FFMpeg 中比较重要的函数以及数据结构如下：

1. 数据结构：
   - **(1) AVFormatContext**
   - **(2) AVOutputFormat**
   - **(3) AVInputFormat**
   - **(4) AVCodecContext**
   - **(5) AVCodec**
   - **(6) AVFrame**
   - **(7) AVPacket**
   - **(8) AVPicture**
   - **(9) AVStream**
2. 初始化函数：
   - (1) av_register_all()
   - (2) avcodec_open()
   - (3) avcodec_close()
   - (4) av_open_input_file()
   - (5) av_find_input_format()
   - (6) av_find_stream_info()
   - (7) av_close_input_file()
3. 音视频编解码函数：
   - (1) avcodec_find_decoder()
   - (2) avcodec_alloc_frame()
   - (3) avpicture_get_size()
   - (4) avpicture_fill()
   - (5) img_convert()
   - (6) avcodec_alloc_context()
   - (7) avcodec_decode_video()
   - (8) av_free_packet()
   - (9) av_free()
4. 文件操作：
   - (1) avnew_steam()
   - (2) av_read_frame()
   - (3) av_write_frame()
   - (4) dump_format()
5. 其他函数：
   - (1) avpicture_deinterlace()
   - (2) ImgReSampleContext()

以下就根据，以上数据结构及函数在 ffmpeg 测试代码 output_example.c 中出现的前后顺进行分析。在此之前还是先谈一下 ffmpeg 的编译问题。在 linux 下的编译比较简单，这里不多说了。在 windows 下的编译可以参考以下网页：

http://bbs.chinavideo.org/viewthread.php?tid=1897&extra=page%3D1

值得一提的是，在使用编译后的 sdk 进行测试时（用到 ffmpeg 目录下的 output_example.c）编译过程中可能会有以下两个问题：

1. Output_example.c 用到了 snprintf.h 这个头文件。然而这个头文件在 win 下和 linux 下有所不同。具体在 win 下可以用以下方法解决：

   http://www.ijs.si/software/snprintf/

2. 如果使用 vc6，或是 vc6 的命令行进行编译，inline 可能不认。错误会出现在 common.h 文件中，可以在 common.h 中加入

   #ifdef _MSC_VAR

   #define inline __inline

   #endif

交待完毕进入正题。

一．FFMpeg 中的数据结构：

I. AVFormatContext

一般在使用 ffmpeg sdk 的代码中 AVFormatContext 是一个贯穿始终的数据结构，很多函数都要用到它作为参数。FFmpeg 代码中对这个数据结构的注释是：format I/O context

此结构包含了一个视频流的格式内容。其中存有了 AVInputFormat（or AVOutputFormat 同一时间 AVFormatContext 内只能存在其中一个），和 AVStream、AVPacket 这几个重要的数据结构以及一些其他的相关信息，比如 title,author,copyright 等。还有一些可能在编解码中会用到的信息，诸如：duration, file_size, bit_rate 等。参考 avformat.h 头文件。

Useage:

声明：

AVFormatContext *oc; **(1)**

初始化： 由于 AVFormatConext 结构包含许多信息因此初始化过程是分步完成，而且有些变量如果没有值可用，也可不初始化。但是由于一般声明都是用指针因此一个分配内存过程不可少：

oc = av_alloc_format_context(); **(2)**

结构中的 AVInputFormat*（或 AVOutputFormat*）是一定要初始化的，基本上这是编译码要使用什么 codec 的依据所在：

oc->oformat = fmt; or oc->iformat = fmt; **(3)**

其中 AVOutputFormat* fmt 或 AVInputFormat* fmt。（AVInputFormat and AVOutputFormat 的初始化在后面介绍。随后在参考代码 output_example.c 中有一行：

snprintf(oc-filename, sizeof(oc->filename), “%s”, filename); **(4)**

还不是十分清楚有什么作用，估计是先要在输出文件中写一些头信息。

在完成以上步骤後，（初始化完毕AVInputFormat*（或AVOutputFormat*）以及AVFormatContext）接下来就是要利用oc初始化本节开始讲到的AVFormatContext中的第二个重要结构。AVStream（假设已经有了声明AVStream *video_st。参考代码中用了一个函数来完成初始化，当然也可以在主函数中做，传递进函数的参数是oc 和fmt->video_codec（这个在下一节介绍**(29)**）：

```
vdeo_st = add_video_stream(oc, fmt->video_codec);                    (5)
```

