# [FFmpeg filter简介](http://www.cnblogs.com/tocy/p/ffmpeg-filter-intro.html)

[时间：2016-08] [状态：Open]  
[关键词：FFmpeg, filter, filter graph，命令行]

## 引言及示例

FFmpeg中的libavfilter提供了一整套的基于filter的机制。filter本身是一个插件的形式，可以快速的组装需要的效果。  
比如下面的filter，可以实现视频的水平镜像效果。  
ffplay.exe sample.rmvb -vf hflip

## FFmpeg为什么重新定义filter API？

FFmpeg定义的libavcodec接口已经成为在编解码领域的事实上的行业标准。但音视频filter并没有类似的标准，多个不同的多媒体项目（比如MPlayer、Xine、GStreamer等）都实现了自定义的filter系统。为了统一filter库API接口，FFmpeg提出了参考DirectDraw实现了高质量、高效、灵活的音视频filter接口。详细的文档资料可以参考[FFmpeg filter](http://ffmpeg.org/ffmpeg-filters.html)。

## 传统概念上filter是什么？

本部分资料参考[filter-def](http://whatis.techtarget.com/definition/filter)  
filter可以翻译成过滤器，滤波器。物理概念上，常见的过滤器跟净化器概念重复，比如滤水器、空气净化器等。

* 在计算机程序中，filter是指一段代码，可用于检查输入或者输出，按照预定的规则处理并传递这些数据。换种说法，filter是一种传递（pass-through）代码块，将输入数据做特定的变换并输出。通常filter自身不做任何输入/输出。举个例子，linux下的grep可以认为是一个filter，按照正则表达式匹配选择，从输入中选择输出数据。
* 在电信工程领域，filter通常指的用于信号处理的设备，比如音频处理比较典型的低通滤波器、高通滤波器、带通滤波器、去噪滤波器等。

## filter的分类

按照处理数据的类型，通常多媒体的filter分为：

* 音频filter
* 视频filter
* 字幕filter

另一种按照处于编解码器的位置划分：

* prefilters: used before encoding
* intrafilters: used while encoding (and are thus an integral part of a video codec)
* postfilters: used after decoding

FFmpeg中filter分为：

* source filter （只有输出）
* audio filter
* video filter
* Multimedia filter
* sink filter （只有输入）

除了source和sink filter，其他filter都至少有一个输入、至少一个输出。

介绍了这么多，下面也是一个例子，使用filter实现宽高减半显示：

ffplay.exe sample.rmvb -vf scale=iw/2:ih/2

下面是使用mptestsrc的source filter作为ffplay输入，直接显示：

ffplay -f lavfi mptestsrc=t=dc\_luma  
ffplay -f lavfi life=s=300x200:mold=10:r=60:ratio=0.1:death\_color=#C83232:life\_color=#00ff00,scale=1200:800:flags=16

## 基本原理

FFmpeg filter可以认为是一些预定义的范式，可以实现类似积木的多种功能的自由组合。每个filter都有固定数目的输入和输出，而且实际使用中不允许有空悬的输入输出端。使用文本描述时我们可以通过标识符指定输入和输出端口，将不同filter串联起来，构成更复杂的filter。这就形成了嵌套的filter。当然每个filter可以通过ffmpeg/ffplay命令行实现，但通常filter更方便。

ffmpeg.exe、ffplay.exe能够通过filter处理原始的音视频数据。ffmpeg将filtergraph分为simple filtergraph和complex filtergraph。通常simple filtergraph只有一个输入和输出，ffmpeg命令行中使用-vf、-af识别，基本原理图如下：

\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_

| | | |

| decoded | | encoded data |

| frames |\ \_ | packets |

|\_\_\_\_\_\_\_\_\_| \ /||\_\_\_\_\_\_\_\_\_\_\_\_\_\_|

\ \_\_\_\_\_\_\_\_\_\_ /

simple \_\|| | / encoder

filtergraph | filtered |/

| frames |

|\_\_\_\_\_\_\_\_\_\_|

complex filtergraph，通常是具有多个输入输出文件，并有多条执行路径；ffmpeg命令行中使用-lavfi、-filter\_complex，基本原理图如下：

