizeof(uint16_t), 64);
inter_matrix = avctx->inter_matrix = (uint16_t*)calloc(sizeof(uint16_t), 64);
ownmatrices = true;

// Fix for new Haali custom media type and fourcc. ffmpeg does not understand it, we have to change it to
FOURCC_AV1
if (fcc == FOURCC_CCV1) {
    fcc = FOURCC_AV1;
}

avctx->codec_tag = fcc;
avctx->workaround_bugs = deci->getParam2(IDFF_workaroundBugs);
#if 0
avctx->error_concealment = FF_EC_GUESS_MVS | FF_EC_DEBLOCK;
avctx->err_recognition      = AV_EF_CRCCHECK | AV_EF_BITSTREAM | AV_EF_BUFFER |
AV_EF_COMPLIANT | AV_EF.Aggressive;
#endif
if (codecId == AV_CODEC_ID_MJPEG) {
    avctx->flags |= CODEC_FLAG_TRUNCATED;
```

## 《FFmpeg 基础库编程开发》

```
}

if (mpeg12_codec(codecId) && deci->getParam2(IDFF_fastMpeg2)) {
    avctx->flags2 = CODEC_FLAG2_FAST;
}

if (codecId == AV_CODEC_ID_H264)
    if (int skip = deci->getParam2(IDFF_fastH264)) {
        avctx->skip_loop_filter = skip & 2 ? AVDISCARD_ALL : AVDISCARD_NONREF;
    }
initialSkipLoopFilter = avctx->skip_loop_filter;

avctx->debug_mv = !using_dxva; // (deci->getParam2(IDFF_isVis) & deci->getParam2(IDFF_visMV));

avctx->idct_algo = limit(deci->getParam2(IDFF_idct), 0, 6);
if (extradata) {
    delete extradata;
}
extradata = new Textradata(mt, FF_INPUT_BUFFER_PADDING_SIZE);
此处代码太长，略...
}
```

从代码中可以看出这个函数的流程是：

- 1.avcodec\_find\_decoder();
- 2.avcodec\_alloc\_context();
- 3.avcodec\_alloc\_frame();
- 4.avcodec\_open();

主要做了 libavcodec 初始化工作。

begin decompress() 用于解码器的初始化。注：这个函数的代码太长了，因此只选择一点关键的代码。

```
HRESULT TvideoCodecLibavcodec::decompress(const unsigned char *src, size_t srcLen0, IMediaSample *pIn)
{
    代码太长，略...
    AVPacket avpkt;
    libavcodec->av_init_packet(&avpkt);
    if (palette_size) {
        uint32_t *pal = (uint32_t *)libavcodec->av_packet_new_side_data(&avpkt, AV_PKT_DATA_PALETTE,
AVPALETTE_SIZE);
        for (int i = 0; i < palette_size / 4; i++) {
            pal[i] = 0xFF << 24 | AV_RL32(palette + i);
        }
    }

    while (!src || size > 0) {
        int used_bytes;

        avctx->reordered_opaque = rtStart;
```

## 《FFmpeg 基础库编程开发》

```
avctx->reordered_opaque2 = rtStop;
avctx->reordered_opaque3 = size;

if (sendextradata && extradata->data && extradata->size > 0) {
    avpkt.data = (uint8_t *)extradata->data;
    avpkt.size = (int)extradata->size;
    used_bytes = libavcodec->avcodec_decode_video2(avctx, frame, &got_picture, &avpkt);
    sendextradata = false;
    if (used_bytes > 0) {
        used_bytes = 0;
    }
    if (mpeg12_codec(codecId)) {
        avctx->extradata = NULL;
        avctx->extradata_size = 0;
    }
} else {
    unsigned int neededsize = size + FF_INPUT_BUFFER_PADDING_SIZE;

    if (ffbuflen < neededsize) {
        ffbuf = (unsigned char*)realloc(ffbuf, ffbuflen = neededsize);
    }

    if (src) {
        memcpy(ffbuf, src, size);
        memset(ffbuf + size, 0, FF_INPUT_BUFFER_PADDING_SIZE);
    }
    if (parser) {
        uint8_t *outBuf = NULL;
        int out_size = 0;
        used_bytes = libavcodec->av_parser_parse2(parser, avctx, &outBuf, &out_size, src ? ffbuf : NULL, size,
AV_NOPTS_VALUE, AV_NOPTS_VALUE, 0);
        if (prior_in_rtStart == REFTIME_INVALID) {
            prior_in_rtStart = rtStart;
            prior_in_rtStop = rtStop;
        }
        if (out_size > 0 || !src) {
            mpeg2_in_doubt = false;
            avpkt.data = out_size > 0 ? outBuf : NULL;
            avpkt.size = out_size;
            if (out_size > used_bytes) {
                avctx->reordered_opaque = prior_in_rtStart;
                avctx->reordered_opaque2 = prior_in_rtStop;
            } else {

```

《FFmpeg 基础库编程开发》

```

avctx->reordered_opaque = rtStart;
avctx->reordered_opaque2 = rtStop;
}
prior_in_rtStart = rtStart;
prior_in_rtStop = rtStop;
avctx->reordered_opaque3 = out_size;
if (h264RandomAccess.search(avpkt.data, avpkt.size)) {
    libavcodec->avcodec_decode_video2(avctx, frame, &got_picture, &avpkt);
    h264RandomAccess.judgeUsability(&got_picture);
} else {
    got_picture = 0;
}
} else {
    got_picture = 0;
}
} else {
    avpkt.data = src ? ffbuff : NULL;
    avpkt.size = size;
    if (codecId == AV_CODEC_ID_H264) {
        if (h264RandomAccess.search(avpkt.data, avpkt.size)) {
            used_bytes = libavcodec->avcodec_decode_video2(avctx, frame, &got_picture, &avpkt);
            if (used_bytes < 0) {
                return S_OK;
            }
            h264RandomAccess.judgeUsability(&got_picture);
        } else {
            got_picture = 0;
            return S_OK;
        }
    } else {
        used_bytes = libavcodec->avcodec_decode_video2(avctx, frame, &got_picture, &avpkt);
    }
}
}
}

代码太长，略...
}

```

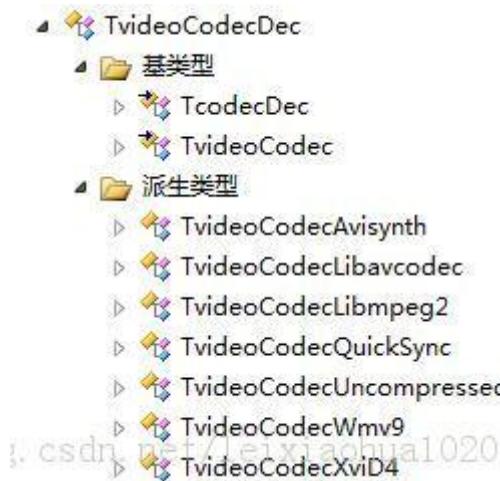
从代码中可以看出这个函数的流程是：

- 1.AVPacket avpkt;
  - 2.av\_init\_packet();
  - 3.avcodec\_decode\_video2();
- 和 ffmpeg 的解码流程相差不大。

## ffdshow 源代码分析 8：视频解码器类（TvideoCodecDec）

其中 libavcodec 的解码器类 TvideoCodecLibavcodec 通过调用 Tlibavcodec 中的方法实现了 libavcodec 的 dll 中方法的调用；而它继承了 TvideoCodecDec，本文正是要分析它继承的这个类。

TvideoCodecDec 是所有视频解码器共有的父类。可以看一下它的继承关系：



可见，除了 TvideoCodecLibavcodec 继承了 TvideoCodecDec 之外，还有好几个类继承了 TvideoCodecDec，比如说：TvideoCodecLibmpeg2，TvideoCodecXviD4 等等...。突然来了兴趣，我们可以看一下其他的解码器类的定义是什么样的。

TvideoCodecLibmpeg2 定义如下：

```
#ifndef _TVIDEOCODECLEBMPEG2_H_
#define _TVIDEOCODECLEBMPEG2_H_
```

```
#include "TvideoCodec.h"
#include "libmpeg2/include/mpeg2.h"

class Tdll;
struct Textradata;
class TccDecoder;
//libmpeg2 解码器
class TvideoCodecLibmpeg2 : public TvideoCodecDec
{
private:
    Tdll *dll;
    uint32_t (*mpeg2_set_accel)(uint32_t accel);
    mpeg2dec_t* (*mpeg2_init)(void);
    const mpeg2_info_t* (*mpeg2_info)(mpeg2dec_t *mpeg2dec);
    mpeg2_state_t (*mpeg2_parse)(mpeg2dec_t *mpeg2dec);
    void (*mpeg2_buffer)(mpeg2dec_t *mpeg2dec, const uint8_t *start, const uint8_t *end);
    void (*mpeg2_close)(mpeg2dec_t *mpeg2dec);
```

## 《FFmpeg 基础库编程开发》

```
void (*mpeg2_reset)(mpeg2dec_t *mpeg2dec, int full_reset);
void (*mpeg2_set_rtStart)(mpeg2dec_t *mpeg2dec, int64_t rtStart);
int (*mpeg2_guess_aspect)(const mpeg2_sequence_t * sequence, unsigned int * pixel_width, unsigned int * pixel_height);

mpeg2dec_t *mpeg2dec;
const mpeg2_info_t *info;
bool wait4Iframe;
int sequenceFlag;
REFERENCE_TIME avgTimePerFrame;
TffPict oldpict;
Textradata *extradata;
TccDecoder *ccDecoder;
Tbuffer *buffer;
uint32_t oldflags;
bool m_fFilm;
int SetDeinterlaceMethod(void);

void init(void);
HRESULT decompressI(const unsigned char *src, size_t srcLen, IMediaSample *pIn);

protected:
    virtual bool beginDecompress(TffPictBase &pict, FOURCC infcc, const CMediaType &mt, int sourceFlags);

public:
    TvideoCodecLibmpeg2(IffdshowBase *Ideci, IdecVideoSink *Isink);
    virtual ~TvideoCodecLibmpeg2();

    static const char_t *dllname;
    virtual int getType(void) const {
        return IDFF_MOVIE_LIBMPEG2;
    }
    virtual int caps(void) const {
        return CAPS::VIS_QUANTS;
    }

    virtual void end(void);
    virtual HRESULT decompress(const unsigned char *src, size_t srcLen, IMediaSample *pIn);
    virtual bool onSeek(REFERENCE_TIME segmentStart);
    virtual HRESULT BeginFlush();
};

#endif
```

TvideoCodecXviD4 定义如下：

```
#ifndef _TVIDEOCODECXVID4_H_
#define _TVIDEOCODECXVID4_H_

#include "TvideoCodec.h"

class Tdll;
struct Textradata;
//xvid 解码器
class TvideoCodecXviD4 : public TvideoCodecDec
{
private:
    void create(void);
    Tdll *dll;
public:
    TvideoCodecXviD4(IffdshowBase *IdecI, IdecVideoSink *IsinkD);
    virtual ~TvideoCodecXviD4();
    int (*xvid_global)(void *handle, int opt, void *param1, void *param2);
    int (*xvid_decore)(void *handle, int opt, void *param1, void *param2);
    int (*xvid_plugin_single)(void *handle, int opt, void *param1, void *param2);
    int (*xvid_plugin_lumimasking)(void *handle, int opt, void *param1, void *param2);
    static const char_t *dllname;
private:
    void *enchandle, *dechandle;
    int psnr;
    TffPict pict;
    Tbuffer pictbuf;
    static int me_hq(int rd3), me_(int me3);
    Textradata *extradata;
    REFERENCE_TIME rtStart, rtStop;
protected:
    virtual bool beginDecompress(TffPictBase &pict, FOURCC infcc, const CMediaType &mt, int sourceFlags);
    virtual HRESULT flushDec(void);
public:
    virtual int getType(void) const {
        return IDFF_MOVIE_XVID4;
    }
    virtual int caps(void) const {
        return CAPS::VIS_QUANTS;
    }
    virtual HRESULT decompress(const unsigned char *src, size_t srcLen, IMediaSample *pIn);
};
```

#endif

从以上这 2 种解码器类的定义，我们可以看出一些规律，比如说：

1. 都有 Tdll \*dll 这个变量，用于加载视频解码器的 dll
2. 都有 beginDecompress() 函数，用于初始化解码器
3. 都有 decompress() 函数，用于解码

好了，闲话不说，回归正题，来看一下这些解码器共有的父类：TvideoCodecDec

//具体 视频 解码器的父类，存一些公共信息

```
class TvideoCodecDec : virtual public TvideoCodec, virtual public TcodecDec
```

```
{
```

```
protected:
```

```
    bool isdvdproc;
    comptrQ<IffdshowDecVideo> deciV;
    IdecVideoSink *sinkD;
    TvideoCodecDec(IffdshowBase *Ideci, IdecVideoSink *Isink);
    Rational guessMPEG2sar(const Trect &r, const Rational &sar2, const Rational &containerSar);
```

```
class TtelecineManager
```

```
{
```

```
private:
```

```
    TvideoCodecDec* parent;
    int segment_count;
    int pos_in_group;
    struct {
        int fieldtype;
        int repeat_pict;
        REFERENCE_TIME rtStart;
    } group[2]; // store information about 2 recent frames.
```

```
    REFERENCE_TIME group_rtStart;
```

```
    bool film;
```

```
    int cfg_softTelecine;
```

```
public:
```

```
    TtelecineManager(TvideoCodecDec* Iparent);
```

```
    void get_timestamps(TffPict &pict);
```

```
    void get_fieldtype(TffPict &pict);
```

```
    void new_frame(int top_field_first, int repeat_pict, const REFERENCE_TIME &rtStart, const
REFERENCE_TIME &rtStop);
```

```
    void onSeek(void);
```

```
} telecineManager;
```

```
public:
```

```
static TvideoCodecDec* initDec(IffdshowBase *deci, IdecVideoSink *Isink, AVCodecID codecId, FOURCC fcc, const
```

```
CMediaType &mt);
```

```
virtual ~TvideoCodecDec();

virtual int caps(void) const {
    return CAPS::NONE;
}

virtual bool testMediaType(FOURCC fcc, const CMediaType &mt) {
    return true;
}

virtual void forceOutputColorspace(const BITMAPINFOHEADER *hdr, int *ilace, TcspInfos &forcedCsp) {
    *ilace = 0; //cspInfos of forced output colorspace, empty when entering function
}

enum {SOURCE_REORDER = 1};

virtual bool beginDecompress(TffPictBase &pict, FOURCC infcc, const CMediaType &mt, int sourceFlags) = 0;
virtual HRESULT decompress(const unsigned char *src, size_t srcLen, IMediaSample *pIn) = 0;
virtual bool onDiscontinuity(void) {
    return false;
}

virtual HRESULT onEndOfStream(void) {
    return S_OK;
}

unsigned int quantsDx, quantsStride, quantsDy, quantBytes, quantType;
//QP 表
void *quants;
uint16_t *intra_matrix, *inter_matrix;
//计算平均 QP
float calcMeanQuant(void);
//画运动矢量
virtual bool drawMV(unsigned char *dst, unsigned int dx, stride_t stride, unsigned int dy) const {
    return false;
}

virtual const char* get_current_idct(void) {
    return NULL;
}

virtual int useDXVA(void) {
    return 0;
};

virtual void setOutputPin(IPin * /*pPin*/) {}

};
```

TvideoCodecDec 这个类中，还定义了一个类 TtelecineManager。这种在类里面再定义一个类的方式还是不太多见的。

## 《FFmpeg 基础库编程开发》

TtelecineManager 这个类的作用还没有研究，先不管它。

可以看出，TvideoCodecDec 类的定义并不复杂，最主要的变量有如下几个，这几个变量都是子类中会用到的：

comprtrQ<IffdshowDecVideo>deciV：重要性不言而喻，回头介绍

IdecVideoSink \*sinkD：重要性不言而喻，回头介绍

void \*quants：QP 表（为什么要存在这里还没搞清）

TvideoCodecDec 类定义了几个函数：

initDec()：初始化解码器（重要）

calcMeanQuant()：计算平均 QP（为什么要在这里计算还没搞清）

TvideoCodecDec 类还定义了一些纯虚函数，作为接口，这些函数的实现都在 TvideoCodecDec 的子类中完成【这几个函数是最重要的】：

beginDecompress();

decompress();

TvideoCodecDec 类中最重要的函数只有一个，就是 initDec()，作用主要是初始化解码器。其他的很多函数大多只是定义了一个名称，并没有实现，因为都是打算在具体各种解码器类中再进行实现的。

看一下 initDec() 的代码：

[cpp] view plaincopy

```
TvideoCodecDec* TvideoCodecDec::initDec(IffdshowBase *deci, IdecVideoSink *sink, AVCodecID codecId, FOURCC fcc, const CMediaType &mt)
```

```
{
```

```
// DXVA mode is a preset setting
```

```
switch (codecId) {
```

```
    case AV_CODEC_ID_H264:
```

```
        if (deci->getParam2(IDFF_filterMode) & IDFF_FILTERMODE_VIDEODXVA) {
```

```
            if (deci->getParam2(IDFF_dec_DXVA_H264)) {
```

```
                codecId = CODEC_ID_H264_DXVA;
```

```
            } else {
```

```
                return NULL;
```

```
}
```

```
}
```

```
        break;
```

```
    case AV_CODEC_ID_VC1:
```

```
    case CODEC_ID_WMV9_LIB:
```

```
        if (deci->getParam2(IDFF_filterMode) & IDFF_FILTERMODE_VIDEODXVA) {
```

```
            if (deci->getParam2(IDFF_dec_DXVA_VC1)) {
```

```
                codecId = CODEC_ID_VC1_DXVA;
```

```
            } else {
```

```
                return NULL;
```

```
}
```

```
}
```

```
        break;
```

```
    default:
```

```
        break;
```

```
}
```

```

TvideoCodecDec *movie = NULL;

if (is_quicksync_codec(codecId)) {
    movie = new TvideoCodecQuickSync(deci, sink, codecId);
} else if (lavc_codec(codecId)) {
    movie = new TvideoCodecLibavcodec(deci, sink);
} else if (raw_codec(codecId)) {
    movie = new TvideoCodecUncompressed(deci, sink);
} else if (wmv9_codec(codecId)) {
    movie = new TvideoCodecWmv9(deci, sink);
} else if (codecId == CODEC_ID_XVID4) {
    movie = new TvideoCodecXviD4(deci, sink);
} else if (codecId == CODEC_ID_LIBMPEG2) {
    movie = new TvideoCodecLibmpeg2(deci, sink);
} else if (codecId == CODEC_ID_AVISYNTH) {
    movie = new TvideoCodecAvisynth(deci, sink);
} else if (codecId == CODEC_ID_H264_DXVA || codecId == CODEC_ID_VC1_DXVA) {
    movie = new TvideoCodecLibavcodecDxva(deci, sink, codecId);
} else {
    return NULL;
}
if (!movie) {
    return NULL;
}
if (movie->ok && movie->testMediaType(fcc, mt)) {
    movie->codecId = codecId;
    return movie;
} else if (is_quicksync_codec(codecId)) {
    // QuickSync decoder init failed, revert to internal decoder.
    switch (codecId) {
        case CODEC_ID_H264_QUICK_SYNC:
            codecId = AV_CODEC_ID_H264;
            break;
        case CODEC_ID_MPEG2_QUICK_SYNC:
            codecId = CODEC_ID_LIBMPEG2;
            break;
        case CODEC_ID_VC1_QUICK_SYNC:
            codecId = CODEC_ID_WMV9_LIB;
            break;
        default:
            ASSERT(FALSE); // this shouldn't happen!
    }
}

```

```

    delete movie;

    // Call this function again with the new codecId.
    return initDec(deci, sink, codecId, fcc, mt);
} else {
    delete movie;
    return NULL;
}
}

```

这个函数的功能还是比较好理解的，根据 CodecID 的不同，创建不同的解码器（从 TvideoCodecLibavcodec, TvideoCodecXviD4, TvideoCodecLibmpeg2 这些里面选择）。

虽然不知道用途是什么，但是我们可以顺便看一下计算平均 QP 的函数，就是把 quants1 指向的 QP 表里面的数据求了一个平均值：

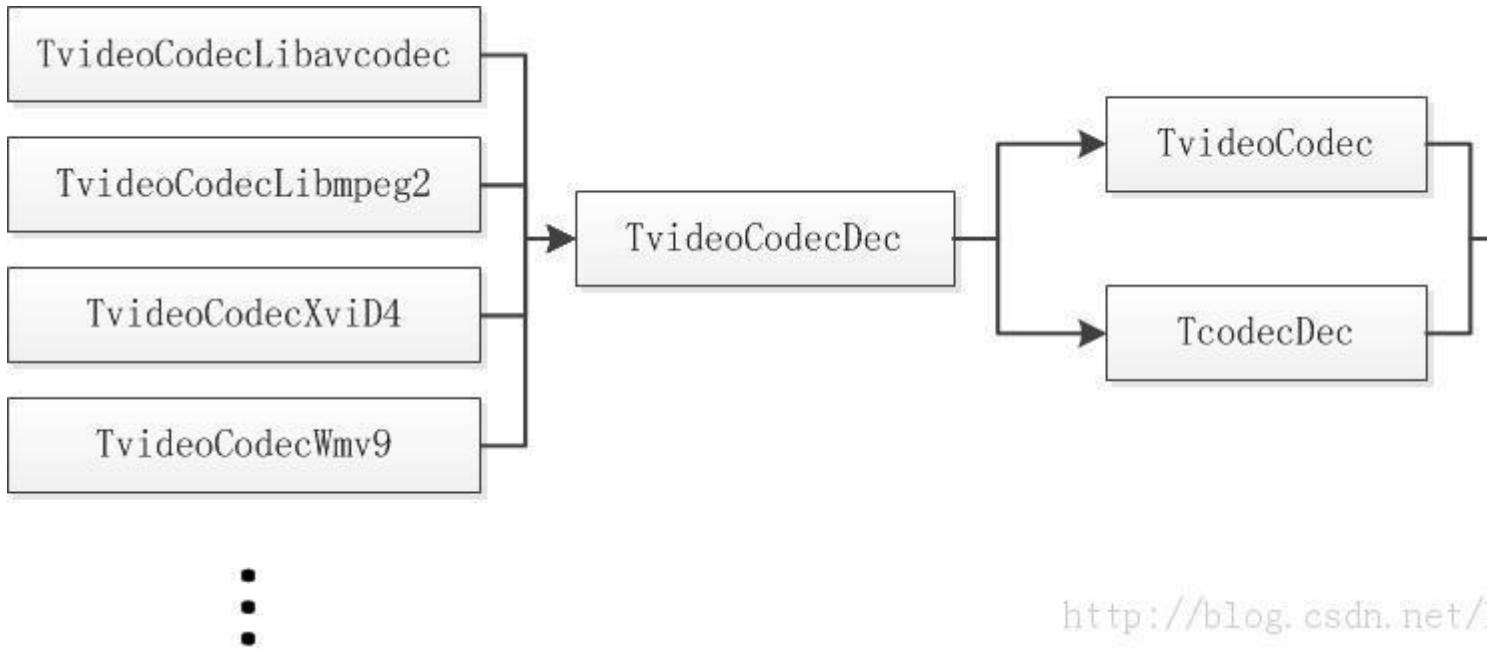
```

[cpp] view plaincopy
//计算平均 QP
float TvideoCodecDec::calcMeanQuant(void)
{
    if (!quants || !quantsDx || !quantsDy) {
        return 0;
    }
    unsigned int sum = 0, num = quantsDx * quantsDy;
    unsigned char *quants1 = (unsigned char*)quants;
    for (unsigned int y = 0; y < quantsDy; y++)
        for (unsigned int x = 0; x < quantsDx; x++) {
            sum += quants1[(y * quantsStride + x) * quantBytes];
        }
    return float(sum) / num;
}

```

## ffdshow 源代码分析 9： 编解码器有关类的总结

本文再做最后一点的分析。在 ffdshow 中有如下继承关系：



<http://blog.csdn.net/le>

前文已经分析过 TvideoCodecLibavcodec, TvideoCodecDec, 在这里我们看一下他们的父类: TvideoCodec, TcodecDec, 以及前两个类的父类 Tcodec。

其实本文介绍的这 3 个类充当了接口的作用, TvideoCodecDec 继承 TvideoCodec, TcodecDec, 以及这两个类继承 Tcodec, 都使用了 virtual 的方式。

先来看看 TvideoCodec。注意这个类强调的是【视频】:

```
[cpp] view plaincopy
//编解码器的父类
class TvideoCodec : virtual public Tcodec
{
public:
    TvideoCodec(IffdshowBase *Idec);
    virtual ~TvideoCodec();
    bool ok;
    int connectedSplitter;
    bool isInterlacedRawVideo;
    Rational containerSar;

    struct CAPS {
        enum {
            NONE = 0,
            VIS_MV = 1,
            VIS_QUANTS = 2
        };
    };

    virtual void end(void) {}
```

};

可以看出 TvideoCodec 定义非常的简单，只包含了视频编解码器会用到的一些变量。注意，是编解码器，不仅仅是解码器。

再来看看 TcodecDec。注意这个类强调的是【解码】：

```
[cpp] view plaincopy
//实现了解码器的祖父类
class TcodecDec : virtual public Tcodec
{
private:
    IdecSink *sink;
protected:
    comptrQ<IffdshowDec> deciD;
    TcodecDec(IffdshowBase *Ideci, IdecSink *Isink);
    virtual ~TcodecDec();
    virtual HRESULT flushDec(void) {
        return S_OK;
    }
public:
    virtual HRESULT flush(void);
};
```

可以看出 TcodecDec 定义非常简单，只包含了解码器需要的一些变量，注意不限于视频解码器，还包含音频解码器。有两个变量比较重要：

```
IdecSink *sink;
comptrQ<IffdshowDec> deciD;
```

最后来看一下 Tcodec。这个类不再继承任何类：

```
[cpp] view plaincopy
//编解码器的祖父类，都是虚函数
class Tcodec
{
protected:
    const Tconfig *config;
    comptr<IffdshowBase> deci;
    Tcodec(IffdshowBase *Ideci);
    virtual ~Tcodec();
public:
    AVCodecID codecId;
    virtual int getType(void) const = 0;
    virtual const char_t* getName(void) const {
        return getMovieSourceName(getType());
    }
    virtual void getEncoderInfo(char_t *buf, size_t buflen) const {
        ff_strncpy(buf, _l("unknown"), buflen);
        buf[buflen - 1] = '\0';
    }
```

```

}

static const char_t* getMovieSourceName(int source);

virtual HRESULT flush() {
    return S_OK;
}

virtual HRESULT BeginFlush() {
    return S_OK;
}

virtual HRESULT EndFlush() {
    return S_OK;
}

virtual bool onSeek(REFERENCE_TIME segmentStart) {
    return false;
}

};

可以看

```

看出，该类定义了一些编解码器会用到的公共函数。有几个变量还是比较重要的：

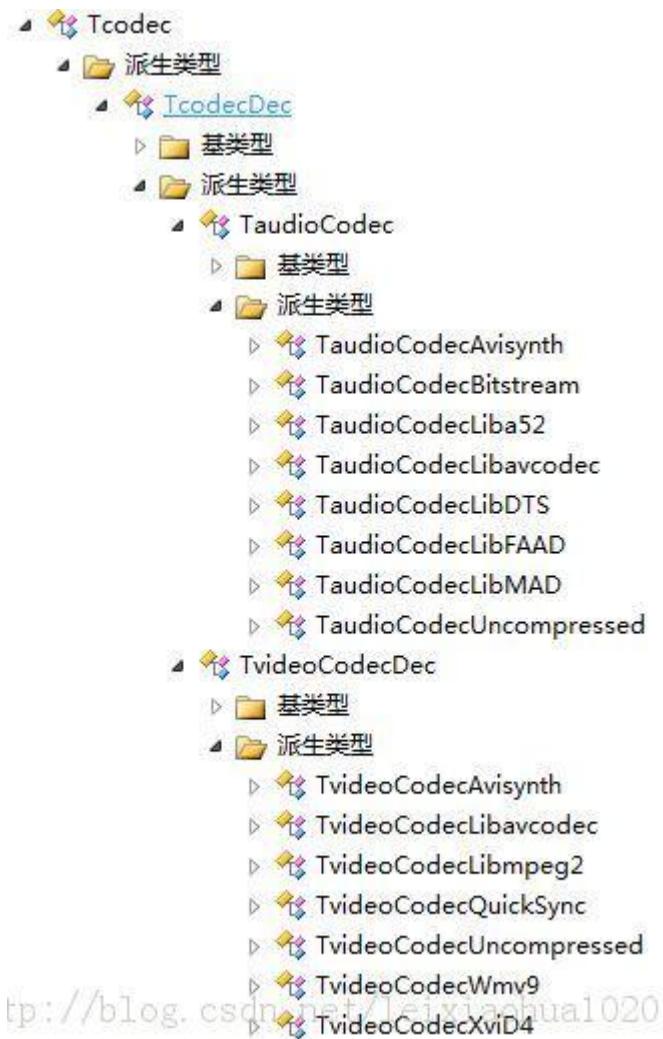
```

const Tconfig *config;
compr<IffdshowBase> deci;
Tcodec(IffdshowBase *Ideci);
AVCodecID codecId

