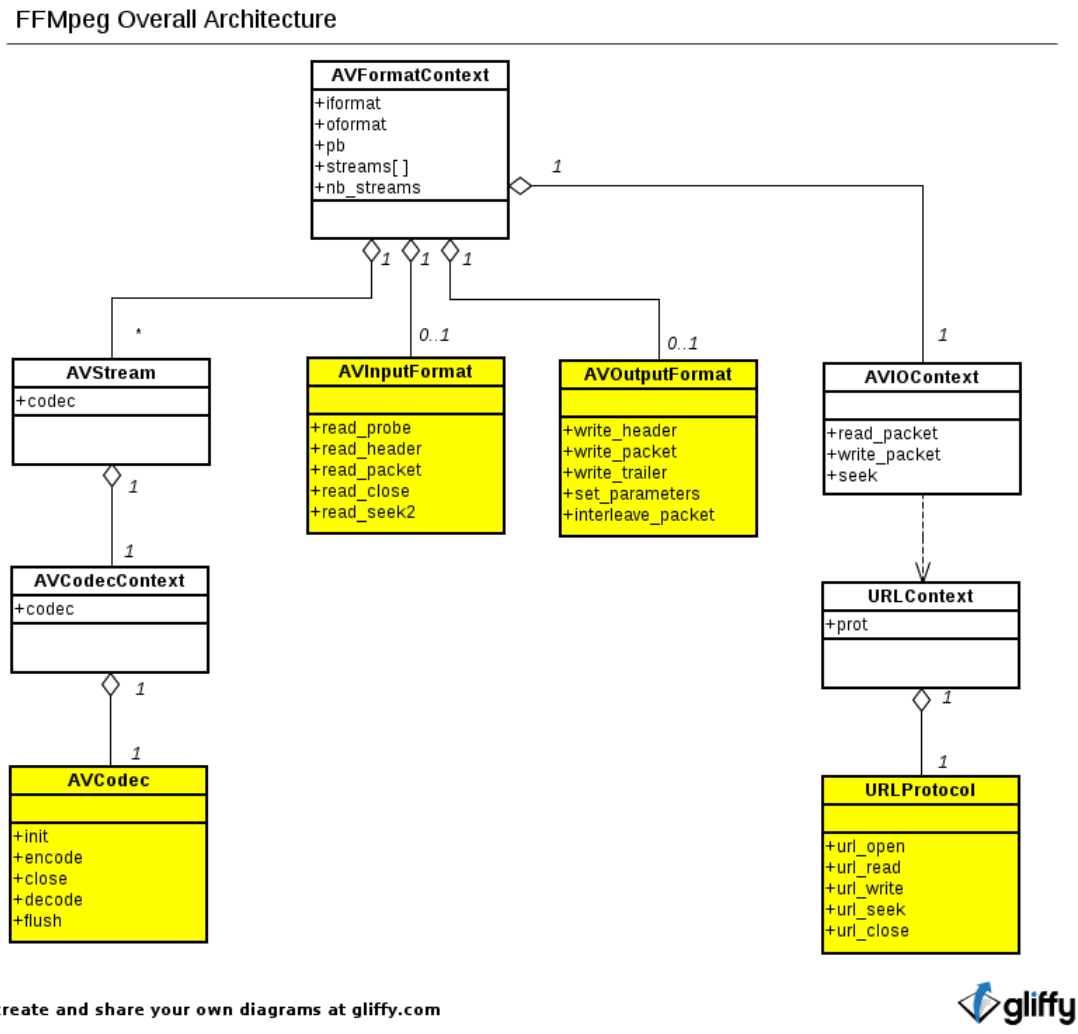
LIBAV

解码全解析

FFMPEG 解码流程：

1. 注册所有容器格式和 CODEC:　av_register_all()
2. 打开文件:　av_open_input_file()
3. 从文件中提取流信息:　av_find_stream_info()
4. 穷举所有的流，查找其中种类为 CODEC_TYPE_VIDEO
5. 查找对应的解码器:　avcodec_find_decoder()
6. 打开编解码器:　avcodec_open()
7. 为解码帧分配内存:　avcodec_alloc_frame()
8. 不停地从码流中提取出帧数据:　av_read_frame()
9. 判断帧的类型，对于视频帧调用: avcodec_decode_video()
10. 解码完后，释放解码器:　avcodec_close()
11. 关闭输入文件:　avformat_close_input_file()

主要数据结构：



基本概念:

编解码器、数据帧、媒体流和容器是数字媒体处理系统的四个基本概念。

首先需要统一术语：

容器／文件（Conainer/File）：即特定格式的多媒体文件。

媒体流（Stream）：指时间轴上的一段连续数据，如一段声音数据，一段视频数据或一段字幕数据，可以是压缩的，也可以是非压缩的，压缩的数据需要关联特定的编解码器。

数据帧／数据包（Frame/Packet）：通常，一个媒体流由大量的数据帧组成，对于压缩数据，帧对应着编解码器的最小处理单元。通常，分属于不同媒体流的数据帧交错复用于容器之中，参见交错。

编解码器：编解码器以帧为单位实现压缩数据和原始数据之间的相互转换。

在 FFMPEG 中，使用 AVFormatContext、AVStream、AVCodecContext、AVCodec 及 AVPacket 等结构来抽象这些基本要素，它们的关系如上图所示：

