ffmpeg 的整体分析

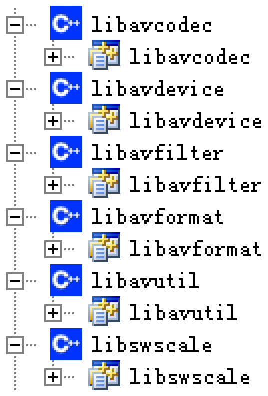
分析的基础：

（一） 源代码：ffmpeg0.63，但是具体的版本要依据笔者电脑 SVN 上版号为：745---ffmpeg-git-a304071-branch

（二） 编译环境：Microsoft Visual C++ 2008 + Intel(R) C++ Compiler 10.1.020

（三） 编译所依赖的库：待续

（四） 所包含工程如下：（各个工程均为 DLL 库）



声明：本书是参考杨书良的《FFMPEG/FFPLAY 源码剖析》编写而成，在此诚挚的感谢他（她）为我们带来如此好的一本书。

参考目录：

1. 《FFMPEG/FFPLAY 源码剖析》作者：杨书良　网络文档

约定：

全文标题： 24 + B + 中

章节标题： 20 + B + 中

章节一级目录：16 + B， 二级目录： 14 + B， 三级目录：12 + B

图片名称：8+中

对于结构体：结构体名：+（库中的位置）+ 注释

对于源代码：无框无底纹表格+10 号字体

对于源代码：凡是已注释的代码，添加字符“//codeInCK”

重点：12+B+/

说明：12+B

第一章　代码综述和基本概念

1.1　ffmpeg 代码综述

本书所依照的源代码是笔者在 ffmpge0.63 版本的基础上移植到 Microsoft Visual C++ 2008 上的，通过 Intel(R) C++ Compiler 10.1.020 支持 C99 语法，大致的修改如下：

一：对所有的汇编代码进行了屏蔽

二：对 Microsoft Visual C++ 2008 的编译为 DLL 的一些问题进行了修改

三：对一些依赖的外部库无法编译通过的代码进行了屏蔽

当然，上面并不是我们本节要讲述的主题，主题是我们所依赖的代码（以后如无特殊

说明， 我们所依赖的代码或者 ffmpeg 代码， 就是指本书所依赖的移植的 ffmpeg 代码， 通常就简称为 ffmpeg 库）的结构：

首先 ffmpeg 库分为 6 个库组成：

（1） libavformat：用于各种音视频封装格式的生成和解析，包括获取编解码所需信息以生成编解码上下文结构和读取音视频帧等功能；

（2） libavcodec：用于各种类型声音/图像编解码；

（3） libavutil：包含一些公共的工具函数；

（4） libswscale：用于视频场景比例缩放、色彩映射转换；

（5） libavfilter： 用于后期效果处理，比如添加水印等；

（6） libavdevice：用于视频源的获取；

上面 6 个库相互协作完成了下面的 3 个示例程序：

（1）ffmpeg：该项目提供的一个工具，可用于格式转换、解码或电视卡即时编码等；

（2）ffsever： 一个 HTTP 多媒体即时广播串流服务器；

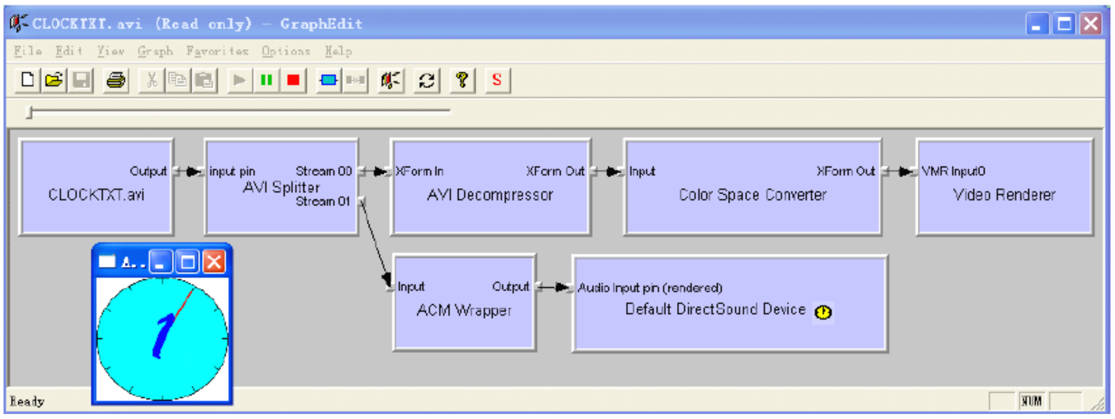
（3）ffplay：是一个简单的播放器，使用 ffmpeg 库解析和解码，通过 SDL 显示；

在我们当前的 ffmpeg 库中并没有包含上面的 3 个示例程序的源代码，后面会专门提出来解析。我们知道了 ffmpeg 库的组成，并不能对 ffmpeg 库有任何深入的认识，后面对于具体的每个库，会罗列每个库文件的作用，详见各个库的章节。

1.2 多媒体框架的基本概念

对于多媒体框架，笔者了解也不多，本着自己的理解，仅作参考。在 Windows 上我们做多媒体方面的开发，一般离不开 DirectShow, Linux 上则用 gstreamer 比较多，至于其他笔者也不甚了解了。

在这里，借鉴一下 DirectShow 的概念进行讲述（参考 1），我们借用 DirectShow GraphEdit 来进行分析：（下图来源于参考 1）



GraphEdit 参考图（图 1.2-1）

在上图中可以直观的看到播放这个媒体文件的基本模块，七个模块按广度顺序：读文件模块， 解复用模块， 视频解码模块， 音频解码音频， 颜色空间转换模块，视频显示模块，音频播放模块。

按照 DirectShow 的称呼， 一个模块叫做一个 filter(过滤器)，模块的输入输出口叫做 pin（管脚）， 有 input pin 和 output pin 两种； 第一个 filter 叫做 Source filter, 每种媒体最后一个 filter 叫做 Sink filter, 如上图所示连成串的所有 filter 组成一个 Graph， 媒体文件的数据

就像流水一样在 Graph 中流动， 各个相关的 filter 各司其职， 最后我们就看到了视频，也听到了声音。

DirectShow 中和播放器有关的 filter 粗略的分为五类， 分别是 Source filter, Demux fiter, Decoder filter， Color Space converter filter, Render filter, 各类的 filter 的功能与作用简述如下：

Source filter：源过滤器， 它的作用是为下级的 demux filter 以包的形式源源不断的提供数据流。在通常情况下，我们有多种方式可以获得数据流， 一种是从本地文件中读取，一种是从网上获取，Sourcefilter 另外的一个作用就是屏蔽读本地文件和获取网络数据的差别，在下一级的 demux filter 看来，本地文件和网络数据是一样的。总结一下， Sourcefilter 有两个作用：其一是为下一级的 demux filter 提供数据流， 其二是屏蔽读本地文件和获取网络数据的差别即对于下一级的 demux filter 看来，本地文件和网络数据是一样的。

Demux filter：解复用过滤器， 它的作用是识别文件类型， 媒体类型， 分离出各媒体的原始数据流， 并打上时钟信息后送给下级的 decoder filter。为识别出不同的文件类型和媒体类型，常规的做法是读取一部分数据， 然后遍历解复用过滤器支持的文件格式和媒体数据格式， 做匹配来确定是哪种文件类型，哪种媒体类型；有些媒体类型的原始数据外面还有其他的信息，比如时间，包大小，是否完整包等等。Demux filter 解析数据包后取出原始数据， 有些类型的媒体不管是否完整包都立即送往下级的 decoder filter, 有些类型的媒体要送完整数据包，此时可能有一些数据包拼接的动作；当然时钟信息的计算也是 demux filter 的工作内容， 这个时钟用于各媒体之间的同步。例如， AVI Splitter 是 Demux filter。

Decoder filter：解码过滤器的作用就是解码数据包，并且把同步时钟信息传递下去，对视频媒体而言，通常是解码成 YUV 数据，然后利用显卡硬件直接支持 YUV 格式数据 Overlay 快速显示的特性让显卡极速显示。YUV 格式是一个统称， 常见的有 YV12, YUY2, UYVY 等等。有些非常古老的显卡和嵌入式系统不支持 YUV 数据显示， 那就要转换成 RGB 格式的数据， 每一帧的每一个像素点都要转换， 分别计算 RGB 分量，并且因为转换是浮点运算， 虽然有定点算法， 还是要耗掉相当一部分 CPU，总体上效率低下。对音频媒体而言，通常是解码成 PCM 数据，然后送给声卡直接输出， 在本例中, AVI Decompress 和 ACM Warper 是 decoder filter。

Color space converter filter: 颜色空间转换过滤器，它的作用是把视频解码器解码出来的数据转换成当前显示系统支持的颜色格式。通常视频解码器解码出来的是 YUV 格式， PC 系统是直接支持 YUV 格式的， 也支持 RGB 格式， 有些嵌入式系统只支持 RGB 格式。在本例中，视频解码器解码出来的是 RGB8 格式的数据， Color space converter filter 把 RGB8 转换为 RGB32 显示。

Render filter：渲染过滤器， 它的作用是在适当的时间渲染相应的媒体，对视频媒体就是直接显示图像， 对音频就是播放声音。视音频同步的策略方法有好几种，其中最简单的一种就是默认视频和音频基准时间相同， 这时音频可以不打时钟信息，通过计算音频的采样频率，量化 bit 数， 声道数等基本参数就知道音频 PCM 的数据速率，按照这个速率往前播放即可；视频必须要使用同步时钟信息来决定什么时候显示。DirectShow 采用一个有序链表，把接受到的数据包放进有序连表中，启动一个定时器， 每次定时器时间就扫描链表，比较时钟信息，或者显示相应的帧，或者什么也不做，每次收到新的数据帧，首先判断时钟信息，如果是历史数据帧就丢弃，如果是将来显示的数据帧就存放在有序链表，如果是当前

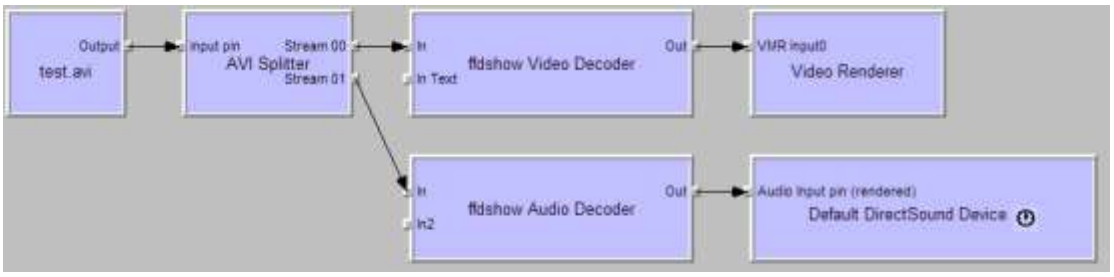
时间帧就直接显示。如果这样，保持视频和音频在人体感觉误差范围内相对的动态同步。在本例中 VideoRender 和 Default DirectSound Device 是 Render filter, 同时也是 Sink filter。

GraphEdit 应用程序：它可以看成是一个支撑平台，支撑框架。它容纳各种 filter，在 filter 间的传递一些通讯消息， 控制 filter 的运行（启动暂停停止），维护 filter 运行状态。GraphEdit 就像超级大管家一样， 既维护管理看得见的 filter，又维护管理看不见的运行支持环境。

通过上面的讲述（主要是复制了参考 1），希望能对多媒体的框架的基本元素有一定的了解，虽然 ffmpeg 库自成一体，但是也是能够归纳到上面的流程之中的。

知识普及：
(一) YUV： 主要用于优化彩色视频信号的传输，使其向后相容老式黑白电视。与 RGB 视频信号传输相比，它最大的优点在于只需占用极少的频宽（RGB 要求三个独立的视频信号同时传输）。其中“Y”表示明亮度（Luminance 或 Luma），也就是灰阶值；而“U”和“V” 表示的则是色度（Chrominance 或 Chroma）。
（二）PCM：主要过程是将话音、图像等模拟信号每隔一定时间进行取样，使其离散化，同时将抽样值按分层单位四舍五入取整量化，同时将抽样值按一组二进制码来表示抽样脉冲的幅值。

笔者也生成了一张图（如下），但不具参考性：

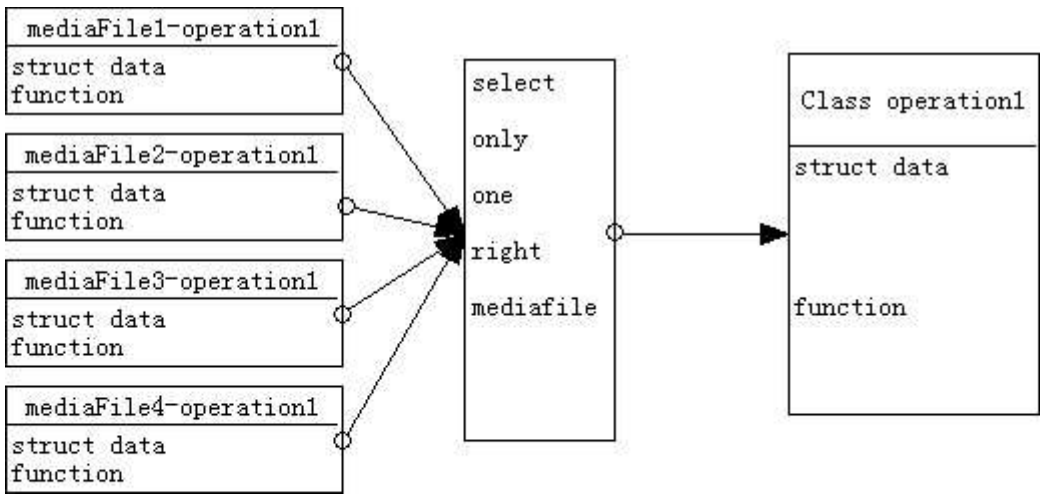


GraphEdit 笔者生成图（图 1.2-2）

1.3 ffmpeg 库的多媒体框架结构

对于 ffmpeg 库，它是可以对应到前一节所讲述的多媒体框架结构的，而且它更加的灵活。可以说，ffmpeg 库设计的两大基本技巧是：无类型数据指针和函数指针。

在设计上，ffmpeg 库首先将多媒体的一些重要的操作功能抽象为相应的一个数据结构，这些功能，比如读文件，识别格式，音视频编解码， 音视频渲染等。每一个抽象的数据结构都包含无类型数据指针和函数指针，他们是可以动态赋值的。这样做的目的是为了能处理各种媒体类型，因为每种媒体类型在一些大的操作功能中都有相应的数据结构和操作函数，这些具体媒体对应的数据结构和操作函数，是在程序编译的时候就确定了，在程序的运行中根据对应的媒体文件，赋值给前面抽象数据结构中的无类型数据指针和函数指针。
如下图所示：



ffmpeg 抽象数据赋值图（图 1.2-2）

下面笔者将对 ffmpeg 库抽象的一些数据结构进行分析，但是在这里的分析，我们并非要分析透彻，主要是为了找到我们所要的关联，并对此进行注释，其他不会进行注释。如需更深入了解，请查看后面章节的分析。

（1）**AVFormatContext**

AVFormatContext:（libavformat\ avformat.h）源的抽象数据结构

```
typedef struct AVFormatContext
{
    //codeInCK
    const AVClass *av_class; //抽象类的标识
    struct AVInputFormat *iformat;//输入源
    struct AVOutputFormat *oformat;//输出源
    void *priv_data;
    AVIOContext *pb;
    unsigned int nb_streams;
#if FF_API_MAX_STREAMS
    AVStream *streams[MAX_STREAMS];
#else
    AVStream **streams;
#endif
    char filename[1024];
    int64_t timestamp;
#if FF_API_OLD_METADATA
    attribute_deprecated char title[512];
    attribute_deprecated char author[512];
    attribute_deprecated char copyright[512];
    attribute_deprecated char comment[512];
    attribute_deprecated char album[512];
    attribute_deprecated int year;    // ID3 year, 0 if none
    attribute_deprecated int track; // track number, 0 if none
    attribute_deprecated char genre[32]; //< ID3 genre
#endif
```

```
    int ctx_flags;
    struct AVPacketList *packet_buffer;
    int64_t start_time;
    int64_t duration;
    int64_t file_size;
    int bit_rate;
    AVStream *cur_st;
#if FF_API_LAVF_UNUSED
    const uint8_t *cur_ptr_deprecated;
    int cur_len_deprecated;
    AVPacket cur_pkt_deprecated;
#endif
    int64_t data_offset; // offset of the first packet
#if FF_API_INDEX_BUILT
    attribute_deprecated int index_built;
#endif
    int mux_rate;
    unsigned int packet_size;
    int preload;
    int max_delay;
#define AVFMT_NOOUTPUTLOOP -1
#define AVFMT_INFINITEOUTPUTLOOP 0
    int loop_output;
    int flags;
#define AVFMT_FLAG_GENPTS           0x0001
#define AVFMT_FLAG_IGNIDX           0x0002
#define AVFMT_FLAG_NONBLOCK         0x0004
#define AVFMT_FLAG_IGNDTS           0x0008
#define AVFMT_FLAG_NOFILLIN         0x0010
#define AVFMT_FLAG_NOPARSE          0x0020
#define AVFMT_FLAG_RTP_HINT         0x0040

    int loop_input;
    unsigned int probesize;
    int max_analyze_duration;
    const uint8_t *key;
    int keylen;
    unsigned int nb_programs;
    AVProgram **programs;
    enum CodecID video_codec_id;//视频编解码器 ID
    enum CodecID audio_codec_id;//音频编解码器 ID
    enum CodecID subtitle_codec_id;//字幕编解码器 ID
    unsigned int max_index_size;
    unsigned int max_picture_buffer;
    unsigned int nb_chapters;
    AVChapter **chapters;
    int debug;
#define FF_FDEBUG_TS        0x0001
    struct AVPacketList *raw_packet_buffer;
    struct AVPacketList *raw_packet_buffer_end;
    struct AVPacketList *packet_buffer_end;
    AVMetadata *metadata;
#define RAW_PACKET_BUFFER_SIZE 2500000
    int raw_packet_buffer_remaining_size;
    int64_t start_time_realtime;
} AVFormatContext;
```

AVFormatContext 的结构如上所示，它是 ffmpeg 库对源进行抽象得到的数据额结构，即 1.2 所说的 Source filter,，但是源也有输入源和输出源，通过其中的 iformat 和 oformat 来进行区分。不过一般都会被第一个成员 const AVClass *av_class;所吸引，它是表示什么数据呢？于是我们引出一个 ffmpeg 库公用的一个类：AVClass

AVClass: （libavutil\ log.h）对所有抽象数据类的标识的抽象数据结构

```
typedef struct
{
    //codeInCK
    const char *class_name;//类的名字
    const char* (*item_name)(void *ctx);//获取类的对象的名字
    const struct AVOption *option;//类的对象的操作项
    int version;//版本号
    int log_level_offset_offset;
    int parent_log_context_offset;
} AVClass;
```

AVClass 我们任然有一个不了解的数据结构，**const struct AVOption *option;**

AVOption:（libavutil\opt.h）

对所有数据结构（不仅仅只是抽象的数据结构，而是所有的）进行赋值的抽象数据结构

```
typedef struct AVOption
{
    //codeInCK
    const char *name;//操作名字
    const char *help;//帮助信息
    int offset;//偏移位置
    enum AVOptionType type;//数据类型
    double default_val;//默认值
    double min;//最大值
    double max;//最小值
    int flags;//标帜
#define AV_OPT_FLAG_ENCODING_PARAM    1
#define AV_OPT_FLAG_DECODING_PARAM    2
#define AV_OPT_FLAG_METADATA          4
#define AV_OPT_FLAG_AUDIO_PARAM         8
#define AV_OPT_FLAG_VIDEO_PARAM         16
#define AV_OPT_FLAG_SUBTITLE_PARAM    32
        const char *unit;
} AVOption;
```

通过上面的信息，我们大致知道了 AVClass 所包含的信息。

那我们继续分析 **AVForm**atContext, 但是据之前所说，它即可以为输入源，也可以为输出源，如果在转码的应用中，就正好是一个闭合循环，即 **AVForm**atContext->iformat 到 **AVForm**atContext->oformat，而且 **AVForm**atContext 可以说包含所有的资源。

在多媒体框架中，Source filter 是输入源，在 ffmpeg 库中 **AVForm**atContext**-**>iformat 是输入流，下面我们将按照一个转码应用的流程进行分析：

从 Source filter 到下一步 Demux filter，实际上就是对应的 AVInputFormat 数据结构

AVInputFormat:（libavformat\avformat.h）对所有输入源的抽象数据结构，属于 demuxer filter

```
ypedef struct AVInputFormat
{
        //codeInCK
        const char *name;//名字
        const char *long_name;//长名字
        int priv_data_size;//数据长度
        int (*read_probe)(AVProbeData *);
        //检测是为指定媒体文件类型的可能性，返回值在 0-100 的范围
        int (*read_header)(struct AVFormatContext *,
                                        AVFormatParameters *ap);
        int (*read_packet)(struct AVFormatContext *,
                                         AVPacket *pkt);
        int (*read_close)(struct AVFormatContext *);
#if 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);
#if FF_API_OLD_METADATA2
      const AVMetadataConv *metadata_conv;
#endif
      struct AVInputFormat *next;
} AVInputFormat;
```

通过上面的结构，我们来粗略的对应一下 Source filter 和 demuxer filter 的承接关系，即 ffmpeg 库中的 AVFormatContext 与 AVInputFormat 的关系。

AVOutputFormat:（libavformat\avformat.h）对所有输出源的抽象数据结构，属于 muxer filter

```
typedef struct AVOutputFormat
{
      //codeInCK
      const char *name;//名字
      const char *long_name;//长名字
      const char *mime_type;//子类型名
      const char *extensions;//扩展名
      int priv_data_size;//数据大小
      enum CodecID audio_codec; //音频编码 ID
      enum CodecID video_codec;//视频编码 ID
      int (*write_header)(struct AVFormatContext *);
      int (*write_packet)(struct AVFormatContext *, AVPacket *pkt);
      int (*write_trailer)(struct AVFormatContext *);
      int flags;
      int (*set_parameters)(struct AVFormatContext *,
            AVFormatParameters *);
      int (*interleave_packet)(
            struct AVFormatContext *,
            AVPacket *out,
            AVPacket *in,
            int flush);
      const struct AVCodecTag *const *codec_tag;
      enum CodecID subtitle_codec;//字幕编码 ID
#if FF_API_OLD_METADATA2
      const AVMetadataConv *metadata_conv;
#endif
      const AVClass *priv_class;
      struct AVOutputFormat *next;
} AVOutputFormat;
```

接着是 Decode filter，但是我们先看看 Decode filter，在 ffmpeg 库中对应 Decode filter 的抽象数据结构是：**AVCodec**

AVCodec：（libavcodec\avcodec.h）对编解码抽象出的数据结构

```
typedef struct AVCodec
{
    //codeInCK
    const char *name;//编解码名字
    enum AVMediaType type;//编解码类型
    enum CodecID id;//编解码 ID
    int priv_data_size;//数据大小
    int (*init)(AVCodecContext *);
    int (*encode)(AVCodecContext *, uint8_t *buf, int buf_size, void *data);
    int (*close)(AVCodecContext *);
    int (*decode)(AVCodecContext *, void *outdata, int *outdata_size, AVPacket *avpkt);
    int capabilities;
    struct AVCodec *next;
    void (*flush)(AVCodecContext *);
    const AVRational *supported_framerates;
    const enum PixelFormat *pix_fmts;
    const char *long_name;
    const int *supported_samplerates;
    const enum AVSampleFormat *sample_fmts;
    const int64_t *channel_layouts;
    uint8_t max_lowres;
    const AVClass *priv_class;
    const AVProfile *profiles;
    int (*init_thread_copy)(AVCodecContext *);
    int (*update_thread_context)(AVCodecContext *dst, const AVCodecContext *src);
} AVCodec;
```

在 ffmpeg 库中 **AVForm**atContext 中有三个成员是与 AVCodec 对应的，它们分别是 **enum**

```
enum CodecID video_codec_id;//视频编解码器 ID
enum CodecID audio_codec_id;//音频编解码器 ID
enum CodecID subtitle_codec_id;//字幕编解码器 ID
```

这三个成员在输入流中对应解码器，在输出流中对应编码器

1.4 ffmpeg 的更深入分析展望

对于 ffmpeg 分析，在后面的章节中会依次对其中的各个库进行解析，但是尽管如此，还不能深入 ffmpeg 所支持的媒体的具体细节，这就需要对具体的媒体文件类型有更多的了解，比如 X264 解码库等。

我们经常说，ffmpeg 是如何强大等，但是它更多的是一个容器，包罗万象的容器。在这个容器之中，笔者如在时间允许的情况下，会选择其中的一些外部库进行一定的分析。

第二章 从 libavformat 开始

2.1 如何从 libavformat 开始？

在前面，提示一切的开始是 Source filter，在 ffmpeg 库中对应的即为 libavformat 中的数据内容。但是 libavformat 库既可以是输入源，也可以是输出源，为了简化问题的分析，我们先从输入源开始分析。

我们知道，任何一个多媒体播放器，转码软件或者视频捕捉软件，都需要一个输入源，这个输入源有不同的种类，但是我们选择使用一个最简单的输入源：一个视频文件名，对于一个视频文件，ffmpeg 是如何开始初始化数据的呢？

首先确定 ffmpeg 在处理输入源的核心数据结构（如下），从它开始：

AVFormatContext：（libavformat\ avformat.h）源的抽象数据结构

```
typedef struct AVFormatContext
{
    //codeInCK
    const AVClass *av_class; //抽象类的标识
    struct AVInputFormat *iformat;//输入源
    struct AVOutputFormat *oformat;//输出源
    void *priv_data;
    AVIOContext *pb;
    unsigned int nb_streams;
#if FF_API_MAX_STREAMS
    AVStream *streams[MAX_STREAMS];
#else
    AVStream **streams;
#endif
    char filename[1024];
    int64_t timestamp;
#if FF_API_OLD_METADATA
    attribute_deprecated char title[512];
    attribute_deprecated char author[512];
    attribute_deprecated char copyright[512];
    attribute_deprecated char comment[512];
    attribute_deprecated char album[512];
    attribute_deprecated int year;    // ID3 year, 0 if none
    attribute_deprecated int track; // track number, 0 if none
    attribute_deprecated char genre[32]; //< ID3 genre
#endif
    int ctx_flags;
    struct AVPacketList *packet_buffer;
    int64_t start_time;
    int64_t duration;
    int64_t file_size;
    int bit_rate;
    AVStream *cur_st;
#if FF_API_LAVF_UNUSED
    const uint8_t *cur_ptr_deprecated;
    int cur_len_deprecated;
    AVPacket cur_pkt_deprecated;
#endif
    int64_t data_offset; // offset of the first packet
#if FF_API_INDEX_BUILT
    attribute_deprecated int index_built;
#endif
    int mux_rate;
    unsigned int packet_size;
    int preload;
    int max_delay;
#define AVFMT_NOOUTPUTLOOP -1
#define AVFMT_INFINITEOUTPUTLOOP 0
    int loop_output;
    int flags;
#define AVFMT_FLAG_GENPTS       0x0001
#define AVFMT_FLAG_IGNIDX       0x0002
#define AVFMT_FLAG_NONBLOCK     0x0004
#define AVFMT_FLAG_IGNDTS       0x0008
#define AVFMT_FLAG_NOFILLIN     0x0010
#define AVFMT_FLAG_NOPARSE      0x0020
#define AVFMT_FLAG_RTP_HINT     0x0040

    int loop_input;
    unsigned int probesize;
    int max_analyze_duration;
    const uint8_t *key;
    int keylen;
    unsigned int nb_programs;
    AVProgram **programs;
    enum CodecID video_codec_id;//视频编解码器 ID
    enum CodecID audio_codec_id;//音频编解码器 ID
    enum CodecID subtitle_codec_id;//字幕编解码器 ID
    unsigned int max_index_size;
    unsigned int max_picture_buffer;
    unsigned int nb_chapters;
    AVChapter **chapters;
    int debug;
#define FF_FDEBUG_TS        0x0001
    struct AVPacketList *raw_packet_buffer;
    struct AVPacketList *raw_packet_buffer_end;
    struct AVPacketList *packet_buffer_end;
    AVMetadata *metadata;
#define RAW_PACKET_BUFFER_SIZE 2500000
    int raw_packet_buffer_remaining_size;
    int64_t start_time_realtime;
} AVFormatContext;
```

在 C 语言中是没有构造函数替你自动的设置一些默认参数的，但是不得不说在对于多媒体的一些程序，很多都需要设置一些默认参数的。而且在 ffmpeg 的设计思想中，面向对象的思想也是贯穿在整个代码中，所以对待一些抽象的数据结构和媒体数据结构，都需要一些函数来初始化设置一些参数的默认值。

对于 AVFormatContext，也是如此，它的默认初始化函数为：

下面是 AVFormatContext 的默认初始化函数

```
AVFormatContext *avformat_alloc_context(void)
{
    //codeInCK
    AVFormatContext *ic;
    //分配一个 AVFormatContext 大小的内存块
    ic = av_malloc(sizeof(AVFormatContext));
    if (!ic)
    {
        return ic;
    }
    //给 AVFormatContext 设置默认值
    avformat_get_context_defaults(ic);//解释 1
    //对于 av_class，给它设置了一个静态数据
    ic->av_class = &av_format_context_class;//解释 2
    return ic;
}
```

解释 1：

```
static void avformat_get_context_defaults(AVFormatContext *s)
{
    //codeInCK
    //全部置零
    memset(s, 0, sizeof(AVFormatContext));
    //对于 av_class，给它设置了一个静态数据
    s->av_class = &av_format_context_class;
    //将 AVFormatContext 的设置选项设置默认值
    av_opt_set_defaults(s);//解释 1-1
}
```

通过上面的代码，我们可以看出 AVFormatContext *avformat_alloc_context(void)

中的 ic->av_class = &av_format_context_class;
与 static void avformat_get_context_defaults(AVFormatContext *s)中的
s->av_class = &av_format_context_class;是重复的代码，应该将他们优化一下。

解释 1-1：

```
void av_opt_set_defaults(void *s)
{
    //codeInCK
    av_opt_set_defaults2(s, 0, 0);//注释 1-1-1
}
```

对于这个函数，笔者会用较为深入的笔墨来注释它，因为它基本上代表了大多数的值的设置方式。

注释 1-1-1：

```
void av_opt_set_defaults2(void *s, int mask, int flags)
{
    //codeInCK
    const AVOption *opt = NULL;
    //寻找操作项
    while ((opt = av_next_option(s, opt)) != NULL)//注释 1-1-1-1
    {
        if ((opt->flags & mask) != flags)
        {
            continue;
        }
        //根据数据类型来设置相应的值
        switch (opt->type)
        {
        case FF_OPT_TYPE_CONST:
            //Nothing to be done here
            break;
        case FF_OPT_TYPE_FLAGS:
        case FF_OPT_TYPE_INT:
        {
            int val;
            val = opt->default_val;
            av_set_int(s, opt->name, val);//注释 1-1-1-2
        }
        break;
        case FF_OPT_TYPE_INT64:
            if ((double)(opt->default_val + 0.6) == opt->default_val)
            {
                av_log(s,
                    AV_LOG_DEBUG,
                    "loss of precision in default of %s\n",
                    opt->name);
            }
            av_set_int(s,
                opt->name,
                opt->default_val);
            break;
        case FF_OPT_TYPE_DOUBLE:
        case FF_OPT_TYPE_FLOAT:
        {
            double val;
            val = opt->default_val;
            av_set_double(s, opt->name, val);//注释 1-1-1-3
        }
        break;
        case FF_OPT_TYPE_RATIONAL:
        {
            AVRational val;
            val = av_d2q(opt->default_val, INT_MAX);//注释 1-1-1-4
            av_set_q(s, opt->name, val);//注释 1-1-1-5
        }
        break;
        case FF_OPT_TYPE_STRING:
        case FF_OPT_TYPE_BINARY:
            //Cannot set default for string as default_val is of type * double
            break;
        default:
            av_log(s,
                AV_LOG_DEBUG,
                "AVOption type %d of option %s not implemented yet\n",
                opt->type, opt->name);
        }
    }
}
```

注释 1-1-1-1：

```
const AVOption *av_next_option(void *obj, const AVOption *last)
{
    //codeInCK
    //如果当前操作项存在并且下一个操作项的操作名存在
    if (last && last[1].name)
    {
        return ++last;
    }
    //如果当前操作项存在但是下一个操作项的操作名不存在
    else if (last)
    {
        return NULL;
    }
    //如果当前操作项不存在
    else
    {
        return (*(AVClass **)obj)->option;
    }
}
```

注释 1-1-1-2：

```
const AVOption *av_set_int(void *obj, const char *name, int64_t n)
{
    //codeInCK
    return av_set_number(obj, name, 1, 1, n);//注释 1-1-1-2-1
}
```

注释 1-1-1-2-1：

```
static const AVOption *av_set_number(void *obj, const char *name, double num, int den,
int64_t intnum)
{
    //codeInCK
    const AVOption *o = NULL;
    if (av_set_number2(obj, name, num, den, intnum, &o) < 0)//注释 1-1-1-2-1-1
    {
        return NULL;
    }
    else
    {
        return o;
    }
}
```

注释 1-1-1-2-1-1：

```
static int av_set_number2(void *obj,
                                  const char *name,
                                  double num,
                                  int den,
                                  int64_t intnum,
                                  const AVOption **o_out)
{
    //codeInCK
    const AVOption *o =
        av_find_opt(obj, name, NULL, 0, 0);注释 1-1-1-2-1-1-1
```

```
void *dst;
if (o_out)
        *o_out = o;
if (!o || o->offset <= 0)
        return AVERROR(ENOENT);

if (o->max * den < num * intnum || o->min * den > num * intnum)
{
        av_log(obj,
                 AV_LOG_ERROR,
                 "Value %lf for parameter '%s' out of range\n",
                 num, name);
        return AVERROR(ERANGE);
}

dst = ((uint8_t *)obj) + o->offset;

switch (o->type)
{
case FF_OPT_TYPE_FLAGS:
case FF_OPT_TYPE_INT:
        *(int *)dst = llrint(num / den) * intnum;
        break;
case FF_OPT_TYPE_INT64:
        *(int64_t *)dst = llrint(num / den) * intnum;
        break;
case FF_OPT_TYPE_FLOAT:
        *(float *)dst = num * intnum / den;
        break;
case FF_OPT_TYPE_DOUBLE:
        *(double *)dst = num * intnum / den;
        break;
case FF_OPT_TYPE_RATIONAL:
        if ((int)num == num) *(AVRational *)dst = (AVRational)
        {
                num *intnum, den
        };
        else
        {
                 *(AVRational *)dst
                         = av_d2q(num * intnum / den, 1 << 24);
        }
        break;
default:
        return AVERROR(EINVAL);
    }
    return 0;
}
```

注释 1-1-1-2-1-1-1：

```
const AVOption *av_find_opt(void *v,
                            const char *name,
                            const char *unit,
                            int mask,
                            int flags)
{
    AVClass *c = *(AVClass **)v; //FIXME silly way of storing AVClass
    const AVOption *o = c->option;
    //只是一个循环的查找
    for (; o && o->name; o++)
    {
        if (!strcmp(o->name, name)
             && (!unit
                    || (o->unit
                    && !strcmp(o->unit, unit)))
                 && (o->flags & mask) == flags)
            return o;
    }
    return NULL;
}
```

上面展开了很多代码，但是我们不应该模糊主次之分，现在的主干是解析 AVFormatContext 是如何进行初始化设置默认值的。在上面的代码中，看到在最后，默认值设置集中在 void av_opt_set_defaults(void *s)函数中，它是 libavutil\opt.c 位置的代码，即它是 ffmpeg 库中的公用库 libautil 的代码。该函数对所有的抽象类或具体类都是进行相同的操作，所不同的是传入的参数。在这里传入的参数是 AVFormatContext。

不过我们先回头看看 void av_opt_set_defaults(void *s)，它就是典型了运用了无类型指针数据来进行传递数据。但是 void av_opt_set_defaults2(void *s, int mask, int flags)中，需要使用 const AVOption *av_next_option(void *obj, const AVOption *last)函数来寻找一个 AVOption，实际上 av_next_option 才是我们第一个要分析的重点所在。对于一个无类型指针数据，它是如何找到 AVOption，其中的一行很关键：(*(AVClass **)obj)->option; 要知道我们最开始传入的是一个 AVFormatContext 即当前的(*(AVClass **)obj)，这两者如何能够划上等号？只有一种情况是 AVFormatContext 的第一个成员必须是 AVClass 类型，事实上正是如此，这就是 av_opt_set_defaults 能够得到正确执行的基础。

*重点：任何需要执行 void av_opt_set_defaults(void *s)的数据类型的第一个成员必须是 AVClass 类型。*

找到了一个 AVOption，就有了具体操作的规则，但是要操作的对象仍然是 AVFormatContext，那如何设置到 AVFormatContext 呢？通过上面的代码，可以看到任何一个设置最终都是 static int av_set_number2(void *obj,

```
                            const char *name,
                                        double num,
                                        int den,
                                        int64_t intnum,
                                        const AVOption **o_out)
```
在这个函数中也是需要用到 const AVOption *av_find_opt(void *v,

```
                                          const char *name,
                                          const char *unit,
                                          int mask,
                                          int flags)
```
来找到具体要设置的 AVOption，与前面的 AVOption 有何不同呢？应该说就是一样的，但为什么要分开进行查找呢？笔者的理解是为了解除一些耦合性，避免一些多余的参数传递。但是他们所依赖的基础都是一样的：该数据结构的第一个成员必须是 AVClass 类型。

有了上面的分析，现在的焦点应该是在 AVOption 上，通过它是如何来设置值的呢？还是再次看一下 AVOption 的数据结构：

```
typedef struct AVOption
{
    //codeInCK
    const char *name;//操作名字---它对应这数据结构对应的成员名
    const char *help;//帮助信息
    int offset;//偏移位置
    enum AVOptionType type;//数据类型
    double default_val;//默认值
    double min;//最大值
    double max;//最小值
    int flags;//标帜
#define AV_OPT_FLAG_ENCODING_PARAM    1
#define AV_OPT_FLAG_DECODING_PARAM    2
#define AV_OPT_FLAG_METADATA          4
#define AV_OPT_FLAG_AUDIO_PARAM       8
#define AV_OPT_FLAG_VIDEO_PARAM       16
#define AV_OPT_FLAG_SUBTITLE_PARAM    32
    const char *unit;
} AVOption;
```
通过它可以了解，通过该结构中的偏移位置，就可以得到该数据要写入的内存地址，通过数据类型，默认值，最大值，最小值，标帜，就可以确定需要设定的值的具体值，如此就可以顺利对一个数据结构变量设置相应的默认值了。

前面有一个基础是有关 AVClass 类型的即数据结构的第一个成员必须是 AVClass 类型，而且 void av_opt_set_defaults(void *s)的实际运行也是以 AVClass 类型的值作为基础的。在 AVFormatContext 类型的默认值设置中，就可以看到它的 AVClass 类型是这样设置的，s->av_class = &av_format_context_class; 那就去看一下 av_format_context_class 有如何的奥秘？

av_format_context_class 的定义如下：

```
static const AVClass av_format_context_class =
{
    "AVFormatContext",
    format_to_name,
    options,
    LIBAVUTIL_VERSION_INT
```

}; 首先它是一个静态 const 值，它指定了 AVClass 的最重要的 option 项，当然还有其他项。

代码讲述到这里，笔者相信，对一个数据结构如何设置默认值，应该有了一个大概的印象。再来延伸一下，实际上在 ffmpeg 库中，无论抽象的数据结构还是具体特定的数据结构都是通过这样的方式来设置参数的。如果后面的代码中再遇到与之类似的情况，请参照此处联想理解，笔者也会简单一笔带过。

如果对 AVFormatContext 在生成的时候，设定了默认值，但是对于我们要传入的对象：一个视频文件， 如何通过它来设定具体的值呢？具体要做的事是通过一个传输的文件来设置好一个 Source filter 和一个 demuxer filter, 在 ffmpeg 中，他们都在 AVFormatContext 中。Source filter 即可以等于 AVFormatContext，demuxer filter 即可以等于 AVFormatContext 中的 AVInputFormat *iformat;。

说明：下面的代码讲述的大致次序是依赖 ffmpeg.c 的转码程序的源代码进行下去的，读者如需参考，请阅读 ffmpeg/ffmpeg.c 的代码。笔者会简称为转码程序。

在一个转码程序中，设置一个输入源的 AVFormatContext，会有一个很重要的函数，这个函数为 int av_open_input_file(AVFormatContext **ic_ptr, const char *filename,
AVInputFormat *fmt,
int buf_size,
AVFormatParameters *ap)，那我们就从这个函数开始分析。

2.2 用 av_open_input_file 打开一个输入源

一个输入源，要是一键开启应该是方便很多。不过，虽然没有如此便利的一个函数能完全做到，但是却有一个非常接近这样的一个函数，它就是：av_open_input_file，下面的代码会对它进行深入的分析，但是接下来的分析不会像前面一样一直深入到底，会有一些纵向展开的内容。

```
int av_open_input_file(AVFormatContext **ic_ptr, const char *filename,
                       AVInputFormat *fmt,
                       int buf_size,
                       AVFormatParameters *ap)
{
    int err;
    AVProbeData probe_data, *pd = &probe_data;
    AVIOContext *pb = NULL;
    void *logctx = ap && ap->prealloced_context ? *ic_ptr : NULL;

    pd->filename = "";
    if (filename)
    {
        pd->filename = filename;
    }
    pd->buf = NULL;
```

```
pd->buf_size = 0;

//如果没有指定输入者的格式，就需要通过文件名获取
if (!fmt)
{
       // guess format if no file can be opened
       fmt = av_probe_input_format(pd, 0);//注释 1
}
//Do not open file if the format does not need it. XXX: specific
//      hack needed to handle RTSP/TCP
if (!fmt || !(fmt->flags & AVFMT_NOFILE))
{
       // if no file needed do not try to open one
       if ((err = avio_open(&pb, filename, AVIO_RDONLY)) < 0)
       {
              goto fail;
       }
       if (buf_size > 0)
       {
              ffio_set_buf_size(pb, buf_size);
       }
       if (!fmt && (err = av_probe_input_buffer(pb,
               &fmt, filename,
               logctx, 0,
               logctx ? (*ic_ptr)->probesize : 0)) < 0)
       {
              goto fail;
       }
}

// if still no format found, error
if (!fmt)
{
       err = AVERROR_INVALIDDATA;
       goto fail;
}

// check filename in case an image number is expected
if (fmt->flags & AVFMT_NEEDNUMBER)
{
       if (!av_filename_number_test(filename))
       {
              err = AVERROR_NUMEXPECTED;
              goto fail;
```

```
        }
    }
    err = av_open_input_stream(ic_ptr, pb, filename, fmt, ap);
    if (err)
    {
        goto fail;
    }
    return 0;
fail:
    av_freep(&pd->buf);
    if (pb)
    {
        avio_close(pb);
    }
    if (ap && ap->prealloced_context)
    {
        av_free(*ic_ptr);
    }
    *ic_ptr = NULL;
    return err;
}
```

对于这个函数，为了便于分析，先假设只有一个文件名传入，其他都是空值。首先是进行整体上的分析：

其一：不管有没有 AVInputFormat *fmt 的参数传过来，在 av_open_input_file 函数中都需要一个非空的 AVInputFormat *fmt。如果传进来是空，通过文件名获取。前面讲过：AVInputFormat 就是 demuxer filter.