**\_\_\_\_\_***\_\_\_*\_

| |

| input 0 |\ **\_\_\_\_\_\_\_\_\_\_**

|**\_\_\_\_\_***\_\_\_*\_| \ | |

\ \_\_\_\_\_\_\_\_\_ /| output 0 |

\ | | / |\_\_\_\_\_\_\_\_\_\_|

**\_\_\_\_\_***\_\_\_*\_ \| complex | /

| | | |/

| input 1 |---->| filter |\

|**\_\_\_\_\_\_\_\_\_| | | \ \_\_\_\_\_\_\_***\_\_\_*

/| graph | \ | |

/ | | \| output 1 |

**\_\_\_\_\_\_\_\_\_ / |\_\_\_\_\_\_\_\_\_| |\_\_\_\_\_\_\_***\_\_\_*|

| | /

| input 2 |/

|**\_\_\_\_\_***\_\_\_*\_|

第三部分会介绍filtergraph的基本语法和构成。

在libavfilter, 一个filter可以包含多个输入、多个输出。下图是一个filtergraph的示例：

[main]

input --> split ---------------------> overlay --> output

| ^

|[tmp] [flip]|

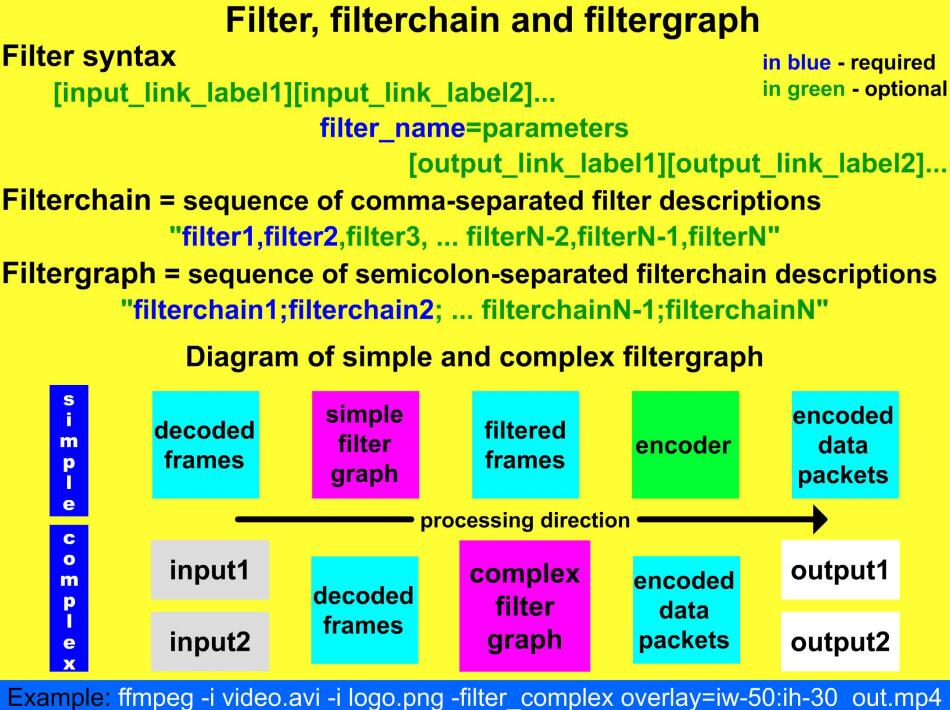
+-----> crop --> vflip -------+

上图中filtergraph将输入流分成两个流，其中一个通过crop filter和vflip filter，然后通过overlay filter将这两个流合成一个流输出。这个filtergraph可以用下面命令行表示：

ffmpeg -i INPUT -vf "split [main][tmp]; [tmp] crop=iw:ih/2:0:0, vflip [flip]; [main][flip] overlay=0:H/2" OUTPUT

## 语法识别

FFmpeg中filter包含三个层次，filter->filterchain->filtergraph。具体可以参考下图：



filter是ffmpeg的libavfilter提供的基础单元。在同一个线性链中的filter使用逗号分隔，在不同线性链中的filter使用分号隔开，比如下面的例子：

ffmpeg -i INPUT -vf "split [main][tmp]; [tmp] crop=iw:ih/2:0:0, vflip [flip]; [main][flip] overlay=0:H/2" OUTPUT

这里crop、vflip处于同一个线性链，split、overlay位于另一个线性链。二者连接通过命名的label实现（位于中括号中的是label的名字）。在上例中split filter有两个输出，依次命名为[main]和[tmp]；[tmp]作为crop filter输入，之后通过vflip filter输出[flip]；overlay的输入是[main]和[flilp]。如果filter需要输入参数，多个参数使用冒号分割。  
对于没有音频、视频输入的filter称为source filter，没有音频、视频输出的filter称为sink filter。