AVCodecContext：

这是一个描述编解码器上下文的数据结构，包含了众多编解码器需要的参数信息，如下列出了部分比较重要的域：

```
typedef struct AVCodecContext {
 / **
      *一些编解码器需要/可以像使用 extradata Huffman 表。
      * MJPEG：Huffman 表
      * RV10 其他标志
      * MPEG4：全球头（也可以是在比特流或这里）
      *分配的内存应该是 FF_INPUT_BUFFER_PADDING_SIZE 字节较大
      *，比 extradata_size 避免比特流器，如果它与读 prolems。
      * extradata 按字节的内容必须不依赖于架构或 CPU 的字节顺序。
      * - 编码：设置/分配/释放由 libavcodec 的。
      * - 解码：由用户设置/分配/释放。
      * /
     uint8_t *extradata;
     int extradata_size;
    / **
      *这是时间的基本单位，在条件（以秒为单位）
      *帧时间戳派代表出席了会议。对于固定 fps 的内容，
      *基应该 1/framerate 和时间戳的增量应该
      *相同的 1。
      * - 编码：必须由用户设置。
      * - 解码：libavcodec 的设置。
      * /
     AVRational time_base;
 /*视频* /
     / **
      *图片宽度/高度。
      * - 编码：必须由用户设置。
      * - 解码：libavcodec 的设置。
      *请注意：兼容性，它是可能的，而不是设置此
      * coded_width/高解码之前。
```

```
     * /
    int width, height;
    .....
    / *仅音频* /
    int sample_rate; ///<  每秒采样
    int channels;       ///<  音频通道数

    / **
     *音频采样格式
     * - 编码：由用户设置。
     * - 解码：libavcodec 的设置。
     * /
    enum SampleFormat sample_fmt;   ///<  样本格式

    / *下面的数据不应该被初始化。* /
    / **
     *每包样品，初始化时调用“init”。
     * /
    int frame_size;
    int frame_number;     ///<音频或视频帧数量
    char codec_name[32];
    enum AVMediaType codec_type; /*  看到 AVMEDIA_TYPE_xxx */
    enum CodecID codec_id; /* see CODEC_ID_xxx */
/ **
     *的 fourcc（LSB 在前，所以“的 ABCD” - >（“D”<< 24）（“C”<< 16）（“B”<< 8）+“A”）。
     *这是用来解决一些编码错误。
     *分路器应设置什么是编解码器用于识别领域中。
     *如果有分路器等多个领域，在一个容器，然后选择一个
     *最大化使用的编解码器有关的信息。
     *如果在容器中的编解码器标记字段然后 32 位大分路器应该
     *重新映射到一个表或其他结构的 32 位编号。也可选择新
     * extra_codec_tag+大小可以添加，但必须证明这是一个明显的优势
     *第一。
     * - 编码：由用户设置，如果没有则默认基础上 codec_id 将使用。
     * - 解码：由用户设置，将被转换成在初始化 libavcodec 的大写。
     * /
    unsigned int codec_tag;
    .....
    / **
     *在解码器的帧重排序缓冲区的大小。
     *对于 MPEG-2，这是 IPB1 或 0 低延时 IP。
     * - 编码：libavcodec 的设置。
     * - 解码：libavcodec 的设置。
     * /
```

```
    int has_b_frames;

   / **
     *每包的字节数，如果常量和已知或 0
     *用于一些 WAV 的音频编解码器。
     * /
    int block_align;
    / **
     *从分路器位每个样品/像素（huffyuv 需要）。
     * - 编码：libavcodec 的设置。
     * - 解码：由用户设置。
     * /
     int bits_per_coded_sample;
     .....
} AVCodecContext;
```

如果是单纯使用 libavcodec，这部分信息需要调用者进行初始化；如果是使用整个 FFMPEG 库，这部分信息在调用 avformat_open_input 和 avformat_find_stream_info 的过程中根据文件的头信息及媒体流内的头部信息完成初始化。其中几个主要域的释义如下：

extradata/extradata_size：这个 buffer 中存放了解码器可能会用到的额外信息，在 av_read_frame 中填充。一般来说，首先，某种具体格式的 demuxer 在读取格式头信息的时候会填充 extradata，其次，如果 demuxer 没有做这个事情，比如可能在头部压根儿就没有相关的编解码信息，则相应的 parser 会继续从已经解复用出来的媒体流中继续寻找。在没有找到任何额外信息的情况下，这个 buffer 指针为空。

time_base：

width/height：视频的宽和高。

sample_rate/channels：音频的采样率和信道数目。

sample_fmt： 音频的原始采样格式。

codec_name/codec_type/codec_id/codec_tag：编解码器的信息。

AVStrea

该结构体描述一个媒体流，定义如下：

```
typedef struct AVStream {
    int index;      /** <在 AVFormatContext 流的索引* /
    int id;         /**<  特定格式的流 ID */
    AVCodecContext *codec; /**< codec context */
    / **
     *流的实时帧率基地。
     *这是所有时间戳可以最低帧率
     *准确代表（它是所有的最小公倍数
     *流的帧率）。请注意，这个值只是一个猜测！
     *例如，如果时间基数为 1/90000 和所有帧
     *约 3600 或 1800 计时器刻度，，然后 r_frame_rate 将是 50/1。
     * /
```

```
    AVRational r_frame_rate;
   / **
     *这是时间的基本单位，在条件（以秒为单位）
     *帧时间戳派代表出席了会议。对于固定 fps 的内容，
     *时基应该是 1/framerate 的时间戳的增量应为 1。
     * /
    AVRational time_base;
    .....
    / **
     *解码流量的第一帧，在流量时-base 分。
     *如果你是绝对 100％的把握，设定值
     *它真的是第一帧点。
     *这可能是未定义（AV_NOPTS_VALUE）的。
     *@注意的业余头不弱者受制与正确的 START_TIME 的业余
     *分路器必须不设定此。
     * /
    int64_t start_time;
    / **
     *解码：时间流流时基。
     *如果源文件中没有指定的时间，但不指定
     *比特率，这个值将被从码率和文件大小的估计。
     * /
    int64_t duration;
#if LIBAVFORMAT_VERSION_INT < (53<<16)
    char language[4]; /** ISO 639-2/B 3-letter language code (empty string if undefined) */
#endif
  /* av_read_frame（）支持* /
    enum AVStreamParseType need_parsing;
    struct AVCodecParserContext *parser;
    .....
   /*函数 av_seek_frame（）支持* /
    AVIndexEntry *index_entries; / **<仅用于如果格式不 notsupport 寻求本身。* /
    int nb_index_entries;
    unsigned int index_entries_allocated_size;
    int64_t nb_frames;                  ///<  在此流的帧，如果已知或 0
    ......
    //*平均帧率

    AVRational avg_frame_rate;
    ......
} AVStream;
```

主要域的释义如下，其中大部分域的值可以由 avformat_open_input 根据文件头的信息确定，缺少的信息需要通过调用 avformat_find_stream_info 读帧及软解码进一步获取：

index/id：index 对应流的索引，这个数字是自动生成的，根据 index 可以从 AVFormatContext::streams 表中索引到该流；而 id 则是流的标识，依赖于具体的容器格式。比如对于 MPEG TS 格式，id 就是 pid。

time_base：流的时间基准，是一个实数，该流中媒体数据的 pts 和 dts 都将以这个时间基准为粒度。通常，使用 av_rescale/av_rescale_q 可以实现不同时间基准的转换。

