6min

第 5 章

FFmpeg 命令行实现音视频转码

音视频转码主要是指容器中音视频数据编码方式转换，例如 H.264 编码转换成 MPEG-4 编码、MP3 转换为 AAC；音视频码率的转换，例如 4Mb 的视频码率降为 2Mb；视频分辨率的转换，例如 1080P 视频转换为 720P、音频重采样等。音视频转码的一般步骤是先解码再编码，可以采取软编软解或硬编硬解的方式。视频解码一般是解压为 YUV 格式，音频解码一般是解码为 PCM 格式。常用的开源编码算法库包括 libx264、libx265 和 libmp3lame 等。

5.1 音视频编解码及转码简介

音视频压缩与编解码知识非常复杂，理论抽象，基础概念多，包括有损和无损压缩、帧内和帧间编码、对称和不对称编码等。技术参数也很多，包括帧率、码率、分辨率、关键帧、位深、压缩比等。视频编码最主要的工作就是压缩，分为帧内压缩和帧间压缩，然后又分为其他几个步骤，主要包括预测、变换、量化、熵编码、滤波处理等。

5.1.1 视频编解码简介

视频技术泛指将一系列静态影像以电信号的方式加以捕捉、记录、处理、存储、传送与重现的各种技术。当连续的图像变化每秒超过 24 帧画面以上时，根据视觉暂留原理，人眼无法辨别单幅的静态画面，看上去是平滑连续的视觉效果，这样连续的画面叫作视频。视频数据往往在时域和空域层面都有极强的相关性，这表示有大量的时域冗余信息和空域冗余信息，压缩技术就是去掉数据中的冗余信息。视频编码就是通过特定的压缩技术，将某个视频格式的文件转换成另外一种视频格式。去除时域冗余信息主要包括运动估计和运动补偿；去除空域冗余信息主要包括变换编码、量化编码和熵编码。运动补偿是通过先前的局部图像来预测、补偿当前的局部图像，可有效地减少帧序列冗余信息。运动表示是指不同区域的图像使用不同的运动向量来描述运动信息，运动向量通过熵编码进行压缩，熵编码在编码过程中不会丢失信息。运动估计是指从视频序列中抽取运动信息，通用的压缩标准使用基于块的运动估计和运动补偿。变换编码是指将空域信号变换到另一正交向量空间，使相关性

下降，数据冗余度减小。

未经编码的数字视频的数据量很大，存储和传输都比较困难。以一个分辨率为1920×1080，帧率为30的视频为例，共有1920×1080＝2 073 600像素，每像素是24b(假设采取RGB24)。也就是每张图片2 073 600×24＝49 766 400b。8b(位)＝1B(字节)，所以，49 766 400b＝6 220 800B≈6.22MB。这才是一幅1920×1080图片的原始大小(6.22MB)，再乘以帧率30。也就是说，1s视频的大小是186.6MB，1min大约是11GB，一部90min的电影约为1000GB。由此可见未经编码的视频数据是非常庞大的，所以必须经过编码压缩之后，视频数据才方便存储，方便在网络上传输。

5.1.2 音频编解码简介

数字音频涉及的基础概念非常多，包括采样、量化、编码、采样率、采样数、声道数、音频帧、比特率、PCM等。从模拟信号到数字信号的过程包括采样、量化、编码3个阶段。原始的音频数据中存在大量的冗余信息，有必要进行压缩处理。音频信号能压缩的基本依据包括声音信号中存在大量的冗余度，以及人的听觉具有强音能抑制同时存在的弱音现象。音频压缩编码，其原理是压缩掉冗余的信号，冗余信号是指不能被人耳感知到的信号，包括人耳听觉范围之外的音频信号及被掩蔽掉的音频信号。模拟音频信号转换为数字信号需要经过采样和量化。量化的过程被称为编码，根据不同的量化策略，产生了许多不同的编码方式，常见的编码方式有PCM和ADPCM。这些数据代表着无损的原始数字音频信号，添加一些文件头信息后就可以存储为WAV文件了，它是一种由微软和IBM联合开发的用于音频数字存储的标准，可以很容易地被解析和播放。在进一步了解音频处理和压缩之前需要明确几个概念，包括音调、响度、采样率、采样精度、声道数、音频帧长等。

5.1.3 音视频转码简介

音视频转码(Audio/Video Transcoding)是指将已经压缩编码的音视频码流转换成另一种格式的码流，以适应不同的网络带宽、不同的终端处理能力和不同的用户需求。音视频转码本质上是一个先解码，再编码的过程，因此转换前后的码流可能遵循相同的视频编码标准，也可能不遵循相同的视频编码标准。不同编码格式之间的数据转码指通过转码方法改变视频数据的编码格式。通常这种数据转码会改变视频数据的现有码流和分辨率。例如可以将基于MPEG-2格式的视频数据转换为DV、MPEG-4或其他编码格式，同时根据其转码目的，指定转码产生视频数据的码流和分辨率。

音视频转码的几个主要概念如下：

(1) 容器格式的转换，例如MP4转换为MOV。

(2) 容器中音视频数据编码方式转换，例如H.264编码转换成MPEG-4编码，MP3编码转换成AAC编码。

(3) 音视频码率的转换，例如4Mb的视频码率降为2Mb。

(4) 视频分辨率的转换，例如1080P视频转换为720P、音频重采样等。

使用 FFmpeg 进行音视频转码的主要流程如图 5-1 所示。

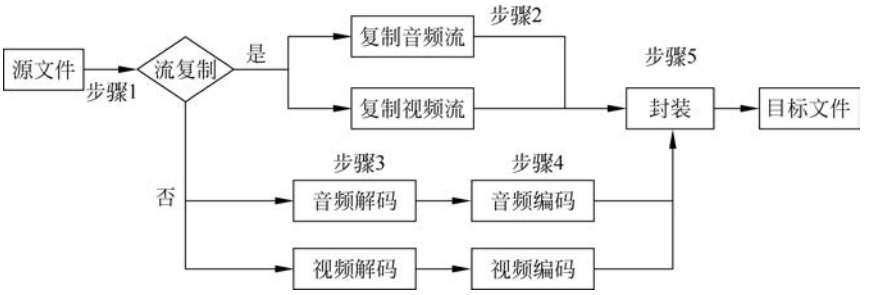

图 5-1 FFmpeg 的音视频转码流程

其中，流复制是指源文件中的音视频编码方式也被目标文件支持，那么此情况下音视频数据复制就可以直接复制到目标文件中，例如可将 MP4 文件中 H.264、AAC 格式的音视频码流直接复制到 FLV 文件中。容器格式的转换有两种情况，第 1 种情况是源容器格式的音视频编码方式在目标容器中也被支持，这样只需进行流复制；第 2 种情况是源容器格式的音视频编码格式在目标容器不被支持，那么就需要先解码再编码。例如将一个 FLV 格式封装的 H.264＋AAC 编码的音视频文件转码为 TS 格式封装的 MPEG-2(Video)＋MP2(Audio)编码的音视频文件，或者也可以解码后直接进行播放(注意音视频同步问题)，具体流程如图 5-2 所示。

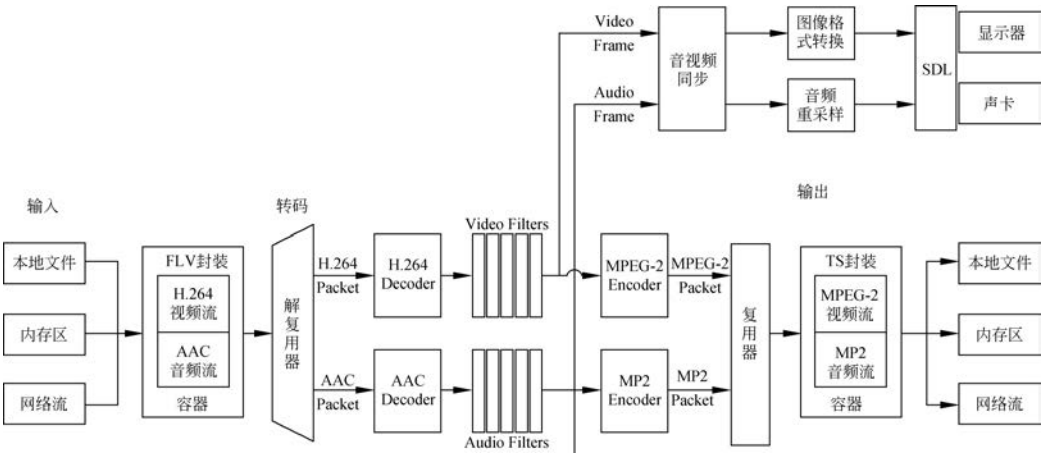

图 5-2 FFmpeg 将 FLV(H.264＋AAC)转码为 MPEG-TS

5.2 提取音视频的 YUV/PCM

音视频的基本信息包括帧率、码率、分辨率、声道数、采样格式、采样位数等，通过 FFmpeg 可以很方便地提取这些信息，也可以提取压缩视频流的 YUV 像素数据和压缩音频流的 PCM 数据。

5.2.1　利用 FFmpeg 提取视频的 YUV 像素数据

1. 提取 YUV420P

从输入的视频文件中提取前 2s 的视频数据，解码格式为 YUV420P，分辨率和源视频保持一致，转换过程如图 5-3 所示，命令如下：

```
ffmpeg -i test4.mp4 -t 2 -pix_fmt yuv420p -s 320x240 yuv420P_test4.yuv
#注意 -pix_fmt 用于指定像素格式
```

参数说明如下。

(1) -i：表示要输入的流媒体文件。

(2) -t：表示截取流媒体文件的长度，单位是秒。

(3) -pix_fmt：指定流媒体要转换的格式，这里是 YUV420P，具体格式可以通过命令 ffmpeg -pix_fmts 来查看。

(4) -s：指定分辨率大小(也可以写为-video_size)，例如可以写为 320x240。

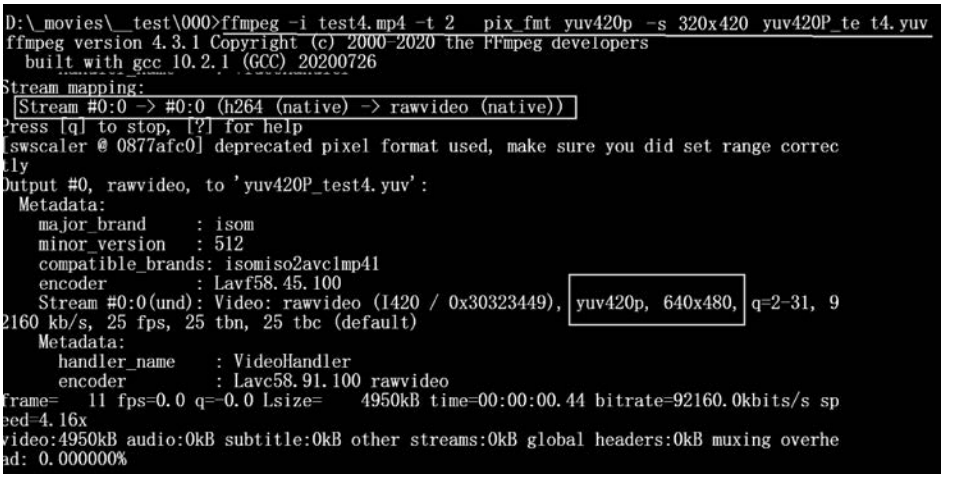

图 5-3　FFmpeg 提取 YUV420P

使用 YUVPlayer 软件打开这个 YUV 文件(yuv420P_test4.yuv)进行播放，可能会出现花屏，如图 5-4 所示，主要因为分辨率不匹配，单击 YUVPlayer 的 Size 下拉菜单项，选择对应的大小(笔者源视频的宽和高为 640×480，也可以单击 Custom 来输入自定义的宽和高)，这样便可以正常播放画面了，如图 5-5 所示。

2. 指定视频的宽和高

也可以转换成指定大小(例如写为 320x240)的 YUV 格式的裸视频，命令如下：

```
ffmpeg -i test4.mp4 -t 2 -pix_fmt yuv420p -s 320x240 yuv420p_test4_320x240.yuv
```

图 5-4　YUVPlayer 播放 YUV420P 文件出现花屏

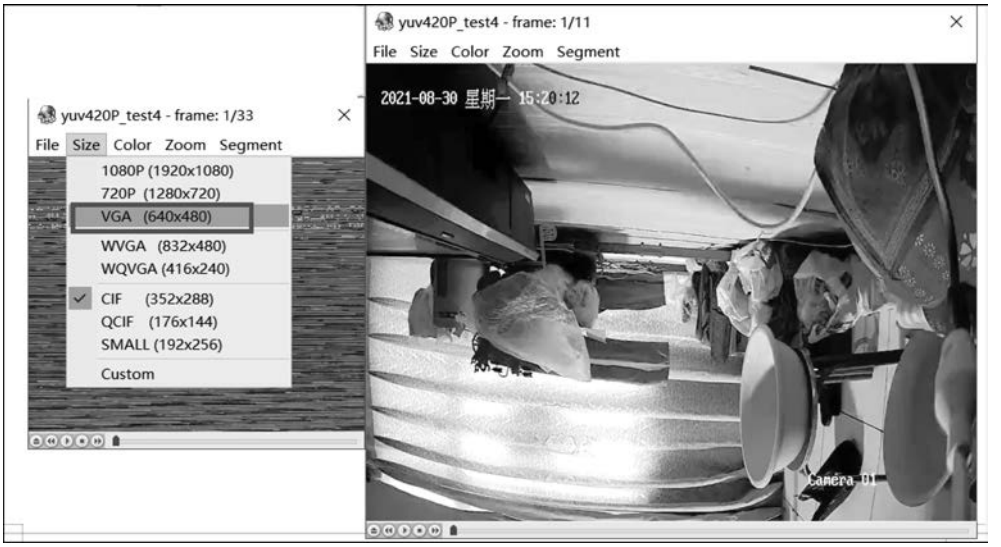

图 5-5　YUVPlayer 修改 Size 后画面正常

可以通过 YUVPlayer 播放，如图 5-6 所示。

3. ffplay 播放 YUV420P

也可以通过 ffplay 来播放 YUV 文件，但需要指定像素格式及宽和高，播放效果如图 5-7 所示，命令如下：

```
ffplay -i yuv420p_test4_320x240.yuv -pixel_format yuv420p -video_size 320x240
#或者:
ffplay -i yuv420p_test4_320x240.yuv -pixel_format yuv420p -s 320x240
```

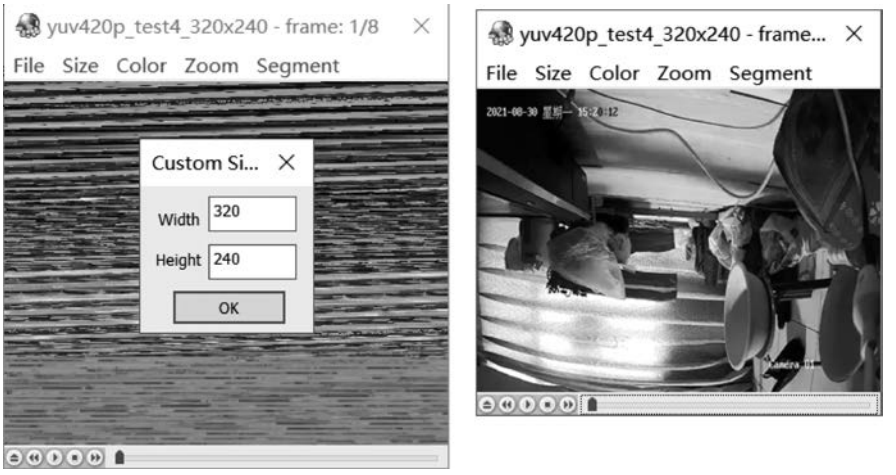

图 5-6　YUVPlayer 指定自定义的 Size

图 5-7　ffplay 播放 YUV420P

参数说明如下。

(1) -pixel_format：表示要输入的像素格式，例如 YUV420P。

(2) -s 或者-video_size：表示像素的宽和高，例如写作 320x240。

4. 提取 YUV422P

从输入的视频文件中提取前 2s 的视频数据，解码格式为 YUV422P，分辨率和源视频保持一致，转换过程如图 5-8 所示，命令如下：

```
ffmpeg -i test4.mp4 -t 2 -pix_fmt yuv422p yuv422p_test4.yuv
#注意 -pix_fmt 用于指定像素格式
```

使用 YUVPlayer 软件打开这个 YUV 文件(yuv422p_test4.yuv)进行播放，可能会出现花屏，如图 5-9 所示，主要因为分辨率(应该为 640×480)及像素格式(应该为 YUV422P)不匹配。

单击 YUVPlayer 的 Size 下拉菜单项，选择对应的大小(笔者源视频的宽和高为 640×480，也可以单击 Custom 来输入自定义的宽和高)，这样便可以播放画面了，但颜色有点失

```
D:\_movies\__test\000>ffmpeg -i test4.mp4 -t 2  -pix_fmt yuv422p yuv422p_test4.yuv
ffmpeg version 4.3.1 Copyright (c) 2000-2020 the FFmpeg developers
  Duration: 00:00:00.44, start: 0.000000, bitrate: 677 kb/s
    Stream #0:0(und): Video: h264 (High) (avc1 / 0x31637661), yuvj420p(pc), 640x480, 6
60 kb/s, 25 fps, 25 tbr, 12800 tbn, 50 tbc (default)
    Metadata:
      handler_name    : VideoHandler
Stream mapping:
  Stream #0:0 -> #0:0 (h264 (native) -> rawvideo (native))
Press [q] to stop, [?] for help
[swscaler @ 08000000] deprecated pixel format used, make sure you did set range correc
tly
Output #0, rawvideo, to 'yuv422p_test4.yuv':
  Metadata:
    major_brand     : isom
    minor_version   : 512
    compatible_brands: isomiso2avc1mp41
    encoder         : Lavf58.45.100
    Stream #0:0(und): Video: rawvideo (Y42B / 0x42323459), yuv422p, 640x480, q=2-31, 1
22880 kb/s, 25 fps, 25 tbn, 25 tbc (default)
    Metadata:
```

图 5-8　FFmpeg 提取 YUV422P

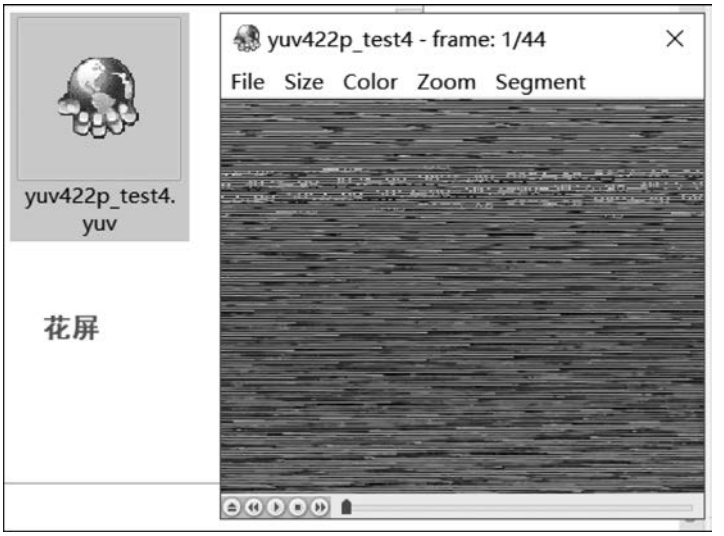

图 5-9　YUVPlayer 播放 YUV422P 文件出现花屏

真，如图 5-10 所示。

单击 YUVPlayer 的 Color 下拉菜单项，选择对应的像素格式 YUV422，然后就可以播放画面了，颜色也正常了，如图 5-11 所示。

5. 提取 YUV444P

从输入的视频文件中提取前 2s 的视频数据，解码格式为 YUV444P，分辨率和源视频保持一致，转换过程如图 5-12 所示，命令如下：

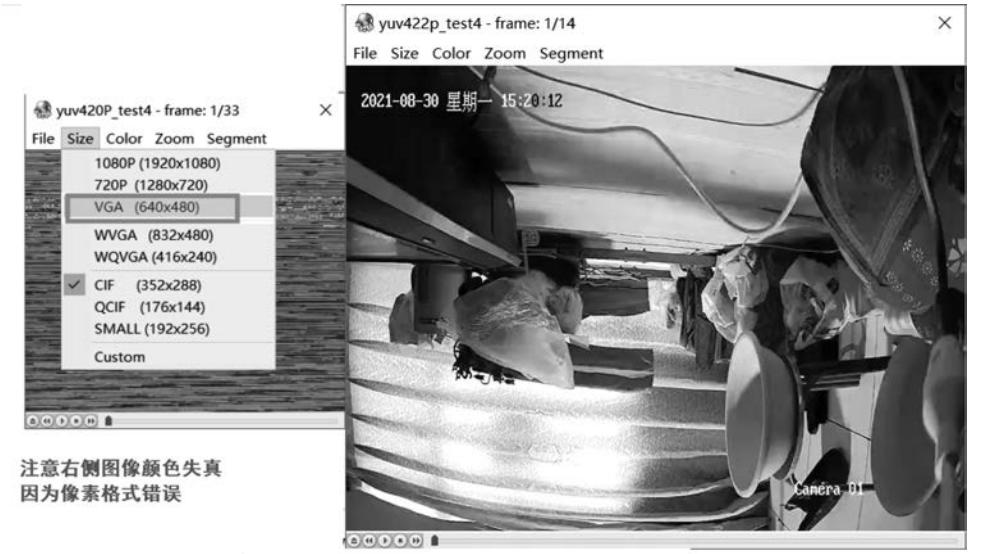


图 5-10 彩图

图 5-10　YUVPlayer 播放 YUV422P 时颜色失真

图 5-11　YUVPlayer 播放 YUV422P 的正常效果

```
ffmpeg -i test4.mp4 -t 2 -pix_fmt yuv444p yuv444p_test4.yuv
#注意-pix_fmt用于指定像素格式
```

使用 YUVPlayer 软件打开这个 YUV 文件(yuv444p_test4.yuv)进行播放，可能会出现花屏，主要因为分辨率(应该为 640×480)及像素格式(应该为 YUV444P)匹配不上。单击 YUVPlayer 的 Size 下拉菜单项，选择对应的大小(笔者源视频的宽和高为 640×480，也可以单击 Custom 来输入自定义的宽和高)，然后单击 Color 下拉菜单项，选择对应的像素格式 YUV444，这样就可以播放画面了，如图 5-13 所示。

```
D:\_movies\__test\000>ffmpeg -i test4.mp4 -t 2  -pix_fmt yuv444p yuv444p_test4.yuv
ffmpeg version 4.3.1 Copyright (c) 2000-2020 the FFmpeg developers
Stream mapping:
  Stream #0:0 -> #0:0 (h264 (native) -> rawvideo (native))
Press [q] to stop, [?] for help
[swscaler @ 08e0af80] deprecated pixel format used, make sure you did set range correc
tly
Output #0, rawvideo, to 'yuv444p_test4.yuv':
  Metadata:
    major_brand     : isom
    minor_version   : 512
    compatible_brands: isomiso2avc1mp41
    encoder         : Lavf58.45.100
    Stream #0:0(und): Video: rawvideo (444P / 0x50343434), yuv444p, 640x480, q=2-31, 1
84320 kb/s, 25 fps, 25 tbn, 25 tbc (default)
    Metadata:
      handler_name    : VideoHandler
      encoder         : Lavc58.91.100 rawvideo
frame=   11 fps=0.0 q=-0.0 Lsize=    9900kB time=00:00:00.44 bitrate=184320.0kbits/s s
peed=5.05x
```

图 5-12　FFmpeg 提取 YUV444P

图 5-13　YUVPlayer 播放 YUV444P 的正常效果

5.2.2　YUV444/YUV422/YUV420

YUV 本质上是一种颜色数字化表示方式。视频通信系统之所以要采用 YUV，而不是 RGB，主要因为 RGB 信号不利于压缩。在 YUV 这种方式里加入了亮度这一概念。视频工程师发现，眼睛对于亮和暗的分辨要比对颜色的分辨更精细一些，也就是说，人眼对色度的敏感程度要低于对亮度的敏感程度，所以在视频存储中，没有必要存储全部颜色信号，可以

把更多带宽留给黑白信号(亮度),将稍少的带宽留给彩色信号(色度),这就是 YUV 的基本原理,Y 是亮度,U 和 V 则是色度。YUV 的成像过程如图 5-14 所示。