此函数会在后面讲到 AVStream 结构时分析。

AVFormatContext 最后的一个设置工作是：

```
if( av_set_paramters(oc,NULL) < 0){                                  (6)
//handle error;
}
dump_format(oc, 0, filename, 1);                                     (7)
```

作用就是看看先前的初始化过程中设置的参数是否符合规范，否则将报错。

上面讲的都是初始化的过程，包括 AVFormatContext 本身的和利用 AVFormatContext 初始化其他数据结构的。接下来要讲讲整个的编解码过程。我想先将 ouput_example.c 中 main 函数内的编解码函数框架描述一下。这样比较清晰，而且编码者为了结构清晰，在写 ouput_example.c 的过程中也基本上在 main 函数中只保持 AVFormatContext 和 AVStream 两个数据结构（AVOutputFormat 其实也在但是包含在 AVFormatContext 中了）。

```
// open video codec and allocate the necessary encode buffers
if(video_st)
        open_video(oc, video_st);                                    (8)

// write the stream header, if any
av_write_header(oc);                                                 (9)

// encode and decode process
for(; ;){
        write_video_frame(oc, video_st);                             (10)
        // break condition…here
}

//close codec
if(video_st)
        close_video(oc, video_st);                                   (11)

//write the trailer , if any
av_write_trailer(oc);                                                (12)

// free the streams
for(i=0; i<oc->b_streams; i++){
        av_freep(&oc->streams[i]->codec);                            (13)
        av_freep(&oc->streams[i]);                                   (14)
}

//close the ouput file
if(!(fmt->flags & AVFMT_NOFILE)){
        url_fclose(&oc->pb);                                         (15)
}

av_free(oc);                                                         (16)
```

通过以上的一串代码，就可以清晰地看出 AVFormatContex* oc 和 AVStream* video_st 是在使用 ffmpeg SDK 开发时贯穿始终的两个数据结构。以下，简要介绍一下三个标为红色的函数，他们是参考代码 output_example.c 开发者自行定义的函数。这样可以使整个代码结构清晰，当然你在使用 ffmpeg SDK 时也可以在主函数中完成对应的功能。在后面我们会专门针对这三个函数做分析。

1. **open_video(oc, video_st);**

此函数主要是对视频编码器（或解码器）的初始化过程。初始化的数据结构为 AVCodec* codec 和 AVCodecContext* c 包括用到了的 SDK 函数有：

c = st->codec;

```
codec = avcodec_find_encoder(c->codec_id); //编码时，找编码器          (17)
codec = avcodec_find_decoder(c->codec_id); //解码时，找解码器          (18)
```

AVCodecContex是结构AVStream中的一个数据结构，因此在AVStream初始化後**(5)**直接复值给c。

```
// internal open video codec
avcodec_open(c,codec);                                                (19)

// allocate video stream buffer
// AVFrame *picture
// uint8_t *video_outbuf
video_outbuf_size=200000;
video_outbuf = av_maloc(video_outbuf_size);                           (20)

// allocate video frame buffer
picture = alloc_picture(c->pix_fmt, c->width, c->height);             (21)
```

上述三步比较容易理解，打开视频编解码codec、分配输出流缓存大小、分配每一帧图像缓存大小。其中AVFrame也是ffmpeg中主要数据结构之一。这一步**(8)**是对编解码器的初始化过程。

2. **write_video_frame(AVFormatContext *oc, AVStream *st)**

这个函数中做了真正的编解码工作，其中的函数比较复杂先列出来慢慢分析。
用到的数据结构有 AVCodecContext *c, SwsContext *img_convert_ctx。其中 SwsContext 是用来变换图像格式的。比如 yuv422 变到 yuv420 等，当然也用到函数，见下面列表。

```
fill_yuv_image(tmp_picture, frame_count, c->width, c->height);        (22)


sws_scale(img_convert_ctx, tmp_picture->, tmp_picture->linesize,
          0, c->height, picture->data, picture->linesize);            (23)

img_convert_ctx = sws_getContxt(c->width, c->height, PIX_FMT_YUV420P,  (24)
                  c->width, c->heigth, c->pix_fmt, sws_flags, NULL, NULL, NULL);
```