start_time：流的起始时间，以流的时间基准为单位，通常是该流中第一个帧的 pts。

duration：流的总时间，以流的时间基准为单位。

need_parsing：对该流 parsing 过程的控制域。

nb_frames：流内的帧数目。

r_frame_rate/framerate/avg_frame_rate：帧率相关。

codec：指向该流对应的 AVCodecContext 结构，调用 avformat_open_input 时生成。

parser：指向该流对应的 AVCodecParserContext 结构，调用 avformat_find_stream_info 时生成。。

AVFormatContext

这个结构体描述了一个媒体文件或媒体流的构成和基本信息，定义如下：

```
typedef struct AVFormatContext {
    const AVClass *av_class; /**<由 avformat_alloc_context 设置的。* /
    / *只能是 iFormat 的，或在同一时间 oformat，不是两个。* /
    struct AVInputFormat *iformat;
    struct AVOutputFormat *oformat;
    void *priv_data;
    ByteIOContext *pb;
    unsigned int nb_streams;
    AVStream *streams[MAX_STREAMS];
    char filename[1024]; / **<输入或输出的       文件名*/
    / *流信息* /
    int64_t timestamp;
#if LIBAVFORMAT_VERSION_INT < (53<<16)
    char title[512];
    char author[512];
    char copyright[512];
    char comment[512];
    char album[512];
    int year;   /**< ID3 year, 0 if none */
    int track; /**< track number, 0 if none */
    char genre[32]; /**< ID3 genre */
#endif
    int ctx_flags; /** <格式特定的标志，看到 AVFMTCTX_xx* /
    /*分处理的私人数据（不直接修改）。* /
    / **此缓冲区只需要当数据包已经被缓冲，但
        不解码，例如，在 MPEG 编解码器的参数
        流。  * /
```

```
        struct AVPacketList *packet_buffer;
        / **解码元件的第一帧的位置，在
            AV_TIME_BASE 分数秒。从来没有设置这个值直接：
            推导的 AVStream 值。  * /
        int64_t start_time;
        / **解码流的时间，在 AV_TIME_BASE 分数
            秒。只设置这个值，如果你知道没有个人流
            工期，也不要设置任何他们。这是从推导
            AVStream 值如果没有设置。
        int64_t duration;
        / **解码：总的文件大小，如果未知 0* /
        int64_t file_size;
        / **解码：在比特/秒的总流率，如果不
            可用。从来没有直接设置它如果得到 file_size 和
            时间是已知的如 FFmpeg 的自动计算。  * /
        int bit_rate;
        /* av_read_frame（）支持* /
        AVStream *cur_st;
    #if LIBAVFORMAT_VERSION_INT < (53<<16)
        const uint8_t *cur_ptr_deprecated;
        int cur_len_deprecated;
        AVPacket cur_pkt_deprecated;
    #endif
        /* av_seek_frame()  支持  */
        int64_t data_offset; /**  第一包抵消  */
        int index_built;
        int mux_rate;
        unsigned int packet_size;
        int preload;
        int max_delay;
    #define AVFMT_NOOUTPUTLOOP -1
    #define AVFMT_INFINITEOUTPUTLOOP 0
        /**  次循环输出的格式支持它的数量  */
        int loop_output;
        int flags;
    #define AVFMT_FLAG_GENPTS    0x0001 ///<  生成失踪分，即使它需要解析未来框架。
    #define AVFMT_FLAG_IGNIDX          0x0002 ///<  忽略指数。
    #define AVFMT_FLAG_NONBLOCK      0x0004 ///<从输入中读取数据包时，不要阻止。
    #define AVFMT_FLAG_IGNDTS          0x0008 ///<  忽略帧的 DTS 包含 DTS 与 PTS
    #define AVFMT_FLAG_NOFILLIN    0x0010 ///<  不要从任何其他值推断值，只是返回存储在容器中
    #define AVFMT_FLAG_NOPARSE         0x0020 ///<  不要使用 AVParsers，你还必须设置为 FILLIN 帧代码的工作，没有解析 AVFMT_FLAG_NOFILLIN - >无帧。也在寻求框架不能工作，如果找到帧边界的解析已被禁用
```

```
#define AVFMT_FLAG_RTP_HINT         0x0040 ///< 暗示到输出文件添加的 RTP
    int loop_input;
   /**解码：对探测数据的大小;编码：未使用。* /
    unsigned int probesize;
    / **
     在此期间，输入*最大时间（在 AV_TIME_BASE 单位）应
     *进行分析在 avformat_find_stream_info（）。
     * /
    int max_analyze_duration;
    const uint8_t *key;
    int keylen;
    unsigned int nb_programs;
    AVProgram **programs;
   / **
     *强迫影片 codec_id。
     * Demuxing：由用户设置。
     * /
    enum CodecID video_codec_id;
    / **
     *强迫音频 codec_id。
     * Demuxing：由用户设置。
     * /
    enum CodecID audio_codec_id;
    / **
     *强制的：字幕 codec_id。
     * Demuxing：由用户设置。
     * /
    enum CodecID subtitle_codec_id;
    / **
     *以字节为单位的最高限额为每个数据流的索引使用的内存。
     *如果该指数超过此大小，条目将被丢弃
     *需要保持一个较小的规模。这可能会导致较慢或更少
     *准确的寻求（分路器）。
     *分路器内存中的一个完整的指数是强制性的将忽略
     *此。
     *混流：未使用
     * demuxing：由用户设置* /
    unsigned int max_index_size;
   / **
     *以字节为单位的最高限额使用帧缓冲内存
     *从实时捕获设备获得。* /
    unsigned int max_picture_buffer;
    unsigned int nb_chapters;
    AVChapter **chapters;
```

```
        / **
         *标志启用调试。* /
        int debug;
#define FF_FDEBUG_TS            0x0001
        / **
         *原始数据包从分路器之前，解析和解码。
         *此缓冲区用于缓冲数据包，直到编解码器可以
         *确定，因为不知道不能做解析
         *编解码器。* /
        struct AVPacketList *raw_packet_buffer;
        struct AVPacketList *raw_packet_buffer_end;
        struct AVPacketList *packet_buffer_end;
        AVMetadata *metadata;
        / **
         *剩余的大小可为 raw_packet_buffer，以字节为单位。
         *不属于公共 API* /
#define RAW_PACKET_BUFFER_SIZE 2500000
        int raw_packet_buffer_remaining_size;
        / **
         *在现实世界中的时间流的开始时间，以微秒
         *自 Unix 纪元（1970 年 1 月 1 日起 00:00）。也就是说，pts= 0
         在这个现实世界的时间*流被抓获。
         * - 编码：由用户设置。
         * - 解码：未使用。 * /
        int64_t start_time_realtime;
} AVFormatContext;
```