## 经典的filter

FFmpeg支持的所有filter可以通过[filters](http://ffmpeg.org/ffmpeg-filters.htm)查看。  
这里选几个相对经典的filter。

### 音频filter

* adelay filter  
  实现不同声道的延时处理。使用参数如下adelay=1500|0|500，这个例子中实现第一个声道的延迟1.5s，第三个声道延迟0.5s，第二个声道不做调整。
* aecho filter  
  实现回声效果，具体参考http://ffmpeg.org/ffmpeg-filters.html#aecho。
* amerge filter  
  将多个音频流合并成一个多声道音频流。具体参考http://ffmpeg.org/ffmpeg-filters.html#amerge-1。
* ashowinfo filter  
  显示每一个audio frame的信息，比如时间戳、位置、采样格式、采样率、采样点数等。具体参考http://ffmpeg.org/ffmpeg-filters.html#ashowinfo。
* panfilter  
  特定声道处理，比如立体声变为单声道，或者通过特定参数修改声道或交换声道。主要有两大类：  
  混音处理，比如下面的例子pan=1c|c0=0.9\*c0+0.1\*c1，实现立体声到单声道的变换；  
  声道变换，比如5.1声道顺序调整，pan="5.1| c0=c1 | c1=c0 | c2=c2 | c3=c3 | c4=c4 | c5=c5"。
* silencedetect和silenceremove filter  
  根据特定参数检测静音和移除静音。
* volume和volumedetect filter  
  这两个filter分别实现音量调整和音量检测。
* audio source filter  
  aevalsrc filter按照特定表达式生成音频信号。  
  anullsrc filter生成特定的原始音频数据，用于模板或测试。  
  anoisesrc filter生成噪声音频信号。  
  sine filter生成正弦波音频信号。
* audio sink filter  
  abuffersink filter和anullsink filter，这些filter只是用于特定情况下结束filter chain。

### 视频filter

* blend和tblend filter  
  将两帧视频合并为一帧。具体参数参考http://ffmpeg.org/ffmpeg-filters.html#blend\_002c-tblend。
* crop filter  
  按照特定分辨率裁剪输入视频，具体参数参考http://ffmpeg.org/ffmpeg-filters.html#crop。
* drawbox、drawgrid、drawtext filter  
  绘制box（对话框）、grid（表格）、text（文本）。
* edgedetect filter  
  边缘检测filter。
* fps filter  
  按照指定帧率输出视频帧（丢帧或者复制）。具体参考http://ffmpeg.org/ffmpeg-filters.html#fps-1。
* hflip、vflip filter  
  水平和垂直镜像。
* histogram filter  
  生成每帧的各颜色分量的直方图。
* noise filter  
  在输入视频帧中添加白噪声。
* overlay filter  
  视频叠加。具体参考http://ffmpeg.org/ffmpeg-filters.html#overlay-1。
* pad filter  
  视频边界填充。具体参考http://ffmpeg.org/ffmpeg-filters.html#pad-1。
* rotate filter  
  视频任意角度旋转。具体参考http://ffmpeg.org/ffmpeg-filters.html#rotate。
* scale filter  
  使用libswscale库完成视频缩放的filter。
* showinfo filter  
  显示视频帧的参数信息，比如时间戳、采样格式、帧类型等。
* subtitles filter  
  使用libass库绘制subtitle（字幕）。
* thumbnail filter  
  提取缩略图的filter。
* transpose filter  
  图像转置的filter。参数参考http://ffmpeg.org/ffmpeg-filters.html#transpose。
* source filter  
  主要有cellatuo、coreimagesrc、mptestsrc、life等filter，具体效果建议参考ffmpeg用户手册。
* source sink  
  主要有buffersink、nullsink两个filter。

### 多媒体filter

* ahistogram filter  
  将音频转化为视频输出，并显示为音量的直方图。
* concat filter  
  将音频流、视频流拼接成一个。具体参考http://ffmpeg.org/ffmpeg-filters.html#concat。
* metadata、ametadata filter  
  操作metadata信息。
* setpts、asetpts filter  
  改变输入音频帧或视频帧的pts。
* showfreqs、showspectrum、showspertrumpic、showvolume、showwaves filter  
  将输入音频转换为视频显示，并显示频谱、音量等信息
* split、asplit filter  
  将输入切分为多个相同的输出。
* source filter  
  主要是movie、amovie filter。从movie容器中读取音频或者视频帧。