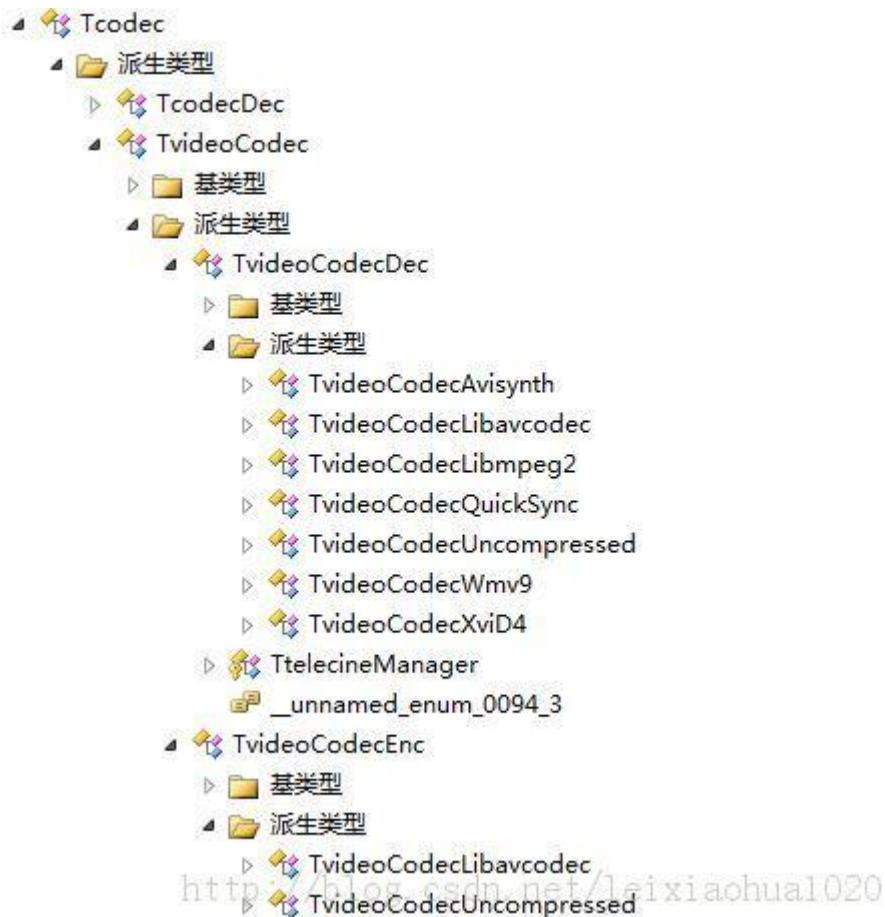
```

自此，我们可以总结出 ffdshow 编解码器这部分继承关系如下（图太大了，截成两张）：

从 TcodecDec 继承下来的如下图所示。包含视频解码器以及音频解码器。



从 TvideoCodec 继承下来的如下图所示。包含了解码器类和编码器类。



总算大体上完成了，关于 ffdshow 解码器封装的内容就先告一段落吧。

## 9.2 LAV filters

LAV Filter 是基于 ffmpeg 的解码器类库 libavcodec，以及解封装器类库 libavformat 的 DirectShow Filter。广泛安装在 PC 上。

### LAV Filter 源代码分析 1： 总体结构

LAV Filter 是一款视频分离和解码软件，他的分离器封装了 FFMPEG 中的 libavformat，解码器则封装了 FFMPEG 中的 libavcodec。它支持十分广泛的视音频格式。

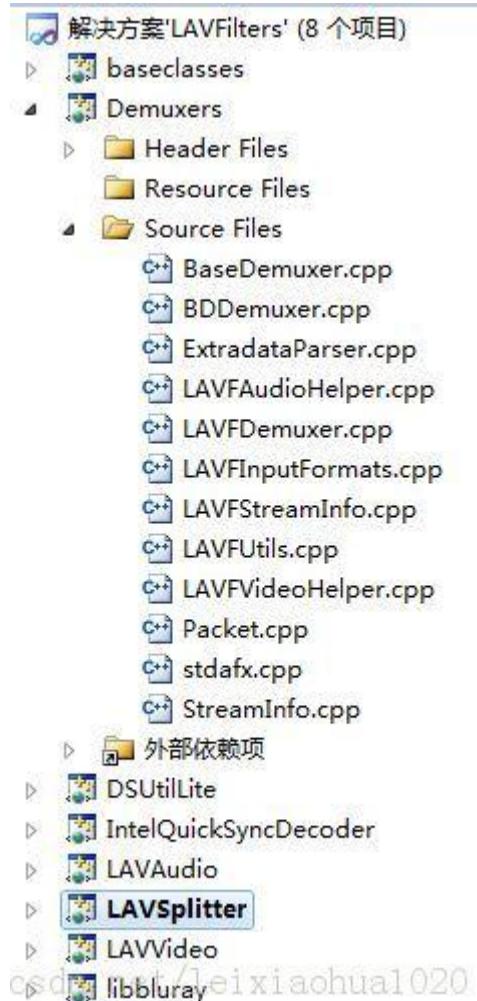
源代码位于 GitHub 或 Google Code：

<https://github.com/Nevcairiel/LAVFilters>

<http://code.google.com/p/lavfilters/>

本文分析了 LAV Filter 源代码的总体架构。

使用 git 获取 LAV filter 源代码之后，使用 VC 2010 打开源代码，发现代码目录结构如图所示：



整个解决方案由 8 个工程组成，介绍一下我目前所知的几个工程：

**baseclasses**: DirectShow 基类，在 DirectShow 的 SDK 中也有，是微软为了简化 DirectShow 开发而提供的。

**Demuxers**: 解封装的基类，**LAVSplitter** 需要调用其中的方法完成解封装操作。

**LAVAudio**: 音频解码 Filter。封装了 libavcodec。

**LAVSplitter**: 解封装 Filter。封装了 libavformat。

**LAVVideo**: 视频解码 Filter。封装了 libavcodec。

**libbluray**: 蓝光的支持。

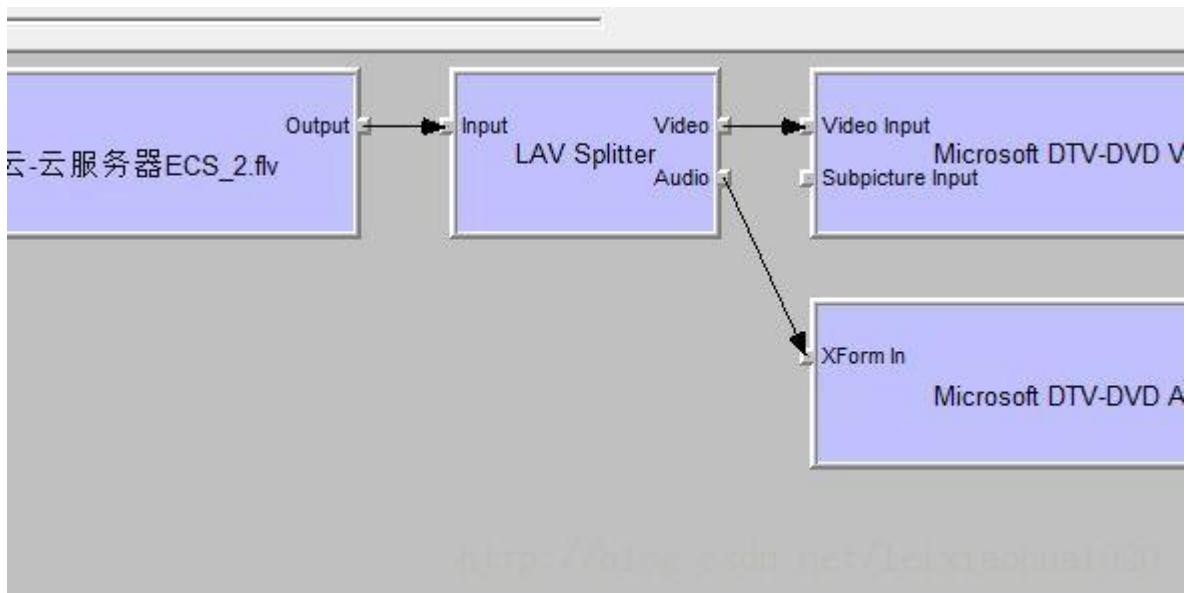
以上标为咖啡色字体的是要重点分析的，也是最重要的工程。

## LAV Filter 源代码分析 2: LAV Splitter

LAV Filter 中最著名的就是 LAV Splitter，支持 Matroska /WebM，MPEG-TS/PS，MP4/MOV，FLV，OGM / OGG，AVI 等其他格式，广泛存在于各种视频播放器（暴风影音这类的）之中。

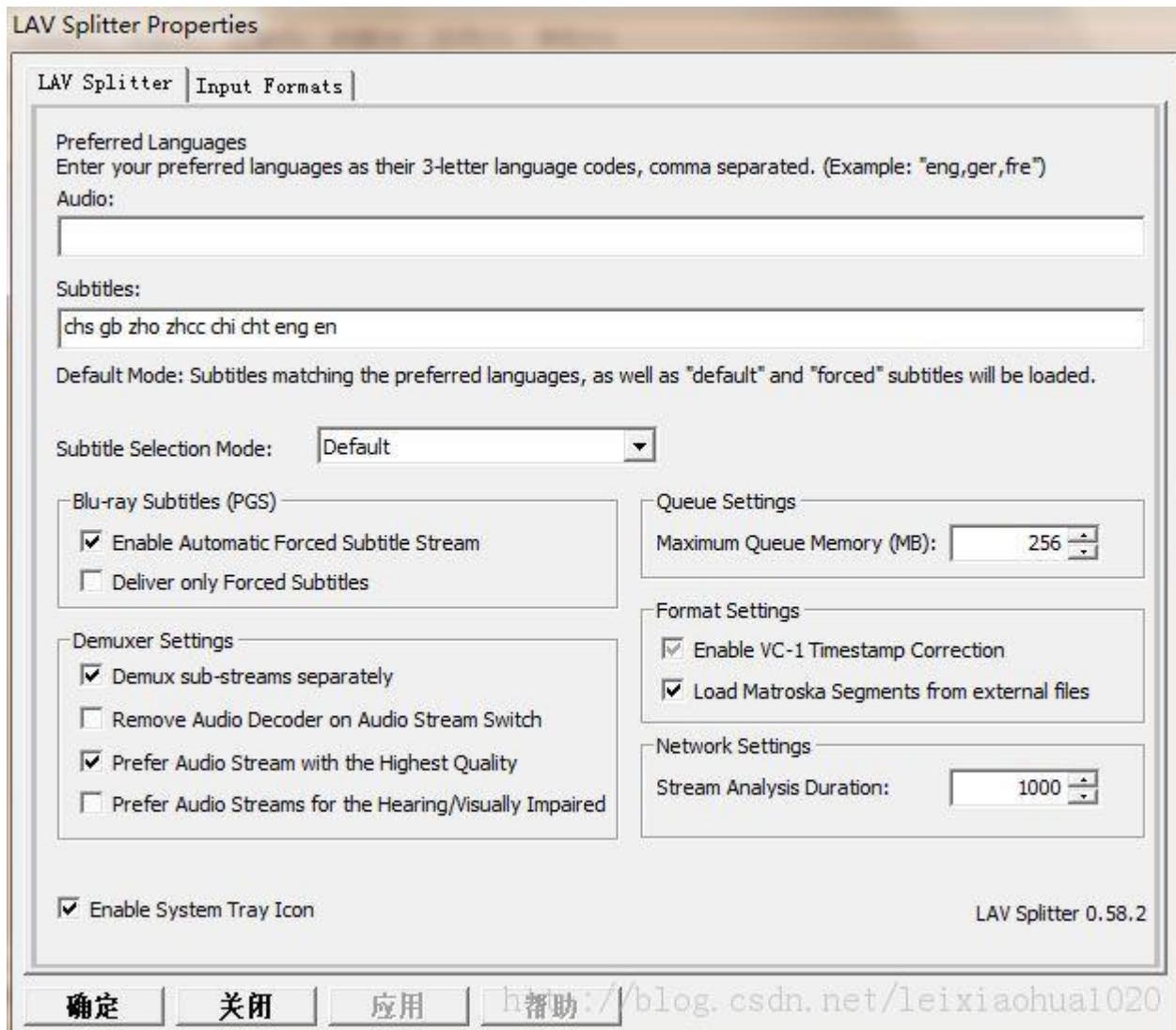
本文分析一下它的源代码。在分析之前，先看看它是什么样的。

使用 GraphEdit 随便打开一个视频文件，就可以看见 LAV Filter:

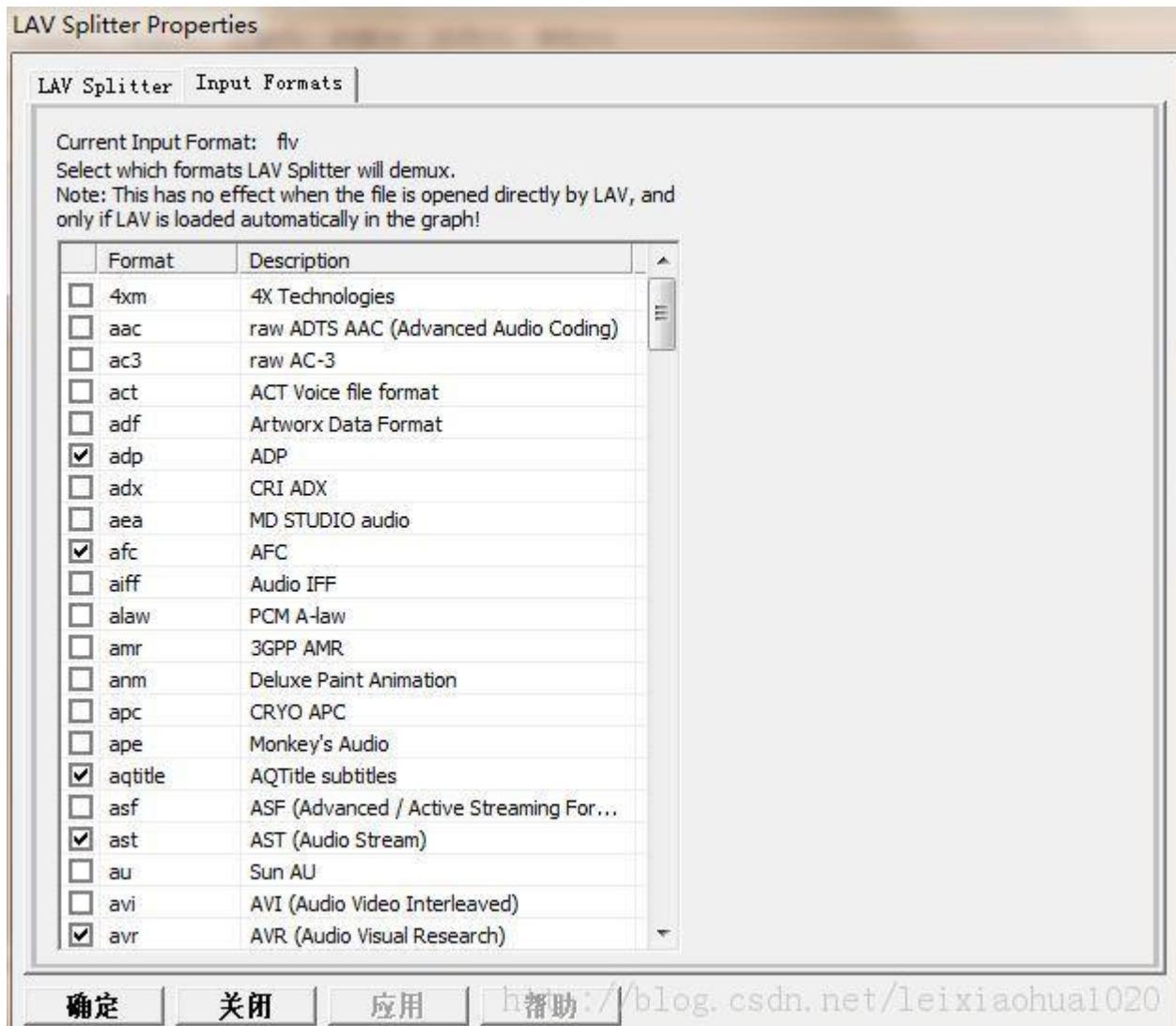


可以右键点击这个 Filter 看一下它的属性页面，如图所示：

属性设置页面：

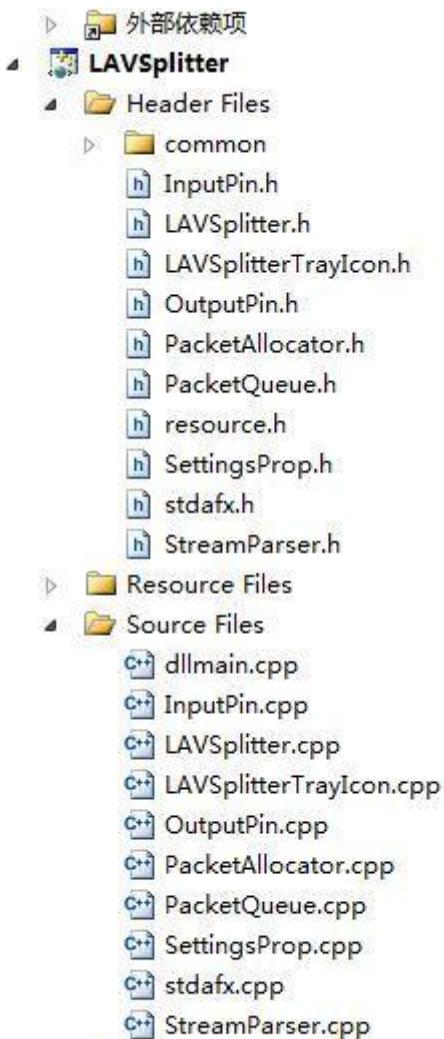


支持输入格式:



下面我们在 VC 2010 中看一下它的源代码：

## 《FFmpeg 基础库编程开发》



blog.sina.com.cn/leixiaohua1020

lavsplitter

从何看起呢？就先从 directshow 的注册函数看起吧，位于 dllmain.cpp 之中。部分代码的含义已经用注释标注上了。从代码可以看出，和普通的 DirectShow Filter 没什么区别。

dllmain.cpp

[cpp] view plaincopy

```
/*
 * Copyright (C) 2010-2013 Hendrik Leppkes
 * http://www.1f0.de
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
```

## 《FFmpeg 基础库编程开发》

```
*   GNU General Public License for more details.  
*  
*   You should have received a copy of the GNU General Public License along  
*   with this program; if not, write to the Free Software Foundation, Inc.,  
*   51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.  
*/
```

```
// Based on the SampleParser Template by GDCL  
// -----  
// Copyright (c) GDCL 2004. All Rights Reserved.  
// You are free to re-use this as the basis for your own filter development,  
// provided you retain this copyright notice in the source.  
// http://www.gdcl.co.uk  
// -----  
//各种定义。。。  
#include "stdafx.h"  
  
// Initialize the GUIDs  
#include <InitGuid.h>  
  
#include <qnetwork.h>  
#include "LAVSplitter.h"  
#include "moreuids.h"  
  
#include "registry.h"  
#include "IGraphRebuildDelegate.h"  
  
// The GUID we use to register the splitter media types  
DEFINE_GUID(MEDIATYPE_LAVSplitter,  
0x9c53931c, 0x7d5a, 0x4a75, 0xb2, 0x6f, 0x4e, 0x51, 0x65, 0x4d, 0xb2, 0xc0);  
  
// --- COM factory table and registration code -----  
//注册时候的信息  
const AMOVIESETUP_MEDIATYPE  
sudMediaTypes[] = {  
    { &MEDIATYPE_Stream, &MEDIASUBTYPE_NULL },  
};  
//注册时候的信息 (PIN)  
const AMOVIESETUP_PIN sudOutputPins[] =  
{  
    {  
        L"Output",           // pin name  
        FALSE,               // is rendered?
```

## 《FFmpeg 基础库编程开发》

```
TRUE,           // is output?
FALSE,          // zero instances allowed?
TRUE,           // many instances allowed?
&CLSID_NULL,    // connects to filter (for bridge pins)
NULL,           // connects to pin (for bridge pins)
0,              // count of registered media types
NULL           // list of registered media types
},
{
L"Input",        // pin name
FALSE,          // is rendered?
FALSE,          // is output?
FALSE,          // zero instances allowed?
FALSE,          // many instances allowed?
&CLSID_NULL,    // connects to filter (for bridge pins)
NULL,           // connects to pin (for bridge pins)
1,              // count of registered media types
&sudMediaTypes[0] // list of registered media types
}
};

//注册时候的信息（名称等）
//CLAVSplitter
const AMOVIESETUP_FILTER sudFilterReg =
{
    &__uuidof(CLAVSplitter),      // filter clsid
    L"LAV Splitter",             // filter name
    MERIT_PREFERRED + 4,         // merit
    2,                           // count of registered pins
    sudOutputPins,               // list of pins to register
    CLSID_LegacyAmFilterCategory
};

//注册时候的信息（名称等）
//CLAVSplitterSource
const AMOVIESETUP_FILTER sudFilterRegSource =
{
    &__uuidof(CLAVSplitterSource), // filter clsid
    L"LAV Splitter Source",       // filter name
    MERIT_PREFERRED + 4,         // merit
    1,                           // count of registered pins
    sudOutputPins,               // list of pins to register
    CLSID_LegacyAmFilterCategory
};
```

// --- COM factory table and registration code -----

// DirectShow base class COM factory requires this table,

// declaring all the COM objects in this DLL

// 注意 g\_Templates 名称是固定的

CFactoryTemplate g\_Templates[] = {

  // one entry for each CoCreate-able object

  {

    sudFilterReg.strName,

    sudFilterReg.clsID,

    CreateInstance<CLAVSplitter>,

    CLAVSplitter::StaticInit,

    &sudFilterReg

  },

  {

    sudFilterRegSource.strName,

    sudFilterRegSource.clsID,

    CreateInstance<CLAVSplitterSource>,

    NULL,

    &sudFilterRegSource

  },

// This entry is for the property page.

// 属性页

  {

    L"lav Splitter Properties",

    &CLSID\_LAVSplitterSettingsProp,

    CreateInstance<CLAVSplitterSettingsProp>,

    NULL, NULL

  },

  {

    L"lav Splitter Input Formats",

    &CLSID\_LAVSplitterFormatsProp,

    CreateInstance<CLAVSplitterFormatsProp>,

    NULL, NULL

  }

};

int g\_cTemplates = sizeof(g\_Templates) / sizeof(g\_Templates[0]);

// self-registration entrypoint

STDAPI DllRegisterServer()

{

  std::list<LPCWSTR> chkbytes;

```

// BluRay
chkbytes.clear();
chkbytes.push_back(L"0,4,494E4458"); // INDX (index.bdmv)
chkbytes.push_back(L"0,4,4D4F424A"); // MOBJ (MovieObject.bdmv)
chkbytes.push_back(L"0,4,4D504C53"); // MPLS
RegisterSourceFilter(__uuidof(CLAVSplitterSource),
    MEDIASUBTYPE_LAVBluRay, chkbytes, NULL);

// base classes will handle registration using the factory template table
return AMovieDllRegisterServer2(true);
}

STDAPI DllUnregisterServer()
{
    UnRegisterSourceFilter(MEDIASUBTYPE_LAVBluRay);

    // base classes will handle de-registration using the factory template table
    return AMovieDllRegisterServer2(false);
}

// if we declare the correct C runtime entrypoint and then forward it to the DShow base
// classes we will be sure that both the C/C++ runtimes and the base classes are initialized
// correctly
extern "C" BOOL WINAPI DllEntryPoint(HINSTANCE, ULONG, LPVOID);
BOOL WINAPI DllMain(HANDLE hDllHandle, DWORD dwReason, LPVOID lpReserved)
{
    return DllEntryPoint(reinterpret_cast<HINSTANCE>(hDllHandle), dwReason, lpReserved);
}

void CALLBACK OpenConfiguration(HWND hwnd, HINSTANCE hinst, LPSTR lpszCmdLine, int nCmdShow)
{
    HRESULT hr = S_OK;
    CUnknown *pInstance = CreateInstance<CLAVSplitter>(NULL, &hr);
    IBaseFilter *pFilter = NULL;
    pInstance->NonDelegatingQueryInterface(IID_IBaseFilter, (void **)&pFilter);
    if (pFilter) {
        pFilter->AddRef();
        CBaseDSPropPage::ShowPropPageDialog(pFilter);
    }
    delete pInstance;
}

```

接下来就要进入正题了，看一看核心的分离器（解封装器）的类 CLAVSplitter 的定义文件 LAVSplitter.h。乍一看这个类确实了得，居然继承了那么多的父类，实在是碉堡了。先不管那么多，看看里面都有什么函数吧。主要的函数

## 《FFmpeg 基础库编程开发》

上面都加了注释。注意还有一个类 CLAVSplitterSource 继承了 CLAVSplitter。

LAVSplitter.h

[cpp] view plaincopy

```
/*
 * Copyright (C) 2010-2013 Hendrik Leppkes
 * http://www.1f0.de
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along
 * with this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Initial design and concept by Gabest and the MPC-HC Team, copyright under GPLv2
 * Contributions by Ti-BEN from the XBMC DSPlayer Project, also under GPLv2
 */
```

#pragma once

```
#include <string>
#include <list>
#include <set>
#include <vector>
#include <map>
#include "PacketQueue.h"

#include "BaseDemuxer.h"

#include "LAVSplitterSettingsInternal.h"
#include "SettingsProp.h"
#include "IBufferInfo.h"

#include "ISpecifyPropertyPages2.h"

#include "LAVSplitterTrayIcon.h"
```

```
#define LAVF_REGISTRY_KEY L"Software\LAV\Splitter"
#define LAVF_REGISTRY_KEY_FORMATS LAVF_REGISTRY_KEY L"\Formats"
#define LAVF_LOG_FILE      L"LAVSplitter.txt"

#define MAX PTS SHIFT 50000000i64

class CLAVOutputPin;
class CLAVInputPin;

#ifndef _MSC_VER
#pragma warning(disable: 4355)
#endif

//核心解复用（分离器）
//暴露的接口在 ILAVFSettings 中
[uuid("171252A0-8820-4AFE-9DF8-5C92B2D66B04")]
class CLAVSplitter
    : public CBaseFilter
    , public CCritSec
    , protected CAMThread
    , public IFileSourceFilter
    , public IMediaSeeking
    , public IAMStreamSelect
    , public IAMOpenProgress
    , public ILAVFSettingsInternal
    , public ISpecifyPropertyPages2
    , public IObjectWithSite
    , public IBufferInfo
{
public:
    CLAVSplitter(LPUNKNOWN pUnk, HRESULT* phr);
    virtual ~CLAVSplitter();

    static void CALLBACK StaticInit(BOOL bLoading, const CLSID *clsid);

    // IUnknown
    //
    DECLARE_IUNKNOWN;
    //暴露接口，使外部程序可以 QueryInterface，关键！
    //翻译（“没有代表的方式查询接口”）
    STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void** ppv);

    // CBaseFilter methods
```

## 《FFmpeg 基础库编程开发》

```
//输入是一个，输出就不一定了！
int GetPinCount();
CBasePin *GetPin(int n);
STDMETHODIMP GetClassID(CLSID* pClsID);

STDMETHODIMP Stop();
STDMETHODIMP Pause();
STDMETHODIMP Run(REFERENCE_TIME tStart);

STDMETHODIMP JoinFilterGraph(IFilterGraph * pGraph, LPCWSTR pName);

// IFileSourceFilter
// 源 Filter 的接口方法
STDMETHODIMP Load(LPCOLESTR pszFileName, const AM_MEDIA_TYPE * pmt);
STDMETHODIMP GetCurFile(LPOLESTR *ppszFileName, AM_MEDIA_TYPE *pmt);

// IMediaSeeking
STDMETHODIMP GetCapabilities(DWORD* pCapabilities);
STDMETHODIMP CheckCapabilities(DWORD* pCapabilities);
STDMETHODIMP IsFormatSupported(const GUID* pFormat);
STDMETHODIMP QueryPreferredFormat(GUID* pFormat);
STDMETHODIMP GetTimeFormat(GUID* pFormat);
STDMETHODIMP IsUsingTimeFormat(const GUID* pFormat);
STDMETHODIMP SetTimeFormat(const GUID* pFormat);
STDMETHODIMP GetDuration(LONGLONG* pDuration);
STDMETHODIMP GetStopPosition(LONGLONG* pStop);
STDMETHODIMP GetCurrentPosition(LONGLONG* pCurrent);
STDMETHODIMP ConvertTimeFormat(LONGLONG* pTarget, const GUID* pTargetFormat, LONGLONG Source,
const GUID* pSourceFormat);
STDMETHODIMP SetPositions(LONGLONG* pCurrent, DWORD dwCurrentFlags, LONGLONG* pStop, DWORD dwStopFlags);
STDMETHODIMP GetPositions(LONGLONG* pCurrent, LONGLONG* pStop);
STDMETHODIMP GetAvailable(LONGLONG* pEarliest, LONGLONG* pLatest);
STDMETHODIMP SetRate(double dRate);
STDMETHODIMP GetRate(double* pdRate);
STDMETHODIMP GetPreroll(LONGLONG* pllPreroll);

// IAMStreamSelect
STDMETHODIMP Count(DWORD *pcStreams);
STDMETHODIMP Enable(long lIndex, DWORD dwFlags);
STDMETHODIMP Info(long lIndex, AM_MEDIA_TYPE **ppmt, DWORD *pdwFlags, LCID *plcid, DWORD
*pdwGroup, WCHAR **ppszName, IUnknown **ppObject, IUnknown **ppUnk);
```

```
// IAMOpenProgress
STDMETHODIMP QueryProgress(ONGLONG *pllTotal, LONGLONG *pllCurrent);
STDMETHODIMP AbortOperation();

// ISpecifyPropertyPages2
STDMETHODIMP GetPages(CAUUID *pPages);
STDMETHODIMP CreatePage(const GUID& guid, IPropertyPage** ppPage);

// IObjectWithSite
STDMETHODIMP SetSite(IUnknown *pUnkSite);
STDMETHODIMP GetSite(REFIID riid, void **ppvSite);

// IBufferInfo
STDMETHODIMP_(int) GetCount();
STDMETHODIMP GetStatus(int i, int& samples, int& size);
STDMETHODIMP_(DWORD) GetPriority();

// ILAVFSettings
STDMETHODIMP SetRuntimeConfig(BOOL bRuntimeConfig);
STDMETHODIMP GetPreferredLanguages(LPWSTR *ppLanguages);
STDMETHODIMP SetPreferredLanguages(LPCWSTR pLanguages);
STDMETHODIMP GetPreferredSubtitleLanguages(LPWSTR *ppLanguages);
STDMETHODIMP SetPreferredSubtitleLanguages(LPCWSTR pLanguages);
STDMETHODIMP_(LAVSubtitleMode) GetSubtitleMode();
STDMETHODIMP SetSubtitleMode(LAVSubtitleMode mode);
STDMETHODIMP_(BOOL) GetSubtitleMatchingLanguage();
STDMETHODIMP SetSubtitleMatchingLanguage(BOOL dwMode);
STDMETHODIMP_(BOOL) GetPGSForcedStream();
STDMETHODIMP SetPGSForcedStream(BOOL bFlag);
STDMETHODIMP_(BOOL) GetPGSOnlyForced();
STDMETHODIMP SetPGSOnlyForced(BOOL bForced);
STDMETHODIMP_(int) GetVC1TimestampMode();
STDMETHODIMP SetVC1TimestampMode(int iMode);
STDMETHODIMP SetSubstreamsEnabled(BOOL bSubStreams);
STDMETHODIMP_(BOOL) GetSubstreamsEnabled();
STDMETHODIMP SetVideoParsingEnabled(BOOL bEnabled);
STDMETHODIMP_(BOOL) GetVideoParsingEnabled();
STDMETHODIMP SetFixBrokenHDPVR(BOOL bEnabled);
STDMETHODIMP_(BOOL) GetFixBrokenHDPVR();
STDMETHODIMP_(HRESULT) SetFormatEnabled(LPCSTR strFormat, BOOL bEnabled);
STDMETHODIMP_(BOOL) IsFormatEnabled(LPCSTR strFormat);
STDMETHODIMP SetStreamSwitchRemoveAudio(BOOL bEnabled);
STDMETHODIMP_(BOOL) GetStreamSwitchRemoveAudio();
```

## 《FFmpeg 基础库编程开发》

```
STDMETHODIMP GetAdvancedSubtitleConfig(LPWSTR *ppAdvancedConfig);
STDMETHODIMP SetAdvancedSubtitleConfig(LPCWSTR pAdvancedConfig);
STDMETHODIMP SetUseAudioForHearingVisuallyImpaired(BOOL bEnabled);
STDMETHODIMP_(BOOL) GetUseAudioForHearingVisuallyImpaired();
STDMETHODIMP SetMaxQueueMemSize(DWORD dwMaxSize);
STDMETHODIMP_(DWORD) GetMaxQueueMemSize();
STDMETHODIMP SetTrayIcon(BOOL bEnabled);
STDMETHODIMP_(BOOL) GetTrayIcon();
STDMETHODIMP SetPreferHighQualityAudioStreams(BOOL bEnabled);
STDMETHODIMP_(BOOL) GetPreferHighQualityAudioStreams();
STDMETHODIMP SetLoadMatroskaExternalSegments(BOOL bEnabled);
STDMETHODIMP_(BOOL) GetLoadMatroskaExternalSegments();
STDMETHODIMP GetFormats(LPSTR** formats, UINT* nFormats);
STDMETHODIMP SetNetworkStreamAnalysisDuration(DWORD dwDuration);
STDMETHODIMP_(DWORD) GetNetworkStreamAnalysisDuration();

// ILAVSplitterSettingsInternal
STDMETHODIMP_(LPCSTR) GetInputFormat() { if (m_pDemuxer) return m_pDemuxer->GetContainerFormat();
return NULL; }
STDMETHODIMP_(std::set<FormatInfo>&) GetInputFormats();
STDMETHODIMP_(BOOL) IsVC1CorrectionRequired();
STDMETHODIMP_(CMediaType *) GetOutputMediatype(int stream);
STDMETHODIMP_(IFilterGraph *) GetFilterGraph() { if (m_pGraph) { m_pGraph->AddRef(); return m_pGraph; }
return NULL; }

STDMETHODIMP_(DWORD) GetStreamFlags(DWORD dwStream) { if (m_pDemuxer) return m_pDemuxer->GetStreamFlags(dwStream); return 0; }
STDMETHODIMP_(int) GetPixelFormat(DWORD dwStream) { if (m_pDemuxer) return m_pDemuxer->GetPixelFormat(dwStream); return AV_PIX_FMT_NONE; }
STDMETHODIMP_(int) GetHasBFrames(DWORD dwStream){ if (m_pDemuxer) return m_pDemuxer->GetHasBFrames(dwStream); return -1; }