这是 FFMpeg 中最为基本的一个结构，是其他所有结构的根，是一个多媒体文件或流的根本抽象。其中:

nb_streams 和 streams 所表示的 AVStream 结构指针数组包含了所有内嵌媒体流的描述；

iformat 和 oformat 指向对应的 demuxer 和 muxer 指针；

pb 则指向一个控制底层数据读写的 ByteIOContext 结构。

start_time 和 duration 是从 streams 数组的各个 AVStream 中推断出的多媒体文件的起始时间和长度，以微妙为单位。

通常，这个结构由 avformat_open_input 在内部创建并以缺省值初始化部分成员。但是，如果调用者希望自己创建该结构，则需要显式为该结构的一些成员置缺省值——如果没有缺省值的话，会导致之后的动作产生异常。以下成员需要被关注：

probesize

mux_rate

packet_size

flags

max_analyze_duration

key

max_index_size

max_picture_buffer

max_delay

AVPacket

AVPacket 定义在 avcodec.h 中，如下：

```
typedef struct AVPacket {
    / **
     AVStream->基 time_base 单位介绍时间戳的时间
     *解压缩包将被提交给用户。
     *可 AV_NOPTS_VALUE 如果没有存储在文件中。
     *分必须大于或等于 DTS 作为演示不能发生之前
     *减压，除非要查看十六进制转储。有些格式滥用
     * DTS 和 PTS/ CTS 的条款意味着不同的东西。如时间戳
     *必须转换为真正的 PTS / DTS 之前，他们在 AVPacket 存储。 * /
    int64_t pts;
    / **
     AVStream->基 time_base 单位时间的减压时间戳记;
     *包解压。
     *可 AV_NOPTS_VALUE 如果没有存储在文件中。 * /
    int64_t dts;
    uint8_t *data;
    int     size;
    int     stream_index;
    int     flags;
    / **
     *这个包的时间 AVStream->基 time_base 单位，如果未知。
     *等于 next_pts -  在呈现顺序 this_pts。* /
    int     duration;
    void    (*destruct)(struct AVPacket *);
    void    *priv;
    int64_t pos;                                    ///<  如果未知字节的位置，在流，-1
    / **
     * AVStream->基 time_base 单位的时差，这点
     *包从解码器输出的已融合在哪个点
     *独立的前一帧的情况下。也就是说，
     *框架几乎是一致的，没有问题，如果解码开始从
     *第一帧或从这个关键帧。
     * AV_NOPTS_VALUE 如果不明。
     *此字段是不是当前数据包的显示时间。
     *
     *这一领域的目的是允许在流，没有寻求
     *在传统意义上的关键帧。它所对应的
     *恢复点 SEI 的 H.264 和 match_time_delta 在螺母。这也是
```

```
        *必不可少的一些类型的字幕流，以确保所有
        *后寻求正确显示字幕。* /
        int64_t convergence_duration;
    } AVPacket;
```

FFMPEG 使用 AVPacket 来暂存解复用之后、解码之前的媒体数据（一个音/视频帧、一个字幕包等）及附加信息（解码时间戳、显示时间戳、时长等）。其中：

dts 表示解码时间戳，pts 表示显示时间戳，它们的单位是所属媒体流的时间基准。

stream_index 给出所属媒体流的索引；

data 为数据缓冲区指针，size 为长度；

duration 为数据的时长，也是以所属媒体流的时间基准为单位；

pos 表示该数据在媒体流中的字节偏移量；

destruct 为用于释放数据缓冲区的函数指针；

flags 为标志域，其中，最低为置 1 表示该数据是一个关键帧。

AVPacket 结构本身只是个容器，它使用 data 成员引用实际的数据缓冲区。这个缓冲区通常是由 av_new_packet 创建的，但也可能由 FFMPEG 的 API 创建（如 av_read_frame）。当某个 AVPacket 结构的数据缓冲区不再被使用时，要需要通过调用 av_free_packet 释放。av_free_packet 调用的是结构体本身的 destruct 函数，它的值有两种情况：1)av_destruct_packet_nofree 或 0；2)av_destruct_packet，其中，情况 1)仅仅是将 data 和 size 的值清 0 而已，情况 2)才会真正地释放缓冲区。

FFMPEG 内部使用 AVPacket 结构建立缓冲区装载数据，同时提供 destruct 函数，如果 FFMPEG 打算自己维护缓冲区，则将 destruct 设为 av_destruct_packet_nofree，用户调用 av_free_packet 清理缓冲区时并不能够将其释放；如果 FFMPEG 打算将该缓冲区彻底交给调用者，则将 destruct 设为 av_destruct_packet，表示它能够被释放。安全起见，如果用户希望自由地使用一个 FFMPEG 内部创建的 AVPacket 结构，最好调用 av_dup_packet 进行缓冲区的克隆，将其转化为缓冲区能够被释放的 AVPacket，以免对缓冲区的不当占用造成异常错误。av_dup_packet 会为 destruct 指针为 av_destruct_packet_nofree 的 AVPacket 新建一个缓冲区，然后将原缓冲区的数据拷贝至新缓冲区，置 data 的值为新缓冲区的地址，同时设 destruct 指针为 av_destruct_packet。

时间信息

时间信息用于实现多媒体同步。

同步的目的在于展示多媒体信息时，能够保持媒体对象之间固有的时间关系。同步有两类，一类是流内同步，其主要任务是保证单个媒体流内的时间关系，以满足感知要求，如按照规定的帧率播放一段视频；另一类是流间同步，主要任务是保证不同媒体流之间的时间关系，如音频和视频之间的关系（lipsync）。

对于固定速率的媒体，如固定帧率的视频或固定比特率的音频，可以将时间信息（帧率或比特率）置于文件首部（header），如 AVI 的 hdrl List、MP4 的 moov box，还有一种相对复杂的方案是将时间信息嵌入媒体流的内部，如 MPEG TS 和 Real video，这种方案可以处理变速率的媒体，亦可有效避免同步过程中的时间漂移。

FFMPEG 会为每一个数据包打上时间标签，以更有效地支持上层应用的同步机制。时间标签有两种，一种是 DTS，称为解码时间标签，另一种是 PTS，称为显示时间标签。对于声音来说 ，这两个时间标签是相同的，但对于某些视频编码格式，由于采用了双向预测技术，会造成 DTS 和 PTS 的不一致。

