

目 录

1. 文档介绍	3
1.1 文档目的	3
1.2 文档范围	3
1.3 读者对象	3
1.4 参考文献	3
1.5 术语与缩写解释	3
2. FFMPEG 支持能力说明	4
2.1 FFMPEG 介绍及安装	4
2.1.1 FFmpeg 简介	4
2.1.2 FFmpeg 安装	4
2.2 FFMPEG 参数说明	4
2.2.1 通用选项	4
2.2.2 视频选项	5
2.2.3 高级视频选项	5
2.2.4 音频选项	6
2.2.5 音频/视频捕获选项	7
2.2.6 高级选项	7
2.2.7 FFmpeg 参数实例	7
2.3 FFMPEG 支持能力说明	8
2.3.1 FFmpeg 对编码解码器的支持	8
2.3.2 FFmpeg 对容器格式的支持	8
2.3.3 FFmpeg 对过滤器的支持	8
2.3.4 FFmpeg 对图像颜色空间的支持	9
2.4 FFMPEG 功能及使用说明	10
2.4.1 ffmpeg 对多媒体的支持能力验证	10
2.4.2 FFmpeg 格式转换	11
2.4.3 FFmpeg 视频截图	13
2.4.4 FFmpeg 屏幕录制	14
2.4.5 FFmpeg 音视频采集	14
2.5 FFMPEG 应用实例	14
2.5.1 用 FFserver 从文件生成流媒体	14
2.5.2 用 FFserver 从设备生成实时流	15
3.FFMPEG 架构	16
3.1 FFMPEG 文件结构	16
3.2 I\O 模块分析	17
3.2.1 概述	17

3.2.2 相关数据结构介绍	18
3.3 DEMUXER 和 MUXER 模块分析	23
3.3.1 概述	23
3.3.2 相关数据结构介绍	23
3.4 DECODER/ENCODER 模块	26
3.4.1 概述	26
3.4.2 相关数据结构的初始化	26
3.5 其他重要数据结构的初始化	27
3.5.1 AVStream	27
3.5.2 AVInputStream/ AVOutputStream	28
3.5.3 AVPacket	28
4.FFMPEG 裁剪说明	28
4.1 CONFIGURE 参数	28
4.1.1 通用选项	28
4.1.2 基本选项介绍	32
4.2 FFMPEG 裁剪优化实例	34
4.3 裁剪优化前后文件比较	35

1. 文档介绍

1.1 文档目的

整理出开源代码 ffmpeg 的资料，方便公司同事后续使用。

1.2 文档范围

较为详细的介绍 ffmpeg 的功能、使用以及二次开发。

1.3 读者对象

希望了解 ffmpeg 知识，从事 USM 及 IPTV 的同事。

1.4 参考文献

1.5 术语与缩写解释

缩略语/术语	全 称	说 明
ffmpeg	Fast forward mpeg	音视频转换器
ffplay	Fast forward play	用 ffmpeg 实现的播放器
ffserver	Fast forward server	用 ffmpeg 实现的 rstp 服务器
ffprobe	Fast forward probe	用来输入分析输入流。

2. FFmpeg 支持能力说明

2.1 FFmpeg 介绍及安装

2.1.1 FFmpeg 简介

FFmpeg 是一个开源免费跨平台的视频和音频流方案，属于自由软件，采用 LGPL 或 GPL 许可证（依据你选择的组件）。它提供了录制、转换以及流化音视频的完整解决方案。它包含了非常先进的音频/视频编解码库 libavcodec，为了保证高可移植性和编解码质量，libavcodec 里很多 codec 都是从头开发的。

ffmpeg 项目由以下几部分组成：

1. ffmpeg 视频文件转换命令行工具，也支持经过实时电视卡抓取和编码成视频文件。
2. ffserver 基于 HTTP、RTSP 用于实时广播的多媒体服务器。也支持时间平移
3. ffmpeg 用 SDL 和 FFmpeg 库开发的一个简单的媒体播放器
4. libavcodec 一个包含了所有 FFmpeg 音视频编解码器的库。为了保证最优性能和高可复用性，大多数编解码器从头开发的。
5. libavformat 一个包含了所有的普通音视频格式的解析器和产生器的库

2.1.2 FFmpeg 安装

1. 将所有源代码压缩在一个文件夹中，例如/绝对路径/ffmpeg。
2. 在终端输入以下指令：

```
Cd /绝对路径/ffmpeg
```

```
./configure (此时，会出现问题。然后重新输入./configure --disable-yasm-)
```

```
Make
```

至此，ffmpeg 安装编译通过，可以进行对音视频的操作。

ffmpeg 的编译需要依赖于 SDL 库，所以要想编译成功 ffmpeg，必须先安装 SDL 库，安装方法：下载最新版本的 SDL 相应版本的 SDL 源码，编译，即可生成 SDL 库。

2.2 FFmpeg 参数说明

2.2.1 通用选项

-L license

-h 帮助

-fromats 显示可用的格式，编解码的，协议的。

-f fmt 强迫采用格式 fmt

-i filename 输入文件

-y 覆盖输出文件

-t duration 设置纪录时间 hh:mm:ss[.xxx]格式的纪录时间也支持

-ss position 搜索到指定的时间 [-]hh:mm:ss[.xxx]的格式也支持

-title string 设置标题

-author string 设置作者

-copyright string 设置版权

-comment string 设置评论
-target type 设置目标文件类型(vcd, svcd, dvd) 所有的格式选项 (比特率, 编解码以及缓冲区大小) 自动设置, 只需要输入如下的就可以了:
ffmpeg -i myfile.avi -target vcd /tmp/vcd.mpg
-hq 激活高质量设置
-itsoffset offset 设置以秒为基准的时间偏移, 该选项影响所有后面的输入文件。该偏移被加到输入文件的时戳, 定义一个正偏移意味着相应的流被延迟了 offset 秒。
[-]hh:mm:ss[.xxx]的格式也支持

2.2.2 视频选项

-b bitrate 设置比特率, 缺省 200kb/s
-r fps 设置帧频 缺省 25
-s size 设置帧大小 格式为 WxH 缺省 160x128.下面的简写也可以直接使用:
Sqcif 128x96 qcif 176x144 cif 252x288 4cif 704x576
-aspect aspect 设置纵横比 4:3 16:9 或 1.3333 1.7777
-croptop size 设置顶部切除带大小 像素单位
-cropbottom size -cropleft size -cropright size 底部, 左边, 右边切除带大小。
-padtop size 设置顶部补齐的大小 像素单位
-padbottom size -padleft size -padright size -padcolor color 设置补齐条大小和颜色(hex, 6 个 16 进制的数, 红:绿:蓝排列, 比如 000000 代表黑色)
-vn 不做视频记录
-bt tolerance 设置视频码率容忍度 kbit/s
-maxrate bitrate 设置最大视频码率容忍度
-minrate bitrate 设置最小视频码率容忍度
-bufsize size 设置码率控制缓冲区大小
-vcodec codec 强制使用 codec 编解码方式。 如果用 copy 表示原始编解码数据必须被拷贝。
-sameq 使用同样视频质量作为源 (VBR)
-pass n 选择处理遍数 (1 或者 2)。两遍编码非常有用。第一遍生成统计信息, 第二遍生成精确的请求的码率
-passlogfile file 选择两遍的纪录文件名为 file

2.2.3 高级视频选项

-g gop_size 设置图像组大小
-intra 仅适用帧内编码
-qscale q 使用固定的视频量化标度(VBR)
-qmin q 最小视频量化标度(VBR)
-qmax q 最大视频量化标度(VBR)
-qdiff q 量化标度间最大偏差 (VBR)
-qblur blur 视频量化标度柔化(VBR)
-qcomp compression 视频量化标度压缩(VBR)
-rc_init_cplx complexity 一遍编码的初始复杂度

-b_qfactor factor 在 p 和 b 帧间的 qp 因子
 -i_qfactor factor 在 p 和 i 帧间的 qp 因子
 -b_qoffset offset 在 p 和 b 帧间的 qp 偏差
 -i_qoffset offset 在 p 和 i 帧间的 qp 偏差
 -rc_eq equation 设置码率控制方程 默认 tex^qComp
 -rc_override override 特定间隔下的速率控制重载
 -me method 设置运动估计的方法 可用方法有 zero phods log x1 epzs(缺省) full
 -dct_algo algo 设置 dct 的算法 可用的有 0 FF_DCT_AUTO 缺省的 DCT 1
 FF_DCT_FASTINT 2 FF_DCT_INT 3 FF_DCT_MMX 4 FF_DCT_MLIB 5
 FF_DCT_ALTIVEC
 -idct_algo algo 设置 idct 算法。可用的有 0 FF_IDCT_AUTO 缺省的 IDCT 1
 FF_IDCT_INT 2 FF_IDCT_SIMPLE 3 FF_IDCT_SIMPLEMMX 4
 FF_IDCT_LIBMPEG2MMX 5 FF_IDCT_PS2 6 FF_IDCT_MLIB 7 FF_IDCT_ARM 8
 FF_IDCT_ALTIVEC 9 FF_IDCT_SH4 10 FF_IDCT_SIMPLEARM
 -er n 设置错误残留为 n 1 FF_ER_CAREFULL 缺省 2 FF_ER_COMPLIANT 3
 FF_ER_AGGRESSIVE 4 FF_ER_VERY_AGGRESSIVE
 -ec bit_mask 设置错误掩蔽为 bit_mask，该值为如下值的位掩码 1
 FF_EC_GUESS_MVS (default=enabled) 2 FF_EC_DEBLOCK (default=enabled)
 -bf frames 使用 frames B 帧，支持 mpeg1，mpeg2，mpeg4
 -mbd mode 宏块决策 0 FF_MB_DECISION_SIMPLE 使用 mb_cmp 1
 FF_MB_DECISION_BITS 2 FF_MB_DECISION_RD
 -4mv 使用 4 个运动矢量 仅用于 mpeg4
 -part 使用数据划分 仅用于 mpeg4
 -bug param 绕过没有被自动监测到编码器的问题
 -strict strictness 跟标准的严格性
 -aic 使能高级帧内编码 h263+
 -umv 使能无限运动矢量 h263+
 -deinterlace 不采用交织方法
 -interlace 强迫交织法编码 仅对 mpeg2 和 mpeg4 有效。当你的输入是交织的并且你
 想要保持交织以最小图像损失的时候采用该选项。可选的方法是不交织，但是损失
 更大
 -psnr 计算压缩帧的 psnr
 -vstats 输出视频编码统计到 vstats_hhmmss.log
 -vhook module 插入视频处理模块 module 包括了模块名和参数，用空格分开