// Settings helper
std::list<std::string> GetPreferredAudioLanguageList();
std::list<CSubtitleSelector> GetSubtitleSelectors();

bool IsAnyPinDrying();
void SetFakeASFReader(BOOL bFlag) { m_bFakeASFReader = bFlag; }

protected:
// CAMThread
enum {CMD_EXIT, CMD_SEEK};
DWORD ThreadProc();
```

## 《FFmpeg 基础库编程开发》

```
HRESULT DemuxSeek(REFERENCE_TIME rtStart);
HRESULT DemuxNextPacket();
HRESULT DeliverPacket(Packet *pPacket);

void DeliverBeginFlush();
void DeliverEndFlush();

STDMETHODIMP Close();
STDMETHODIMP DeleteOutputs();
//初始化解复用器
STDMETHODIMP InitDemuxer();

friend class CLAVOutputPin;
STDMETHODIMP SetPositionsInternal(void *caller, LONGLONG* pCurrent, DWORD dwCurrentFlags,
LONGLONG* pStop, DWORD dwStopFlags);

public:
    CLAVOutputPin *GetOutputPin(DWORD streamId, BOOL bActiveOnly = FALSE);
    STDMETHODIMP RenameOutputPin(DWORD TrackNumSrc, DWORD TrackNumDst, std::vector<CMediaType>
pmts);
    STDMETHODIMP UpdateForcedSubtitleMediaType();

STDMETHODIMP CompleteInputConnection();
STDMETHODIMP BreakInputConnection();

protected:
    //相关的参数设置
    STDMETHODIMP LoadDefaults();
    STDMETHODIMP ReadSettings(HKEY rootKey);
    STDMETHODIMP LoadSettings();
    STDMETHODIMP SaveSettings();
    //创建图标
    STDMETHODIMP CreateTrayIcon();

protected:
    CLAVInputPin *m_pInput;

private:
    CCritSec m_csPins;
    //用 vector 存储输出 PIN (解复用的时候是不确定的)
    std::vector<CLAVOutputPin *> m_pPins;
    //活动的
    std::vector<CLAVOutputPin *> m_pActivePins;
```

```

//不用的
std::vector<CLAVOutputPin *> m_pRetiredPins;
std::set<DWORD> m_bDiscontinuitySent;

std::wstring m_fileName;
std::wstring m_processName;
//有很多纯虚函数的基本解复用类
//注意：绝大部分信息都是从这获得的
//这里的信息是由其派生类从 FFMPEG 中获取到的
CBaseDemuxer *m_pDemuxer;

BOOL m_bPlaybackStarted;
BOOL m_bFakeASFReader;

// Times
REFERENCE_TIME m_rtStart, m_rtStop, m_rtCurrent, m_rtNewStart, m_rtNewStop;
REFERENCE_TIME m_rtOffset;
double m_dRate;
BOOL m_bStopValid;

// Seeking
REFERENCE_TIME m_rtLastStart, m_rtLastStop;
std::set<void *> m_LastSeekers;

// flushing
bool m_fFlushing;
CAMEvent m_eEndFlush;

std::set<FormatInfo> m_InputFormats;

// Settings
//设置
struct Settings {
    BOOL TrayIcon;
    std::wstring prefAudioLangs;
    std::wstring prefSubLangs;
    std::wstring subtitleAdvanced;
    LAVSubtitleMode subtitleMode;
    BOOL PGSForcedStream;
    BOOL PGSOOnlyForced;
    int vc1Mode;
    BOOL substreams;
}

```

## 《FFmpeg 基础库编程开发》

```
BOOL MatroskaExternalSegments;

BOOL StreamSwitchRemoveAudio;
BOOL ImpairedAudio;
BOOL PreferHighQualityAudio;
DWORD QueueMaxSize;
DWORD NetworkAnalysisDuration;

std::map<std::string, BOOL> formats;
} m_settings;

BOOL m_bRuntimeConfig;

IUnknown *m_pSite;

CBaseTrayIcon *m_pTrayIcon;
};

[uuid("B98D13E7-55DB-4385-A33D-09FD1BA26338")]
class CLAVSplitterSource : public CLAVSplitter
{
public:
    // construct only via class factory
    CLAVSplitterSource(LPUNKNOWN pUnk, HRESULT* phr);
    virtual ~CLAVSplitterSource();

    // IUnknown
    DECLARE_IUNKNOWN;
    //暴露接口，使外部程序可以 QueryInterface，关键！
    //翻译（“没有代表的方式查询接口”）
    STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void*** ppv);
};

先来看一下查询接口的函数 NonDelegatingQueryInterface()吧
[cpp] view plaincopy
//暴露接口，使外部程序可以 QueryInterface，关键！
STDMETHODIMP CLAVSplitter::NonDelegatingQueryInterface(REFIID riid, void*** ppv)
{
    CheckPointer(ppv, E_POINTER);

    *ppv = NULL;

    if (m_pDemuxer && (riid == __uuidof(IKeyFrameInfo) || riid == __uuidof(ITrackInfo) || riid ==
IID_IAMExtendedSeeking || riid == IID_IAMMediaContent)) {
```

## 《FFmpeg 基础库编程开发》

```
return m_pDemuxer->QueryInterface(rIID, ppv);
}
//写法好特别啊，意思是一样的
return
    QI(IMediaSeeking)
    QI(IAMStreamSelect)
    QI(ISpecifyPropertyPages)
    QI(ISpecifyPropertyPages2)
    QI2(ILAVFSettings)
    QI2(ILAVFSettingsInternal)
    QI(IObjectWithSite)
    QI(IBufferInfo)
    __super::NonDelegatingQueryInterface(rIID, ppv);
}
```

这个 NonDelegatingQueryInterface()的写法确实够特别的，不过其作用还是一样的：根据不同的 REFIID，获得不同的接口指针。在这里就不多说了。

再看一下 Load()函数

```
[cpp] view plaincopy
// IFileSourceFilter
// 打开
STDMETHODIMP CLAVSplitter::Load(LPCOLESTR pszFileName, const AM_MEDIA_TYPE * pmt)
{
    CheckPointer(pszFileName, E_POINTER);

    m_bPlaybackStarted = FALSE;

    m_fileName = std::wstring(pszFileName);

    HRESULT hr = S_OK;
    SAFE_DELETE(m_pDemuxer);
    LPWSTR extension = PathFindExtensionW(pszFileName);

    DbgLog((LOG_TRACE, 10, L"::Load(): Opening file '%s' (extension: %s)", pszFileName, extension));

    // BDMV uses the BD demuxer, everything else LAVF
    if (_wcsicmp(extension, L".bdmv") == 0 || _wcsicmp(extension, L".mpls") == 0) {
        m_pDemuxer = new CBDDemuxer(this, this);
    } else {
        m_pDemuxer = new CLAVFDemuxer(this, this);
    }
    //打开
    if(FAILED(hr = m_pDemuxer->Open(pszFileName))) {
        SAFE_DELETE(m_pDemuxer);
```

```

    return hr;
}
m_pDemuxer->AddRef();

return InitDemuxer();
}

```

在这里我们要注意 CLAVSplitter 的一个变量： m\_pDemuxer。这是一个指向 CBaseDemuxer 的指针。因此在这里 CLAVSplitter 实际上调用了 CBaseDemuxer 中的方法。

从代码中的逻辑我们可以看出：

1. 寻找文件后缀
2. 当文件后缀是：" .bdmv " 或者 ".mpls" 的时候， m\_pDemuxer 指向一个 CBDDemuxer (我推测这代表目标文件是蓝光文件什么的)，其他情况下 m\_pDemuxer 指向一个 CLAVFDemuxer。
3. 然后 m\_pDemuxer 会调用 Open() 方法。
4. 最后会调用一个 InitDemuxer() 方法。

在这里我们应该看看 m\_pDemuxer->Open() 这个方法里面有什么。我们先考虑 m\_pDemuxer 指向 CLAVFDemuxer 的情况。

```
[cpp] view plaincopy
// Demuxer Functions
// 打开 (就是一个封装)
STDMETHODIMP CLAVFDemuxer::Open(LPCOLESTR pszFileName)
{
    return OpenInputStream(NULL, pszFileName, NULL, TRUE);
}
```

发现是一层封装，于是果断决定层层深入。

```
[cpp] view plaincopy
// 实际的打开, 使用 FFMPEG
STDMETHODIMP CLAVFDemuxer::OpenInputStream(AVIOContext *byteContext, LPCOLESTR pszFileName, const
char *format, BOOL bForce)
{
    CAutoLock lock(m_pLock);
    HRESULT hr = S_OK;
```

```
int ret; // return code from avformat functions
```

```
// Convert the filename from wchar to char for avformat
char fileName[4100] = {0};
if (pszFileName) {
    ret = WideCharToMultiByte(CP_UTF8, 0, pszFileName, -1, fileName, 4096, NULL, NULL);
}
```

```
if (_strnicmp("mms:", fileName, 4) == 0) {
    memmove(fileName+1, fileName, strlen(fileName));
    memcpy(fileName, "mmsh", 4);
```

```
}
```

```
AVIOInterruptCB cb = {avio_interrupt_cb, this};
```

trynoformat:

```
// Create the avformat_context
```

```
// FFMPEG 中的函数
```

```
m_avFormat = avformat_alloc_context();
```

```
m_avFormat->pb = byteContext;
```

```
m_avFormat->interrupt_callback = cb;
```

```
if (m_avFormat->pb)
```

```
    m_avFormat->flags |= AVFMT_FLAG_CUSTOM_IO;
```

```
LPWSTR extension = pszFileName ? PathFindExtensionW(pszFileName) : NULL;
```

```
AVInputFormat *inputFormat = NULL;
```

```
//如果指定了格式
```

```
if (format) {
```

```
    //查查有木有
```

```
    inputFormat = av_find_input_format(format);
```

```
} else if (pszFileName) {
```

```
    LPWSTR extension = PathFindExtensionW(pszFileName);
```

```
    for (int i = 0; i < countof(wszImageExtensions); i++) {
```

```
        if (_wcsicmp(extension, wszImageExtensions[i]) == 0) {
```

```
            if (byteContext) {
```

```
                inputFormat = av_find_input_format("image2pipe");
```

```
            } else {
```

```
                inputFormat = av_find_input_format("image2");
```

```
            }
```

```
            break;
```

```
}
```

```
}
```

```
for (int i = 0; i < countof(wszBlockedExtensions); i++) {
```

```
    if (_wcsicmp(extension, wszBlockedExtensions[i]) == 0) {
```

```
        goto done;
```

```
}
```

```
}
```

```
}
```

```
// Disable loading of external mkv segments, if required
```

```
if (!m_pSettings->GetLoadMatroskaExternalSegments())
```

```
    m_avFormat->flags |= AVFMT_FLAG_NOEXTERNAL;
```

```

m_timeOpening = time(NULL);
//实际的打开
ret = avformat_open_input(&m_avFormat, fileName, inputFormat, NULL);
//出错了
if (ret < 0) {
    DbgLog((LOG_ERROR, 0, TEXT("::OpenInputStream(): avformat_open_input failed (%d)", ret));
    if (format) {
        DbgLog((LOG_ERROR, 0, TEXT(" -> trying again without specific format")));
        format = NULL;
    //实际的关闭
    avformat_close_input(&m_avFormat);
    goto trynoformat;
}
    goto done;
}
DbgLog((LOG_TRACE, 10, TEXT("::OpenInputStream(): avformat_open_input opened file of type '%S' (took %I64d
seconds)", m_avFormat->iformat->name, time(NULL) - m_timeOpening));
m_timeOpening = 0;
//初始化 AVFormat
CHECK_HR(hr = InitAVFormat(pszFileName, bForce));

return S_OK;
done:
CleanupAVFormat();
return E_FAIL;
}

```

看到这个函数，立马感受到了一种“拨云见日”的感觉。看到了很多 FFMPEG 的 API 函数。最重要的依据当属 avformat\_open\_input()了，通过这个函数，打开了实际的文件。如果出现错误，则调用 avformat\_close\_input()进行清理。

最后，还调用了 InitAVFormat()函数：

```

[cpp] view plaincopy
//初始化 AVFormat
STDMETHODIMP CLAVFDemuxer::InitAVFormat(LPCOLESTR pszFileName, BOOL bForce)
{
    HRESULT hr = S_OK;
    const char *format = NULL;
    //获取 InputFormat 信息（，短名称，长名称）
    lavf_get_iformat_infos(m_avFormat->iformat, &format, NULL);
    if (!bForce && (!format || !m_pSettings->IsFormatEnabled(format))) {
        DbgLog((LOG_TRACE, 20, L"::InitAVFormat() - format of type '%S' disabled, failing", format ? format :
m_avFormat->iformat->name));
        return E_FAIL;
    }
}

```

}

```
m_pszInputFormat = format ? format : m_avFormat->iFORMAT->name;
```

```
m_bVC1SeenTimestamp = FALSE;
```

```
LPWSTR extension = pszFileName ? PathFindExtensionW(pszFileName) : NULL;
```

```
m_bMatroska = (_strnicmp(m_pszInputFormat, "matroska", 8) == 0);
```

```
m_bOgg = (_strnicmp(m_pszInputFormat, "ogg", 3) == 0);
```

```
m_bAVI = (_strnicmp(m_pszInputFormat, "avi", 3) == 0);
```

```
m_bMPEGTS = (_strnicmp(m_pszInputFormat, "mpegs", 6) == 0);
```

```
m_bMPEGPS = (_stricmp(m_pszInputFormat, "mpeg") == 0);
```

```
m_bRM = (_stricmp(m_pszInputFormat, "rm") == 0);
```

```
m_bPMP = (_stricmp(m_pszInputFormat, "pmp") == 0);
```

```
m_bMP4 = (_stricmp(m_pszInputFormat, "mp4") == 0);
```

```
m_bTSDiscont = m_avFormat->iFORMAT->flags & AVFMT_TS_DISCONT;
```

```
WCHAR szProt[24] = L"file";
```

```
if (pszFileName) {
```

```
    DWORD dwNumChars = 24;
```

```
    hr = UrlGetPart(pszFileName, szProt, &dwNumChars, URL_PART_SCHEME, 0);
```

```
    if (SUCCEEDED(hr) && dwNumChars && (_wcsicmp(szProt, L"file") != 0)) {
```

```
        m_avFormat->flags |= AVFMT_FLAG_NETWORK;
```

```
        DbgLog((LOG_TRACE, 10, TEXT("::InitAVFormat(): detected network protocol: %s"), szProt));
```

```
}
```

```
}
```

```
// TODO: make both durations below configurable
```

```
// decrease analyze duration for network streams
```

```
if (m_avFormat->flags & AVFMT_FLAG_NETWORK || (m_avFormat->flags & AVFMT_FLAG_CUSTOM_IO
&& !m_avFormat->pb->seekable)) {
```

```
    // require at least 0.2 seconds
```

```
    m_avFormat->max_analyze_duration = max(m_pSettings->GetNetworkStreamAnalysisDuration() * 1000, 200000);
```

```
} else {
```

```
    // And increase it for mpeg-ts/ps files
```

```
    if (m_bMPEGTS || m_bMPEGPS)
```

```
        m_avFormat->max_analyze_duration = 10000000;
```

```
}
```

```
av_opt_set_int(m_avFormat, "correct_ts_overflow", !m_pBluRay, 0);
```

```

if (m_bMatroska)
    m_avFormat->flags |= AVFMT_FLAG_KEEP_SIDE_DATA;

m_timeOpening = time(NULL);
//获取媒体流信息
int ret = avformat_find_stream_info(m_avFormat, NULL);
if (ret < 0) {
    DbgLog((LOG_ERROR, 0, TEXT("::InitAVFormat(): av_find_stream_info failed (%d)", ret));
    goto done;
}
DbgLog((LOG_TRACE, 10, TEXT("::InitAVFormat(): avformat_find_stream_info finished, took %I64d seconds"),
time(NULL) - m_timeOpening));
m_timeOpening = 0;

// Check if this is a m2ts in a BD structure, and if it is, read some extra stream properties out of the CLPI files
if (m_pBluRay) {
    m_pBluRay->ProcessClipLanguages();
} else if (pszFileName && m_bMPEGTS) {
    CheckBDM2TSCPLI(pszFileName);
}

SAFE_CO_FREE(m_stOrigParser);
m_stOrigParser = (enum AVStreamParseType *)CoTaskMemAlloc(m_avFormat->nb_streams * sizeof(enum AVStreamParseType));
if (!m_stOrigParser)
    return E_OUTOFMEMORY;

for(unsigned int idx = 0; idx < m_avFormat->nb_streams; ++idx) {
    AVStream *st = m_avFormat->streams[idx];

    // Disable full stream parsing for these formats
    if (st->need_parsing == AVSTREAM_PARSE_FULL) {
        if (st->codec->codec_id == AV_CODEC_ID_DVB_SUBTITLE) {
            st->need_parsing = AVSTREAM_PARSE_NONE;
        }
    }

    if (m_bOgg && st->codec->codec_id == AV_CODEC_ID_H264) {
        st->need_parsing = AVSTREAM_PARSE_FULL;
    }

    // Create the parsers with the appropriate flags
    init_parser(m_avFormat, st);
}

```

```

UpdateParserFlags(st);

#ifndef DEBUG
    AVProgram *streamProg = av_find_program_from_stream(m_avFormat, NULL, idx);
    DbgLog((LOG_TRACE, 30, L"Stream %d (pid %d) - program: %d, codec: %S; parsing: %S;", idx, st->id, streamProg ?
            streamProg->pmt_pid : -1, avcodec_get_name(st->codec->codec_id), lavf_get_parsing_string(st->need_parsing)));
#endif

    m_stOrigParser[idx] = st->need_parsing;

    if ((st->codec->codec_id == AV_CODEC_ID_DTS && st->codec->codec_tag == 0xA2)
        || (st->codec->codec_id == AV_CODEC_ID_EAC3 && st->codec->codec_tag == 0xA1))
        st->disposition |= LAVF_DISPOSITION_SECONDARY_AUDIO;

    UpdateSubStreams();

    if (st->codec->codec_type == AVMEDIA_TYPE_ATTACHMENT && (st->codec->codec_id == AV_CODEC_ID_TTF || st->codec->codec_id == AV_CODEC_ID_OTF)) {
        if (!m_pFontInstaller) {
            m_pFontInstaller = new CFontInstaller();
        }
        m_pFontInstaller->InstallFont(st->codec->extradata, st->codec->extradata_size);
    }
}

CHECK_HR(hr = CreateStreams());

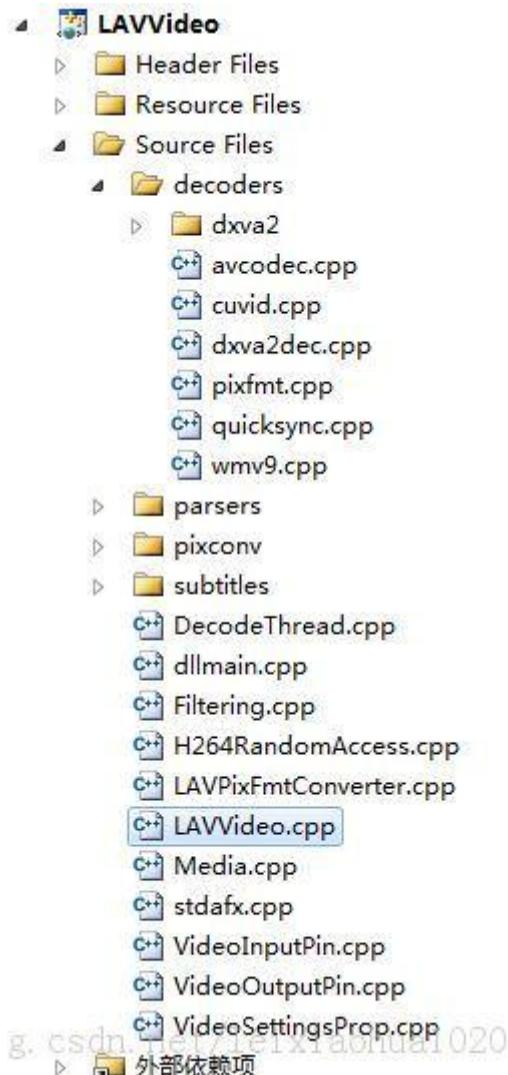
return S_OK;
done:
//关闭输入
CleanupAVFormat();
return E_FAIL;
}
该函数通过 avformat_find_stream_info() 等获取到流信息，这里就不多说了。

```

## LAV Filter 源代码分析 3： LAV Video （1）

LAV Video 是使用很广泛的 DirectShow Filter。它封装了 FFMPEG 中的 libavcodec，支持十分广泛的视频格式的解码。在这里对其源代码进行详细的分析。

LAV Video 工程代码的结构如下图所示



直接看 LAV Video 最主要的类 CLAVVideo 吧，它的定义位于 LAVVideo.h 中。

```
LAVVideo.h
[cpp] view plaincopy
/* 雷霄骅
 * 中国传媒大学/数字电视技术
 * leixiaohua1020@126.com
 *
 */
/*
 * Copyright (C) 2010-2013 Hendrik Leppkes
 * http://www.1f0.de
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 */
```

## 《FFmpeg 基础库编程开发》

```
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.

*
* You should have received a copy of the GNU General Public License along
* with this program; if not, write to the Free Software Foundation, Inc.,
* 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

*/
```

```
#pragma once
```

```
#include "decoders/ILAVDecoder.h"
#include "DecodeThread.h"
#include "ILAVPinInfo.h"

#include "LAVPixFmtConverter.h"
#include "LAVVideoSettings.h"
#include "H264RandomAccess.h"
#include "FloatingAverage.h"

#include "ISpecifyPropertyPages2.h"
#include "SynchronizedQueue.h"

#include "subtitles/LAVSubtitleConsumer.h"
#include "subtitles/LAVVideoSubtitleInputPin.h"

#include "BaseTrayIcon.h"

#define LAVC_VIDEO_REGISTRY_KEY L"Software\\LAV\\Video"
#define LAVC_VIDEO_REGISTRY_KEY_FORMATS L"Software\\LAV\\Video\\Formats"
#define LAVC_VIDEO_REGISTRY_KEY_OUTPUT L"Software\\LAV\\Video\\Output"
#define LAVC_VIDEO_REGISTRY_KEY_HWACCEL L"Software\\LAV\\Video\\HWAccel"

#define LAVC_VIDEO_LOG_FILE L"LAVVideo.txt"

#define DEBUG_FRAME_TIMINGS 0
#define DEBUG_PIXELCONV_TIMINGS 0

#define LAV_MT_FILTER_QUEUE_SIZE 4

typedef struct {
    REFERENCE_TIME rtStart;
```

```

REFERENCE_TIME rtStop;
} TimingCache;
//解码核心类
//Transform Filter
[uuid("EE30215D-164F-4A92-A4EB-9D4C13390F9F")]
class CLAVVideo : public CTransformFilter, public ISpecifyPropertyPages2, public ILAVVideoSettings, public
ILAVVideoStatus, public ILAVVideoCallback
{
public:
    CLAVVideo(LPUNKNOWN pUnk, HRESULT* phr);
    ~CLAVVideo();

static void CALLBACK StaticInit(BOOL bLoading, const CLSID *clsid);

// IUnknown
// 查找接口必须实现
DECLARE_IUNKNOWN;
STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void** ppv);

// ISpecifyPropertyPages2
// 属性页
// 获取或者创建
STDMETHODIMP GetPages(CAUUID *pPages);
STDMETHODIMP CreatePage(const GUID& guid, IPropertyPage** ppPage);

// ILAVVideoSettings
STDMETHODIMP SetRuntimeConfig(BOOL bRuntimeConfig);
STDMETHODIMP SetFormatConfiguration(LAVVideoCodec vCodec, BOOL bEnabled);
STDMETHODIMP_(BOOL) GetFormatConfiguration(LAVVideoCodec vCodec);
STDMETHODIMP SetNumThreads(DWORD dwNum);
STDMETHODIMP_(DWORD) GetNumThreads();
STDMETHODIMP SetStreamAR(DWORD bStreamAR);
STDMETHODIMP_(DWORD) GetStreamAR();
STDMETHODIMP SetPixelFormat(LAVOutPixFmts pixFmt, BOOL bEnabled);
STDMETHODIMP_(BOOL) GetPixelFormat(LAVOutPixFmts pixFmt);
STDMETHODIMP SetRGBOutputRange(DWORD dwRange);
STDMETHODIMP_(DWORD) GetRGBOutputRange();

STDMETHODIMP SetDeintFieldOrder(LAVDeintFieldOrder fieldOrder);
STDMETHODIMP_(LAVDeintFieldOrder) GetDeintFieldOrder();
STDMETHODIMP SetDeintForce(BOOL bForce);
STDMETHODIMP_(BOOL) GetDeintForce();

```

## 《FFmpeg 基础库编程开发》

```
STDMETHODIMP SetDeintAggressive(BOOL bAggressive);
STDMETHODIMP_(BOOL) GetDeintAggressive();

STDMETHODIMP_(DWORD) CheckHWAccelSupport(LAVHWAcel hwAccel);
STDMETHODIMP SetHWAccel(LAVHWAcel hwAccel);
STDMETHODIMP_(LAVHWAcel) GetHWAccel();
STDMETHODIMP SetHWAccelCodec(LAVVideoHWCodec hwAccelCodec, BOOL bEnabled);
STDMETHODIMP_(BOOL) GetHWAccelCodec(LAVVideoHWCodec hwAccelCodec);
STDMETHODIMP SetHWAccelDeintMode(LAVHWDeintModes deintMode);
STDMETHODIMP_(LAVHWDeintModes) GetHWAccelDeintMode();
STDMETHODIMP SetHWAccelDeintOutput(LAVDeintOutput deintOutput);
STDMETHODIMP_(LAVDeintOutput) GetHWAccelDeintOutput();
STDMETHODIMP SetHWAccelDeintHQ(BOOL bHQ);
STDMETHODIMP_(BOOL) GetHWAccelDeintHQ();
STDMETHODIMP SetSWDeintMode(LAVSWDeintModes deintMode);
STDMETHODIMP_(LAVSWDeintModes) GetSWDeintMode();
STDMETHODIMP SetSWDeintOutput(LAVDeintOutput deintOutput);
STDMETHODIMP_(LAVDeintOutput) GetSWDeintOutput();

STDMETHODIMP SetDeintTreatAsProgressive(BOOL bEnabled);
STDMETHODIMP_(BOOL) GetDeintTreatAsProgressive();

STDMETHODIMP SetDitherMode(LAVDitherMode ditherMode);
STDMETHODIMP_(LAVDitherMode) GetDitherMode();

STDMETHODIMP SetUseMSWMV9Decoder(BOOL bEnabled);
STDMETHODIMP_(BOOL) GetUseMSWMV9Decoder();

STDMETHODIMP SetDVDVideoSupport(BOOL bEnabled);
STDMETHODIMP_(BOOL) GetDVDVideoSupport();

STDMETHODIMP SetHWAccelResolutionFlags(DWORD dwResFlags);
STDMETHODIMP_(DWORD) GetHWAccelResolutionFlags();

STDMETHODIMP SetTrayIcon(BOOL bEnabled);
STDMETHODIMP_(BOOL) GetTrayIcon();

STDMETHODIMP SetDeinterlacingMode(LAVDeintMode deintMode);
STDMETHODIMP_(LAVDeintMode) GetDeinterlacingMode();

// ILAVVideoStatus
STDMETHODIMP_(const WCHAR *) GetActiveDecoderName() { return m_Decoder.GetDecoderName(); }
```

```

// CTransformFilter
// 核心的
HRESULT CheckInputType(const CMediaType* mtIn);
HRESULT CheckTransform(const CMediaType* mtIn, const CMediaType* mtOut);
HRESULT DecideBufferSize(IMemAllocator * pAllocator, ALLOCATOR_PROPERTIES *pprop);
HRESULT GetMediaType(int iPosition, CMediaType *pMediaType);

HRESULT SetMediaType(PIN_DIRECTION dir, const CMediaType *pmt);
HRESULT EndOfStream();
HRESULT BeginFlush();
HRESULT EndFlush();
HRESULT NewSegment(REFERENCE_TIME tStart, REFERENCE_TIME tStop, double dRate);
//处理的核心
//核心一般才有 IMediaSample
HRESULT Receive(IMediaSample *pIn);

HRESULT CheckConnect(PIN_DIRECTION dir, IPin *pPin);
HRESULT BreakConnect(PIN_DIRECTION dir);
HRESULT CompleteConnect(PIN_DIRECTION dir, IPin *pReceivePin);

int GetPinCount();
CBasePin* GetPin(int n);

STDMETHODIMP JoinFilterGraph(IFilterGraph * pGraph, LPCWSTR pName);

// ILAVVideoCallback
STDMETHODIMP AllocateFrame(LAVFrame **ppFrame);
STDMETHODIMP ReleaseFrame(LAVFrame **ppFrame);
STDMETHODIMP Deliver(LAVFrame *pFrame);
STDMETHODIMP_(LPWSTR) GetFileExtension();
STDMETHODIMP_(BOOL) FilterInGraph(PIN_DIRECTION dir, const GUID &clsid) { if (dir == PINDIR_INPUT)
return FilterInGraphSafe(m_pInput, clsid); else return FilterInGraphSafe(m_pOutput, clsid); }
STDMETHODIMP_(DWORD) GetDecodeFlags() { return m_dwDecodeFlags; }
STDMETHODIMP_(CMediaType&) GetInputMediaType() { return m_pInput->CurrentMediaType(); }
STDMETHODIMP GetLAVPinInfo(LAVPinInfo &info) { if (m_LAVPinInfoValid) { info = m_LAVPinInfo; return S_OK; } return E_FAIL; }
STDMETHODIMP_(CBasePin*) GetOutputPin() { return m_pOutput; }
STDMETHODIMP_(CMediaType&) GetOutputMediaType() { return m_pOutput->CurrentMediaType(); }
STDMETHODIMP DVDStripPacket(BYTE*& p, long& len)
{ static_cast<CDeCSSTransformInputPin*>(m_pInput)->StripPacket(p, len); return S_OK; }
STDMETHODIMP_(LAVFrame*) GetFlushFrame();
STDMETHODIMP ReleaseAllDXVAResources() { ReleaseLastSequenceFrame(); return S_OK; }

```

```
public:  
    // Pin Configuration  
    const static AMOVIESETUP_MEDIATYPE    sudPinTypesIn[];  
    const static int                     sudPinTypesInCount;  
    const static AMOVIESETUP_MEDIATYPE    sudPinTypesOut[];  
    const static int                     sudPinTypesOutCount;  
  
private:  
    HRESULT LoadDefaults();  
    HRESULT ReadSettings(HKEY rootKey);  
    HRESULT LoadSettings();  
    HRESULT SaveSettings();  
  
    HRESULT CreateTrayIcon();  
  
    HRESULT CreateDecoder(const CMediaType *pmt);  
  
    HRESULT GetDeliveryBuffer(IMediaSample** ppOut, int width, int height, AVRational ar, DXVA2_ExtendedFormat  
dxvaExtFormat, REFERENCE_TIME avgFrameDuration);  
    HRESULT ReconnectOutput(int width, int height, AVRational ar, DXVA2_ExtendedFormat dxvaExtFlags,  
REFERENCE_TIME avgFrameDuration, BOOL bDXVA = FALSE);  
  
    HRESULT SetFrameFlags(IMediaSample* pMS, LAVFrame *pFrame);  
  
    HRESULT NegotiatePixelFormat(CMediaType &mt, int width, int height);  
    BOOL IsInterlaced();  
  
    HRESULT Filter(LAVFrame *pFrame);  
    HRESULT DeliverToRenderer(LAVFrame *pFrame);  
  
    HRESULT PerformFlush();  
    HRESULT ReleaseLastSequenceFrame();  
  