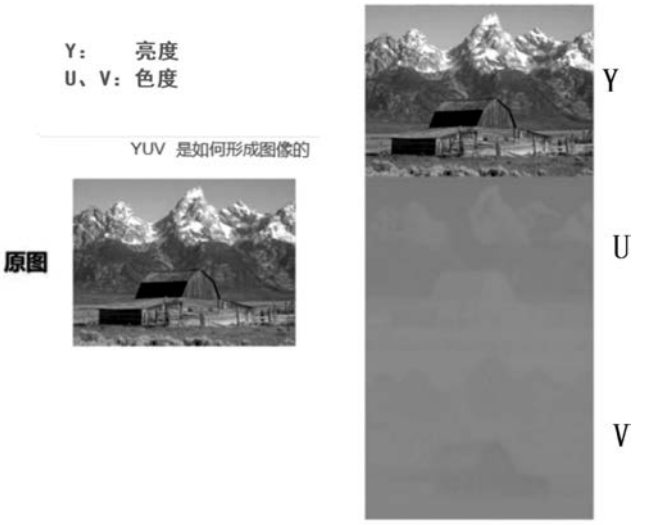

图 5-14　YUV 是如何形成图像的

根据亮度和色度分量的采样比率,YUV 图像通常有以下几种格式,如图 5-15 所示。

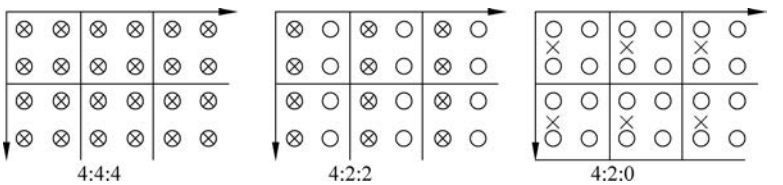

图 5-15　YUV 的几种采样格式

YUV 是视频、图片、相机等应用中经常使用的一类图像格式,是所有 YUV 像素格式共有的颜色空间的名称。与 RGB 格式不同,YUV 格式用一个称为 Y(相当于灰度)的“亮度”分量和两个“色度”分量表示,分别称为 U(蓝色投影)和 V(红色投影)。

Y 表示亮度分量,如果只显示 Y,图像看起来会是一张黑白照。U(Cb)表示色度分量,图像蓝色部分去掉亮度,反映了 RGB 输入信号蓝色部分与 RGB 信号亮度值之间的差异。V(Cr)表示色度分量,图像红色部分去掉亮度,反映了 RGB 输入信号红色部分与 RGB 信号亮度值之间的差异。

从前述定义可以知道,在 YUV 空间中像素颜色按“亮度”分量和两个“色度”分量进行了表示。这种编码表示也更加适应于人眼,据研究表明,人眼对亮度信息比色彩信息更加敏感,而 YUV 下采样就是根据人眼的特点,将人眼相对不敏感的色彩信息进行压缩采样,得到相对小的文件进行播放和传输。

1. YUV444

色度信号分辨率最高的格式是 YUV4:4:4，每 4 点 Y 采样，就有相对应的 4 点 U 和 4 点 V。换句话说，每个 Y 值对应一个 U 和一个 V 值。在这种格式中，色度信号的分辨率和亮度信号的分辨率是相同的。这种格式主要应用在视频处理设备内部，避免画面质量在处理过程中降低。

2. YUV422

色度信号分辨率格式 YUV4:2:2，每 4 点 Y 采样，就有相对应的 2 点 U 和 2 点 V。可以看到在水平方向上的色度表示进行了 2 倍下采样，因此 YUV422 色度信号分辨率是亮度信号分辨率的一半。

3. YUV420

色度信号分辨率格式 YUV4:2:0，每 4 点 Y 采样，就有相对应的 1 点 U 和 1 点 V。YUV420 色度信号分辨率是亮度信号分辨率的 1/4，即在水平方向压缩的基础上，再在垂直方向上进行了压缩。

4. YUYV

YUV4:4:4 采样，每个 Y 对应一组 UV 分量；YUV4:2:2 采样，每两个 Y 共用一组 UV 分量；YUV4:2:0 采样，每 4 个 Y 共用一组 UV 分量。下面用图的形式给出常见的 YUV 码流的存储方式，并在存储方式后面附有取样每像素的 YUV 数据的方法，其中，Cb、Cr 的含义等同于 U、V。

YUYV 为 YUV422 采样的存储格式中的一种，相邻的两个 Y 共用其相邻的两个 Cb、Cr，如图 5-16 所示。

start+0:	Y'_{00}	Cb_{00}	Y'_{01}	Cr_{00}	Y'_{02}	Cb_{01}	Y'_{03}	Cr_{01}
start+8:	Y'_{10}	Cb_{10}	Y'_{11}	Cr_{10}	Y'_{12}	Cb_{11}	Y'_{13}	Cr_{11}
start+16:	Y'_{20}	Cb_{20}	Y'_{21}	Cr_{20}	Y'_{22}	Cb_{21}	Y'_{23}	Cr_{21}
start+24:	Y'_{30}	Cb_{30}	Y'_{31}	Cr_{30}	Y'_{32}	Cb_{31}	Y'_{33}	Cr_{31}

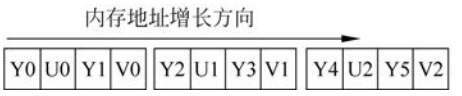

图 5-16　YUYV 的内存布局

5. UYVY

UYVY 格式也是 YUV422 采样的存储格式中的一种，只不过与 YUYV 不同的是 UV 的排列顺序不一样，还原其每像素的 YUV 值的方法与上面一样，如图 5-17 所示。

6. YUV422P

YUV422P 也属于 YUV422 的一种，它是一种 Plane 模式，即平面模式，并不是将 YUV 数据交错存储，而是先存放所有的 Y 分量，然后存储所有的 U(Cb)分量，最后存储所有的 V(Cr)分量，如图 5-18 所示。

start+0:	Cb_{00}	Y'_{00}	Cr_{00}	Y'_{01}	Cb_{01}	Y'_{02}	Cr_{01}	Y'_{03}
start+8:	Cb_{10}	Y'_{10}	Cr_{10}	Y'_{11}	Cb_{11}	Y'_{12}	Cr_{11}	Y'_{13}
start+16:	Cb_{20}	Y'_{20}	Cr_{20}	Y'_{21}	Cb_{21}	Y'_{22}	Cr_{21}	Y'_{23}
start+24:	Cb_{30}	Y'_{30}	Cr_{30}	Y'_{31}	Cb_{31}	Y'_{32}	Cr_{31}	Y'_{33}

图 5-17　UYVY 的内存布局

start+0:	Y'_{00}	Y'_{01}	Y'_{02}	Y'_{03}
start+4:	Y'_{10}	Y'_{11}	Y'_{12}	Y'_{13}
start+8:	Y'_{20}	Y'_{21}	Y'_{22}	Y'_{23}
start+12:	Y'_{30}	Y'_{31}	Y'_{32}	Y'_{33}
start+16:	Cb_{00}	Cb_{01}		
start+18:	Cb_{10}	Cb_{11}		
start+20:	Cb_{20}	Cb_{21}		
start+22:	Cb_{30}	Cb_{31}		
start+24:	Cr_{00}	Cr_{01}		
start+26:	Cr_{10}	Cr_{11}		
start+28:	Cr_{20}	Cr_{21}		
start+30:	Cr_{30}	Cr_{31}		

图 5-18　YUV422P 的内存布局

其每像素的 YUV 值的提取方法也是遵循 YUV422 格式的最基本提取方法，即两个 Y 共用一个 UV。例如，对于像素 Y'00、Y'01 而言，其 Cb、Cr 的值均为 Cb00、Cr00。

7. YV12/YU12

YU12(I420)和 YV12 属于 YUV420 格式，也是一种“三平面”(three-plane)模式，将 Y、U、V 分量分别打包，依次存储。其每像素的 YUV 数据提取遵循 YUV420 格式的提取方式，即 4 个 Y 分量共用一组 UV，如图 5-19 所示，Y'00、Y'01、Y'10、Y'11 共用 Cr00、Cb00，其他像素以此类推。

YU12，也称为 I420 或 YUV420P，先存储所有的 Y，然后存储所有的 U，最后存储所有的 V。YV12 先存储所有的 Y，然后存储所有的 V，最后存储所有的 U。伪代码如下：

```
//chapter5/5.1.txt
#YU12(I420):
yyyyyyyy yyyyyyyy yyyyyyyy yyyyyyyy yyyyyyyy yyyyyyyy yyyyyyyy yyyyyyyy     (w * h)
uuuuuuu uuuuuuu                                          (w * h/4)
vvvvvvv vvvvvvv                                          (w * h/4)

#YV12:
yyyyyyyy yyyyyyyy yyyyyyyy yyyyyyyy yyyyyyyy yyyyyyyy yyyyyyyy yyyyyyyy     (w * h)
vvvvvvv vvvvvvv                                          (w * h/4)
uuuuuuu uuuuuuu                                          (w * h/4)
```

start+0:	Y'_{00}	Y'_{01}	Y'_{02}	Y'_{03}
start+4:	Y'_{10}	Y'_{11}	Y'_{12}	Y'_{13}
start+8:	Y'_{20}	Y'_{21}	Y'_{22}	Y'_{23}
start+12:	Y'_{30}	Y'_{31}	Y'_{32}	Y'_{33}
start+16:	Cr_{00}	Cr_{01}		
start+18:	Cr_{10}	Cr_{11}		
start+20:	Cb_{00}	Cb_{01}		
start+22:	Cb_{10}	Cb_{11}		

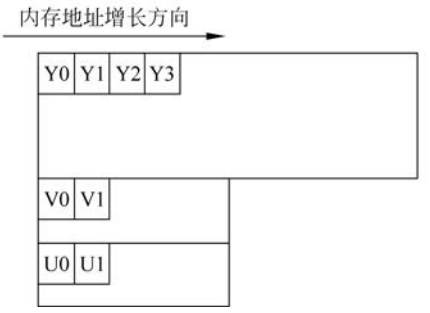

图 5-19　YU12/YV12 的内存布局

8. NV12/NV21

NV12 和 NV21 属于 YUV420 格式，是一种“两平面”(two-plane)模式，即 Y 和 UV 分为两个平面，但是 UV(CbCr)为交错存储，而不是分为 3 个平面，如图 5-20 所示。其提取方式与上一种类似，即 Y'00、Y'01、Y'10、Y'11 共用 Cr00、Cb00。

start+0:	Y'_{00}	Y'_{01}	Y'_{02}	Y'_{03}
start+4:	Y'_{10}	Y'_{11}	Y'_{12}	Y'_{13}
start+8:	Y'_{20}	Y'_{21}	Y'_{22}	Y'_{23}
start+12:	Y'_{30}	Y'_{31}	Y'_{32}	Y'_{33}
start+16:	Cb_{00}	Cr_{00}	Cb_{01}	Cr_{01}
start+20:	Cb_{10}	Cr_{10}	Cb_{11}	Cr_{11}

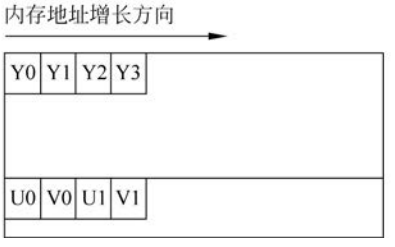

图 5-20　NV12/NV21 的内存布局

NV12 先存储 Y，然后 UV 交替，U 在前，V 在后。NV21 先存储 Y，然后 VU 交替，V 在前，U 在后。伪代码如下：

```
//chapter5/5.1.txt
#NV12: 先存储 Y,然后 UV 交替,U 在前,V 在后
YYYYYYYY YYYYYYYY YYYYYYYY YYYYYYYY YYYYYYYY YYYYYYYY YYYYYYYY YYYYYYYY
uvuvuvuv uvuvuvuv uvuvuvuv uvuvuvuv

#NV21: 先存储 Y,然后 VU 交替,V 在前,U 在后
YYYYYYYY YYYYYYYY YYYYYYYY YYYYYYYY YYYYYYYY YYYYYYYY YYYYYYYY YYYYYYYY
vuvuvuvu vuvuvuvu vuvuvuvu vuvuvuvu
```

9. YUV420SP 及 YUV420P

在 YUV420 中，一像素对应一个 Y，一个 2×2 的小方块对应一个 U 和 V。对于所有的 YUV420 图像，它们的 Y 值排列是完全相同的，只有 Y 的图像是灰度图像。

YUV420SP 与 YUV420P 数据格式的区别在于 UV 排列上的完全不同。YUV420P 是先把 U 存放完后，再存放 V，分为 3 个平面，Y、U、V 各占一个平面，而 YUV420SP 是按 UV、UV 的顺序交替存放的，分为两个平面，Y 占一个平面，UV 交织在一起占一个平面。根据此理论，就可以准确地计算出一个 YUV420 在内存中存放的大小，其中 Y＝width×height(Y 亮度点总数)，U＝Y÷4(U 色度点总数)，V＝Y÷4(V 色度点总数)，所以 YUV420 数据在内存中的大小是 width×height×3÷2 字节，例如一个分辨率为 8×4 的 YUV 图像，它们的格式如图 5-21 所示。

Y1	Y2	Y3	Y4	Y5	Y6	Y7	Y8
Y9	Y10	Y11	Y12	Y13	Y14	Y15	Y16
Y17	Y18	Y19	Y20	Y21	Y22	Y23	Y24
Y25	Y26	Y27	Y28	Y29	Y30	Y31	Y32
U1	V1	U2	V2	U3	V3	U4	V4
U5	V5	U6	V6	U7	V7	U8	V8

YUV420SP格式

Y1	Y2	Y3	Y4	Y5	Y6	Y7	Y8
Y9	Y10	Y11	Y12	Y13	Y14	Y15	Y16
Y17	Y18	Y19	Y20	Y21	Y22	Y23	Y24
Y25	Y26	Y27	Y28	Y29	Y30	Y31	Y32
U1	U2	U3	U4	U5	U6	U7	U8
V1	V2	V3	V4	V5	V6	V7	V8

YUV420P数据格式

图 5-21　YUV420SP 与 YUV420P 在内存中的分布情况

10. YUV 文件大小计算

以格式为 YUV420P、宽和高为 720×480 的图像为例，总大小为 720×480×3÷2 字节，分为 3 部分：

(1) Y 分量：720×480 字节。

(2) U(Cb)分量：720×480÷4 字节。

(3) V(Cr)分量：720×480÷4 字节。

这 3 部分内部均是行优先存储，它们之间按 Y、U、V 的顺序进行存储，如下所示。

(1) 0～720×480 字节是 Y 分量值。

(2) 720×480～720×480×5÷4 字节是 U 分量。

(3) 720×480×5÷4～720×480×3÷2 字节是 V 分量。

11. YV12 和 I420 的区别

一般来讲，如果采集到的视频数据是 RGB24 格式，RGB24 一帧的大小 size＝width×height×3 Bytes；RGB32 一帧的大小 size＝width×height×4 Bytes；YUV420 格式的数据量是 size＝width×height×1.5 Bytes。

当采集到 RGB24 数据后，需要对这个格式的数据进行第 1 次压缩，即将图像的颜色空间由 RGB 转换成 YUV。例如 libx264 在进行编码时需要输入的是标准 YUV420 格式，但是这里需要注意的是，虽然 YV12 也是 YUV420，但是 YV12 和 I420 却是不同的，在存储空间上有些区别，其区别如下：

(1) YV12：亮度(行×列)＋U(行×列÷4)＋V(行×列÷4)。

(2) I420：亮度(行×列)＋V(行×列÷4)＋U(行×列÷4)。

由此可以看出，YV12 和 I420 基本上是一样的，只是 UV 的顺序不同。经过第 1 次数据压缩后由 RGB24 转换成了 YUV420。这样，数据量就减少了一半，然后经过编码器的专业压缩，视频数据量就会减少很多。

注意：RGB24 格式的一像素占 24 位，即 3 字节；RGB32 格式的一像素占 32 位，即 4 字节。

5.2.3 利用 FFmpeg 提取视频的 RGB 像素数据

FFmpeg 除了可以提取 YUV 格式的像素数据，还可以提取 RGB 格式的像素数据。

1. 使用 FFmpeg 提取 RGB24 格式的像素数据

从输入的视频文件中提取前 2s 的视频数据，解码格式为 RGB24，分辨率和源视频保持一致，转换过程如图 5-22 所示，命令如下：

```
ffmpeg -i test4.mp4 -t 2 -pix_fmt rgb24 rgb24_test4.rgb
#注意-pix_fmt 用于指定像素格式
```

```
D:\_movies\__test\000>ffmpeg -i test4.mp4 -t 2 -pix_fmt rgb24 -y rgb24_test4.rgb
ffmpeg version 4.3.1 Copyright (c) 2000-2020 the FFmpeg developers
  Stream #0:0 -> #0:0 (h264 (native) -> rawvideo (native))
Press [q] to stop, [?] for help
[swscaler @ 0884d5c0] deprecated pixel format used, make sure you did set range correc
tly
Output #0, rawvideo, to 'rgb24_test4.rgb':
  Metadata:
    major_brand     : isom
    minor_version   : 512
    compatible_brands: isomiso2avc1mp41
    encoder         : Lavf58.45.100
    Stream #0:0(und): Video: rawvideo (RGB[24] / 0x18424752), rgb24, 640x480, q=2-31,
184320 kb/s, 25 fps, 25 tbn, 25 tbc (default)
    Metadata:
      handler_name    : VideoHandler
```

图 5-22 FFmpeg 提取 RGB24 格式的像素数据

2. 使用 YUVPlayer 播放 RGB24 文件

使用 YUVPlayer 软件打开这个 RGB24 文件(rgb24_test4.rgb)进行播放，可能会出现花屏，如图 5-23 所示，主要因为分辨率、颜色格式不匹配。

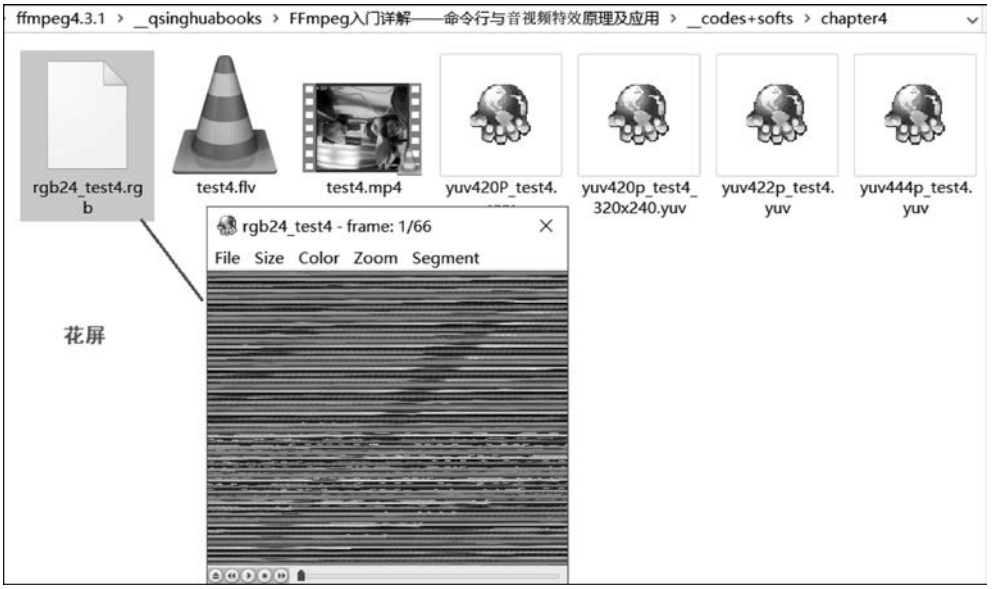

图 5-23　YUVPlayer 播放 RGB24 文件出现花屏

单击 YUVPlayer 的 Size 下拉菜单项，选择对应的大小（笔者源视频的宽和高为 640×480，也可以单击 Custom 来输入自定义的宽和高），然后单击 Color 下拉菜单，选择 RGB24，这样就可以正常播放画面了，如图 5-24 所示。

图 5-24　YUVPlayer 播放 RGB24 并选择正确的颜色格式

3. 使用 ffplay 播放 RGB24 文件

也可以通过 ffplay 播放 RGB24 文件，但需要指定像素格式及宽和高，播放效果如图 5-25 所示，命令如下：

```
ffplay -i rgb24_test4.rgb -pixel_format rgb24 -video_size 640x480
#注意：当 ffplay 播放 RGB、YUV 文件时，需要指定像素格式及视频的宽和高
```

图 5-25　ffplay 播放 RGB24 文件(需要指定宽和高及颜色格式)

4. 使用 ffmpeg 提取 RGB32 格式的像素数据

从输入的视频文件中提取前 2s 的视频数据，解码格式为 RGB32，分辨率和源视频保持一致，转换过程如图 5-26 所示，命令如下：

```
ffmpeg -i test4.mp4 -t 2 -pix_fmt rgb32 -y rgb32_test4.rgb
#注意 -pix_fmt 用于指定像素格式
```

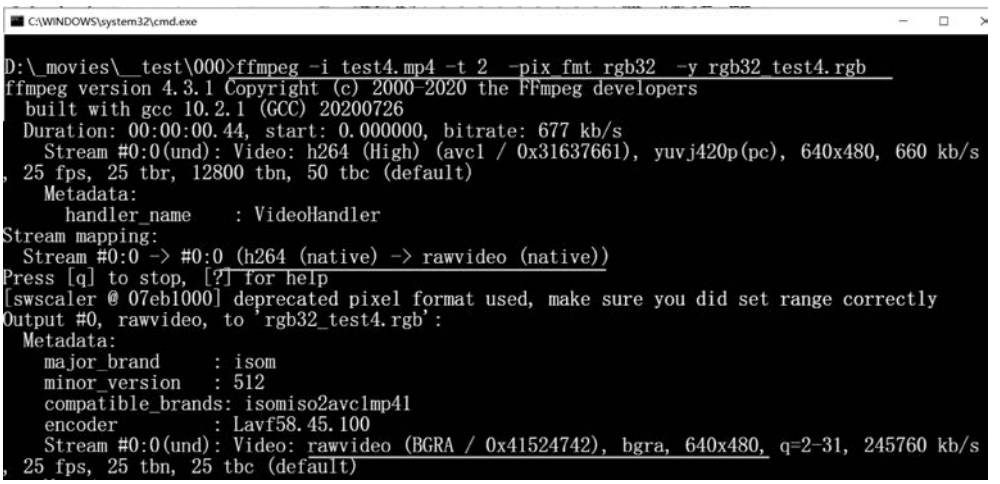

图 5-26　FFmpeg 提取 RGB32 格式的像素数据

注意：RGB32 格式的一像素占 32 位，即 4 字节。FFmpeg 转出来的 RGB32 的顺序为 BGRA，所以用 YUVPlayer 的 RGB32 播放时，颜色会失真，而用 ffplay 播放时指定为

RGB32，由于它的默认顺序是 BGRA，所以颜色正常。

5. 使用 YUVPlayer 播放 RGB32 文件

使用 YUVPlayer 软件打开这个 RGB32 文件(rgb32_test4.rgb)进行播放，可能会出现花屏，如图 5-27 所示，主要因为分辨率、颜色格式不匹配。

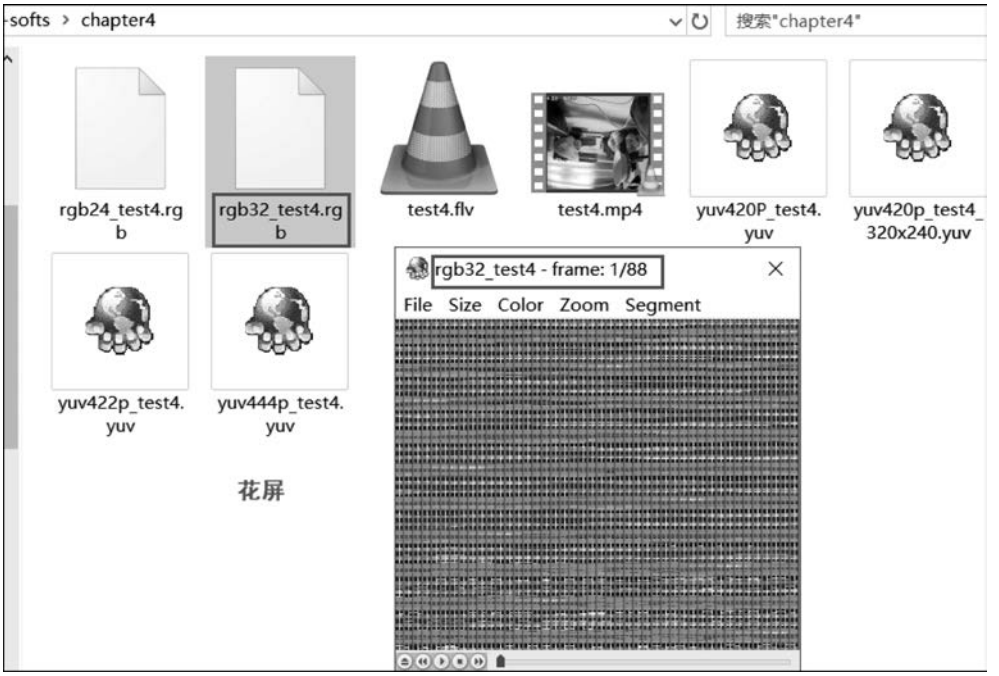

图 5-27　YUVPlayer 播放 RGB32 出现花屏

单击 YUVPlayer 的 Size 下拉菜单项，选择对应的大小(笔者源视频的宽和高为 640×480，也可以单击 Custom 来输入自定义的宽和高)，然后单击 Color 下拉菜单，选择 RGB32，这样就可以正常播放画面了，但是颜色失真(因为 FFmpeg 转换的 RGB32 的顺序为 BGRA)，如图 5-28 所示(蓝色的盆子显示成了黄色)。

6. 使用 ffplay 播放 RGB32 文件

也可以通过 ffplay 来播放 RGB32 文件，但需要指定像素格式及宽和高，播放效果如图 5-29 所示(颜色正常)，命令如下：

```
ffplay -i rgb32_test4.rgb -pixel_format rgb32 -video_size 640x480
#注意：ffplay播放RGB、YUV文件时，需要指定像素格式及视频的宽和高
#ffplay播放时指定为RGB32，由于它的默认顺序是BGRA，所以颜色正常
```