2.2.4 音频选项

-ab bitrate 设置音频码率
 -ar freq 设置音频采样率
 -ac channels 设置通道 缺省为 1
 -an 不使能音频纪录
 -acodec codec 使用 codec 编解码

2.2.5 音频/视频捕获选项

- vd device 设置视频捕获设备。比如/dev/video0
- vc channel 设置视频捕获通道 DV1394 专用
- tvstd standard 设置电视标准 NTSC PAL(SECAM)
- dv1394 设置 DV1394 捕获
- av device 设置音频设备 比如/dev/dsp

2.2.6 高级选项

- map file:stream 设置输入流映射
- debug 打印特定调试信息
- benchmark 为基准测试加入时间
- hex 倾倒每一个输入包
- bitexact 仅使用位精确算法 用于编解码测试
- ps size 设置包大小，以 bits 为单位
- re 以本地帧频读数据，主要用于模拟捕获设备
- loop 循环输入流。只工作于图像流，用于 ffserver 测试

2.2.7 FFmpeg 参数实例

[illegible]

-vcodec xvid 使用 xvid 压缩

2.3 FFmpeg 支持能力说明

2.3.1 FFmpeg 对编解码器的支持

ffmpeg 支持的编解码器种类共有 280 多种，涵盖了几乎所有常见音视频编码格式，能解码几乎所有的音视频，每种音视频编解码器的实现都在 libavcodec 目录下有具体的 C 语言实现，具体的支持情况详见附件：



ffmpeg支持的编解码器

附件：

注：编码器和解码器的名称不是完全匹配的，因此有些编码器没有对应相同名称的解码器，反之，解码器也一样。即使编码和解码都支持也不一定是完全对应的，例如 h263 解码器对应有 h263p 和 h263 编码器。

2.3.2 FFmpeg 对容器格式的支持

ffmpeg 支持对绝大多数的容器格式的读写操作，共计 190 多种，涵盖了互联网上各种常见媒体格式及日常生活中及专业应用中的各种媒体格式。详细的支持情况详见附件。



FFmpeg支持的媒体文件格式

附件：

2.3.3 FFmpeg 对过滤器的支持

Filters	说明
aformat	Convert the input audio to one of the specified formats.
anull	Pass the source unchanged to the output.
aresample	Resample audio data.
ashowinfo	Show textual information for each audio frame.
abuffer	Buffer audio frames, and make them accessible to the filterchain.
anullsrc	Null audio source, never return audio frames.
abuffersink	Buffer audio frames, and make them available to the end of the filter graph.
anullsink	Do absolutely nothing with the input audio.
copy	Copy the input video unchanged to the output.
crop	Crop the input video to width:height:x:y.
drawbox	Draw a colored box on the input video.
fade	Fade in/out input video
fieldorder	Set the field order.
fifo	Buffer input images and send them when they are requested.
format	Convert the input video to one of the specified pixel formats.
gradfun	Debands video quickly using gradients.
hflip	Horizontally flip the input video.

lut	Compute and apply a lookup table to the RGB/YUV input video.
lutrgb	Compute and apply a lookup table to the RGB input video.
lutyuv	Compute and apply a lookup table to the YUV input video.
negate	Negate input video.
noformat	Force libavfilter not to use any of the specified pixel formats for the input to the next filter.
null	Pass the source unchanged to the output.
overlay	Overlay a video source on top of the input.
pad	Pad input image to width:height[:x:y[:color]] (default x and y: 0, default color: black).
pixdesctest	Test pixel format definitions.
scale	Scale the input video to width:height size and/or convert the image format.
select	Select frames to pass in output.
setdar	Set the frame display aspect ratio.
setpts	Set PTS for the output video frame.
setsar	Set the pixel sample aspect ratio.
settb	Set timebase for the output link.
showinfo	Show textual information for each video frame.
slicify	Pass the images of input video on to next video filter as multiple slices.
split	Pass on the input to two outputs.
transpose	Transpose input video.
unsharp	Sharpen or blur the input video.
vflip	Flip the input video vertically.
buffer	Buffer video frames, and make them accessible to the filterchain.
color	Provide an uniformly colored input, syntax is: [color[:size[:rate]]]
movie	Read from a movie source.
nullsrc	Null video source, never return images.
rgbtestsrc	Generate RGB test pattern.
testsrc	Generate test pattern.
buffersink	Buffer video frames, and make them available to the end of the filter graph.
nullsink	Do absolutely nothing with the input video.

2.3.4 FFmpeg 对图像颜色空间的支持

ffmpeg 支持常见的图像颜色空间, 并且在 libavswscale 中定义了颜色空间转换的相关函数实现各种颜色模式的互转。具体的支持情况见附件:



FFmpeg 支持的图像
颜色空间

附件:

2.4 FFmpeg 功能及使用说明

2.4.1 ffplay 对多媒体的支持能力验证

一、视频

3gp 177X144 支持播放，在 windows 下播放正常，但是在 linux 下面偶有 BUG 如果发现画面无法显示而声音可以播放的情况下 可以试着切换全屏或者切换分辨率。

AVI	208X176	支持
	320X240	支持
	720X400	支持
	720X576	支持
DAT	352X288	支持
DiVX	720X576	支持
MKV	320X240	支持
	352X288	支持
	704X304	支持
	720X576	支持
MP4	320X240	支持
	352X288	支持
	720X400	支持
MPG	320X240	支持
	352X288	支持
	480X576	支持
	720X576	支持
	720X480	支持
VOB	352X288	支持
XVID	720X576	支持
MOV		支持
RMVB		支持

二、音频

AC3	48KHZ		支持
APE	11KHZ		支持
	22KHZ		支持
	44KHZ		支持
	48KHZ		支持
MP3	32KHZ	64Kbps	支持
	32KHZ	128Kbps	支持
	32KHZ	160Kbps	支持
	32KHZ	192Kbps	支持
	32KHZ	320Kbps	支持
	44KHZ	64Kbps	支持

OGG	44KHZ	128Kbps	支持
	44KHZ	160Kbps	支持
	44KHZ	192Kbps	支持
	44KHZ	320Kbps	支持
	48KHZ	64Kbps	支持
	48KHZ	128Kbps	支持
	48KHZ	160Kbps	支持
	48KHZ	192Kbps	支持
	48KHZ	320Kbps	支持
	32KHZ	128Kbps	支持
	32KHZ	192Kbps	支持
	44KHZ	64Kbps	支持
	44KHZ	128Kbps	支持
	44KHZ	192Kbps	支持
	48KHZ	64Kbps	支持
	44KHZ	128Kbps	支持
	44KHZ	192Kbps	支持
WAV	11KHZ		支持
	22KHZ	16Kbps	支持
	44KHZ	16Kbps	支持
	48KHZ	16Kbps	支持
WMA	8KHZ	16Kbps	支持
	11KHZ	16Kbps	支持
	16KHZ	16Kbps	支持
	22KHZ	16Kbps	支持
	44KHZ	16Kbps	支持
	48KHZ	16Kbps	支持

三、图像

PNG	支持
JPG	支持
JPEG	支持
GIF	支持
BMP	支持

2.4.2 FFmpeg 格式转换

第一步：准备媒体

前面已经讲的很清楚了，ffmpeg 如何安装不在赘述。准备好相应的文件，如图 2-1 所示。



图 2-1

第二步：启动 ffmpeg

由于做的是格式转换，在 ffserver 上不能直观的看见结果，故我是在 linux 下进行的。打开终端，值得一提的是格式转换需要超级用户才能进行，故在命令行输入：`su`，<回车>，输入密码进入超级用户，本例中，以 FFmpeg 将 `test.avi` 转换为 `test.mpg`。在命令行中输入：`./ffmpeg -i test.avi -r 25 -s 720x400 test.mpg`<回车>。其中原格式分辨率为 320x240，将转为 720x400，`-r` 前面已经解释其含义，表示设置帧频为 25。转换成功后如图 2-2 所示，前后两种格式播放效果如图 2-3 所示。相应的，转换为其他格式做相应的变化即可。

同时还可以在转换格式时进行强制的音视频转换，如`-vcodec + 格式`，将会强制将视频按指定格式编码，`-acodec + 格式`，将会强制按指定格式编码音频信息。在转换中有很多其他参数可以指定，如码率、分辨率、帧率等，具体按照 ffmpeg 的参数说明指定参数即可。但有一条转低不转高的原则需要注意，即品质差的音视频转换不建议转换到品质好的音视频。



图 2-2



图 2-3

再说说如何在转换视频的时候将音频合成到视频中，且覆盖其原来的音频。这个现在摸索出两种方法。

方法一：需要两条命令实现，先在命令行中输入：

```
./ffmpeg -i test.avi -an -r 25 test.mpg <回车>
```

此时将生成一个没有声音的 test.mpg 视频，再在命令行中输入：

```
./ffmpeg -i test.mpg -i test.mp3 -r 25 test1.mpg<回车>
```

此时将会生成一个名为 test1.mpg 的视频。该视频播放时视频为 test.avi 的视频，但音频变为了 test.mp3 的音频了。

方法二：只要一条指令即可实现。在命令行中输入：

```
./ffmpeg -i test.avi -i test.mp3 -vcodec copy -acodec copy -r 25 test2.mpg<回车>
```

此时将会生成一个名为 test2.mpg 的视频，播放时其视频为 test.avi 的视频，音频为 test.mp3。-vcodec copy 为 force video codec(‘copy’ to copy stream)。

有一点需要注意，文件命名不能有空格，否则会导致编译时不能通过。另外，-an 为不能使音频记录。

第三步：播放媒体

播放我们转换的媒体，看看是否满足我们当初的愿望，不出什么差错的话，是完全能够满足我们的要求的。

2.4.3 FFmpeg 视频截图

截取一张 300x200 尺寸大小的格式为 jpg 的一张图片：

```
./ffmpeg -i test.avi -y -f image2 -t 0.001 -s 300x200 test.jpg
```