无双向预测帧的情况：

图像类型: I P P P P P P ... I P P

DTS: 0 1 2 3 4 5 6... 100 101 102

PTS: 0 1 2 3 4 5 6... 100 101 102

有双向预测帧的情况：

图像类型: I P B B P B B ... I P B

DTS: 0 1 2 3 4 5 6 ... 100 101 102

PTS: 0 3 1 2 6 4 5 ... 100 104 102

对于存在双向预测帧的情况，通常要求解码器对图像重排序，以保证输出的图像顺序为显示顺序：

解码器输入：I P B B P B B

(DTS) 0 1 2 3 4 5 6

(PTS) 0 3 1 2 6 4 5

解码器输出：X I B B P B B P

(PTS) X 0 1 2 3 4 5 6

时间信息的获取：

通过调用 avformat_find_stream_info，多媒体应用可以从 AVFormatContext 对象中拿到媒体文件的时间信息：主要是总时间长度和开始时间，此外还有与时间信息相关的比特率和文件大小。其中时间信息的单位是 AV_TIME_BASE：微秒。

```
typedef struct AVFormatContext {
    / **解码元件的第一帧的位置，在
        AV_TIME_BASE 分数秒。从来没有设置这个值直接：
        推导的 AVStream 值。 * /
    int64_t start_time;
    / **解码流的时间，在 AV_TIME_BASE 分数秒。只设置这个值，如果你知道没有个人流工期，也不要设置任何他们。这是从推导 AVStream 值如果没有设置。 * /
    int64_t duration;
    / **解码：总的文件大小，如果未知=0* /
    int64_t file_size;
    / **解码：在比特/秒的总流率，如果不可用。从来没有直接设置它如果得到 file_size 和时间是已知的如 FFmpeg 的自动计算。 * /
    int bit_rate;
    .....
} AVFormatContext;
```

以上 4 个成员变量都是只读的，基于 FFMpeg 的中间件需要将其封装到某个接口中，如：

```
LONG GetDuratioin(IntfX*);
LONG GetStartTime(IntfX*);
LONG GetFileSize(IntfX*);
LONG GetBitRate(IntfX*);
```

APIs

avformat_open_input:

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

avformat_open_input 完成两个任务：

打开一个文件或 URL，基于字节流的底层输入模块得到初始化。

解析多媒体文件或多媒体流的头信息，创建 AVFormatContext 结构并填充其中的关键字段，依次为各个原始流建立 AVStream 结构。

一个多媒体文件或多媒体流与其包含的原始流的关系如下：

多媒体文件/多媒体流 (movie.mkv)

原始流 1 (h.264 video)

原始流 2 (aac audio for Chinese)

原始流 3 (aac audio for english)

原始流 4 (Chinese Subtitle)

原始流 5 (English Subtitle)

...

关于输入参数：

ic_ptr，这是一个指向指针的指针，用于返回 avformat_open_input 内部构造的一个 AVFormatContext 结构体。

filename，指定文件名。

fmt，用于显式指定输入文件的格式，如果设为空则自动判断其输入格式。

options

这个函数通过解析多媒体文件或流的头信息及其他辅助数据，能够获取足够多的关于文件、流和编解码器的信息，但由于任何一种多媒体格式提供的信息都是有限的，而且不同的多媒体内容制作软件对头信息的设置不尽相同，此外这些软件在产生多媒体内容时难免会引入一些错误，因此这个函数并不保证能够获取所有需要的信息，在这种情况下，则需要考虑另一个函数：

avformat_find_stream_info:

int avformat_find_stream_info(AVFormatContext *ic, AVDictionary **options);

这个函数主要用于获取必要的编解码器参数，设置到 ic→streams[i]→codec 中。

首先必须得到各媒体流对应编解码器的类型和 id，这是两个定义在 avutils.h 和 avcodec.h 中的枚举：

```
enum AVMediaType {
    AVMEDIA_TYPE_UNKNOWN = -1,
    AVMEDIA_TYPE_VIDEO,
    AVMEDIA_TYPE_AUDIO,
    AVMEDIA_TYPE_DATA,
    AVMEDIA_TYPE_SUBTITLE,
    AVMEDIA_TYPE_ATTACHMENT,
    AVMEDIA_TYPE_NB
};
enum CodecID {
    CODEC_ID_NONE,
    /* video codecs */
    CODEC_ID_MPEG1VIDEO,
    CODEC_ID_MPEG2VIDEO, ///< preferred ID for MPEG-1/2 video decoding
    CODEC_ID_MPEG2VIDEO_XVMC,
    CODEC_ID_H261,
    CODEC_ID_H263,
```

...

};

通常，如果某种媒体格式具备完备而正确的头信息，调用 avformat_open_input 即可以得到这两个参数，但若是因某种原因 avformat_open_input 无法获取它们，这一任务将由 avformat_find_stream_info 完成。

其次还要获取各媒体流对应编解码器的时间基准。

此外，对于音频编解码器，还需要得到：

采样率，

声道数，

位宽，

帧长度（对于某些编解码器是必要的），

对于视频编解码器，则是：

图像大小，

色彩空间及格式，

av_read_frame

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

这个函数用于从多媒体文件或多媒体流中读取媒体数据，获取的数据由 AVPacket 结构 pkt 来存放。对于音频数据，如果是固定比特率，则 pkt 中装载着一个或多个音频帧；如果是可变比特率，则 pkt 中装载有一个音频帧。对于视频数据，pkt 中装载有一个视频帧。需要注意的是：再次调用本函数之前，必须使用 av_free_packet 释放 pkt 所占用的资源。

通过 pkt→stream_index 可以查到获取的媒体数据的类型，从而将数据送交相应的解码器进行后续处理。

av_seek_frame

int av_seek_frame(AVFormatContext *s, int stream_index, int64_t timestamp, int flags);

这个函数通过改变媒体文件的读写指针来实现对媒体文件的随机访问，支持以下三种方式：

基于时间的随机访问：具体而言就是将媒体文件读写指针定位到某个给定的时间点上，则之后调用 av_read_frame 时能够读到时间标签等于给定时间点的媒体数据，通常用于实现媒体播放器的快进、快退等功能。

基于文件偏移的随机访问：相当于普通文件的 seek 函数，timestamp 也成为文件的偏移量。

基于帧号的随机访问：timestamp 为要访问的媒体数据的帧号。

关于参数：

s：是个 AVFormatContext 指针，就是 avformat_open_input 返回的那个结构。

stream_index：指定媒体流，如果是基于时间的随机访问，则第三个参数 timestamp 将以此媒体流的时间基准为单位；如果设为负数，则相当于不指定具体的媒体流，FFMPEG 会按照特定的算法寻找缺省的媒体流，此时，timestamp 的单位为 AV_TIME_BASE（微秒）。

timestamp：时间标签，单位取决于其他参数。

flags：定位方式，AVSEEK_FLAG_BYTE 表示基于字节偏移，AVSEEK_FLAG_FRAME 表示基于帧号，其它表示基于时间。

av_close_input_file:

void av_close_input_file(AVFormatContext *s);

关闭一个媒体文件：释放资源，关闭物理 IO。

avcodec_find_decoder:

AVCodec *avcodec_find_decoder(enum CodecID id);