7. 使用 FFmpeg 提取 RGB16 格式的像素数据

从输入的视频文件中提取前 2s 的视频数据，解码格式为 RGB16。RGB16 又分为 RGB555 和 RGB565，这里默认为小端字节序。

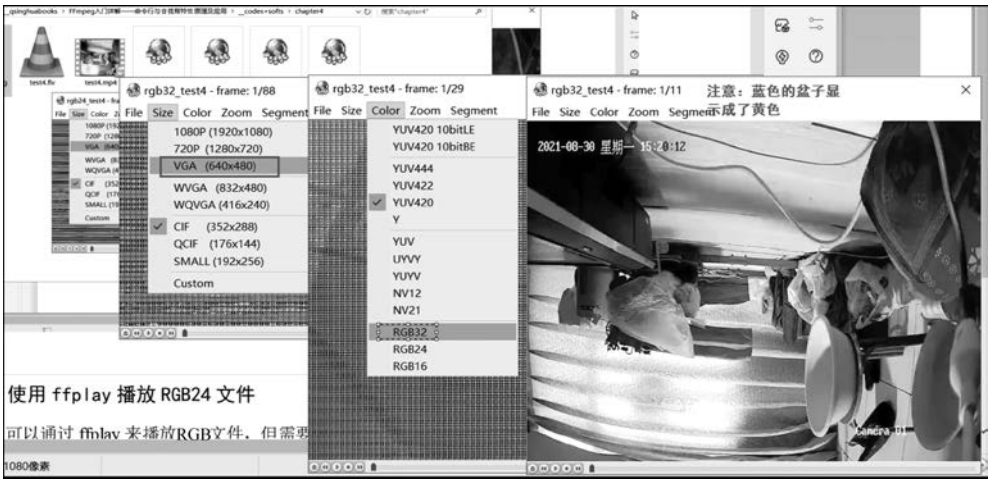

图 5-28　YUVPlayer 播放 RGB32(颜色失真)

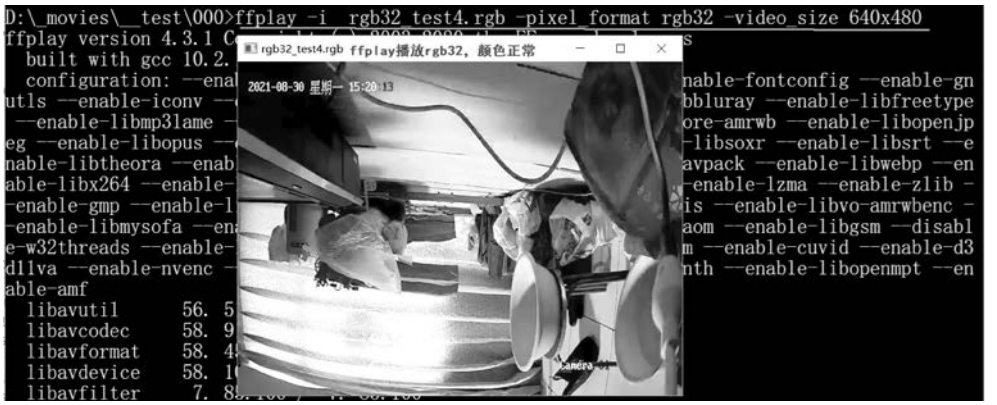

图 5-29　ffplay 播放 RGB32(指定宽和高及颜色格式)

先测试 RGB565,分辨率和源视频保持一致,转换过程如图 5-30 所示,命令如下:

```
ffmpeg -i test4.mp4 -t 2 -pix_fmt rgb_565le rgb565le_test4.rgb
#注意 - pix_fmt 用于指定像素格式: rgb_565le
```

再测试 RGB555,分辨率和源视频保持一致,转换过程如图 5-31 所示,命令如下:

```
ffmpeg -i test4.mp4 -t 2 -pix_fmt rgb_555le rgb555le_test4.rgb
#注意 - pix_fmt 用于指定像素格式: rgb_555le
```

注意:读者可以自己测试对应的大端字节序。

```
D:\_movies\__test\000>ffmpeg -i test4.mp4 -t 2  -pix_fmt rgb565le  rgb565le_test4.rgb
Stream mapping:
  Stream #0:0 -> #0:0 (h264 (native) -> rawvideo (native))
Press [q] to stop, [?] for help
[swscaler @ 07e8e000] deprecated pixel format used, make sure you did set range correctly
Output #0, rawvideo, to 'rgb565le_test4.rgb':
  Metadata:
    major_brand     : isom
    minor_version   : 512
    compatible_brands: isomiso2avc1mp41
    encoder         : Lavf58.45.100
    Stream #0:0(und): Video: rawvideo (RGB[16] / 0x10424752), rgb565le, 640x480, q=2-31, 1228
80 kb/s, 25 fps, 25 tbn, 25 tbc (default)
    Metadata:
      handler_name    : VideoHandler
      encoder         : Lavc58.91.100 rawvideo
```

图 5-30　FFmpeg 提取 RGB16(RGB565le)

```
D:\_movies\__test\000>ffmpeg -i test4.mp4 -t 2  -pix_fmt rgb555le  rgb555le_test4.rgb
ffmpeg version 4.3.1 Copyright (c) 2000-2020 the FFmpeg developers
Stream mapping:
  Stream #0:0 -> #0:0 (h264 (native) -> rawvideo (native))
Press [q] to stop, [?] for help
[swscaler @ 0807be00] deprecated pixel format used, make sure you did set range correctly
Output #0, rawvideo, to 'rgb555le_test4.rgb':
  Metadata:
    major_brand     : isom
    minor_version   : 512
    compatible_brands: isomiso2avc1mp41
    encoder         : Lavf58.45.100
    Stream #0:0(und): Video: rawvideo (RGB[15] / 0xF424752), rgb555le, 640x480, q=2-31, 11520
0 kb/s, 25 fps, 25 tbn, 25 tbc (default)
    Metadata:
      handler_name    : VideoHandler
      encoder         : Lavc58.91.100 rawvideo
frame=   11 fps=0.0 q=-0.0 Lsize=    6600kB time=00:00:00.44 bitrate=122880.0kbits/s speed=9.
94x
```

图 5-31　FFmpeg 提取 RGB16(RGB555le)

8. 使用 YUVPlayer 播放 RGB16 文件

使用 YUVPlayer 软件打开这个 RGB565le 文件(rgb565le_test4.rgb)进行播放，可能会出现花屏，如图 5-32 所示，主要因为分辨率、颜色格式不匹配。

单击 YUVPlayer 的 Size 下拉菜单项，选择对应的大小(笔者源视频的宽和高为 640×480，也可以单击 Custom 来输入自定义的宽和高)，然后单击 Color 下拉菜单，选择 RGB16，这样就可以正常播放画面了，颜色也正常(因为 YUVPlayer 的 RGB16 采取 RGB565 格式)，如图 5-33 所示。

使用 YUVPlayer 软件打开这个 RGB555le 文件(rgb555le_test4.rgb)进行播放，可能会出现花屏，然后修改 Size(640×480)和 Color(RGB16)，可能会出现花屏，并且颜色失真，如图 5-34 所示。

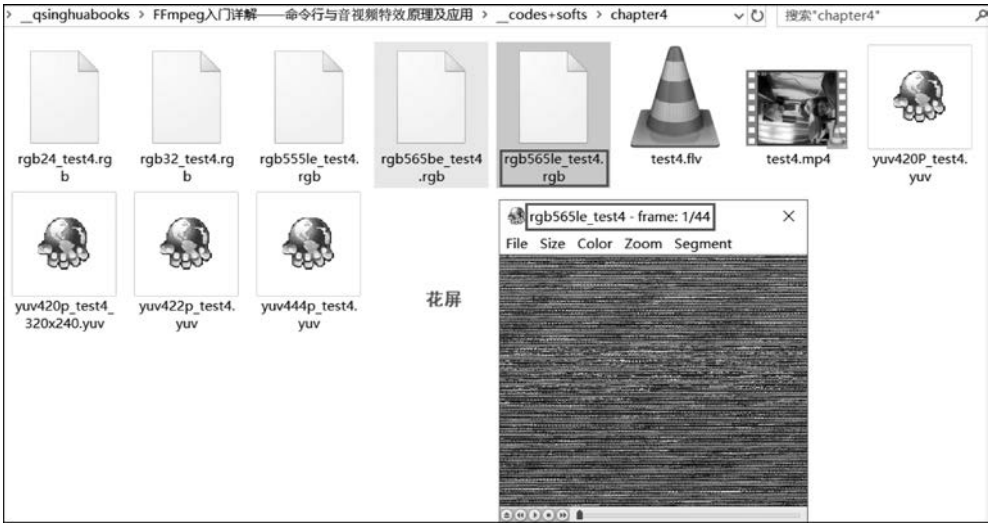

图 5-32　YUVPlayer 播放 RGB16(RGB565le)出现花屏

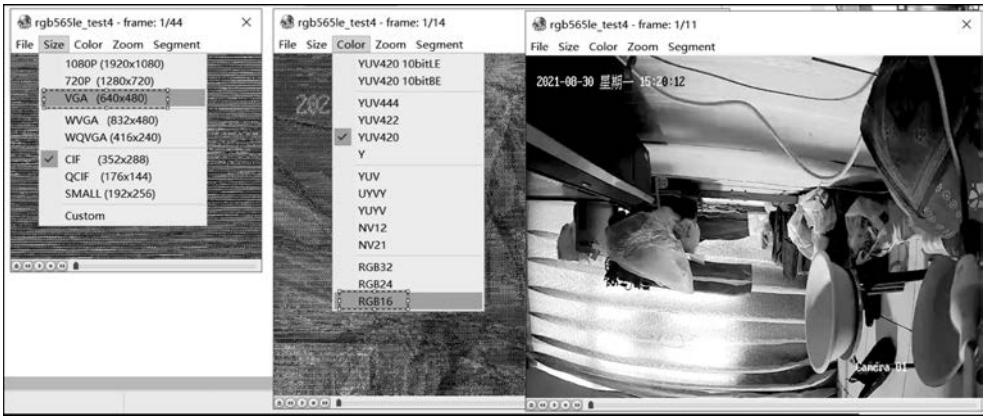

图 5-33　YUVPlayer 播放 RGB16(RGB565le)画面正常

9. 使用 ffplay 播放 RGB16 文件

也可以通过 ffplay 来播放 RGB16 文件(包括 RGB565 和 RGB555),但需要指定像素格式及宽和高,播放效果如图 5-35 和图 5-36 所示,命令如下:

```
//chapter5/5.1.txt
#注意: 当 ffplay 播放 RGB、YUV 文件时,需要指定像素格式及视频的宽和高: RGB565le
ffplay -i rgb565le_test4.rgb -pixel_format rgb565le -video_size 640x480

#注意: 当 ffplay 播放 RGB、YUV 文件时,需要指定像素格式及视频的宽和高: RGB555le
ffplay -i rgb555le_test4.rgb -pixel_format rgb555le -video_size 640x480
```

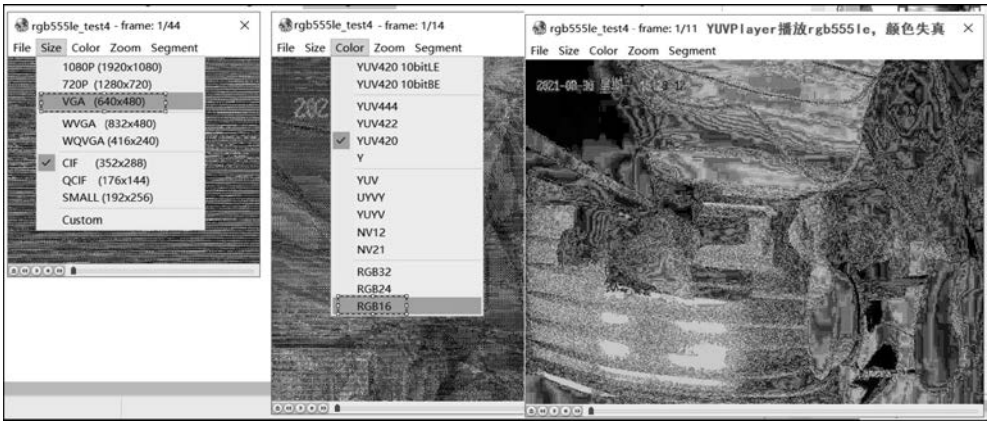

图 5-34　YUVPlayer 播放 RGB16(RGB555le)出现花屏

图 5-35　ffplay 播放 RGB16(RGB565le)

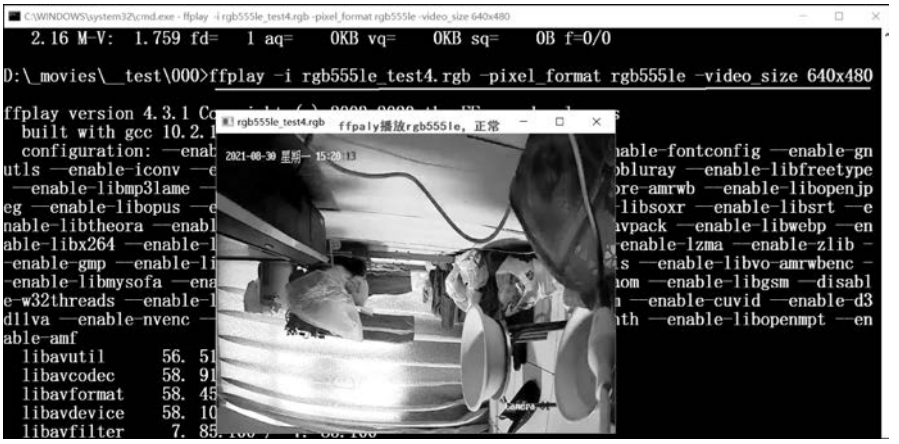

图 5-36　ffplay 播放 RGB16(RGB555le)


图 5-37 彩图

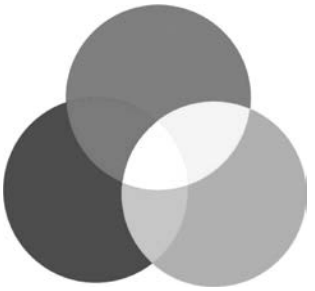
图 5-37　RGB 三基色

5.2.4　RGB16/RGB24/RGB32

RGB 指的是 R(Red)红色、G(Green)绿色、B(Blue)蓝色这 3 种颜色，如图 5-37 所示，所有的颜色都可以用这 3 种颜色配出来。通常所说的 RGB 是指 RGB24，其中 R、G、B 各有 256 级亮度，用数字表示为 0，1，2，…，255，最多 256 × 256 × 256 = 16 777 216，简称为 1600 万色、千万色、24 位色(2 的 24 次方)。

色彩深度是指每像素可以显示的颜色数，一般是用“位”(bit)为单位来描述。位数越多，可用的颜色就越多，图像的色彩表现就越准确，并且图像的文件大小也会随着位深的增加而增大，因为在高位深度的图像中，每像素存储了更多的颜色信息。

在进一步了解色彩深度之前，先来讨论一下什么是“位”(bit)。众所周知，计算机是以二进制的方式处理和存储信息的，因此任何信息输入计算机后都会变成 0 和 1 不同位数的组合，色彩也是如此。1 位代表一个二级制数据 0 或 1，8 个连续的位组成一个“字节”(Byte)，如图 5-38 所示。

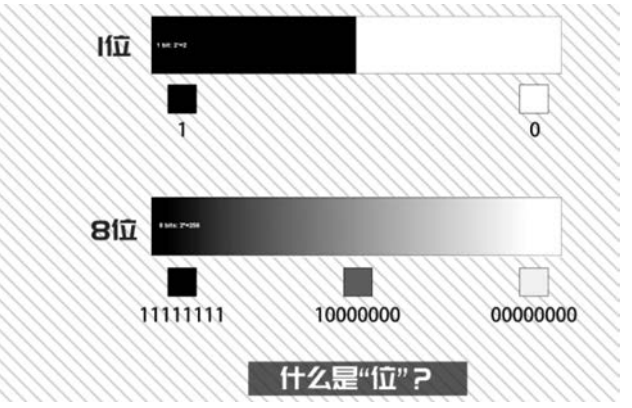

图 5-38　位(bit)与字节(Byte)

1 位的色彩深度，在计算机中只能显示 0 或 1，所以能够展现的色彩信息只有两种颜色，包括白色和黑色。将色彩深度提升到 2 位时，将出现 00、01、10、11，共 4(2^2)种组合方式，从而产生了相对简单的黑白灰关系。当色彩深度达到 3 位时，将带来 8(2^3)种不同的组合方式，包括 000、001、010、011、100、101、110 和 111，黑白灰的过渡将变得更加细致。由此可见，每一次位数的增加，组合方式都会带来指数级的上涨，如图 5-39 所示。

256 是一个非常熟悉的数值，在 PS 图像后期处理软件中频繁出现，也是 8 位色彩深度所能展现的最大色彩数量。说到这里，很多读者在心里肯定思考“难道照片只能显示出 256 种颜色?”。答案绝非如此，当打开 PS 图像菜单下的模式选择时就会明白，当前图片所展现的色彩深度绝非单纯的“8 位”，而是“8 位/每通道”。

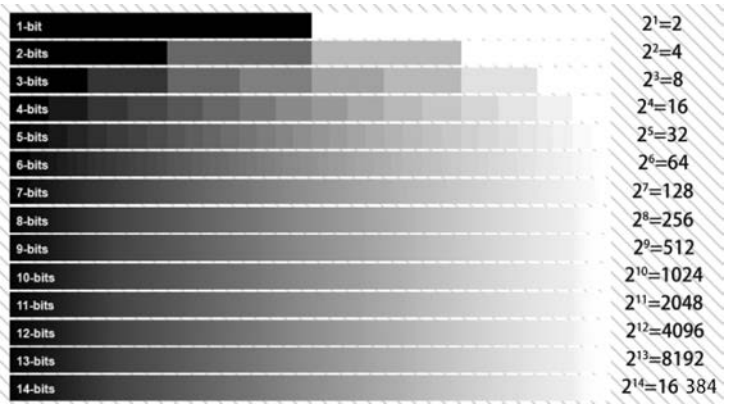

图 5-39　色彩深度

需要注意的是，上面的图像一直都是黑白图像，而彩色图像通常由不同强度的红(R)、绿(G)、蓝(B)3 种原色进行调和，以呈现不同的颜色。这些颜色在后期程序中用单独的“通道”进行处理，因此，“8 位/每通道”的 RGB 图像，意味着每像素共有 24 位的色彩深度(8 位为红色、8 位为绿色、8 位为蓝色)，实际上能够显示 16 777 216 个色彩值(256×256×256)，也就是相机拍摄 JPG 格式图像所能够呈现的色彩深度。

RGB-num：num 数字就是 num 位(num/8 字节)为一个存储单元，以此来存储一个 RGB 像素，RGB16/RGB24/RGB32 三者的排列顺序不同。

(1) RGB16，16 位为一个存储单元，以此来存储一个 RGB 像素；因为人眼对绿色比较敏感，所以有时会用 6 位绿色，有时会用 5 位，所以分为 RGB565 和 RGB555(注意大小端字节序)。RGB565 就是 R 占比 5 位、G 占比 6 位、B 占比 5 位，伪代码如下：

```
高字节                    低字节
RRRRRGGGGGGBBBBB
```

RGB555 就是 R 占比 5 位、G 占比 5 位、B 占比 5 位，伪代码如下：

```
高字节              低字节
空 RRRRRGGGGGBBBBB
```

(2) RGB24 将 RGB 分为 3 份，每一份占 8 位，即 R 占 8 位、G 占 8 位、B 占 8 位，伪代码如下：

```
高字节                    低字节
BBBBBBBBGGGGGGGGRRRRRRRR
```

(3) RGB32 与 RGB24 的排列方式一样，都是从高到低，从 B 到 R 排列，唯一不同是在低字节保留了 8 位(这 8 位也可以用来存储透明度)，伪代码如下：

```
高字节                                    低字节
BBBBBBBBGGGGGGGGRRRRRRRR空 空 空 空 空 空 空 空
```

下面介绍几个与图像相关的概念：

(1) 分辨率是指图像的精密度，例如手机分辨率(屏幕分辨率)是指手机所能显示的像素有多少；屏幕是由像数组成的，像素越多，画面就越精细。显示分辨率是指屏幕图像的精密度，是指显示器所能显示的像素有多少。图像分辨率是指单位英寸中所包含的像素数，其定义更趋近于分辨率本身的定义。像素是指在一张照片中采集了多少个点，可将像素视为图像中不可分割的单位或元素，像素的色彩由 RGB 通道决定。

(2) 图像深度又称为色深(Color Depth)，每像素保存在一个存储单元内，表示图像颜色的位数，例如 16、24、32 就是图像的深度。它确定了一张图像中最多能使用的颜色数，即彩色图像的每像素最大的颜色数。

(3) 像素深度是指存储每像素所用的位数，这些位数不仅包含表示颜色的位数，还可能包含表示图像属性的位数，因此像素深度大于或等于图像深度。

(4) 位深是指数字图像中像素的各通道的占用位数，即位深度的描述对象是通道而不是像素。

综上所述，图像深度是针对像素的 RGB 色彩占用位数描述的，不含像素的其他扩展属性位。像素深度是整像素占用的位数。位深度是指像素构成通道的占用位数，因此，RGB555 的每像素用 16 位表示，占 2 字节，如果 RGB 分量都使用 5 位(最高位不用)，则图像深度为 15 位、像素深度为 16 位、位深为 5 位。RGB24 的每像素用 24 位表示，占 3 字节，如果 RGB 分量都使用 8 位，则图像深度和像素深度都为 24 位、位深为 8 位。ARGB32 是带 alpha 通道的 RGB24，占 4 字节，RGB 分量都使用 8 位，则图像深度为 24 位、像素深度为 32 位、位深为 8 位。ARGB_4444 是指每像素用 16 位表示，占 2 字节，由 4 个 4 位组成，如果 ARGB 分量都是 4 位，则图像深度为 12 位、像素深度为 16 位、位深为 4 位。

5.2.5 利用 FFmpeg 提取音频的 PCM

利用 FFmpeg 还可以提取音频的 PCM 数据。

1. 使用 FFmpeg 提取 PCM 格式的音频数据

直接来看几个从音频文件中提取 PCM 的案例，代码如下：

```
//chapter5/5.1.txt
ffmpeg -i hello.mp3 -ar 48000 -ac 2 -f s16le 48000_2_s16le.pcm
ffmpeg -i hello.mp3 -ar 44100 -ac 2 -sample_fmt s16 44100_2_s16.wav
ffmpeg -i hello.mp3 -ar 44100 -ac 2 -codec: a pcm_s16le 44100_out2_s16le.wav
```

参数说明如下。

(1) -ar：表示采样率，包括 48 000、44 100、22 050 等。

(2) -ac：表示声道数。

(3) -f：表示输出格式。

这里指定了 3 种输出格式，包括 s16le、s16 和 pcm_s16le。这些格式可以通过命令行来查看，如图 5-40 和 5-41 所示，代码如下：

```
ffmpeg -encoders | findstr pcm
ffmpeg -sample_fmts
```

```
D:\_movies\__test\000>ffmpeg -encoders | findstr pcm
ffmpeg version 4.3.1 Copyright (c) 2000-2020 the FFmpeg developers
 A..... pcm_f32be            PCM 32-bit floating point big-endian
 A..... pcm_f32le            PCM 32-bit floating point little-endian
 A..... pcm_f64be            PCM 64-bit floating point big-endian
 A..... pcm_f64le            PCM 64-bit floating point little-endian
 A..... pcm_mulaw            PCM mu-law / G.711 mu-law
 A..... pcm_s16be            PCM signed 16-bit big-endian
 A..... pcm_s16be_planar     PCM signed 16-bit big-endian planar
 A..... pcm_s16le            PCM signed 16-bit little-endian
 A..... pcm_s16le_planar     PCM signed 16-bit little-endian planar
 A..... pcm_s24be            PCM signed 24-bit big-endian
 A..... pcm_s24daud          PCM D-Cinema audio signed 24-bit
 A..... pcm_s24le            PCM signed 24-bit little-endian
 A..... pcm_s24le_planar     PCM signed 24-bit little-endian planar
 A..... pcm_s32be            PCM signed 32-bit big-endian
 A..... pcm_s32le            PCM signed 32-bit little-endian
 A..... pcm_s32le_planar     PCM signed 32-bit little-endian planar
 A..... pcm_s64be            PCM signed 64-bit big-endian
 A..... pcm_s64le            PCM signed 64-bit little-endian
 A..... pcm_s8               PCM signed 8-bit
 A..... pcm_s8_planar        PCM signed 8-bit planar
 A..... pcm_u16be            PCM unsigned 16-bit big-endian
 A..... pcm_u16le            PCM unsigned 16-bit little-endian
 A..... pcm_u24be            PCM unsigned 24-bit big-endian
```

图 5-40　FFmpeg 支持的 PCM 编码格式

```
D:\_movies\__test\000>ffmpeg -sample_fmts
ffmpeg version 4.3.1 Copyright (c) 2000-2020
  built with gcc 10.2.1 (GCC) 20200726
name   depth
u8         8
s16       16
s32       32
flt       32
dbl       64
u8p        8
s16p      16
s32p      32
fltp      32
dblp      64
s64       64
s64p      64
```