## 实例demo

FFmpeg提供了很多有趣的filter实例，详见[Fancy Filtering Examples](https://trac.ffmpeg.org/wiki/FancyFilteringExamples)。  
我们这里先从几个简单的实例开始。

### 实例一：缩放scale

将输入缩小宽度缩小一半，并保持宽高比。

ffmpeg -i input.jpg -vf scale=iw/2:-1 output.jpg

### 实例二：filter、filterchain和filtergraph的使用

先将输入去交织，然后减半显示。  
以下三个命令是等价的。

# 2 chains form, one filter per chain, chains linked by the [middle] pad  
ffmpeg -i input -vf [in]yadif=0:0:0[middle];[middle]scale=iw/2:-1[out] output

# 1 chain form, with 2 filters in the chain, linking implied  
ffmpeg -i input -vf [in]yadif=0:0:0,scale=iw/2:-1[out] output

# the input and output are implied without ambiguity  
ffmpeg -i input -vf yadif=0:0:0,scale=iw/2:-1 output

### 实例三：2x2布局画面拼接

这个实例主要说明下filtergraph使用。命令行如下：

./ffmpeg -f lavfi -i testsrc -f lavfi -i testsrc -f lavfi -i testsrc -f lavfi -i testsrc -filter\_complex \

"[0:v]pad=iw*\*2:ih\**2[a]; \

[1:v]negate[b]; \

[2:v]hflip[c]; \

[3:v]edgedetect[d]; \

[a][b]overlay=w[x]; \

[x][c]overlay=0:h[y]; \

[y][d]overlay=w:h[out]" -map "[out]" -c:v ffv1 -t 5 multiple*\_input\_*grid.avi

## 参考资料

1. [libavfilter-multimedia](https://wiki.multimedia.cx/index.php?title=Libavfilter)
2. [FFmpeg filter HOWTO](https://wiki.multimedia.cx/index.php?title=FFmpeg_filter_howto)
3. [FFmpeg Filtering Guide](https://trac.ffmpeg.org/wiki/FilteringGuide)
4. [ffmpeg-filtering](http://ffmpeg.org/ffmpeg.html#Filtering)
5. [FFmpeg Bug Tracker and Wiki](https://trac.ffmpeg.org/wiki)

# 多媒体文件格式探测

[时间：2016-07] [状态：Open]

## 引言

前边整理了很多关于标准容器的资料，有点偏重理论化，本篇整理下实际点的，并结合ffmpeg的probe机制，分析下如何实现多媒体格式的探测（probe）。  
希望读完本文后，读者可以理解ffmpeg对avi、rm/rmvb、mkv、mp4、asf/wmv、ts、flv等文件探测的实现原理。

注意本文引用的FFmpeg 3.1，其他版本可能源码目录不太一样，但原理基本一致。

## FFmpeg探测机制

探测的前提是我们有一段媒体数据，基本原理就是根据媒体封装格式的特点，对全部已知的格式进行判断并设置一个分值，取其中最高的分值。  
FFmpeg中实现探测的函数是[av\_probe\_input\_buffer2](http://ffmpeg.org/doxygen/3.1/group__lavf__decoding.html#gafba896fafa5947fefeb47360f0bb0237)和[av\_probe\_input\_format3](http://ffmpeg.org/doxygen/3.1/group__lavf__decoding.html#ga7e4318244afd6c5fe55cec75d27258b2)。其核心代码如下：

// code in av\_probe\_input\_format3 (segment)

while ((fmt1 = av\_iformat\_next(fmt1))) {

if (!is\_opened == !(fmt1->flags & AVFMT\_NOFILE) && strcmp(fmt1->name, "image2"))

continue;

score = 0;

if (fmt1->read\_probe) {

score = fmt1->read\_probe(&lpd);

if (score)

av\_log(NULL, AV\_LOG\_TRACE, "Probing %s score:%d size:%d\n", fmt1->name, score, lpd.buf\_size);

if (fmt1->extensions && av\_match\_ext(lpd.filename, fmt1->extensions)) {

switch (nodat) {

case NO\_ID3:

score = FFMAX(score, 1);

break;

case ID3\_GREATER\_PROBE:

case ID3\_ALMOST\_GREATER\_PROBE:

score = FFMAX(score, AVPROBE\_SCORE\_EXTENSION / 2 - 1);

break;

case ID3\_GREATER\_MAX\_PROBE:

score = FFMAX(score, AVPROBE\_SCORE\_EXTENSION);

break;