由于参考代码中做的是一个编码。因此，它总是要求编码器输入的是 yuv 文件，而且是 yuv420 格式的。就会有了以上一些处理过程。接下来调用编码器编码，数据规则化（打包）用到 AVPacket，这也是 ffmpeg 中一个比较不好理解的地方。

```
out_size = avcodec_encode_video(c, video_outbuf, video_outbuf_size, picture);   (25)

AVPacket pkt;
av_init_packet(&pkt);                                                 (26)
//……handle pkt process, we will analyze later

ret = av_write_frame(oc, &pkt);                                       (27)
```

有 encode 就一定会有 decode。而且 ffmpeg 专为解码而生，但是为什么在参考代码中只用了 encoder 呢？个人猜想是因为 encode 只是用 yuv420 来编码，这样的 yuv420 生成比较容易，要是用到解码的化，还要在代码中附带一个其他格式的音视频文件。在源代码 libavcodec 文件夹中有一个 apiexample.c 的参考代码，其中就做了编解码。有空的化我会分析一下。

3. **close_video(AVFormatContext *oc, AVStream *st)**

```
avcodec_close(st->codec);
av_free(picture->data[0]);
av_free(picture);
av_free(video_outbuf);
```

比较容易理解，不多说了。

以上一大段虽然名为介绍AVFormatContext。但基本上把ouput_example.c的视频编码部分的框架走了一遍，其一是想说明结构AVFormatContext的重要性，另一方面也是希望对使用FFMpeg SDK开发者有一个大致的框架。

其实，真正的一些编码函数，内存分配函数在SDK中都已经封装好了，只要搞清楚结构就能用了。而开发者要做的就是一些初始化的过程，基本上就是针对数据结构1的初始化。

II. AVOutputFormat

虽然简单（初始化）但是十分重要，他是编解码器将要使用哪个 codec 的“指示”。在其成员数据中最重要的就是关于视频 codec 的了：enum CodecID video_codec;

```
AVOutputFormat *fmt;
fmt = guess_format(NULL, filename, NULL);                    (28)
```

根据 filename 来判断文件格式，同时也初始化了用什么编码器。当然，如果是用 AVInputFormat *fmt 的化，就是 fix 用什么解码器。（指定输出序列->fix 编码器，指定输入序列->fix 解码器？）

III. AVStream

AVStream作为继AVFormatContext後第二个贯穿始终的结构是有其理由的。他的成员数据中有AVCodecContext这基本的上是对所使用的Video Codec的参数进行设定的（包括bit rate、分辨率等重要信息）。同时作为“Stream”，它包含了“流”这个概念中的一些数据，比如：帧率（r_frame_rate）、基本时间计量单位（time_base）、（需要编解码的）首帧位置（start_time）、持续时间（duration）、帧数（nb_frames）以及一些ip信息。当然后面的这些信息中有些不是必须要初始化的，但是AVCodecContex是一定要初始化的，而且就是作为初始化AVStream最重要的一个部分。我们在前面就谈到了AVStream的初始化函数**(5)**，现在来看看他是怎么做的：

```
// declaration
AVStream *video_st;
video_st = add_video_stream(oc, fmt->video_codec);

static AVStream *add_video_stream(AVFormatContex *oc, int codec_id){        (29)
    AVCodecContext *c;          // member of AVStream, which will be initialized here
    AVStream *st;               // temporary data, will be returned

    st = av_new_stream(oc, 0);                                              (30)
    c = st->codec;

    // 以下基本是针对 c 的初始化过程。包括比特率、分辨率、GOP 大小等。
    ……
    // 以下的两行需要注意一下，特别是使用 MP4 的
if(!strcmp(oc->oformat->name, “mp4”) || !strcmp(oc->oformat->name, “mov”)
|| !strcmp(oc->oformat->name, “3gp”))
        c->flags |= CODEC_FLAG_GLOBAL_HEADER;

    // 将 st 传给 video_st;
    return st;
}
```

以上代码中，有几点需要注意的。一个是**(30)**和c = st->codec是一定要做的，当然这是编程中最基本的问题，**(30)**是将st这个AVSteam绑定到AVFormatContext* oc上。后面的c = st->codec是将c绑定到st的AVCodecContext上。其二是对c的初始化过程中，ouput_example.c里做的是一些基本的配置，当然作为使用者的你还希望对codec加入其他的一些编解码的条件。可以参考avcodec.h里关于AVCodecContext结构的介绍，注释比较详细的。