其二：AVIOContext *pb 作为 AVFormatContext 的成员，在 av_open_input_file 函数中发挥着重要的作用。

首先对如何通过文件名获取 AVInputFormat *fmt 进行分析，它的主要代码是：

```
fmt = av_probe_input_format(pd, 0);//注释 1
```

注释 1：

```
AVInputFormat *av_probe_input_format(AVProbeData *pd, int is_opened)
{
    //codeInCK
    int score = 0;
    return av_probe_input_format2(pd, is_opened, &score);
}
```

对于它，先不急于展开，我们注意到它的传入参数 AVProbeData *pd，这是一个什么样的数据结构呢？

AVProbeData：（libavformat\avformat.h）它主要是为了识别各种媒体文件的格式而保留一部分文件头数据来分析配备各种媒体文件的类型，主要用于 demuxer filter 中。生成该结构数据读取媒体文件与其他的读取媒体文件数据是独立的，不可混淆。

```
typedef struct AVProbeData
{
    //codeInCK
    const char *filename;//文件名
    unsigned char *buf;
    //Buffer must have AVPROBE_PADDING_SIZE
    //of extra allocated bytes filled with zero.
    int buf_size;
    //< Size of buf except extra allocated bytes
} AVProbeData;
```

现在利用 AVProbeData 数据结构来执行 av_probe_input_format，最终执行的地方是函数：

```
AVInputFormat *av_probe_input_format3(AVProbeData *pd,
                                      int is_opened, int *score_ret)
{
    //codeInCK
    AVProbeData lpd = *pd;
    AVInputFormat *fmt1 = NULL, *fmt;
    int score, score_max = 0;
    //检查是否含有 id3v3 格式的头部信息
    if (lpd.buf_size > 10
        && ff_id3v2_match(lpd.buf, ID3v2_DEFAULT_MAGIC))
    {
        //如果包含 id3v3 格式的头部信息，则跳过该 id3v3 信息头部
        int id3len = ff_id3v2_tag_len(lpd.buf);
        if (lpd.buf_size > id3len + 16)
        {
            lpd.buf += id3len;
            lpd.buf_size -= id3len;
        }
    }
    //此处的设计思路是：
    //遍历每种文件格式(每次都是完全遍历)，选出其中匹配百分比最高的一个
    //将该文件格式的静态数据结构返回
    fmt = NULL;
    while ((fmt1 = av_iformat_next(fmt1)))
    {
        if (!is_opened == !(fmt1->flags & AVFMT_NOFILE))
            continue;
        score = 0;
        if (fmt1->read_probe)
        {
            score = fmt1->read_probe(&lpd);
            if(!score
                && fmt1->extensions
                && av_match_ext(lpd.filename, fmt1->extensions))
            {
                score = 1;
            }
        }
        else if (fmt1->extensions)
        {
            if (av_match_ext(lpd.filename, fmt1->extensions))
            {
                score = 50;
            }
        }
        //获取更高的匹配度的，设置要返回的 fmt
        if (score > score_max)
        {
            score_max = score;
            fmt = fmt1;
        }
        //如果当前媒体文件格式的匹配度和之前的有相同，则将返回的 fmt 设置为空
        else if (score == score_max)
        {
            fmt = NULL;
        }
    }
    *score_ret = score_max;
    return fmt;
}
```

对待这个函数，其他先不说，有一点，传入的参数 **AVProbeData *pd** 是在 **AVInputFormat** 的成员 **int (*read_probe)(AVProbeData *);**中执行是不应该做任何改变的即不应该在该函数中被改变。

在函数 **av_probe_input_format3** 中，它的目标是获取匹配度最高的媒体格式类型并返回匹配度，所以并不是一种绝对正确的选择。它通过遍历所有的媒体文件格式，通过 **int (*read_probe)(AVProbeData *)**函数或后缀名两项来判断匹配度，前者有更高的优先性。

讲到这里，需要暂停一下，前面讲到 **AVInputFormat** 对于 **av_open_input_file** 的重要性，对于代码 **fmt = av_probe_input_format(pd, 0);**来说，因为传入的参数 **AVProbeData *pd** 里面的 **buf** 成员没有任何数据，所以更多的作用也就是通过文件名或者后缀名来寻找一下媒体文件格式。但是前面讲到的最后的判断函数

```
AVInputFormat *av_probe_input_format3(AVProbeData *pd,
                                    int is_opened, int *score_ret)
```

它却是通用的，在任何需要判断格式的地方都可以使用它来寻找一个最优媒体文件格式。另外就是在在 **AVInputFormat** 的成员 **int (*read_probe)(AVProbeData *);**，特别强调的是它的作用就是判断媒体文件的匹配性。

现在回到我们的主干：**av_open_input_file**，顺着这个函数阅读下面的代码：

err=avio_open(&pb, filename, AVIO_RDONLY);

```
int avio_open(AVIOContext **s, const char *filename, int flags)
{
    URLContext *h;
    int err;

    err = ffurl_open(&h, filename, flags);
    if (err < 0)
    {
        return err;
    }
    err = ffio_fdopen(s, h);
    if (err < 0)
    {
        ffurl_close(h);
        return err;
    }
    return 0;
}
```

这个函数会设计到一个非常重要的数据结构：AVIOContext，在围绕着 AVIOContext 周围牵连着 ffmpeg 库的一个大的的模块: IO 模块，它的代码主要集中在 libavformat\avio.c、libavformat\avio.h。

2.2.1 ffmpeg 库的 IO 模块

谈到 IO 模块，一切的输入输出数据都是 IO 模块的职责，本地数据和网络数据的输入和输出无疑是其中最重要的两个方面。Ffmpeg 库经过巧妙的设计，正好将两者通过抽象统一起来。可以先来探讨一下 ffmpeg 库的 IO 模块的设计思路，它的核心思路就是：协议。通过各种协议将各种数据的输入输出进行了规范，然后在通过对各种协议提取出一个抽象的数据结构，从而合理的统一了各种 IO 操作。

重点：ffmpeg 的 IO 模块，通过建立各种协议来规范不同数据的 IO 操作，然后再对各种协议进行抽象，提取一个抽象数据结构，从而统一个各种 IO 操作的接口。

虽然笔者要开始分析 ffmpeg 库的 IO 模块，但是不喜欢把几个源代码文件一个个从头分析到尾，然后告诉大家分析结束了。笔者认为，这对初学者是很难深入理解的，我们需要一个进入的开始点，通过这个开始点，一步步深入探究。现在对于 ffmpeg 库的 IO 模块的开始点，笔者选择从函数：int avio_open(AVIOContext **s, const char *filename, int flags)开始，与此相关的数据结构会相继展开，最后融合为一个整体来理解。

选择了开始点，我们还有一个核心的数据结构：AVIOContext

```
typedef struct
{
    //codeInCK
    unsigned char *buffer;   //缓存 buf
    int buffer_size; //缓存 buf 大小
    unsigned char *buf_ptr;//buf 的当前可写的位置
    unsigned char *buf_end;//buf 的尾部
    void *opaque;//保存 URLContext 的数据
    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;//当前 AVIOContext 读取到的位置
    int must_flush;//是否立即处理缓存
    int eof_reached;//是否到了结尾处
    int write_flag;//是否可写
#if FF_API_OLD_AVIO
    attribute_deprecated int is_streamed;
#endif
    int max_packet_size;
    unsigned long checksum;//校验和
    unsigned char *checksum_ptr;//校验和所计算的当前校验位置
    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;
```

看了这个数据结构，或许之前就注意到了，每个抽象数据结构所包含的函数指针成员也是对各种不同具体类型对象的某个操作的一个抽象，这也算是 ffmpeg 核心设计思想之一吧。

再回过头来看看前面选定的开始点：

```
int avio_open(AVIOContext **s, const char *filename, int flags)
{
    //codeInCK
    URLContext *h;
    int err;
    //通过文件名打开一种协议类型
    err = ffurl_open(&h, filename, flags);//注释 1
    if (err < 0)
    {
        return err;
    }
    err = ffio_fdopen(s, h);//注释 2
    if (err < 0)
    {
        ffurl_close(h);//注释 3
        return err;
    }
    return 0;
}
```

注释 1：

```
int ffurl_open(URLContext **puc, const char *filename, int flags)
{
    //codeInCK
    int ret = ffurl_alloc(puc, filename, flags);//注释 1-1
    if (ret)
    {
        return ret;
    }
    ret = ffurl_connect(*puc);//注释 1-2
    if (!ret)
    {
        return 0;
    }
    ffurl_close(*puc);//注释 1-3
    *puc = NULL;
    return ret;
}
```

注释 1-1：

```
int ffurl_alloc(URLContext **puc, const char *filename, int flags)
{
    URLProtocol *up;
    char proto_str[128], proto_nested[128], *ptr;
    //寻找第一个特别的字符(即数字和字母除外的)所在的位置的长度
    size_t proto_len = strspn(filename, URL_SCHEME_CHARS);

    //获取协议的类型的名字(这个由 proto_len 来决定)
    if (filename[proto_len] != ':' || is_dos_path(filename))
    {
        strcpy(proto_str,
        "file");
    }
    else
    {
        av_strlcpy(proto_str,
        filename,
        FFMIN(proto_len + 1,
        sizeof(proto_str)));
    }
    av_strlcpy(proto_nested, proto_str, sizeof(proto_nested));
    if ((ptr = strchr(proto_nested, '+')))
    {
        *ptr = '\0';
    }
    //这里存在疑问？既然得到了协议的名字: proto_str，
    //但是为什么还要再增加一个：proto_nested，为了更大的精确性，
    //也许这里只是一个嵌套协议的一部分，所以这里和之前很多有类似性，
    //为了最大可能的寻找到对应的协议
    //通过遍历来寻找协议的静态变量
    up = first_protocol;
    while (up != NULL)
    {
        if (!strcmp(proto_str, up->name))
        {
            return url_alloc_for_protocol (puc, up, filename, flags);//注释 1-1-1
        }
        if (up->flags & URL_PROTOCOL_FLAG_NESTED_SCHEME
        && !strcmp(proto_nested, up->name))
        {
            return url_alloc_for_protocol (puc, up, filename, flags);
        }
        up = up->next;
    }
    *puc = NULL;
    return AVERROR(ENOENT);
}
```

一下子延伸到这里，实际上我们需要关注的 ffmpeg 库 IO 模块的另一个非常重要的数据结构已经出现了。

URLContext：（libavformat\url.h）对各种协议所代表的具体数据的抽象

```
typedef struct URLContext
{
    //codeInCK
    const AVClass *av_class;//抽象类的信息标识
    struct URLProtocol *prot;//协议类型数据
    void *priv_data;//协议具体的对象数据
    char *filename;//文件名
    int flags;
    int max_packet_size;//最大包的大小
    int is_streamed;
    int is_connected;
} URLContext;
```

实际上在具体的代码中还有一个另外的 URLContext（libavformat\avio.h），在这里我们屏蔽调这个定义，因为它的存在是为了兼容旧的 ffmpeg 的 API，显然我们目前是不会用到旧的 API。

重点：ffmpeg 的新的和旧的 IO 模块的的 API 有什么区别？最大的区别就是有没有缓存,新的有缓存，旧的没有缓存。

在 **URLContext** 的定义中，我们发现了常见的 **void *priv_data;** 像这样的数据结构是在抽象数据结构是经常出现的，这种无类型的数据指针，所代表的数据往往是针对具体对象的数据。通过统一的基于抽象数据结构函数接口，在落实到具体的设置的时候，就是通过类似 **void *priv_data;**的数据成员来进行具体对象的设置。、

在 **URLContext** 的定义中，也包含着同样一个抽象的数据结构 **struct URLProtocol *prot;** 该数据结构是对各种 **IO** 数据类型的协议规范的抽象即协议抽象数据结构。

URLProtocol（**libavformat\url.h**）：对各种协议的一个抽象数据

```
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;
```

对于该数据结构的成员，很容易理解，大多数都是一些函数指针，在具体的运行过程中，这些函数指针会被用静态的全局变量进行赋值。

注释 **1-1-1**：

```
static int url_alloc_for_protocol (URLContext **puc,
            struct URLProtocol *up,
            const char *filename,
            int flags)
{
    URLContext *uc;
    int err;

#if CONFIG_NETWORK
    //不管什么情况都会检查一下是否初始化网络环境
    if (!ff_network_init())
        return AVERROR(EIO);
#endif
    uc = av_mallocz(sizeof(URLContext) + strlen(filename) + 1);
    if (!uc)
    {
        err = AVERROR(ENOMEM);
        goto fail;
    }
#if FF_API_URL_CLASS
    uc->av_class = &urlcontext_class;
#endif
    uc->filename = (char *) &uc[1];
    strcpy(uc->filename, filename);
    uc->prot = up;
    uc->flags = flags;
    uc->is_streamed = 0; // default = not streamed
    uc->max_packet_size = 0; //default: stream file
    if (up->priv_data_size)
    {
        //分配一个具体协议对象数据的内存块
        uc->priv_data = av_mallocz(up->priv_data_size);
        //对分配了内存块的具体协议对象数据设置默认值
        if (up->priv_data_class)
        {
            *(const AVClass **)uc->priv_data = up->priv_data_class;
            av_opt_set_defaults(uc->priv_data);
        }
    }

    *puc = uc;
    return 0;
fail:
    *puc = NULL;
#if CONFIG_NETWORK
    //关闭网络流，只有在极个别的特殊情况下
    ff_network_close();
#endif
    return err;
}
```

在 **alloc_for_protocol** 的代码中，我们应该要注意两点：第一点是传入 **alloc_for_protocol** 函数的参数 **struct URLProtocol *up**，它的来历是由一个大多数的规律的，它的值一般由 **const** 的静态数据指定；第二点是在该函数中有一个类似前面指定默认值的代码，这和前面的 **av_opt_set_defaults** 具有类型的原理，在此不再深入解析。

经过 **alloc_for_protocol** 的函数，我们需要一步步返回，回到我们当前的主干 **ffurl_open**。我们接下来需要关注的函数是：**int ffurl_connect(URLContext *uc)**；
通过 **ffurl_open** 函数中的 **ffurl_alloc** 的执行，我们得到了输入对象所对应的协议，这在 **ffurl_connect** 中将发挥作用。

注释 1-2：

```
int ffurl_connect(URLContext *uc)
{
    //codeInCK
    //通过获取的协议类型数据的相关函数带来 IO 端口
    int err = uc->prot->url_open(uc, uc->filename, uc->flags);
    if (err)
    {
        return err;
    }
    uc->is_connected = 1;
    //We must be careful here as ffurl_seek() could be slow, for example for http
    if((uc->flags & (AVIO_WRONLY | AVIO_RDWR))
        || !strcmp(uc->prot->name, "file"))
    {
        if(!uc->is_streamed && ffurl_seek(uc, 0, SEEK_SET) < 0)
        {
            uc->is_streamed = 1;
        }
    }
    return 0;
}
int ffurl_close(URLContext *h)
{
    int ret = 0;
    if (!h) return 0; // can happen when ffurl_open fails

    //通过协议对应的函数来关闭 IO 端口
    if (h->is_connected && h->prot->url_close)
    {
        ret = h->prot->url_close(h);
    }
#if CONFIG_NETWORK
    //这里有一点不可理解了，难道每次都必须关闭整个网络环境吗？
    //存有疑问，期待后面的回复？
    ff_network_close();
#endif
    if (h->prot->priv_data_size)
    {
        av_free(h->priv_data);
    }
    av_free(h);
    return ret;
}
```

代码解析进行到这里，似乎进入了一个不那么有趣的地方，其实不然。虽然这些函数的接口是很容易理解的，但是这些接口是可以嵌套的，这是一个重点所在。

重点：ffmpeg 库的 IO 模块，它的 IO 操作函数 API 是可以嵌套的，比如 RTP 协议会包含 UDP 或 TCP 等协议。

对于 **int ffurl_open(URLContext **puc, const char *filename, int flags)**；到这里，已经分析完了，我们要回到我们的主干 **avio_open** 函数中，开始关注下一个函数：

int ffio_fdopen(AVIOContext **s, URLContext *h)

注释 **2**：

```
int ffio_fdopen(AVIOContext **s, URLContext *h)
{
    //codeInCK
    uint8_t *buffer;
    int buffer_size, max_packet_size;

    //获取缓存的大小
    max_packet_size = h->max_packet_size;
    if (max_packet_size)
    {
        //no need to bufferize more than one packet
        buffer_size = max_packet_size;
    }
    else
    {
        buffer_size = IO_BUFFER_SIZE;
    }
    //分配缓存
    buffer = av_malloc(buffer_size);
    if (!buffer)
    {
        return AVERROR(ENOMEM);
    }
    //分配 AVIOContext
    *s = av_mallocz(sizeof(AVIOContext));
    if(!*s)
    {
        av_free(buffer);
        return AVERROR(ENOMEM);
    }

    //初始化 AVIOContext
    //注释 2-1
    if (ffio_init_context(*s,
        buffer,
        buffer_size,
            (h->flags & AVIO_WRONLY || h->flags & AVIO_RDWR),
            h,
            ffurl_read,
            ffurl_write,
            ffurl_seek) < 0)
    {
        av_free(buffer);
        av_freep(s);
        return AVERROR(EIO);
    }
#if FF_API_OLD_AVIO
    (*s)->is_streamed = h->is_streamed;
#endif
    (*s)->seekable = h->is_streamed ? 0 : AVIO_SEEKABLE_NORMAL;
    (*s)->max_packet_size = max_packet_size;
    if(h->prot)
    {
        //将协议函数设置给 AVIOContext
        (*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;
    }
    return 0;
}
```

通过上面的代码，可以看到 ffio_fdopen 的作用就是通过 URLContext *h 来生成一个 AVIOContext s, 其中的代码不难理解。但是从前面的 ffurl_open 中的 ffurl_connect 和现在的 ffio_fdopen，可以看到 URLContext 和 AVIOContext 能够初始化的基础是 struct URLProtocol *prot，通过文件名得到 struct URLProtocol *prot，然后再通过它初始化 URLContext 和 AVIOContext。

不过在 ffio_fdopen 函数中，有一个显眼的函数 ffio_init_context，它是一个具体来初始化 AVIOContext 的函数：

注释 2-1

```
int ffio_init_context(AVIOContext *s,
                      unsigned char *buffer,
                      int buffer_size,
                      int write_flag,
                      void *opaque,
                      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))
{
    //codeInCK
    //具体的设置 AVIOContext 的各个成员
    s->buffer = buffer;
    s->buffer_size = buffer_size;
    s->buf_ptr = buffer;
    s->opaque = opaque;
    url_resetbuf(s, write_flag ? AVIO_WRONLY : AVIO_RDONLY);
    s->write_packet = write_packet;
    s->read_packet = read_packet;
    s->seek = seek;
    s->pos = 0;
    s->must_flush = 0;
    s->eof_reached = 0;
    s->error = 0;
#if FF_API_OLD_AVIO
    s->is_streamed = 0;
#endif
    s->seekable = AVIO_SEEKABLE_NORMAL;
    s->max_packet_size = 0;
    s->update_checksum = NULL;
    if(!read_packet && !write_flag)
    {
        s->pos = buffer_size;
        s->buf_end = s->buffer + buffer_size;
    }
    s->read_pause = NULL;
    s->read_seek    = NULL;
    return 0;
}
```

在 ffio_fdopen 函数中的 ffio_init_context 执行中，有意思的是传给它的几个特别的参数：ffurl_read, ffurl_write, ffurl_seek, 这三函数仍然是一个抽象接口的函数，它的具体实例化要依赖它的传入对象。

通过分析 avio_open 函数，代码在各个源代码文件中跳来跳去，但是范围是一定的，我们总结一下：

ffmpeg 库的 IO 模块，外层的接口：libavformat\avio.h（libavformat\aviobuf.c），内层接口是：libavformat\url.h（libavformat\avio.c），中间的粘合层是：libavformat\avio_internal.h（libavformat\aviobuf.c）。但是在 ffmpeg 库的 IO 模块中，它的抽象数据主要是在内层 libavformat\url.h（libavformat\avio.c）。

代码分析到这里，对 ffmpeg 库的 IO 模块，已经有了一个大致的认识，对于 IO 模块的具体数据是如何修改，如何保存等，当后面有代码涉及到数据的读取时，会回过头来进行研究。现在对于 ffmpeg 库的 IO 模块先分析到这里，继续返回我们 ffmpeg 库的 libavformat 的主干 av_open_input_file。

2.3　av_open_input_file 的第二步 av_probe_input_buffer

在本章的前面内容，从 av_open_input_file 开发，分析它的前面的一些代码，那么这里的第二是什么内容呢？记得前面曾分析过使用 av_probe_input_format 来分析输入源的媒体格式，但是这并不是一个确定会有正确结果的函数，因此就需要进一步的分析，以确定最佳的格式，这也是分析一个输入源的基础。

好了，我们的第二步就是：av_probe_input_buffer 函数，与 av_probe_input_format 仅仅只是最后一个字符串的区别。不过 av_probe_input_buffer 函数，是通过分析一段源的数据来得到结果。

```
int av_probe_input_buffer(AVIOContext *pb, AVInputFormat **fmt,
                          const char *filename, void *logctx,
                          unsigned int offset, unsigned int max_probe_size)
{
    //codeInCK
    AVProbeData pd =
    { filename ? filename : "", NULL, -offset };
    unsigned char *buf = NULL;
    int ret = 0, probe_size;

    //计算合理的 max_probe_size
    if (!max_probe_size)
    {
        max_probe_size = PROBE_BUF_MAX;
    }
    else if (max_probe_size > PROBE_BUF_MAX)
    {
        max_probe_size = PROBE_BUF_MAX;
    }
    else if (max_probe_size < PROBE_BUF_MIN)
    {
        return AVERROR(EINVAL);
    }

    if (offset >= max_probe_size)
    {
        return AVERROR(EINVAL);
    }

    for(probe_size = PROBE_BUF_MIN;
         probe_size <= max_probe_size && !*fmt && ret >= 0;
        probe_size = FFMIN(probe_size << 1, FFMAX(max_probe_size, probe_size + 1)))
    {
        int ret, score
            = probe_size < max_probe_size ? AVPROBE_SCORE_MAX / 4 : 0;
```

```
int buf_offset
      = (probe_size == PROBE_BUF_MIN) ? 0 : probe_size >> 1;

if (probe_size < offset)
{
     continue;
}

//read probe data
buf = av_realloc(buf, probe_size + AVPROBE_PADDING_SIZE);
//读取 pb 的文件流并保存在 buf 中
//注释 1
if ((ret = avio_read(pb, buf + buf_offset, probe_size - buf_offset)) < 0)
{
     // fail if error was not end of file, otherwise, lower score
     if (ret != AVERROR_EOF)
     {
          av_free(buf);
          return ret;
     }
     score = 0;
     ret = 0; // error was end of file, nothing read
}
pd.buf_size += ret;
pd.buf = &buf[offset];

memset(pd.buf + pd.buf_size, 0, AVPROBE_PADDING_SIZE);

//guess file format
*fmt = av_probe_input_format2(&pd, 1, &score);
if(*fmt)
{
     if(score <= AVPROBE_SCORE_MAX / 4)
      //this can only be true in the last iteration
     {
          av_log(logctx,
                AV_LOG_WARNING,
                "Format detected only with low score of %d,
                misdetection possible!\n",
                score);
     }
     else
          av_log(logctx,
                AV_LOG_DEBUG,
                    "Probed with size=%d and score=%d\n",
                    probe_size,
                    score);
            }
        }

        if (!*fmt)
        {
            av_free(buf);
            return AVERROR_INVALIDDATA;
        }
        //重新定义 AVIOContext *pb 的缓存
        // rewind. reuse probe buffer to avoid seeking
        if ((ret = ffio_rewind_with_probe_data(pb, buf, pd.buf_size)) < 0)
        {
            av_free(buf);
        }
        return ret;
}
```

这个函数实际上也有这清晰的思路，不断的通过 AVIOContext *pb 来读取文件保存到 buf 中，然后通过 av_probe_input_format2 获得文件的媒体格式信息。但是让我更感兴趣的地方是 AVIOContext *pb 如何来读取文件的？这个也可算作是对 ffmpeg 库 IO 模块的一个深入：int avio_read(AVIOContext *s, unsigned char *buf, int size)

注释 1：

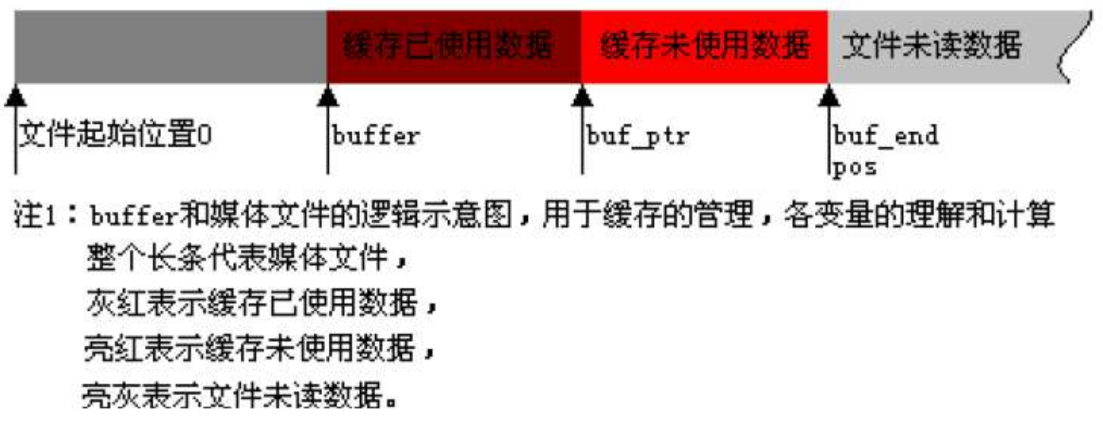
```
int avio_read(AVIOContext *s, unsigned char *buf, int size)
{
        //codeInCK
        int len, size1;

        size1 = size;
        while (size > 0)
        {
            len = s->buf_end - s->buf_ptr;
            if (len > size)
            {
                len = size;
            }
            if (len == 0)
            {
                if(size > s->buffer_size
                     && !s->update_checksum)
                {
                    if(s->read_packet)
                    {
```

```
                len = s->read_packet(s->opaque, buf, size);
             }
            if (len <= 0)
            {
                // do not modify buffer if EOF reached so that a seek back can
                //be done without rereading data
                s->eof_reached = 1;
                if(len < 0)
                 {
                    s->error = len;
                 }
                break;
            }
            else
            {
                s->pos += len;
                size -= len;
                buf += len;
                s->buf_ptr = s->buffer;
                s->buf_end = s->buffer/* + len*/;
            }
        }
        else
        {
            fill_buffer(s);
            len = s->buf_end - s->buf_ptr;
            if (len == 0)
             {
                break;
             }
        }
    }
    else
    {
        memcpy(buf, s->buf_ptr, len);
        buf += len;
        s->buf_ptr += len;
        size -= len;
    }
}
if (size1 == size)
{
    if (s->error)
     {
```

```
                return s->error;
            }
            if (url_feof(s))
            {
                return AVERROR_EOF;
            }
        }
        return size1 - size;
}
```

从这个函数中，可以看到 **AVIOContext** 里面的缓存开始起作用了。对于 **AVIOContext** 的各项操作，笔者将在下面开始展开，对 **ffmpeg** 库的 **IO** 模块进行更深入的分析，采取对整个源代码的**.c**，**.h** 文件进行分析。

实际上，介绍到这里，还是对 **AVIOContext** 里面的缓存机制不是非常透彻，那笔者借用参考资料 **1** 的图片来配合进行说明：



注1：buffer和媒体文件的逻辑示意图，用于缓存的管理，各变量的理解和计算

整个长条代表媒体文件，

灰红表示缓存已使用数据，

亮红表示缓存未使用数据，

亮灰表示文件未读数据。

AVIOContext 的缓存机制图片 2.3-1

上面的图片已经较为清楚的说明了 **AVIOContext** 里面的缓存机制，但是笔者还是在这里啰嗦几句：

其一：**AVIOContext** 每次读取文件就是一大块，读取以后保存在缓存中供上层使用。针对仅仅是读取这一大块文件，仅仅需要保存一个变量，文件现在读取到的位置：**pos.**

其二：对于保存在缓存中的数据，需要注意的变量。在申请缓存的时候要注意：**buf, buf_size**；在使用缓存的数据大小的时候，要注意已经使用的缓存的位置 **buf_ptr**， 缓存的结尾位置 **buf_end.**

其三：对于 **AVIOContext** 的缓存机制，它不仅仅应用于 **AVIOContext** 对象的读取数据，也能应用于 **AVIOContext** 对象的写入数据。