AVCodec *avcodec_find_decoder_by_name(const char *name);

根据给定的 codec id 或解码器名称从系统中搜寻并返回一个 AVCodec 结构的指针。

avcodec_open:

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

此函数根据输入的 AVCodec 指针具体化 AVCodecContext 结构。在调用该函数之前，需要首先调用 avcodec_alloc_context 分配一个 AVCodecContext 结构，或调用 avformat_open_input 获取媒体文件中对应媒体流的 AVCodecContext 结构；此外还需要通过 avcodec_find_decoder 获取 AVCodec 结构。

这一函数还将初始化对应的解码器。

avcodec_decode_video2

int avcodec_decode_video2(AVCodecContext *avctx, AVFrame *picture, int *got_picture_ptr, AVPacket *avpkt);

解码一个视频帧。got_picture_ptr 指示是否有解码数据输出。

输入数据在 AVPacket 结构中，输出数据在 AVFrame 结构中。AVFrame 是定义在 avcodec.h 中的一个数据结构：

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

FF_COMMON_FRAME 定义了诸多数据域，大部分由 FFMpeg 内部使用，对于用户来说，比较重要的主要包括：

```
#define FF_COMMON_FRAME \
.....
    uint8_t *data[4];\
    int linesize[4];\
    int key_frame;\
    int pict_type;\
    int64_t pts;\
    int reference;\
.....
```

FFMpeg 内部以 planar 的方式存储原始图像数据，即将图像像素分为多个平面（R/G/B 或 Y/U/V），data 数组内的指针分别指向四个像素平面的起始位置，linesize 数组则存放各个存贮各个平面的缓冲区的行宽：