关于AVStream的使用在前面介绍AVFormatContext时已有所涉及，在主函数中三个编解码函数中**(8)**、**(10)**和**(11)**中。观察相关的代码，可以发现主要还是将AVStream中的AVCodecContext提取出来，再从中提取出AVCodec结构如在**(8)**中：

```
// open_video(oc, video_st);
// AVFormatContext *oc, AVStream *st
AVCodec *codec;
AVCodecContext *c;

c = st->codec;
codec = avcodec_find_encoder(c->codec_id);                              (31)

// open the codec
avcodec_open(c, codec);                                                 (32)
```

同样，我们可以看到在**(10)**(write_video_frame())中AVFrame也是做为传递AVCodecContext结构的载体而存在。（11）(close_video())比较简单，不熬述。

IV. AVCodecContext

此结构在Ffmpeg SDK中的注释是：main external api structure其重要性可见一斑。而且在avcodec它的定义处，对其每个成员变量，都给出了十分详细的介绍。应该说AVCodecContext的初始化是Codec使用中最重要的一环。虽然在前面的AVStream中已经有所提及，但是这里还是要在说一遍。AVCodecContext作为Avstream的一个成员结构，必须要在Avstream初始化後(30)再对其初始化(AVStream的初始化用到AVFormatContex)。虽然成员变量比较多，但是这里只说一下在output_example.c中用到了，其他的请查阅avcodec.h文件中介绍。

```
// static AVStream *add_video_stream(AVFormatContext *oc, int codec_id)
AVCodecContext *c;
st = av_new_stream(oc, 0);
c = st->codec;
c->codec_id = codec_id;
c->codec_type = CODEC_TYPE_VIDEO;

c->bit_rate = 400000;               // 400 kbits/s
c->width = 352;
c->height = 288;                    // CIF

// 帧率做分母，秒做分子，那么 time_base 也就是一帧所用时间。（时间基!）
c->time_base.den = STREAM_FRAME_RATE;
c->time_base.num = 1;

c->gop_size =12;
// here define:
// #define STREAM_PIX_FMT PIX_FMT_YUV420P
// pixel format, see PIX_FMT_xxx
// -encoding: set by user.
// -decoding: set by lavc.
c->pix_fmt = STREAM_PIX_FMT;
```

除了以上列出了的。还有诸如指定运动估计算法的：**me_method**。量化参数、最大 b 帧数：**max_b_frames**。码率控制的参数、差错掩盖 **error_concealment**、模式判断模式：**mb_decision** (这个参数蛮有意思的，可以看看 avcodec.h 1566 行)、Lagrange multipler 参数：**lmin & lmax** 和 宏块级 Lagrange multipler 参数：**mb_lmin & mb_lmax**、constant quantization parameter rate control method: **cqp** 等。

值得一提的是在 AVCodecContext 中有两个成员数据结构：***AVCodec***、***AVFrame***。AVCodec 记录了所要使用的 Codec 信息并且含有 5 个函数：init、encoder、close、decode、flush 来完成编解码工作（参见 avcode.h 2072 行）。AVFrame 中主要是包含了编码後的帧信息，包括本帧是否是 key frame、*data[4]定义的 Y、Cb 和 Cr 信息等，随后详细介绍。

初始化後，可以说AVCodecContext在**(8)**&**(10)**中大显身手。先在(8)open_video()中初始化AVCodec *codec以及AVFrame* picture：

```
// AVCodecContext *c;
codec = avcodec_find_encoder(c->codec_id);
……
picture = alloc_picture(PIX_FMT_YUV420P, c->width, c->height);
```

後在 writer_video_frame(AVFormatContext *oc, AVStream *st)中作为一个编解码器的主要参数被利用：

```
AVCodecContext *c;
c = st->codec;
……
out_size = avcodec_encode_video(c, video_outbuf, video_outbuf_size, picture);
```

V. AVCodec

结构 AVCodec 中成员变量和成员函数比较少，但是很重要。他包含了 CodecID，也就是用哪个 Codec、

像素格式信息。还有前面提到过的 5 个函数（init、encode、close、decoder、flush）。顺便提一下，虽然在参考代码 output_example.c 中的编码函数用的是 avcodec_encode_video（），我怀疑在其中就是调用了 AVCodec 的 encode 函数，他们传递的参数和返回值都是一致的，当然还没有得到确认，有兴趣可以看看 ffmpeg 源代码。在参考代码中，AVCodec 的初始化後的使用都是依附于 AVCodecContex，前者是后者的成员。在 AVCodecContext 初始化後（add_video_stream()），AVCodec 也就能很好的初始化了：

```
//初始化
codec = avcodec_find_encoder(c->codec_id);                    (33)
//打开 Codec
avcodec_open（c, codec）                                      (34)
```