其四：**AVIOContext** 的缓存机制所处的层次？如何判断已使用和未使用？

对于 **AVIOContext** 的读、写操作就是使用状态，对于内部则是没有任何状态。

```
#include "libavutil/crc.h"
#include "libavutil/intreadwrite.h"
#include "avformat.h"
#include "avio.h"
```

```
#include "avio_internal.h"
#include "internal.h"
#include "url.h"
#include <stdarg.h>

#define IO_BUFFER_SIZE 32768
#define SHORT_SEEK_THRESHOLD 4096

static void fill_buffer(AVIOContext *s);
#if !FF_API_URL_RESETBUF
static int url_resetbuf(AVIOContext *s, int flags);
#endif

//AVIOContext 结构的初始化函数
int ffio_init_context(AVIOContext *s,
                      unsigned char *buffer,
                      int buffer_size,
                      int write_flag,
                      void *opaque,
                      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))
{
    //codeInCK
    //具体的设置 AVIOContext 的各个成员
    s->buffer = buffer;
    s->buffer_size = buffer_size;
    s->buf_ptr = buffer;
    s->opaque = opaque;
    //给 s->buf_end 赋值
    url_resetbuf(s, write_flag ? AVIO_WRONLY : AVIO_RDONLY);
    s->write_packet = write_packet;
    s->read_packet = read_packet;
    s->seek = seek;
    s->pos = 0;
    s->must_flush = 0;
    s->eof_reached = 0;
    s->error = 0;
#if FF_API_OLD_AVIO
    s->is_streamed = 0;
#endif
    s->seekable = AVIO_SEEKABLE_NORMAL;
    s->max_packet_size = 0;
    s->update_checksum = NULL;
    if(!read_packet && !write_flag)
    {
        s->pos = buffer_size;
        s->buf_end = s->buffer + buffer_size;
    }
    s->read_pause = NULL;
    s->read_seek    = NULL;
    return 0;
}

#if FF_API_OLD_AVIO
int init_put_byte(AVIOContext *s,
                  unsigned char *buffer,
                  int buffer_size,
                  int write_flag,
                  void *opaque,
                  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))
{
    return ffio_init_context(s, buffer, buffer_size, write_flag, opaque,
                             read_packet, write_packet, seek);
}
AVIOContext *av_alloc_put_byte(
    unsigned char *buffer,
    int buffer_size,
    int write_flag,
    void *opaque,
    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))
{
    return avio_alloc_context(buffer, buffer_size, write_flag, opaque,
                              read_packet, write_packet, seek);
}
#endif

//创造一个 AVIOContext，并对它进行初始化
AVIOContext *avio_alloc_context(
    unsigned char *buffer,
    int buffer_size,
    int write_flag,
    void *opaque,
    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))
{
    AVIOContext *s = av_mallocz(sizeof(AVIOContext));
    ffio_init_context(s, buffer, buffer_size, write_flag, opaque,
                                    read_packet, write_packet, seek);
    return s;
}

//将 AVIOContext 的缓存全部处理掉
static void flush_buffer(AVIOContext *s)
{
    //是否存在缓存数据
    if (s->buf_ptr > s->buffer)
    {
            //写入所有缓存
        if (s->write_packet && !s->error)
        {
                int ret = s->write_packet(s->opaque, s->buffer, s->buf_ptr - s->buffer);
                if(ret < 0)
                {
                        s->error = ret;
                }
        }
        //计算缓存数据的校验和
        if(s->update_checksum)
        {
                s->checksum = s->update_checksum(s->checksum, s->checksum_ptr, s->buf_ptr -
s->checksum_ptr);
                s->checksum_ptr = s->buffer;
        }
        s->pos += s->buf_ptr - s->buffer;
    }
    s->buf_ptr = s->buffer;
}

//对 AVIOContext 写入一个为 b 的数
void avio_w8(AVIOContext *s, int b)
{
    *(s->buf_ptr)++ = b;
    if (s->buf_ptr >= s->buf_end)
    {
        flush_buffer(s);
    }
```

```
}


//对 AVIOContext 填充 count 个为 b 的数
void ffio_fill(AVIOContext *s, int b, int count)
{
    while (count > 0)
    {
        //根据剩余的已存在的缓存数据大小与写入数据的大小，
        //计算可以写入的数据的大小
        int len = FFMIN(s->buf_end - s->buf_ptr, count);
        memset(s->buf_ptr, b, len);
        s->buf_ptr += len;
        //如果达到了缓存大小的结尾出，则写入数据
        if (s->buf_ptr >= s->buf_end)
        {
            flush_buffer(s);
        }
        count -= len;
    }
}

//对 AVIOContext 写入数据
void avio_write(AVIOContext *s, const unsigned char *buf, int size)
{
    //循环的写入数据
    while (size > 0)
    {
        int len = FFMIN(s->buf_end - s->buf_ptr, size);
        memcpy(s->buf_ptr, buf, len);
        s->buf_ptr += len;

        if (s->buf_ptr >= s->buf_end)
        {
            flush_buffer(s);
        }
        buf += len;
        size -= len;
    }
}

//对 AVIOContext 的缓存数据进行处理
void avio_flush(AVIOContext *s)
{
    flush_buffer(s);
    //将标志位置零
    s->must_flush = 0;
}

//对 AVIOContext 进行 seek 操作
int64_t avio_seek(AVIOContext *s, int64_t offset, int whence)
{
    int64_t offset1;
    int64_t pos;
    //计算标帜
    int force = whence & AVSEEK_FORCE;
    whence &= ~AVSEEK_FORCE;

    if(!s)
    {
        return AVERROR(EINVAL);
    }
    //计算位置(如果可写的话，就没有与文件相关的缓存，如果不是可写，就有)
    pos = s->pos - (s->write_flag ? 0 : (s->buf_end - s->buffer));

    if (whence != SEEK_CUR && whence != SEEK_SET)
    {
        return AVERROR(EINVAL);
    }
    if (whence == SEEK_CUR)
    {
        //如果是设置为当前位置，则当前位置就是 s->pos
        offset1 = pos + (s->buf_ptr - s->buffer);
        if (offset == 0)
        {
            return offset1;
        }
        offset += offset1;
    }
    //计算在已有的 pos 位置上，还需要偏移多少
    offset1 = offset - pos;
    if (!s->must_flush &&
            offset1 >= 0 && offset1 <= (s->buf_end - s->buffer))
    {
        //设置的 offset 刚好在缓存中
        /* can do the seek inside the buffer */
        s->buf_ptr = s->buffer + offset1;
    }
    else if ((!s->seekable
                  || offset1 <= s->buf_end + SHORT_SEEK_THRESHOLD - s->buffer)
        && !s->write_flag && offset1 >= 0
        && (whence != SEEK_END || force))
    {
        //一步步读取 IO 进行偏移
        while(s->pos < offset && !s->eof_reached)
        {
            fill_buffer(s);
        }
        if (s->eof_reached)
        {
            return AVERROR_EOF;
        }
        s->buf_ptr = s->buf_end + offset - s->pos;
    }
    else
    {
        int64_t res;

#if CONFIG_MUXERS || CONFIG_NETWORK
        if (s->write_flag)
        {
            flush_buffer(s);
            s->must_flush = 1;
        }
#endif /* CONFIG_MUXERS || CONFIG_NETWORK */
        if (!s->seek)
        {
            return AVERROR(EPIPE);
        }
        if ((res = s->seek(s->opaque, offset, SEEK_SET)) < 0)
        {
            return res;
        }
        if (!s->write_flag)
        {
            s->buf_end = s->buffer;
        }
        s->buf_ptr = s->buffer;
        s->pos = offset;
    }
    s->eof_reached = 0;
    return offset;
```

```
}

//对 AVIOContext 进行 skip 操作
int64_t avio_skip(AVIOContext *s, int64_t offset)
{
    return avio_seek(s, offset, SEEK_CUR);
}

#if FF_API_OLD_AVIO
int url_fskip(AVIOContext *s, int64_t offset)
{
    int64_t ret = avio_seek(s, offset, SEEK_CUR);
    return ret < 0 ? ret : 0;
}

int64_t url_ftell(AVIOContext *s)
{
    return avio_seek(s, 0, SEEK_CUR);
}
#endif

//获取 AVIOContext 的 IO 数据大小
int64_t avio_size(AVIOContext *s)
{
    int64_t size;

    if(!s)
    {
        return AVERROR(EINVAL);
    }
    if (!s->seek)
    {
        return AVERROR(ENOSYS);
    }
    size = s->seek(s->opaque, 0, AVSEEK_SIZE);
    if(size < 0)
    {
        if ((size = s->seek(s->opaque, -1, SEEK_END)) < 0)
        {
            return size;
        }
        size++;
        s->seek(s->opaque, s->pos, SEEK_SET);
    }
    return size;
}

//判断 AVIOContext 是否处于结尾处
int url_feof(AVIOContext *s)
{
    if(!s)
        return 0;
    if(s->eof_reached)
    {
        s->eof_reached = 0;
        fill_buffer(s);
    }
    return s->eof_reached;
}

#if FF_API_OLD_AVIO
int url_ferror(AVIOContext *s)
{
    if(!s)
        return 0;
    return s->error;
}
#endif

//对 AVIOContext 写入倒序一个 32 位数
void avio_wl32(AVIOContext *s, unsigned int val)
{
    avio_w8(s, val);
    avio_w8(s, val >> 8);
    avio_w8(s, val >> 16);
    avio_w8(s, val >> 24);
}

//对 AVIOContext 写入正序一个 32 位数
void avio_wb32(AVIOContext *s, unsigned int val)
{
    avio_w8(s, val >> 24);
    avio_w8(s, val >> 16);
    avio_w8(s, val >> 8);
    avio_w8(s, val);
}

#if FF_API_OLD_AVIO
```

```
//对 AVIOContext 写入一个字符串
void put_strz(AVIOContext *s, const char *str)
{
    avio_put_str(s, str);
}

#define GET(name, type) \
    type get_be ##name(AVIOContext *s) \
{\
    return avio_rb ##name(s);\
}\
    type get_le ##name(AVIOContext *s) \
{\
    return avio_rl ##name(s);\
}

GET(16, unsigned int)
GET(24, unsigned int)
GET(32, unsigned int)
GET(64, uint64_t)

#undef GET

#define PUT(name, type ) \
    void put_le ##name(AVIOContext *s, type val)\
{\
        avio_wl ##name(s, val);\
}\
    void put_be ##name(AVIOContext *s, type val)\
{\
        avio_wb ##name(s, val);\
}

PUT(16, unsigned int)
PUT(24, unsigned int)
PUT(32, unsigned int)
PUT(64, uint64_t)
#undef PUT

//对 AVIOContext 读取一个字节
int get_byte(AVIOContext *s)
{
    return avio_r8(s);
}
```

```
//对 AVIOContext 获取固定大小的一段数据
int get_buffer(AVIOContext *s, unsigned char *buf, int size)
{
    return avio_read(s, buf, size);
}


//获取 AVIOContext 缓存中的数据填充到 buf 中
int get_partial_buffer(AVIOContext *s, unsigned char *buf, int size)
{
    return ffio_read_partial(s, buf, size);
}

//对 AVIOContext 写入一个字节的数据 val
void put_byte(AVIOContext *s, int val)
{
    avio_w8(s, val);
}

//对 AVIOContext 写入一片数据
void put_buffer(AVIOContext *s, const unsigned char *buf, int size)
{
    avio_write(s, buf, size);
}

//对 AVIOContext 填充 count 个 b 的数据
void put_nbyte(AVIOContext *s, int b, int count)
{
    ffio_fill(s, b, count);
}

//将 AVIOContext 打开
int url_fopen(AVIOContext **s, const char *filename, int flags)
{
    return avio_open(s, filename, flags);
}

//将 AVIOContext 关闭
int url_fclose(AVIOContext *s)
{
    return avio_close(s);
}
```

```
//将 AVIOContext 设置偏移位置
int64_t url_fseek(AVIOContext *s, int64_t offset, int whence)
{
    return avio_seek(s, offset, whence);
}

//获取 AVIOContext 的 IO 数据大小
int64_t url_fsize(AVIOContext *s)
{
    return avio_size(s);
}

//对 AVIOContext 设置缓存大小
int url_setbufsize(AVIOContext *s, int buf_size)
{
    return ffio_set_buf_size(s, buf_size);
}

//对 AVIOContext 打印信息，并写入 AVIOContext 中
int url_fprintf(AVIOContext *s, const char *fmt, ...)
{
    va_list ap;
    char buf[4096];
    int ret;

    va_start(ap, fmt);
    ret = vsnprintf(buf, sizeof(buf), fmt, ap);
    va_end(ap);

    avio_write(s, buf, strlen(buf));
    return ret;
}

//对 AVIOContext 处理缓存包
void put_flush_packet(AVIOContext *s)
{
    avio_flush(s);
}

//暂停
int av_url_read_fpause(AVIOContext *s, int pause)
{
    return avio_pause(s, pause);
}
```

```
//设定时间戳
int64_t av_url_read_fseek(AVIOContext *s, int stream_index,
                          int64_t timestamp, int flags)
{
    return avio_seek_time(s, stream_index, timestamp, flags);
}


//初始化校验和函数
void init_checksum(AVIOContext *s,
                   unsigned long (*update_checksum)(unsigned long c, const uint8_t *p,
unsigned int len),
                   unsigned long checksum)
{
    ffio_init_checksum(s, update_checksum, checksum);
}

//获取校验和
unsigned long get_checksum(AVIOContext *s)
{
    return ffio_get_checksum(s);
}

//不明白？
int url_open_dyn_buf(AVIOContext **s)
{
    return avio_open_dyn_buf(s);
}

//不明白？
int url_open_dyn_packet_buf(AVIOContext **s, int max_packet_size)
{
    return ffio_open_dyn_packet_buf(s, max_packet_size);
}

//不明白？
int url_close_dyn_buf(AVIOContext *s, uint8_t **pbuffer)
{
    return avio_close_dyn_buf(s, pbuffer);
}

//打开 AVIOContext
```

```
int url_fdopen(AVIOContext **s, URLContext *h)
{
    return ffio_fdopen(s, h);
}
#endif

//对 AVIOContext 写入一个字符串
int avio_put_str(AVIOContext *s, const char *str)
{
    int len = 1;
    if (str)
    {
        len += strlen(str);
        avio_write(s, (const unsigned char *) str, len);
    }
    else
        avio_w8(s, 0);
    return len;
}

//写入 UTF 的字符串
int avio_put_str16le(AVIOContext *s, const char *str)
{
    const uint8_t *q = str;
    int ret = 0;

    while (*q)
    {
        uint32_t ch;
        uint16_t tmp;

        GET_UTF8(ch, *q++, break;)
        PUT_UTF16(ch, tmp, avio_wl16(s, tmp); ret += 2;)
    }
    avio_wl16(s, 0);
    ret += 2;
    return ret;
}

//获取数据的字节数
int ff_get_v_length(uint64_t val)
{
    int i = 1;
    while(val >>= 7)
        i++;

    return i;
}

//写入一个数 val
void ff_put_v(AVIOContext *bc, uint64_t val)
{
    int i = ff_get_v_length(val);

    while(--i > 0)
        avio_w8(bc, 128 | (val >> (7 * i)));

    avio_w8(bc, val & 127);
}

//写入 64 位数
void avio_wl64(AVIOContext *s, uint64_t val)
{
    avio_wl32(s, (uint32_t)(val & 0xffffffff));
    avio_wl32(s, (uint32_t)(val >> 32));
}

//写入 64 位数
void avio_wb64(AVIOContext *s, uint64_t val)
{
    avio_wb32(s, (uint32_t)(val >> 32));
    avio_wb32(s, (uint32_t)(val & 0xffffffff));
}

//写入 16 位数
void avio_wl16(AVIOContext *s, unsigned int val)
{
    avio_w8(s, val);
    avio_w8(s, val >> 8);
}

//写入 16 位数
void avio_wb16(AVIOContext *s, unsigned int val)
{
    avio_w8(s, val >> 8);
    avio_w8(s, val);
}
```

```
//写入 24 位数
void avio_wl24(AVIOContext *s, unsigned int val)
{
    avio_wl16(s, val & 0xffff);
    avio_w8(s, val >> 16);
}

//写入 24 位数
void avio_wb24(AVIOContext *s, unsigned int val)
{
    avio_wb16(s, val >> 8);
    avio_w8(s, val);
}

#if FF_API_OLD_AVIO
void put_tag(AVIOContext *s, const char *tag)
{
    while (*tag)
    {
        avio_w8(s, *tag++);
    }
}
#endif

//Input stream
static void fill_buffer(AVIOContext *s)
{
    //获取可以使用的 AVIOContext 的缓存
    uint8_t *dst = (!s->max_packet_size
        && ((s->buf_end - s->buffer) < s->buffer_size))
        ? s->buf_end : s->buffer;
    //获取当前 AVIOContext 的剩余缓存大小
    int len =
        s->buffer_size - (dst - s->buffer);
    //获取 AVIOContext 的缓存的最大大小
    int max_buffer_size
        = s->max_packet_size ?
        s->max_packet_size : IO_BUFFER_SIZE;
    // no need to do anything if EOF already reached
    //如果 AVIOContext 已经达到结尾处，则直接返回
    if (s->eof_reached)
    {
        return;
}
//如果 AVIOContext 的校验函数存在并且正好处于缓存的基点
if(s->update_checksum && dst == s->buffer)
{
        //如果校验和的校验位置是处于缓存的合法位置，则计算校验位
        if(s->buf_end > s->checksum_ptr)
        {
                s->checksum = s->update_checksum(s->checksum,
                 s->checksum_ptr, s->buf_end - s->checksum_ptr);
        }
        //给校验和的当前校验位赋值
        s->checksum_ptr = s->buffer;
}
//如果 read_packet 存在，并且最大的缓存大小超过当前的需要，则缩小当前的缓存
// make buffer smaller in case it ended up large after probing
if (s->read_packet && s->buffer_size > max_buffer_size)
{
        //缩小当前的缓存大小
        ffio_set_buf_size(s, max_buffer_size);
        s->checksum_ptr = dst = s->buffer;
        len = s->buffer_size;
}

//读取 len 个数据, len 是如何得到呢？
//从传入函数开始计算，实际上 len 就是 AVIOContext 的剩余缓存大小
if(s->read_packet)
{
        len = s->read_packet(s->opaque, dst, len);
}
else
{
        len = 0;
}
//如果 AVIOContext 到了结尾处
if (len <= 0)
{
        //do not modify buffer if EOF reached so that a seek back can
        //be done without rereading data
        s->eof_reached = 1;
        if(len < 0)
        {
                s->error = len;
        }
}
```

```
    //读取成功，记录当前读取的位置
    else
    {
        s->pos += len;
        s->buf_ptr = dst;
        s->buf_end = dst + len;
    }
}

//获取 CRC 的值
unsigned long ff_crc04C11DB7_update(unsigned long checksum, const uint8_t *buf,
                                    unsigned int len)
{
    return av_crc(av_crc_get_table(AV_CRC_32_IEEE), checksum, buf, len);
}

//获取校验和的值
unsigned long ffio_get_checksum(AVIOContext *s)
{
    s->checksum = s->update_checksum(s->checksum, s->checksum_ptr, s->buf_ptr -
s->checksum_ptr);
    s->update_checksum = NULL;
    return s->checksum;
}

//初始化校验和
void ffio_init_checksum(AVIOContext *s,
                        unsigned long (*update_checksum)(unsigned long c, const uint8_t
*p, unsigned int len),
                        unsigned long checksum)
{
    s->update_checksum = update_checksum;
    if(s->update_checksum)
    {
        s->checksum = checksum;
        s->checksum_ptr = s->buf_ptr;
    }
}

//对 AVIOContext 读取一个字符
/* XXX: put an inline version */
int avio_r8(AVIOContext *s)
{
    if (s->buf_ptr >= s->buf_end)
    {
        fill_buffer(s);
    }
    if (s->buf_ptr < s->buf_end)
    {
        return *s->buf_ptr++;
    }
    return 0;
}

#if FF_API_OLD_AVIO
int url_fgetc(AVIOContext *s)
{
    if (s->buf_ptr >= s->buf_end)
        fill_buffer(s);
    if (s->buf_ptr < s->buf_end)
        return *s->buf_ptr++;
    return URL_EOF;
}
#endif


//读取 AVIOContext 的数据，更确切的说应该是 avio_get_buffer，
//但是现在的名字
//可以更加的贴合抽象的各种 IO 的 read 操作
//具体的功能是读取 AVIOContext 的 size 个字节的数据，
//然后保存到 buf 中
int avio_read(AVIOContext *s, unsigned char *buf, int size)
{
    //codeInCK
    int len, size1;

    //s->buf_size 的大小是固定的
    //s->buf 也是传入之前固定的
    //保存原始的要读取的数据的大小，
    //这个时候的 size 转换为记录当前还需读取的数据大小
    size1 = size;
    //循环读取数据，直到还需读取的数据大小 size 为 0
    while (size > 0)
    {
        //在这里遇到的情况，可能如下
        //1.AVIOContext 已经存在一部分数据
        //2.AVIOContext 根本就是一个空的缓存
        //获取 AVIOContext 剩余的已存在的缓存数据的大小
```

```
len = s->buf_end - s->buf_ptr;
 //比较 AVIOContext 剩余已存在的缓存数据的大小与目前还需要读取的数据大小
 //如果大于的话，本次读取数据大小则是目前还需要读取的数据大小
 //否则，本次读取数据大小就是 AVIOContext 剩余已存在的缓存数据的大小
if (len > size)
 {
     len = size;
 }
 //如果没有 AVIOContext 剩余已存在的缓存数据的大小可以使用
if (len == 0)
{
      //如果当前要读取的数据大小比 AVIOContext 缓存总大小小的话
      //另外还必须有 s->update_checksum 为空即不需要校验和，
      //则是直接绕过 AVIOContext 缓存(即不需要)，
      //直接通过内层函数读取到目标 buf 中
     if(size > s->buffer_size
            && !s->update_checksum)
     {
          if(s->read_packet)
           {
               len = s->read_packet(s->opaque, buf, size);
           }
           //判断是否读取到结尾处
          if (len <= 0)
          {
               // do not modify buffer if EOF reached so that a seek back can
               //be done without rereading data
               s->eof_reached = 1;
               if(len < 0)
                {
                    s->error = len;
                }
               break;
          }
           //否则将进行正常的记录，记录什么呢？这是个好问题
           //记录的目标有：
           //1.AVIOContext 当前读取到位置
           //2.当前不需要 AVIOContext 的缓存，
           //所以 buf_ptr，buf_end 应该置为初始化状态
          else
          {
               s->pos += len;
               size -= len;
               buf += len;
```

```
                    s->buf_ptr = s->buffer;
                    s->buf_end = s->buffer/* + len*/;
                }
            }
             //否则，则使用 AVIOContext 的缓存，
             //那它是如何使用的呢？请看 fill_buffer
            else
            {
                 //fill_buffer 没有指定读取的数据大小，实际上
                 //它的作用就是读取数据填满剩余的缓存
                fill_buffer(s);
                 //计算读取的数据的长度
                len = s->buf_end - s->buf_ptr;
                if (len == 0)
                 {
                    break;
                 }
            }
        }
         //使用 AVIOContext 剩余已存在的缓存数据
        else
        {
            memcpy(buf, s->buf_ptr, len);
            buf += len;
            s->buf_ptr += len;
            size -= len;
        }
    }
    if (size1 == size)
    {
        if (s->error)
         {
             return s->error;
         }
         //判断 AVIOContext 是否处于结尾处
        if (url_feof(s))
         {
             return AVERROR_EOF;
         }
    }
    return size1 - size;
}
```

```
//对 AVIOContext 获取缓存中的数据，如没缓存中没有数据则读取一次数据
int ffio_read_partial(AVIOContext *s, unsigned char *buf, int size)
{
    int len;

    if(size < 0)
    {
        return -1;
    }
    len = s->buf_end - s->buf_ptr;
    //如果缓存中没有数据，则读取一次数据
    if (len == 0)
    {
        fill_buffer(s);
        len = s->buf_end - s->buf_ptr;
    }
    if (len > size)
    {
        len = size;
    }
    memcpy(buf, s->buf_ptr, len);
    s->buf_ptr += len;
    if (!len)
    {
        if (s->error)
        {
            return s->error;
        }
        if (url_feof(s))
        {
            return AVERROR_EOF;
        }
    }
    return len;
}

unsigned int avio_rl16(AVIOContext *s)
{
    unsigned int val;
    val = avio_r8(s);
    val |= avio_r8(s) << 8;
    return val;
}
```

```
unsigned int avio_rl24(AVIOContext *s)
{
    unsigned int val;
    val = avio_rl16(s);
    val |= avio_r8(s) << 16;
    return val;
}

unsigned int avio_rl32(AVIOContext *s)
{
    unsigned int val;
    val = avio_rl16(s);
    val |= avio_rl16(s) << 16;
    return val;
}

uint64_t avio_rl64(AVIOContext *s)
{
    uint64_t val;
    val = (uint64_t)avio_rl32(s);
    val |= (uint64_t)avio_rl32(s) << 32;
    return val;
}

unsigned int avio_rb16(AVIOContext *s)
{
    unsigned int val;
    val = avio_r8(s) << 8;
    val |= avio_r8(s);
    return val;
}

unsigned int avio_rb24(AVIOContext *s)
{
    unsigned int val;
    val = avio_rb16(s) << 8;
    val |= avio_r8(s);
    return val;
}
unsigned int avio_rb32(AVIOContext *s)
{
    unsigned int val;
    val = avio_rb16(s) << 16;
    val |= avio_rb16(s);
    return val;
}

#if FF_API_OLD_AVIO
char *get_strz(AVIOContext *s, char *buf, int maxlen)
{
    avio_get_str(s, INT_MAX, buf, maxlen);
    return buf;
}
#endif

int ff_get_line(AVIOContext *s, char *buf, int maxlen)
{
    int i = 0;
    char c;

    do
    {
        c = avio_r8(s);
        if (c && i < maxlen - 1)
            buf[i++] = c;
    }
    while (c != '\n' && c);

    buf[i] = 0;
    return i;
}

int avio_get_str(AVIOContext *s, int maxlen, char *buf, int buflen)
{
    int i;

    // reserve 1 byte for terminating 0
    buflen = FFMIN(buflen - 1, maxlen);
    for (i = 0; i < buflen; i++)
        if (!(buf[i] = avio_r8(s)))
            return i + 1;
    if (buflen)
        buf[i] = 0;
    for (; i < maxlen; i++)
        if (!avio_r8(s))
            return i + 1;
    return maxlen;
}
```

```
#define GET_STR16(type, read) \
    int avio_get_str16 ##type(AVIOContext *pb, int maxlen, char *buf, int buflen)\
{\
    char* q = buf;\
    int ret = 0;\
    while (ret + 1 < maxlen) {\
        uint8_t tmp;\
        uint32_t ch;\
        GET_UTF16(ch, (ret += 2) <= maxlen ? read(pb) : 0, break;)\
        if (!ch)\
            break;\
        PUT_UTF8(ch, tmp, if (q - buf < buflen - 1) *q++ = tmp;)\
    }\
    *q = 0;\
    return ret;\
}\

GET_STR16(le, avio_rl16)
GET_STR16(be, avio_rb16)

#undef GET_STR16

uint64_t avio_rb64(AVIOContext *s)
{
    uint64_t val;
    val = (uint64_t)avio_rb32(s) << 32;
    val |= (uint64_t)avio_rb32(s);
    return val;
}

uint64_t ffio_read_varlen(AVIOContext *bc)
{
    uint64_t val = 0;
    int tmp;

    do
    {
        tmp = avio_r8(bc);
        val = (val << 7) + (tmp & 127);
    }
    while(tmp & 128);
    return val;
}
```

```
int ffio_fdopen(AVIOContext **s, URLContext *h)
{
    //codeInCK
    uint8_t *buffer;
    int buffer_size, max_packet_size;

    //获取缓存的大小
    max_packet_size = h->max_packet_size;
    if (max_packet_size)
    {
        //no need to bufferize more than one packet
        buffer_size = max_packet_size;
    }
    else
    {
        buffer_size = IO_BUFFER_SIZE;
    }
    //分配缓存
    buffer = av_malloc(buffer_size);
    if (!buffer)
    {
        return AVERROR(ENOMEM);
    }
    //分配 AVIOContext
    *s = av_mallocz(sizeof(AVIOContext));
    if(!*s)
    {
        av_free(buffer);
        return AVERROR(ENOMEM);
    }

    //初始化 AVIOContext
    //注释 2-1
    if (ffio_init_context(*s,
        buffer,
        buffer_size,
        (h->flags & AVIO_WRONLY || h->flags & AVIO_RDWR),
        h,
        ffurl_read,
        ffurl_write,
        ffurl_seek) < 0)
    {
        av_free(buffer);
```

```
            av_freep(s);
            return AVERROR(EIO);
        }
#if FF_API_OLD_AVIO
        (*s)->is_streamed = h->is_streamed;
#endif
        (*s)->seekable = h->is_streamed ? 0 : AVIO_SEEKABLE_NORMAL;
        (*s)->max_packet_size = max_packet_size;
        if(h->prot)
        {
            //将协议函数设置给 AVIOContext
            (*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;
        }
        return 0;
}

//缩小 AVIOContext 的缓存大小
int ffio_set_buf_size(AVIOContext *s, int buf_size)
{
        uint8_t *buffer;
        buffer = av_malloc(buf_size);
        if (!buffer)
        {
            return AVERROR(ENOMEM);
        }
        //注意没有保存当前的缓存内容，直接释放掉
        av_free(s->buffer);
        s->buffer = buffer;
        s->buffer_size = buf_size;
        s->buf_ptr = buffer;
        //在这里，我们看到 buffer，  buf_size，  buf_ptr 都赋值了，
        //但是 buf_end 没有赋值，  url_resetbuf 函数将对它进行赋值
        url_resetbuf(s, s->write_flag ? AVIO_WRONLY : AVIO_RDONLY);
        return 0;
}

#if FF_API_URL_RESETBUF
int url_resetbuf(AVIOContext *s, int flags)
#else
static int url_resetbuf(AVIOContext *s, int flags)
#endif
```

```
{
#if FF_API_URL_RESETBUF
    if (flags & AVIO_RDWR)
        return AVERROR(EINVAL);
#else
    assert(flags == AVIO_WRONLY || flags == AVIO_RDONLY);
#endif

    if (flags & AVIO_WRONLY)
    {
         //当前的 AVIOContext 可写，则是将 buf_end 置于缓存的末尾
        s->buf_end = s->buffer + s->buffer_size;
        s->write_flag = 1;
    }
    else
    {
         //当前的 AVIOContext 不可写，则将 buf_end 置于缓存开始处
        s->buf_end = s->buffer;
        s->write_flag = 0;
    }
    return 0;
}

//通过外部一段内存和内存大小来重新分配 AVIOContext 的缓存
int ffio_rewind_with_probe_data(AVIOContext *s, unsigned char *buf, int buf_size)
{
    int64_t buffer_start;
    int buffer_size;
    int overlap, new_size, alloc_size;

    if (s->write_flag)
    {
        return AVERROR(EINVAL);
    }
    buffer_size = s->buf_end - s->buffer;
    /* the buffers must touch or overlap */
    if ((buffer_start = s->pos - buffer_size) > buf_size)
    {
        return AVERROR(EINVAL);
    }
    overlap = buf_size - buffer_start;
    new_size = buf_size + buffer_size - overlap;

    alloc_size = FFMAX(s->buffer_size, new_size);
    if (alloc_size > buf_size)
    {
        if (!(buf = av_realloc(buf, alloc_size)))
        {
            return AVERROR(ENOMEM);
        }
    }
    if (new_size > buf_size)
    {
        memcpy(buf + buf_size, s->buffer + overlap, buffer_size - overlap);
        buf_size = new_size;
    }

    av_free(s->buffer);
    s->buf_ptr = s->buffer = buf;
    s->buffer_size = alloc_size;
    s->pos = buf_size;
    s->buf_end = s->buf_ptr + buf_size;
    s->eof_reached = 0;
    s->must_flush = 0;

    return 0;
}

int avio_open(AVIOContext **s, const char *filename, int flags)
{
    //codeInCK
    URLContext *h;
    int err;
    //通过文件名打开一种协议类型
    err = ffurl_open(&h, filename, flags);//注释 1
    if (err < 0)
    {
        return err;
    }
    err = ffio_fdopen(s, h);//注释 2
    if (err < 0)
    {
        ffurl_close(h);//注释 3
        return err;
    }
    return 0;
}
```

```
//关闭 AVIOContext
int avio_close(AVIOContext *s)
{
    URLContext *h = s->opaque;
    //释放缓存
    av_free(s->buffer);
    av_free(s);
    return ffurl_close(h);
}

#if FF_API_OLD_AVIO
URLContext *url_fileno(AVIOContext *s)
{
    return s->opaque;
}
#endif

//打印信息
int avio_printf(AVIOContext *s, const char *fmt, ...)
{
    va_list ap;
    char buf[4096];
    int ret;

    va_start(ap, fmt);
    ret = vsnprintf(buf, sizeof(buf), fmt, ap);
    va_end(ap);
    avio_write(s, buf, strlen(buf));
    return ret;
}

#if FF_API_OLD_AVIO
char *url_fgets(AVIOContext *s, char *buf, int buf_size)
{
    int c;
    char *q;

    c = avio_r8(s);
    if (url_feof(s))
        return NULL;
    q = buf;
    for(;;)
    {
        if (url_feof(s) || c == '\n')
```

```
            break;
        if ((q - buf) < buf_size - 1)
            *q++ = c;
        c = avio_r8(s);
    }
    if (buf_size > 0)
        *q = '\0';
    return buf;
}

int url_fget_max_packet_size(AVIOContext *s)
{
    return s->max_packet_size;
}
#endif

int avio_pause(AVIOContext *s, int pause)
{
    if (!s->read_pause)
        return AVERROR(ENOSYS);
    return s->read_pause(s->opaque, pause);
}

int64_t avio_seek_time(AVIOContext *s, int stream_index,
                       int64_t timestamp, int flags)
{
    URLContext *h = s->opaque;
    int64_t ret;
    if (!s->read_seek)
        return AVERROR(ENOSYS);
    ret = s->read_seek(h, stream_index, timestamp, flags);
    if(ret >= 0)
    {
        int64_t pos;
        s->buf_ptr = s->buf_end; // Flush buffer
        pos = s->seek(h, 0, SEEK_CUR);
        if (pos >= 0)
            s->pos = pos;
        else if (pos != AVERROR(ENOSYS))
            ret = pos;
    }
    return ret;
}
```

```
/* buffer handling */
#if FF_API_OLD_AVIO
int url_open_buf(AVIOContext **s, uint8_t *buf, int buf_size, int flags)
{
	int ret;
	*s = av_mallocz(sizeof(AVIOContext));
	if(!*s)
		return AVERROR(ENOMEM);
	ret = ffio_init_context(*s, buf, buf_size,
				(flags & AVIO_WRONLY || flags & AVIO_RDWR),
				NULL, NULL, NULL, NULL);
	if(ret != 0)
		av_freep(s);
	return ret;
}

int url_close_buf(AVIOContext *s)
{
	avio_flush(s);
	return s->buf_ptr - s->buffer;
}
#endif

/* output in a dynamic buffer */

typedef struct DynBuffer
{
	int pos, size, allocated_size;
	uint8_t *buffer;
	int io_buffer_size;
	uint8_t io_buffer[1];
} DynBuffer;

static int dyn_buf_write(void *opaque, uint8_t *buf, int buf_size)
{
	DynBuffer *d = opaque;
	unsigned new_size, new_allocated_size;

	/* reallocate buffer if needed */
	new_size = d->pos + buf_size;
	new_allocated_size = d->allocated_size;
	if(new_size < d->pos || new_size > INT_MAX / 2)
		return -1;
	while (new_size > new_allocated_size)
```

```
        {
                if (!new_allocated_size)
                        new_allocated_size = new_size;
                else
                        new_allocated_size += new_allocated_size / 2 + 1;
        }

        if (new_allocated_size > d->allocated_size)
        {
                d->buffer = av_realloc(d->buffer, new_allocated_size);
                if(d->buffer == NULL)
                        return AVERROR(ENOMEM);
                d->allocated_size = new_allocated_size;
        }
        memcpy(d->buffer + d->pos, buf, buf_size);
        d->pos = new_size;
        if (d->pos > d->size)
                d->size = d->pos;
        return buf_size;
}

static int dyn_packet_buf_write(void *opaque, uint8_t *buf, int buf_size)
{
        unsigned char buf1[4];
        int ret;

        /* packetized write: output the header */
        AV_WB32(buf1, buf_size);
        ret = dyn_buf_write(opaque, buf1, 4);
        if(ret < 0)
                return ret;

        /* then the data */
        return dyn_buf_write(opaque, buf, buf_size);
}

static int64_t dyn_buf_seek(void *opaque, int64_t offset, int whence)
{
        DynBuffer *d = opaque;

        if (whence == SEEK_CUR)
                offset += d->pos;
        else if (whence == SEEK_END)
                offset += d->size;
```

```
    if (offset < 0 || offset > 0x7fffffffLL)
        return -1;
    d->pos = offset;
    return 0;
}

static int url_open_dyn_buf_internal(AVIOContext **s, int max_packet_size)
{
    DynBuffer *d;
    unsigned io_buffer_size = max_packet_size ? max_packet_size : 1024;

    if(sizeof(DynBuffer) + io_buffer_size < io_buffer_size)
        return -1;
    d = av_mallocz(sizeof(DynBuffer) + io_buffer_size);
    if (!d)
        return AVERROR(ENOMEM);
    d->io_buffer_size = io_buffer_size;
    *s = avio_alloc_context(d->io_buffer, d->io_buffer_size, 1, d, NULL,
                            max_packet_size ? dyn_packet_buf_write : dyn_buf_write,
                            max_packet_size ? NULL : dyn_buf_seek);
    if(!*s)
    {
        av_free(d);
        return AVERROR(ENOMEM);
    }
    (*s)->max_packet_size = max_packet_size;
    return 0;
}

int avio_open_dyn_buf(AVIOContext **s)
{
    return url_open_dyn_buf_internal(s, 0);
}

int ffio_open_dyn_packet_buf(AVIOContext **s, int max_packet_size)
{
    if (max_packet_size <= 0)
        return -1;
    return url_open_dyn_buf_internal(s, max_packet_size);
}

int avio_close_dyn_buf(AVIOContext *s, uint8_t **pbuffer)
{
    DynBuffer *d = s->opaque;
    int size;
    static const char padbuf[FF_INPUT_BUFFER_PADDING_SIZE] = {0};
    int padding = 0;

    /* don't attempt to pad fixed-size packet buffers */
    if (!s->max_packet_size)
    {
        avio_write(s, padbuf, sizeof(padbuf));
        padding = FF_INPUT_BUFFER_PADDING_SIZE;
    }

    avio_flush(s);

    *pbuffer = d->buffer;
    size = d->size;
    av_free(d);
    av_free(s);
    return size - padding;
}
```

对 aviobuf.c 的分析都在上面的代码中，对于这些分析，最好能顺着函数的一步步调用，逐步深入，最后再融合起来思考。在 aviobuf.c 的代码中，几乎所有的操作对象都是 AVIOContext，可能让人比较繁琐的是其中的有关写入字节数据的函数，但同时他们也是最简单的函数构造而成。

在分析了重要的 aviobuf.c 源代码后，笔者接下来分析 avio.c 源代码，这两者是 ffmpeg 库的 IO 模块的核心代码。

```
/* needed for usleep() */
#define _XOPEN_SOURCE 600
#include <unistd.h>
#include "libavutil/avstring.h"
#include "libavutil/opt.h"
#include "os_support.h"
#include "avformat.h"
#if CONFIG_NETWORK
#include "network.h"
#endif
#include "url.h"

#if FF_API_URL_CLASS
//get url name Logging context
//获取 URLContext 的协议的名称
static const char *urlcontext_to_name(void *ptr)
{
    URLContext *h = (URLContext *)ptr;
    if(h->prot)
    {
        return h->prot->name;
    }
    else
    {
        return "NULL";
    }
}

//对于 URLContext 的全局设置参数
static const AVOption options[] = {{NULL}};
static const AVClass urlcontext_class =
{ "URLContext", urlcontext_to_name, options, LIBAVUTIL_VERSION_INT };
#endif

static int default_interrupt_cb(void);

URLProtocol *first_protocol = NULL;
int (*url_interrupt_cb)(void) = default_interrupt_cb;

//通过 URLProtocol 获取下一个 URLProtocol
URLProtocol *av_protocol_next(URLProtocol *p)
{
    if(p)
    {
        return p->next;
    }
    else
    {
        return first_protocol;
    }
}

//获取 URLProtocol
const char *avio_enum_protocols(void **opaque, int output)
{
    URLProtocol **p = opaque;
    *p = *p ? (*p)->next : first_protocol;
    if (!*p) return NULL;
    if ((output && (*p)->url_write) || (!output && (*p)->url_read))
    {
        return (*p)->name;
    }
    return avio_enum_protocols(opaque, output);
```

```
}

//注册一个 URLProtocol
int ffurl_register_protocol(URLProtocol *protocol, int size)
{
    URLProtocol **p;
    if (size < sizeof(URLProtocol))
    {
        URLProtocol *temp = av_mallocz(sizeof(URLProtocol));
        memcpy(temp, protocol, size);
        protocol = temp;
    }
    p = &first_protocol;
    while (*p != NULL)
    {
        p = &(*p)->next;
    }
    *p = protocol;
    protocol->next = NULL;
    return 0;
}

#if FF_API_REGISTER_PROTOCOL
/* The layout of URLProtocol as of when major was bumped to 52 */
struct URLProtocol_compat
{
    const char *name;
    int (*url_open)(URLContext *h, const char *filename, int flags);
    int (*url_read)(URLContext *h, unsigned char *buf, int size);
    int (*url_write)(URLContext *h, unsigned char *buf, int size);
    int64_t (*url_seek)(URLContext *h, int64_t pos, int whence);
    int (*url_close)(URLContext *h);
    struct URLProtocol *next;
};

//注册一个 URLProtocol
int av_register_protocol(URLProtocol *protocol)
{
    return ffurl_register_protocol(protocol, sizeof(struct URLProtocol_compat));
}

//注册一个 URLProtocol
int register_protocol(URLProtocol *protocol)
{
    return ffurl_register_protocol(protocol, sizeof(struct URLProtocol_compat));
}
#endif



//通过一个 URLProtocol 初始化一个 URLContext
static int url_alloc_for_protocol (URLContext **puc,
            struct URLProtocol *up,
            const char *filename,
            int flags)
{
    //codeInCK
    URLContext *uc;
    int err;

#if CONFIG_NETWORK
    //不管什么情况都会检查一下是否初始化网络环境
    if (!ff_network_init())
        return AVERROR(EIO);
#endif
    uc = av_mallocz(sizeof(URLContext) + strlen(filename) + 1);
    if (!uc)
    {
        err = AVERROR(ENOMEM);
        goto fail;
    }
#if FF_API_URL_CLASS
    uc->av_class = &urlcontext_class;
#endif
    uc->filename = (char *) &uc[1];
    strcpy(uc->filename, filename);
    uc->prot = up;
    uc->flags = flags;
    uc->is_streamed = 0; // default = not streamed
    uc->max_packet_size = 0; //default: stream file
    if (up->priv_data_size)
    {
        //分配一个具体协议对象数据的内存块
        uc->priv_data = av_mallocz(up->priv_data_size);
        //对分配了内存块的具体协议对象数据设置默认值
        if (up->priv_data_class)
        {
            *(const AVClass **)uc->priv_data = up->priv_data_class;
            av_opt_set_defaults(uc->priv_data);
        }
    }

    *puc = uc;
    return 0;
fail:
    *puc = NULL;
#if CONFIG_NETWORK
    //关闭网络流，只有在极个别的特殊情况下
    ff_network_close();
#endif
    return err;
}

int ffurl_connect(URLContext *uc)
{
    //codeInCK
    //通过获取的协议类型数据的相关函数带来 IO 端口
    int err = uc->prot->url_open(uc, uc->filename, uc->flags);
    if (err)
    {
        return err;
    }
    uc->is_connected = 1;
    //We must be careful here as ffurl_seek() could be slow, for example for http
    if((uc->flags & (AVIO_WRONLY | AVIO_RDWR))
        || !strcmp(uc->prot->name, "file"))
    {
        if(!uc->is_streamed && ffurl_seek(uc, 0, SEEK_SET) < 0)
        {
            uc->is_streamed = 1;
        }
    }
    return 0;
}

#if FF_API_OLD_AVIO
int url_open_protocol (URLContext **puc, struct URLProtocol *up,
const char *filename, int flags)
{
    int ret;

    ret = url_alloc_for_protocol(puc, up, filename, flags);
    if (ret)
```

```
        goto fail;
    ret = ffurl_connect(*puc);
    if (!ret)
        return 0;
fail:
    ffurl_close(*puc);
    *puc = NULL;
    return ret;
}
int url_alloc(URLContext **puc, const char *filename, int flags)
{
    return ffurl_alloc(puc, filename, flags);
}
int url_connect(URLContext *uc)
{
    return ffurl_connect(uc);
}
int url_open(URLContext **puc, const char *filename, int flags)
{
    return ffurl_open(puc, filename, flags);
}
int url_read(URLContext *h, unsigned char *buf, int size)
{
    return ffurl_read(h, buf, size);
}
int url_read_complete(URLContext *h, unsigned char *buf, int size)
{
    return ffurl_read_complete(h, buf, size);
}
int url_write(URLContext *h, const unsigned char *buf, int size)
{
    return ffurl_write(h, buf, size);
}
int64_t url_seek(URLContext *h, int64_t pos, int whence)
{
    return ffurl_seek(h, pos, whence);
}
int url_close(URLContext *h)
{
    return ffurl_close(h);
}
int64_t url_filesize(URLContext *h)
{
    return ffurl_size(h);
```

```
}
int url_get_file_handle(URLContext *h)
{
    return ffurl_get_file_handle(h);
}
int url_get_max_packet_size(URLContext *h)
{
    return h->max_packet_size;
}
void url_get_filename(URLContext *h, char *buf, int buf_size)
{
    av_strlcpy(buf, h->filename, buf_size);
}
void url_set_interrupt_cb(URLInterruptCB *interrupt_cb)
{
    avio_set_interrupt_cb(interrupt_cb);
}
int av_register_protocol2(URLProtocol *protocol, int size)
{
    return ffurl_register_protocol(protocol, size);
}
#endif

#define URL_SCHEME_CHARS                                        \
    "abcdefghijklmnopqrstuvwxyz"                                \
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ"                                \
    "0123456789+-."


int ffurl_alloc(URLContext **puc, const char *filename, int flags)
{
    //codeInCK
    URLProtocol *up;
    char proto_str[128], proto_nested[128], *ptr;
    //寻找第一个特别的字符(即数字和字母除外的)所在的位置的长度
    size_t proto_len = strspn(filename, URL_SCHEME_CHARS);

    //获取协议的类型的名字(这个由 proto_len 来决定)
    if (filename[proto_len] != ':' || is_dos_path(filename))
    {
        strcpy(proto_str,
        "file");
    }
    else
    {
        av_strlcpy(proto_str,
        filename,
        FFMIN(proto_len + 1,
        sizeof(proto_str)));
    }
    av_strlcpy(proto_nested, proto_str, sizeof(proto_nested));
    if ((ptr = strchr(proto_nested, '+')))
    {
        *ptr = '\0';
    }
    //这里存在疑问？既然得到了协议的名字: proto_str，
    //但是为什么还要再增加一个：proto_nested，为了更大的精确性，
    //也许这里只是一个嵌套协议的一部分，所以这里和之前很多有类似性，
    //为了最大可能的寻找到对应的协议
    //通过遍历来寻找协议的静态变量
    up = first_protocol;
    while (up != NULL)
    {
        if (!strcmp(proto_str, up->name))
        {
            return url_alloc_for_protocol (puc, up, filename, flags);//注释 1-1-1
        }
        if (up->flags & URL_PROTOCOL_FLAG_NESTED_SCHEME
        && !strcmp(proto_nested, up->name))
        {
            return url_alloc_for_protocol (puc, up, filename, flags);
        }
        up = up->next;
    }
    *puc = NULL;
    return AVERROR(ENOENT);
}

int ffurl_open(URLContext **puc, const char *filename, int flags)
{
    //codeInCK
    int ret = ffurl_alloc(puc, filename, flags);//注释 1-1
    if (ret)
    {
        return ret;
    }
    ret = ffurl_connect(*puc);//注释 1-2
    if (!ret)
```

```
        {
                return 0;
        }
        ffurl_close(*puc);//注释 1-3
        *puc = NULL;
        return ret;
}

//这个函数该大书特书一番，很多地方都会使用到它，
//但是它就是起得是一个什么样的作用呢？
//retry_transfer_wrapper 的作用如下：
//通过传入的函数指针 transfer_func 对传入的长度为 size 的 buf 产生作用
static inline int retry_transfer_wrapper(URLContext *h,
                         unsigned char *buf,
                         int size,
                         int size_min,
                         int (*transfer_func)(URLContext *h,
                                 unsigned char *buf, int size))
{
        int ret, len;
        int fast_retries = 5;

        len = 0;
        while (len < size_min)
        {
                ret = transfer_func(h, buf + len, size - len);
                if (ret == AVERROR(EINTR))
                {
                        continue;
                }
                if (h->flags & AVIO_FLAG_NONBLOCK)
                {
                        return ret;
                }
                if (ret == AVERROR(EAGAIN))
                {
                        ret = 0;
                        if (fast_retries)
                        {
                                fast_retries--;
                        }
                        else
                        {
                                usleep(1000);
                    }
                }
                else if (ret < 1)
                {
                        return ret < 0 ? ret : len;
                }
                if (ret)
                {
                        fast_retries = FFMAX(fast_retries, 2);
                }
                len += ret;
                if (len < size && url_interrupt_cb())
                {
                        return AVERROR_EXIT;
                }
        }
        return len;
}

//读取数据 URLContext---
//ffurl_read 与 ffurl_read_complete 在于读取数据的最小大小不一样
int ffurl_read(URLContext *h, unsigned char *buf, int size)
{
        if (h->flags & AVIO_WRONLY)
                return AVERROR(EIO);
        return retry_transfer_wrapper(h, buf, size, 1, h->prot->url_read);
}

//读取数据 URLContext
int ffurl_read_complete(URLContext *h, unsigned char *buf, int size)
{
        if (h->flags & AVIO_WRONLY)
                return AVERROR(EIO);
        return retry_transfer_wrapper(h, buf, size, size, h->prot->url_read);
}

//写入数据 URLContext
int ffurl_write(URLContext *h, const unsigned char *buf, int size)
{
        if (!(h->flags & (AVIO_WRONLY | AVIO_RDWR)))
                return AVERROR(EIO);
        /* avoid sending too big packets */
        if (h->max_packet_size && size > h->max_packet_size)
                return AVERROR(EIO);
```

```
    return retry_transfer_wrapper(h, buf, size, size, h->prot->url_write);
}

//seek 的 URLContext
int64_t ffurl_seek(URLContext *h, int64_t pos, int whence)
{
    int64_t ret;

    if (!h->prot->url_seek)
        return AVERROR(ENOSYS);
    ret = h->prot->url_seek(h, pos, whence & ~AVSEEK_FORCE);
    return ret;
}

//关闭 URLContext
int ffurl_close(URLContext *h)
{
    int ret = 0;
    if (!h) return 0; // can happen when ffurl_open fails

    //通过协议对应的函数来关闭 IO 端口
    if (h->is_connected && h->prot->url_close)
    {
        ret = h->prot->url_close(h);
    }
#if CONFIG_NETWORK
    //这里有一点不可理解了，难道每次都必须关闭整个网络环境吗？
    //存有疑问，期待后面的回复？
    ff_network_close();
#endif
    if (h->prot->priv_data_size)
    {
        av_free(h->priv_data);
    }
    av_free(h);
    return ret;
}

#if FF_API_OLD_AVIO
int url_exist(const char *filename)
{
    URLContext *h;
    if (ffurl_open(&h, filename, AVIO_RDONLY) < 0)
```

```
        return 0;
    ffurl_close(h);
    return 1;
}
#endif

//检查 URLContext
int avio_check(const char *url, int flags)
{
    URLContext *h;
    int ret = ffurl_alloc(&h, url, flags);
    if (ret)
    {
        return ret;
    }
    if (h->prot->url_check)
    {
        ret = h->prot->url_check(h, flags);
    }
    else
    {
        ret = ffurl_connect(h);
        if (ret >= 0)
        {
            ret = flags;
        }
    }
    ffurl_close(h);
    return ret;
}

//获取 URLContext 的大小
int64_t ffurl_size(URLContext *h)
{
    int64_t pos, size;

    size = ffurl_seek(h, 0, AVSEEK_SIZE);
    if(size < 0)
    {
        pos = ffurl_seek(h, 0, SEEK_CUR);
        if ((size = ffurl_seek(h, -1, SEEK_END)) < 0)
            return size;
        size++;
        ffurl_seek(h, pos, SEEK_SET);
        }
        return size;
}

//获取 URLContext 的句柄
int ffurl_get_file_handle(URLContext *h)
{
        if (!h->prot->url_get_file_handle)
                return -1;
        return h->prot->url_get_file_handle(h);
}

//URLContext 的默认回调函数
static int default_interrupt_cb(void)
{
        return 0;
}

//设置中断回调函数
void avio_set_interrupt_cb(int (*interrupt_cb)(void))
{
        if (!interrupt_cb)
                interrupt_cb = default_interrupt_cb;
        url_interrupt_cb = interrupt_cb;
}

#if FF_API_OLD_AVIO
//对 URLContext 的 read 操作
int av_url_read_pause(URLContext *h, int pause)
{
        if (!h->prot->url_read_pause)
                return AVERROR(ENOSYS);
        return h->prot->url_read_pause(h, pause);
}

//对 URLContext 的 seek
int64_t av_url_read_seek(URLContext *h,
        int stream_index, int64_t timestamp, int flags)
{
        if (!h->prot->url_read_seek)
        {
                return AVERROR(ENOSYS);
        }
        return h->prot->url_read_seek(h,
            stream_index, timestamp, flags);
}
#endif
```

针对 **libavformat\avio.c** 的分析，与之前的 **libavformat\aviobuf.c** 的方式一样，在代码中注释。在这里有一个很重要的疑问 **AVIOContext** 与 **URLContext** 是如何结合起来的？我们知道 **URLContext** 是如何与 **URLProtocol** 联系起来的，这太明显了，但是 **AVIOContext** 与 **URLContext** 联系起来的方式也是非常直接，**AVIOContext** 中的成员 **void *opaque;**即是 **URLContext**，所以就这样自然的联系起来了。

分析到这里对 **ffmpeg** 库的 **IO** 模块的核心源代码都分析完毕，虽然其中还有少许地方不甚明了，但是大多数都是明白了思路。所以后面有关 **ffmpeg** 库的 **IO** 模块的源代码不会再深入的展开分析。

现在，我们回到本节的主干函数 **av_probe_input_buffer**，在之前已分析了 **avio_read** 函数，现在对 **ffio_rewind_with_probe_data** 也已经进行过分析，所以本节的分析到此结束。

2.4 av_open_input_file 重要函数 av_open_input_stream

在 2.3 节，我们已经分析完毕 av_probe_input_buffer，现在会沿着 av_open_input_file 的函数路径继续走下去，但是先不管旁根末枝，直接到达下一个最重要的函数 av_open_input_stream.