要截取指定时间的图片，如 5 秒之后的：

```
./ffmpeg -i test.avi -y -f image2 -ss 5 -t 0.001 -s 300x200 test.jpg
```

其中，-ss 后的单位为秒，也可写成：-ss 00: 00:05。

把视频的前 30 帧转换为一个动态的 gif 图。需要说明的是，转换成功之后，如果用 ffplay 播放是看不出效果的，建议换成其他图片播放器播放。其转换命令为：

```
./ffmpeg -i test.avi -vframes 30 -pix_fmt rgb24 -y -f gif test.gif
```

也可以从视频中的第 10 秒开始截取后面的 5 秒内容转换为一个无限重播的动态 gif 图。其命令为：

```
./ffmpeg -i test.avi -pix_fmt rgb24 -ss 10 -t 5 -y -f gif test.gif
```

上面两种动态 gif 都是只播一次，想让其一直播，可再加一个参数：-loop_output 0。

2.4.4 FFmpeg 屏幕录制

屏幕录制其命令为：

```
./ffmpeg -f x11grab -r 25 -s wxga -i :0.0 /tmp/outputFile.mpg
```

其他相关参数可自行添加。需要说明的是，各个版本的 ffmpeg 对屏幕录制的命令不一。如果你只想录制一个应用程序窗口或者桌面上的一个固定区域，那么可以指定偏移位置和区域大小。使用 xwininfo -frame 命令可以完成查找上述参数。

注：ffmpeg 的屏幕录制功能只能在 Linux 环境下有效。并且在配置时需要添加 -enable-x11grab 指令，默认关闭。

2.4.5 FFmpeg 音视频采集

把摄像头的实时视频录制下来，存储为文件

```
./ffmpeg -f video4linux -s 320x240 -r 10 -i /dev/video0 test.asf
```

录音，其命令为：

```
./ffmpeg -i /dev/dsp -f oss test.mp3
```

2.5 FFmpeg 应用实例

2.5.1 用 FFserver 从文件生成流媒体

一、安装 ffmpeg

在 ubuntu 下，运行 `sudo apt-get ffmpeg` 安装 ffmpeg，在其他 linux 操作系统下，见 ffmpeg 的编译过程（编译完成后可执行自动安装）。

二、准备预播放的媒体文件

如 test.Mp3，在本文档中，默认放入用户文件夹下得 Music 文件夹内。（直接从设备采集不在本文档叙述范围之内）

三、修改 ffserver 配置信息

ffserver 配置文件为：/etc/ffserver.conf 打开，填写配置信息。配置信息包括三方面：

1. 端口绑定等基本信息，在 /etc/ffserver.conf 中有详细注释，在此不再重复，最终配置信息为：

Port 8090

BindAddress 0.0.0.0

MaxHTTPConnections 2000

MaxClients 1000

MaxBandwidth 1000

2. 媒体文件配置信息。本信息根据具体的媒体文件类型直接在配置文件中取消注释掉相应文件类型的配置信息，然后填写文件路径即可：

```
# MP3 audio
<Stream test.mp3>
File "/home/xiaoma/Music/test.mp3"
Format mp2
NoVideo
</Stream>
```

四、启动 ffserver

在终端中运行: `sudo ffserver -f /etc/ffserver.conf` 启动 ffserver.

五、播放流媒体

在浏览器中输入 <http://127.0.0.1:8090/test.mp3> 即可播放音乐.

在终端中输入 `ffplay http://localhost:8090/test.mp3` 可播放流媒体.

2.5.2 用 FFserver 从设备生成实时流

一、准备媒体

按照上节步骤安装 `ffmpeg`, 保证摄像头和声卡可用, 将从摄像头和声卡获取音视频信息。

二、修改 ffserver 配置信息

ffserver 配置文件为: `/etc/ffserver.conf` 打开, 填写配置信息.配置信息包括三方面:

1.端口绑定等基本信息, 在`/etc/ffserver.conf` 中有详细注释, 在此不再重复, 最终配置信息为:

```
Port 8090
BindAddress 0.0.0.0
MaxHTTPConnections 2000
MaxClients 1000
MaxBandwidth 1000
```

2.fend(传冲信息), 在文件播放中, 基本不用动本配置信息, 只需要根据具体情况分配缓冲文件.最终配置信息如下:

```
<Feed feed1.ffm>
File /tmp/feed1.ffm
FileMaxSize 2M
ACL allow 127.0.0.1
</Feed>
```

3.媒体文件配置信息.本信息根据具体的媒体文件类型直接在配置文件中取消注释掉相应文件类型的配置信息, 然后填写文件路径即可:

(中间会有很多很多配置信息, 都是关于音视频的, 有些配置还不懂, 慢慢摸索吧)

```
<Stream test1.mpg>
Feed feed1.ffm
Format mpeg
AudioBitRate 32
AudioChannels 1
AudioSampleRate 44100
```

```

VideoBitRate 64
VideoBufferSize 40
VideoFrameRate 3
VideoSize 160x128
VideoGopSize 12
</Stream>
# Flash
<Stream test.swf>
Feed feed1.ffm
Format swf
VideoFrameRate 2
VideoIntraOnly
NoAudio
</Stream>

```

三、启动 FFserver

在终端中运行: `sudo ffserver -f /etc/ffserver.conf` 启动 ffserver.

四、启动 ffmpeg

本例中, 以 ffmpeg 作为实时摄像头采集输入.在命令行中输入:

```
./ffmpeg -f video4linux2 -r 25 -i /dev/video0 /tmp/feed1.ffm
```

如果有音频设备, 则采集音频的命令如下:

```
./ffmpeg -f oss -i /dev/dsp -f video4linux2 -r 25 -i /dev/video0 /tmp/feed1.ffm
```

(音频格式参数自己配置)

五、播放流媒体

在浏览器中输入 `http://127.0.0.1:8090/test1.mpg` 即可播放音乐.

在终端中输入 `ffplay http://localhost:8090/test.swf` 可播放流媒体.

3.FFmpeg 架构

3.1 FFmpeg 文件结构

目录	文件	简要说明
libavformat 主要存放 ffmpeg 支持的 各种编解码器 的实现及 ffmpeg 编解码 功能相关的数	allcodecs.c	简单的注册类函数
	avcodec.h	编解码相关结构体定义和函数原型声明
	dsputil.c	限幅数组初始化
	dsputil.h	限幅数组声明
	imgconvert.c	颜色空间转换相关函数实现
	imgconvert_template.h	颜色空间转换相关结构体定义和函数声明
	utils_codec.c	一些解码相关的工具类函数的实现

据结构定义及函数定义和声明	mpeg4audio.c	mpeg4 音频编解码器的函数实现
	mpeg4audio.h	mpeg4 音频编解码器的函数声明
	mpeg4data.h	mpeg4 音视频编解码器的公用的函数声明及数据结构定义
	mpeg4video.c	mpeg4 视频编解码器的函数实现
	mpeg4video.h	mpeg4 视频编解码器的函数的声明及先关数据结构的定义
	mpeg4videodec.c	mpeg4 视频解码器的函数实现
	mpeg4videoenc.c	mpeg4 视频编码器的函数实现
libavformat 本目录主要存放 FFMPEG 支持的各种媒体格式 MUXER/DEMUXER 和数据流协议的定义和实现文件以及 ffmpeg 解复用相关的数据结构及函数定义	allformats.c	简单注册类函数
	avformat.h	文件和媒体格式相关函数声明和数据结构定义
	avio.c	无缓冲 IO 相关函数实现
	avio.h	无缓冲 IO 相关结构定义和函数声明
	aviobuf.c	有缓冲数据 IO 相关函数实现
	cutils.c	简单的字符串操作函数
	utils_format.c	文件和媒体格式相关的工具函数的实现
	file.c	文件 io 相关函数
	其他相关媒体流 IO 的函数和数据结构实现文件。如：rtsp、http 等。
	avi.c	AVI 格式的相关函数定义
	avi.h	AVI 格式的相关函数声明及数据结构定义
	avidec.c	AVI 格式 DEMUXER 相关函数定义
	avienc.c	AVI 格式 MUXER 相关函数定义
	其他媒体格式的 muxer/demuxer 相关函数及数据结构定义和声明文件
libavutil 主要存放 ffmpeg 工具类函数的定义	avutil.h	简单的像素格式宏定义
	bswap.h	简单的大小端转换函数的实现
	commom.h	公共的宏定义和简单函数的实现
	mathematics.c	数学运算函数实现
	rational.h	分数相关表示的函数实现