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++data[0]->####################################++++++++++++
++++++++++++###########picture data##########++++++++++++
                    ......................
++++++++++++####################################++++++++++++
|<--------------------line_size[0]---------------------->|
```

此外，key_frame 标识该图像是否是关键帧；pict_type 表示该图像的编码类型：I(1)/P(2)/B(3)……；pts 是以 time_base 为单位的时间标签，对于部分解码器如 H.261、H.263 和 MPEG4，可以从头信息中获取；reference 表示该图像是否被用作参考。

avcodec_decode_audio4

int avcodec_decode_audio4(AVCodecContext *avctx, AVFrame *frame, int *got_frame_ptr, AVPacket *avpkt);

解码一个音频帧。输入数据在 AVPacket 结构中，输出数据在 frame 中，got_frame_ptr 表示是否有数据输出。

avcodec_close

int avcodec_close(AVCodecContext *avctx);

关闭解码器，释放 avcodec_open 中分配的资源。

测试程序

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/time.h>
#include "libavutil/avstring.h"
#include "libavformat/avformat.h"
#include "libavdevice/avdevice.h"
#include "libavcodec/opt.h"
#include "libswscale/swscale.h"
#define DECODED_AUDIO_BUFFER_SIZE                    192000
struct options
{
    int streamId;
    int frames;
    int nodec;
    int bplay;
    int thread_count;
    int64_t lstart;
    char finput[256];
    char foutput1[256];
    char foutput2[256];
};
int parse_options(struct options *opts, int argc, char** argv)
{
    int optidx;
    char *optstr;
    if (argc < 2) return -1;
    opts->streamId = -1;
    opts->lstart = -1;
    opts->frames = -1;
    opts->foutput1[0] = 0;
    opts->foutput2[0] = 0;
    opts->nodec = 0;
    opts->bplay = 0;
```

```
	opts->thread_count = 0;
	strcpy(opts->finput, argv[1]);
	optidx = 2;
	while (optidx < argc)
	{
		optstr = argv[optidx++];
		if (*optstr++ != '-') return -1;
		switch (*optstr++)
		{
		case 's':    //< stream id
			opts->streamId = atoi(optstr);
			break;
		case 'f':    //< frames
			opts->frames = atoi(optstr);
			break;
		case 'k':    //< skipped
			opts->lstart = atoll(optstr);
			break;
		case 'o':    //< output
			strcpy(opts->foutput1, optstr);
			strcat(opts->foutput1, ".mpg");
			strcpy(opts->foutput2, optstr);
			strcat(opts->foutput2, ".raw");
			break;
		case 'n': //decoding and output options
			if (strcmp("dec", optstr) == 0)
				opts->nodec = 1;
			break;
		case 'p':
			opts->bplay = 1;
			break;
		case 't':
			opts->thread_count = atoi(optstr);
			break;
		default:
			return -1;
		}
	}

	return 0;
}
void show_help(char* program)
{
	printf("简单的 FFMPEG 测试方案\n");
```

```
        printf("Usage: %s inputfile [-sstreamid [-fframes] [-kskipped]
[-ooutput_filename(without extension)] [-p] [-tthread_count]]\n",
                program);
        return;
}
static void log_callback(void* ptr, int level, const char* fmt, va_list vl)
{
        vfprintf(stdout, fmt, vl);
}
/ *音频渲染器的代码（OSS）*/
#include <sys/ioctl.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/soundcard.h>
#define OSS_DEVICE "/dev/dsp0"
struct audio_dsp
{
        int audio_fd;
        int channels;
        int format;
        int speed;
};
int map_formats(enum SampleFormat format)
{
        switch(format)
        {
                case SAMPLE_FMT_U8:
                        return AFMT_U8;
                case SAMPLE_FMT_S16:
                        return AFMT_S16_LE;
                default:
                        return AFMT_U8;
        }
}
int set_audio(struct audio_dsp* dsp)
{
        if (dsp->audio_fd == -1)
        {
                printf("无效的音频 DSP ID!\n");
                return -1;
        }
        if (-1 == ioctl(dsp->audio_fd, SNDCTL_DSP_SETFMT, &dsp->format))
        {
                printf("无法设置 DSP 格式!\n");
```

```
            return -1;
        }
        if (-1 == ioctl(dsp->audio_fd, SNDCTL_DSP_CHANNELS, &dsp->channels))
        {
            printf("无法设置 DSP 格式!\n");
            return -1;
        }
        if (-1 == ioctl(dsp->audio_fd, SNDCTL_DSP_SPEED, &dsp->speed))
        {
            printf("无法设置 DSP 格式!\n");
            return -1;
        }
        return 0;
}
int play_pcm(struct audio_dsp* dsp, unsigned char *buf, int size)
{
        if (dsp->audio_fd == -1)
        {
            printf("无效的音频 DSP ID！ \n");
            return -1;
        }
        if (-1 == write(dsp->audio_fd, buf, size))
        {
            printf("音频 DSP 无法写入！ \n");
            return -1;
        }
        return 0;
}
/*  音频渲染代码结束  */
/*  视频渲染代码*/
#include <linux/fb.h>
#include <sys/mman.h>

#define FB_DEVICE "/dev/fb0"

enum pic_format
{
        eYUV_420_Planer,
};
struct video_fb
{
        int video_fd;
        struct fb_var_screeninfo vinfo;
        struct fb_fix_screeninfo finfo;
```

```
        unsigned char *fbp;
        AVFrame *frameRGB;
        struct
        {
                int x;
                int y;
        } video_pos;
};
int open_video(struct video_fb *fb, int x, int y)
{
        int screensize;
        fb->video_fd = open(FB_DEVICE, O_WRONLY);
        if (fb->video_fd == -1) return -1;
        if (ioctl(fb->video_fd, FBIOGET_FSCREENINFO, &fb->finfo)) return -2;
        if (ioctl(fb->video_fd, FBIOGET_VSCREENINFO, &fb->vinfo)) return -2;
        printf("视频设备：分解  %dx%d, %dbpp\n", fb->vinfo.xres, fb->vinfo.yres,
fb->vinfo.bits_per_pixel);
        screensize = fb->vinfo.xres * fb->vinfo.yres * fb->vinfo.bits_per_pixel / 8;
        fb->fbp = (unsigned char *) mmap(0, screensize, PROT_READ|PROT_WRITE,
MAP_SHARED, fb->video_fd, 0);
        if (fb->fbp == -1) return -3;
        if (x >= fb->vinfo.xres || y >= fb->vinfo.yres)
        {
                return -4;
        }
        else
        {
                fb->video_pos.x = x;
                fb->video_pos.y = y;
        }

        fb->frameRGB = avcodec_alloc_frame();
        if (!fb->frameRGB) return -5;
        return 0;
}
#if 0
/* only 420P supported now */
int show_picture(struct video_fb *fb, AVFrame *frame, int width, int height, enum
pic_format format)
{
        struct SwsContext *sws;
        int i;
        unsigned char *dest;
        unsigned char *src;
```

```
	if (fb->video_fd == -1) return -1;
	if ((fb->video_pos.x >= fb->vinfo.xres) || (fb->video_pos.y >= fb->vinfo.yres))
return -2;
	if (fb->video_pos.x + width > fb->vinfo.xres)
	{
		width = fb->vinfo.xres - fb->video_pos.x;
	}
	if (fb->video_pos.y + height > fb->vinfo.yres)
	{
		height = fb->vinfo.yres - fb->video_pos.y;
	}

	if (format == PIX_FMT_YUV420P)
	{
		sws = sws_getContext(width, height, format, width, height,
PIX_FMT_RGB32, SWS_FAST_BILINEAR, NULL, NULL, NULL);
		if (sws == 0)
		{
			return -3;
		}
		if (sws_scale(sws, frame->data, frame->linesize, 0, height,
fb->frameRGB->data, fb->frameRGB->linesize))
		{
			return -3;
		}
		dest = fb->fbp + (fb->video_pos.x+fb->vinfo.xoffset) *
(fb->vinfo.bits_per_pixel/8) +(fb->video_pos.y+fb->vinfo.yoffset) *
fb->finfo.line_length;
		for (i = 0; i < height; i++)
		{
			memcpy(dest, src, width*4);
			src += fb->frameRGB->linesize[0];
			dest += fb->finfo.line_length;
		}
	}
	return 0;
}
#endif
void close_video(struct video_fb *fb)
{
	if (fb->video_fd != -1)
	{
		munmap(fb->fbp, fb->vinfo.xres * fb->vinfo.yres *
fb->vinfo.bits_per_pixel / 8);
```

```
            close(fb->video_fd);
            fb->video_fd = -1;
        }
}
/*  视频渲染代码结束  */

int main(int argc, char **argv)
{
        AVFormatContext* pCtx = 0;
        AVCodecContext *pCodecCtx = 0;
        AVCodec *pCodec = 0;
        AVPacket packet;
        AVFrame *pFrame = 0;
        FILE *fpo1 = NULL;
        FILE *fpo2 = NULL;
        int nframe;
        int err;
        int got_picture;
        int picwidth, picheight, linesize;
        unsigned char *pBuf;
        int i;
        int64_t timestamp;
        struct options opt;
        int usefo = 0;
        struct audio_dsp dsp;
        int dusecs;
        float usecs1 = 0;
        float usecs2 = 0;
        struct timeval elapsed1, elapsed2;
        int decoded = 0;

        av_register_all();

        av_log_set_callback(log_callback);
        av_log_set_level(50);

        if (parse_options(&opt, argc, argv) < 0 || (strlen(opt.finput) == 0))
        {
            show_help(argv[0]);
            return 0;
        }
        err = avformat_open_input(&pCtx, opt.finput, 0, 0);
        if (err < 0)
        {
```

```
            printf("\n->(avformat_open_input)\tERROR:\t%d\n", err);
            goto fail;