要走到 av_open_input_stream 这个函数中，必须要准备两个变量：AVIOContext *pb 和 AVInputFormat *fmt, 好在前面的 avio_open 和 av_probe_input_buffer 已经分别帮我们准备好了这两个变量。

```
//Open a media file from an IO stream. 'fmt' must be specified.
int av_open_input_stream(AVFormatContext **ic_ptr,
                         AVIOContext *pb,
                          const char *filename,
                         AVInputFormat *fmt,
                          AVFormatParameters *ap)
{
    int err;
    AVFormatContext *ic;
    AVFormatParameters default_ap;

    if(!ap)
    {
        ap = &default_ap;
        memset(ap, 0, sizeof(default_ap));
    }

    if(!ap->prealloced_context)
    {
        ic = avformat_alloc_context();
    }
    else
```

```
{
    ic = *ic_ptr;
}
if (!ic)
{
    err = AVERROR(ENOMEM);
    goto fail;
}
ic->iformat = fmt;
ic->pb = pb;
ic->duration = AV_NOPTS_VALUE;
ic->start_time = AV_NOPTS_VALUE;
av_strlcpy(ic->filename,
     filename, sizeof(ic->filename));
//allocate private data
if (fmt->priv_data_size > 0)
{
     //给具体的对象数据分配内存
    ic->priv_data = av_mallocz(fmt->priv_data_size);
    if (!ic->priv_data)
    {
        err = AVERROR(ENOMEM);
        goto fail;
    }
}
else
{
    ic->priv_data = NULL;
}
// e.g. AVFMT_NOFILE formats will not have a AVIOContext
if (ic->pb)
{
     //读取 id3v2 的数据
    ff_id3v2_read(ic, ID3v2_DEFAULT_MAGIC);
}
//read_header 这个函数是最重要的解析具体的媒体文件信息的函数
if (ic->iformat->read_header)
{
     //通过 ic->iformat->read_header(ic, ap)获取具体文件中一切有关的信息
    err = ic->iformat->read_header(ic, ap);
    if (err < 0)
     {
        goto fail;
     }
    }
    if (pb && !ic->data_offset)
    {
        //只不过是将文件的初始位置设置为 0
        ic->data_offset = avio_tell(ic->pb);
    }
#if FF_API_OLD_METADATA
    ff_metadata_demux_compat(ic);
#endif
    ic->raw_packet_buffer_remaining_size = RAW_PACKET_BUFFER_SIZE;
    *ic_ptr = ic;
    return 0;
fail:
    //失败后，释放相关变量
    if (ic)
    {
        int i;
        av_freep(&ic->priv_data);
        for(i = 0; i < ic->nb_streams; i++)
        {
            AVStream *st = ic->streams[i];
            if (st)
            {
                av_free(st->priv_data);
                av_free(st->codec->extradata);
                av_free(st->codec);
                av_free(st->info);
            }
            av_free(st);
        }
    }
    av_free(ic);
    *ic_ptr = NULL;
    return err;
}
```

本来准备大张旗鼓的将 av_open_input_stream 分析一番，但是看了它的实现以后，其重点都被隐藏在函数指针 err = ic->iformat->read_header(ic, ap);中了。对于一个媒体文件所包含的信息是很多的，利用这些信息对 AVFormatContext 进行全面的设置，现在这个设置的过程，我们没有看到，它被放在了具体的媒体文件格式的 read_header 函数中去了。

本节的内容不够充实，我们要分析的重点没有看到，但是笔者也看到了一个更大的重点，对于源代码 libavformat\utils.h, libavformat\utils.c 的理解，这两个源文件是一个什么样的地位呢？ “大门”，进入和出去 ffmpeg 库中的 libavformat 的大门，当然，后面的章节中也会看到 ffmpeg 库中的每个成员库，都有这样的“utils.h, utils.c”这样的大门。

本节的内容讲到这里，对于 av_open_input_file 函数，将会暂时告一段落，这并不代表结束，你可能会说，还有好几个函数都没有分析到呢？不用急，后面会补充的，因为后面肯定会对 ffmpeg 库中的 libavformat 库的大门 libavformat\utils.h, libavformat\utils.c 进行源代码级别的分析。

2.5　一个转码程序的输入

阅读在 2.1 节最后段落，我们知道本书所有的转码程序字眼都是仅仅指 ffmpeg.c 源代码的转码程序，一个转码程序的输入也就是 ffmpeg.c 源码码的输入。

《libavformat\utils.c》

```
#include "avformat.h"
#include "avio_internal.h"
#include "internal.h"
#include "libavcodec/internal.h"
#include "libavcodec/raw.h"
#include "libavutil/opt.h"
#include "metadata.h"
#include "id3v2.h"
#include "libavutil/avstring.h"
#include "riff.h"
#include "audiointerleave.h"
#include "url.h"
#if HAVE_TERMIOS_H
#include <sys/time.h>
#endif
#include <time.h>
#include <strings.h>
#include <stdarg.h>
#if CONFIG_NETWORK
#include "network.h"
#endif

#undef NDEBUG
#include <assert.h>

//various utility functions for use within FFmpeg
unsigned avformat_version(void)
{
    return LIBAVFORMAT_VERSION_INT;
}

const char *avformat_configuration(void)
```

```
{
        return FFMPEG_CONFIGURATION;
}

const char *avformat_license(void)
{
#define LICENSE_PREFIX "libavformat license: "
        return LICENSE_PREFIX FFMPEG_LICENSE + sizeof(LICENSE_PREFIX) - 1;
}

/* fraction handling */

/**
  * f = val + (num / den) + 0.5.
  *
  * 'num' is normalized so that it is such as 0 <= num < den.
  *
  * @param f fractional number
  * @param val integer value
  * @param num must be >= 0
  * @param den must be >= 1
  */
static void av_frac_init(AVFrac *f, int64_t val, int64_t num, int64_t den)
{
        num += (den >> 1);
        if (num >= den)
        {
                val += num / den;
                num = num % den;
        }
        f->val = val;
        f->num = num;
        f->den = den;
}

/**
  * Fractional addition to f: f = f + (incr / f->den).
  *
  * @param f fractional number
  * @param incr increment, can be positive or negative
  */
static void av_frac_add(AVFrac *f, int64_t incr)
{
        int64_t num, den;
```

```
    num = f->num + incr;
    den = f->den;
    if (num < 0)
    {
        f->val += num / den;
        num = num % den;
        if (num < 0)
        {
            num += den;
            f->val--;
        }
    }
    else if (num >= den)
    {
        f->val += num / den;
        num = num % den;
    }
    f->num = num;
}

/** head of registered input format linked list */
#if !FF_API_FIRST_FORMAT
static
#endif
AVInputFormat *first_iformat = NULL;
/** head of registered output format linked list */
#if !FF_API_FIRST_FORMAT
static
#endif
AVOutputFormat *first_oformat = NULL;

//类似的函数***_next 差不多是相同的模式，如果为空这是返回 first 成员
//否则就是 next
AVInputFormat   *av_iformat_next(AVInputFormat   *f)
{
    if(f)
    {
        return f->next;
    }
    else
    {
        return first_iformat;
    }
```

```
}


//类似的函数***_next 差不多是相同的模式，如果为空这是返回 first 成员
//否则就是 next
AVOutputFormat *av_oformat_next(AVOutputFormat *f)
{
    if(f)
    {
        return f->next;
    }
    else
    {
        return first_oformat;
    }
}

void av_register_input_format(AVInputFormat *format)
{
    AVInputFormat **p;
    p = &first_iformat;
    while (*p != NULL) p = &(*p)->next;
    *p = format;
    format->next = NULL;
}

void av_register_output_format(AVOutputFormat *format)
{
    AVOutputFormat **p;
    p = &first_oformat;
    while (*p != NULL) p = &(*p)->next;
    *p = format;
    format->next = NULL;
}

int av_match_ext(const char *filename, const char *extensions)
{
    const char *ext, *p;
    char ext1[32], *q;

    if(!filename)
        return 0;

    ext = strrchr(filename, '.');
```

```
	if (ext)
	{
		ext++;
		p = extensions;
		for(;;)
		{
			q = ext1;
			while (*p != '\0' && *p != ',' && q - ext1 < sizeof(ext1) - 1)
				*q++ = *p++;
			*q = '\0';
			if (!strcasecmp(ext1, ext))
				return 1;
			if (*p == '\0')
				break;
			p++;
		}
	}
	return 0;
}

//判断两个格式名字的匹配程度
static int match_format(const char *name, const char *names)
{
	const char *p;
	int len, namelen;

	//如果某一个名字为空，则返回
	if (!name || !names)
	{
		return 0;
	}
	namelen = strlen(name);
	//通过字符","解析出一个格式名
	while ((p = strchr(names, ',')))
	{
		//寻找要比较的两个格式名的较长名字
		len = FFMAX(p - names, namelen);
		//忽略大小写进行字符串的比较
		if (!strncasecmp(name, names, len))
		{
			return 1;
		}
		names = p + 1;
	}
```

```
    //忽略大小写进行字符串的比较
    return !strcasecmp(name, names);
}

#if FF_API_GUESS_FORMAT
AVOutputFormat *guess_format(const char *short_name, const char *filename,
                             const char *mime_type)
{
    return av_guess_format(short_name, filename, mime_type);
}
#endif

AVOutputFormat *av_guess_format(const char *short_name, const char *filename,
                                const char *mime_type)
{
    AVOutputFormat *fmt = NULL, *fmt_found;
    int score_max, score;

    /* specific test for image sequences */
#if CONFIG_IMAGE2_MUXER
    if (!short_name && filename &&
            av_filename_number_test(filename) &&
            ff_guess_image2_codec(filename) != CODEC_ID_NONE)
    {
        return av_guess_format("image2", NULL, NULL);
    }
#endif
    /* Find the proper file type. */
    fmt_found = NULL;
    score_max = 0;
    while ((fmt = av_oformat_next(fmt)))
    {
        score = 0;
        if (fmt->name && short_name && !strcmp(fmt->name, short_name))
            score += 100;
        if (fmt->mime_type && mime_type && !strcmp(fmt->mime_type, mime_type))
            score += 10;
        if (filename && fmt->extensions &&
                av_match_ext(filename, fmt->extensions))
        {
            score += 5;
        }
        if (score > score_max)
        {
            score_max = score;
            fmt_found = fmt;
        }
    }
    return fmt_found;
}

#if FF_API_GUESS_FORMAT
AVOutputFormat *guess_stream_format(const char *short_name, const char *filename,
                                    const char *mime_type)
{
    AVOutputFormat *fmt = av_guess_format(short_name, filename, mime_type);

    if (fmt)
    {
        AVOutputFormat *stream_fmt;
        char stream_format_name[64];

        snprintf(stream_format_name, sizeof(stream_format_name), "%s_stream",
fmt->name);
        stream_fmt = av_guess_format(stream_format_name, NULL, NULL);

        if (stream_fmt)
            fmt = stream_fmt;
    }

    return fmt;
}
#endif

enum CodecID av_guess_codec(AVOutputFormat *fmt, const char *short_name,
                            const char *filename, const char *mime_type, enum
AVMediaType type)
{
    if(type == AVMEDIA_TYPE_VIDEO)
    {
        enum CodecID codec_id = CODEC_ID_NONE;

#if CONFIG_IMAGE2_MUXER
        if(!strcmp(fmt->name, "image2") || !strcmp(fmt->name, "image2pipe"))
        {
            codec_id = ff_guess_image2_codec(filename);
        }
#endif
```

```
        if(codec_id == CODEC_ID_NONE)
            codec_id = fmt->video_codec;
        return codec_id;
    }
    else if(type == AVMEDIA_TYPE_AUDIO)
        return fmt->audio_codec;
    else if (type == AVMEDIA_TYPE_SUBTITLE)
        return fmt->subtitle_codec;
    else
        return CODEC_ID_NONE;
}

//通过名字寻找输入者的类型(如 file, rtp, sdp 等)
AVInputFormat *av_find_input_format(const char *short_name)
{
    AVInputFormat *fmt = NULL;
    //循环遍历的过程寻找
    while ((fmt = av_iformat_next(fmt)))
    {
        //比较字符串
        if (match_format(short_name, fmt->name))
        {
            return fmt;
        }
    }
    return NULL;
}

#if FF_API_SYMVER && CONFIG_SHARED && HAVE_SYMVER
//FF_SYMVER(void, av_destruct_packet_nofree, (AVPacket *pkt), "LIBAVFORMAT_52")
//{
//     av_destruct_packet_nofree(pkt);
//}
//
//FF_SYMVER(void, av_destruct_packet, (AVPacket *pkt), "LIBAVFORMAT_52")
//{
//     av_destruct_packet(pkt);
//}
//
//FF_SYMVER(int, av_new_packet, (AVPacket *pkt, int size), "LIBAVFORMAT_52")
//{
//     return av_new_packet(pkt, size);
//}
//
```

```
//FF_SYMVER(int, av_dup_packet, (AVPacket *pkt), "LIBAVFORMAT_52")
//{
//      return av_dup_packet(pkt);
//}
//
//FF_SYMVER(void, av_free_packet, (AVPacket *pkt), "LIBAVFORMAT_52")
//{
//      av_free_packet(pkt);
//}
//
//FF_SYMVER(void, av_init_packet, (AVPacket *pkt), "LIBAVFORMAT_52")
//{
//          av_log(NULL, AV_LOG_WARNING, "Diverting av_*_packet function calls to
libavcodec. Recompile to improve performance\n");
//      av_init_packet(pkt);
//}
#endif

int av_get_packet(AVIOContext *s, AVPacket *pkt, int size)
{
    int ret = av_new_packet(pkt, size);

    if(ret < 0)
        return ret;

    pkt->pos = avio_tell(s);

    ret = avio_read(s, pkt->data, size);
    if(ret <= 0)
        av_free_packet(pkt);
    else
        av_shrink_packet(pkt, ret);

    return ret;
}

int av_append_packet(AVIOContext *s, AVPacket *pkt, int size)
{
    int ret;
    int old_size;
    if (!pkt->size)
        return av_get_packet(s, pkt, size);
    old_size = pkt->size;
    ret = av_grow_packet(pkt, size);
    if (ret < 0)
        return ret;
    ret = avio_read(s, pkt->data + old_size, size);
    av_shrink_packet(pkt, old_size + FFMAX(ret, 0));
    return ret;
}


int av_filename_number_test(const char *filename)
{
    char buf[1024];
    return filename && (av_get_frame_filename(buf, sizeof(buf), filename, 1) >= 0);
}

//通过 AVProbeData 计算输入者的格式
AVInputFormat *av_probe_input_format3(AVProbeData *pd,
                                      int is_opened, int *score_ret)
{
    //codeInCK
    AVProbeData lpd = *pd;
    AVInputFormat *fmt1 = NULL, *fmt;
    int score, score_max = 0;
    //检查是否含有 id3v3 格式的头部信息
    if (lpd.buf_size > 10
            && ff_id3v2_match(lpd.buf, ID3v2_DEFAULT_MAGIC))
    {
        //如果包含 id3v3 格式的头部信息，则跳过该 id3v3 信息头部
        int id3len = ff_id3v2_tag_len(lpd.buf);
        if (lpd.buf_size > id3len + 16)
        {
            lpd.buf += id3len;
            lpd.buf_size -= id3len;
        }
    }
    //此处的设计思路是：
    //遍历每种文件格式(每次都是完全遍历)，选出其中匹配百分比最高的一个
    //将该文件格式的静态数据结构返回
    fmt = NULL;
    while ((fmt1 = av_iformat_next(fmt1)))
    {
        if (!is_opened == !(fmt1->flags & AVFMT_NOFILE))
            continue;
        score = 0;
        if (fmt1->read_probe)
```

```
            {
                score = fmt1->read_probe(&lpd);
                if(!score
                        && fmt1->extensions
                        && av_match_ext(lpd.filename, fmt1->extensions))
                {
                    score = 1;
                }
            }
            else if (fmt1->extensions)
            {
                if (av_match_ext(lpd.filename, fmt1->extensions))
                {
                    score = 50;
                }
            }
            //获取更高的匹配度的，设置要返回的 fmt
            if (score > score_max)
            {
                score_max = score;
                fmt = fmt1;
            }
            //如果当前媒体文件格式的匹配度和之前的有相同，则将返回的 fmt 设置为空
            else if (score == score_max)
            {
                fmt = NULL;
            }
        }
        *score_ret = score_max;
        return fmt;
}

//通过 AVProbeData 计算输入者的格式
AVInputFormat *av_probe_input_format2(AVProbeData *pd, int is_opened, int *score_max)
{
        int score_ret;
        AVInputFormat *fmt = av_probe_input_format3(pd, is_opened, &score_ret);
        if(score_ret > *score_max)
        {
            *score_max = score_ret;
            return fmt;
        }
        else
            return NULL;
```

```
}

//通过 AVProbeData 计算输入者的格式
AVInputFormat *av_probe_input_format(AVProbeData *pd, int is_opened)
{
    //codeInCK
    int score = 0;
    return av_probe_input_format2(pd, is_opened, &score);
}

static int set_codec_from_probe_data(AVFormatContext *s, AVStream *st, AVProbeData *pd)
{
    static const struct
    {
        const char *name;
        enum CodecID id;
        enum AVMediaType type;
    } fmt_id_type[] =
    {
        { "aac"      , CODEC_ID_AAC       , AVMEDIA_TYPE_AUDIO },
        { "ac3"      , CODEC_ID_AC3       , AVMEDIA_TYPE_AUDIO },
        { "dts"      , CODEC_ID_DTS       , AVMEDIA_TYPE_AUDIO },
        { "eac3"     , CODEC_ID_EAC3      , AVMEDIA_TYPE_AUDIO },
        { "h264"     , CODEC_ID_H264      , AVMEDIA_TYPE_VIDEO },
        { "m4v"      , CODEC_ID_MPEG4     , AVMEDIA_TYPE_VIDEO },
        { "mp3"      , CODEC_ID_MP3       , AVMEDIA_TYPE_AUDIO },
        { "mpegvideo", CODEC_ID_MPEG2VIDEO, AVMEDIA_TYPE_VIDEO },
        { 0 }
    };
    int score;
    AVInputFormat *fmt = av_probe_input_format3(pd, 1, &score);

    if (fmt)
    {
        int i;
        av_log(s, AV_LOG_DEBUG, "Probe with size=%d, packets=%d detected %s with
score=%d\n",
               pd->buf_size, MAX_PROBE_PACKETS - st->probe_packets, fmt->name,
score);
        for (i = 0; fmt_id_type[i].name; i++)
        {
            if (!strcmp(fmt->name, fmt_id_type[i].name))
            {
                st->codec->codec_id    = fmt_id_type[i].id;
```

```
                    st->codec->codec_type = fmt_id_type[i].type;
                    break;
                }
            }
        }
        return score;
}

/************************************************************/
/* input media file */

//Open a media file from an IO stream. 'fmt' must be specified.
int av_open_input_stream(AVFormatContext **ic_ptr,
                         AVIOContext *pb,
                          const char *filename,
                         AVInputFormat *fmt,
                          AVFormatParameters *ap)
{
        int err;
        AVFormatContext *ic;
        AVFormatParameters default_ap;

        if(!ap)
        {
            ap = &default_ap;
            memset(ap, 0, sizeof(default_ap));
        }

        if(!ap->prealloced_context)
        {
            ic = avformat_alloc_context();
        }
        else
        {
            ic = *ic_ptr;
        }
        if (!ic)
        {
            err = AVERROR(ENOMEM);
            goto fail;
        }
        ic->iformat = fmt;
        ic->pb = pb;
        ic->duration = AV_NOPTS_VALUE;
```

```
    ic->start_time = AV_NOPTS_VALUE;
    av_strlcpy(ic->filename,
        filename, sizeof(ic->filename));
    //allocate private data
    if (fmt->priv_data_size > 0)
    {
        //给具体的对象数据分配内存
        ic->priv_data = av_mallocz(fmt->priv_data_size);
        if (!ic->priv_data)
        {
            err = AVERROR(ENOMEM);
            goto fail;
        }
    }
    else
    {
        ic->priv_data = NULL;
    }
    // e.g. AVFMT_NOFILE formats will not have a AVIOContext
    if (ic->pb)
    {
        //读取 id3v2 的数据
        ff_id3v2_read(ic, ID3v2_DEFAULT_MAGIC);
    }
    //read_header 这个函数是最重要的解析具体的媒体文件信息的函数
    if (ic->iformat->read_header)
    {
        //通过 ic->iformat->read_header(ic, ap)获取具体文件中一切有关的信息
        err = ic->iformat->read_header(ic, ap);
        if (err < 0)
        {
            goto fail;
        }
    }
    if (pb && !ic->data_offset)
    {
        //只不过是将文件的初始位置设置为 0
        ic->data_offset = avio_tell(ic->pb);
    }
#if FF_API_OLD_METADATA
    ff_metadata_demux_compat(ic);
#endif
    ic->raw_packet_buffer_remaining_size = RAW_PACKET_BUFFER_SIZE;
    *ic_ptr = ic;
    return 0;
fail:
    //失败后，释放相关变量
    if (ic)
    {
        int i;
        av_freep(&ic->priv_data);
        for(i = 0; i < ic->nb_streams; i++)
        {
            AVStream *st = ic->streams[i];
            if (st)
            {
                av_free(st->priv_data);
                av_free(st->codec->extradata);
                av_free(st->codec);
                av_free(st->info);
            }
            av_free(st);
        }
    }
    av_free(ic);
    *ic_ptr = NULL;
    return err;
}

/** size of probe buffer, for guessing file type from file contents */
#define PROBE_BUF_MIN 2048
#define PROBE_BUF_MAX (1<<20)

int av_probe_input_buffer(AVIOContext *pb, AVInputFormat **fmt,
                          const char *filename, void *logctx,
                          unsigned int offset, unsigned int max_probe_size)
{
    //codeInCK
    AVProbeData pd =
    { filename ? filename : "", NULL, -offset };
    unsigned char *buf = NULL;
    int ret = 0, probe_size;

    //计算合理的 max_probe_size
    if (!max_probe_size)
    {
        max_probe_size = PROBE_BUF_MAX;
    }
```

```
else if (max_probe_size > PROBE_BUF_MAX)
{
    max_probe_size = PROBE_BUF_MAX;
}
else if (max_probe_size < PROBE_BUF_MIN)
{
    return AVERROR(EINVAL);
}

if (offset >= max_probe_size)
{
    return AVERROR(EINVAL);
}

for(probe_size = PROBE_BUF_MIN;
     probe_size <= max_probe_size && !*fmt && ret >= 0;
    probe_size = FFMIN(probe_size << 1, FFMAX(max_probe_size, probe_size + 1)))
{
    int ret, score
         = probe_size < max_probe_size ? AVPROBE_SCORE_MAX / 4 : 0;
    int buf_offset
         = (probe_size == PROBE_BUF_MIN) ? 0 : probe_size >> 1;

    if (probe_size < offset)
    {
        continue;
    }

    //read probe data
    buf = av_realloc(buf, probe_size + AVPROBE_PADDING_SIZE);
     //读取 pb 的文件流并保存在 buf 中
     //注释 1
    if ((ret = avio_read(pb, buf + buf_offset, probe_size - buf_offset)) < 0)
    {
        // fail if error was not end of file, otherwise, lower score
        if (ret != AVERROR_EOF)
        {
            av_free(buf);
            return ret;
        }
        score = 0;
        ret = 0; // error was end of file, nothing read
    }
    pd.buf_size += ret;
```

```
            pd.buf = &buf[offset];

            memset(pd.buf + pd.buf_size, 0, AVPROBE_PADDING_SIZE);

            //guess file format
            *fmt = av_probe_input_format2(&pd, 1, &score);
            if(*fmt)
            {
                  if(score <= AVPROBE_SCORE_MAX / 4)
                   //this can only be true in the last iteration
                  {
                        av_log(logctx,
                               AV_LOG_WARNING,
                               "Format detected only with low score of %d,
                               misdetection possible!\n",
                               score);
                  }
                  else
                        av_log(logctx,
                               AV_LOG_DEBUG,
                               "Probed with size=%d and score=%d\n",
                               probe_size,
                               score);
            }
      }

      if (!*fmt)
      {
            av_free(buf);
            return AVERROR_INVALIDDATA;
      }
      //重新定义 AVIOContext *pb 的缓存
      // rewind. reuse probe buffer to avoid seeking
      if ((ret = ffio_rewind_with_probe_data(pb, buf, pd.buf_size)) < 0)
      {
            av_free(buf);
      }
      return ret;
}

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

```
{
    //codeInCK
    int err;
    AVProbeData probe_data, *pd = &probe_data;
    AVIOContext *pb = NULL;
    void *logctx = ap && ap->prealloced_context ? *ic_ptr : NULL;

    pd->filename = "";
    if (filename)
    {
        pd->filename = filename;
    }
    pd->buf = NULL;
    pd->buf_size = 0;

    //如果没有指定输入者的格式，就需要通过文件名获取
    if (!fmt)
    {
        // guess format if no file can be opened
        fmt = av_probe_input_format(pd, 0);//注释 1
    }

    //Do not open file if the format does not need it. XXX: specific
    //    hack needed to handle RTSP/TCP
    if (!fmt || !(fmt->flags & AVFMT_NOFILE))
    {
        // if no file needed do not try to open one
        if ((err = avio_open(&pb, filename, AVIO_RDONLY)) < 0)
        {
            goto fail;
        }
        if (buf_size > 0)
        {
            ffio_set_buf_size(pb, buf_size);
        }
        if (!fmt && (err = av_probe_input_buffer(pb,
                                   &fmt, filename,
                                   logctx, 0,
                                   logctx ? (*ic_ptr)->probesize : 0)) < 0)
        {
            goto fail;
        }
    }
    // if still no format found, error
    if (!fmt)
    {
        err = AVERROR_INVALIDDATA;
        goto fail;
    }

    // check filename in case an image number is expected
    if (fmt->flags & AVFMT_NEEDNUMBER)
    {
        if (!av_filename_number_test(filename))
        {
            err = AVERROR_NUMEXPECTED;
            goto fail;
        }
    }
    err = av_open_input_stream(ic_ptr, pb, filename, fmt, ap);
    if (err)
    {
        goto fail;
    }
    return 0;
fail:
    av_freep(&pd->buf);
    if (pb)
    {
        avio_close(pb);
    }
    if (ap && ap->prealloced_context)
    {
        av_free(*ic_ptr);
    }
    *ic_ptr = NULL;
    return err;
}

/********************************************************/

static AVPacket *add_to_pktbuf(AVPacketList **packet_buffer, AVPacket *pkt,
                               AVPacketList **plast_pktl)
{
    AVPacketList *pktl = av_mallocz(sizeof(AVPacketList));
    if (!pktl)
        return NULL;
    if (*packet_buffer)
        (*plast_pktl)->next = pktl;
    else
        *packet_buffer = pktl;

    /* add the packet in the buffered packet list */
    *plast_pktl = pktl;
    pktl->pkt = *pkt;
    return &pktl->pkt;
}

//读取一个包
int av_read_packet(AVFormatContext *s, AVPacket *pkt)
{
    int ret, i;
    AVStream *st;
    for(;;)
    {
        AVPacketList *pktl = s->raw_packet_buffer;
        if (pktl)
        {
            *pkt = pktl->pkt;
            if(s->streams[pkt->stream_index]->request_probe <= 0)
            {
                s->raw_packet_buffer = pktl->next;
                s->raw_packet_buffer_remaining_size += pkt->size;
                av_free(pktl);
                return 0;
            }
        }
        av_init_packet(pkt);
        ret = s->iformat->read_packet(s, pkt);
        if (ret < 0)
        {
            if (!pktl || ret == AVERROR(EAGAIN))
                return ret;
            for (i = 0; i < s->nb_streams; i++)
                if(s->streams[i]->request_probe > 0)
                    s->streams[i]->request_probe = -1;
            continue;
        }
        st = s->streams[pkt->stream_index];
        switch(st->codec->codec_type)
        {
        case AVMEDIA_TYPE_VIDEO:
            if(s->video_codec_id)    st->codec->codec_id = s->video_codec_id;
            break;
        case AVMEDIA_TYPE_AUDIO:
            if(s->audio_codec_id)    st->codec->codec_id = s->audio_codec_id;
            break;
        case AVMEDIA_TYPE_SUBTITLE:
            if(s->subtitle_codec_id)st->codec->codec_id = s->subtitle_codec_id;
            break;
        }

        if(!pktl && st->request_probe <= 0)
            return ret;

        add_to_pktbuf(&s->raw_packet_buffer, pkt, &s->raw_packet_buffer_end);
        s->raw_packet_buffer_remaining_size -= pkt->size;

        if(st->request_probe > 0)
        {
            AVProbeData *pd = &st->probe_data;
            int end;
            av_log(s, AV_LOG_DEBUG, "probing stream %d pp:%d\n", st->index,
st->probe_packets);
            --st->probe_packets;

            pd->buf = av_realloc(pd->buf, pd->buf_size + pkt->size +
AVPROBE_PADDING_SIZE);
            memcpy(pd->buf + pd->buf_size, pkt->data, pkt->size);
            pd->buf_size += pkt->size;
            memset(pd->buf + pd->buf_size, 0, AVPROBE_PADDING_SIZE);

            end =    s->raw_packet_buffer_remaining_size <= 0
                     || st->probe_packets <= 0;

            if(end || av_log2(pd->buf_size) != av_log2(pd->buf_size - pkt->size))
            {
                int score = set_codec_from_probe_data(s, st, pd);
                if(    (st->codec->codec_id != CODEC_ID_NONE && score >
AVPROBE_SCORE_MAX / 4)
                        || end)
                {
                    pd->buf_size = 0;
                    av_freep(&pd->buf);
                    st->request_probe = -1;
                    if(st->codec->codec_id != CODEC_ID_NONE)
                    {
                        av_log(s, AV_LOG_DEBUG, "probed stream %d\n", st->index);
                    }
                    else
                        av_log(s,  AV_LOG_WARNING,  "probed  stream  %d  failed\n",
st->index);
                }
            }
        }
    }
}

/**********************************************************/

/**
 * Get the number of samples of an audio frame. Return -1 on error.
 */
static int get_audio_frame_size(AVCodecContext *enc, int size)
{
    int frame_size;

    if(enc->codec_id == CODEC_ID_VORBIS)
        return -1;

    if (enc->frame_size <= 1)
    {
        int bits_per_sample = av_get_bits_per_sample(enc->codec_id);

        if (bits_per_sample)
        {
            if (enc->channels == 0)
                return -1;
            frame_size = (size << 3) / (bits_per_sample * enc->channels);
        }
        else
        {
            /* used for example by ADPCM codecs */
            if (enc->bit_rate == 0)
                return -1;
            frame_size = ((int64_t)size * 8 * enc->sample_rate) / enc->bit_rate;
        }
    }
    else
    {
        frame_size = enc->frame_size;
    }
    return frame_size;
}


/**
  * Return the frame duration in seconds. Return 0 if not available.
  */
static void compute_frame_duration(int *pnum, int *pden, AVStream *st,
                                   AVCodecParserContext *pc, AVPacket *pkt)
{
    int frame_size;

    *pnum = 0;
    *pden = 0;
    switch(st->codec->codec_type)
    {
    case AVMEDIA_TYPE_VIDEO:
        if(st->time_base.num * 1000LL > st->time_base.den)
        {
            *pnum = st->time_base.num;
            *pden = st->time_base.den;
        }
        else if(st->codec->time_base.num * 1000LL > st->codec->time_base.den)
        {
            *pnum = st->codec->time_base.num;
            *pden = st->codec->time_base.den;
            if (pc && pc->repeat_pict)
            {
                *pnum = (*pnum) * (1 + pc->repeat_pict);
            }
            //If this codec can be interlaced or progressive then we need a parser to compute
duration of a packet
            //Thus if we have no parser in such case leave duration undefined.
            if(st->codec->ticks_per_frame > 1 && !pc)
            {
                *pnum = *pden = 0;
            }
        }
        break;
    case AVMEDIA_TYPE_AUDIO:
        frame_size = get_audio_frame_size(st->codec, pkt->size);
        if (frame_size <= 0 || st->codec->sample_rate <= 0)
            break;
        *pnum = frame_size;
        *pden = st->codec->sample_rate;
        break;
    default:
        break;
    }
}

static int is_intra_only(AVCodecContext *enc)
{
    if(enc->codec_type == AVMEDIA_TYPE_AUDIO)
    {
        return 1;
    }
    else if(enc->codec_type == AVMEDIA_TYPE_VIDEO)
    {
        switch(enc->codec_id)
        {
        case CODEC_ID_MJPEG:
        case CODEC_ID_MJPEGB:
        case CODEC_ID_LJPEG:
        case CODEC_ID_RAWVIDEO:
        case CODEC_ID_DVVIDEO:
        case CODEC_ID_HUFFYUV:
        case CODEC_ID_FFVHUFF:
        case CODEC_ID_ASV1:
        case CODEC_ID_ASV2:
        case CODEC_ID_VCR1:
        case CODEC_ID_DNXHD:
        case CODEC_ID_JPEG2000:
            return 1;
        default:
            break;
        }
    }
    return 0;
}

static void update_initial_timestamps(AVFormatContext *s, int stream_index,
                                                    int64_t dts, int64_t pts)
```

```
{
    AVStream *st = s->streams[stream_index];
    AVPacketList *pktl = s->packet_buffer;

    if(st->first_dts != AV_NOPTS_VALUE || dts == AV_NOPTS_VALUE || st->cur_dts ==
AV_NOPTS_VALUE)
        return;

    st->first_dts = dts - st->cur_dts;
    st->cur_dts = dts;

    for(; pktl; pktl = pktl->next)
    {
        if(pktl->pkt.stream_index != stream_index)
            continue;
        //FIXME think more about this check
        if(pktl->pkt.pts != AV_NOPTS_VALUE && pktl->pkt.pts == pktl->pkt.dts)
            pktl->pkt.pts += st->first_dts;

        if(pktl->pkt.dts != AV_NOPTS_VALUE)
            pktl->pkt.dts += st->first_dts;

        if(st->start_time == AV_NOPTS_VALUE && pktl->pkt.pts != AV_NOPTS_VALUE)
            st->start_time = pktl->pkt.pts;
    }
    if (st->start_time == AV_NOPTS_VALUE)
        st->start_time = pts;
}

static void update_initial_durations(AVFormatContext *s, AVStream *st, AVPacket *pkt)
{
    AVPacketList *pktl = s->packet_buffer;
    int64_t cur_dts = 0;

    if(st->first_dts != AV_NOPTS_VALUE)
    {
        cur_dts = st->first_dts;
        for(; pktl; pktl = pktl->next)
        {
            if(pktl->pkt.stream_index == pkt->stream_index)
            {
                if(pktl->pkt.pts != pktl->pkt.dts || pktl->pkt.dts != AV_NOPTS_VALUE ||
pktl->pkt.duration)
                    break;
                cur_dts -= pkt->duration;
            }
        }
        pktl = s->packet_buffer;
        st->first_dts = cur_dts;
    }
    else if(st->cur_dts)
        return;

    for(; pktl; pktl = pktl->next)
    {
        if(pktl->pkt.stream_index != pkt->stream_index)
            continue;
        if(pktl->pkt.pts == pktl->pkt.dts && pktl->pkt.dts == AV_NOPTS_VALUE
                && !pktl->pkt.duration)
        {
            pktl->pkt.dts = cur_dts;
            if(!st->codec->has_b_frames)
                pktl->pkt.pts = cur_dts;
            cur_dts += pkt->duration;
            pktl->pkt.duration = pkt->duration;
        }
        else
            break;
    }
    if(st->first_dts == AV_NOPTS_VALUE)
        st->cur_dts = cur_dts;
}

static void compute_pkt_fields(AVFormatContext *s, AVStream *st,
                               AVCodecParserContext *pc, AVPacket *pkt)
{
    int num, den, presentation_delayed, delay, i;
    int64_t offset;

    if (s->flags & AVFMT_FLAG_NOFILLIN)
        return;

    if((s->flags & AVFMT_FLAG_IGNDTS) && pkt->pts != AV_NOPTS_VALUE)
        pkt->dts = AV_NOPTS_VALUE;

    if (st->codec->codec_id != CODEC_ID_H264 && pc && pc->pict_type == FF_B_TYPE)
        //FIXME Set low_delay = 0 when has_b_frames = 1
        st->codec->has_b_frames = 1;
    /* do we have a video B-frame ? */
    delay = st->codec->has_b_frames;
    presentation_delayed = 0;