VI. AVFrame

AVFrame 是个很有意思的结构，它本身是这样定义的：

```
typedef struct AVFrame {
    FF_COMMON_FRAME
}AVFrame;
```

其中，FF_COMMON_FRAME 是以一个宏出现的。由于在编解码过程中 AVFrame 中的数据是要经常存取的。为了加速，要采取这样的代码手段。

AVFrame 是作为一个描述“原始图像”（也就是 YUV 或是 RGB…还有其他的吗？）的结构，他的头两个成员数据，uint8_t *data[4]，int linesize[4]，第一个存放的是 Y、Cb、Cr（yuv 格式），linesize 是啥？由这两个数据还能提取处另外一个数据结构：

```
typedef struct AVPicture {
    uint8_t *data[4];
    int linesize[4];                    // number of bytes per line
}AVPicture ;
```

此外，AVFrame 还含有其他一些成员数据，比如。是否 key_frame、已编码图像书 coded_picture_number、是否作为参考帧 reference、宏块类型 *mb_type 等等（avcodec.h 446 行）。

AVFrame 的初始化并没有他结构上看上去的那么简单。由于 AVFrame 还有一个承载图像数据的任务（data[4]）因此，对他分配内存应该要小心完成。output_example.c 中提供了 alloc_picute()来完成这项工作。参考代码中定义了两个全局变量：AVFrame *picture，*tmp_picture。（如果使用 yuv420 格式的那么只用到前一个数据 picture 就行了，将图像信息放入 picture 中。如果是其他格式，那么先要将 yuv420 格式初始化后放到 tmp_picture 中在转到需求格式放入 picture 中。）在 open_video（）打开编解码器后初始化 AVFrame：

```
picture = alloc_picture(c->pix_fmt, c->width, c->height);
tmp_picture = alloc_picture(PIX_FMT_YUV420P, c->width, c->height);

static AVFrame *alloc_picture(int pix_fmt, int width, int height){
    AVFrame *picture;
    uint8_t *picture_buf;     // think about why use uint8_t? a byte!
    picture = avcodec_alloc_frame();                          (35)
    if(!picture)
        return NULL;
    size = avpicture_get_size(pix_fmt, width, height);        (36)
    picture_buf = av_malloc(size);                            (37)
    if(!picture_buf){
        av_free(picture);                                     (38)
        return NULL;
    }
    avpicture_fill ( (AVPicture *)picture, picture_buf, pix_fmt, width, height);   (39)
    return picture;
}
```

从以上代码可以看出，完成对一个AVFrame的初始化（其实也就是内存分配），基本上是有这样一个固定模式的。至于**(35)(39)**分别完成了那些工作，以及为什么有这样两步，还没有搞清楚，需要看原代码。我的猜测是**(35)**对AVFrame做了基本的内存分配，保留了对可以提取出AVPicture的前两个数据的内存分配到**(39)**来完成。

说到这里，我们观察到在**(39)**中有一个(AVPicture *)picture，AVPicture这个结构也很有用。基本上他的大小也就是要在网络上传输的包大小，我们在后面可以看到AVPacket跟AVPicture有密切的关系。

VII．AVPicture

AVPicture 在参考代码中没有自己本身的申明和初始化过程。出现了的两次都是作为强制类型转换由 AVFrame 中提取出来的：

```
// open_video() 中
avpicture_fill((AVPicture *)picture, picture_buf, pix_fmt, width, height);          (40)

//write_video_frame 中
// AVPacket pkt;
if(oc->oformat->flags & AVFMT_RAWPICTURE){
    ……
    pkt.size = sizeof(AVPicture);                                                   (41)
}
```

在**(40)**中，实际上是对AVFrame的data[4]、linesize[4]分配内存。由于这两个数据大小如何分配确实需要有pix_fmt、width、height来确定。如果输出文件格式就是RAW 图片（如YUV和RGB），AVPacket作为将编码后数据写入文件的基本数据单元，他的单元大小、数据都是由AVPacket来的。

总结起来就是，AVPicture 的存在有以下原因，AVPicture 将 Picture 的概念从 Frame 中提取出来，就只由 Picture（图片）本身的信息，亮度、色度和行大小。而 Frame 还有如是否是 key frame 之类的信息。这样的类似“分级”是整个概念更加清晰。