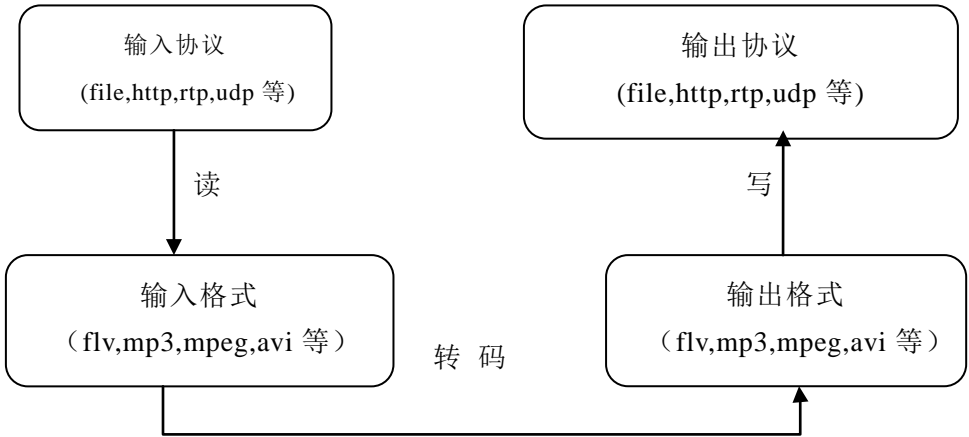
3.2 I/O 模块分析

3.2.1 概述

ffmpeg 项目的数据 IO 部分主要是在 libavformat 库中实现，某些对于内存的操作

部分在 libavutil 库中。数据 IO 是基于文件格式(Format)以及文件传输协议(Protocol)的，与具体的编解码标准无关。

ffmpeg 工程转码时数据 IO 层次关系如图所示：



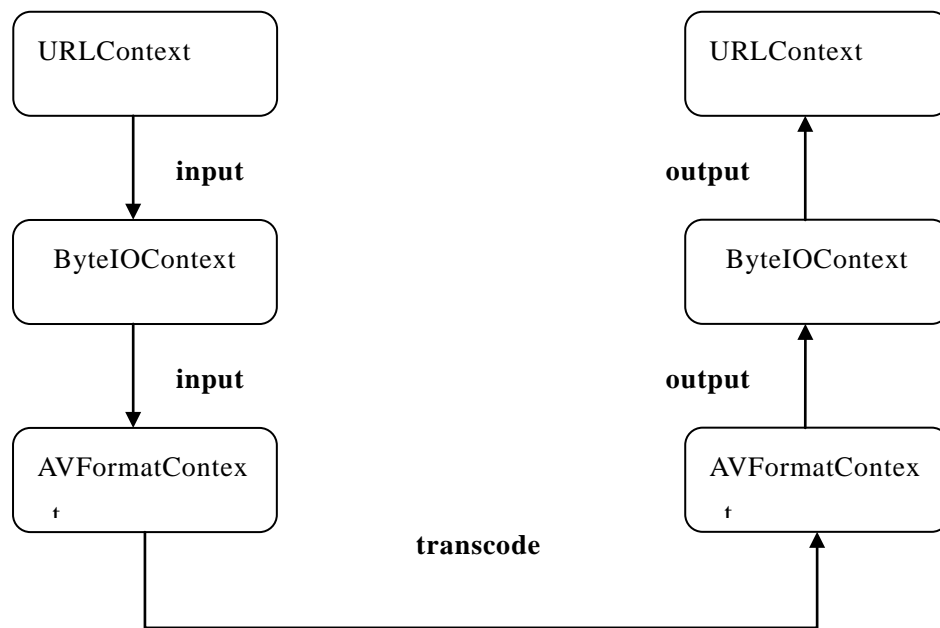
ffmpeg 转码数据 IO 流程

对于上面的数据 IO 流程，具体可以用下面的例子来说明，我们从一个 http 服务器获取音视频数据，格式是 flv 的，需要通过转码后变成 avi 格式，然后通过 udp 协议进行发布。其过程就如下所示：

- 1、读入 http 协议数据流，根据 http 协议获取真正的文件数据（去除无关报文信息）；
- 2、根据 flv 格式对数据进行解封装；
- 3、读取帧进行转码操作；
- 4、按照目标格式 avi 进行封装；
- 5、通过 udp 协议发送出去。

3.2.2 相关数据结构介绍

在 libavformat 库中与数据 IO 相关的数据结构主要有 URLProtocol、URLContext、ByteIOContext、AVFormatContext 等，各结构之间的关系如图所示。



libavformat 库中 IO 相关数据结构之间的关系

1、URLProtocol 结构

表示广义的输入文件，该结构体提供了很多的功能函数，每一种广义的输入文件（如：file、pipe、tcp、rtsp 等等）对应着一个 URLProtocol 结构，在 `av_register_all()` 中将该结构体初始化为一个链表，表头为 `avio.c` 里的 `URLProtocol *first_protocol = NULL`；保存所有支持的输入文件协议，该结构体的定义如下：

```

typedef struct URLProtocol
{
    const char *name;
    int (*url_open)(URLContext *h, const char *url, int flags);
    int (*url_read)(URLContext *h, unsigned char *buf, int size);
    int (*url_write)(URLContext *h, const unsigned char *buf, int size);
    int64_t (*url_seek)(URLContext *h, int64_t pos, int whence);
    int (*url_close)(URLContext *h);
    struct URLProtocol *next;
    int (*url_read_pause)(URLContext *h, int pause);
    int64_t (*url_read_seek)(URLContext *h, int stream_index,
                            int64_t timestamp, int flags);
    int (*url_get_file_handle)(URLContext *h);
    int priv_data_size;
    const AVClass *priv_data_class;
    int flags;
    int (*url_check)(URLContext *h, int mask);
} URLProtocol;

```

注意到，URLProtocol 是一个链表结构，这是为了协议的统一管理，ffmpeg 项目中将所有的用到的协议都存放在一个全局变量 first_protocol 中，协议的注册是在 av_register_all 中完成的，新添加单个协议可以调用 av_register_protocol2 函数实现。而协议的注册就是将具体的协议对象添加至 first_protocol 链表的末尾。

URLProtocol 在各个具体的文件协议中有一个具体的实例，如在 file 协议中定义为：

```
URLProtocol ff_file_protocol = {  
    .name                = "file",  
    .url_open             = file_open,  
    .url_read             = file_read,  
    .url_write            = file_write,  
    .url_seek            = file_seek,  
    .url_close            = file_close,  
    .url_get_file_handle = file_get_handle,  
    .url_check           = file_check,  
};
```

2、URLContext 结构

URLContext 提供了与当前打开的具体的文件协议（URL）相关数据的描述，在该结构中定义了指定当前 URL（即 filename 项）所要用到的具体的 URLProtocol，即：提供了一个在 URLprotocol 链表中找到具体项的依据，此外还有一些其它的标志性的信息，如 flags，is_streamed 等。它可以看成某一种协议的载体。其结构定义如下：

```
typedef struct URLContext  
{  
    const AVClass *av_class; ///  
    struct URLProtocol *prot;  
    int flags;  
    int is_streamed;  /**< true if streamed (no seek possible), default = false *  
    int max_packet_size;    void *priv_data;  
    char *filename; /**< specified URL */  
    int is_connected;  
} URLContext;
```

那么 ffmpeg 依据什么信息初始化 URLContext？然后又是如何初始化 URLContext 的呢？

在打开一个 URL 时，全局函数 ffurl_open 会根据 filename 的前缀信息来确定 URL 所使用的具体协议，并为该协议分配好资源，再调用 ffurl_connect 函数打开具体协议，即调用协议的 url_open，调用关系如下：

```
int av_open_input_file(AVFormatContext **ic_ptr, const char *filename,  
                      AVInputFormat *fmt,  
                      int buf_size,  
                      AVFormatParameters *ap)
```

```

int avformat_open_input(AVFormatContext **ps, const char *filename,
                        AVInputFormat *fmt, AVDictionary **options)

static int init_input(AVFormatContext *s, const char *filename)

    int avio_open(AVIOContext **s, const char *filename, int flags)
    int ffurl_open(URLContext **puc, const char *filename, int flags)
int ffurl_alloc(URLContext **puc, const char *filename, int flags)
    static int url_alloc_for_protocol(URLContext **puc, struct
URLProtocol *up, const char *filename, int flags)

```

浅蓝色部分的函数完成了 URLContext 函数的初始化，URLContext 使 ffmpeg 外所暴露的接口是统一的，而不是对于不同的协议用不同的函数，这也是面向对象思维的体现。在此结构中还有一个值得说的是 priv_data 项，这是结构的一个可扩展项，具体协议可以根据需要添加相应的结构，将指针保存在这就行。

3、AVIOContext 结构

AVIOContext（即：ByteIOContext）是由 URLProtocol 和 URLContext 结构扩展而来，也是 ffmpeg 提供给用户的接口，它将以上两种不带缓冲的读取文件抽象为带缓冲的读取和写入，为用户提供带缓冲的读取和写入操作。数据结构定义如下：

typedef struct

```

{
    unsigned char *buffer; /**< Start of the buffer. */
    int buffer_size; /**< Maximum buffer size */
    unsigned char *buf_ptr; /**< Current position in the buffer */
    unsigned char *buf_end;
    void *opaque; /**< 关联 URLContext
    int (*read_packet)(void *opaque, uint8_t *buf, int buf_size);
    int (*write_packet)(void *opaque, uint8_t *buf, int buf_size);
    int64_t (*seek)(void *opaque, int64_t offset, int whence);
    int64_t pos;
    int must_flush;
    int eof_reached; /**< true if eof reached */
    int write_flag; /**< true if open for writing */
    int max_packet_size;
    unsigned long checksum;
    unsigned char *checksum_ptr;
    unsigned long (*update_checksum)(unsigned long checksum, const uint8_t *buf,
                                     unsigned int size);
    int error;
    int (*read_pause)(void *opaque, int pause)
    int64_t (*read_seek)(void *opaque, int stream_index, int64_t timestamp, int flags);
    int seekable;
} AVIOContext;

```

结构简单的为用户提供读写容易实现的四个操作，read_packet write_packet read_pause read_seek，极大的方便了文件的读取，四个函数在加了缓冲机制后被中转到，URLContext 指向的实际的文件协议读写函数中。

下面给出 0.8 版本中是如何将 AVIOContext 的读写操作中转到实际文件中的。

在 avio_open() 函数中调用了 ffio_fdopen() 函数完成了对 AVIOContext 的初始化，其调用过程如下：

```
int avio_open(AVIOContext **s, const char *filename, int flags)
{
    ffio_fdopen(s, h); //h 是 URLContext 指针
    ffio_init_context(*s, buffer, buffer_size,
                     h->flags & AVIO_FLAG_WRITE, h,
                     (void*)ffurl_read, (void*)ffurl_write, (void*)ffurl_seek)
}
```

蓝色部分的函数调用完成了对 AVIOContext 的初始化，在初始化的过程中，将 AVIOContext 的 read_packet、write_packet、seek 分别初始化为：ffurl_read ffurl_write ffurl_seek，而这三个函数又将具体的读写操作中转为：h->prot->url_read、h->prot->url_write、h->prot->url_seek，另外两个变量初始化时也被相应的中转，如下：

```
(*s)->read_pause = (int (*)(void *, int))h->prot->url_read_pause;
(*s)->read_seek = (int64_t (*)(void *, int, int64_t, int))h->prot->url_read_seek;
```