    // ignore delay caused by frame threading so that the mpeg2-without-dts
    // warning will not trigger
    if (delay && st->codec->active_thread_type & FF_THREAD_FRAME)
        delay -= st->codec->thread_count - 1;

    /* XXX: need has_b_frame, but cannot get it if the codec is
        not initialized */
    if (delay &&
            pc && pc->pict_type != FF_B_TYPE)
        presentation_delayed = 1;

    if(pkt->pts != AV_NOPTS_VALUE && pkt->dts != AV_NOPTS_VALUE && pkt->dts >
pkt->pts && st->pts_wrap_bits < 63
            /*&& pkt->dts-(1LL<<st->pts_wrap_bits) < pkt->pts*/)
    {
        pkt->dts -= 1LL << st->pts_wrap_bits;
    }

    // some mpeg2 in mpeg-ps lack dts (issue171 / input_file.mpg)
    // we take the conservative approach and discard both
    // Note, if this is misbehaving for a H.264 file then possibly presentation_delayed is not set
correctly.
    if(delay == 1 && pkt->dts == pkt->pts && pkt->dts != AV_NOPTS_VALUE &&
presentation_delayed)
    {
        av_log(s, AV_LOG_DEBUG, "invalid dts/pts combination\n");
        pkt->dts = pkt->pts = AV_NOPTS_VALUE;
    }

    if (pkt->duration == 0)
    {
        compute_frame_duration(&num, &den, st, pc, pkt);
        if (den && num)
        {
            pkt->duration = av_rescale_rnd(1, num * (int64_t)st->time_base.den, den *
(int64_t)st->time_base.num, AV_ROUND_DOWN);

            if(pkt->duration != 0 && s->packet_buffer)
                update_initial_durations(s, st, pkt);
        }
    }

    /* correct timestamps with byte offset if demuxers only have timestamps
       on packet boundaries */
    if(pc && st->need_parsing == AVSTREAM_PARSE_TIMESTAMPS && pkt->size)
    {
        /* this will estimate bitrate based on this frame's duration and size */
        offset = av_rescale(pc->offset, pkt->duration, pkt->size);
        if(pkt->pts != AV_NOPTS_VALUE)
            pkt->pts += offset;
        if(pkt->dts != AV_NOPTS_VALUE)
            pkt->dts += offset;
    }

    if (pc && pc->dts_sync_point >= 0)
    {
        // we have synchronization info from the parser
        int64_t den = st->codec->time_base.den * (int64_t) st->time_base.num;
        if (den > 0)
        {
            int64_t num = st->codec->time_base.num * (int64_t) st->time_base.den;
            if (pkt->dts != AV_NOPTS_VALUE)
            {
                // got DTS from the stream, update reference timestamp
                st->reference_dts = pkt->dts - pc->dts_ref_dts_delta * num / den;
                pkt->pts = pkt->dts + pc->pts_dts_delta * num / den;
            }
            else if (st->reference_dts != AV_NOPTS_VALUE)
            {
                // compute DTS based on reference timestamp
                pkt->dts = st->reference_dts + pc->dts_ref_dts_delta * num / den;
                pkt->pts = pkt->dts + pc->pts_dts_delta * num / den;
            }
            if (pc->dts_sync_point > 0)
                st->reference_dts = pkt->dts; // new reference
        }
    }

    /* This may be redundant, but it should not hurt. */
    if(pkt->dts != AV_NOPTS_VALUE && pkt->pts != AV_NOPTS_VALUE && pkt->pts >
pkt->dts)
        presentation_delayed = 1;
```

```
//          av_log(NULL, AV_LOG_DEBUG, "IN delayed:%d pts:%"PRId64", dts:%"PRId64" cur_dts:%"PRId64" st:%d pc:%p\n", presentation_delayed, pkt->pts, pkt->dts, st->cur_dts, pkt->stream_index, pc);
    /* interpolate PTS and DTS if they are not present */
    //We skip H264 currently because delay and has_b_frames are not reliably set
    if((delay == 0 || (delay == 1 && pc)) && st->codec->codec_id != CODEC_ID_H264)
    {
        if (presentation_delayed)
        {
            /* DTS = decompression timestamp */
            /* PTS = presentation timestamp */
            if (pkt->dts == AV_NOPTS_VALUE)
                pkt->dts = st->last_IP_pts;
            update_initial_timestamps(s, pkt->stream_index, pkt->dts, pkt->pts);
            if (pkt->dts == AV_NOPTS_VALUE)
                pkt->dts = st->cur_dts;

            /* this is tricky: the dts must be incremented by the duration
            of the frame we are displaying, i.e. the last I- or P-frame */
            if (st->last_IP_duration == 0)
                st->last_IP_duration = pkt->duration;
            if(pkt->dts != AV_NOPTS_VALUE)
                st->cur_dts = pkt->dts + st->last_IP_duration;
            st->last_IP_duration  = pkt->duration;
            st->last_IP_pts = pkt->pts;
            /* cannot compute PTS if not present (we can compute it only
            by knowing the future */
        }
        else if(pkt->pts != AV_NOPTS_VALUE || pkt->dts != AV_NOPTS_VALUE || pkt->duration)
        {
            if(pkt->pts != AV_NOPTS_VALUE && pkt->duration)
            {
                int64_t old_diff = FFABS(st->cur_dts - pkt->duration - pkt->pts);
                int64_t new_diff = FFABS(st->cur_dts - pkt->pts);
                if(old_diff < new_diff && old_diff < (pkt->duration >> 3))
                {
                    pkt->pts += pkt->duration;
                    //                    av_log(NULL, AV_LOG_DEBUG, "id:%d old:%"PRId64" new:%"PRId64" dur:%d cur:%"PRId64" size:%d\n", pkt->stream_index, old_diff, new_diff, pkt->duration, st->cur_dts, pkt->size);
                }
            }
```

```
                /* presentation is not delayed : PTS and DTS are the same */
                if(pkt->pts == AV_NOPTS_VALUE)
                    pkt->pts = pkt->dts;
                update_initial_timestamps(s, pkt->stream_index, pkt->pts, pkt->pts);
                if(pkt->pts == AV_NOPTS_VALUE)
                    pkt->pts = st->cur_dts;
                pkt->dts = pkt->pts;
                if(pkt->pts != AV_NOPTS_VALUE)
                    st->cur_dts = pkt->pts + pkt->duration;
            }
        }

        if(pkt->pts != AV_NOPTS_VALUE && delay <= MAX_REORDER_DELAY)
        {
            st->pts_buffer[0] = pkt->pts;
            for(i = 0; i < delay && st->pts_buffer[i] > st->pts_buffer[i+1]; i++)
                FFSWAP(int64_t, st->pts_buffer[i], st->pts_buffer[i+1]);
            if(pkt->dts == AV_NOPTS_VALUE)
                pkt->dts = st->pts_buffer[0];
            if(st->codec->codec_id == CODEC_ID_H264)    //we skiped it above so we try here
            {
                update_initial_timestamps(s, pkt->stream_index, pkt->dts, pkt->pts); // this
    should happen on the first packet
            }
            if(pkt->dts > st->cur_dts)
                st->cur_dts = pkt->dts;
        }

        //          av_log(NULL, AV_LOG_ERROR, "OUTdelayed:%d/%d pts:%"PRId64",
    dts:%"PRId64" cur_dts:%"PRId64"\n", presentation_delayed, delay, pkt->pts, pkt->dts,
    st->cur_dts);

        /* update flags */
        if(is_intra_only(st->codec))
            pkt->flags |= AV_PKT_FLAG_KEY;
        else if (pc)
        {
            pkt->flags = 0;
            /* keyframe computation */
            if (pc->key_frame == 1)
                pkt->flags |= AV_PKT_FLAG_KEY;
            else if (pc->key_frame == -1 && pc->pict_type == FF_I_TYPE)
                pkt->flags |= AV_PKT_FLAG_KEY;
        }
        if (pc)
            pkt->convergence_duration = pc->convergence_duration;
    }


    static int av_read_frame_internal(AVFormatContext *s, AVPacket *pkt)
    {
        AVStream *st;
        int len, ret, i;

        av_init_packet(pkt);

        for(;;)
        {
            /* select current input stream component */
            st = s->cur_st;
            if (st)
            {
                if (!st->need_parsing || !st->parser)
                {
                    /* no parsing needed: we just output the packet as is */
                    /* raw data support */
                    *pkt = st->cur_pkt;
                    st->cur_pkt.data = NULL;
                    compute_pkt_fields(s, st, NULL, pkt);
                    s->cur_st = NULL;
                    if ((s->iformat->flags & AVFMT_GENERIC_INDEX) &&
                            (pkt->flags    &    AV_PKT_FLAG_KEY)    &&    pkt->dts    !=
    AV_NOPTS_VALUE)
                    {
                        ff_reduce_index(s, st->index);
                        av_add_index_entry(st,      pkt->pos,      pkt->dts,      0,      0,
    AVINDEX_KEYFRAME);
                    }
                    break;
                }
                else if (st->cur_len > 0 && st->discard < AVDISCARD_ALL)
                {
                    len = av_parser_parse2(st->parser, st->codec, &pkt->data, &pkt->size,
                                           st->cur_ptr, st->cur_len,
                                           st->cur_pkt.pts, st->cur_pkt.dts,
                                           st->cur_pkt.pos);
                    st->cur_pkt.pts = AV_NOPTS_VALUE;
                    st->cur_pkt.dts = AV_NOPTS_VALUE;
                    /* increment read pointer */
                    st->cur_ptr += len;
                    st->cur_len -= len;

                    /* return packet if any */
                    if (pkt->size)
                    {
    got_packet:
                        pkt->duration = 0;
                        pkt->stream_index = st->index;
                        pkt->pts = st->parser->pts;
                        pkt->dts = st->parser->dts;
                        pkt->pos = st->parser->pos;
                        if(pkt->data == st->cur_pkt.data && pkt->size == st->cur_pkt.size)
                        {
                            s->cur_st = NULL;
                            pkt->destruct = st->cur_pkt.destruct;
                            st->cur_pkt.destruct = NULL;
                            st->cur_pkt.data     = NULL;
                            assert(st->cur_len == 0);
                        }
                        else
                        {
                            pkt->destruct = NULL;
                        }
                        compute_pkt_fields(s, st, st->parser, pkt);

                        if((s->iformat->flags & AVFMT_GENERIC_INDEX) && pkt->flags &
    AV_PKT_FLAG_KEY)
                        {
                            ff_reduce_index(s, st->index);
                            av_add_index_entry(st, st->parser->frame_offset, pkt->dts,
                                               0, 0, AVINDEX_KEYFRAME);
                        }

                        break;
                    }
                }
                else
                {
                    /* free packet */
                    av_free_packet(&st->cur_pkt);
                    s->cur_st = NULL;
                }
```

```
        }
        else
        {
            AVPacket cur_pkt;
            /* read next packet */
            ret = av_read_packet(s, &cur_pkt);
            if (ret < 0)
            {
                if (ret == AVERROR(EAGAIN))
                    return ret;
                /* return the last frames, if any */
                for(i = 0; i < s->nb_streams; i++)
                {
                    st = s->streams[i];
                    if (st->parser && st->need_parsing)
                    {
                        av_parser_parse2(st->parser, st->codec,
                                         &pkt->data, &pkt->size,
                                         NULL, 0,
                                         AV_NOPTS_VALUE, AV_NOPTS_VALUE,
                                         AV_NOPTS_VALUE);
                        if (pkt->size)
                            goto got_packet;
                    }
                }
                /* no more packets: really terminate parsing */
                return ret;
            }
            st = s->streams[cur_pkt.stream_index];
            st->cur_pkt = cur_pkt;

            if(st->cur_pkt.pts != AV_NOPTS_VALUE &&
                    st->cur_pkt.dts != AV_NOPTS_VALUE &&
                    st->cur_pkt.pts < st->cur_pkt.dts)
            {
                av_log(s,    AV_LOG_WARNING,    "Invalid    timestamps    stream=%d,
pts=%"PRId64", dts=%"PRId64", size=%d\n",
                       st->cur_pkt.stream_index,
                       st->cur_pkt.pts,
                       st->cur_pkt.dts,
                       st->cur_pkt.size);
    //                  av_free_packet(&st->cur_pkt);
    //                  return -1;
            }
```

```
                if(s->debug & FF_FDEBUG_TS)
                    av_log(s,      AV_LOG_DEBUG,      "av_read_packet      stream=%d,
pts=%"PRId64", dts=%"PRId64", size=%d, duration=%d, flags=%d\n",
                           st->cur_pkt.stream_index,
                           st->cur_pkt.pts,
                           st->cur_pkt.dts,
                           st->cur_pkt.size,
                           st->cur_pkt.duration,
                           st->cur_pkt.flags);

                s->cur_st = st;
                st->cur_ptr = st->cur_pkt.data;
                st->cur_len = st->cur_pkt.size;
                if     (st->need_parsing     &&     !st->parser     &&     !(s->flags     &
AVFMT_FLAG_NOPARSE))
                {
                    st->parser = av_parser_init(st->codec->codec_id);
                    if (!st->parser)
                    {
                        /* no parser available: just output the raw packets */
                        st->need_parsing = AVSTREAM_PARSE_NONE;
                    }
                    else if(st->need_parsing == AVSTREAM_PARSE_HEADERS)
                    {
                        st->parser->flags |= PARSER_FLAG_COMPLETE_FRAMES;
                    }
                    else if(st->need_parsing == AVSTREAM_PARSE_FULL_ONCE)
                    {
                        st->parser->flags |= PARSER_FLAG_ONCE;
                    }
                }
            }
        }
        if(s->debug & FF_FDEBUG_TS)
            av_log(s, AV_LOG_DEBUG,
            "av_read_frame_internal stream=%d,
            pts=%"PRId64", dts=%"PRId64",
            size=%d, duration=%d, flags=%d\n",
                   pkt->stream_index,
                   pkt->pts,
                   pkt->dts,
                   pkt->size,
                   pkt->duration,
                   pkt->flags);

        return 0;
}

//读取一个帧
int av_read_frame(AVFormatContext *s, AVPacket *pkt)
{
        AVPacketList *pktl;
        int eof = 0;
        const int genpts = s->flags & AVFMT_FLAG_GENPTS;

        for(;;)
        {
            pktl = s->packet_buffer;
            if (pktl)
            {
                AVPacket *next_pkt = &pktl->pkt;
                if(genpts
                     && next_pkt->dts != AV_NOPTS_VALUE)
                {
                    int wrap_bits
                         = s->streams[next_pkt->stream_index]->pts_wrap_bits;
                    while(pktl
                         && next_pkt->pts == AV_NOPTS_VALUE)
                    {
                        if(pktl->pkt.stream_index == next_pkt->stream_index
                             && (0 > av_compare_mod(next_pkt->dts,
                                 pktl->pkt.dts, 2LL << (wrap_bits - 1)))
                             && av_compare_mod(pktl->pkt.pts,
                                 pktl->pkt.dts, 2LL << (wrap_bits - 1)))    //not b frame
                        {
                            next_pkt->pts = pktl->pkt.dts;
                        }
                        pktl = pktl->next;
                    }
                    pktl = s->packet_buffer;
                }

                if(next_pkt->pts != AV_NOPTS_VALUE
                     || next_pkt->dts == AV_NOPTS_VALUE
                    || !genpts
                     || eof)
                {
                    // read packet from packet buffer, if there is data
                    *pkt = *next_pkt;
                    s->packet_buffer = pktl->next;
                    av_free(pktl);
                    return 0;
                }
            }
            if(genpts)
            {
                int ret = av_read_frame_internal(s, pkt);
                if(ret < 0)
                {
                    if(pktl && ret != AVERROR(EAGAIN))
                    {
                        eof = 1;
                        continue;
                    }
                    else
                     {
                        return ret;
                     }
                }

                if(av_dup_packet(add_to_pktbuf(
                     &s->packet_buffer, pkt,
                     &s->packet_buffer_end)) < 0)
                 {
                    return AVERROR(ENOMEM);
                 }
            }
            else
            {
                assert(!s->packet_buffer);
                return av_read_frame_internal(s, pkt);
            }
        }
}

/* XXX: suppress the packet queue */
static void flush_packet_queue(AVFormatContext *s)
{
        AVPacketList *pktl;

        for(;;)
        {
            pktl = s->packet_buffer;
            if (!pktl)
                break;
            s->packet_buffer = pktl->next;
            av_free_packet(&pktl->pkt);
            av_free(pktl);
        }
        while(s->raw_packet_buffer)
        {
            pktl = s->raw_packet_buffer;
            s->raw_packet_buffer = pktl->next;
            av_free_packet(&pktl->pkt);
            av_free(pktl);
        }
        s->packet_buffer_end =
            s->raw_packet_buffer_end = NULL;
        s->raw_packet_buffer_remaining_size = RAW_PACKET_BUFFER_SIZE;
}

/*******************************************************/
/* seek support */

int av_find_default_stream_index(AVFormatContext *s)
{
        int first_audio_index = -1;
        int i;
        AVStream *st;

        if (s->nb_streams <= 0)
            return -1;
        for(i = 0; i < s->nb_streams; i++)
        {
            st = s->streams[i];
            if (st->codec->codec_type == AVMEDIA_TYPE_VIDEO)
            {
                return i;
            }
            if (first_audio_index < 0 && st->codec->codec_type == AVMEDIA_TYPE_AUDIO)
                first_audio_index = i;
        }
        return first_audio_index >= 0 ? first_audio_index : 0;
}
```

```
/**
  * Flush the frame reader.
  */
void ff_read_frame_flush(AVFormatContext *s)
{
        AVStream *st;
        int i, j;

        flush_packet_queue(s);

        s->cur_st = NULL;

        /* for each stream, reset read state */
        for(i = 0; i < s->nb_streams; i++)
        {
                st = s->streams[i];

                if (st->parser)
                {
                        av_parser_close(st->parser);
                        st->parser = NULL;
                        av_free_packet(&st->cur_pkt);
                }
                st->last_IP_pts = AV_NOPTS_VALUE;
                st->cur_dts = AV_NOPTS_VALUE; /* we set the current DTS to an unspecified origin
*/
                st->reference_dts = AV_NOPTS_VALUE;
                /* fail safe */
                st->cur_ptr = NULL;
                st->cur_len = 0;

                st->probe_packets = MAX_PROBE_PACKETS;

                for(j = 0; j < MAX_REORDER_DELAY + 1; j++)
                        st->pts_buffer[j] = AV_NOPTS_VALUE;
        }
}

void av_update_cur_dts(AVFormatContext *s, AVStream *ref_st, int64_t timestamp)
{
        int i;

        for(i = 0; i < s->nb_streams; i++)
        {
```

```
        AVStream *st = s->streams[i];

        st->cur_dts = av_rescale(timestamp,
                                 st->time_base.den * (int64_t)ref_st->time_base.num,
                                 st->time_base.num * (int64_t)ref_st->time_base.den);
    }
}

void ff_reduce_index(AVFormatContext *s, int stream_index)
{
    AVStream *st = s->streams[stream_index];
    unsigned int max_entries = s->max_index_size / sizeof(AVIndexEntry);

    if((unsigned)st->nb_index_entries >= max_entries)
    {
        int i;
        for(i = 0; 2 * i < st->nb_index_entries; i++)
            st->index_entries[i] = st->index_entries[2*i];
        st->nb_index_entries = i;
    }
}

int ff_add_index_entry(AVIndexEntry **index_entries,
                       int *nb_index_entries,
                       unsigned int *index_entries_allocated_size,
                       int64_t pos, int64_t timestamp, int size, int distance, int flags)
{
    AVIndexEntry *entries, *ie;
    int index;

    if((unsigned)*nb_index_entries + 1 >= UINT_MAX / sizeof(AVIndexEntry))
        return -1;

    entries = av_fast_realloc(*index_entries,
                              index_entries_allocated_size,
                              (*nb_index_entries + 1) *
                              sizeof(AVIndexEntry));
    if(!entries)
        return -1;

    *index_entries = entries;

    index = ff_index_search_timestamp(*index_entries, *nb_index_entries, timestamp,
AVSEEK_FLAG_ANY);
```

```
    if(index < 0)
    {
        index = (*nb_index_entries)++;
        ie = &entries[index];
        assert(index == 0 || ie[-1].timestamp < timestamp);
    }
    else
    {
        ie = &entries[index];
        if(ie->timestamp != timestamp)
        {
            if(ie->timestamp <= timestamp)
                return -1;
            memmove(entries + index + 1, entries + index,
sizeof(AVIndexEntry)*(*nb_index_entries - index));
            (*nb_index_entries)++;
        }
        else if(ie->pos == pos && distance < ie->min_distance)    //do not reduce the distance
            distance = ie->min_distance;
    }

    ie->pos = pos;
    ie->timestamp = timestamp;
    ie->min_distance = distance;
    ie->size = size;
    ie->flags = flags;

    return index;
}

int av_add_index_entry(AVStream *st,
                       int64_t pos, int64_t timestamp, int size, int distance, int flags)
{
    return ff_add_index_entry(&st->index_entries, &st->nb_index_entries,
                              &st->index_entries_allocated_size, pos,
                              timestamp, size, distance, flags);
}

int ff_index_search_timestamp(const AVIndexEntry *entries, int nb_entries,
                              int64_t wanted_timestamp, int flags)
{
    int a, b, m;
    int64_t timestamp;
    a = - 1;
    b = nb_entries;

    //optimize appending index entries at the end
    if(b && entries[b-1].timestamp < wanted_timestamp)
        a = b - 1;

    while (b - a > 1)
    {
        m = (a + b) >> 1;
        timestamp = entries[m].timestamp;
        if(timestamp >= wanted_timestamp)
            b = m;
        if(timestamp <= wanted_timestamp)
            a = m;
    }
    m = (flags & AVSEEK_FLAG_BACKWARD) ? a : b;

    if(!(flags & AVSEEK_FLAG_ANY))
    {
        while(m >= 0 && m < nb_entries && !(entries[m].flags & AVINDEX_KEYFRAME))
        {
            m += (flags & AVSEEK_FLAG_BACKWARD) ? -1 : 1;
        }
    }

    if(m == nb_entries)
        return -1;
    return    m;
}

int av_index_search_timestamp(AVStream *st, int64_t wanted_timestamp,
                              int flags)
{
    return ff_index_search_timestamp(st->index_entries, st->nb_index_entries,
                                     wanted_timestamp, flags);
}

#define DEBUG_SEEK

int av_seek_frame_binary(AVFormatContext *s, int stream_index, int64_t target_ts, int flags)
{
    AVInputFormat *avif = s->iformat;
    int64_t av_uninit(pos_min), av_uninit(pos_max), pos, pos_limit;
    int64_t ts_min, ts_max, ts;
    int index;
    int64_t ret;
    AVStream *st;

    if (stream_index < 0)
        return -1;

#ifdef DEBUG_SEEK
    av_log(s, AV_LOG_DEBUG, "read_seek: %d %"PRId64"\n", stream_index, target_ts);
#endif

    ts_max =
        ts_min = AV_NOPTS_VALUE;
    pos_limit = -1; //gcc falsely says it may be uninitialized

    st = s->streams[stream_index];
    if(st->index_entries)
    {
        AVIndexEntry *e;

        index = av_index_search_timestamp(st, target_ts, flags | AVSEEK_FLAG_BACKWARD); //FIXME whole func must be checked for non-keyframe entries in index case, especially read_timestamp()
        index = FFMAX(index, 0);
        e = &st->index_entries[index];

        if(e->timestamp <= target_ts || e->pos == e->min_distance)
        {
            pos_min = e->pos;
            ts_min = e->timestamp;
#ifdef DEBUG_SEEK
            av_log(s, AV_LOG_DEBUG, "using cached pos_min=0x%"PRIx64" dts_min=%"PRId64"\n",
                   pos_min, ts_min);
#endif
        }
        else
        {
            assert(index == 0);
        }

        index = av_index_search_timestamp(st, target_ts, flags &
~AVSEEK_FLAG_BACKWARD);
            assert(index < st->nb_index_entries);
            if(index >= 0)
            {
                  e = &st->index_entries[index];
                  assert(e->timestamp >= target_ts);
                  pos_max = e->pos;
                  ts_max = e->timestamp;
                  pos_limit = pos_max - e->min_distance;
#ifdef DEBUG_SEEK
                  av_log(s,     AV_LOG_DEBUG,     "using     cached     pos_max=0x%"PRIx64"
pos_limit=0x%"PRIx64" dts_max=%"PRId64"\n",
                              pos_max, pos_limit, ts_max);
#endif
            }
      }

      pos = av_gen_search(s, stream_index, target_ts, pos_min, pos_max, pos_limit, ts_min,
ts_max, flags, &ts, avif->read_timestamp);
      if(pos < 0)
            return -1;

      /* do the seek */
      if ((ret = avio_seek(s->pb, pos, SEEK_SET)) < 0)
            return ret;

      av_update_cur_dts(s, st, ts);

      return 0;
}

int64_t  av_gen_search(AVFormatContext  *s,  int  stream_index,  int64_t  target_ts,  int64_t
pos_min, int64_t pos_max, int64_t pos_limit, int64_t ts_min, int64_t ts_max, int flags, int64_t
*ts_ret, int64_t (*read_timestamp)(struct AVFormatContext *, int , int64_t *, int64_t ))
{
      int64_t pos, ts;
      int64_t start_pos, filesize;
      int no_change;

#ifdef DEBUG_SEEK
      av_log(s, AV_LOG_DEBUG, "gen_seek: %d %"PRId64"\n", stream_index, target_ts);
#endif

      if(ts_min == AV_NOPTS_VALUE)
```

```
{
    pos_min = s->data_offset;
    ts_min = read_timestamp(s, stream_index, &pos_min, INT64_MAX);
    if (ts_min == AV_NOPTS_VALUE)
        return -1;
}

if(ts_max == AV_NOPTS_VALUE)
{
    int step = 1024;
    filesize = avio_size(s->pb);
    pos_max = filesize - 1;
    do
    {
        pos_max -= step;
        ts_max = read_timestamp(s, stream_index, &pos_max, pos_max + step);
        step += step;
    }
    while(ts_max == AV_NOPTS_VALUE && pos_max >= step);
    if (ts_max == AV_NOPTS_VALUE)
        return -1;

    for(;;)
    {
        int64_t tmp_pos = pos_max + 1;
        int64_t tmp_ts = read_timestamp(s, stream_index, &tmp_pos, INT64_MAX);
        if(tmp_ts == AV_NOPTS_VALUE)
            break;
        ts_max = tmp_ts;
        pos_max = tmp_pos;
        if(tmp_pos >= filesize)
            break;
    }
    pos_limit = pos_max;
}

if(ts_min > ts_max)
{
    return -1;
}
else if(ts_min == ts_max)
{
    pos_limit = pos_min;
}
```

```
    no_change = 0;
    while (pos_min < pos_limit)
    {
#ifdef DEBUG_SEEK
        av_log(s, AV_LOG_DEBUG, "pos_min=0x%"PRIx64" pos_max=0x%"PRIx64"
dts_min=%"PRId64" dts_max=%"PRId64"\n",
               pos_min, pos_max,
               ts_min, ts_max);
#endif
        assert(pos_limit <= pos_max);

        if(no_change == 0)
        {
            int64_t approximate_keyframe_distance = pos_max - pos_limit;
            // interpolate position (better than dichotomy)
            pos = av_rescale(target_ts - ts_min, pos_max - pos_min, ts_max - ts_min)
                  + pos_min - approximate_keyframe_distance;
        }
        else if(no_change == 1)
        {
            // bisection, if interpolation failed to change min or max pos last time
            pos = (pos_min + pos_limit) >> 1;
        }
        else
        {
            /* linear search if bisection failed, can only happen if there
               are very few or no keyframes between min/max */
            pos = pos_min;
        }
        if(pos <= pos_min)
            pos = pos_min + 1;
        else if(pos > pos_limit)
            pos = pos_limit;
        start_pos = pos;

        ts = read_timestamp(s, stream_index, &pos, INT64_MAX); //may pass pos_limit
instead of -1
        if(pos == pos_max)
            no_change++;
        else
            no_change = 0;
#ifdef DEBUG_SEEK
        av_log(s, AV_LOG_DEBUG, "%"PRId64" %"PRId64" %"PRId64"
/    %"PRId64"    %"PRId64"    %"PRId64"    target:%"PRId64"    limit:%"PRId64"
start:%"PRId64" noc:%d\n",
                    pos_min, pos, pos_max, ts_min, ts, ts_max, target_ts, pos_limit,
                    start_pos, no_change);
#endif
        if(ts == AV_NOPTS_VALUE)
        {
            av_log(s, AV_LOG_ERROR, "read_timestamp() failed in the middle\n");
            return -1;
        }
        assert(ts != AV_NOPTS_VALUE);
        if (target_ts <= ts)
        {
            pos_limit = start_pos - 1;
            pos_max = pos;
            ts_max = ts;
        }
        if (target_ts >= ts)
        {
            pos_min = pos;
            ts_min = ts;
        }
    }

    pos = (flags & AVSEEK_FLAG_BACKWARD) ? pos_min : pos_max;
    ts  = (flags & AVSEEK_FLAG_BACKWARD) ?  ts_min :  ts_max;
#ifdef DEBUG_SEEK
    pos_min = pos;
    ts_min = read_timestamp(s, stream_index, &pos_min, INT64_MAX);
    pos_min++;
    ts_max = read_timestamp(s, stream_index, &pos_min, INT64_MAX);
    av_log(s,                                                        AV_LOG_DEBUG,
"pos=0x%"PRIx64" %"PRId64"<=%"PRId64"<=%"PRId64"\n",
            pos, ts_min, target_ts, ts_max);
#endif
    *ts_ret = ts;
    return pos;
}

static int av_seek_frame_byte(AVFormatContext *s, int stream_index, int64_t pos, int flags)
{
    int64_t pos_min, pos_max;
#if 0
    AVStream *st;
    if (stream_index < 0)
        return -1;

    st = s->streams[stream_index];
#endif

    pos_min = s->data_offset;
    pos_max = avio_size(s->pb) - 1;

    if      (pos < pos_min) pos = pos_min;
    else if(pos > pos_max) pos = pos_max;

    avio_seek(s->pb, pos, SEEK_SET);

#if 0
    av_update_cur_dts(s, st, ts);
#endif
    return 0;
}

static int av_seek_frame_generic(AVFormatContext *s,
                                 int stream_index, int64_t timestamp, int flags)
{
    int index;
    int64_t ret;
    AVStream *st;
    AVIndexEntry *ie;

    st = s->streams[stream_index];

    index = av_index_search_timestamp(st, timestamp, flags);

    if(index < 0 && st->nb_index_entries && timestamp < st->index_entries[0].timestamp)
        return -1;

    if(index < 0 || index == st->nb_index_entries - 1)
    {
        int i;
        AVPacket pkt;

        if(st->nb_index_entries)
        {
            assert(st->index_entries);
            ie = &st->index_entries[st->nb_index_entries-1];
            if ((ret = avio_seek(s->pb, ie->pos, SEEK_SET)) < 0)
                return ret;
            av_update_cur_dts(s, st, ie->timestamp);
        }
        else
        {
            if ((ret = avio_seek(s->pb, s->data_offset, SEEK_SET)) < 0)
                return ret;
        }
        for(i = 0;; i++)
        {
            int ret;
            do
            {
                ret = av_read_frame(s, &pkt);
            }
            while(ret == AVERROR(EAGAIN));
            if(ret < 0)
                break;
            av_free_packet(&pkt);
            if(stream_index == pkt.stream_index)
            {
                if((pkt.flags & AV_PKT_FLAG_KEY) && pkt.dts > timestamp)
                    break;
            }
        }
        index = av_index_search_timestamp(st, timestamp, flags);
    }
    if (index < 0)
        return -1;

    ff_read_frame_flush(s);
    if (s->iformat->read_seek)
    {
        if(s->iformat->read_seek(s, stream_index, timestamp, flags) >= 0)
            return 0;
    }
    ie = &st->index_entries[index];
    if ((ret = avio_seek(s->pb, ie->pos, SEEK_SET)) < 0)
        return ret;
    av_update_cur_dts(s, st, ie->timestamp);

    return 0;
```

```
}

int av_seek_frame(AVFormatContext *s, int stream_index, int64_t timestamp, int flags)
{
    int ret;
    AVStream *st;

    ff_read_frame_flush(s);

    if(flags & AVSEEK_FLAG_BYTE)
        return av_seek_frame_byte(s, stream_index, timestamp, flags);

    if(stream_index < 0)
    {
        stream_index = av_find_default_stream_index(s);
        if(stream_index < 0)
            return -1;

        st = s->streams[stream_index];
        /* timestamp for default must be expressed in AV_TIME_BASE units */
        timestamp = av_rescale(timestamp, st->time_base.den, AV_TIME_BASE * 
(int64_t)st->time_base.num);
    }

    /* first, we try the format specific seek */
    if (s->iformat->read_seek)
        ret = s->iformat->read_seek(s, stream_index, timestamp, flags);
    else
        ret = -1;
    if (ret >= 0)
    {
        return 0;
    }

    if(s->iformat->read_timestamp)
        return av_seek_frame_binary(s, stream_index, timestamp, flags);
    else
        return av_seek_frame_generic(s, stream_index, timestamp, flags);
}

int avformat_seek_file(AVFormatContext *s, int stream_index, int64_t min_ts, int64_t ts, 
int64_t max_ts, int flags)
{
    if(min_ts > ts || max_ts < ts)
```

```
        return -1;

    ff_read_frame_flush(s);

    if (s->iformat->read_seek2)
        return s->iformat->read_seek2(s, stream_index, min_ts, ts, max_ts, flags);

    if(s->iformat->read_timestamp)
    {
        //try to seek via read_timestamp()
    }

    //Fallback to old API if new is not implemented but old is
    //Note the old has somewat different sematics
    if(s->iformat->read_seek || 1)
        return av_seek_frame(s, stream_index, ts, flags | (ts - min_ts > (uint64_t)(max_ts -
ts) ? AVSEEK_FLAG_BACKWARD : 0));

    // try some generic seek like av_seek_frame_generic() but with new ts semantics
}

/*******************************************************/

/**
  * Return TRUE if the stream has accurate duration in any stream.
  *
  * @return TRUE if the stream has accurate duration for at least one component.
  */
static int av_has_duration(AVFormatContext *ic)
{
    int i;
    AVStream *st;

    for(i = 0; i < ic->nb_streams; i++)
    {
        st = ic->streams[i];
        if (st->duration != AV_NOPTS_VALUE)
            return 1;
    }
    return 0;
}

/**
  * Estimate the stream timings from the one of each components.
```

```
 *
 * Also computes the global bitrate if possible.
 */
static void av_update_stream_timings(AVFormatContext *ic)
{
    int64_t start_time, start_time1, end_time, end_time1;
    int64_t duration, duration1;
    int i;
    AVStream *st;

    start_time = INT64_MAX;
    end_time = INT64_MIN;
    duration = INT64_MIN;
    for(i = 0; i < ic->nb_streams; i++)
    {
        st = ic->streams[i];
        if (st->start_time != AV_NOPTS_VALUE && st->time_base.den)
        {
            start_time1 = av_rescale_q(st->start_time, st->time_base, AV_TIME_BASE_Q);
            if (start_time1 < start_time)
                start_time = start_time1;
            if (st->duration != AV_NOPTS_VALUE)
            {
                end_time1 = start_time1
                            +         av_rescale_q(st->duration,        st->time_base,
AV_TIME_BASE_Q);
                if (end_time1 > end_time)
                    end_time = end_time1;
            }
        }
        if (st->duration != AV_NOPTS_VALUE)
        {
            duration1 = av_rescale_q(st->duration, st->time_base, AV_TIME_BASE_Q);
            if (duration1 > duration)
                duration = duration1;
        }
    }
    if (start_time != INT64_MAX)
    {
        ic->start_time = start_time;
        if (end_time != INT64_MIN)
        {
            if (end_time - start_time > duration)
                duration = end_time - start_time;
        }
    }
    if (duration != INT64_MIN)
    {
        ic->duration = duration;
        if (ic->file_size > 0)
        {
            /* compute the bitrate */
            ic->bit_rate = (double)ic->file_size * 8.0 * AV_TIME_BASE /
                               (double)ic->duration;
        }
    }
}

static void fill_all_stream_timings(AVFormatContext *ic)
{
    int i;
    AVStream *st;

    av_update_stream_timings(ic);
    for(i = 0; i < ic->nb_streams; i++)
    {
        st = ic->streams[i];
        if (st->start_time == AV_NOPTS_VALUE)
        {
            if(ic->start_time != AV_NOPTS_VALUE)
                st->start_time = av_rescale_q(ic->start_time, AV_TIME_BASE_Q,
st->time_base);
            if(ic->duration != AV_NOPTS_VALUE)
                st->duration = av_rescale_q(ic->duration, AV_TIME_BASE_Q,
st->time_base);
        }
    }
}

static void av_estimate_timings_from_bit_rate(AVFormatContext *ic)
{
    int64_t filesize, duration;
    int bit_rate, i;
    AVStream *st;

    /* if bit_rate is already set, we believe it */
    if (ic->bit_rate <= 0)
    {
        bit_rate = 0;
        for(i = 0; i < ic->nb_streams; i++)
        {
            st = ic->streams[i];
            if (st->codec->bit_rate > 0)
                bit_rate += st->codec->bit_rate;
        }
        ic->bit_rate = bit_rate;
    }

    /* if duration is already set, we believe it */
    if (ic->duration == AV_NOPTS_VALUE &&
            ic->bit_rate != 0 &&
            ic->file_size != 0)
    {
        filesize = ic->file_size;
        if (filesize > 0)
        {
            for(i = 0; i < ic->nb_streams; i++)
            {
                st = ic->streams[i];
                duration = av_rescale(8 * filesize, st->time_base.den, ic->bit_rate *
(int64_t)st->time_base.num);
                if (st->duration == AV_NOPTS_VALUE)
                    st->duration = duration;
            }
        }
    }
}

#define DURATION_MAX_READ_SIZE 250000
#define DURATION_MAX_RETRY 3

/* only usable for MPEG-PS streams */
static void av_estimate_timings_from_pts(AVFormatContext *ic, int64_t old_offset)
{
    AVPacket pkt1, *pkt = &pkt1;
    AVStream *st;
    int read_size, i, ret;
    int64_t end_time;
    int64_t filesize, offset, duration;
    int retry = 0;

    ic->cur_st = NULL;
    /* flush packet queue */
    flush_packet_queue(ic);

    for (i = 0; i < ic->nb_streams; i++)
    {
        st = ic->streams[i];
        if (st->start_time == AV_NOPTS_VALUE && st->first_dts == AV_NOPTS_VALUE)
            av_log(st->codec,    AV_LOG_WARNING,    "start    time    is    not    set    in
av_estimate_timings_from_pts\n");

        if (st->parser)
        {
            av_parser_close(st->parser);
            st->parser = NULL;
            av_free_packet(&st->cur_pkt);
        }
    }

    /* estimate the end time (duration) */
    /* XXX: may need to support wrapping */
    filesize = ic->file_size;
    end_time = AV_NOPTS_VALUE;
    do
    {
        offset = filesize - (DURATION_MAX_READ_SIZE << retry);
        if (offset < 0)
            offset = 0;

        avio_seek(ic->pb, offset, SEEK_SET);
        read_size = 0;
        for(;;)
        {
            if (read_size >= DURATION_MAX_READ_SIZE << (FFMAX(retry - 1, 0)))
                break;

            do
            {
                ret = av_read_packet(ic, pkt);
            }
            while(ret == AVERROR(EAGAIN));
            if (ret != 0)
                break;
            read_size += pkt->size;
            st = ic->streams[pkt->stream_index];
            if (pkt->pts != AV_NOPTS_VALUE &&
                    (st->start_time != AV_NOPTS_VALUE ||
                     st->first_dts    != AV_NOPTS_VALUE))
            {
                duration = end_time = pkt->pts;
                if (st->start_time != AV_NOPTS_VALUE)    duration -= st->start_time;
                else                                      duration -= st->first_dts;
                if (duration < 0)
                    duration += 1LL << st->pts_wrap_bits;
                if (duration > 0)
                {
                    if (st->duration == AV_NOPTS_VALUE ||
                            st->duration < duration)
                        st->duration = duration;
                }
            }
            av_free_packet(pkt);
        }
    }
    while(     end_time == AV_NOPTS_VALUE
                && filesize > (DURATION_MAX_READ_SIZE << retry)
                && ++retry <= DURATION_MAX_RETRY);

    fill_all_stream_timings(ic);

    avio_seek(ic->pb, old_offset, SEEK_SET);
    for (i = 0; i < ic->nb_streams; i++)
    {
        st = ic->streams[i];
        st->cur_dts = st->first_dts;
        st->last_IP_pts = AV_NOPTS_VALUE;
        st->reference_dts = AV_NOPTS_VALUE;
    }
}

static void av_estimate_timings(AVFormatContext *ic, int64_t old_offset)
{
    int64_t file_size;