图 5-41　FFmpeg 支持的采样格式

2. FFmpeg 支持的 PCM 格式

这些 PCM 格式对于音频重采样非常重要，输出信息如下：

```
//chapter5/ffmpeg - 431 - encoders - pcm.txt
A..... adpcm_adx          SEGA CRI ADX ADPCM
A..... g722               G.722 ADPCM(codec adpcm_g722)
A..... g726               G.726 ADPCM(codec adpcm_g726)
A..... g726le             G.726 little endian ADPCM("right - justified")(codec adpcm_g726le)
A..... adpcm_ima_qt       ADPCM IMA QuickTime
A..... adpcm_ima_ssi      ADPCM IMA Simon & Schuster Interactive
A..... adpcm_ima_wav      ADPCM IMA WAV
A..... adpcm_ms           ADPCM Microsoft
A..... adpcm_swf          ADPCM Shockwave Flash
A..... adpcm_yamaha       ADPCM Yamaha
A..... pcm_alaw           PCM A - law / G.711 A - law
A..... pcm_dvd            PCM signed 16|20|24 - bit big - endian for DVD media
A..... pcm_f32be          PCM 32 - bit floating point big - endian
A..... pcm_f32le          PCM 32 - bit floating point little - endian
A..... pcm_f64be          PCM 64 - bit floating point big - endian
A..... pcm_f64le          PCM 64 - bit floating point little - endian
A..... pcm_mulaw          PCM mu - law / G.711 mu - law
A..... pcm_s16be          PCM signed 16 - bit big - endian
A..... pcm_s16be_planar   PCM signed 16 - bit big - endian planar
A..... pcm_s16le          PCM signed 16 - bit little - endian
A..... pcm_s16le_planar   PCM signed 16 - bit little - endian planar
A..... pcm_s24be          PCM signed 24 - bit big - endian
A..... pcm_s24daud        PCM D - Cinema audio signed 24 - bit
A..... pcm_s24le          PCM signed 24 - bit little - endian
A..... pcm_s24le_planar   PCM signed 24 - bit little - endian planar
A..... pcm_s32be          PCM signed 32 - bit big - endian
A..... pcm_s32le          PCM signed 32 - bit little - endian
A..... pcm_s32le_planar   PCM signed 32 - bit little - endian planar
A..... pcm_s64be          PCM signed 64 - bit big - endian
A..... pcm_s64le          PCM signed 64 - bit little - endian
A..... pcm_s8             PCM signed 8 - bit
A..... pcm_s8_planar      PCM signed 8 - bit planar
A..... pcm_u16be          PCM unsigned 16 - bit big - endian
A..... pcm_u16le          PCM unsigned 16 - bit little - endian
A..... pcm_u24be          PCM unsigned 24 - bit big - endian
A..... pcm_u24le          PCM unsigned 24 - bit little - endian
A..... pcm_u32be          PCM unsigned 32 - bit big - endian
A..... pcm_u32le          PCM unsigned 32 - bit little - endian
A..... pcm_u8             PCM unsigned 8 - bit
A..... pcm_vidc           PCM Archimedes VIDC
A..... roq_dpcm           id RoQ DPCM
```

3. 使用 ffplay 播放 PCM 数据

使用 ffplay 播放 PCM 数据，可以播放出声音，还可以看到音频波形图，如图 5-42 所示，命令如下：

```
ffplay -ar 48000 -ac 2 -f s16le 48000_2_s16le.pcm
```

参数说明如下。

(1) -ar：表示采样率，包括 48 000、44 100、22 050 等。

(2) -ac：表示声道数。

(3) -f：表示采样格式。

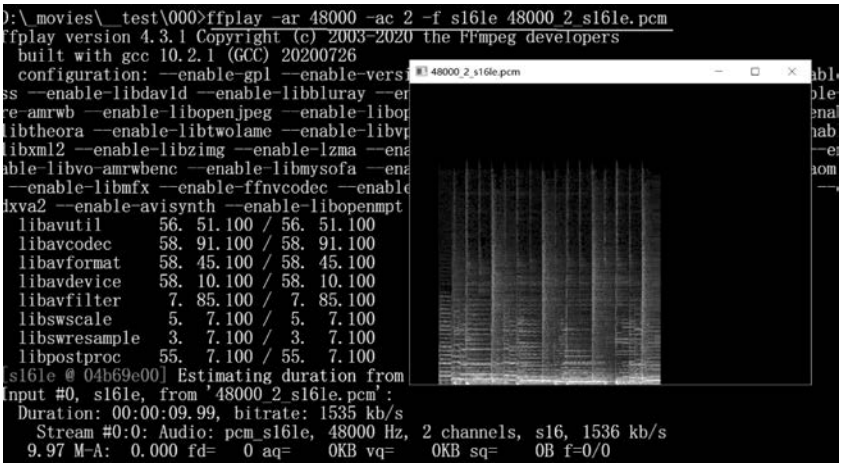

图 5-42　ffplay 播放 PCM 数据(需要指定采样率、声道数、采样格式)

4. 使用 ffplay 播放 WAV 文件

使用 ffplay 播放 WAV 文件，可以播放出声音，还可以看到音频波形图，如图 5-43 所示，命令如下：

```
ffplay -i 44100_2_s16.wav
```

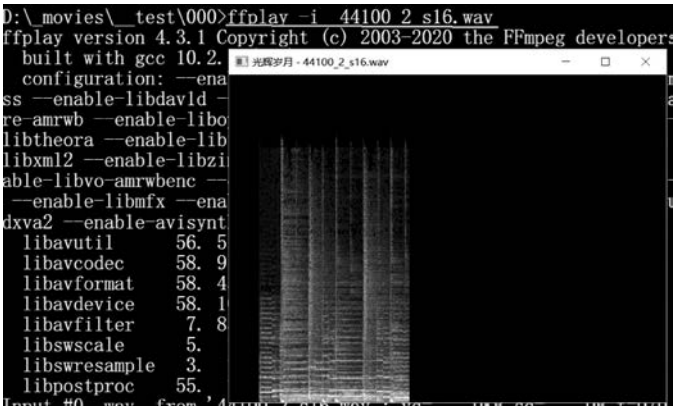

图 5-43　ffplay 播放 WAV 文件

注意：WAV 文件有头文件信息，包括音频的采样率、采样格式、声道数等，所以常见的播放器（如 ffplay、VLC 等）可以直接播放出来。

5.2.6 PCM 数据与 WAV 格式

想要了解音频首先要了解它的构造，知道它怎么从声音变成文件，又怎么从文件变成声音。文件格式根据需求和技术的进步有了不同的版本，不同的文件格式有其不同的文件构造。首先从最原始的音频格式 PCM 入手，然后介绍 WAV。

1. PCM 格式简介

PCM(Pulse-Code Modulation)，即脉冲编码调制，是音频的原始数据，是采样器（如话筒）电信号转化的数字信号，这就是常说的采样到量化的过程，所以其实 PCM 不仅可以用在音频录制方面，还可以用在其他电信号转数字信号的所有场景。由这样一段原始数据组成的音频文件叫作 PCM 文件，通常以.PCM 结尾。一个 PCM 文件的大小取决于以下几个元素。

(1) 采样率：指每秒电信号采集数据的频率，常见的音频采样率有 8000Hz、16 000Hz、44 000Hz、48 000Hz、96 000Hz 等。

(2) 采样位深：表示每个电信号用多少位来存储，例如 8 位的采样位深能够划分的等级位共 256 份，人耳可识别的声音频率为 20～20 000Hz，那么每个位的误差就达到了 80Hz，这对音频的还原度大幅降低，但是它的大小也相应减小了，更有利于音频传输。早期的电话就使用了比较低的采样率来达到更稳定的通话质量。对于采样位深其实并没有 8 位、16 位、32 位这么简单，对于计算机来讲 16 位既可以用 short 表示，也可以用 16 位 int 表示；32 位既可以用 32 位 int 表示，也可以用 32 位 float 表示；除此之外还有有符号和无符号之分，所以在编解码时需要注意这些具体的格式。

(3) 采样通道：常见的有单通道和双通道，双通道能区分左右耳的声音，单通道不区分左右耳的声音，两只耳朵所听到的都是一样的声音。通常在追求立体感时会使用双通道，所以双通道采集的声音也叫立体声。除此之外还有要求更高的 2.1、5.1、6.1、7.1 等通道类型，这些规格对录制音频筒有一定要求。

(4) 数据存储方式：表明数据是以交叉方式存放还是以分通道的方式存放，交叉排列只针对多通道的音频文件，单通道的音频文件不存在交叉排列。采样通道和数据存储方式决定了数据具体如何存储。如果是单声道的文件，则采样数据按时间的先后顺序依次存入。如果是双声道的文件，则通常按照 LRLRLR 的方式存储，存储时还和机器的大小端有关。例如 PCM 的存储方式为小端模式，存储数据排列如图 5-44 所示。

描述 PCM 音频数据的参数时有以下描述方式：

```
//chapter5/pcm - info - help.txt
44100Hz 16b stereo: 每秒有 44100 次采样,采样数据用 16 位(2 字节)记录,双声道(立体声)
22050Hz 8b  mono: 每秒有 22050 次采样,采样数据用 8 位(1 字节)记录,单声道
48000Hz 32b 51ch: 每秒有 48000 次采样,采样数据用 32 位(4 字节浮点型)记录,5.1 声道
```

图 5-44　PCM 声道数据的存储顺序

(1) 44 100Hz 指的是采样率，意思是每秒采样 44 100 次。采样率越大，存储数字音频所占的空间就越大。

(2) 16b 指的是采样精度，意思是原始模拟信号被采样后，每个采样点在计算机中用 16 位(两字节)来表示。采样精度越高越能精细地表示模拟信号的差异。

(3) stereo 指的是双声道，即采样时用到话筒的数量，话筒越多就越能还原真实的采样环境(当然话筒的放置位置也是有规定的)。

2. WAV 格式简介

WAV 是微软公司开发的一种声音文件格式，也叫波形声音文件，是最早的数字音频格式之一，被 Windows 平台及其应用程序广泛支持，但压缩率比较低。WAV 编码是在 PCM 数据格式的前面加上 44 字节的头部，分别用来描述 PCM 的采样率、声道数、数据格式等信息。特点是音质非常好、大量软件都支持。一般应用在多媒体开发的中间文件中，用于保存音乐和音效素材等。该格式的文件能记录各种单声道或立体声的声音信息，并能保证声音不失真，但 WAV 文件有一个致命的缺点，就是它所占用的磁盘空间相对来讲很大(每分钟的音乐大约需要 12MB 磁盘空间)。它符合资源互换文件格式(RIFF)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广泛支持。一般来讲，由 WAV 文件还原而成的声音的音质取决于声卡采样样本的尺寸，采样频率越高，音质就越好，但开销就越大，WAV 文件也就越大。

1) WAV 简介

WAV 文件是 Windows 标准的文件格式，WAV 文件为多媒体中使用的声音文件格式之一，它是以 RIFF 格式为标准的。RIFF 是英文 Resource Interchange File Format 的缩写，每个 WAV 文件的前 4 字节便是 RIFF。WAV 文件由文件头和数据体两部分组成，其中文件头又分为 RIFF/WAV 文件标识段和声音数据格式说明段两部分。常见的声音文件主要有两种，分别对应于单声道(11.025kHz 采样率、8b 的采样值)和双声道(44.1kHz 采样率、16b 的采样值)。采样率是指声音信号在“模→数”转换过程中单位时间内采样的次数。采样值是指每一采样周期内声音模拟信号的积分值。对于单声道声音文件，采样数据为 8b

的短整数(00H～FFH),而对于双声道立体声声音文件,每次采样数据为一个 16b 的整数(int),高 8 位和低 8 位分别代表左右两个声道。WAV 文件数据块包含以 PCM(脉冲编码调制)格式表示的样本。WAV 文件是由样本组织而成的。在多声道 WAV 文件中,样本是交替出现的。

RIFF 是一种按照标记区块存储数据的通用文件存储格式,多用于存储音频、视频等多媒体数据。Microsoft 在 Windows 下的 WAV、AVI 等格式都是基于 RIFF 实现的。一个标准的 RIFF 规范文件的最小存储单位为“块”(Chunk),每个 Chunk 包含以下三部分信息,如表 5-1 所示。

表 5-1 RIFF 格式说明

名　　称	大小/B	类型	字节序	内　　容
FOURCC	4	字符	大端	用于标识 Chunk ID 或 Chunk 类型,通常为 Chunk ID
Data Field Size	4	整数	小端	特别注意,该长度不包含其本身,以及 FOURCC
Data Field	—	—	—	数据域,如果 Chunk ID 为"RIFF"或"LIST",则开始的 4 字节为类型码

只有 ID 为 RIFF 或者 LIST 的块允许拥有子块(SubChunk)。RIFF 文件的第 1 个块的 ID 必须是 RIFF,也就是说 ID 为 LIST 的块只能是子块,它们和各个子块形成了复杂的 RIFF 文件结构。RIFF 数据域的起始位置的 4 字节为类型码(Form Type),用于说明数据域的格式,例如 WAV 文件的类型码为 WAVE。LIST 块的数据域的起始位置也有一个 4 字节类型码,用于说明 LIST 数据域的数据内容。例如,类型码为 INFO 时,其数据域可能包括 ICOP、ICRD 块,用于记录文件版权和创建时间信息。

2) WAV 头结构

WAV 头共 44 字节,标准结构体的代码如下:

```
//chapter5/wav - header.txt
/* RIFF WAVE file struct. : : RIFF WAV 头结构体
 * For details see WAVE file format documentation
 * (for example at <a href = "http://www.wotsit.org)." target = "_blank"> http://www.wotsit.
org).</a> */
typedef struct WAV_HEADER_S
{
    char            riffType[4];        //4Byte,资源交换文件标志: RIFF
    unsigned int    riffSize;           //4Byte,从下个地址到文件结尾的总字节数
    char            waveType[4];        //4Byte,WAV 文件标志: WAVE
    char            formatType[4];      //4Byte,波形文件标志: FMT(最后一位空格符)
    unsigned int    formatSize;         //4Byte,音频属性
```

```
//(compressionCode,numChannels,sampleRate,BytesPerSecond,blockAlign,bitsPerSample)所占字
//节数
    unsigned short   compressionCode;   //2Byte,格式种类(1-线性
                                        //pcm-WAVE_FORMAT_PCM,WAVEFORMAT_ADPCM)
    unsigned short   numChannels;       //2Byte,通道数
    unsigned int     sampleRate;        //4Byte,采样率
    unsigned int     BytesPerSecond;    //4Byte,传输速率
    unsigned short   blockAlign;        //2Byte,数据块的对齐,即DATA数据块长度
    unsigned short   bitsPerSample;     //2Byte,采样精度-PCM位宽
    char             dataType[4];       //4Byte,数据标志:data
    unsigned int     dataSize;          //4Byte,从下个地址到文件结尾的总字节数,即除了WAV
                                        //Header以外的PCM Data Length
}WAV_HEADER;
```

WAV头字段格式说明如表5-2所示。

表5-2　WAV头字段格式说明

偏移地址	大小/B	类型	字节序	内　　容
00H～03H	4	字符	大端	"RIFF"块(0x52494646),标记为RIFF文件格式
04H～07H	4	长整数	小端	块数据域大小(Chunk Size),即从下一个地址开始到文件末尾的总字节数,或者文件总字节数减8。从0x08开始一直到文件末尾都是ID为"RIFF"块的内容,其中包含两个子块,即"fmt"和"data"
08H～0BH	4	字符	大端	类型码(Form Type),WAV文件格式标记,即"WAVE"四个字母
0CH～0FH	4	字符	大端	"fmt"子块(0x666D7420),注意末尾的空格
10H～13H	4	整数	小端	子块数据域大小(SubChunk Size)
14H～15H	2	整数	小端	编码格式(Audio Format),1代表PCM无损格式,表示数据为线性PCM编码
16H～17H	2	整数	小端	通道数,单声道为1,双声道为2
18H～1BH	4	长整数	小端	采样频率
1CH～1FH	4	长整数	小端	传输速率(Byte Rate),每秒数据字节数,SampleRate * Channels * BitsPerSample/8
20H～21H	2	整数	小端	每个采样所需的字节数 BlockAlign,BitsPerSample * Channels/8
22H～23H	2	整数	小端	单个采样位深(Bits Per Sample),可选8、16或32
24H～27H	4	字符	大端	"data"子块(0x64617461)
28H～2BH	4	长整数	小端	子块数据域大小(SubChunk Size)
0x2C～EOS	—	—	—	PCM具体数据

注意：H结尾表示的是十六进制的数据,如10H表示十进制的16,20H表示十进制的32等。

定义 WAV_HEADER 类型的结构体变量 wavHeader(头部含有 44 字节的标志信息)，各个字段的含义如下：

```
//chapter5/wav - header.txt

   //ckid: 4 字节 RIFF 标志,大写
    wavHeader[0] = 'R';
    wavHeader[1] = 'I';
    wavHeader[2] = 'F';
    wavHeader[3] = 'F';


    //cksize: 4 字节文件长度,这个长度不包括"RIFF"标志(4 字节)和文件长度本身所占字节(4 字
    //节),即该长度等于整个文件长度 - 8
    wavHeader[4] = (Byte)(totalDataLen & 0xff); //totalDataLen: 数据长度
    wavHeader[5] = (Byte)((totalDataLen >> 8) & 0xff);
    wavHeader[6] = (Byte)((totalDataLen >> 16) & 0xff);
    wavHeader[7] = (Byte)((totalDataLen >> 24) & 0xff);


    //FOURCC Type: 4 字节 "WAVE"类型块标识,大写
    wavHeader[8] = 'W';
    wavHeader[9] = 'A';
    wavHeader[10] = 'V';
    wavHeader[11] = 'E';


    //ckid: 4 字节,表示"fmt"Chunk 的开始,此块中包括文件内部格式信息,小写,最后一个字符是
    //空格
    wavHeader[12] = 'f';
    wavHeader[13] = 'm';
    wavHeader[14] = 't';
    wavHeader[15] = ' ';


  //cksize: 4 字节,文件内部格式信息数据的大小,过滤字节(一般为 00000010H)
    wavHeader[16] = 0x10;
    wavHeader[17] = 0;
    wavHeader[18] = 0;
    wavHeader[19] = 0;


    //FormatTag: 2 字节,音频数据的编码方式,1: 表示是 PCM 编码
    wavHeader[20] = 1;
    wavHeader[21] = 0;
```

```
//Channels: 2 字节,声道数,单声道为 1,双声道为 2
wavHeader[22] = (Byte) channels;            //channels: 声道数
wavHeader[23] = 0;


//SamplesPerSec: 4 字节,采样率,如 44 100
wavHeader[24] = (Byte)(sampleRate & 0xff);  //sampleRate : 采样率
wavHeader[25] = (Byte)((sampleRate >> 8) & 0xff);
wavHeader[26] = (Byte)((sampleRate >> 16) & 0xff);
wavHeader[27] = (Byte)((sampleRate >> 24) & 0xff);



//BytesPerSec: 4 字节,音频数据传送速率,单位是字节
//其值为采样率 × 每次采样大小.播放软件利用此值可以估计缓冲区的大小
//BytePerSecond = sampleRate * (bitsPerSample / 8) * channels
wavHeader[28] = (Byte)(BytePerSecond & 0xff);
wavHeader[29] = (Byte)((BytePerSecond >> 8) & 0xff);
wavHeader[30] = (Byte)((BytePerSecond >> 16) & 0xff);
wavHeader[31] = (Byte)((BytePerSecond >> 24) & 0xff);

//BlockAlign: 2 字节,每次采样的大小 = 采样精度 * 声道数/8(单位是字节); 这也是字节对齐
//的最小单位,譬如 16 位立体声,在这里的值是 4 字节.
//播放软件需要一次处理多个该值大小的字节数据,以便将其值用于缓冲区的调整
wavHeader[32] = (Byte)(bitsPerSample * channels / 8);
wavHeader[33] = 0;


//BitsPerSample: 2 字节,每个声道的采样精度; 譬如 16 位,在这里的值就是 16.如果有多个声
//道,则每个声道的采样精度大小都一样
wavHeader[34] = (Byte) bitsPerSample;
wavHeader[35] = 0;


//ckid: 4 字节,数据标志符(data),表示 "data"Chunk 的开始.此块中包含音频数据,小写
wavHeader[36] = 'd';
wavHeader[37] = 'a';
wavHeader[38] = 't';
wavHeader[39] = 'a';


//cksize: 音频数据的长度,4 字节,audioDataLen = totalDataLen - 36 = fileLenIncludeHeader - 44
wavHeader[40] = (Byte)(audioDataLen & 0xff);
```

```
    wavHeader[41] = (Byte)((audioDataLen >> 8) & 0xff);
    wavHeader[42] = (Byte)((audioDataLen >> 16) & 0xff);
    wavHeader[43] = (Byte)((audioDataLen >> 24) & 0xff);
```

3) WAV 扩展

WAV 扩展是指有一些 WAV 的头部并不只有 44 字节，例如通过 FFmpge 编码而来的 WAV 文件的头部信息通常大于 44 字节。这是因为根据 WAV 规范，其头部还支持携带附加信息，所以只按照 44 字节的长度去解析 WAV 头部信息不一定正确，还需要考虑附加信息。可以根据"fmt"子块长度来判断一个 WAV 文件头部是否包含附加信息。如果 fmt SubChunk Size 等于 0x10(十进制的 16)，则表示头部不包含附加信息，即 WAV 头部信息的长度为 44；如果等于 0x12(十进制的 18)，则包含附加信息，此时头部信息的长度大于 44。当 WAV 头部包含附加信息时，fmt SubChunk Size 的长度为 18，并且紧随着另一个子块，这个子块包含了一些自定义的附加信息，接着才是"data"子块。

如果一个无损 WAV 文件头部包含了附加信息，则 PCM 音频所在的位置就不确定了，但由于附加信息也是一个子块(SubChunk)，根据 RIFF 规范，该子块也必然记录着其长度信息，所以有办法动态地计算出其位置，下面是计算步骤：

(1) 判断 fmt 块的长度是否为 18。

(2) 如果 fmt 块的长度为 18，则从 0x26 位置开始为附加信息块，0x30～0x33 位置记录着该子块长度。

(3) 根据步骤(2)获取的子块长度，假定为 N(十六进制)，则 PCM 音频信息的开始位置为 0x34＋N＋8。

4) WAV 头文件案例解析

新建头文件，文件名为 wav.h，定义两个结构体 WAV_HEADER 和 WAV_INFO，代码如下：

```
//chapter5/wav.h
#ifndef __WAV_H__
#define __WAV_H__

#define DeBug(fmt...) do \
            { \
                printf("[%s: : %d] ",__func__,__LINE__); \
                printf(fmt); \
            }while(0)

/* RIFF WAVE file struct.
 * For details see WAVE file format documentation
 *(for example at <a href = "http://www.wotsit.org)." target = "_blank"> http://www.wotsit.
org).</a> */
```

```
typedef struct WAV_HEADER_S
{
    char                riffType[4];        //4Byte,资源交换文件标志: RIFF
    unsigned int        riffSize;           //4Byte,从下个地址到文件结尾的总字节数
    char                waveType[4];        //4Byte,wave 文件标志: WAVE
    char                formatType[4];      //4Byte,波形文件标志: FMT
    unsigned int        formatSize;         //4Byte,音频属性(compressionCode,numChannels,
//sampleRate,BytesPerSecond,blockAlign,bitsPerSample)所占字节数
    unsigned short      compressionCode;    //2Byte,编码格式(1 - 线性 pcm - WAVE_FORMAT_PCM,
                                            //WAVEFORMAT_ADPCM)
    unsigned short      numChannels;        //2Byte,通道数
    unsigned int        sampleRate;         //4Byte,采样率
    unsigned int        BytesPerSecond;     //4Byte,传输速率
    unsigned short      blockAlign;         //2Byte,数据块的对齐
    unsigned short      bitsPerSample;      //2Byte,采样精度
    char                dataType[4];        //4Byte,数据标志: data
    unsigned int        dataSize;           //4Byte,从下个地址到文件结尾的总字节数,即除了
                                            //WAV Header 以外的 PCM Data Length
}WAV_HEADER;

typedef struct WAV_INFO_S
{
  WAV_HEADER            header;
  FILE                  *fp;
  unsigned int          channelMask;
}WAV_INFO;

#endif
```