    HRESULT GetD3DBuffer(LAVFrame *pFrame);  
    HRESULT RedrawStillImage();  
    HRESULT SetInDVDMenu(bool menu) { m_bInDVDMenu = menu; return S_OK; }  
  
    enum {CNTRL_EXIT, CNTRL_REDRAW};  
    HRESULT ControlCmd(DWORD cmd) {  
        return m_ControlThread->CallWorker(cmd);  
    }
```

private:

```

friend class CVideoOutputPin;
friend class CDecodeThread;
friend class CLAVControlThread;
friend class CLAVSubtitleProvider;
friend class CLAVSubtitleConsumer;
//解码线程
CDecodeThread      m_Decoder;
CAMThread          *m_ControlThread;

REFERENCE_TIME     m_rtPrevStart;
REFERENCE_TIME     m_rtPrevStop;

BOOL               m_bForceInputAR;
BOOL               m_bSendMediaType;
BOOL               m_bFlushing;

HRESULT            m_hrDeliver;

CLAVPixFmtConverter m_PixFmtConverter;
std::wstring        m_strExtension;

DWORD              m_bDXVAExtFormatSupport;
DWORD              m_bMadVR;
DWORD              m_bOverlayMixer;
DWORD              m_dwDecodeFlags;

BOOL               m_bInDVDMenu;

AVFilterGraph      *m_pFilterGraph;
AVFilterContext    *m_pFilterBufferSrc;
AVFilterContext    *m_pFilterBufferSink;

LAVPixelFormat     m_filterPixFmt;
int                m_filterWidth;
int                m_filterHeight;
LAVFrame           m_FilterPrevFrame;

BOOL               m_LAVPinInfoValid;
LAVPinInfo         m_LAVPinInfo;

CLAVVideoSubtitleInputPin *m_pSubtitleInput;
CLAVSubtitleConsumer *m_SubtitleConsumer;

```

```

LAVFrame *m_pLastSequenceFrame;

AM_SimpleRateChange m_DVDRate;

BOOL m_bRuntimeConfig;
struct VideoSettings {
    BOOL TrayIcon;
    DWORD StreamAR;
    DWORD NumThreads;
    BOOL bFormats[Codec_VideoNB];
    BOOL bMSWMV9DMO;
    BOOL bPixFmts[LAVOutPixFmt_NB];
    DWORD RGBRange;
    DWORD HWAccel;
    BOOL bHWFormats[HWCodec_NB];
    DWORD HWAccelResFlags;
    DWORD HWDeintMode;
    DWORD HWDeintOutput;
    BOOL HWDeintHQ;
    DWORD DeintFieldOrder;
    LAVDeintMode DeintMode;
    DWORD SWDeintMode;
    DWORD SWDeintOutput;
    DWORD DitherMode;
    BOOL bDVDDVideo;
} m_settings;
}

CBaseTrayIcon *m_pTrayIcon;

#endif DEBUG
    FloatingAverage<double> m_pixFmtTimingAvg;
#endif
};


```

可见该类继承了 CTransformFilter，其的功能真的是非常丰富的。在这里肯定无法对其进行一一分析，只能选择其中重点的函数进行一下分析。

该类中包含了解码线程类： CDecodeThread m\_Decoder;，这里封装了解码功能。

同时该类中包含了函数 Receive(IMediaSample \*pIn);，是发挥解码功能的函数，其中 pIn 是输入的解码前的视频压缩编码数据。

下面来看看 Receive()函数：

[cpp] view plaincopy  
//处理的核心

```

//核心一般才有 IMediaSample
HRESULT CLAVVideo::Receive(IMediaSample *pIn)
{
    CAutoLock cAutoLock(&m_csReceive);
    HRESULT hr = S_OK;

    AM_SAMPLE2_PROPERTIES const *pProps = m_pInput->SampleProps();
    if(pProps->dwStreamId != AM_STREAM_MEDIA) {
        return m_pOutput->Deliver(pIn);
    }

    AM_MEDIA_TYPE *pmt = NULL;
    //获取媒体类型等等
    if (SUCCEEDED(pIn->GetMediaType(&pmt)) && pmt) {
        CMediaType mt = *pmt;
        DeleteMediaType(pmt);
        if (mt != m_pInput->CurrentMediaType() || !(m_dwDecodeFlags & LAV_VIDEO_DEC_FLAG_DVD)) {
            DbgLog((LOG_TRACE, 10, L"::Receive(): Input sample contained media type, dynamic format change..."));
            m_Decoder.EndOfStream();
            hr = m_pInput->SetMediaType(&mt);
            if (FAILED(hr)) {
                DbgLog((LOG_ERROR, 10, L"::Receive(): Setting new media type failed..."));
                return hr;
            }
        }
    }
}

m_hrDeliver = S_OK;

// Skip over empty packets
if (pIn->GetActualDataLength() == 0) {
    return S_OK;
}
//解码
hr = m_Decoder.Decode(pIn);
if (FAILED(hr))
    return hr;

if (FAILED(m_hrDeliver))
    return m_hrDeliver;

return S_OK;
}

```

## 《FFmpeg 基础库编程开发》

由代码我们可以看出，实际发挥出解码功能的函数是 `hr = m_Decoder.Decode(pIn);`。

下面我们来看看 CDecodeThread 类的 Decode()方法：

```
[cpp] view plaincopy
//解码线程的解码函数
STDMETHODIMP CDecodeThread::Decode(IMediaSample *pSample)
{
    CAutoLock decoderLock(this);

    if (!CAMThread::ThreadExists())
        return E_UNEXPECTED;

    // Wait until the queue is empty
    while(HasSample())
        Sleep(1);

    // Re-init the decoder, if requested
    // Doing this inside the worker thread alone causes problems
    // when switching from non-sync to sync, so ensure we're in sync.
    if (m_bDecoderNeedsReInit) {
        CAMThread::CallWorker(CMD_REINIT);
        while (!m_evEOSDone.Check()) {
            m_evSample.Wait();
            ProcessOutput();
        }
    }

    m_evDeliver.Reset();
    m_evSample.Reset();
    m_evDecodeDone.Reset();

    pSample->AddRef();

    // Send data to worker thread, and wake it (if it was waiting)
    PutSample(pSample);

    // If we don't have thread safe buffers, we need to synchronize
    // with the worker thread and deliver them when they are available
    // and then let it know that we did so
    if (m_bSyncToProcess) {
        while (!m_evDecodeDone.Check()) {
            m_evSample.Wait();
            ProcessOutput();
        }
    }
}
```

## 《FFmpeg 基础库编程开发》

}

ProcessOutput();

return S\_OK;

}

这个方法乍一看感觉很抽象，好像没看见直接调用任何解码的函数。如果 LAVVideo 的封装的 ffmpeg 的 libavcodec 的话，应该是最终调用 avcodec\_decode\_video2() 才对啊。。。先来看看 CDecodeThread 这个类的定义吧！

DecodeThread.h

[cpp] view plaincopy

/\* 雷霄骅

\* 中国传媒大学/数字电视技术

\* leixiaohua1020@126.com

\*

\*/

/\*

\* Copyright (C) 2010-2013 Hendrik Leppkes

\* http://www.1f0.de

\*

\* This program is free software; you can redistribute it and/or modify

\* it under the terms of the GNU General Public License as published by

\* the Free Software Foundation; either version 2 of the License, or

\* (at your option) any later version.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\* You should have received a copy of the GNU General Public License along

\* with this program; if not, write to the Free Software Foundation, Inc.,

\* 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

\*/

#pragma once

#include "decoders/ILAVDecoder.h"

#include "SynchronizedQueue.h"

class CLAVVideo;

class CDecodeThread : public ILAVVideoCallback, protected CAMThread, protected CCritSec

{

## 《FFmpeg 基础库编程开发》

```
public:
    CDecodeThread(CLAVVideo *pLAVVideo);
    ~CDecodeThread();

    // Parts of ILAVDecoder
    STDMETHODIMP_(const WCHAR*) GetDecoderName() { return m_pDecoder ? m_pDecoder->GetDecoderName() : NULL; }
    STDMETHODIMP_(long) GetBufferCount() { return m_pDecoder ? m_pDecoder->GetBufferCount() : 4; }
    STDMETHODIMP_(BOOL) IsInterlaced() { return m_pDecoder ? m_pDecoder->IsInterlaced() : TRUE; }
    STDMETHODIMP GetPixelFormat(LAVPixelFormat *pPix, int *pBpp) { ASSERT(m_pDecoder); return m_pDecoder->GetPixelFormat(pPix, pBpp); }
    STDMETHODIMP_(REFERENCE_TIME) GetFrameDuration() { ASSERT(m_pDecoder); return m_pDecoder->GetFrameDuration(); }
    STDMETHODIMP HasThreadSafeBuffers() { return m_pDecoder ? m_pDecoder->HasThreadSafeBuffers() : S_FALSE; }

    STDMETHODIMP CreateDecoder(const CMediaType *pmt, AVCCodecID codec);
    STDMETHODIMP Close();
    //解码线程的解码函数
    STDMETHODIMP Decode(IMediaSample *pSample);
    STDMETHODIMP Flush();
    STDMETHODIMP EndOfStream();

    STDMETHODIMP InitAllocator(IMemAllocator **ppAlloc);
    STDMETHODIMP PostConnect(IPin *pPin);

    STDMETHODIMP_(BOOL) IsHWDecoderActive() { return m_bHWDecoder; }

    // ILAVVideoCallback
    STDMETHODIMP AllocateFrame(LAVFrame **ppFrame);
    STDMETHODIMP ReleaseFrame(LAVFrame **ppFrame);
    STDMETHODIMP Deliver(LAVFrame *pFrame);
    STDMETHODIMP_(LPWSTR) GetFileExtension();
    STDMETHODIMP_(BOOL) FilterInGraph(PIN_DIRECTION dir, const GUID &clsid);
    STDMETHODIMP_(DWORD) GetDecodeFlags();
    STDMETHODIMP_(CMediaType&) GetInputMediaType();
    STDMETHODIMP GetLAVPinInfo(LAVPinInfo &info);
    STDMETHODIMP_(CBasePin*) GetOutputPin();
    STDMETHODIMP_(CMediaType&) GetOutputMediaType();
    STDMETHODIMP DVStripPacket(BYTE*& p, long& len);
    STDMETHODIMP_(LAVFrame*) GetFlushFrame();
    STDMETHODIMP ReleaseAllDXVAResources();
```

protected:

```
//包含了对进程的各种操作，重要  
DWORD ThreadProc();
```

private:

```
STDMETHODIMP CreateDecoderInternal(const CMediaType *pmt, AVCodecID codec);  
STDMETHODIMP PostConnectInternal(IPin *pPin);
```

```
STDMETHODIMP DecodeInternal(IMediaSample *pSample);
```

```
STDMETHODIMP ClearQueues();
```

```
STDMETHODIMP ProcessOutput();
```

```
bool HasSample();
```

```
void PutSample(IMediaSample *pSample);
```

```
IMediaSample* GetSample();
```

```
void ReleaseSample();
```

```
bool CheckForEndOfSequence(IMediaSample *pSample);
```

private:

//各种对进程进行的操作

```
enum {CMD_CREATE_DECODER, CMD_CLOSE_DECODER, CMD_FLUSH, CMD_EOS, CMD_EXIT,  
CMD_INIT_ALLOCATOR, CMD_POST_CONNECT, CMD_REINIT};
```

//注意 DecodeThread 像是一个处于中间位置的东西

//连接了 Filter 核心类 CLAVVideo 和解码器的接口 ILAVDecoder

```
CLAVVideo *m_pLAVVideo;
```

```
ILAVDecoder *m_pDecoder;
```

```
AVCodecID m_Codec;
```

```
BOOL m_bHWDecoder;
```

```
BOOL m_bHWDecoderFailed;
```

```
BOOL m_bSyncToProcess;
```

```
BOOL m_bDecoderNeedsReInit;
```

```
CAMEvent m_evInput;
```

```
CAMEvent m_evDeliver;
```

```
CAMEvent m_evSample;
```

```
CAMEvent m_evDecodeDone;
```

```
CAMEvent m_evEOSDone;
```

```
CCritSec m_ThreadCritSec;
```

```

struct {
    const CMediaType *pmt;
    AVCodecID codec;
    IMemAllocator **allocator;
    IPin *pin;
} m_ThreadCallContext;
CSynchronizedQueue<LAVFrame *> m_Output;

CCritSec      m_SampleCritSec;
IMediaSample *m_NextSample;

IMediaSample *m_TempSample[2];
IMediaSample *m_FailedSample;

std::wstring m_processName;
};


```

从名字上我们可以判断，这个类用于管理解码的线程。在这里我们关注该类里面的两个指针变量：`CLAVVideo *m_pLAVVideo;`

`ILAVDecoder *m_pDecoder;`

其中第一个指针变量就是这个工程中最核心的类 `CLAVVideo`，而第二个指针变量则是解码器的接口。通过这个接口就可以调用具体解码器的相应方法了。（注：在源代码中发现，解码器不光包含 `libavcodec`，也可以是 `wmv9` 等等，换句话说，是可以扩展其他种类的解码器的。不过就目前的情况来看，`lavvideo` 似乎不如 `ffdshow` 支持的解码器种类多）

该类里面还有一个函数：

`ThreadProc()`

该函数中包含了对线程的各种操作，其中包含调用了 `ILAVDecoder` 接口的各种方法：

[cpp] view plaincopy

//包含了对进程的各种操作

`DWORD CDecodeThread::ThreadProc()`

{

    HRESULT hr;

    DWORD cmd;

    BOOL bEOS = FALSE;

    BOOL bReinit = FALSE;

    SetThreadName(-1, "LAVVideo Decode Thread");

    HANDLE hWaitEvents[2] = { GetRequestHandle(), m\_evInput };

    //不停转圈，永不休止

    while(1) {

        if (!bEOS && !bReinit) {

            // Wait for either an input sample, or an request

## 《FFmpeg 基础库编程开发》

```
WaitForMultipleObjects(2, hWaitEvents, FALSE, INFINITE);  
}  
//根据操作命令的不同  
if (CheckRequest(&cmd)) {  
    switch (cmd) {  
        //创建解码器  
        case CMD_CREATE_DECODER:  
        {  
            CAutoLock lock(&m_ThreadCritSec);  
            //创建  
            hr = CreateDecoderInternal(m_ThreadCallContext.pmt, m_ThreadCallContext.codec);  
            Reply(hr);  
  
            m_ThreadCallContext.pmt = NULL;  
        }  
        break;  
        case CMD_CLOSE_DECODER:  
        {  
            //关闭  
            ClearQueues();  
            SAFE_DELETE(m_pDecoder);  
            Reply(S_OK);  
        }  
        break;  
        case CMD_FLUSH:  
        {  
            //清楚  
            ClearQueues();  
            m_pDecoder->Flush();  
            Reply(S_OK);  
        }  
        break;  
        case CMD_EOS:  
        {  
            bEOS = TRUE;  
            m_evEOSDone.Reset();  
            Reply(S_OK);  
        }  
        break;  
        case CMD_EXIT:  
        {  
            //退出  
            Reply(S_OK);  
    }
```

```

    return 0;
}
break;
case CMD_INIT_ALLOCATOR:
{
    CAutoLock lock(&m_ThreadCritSec);
    hr = m_pDecoder->InitAllocator(m_ThreadCallContext.allocator);
    Reply(hr);

    m_ThreadCallContext.allocator = NULL;
}
break;
case CMD_POST_CONNECT:
{
    CAutoLock lock(&m_ThreadCritSec);
    hr = PostConnectInternal(m_ThreadCallContext.pin);
    Reply(hr);

    m_ThreadCallContext.pin = NULL;
}
break;
case CMD_REINIT:
{
    //重启
    CMediaType &mt = m_pLAVVideo->GetInputMediaType();
    CreateDecoderInternal(&mt, m_Codec);
    m_TempSample[1] = m_NextSample;
    m_NextSample = m_FailedSample;
    m_FailedSample = NULL;
    bReinit = TRUE;
    m_evEOSDone.Reset();
    Reply(S_OK);
    m_bDecoderNeedsReInit = FALSE;
}
break;
default:
    ASSERT(0);
}
}

if (m_bDecoderNeedsReInit) {
    m_evInput.Reset();
    continue;
}

```

```

}

if (bReinit && !m_NextSample) {
    if (m_TempSample[0]) {
        m_NextSample = m_TempSample[0];
        m_TempSample[0] = NULL;
    } else if (m_TempSample[1]) {
        m_NextSample = m_TempSample[1];
        m_TempSample[1] = NULL;
    } else {
        bReinit = FALSE;
        m_evEOSDone.Set();
        m_evSample.Set();
        continue;
    }
}

//获得一份数据
IMediaSample *pSample = GetSample();
if (!pSample) {
    // Process the EOS now that the sample queue is empty
    if (bEOS) {
        bEOS = FALSE;
        m_pDecoder->EndOfStream();
        m_evEOSDone.Set();
        m_evSample.Set();
    }
    continue;
}

//解码
DecodeInternal(pSample);

// Release the sample
//释放
SafeRelease(&pSample);

// Indicates we're done decoding this sample
m_evDecodeDone.Set();

// Set the Sample Event to unblock any waiting threads
m_evSample.Set();
}

return 0;

```

}

先分析到这里了，至于 ILAVDecoder 接口方面的东西下篇文章再写。

## LAV Filter 源代码分析 4： LAV Video （2）

文章中提到 LAVVideo 主要通过 CDecodeThread 这个类进行解码线程的管理，其中有一个关键的管理函数：ThreadProc()，包含了对解码线程的各种操作。函数如下所示：

```
//包含了对进程的各种操作
DWORD CDecodeThread::ThreadProc()
{
    HRESULT hr;
    DWORD cmd;

    BOOL bEOS = FALSE;
    BOOL bReinit = FALSE;

    SetThreadName(-1, "LAVVideo Decode Thread");

    HANDLE hWaitEvents[2] = { GetRequestHandle(), m_evInput };
    //不停转圈，永不休止
    while(1) {
        if (!bEOS && !bReinit) {
            // Wait for either an input sample, or an request
            WaitForMultipleObjects(2, hWaitEvents, FALSE, INFINITE);
        }
        //根据操作命令的不同
        if (CheckRequest(&cmd)) {
            switch (cmd) {
                //创建解码器
                case CMD_CREATE_DECODER:
                {
                    CAutoLock lock(&m_ThreadCritSec);
                    //创建
                    hr = CreateDecoderInternal(m_ThreadCallContext.pmt, m_ThreadCallContext.codec);
                    Reply(hr);

                    m_ThreadCallContext.pmt = NULL;
                }
                break;
                case CMD_CLOSE_DECODER:
                {
                    //关闭
                }
            }
        }
    }
}
```

```

ClearQueues();
SAFE_DELETE(m_pDecoder);
Reply(S_OK);
}
break;
case CMD_FLUSH:
{
    //清楚
    ClearQueues();
    m_pDecoder->Flush();
    Reply(S_OK);
}
break;
case CMD_EOS:
{
    bEOS = TRUE;
    m_evEOSDone.Reset();
    Reply(S_OK);
}
break;
case CMD_EXIT:
{
    //退出
    Reply(S_OK);
    return 0;
}
break;
case CMD_INIT_ALLOCATOR:
{
    CAutoLock lock(&m_ThreadCritSec);
    hr = m_pDecoder->InitAllocator(m_ThreadCallContext.allocator);
    Reply(hr);

    m_ThreadCallContext.allocator = NULL;
}
break;
case CMD_POST_CONNECT:
{
    CAutoLock lock(&m_ThreadCritSec);
    hr = PostConnectInternal(m_ThreadCallContext.pin);
    Reply(hr);

    m_ThreadCallContext.pin = NULL;
}

```

```

    }
    break;
case CMD_REINIT:
{
    //重启
    CMediaType &mt = m_pLAVVideo->GetInputMediaType();
    CreateDecoderInternal(&mt, m_Codec);
    m_TempSample[1] = m_NextSample;
    m_NextSample = m_FailedSample;
    m_FailedSample = NULL;
    bReinit = TRUE;
    m_evEOSDone.Reset();
    Reply(S_OK);
    m_bDecoderNeedsReInit = FALSE;
}
break;
default:
ASSERT(0);
}
}

if (m_bDecoderNeedsReInit) {
    m_evInput.Reset();
    continue;
}

if (bReinit && !m_NextSample) {
    if (m_TempSample[0]) {
        m_NextSample = m_TempSample[0];
        m_TempSample[0] = NULL;
    } else if (m_TempSample[1]) {
        m_NextSample = m_TempSample[1];
        m_TempSample[1] = NULL;
    } else {
        bReinit = FALSE;
        m_evEOSDone.Set();
        m_evSample.Set();
        continue;
    }
}

//获得一份数据
IMediaSample *pSample = GetSample();
if (!pSample) {

```

《FFmpeg 基础库编程开发》

```

// Process the EOS now that the sample queue is empty
if (bEOS) {
    bEOS = FALSE;
    m_pDecoder->EndOfStream();
    m_evEOSDone.Set();
    m_evSample.Set();
}
continue;
}

//解码
DecodeInternal(pSample);

// Release the sample
//释放
SafeRelease(&pSample);

// Indicates we're done decoding this sample
m_evDecodeDone.Set();

// Set the Sample Event to unblock any waiting threads
m_evSample.Set();
}

return 0;
}

该函数中， DecodeInternal(pSample)为实际上真正具有解码功能的函数，来看看它的源代码吧：
STDMETHODIMP CDecodeThread::DecodeInternal(IMediaSample *pSample)
{
    HRESULT hr = S_OK;

    if (!m_pDecoder)
        return E_UNEXPECTED;
    //调用接口进行解码
    hr = m_pDecoder->Decode(pSample);

    // If a hardware decoder indicates a hard failure, we switch back to software
    // This is used to indicate incompatible media
    if (FAILED(hr) && m_bHWDecoder) {
        DbgLog((LOG_TRACE, 10, L"::Receive(): Hardware decoder indicates failure, switching back to software"));
        m_bHWDecoderFailed = TRUE;
    }

    // Store the failed sample for re-try in a moment
    m_FailedSample = pSample;
}

```

```

m_FailedSample->AddRef();

// Schedule a re-init when the main thread goes there the next time
m_bDecoderNeedsReInit = TRUE;

// Make room in the sample buffer, to ensure the main thread can get in
m_TempSample[0] = GetSample();
}

return S_OK;
}

```

该函数比较简短，从源代码中可以看出，调用了 m\_pDecoder 的 Decode()方法。其中 m\_pDecoder 为 ILAVDecoder 类型的指针，而 ILAVDecoder 是一个接口，并不包含实际的方法，如下所示。注意，从程序注释中可以看出，每一个解码器都需要实现该接口规定的函数。

```

//接口
interface ILAVDecoder
{
    /**
     * Virtual destructor
     */
    virtual ~ILAVDecoder(void) { };

    /**
     * Initialize interfaces with the LAV Video core
     * This function should also be used to create all interfaces with external DLLs
     *
     * @param pSettings reference to the settings interface
     * @param pCallback reference to the callback interface
     * @return S_OK on success, error code if this decoder is lacking an external support dll
     */
    STDMETHOD(InitInterfaces)(ILAVVideoSettings *pSettings, ILAVVideoCallback *pCallback) PURE;

    /**
     * Check if the decoder is functional
     */
    STDMETHOD(Check)() PURE;

    /**
     * Initialize the codec to decode a stream specified by codec and pmt.
     *
     * @param codec Codec Id
     * @param pmt DirectShow Media Type
     * @return S_OK on success, an error code otherwise
     */
}

```

```
*/
STDMETHOD(InitDecoder)(AVCodecID codec, const CMediaType *pmt) PURE;

/***
 * Decode a frame.
 *
 * @param pSample Media Sample to decode
 * @return S_OK if decoding was successfull, S_FALSE if no frame could be extracted, an error code if the decoder is
not compatible with the bitstream
 *
 * Note: When returning an actual error code, the filter will switch to the fallback software decoder! This should only be
used for catastrophic failures,
 * like trying to decode a unsupported format on a hardware decoder.
 */
STDMETHOD(Decode)(IMediaSample *pSample) PURE;

/***
 * Flush the decoder after a seek.
 * The decoder should discard any remaining data.
 *
 * @return unused
 */
STDMETHOD(Flush)() PURE;

/***
 * End of Stream
 * The decoder is asked to output any buffered frames for immediate delivery
 *
 * @return unused
 */
STDMETHOD(EndOfStream)() PURE;

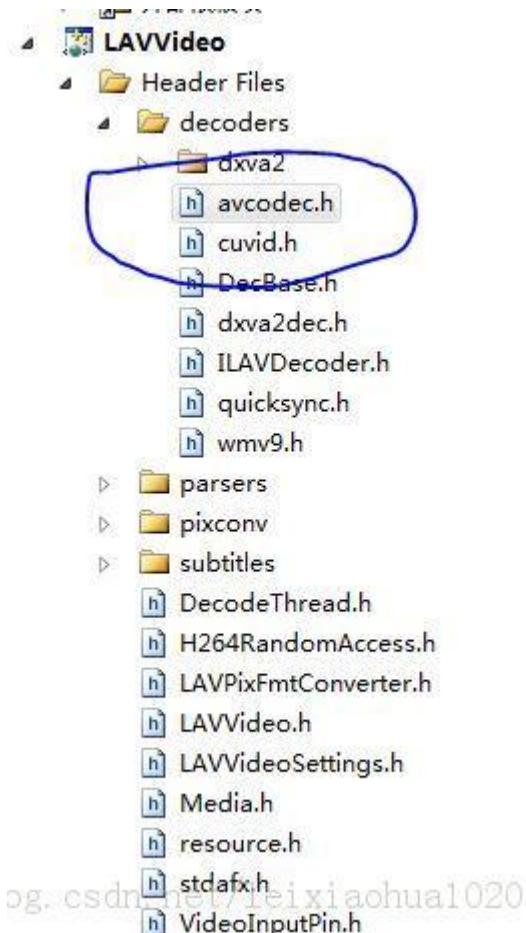
/***
 * Query the decoder for the current pixel format
 * Mostly used by the media type creation logic before playback starts
 *
 * @return the pixel format used in the decoding process
 */
STDMETHOD(GetPixelFormat)(LAVPixelFormat *pPix, int *pBpp) PURE;
```

```
/***
 * Get the frame duration.
 */
```

## 《FFmpeg 基础库编程开发》

```
* This function is not mandatory, and if you cannot provide any specific duration, return 0.  
*/  
STDMETHOD_(REFERENCE_TIME, GetFrameDuration)() PURE;  
  
/**  
 * Query whether the format can potentially be interlaced.  
 * This function should return false if the format can 100% not be interlaced, and true if it can be interlaced (but also  
progressive).  
 */  
STDMETHOD_(BOOL, IsInterlaced)() PURE;  
  
/**  
 * Allows the decoder to handle an allocator.  
 * Used by DXVA2 decoding  
 */  
STDMETHOD(InitAllocator)(IMemAllocator **ppAlloc) PURE;  
  
/**  
 * Function called after connection is established, with the pin as argument  
 */  
STDMETHOD(PostConnect)(IPin *pPin) PURE;  
  
/**  
 * Get the number of sample buffers optimal for this decoder  
 */  
STDMETHOD_(long, GetBufferCount)() PURE;  
  
/**  
 * Get the name of the decoder  
 */  
STDMETHOD_(const WCHAR*, GetDecoderName)() PURE;  
  
/**  
 * Get whether the decoder outputs thread-safe buffers  
 */  
STDMETHOD(HasThreadSafeBuffers)() PURE;  
  
/**  
 * Get whether the decoder should sync to the main thread  
 */  
STDMETHOD(SyncToProcessThread)() PURE;  
};
```

下面来看看封装 libavcodec 库的类吧，该类的定义位于 decoders 文件夹下，名为 avcodec.h，如图所示：



该类名字叫 CDecAvcodec，其继承了 CDecBase。而 CDecBase 继承了 ILAVDecoder。

```
/*
 * Copyright (C) 2010-2013 Hendrik Leppkes
 * http://www.1f0.de
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along
 * with this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
*/

```

```
#pragma once
```

```
#include "DecBase.h"
#include "H264RandomAccess.h"

#include <map>

#define AVCODEC_MAX_THREADS 16

typedef struct {
    REFERENCE_TIME rtStart;
    REFERENCE_TIME rtStop;
} TimingCache;

//解码器（AVCODEC）（其实还有 WMV9， CUVID 等）
class CDecAvcodec : public CDecBase
{
public:
    CDecAvcodec(void);
    virtual ~CDecAvcodec(void);

    // ILAVDecoder
    STDMETHODIMP InitDecoder(AVCodecID codec, const CMediaType *pmt);
    //解码
    STDMETHODIMP Decode(const BYTE *buffer, int buflen, REFERENCE_TIME rtStart, REFERENCE_TIME rtStop,
    BOOL bSyncPoint, BOOL bDiscontinuity);
    STDMETHODIMP Flush();
    STDMETHODIMP EndOfStream();
    STDMETHODIMP GetPixelFormat(LAVPixelFormat *pPix, int *pBpp);
    STDMETHODIMP_(REFERENCE_TIME) GetFrameDuration();
    STDMETHODIMP_(BOOL) IsInterlaced();
    STDMETHODIMP_(const WCHAR*) GetDecoderName() { return L"avcodec"; }
    STDMETHODIMP HasThreadSafeBuffers() { return S_OK; }
    STDMETHODIMP SyncToProcessThread() { return m_pAVCtx && m_pAVCtx->thread_count > 1 ? S_OK : S_FALSE; }

    // CDecBase
    STDMETHODIMP Init();

protected:
    virtual HRESULT AddDecoderInit() { return S_FALSE; }
    virtual HRESULT PostDecode() { return S_FALSE; }
    virtual HRESULT HandleDXVA2Frame(LAVFrame *pFrame) { return S_FALSE; }
    //销毁解码器，各种 Free
```

## 《FFmpeg 基础库编程开发》

STDMETHODIMP DestroyDecoder();

private:

STDMETHODIMP ConvertPixFmt(AVFrame \*pFrame, LAVFrame \*pOutFrame);

protected:

AVCodecContext \*m\_pAVCtx;  
AVFrame \*m\_pFrame;  
AVCodecID m\_nCodecId;  
BOOL m\_bDXVA;

private:

AVCodec \*m\_pAVCodec;  
AVCodecParserContext \*m\_pParser;

BYTE \*m\_pFFBuffer;  
BYTE \*m\_pFFBuffer2;  
int m\_nFFBufferSize;  
int m\_nFFBufferSize2;

SwsContext \*m\_pSwsContext;

CH264RandomAccess m\_h264RandomAccess;

BOOL m\_bNoBufferConsumption;  
BOOL m\_bHasPalette;

// Timing settings

BOOL m\_bFFReordering;  
BOOL m\_bCalculateStopTime;  
BOOL m\_bRVDropBFrameTimings;  
BOOL m\_bInputPadded;

BOOL m\_bBFrameDelay;  
TimingCache m\_tcBFrameDelay[2];  
int m\_nBFramePos;

TimingCache m\_tcThreadBuffer[AVCODEC\_MAX\_THREADS];  
int m\_CurrentThread;

REFERENCE\_TIME m\_rtStartCache;  
BOOL m\_bResumeAtKeyFrame;  
BOOL m\_bWaitingForKeyFrame;

```

int          m_iInterlaced;
};

从 CDecAvcodec 类的定义可以看出，包含了各种功能的函数。首先我们看看初始化函数 Init()
[cpp] view plaincopy
// ILAVDecoder
STDMETHODIMP CDecAvcodec::Init()
{
#ifndef DEBUG
    DbgSetModuleLevel (LOG_CUSTOM1, DWORD_MAX); // FFMPEG messages use custom1
    av_log_set_callback(lavf_log_callback);
#else
    av_log_set_callback(NULL);
#endif
//注册
avcodec_register_all();
return S_OK;
}

```

可见其调用了 ffmpeg 的 API 函数 avcodec\_register\_all() 进行了解码器的注册。

我们再来看看其解码函数 Decode():