    /* get the file size, if possible */
    if (ic->iformat->flags & AVFMT_NOFILE)
    {
        file_size = 0;
    }
    else
    {
        file_size = avio_size(ic->pb);
        if (file_size < 0)
            file_size = 0;
    }
    ic->file_size = file_size;

    if ((!strcmp(ic->iformat->name, "mpeg") ||
            !strcmp(ic->iformat->name, "mpegts")) &&
            file_size && ic->pb->seekable)
    {
        /* get accurate estimate from the PTSes */
        av_estimate_timings_from_pts(ic, old_offset);
    }
    else if (av_has_duration(ic))
    {
        /* at least one component has timings - we use them for all
           the components */
        fill_all_stream_timings(ic);
    }
    else
    {
        av_log(ic, AV_LOG_WARNING, "Estimating duration from bitrate, this may be
inaccurate\n");
        /* less precise: use bitrate info */
        av_estimate_timings_from_bit_rate(ic);
    }
    av_update_stream_timings(ic);

#if 0
    {
        int i;
        AVStream *st;
        for(i = 0; i < ic->nb_streams; i++)
        {
            st = ic->streams[i];
            printf("%d: start_time: %0.3f duration: %0.3f\n",
                   i, (double)st->start_time / AV_TIME_BASE,
                   (double)st->duration / AV_TIME_BASE);
        }
        printf("stream: start_time: %0.3f duration: %0.3f bitrate=%d kb/s\n",
               (double)ic->start_time / AV_TIME_BASE,
                    (double)ic->duration / AV_TIME_BASE,
                    ic->bit_rate / 1000);
    }
#endif
}

static int has_codec_parameters(AVCodecContext *enc)
{
    int val;
    switch(enc->codec_type)
    {
    case AVMEDIA_TYPE_AUDIO:
        val  =  enc->sample_rate  &&  enc->channels  &&  enc->sample_fmt  !=
AV_SAMPLE_FMT_NONE;
        if(!enc->frame_size &&
                (enc->codec_id == CODEC_ID_VORBIS ||
                 enc->codec_id == CODEC_ID_AAC ||
                 enc->codec_id == CODEC_ID_MP1 ||
                 enc->codec_id == CODEC_ID_MP2 ||
                 enc->codec_id == CODEC_ID_MP3 ||
                 enc->codec_id == CODEC_ID_SPEEX))
            return 0;
        break;
    case AVMEDIA_TYPE_VIDEO:
        val = enc->width && enc->pix_fmt != PIX_FMT_NONE;
        break;
    default:
        val = 1;
        break;
    }
    return enc->codec_id != CODEC_ID_NONE && val != 0;
}

static int has_decode_delay_been_guessed(AVStream *st)
{
    return st->codec->codec_id != CODEC_ID_H264 ||
           st->codec_info_nb_frames >= 6 + st->codec->has_b_frames;
}

static int try_decode_frame(AVStream *st, AVPacket *avpkt)
{
    int16_t *samples;
    AVCodec *codec;
    int got_picture, data_size, ret = 0;
    AVFrame picture;

    if(!st->codec->codec)
    {
        codec = avcodec_find_decoder(st->codec->codec_id);
        if (!codec)
            return -1;
        ret = avcodec_open(st->codec, codec);
        if (ret < 0)
            return ret;
    }

    if(!has_codec_parameters(st->codec) || !has_decode_delay_been_guessed(st))
    {
        switch(st->codec->codec_type)
        {
        case AVMEDIA_TYPE_VIDEO:
            avcodec_get_frame_defaults(&picture);
            ret = avcodec_decode_video2(st->codec, &picture,
                                        &got_picture, avpkt);
            break;
        case AVMEDIA_TYPE_AUDIO:
            data_size = FFMAX(avpkt->size, AVCODEC_MAX_AUDIO_FRAME_SIZE);
            samples = av_malloc(data_size);
            if (!samples)
                goto fail;
            ret = avcodec_decode_audio3(st->codec, samples,
                                        &data_size, avpkt);
            av_free(samples);
            break;
        default:
            break;
        }
    }
fail:
    return ret;
}

unsigned int ff_codec_get_tag(const AVCodecTag *tags, enum CodecID id)
{
    while (tags->id != CODEC_ID_NONE)
    {
        if (tags->id == id)
            return tags->tag;
        tags++;
    }
    return 0;
}

enum CodecID ff_codec_get_id(const AVCodecTag *tags, unsigned int tag)
{
    int i;
    for(i = 0; tags[i].id != CODEC_ID_NONE; i++)
    {
        if(tag == tags[i].tag)
            return tags[i].id;
    }
    for(i = 0; tags[i].id != CODEC_ID_NONE; i++)
    {
        if (ff_toupper4(tag) == ff_toupper4(tags[i].tag))
            return tags[i].id;
    }
    return CODEC_ID_NONE;
}

unsigned int av_codec_get_tag(const AVCodecTag *const *tags, enum CodecID id)
{
    int i;
    for(i = 0; tags && tags[i]; i++)
    {
        int tag = ff_codec_get_tag(tags[i], id);
        if(tag) return tag;
    }
    return 0;
}

enum CodecID av_codec_get_id(const AVCodecTag *const *tags, unsigned int tag)
{
    int i;
    for(i = 0; tags && tags[i]; i++)
    {
        enum CodecID id = ff_codec_get_id(tags[i], tag);
        if(id != CODEC_ID_NONE) return id;
    }
    return CODEC_ID_NONE;
}

static void compute_chapters_end(AVFormatContext *s)
```

```
{
    unsigned int i, j;
    int64_t  max_time  =  s->duration  +  (s->start_time  ==  AV_NOPTS_VALUE)  ?  0  :
s->start_time;

    for (i = 0; i < s->nb_chapters; i++)
        if (s->chapters[i]->end == AV_NOPTS_VALUE)
        {
            AVChapter *ch = s->chapters[i];
            int64_t     end  =  max_time  ?  av_rescale_q(max_time,  AV_TIME_BASE_Q,
ch->time_base)
                            : INT64_MAX;

            for (j = 0; j < s->nb_chapters; j++)
            {
                AVChapter *ch1 = s->chapters[j];
                int64_t     next_start     =     av_rescale_q(ch1->start,     ch1->time_base,
ch->time_base);
                if (j != i && next_start > ch->start && next_start < end)
                    end = next_start;
            }
            ch->end = (end == INT64_MAX) ? ch->start : end;
        }
}

static int get_std_framerate(int i)
{
    if(i < 60 * 12) return i * 1001;
    else            return ((const int[])
    {
        24, 30, 60, 12, 15
    })[i-60*12] * 1000 * 12;
}

/*
 * Is the time base unreliable.
 * This is a heuristic to balance between quick acceptance of the values in
 * the headers vs. some extra checks.
 * Old DivX and Xvid often have nonsense timebases like 1fps or 2fps.
 * MPEG-2 commonly misuses field repeat flags to store different framerates.
 * And there are "variable" fps files this needs to detect as well.
 */
static int tb_unreliable(AVCodecContext *c)
{
    if(     c->time_base.den >= 101L * c->time_base.num
                || c->time_base.den <       5L * c->time_base.num
                /*          || c->codec_tag == AV_RL32("DIVX")
                            || c->codec_tag == AV_RL32("XVID")*/
                || c->codec_id == CODEC_ID_MPEG2VIDEO
                || c->codec_id == CODEC_ID_H264
      )
        return 1;
    return 0;
}

//获取 AVFormatContext *ic 中包含的媒体流的信息
int av_find_stream_info(AVFormatContext *ic)
{
    int i, count, ret, read_size, j;
    AVStream *st;
    AVPacket pkt1, *pkt;
    int64_t old_offset = avio_tell(ic->pb);

    //在开始执行这个函数的代码之前有以下两点需要考虑：
    //1.我们设置的依据是什么？
    //设置的依据是传入的媒体信息，通过前面的打开媒体文件等操作，我们已经获取了一些
信息，这些信息就是我们的依据，
    //但是这些信息保存在哪里呢？首先他们主要不是保存在媒体流的 Codec 中，而是媒体流
本身和 AVFormatContext 上，所以
    //我们设置的依据就是：媒体流本身和 AVFormatContext

    //2.我们设置的对象是什么？
    //我们这只的对象主要就是媒体流的 codec，把它设置好了，才能正常的解码等。
    //通过遍历媒体流
    for(i = 0; i < ic->nb_streams; i++)
    {
        AVCodec *codec;
        st = ic->streams[i];
         //设置 AVStream 的 AVCodec
        if (st->codec->codec_id == CODEC_ID_AAC)
        {
            st->codec->sample_rate = 0;
            st->codec->frame_size = 0;
            st->codec->channels = 0;
        }
         //设置 AVStream 的 AVCodec
        if (st->codec->codec_type == AVMEDIA_TYPE_VIDEO
              || st->codec->codec_type == AVMEDIA_TYPE_SUBTITLE)
```

```
{
        /*if(!st->time_base.num)
            st->time_base= */
        if(!st->codec->time_base.num)
         {
                st->codec->time_base = st->time_base;
         }
}
//only for the split stuff
if (!st->parser && !(ic->flags & AVFMT_FLAG_NOPARSE))
{
         //设置 AVStream 的 struct AVCodecParserContext *parser;
        st->parser = av_parser_init(st->codec->codec_id);
        if(st->need_parsing == AVSTREAM_PARSE_HEADERS && st->parser)
        {
                 //设置 AVStream 的 struct AVCodecParserContext *parser;
                st->parser->flags |= PARSER_FLAG_COMPLETE_FRAMES;
        }
}
assert(!st->codec->codec);
 //检查 AVStream 的 codec 的 codec_id 是否合法
codec = avcodec_find_decoder(st->codec->codec_id);

/* Force decoding of at least one frame of codec data
  * this makes sure the codec initializes the channel configuration
  * and does not trust the values from the container.
  */
if (codec && codec->capabilities & CODEC_CAP_CHANNEL_CONF)
 {
        st->codec->channels = 0;
 }
/* Ensure that subtitle_header is properly set. */
if (st->codec->codec_type == AVMEDIA_TYPE_SUBTITLE
                && codec && !st->codec->codec)
 {
        avcodec_open(st->codec, codec);
 }
 //检查 AVStream 的 codec 是否包含必要的参数
//try to just open decoders, in case this is enough to get parameters
if(!has_codec_parameters(st->codec))
{
        if (codec && !st->codec->codec)
         {
                avcodec_open(st->codec, codec);
            }
        }
    }

    //设置 AVStream 中的 info 对象数据
    for (i = 0; i < ic->nb_streams; i++)
    {
        ic->streams[i]->info->last_dts = AV_NOPTS_VALUE;
    }

    count = 0;
    read_size = 0;
    for(;;)
    {
        //是否需要终止
        if(url_interrupt_cb())
        {
            ret = AVERROR_EXIT;
            av_log(ic, AV_LOG_DEBUG, "interrupted\n");
            break;
        }

        /* check if one codec still needs to be handled */
        for(i = 0; i < ic->nb_streams; i++)
        {
            int fps_analyze_framecount = 20;

            st = ic->streams[i];
            //检查 AVStream 的 codec 是否包含必要的参数
             if (!has_codec_parameters(st->codec))
             {
                 break;
             }
            /* if the timebase is coarse (like the usual millisecond precision
               of mkv), we need to analyze more frames to reliably arrive at
               the correct fps */
            if (av_q2d(st->time_base) > 0.0005)
             {
                 fps_analyze_framecount *= 2;
             }
            /* variable fps and no guess at the real fps */
            if(tb_unreliable(st->codec)
                 && !(st->r_frame_rate.num && st->avg_frame_rate.num)
                 && st->info->duration_count < fps_analyze_framecount
```

```
                    && st->codec->codec_type == AVMEDIA_TYPE_VIDEO)
                {
                    break;
                }
                if(st->parser
                     && st->parser->parser->split
                     && !st->codec->extradata)
                {
                    break;
                }
                if(st->first_dts == AV_NOPTS_VALUE)
                {
                    break;
                }
            }
            if (i == ic->nb_streams)
            {
                /* NOTE: if the format has no header, then we need to read
                   some packets to get most of the streams, so we cannot
                   stop here */
                if (!(ic->ctx_flags & AVFMTCTX_NOHEADER))
                {
                    /* if we found the info for all the codecs, we can stop */
                    ret = count;
                    av_log(ic, AV_LOG_DEBUG, "All info found\n");
                    break;
                }
            }
            /* we did not get all the codec info, but we read too much data */
            if (read_size >= ic->probesize)
            {
                ret = count;
                av_log(ic,   AV_LOG_DEBUG,   "Probe   buffer   size   limit   %d   reached\n",
ic->probesize);
                break;
            }

            /* NOTE: a new stream can be added there if no header in file
                 (AVFMTCTX_NOHEADER) */
             //通过读取文件来判断合法性
            ret = av_read_frame_internal(ic, &pkt1);
            if (ret < 0 && ret != AVERROR(EAGAIN))
            {
                /* EOF or error */
```

```
        ret = -1; /* we could not have all the codec parameters before EOF */
        for(i = 0; i < ic->nb_streams; i++)
        {
            st = ic->streams[i];
            if (!has_codec_parameters(st->codec))
            {
                char buf[256];
                avcodec_string(buf, sizeof(buf), st->codec, 0);
                av_log(ic, AV_LOG_WARNING,
                     "Could not find codec parameters (%s)\n", buf);
            }
            else
            {
                ret = 0;
            }
        }
        break;
    }

    if (ret == AVERROR(EAGAIN))
        continue;
    //检查 pkt
    pkt = add_to_pktbuf(&ic->packet_buffer, &pkt1, &ic->packet_buffer_end);
    if ((ret = av_dup_packet(pkt)) < 0)
        goto find_stream_info_err;

    read_size += pkt->size;
    //看着这里，感觉真是一片混乱的代码，
    //但是它们居然存活下来，还能正常工作，真是他妈的奇迹啊！  狗屎！
    st = ic->streams[pkt->stream_index];
    if (st->codec_info_nb_frames > 1)
    {
        if (st->time_base.den > 0
             && av_rescale_q(st->info->codec_info_duration,
                 st->time_base, AV_TIME_BASE_Q) >= ic->max_analyze_duration)
        {
            av_log(ic,
                 AV_LOG_WARNING,
                 "max_analyze_duration reached\n");
            break;
        }
        st->info->codec_info_duration += pkt->duration;
    }
    {
            int64_t last = st->info->last_dts;
            int64_t duration = pkt->dts - last;

            if(pkt->dts != AV_NOPTS_VALUE
                 && last != AV_NOPTS_VALUE && duration > 0)
            {
                double dur =
                      duration * av_q2d(st->time_base);

                //                     if(st->codec->codec_type == AVMEDIA_TYPE_VIDEO)
                //                         av_log(NULL, AV_LOG_ERROR, "%f\n", dur);
                if (st->info->duration_count < 2)
                    memset(st->info->duration_error, 0, sizeof(st->info->duration_error));
                for (i = 1; i < FF_ARRAY_ELEMS(st->info->duration_error); i++)
                {
                    int framerate = get_std_framerate(i);
                    int ticks = lrintf(dur * framerate / (1001 * 12));
                    double error = dur - ticks * 1001 * 12 / (double)framerate;
                    st->info->duration_error[i] += error * error;
                }
                st->info->duration_count++;
                // ignore the first 4 values, they might have some random jitter
                if (st->info->duration_count > 3)
                    st->info->duration_gcd = av_gcd(st->info->duration_gcd, duration);
            }
            if (last == AV_NOPTS_VALUE || st->info->duration_count <= 1)
                st->info->last_dts = pkt->dts;
        }
        if(st->parser && st->parser->parser->split && !st->codec->extradata)
        {
            int i = st->parser->parser->split(st->codec, pkt->data, pkt->size);
            if(i)
            {
                st->codec->extradata_size = i;
                st->codec->extradata        =        av_malloc(st->codec->extradata_size        +
FF_INPUT_BUFFER_PADDING_SIZE);
                memcpy(st->codec->extradata, pkt->data, st->codec->extradata_size);
                memset(st->codec->extradata                 +                 i,                 0,
FF_INPUT_BUFFER_PADDING_SIZE);
            }
        }

        /* if still no information, we try to open the codec and to
           decompress the frame. We try to avoid that in most cases as
           it takes longer and uses more memory. For MPEG-4, we need to
           decompress for QuickTime. */
        if (!has_codec_parameters(st->codec) || !has_decode_delay_been_guessed(st))
            try_decode_frame(st, pkt);

        st->codec_info_nb_frames++;
        count++;
    }

    // close codecs which were opened in try_decode_frame()
    for(i = 0; i < ic->nb_streams; i++)
    {
        st = ic->streams[i];
        if(st->codec->codec)
            avcodec_close(st->codec);
    }
    for(i = 0; i < ic->nb_streams; i++)
    {
        st = ic->streams[i];
        if (st->codec_info_nb_frames > 2
             && !st->avg_frame_rate.num
             && st->info->codec_info_duration)
        {
            av_reduce(&st->avg_frame_rate.num,
                 &st->avg_frame_rate.den,
                (st->codec_info_nb_frames - 2)*(int64_t)st->time_base.den,
                st->info->codec_info_duration*(int64_t)st->time_base.num,
                 60000);
        }
        if (st->codec->codec_type == AVMEDIA_TYPE_VIDEO)
        {
            if(st->codec->codec_id == CODEC_ID_RAWVIDEO
                 && !st->codec->codec_tag
                 && !st->codec->bits_per_coded_sample)
            {
                uint32_t tag =
                     avcodec_pix_fmt_to_codec_tag(st->codec->pix_fmt);
                if(ff_find_pix_fmt(av_getff_raw_pix_fmt_tags(), tag)
                     == st->codec->pix_fmt)
                 {
                    st->codec->codec_tag = tag;
                 }
            }
            // the check for tb_unreliable() is not completely correct, since this is not about
handling
            // a unreliable/inexact time base, but a time base that is finer than necessary, as
e.g.
            // ipmovie.c produces.
            if (tb_unreliable(st->codec)
                 && st->info->duration_count > 15
                 && st->info->duration_gcd >
                      FFMAX(1, st->time_base.den
                      / (500LL * st->time_base.num))
                 && !st->r_frame_rate.num)
            {
                av_reduce(&st->r_frame_rate.num,
                     &st->r_frame_rate.den,
                     st->time_base.den,
                     st->time_base.num * st->info->duration_gcd,
                     INT_MAX);
            if (st->info->duration_count && !st->r_frame_rate.num
                     && tb_unreliable(st->codec) /*&&
               //FIXME we should not special-case MPEG-2, but this needs testing with
non-MPEG-2 ...
               st->time_base.num*duration_sum[i]/st->info->duration_count*101LL       >
st->time_base.den*/)
            {
                int num = 0;
                double best_error = 2 * av_q2d(st->time_base);
                best_error = best_error * best_error * st->info->duration_count * 1000 * 12
* 30;

                for (j = 1; j < FF_ARRAY_ELEMS(st->info->duration_error); j++)
                {
                    double error = st->info->duration_error[j] * get_std_framerate(j);
                    //                                    if(st->codec->codec_type   ==
AVMEDIA_TYPE_VIDEO)
                    //                              av_log(NULL,  AV_LOG_ERROR,
"%f %f\n", get_std_framerate(j) / 12.0/1001, error);
                    if(error < best_error)
                    {
                        best_error = error;
                        num = get_std_framerate(j);
                    }
                }
                // do not increase frame rate by more than 1 % in order to match a standard
rate.
```

```
                if (num && (!st->r_frame_rate.num
                     || (double)num / (12 * 1001) < 1.01 * av_q2d(st->r_frame_rate)))
                    av_reduce(&st->r_frame_rate.num, &st->r_frame_rate.den, num, 12 *
1001, INT_MAX);
            }

            if (!st->r_frame_rate.num)
            {
                if(       st->codec->time_base.den * (int64_t)st->time_base.num
                        <=   st->codec->time_base.num   *   st->codec->ticks_per_frame   *
(int64_t)st->time_base.den)
                {
                    st->r_frame_rate.num = st->codec->time_base.den;
                    st->r_frame_rate.den          =          st->codec->time_base.num          *
st->codec->ticks_per_frame;
                }
                else
                {
                    st->r_frame_rate.num = st->time_base.den;
                    st->r_frame_rate.den = st->time_base.num;
                }
            }
        }
        else if(st->codec->codec_type == AVMEDIA_TYPE_AUDIO)
        {
            if(!st->codec->bits_per_coded_sample)
                st->codec->bits_per_coded_sample                                          =
av_get_bits_per_sample(st->codec->codec_id);
            // set stream disposition based on audio service type
            switch (st->codec->audio_service_type)
            {
            case AV_AUDIO_SERVICE_TYPE_EFFECTS:
                st->disposition = AV_DISPOSITION_CLEAN_EFFECTS;
                break;
            case AV_AUDIO_SERVICE_TYPE_VISUALLY_IMPAIRED:
                st->disposition = AV_DISPOSITION_VISUAL_IMPAIRED;
                break;
            case AV_AUDIO_SERVICE_TYPE_HEARING_IMPAIRED:
                st->disposition = AV_DISPOSITION_HEARING_IMPAIRED;
                break;
            case AV_AUDIO_SERVICE_TYPE_COMMENTARY:
                st->disposition = AV_DISPOSITION_COMMENT;
                break;
            case AV_AUDIO_SERVICE_TYPE_KARAOKE:
                st->disposition = AV_DISPOSITION_KARAOKE;
                break;
            }
        }
    }

    av_estimate_timings(ic, old_offset);

    compute_chapters_end(ic);

#if 0
    /* correct DTS for B-frame streams with no timestamps */
    for(i = 0; i < ic->nb_streams; i++)
    {
        st = ic->streams[i];
        if (st->codec->codec_type == AVMEDIA_TYPE_VIDEO)
        {
            if(b - frames)
            {
                ppktl = &ic->packet_buffer;
                while(ppkt1)
                {
                    if(ppkt1->stream_index != i)
                        continue;
                    if(ppkt1->pkt->dts < 0)
                        break;
                    if(ppkt1->pkt->pts != AV_NOPTS_VALUE)
                        break;
                    ppkt1->pkt->dts -= delta;
                    ppkt1 = ppkt1->next;
                }
                if(ppkt1)
                    continue;
                st->cur_dts -= delta;
            }
        }
    }
#endif

find_stream_info_err:
    for (i = 0; i < ic->nb_streams; i++)
        av_freep(&ic->streams[i]->info);
    return ret;
}
```

```
static AVProgram *find_program_from_stream(AVFormatContext *ic, int s)
{
    int i, j;

    for (i = 0; i < ic->nb_programs; i++)
        for (j = 0; j < ic->programs[i]->nb_stream_indexes; j++)
            if (ic->programs[i]->stream_index[j] == s)
                return ic->programs[i];
    return NULL;
}

int av_find_best_stream(AVFormatContext *ic,
                        enum AVMediaType type,
                        int wanted_stream_nb,
                        int related_stream,
                        AVCodec **decoder_ret,
                        int flags)
{
    int i, nb_streams = ic->nb_streams;
    int ret = AVERROR_STREAM_NOT_FOUND, best_count = -1;
    unsigned *program = NULL;
    AVCodec *decoder = NULL, *best_decoder = NULL;

    if (related_stream >= 0 && wanted_stream_nb < 0)
    {
        AVProgram *p = find_program_from_stream(ic, related_stream);
        if (p)
        {
            program = p->stream_index;
            nb_streams = p->nb_stream_indexes;
        }
    }
    for (i = 0; i < nb_streams; i++)
    {
        int real_stream_index = program ? program[i] : i;
        AVStream *st = ic->streams[real_stream_index];
        AVCodecContext *avctx = st->codec;
        if (avctx->codec_type != type)
            continue;
        if (wanted_stream_nb >= 0 && real_stream_index != wanted_stream_nb)
            continue;
        if      (st->disposition      &      (AV_DISPOSITION_HEARING_IMPAIRED      |
AV_DISPOSITION_VISUAL_IMPAIRED))
```

```
            continue;
        if (decoder_ret)
        {
            decoder = avcodec_find_decoder(st->codec->codec_id);
            if (!decoder)
            {
                if (ret < 0)
                    ret = AVERROR_DECODER_NOT_FOUND;
                continue;
            }
        }
        if (best_count >= st->codec_info_nb_frames)
            continue;
        best_count = st->codec_info_nb_frames;
        ret = real_stream_index;
        best_decoder = decoder;
        if (program && i == nb_streams - 1 && ret < 0)
        {
            program = NULL;
            nb_streams = ic->nb_streams;
            i = 0; /* no related stream found, try again with everything */
        }
    }
    if (decoder_ret)
        *decoder_ret = best_decoder;
    return ret;
}

/***************************************************/

int av_read_play(AVFormatContext *s)
{
    if (s->iformat->read_play)
        return s->iformat->read_play(s);
    if (s->pb)
        return avio_pause(s->pb, 0);
    return AVERROR(ENOSYS);
}

int av_read_pause(AVFormatContext *s)
{
    if (s->iformat->read_pause)
        return s->iformat->read_pause(s);
    if (s->pb)
```

```
        return avio_pause(s->pb, 1);
    return AVERROR(ENOSYS);
}

void av_close_input_stream(AVFormatContext *s)
{
    flush_packet_queue(s);
    if (s->iformat->read_close)
        s->iformat->read_close(s);
    avformat_free_context(s);
}

void avformat_free_context(AVFormatContext *s)
{
    int i;
    AVStream *st;

    for(i = 0; i < s->nb_streams; i++)
    {
        /* free all data in a stream component */
        st = s->streams[i];
        if (st->parser)
        {
            av_parser_close(st->parser);
            av_free_packet(&st->cur_pkt);
        }
        av_metadata_free(&st->metadata);
        av_free(st->index_entries);
        av_free(st->codec->extradata);
        av_free(st->codec->subtitle_header);
        av_free(st->codec);
#if FF_API_OLD_METADATA
        av_free(st->filename);
#endif
        av_free(st->priv_data);
        av_free(st->info);
        av_free(st);
    }
    for(i = s->nb_programs - 1; i >= 0; i--)
    {
#if FF_API_OLD_METADATA
        av_freep(&s->programs[i]->provider_name);
        av_freep(&s->programs[i]->name);
#endif
```

```
            av_metadata_free(&s->programs[i]->metadata);
            av_freep(&s->programs[i]->stream_index);
            av_freep(&s->programs[i]);
        }
        av_freep(&s->programs);
        av_freep(&s->priv_data);
        while(s->nb_chapters--)
        {
#if FF_API_OLD_METADATA
            av_free(s->chapters[s->nb_chapters]->title);
#endif
            av_metadata_free(&s->chapters[s->nb_chapters]->metadata);
            av_free(s->chapters[s->nb_chapters]);
        }
        av_freep(&s->chapters);
        av_metadata_free(&s->metadata);
        av_freep(&s->key);
        av_free(s);
}

void av_close_input_file(AVFormatContext *s)
{
        AVIOContext *pb = s->iformat->flags & AVFMT_NOFILE ? NULL : s->pb;
        av_close_input_stream(s);
        if (pb)
            avio_close(pb);
}

//给 AVFormatContext *s 生成一个新的媒体流(不管输入和输出)
AVStream *av_new_stream(AVFormatContext *s, int id)
{
        AVStream *st;
        int i;

#if FF_API_MAX_STREAMS
        if (s->nb_streams >= MAX_STREAMS)
        {
            av_log(s, AV_LOG_ERROR, "Too many streams\n");
            return NULL;
        }
#else
        AVStream **streams;
        if (s->nb_streams >= INT_MAX / sizeof(*streams))
        {
```

```
        return NULL;
    }
    //此处的内存分配具有的技巧是：对一个 AVStream 的数组类型数据
    //(AVFormatContext  *s)-(streams), 需要保存已有的数据并在已有的基础上增加一个媒体
流
    streams = av_realloc(s->streams,
            (s->nb_streams + 1) * sizeof(*streams));
    if (!streams)
    {
        return NULL;
    }
    s->streams = streams;
#endif
    //分配一个 AVStream，并置零
    st = av_mallocz(sizeof(AVStream));
    if (!st)
    {
        return NULL;
    }
    //对 AVStream *st 设置 info
    if (!(st->info = av_mallocz(sizeof(*st->info))))
    {
        av_free(st);
        return NULL;
    }
    //对 AVStream *st 设置 codec
    st->codec = avcodec_alloc_context();
    if (s->iformat)
    {
        /* no default bitrate if decoding */
         //对 AVStream *st 设置 codec 中的 bit_rate
        st->codec->bit_rate = 0;
    }
    //对 AVStream *st 设置 index
    st->index = s->nb_streams;
    //对 AVStream *st 设置 id
    st->id = id;
    //对 AVStream *st 设置 start_time
    st->start_time = AV_NOPTS_VALUE;
    //对 AVStream *st 设置 duration
    st->duration = AV_NOPTS_VALUE;
    /* we set the current DTS to 0 so that formats without any timestamps
       but durations get some timestamps, formats with some unknown
       timestamps have their first few packets buffered and the
```

```
            timestamps corrected before they are returned to the user */
        //对 AVStream *st 设置 cur_dts
        st->cur_dts = 0;
        //对 AVStream *st 设置 first_dts
        st->first_dts = AV_NOPTS_VALUE;
        //对 AVStream *st 设置 probe_packets
        st->probe_packets = MAX_PROBE_PACKETS;

        /* default pts setting is MPEG-like */
        //对 AVStream *st 设置 time_base, pts_wrap_bits
        av_set_pts_info(st, 33, 1, 90000);
        //对 AVStream *st 设置 last_IP_pts
        st->last_IP_pts = AV_NOPTS_VALUE;
        //对 AVStream *st 设置 pts_buffer
        for(i = 0; i < MAX_REORDER_DELAY + 1; i++)
        {
                st->pts_buffer[i] = AV_NOPTS_VALUE;
        }
        //对 AVStream *st 设置 reference_dts
        st->reference_dts = AV_NOPTS_VALUE;
        //对 AVStream *st 设置 sample_aspect_ratio
        st->sample_aspect_ratio = (AVRational)
        {
                0, 1
        };
        //最后一步也是最重要的一步就是有两个作用：
        //1.将生成的 AVStream 添加到(AVFormatContext *s)-(streams)
        //2.增加(AVFormatContext *s)-(nb_streams)，同时也为下一个 AVStream 做了准备
        s->streams[s->nb_streams++] = st;
        return st;
}

AVProgram *av_new_program(AVFormatContext *ac, int id)
{
        AVProgram *program = NULL;
        int i;

#ifdef DEBUG_SI
        av_log(ac, AV_LOG_DEBUG, "new_program: id=0x%04x\n", id);
#endif

        for(i = 0; i < ac->nb_programs; i++)
                if(ac->programs[i]->id == id)
                        program = ac->programs[i];
```

```
    if(!program)
    {
        program = av_mallocz(sizeof(AVProgram));
        if (!program)
            return NULL;
        dynarray_add(&ac->programs, &ac->nb_programs, program);
        program->discard = AVDISCARD_NONE;
    }
    program->id = id;

    return program;
}

AVChapter *ff_new_chapter(AVFormatContext *s, int id, AVRational time_base, int64_t start,
int64_t end, const char *title)
{
    AVChapter *chapter = NULL;
    int i;

    for(i = 0; i < s->nb_chapters; i++)
        if(s->chapters[i]->id == id)
            chapter = s->chapters[i];

    if(!chapter)
    {
        chapter = av_mallocz(sizeof(AVChapter));
        if(!chapter)
            return NULL;
        dynarray_add(&s->chapters, &s->nb_chapters, chapter);
    }
#if FF_API_OLD_METADATA
    av_free(chapter->title);
#endif
    av_metadata_set2(&chapter->metadata, "title", title, 0);
    chapter->id        = id;
    chapter->time_base = time_base;
    chapter->start = start;
    chapter->end      = end;