VIII．AVPacket

AVPacket 的存在是作为写入文件的基本单元而存在的。我们可能会认为直接把编码后的比特流写入文件不就可以了，为什么还要麻烦设置一个 AVPacket 结构。在我看来这样的编码设置是十分有必要的，特别是在做视频实时传输，同步、边界问题可以通过 AVPacket 来解决。AVPacket 的成员数据有两个时间戳、数据 data（通常是编码后数据）、大小 size 等等（参见 avformat.h 48 行）。讲 AVPacket 的用法就不得不提到编解码函数，因为 AVPacket 的好些信息只有在编解码后才能的知。在参考代码中（ouput_example.c 从 362 到 394 行），做的一个判断分支。如果输出文件格式是 RAW 图像（即 YUV 或 RGB）那么就没有编码函数，直接写入文件（因为程序本身生成一个 YUV 文件），这里的代码虽然在此看来没什么价值，但是如果是解码函数解出 yuv 文件（或 rgb）那么基本的写文件操作就是这样的：

```
if(oc->oformat->flags & AVFMT_RAWPICTURE) {
    AVPacket pkt;                                   // 这里没有用指针！
    av_init_packet(&pkt);

    pkt.flags |= PKT_FLAG_KEY                       // raw picture 中，每帧都是 key frame?
    pkt.stream_index = st->index;
    pkt.data = (uint8_t *)picture;
    pkt.size = sizeof(AVPicture);

    ret = av_write_frame(oc, &pkt);
}
```

输出非 raw picture，编码后：

```
else{
    // video_outbuf & video_outbuf_size 在 open_video() 中初始化
    out_size = avcodec_encode_video(c, video_outbuf, video_outbuf_size, picture);   (42)

    if(out_size > 0){
        AVPacket pkt;
        av_init_packet(&pkt);                                                       (43)

        pkt.pts= av_rescale_q(c->coded_frame->pts, c->time_base, st->time_base);    (44)
        if(c->coded_frame->key_frame)
          pkt.flags |= PKT_FLAG_KEY;
        pkt.stream_index= st->index;
        pkt.data= video_outbuf;
        pkt.size= out_size;
```

```
            /* write the compressed frame in the media file */
            ret = av_write_frame(oc, &pkt);                                    (45)
        } else {
            ret = 0;
        }
    if (ret != 0) {
        fprintf(stderr, "Error while writing video frame\n");
        exit(1);
    }
```

其中 video_outbuf 和 video_outbuf_size 在 open_video()里的初始化是这样的：

```
    video_outbuf = NULL;
    // 输出不是 raw picture，而确实用到编码 codec
    if( !(oc->oformat->flags & AVFMT_RAWPICTURE)){
        video_outbuf_size = 200000;
        video_outbuf = av_malloc(video_outbuf_size);
    }
```

(43)是AVPacket结构的初始化函数。**(44)**比较难理解，而且为什么会有这样的一些时间戳我也没有搞明白。其他的AVPacket成员数据的赋值比较容易理解，要注意的是video_outbuf和video_outbuf_size的初始化问题，由于在参考代码中初始化和使用不在同一函数中，所以比较容易忽视。**(45)**是写文件函数，AVFormatContext* oc中含有文件名等信息，返回值ret因该是一共写了多少数据信息，如果返回 0 则说明写失败。**(42)**和**(45)**作为比较重要的SDK函数，后面还会介绍的。.

IX. Conclusion

以上分析了 FFMpeg 中比较重要的数据结构。下面的这个生成关系理一下思路：（->表示 派生出）

AVFormatContext->AVStream->AVCodecContext->AVCodec
|
AVOutputFormat or AVInputFormat

AVFrame->AVPicture….>AVPacket

二．FFMpeg 中的函数：

在前一部分的分析中我们已经看到 FFMpeg SDK 提供了许多初始化函数和编码函数。我们要做的就是对主要数据结构正确的初始化，以及正确使用相应的编解码函数以及读写（I/O）操作函数。作为一个整体化的代码 SDK，FFMpeg 有一些他自己的标准化使用过程。比如函数 av_register_all(); 就是一个最开始就该调用的“注册函数”，他初始化了 libavcodec，“注册”了所有的的 codec 和视频文件格式（format）。下面，我沿着参考代码（ouput_example.c）的脉络，介绍一下相关函数。

```
/*****************************************************************
      main()
*****************************************************************/
```

1. **av_register_all ()**;

usage:　　initialize ibavcoded, and register all codecs and formats
每个使用 FFMpeg SDK 的工程都必须调用的函数。进行 codec 和 format 的注册，然后才能使用。声明在 allformats.c 中，都是宏有兴趣看看。