```

//解码
STDMETHODIMP CDecAvcodec::Decode(const BYTE *buffer, int buflen, REFERENCE_TIME rtStartIn,
REFERENCE_TIME rtStopIn, BOOL bSyncPoint, BOOL bDiscontinuity)
{
    int      got_picture = 0;
    int      used_bytes  = 0;
    BOOL     bParserFrame = FALSE;
    BOOL     bFlush = (buffer == NULL);
    BOOL     bEndOfSequence = FALSE;
//初始化 Packet
AVPacket avpkt;
av_init_packet(&avpkt);

if (m_pAVCtx->active_thread_type & FF_THREAD_FRAME) {
    if (!m_bFFReordering) {
        m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
        m_tcThreadBuffer[m_CurrentThread].rtStop  = rtStopIn;
    }
}

m_CurrentThread = (m_CurrentThread + 1) % m_pAVCtx->thread_count;
} else if (m_bBFrameDelay) {
    m_tcBFrameDelay[m_nBFramePos].rtStart = rtStartIn;
    m_tcBFrameDelay[m_nBFramePos].rtStop = rtStopIn;
    m_nBFramePos = !m_nBFramePos;
}

```

```

}

uint8_t *pDataBuffer = NULL;
if (!bFlush && buflen > 0) {
    if (!m_bInputPadded && (!(m_pAVCtx->active_thread_type & FF_THREAD_FRAME) || m_pParser)) {
        // Copy bitstream into temporary buffer to ensure overread protection
        // Verify buffer size
        if (buflen > m_nFFBufferSize) {
            m_nFFBufferSize = buflen;
            m_pFFBuffer = (BYTE *)av_realloc_f(m_pFFBuffer, m_nFFBufferSize + FF_INPUT_BUFFER_PADDING_SIZE, 1);
            if (!m_pFFBuffer) {
                m_nFFBufferSize = 0;
                return E_OUTOFMEMORY;
            }
        }
    }

    memcpy(m_pFFBuffer, buffer, buflen);
    memset(m_pFFBuffer+buflen, 0, FF_INPUT_BUFFER_PADDING_SIZE);
    pDataBuffer = m_pFFBuffer;
} else {
    pDataBuffer = (uint8_t *)buffer;
}

if (m_nCodecId == AV_CODEC_ID_H264) {
    BOOL bRecovered = m_h264RandomAccess.searchRecoveryPoint(pDataBuffer, buflen);
    if (!bRecovered) {
        return S_OK;
    }
} else if (m_nCodecId == AV_CODEC_ID_VP8 && m_bWaitingForKeyFrame) {
    if (!(pDataBuffer[0] & 1)) {
        DbgLog((LOG_TRACE, 10, L":::Decode(): Found VP8 key-frame, resuming decoding"));
        m_bWaitingForKeyFrame = FALSE;
    } else {
        return S_OK;
    }
}
}

while (buflen > 0 || bFlush) {
    REFERENCE_TIME rtStart = rtStartIn, rtStop = rtStopIn;

    if (!bFlush) {

```

```

//设置 AVPacket 中的数据
avpkt.data = pDataBuffer;
avpkt.size = buflen;
avpkt pts = rtStartIn;
if (rtStartIn != AV_NOPTS_VALUE && rtStopIn != AV_NOPTS_VALUE)
    avpkt.duration = (int)(rtStopIn - rtStartIn);
else
    avpkt.duration = 0;
avpkt.flags = AV_PKT_FLAG_KEY;

if (m_bHasPalette) {
    m_bHasPalette = FALSE;
    uint32_t *pal = (uint32_t *)av_packet_new_side_data(&avpkt, AV_PKT_DATA_PALETTE,
AVPALETTE_SIZE);
    int pal_size = FFMIN((1 << m_pAVCtx->bits_per_coded_sample) << 2, m_pAVCtx->extradata_size);
    uint8_t *pal_src = m_pAVCtx->extradata + m_pAVCtx->extradata_size - pal_size;

    for (int i = 0; i < pal_size/4; i++)
        pal[i] = 0xFF<<24 | AV_RL32(pal_src+4*i);
}
} else {
    avpkt.data = NULL;
    avpkt.size = 0;
}

// Parse the data if a parser is present
// This is mandatory for MPEG-1/2
// 不一定需要
if (m_pParser) {
    BYTE *pOut = NULL;
    int pOut_size = 0;

    used_bytes = av_parser_parse2(m_pParser, m_pAVCtx, &pOut, &pOut_size, avpkt.data, avpkt.size,
AV_NOPTS_VALUE, AV_NOPTS_VALUE, 0);

    if (used_bytes == 0 && pOut_size == 0 && !bFlush) {
        DbgLog(LOG_TRACE, 50, L"::Decode() - could not process buffer, starving?");
        break;
    }

    // Update start time cache
    // If more data was read then output, update the cache (incomplete frame)
    // If output is bigger, a frame was completed, update the actual rtStart with the cached value, and then overwrite the

```

cache

```

if (used_bytes > pOut_size) {
    if (rtStartIn != AV_NOPTS_VALUE)
        m_rtStartCache = rtStartIn;
} else if (used_bytes == pOut_size || ((used_bytes + 9) == pOut_size)) {
    // Why +9 above?
    // Well, apparently there are some broken MKV muxers that like to mux the MPEG-2 PICTURE_START_CODE
    // block (which is 9 bytes) in the package with the previous frame
    // This would cause the frame timestamps to be delayed by one frame exactly, and cause timestamp reordering to
    // go wrong.
    // So instead of failing on those samples, lets just assume that 9 bytes are that case exactly.
    m_rtStartCache = rtStartIn = AV_NOPTS_VALUE;
} else if (pOut_size > used_bytes) {
    rtStart = m_rtStartCache;
    m_rtStartCache = rtStartIn;
    // The value was used once, don't use it for multiple frames, that ends up in weird timings
    rtStartIn = AV_NOPTS_VALUE;
}

bParserFrame = (pOut_size > 0);

if (pOut_size > 0 || bFlush) {

    if (pOut && pOut_size > 0) {
        if (pOut_size > m_nFFBufferSize2) {
            m_nFFBufferSize2 = pOut_size;
            m_pFFBuffer2 = (BYTE *)av_realloc_f(m_pFFBuffer2, m_nFFBufferSize2 +
FF_INPUT_BUFFER_PADDING_SIZE, 1);
            if (!m_pFFBuffer2) {
                m_nFFBufferSize2 = 0;
                return E_OUTOFMEMORY;
            }
        }
        memcpy(m_pFFBuffer2, pOut, pOut_size);
        memset(m_pFFBuffer2+pOut_size, 0, FF_INPUT_BUFFER_PADDING_SIZE);

        avpkt.data = m_pFFBuffer2;
        avpkt.size = pOut_size;
        avpkt.pts = rtStart;
        avpkt.duration = 0;

        const uint8_t *eosmarker = CheckForEndOfSequence(m_nCodecId, avpkt.data, avpkt.size,
&m_MpegParserState);
    }
}

```

```

if (eosmarker) {
    bEndOfSequence = TRUE;
}
} else {
    avpkt.data = NULL;
    avpkt.size = 0;
}
//真正的解码
int ret2 = avcodec_decode_video2 (m_pAVCtx, m_pFrame, &got_picture, &avpkt);
if (ret2 < 0) {
    DbgLog((LOG_TRACE, 50, L"::Decode() - decoding failed despite successfull parsing"));
    got_picture = 0;
}
} else {
    got_picture = 0;
}
} else {
    used_bytes = avcodec_decode_video2 (m_pAVCtx, m_pFrame, &got_picture, &avpkt);
}

if (FAILED(PostDecode())) {
    av_frame_unref(m_pFrame);
    return E_FAIL;
}

// Decoding of this frame failed ... oh well!
if (used_bytes < 0) {
    av_frame_unref(m_pFrame);
    return S_OK;
}

// When Frame Threading, we won't know how much data has been consumed, so it by default eats everything.
// In addition, if no data got consumed, and no picture was extracted, the frame probably isn't all that useful.
// The MJPEB decoder is somewhat buggy and doesn't let us know how much data was consumed really...
if (!m_pParser && (m_pAVCtx->active_thread_type & FF_THREAD_FRAME || (!got_picture && used_bytes == 0)))
|| m_bNoBufferConsumption || bFlush) {
    buflen = 0;
} else {
    buflen -= used_bytes;
    pDataBuffer += used_bytes;
}

// Judge frame usability

```

## 《FFmpeg 基础库编程开发》

```
// This determines if a frame is artifact free and can be delivered
// For H264 this does some wicked magic hidden away in the H264RandomAccess class
// MPEG-2 and VC-1 just wait for a keyframe..
if (m_nCodecId == AV_CODEC_ID_H264 && (bParserFrame || !m_pParser || got_picture)) {
    m_h264RandomAccess.judgeFrameUsability(m_pFrame, &got_picture);
} else if (m_bResumeAtKeyFrame) {
    if (m_bWaitingForKeyFrame && got_picture) {
        if (m_pFrame->key_frame) {
            DbgLog((LOG_TRACE, 50, L"::Decode() - Found Key-Frame, resuming decoding at %I64d",
m_pFrame->pkt_pts));
            m_bWaitingForKeyFrame = FALSE;
        } else {
            got_picture = 0;
        }
    }
}

// Handle B-frame delay for frame threading codecs
if ((m_pAVCtx->active_thread_type & FF_THREAD_FRAME) && m_bBFrameDelay) {
    m_tcBFrameDelay[m_nBFramePos] = m_tcThreadBuffer[m_CurrentThread];
    m_nBFramePos = !m_nBFramePos;
}

if (!got_picture || !m_pFrame->data[0]) {
    if (!avpkt.size)
        bFlush = FALSE; // End flushing, no more frames
    av_frame_unref(m_pFrame);
    continue;
}

///////////////////////////////
// Determine the proper timestamps for the frame, based on different possible flags.
/////////////////////////////
if (m_bFFReordering) {
    rtStart = m_pFrame->pkt_pts;
    if (m_pFrame->pkt_duration)
        rtStop = m_pFrame->pkt_pts + m_pFrame->pkt_duration;
    else
        rtStop = AV_NOPTS_VALUE;
} else if (m_bBFrameDelay && m_pAVCtx->has_b_frames) {
    rtStart = m_tcBFrameDelay[m_nBFramePos].rtStart;
    rtStop = m_tcBFrameDelay[m_nBFramePos].rtStop;
} else if (m_pAVCtx->active_thread_type & FF_THREAD_FRAME) {
```

## 《FFmpeg 基础库编程开发》

```
unsigned index = m_CurrentThread;
rtStart = m_tcThreadBuffer[index].rtStart;
rtStop = m_tcThreadBuffer[index].rtStop;
}

if (m_bRVDropBFrameTimings && m_pFrame->pict_type == AV_PICTURE_TYPE_B) {
    rtStart = AV_NOPTS_VALUE;
}

if (m_bCalculateStopTime)
    rtStop = AV_NOPTS_VALUE;

///////////////////////////////
// All required values collected, deliver the frame
/////////////////////////////
LAVFrame *pOutFrame = NULL;
AllocateFrame(&pOutFrame);

AVRational display_aspect_ratio;
int64_t num = (int64_t)m_pFrame->sample_aspect_ratio.num * m_pFrame->width;
int64_t den = (int64_t)m_pFrame->sample_aspect_ratio.den * m_pFrame->height;
av_reduce(&display_aspect_ratio.num, &display_aspect_ratio.den, num, den, 1 << 30);

pOutFrame->width      = m_pFrame->width;
pOutFrame->height     = m_pFrame->height;
pOutFrame->aspect_ratio = display_aspect_ratio;
pOutFrame->repeat      = m_pFrame->repeat_pict;
pOutFrame->key_frame   = m_pFrame->key_frame;
pOutFrame->frame_type  = av_get_picture_type_char(m_pFrame->pict_type);
pOutFrame->ext_format  = GetDXVA2ExtendedFlags(m_pAVCtx, m_pFrame);

if (m_pFrame->interlaced_frame || (!m_pAVCtx->progressive_sequence && (m_nCodecId == AV_CODEC_ID_H264
|| m_nCodecId == AV_CODEC_ID_MPEG2VIDEO)))
    m_iInterlaced = 1;
else if (m_pAVCtx->progressive_sequence)
    m_iInterlaced = 0;

pOutFrame->interlaced      = (m_pFrame->interlaced_frame || (m_iInterlaced == 1 &&
m_pSettings->GetDeinterlacingMode() == DeintMode_Aggressive) || m_pSettings->GetDeinterlacingMode() == DeintMode_Force) && !(m_pSettings->GetDeinterlacingMode() == DeintMode_Disable);

LAVDeintFieldOrder fo      = m_pSettings->GetDeintFieldOrder();
pOutFrame->tff             = (fo == DeintFieldOrder_Auto) ? m_pFrame->top_field_first : (fo ==
```

```

DeintFieldOrder_TopFieldFirst);

pOutFrame->rtStart      = rtStart;
pOutFrame->rtStop       = rtStop;

PixelFormatMapping map  = getPixFmtMapping((AVPixelFormat)m_pFrame->format);
pOutFrame->format      = map.lavpixfmt;
pOutFrame->bpp         = map.bpp;

if (m_nCodecId == AV_CODEC_ID_MPEG2VIDEO || m_nCodecId == AV_CODEC_ID_MPEG1VIDEO)
    pOutFrame->avgFrameDuration = GetFrameDuration();

if (map.conversion) {
    ConvertPixFmt(m_pFrame, pOutFrame);
} else {
    for (int i = 0; i < 4; i++) {
        pOutFrame->data[i]   = m_pFrame->data[i];
        pOutFrame->stride[i] = m_pFrame->linesize[i];
    }

    pOutFrame->priv_data = av_frame_alloc();
    av_frame_ref((AVFrame *)pOutFrame->priv_data, m_pFrame);
    pOutFrame->destruct = lav_avframe_free;
}

if (bEndOfSequence)
    pOutFrame->flags |= LAV_FRAME_FLAG_END_OF_SEQUENCE;

if (pOutFrame->format == LAVPixFmt_DXVA2) {
    pOutFrame->data[0] = m_pFrame->data[4];
    HandleDXVA2Frame(pOutFrame);
} else {
    Deliver(pOutFrame);
}

if (bEndOfSequence) {
    bEndOfSequence = FALSE;
    if (pOutFrame->format == LAVPixFmt_DXVA2) {
        HandleDXVA2Frame(m_pCallback->GetFlushFrame());
    } else {
        Deliver(m_pCallback->GetFlushFrame());
    }
}

```

```

if (bFlush) {
    m_CurrentThread = (m_CurrentThread + 1) % m_pAVCtx->thread_count;
}
av_frame_unref(m_pFrame);
}

return S_OK;
}

终于，我们从这个函数中看到了很多的 ffmpeg 的 API，结构体，以及变量。比如解码视频的函数
avcodec_decode_video2()。

解码器初始化函数：InitDecoder()
//创建解码器
STDMETHODIMP CDecAvcodec::InitDecoder(AVCodecID codec, const CMediaType *pmt)
{
    //要是有，先销毁
    DestroyDecoder();
    DbgLog((LOG_TRACE, 10, L"Initializing ffmpeg for codec %S", avcodec_get_name(codec)));

    BITMAPINFOHEADER *pBMI = NULL;
    videoFormatTypeHandler((const BYTE *)pmt->Format(), pmt->FormatType(), &pBMI);
    //查找解码器
    m_pAVCodec = avcodec_find_decoder(codec);
    CheckPointer(m_pAVCodec, VFW_E_UNSUPPORTED_VIDEO);
    //初始化上下文环境
    m_pAVCtx = avcodec_alloc_context3(m_pAVCodec);
    CheckPointer(m_pAVCtx, E_POINTER);

    if(codec == AV_CODEC_ID_MPEG1VIDEO || codec == AV_CODEC_ID_MPEG2VIDEO || pmt->subtype ==
FOURCCMap(MKTAG('H','2','6','4')) || pmt->subtype == FOURCCMap(MKTAG('h','2','6','4'))) {
        m_pParser = av_parser_init(codec);
    }
}

DWORD dwDecFlags = m_pCallback->GetDecodeFlags();

LONG biRealWidth = pBMI->biWidth, biRealHeight = pBMI->biHeight;
if (pmt->formattype == FORMAT_VideoInfo || pmt->formattype == FORMAT_MPEGVideo) {
    VIDEOINFOHEADER *vih = (VIDEOINFOHEADER *)pmt->Format();
    if (vih->rcTarget.right != 0 && vih->rcTarget.bottom != 0) {
        biRealWidth = vih->rcTarget.right;
        biRealHeight = vih->rcTarget.bottom;
    }
} else if (pmt->formattype == FORMAT_VideoInfo2 || pmt->formattype == FORMAT_MPEG2Video) {

```

## 《FFmpeg 基础库编程开发》

```
VIDEOINFOHEADER2 *vih2 = (VIDEOINFOHEADER2 *)pmt->Format();
if (vih2->rcTarget.right != 0 && vih2->rcTarget.bottom != 0) {
    biRealWidth  = vih2->rcTarget.right;
    biRealHeight = vih2->rcTarget.bottom;
}
//各种赋值
m_pAVCtx->codec_id          = codec;
m_pAVCtx->codec_tag          = pBMI->biCompression;
m_pAVCtx->coded_width        = pBMI->biWidth;
m_pAVCtx->coded_height       = abs(pBMI->biHeight);
m_pAVCtx->bits_per_coded_sample = pBMI->biBitCount;
m_pAVCtx->error_concealment   = FF_EC_GUESS_MVS | FF_EC_DEBLOCK;
m_pAVCtx->err_recognition     = AV_EF_CAREFUL;
m_pAVCtx->workaround_bugs      = FF_BUG_AUTODETECT;
m_pAVCtx->refcounted_frames    = 1;

if (codec == AV_CODEC_ID_H264)
    m_pAVCtx->flags2 |= CODEC_FLAG2_SHOW_ALL;

// Setup threading
int thread_type = getThreadFlags(codec);
if (thread_type) {
    // Thread Count. 0 = auto detect
    int thread_count = m_pSettings->GetNumThreads();
    if (thread_count == 0) {
        thread_count = av_cpu_count() * 3 / 2;
    }
    m_pAVCtx->thread_count = max(1, min(thread_count, AVCODEC_MAX_THREADS));
    m_pAVCtx->thread_type = thread_type;
} else {
    m_pAVCtx->thread_count = 1;
}

if (dwDecFlags & LAV_VIDEO_DEC_FLAG_NO_MT) {
    m_pAVCtx->thread_count = 1;
}
//初始化 AVFrame
m_pFrame = av_frame_alloc();
CheckPointer(m_pFrame, E_POINTER);

m_h264RandomAccess.SetAVCNALSize(0);
```

```

// Process Extradata
//处理 ExtraData
BYTE *extra = NULL;
size_t extralen = 0;
getExtraData(*pmt, NULL, &extralen);

BOOL bH264avc = FALSE;
if (extralen > 0) {
    DbgLog((LOG_TRACE, 10, L"-> Processing extradata of %d bytes", extralen));
    // Reconstruct AVC1 extradata format
    if (pmt->formattype == FORMAT_MPEG2Video && (m_pAVCtx->codec_tag == MAKEFOURCC('a','v','c','1') ||
m_pAVCtx->codec_tag == MAKEFOURCC('A','V','C','1') || m_pAVCtx->codec_tag == MAKEFOURCC('C','C','V','1'))) {
        MPEG2VIDEOINFO *mp2vi = (MPEG2VIDEOINFO *)pmt->Format();
        extralen += 7;
        extra = (uint8_t *)av_mallocz(extralen + FF_INPUT_BUFFER_PADDING_SIZE);
        extra[0] = 1;
        extra[1] = (BYTE)mp2vi->dwProfile;
        extra[2] = 0;
        extra[3] = (BYTE)mp2vi->dwLevel;
        extra[4] = (BYTE)(mp2vi->dwFlags ? mp2vi->dwFlags : 4) - 1;

        // Actually copy the metadata into our new buffer
        size_t actual_len;
        getExtraData(*pmt, extra+6, &actual_len);

        // Count the number of SPS/PPS in them and set the length
        // We'll put them all into one block and add a second block with 0 elements afterwards
        // The parsing logic does not care what type they are, it just expects 2 blocks.
        BYTE *p = extra+6, *end = extra+6+actual_len;
        BOOL bSPS = FALSE, bPPS = FALSE;
        int count = 0;
        while (p+1 < end) {
            unsigned len = (((unsigned)p[0] << 8) | p[1]) + 2;
            if (p + len > end) {
                break;
            }
            if ((p[2] & 0x1F) == 7)
                bSPS = TRUE;
            if ((p[2] & 0x1F) == 8)
                bPPS = TRUE;
            count++;
            p += len;
        }
    }
}

```

```

}

extra[5] = count;
extra[extralen-1] = 0;

bH264avc = TRUE;
m_h264RandomAccess.SetAVCNALSize(mp2vi->dwFlags);
} else if (pmt->subtype == MEDIASUBTYPE_LAV_RAWVIDEO) {
if (extralen < sizeof(m_pAVCtx->pix_fmt)) {
    DbgLog((LOG_TRACE, 10, L"-> LAV RAW Video extradata is missing.."));
} else {
    extra = (uint8_t *)av_mallocz(extralen + FF_INPUT_BUFFER_PADDING_SIZE);
    getExtraData(*pmt, extra, NULL);
    m_pAVCtx->pix_fmt = *(AVPixelFormat *)extra;
    extralen -= sizeof(AVPixelFormat);
    memmove(extra, extra+sizeof(AVPixelFormat), extralen);
}
} else {
// Just copy extradata for other formats
extra = (uint8_t *)av_mallocz(extralen + FF_INPUT_BUFFER_PADDING_SIZE);
getExtraData(*pmt, extra, NULL);
}

// Hack to discard invalid MP4 metadata with AnnexB style video
if (codec == AV_CODEC_ID_H264 && !bH264avc && extra[0] == 1) {
    av_freep(&extra);
    extralen = 0;
}
m_pAVCtx->extradata = extra;
m_pAVCtx->extradata_size = (int)extralen;
} else {
if (codec == AV_CODEC_ID_VP6 || codec == AV_CODEC_ID_VP6A || codec == AV_CODEC_ID_VP6F) {
    int cropH = pBMI->biWidth - biRealWidth;
    int cropV = pBMI->biHeight - biRealHeight;
    if (cropH >= 0 && cropH <= 0x0f && cropV >= 0 && cropV <= 0x0f) {
        m_pAVCtx->extradata = (uint8_t *)av_mallocz(1 + FF_INPUT_BUFFER_PADDING_SIZE);
        m_pAVCtx->extradata_size = 1;
        m_pAVCtx->extradata[0] = (cropH << 4) | cropV;
    }
}
}

m_h264RandomAccess.flush(m_pAVCtx->thread_count);
m_CurrentThread = 0;
m_rtStartCache = AV_NOPTS_VALUE;

```

```

LAVPinInfo lavPinInfo = {0};
BOOL bLAVInfoValid = SUCCEEDED(m_pCallback->GetLAVPinInfo(lavPinInfo));

m_bInputPadded = dwDecFlags & LAV_VIDEO_DEC_FLAG_LAVSPLITTER;

// Setup codec-specific timing logic
BOOL bVC1IsPTS = (codec == AV_CODEC_ID_VC1 && !(dwDecFlags & LAV_VIDEO_DEC_FLAG_VC1_DTS));

// Use ffmpeg's logic to reorder timestamps
// This is required for H264 content (except AVI), and generally all codecs that use frame threading
// VC-1 is also a special case. Its required for splitters that deliver PTS timestamps (see bVC1IsPTS above)
m_bFFReordering = (codec == AV_CODEC_ID_H264 && !(dwDecFlags &
LAV_VIDEO_DEC_FLAG_H264_AVI))
    || codec == AV_CODEC_ID_VP8
    || codec == AV_CODEC_ID_VP3
    || codec == AV_CODEC_ID_THEORA
    || codec == AV_CODEC_ID_HUFFYUV
    || codec == AV_CODEC_ID_FFVHUFF
    || codec == AV_CODEC_ID_MPEG2VIDEO
    || codec == AV_CODEC_ID_MPEG1VIDEO
    || codec == AV_CODEC_ID_DIRAC
    || codec == AV_CODEC_ID_UTVIDEO
    || codec == AV_CODEC_ID_DNXHD
    || codec == AV_CODEC_ID_JPEG2000
    || (codec == AV_CODEC_ID_MPEG4 && pmt->formattype == FORMAT_MPEG2Video)
    || bVC1IsPTS;

// Stop time is unreliable, drop it and calculate it
m_bCalculateStopTime = (codec == AV_CODEC_ID_H264 || codec == AV_CODEC_ID_DIRAC || (codec == AV_CODEC_ID_MPEG4 && pmt->formattype == FORMAT_MPEG2Video) || bVC1IsPTS);

// Real Video content has some odd timestamps
// LAV Splitter does them alright with RV30/RV40, everything else screws them up
m_bRVDropBFrameTimings = (codec == AV_CODEC_ID_RV10 || codec == AV_CODEC_ID_RV20 || ((codec == AV_CODEC_ID_RV30 || codec == AV_CODEC_ID_RV40) && !(dwDecFlags & LAV_VIDEO_DEC_FLAG_LAVSPLITTER) || (bLAVInfoValid && (lavPinInfo.flags & LAV_STREAM_FLAG_RV34_MKV))));

// Enable B-Frame delay handling
m_bBFrameDelay = !m_bFFReordering && !m_bRVDropBFrameTimings;

```

## 《FFmpeg 基础库编程开发》

```
m_bWaitingForKeyFrame = TRUE;
m_bResumeAtKeyFrame =      codec == AV_CODEC_ID_MPEG2VIDEO
                           || codec == AV_CODEC_ID_VC1
                           || codec == AV_CODEC_ID_RV30
                           || codec == AV_CODEC_ID_RV40
                           || codec == AV_CODEC_ID_VP3
                           || codec == AV_CODEC_ID_THEORA
                           || codec == AV_CODEC_ID_MPEG4;

m_bNoBufferConsumption =    codec == AV_CODEC_ID_MJPEGB
                           || codec == AV_CODEC_ID_LOCO
                           || codec == AV_CODEC_ID_JPEG2000;

m_bHasPalette = m_pAVCtx->bits_per_coded_sample <= 8 && m_pAVCtx->extradata_size && !(dwDecFlags &
LAV_VIDEO_DEC_FLAG_LAVSPLITTER)
&& (codec == AV_CODEC_ID_MSVIDEO1
     || codec == AV_CODEC_ID_MSRLE
     || codec == AV_CODEC_ID_CINEPAK
     || codec == AV_CODEC_ID_8BPS
     || codec == AV_CODEC_ID_QPEG
     || codec == AV_CODEC_ID_QTRLE
     || codec == AV_CODEC_ID_TSCC);

if (FAILED(AdditionaDecoderInit())) {
    return E_FAIL;
}

if (bLAVInfoValid) {
    // Setting has_b_frames to a proper value will ensure smoother decoding of H264
    if (lavPinInfo.has_b_frames >= 0) {
        DbgLog((LOG_TRACE, 10, L"-> Setting has_b_frames to %d", lavPinInfo.has_b_frames));
        m_pAVCtx->has_b_frames = lavPinInfo.has_b_frames;
    }
}

// Open the decoder
// 打开解码器
int ret = avcodec_open2(m_pAVCtx, m_pAVCodec, NULL);
if (ret >= 0) {
    DbgLog((LOG_TRACE, 10, L"-> ffmpeg codec opened successfully (ret: %d)", ret));
    m_nCodecId = codec;
} else {
    DbgLog((LOG_TRACE, 10, L"-> ffmpeg codec failed to open (ret: %d)", ret));
```

```

DestroyDecoder();
return VFW_E_UNSUPPORTED_VIDEO;
}

m_iInterlaced = 0;
for (int i = 0; i < countof(ff_interlace_capable); i++) {
    if (codec == ff_interlace_capable[i]) {
        m_iInterlaced = -1;
        break;
    }
}

// Detect chroma and interlaced
if (m_pAVCtx->extradata && m_pAVCtx->extradata_size) {
    if (codec == AV_CODEC_ID_MPEG2VIDEO) {
        CMPEG2HeaderParser mpeg2Parser(extra, extralen);
        if (mpeg2Parser.hdr.valid) {
            if (mpeg2Parser.hdr.chroma < 2) {
                m_pAVCtx->pix_fmt = AV_PIX_FMT_YUV420P;
            } else if (mpeg2Parser.hdr.chroma == 2) {
                m_pAVCtx->pix_fmt = AV_PIX_FMT_YUV422P;
            }
            m_iInterlaced = mpeg2Parser.hdr.interlaced;
        }
    } else if (codec == AV_CODEC_ID_H264) {
        CH264SequenceParser h264parser;
        if (bH264avc)
            h264parser.ParseNALs(extra+6, extralen-6, 2);
        else
            h264parser.ParseNALs(extra, extralen, 0);
        if (h264parser.sps.valid)
            m_iInterlaced = h264parser.sps.interlaced;
    } else if (codec == AV_CODEC_ID_VC1) {
        CVC1HeaderParser vc1parser(extra, extralen);
        if (vc1parser.hdr.valid)
            m_iInterlaced = (vc1parser.hdr.interlaced ? -1 : 0);
    }
}

if (codec == AV_CODEC_ID_DNXHD)
    m_pAVCtx->pix_fmt = AV_PIX_FMT_YUV422P10;
else if (codec == AV_CODEC_ID_FRAPS)
    m_pAVCtx->pix_fmt = AV_PIX_FMT_BGR24;

```

```
if (bLAVInfoValid && codec != AV_CODEC_ID_FRAPS && m_pAVCtx->pix_fmt != AV_PIX_FMT_DXVA2_VLD)
    m_pAVCtx->pix_fmt = lavPinInfo.pix_fmt;

DbgLog((LOG_TRACE, 10, L"AVCodec init successfull. interlaced: %d", m_iInterlaced));

return S_OK;
}

解码器销毁函数: DestroyDecoder()
[cpp] view plaincopy
//销毁解码器，各种 Free
STDMETHODIMP CDecAvcodec::DestroyDecoder()
{
    DbgLog((LOG_TRACE, 10, L"Shutting down ffmpeg..."));
    m_pAVCodec     = NULL;

    if (m_pParser) {
        av_parser_close(m_pParser);
        m_pParser = NULL;
    }

    if (m_pAVCtx) {
        avcodec_close(m_pAVCtx);
        av_freep(&m_pAVCtx->extradata);
        av_freep(&m_pAVCtx);
    }
    av_frame_free(&m_pFrame);

    av_freep(&m_pFFBuffer);
    m_nFFBufferSize = 0;

    av_freep(&m_pFFBuffer2);
    m_nFFBufferSize2 = 0;

    if (m_pSwsContext) {
        sws_freeContext(m_pSwsContext);
        m_pSwsContext = NULL;
    }

    m_nCodecId = AV_CODEC_ID_NONE;

    return S_OK;
}
```

## 9.3 MPlayer

### 9.3.1 Mplayer 支持的格式

MPlayer 是一个 LINUX 下的视频播放器，它支持相当多的媒体格式，无论在音频播放还是在视频播放方面，可以说它支持的格式是相当全面的。

视频格式支持：MPEG、AVI、ASF 与 WMV、QuickTime 与 OGG/OGM、SDP、PVA、GIF。

音频格式支持：MP3、WAV、OGG/OGM 文件(Vorbis)、WMA 与 ASF、MP4、CD 音频、XMMS。

### 9.3.2 Mplayer 中头文件的功能分析

config.h // 各种本地配置宏定义头  
version.h // 版本定义头 #define VERSION "1.0pre7try2-3.4.2"  
mp\_msg.h // 消息处理头  
help\_mp.h // 根据配置自动生成的帮助头 #include "help/help\_mpen.h"  
cfg-mplayer-def.h // Mplayer 运行时的选项缺省值头文件 char\*  
default\_config =  
sub\_reader.h // 拥有格式自动发现功能的字幕(subtitle)阅读器  
libvo/video\_out.h // 该文件包含 libvo 视频输出的公共函数、变量  
libvo/font\_load.h // 有关字体装载的例程  
libao2/audio\_out.h // 音频输出驱动程序相关结构定义和全局数据  
libmpcodecs/dec\_audio.h // 音频解码  
libmpcodecs/dec\_video.h // 视频解码  
libmpdemux/matroska.h // 多路解复用，媒体容器格式 matroska 处理头  
libmpdemux/stream.h // 流处理  
libmpdemux/demuxer.h // 多路解复用头文件  
libmpdemux/stheader.h // 媒体流头处理  
get\_path.c // 路径获取头文件  
spudec.h // SPU 子画面单元头，DVD 字幕流  
edl.h // 剪辑控制清单  
m\_option.h // 选项类型处理头  
m\_config.h // 配置处理头文件

### 9.3.3 MPlayer.main 主流程简要说明