所以，可以简要的描述为：AVIOContext 的接口是加了缓冲后的 URLProtocol 的函数接口。

在 aviobuf.c 中定义了一系列关于 ByteIOContext 这个结构体的函数，如下

put_xxx 系列：

```
void put_byte(ByteIOContext *s, int b);
void put_buffer(ByteIOContext *s, const unsigned char *buf, int size);
void put_le64(ByteIOContext *s, uint64_t val);
void put_be64(ByteIOContext *s, uint64_t val);
void put_le32(ByteIOContext *s, unsigned int val);
void put_be32(ByteIOContext *s, unsigned int val);
void put_le24(ByteIOContext *s, unsigned int val);
void put_be24(ByteIOContext *s, unsigned int val);
void put_le16(ByteIOContext *s, unsigned int val);
void put_be16(ByteIOContext *s, unsigned int val);
void put_tag(ByteIOContext *s, const char *tag);
```

get_xxx 系列：

```
int get_buffer(ByteIOContext *s, unsigned char *buf, int size);
int get_partial_buffer(ByteIOContext *s, unsigned char *buf, int size);
int get_byte(ByteIOContext *s);
unsigned int get_le24(ByteIOContext *s);
unsigned int get_le32(ByteIOContext *s);
uint64_t get_le64(ByteIOContext *s);
```

```

unsigned int get_le16(ByteIOContext *s);
char *get_strz(ByteIOContext *s, char *buf, int maxlen);
unsigned int get_be16(ByteIOContext *s);
unsigned int get_be24(ByteIOContext *s);
unsigned int get_be32(ByteIOContext *s);
uint64_t get_be64(ByteIOContext *s);

```

这些 put_xxx 及 get_xxx 函数是用于从缓冲区 buffer 中写入或者读取若干字节，对于读写整型数据，分别实现了大端和小端字节序的版本。而缓冲区 buffer 中的数据又是从何而来呢，有一个 fill_buffer 的函数，在 fill_buffer 函数中调用了 ByteIOContext 结构的 read_packet 接口。在调用 put_xxx 函数时，并没有直接进行真正写入操作，而是先缓存起来，直到缓存达到最大限制或调用 flush_buffer 函数对缓冲区进行刷新，才使用 write_packet 函数进行写入操作。

3.3 Demuxer 和 muxer 模块分析

3.3.1 概述

ffmpeg 的 demuxer 和 muxer 接口分别在 AVInputFormat 和 AVOutputFormat 两个结构体中实现，在 av_register_all() 函数中将两个结构分别静态初始化为两个链表，保存在全局变量：first_iformat 和 first_oformat 两个变量中。在 FFmpeg 的文件转换或者打开过程中，首先要做的就是根据传入文件和传出文件的后缀名匹配合适的 demuxer 和 muxer，得到合适的信息后保存在 AVFormatContext 中。

3.3.2 相关数据结构介绍

1、AVInputFormat

该结构被称为 demuxer，是音视频文件的一个解封装器，它的定义如下：

```

typedef struct AVInputFormat
{
    const char *name;
    const char *long_name;
    int priv_data_size; //具体文件容器格式对应的 Context 的大小，如：avicontext
    int (*read_probe)(AVProbeData *);
    int (*read_header)(struct AVFormatContext *,
                      AVFormatParameters *ap);
    int (*read_packet)(struct AVFormatContext *, AVPacket *pkt);
    int (*read_close)(struct AVFormatContext *);
#ifdef FF_API_READ_SEEK
    attribute_deprecated int (*read_seek)(struct AVFormatContext *,
                                           int stream_index, int64_t timestamp, int flags);
#endif
    int64_t (*read_timestamp)(struct AVFormatContext *s, int stream_index,
                             int64_t *pos, int64_t pos_limit);
    int flags;
}

```

```

    const char *extensions;
    int value;
    int (*read_play)(struct AVFormatContext *);
    int (*read_pause)(struct AVFormatContext *);
    const struct AVCodecTag * const *codec_tag;
    int (*read_seek2)(struct AVFormatContext *s, int stream_index, int64_t min_ts,
                      int64_t ts, int64_t max_ts, int flags);
#ifdef FF_API_OLD_METADATA
    const AVMetadataConv *metadata_conv;
#endif

    const AVClass *priv_class; ///< AVClass for the private context
    struct AVInputFormat *next;
} AVInputFormat;

```

对于不同的文件格式要实现相应的函数接口，这样每一种格式都有一个对应的 demuxer，所有的 demuxer 都保存在全局变量 `first_iformat` 中。红色表示提供的接口。

2、AVOutputFormat

该结构与 `AVInputFormat` 类似也是在编译时静态初始化，组织为一个链表结构，提供了多个 muxer 的函数接口。

```

int (*write_header)(struct AVFormatContext *);
int (*write_packet)(struct AVFormatContext *, AVPacket *pkt);
int (*write_trailer)(struct AVFormatContext *);

```

对于不同的文件格式要实现相应的函数接口，这样每一种格式都有一个对应的 muxer，所有的 muxer 都保存在全局变量 `first_oformat` 中。

3、AVFormatContext

该结构表示与程序当前运行的文件容器格式使用的上下文，着重于所有文件容器共有的属性，在运行时动态的确定其值，是 `AVInputFormat` 和 `AVOutputFormat` 的载体，但同一个结构对象只能使 `AVInputFormat` 和 `AVOutputFormat` 中的某一个有效。每一个输入和输出文件，都在

```

static AVFormatContext *output_files[MAX_FILES] 和
static AVFormatContext *input_files[MAX_FILES];

```

定义的指针数组全局变量中有对应的实体。对于输入和输出，因为共用的是同一个结构体，所以需要分别对该结构中如下定义的 `iformat` 或 `oformat` 成员赋值。在转码时读写数据是通过 `AVFormatContext` 结构进行的。定义如下：

```

typedef struct AVFormatContext
{
    const AVClass *av_class;

    struct AVInputFormat *iformat; //指向具体的 demuxer
    struct AVOutputFormat *oformat; //指向具体的 muxer
    void *priv_data; //具体文件容器格式的 Context 如：avicontext
    AVIOContext *pb; //广义的输入输出；

```



```

    unsigned int nb_streams; //本次打开的文件容器中流的数量
    AVStream **streams; //每个流的相关描述
    char filename[1024]; // input or output filename */
    int64_t timestamp;
    int ctx_flags;
    struct AVPacketList *packet_buffer;
    .....
    enum CodecID video_codec_id;
    enum CodecID audio_codec_id;
    enum CodecID subtitle_codec_id;
    unsigned int max_index_size;
    unsigned int max_picture_buffer;
    .....
    struct AVPacketList *raw_packet_buffer;
    struct AVPacketList *raw_packet_buffer_end;
    struct AVPacketList *packet_buffer_end;
    .....
} AVFormatContext;

```

红色部分的成员是 AVFormatContext 中最为重要的成员变量，这些变量的初始化是 ffmpeg 能正常工作的必要条件，那么，AVFormatContext 是如何被初始化的呢？文件的格式是如何被探测到的呢？

首先我们来探讨：

```

struct AVInputFormat *iformat; //指向具体的 demuxer
struct AVOutputFormat *oformat; //指向具体的 muxer
void *priv_data; //具体文件容器格式的 Context 如：avicontext

```

三个成员的初始化。

在 avformat_open_input() 函数中调用了 init_input() 函数，然后用调用了 av_probe_input_format() 函数实现了对 AVFormatContext 的初始化。其调用关系如下：

```

int av_open_input_file(AVFormatContext **ic_ptr, const char *filename,
                      AVInputFormat *fmt, int buf_size, AVFormatParameters *ap);
int avformat_open_input(ic_ptr, filename, fmt, &opts);
static int init_input(s, filename);
av_probe_input_format(&pd, 0);

```

av_probe_input_format (AVProbeData *pd, int is_opened, int *score_max) 函数用途是根据传入的 probe data 数据，依次调用每个 demuxer 的 read_probe 接口，来进行该 demuxer 是否和传入的文件内容匹配的判断。与 demuxer 的匹配不同，muxer 的匹配是调用 guess_format 函数，根据 main() 函数的 argv 里的输出文件后缀名来进行的。至此完成了前三个重要成员的初始化，具体的做法就不在深入分析。

下面分别给出 av_read_frame 函数以及 av_write_frame 函数的基本流程。

```

int av_read_frame(AVFormatContext *s, AVPacket *pkt);

```

```

→av_read_frame_internal
  → av_read_packet
    →iformat->read_packet (在实现中会丢弃多余信息)
      →av_get_packet
        →get_xxx

```

```

int av_write_frame(AVFormatContext *s, AVPacket *pkt);
  →oformat->write_packet
    →put_xxx

```

由上可见，对 AVFormatContext 的读写操作最终是通过 ByteIOContext 来实现的，这样，AVFormatContext 与 URLContext 就由 ByteIOContext 结构联系到一起了。在 AVFormat 结构体中有一个 packet 的缓冲区 raw_packet_buffer，是 AVPacketList 的指针类型，av_read_packet 函数将读到的包添加至 raw_packet_buffer 链表末尾。

3.4 Decoder/Encoder 模块

3.4.1 概述

编解码模块主要包含的数据结构为：AVCodec、AVCodecContext 每一个解码类型都会有自己的 Codec 静态对象，Codec 的 int priv_data_size 记录该解码器上下文的结构大小，如 MsrleContext。这些都是编译时确定的，程序运行时通过 avcodec_register_all() 将所有的解码器注册成一个链表。在 av_open_input_stream() 函数中调用 AVInputFormat 的 read_header() 中读文件头信息时，会读出数据流的 CodecID，即确定了他的解码器 Codec。

在 main() 函数中除了解析传入参数并初始化 demuxer 与 muxer 的 parse_options() 函数以外，其他的功能都是在 av_encode() 函数里完成的。在 libavcodec\utils.c 中有如下二个函数：AVCodec *avcodec_find_encoder(enum CodecID id) 和 AVCodec *avcodec_find_decoder(enum CodecID id) 他们的功能就是根据传入的 CodecID，找到匹配的 encoder 和 decoder。在 av_encode() 函数的开头，首先初始化各个 AVInputStream 和 AVOutputStream，然后分别调用上述二个函数，并将匹配上的 encoder 与 decoder 分别保存在：

```

AVInputStream->AVStream *st->AVCodecContext *codec->struct AVCodec *codec
与 AVOutputStream->AVStream *st->AVCodecContext *codec->struct AVCodec *codec
变量。

```

3.4.2 相关数据结构的初始化

AVCodecContext 结构

AVCodecContext 保存 AVCodec 指针和与 codec 相关数据，如 video 的 width、height，audio 的 sample rate 等。

AVCodecContext 中的 codec_type，codec_id 二个变量对于 encoder/decoder 的匹配来说，最为重要。

```

enum CodecType codec_type; /* see CODEC_TYPE_xxx */