    return chapter;
}

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

```
/* output media file */
#include "rtsp.h"

int av_set_parameters(AVFormatContext *s, AVFormatParameters *ap)
{
    int ret;
    if(NULL == ap && s)
    {
        if(strstr(s->filename, ".sdp")
                || strstr(s->filename, "rtsp:"))
        {
            int i = 0, socketIndex = 0;
            RTSPStream *rtsp_st = NULL;
            RTSPState *rt = (RTSPState *)s->priv_data;
            switch(rt->lower_transport)
            {
            case RTSP_LOWER_TRANSPORT_TCP:
                socketIndex = ffurl_get_file_handle(rt->rtsp_hd);
                rt->rtsp_hd;
                break;
            case RTSP_LOWER_TRANSPORT_UDP:
            case RTSP_LOWER_TRANSPORT_UDP_MULTICAST:
                for (i = 0; i < rt->nb_rtsp_streams; i++)
                {
                    rtsp_st = rt->rtsp_streams[i];
                    if (rtsp_st->rtp_handle)
                    {
                        socketIndex = 0;
                        socketIndex = ffurl_get_file_handle(rtsp_st->rtp_handle);
                        if(socketIndex > 0)
                        {
                            return socketIndex;
                        }
                    }
                }
                break;
            }
            if(socketIndex > 0)
            {
                return socketIndex;
            }
        }
        return 0;
    }
    if (s->oformat->priv_data_size > 0)
    {
        s->priv_data = av_mallocz(s->oformat->priv_data_size);
        if (!s->priv_data)
            return AVERROR(ENOMEM);
        if (s->oformat->priv_class)
        {
            *(const AVClass **)s->priv_data = s->oformat->priv_class;
            av_opt_set_defaults(s->priv_data);
        }
    }
    else
        s->priv_data = NULL;

    if (s->oformat->set_parameters)
    {
        ret = s->oformat->set_parameters(s, ap);
        if (ret < 0)
            return ret;
    }
    return 0;
}

static int validate_codec_tag(AVFormatContext *s, AVStream *st)
{
    const AVCodecTag *avctag;
    int n;
    enum CodecID id = CODEC_ID_NONE;
    unsigned int tag = 0;

    /**
      * Check that tag + id is in the table
      * If neither is in the table -> OK
      * If tag is in the table with another id -> FAIL
      * If id is in the table with another tag -> FAIL unless strict < normal
      */
    for (n = 0; s->oformat->codec_tag[n]; n++)
    {
        avctag = s->oformat->codec_tag[n];
        while (avctag->id != CODEC_ID_NONE)
        {
            if (ff_toupper4(avctag->tag) == ff_toupper4(st->codec->codec_tag))
            {
                id = avctag->id;
                if (id == st->codec->codec_id)
                    return 1;
            }
            if (avctag->id == st->codec->codec_id)
                tag = avctag->tag;
            avctag++;
        }
    }
    if (id != CODEC_ID_NONE)
        return 0;
    if (tag && (st->codec->strict_std_compliance >= FF_COMPLIANCE_NORMAL))
        return 0;
    return 1;
}

int av_write_header(AVFormatContext *s)
{
    int ret, i;
    AVStream *st;

    // some sanity checks
    if (s->nb_streams == 0 && !(s->oformat->flags & AVFMT_NOSTREAMS))
    {
        av_log(s, AV_LOG_ERROR, "no streams\n");
        return AVERROR(EINVAL);
    }

    for(i = 0; i < s->nb_streams; i++)
    {
        st = s->streams[i];

        switch (st->codec->codec_type)
        {
        case AVMEDIA_TYPE_AUDIO:
            if(st->codec->sample_rate <= 0)
            {
                av_log(s, AV_LOG_ERROR, "sample rate not set\n");
                return AVERROR(EINVAL);
            }
            if(!st->codec->block_align)
                st->codec->block_align = st->codec->channels *

av_get_bits_per_sample(st->codec->codec_id) >> 3;
            break;
        case AVMEDIA_TYPE_VIDEO:
            if(st->codec->time_base.num <= 0 || st->codec->time_base.den <= 0) //FIXME
audio too?
            {
                av_log(s, AV_LOG_ERROR, "time base not set\n");
                return AVERROR(EINVAL);
            }
            if((st->codec->width <= 0 || st->codec->height <= 0) && !(s->oformat->flags &
AVFMT_NODIMENSIONS))
            {
                av_log(s, AV_LOG_ERROR, "dimensions not set\n");
                return AVERROR(EINVAL);
            }
            if(av_cmp_q(st->sample_aspect_ratio, st->codec->sample_aspect_ratio))
            {
                av_log(s, AV_LOG_ERROR, "Aspect ratio mismatch between encoder and
muxer layer\n");
                return AVERROR(EINVAL);
            }
            break;
        }

        if(s->oformat->codec_tag)
        {
            if(st->codec->codec_tag && st->codec->codec_id == CODEC_ID_RAWVIDEO
&&     av_codec_get_tag(s->oformat->codec_tag,     st->codec->codec_id)     ==     0
&& !validate_codec_tag(s, st))
            {
                //the current rawvideo encoding system ends up setting the wrong codec_tag
for avi, we override it here
                st->codec->codec_tag = 0;
            }
            if(st->codec->codec_tag)
            {
                if (!validate_codec_tag(s, st))
                {
                    char tagbuf[32];
                    av_get_codec_tag_string(tagbuf, sizeof(tagbuf), st->codec->codec_tag);
                    av_log(s, AV_LOG_ERROR,
                           "Tag %s/0x%08x incompatible with output codec id '%d'\n",
                           tagbuf, st->codec->codec_tag, st->codec->codec_id);
                    return AVERROR_INVALIDDATA;
                }
            }
            else
                st->codec->codec_tag = av_codec_get_tag(s->oformat->codec_tag,
st->codec->codec_id);
        }

        if(s->oformat->flags & AVFMT_GLOBALHEADER &&
                !(st->codec->flags & CODEC_FLAG_GLOBAL_HEADER))
            av_log(s, AV_LOG_WARNING, "Codec for stream %d does not use global
headers but container format requires global headers\n", i);
    }

    if (!s->priv_data && s->oformat->priv_data_size > 0)
    {
        s->priv_data = av_mallocz(s->oformat->priv_data_size);
        if (!s->priv_data)
            return AVERROR(ENOMEM);
    }

#if FF_API_OLD_METADATA
    ff_metadata_mux_compat(s);
#endif

    /* set muxer identification string */
    if (s->nb_streams && !(s->streams[0]->codec->flags & CODEC_FLAG_BITEXACT))
    {
        av_metadata_set2(&s->metadata, "encoder", LIBAVFORMAT_IDENT, 0);
    }

    if(s->oformat->write_header)
    {
        ret = s->oformat->write_header(s);
        if (ret < 0)
            return ret;
    }

    /* init PTS generation */
    for(i = 0; i < s->nb_streams; i++)
    {
        int64_t den = AV_NOPTS_VALUE;
        st = s->streams[i];

        switch (st->codec->codec_type)
        {
        case AVMEDIA_TYPE_AUDIO:
            den = (int64_t)st->time_base.num * st->codec->sample_rate;
            break;
        case AVMEDIA_TYPE_VIDEO:
            den = (int64_t)st->time_base.num * st->codec->time_base.den;
            break;
        default:
            break;
        }
        if (den != AV_NOPTS_VALUE)
        {
            if (den <= 0)
                return AVERROR_INVALIDDATA;
            av_frac_init(&st->pts, 0, 0, den);
        }
    }
    return 0;
}

//FIXME merge with compute_pkt_fields
static int compute_pkt_fields2(AVFormatContext *s, AVStream *st, AVPacket *pkt)
{
    int delay = FFMAX(st->codec->has_b_frames, !!st->codec->max_b_frames);
    int num, den, frame_size, i;

    av_dlog(s,  "compute_pkt_fields2:  pts:%"PRId64"  dts:%"PRId64"  cur_dts:%"PRId64"
b:%d size:%d st:%d\n",
            pkt->pts, pkt->dts, st->cur_dts, delay, pkt->size, pkt->stream_index);

    /*      if(pkt->pts == AV_NOPTS_VALUE && pkt->dts == AV_NOPTS_VALUE)
            return AVERROR(EINVAL);*/

    /* duration field */
    if (pkt->duration == 0)
    {
        compute_frame_duration(&num, &den, st, NULL, pkt);
        if (den && num)
        {
            pkt->duration    =    av_rescale(1,    num    *    (int64_t)st->time_base.den    *
st->codec->ticks_per_frame, den * (int64_t)st->time_base.num);
        }
    }

    if(pkt->pts == AV_NOPTS_VALUE && pkt->dts != AV_NOPTS_VALUE && delay == 0)
        pkt->pts = pkt->dts;
```

```
    //XXX/FIXME this is a temporary hack until all encoders output pts
    if((pkt->pts == 0 || pkt->pts == AV_NOPTS_VALUE) && pkt->dts ==
AV_NOPTS_VALUE && !delay)
    {
        pkt->dts =
            //          pkt->pts= st->cur_dts;
            pkt->pts = st->pts.val;
    }

    //calculate dts from pts
    if(pkt->pts != AV_NOPTS_VALUE && pkt->dts == AV_NOPTS_VALUE && delay <=
MAX_REORDER_DELAY)
    {
        st->pts_buffer[0] = pkt->pts;
        for(i = 1; i < delay + 1 && st->pts_buffer[i] == AV_NOPTS_VALUE; i++)
            st->pts_buffer[i] = pkt->pts + (i - delay - 1) * pkt->duration;
        for(i = 0; i < delay && st->pts_buffer[i] > st->pts_buffer[i+1]; i++)
            FFSWAP(int64_t, st->pts_buffer[i], st->pts_buffer[i+1]);

        pkt->dts = st->pts_buffer[0];
    }

    if(st->cur_dts && st->cur_dts != AV_NOPTS_VALUE && st->cur_dts >= pkt->dts)
    {
        av_log(s, AV_LOG_ERROR,
               "Application provided invalid, non monotonically increasing dts to muxer in
stream %d: %"PRId64" >= %"PRId64"\n",
               st->index, st->cur_dts, pkt->dts);
        return AVERROR(EINVAL);
    }
    if(pkt->dts != AV_NOPTS_VALUE && pkt->pts != AV_NOPTS_VALUE && pkt->pts <
pkt->dts)
    {
        av_log(s, AV_LOG_ERROR, "pts < dts in stream %d\n", st->index);
        return AVERROR(EINVAL);
    }

    //          av_log(s, AV_LOG_DEBUG, "av_write_frame: pts2:%"PRId64"
dts2:%"PRId64"\n", pkt->pts, pkt->dts);
    st->cur_dts = pkt->dts;
    st->pts.val = pkt->dts;

    /* update pts */
    switch (st->codec->codec_type)
    {
    case AVMEDIA_TYPE_AUDIO:
        frame_size = get_audio_frame_size(st->codec, pkt->size);

        /* HACK/FIXME, we skip the initial 0 size packets as they are most
           likely equal to the encoder delay, but it would be better if we
           had the real timestamps from the encoder */
        if (frame_size >= 0 && (pkt->size || st->pts.num != st->pts.den >> 1 || st->pts.val))
        {
            av_frac_add(&st->pts, (int64_t)st->time_base.den * frame_size);
        }
        break;
    case AVMEDIA_TYPE_VIDEO:
        av_frac_add(&st->pts, (int64_t)st->time_base.den * st->codec->time_base.num);
        break;
    default:
        break;
    }
    return 0;
}

int av_write_frame(AVFormatContext *s, AVPacket *pkt)
{
    int ret = compute_pkt_fields2(s, s->streams[pkt->stream_index], pkt);

    if(ret < 0 && !(s->oformat->flags & AVFMT_NOTIMESTAMPS))
        return ret;

    ret = s->oformat->write_packet(s, pkt);
    if(!ret)
        ret = url_ferror(s->pb);
    return ret;
}

void ff_interleave_add_packet(AVFormatContext *s, AVPacket *pkt,
                              int (*compare)(AVFormatContext *, AVPacket *, AVPacket
*))
{
    AVPacketList **next_point, *this_pktl;

    this_pktl = av_mallocz(sizeof(AVPacketList));
    this_pktl->pkt = *pkt;
    pkt->destruct = NULL;            // do not free original but only the copy
    av_dup_packet(&this_pktl->pkt);    // duplicate the packet if it uses non-alloced memory

    if(s->streams[pkt->stream_index]->last_in_packet_buffer)
    {
        next_point = &(s->streams[pkt->stream_index]->last_in_packet_buffer->next);
    }
    else
        next_point = &s->packet_buffer;

    if(*next_point)
    {
        if(compare(s, &s->packet_buffer_end->pkt, pkt))
        {
            while(!compare(s, &(*next_point)->pkt, pkt))
            {
                next_point = &(*next_point)->next;
            }
            goto next_non_null;
        }
        else
        {
            next_point = &(s->packet_buffer_end->next);
        }
    }
    assert(!*next_point);

    s->packet_buffer_end = this_pktl;
next_non_null:

    this_pktl->next = *next_point;

    s->streams[pkt->stream_index]->last_in_packet_buffer =
        *next_point = this_pktl;
}

static int ff_interleave_compare_dts(AVFormatContext *s, AVPacket *next, AVPacket *pkt)
{
    AVStream *st = s->streams[ pkt ->stream_index];
    AVStream *st2 = s->streams[ next->stream_index];
    int64_t a = st2->time_base.num * (int64_t)st ->time_base.den;
    int64_t b = st ->time_base.num * (int64_t)st2->time_base.den;
    int64_t dts1 = av_rescale_rnd(pkt->dts, b, a, AV_ROUND_DOWN);
    if (dts1 == next->dts && dts1 == av_rescale_rnd(pkt->dts, b, a, AV_ROUND_UP))
        return pkt->stream_index < next->stream_index;
        return dts1 < next->dts;
}

int av_interleave_packet_per_dts(AVFormatContext *s, AVPacket *out, AVPacket *pkt, int
flush)
{
        AVPacketList *pktl;
        int stream_count = 0;
        int i;

        if(pkt)
        {
                ff_interleave_add_packet(s, pkt, ff_interleave_compare_dts);
        }

        for(i = 0; i < s->nb_streams; i++)
                stream_count += !!s->streams[i]->last_in_packet_buffer;

        if(stream_count && (s->nb_streams == stream_count || flush))
        {
                pktl = s->packet_buffer;
                *out = pktl->pkt;

                s->packet_buffer = pktl->next;
                if(!s->packet_buffer)
                        s->packet_buffer_end = NULL;

                if(s->streams[out->stream_index]->last_in_packet_buffer == pktl)
                        s->streams[out->stream_index]->last_in_packet_buffer = NULL;
                av_freep(&pktl);
                return 1;
        }
        else
        {
                av_init_packet(out);
                return 0;
        }
}

/**
  * Interleave an AVPacket correctly so it can be muxed.
  * @param out the interleaved packet will be output here
  * @param in the input packet
  * @param flush 1 if no further packets are available as input and all
```

```
 *                         remaining packets should be output
 * @return 1 if a packet was output, 0 if no packet could be output,
 *                 < 0 if an error occurred
 */
static int av_interleave_packet(AVFormatContext *s, AVPacket *out, AVPacket *in, int flush)
{
    if(s->oformat->interleave_packet)
        return s->oformat->interleave_packet(s, out, in, flush);
    else
        return av_interleave_packet_per_dts(s, out, in, flush);
}

int av_interleaved_write_frame(AVFormatContext *s, AVPacket *pkt)
{
    AVStream *st = s->streams[ pkt->stream_index];
    int ret;

    //FIXME/XXX/HACK drop zero sized packets
    if(st->codec->codec_type == AVMEDIA_TYPE_AUDIO && pkt->size == 0)
        return 0;

    av_dlog(s, "av_interleaved_write_frame size:%d dts:%"PRId64" pts:%"PRId64"\n",
            pkt->size, pkt->dts, pkt->pts);
    if((ret = compute_pkt_fields2(s, st, pkt)) < 0 && !(s->oformat->flags & 
AVFMT_NOTIMESTAMPS))
        return ret;

    if(pkt->dts == AV_NOPTS_VALUE && !(s->oformat->flags & 
AVFMT_NOTIMESTAMPS))
        return AVERROR(EINVAL);

    for(;;)
    {
        AVPacket opkt;
        int ret = av_interleave_packet(s, &opkt, pkt, 0);
        if(ret <= 0) //FIXME cleanup needed for ret<0 ?
            return ret;

        ret = s->oformat->write_packet(s, &opkt);

        av_free_packet(&opkt);
        pkt = NULL;

        if(ret < 0)
```

```
                return ret;
            if(url_ferror(s->pb))
                return url_ferror(s->pb);
        }
}

int av_write_trailer(AVFormatContext *s)
{
        int ret, i;

        for(;;)
        {
            AVPacket pkt;
            ret = av_interleave_packet(s, &pkt, NULL, 1);
            if(ret < 0) //FIXME cleanup needed for ret<0 ?
                goto fail;
            if(!ret)
                break;

            ret = s->oformat->write_packet(s, &pkt);

            av_free_packet(&pkt);

            if(ret < 0)
                goto fail;
            if(url_ferror(s->pb))
                goto fail;
        }

        if(s->oformat->write_trailer)
            ret = s->oformat->write_trailer(s);
fail:
        if(ret == 0)
            ret = url_ferror(s->pb);
        for(i = 0; i < s->nb_streams; i++)
        {
            av_freep(&s->streams[i]->priv_data);
            av_freep(&s->streams[i]->index_entries);
        }
        av_freep(&s->priv_data);
        return ret;
}

void ff_program_add_stream_index(AVFormatContext *ac, int progid, unsigned int idx)
```

```
{
    int i, j;
    AVProgram *program = NULL;
    void *tmp;

    if (idx >= ac->nb_streams)
    {
        av_log(ac, AV_LOG_ERROR, "stream index %d is not valid\n", idx);
        return;
    }

    for(i = 0; i < ac->nb_programs; i++)
    {
        if(ac->programs[i]->id != progid)
            continue;
        program = ac->programs[i];
        for(j = 0; j < program->nb_stream_indexes; j++)
            if(program->stream_index[j] == idx)
                return;

        tmp = av_realloc(program->stream_index, sizeof(unsigned int) *
(program->nb_stream_indexes + 1));
        if(!tmp)
            return;
        program->stream_index = tmp;
        program->stream_index[program->nb_stream_indexes++] = idx;
        return;
    }
}

static void print_fps(double d, const char *postfix)
{
    uint64_t v = lrintf(d * 100);
    if     (v % 100      ) av_log(NULL, AV_LOG_INFO, ", %3.2f %s", d, postfix);
    else if(v % (100 * 1000)) av_log(NULL, AV_LOG_INFO, ", %1.0f %s", d, postfix);
    else                     av_log(NULL, AV_LOG_INFO, ", %1.0fk %s", d / 1000, postfix);
}

static void dump_metadata(void *ctx, AVMetadata *m, const char *indent)
{
    if(m && !(m->count == 1 && av_metadata_get(m, "language", NULL, 0)))
    {
        AVMetadataTag *tag = NULL;
```

```
            av_log(ctx, AV_LOG_INFO, "%sMetadata:\n", indent);
            while((tag = av_metadata_get(m, "", tag, AV_METADATA_IGNORE_SUFFIX)))
            {
                  if(strcmp("language", tag->key))
                        av_log(ctx,   AV_LOG_INFO,   "%s     %-16s:   %s\n",   indent,   tag->key,
tag->value);
            }
      }
}

/* "user interface" functions */
static void dump_stream_format(AVFormatContext *ic, int i, int index, int is_output)
{
      char buf[256];
      int flags = (is_output ? ic->oformat->flags : ic->iformat->flags);
      AVStream *st = ic->streams[i];
      int g = av_gcd(st->time_base.num, st->time_base.den);
      AVMetadataTag *lang = av_metadata_get(st->metadata, "language", NULL, 0);
      avcodec_string(buf, sizeof(buf), st->codec, is_output);
      av_log(NULL, AV_LOG_INFO, "        Stream #%d.%d", index, i);
      /* the pid is an important information, so we display it */
      /* XXX: add a generic system */
      if (flags & AVFMT_SHOW_IDS)
            av_log(NULL, AV_LOG_INFO, "[0x%x]", st->id);
      if (lang)
            av_log(NULL, AV_LOG_INFO, "(%s)", lang->value);
      av_log(NULL,    AV_LOG_DEBUG,    ",    %d,    %d/%d",    st->codec_info_nb_frames,
st->time_base.num / g, st->time_base.den / g);
      av_log(NULL, AV_LOG_INFO, ": %s", buf);
      if (st->sample_aspect_ratio.num && // default
                  av_cmp_q(st->sample_aspect_ratio, st->codec->sample_aspect_ratio))
      {
            AVRational display_aspect_ratio;
            av_reduce(&display_aspect_ratio.num, &display_aspect_ratio.den,
                          st->codec->width * st->sample_aspect_ratio.num,
                          st->codec->height * st->sample_aspect_ratio.den,
                          1024 * 1024);
            av_log(NULL, AV_LOG_INFO, ", PAR %d:%d DAR %d:%d",
                      st->sample_aspect_ratio.num, st->sample_aspect_ratio.den,
                      display_aspect_ratio.num, display_aspect_ratio.den);
      }
      if(st->codec->codec_type == AVMEDIA_TYPE_VIDEO)
      {
            if(st->avg_frame_rate.den && st->avg_frame_rate.num)
            print_fps(av_q2d(st->avg_frame_rate), "fps");
        if(st->r_frame_rate.den && st->r_frame_rate.num)
            print_fps(av_q2d(st->r_frame_rate), "tbr");
        if(st->time_base.den && st->time_base.num)
            print_fps(1 / av_q2d(st->time_base), "tbn");
        if(st->codec->time_base.den && st->codec->time_base.num)
            print_fps(1 / av_q2d(st->codec->time_base), "tbc");
    }
    if (st->disposition & AV_DISPOSITION_DEFAULT)
        av_log(NULL, AV_LOG_INFO, " (default)");
    if (st->disposition & AV_DISPOSITION_DUB)
        av_log(NULL, AV_LOG_INFO, " (dub)");
    if (st->disposition & AV_DISPOSITION_ORIGINAL)
        av_log(NULL, AV_LOG_INFO, " (original)");
    if (st->disposition & AV_DISPOSITION_COMMENT)
        av_log(NULL, AV_LOG_INFO, " (comment)");
    if (st->disposition & AV_DISPOSITION_LYRICS)
        av_log(NULL, AV_LOG_INFO, " (lyrics)");
    if (st->disposition & AV_DISPOSITION_KARAOKE)
        av_log(NULL, AV_LOG_INFO, " (karaoke)");
    if (st->disposition & AV_DISPOSITION_FORCED)
        av_log(NULL, AV_LOG_INFO, " (forced)");
    if (st->disposition & AV_DISPOSITION_HEARING_IMPAIRED)
        av_log(NULL, AV_LOG_INFO, " (hearing impaired)");
    if (st->disposition & AV_DISPOSITION_VISUAL_IMPAIRED)
        av_log(NULL, AV_LOG_INFO, " (visual impaired)");
    if (st->disposition & AV_DISPOSITION_CLEAN_EFFECTS)
        av_log(NULL, AV_LOG_INFO, " (clean effects)");
    av_log(NULL, AV_LOG_INFO, "\n");
    dump_metadata(NULL, st->metadata, "    ");
}

#if FF_API_DUMP_FORMAT
void dump_format(AVFormatContext *ic,
                 int index,
                 const char *url,
                 int is_output)
{
    av_dump_format(ic, index, url, is_output);
}
#endif

void av_dump_format(AVFormatContext *ic,
                    int index,
                     const char *url,
                     int is_output)
{
    int i;
    uint8_t *printed = av_mallocz(ic->nb_streams);
    if (ic->nb_streams && !printed)
        return;

    av_log(NULL, AV_LOG_INFO, "%s #%d, %s, %s '%s':\n",
           is_output ? "Output" : "Input",
           index,
           is_output ? ic->oformat->name : ic->iformat->name,
           is_output ? "to" : "from", url);
    dump_metadata(NULL, ic->metadata, "    ");
    if (!is_output)
    {
        av_log(NULL, AV_LOG_INFO, "    Duration: ");
        if (ic->duration != AV_NOPTS_VALUE)
        {
            int hours, mins, secs, us;
            secs = ic->duration / AV_TIME_BASE;
            us = ic->duration % AV_TIME_BASE;
            mins = secs / 60;
            secs %= 60;
            hours = mins / 60;
            mins %= 60;
            av_log(NULL, AV_LOG_INFO, "%02d:%02d:%02d.%02d", hours, mins, secs,
                   (100 * us) / AV_TIME_BASE);
        }
        else
        {
            av_log(NULL, AV_LOG_INFO, "N/A");
        }
        if (ic->start_time != AV_NOPTS_VALUE)
        {
            int secs, us;
            av_log(NULL, AV_LOG_INFO, ", start: ");
            secs = ic->start_time / AV_TIME_BASE;
            us = abs(ic->start_time % AV_TIME_BASE);
            av_log(NULL, AV_LOG_INFO, "%d.%06d",
                   secs, (int)av_rescale(us, 1000000, AV_TIME_BASE));
        }
        av_log(NULL, AV_LOG_INFO, ", bitrate: ");
        if (ic->bit_rate)
        {
            av_log(NULL, AV_LOG_INFO, "%d kb/s", ic->bit_rate / 1000);
        }
        else
        {
            av_log(NULL, AV_LOG_INFO, "N/A");
        }
        av_log(NULL, AV_LOG_INFO, "\n");
    }
    for (i = 0; i < ic->nb_chapters; i++)
    {
        AVChapter *ch = ic->chapters[i];
        av_log(NULL, AV_LOG_INFO, "    Chapter #%d.%d: ", index, i);
        av_log(NULL, AV_LOG_INFO, "start %f, ", ch->start * av_q2d(ch->time_base));
        av_log(NULL,    AV_LOG_INFO,    "end    %f\n",          ch->end          *
av_q2d(ch->time_base));

        dump_metadata(NULL, ch->metadata, "    ");
    }
    if(ic->nb_programs)
    {
        int j, k, total = 0;
        for(j = 0; j < ic->nb_programs; j++)
        {
            AVMetadataTag *name = av_metadata_get(ic->programs[j]->metadata,
                                                  "name", NULL, 0);
            av_log(NULL, AV_LOG_INFO, "  Program %d %s\n", ic->programs[j]->id,
                   name ? name->value : "");
            dump_metadata(NULL, ic->programs[j]->metadata, "    ");
            for(k = 0; k < ic->programs[j]->nb_stream_indexes; k++)
            {
                dump_stream_format(ic,     ic->programs[j]->stream_index[k],     index,
is_output);
                printed[ic->programs[j]->stream_index[k]] = 1;
            }
            total += ic->programs[j]->nb_stream_indexes;
        }
        if (total < ic->nb_streams)
            av_log(NULL, AV_LOG_INFO, "  No Program\n");
    }
    for(i = 0; i < ic->nb_streams; i++)
        if (!printed[i])
            dump_stream_format(ic, i, index, is_output);
    av_free(printed);
}

#if FF_API_PARSE_FRAME_PARAM
#include "libavutil/parseutils.h"

int parse_image_size(int *width_ptr, int *height_ptr, const char *str)
{
    return av_parse_video_size(width_ptr, height_ptr, str);
}

int parse_frame_rate(int *frame_rate_num, int *frame_rate_den, const char *arg)
{
    AVRational frame_rate;
    int ret = av_parse_video_rate(&frame_rate, arg);
    *frame_rate_num = frame_rate.num;
    *frame_rate_den = frame_rate.den;
    return ret;
}
#endif



//
//#if defined(_MSC_VER) || defined(_WINDOWS_)
//#include <time.h>
//#if !defined(_WINSOCK2API_) && !defined(_WINSOCKAPI_)
//struct timeval
//{
//    long tv_sec;
//    long tv_usec;
//};
//#endif
//#else
//#include <sys/time.h>
//#endif
//#if defined(_MSC_VER) || defined(_WINDOWS_)
//static int gettimeofday(struct timeval* tv)
//{
//    union {
//        long long ns100;
//        FILETIME ft;
//    } now;
//
```

```
//	GetSystemTimeAsFileTime (&now.ft);
//	tv->tv_usec = (long) ((now.ns100 / 10LL) % 1000000LL);
//	tv->tv_sec = (long) ((now.ns100 - 116444736000000000LL) / 10000000LL);
//	return (0);
//}
//#endif


#include <time.h>

typedef union
{
	long long ns100;
	FILETIME ft;
} NOWTime;
static int gettimeofday(struct timeval *tv)
{
	NOWTime now;

	GetSystemTimeAsFileTime (&now.ft);
	tv->tv_usec = (long) ((now.ns100 / 10LL) % 1000000LL);
	tv->tv_sec = (long) ((now.ns100 - 116444736000000000LL) / 10000000LL);
	return (0);
}



int64_t av_gettime(void)
{
	struct timeval tv;
	gettimeofday(&tv, NULL);
	return (int64_t)tv.tv_sec * 1000000 + tv.tv_usec;
}

uint64_t ff_ntp_time(void)
{
	return (av_gettime() / 1000) * 1000 + NTP_OFFSET_US;
}

#if FF_API_PARSE_DATE
#include "libavutil/parseutils.h"

int64_t parse_date(const char *timestr, int duration)
{
    int64_t timeval;
    av_parse_time(&timeval, timestr, duration);
    return timeval;
}
#endif

#if FF_API_FIND_INFO_TAG
#include "libavutil/parseutils.h"

int find_info_tag(char *arg, int arg_size, const char *tag1, const char *info)
{
    return av_find_info_tag(arg, arg_size, tag1, info);
}
#endif

int av_get_frame_filename(char *buf, int buf_size,
                          const char *path, int number)
{
    const char *p;
    char *q, buf1[20], c;
    int nd, len, percentd_found;

    q = buf;
    p = path;
    percentd_found = 0;
    for(;;)
    {
        c = *p++;
        if (c == '\0')
            break;
        if (c == '%')
        {
            do
            {
                nd = 0;
                while (isdigit(*p))
                {
                    nd = nd * 10 + *p++ - '0';
                }
                c = *p++;
            }
            while (isdigit(c));

            switch(c)
            {
            case '%':
                goto addchar;
            case 'd':
                if (percentd_found)
                    goto fail;
                percentd_found = 1;
                snprintf(buf1, sizeof(buf1), "%0*d", nd, number);
                len = strlen(buf1);
                if ((q - buf + len) > buf_size - 1)
                    goto fail;
                memcpy(q, buf1, len);
                q += len;
                break;
            default:
                goto fail;
            }
        }
        else
        {
addchar:
            if ((q - buf) < buf_size - 1)
                *q++ = c;
        }
    }
    if (!percentd_found)
        goto fail;
    *q = '\0';
    return 0;
fail:
    *q = '\0';
    return -1;
}

static void hex_dump_internal(void *avcl, FILE *f, int level, uint8_t *buf, int size)
{
    int len, i, j, c;
#undef fprintf
#define PRINT(...) do { if (!f) av_log(avcl, level, __VA_ARGS__); else fprintf(f,
__VA_ARGS__); } while(0)

    for(i = 0; i < size; i += 16)
    {
        len = size - i;
        if (len > 16)
            len = 16;
        PRINT("%08x ", i);
        for(j = 0; j < 16; j++)
        {
            if (j < len)
                PRINT(" %02x", buf[i+j]);
            else
                PRINT("   ");
        }
        PRINT(" ");
        for(j = 0; j < len; j++)
        {
            c = buf[i+j];
            if (c < ' ' || c > '~')
                c = '.';
            PRINT("%c", c);
        }
        PRINT("\n");
    }
#undef PRINT
}

void av_hex_dump(FILE *f, uint8_t *buf, int size)
{
    hex_dump_internal(NULL, f, 0, buf, size);
}

void av_hex_dump_log(void *avcl, int level, uint8_t *buf, int size)
{
    hex_dump_internal(avcl, NULL, level, buf, size);
}

static void pkt_dump_internal(void *avcl, FILE *f, int level, AVPacket *pkt, int dump_payload,
AVRational time_base)
{
#undef fprintf
#define PRINT(...) do { if (!f) av_log(avcl, level, __VA_ARGS__); else fprintf(f,
__VA_ARGS__); } while(0)
    PRINT("stream #%d:\n", pkt->stream_index);
    PRINT("  keyframe=%d\n", ((pkt->flags & AV_PKT_FLAG_KEY) != 0));
    PRINT("  duration=%0.3f\n", pkt->duration * av_q2d(time_base));
    /* DTS is _always_ valid after av_read_frame() */
    PRINT("  dts=");
    if (pkt->dts == AV_NOPTS_VALUE)
        PRINT("N/A");
    else
        PRINT("%0.3f", pkt->dts * av_q2d(time_base));
    /* PTS may not be known if B-frames are present. */
    PRINT("    pts=");
    if (pkt->pts == AV_NOPTS_VALUE)
        PRINT("N/A");
    else
        PRINT("%0.3f", pkt->pts * av_q2d(time_base));
    PRINT("\n");
    PRINT("    size=%d\n", pkt->size);
#undef PRINT
    if (dump_payload)
        av_hex_dump(f, pkt->data, pkt->size);
}

#if FF_API_PKT_DUMP
void av_pkt_dump(FILE *f, AVPacket *pkt, int dump_payload)
{
    AVRational tb = { 1, AV_TIME_BASE };
    pkt_dump_internal(NULL, f, 0, pkt, dump_payload, tb);
}
#endif

void av_pkt_dump2(FILE *f, AVPacket *pkt, int dump_payload, AVStream *st)
{
    pkt_dump_internal(NULL, f, 0, pkt, dump_payload, st->time_base);
}

#if FF_API_PKT_DUMP
void av_pkt_dump_log(void *avcl, int level, AVPacket *pkt, int dump_payload)
{
    AVRational tb = { 1, AV_TIME_BASE };
    pkt_dump_internal(avcl, NULL, level, pkt, dump_payload, tb);
}
#endif

void av_pkt_dump_log2(void *avcl, int level, AVPacket *pkt, int dump_payload,
                      AVStream *st)
{
    pkt_dump_internal(avcl, NULL, level, pkt, dump_payload, st->time_base);
}
```

```
#if FF_API_URL_SPLIT
attribute_deprecated
void ff_url_split(char *proto, int proto_size,
                  char *authorization, int authorization_size,
                  char *hostname, int hostname_size,
                  int *port_ptr,
                  char *path, int path_size,
                  const char *url)
{
    av_url_split(proto, proto_size,
                 authorization, authorization_size,
                 hostname, hostname_size,
                 port_ptr,
                 path, path_size,
                 url);
}
#endif

void av_url_split(char *proto, int proto_size,
                  char *authorization, int authorization_size,
                  char *hostname, int hostname_size,
                  int *port_ptr,
                  char *path, int path_size,
                  const char *url)
{
    const char *p, *ls, *at, *col, *brk;

    if (port_ptr)               *port_ptr = -1;
    if (proto_size > 0)         proto[0] = 0;
    if (authorization_size > 0) authorization[0] = 0;
    if (hostname_size > 0)      hostname[0] = 0;
    if (path_size > 0)          path[0] = 0;

    /* parse protocol */
    if ((p = strchr(url, ':')))
    {
        av_strlcpy(proto, url, FFMIN(proto_size, p + 1 - url));
        p++; /* skip ':' */
        if (*p == '/') p++;
        if (*p == '/') p++;
    }
    else
    {
        /* no protocol means plain filename */
```

```
            av_strlcpy(path, url, path_size);
            return;
        }

        /* separate path from hostname */
        ls = strchr(p, '/');
        if(!ls)
            ls = strchr(p, '?');
        if(ls)
            av_strlcpy(path, ls, path_size);
        else
            ls = &p[strlen(p)]; // XXX

        /* the rest is hostname, use that to parse auth/port */
        if (ls != p)
        {
            /* authorization (user[:pass]@hostname) */
            if ((at = strchr(p, '@')) && at < ls)
            {
                av_strlcpy(authorization, p,
                           FFMIN(authorization_size, at + 1 - p));
                p = at + 1; /* skip '@' */
            }

            if (*p == '[' && (brk = strchr(p, ']')) && brk < ls)
            {
                /* [host]:port */
                av_strlcpy(hostname, p + 1,
                           FFMIN(hostname_size, brk - p));
                if (brk[1] == ':' && port_ptr)
                    *port_ptr = atoi(brk + 2);
            }
            else if ((col = strchr(p, ':')) && col < ls)
            {
                av_strlcpy(hostname, p,
                           FFMIN(col + 1 - p, hostname_size));
                if (port_ptr) *port_ptr = atoi(col + 1);
            }
            else
                av_strlcpy(hostname, p,
                           FFMIN(ls + 1 - p, hostname_size));
        }
    }
```

```
char *ff_data_to_hex(char *buff, const uint8_t *src, int s, int lowercase)
{
    int i;
    static const char hex_table_uc[16] = { '0', '1', '2', '3',
                                           '4', '5', '6', '7',
                                           '8', '9', 'A', 'B',
                                           'C', 'D', 'E', 'F'
                                         };
    static const char hex_table_lc[16] = { '0', '1', '2', '3',
                                           '4', '5', '6', '7',
                                           '8', '9', 'a', 'b',
                                           'c', 'd', 'e', 'f'
                                         };
    const char *hex_table = lowercase ? hex_table_lc : hex_table_uc;

    for(i = 0; i < s; i++)
    {
        buff[i * 2]     = hex_table[src[i] >> 4];
        buff[i * 2 + 1] = hex_table[src[i] & 0xF];
    }

    return buff;
}

int ff_hex_to_data(uint8_t *data, const char *p)
{
    int c, len, v;

    len = 0;
    v = 1;
    for (;;)
    {
        p += strspn(p, SPACE_CHARS);
        if (*p == '\0')
            break;
        c = toupper((unsigned char) * p++);
        if (c >= '0' && c <= '9')
            c = c - '0';
        else if (c >= 'A' && c <= 'F')
            c = c - 'A' + 10;
        else
            break;
        v = (v << 4) | c;
        if (v & 0x100)
        {
            if (data)
                data[len] = v;
            len++;
            v = 1;
        }
    }
    return len;
}

//通过参数设置 AVStream *s 的 time_base，pts_wrap_bits
void av_set_pts_info(AVStream *s, int pts_wrap_bits,
                     unsigned int pts_num, unsigned int pts_den)
{
    AVRational new_tb;
    if(av_reduce(&new_tb.num, &new_tb.den, pts_num, pts_den, INT_MAX))
    {
        if(new_tb.num != pts_num)
            av_log(NULL, AV_LOG_DEBUG, "st:%d removing common factor %d from
timebase\n", s->index, pts_num / new_tb.num);
    }
    else
        av_log(NULL, AV_LOG_WARNING, "st:%d has too large timebase, reducing\n",
s->index);

    if(new_tb.num <= 0 || new_tb.den <= 0)
    {
        av_log(NULL, AV_LOG_ERROR, "Ignoring attempt to set invalid timebase for
st:%d\n", s->index);
        return;
    }
    //对 AVStream *s 设置 time_base
    s->time_base = new_tb;
    //对 AVStream *s 设置 pts_wrap_bits
    s->pts_wrap_bits = pts_wrap_bits;
}

int ff_url_join(char *str, int size, const char *proto,
                const char *authorization, const char *hostname,
                int port, const char *fmt, ...)
{
#if CONFIG_NETWORK
    struct addrinfo hints, *ai;
#endif
```

```
	str[0] = '\0';
	if (proto)
		av_strlcatf(str, size, "%s://", proto);
	if (authorization && authorization[0])
		av_strlcatf(str, size, "%s@", authorization);
#if CONFIG_NETWORK && defined(AF_INET6)
	/* Determine if hostname is a numerical IPv6 address,
	 * properly escape it within [] in that case. */
	memset(&hints, 0, sizeof(hints));
	hints.ai_flags = AI_NUMERICHOST;
	if (!getaddrinfo(hostname, NULL, &hints, &ai))
	{
		if (ai->ai_family == AF_INET6)
		{
			av_strlcat(str, "[", size);
			av_strlcat(str, hostname, size);
			av_strlcat(str, "]", size);
		}
		else
		{
			av_strlcat(str, hostname, size);
		}
		freeaddrinfo(ai);
	}
	else
#endif
		/* Not an IPv6 address, just output the plain string. */
		av_strlcat(str, hostname, size);

	if (port >= 0)
		av_strlcatf(str, size, ":%d", port);
	if (fmt)
	{
		va_list vl;
		int len = strlen(str);

		va_start(vl, fmt);
		vsnprintf(str + len, size > len ? size - len : 0, fmt, vl);
		va_end(vl);
	}
	return strlen(str);
}
```

```
int ff_write_chained(AVFormatContext *dst, int dst_stream, AVPacket *pkt,
                     AVFormatContext *src)
{
    AVPacket local_pkt;

    local_pkt = *pkt;
    local_pkt.stream_index = dst_stream;
    if (pkt->pts != AV_NOPTS_VALUE)
        local_pkt.pts = av_rescale_q(pkt->pts,
                                     src->streams[pkt->stream_index]->time_base,
                                     dst->streams[dst_stream]->time_base);
    if (pkt->dts != AV_NOPTS_VALUE)
        local_pkt.dts = av_rescale_q(pkt->dts,
                                     src->streams[pkt->stream_index]->time_base,
                                     dst->streams[dst_stream]->time_base);
    return av_write_frame(dst, &local_pkt);
}

void ff_parse_key_value(const char *str, ff_parse_key_val_cb callback_get_buf,
                        void *context)
{
    const char *ptr = str;

    /* Parse key=value pairs. */
    for (;;)
    {
        const char *key;
        char *dest = NULL, *dest_end;
        int key_len, dest_len = 0;

        /* Skip whitespace and potential commas. */
        while (*ptr && (isspace(*ptr) || *ptr == ','))
            ptr++;
        if (!*ptr)
            break;

        key = ptr;

        if (!(ptr = strchr(key, '=')))
            break;
        ptr++;
        key_len = ptr - key;

        callback_get_buf(context, key, key_len, &dest, &dest_len);
        dest_end = dest + dest_len - 1;

        if (*ptr == '\"')
        {
            ptr++;
            while (*ptr && *ptr != '\"')
            {
                if (*ptr == '\\')
                {
                    if (!ptr[1])
                        break;
                    if (dest && dest < dest_end)
                        *dest++ = ptr[1];
                    ptr += 2;
                }
                else
                {
                    if (dest && dest < dest_end)
                        *dest++ = *ptr;
                    ptr++;
                }
            }
            if (*ptr == '\"')
                ptr++;
        }
        else
        {
            for (; *ptr && !(isspace(*ptr) || *ptr == ','); ptr++)
                if (dest && dest < dest_end)
                    *dest++ = *ptr;
        }
        if (dest)
            *dest = 0;
    }
}

int ff_find_stream_index(AVFormatContext *s, int id)
{
    int i;
    for (i = 0; i < s->nb_streams; i++)
    {
        if (s->streams[i]->id == id)
            return i;
    }
```

```
        return -1;
}

void ff_make_absolute_url(char *buf, int size, const char *base,
                                                    const char *rel)
{
        char *sep;
        /* Absolute path, relative to the current server */
        if (base && strstr(base, "://") && rel[0] == '/')
        {
                if (base != buf)
                        av_strlcpy(buf, base, size);
                sep = strstr(buf, "://");
                if (sep)
                {
                        sep += 3;
                        sep = strchr(sep, '/');
                        if (sep)
                                *sep = '\0';
                }
                av_strlcat(buf, rel, size);
                return;
        }
        /* If rel actually is an absolute url, just copy it */
        if (!base || strstr(rel, "://") || rel[0] == '/')
        {
                av_strlcpy(buf, rel, size);
                return;
        }
        if (base != buf)
                av_strlcpy(buf, base, size);
        /* Remove the file name from the base url */
        sep = strrchr(buf, '/');
        if (sep)
                sep[1] = '\0';
        else
                buf[0] = '\0';
        while (av_strstart(rel, "../", NULL) && sep)
        {
                /* Remove the path delimiter at the end */
                sep[0] = '\0';
                sep = strrchr(buf, '/');
                /* If the next directory name to pop off is "..", break here */
                if (!strcmp(sep ? &sep[1] : buf, ".."))
```

```
            {
                /* Readd the slash we just removed */
                av_strlcat(buf, "/", size);
                break;
            }
            /* Cut off the directory name */
            if (sep)
                sep[1] = '\0';
            else
                buf[0] = '\0';
            rel += 3;
        }
        av_strlcat(buf, rel, size);
}
```

如果全面开始分析 utils.c，utils.h 文件，就需要很大的精力，而且显得很散乱，因为笔者发现其中设计到很多的数据结构，抽象和具体的都有，而且很多结构体中包含大量的数据成员，这些柔和在一起对代码的理解造成很大困扰。

所以对于 ffmpeg 库的 libavformat 成员库，决定采取一个整套的流程来分析，这里将展现两套分析流程，一套是 avi 文件的转码读取输入流程，另一套是 SDP 文件的转码输入流程。通过这两套流程的解析，从而最终完成对 utils.c, utils.h 的分析。

2.5.1 AVI 文件的转码输入流程解析

对于 AVI 文件的转码输入流程，它的起点是一个 AVI 文件的路径，需要准备好的还有关于转码程序中对输入的设置变量（在本小结中简称设置变量），现在就开始对这个流程进行解析：

其一：通过设置变量的输入格式名字寻找一个输入格式；

通过函数 AVInputFormat *av_find_input_format(const char *short_name)来实现，具体的代码解析在 libavformat\utils.c，libavformat\utils.h 中。在前面的代码中，我们知道了 AVInputFormat 对输入源的重要性，现在我们更多的关心 AVInputFormat 内部的各个成员是如何被设置的。虽然对于 AVI 文件的输入，设置变量的输入格式名字一般为空，不过这里，我们可以假设设置变量的输入格式名字为“avi”，得到的 AVInputFormat 的值为：

```
AVInputFormat ff_avi_demuxer =
{
        "avi",
            ===const char *name;//名字
        NULL_IF_CONFIG_SMALL("AVI format"),
            ===const char *long_name;//长名字
        sizeof(AVIContext),
            ===int priv_data_size;//数据长度
        avi_probe,
            ===int (*read_probe)(AVProbeData *);
            //检测是为指定媒体文件类型的可能性，返回值在 0-100 的范围
        avi_read_header,
            ===int (*read_header)(struct AVFormatContext *,
```

AVFormatParameters *ap);
avi_read_packet,
===int (*read_packet)(struct AVFormatContext *,
AVPacket *pkt);
avi_read_close,
===int (*read_close)(struct AVFormatContext *);
avi_read_seek,
===attribute_deprecated int (*read_seek)(
struct AVFormatContext *,
int stream_index,
int64_t timestamp,
int flags);
};

通过这个赋值过程，我们初步明白了 AVInputFormat 的值是在这一步是如何被设置的，要知道其他没有被显式设置的值，就是默认被设置为 0.

其二：创造一个 AVFormatContext，并设置默认值
在这里操作的对象是 AVFormatContext，操作的函数是
AVFormatContext *avformat_alloc_context(void), 另外设置默认值的函数是：
void av_opt_set_defaults(void *s)。这两个函数，笔者在前面的章节中都进行过解析，现在重点关注的是 AVFormatContext 的成员是如何被赋值的。

static const AVOption options[] =
{
{"probesize", "set probing size", OFFSET(probesize), FF_OPT_TYPE_INT, 5000000, 32, INT_MAX, D},
===unsigned int probesize;
{"muxrate", "set mux rate", OFFSET(mux_rate), FF_OPT_TYPE_INT, DEFAULT, 0, INT_MAX, E},
===int mux_rate;
{"packetsize", "set packet size", OFFSET(packet_size), FF_OPT_TYPE_INT, DEFAULT, 0, INT_MAX, E},
===unsigned int packet_size;
{"fflags", NULL, OFFSET(flags), FF_OPT_TYPE_FLAGS, DEFAULT, INT_MIN, INT_MAX, D | E, "fflags"},
===int flags;
{"ignidx", "ignore index", 0, FF_OPT_TYPE_CONST, AVFMT_FLAG_IGNIDX, INT_MIN, INT_MAX, D, "fflags"},
=== NULL;
{"genpts", "generate pts", 0, FF_OPT_TYPE_CONST, AVFMT_FLAG_GENPTS, INT_MIN, INT_MAX, D, "fflags"},
=== NULL;
{"nofillin", "do not fill in missing values that can be exactly calculated", 0, FF_OPT_TYPE_CONST, AVFMT_FLAG_NOFILLIN, INT_MIN, INT_MAX, D, "fflags"},
=== NULL;

```
    {"noparse", "disable AVParsers, this needs nofillin too", 0, FF_OPT_TYPE_CONST, AVFMT_FLAG_NOPARSE, INT_MIN, INT_MAX, D, "fflags"},
        === NULL;
    {"igndts", "ignore dts", 0, FF_OPT_TYPE_CONST, AVFMT_FLAG_IGNDTS, INT_MIN, INT_MAX, D, "fflags"},
        === NULL;
    {"rtphint", "add rtp hinting", 0, FF_OPT_TYPE_CONST, AVFMT_FLAG_RTP_HINT, INT_MIN, INT_MAX, E, "fflags"},
        === NULL;
#if FF_API_OLD_METADATA
    {"track", " set the track number", OFFSET(track), FF_OPT_TYPE_INT, DEFAULT, 0, INT_MAX, E},
        ===attribute_deprecated int track; // track number, 0 if none
    {"year", "set the year", OFFSET(year), FF_OPT_TYPE_INT, DEFAULT, INT_MIN, INT_MAX, E},
        ===attribute_deprecated int year;    // ID3 year, 0 if none
#endif
    {"analyzeduration", "how many microseconds are analyzed to estimate duration", OFFSET(max_analyze_duration), FF_OPT_TYPE_INT, 5 * AV_TIME_BASE, 0, INT_MAX, D},
        === NULL;
    {"cryptokey", "decryption key", OFFSET(key), FF_OPT_TYPE_BINARY, 0, 0, 0, D},
        ===const uint8_t *key;
    {"indexmem", "max memory used for timestamp index (per stream)", OFFSET(max_index_size), FF_OPT_TYPE_INT, 1 << 20, 0, INT_MAX, D},
        ===unsigned int max_index_size;
    {"rtbufsize", "max memory used for buffering real-time frames", OFFSET(max_picture_buffer), FF_OPT_TYPE_INT, 3041280, 0, INT_MAX, D}, /* defaults to 1s of 15fps 352x288 YUYV422 video */
        ===unsigned int max_picture_buffer;
    {"fdebug", "print specific debug info", OFFSET(debug), FF_OPT_TYPE_FLAGS, DEFAULT, 0, INT_MAX, E | D, "fdebug"},
        ===int debug;
    {"ts", NULL, 0, FF_OPT_TYPE_CONST, FF_FDEBUG_TS, INT_MIN, INT_MAX, E | D, "fdebug"},
        === NULL;
    {"max_delay", "maximum muxing or demuxing delay in microseconds", OFFSET(max_delay), FF_OPT_TYPE_INT, DEFAULT, 0, INT_MAX, E | D},
        ===int max_delay;
    {NULL},
};
```

以上的代码就是对 AVFormatContext 设置默认值的依据，当然其中有一些是不能具体设置到成员变量的选项，目前先不要管它。

虽然设置了 AVFormatContext 的默认值，但是在设置变量里面有一些值也是需要当作默

认值来设置的。我们看看是些什么样的值，这些值又是如何设置给 **AVFormatContext** 的.