}

}

} else if (fmt1->extensions) {

if (av\_match\_ext(lpd.filename, fmt1->extensions))

score = AVPROBE\_SCORE\_EXTENSION;

}

if (av\_match\_name(lpd.mime\_type, fmt1->mime\_type)) {

if (AVPROBE\_SCORE\_MIME > score) {

av\_log(NULL, AV\_LOG\_DEBUG, "Probing %s score:%d increased to %d due to MIME type\n", fmt1->name, score, AVPROBE\_SCORE\_MIME);

score = AVPROBE\_SCORE\_MIME;

}

}

if (score > score\_max) {

score\_max = score;

fmt = fmt1;

} else if (score == score\_max)

fmt = NULL;

}

上面代码的原理就是优先使用demuxer的read\_probe函数，然后参考文件名的后缀。至于AVInputFormat的定义，建议参考http://ffmpeg.org/doxygen/3.1/structAVInputFormat.html。

## read\_probe实现

一般probe有两种情况。  
对于在文件开头存在特征码的封装格式，比如avi、rm/rmvb、flv、mkv、asf可以直接使用特征码。  
AVI的特征码是RIFF AVI，下面是FFmpeg中的avi探测函数实现：

// from libavformat/avidec.c

static const char avi\_headers[][8] = {

{ 'R', 'I', 'F', 'F', 'A', 'V', 'I', ' ' },

{ 'R', 'I', 'F', 'F', 'A', 'V', 'I', 'X' },

{ 'R', 'I', 'F', 'F', 'A', 'V', 'I', 0x19 },

{ 'O', 'N', '2', ' ', 'O', 'N', '2', 'f' },

{ 'R', 'I', 'F', 'F', 'A', 'M', 'V', ' ' },

{ 0 }

};

static int avi\_probe(AVProbeData \*p)

{

int i;

/\* check file header \*/

for (i = 0; avi\_headers[i][0]; i++)

if (AV\_RL32(p->buf ) == AV\_RL32(avi\_headers[i] ) &&

AV\_RL32(p->buf + 8) == AV\_RL32(avi\_headers[i] + 4))

return AVPROBE\_SCORE\_MAX;

return 0;

}

rm/rmvb文件的特征码是.RMF、.ra，下面是FFmpeg中rm探测函数：

// from libavformat/rmdec.c

static int rm\_probe(AVProbeData \*p)

{

/\* check file header \*/

if ((p->buf[0] == '.' && p->buf[1] == 'R' &&

p->buf[2] == 'M' && p->buf[3] == 'F' &&

p->buf[4] == 0 && p->buf[5] == 0) ||

(p->buf[0] == '.' && p->buf[1] == 'r' &&

p->buf[2] == 'a' && p->buf[3] == 0xfd))

return AVPROBE\_SCORE\_MAX;

else

return 0;

}

flv文件的特征码是FLV，下面是FFmpeg中flv探测函数：

// from libavformat/flvdec.c

static int probe(AVProbeData \*p, int live)

{

const uint8\_t \*d = p->buf;

unsigned offset = AV\_RB32(d + 5);

if (d[0] == 'F' &&

d[1] == 'L' &&

d[2] == 'V' &&

d[3] < 5 && d[5] == 0 &&

offset + 100 < p->buf\_size &&

offset > 8) {

int is\_live = !memcmp(d + offset + 40, "NGINX RTMP", 10);

if (live == is\_live)

return AVPROBE\_SCORE\_MAX;

}

return 0;

}

static int flv\_probe(AVProbeData \*p)

{

return probe(p, 0);

}

mkv的特征码就是EBML\_Header\_ID和已知的DocType，下面是FFmpeg的mkv探测函数：

// from libavformat/matroskadec.c

/\* top-level master-IDs \*/

#define EBML\_ID\_HEADER 0x1A45DFA3

static int matroska\_probe(AVProbeData \*p)

{

uint64\_t total = 0;

int len\_mask = 0x80, size = 1, n = 1, i;

/\* EBML header? \*/

if (AV\_RB32(p->buf) != EBML\_ID\_HEADER)

return 0;

/\* length of header \*/

total = p->buf[4];

while (size <= 8 && !(total & len\_mask)) {

size++;

len\_mask >>= 1;

}

if (size > 8)

return 0;

total &= (len\_mask - 1);

while (n < size)

total = (total << 8) | p->buf[4 + n++];

/\* Does the probe data contain the whole header? \*/

if (p->buf\_size < 4 + size + total)

return 0;

/\* The header should contain a known document type. For now,

\* we don't parse the whole header but simply check for the

\* availability of that array of characters inside the header.