新建源文件,文件名为 wav.c,功能可参考注释信息,代码如下:

```
//chapter5/wav.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "wav.h"

/* 解析 WAV 头
*/
int IS_LITTLE_ENDIAN(void)        //判断是否为小端字节序
{
    int __dummy = 1;
    return( *((unsigned char *)(&(__dummy) ) ) );
}
```

```
//读取头
unsigned int readHeader(void * dst,signed int size,signed int nmemb,FILE * fp)
{
    unsigned int n,s0,s1,err;
    unsigned char tmp, * ptr;

//从文件中读取指定的字节数
    if((err = fread(dst,size,nmemb,fp)) != nmemb)
    {
        return err;
    }
    if(!IS_LITTLE_ENDIAN() && size > 1)
    {
        //DeBug("big - endian \n");
        ptr = (unsigned char * )dst;
        for(n = 0; n < nmemb; n++)
        {
             for(s0 = 0,s1 = size - 1; s0 < s1; s0++,s1 -- )
             {
             tmp = ptr[s0];
             ptr[s0] = ptr[s1];
             ptr[s1] = tmp;
             }
             ptr += size;
        }
    }
    else
    {
        //DeBug("little - endian \n");
    }

      return err;
}

//显示出 WAV 关键字段信息
void dumpWavInfo(WAV_INFO wavInfo)
{
    DeBug("compressionCode: %d \n",wavInfo.header.compressionCode);
    DeBug("numChannels: %d \n",wavInfo.header.numChannels);
    DeBug("sampleRate: %d \n",wavInfo.header.sampleRate);
    DeBug("BytesPerSecond: %d \n",wavInfo.header.BytesPerSecond);
    DeBug("blockAlign: %d \n",wavInfo.header.blockAlign);
    DeBug("bitsPerSample: %d \n",wavInfo.header.bitsPerSample);

}
```

```
//打开 WAV 文件,输入文件路径,然后开始解析
int wavInputOpen(WAV_INFO * pWav,const char * filename)
{
    signed int offset;
    WAV_INFO * wav = pWav;

    if(wav == NULL)
    {
      DeBug("Unable to allocate WAV struct.\n");
      goto error;
    }
    wav->fp = fopen(filename,"rb");    //以二进制方式打开 WAV 文件
    if(wav->fp == NULL)
    {
      DeBug("Unable to open WAV file. %s\n",filename);
      goto error;
    }

    /* RIFF 标志符判断 */
    if(fread(&(wav->header.riffType),1,4,wav->fp) != 4)
    {
      DeBug("couldn't read RIFF_ID\n");
      goto error; /* bad error "couldn't read RIFF_ID" */
    }
    if(strncmp("RIFF",wav->header.riffType,4))
    {
      DeBug("RIFF descriptor not found.\n") ;
      goto error;
    }
    DeBug("Find RIFF \n");

    /* Read RIFF size. Ignored. */
    readHeader(&(wav->header.riffSize),4,1,wav->fp);
    DeBug("wav->header.riffSize: %d \n",wav->header.riffSize);

    /* WAVE 标志符判断 */
    if(fread(&wav->header.waveType,1,4,wav->fp) != 4)
    {
          DeBug("couldn't read format\n");
          goto error; /* bad error "couldn't read format" */
    }
    if(strncmp("WAVE",wav->header.waveType,4))
    {
          DeBug("WAVE chunk ID not found.\n") ;
          goto error;
    }
```

```
 DeBug("Find WAVE \n");

 /* fmt 标志符判断 */
 if(fread(&(wav->header.formatType),1,4,wav->fp) != 4)
 {
       DeBug("couldn't read format_ID\n");
       goto error; /* bad error "couldn't read format_ID" */
 }
 if(strncmp("fmt",wav->header.formatType,3))
 {
       DeBug("fmt chunk format not found.\n") ;
  goto error;
 }
 DeBug("Find fmt \n");

readHeader(&wav->header.formatSize,4,1,wav->fp); //Ignored
DeBug("wav->header.formatSize: %d \n",wav->header.formatSize);

 /* read info: 读取 WAV 头字段 */
 readHeader(&(wav->header.compressionCode),2,1,wav->fp);
 readHeader(&(wav->header.numChannels),2,1,wav->fp);
 readHeader(&(wav->header.sampleRate),4,1,wav->fp);
 readHeader(&(wav->header.BytesPerSecond),4,1,wav->fp);
 readHeader(&(wav->header.blockAlign),2,1,wav->fp);
 readHeader(&(wav->header.bitsPerSample),2,1,wav->fp);

 offset = wav->header.formatSize - 16;

 /* Wav format extensible: WAV 扩展信息 */
 if(wav->header.compressionCode == 0xFFFE)
 {
          static const unsigned char guidPCM[16] = {
                  0x01,0x00,0x00,0x00,0x00,0x00,0x10,0x00,
                  0x80,0x00,0x00,0xaa,0x00,0x38,0x9b,0x71
          };
          unsigned short extraFormatBytes,validBitsPerSample;
          unsigned char guid[16];
          signed int i;

          /* read extra Bytes */
          readHeader(&(extraFormatBytes),2,1,wav->fp);
          offset -= 2;

          if(extraFormatBytes >= 22)
          {
               readHeader(&(validBitsPerSample),2,1,wav->fp);
```

```
                readHeader(&(wav->channelMask),4,1,wav->fp);
                readHeader(&(guid),16,1,wav->fp);

                /* check for PCM GUID */
                for(i=0; i<16; i++) if(guid[i] != guidPCM[i]) break;
                if(i == 16) wav->header.compressionCode=0x01;

                offset -= 22;
            }
    }
    DeBug("wav->header.compressionCode: %d \n",wav->header.compressionCode);

    /* Skip rest of fmt header if any. */
    for(; offset>0; offset--)
    {
            fread(&wav->header.formatSize,1,1,wav->fp);
    }

    #if 1
    do
    {
        /* Read data chunk ID : 读取数据块 */
        if(fread(wav->header.dataType,1,4,wav->fp) != 4)
        {
            DeBug("Unable to read data chunk ID.\n");
            free(wav);
            goto error;
        }
        /* Read chunk length. : 块长度 */
     readHeader(&offset,4,1,wav->fp);

        /* Check for data chunk signature.: 检测数据块的标识: data */
        if(strncmp("data",wav->header.dataType,4) == 0)
        {
            DeBug("Find data \n");
            wav->header.dataSize=offset;
            break;
        }

        /* Jump over non data chunk. */
        for(; offset>0; offset--)
        {
            fread(&(wav->header.dataSize),1,1,wav->fp);
        }
    } while(!feof(wav->fp));
    DeBug("wav->header.dataSize: %d \n",wav->header.dataSize);
```

```
    #endif

    /* return success */
    return 0;

/* Error path */
error:
    if(wav)
    {
      if(wav->fp)
      {
        fclose(wav->fp);
        wav->fp = NULL;
      }
      //free(wav);
    }
    return -1;
}

#if 0
int main(int argc,char **argv)

{
    WAV_INFO wavInfo;
    char fileName[128];
    if(argc<2 || strlen(&argv[1][0])>=sizeof(fileName))
    {
        DeBug("argument error !!! \n");
        return -1 ;
    }
    DeBug("size : %d \n",sizeof(WAV_HEADER));
    strcpy(fileName,argv[1]);
    wavInputOpen(&wavInfo,fileName);
    return 0;
}
#endif
```

5.3 音频编解码简介及命令行案例

音频信号数字化是指将连续的模拟信号转换成离散的数字信号，完成采样、量化和编码3个步骤。它又称为脉冲编码调制，通常由A/D转换器实现。音频编码有3类常用方法，包括波形编码、参数编码和混合编码。波形编码会尽量保持输入的波形不变，即重建的语音信号基本上与原始语音信号波形相同，压缩比较低。参数编码会要求重建的信号听起来与

输入语音一样，但其波形可以不同，它是以语音信号所产生的数学模型为基础的一种编码方法，压缩比较高。混合编码是综合了波形编码的高质量潜力和参数编码的高压缩效率的混合编码方法，这类方法也是目前低码率编码的发展方向。常见的音频编解码格式包括MP3、AAC、AC-3 等。

5.3.1　PCM 编码为 AAC

使用 FFmpeg 可以将 PCM 格式的音频编码为 AAC 格式，命令如下：

```
ffmpeg -ar 48000 -ac 2 -f s16le -i 48000_2_s16le.pcm -acodec aac
out-48000-s16le-c2.aac
```

在该案例中，需要指定输入的 PCM 文件的采样率(-ar 48000)、声道数(-ac 2)、采样格式(-f s16le)，否则当 FFmpeg 无法识别出来时会导致编码失败，-acodec aac 指定了使用 AAC 进行编码(FFmpeg 内置了 AAC 编码器)。FFmpeg 转换过程的主要输出信息如下：

```
//chapter5/ffmpeg-pcm-1-out.txt
ffmpeg -ar 48000 -ac 2 -f s16le -i 48000_2_s16le.pcm -acodec aac
out-48000-s16le-c2.aac
Input #0,s16le,from '48000_2_s16le.pcm':
  Duration: 00:00:09.99,bitrate: 1535 kb/s
    Stream #0:0: Audio: pcm_s16le,48000 Hz,stereo,s16,1536 kb/s
Stream mapping:
  Stream #0:0 -> #0:0(pcm_s16le(native) -> aac(native))
Press [q] to stop,[?] for help
Output #0,adts,to 'out-48000-s16le-c2.aac':
  Metadata:
    encoder         : Lavf58.45.100
    Stream #0:0: Audio: aac(LC),48000 Hz,stereo,fltp,128 kb/s
    Metadata:
      encoder       : Lavc58.91.100 aac
```

使用 MediaInfo 查看生成的 out-48000-s16le-c2.aac 的流信息，如图 5-45 所示。

在该案例中，也可以指定新的采样率(-ar 4800)及声道数(-ac 1)等，命令如下：

```
ffmpeg -ar 48000 -ac 2 -f s16le -i 48000_2_s16le.pcm -acodec aac -r 44100 -ac 1 out-
44100-s16le-c1.aac
#注意：-ar需要放在-i之前，用于指定输入文件的采样率；-r需要放在-i之后，用于指定输出
#文件的采样率。-ac用于指定声道数，-f用于指定采样格式
```

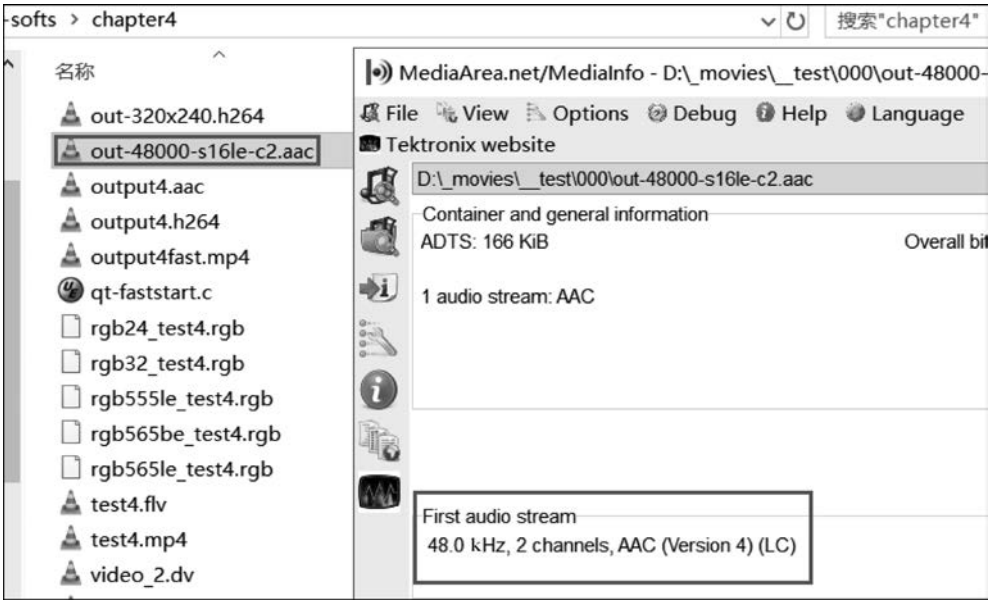

图 5-45　MediaInfo 查看 aac 文件的流信息

5.3.2　AAC 转码为 MP3

使用 FFmpeg 可以将 AAC 格式的音频转码为 MP3 格式，转码过程如图 5-46 所示，命令如下：

```
ffmpeg -i test4.aac -acodec libmp3lame test4.mp3
```

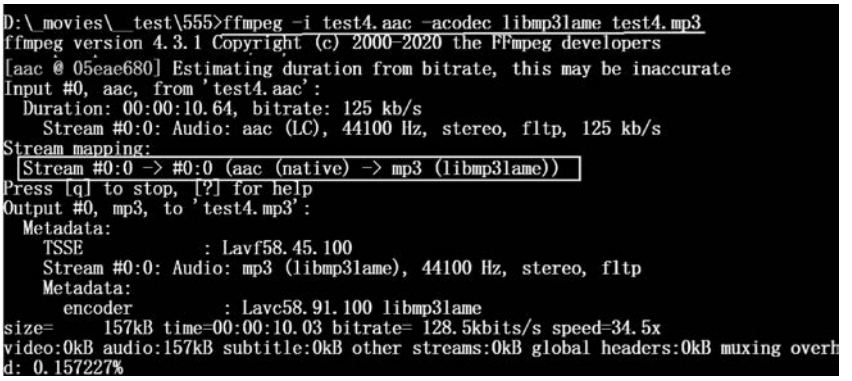

图 5-46　FFmpeg 将 AAC 格式转码为 MP3 格式

在该案例中，FFmpeg 使用了第三方的编码器 libmp3lame，这个不是 FFmpeg 自带的，所以编译时需要配置好(--enable-libmp3lame)。分析输入文件 test4.aac 和输出文件 test4.

mp3，如图 5-47 所示，可以发现转码前后的帧率(44.1kHz)、声道数(2 Channels)一致，但封装格式不同。

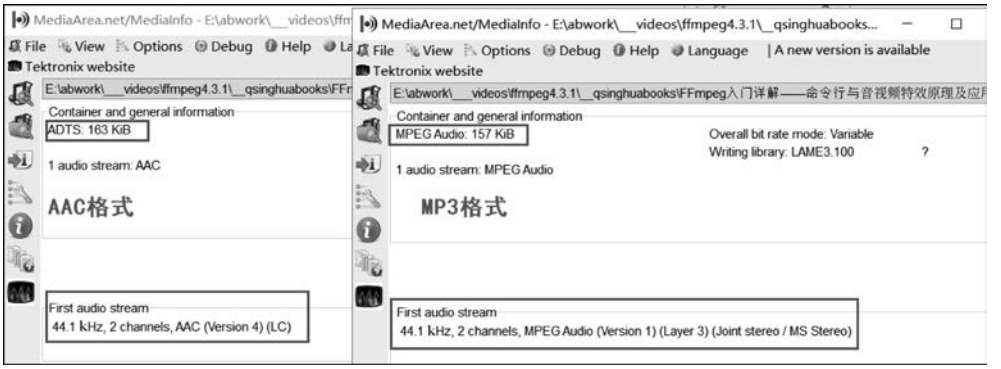

图 5-47　通过 MediaInfo 比较 AAC 格式与 MP3 格式的区别

MP3 全称是 MPEG-1 Audio Layer 3，它在 1992 年合并至 MPEG 规范中。它能够以高音质、低采样率对数字音频文件进行压缩，应用最普遍。MP3 具有不错的压缩比，使用 LAME 编码的中高码率的 MP3 文件，听感上非常接近源 WAV 文件。其特点是音质在 128kb/s 以上表现还不错，压缩比比较高，兼容性好。在高比特率下可欣赏对兼容性有要求的音乐。

高级音频编码(Advanced Audio Coding，AAC)是由 Fraunhofer IIS-A、杜比和 AT&T 共同开发的一种音频格式，它是 MPEG-2 规范的一部分。AAC 所采用的运算法则与 MP3 的运算法则有所不同，AAC 通过结合其他的功能来提高编码效率。AAC 的音频算法在压缩能力上远远超过了以前的一些压缩算法，如 MP3。它还同时支持多达 48 个音轨、15 个低频音轨，支持更多种采样率和比特率，具有多种语言的兼容能力和更高的解码效率。总之，AAC 可以在比 MP3 文件缩小 30%的前提下提供更好的音质。

5.3.3　AAC 转码为 AC-3

使用 FFmpeg 可以将 AAC 格式的音频转码为 AC-3 格式，转码过程如图 5-48 所示，命令如下：

```
ffmpeg -i test4.aac -acodec ac3 -y test4.ac3
```

在该案例中，FFmpeg 使用了内置的编码器 ac3，通过 MediaInfo 查看输出文件 test4.ac3 的流信息，如图 5-49 所示。

查看 FFmpeg 支持的编码器，命令如下：

```
ffmpeg -encoders | findstr ac3
ffmpeg -encoders | findstr aac
```

```
D:\_movies\__test\555>ffmpeg -i test4.aac  -acodec ac3  -y test4.ac3
ffmpeg version 4.3.1 Copyright (c) 2000-2020 the FFmpeg developers
[aac @ 0687e680] Estimating duration from bitrate, this may be inaccurate
Input #0, aac, from 'test4.aac':
  Duration: 00:00:10.64, bitrate: 125 kb/s
    Stream #0:0: Audio: aac (LC), 44100 Hz, stereo, fltp, 125 kb/s
Stream mapping:
  Stream #0:0 -> #0:0 (aac (native) -> ac3 (native))
Press [q] to stop, [?] for help
Output #0, ac3, to 'test4.ac3':
  Metadata:
    encoder         : Lavf58.45.100
    Stream #0:0: Audio: ac3, 44100 Hz, stereo, fltp, 192 kb/s
    Metadata:
      encoder         : Lavc58.91.100 ac3
size=     235kB time=00:00:10.02 bitrate= 192.1kbits/s speed= 176x
video:0kB audio:235kB subtitle:0kB other streams:0kB global headers:0kB muxing overhea
```

图 5-48　FFmpeg 将 AAC 格式转码为 AC-3 格式

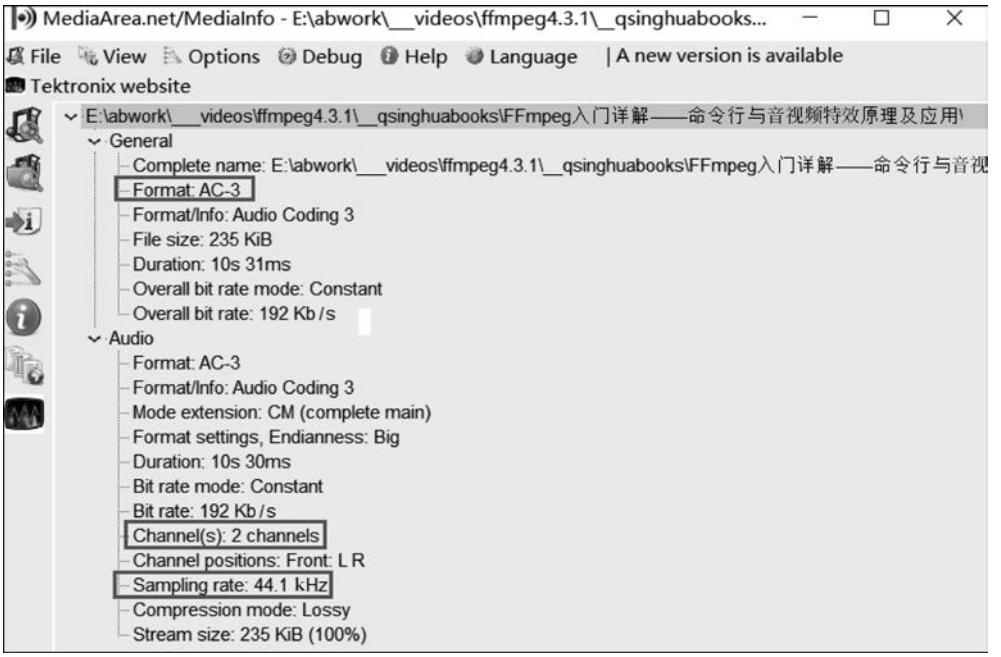

图 5-49　通过 MediaInfo 查看 AC-3 文件的流信息

FFmpeg 的输出信息如下：

```
//chapter5/ffmpeg - pcm - 1 - out.txt
A..... ac3              ATSC A/52A(AC - 3)
A..... ac3_fixed        ATSC A/52A(AC - 3)(codec ac3)
A..... ac3_mf           AC3 via MediaFoundation(codec ac3)
A..... eac3             ATSC A/52 E - AC - 3
 ------------------------------------------
A..... aac              AAC(Advanced Audio Coding)
A..... aac_mf           AAC via MediaFoundation(codec aac)
```

1994年,日本先锋公司宣布与美国杜比实验室合作研制成功一种崭新的环绕声制式,并命名为“杜比AC-3”(Dolby Surround Audio Coding-3)。1997年年初,杜比实验室正式将“杜比AC-3环绕声”改为“杜比数码环绕声”(Dolby Surround Digital),常称为Dolby Digital。AC-3提供的环绕声系统由5个全频域声道加一个超低音声道组成,所以被称作5.1声道。5个声道包括前置的左声道、中置声道、右声道、后置的左环绕声道和右环绕声道。这些声道的频率范围均为全频域响应3～20 000Hz。第6个声道为超低音声道,包含了一些额外的低音信息,使一些场景(如爆炸、撞击声等)的效果更好。由于这个声道的频率响应为3～120Hz,所以称为“5.1声道”。6个声道的信息在制作和还原过程中全部数字化,信息损失很少,全频段的细节十分丰富。杜比数字AC-3是根据感觉来开发的编码系统多声道环绕声。它将每种声音的频率根据人耳的听觉特性区分为许多窄小频段,在编码过程中再根据音响心理学的原理进行分析,保留有效的音频,删除多余的信号和各种噪音频率,使重现的声音更加纯净,分离度极高。

5.4 视频编解码简介及命令行案例

有两大正式组织研制视频编码标准,包括ISO/IEC和ITU-T。ISO/IEC制定的编码标准有MPEG-1、MPEG-2、MPEG-4、MPEG-7、MPEG-21和MPEG-H等。ITU-T制定的编码标准有H.261、H.262、H.263、H.264和H.265等。实际上,真正在业界产生较强影响力的标准均是由这两个组织合作制定的,例如MPEG-2、H.264/AVC和H.265/HEVC等。不同标准组织在不同时期制定的视频编码标准,如图5-50所示。

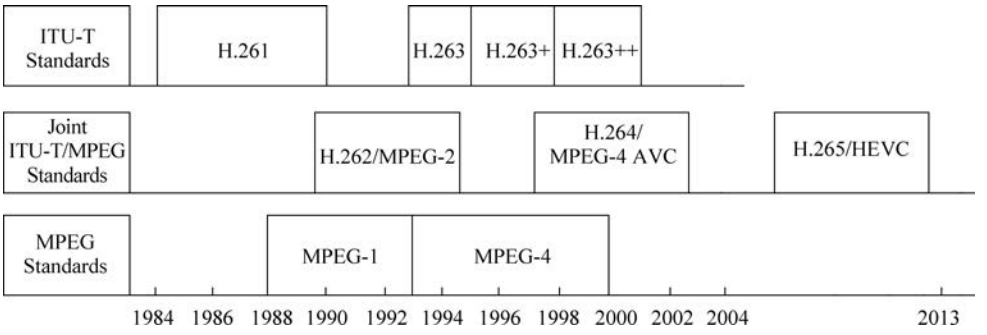

图5-50 H.26x与MPEGx

30多年以来,世界上主流的视频编码标准,基本上是由ITU和ISO/IEC制定的。ITU制定了H.261、H.262、H.263、H.263+、H.263++,这些统称为H.26x系列,主要应用于实时视频通信领域,如会议电视、可视电话等。ISO/IEC制定了MPEG-1、MPEG-2、MPEG-4、MPEG-7、MPEG-21,统称为MPEG系列。ITU和ISO/IEC一开始各自为战,后来这两个组织成立了一个联合小组,名叫JVT,即视频联合工作组(Joint Video Team)。H.264就是由JVT联合制定的,它也属于MPEG-4家族的一部分,即MPEG-4系列文档

ISO-14496 的第 10 部分，因此又称作 MPEG-4/AVC。同 MPEG-4 重点考虑的灵活性和交互性不同，H.264 着重强调更高的编码压缩率和传输可靠性，在数字电视广播、实时视频通信、网络流媒体等领域具有广泛的应用。

5.4.1 YUV 编码为 H.264

使用 FFmpeg 可以将 YUV 格式的视频编码为 H.264 格式，命令如下：

```
ffmpeg -s 320x240 -pix_fmt yuv420p -i yuv420p_test4_320x240.yuv -c:v libx264 out-320x240.h264
```

在该案例中，需要指定输入的 YUV 文件的格式(-pix_fmt yuv420p)及宽和高(-s 320x240)，否则 FFmpeg 无法识别出来，从而可导致编码失败，-c:v libx264 指定了使用 libx264 进行编码。FFmpeg 转换过程的主要输出信息如下：

```
//chapter5/ffmpeg-pcm-1-out.txt
D:\_movies\__test\000>ffmpeg -s 320x240 -pix_fmt yuv420p -i yuv420p_test4_320x240.yuv
-c:v libx264 out-320x240.h264
[rawvideo @ 0530e200] Estimating duration from bitrate, this may be inaccurate
Input #0, rawvideo, from 'yuv420p_test4_320x240.yuv':
  Duration: 00:00:00.44, start: 0.000000, bitrate: 23040 kb/s
    Stream #0:0: Video: rawvideo(I420 / 0x30323449), yuv420p, 320x240, 23040 kb/s, 25 tbr, 25
tbn, 25 tbc
Stream mapping://以下是流映射信息，从 yuv420p 转换为 h264 格式，使用 libx264 编码器
  Stream #0:0 -> #0:0(rawvideo(native) -> h264(libx264))
Press [q] to stop, [?] for help
[libx264 @ 06941540] using cpu capabilities: MMX2 SSE2Fast SSSE3 SSE4.2 AVX FMA3 BMI2 AVX2
[libx264 @ 06941540] profile High, level 1.3, 4:2:0, 8-bit【High Profile】
Output #0, h264, to 'out-320x240.h264':
  Metadata:
    encoder         : Lavf58.45.100
    Stream #0:0: Video: h264(libx264), yuv420p, 320x240, q=-1--1, 25 fps, 25 tbn, 25 tbc
    Metadata:
      encoder         : Lavc58.91.100 libx264 #指定使用 libx264 编码器
...
[libx264 @ 06941540] frame I:1       Avg QP:25.31 size:14405    #I帧总数
[libx264 @ 06941540] frame P:4       Avg QP:27.93 size:1010     #P帧总数
[libx264 @ 06941540] frame B:6       Avg QP:30.77 size:251      #B帧总数
...
[libx264 @ 06941540] Weighted P-Frames: Y:0.0% UV:0.0%
...
[libx264 @ 06941540] kb/s:362.73
```