第一个问题有哪些设置变量的值要当作默认值来设置的？ 首先便是 **video_codec_name**，**audio_codec_name**，**subtitle_codec_name**，通过解码器（输入者需要解码）的名字找到解码器的 **ID**，然后赋值给 **AVFormatContext** 的成员：

```
enum CodecID video_codec_id;//视频编解码器 ID
~~~video_codec_name
enum CodecID audio_codec_id;//音频编解码器 ID
~~~ audio_codec_name
enum CodecID subtitle_codec_id;//字幕编解码器 ID
~~~ subtitle_codec_name
```

另外 **AVFormatContext** 还有一个重要的成员在此时被赋值了(将 **flags** 设置为非阻塞模式)：**AVFormatContext->flags |= AVFMT_FLAG_NONBLOCK;**除此之外，还有一个在转码程序使用很多的函数：**void set_context_opts(void *ctx, void *opts_ctx, int flags, AVCodec *codec)**，下面将进行适当展开，分析一下 **set_context_opts** 的运行机制。

第二个问题一些值是如何被赋值的？这也就是 **set_context_opts** 的运行机制是什么样的？

对 **set_context_opts** 的运行机制，最大的困惑是为什么需要一个类，还包括那么多成员？

```
class MediaTran_opt
{
private:
        SwsContext *sws_opts;
        AVFormatContext *avformat_opts;//格式的通用变量
        AVCodecContext *avcodec_opts[AVMEDIA_TYPE_NB];
                                        //编解码的通用变量数组
};
MediaTran_opt::MediaTran_opt(void)
{
        memset(avcodec_opts, 0, sizeof(avcodec_opts));

#if CONFIG_SWSCALE
       sws_opts = sws_getContext(16, 16,
       (PixelFormat)0, 16, 16, (PixelFormat)0, SWS_BICUBIC, NULL, NULL, NULL);
        ASSERTTRY(NULL != sws_opts);
#endif
        avformat_opts = avformat_alloc_context();
        ASSERTTRY(NULL != avformat_opts);

        for (int i = 0; i < AVMEDIA_TYPE_NB; i++)
        {
               avcodec_opts[i] = avcodec_alloc_context2((AVMediaType)i);
                ASSERTTRY(NULL != avcodec_opts[i]);
        }
}
```

看了类的成员和构造函数，为什么要这样呢？不就是使用一下 **set_context_opts** 函数吗？为什么需要构造那么多成员数据呢？不用着急，我们还是来看一下函数 **set_context_opts**：

```
BOOL MediaTran_opt::set_context_opts(void *ctx,
                                     void *const opts_ctx,
                                     int flags,
                                       AVCodec *codec)
{
       void *priv_ctx = NULL;
       if(!strcmp("AVCodecContext", (*(AVClass **)ctx)->class_name))
       {
              AVCodecContext *avctx = (AVCodecContext *)ctx;
              if(codec && codec->priv_class && avctx->priv_data)
              {
                     priv_ctx = avctx->priv_data;
              }
       }
       else if (!strcmp("AVFormatContext", (*(AVClass **)ctx)->class_name))
       {
              AVFormatContext *avctx = (AVFormatContext *)ctx;
              if (avctx->oformat && avctx->oformat->priv_class)
              {
                     priv_ctx = avctx->priv_data;
              }
       }

       for(int i = 0; i < opt_name_count; i++)
       {
              char buf[STRING_MAXSIZE];
              const AVOption *opt;
              const char *str = av_get_string(opts_ctx, opt_names[i], &opt, buf, sizeof(buf));//注释 1
              // if an option with name opt_names[i] is present in opts_ctx then str is non-NULL
              if(str && ((opt->flags & flags) == flags))
               {
                     av_set_string3(ctx, opt_names[i], str, 1, NULL);//注释 2
               }
              //We need to use a differnt system to pass options to the private context because
              //    it is not known which codec and thus context
             // kind that will be when parsing options
              //    we thus use opt_values directly instead of opts_ctx
               if(!str && priv_ctx)
               {
                      if (av_find_opt(priv_ctx, opt_names[i], NULL, flags, flags))
                      {
                             av_set_string3(priv_ctx, opt_names[i], opt_values[i], 0, NULL);
                      }
               }
```

```
    }
    return TRUE;
}
```
在 set_context_opts 代码中使用类的那些成员的主要有两个函数 av_get_string 和 av_set_string3，那我们继续深入分析：

注释 1:

```
//获取 obj 的 name 操作项的值的字符串并返回对应的 name 的 AVOption
const char *av_get_string(void *obj,
                          const char *name,
                          const AVOption **o_out,
                          char *buf, int buf_len)
{
    //通过传入的 obj 找到一个 AVOption
    const AVOption *o
        = av_find_opt(obj, name, NULL, 0, 0);
    void *dst;
    uint8_t *bin;
    int len, i;
    //检查 AVOption 是否合法
    if (!o || o->offset <= 0)
    {
        return NULL;
    }
    if (o->type != FF_OPT_TYPE_STRING
        && (!buf || !buf_len))
    {
        return NULL;
    }
    dst = ((uint8_t *)obj) + o->offset;
    if (o_out)
    {
        *o_out = o;
    }
    //填写字符串
    switch (o->type)
    {
    case FF_OPT_TYPE_FLAGS:
        snprintf(buf, buf_len, "0x%08X", *(int *)dst);
        break;
    case FF_OPT_TYPE_INT:
        snprintf(buf, buf_len, "%d" , *(int *)dst);
        break;
    case FF_OPT_TYPE_INT64:
        snprintf(buf, buf_len, "%"PRId64, *(int64_t *)dst);
```

```
            break;
        case FF_OPT_TYPE_FLOAT:
            snprintf(buf, buf_len, "%f" , *(float *)dst);
            break;
        case FF_OPT_TYPE_DOUBLE:
            snprintf(buf, buf_len, "%f" , *(double *)dst);
            break;
        case FF_OPT_TYPE_RATIONAL:
            snprintf(buf, buf_len, "%d/%d",
                ((AVRational *)dst)->num, ((AVRational *)dst)->den);
            break;
        case FF_OPT_TYPE_STRING:
            return *(void **)dst;
        case FF_OPT_TYPE_BINARY:
            len = *(int *)(((uint8_t *)dst) + sizeof(uint8_t *));
            if (len >= (buf_len + 1) / 2)
            {
                return NULL;
            }
            bin = *(uint8_t **)dst;
            for (i = 0; i < len; i++)
            {
                snprintf(buf + i * 2, 3, "%02X", bin[i]);
            }
            break;
        default:
            return NULL;
        }
        return buf;
}
```

通过阅读 **av_get_string** 函数的作用，我们知道必须要有一个 **void *obj**，这个 **void *obj** 就是用 **MediaTran_opt** 的成员来代表的，它的原理是这样的，如果有什么需要设置到某个类型的值，先将它设置到在 **MediaTran_opt** 的同一类型成员身上，然后再通过 **MediaTran_opt** 的同一类型成员获取该值的字符串，然后将该字符串设置到要设置的对象上。在这里，觉得这样的设计是不是有点绕？可以说这种设计是为了专门对付字符串设置方式的转码程序。如果我们需要自己设置某个值到某个对象上，可以直接设置，不用绕这个弯子。

看了 **av_get_string** 函数，我们再接着看 **av_set_string3** 函数，这个主要是设置对象值的操作。

```
int av_set_string3(void *obj,
                   const char *name,
                   const char *val,
                   int alloc,
                   const AVOption **o_out)
{
```

```
int ret;
//通过名字寻找 AVOption
const AVOption *o
        = av_find_opt(obj, name, NULL, 0, 0);
if (o_out)
{
       *o_out = o;
}
if (!o)
{
       return AVERROR(ENOENT);
}
//检查设置的值和 AVOption 是否合法
if (!val || o->offset <= 0)
{
       return AVERROR(EINVAL);
}
if (o->type == FF_OPT_TYPE_BINARY)
{
        //这里有个默认的数据位置：
        //data: obj + o->offset
        //datalen: (int*)((int*)(obj + o->offset) + 1)
       uint8_t **dst =
               (uint8_t **)(((uint8_t *)obj) + o->offset);
       int *lendst = (int *)(dst + 1);
       uint8_t *bin, *ptr;
       int len = strlen(val);
        //释放该值原先的内存
       av_freep(dst);
       *lendst = 0;
        //如果设置的值长度为 0，则返回错误
       if (len & 1)
        {
               return AVERROR(EINVAL);
        }
       len /= 2;
       ptr = bin = av_malloc(len);
       while (*val)
       {
               //将字符串转化为数字
              int a = hexchar2int(*val++);
              int b = hexchar2int(*val++);
              if (a < 0 || b < 0)
              {
                av_free(bin);
                return AVERROR(EINVAL);
            }
            *ptr++ = (a << 4) | b;
        }
        *dst = bin;
        *lendst = len;
        return 0;
    }
    if (o->type != FF_OPT_TYPE_STRING)
    {
        int notfirst = 0;
        for (;;)
        {
            int i;
            char buf[256];
            int cmd = 0;
            double d;

             //跳过'+', '-'字符串
            if (*val == '+' || *val == '-')
             {
                cmd = *(val++);
             }
            for (i = 0;
                 i < sizeof(buf) - 1
                     && val[i]
                     && val[i] != '+'
                     && val[i] != '-';
                 i++)
             {
                buf[i] = val[i];
             }
            buf[i] = 0;
             //buf 为过滤过的字符串
            {
                const AVOption *o_named
                     = av_find_opt(obj, buf, o->unit, 0, 0);
                 //获取要设置的值
                if (o_named
                     && o_named->type == FF_OPT_TYPE_CONST)
                 {
                    d = o_named->default_val;
                 }
                else if (!strcmp(buf, "default"))
                 {
                     d = o->default_val;
                 }
                else if (!strcmp(buf, "max"      ))
                 {
                     d = o->max;
                 }
                else if (!strcmp(buf, "min"      ))
                 {
                     d = o->min;
                 }
                else if (!strcmp(buf, "none"     ))
                 {
                     d = 0;
                 }
                else if (!strcmp(buf, "all"      ))
                 {
                     d = ~0;
                 }
                else
                {
                    int res =
                         av_expr_parse_and_eval(&d,
                             buf,
                             const_names,
                             const_values,
                             NULL,
                             NULL,
                             NULL,
                             NULL,
                             NULL,
                             0,
                             obj);
                    if (res < 0)
                    {
                        av_log(obj,
                             AV_LOG_ERROR,
                             "Unable to parse
                             option value \"%s\"\n", val);
                        return res;
                    }
                }
            }
```

```
            if (o->type == FF_OPT_TYPE_FLAGS)
            {
                if          (cmd == '+')
                 {
                        d = av_get_int(obj, name, NULL) | (int64_t)d;
                 }
                else if (cmd == '-')
                 {
                        d = av_get_int(obj, name, NULL) &~(int64_t)d;
                 }
            }
            else
            {
                if          (cmd == '+')
                 {
                        d = notfirst * av_get_double(obj, name, NULL) + d;
                 }
                else if (cmd == '-')
                 {
                        d = notfirst * av_get_double(obj, name, NULL) - d;
                 }
            }
             //设置值
            if ((ret = av_set_number2(
                     obj, name, d, 1, 1, o_out)) < 0)
             {
                return ret;
             }
            val += i;
            if (!*val)
             {
                return 0;
             }
            notfirst = 1;
        }
        return AVERROR(EINVAL);
}
if (alloc)
{
        av_free(*(void **)(((uint8_t *)obj) + o->offset));
        val = av_strdup(val);
}
//如果上面都没有设置，则在最后这里进行值的设置
memcpy(((uint8_t *)obj) + o->offset, &val, sizeof(val));
```

return 0;

}

实际上 av_set_string3 实际上是可以对任何对象进行值的设置的，只是在函数 set_context_opts 中需要被绕个弯子使用。代码解析到这里，我想对 set_context_opts 的运行机制应该有个了解了吧，在这里再强调一边。

重点：set_context_opts 的运行机制是通过 av_set_string3 将需要设置的值预先设置到相同类型的对象上（因为需要设置的对象还没有创建），当创建好需要的对象后，再从那个预先设置了值的对象身上通过函数 av_get_string 获取该设置值的字符串，将该字符串通过函数 av_set_string3 设置到创建好的对象上。

说明：为了便于本小节数据结构成员的表达，特采用一下方式标识数据结构的成员：例如：AVFormatContext 结构的成员 struct AVInputFormat *iformat，特表示为：

(AVFormatContext) -(struct AVInputFormat *iformat)

其三：通过 AVI 文件设置 AVFormatContext；

前面的两步，设置了一些变量，对于一个 AVI 文件，对 AVFormatContext 的设置主要就是本身的默认值和解码的 ID，现在就来着重看一下一个 AVI 文件对 AVFormatContext 的成员变量进行了怎样的设置，这个设置函数主要是 av_open_input_file。

在 av_open_input_file 函数中设置 AVFormatContext 的成员数据最重要的两个成员是 (AVFormatContext)-(struct AVInputFormat *iformat)和(AVFormatContext)-(AVIOContext *pb)。先解析一下(AVFormatContext)-(AVIOContext *pb)是如何被赋值的，从 avio_open 开始。

计算(AVFormatContext)-(AVIOContext *pb)，首先要设置 (AVFormatContext)-(AVIOContext *pb)-(void *opaque), 此变量即是 URLContext，对于这个 URLContext，通过文件名得到（URLContext)-(struct URLProtocol *prot）,，然后通过 url_alloc_for_protocol 函数对 URLContext 赋值，具体如下：

(AVFormatContext)-(av_class) = &urlcontext_class;

(AVFormatContext)-(filename) = filename;

(AVFormatContext)-(flags) = flags;

(AVFormatContext)-(is_streamed) = 0;

(AVFormatContext)-(max_packet_size) = 0;

以上只是一个初始化内容，接下来就是对 URLContext 的具体设置，在函数 ffurl_connect 中，AVI 文件的协议类型是“FILE”，所以在 ffurl_connect 中使用 file_open 打开，同时在 file_open 中将（URLContext)- (void *priv_data)设置为 open(filename, access, 0666), 在 ffurl_connect 中，有（URLContext)- (int is_connected)= 1。

接下就是在 ffio_fdopen 中对(AVFormatContext)-(AVIOContext *pb)进行设置，代码如下：

(AVFormatContext)-(AVIOContext *pb)-(max_packet_size)
    =(AVFormatContext)-(AVIOContext*pb)-(URLContext)-(max_packet_size);
(AVFormatContext)-(AVIOContext *pb)-(buffer_size)
    = max_packet_size > 0 ? max_packet_size: IO_BUFFER_SIZE;
(AVFormatContext)-(AVIOContext *pb)-(buffer) = av_malloc(buffer_size);
(AVFormatContext)-(AVIOContext *pb)-(buf_ptr) =buffer;
(AVFormatContext)-(AVIOContext *pb)-(write_packet) =ffurl_write;

(AVFormatContext)-(AVIOContext *pb)-(read_packet)= ffurl_read;
(AVFormatContext)-(AVIOContext *pb)-(seek) =ffurl_seek;
(AVFormatContext)-(AVIOContext *pb)-(read_pause) =NULL;
(AVFormatContext)-(AVIOContext *pb)-(read_seek) =NULL;
(AVFormatContext)-(AVIOContext *pb)-(pos) =0;
(AVFormatContext)-(AVIOContext *pb)-(must_flush)=0;
(AVFormatContext)-(AVIOContext *pb)-(eof_reached) =0;
(AVFormatContext)-(AVIOContext *pb)-(error) =0;
(AVFormatContext)-(AVIOContext *pb)-(is_streamed) =0;
(AVFormatContext)-(AVIOContext *pb)-(seekable) =AVIO_SEEKABLE_NORMAL;
(AVFormatContext)-(AVIOContext *pb)-(max_packet_size)=0;
(AVFormatContext)-(AVIOContext *pb)-(update_checksum) =NULL;

通过上面的设置我们完成对(AVFormatContext)-(AVIOContext *pb)的设置，现在我们来看一下另一个(AVFormatContext)-(struct AVInputFormat *iformat)的设置，由于我们已经计算好了(AVFormatContext)-(AVIOContext *pb)，通过读取(AVFormatContext)-(AVIOContext *pb)的数据（保存在 AVProbeData pd），然后通过这段数据来分析输入文件的格式，获得(AVFormatContext)-(struct AVInputFormat *iformat)。

其四：通过 AVI 文件在函数 av_open_input_stream 中设置 AVFormatContext；

对于函数 av_open_input_stream，它是初始化 AVFormatContext 的重要函数，它初始化着大部分的 AVFormatContext 的成员。

(AVFormatContext)-(AVIOContext *pb)-(duration)= AV_NOPTS_VALUE;
(AVFormatContext)-(AVIOContext *pb)-(start_time)= AV_NOPTS_VALUE;
(AVFormatContext)-(AVIOContext *pb)-(filename)= filename;
(AVFormatContext)-(AVIOContext *pb)-(priv_data)= av_mallocz(fmt->priv_data_size);
(AVFormatContext)-(AVIOContext *pb)-(raw_packet_buffer_remaining_size)
    = RAW_PACKET_BUFFER_SIZE;
(AVFormatContext)-(AVIOContext *pb)-(data_offset)= avio_tell(ic->pb);

不过，要具体探知 AVI 文件的内部信息，还需要在 AVI 文件的 demuxer 里去解析，该函数为 static int avi_read_header(AVFormatContext *s, AVFormatParameters *ap)，这个函数是对具体媒体文件解析的一个示例，将在下面的代码中解析：

```
static int avi_read_header(AVFormatContext *s, AVFormatParameters *ap)
{
    //在之前的代码中了解到 s->priv_data = av_mallocz(fmt->priv_data_size);
    //所以在此之前 s->priv_data 就是一片空白
    AVIContext *avi = s->priv_data;
    //不过这个 s->pb 就是已经在前面的代码中已经初始化好
    AVIOContext *pb = s->pb;
    unsigned int tag, tag1, handler;
    int codec_type,
        stream_index,
        frame_period,
        bit_rate;
    unsigned int size;
    int i;
```

```
AVStream *st;
AVIStream *ast = NULL;
int avih_width = 0,
    avih_height = 0;
int amv_file_format = 0;
uint64_t list_end = 0;
int ret;

//在这个函数的具体代码执行前，我们要明白需要追踪的主要有两个变量：
//AVFormatContext *s 和 AVIContext *avi
//对 AVIContext *avi 设置 stream_index
avi->stream_index = -1;
//获取 AVI 文件的 RIFF 数据
if (get_riff(s, pb) < 0)
{
    //对 AVIContext *avi 设置 rtff_end
    return -1;
}
//对 AVIContext *avi 设置 fsize
avi->fsize = avio_size(pb);
//检查 AVIContext *avi 设置 fsize
if(avi->fsize <= 0)
{
    avi->fsize = avi->riff_end == 8 ?
                 INT64_MAX : avi->riff_end;
}
/* first list tag */
stream_index = -1;
codec_type = -1;
frame_period = 0;
for(;;)
{
    if (url_feof(pb))
    {
        goto fail;
    }
    tag = avio_rl32(pb);
    size = avio_rl32(pb);

    print_tag("tag", tag, size);

    switch(tag)
    {
    case MKTAG('L', 'I', 'S', 'T'):
        list_end = avio_tell(pb) + size;
        /* Ignored, except at start of video packets. */
        tag1 = avio_rl32(pb);
        print_tag("list", tag1, 0);
        if (tag1 == MKTAG('m', 'o', 'v', 'i'))
        {
            //检查 AVIContext *avi 设置 movi_list，movi_end
            avi->movi_list = avio_tell(pb) - 4;
            if(size)
            {
                avi->movi_end
                = avi->movi_list + size + (size & 1);
            }
            else
            {
                avi->movi_end = avio_size(pb);
            }
            av_dlog(NULL,
                    "movi end=%"PRIx64"\n",
                    avi->movi_end);
            goto end_of_header;
        }
        else if (tag1 == MKTAG('I', 'N', 'F', 'O'))
        {
            avi_read_info(s, list_end);
        }
        else if (tag1 == MKTAG('n', 'c', 'd', 't'))
        {
            avi_read_nikon(s, list_end);
        }
        break;
    case MKTAG('I', 'D', 'I', 'T'):
    {
        unsigned char date[64] = {0};
        size += (size & 1);
        size -= avio_read(pb,
                                  date, FFMIN(size, sizeof(date) - 1));
        avio_skip(pb, size);
        avi_metadata_creation_time(&s->metadata, date);
        break;
    }
    case MKTAG('d', 'm', 'l', 'h'):
        avi->is_odml = 1;
        avio_skip(pb, size + (size & 1));
        break;
case MKTAG('a', 'm', 'v', 'h'):
        amv_file_format = 1;
case MKTAG('a', 'v', 'i', 'h'):
        /* AVI header */
        /* using frame_period is bad idea */
        frame_period = avio_rl32(pb);
        bit_rate = avio_rl32(pb) * 8;
        avio_rl32(pb);
        //检查 AVIContext *avi 设置 non_interleaved
        avi->non_interleaved
        |= avio_rl32(pb) & AVIF_MUSTUSEINDEX;
        avio_skip(pb, 2 * 4);
        avio_rl32(pb);
        avio_rl32(pb);
        avih_width = avio_rl32(pb);
        avih_height = avio_rl32(pb);

        avio_skip(pb, size - 10 * 4);
        break;
case MKTAG('s', 't', 'r', 'h'):
        /* stream header */
        tag1 = avio_rl32(pb);
        handler = avio_rl32(pb); /* codec tag */

        if(tag1 == MKTAG('p', 'a', 'd', 's'))
        {
                avio_skip(pb, size - 8);
                break;
        }
        else
        {
                //增加一个输入媒体流
                stream_index++;
                 //在 AVFormatContext *s 增加一个媒体流
                st = av_new_stream(s, stream_index);
                if (!st)
                {
                        goto fail;
                }
                 //重点：这里值得好好说一说
                 //为什么在 AVFormatContext *s 中创建了一个媒体流，
                 //马上就需要在 AVIContext *avi 创建一个 AVI 媒体流？
                 //因为这两者是一一对应的关系，
```

```
         //实际上可以引申的说，对说有的具体的媒体类型来说，
         //如果一个对象在 AVFormatContext 中拥有一个媒体流，
         //就一定会在该对象的数据结构中对应这一个该媒体类型的一个媒体流
         //两者联系起来的纽带就是：AVIStream 中的 priv_data 成员
        ast = av_mallocz(sizeof(AVIStream));
        if (!ast)
        {
            goto fail;
        }
        st->priv_data = ast;
    }
    if(amv_file_format)
    {
        tag1 = stream_index ?
                MKTAG('a', 'u', 'd', 's') : MKTAG('v', 'i', 'd', 's');
    }
    print_tag("strh", tag1, -1);
    if(tag1 == MKTAG('i', 'a', 'v', 's')
            || tag1 == MKTAG('i', 'v', 'a', 's'))
    {
        int64_t dv_dur;
        /*
         * After some consideration -- I don't think we
         * have to support anything but DV in type1 AVIs.
         */
        if (s->nb_streams != 1)
        {
            goto fail;
        }
        if (handler != MKTAG('d', 'v', 's', 'd') &&
                handler != MKTAG('d', 'v', 'h', 'd') &&
                handler != MKTAG('d', 'v', 's', 'l'))
        {
            goto fail;
        }
        ast = s->streams[0]->priv_data;
        av_freep(&s->streams[0]->codec->extradata);
        av_freep(&s->streams[0]->codec);
        av_freep(&s->streams[0]);
        s->nb_streams = 0;
        if (CONFIG_DV_DEMUXER)
        {
            avi->dv_demux = dv_init_demux(s);
            if (!avi->dv_demux)
```

```
                {
                        goto fail;
                }
        }
        s->streams[0]->priv_data = ast;
        avio_skip(pb, 3 * 4);
        ast->scale = avio_rl32(pb);
        ast->rate = avio_rl32(pb);
        avio_skip(pb, 4);    /* start time */
        dv_dur = avio_rl32(pb);
        if (ast->scale > 0
                && ast->rate > 0 && dv_dur > 0)
        {
                dv_dur *= AV_TIME_BASE;
                s->duration =
                        av_rescale(dv_dur, ast->scale, ast->rate);
        }
        /*
          * else, leave duration alone; timing estimation in utils.c
          *           will make a guess based on bitrate.
          */
        stream_index = s->nb_streams - 1;
        avio_skip(pb, size - 9 * 4);
        break;
}
assert(stream_index < s->nb_streams);
 //设置 AVStream 中的 codec 中的 stream_codec_tag
st->codec->stream_codec_tag = handler;
avio_rl32(pb); /* flags */
avio_rl16(pb); /* priority */
avio_rl16(pb); /* language */
avio_rl32(pb); /* initial frame */
 //对 AVIStream 设置 scale
ast->scale = avio_rl32(pb);
 //对 AVIStream 设置 rate
ast->rate = avio_rl32(pb);
 //检查 AVIStream 中的 scale 和 rate
if(!(ast->scale && ast->rate))
{
        av_log(s, AV_LOG_WARNING,
                      "scale/rate is %u/%u
                      which is invalid.
                      (This file has been
                      generated by broken
                software.)\n",
                ast->scale,
                ast->rate);
        if(frame_period)
        {
                ast->rate = 1000000;
                ast->scale = frame_period;
        }
        else
        {
                ast->rate = 25;
                ast->scale = 1;
        }
}
 //对 AVIStream 设置 time_base，pts_wrap_bits
av_set_pts_info(st, 64,
                        ast->scale, ast->rate);
 //对 AVIStream 设置 cum_len
ast->cum_len = avio_rl32(pb); /* start */
 //对 AVStream 设置 nb_frames
st->nb_frames = avio_rl32(pb);
 //对 AVStream 设置 start_time
st->start_time = 0;
avio_rl32(pb); /* buffer size */
avio_rl32(pb); /* quality */
 //对 AVIStream 设置 sample_size
ast->sample_size = avio_rl32(pb); /* sample ssize */
 //对 AVIStream 设置 cum_len
ast->cum_len *= FFMAX(1, ast->sample_size);
//av_log(s, AV_LOG_DEBUG,
//"%d %d %d %d\n",
//ast->rate, ast->scale,
//ast->start, ast->sample_size);

switch(tag1)
{
case MKTAG('v', 'i', 'd', 's'):
        codec_type = AVMEDIA_TYPE_VIDEO;
         //对 AVIStream 设置 sample_size
        ast->sample_size = 0;
        break;
case MKTAG('a', 'u', 'd', 's'):
        codec_type = AVMEDIA_TYPE_AUDIO;
        break;
        case MKTAG('t', 'x', 't', 's'):
            codec_type = AVMEDIA_TYPE_SUBTITLE;
            break;
        case MKTAG('d', 'a', 't', 's'):
            codec_type = AVMEDIA_TYPE_DATA;
            break;
        default:
            av_log(s, AV_LOG_ERROR, "unknown stream type %X\n", tag1);
            goto fail;
        }
        if(ast->sample_size == 0)
         {
             //对 AVStream 设置 duration
            st->duration = st->nb_frames;、
         }
         //对 AVIStream 设置 frame_offset
        ast->frame_offset = ast->cum_len;
        avio_skip(pb, size - 12 * 4);
        break;
    case MKTAG('s', 't', 'r', 'f'):
        /* stream header */
        if (stream_index >= (unsigned)s->nb_streams || avi->dv_demux)
        {
            avio_skip(pb, size);
        }
        else
        {
            uint64_t cur_pos = avio_tell(pb);
            if (cur_pos < list_end)
             {
                size = FFMIN(size, list_end - cur_pos);
             }
            st = s->streams[stream_index];
            switch(codec_type)
            {
            case AVMEDIA_TYPE_VIDEO:
                if(amv_file_format)
                {
                     //设置 AVStream 中的 codec 中的 width
                    st->codec->width = avih_width;
                     //设置 AVStream 中的 codec 中的 height
                    st->codec->height = avih_height;
                     //设置 AVStream 中的 codec 中的 codec_type
                    st->codec->codec_type = AVMEDIA_TYPE_VIDEO;
        //设置 AVStream 中的 codec 中的 codec_id
       st->codec->codec_id = CODEC_ID_AMV;
        //跳过一些自己的解析
       avio_skip(pb, size);
       break;
}
 //通过 AVIOContext 获取信息设置 AVStream
tag1 = ff_get_bmp_header(pb, st);

 //判断是否为字幕流
if (tag1 == MKTAG('D', 'X', 'S', 'B')
              || tag1 == MKTAG('D', 'X', 'S', 'A'))
{
        //设置 AVStream 中的 codec 中的 codec_type
       st->codec->codec_type = AVMEDIA_TYPE_SUBTITLE;
        //设置 AVStream 中的 codec 中的 codec_tag
       st->codec->codec_tag = tag1;
        //设置 AVStream 中的 codec 中的 codec_id
       st->codec->codec_id = CODEC_ID_XSUB;
       break;
}

if(size > 10 * 4 && size < (1 << 30))
{
        //设置 AVStream 中的 codec 中的 extradata_size
       st->codec->extradata_size = size - 10 * 4;
        //设置 AVStream 中的 codec 中的 extradata
       st->codec->extradata =
              av_malloc(st->codec->extradata_size
                            + FF_INPUT_BUFFER_PADDING_SIZE);
       if (!st->codec->extradata)
       {
              st->codec->extradata_size = 0;
              return AVERROR(ENOMEM);
       }
       avio_read(pb, st->codec->extradata, st->codec->extradata_size);
}

if(st->codec->extradata_size & 1)
       //FIXME check if the encoder really did this correctly
{
       avio_r8(pb);
}
/* Extract palette from extradata if bpp <= 8. */
```

```
                /* This code assumes that extradata contains only palette. */
                /* This is true for all paletted codecs implemented in FFmpeg. */
                if (st->codec->extradata_size && (st->codec->bits_per_coded_sample
<= 8))
                {
#if HAVE_BIGENDIAN
                    for (i = 0; i < FFMIN(st->codec->extradata_size,
AVPALETTE_SIZE) / 4; i++)
                    {
                        ast->pal[i] = av_bswap32(((uint32_t
*)st->codec->extradata)[i]);
                    }
#else
                    memcpy(ast->pal, st->codec->extradata,
                           FFMIN(st->codec->extradata_size, AVPALETTE_SIZE));
#endif
                    ast->has_pal = 1;
                }

                print_tag("video", tag1, 0);

                 //设置 AVStream 中的 codec 中的 codec_type
                st->codec->codec_type = AVMEDIA_TYPE_VIDEO;
                 //设置 AVStream 中的 codec 中的 codec_tag
                st->codec->codec_tag = tag1;
                 //设置 AVStream 中的 codec 中的 codec_id
                st->codec->codec_id = ff_codec_get_id(ff_codec_bmp_tags, tag1);
                 //设置 AVStream 中的 need_parsing
                st->need_parsing = AVSTREAM_PARSE_HEADERS;
                // This is needed to get the
                //pict type which is necessary for generating correct pts.
                // Support "Resolution 1:1" for Avid AVI Codec
                if(tag1 == MKTAG('A', 'V', 'R', 'n') &&
                        st->codec->extradata_size >= 31 &&
                        !memcmp(&st->codec->extradata[28], "1:1", 3))
                 {
                     //设置 AVStream 中的 codec 中的 codec_id
                    st->codec->codec_id = CODEC_ID_RAWVIDEO;
                 }
                if(st->codec->codec_tag == 0
                        && st->codec->height > 0
                        && st->codec->extradata_size < 1U << 30)
                {
                     //设置 AVStream 中的 codec 中的 extradata_size
```

```
                                        st->codec->extradata_size += 9;
                                        st->codec->extradata =
                                              av_realloc(st->codec->extradata,
                                            st->codec->extradata_size                                  +
FF_INPUT_BUFFER_PADDING_SIZE);
                                          //设置 AVStream 中的 codec 中的 extradata
                                        if(st->codec->extradata)
                                          {
                                            memcpy(st->codec->extradata
                                                    + st->codec->extradata_size - 9, "BottomUp", 9);
                                          }
                                   }
                                    //设置 AVStream 中的 codec 中的 height
                                   st->codec->height
                                   = FFABS(st->codec->height);

                                   //                      avio_skip(pb, size - 5 * 4);
                                   break;
                              case AVMEDIA_TYPE_AUDIO:
                                    //通过 AVIOContext 设置 AVStream 中的 codec
                                   ret = ff_get_wav_header(pb, st->codec, size);
                                   if (ret < 0)
                                    {
                                        return ret;
                                    }
                                    //设置 AVIStream 中的 dshow_block_align
                                   ast->dshow_block_align = st->codec->block_align;
                                   if(ast->sample_size
                                            && st->codec->block_align
                                            && ast->sample_size != st->codec->block_align)
                                   {
                                        av_log(s, AV_LOG_WARNING,
                                                "sample size (%d) != block alig
                                                n (%d)\n", ast->sample_size,
                                                st->codec->block_align);
                                          //设置 AVIStream 中的 sample_size
                                        ast->sample_size = st->codec->block_align;
                                   }
                                   if (size & 1) /* 2-aligned (fix for Stargate SG-1 - 3x18 - Shades of
Grey.avi) */
                                    {
                                        avio_skip(pb, 1);
                                    }
                                   /* Force parsing as several audio frames can be in
```

```
             * one packet and timestamps refer to packet start. */
             //设置 AVStream 中的 need_parsing
            st->need_parsing = AVSTREAM_PARSE_TIMESTAMPS;
            /* ADTS header is in extradata, AAC without header must be
             * stored as exact frames. Parser not needed and it will
             * fail. */
            if (st->codec->codec_id
                    == CODEC_ID_AAC && st->codec->extradata_size)
             {
                 //设置 AVStream 中的 need_parsing
                st->need_parsing = AVSTREAM_PARSE_NONE;
             }
            /* AVI files with Xan DPCM audio (wrongly) declare PCM
             * audio in the header but have Axan as stream_code_tag. */
            if (st->codec->stream_codec_tag == AV_RL32("Axan"))
            {
                 //设置 AVStream 中的 codec 中的 codec_id
                st->codec->codec_id    = CODEC_ID_XAN_DPCM;
                 //设置 AVStream 中的 codec 中的 codec_tag
                st->codec->codec_tag = 0;
            }
            if (amv_file_format)
            {
                 //设置 AVStream 中的 codec 中的 codec_id
                st->codec->codec_id    = CODEC_ID_ADPCM_IMA_AMV;
                 //设置 AVIStream 中的 codec 中的 codec_id
                ast->dshow_block_align = 0;
            }
            break;
        case AVMEDIA_TYPE_SUBTITLE:
             //设置 AVStream 中的 codec 中的 codec_type
            st->codec->codec_type = AVMEDIA_TYPE_SUBTITLE;
             //设置 AVStream 中的 request_probe
            st->request_probe = 1;
            break;
        default:
             //设置 AVStream 中的 codec 中的 codec_type
            st->codec->codec_type = AVMEDIA_TYPE_DATA;
             //设置 AVStream 中的 codec 中的 codec_id
            st->codec->codec_id = CODEC_ID_NONE;
             //设置 AVStream 中的 codec 中的 codec_tag
            st->codec->codec_tag = 0;
            avio_skip(pb, size);
            break;
            }
        }
        break;
    case MKTAG('i', 'n', 'd', 'x'):
        i = avio_tell(pb);
        if(pb->seekable && !(s->flags & AVFMT_FLAG_IGNIDX))
        {
            read_braindead_odml_indx(s, 0);
        }
        avio_seek(pb, i + size, SEEK_SET);
        break;
    case MKTAG('v', 'p', 'r', 'p'):
        if(stream_index <
                (unsigned)s->nb_streams && size > 9 * 4)
        {
            AVRational active, active_aspect;

            st = s->streams[stream_index];
            avio_rl32(pb);
            avio_rl32(pb);
            avio_rl32(pb);
            avio_rl32(pb);
            avio_rl32(pb);

            active_aspect.den = avio_rl16(pb);
            active_aspect.num = avio_rl16(pb);
            active.num              = avio_rl32(pb);
            active.den              = avio_rl32(pb);
            avio_rl32(pb); //nbFieldsPerFrame

            if(active_aspect.num
                    && active_aspect.den
                    && active.num && active.den)
            {
                st->sample_aspect_ratio
                = av_div_q(active_aspect, active);
                //                          av_log(s, AV_LOG_ERROR,
                //                          "vprp %d/%d %d/%d\n",
                //                          active_aspect.num,
                //                          active_aspect.den,
                //                          active.num, active.den                    }
                size -= 9 * 4;
            }
            avio_skip(pb, size);
```

```
                break;
            case MKTAG('s', 't', 'r', 'n'):
                if(s->nb_streams)
                {
                    avi_read_tag(s, s->streams[s->nb_streams-1], tag, size);
                    break;
                }
            default:
                if(size > 1000000)
                {
                    av_log(s, AV_LOG_ERROR,
                            "Something went wrong
                            during header parsing, "
                            "I will ignore it and
                            try to continue anyway.\n");
                    avi->movi_list = avio_tell(pb) - 4;
                    avi->movi_end    = avio_size(pb);
                    goto end_of_header;
                }
                /* skip tag */
                size += (size & 1);
                avio_skip(pb, size);
                break;
            }
        }
end_of_header:
        /* check stream number */
        if (stream_index != s->nb_streams - 1)
        {
fail:
            return -1;
        }

        if(!avi->index_loaded && pb->seekable)
        {
            avi_load_index(s);
        }
        //对 AVIContext *avi 设置 index_loaded
        avi->index_loaded = 1;
        //对 AVIContext *avi 设置 non_interleaved
        avi->non_interleaved |= guess_ni_flag(s);
        for(i = 0; i < s->nb_streams; i++)
        {
            AVStream *st = s->streams[i];
            if(st->nb_index_entries)
             {
                break;
             }
        }
        if(i == s->nb_streams && avi->non_interleaved)
        {
            av_log(s, AV_LOG_WARNING,
                     "non-interleaved AVI without
                     index, switching to interleaved\n");
             //对 AVIContext *avi 设置 non_interleaved
            avi->non_interleaved = 0;
        }

        if(avi->non_interleaved)
        {
            av_log(s, AV_LOG_INFO,
                     "non-interleaved AVI\n");
            clean_index(s);
        }

        //对 AVFormatContext 设置各个媒体流的 metadata
        ff_metadata_conv_ctx(s,
                                  NULL, ff_avi_metadata_conv);

        return 0;
     }
```

这个函数的代码的确够长的，但是里面所有涉及到设置我们所关注的对象的成员值的地方，都添加了注释，所以不会再进一步去仔细分析。

经过函数 av_open_input_stream，我们就算完成了 av_open_input_file 函数的功能，这样就完成了通过 AVI 文件设置 AVFormatContext 的步骤。

其四：设置 AVFormatContext 的 programid;
在这步中，主要就是设置 AVFormatContext 内部的媒体流的(AVStream)-(discard)的值，如果与设置变量中的 programid 不相等，则(AVFormatContext)-((AVStream)-(discard) = AVDISCARD_ALL，否则则有
(AVFormatContext)-((AVStream)-(discard) = AVDISCARD_DEFAULT，
这样的设置规律也是应用在(AVFormatContext)-(AVProgram **programs)身上。

其五：设置 AVFormatContext 的 loop_input;
直接依据设置变量中的 loop_input 设置即可，至于有什么具体的作用，目前还没有看出来。

其六：重要的设置函数 av_find_stream_info;
对于 AVFormatContext，函数 av_find_stream_info 的作用就是为了获取 AVFormatContext 中

媒体流 AVStream *streams[MAX_STREAMS]更多的参数。av_find_stream_info 函数是一个很复杂的函数，它的具体解析将放在前面的 libavformat\utils.c 源代码中。对于 av_find_stream_info 函数的分析，让人头疼啊，真他妈的狗屎代码啊，但是这些代码还是正确的在运行着，只能让笔者产生一种强烈的改写该代码的冲动啊！我说主啊，你是不是不想让我阅读这样的代码啊。

后面也许还有很多没有分析到代码，但是由于本人的任务安排，必须放下目前文档的撰写，去实质性的解决问题。

2.5.2 SDP 文件的转码输入流程解析

其一：SDP 输入，对应协议“file”，通过读取文件内容，得到文件类型，“SDP”