2. **AVOutputFormat guess_format(const char *short_name, const char *filename, const char *mime_type)**

usage:　　通过文件后缀名，猜测文件格式，其实也就是要判断使用什么编码器（or 解码器）。

```
AVOutputFormat *fmt;
fmt = guess_format(NULL, filename, NULL);
```

3. **AVFormatContext *av_alloc_format_context(void)**

usage: allocate the output media context.实际是初始化 AVFormatContext 的成员数据 AVClass：

```
AVFormatContext *ic;
ic->av_class = &av_format_context_class;

//where
// format_to_name, options are pointer to function
static const AVClass av_format_context_class = {“AVFormatContext”, format_to_name, options};
```

4. **static AVStream *add_video_stream(AVFormatContext *ox, int codec_id);**

```
AVStream *video_st;
video_st = add_video_stream(oc, fmt->video_codec);
```

5. **int av_set_parameters(AVFormatContext *s, AVFormatParameters *ap)**

usage: set the output parameters (must be done even if no parameters).

```
AVFormatContext *oc;
// if failed, return integer smaller than zero
av_set_parameters(oc, NULL);
```

6. **void dump_format(AVFormatContext *ic, int index, const char *url, int is_output);**

usage: 这一步会用有效的信息把 AVFormatContext 的流域（streams field）填满。作为一个可调试的诊断，我们会将这些信息全盘输出到标准错误输出中，不过你在一个应用程序的产品中并不用这么做：

```
dump_format(oc, 0, filename, 1);  // 也就是指明 AVFormatContext 中的事 AVOutputFormat，还是
                                  // AVInputFormat
```

7. **static void open_video(AVFormatContext *oc, AVStream *st)**

```
open_video(oc, video_st);
```

8. **int av_write_header(AVFormatContext *s)**

usage: allocate the stream private data and writer the stream header to an output media file. param s media file handle, return 0 if OK, AVERROR_xxx if error.
write the stream header, if any

```
av_write_header(oc);
```

9. **static void write_video_frame(AVFormatContext *oc, AVStream *st)**

```
write_video_frame(oc, video_st);
```

10. **static void close_video(AVFormatContext *oc, AVStream *st)**

```
// close each codec
close_video(oc, video_st);
```

11. **int av_write_trailer(AVFormatContext *s)**

usage: write the trailer, if any. Write the stream trailer to an output media file and free the file private data.

```
av_write_trailer(oc);
```

12. **void av_freep(void *arg)**

usage: free the streams. Frees memory and sets the pointer to NULL. arg pointer to the pointer which should be freed .

```
av_freep(&oc->streams[i]->codec);
av_freeep(&oc->streams[s]);
```

13. **int url_fclose(ByteIOContext *s);**

usage: close the output file

url_fclose(&oc->pb);

14. **void av_free(void *ptr)**

usage: free the stream. Free memory which has been allocated with av_malloc(z)() or av_realloc().

av_free(oc);

```
/*****************************************************************
******************************************************************
    add_video_stream()

        AVCodecContext *c
        AVStream       *st
******************************************************************/
******************************************************************
```

1.**AVStream *av_new_stream(AVFormatContext *s, int id)**

usage: add a new stream to a media file. s: media file handle, id: file format dependent stream id

st = av_new_stream(oc, 0);

```
/*****************************************************************
******************************************************************
    open_video()

        AVCodecContext *c
        AVCodec *codec
        AVFrame *picture, *tmp_picture
        uint8_t *video_output
        int frame_count, video_outbuf_size;
******************************************************************
******************************************************************/
```

1 **AVCodec *avcodec_find_encoder(enum CodecID id)**

usage: find the codec of encoder by CodecID. 在前面 main 中的 guess_format()就已经开始为此准备了。

codec = avcodec_find_encoder(c->codec_id);

2 **int avcodec_open(AVCodecContext *avctx, AVCodec *codec);**

usage: opens / inits the AVCodecContext. 打开失败的话，返回值小于零。

avcodec_open(c, codec);

3 **void *av_malloc(unsigned in size);**

usage: you can redefine av_malloc and av_free in your project to use your memory allocator. You do not need to suppress this file because the linker will do it automatically

Memory allocation of size byte with alignment suitable for all memory accesses (including vectors if available on the CPU). av_malloc(0) must return a non NULL pointer.

video_outbuf_size = 200000;
video_outbuf = avmalloc(video_outbuf_size);