```

```
enum CodecID codec_id; /* see CODEC_ID_xxx */
```

如上所示，codec_type 保存的是 CODEC_TYPE_VIDEO，CODEC_TYPE_AUDIO 等媒体类型，codec_id 保存的是 CODEC_ID_FLV1，CODEC_ID_VP6F 等编码方式。

以支持 flv 格式为例，在前述的 av_open_input_file(……) 函数中，匹配到正确的 AVInputFormat demuxer 后，通过 av_open_input_stream() 函数中调用 AVInputFormat 的 read_header 接口来执行 flvdec.c 中的 flv_read_header() 函数。flv_read_header() 函数内，根据文件头中的数据，创建相应的视频或音频 AVStream，并设置 AVStream 中 AVCodecContext 的正确的 codec_type 值。codec_id 值是在解码过程。flv_read_packet() 函数执行时根据每一个 packet 头中的数据来设置的。

以 aivec 为例 有如下初始化，我们主要知道的就是 code_id 和 code_type 该字段关联具体的解码器，和解码类型（音视频或 subtitle）

```
if (st->codec->stream_codec_tag == AV_RL32("Axan"))
{
    st->codec->codec_id = CODEC_ID_XAN_DPCM;
    st->codec->codec_tag = 0;
}
if (amv_file_format)
{
    st->codec->codec_id = CODEC_ID_ADPCM_IMA_AMV;
    ast->dshow_block_align = 0;
}
break;
case AVMEDIA_TYPE_SUBTITLE:
    st->codec->codec_type = AVMEDIA_TYPE_SUBTITLE;
    st->request_probe = 1;
    break;
default:
    st->codec->codec_type = AVMEDIA_TYPE_DATA;
    st->codec->codec_id = CODEC_ID_NONE;
    st->codec->codec_tag = 0;
    avio_skip(pb, size);
```

3.5 其他重要数据结构的初始化

3.5.1 AVStream

AVStream 结构保存与数据流相关的编解码器，数据段等信息。比较重要的有如下二个成员：

```
AVCodecContext *codec; /**< codec context */
void *priv_data;
```

其中 codec 指针保存的就是上节所述的 encoder 或 decoder 结构。priv_data 指针保存的是和具体编解码流相关的数据，如下代码所示，在 ASF 的解码过程中，priv_data 保存的就是 ASFStream 结构的数据。

```
AVStream *st;
ASFStream *asf_st;
... ..
st->priv_data = asf_st;
```

3.5.2 AVInputStream/ AVOutputStream

根据输入和输出流的不同，前述的 AVStream 结构都是封装在 AVInputStream 和 AVOutputStream 结构中，在 av_encode() 函数中使用。AVInputStream 中还保存的有与时间有关的信息。AVOutputStream 中还保存有与音视频同步等相关的信息。

3.5.3 AVPacket

AVPacket 结构定义如下，其是用于保存读取的 packet 数据。

```
typedef struct AVPacket
{
    int64_t pts;           ///< presentation time stamp in time_base units
    int64_t dts;           ///< decompression time stamp in time_base units
    uint8_t *data;
    int size;
    int stream_index;
    int flags;
    int duration; ///< presentation duration in time_base units
    void (*destruct)(struct AVPacket *);
    void *priv;
    int64_t pos;          ///< byte position in stream, -1 if unknown
} AVPacket;
```

在 av_encode() 函数中，调用 AVInputFormat 的

(*read_packet)(struct AVFormatContext *, AVPacket *pkt) 接口，读取输入文件的一帧数据保存在当前输入 AVFormatContext 的 AVPacket 成员中。

4.FFmpeg 裁剪说明

本文对 ffmpeg 进行裁剪采用的是配置所需的接口，不需要的不配置，而不是采用修改源代码的方式。

4.1 configure 参数

4.1.1 通用选项

在 linux 下进入终端，找到 ffmpeg 解压位置，输入如下命令：

```
root@web ffmpeg]# ./configure -help
```

得到 configure 的基本选项参数，其并没有中文解释。

--help 显示此帮助信息|print this message

--log[=FILE|yes|no] 记录测试并输出到 config.err 文件|log tests and output to FILE [config.err]

--prefix=PREFIX 安装程序到指定目录（默认/usr/local）|install in PREFIX [/usr/local]

--libdir=DIR 安装库到指定目录（默认 prefix/lib）|install libs in DIR [PREFIX/lib]

--shlibdir=DIR 指定共享库路径（默认 prefix/lib）|install shared libs in DIR [PREFIX/lib]

--incdir=DIR 指定 includes 路径（默认 prefix/include/ffmpeg）|install includes in DIR [PREFIX/include/ffmpeg]

--mandir=DIR 指定 man page 路径（默认 prefix/man）install man page in DIR [PREFIX/man]

--enable-mp3lame 启用 mp3 编码 libmp3lame（默认关闭）enable MP3 encoding via libmp3lame [default=no]

--enable-libogg 启用 ogg 支持 libogg（默认关闭）enable Ogg support via libogg [default=no]

--enable-vorbis 启用 Vorbis 支持 libvorbis（默认关闭）enable Vorbis support via libvorbis [default=no]

--enable-faad 启用 faad 支持 libfaad（默认关闭）enable FAAD support via libfaad [default=no]

--enable-faadbin 启用 faad 运行时链接支持（默认关闭）build FAAD support with runtime linking [default=no]

--enable-faac 启用 faac 支持 libfaac（默认关闭）enable FAAC support via libfaac [default=no]

--enable-libgsm 启用 GSM 支持 libgsm（默认关闭）enable GSM support via libgsm [default=no]

--enable-xvid 启用 xvid 支持 xvidcore（默认关闭）enable XviD support via xvidcore [default=no]

--enable-x264 启用 H.264 编码（默认关闭）enable H.264 encoding via x264 [default=no]

--enable-mingw32 启用 MinGW 本地/交叉 win 环境编译|enable MinGW native/cross Windows compile

--enable-mingwce 启用 MinGW 本地/交叉 winCE 环境编译 enable MinGW native/cross WinCE compile

--enable-a52 启用 A52 支持（默认关闭）enable GPLed A52 support [default=no]

--enable-a52bin 启用运行时打开 liba52.so.0（默认关闭）open liba52.so.0 at runtime [default=no]

--enable-dts 启用 DTS 支持（默认关闭）enable GPLed DTS support [default=no]

--enable-pp 启用后加工支持（默认关闭）enable GPLed postprocessing support [default=no]

--enable-static 构建静态库（默认启用）build static libraries [default=yes]

--disable-static 禁止构建静态库（默认关闭）do not build static libraries [default=no]

--enable-shared 构建共享库（默认关闭） build shared libraries [default=no]
 --disable-shared 禁止构建共享库（默认启用） do not build shared libraries [default=yes]
 --enable-amr_nb 启用 amr_nb float 音频编解码器|enable amr_nb float audio codec
 --enable-amr_nb-fixed 启用 fixed amr_nb codec | use fixed point for amr-nb codec
 --enable-amr_wb 启用 amr_wb float 音频编解码器|enable amr_wb float audio codec
 --enable-amr_if2 启用 amr_wb IF2 音频编解码器|enable amr_wb IF2 audio codec
 --enable-sunmlib 启用 Sun medialib（默认关闭） | use Sun medialib [default=no]
 --enable-pthreads 启用 pthreads（多线程）（默认关闭） use pthreads [default=no]
 --enable-dc1394 启用 libdc1394、libraw1394 抓取 IIDC-1394（默认关闭） enable IIDC-1394 grabbing using libdc1394 and libraw1394 [default=no]
 --enable-swscaler 启用计数器支持？（默认关闭） software scaler support [default=no]
 --enable-avisynth 允许读取 AVISynth 脚本文件（默认关闭） allow reading AVISynth script files [default=no]
 --enable-gpl 允许使用 GPL（默认关闭） allow use of GPL code, the resulting libav* and ffmpeg will be under GPL [default=no]
 Advanced options (experts only): 高级选项参数（供专业人员使用）
 --source-path=PATH 源码的路径（当前为/root/flv/ffmpeg） | path to source code [/root/flv/ffmpeg]
 --cross-prefix=PREFIX 为编译工具指定路径 | use PREFIX for compilation tools []
 --cross-compile 假定使用了交叉编译 | assume a cross-compiler is used
 --cc=CC 指定使用何种 C 编译器（默认 gcc） use C compiler CC [gcc]
 --make=MAKE 使用特定的 make | use specified make [make]
 --extra-cflags=ECFLAGS 添加 ECFLAGS 到 CFLAGS | add ECFLAGS to CFLAGS []
 --extra-ldflags=ELDFLAGS 添加 ELDFLAGS 到 LDFLAGS（默认-Wl, --as-needed） | add ELDFLAGS to LDFLAGS [-Wl, --as-needed]
 --extra-libs=ELIBS 添加 ELIBS | add ELIBS []
 --build-suffix=SUFFIX 为专用程序添加后缀 | suffix for application specific build []
 --arch=ARCH 选择机器架构（默认 x86） select architecture [x86]
 --cpu=CPU 选用最低的 cpu（影响指令的选择，可以在老 CPU 上出错） | selects the minimum cpu required (affects instruction selection, may crash on older CPUs)
 --powerpc-perf-enable 启用 PPC 上面的性能报告（需要启用 PMC） enable performance report on PPC (requires enabling PMC)
 --disable-mmx 禁用 MMX | disable MMX usage
 --disable-armv5te 禁用 armv5te | disable armv5te usage
 --disable-iwmmxt 禁用 iwmmxt | disable iwmmxt usage
 --disable-altivec 禁用 AltiVec | disable AltiVec usage
 --disable-audio-oss 禁用 OSS 音频支持（默认启用） disable OSS audio support [default=no]
 --disable-audio-beos 禁用 BeOS 音频支持（默认启用） disable BeOS audio support [default=no]