\* Not fully fool-proof, but good enough. \*/

for (i = 0; i < FF\_ARRAY\_ELEMS(matroska\_doctypes); i++) {

size\_t probelen = strlen(matroska\_doctypes[i]);

if (total < probelen)

continue;

for (n = 4 + size; n <= 4 + size + total - probelen; n++)

if (!memcmp(p->buf + n, matroska\_doctypes[i], probelen))

return AVPROBE\_SCORE\_MAX;

}

// probably valid EBML header but no recognized doctype

return AVPROBE\_SCORE\_EXTENSION;

}

ASF/WMV文件的特征码就是ASF\_HEADER\_GUID，下面是FFmpeg中asf探测函数：

// from libavformat/asfdec\_f.c

const ff\_asf\_guid ff\_asf\_header = {

0x30, 0x26, 0xB2, 0x75, 0x8E, 0x66, 0xCF, 0x11, 0xA6, 0xD9, 0x00, 0xAA, 0x00, 0x62, 0xCE, 0x6C

};

static int asf\_probe(AVProbeData \*pd)

{

/\* check file header \*/

if (!ff\_guidcmp(pd->buf, &ff\_asf\_header))

return AVPROBE\_SCORE\_MAX;

else

return 0;

}

MP4文件是由不同BOX构成的，需要根据box type来探测实际类型，FFmpeg中mp4探测函数（也包括mov,m4a,3gp,3g2）如下：

static int mov\_probe(AVProbeData \*p)

{

int64\_t offset;

uint32\_t tag;

int score = 0;

int moov\_offset = -1;

/\* check file header \*/

offset = 0;

for (;;) {

/\* ignore invalid offset \*/

if ((offset + 8) > (unsigned int)p->buf\_size)

break;

tag = AV\_RL32(p->buf + offset + 4);

switch(tag) {

/\* check for obvious tags \*/

case MKTAG('m','o','o','v'):

moov\_offset = offset + 4;

case MKTAG('m','d','a','t'):

case MKTAG('p','n','o','t'): /\* detect movs with preview pics like ew.mov and april.mov \*/

case MKTAG('u','d','t','a'): /\* Packet Video PVAuthor adds this and a lot of more junk \*/

case MKTAG('f','t','y','p'):

if (AV\_RB32(p->buf+offset) < 8 &&

(AV\_RB32(p->buf+offset) != 1 ||

offset + 12 > (unsigned int)p->buf\_size ||

AV\_RB64(p->buf+offset + 8) == 0)) {

score = FFMAX(score, AVPROBE\_SCORE\_EXTENSION);

} else if (tag == MKTAG('f','t','y','p') &&

( AV\_RL32(p->buf + offset + 8) == MKTAG('j','p','2',' ')

|| AV\_RL32(p->buf + offset + 8) == MKTAG('j','p','x',' ')

)) {

score = FFMAX(score, 5);

} else {

score = AVPROBE\_SCORE\_MAX;

}

offset = FFMAX(4, AV\_RB32(p->buf+offset)) + offset;

break;

/\* those are more common words, so rate then a bit less \*/

case MKTAG('e','d','i','w'): /\* xdcam files have reverted first tags \*/

case MKTAG('w','i','d','e'):

case MKTAG('f','r','e','e'):

case MKTAG('j','u','n','k'):

case MKTAG('p','i','c','t'):

score = FFMAX(score, AVPROBE\_SCORE\_MAX - 5);

offset = FFMAX(4, AV\_RB32(p->buf+offset)) + offset;

break;

case MKTAG(0x82,0x82,0x7f,0x7d):

case MKTAG('s','k','i','p'):

case MKTAG('u','u','i','d'):

case MKTAG('p','r','f','l'):

/\* if we only find those cause probedata is too small at least rate them \*/

score = FFMAX(score, AVPROBE\_SCORE\_EXTENSION);

offset = FFMAX(4, AV\_RB32(p->buf+offset)) + offset;

break;

default:

offset = FFMAX(4, AV\_RB32(p->buf+offset)) + offset;