        }
        err = avformat_find_stream_info(pCtx, 0);
        if (err < 0)
        {
            printf("\n->(avformat_find_stream_info)\tERROR:\t%d\n", err);
            goto fail;
        }
        if (opt.streamId < 0)
        {
            av_dump_format(pCtx, 0, pCtx->filename, 0);
            goto fail;
        }
        else
        {
            printf("\n  额外的数据流  %d (%dB):", opt.streamId,
pCtx->streams[opt.streamId]->codec->extradata_size);
            for (i = 0; i < pCtx->streams[opt.streamId]->codec->extradata_size; i++)
            {
                if (i%16 == 0) printf("\n");
                printf("%2x    ", pCtx->streams[opt.streamId]->codec->extradata[i]);
            }
        }
        / *尝试打开输出文件*/
        if (strlen(opt.foutput1) && strlen(opt.foutput2))
        {
            fpo1 = fopen(opt.foutput1, "wb");
            fpo2 = fopen(opt.foutput2, "wb");
            if (!fpo1 || !fpo2)
            {
                printf("\n->error  打开输出文件\n");
                goto fail;
            }
            usefo = 1;
        }
        else
        {
            usefo = 0;
        }
        if (opt.streamId >= pCtx->nb_streams)
        {
            printf("\n->StreamId\tERROR\n");
            goto fail;
```

```
    }
    if (opt.lstart > 0)
    {
        err = av_seek_frame(pCtx, opt.streamId, opt.lstart,
AVSEEK_FLAG_ANY);
        if (err < 0)
        {
            printf("\n->(av_seek_frame)\tERROR:\t%d\n", err);
            goto fail;
        }
    }
   / *解码器的配置*/
    if (!opt.nodec)
    {
        /* prepare codec */
        pCodecCtx = pCtx->streams[opt.streamId]->codec;

        if (opt.thread_count <= 16 && opt.thread_count > 0 )
        {
            pCodecCtx->thread_count = opt.thread_count;
            pCodecCtx->thread_type = FF_THREAD_FRAME;
        }
        pCodec = avcodec_find_decoder(pCodecCtx->codec_id);
        if (!pCodec)
        {
            printf("\n->不能找到编解码器!\n");
            goto fail;
        }
        err = avcodec_open2(pCodecCtx, pCodec, 0);
        if (err < 0)
        {
            printf("\n->(avcodec_open)\tERROR:\t%d\n", err);
            goto fail;
        }
        pFrame = avcodec_alloc_frame();

        / *准备设备* /
        if (opt.bplay)
        {
           / *音频设备* /
            dsp.audio_fd = open(OSS_DEVICE, O_WRONLY);
            if (dsp.audio_fd == -1)
            {
                printf("\n->  无法打开音频设备\n");
```

```
                goto fail;
            }
            dsp.channels = pCodecCtx->channels;
            dsp.speed = pCodecCtx->sample_rate;
            dsp.format = map_formats(pCodecCtx->sample_fmt);
            if (set_audio(&dsp) < 0)
            {
                printf("\n-> 不能设置音频设备\n");
                goto fail;
            }
           / *视频设备* /
        }
    }
    nframe = 0;
    while(nframe < opt.frames || opt.frames == -1)
    {
        gettimeofday(&elapsed1, NULL);
        err = av_read_frame(pCtx, &packet);
        if (err < 0)
        {
            printf("\n->(av_read_frame)\tERROR:\t%d\n", err);
            break;
        }
        gettimeofday(&elapsed2, NULL);
        dusecs = (elapsed2.tv_sec - elapsed1.tv_sec)*1000000 + (elapsed2.tv_usec
- elapsed1.tv_usec);
        usecs2 += dusecs;
        timestamp = av_rescale_q(packet.dts,
pCtx->streams[packet.stream_index]->time_base, (AVRational){1,
AV_TIME_BASE});
        printf("\nFrame No %5d stream#%d\tsize %6dB, timestamp:%6lld,
dts:%6lld, pts:%6lld, ", nframe++, packet.stream_index, packet.size,
                timestamp, packet.dts, packet.pts);
        if (packet.stream_index == opt.streamId)
        {
#if 0
            for (i = 0; i < 16; /*packet.size;*/ i++)
            {
                if (i%16 == 0) printf("\n pktdata: ");
                printf("%2x    ", packet.data[i]);
            }
            printf("\n");
#endif
            if (usefo)
```

```
                {
                    fwrite(packet.data, packet.size, 1, fpo1);
                    fflush(fpo1);
                }
                if (pCtx->streams[opt.streamId]->codec->codec_type ==
AVMEDIA_TYPE_VIDEO && !opt.nodec)
                {
                    picheight = pCtx->streams[opt.streamId]->codec->height;
                    picwidth = pCtx->streams[opt.streamId]->codec->width;

                    gettimeofday(&elapsed1, NULL);
                    avcodec_decode_video2(pCodecCtx, pFrame, &got_picture,
&packet);
                    decoded++;
                    gettimeofday(&elapsed2, NULL);
                    dusecs = (elapsed2.tv_sec - elapsed1.tv_sec)*1000000 +
(elapsed2.tv_usec - elapsed1.tv_usec);
                    usecs1 += dusecs;
                    if (got_picture)
                    {
                        printf("[Video: type %d, ref %d, pts %lld, pkt_pts %lld,
pkt_dts %lld]",
                                pFrame->pict_type, pFrame->reference,
pFrame->pts, pFrame->pkt_pts, pFrame->pkt_dts);

                        if (pCtx->streams[opt.streamId]->codec->pix_fmt ==
PIX_FMT_YUV420P)
                        {
                            if (usefo)
                            {
                                linesize = pFrame->linesize[0];
                                pBuf = pFrame->data[0];
                                for (i = 0; i < picheight; i++)
                                {
                                    fwrite(pBuf, picwidth, 1, fpo2);
                                    pBuf += linesize;
                                }
                                linesize = pFrame->linesize[1];
                                pBuf = pFrame->data[1];
                                for (i = 0; i < picheight/2; i++)
                                {
                                    fwrite(pBuf, picwidth/2, 1, fpo2);
                                    pBuf += linesize;
                                }
```

```
                                linesize = pFrame->linesize[2];
                                pBuf = pFrame->data[2];
                                for (i = 0; i < picheight/2; i++)
                                {
                                    fwrite(pBuf, picwidth/2, 1, fpo2);
                                    pBuf += linesize;
                                }
                                fflush(fpo2);
                            }

                            if (opt.bplay)
                            {
                                /* show picture */
                            }
                        }
                    }
                    av_free_packet(&packet);
                }
                else if (pCtx->streams[opt.streamId]->codec->codec_type ==
AVMEDIA_TYPE_AUDIO && !opt.nodec)
                {
                    int got;
                    gettimeofday(&elapsed1, NULL);
                    avcodec_decode_audio4(pCodecCtx, pFrame, &got, &packet);
                    decoded++;
                    gettimeofday(&elapsed2, NULL);
                    dusecs = (elapsed2.tv_sec - elapsed1.tv_sec)*1000000 +
(elapsed2.tv_usec - elapsed1.tv_usec);
                    usecs1 += dusecs;
                  if (got)
                  {
                        printf("[Audio: %5dB raw data, decoding time: %d]",
pFrame->linesize[0], dusecs);
                        if (usefo)
                        {
                            fwrite(pFrame->data[0],    pFrame->linesize[0], 1,
fpo2);
                            fflush(fpo2);
                        }
                        if (opt.bplay)
                        {
                            play_pcm(&dsp, pFrame->data[0],
pFrame->linesize[0]);
                        }
```

```
                }
            }
        }
    }
    if (!opt.nodec && pCodecCtx)
    {
        avcodec_close(pCodecCtx);
    }
    printf("\n%d  帧解析, average %.2f us per frame\n", nframe, usecs2/nframe);
    printf("%d  帧解码，平均  %.2f  我们每帧\n", decoded, usecs1/decoded);

fail:
    if (pCtx)
    {
        avformat_close_input(&pCtx);
    }
    if (fpo1)
    {
        fclose(fpo1);
    }
    if (fpo2)
    {
        fclose(fpo2);
    }
    if (!pFrame)
    {
        av_free(pFrame);
    }
    if (!usefo && (dsp.audio_fd != -1))
    {
        close(dsp.audio_fd);
    }
    return 0;}
```