--disable-v4l	禁用 video4linux 提取（默认启用）disable video4linux grabbing [default=no]
--disable-v4l2	禁用 video4linux2 提取（默认启用）disable video4linux2 grabbing [default=no]
--disable-bktr	禁用 bktr 视频提取（默认启用）disable bktr video grabbing [default=no]
--disable-dv1394	禁用 DV1394 提取(默认启用)disable DV1394 grabbing [default=no]
--disable-network	禁用网络支持（默认支持）disable network support [default=no]
--disable-ipv6	禁用 ipv6 支持（默认支持）disable ipv6 support [default=no]
--disable-zlib	禁用 zlib（默认支持）disable zlib [default=no]
--disable-simple_idct	禁用 simple IDCT 例程（默认启用）disable simple IDCT routines [default=no]
--disable-vhook	禁用 video hooking 支持 disable video hooking support
--enable-gprof	enable profiling with gprof [no]
--disable-debug	禁用调试符号 disable debugging symbols
--disable-opts	禁用编译器最优化 disable compiler optimizations
--disable-mpegaudio-hp	启用更快的解码 MPEG 音频(但精确度较低)(默认禁用)faster (but less accurate) MPEG audio decoding [default=no]
--disable-protocols	禁用 I/O 协议支持（默认启用）disable I/O protocols support [default=no]
--disable-ffserver	禁用生成 ffserver disable ffserver build
--disable-ffplay	禁用生成 ffplay disable ffplay build
--enable-small	启用优化文件尺寸大小（牺牲速度）optimize for size instead of speed
--enable-memalign-hack	启用模拟内存排列，由内存调试器干涉？ emulate memalign, interferes with memory debuggers
--disable-strip	禁用剥离可执行程序 and 共享库 disable stripping of executables and shared libraries
--disable-encoder=NAME	禁用 XX 编码器 disables encoder NAME
--enable-encoder=NAME	启用 XX 编码器 enables encoder NAME
--disable-decoder=NAME	禁用 XX 解码器 disables decoder NAME
--enable-decoder=NAME	启用 XX 解码器 enables decoder NAME
--disable-encoders	禁用所有编码器 disables all encoders
--disable-decoders	禁用所有解码器 disables all decoders
--disable-muxer=NAME	禁用 XX 混音器 disables muxer NAME
--enable-muxer=NAME	启用 XX 混音器 enables muxer NAME
--disable-muxers	禁用所有混音器 disables all muxers
--disable-demuxer=NAME	禁用 XX 解轨器 disables demuxer NAME
--enable-demuxer=NAME	启用 XX 解轨器 enables demuxer NAME
--disable-demuxers	禁用所有解轨器 disables all demuxers

<code>--enable-parser=NAME</code>	启用 XX 剖析器 enables parser NAME
<code>--disable-parser=NAME</code>	禁用 XX 剖析器 disables parser NAME
<code>--disable-parsers</code>	禁用所有剖析器 disables all parsers

4.1.2 基本选项介绍

以下为配置 `ffmpeg` 的基本选项，其含义如下：

`--cache-file=FILE`

`configure` 会在你的系统上测试存在的特性(或者 bug!)。为了加速随后进行的配置，测试的结果会存储在一个 `cache file` 里。当 `configure` 到每个子树里都有 `configure` 脚本的复杂的源码树时，一个很好的 `cache file` 的存在会有很大帮助。

`--help`

输出帮助信息。即使是有经验的用户也偶尔需要使用使用 `--help` 选项，因为一个复杂的项目会包含附加的选项。例如，GCC 包里的 `configure` 脚本就包含了允许你控制是否生成和在 GCC 中使用 GNU 汇编器的选项。

`--no-create`

`configure` 中的一个主要函数会制作输出文件。此选项阻止 `configure` 生成这个文件。你可以认为这是一种演习(dry run)，尽管缓存(cache)仍然被改写了。

`--quiet`

`--silent`

当 `configure` 进行他的测试时，会输出简要的信息来告诉用户正在作什么。这样做是因为 `configure` 可能会比较慢，没有这种输出的话用户将会被扔在一旁疑惑正在发生什么。使用这两个选项中的任何一个都会把你扔到一旁。(译注：这两句话比较有意思，原文是这样的：If there was no such output, the user would be left wondering what is happening. By using this option, you too can be left wondering!)

`--version`

打印用来产生'configure'脚本的 Autoconf 的版本号。

`--prefix=PEWFIX`

`--prefix` 是最常用的选项。制作出的 `Makefile` 会查看随此选项传递的参数，当一个包在安装时可以彻底的重新安置他的结构独立部分。举一个例子，当安装一个包，例如说 Emacs，下面的命令将会使 Emacs Lisp file 被安装到"/opt/gnu/share"：

```
$ ./configure --prefix=/opt/gnu
```

`--exec-prefix=EPREFIX`

与 `--prefix` 选项类似，但是他是用来设置结构倚赖的文件的安装位置。编译好的 emacs 二进制文件就是这样一个文件。如果没有设置这个选项的话，默认使用的选项值将被设为和 `--prefix` 选项值一样。

`--bindir=DIR`

指定二进制文件的安装位置。这里的二进制文件定义为可以被用户直接执行的程序。

`--sbindir=DIR`

指定超级二进制文件的安装位置。这是一些通常只能由超级用户执行的程序。

`--libexecdir=DIR`

指定可执行支持文件的安装位置。与二进制文件相反，这些文件从来不直接由用户

执行，但是可以被上面提到的二进制文件所执行。

--datadir=DIR

指定通用数据文件的安装位置。

--sysconfdir=DIR

指定在单个机器上使用的只读数据的安装位置。

--sharedstatedir=DIR

指定可以在多个机器上共享的可写数据的安装位置。

--localstatedir=DIR

指定只能单机使用的可写数据的安装位置。

--libdir=DIR

指定库文件的安装位置。

--includedir=DIR

指定 C 头文件的安装位置。其他语言如 C++ 的头文件也可以使用此选项。

--oldincludedir=DIR

指定为除 GCC 外编译器安装的 C 头文件的安装位置。

--infodir=DIR

指定 Info 格式文档的安装位置。Info 是被 GNU 工程所使用的文档格式。

--mandir=DIR

指定手册页的安装位置。

--srcdir=DIR

这个选项对安装没有作用。他会告诉 configure 源码的位置。一般来说不用指定此选项，因为 configure 脚本一般和源码文件在同一个目录下。

--program-prefix=PREFIX

指定将被加到所安装程序的名字上的前缀。例如，使用 **--program-prefix=g** 来 configure 一个名为 tar 的程序将会使安装的程序被命名为 **gtar**。当和其他的安装选项一起使用时，这个选项只有当他被 **Makefile.in** 文件使用时才会工作。

--program-suffix=SUFFIX

指定将被加到所安装程序的名字上的后缀。

--program-transform-name=PROGRAM

这里的 **PROGRAM** 是一个 sed 脚本。当一个程序被安装时，他的名字将经过 **sed -e PROGRAM** 来产生安装的名字。

--build=BUILD

指定软件包安装的系统平台。如果没有指定，默认值将是 **--host** 选项的值。

--host=HOST

指定软件运行的系统平台。如果没有指定，将会运行 **config.guess** 来检测。

--target=GARGET'

指定软件面向(target to)的系统平台。这主要在程序语言工具如编译器和汇编器上下文中起作用。如果没有指定，默认将使用 **--host** 选项的值。

--disable-FEATURE

一些软件包可以选择这个选项来提供为大型选项的编译时配置，例如使用 **Kerberos**

认证系统或者一个实验性的编译器最优配置。如果默认是提供这些特性，可以使用 `--disable-FEATURE` 来禁用它，这里 `FEATURE` 是特性的名字。例如：

```
$ ./configure --disable-gui
--enable-FEATURE[=ARG]
```

相反的，一些软件包可能提供了一些默认被禁止的特性，可以使用 `--enable-FEATURE` 来起用它。这里 `FEATURE` 是特性的名字。一个特性可能会接受一个可选的参数。例如：

```
$ ./configure --enable-buffers=128
--enable-FEATURE=no 与上面提到的--disable-FEATURE 是同义的。
--with-PACKAGE[=ARG]
```

在自由软件社区里，有使用已有软件包和库的优秀传统。当用 `configure` 来配置一个源码树时，可以提供其他已经安装的软件包的信息。例如，倚赖于 `Tcl` 和 `Tk` 的 `BLT` 器件工具包。要配置 `BLT`，可能需要给 `configure` 提供一些关于我们把 `Tcl` 和 `Tk` 装的何处的信息：

```
$ ./configure --with-tcl=/usr/local --with-tk=/usr/local
--with-PACKAGE=no 与下面将提到的--without-PACKAGE 是同义的。
--without-PACKAGE
```

有时候你可能不想让你的软件包与系统已有的软件包交互。例如，你可能不想让你的新编译器使用 `GNU ld`。通过使用这个选项可以做到这一点：

```
$ ./configure --without-gnu-ld
--x-includes=DIR
```

这个选项是 `--with-PACKAGE` 选项的一个特例。在 `Autoconf` 最初被开发出来时，流行使用 `configure` 来作为 `Imake` 的一个变通方法来制作运行于 `X` 的软件。`--x-includes` 选项提供了向 `configure` 脚本指明包含 `X11` 头文件的目录的方法。

```
--x-libraries=DIR
```

类似的，`--x-libraries` 选项提供了向 `configure` 脚本指明包含 `X11` 库的目录的方法。

4.2 FFmpeg 裁剪优化实例

对 `ffmpeg` 的裁剪优化主要是对 `ffplay` 的裁剪优化，我们制定的需求是能播放测试文件（视频为 `mpeg4` 编码、音频为 `mp2` 编码，且为 `AVI` 复用），根据需求，找到相应的选项，或禁用或启用，最后的命令如下：