}

}

if(score > AVPROBE\_SCORE\_MAX - 50 && moov\_offset != -1) {

/\* moov atom in the header - we should make sure that this is not a

\* MOV-packed MPEG-PS \*/

offset = moov\_offset;

while(offset < (p->buf\_size - 16)){ /\* Sufficient space \*/

/\* We found an actual hdlr atom \*/

if(AV\_RL32(p->buf + offset ) == MKTAG('h','d','l','r') &&

AV\_RL32(p->buf + offset + 8) == MKTAG('m','h','l','r') &&

AV\_RL32(p->buf + offset + 12) == MKTAG('M','P','E','G')){

av\_log(NULL, AV\_LOG\_WARNING, "Found media data tag MPEG indicating this is a MOV-packed MPEG-PS.\n");

/\* We found a media handler reference atom describing an

\* MPEG-PS-in-MOV, return a

\* low score to force expanding the probe window until

\* mpegps\_probe finds what it needs \*/

return 5;

}else

/\* Keep looking \*/

offset+=2;

}

}

return score;

}

对于像TS这种流媒体格式，通常通过0x47同步字节和pid判断，FFmpeg中ts的探测函数如下：

// from libavformat/mpegts.c

static int analyze(const uint8\_t \*buf, int size, int packet\_size,

int probe)

{

int stat[TS\_MAX\_PACKET\_SIZE];

int stat\_all = 0;

int i;

int best\_score = 0;

memset(stat, 0, packet\_size \* sizeof(\*stat));

for (i = 0; i < size - 3; i++) {

if (buf[i] == 0x47) {

int pid = AV\_RB16(buf+1) & 0x1FFF;

int asc = buf[i + 3] & 0x30;

if (!probe || pid == 0x1FFF || asc) {

int x = i % packet\_size;

stat[x]++;

stat\_all++;

if (stat[x] > best\_score) {

best\_score = stat[x];

}

}

}

}

return best\_score - FFMAX(stat\_all - 10\*best\_score, 0)/10;

}

static int mpegts\_probe(AVProbeData \*p)

{

const int size = p->buf\_size;

int maxscore = 0;

int sumscore = 0;

int i;

int check\_count = size / TS\_FEC\_PACKET\_SIZE;

#define CHECK\_COUNT 10

#define CHECK\_BLOCK 100

if (!check\_count)

return 0;

for (i = 0; i<check\_count; i+=CHECK\_BLOCK) {

int left = FFMIN(check\_count - i, CHECK\_BLOCK);

int score = analyze(p->buf + TS\_PACKET\_SIZE \*i, TS\_PACKET\_SIZE \*left, TS\_PACKET\_SIZE , 1);

int dvhs\_score = analyze(p->buf + TS\_DVHS\_PACKET\_SIZE\*i, TS\_DVHS\_PACKET\_SIZE\*left, TS\_DVHS\_PACKET\_SIZE, 1);

int fec\_score = analyze(p->buf + TS\_FEC\_PACKET\_SIZE \*i, TS\_FEC\_PACKET\_SIZE \*left, TS\_FEC\_PACKET\_SIZE , 1);

score = FFMAX3(score, dvhs\_score, fec\_score);

sumscore += score;

maxscore = FFMAX(maxscore, score);

}

sumscore = sumscore \* CHECK\_COUNT / check\_count;

maxscore = maxscore \* CHECK\_COUNT / CHECK\_BLOCK;

ff\_dlog(0, "TS score: %d %d\n", sumscore, maxscore);

if (check\_count > CHECK\_COUNT && sumscore > 6) {

return AVPROBE\_SCORE\_MAX + sumscore - CHECK\_COUNT;

} else if (check\_count >= CHECK\_COUNT && sumscore > 6) {

return AVPROBE\_SCORE\_MAX/2 + sumscore - CHECK\_COUNT;

} else if (check\_count >= CHECK\_COUNT && maxscore > 6) {

return AVPROBE\_SCORE\_MAX/2 + sumscore - CHECK\_COUNT;

} else if (sumscore > 6) {

return 2;

} else {

return 0;

}

}

## 总结

说白了，媒体文件封装格式的探测都是通过其标准定义的特征字段及结构进行匹配检索。只要对封装格式有所了解基本上可以快速的确定文件封装格式。