```
int main() {  
    1) 变量声明，电影信息 movie info:  
    2) 初始化，消息系统.....  
    play_next_file:  
    3)播放文件 filename 的循环 goto play_next_file 开始
```

main:

- 4) 主处理 main
- 5) 播放真正主循环 2010 ~3541 while (!eof)
  - while (!eof) {
    - 5.1) 播放音频 PLAY AUDIO 2017 ~ 2064 decode\_audio(sh\_audio, ...);
    - 5.2) 播放视频 PLAY VIDEO, 2068 ~ 2300 decode\_video(sh\_video, ...);
    - 5.3) 处理暂停 PAUSE
    - 5.4) 处理 EDL
    - 5.5) 键盘事件处理, 搜索 2400~3216 while (!brk\_cmd && (cmd=mp\_input\_get\_cmd(0,0,0))!=NULL)
    - 5.6) 时间寻道(秒) if (seek\_to\_sec)
    - 5.7) 寻道 3243 ~ 3306, if (rel\_seek\_secs || abs\_seek\_pos)
    - 5.8) 处理 GUI
    - 5.9) 变更 Update OSD
    - 5.10) 找到字幕 find sub
    - 5.11) 处理 X11 窗口
    - 5.12) DVD 字幕 sub:
- 6) 播放结束, 转到下个文件 goto\_next\_file:
  - }

### 9.3.4 Mplayer 源码分析

从 Mplayer.c 的 main 开始处理参数

```
mconfig = m_config_new();
m_config_register_options(mconfig, mplayer_opts);
// TODO : add something to let modules register their options
mp_input_register_options(mconfig);
parse_cfgfiles(mconfig);

初始化 mpctx 结构体, mpctx 应该是 mplayer context 的意思, 顾名思义是一个统筹全局的变量。
[cpp] view plaincopy
static MPCContext *mpctx = &mpctx_s;
// Not all functions in mplayer.c take the context as an argument yet
static MPCContext mpctx_s = {
.osd_function = OSD_PLAY,
.begin_skip = MP_NOPTS_VALUE,
.play_tree_step = 1,
.global_sub_pos = -1,
.set_of_sub_pos = -1,
.file_format = DEMUXER_TYPE_UNKNOWN,
.loop_times = -1,
```

```

#ifndef HAS_DVBIN_SUPPORT
.last_dvb_step = 1,
#endif
};

原型
//真正统筹全局的结构
typedef struct MPCContext {
    int osd_show_percentage;
    int osd_function;
    const ao_functions_t *audio_out;
    play_tree_t *playtree;
    play_tree_iter_t *playtree_iter;
    int eof;
    int play_tree_step;
    int loop_times;

    stream_t *stream;
    demuxer_t *demuxer;
    sh_audio_t *sh_audio;
    sh_video_t *sh_video;
    demux_stream_t *d_audio;
    demux_stream_t *d_video;
    demux_stream_t *d_sub;
    mixer_t mixer;
    const vo_functions_t *video_out;
    // Frames buffered in the vo ready to flip. Currently always 0 or 1.
    // This is really a vo variable but currently there's no suitable vo
    // struct.
    int num_buffered_frames;

    // used to retry decoding after startup/seeking to compensate for codec delay
    int startup_decode_retry;
    // how long until we need to display the "current" frame
    float time_frame;

    // AV sync: the next frame should be shown when the audio out has this
    // much (in seconds) buffered data left. Increased when more data is
    // written to the ao, decreased when moving to the next frame.
    // In the audio-only case used as a timer since the last seek
    // by the audio CPU usage meter.
    double delay;

float begin_skip; // start time of the current skip while on edlout mode

```

## 《FFmpeg 基础库编程开发》

```
// audio is muted if either EDL or user activates mute
short edl_muted; // Stores whether EDL is currently in muted mode.
short user_muted; // Stores whether user wanted muted mode.

int global_sub_size; // this encompasses all subtitle sources
int global_sub_pos; // this encompasses all subtitle sources
int set_of_sub_pos;
int set_of_sub_size;
int sub_counts[SUB_SOURCES];

#ifndef CONFIG_ASS
    // set_of_ass_tracks[i] contains subtitles from set_of_subtitles[i]
    // parsed by libass or NULL if format unsupported
    ASS_Track* set_of_ass_tracks[MAX_SUBTITLE_FILES];
#endif

sub_data* set_of_subtitles[MAX_SUBTITLE_FILES];

int file_format;

#ifndef CONFIG_DVBIN
    int last_dvb_step;
    int dvbin_reopen;
#endif

int was_paused;

#ifndef CONFIG_DVDNAV
    struct mp_image *nav_smpi; // last decoded dvdnav video image
    unsigned char *nav_buffer; // last read dvdnav video frame
    unsigned char *nav_start; // pointer to last read video buffer
    int nav_in_size; // last read size
#endif

} MPContext;
一些 GUI 相关的操作
打开字幕流
打开音视频流
mpctx->stream=open_stream(filename,0,&mpctx->file_format);
fileformat 文件还是 TV 流 DEMUXER_TYPE_PLAYLIST 或 DEMUXER_TYPE_UNKNOWN
DEMUXER_TYPE_TV
current_module 记录状态 vobsub open_stream handle_playlist dumpstream
stream_reset(mpctx->stream);
stream_seek(mpctx->stream,mpctx->stream->start_pos);
f=fopen(stream_dump_name,"wb"); dump 文件流
stream->type==STREAMTYPE_DVD
```

## 《FFmpeg 基础库编程开发》

```
//===== Open DEMUXERS — DETECT file type =====
Demux。分离视频流和音频流
mpctx->demuxer=demux_open(mpctx->stream,mpctx-
>file_format,audio_id,video_id,dvdsub_id,filename);
Demux 过程
demux_open
get_demuxer_type_from_name
.....
mpctx->d_audio=mpctx->demuxer->audio;
mpctx->d_video=mpctx->demuxer->video;
mpctx->d_sub=mpctx->demuxer->sub;
mpctx->sh_audio=mpctx->d_audio->sh;
mpctx->sh_video=mpctx->d_video->sh;
分离了之后就开始分别 Play audio 和 video
这里只关心 play video
/*===== PLAY VIDEO =====*/
vo_pts=mpctx->sh_video->timer*90000.0;
vo_fps=mpctx->sh_video->fps;
if (!mpctx->num_buffered_frames) {
    double frame_time = update_video(&blit_frame);
    mp_dbg(MSGT_AVSYNC,MSGL_DBG2,"*** ftime=%5.3f ***\n",frame_time);
    if (mpctx->sh_video->vf_initiated < 0) {
        mp_msg(MSGT_CPLAYER,MSGL_FATAL, MSGTR_NotInitializeVOPorVO);
        mpctx->eof = 1; goto goto_next_file;
    }
    if (frame_time < 0)
        mpctx->eof = 1;
    else {
        // might return with !eof && !blit_frame if !correct_pts
        mpctx->num_buffered_frames += blit_frame;
        time_frame += frame_time / playback_speed; // for nosound
    }
}
```

关键的函数是 update\_video 根据 pts 是否正确调整一下同步并在必要的时候丢帧处理。最终调用 decode\_video 开始解码（包括 generate\_video\_frame 里）。mpi = mpvdec->decode(sh\_video, start, in\_size, drop\_frame);mpvdec 是在 main 里通过 reinit\_video\_chain 的一系列调用动态选定的解码程序。其实就一结构体。它的原型是

```
typedef struct vd_functions_s
{
    vd_info_t *info;
    int (*init)(sh_video_t *sh);
    void (*uninit)(sh_video_t *sh);
    int (*control)(sh_video_t *sh,int cmd,void* arg, ...);
```

## 《FFmpeg 基础库编程开发》

```
mp_image_t* (*decode)(sh_video_t *sh,void* data,int len,int flags);
} vd_functions_t;
```

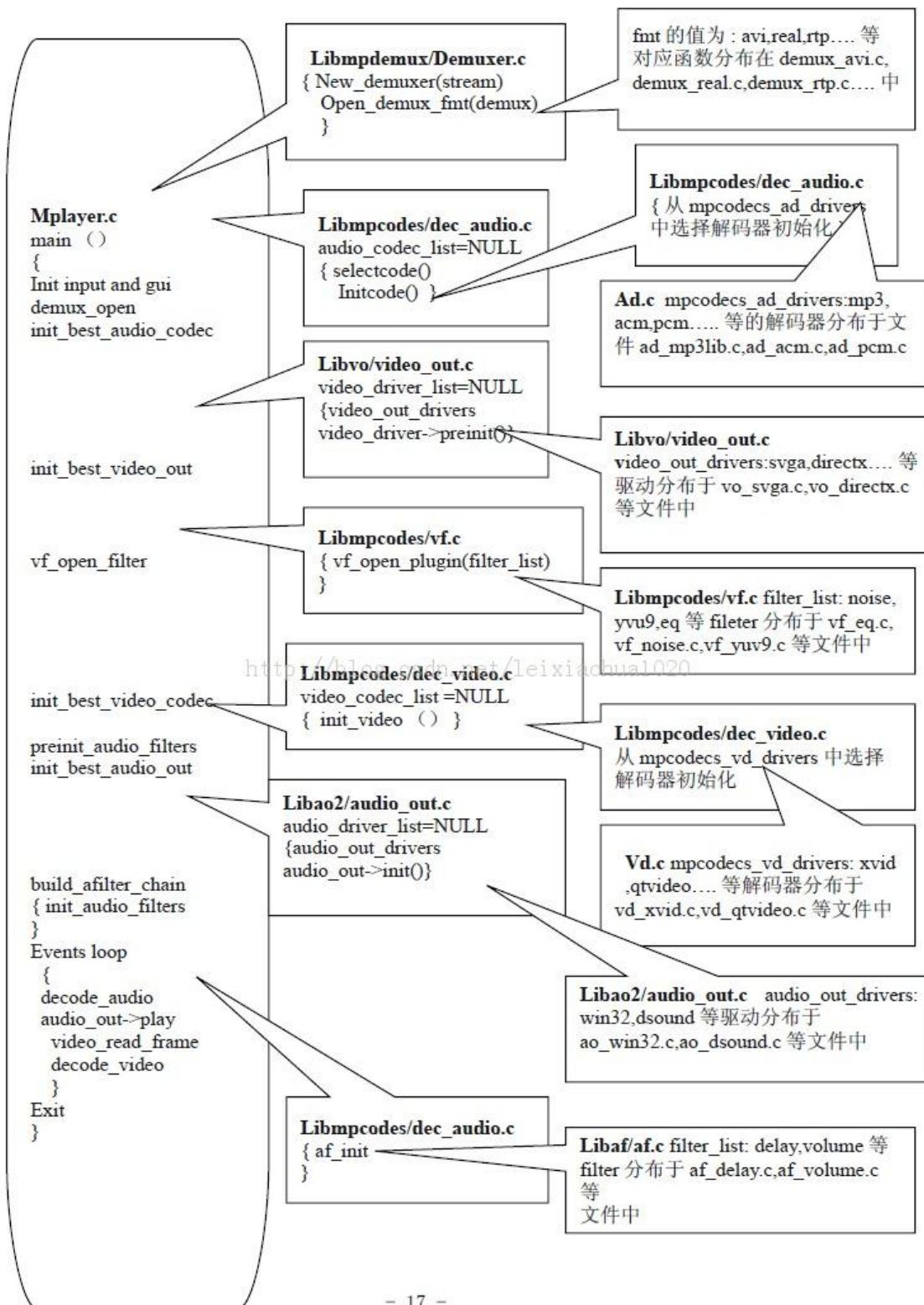
这是所有解码器必须实现的接口。

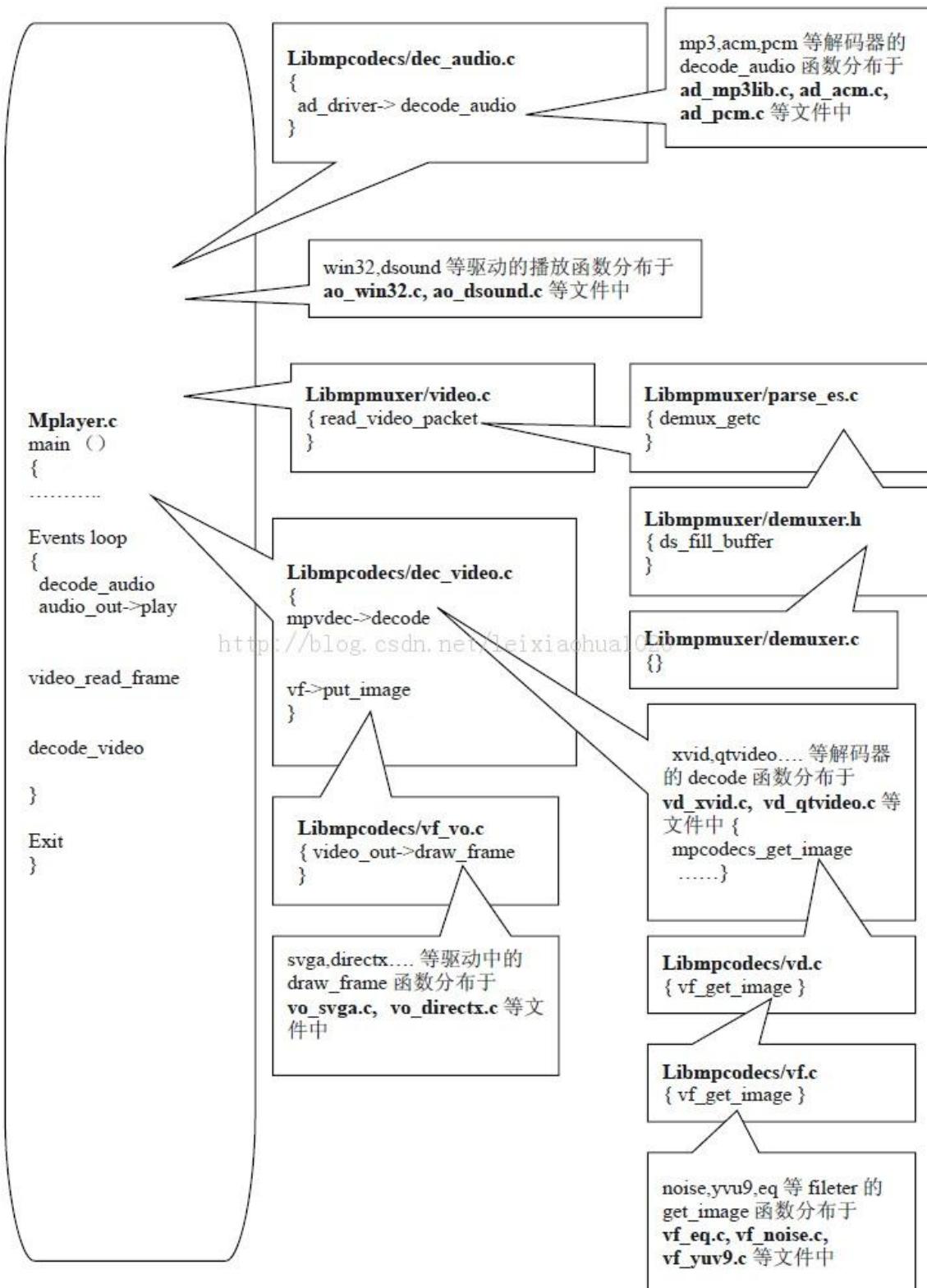
int (\*init)(sh\_video\_t \*sh);是一个名为 init 的指针，指向一个接受 sh\_video\_t \*类型参数，并返回 int 类型值的函数地址。那些 vd\_ 开头的文件都是解码相关的。随便打开一个 vd 文件以上几个函数和 info 变量肯定都包含了。mpi 被 mplayer 用来存储解码后的图像。在 mp\_image.h 里定义。

```
typedef struct mp_image_s {
    unsigned short flags;
    unsigned char type;
    unsigned char bpp; // bits/pixel. NOT depth! for RGB it will be n*8
    unsigned int imgfmt;
    int width,height; // stored dimensions
    int x,y,w,h; // visible dimensions
    unsigned char* planes[MP_MAX_PLANES];
    int stride[MP_MAX_PLANES];
    char *qscale;
    int qstride;
    int pict_type; // 0->unknown, 1->I, 2->P, 3->B
    int fields;
    int qscale_type; // 0->mpeg1/4/h263, 1->mpeg2
    int num_planes;
    /* these are only used by planar formats Y,U(Cb),V(Cr) */
    int chroma_width;
    int chroma_height;
    int chroma_x_shift; // horizontal
    int chroma_y_shift; // vertical
    /* for private use by filter or vo driver (to store buffer id or dmpi) */
    void* priv;
} mp_image_t;
```

图像在解码以后会输出到显示器，mplayer 本来就是一个视频播放器么。但也有可能作为输入提供给编码器进行二次编码，MP 附带的 mencoder.exe 就是专门用来编码的。在这之前可以定义 filter 对图像进行处理，以实现各种效果。所有以 vf\_ 开头的文件，都是这样的 filter。图像的显示是通过 vo，即 video out 来实现的。解码器只负责把解码完成的帧传给 vo，怎样显示就不用管了。这也是平台相关性最大的部分，单独分出来的好处是不言而喻的，像在 Windows 下有通过 direcx 实现的 vo，Linux 下有输出到 X 的 vo。vo\_\*文件是各种不同的 vo 实现，只是他们不都是以显示为目的，像 vo\_md5sum.c 只是计算一下图像的 md5 值。在解码完成以后，即得到 mpi 以后，filter\_video 被调用，其结果是整个 filter 链上的所有 filter 都被调用了一遍，包括最后的 VO，在 vo 的 put\_image 里把图像输出到显示器。这个时候需要考虑的是图像存储的方法即用哪种色彩空间。

附上两张 MPlayer 结构图：





MPlayer 源代码下载地址: <http://download.csdn.net/detail/leixiaohua1020/6374337>

# 第十章 开发实例

## 第十一章 mp4 文件封装协议分析

### 11.1 概述

MP4 文件格式中，所有的内容存在一个称为 movie 的容器中。一个 movie 可以由多个 trak 组成。每个 trak 就是一个随时间变化的媒体序列，例如，视频帧序列。trak 里的每个时间单位是一个 sample，它可以是一帧视频，或者音频。sample 按照时间顺序排列。注意，一帧音频可以分解成多个音频 sample，所以音频一般用 sample 作为单位，而不用帧。MP4 文件格式的定义里面，用 sample 这个单词表示一个时间帧或者数据单元。每个 trak 会有一个或者多个 sample descriptions。track 里面的每个 sample 通过引用关联到一个 sample description。这个 sample descriptions 定义了怎样解码这个 sample，例如使用的压缩算法。

与其他的多媒体文件格式不同的是，MP4 文件格式经常使用几个不同的概念，理解其不同是理解这个文件格式的关键。

这个文件的物理格式没有限定媒体本身格式。例如，许多文件格式将媒体数据分成帧，头部或者其他数据紧跟跟随每一帧视频。而 MP4 文件格式不是如此。

文件的物理格式和媒体数据的排列都不受媒体的时间顺序的限制。视频帧不需要在文件按时间顺序排列。这就意味着如果文件中真的存在这样的一些帧，那么就有一些文件结构来描述媒体的排列和对应的时间信息。

MP4 文件中所有的数据都封装在一些 box 中（以前叫 atom）。所有的 metadata(媒体描述元数据)，包括定义媒体的排列和时间信息的数据都包含在这样的一些结构 box 中。MP4 文件格式定义了这些这些 box 的格式。Metadata 对媒体数据（例如，视频帧）引用说明。媒体数据可以包含在同一个的一个或多个 box 里，也可以在其他文件中，metadata 允许使用 URLs 来引用其他的文件，而媒体数据在这些引用文件中的排列关系全部在第一个主文件中的 metadata 描述。其他的文件不一定是 MP4 文件格式，例如，可能就没有一个 box。

有很多种类的 trak，其中有三个最重要，video track 包含了视频 sample；audio trak 包含了 audio sample；hint trak 稍有不同，它描述了一个流媒体服务器如何把文件中的媒体数据组成符合流媒体协议的数据包。如果文件只是本地播放，可以忽略 hint track，他们只与流媒体有关系。

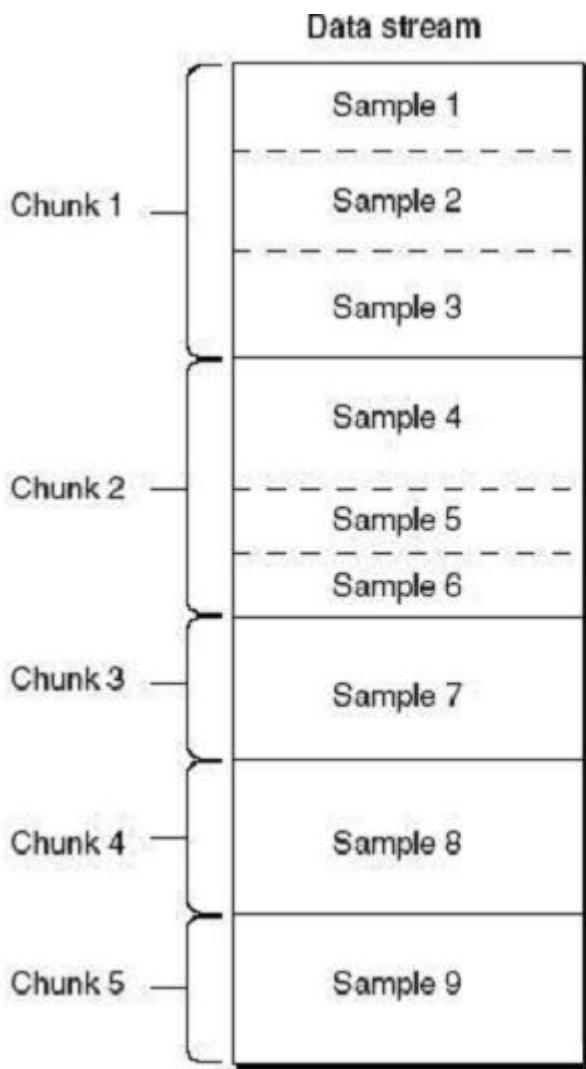
### 11.2 mp4 的物理结构

Box 定义了如何在 sample table 中找到媒体数据的排列。这包括 data reference(数据引用)，the sample size table，the sample to chunk table，and the chunk offset table. 这些表就可以找到 trak 中每个 sample 在文件中的位置和大小。为了节约空间，这些表都很紧凑。另外，interleave 不是 sample by sample，而是把单个 trak 的几个 samples 组合到一起，然后另外几个 sample 又进行新的组合。一个 trak 的连续几个 sample 组成的单元就被称为 chunk。每个 chunk 在文件中有一个偏移量，这个偏移量是从文件开头算起的，在这个 chunk 内，sample 是连续存储的。

这样，如果一个 chunk 包含两个 sample，第二个 sample 的位置就是 chunk 的偏移量加上第一个 sample 的大小。chunk offset table 说明了每个 chunk 的偏移量，sample to chunk table 说明了 sample 序号和 chunk 序号的映射关系。

注意 chunk 之间可能会有死区，没有任何媒体数据引用到这部分区域，但是 chunk 内部不会有这样的死区。

## 11.3 数据的组织结构



## 11.4 mp4 的时间结构

文件中的时间可以理解为一些结构。电影以及每个 trak 都有一个 timescale。它定义了一个时间轴来说明每秒钟有多少个 ticks。合理的选择这个数目，就可以实现准确的计时。一般来说，对于 audio track，就是 audio 的 sampling rate。对于 video track，情况稍微复杂，需要合理选择。例如，如果一个 media TimeScale 是 30000, media sample durations 是 1001，就准确的定义了 NTSC video 的时间格式（虽然不准确，但一般就是 29.97）。

每个 trak 的全部 duration 定义在文件头部，这就是对 track 的总结，每个 sample 有一个规定的 duration。一个 sample 的准确描述时间，也就是他的时间戳(time-stamp)就是以前的 sample 的 duration 之和。

关键词：

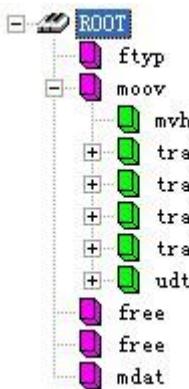
1. **trak** 表示一些sample的集合，对于媒体数据来说，track表示一个视频或音频序列。
2. **sample** video sample即为一帧视频，或一组连续视频帧，audio sample即为一段连续的压缩音频，它们统称

sample。

3. **chunk** 一个trak的几个sample组成的单元。

4. **box** box 由 header 和 body 组成，其中 header 统一指明 box 的大小和类型，body 根据类型有不同的意义和格式。标准的 box 开头的 4 个字节（32 位）为 box size，该大小包括 box header 和 box body 整个 box 的大小，这样我们就可以在文件中定位各个 box。size 后面紧跟的 32 位为 box type，一般是 4 个字符，如“ftyp”、“moov”等，这些 box type 都是已经预定义好的，分别表示固定的意义。

下图为一个典型的 MP4 文件的结构树：



## 11.5 文件结构分析

### 11.5.1 File Type Box (ftyp)

该 box 有且只有 1 个，并且只能被包含在文件层，而不能被其他 box 包含。该 box 应该被放在文件的最开始，指示该 MP4 文件应用的相关信息。

“ftyp” body 依次包括 1 个 32 位的 major brand (4 个字符)，1 个 32 位的 minor version (整数) 和 1 个以 32 位 (4 个字符) 为单位元素的数组 compatible brands。这些都是用来指示文件应用级别的信息。

该 box 的字节实例如下：

0000000000h:	00 00 00 18 66 74 79 70 6D 70 34 32 00 00 00 01 ; ...ftypmp42...
000000010h:	6D 70 34 32 6D 70 34 31 00 00 5A EB 6D 6F 6F 76 ; mp42mp41..2號moov

### 11.5.2 Movie Box (moov)

该 box 包含了文件媒体的 metadata 信息，“moov”是一个 container box，具体内容信息由子 box 诠释。同 File Type Box 一样，该 box 有且只有一个，且只被包含在文件层。一般情况下，“moov”会紧随“ftyp”出现。

一般情况下，“moov”中会包含 1 个“mvhd”和若干个“trak”。其中“mvhd”为 header box，一般作为“moov”的第一个子 box 出现。

#### 11.5.1.1 Movie Header Box (mvhd)

mvhd 定义了整个 movie 的特性，例如 time scale 和 duration，它的 atom 类型是'mvhd'。具体字段的表结构如下：

《FFmpeg 基础库编程开发》

字段	长度(单位: byte)	描述
box size	4	box size
box type	4	box 类型
version	1	box 版本, 0 或 1, 一般为 0。(以下字节数均按 version=0)
flags	3	
creation time	4	创建时间 (相对于 UTC 时间 1904-01-01 零点的秒数)
modification time	4	修改时间
time scale	4	时间缩放因子
duration	4	该视频的时长(整体标记)
rate	4	推荐播放速率, 高 16 位和低 16 位分别为小数点整数部分和小数部分, 即[16.16] 格式, 该值为 1.0 (0x00010000) 表示正常前向播放
volume	2	与 rate 类似, [8.8] 格式, 1.0 (0x0100) 表示最大音量
reserved	10	保留位
matrix	36	视频变换矩阵
pre-defined	24	
next track id	4	下一个 track 使用的 id 号

“mvhd”的字节实例如下图，各字段已经用颜色区分开：

000000020h:	00 00 00 60 6D 76 68 64 00 00 00 00 BE 44 3F 8D	; ... lmvh... 繼??
000000030h:	BE 44 3F 8D 00 00 02 58 00 00 A4 10 00 01 00 00	; 繼?? ..X..? ....
000000040h:	01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	; ..
000000050h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	; ..
000000060h:	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 00	; .. ..@..
000000070h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	; .. ..
000000080h:	00 00 00 00 00 00 00 00 00 00 00 05 00 00 11 75	; .. .. ..u.

### 11.5.1.2 Track Box (trak)

主数据存储结构，一部 movie 可以包含一个或多个 tracks，它们之间相互独立，各自有各自的时间和空间信息。每个 track atom 都有与之关联的 media atom。

trak atoms 的 atom 类型是'trak'. trak atom 要求必须有一个 trak header atom ('tkhd') 和一个 media atom ('mdia')。其他的 track clipping atom ('clip'), track matte atom ('matt'), edit atom ('edts'), track reference atom ('tref'), track load settings atom ('load'), a track input map atom ('imap')以及 user data atom ('udta')都是可选的。 具体表结构如下：

#### 1. Track Header Box (tkhd)

trak 的头信息，具体表结构如下：

字段	长度（单位： byte）	意义
box size	4	box大小
box type	4	box类型
version	1	box版本，0或1，一般为0。(以下字节数均按version=0)
flags	3	按位或操作结果值，预定义如下： 0x000001 track_enabled，否则该track不被播放； 0x000002 track_in_movie，表示该track在播放中被引用； 0x000004 track_in_preview，表示该track在预览时被引用。 一般该值为7，如果一个媒体所有track均未设置track_in_movie和track_in_preview，将被理解为所有track均设置了这两项。

《FFmpeg 基础库编程开发》

creation time	4	创建时间（相对于 UTC 时间 1904-01-01 零点的秒数）
modification time	4	修改时间
track id	4	id 号，不能重复且不能为 0
reserved	4	保留位
duration	4	trak 的时间长度
reserved	8	保留位

《FFmpeg 基础库编程开发》

layer	2	视频层， 默认为 0(跳过)
alternate group	2	trak 分组信息， 默认为 0 表示该 trak 未与其他 trak 有群组关系
volume	2	[8.8] 格式， 如果为音频 trak, 1.0 (0x0100) 表示最大音量； 否则为 0
reserved	2	保留位
matrix	36	视频变换矩阵
width	4	宽

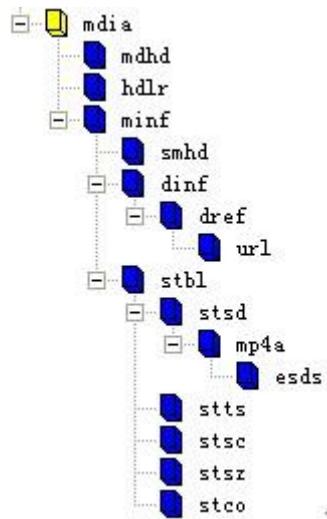
height	4	高
--------	---	---

“tkhd” 的字节实例如下图，各字段已经用颜色区分开：