使用 MediaInfo 查看生成的 out-320x240.h264 文件的流信息，如图 5-51 所示。

在该案例中，也可以指定新的宽和高(-s 160x120)及帧率(-r 30)等，命令如下：

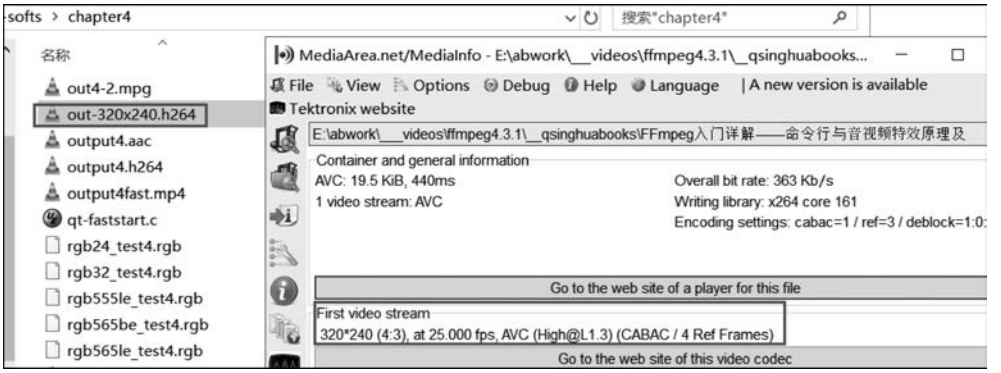

图 5-51　MediaInfo 查看 H.264 文件的流信息

```
ffmpeg -s 320x240 -pix_fmt yuv420p -i yuv420p_test4_320x240.yuv -c:v libx264 -s
160x120 -r 30 out-160x120.h264
#注意：第1个-s需要放在-i之前，用于指定输入文件的宽和高；第2个-s需要放在-i之后，
#用于指定输出文件的宽和高
```

注意：由于 YUV 文件本身不带参数信息，所以需要在命令行中指定具体的 YUV 格式及宽和高。

5.4.2　MP4 格式转码为 FLV 格式

使用 FFmpeg 可以将 MP4 格式的音视频编码为 FLV 格式，命令如下：

```
ffmpeg -i test4.mp4 -y test4-default.flv
```

在该案例中，由于没有指定转码格式，而仅将输出文件指定为.flv 格式，所以 FFmpeg 使用了默认的转码格式。可以看出，.flv 格式使用的默认视频格式为 flv1，默认音频格式为 mp3。由于需要进行解码，然后编码，所以 FFmpeg 的转换速度比较慢，并且输出了转码进度信息，具体的转码过程如图 5-52 所示。

通过 MediaInfo 可以观察输入文件 test4.mp4 和输出文件 test4-default.flv 的流信息，如图 5-53 所示。

如果不想重新编码而只是对封装格式进行转换，则只需指定-c copy（相当于-acodec copy -vcodec copy -scodec copy），命令如下：

```
ffmpeg -i test4.mp4 -c copy -y test4-copy.flv
```

在该案例中，音视频流没有重新转码，只是进行了复制，速度很快，具体过程如图 5-54 所示。

注意：FLV 封装格式完全兼容 H.264、AAC，所以使用-c copy 可以成功。如果遇到不兼容的音视频编码格式，则会失败。

```
D:\_movies\__test\555>ffmpeg -i test4.mp4 -y test4-default.flv
ffmpeg version 4.3.1 Copyright (c) 2000-2020 the FFmpeg developers
 Stream mapping:
   Stream #0:0 -> #0:0 (h264 (native) -> flv1 (flv))
a  Stream #0:1 -> #0:1 (aac (native) -> mp3 (mp3_mf))
]Press [q] to stop, [?] for help
w[mp3_mf @ 07ada1c0] MFT name: 'MP3 Encoder ACM Wrapper MFT'
-Output #0, flv, to 'test4-default.flv':
    Metadata:
      handler_name    : SoundHandler
      encoder         : Lavc58.91.100 mp3_mf
frame=    1 fps=0.0 q=9.8 size=     157kB time=00:00:00.21 bitrate=6088.4kbits/s speed
frame=   63 fps= 61 q=31.0 size=     768kB time=00:00:02.27 bitrate=2761.8kbits/s spee
frame=  126 fps= 82 q=31.0 size=    1280kB time=00:00:04.36 bitrate=2401.1kbits/s spee
frame=  190 fps= 93 q=31.0 size=    1792kB time=00:00:06.48 bitrate=2265.4kbits/s spee
frame=  254 fps=100 q=31.0 size=    2560kB time=00:00:08.64 bitrate=2427.3kbits/s spee
frame=  319 fps=105 q=31.0 size=    3072kB time=00:00:10.80 bitrate=2330.0kbits/s spee
frame=  384 fps=108 q=31.0 size=    3840kB time=00:00:12.95 bitrate=2427.4kbits/s spee
frame=  403 fps=109 q=31.0 Lsize=    4113kB time=00:00:13.47 bitrate=2501.5kbits/s spe
ed=3.64x
video:3888kB audio:211kB subtitle:0kB other streams:0kB global headers:0kB muxing over
head: 0.344596%
```

视频转码：h264->flv1
音频转码：aac->mp3
下面是转码进度

图 5-52　FFmpeg 将 MP4 格式的文件转码为 FLV 格式

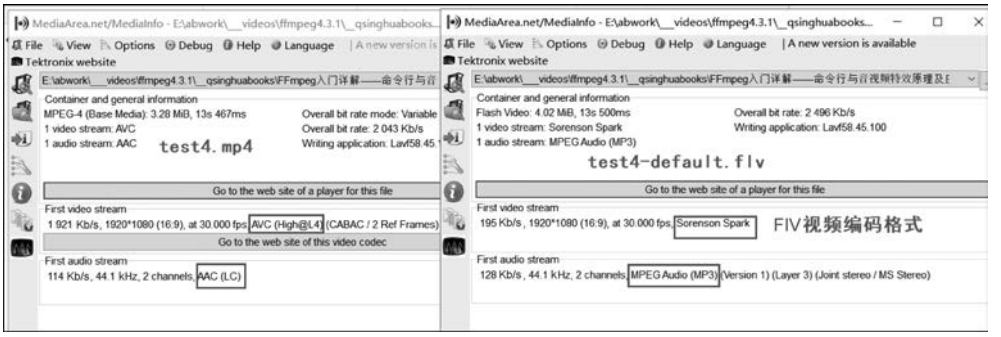

图 5-53　通过 MediaInfo 查看 MP4 格式与 FLV 格式的区别

```
    Metadata:
      handler_name    : SoundHandler
Stream mapping:
  Stream #0:0 -> #0:0 (copy)
  Stream #0:1 -> #0:1 (copy)
Press [q] to stop, [?] for help
frame=  403 fps=0.0 q=-1.0 Lsize=    3356kB time=00:00:13.42 bitrate=2048.7kbits/s spe
ed=2.05e+03x
video:3151kB audio:188kB subtitle:0kB other streams:0kB global headers:0kB muxing over
head: 0.538390%
```

图 5-54　FFmpeg 将 MP4 格式转换为 FLV 格式(-c copy)

也可以将音频编码格式指定为 AC-3，命令如下：

```
ffmpeg -i test4.mp4 -vcodec copy -acodec ac3 -y test4-vcopy-ac3.flv
```

在该案例中，直接复制视频流(H.264 格式)，试图将音频流从 AAC 格式转换为 AC-3 格式，但是 FLV 封装格式不兼容 AC-3 音频编码格式，所以会报错，具体过程如图 5-55 所示。

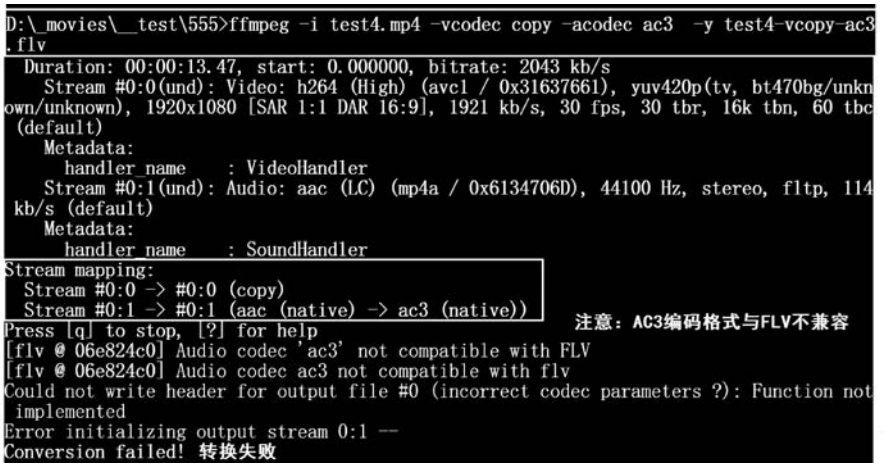

图 5-55　FLV 封装格式不兼容 AC-3 编码格式

5.4.3　MP4 格式转码为 AVI 格式

使用 FFmpeg 可以将 MP4 格式的音视频编码为 AVI 格式，命令如下：

```
ffmpeg -i test4.mp4 -y test4-default.avi
```

在该案例中，由于没有指定转码格式，而仅将输出文件指定为.avi 格式，所以 FFmpeg 使用了默认的转码格式。可以看出，.avi 格式使用的默认视频格式为 MPGE-4（MPEG4-Video），默认音频格式为 MP3。由于需要进行解码，然后编码，所以 FFmpeg 的转换速度比较慢（音视频分别进行了转码），并且输出了转码进度信息，具体的转码过程如图 5-56 所示。

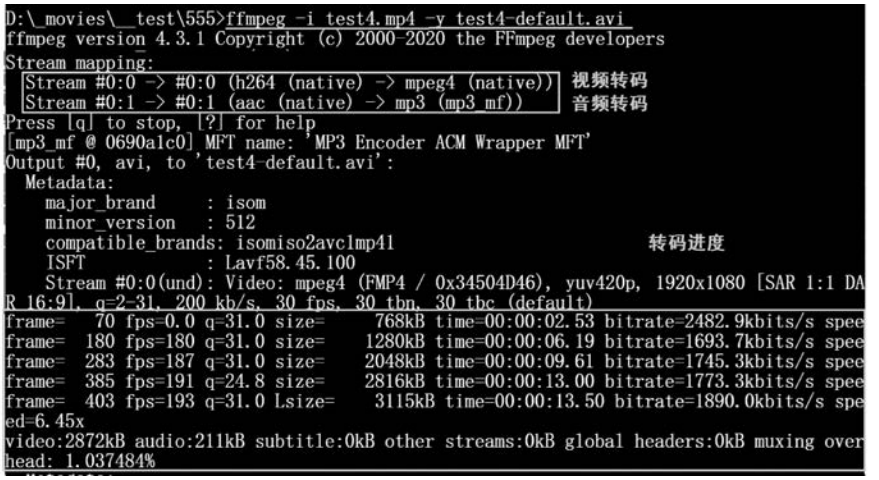

图 5-56　FFmpeg 将 MP4 格式转换为 AVI 格式

如果不想重新编码而只是对封装格式进行转换，则只需指定-c copy（相当于-acodec copy -vcodec copy -scodec copy），命令如下：

```
ffmpeg -i test4.mp4 -c copy -y test4-copy.avi
```

在该案例中，音视频流没有重新转码，只是进行了复制，速度很快，具体过程如图 5-57 所示。

```
D:\_movies\__test\555>ffmpeg -i test4.mp4 -c copy -y test4-copy.avi
ffmpeg version 4.3.1 Copyright (c) 2000-2020 the FFmpeg developers
    minor_version   : 512
    compatible_brands: isomiso2avc1mp41
    ISFT            : Lavf58.45.100
    Stream #0:0(und): Video: h264 (High) (avc1 / 0x31637661), yuv420p(tv, bt470bg/unkn
own/unknown), 1920x1080 [SAR 1:1 DAR 16:9], q=2-31, 1921 kb/s, 30 fps, 30 tbr, 60 tbn,
 60 tbc (default)
    Metadata:
      handler_name    : VideoHandler
    Stream #0:1(und): Audio: aac (LC) ([255][0][0][0] / 0x00FF), 44100 Hz, stereo, flt
p, 114 kb/s (default)
    Metadata:
      handler_name    : SoundHandler
Stream mapping:
  Stream #0:0 -> #0:0 (copy)
  Stream #0:1 -> #0:1 (copy)
Press [q] to stop, [?] for help
frame=  403 fps=0.0 q=-1.0 Lsize=    3381kB time=00:00:13.42 bitrate=2063.7kbits/s spe
ed=2.03e+03x
video:3151kB audio:188kB subtitle:0kB other streams:0kB global headers:0kB muxing over
head: 1.280796%
```

音视频流直接复制，转码速度很快

图 5-57　FFmpeg 将 MP4 格式转换为 AVI 格式(-c copy)

通过 MediaInfo 可以观察输出文件 test4-default.avi 的流信息，如图 5-58 所示。

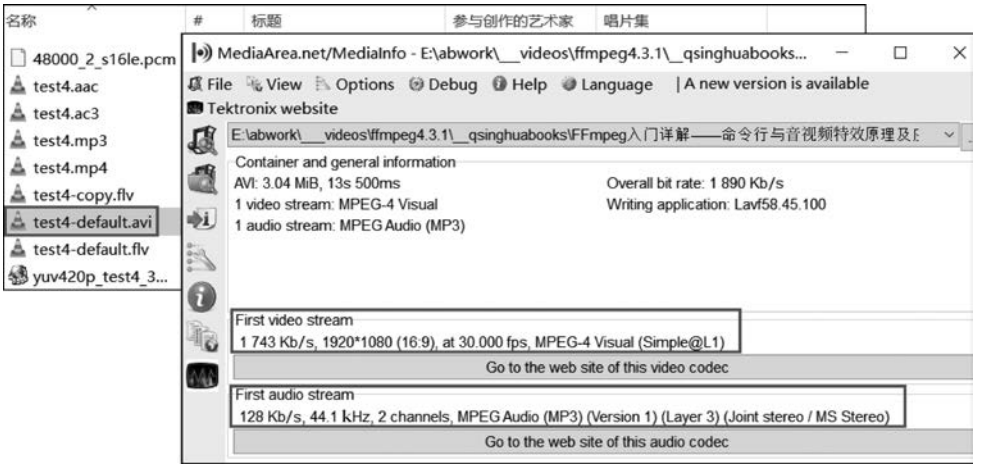

图 5-58　通过 MediaInfo 查看 AVI 文件中的流信息

也可以将音频编码格式指定为 AC-3，命令如下：

```
ffmpeg -i test4.mp4 -vcodec copy -acodec ac3 -y test4-vcopy-ac3.avi
```

在该案例中，直接复制视频流（H.264 格式），试图将音频流从 AAC 格式转换为 AC-3 格式，由于 AVI 封装格式兼容 AC-3 音频编码格式，所以转码成功，具体过程如图 5-59 所示。

图 5-59　通过 FFmpeg 指定 AC-3 转码方式，封装格式为 AVI

通过 MediaInfo 可以观察输出文件 test4-vcopy-ac3.avi 的流信息，如图 5-60 所示。

图 5-60　通过 MediaInfo 查看 AVI（AVC＋AC-3）文件的流信息

5.4.4　MP4 格式转码为 TS 格式

使用 FFmpeg 可以将 MP4 格式的音视频编码为 TS 格式，命令如下：

```
ffmpeg -i test4.mp4 -y test4-default.ts
```

在该案例中，由于没有指定转码格式，而仅将输出文件指定为.ts 的格式，所以 FFmpeg 使用了默认的转码格式。可以看出，.ts 格式使用的默认视频格式为 mpeg2video(MPEG2-Video)，默认音频格式为 MP2。由于需要进行解码，然后编码，所以 FFmpeg 的转换速度比较慢，并且输出了转码进度信息，具体的转码过程如图 5-61 所示。

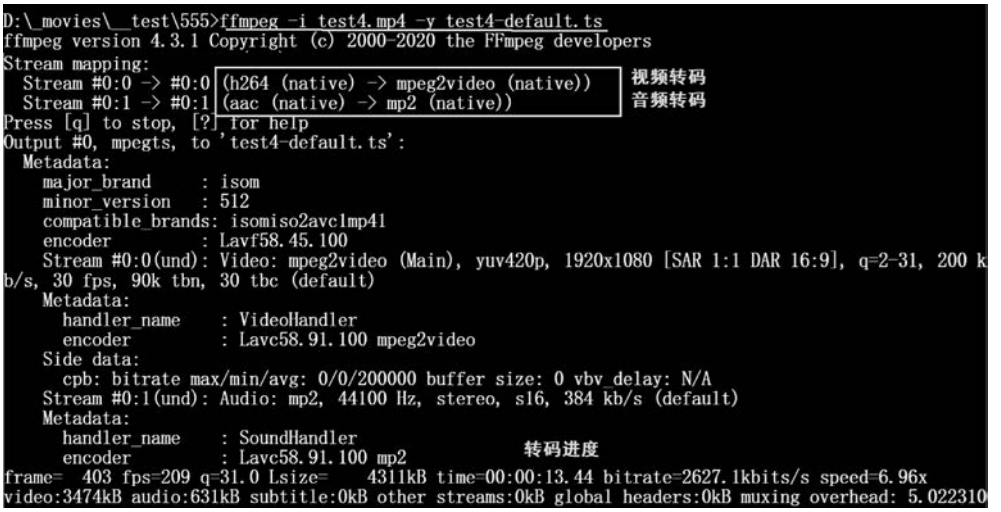

图 5-61 通过 FFmpeg 将 MP4 文件转码为 TS 文件

如果不想重新编码而只是对封装格式进行转换，则只需指定-c copy(相当于-acodec copy -vcodec copy -scodec copy)，命令如下：

```
ffmpeg -i test4.mp4 -c copy -y test4-copy.ts
```

在该案例中，音视频流没有重新转码，只是进行了复制，速度很快。也可以将音频编码格式指定为 AC-3，命令如下：

```
ffmpeg -i test4.mp4 -vcodec copy -acodec ac3 -y test4-vcopy-ac3.ts
```

在该案例中，直接复制视频流(H.264 格式)，试图将音频流从 AAC 格式转换为 AC-3 格式，由于 TS 封装格式兼容 AC-3 音频编码格式，所以转码成功，具体过程如图 5-62 所示。

通过 MediaInfo 可以观察输出文件 test4-vcopy-ac3.ts 的流信息，如图 5-63 所示。

5.4.5 其他格式之间互转

使用 FFmpeg 几乎可以完成各种格式之间的互转，有的可以直接复制音视频流，有的需要重新编解码。因为音视频格式非常多，笔者无法一一列举，读者可以自行实验，遇到格式不兼容的情况，FFmpeg 会报错，并且会输出完整的出错信息。这些错误信息非常重要，读者可以多分析，从中可以学到很多知识。

图 5-62　通过 FFmpeg 指定 AC-3 编码格式，封装为 TS 格式

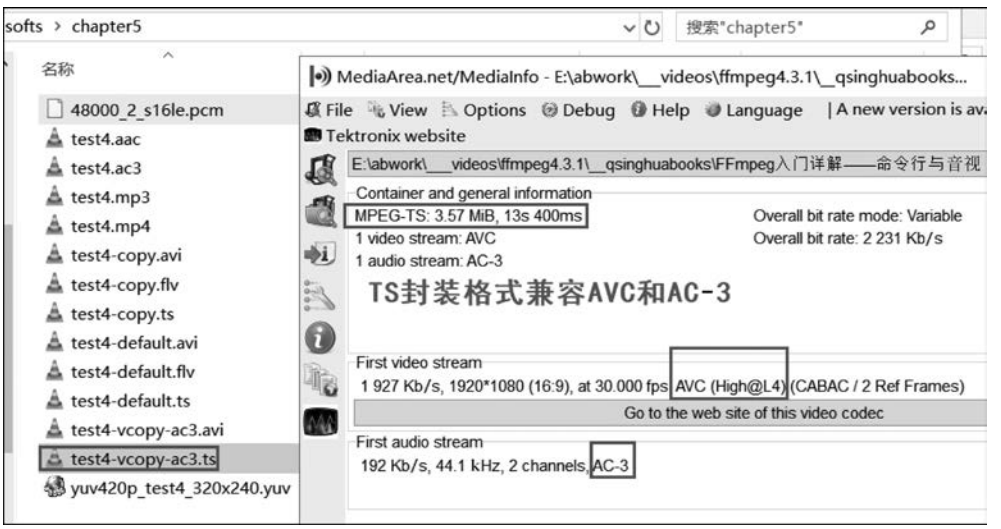

图 5-63　通过 MediaInfo 查看 TS 文件的流信息（AVC＋AC-3）

5.5　控制音频的声道数、采样率及采样格式

音频有三大常用参数，包括声道数、采样率和采样格式，这些参数都可以通过 FFmpeg 实现转码控制。

5.5.1　单声道与立体声互转

使用 FFmpeg 可以将立体声转码为单声道，命令如下：

```
ffmpeg -i test4.mp3 -ac 1 -y test4-ac1.mp3
```

输入文件 test4.mp3 中的声道数是 2，即立体声，使用转码参数-ac 来指定转码后的声道数(Audio Channel)。输入文件 test4.mp3 与输出文件 test4-ac1.mp3 的流信息如图 5-64 所示，可见声道数确实从 2 变成了 1。

图 5-64　通过 MediaInfo 查看声道数

5.5.2　采样率转换

使用 FFmpeg 可以将采样率从 44.1kHz 转码为 48kHz，命令如下：

```
ffmpeg -i test4.mp3 -ar 48000 -y test4-ar48000.mp3
```

输入文件 test4.mp3 中的采样率是 44.1kHz，使用转码参数-ar 来指定转码后的采样率(sample rate)。转换后的文件 test4-ar48000.mp3，通过 MediaInfo 观察，采样率确实变成了 48kHz，如图 5-65 所示。

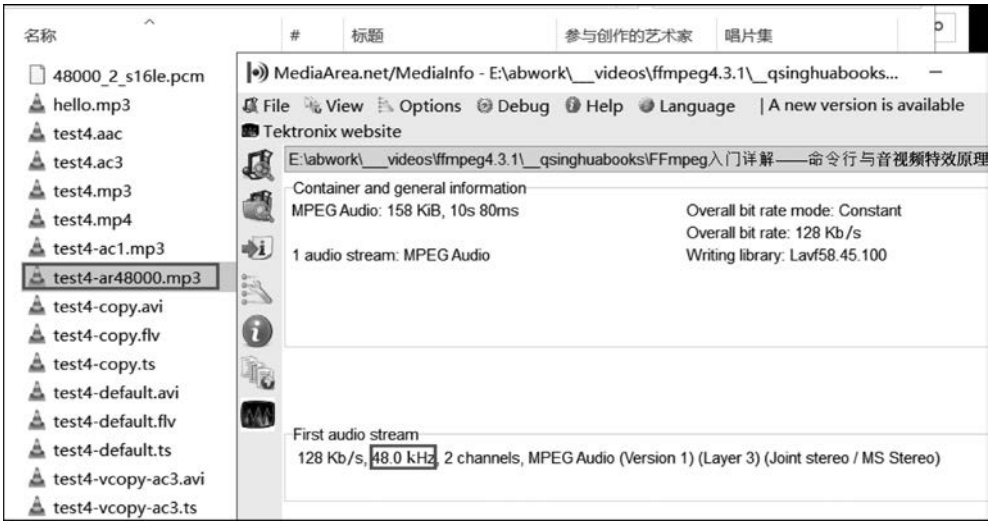

图 5-65　通过 MediaInfo 查看采样率

5.5.3 采样格式转换及音频重采样

通俗地讲，音频重采样就是改变音频的采样率、采样格式(Sample Format)、声道数(channel)等参数，使之按照期望的参数进行输出。当原有的音频参数不满足实际要求时，例如在 FFmpeg 解码音频时，不同的音源有不同的格式和采样率等，所以在解码后的数据中的这些参数也会不一致(最新的 FFmpeg 解码音频后，音频格式为 AV_SAMPLE_FMT_TLTP)。如果接下来需要使用解码后的音频数据进行其他操作，由于这些参数不一致，所以会导致很多额外工作，此时可直接对其进行重采样，获取指定的音频参数，这样就会方便很多。再例如，在将音频进行 SDL 播放时，因为当前的 SDL 2.0 不支持平面(Planar)格式，也不支持浮点型，而最新的 FFmpeg 会将音频解码为浮点型格式(AV_SAMPLE_FMT_FLTP)，这时对它进行重采样之后，就可以在 SDL 2.0 上播放这个音频了。音频重采样包括以下几个重要参数。

(1) Sample Rate(采样率)：采样设备每秒抽取样本的次数。

(2) Sample Format(采样格式)和量化精度：采用什么格式进行采集数据；每种音频格式有不同的量化精度(位宽)，位数越多，表示值就越精确，声音表现就越精准。