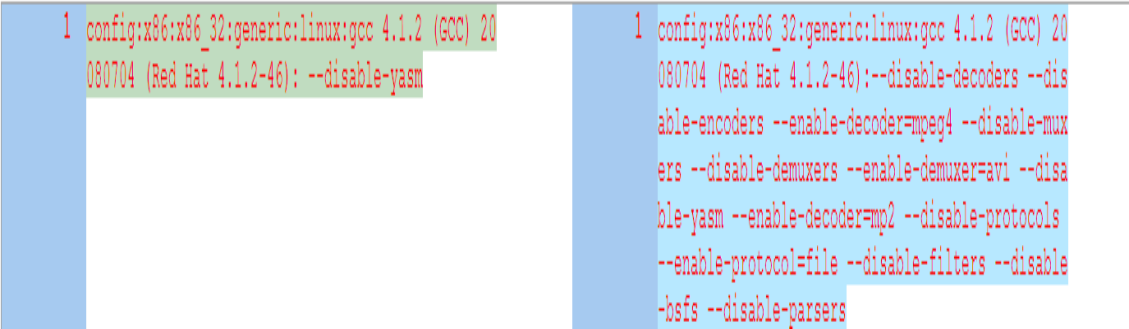
```
./configure --disable-yasm --disable-parsers --disable-decoders
--disable-encoders --enable-decoder=mpeg4 --disable-muxers
--disable-demuxers --enable-demuxer=avi --enable-decoder=mp2
--disable-protocols --enable-protocol=file --disable-filters --disable-bsfs
其中针对需求，--disable-parsers 为禁用所有解析器，--disable-decoders 为禁用所有解码器，--disable-encoders 为禁用所有编码器，--enable-decoder=mpeg4 为启用 mpeg4 的编码器，--disable-muxers 为禁用所有复用，--disable-demuxers 为禁用所有解复用，--enable-demuxer=avi 为启用 AVI 复用，--enable-decoder=mp2 为启用 mp2 编码，--disable-protocols 为禁用所有协议，--enable-protocol=file 为启用文件协议，--disable-filters 为禁用所有过滤器，--disable-bsfs 为禁用所有码流过滤器。通过以上配置
```

之后，编译，安装，就生成了我们要求的 ffplay，其大小为 1.8M（1864012 字节）。

此次是在 linux 环境下进行的，在以后的配置中，如果需要其他的什么编码器或什么的，按照选项要求进行配置即可。

4.3 裁剪优化前后文件比较

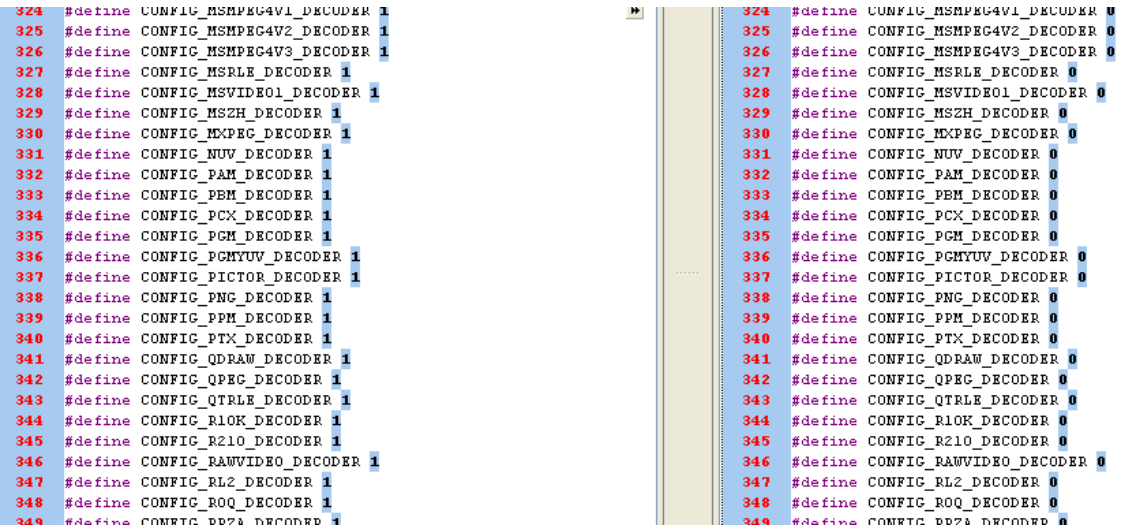
前面已经提到本次裁剪优化的内容。经过裁剪优化之后，对其文件夹进行比较，主要有 3 个地方不同，分别是 config.fate、config.h 和 config.mak。在 config.fate 中，其记录的是配置命令，由于前后两次配置命令不同，故相应内容也不同，如图 4-1 所示。在 config.h 中，其主要是根据配置命令来改变相应预定义的值，达到裁剪优化之效果如图 4-2 所示。在 config.mak 中，改变的也是配置命令中需要改变的选项，如图 4-3 所示。



```
1 config:x86:x86_32:generic:linux:gcc 4.1.2 (GCC) 20
080704 (Red Hat 4.1.2-46): --disable-yasm

1 config:x86:x86_32:generic:linux:gcc 4.1.2 (GCC) 20
080704 (Red Hat 4.1.2-46):--disable-decoders --dis
able-encoders --enable-decoder=mpg4 --disable-mux
ers --disable-demuxers --enable-demuxer=avi --disa
ble-yasm --enable-decoder=mp2 --disable-protocols
--enable-protocol=file --disable-filters --disable
-bsfs --disable-parsers
```

图 4-1 config.fate 前后比较



```
324 #define CONFIG_MSMPEG4V1_DECODER 1
325 #define CONFIG_MSMPEG4V2_DECODER 1
326 #define CONFIG_MSMPEG4V3_DECODER 1
327 #define CONFIG_MSRLE_DECODER 1
328 #define CONFIG_MSVIDEO1_DECODER 1
329 #define CONFIG_MSZH_DECODER 1
330 #define CONFIG_MPEG_DECODER 1
331 #define CONFIG_NUV_DECODER 1
332 #define CONFIG_PAM_DECODER 1
333 #define CONFIG_PBM_DECODER 1
334 #define CONFIG_PCX_DECODER 1
335 #define CONFIG_PGM_DECODER 1
336 #define CONFIG_PGM_YUV_DECODER 1
337 #define CONFIG_PICTOR_DECODER 1
338 #define CONFIG_PNG_DECODER 1
339 #define CONFIG_PPM_DECODER 1
340 #define CONFIG_PTX_DECODER 1
341 #define CONFIG_QDRAW_DECODER 1
342 #define CONFIG_QPEG_DECODER 1
343 #define CONFIG_QTRLE_DECODER 1
344 #define CONFIG_R10K_DECODER 1
345 #define CONFIG_R210_DECODER 1
346 #define CONFIG_RAWVIDEO_DECODER 1
347 #define CONFIG_RL2_DECODER 1
348 #define CONFIG_ROQ_DECODER 1
349 #define CONFIG_S302M_DECODER 1

324 #define CONFIG_MSMPEG4V1_DECODER 0
325 #define CONFIG_MSMPEG4V2_DECODER 0
326 #define CONFIG_MSMPEG4V3_DECODER 0
327 #define CONFIG_MSRLE_DECODER 0
328 #define CONFIG_MSVIDEO1_DECODER 0
329 #define CONFIG_MSZH_DECODER 0
330 #define CONFIG_MPEG_DECODER 0
331 #define CONFIG_NUV_DECODER 0
332 #define CONFIG_PAM_DECODER 0
333 #define CONFIG_PBM_DECODER 0
334 #define CONFIG_PCX_DECODER 0
335 #define CONFIG_PGM_DECODER 0
336 #define CONFIG_PGM_YUV_DECODER 0
337 #define CONFIG_PICTOR_DECODER 0
338 #define CONFIG_PNG_DECODER 0
339 #define CONFIG_PPM_DECODER 0
340 #define CONFIG_PTX_DECODER 0
341 #define CONFIG_QDRAW_DECODER 0
342 #define CONFIG_QPEG_DECODER 0
343 #define CONFIG_QTRLE_DECODER 0
344 #define CONFIG_R10K_DECODER 0
345 #define CONFIG_R210_DECODER 0
346 #define CONFIG_RAWVIDEO_DECODER 0
347 #define CONFIG_RL2_DECODER 0
348 #define CONFIG_ROQ_DECODER 0
349 #define CONFIG_S302M_DECODER 0
```

图 4-2 config.h 前后比较

<pre> 952 CONFIG_IMAGE2_MUXER=yes 953 CONFIG_IMAGE2PIPE_MUXER=yes 954 CONFIG_IPOD_MUXER=yes 955 CONFIG_IVF_MUXER=yes 956 CONFIG_M4V_MUXER=yes 957 CONFIG_MD5_MUXER=yes 958 CONFIG_MATROSKA_MUXER=yes 959 CONFIG_MATROSKA_AUDIO_MUXER=yes 960 CONFIG_MICRODVD_MUXER=yes 961 CONFIG_MJPEG_MUXER=yes 962 CONFIG_MLP_MUXER=yes 963 CONFIG_MMF_MUXER=yes 964 CONFIG_MOV_MUXER=yes 965 CONFIG_MP2_MUXER=yes 966 CONFIG_MP3_MUXER=yes 967 CONFIG_MP4_MUXER=yes 968 CONFIG_MPEG1SYSTEM_MUXER=yes 969 CONFIG_MPEG1VCD_MUXER=yes 970 CONFIG_MPEG1VIDEO_MUXER=yes 971 CONFIG_MPEG2DVD_MUXER=yes 972 CONFIG_MPEG2SVCD_MUXER=yes 973 CONFIG_MPEG2VIDEO_MUXER=yes 974 CONFIG_MPEG2VOB_MUXER=yes 975 CONFIG_MPEGTS_MUXER=yes </pre>	<pre> 952 !CONFIG_IMAGE2_MUXER=yes 953 !CONFIG_IMAGE2PIPE_MUXER=yes 954 !CONFIG_IPOD_MUXER=yes 955 !CONFIG_IVF_MUXER=yes 956 !CONFIG_M4V_MUXER=yes 957 !CONFIG_MD5_MUXER=yes 958 !CONFIG_MATROSKA_MUXER=yes 959 !CONFIG_MATROSKA_AUDIO_MUXER=yes 960 !CONFIG_MICRODVD_MUXER=yes 961 !CONFIG_MJPEG_MUXER=yes 962 !CONFIG_MLP_MUXER=yes 963 !CONFIG_MMF_MUXER=yes 964 !CONFIG_MOV_MUXER=yes 965 !CONFIG_MP2_MUXER=yes 966 !CONFIG_MP3_MUXER=yes 967 !CONFIG_MP4_MUXER=yes 968 !CONFIG_MPEG1SYSTEM_MUXER=yes 969 !CONFIG_MPEG1VCD_MUXER=yes 970 !CONFIG_MPEG1VIDEO_MUXER=yes 971 !CONFIG_MPEG2DVD_MUXER=yes 972 !CONFIG_MPEG2SVCD_MUXER=yes 973 !CONFIG_MPEG2VIDEO_MUXER=yes 974 !CONFIG_MPEG2VOB_MUXER=yes 975 !CONFIG_MPEGTS_MUXER=yes </pre>
--	--

图 4-3 config.mak 前后比较