```
000000090h: 74 72 61 6B 00 00 00 5C 74 6B 68 64 00 00 00 01 ; trak...\tkhd...
0000000a0h: BE 44 3F 8C BE 44 3F 8D 00 00 00 01 00 00 00 00 ; 錄?標D?? .....
0000000b0h: 00 00 A4 10 00 00 00 00 00 00 00 00 00 00 00 00 ; ...? .....
0000000c0h: 01 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 ; .....
0000000d0h: 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 ; .....
0000000e0h: 00 00 00 00 40 00 00 00 00 00 00 00 00 00 00 00 ; ....@.....
```

## 2. Media Box (mdia)

“mdia” 也是个 container box，其子 box 的结构和种类还是比较复杂的。树结构图如下：



总体来说，“mdia” 定义了 trak 媒体类型以及 sample 数据，描述 sample 信息。一般 “mdia” 包含一个 “mdhd”，一个 “hdrl” 和一个 “minf”，其中 “mdhd” 为 media header box，“hdrl” 为 handler reference box，“minf” 为 media information box。

### 1. Media Header Box (mdhd)

Media header atom 定义了媒体的特性，例如 time scale 和 duration。它的类型是'mdhd'。

具体表结构如下：

字段	长度(单位: byte)	意义
box size	4	box 大小
box type	4	box 类型
version	1	box 版本, 0 或 1, 一般为 0。(以下字节数均按 version=0)
flags	3	
creation time	4	创建时间 (相对于 UTC 时间 1904-01-01 零点的秒数)
modification time	4	修改时间
time scale	4	时间缩放因子
duration	4	track 的时间长度
language	2	媒体语言码。最高位为 0, 后面 15 位为 3 个字符 (见 ISO 639-2/T 标准中定义)
pre-defined	2	

## 2. Handler Reference Box (hdlr)

Handler reference atom 定义了描述此媒体数据的 media handler component, 类型是'hdlr'。在过去, handler reference atom 也可以用来数据引用, 现在废弃。一个 media atom 内的 handler atom 解释了媒体流的播放过程。例如, 一个视频 handler 处理一个 video track。具体表结构如下:

字段	长度(单位: byte)	描述
尺寸	4	这个 atom 的字节数
类型	4	hdlr
版本	1	这个 atom 的版本
标志	3	这里为 0
Component type	4	handler 的类型。当前只有两种类型: 'mhlr': media handlers 'dhlr': data handlers(废弃)

Component subtype	4	media handler or data handler 的类型。 如果 component type 是 mhlr, 这个字段定义了数据的类型, 可以用来判断该 trak 的类型, 例如, 'vide'是 video 数据, 'soun'是 sound 数据 如果 component type 是 dhlr, 这个字段定义了数据引用的类型(废弃)
Component manufacturer	4	保留字段, 缺省为 0
Component flags	4	保留字段, 缺省为 0
Component flags mask	4	保留字段, 缺省为 0
Component name	可变	这个 component 的名字, 也就是生成此 media 的 media handler。该字段的长度可以为 0

### 3. Media Information Atoms - MINF

“minf”存储了解释 trak 媒体数据的 handler-specific 信息, media handler 用这些信息将媒体时间映射到媒体数据并进行处理。“minf”中的信息格式和内容与媒体类型以及解释媒体数据的 media handler 密切相关, 其他 media handler 不知道如何解释这些信息。“minf”是一个 container box, 其实际内容由子 box 说明。

一般情况下, “minf”包含一个 header box, 一个“dinf”和一个“stbl”, 其中, header box 根据 track type (即 media handler type) 分为“vmhd”、“smhd”, “dinf”为 data information box, “stbl”为 sample table box。

### 3.1 Media Information Header Box (vmhd、smhd) (拆包时可直接跳过)

#### Video Media Header Box (vmhd)

字段	长度(单位: byte)	描述
box size	4	box 大小
box type	4	box 类型
version	1	box 版本, 0 或 1, 一般为 0。
flags	3	
graphics mode	4	跳过
opcolor	2×3	{red, green, blue}

#### Sound Media Header Box (smhd)

字段	长度(单位: byte)	描述
box size	4	box 大小
box type	4	box 类型
version	1	box 版本, 0 或 1, 一般为 0。(以下字节数均按 version=0)
flags	3	
balance	2	立体声平衡(跳过)
reserved	2	

### 3.2 Data Information Box (dinf)

“dinf”解释如何定位媒体信息，是一个 container box。“dinf”一般包含一个“dref”，即 data reference box；“dref”下会包含若干个“url”或“urn”，这些 box 组成一个表，用来定位 trak 数据。简单的说，trak 可以被分成若干段，每一段都可以根据“url”或“urn”指向的地址来获取数据，sample 描述中会用这些片段的序号将这些片段组成一个完整的 trak。一般情况下，当数据被完全包含在文件中时，“url”或“urn”中的定位字符串是空的。

“dref”的字节结构如下表：

字段	长度(单位: byte)	描述
box size	4	box 大小
box type	4	box 类型
version	1	box 版本, 0 或 1, 一般为 0。(以下字节数均按 version=0)
flags	3	
entry count	4	“url”或“urn”表的元素个数, 每个 data reference 就像 atom 的格式一样,

		包含以下的数据成员
“url” 或 “urn” 列表	不定	“url” 或 “urn” 列表

**entry** 的结构如下表：

box size	4	box 大小
box type	4	见下表
version	1	这个 data reference 的版本
flags	3	<p>目前只有一个标志：</p> <p>Self reference</p> <p>This flag indicates that the media's data is in the same file as the movie atom. On the Macintosh, and other file systems with multifork files, set this flag to 1 even if the data resides in a different fork from the movie atom. This flag's value is 0x0001.</p>
数据	可变	data reference 信息

**data reference** 具体结构如下：

类型	描述
----	----

alias Data reference 是一个 Macintosh alias。一个 alias 包含文件信息，例如全路径名。

rsrc Data reference 是一个 Macintosh alias。Alias 末尾是文件使用的资源类型（32bit 整数）和 ID（16bit 带符号的整数）

url 一个 C 类型的字符串，表示一个 URL。字符串后可以有其他的数据。

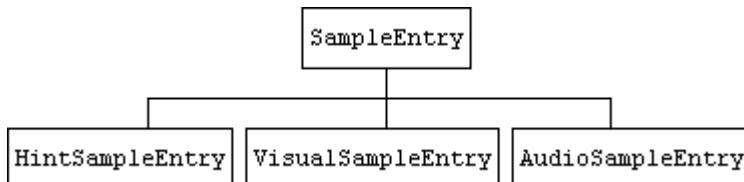
### 3.3 Sample Table Box (stbl) (重要)

“stbl”包含了关于 trak 中 sample 所有时间和位置的信息，以及 sample 的编解码等信息。利用这个表，可以解释 sample 的时序、类型、大小以及在各自存储容器中的位置。“stbl”是一个 container box，其子 box 包括：sample description box (stsd)、time to sample box (stts)、sample size box (stsz 或 stz2)、sample to chunk box (stsc)、chunk offset box (stco 或 co64)、composition time to sample box (ctts)、sync sample box (stss) 等。

#### 3.3.1 Sample Description Atoms - STSD

“stsd”必不可少，且至少包含一个条目，该 box 包含了 data reference box 进行 sample 数据检索的信息。没有“stsd”就无法计算 media sample 的存储位置。“stsd”包含了编码的信息，其存储的信息随媒体类型不同而不同。

在认识 stsd 之前我们首先需要了解一个数据结构 SampleEntry 和它的子类 AudioSampleEntry, VisualSampleEntry, HintSampleEntry(不作分析)，具体关系如下：



SampleEntry 是一个继承 box 的抽象的数据结构模型，具体如下表：

字段	长度 (单位: byte)	描述
box size	4	box 大小
box type	4	box 类型(根据该值查找视频格式 id 表获得编码器类型，如"avc1"通过查表标记为 H264_ID 类型)(重要)
resved	6	保留字段,(跳过)
drefid	2	无用(跳过)

VisualSampleEntry(类型为"avc1")继承于 SampleEntry，具体结构如下表：

字段	长度 (单位: byte)	描述
SampleEntry	16	SampleEntry

### 《FFmpeg 基础库编程开发》

resved	16	保留字段(跳过)
width	2	宽度
height	2	高度
hrs1	4	水平分辨率
vts1	4	垂直分辨率
reserved	4	一直为 0
frame_count	2	每个采样里面的贞数,一般是 1
compressorname	4	字符串, 对齐到 32 位, (无用跳过)
depth	2	视频的色深 0x18 表示 24 位色

AudioSampleEntry(类型为"mp4a")继承于 SampleEntry ,具体结构如下表:

字段	长度 (单位: byte)	描述
SampleEntry	16	SampleEntry
resved	16	保留字段(跳过)
channelcount	2	声道数 1 或者 2
samplesize	2	采样位宽 一般为 8bit 或 16bit
reserved	4	保留字段(跳过)
samplerate	4	采样率
esds 扩展(重要)		如果 audio type 为 AAC,需要读取 esds 扩展, 否则音频无法解码。
version + flags	4	version + flags
tag	1	决定的后续的解析
descr length	4	跳过
Id+priority	2 或 2+1	如果 tag='0x03'为 2 个字节(跳过)

《FFmpeg 基础库编程开发》

		其他值为 2+1 个字节(跳过)
tag	1	决定的后续的解析, 如果解析正确该值为"0x04"
descr length	4	跳过
audio type id	1	如果解析正确为'0x40', 为 CODEC_ID_AAC 类型
resved	1+3+4+4	跳过
tag	1	如果解析正确为'0x05'

descr length	4	Descr data 的长度
Descr data	n	0-3 位为采样率查表 index 4-7 位为声道的数目 具体其他信息关系到 sbr 的一些参数，具体请参看官方文档

其他的实体格式如 AMRSampleEntry AMRWPSampleEntry H263SampleEntry 等分析同上。

对于"stsd"的表结构如下：

字段	长度 (单位: byte)	描述
box size	4	box 大小
box type	4	该类型为"stsd"
version	1	box 版本, 0 或 1, 一般为 0
flags	3	
entry count	4	entry 的个数
entry	n	具体参考上表

### 3.3.2 Time-to-Sample Atoms - STTS

## 《FFmpeg 基础库编程开发》

Time-to-sample atoms 存储了 media sample 的 duration 信息，提供了时间对具体 data sample 的映射方法，通过这个 atom，你可以找到任何时间的 sample，类型是'stts'。

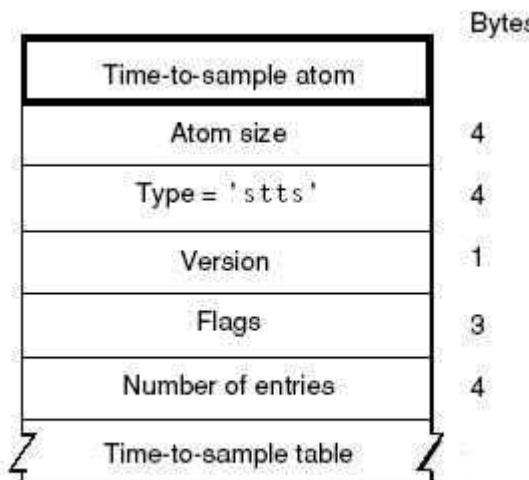
这个 atom 可以包含一个压缩的表来映射时间和 sample 序号，用其他的表来提供每个 sample 的长度和指针。表中每个条目提供了在同一个时间偏移量里面连续的 sample 序号，以及 samples 的偏移量。递增这些偏移量，就可以建立一个完整的 time-to-sample 表，计算公式如下

$$DT(n+1) = DT(n) + STTS(n)$$

其中 STTS(n) 是没有压缩的 STTS 第 n 项信息，DT 是第 n 个 sample 的显示时间。Sample 的排列是按照时间戳的顺序，这样偏移量永远是非负的。DT 一般以 0 开始，如果不为 0，edit list atom 设定初始的 DT 值。DT 计算公式如下

$$DT(i) = \text{SUM} (\text{for } j=0 \text{ to } i-1 \text{ of } \delta(j))$$

所有偏移量的和就是 trak 中 media 的时间的长度。



字段	长度(单位: byte)	描述
尺寸	4	这个 atom 的字节数
类型	4	stts
版本	1	这个 atom 的版本
标志	3	这里为 0
条目数目	4	time-to-sample 的数目
time-to-sample		Media 中每个 sample 的 duration。包含如下结构
Sample count	4	有相同 duration 的连续 sample 的数目
Sample duration	4	每个 sample 的 duration

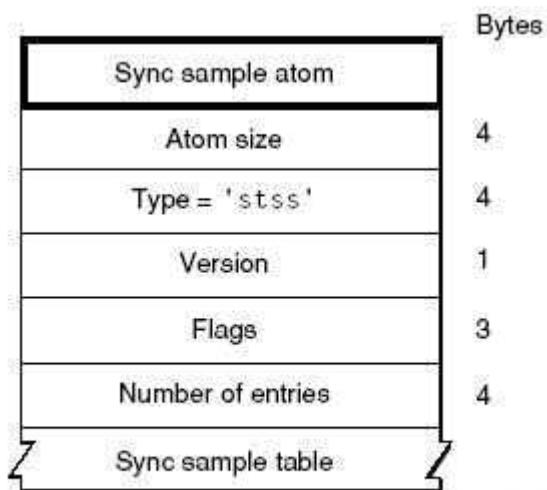
如果多个 sample 有相同的 duration，可以只用一项描述所有这些 samples，数量字段说明 sample 的个数。例如，如果一个视频媒体的帧率保持不变，整个表可以只有一项，数量就是全部的帧数。

### 3.3.3 Sync Sample Atoms - STSS

sync sample atom 确定 media 中的关键帧。对于压缩的媒体，关键帧是一系列压缩序列的开始帧，它的解压缩是不依赖于以前的帧。后续帧的解压缩依赖于这个关键帧。

sync sample atom 可以非常紧凑的标记媒体内的随机存取点。它包含一个 sample 序号表，表内的每一项严格按照 sample 的序号排列，说明了媒体中的哪一个 sample 是关键帧。如果此表不存在，说明每一个 sample 都是一个关键帧，是一个随机存取点。

Sync sample atoms 的类型是'stss'。



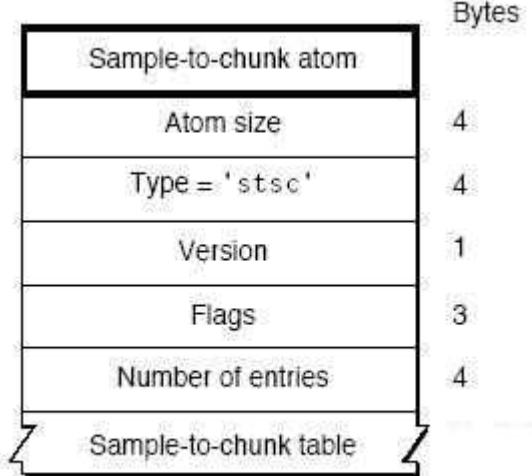
具体表结构如下：

字段	长度(单位: byte)	描述
尺寸	4	这个 atom 的字节数
类型	4	stss
版本	1	这个 atom 的版本
标志	3	这里为 0
条目数目	4	sync sample 的数目
sync sample		sync sample 表的结构
Sample 序号	4	是关键帧的 sample 序号

### 3.3.4 Sample-to-Chunk Atoms - STSC

当添加 samples 到 media 时，用 chunks 组织这些 sample，这样可以方便优化数据获取。一个 trunk 包含一个或多个 sample，chunk 的长度可以不同，chunk 内的 sample 的长度也可以不同。sample-to-chunk atom 存储 sample 与 chunk 的映射关系。

Sample-to-chunk atoms 的类型是'stsc'。它也有一个表来映射 sample 和 trunk 之间的关系，查看这张表，就可以找到包含指定 sample 的 trunk，从而找到这个 sample。



具体表结构如下：

字段	长度 (单位: byte)	描述
尺寸	4	这个 atom 的字节数
类型	4	stsc
版本	1	这个 atom 的版本
标志	3	这里为 0
条目数目	4	sample-to-chunk 的数目
sample-to-chunk		sample-to-chunk 表的结构
First chunk	4	这个 table 使用的第一个 chunk 序号
Samples per chunk	4	当前 trunk 内的 sample 数目
Sample description ID	4	与这些 sample 关联的 sample description 的序号

### 3.3.5 Sample Size Atoms - STSZ

sample size atoms 定义了每个 sample 的大小，它的类型是'stsz'，包含了媒体中全部 sample 的数目和一张给出每个 sample 大小的表。

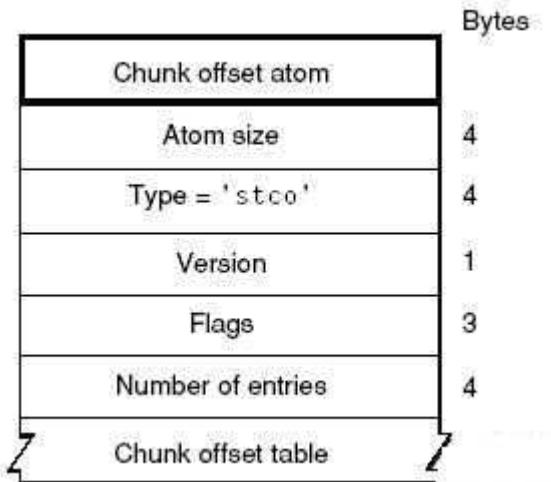
Sample size atom	Bytes
Atom size	4
Type = 'stsz'	4
Version	1
Flags	3
Sample size	4
Number of entries	4
Sample size table	

具体的表的结构如下：

字段	长度(单位: byte)	描述
尺寸	4	这个 atom 的字节数
类型	4	stsz
版本	1	这个 atom 的版本
标志	3	这里为 0
Sample size	4	全部 sample 的数目。如果所有的 sample 有相同的长度，这个字段就是这个值。否则，这个字段的值就是 0。那些长度存在 sample size 表中
条目数目	4	sample size 的数目
sample size	4	sample size 表的结构。这个表根据 sample number 索引，第一项就是第一个 sample，第二项就是第二个 sample

### 3.3.5 Chunk Offset Atoms - STCO

Chunk offset atoms 定义了每个 trunk 在媒体流中的位置，它的类型是'stco'。位置有两种可能，32 位的和 64 位的，后者对非常大的电影很有用。在一个表中只会有一种可能，这个位置是在整个文件中的，而不是在任何 atom 中的，这样做就可以直接在文件中找到媒体数据，而不用解释 atom。需要注意的是一旦前面的 atom 有了任何改变，这张表都要重新建立，因为位置信息已经改变了。



具体表结构如下：

字段	长度 (单位: byte)	描述
尺寸	4	这个 atom 的字节数
类型	4	"stco"或"co64"
版本	1	这个 atom 的版本
标志	3	这里为 0
条目数目	4	chunk offset 的数目
chunk offset		字节偏移量从文件开始到当前 chunk。这个表根据 chunk number 索引，第一项就是第一个 chunk，第二项就是第二个 chunk
大小	n	每个 sample 的大小,如果类型="scto" 大小为 4 个字节, 如果类型="co64", 大小为 8 个字节

### 3.3.6 Composition Time to Sample Box- CTTS

Composition Time to Sample Box 提供了在 dts(解码时间戳)与 pts(显示时间戳)的时间的偏移量, 它的类型是'ctts'。因为需要纠正时间的帧的 pts 一定比 dts 要大, 所以每一个项的值一定是正值。具体可以通过  $\text{pts}(n)=\text{dts}(n)+\text{ctts}(n)$  进行简单计算。

具体的表结构如下：

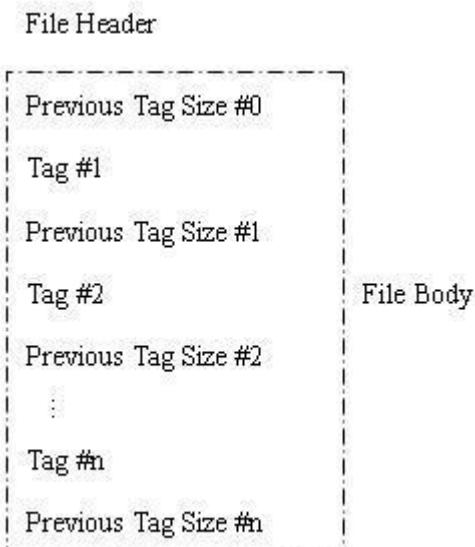
字段	长度 (单位: byte)	描述
尺寸	4	这个 atom 的字节数
类型	4	"ctts"
版本	1	这个 atom 的版本
标志	3	这里为 0
条目数目	4	ctts 的数目
Sample count	4	有相同的 Sample_offset 的连续 sample 的数目

# 第十二章 flv 文件格式分析

## 12.1 概述

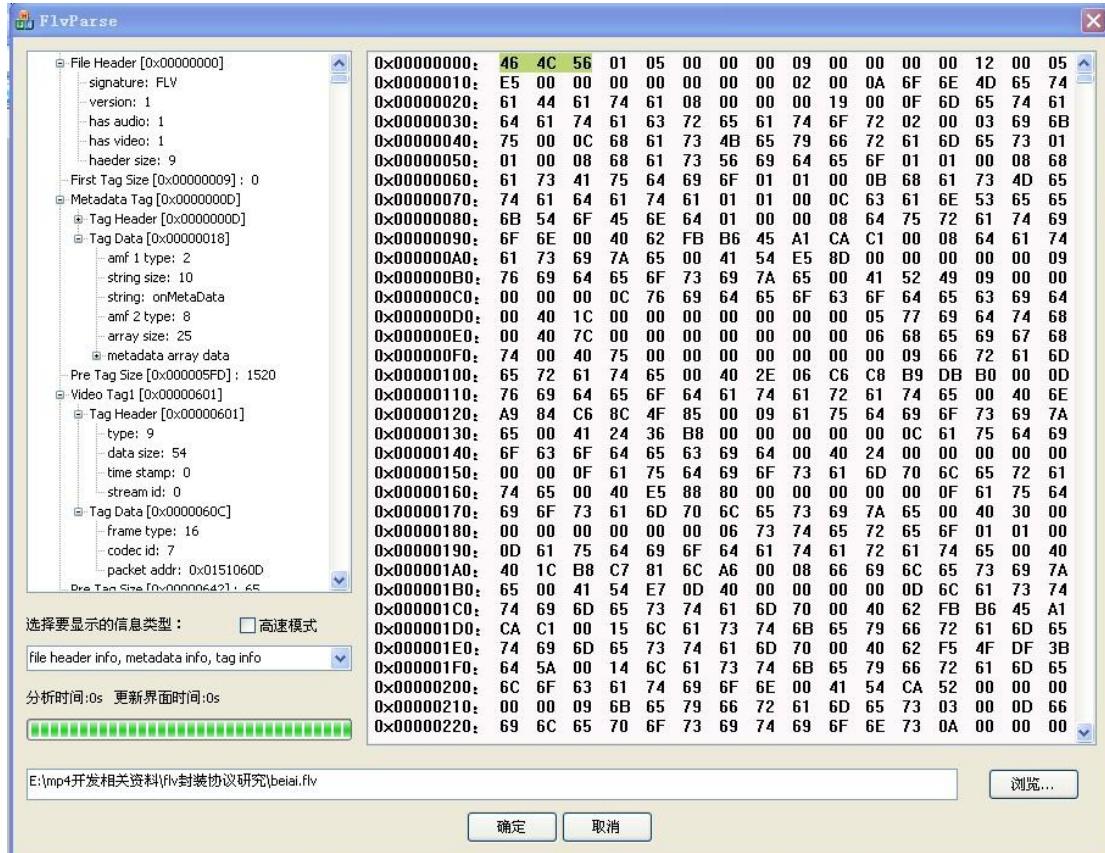
FLV 视频格式是 Adobe 公司设计开发的一种流媒体的封装格式，总体上看，FLV 包括文件头（Flv Header）和文件体（Flv Body）两部分，其中文件体由一系列的 Tag 及 Tag Size 对组成。Tag 又可以分成三类：audio, video, script，分别代表音频流，视频流，脚本流（关键字或者文件信息之类）。

## 12.2 文件总体结构



其中，Previous Tag Size 紧跟在每个 Tag 之后，占 4 个字节表示一个 UI32 类型的数值，表示前面一个 Tag 的大小。需要注意的是，Previous Tag Size #0 的值总是为 0。Tag 类型包括视频、音频和 Script，且每个 Tag 只能包含一种类型的数据。

具体的工具分析如下：



## 12.3 文件结构分析

### 12.3.1 flv 文件头的结构

在 ffmpeg 进行格式识别的时候，是以头部的前 3 个字节进行探测，识别到'F' 'L' 'V'即可认定该格式为 flv 格式。

FLV 头部	长度(byte)	描述
文件类型	3	'F' (0x46) 'L' (0x4C) 'V' (0x56)
版本	1	版本，目前为 1 (0x01)
流信息	1	1. UB[7]~UB[3], 前 5 位保留，必须为 0。 2. UB[2] 表示是否存在音频 Tag。 3. UB[1] 该位保留，必须为 0。 4. UB[0] 表示是否存在视频。 5.

		6.
header 长度	4	整个文件头的长度，一般是 9 (3+1+1+4)。个人感觉该字段多余，可以去掉。

由于第一个标识长度的 4 字节签名无 tag，但为了格式上的统一，所以可以划为头部。

### 12.3.2 body 主体结构

Tag 包括 Tag Header 和 Tag Data 两部分。不同类型的 Tag 的 Header 结构是相同的，但是 Data 结构各不相同。当前版本的 Tag Header 结构占用 11 个字节。

#### 12.3.2.1 Tag Header 结构

基于现在版本的 Tag Header 结构固定占用 11 个字节，具体描述见下表：

字段	长度 (byte)	描述
tag type	1	Tag 的类型，值：audio= (0x08)、video= (0x09) 和 script data= (0x12)，其他类型值被保留，一般可以直接忽略掉。
data size	3	表示该 Tag 真实 Data 部分的大小。
timestamp	3	表示该 Tag 的时间戳 (单位为 ms)，第一个 Tag 的时间戳总是 0。
timestampExten ded	1	当时间戳 24 位数值不够时，该字节作为最高位将时间戳扩展为 32 位值。左移 24 位与 Timestamp 值进行或操作
streamID	3	表示 stream id，总是 0

#### 12.3.2.2 Tag Data 结构

音视频 Tag 用开始的第一个字节包含视频数据的参数信息，根据 Tag Header 中的 Tag Type 类型值为 8(音频)，值为 9(视频)，该头部表示的意义会不同，具体结构如下：

##### ① 音频类型结构如下

字段	长度(单 位:bit)	描述
audio format	4	0 -- 未压缩 1 -- ADPCM 2 -- MP3 5 -- Nellymoser 8kHz momo 6 -- Nellymoser 7 -- G.711 A-law logarithmic PCM

		<p style="text-align: center;">《FFmpeg 基础库编程开发》</p> <p>8 --G.711 mu-law logarithmic PCM      9 -- reserved      10 --AAC(需要后面附加1个字节, 具体见下文)      11 --Speex      14--MP3 8-kHz      15 --Device -specific sound      Formats 7, 8, 14, and 15 为保留字段, ffmpeg 拆包直接跳过</p>
samplerate	2	<p>0 -- 5.5kHz      1 -- 11kHz      2 -- 22kHz      3 -- 44kHz      没有8kHz 的采样率, 音频8KHz, 一般为人声通话的 amr 格式所采用, 所以该字段和 Audio format 字段都没有提及。</p>
sample length	1	<p>即每一帧所占用的位宽。      0 -- 8Bit      1 -- 16Bit</p>
channel type	1	<p>0 --Momo(单声道)      1 -- Stereo(立体声)</p>

AAC 需要注意：

如果 SoundFormat 是 10 (AAC), TagDataHeader 后紧随着一个 1 个字节的数据 AACPacketType, 这个字段来表示 AACAUDIODATA 的类型: 0 = AAC sequence header, 1 = AAC raw。在 flv 中一般情况下, 带有该 AACPacketType 的 Tag 只会在第一个 audio Tag 中出现一次, 因为 aac 格式的音频需要在每帧 AAC ES 流前边添加 7 个字节 ADST 头(相当于帧头), 就是 AAC 的纯 ES 流要打包成 ADST 格式的 AAC 文件, 解码器才能正常播放.特别对于 RTSP,RTP 等实时传输流, ADST 必须存在, 否则传输过来的流不知道该怎么播放。

## ② 视频类型结构如下

字段	长度(单位:bit)	描述
video format	4	<p>1 -- keyframe      2 -- inner frame      3 -- disposable inner frame (H.263 only)</p>
codec id	4	1 = JPEG (废弃)

### 《FFmpeg 基础库编程开发》

2 -- Seronson H.263  
3 -- Screen video  
4 -- On2 VP6  
5 -- On2 VP6 without channel  
6 -- Screen video version 2  
7 -- AVC(h264)

大小为：Tag Header 中的 Data size - Tag Data Header,根据大小读取数据即可。

#### 12.3.2.3 script Tag 结构如下

如果 TAG 包中的 TagType==18 时，就表示这个 TAG 是 SCRIPT Tag。该类型 Tag 又通常被称为 Metadata Tag，会放一些关于 FLV 视频和音频的参数信息，如 duration、width、height 等。通常该类型 Tag 会跟在 File Header 后面作为第一个 Tag 出现，而且只有一个。

一般来说，该 Tag Data 结构包含两个 AMF 包。AMF (Action Message Format) 是 Adobe 设计的一种通用数据封装格式，在 Adobe 的很多产品中应用，简单来说，AMF 不区分根节点与子节点，将不同类型的数据用统一的格式来描述。第一个 AMF 包封装字符串类型数据，即：“02” type+string length+“onMetaData”。第二个 AMF 包封装一个数组类型，这个数组中包含了音视频信息项的名称和值。

AMF 具体表定义和结构如下：

字段	长度(单位:byte)	描述
data type	1	数据的类型
data	n	If Type = 0, DOUBLE(8 个字节)  If Type = 1, BOOL(1 个字节)  If Type = 2, 后续:2 字节(表征字符串长度)+字符串数据  If Type = 3, 遵从 Object memeber 表结构，可以看做 array 的一个数据项。  If Type = 8, 遵从 MixedArray 结构表。  If Type = 10, 遵从 normal array 表。  If Type = 11, 日期类型

Type='0x08' MixedArray 内部结构定义:

字段	长度(单位:byte)	描述
object number	4	数组中包括的对象数目。
object member	n	数据成员具体见下表。
end flag	3	数组的结束标志总为'0x09'。

MixedArray Object member 具体结构如下 (采用 key-value 结构):

字段	长度(单位:byte)	描述
key length	2	对象的名称长度。
stringData	n	对象名称, 长度由 StringLength 指出。
object type	1	遵从 AMF 定义, 可以为数组。
data	n	遵从 AMF 定义。

Type='0x0a' normal array 表结构:

字段	长度(单位:byte)	描述
object number	4	数组中包括的对象数目。
object member	n	遵从 AMF 表结构。
end flag	3	数组的结束标志总为'0x09'。

在 script tag 中常用的字段的键表如下:

字段	类型	描述
hasKeyFrames	bool	无
hasVideo	bool	无
hasAudio	bool	无
hasMetaData	bool	无

canSeekToEnd	bool	无
duration	Number	单位为秒
datasize	Number	实际的音视频数据的总的大小
videosize	Number	实际的视频数据的大小
audiosize	Number	实际的音频数据的大小
width	Number	视频的原始的宽度
height	Number	视频的原始的高度
framerate	Number	视频的帧率
videodatarate	Number	数值*1024 为比特率
audiosamplerate	Number	音频采样率
audiosamplesize	Number	音频每个 sample 的位宽
filesize	Number	整体文件的大小
lastkeyframesta mp	Number	最后关键帧的时间戳
lastkeyframela ction	Number	最后关键帧的在文件中的偏移量

#### 附加关键帧索引(Keyframes)

Adobe 的官方文档中并没有 keyframes 头，但是由于 flv 的每一个 tag 没有同步头，所以在进行 seek 时只能不断的通过往下读取数据来进行判断，这在网络流媒体播放时是不能忍受的(优酷的 flv 都带有 keyframes)所以在 script tag 中加入了该关键帧的索引表，以进行快速的 seek 等操作。包含着 2 个内容 'filepositions' and 'times' 分别指的是关键帧的文件位置和关键帧的 PTS. 通过 keyframes 可以建立起自己的 Index，然后再 seek 和快进快退的操作中，快速有效的跳转到你想要找的关键帧的位置进行处理。

具体结构如下：

#### keyframes

- filepositions(在文件中的 offset) value(普通数组，遵从 amf 协议)
- times(关键帧的时间) value(普通数组，遵从 amf 协议)。

# 附录 A：常见问题

## 1 ffmpeg 从内存中读取数据

ffmpeg 一般情况下支持打开一个本地文件，例如“C:\test.avi”

或者是一个流媒体协议的 URL，例如“rtmp://222.31.64.208/vod/test.flv”

其打开文件的函数是 avformat\_open\_input()，直接将文件路径或者流媒体 URL 的字符串传递给该函数就可以了。但其是否支持从内存中读取数据呢？这个问题困扰了我很长时间。当时在做项目的时候，通过 Winpcap 抓取网络上的 RTP 包，打算直接送给 ffmpeg 进行解码。一直没能找到合适的方法。因为抓取的数据包是存在内存中的，所以无法传递给 avformat\_open\_input() 函数其路径（根本没有路径==）。当然也可以将抓取的数据报存成文件，然后用 ffmpeg 打开这个文件，但是这样的话，程序的就太难控制了。

后来经过分析 ffmpeg 的源代码，发现其竟然是可以从内存中读取数据的，代码很简单，如下所示：