(3) Channel Layout(通道布局，也就是声道数)：采样的声道数。

在 FFmpeg 里主要有两种采样格式，包括浮点格式(Floating-Point Formats)和平面格式(Planar Sample Format)，具体采样参数如下(在 libavutil/samplefmt.h 头文件里面)：

```
//chapter5/AVSampleFormat.txt
#注意：P 结尾代表 Planar,即平面模式
enum AVSampleFormat {
    AV_SAMPLE_FMT_NONE = -1,
    AV_SAMPLE_FMT_U8,          //< 8 位元符号
    AV_SAMPLE_FMT_S16,         //< 16 位有符号
    AV_SAMPLE_FMT_S32,         //< 32 位有符号
    AV_SAMPLE_FMT_FLT,         //< 浮点类型
    AV_SAMPLE_FMT_DBL,         //< 双精度浮点类型

    AV_SAMPLE_FMT_U8P,         //< 8 位无符号,平面模式
    AV_SAMPLE_FMT_S16P,        //< 16 位有符号,平面模式
    AV_SAMPLE_FMT_S32P,        //< 32 位有符号,平面模式
    AV_SAMPLE_FMT_FLTP,        //< 符点类型,平面模式
    AV_SAMPLE_FMT_DBLP,        //< 双精度浮点类型,平面模式
    AV_SAMPLE_FMT_S64,         //< 64 位有符号
    AV_SAMPLE_FMT_S64P,        //< 64 位有符号,平面模式

    AV_SAMPLE_FMT_NB           //< Number of sample formats. DO NOT USE if linking dynamically
};
```

使用 FFmpeg 进行音频重采样，主要针对解码后的 PCM 数据，例如可以将一个输入文

件 48000_2_s16le.pcm(采样率为 48 000、声道数为 2、采样格式为 s16le)转码为采样率为 44 100、声道数为 1、采样格式为 s32le,转码过程如图 5-66 所示,命令如下:

```
//chapter5/ffmpeg-pcm-1-out.txt
ffmpeg -ar 48000 -ac 2 -f s16le -i 48000_2_s16le.pcm -ar 44100 -ac 1 -f s32le
-y 44100_1_s32le.pcm
#-ac: -i之前用于指定输入的声道数,-i之后用于指定输出的声道数
#-ar: -i之前用于指定输入的采样率,-i之后用于指定输出的采样率
#-f: -i之前用于指定输入的采样格式,-i之后用于指定输出的采样格式
```

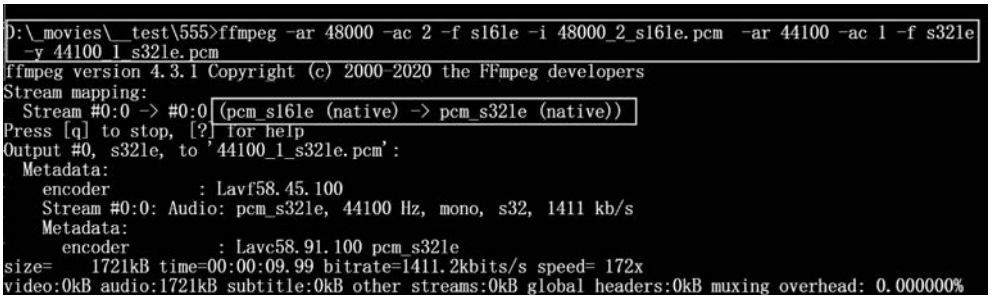

图 5-66 通过 FFmpeg 进行 PCM 声音数据的重采样

可以使用 ffplay 来播放 PCM 文件,需要指定相关参数,命令如下:

```
ffplay -ar 44100 -ac 1 -f s32le -i 44100_1_s32le.pcm
```

注意:可以使用 ffmpeg -sample_fmts 列举所有可用的采样格式。

5.6 控制视频的帧率、码率及分辨率

视频有几个常用参数,包括帧率、码率、分辨率、GOP 等,这些参数都可以通过 FFmpeg 实现转码控制。

5.6.1 控制视频的帧率

使用 FFmpeg 的-r 参数可以控制视频的帧率,可以放到-i 前边或后边,但含义不同。

1. -r 放到-i 后边

使用 FFmpeg 的-r 参数可以控制视频的帧率,放到-i 后边,用于控制转码后的视频帧率,命令如下:

```
ffmpeg -i test4.mp4 -r 15 -y test4-r15.mp4
```

在该案例中,输入文件 test4.mp4 本来的帧率是 30,转码后的帧率是 15,转码过程如

图 5-67 所示。由于封装格式是 MP4，所以 FFmpeg 使用 libx264 进行视频转码，使用 AAC 进行音频转码。

```
D:\_movies\__test\555>ffmpeg  -i test4.mp4  -r 15 -y test4-r15.mp4
ffmpeg version 4.3.1 Copyright (c) 2000-2020 the FFmpeg developers
Stream mapping:
  Stream #0:0 -> #0:0 (h264 (native) -> h264 (libx264))
  Stream #0:1 -> #0:1 (aac (native) -> aac (native))
frame=   39 fps=0.0 q=0.0 size=       0kB time=00:00:02.55 bitrate=   0.2kbits/s dup=0
frame=   57 fps= 53 q=27.0 size=       0kB time=00:00:03.76 bitrate=   0.1kbits/s dup=
frame=   87 fps= 55 q=27.0 size=     256kB time=00:00:05.75 bitrate= 364.3kbits/s dup=
frame=  106 fps= 48 q=27.0 size=     512kB time=00:00:07.01 bitrate= 598.2kbits/s dup=
frame=  129 fps= 48 q=27.0 size=     512kB time=00:00:08.54 bitrate= 490.9kbits/s dup=
frame=  148 fps= 46 q=27.0 size=     768kB time=00:00:09.82 bitrate= 640.6kbits/s dup=
frame=  168 fps= 45 q=27.0 size=    1024kB time=00:00:11.14 bitrate= 752.8kbits/s dup=
frame=  184 fps= 43 q=27.0 size=    1280kB time=00:00:12.21 bitrate= 858.6kbits/s dup=
frame=  204 fps= 37 q=-1.0 Lsize=    2176kB time=00:00:13.44 bitrate=1325.9kbits/s dup
=0 drop=199 speed=2.43x
video:1948kB audio:216kB subtitle:0kB other streams:0kB global headers:0kB muxing over
head: 0.551796%
```

图 5-67　通过 FFmpeg 控制输出视频的帧率

转码前和转码后的文件通过 MediaInfo 观察，可以发现视频的总时长(Duration)一致，帧率不同。由于转码后的帧率(15)是转码前帧率(30)的一半，所以转码后的文件字节数明显比转码前小了很多，转码后的码率也降低了很多，如图 5-68 所示。另外，读者可以仔细观看，转码后的视频流畅度会降低很多。

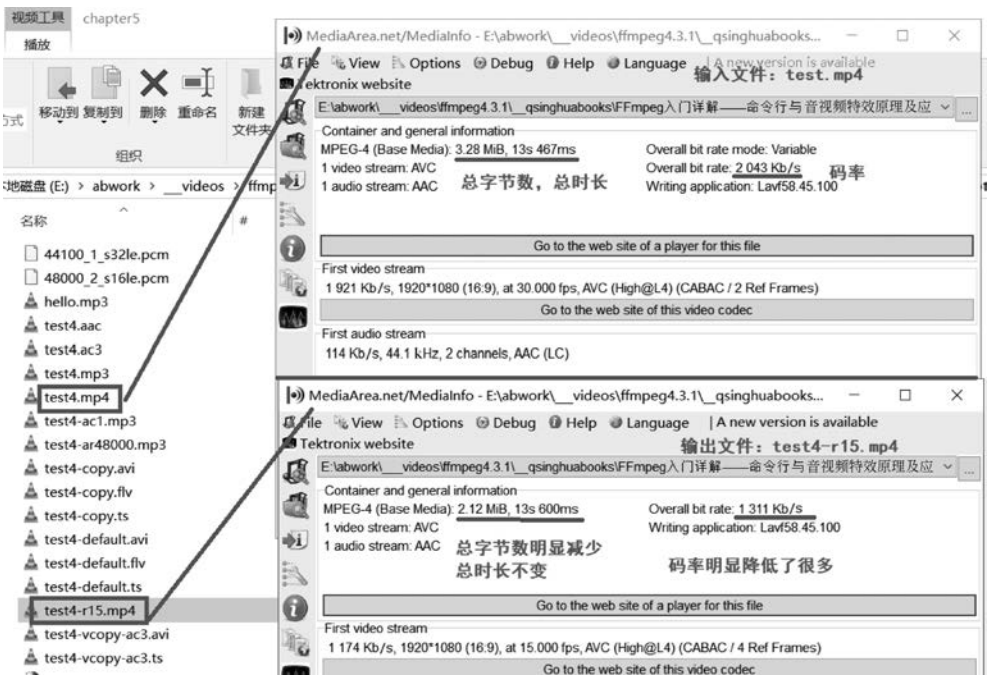

图 5-68　通过 MediaInfo 对比输入、输出视频的帧率

注意：-r 参数放到-i 后边，用于控制转码后的输出视频的帧率。如果放到-i 前边，则用于控制输入视频的帧率。

2. -r 放到-i 前边

使用 FFmpeg 的-r 参数可以控制视频的帧率，放到-i 前边，用于控制转码前视频的输入帧率(注意，可以与原视频本身的帧率不同)，命令如下：

```
ffmpeg -r 15 -i test4.mp4 -y test4-front-r15.mp4
#注意: 原视频本身的帧率是 30,这里控制视频的输入帧率为 15
#那么,转码后视频的总时长(Duration)会增加一倍
```

在该案例中，输入文件 test4.mp4 本来的帧率是 30，而转码前的输入帧率使用-i 前边的-r 15 进行控制，转码过程如图 5-69 所示。由于封装格式是 MP4，所以 FFmpeg 使用 libx264 进行视频转码，使用 AAC 进行音频转码。

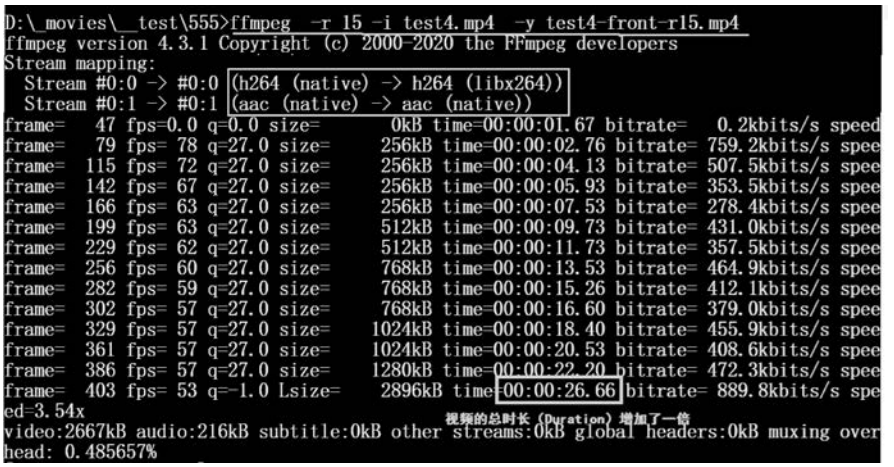

图 5-69 通过 FFmpeg 控制视频的输入帧率

可能读者有一个疑问："-r 放到-i 前用于控制视频的输入帧率，那么转码后的视频帧率是多少呢?"。通过 MediaInfo 观察输出文件的流信息，会发现转码后的视频帧率是 15(注意不是原视频帧率的 30)，如图 5-70 所示。

注意：-r 参数放到-i 前边，用于控制输入视频的帧率。转码后的视频总时长(Duration)增加了一倍，但音频的总时长保持不变(这点读者一定要注意)。

3. -r 既放到-i 前边，又放到-i 后边

使用 FFmpeg 的-r 参数可以控制视频的帧率，可以放到-i 前边，用于控制转码前视频的输入帧率；同时也可以放到-i 后边，用于控制输出视频的帧率，命令如下：

```
ffmpeg -r 15 -i test4.mp4 -r 20 -y test4-front-r15-back-r20.mp4
```

在该案例中，输入文件 test4.mp4 本来的帧率是 30，而转码前的输入帧率使用-i 前边的-r 15 进行控制，转码后的输出帧率通过-i 后边的-r 20 进行控制，转码过程如图 5-71

图 5-70　通过 MediaInfo 查看转码后视频的帧率

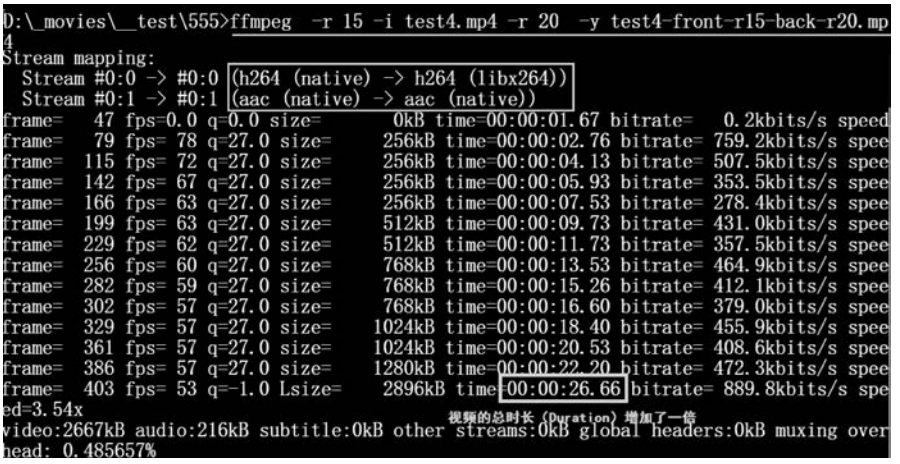

图 5-71　通过 FFmpeg 同时控制输入视频的帧率及转码后视频的帧率

所示。

通过 MediaInfo 观察可知，输出后视频（test4-front-r15-back-r20. mp4）的帧率变成了20，说明-i 后边的-r 20 起到了作用，如图 5-72 所示。

5.6.2　控制视频的码率及分辨率

使用 FFmpeg 的-s 参数可以控制视频的分辨率，格式为-s Width x Height（注意这里的x 是小写的英文字母 x），也可以使用-b:v 参数控制视频的码率。这些参数用于控制输出视频的，所以需要放到-i 后边，命令如下：

```
ffmpeg -i test4.mp4 -s 640x360 -b:v 300k -y test4-640x360-300k.mp4
```

图 5-72　通过 MediaInfo 查看转码后视频的帧率

在该案例中，原视频的分辨率是 1920×1080，码率为 1921Kb/s，转码后的分辨率是 640×360，码率为 255Kb/s(接近 300Kb/s)，如图 5-73 所示。

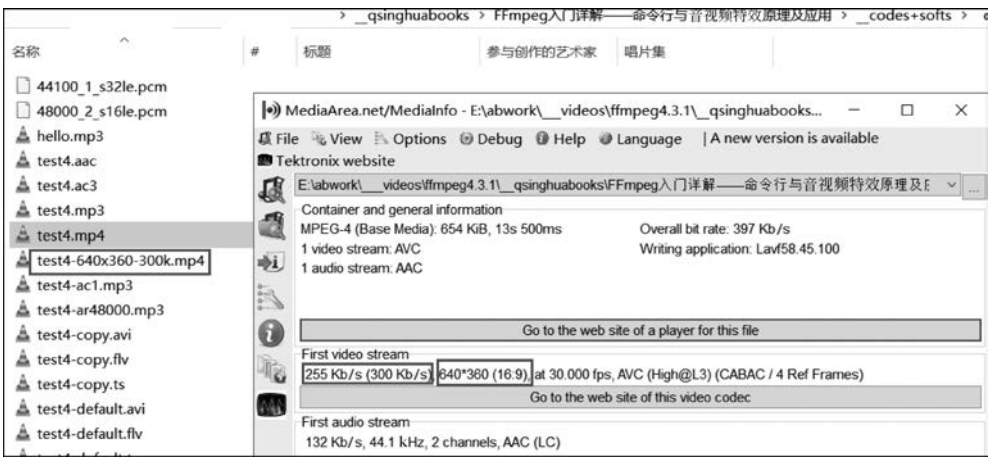

图 5-73　通过 MediaInfo 查看转码后视频的宽、高和码率

码率的计算相对比较简单，bitrate＝file size÷duration，例如一个文件的大小为 20.8MB，时长为 1min，那么，bitrate＝20.8MB÷60s＝20.8×1024×1024×8b÷60s≈2840Kb/s，而一般音频的码率只有固定几种，例如 128Kb/s，则视频码率就是 video bitrate＝2840Kb/s－128Kb/s＝2712Kb/s。

5.6.3　控制视频的 GOP

使用 FFmpeg 的-g 参数可以控制视频的 GOP，放到-i 后边，用于控制转码后的 GOP，命令如下：

```
ffmpeg -i test4-r15.mp4 -r 15 -g 5 -y test4-r15-g5.mp4
```

在该案例中，输入视频文件 test4-r15.mp4 本来的 GOP 是 250，而转码后的 GOP 是 5。可以通过 MediaInfo 的 View 菜单下的 Text 选项来查看 GOP，如图 5-74 所示。

图 5-74　通过 MediaInfo 查看视频的 GOP

5.6.4　视频 GOP 简介

图像组(Group Of Picture，GOP)，指两个 I 帧之间的距离。Reference 即参考周期，指两个 P 帧之间的距离。一个 I 帧所占用的字节数大于一个 P 帧，一个 P 帧所占用的字节数大于一个 B 帧，所以在码率不变的前提下，GOP 值越大，P、B 帧的数量会越多，平均每个 I、P、B 帧所占用的字节数就越多，也就更容易获取较好的图像质量；Reference 越大，B 帧的数量越多，同理也更容易获得较好的图像质量。I、P、B 帧的字节大小为 I ＞ P ＞ B。GOP 解码顺序和显示顺序如图 5-75 所示。

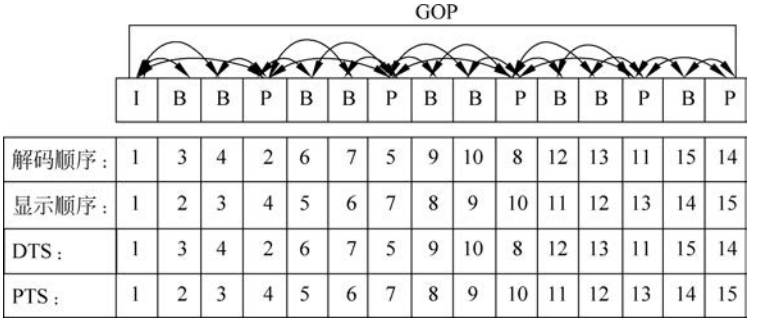

图 5-75　GOP 解码顺序和显示顺序

有时在一些特殊场景下，例如镜头切换等，变化的信息量很大，那么使用 P 帧或者 B 帧反而得不偿失。使用 H.264 编码，这时可以强制插入关键帧(Key Frame)，也就是不依赖前后帧的独立的一帧图像。Key Frame 也叫 I-Frame，也就是 Intra-Frame。理论上，任何时候都可以插入 Key Frame。假设一个视频从头到尾都没有这样的剧烈变化的镜头，那么

就只有第 1 帧是 Key Frame，但是在进行随机播放（Random Seek）时，就很麻烦，例如想从第 1h 开始播放，那么程序就得先解码 1h 的视频，然后才能计算出要播放的帧。针对这种情况，一般以有规律的时间间隔（interval）来插入 Key Frame，这个有规律的 interval 就叫作 I-Frame interval，或者叫作 I-Frame distance，这个值一般是 10 倍的 fps（libx264 默认将这个 interval 设置为 250，另外，libx264 编码器在检测到大的场景变化时，会在变化开始处插入 Key Frame）。直播场景下，GOP 要适当小一些。ESPN 是每 10s 插入一个 Key Frame，YouTube 每 2s 插入一个关键帧，Apple 每 3～10s 插入一个 Key Frame。

GOP 结构一般会使用两个数字来描述：M＝3，N＝12。第 1 个数字 3 表示的是两个 Anchor Frame（I 帧或者 P 帧）之间的距离，第 2 个数字 12 表示两个 Key Frame 之间的距离（GOP size 或者 GOP length），对于这个例子来讲，GOP 结构就是 IBBPBBPBBPBBI。

IDR（Instantaneous Decoder Refresh，以及时刷新帧）Frame 首先是 Key Frame，对于普通的 Key Frame（non-IDR Key Frame）来讲，其后的 P-Frame 和 B-Frame 可以引用此 Key Frame 之前的帧，但是 ID 却不行，IDR 后的 P-Frame 和 B-Frame 不能引用此 IDR 之前的帧，所以当解码器遇到 IDR 后，就必须抛弃之前的解码序列，重新开始。这样当遇到解码错误时，错误不会影响太远，将止步于 IDR。

5.7 libx264 的常用编码选项及应用案例

可以使用 libx264 对视频进行编码，但是需要在编译 FFmpeg 时通过--enable-libx264 将第三方库 libx264 集成进来。例如有一个视频编码格式为 MPEG-4 Video 的文件 test4-default.avi，使用 libx264 对它进行视频转码，转码过程如图 5-76 所示，命令如下：

```
ffmpeg -i test4-default.avi -vcodec libx264 -y test4-libx264-avi.mp4
```

```
D:\_movies\__test\555>ffmpeg -i test4-default.avi -vcodec libx264 -y test4-libx264-av
i.mp4
ffmpeg version 4.3.1 Copyright (c) 2000-2020 the FFmpeg developers
Stream mapping:
  Stream #0:0 -> #0:0 (mpeg4 (native) -> h264 (libx264))
  Stream #0:1 -> #0:1 (mp3 (mp3float) -> aac (native))
[libx264 @ 053ae6c0] frame I:2     Avg QP:19.93  size:115828
[libx264 @ 053ae6c0] frame P:171   Avg QP:19.69  size:  8004
[libx264 @ 053ae6c0] frame B:232   Avg QP:23.18  size:  1580
[libx264 @ 053ae6c0] consecutive B-frames: 20.2%  8.4%  5.2% 66.2%
[libx264 @ 053ae6c0] mb I  I16..4: 26.4% 66.9%  6.7%
[libx264 @ 053ae6c0] mb P  I16..4:  2.0%  3.4%  0.4%  P16..4:  5.8%  1.7%  1.7%  0.0%
 0.0%    skip:84.9%
[libx264 @ 053ae6c0] mb B  I16..4:  0.5%  0.6%  0.0%  B16..8: 11.6%  0.8%  0.1%  direc
t: 0.2%  skip:86.2%  L0:55.6% L1:43.8% BI: 0.6%
[libx264 @ 053ae6c0] 8x8 transform intra:59.1% inter:88.1%
[libx264 @ 053ae6c0] coded y,uvDC,uvAC intra: 27.1% 32.0% 2.8% inter: 1.3% 1.1% 0.1%
[libx264 @ 053ae6c0] i16 v,h,dc,p: 45% 44% 10%  0%
[libx264 @ 053ae6c0] i8 v,h,dc,ddl,ddr,vr,hd,vl,hu: 19% 18% 49%  4%  1%  0%  1%  1%  7
%
[libx264 @ 053ae6c0] i4 v,h,dc,ddl,ddr,vr,hd,vl,hu: 45% 40%  8%  1%  1%  1%  1%  1%  1
%
```

图 5-76　通过 FFmpeg 进行 libx264 编码

5.7.1　FFmpeg 中 libx264 的选项

libx264 有很多选项，例如可以使用-slice-max-size 4 参数来控制最大的条带数(Slice)，此处最大的条带数为 4，命令如下：

```
ffmpeg -i test4-default.avi -vcodec libx264 -r 30 -slice-max-size 4 -y test4-libx264-avi-r30-slice4.mp4
```

H.264 编码可以将一张图片分割成若干条带，Slice 承载固定个数的宏块。将一张图片分割成若干 Slice 的目的是限制误码的扩散和传输。在 H.264 编码协议中规定当前帧的当前 Slice 片内宏块不允许参考其他 Slice 的宏块。H.264 的视频序列、帧、条带、宏块及像素的关系如图 5-77 所示。

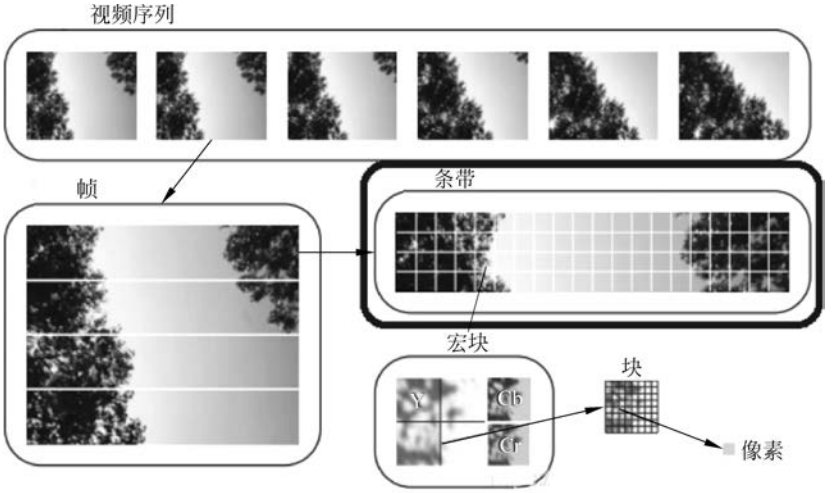

图 5-77　H.264 的视频序列、帧、条带、宏块及像素的关系

FFmpeg 中可用的 libx264 的所有选项可以通过 cmd 窗口进行查看，命令如下：

```
ffmpeg -h encoder=libx264
```