4 **static AVFrame *alloc_picture(int pix_fmt, int width, int height)**

picture = alloc_picture(c->pix_fmt, c->width, c->height);

```
/*************************************************************************
*************************************************************************
*************************************************************************
    alloc_picture()

        AVFrame *picture
        uint8_t *picture_buf
        int size
*************************************************************************
*************************************************************************/
*************************************************************************
```

1. avcodec_alloc_frame()

usage: initialize AVFrame* picture

```
picture = avcodec_alloc_frame()
```

2. int avpicture_get_size(int pix_fmt, int width, int height)

usage: 根据像素格式和视频分辨率获得 picture 存储大小。

```
size = avpicture_get_size(pix_fmt, width, height);
picture_buf = av_malloc(size)
```

3. int avpicture_fill(AVPicture *picture, uint8_t *ptr, int pix_fmt, int width, int height)

usage: Picture field are filled with 'ptr' addresses, also return size。用 ptr 中的内容根据文件格式（YUV…）和分辨率填充 picture。这里由于是在初始化阶段，所以填充的可能全是零。

```
avpicture_fill((AVPicture*)picture, picture_buf, pix_fmt, width, height);
```

```
/*************************************************************************
*************************************************************************
    write_video_frame()

        int out_size, ret;
        AVCodecContext *c;
        static struct SwsContext *img_convert_ctx
*************************************************************************
*************************************************************************/
```

1 **struct SwsContext *sws_getContext(int srcW, ……)**

usage: 转变 raw picture 格式的获取 context 函数，比如下面的代码就是将其他格式的（如 yuv422）转为 yuv420，就要将 context 保存在 img_convert_ctx 中，然后再利用后面的介绍函数做转化。

```
img_convert_ctx = sws_getContext(c->width, c->height,
                                 PIX_FMT_YUV420P,
                                 c->width, c->height,
                                 c->pix_fmt,
                                 sws_flags, NULL, NULL, NULL);
```

2 **int sws_scale(struct SwsContext *ctx, uing8_t* src[], int srcStride[], int srcSliceY, int srcSliceH, uint8_t* dst[], int dstStride[]);**

usage: 根据 SwsContext 保存的目标文件 context 将 src（source）转为 dst(destination)。

```
sws_scale(img_convert_ctx, tmp_picture->data, tmp_picture->linesize, 0, c->height, picture->data,
picture->linesize);
```

4. int avcodec_encode_video(AVCodecContext *avctx, uint8_t *buf, int buf_size, const AVFrame *pict);

usage: 根据 AVCodecContext 将 pict 编码到 buf 中，buf_size 是 buf 的大小。

out_size = avcodec_encode_video(c, video_outbuf, video_outbuf_size, picture);

5 **static inline void av_init_packet(AVPacket *pkt)**

usage: initialize optional fields of a packet.初始化 AVPacket。

AVPacket pkt;
av_init_packet(&pkt)

6 int64_t av_rescale_q(int64_t a, AVRational bq, AVRational cq)

usage: 校准时间基（maybe）

pkt.pts = av_rescale_q(c->coded_frame->pts, c->time_base, st->time_base);

7 int av_write_frame(AVFormatContext *s, AVPacket *pkt)

usage: write a packet to an output media file . pkt: the packet, which contains the stream_index, buf/buf_size, dts/pts, …if error return<0, if OK =0, if end of stream wanted =1

ret = av_write_frame(oc, &pke);

```
/**************************************************************************
**************************************************************************

static void close_video(AVFormatContext *oc, AVStream *st)
{
    avcodec_close(st->codec);
    av_free(picture->data[0]);
    av_free(picture);
    if (tmp_picture) {
        av_free(tmp_picture->data[0]);
        av_free(tmp_picture);
    }
    av_free(video_outbuf);
}

/**************************************************************************
**************************************************************************
```

讲初始化过的，特别是分配过内存的，都要释放。

注：标定红色的函数为需要编程人员自己按照实际应用情况编写的，蓝色的是 FFMpeg SDK 中定义的函数。我们会分析函数的作用和调用方法。

由于时间关系，写得很乱。希望没有给你带来太大困扰，这份不成熟的文档能对你有一点点地帮助。chinavideo版上有很多大牛，我想做抛砖引玉者，希望以后能有更好的文档出现。我也是ffmpeg的初学者，如果大家有什么问题也希望能在版上一起讨论。我的email：yunfhu@gmail.com

2007-7-13