```
AVFormatContext *ic = NULL;
ic = avformat_alloc_context();
unsigned char * iobuffer=(unsigned char *)av_malloc(32768);
AVIOContext *avio =avio_alloc_context(iobuffer, 32768,0,buffer,fill_iobuffer,NULL,NULL);
ic->pb=avio;
err = avformat_open_input(&ic, is->filename, is->iformat, &format_opts);
```

关键要在 avformat\_open\_input() 之前初始化一个 AVIOContext，而且将原本的 AVFormatContext 的指针 pb (AVIOContext 类型) 指向这个自行初始化 AVIOContext。当自行指定了 AVIOContext 之后，avformat\_open\_input() 里面的 URL 参数就不起作用了。示例代码开辟了一块空间 iobuffer 作为 AVIOContext 的缓存。

此外 buffer 就是期望读取数据的内存，fill\_iobuffer 则是读取 buffer 数据至 iobuffer 的回调函数。fill\_iobuffer() 形式(参数，返回值)是固定的，是一个回调函数，如下所示(只是个例子，具体怎么读取数据可以自行设计)。

//把数据从 buffer 向 iobuf 传-----

//AVIOContext 使用的回调函数！

//注意：返回值是读取的字节数

//手动初始化 AVIOContext 只需要两个东西：内容来源的 buffer，和读取这个 Buffer 到 FFmpeg 中的函数

```
int fill_iobuffer(void * buffer,uint8_t *iobuf, int bufsize){
```

```
    int i;
    for(i=0;i<bufsize;i++){
        iobuf[i]=mediabuf_get();
    }
    return i;
}
```

## 2 MFC 中使用 SDL 播放音频没有声音的解决方法

此处所说的音频是指的纯音频，不包含视频的那种。

在控制台中使用 SDL 播放音频，一般情况下不会有问题。

但是在 MFC 中使用 SDL 播放音频的时候，会出现没有声音的情况。经过长时间探索，没有找到特别好的解决方案，但是有一种方式可以让声音播放出来：那就是让 SDL 显示图像（视频）时候的那个对话框弹出来，声音就会出现了。具体的方法可以加载一张图片（比如说 BMP），下图所示代码片段为加载 BMP 图片的代码。

```
SDL_Surface *screen = SDL_SetVideoMode(640, 480, 8, SDL_SWSURFACE);
SDL_Surface *image;
/* Load the BMP file into a surface */
image = SDL_LoadBMP("background.bmp");
if (image == NULL) {
    return 0;
}
/*
* Palettized screen modes will have a default palette (a standard
* 8*8*4 colour cube), but if the image is palettized as well we can
* use that palette for a nicer colour matching
*/
if (image->format->palette && screen->format->palette) {
    SDL_SetColors(screen, image->format->palette->colors, 0,
                  image->format->palette->nColors);
}
/* Blit onto the screen surface */
if(SDL_BlitSurface(image, NULL, screen, NULL) < 0)
    fprintf(stderr, "BlitSurface error: %s\n", SDL_GetError());
SDL_UpdateRect(screen, 0, 0, image->w, image->h);
```

不明白这是为什么，但是程序就可以出声了。

## 附录 B：经典代码示例

### output\_example.c 事例代码

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif
#include "libavformat/avformat.h"
#include "libswscale/swscale.h"
#undef exit
/* 5 seconds stream duration */
#define STREAM_DURATION 5.0
#define STREAM_FRAME_RATE 25 /* 25 images/s */
```

## 《FFmpeg 基础库编程开发》

```
#define STREAM_NB_FRAMES ((int)(STREAM_DURATION * STREAM_FRAME_RATE))
#define STREAM_PIX_FMT PIX_FMT_YUV420P /* default pix_fmt */
static int sws_flags = SWS_BICUBIC;
/*****************/
/* audio output */
float t, tincr, tincr2;
int16_t *samples;
uint8_t *audio_outbuf;
int audio_outbuf_size;
int audio_input_frame_size;
/*
 * add an audio output stream
 */
static AVStream *add_audio_stream(AVFormatContext *oc, int codec_id)
{
    AVCodecContext *c;
    AVStream *st;
    st = av_new_stream(oc, 1);
    if (!st) {
        fprintf(stderr, "Could not alloc stream\n");
        exit(1);
    }
    c = st->codec;
    c->codec_id = codec_id;
    c->codec_type = CODEC_TYPE_AUDIO;
    /* put sample parameters */
    c->bit_rate = 64000;
    c->sample_rate = 44100;
    c->channels = 2;
    return st;
}
static void open_audio(AVFormatContext *oc, AVStream *st)
{
    AVCodecContext *c;
    AVCodec *codec;
    c = st->codec;
    /* find the audio encoder */
    codec = avcodec_find_encoder(c->codec_id);
    if (!codec) {
        fprintf(stderr, "codec not found\n");
        exit(1);
    }
    /* open it */
```

## 《FFmpeg 基础库编程开发》

```
if (avcodec_open(c, codec) < 0) {
    fprintf(stderr, "could not open codec\n");
    exit(1);
}

/* init signal generator */
t = 0;
tincr = 2 * M_PI * 110.0 / c->sample_rate;
/* increment frequency by 110 Hz per second */
tincr2 = 2 * M_PI * 110.0 / c->sample_rate / c->sample_rate;
audio_outbuf_size = 10000;
audio_outbuf = av_malloc(audio_outbuf_size);
/* ugly hack for PCM codecs (will be removed ASAP with new PCM
   support to compute the input frame size in samples */
if (c->frame_size <= 1) {
    audio_input_frame_size = audio_outbuf_size / c->channels;
    switch(st->codec->codec_id) {
        case CODEC_ID_PCM_S16LE:
        case CODEC_ID_PCM_S16BE:
        case CODEC_ID_PCM_U16LE:
        case CODEC_ID_PCM_U16BE:
            audio_input_frame_size >>= 1;
            break;
        default:
            break;
    }
} else {
    audio_input_frame_size = c->frame_size;
}
samples = av_malloc(audio_input_frame_size * 2 * c->channels);
}

/* prepare a 16 bit dummy audio frame of 'frame_size' samples and
   'nb_channels' channels */
static void get_audio_frame(int16_t *samples, int frame_size, int nb_channels)
{
    int j, i, v;
    int16_t *q;
    q = samples;
    for(j=0;j<frame_size;j++) {
        v = (int)(sin(t) * 10000);
        for(i = 0; i < nb_channels; i++)
            *q++ = v;
        t += tincr;
        tincr += tincr2;
    }
}
```

```

    }
}

static void write_audio_frame(AVFormatContext *oc, AVStream *st)
{
    AVCodecContext *c;
    AVPacket pkt;
    av_init_packet(&pkt);
    c = st->codec;
    get_audio_frame(samples, audio_input_frame_size, c->channels);
    pkt.size= avcodec_encode_audio(c, audio_outbuf, audio_outbuf_size, samples);
    pkt.pts= av_rescale_q(c->coded_frame->pts, c->time_base, st->time_base);
    pkt.flags |= PKT_FLAG_KEY;
    pkt.stream_index= st->index;
    pkt.data= audio_outbuf;
    /* write the compressed frame in the media file */
    if (av_write_frame(oc, &pkt) != 0) {
        fprintf(stderr, "Error while writing audio frame\n");
        exit(1);
    }
}

static void close_audio(AVFormatContext *oc, AVStream *st)
{
    avcodec_close(st->codec);
    av_free(samples);
    av_free(audio_outbuf);
}

/*****************************************/
/* video output */
AVFrame *picture, *tmp_picture;
uint8_t *video_outbuf;
int frame_count, video_outbuf_size;
/* add a video output stream */
static AVStream *add_video_stream(AVFormatContext *oc, int codec_id)
{
    AVCodecContext *c;
    AVStream *st;
    st = av_new_stream(oc, 0);
    if (!st) {
        fprintf(stderr, "Could not alloc stream\n");
        exit(1);
    }
    c = st->codec;
    c->codec_id = codec_id;
}

```

```

c->codec_type = CODEC_TYPE_VIDEO;
/* put sample parameters */
c->bit_rate = 400000;
/* resolution must be a multiple of two */
c->width = 352;
c->height = 288;
/* time base: this is the fundamental unit of time (in seconds) in terms
   of which frame timestamps are represented. for fixed-fps content,
   timebase should be 1/framerate and timestamp increments should be
   identically 1. */
c->time_base.den = STREAM_FRAME_RATE;
c->time_base.num = 1;
c->gop_size = 12; /* emit one intra frame every twelve frames at most */
c->pix_fmt = STREAM_PIX_FMT;
if (c->codec_id == CODEC_ID_MPEG2VIDEO) {
    /* just for testing, we also add B frames */
    c->max_b_frames = 2;
}
if (c->codec_id == CODEC_ID_MPEG1VIDEO){
    /* Needed to avoid using macroblocks in which some coeffs overflow.
       This does not happen with normal video, it just happens here as
       the motion of the chroma plane does not match the luma plane. */
    c->mb_decision=2;
}
// some formats want stream headers to be separate
if(!strcmp(oc->oformat->name, "mp4") || !strcmp(oc->oformat->name, "mov") || !strcmp(oc->oformat->name, "3gp"))
    c->flags |= CODEC_FLAG_GLOBAL_HEADER;
return st;
}

static AVFrame *alloc_picture(int pix_fmt, int width, int height)
{
    AVFrame *picture;
    uint8_t *picture_buf;
    int size;
    picture = avcodec_alloc_frame();
    if (!picture)
        return NULL;
    size = avpicture_get_size(pix_fmt, width, height);
    picture_buf = av_malloc(size);
    if (!picture_buf) {
        av_free(picture);
        return NULL;
    }
}

```

## 《FFmpeg 基础库编程开发》

```
avpicture_fill((AVPicture *)picture, picture_buf,
               pix_fmt, width, height);
return picture;
}

static void open_video(AVFormatContext *oc, AVStream *st)
{
    AVCodec *codec;
    AVCodecContext *c;
    c = st->codec;
    /* find the video encoder */
    codec = avcodec_find_encoder(c->codec_id);
    if (!codec) {
        fprintf(stderr, "codec not found\n");
        exit(1);
    }
    /* open the codec */
    if (avcodec_open(c, codec) < 0) {
        fprintf(stderr, "could not open codec\n");
        exit(1);
    }
    video_outbuf = NULL;
    if (!(oc->oformat->flags & AVFMT_RAWPICTURE)) {
        /* allocate output buffer */
        /* XXX: API change will be done */
        /* buffers passed into lav* can be allocated any way you prefer,
           as long as they're aligned enough for the architecture, and
           they're freed appropriately (such as using av_free for buffers
           allocated with av_malloc) */
        video_outbuf_size = 200000;
        video_outbuf = av_malloc(video_outbuf_size);
    }
    /* allocate the encoded raw picture */
    picture = alloc_picture(c->pix_fmt, c->width, c->height);
    if (!picture) {
        fprintf(stderr, "Could not allocate picture\n");
        exit(1);
    }
    /* if the output format is not YUV420P, then a temporary YUV420P
       picture is needed too. It is then converted to the required
       output format */
    tmp_picture = NULL;
    if (c->pix_fmt != PIX_FMT_YUV420P) {
        tmp_picture = alloc_picture(PIX_FMT_YUV420P, c->width, c->height);
```

## 《FFmpeg 基础库编程开发》

```
if (!tmp_picture) {
    fprintf(stderr, "Could not allocate temporary picture\n");
    exit(1);
}
}

/* prepare a dummy image */
static void fill_yuv_image(AVFrame *pict, int frame_index, int width, int height)
{
    int x, y, i;
    i = frame_index;
    /* Y */
    for(y=0;y<height;y++) {
        for(x=0;x<width;x++) {
            pict->data[0][y * pict->linesize[0] + x] = x + y + i * 3;
        }
    }
    /* Cb and Cr */
    for(y=0;y<height/2;y++) {
        for(x=0;x<width/2;x++) {
            pict->data[1][y * pict->linesize[1] + x] = 128 + y + i * 2;
            pict->data[2][y * pict->linesize[2] + x] = 64 + x + i * 5;
        }
    }
}

static void write_video_frame(AVFormatContext *oc, AVStream *st)
{
    int out_size, ret;
    AVCodecContext *c;
    static struct SwsContext *img_convert_ctx;
    c = st->codec;
    if (frame_count >= STREAM_NB_FRAMES) {
        /* no more frame to compress. The codec has a latency of a few
         frames if using B frames, so we get the last frames by
         passing the same picture again */
    } else {
        if (c->pix_fmt != PIX_FMT_YUV420P) {
            /* as we only generate a YUV420P picture, we must convert it
             to the codec pixel format if needed */
            if (img_convert_ctx == NULL) {
                img_convert_ctx = sws_getContext(c->width, c->height,
                                                PIX_FMT_YUV420P,
                                                c->width, c->height,
```

《FFmpeg 基础库编程开发》  
c->pix\_fmt,  
sws\_flags, NULL, NULL, NULL);

```
if (img_convert_ctx == NULL) {
    fprintf(stderr, "Cannot initialize the conversion context\n");
    exit(1);
}
fill_yuv_image(tmp_picture, frame_count, c->width, c->height);
sws_scale(img_convert_ctx, tmp_picture->data, tmp_picture->linesize,
          0, c->height, picture->data, picture->linesize);
} else {
    fill_yuv_image(picture, frame_count, c->width, c->height);
}
}

if (oc->oformat->flags & AVFMT_RAWPICTURE) {
/* raw video case. The API will change slightly in the near
futur for that */
AVPacket pkt;
av_init_packet(&pkt);
pkt.flags |= PKT_FLAG_KEY;
pkt.stream_index= st->index;
pkt.data= (uint8_t *)picture;
pkt.size= sizeof(APPicture);
ret = av_write_frame(oc, &pkt);
} else {
/* encode the image */
out_size = avcodec_encode_video(c, video_outbuf, video_outbuf_size, picture);
/* if zero size, it means the image was buffered */
if (out_size > 0) {
    AVPacket pkt;
    av_init_packet(&pkt);
    pkt.pts= av_rescale_q(c->coded_frame->pts, c->time_base, st->time_base);
    if(c->coded_frame->key_frame)
        pkt.flags |= PKT_FLAG_KEY;
    pkt.stream_index= st->index;
    pkt.data= video_outbuf;
    pkt.size= out_size;
    /* write the compressed frame in the media file */
    ret = av_write_frame(oc, &pkt);
} else {
    ret = 0;
}
```

```

}

if (ret != 0) {
    fprintf(stderr, "Error while writing video frame\n");
    exit(1);
}

frame_count++;

}

static void close_video(AVFormatContext *oc, AVStream *st)
{
    avcodec_close(st->codec);
    av_free(picture->data[0]);
    av_free(picture);
    if (tmp_picture) {
        av_free(tmp_picture->data[0]);
        av_free(tmp_picture);
    }
    av_free(video_outbuf);
}

/*****************************************/
/* media file output */
int main(int argc, char **argv)
{
    const char *filename;
    AVOutputFormat *fmt;
    AVFormatContext *oc;
    AVStream *audio_st, *video_st;
    double audio_pts, video_pts;
    int i;

    /* initialize libavcodec, and register all codecs and formats */
    av_register_all();
    if (argc != 2) {
        printf("usage: %s output_file\n"
               "API example program to output a media file with libavformat.\n"
               "The output format is automatically guessed according to the file extension.\n"
               "Raw images can also be output by using '%d' in the filename\n"
               "\n", argv[0]);
        exit(1);
    }
    filename = argv[1];
    /* auto detect the output format from the name. default is
       mpeg. */
    fmt = guess_format(NULL, filename, NULL);
    if (!fmt) {

```

## 《FFmpeg 基础库编程开发》

```
printf("Could not deduce output format from file extension: using MPEG\n");
fmt = guess_format("mpeg", NULL, NULL);
}
if (!fmt) {
    fprintf(stderr, "Could not find suitable output format\n");
    exit(1);
}
/* allocate the output media context */
oc = av_alloc_format_context();
if (!oc) {
    fprintf(stderr, "Memory error\n");
    exit(1);
}
oc->oformat = fmt;
snprintf(oc->filename, sizeof(oc->filename), "%s", filename);
/* add the audio and video streams using the default format codecs
   and initialize the codecs */
video_st = NULL;
audio_st = NULL;
if (fmt->video_codec != CODEC_ID_NONE) {
    video_st = add_video_stream(oc, fmt->video_codec);
}
if (fmt->audio_codec != CODEC_ID_NONE) {
    audio_st = add_audio_stream(oc, fmt->audio_codec);
}
/* set the output parameters (must be done even if no
   parameters). */
if (av_set_parameters(oc, NULL) < 0) {
    fprintf(stderr, "Invalid output format parameters\n");
    exit(1);
}
dump_format(oc, 0, filename, 1);
/* now that all the parameters are set, we can open the audio and
   video codecs and allocate the necessary encode buffers */
if (video_st)
    open_video(oc, video_st);
if (audio_st)
    open_audio(oc, audio_st);
/* open the output file, if needed */
if (!(fmt->flags & AVFMT_NOFILE)) {
    if (url_fopen(&oc->pb, filename, URL_WRONLY) < 0) {
        fprintf(stderr, "Could not open '%s'\n", filename);
        exit(1);
    }
}
```

```

    }

}

/* write the stream header, if any */
av_write_header(oc);
for(;;) {
    /* compute current audio and video time */
    if (audio_st)
        audio_pts = (double)audio_st->pts.val * audio_st->time_base.num / audio_st->time_base.den;
    else
        audio_pts = 0.0;
    if (video_st)
        video_pts = (double)video_st->pts.val * video_st->time_base.num / video_st->time_base.den;
    else
        video_pts = 0.0;
    if ((!audio_st || audio_pts >= STREAM_DURATION) &&
        (!video_st || video_pts >= STREAM_DURATION))
        break;
    /* write interleaved audio and video frames */
    if (!video_st || (video_st && audio_st && audio_pts < video_pts)) {
        write_audio_frame(oc, audio_st);
    } else {
        write_video_frame(oc, video_st);
    }
}
/* close each codec */
if (video_st)
    close_video(oc, video_st);
if (audio_st)
    close_audio(oc, audio_st);
/* write the trailer, if any */
av_write_trailer(oc);
/* free the streams */
for(i = 0; i < oc->nb_streams; i++) {
    av_freep(&oc->streams[i]->codec);
    av_freep(&oc->streams[i]);
}
if (!(fmt->flags & AVFMT_NOFILE)) {
    /* close the output file */
    url_fclose(&oc->pb);
}
/* free the stream */
av_free(oc);
return 0;
}

```

}

## 附录 c: ffmpeg 参数中文详细解释

### a) 通用选项

-L license

-h 帮助

-formats 显示可用的格式，编解码的，协议的...

-f fmt 强迫采用格式 fmt

-I filename 输入文件

-y 覆盖输出文件

-t duration 设置纪录时间 hh:mm:ss[.xxx]格式的记录时间也支持

-ss position 搜索到指定的时间 [-]hh:mm:ss[.xxx]的格式也支持

-title string 设置标题

-author string 设置作者

-copyright string 设置版权

-comment string 设置评论

-target type 设置目标文件类型(vcd,svcd,dvd) 所有的格式选项（比特率，编解码以及缓冲区大小）自动设置，只需要输入如下的就可以了：ffmpeg -i myfile.avi -target vcd /tmp/vcd.mpg

-hq 激活高质量设置

-itsoffset offset 设置以秒为基准的时间偏移，该选项影响所有后面的输入文件。该偏移被加到输入文件的时戳，定义一个正偏移意味着相应的流被延迟了 offset 秒。 [-]hh:mm:ss[.xxx]的格式也支持

### b) 视频选项

-b bitrate 设置比特率，缺省 200kb/s

-r fps 设置帧频 缺省 25

-s size 设置帧大小 格式为 WXH 缺省 160X128.下面的简写也可以直接使用：

Sqcif 128X96 qcif 176X144 cif 252X288 4cif 704X576

-aspect aspect 设置横纵比 4:3 16:9 或 1.3333 1.7777

-croptop size 设置顶部切除带大小 像素单位

-cropbottom size - cropleft size - cropright size

-padtop size 设置顶部补齐的大小 像素单位

-padbottom size - padleft size - padright size - padcolor color 设置补齐条颜色(hex,6 个 16 进制的数, 红:绿:蓝排列, 比如 000000 代表黑色)

-vn 不做视频记录

-bt tolerance 设置视频码率容忍度 kbit/s

-maxrate bitrate 设置最大视频码率容忍度

-minrate bitrate 设置最小视频码率容忍度

-bufsize size 设置码率控制缓冲区大小

-vcodec codec 强制使用 codec 编解码方式。如果用 copy 表示原始编解码数据必须被拷贝。

-sameq 使用同样视频质量作为源 (VBR)

-pass n 选择处理遍数 (1 或者 2)。两遍编码非常有用。第一遍生成统计信息，第二遍生成精确的请求的码率

## 《FFmpeg 基础库编程开发》

-passlogfile file 选择两遍的纪录文件名为 file

### c)高级视频选项

-g gop\_size 设置图像组大小

-intra 仅适用帧内编码

-qscale q 使用固定的视频量化标度(VBR)

-qmin q 最小视频量化标度(VBR)

-qmax q 最大视频量化标度(VBR)

-qdiff q 量化标度间最大偏差 (VBR)

-qblur blur 视频量化标度柔化(VBR)

-qcomp compression 视频量化标度压缩(VBR)

-rc\_init\_cplx complexity 一遍编码的初始复杂度

-b\_qfactor factor 在 p 和 b 帧间的 qp 因子

-i\_qfactor factor 在 p 和 i 帧间的 qp 因子

-b\_qoffset offset 在 p 和 b 帧间的 qp 偏差

-i\_qoffset offset 在 p 和 i 帧间的 qp 偏差

-rc\_eq equation 设置码率控制方程 默认 tex<sup>qComp</sup>

-rc\_override override 特定间隔下的速率控制重载

-me method 设置运动估计的方法 可用方法有 zero phods log x1 epzs(缺省) full

-dct\_algo algo 设置 dct 的算法 可用的有 0 FF\_DCT\_AUTO 缺省的 DCT 1 FF\_DCT\_FASTINT 2 FF\_DCT\_INT 3 FF\_DCT\_MMX 4 FF\_DCT\_MLIB 5 FF\_DCT\_ALTIVEC

-idct\_algo algo 设置 idct 算法。可用的有 0 FF\_IDCT\_AUTO 缺省的 IDCT 1 FF\_IDCT\_INT 2 FF\_IDCT\_SIMPLE 3 FF\_IDCT\_SIMPLEMMX 4 FF\_IDCT\_LIBMPEG2MMX 5 FF\_IDCT\_PS2 6 FF\_IDCT\_MLIB 7 FF\_IDCT\_ARM 8 FF\_IDCT\_ALTIVEC 9 FF\_IDCT\_SH4 10 FF\_IDCT\_SIMPLEARM

-er n 设置错误残留为 n 1 FF\_ER\_CAREFULL 缺省 2 FF\_ER\_COMPLIANT 3 FF\_ER.Aggressive 4 FF\_ER\_VERY.Aggressive

-ec bit\_mask 设置错误掩蔽为 bit\_mask, 该值为如下值的位掩码 1 FF\_EC\_GUESS\_MVS (default=enabled) 2 FF\_EC\_DEBLOCK (default=enabled)

-bf frames 使用 frames B 帧, 支持 mpeg1,mpeg2,mpeg4

-mbd mode 宏块决策 0 FF\_MB\_DECISION\_SIMPLE 使用 mb\_cmp 1 FF\_MB\_DECISION\_BITS 2 FF\_MB\_DECISION\_RD

-4mv 使用 4 个运动矢量 仅用于 mpeg4

-part 使用数据划分 仅用于 mpeg4

-bug param 绕过没有被自动监测到编码器的问题

-strict strictness 跟标准的严格性

-aic 使能高级帧内编码 h263+

-umv 使能无限运动矢量 h263+

-deinterlace 不采用交织方法

-interlace 强迫交织法编码仅对 mpeg2 和 mpeg4 有效。当你的输入是交织的并且你想要保持交织以最小图像损失的时候采用该选项。可选的方法是不交织, 但是损失更大

-psnr 计算压缩帧的 psnr

-vstats 输出视频编码统计到 vstats\_hhmmss.log

-vhook module 插入视频处理模块 module 包括了模块名和参数, 用空格分开

### D)音频选项

-ab bitrate 设置音频码率

-ar freq 设置音频采样率

-ac channels 设置通道 缺省为 1

-an 不使能音频纪录

-acodec codec 使用 codec 编解码

#### E) 音频/视频捕获选项

-vd device 设置视频捕获设备。比如/dev/video0

-vc channel 设置视频捕获通道 DV1394 专用

-tvstd standard 设置电视标准 NTSC PAL(SECAM)

-dv1394 设置 DV1394 捕获

-av device 设置音频设备 比如/dev/dsp

#### F) 高级选项

-map file:stream 设置输入流映射

-debug 打印特定调试信息

-benchmark 为基准测试加入时间

-hex 倾倒每一个输入包

-bitexact 仅使用位精确算法 用于编解码测试

-ps size 设置包大小, 以 bits 为单位

-re 以本地帧频读数据, 主要用于模拟捕获设备

-loop 循环输入流 (只工作于图像流, 用于 ffserver 测试)

## 附录 D: ffplay 的快捷键以及选项

ffplay 是 ffmpeg 工程中提供的播放器, 功能相当的强大, 凡是 ffmpeg 支持的视音频格式它基本上都支持。甚至连 VLC 不支持的一些流媒体都可以播放 (比如说 RTMP), 但是它的缺点是其不是图形化界面的, 必须通过键盘来操作。因此本文介绍一下它的快捷键以及选项。

### 快捷键

播放视音频文件的时候, 可以通过下列按键控制视音频的播放

按键	作用
q, ESC	退出
F	全屏
p, 空格	暂停
w	显示音频波形
s	逐帧显示
左/右方向键	向后/前 10s
上/下方向键	向后/前 1min
page down/page up	向后/前 10min
鼠标点击屏幕	跳转到指定位置 (根据鼠标位置相对屏幕的宽度计算)

### 选项

在播放视频前, 可以预设一些参数。

一般播放视频的时候，使用命令：

```
#ffplay "abc.flv"
```

如果我们希望能在播放完成后自动退出，则可以使用命令：

```
ffplay -autoexit "abc.flv";
```

所有的命令如下列表所示：

名称	有参数	作用
x	Y	强制屏幕宽度
y	Y	强制屏幕高度
s	Y	强制屏幕大小
fs	N	全屏
an	N	关闭音频
vn	N	关闭视频
ast	Y	设置想播放的音频流（需要指定流 ID）
vst	Y	设置想播放的视频流（需要指定流 ID）
sst	Y	设置想播放的字幕流（需要指定流 ID）
ss	Y	从指定位置开始播放，单位是秒
t	Y	播放指定时长的视频
nodisp	N	无显示屏幕
f	Y	强制封装格式
pix_fmt	Y	指定像素格式
stats	N	显示统计信息
idct	Y	IDCT 算法
ec	Y	错误隐藏方法
sync	Y	视 音 频 同 步 方 式 (type=audio/video/ext)
autoexit	N	播放完成自动退出
exitonkeydown	N	按下按键退出
exitonmousedown	N	按下鼠标退出
loop	Y	指定循环次数
framedrop	N	CPU 不够的时候丢帧
window_title	Y	显示窗口的标题
rdftspeed	Y	Rdft 速度
showmode	Y	显示方式(0 = video, 1 = waves, 2 = RDFT)
codec	Y	强制解码器

## 附录 E： ffmpeg 处理 rtmp 流媒体

1、将文件当做直播送至 live

```
ffmpeg -re -i localFile.mp4 -c copy -f flv rtmp://server/live/streamName
```

## 《FFmpeg 基础库编程开发》

2、将直播媒体保存至本地文件

```
ffmpeg -i rtmp://server/live/streamName -c copy dump.flv
```

3、将其中一个直播流，视频改用 h264 压缩，音频不变，送至另外一个直播服务流

```
ffmpeg -i rtmp://server/live/originalStream -c:a copy -c:v libx264 -vpre slow -f flv rtmp://server/live/h264Stream
```

4、将其中一个直播流，视频改用 h264 压缩，音频改用 faac 压缩，送至另外一个直播服务流

```
ffmpeg -i rtmp://server/live/originalStream -c:a libfaac -ar 44100 -ab 48k -c:v libx264 -vpre slow -vpre baseline -f flv  
rtmp://server/live/h264Stream
```

5、将其中一个直播流，视频不变，音频改用 faac 压缩，送至另外一个直播服务流

```
ffmpeg -i rtmp://server/live/originalStream -acodec libfaac -ar 44100 -ab 48k -vcodec copy -f flv  
rtmp://server/live/h264_AAC_Stream
```

6、将一个高清流，复制为几个不同视频清晰度的流重新发布，其中音频不变

```
ffmpeg -re -i rtmp://server/live/high_FMLE_stream -acodec copy -vcodec x264lib -s 640×360 -b 500k -vpre medium -vpre  
baseline rtmp://server/live/baseline_500k -acodec copy -vcodec x264lib -s 480×272 -b 300k -vpre medium -vpre baseline  
rtmp://server/live/baseline_300k -acodec copy -vcodec x264lib -s 320×200 -b 150k -vpre medium -vpre baseline  
rtmp://server/live/baseline_150k -acodec libfaac -vn -ab 48k rtmp://server/live/audio_only_AAC_48k
```

7、功能一样，只是采用-x264opts 选项

```
ffmpeg -re -i rtmp://server/live/high_FMLE_stream -c:a copy -c:v x264lib -s 640×360 -x264opts  
bitrate=500:profile=baseline:preset=slow rtmp://server/live/baseline_500k -c:a copy -c:v x264lib -s 480×272 -x264opts  
bitrate=300:profile=baseline:preset=slow rtmp://server/live/baseline_300k -c:a copy -c:v x264lib -s 320×200 -x264opts  
bitrate=150:profile=baseline:preset=slow rtmp://server/live/baseline_150k -c:a libfaac -vn -b:a 48k  
rtmp://server/live/audio_only_AAC_48k
```

8、将当前摄像头及音频通过 DSSHOW 采集，视频 h264、音频 faac 压缩后发布

```
ffmpeg -r 25 -f dshow -s 640×480 -i video="video source name":audio="audio source name" -vcodec libx264 -b 600k -vpre  
slow -acodec libfaac -ab 128k rtmp://server/application/stream_name
```

9、将一个 JPG 图片经过 h264 压缩循环输出为 mp4 视频

```
ffmpeg.exe -i INPUT.jpg -an -vcodec libx264 -coder 1 -flags +loop -cmp +chroma -subq 10 -qcomp 0.6 -qmin 10 -qmax 51  
-qdiff 4 -flags2 +dct8x8 -trellis 2 -partitions +parti8x8+parti4x4 -crf 24 -threads 0 -r 25 -g 25 -y OUTPUT.mp4
```

10、将普通流视频改用 h264 压缩，音频不变，送至高清流服务(新版本 FMS live=1)

```
ffmpeg -i rtmp://server/live/originalStream -c:a copy -c:v libx264 -vpre slow -f flv "rtmp://server/live/h264Stream live=1"
```