这些是 FFmpeg 命令行中可以使用的选项名，有几个选项非常重要，包括-preset、-tune、-profile、-level 等，具体的输出信息如下：

```
//chapter5/ffmpeg-libx264-help.txt
Encoder libx264 [libx264 H.264 / AVC / MPEG-4 AVC / MPEG-4 part 10]:
    General capabilities: delay threads
```

```
    Threading capabilities: auto
    Supported pixel formats: yuv420p yuvj420p yuv422p yuvj422p yuv444p yuvj444p nv12 nv16
nv21 yuv420p10le yuv422p10le yuv444p10le nv20le gray gray10le
libx264 AVOptions:
  -preset           <string>    E..V..... Set the encoding preset(cf. x264 --fullhelp)
(default "medium")#非常重要
  -tune             <string>    E..V..... Tune the encoding params(cf. x264 --fullhelp)
                                                                              #非常重要
  -profile          <string>    E..V..... Set profile restrictions(cf. x264 --fullhelp)
                                                                              #非常重要
  -fastfirstpass    <boolean>   E..V..... Use fast settings when encoding first pass(default
true)
  -level            <string>    E..V..... Specify level(as defined by Annex A)  #非常重要
  -passlogfile      <string>    E..V..... Filename for 2 pass stats
  -wpredp           <string>    E..V..... Weighted prediction for P-frames
  -a53cc            <boolean>   E..V..... Use A53 Closed Captions(if available)(default
true)
  -x264opts         <string>    E..V..... x264 options
  -crf              <float>     E..V..... Select the quality for constant quality mode(from
-1 to FLT_MAX)(default -1)
  -crf_max          <float>     E..V..... In CRF mode, prevents VBV from lowering quality
beyond this point.(from -1 to FLT_MAX)(default -1)
  -qp               <int>       E..V..... Constant quantization parameter rate control
method(from -1 to INT_MAX)(default -1)
  -aq-mode          <int>       E..V..... AQ method(from -1 to INT_MAX)(default -1)
     none           0           E..V.....
     variance       1           E..V..... Variance AQ(complexity mask)
     autovariance   2           E..V..... Auto-variance AQ
     autovariance-biased  3     E..V..... Auto-variance AQ with bias to dark scenes
  -aq-strength      <float>     E..V..... AQ strength. Reduces blocking and blurring in flat
and textured areas.(from -1 to FLT_MAX)(default -1)
  -psy              <boolean>   E..V..... Use psychovisual optimizations.(default auto)
  -psy-rd           <string>    E..V..... Strength of psychovisual optimization, in <psy-rd>:
<psy-trellis> format.
  -rc-lookahead     <int>       E..V..... Number of frames to look ahead for frametype and
ratecontrol(from -1 to INT_MAX)(default -1)
  -weightb          <boolean>   E..V..... Weighted prediction for B-frames.(default auto)
  -weightp          <int>       E..V..... Weighted prediction analysis method.(from -1 to
INT_MAX)(default -1)
     none           0           E..V.....
     simple         1           E..V.....
     smart          2           E..V.....
  -ssim             <boolean>   E..V..... Calculate and print SSIM stats.(default auto)
  -intra-refresh    <boolean>   E..V..... Use Periodic Intra Refresh instead of IDR frames.
(default auto)
```

```
  -bluray-compat      <boolean>   E..V..... Bluray compatibility workarounds. (default auto)
  -b-bias             <int>       E..V..... Influences how often B-frames are used(from INT_
MIN to INT_MAX)(default INT_MIN)
  -b-pyramid          <int>       E..V..... Keep some B-frames as references. (from -1 to INT
_MAX)(default -1)
     none             0           E..V.....
     strict           1           E..V..... Strictly hierarchical pyramid
     normal           2           E..V..... Non-strict(not Blu-ray compatible)
  -mixed-refs         <boolean>   E..V..... One reference per partition,as opposed to one
reference per macroblock(default auto)
  -8x8dct             <boolean>   E..V..... High profile 8x8 transform. (default auto)
  -fast-pskip         <boolean>   E..V.....(default auto)
  -aud                <boolean>   E..V..... Use access unit delimiters. (default auto)
  -mbtree             <boolean>   E..V..... Use macroblock tree ratecontrol. (default auto)
  -deblock            <string>    E..V..... Loop filter parameters,in <alpha: beta> form.
  -cplxblur           <float>     E..V..... Reduce fluctuations in QP(before curve
compression)(from -1 to FLT_MAX)(default -1)
  -partitions         <string>    E..V..... A comma-separated list of partitions to consider.
Possible values: p8x8,p4x4,b8x8,i8x8,i4x4,none,all
  -direct-pred        <int>       E..V..... Direct MV prediction mode(from -1 to INT_MAX)
(default -1)
     none             0           E..V.....
     spatial          1           E..V.....
     temporal         2           E..V.....
     auto             3           E..V.....
  -slice-max-size     <int>       E..V..... Limit the size of each slice in Bytes(from -1 to
INT_MAX)(default -1)
  -stats              <string>    E..V..... Filename for 2 pass stats
  -nal-hrd            <int>       E..V..... Signal HRD information(requires vbv-bufsize;
cbr not allowed in .mp4)(from -1 to INT_MAX)(default -1)
     none             0           E..V.....
     vbr              1           E..V.....
     cbr              2           E..V.....
  -avcintra-class     <int>       E..V..... AVC-Intra class 50/100/200(from -1 to 200)
(default -1)
  -me_method          <int>       E..V..... Set motion estimation method(from -1 to 4)
(default -1)
     dia              0           E..V.....
     hex              1           E..V.....
     umh              2           E..V.....
     esa              3           E..V.....
     tesa             4           E..V.....
  -motion-est         <int>       E..V..... Set motion estimation method(from -1 to 4)
(default -1)
     dia              0           E..V.....
     hex              1           E..V.....
```

```
      umh              2              E..V.....
      esa              3              E..V.....
      tesa             4              E..V.....
   -forced-idr         <boolean>      E..V..... If forcing keyframes,force them as IDR frames.
(default false)
   -coder              <int>          E..V..... Coder type(from -1 to 1)(default default)
      default          -1             E..V.....
      cavlc            0              E..V.....
      cabac            1              E..V.....
      vlc              0              E..V.....
      ac               1              E..V.....
   -b_strategy         <int>          E..V..... Strategy to choose between I/P/B-frames(from
-1 to 2)(default -1)
   -chromaoffset       <int>          E..V..... QP difference between chroma and luma(from INT_
MIN to INT_MAX)(default -1)
   -sc_threshold       <int>          E..V..... Scene change threshold(from INT_MIN to INT_MAX)
(default -1)
   -noise_reduction    <int>          E..V..... Noise reduction(from INT_MIN to INT_MAX)
(default -1)
   -x264-params        <dictionary>   E..V..... Override the x264 configuration using a : -separated
list of key=value parameters
```

5.7.2 x264.exe中的选项名与选项值

libx264中的这些选项值需要通过x264.exe来查看,部分截图如5-78所示,命令行如下:

```
x264.exe --help
x264.exe --fullhelp
```

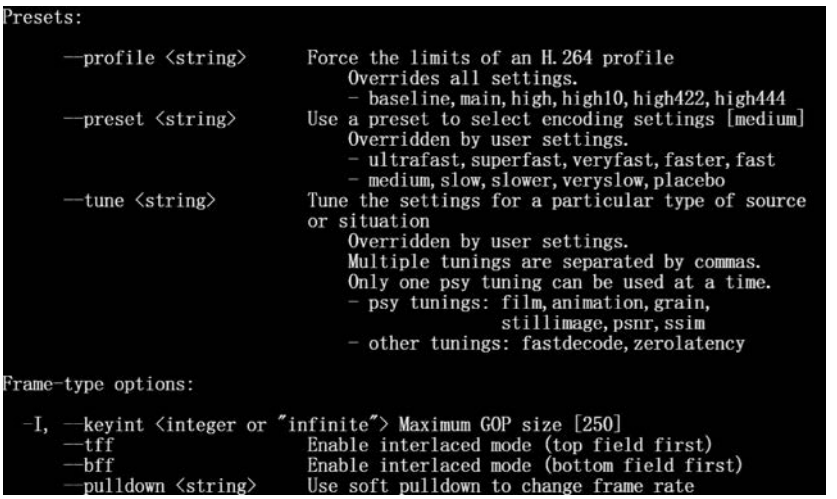

图5-78 x264中的参数选项部分截图

注意，输入 x264.exe --fullhelp 可以查看所有的详细参数值，例如-profile 的选项值可以是 baseline、main、high、high10、high422 及 high444 等，所以 FFmpeg 在转码时可以使用的命令如下：

```
//chapter5/x264 - encode.txt
ffmpeg - i test4 - default.avi - vcodec libx264 - r 30 - profile: v high - y test4 - libx264 -
avi - r30 - slice4.mp4
# 也可以用 baseline、main 等
```

输入 x264.exe --fullhelp 后，部分输出信息如下：

```
//chapter5/x264 - encode.txt
Presets:

      -- profile < string > Force the limits of an H.264 profile
                                  Overrides all settings.
                                  - baseline:
                                    -- no - 8x8dct -- bframes 0 -- no - cabac
                                    -- cqm flat -- weightp 0
                                    No interlaced.
                                    No lossless.
                                  - main:
                                    -- no - 8x8dct -- cqm flat
                                    No lossless.
                                  - high:
                                    No lossless.
                                  - high10:
                                    No lossless.
                                    Support for bit depth 8 - 10.
                                  - high422:
                                    No lossless.
                                    Support for bit depth 8 - 10.
                                    Support for 4: 2: 0/4: 2: 2 chroma subsampling.
                                  - high444:
                                    Support for bit depth 8 - 10.
                                    Support for 4: 2: 0/4: 2: 2/4: 4: 4 chroma subsampling.
      -- preset < string > Use a preset to select encoding settings [medium]
                                  Overridden by user settings.
                                  - ultrafast:
                                    -- no - 8x8dct -- aq - mode 0 -- b - adapt 0
                                    -- bframes 0 -- no - cabac -- no - deblock
                                    -- no - mbtree -- me dia -- no - mixed - refs
                                    -- partitions none -- rc - lookahead 0 -- ref 1
```

```
                                    -- scenecut 0 -- subme 0 -- trellis 0
                                    -- no-weightb -- weightp 0
                                  - superfast:
                                    -- no-mbtree -- me dia -- no-mixed-refs
                                    -- partitions i8x8,i4x4 -- rc-lookahead 0
                                    -- ref 1 -- subme 1 -- trellis 0 -- weightp 1
                                  - veryfast:
                                    -- no-mixed-refs -- rc-lookahead 10
                                    -- ref 1 -- subme 2 -- trellis 0 -- weightp 1
                                  - faster:
                                    -- no-mixed-refs -- rc-lookahead 20
                                    -- ref 2 -- subme 4 -- weightp 1
                                  - fast:
                                    -- rc-lookahead 30 -- ref 2 -- subme 6
                                    -- weightp 1
                                  - medium:
                                    Default settings apply.
                                  - slow:
                                    -- direct auto -- rc-lookahead 50 -- ref 5
                                    -- subme 8 -- trellis 2
                                  - slower:
                                    -- b-adapt 2 -- direct auto -- me umh
                                    -- partitions all -- rc-lookahead 60
                                    -- ref 8 -- subme 9 -- trellis 2
                                  - veryslow:
                                    -- b-adapt 2 -- bframes 8 -- direct auto
                                    -- me umh -- merange 24 -- partitions all
                                    -- ref 16 -- subme 10 -- trellis 2
                                    -- rc-lookahead 60
                                  - placebo:
                                    -- bframes 16 -- b-adapt 2 -- direct auto
                                    -- slow-firstpass -- no-fast-pskip
                                    -- me tesa -- merange 24 -- partitions all
                                    -- rc-lookahead 60 -- ref 16 -- subme 11
                                    -- trellis 2
-- tune <string>    Tune the settings for a particular type of source
                          or situation
                               Overridden by user settings.
                               Multiple tunings are separated by commas.
                               Only one psy tuning can be used at a time.
                               - film(psy tuning):
                                 -- deblock -1:-1 -- psy-rd <unset>:0.15
                               - animation(psy tuning):
                                 -- bframes {+2} -- deblock 1:1
                                 -- psy-rd 0.4:<unset> -- aq-strength 0.6
```

```
                                    --ref {Double if > 1 else 1}
                                  - grain(psy tuning):
                                    --aq-strength 0.5 --no-dct-decimate
                                    --deadzone-inter 6 --deadzone-intra 6
                                    --deblock -2:-2 --ipratio 1.1
                                    --pbratio 1.1 --psy-rd <unset>: 0.25
                                    --qcomp 0.8
                                  - stillimage(psy tuning):
                                    --aq-strength 1.2 --deblock -3:-3
                                    --psy-rd 2.0: 0.7
                                  - psnr(psy tuning):
                                    --aq-mode 0 --no-psy
                                  - ssim(psy tuning):
                                    --aq-mode 2 --no-psy
                                  - fastdecode:
                                    --no-cabac --no-deblock --no-weightb
                                    --weightp 0
                                  - zerolatency:
                                    --bframes 0 --force-cfr --no-mbtree
                                    --sync-lookahead 0 --sliced-threads
                                    --rc-lookahead 0
    --slow-firstpass      Don't force these faster settings with --pass 1:
                                  --no-8x8dct --me dia --partitions none
                                  --ref 1 --subme {2 if > 2 else unchanged}
                                  --trellis 0 --fast-pskip
```

5.8　libx265 的常用编码选项及应用案例

可以使用 libx265 对视频进行编码，但是需要在编译 FFmpeg 时通过--enable-libx265 参数将第三方库 libx265 集成进来。例如对于一个视频编码格式为 H.264 的文件(test4.mp4)使用 libx265 进行转码，过程如图 5-79 所示，命令如下：

```
ffmpeg -i test4.mp4 -vcodec libx265 -acodec copy -y test4-libx265.mp4
```

在该案例中，使用的视频转码格式是 HEVC(H.264)，转码器是开源的 libx265，非常耗费 CPU，使用率接近 100%，如图 5-80 所示。转码速度比较慢，但转码后的视频文件变小了很多(大约为源 H.264 文件的一半)，这是因为 H.265 的压缩效率比 H.264 高很多。

注意：由于 libx265 是纯软编，所以 CPU 耗费得非常多(几乎为 100%)，但几乎没有用到 GPU(该案例中大约为 1%)。

```
D:\_movies\__test\555>ffmpeg  -i test4.mp4 -vcodec libx265  -acodec copy -y test4-libx
265.mp4
Stream mapping:
  Stream #0:0 -> #0:0 (h264 (native) -> hevc (libx265))   视频转码
  Stream #0:1 -> #0:1 (copy)
Press [q] to stop, [?] for help
x265 [info]: HEVC encoder version 3.4+2-73ca1d7be377
x265 [info]: build info [Windows][GCC 10.2.1][32 bit] 8bit
x265 [info]: using cpu capabilities: MMX2 SSE2Fast LZCNT SSSE3 SSE4.2 AVX FMA3 BMI2 AV
X2
x265 [info]: Main profile, Level-4 (Main tier)
x265 [info]: Thread pool created using 4 threads
x265 [info]: Slices                              : 1
x265 [info]: frame threads / pool features       : 2 / wpp(17 rows)
x265 [info]: Coding QT: max CU size, min CU size : 64 / 8
x265 [info]: Residual QT: max TU size, max depth : 32 / 1 inter / 1 intra
x265 [info]: ME / range / subpel / merge         : hex / 57 / 2 / 3
x265 [info]: Keyframe min / max / scenecut / bias  : 25 / 250 / 40 / 5.00
x265 [info]: Lookahead / bframes / badapt        : 20 / 4 / 2
x265 [info]: b-pyramid / weightp / weightb       : 1 / 1 / 0
x265 [info]: References / ref-limit  cu / depth  : 3 / off / on
x265 [info]: AQ: mode / str / qg-size / cu-tree  : 2 / 1.0 / 32 / 1
x265 [info]: Rate Control / qCompress            : CRF-28.0 / 0.60
x265 [info]: tools: rd=3 psy-rd=2.00 early-skip rskip mode=1 signhide tmvp
```

图 5-79　使用 FFmpeg 进行 libx265 编码

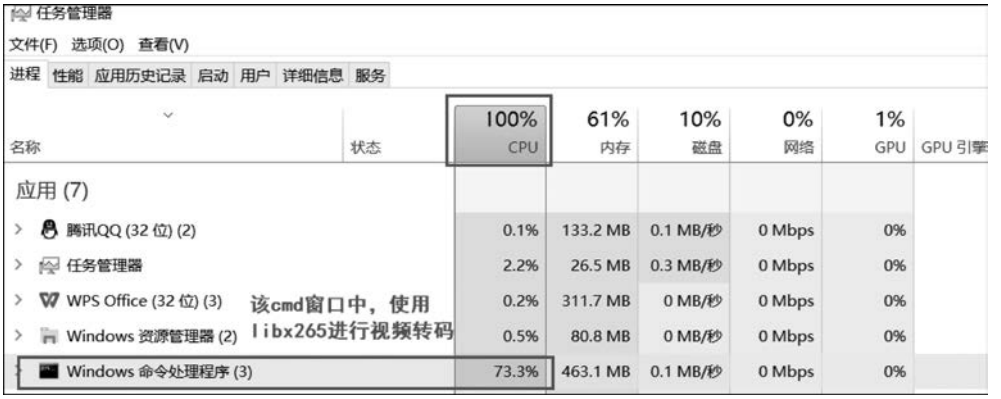

图 5-80　libx265 编码的 CPU 占用率

FFmpeg 中可用的 libx265 的所有选项可以通过 cmd 窗口进行查看，命令如下：

```
ffmpeg -h encoder=libx265
```

这些是 FFmpeg 命令行中可以使用的选项名，输出信息如下：

```
//chapter5/x265-help.txt
Encoder libx265 [libx265 H.265 / HEVC]:
    General capabilities: delay threads
    Threading capabilities: auto
    Supported pixel formats: yuv420p yuvj420p yuv422p yuvj422p yuv444p yuvj444p gbrp gray
##这些是支持的输入像素格式
libx265 AVOptions: ##这些是支持的参数选项名
```

```
  -crf              <float>        E..V..... set the x265 crf(from -1 to FLT_MAX)(de
fault -1)
  -qp               <int>          E..V..... set the x265 qp(from -1 to INT_MAX)(def
ault -1)
  -forced-idr       <boolean>      E..V..... if forcing keyframes,force them as IDR f
rames(default false)
  -preset           <string>       E..V..... set the x265 preset
  -tune             <string>       E..V..... set the x265 tune parameter
  -profile          <string>       E..V..... set the x265 profile
  -x265-params      <dictionary>   E..V..... set the x265 configuration using a : -sepa
rated list of key=value parameters
```

5.9　FFmpeg 的 GPU 硬件加速原理及应用案例

FFmpeg 的软编功能非常强大，并且兼容性很好，但是太耗费 CPU，使用硬件加速可以充分利用 GPU，从而将 CPU 释放出来。可以查看 FFmpeg 支持的硬件加速，命令如下：

```
ffmpeg -hwaccels
```

笔者安装的 FFmpeg 的输出信息如图 5-81 所示。

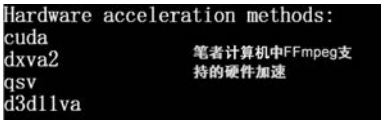

图 5-81　FFmpeg 支持的硬件加速方式

其中，FFmpeg 的编译选项如下：

```
//chapter5/ffmpeg-hwaccel-config.txt
ffmpeg version 4.3.1 Copyright(c) 2000-2020 the FFmpeg developers
  built with gcc 10.2.1(GCC) 20200726
  configuration: --enable-gpl --enable-version3 --enable-sdl2 --enable-
fontconfig --en
able-gnutls --enable-iconv --enable-libass --enable-libdav1d --enable-libbluray
 --enab
le-libfreetype --enable-libmp3lame --enable-libopencore-amrnb --enable-
libopencore-amr
wb --enable-libopenjpeg --enable-libopus --enable-libshine --enable-libsnappy --
enable
 -libsoxr --enable-libsrt --enable-libtheora --enable-libtwolame --enable-libvpx
 --enab
```

```
le - libwavpack -- enable - libwebp -- enable - libx264 -- enable - libx265 -- enable - libxml2 -- en
able - libzimg -- enable - lzma -- enable - zlib -- enable - gmp -- enable - libvidstab -- enable - lib
vmaf -- enable - libvorbis -- enable - libvo - amrwbenc -- enable - libmysofa -- enable - libspeex -
 - enable - libxvid -- enable - libaom -- enable - libgsm -- disable - w32threads -- enable - libmfx -
 - enable - ffnvcodec -- enable - CUDA - llvm -- enable - cuvid -- enable - d3d11va -- enable - nvenc --
enable - nvdec -- enable - dxva2 -- enable - avisynth -- enable - libopenmpt -- enable - amf
  libavutil 56. 51.100 / 56. 51.100
```

注意：FFmpeg 中默认的编译选项不支持硬件加速，编译前需要手工开启这些选项。

通过 FFmpeg 可以使用完全硬件加速方式，即解码和编码都使用硬件加速，例如解码 H.264 格式的视频可以使用 cuvid，编码 H.264 格式的视频可以使用 nvenc，这里需要在-i 前添加 -hwaccel cuvid 参数，这种情况就可以完全通过显卡 GPU 完成转码，过程如图 5-82 所示，命令如下：

```
//chapter5/ffmpeg - hwaccel - config.txt
ffmpeg - hwaccel cuvid - c: v h264_cuvid - i test4.mp4 - c: v h264_nvenc - preset medium - crf 28 - c: a copy - y test4 - cuvid - nvenc.mp4
```

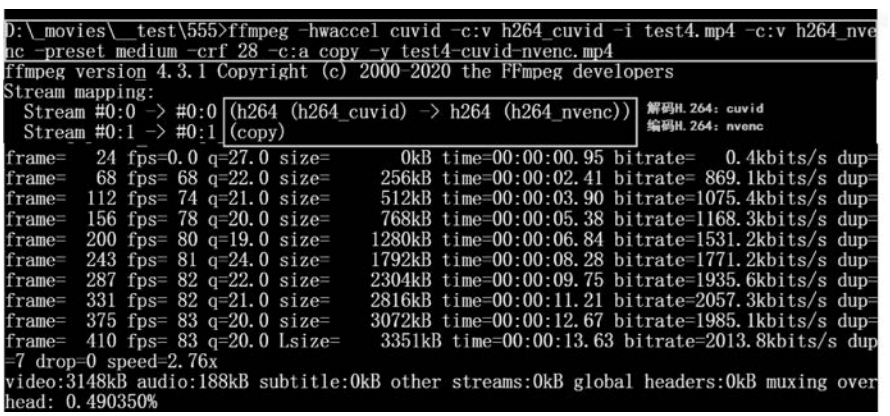

图 5-82　FFmpeg 以完全硬件加速方式实现 H.264 的解码与编码

在该案例中，视频转码分为解码 H.264(cuvid)和重新编码 H.264(nvenc)，几乎没用到 CPU，而 GPU 的占用率几乎是 100%，如图 5-83 所示。由此可见，FFmpeg 使用硬件加速后充分利用了 GPU。

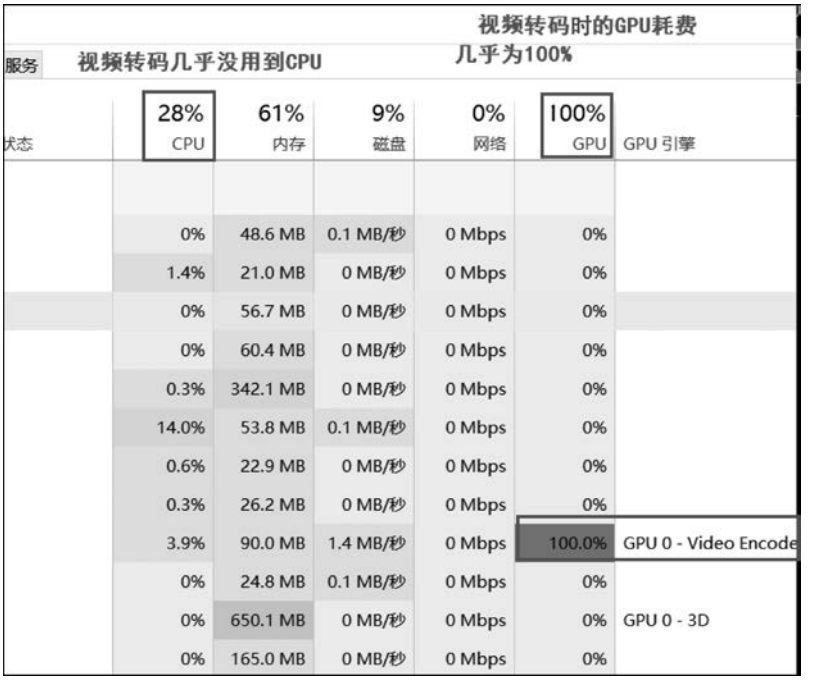

图 5-83　FFmpeg 硬件加速的 GPU 占用率